Open Microscopy Environment

Installation Program
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 Installation program 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 required functionality and features of the Installation program. After that, we'll also give an outline of the solution, which is fully described in the next sections.

1.1. Requirements.

The Installation program is required to:

1. Perform automated installation.
    - Install and configure our versions of required libraries in private namespace (not possible for all of them).
    - Build C and Perl code.
    - Set up OME file system.
    - Configure Apache.
    - Create and configure OME database.
    - Install the core semantic types.
    - Create core analysis chains.

2. Rollback installation in the case of failure.

3. Support multiple platforms and DBMS.

4. Allow for more tasks to be added in future.

5. Perform automated removal of OME from the system (uninstall process).

<Add more detailed list of tasks and complete discussion>

1.2. Solution outline.

An installation program takes care of installation tasks. Those tasks are arranged using the Command Processor [POSA1] pattern. Every task is a class that implements a common *InstallationTask* interface, which declares two operations: *execute* and *rollback*. The *execute* method of a task object carries out the job by using one or more suppliers and stores the task state as it goes along in order to allow the *rollback* method to undo the work. The suppliers are Wrapper Façades [POSA2] to the platform, the database and the Web server. Every supplier class has a Factory Method [GoF95] to return a proper Wrapper Façade — façades are instantiated and configured as specified by a configuration file. A task and its suppliers are in a Publisher-Subscriber [POSA1] (a.k.a. Observer [GoF95]) relationship: a supplier fires events related to the state of execution of one of its methods and the task gets notified of such events in order to maintain the execution state for rollback. An *Installer* object takes care of instantiating, configuring and ordering the tasks that have to be performed for installation. It maintains a queue of tasks to be executed and a stack of already executed tasks. The *Installer* cyclically removes a task from the queue, calls its *execute* method and then pushes the task on the stack. In the case of failure, the *Installer* pops every element from the executed tasks stack and calls its *rollback* method. If all tasks successfully complete, the *Installer* serializes the executed tasks stack to disk in order to be able to uninstall — simply popping tasks and calling *rollback*.

1.3. Document overview.

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.

The Install program architecture is quite easy. It basically combines and builds on the Command Processor [POSA1] and the PAC [POSA1] patterns in order to group installation actions into tasks, to track the execution of those tasks — either to be able to roll back a faulty installation or to uninstall, and to have every task manage the required task-specific user input, if any.

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.

The overall structure of the Installation program is organized around three main abstractions:

- *Tasks to be performed*. Closely related installation actions are grouped into tasks. Every task is responsible for keeping track and maintaining state relative to the ongoing activity as well as managing user interaction, if any is needed to carry out the work. All tasks share a common interface, *InstallationTask*, which defines two operations: *execute* and *rollback*. The first operation is obviously implemented to start the activity and the second one to undo whatever has been done at the time it is invoked — this is possible because the task maintains execution state.
- *Service suppliers that those tasks need to carry out their work.* These are Wrapper Façades [POSA2] around specific platforms, DBMS and Web servers. All platform façades share the same public interface, but they obviously have different implementations. Every supplier class has a Factory Method [GoF95] to return a proper Wrapper Façade, specific to the platform at hand — façades are instantiated

and configured as specified by a configuration file. The same applies to DBMS and Web server façades. A task and its suppliers are in a Publisher-Subscriber [POSA1] (a.k.a. Observer [GoF95]) relationship: a supplier fires events related to the state of execution of one of its methods and the task gets notified of such events in order to maintain the execution state for rollback.

- *A task executor*. An *Installer* object takes care of instantiating, configuring and ordering the tasks that have to be performed for installation. It maintains a queue of tasks to be executed and a stack of already executed tasks. The *Installer* cyclically removes a task from the queue, calls its *execute* method and then pushes the task on the stack. In the case of failure, the *Installer* pops every element from the executed tasks stack and calls its *rollback* method. If all tasks successfully complete, the *Installer* serializes the executed tasks stack to disk in order to be able to uninstall — simply popping tasks and calling *rollback*.

The following UML class diagram depicts the overall program structure.

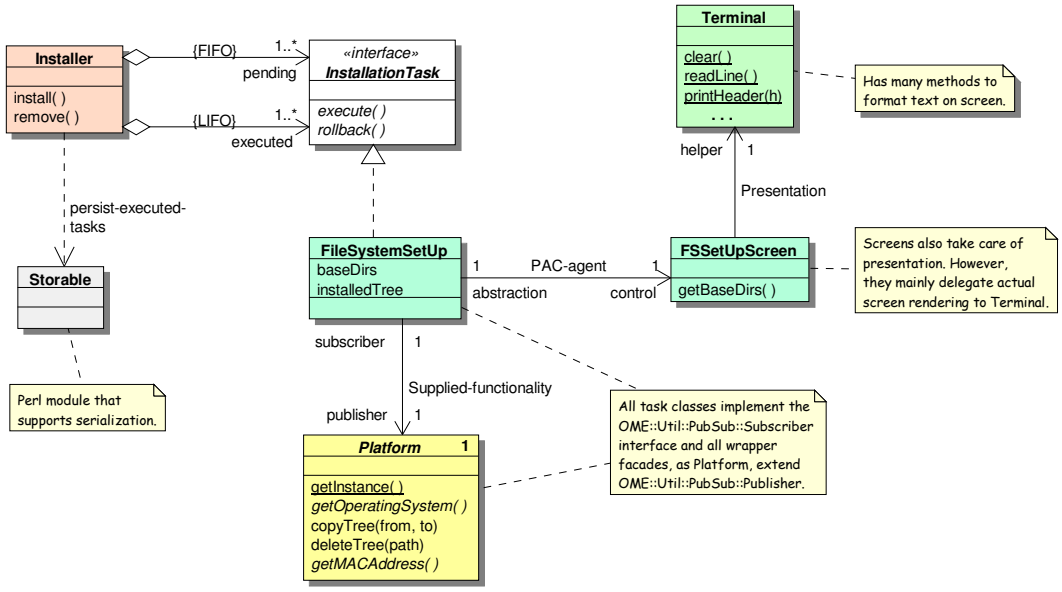

Fig 2-1: Overall static model.

The above diagram refers to the task of setting up the OME file system. Its only supplier is the *Platform* façade that encapsulates access to the specific operating system where OME is being installed. The *FileSystemSetUp* class knows what are the base directories where OME files shall go and maintains a list of the items currently installed. This list is dynamically updated as the *Platform* object copies the files. In fact, the *FileSystemSetUp* object registers interest in file-copied events which are fired by *Platform* every time a file is copied — recall the Publisher-Subscriber relationship. The *FileSystemSetUp* object gets to know about the actual base directories by asking its *FSSetUpScreen* object, which controls and oversees the acquisition of user input from the terminal.

Other tasks are arranged in a similar fashion.

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

<Necessary??>

2.1.2. Dynamics.

The overall behavior of the Installation program during a typical interaction scenario can be characterized by the following phases:

- Removing a task from the pending queue.
- Executing that task.
- Pushing the executed task on the executed stack.

The following UML sequence diagram further details a typical interaction. This diagram refers to the *FileSystemSetUp* task depicted in the overall static structure earlier. Similar concepts apply to all other tasks though.

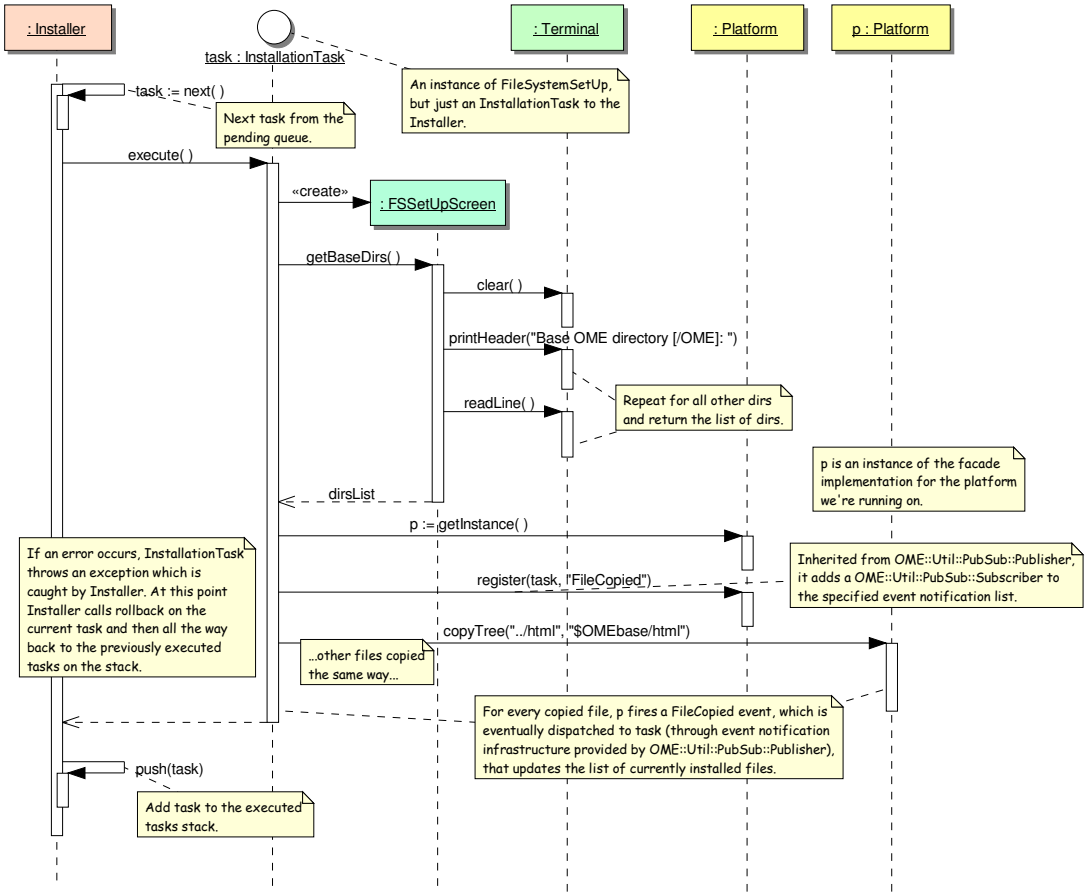

Fig 2-2: Overall dynamic model.

<
Maybe I should say a few more words on this section . . .
>

A final consideration on design patterns.

<
I guess it'd be better to explain why we don't have a Controller in our incarnation of Command Processor and why the Control takes on the role of Presentation (partially) in our version of PAC. Beneficial for those who know about design patterns . . .
>

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 requirements outlined in section 1.1.

Every task outlined in the Requirements section is implemented by a specific task class. If something goes wrong during installation, then whatever has been performed can be undone because the *Installer* keeps track of the tasks that have already been executed and every task maintains its execution state. This is also the key to an automated removal of the software from the system: when the program is asked to uninstall, then it simply has to retrieve all the tasks that were performed during installation and undo them.

Multiple platforms are supported by writing a specific Wrapper façade for each of them. All the façades have a common interface, but obviously a different implementation. Same story for different DBMS.

The installation procedure is extensible. In fact, more installation tasks can be easily added in future. Adding a task is only a matter of coding the specific task class and tell the *Installer* when it has to be executed during the installation process. The key to extensibility is obviously the *InstallationTask* interface, which decouples a task specific implementation from the environment in which it is run.

2.1.4. Rationale.

Installation has become an annoying obstacle to whomever tried to run OME. This is mainly due to the following reasons:

- OME relies on specific libraries and settings that may conflict with the current status of the platform where the system is being installed.
- Various platform-specific hacks have to be taken into account when installing.
- The average biologist has little understanding of the above factors (honestly, some of them could be classified as hacker's tricks) and so it is quite normal for them to get something wrong during a manual installation procedure.

The installation program aims to reduce to the minimum the above problems. Where possible, the third-party libraries that OME depends upon are installed in a private namespace in order to avoid conflicts with existing ones. All platform-specific hacks are enforced by the program and the installation procedure is automated in order to avoid human errors.

Now a few words on design choices. One possibility would have been to extend the glorious bootstrap script (*bootstrapOME.pl*) in order to accommodate a new, automated installation procedure. This script has already a lot of ready-to-use functionality and it has recently been re-factored to a cleaner and neater procedural style. However, this script, at present, comprises over 700 lines of code and it's not a good idea to inflate it even more. Also, the procedural paradigm wouldn't cope very well with the increased complexity and required flexibility and extensibility. A sound object model is needed. To this end, we can achieve a clean, flexible and extensible design through the use of the:

- Command Processor [POSA1] design pattern to manage the installation/removal procedure and to allow for rollback of a faulty installation.
- *InstallationTask* interface for extensibility.
- Wrapper façade [POSA2] and Factory Method [GoF95] design patterns to have a clean and uniform access to low-level platform and DBMS functionalities and to allow the program to be flexibly configured with different combinations of platforms and DBMS.

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.

<Necessary??>

References

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

[GoF95] E. Gamma, R. Helm, R. Johnson, J. Vlissides:
Design Patterns — Elements of Reusable Object-Oriented Software
Addison-Wesley, 1995

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

[POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal:
Pattern-Oriented Software Architecture —A System of Patterns
John Wiley & Sons, 1996

[POSA2] D. Schmidt, M. Stal, , H. Rohnert, F. Buschmann:
Pattern-Oriented Software Architecture —
Patterns for Concurrent and Networked Objects
John Wiley & Sons, 2001