Open Microscopy Environment

<SW Unit Name>
Software Design Document

<*Date*>

<Author>

<Lab> — <Division/Department>
<Institute/University>

<author email>

DISCLAIMER OF WARRANTY

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the

```
Free Software Foundation, Inc.,
59 Temple Place, Suite 330,
Boston, MA  02111-1307  USA
```

Contents

1. Introduction.

This document details the design of the <SW unit name> within the OME Reference Implementation. Relative test cases are described in a separate document, <reference to test document>.

Before diving into detailed design, let's briefly outline the <SW unit name> required functionality and features. After that, we'll also give an outline of the solution, which is fully described in the next sections.

1.1. Requirements.

<
Briefly outline the requirements that have been allocated to the software unit under design and reference the requirements document. Make clear the objectives to be achieved.

Pay attention to non-functional requirements such as performance, scalability, reliability, security, adaptability and extendibility, and so on. Those requirements usually have a deep impact on the flavor and shape of the architecture. Refer to the SAD template for a discussion of quality attributes and scenarios.

Sometimes it might be worth supplying concrete usage scenarios and examples. However, those issues should have already been unfolded into the requirements document. Rather point the reader there and try not to inflate this document.
>

1.2. Solution outline.

<
Sum up the solution model. Focus on the logical solution. Mention where appropriate other views such as process and deployment view — refer to the SAD template for a discussion of views. Also outline how the solution model addresses the given requirements and mention where appropriate the rationale for the design choices. Be concise.
>

1.3. Document overview.

<
Sum up the documents contents. State noteworthy assumptions made about

`the readership and give reading guidelines where needed.`
>

The following sections in this document will deal with:

- *Object Model*: The core of this document, depicting both the static and dynamic model of the software in terms of objects.
- *Process Model*: In this section, we examine synchronization issues, describe IPC and see how the object model can fit into different concurrency models.
- *Mapping to Code*: How the object model relates to concrete Perl <`or other target language`> classes and namespaces.
- *Deployment*: Configuration, dependencies, distribution and hardware topology.
- *Failure Model*: How failures are handled and recovered.
- *Wrapping up*: We put all the pieces together into a big picture, we explain how to use and configure the <`SW unit name`> from an outsider's point of view and make some final considerations.

UML [OMG01] diagrams are extensively used throughout this document to precisely depict design. Even though all presented diagrams are commented out and many of them are quite self-explanatory, in order to understand in full the semantics of the diagrams a certain familiarity with UML is necessary. Those that are unfamiliar with UML may want to keep a reference at hand, such as [BRJ00].

2. Object Model.

This section describes both the static and dynamic model of the software in terms of objects. We first introduce the overall logical architecture of the solution model and we show how the solution addresses the requirements. We then dive deeper into detailed object design.

2.1. Overall architecture.

<
A few introductory lines to highlight the key points. Very concise and to the point.
>

Follows a summarized description of the logical structure and behavior of the object model. Focus is on the key elements and on how they relate and cooperate to fulfill the requirements. Notice that what follows is not a detailed description of all elements, relationships and behaviors. This is a bird-eye description that elides many details for the sake of presenting the key ideas to the reader. Detailed static and dynamic models are discussed later.

2.1.1. Structure.

<
Overall organization of the key component elements. Description of the key elements, their responsibilities and their relationships. Focus on macro-organization.

Group responsibilities into cohesive sets and assign each set to a software component. Make use of design patterns that deal with macro-elements (architectural patterns, such as Layers, Pipes and Filters, Model-View-Controller, etcetera).

Assigning responsibilities to software components is a crucial step in achieving good design. Some basic useful principles:

- *Low Coupling*: assign responsibilities so that unnecessary coupling among elements remains low.
- *Indirection*: when direct coupling is undesirable, assign the responsibility to an intermediate component to mediate between other components or services, so that they're not directly coupled.
- *High Cohesion*: assign responsibilities so that cohesion remains high. Every element should be assigned a set of highly related responsibilities.
- *Protected Variations*: identify points of variation or instability and assign responsibilities to create a stable interface around them.

- *Information Expert*: assign a responsibility to the component that has the information necessary to fulfill the responsibility.

>

Fig 2-1: Overall static model.

<
Provide at least a UML class diagram that presents the key classes, their roles and relationships. Don't clutter diagrams with too many details and don't try to include all possible classes (those needed to support key elements, remember this is a bird-eye view); compare the static model in Fig 2-1 in the Log Service SDD with the detailed state models in sections 2.2.1 and 2.3.1 for a further clue.
>

Let's now take a closer look at the key elements.

<
Briefly detail each key class, focusing on the class responsibilities and explaining roles and relationships with other classes.
>

2.1.2. Dynamics.

<
Describe the key interaction scenarios that characterize the behavior of the key elements and highlight how those elements collaborate to fulfill the requirements. Focus on macro-behavior.
>

The overall behavior of the <SW unit name> during a typical interaction scenario can be characterized by the following phases:

<
The goal here is to show how the key elements in the static model collaborate to fulfill the tasks they were assigned to. Identify and briefly describe the key phases in a typical interaction scenario. Focus on macro-behavior.
>

The following UML sequence diagram further details a typical interaction:

Fig 2-2: Overall dynamic model.

<
Provide at least a UML sequence diagram that depicts a typical interaction scenario. Don't clutter diagrams with too many details and don't try to include all possible messages (those needed to support key messages that are exchanged during the collaboration, remember this is a bird-eye view); compare the dynamic model in Fig 2-2 in the Log Service SDD with the detailed dynamic models in sections 2.2.2 and 2.3.2 for a further clue.
>

A final consideration on design patterns.

<
Relate the used design patterns to the actual design.
>

2.1.3. Addressing the requirements.

The reader should have, by now, a grasp of the key ideas within the solution model. Thus, it's a good time to point out how the solution model addresses the <SW unit name> requirements outlined in section 1.1.

<
Show how the solution model works the given requirements out.
>

2.1.4. Rationale.

<
Explain the design choices and any trade-off.

Often we'll come up with more than one solution and we'll have to decide which one best meets our goals. To make things more difficult, many times those goals are contrasting. It turns out that we'll have to understand the consequences of our decisions with respect to our goals, weight and trade-off different solutions. We'll eventually have to come up with a solution that strikes a good balance among contrasting goals and satisfies the given requirements.

This section is the right place to explain all that.>

2.2. Detailed design.

Follows a detailed description of the components of the <SW unit name>. Classes and relationships are discussed in the static model. The dynamic model addresses the collective behavior of those elements.

2.2.1. Static model.

We describe here the static structure, encompassing classes and relationships. Responsibilities, roles and collaborators of each element are detailed.

<SW unit name> classes serve the following purposes:

<
Break down and group classes according to the purpose they serve. State those purposes. Apply design patterns.
>

The following UML class diagram presents the structure, which is then further detailed, specifying the responsibilities, roles and collaborators of each element.

Fig 2-3: Static structure.

<
Provide at least a UML class diagram that presents the classes, their roles and relationships. Don't blow diagrams up with too many classes and don't try to include all possible classes into just one diagram. Rather provide a class diagram for each of the purposes identified above and only show the organization of the classes serving that specific purpose. See section 2.3.1 of the Log Service SDD for an example.
>

The classes in the above diagram are detailed below.

<
Describe classes as follows (similar to CRC style):

Class

Description.

Responsibilities

- Responsibility.

Collaborators
- Collaborator.

Fields
- *Field name* — Description. Mention type if appropriate.

Methods
- *Method signature* — Describe what the method does and the meaning of its parameters (mention type where appropriate). Detail return value and any thrown exception.

>

2.2.2. Dynamic model.

We now focus on the collective behavior of those elements described in the static model. The <SW unit name> classes have to collaborate in order to:

<
Combine single class behavior into larger tasks. State those tasks. Apply design patterns.
>

The first of the above tasks is illustrated by the following interaction scenario:

Fig 2-4: Dynamics.

<
Provide UML collaboration and sequence diagrams as appropriate for each interaction scenario.
>

A final note. Some trivial issues (such as routine initialization code, constructors and destructors, accessors and mutators or, in general, features that are required to write working source code, but that can be trivially derived from this object model) haven‘t been addressed by this object model explicitly. Those are routine programming tasks that would add little to the semantics of this model — but would inflate this document quite a bit, and for this reason are omitted.

3. Process Model.

In this section, we examine synchronization issues, describe IPC and see how the object model can fit into different concurrency models.

3.1. Flows of control and synchronization.

We have already mentioned some concurrency issues in the object model. Now it' time to completely unfold that matter. Before talking about processes and threads, it's worth studying concurrency from a logical and general point of view. Specific process and thread semantics introduce many details that would add nothing to the concurrency issues that we're going to study, but would simply obfuscate the matter. We discuss later allocation of flows of control to processes and threads as well as specific issues relative to the Perl environment <or other target environment>.

<
Study how the collaborations described in the object model may be carried out concurrently. Focus on logical flows of control that execute concurrently rather than on processes and threads. Allocate data and behavior to those flows of control. Find out what data those flows need share and how they coordinate their behavior. Establish an abstract synchronization policy.

Concurrent state machines can be used to model those flows of control. They are represented in UML by statechart diagrams. Another useful tool is a UML collaboration diagram with focus on active objects and showing messages rooted by flow of control.
>

3.2. Allocation of processes and threads.

The <SW unit name> makes use of the Perl 5.8 built-in threading environment <or other target environment>.

<
A couple of lines to state the mapping of abstract flows of control to actual processes and threads.
>

The Perl threading environment that we're using is called *ithreads* (a short for interpreter threads). <or other target environment>

<
A couple of lines to explain why this environment has been chosen.
>

In the *ithreads* model each thread runs in its own Perl interpreter, all of its data is private and any data sharing must be explicit. In fact, when a new thread is created, all the data associated with the parent thread (the thread that spawned the new one) is copied to the new thread and can't be accessed from any other thread (including the parent thread) unless is marked as shared — through the *shared* attribute.
However, there are restrictions on what data may be shared. In fact, you can share scalars, arrays and hashes, but only simple values or references to shared variables may be assigned to shared array and hash elements. Moreover, even though references to shared variables can be passed among threads, it is not possible to share a blessed reference, which makes impossible to share objects directly.
Also notice that spawning a new thread after the parent thread has already accumulated a lot of data involves a considerable creation time overhead and subsequent expensive memory usage because of the cloning semantics of *ithreads*.

<
If not using ithreads, replace the above with information about the chosen environment. Briefly state the bits of the specific process/thread semantics that are going to affect the mapping of the logical concurrency model in 3.1 onto the target environment.
>

<
Now map the logical process model onto the target environment. Explain how processes and threads are created and managed. Detail what data belong to what process/thread and what data is to be shared. Decide what mechanism to use in order to maintain consistency on shared data. UML object diagrams can help focus on the discussion. Decide on the actual mechanism to coordinate processes and threads. In the case of processes, this is best described in the IPC section.
Tricky algorithms can be described with the help of UML activity diagrams.
>

3.3. IPC.

This section details the inter-process communication (IPC) in terms of communication protocols and means.

<

Specify the communication protocol, rules and roles as well as the format of the data that has to cross process boundaries.

>

<

Specify the communication means to be used among processes. Add noteworthy things to take into account during implementation.

>

References

[BRJ00] G. Booch, J. Rumbaugh, I. Jacobson:
The Unified Modeling Language User Guide
Addison-Wesley, 2000

[OMG01] Object Management Group:
Unified Modeling Language Specification v1.4
Available from http://www.uml.org/

<Other references>