**Created:June 13, 1994**
**Modified:March 18, 1996**

**Gemini Controls Group Report**

# The OCS PDR Review

**Steve Wampler**

REV-C-G0053/01

**This is the final report on the Gemini Observatory Control System System Design Review held September 6, 1995, in Edinburgh Scotland.**

## 1.0 Introduction

The OCS Preliminary Design Review was held September 6, 1995 at ROE in Edinburgh Scotland. This report collects together the documenation from the review, including:

- The report of the review committee
- The responses of the OCS design team to comments and concerns raised as part of the review
- Updated copies of all documents that were reviewed during this process.

Not included are copies of the supplemental documentation that was available during the review.

**Created:June 13, 1994**
**Modified:March 18, 1996**

**Gemini Controls Group Report**

# The OCS PDR Review

**Steve Wampler**

gscg.sbw.072/02

**This report summarizes the results of the Observatory Control System Preliminary Design Review held September 6, 1995, in Edinburgh Scotland.**

## 1.0 Introduction

The OCS Preliminary Design Review was held September 6, 1995 are ROE in Edinburgh Scotland. This report describes the results of the review and all action items that are a result of the review. A companion report,[16], gives detailed comments that were raised as part of the review and the responses of the OCS development team.

## 2.0 References

[1] ocs._sw.007-pdrProducts, OCS Preliminary Design Review Product Overview, Gemini Observatory Control System Group, 1995.

[2] ocs.kkg.014, *Observatory Control System Software Requirements Document*, 6/5/95, Gemini Observatory Control System Group, 1995.

[3] ocs._sw.004, *Observing Tool Track Preliminary Design*

[4] ocs._sw.006, *Preliminary Design for Configurable Control System Concurrency*

[5] ocs.kkg.031, *Preliminary Sequence Command Design*

[6] ocs.kkg.032, *OCS Interactive Infrastructure Track PDR*

[7] ocs.kkg.033, *Telescope Control Console Track Preliminary Design*

[8] ocs.kkg.034, *User/Observing Console Track Preliminary Design*

[9] ocs.kkg.035, *Observatory Control System Development Plan*

[10] ocs.kkg.036, *Preliminary Design for Access Control in the OCS*

[11] ocs.kkg.037, *User Activity Models in OCS*

[12] ocs.kkg.038, *Planned Observing Support Track Preliminary Design* (VERIFY)

[13] ocs.ocs.001, *Observatory Control System PDR Data Dictionary*

[14] ocs.ocs.002, *OCS Physical Model Description*
[15] ocs.ocs.003, *Scheduling Track Preliminary Design*
[16] ocs.ocs.007-pdrResponse, Response to the Comments on the OCS PDR Documents

## 3.0 Reviewers

The reviewers were split into several groups:

- 'virtual' reviewers who were asked to review the materials and provide comments via email for discussion during the review, and
- 'real' reviewers who were asked to be present at ROE for the review meeting

We would like to thank all these people for their support and comments.

**TABLE 1.** Reviewers present at OCS PDR review

| Name | Organization |
|---|---|
| Steve Beard | ROE |
| Severin Gaudet | DAO |
| Chris Mayer | RGO |
| Jim Oschmann | Gemini |
| Rick McGonegal | Gemini |
| Malcolm Stewart | ROE |
| Pat Wallace | DRAL |
| Steve Wampler | Gemini (chair) |

**TABLE 2.** 'Virtual' reviewers

| Name | Organization |
|---|---|
| Peter Biereichel | ESO VLT |
| Fred Gillett | Gemini |
| Keith Shortridge | AAO |
| Doug Simons | Gemini |
| Susan Wieland | Gemini |

## 4.0 Overall review comments:

The review committee felt that the OCS was successful in meeting the requirements of the PDR. In fact, the committee believes the OCS team has done an outstanding job in

preparing the OCS preliminary design. There has been an impressive amount of effort put into the OCS design.

The committee believes that the project would be best served by replacing further OCS reviews with the prototyping approach proposed by the OCS team in conjunction with more frequent meetings between the Principal Systems developers.

The documentation was lacking a definitive overview. The committee asks the OCS to develop a complete design overview, preferably including a graphic diagram, as the work progresses. This overview should be separated from the detailed design.

## 5.0 Concerns

The biggest concerns of the committee centered on scheduling and infrastructure support.

### 5.1 Concern over schedule

The committee is concerned that the OCS is on an ambitious time line and that there is too much to do in the time allocated with the number of personnel available.

### 5.2 Concern over infrastructure

There is still concern over the functionality provided by CAD/CAR records, but the committee expects these concerns to be addressed during the next phase of the design as the Interactive Observing Infrastructure track gets underway.

The committee urges the Gemini Project to make the next release of the CAD/CAR record implementation available as soon as possible, and encourages the OCS development team to remain flexible as other work packages get underway and impact the overall system design.

The committee further urges the OCS to concentrate on the IOI infrastructure and avoid the temptation to devote too much time to the upper-levels of the design until the point in the WBS where such attention is scheduled.

## 6.0 Review Comments and Responses

Because of the success of the review, the comments of the reviewers and the responses of the OCS development team are listed in [16] and not repeated here. Readers should refer to that document for details of the issues raised during the review.

## 7.0 Action Item Status

A number of action items have resulted from the OCS PDR review. These action items are listed in this section.

**TABLE 3.** OCS PDR Action Items

| Item | Description | Who? | Status? |
|---|---|---|---|
| 1 | The OCS team should illustrate how the infrastructure works by developing a few observing screens and implement their functionality down through the IOC level. | OCS | |
| 2 | An OOD (Object-Oriented Design) tool should be chosen soon, and the choice should be discussed with Severin Gaudet (DHS). | OCS | |
| 3 | The funtional requirements for the external database need to be developed. | OCS and DHS | |
| 4 | The OCS needs to pick a message system service for intraOCS communications. | OCS | |
| 5 | The CAR state diagram should include a transition from PAUSED to ERROR on error status input. | OCS | |
| 6 | Resolve remaining CAD/CAR issues. | All principal systems | done |
| 7 | Address issue of timeouts during long actions - specifically, how does the OCS detect that a system as ‘hung’ while busy and will never complete the action. | OCS | |
| 8 | Describe choices for a scripting language to gemini-software for discussion. | OCS | |
| 9 | Resolve OBSERVE/ENDOBSERVE behavior and how date flows from instruments to the DHS during these actions. | All principal systems | in progress |
| 10 | Discuss how concurrent observations impact on the DHS, which must handle data from several sources at the same time. | OCS and DHS | |
| 11 | Fix epoch/equinox use in example screens in documentation. | OCS | |
| 12 | Examine session manager and OT interface for common functionality to see if they can share common approaches. | OCS | |
| 13 | Discuss use of various devices for target acquisition and how the principal systems are impacted by need to save acquisition images. | GPO and DHS | |
| 14 | Add COB to list of instruments (commissioning device) | OCS/CICS | |
| 15 | Look at role of PV-Wave in the OCS and DHS | OCS and DHS | |
| 16 | Look at data access control procedures. | OCS and DHS | |
| 17 | Ask scientists about data access control requirements to see what the expectations are. | GPO | |
| 18 | Point out questionable requirements to GPO. | OCS | |

Created:May 2, 1995
Modified:April 29, 1996

Gemini Observatory Control System Report

# *Response to the Comments on the OCS PDR Documents*

**Shane Walker, Kim Gillies**

ocs.ocs.007-pdrResponse/03

**This report contains actions and responses resulting from the comments made by OCS PDR electronic reviewers.**

## 1.0 Introduction

The Gemini OCS Preliminary Design Review was held on September 6, 1995 at the Royal Observatory, Edinburgh. Comments were solicited and received electronically between August 16, and September 6, 1995. Participants at the review meeting included programmers Chris Mayer (RGO), Patrick Wallace (RGO), Steven Beard (ROE), and Severin Gaudet (DAO), and project managers Rick McGonegall (Gemini Project) and Malcolm Stewart (ROE). Reviewers who did attend the meeting include Keith Shortridge (AAO) and project scientists Fred Gillett (Gemini Project) and Doug Simons (Gemini Project). Others at each site also reviewed the documents and their input is also included in each participants comments. In conjunction with the OCS PDR meeting, a series of Principal Systems meetings were held on September 4, 5, and 8. There was much overlap in the topics covered.

The following paper examines the comments that were submitted electronically and gives a response or an action to each, if necessary. Many of the topics were covered in detail in the meetings, and the decisions made there are folded into the responses.

The corrections to the documents pointed out by reviewers are not mentioned here but those corrections will be made to the documents. The comments of the reviewers are organized by reviewer and then by document.

## 2.0 References

[1] *Journal of Object-Oriented Programming*, May 1995. James Rumbaugh, Modeling and Design column.

# 3.0 Chris Mayer and Patrick Wallace's Remarks

Chris and Patrick divided their comments between general remarks and comments on specific documents.

## 3.1 General Remarks

### 3.1.1 Complexity of the Design

Having read the documentation we are concerned about the amount and complexity of the work given the tight time constraints. The work package appears very ambitious. It looks as if the OCS developers are being set an impossible task. Consideration should be given to lengthening the time scales and increasing the allocation of manpower to this work package.

***Response.*** *This concern was expressed by others as well. Previous reviews have examined the complexity of the design and have found it to be a reasonable match to the complexity of the job of the OCS.*

*The Gemini Project Office is aware of the perception that the OCS complexity will be hard to manage. It is our job to insure that we monitor the development and keep the project office informed of our progress. Manpower issues will be addressed by the project office.*

### 3.1.2 Aliases

One aspect of the documentation that didn't help here was the generation of "aliases" for terms that had already been used. The most obvious example of this was the introduction of "action variable" for CAR record.

***Response.*** *The term "action variable" is more general than CAR record. A CAR record represents a particular kind of action variable used in conjunction with actions that are initiated and monitored through the EPICS CAD/CAR interface. However, OCSApps will have to be able to monitor actions in other OCSApps as well, and this functionality will not be supported using CAR records. For this reason, we adopted the more general term "action variable." We regret that the distinction was not made clearer in the documentation.*

## 3.2 General Design Comments

Turning to more specific topics, the aspects of the design that concern us most are (not in order of priority)

### 3.2.1 Concurrency within a session

The intention to support concurrency within a session if the resources for the different observations don't conflict. If concurrency was only allowed for different sessions then this would simplify one part of the software without removing the ability to execute observations concurrently.

***Response.*** *This point was discussed at the review meeting. We agree that the design would be simplified by disallowing concurrency within a session. However, it has been deemed essential that an observation should be able to release one or more of the resources it holds before it has completed execution in the system so that other observations can proceed. The most obvious example of the need for this is to support slewing the telescope to the next target while the detector for the previous observation is still reading out.*

### 3.2.2 Introduction of cl_data in CAD/CAR design

The concentration of the design on handling conflicting requests from different operators i.e. the introduction of the client_id data. Is this really necessary? It complicates the PS and the implementation of the CAR records considerably in order to handle a situation that will very rarely occur. If users want to issue conflicting commands from different OTs then it is up to them to keep track of what they are doing

***Response.*** *First, we do not believe that adding cl_data is a considerable complication, though neither of us are as familiar with EPICS as the TCS designers are at this point. A PS simply passes the cl_data along when it updates the CAR record for the action. The CAR record does nothing with the client data except reflect it in its STATUS output.*

*It was pointed out in the Principal Systems meeting that the EPICS dm tool will be awkward to use when a cl_data field is required for each CAD record. Also, if the TCS to TCS subsystem interface is to use the same CAD/CAR records, then cl_data will only complicate matters. However, we believe that the Principal System interface should not be constrained by either the shortcomings of the dm tool, or by the needs of internal system interfaces, which are not the intended use of CAD/CAR/SIR.*

*The cl_data field is needed in the OCS to associate a command with the actions is causes in a PS. It makes it possible to unambiguously determine when an action completes. This capability is required in the OCS but may not be required between other systems and their subsystems. Even if we decide that we don't care about simultaneous use of the same commands by different operators, the design of the OCS is tremendously simplified if cl_data is introduced.*

### 3.2.3 Changes to the PS interface

- The desire to layer a full message system on top of EPICS. If it is really required to know the origin of every command then perhaps we should look at an architecture written to support this model rather than trying to bend EPICS to fit.

***Response.*** *It isn't clear that the changes we propose constitute a "full message system." At any rate, your point is still valid. It seems that we are using EPICS as nothing more than a message passing system at the PS interface level. On the other hand, we are confident that the CAD/CAR design will work and if we remove EPICS then it has to be replaced with something else, and a new ICD design developed.*

- The requirement to support spontaneous monitors i.e. to raise more than one monitor per processing of a CAR record. This may be possible but we don't believe it is currently supported by any other records and questions of EPICS buffering and lock sets will have to be resolved. This requirement would go away if we dropped the need for cl_data and used other records to show progress of an action i.e. drop ALERT

***Response.*** *ALERT was added to parallel the functionality of the DRAMA TRIGGER message. It isn't required in the OCS, but it does seem like a nice way of providing feedback, especially if there are no SIR records associated with the action or if the SIR record values don't change quickly. However, the MODIFIED state (which must be implemented in the same manner as ALERT) is important. It allows an OCSApp to determine when an action it initiated has been interrupted before completing.*

***Action.*** *The issue of buffering and lock sets needs to be examined, perhaps by Bret Goodrich when he gets up to speed with EPICS. We will bring these issues up with the EPICS experts that will be reviewing the CAD/CAR records.*

- The mention of setting the CAR record to BUSY from within a subroutine attached to a CAD. This has all sorts of implications for the lock sets that records are allowed to be placed in.

***Action.*** *It was agreed in the Principal Systems meeting that this requirement is no longer needed, provided that cl_data is used, and that CAR record updates eventually occur in the correct order (the order CADs are applied). The OCS will not rely on the PS setting a CAR record to BUSY before accepting a command.*

### 3.2.4 Sequencing

The unresolved question of what sequencing is done by the OCS and what is done by the PS. The design appears to be evolving towards the approach of the PS receiving their commands in any order. In this case commands need to be of the preset type. However, there doesn't seem to be any command to tell a PS when to apply all the presets ie. when a configuration is complete and should be executed.

***Action.*** *Sequencing problems appear to have been solved in the Principal Systems meeting by adopting Patrick Wallace's approach of presetting one or more CAD records, then using an APPLY command to actually initiate the actions. New command verbs must be invented to disambiguate the overloaded meaning of the term APPLY.*

Has the model of "here's a configuration - fire it off and forget about it for 10 mins" been selected because it's what astronomers want or because it's hard to do better? For example, is the system going to be fast enough to do automatic peaking-up, or will such things just have to bypass the OCS, typically by simply using engineering features operationally?

***Response.*** *It is difficult to interpret the meaning of this statement. I don't believe astronomers care how we handle the PS interface. They're only interested in what the system does, not how its implemented. Also, we don't recall discussing a "fire it off and forget about it" model. The approach we settled upon at the meeting will satisfy our system requirements. For common actions like peaking-up, the OCS must provide adequate support. If there is a frequent need to bypass the OCS then we have failed to do our job.*

#### 3.2.5 Separate IOCs

The records for the SAD and ARD should be supplied by the PS in their own IOC's. If the OCS needs them in a separate IOC then copies should be created and the ones in the PS monitored. If this is not done, standalone operation of the PS will be difficult.

***Response.*** *Separate IOC copies should not be needed. The SAD and ARD are comprised of the distributed EPICS databases. The actual location of the various SIR and CAR records is immaterial, since EPICS hides this detail. It was agreed at the meeting that IOC developers will structure their databases so they can be moved from their IOCs to the SAD IOC if it is determined that is necessary by the operations staff.*

#### 3.2.6 Use of available packages

At several places in the OCS documentation the intention is stated to use available packages or products but that a package has not yet been identified. Is there any contingency in the work package to allow for no suitable package being found so that extra work has to be done to extend what is available?

***Response.*** *Using commercial and "netware" whenever necessary is a requirement of the OCS, and we must take advantage of using the work of other programmers whenever possible. Luckily, we are confident that products can be found to satisfy many of the OCS needs (such as a message system, and database). If these tools could not be found we will have to revisit the WBS, prioritize, and make any required adjustments.*

### 3.3 Work Breakdown Schedule

We have not printed the detailed work breakdown at this stage but the docs. indicate an awful lot of work. Can it be done? In particular need to see if there is any slack in system and the three concurrent tracks have more work than the duration! With only 3 staff (?) this can't be done. Need more details of staff etc. to resolve this.

***Response.*** *See "Complexity of the Design" on page 2.*

### 3.4 OCS Physical Model Description

#### 3.4.1 Section 1.7

Does this mean a tool will not be used nor the Ward/Mellor methodology? Also implies that other structured design tools may only be of limited use. What is Gemini viewpoint on this? or is this the Gemini viewpoint? We hope that the search for object-oriented tools doesn't hold up implementation, and that if the OCS group find any other people will be able to comprehend the output.

***Response.*** *An object-oriented design methodology was chosen for the internal design of the OCS for the reasons indicated in section 1.1. The search for appropriate tools is underway and we have identified at least one such tool that would be adequate. The search will not hold up implementation beyond the few days that it takes to evaluate a couple of trial copies (we are building upon a previous investigation of CASE tools by NOAO's Richard Wolf). If we continue the implementation without tool support however, we believe that the design would suffer since changes would be difficult to document and would tend to be reflected only in the implementation.*

*As for comprehending the output, we are confident in our own abilities to master the OO design methodologies. Others may in fact have to familiarize themselves with our chosen methodology before attempting to comprehend the design. Object-oriented software design should not be treated any differently than any other discipline. One would not expect to immediately grasp an electrical engineer's design with no prior introduction to his field's notation.*

### 3.4.2 Section 1.7.1

What happens if no tool comes along, does this mean OCS will not produce data flow diagrams, state diagrams etc.

***Response.*** *See above.*

### 3.4.3 Section 1.8

What does "opaquely" mean in point 3

***Response.*** *Point 3 is getting at the idea that once steps 1 and 2 are complete, we no longer have to focus on the details of each OCS application. Instead, we focus on interactions between applications to structure the OCS.*

### 3.4.4 Section 2.1

Consoles - does this imply many OCS clients connecting to a PS? or is the PSA the only client that connects in a CA sense?

***Response.*** *In the PDR design, only the PSA for a PS actually issues commands. However, consoles and other clients monitor the ARD and SAD using Channel Access.*

### 3.4.5 Section 3.2

What would be impact of 1) having to write it all from scratch 2) extending a package if it doesn't do everything that is required?

i.e. is there any contingency if a package can't be found?

***Response.*** *See above.*

### 3.4.6 Fig 3-2

What do dashed lines indicate on an object diagram (they don't appear on fig 1-1)

***Response.*** *Unfortunately Figure 1-1 is incomplete. It is difficult to reduce an entire book's worth of methodology to one figure illustrating the major notation. Perhaps it is deceptive to even attempt to introduce the methodology? The dashed lines are used whenever an association between two classes exists but cannot be shown directly. The SystemSubjectCommunications subsystem is not expanded to show which classes it contains, so the direct association between the class outside of the subsystem and the class inside of the subsystem cannot be shown. Dashed lines are used in this instance.*

### 3.4.7 Fig 3-3

Why do you use EpicsService rather than CADService?

***Action.*** *This is an error and will be changed to CADService.*

#### 3.4.8 Fig 4-2

Not clear why there are only two controllers here. Does setting anything in the view then need the accept button to be pressed. The implication of the text is that setting track on will cause it to happen. If this is the case what is accept for?

***Response.*** *Only two controllers are shown because the purpose of this section was just to demonstrate how the model works—two examples were felt to be sufficient.*

*User interface details will be worked out in the Telescope Control Console track. Some widgets, such as text entry fields should be activated by a separate "Accept" button. This gives the user a chance to make all the edits he wishes before applying the changes. Other widgets, such as simple toggle buttons, may immediately activate in some cases. This avoids the hassle of having to press two buttons to achieve one result.*

#### 3.4.9 Fig 4-4

Not sure that ca_put is the correct terminology to use when a client has registered a monitor.

***Response.*** *Will correct.*

#### 3.4.10 p4-6

This states that apply must be pressed when the target bag pressure is entered. So why aren't zenith set point and altitude slope also considered as controllers?

***Response.*** *See the comment above about using just two controllers to demonstrate system.*

#### 3.4.11 Chapter 5

Seems at variance with the rest of the document. Very difficult to see what is being added here.

***Response.*** *This chapter re-emphasizes the point that the OCS is structured as interacting OCSApps. It is not a long chapter because the other PDR papers contain numerous instances. In particular, the last sentence refers the reader to the Planned Observing Support track paper for more examples.*

### 3.5 Interactive Observing Infrastructure

#### 3.5.1 Section 1.7.1

Need to justify the opcodes that go with the CAD command. In particular, is there any standard behaviour expected of abort etc. E.g what is expected if you abort the offset command, should it do nothing if the action is complete, zero out the current offset and go back to where it was, zero out the accumulated offset? What happened to the Verify opcode?

***Response.*** *In the Principal Systems meeting it was decided that the PS designers must support all the directives, but it is up to them to interpret their meaning, within some general guidelines. For instance, an abort is generally taken to mean a very urgent, non-graceful stop. If the PS designer doesn't feel that there is any difference between abort and stop for a particular CAD, then he will use the same code for each command.*

*The opcodes PAUSE and CONTINUE have been removed, since it was considered that pauseable actions would be the exception rather than the rule. If an action can be paused, then a special pause CAD will have to be created to access the pause functionality.*

#### 3.5.2 Section 1.7.2.1

"The parts are then sent as the arguments to separate sequence commands to the appropriate PSAs" - will there be any constraints on the order they will be sent. If not then the TCS CAD/CAR interface will have to consist entirely of records that preset everything and the PSA will then have to send something to say that the configuration is over so that the TCS will know to act on it. This seems to be back to sending an apply sequence command.

***Response.*** *See "Sequencing" on page 3.*

### 3.5.3 Section 1.7.2.2

This seems to indicate there will only be one OCS client connected to each principal system? (see comments on Physical Model Description section 2.1)

***Response.*** *See "Section 2.1" on page 5.*

Does para 1 on page 1-6 mean that CAR records have to be placed in a separate IOC. If so then we lose 1) ability to have links on a CapFast diagram 2) standalone operation. If the OCS requires an Action Response Database then it should monitor the principal system CAR records and copy the values into its own local copies.

***Response.*** *See "Separate IOCs" on page 4.*

### 3.5.4 Section 1.7.3

If the identity is to be copied into the CAR record then new fields will be needed in the CAD record as well. This seems to be contradicted by the first line of p1-7 that says "the ps has no knowledge of the sender of the command"

***Response.*** *This point seems to saying that because the cl_data field is copied by PS into the CAR record, the PS has to know the originator of the command. In fact, the cl_data field should have no meaning to the PS, and the PS should not rely upon the information in the field being in a particular format. The job of the PS is to copy the string when it writes busy to the appropriate CAR record.*

### 3.5.5 Figure 1-2

Why do action/responses not go via the PSA? In the diagram shown, the command layer needs to know the CAR record names but I thought this was a function of the PSA. Or, is it only the mapping of attributes to fields of the CAD records that the PSA handles?

***Response.*** *Action responses could go via the PSA under a different design. The trade-offs for OCSApp-PS communication structure are covered in the IOI Track paper "IOI Track Design Decisions" chapter. You are correct in noting that the CLL will have to use the mapping of CAD records to CAR records that should be monitored. The current plan is to store this information in the Observing Database, and to cache the entries in the OCSApps so that they need be retrieved only once.*

### 3.5.6 Page 1-7, 2nd paragraph

"The PSA monitors the ARD" seems to directly contradict the last sentence of section 1.7.3 "The PSA does not monitor CAR records"

***Action.*** *Yes, that is a direct contradiction. The "PSA monitors the ARD" paragraph was unfortunately a leftover portion of a previous design iteration and will be removed. Section 1.7.3 is correct.*

### 3.5.7 Section 2.5.4, point 1

What does "when all requests when the one action is completed" mean?

***Response.*** *Unfortunately that sentence was mangled. The situation being referred to is a case in which the client issues multiple action requests, say several offset commands, and then wishes to wait until all have completed. Rather than waiting on each individual request, the CLL only needs to wait for the last to complete.*

How does a principal system know when all parts of a configuration have been received?

***Response.*** *See "Sequencing" on page 3.*

### 3.5.8 Section 2.6.1, R28

What is meant by a subscription based status capability?

***Response.*** *By subscription based, we mean that the OCS message system must allow Unix-based applications to acquire and publish status information in the same way EPICS IOCs publish status information (the monitor ability). OCSApps will register an interest in particular status variables and will be notified of changes whenever the owner of the status item updates it. The CLL must provide the capability to express this interest, and to notify the client when the values change.*

#### 3.5.9 Section 3.2, last paragraph

As mentioned before, the fact that the PSA does not monitor CAR records means the CCL must also be able to map attribute value pairs to CAD records or else it won't know which CAR records to monitor.

***Response.*** *Yes. See "Figure 1-2" on page 7.*

#### 3.5.10 Page 3-4, PSAR9

Again, what happened to VALIDATE?

***Response.*** *VALIDATE was removed at a previous OCS review because it was believed to be extraneous and to cause too much extra work. Validation is only accurate at the time that the checks are performed and may not be accurate when the actual CAD is applied. In addition, a validation is needed when a CAD record is applied before the actions can be performed anyway.*

*In the last Principal Systems meeting, the new "preset" design was introduced (see "Sequencing" on page 3). The VALIDATE directive may again be useful in this design, but under altered semantics as a preset. This is TBD.*

#### 3.5.11 Section 3.4, 1st bullet

We'd like to keep the CAR records for action signalling and the CAD VAL and MESS fields for command signalling. E.g a successful source command has been given and VAL is set to 0 and MESS to OK whilst the corresponding CAR goes to BUSY. Now the observer changes their mind and sends another SOURCE command but gets it wrong e.g. types in an hours field greater then 23. If the CAR record is the only way of signalling this as an ERROR then the CLL will see the action response of the previous valid command go from BUSY to ERROR.

***Response.*** *This is a good point and ultimately a major reason why the accept/reject protocol was retained in the Principal Systems meeting.*

#### 3.5.12 Section 3.5

If the last paragraph is true then the CLL will also need to be able to map configuration part names onto PS CAD names or else it won't know which CAR record to monitor. (see comments on section 3.2)

***Response.*** *See "Figure 1-2" on page 7.*

#### 3.5.13 Section 4.2

A general point here. Why introduce aliases for terms that already exist. e.g. action variable = CAR record. This makes it hard to keep track of what is being discussed. (see last bullet of 4.4.3.1 also)

***Response.*** *See "Aliases" on page 2.*

Paragraph 2. It may be better to let all CAD records have a corresponding CAR. If you don't then something has to keep track of which do and which don't and inform the CLL that it shouldn't look for a CAR response. Is this the function of an immediate completion response from a CAD?

***Response.*** *Immediate completion was rejected at the Principal Systems meeting because it was believed to be an unnecessary feature.*

3rd paragraph. This idea might be valid for actions that take place very infrequently but can't be applied for internal actions that place continuously. For example once a source command has completed and the telescope is tracking the rotator, a&g probes, secondary mirror, primary etc. will all be continuously

adjusted. Is it necessary for all of these to somehow show busy and if so through which TCS CAR record? At the moment can't think of any actions that will be self-initiated.

***Response.*** *Whether there are any TCS self-initiated actions or not, the design remains the same. It will support self-initiated actions should any arise. If the TCS (and other systems) do take actions on their own and if the operator's effectiveness might be enhanced by knowing those operations are underway, it would be good if the IOCs present that in the CAR records as suggested rather than hiding them.*

### 3.5.14 Section 4.4

The implication is that the CAR record is special in that it raises more than one monitor when it is processed. This may not be possible with strings that are buffered. Only the last monitor would be seen.

***Response.*** *See "Changes to the PS interface" on page 3. This potential problem will be researched.*

Where does cl_data come from - presumably from the CAD record but then how does something like dm set this?

***Response.*** *Yes, the cl_data field is determined by the OCS and set in a CAD record field. See "Introduction of cl_data in CAD/CAR design" on page 2 for further comments.*

Unavailable should be a property of the CAD record not the CAR.

***Action.*** *As long as systems are going to have unavailable actions, we will retain the UNAVAILABLE state in the CAR. Alternate implementations for unavailable were discussed at the principal systems meeting and none had any real advantages over displaying UNVAVAILABLE in the CAR. At any rate, the CAD record should always be available. It is the* actions *that are caused by the CAD record's application that are or are not available.*

### 3.5.15 Section 4.4.2

I am not really sure what is being got at here and what the problem is but I am concerned about doing ca_puts from within a subroutine linked to a record. The implications of doing this for the lock sets that records are placed in and the priority of the various tasks that make up EPICS need to be considered.

Table 4-1 MODIFY, ALERTED - this idea of spontaneous monitors needs verifying

***Response.*** *For both these issues, see "Changes to the PS interface" on page 3.*

ERROR -> IDLE - can't see point of this transition

***Response.*** *We will examine this and change the documents if needed.*

### 3.5.16 Figure 4-4

"It must set CAR before returning" - why is this if the command is to be rejected? What does "returning" mean here?

***Response.*** *The subroutine no longer sets the CAR before accept/reject (see "Changes to the PS interface" on page 3). Returning means that the CAD subroutine has completed and it is known whether or not the CAD request can be executed. Accept/reject is returned to the PSA.*

### 3.5.17 Section 4.4.3.1

How will engineering screens handle this?

***Response.*** *I am assuming this is the EPICS dm tool issue. See "Introduction of cl_data in CAD/CAR design" on page 2.*

Last bullet appears to contradict 2nd bullet.

***Response.*** *The last bullet applied only to commands that cause actions. Immediate completion has since been removed, but the last bullet is still valid.*

### 3.5.18 Section 4.4.4.1

Do we really need to design our system to cope with multiple independent consoles entering commands at any time? Who and why would we be allowing this type of behaviour?

***Response.*** *See "Introduction of cl_data in CAD/CAR design" on page 2.*

### 3.5.19 Section 5.1

This shows a difference of viewpoint (from this TCS developer anyway!) The CAD/CAR appear to be tied to specific pieces of hardware. Frequently examples involve filters. But, what is the device for the SOURCE command? It appears to be the whole telescope but then the device is always busy. In most cases I believe the CAR records reflect the state of the "action" not the device. For devices like filters the two are the same but for most of the telescope they are not.

***Action.*** *The TCS developer's viewpoint is correct. The wording in this section needs to be fixed. CAR records do reflect the state of an action, not necessarily a particular device.*

The example given of the filter wheel ending up in position 4 when it was asked to go to 5 does not convince, why would the CAR record go to IDLE and not ERROR in this case?

***Response.*** *The filter wheel example was not meant to convince. It simply illustrated a type of error that the protocol does* not *handle. Indeed, the CAR record should go to ERROR, but if it does not, our protocol will not check SIR records. Instead, we will believe that a transition from BUSY to IDLE means that the action completed successfully, even if it has not.*

### 3.5.20 Section 5.2, R17

This has been brought up before. What does "before accepting" mean here?

***Response.*** *Accepting means determining that the arguments are valid and setting the appropriate response field in the CAD record. Note that R17 is no longer valid (see "Changes to the PS interface" on page 3).*

### 3.5.21 Section 5.3.1

Given this requirement it must be asked are we using the correct infrastructure. EPICS doesn't support commands so we seem to be trying to layer a command structure on top.

***Response.*** *This is handled in "Changes to the PS interface" on page 3.*

### 3.5.22 Section 6.2.1, Scenario 1

As has been brought up before points 10 & 11 may cause problems. The sequence as I see it would go

10. The PS finds the arguments acceptable and set the accept field to accept
11. Busy:client_id is written to the action's CAR record
11a. The action is started

***Response.*** *This is the modified scenario. See "Changes to the PS interface" on page 3.*

### 3.5.23 Section 6.2.1, Scenario 6

This implies the offset completion state includes the completion states of all the other mechanisms that will be affected by the offset command e.g. rotator, probes etc.

***Response.*** *That is an accurate statement.*

### 3.5.24 Section 6.2.1, Scenario 7

The TCs presumably just writes busy:client_id into the offset CAR and the internal workings of the CAR record raise the necessary monitors?

***Response.*** *Yes, the IOC only writes a few values to the CAR record when appropriate: busy:client_data, done, pause, error, and perhaps unavailable. The CAR record processing handles transitions in the Action Response Protocol, and interested parties receive the updated CAR states.*

#### 3.5.25 Section 6.2.1, Scenario 8

Should read three pushes not five pushes. This scenario is not quite the same as the filter wheel case. For example, the "position of the filter wheel is not determined" does not apply to the TCS case. When the offset is aborted the TCS will continue to track the last base target.

***Action.*** *The scenario will be updated with these points in mind.*

#### 3.5.26 Section 6.2.1, Scenario 9

My feeling is that we shouldn't complicate our system by worrying about cases like this. I agree it can cause problems but realistically how often will it happen. If observers are using multiple consoles (presumably within the same session) to give conflicting commands then they should be responsible for the consequences.

***Response.*** *Covered in "Introduction of cl_data in CAD/CAR design" on page 2.*

#### 3.5.27 Section 6.2.1, Scenario 13

I don't think the CAR record should be overloaded in this way. There should be an SIR record to handle this.

***Response.*** *The issue of ALERTED is discussed in "Changes to the PS interface" on page 3. Again, ALERTED is not needed by the OCS. For now it can be left in the design whether or not it is ever used. It adds no real complexity to the CAR record implementation.*

#### 3.5.28 Section 6.2.1, Scenario 18

This might just be a bad example but I can't see the TCS starting any actions of this sort. In the specific example mentioned here the reshaping of the primary mirror is a continuous action that will take place whenever the telescope is tracking or slewing to a new source. It is true that it will happen relatively infrequently but otherwise it is no different from the continuous updates of the az/el mount axes for example and there is no intention of setting the CAR record of say the source record to BUSY every time an update is sent to them.

***Response.*** *See "Section 4.2" on page 8.*

#### 3.5.29 Section 6.2.1, Scenario 19

This scenario can't happen. The observer will have either configured the system so that the mirror shape tracks the elevation and temperature or not. If they want to force a reconfigure then they will turn primary mirror tracking off.

***Action.*** *The scenario will be updated with this in mind. Though it may not be accurate for the TCS, it does correctly discuss how the OCS would handle a self-initiated action.*

#### 3.5.30 Section 6.2.1, Scenario 20

As for scenario 19. It is based on the assumption that the TCS works in a different way than is planned.

***Action.*** *The scenario will be updated as noted for Scenario 19.*

#### 3.5.31 Timeouts

There is no scenario to handle the case of timeouts. What happens for example if a script or something is waiting for a completion status that never happens due say to an IOC locking up. Is there a way of the OCS breaking out of this type of situation?

***Response.*** *The issues related to timeouts were discussed at the PS meeting. We will integrate the results of the discussions into the OCS documents.*

## 3.6 Planned observing support track

### 3.6.1 Section 5.2

Why is a study of multiprocessor computers needed?

***Response.*** *There are many compute intensive applications in the OCS and DHS. The use of multiprocessors can possibly simplify our system and increase performance affordably. We will do this testing as we develop the OCS.*

### 3.6.2 Section 6

The terms Science Plan and Science Program are easy to confuse. For example the sentence "a session is associated with a single science plan" in the terminology of the User activity models document would read "a session is associated with a single observing plan".

This seems to be another example of generating synonyms for the same concept.

***Response.*** *There's an error here that will be fixed. The terms Science Plan and Science Program are similar. The major difference is that Plans have an ordering associated with the observations they contain. Also, Plans will likely consist mainly of links to observations in Programs. Programs will have links as well, but these will mainly be links to shared calibration observations. An observing session is associated with a Science Plan, since ordering of the observations is important.*

### 3.6.3 Section 7.1

The need for the requirement to support concurrent observations within a single session has already been raised in the comments on the IOI track.

***Response.*** *See "Concurrency within a session" on page 2.*

### 3.6.4 Section 8.1, Session Information Management

1st bullet. How is this achieved? The entity controlling the observation is the sequence executor. Does it somehow get routed from the sequence executor to the session manager and thence to any other participants in the session.

***Response.*** *Yes, the Session Manager creates Executors to carry out an observation. The Executor reports its status to the Session Manager, which knows which participants are associated with the session in which the observation is running.*

### 3.6.5 Section 8.1, Session observation Execution

The requirement for concurrency has already been queried elsewhere.

***Response.*** *See "Concurrency within a session" on page 2.*

### 3.6.6 Section 10.0

"some typical use cases..."?

***Response.*** *A use case is a description of an entire transaction in a system. All the interactions between objects that are involved in the transaction are described informally. The use cases describe the activities users will want to do with our system and allow us to check the design.*

Fig 6 - the methods don't have any parameters associated with them. Is this deliberate?

***Response.*** *Yes, the idea is to focus upon the messages and methods, and not to get caught up in the details.*

Fig 7 - how does the session manager determine to which session to add the participant if there is more than one session?

***Response.*** *There is no hardware support for this. The operator knows who to expect to participate based upon the proposal, and so he can assign them to the appropriate sessions.*

C5 background - "the SM does this to the first next Observation..."?

***Action.*** *The wording will be adjusted. Before starting an observation, the Session Manager queries it to determine the resources it needs.*

## 3.7 Observing Tool Track Preliminary Design

Figure 2

o Is "equatorial" enough? What about "FK4", "FK5", "geocentric apparent"...?

***Response.*** *The Coordinate System is actually a pull-down menu. When a system is selected the "Position" information box changes to match the particular system.*

o Why are the RA and Dec resolutions the same? RA should have one more digit than Dec. And it might encourage users to supply accurate positions if the example used 0.01s and 0.1" resolutions.

o "Epoch" should surely be "equinox".

o "Years AD" is not an astronomical timescale. If you put "equinox" there won't be any doubt what you're after.

o Why is pm "arcsec" and parallax "sec/arc"?

***Action.*** *All of these are valid points, and the prototype screen will be updated at some point. Please note that Figure 2 is for reference only. The real meat of the chapter does not depend upon the details of astronomical coordinate systems. We will get it straight eventually.*

## 3.8 Telescope Control Console Track Preliminary Design

### 3.8.1 Section 8.0

Is it UOB or UOC?

***Action.*** *Neither, it should be TCC, Telescope Control Console. The document will be fixed.*

### 3.8.2 Section 10

Table 1 - what do asterisks on Mount & Rotator console descriptions signify?

***Action.*** *They signify that the requirements for these two consoles might be satisfied by the engineering consoles that must be produced. Though the text indicates this, the table will be updated to make it more explicit.*

10.1 Third bullet. This statement is too one sided. There may be engineering functions that are essential that don't fit well into the OCS model. Engineering screens are directed at different users than the OCS screens.

***Response.*** *We will consider this comment and make changes to this section after consulting with GCS managers.*

10.1.1 Are the prototype screens referred to here those in the SDD or something that will be generated as part of the OCS work package?

***Response.*** *They refer to the screens in the SDD.*

## 3.9 User/Observing Console Track Preliminary Design

Section 7.3.1

How do the facilities provided by the PV-Wave scripting language compare with those provided by the Tcl scripting interface which is a product of the IOI track?

***Response.*** *This is a very broad question which we are not willing to undertake. PV-Wave is specified by the project, not the OCS team. There is a requirement to support it so we will.*

## 3.10 Preliminary Sequence Command Specifications

### 3.10.1 Section 5.1.1

The first paragraph implies that either the OCS must apply the commands in an order acceptable to the PS or there must be some means of notifying a PS when it wants the configuration applied. This in turn means that all PS commands must be of the "preset" type where the data is validated and stored internally but not acted on until the 'apply" is received.

***Response.*** *This issue has been covered in "Sequencing" on page 3.*

### 3.10.2 Table 1

The definition of config(reset) seems inconsistent. How can the system be set back to its start-up state if it doesn't read its initialisation file?

***Response.*** *Many systems support the concept of a "warm start" and a "cold start". That's what we are getting at here. These commands are not new and have been reviewed before.*

config(apply) does this mean a PS still needs a CAR apply even though it doesn't have a CAD apply

***Action.*** *The comment about the apply CAR will be removed from the table.*

### 3.10.3 Section 6.0

It is unclear whether a principal system can stipulate what order commands are sent in. The second and third sentences in Section 6.0, para 2 seem to be contradictory.

***Action.*** *This should no longer be an issue (see "Sequencing" on page 3). The wording must be updated to reflect the new paradigm.*

### 3.10.4 Section 7.1

Is the scenario being discussed here the implementation of a "clear" on a CCD?

***Response.*** *Yes.*

## 3.11 Preliminary Design for Access Control in the OCS

Section 7.4 This section will need to be reviewed in the light of the PS requirements. e.g. it needs to be decided under what circumstances access from the engineering screens will be allowed for example. Also, my reading of the CA access security mechanism is it operates by user id not group id. This would mean that in the security configuration file we will have to list all the potential members of the operator and operations group.

***Response.*** My reading of the CA Security docs suggests groups are okay. We will read the sections again. The CA Security is designed to work with User Access Groups. Each User Access Group is a set of User IDs so Chris is really right. We will need to keep parallel groups in the IOC and in the Unix hosts to make this work as described in the paper. This will need to be done anyway.

## 3.12 Preliminary Design for Configurable Control System Concurrency

It is not clear to me that allowing concurrency within a session is any different to the scheme originally outlined in the SDD which is described in section 5 as "a more complex design than is required to support efficient use of the telescope".

***Response.*** *The paper may be somewhat over zealous in describing the benefits of the new approach. The SDD design allowed resources to pass freely between instruments (what we call sessions). As long as there were observations to run, they would. It was much more automatic than the current design. The PDR design requires the operator to actively participate in assigning resources to sessions. The session concept does simplify thinking about the design, and it meshes well with the way we believe the telescope will be used.*

If all observations were assigned to a single session then the description in the 2nd bullet implies we would get full automatic concurrency so why bother creating multiple sessions in the first place? I can see the purpose if the other session belongs to another observer but for planned observing this wouldn't be the case.

***Response.*** *It is true that if all resources were assigned to one session and there were observations in the session that required disjoint sets of resources, they would execute as suggested. The system supports this but it is probably not how the system would be used.*

What would be the consequences of dropping the requirement that concurrent operation of observations within a session be supported? Concurrency could still be achieved but only by starting another session.

***Response.*** *See "Concurrency within a session" on page 2.*

## 3.13 User Activity Models in the OCS

### 3.13.1 Section 5.0, Point 3 of the Scenario section

Would it not be better to submit the proposals to the NGO for technical assessment and prioritising in one step rather than sending them to the IGO then back to the NGO?

***Response.*** *This is an issue for the Science Operations group. The scenario described here was presented by a project scientist in the Operations group. Either way, there is no impact on the OCS.*

### 3.13.2 Section 7.0, Comments section

Does the term "observer" used here refer to the OSO or the external observer?

***Response.*** *It refers to the on-site classical observer or the on-site staff observer (whomever is blessed). The burden should be upon him/her to provide as much information as possible.*

Under "OTHERS?" are there any planning tools to efficiently schedule observations? even just tools to show rise/set times, moon phase etc.

***Response.*** *This is the subject of the Scheduling track paper. I think the limited planning tools in the OT should show things like rise/set times, as you suggest.*

### 3.13.3 Section 8.0

A comment only. I think how "ambitious" an observer is will be directly related to the quality of the OT. If the tool makes life easier for an observer then all observers will fall into the "ambitious" category. If the OT can not be shown to give an advantage then observers will fall into the "sloven" category. Following on from this, Gemini should quantify the increased efficiency that is achieved by using the OT and then require all observers to use it.

***Response.*** *I agree the quality of the OT is directly related to its usefulness, and likelihood that it will be used. We will work very hard to make sure the OT is successful.*

#### 3.13.4 Section 9.0, Point 10

How is the operator to judge that the configuration looks good? What is he going to be looking for that the system won't check for when he pushes "Go"? If there is nothing then the observation should start as soon as the resources are allocated with the option that the operator can over ride if he wants to.

***Response.*** *Giving the operator an opportunity to evaluate the configuration doesn't imply that he always must. It also doesn't mean that he will catch a problem if one exists. We believe it is important to give the on-site operator a convenient means of finding out what the requested configuration is before it is applied. This currently means his screens will be configured to match the target configuration before it is applied. No active checks of the configuration are planned. In fact, it is an OCS/Gemini requirement that only the system operator control the telescope. Making him push "go" is the way the operator "controls" the telescope.*

## 4.0 Steven Beard's Remarks

Steven presented his comments in two separate responses, to allow us adequate time to review them. They have been recombined below. Like Chris Mayer and Patrick Wallace, Steven's comments are divided between general remarks and comments specific to particular documents.

### 4.1 General Remarks

#### 4.1.1 ICD1 and ICD2

I notice that ICD/1a, ICD/1b and ICD/2 are not part of the review material. These ICDs are vitally important for the Gemini software system, and I think their contents needs to be reviewed. I assume that a description of the ICD/1 and ICD/2 interface is contained throughout the OCS PDR material and these ICDs will be updated if necessary after the PDR.

Who is responsible for designing and implementing the CAD, CAR and SIR records, and who is responsible for reviewing them? Are they outside the scope of the OCS work package? (The OCS documentation says these are under the control of the Gemini Project Office). I think their design needs to be discussed and reviewed, and the OCS PDR would be the best time to do it.

***Response.*** *The OCS is not charged with the design and implementation of the ICDs. Like any other Principal System group, we can suggest changes to the design, but control of the ICDs and development of and records is in the hands of the project office.*

#### 4.1.2 Adding arguments to CAD records

I am worried by the fixed number of arguments a CAD record has, and the fact that some of the commands being thought of for the A&G system need a lot of arguments already. Can the CAD record be redesigned with a variable number of arguments (e.g. by pointing to a linked list of EPICS records), or do we need to invent different-sized CAD records with differing numbers of arguments (e.g. SMALLCAD, CAD, BIGCAD)?

***Response.*** *This issue was discussed at the Principal Systems meeting. Since it does not directly involve the OCS design, it is not covered further here.*

#### 4.1.3 CAD record directives and sequencer commands

If a CAD record can accept "apply", "pause", "continue" and "cancel" opcodes to control the execution of its command, why is it necessary for principal systems to recognise the "PAUSE", "CONTINUE", "STOP" and "ABORT" sequencer commands? Can these not be implemented by sending the appropriate opcode to the OBSERVE CAD record?

***Response.*** *First recall that PAUSE and CONTINUE directives were removed in the Principal Systems meeting (also see "Section 1.7.1" on page 6). The PAUSE, CONTINUE, etc. sequencer commands, as*

*defined in our sequence command paper, are used to pause and continue the action of observing. Even though aborting and stopping the OBSERVE action could be implemented in CADs as the abort and stop directives this solution is not general enough. The sequencer commands are to be system-independent and that means that they can't be tied to CAD record features. For instance, the DHS may not have a CAD record interface. An ICS developer may implement the abort and stop directives in the Observe CAD, but they won't be used by the OCS. We need to retain separate CADs for the sequence commands. We will make name changes to make sure they aren't confused with the record directives.*

#### 4.1.4 OCS to non-EPICS bases Principal Systems interface

The OCS documentation says that principal systems will normally be commanded using EPICS channel access, and only unusual or visitor instruments would use anything else. However, the DHS is a standard principal system that doesn't use EPICS. What mechanism will the OCS use to command the DHS? Portable channel access and DRAMA/DITS are possibilities.

***Response.*** *We are having discussions with the DHS on this issue.*

#### 4.1.5 PSAs

The documentation says there will be one Principal System Agent for each Principal System. I assume the OCS will have the following PSAs:

- One for the TCS
- One for the DHS
- One for each of the four instruments that are currently installed
- One for the High Resolution Wavefront Sensor and Acquisition Camera

i.e. there can be up to 7 PSAs active at one time.

***Response.*** *The idea that the HRWS/AC is a separate instrument is new to us. However, you are correct that there is a separate PSA for each instrument, for the TCS, and for the DHS. This should not cause any problems.*

#### 4.1.6 Immediate completion

Is the special case for immediate completion of a command executed through a CAD record really necessary? What are the benefits, for example, of activating a CAD record to set a variable rather than just setting that variable with a channel access put (to a SIR record)?

***Response.*** *Immediate completion was rejected at the Principal Systems meeting because it was believed to be an unnecessary feature. If we find that there are many commands that complete immediately, then this decision may have to be revisited. At any rate, CAD/CAR is the interface between the OCS and EPICS based principal systems. We feel it would be poor design to try to bypass this interface, even for simple commands that just set a variable.*

#### 4.1.7 Outside interruption of commands

Can the OCS cope with outside interruption of a command, for example someone using an engineering console to alter the target position for the telescope? Will this modification be noticed or will it be missed because the request didn't go through the TCS PSA? I guess this sort of access can be controlled using Channel Access Security.

***Response.*** *If the outside command uses the CAD record interface, then the cl_data field will be sufficient to detect the interruption. If the IOC sets the CAR status, the rest of the system will know about its actions. As you have pointed out, if we rely on the PSA to monitor conflicting command application, then any command originating outside of the OCS would give us trouble.*

#### 4.1.8 Need for cl_data field

On a related note, the OCS has been designed to be able to cope with one operator interrupting another operator's command, and the first operator being notified if this happens. Will this ever happen in real life if access to resources is carefully controlled?

Am I right in thinking the most common use for this interruption capability would be to allow an operator to interrupt a self-initiated command or be notified when a self-initiated command interrupts the operator's command?

***Response.*** *See "Introduction of cl_data in CAD/CAR design" on page 2.*

#### 4.1.9 Action variable protocol state machine

I think there are a couple of flaws in the action variable protocol state machine. In the current model an error will be ignored if it occurs during the transitory period while the CAR record is passing through the MODIFIED or ALERTED state. I think these states should also recognise an "error" input and move to the ERROR state.

***Response.*** *This situation is not possible. The CAR record should move to the MODIFIED (or ALERT) state and back to BUSY atomically in one record processing. If the next input is error, the transition will be made from BUSY to ERROR at that point.*

I think the PAUSED state should also recognise an "error" input. What if some hardware goes down while an action is paused, for example?

***Action.*** *We will add an error transition from PAUSED in our proposal.*

While debugging a Gemini system it would be useful to be able to log the inputs received by the CAR record, even if they are ignored and result in no change of state. (at the moment I can think of no circumstances in which an "error" would be received in the IDLE state, but you never know...).

***Response.*** *Agreed.*

#### 4.1.10 Argument validation in CAD record subroutines

Sometimes there may not be sufficient information to check all of the arguments of a CAD record at the time the command is accepted. I haven't seen any recommendations for what to do in this case in the OCS documentation, but I assume the correct thing would be to accept the command and trigger an "error" in the CAR record if when the time came an argument was not acceptable.

***Response.*** *I believe this is the only appropriate action that could be taken. Before a command is accepted the CAD record subroutine do its best to check the arguments. If a problem is encountered later, an error should be written to the CAR record.*

#### 4.1.11 OBSERVE/ENDOBSERVE semantics

Your model of what an instrument does on receipt of an ENDOBSERVE command is not correct. An instrument will read out its detector and store the data to disk as soon as an OBSERVE command finishes. ENDOBSERVE tells the DHS when the data are available. I think we will need to discuss this.

***Response.*** *We disagree on the semantics of ENDOBSERVE. In our view, the OBSERVE command contains an "observation id" argument. OBSERVE tells the DHS to expect data from an ICS with this id, and tells the ICS to begin an exposure. When OBSERVE completes the instrument reads out its detector and begins to pass the data along to the DHS, using the given observation id. ENDOBSERVE is issued when OBSERVE finishes, and it completes whenever the detector is finished reading out.*

***Action.*** *We are discussing this discontinuity with Steven.*

#### 4.1.12 Observer interactions with executing observations

The design for planned observing seems good. Will an observer have the ability to: (a) redo an observation if it was not satisfactory; and (b) extend an observation if it looks likely that a sufficient signal to noise will not be achieved? (This comment is related to the next one).

***Response.*** *Case (a) is handled by the design. The observer has control of the observation queue and can reorder/add to it at will. In talking with the project scientists, we don't believe support for (b) is required by the Gemini Control System.*

I think an observer needs more ways to interact with an executing sequencer recipe than just pausing and continuing the current observation. There needs to be an ability to pause the recipe as well. Here is a list of examples:

a) Pause the current observation.
b) Continue, stop or abort a paused observation.
c) Wait for the current observation to complete and then pause the sequence recipe.
d) Stop or abort the current observation and then pause the sequence recipe.
e) Modify a paused sequence recipe (e.g. add more observations or increase the exposure time).
f) Continue or abort a paused sequence recipe.

All these functions were needed by the CGS4 EXEC system, which had a similar purpose to the OCS sequence executor. This degree of interactive control enabled the observer to adapt to changing weather conditions (e.g. a deterioration of seeing in the middle of an observation resulting in an increase in the required number of observations). It would be useful to talk with Alan Pickup on this subject, as he designed the CGS4 EXEC system.

***Response.*** *The queue of observations is dynamic, being controlled by the observer with the ultimate approval of the on-site operator. The design will allow all the operations above. Some operations are done by the operator and some by the observer.*

#### 4.1.13 Use of PVWave for data visualization

I notice the OCS design assumes that PVWave will be used for data visualisation. Is this Gemini decision now cast in stone? If it is not, when does the OCS need a firm decision?

***Response.*** *PV-Wave continues to be one of the baseline software tool decisions. Severin and OCS will be discussing this (I think).*

#### 4.1.14 High Resolution Wavefront Sensor and Acquisition Camera

Please note that the High Resolution Wavefront Sensor and Acquisition Camera (HRWFS/AC) should be treated as a Gemini instrument rather than a TCS subsystem. It should therefore have its own Instrument Console. When it behaves as a HRWFS it generates data that needs reducing, just like other instruments. When it behaves as an acquisition camera it generates images to be displayed by a quick look server.

I think it would be useful to treat the HRWFS/AC as a separate resource, as this will allow the OCS greater flexibility (in case an observer wishes to use another instrument as an acquisition camera for example). However, in normal use the HRWFS/AC should be allocated to any observing session which requests the telescope beam resource.

***Response.*** *As mentioned above, the idea that the HRWFS/AC is a separate resource is news to us. Fortunately we have taken an abstract view of resources in our design, and so can accommodate new ones.*

#### 4.1.15 Development of instrument consoles

I agree with your plan for the development of instrument consoles. There needs to be a collaboration with the instrument groups, especially with the project scientists for the instruments. However, what happens after the OCS work package has been completed and a group somewhere wishes to develop a

completely new instrument (I am optimistically assuming that Gemini will be so successful that its collection of instruments will grow steadily)? In this case who develops the instrument console?

***Response.*** *I would imagine the maintenance staff will be responsible for developing new consoles using the support libraries provided by the OCS work package. In any case, this isn't an OCS development issue, and so it won't be commented upon further here.*

#### 4.1.16 Planning Tool usage

Can the Planning Tool be used off-line at an Observer's home site? I notice this tool interacts with the Status Alarm Database, which I assume will only be available on the Gemini Local Area Network. I think it would be useful if observers could try out the OCS tools and construct their science programs in advance. (I may be barking up the wrong tree here. Is the Planning Tool the one that allows this, or is it something else?)

***Response.*** *Observers should be able to construct both Science Programs and Plans in advance. The Planning Tool is used to query available Science Programs to aid the construction of Plans. If used on-line, one of the query options will be something like "show me the observations that could be executed given the current conditions." This query would need the up-to-date environmental information in the SAD. If used for preplanning observations, this feature would not be meaningful.*

#### 4.1.17 Consultation of the Gemini scientific community

I think consulting the Gemini scientific community to find out how they would like the OCS to work (in "User Activity Models in the OCS") is a good idea. Are the project scientists for Gemini instruments included in the group of people consulted? These people will have an idea how they would like their instruments to be used. (It may be worth having a discussion on the OCS at a meeting of the science instrument working group, for example).

***Response.*** *We will work with the various Gemini committees to assemble a group of testers for the observer programs as outlined in the PDR documentation.*

### 4.2 OCS Physical Model Description

I must admit I found the contents of this document difficult to grasp in the time available.

Section 1.7: I can see there is a potential for effort to be wasted generating OMT diagrams with an ordinary drawing tool and then redoing them for an OMT design tool. Is this likely to be a significant amount of effort?

***Response.*** *It will not be a significant effort to reproduce the drawings using a design tool. In fact, many of the drawings in the PDR booklet are repeated, showing slightly different aspects in each picture. At any rate, the design will continue to evolve through the detailed design and implementation stages. The effort of using an object-oriented methodology without tool support more than makes up for the initial cost of reproducing a few diagrams.*

Figure 4-2: This console makes a good example, but will the OCS really have a "Primary Support" console like this? It seems the sort of console that would be implemented in an engineering user interface.

***Response.*** *The Primary Support console is among those listed in the table of prototype OCS console screens in the SDD (Table 5-2, page 5-37).*

Figure 4-5: The bag pressure setting example does not use CAD and CAR records, so it doesn't seem a typical example.

***Action.*** *The diagram will be updated to match the current design of the Interactive Observing Infrastructure.*

### 4.3 OCS Interactive Observing Infrastructure Track PD

Section 1.7.3 says "The PSA does not monitor CAR records", yet section 1.7.4 says "The PSA monitors the ARD". Is there a contradiction here?

***Response.*** *Yes, this is a contradiction. The paragraph in Section 1.7.4 is wrong (see "Page 1-7, 2nd paragraph" on page 7).*

Section 2.6.1: I don't think R25 is true. The SAD by its nature is a public database and could therefore be monitored (or even changed) by other software outside the OCS. Maybe by "software library" you meant "OCS software library"?

***Action.*** *Correct. This requirement refers to OCS software and the document will be updated.*

Section 4.4: Table 4-1 says that the ERROR state can be cleared back to IDLE by receiving a "done", but this path is not shown in Figure 4-3.

***Action.*** *Will be fixed in Figure 4-3.*

Section 5.2 (R14): Can a single CAR record be shared by several CAD records? I note that have a single shared CAR record would make the situation described in section 5.4 worse. We will need to discuss this.

***Response.*** *Yes, there is no restriction on the number of CAD records that affect the same CAR. It isn't clear to what situation the second sentence above refers.*

Chapter 6, scenario 4: If I were designing the user interface I would leave the light amber rather than green to indicate that the filter wheel has been left in an illegal position. I would also set the OBSERVE command CAR record to UNAVAILABLE. Does this sound ok?

***Response.*** *The details of colors/situations can be worked out in the detailed design. However, the action variable protocol does not easily support the situation you've described. Unless we extend the protocol and require that the PS write a special value to the CAR record, we will only notice a transition from BUSY to DONE. Also, the abort command completes successfully, so moving back to green is consistent. The fact that the command was aborted will be reflected since it will cause the CAR record to move to MODIFIED temporarily when the abort command is accepted. Finally, simply because a command was aborted does not imply that the actions are UNAVAILABLE.*

Scenario 7: Although there will be a few milliseconds when the offset button push cannot be accepted, will the push not be queued in a buffer by the user interface anyway?

***Response.*** *No, the button cannot be pressed a second time until the command resulting from the first push is accepted or rejected. GUI toolkits are usually designed so that when the button appears depressed and any clicks are ignored.*

### 4.4 Preliminary Sequence Command Specifications

Table 1 (apply): Do all principal systems have to have an APPLY CAR record, even though they don't have an APPLY CAD record? This violates the recommendation in other parts of the OCS documentation that each CAD record should have only up to 1 CAR record.

***Action.*** *The statement "Principal systems should cause the apply CAR record to become BUSY..." should not have been included in Table 1. It will be removed.*

Table 1: (observe/endobserve): Both these commands need a file name argument as well as observation ID (unless you are planning to name files based on observation ID).

***Response.*** *The DHS handles data storage. The OCS doesn't work with files or tell the DHS which file names to use. The DHS will likely map observation IDs to filenames, but that is internal to the DHS.*

Figure 1: Instruments do not read out their detector during ENDOBSERVE. Instruments will usually ignore ENDOBSERVE.

***Response.*** *See "OBSERVE/ENDOBSERVE semantics" on page 18.*

Section 7.1: Possible solutions are: (a) monitor a shutter open record; or (b) use a shutter open "alert" sent to the OBSERVE CAR record.

***Response.*** *We would like the recipes to remain principal system independent if possible. When is an IR instrument shutter open? We are discussing sequence commands with Steven.*

Section 8.0: I am completely confused by this section. It implies that every principal system has an APPLY CAD and CAR record, which contradicts all the other OCS documents I have read. Please can you explain.

***Action.*** *You were right to be confused. Apply CAD/CAR did not exist in the PDR version and should not be in Table 2.*

## 4.5 Preliminary Design for Access Control in the OCS

This looks good. My only comment is that saying you trust users (e.g. page 4) makes you sound a little cavalier about security. The fact is you can trust users because they have already been through a screening process (telescope allocation committee) and have already had to log in to the system and supply a password.

***Response.*** *We never meant to imply otherwise and apologize for the cavalier tone.*

## 4.6 Planned Observing Support Track Preliminary Design

Section 9.1: Despite its preliminary nature, I like the way the session manager is going. It looks like a much more powerful system than I have seen at any other observatory. I can see some features that might enhance this tool, for example: (a) Each observation could be displayed inside a time window in which that observation had to take place. (b) Each observation could have an icon summarising the observing conditions (good, moderate or bad) it requires, and the weather monitor could be displaying the icon representing current conditions. This might allow the operator to make decisions about which observation to do next. See comment 13 above about the need to be able to redo failed observations and extend observations.

***Response.*** *These are good ideas and will be considered during the detailed design*

Section 13.0: It may be worth including some KPNO telescope operators in these discussions. They may not be officially connected with Gemini but they are a local resource you could use for ideas.

***Response.*** *We will consider and research this idea.*

Table 23: Why are the GUIDE and ENDGUIDE commands not mentioned?

***Response.*** *The interface describe in Table 23 is the Executor Controller interface. It is the public interface that allows control of an executing Sequence Executor. A system controlling a sequence executor will probably not need to tell the executor to send GUIDE and ENDGUIDE. This may need to change as we further define this part of the system.*

## 4.7 User/Observing Console Track Preliminary Design

Section 9.0: This implies that the GMOS PDR material needs to contain some preliminary user interface requirements. (Not an OCS comment, just an observation for GMOS).

Table 1: Please add "HRWFS/AC" at both sites.

***Response.*** *Even though we are now viewing the HRWFS/AC as a principal system, I'm not sure it is a user instrument. It seems to me it's more like the secondary mirror than the IR Imager.*

Are you sure a mid IR imager is being delivered to Cerro Pachon? That sounds strange. Has Susan Wieland verified this list?

***Response.*** *I believe this list is accurate. We will double check the table.*

Please note that Michelle is a mid IR spectrograph.

***Action.*** *Table 4 will be updated.*

### 4.8 Preliminary Design for Configurable Control System Concurrency

Section 7.0: I mentioned in part 1 of these comments that instruments do not normally read out their data at the ENDOBSERVE stage.

***Response.*** *Agreed, they begin to read out as soon as OBSERVE completes. I don't find a reference to ENDOBSERVE in Section 7.0. Also see "OBSERVE/ENDOBSERVE semantics" on page 18.*

Note that there is the possibility that the observer might verify an observation *before* the detector has finished reading out. Some detector controllers will have the ability to transmit their data a bit at a time, and the quick look display will gradually build up. The observer might watch this display and then give the go-ahead for the slew to the next observation when he/she has seen enough to be convinced the observation has been a good one. We need to consider this possibility as well.

***Response.*** *I believe this case is handled by option 3: move to the next observation after the observer explicitly verifies the observation.*

## 5.0 Keith Shortridge's Comments

The comments are divided between general and specific topics.

### 5.1 General Remarks

I've worked my way through the OCS PDR documents, although it took me much longer than I expected, so as usual I'm sending in my comments at the very last minute. I'm sorry about that.

#### 5.1.1 Complexity of the Design (Reprise)

I'm actually finding it hard to say anything very much that's useful. I've just reviewed my comments, and most of them seem to be vague and occasionally philosophical rather than really concrete. I'm impressed by the evidence that the various ways the system will work have been thought through pretty carefully. This is mainly a reference to the use of scenarios, which I found very useful; not only do they demonstrate that the system can handle the situations they cover, but they also help to give an outsider a 'feel' for the way the various bits of the system fit together. Getting this 'feel' was the hardest bit for me. Going through the OCS documents, there were times I felt I was getting a detailed description of some of the trees, but the overall shape of the wood was still something of a mystery.

When a group is sufficiently far advanced with a design, they know the overall framework so well that they no longer feel the need to describe it, and indeed they shouldn't need to describe it in the detailed documentation for individual parts of the system. However, I felt that an informal description of the 'big picture' would have helped. However, in the end I think I got there.

***Response.*** *It is becoming clear that effort should have been put into an overview document of some sort, perhaps a series of diagrams labeled with pointers to applicable documents. We did tend to focus*

*on the details of how the various applications interact, since we believe that is the purpose of the preliminary design. See also "Complexity of the Design" on page 2.*

*As mentioned in the SDR documents, we viewed the SDD OCS design as a prototype to be used to extract requirements for the OCS. The OCS is now in the Architectural Design phase of the ESA software process, also called the solution phase. In this phase we are to show an implementation-independent solution that can satisfy the requirements and the uses described in the SDD. An important part of the PDR documents is the rather abstract physical model which we are committed to. So while the PDR documents may seem abstract, I believe the design is more comprehensive and now considers many more issues that previous designs including communication with systems other than EPICS-based systems, operator activities, data-flow in the OCS, access control, and simultaneous observations.*

### 5.1.2 Abstract Nature of the Design

The big picture, in fact, seems to have become more abstract than it was at the SDR. In a way, that's good, because it's good to describe a layer of the system in terms of just the capabilities of, say, a GUI, rather than explicitly assuming this is TCL/TK. However, it does make it harder to visualise the system in operation, and I'm a little concerned that the project is this far advanced and still giving the impression that so many implementation decisions are still to be made. I hope this is a false impression that comes mainly from the abstract nature of some of the system descriptions.

***Response.*** *It isn't clear to us that specifying particular software packages to use during implementation aids understanding of the design. Further, it unnecessarily constrains the design. As you know, the design should drive the implementation choices, not vice-versa. The project is only beginning the detailed design of the first track (the Interactive Observing Infrastructure track), so we believe that the lack of implementation specifics is appropriate.*

### 5.1.3 EPICS Driving the Design?

Having said that, another general concern I have is that the whole system design is nevertheless very heavily influenced by the use of EPICS at the lower layers. However, I think the move towards working with partial configurations is good - I was always rather concerned about the need to work with a complete configuration at the top levels of the system. There is a tendency in the documents to give the impression that you're having difficulty meshing a status-driven system (which is how EPICS comes across) with the command-driven system you want at the higher levels. I think that what's really happening here is not so much that EPICS is not event driven (after all, it is!) but that you are building a hierarchy on top of an event- driven flat lowest level, and some of the crispness of the lowest levels is in danger of being lost in the hierarchy.

***Response.*** *It is difficult to respond to this comment. You are correct that a conflict is occurring between our requirements and the support that EPICS provides. We have a system based upon commands, yet there is no support for traditional commands in EPICS. We must be able to send instructions to Principal Systems, and we must know when those instructions have been carried out. This requirement has lead us to the command based system described in the PDR. We feel in our environment building this upon EPICS is absolutely necessary. As for losing "some of the crispness", we see it more in terms of making EPICS more usable. We also feel that we need to use as many of the features of EPICS as possible so traces of the EPICS design are visible in the design. These are usually features of EPICS that we feel are good features.*

*We are seeking the advice of members of the EPICS consortium who will review portions of our EPICS interface, and will fold their comments into the design.*

As you clearly realise, it's largely a question of completion of commands and the origin of commands. If someone starts a filter moving in EPICS, eventually the filter gets to the required place, but by then EPICS itself has lost track (never cared, actually) of who asked for it to move, and so getting a report back to the command originator gets complex. And having two competing commands trying to move the same filter gets really messy. This is something that you've tackled in some detail with the Action

Variable Protocol, and the scenarios convince me that it's been well thought through. The description of it keeps using the term 'monitor' and I would like to see it made explicit that this does not involve polling, but is a proper event- driven mechanism. I'm thinking here of phrases such as 'The CLL CAR monitor protocol notices that the CAR went from BUSY to IDLE' (IOI Track, page 6-2, item 7 in scenario 2), where 'notices' is not the same as 'is told'. Am I being too fussy here? I hope this is just a question of presentation, rather than anything deeper.

***Response.*** *You are correct that monitoring should not (and does not) imply polling in our system. We are planning a subscription based system where applications register their interest in a variable with a callback function. When the value of the variable is updated, the callback function is invoked.*

#### 5.1.4 Configurations Make Interactive Observing Awkward?

Interestingly, I suspect that working with configurations, previously set up as part of a planned observing system, just split up and fed to the relevant parts of the system that have to configure their parts of the system, makes for a neat meshing between the planned observing and the EPICS systems, which may make it harder to modify things piecemeal in an interactive observing mode. I know that Gemini is predicated on the need for scheduled observing, and my only concern here is that I expect that it will take some time for this to start operating smoothly, the initial usage of the system will tend to have a larger interactive component than later usage, and that any awkwardness in the interactive part will reflect badly (albeit unfairly) on the Gemini software system. There may be a number of 'sloven' observers ('User Activity models, p6) in the early days.

***Response.*** *We don't share the concern that the configuration idea will complicate interactive observing. Classical observing will be conducted primarily with instrument and telescope consoles, in much the same was as observatories have been running for years (albeit with more complex hardware). The fact that configurations are the underlying implementation mechanism hopefully will not be apparent to the users of the system.*

#### 5.1.5 Summary of the General Remarks

I'm struggling here to try to articulate some nagging worry. I think the design has become rather abstract, rather complicated, and while I actually think it's been well thought through, I worry about your chances of getting it completed in the specified time-scale. I'm open to reassurance on that score. For all I know, most of this has already been coded and is running! In fact, revisiting these comments having just typed in my detailed comments, I get the real feeling that the system design is rather better than the impression of it provided by the PDR documents, which in many places have a lot of very abstract formalism where a plain English description of what these things mean in practice would have been much clearer.

***Response.*** *Others have expressed similar concerns (see "Complexity of the Design" on page 2). As for plain English descriptions, I would have to see examples cases that were treated with too much formalism. I'm sure they exist, but its easier to explain individual cases and to get an idea of exactly what you consider to be unnecessarily confusing.*

### 5.2 Access Control in the OCS

I see this was 'reference only', but I'd like to say that it doesn't seem to allow for trustworthiness changing with time. It isn't really that 'user x' is 'trustworthy', it's more that 'user x is tonight's observer, so we have to risk trusting him for this night but tomorrow he won't be the observer and we don't want him anywhere near the system.'

***Response.*** *The model described above is just what is supported in the Planned Observing Infrastructure. For planned observing, the observer must contact the Session Manager application to initiate a session. The on-site operator explicitly grants resources to the session. The observer's Observing Tool then interacts with the Session Manager alone to start/stop/pause observations. Ultimately, access to system resources is controlled by the on-site operator, through the Session Manager.*

### 5.3 Configurable Control System Concurrency

Again, 'reference only', but let me make one general comment. At present we don't often see concurrent observing. What we do see is observing overlapping with testing and with calibration. In these cases, the testing or calibrating systems run better by simply starting up a completely new observing system but with the shared resources (which they don't want to use anyway, such as the telescope) running in simulation modes. Running multiple instances of a whole system is often easier than having one huge system trying to manipulate and separate multiple users. I realise that when you have genuine contention for systems, then the sort of thing described here is necessary, of course.

***Response.*** *For the example of observing overlapping with testing, we also intend to use separate sessions. Concurrent observations within a session are meant to deal with those cases in which an observation needs a resource for only a portion of its duration. Once one observation is finished with the resource (not necessarily when the observation completes), it can release it for other observations to use. We can't predict how often concurrent observing will occur, but supporting it is one of the requirements of our design.*

*See also "Concurrency within a session" on page 2.*

### 5.4 OCS Physical Model Description

#### 5.4.1 Section 1.6.4, the development process

The various calls documented in this all seem to be 'conceptual' (and reasonably so). I mean that generally when an API is completely fleshed out and implemented all sorts of additional parameters often appear connected merely with housekeeping and making sure the various libraries actually have all the information needed to do the job. There isn't any sign of that here; I assume this means that the implementation step hasn't happened here, and only wanted to comment that this is the point when a number of changes often end up getting added, and I can't see that feedback step in the various design stages. (Maybe it's only me who designs things so much on the fly that these changes are needed.)

***Response.*** *The development process isn't strictly sequential, as implied by the terse discussion in Section 1.6.4. In fact, an iterative approach is recommended by Rumbaugh in which a minimal skeleton is first implemented. The skeleton handles a single task and is developed at each step of the process. Additional functionality is then added to the skeleton incrementally [1]. This idea is present in the development process through multiple alpha and beta releases.*

#### 5.4.2 Section 2.4.1, the SVC architecture

Only to comment that dividing systems into layers makes for good systems design, but often can result in systems that don't work as well as monolithic systems because 'this bit can't do this because to do so it would need to know such and such and that's the perogative of a different part of the system'. I seem to have had to use this line too many times to frustrated users, none of whom ever accept it, and who certainly don't care that having done it this way is the only way to make it maintainable, upgradeable, etc. I don't think there's a solution to this, and I'm sure you're all aware of the problem. This seems to get worse with distributed systems, as well, where the 'localisation' of information is often a big problem - I was amazed in our 2dF system how many parts of the system needed to know the telescope position, and one part even needed to know the details of the telescope distortion model! (I forget why - I've tried to erase the whole thing from my mind..)

***Response.*** *Noted.*

#### 5.4.3 Section 2.4.1.3, Views

Views need to know about subject data but subjects do not know anything about views True in theory, and desirable, but often in practice a subject needs to send the data to the view and so needs to know that it needs it. (Otherwise, the view has to poll for it.) At AAO we certainly end up with a lot of specifications that allow a lower level task to be told the name/location whatever of a higher level task to which it

is to send information, which allows a low level task to be built without assuming the existence of specific higher level tasks, but still able to communicate directly with them.

***Response.*** *We do not plan to rely upon polling. Using a package like Tcl/Tk for instance, you can assign callback functions to variable changes (variable traces). All that is needed is some mechanism that allows the subject to alert the views and controllers of changes.*

#### 5.4.4 Section 3.3.2, System subject communications

Just a moment to ride a personal hobby horse about not liking calls such as 'putWait()', which block until a response is received. You always have to ask 'what happens if the response doesn't arrive?'. I get the impression that some of the blocking calls look better if implemented in a multi-threaded system, and would like to comment that even in a multi-threaded system you have the problem of dealing with the tidy closing down of errant threads. Although somewhere in the documentation the statement is made that Solaris supports multi-threading it isn't quite clear to me how much use is intended to be made of this. If you are going to use threads, you have to know how you are going to handle the case where one blocks and needs to be closed down. (Which can be worse than trying to close down a process in a multi-processing system, since the system will do more of the garbage collection in that case.)

***Response.*** *We do plan to use a threads package, and supporting for creating and destroying them is a minimal requirement.*

#### 5.4.5 Table 3.7, Logging manager methods

Error logging is a problem, and I wondered if you had looked at the ADAM or DRAMA error logging systems, which allow detailed error logging by low levels of the system which can be annulled by higher levels - this is because often only a low level knows all the details of an error, but only a high level knows if the 'error' is really an error that should be reported, or just an enquiry that failed or something equally innocuous.

***Response.*** *We will take a look at this. The DHS team is responsible for some of the error logging.*

#### 5.4.6 Section 4.3.5, The Air Bag console

I started to wonder about how values such as pressure were displayed in a meaningful way. If you have a spectrograph, for example, you might know the slit width in mm, but to display this meaningfully, you need to know what this comes to in arcsec. Doing this conversion might require knowledge of the optics of the system, telescope details, etc. Getting hold of this poses problems for a distributed system, but is something an astronomer expects to be simple! See the comments above on 'Architecture'. The physical values that a sub-system knows are often not what an astronomer needs to know, and the conversion can require information from many other sub-systems.

***Response.*** *In this case, the instrument would provide status items in the correct units. If values in different units are needed, the instrument makes the value simultaneously available in all units. The screen then shows a different database value depending on the observer's wishes.*

#### 5.4.7 OCSApp Apprehensions

General. I had difficulty getting to grips with the question 'just what is an OCSApp and who writes it?' Since it isn't a single task, but a multi-layered thing with views and subjects, it isn't necessarily all produced by the one team. If the user communicates with the 'controller', just what sort of a piece of software is the 'controller'? The diagrams seem to imply that the user communicates individually with the controllers of the various OCSApps that comprise the whole system. This sounds like a system built out of engineering level interfaces. Shouldn't the user be presented with a view of the Gemini telescope, including its instruments, as a coherent whole? That is, shouldn't the user talk to some controller, which then talks to the controllers of the various OCSApps? Do you plan to build an overall user interface as an OCSApp that does nothing other than talk to other OCSApps? I felt this wasn't made very clear.

***Response.*** *Each OCSApp is produced by the OCS team alone, though the "fixed-part", the System Subject is produced once during the Interactive Observing Infrastructure track.*

*We do not believe that the use of controllers implies or tends toward a system built upon engineering consoles. Controllers are simply the buttons, keyboard entries, dials, etc. that allow the user to interact with the system. No matter how we structure the OCSApps, controllers must be present. It is important to note that not all OCSApps will have a controller, or even a view for that matter. Only applications that directly interact with users (e.g., consoles, script executors) need these. The SVC model only describes how functionality will be organized in an OCSApp. It does not contain any functionality or requirements by itself.*

## 5.5 OCS Interactive Observing Infrastructure Track PD

### 5.5.1 Section 1.7.1, Terminology

'Clients of the IOI are consoles, script executors, and shell tools'. I'm surprised that there doesn't seem to be a place for what the ADAM/DRAMA world would call a control task - a sequencing program written in a high level language such as C or C++, which controls a particularly complex sequence involving a number of separate instruments, so as to be able to present a higher level of abstraction to the user, who can see a subsystem that combines spectrograph and detector to take an exposure at a given central wavelength with a specified exposure time, rather than having to interact with the various components. It is possible to write this sort of thing in TCL/TK, but there is a level of complexity - particularly when a lot of asynchronous actions are being juggled - at which C/C++ become a better match to the problem than is TCL/TK. I don't think the design of the system rules out such things, but the documents don't seem to consider them.

***Response.*** *I believe scripts provide the kind of functionality you are referring to by "sequencing program." Script executors will be used to execute scripts, and they will be linked to the Command Layer Library of the IOI track. This library will provide a high level interface for all the sequencing requirements we believe will be necessary. At this time we hope not to have to use C coded "control tasks". This would be possible, however, if we find it is needed,*

### 5.5.2 Section 1.7.2.2, Principal system agents

'Each principal system will have a PSA that represents its functionality in the OCS'. Can it provide a more abstract (astronomically meaningful) view of the functionality? I mean, does a spectrograph PSA represent a system with a grating that can be set to so many degrees, or can it represent a system that can be set to a central wavelength of so many angstroms? Again, the design doesn't seem to prevent this, but it isn't clear just how it is seen. (Although if all it does is act as a conduit for EPICS attribute/value pairs, the former seems more likely.)

***Response.*** *The PSA presents whatever functionality that the Principal Systems supports (as defined in its PDF document). Chapter 3 lists some of the reasons why we have an agent for each principal system. Perhaps the chief use of PSAs is to simplify the remainder of the OCS. From the viewpoint of other OCSApps, the PSA is the principal system. Its features and peculiarities are collected in one place, instead of being propagated throughout the OCS.*

*In the current design the PSA is like a DRAMA translator task. All the smarts for the target system are handled in that system. With the changes we've made to the EPICS CAD interface, controlling/sequencing system should be easier. We could eventually move the PSA into the principal systems themselves. Then instruments would receive configurations only*

### 5.5.3 Section 6.2.1, scenario 6

'Commands that have no real actions can choose to complete immediately meaning that the PS doesn't return until the command is complete.' I must be sounding boring by now, but this is only true if nothing goes wrong. If the task running the PS is hung, this will jam the system, or at least one thread of it.

***Action.*** *There will be a need for a time-out mechanism. This will be considered as part of the detailed design stage. We had a good discussion on time-outs at the principal systems meeting.*

#### 5.5.4 Section 7.4.0.2, middle of page 7-7, on trade-offs

'Sending an entire configuration sounds good from a design viewpoint..' I'm not so sure. I keep thinking of old hardware interfaces that were controlled by setting 16 bits in some parallel register. Some would let you switch on and off individual bits, which others would only let you send the whole 16 in one go. Working with the latter was much harder, because parts of the code had to know what other quite unrelated parts were doing, just to make sure they didn't interfere with the bits they were controlling. Maybe that's not a good analogy.

***Response.*** *If decided to send the entire configuration to the PSA (which we didn't in the PDR), there would be no requirement to send a configuration consisting of every possible attribute. The term "entire configuration" only applies to the idea that the parts are not separated by the CLL before sending them down to the PSA.*

### 5.6 Planned Observing Support Track Preliminary Design

Section 6.0 'Only one observer, the blessed observer...' I love the choice of 'blessed' as a term here. How is it pronounced, and is the ambiguity deliberate? (Or is it only ambiguous in British English?)

***Response.*** *The ambiguity wasn't deliberate, we meant blessed in the "chosen" or "anointed" sense (monosyllabic pronunciation). Though now that the ambiguity has been pointed out, it seems an especially good term.*

Section 16.7.1 What can be done and what can't be done. This sounds to me as if interactive classical observing might be quite limited. How limiting is the statement that 'sequencing of the principal systems is done by operator and observer?' Does this include sequencing the data handling - do you not only have to tell the CCD to readout but also tell the DHS to record the data? This seems to me to be connected with the point I raised earlier about levels of abstraction and C/C++ programming. If you have a hierarchy of layers of abstraction, then one could present the detector and DHS as an entity with a single command that did an 'expose, readout and record' and this layer would be eminently usable classically. However, if the observers have to do their own sequencing at a more detailed level, then this is not as good as one gets classically. I feel this is an important question.

***Response.*** *What we mean here is that observing would be done in the way it is done now at just about every observatory. The operator and the observer verbally communicate to "sequence the systems."*

### 5.7 High Level Design of the Observing Tool Track

General: I'd like to see a scenario detailing how an observer using the OT pauses an observation. Section 7.4 says this can be done, which is good, but figure 3 (the software environment) doesn't show any actual observing software below the session manager. As a result, I wasn't clear from figure 3 that it could actually be done.

***Response.*** *The OT doesn't interact with any software below the Session Manager. To get a better idea of the role of the Session Manager and lower level software, see the Planned Observing Support Track paper. Methods for starting/stopping/pausing etc. observations are shown there. We were rather sparse on high-level scenarios and will add more including the one you've suggested.*

### 5.8 Summary

Hard to summarise, I'm afraid. An impressive amount of work, obviously carefully thought through, and I think this can be the basis of a good control system. I think the Gemini team deserve to be congratulated for that, and the way they have gone to a lot of trouble to address questions such as the 'repeated offset button' scenario. Some rather philosophical reservations, connected mainly with questions of hierarchy, abstraction, and distribution of distributed data. A concern that the layering of the design may impose unnecessary limitations on classical observing. And a feeling that in many places the potential of the design is being obscured by the rather abstract nature of presentation in the PDR documents.

***Response.*** *I think some of Keith's philosophical reservations could be cleared up with some face to face conversations. Maybe during ADASS (if Keith is coming to Tucson) we can get together.*

# 6.0 Malcolm Stewart's Remarks

We didn't receive Malcolm's remarks via e-mail, so they have not been included verbatim here. Instead, the main points that haven't been addressed are presented. Another problem that is filtered in these responses is that an older, draft version of the documentation ended up in Malcolm's hands.

## 6.1 Multi-track Problems

Multi-track approach risks too many tasks being done concurrently. It is important that all post-PDR effort is devoted to the IOI track until complete, as indicated in the Gantt chart in the WBS.

The multi-track approach also risks several tracks failing to meet requirements, should resources have to be diverted to other "make or break" systems or if the tracks' effort are on average underestimated.

***Response.*** *Actually, the work is fairly sequential, especially at the outset. Our goal is to create a functional alpha version that will lay the foundation for the other tracks including the Observing Tool track. Next a version of the Telescope Control Console track (including the Console Track Library that will be used by all consoles) will be completed to give us a working interactive system from the console level all the way down to the CAD/CAR interface. We hope to have a test system running by the end of the year.*

## 6.2 OCS Consoles vs. Engineering Consoles

Do the TCC consoles duplicate functionality of EUIs (engineering user interfaces)? Could the consoles be used by developers of low level systems?

***Response.*** *In some cases this may be true. We've addressed this issue in the TCC track paper.*

# 7.0 Summary

We wish to thank all the reviewers for their insightful comments and questions. We are particularly appreciate the efforts of those reviewers who are not involved in Gemini on a daily basis. We can see how jumping into our documentation would be daunting.

Created:November 30, 1995
Modified:March 18, 1996

Gemini Observatory Control System Report

# *Comments on Final OCS PDR Report*

**Kim Gillies, Shane Walker**

Final Report Comments/01

## 1.0 Final PDR Comments

This book contains the design documentation for the Gemini Observatory Control system that was presented and reviewed at the Preliminary Design Review in August, 1995. These documents have been updated based on the comments of PDR reviewers. Those changes are described in the PDR response document also included in this documentation package.

There are sections of these documents—primarily the parts of the IOI track related to EPICS communication— that are not up to date with the current system design. The extensive developer discussions following the OCS review yielded several significant changes to the OCS-EPICS interface. We (the OCS team) decided that the PDR documents should reflect the reviewed design at the time of PDR, and consequently some sections do not match the current state of the design.

We plan on using this document set as the basis for future OCS design documentation and future versions will match the design. The ICD documents, maintained by the project office, should be used as the definition of the OCS to EPICS interface.

PDR

# OCS Physical Model Description

Kim Gillies*, Shane Walker*, Steve Wampler**

* National Optical Astronomy Observatories
** Gemini Project Office, Gemini Controls Group

**O**bservatory
**C**ontrol
**S**ystem
**P**roject

# 1 Physical Model Overview

*This chapter gives an overview of the process used to model the OCS and the model itself.*

## 1.1 Introduction

To understand the operation and design of the Observatory Control System and the Gemini Control System as a whole, the OCS has been modeled using an object-oriented design methodology. The Object Modeling Technique (OMT or the Rumbaugh method) is one of the most popular and frequently used object-oriented modeling techniques in use today. We have chosen to use the OMT method rather than a structured analysis methodology for the following reasons.

- The OMT method (as well as other OO design methods) focuses on the interfaces systems and subsystems present rather than the internal data transformations. Our distributed system also focuses on the various system interfaces making it easier to model how our system works.
- OMT also includes the important data flow and event flow information of structured analysis.
- The goals of the OCS design and the design of several of the products of the OCS map well to an object-oriented approach.

The primary references for the OMT method are Rumbaugh's original book [3] where he describes the method and a series of articles describing updates to the method also by Rumbaugh [4]. The method is not difficult to understand assuming some object-oriented background, but the notation takes some time to learn.

The following sections will present a brief summary of the method followed by the design of a single OCS application in the next chapter. The design of each OCS development track as a set of OCS application is then presented with supporting modeling information in subsequent chapters.

## 1.2 Acronyms

| | |
|---|---|
| API | Application Programmer Interface |
| ARD | Action Response Database |
| CAD | Command Action Directive |
| CCS | Configurable Control System |
| CLL | Command Layer Library |
| EPICS | Experimental Physics and Industrial Control System |
| GCS | Gemini Control System |
| ICD | Interface Control Document |
| ICS | Instrument Control System |
| IOC | Input/Output Controller |
| IOI | Interactive Observing Infrastructure |
| OCS | Observatory Control System |

| | |
|---|---|
| ODB | Observing Database |
| OMT | Object Modeling Technique |
| OO | Object-oriented |
| PS | Principal System |
| PSA | Principal System Agent |
| SAD | Status Alarm Database |
| SDD | Software Design Description |
| SDR | Software Design Review |
| SIR | Status Information Record |
| SVC | Subject-View-Controller |
| TBD | To Be Determined |
| TCS | Telescope Control System |

## 1.3 Glossary

**Attribute —** An attribute is a textual description of some part of a Gemini based hardware or software system. An attribute has an associated value.

**Value —** The value is the data associated with a particular attribute.

## 1.4 References

[1] SPE-C-G0037, *Software Design Description*, Gemini 8m Telescope Project.

[2] gscg.grp.020.icdbook, Interface Control Document 1a, 1b, 2. *Gemini Interface Control Documents*, Gemini Controls Group

[3] *Object-Oriented Modeling and Design*, James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, Prentice-Hall, 1991.

[4] *Journal of Object-Oriented Programming*, May 1994, October 1994, November/December 1994, February 1995, March/April 1995, May 1995, James Rumbaugh, *Modeling and Design* column.

[5] ocs.kkg.014, *Observatory Control System Software Requirements Document*, Gemini Observatory Control System Group

[6] ocs.kkg.032, *Interactive Observing Infrastructure Preliminary Design*, Kim Gillies, Shane Walker, Steve Wampler, Gemini Observatory Control System Group, 1995.

[7] cdev User's Guide, Chip Watson, Jie Chen, Danjin Wu, Walt Akers, CEBAF, Version 1.0.1, 7-20-95. (Available at cebaf ftp site.)

[8] ocs.kkg.036, *Preliminary Design for Access Control in the OCS*, Gemini Observatory Control System Group, 1995.

[9] *KoalaTalk: An ICE-Based Message Bus*, Cedric Beust, The X Resource, Issue 14, O'Reilly Associates.

[10] *KoalaTalk Reference Manual*, Cedric Beust, Version 1.68, see http://www.inria.fr/koala/beust/koalatalk_toc.thml

[11] ocs.kkg.033, *Telescope Control Console Track Preliminary Design*, Kim Gillies, Shane Walker, Steve Wampler, Gemini Observatory Control System Group, 1995.

[12] ocs.kkg.038, *Planned Observing Support Track Preliminary Design*, Kim Gillies, Shane Walker, Steve Wampler, Gemini Observatory Control System Group, 1995.

## 1.5 Document Revision History

**First Release —** July 30, 1995. Pre-release draft.

**PDR Release —** August 16, 1995.

**Final PDR Release —** September 26, 1995.

## 1.6 Brief OMT Overview

The OMT development process is to build models during the analysis phase of the project and then continually refine and enhance the models until the system is implemented. Unlike the phases of a "waterfall" method, the OMT phases can be re-entered to make changes when required. The following is mostly from [3]. Figure 1 - 1 shows the symbols used in the object diagrams. The object models are reasonably easy to understand and read. Other kinds of diagrams are discussed later in the document.

In the OMT model, the developer has three viewpoints of the development problem, each capturing important aspects of the system.

### 1.6.1 The Object Model

The *object model* describes the structure of objects in a system — their identity, their relationships to other objects, their attributes, and their operations. The object model provides the essential framework into which the dynamic and functional models can be placed. The goal in constructing an object model is to capture those concepts from the real work that are important to an application.

The object model is represented graphically with object diagrams containing object classes. Classes are arranged into hierarchies sharing common structure and behavior and are associated with other classes. Classes define the attribute values carried by each object instance and the operations which each object performs or undergoes.

### 1.6.2 The Functional Model

The *functional model* describes the operations in the system. In particular, it describes how the execution of an operation affects the values of the objects in the system. The functional model consists of operation descriptions and occasional object-oriented data flow diagrams. Object-interaction diagrams can be used to show how an operation works by specifying the messages that flow between objects involved in the operation.

### 1.6.3 Dynamic Model

The *dynamic model* describes those aspects of a system concerned with time and the sequencing of operations — events that mark changes, sequences of events, and states that define the context for events.

The dynamic model is defined by state diagrams, each of which describes the life history of objects of a particular class. Other views can also be used to present dynamic model information. These other views are use cases and scenarios. Scenarios can be viewed in event trace diagrams that show the interactions among a set of objects in temporal order.

### 1.6.4 The OMT Development Process

There are fives steps to the OMT software development process.

**Conceptualization.** Conceive a problem to be solved and a system approach that solves it.

**Analysis.** Describe the behavior of the system as a "black-box" by building OMT models of it in user meaningful terms. Focus on a user-centered description. Avoid making design decisions.

**System Design.** Make the high-level global decisions about the system implementation, including its overall structure.

**Object Design.** Elaborate the analysis models by expanding high-level operations into available operations. When needed make algorithmic and data structure decisions without getting stuck in the details of a particular language. Make design decisions in a language-independent way. Get a logically correct (if inefficient) implementation; then transform the design if needed.

**Final Object Design.** Map the design into a particular language. Coding should be a localized process, as all the global design decisions should have been made already. Methods can be added for convenience or to enhance encapsulation.

### 1.6.5 *Where is the OCS in the OMT Development Process*

There are many parts to the OCS and not all the parts are equally well-developed. The earlier development phases of the OCS were not done in an object-oriented way using OMT, but we feel that overall, the OCS best fits at the Object Design stage, and the documents that are included for the OCS Preliminary Design Review are focused on completing the design. The focus of the OCS PDR design is to indicate interactions between the OCS tracks so that OCS development of the tracks can continue as independently of the other tracks as possible.

This is not all that easy or even possible since there is a logical ordering among the OCS tracks and there is some information from other work packages (as well as our own) that is not yet available. Our approach is to try and isolate any unknowns to a single place in the OCS software system and to push on assuming issues will eventually be resolved or information will become available.

## 1.7 *OCS Physical Design Method*

The figures, diagrams, and tables that make up the OCS OMT design information were created by hand using Frame. Consequently, the information is not as complete or as rigorously verified and checked as a tool-supported effort.

Although we have auditioned a few OMT design tools, we (Shane Walker and Kim Gillies) feel that so far, none of them has worked well enough or provided performance or usability to make them worth their cost. (This is also true of the structured analysis tools we have used.)

We are continuing to review object-oriented design tools and when we find one that works the work in the diagrams in this document will be redone with the tool. It is our intention to use the OO design approach an our chosen tool throughout the detailed design of the infrastructure, applications, and tracks.

### 1.7.1 *OCS PDR Physical Design Products*

Our goal with the physical design is to impart information about how the OCS is to be structured and then to show how the parts of the structure interact with one another to accomplish OCS/Gemini tasks. The following diagrams and information will be used for each of the OMT views in these documents.

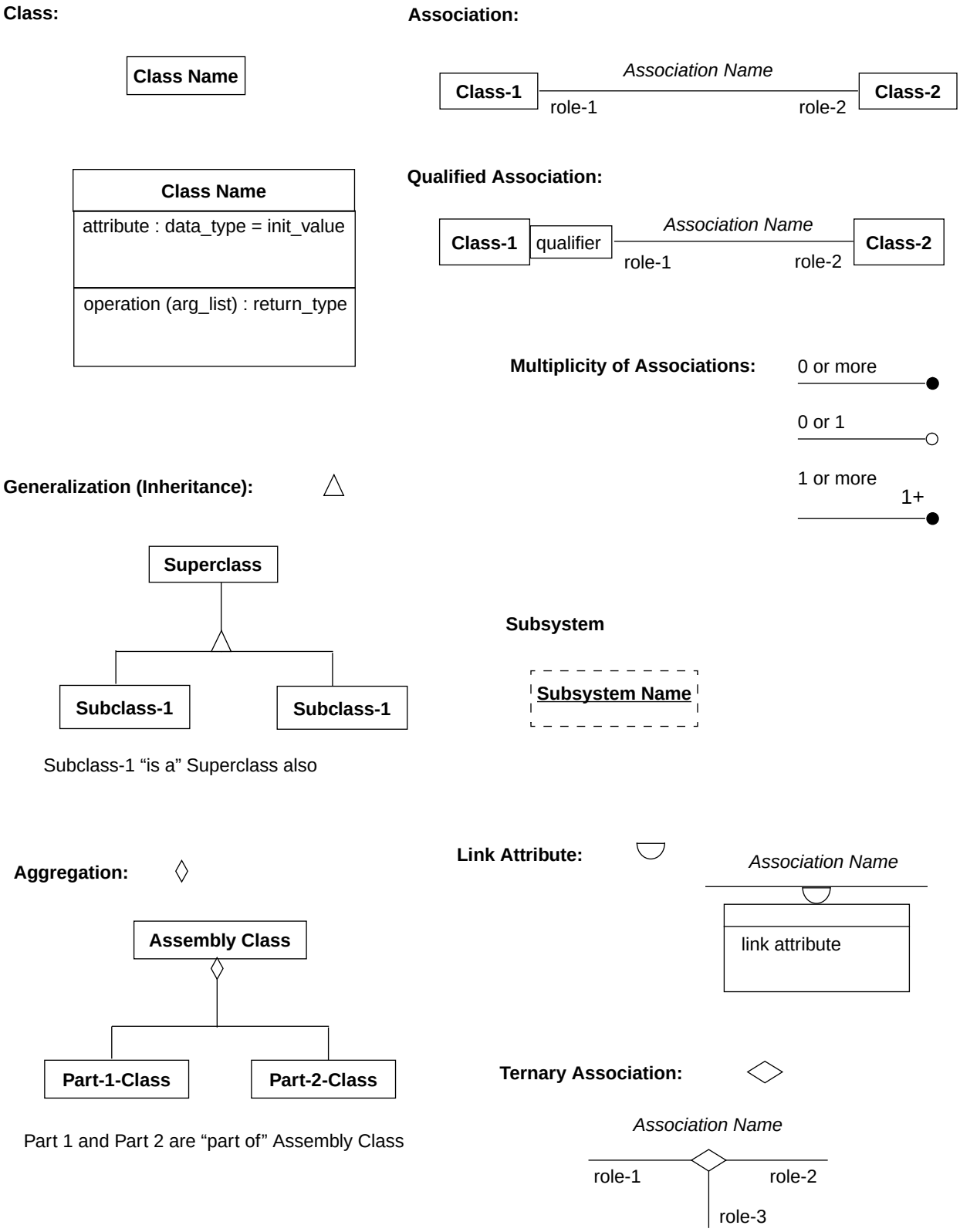
**Object Model.** Object models will be constructed for important parts of the OCS design. These models will show important classes or composite objects when the model is describing an instance of a system.

**Functional Model.** At this phase of the design textual descriptions of methods will be used to describe a system operation in the object model. OO data flow diagrams are too difficult without tools support.

**Dynamic Model.** OO state diagrams are difficult without the help of a tool. We will use *scenario diagrams* (also called event trace diagrams) or *object interaction diagrams*, both derived from use cases, to illustrate the OCS dynamic view.

We believe that at this time in the overall design of the OCS and the GCS as a whole, this level of modeling is appropriate and provides the kind of information that is most important for the users of these documents.

*FIGURE 1 - 1* OMT Symbols



## 1.8 OCS Physical Design Overview

The OCS is a big project with many parts and levels. Parts of the development of the OCS range from the lowest level interactions and protocols to the user interfaces for astronomers and the events and interactions of separate applications within the OCS.

No single model can be used for the entire OCS. Some development tracks will be focused on creating user interfaces (Telescope Control Console track, Observing Tool track) and others focus on creating the OCS environment that allows the applications to cooperatively achieve the acquisition of data during the observing processes.

The input to this stage of the development process is the set of requirements reviewed during the OCS SDR process, user consultations since the SDR, and the original Gemini Software SDD [1].

We have broken the physical design of the OCS into two parts. The first part is the creation of the design of a single OCS application (what we call an OCSApp). All applications within the OCS are instances of OCSApp and each instance can use the basic functionality and capabilities that are provided by the OCSApp.

Once the capabilities of a single OCSApp are known, it is possible to model the interactions of multiple OCSApp instances to provide the overall OCS functionality. It is vitally important to understand the layers of the OCS design and their focus so that at each layer the details of the previous layers can be ignored. A summary of the steps follows. Figure 1 - 2 shows a little more detail for each step.

1. Design the basic object structure and functionality of a single OCS application.
2. Specialize the basic OCS application to design the important kinds of applications that appear in the OCS software system.
3. Use the OCS applications opaquely to design the kinds of functionality and to demonstrate the interactions that the OCS software system must have as a whole.

*FIGURE 1 - 2* OCS Physical Design Overview

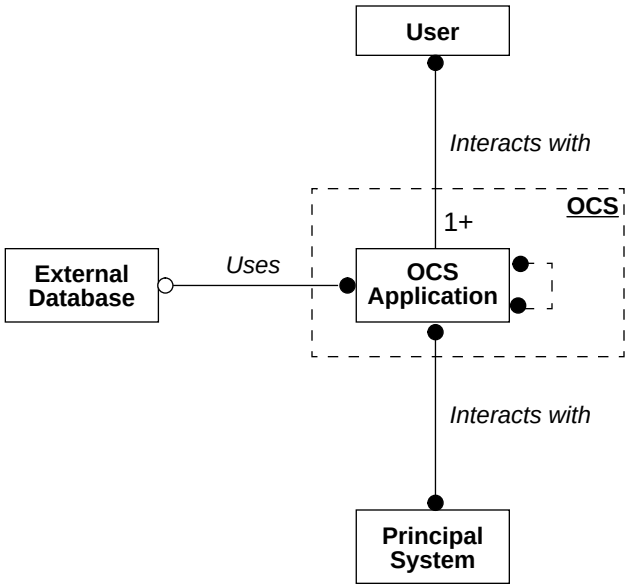
| | |
|---|---|
| ***OCS Application Design*** | *Design the basic OCS application*<br>*Taking into account OCS requirements:*<br>*1. Access control*<br>*2. Communications*<br>*3. Logging* |
| ***OCS Application Specification*** | *Design a few OCS application types showing how each uses the basic OCS application functionality.*<br>*1. GUI Applications/Consoles*<br>*2. Script Executors*<br>*3. Filter Applications* |
| ***OCS Track Designs*** | *Design tracks as sets of OCS applications showing how track products interact to produce OCS functionality.*<br>*1. Console-based observing*<br>*2. Planned observing*<br>*3. Multi-instrument observing/plans* |

## 1.9 OCS Context Diagram

The context diagram shows what is in the OCS and what isn't by identifying external systems. The context diagram for the OCS, based upon the OCS SDR documents, is shown in the following figure using OMT. The OCS is a composite object to be refined during the OCS development.

This diagram models the OCS as high-level composite objects and classes. Zero or more Users are interacting with one or more OCS applications. The OCS applications can interact with an external database and the other peer principal systems. The other principal systems also interact with the OCS.

*FIGURE 1 - 3* The OCS Context Diagram



# 2 OCS Application Model

*The chapter describes the model of a single OCS application—the basis of all OCS applications.*

## 2.1 Introduction

At the highest level, we view the Observatory Control System as a collection of interacting applications. Each application can interact with other OCS applications as well as the external principal systems and external database. Before the OCS can be modeled as a whole, a single application must be modeled. The resultant application model is a building block that is to be used to structure the entire OCS.

The preliminary design focuses on the interfaces between OCS applications and between OCS applications and other principal systems. Its other important function is to help partition the required work and the scope of the work between the OCS development tracks.

## 2.2 Scope of the OCS Application Model

There are several types of OCS applications in the prototype OCS of the SDD [1]. The types of applications are described here briefly. Some of the category names are new to the OCS design.

**Consoles.** A console is an application that presents a "control panel" for one or more systems or subsystems. The control panel is created using a Graphical User Interface (GUI). The console allows commands to be sent to the systems or subsystems and can display raw status information from the subystems or it can synthesize and display higher level status information based upon the raw status information. Consoles are the primary tool for interactive observing.

**Script Executors.** Some applications execute scripts to send commands to systems and use status information. For our purposes, a *script* is a textual description of operations to be performed by the OCS. Most script executors do not have an associated GUI, but that is not a requirement.

**Service.** A service is a software component in the OCS that provides functionality to other clients applications in the OCS. A service can be provided by a dedicated, stand-alone application or it can be a set of public operations provided by any OCS application (such as a console). Some services have user interfaces and some do not.

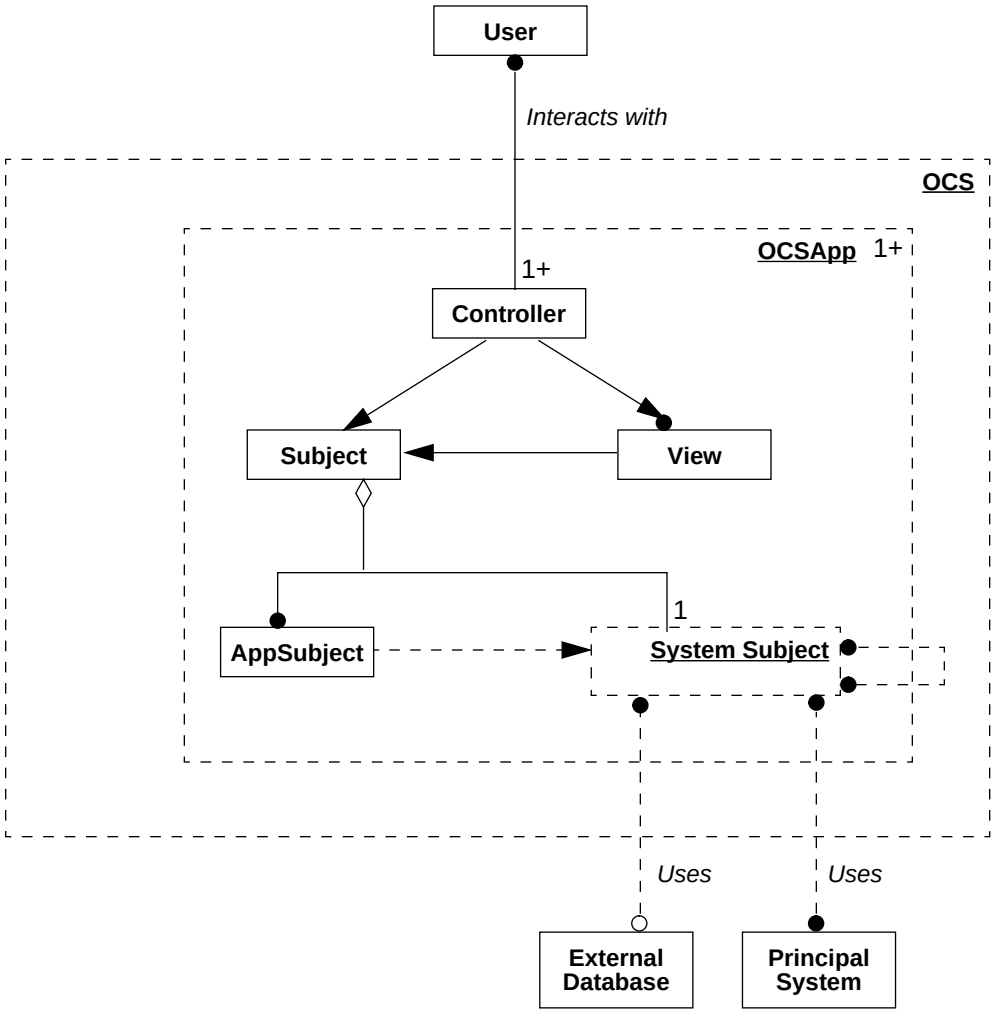
**Agent Applications.** An agent is a service application that represents another software service or hardware system inside the OCS software system. The agents function is to isolate the OCS from the peculiarities of another system. An agent can provide a translation function that allows OCS applications to communicate with systems or services outside the OCS in a standard way. In a client-server model, the agent often operates as the server for a system or subsystem for the OCS clients.

## 2.3 OMT Model of an OCS Application

The model for an application in the OCS must be capable of supporting all the types of applications that will be required for the OCS. It must do this without requiring that all features be present in every instance of the application.

The model for an OCS application is an OMT subsystem called OCSApp. A subsystem is a high-level subset of the entire model. (The OCS and OCSApp are subsystems in the larger Gemini Control Software system.) The OCSApp is an organizational entity; it consists of classes and subsystems that work together to provide an architecture for applications. A subsystem is a collection of modeling elements such as classes or associations. The following figure shows the high-level OCSApp classes and subsystems.

FIGURE 2 - 1 OCSApp Subsystems and Classes




An instance of OCSApp is an application that creates the objects and associations of OCSApp. This model shows the entire OCS is made up of one or more OCSApp instances. Zero or more users can interact with one or more OCSApps. Each OCSApp can communicate with zero or more other OCSApp instances. A single OCSApp can use the services of up to one external database (set by the GCS system design) or zero or more principal systems. The principal systems can also use (communicate with might be better) zero or more OCSApps.

## 2.4 Model Part Descriptions

The classes and associations of the OCSApp are designed to provide a structure for building instances of OCSApp. This section describes the role of each of the classes in OCSApp.

### 2.4.1 Subject-View-Controller Architecture

User interface applications seem as though they should be easy to build, but if they are poorly designed they can introduce enormous interdependencies that make maintenance difficult or impossible. It is a goal of the OCS to provide user interfaces that can be modified by the operations staff as simply as possible so it is important to design all OCSApps from day one to support that goal.

A good design for a user application separates the user interface from the rest of the system by dividing an application into a number of loosely coupled parts. The Model-View-Controller framework was developed in the Smalltalk world to provide an object-oriented framework that separates the user interface and the domain-specific application code. The architecture of an OCSApp uses this framework but calls it Subject-View-Controller (SVC) since the term model is used for other things in our papers. This framework is discussed fully in [4]. The idea is to separate the underlying information of a problem (subject) from the various ways of presenting the information to the user (the view). The interactive aspect of the problem (the controller) is distinguished from the relationships between the subject and its views.

The important feature of the architecture is shown by the associations between the three classes.Subjects, at the lowest level of the application, contain the fundamental application information and the operations used to manipulate the information. Views are aware of the subjects. The Controller converts user inputs into operations on views and subjects so controllers are designed last. Because of the one-way dependencies, it should be possible to build applications one layer at a time. This is why we have chosen an SVC application architecture.

#### 2.4.1.1 AppSubjects

When building an application the AppSubject classes are designed first to include the basic application classes and operations. This is where the functionality that makes an application unique is found in the OCSApp architecture. For instance, in the Observing Tool, the fundamental AppSubject data are the Science Programs, Plans, Observations, and Configuration Components. The OT provides methods that make it possible to manipulate the OT AppSubjects. The AppSubject may contain many kinds of data and operations.

#### 2.4.1.2 SystemSubject

The subjects for OCS applications often come from outside of an individual OCSApp. For instance, a console application might wish to display status values for a TCS subsystem. This *external subject* is provided by the functionality of the SystemSubject subsystem. The SystemSubject subsystem is common to all OCSApp instances. It is designed and built to provide standard functionality and subject information for all OCS applications.

The SystemSubject provides the capabilities common to all OCSApp instances such as access to principal systems for status and commanding as well as the ability to communicate with other OCSApp instances. Classes in the SystemSubject standardize the interface to databases. The SystemSubject architecture is described in the Interactive Observing Infrastructure track documents.

#### 2.4.1.3 Views

A view is an external format for presenting or visualizing subject information. There may be many ways to view subject information so a View is a class. Views can be textual or visual. Views in the Observing Tool might show the Observations in a Science Program as a textual list or as icons or another view would show the attributes that

describe the Science Program itself. Views need to know about subject data (but not subject operations) but subjects do not know anything about views. A single subject may be mapped to one or more views. Views must be kept consistent with the changes to subject data.

#### 2.4.1.4 *Controllers*

Generally users control applications by interacting with views. Input events, such as button clicks and keyboard entries are mapped by the controller into operations that are applied to the views and subjects. The controller is the surrogate for the user (the actor). For most situations the controller acts and represents the user and his requests in the operation of the application. A controller can be implemented as a GUI screen (as in a console) or as a script (a script executor).

## 2.5 *OCSApp Model Use*

What good is the OCSApp model?

- First of all it shows how we intend to structure *every* OCS application. This approach of having a single architecture for applications and a shared "library" of functionality (the SystemSubject) is common. Examples are X-Window Xt programs where the Xt library is the SystemSubject. Another example is Adam/Drama where the "fixed part" is the SystemSubject.
- The OCSApp only indicates application associations but no actual operations so it is only organizational. However it shows where in an OCSApp specific functionality is placed. This should make maintenance easier for the future operations folks.
- Using the OCSApp should make it easier to make changes to applications and the SystemSubject since the roles of the pieces are defined.
- The OCSApp is an abstraction of a real applications. The designs for individual OCSApps can now approached as the design of AppSubjects, Views, and Controllers rather than entirely new applications.

## 2.6 *What Next?*

The next chapter shows the preliminary object diagrams for the SystemSubject subsystem.

# 3 The System Subject Model

*This chapter describes the object model for the SystemSubject of an OCSApp*

## 3.1 Introduction

The SystemSubject is a subsystem within the OCSApp subsystem. Its role is to provide access to subject data when the data is external to the application itself. Examples of external subject data are Science Programs or principal system status information. This chapter shows what objects and associates we consider to be part of the SystemSubject; thence, it shows the functionality that will be present in the SystemSubject.

This chapter shows the physical model for the SystemSubject. The instance of the SystemSubject that is to be created from this model is called the Command Layer Library. Because of the importance of the Command Layer Library it is more developed than other parts of the design. The document *OCS Interactive Observing Infrastructure Track PD* is a companion document to this chapter. That document describes details of the interactions between principal systems which this document assumes. This document is not a detailed design of the Command Layer Library. Instead it focuses on high-level design issues and interactions with the OCSApp model.

The model for the SystemSubject has been arrived at by examining the SDD [1], the OCS Software Requirements Document [5], and the overall design of the OCS.

## 3.2 Scope of the SystemSubject

The fundamental job of the SystemSubject is to provide an instance of OCSApp with access to subject data that is outside of its boundaries. More specifically the following items can be extracted from the documents.

- OCSApps need to access status information from other applications and principal systems.
- OCSApps need to command other applications and principal systems to take actions on their behalf.
- OCSApps need to control access to their own operations and data. They also need access to other OCSApps to allow the first two bullets.
- OCSApps need access to external systems. Currently the only external system is a site database.

This is not as simple as it appears since OCSApps must communicate with a variety of kinds of systems using a variety of protocols. Quite a bit of work has been done on modeling communications and distributed control (for instance industry standard CORBA). Within the EPICS community, CEBAF has produced a package called *cdev* a library to provide a standard interface to one or more control packages or systems [7]. The preliminary methods for some SystemSubject objects come from this paper.

At this time, plans for the OCS are to use an available package or product that meets some or all the functional requirements of the OCS project as modeled in this chapter. That package is not known at this time.

The SystemSubject must also provide other functions that are required and that should be shared by all OCSApps.

- All instances of OCSApp must be able to log messages using the DHS logging system or a disk file.

## 3.3 A Description of the SystemSubject Model

Figure 3 - 1 and Figure 3 - 2 shows the preliminary object model for the SystemSubject. The first figure shows the low-level portion of the SystemSubject that is responsible for communications with the parts of the GCS. The second figure shows the high-level objects that are visible to the OCSApp developers.

### 3.3.1 SystemSubject Communications Subsystem

The SystemSubject Communications Subsystem is designed to present a uniform interface (set of methods) for all the different communication methods and protocols that might be used in the system. The communications subsystem common functionality is modeled upon the message bus concept used in Sun's Tooltalk or KoalaTalk ([9], [10]). The EPICS system uses a message-bus idea to distribute updates (monitors). Having a common message paradigm in the OCS and the real-time systems seems like a good thing. First some definitions of two fundamental classes.

**Name.** A name is a token for some system, subsystem, or device that is located somewhere in the GCS software world.

**Service.** A service is a communications service that is used to communicate with something associated with *name*. This might be a specific protocol or message system unique to the server of "name".

Figure 3 - 1 shows that a name is associated with at most one **Service**. **Service** is an abstract class that describes the generic interface for all services. It is abstract because there must be subclasses of it for each of the kinds of systems where names can reside.

The **ServiceManager** is a singleton class; it is a collection of **Service** instances. Its function is to locate, load and generally keep track of services. To do this it uses a **ServiceNameMap,** which maps names to service names, and the **ServiceMap**, which maps service names to **Service** instances. The figure shows five instances of **Service,** one for each of the communications methods that are currently known to exist.

Note that instances of **Service** are loadable by the **ServiceManager**. An OCSApp will only attach to the communications services that it needs for its work. The following are the known services. **Service** also contains the methods which instances must override to provide the "generic" interface.

**EpicsService.** This service provides a thin layer over "raw" EPICS Channel Access. Raw is added to differentiate this from the more elaborate CAD protocol. It would be used for status and alarm monitoring.

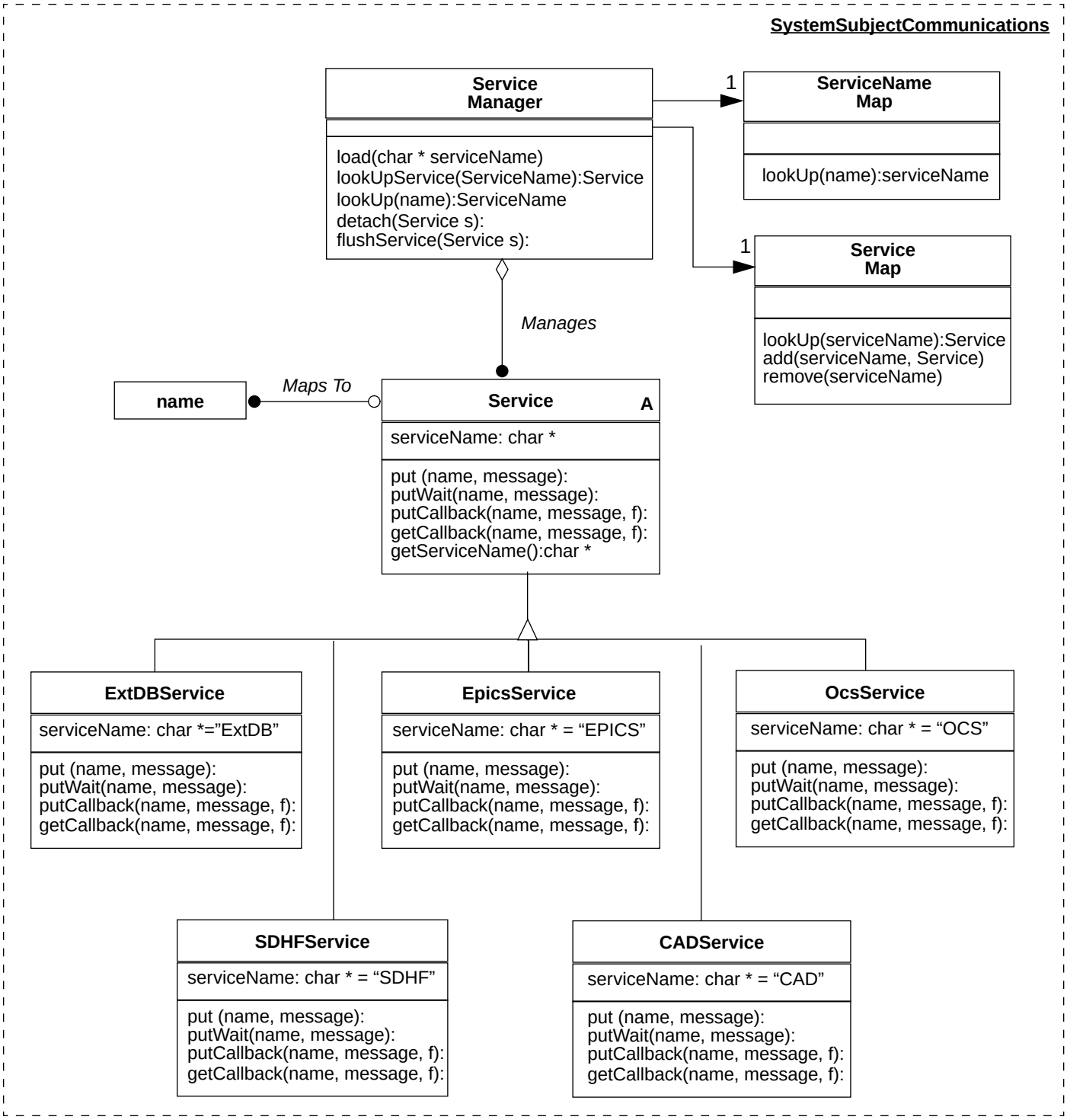
**CADService.** The CADService is part of the communication link to the EPICS/Cad-based principal systems. Because of the need to control access to the principal systems, the CAD communication is implemented partially in the CADService and partially as an agent process.

**ExtDBService.** This will encapsulate any special protocol that might be required to communicate with the external database.

**SDRFService.** This protocol may be required to allow OCSApp to communicate with the Synchronous Data Reduction Facility.

**OCSService.** This is the message bus service that will be used by OCSApp instances to "talk among themselves".

FIGURE 3 - 1 The SystemSubject Communications Subsystem Model




### 3.3.2 *SystemSubject Communications Subsystem Functional Model*

The following tables describe briefly what each of the methods in the SystemSubjectCommunications subsystem is to do.

TABLE 3 - 1 Service Manager Methods

| Method | Use |
|---|---|
| load(char *serviceName):Service s | The Service Manager dynamically loads the requested service if it isn't already loaded. It makes all the internal connections needed to allow data to be sent to and received from a "name" using serviceName. The newly loaded Service object is returned. |
| lookUp(char *name): char *ServiceName | The ServiceManager consults the ServiceNameMap to find out what ServiceName to use to access "name". That ServiceName object is then mapped by ServiceManager to an instance of Service to be used to communicate with the "name". lookUp returns NULL if there is no serviceName associated with the "name" is not loaded. |
| lookUpService(char *serviceName): Service s | The ServiceManager consults the ServiceMap to get access to Service instance with "serviceName". lookUpService returns NULL if the "serviceName" doesn't associate with anything. |
| remove(char *serviceName) | The ServiceManager removes any knowledge of "serviceName" from the SystemSubject. |
| flush(char *serviceName) | Complete any incomplete operations that are using "serviceName.". |

TABLE 3 - 2 ServiceName Map Methods

| Method | Use |
|---|---|
| lookUp(char *name); char *serviceName | Attempt to map the "name" to a serviceName. Return the serviceName or Null if no mapping is found. |

TABLE 3 - 3 Service Map Methods

| Method | Use |
|---|---|
| lookUp(char *serviceName): Service s | Lookup any Service associated with "serviceName". Return NULL if no Service exists. |
| add(char *serviceName, Service s) | Add a mapping between "serviceName" and the Service object s. |
| remove(char *serviceName): | Remove any mapping present for "serviceName". |

TABLE 3 - 4 Service Methods

| Method | Use |
|---|---|
| put(name, message) | This method will use the Service to deliver a message associated with the name "name". The call will return immediately. |
| putWait(name, message) | This method will use the Service to deliver a message associated with the name "name". The call will block waiting until a response has been received for the message. |
| putCallback(name, message, f) | This method will use the Service to deliver a message associated with the name "name". The call will return immediately and will associate the callback function f with a response to the message. |

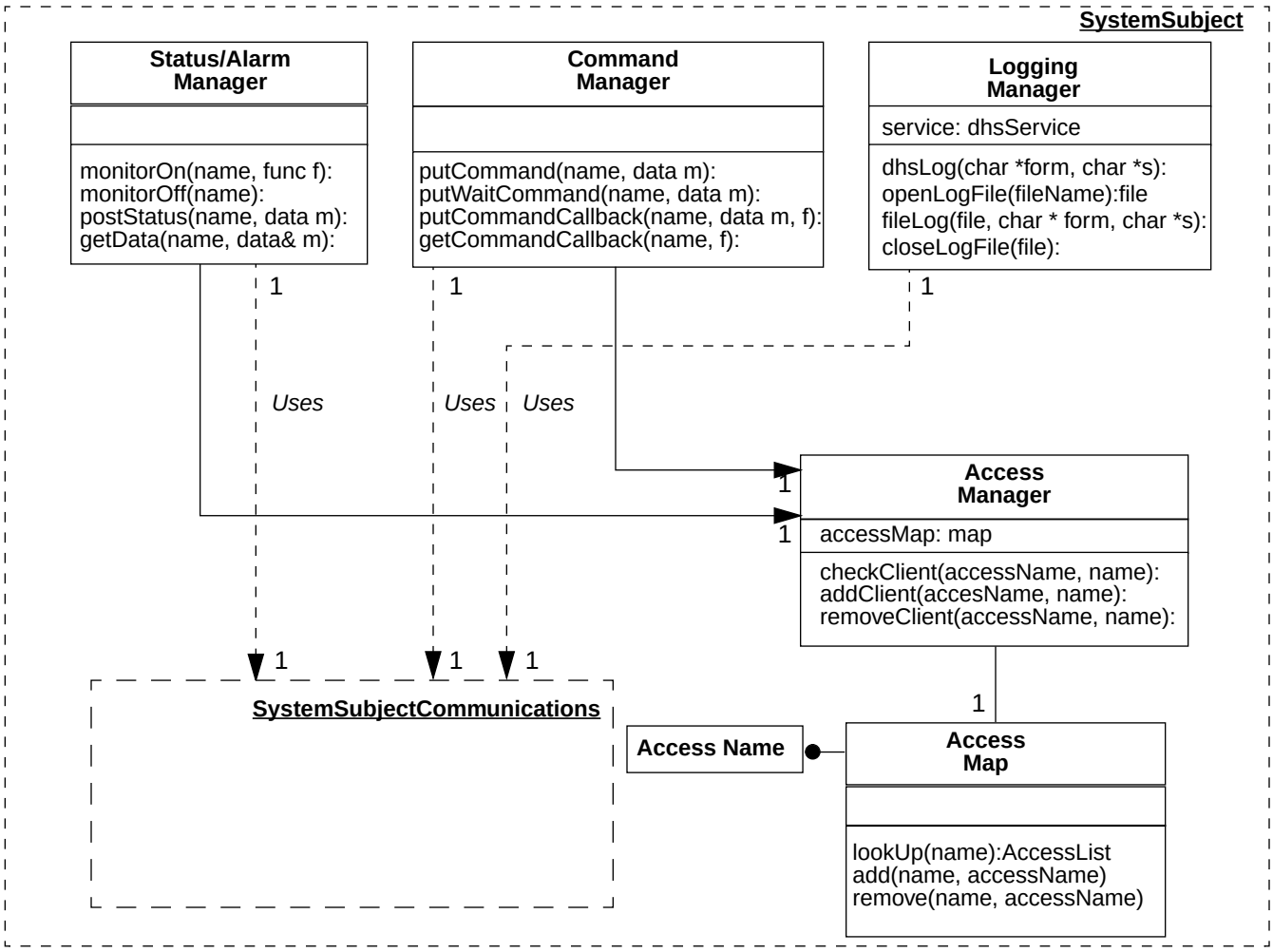
| Method | Use |
|---|---|
| getCallback(name, message, f) | This method will associate a callback function f with the reception of a message on the Service with “name”. This is the way systems mark an interest in receiving commands. |
| getServiceName(): char *serviceName | This method will return the value of the serviceName attribute. |

### 3.3.3 *High-Level SystemSubject Object Model*

The high-level object model presents the methods used to access the high-level functionality of the SystemSubject. Whereas the methods of the previous section are private to the SystemSubject, the methods in these classes are used by the other classes of an OCSApp instance to access the rest of the GCS.

The SystemSubject Communications subsystem methods are designed to support the presentation in the SystemSubject high-level objects of a uniform interface (set of methods) for all the different communication methods and protocols that might be used in the system. Figure 3 - 2 shows the classes that make up the high-level SystemSubject. (In the figure the dashed arrows are used to indicate that the direct association between classes is not shown since the SystemSubjectCommunications subsystem has not been expanded.)

*FIGURE 3 - 2* High Level SystemSubject Object Model



There are four parts of the high-level SystemSubject model. Each is related to a primary function of the Command Layer Library, which is what the instance of the SystemSubject is called in the IOI track.

**Status/Alarm Manager.** Status and alarms are presented at the highest level through a “monitor” approach. Every system “name” must provide its status and alarms asynchronously when a status item changes or an alarm state occurs. Clients of status and alarms do not poll. The source systems must post their status when they have information that others need in a timely fashion.

The Status/Alarm Manager provides the methods for monitoring status and posting status.

**Command Manager.** OCSApp instances can send commands or receive commands. The CommandManager provides the ways to send a command and one way to respond to commands.

**Logging Manager.** The SystemSubject must support logging to the DHS and to a local file. These functions are handled by the Logging Manager.

**Access Name.** Every message that flows in the OCS will have an associated AccessName that describes the identity of the sender of the message. Message objects and the association between Messages and Access Names are not shown in the object diagrams at this phase of the design.

**Access Manager.** Every OCSApp can control access to its commands. This function is handled by the AccessManager. The Access Manager tests and manages the relationships between “names” and AccessNames.

**Access Map.** This object used by the Access Manager provides the association of AccessNames with “names”. This is the data store used by the Access Manager.

Each OCSApp SystemSubject has a single instance of **CommandManager, Status/Alarm Manager, Access Manager, and Logging Manager.** All these classes depend upon and use the SystemSubjectCommunications subsystem described in the previous sections.

The interactions between the SystemSubject and the SystemSubjectCommunications subsystem occur through the use of a “name”. When a system is monitored or a command is attempted, the appropriate Service instance is loaded by the ServiceManager.

### 3.3.4 High-Level SystemSubject Functional Model

The following tables describe briefly what each of the methods in the high-level classes is to do.

*TABLE 3 - 5* Status/Alarm Manager Methods

| Method | Use |
|---|---|
| monitorOn(name, f) | This method indicates that the function f should be called whenever status or alarm information arrives which is associated with name “name”. Only one callback can be associated with a “name”. |
| monitorOff(name) | This method removes any monitor callback associated with name “name”. |
| postStatus(name, data m) | As a producer of status or alarms, an OCSApp uses this method to post status or alarms to the system. The information is associated with name “name”. |
| getData(name, data &m) | This method is used by an OCSApp to ask for any data associated with name “name”. Although.OCSApp instances must keep clients updated with the latest status/alarms values, sometimes it is also useful to fetch them. |

TABLE 3 - 6 Command Manager Methods

| Method | Use |
|---|---|
| putCommand(name, data m): boolean | This method will cause a command containing data m and associated with name “name” to be sent from the OCSApp instance. The method will return immediately indicating whether the command was sent properly. Any returned reply is thrown away. |
| putWaitCommand(name, data m) | This method issues a command like putCommand, but this method blocks until the actions associated with the command are completed. It returns any reply received. |
| putCommandCallback(name, data m, f) | This method sends a command like putCommand, but in this case the method returns immediately and when the actions associated with the command is complete, the function f is called in the source OCSApp with any reply. |
| getCommandCallback(name, f) | This method is used by an OCSApp to associate a name with a function that will be called whenever a command arrives at the OCSApp that is directed towards “name”. |

TABLE 3 - 7 Logging Manager Methods

| Method | Use |
|---|---|
| dhsLog(char *form, varargs args) | This method is used to write a log message to the DHS log service. The format is like Unix “printf”. |
| openLogFile(char *fileName): file | This method supports the local logging facility. This method opens a log file and appends future log messages. |
| fileLog(file f, char *form, varargs args) | This method will log messages to file f, which has been previously opened with openLogFile |
| closeLogFile(file f) | This method with close a log file. |

TABLE 3 - 8 Access Manager Methods

| Method | Use |
|---|---|
| checkClient(accessName): boolean | An OCSApp as a server of functionality uses this method to test whether or not an accessName is currently allowed to access a “name.” |
| addClient(accessName, name) | This method is used to add a specific accessName to the access list associated with “name”. |
| removeClient(accessName, name) | This method will remove the “accessName” from the access list associated with “name”. |

TABLE 3 - 9 Access Map Methods (a private class)

| Method | Use |
|---|---|
| lookUp(name): AccessList | This method will return the access list of AccessNames that have been previously associated with “name”. |

| Method | Use |
|---|---|
| add(name, accessName) | This method will add accessName to the list associated with “name”. |
| remove(name, accessName) | This method will remove “accessName” from the access list associated with “name”. |

### 3.3.5 SystemSubject Functional Model

The object interaction diagram is used to show control and operational behavior of an object system by showing the sequence of messages that implements an operation. It only includes the objects relevant to an operation. It shows objects that exist before the operation and ones that are created during the operation. Pre-existing objects and links are shown with solid lines and newly created objects and links are shown with a fill pattern and dashed lines. The control flow follows data links, so an object diagram contains all the paths that control can follow. Transient links that are links only during the operation are drawn in dotted lines. A message from one object to another is indicated by a label consisting of a text string with an arrow showing the message flow direction; the label is drawn next to the link that is used to send the message. The label contains the following elements (some are optional):

**Sequence number.** - the numbers show the nested calling sequence in “Dewey Decimal” notation (2.1.4). We don’t have any examples that are nested.

**An iteration indicator.** This is an *, optionally followed by an interaction expression in parentheses (such as (i=1...n).

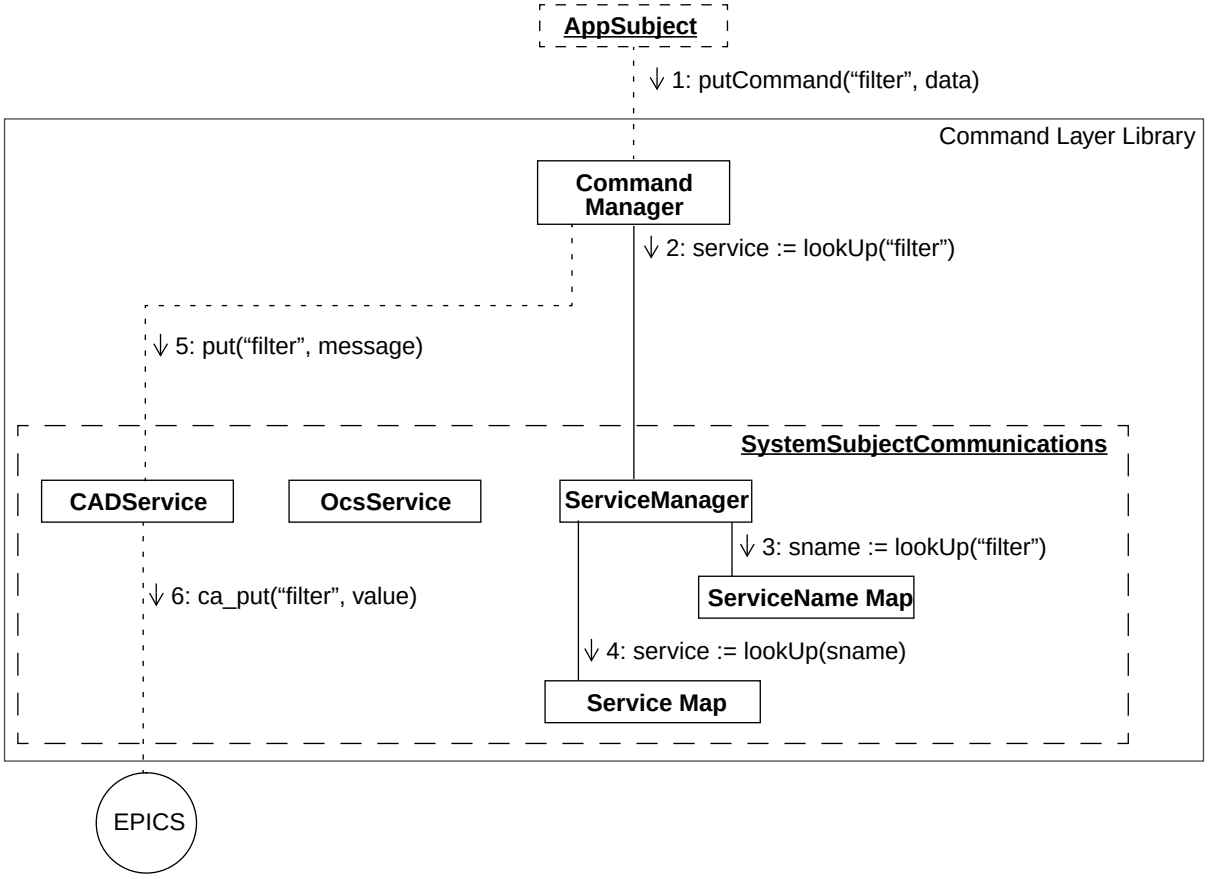
**A return value.** A name followed by an := assignment sign. The names used as return values can be used in other messages.

**Message name.** The message name or method name is used. The class isn’t needed since it is clear from the diagram which object the message is going to.

**Argument list.** The argument list contains values or names representing values.

The basic dynamic operation of the SystemSubject command subsystem can be shown with an object interaction diagram. The first case (Figure 3 - 3) shows how a command is sent to a system when the service is already available.

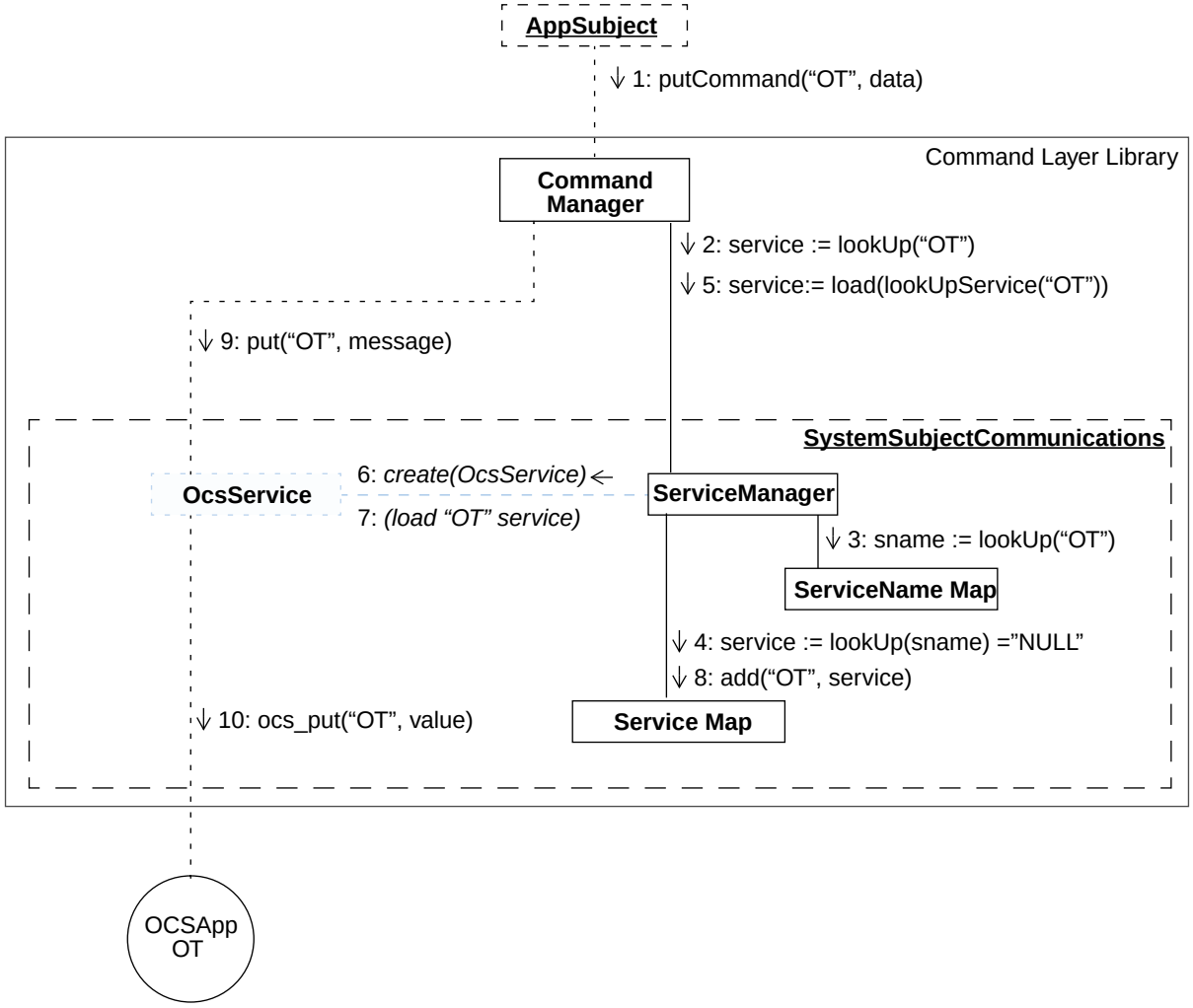
*FIGURE 3 - 3* Sending a Command to a “filter” mapped to an already loaded service.




The AppSubject of an OCSApp puts a command to a “filter” name. The instance of the SystemSubject, the Command Layer Library, looks up “filter” in the ServiceName map, finds it and returns the Service to the Command Manager. The Command Manager then uses the returned Service, the CADService, to put the message. The CADService translates the request into EPICS-specific calls, in this case ca_put.

The case in Figure 3 - 4 is a little more complicated. In this case, a message is being sent from one OCSApp to something name “OT”. The Command Layer Library attempts to map the name to a Service and fails requiring that the library be loaded.

*FIGURE 3 - 4* Sending a Command to a name “OT” when the service isn’t loaded.




The initial lookup at step 4 fails and returns NULL. The Command Manager looks up the Service associated with “OT”, finds one, and loads it. The newly created OcsService is shown in a grey pattern to indicate it is loaded during the execution of the object interaction diagram scenario. Once the OcsService is loaded, it is added to the ServiceName map and the Service is returned. The message then goes out as in the previous example, this time being translated into the OCS Message System in the OCSService.

## 3.4 *SystemSubject Summary*

This chapter has shown how the Command Layer Library, the instance of the SystemSubject the OCS group will create, will be structured and how some of its functionality will operate. The creation of the SystemSubject is the job of the Interactive Observing Infrastructure Track. More information on the IOI track design and the design of the “CADService” are given there.

The next level of OCS design is the creation of individual applications using the design for OCSApp and the functionality of the SystemSubject. The next chapter will show a few examples of typical OCSApp instances.

# 4 OCSApp Instances

*This chapter contains additional information on designing an instance of OCSApp*

## 4.1 Introduction

The OCS is made up of several OCSApp instances. The functionality required by every OCSApp is encapsulated in the SystemSubject. The single instance of SystemSubject is the Command Layer Library that is to be created during the Interactive Observing Infrastructure track and supplemented during the Telescope Console Control track.

Chapter 2 described the overall model for OCSApp and the motivation for having a single application model. This chapter will examine some of the common types of applications in the OCS and show how they relate to the OCSApp model.

## 4.2 The OCSApp Model

OCSApp is a subsystem, not a class. A *subsystem* is just a name for a group of classes and associations; therefore, it is not really fair to speak of *instances* of OCSApp. At runtime, an OCSApp is a composite object, which is made up of the many objects that are included in the Controller, View, and AppSubject subsystems. Most of the time the distinction is not important and OCSApp instance is used.

Figure 2 - 1 is shown again in Figure 4 - 1. Chapter 3 discussed the design for the SystemSubject. The associations from the SystemSubject are now known to be those associated with the Service instances a particular OCSApp uses. Building an OCS application consists of designing and adding the following subsystems when they are appropriate.

- AppSubjects

  The AppSubject contains the classes that make an application unique. This includes any objects and methods or algorithms that make the AppSubject data useful for the layers of the application above the AppSubject.

- Views

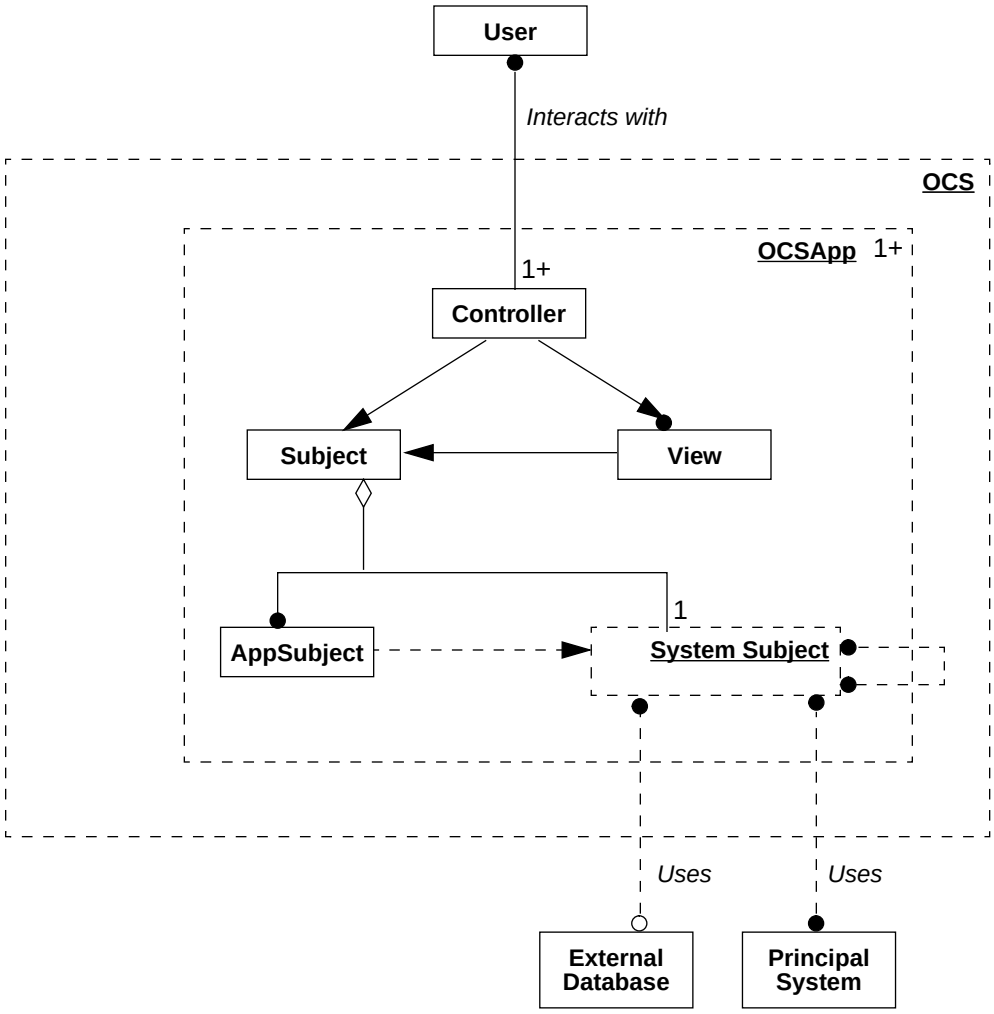
  The views present the AppSubject data for the application. There may be many ways to view subject information. Views can be textual or visual. Views must be kept consistent with the changes to subject data.

- Controllers

  Generally users control applications by interacting with views. The controller is the surrogate for the user (the actor). For most situations the controller acts and represents the user and his requests in the operation of the application. A controller can be implemented as a GUI screen (as in a console) or as a script (a script executor).

  The next sections show what parts are present in some OCS application types.

FIGURE 4 - 1 OCSApp Subsystems and Classes



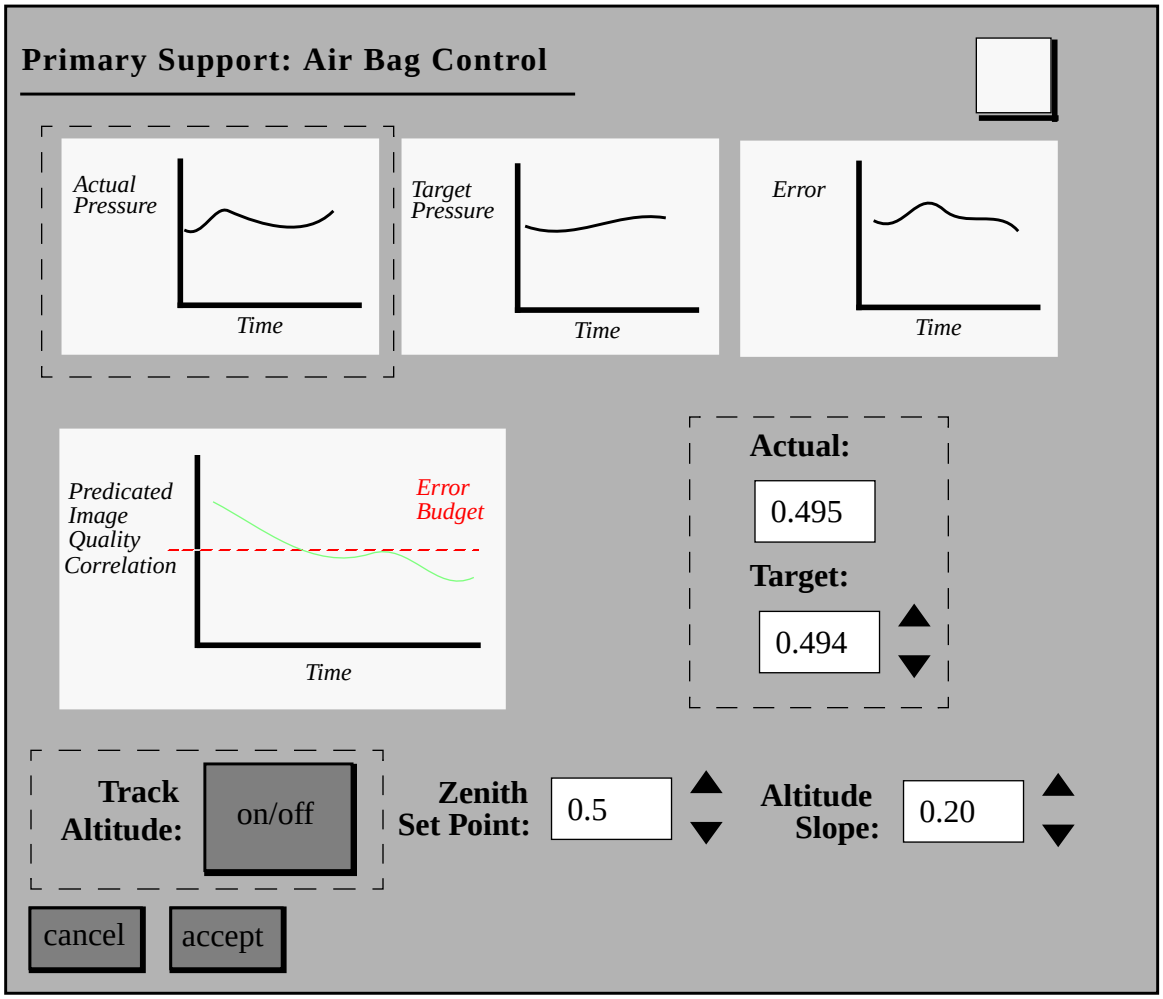

## 4.3 OCS Application Console Instances

This section will examine the design of a typical single system console. The design is very high-level. The actual design of the consoles is part of future OCS development tracks.

A prototype console from the SDD that is to be used for controlling the primary mirror air bag is shown Figure 4 - 2. The screen contains a number of GUI widgets including a strip chart widget for displaying values in time, a two position button for turning tracking on and off, and editable text entries for setting some parameter values. The drag-and-drop box and the cancel and accept buttons are ignored here because they don't add any new problems to the design of an application. One case of each of the different kinds of items is modeled here. (Shown with dashed boxes.)

A console has all the parts of an OCSApp: an AppSubject, Views, and a Controller.

FIGURE 4 - 2 An example OCS console




### 4.3.1 *The Air Bag Subject*

The boxed items in Figure 4 - 2 point out the console subject data including: the actual air bag pressure, the target air bag pressure, and the track altitude control state. Some of the data is status (actual pressure, current track control state) and some is related to controlling the airbag (target values).

The status values from the TCS air bag subsystem are made available to the console through the functionality of the Command Layer Library.

### 4.3.2 *The Air Bag Views*

Within the SVC design model, a widget can be viewed as a “black box” that provides a view and sometimes it also acts as a controller by generating input events (when a button is pushed). Status or read-only widgets provide a view, but widgets like the Target air pressure widget and the Track Altitude button include both a view and a controller to interact with the view.

There are no views other than widget views in this example console. Other applications might draw a figure (for instance, the graphical representation of the shape of the primary) or present the subject data in a way not supported directly by a primitive widget type.

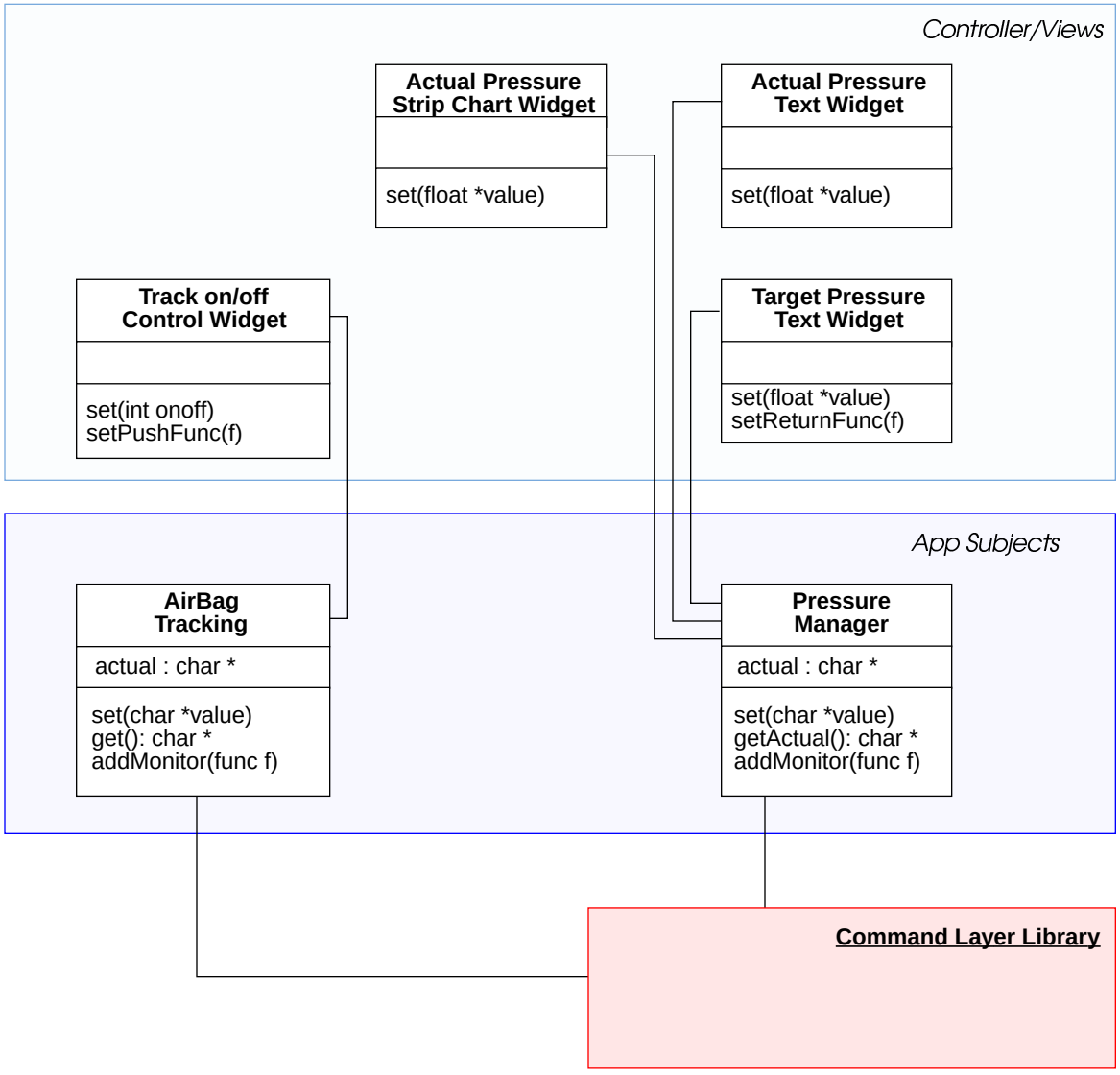
### 4.3.3 The Airbag Controller

The surrogate for the user in the OCSApp model is the Controller. In this application the only controllers are the two control widgets on the screen. The user enters text in the Target pressure box or pushes the Track Altitude button and the widget generates events that operate through callbacks on the AppSubject and Views. More elaborate applications may have controllers other than widgets.

### 4.3.4 The Air Bag Object Model

A simple object model for the Air Bag console and the boxed widgets discussed is shown in Figure 4 - 3.

FIGURE 4 - 3 Air Bag Console Object Model




The widgets comprise the Controller and Views for the application (shown in the shaded area at the top of the figure). The AppSubjects, in the next lower shaded box, include two objects called AirBag Tracking and Pressure Manager. These objects would be subclassed from some class of generic Subjects (they have the same methods) that have the job of interfacing GUI widgets to the lower-level functionality of the Command Layer Library. The

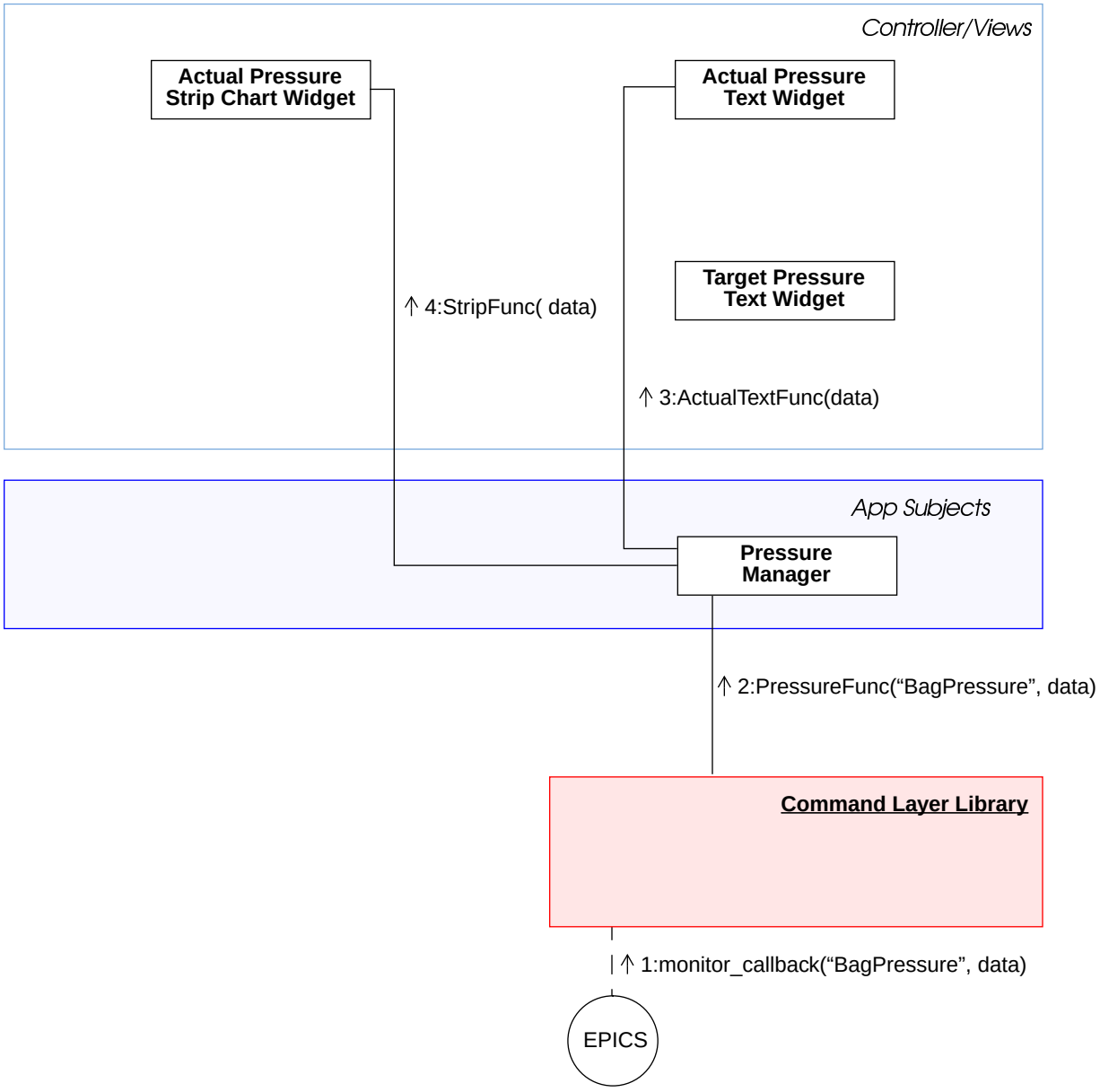
creation of these kinds of AppSubject classes is the role of the Telescope Control Console library, a product of the Telescope Console Track. (See [11].)

The lowest level in the object diagram shows the Command Layer Library where the SystemSubject data that originates in the TCS principal system is located. The functionality in the Command Layer Library is used to update the AppSubjects which in turn update the GUI items.

### 4.3.5 *Parts of the Air Bag Console Dynamic Model*

A few object interaction diagrams will show the important interactions among the components of a console. The example of Figure 4 - 4 shows how an EPICS record update for the BagPressure flows up through the console.

*FIGURE 4 - 4* TCS Updates the “BagPressure” SIR Record



The EPICS monitor features causes a monitor callback for the BagPressure which the Command Layer Library responds to by calling the function the Pressure Manager has registered with the BagPressure name. The Pressure Manager then updates all the widgets that have registered a function for actual pressure updates.

*FIGURE 4 - 5* Setting the BagPressure from the Text Widget

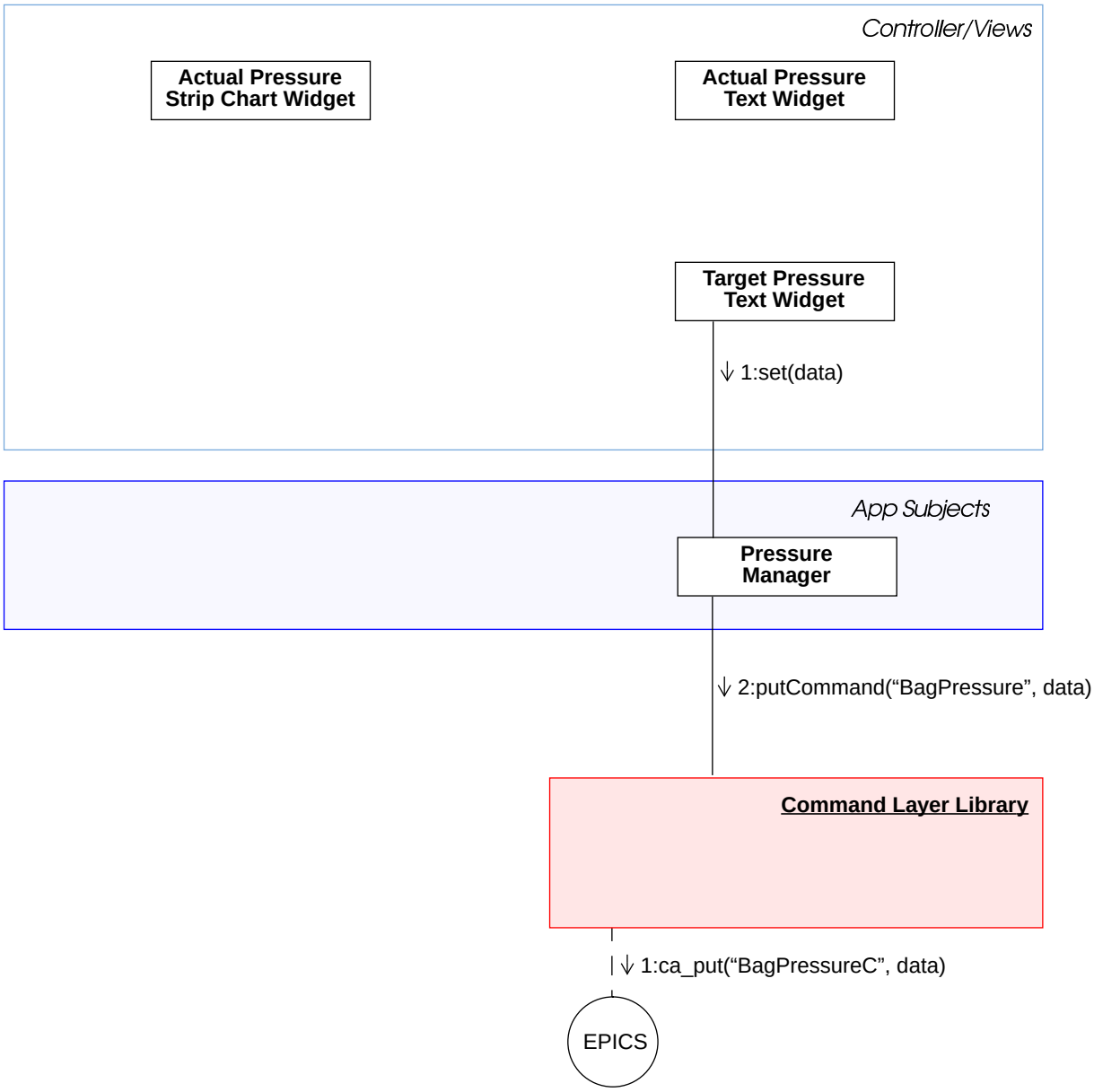



Figure 4 - 5 shows the methods called when the user types a new value for the bag pressure in the text input widget and then clicks the apply button. The "return function" for the text widget is called which calls the "set" method of the Pressure Manager. The Pressure Manager then uses the putCommand method of the Command Layer Library which finds the service associated with BagPressure. The request is then mapped to the EPICS ca_put low-level command.

## 4.4 Script Executor Instances

A script executor is an important application within the OCS. Its function is to take text input from the user or a file and translate the commands or file contents into low-level system-specific commands. Further work on the Object Model for this application type is found in the Planned Observing Support track documentation [12]. This section just gives an overview of how this application is mapped to the OCSApp object model.

**Subjects.** With this application there is no application-specific data; all the data is associated with the System-Subject. There are objects that accept commands and map them to names for the SystemSubject. Functionality for command completion in the scripting language is required.

**Views.** There are probably no views for the Script Executor.

**Controller.** The controller for the Script Executor is the part of the program that accepts lines of text from the user or prints output if needed when the application is used as a shell application.

## 4.5 Agent Application Instances

Agent applications appear in the OCS to isolate the OCS from particular protocols used in external systems. These instances of OCSApp are demons that are started up once and run throughout the observing session (or maybe always). They generally have no user interface (no controller) and display no subject data (no views).

Their only functionality appears in their Subjects. They receive one kind of message and translate it into commands for another.

# 5 Interacting OCSApps

*This chapter discusses how the OCS is modeled upon instances of OCSApp*

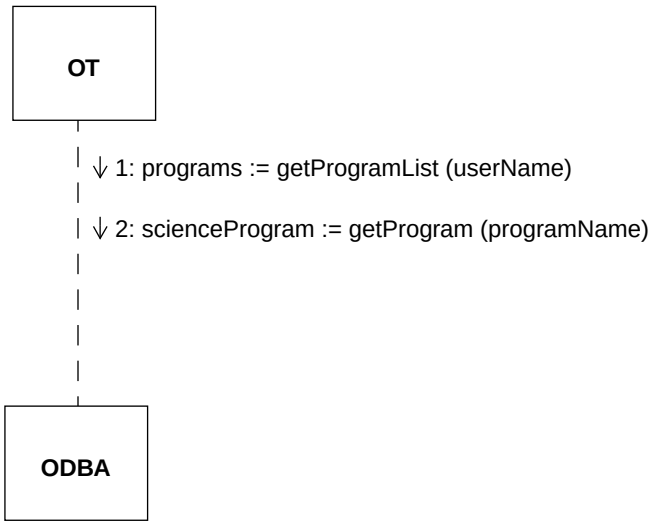
## 5.1 Introduction and Summary

The OCS software system is made up of many OCSApp instances. Previous chapters have described how a single OCSApp is structured and how it becomes reality as an instance of OCSApp functioning upon the capabilities of the SystemSubject. The task of designing a new application is now encapsulated inside the OCSApp and its interactions methods with the outside world are known. Once the abstraction of an OCSApp is understood, the details of individual applications can be ignored and the OCS can be modeled as a set of communicating OCSApp instances.

The functionality and dynamic operation of the OCS can be modeled using the OMT object model and object interaction diagrams and OO-data flow diagrams can be used to understand the behavior of the OCS as a whole.

At this stage of the design the details of the planned observing track and interactive observing are known. These operations are modeled in the appropriate track documents [12] and are not repeated here (for Steven B.). To complete the physical model and to demonstrate the large-scale modeling of the OCS, the following object interaction diagram shows an Observing Tool instance fetching data from the Observing Database.

*FIGURE 5 - 1* Observing Tool Instance Storing Data in the Observing Database




In this example the user of the OT has started an Observing Tool. He requests a list of all the Science Programs he owns. Once he views his Science Programs, he selects one and opens it. The two messages and their replies are shown in the object interaction diagram. The use of object interaction diagrams maps well to scenarios and use cases, which are used throughout the GCS documentation.

Other more complex interactions appear in the OCS Planned Observing Support track documents.


# OCS Interactive Observing Infrastructure Track PD

Kim Gillies*, Shane Walker*, Steve Wampler**

* National Optical Astronomy Observatories
** Gemini Project Office, Gemini Controls Group

**O**bservatory
**C**ontrol
**S**ystem
**P**roject

# 1 Interactive Infrastructure Track Overview

*This chapter introduces the Interactive Infrastructure Track.*

## 1.1 Introduction

The Interactive Observing Infrastructure (IOI) provides a foundation for the OCS applications. Its function is roughly analogous to the function of the kernel of a computer operating system in that it provides the lowest level programmer interface in the OCS. It provides for interactions between OCS applications (instances of OCSApp) and presents an abstract interface to the system's resources. All OCS application interactions with principal systems use the IOI during all planned and interactive observing scenarios. For example, the IOI functionality takes care of the details of sending commands and monitoring status and system health. Consoles, executors, and scripts will all use the functionality provided by the IOI.

A design model for much of the IOI track work was presented in [12]. Because the IOI track plays such a key role in interfacing with other principal systems, it is discussed here in greater detail than the other OCS tracks and in a more implementation-oriented way than the design model of [12]. In particular, we have focused on the details for the communications between OCSApp instances and the EPICS-based principal systems. This chapter contains:

- A discussion of the high-level design of the IOI track,
- An overview of the major software products of the IOI: the Command Layer Library (CLL) and the Principal System Agent (PSA) application,
- Specific details for the approach the IOI track will use to communicate with CAD-based systems,
- Remaining decisions for the detailed design of the IOI track, and
- A list of the documentation that must accompany the IOI release.

Further details on the IOI software products and protocols may be found in the other chapters.

This document and the others in this book have been altered to reflect changes in the design that were a result of the Preliminary Design Review and the Principal Systems Meeting held during the week of the OCS Preliminary Design Review [13].

## 1.2 Acronyms

| | |
|---|---|
| API | Application Programmer Interface |
| ARD | Action Response Database |
| CAD | Command Action Directive |
| CCS | Configurable Control System |
| CLL | Command Layer Library |
| EPICS | Experimental Physics and Industrial Control System |
| GCS | Gemini Control System |
| ICD | Interface Control Document |

| | |
|---|---|
| ICS | Instrument Control System |
| IOC | Input/Output Controller |
| IOI | Interactive Observing Infrastructure |
| OCS | Observatory Control System |
| ODB | Observing Database |
| PS | Principal System |
| PSA | Principal System Agent |
| SAD | Status Alarm Database |
| SDD | Software Design Description |
| SIR | Status Information Record |
| TBD | To Be Determined |
| TCS | Telescope Control System |

## 1.3 Glossary

Additional glossary information is available in [8].

**Attribute —** An attribute is a textual description of some part of a Gemini based hardware or software system. An attribute has an associated value.

**Complete Configuration —** A complete configuration is a list of attribute/value pairs for one or more principal systems, and one or more subsystems within the principal systems. The Command Layer Library splits complete configurations into principal system configurations.

**Configuration Part —** A configuration part is a set of attributes and values that refer to a single system capability, for example a filter wheel motion.

**OCSApp —** A term used to indicate an instance of the OCSApp subsystem defined in the physical design of the OCS [12]. All applications in the OCS software system are OCSApp instances.

**Principal System —** At the highest level in the GCS software decomposition, the software system is divided into four kinds of software systems called principal systems. The four types are called: the Data Handling System, the Observatory Control System, the Telescope Control System, and the Instrument Control System. There may be up to four concurrently executing Instrument Control Systems.

**Principal System Configuration —** A principal system configuration is a list of attribute/value pairs for a single principal system. It is composed of one or more configuration parts.

**Value —** A value is the data associated with a particular attribute.

## 1.4 References

[1] SPE-C-G0037, *Software Design Description*, Gemini 8m Telescope Project.

[2] gscg.grp.020.icdbook, Interface Control Document 1a, 1b, 2. *Gemini Interface Control Documents*, Gemini Controls Group

[3] gscg.kkg.017, *Two Command Models*, Kim Gillies, Gemini Controls Group

[4] gscg.kkg.002, *OCS Behavioral Interface Model Report*, Gemini Controls Group

[5] gscg.kkg.005, *Baseline Major Systems Interface*, Kim Gillies, Gemini Controls Group

[6] GSCG.kkg.009, *ICD 1a - The System Command Interface*, Gemini 8m Telescope Project.

[7] GSCG.grp.024, *ICD 1b - The Baseline Attribute/Value Interface*, Gemini 8m Telescope Project.

[8] GSCG.grp.016, *Glossary for the Gemini Software*, Gemini 8m Telescopes.

[9] ocs.kkg.014, *Observatory Control System Software Requirements Document*, Gemini Observatory Control System Group

[10] ocs.kkg.031, *Preliminary Sequence Command Design*, Gemini Observatory Control System Group.

[11] ocs.kkg.036, *Preliminary Design for Access Control in the OCS*, Gemini Observatory Control System Group, 1995.

[12] ocs.ocs.002, *OCS Physical Model Description*, Gemini Observatory Control System Group, 1995.

[13] gscg.sbw.071/02, *Notes from Principal Systems Meeting #2*, Steve Wampler and Steven Beard, Sept. 20, 1995.

## 1.5 Document Revision History

**First Release —** 17 July, 1995. Pre-release draft.

**PDR Release —** 16 August, 1995.

**Final PDR Release —** 9 October, 1995

## 1.6 Studies/Decisions During Interactive Observing Track

From the OCS Software Design Review, the following decision must be made as part of the IOI detailed design step:

**OCS Communication Infrastructure.** There will be many processes running concurrently in the OCS that must exchange information (e.g., consoles and the Principal System Agent discussed later in this document). A study must be done to decide upon the infrastructure that will be used to accomplish this.

## 1.7 High-Level Design Goals of the Interactive Observing Infrastructure

The Software Design Description [1], describes the Gemini Control System software. The interface between principal systems is discussed in detail in documents [6] and [7]. This document assumes familiarity with the software design and terms defined in the design document and interface documents. Familiarity with the discussion of the IOI track in [12] is also assumed.

The following factors influence the design of the IOI:

- Since the IOI handles all interactions with principal systems and within the OCS, it must be simple and fast.
- The IOI must be designed to work with EPICS (or EPICS-like) principal systems.
- The IOI must provide a convenient command interface to its OCS clients. Both synchronous (waiting) and asynchronous (non-waiting) command options are needed.
- The IOI must support monitoring both the actions and status of the systems being controlled.

The following subsection introduces some terminology that is used throughout the IOI design. The remainder of the section covers the primary products of the IOI and a summary view of the dataflow in the IOI.

### 1.7.1 *Terminology*

Every OCS application is a client of the IOI, since the functionality of the IOI track is used by OCS applications to "talk amongst themselves." (In fact, the Command Layer Library discussed below is the implementation of the SystemSubject subsystem in [12].) In this document however, we are primarily interested in the communication infrastructure required for OCS applications to command other principal systems.

Clients of the IOI that communicate with other principal systems are consoles, script executors, and shell tools. Each must send system-independent *sequence commands (SC)* to the principal systems. A sequence command consists of an *opcode* with an optional argument. The argument can be a *configuration* or a simple string. A configuration can be viewed in our system as a set of system dependent commands in the form of attributes and values. A *complete configuration* consists of configurations for more than one principal system. A *principal system configuration* is the portion of a complete configuration which goes to one particular principal system. A *configuration part* is a group of attributes that are associated with a single system capability (such as a filter wheel motion). A configuration part is called a command when it can be mapped to a principal system command. Note that a configuration can have just one part so it is proper to refer to a configuration part as a configuration. However, the terms are not interchangeable; it is not true that a configuration is always a configuration part.

The sequence commands are very high-level and global, meaning that the execution of the command influences the operation of the entire principal system. A principal system (PS) can execute at most one instance of a global command at a time and it is difficult to imagine scenarios where two sequence commands would be executed simultaneously in the same principal system. The one exception is **config(apply)**. This command causes the target principal system to match the configuration.

Sequence commands are covered in greater detail in [10].

### 1.7.2 *Principal Components of the IOI*

The Command Layer Library supports a flexible, open architecture for communicating with the software systems of the GCS. The CLL and its Services are the primary software products of the IOI. In addition, the IOI track uses Principal System Agent processes to provide access control and to serialize communications from the services to the principal systems. The products are introduced below.

#### 1.7.2.1 *Command Layer Library Overview*

All communication from the OCS to other principal systems and communication between OCS applications is handled by a software library called the Command Layer Library (CLL). The CLL is the implementation of the SystemSubject subsystem in the physical model. Each software entity in the OCS that must communicate with another principal system is linked with the CLL. Again, in this document we are stressing communication with other principal systems.

An important job of the CLL is to abstract the details of updating status and sending sequence commands and monitoring their completion. It must provide its client with the option of waiting for a sequence command to finish before continuing. In the case of interactive observing, command completion will be reflected on the console, and the application will probably not need to wait until all the actions it initiated have finished (see section 5.2.4.2 "Console Interactive Graphical Control Semantics" in [1]). However, the functionality to wait for a command to complete will be required for executors and scripts and will remain an option for console commands as well.

The CLL delivers sequence commands (e.g., **config(observe)**, **config(apply)**) bound for principal systems other than the OCS through processes called Principal System Agents (PSAs) (see below). In the case of **con-**

**fig(apply)**, the CLL is given a complete configuration consisting of attribute/value pairs for one or more principal systems. The CLL splits this configuration into one or more principal system configurations and sends them as the arguments to separate sequence commands to the appropriate PSAs using the OCS Message System (see Figure 1 - 1 on page 1 - 8).

If a sequence command causes actions and the application must wait for completion of the actions, the PSA monitors the actions (using its own CLL) for error/completion/interruption updates and returns this information to the waiting application. Completion functionality is required in the OCS, but is not directly supported by EPICS. When the command completes or is interrupted by another command, this information is returned asynchronously to the PSA and then client through the EPICS Command Action Response (CAR) monitoring protocol (see Chapter 5).

There are some less important functions within the CLL including access control and logging. More details on this library can be found in Chapter 2.

#### 1.7.2.2 *Principal System Agents*

Each principal system other than the OCS will have one principal system agent (PSA) that represents its functionality in the OCS. PSAs are *not* used for intra-OCS communication. Their chief purpose is to isolate the details of communicating with a foreign system in one place. Since the PSA is an OCSApp too, this simplifies the modelling of the remainder of the OCS since all principal systems appear as OCSApps.

As an OCSApp, the PSA also includes the CLL. All communication from an OCSApp to a principal system passes from the application to the principal system's agent process. The PSA first evaluates a request to determine if the caller has adequate permissions to control its principal system. If the application does not have permission, the configuration is rejected immediately and the client is alerted of the error.

In the case of **config(apply)**, the PSA processes the principal system configuration argument one part at a time. The PSA maps the attribute names to the format expected by its principal system and *presets* the configuration part. For EPICS-based principal systems, this means that attribute names are translated into the ARG fields of the appropriate command application directive (CAD) record(s). When the PSA sets the *preset* directive, it causes the CAD to process and validate the arguments - no other activities happen at this time.

A principal system may either accept or reject a new command when it is preset. When a part is rejected by the PS, no more parts of the configuration are processed and a *reject* message is returned to the client. If all of the parts in a configuration are successfully preset, the PSA applies the configuration using **config(apply)**. At that time, the PS evaluates the configuration in its entirety for any inconsistencies or problems and either accepts or rejects the entire configuration. In either case, an accept/reject response is returned to the client CLL to complete the synchronous command request. This message is called the *agent response*.

After accepting a **config(apply)**, the PS then takes whatever actions are needed to match the configuration. The PS updates the command application response (CAR) records associated with the parts as they progress. The CAR records reside in the Action Response Database (ARD) and are used to determine the current state (e.g., *IDLE*, *BUSY*, *ERROR*) of the action being commanded. The PS also keeps the global **apply** CAR record busy until the configuration is mapped.

The PSA can also support non-EPICS-based principal systems by simply changing the mapping. A non-EPICS-based system would communicate with its PSA through some method other than Channel Access. This detail would be known only to the principal system and its PSA.

### 1.7.3 Action Response and Status Alarm Databases

The Action Response Database (ARD) and Status Alarm Database (SAD) comprise the EPICS part of the OCS Observing Database. The ARD provides information on any ongoing actions associated with a commanded entity, for example whether it is *IDLE* or *BUSY*. The CAR record for an entity also contains the identification of the client that last controlled it. Higher OCS software layers use this information to ensure that actions are not interrupted. The SAD provides current status values for all attributes using status information records (SIR). The principal system keeps the ARD and SAD up to date using channel access as the command is being executed. The CLL includes the ability to monitor this information to keep clients up-to-date.

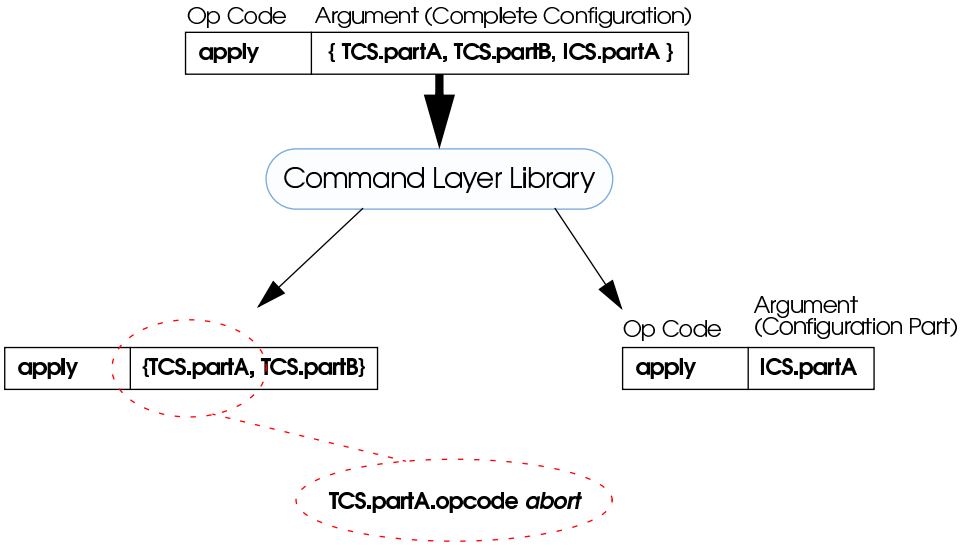
## 1.8 Data Flow Summary

The first section below illustrates how the CLL splits a complete configuration into one or more principal system configurations. The next section places the IOI in the context of the greater Gemini Control System and the final section further details the duties of the CLL and PSA when commanding a principal system.

### 1.8.1 CLL Example

The **config(apply)** command must be accompanied by a configuration argument. The Command Layer Library splits the complete configuration into principal system configurations and sends each to the principal system's Principal System Agent.

*FIGURE 1 - 1* Sequence Commands, Configurations, and System-dependent Commands



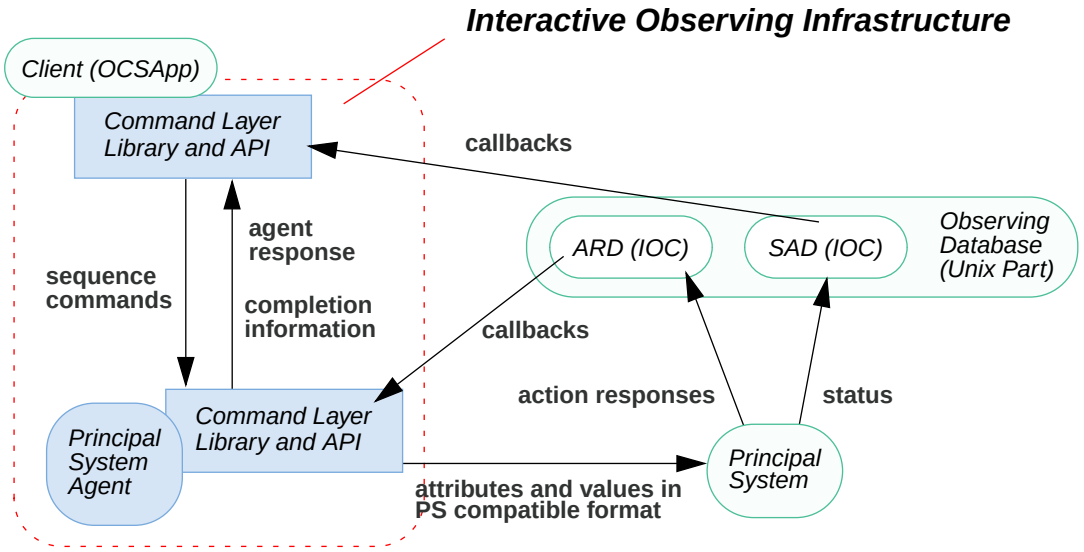

These points are illustrated in Figure 1 - 1. The **config(apply)** sequence command handed to a CLL contains two parts for the TCS, and one part for an ICS. The configuration is split into two principal system configurations and each is sent separately to the appropriate PSA. Each configuration part argument (TCS.partA, TCS.partB, and ICS.partA) is a separate system-dependent command. In the case of TCS.partA, the command is "abort". This abort applies to TCS.partA only, not to the entire TCS.

Each configuration part is viewed as a system-dependent command. Accordingly, it has an *opcode* attribute that can take one of the values **abort**, **preset**, or **stop**. These opcodes refer specifically to a single-function command and should not be confused with the global opcodes of sequence commands.

### 1.8.2 IOI Data Flow

Figure 1 - 2 illustrates the flow of data within the parts of the IOI during a sequence command application to another Principal System. The “Client” in the figure could be a console, an executor, or a shell program. The client’s CLL sends the appropriate sequence command (along with configuration part information if necessary) to the destination PSA using an RPC-like, request/response protocol. The PSA receives the command, evaluates the caller’s permissions, translates it into a format accepted by its PS, and then applies it. The PS must immediately accept or reject each command. To complete the request, the PSA returns the result of the application, the agent response, to the client.

FIGURE 1 - 2 Data flow in the IOI

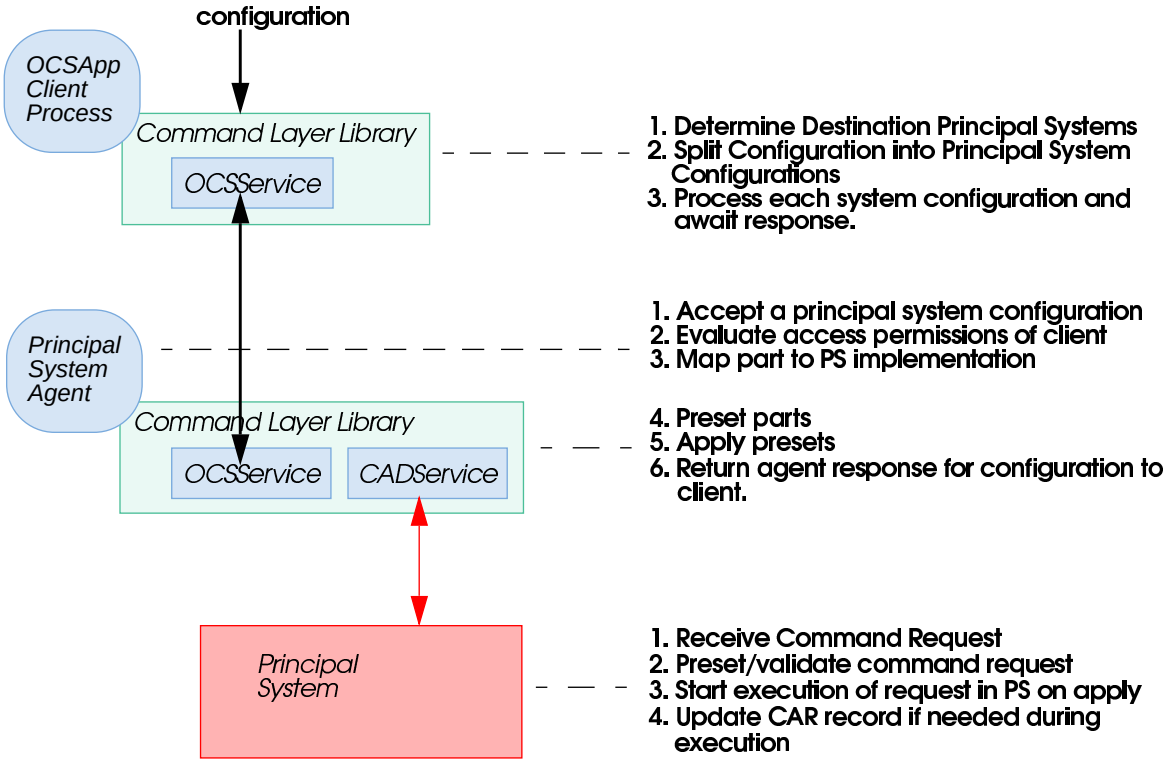



The principal system has no knowledge of the sender of a command. Once it accepts a command, it simply takes whatever actions are required recording updates in the ARD and SAD. Each command includes action response attributes in the ARD that can be monitored by the PSA to determine command completion. This information is passed back to the client. In the case of a console, changes to the SAD can be monitored by the client and displayed. (A console could also monitor the ARD strictly for display purposes, though this data path is not pictured.)

### 1.8.3 CLL and PSA Responsibilities

The responsibilities of the principal parts of the IOI track involved in OCS/principal system communication are detailed further in Figure 1 - 3, which focuses on communication with an EPICS system. See Chapter 2 and Chapter 3 for more information on the CLL and PSA. For non-EPICS-based principal systems, the details of the PSA would differ.

FIGURE 1 - 3 IOI Layer Responsibilities (Commanding an EPICS-based PS)




In the figure, the OCSApp client process and the principal system are not part of the IOI, but have been included for reference. Again, the client could be a console, an executor, or a shell program.

The OCSService and CADService are parts of the CLL and are discussed in Chapter 2. Briefly, the OCSService is used for intra-OCS communication, and the CADService is used by a PSA's CLL to apply CAD records to EPICS-based principal systems.

The CLL splits configurations according to principal system. The PS configurations are sent to the Principal System Agents for processing. The CLL waits for the configurations to be accepted or rejected before continuing.

## 1.9 Interfaces

The IOI is used by OCS processes to communicate with other principal systems. The interface between the rest of the GCS and the IOI is provided by the Command Layer Library. As mentioned previously, every process that must communicate with another principal system is linked with the CLL.

Communication between the IOI and a PS is usually handled via CAD/CAR/SIR records as described in [6]. Some principal systems will reside in the Unix world in which case other means based on the use of the OCS Message System may be employed (the OCSService). In either case, the Principal System Agent process handles the direct command application to its principal system.

## 1.10 Documentation

The OCS SDR documents provide the guidelines for software documentation (SR66, SR67, SR68). The IOI track does not provide end-user applications, so only library and testing documents are needed. The IOI will be released with the following documentation:

**The Interactive Observing Infrastructure Technical Document.** This document describes how the IOI works and will include sections for both the CLL and PSA.

**The IOI Command Layer Library Programmer's Document.** This document contains the CLL API and discusses how the library should be used.

**The Interactive Observing Infrastructure Testing Manual.** This manual will describe how to use the testing procedures required for the acceptance tests of the IOI.

## 1.11 Deliverables

The deliverables of the IOI track include the following.

- The Command Layer Library and testing software
- Documentation as listed in this document and the OCS Development Plan

# 2 Preliminary Requirements for the Command Layer Library

*This chapter introduces the functionality of the IOI Command Layer Library and states the requirements that it must satisfy.*

## 2.1 Introduction

There are three types of software applications in the OCS that must communicate with the other principal systems: consoles, script executors (what was called Sequence Executor in previous documents, but are a little more general), and shell programs that execute some kind of scripting language (i.e. TCL, PVWave language, Java, Python, etc.). These applications must also be able to communicate among themselves within the OCS.

This document describes the required functionality of the Command Layer Library (CLL) interface which provides OCS application programs with a single common software communication interface to the multiple communication protocols. A particular focus is on the interactions of the OCS with the EPICS-based principal systems. The high-level design is discussed in [12]. The CLL is the implementation design for the SystemSubject subsystem.

## 2.2 Goals of Design

The low-level, principal systems interface part of the OCS design has been through many iterations. The following goals seem to survive all our discussions of the IOI.

- The IOI should be lightweight—simple, understandable, and quick.
- The IOI should be matched to the functionality provided by the other principal systems, but should add whatever other functionality is required in the software system.

  It is assumed that the majority of the PSs are EPICS-based using CAD records. The IOI is designed to work with an EPICS or an EPICS-like system (i.e. one that mimics the behavior of an EPICS system.)

## 2.3 Preliminary Library Design

The environment of the software described in this document is shown in Chapter 1 along with a high-level view of the overall design/functionality of the IOI. The IOI track provides a common software interface that enables one OCSApp to communicate with other OCSApps and with principal systems. OCSApps send and apply system-dependent configurations to target applications and principal systems. See Figure 1 - 3 on page 1 - 10 for an overview of the CLL responsibilities when commanding a principal system.

A software library will be created that provides a software interface to the command and status functionality of the Gemini Control System for the use of OCSApp instances. This Command Layer Library will be available as a statically linked or dynamically loaded library that must be included by all software applications within the OCS that must communicate with any other process in a principal system.

The preliminary design calls for a number of Service libraries that encapsulate the implementation details of particular communication systems and provide the functionality demanded of the common software interface. The CLL Services are to be loaded by the CLL on demand so no application must be needlessly bloated with unneeded communication code.

All three types of OCS applications will link with the CLL and their application functionality will be built upon the functionality of the CLL. They will use only the public interface to the CLL API to achieve their functionality. See Chapter 7 for more information on design decisions made during the IOI development.

Including the CLL functionality in an IOI track application will most likely require the use of a threads mechanism to allow monitoring and notification of waiting/completion independently of the main function of an application or console. Threads are available in Solaris so this is not a problem on the chosen/specified Gemini hardware.

The terms “configuration” and “attributes/values” are used here as they are used in the OCS SDD sections [1]. The exact internal form/content of a configuration or attribute is not known at this time, but it is not important to this paper or the IOI design as a whole.

In the following discussion, a user of the CLL (a program linked or making calls to the CLL) is called a *client* or *a client of the CLL*.

Preliminary specifications/requirements for the CLL are indicated with an *RXXX* in the left margin of the page. These requirements will be refined during the detailed design of the IOI track.

## 2.4 General CLL Requirements

### 2.4.1 Initialization

Each application in the OCS will be given one or more unique identification strings that are associated with it and identifies it within the Gemini Control System. This identification string is used with many CLL API calls.

R1 • *The CLL must be initialized by the application.*

R2 • *The CLL API must include a way to obtain or set an application’s unique identification. The method will be through a CLL method call.*

### 2.4.2 Authentication/Access

Any application within the OCS will have the capability of accepting commands and publishing status information. The application must be able to control what clients have access to its information.

R3 • *All applications must be able to control access to their functionality. Applications will need a way to validate the identification strings of applications that are trying to send them commands.*

R4 • *Applications need to be able to store the list of clients from which they will accept commands.*

### 2.4.3 Configuration Input/Output

A sequence command consists of an *opcode* and an optional argument, which may be a simple string label or a *configuration*. A configuration can be viewed in our system as an opaque set of system dependent commands in the form of attributes and values. A *complete configuration* consists of configurations for more than one principal system. A *Principal System (PS) configuration* is the portion of a complete configuration that goes to one particular principal system. A *configuration part* is a group of attributes that are associated with a single system capability (such as a filter wheel motion). A configuration part becomes a *command* if that part maps to a command in another application or system.

Note that a configuration can have just one configuration part so it is proper to refer to a configuration part as a configuration but it is not true that a configuration is a configuration part.

Note that there will be information in configurations in the form of attributes and values that does not map to commands in principal systems or OCSApps. Not all configuration parts are commands.

#### 2.4.3.1 *Input Requirements and Constraints*

R5 • *All CLL requests must include the application's unique identifier. (constraint)*

R6 • *The CLL will accept sequence commands made up of opcodes and optional arguments as inputs. An argument may be a configuration or a simple string. A configuration may be represented by an associated "configuration" name betokened by a "configuration name" argument (also a simple string).*

R7 • *"Configurations" includes configuration parts, PS configurations, or complete configurations as inputs.*

R8 • *The CLL will have the capability of examining a complete configuration to determine which principal system agents should receive PS configurations.*

R9 • *The client of the CLL will be required to indicate whether or not the client wishes to wait for completion of the commanded actions when the sequence command is submitted to the CLL (constraint).*

R10 • *PS Configurations or entire configurations complete when all the actions that are associated with their parts are complete. The CLL must provide a function that will synthesize completion for PS and entire configurations from the completion information of the parts, which is set by the other principal systems.*

These requirements indicate that the OCS messages will be made up of attribute/value lists.

### 2.4.4 *Action Monitoring*

The OCS software design uses only the *action values* that are present in the Action Response Database (ARD) to monitor completion (and other states) of commanded actions in the principal systems.

R11 • *The CLL provides the ability for an OCSApp to monitor action variables.*

R12 • *The ARD is hidden from the higher software libraries by the CLL public interface and no other software library or OCS software entity uses the ARD.*

R13 • *The CLL will provide a common interface that allows a user of the library to monitor a specific named action variable for changes and to assign a callback to a change of the action variable. The application programmer interface must be independent of the details of the implementation of the Service that provides the action variable.*

R14 • *It will also be possible to assign a callback to a specific action value (i.e. the callback will only be called when the variable goes to "error").*

For EPICS-based action variables, CLL will allow clients to monitor action variables directly to show transitions on CAR records visually.

It may be desirable to keep the action variables that must be monitored for a part to complete in the Observing Database. This is to be decided by the IOI developers during the IOI track detailed design.

### 2.4.5 *Completion/Waiting*

If a client needs notification of completion of a configuration, the CLL must keep a record of the action variable in the requested configuration part and some state information so that the client caller can be notified when the configuration completes.

R15 • *The CLL must provide the client with the ability to wait for completion of a configuration. Completion is determined through the action variable mechanism and provides no more capabilities than the action variable mechanism provides.*

R16 • *The CLL must be able to combine the completion information from multiple principal system configurations and OCSApp completion information to synthesize completion for a complete configuration.*

R17 • *The CLL must provide a synchronous wait capability where the client will block until the configuration is completed.*

R18 • *The CLL must provide an asynchronous wait capability where the client can supply a callback function that will be executed when the configuration completes (or goes to the error state as described in other action-related documents).*

Notification of correct completion and modification of actions comes from the ARD and other services, and the CLL must be available to receive those messages and to notify the operators/users of problems.

## 2.5 *Status/Alarm Functionality*

The CLL will also provide the application programmer interface to general status and the Status/Alarm Database.

### 2.5.1 *Status Functionality*

The status values for all EPICS-based principal systems are represented by SIR records in the Status Alarm Database (SAD). OCSApps can also provide status information and the model will be the SIR record.

R19 • *The CLL provides a Service-independent application programmer interface to status value functionality.*

R20 • *The SAD is hidden from the higher software libraries by the CLL programmer interface and no other software library or OCS software entity uses the SAD.*

R21 • *The CLL must provide calls to allow a client to monitor the value of a status variable.*

R22 • *The CLL must provide a way to interact with the other fields of a status record (SIR), which include: name, description, type, units, and alarm value and message.*

### 2.5.2 *Alarm Functionality*

The CLL will provide a service-independent software interface to the alarm mechanism as presented in the SIR records of the SAD. The SIR alarm functionality provides the model for alarms for all Services.

The alarm functionality will primarily be used by a single OCS client program that is charged with presenting alarm information. This client is called the Alarm Manager.

R23 • *The CLL must provide calls to allow a client to monitor the alarm value of a status variable.*

R24 • *EPICS-system status variables exist in the "logical" SAD.*

R25 • *EPICS-system status records can exist in other physical EPICS databases, but they must be able to be moved to the physical OCS SAD if needed.*

R26 • *The CLL must provide a way to show the message fields of status records.*

R27 • *The CLL must provide a subscription-based alarm capability for the use of OCSApps.*

The specifications and requirements for the Alarm Manager program will be determined during the Telescope Console Track.

## 2.6 Service Support

The SystemSubject design calls for a number of Services that can be used by an OCSApp to communicate with other parts of the software system. The design places some general requirements on the CLL.

R28 • *The CLL must have the capability of supporting the multiple interprocess communication systems present in the project.*

R29 • *A Service provides access to one communication system/protocol.*

R30 • *The communication API must be the same for all Services.*

R31 • *It must be possible for an OCSApp to link with/use only the Services it needs and no others.*

R32 • *The CLL communication API should not require the programmer to be aware of Services.*

## 2.7 Services

Two Services are positively known at this time. A Service that provides access to EPICS systems using Channel Access and an OCSService that provides an OCSApp with access to other OCSApp instances. The implementation of the OCSService is not yet known. The following are requirements for any Service.

R33 • *The functionality of all Services is based upon the model of EPICS Channel Access.*

R34 • *A CLL Service must provide a subscription-based status capability for the use of OCSApps. OCSApps will not use the Status/Alarm Database for their status.*

R35 • *A CLL Service must provide an alarm interface similar to that of EPICS.*

### 2.7.1 EPICSService

The EPICSService allows an OCSApp to use “raw” EPICS Channel Access functionality. This service is used for status and alarms from EPICS-based systems.

R36 • *A Service must be provided that allows OCSApps to communicate EPICS records. All the features of the communication model must be supported (status, commands, alarms).*

### 2.7.2 OCSService

R37 • *A Service must be provided that allows OCSApps to communicate with other OCSApps. All the features of the communication model must be supported (status, commands, alarms).*

### 2.7.3 The CADService

The design for command communication between an OCSApp and an EPICS-based principal system calls for an intermediate process called the Principal System Agent. The agent uses the CADService to communicate with the principal system. The CADService adds a protocol to the EPICSService. An OCSApp sends a configuration to a PSA using the OCSService.

R38 • *All PS configurations will be sent in their entirety to a PS Agent process.*

A PSA will process the contents of a configuration by sending the parts to its principal system using the CAD record protocol.

R39 • *The CLL in a PSA OCSApp will check for acceptance/rejection of the configuration parts before the PSA’s activity, the delivery of the configuration, is completed.*

Each principal system will have one Principal System Agent (See Chapter 3). There is no other communication between the Command Layer and the PS agent during the delivery of a part. The following recipe shows the steps involved in the delivery of a part from the CADService to a Principal System Agent.

1. If the client wishes to wait, the CLL keeps one record of the action variable required to determine when the requested configuration is completed. For multiple requests, the CLL still keeps just one record of waiting since completion is determined for all requests when the one action is completed.
2. Optionally, the CLL may consult the Observing Database to determine relevant information required to communicate with the PSA that is responsible for the configuration part. The information could be cached so that the ODB is only contacted once.
3. CLL delivers a principal system configuration to a PS Agent using the OCS Message System and awaits acceptance or rejection of the configuration. The response is called the *agent response.*
4. Once the configuration part is accepted or rejected the CLL has no knowledge of the configuration part unless the client made a request to wait for completion.

The protocol required with this approach for communication between the CLL/PSA/PS will be fine-tuned during prototyping. The preliminary design of the PSA protocol and action variable protocol is defined in Chapter 4 and Chapter 5. Here are some issues:

1. We require that the CLL wait for an agent response from the PSA/PS before going on. This requires a request/response protocol between the CLL/PSA/PS that places a requirement on the OCS Message System for an RPC-like synchronous capability. This is more than likely required anyway. This approach simplifies the CLL design since the actions would only be used to indicate conditions after a configuration part has been accepted and would not overload the “error” action state. Potential race conditions are thus minimized as well.
2. The CAR/PS protocol requires changes to the CAR and CAD records along with a commitment by the principal system developers to provide the CAR information the protocols require. This requires work from the controls group office.

## 2.8 Other CLL Functionality

### 2.8.1 Logging

R40 • *The CLL is required to support logging as specified in the OCS requirements. The logging functionality will allow the CLL client to log important messages. The library itself will also optionally log information.*

R41 • *Logging should work with the system-wide logging interface in the DHS.*

R42 • *There should also be the capability to send the logging information to a local file to allow independent development.*

The content of logging messages will be determined by the IOI developers during the IOI track.

### 2.8.2 The status(commands) from the SDD.

The SDD describes two commands that can be used by high-level software to learn about the exported status information of the principal systems. These commands are status(all) and status(one).

R43 • *The CLL must provide the capability described by the SDD for the status(all) and status(one) commands in the CLL API.*

### 2.8.3 Error Handling

Some error handling for the OCS is built upon the EPICS system functionality.

R44 • *The CLL API must provide a way for applications to monitor the status of the connections they make to other applications.*

R45 • *The CLL must notify the application when monitored variables in the EPICS systems become unavailable.*

# 3 Preliminary Design for OCS Principal System Agents

*This chapter introduces the functionality of an IOI Principal System Agent and states the requirements on its functionality.*

## 3.1 Introduction

This document describes the required functionality of the Principal System Agent (PSA). The PSA is a primary product of the IOI track along with the Command Layer Library and the Services.

### 3.1.1 Historical Information

The ICD1 document [2] describes a previous iteration of the principal systems interface. In that design there were two processes in the system for each principal system called *Command Servers*. The command server in the principal system received entire configurations from the OCS command server and mapped the configuration information from the OCS to the proper commands/records in the target principal system.

This design was abandoned because of the possible difficulties with the OCS group developing code that must run in a principal system developed by another group. There were also changes to the attribute/value layer in the first ICD1 which were brought about by comments during the SDD design review that allowed the required functionality to exist in the OCS itself.

### 3.1.2 Why have a PSA?

One of the goals of the OCS design is to keep the details of the other principal systems from spreading throughout the OCS. Configurations support this concept because they are opaque to most of the OCS components. Treating the configurations as opaque data provides the maximum amount of encapsulation and modularity. Here are some benefits to having an agent for each principal system.

- The features and peculiarities of a particular principal system are centered in one OCS process making long-term maintenance less difficult.
- The PSA places the configuration unpacking and mapping in a single process in the OCS allowing applications to continue to pass configurations as opaque blocks of data, but places no burden on the other principal systems.
- The PSA makes the other principal systems appear to be instances of OCSApp making modeling of the OCS simpler.
- A PSA can, if needed, isolate a particular interprocess communication protocol from the OCS.
- The PSA provides a single place in the OCS that can be modified to support visitor instruments.

## 3.2 Preliminary Design

Each Principal Systems Agent (PSA) is a process which runs on the Configurable Control System machine. Its functionality replaces the functionality of the command servers of the original ICD1.

The following definitions are from Chapter 1. A sequence command consists of an *opcode* and an optional argument, which can be a *configuration* or a simple string. A configuration can be viewed in our system as a set of system dependant commands in the form of attributes and values. A *complete configuration* consists of configurations for more than one principal system. A *Principal System (PS) configuration* is the portion of a complete configuration which goes to one particular principal system. A *configuration part* is a group of attributes that are associated with a single system capability (such as a filter wheel motion).

A configuration part is called a command when it can be mapped to a principal system command in a PSA (a CAD record with an EPICS PS). The environment of the PSA in the IOI and the OCS is shown in Figure 1 - 2 on page 1 - 9. The responsibilities of the CLL and the PSA are shown in Figure 1 - 3 on page 1 - 10.

The PSA receives Sequence Commands one at a time from applications that link with the Command Layer Library. The primary function of the PSA is to process a configuration by mapping configuration parts to commands in a principal system and to apply the configuration using the system's attribute/value interface. On an EPICS system applying a part is done with a number of Channel Access ca_put calls to the system's CAD records.

In the current model of PS communication, the PSA synchronously waits for the acceptance or rejection of each command as it is preset. It passes acceptance or rejection of a configuration back to the caller CLL before accepting another configuration.

The PSA handles command completion when action variables are associated with commands.

### 3.2.1 *Other Kinds of Systems*

The PSA is the part of the OCS that would be adopted to support new/non-EPICS principal systems (instruments). The PSA would map the functionality/interface of the principal system to that provided/required by the OCS.

The PSA can also serve as a Command Layer level interface for a Unix-based system. In this situation, a PSA would be located on a host that serves the Unix-based system. The PSA receives messages using the OCS Message System and maps the configuration parts to whatever is required for the Unix-based system. It returns configuration completion information using the OCS Message System.

### 3.2.2 *Security/Access to Principal Systems*

A limited, simple form of access control is part of the OCS software requirements. Basically, the OCS must prevent the activities and control of owned resources of one observer from interfering with the activities and resources of another.

The PSA is the principal system server and must be the one OCS process that determines whether a client application is allowed to control a principal system. The PSA must evaluate the identity of the caller and determine if that caller has permission to control the principal system.

The OCS access control mechanism will be based upon keys and Unix-style owner, group, and other permissions (See [11]). The PSA will examine the key that is passed to it from a CLL as part of a configuration part. If the key is not acceptable, the command will be rejected before passing it on to other PSA software layers.

Control of the resources is determined by the telescope operator through the Session Manager application interface. The act of awarding resources communicates to the PSA which observers have access to what systems.

## 3.3 *Required Functionality*

### 3.3.1 *High Level Responsibilities*

PSAR1 • *The PSA represents the principal system in the OCS software environment. Each principal system has one PSA in the CCS. All configurations pass through a PSA before being applied to a principal system.*

### 3.3.2 *Initialization*

PSAR2 • *For EPICS-based principal systems, the PSA must load a file (or fetch from the ODB) the mapping of configuration attributes to principal system record names and field names. Since the functionality of a principal system is reasonably static (it won't change during an observing session), this is reasonable.*

PSAR3 • *The PSA must have access to the mapping of configuration parts to action variables. This may be loaded from a file or fetched from the ODB as in the previous requirement.*

### 3.3.3 *Input/Output*

PSAR4 • *The PSA accepts sequence commands from the CLL.*

PSAR5 • *The PSA applies sequence command parts to principal systems and synchronously waits for acceptance or rejection of commands during the preset phase.*

PSAR6 • *The PSA formulates the Agent Response for the configuration and returns it to the CLL caller.*

### 3.3.4 *PSA System Functions*

PSAR7 • *The PSA will have the capability of breaking a principal system configuration into its configuration parts.*

PSAR8 • *The PSA operates on only one opcode/configuration at a time. In a sense, it is a principal system server.*

PSAR9 • *The PSA maps configuration part information to the attribute/value layer of its principal system.*

In most Gemini systems, the attribute/value layer is represented by CAD records. With CAD records the PSA must map attributes to the ARG fields of the CAD record.

In some other kind of principal system, the PSA would map the configuration information to whatever command system is appropriate for the system. The PSA must evaluate the command using the current principal system protocol and return the agent response regardless of whether the principal system is EPICS-based or not.

PSAR10 • *For EPICS/CAD systems, the PSA must toggle the correct directive in CAD records based on the contents of the configuration part. These directives are: PRESET, ABORT, and STOP.*

PSAR11 • *The PSA must APPLY a configuration after the parts have been PRESET. Any configuration errors noted by the principal system that couldn't be evaluated during the PRESET are evaluated when APPLY is executed.*

PSAR12 • *The PSA must use the PRESET and APPLY results to form the configuration's agent response, and return the response to the CLL caller.*

#### 3.3.4.1 *Security/Access.*

PSAR13 • *The PSA must evaluate the keys passed from the CLL and determine whether the caller can control the PSA or a portion of the PSA.*

PSAR14 • *The access control required by the OCS software requirements is coarse. The model is of the permission system in Unix with owners, groups, and others.*

#### 3.3.4.2 *Logging*

PSAR15 • *The PSA must be able to optionally log its activities using the Gemini standard logging interface or to a local file.*

#### 3.3.4.3 Error Recovery

A principal systems may go down or for some other reason become unavailable. The PSA must react when systems disconnect. This is probably a problem only at certain times.

PSAR16 • *While a PSA is interacting with a principal system (a command is ongoing) it must handle the disconnection of the principal system. When connected to an EPICS system, the PSA would get a connection alarm from channel access.*

PSAR17 • *The PSA must notify its current client if a system it is commanding on the client's behalf becomes unavailable during the application of the command. The action taken by the PSA will be to "forget" the command if a PS goes down. Operators will need to resubmit their requests once the PS becomes available.*

This sounds worse than it really is. There can be at most one command ongoing in a PSA at any time.

## 3.4 Remaining Design Decisions

The PSA must await the acceptance/rejection of each part by the principal system, apply the configuration noting any errors, and immediately return the result to the caller of the PSA. This request/response would appear as a remote procedure call to the CLL. At this time there is no reason to assume a synchronous connection will not provide adequate performance based upon typical EPICS and Unix interprocess communication times and the probable rather low OCS command rate.

## 3.5 PSA EPICS Implementation Details

The PSA must have the capability of mapping the configuration part attributes to the functionality of the principal system the agent represents. In our system, the destination is an EPICS system and the functionality of that system is provided by a set of EPICS CAD records.

The PSA will receive an APPLY sequence command as an opcode and configuration, which is a a set of attributes and values. The PSA must find the configuration parts and map each part. The attributes must be mapped to the *fields* of a specific CAD record (a part is by definition, the parameters required for a single operation). An example is now shown for an offset configuration part:

```
telescope:offset:ra              20
telescope:offset:dec             25
telescope:offset:unit            arcsec
telescope:offset:command         apply
```

The PSA must take the *offset* and map it to an EPICS CAD record. It then maps the fields: *ra, dec, unit, action, command* (in this example) to the fields of the offset record. This mapping currently must be kept in the PSA and is loaded once at initialization time. An example of field mapping is:

```
telescope:offset:ra              tel:offset:A
telescope:offset:dec             tel:offset:B
telescope:offset:unit            tel:offset:F
```

The simple mapping described here could be done through simple hash table data structures which use attribute names as keys.

It is not necessary that the configuration part on the left (offset) map directly to a CAD record of the same name on the right.

# 4 The EPICS Action Variable Protocol

*This Chapter describes how CAR records provide information on actions in the Principal Systems.*

## 4.1 Introduction

In Gemini Control System terminology, actions are what happen in an EPICS system when a command is issued. Actions are the side-effect of issuing commands. Much of the functionality of the OCS and the IOI relies on having some way to monitor what and when actions are happening in the other principal systems. This is important in the Gemini system because our software system must be able to tell when operations are complete in order to successfully implement planned observing or waiting in scripting languages.

The EPICS system, upon which most of our principal systems are based, provides a wealth of information on the values or status of hardware devices. However, it provides very little that assists software above EPICS in determining when activities in an EPICS system are occurring or completing. The traditional EPICS solution of determining completion based on status values is a poor solution for Gemini because it results in very device-specific code spread throughout the EPICS and OCS clients resulting in code that is difficult to maintain. That is a big problem in the Gemini distributed software development environment. While this paper is focused on the implementation of action variables in EPICS systems, our intent is to use action variables for the same purpose within the OCS.

The Gemini project solution to this problem is to supplement the EPICS system with a new record that provides client software with additional information on what parts of an EPICS system are doing. This record, the Command Action Response or CAR, enables client software to determine when actions in an EPICS system are completed without monitoring status values and performing device-specific processing of status values. The impact on the EPICS systems themselves is minimal.

This paper is based on and extends the preliminary work in ICD2 [2] (since updated) and [3]. The approach is to look at the action variable model, define the meaning of the action variable values and protocol, and show how principal systems use CARs. A number of IOI scenarios are covered in another chapter of this book (Chapter 6)

Changes the OCS proposes to the CAD/CAR/SIR records are given here. (See “Changes To CAD/CAR Interactions” on page 4 - 31.)

This paper includes changes agreed upon during the Second Principal Systems Meeting [13].

## 4.2 Action Model Summary

Each controllable part of an EPICS principal system is associated with one or more *action variables* (in the OCS Action/Response Database). One implementation of an action variable is an instance of the EPICS Command Action Response (CAR) record. When a Command Action Directive (CAD) record is applied one possible result is that some actions execute in the system. Associated with any CAD record is one CAR record that can be monitored by software above the EPICS system to determine when any actions caused by the application of the CAD record are complete. There is a coupling between CAD and CAR records; actions in a principal system occur as a result of the application of a CAD.

The static (unchanging during an observing session) associations between CAD and CAR records are documented in the Parameter Description Format, which each principal system produces during development.

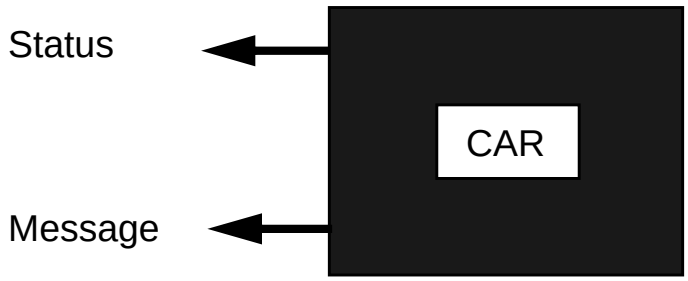
However, a principal system might also cause its devices to operate on its own, not as a result of a CAD record application. This is called a *self-initiating action*. A PS still updates the action variables so that upper layer software can monitor that actions are taking place in the principal system. For instance, while tracking a target the TCS might adjust the primary support structure. The status values for each of the support devices would change as would the action variable that indicate that the “primary support device” is busy.

The goal of the Action Variable Protocol is to allow the OCS or another system to track the actions of the principal systems and to give operators the best feedback possible when actions are interrupted or modified by other sources.

## 4.3 The Action Variable

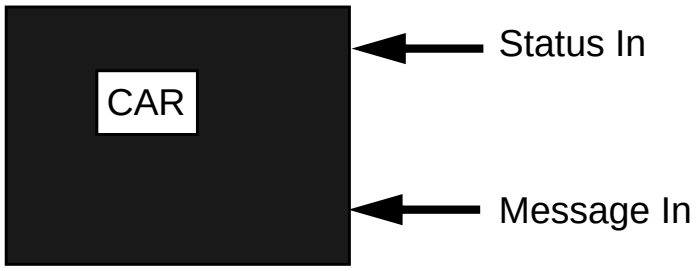
The CAR record is a simple status record. Figure 4 - 1 shows the external CAR interface and Figure 4 - 2 the internal fields. Both figures are from [7] (normal EPICS fields are not shown).

*FIGURE 4 - 1* External interface for CAR records




The **Status** may be one of a few known values (documented later in this document). The **Message** provides information about the nature of the Status response if necessary.

*FIGURE 4 - 2* ‘Internal’ Connections for CAR records




The **Status In** field is set by the internal caller to change the **Status** of the CAR record.

The CAR record has been implemented by the Gemini Project Office and is under their change control. New versions will need to be written to support the ideas in this paper.

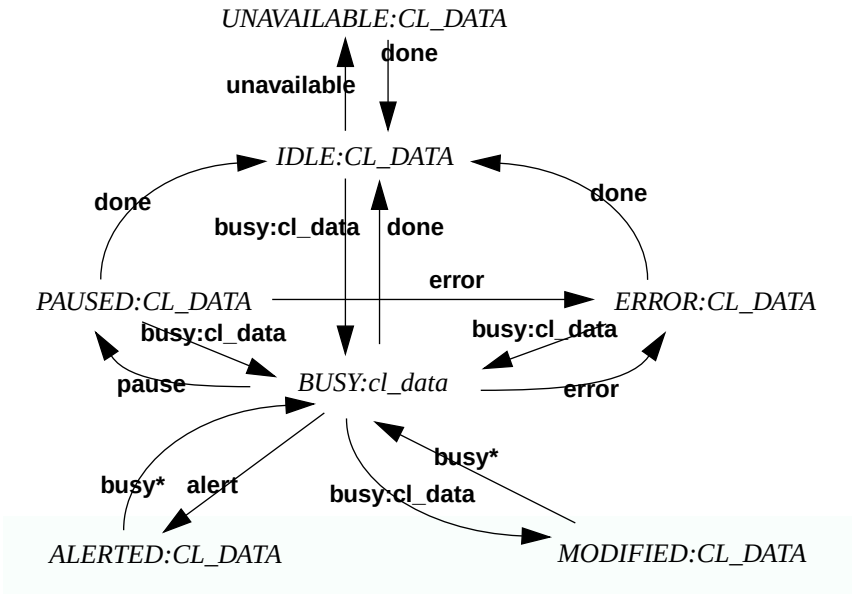
## 4.4 Action Variable Protocol

The kind of information that can be obtained from the CAR data is determined by the Action Variable Protocol. This section describes the possible values for the status field of a CAR record or any other action variable.

Figure 4 - 3 shows a state machine for the Action Response Protocol. The possible states for a CAR record are in capitalized italics (IDLE, ERROR, PAUSED, MODIFIED, BUSY, ALERTED, UNAVAILABLE). The values an EPICS database sets in the CAR record to move the machine between states label the arcs (busy:cl_data, done, pause, alert, error, unavailable). The client data (CL_DATA) of the application that last caused the action to go to the BUSY state is appended to the current value of the CAR record as in BUSY:cl_data. The OCS CLL CAR Protocol uses this information to ensure that no other process has interrupted an action before it completes.

The two states inside the shaded box are temporary states that the CAR record enters and spontaneously decays back to the BUSY state. These temporary states are used to communicate certain conditions to programs monitoring the CAR record.

*FIGURE 4 - 3* Action Response Protocol




*** Indicates spontaneous decay into a state**

The *caller* in the following discussion is some internal code written by the principal system developer. This code is responsible for the implementation of the actions in the principal system. During execution of the actions, it is the caller's responsibility to set CAR **Status In** fields of any associated CAR records with the appropriate values at the appropriate times (discussed later). The meaning of the CAR states is as follows.

**IDLE.** No action represented by this variable is underway. Any actions associated with this variable have completed or were never started.

**UNAVAILABLE.** Some actions may become unavailable in particular modes of operation. Systems make this known by setting appropriate CARs to UNAVAILABLE. The only way to leave unavailable is to set a CAR with done or some alias for done.

CAD records that control actions that can become unavailable should check for this condition before accepting a command.

**BUSY.** The actions associated with a variable are occurring. This variable will have the value BUSY:cl_data on **Status** as long as the actions associated with this variable continue. Once the caller knows the actions associated with the variable are complete, the variable must move to IDLE by setting **Status In** to *done*.

Whenever a client writes *busy* it must also write the client data of the client that is causing the action to become busy. The CAR record saves the client record whenever *busy* is written. Systems should do not need to write (and therefore save) cl_data with other state values.

**ERROR.** The ERROR state indicates that a problem occurred *during* the execution of an action. The internal caller sets the **Message In** before setting **Status In** to *error.*

This state should only be used to notify the users that a problem occurred with the execution of the actions themselves. It should not be used for the following:

- Improper arguments - These are to be trapped in the CAD records. Only good requests should make it past the CAD records for execution in the database or principal system code.
- Requests for unsupported capabilities - If a particular action can not be modified or stopped/aborted, the CAD record must check for this and reject the request. This should not be handled by ERROR.
- Bad status values or out of range conditions - These are to be handled by status or alarms in the SIR records, not in the CAR.

All these issues should be addressed in the CAD subroutine, not by CAR record updates. When it is impossible for the system to evaluate these issues in the CAD subroutine, the ERROR state should be used with an appropriate message. This should be used rarely.

The ERROR action is set when a problem occurs with the action itself. For instance, during execution of the command the software notices that the hardware has become defective. Errors of this kind could also become evident through health if the system cannot continue or status if the errant action causes out of range values.

**PAUSED.** The PAUSED state indicates that some actions associated with this variable, which are already underway, have been paused by an operator/observer. This implication is that at some later time they will be continued by setting **Status In** to *busy.* Since relatively few operations can be paused once started, “pause” is not one of the directives supported by every CAD record. If an action can be paused, a special “pause” CAD must be created for that action. An IOC would not typically pause an action.

PAUSED actions that are aborted become IDLE actions. PAUSED actions can go to the ERROR state causing an alarm.

**MODIFIED.** The MODIFIED state is a temporary state entered when an action associated with a variable, which is already underway, is applied again. This gives systems monitoring this action an indication that something interacted with an ongoing action (like a DRAMA kick or a repeatedly pushed offset button). From the MODIFIED state, the CAR record spontaneously decays back to BUSY after firing off all monitors.

Some actions can not be modified once they have started, CAD records controlling these actions should not accept applies if they are busy and can not be modified. A decision on whether a CAD can accept new applies

once actions are busy is made by the principal system itself, not the structure of CAD/CAR records or the OCS. CAD records probably do not exist for self-initiated actions.

The ERROR state should not be used by systems to declare that they can not be modified.

**ALERTED.** The ALERTED state is a temporary state entered when an ongoing, long action wishes to notify users with some useful text information. The internal caller sets the **Message In** field before setting **Status In**. (This functionality is similar to the DRAMA trigger.)

This state can only be entered once an action is busy and can't be used to send messages back to operators that aren't associated with ongoing actions.

### 4.4.1 *The State Transitions*

The following table defines the meaning and use of the CAR states and the possible transitions between states. The CAR state machine is implemented in the CAR record and it is not the responsibility of the principal system programmer to keep track of what state his code is in. The design allows PS developers to always do the same thing when they receive a command request. The CAR record produces the correct information for monitoring systems based on its state and input.

Setting **Status In** to a value that is not a defined transition for a particular CAR state does not cause an error. The CAR record simply ignores the caller's request and remains in its current state.

### 4.4.2 *How IOCs Support the Action Variable Protocol*

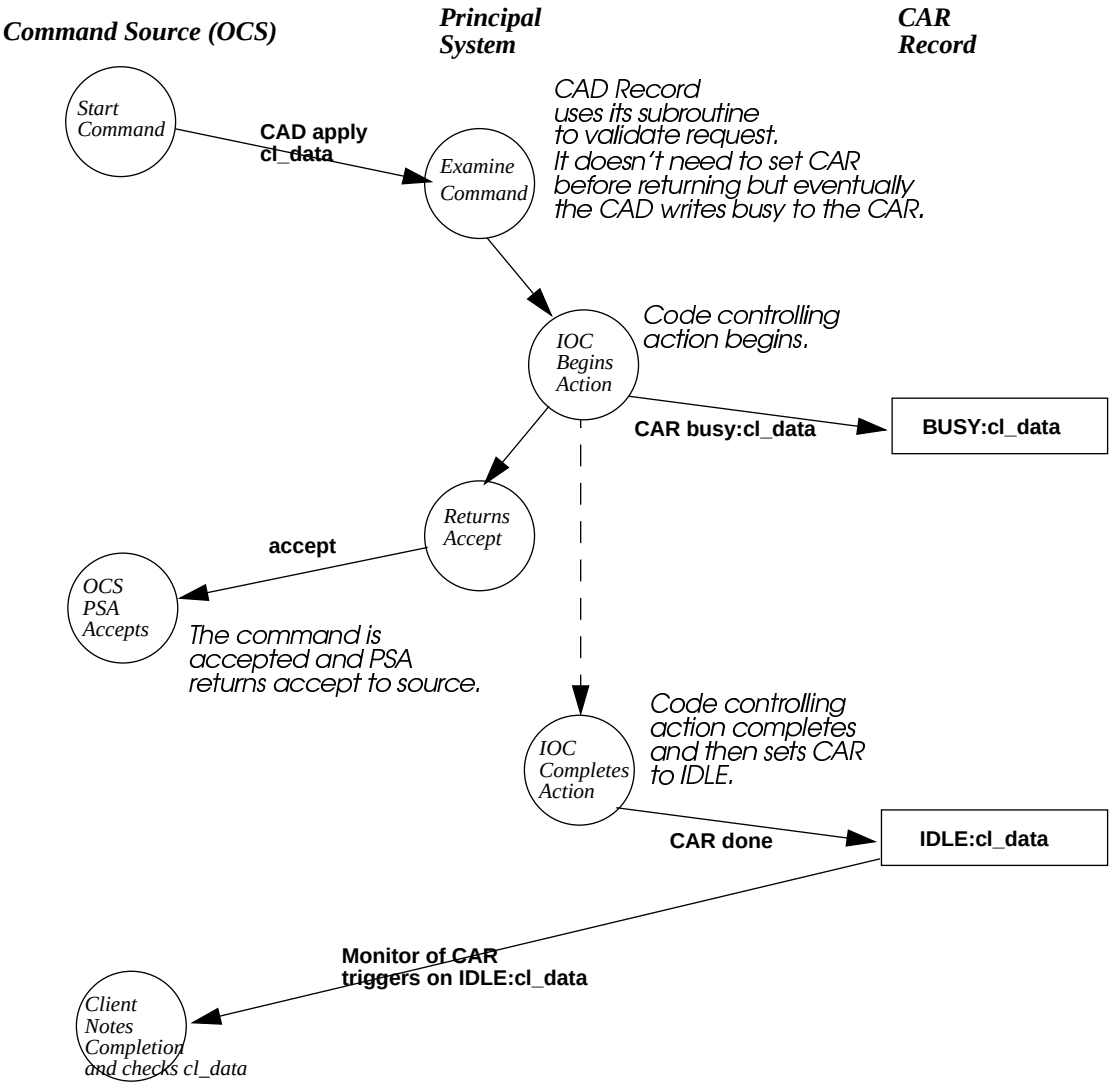
To support the Action Variable Protocol a PS system IOC is required to set the CAR records at the proper times. The design of the CAR protocol is meant to require as little work by the PS developers as possible. Figure 4 - 4 shows a typical CAD-started action that is accepted by the PS and completes successfully.

CAD subroutines are not required to set appropriate CAR records before accepting or rejecting a preset, but they are required to set the CAR eventually and in the same order that requests are made. Without the cl_data in the CAR record, this relaxed approach is not possible. Without cl_data the CAR record must be set *before* the CAD record finishes processing the PRESET directive.

TABLE 4 - 1 State Machine Transition Descriptions.

| Start State | Final State | Entered With | Description |
|---|---|---|---|
| IDLE | BUSY | busy:cl_data | The PS CAD accepted a command and sets *busy:cl_data* just before starting the requested actions. |
| | UNAVAILABLE | unavailable | The principal system has determined that the actions associated with this CAR record cannot be commanded at this time. Each unavailable action should be set with *unavailable.* |
| UNAVAILABLE | IDLE | done | Once in the UNAVAILABLE state the actions it represents are no longer available. The PS must set the CAR record with *done* before accepting self-initiated or CAD commands. |
| BUSY | IDLE | done | The CAD-initiated PS actions have completed. The final PS action is to set the action variables with *done*. |
| | ERROR | error | An ongoing busy action has encountered an action-related problem. It sets Message_In and then Status_In to *error.* |
| | PAUSED | pause | A user wishes to pause an ongoing CAD-related action. The request has been accepted by the responsible CAD record. The PS pauses the action and sets its action variables with *pause*. |
| | MODIFIED* | busy:cl_data | An apply has been accepted by a CAD record that can modify an ongoing action. When the PS sets *busy:cl_data* again, it causes the CAR to temporarily enter the MODIFIED state, fire off monitors and return to the BUSY state. |
| | ALERTED* | alert | The PS can relay information to systems monitoring an ongoing action by setting Message_In and then *alert*. The CAR must be in the BUSY state. Self-initiated actions can not send alert messages to consoles. |
| ERROR | IDLE | done | The PS writes a done to a CAR in an ERROR state to put it in the IDLE state. This is used to clear an error, but the next request for the action is also sufficient to move the CAR out of the ERROR state. There is currently no way to clear an error like this using the CAD record. |
| | BUSY | busy:cl_data | A PS setting Status_In to *busy:cl_data* on a CAR in an ERROR state causes the CAR to return to the BUSY state. This is what happens when an operator tries to re-execute a failed action. |
| PAUSED | IDLE | done | A PAUSED CAR can go directly to IDLE by setting Status_In to *done*. |
| | BUSY | busy:cl_data | A PS notifies users that a PAUSED action is resumed by setting Status_In to *busy:cl_data*. |
| | ERROR | error | A currently paused action has encountered an action-related problem. It sets Message_In and then Status_In to *error.* |
| MODIFIED | BUSY | busy:cl_data | Once in the MODIFIED state, a CAR record notifies any monitoring clients that its action has been modified and returns to the BUSY state on its own. |
| ALERTED | BUSY | busy:cl_data | Once in the ALERTED state, a CAR record notifies any monitoring clients that it has an alert message and returns to the BUSY state on its own. |
| *Indicates Spontaneous Decay to BUSY | | | |

FIGURE 4 - 4 Activities that must occur for a CAD-initiated action in an EPICS IOC.




### 4.4.3 *Changes To CAD/CAR Interactions*

To ensure that a client starting an action can determine that his action has not been modified by some other client while it is busy, the CAD and CAR design needs some minor changes from the ICD documentation. In this section, we detail the changes required by the OCS as well as the changes and new features requested by the UK partners.

#### 4.4.3.1 *cl_data Identifiers*

Each application that is potentially going to write to another principal system must have a unique identifier and this identifier is written to the CAD record as part of a command request. The cl_data identifier will consist of the client id and some other information. The client identifiers will be managed by the OCS and is opaque to principal systems. In other words, other principal systems will blindly handle cl_data identifiers as discussed below without interpreting their value.

#### 4.4.3.2 CAD Record Changes

- The CAD record is modified to allow a PSA (or other client) to set data in the CAD record's cl_data. This requires a new CAD field.
- UK developers suggest CAD records that handle different numbers of input/output records (1, 5, 8, etc.). Change names from INPX to A, B, C. Provide inputs as inputs or links.
- Add a Stop directive to complement Abort and remove the Verify directive for a CAD. Developers decide what Stop and Abort mean for their systems within the broad project definitions.
- Change the name of the apply CAD directive to *preset.*
- Remove the Pause and Continue directives for CAD. (These will be handled by a separate CAD when needed.
- When a CAD record starts an action (or modifies an action) it writes *busy:cl_data* to the appropriate CAR record. This allows the client to ensure that when the action goes to IDLE, no one has interrupted/modified his action. The IOC is required to eventually write to the appropriate CAR, but it is not required to do it from the CAD subroutine. The CAR must be written in the order requests were made to the CAD. This change requires the cl_data be propagated from the CAD to the CAR.
- The CAD test subroutine must check for at least four things.
  - -- It must check for appropriate arguments and reject if they are bad.
  - -- It must determine whether or not it can support the requested directive. For instance, can the action be stopped?
  - -- If an action is in progress and a new request is made, the test subroutine must decide whether or not the actions controlled by the CAD can be modified.
  - -- If actions can become unavailable, the CAD must check to make sure it's available.
- If no device actions are required to match a CAD request, the PS must still write BUSY then IDLE to show completion.

#### 4.4.3.3 CAR Record Changes

- Add new CAR state and value: UNAVAILABLE.
- CAR record **Status In** argument is now a string argument.
- The CAR record values are changed to strings with the cl_data appended as in: BUSY:cl_data, MODIFIED:cl_data, ALERTED:cl_data, PAUSED:cl_data, IDLE:cl_data.
- Require one CAR record per CAD record.
- The CAR record must be altered to provide the state machine described in this document.

**Note:** When writing CAR messages other than *busy:cl_data* (i.e., *done, pause, alert, error, unavailable*), the principal system only needs to write the message value, not the cl_data. This means that the IOC is not required to keep the cl_data around in his system.

**Note:** Self-initiated actions do not go through the CAD records, but the IOC must still write *busy:cl_data* with the cl_data of the IOC. If they do that, other systems can also determine when IOC initiated actions are complete

#### 4.4.3.4 SIR Record Changes

- Provide a SIR with an float value and a SIR with a string value.
- Remove the FITS keyword and FITS comment items from the SIR.

### 4.4.4 Protocol Notes

#### 4.4.4.1 Vague Information from MODIFIED

Note that without the cl_data information there is not really enough information from the states of the CAR record to relay much information about *who* (meaning which process or client) modified an ongoing action. This is one reason why we need the cl_data. For instance:

- Two consoles each enter a single offset for 10 arcsecs at roughly the same time. The TCS starts one and the offset action variable goes BUSY. The second offset is executed and the action variable passes through MODIFIED and returns to BUSY. Both consoles register a modified request and neither really gets what was requested. The first console will note that the cl_data isn't its cl_data with the MODIFIED value. The second console should see that the current cl_data is not his before it starts. Since the second console knows it is modifying an ongoing action it is reasonable to assume that it knows what it is doing.
- Now one console issues two consecutive offsets with quick pushes to the offset button. The first offset action is again modified and the console could display it indicating that the second offset was indeed accepted and executed, but warning the operator at the conclusion of the action would be misleading.

The OCS removes this ambiguity in the software layers above the attribute/value layer in the CLL by checking cl_data for changes. See Chapter 5. The goal is to limit the occurrence of these situations with observatory management and resource management, but notification of problems remains an important issue with which we must contend.

#### 4.4.4.2 CAR Record Monitoring

Another benefit of cl_data is that it tremendously simplifies CAR record monitoring. The cl_data fields provides a direct association between a command and the actions it causes in a principal system. Without it, there is no way to determine with certainty when a requested action has begun or completed.

#### 4.4.4.3 Error Ambiguity

The ERROR state of an action variable can be combined with the use of health and status alarms in the SAD to help the operator pin down problems quickly. There is some overlap with multiple ways to report problems (health, SIR alarms, action ERROR). The action ERROR should only be used to report unrecoverable problems with ongoing actions that keep the action from properly completing.

For instance if a filter wheel belt broke, the health of the instrument would be bad, an alarm might be triggered showing an out of range filter value, and the action variable could show ERROR with an associated message stating a belt broke because the action can never complete with a broken belt.

#### 4.4.4.4 Disconnect Information

The CAR record has no information or logic for dealing with network disconnection that occurs when an IOC goes down or becomes unresponsive. The monitoring for this condition must be handled on the client side, not in the IOC or the record itself. The OCS Command Layer Library and PSA will monitor disconnect alarms in the OCS. Principal systems other than the OCS commanding principal systems through CAD records must invent their own disconnect response.

# 5 CAR Record Monitoring Protocol

*This chapter describes the use of CAR record values by the OCS Command Layer Library.*

## 5.1 Introduction

Commands are sent from processes in the OCS (e.g. sequence executors, consoles, or shell programs) to other principal systems. In order to properly control these systems, the OCS needs to know when a command completes or is interrupted prior to completing. While EPICS provides a great deal of information on the current status of an action, it does not support monitoring command completion.

To help solve this problem without resorting to examining device-specific status values, the Gemini project has developed a new record, the Command Action Response or CAR record. The CAR record provides client software with additional information on the state of each action. For instance, an OCS client process can monitor an action's CAR record to determine whether it is idle or busy.

Though it goes a long way toward a solution, the CAR record alone is not sufficient to obtain accurate command completion information. It is limited from the point of view of an OCS client in that it only reflects the state of an action itself, not the state of a specific command. Problems can arise when two or more OCS clients request the same action, or when an OCS client commands a device that can periodically initiate its own actions. In these cases, simply waiting until an action is no longer busy does not mean that it completed the last command without being interrupted.

To solve this problem, we have built a protocol on top of the CAR record to monitor command requests as well as actions. In this way, commands can be directly associated with the actions they cause in principal systems. However, this protocol only deals with monitoring the actions caused by a command, not with insuring *successful* completion. For example, no warning is issued by this protocol if a command directs a filter wheel to go to position 5, but the resulting actions complete leaving the filter wheel at position 4.

This chapter discusses the protocol used in the OCS to obtain accurate command completion information. Although the protocol will work for arbitrary action variables (whether or not they are implemented as CAR records), it is discussed solely in terms of the CAD/CAR EPICS environment for simplicity.

## 5.2 Principal System Requirements

In order to implement the monitoring protocol, a small number of constraints must be placed on the behavior of principal systems. Each requirement listed below is indicated with a *CRx* in the left margin of the page.

*CR1* • *There is one CAR record per CAD.*

Allowing multiple action response variables per command does not increase functionality, but it does create a much more complex system. Instead of monitoring a single variable, the command protocol would have to be equipped to examine a set of variables and combine them.

*CR2* • *Regardless of whether a commanded entity is already in the state specified by a new command, the principal system must still write "busy" followed by "done" to the action's CAR record.*

The protocol relies on monitoring action responses to determine command completion. If the principal system could decide to sometimes bypass writing to the CAR record, then the protocol could not work.

CR3 • *A principal system that accepts a command must eventually write busy:cl_data to the appropriate CAR. It must do so before writing busy:cl_data for any other future request for the same command.*

Without this requirement, commands could get "lost" in the PS. Should a new command be applied before the actions for the first one begin, only the second command would cause actions. This stipulation guarantees that the actions for a command are seen before a new command is accepted.

## 5.3 Command Protocol Details

The PSA is responsible for monitoring commands for completion (using its CLL) and returning this information to the client process. However, the client itself may also use its CLL to monitor CAR records for display purposes. The CLL service used in the two cases differs. The PSA uses the CADService for the command completion protocol, and the client uses the simpler EPICSService for displaying CAR record values. See Chapter 2 for more information on the CLL.

The protocol described here only involves monitoring the completion of individual actions. Of course, the PSA will also need the ability to monitor an entire principal system configuration for completion. This functionality can be implemented on top of the single-action monitoring protocol by combining the completion states of the various actions. Alternatively, the APPLYC CAR record could be used in some cases to monitor the completion of a set of commands since the APPLY CAD is always set to enact a configuration [10]. This distinction is not important here since the protocol is the same regardless of how it is used.

An understanding of the Action Variable Protocol, covered in Chapter 4, is crucial to this discussion. As mentioned above, the PSA's CLL is given the central role in determining completion for sequence commands. To accomplish this, it uses the action variable associated with each command. The major parts of the protocol are detailed below.

### 5.3.1 Command ID

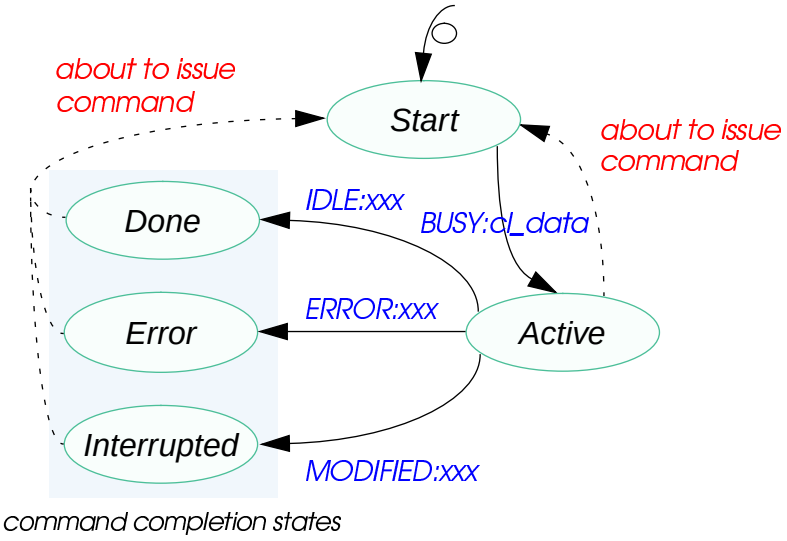
Before a sequence command is sent to a PSA, it is tagged with a unique *cl_data* value that is formed by appending a sequence number to the client's id. Each client is given a unique id when it is created, and sequence numbers are incremented each time a new command is issued. The cl_data becomes a unique argument to the CAD record for the action when the command is applied by the PSA.

Since each command is given a unique cl_data value, it is a relatively easy task to trace the actions associated with the command and hence to determine command completion. This process is best described as a simple state machine as discussed below.

### 5.3.2 Action State Machine

The CLL CADService monitors the CAR record for each action with which it is concerned. Each action has an associated Action State Machine (ASM) in the CLL. When the PS updates a CAR, the record enters a new state, triggering a callback routine in the CLL. The callback routine looks up the current ASM state of the action and makes a transition based upon the value of the CAR record (see Figure 5 - 1).

FIGURE 5 - 1 Action State Machine




In the figure, the dashed arrows represent the transition that occurs *before* issuing the command to the PS. Thus all actions begin in the *Start* state. The remaining transitions occur as a result of CAR record monitor callbacks (and so are labeled with CAR record states). The colon in the label separates the CAR record state from the cl_data field. Where cl_data is “xxx”, it is ignored by the CLL. The transition from *Start* to *Active* is labeled with *BUSY:cl_data*. This indicates that the move is only made when the PS writes **busy:cl_data** for the command that the CLL is monitoring. Thus when in the *Active* state, the CLL knows that the actions in the PS are associated with the command that it just issued.

Once in the *Active* state, an *IDLE* callback indicates that the command has completed without being interrupted. Likewise *ERROR* means the command has ended because something went wrong while it was executing, and *MODIFIED* means that another client has interrupted the sequence command with its own actions. In each case, the ASM remains in one of the command completion states (*Done*, *Error*, or *Interrupted*) until a new command is issued.

Any CAR record values not specifically mentioned in the ASM transitions cause no change of state (i.e., they can be viewed as self-loops on each state). For instance, *ALERTED*, *BUSY*, and *PAUSED* actions are ignored once the command is *Active*, and every callback is ignored once in a command completion state.

### 5.3.3 *Command Completion and Waiting*

The CLL must provide at least three forms of sequence command execution: asynchronous, asynchronous/join, and synchronous. In the first case, the client does not care about waiting for command completion. It sends a command, receives an agent response, and continues with other processing. In some cases though, the client will want to continue processing after sending a command, and then later wait for it to finish before continuing. This requires a “join” type capability. In a scripting language, these interactions will be provided for with statements like **waitlater** and **waitnow** as illustrated below:

```
set waitstate [waitlater $cmd_info]
do something else
waitnow $waitstate
```

Finally, completely synchronous command execution, where the client blocks until the command completes is also required:

```
waitwhile $cmd_info
```

For the completely asynchronous case, callbacks will still occur and the ASM will change states as appropriate, but the command completion information that the protocol provides is not used. For either of **waitnow** or **waitwhile**, the main thread of the client must block until the command terminates in some fashion. This is accomplished by blocking until the ASM for the action enters a command completion state (*Done*, *Error*, or *Interrupted*).

## 5.4 Limitations

The protocol is fairly straightforward. The idea is that the completion state of the last command sent by a client is kept in the ASM for the action until a client tries to initiate new actions. This simple protocol should work well but a couple of limitations on the information it provides must be noted.

- *Whenever a sequence command is sent to a PSA, regardless of whether it is accepted, command completion information for any previous commands to the same capability is lost.*

  Put simply, this means that the CLL can only wait on the last accepted command. If three "offset 10" commands are sent to the TCS, but the third one is rejected, then there is no means for the CLL to wait on the previous two to complete. Even if the third offset is accepted, and completes successfully, there is no guarantee that an offset of 30 has been achieved. The second offset could have been interrupted by another client, in which case the final offset from the initial position could be anything, yet the CLL sees a "*Done*", not "*Interrupted*" completion state.

- *The protocol only provides accurate command completion information, it does not deal with successful command completion.*

  This stipulation was mentioned at the outset. If a device claims to have completed a command, the protocol does not check SIR records against command parameters to check for successful completion.

- *Command completion information refers only to sequence commands, not to the entities being commanded. Before retrieving command completion information with a "wait" statement, the commanded capability may be commanded by another client.*

  The commanded device does not pause or block until the CLL examines the state of its last sequence command. The PS knows nothing about CLLs or completion protocols what-so-ever. Therefore, even if the command completed successfully, leaving the SIR record(s) with the expected values, they will not necessarily retain those values.

To overcome these limitations, if desirable in the first place, would require constructing a fairly complex framework on top of EPICS. It is deemed far better to take advantage of the structure of EPICS rather than attempting to impose a rigid command protocol.

# 6 IOI Track Scenarios

*This chapter describes some typical interactive telescope situations to show how the IOI track performs. (ATTENTION: This document is out of date.)*

## 6.1 Introduction

This paper tests the IOI design through a set of common OCS level scenarios. The IOI track is described in the preceding chapters and this paper assumes you've read them all!

The format of this document is to describe typical OCS scenarios and then show how the IOI behaves for that scenario. Each new scenario can build upon previous scenarios and focuses on what is new to avoid repetitive details. The result is fairly difficult to read, but the alternative would have been to repeat every step for each scenario making it difficult to discern differences between scenarios. In the following, no distinction is made between the CLL and the CADService.

## 6.2 Common Scenarios

Scenario 1 is the basic scenario—it includes all the activities required during processing of a normal command. The other scenarios will refer to this one and will insert new steps in the basic scenario.

### 6.2.1 Console Scenarios

*Scenario 1*

**Short Description**: An observer uses a console to set an instrument filter wheel and applies the change. He does not care about completion.

**Run-through.** The following is the basic building block of IOI commands.

1. The observer modifies the screen to indicate the new filter value and presses the "Apply" button on the GUI.
2. The GUI creates a configuration part with the *preset* directive. It then accesses a service provided by the CLL to send the part to the instrument.
3. The CLL examines the configuration to determine which PSA it goes to.
4. The CLL sends a **config(apply)** sequence command, with the configuration as its argument, to the appropriate PSA.
5. The CLL waits for acceptance or rejection.
6. The PSA receives the sequence command.
7. The PSA extracts and evaluates the client's access to the part and returns reject if the client has no access.
8. The PSA examines the part and maps the part name and the arguments to the appropriate CAD record (or whatever is relevant in the PS).
9. The part becomes a command. The PSA performs the initial preset by writing the appropriate arguments to the CAD record. This includes writing the given cl_data value to the cl_data argument, and writing *preset* to the directive argument.
10. The PS reacts to the setting of the CAD record, processes, and executes the CAD-specific routine that validates the command arguments.
11. The PS finds the arguments acceptable, but does not yet start the action. It simply sets the CAD record's accept/reject field to *accept* and returns.

12 The PSA checks to see if the command was accepted by querying the CAD field as a result of the put callback.

13 Since the command was accepted, the PSA completes the application by setting the APPLY CAD.

14 The PS then evalutes the entire requested configuration.

15 The PS finds the configuration acceptable and sets the APPLY CAD record's accept/reject field to accept and starts the action, writing *busy:cl_data* to the APPLYC CAR record.

16 The PSA checks to see if the command application was accepted by querying the APPLY CAD accept/reject field as a result of the put callback.

17 The command was accepted and has been started. The PSA protocol returns accept to the CLL of the console caller.

18 The CLL of the client forgets everything it ever knew about the request.

19 Once the filter wheel action is complete in the instrument IOC, the filter wheel code sets its CAR record to *done* causing the record to have the value IDLE:CL_DATA.

*Scenario 2*

**Short Description**: An observer uses a console to set an instrument filter wheel and applies the change. He interactively waits for completion.

**Run-through.** *Interactively waits* means that the observer waits for completion by examining something on the screen that shows him when the action has completed. In this scenario a graphical light goes red when the filter wheel is moving and green when it is idle. The observer knows the command is completed when the light becomes green.

1 The console light device is monitoring the filter wheel action variable (CAR record). It uses a CLL API call to this. The name of the action variable was obtained by the developer of the console from the instrument PDF document.

2 Steps 1 - 15 of Scenario 1.

3 The console's CLL monitoring protocol notices that the CAR went from IDLE to BUSY. It call's the application's CAR callback and it changes the color of the light to red, indicating that the filter wheel is moving.

4 Steps 16 - 19 of Scenario 1. Note in step 18, "forgetting everything" does not imply that the CAR record isn't still being monitored.

5 The console's CLL CAR monitoring protocol notices that the CAR went from BUSY to IDLE. It calls the application's CAR callback and it changes the color of the light back to green.

6 The observer notices the transition from red to green and continues his work. He might also choose to look at the status value of the filter wheel but since there was no error or modification, he should be confident that the action completed successfully.

*Scenario 3*

**Short Description**: An observer uses a console to set an instrument filter wheel and applies the change. The value he requests is not valid and the ICS rejects the request.

**Run-through.** The purpose of this scenario is to show how the parameters for a request are evaluated and how our system behaves in this kind of situation.

1 Steps 1 - 10 of Scenario 1.

2 The CAD-specific routine finds the request to be unacceptable. Note that future **config(apply)** commands must be able to rely upon the value of the filter wheel being unmodified/uncorrupted by this aborted command. Thus, the PS must perform whatever actions are necessary to make this true.

3 The PS places a message in the CAD *reason* field. The PS sets the CAD record's accept/reject field to *reject* and returns.

4 The PSA checks to see if the command was accepted by querying the CAD field as a result of the put callback.

5 The command was rejected and it reads the reason.

6 The CLL notes that the request was rejected and hands rejection to the console CLL along with the reason.

7 The CLL forgets everything it ever knew about the request.

*Scenario 4*

**Short Description**: An observer uses a console to set an instrument filter wheel and applies the change. He interactively waits for completion. Before the filter wheel completes, the observer cancels the operation.

**Run-through.** This scenario is the much the same as the previous ones. The difference is that before the action is completed, the observer decides to cancel the operation. All CAD records must support an abort directive which should cause their action to abort if possible. (If not, the command should be rejected.)

1 Steps 1 - 3 of Scenario 2.

2 The PSA checks to see if the command application was accepted by querying the APPLY CAD accept/reject field as a result of the put callback.

3 The command was accepted and has been started. The PSA protocol returns accept to the CLL of the console caller.

4 The console uses the CLL to send the *apply* sequence command for the filter part with the *abort* directive.

5 Steps 3 - 8 of Scenario 1.

6 The part becomes a command. The PSA sets the *abort* directive to the CAD record.

7 The PS reacts to the setting of the CAD record by processing and executes the CAD-specific routine that validates the command arguments. The routine must know whether or not the filter wheel can *abort*.

8 The PS can abort the action. It does the abort of the action and writes *idle* to the action's CAR record.

9 The PS sets the CAD record's accept/reject field to *accept* and returns. (Note the ordering of steps 8 and 9 is dependent upon the PS implementation.)

10 The command was accepted and has been started. The PSA protocol returns accept to the CLL of the console caller.

11 The console's CLL CAR monitoring protocol notices that the CAR went from BUSY to IDLE. It calls the application's CAR callback and it changes the color of the light back to green. (This could happen at any point after step 8).

*Scenario 5*

**Short Description**: An observer uses a console to set an instrument filter wheel and applies the change. He interactively waits for completion. Before the filter wheel completes the observer cancels the operation, but this filter wheel can not be canceled.

**Run-through.** This scenario is essentially the same as the last but this time the CAD record in the EPICS PS evaluates the request and finds it can't do the cancel and rejects it.

1 Steps 1 - 7 of Scenario 4.

2 The PS cannot abort the action.

3 The PS sets the CAD record's accept/reject field to *reject* and returns.

4 The PSA returns reject to the CLL of the console caller.

5 The console posts a message saying the action couldn't be accomplished. Of course, we wouldn't put up a cancel filter wheel button if we couldn't cancel the filter wheel!.

*Scenario 6*

**Short Description**: An operator uses a console to offset the telescope. The offset is set to 20 arcseconds and he wishes to go 60 arcsecs. He waits for each of the offsets to complete before entering the next one.

**Run-through.** This scenario is simple because it is really the application of Scenario 2 (with a telescope offset rather than a filter wheel) three times in a row. The operator sees the offset action light go green

to red then back to green. Following the transition back to green he pushes the offset button again. He repeats to get 60 total arcseconds.

*Scenario 7*

**Short Description**: An operator uses a console to offset the telescope. The offset is set to 20 arcsecs and he wishes to go 60 arcsecs. This time he pushes the offset button three times.

**Run-through.** The only thing that is different here is that the ongoing offsetting action is being interrupted/ modified before it completes. The PSA monitor protocol monitors for this.

1. The operator modifies the screen to indicate a 20 arcsecond offset value.
2. The operator selects *apply.*
3. A console light device is monitoring the offset action variable. It uses a CLL API call to do this. The name of the action variable was obtained by the developer of the console from the telescope PDF document.
4. Same as Scenario 1 steps 3 - 15.
5. Some brief time later, the console light on the screen has gone from green to red indicating the telescope is offsetting.
6. While the command is being accepted or rejected by the PSA/PS, the offset button can not be pushed. This should only be a brief amount of time (milliseconds, I suppose).
7. Once the first request is accepted, the operator pushes the button a second time.
8. Do the steps in this Scenario 2 - 6.
9. As a result of the IOC setting the CAR to *busy*, the CAR record notifies its monitors that it was MODIFIED. It then goes back to BUSY and also notifies its monitors.
10. The CLL monitor protocol notes that the part's action is MODIFIED. It does not alert its client because the source of the modification and the current client are the same.
11. The offset light on the console momentarily flashes indicating that the second offset has been accepted/modified.
12. Repeat these steps 2 - 11 in this Scenario.
13. Eventually, the 60 second offset is complete and the IOC writes *done* to the offset CAR.
14. The CLL Monitor Protocol notes that no one has interrupted the action.
15. The operator notices the transition from red to green noticing that the offsetting action has successfully completed.

*Scenario 8*

**Short Description**: An operator uses a console to offset the telescope. The offset is set to 20 arcsecs and he wishes to go 60 arcsecs. This time he pushes the offset button three times. He then decides that he meant to go west rather than east and cancels the offset.

**Run-through.** This scenario is a combination of Scenario 7, with five pushes of the offset button rather than three, and Scenario 4 where an operator cancels an ongoing operation.

1. The observer modifies the offset console screen value to 20 arcseconds.
2. The observer selects *apply.*
3. Steps 2 - 11 of Scenario 7 are run 5 times.
4. Sometime before completion, the operator pushes the *cancel offset* button which means the telescope should stop doing any offset action now.
5. Steps 2 - 15 of Scenario 4 with appropriate changes of "filter wheel" to "offset".
6. The operator notes the transition from red to green indicating that the offsetting action has successfully aborted.

Scenario 9

**Short Description**: An operator uses a console to offset the telescope. The offset is set to 100 arcsecs and he wishes to go 100 arcsecs. He pushes the offset button and waits. Another operator uses his console to offset 20 arcsecs. What do the operators see?

**Run-through.** This scenario is designed to show that this kind of thing could be a problem and that the PSA protocol gives warnings when it occurs. This kind of situation would result in unknown problems without the PSA protocol.

1. Operator 1 modifies his console screen to indicate a 100 arcsecond offset value.
2. Operator 1 selects *apply.*
3. Steps 3 - 15 of Scenario 1
4. Some time later (doesn't matter since the PSA serializes operations and one of the two must get it first), operator 2 modifies his screen to indicate a 20 arcsecond offset value.
5. Operator 2 selects *apply.*
6. The part becomes a command. The PSA sets the *apply* directive to the CAD record.
7. The PS reacts to the setting of the CAD record by processing and executes the CAD-specific routine that validates the command arguments. The routine must know whether or not the offset action can be modified.
8. The PS can modify the offset action. It does the modify of the action and writes *busy:cl_data* to the action's CAR record.
9. The PS sets the CAD record's accept/reject field to *accept* and returns.
10. The command was accepted and has been started. The PSA protocol returns accept to the CLL of console2.
11. Console 1's CLL CAR Monitor notes that its offset action has been modified before it completed and that someone else modified it.
12. Console1's offset light temporarily shows the modified color but remains green because the offset action is still busy.
13. The console1 operator gets a message saying his offset request probably didn't complete correctly because console2 interrupted it.
14. The console2 operator sees the transition from red to green when offsetting has successfully completed. Chances are pretty good that he didn't get what he wanted, but since the screen showed that the telescope was busy when he sent the command, we don't warn him.

**Note.** When a second source modifies an ongoing operation the original source is notified and the second source becomes the "owner" of the action. This means that the second source can now apply further changes to the action without being notified. If console 1 modified the offset, console 2 would be notified that console 1 was altering console 2's action. This transfer of ownership idea is true for the other directives (pause, stop, continue, and abort) also.

Scenario 10

**Short Description**: An operator uses a console to slew the telescope to a new target. A short time later he changes his mind and enters and applies a new target.

**Run-through.** This scenario is focused on showing that both the two kinds of command modification discussed in the SDD are handled correctly in our system. There is really no difference between this and Scenario 7. It must be possible for the TCS CAD to accept a new target while it is busy moving to a target for this scenario to work.

1. The operator sets the slew target in the console screen value to a new position.
2. Steps 2 - 11 of Scenario 7 substituting *slew* for *offset*.
3. Sometime before completion, the operator enters a new target position and pushes the apply button.
4. Steps 2 - 6 of Scenario 7.

5 As a result of the IOC setting the CAR to *busy:cl_data*, the CAR record notifies its monitors that it was MODIFIED. It then goes back to BUSY and also notifies its monitors.

6 The CLL protocol notes that the action is MODIFIED. It does not alert its client because the source of the modification and the current client are the same.

7 The slewing light on the console momentarily flashes indicating that the second slew has been accepted/ modified.

8 Eventually, the second slew is complete and the IOC writes *done* to the slewing CAR.

9 The CLL Monitor Protocol notes that no one has interrupted the action.

10 The operator notes the transition from red to green indicating that the slew action has completed successfully.

*Scenario 11*

**Short Description**: The operator's TCS console has a button to turn on and off telescope tracking. The execution of this command is instantaneous since there is really no "action" associated with the request. He turns off the tracking.

**Run-through.** The purpose of this scenario is to show that there can be CAD records in a PS that have no actions. Commands that have no real actions can choose to complete immediately meaning that the PS doesn't return until the command is complete. This is acceptable for commands that take so little time that the action mechanism is inappropriate or too heavyweight. To use the immediate completion option, the CAD record must know that the command has completed correctly before returning. There can be no CAR record for an immediate completion CAD record. If there are no action, the CAD returns rejection or immediate completion, which is the same as acceptance.

1 The observer pushes the track-off button.

2 The observer selects *apply.*

3 The CLL sends the apply sequence command to the appropriate PSA. The part contains the *apply* directive.

4 The CLL waits for rejection, or immediate completion.

5 The PSA accepts the sequence command.

6 The PSA examines the part and maps the part name and the arguments to the appropriate CAD record (or whatever is relevant in the PS).

7 The PSA notes that this CAD has no associated CAR record meaning it completes immediately—it can't be interrupted/modified.

8 The part becomes a command. The PSA applies the part (sets the apply directive) to the CAD record.

9 The PS reacts to the setting of the CAD record, processes, and executes the CAD-specific routine that validates the command arguments.

10 The PS finds the arguments acceptable. It executes the action and checks its completion.

11 It completes successfully.

12 The PS sets the CAD record's accept/reject field to *complete* and returns.

13 The PSA checks to see if the command was completed by querying the CAD field as a result of the put callback.

14 The PSA finds the command was completed and returns complete to the CLL caller.

15 The CLL forgets everything it ever knew about the request.

*Scenario 12*

**Short Description**: The operator's TCS console has a button to turn on and off telescope tracking. He turns off the tracking but for some reason the command is rejected by the TCS.

**Run-through.** This scenario is just like Scenario 11 except that the immediate completion command is rejected by the CAD record.

1. The observer pushes the track-off button.
2. The observer selects *apply.*
3. The CLL sends the apply sequence command to the appropriate PSA. The part contains the *apply* directive.
4. The CLL waits for rejection, or immediate completion.
5. The PSA accepts the sequence command.
6. The PSA examines the part and maps the part name and the arguments to the appropriate CAD record (or whatever is relevant in the PS).
7. The PSA notes that this CAD has no associated CAR record meaning it completes immediately—it can't be interrupted/modified.
8. The part becomes a command. The PSA applies the part (sets the apply directive) to the CAD record.
9. The PS reacts to the setting of the CAD record, processes, and executes the CAD-specific routine that validates the command arguments.
10. The PS finds that it can't execute the command right now.
11. The PS sets the CAD record's reason for failure and the accept/reject field to *reject* and returns.
12. The PSA checks to see if the command was completed by querying the CAD field as a result of the put callback.
13. The PSA finds the command was rejected and returns the reason and rejection to the CLL caller.
14. The CLL in the console posts the reason for rejection.
15. The CLL forgets everything it ever knew about the request.

*Scenario 13*

**Short Description**: An observer uses a console to set an instrument filter wheel that takes a long time to move between filters and he applies the change. He interactively waits for completion. The filter wheel sends information whenever it goes through an intermediate value.

**Run-through.** This scenario is designed to show how a console can receive information from an action that is ongoing through the ALERTED state.

1. The observer modifies the screen to indicate the new filter value.
2. The observer selects *apply.*
3. The console light device is monitoring the filter wheel action variable. The console code uses a CLL API call to do this. The name of the action variable was obtained by the developer of the console from the instrument PDF document.
4. Steps 3 - 15 of Scenario 1.
5. Some brief time later, the console light shows that command has been accepted. The light on the screen has gone from green to red indicating the filter wheel is moving.
6. During the execution of the long command, the IOC sets the CAR message_in and then writes *alerted* to the CAR record.
7. The console is monitoring the filter wheel's action variable and its callback responds to the monitor.
8. The CLL code reads the alerted message from the CAR record and posts it on the screen.
9. Steps 6-8 of this scenario occur 3 times during the action.
10. Eventually, the IOC completes the filter action and writes *done* to the filter wheel CAR.
11. The CLL Monitor Protocol notes that no one has interrupted the action.
12. The console monitor for the CAR record goes off and changes the color of the light back to green.
13. The observer notices the transition from red to green and continues his work. He might also choose to look at the status value of the filter wheel but since there was no error or modification, he should be confident that the action completed successfully.

*Scenario 14*

**Short Description**: An observer uses a console to set an instrument filter wheel that used to take a long time but has been tuned up to work quickly. He interactively waits for completion. The filter wheel sends information whenever it goes through an intermediate value but now the information comes fast and furious.

**Run-through.** This scenario shows how the use of alerted can be used improperly resulting in lost data. The scenario is exactly as in Scenario 14 except that this time the IOC writes *alerted* to the filter wheel CAR very quickly. Since the callback response in the client requires a ca_get call to the IOC, there is a good chance that the message that in the CAR record has changed since it posted the original monitor. In other words, when the client reads the CAR record message it always gets the most recent value of the message. The most recent message is probably the correct one. If the CAR was written quickly four times, the client might read the last message four times. This is viewed as acceptable. Alerted is meant to be used as an unreliable status mechanism, not a inter-machine message protocol.

*Scenario 15*

**Short Description**: An observer interactively uses a console to set an instrument filter wheel that takes a long time to complete. An engineer asks the observer to pause the filter wheel before it has completed its motion so he can inspect the mechanism. Later the observer resumes the operation.

**Run-through.** Every configuration part that maps to a command/CAD record can expect the paused and continued directives. Not all hardware devices can be paused and the CAD record subroutine must reject requests if it can not pause whether or not the CAD is busy.

1. The observer modifies the screen to indicate the new filter value.
2. Steps 2- 15 of Scenario 1.
3. Some brief time later, the console light shows that command has been accepted. The light on the screen has gone from green to red indicating the filter wheel is moving.
4. The engineer requests the operator to pause the filter wheel. The operator pushes the pause button.
5. Steps 2- 7 of Scenario 1 with the *pause* directive rather than the *apply* directive.
6. The part becomes a command. The PSA pauses the part (sets the *pause* directive) to the CAD record.
7. The PS reacts to the setting of the CAD record, processes, and executes the CAD-specific routine that validates the command arguments.
8. The PS is pause-capable. It pauses the filter wheel, and writes *paused* to the already busy action's CAR record.
9. The PS sets the CAD record's accept/reject field to *accept* and returns.
10. The PSA checks to see if the command was accepted by querying the CAD field as a result of the put callback.
11. The command was accepted and has been started to the PSA protocol returns accept to the CLL of the console caller.
12. The CLL forgets everything it ever knew about the request.
13. The console monitor for the CAR record goes off and changes the color of the light to the paused color. The operator can visually tell that the filter wheel is paused.
14. At a later time, the engineer finishes his check and the operator continues the filter wheel action.
15. Steps 2- 7 of Scenario 1 with the *continue* directive rather than the *apply* directive.
16. The part becomes a command. The PSA continues the part (sets the *continue* directive) to the CAD record.
17. The PS reacts to the setting of the CAD record, processes, and executes the CAD-specific routine that validates the command arguments.
18. The PS continues the filter wheel, and writes *busy:cl_data* to the action's CAR record.
19. The PS sets the CAD record's accept/reject field to *accept* and returns.

20 The command was accepted and has been started so the PSA returns accept to the CLL of the console caller.

21 The CLL forgets everything it ever knew about the request.

22 The console is monitoring the filter wheel's action variable and its callback responds showing the filter wheel becomes busy.

23 Eventually, the IOC completes the filter action and writes *done* to the filter wheel CAR.

24 The console monitor for the CAR record goes off and changes the color of the light back to green.

25 The CLL Monitor Protocol notes that no one has interrupted the action.

Note that when the system is paused the IOC writes *pause* and not *pause:cl_data* to the CAR record for the action. This assumes that the only entity who will pause an action is the entity that started the action. If a second console paused an action started by the first console no one would know. This is viewed as an acceptable trade-off over requiring the IOC to write the cl_data with *paused.* See the next scenario for what happens when a second console continues the first console's action.

*Scenario 16*

**Short Description**: An observer interactively uses a console to set an instrument filter wheel that takes a long time to complete. An engineer asks the observer to pause the filter wheel before it has completed its motion so he can inspect the mechanism. Later some other operator with another console notices the filter wheel is paused and pushes continue.

**Run-through.** I won't go through the steps again because this is very similar to other scenarios. In this case, the first operator is notified that someone continued an action they had paused. The CLL protocol receives the BUSY:cl_data monitor and notes that some process other than itself continued the action. This could be shown as an error.

Access/permission policies are built upon the PSA interface and CAR monitoring protocol. Clients are checked and stopped by the PSA in software layers above the PSA principal systems interface.

Of course, the OCS knows little about other principal systems communicating between each other because they do not go through the OCS CLL. It is impossible for the OCS to deal with all the possible scenarios.

*Scenario 17*

**Short Description**: An observer uses a console to set an instrument filter wheel and applies the request. While the action is underway, the filter wheel belt breaks and the instrument is sophisticated enough to note it. What happens?

**Run-through.** This scenario is used to show how an error which occurs during an operator-initiated action shows itself in the system and also how the other PS must set the error condition.

1 The observer modifies the screen to indicate the new filter value.

2 Steps 2- 15 of Scenario 1.

3 Some brief time later, the console light shows that command has been accepted. The light on the screen has gone from green to red indicating the filter wheel is moving.

4 While the filter wheel is moving (before it completes and the action is busy), the filter wheel belt breaks and the hardware which has a sense switch on the belt closes. The IOC notices the filter wheel belt has broken through MBBI status record.

5 The IOC writes "Filter belt busted." to the filter movement CAR record message_in field.

6 The IOC sets the CAR record status_in to *error* causing the monitors to fire. The CAR record stays in the ERROR state.

7 If the IOC has a "belt status" SIR record, it also sets the "broken" value and causes the SIR alarm.

8 The IOC health SIR should go to the "BAD" state.

9 The instrument console shows the filter wheel action has failed.

10 The Alarm Manager Program "beeps." and opens showing an instrument failure.

11 The system status display shows the health of the instrument to be “BAD.”

12 The operator goes and gets a replacement belt and makes the world right again.

*Scenario 18*

**Short Description**: During the course of the night the TCS starts various actions on its own. Every 15 minutes the TCS checks the telescope elevation and reshapes the primary mirror. How does this show up in the Primary Mirror Console and the OCS in general?

**Run-through.** All previous scenarios have involved operator or OCS initiated actions. These requests/commands go through the OCS PSAs and CLL. When an IOC starts an action we call it a self-initiated action. The principal system itself is viewed as just another client starting actions. Therefore, the IOC must still set the CAR record with its *cl_data* so that consoles can monitor self-initiated actions too.

1 The 15 minute timer goes off and the TCS checks the elevation of the telescope and starts the mirror shaping action.

2 The IOC writes *busy:tcs_id* for self-initiated action busy to the mirror reconfigure action and begins the reconfigure process. Tcs_id is just a synonym for whatever name the TCS has.

3 The Primary Mirror Console monitoring the mirror reconfigure CAR lights the *busy* color possibly noting in a special way that the busy is from a self-initiated action. (It might know the TCS cl_data.)

4 The operator smiles warmly knowing the mirror is reshaping itself on schedule.

5 If no one has previously commanded the mirror reconfigure CAR the PSA gets no notification and no one even cares.

6 Eventually the reconfigure action is completed and the IOC sets the action CAR with the *done* causing the CAR state to go to IDLE.

7 The Primary Mirror Console display shows the green completed color.

**Note.** If an IOC modifies its own ongoing action, it must also send *busy:cl_data* to cause the MODIFY state to briefly show.

*Scenario 19*

**Short Description**: An operator makes a request that results in an action that takes a long time to complete. While the action is busy, the IOC which periodically triggers the same action, starts its action interrupting the operator’s action.

**Run-through.** In this scenario, a self-initiated action starts up before the operator-initiated action is complete. Conceivably this might never happen or there might be IOC code that permits it, but the CLL monitor protocol must handle it properly. In this case the operator receives a message telling him that the self-initiating action interrupted his action.

1 The operator pushes the reconfigure button because he thinks that the mirror needs it.

2 Steps 2- 15 of Scenario 1.

3 The Primary Mirror Console monitoring the mirror reconfigure CAR lights the *busy* color.

4 The 15 minute timer goes off and the TCS checks the elevation of the telescope and starts the mirror shaping action.

5 The IOC writes *busy:tcs_id* for self-initiated action busy to the mirror reconfigure action and begins the reconfigure process.

6 The CLL receives an *modified:tcs_id* and *busy:tcs_id* from the mirror reconfigure CAR. The CLL notifies the operator that his ongoing mirror reconfigure action was interrupted/modified by the IOCs action.

7 The Primary Mirror Console monitoring the mirror reconfigure CAR briefly lights *modified* and then goes back to the *busy* color.

8 Eventually the reconfigure action is completed and the IOC sets the action CAR with the *done* causing the CAR state to go to IDLE.

9 The Primary Mirror Console display shows the green completed color.

**Note.** The CLL Monitor Protocol will not notify an operator of interruptions of an action unless that application has started the action.

*Scenario 20*

**Short Description**: An IOC starts an action as a result of some internal computations. The action takes a long time to complete. The operator decides he wishes to control the same action and sends a command to the IOC that is accepted by the CAD.

**Run-through.** The difference between this scenario and Scenario 18 is that in this scenario, before the self-initiated action completes, the operator uses a console to command the same action the IOC has self-initiated causing the self-initiated action to be modified. The IOC gets no notification since there is no one to notify. The TCS is not using the OCS CLL. Once the operator-initiated action is accepted, the console will see the *modified* action light go on briefly.

1 The 15 minute timer goes off and the TCS checks the elevation of the telescope and starts the mirror shaping action.

2 The IOC writes *busy:tcs_id* for self-initiated action busy to the mirror reconfigure action and begins the reconfigure process.

3 The Primary Mirror Console monitoring the mirror reconfigure CAR lights the *busy* color.

4 The operator pushes the reconfigure mirror button anyway.

5 Steps 2- 15 of Scenario 1.

6 When the IOC receives the operator's command it accepts it and restarts the mirror reconfigure operation. It sets the mirror reconfigure CAR *busy:cl_data* when it restarts the process.

7 Eventually the reconfigure action is completed and the IOC sets the action CAR with the *done* causing the CAR state to go to IDLE.

8 The Primary Mirror Console display shows the green completed color.

9 The CLL Monitor Protocol notes that no one has interrupted the action.

### 6.2.2 Scripting Language Shell Scenarios

The scripting language of the system is not yet designed at this phase of the project so the first few steps of these scenarios are best guesses. The requirements of the scripting language are independent of the actual language syntax.

*Scenario 21*

**Short Description**: A script is written that offsets the telescope. The author is not concerned with completion of the offset and the next line of the script is executed immediately.

**Run-through.** A single script would consist of many lines of language constructs. Some of the lines contain commands to our systems. When one command is executed from a scripting language the generated command is the same as a console command. In this run-through it is assumed that TCL is used and that there is a software layer below the scripting language and above the CLL that translates TCL commands to sequence commands.

1 The script language interpreter interprets the following:

```
set tcs:offset:x 20
set tcs:offset:y 20
set tcs:offset:units: arcsecs
put tcs:offset
```

2 The first three lines build an attribute/value set. The last line causes the configuration part to be sent to the TCS.

3 The "middle layer" does any modification of the part and makes the call to the CLL to send the part.

4 Steps 2- 14 of Scenario 1.

5 The CLL linked to the scripting language receives the *accept* from the TCS PSA noting the command was accepted and it doesn't need to call the "rejected" callback.

6 The CLL linked to the scripting language forgets everything about the request.

*Scenario 22*

**Short Description**: A script is written that offsets the telescope. The next line in the script opens the instrument shutter so the script author needs to know that the offset is completed before the next line of the script is executed.

**Run-through.** Scripting languages will often have to wait for completion of actions before continuing interpretation. This scenario is the same as Scenario 21 but this script waits. This is what we have called a *synchronous wait* because the script blocks right after executing the command it needs to wait for until the action is complete. A new wait command is added.

1 The script language interpreter interprets the following:

```
set tcs:offset:x 20
set tcs:offset:y 20
set tcs:offset:units: arcsecs
waitnow put tcs:offset
set ics:filter:value blue
put ics:filter
```

2 The first three lines build an attribute/value set. The last line causes the configuration part to be sent to the TCS.

3 The "middle layer" does any modification of the part and makes the call to the CLL to send the part.

4 Since the *waitnow* command has been used, the CLL stores information on the request and blocks until the offset action completes. The "middle layer" provides the CLL with a "IDLE" callback (as well as an exception callback to handle: MODIFIED, PAUSED, CONTINUE, ERROR, etc. It also supplies a "rejection" callback.

5 Steps 2- 14 of Scenario 1.

6 The CLL linked to the scripting language receives the *accept* from the TCS PSA noting the command was accepted and it doesn't need to call the "rejected" callback.

7 Eventually, the CLL receives completion notification from the ARD monitor. The CLL notes that the action was not interrupted. The script unblocks and script interpretation continues.

*Scenario 23*

**Short Description**: A script is written that offsets the telescope. The script needs to wait for the offset to complete but the writer wants to go ahead and execute some other commands before "joining" the offset in progress and waiting.

**Run-through.** This scenario is the same as Scenario 22 but this script executes some other commands before waiting. This will probably require a different wait primitive that would save some "wait-state" that would allow the join to happen later. Two approaches are possible and probably necessary. One would allow a callback and one would allow a wait in the script. Both are sketched out below.

1 The script language interpreter interprets the following:

```
set tcs:offset:x 20
set tcs:offset:y 20
set tcs:offset:units: arcsecs
waitlater put tcs:offset callback-proc-name
set ics:filter:value blue
put ics:filter
```

The callback proc “callback-proc-name” is called when tcs:offset completes. Or:

```
set tcs:offset:x 20
set tcs:offset:y 20
set tcs:offset:units: arcsecs
set waitstate [waitlater put tcs:offset]
set ics:filter:value blue
put ics:filter
waitnow $waitstate
```

2 The first three lines build an attribute/value set. The last line causes the configuration part to be sent to the TCS.

3 The “middle layer” does any modification of the part and makes the call to the CLL to send the part.

4 In the first example, the CLL notes that it must provide completion notification for tcs:offset and saves the name of the callback function.

5 In the second example, the CLL notes that it must provide completion notification for tcs:offset and it passes a handle back to the script that can be used in a later command. This is common in TCL.

6 Steps 2- 14 of Scenario 1 possibly multiple times.

7 Eventually, the CLL receives completion notification from the ARD. The CLL protocol notes that the action was not interrupted.

8 In the first case, the callback function is called by the CLL.

9 In the second case, the script is unblocked and execution continues.

10 The CLL throws away knowledge of the waiting.

A number of scenarios could be generated for the various CAR conditions that can occur while a script is running. These scenarios may be extended as part of the Instrument Console Track. These three are enough to indicate what features are needed in the CLL to support scripting and to show how scripting can work.

### 6.2.3 *Sequence Executor Scenarios*

The Sequence Executor adds the requirement that the CLL/IOI synthesize completion information for configurations from the various principal system configurations and configuration parts that make of the configuration. This feature can be built upon the waiting primitives required for simple scripting in the previous section. Callbacks are made or scripts block until a group of actions complete rather than single actions.

It should also be possible to execute a complete configuration and not wait for it to complete although this feature will be less used.

These scenarios are started here but will be completed during the detailed design for the planned observing track and after the syntax for configuration is known.

Here is an example of a configuration TCL script.

```
set x [config new]
$x add {tcs:offset:x 20}
$x add {tcs:offset:y 20}
$x add {tcs:offset:units: arcsecs}
$x add {ics:filter:value blue}
waitlater put $x callback-proc-name
config destroy $x
```

The callback proc “callback-proc-name” is called when configuration x completes. Or:

```
set x [config new]
$x add {tcs:offset:x 20}
$x add {tcs:offset:y 20}
$x add {tcs:offset:units: arcsecs}
$x add {ics:filter:value blue}
set waitstate [waitlater put $x callback-proc-name]
set ics:otherfilter:value bb239
put ics:filter
waitnow $waitstate
config destroy $x
```

The configuration syntax is just a generalization of the single part syntax.

*Scenario 24* **Short Description**: A sequence executor sends an APPLY command with a configuration which contains components for just one principal system.

**Run-through.** This scenario shows how configurations for one PS are constructed.

*Scenario 25* **Short Description**: A sequence executor sends an APPLY command with a configuration which contains components for multiple principal systems.

**Run-through.** This scenario shows how the CLL combines PS configuration to wait for a complete configuration to complete.

*Scenario 26* **Short Description**: An observation is ongoing. The observer sees clouds and wishes to pause the currently executing observation.

**Run-through.** This scenario shows how the PAUSE sequence command is used with an ongoing sequence command.

*Scenario 27* **Short Description**: An observation is paused. The clouds are gone and the operator wishes to continue the observation.

*Scenario 28* **Short Description**: An observation is paused. The clouds are gone and the operator wishes to continue the observation.

*Scenario 29* **Short Description**: An observation is paused. The clouds are gone and the operator wishes to continue the observation.

*Scenario 30* **Short Description**: An observation is paused. The clouds are gone and the operator wishes to continue the observation.

*Scenario 31* **Short Description**: An observation is paused. The clouds are gone and the operator wishes to continue the observation.

### 6.2.4 OCS Communication Scenarios

Waiting for the OCS detailed design.

# 7 IOI Track Design Decisions

*This chapter presents further information on a number of IOI track design decisions.*

## 7.1 Introduction

This document describes the issues and trade-offs that were considered by the OCS group that resulted in the three tiered IOI design. This chapter assumes you have read the other chapters. The IOI itself is based upon earlier work by the Gemini Controls Group which produced the command layering model [3] and [4]. The rationale behind this model is not discussed here.

## 7.2 Goals of Design

The following are design goals for the IOI.

- The IOI should be lightweight—simple, understandable, and quick.

  The IOI is the core upon which the entire OCS software is built. It is extremely important that it provide its functionality as simply as possible without imposing excess restrictions upon the layers above it. Information passing through the IOI software layer must pass through quickly requiring minimal mandatory processing. This is what is meant by lightweight. In a sense the IOI track is the OCS kernel; it provides the basic, required OCS communication functionality and that is all. The quality of being lightweight is subjective; however, it is very clear when a piece of software is not lightweight.

- The IOI should be matched to the functionality provided by the other principal systems. It is assumed that the majority of the PS are EPICS-based. The IOI is designed to work with an EPICS or an EPICS-like system (i.e. one that mimics the behavior of an EPICS system).

  Knowing that the majority of the other principal systems are EPICS-based it would be foolish to construct an OCS software system that did its business in a way that made communication with EPICS difficult. The features of EPICS must be used to the advantage of the OCS whenever possible. However, the OCS design [1] and requirements [9] contain features that are either currently unsupported in EPICS, are supported in EPICS in ways that are not useful to the OCS, or are very specific to the OCS required functionality and are not relevant to EPICS. For example.

- The OCS has a requirement for completion notification for commands from consoles, scripting language shells, and script executors. EPICS completion is based upon status values.
- The OCS has a requirement for verification of command arguments that is not supported in EPICS in a consistent way.
- The OCS has requirement for flexible and changing access to systems and hardware subsystems. EPICS access and security is not terribly flexible.
- The OCS is dynamic and needs to create and destroy status information during an observing session; this is currently not possible in an IOC.
- The OCS programs must communicate with one another and they are not EPICS-based.

  These features drive the need for the IOI functionality in the OCS.

## 7.3 Design for Principal System Communication

The IOI track must provide the software interface between the OCS and the other principal systems. The IOI parts are shown in Figure 7 - 1.

The two new parts of the OCS that are part of the IOI track are the Command Layer Library and the Principal System Agent. These two parts are built upon the features of the Status/Alarm Database and the Action/Response Database to provide the three kinds of OCS applications with the capability of communicating with the other principal systems in the ways required by their specific functionality.

The Command Layer Library (CLL) provides OCS applications with a programming API (a shell program built upon the IOI would provide a scripting environment) that can be used to communicate with the other principal systems.

The CLL accepts sequence commands (opcode with configuration argument) from an application, sends them to the appropriate Principal System Agent, and waits for their immediate acceptance, rejection, or completion. Acceptance means that the principal system can do what it is requested to do and has begun the request; rejection means that it can't do it. Immediate completion means that the operation is so fast that it can be done immediately and that it is all finished.

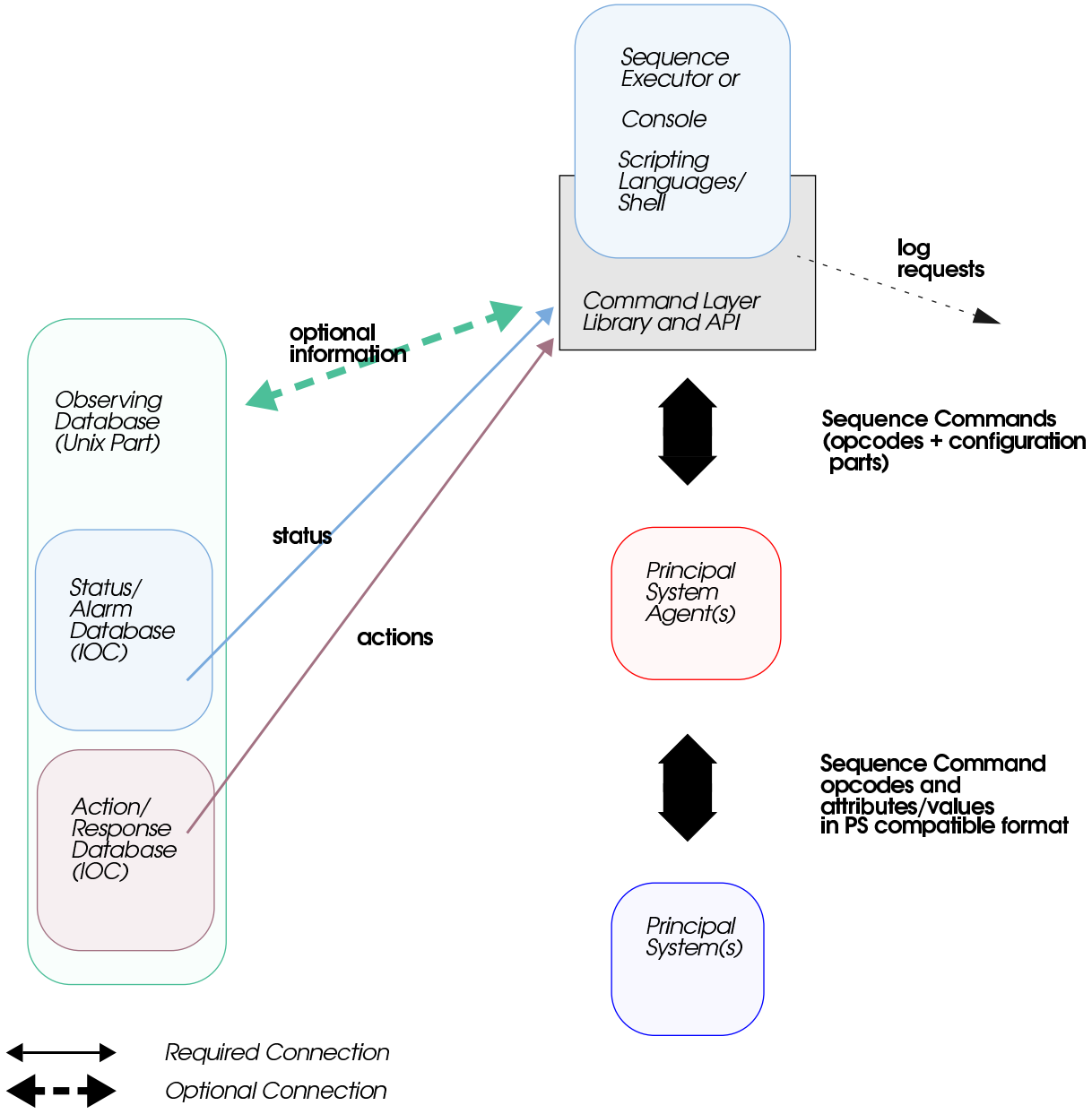
The Principal System Agent (PSA) process, of which there is one per principal system, provides the Command Layer to Attribute/Value layer interface of the behavioral model. It provides the translation, if required, of configurations to the attribute/value interface of a particular principal system. The PSA encapsulates the peculiarities of a specific principal system and hides it from the rest of the OCS software.

A PSA accepts sequence command and configuration part information from the CLL and maps it to the attribute/value layer of the principal system. It then executes the configuration and waits for acceptance, rejection, or immediate completion from the principal system. It then passes this information back to the calling command layer library in the original application.

This design is lightweight. For commands only one process sits between any client and a principal system. For status the CLL provides a very thin software interface above the status/alarm database with no intervening process.

A number of design decisions were made to come up with this design. The format is to pose a design question and then give the answer.

FIGURE 7 - 1 Software Library Environment




## 7.4 Design Issues

### 7.4.0.1 General Issues

- *What requirements do consoles place on the CLL?*

Response - Consoles are generally made up of a number of graphical elements. Some elements display status information or action information. Other elements are there to allow the console user to issue commands to the principal systems.

In our system, once a command is issued and accepted by the principal system a console no longer needs to keep track of the command. The console advises the user of the progress of the command through the presentation and changes of status and action variables (which are transparent to the console user). In most cases consoles do not have to wait for completion of the actions that they initiate. A console might wish to wait if, for instance, it had a button such as *setup* which really did several things in sequence. (Even in this situation the console would not block disallowing user input.) The action information for such a button would need to be synthesized from the action variables of the individual commands requiring a waiting functionality. (This is really the scripting requirement below.)

Consoles will present status and action variable information so the ability to monitor these kinds of variables in the SAD is required.

Console command completion is visually apparent through the monitoring of action variables. It is important for consoles users to know whether or not the actions they request have completed correctly. They also need to know if some other process modifies and action they have started. This capability is not available in basic EPICS or even the CAR record.

Summary. A console must be able to send sequence commands and monitor status and action variables. The ability to perform actions based on variable changes (monitoring) is required by consoles. A method of notifying OCS clients when their actions do not complete properly is needed.

- *What requirements do scripting language shells place on the CLL.*

Response - Some programs in our system may be shell applications that use a scripting language (PV-Wave, TCL, others) to provide access to our principal systems. The only requirement shell environments add is the need to wait for completion of operations. This occurs in scripts where the *next* command can't be executed until the current one completes. There are also cases where a command is issued and allowed to run while other script commands are executed. At a later time a scripting language call is used to allow the script to wait until completion of a previously executed command.

Summary. A shell program and scripting language adds waiting for completion of commands to the CLL requirements.

- *What requirements do sequence executors place on the CLL?*

RESPONSE - Sequence executors are specialized scripting applications. They will send sequence executor commands including configurations to multiple principal systems. They also need the ability to wait for completion of applied configurations and executor commands.

Summary: A sequence executor adds the requirement that the CLL have the ability to determine when the application of entire configurations is completed.

- *What requirements does the OCS place on the CLL?*

RESPONSE - Programs within the OCS will need to communicate with one another. An example is a console and a principal system agent. We feel that at this point in the design that the job of communicating with the other principal systems provides a set of requirements for an OCS message system that is also adequate for inter-OCS communications. These requirements are:

1. The message system must support an ability to send RPC-like request/reply messages. At this time, this mechanism is only used to sending commands/configurations and waiting for immediate acceptance/rejection/immediate completion of a command.
2. All other completion information is through action variables which will be extended for Unix-system use within the OCS. In other words, the OCS processes will provide action status for its processes too.

3. The message system must support a subscription-based or EPICS-like monitor facility to allow clients to be alerted of changes to status-like information (including actions).
4. The message size should not be limited to 8K which is the UDP message size limit.

### 7.4.0.2 *Trade-offs for IOI EPICS Interface Design*

• *Why is the PSA needed?*

RESPONSE - At this time the PSA provides two important functions.

1. It is a single point of command control for principal systems allowing centralized control of access to the PS.
2. As a single point of control it allows us to guarantee that two processes do not interleave their requests to a single CAD record. The CAD protocol requires the setting of multiple fields and two processes could use Channel Access to write to the fields of one record at the same time.

   If the item 2 problem isn't important or can be solved another way (possibly by limiting simultaneous access some other way), then the PSA could be eliminated and an OCSApp could communicate directly with a CAD through the EPICSService.

• *What approaches are there for the OCSApp-PS communication structure?*

RESPONSE - There are three approaches we can think of.

1. A CLL-no PSA approach. In this case each application can put to all EPICS fields whenever it wants. The problem with interleaved access discussed above is an issue.
2. All PSA-no CLL approach. In this case configurations are sent to the PSA and processed. The PSA handles synthesizing completion for the configurations and sends completion messages to all who are interested.
3. A system that shares duties between the PSA and the CLL. Monitoring of action variables and command completion is done in the CLL, and the sending of the configuration parts is handled by the PSA.

   Choice three is what is described in the PDR documents. The reasons for doing this are:

1. We wanted to use "as much of the EPICS features" as possible. This means an application does its monitoring (in the CLL) using Channel Access rather than having a single process do the monitoring. This might be seen as a performance issue although there is no reason to believe that putting completion determination in a PSA would cause a performance problem.
2. We felt dealing with the interleave problem is important and using the PSA as the single point where commands are actually sent to the PS solves the problem.

• *What are the advantages to having a linked library versus an OCS command server?*

RESPONSE - Among other things, the communication facility must allow the clients to communicate with the other entities in the software system. To do this the specific functions a communication facility must provide include:

1. It must split configurations based on principal system.
2. It must forward principal system configurations to the principal systems.
3. It must return information on acceptance/rejection/immediate completion to the caller.
4. It must occasionally wait for completion of configurations or parts of configurations.
5. The library must provide an abstract interface to the SAD and ARD.

   A command server can be viewed as a more capable Principal System Agent. The command server (CS) in the OCS approach would consist of a single process that accepts configurations and sequence commands from the

applications and forwards them to the appropriate principal system. Each client still must include a relatively small CLL library that allows it to connect directly to the single command server. The command server would do the things above within its own process and communicate with other processes through the OCS message system.

Another approach is to include some parts of the above functionality in a code library that is linked as part of every application. That application and the library would communicate directly with the principal system agents requiring no intermediate process.

No other possible solutions are known at this time.

Trade-offs for a CS:

-- A CS is a separate process and allows most of the functionality to be centralized in a single process.
-- A CS adds a required communication for every message and every returned response. (PSA also requires this for commands.)
-- A CS would need to monitor all the action variables for all configurations that come through it. Client programs would also need to set up monitors to status variables as well as all the action variables since consoles will be displaying action information. A CS doubles the number of action variable monitors.
-- If a CS is reasonable it is probably true that there should be a status information server that sits above the SAD.
-- The CS limits the EPICS-specific code.

Trade-offs for a linked implementation:

-- A static library is difficult to change requiring recompilation of all programs when the library is updated.
-- A library implementation simplifies (less code to maintain) the communication and integrates status and command functions in one place.

The traditional problem with a linked library is that whenever you change anything in a library, everything that links with that library must be recompiled. Many Unix versions including Solaris now support dynamically linked libraries that allow unresolved references within an application to be resolved at run-time. In addition, this library can be replaced with updated versions without recompiling programs. Of course this is only true when the library API does not change.

We have chosen to go with a dynamically linked library because it is a more efficient implementation (one less process) and it centralizes status and command functionality in one place. The dynamic linked library essentially provides the features of the separate process without the interprocess communication. Commands still must pass through a PSA so our solution is a combination of a command server with a shared library.

*• Should the OCS/IOI hide the fact that the principal systems are implemented in EPICS?*

RESPONSE - EPICS is a major part of the Gemini Control System but we believe that it is bad software design to let the EPICS Channel Access interface migrate up too far into the OCS. For this reason our CLL API will hide EPICS from the OCS client programs. Client applications will not know they are communicating with EPICS systems. The functionality of our system is modeled/matched to EPICS however and keeping with the goals of the IOI there will be a lightweight software interface to EPICS functionality whenever possible. Decisions on how to use and build upon EPICS are very important and difficult to make.

However the communications within the OCS are modeled on the functionality of Channel Access. All processes “monitor” status variables and are notified when things change just like Channel Access.

• *What is the nature of the status/alarm database interface?*

RESPONSE - The CLL must provide OCS clients with the ability to monitor changes in EPICS variables in the SAD. Client programs will call functions in the CLL API to monitor attributes which are part of the principal system's published public interface (PDF file). Client programs will not make channel access calls.

Our system must also support status variables that are created on the fly. There are a number of things in the OCS that are created and destroyed during an observing session which can also have public status values. Examples are observations, sequence executors, and observing sessions. This fact argues for an interface for status information that is layered above the EPICS status information interface.

• *How should the action/response database be used?*

RESPONSE - The action/response database is used by the CLL to determine configuration/command completion for clients. In addition, consoles can monitor action variables to show progress of actions caused by user interactions with consoles

• *What information should pass between the CLL and the PSA?*

• *And why not send an entire configuration to a PSA and let it worry about it?*

- RESPONSE - A PSA handles the commands for one principal system so the CLL can either send an opcode and entire principal system configuration to a PSA or an opcode and a configuration part. A configuration part is the set of attributes/values that are required for just one principal system command.

Trade-offs?

-- Sending an entire configuration sounds good from a design viewpoint and it has some nice features. It allows the logic for splitting configurations into parts to be placed in the PSA rather than the CLL making a simpler CLL. And it should be faster to send two big messages rather than 2N little ones where N is the number of configuration parts.

-- However, it makes for a more complicated PSA and the communication between the CLL and the PSA can be more complicated. When the PSA executes just one part it is simple to return the accept/reject/immediate completion status (action response). The protocol is a just a request/response and no bookkeeping is required on either end. When a configuration is sent, the PSA must create a message that describes the action response for all the parts of the configuration. Then the CLL must parse the message, etc. leading to more complicated software. There are also issues that must be dealt with when a part halfway through a configuration is rejected. Do the other parts get executed? Are the completed ones rolled back? Some of these issues are simpler to deal with on the client side rather than in the PSA.

-- If a client wanted to wait for completion of a configuration, a PSA-based solution would either have to monitor all the action variables for the configuration (and all the other outstanding configurations) and send completion to the original application which issued the configuration or the structures required to wait would be redundantly contained in the CLL since the CLL would still need to break open the configuration to get at the action variables anyway to display action information.

The initial design of the CLA will send one configuration at a time to the PSA. The CLA will break the configurations into principal system configurations and then into parts. The CLL will send the parts one at a time to a PSA and await the action response before going on. Sending an entire principal system configuration might be faster, but it would be somewhat more complex. As a first pass, we choose simplicity over speed. Using the PSA this way is something that we will examine during prototyping.

• *What communication method is used between the PSA and the PS?*

RESPONSE - The communication between a PSA and its PS will generally be EPICS channel access, but the PSA concept allows the OCS design to deal with unusual systems if it needs to. This may be useful in the future when visitor instruments come to the telescopes. A visitor instrument would modify a PSA to communicate with their peculiar system.

- *What does the PSA monitor/callback protocol add to our system?*

RESPONSE - The EPICS system is value based but determining completion based on values in clients is a bad software design. Our system uses action variables to determine when actions complete. It must be possible to tie actions to commands to allow sequencing of our systems by the OCS. The PSA action protocol allows the OCS to control the other principal systems more reliably because it allows the OCS to determine when actions are modified during execution. Using the PSA action variable protocol, if clients are not notified during execution, they can rely on the results of successfully completed actions.

It should be noted that the entire action variable method relies on principal systems performing the actions they are requested and only returning to IDLE when they have successfully done what they were requested to do.

Created:July 20, 1995
Modified:March 18, 1996

Gemini Observatory Control System Report

# Planned Observing Support Track Preliminary Design

**Kim Gillies, Shane Walker, Steve Wampler**

ocs.ocs.004-PlannedTrkPD/03

**This report presents the preliminary design for the Planned Observing Support Track.**

## 1.0 Introduction

The Planned Observing Support Track is one of the development tracks in the development plan for the Observatory Control System (OCS) of the Gemini Telescope. The following statements are from the Software Design Review OCS Development Plan [4].

> This second phase of infrastructure adds the functionality required for the planned observing modes. The majority of the Configurable Control System is developed here including the Observing Database, and the various processes and structures required to use Science Programs and the Observing Tool. (page 3).
>
> These two phases must be completed before automatic control of the telescope configuration is possible. This functionality is required for the operational phase planned observing capabilities (page 7).

The Planned Observing Support (POS) track is primarily the development of the Configurable Control System. This track must provide the software support that will allow the GCS to be used effectively in the planned observing modes. The products of the POS include infrastructure-related applications and libraries as well as operator applications that are part of the visible user interface.

This report presents the preliminary design of the POS track to a depth such that the track can continue on in its development independently of the other OCS tracks. The following information is contained in this report.

- A high-level preliminary design for the track.
- The message/event flow between applications in the POS track and the other OCS tracks.
- The message/event flow between applications in the POS track and the other principal systems.
- A list of the known POS products.
- Remaining decisions for the detailed design of the track.

## 2.0 Acronyms

| | |
|---|---|
| API | Application Programmer Interface |
| CLL | Command Layer Library |
| EC | Executor Controller |
| GUI | Graphical User Interface |
| IOI | Interactive Observing Infrastructure |
| OCS | Observatory Control System |
| ODB | Observing Database |
| OT | Observing Tool |
| PD | Preliminary Design |
| POS | Planned Observing Support |
| PS | Principal System |
| PSA | Principal System Agent |
| SAD | Status Alarm Database |
| SE | Sequence Executor |
| SIR | Status Information Record |
| SM | Session Manager |

## 3.0 References


[1] SPE-C-G0037, *Software Design Description*, 9/9/94, Gemini Controls Group, 1994.

[2] ocs.kkg.032, *Interactive Observing Infrastructure Preliminary Design*, Kim Gillies, Shane Walker, Steve Wampler, Gemini Observatory Control System Group, 1995.

[3] ocs.kkg.014, *Observatory Control System Software Requirements Document*, 6/5/95, Gemini Observatory Control System Group, 1995.

[4] ocs.kkg.016/06, *Observatory Control System Development Plan*, Gemini Observatory Control System Group, 1995.

[5] *ESO Graphical User Interface Common Conventions*, Doc. No. GEN-SPE-ESO-00000-0266, Issue 1.0, 10/05/94.

[6] ocs.ocs.002, *OCS Physical Model Description*, Gemini Observatory Control System Group, 1995.

[7] ocs._sw.004, *Observing Tool Track Preliminary Design*, Gemini Observatory Control System Group, 1995.

[8] *Object-Oriented Modeling and Design*, James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, Prentice-Hall, 1991.

[9] ocs.kkg.033, *Telescope Control Console Track Preliminary Design*, Gemini Observatory Control System Group, 1995.

[10] ocs.kkg.031, *Preliminary Definitions of Sequence Commands*, Gemini Observatory Control System Group, 1995.


## 4.0 Document Revision History

**First Release —** August 5, 1995. Pre-release draft.

**Final PDR Release —** 27 September, 1995.

# 5.0 Studies/Decisions During POS Track

The following design studies or trade studies were specified during the OCS SDR to be done during the POS track.

## 5.1 Trade Studies

**Observing Database.** A study will be required to examine the OCS requirements for the non-EPICS parts of the run-time Observing Database and the external database. A study will be required to examine the requirements and choose an implementation. The study must discuss the relationship between the Observing Database and the external database that is part of the DHS.

## 5.2 Design Reports

**Possible Benefits of Multi-processors.** Two multiprocessor Sparcstations have been purchased for use during the development of the OCS and for eventual site use. A study/report will be done to summarize and make recommendations for multi-processor machine use in the Gemini Control System.

**Remote Monitoring Requirements.** The prototype remote monitoring interface describes a system that allows remote observers to view live video and scanned images. Sound transmission capability between the sites and remote observers is also suggested. A study must be done to determine which (if any) commercial product can be used to provide these capabilities. The result will be a recommendation for a product or a recommendation to limit this capability.

No other studies or reports are required for this track at this time.

# 6.0 POS Track Overview

The Planned Observing Support track is developed around the concept of an *Observing Session*. An Observing Session corresponds naturally with the activities that must take place at the telescope in order to accomplish an observing run. Support for both Observing Tool Interactive Observing and all the Planned Observing modes is developed in this track.

A *Session* is created when an observer (staff or otherwise) wishes to execute a set of observations, or *observing plan* (or just *plan*). The Session is associated with a set of system resources that are available to the observer's observations. These will include the telescope beam and four instruments at a minimum, but we use the term "resource" in a generic sense to promote a general design.

A system operator at the telescope site manages the creation of Sessions and is in control of allocating resources using the Session Manager (SM) application. Observers will contact the SM with their Observing Tool (OT) to start a Session or to monitor an existing Session. The operator can accept or reject this request, and is in charge of assigning the observer to a particular Session. Only one observer, the *blessed* observer, can actually execute an observation with the OT. Other participants assigned to the same session simply monitor run-time information.

Each Session is associated with a single observing plan. The plan details the sequence of observations to be executed and is constructed and modified with the OT. Observing plans are stored in the Observing Data-

base, and their representation is reflected in the GUI presented to the SM operator. The execution of an observing plan is controlled through the OT, but ultimately verified or accepted by the system operator sitting at the SM console.

To execute an observation, the SM uses a Sequence Executor (SE) application. Its job is to interpret the information in the observation and send the appropriate sequence commands to the various principal systems. It also records header information in the ODB and is in charge of monitoring completion after an **apply** sequence command.

The high-level physical model introduced in the next section captures and expands upon these concepts.

# 7.0 Physical Model of the POS Track

The layered physical model of the OCS is introduced in [6]. This track document also uses the Object Modelling Technique of Rumbaugh [8] to present the design of this track. The notation used in the diagrams is not explained here, see [8] or [6] for definitions.

The POS track is modelled as a group of OCSApp instances that cooperate to provide the functionality of the track. This preliminary design document is not focused on the design of the individual applications that make up the track; rather, it is focused on identifying the applications required to provide the functionality and showing how the applications cooperate. The following approach is used to achieve this.

1. Build an object model of OCSApp instances for the POS track.
2. Identify and describe the methods that each application must respond to for the functional model.
3. Examine the dynamic operation of the entire system using scenarios/use cases, object interaction diagrams, and event trace diagrams.

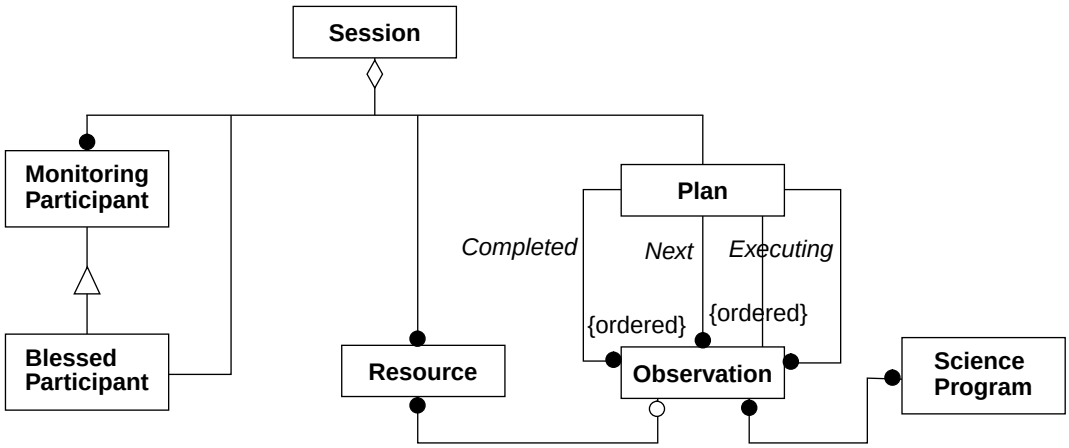
The PDR design is concerned with the external interface for each of the POS applications. The internal design of each application or application type is left for the detailed design of the track; however, the last sections of this paper present the detailed design work that was required in order to complete this document.

The first step above, developing the object model of OCSApp instances, is handled in this section. The remaining sections cover the other points. However since the Session idea drives the design, we begin by examining its structure in more detail.

## 7.1 Session Model

The Session concept is modelled as an association of participants (observers), resources, and plans (Figure 1). More than one observer can participate in a session, but a single blessed observer controls the observation plan execution at any one time. The remaining observers just monitor the run-time information provided by the SM. The blessed observer is viewed as a special kind of monitoring participant.

*FIGURE 1.* The Session Object

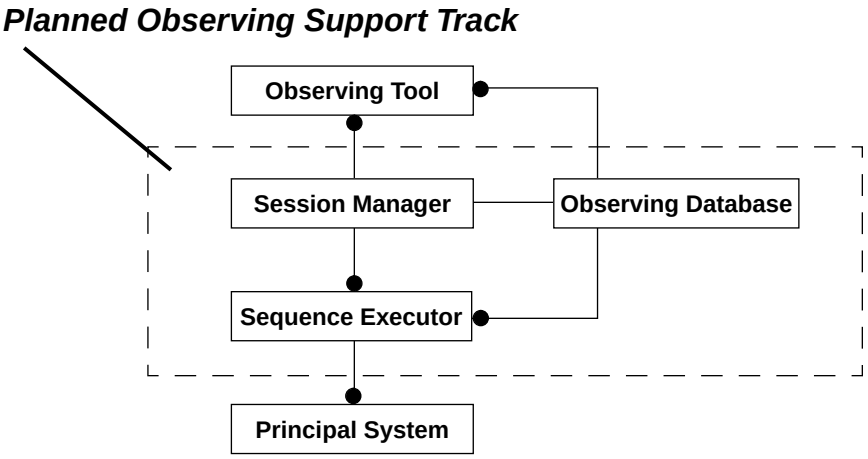



There is a one-to-one association of Sessions with Plans. Plans are composed of observations, most of which will just be links to observations in Science Programs. The observations in a plan fall into three categories, *Completed*, *Executing*, and *Next*. Completed observations have finished executing and Next observations are ahead in the queue. Both Completed and Next observations are ordered, though the arrangement of upcoming observations can be changed with the blessed observer's OT. Executing observations are associated with a subset of the resources belonging to the session. More than one observation in a given session may be executing concurrently, provided that the observations use disjoint sets of resources.

## 7.2 Application Model

The associations between the Observing Tool, Session Manager, Observing Database, and Sequence Executor are indicated in Figure 2 below. The POS Track consists of the SM, SE, and ODB. The OT application is required for planned observing as well, but it is developed separately in the Observing Tool Track [7].

*FIGURE 2.* POS applications and their interactions




The SM receives connection requests from one or more Observing Tools, and instructions from the blessed observer's OT. It sends run-time status information back to all the participants. Both the SM and OT examine the same observation plan in the ODB, and the OT is charged with creating and modifying the plan.

Sequence Executors are allocated by the SM to execute observations. They receive start/pause/continue/abort commands from the SM and return status information to be relayed to the participants. The observation that they execute is read from the ODB, and header information is stored back to the ODB as well.

The diagram shows a direct connection between the SE and the Principal Systems, though the connection goes through the Principal System Agent of the IOI track [2].

# 8.0 High-Level Design of the POS Track

The POS architecture is modelled as a set of cooperating OCSApp instances. This section takes a closer look at the responsibilities and tasks of each of the components of the architecture described in the previous section. The following section details the public interface for each of the components.

## 8.1 The Session Manager

The Session Manager (SM) is the central coordinator of the automated operations of the POS. The visible user interface of the SM is one of the important operator interfaces. The following are the functions and tasks of the SM given our design.

**Session Management.** The SM provides the operator with the ability to create and destroy observing sessions.

- The SM allows the user to associate a plan with a session.
- The SM keeps a record of its operation during observing sessions.
- The SM allows the operator to control the Observing Tool monitoring of the observations in sessions.

**Connection Management.** The SM allows users to participate in observing sessions. A user associated with an observing session is called a participant.

- The SM must support software connection requests from Observing Tools. The SM and its GUI must notify the operator of connection requests and allow him to accept or reject requests.
- The SM must allow the operator to assign one or more participants to an observing session.
- Each session must have one participant who is has the capability of interacting with the session and rearranging the plan. Others can only watch. The SM allows the operator to determine which participant is *blessed*, and to change a session's blessed participant while the session is executing.

**Resource Management.** The operator is the sole owner of all the resources in the GCS. The operator makes decisions to share his resources with observations executing in sessions. Through his actions observations gain access to all the shared observatory resources they need to run.

- The SM supports the operator by allowing him to determine which observing sessions have access to the resources of the system.
- The SM allows resources to be dedicated to sessions for long periods of time.
- The SM controls access to the principal system resources.

**Session Information Management.** All Observing Tools involved in run-time control or monitoring of observations are connected to the OCS through the SM. Consequently, the SM knows all of the participants connected to the observatory and the associations between participants and sessions.

- Status information generated by the executing observations can be routed to all of a session's participants. This status information appears in the run-time OT display.

- Status information created while managing the session can be forwarded to participants. This kind of information indicates when observations have started and ended and when they have entered or completed various recipe phases.
- The operator can send messages to one or more participants in a session.
- Any video or sound transmission will use the SM participant information. However, sound or video information would probably not be routed *through* the SM. This is a design decision for the future.

**Session Observation Execution.** Multiple observations can be executing in a single session if the sets of resources they require are disjoint. Multiple sessions can also operate concurrently.

- A Sequence Executor is a process with the job of executing a single observation. The Session Manager starts, monitors, and kills Sequence Executors as required by the contents of the session's observing plan and the session's execution policy.
- The SM execution policy (the Session Manager Recipe) determines when and how observations can be executed in a session. Examples are one observation at-a-time, one observation after another automatically, or execute as many observations concurrently as are possible. The SM allows the operator to configure the sessions observation execution recipe.

**System Integration Features.** The SM is the primary Operator user interface for planned observing and it supports a number of integration features for the operator's use.

- The operator can configure his console screens to display TCS information for any of the observations in the session.
- The operator can configure the SM execution policy to require his intervention before any observation will be executed. This supports the OCS requirement that the operator be the one individual responsible for hardware control.

## 8.2 Sequence Executor

The Sequence Executor (SE) is a subsystem with the job of executing a single observation. To accomplish this fairly complex job it has a number of capabilities and responsibilities.

**Session Commands.** The SE must accept and respond to commands to control its operation.

- An observation can be assigned to an SE for execution.
- A caller can send commands to start, stop, abort, pause, and continue an observation. These terms have the typical Gemini definitions.
- The SE associates a recipe with the execution of an observation. The recipe describes the sequence and interactions of an observing pattern.

**Resource Use.** The SE is charged with acquiring the resources an observation needs before it can execute.

- The SE examines an observation for resources.
- The SE requests resources from the SM and relinquishes them according to its recipe and/or the desire of the observer as specified in the attributes of the observation.

**Status Use and Generation.** The SE must make the system aware of its state. It also has several status-related responsibilities during the execution of an observation.

- The SE publishes status information describing its operations and state for the Session Manager and the GCS.
- The SE generates status information for the OT run-time display and forwards the information to the Session Manager.

- The SE can take snapshots of status information during observation execution and store the status data with the observation in the ODB.
- The SE sends observation status data required for the data headers to the DHS once the observation is completed.
- The SE watches for changes to important observation status information during Verify/EndVerify sequence commands. It notifies participants of changes through the Session Manager and then updates the observation in the ODB if needed.
- An SE can monitor an observation's configuration for changes while the observation is executing. It can post alarms if some parameters change when they shouldn't.

**System Interactions.** The SE is an active component in the GCS and must interact with other systems.

- The SE interacts with the GCS systems using sequence commands.
- The SE interacts with the ODB to fetch and store information associated with its observation.
- The SE interacts with the DHS when it sends it header-related status information at the conclusion of an observation.

### 8.3 Observing Database

The Observing Database (ODB) is the persistent repository of all information associated with Science Programs, Observing Plans, Observations, components of Observations. When the POS is in run-time use supporting the planned observing modes, the ODB provides several functions.

- The ODB responds to changes in Science Programs and Observing Plans related to operations performed by observers using the Observing Tool.
- The ODB can receive updates to observations from the Sequence Executors while an observation is executing.
- The ODB receives updates from the Session Manager related to session operations and changes.
- The ODB must ensure that the views of open Plans, Science Programs, and Observations in Observing Tools remain consistent in response to changes by the "blessed" participant.

## 9.0 Planned Observing Support Functional Model

Figure 3 shows an expanded Figure 2 that includes the public methods for each of the OCSApp components of the POS. The public methods are the methods that an application allows other applications to execute. The expanded object model shows the external database is used by the ODB, but OCS applications interact only with the ODB.

In the object models showing methods, + and - are used to indicate that a method is public or private. In this document *private* means the method is only available to classes *inside* an application instance. *Public* means that the method can be called by other applications to cause actions.

Figure 4 shows the important POS document data types and the associations between the documents and data. The detailed composition of the Observation is not shown.

- An Observing Plan can reference many Science Programs and a Science Program can be referenced by many Plans.
- An Observing Plan can also contain Observation objects. For instance, an on-site observer will include calibration observations which don't appear in any one user's Science Program. The ability for an Obser-

vation to appear in many Plans has been included even though it may be undesirable rather than enforcing a constraint at this time.

- Associated with each Observation object is a number of DataFile objects. These DataFiles contain the OCS data to tie the observations in a Science Program to the actual data in the DHS. The actual data is modelled as references to the external database.
- The Recipe Database is used by the POS applications. It contains the Session Recipes and the Sequence Executor Recipes. These recipes are derived from the imported File type meaning that these files will be text-based scripts that simply stored in the database

Figure 4 also shows that the Science Program object can be associated with a file. The Science Program object must be able to store itself to a traditional file and to build itself from a previously stored Science Program file. The goal is that the file will be text based.

The methods of the application classes in Figure 3 in addition to the data object methods in Figure 4 comprise the public interface for the POS. In addition, the Session Manager application will use the methods listed below to implement its GUI functionality. Though they are private to the SM, they are useful in demonstrating how the operator interacts with the SM.

- Managing Sessions (Figure 20 on page 26, Table 8 on page 27)
  -addSession(name, session)
  -getSessions()
  -getSession(name)
  -removeSession(name)
- Executing a Session (Figure 21 on page 27, Table 8 on page 27)
  -setSessionRecipe(name)
  -abortSession()
  -getSessionRecipe()
  -pauseSession()
  -startSession()
  -continueSession()
  -stopSession()
- Interacting with a Session (Figure 21 on page 27, Table 8 on page 27)
  +sendToAllParticipants(message)
  -getParticipants()
  -addParticipant(participant)
  -addResource(resource)
  -removeParticipant(participant)
  -removeResource(resource)
  -getBlessedParticipant()
  -getResourceList()
  -setBlessedParticipant(participant)
- Managing Participant Access (Figure 22 on page 30, Table 8 on page 27)
  -connectionRequest(participantName)
  -getParticipantConnInfo(participant)
  -connectionAccept(session, participant)
  -getParticipantByName(participantName)
  -connectionReject()
- Managing Resources (Figure 23 on page 31, Table 8 on page 27)
  -shareResource(resName, session)
  -unshareResource(resName)

These methods are used to demonstrate the dynamic behavior of the POS in upcoming sections.

*FIGURE 3.* POS Applications Showing Public Methods

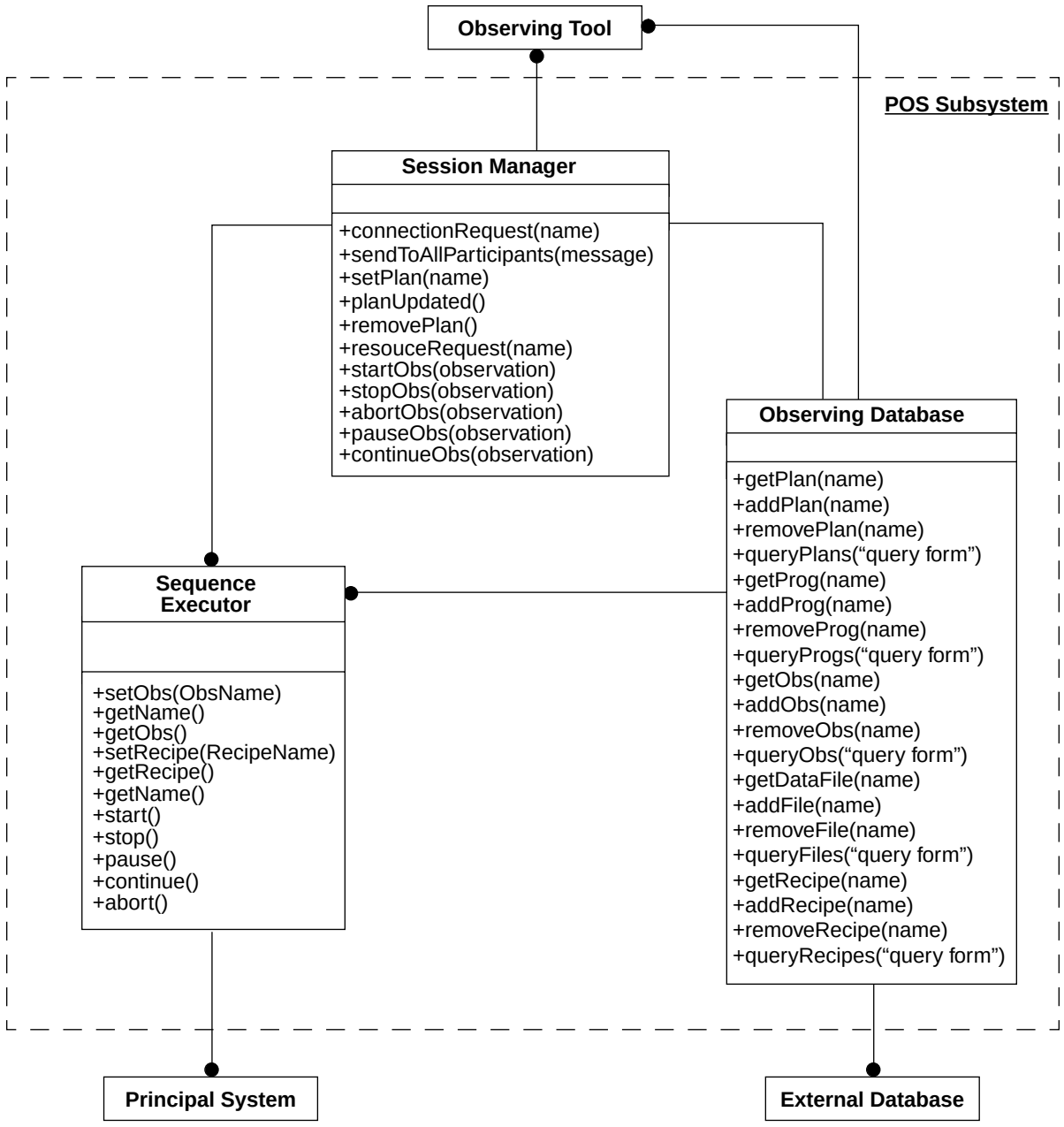
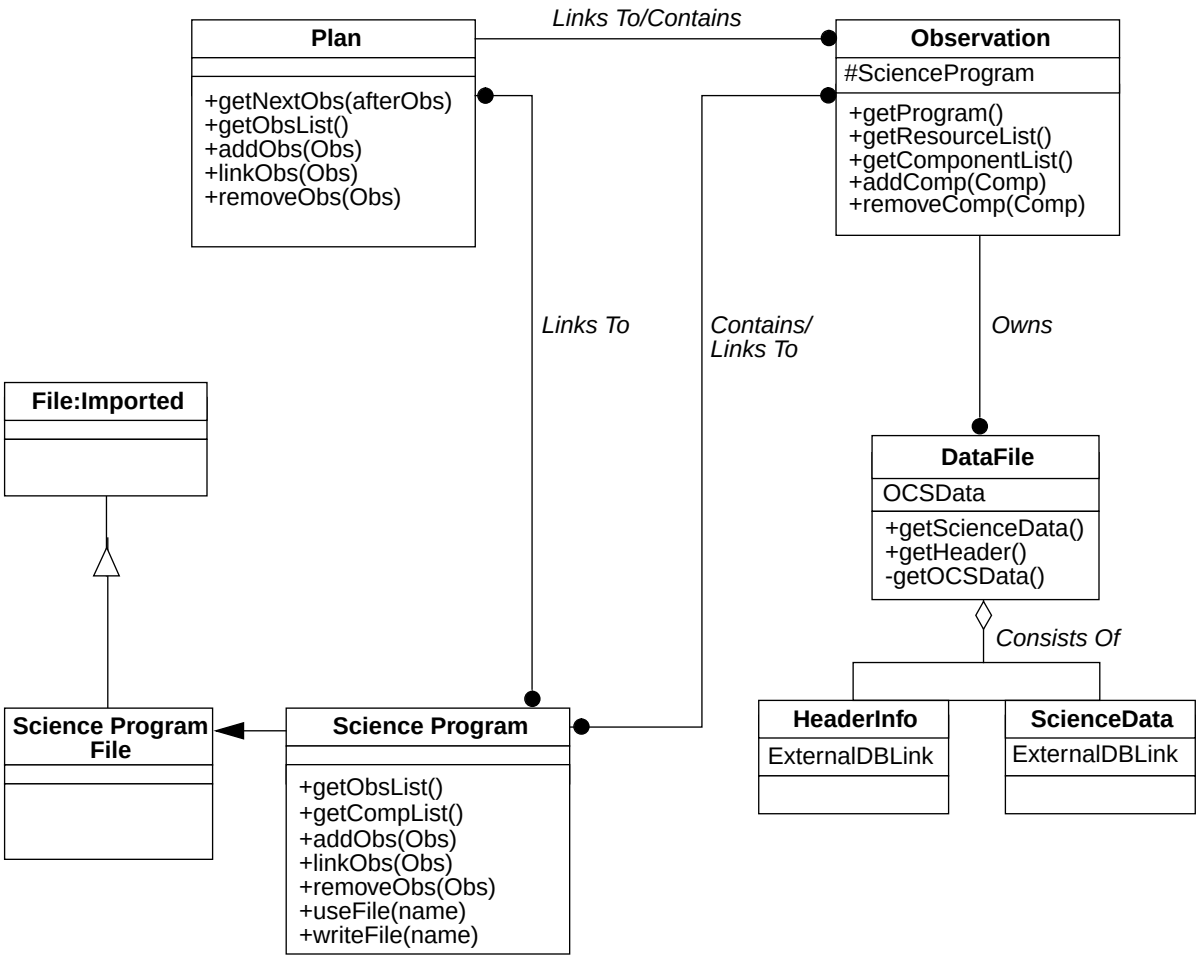


*FIGURE 4.* POS Data Associations




## 9.1 A Primitive Session Manager Prototype

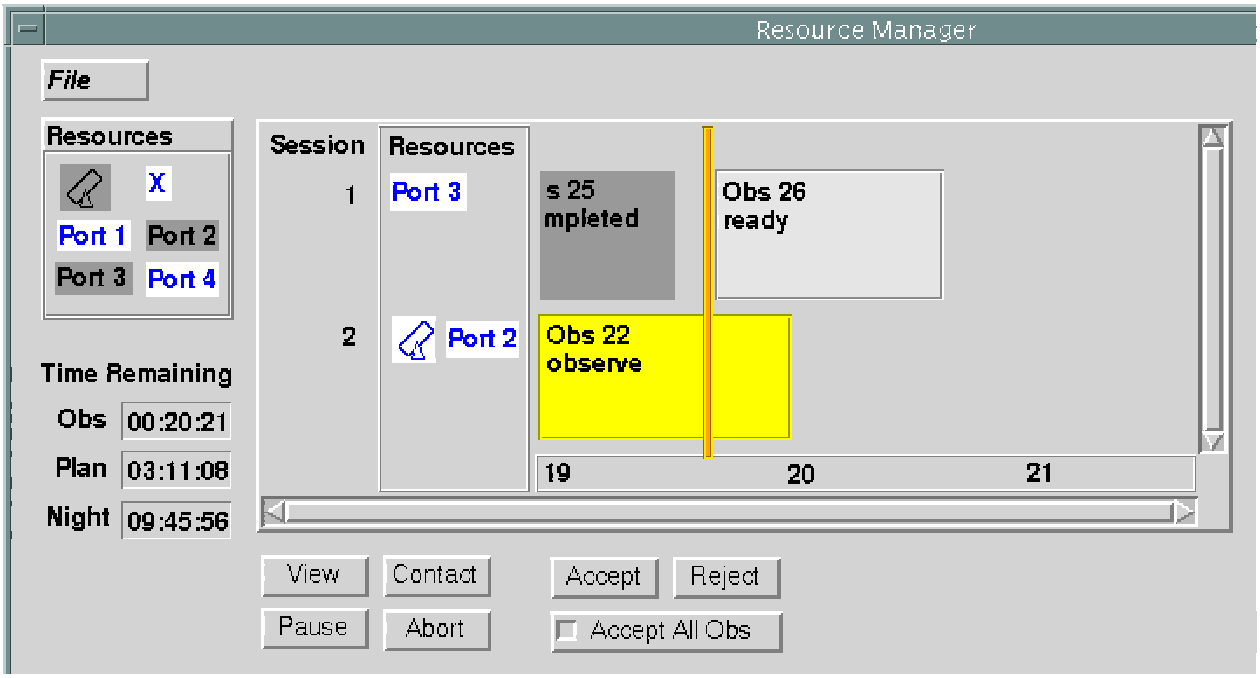
Some very preliminary work has been done on the Session Manager user interface and that work is shown in Figure 5 in order to help readers visualize what we have in mind. This screen shot is just a demo and has received very little work; the design of the GUI is guaranteed to change. The screen was done using Tk and ESO's Panel Editor.

The figure shows two horizontally displayed sessions. Along the x axis is the time and a vertical line passes through the sessions showing the current time. In Session 1, Observation 25 is completed and Observation 26 is ready to run. In Session 2, Observation 22 is executing. As time increases, the session display scrolls to the right providing a history of what has happened at the telescope.

The resources for each session are shown next to the sessions. Session 1 only uses Port 3 (ports were used as resources although we would rather refer to instruments). Session 2 uses Port 2 and the telescope beam resource. The operator shares resources by dragging icons from the Resource Pool on the left on to the Session.

The operator can view the details of an observation by selecting one of the observations. The figure shows Observation 22 selected. The sample display on the left shows that the observation has 20 minutes left and some additional information.

*FIGURE 5.* The Very Primitive Session Manager Prototype




This is the method the operator will use to evaluate and authorize an observation for execution when needed. In the prototype, he selects a Observation 22 and presses *view.* This causes the operator's telescope consoles to display the values associated with Observation. He presses *accept* to allow the observation to continue. This is just one approach to session management there will be others. The prototype shows a button that allows the operator to accept all observations in the Session without needing his intervention.

### 9.1.1 Session Manager Processor Model

The model for the Session Manager is to be split into two parts: a Session Manager GUI, and a Session Manager Server. The Session Manager Server will be located in the Configurable Control System. The Session Manager GUI will be executed from the operator's machine. The two components have been split so that the Session Manager will be available at all times. The project processor allocation calls for a single dedicated Sun machine at each site for the Configurable Control System.

## 9.2 Method Descriptions for POS Application Classes

The following tables give a brief description of the public methods available in each of the POS application types.

*TABLE 1.* Public Methods for Class Session Manager

| Method | Use |
|---|---|
| connectionRequest(Name) | The Observing Tool contacts the Session Manager using this method when an observer wishes to participate in a session. The Session Manager operator is notified and either accepts or rejects the connection. |
| sendToAllParticipants (Message) | Sends a message, such as a status update, to all participants in the session. |
| setPlan (Name) | Associate the named observing plan with the session. If there is already a plan associated with the session, then it is forgotten in favour of the new plan. |
| planUpdated() | Informs a session that its plan has been modified in the ODB. |
| removePlan() | No longer associate a plan with the session. |
| resourceRequest (Name) | The Sequence Executor uses this method to request a resource. If the session does not already have access to the resource, the system operator is notified. |
| startObs(observation) | Start an observation in a session, subject to the approval of the system operator. |
| stopObs(observation) | Stop an executing observation. |
| abortObs(observation) | Abort an executing observation. |
| pauseObs(observation) | Pause an executing observation. |
| continueObs(observation) | Continue an executing observation. |

*TABLE 2.* Public Methods for Class Observing Database

| Method | Use |
|---|---|
| getPlan(name): Plan | A reference to an Observing Plan object associated with *name* is returned. |
| addPlan(name) | This method adds Plan *name* to the Plan Database. |
| removePlan(name):Plan | This method removes the Plan *name* from the Plan Database and returns a reference to it. |
| queryPlans(“Query form”):PlanList | This method is used to select a subset of Plans from the Plan Database. The query results in a list of Plans. |
| getProg(name): ScienceProgram | A reference to a ScienceProgram object associated with *name* is returned. |
| addProg(name) | This method adds ScienceProgram *name* to the Science Program Database. |
| removeProg(name):ScienceProgram | This method removes the ScienceProgram *name* from the Science Program Database and returns a reference to it. |
| queryProgs(“Query form”): ScienceProgramList | This method is used to select a subset of SciencePrograms from the Science Program Database. The query results in a list of SciencePrograms. |
| getObs(name): Observation | A reference to an Observation object associated with *name* is returned. |
| addObs(name) | This method adds Observation *name* to the Observation Database. |
| removeObs(name):Observation | This method removes the Observation *name* from the Observation Database and returns a reference to it. |

| Method | Use |
|---|---|
| queryObss(“Query form”):ObservationList | This method is used to select a subset of Observations from the Observation Database. The query results in a list of Observations. |
| getDataFile(name): DataFile | A reference to a DataFile object associated with *name* is returned. |
| addFile(name) | This method adds DataFile *name* to the Data Database. |
| removeFile(name):DataFile | This method removes the DataFile *name* from the Data Database and returns a reference to it. |
| queryData(“Query form”):DataFileList | This method is used to select a subset of DataFiles from the Data Database. The query results in a list of DataFiles. |
| getRecipe(name): Recipe | A reference to a Recipe object associated with *name* is returned. |
| addRecipe(name) | This method adds Recipe *name* to the Recipe Database. |
| removeRecipe(name):Recipe | This method removes the Recipe *name* from the Recipe Database and returns a reference to it. |
| queryRecipes(“Query form”):RecipeList | This method is used to select a subset of Recipes from the Recipe Database. The query results in a list of Recipes. |

*TABLE 3.* Public Methods for Class Sequence Executor

| Method | Use |
|---|---|
| setObs(obsName) | This method allows the caller to set the Observation that will be executed by the SE. The name of the observation is a character string. |
| getName(): Name | This method returns the name of the observation associated with the SE. |
| getObs(): Observation | This method returns the Observation object associated with the Sequence Executor instance. |
| setRecipe(recipeName): | This method allows the caller to set the Recipe that will be used to execute the observation. The name of the Recipe is a character string. |
| getRecipe():recipeName | This method returns the name of the recipe associated with the SE. |
| getName(): Name | This public method returns the name of the recipe. |
| start() | This public method starts the execution of the observation. |
| stop() | This public method stops the execution of the observation at the next appropriate time. |
| pause() | This public method pauses the execution of the observation if it is executing. |
| continue() | This public method continues the execution of the observation if it is paused. |
| abort() | This public method causes an observation to terminate immediately. |

## 9.3 Data Object Public Methods

The ODB public interface can be used to acquire data objects. The method interface for the data object is then used to access and manipulate the data object state. The following tables give a brief description of the data object methods.

*TABLE 4.* Plan Class Methods

| Method | Use |
|---|---|
| getNextObs(afterObs):ObservationList | This method returns a list of unexecuted Observations in a Plan. |
| getObsList(): ObsList | This method returns a list of all the Observation objects in a Plan. |
| getCompList(): CompList | This method returns a list of all the Component SPItems in a Plan. |
| addObs(Obs) | This method adds an Observation object to a Plan. |
| linkObs(Obs) | An Observation object in a Science Program can be linked to a Plan using this method. |
| removeObs(Obs):Observation | An Observation is removed from a Plan using this method. |

*TABLE 5.* Science Program Class Methods

| Method | Use |
|---|---|
| getObsList(): ObsList | This method returns a list of all the Observation objects in a Science Program. |
| getCompList(): CompList | This method returns a list of all the Component SPItems in a Science Program. |
| addObs(Obs) | This method adds an Observation object to a Science Program. |
| linkObs(Obs) | An Observation object in another Science Program or a Plan can be linked to a Science Program using this method. |
| removeObs(Obs):Obs | An Observation is removed from a Science Program using this method. |

*TABLE 6.* Methods for Observation Class

| Method | Use |
|---|---|
| getProgram():ScienceProgram | Return the Science Program or Plan owner of the Observation. |
| getResourceList(): ResourceList | This method returns a list of all the Resources required by an Observation. |
| getCompList(): CompList | This method returns a list of all the Component SPItems in an Observation. |
| addComp(Comp) | This method adds a Component to an Observation. |
| removeComp(Comp):Comp | A Component is removed from the Observation and returned to the caller. |

*TABLE 7.* Methods for Class DataFile

| Method | Use |
|---|---|
| getOCSData(): OCSData | Each DataFile in the Data Database is associated with some data used by the OCS. This method returns that data. |
| getScienceData(): ScienceData | This method returns the ScienceData object associated with a DataFile. |
| getHeaderInfo(): HeaderInfo | This method returns the HeaderInfo object associated with a DataFile. |

## 10.0 POS Dynamic Model

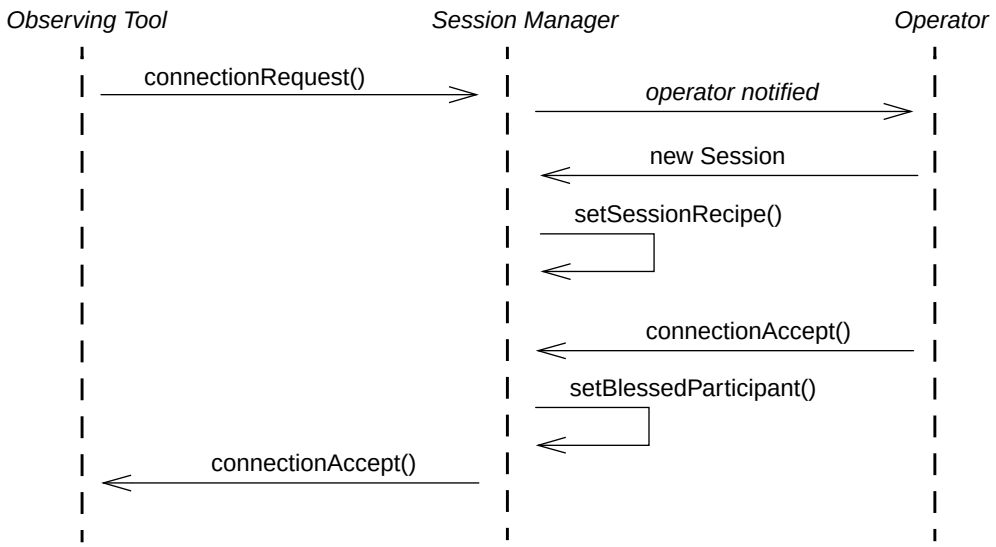
The functional interface for the application types can now be used to demonstrate the behaviour of the POS track under a number of typical scenarios. The following examples show how the design handles some typical use cases and scenarios. A use case is a simple task users will need to execute with the software system. A scenario is usually larger than a use case and can include several use cases in its description. Some scenarios will consist of text-based steps when the interactions are outside this track or covered in other track documents. The labels for each example below is CX for *Case X*.

*C1* • *An Observing Tool makes a connection request to the Session Manager to start a new Session.*

*Background.* An observer wishes to begin observing. He starts the Observing Tool and makes a connection request to the Session Manager using the GUI.

*Scenario.* An Event Diagram is shown in Figure 6. The OT makes the connection request to the known location of the Session Manager. The operator is prompted and must take an action as a result of the request. He uses the GUI to create a new Session object. The default Session Recipe is assigned to the new session. The operator assigns the participant to the new session and since he is the first participant he is blessed. The OT is notified of his acceptance.

*FIGURE 6.* Connection Event Diagram

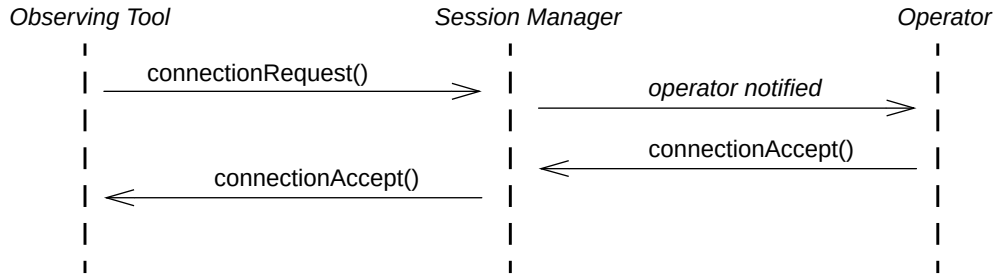



*C2* • *An Observing Tool makes a connection request to the Session Manager to join an ongoing session.*

*Background.* An remote observer wishes to join a session to monitor his observation. He starts the Observing Tool and makes a connection request to the Session Manager using the GUI.

*Scenario.* An Event Diagram is shown in Figure 7. The OT makes the connection request to the known location of the Session Manager. The operator is prompted and must take an action as a result of the request. He accepts the request and assigns the observer to a session. The OT is notified of his acceptance. He is not blessed.

*FIGURE 7.* Joining an ongoing session

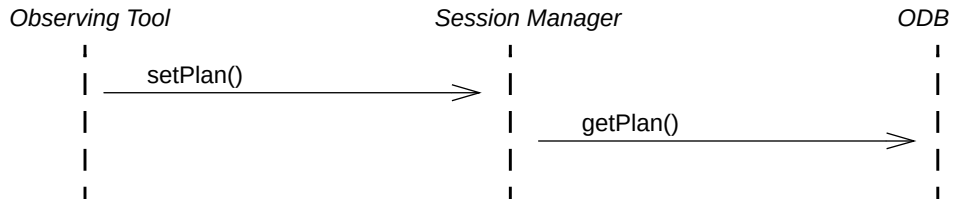



### *C3 • An on-site observer selects an Observing Plan/Science Program.*

*Background.* Once the connection is accepted, the blessed observer can set the Plan that will be examined for observations.

*Scenario.* Opening the Plan in the Observing Tool causes the name of the plan to be associated with the session. The Session Manager gets the Plan object from the ODB.

*FIGURE 8.* Setting the Plan

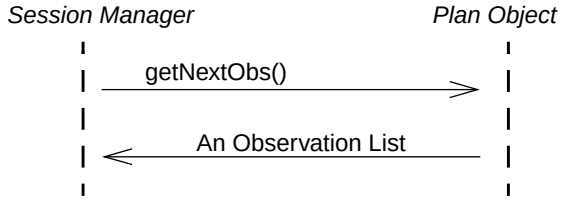



### *C4 • The Session Manager gets the next observations from the plan.*

*Background.* The Session Manager looks in the shared Plan document in the ODB to find the next observations.

*Scenario.* A list, in the observer's order, is returned to the Session manager when the getNextObs() method is sent to the Plan object. This is also how the Session Manager notices additions to and changes in the ordering of the Plan.

*FIGURE 9.* Getting the Next Observations

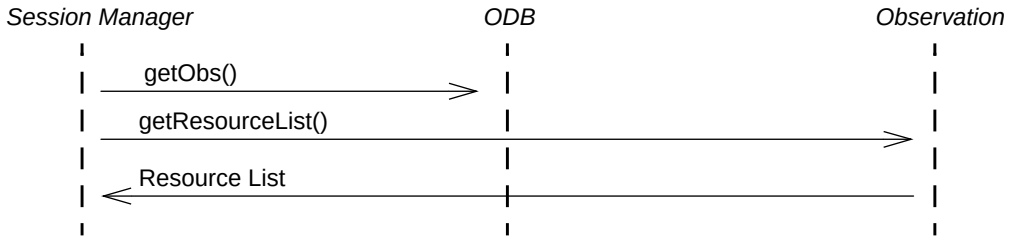



### *C5 • The Session Manager checks observation resources.*

*Background.* The Session Manager must be able to determine what resources an Observation uses. The SM does this by querring the next Observation in the Plan. It can then use the Resource List for the Observation.

*Scenario.* The Observation object is obtained from the ODB and examined for resources.

*FIGURE 10.* Checking for Resources

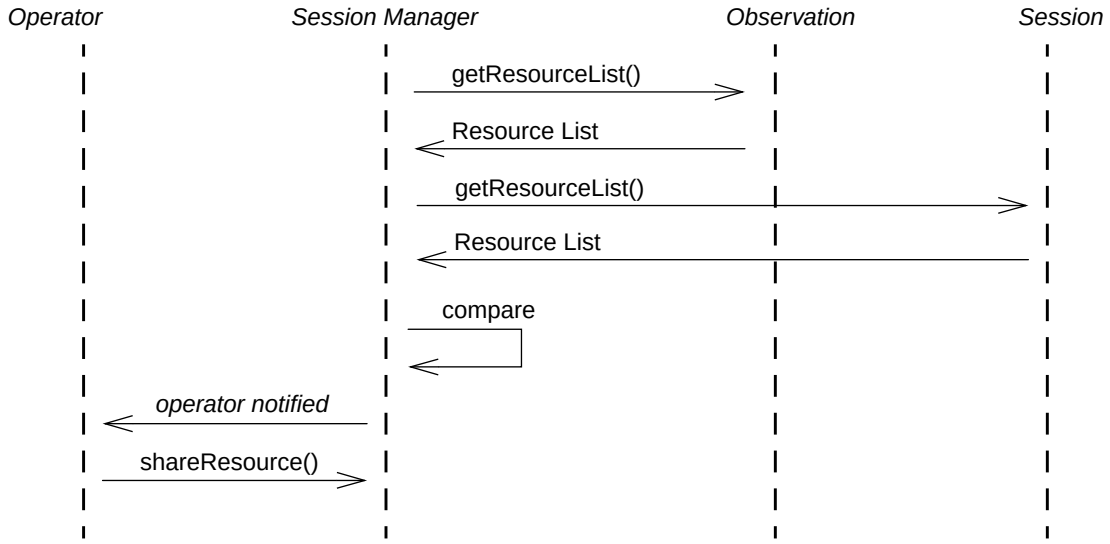



### *C6 • The Operator shares resources with an Observation.*

*Background.* An Observation must have at least one of its Resources before it makes sense for it to start executing. The Session Manager compares the list of Resources required by the Observation to the list already shared with the Session. If the Session needs additional Resources, the operator must intervene.

*Scenario.* The Observation resource list is compared with the Session resource list. The operator uses the Session Manager GUI to share his resources with the Session.

*FIGURE 11.* Sharing a Resource with a Session

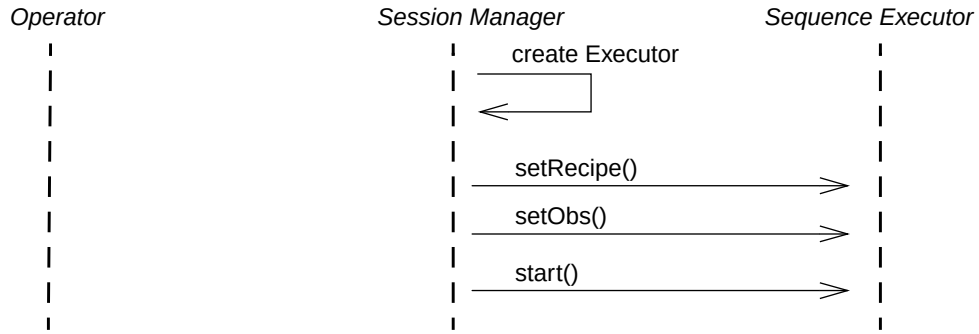



### *C7 • The Session Manager starts an observation.*

*Background.* The Session Manager gets the first observation from the Plan that is ready to execute (the observer marked them as ready). The Session Recipe is simple. The operator must select an Observation in the Session and run it (no automatic execution in this session).

*Scenario.* For this scenario the Operator has already allocated all the Resources to the Session the Observation needs as in the previous two cases. The Session Manager starts a new Executor, initializes it, and starts the Recipe. The operator can specify his own Recipe if he needs to or the Observation might have a Recipe attribute.

*FIGURE 12.* Starting a Session




The normal interactions of the Sequence Executor and the principal systems is covered in the IOI track documentation and need not be repeated here. During the execution of the recipe, the SE sends Sequence Commands to the principal systems.

However, there are several things the OCS must do during planned observing to support the instruments.

*C8 • The Sequence Executor snapshots status data.*

*Background.* All the status values that can be included in an Observation's header are present in the Status/Alarm Database. The design calls for the ability to add data to the headers (beyond the required default values) at run-time. A GUI will display all the status values associated with a component and allow the observer to indicate that a status value should be sampled just before OBSERVE or as soon as OBSERVE completes. The SE must snapshot the header data and save it with the observation in the ODB.

*Scenario.* The extraction of the snapshot information is done internally in the Sequence Executor. A new component is added to the Observation with a known name ("snapshot1").

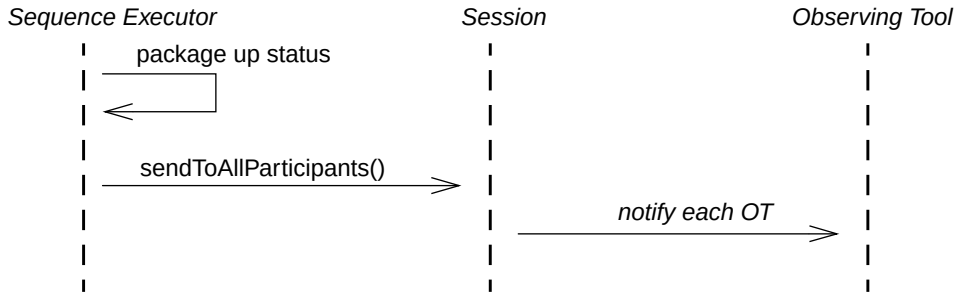
*FIGURE 13.* Checking for Resources


*C9 • The Sequence Executor posts status.*

*Background.* The Session Manager is charged with keeping the OT run-time status display up-to-date. Since the OT may be remote a packet of vital information is packaged up by the SE and forwarded to the Session Manager (rather than relying upon CLL monitoring) where it is forwarded to all the Session's participants.

*Scenario.* Every so often, as specified by the Recipe, the status is packaged up by the Sequence Executor using information in the Observation components. The information is forwarded to the SE's parent Session and forwarded to all the participants.

*FIGURE 14.* Posting Status to Participants




*C10• The Sequence Executor verifies a configuration.*

*Background.* The SE must notify the operator if any important part of the system's configuration is mistakenly altered during the time the Science Data is being acquired. The point of this case is to show that the SE will provide this capability.

*Scenario.* This case is taken care of by the SE Recipe using data from the Observation components. The SE monitors important CAR variables that are included in an Observation Component specified for this purpose. If one of these CAR variables is modified during the time the science data is being acquired the SE will post an alarm to notify the operator.

*C11• The Sequence Executor sends the DHS snapshot data.*

*Background.* A previous case showed how the SE snapshots SIR data during observation execution. The SDD calls for the OCS to send the snapshot data, the header data, to the DHS as an argument to the ENDOBSERVE command.

*Scenario.* Sending Sequence Commands is discussed in the IOI track documentation. The SE will send the snapshot data stored in the Observation object to the DHS at the appropriate time in the recipe. The DHS will handle the ENDOBSERVE command and write the data headers.

*C12• The Observer adds a Science Program Observation to the Session Plan/Science Program.*

*Background.* This case is to show how the Session Manager session view stays up to date when the blessed observer updates the session Plan or Science Program by dropping into the Plan an Observation from another observer's Science Program (for instance, an observation for queue observing).

*Scenario.* The modification to Plan causes the Observation in the Science Program to be linked to the session's Plan as shown in Figure 15. In the current design, when a connected, blessed OT updates the Plan, the OT sends a planUpdated() message to the Session Manager. This causes the Session Manager to check the plan. A list, in the observer's order, is returned to the Session Manager when the getNextObs() method is sent to the Plan. This is shown in Figure 16.

The chosen ODB product may provide an event notification capability. In this case, the database will keep the data consistent for all the clients using the data making the planUpdated() method unneeded.

*FIGURE 15.* Adding an Observation in a Science Program to a Plan

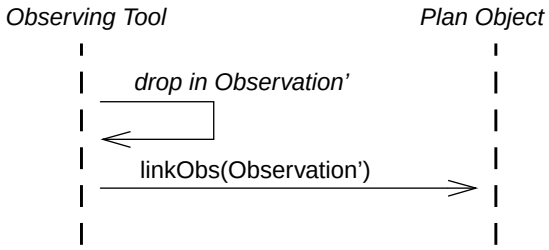
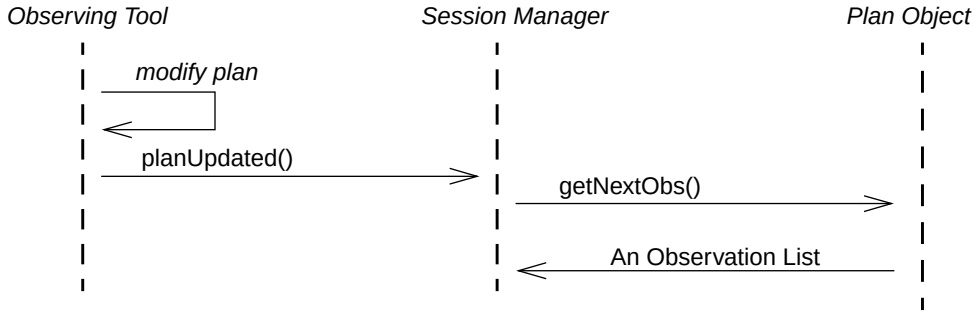



*FIGURE 16.* Getting the Next Observations after an update

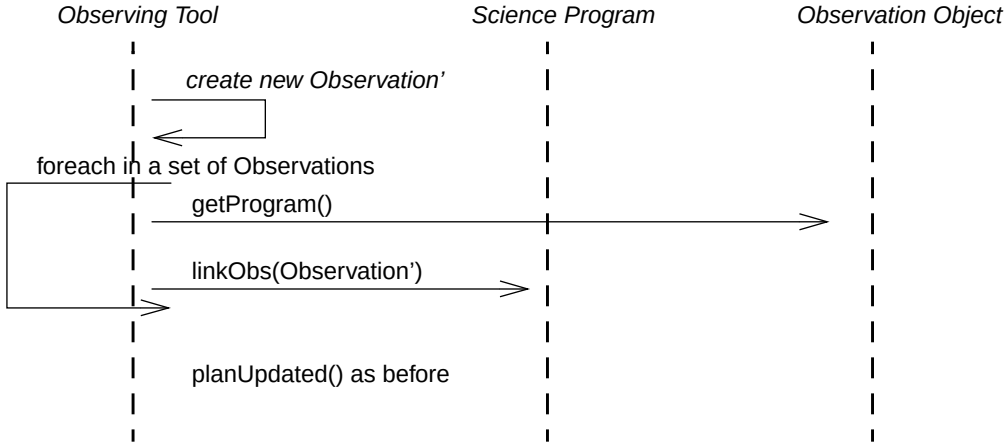



### *C13• The Observer creates an Observation into a Plan/Science Program.*

*Background.* Some Observations will also be created in Plans and shared with some of the Science Programs which have Observations in the Plan. The observer will select Observations in the Plan and *share* the new Observation.

*Scenario.* Plans and Science Programs can contain links to Observations in other Science Programs or the Observation objects themselves. (Actually, all may be implemented as *links* if all Observations are in the Observation Database in the ODB. The links/actual object distinction has more to do with ownership of an Observation than implementation.) In this case, links to the new Observation must be created in a number of associated Science Programs. Then, the Session view must then be updated as in previous cases.

*FIGURE 17.* Updating a number of Observations



*C14• The Operator previews an observation.*

*Background.* The OCS requirements [3] state that the operator must be responsible for all telescope motions. In some session recipes it will be necessary for the operator to examine the contents of the Observation before he allows it to execute. This case shows how this works in the POS track.

*Scenario.* An Observation in a Session is next and the Operator wants to look at it. He will select the representation of the observation in the Session Manager GUI and select a *view* command. This will cause status to be sent out from the Session Manager and any console configured to monitor the value of the "view observation" will see the change and respond by reading the Observation data and displaying it.

### 10.1 Dynamic Model Summary

These scenarios have shown many of the important operations of the POS. Some issues, such as concurrency in SEs or multiple SEs in a session, have been ignored but as far as we can tell they are just extensions of the given scenarios. Without a tool to develop pictures the work to do these scenarios is overwhelming. More scenarios will be produced as part of the detailed design of the track.

## 11.0 Interfaces

The POS track applications are OCSApp instances and rely upon the software interfaces provided by the IOI track CLL, primarily the OCS Message System interface.

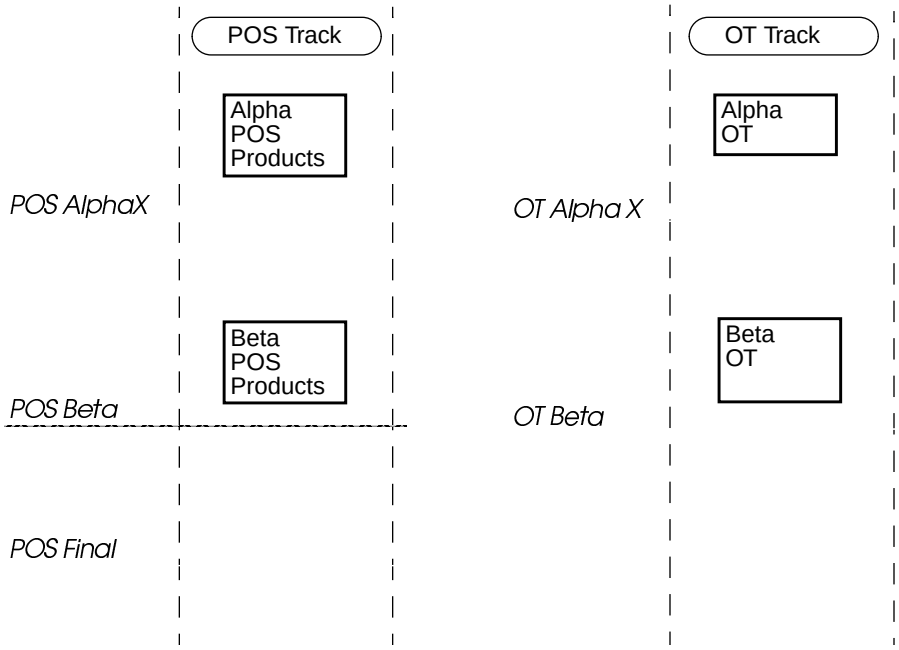
The format of the messages is not known at this time. The preliminary content of the messages is specified in the public interfaces for each kind of OCSApp in the Functional Model section of this paper.

## 12.0 Development Plan

The Planned Observing Support along with the Interactive Observing Infrastructure track provide essentially all the observing infrastructure for the GCS. However, the POS provides its infrastructure as applications rather than libraries and relies upon the availability of the IOI track products. The entire OCS development plan is discussed elsewhere [3].

The POS track is scheduled for a year of time in the WBS. The POS development plan will be phased to allow maximum OCS track parallelism. The POS products will be released in alpha and beta form before the entire POS track is completed so that work on the Observing Tool track can continue to progress in its own track. The POS releases will be timed with OT releases if possible. There may be multiple alpha and beta releases of POS and OT and then Final Release as specified in the OCS development plan.

*FIGURE 18.* POS Track Delivery




## 13.0 Usability Testing

The quality of a graphical user interface is generally better when the users of a product are involved in its development. However, the final users of the user interface products of the POS track (primarily the Session Manager), the telescope operators, will not be available for user involvement and testing. The usability testing approach for the POS track is the same as that used in the Telescope Control Console Track [9].

## 14.0 Documentation

The documentation for the POS track will follow the documentation requirements in the OCS SDR documents (SR66, SR67, SR68).

### 14.1 Observing Database Documentation

The Observing Database will be released with the following documentation for the components not related to EPICS status. The EPICS portions are covered in the IOI track documentation.

**The ODB Technical Document.** This document describes how the ODB data is structured, and the kinds of operations that are available.

**The ODB Programmer's Document.** This document describes how software components in the POS and the OT use the ODB. The software interface will be described here in case other future products wish to use the ODB.

**The ODB Testing Manual.** This manual will describe how to use the testing procedures required for the acceptance tests of the ODB.

### 14.2 Session Manager Documentation

The Session Manager will be accompanied by a technical design document and a user manual for the GUI.

**The Session Manager Operator Manual.** This manual will provide an overview of the control planned observing using the Session Manager GUI.

**The Session Manager Technical Document.** This document describes how the Session Manager software works and the software interfaces its components provide.

### 14.3 Sequence Executor Documentation

The Sequence Executor code will be accompanied by a technical document.

**The Sequence Executor Technical Document.** This document describes how the Sequence Executor software works and the software interfaces its components provide. This document will be used by the creators of stand-alone consoles to add the ability to execute observations.

## 15.0 Deliverables

The following items are the deliverables of the final release of the POS track.

- Session Manager server and GUI
- Sequence Executor and associated code
- Observing Database
- Documentation described previously and in the development plan

# 16.0 A Somewhat More Detailed Design of the POS Track

Section 8.0 described a high-level view of the POS by presenting the public methods for each of the track products. This section builds the physical model for each of the track products showing the high-level components of each product and the interactions between the components. Additional private methods are presented along with additional details on the structuring of the POS data.

## 16.1 Session Manager High-Level Design

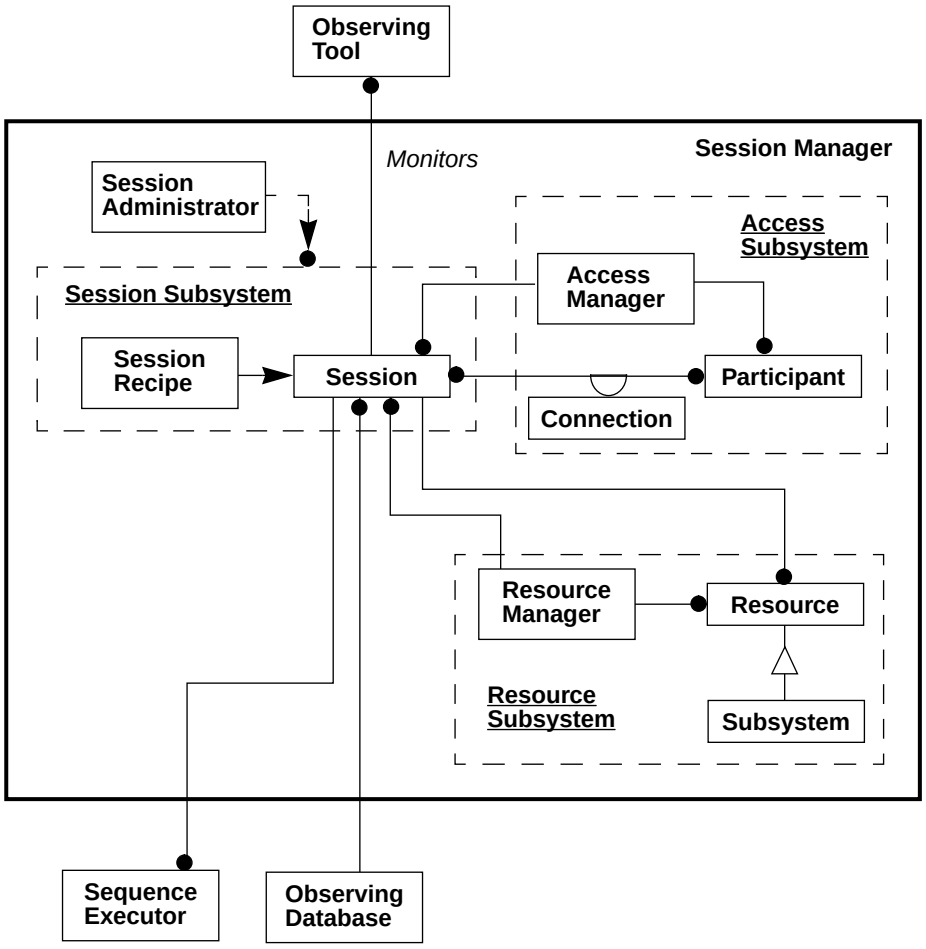
The Session Manager depicted in Figure 2 on page 5 is decomposed and explored in further detail in this section. The Session Manager application has evolved around the natural concept of the Observing Session. It is responsible for creating and controlling sessions, managing resource allocation, and directing run-time information to observers. The SM model is comprised of several subsystems and a Session Administrator as indicated in Figure 19. The Observing Tool, Observing Database, and Sequence Executor are also shown to demonstrate how the SM fits in with other OCSApps. The Session Administrator creates and destroys Session Subsystems, and each subsystem fulfils a distinct task as indicated below.

**Session Subsystem.** This subsystem is the heart of the Session Manager. A new Session Subsystem is created for each session, and it relates the remaining subsystems.

**Access Subsystem.** Access to the Session Manager is handled here.

**Resource Subsystem.** Resources are allocated to sessions using the functionality of this subsystem.

*FIGURE 19.* Session Manager




The Session Administrator and the subsystems are detailed in the remaining sections.

### 16.1.1 The Session Administrator Object

The Session Administrator presents an interface to the SM GUI that is used by the system operator to create and destroy the objects in the Session Subsystem (see Figure 20). A few simple methods are indicated in the figure and discussed in Table 8. Any number of sessions may exist concurrently, although it is expected that the most common case will be a single session. When a session is created, Session Recipe and Session objects are instantiated as discussed in the next section.

*FIGURE 20.* The Session Administrator Object

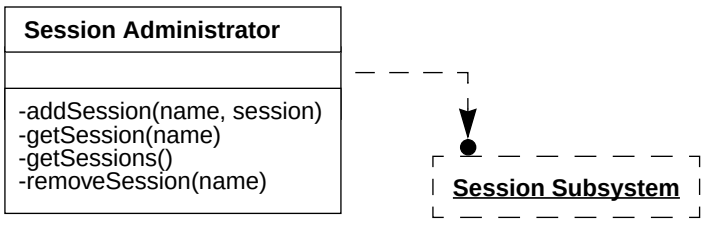


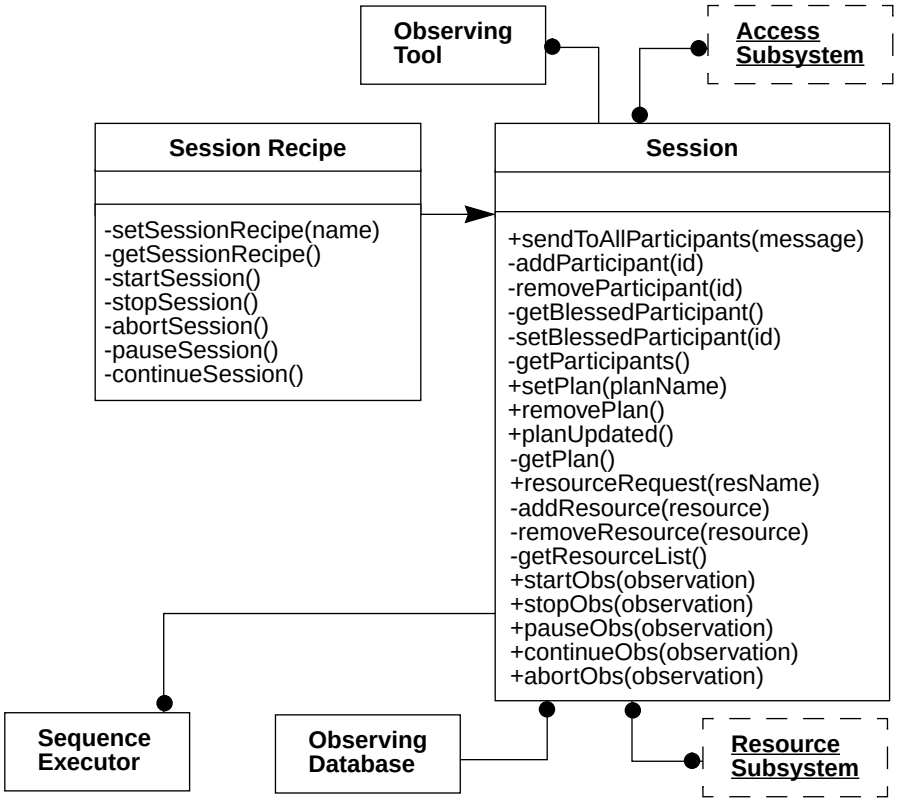
*TABLE 8.* Session Administrator Methods

| Method | Use |
|---|---|
| addSession(name, session) | This method adds a new session to the pool of sessions being administered by the Session Administrator. |
| getSession(name): Session | Returns the session with the given name. |
| getSessions(): SessionList | Returns a list of all the sessions. |
| removeSession(name) | This method removes the named session from pool of sessions being administered by the Session Administrator. |

### 16.1.2 Session Subsystem

The Session Administrator creates a Session Recipe and Session object for each session as shown in Figure 21. The Session Recipe object controls the execution of the session itself. Sessions can be handled by the SM in several ways. For instance, they can be executed in "automatic" mode wherein each observation in the plan is started as soon as the resources become available, or the system operator may desire to explicitly check each observation before it is executed. The Session Recipe is set by the operator to control the session as a whole and contains methods to start/stop/pause/continue the session.

*FIGURE 21.* Session Subsystem




The Session is the central object of the entire SM application. It links together the functionality of all the subsystems and coordinates the session's interactions with the Observing Tool, Sequence Executor, and Observing Database. In particular it handles the following tasks for the session:

- **Interactions with participants**.

  — The Sequence Executor and the SM itself generate status that must be reflected to the participants. The Session object relays this information to the proper Observing Tools.

  — The system operator uses a method in the Session object to set the “blessed” participant and grant permission to his Observing Tool to modify the plan.

  — The blessed observer contacts the Session object to set the plan that the SM executes.

- **Resource allocation for the session.** Before a sequence executor can execute an observation, it must first obtain the resources it needs from the Session object. If the resources have been allocated to the session by the system operator, and if they are not in use, then the Session grants the resources. If a resource has not been allocated, the system operator is contacted to request it.
- **Control of the execution of objects in the session.** The Session contains methods used by the Session Recipe and other applications to start/stop/pause/continue/abort its observations.

The Session object also provides methods used by the Access Subsystem to add/remove participants, and by the Resource Subsystem to add/remove resources. All the methods are detailed below.

*TABLE 9.* Session Methods

| Method | Use |
|---|---|
| sendToAllParticipants(message) | Sends a message, such as a status update, to all participants in the session. |
| addParticipant(participant) | Add a monitoring participant to the session. This method is invoked by the system operator in response to a connection request from an OT. |
| removeParticipant(participant) | Remove a participant from the session. |
| getBlessedParticipant(): Participant | Determine who the blessed participant is. |
| setBlessedParticipant(participant) | Set the blessed participant. If there is already a blessed participant, he becomes a monitoring participant. |
| getParticipants(): IdList | Obtain a list of all the participants in the session. |
| setPlan(name) | Associate the named plan with the session. If there is already a plan associated with the session, then it is forgotten in favor of the new plan. |
| removePlan() | No longer associate a plan with the session. |
| planUpdated() | Informs the Session that its plan has been updated. The Session should respond by retrieving the updated plan from the ODB. |
| getPlan(): name | Determine which plan is associated with the session. |
| resourceRequest(name) | The Sequence Executor uses this method to request a resource. If the session does not already have access to the resource, the system operator is notified. |
| addResource(resource) | This method is invoked to notify the session that it has access to the given resource. |
| removeResource(resource) | Notify the session that it no longer has access to the given resource. |
| getResourceList(): ResourceList | Returns the list of resources available to the session. |
| startObs(observation) | Starts executing an observation in the plan, subject to the approval of the system operator. This method calls the startObs() method in the SE for the observation. |

*TABLE 9.* Session Methods

| Method | Use |
|---|---|
| stopObs(observation) | Stops an executing observation associated with the session. This is a “graceful” stop, resulting from invoking the SE’s stopObs() method. |
| pauseObs(observation) | Pauses an observation, resulting in pausing the subsystems that it uses. This method invokes the SE’s pauseObs() method. |
| continueObs(observation) | Continues a paused observation. This method invokes the SE’s continueObs() method. |
| abortObs(observation) | Halts an executing observation immediately. This method invokes the SE’s abortObs() method. |

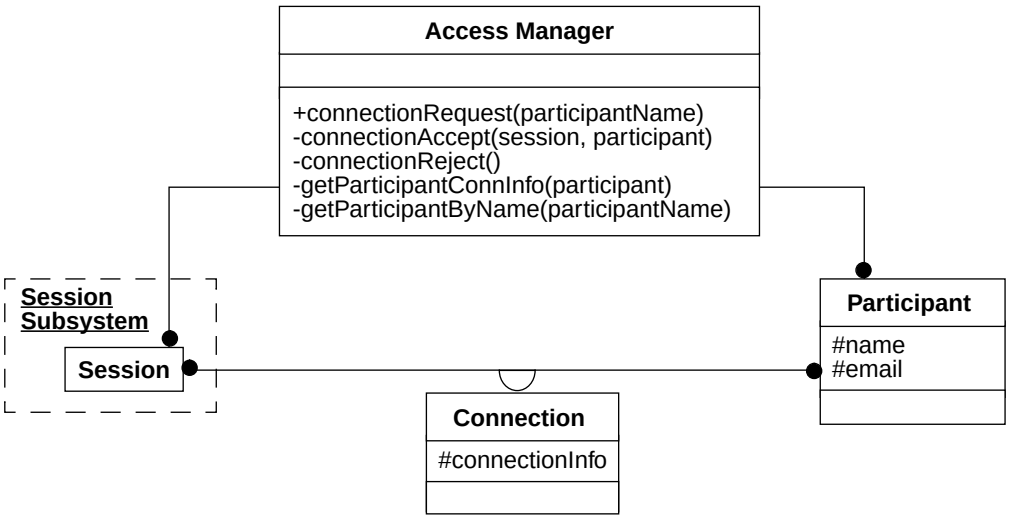
*TABLE 10.* Session Recipe Methods

| Method | Use |
|---|---|
| setSessionRecipe(name) | This method is invoked to change the session recipe. |
| getSessionRecipe(): name | Returns the current session recipe name. |
| startSession() | Starts to execute the session recipe. |
| stopSession() | Stops the session gracefully. |
| abortSession() | Stops the session immediately. |
| pauseSession() | Pause the session recipe execution. |
| continueSession() | Continue a paused session. |

### 16.1.3 Access Manager Subsystem

The Access Subsystem, shown in Figure 22 below, is used to maintain associations between participants and sessions. The Access Manager is the coordinating object of this subsystem. When a participant contacts the SM application with his Observing Tool, a method in the Access Manager is called to notify the system operator. Other methods are used to accept or reject the connection request. If the connection is accepted, a Participant object is created (if one does not exist already) to represent the participant in the system. The Participant object stores information about the observer such as his name and email address. The Session object of the session that the observer is interested in is also informed of the new participant. Since a participant may be interested in more than one session, and a session may have more than one participant, information about particular session/participant associations is stored in the Connection link attribute.

*FIGURE 22.* Access Subsystem Details




The methods that the Access Manager responds to are detailed in Table 8.

*TABLE 11.* Access Manager Methods

| Method | Use |
|---|---|
| connectionRequest(name) | The Observing Tool contacts the Access Manager using this method when an observer wishes to participate in a session. The Session Manager operator is notified and responds, resulting in one of the next two methods. |
| connectionAccept(session, participant) | If the system operator at the Session Manager console accepts a connection, this method is invoked. Session and Participant objects are first created if necessary before this method is invoked, and then the Access Manager creates the association between the two. |
| connectionReject() | When a connection request is rejected by the system operator, this method is used to inform the observer. |
| getParticipantConnInfo(participant): ConnectionInfo | This method returns all the connection information for the given participant. |
| getParticipantByName(name): Participant | This method returns the participant identifier for the given observer's name. |

### 16.1.4 Resource Subsystem

The Resource Subsystem is used to allocate resources to sessions (see Figure 23). The SM GUI provides a means by which the system operator can indicate which resources should be granted to each session. The Resource Manager object presents an interface to implement this functionality.

A Resource object exists for each system resource. It contains any necessary resource information along with methods to permit the resource to be shared among sessions and staff.

*FIGURE 23.* Resource Subsystem Details

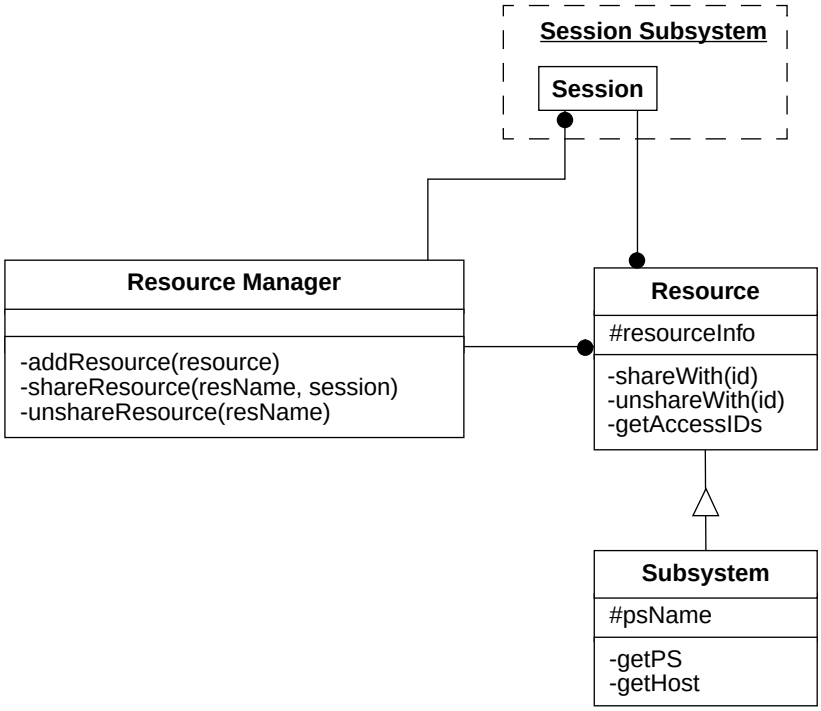



*TABLE 12.* Resource Manager Methods

| Method | Use |
|---|---|
| addResource(resource) | Adds the given resource to the pool of resources being administered by the Resource Manager. |
| shareResource(name, session) | This method is invoked when the system operator wishes to grant a resource to a particular session. |
| unshareResource(name) | This method is invoked to remove access to the given resource. |

*TABLE 13.* Resource Methods

| Method | Use |
|---|---|
| shareWith(Id) | Grants the given ID permission to use the resource. |
| unshareWith(Id) | Removes permission to use the resource. |
| getAccessIDs(): IdList | This method is invoked to determine who has access to the resource. |

## 16.2 Observing Database High-Level Object Model Design

The Observing Database is the persistent store for all the data the OCS uses during an observing session. At the highest level it is modelled as a set of collections of document types. The documents managed by the collections are themselves containers for more primitive data components. The documents share a common, open construction that allows them to easily change their content. The data structure is discussed in the next section.

The ODB is modeled as an object-oriented database, not as a set of tables and keys as would be found in a relational database. An object-oriented database provides the query capabilities of a relational database, but also provides access methods that are familiar to programmers. The ODB appears as a persistent store for objects. This means that the data in the ODB can be complex and flexible and when changes are made, they are permanent. The relationships between the components and the contents of the components can change and need not all be identical. This is required in order for the OCS to support the ability to add arbitrary components to Science Programs and the ability to change the content of the data headers at run-time.

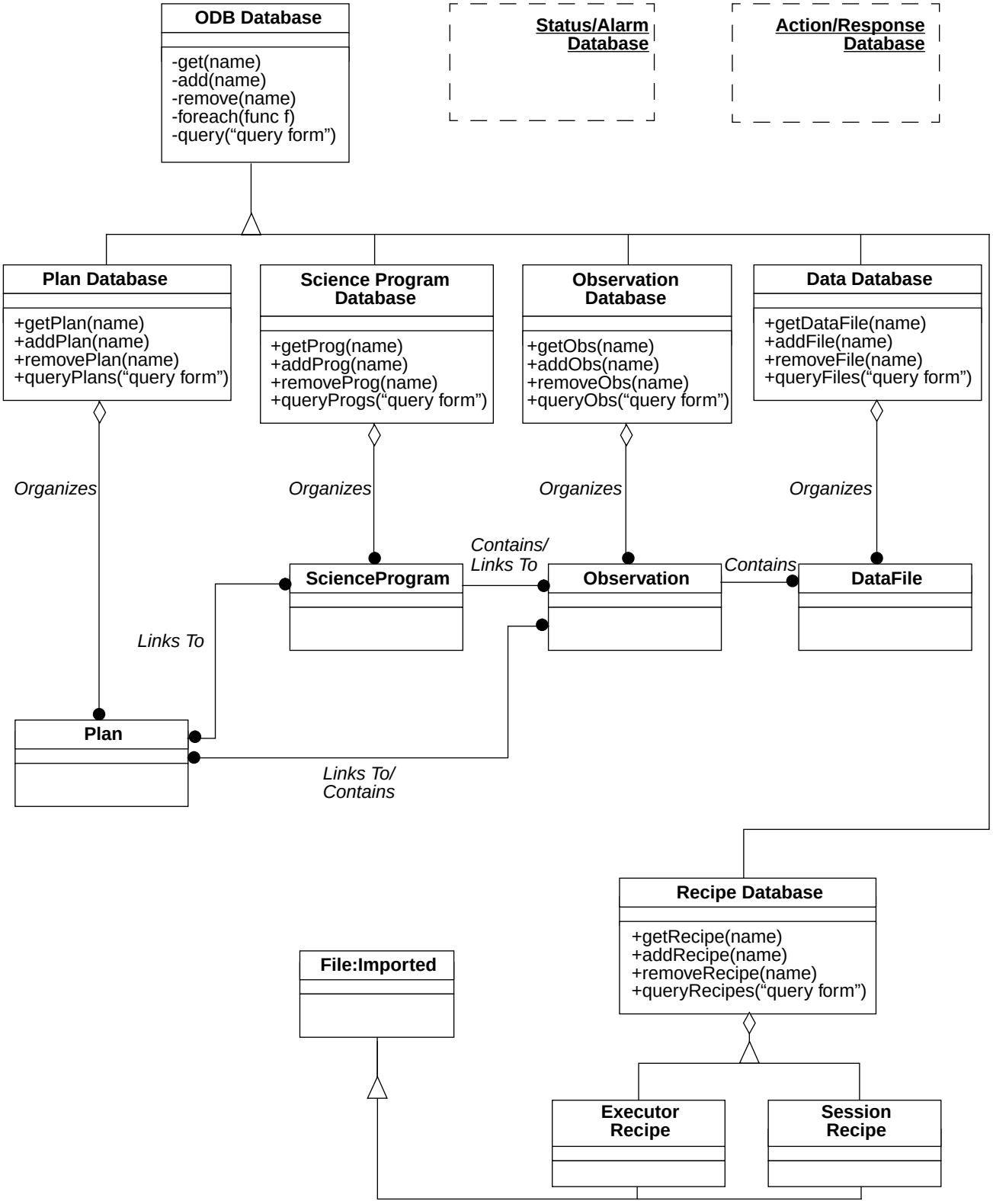
Figure 24 shows the Observing Database object model. There is a logical databases for each of the primary persistent types: Plans, Science Programs, Observations, and Recipes. Each database is derived from a base ODB Database type that provides basic database operations. Each database specialization adds convenience methods that are built upon the base class methods and operate upon the correct data types.

Below the databases themselves are shown the associations between the documents and data.

- A Plan can reference many Science Programs and a Science Program can be referenced by many Plans.
- A Plan can also contain Observation objects. For instance, an on-site observer will include calibration observations which don't appear in any one user's Science Program. The ability for an Observation to appear in many Plans has been included event though that may be undesirable rather than enforcing a constraint at this time.
- Associated with each Observation object is a number of DataFile objects. These DataFiles contain the OCS data to tie the observations in a Science Program to the actual data in the DHS. The actual data is modelled as references to the external database.
- The Recipe Database is used by the POS applications. It contains the Session Recipes and the Sequence Executor Recipes. These recipes are derived from the imported File type meaning that these files will be text-based scripts that simply stored in the database.

The detailed model of the data in the databases was shown in Figure 4 of Section 9.0 and is not repeated here. The basic structure of the documents in the databases is described in the next section.

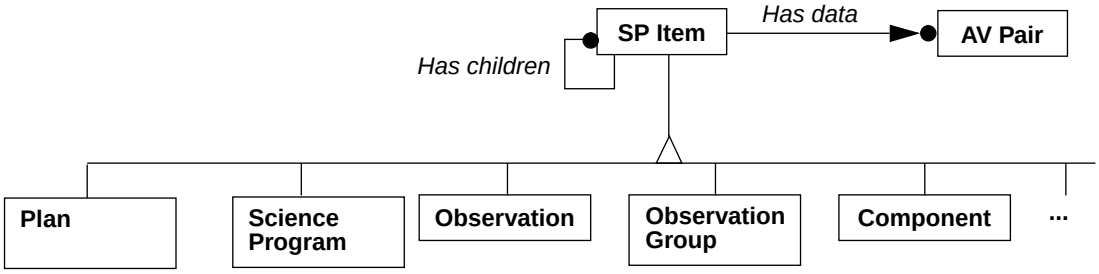
*FIGURE 24.* Observing Database Object Model



## 16.3 Database Data Design and Functional Model

The Plans, Science Programs, Observations, Observation Groups, and Components are all made up of (are derived from) a simple building block shown in Figure 25 called the SPItem (Science Program Item). Each SP Item contains zero or more attribute/value pairs and zero or more SP Item children. The SP Item is a container class with some data.

*FIGURE 25.* The Database Member Object Model




The SP Item Class is an abstract class that can be instantiated to any of the concrete program elements such as observations and instrument configurations. An SP Items consists of the attributes and methods depicted in Figure 26. The exact specification of methods is left to the detailed design, but a preliminary method set is indicated Figure 26 and discussed in Table 14 to illustrate the ideas that make SP Item flexible and open.

*FIGURE 26.* The SP Item Abstract Class

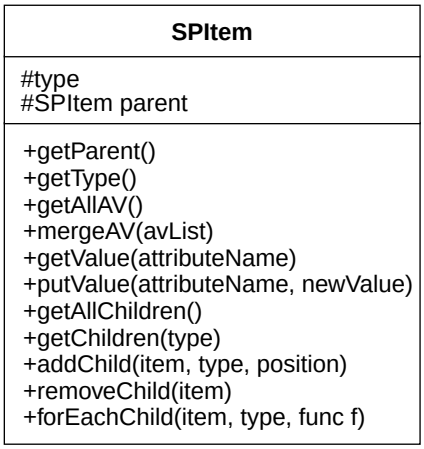



*TABLE 14.* Methods of Abstract Class SPItem

| Method | Use |
|---|---|
| getParent():SPItem | Return the “parent” SP Item into which the item has been nested. This is important for many operations, including moving an item within the hierarchy. |
| getType():type | Return the “type” of the SP Item. |
| getAllAV | Return a list of all the attribute/value pairs associated with the item. This is useful when the item is copied. |

| Method | Use |
|---|---|
| mergeAV(avList) | Merge the given attributes and values with the existing attribute set, replacing the values of existing attributes. Drag and drop operations will use this method. |
| getValue(attributeName) | Return the value associated with a particular attribute. This method is useful for initializing a form containing widgets that are used to manipulate the attribute. |
| putValue(attributeName, newValue) | Update a specific attribute with a given value. This is used when editing a form for the SP Item. |
| getAllChildren():SPItemList | Return all the children SP Items associated with item. The list of children is needed for many operations, including expanding and collapsing the hierarchy. |
| getChildren(type):SPItemList | Return all the children SP Items associated with item of type *type.* |
| addChild(item, type, position) | Add a child SP Item at the indicated position. This method is used to establish the hierarchy of SP Items. |
| removeChild(item) | Remove the given item from the set of children associations. Move and delete operations will make use of this. |
| forEachChild(item, type, func f) | Apply function f to all the children with correct *type* belonging to *item*. |

The table shows that the basic SP Item interface supports most of the operations required to compose and manipulate Science Programs and Plans as prototyped in the OT track Observing Tool (See also [7].) To get to a component (a leaf in the structure), a program recursively examines an SP Item's children until the component is either found or has no children. Attributes and values can be present at any level. Each component has a *type* (observation, plan, science program, etc.) to speed up frequent operations.

## 16.4 ODB Database Design and Functional Model

Table 2 of Section 9.2 showed the high-level POS view where the ODB appeared as a single OCSApp instance with a set of public methods. Figure 24 shows a more detailed object model of the ODB where the application is broken down into a number of individual database components. This section shows the public methods of the ODB separated by document type.

Table 15 shows the methods for the abstract ODB base class. The methods of this table provide the basic functionality of the database classes.

*TABLE 15.* Methods for Abstract ODB Class

| Method | Use |
|---|---|
| get(name): ODBObject | This private method returns a named object in an ODB database as a ODBObject. |
| add(ODBObject) | This private method adds an ODBObject to a database. |
| remove(name): ODBObject | The named ODBObject is removed from the ODB database and returned to the caller. |
| foreach(func f) | This private method can is used to apply the function f to all the items once in an ODB database. |
| query("Query form"): ODBObject List | This private method is used to make a query on objects in an ODB database. The form of the query is probably some text string. The result is a list of ODBObjects. |

Each of the individual databases extend the base class and personalize the methods for their own types to provide convenience and the opportunity to hide class-specific implementation details.

*TABLE 16.* Methods for Plan Database Class

| Method | Use |
|---|---|
| getPlan(name): Plan | A reference to a Plan object associated with *name* is returned. |
| addPlan(name) | This method adds Plan *name* to the Plan Database. |
| removePlan(name):Plan | This method removes the Plan *name* from the Plan Database and returns a reference to it. |
| queryPlans(“Query form”):PlanList | This method is used to select a subset of Plans from the Plan Database. The query results in a list of Plans. |

*TABLE 17.* Methods for ScienceProgram Database Class

| Method | Use |
|---|---|
| getProg(name): ScienceProgram | A reference to a ScienceProgram object associated with *name* is returned. |
| addProg(name) | This method adds ScienceProgram *name* to the Science Program Database. |
| removeProg(name):ScienceProgram | This method removes the ScienceProgram *name* from the Science Program Database and returns a reference to it. |
| queryProgs(“Query form”): ScienceProgramList | This method is used to select a subset of SciencePrograms from the Science Program Database. The query results in a list of SciencePrograms. |

*TABLE 18.* Methods for Observation Database Class

| Method | Use |
|---|---|
| getObs(name): Observation | A reference to an Observation object associated with *name* is returned. |
| addObs(name) | This method adds Observation *name* to the Observation Database. |
| removeObs(name):Observation | This method removes the Observation *name* from the Observation Database and returns a reference to it. |
| queryObss(“Query form”):ObservationList | This method is used to select a subset of Observations from the Observation Database. The query results in a list of Observations. |

*TABLE 19.* Methods for Data Database Class

| Method | Use |
|---|---|
| getDataFile(name): DataFile | A reference to a DataFile object associated with *name* is returned. |
| addFile(name) | This method adds DataFile *name* to the Data Database. |
| removeFile(name):DataFile | This method removes the DataFile *name* from the Data Database and returns a reference to it. |
| queryData(“Query form”):DataFileList | This method is used to select a subset of DataFiles from the Data Database. The query results in a list of DataFiles. |

*TABLE 20.* Methods for Recipe Database Class

| Method | Use |
|---|---|
| getRecipe(name): Recipe | A reference to a Recipe object associated with *name* is returned. |
| addRecipe(name) | This method adds Recipe *name* to the Recipe Database. |
| removeRecipe(name):Recipe | This method removes the Recipe *name* from the Recipe Database and returns a reference to it. |
| queryRecipes(“Query form”):RecipeList | This method is used to select a subset of Recipes from the Recipe Database. The query results in a list of Recipes. |

## 16.5 Sequence Executor Design and Functional Model

Table 2 of Section 9.2 showed the high-level POS view where the Sequence Executor appeared as a single Class derived from OCSApp with a set of public methods. Figure 27 shows a more detailed object model of the Sequence Executor where the application is broken down into a number of cooperating classes. This section describes the object model of the SE more fully. The public and private methods of the SE, shown here as parts of other classes, are presented for each class.

The SE is an engine for executing a single observation, which can produce zero or more science data files. The object model of the SE is very similar to what was functionally described in the SDD. In Figure 27 the Sequence Executor composite object is shown as an association of four object classes: the Sequence Executor, the Executor Recipe, the Executor Controller, and the Observation. The Observation is not a part of the SE; it is the instance of class Observation that is being executed by the SE. It is included here with its methods for clarity.

**Sequence Executor.** The job of the Sequence Executor object is to coordinate the efforts of the other objects. This object supports methods that allow the caller to identify which Recipe to use and which Observation is to be executed. Setting the Recipe associates a particular Recipe with the Executor Recipe. Setting the observation associates an Observation in the ODB with the Sequence Executor.

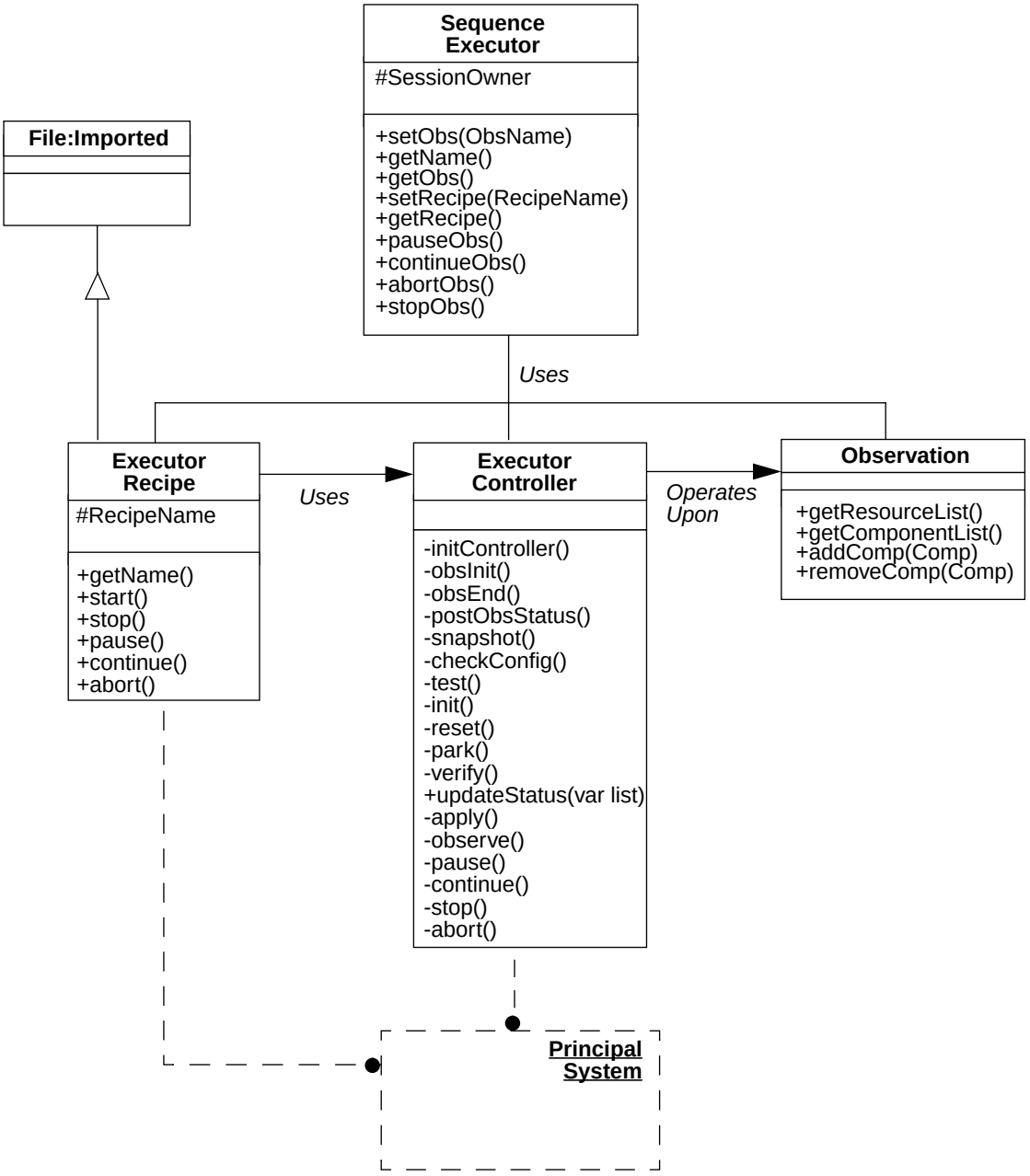
**Executor Recipe.** The Recipe is a text file that describes the consistent operations that take place every time an observation is made. Recipes are created by the operations staff, not observers. The recipes are to be scripts (derived from File) editable with any text editor. The Recipe class supports a few methods that are used by a Sequence Executor object to control the gross operation of the observation.

**Executor Controller.** The Executor Controller is the code-level support for the recipe (the Recipe Library in the SDD). An Executor Recipe uses the calls in the pubic interface of the Executor Controller to execute the Observation pattern and to control the principal systems. The Executor Controller sends Sequence Commands (see [10] for definitions) to the principal systems as has been discussed too much elsewhere.

**Observation.** The Observation object is accessed in the ODB by the Sequence Executor and used by the Executor Controller to provide information it needs to successfully execute the observation. The Executor Controller can modify an observation when it saves header information or updates component configurations at the conclusion of a Verify. It relies upon the Observation to tell it what it needs to execute (what resources it needs.)

The Sequence Executor is a cooperative effort of all four object classes and an instance of the Sequence Executor will have one instance of each class. Dynamic modeling of the SE is reserved for the DD phase.

*FIGURE 27.* Sequence Executor Object Model




## 16.6 Sequence Executor Functional Model

The following tables describe the methods each SE class exports. Note that all but one of the Executor Controller methods are private meaning they are only available for use by other objects in the Sequence Executor.

*TABLE 21.* Sequence Executor Class Methods

| Method | Use |
|---|---|
| setObs(obsName) | This method allows the caller to set the Observation that will be executed by the SE. The name of the observation is a character string. |
| getName(): Name | This method returns the name of the observation associated with the SE. |
| getObs(): Observation | This method returns the Observation object associated with the Sequence Executor instance. |
| setRecipe(recipeName): | This method allows the caller to set the Recipe that will be used to execute the observation. The name of the Recipe is a character string. Calling this method causes the SE object to search for the recipe in the ODB. |
| getRecipe():recipeName | This method returns the name of the recipe associated with the SE. |

The methods in Table 22 are used by the Sequence Executor object to control the high-level operation the observation.

*TABLE 22.* Executor Recipe Methods

| Method | Use |
|---|---|
| getName(): Name | This public method returns the name of the recipe. |
| start() | This public method starts the execution of the observation. |
| stop() | This public method stops the execution of the observation at the next appropriate time. |
| pause() | This public method pauses the execution of the observation if it is executing. |
| continue() | This public method continues the execution of the observation if it is paused. |
| abort() | This public method causes an observation to terminate immediately. |

The methods of Table 23 are the primitives that can be used to build the observing pattern in a recipe. There are several methods that do not map to Sequence Commands, but are used to allow the controller to manage the operation of the controller. Most Sequence Commands are present in the controller interface. Some like ObsEnd are not present in the Executor Controller interface because the controller's observe() only returns once the complete OBSERVE sequence is completed.

*TABLE 23.* Executor Controller (all methods are private unless otherwise noted)

| Method | Use |
|---|---|
| initController() | This method is provided to allow the Executor Controller (EC) object to perform any self initialization required at the beginning of observation execution. |
| obsInit() | This method is called to have the EC to perform operations related to the observation (such as resource allocation) that are required before actual execution begins. |

| Method | Use |
|---|---|
| obsEnd() | This method is called to have the EC do final clean up operations once the observation execution is completed (such as freeing resources). |
| postObsStatus() | This method is used to cause the EC to post status information related to the execution of the observation. |
| snapshot() | This method.causes the EC to sample the values of status values for headers. It then stores the data set with the observation in the ODB. |
| checkConfig() | This method is used to force the EC to immediately check to see if any important status values have changed during observation execution. Important status values are associated with the observation components. |
| test() | This method causes the EC to perform the TEST sequence command. |
| init() | This method causes the EC to perform the INIT sequence command. |
| reset() | This method causes the EC to perform the RESET sequence command. |
| park() | This method causes the EC to perform the PARK sequence command. |
| verify() | This method causes the EC to perform the VERIFY sequence command. The EC will return when the VERIFY is complete with a list of changed variable names. |
| updateStatus(var name list) | This public method is used to cause the EC to update variable names in the Observation. |
| apply() | This method causes the EC to perform the APPLY sequence command. |
| observe() | This method causes the EC to perform the OBSERVE sequence command. The method completes when the observation is completed. This may involve execution of a complex observe script depending on the Observation. |
| pause() | This method causes the EC to perform the PAUSE sequence command. |
| continue() | This method causes the EC to perform the CONTINUE sequence command. |
| stop() | This method causes the EC to perform the STOP sequence command. |
| abort() | This method causes the EC to perform the ABORT sequence command. |

## 16.7 Acquisition of Data from Stand-Alone Consoles

As an alternative, it has been stated that instrument consoles should be able to acquire data on their own, without the support or assistance of the POS track. This section presents the issue and discusses how it is supported in the OCS. This feature of OCS instrument consoles should not be confused with the functionality provided by the instruments themselves through their engineering consoles, which must also have the ability to acquire data.

### 16.7.1 What can be done and what can't be done

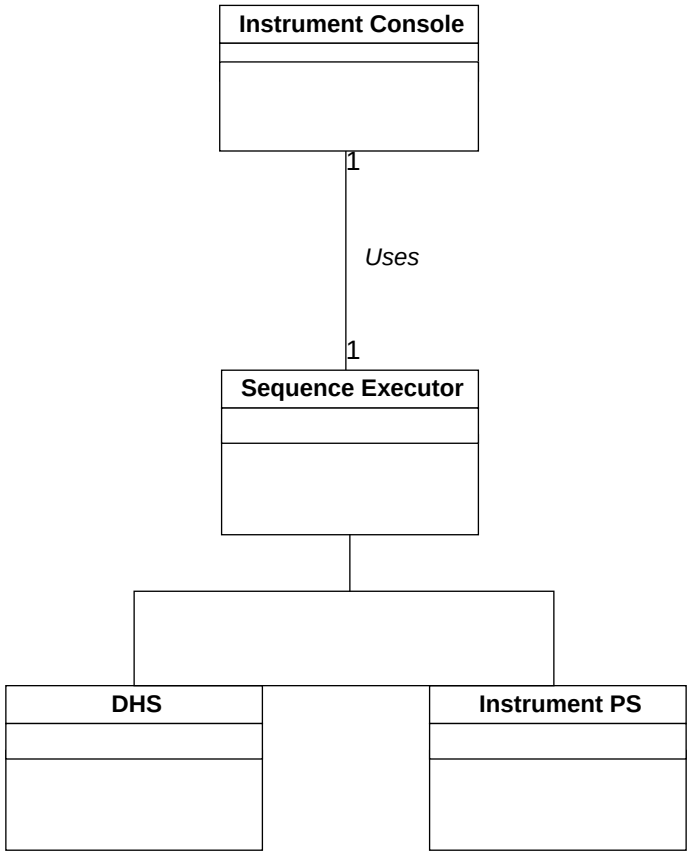
The features and advantages of the POS are not available when using an instrument console to acquire data. When observers are acquiring data with an instrument's console they are doing interactive classical observing.

- Observer's can configure the instrument and obtain data with their console.
- Configuration and verification of the configuration of the telescope and any other subsystems is done by the observers and operators.
- Any sequencing of the principal systems is done by the operator and observer.
- The data acquired with a console is not associated with any Science Program so the OT can not be used. There is no record of the data or observations in the ODB. The data record will be in the DHS and the data headers.

### 16.7.2 Solution Design

Our solution is to use the products of the POS track, namely the Sequence Executor, to acquire data from consoles. The SE design has been done to support this feature. Figure 28 shows the Object Model.

*FIGURE 28.* Stand-alone console Object Model




The instrument console and Sequence Executor are both instances of OCSApp so they have the capability of communicating with principal systems. A single SE is associated with an instrument console used in this mode. The SE runs a very simple static recipe that takes the instrument configuration from the console and applies it to the instrument. The recipe is a script that is used over and over each time a new data is acquired

with the console. The data flows to the DHS from the instrument as it always does. The following steps are taken when the *observe* button is pushed.

- The console builds a configuration and makes it available to the SE.
- The SE takes the configuration and applies it to the instrument (instrument consoles don't configure other principal systems).
- The SE sends OBSERVE and ENDOBSERVE at appropriate times.
- The SE can send values from its console to the DHS with ENDOBSERVE. Other header data should be sampled by the instrument itself in this mode.
- During OBSERVE, the console can be used to *pause* or *continue* the data acquisition process.

Since the OCS provides the ability to write scripts that can control the principal systems and access data in the status/alarm database, there are many approaches to console-based interactive observing. We have chosen the approach presented here because it allows us to reuse code from the POS.

Created:July 6, 1995
Modified:March 18, 1996

Gemini Observatory Control System Report

# Observing Tool Track Preliminary Design

**Shane Walker, Kim Gillies, Steve Wampler**

ocs._sw.004-OTTrackPD/01

**This report presents the preliminary design of the Observing Tool Track, a part of the Gemini Observatory Control System.**

## 1.0 Introduction

The Observing Tool Track is one of the development tracks in the development plan for the Observatory Control System (OCS) of the Gemini Telescopes. The following statements are from the Software Design Review OCS Development Plan [6].

> This track is dedicated to developing the Observing Tool and its associated functionality including run-time monitoring of observations. (page 3)
>
> [This track, along with the Planned Observing Support Track] must be completed before automatic control of the telescope configuration is possible. This functionality is required for the operational phase planned observing capabilities. (page 7)

The product of the Observing Tool track is the Observing Tool (OT) application, the primary astronomer interface of the Gemini control system. A prototype has been created that simulates the planning and remote data collection functions of the OT. This prototype, along with the discussion of the OT in the SDD, represent baseline functionality of the OT interface [1]. The purpose of this document is to describe how the OT fits in with the remainder of the GCS software.

This report presents the preliminary design of the OT track to a depth such that the track can continue on in its development independently of the other OCS tracks. The following information is contained in this report:

- Summary view of the OT prototype.
- Remaining decisions for the detailed design of the track.
- High-level preliminary design for the track.
- Dataflow between applications in the OT track and the other OCS tracks.
- Usability testing plans.
- List of required documentation.

- OMT Modelling information including a description of the Subject, View, and Controller of the OT OCS-App.

## 2.0 Acronyms

| | |
|---|---|
| API | Application Programmer Interface |
| CLL | Command Layer Library |
| CTT | Control Track Library |
| GUI | Graphical User Interface |
| GCS | Gemini Control System |
| IOC | Input/Output Controller |
| IOI | Interactive Observing Infrastructure |
| OCS | Observatory Control System |
| ODB | Observing Database |
| OT | Observing Tool |
| PD | Preliminary Design |
| POS | Planned Observing Support |
| PS | Principal System |
| PSA | Principal System Agent |
| SAD | Status Alarm Database |
| SDD | Software Design Description |
| SIR | Status Information Record |
| SM | Session Manager |

## 3.0 References


[1] SPE-C-G0037, *Software Design Description*, 9/9/94, Gemini Controls Group, 1994.

[2] ocs.kkg.032, *Interactive Observing Infrastructure Preliminary Design*, Kim Gillies, Shane Walker, Steve Wampler, Gemini Observatory Control System Group, 1995.

[3] ocs.ocs.004, *Planned Observing Support Track Preliminary Design*, Kim Gillies, Shane Walker, Steve Wampler, Gemini Observatory Control System Group, 1995.

[4] ocs.kkg.033, *Telescope Control Console Track Preliminary Design*, Kim Gillies, Shane Walker, Steve Wampler, Gemini Observatory Control System Group, 1995.

[5] ocs.kkg.014, *Observatory Control System Software Requirements Document*, 6/5/95, Gemini Observatory Control System Group, 1995.

[6] ocs.kkg.016/06, *Observatory Control System Development Plan*, Gemini Observatory Control System Group, 1995.

[7] ocs.ocs.002, *OCS Physical Model Description*, Gemini Observatory Control System Group, 1995.

[8] *Object-Oriented Modeling and Design*, James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, Prentice-Hall, 1991.

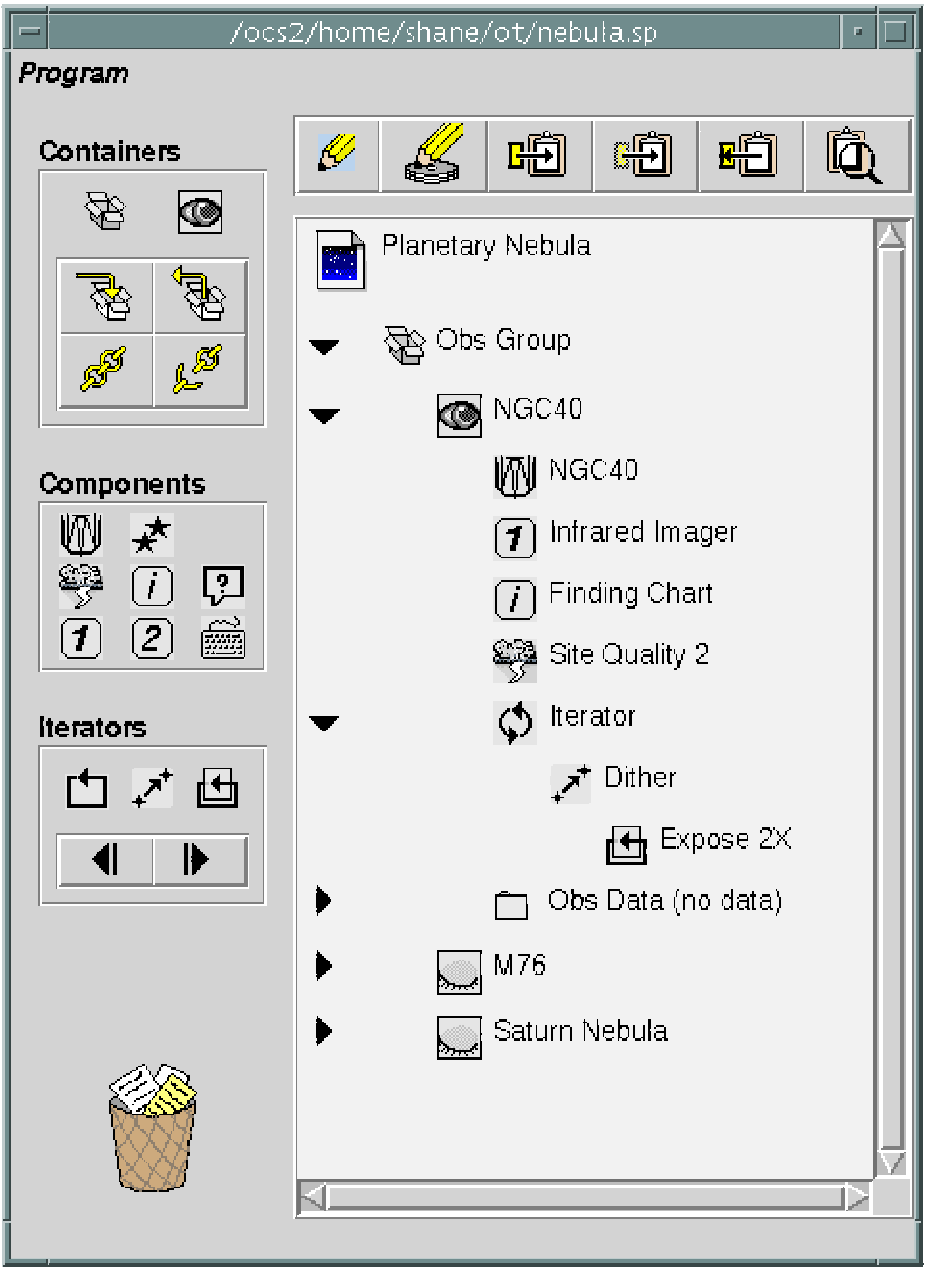
[9] *Journal of Object-Oriented Programming*, May 1994, October 1994, November/December 1994, February 1995, March/April 1995, May 1995, James Rumbaugh, *Modeling and Design* column.

## 4.0 Document Revision History

**First Release —** July 21, 1995. Pre-release draft.

**PDR Release —** August 16, 1995.

*FIGURE 1.* Observing Tool Prototype Screen



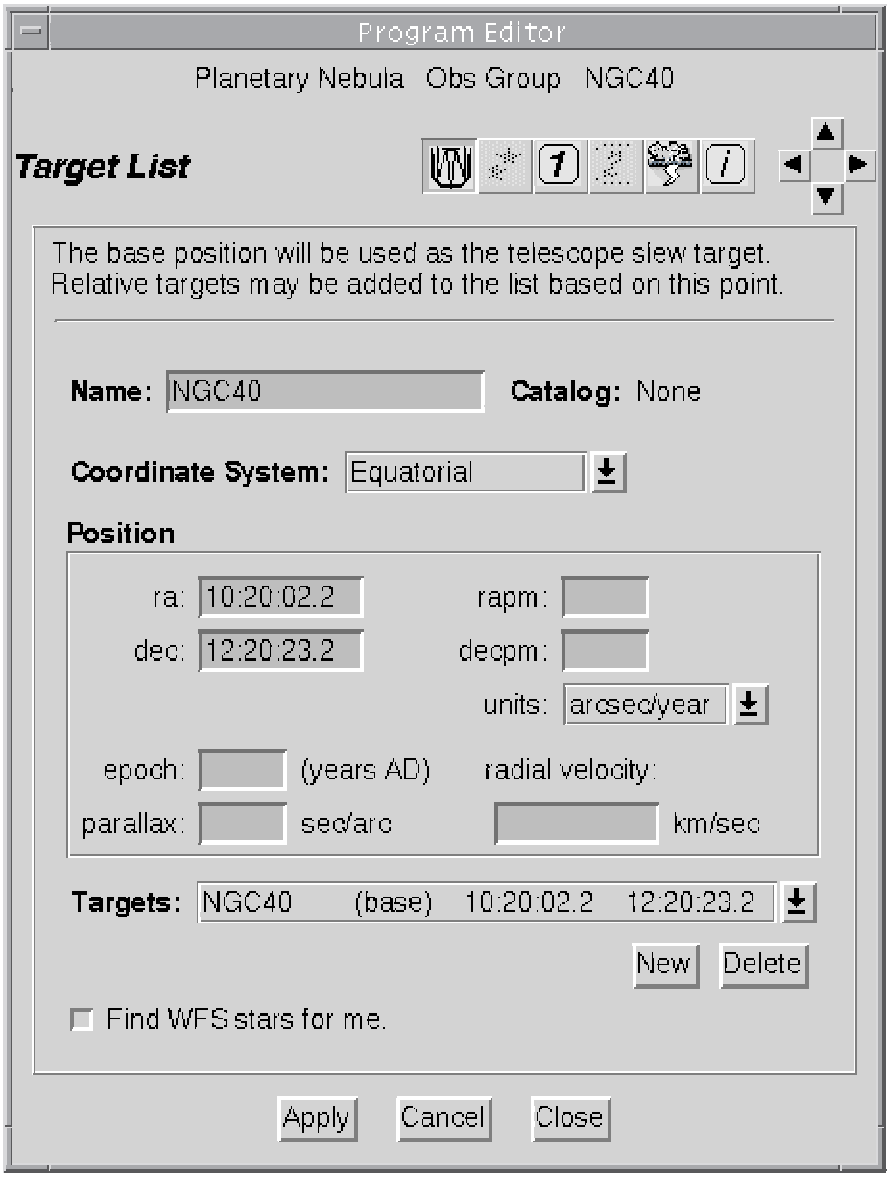
## 5.0 Summary of the OT Prototype

The OT prototype was created to try new ideas and identify problems. It provides a working description of OT concepts that can be used to point out problems and make suggestions. A picture of the main screen of the prototype is shown in Figure 1 on page 3 for reference.

This screen is used for editing the structure of a science program document. The program is represented by various icons in an expandable, hierarchical outline format. For instance, observations are represented by “eyeball” icons. The NGC40 observation has been expanded to show its contents while M76 and the Saturn Nebula are both collapsed. Components can be cut/copied/pasted and dragged/dropped in familiar ways.

The contents of a component are edited by double clicking the appropriate icon in the outline. For instance, the form used to edit a telescope target list is shown in Figure 2.

*FIGURE 2.* A Prototype Form for Editing a Science Program Component



One of the goals of the prototype is to demonstrate how an observation, or groups of observations, can be specified in sufficient detail to permit service observing and (semi) automated scheduling. Another important goal is to show that, with the proper interface, this process can be relatively painless. However, it is not the purpose of this paper to review the details of the OT prototype or to discuss how it will be improved during the detailed design. Rather, the focus should be on identifying inter-track dependencies and discussing how the OT fits in with the remainder of the control system.

## 6.0 Studies/Decisions During Observing Tool Track

The reviewed, OCS Software Design Review states that the following study will be done as part of the OT detailed design step.

**Evaluation of the Observing Tool Prototype.** A prototype of the Observing Tool and its concepts has been developed ... The lessons learned from the development of this prototype and the comments of the astronomers who came in contact with this prototype must be evaluated.

This prototype has been reviewed at various meetings and conferences, including the Gemini Science Committee in Tucson on 4/29/95, and the New Modes of Observing conference in Hilo on 7/8/95. It has been generally well received, with suggestions for improvement here and there. These suggestions will be taken into account during the detailed design.

## 7.0 High-Level Design of the Observing Tool Track

The Observing Tool application is built upon the functionality of the Console Track Library [4], and the Planned Observing Support (POS) track [3]. The Console Track Library provides an interface that allows the OT to configure a console from a science program component, and to place a console configuration into a science program. The POS track provides the OT with the following:

- A Session Manager (SM) application that handles run-time interactions with the Observing Tool.
- An interface to the Observing Database for Science Program and Science Plan storage.

The SM is used by on-site operators to allocate system resources and execute observations. Except for interactive observing via consoles, all observing sessions will go through the SM. An OT, whether used by on-site or remote observers, will contact the SM to submit observations and/or monitor their progress. Observing information passes up from executors to the SM and then to the appropriate OT(s).

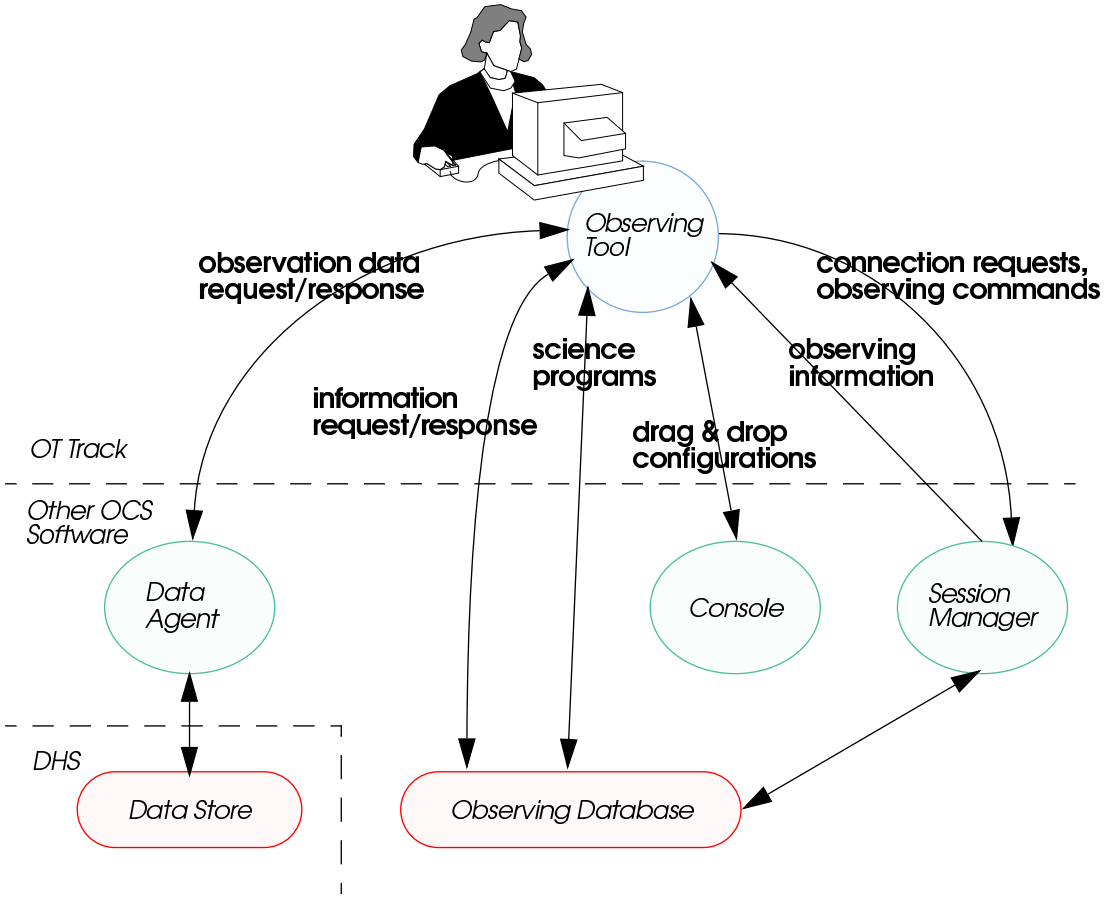
The OT will also retrieve observing data from the DHS data store. If requested fairly soon after the data is collected, it will be retrieved directly from the telescope site. Otherwise, the Data Archive may have to be contacted. In any case,

- *the DHS must provide a public interface that will permit an OCS process to access the data collected during an observing run.*

An agent process will be constructed by the OCS to communicate with the DHS, using the API that they decide upon. The OT will contact the agent process to collect the data.

A diagram showing the relationship between an OT, the remainder of the OCS, and the DHS is shown in Figure 3. The OT is built upon the functionality of the toolkit for the chosen graphical user interface (currently Tk), using agreed upon conventions. The SM will require information in the ODB (e.g., the Observing Pool) so a connection is shown in the diagram, but is not discussed in depth. A more detailed discussion of the SM is left to [3].

*FIGURE 3.* The Software Environment of the OT Track




The OT design is driven by the Software Requirements document [5]. The points relevant to the high-level, inter-track design are discussed below.

## 7.1 Planning Observations

A formal document, the Science Program, is required to specify observations in enough detail that they can be carried out by staff observers (*SR5*, *SR25*, *SR26*). The OT is designed to produce and modify science programs, and to submit/retrieve (parts of) them from the telescope sites (*SR27*). To support these requirements, the ODB interface must provide at least

- A means to browse all the programs for given users,
- Support for accepting and storing science programs, and
- Support retrieving/accepting observations and parts of observations for modification.

In addition to listing available programs, there may be other database information that the observer needs for the planning process. The ODB will provide access to this information.

## 7.2 Integrating Consoles with the OT

To provide an integrated operating environment (*SR2*), consoles and the OT should work together in a natural way. A console allows an operator to set up a future configuration for a system in much the same way as

an OT form. Using a console, an “Accept” button is pressed after the setup is entered and the system is commanded to match the configuration immediately. Using the OT, the setup information becomes part of a science program to be applied at an indeterminate time in the future. However, we can use the common features of both environments to simplify the observing and planning processes. The method of communication between an OT and a console is the familiar drag and drop protocol used in other portions of the OT. Namely, it will be possible to both

- Drag a configuration from an OT form and drop it onto a console, causing the console to match the configuration, and
- Drag a console configuration into the OT, either onto a form or as a separate unit in the science program.

This feature is intended to be used exclusively on-site. For safety reasons, consoles that directly control hardware systems should not be available away from the telescope (*SR41*, *SR42*).

## 7.3 Monitoring an Observation’s Progress

The OCS user interface must support interactive observing, remote observing, queue observing, and service observing (*SR22*, *SR30*, *SR25*, *SR26*). Furthermore, it must integrate the various observing modes to present one common observing environment (*SR2*). The Observing Tool is the application that fulfills these requirements.

Whether away from the telescope “eavesdropping” on a service observation, remotely executing observations, or on-site running a queue, the Observing Tool should present the same information to the user (limited by network bandwidth of course) (*SR40*). The types of information required should be worked out in the detailed design phase, but the requirement must be supported regardless.

From the OT’s perspective, the source of run-time observing information is the Session Manager (SM) application. Whether remote or on-site, the OT will have to establish a connection with the SM to monitor observations. Since the SM is used to allocate resources and execute observations, it is in the domain of the on-site system operator, and connections must be approved by him/her. For remote users, one can envision contacting the PI before his service observations are executed or before his remote observing session begins. The remote user could then start up his OT, submit a connection request and wait for it to be granted.

After establishing a connection, the OT will receive updates from the SM as an observation progress. If the connection is remote, then the information received may be a subset of that available to the on-site observer depending upon available bandwidth. However, the design does not make arbitrary distinctions between observing modes.

## 7.4 Executing a Science Plan

The OCS user interface must provide support for creating and executing a science plan constructed from the observations of one or more science programs (*SR32*, *SR33*). The OT will be used to fulfill this requirement.

A science plan is a queue of observations that should be carried out during an observing session. As discussed in the previous section, a connection with the Session Manager is required to interact with executing observations. To actually control which observation is executed next, the required system resources must be granted by the operator using the SM. It is ultimately the responsibility of the on-site operator to approve the execution of observations in the GCS, but the observer can control their ordering using the OT.

Executing and ordering observations involves sending information to the SM application. The SM will need access to information in the Observing Database to actually start an executor and run the observation. Regardless of the details of how the observation is carried out in the underlying system though, the SM and OT must have an interface that permits the OT to:

- Specify which observation(s) should be executed next, and

- Start, stop, or pause the ongoing observation.

Obviously, the monitoring information discussed in the previous section is used for executing a science plan as well.

## 8.0 Interfaces

The OT application communicates with other OCS processes (Consoles, the Session Manager, the Observing Database Agent, and the Data Agent). The communication between the OT and OCS applications will utilize the internal OCS Message System, which is TBD. The information that must be passed between the OT and other OCS applications has been detailed above.

## 9.0 Development Plan

The Observing Tool is among the highest level software products in the OCS. Accordingly, it relies upon the other tracks, particularly the Planned Observing Support Track, to be fully functional. However, the OT design should be able to proceed fairly independently of the other tracks since the types of communications required are known. The OCS development plan is discussed elsewhere [6].

## 10.0 Usability Testing

To insure that the OT application is of the highest quality, input from observers will be solicited whenever possible. An experienced, volunteer “OT testing team” should be assembled when the detailed design phase begins. This team will be called upon to evaluate the OT at various points as it is developed. Their input will then be folded back into the design for future evaluations. A critical consideration here is the size and makeup of the testing team. If the team is too small then we risk producing a tool that has been too finely customized for a small subset of users. However, if we make the tool available to the world then we risk being inundated with conflicting and often irrelevant viewpoints.

The following approaches, modified from the TCC Track design, also apply to the OT Track:

- The design of the OT will be solely in the hands of the OCS programming team. Their experiences with similar systems will guide the screen layout and design. The OCS programmers will be responsible for producing a usable system.
- The design of the OT will fold in the useful innovations of the operator interfaces of telescope control systems the OCS programmers visit.
- Any suggestions from the project scientists and the results of any discussions on operations will be folded into the design.
- All work will adhere to Gemini GUI style standards.

To ensure that operations staff programmers can make changes that will inevitably be required to the OT once the telescopes go on-line, the following is an OT track implementation goal.

- The implementation of the OT will be done to make layout and cosmetic changes to the consoles as simple as the chosen GUI toolkit allows.

# 11.0 Documentation

The documentation for the OT track will follow the documentation requirements in the OCS SDR documents (*SR66*, *SR67*, *SR68*). The OT track must provide both user applications and testing software.

## 11.1 Observing Tool User Application Documentation

**The Observing Tool Manual.** The Observing Tool is a large application with many features. It is intended to be the primary observer interface to the Gemini Telescopes. Therefore it must be accompanied by appropriately detailed documentation.

## 11.2 Observing Tool Testing Documentation

At this time there is no Gemini software requirement to provide automated testing of GUI programs. However, the OT must communicate with many other processes to fulfill its functionality. Some of these communications can be tested and documented without the graphical portion of the OT.

# 12.0 Deliverables

The primary deliverable of the OT Track is the Observing Tool application. Extensive documentation must accompany the OT as well since it represents the primary observer interface to the Gemini Telescopes.

# 13.0 Modelling the Observing Tool

The OCS has been modelled using the Rumbaugh (OMT) object-oriented design methodology. The overall design is present in [7], using concepts and notation from [8] and [9]. An understanding of this material is essential for this section.

The Observing Tool is just another instance of an OCSApp that interacts with other OCSApps. An initial decomposition of the OT into its Subject and Views is attempted below. The Controller interface is not stressed since a prototype exists that serves to get the idea across.

## 13.1 OT Subject and Views

The ApplicationSubject of the OT consists of three main entities:

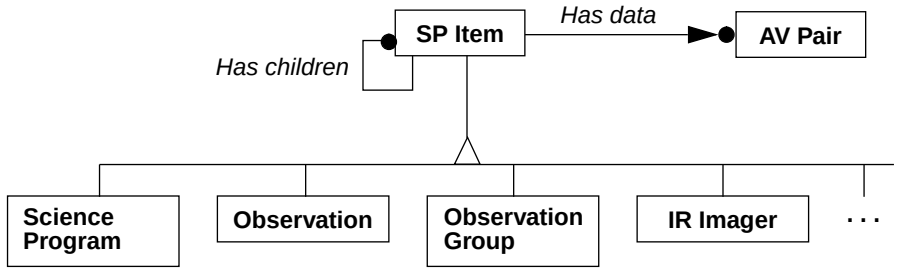
- Science Programs,
- Science Plans, and
- Run-Time Information

Each is discussed in the following sections along with the views presented to the user. Other information, such as available resources and programs may be found in the subject as well, but these are not covered in detail.

### 13.1.1 Science Program

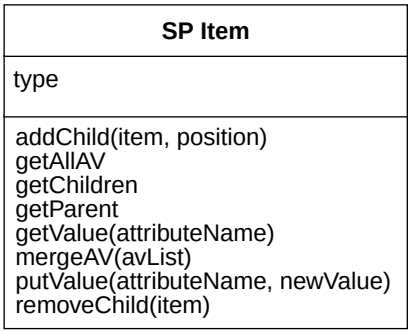
The Science Program (SP) is the focus of the Observing Tool. Programs are created, edited, and stored with the OT, and SP components make up Science Plans as well. Science programs are composed of a number of distinct objects including Observations, Observation Groups, Iterators, and links to other objects. However, each type of object (each class) has a great deal in common with the other classes. Each has data consisting of a set of attribute/value pairs and is related to zero or more “children” objects. The children are used to express the SP hierarchy discussed in [1]. The object model for science programs is presented in Figure 4 below.

*FIGURE 4.* The Science Program Object Model




The SP Item (Science Program Item) Class is an abstract class that can be instantiated to any of the concrete program elements such as observations and instrument configurations. At a minimum, SP Items will contain the attributes and methods depicted in Figure 5. The exact specification of methods should be left to the detailed design, but this set is indicated here (and discussed in Table 1) to illustrate these ideas.

*FIGURE 5.* The SP Item Abstract Class




The Observing Tool will present at least two views of the program (see Figure 1 on page 3 and Figure 2 on page 4 for prototypes). The first view is a hierarchical outline that may be expanded and collapsed to obtain the desired level of detail. The controller uses this view to permit editing the program's structure. The contents of individual items are edited through another view, a form that presents the item's attribute/value data in a meaningful way. The methods of SP Items support these views as discussed below.

*TABLE 1.* SP Item Methods

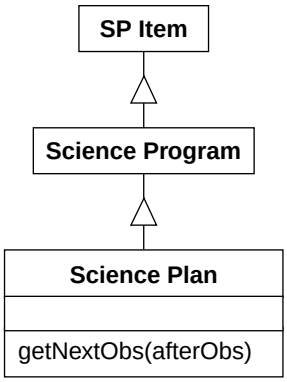
| Method | Use |
|---|---|
| addChild(item, position) | Add a child SP Item at the indicated position. This method is used to establish the hierarchy of SP Items. |
| getAllAV | Return a list of all the attribute/value pairs associated with the item. This is useful when the item is copied. |
| getChildren | Return all the children SP Items associated with item. The list of children is needed for many operations, including expanding and collapsing the hierarchy. |
| getParent | Return the "parent" SP Item into which the item has been nested. This is important for many operations, including moving an item within the hierarchy. |
| getValue(attributeName) | Return the value associated with a particular attribute. This method is useful for initializing a form containing widgets that are used to manipulate the attribute. |
| mergeAV(avList) | Merge the given attributes and values with the existing attribute set, replacing the values of existing attributes. Drag and drop operations will use this method. |
| putValue(attributeName, newValue) | Update a specific attribute with a given value. This is used when editing a form for the SP Item. |
| removeChild(item) | Remove the given item from the set of children associations. Move and delete operations will make use of this. |

The "type" attribute in an SP Item is set to the concrete class name that is instantiated from SP Item. It is needed so that restrictions on certain operations may be enforced, for instance "Science Program" items cannot be children of "Observation" items. Other attributes will no doubt be needed as well.

### 13.1.2 Science Plan

The Science Plan is really just a Science Program that is interpreted as an ordered list of observations. Accordingly, it is viewed as an extension of the Science Program class that adds a few new methods. At a minimum, a method is required to get the "next" observation after a given observation.

*FIGURE 6.* The Science Plan Class




Science Plans are constructed by adding observations from one or more programs. The observations should not be copied into the plan, but rather should be referenced from the plan. Programs will contain references to plans as well, for instance for shared calibrations that are specified once in a science plan for many observations. The “reference” objects are just another subclass of SP Item.

Both views of science programs should be available for plans as well. In addition, a special Plan Progress view is required (see the prototype in chapter 5 of [1]). This view shows the plan as a queue of observations, where the size of an observation’s icon is relative to its duration. This view is just a projection of the Science Plan which has been flattened to show only a single stream of observations. When actually running a plan with the OT, the current time can be indicated on the view and completed observations can be shown in a different color.

The Session Manager develops the Science Program and Plan objects in its section on the Observing Database [3]. This document should be consulted for a more complete picture of how the Planned Observing Support track will make use of programs and plans.

### 13.1.3 Run-time Information

Since the OT can be used both to monitor and to execute observations, a certain amount of run-time information should be part of the Application Subject. Examples of this information include the telescope and instrument statuses, the current observation being executed, the acquisition camera view, and a quick look image. Since the source of run-time information is another OCSApp (the Session Manager), it is acquired from the SystemSubject. The view on this subject should be familiar to the observer from his experience with other observatories.

## 13.2 Use Cases

The Observing Tool is used by observers; whether or not they are staff astronomers, no distinction is made in the interface presented to the user or in the types of interactions that may occur. The staff has access to all programs from any observer, whereas non-staff observers only have access to their own programs. However this does not affect the way that the OT is used by either group. From the point of view of the OT Track, there is a single “Observer” actor that initiates events in the system.

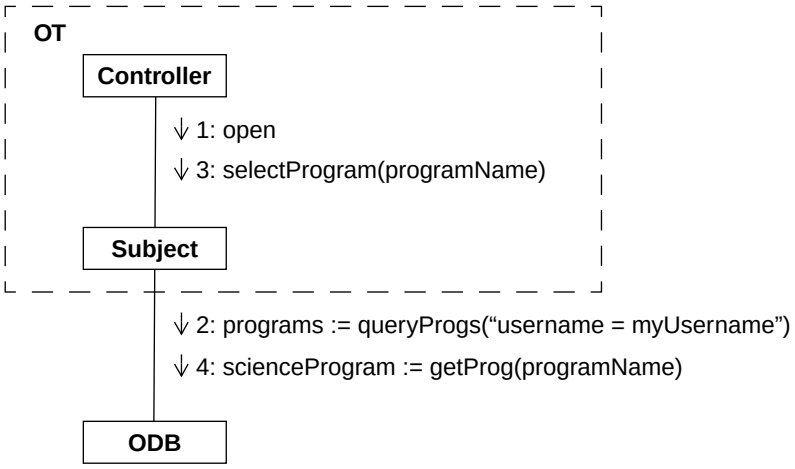
The OT also interacts with the observation data store, with the Observing Database, with consoles via drag and drop, and with the Session Manager application. To show interactions among the various objects, a set of “use cases” has been developed. Since there are too many types of interactions to detail each one individually, a set of some of the fundamentally different ways that the observer interacts with the system is listed. Representative instances of each are indicated below and then explored in the following sections.

1. **Editing a Science Program or Plan**
2. **Using the OT with consoles**
   a. Drag a program item onto a console
   b. Place a console configuration in a program
3. **Monitoring an executing observation's progress in the system**
4. **Executing an Observation in a Plan**

### 13.2.1 Case 1, Editing a Science Program at the Site

This case begins with the assumption that the observer has established a connection to the Observing Database (ODB). He now wishes to edit a program that was previously stored in the ODB. An object interaction diagram corresponding to this case is presented in Figure 7.

*FIGURE 7.* Use Case 1, Opening a Science Program at the Site




For the purpose of modelling the flow of information between objects in the OCS, the Observer's interaction with the Controller is not important. Hence, the source of all events in these diagrams is taken to be the Controller itself; it is the surrogate for the user. Of course, an Observer would have to be physically present clicking on buttons to cause the Controller to generate events.

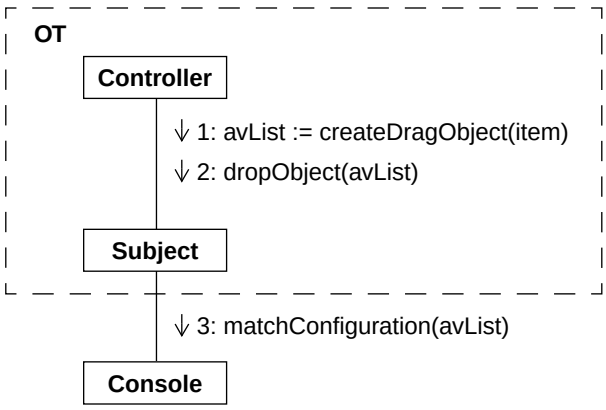
The interaction begins when the Controller requests an "open science program" operation. The ODB is then queried in step 2 to obtain a listing of the available programs for the given user. The programs returned as a result of this operation are listed in a file selection box (in SVC terminology, a view on the available programs is presented). One is eventually selected causing the Controller to generate event 3. After a reference to the program has been retrieved in the final step, the Science Program view is updated to show the new information.

Since a *link* to the information is returned in step 4, when a change to the program is applied by the Controller, the updated information is stored in the ODB.

### 13.2.2 Case 2.a, Dragging a Science Program Item onto a Console

In this use case, the observer drags an item (such as a Target List) from the science program, and drops it on a console. This interaction is illustrated in Figure 8.

*FIGURE 8.* Case 2.a, Dragging a Science Program Item onto a Console




The interaction begins when the observer presses the appropriate mouse button on the program item. This causes the Controller to command the Subject to create a “drag object”. This is essentially a copy of the item’s attribute/value set. The view on this object is the icon that is moved around the screen with the mouse. When the observer drops the object on a telescope console, the console is commanded to match the configuration specified by the attribute/value set.

As discussed above, an implementation that closely matches BLT’s drag&drop utility is envisioned. Whether BLT is ultimately used, the attribute/value list corresponding to an item must be saved when the drag and drop session begins, and this information must be transferred to the destination console when the session ends. The exact sequence of operations in the implementation may differ however.

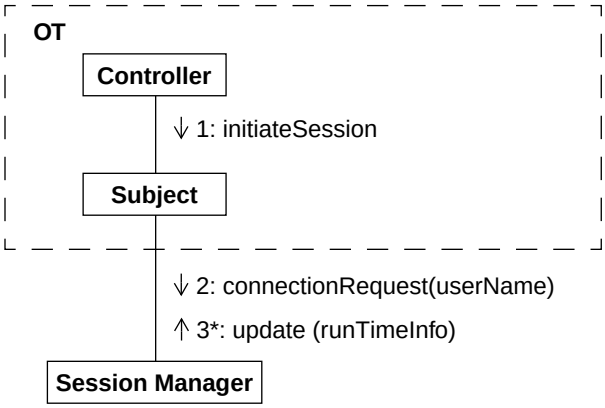
### 13.2.3 Case 2.b, Dragging a Console Configuration onto the Science Program

This case is similar to the previous case, except that the flow of information is reversed. Also, the console configuration can be dropped on one of two views in the science program, and the resulting actions vary accordingly. In one case, the configuration is dropped onto the outline view, causing a new SP Item to be created and added to the program. In the other case, the configuration is dropped onto the editing form for a particular item, causing the form’s widgets to match the configuration.

### 13.2.4 Case 3, Monitoring an Executing Observation’s Progress

To monitor an observation, a connection with the Session Manager application must first be established. From the point of view of the Observing Tool, the Session Manager is the source of run-time observing information as illustrated in Figure 9.

*FIGURE 9.* Case 3, Monitoring an Executing Observation's Progress




To initiate a session, the OT must provide access permission information, initially specified as a userName. The system operator at the site must create the session (if it does not already exist) to establish the connection. He will know which session the user is interested in monitoring and can decide whether to permit the connection. More than one OT can monitor a given session, though only one can control the execution of observations (see [3]).

Once the session is established, any run-time information generated as a result of executing the observations in the session is passed to the OT. The view on the run-time subject would then be updated accordingly. The asterisk on step 3 indicates that this event can occur repeatedly.

### 13.2.5 Case 4, Executing an Observation in a Plan

When the OT is being used to execute observations from a plan, it is sharing the information in the plan with the Session Manager application of the Planned Observing Support Track. In other words, the same plan is part of the OT's Subject and the Session Manager's Subject. Changes made to the Plan from either application must be reflected in the other.

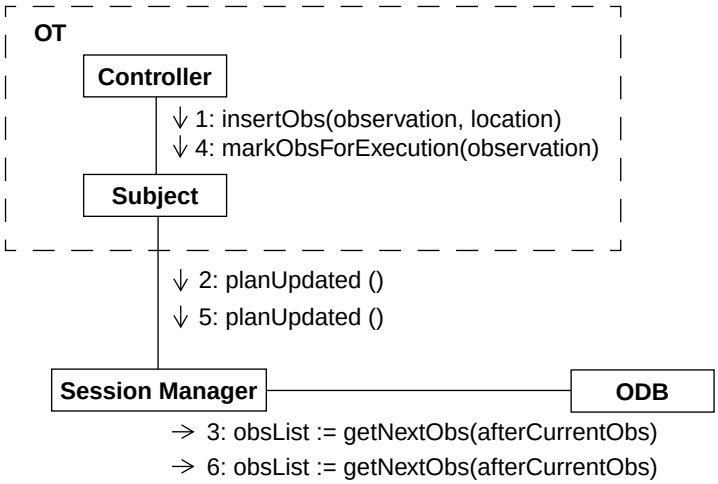
There are many ways that the high-level interface to the plan execution facility can be organized. The details must be left to the detailed design phase, but a likely scenario is presented here. The information that must flow between the various applications is known however so the scenario is useful for modeling object interactions.

When executing a plan with the OT, the user is given control over ordering the observations in the plan and marking which observations are ready to be considered for execution. When an observation has been marked, it shows up in the session granted to the observer in the Session Manager (SM). Before a "ready" observation begins execution, it may still be rearranged using the OT, and these changes should be reflected in the SM. The object interaction diagram in Figure 10 shows what happens when an observation is inserted into the plan and then is marked for execution.

The observer first inserts an observation into the plan using the controller. The observation and location argument types are unspecified here because we want to focus on which events are sent and what information has to be imparted, not the implementation details. The change to the Science Plan Subject causes the plan in the Observing Database to be updated since the OT plan is a link to a plan in the ODB. Since the OT is connected to a session in the SM, a planUpdated() message is sent in step 2. The SM then consults the ODB to get an updated list of observations. Note that the ODB may support an event notification capability. In this case, the ODB would notify clients whenever the data they are monitoring is updated, and the planUpdated() method would not be needed.

The observation is marked ready triggering an update in the Session Manager as well (steps 4,5). As before, an updated list of observations is then fetched from the ODB and displayed on the SM GUI.

*FIGURE 10.* Case 4, Inserting an Observation into a Science Plan




The observer also controls when he wants to stop/pause/continue an observation. The system operator sitting at the SM has the ultimate control over observations, but it is expected that in most cases he will simply place the session in “auto” mode, immediately accepting the observer’s input. For instance, if the observer decides to stop an executing observation, the SM stopObs(obs) method will be invoked by the OT.

## 13.3 Modelling Summary

This section has presented additional design information for the Observing Tool track. A simple model of Science Program/Plan data was given followed by a discussion of the SVC decomposition of the OT. A few object interaction diagrams then provided an indication of how the OT interacts with the products of the Planned Observing Support track. For more detailed information on these interactions, see [3].

Created:July 6, 1995
Modified:March 18, 1996

TCC PDR

Gemini Observatory Control System Report

# Telescope Control Console Track Preliminary Design

**Kim Gillies, Shane Walker, Steve Wampler**

ocs.kkg.033-TelConTrkPD/05

**This report presents the preliminary design of the Telescope Console Track, a part of the Gemini Observatory Control System.**

## 1.0 Introduction

The Telescope Control Console Track (the Console Track in the Work Breakdown Schedule) is one of the development tracks in the development plan for the Observatory Control System (OCS) of the Gemini Telescope. The following statements are from the Software Design Review OCS Development Plan [4].

> Building upon the Interactive Observing Support Track, this track is focused on the development of the consoles that will be used at the site by operators/staff to control the telescope and its related subsystems. (page 3)
>
> This track provides the telescope consoles and is built directly upon the previous phase. Once completed, interactive control of the telescope is possible for hardware and software system testing. Interactive observing is possible using the engineering interfaces provided by instrument builders. (page 7)

The job of the Telescope Control Console (TCC) track is to produce the console applications that will be used by the operators to control the telescope and its subsystems interactively during the commissioning and operational operations phases. This track also includes any consoles the OCS must produce to make the job of the operator simpler that may not be directly related to the telescope and its hardware (an example might be a star catalog access application).

This report presents the preliminary design of the TCC track to a depth such that the track can continue on in its development independently of the other OCS tracks. The following information is contained in this report.

- A high-level preliminary design for the track.
- The dataflow between applications in the TCC track and the other OCS tracks.
- The dataflow between applications in the TCC track and the other principal systems.
- A preliminary list of the known TCS consoles.
- The status of the information from other principal systems and the timetable for its arrival.

This paper often focuses on EPICS-related issues and problems since the consoles of this track are tied to the TCS, which is a CAD-based EPICS system.

## 2.0 Acronyms

| | |
|---|---|
| API | Application Programmer Interface |
| CLL | Command Layer Library |
| CTL | Control Track Library |
| GUI | Graphical User Interface |
| IOC | Input/Output Controller |
| IOI | Interactive Observing Infrastructure |
| OCS | Observatory Control System |
| ODB | Observing Database |
| PD | Preliminary Design |
| PS | Principal System |
| PSA | Principal System Agent |
| SIR | Status Information Record |
| SAD | Status Alarm Database |
| TCC | Telescope Control Console |

## 3.0 References


[1] SPE-C-G0037, *Software Design Description*, 9/9/94, Gemini Controls Group, 1994.

[2] ocs.kkg.032, *Interactive Observing Infrastructure Preliminary Design*, Kim Gillies, Shane Walker, Steve Wampler, Gemini Observatory Control System Group, 1995.

[3] ocs.kkg.014, *Observatory Control System Software Requirements Document*, 6/5/95, Gemini Observatory Control System Group, 1995.

[4] ocs.kkg.016/06, *Observatory Control System Development Plan*, Gemini Observatory Control System Group, 1995.

[5] ESO *Graphical User Interface Common Conventions*, Doc. No. GEN-SPE-ESO-00000-0266, Issue 1.0, 10/05/94.

[6] ocs.ocs.002, *OCS Physical Model Description*, Gemini Observatory Control System Group, 1995.


## 4.0 Document Revision History

**First Release —** June 9, 1995. Pre-release draft.

**PDR Release —** August 16, 1995.

**Final PDR Release —** September 27, 1995.

## 5.0 Studies/Decisions During Telescope Control Console Track

The reviewed, OCS Software Design Review states that the following decisions will be made as part of the TCC detailed design step.

**Graphical User Interface Standards.** The requirements for the graphical user interface look and feel and GUI toolkit that will be used for all OCS observer and staff tools will be reviewed in order to take account of any changes to project requirements. One or more toolkit implementations will be selected. The ESO/VLT GUI Common Conventions [5] style guide, a Gemini Standard, will be reviewed and any changes or additions will be proposed to ESO and the project for consideration.

The baseline decision to use Tk has been made by the project in this area, but it is important to re-visit this decision one more time in light of the fast-paced industry development in this area.

Associated with this trade study will be analysis and choice of GUI development-related software tools, if needed.

## 6.0 The Physical Model of the OCS and the Telescope Control Console Track

The physical model of the OCS [6] describes the layering present in the OCS software system. For the TCC track, the OCS consists of a set of cooperating applications. Each application is an instance of OCSApp and it inherits all the functionality provided by the SystemSubject subystem. The physical model of the OCS as a set of OCSApps is shown in Figure 1.

The consoles are an important part of the TCC track. Designing and implementing a console consists of designing the AppSubject, Views, and Controller that are unique to the console application and implementing them in the chosen language.

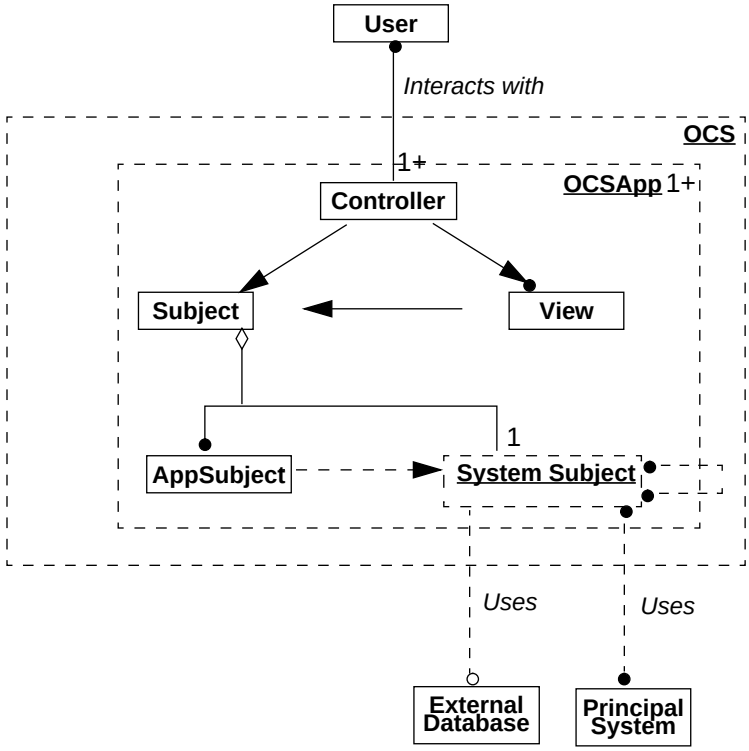
So the model for an OCSApp is the basis for the models for the specific console applications that will be done during the TCC track. An example of modeling a console was discussed in [6] and will not be repeated here.

The functionality of the SystemSubject (the Command Layer Library) provides the AppSubject objects with command and status communications capabilities. This allows the console designers to focus on the design of their application-specific code.

At this time, it is impossible to model the individual console instances in any further detail. Each console detailed design will be accompanied by a physical model.

A software library is also produced as a product of this track. The physical model of this library will be discussed later in the paper.

*FIGURE 1.* OCSApp Subsystems and Classes




## 7.0 High-Level Design of the Telescope Control Console Track

The applications in the TCC track are built directly upon the functionality of the Interactive Observing Infrastructure (IOI) track [2]. The IOI track provides TCC applications with the following capabilities:

- Applications can send commands and acquire the status of the other principal systems.
- Applications can send commands and acquire the status of other OCS applications.
- Applications can wait for the completion of sequence commands executing in the other principal systems or other OCSApp instances.
- Applications can subscribe to the status information of the other principal systems and other OCSApps including alarms and health.
- Applications acquire access to the functions of other systems through the IOI track functionality.
- Applications can log messages to a file and to the DHS logging system.

The Command Layer Library of the IOI is dynamically linked with every application in the TCC track. The CLL API provides a programmer interface to the above functionality. A scripting interface (probably written in Tool Command Language) will also be available as a product of the IOI track.

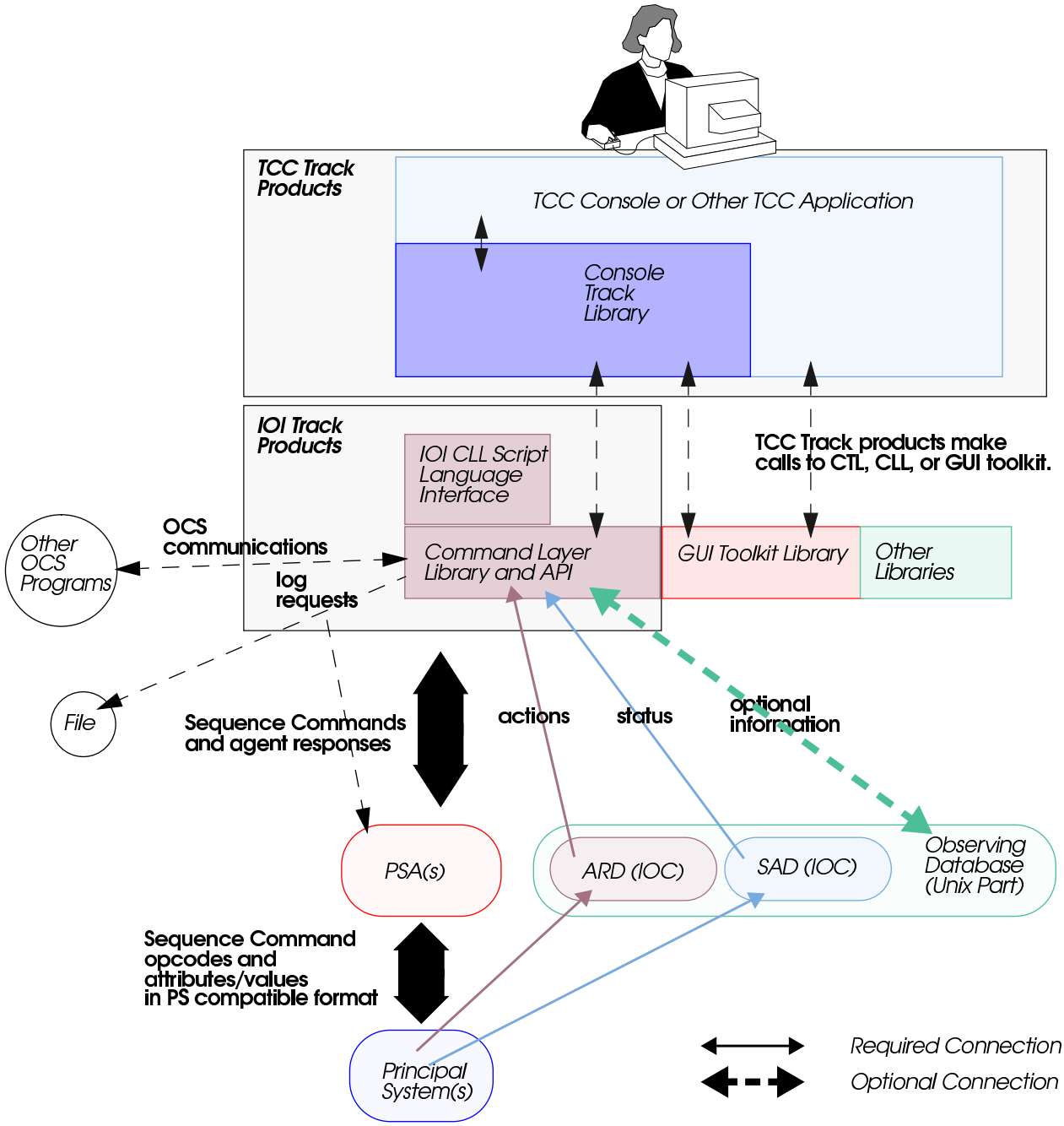
A block diagram showing the relationship between console applications in the TCC and the IOI track is shown in Figure 2. The consoles are instances of OCSApp.

Applications are built upon the functionality of the IOI track CLL and the toolkit for the chosen graphical user interface (currently Tk). Also shown is a Console Track Library (CTL) that will be produced as part of

the TCC track to provide common console functionality at a level higher than the CLL. Some functionality of the CTL is known now and is discussed later in this report. Other functionality will be determined by the TCC track developers later in the design/implementation process.

The dataflow to and from the other principal systems through the CLL is also shown for completeness but is not discussed here. (See [2].)

*FIGURE 2.* The Software Environment of the TCC Track



## 8.0 Interfaces

The TCC track applications communicate with other OCS processes including:

- Observing Tool - to save and restore their configuration.
- Observing Database - to obtain shared system data. This communication uses the OCS Message System or a ODB specific Service in the CLL. The implementation of the ODB is not known at this time.
- Any other OCS CLL-based application.

The TCC track applications must also communicate with the other principal systems.

- To send commands and receive synchronous and asynchronous responses.
- To receive status.

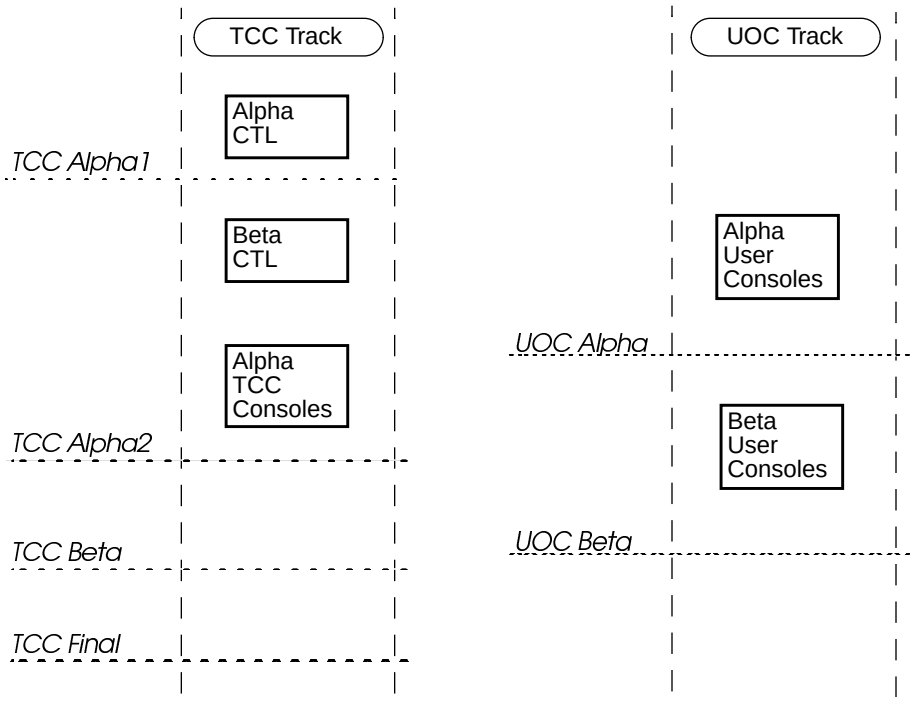
Inter-OCS communication uses the OCS Message System, and the software interface for these communications is provided by the Command Layer Library OCS Service product. Both are part of the Interactive Observing Infrastructure track. The requirements for this CLL interface are in the IOI track document [2].

## 9.0 Development Plan

The Telescope Control Console track relies completely upon the functionality of the Interactive Observing Infrastructure track and must follow that track in the development process. The OCS development plan is discussed elsewhere [3].

The TCC track development plan will be phased to allow maximum track parallelism. The CTL will be released in alpha and beta form before the entire TCC track is completed so that work on the User/Observer Track consoles can begin if needed. After Alph2 of the TCC track, there will be a one Beta and then Final Release as specified in the OCS development plan.

*FIGURE 3.* TCC Track Delivery




## 10.0 Principal Parts of the TCC Track

The products of the TCC track are the TCC console applications and the CTL. Each will be discussed only briefly here since the role of the PD is to identify track interdependencies within the OCS. The internals and software interface of the CTL and the specific design and layout of the consoles can not be delineated at this time.

### 10.1 Applications and Consoles

The SDD [1] provided a preliminary list of required hardware-oriented telescope control consoles. The list that is shown here is from Chapter 5 of the SDD. The role of each console is also shown. The fact that the screens are separate in this list is not a requirement that there be a separate console for each screen. Some screens in the table may be combined within a single application.

*TABLE 1.* Modified Table 5-2 from the SDD

| Screen Name | Description |
|---|---|
| TCS Control Console | Primary operator interface. Allows the operator to perform common tasks related to normal telescope operations/observing. |
| TCS WFS Signal Routing Console | Allows the operator to visually determine and change the sources, destinations, and routing of wave front sensor related errors in the control system. |

| Screen Name | Description |
|---|---|
| Weather Monitoring Console | A status console that provides information from the Gemini and mountain top weather monitoring equipment/sensors. Content somewhat dependant on site. |
| Structure Temperature Monitoring Console | Provides a visual display of structure temperature distribution. |
| Mount Control Console | A console that provides low-level mount information.* |
| Cassegrain Rotator Control Console | Provides status and engineering information for the cassegrain rotator.* |
| Primary: Active Support System | Provides engineering status and fine-grain control of the primary mirror support systems. The functionality in these screens will probably be important to normal telescope operations. |
| Primary: Passive Support System | |
| Primary: Air Support System | |
| Secondary: Alignment System Console | Provides engineering and fine-grain control of the secondary hardware. The functionality of these screens will be important to normal telescope operations. |
| Secondary: Chopper System Control | |
| Secondary: Tracker Control Console | |
| A&G: Main Console | This screen provides a overall view of the operational state of the A&G system. |
| A&G: WFS 1 Console | Low-level screen specific to PWFS1. |
| A&G: WFS 2 Console | Low-level screen specific to PWFS2. |
| A&G: Calibration Sensor Console | Provides a detailed status display of the operation and state of the calibration WFS. |
| Enclosure Console | Provides operator access to the control of the enclosure functions. |
| Alarm Interface | An operator tool that focuses on the presentation of system alarms and errors. |
| Status Interface | Provides a single display of important telescope status information. The error budget display is located here if the error budget display is possible. |

* These requirement for these consoles might be satisfied by engineering consoles.

The table suggests that some console requirements may be satisfied by the engineering consoles constructed by the other work package groups. The issue of who develops OCS consoles often comes up and the following is the view of the OCS group. First, a few relevant OCS requirements [3].

*SR1* **Need**:E **Source**:ASDD/URD **Priority**:1
**Short Description**: VUI Purpose

**Description.** The VUI of the OCS is required to provide the user interface and related concepts astronomers and operations staff will use during the operations/maintenance phase of the Gemini Telescope's life cycle defined in the ASDD.

*SR2* **Need**:E **Source**:URD18 **Priority**:2
**Short Description**: The VUI must provide an integrated set of observing tools.

**Description.** This requirement means that the software that is used when interactive observing and the software that is used for planned observing must behave similarly, provide the same look and feel, and they must function together to provide what seems to be one observing environment. This is often called application interoperability. There should not be completely different sets of tools to support the various observing modes required in this document.

These two requirements state that the OCS consoles must be observer/operator oriented and integrated with the other OCS tools. The OCS group could formally define what it means to be integrated and define how a console becomes part of the OCS, but this process would add significant work requiring the OCS group to provide code, documentation, and assistance to the widely dispersed Gemini developers. This is not possible under the current OCS development plan. The following statements summarize the OCS group approach concerning the use and implementation of operator consoles.

- Engineering consoles for subsystems will not generally be duplicated by the OCS group. The operator will use engineering screens for engineering and observing screens for his normal system interactions.
- Operator/observer functionality that may be present in an engineering interface may be duplicated in an OCS console to improve the operator's efficiency or effectiveness.
- The OCS group may choose to modify some engineering consoles to integrate them with the OCS tools if that approach is the best for the OCS group.
- The OCS group will continue to work with the principal system groups to provide commands and status that is oriented towards the observing process.

### 10.1.1 Prototypes Generate Requirements for User Interface Toolkit

The prototype screens provide examples of how telescope information might be displayed on the screens; hence, they provide a basic set of requirements for what kinds of functionality must be available in any GUI toolkit used to construct the TCC consoles and applications.

- The ability to enter non-editable text with multiple styles and faces.
- Various kinds of buttons including momentary contact and toggle buttons.
- Combobox or menu buttons.
- Button groups with one of many and several of many behaviour. Also radio buttons or check boxes with the same behaviour as button groups.
- Data entry fields that allow the user to type a value or push a button to increment or decrement the value of the field.
- Various graph types including histograms, and strip-charts.
- Scrollable text displays. The toolkit must allow the user to interact with the contents of a text display to select lines and to mark a line with a graphical marker such as a check mark.
- Drag sites and drop sites including toolkit support.
- Toolkit must allow relatively complex schematics to be drawn on the console with images, buttons, and text connected by other graphical items such as lines or images.
- Toolkit must support the display of images and must allow interactions with images. For instance, it may be useful to click on a location in the primary mirror figure display to see an x/y position or actuator pressure on the mirror.
- Integrated help including spot-help and on-line manuals.

The baseline GUI toolkit, Ousterhout's Tk, does not contain this functionality without a number of third party additions.

### 10.2 Console Track Library

Some functionality will be common to all consoles built by the OCS group including the consoles constructed as part of the User/Observing Console Track. The SDD and the OCS SDR documents show and require some functionality that would be reasonably implemented in the CTL. The primary role of the CTL is to provide a programming library that will tie together the functionality of the CLL and the chosen GUI toolkit.

**Console Interactive Graphical Control Semantics.** The SDD and OCS requirements (SR23) state that consoles should visually display information that allows the user to determine when commands are executing and completed. The support for this feature is found within the CLL. The CTL would provide a standard way to display this information on a console screen.

**Status/Health Interface.** The CTL will integrate the status, alarm, and health functions of the CLL with the chosen GUI toolkit functionality.

**CLL and GUI Toolkit Integration.** Most GUI toolkits provide similar functionality. The CTL must tie the chosen GUI toolkit to the functionality of the CLL. For example, tieing a screen button callback to the CLL ability to send a command will be needed. This functionality builds upon both the CLL and GUI toolkits and belongs in the CTL.

**Observing Tool Integration.** Each OCS console must have the ability to include its configuration in the Observing Tool Science Program. This feature is very dependent on the capabilities of the GUI toolkit. The CTL will provide an interface to allow consoles to place their configurations into the Observing Tool and for the Observing Tool to configure a console from the contents of a Science Program.

#### 10.2.1 Physical Model of the CTL

The CTL is viewed as an extension of the SystemSubject subsystem since the CTL is to provide common capabilities for all OCSApp consoles. The new classes must be built upon the current SystemSubject classes and the capabilities of the GUI toolkit. The physical model for this library will be made part of the detailed design of the track.

## 11.0 Status of Principal System Information

The information that describes the public interface to the telescope and its subsystems must be available and settled before the TCC track begins. The content of the public interface must be agreed upon before the console work begins. The preliminary PDF document for the TCS interface is to be available at the TCS SDR in early September, 1995.

## 12.0 Usability Testing

The quality of a graphical user interface is generally better when the users of a product are involved in its development. However, the final users of the console products of the telescope control track, the telescope operators, will not be available for user involvement and testing of the TCC consoles. The following is the approach that will be used to provide the highest quality TCC consoles possible.

- The design and screen layout of the operator consoles will be guided by OCS programming team's experience with similar systems. The OCS programmers will be responsible for producing a usable system.
- The design of the operator consoles will fold in the useful innovations of the operator interfaces of telescope control systems the OCS programmers visit including Kitt Peak National Observatory; Keck Observatory; Canada, Hawaii, France Telescope, and the Very Large Telescope.

- Any suggestions from the project scientists and the results of any discussions on operations will be folded into the design of the operator consoles. Prototype screen designs for operator tools will be shown to and reviewed by project scientists during weekly meetings.
- All consoles will adhere to Gemini GUI style standards.

To ensure that operations staff programmers can make changes that will inevitably be required to the TCC consoles once the telescopes go on-line, the following is a TCC track implementation goal. The physical design of the OCS consoles supports this goal.

- The implementation of the OCS telescope operator consoles will be done to make layout and cosmetic changes to the consoles as simple as the chosen GUI toolkit allows.

# 13.0 Documentation

The documentation for the TCC track will follow the documentation requirements in the OCS SDR documents (SR66, SR67, SR68). The TCC provides all three kinds of OCS products: the CTL software library, user applications (consoles), and testing software for the CTL library.

## 13.1 Console Track Library

This library will be released with the following documentation.

**The Console Track Library Technical Document.** This document describes how the library works.

**The Console Track Library Programmer's Document.** This document describes how interactive consoles should be produced and how the programmer uses the CTT Library.

### 13.1.1 Console Track Library Testing Documentation

Some simple consoles will be provided to demonstrate the use of the console track library.

**The Console Track Testing Manual.** This manual will describe how to use the testing procedures required for the acceptance tests of the CTT Library.

## 13.2 Console Track Applications/Consoles

Each console or CTT application will be accompanied by user documentation. Some applications may have standalone manuals but the use of the integrated OCS operator consoles will be covered in one operator manual.

**The Gemini Telescope Operator Manual.** This manual will provide an overview of the control of the telescope using the tools, applications, and consoles provided by the OCS group. Individual chapters will be dedicated to the use of each of the consoles.

### 13.2.1 Console/Application Testing Documentation

At this time there is no Gemini software requirement to provide automated testing of GUI programs.

# 14.0 Deliverables

The deliverables of the track are as follows.

- The CTL and testing software.

- The system operator consoles.
- The documentation listed in this document and the OCS Development Plan.

Created:July 11, 1995
Modified:March 18, 1996

Gemini Observatory Control System Report

# User/Observing Console Track Preliminary Design

**Kim Gillies, Shane Walker, Steve Wampler**

ocs.kkg.034-UObTrkPD/04

**This report presents the preliminary design of the User Observing Console Track. A part of the Gemini Observatory Control System.**

## 1.0 Introduction

The User/Observing Console track (the Observing/User Screen Track in the Work Breakdown Schedule) is one of the development tracks in the development plan for the Observatory Control System (OCS) of the Gemini Telescope. The following statements are from the OCS Software Design Review Development Plan [4].

> This track develops the observing related consoles and Observing Tool screens associated with the site instruments. Since the instruments are developed external to the OCS, the instrument consoles must be developed later in the project. This track also provides the OCS PVWave interface. (page 3)
>
> This phase provides the integrated consoles for the individual instruments. This capability is required once instrument development nears completion. (page 7)

The primary purpose of the User/Observing Console (UOC) track is to provide the functionality that is needed to create integrated OCS observer tools that can access and interact with all the principal systems.

This track must occur late in the OCS development plan in order to follow the development of the site instruments that are being developed concurrently with the OCS. When this track commences each instrument will be refined such that its operations and modes are known and tested. Its Parameter Description Form, the public document describing and instrument's public interface, will be completed and reviewed allowing this track to begin.

This report presents the preliminary design of the UOC track to a depth such that the track can continue on in its development independently of the other OCS tracks within the structure of the OCS development plan. The following information is contained in this report.

- A high-level preliminary design for the track.
- The dataflow between applications in the UOC track and the other OCS tracks.
- The dataflow between applications in the UOC track and the other principal systems.
- A preliminary list of the known UOC consoles.

- The status of the information from other principal systems and the timetable for its arrival.
- Remaining decisions for the detailed design of the track.

### 1.1 Track Products/Deliverables

The following development tasks are included in the UOC track.

#### 1.1.1 Infrastructure Support

- Provide a shell environment that can be used by operations staff to develop observing scripts that integrate and interact with the principal systems. This shell environment will be based upon the PV-Wave product standard specified by the Gemini Project.
- Provide the scripting level support for the Asynchronous Data Reduction Facility of the DHS.

#### 1.1.2 User Interface Products

A major goal of this track is to *integrate* the use of the site instruments with the other software provided by the OCS Planned Observing Support track and the Observing Tool track.

- This track will produce any console applications that will be used by the operators and observers to interact and control with observatory instruments.
- This track will produce any forms or other screens required for the site instruments to integrate with the Observing Tool.

These last two bulleted items do not state that this track will produce these items for every instrument. Each instrument's functionality must be considered separately. Since they are completely undefined at this time it is difficult to state with absolute confidence how each will be integrated into the OCS observing environment.

## 2.0 Acronyms

| | |
|---|---|
| API | Application Programmer Interface |
| CLL | Command Layer Library |
| CTL | Control Track Library |
| GUI | Graphical User Interface |
| IOC | Input/Output Controller |
| IOI | Interactive Observing Infrastructure |
| OCS | Observatory Control System |
| ODB | Observing Database |
| PS | Principal System |
| PSA | Principal System Agent |
| SIR | Status Information Record |
| SAD | Status Alarm Database |
| SDRF | Synchronous Data Reduction Functionality |
| TCC | Telescope Control Console |
| UOC | User/Observing Console |

## 3.0 References

[1] SPE-C-G0037, *Software Design Description*, 9/9/94, Gemini Controls Group, 1994.

[2] ocs.kkg.032, *Interactive Observing Infrastructure Preliminary Design*, Kim Gillies, Shane Walker, Steve Wampler, Gemini Observatory Control System Group, 1995.

[3] ocs.kkg.014, *Observatory Control System Software Requirements Document*, 6/5/95, Gemini Observatory Control System Group, 1995.

[4] ocs.kkg.016/06, *Observatory Control System Development Plan*, Gemini Observatory Control System Group, 1995.

[5] ocs.kkg.038, *Planned Observing Support Preliminary Design*, Kim Gillies, Shane Walker, Steve Wampler, Gemini Observatory Control System Group, 1995.

[6] ocs.kkg.033, *Telescope Control Console Preliminary Design*, Kim Gillies, Shane Walker, Steve Wampler, Gemini Observatory Control System Group, 1995.

[7] ocs.ocs.002, *OCS Physical Model Description*, Gemini Observatory Control System Group, 1995.

## 4.0 Document Revision History

**First Release —** June 9, 1995. Pre-release draft.

**PDR Release —** 16 August, 1995.

**PDR Final Release —** 27 September 1995.

## 5.0 Studies/Decisions During the User/Observing Console Track

No studies are specified for this track.

## 6.0 The Physical Model of the OCS and the User/Observing Console Track

The physical model of the OCS [7] describes the layering present in the OCS software system. For the TCC track, the OCS consists of a set of cooperating applications. Each application is an instance of OCSApp and it inherits all the functionality provided by the SystemSubject subystem. The physical model of the OCS as a set of OCSApps is shown in Figure 1.

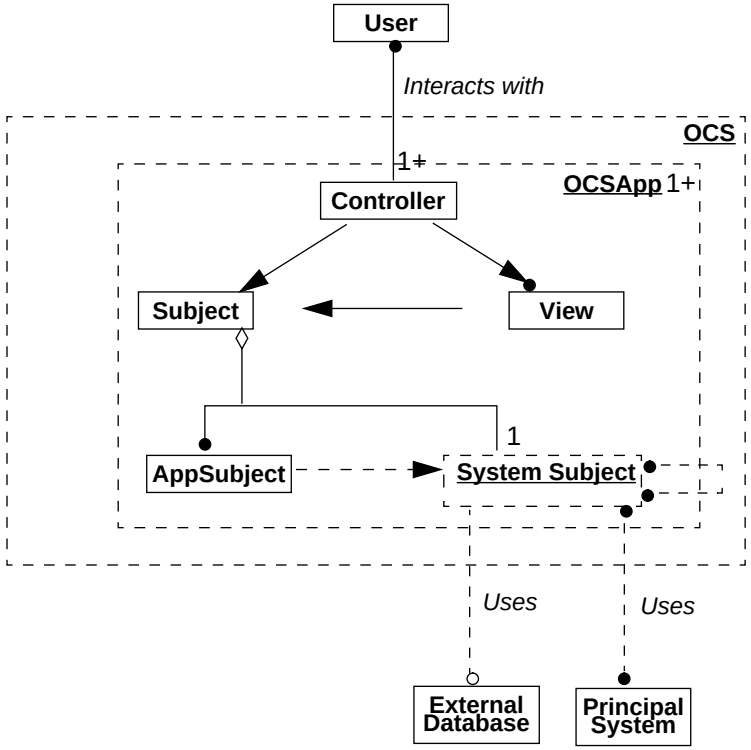
Observing consoles are an important part of the UOC track. Designing and implementing a console consists of designing the AppSubject, Views, and Controller that are unique to the console application and implementing them in the chosen language.

So the model for an OCSApp is the basis for the models for the specific console applications that will be done during the UOC track. An example of modeling a console was discussed in [7] and will not be repeated here.

The functionality of the SystemSubject (the Command Layer Library) provides the AppSubject objects with command and status communications capabilities. This allows the console designers to focus on the design of their application-specific code.

At this time, it is impossible to model the individual console instances in any further detail. Each console detailed design will be accompanied by a physical model.

*FIGURE 1.* OCSApp Subsystems and Classes




## 7.0 High Level Design of the User/Observing Console Track

The products of the UOC track are not too related to one another. The primary track work is focused towards track integration and adding features and functions that depend strongly on the products of other Work Packages and other OCS tracks. Each kind of product will be viewed separately in the following sections.

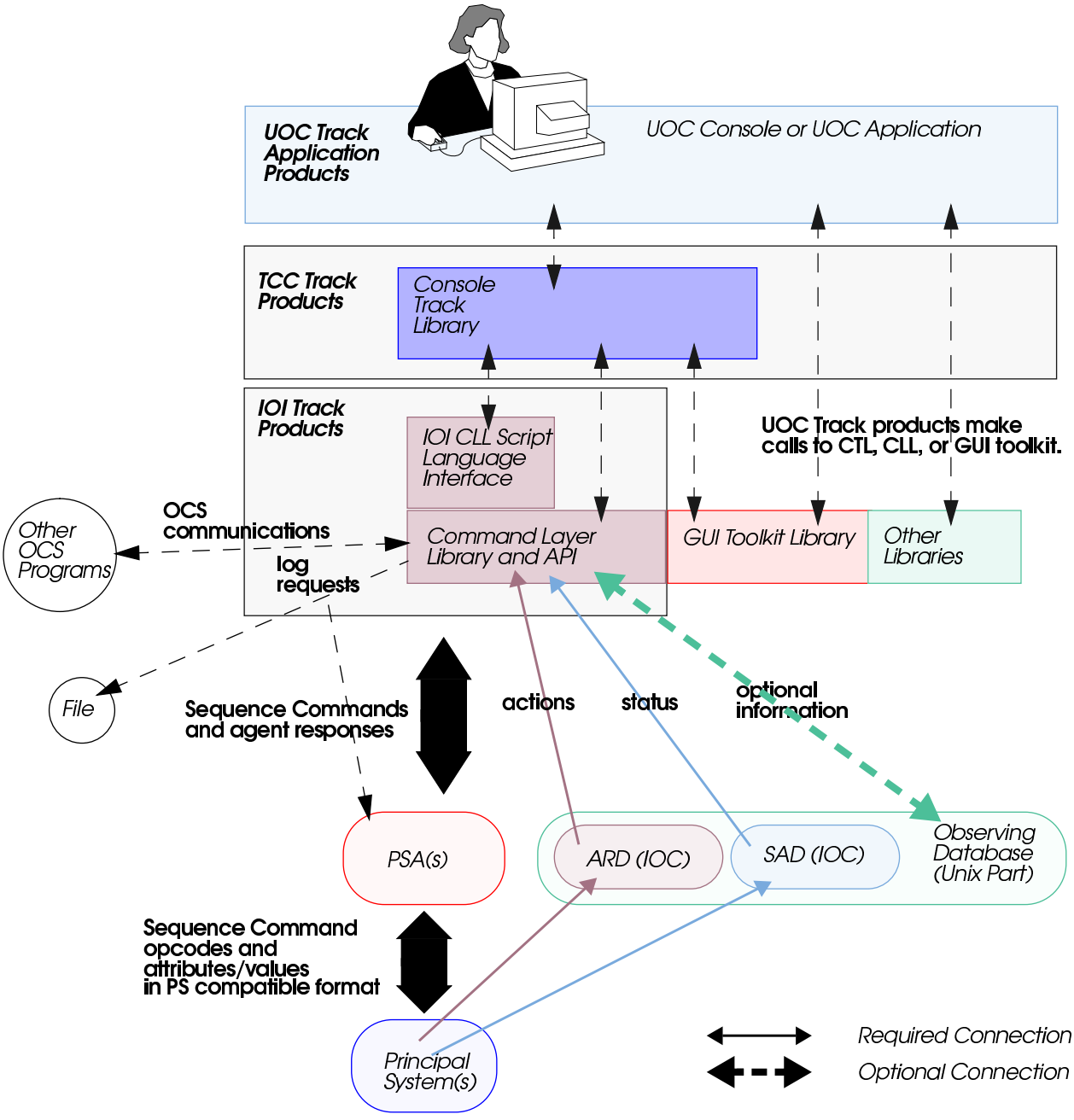
### 7.1 User Interface Application Products

The functionality of applications in the UOC track at the lowest level is based directly upon the functionality of the Interactive Observing Infrastructure (IOI) track [2]. The IOI track provides UOC applications with the following:

- Applications can send commands and acquire the status of the other principal systems.
- Applications can send commands and acquire the status of other OCS applications.
- Applications can wait for the completion of sequence commands executing in the other principal systems.
- Applications can subscribe to the status information of the other principal systems including alarms and health.
- Applications acquire access keys through the IOI track functionality.
- Applications can log messages to a file and to the DHS logging system.

The Command Layer Library of the IOI is dynamically linked with every console application in the UOC track. The CLL API provides a programmer interface to the above functionality. A scripting interface (probably written in Tool Command Language) will also be available as a product of the IOI track.

*FIGURE 2.* The Software Environment of an Application in the UOC




A block diagram showing the relationship between the applications of the UOC track and the products of the IOI and TCC track is shown in Figure 2. The software libraries produced as part of the Telescope Control Console track and the Interactive Observing Infrastructure track are used to produce any UOC applications. The Console Track Library (CTL), part of the TCC track, ties together the CLL and the GUI toolkit to pro-

vide common console functionality at a level higher than the CLL. The CTL provides the following capabilities; other functions will be added during the detailed design of the CTL (See [6]).

- Console Interactive Graphical Control Semantics.
- Status/Health Interface.
- CLL and GUI Toolkit Integration
- Observing Tool Integration

The dataflow to and from the other principal systems through the CLL is also shown for completeness and is not discussed here. (See [2].)

## 7.2 User Interface Observing Tool Products

Instrument control must be integrated into the Observing Tool product. It is a design goal of the Observing Tool to provide a clean, standard, documented software interface to allow this kind of modification to the Observing Tool during its lifetime. How each instrument might be best integrated with the Observing Tool is not known at this time since it is specific to each instrument. The following methods will be available according to the SDD [1].

- It will be possible to drag and drop configurations to and from an application console.
- The Observing Tool may also include specific built-in screens for an instrument that are focused on common observing modes and tasks.

## 7.3 Infrastructure High Level Design

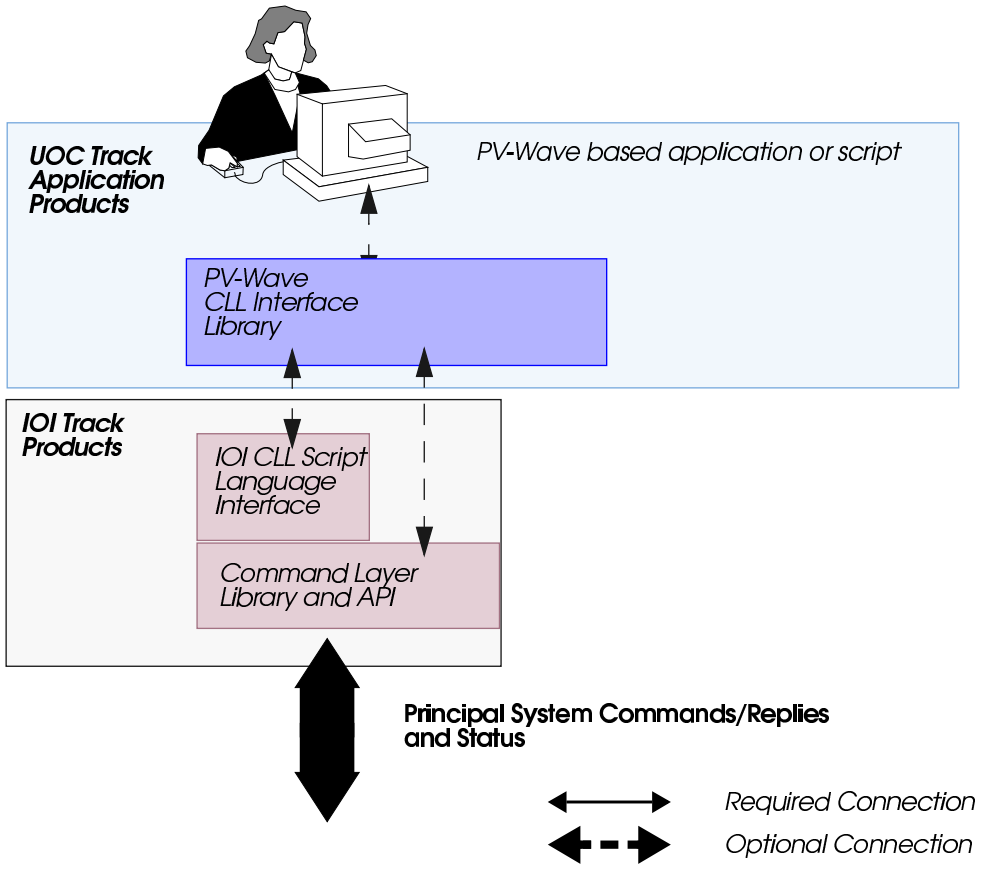
During this track the OCS infrastructure is supplemented with facilities that depend on the products of other Work Package groups.

### 7.3.1 PV-Wave Integration

The Gemini software design has specified the PV-Wave product as a tool that will be used in the observer's environment to execute high-level observing and engineering scripts. PV-Wave provides extensive libraries for image processing, plotting, and other data analysis tasks. The operations staff will use PV-Wave to write interactive scripts that require the data analysis features provided by PV-Wave. The OCS must provide a PV-Wave compatible interface into the Gemini software system to allow PV-Wave scripts to control the principal systems.

PV-Wave applications do not need to access the CTL or GUI libraries since PV-Wave provides its own interface to Motif. Figure 3 shows Figure 2 with only the IOI track CLL communication service connections. A PV-Wave language library must be written to "glue" PV-Wave to the CLL library functionality. This will allow PV-Wave scripts to send commands and receive the status information from the other principal systems. The functionality of the PV-Wave CLL interface library will match the functionality provided by the CLL itself.

*FIGURE 3.* The Software Environment of an PV-Wave Application in the UOC




### 7.3.2 Synchronous Data Reduction Functionality

The majority of data analysis functionality required by the scripting requirement is to be provided by the PV-Wave product. Previous reviews of the SDD suggested that in some cases, when processing is very CPU-intensive or the problem to be solved is very astronomy-specific, it would be better to be able to hand-off processing to a dedicated astronomical data analysis system such as IRAF. The Synchronous Data Reduction Functionality (SDRF) was added to the design to provide this fallback ability.

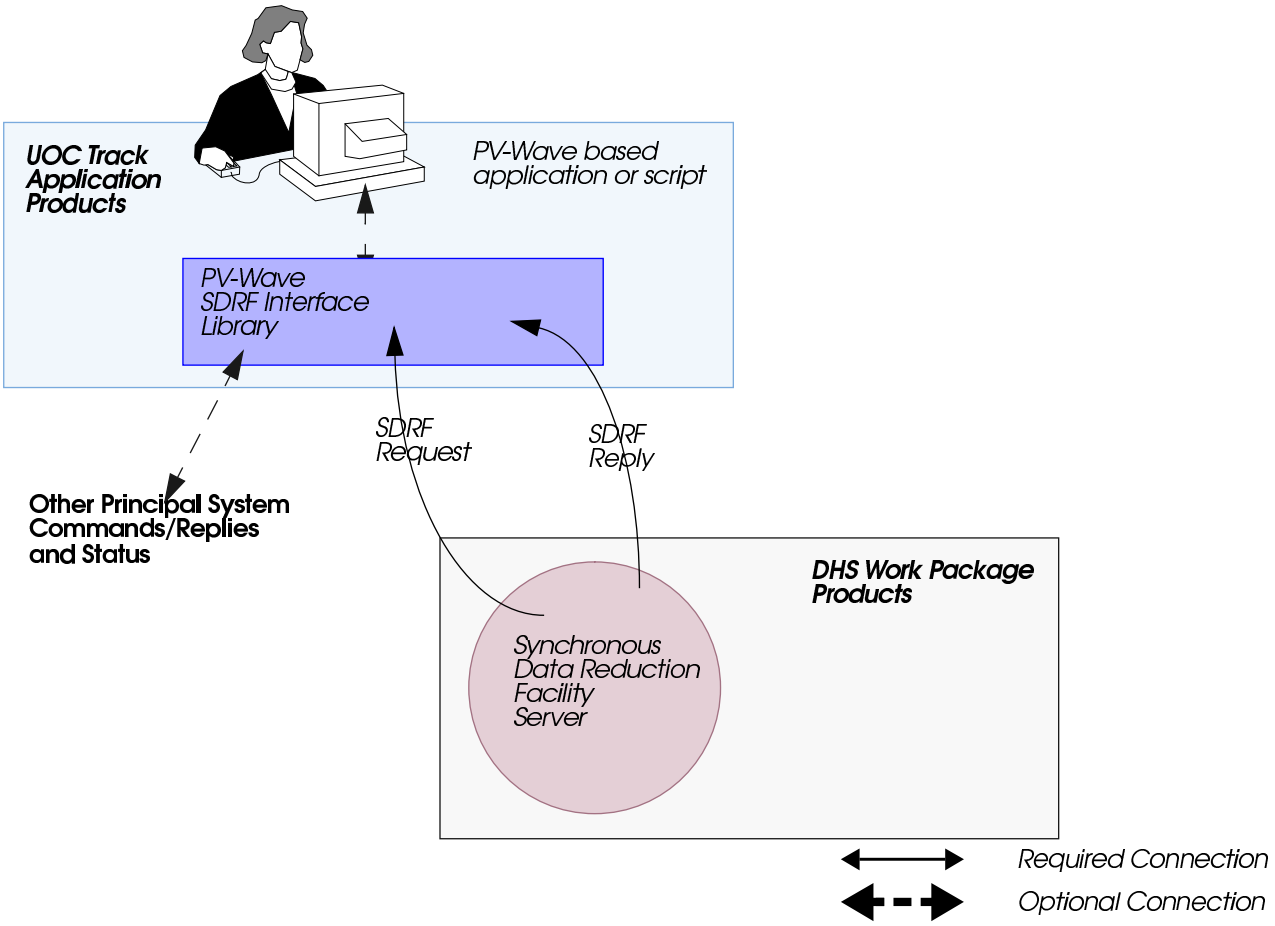
The following are important design points.

- Each synchronous request will return at most one result consisting of the simple data types or an array of simple data types. No images or image parts will be returned from or sent to the SDRF.
- The SDRF will be procedural based. A command and arguments will be sent to the SDRF. The command may be represented by attributes and values.
- An interaction between the SDRF and a client will cause the client to block until the request is completed (hence, the *synchronous* name).
- The SDRF functionality is only associated with the PV-Wave environment. It is not a part of the basic CLL functionality.

Based on these design points the SDRF is modeled as client-server system where PV-Wave script-based applications are the clients. The SDRF is a stand-alone server provided by the DHS. This is shown in Figure 4.

This track must provide the PV-Wave language interface that will allow the programmers of the PV-Wave scripts to synchronously send requests to the SDRF and receive replies.

*FIGURE 4.* The Software Environment of an PV-Wave Application in the UOC using the SDRF




At this time it is assumed that the OCS Message System (OCSService) will be used to allow a PV-Wave program to communicate with the SDRF. A SDRF Service can be included in the IOI track to allow communication with the SDRF Server in case the interprocess communication with the SDRF is not based upon the OCS Message System (OCSService). See [7]. This Service may be added to the IOI during the UOC track rather than the IOI track if the SDRF is not available during the IOI track.

# 8.0 Interfaces

## 8.1 Application Products

The UOC track applications can communicate with other OCS processes including:

- Observing Tool - to save and restore their configuration.
- Observing Database - to obtain shared system data.
- Any other OCS CLL-based application.

The UOC track applications must also communicate with the other principal systems.

- To send commands and receive synchronous and asynchronous responses.
- To receive status.

Inter-OCS communication uses the OCS Message System, and the software interface for these communications is provided by the Command Layer Library product. Both are part of the Interactive Observing Infrastructure track. The requirements for this CLL interface are in the IOI track document [2].

### 8.2 Infrastructure Products

The PV-Wave shell integration with the CLL only requires the programming interface provided by PV-Wave. This interface is documented in the PV-Wave manuals.

The SDRF integration uses the PV-Wave library documents and interface and documents provided by the DHS for the SDRF interface. At this time, it is assumed that the OCS Message System will be used to communicate with the SDRF server.

## 9.0 Development Plan

The User Observing Console track uses the functionality of the Interactive Observing Infrastructure track, the Observing Tool track, and the Telescope Control Console Track. In addition, the instrument consoles rely upon the development plans for the site instruments, and the PV-Wave SDRF relies upon work to be done by the Data Handling System Work Package group. Consequently this track can not be *completed* until all the depended upon products are completed. However, not all of the UOC products depend upon the completion of all the other products so it is possible to create beta and alpha releases of the UOC track.

The UOC track development plan will be phased to allow maximum track parallelism. The PV-Wave integration with the CLL can be completed first since it only depends upon the IOI and TCC track products which precede this track in the OCS development plan (See [4]). The PV-Wave CLL integration will be released in alpha and beta form before the UOC track is final.

A UOC observing instrument console and Observing Tool integration can be started once the PDF document for the instrument is available. Instrument consoles will include prototype phases and alpha and beta releases.

The creation of the SDRF integration library can take place once the use of the OCS Message System is agreed upon. There will be a beta and alpha release of the SDRF before producing a final product.

This development plan suggests one or more alpha UOC track releases and at least one beta UOC release.

## 10.0 Principal Parts of the UOC Track

The products of the UOC track are the instrument consoles with Observing Tool support for instruments, and the additions to PV-Wave. Each will be discussed only briefly here since the role of the PD is to identify track interdependencies within the OCS. The internals and software interfaces for PV-Wave access to CLL and the design and layout of the instrument consoles can not be delineated at this time.

### 10.1 Applications and Consoles

The Gemini Project has produced a list of site instruments. There is also a list of instruments that will be shared between Gemini and other observatories for Gemini north and south. The following lists show the

site instruments and shared instruments for both sites. The short programmer description of each instrument is included. The site lists contain the set of instruments that must be integrated with the Observing Tool and OCS during the UOC track. At this time it is not known how well integrated the shared instruments will be with the OCS.

*TABLE 1.* Cerro Pachon Site Instruments

| Screen Name | Description |
|---|---|
| High Resolution Optical Spectrograph | Single object optical wavelength instrument that can operate as imager or spectrograph. |
| Multi-Object Spectrograph | A mask-based instrument used to observe many objects at one telescope target position concurrently. |
| Mid-IR Imager | Single object mid-IR instrument. Can operate as an imager or spectrograph. |

*TABLE 2.* Manua Kea Site Instruments

| Screen Name | Description |
|---|---|
| Multi-Object Spectrograph | A mask-based instrument used to observe many objects at one telescope target position concurrently. |
| Near-IR Imager | Single object near-IR imaging instrument. |
| Near-IR Spectrograph | Single object near-IR spectrographic instrument. |
| Mid-IR Imager | Single object mid-IR instrument. Can operate as an imager or spectrograph. |

*TABLE 3.* Southern Shared Instruments

| Screen Name | Description |
|---|---|
| Phoenix | IR Instrument |

*TABLE 4.* Northern Shared Instruments

| Screen Name | Description |
|---|---|
| Michelle | Single object mid IR spectrograph. |

The issue of who develops OCS consoles often comes up. The following is the view of the OCS group with regard to site instrument consoles. First, a few relevant OCS requirements [3].

*SR1* **Need**:E **Source**:ASDD/URD **Priority**:1
**Short Description**: VUI Purpose

**Description.** The VUI of the OCS is required to provide the user interface and related concepts astronomers and operations staff will use during the operations/maintenance phase of the Gemini Telescope's life cycle defined in the ASDD.

*SR2* **Need**:E **Source**:URD18 **Priority**:2
**Short Description**: The VUI must provide an integrated set of observing tools.

**Description.** This requirement means that the software that is used when interactive observing and the software that is used for planned observing must behave similarly, provide the same look and feel, and they must function together to provide what seems to be one observing environment. This is often called application interoperability. There should not be completely different sets of tools to support the various observing modes required in this document.

These two requirements state that the OCS consoles must be observer/operator oriented and integrated with the other OCS tools. The OCS group could formally define what it means to be integrated and define how a console becomes part of the OCS and that may eventually happen, but this process would add significant work requiring the OCS group to provide code, documentation, and assistance to the widely dispersed Gemini developers. This is not possible under the current OCS development plan. The following statements summarize the approach concerning the implementation of instrument-related consoles at this time.

- Engineering consoles for instruments will not generally be duplicated by the OCS group. The operator will use engineering screens for engineering and observing screens for his normal system interactions.
- Operator/observer functionality that may be present in an engineering interface may be duplicated in an OCS console to improve the observer's efficiency or effectiveness.
- The OCS group will decide how to use the consoles provided by the instrument developers in conjunction with the instrument developers.
- The OCS group will work with the instrument development groups to help them provide commands and status that is oriented towards the observing process and the Gemini Control System software.

## 11.0 Data Acquisition Using Stand-alone Consoles

It has been decided that for site instruments it must be possible to acquire data using only an instruments stand-alone console. There are a number of trade-offs when using consoles to acquire data.

- The features of the Observing Tool are not available. There is no Science Program or Observations that exist past the observing session.
- The data and the data headers must contain all the observing information.
- The control and interactions with the other systems must be sequenced by the operators and observers.

The dataflow during stand-alone console observing is discussed in the Planned Observing Support track since this capability is built upon a product of that track (See [5]).

## 12.0 Usability Testing

The quality of a graphical user interface is generally better when the users of a product are involved in its development. The final users of the instrument products of the UOC track, the astronomers, should be involved in the development and testing of the UOC instrument consoles. The following is the approach that will be used to provide the highest quality UOC consoles possible.

- The design and screen layout of the instrument consoles will be guided by OCS programming team's experience with similar systems, the engineering consoles for the instruments, and the knowledge and documentation of the instrument developers. The OCS programmers will be responsible for producing a final, usable system.

- The design of the instrument consoles will fold in the useful innovations of the operator interfaces of telescope control systems the OCS programmers visit including Kitt Peak National Observatory; Keck Observatory; Canada, Hawaii, France Telescope, and the Very Large Telescope.
- Any suggestions from the project scientists and the results of any discussions on operations will be folded into the design of the instrument applications.
- A group of scientists interested in instrument console usability testing will be assembled for each instrument. Prototype screen designs and alpha and beta versions of instrument applications will be tested by the groups.
- All consoles will adhere to Gemini GUI style standards.

To ensure that operations staff programmers can make changes that will inevitably be required to the UOC consoles once the telescopes go on-line, the following is a UOC track implementation goal. The physical design of the OCS consoles supports this goal.

- The implementation of the UOC instrument applications will be done to make layout and cosmetic changes as simple as the chosen GUI toolkit allows.

# 13.0 Documentation

The documentation for the UOC track will follow the documentation requirements in the OCS SDR documents (SR66, SR67, SR68). The UOC provides all two kinds of OCS products: user applications (instrument consoles and OT screens), and testing software for the PV-Wave access library to the CLL and SDRF.

## 13.1 PV-Wave Integration Testing Documentation

This library will be released with the following documentation.

**The PV-Wave CLL Integration Technical Document.** This document describes how the PV-Wave language is interfaces to the CLL library.

A set of testing scripts will be provided in PV-Wave scripting language to test the PV-Wave interface.

**The PV-Wave Integration Testing Manual.** This manual will describe how to use the testing procedures required for the acceptance tests of the PV-Wave integration product.

## 13.2 Instrument Applications/Consoles

Each console or UOC application will be accompanied by user documentation that covers the stand-alone console and any instrument-related Observing Tool screens.

### 13.2.1 Instrument Console/Application Testing Documentation

At this time there is no Gemini software requirement to provide automated testing of GUI programs.

Created:December 16, 1994
Modified:March 18, 1996

S PDR

Gemini Observatory Control System Report

# Scheduling Track Preliminary Design

**Steve Wampler, Kim Gillies, Shane Walker**

ocs.ocs.003-schedulingTrackPD/02

**This report presents a preliminary design for the Scheduling Track.**

## 1.0 Introduction

The Scheduling Track is the last development track in the development plan for the Observatory Control System (OCS) of the Gemini Telescope. The following statements are from the Software Design Review OCS Development Plan [2].

> This track provides the infrastructure and tools required to do the various kinds of telescope time scheduling required in the SDD. (page 3)
>
> This track provides the tools operators and staff will use to pick observations and develop observing plans. This track must be completed to satisfy the planned observing requirements but is not needed to use the telescope during the initial system testing phases. (page 7)

The job of the Scheduling track is to produce the tools that will be used by observers and staff to prepare plans for effective use of the telescope.

This report presents the preliminary design of the Scheduling track to a depth such that the track can continue on in its development independently of the other OCS tracks. The following information is contained in this report.

- A high-level preliminary design for the track.
- The dataflow between applications in the Scheduling track and the other OCS tracks.
- Remaining decisions for the detailed design of the track.
- Usability testing plans.
- List of required documentation.

## 2.0 Acronyms

| | |
|---|---|
| GUI | Graphical User Interface |

| | |
|---|---|
| OCS | Observatory Control System |
| ODB | Observing Database |
| ODBA | Observing Database Agent |
| OT | Observing Tool |
| PT | Planning Tool |
| TAC | Time Allocation Committee |

## 3.0 References


[1] SPE-C-G0037, *Software Design Description*, 9/9/94, Gemini Controls Group, 1994.

[2] ocs.kkg.016, *Observatory Control System Development Plan*, Kim Gillies, Gemini Observatory Control System Group, 1995.

[3] ocs.kkg.032, *Interactive Observing Infrastructure Preliminary Design*, Kim Gillies, Shane Walker, Steve Wampler, Gemini Observatory Control System Group, 1995.

[4] ocs.kkg.038, *Planned Observing Support Preliminary Design*, Kim Gillies, Shane Walker, Steve Wampler, Gemini Observatory Control System Group, 1995.

[5] ocs._sw.004, *Observing Tool Track Preliminary Design*, Shane Walker, Kim Gillies, Steve Wampler, Gemini Observatory Control System Group, 1995.


## 4.0 Document Revision History

**First Release —** July 28, 1995. Pre-release draft.

**PDR Release —** August 16, 1995.

## 5.0 How the Scheduling Track Integrates with Overall Gemini Scheduling

The Gemini 8m Telescopes are scheduled from initial proposal to telescope time using three phases:

- Long term scheduling - performed across an entire semester. Here all submitted proposals are reviewed, ranked, and accepted or rejected as candidates for the observing season. Long term scheduling involves a number of groups including National TACs from the partner countries, a central Gemini TAC, the staff astronomers and the observatory directors.
- Medium term scheduling - performed across a short number of weeks and blocked out by sidereal time. Medium term scheduling is performed by the staff astronomers and the observatory directors.
- Short term scheduling - performed across an observing session. Short term scheduling determines the actual order of observations that are performed by the system and should be distinguished from the *planning* process. Planning involves querying the Observing Database to select a set of eligible observations. Many plans can be created ahead of time. Deciding which observations are next in the queue when actually executing a plan is the short term scheduling process.

Most of the process of performing long and medium term scheduling is outside the scope of this work package, although it must be supported by the Gemini project. Scheduling tools being developed by ESO and HST are under consideration to fulfill the long and medium term requirements.

The primary goal of the scheduling track is the direct support of short term planning and scheduling. A secondary goal is to make any decision support software (such as processes for checking guide-star suitability) available for possible use during long and medium term scheduling. This secondary goal must not interfere with the primary goal.

## 6.0 Studies/Decisions During Scheduling Track

There are no trade studies expected during the Scheduling track. A decision needs to be made as how to best support the long-term and medium-term scheduling requirements that are a expected to exist outside of the scope of the OCS work package.

## 7.0 High-Level Design of the Scheduling Track

The Scheduling track is built directly upon the functionality of the Interactive Observing Infrastructure (IOI) track [3], the Planned Observing Track [4], and the Observing Tool Track [5]. The primary product of the Scheduling Track is the Planning Tool (PT). It is considered to be an extension of the Observing Tool (OT), though it will be developed separately, after the remainder of the OT is complete.

The PT supports effective scheduling and rescheduling of short term plans. The PT is used to match program requirements against existing and expected conditions, and check target availability and guide star suitability in order to construct or modify a plan for scheduling the telescope. Once the plan has been created or updated with the PT, it can be executed and monitored with the OT.

The PT is essentially a database query interface that is used to facilitate the planning process. The set of Science Programs and Plans that are accessible depend upon the permissions of the user. When used by staff, all the programs in the Observing Pool are available. Non-staff observers only have access to their own programs. The PT is used to select among the available programs based upon various criteria, such as instrument and seeing conditions required. The output of the PT is a Science Plan, which can be edited or executed using the remainder of the OT.

*FIGURE 1.* The Software Environment of the Scheduling Track

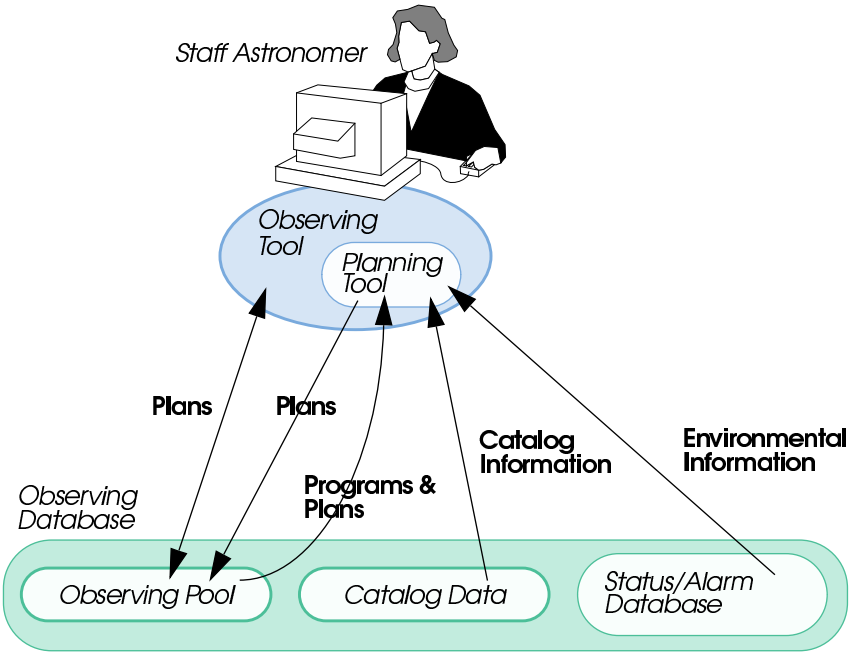



Figure 1 diagrams the data that is passed between the Planning Tool and the Observing Database (ODB). Instead of showing implementation details, we concentrate on depicting the information that must ultimately pass between the OT and the ODB.

The Planning Tool uses the Status/Alarm Database (SAD) for queries involving the suitability of observations in the Pool to the current conditions. Catalog information includes guide star parameters and science target information.

## 8.0 Interfaces

The Planning Tool interfaces directly with the Observing Database, using the OCS Message System. The ODB presents the OT (and hence PT) with an interface to the ODB that provides access to an observer's science programs and other information, but hides most of its contents [4]. The PT does not interact with any other applications.

## 9.0 Development Plan

The Planning Tool development is based on a phased-released model. An alpha release of the tool only provides limited functionality but allows the construction of Plans from Science Programs. A beta release provides limited support for checking target/guide star parameters and adds support for criteria-based selection of observation. The final release refines the components developed for the beta release.

The Scheduling track relies upon the functionality of the Interactive Observing Infrastructure track, Planned Observing track and Observing Tool track and must follow those tracks in the development process. The OCS development plan is discussed elsewhere [2].

## 10.0 Usability Testing

The Scheduling track uses the same GUI interface testing procedures as the Observing Tool track.

## 11.0 Documentation

The documentation for the Scheduling track follows the documentation requirements in the OCS SDR documents (*SR66*, *SR67*, *SR68*). The Scheduling track extends the Observing Tool and so must extend its documentation. The documentation will include:

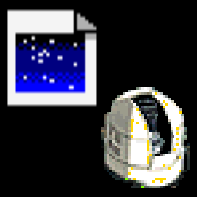
- User's guide
- Installation and maintenance guide
- On-line help

## 12.0 Deliverables

The Scheduling Track deliverables are the Planning Tool extension to the Observing Tool, and the documentation discussed in the previous section.

Created:June 19, 1995
Modified:March 18, 1996

Gemini Observatory Control System Report



# Sequence Command Specifications

**Kim Gillies, Shane Walker, Steven Beard**

ocs.kkg.031-SeqComms/04

**This report updates the OCS Sequence Commands defined in the SDD based on the most recent conversations between OCS and CICS. A prototype implementation is also discussed.**

## 1.0 Introduction

To implement planned observing, the OCS provides a way to save the system configuration required to produce a desired observational setup. The configuration is then played back at a later time to setup the principal systems and obtain the science data.

The OCS Configurable Control System uses a process called the *sequence executor* to provide the playback functionality. Each sequence executor uses a text script called a recipe to describe the high-level structure and sequencing of the principal systems. The recipe sends system-independent *Sequence Commands (SC)* to the principal systems. A sequence command consists of an *opcode* and an optional argument. A configuration can be viewed in our system as a set of system dependent commands in the form of attributes and values. The recipe operates upon and processes configurations. The recipe can also send system dependent commands to implement changes to the principal systems during an observation. It is our goal that recipes (besides the Observe script) be written entirely in terms of Sequence Commands and system independent sequence command status. The definitions in this paper allow that to be true.

The sequence commands were first defined in the SDD [3]. This paper updates the definitions of the sequencer command opcodes, discussing what principal systems must do when they receive each one. In addition, mandatory status variables are defined that describe the state of an ongoing observation. An implementation described in terms of CAD and CAR records is also included along with standard names for the sequence command CAD records, CAR records, and status variables.

This paper assumes knowledge of CAD/CAR and SIR records [2]. The use of these records was introduced after the SDD and they are very important in the implementation of sequence commands.

## 2.0 Acronyms

ARD Action Response Database

| | |
|---|---|
| CAD | Command Action Directive |
| CAR | Command Action Response |
| CCS | Configurable Control System |
| IOI | Interactive Observing Infrastructure Track |
| OCS | Observatory Control System |
| ODB | Observing Database |
| PS | Principal System |
| PSA | Principal System Agent |
| SIR | Status Information Record |
| SAD | Status Alarm Database |
| SC | Sequence Commands |
| SDD | Software Design Description |

## 3.0 References


[1] gscg.grp.020.icdbook, Interface Control Document 1a, 1b, 2. Gemini Interface Control Documents, Gemini Controls Group

[2] gscg.grp.024/02, ICD1b - The Baseline Attribute/Value Interface, Gemini Controls Group

[3] SPE-C-G0037, *Software Design Description*, 9/9/94, Gemini Controls Group.

[4] gscg.kkg.005, Baseline Major Systems Interface, Kim Gillies, Gemini Controls Group

[5] ocs._sw.002, Interactive Observing Infrastructure Preliminary Design, Gemini Observatory Control System Group, 1995.

[6] ocs.kkg.028, Action Variable Protocol Preliminary Design, Gemini Observatory Control System Group, 1995.


## 4.0 Glossary

**Configuration Part —** The part of a configuration, the set of attributes and values, that is associated with a single principal system function. A configuration part is equivalent to a command if it can be mapped to a CAD record in the PSA (for EPICS-based principal systems). The term command and configuration part are often used interchangeably but not all configuration parts are commands.

**Entire Configuration —** All the configuration parts make an entire configuration.

**Principal System Configuration —** The subset of a configuration that is destined for a single principal system. Or all the configuration parts that go to one principal system.

## 5.0 Sequence Command Design

The sequence command design has been modified from the original SDD version to reflect a change in where the principal system command interface resides. This change is outlined in general below. The **config(apply)** sequence command is unique in many respects and is discussed separately.

## 5.1 Changes to the Sequence Command Design

In the original SDD the activities of the Principal System Agent (PSA) were implemented in command servers that were present in each of the principal systems. Only sequence commands passed between principal systems. A principal system (PS) was to receive a configuration, apply it, and tell the OCS when the application of the configuration was completed. The system specific attribute/value layer interface and functionality was entirely within a PS.

The principal system command interface is now at the attribute/value layer and in all known cases this interface is implemented upon EPICS Channel Access and CAD records. The principal systems are required to make action information available to the OCS and other principal systems using EPICS CAR records. Status information is published using SIR records. The CAD/CAR/SIR records make up an EPICS-based principal system's entire public interface.

The IOI design dictates that no knowledge of the configuration abstraction be known in the attribute/value layer below the configuration layer. Only the **config(apply)** sequence command has a configuration as an argument so this command must be specially treated in the OCS. Principal system configurations are broken into configuration parts, which if mapped to a CAD record in a principal system's PSA, are applied at the attribute/value layer interface and become system dependent commands in the EPICS system (see Section 5.2).

A principal system still needs to know the opcode of a sequence command it is being requested to perform so it is necessary that the opcodes make their way into the PS. Therefore, the opcodes must be viewed in the attribute/value layer as commands that all systems must implement in a uniform way.

As system-dependent commands, the OCS must know when a principal system has completed a particular sequence command. The mechanism provided in the Gemini system for this functionality is the CAR record and it should be used to notify the OCS and others of the completion of sequence commands.

An implementation of sequence commands using CAD and CAR records is assumed in the following discussion. A prototype implementation will be presented in a later section of this paper.

## 5.2 The Apply Sequence Command

Almost all of the sequence commands are very high-level and global, meaning that the execution of the command influences the operation of the entire principal system. A PS can execute at most one instance of a global command at a time and it is difficult to imagine scenarios where two sequence commands would be executing simultaneously in the same principal system. The one problematic exception is **config(apply)**.

A principal system configuration consists of one or more configuration parts (see the definitions in Section 4.0 on page 2). Principal system configurations are applied by the OCS Principal System Agent process in two steps:

1. Each configuration part is set and validated independently. This is called the *preset* step.
2. Next, the **config(apply)** sequence command is issued. This command causes the target system to match the configuration specified by the preset step.

This two phase application enables the principal system to use writing of the **config(apply)** CAD to perform any sequencing it might need. Otherwise, principal systems would have to be designed so that they could accept any ordering of system-dependent commands.

The OCS can monitor completion through the **config(apply)** action variable or the individual part action variables. It clearly must be possible for systems to accept and apply more than one configuration at a time and this is what makes **config(apply)** different from the other sequence commands.

## 6.0 Sequence Command Opcode Definitions

The sequence commands from the SDD are shown in Table 1 on page 4 with the Version 3 definitions of their opcodes. Most of the command definitions are the same as the SDD definitions. Late in the SDD document preparation Steven Beard noticed that the definitions of the **config(observe)** and **config(endobserve)** sequence commands did not always allow optimum performance from an instrument. It is sometimes important to allow observing to continue when a detector is reading out, and it is vitally important that the next **config(observe)** not begin until the last image is out of the detectors. For this reason, the observing sequence has been modified significantly in this version and is discussed in Section 7.0 on page 7.

The definitions of the sequence commands are not attempting to enforce a state machine design on principal systems. The principal systems should assume that any sequence command can arrive at any time. Any restrictions a principal system wishes to place on the ordering of sequence commands should be noted in the systems Parameter Definition Format document so it is visible to all.

*Note:* The **config(check)** and **config(endobserve)** sequence commands have been removed from the list. They no longer have any purpose in our system.

*TABLE 1.* Sequence Command definitions (revision 3.0)

| Event (Configuration Command) | Command Arguments | Action/ Definition |
|---|---|---|
| **config(test)** | | A system should assume it has just been switched on and perform self-tests for its software and hardware systems to check that it is healthy. A **config(test)** could be the first step in a **config(init)** following a reboot but it is not required to be the first step. Following a **config(test)** a system should be ready to accept commands. **config(test)** should not require it be followed by **config(init)** or **config(reset)**.<br>This command completes successfully when it passes its tests or it fails during execution and enters the ERROR state in the *test* CAR record. |
| **config(init)** | | The system should execute its most complete initialization sequence. This can include rebooting and reloading any internal setup files (a *hard* init).<br>The command completes successfully when the system it determines it is initialized or it fails during execution and enters the ERROR state in the *init* CAR record.<br>If a system reboots as part of its **config(init)** it must continue the **config(init)** action following the reboot. This means that the *init* CAR record must be set to BUSY and then IDLE following a reboot. See notes following this table. |
| **config(reset)** | | A system should do whatever is needed to reset its internal system state to the state it had at start-up and become ready for new commands (a *soft* init). It should NOT reboot or re-read any setup files.<br>This initialization command is less severe than **config(init)** and **config(reset)** should be the final phase of **config(init)**.<br>The command completes successfully when the system determines it has completed reset or it fails during the process and enters the ERROR state in the *reset* CAR record. |

| Event (Configuration Command) | Command Arguments | Action/ Definition |
|---|---|---|
| **config(park)** | | A system should adopt an internal configuration in which it can be safely switched off. This will occur at the end of an observing session or when a principal system will enter a time of extended disuse.<br>The command completes successfully when the system is ready to be powered down. This command fails if a problem is encountered while preparing to park. It then uses ERROR in the *park* CAR record to return the fault. |
| **config(apply)** | | The apply command represents the second part of the two phase command application sequence. A series of system dependent CAD records are first preset independently by the OCS PSA. This essentially builds up a new desired configuration which must be matched by the principal system. The the apply command is issued to cause the system to take any actions required to match the configuration built up by the series of presets.<br>Principal systems should cause the *apply* CAR record to become BUSY whenever any of the actions associated with the preset commands are busy. The apply command completes successfully when all the actions have completed. |
| **config(verify)** | | This command indicates to a principal system that verification of configurations is underway by the OCS, operators, and observers. A principal system must be capable of executing changes to its state during a verify, and it must also update its status and state in the ODB SAD and ARD.<br>Interactive commands must *always* be accepted by a system.<br>This command only provides information for principal systems and requires no special action although a system may wish to have special actions for **config(verify)**.<br>A principal system should successfully complete immediately after noting the **config(verify)** command. Therefore the *verify* CAR record should transition briefly to BUSY and then to IDLE. |
| **config(endverify)** | | This command indicates to a system that verification of configurations is finished.<br>This command can be executed at any time.<br>This command only provides information for principal systems and requires no special actions although a system may wish to have special actions for **config(endverify).**<br>A principal system should successfully complete immediately after noting the **config(endverify)** command. Therefore the *endverify* CAR record should transition briefly to BUSY and then to IDLE. |
| **config(guide)** | | This command indicates to the principal systems that it should do whatever is needed to *start* the guiding operation.<br>The command completes successfully when the **config(guide)** actions have *begun* successfully. The *guide* CAR record should transition briefly to BUSY and then to IDLE once the guiding operations have begun properly and it is safe for the sequence executor to go on.<br>Systems that choose to ignore the **config(guide)** command should go BUSY briefly followed by the transition to IDLE. |

| Event (Configuration Command) | Command Arguments | Action/ Definition |
|---|---|---|
| **config(endguide)** | | This command indicates to all principal systems that they should stop their guiding actions. A principal system should execute any particular behaviour that should occur when the telescope stops guiding.<br>The *endguide* CAR record should transition briefly to BUSY and then to IDLE. Systems that choose to ignore the **config(endguide)** command should go BUSY briefly followed by the transition to IDLE. |
| **config(observe)** | ObservationID | This command indicates that data acquisition should begin in an instrument system based on its current internal values.<br>Instruments executing a **config(observe)** remain busy until they have completed the configured observation. The sequence executor uses completion of **config(observe)** to determine when an observation completes.<br>The OCS uses the *observe* CAR to determine when the integration is complete and the data is out of the instrument and in the DHS. The *observe* CAR must remain BUSY for this entire period. In addition, instruments are required to update three status values (SIR records) corresponding to the current phase of the observation. These are PREP (preparing to acquire), ACQ (acquiring), and RDOUT (reading out the detector and transferring data). These are detailed further in Section 7.0 on page 7.<br>Systems other than instruments can view **config(observe)** as informational. For those systems the *observe* CAR record should transition briefly to BUSY and then to IDLE.<br>The OBSERVE command argument is an identifier that should be used by the instrument when it sends its data to the DHS. The DHS uses the ObservationID to create the data files in a predictable way (to be determined during detailed design). |
| **config(pause)** | | This command indicates that a system should do whatever is appropriate for it to pause data acquisition. Pause indicates to the principal system that the user intends on continuing at a later time.<br>The command completes successfully when the **config(pause)** actions have begun successfully.<br>Systems that choose to ignore the **config(pause)** command should immediately complete successfully by setting the *pause* CAR record briefly to BUSY and then to IDLE |
| **config(continue)** | | This command is the reverse of pause. A system should do whatever is appropriate for it to resume data acquisition.<br>The command completes successfully when the **config(continue)** actions have taken place successfully.<br>Systems that choose to ignore the **config(continue)** command should immediately complete successfully by setting the *continue* CAR record briefly to BUSY and then to IDLE |

| Event (Configuration Command) | Command Arguments | Action/ Definition |
|---|---|---|
| **config(stop)** | | This command indicates that a system should stop the current data acquisition process normally, as if it were the end of the data acquisition period.<br>The command completes successfully when the **config(stop)** actions have taken place successfully. The OCS would then notice the *observe* CAR record go to IDLE and would send the config(endobserve).<br>Systems that choose to ignore the **config(stop)** command should immediately complete successfully by setting the *stop* CAR record briefly to BUSY and then to IDLE |
| **config(abort)** | | This command indicates that a system should stop the current data acquisition process immediately and discard any data.<br>The command completes successfully when the **config(abort)** actions have taken place successfully. The OCS would then notice the *abort* CAR briefly go through BUSY and the *observe* CAR record go to IDLE. The recipe would then do whatever abort recovery is required.<br>Systems that choose to ignore the **config(abort)** command should acknowledge the command by setting their *abort* CAR record briefly to BUSY and then to IDLE. |

## 7.0 Observing Sequence

For maximum efficiency at the telescope it must be possible to configure the principal systems efficiently. Here are some common observing situations that came up during SDD that the OCS must be able to sequence:

- An observation consists of a series of science frames. Between frames a filter change is applied. The application of the filter change is faster than the read-out of the detector so the system is ready to start the next **config(observe)** before the last read-out is complete.
- Sometimes observers will want to move to the next observation as quickly as possible, without waiting to perform detailed quality control on previous observations. In this case, the OCS can begin to configure the next observation while waiting for the detector to complete its read-out.
- Some observers do care about quality control, and for some specific instruments an apply might interfere with the detector read-out.

The common requirement in these examples is that the OCS must have more information about what is going on in the instrument in order to sequence the observations. The observe CAR alone does not offer enough granularity to handle the first two cases. It must remain BUSY the entire time that the instrument is setting up, exposing, reading out, and transferring data to the DHS.

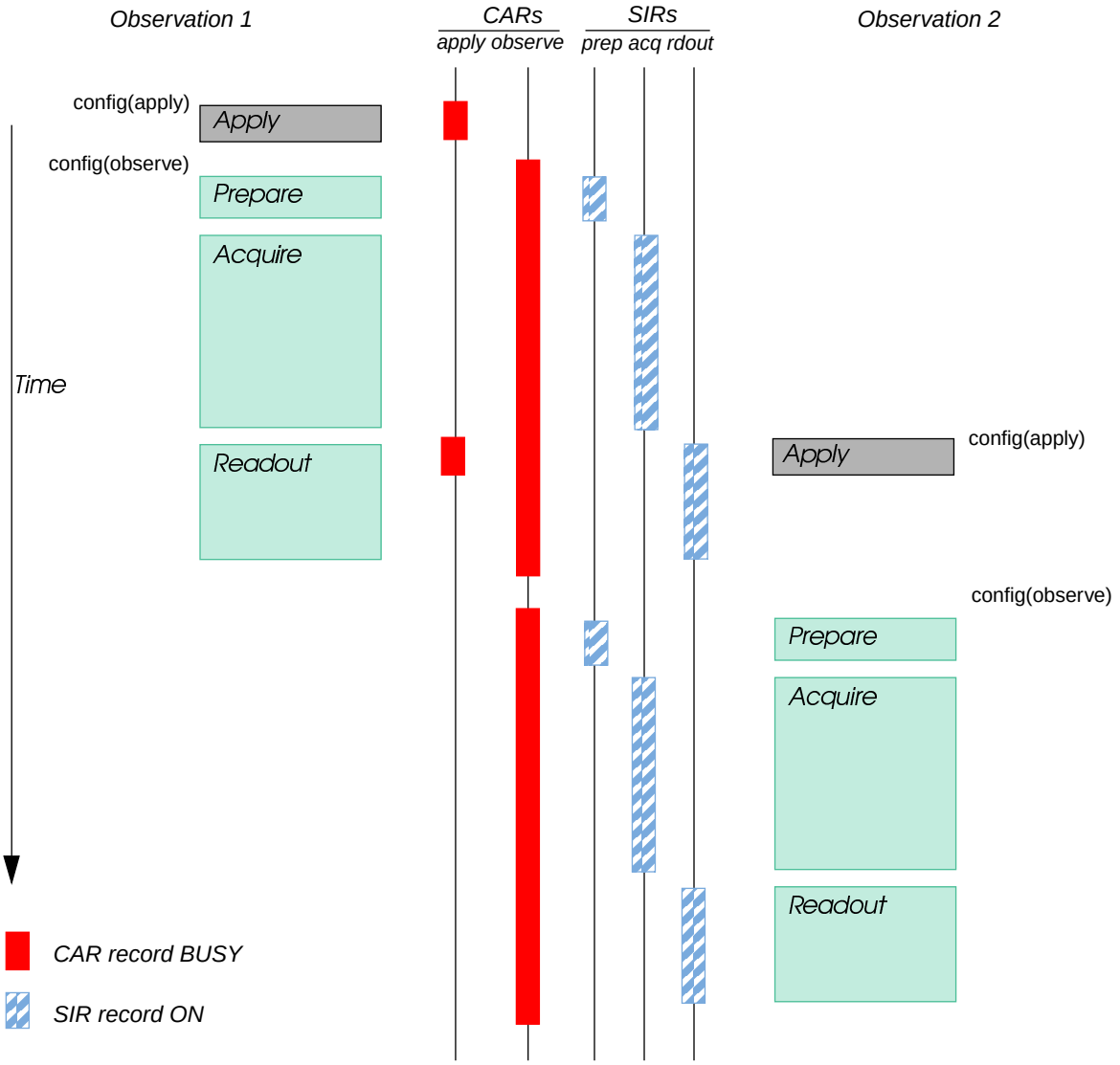
To solve this problem, all instruments, both optical and IR, must maintain three status variables corresponding to their current activities:

- **PREP.** This status variable is ON while the detector is preparing to acquire a science frame. This is the period between the start of an observation and data acquisition, and could involve an initial reset or read of the detector.
- **ACQ.** The ACQ variable is ON while the detector is acquiring science data. For an optical detector, this is the time when the shutter is open. For IR instruments, it is the entire period while exposures are being made.

• **RDOUT.** While the detector is reading out and data is being transferred to the DHS, this variable is ON.

The OCS can use changes to these status variables to obtain finer grained control. For instance, when the ACQ flag transitions from ON to OFF, it may possible to apply a new configuration to the instrument, even if RDOUT is still ON. This case is illustrated in Figure 1 below.

*FIGURE 1.* ICS actions when observations follow one another quickly and apply overlaps readout.




In the figure the CAR records are assumed to be IDLE when not BUSY, and the SIR records are OFF when not ON. Note that the observe CAR stays on for the duration of each observation. Without the status records, the recipe controlling the observation could not safely issue the second **config(apply)** until the observe CAR went IDLE.

Of course, this method simply makes overlapping the apply with the readout *possible*. The observer will specify restrictions on when this may occur in the observation configuration using the Observing Tool. For some instruments, it may never be a good idea to allow any activity during the readout. In this case the recipe would delay the next **config(apply)** until the *endobserve* CAR is IDLE.

The following sections cover a few remaining details and notes.

### 7.1 All Instruments Must Adhere to the Definitions

Much of the discussion so far has tacitly assumed an optical instrument is in use. However, for IR instruments, the definitions of PREP, ACQ, and RDOUT should remain exactly the same. ACQ should transition to ON when acquiring science data, and RDOUT should transition to ON when reading out the detector. This implies that both ACQ and RDOUT will be ON simultaneously for IR instruments, but this will not cause a problem for the OCS.

Provided every instrument follows the definitions of **config(observe)**, PREP, ACQ, and RDOUT, the OCS will be able to sequence any instrument efficiently. Namely,

- **PREP, ACQ, and RDOUT should each transition from OFF to ON and back to OFF again *once* during an observation at the appropriate times.**

Note that this rule applies even when the instrument is paused. If the OCS cannot rely on a set pattern of transitions for PREP, ACQ, and RDOUT, then sequencing becomes much more difficult if possible at all.

### 7.2 Obtaining Header Information

With the introduction of the ACQ status variable, the OCS now has a much more accurate picture of when header information should be obtained. When ACQ goes from OFF to ON at the beginning of the observation, the OCS can snapshot header data as close to the actual time that shutter opens as possible. Likewise for the ON to OFF transition at the end of the observation.

## 8.0 A Sequence Command Implementation

This paper has suggested that the approach to implementing sequence commands in a Gemini EPICS-based principal system is to use CAD and CAR records. The approach suggested here is to use a single CAD record and a single CAR record for every sequence command. Since all systems must implement the sequencer commands it is reasonable to specify a standard name for a CAD record for each sequence command opcode as well as a CAR record name for each opcode.

To save characters, the command CAD record names are limited to the first 6 characters of the opcode. The CAR record name is found by adding C to the CAD record name.

*TABLE 2.* Standard Sequence Command CAD/CAR Record Names

| Sequence Command | Opcode | CAD Record Name | CAR Record Name |
|---|---|---|---|
| **config(test)** | test | Test | TestC |
| **config(init)** | init | Init | InitC |
| **config(reset)** | reset | Reset | ResetC |
| **config(park)** | park | Park | ParkC |
| **config(apply)** | apply | Apply | ApplyC |
| **config(verify)** | verify | Verify | VerifyC |
| **config(endverify)** | endverify | EndVer | EndVerC |

| Sequence Command | Opcode | CAD Record Name | CAR Record Name |
|---|---|---|---|
| **config(guide)** | guide | Guide | GuideC |
| **config(endguide)** | endguide | EndGui | EndGuiC |
| **config(observe)** | observe | Observ | ObservC |
| **config(pause)** | pause | Pause | PauseC |
| **config(continue)** | continue | Contin | ContinC |
| **config(stop)** | stop | Stop | StopC |
| **config(abort)** | abort | Abort | AbortC |

## 8.1 Sequence Command Opcode CAD Record Arguments

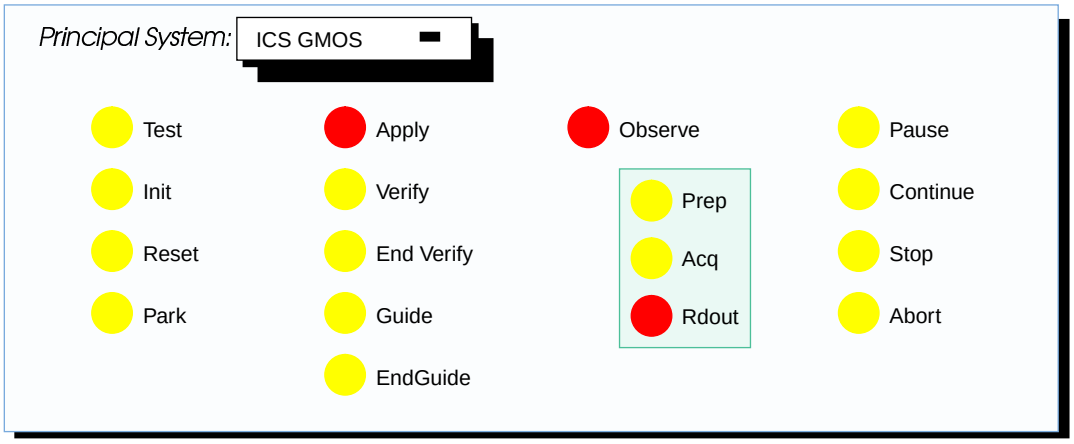
Note that CAD records come equipped with an opcode argument. Since there is a separate CAD record for each sequence command, the opcode argument is meaningless for sequence commands.

## 8.2 Sequence Command Monitor Prototype

A sequence command monitor is prototyped in Figure 2. Since this principal system is an instrument, the monitor also includes the three observation status SIR records. The information from the CAR action records will allow the operator to quickly view what is happening in the principal systems allowing him to monitor the progress of recipes. A more realistic console might display the CAR records for all the operating principal systems at one time.

A similar console could be constructed to allow testing of sequence commands and interactive observing using the sequence opcode commands. In fact, this could be one method for observing during the commissioning operational phase.

*FIGURE 2.* Example of a Sequence Command Monitor Console




# 9.0 Sequence Command Notes

Some final notes on implementing sequence commands and future work.

**Guiding Command.** The **config(guide)** command may not be adequate for the recipes that will run on the telescopes. It may be necessary to introduce additional opcodes or modify the definition of the **config(guide)**

sequence command once the interactive operations that take place to setup an observation are a little better known. The OCS staff is currently meeting with project scientists and others regularly to discuss probable telescope operational procedures. Any changes to the sequencer commands will be under the control of the Gemini Controls Group processes and reviewed. No changes are known or expected at this time.

**Ignored Commands.** All systems must implement all the sequence command opcodes. The operation of the recipe in the sequence executor relies on responses from all principal systems. A principal system that doesn't care about a command should indicate completion immediately by toggling the appropriate CAR record to BUSY and then IDLE.

**Already There.** When a system is requested to perform an action and it finds that it doesn't need to do it because the action is not required, it must still indicate completion immediately by toggling the appropriate CAR record to BUSY and then IDLE. For instance, if a filter wheel is requested to move to position 4 and the controlling system notes that it is already at position four, the controlling system must still toggle the CAR record from BUSY to IDLE. This is true whenever any CAD/CAR command is applied.

**Errors in Sequence Commands.** The CAR protocol defines that the only use of the ERROR state is to notify operators that an error occurred during the execution of an action. All other errors should cause the rejection of the sequence command opcode in its CAD record.

**Acceptance/Rejection of Sequence Commands.** As with all CAD records a principal system has the capability of accepting and rejecting the sequence commands when they arrive at the principal system. Rejection should occur if a system is unable to perform the requested sequence command opcode. For instance, if one sent a **config(observe)** to an instrument when it was reading out a detector (executing **config(endobserve)**), the instrument could reject the **config(observe)**.