Herschel Common Science System

Overall Architecture and Design

Document ID: Herschel-HSC-DOC-0436
Issue: 1.2, 4 December 2007
Author: Jon Brumfitt
Tilmann Zäschke

Contents

---

1. System Overview

1.1 Scope of System

The Herschel Common Science System (HCSS) provides the scientific part of the Herschel ground segment software. This software is used by the Herschel Science Centre (HSC) and the three Instrument Control Centres (ICCs), to perform the following functions:

- Information provision
- Proposal submission and handling
- Scheduling of observations
- Observation product generation & quality control
- Storage, access & retrieval of data, products & software
- Support for calibration and cross-calibration

An overview of the mission and the way the system is operated is given in the Operations Scenario Document [12]. A detailed description of the overall ground segment, including the HCSS, is given in the Herschel Ground Segment Design Description [3].

This architecture document provides a top-level description of the HCSS and its decomposition into sub-systems. The architecture and design of the subsystems are described in separate documents.

The HCSS is designed to support all phases of the mission, from initial Instrument Level Tests (ILT), through Integrated System Tests (IST) and in-orbit operations, to post-mission archiving.

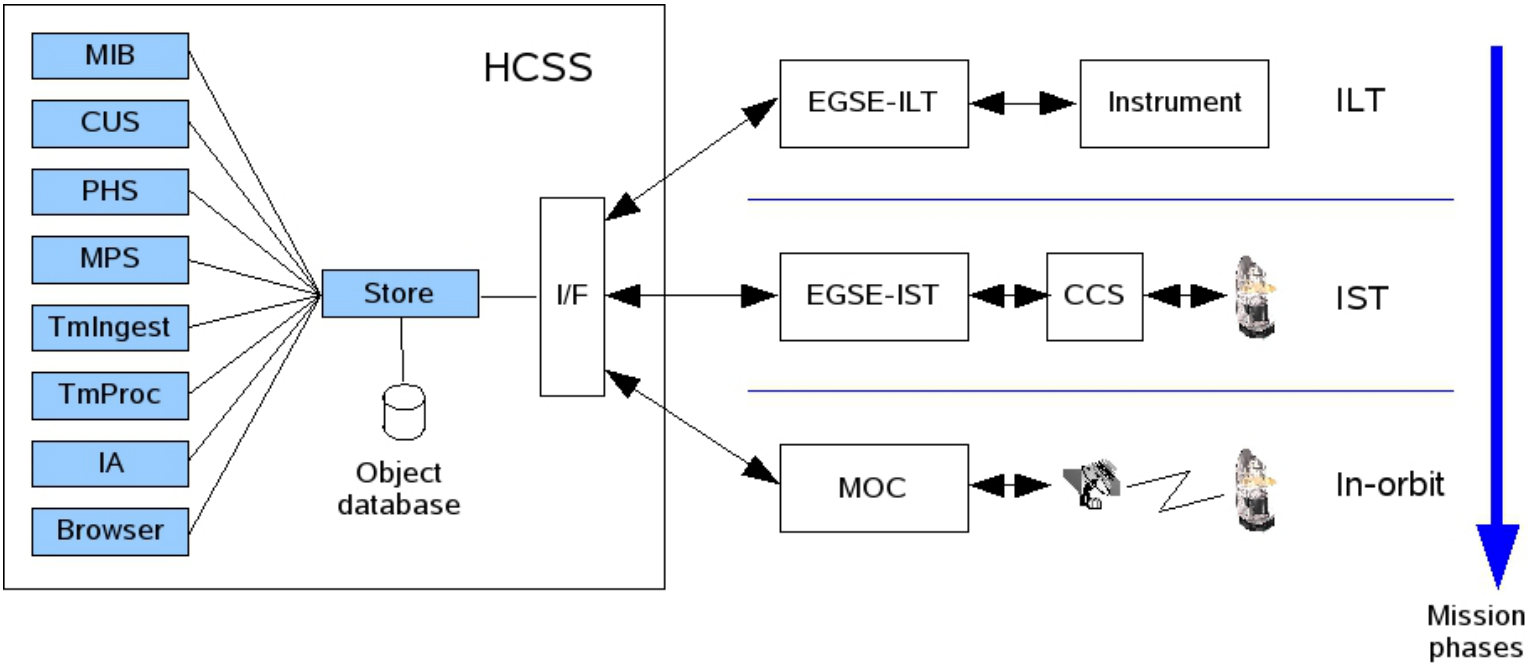
1.2 Smooth Transition between Phases

The concept of a single evolving system, that supports successive mission phases, is referred to as "smoooth transition". It ensures that:

- software developed for a given phase can be largely reused for subsequent phases
- data collected in a given phase can still be accessed and processed in later phases

This approach has the considerable advantage that much of the HCSS software is routinely used well before launch. This allows many problems to be found and solved early on in the development, which might otherwise not be detected until near or after launch. The disadvantage is that the data in the database has to undergo *evolution* in line with any schema changes that occur during development.

The following diagram shows the smooth transition between phases:

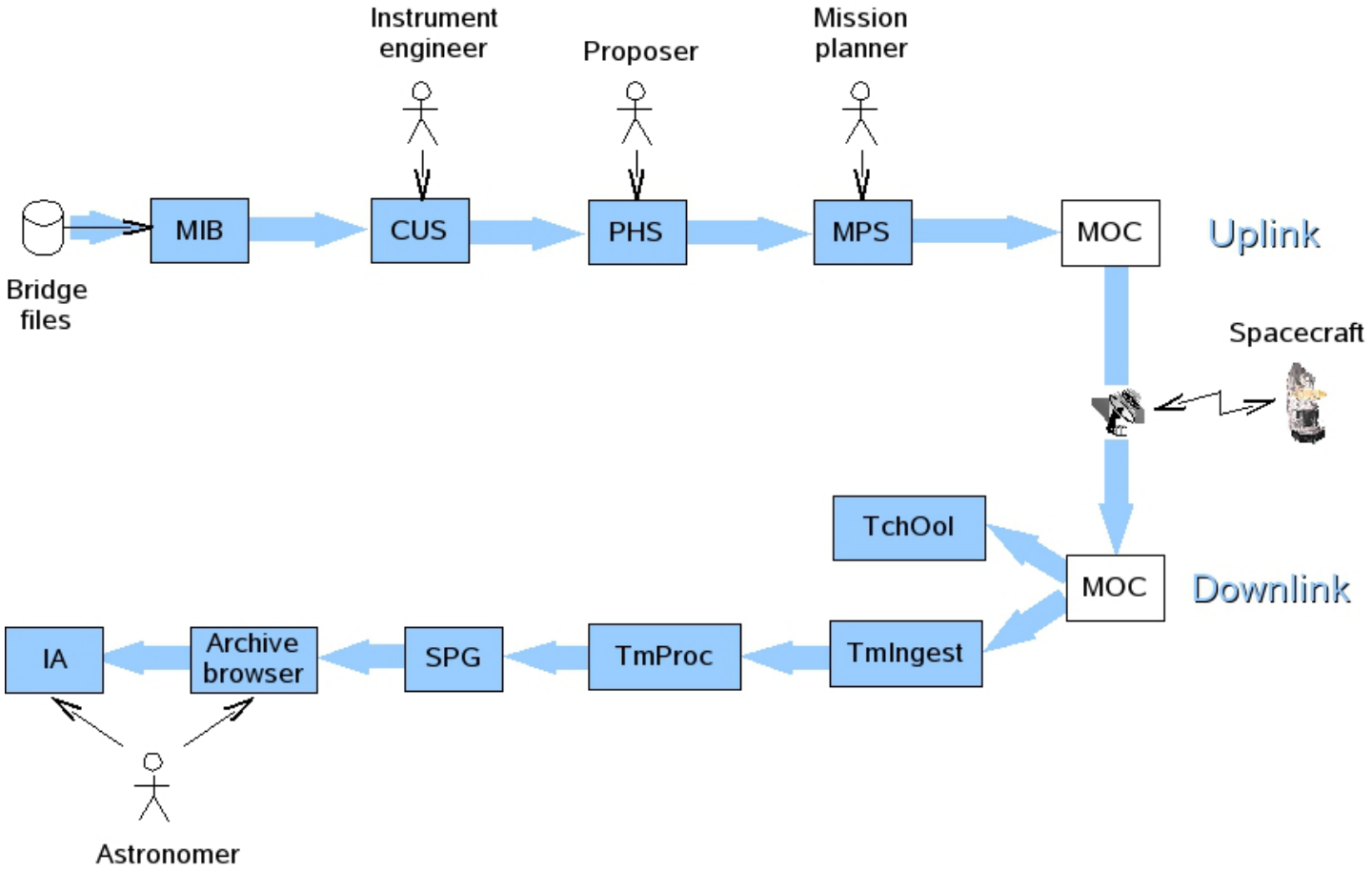


- The HCSS is first used in the Instrument Level Test (ILT) phase, to test and characterise the instruments. At this stage, there is no spacecraft and the instruments are interfaced to the HCSS via the Electrical Ground Support Equipment (EGSE). The Proposal Handling System (PHS) and Mission Planning System (MPS) are not needed for this phase.
- Later, the instruments are integrated with the spacecraft for the Integrated System Tests (IST). For this phase, the EGSE is interfaced to the spacecraft via the Central Checkout System (CCS). The PHS and MPS are again not needed for this phase.
- For in-orbit operation, the HCSS communicates with the spacecraft via the Mission Operations Centre (MOC) and the ground station. The PHS and MPS are used to submit observation proposals and schedule the observations for onboard execution.
- Finally, the database and archive browser remain in operation after the in-orbit phase has finished, to provide continued availability of data to the scientic community. This phase is not shown in the diagram.

The overall operation of the HCSS, in the context of both ILT and normal operations is described by the "FIRST Ground Segment Design Description" [3]. The operation for ILT and IST is described by the relevant use cases [4, 7] and the technical note "Analysis of ILT Use-Cases" [5].

1.3 Uplink and Downlink Overview

The HCSS architecture consists of an *uplink* chain, that is used to command the spacecraft and instruments, and a *downlink* chain, that is used to process the data received from the spacecraft. The following diagram shows the overall flow for the in-orbit phase:



The uplink starts with the ingestion of the instrument *bridge files* by the MIB subsystem. These files, which are received from the CCS during IST and MOC during operations, contain the definitions of instrument commands and telemetry. The instrument engineer uses the Common Uplink System (CUS) to define instrument *observing modes* which make use of these instrument commands. The proposer (i.e. an astronomer) submits a proposal containing a number of observation requests. Each request specifies the observing mode to be used, together with its parameters. The mission planner uses the Mission Planning System (MPS) to generate schedules (one per operational day) containing a sequence of requested observations. The schedule is expanded into a sequence of instrument telecommands (using the CUS engine) and sent to the Mission Operations Centre (MOC) for further processing and uplink to the spacecraft.

Approximately 24 hours later, when the spacecraft is next in contact with the ground station, the data from the observations is downlinked as a sequence of telemetry packets. These are ingested into the database by the 'tmingest' subsystem and the dataframes are extracted by 'tmproc'. MOC also provides telecommand history and Out-of-Limits (OOL) data, which are ingested by the TchOol subsystem. The dataframes are processed into first-level products by the Standard Product Generation (SPG) software, which uses much of the Interactive Analysis (IA) software in batch mode. The resulting data products are made available by the Herschel Science Archive, from which they can be downloaded by astronomers. The data can be downloaded via the HSA Browser, the VO-Interface, the Scriptable Interface or via the PAL (Product Access Layer, part of IA). The astronomer can download the IA software and use it also to perform further specialised processing of the data.

The diagram above does not show the handling of calibration observations. These are processed by the Calibration Engineer, using the IA system. The calibrations are made available via the same interfaces as data product (see above). They are also used to create calibration tables for use by the CUS, providing a feedback link from downlink to uplink.

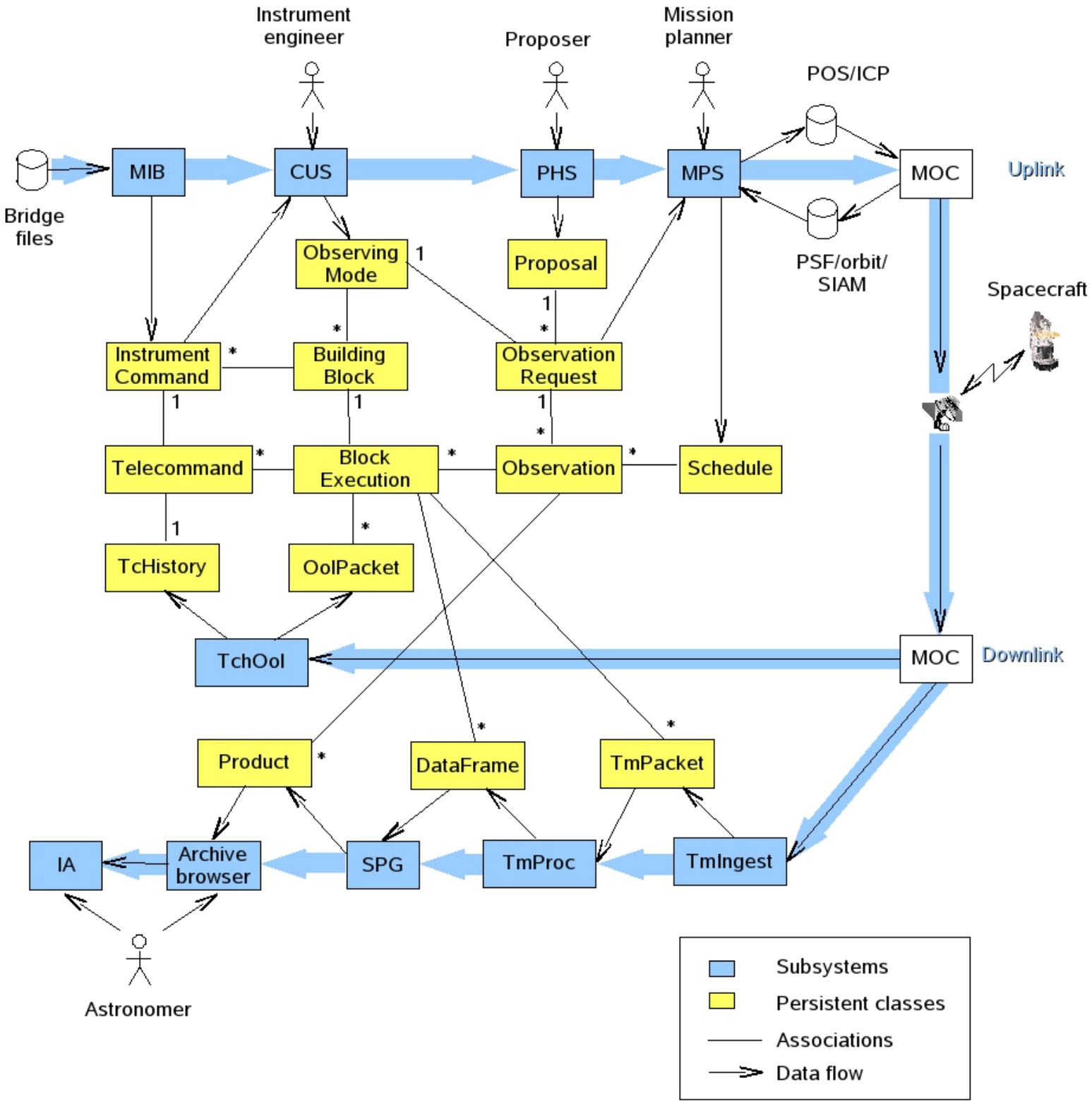
1.4 Relation to CCM

A major part of the HCSS architecture is the Core Class Model (CCM). This provides a set of persistence-capable classes that allow objects to be stored in the database. The persistent classes include the key entities from the domain model, such as *Proposal* and *Observation*.

The Core Class Model is described in the following documents:

- Core Class Model Developer's Guide (includes UML class diagram)
- Core Class Model Architecture and Design

The following diagram expands the uplink-downlink diagram, given above, to show how it is realised using the core classes:



The logical flow of information follows the blue line, although in fact the subsystems communicate information via objects in the database, which are instances of the persistent classes shown in yellow.

The HCSS is a distributed Object-Oriented system, centred around an object database. Each of the major subsystems, such as proposal handling and mission planning are clients of this database. The database provides the primary means by which the subsystems communicate with one another. For example, proposal handling creates persistent ObservationRequest objects in the database; the mission planning system uses these as input, to generate observation schedules.

It can be seen from the above diagram that database associations are created between downlink entities and the corresponding uplink entities. For example, Products are associated with the appropriate Observation, DataFrames are associated with the BlockExceutions commands which

give rise to them and TcHistory is associated with the relevant Telecommand. Thus, in terms of a data model, the uplink and downlink chanins are tightly integrated. As a result, it is possible to *navigate*, by following associations from one object to another, between uplink commands and their resulting data.

2. Database Architecture

2.1 Overall Database Architecture

The key part of the overall architecture is formed by the *Core Class Model* (CCM), that defines the persistent classes. This is supported by a Persistent Storage Manager (store), that provides an abstraction of the underlying persistence mechanism (i.e. object database). Together, the CCM and store are roughly equivalent to the 'FINDAS' concept, that is used in older FIRST documentation.

In the final operational system, client applications will be deployed at multiple sites, including the Herschel Science Centre (HSC) and the three ICCs. There are also three-tier applications, such as proposal submission and IA, which may be used remotely (see below).

The Versant Object Database Management System (ODBMS) allows a distributed database, in which the data is split across multiple servers at multiple sites. Each persistent object has an object identifier (OID) that uniquely identifies it across all databases in the whole project. Within a database system (a configured set of databases), programs can transparently follow associations from one object to another, even if they are located in a different database of that system.

In order to avoid problems with locking and security, and to improve performance, it is desirable to *propagate* data to other sites (e.g. ICC institutes), so that a local copy is available. The implementation of the HCSS `store` package is automatically notified when a database object is modified, triggering the transfer. Further details are given in the Herschel Database Administration Manual [10].

The main Herschel database system at HSC is expected to reach 15 - 20 TB by the end of the mission. This large database system consistes of a number of smaller database nodes, which simplifies amongst others database maintenance and backup. The organisation of these database nodes and systems is described in the techinal note on Herschel Database System Architecture [8].

Database performance tests are being carried out to ensure that the database design will *scale* to the final size, without an unacceptable loss of performance. This work is described in the Herschel technical note on Database Performance Testing [9].

2.2 Core Classes

The central part of the whole HCSS architecture is formed by the object database. There are two parts to this:

- The Core Class Model (CCM) defines persistence-capable classes for the problem domain.
- The Persistent Storage Manager (store) makes objects persistent and manages access to them.

Unlike normal *transient* objects, that are lost when a program terminates, *persistent* objects remain in the database so they are available to successive executions of a program. They may also be shared by multiple concurrent processes (with locking provided by the database). This allows one program to create an object and another program to use it. For example the Proposal Handling system creates `ObservationRequest` objects and the Mission Planning system schedules them. This provides a means for sub-systems to work together, without having any direct communications (e.g. API calls) between them.

The Core Class Model, which forms the basis for the whole architecture is described in the following documents:

- Core Class Model Developer's Guide (includes UML class diagram)
- Core Class Model Architecture and Design

2.3 Database Abstraction

Object database technology is still evolving rapidly, but the HCSS is required to operate until at least the year 2020. It is therefore important to decouple the HCSS client applications from the underlying database, so that changes to the database API have the minimum effect on the client applications. Ideally, all such changes should be localised in an abstraction layer, which hides vendor-specific aspects of the database from the clients.

This abstraction is provided by putting the API and implementations in separate packages:

- `herschel.ccm` - CCM API
- `herschel.versant.ccm` - Versant implementation of CCM
- `herschel.vanilla.ccm` - Versant-free subset of classes of the CCM
- `herschel.store` - Store API

`herschel.versant.store` - Versant implementation of store

The API packages provide an implementation-independent API definition, comprising Java interfaces, factory classes and exception classes. The implementation packages are *pluggable* components, for a particular persistence mechanism (currently a Versant object database).

This pluggable component-based architecture is described in more detail in the following documents:

- Core Class Model Architecture and Design
- Persistent Storage Manager Architecture and Design

2.4 Multi-Tier Applications

Three-tier architectures will be used for the following HCSS applications:

- Proposal submission
- Access to data by observers

These applications have the following properties:

- Security is important as they involve remote access (by astronomers).
- There can be a large number of simultaneous users (especially for proposal submission).
- End-users (astronomers) should not have to purchase a database licence to use the system.

Many other HCSS applications, such as Mission Planning, are well-suited to a simple two-tier architecture:

- They are run internally to the HSC and/or ICCs, so that security can be handled by normal system accounts.
- There is only one (or a very small number) of instances running.
- The processing, which is complex and interactive, is best handled on the same machine as the user interface.
- Performance is significantly better as the database client cache is in the application's address space.
- Development effort is reduced (no RMI proxy classes, etc).

This does not preclude the possibility of middleware, in the form of an abstraction layer (e.g. Facade), to decouple clients from the underlying object model. This does not require an additional process, as is normally implied by a three-tier architecture.

It would be possible to evolve applications, such as the CUS and mission planning, into multi-tier designs, at a later date, if necessary.

3. Interfaces

3.1 External Interfaces

External interfaces, with other systems are covered by separate Interface Control Documents (ICDs). An overview of these is given in the Herschel Ground Segment Interface Requirements Document [6].

These interfaces are implemented by the appropriate HCSS subsystems. For example:

```
mib          - Import of external MIB database from files
tchool       - Ingestion of telecommand history and out-of-limits data
tmingest     - Ingestion of telemetry
mps          - Mission planning interfaces to MOC (PSF, orbit file, SIAM, POS, etc)
phs          - Interface for proposal submission
access       - Read-only access to object streams (e.g. telemetry) via HTTP  for ILT/IST
testcontrol  - Interface to EGSE for ILT/IST
```

3.2 Internal Interfaces

Internal interfaces between HCSS subsystems are defined by JavaDoc API specifications, as discussed below.

4. System Design

4.1 Design and Development Method

The software is written in the Java programming language and follows an Object Oriented approach. The development process is described in

the HCSS 'Roadmap' document [2], and is based on Fowler's book "UML Distilled" [1]. Requirements are specified by a set of *use-cases* and supplementary specifications. Models are described using the UML notation. The HCSS Java Coding Standard and Guidelines [11] provides a set of recommended design practices as well as lower-level coding rules.

The overall architectural design breaks the system down into a number of *components*. Each component has its own version number and is the responsibility of one developer (see section 4.4 for the list of deliverable packages). The components are delivered regularly and a continuous integration is maintained by running a nightly build, which includes test harnesses for all the components. In the Construction Phase, development is driven by a succession of regular incremental steps, each of which implements a specific set of use-cases. This incremental, iterative development approach leads to a succession of regular developer releases, with occasional system releases for users.

To enable semi-independent development of components, by developers working at different sites, each deliverable component has a well-defined public API, which is specified in JavaDoc. Where possible, the API should hide the underlying implementation, so that the developer has the freedom to change and optimise the implementation without affecting clients. It should also make public only those features which are intended for use by other components and which the developer will maintain. In several places, *pluggable components* are used, with Java interfaces and factories, to provide a high degree of abstraction and decoupling.

4.2 Subsystem Decomposition

The HCSS system is hiearchically decomposed into a number of major subsystems. This document describes the overall architecture and decomposition into subsystems. The major subsystems have their own architecture and design documentation, which provides the next level of detail. [*In the online version of this document, these may be found by following the hyperlinks below.*]

The major HCSS subsystems are as follows:

- `ccm` - The core class model
- `store` - Persistent storage manager
- `cus` - The common uplink system
- `egserouter` - Router for telemetry packets (ILT and IST only)
- `ia` - Interactive analysis system
- `mib` - Ingestion, validation of and access to instrument command definitions
- `phs` - Proposal submission and handling subsystem
- `tmproc` - Telemetry processor
- `vanilla` - Implementations independent of persistence mechanism (documented in interface packages)
- `versant` - Implementations specific to Versant database (documented in interface packages)
- `release` - Top-level documentation, release notes and configuration files

There are also several packages containing software libraries that are used by these subsystems:

- `share` - General-purpose infrastructure libraries
- `access` - Middle-tier for remote access to telemetry packets and data-frames
- `binstruct` - Framework for handling binary data

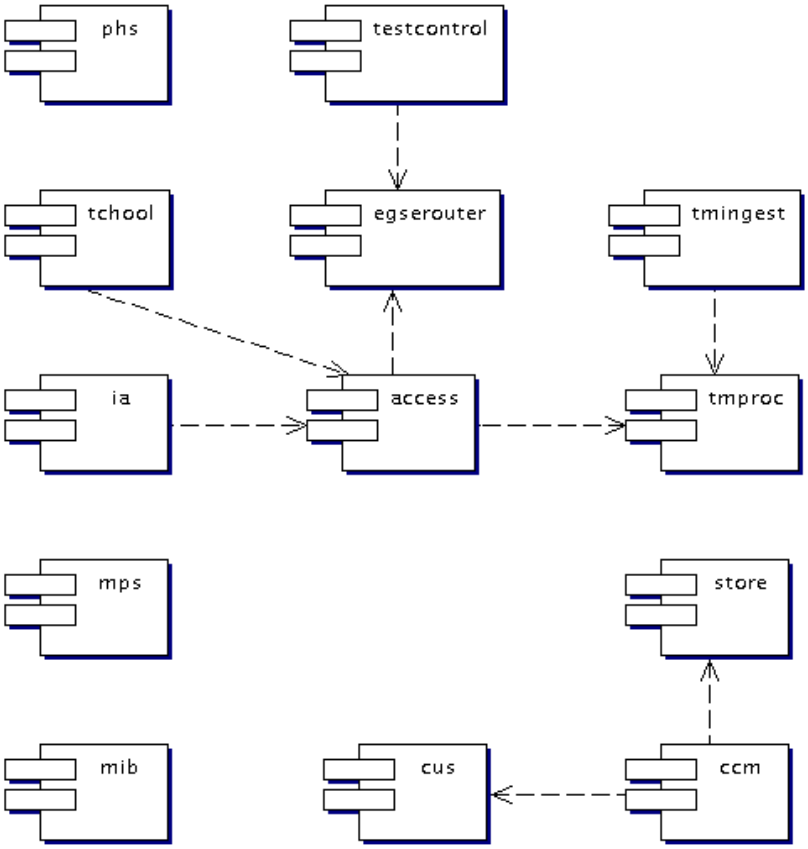
Finally, there are a number of packages containing development and configuration tools:

- `devel` - Development tools for building / testing HCSS
- `tools` - Tools for configuring HCSS environment

The above subsystems, libraries and tools are located in Java sub-packages of the top-level `herschel` package. For example, the ccm is in the package `herschel.ccm`.

4.3 Subsystem Dependencies

The following UML component diagram shows the dependencies between the main HCSS subsystems:



Notes:

- Dependencies on 'ccm' and 'store' and 'share' are not shown to avoid cluttering the diagram, since all subsystems make use of them.
- The 'testcontrol' and 'egserouter' packages are specific to ILT/IST. The 'access' package provides a temporary means for 'ia' to retrieve data, until the archive browser is available.
- Each of these components corresponds to a Java package with the prefix `'herschel'`.

4.4 Subsystem Interfaces

The JavaDoc API specifications define the API interfaces for each of these subsystems. These are (ideally) independent of the actual implementation.

Note: Java's package scoping is such that classes in sub-packages must be *public*, in order for them to be visible to other sub-packages of the same subsystem. However, such classes are not always intended to be part of the public API of the subsystem as a whole. A number of HCSS subsystems, such as the CCM, place their public API in a sub-package called 'api' to make this clear. In other cases, classes are commented to indicate that they are not part of the public API.

Design details that are specific to the implementation are covered by separate "Architecture and Design" guides, where appropriate. Subsystems that have complex APIs, may also provide a "Developer's Guide" which serves as a tutorial for programmers using the API.

The majority of interactions between subsystems take place via the CCM (i.e. using persistent objects). There are a few additional direct API interfaces, between subsystems, as shown in the component diagram shown earlier.

4.5 Package Structure

All Java packages that form part of the HCSS system have the package prefix '`herschel`'. For example:

```
herschel.ccm
herschel.store
```

All Java source files are organised in a directory tree that is isomorphic with the package tree. For example, the class `herschel.mps.gui.Mps` is in the directory `herschel/mps/gui`.

The system is split into a number of top-level packages, which form the unit for delivery. Each such package has its own version number and has an owner who is reponsible for it. This helps to ensure that the package maintains a coherent concept, which might not be the case if several developers tried to modify the same package. Each developer is responsible for the evolution of his packages. This includes keeping them in a consistent state, with up-to-date test harnesses and documentation, and making periodic releases of the packages.

These deliverable packages are currently as follows:

```
herschel.access
herschel.binstruct
herschel.ccm
herschel.cus
herschel.devel
herschel.egserouter
herschel.ia.cal
herschel.ia.classloader
herschel.ia.dataset
herschel.ia.dataflow
herschel.ia.demo
herschel.ia.doc
herschel.ia.help
herschel.ia.image
herschel.ia.io
herschel.ia.jconsole
herschel.ia.numeric
herschel.ia.plot
herschel.ia.task
herschel.ia.testbed
herschel.ia.ui
herschel.mib
herschel.mps
herschel.release
herschel.share
herschel.store
herschel.tchool
herschel.test
herschel.testcontrol
herschel.tmingest
herschel.tmproc
herschel.tools.propertygenerator
herschel.tools.sandbox
herschel.vanilla
herschel.versant.ccm
herschel.versant.store
herschel.versant.tools.browser
herschel.versant.tools.schema
```

These deliverable packages are typically split into sub-packages (e.g. '`gui`'), to decompose the subsystem into smaller components.

5. References

1. Fowler M, *UML Distilled,* Addison Wesley, 1997.
2. *HCSS Development Roadmap*, Herschel-HSC-DOC-0408, issue 1.0.
3. *FIRST Ground Segment Design Description*, FIRST/FSC/DOC/0146, issue 1.4.
4. *HCSS Use-Cases for ILT*, FSCDT/TN-023, issue 2.8.
5. *Analysis of ILT Use-Cases*, Technical note, FSCDT/TN-012, issue 0.5.
6. *Herschel Ground Segment Interface Requirements Document*, FIRST/HSC/DOC/0117, issue 2.3.
7. *HCSS Use-Cases for IST*, FSCDT/TN-035, issue 1.1.
8. *Database System Architecture*, HSCDT-TN042, issue 0.1.
9. *Database Performance Testing*, HSCDT-TN045, issue 0.4.
10. *Database Administration Manual* , HSCDT-TN044, issue 0.19.
11. *Java Coding Standard and Guidelines for the HCSS*, HSCDT-TN009, issue 2.0.
12. *Herschel Space Observatory Operations Scenario Document*, Herschel/HSC/DOC/0114, issue 1.2.