System design document for SDMAPEG

Contents

Version: 1.0

Date: 2013-04-26

Authors: Victor Nilsson, Niclas Alexandersson, William Dahlberg, Oskar Nyberg

1 Introduction

1.1 Design goals

As this project includes a collection of three applications, there is a plethora of subjects that needs to be addressed in this document. First and foremost, the most basic goal is, all according to convention, to avoid complex connections, especially since the goal is to send data between different machines. In order to make the system as elegant and efficient as possible, this much is given. The GUI is not a vital part of the system and should be as loosely coupled as possible in order to allow further development of this interface. See the RAD for a description and usability of the project.

1.2 Definitions, acronyms and abbreviations

- GUI - Graphical User Interface
- Worker - the application performing the actual work
- Client - the application sending work requests
- Server - the application distributing work requests and returning the results
- Volunteer - the human user of the worker
- User - the human user of the client
- Administrator - the human user of the server
- Task - reference to a type of computation to be sent from the client to the worker.
- Result - reference to the result of a task
- Module - collective name for the worker, client and server applications

2 System design

2.1 Overview

The project is based on the MVC principle - however, since much of the work is handled automatically in the backend, it is hard to define this project as such. While the end product is to be made up of three applications, the actual project consists of six sub-projects.

2.1.1 API

In order to properly build this project, there was a need to define certain models beforehand, such as the connection between modules, the messages to be sent between them, and so forth. This results in the making of a miniscule API of sorts, allowing networking between applications and identification of sent data, as well as the definitions of the tasks that the system can process. All of the applications use this API in some manner.

2.1.2 Runnable applications

The three runnable applications do not carry any models for the actual tasks that are sent - those are instead kept in a separate package. The three runnable applications are mostly responsible for handling the network connections - with the exception of the worker module, which also contains everything that is necessary to actually perform the given tasks.

2.1.3 Networking

There are two packages dedicated to the messages that can be sent between the modules. Every interaction between two modules is represented by a class extending the Message interface. These are split up between messages that can be sent between the server and the client, and messages that can be sent between the server and the worker.

2.2 Software decomposition

2.2.1 General
The system is made up of a number of different modules. See the APPENDIX for package diagrams of all the modules.

2.2.2 Decomposition into subsystems
The system consists of three major subsystems; the server, the client, and the worker. There are also three additional modules containing classes common to more than one module; server-client, server-worker, and common.

The server is the central module, responsible for all communication between the other subsystems, as well as the distribution of tasks. The main responsibility of the server is to accept tasks from the clients, delegate them to workers, and then notify the appropriate client when a task has been completed. The server is itself divided into three submodules; the clients package for managing communication with clients, the workers package for coordinating workers, and the server package as a central bridge between the two.

The client is basically following the MVC model, with each graphical window having its own controller. The model classes are shared between the GUI classes and as loosely coupled with the view as possible.

The worker does also come with a GUI, but is not able to interact with the actual processing in any significant manner. Therefore a volunteer is unable to do much more with the worker other than view the tasks that this module has received. This module also contains all of the processing instructions for the supported tasks - those are contained and used only in this module.

The server-client and server-worker module mainly containts implementations of the Message interface - in other words these modules specify what messages that can be sent between the main modules. They also provide each task with an identification number, in order to keep track of where each task is sent and where its corresponding result is to be returned.

2.2.3 Layering
The system consists of three layers; the GUI layer, the logic layer, and the network layer. The GUI layer is responsible for the interaction between the system and a human user, while the network layer handles the communication between the different subsystems. The rest is handled by the logic layer.

2.2.4 Dependency analysis
Dependencies are shown in the figure below. There are no circular dependencies. Consult the APPENDIX for further diagrams on each of the modules shown in figure 1.

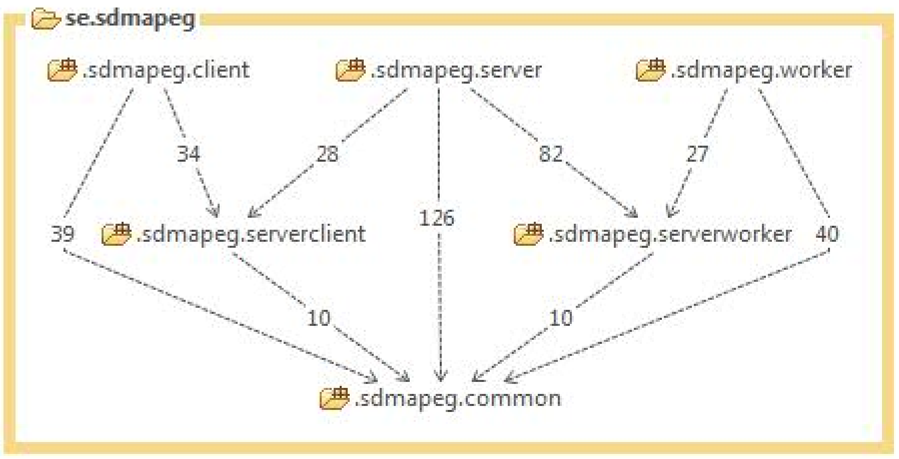


Figure 1: Dependency analysis

2.3 Concurrency issues

The system being designed in this project is largely concurrent. All applications need to listen for network input at all times, while still being capable of taking care of their other responsibilities and remaining responsive at the same time. This means that in all applications, at least two threads are needed – one for listening to network input, and one for executing user commands. For the server and the worker, however, even more threads will be needed.

The worker will, in addition to listening for network and user input, need to execute its long-running tasks in a separate thread in order to remain responsive. Additionally, to improve throughput, tasks will not run in a single thread, but rather in a thread pool in order to maximise CPU utilisation.

The server will be a largely concurrent system. In addition to the thread responsible for handling user input, it will have two threads listening for new connections – one for clients and one for workers. For each open connection to a client or a worker, an additional thread will be needed in order to listen for network input from them.

To reduce complexity as much as possible, thread confinement will be used extensively in order to make the points of interaction between different threads small and manageable. With smaller and more well defined points of interaction between threads, thread safety will be easier to reason about and enforce, using a combination of locking and atomic operations.

Special care also needs to be taken to ensure that all threads shut down properly when shutting down the applications. Threads need to be able to be stopped, and as such must be constructed in such a way that they can respond to interrupts or otherwise notice when it is time for them to shut down.

2.4 Persistent data management
Persistence is currently out of scope for this project.

2.5 Access control and security
In general, all communicating parties will be assumed to be trustworthy, with no authentication requirements. There will however, be some security restrictions in place for Python scripts being run by the workers, preventing malicious scripts from causing too much harm.

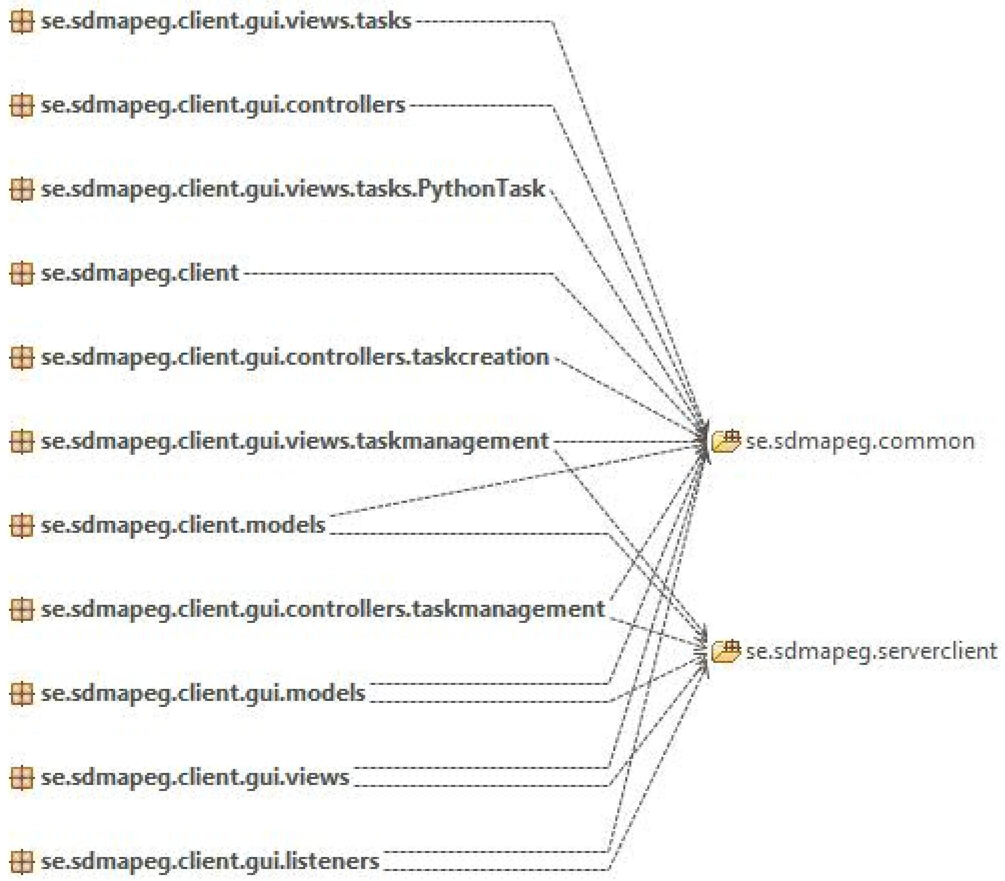
2.6 Boundary conditions
There are no currently known boundary conditions.

3 References
1: MVC: http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
2: Python: http://www.python.org/

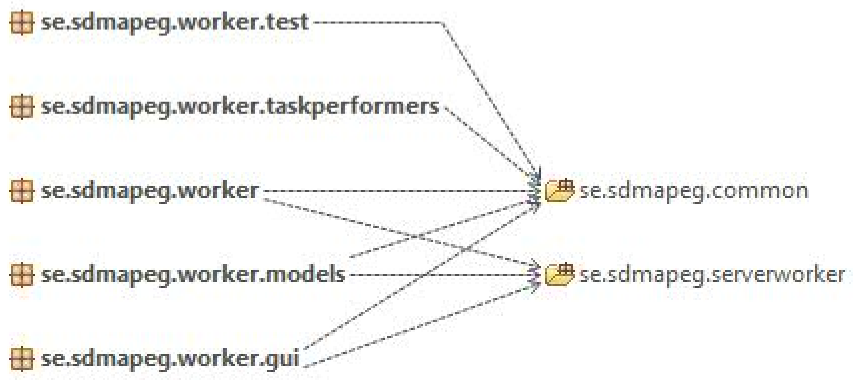
APPENDIX

Package diagrams - runnable modules

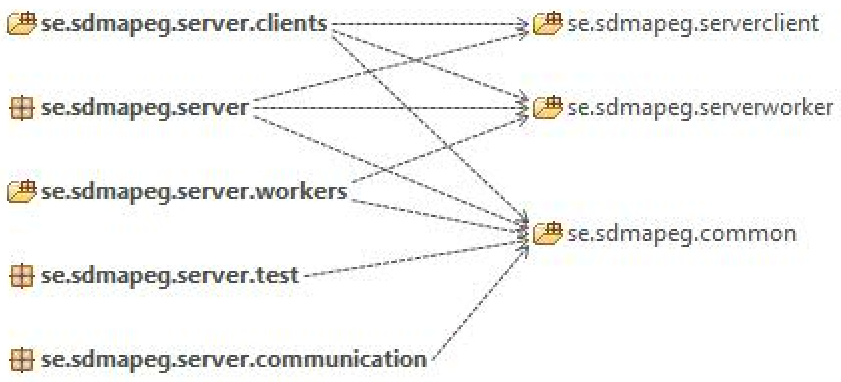
Each of the modules are represented by the diagrams below. Note that the server requires three separate diagrams in order to make them readable, and thus are separated.


se.sdmapeg.client.gui.views.tasks
se.sdmapeg.client.gui.controllers
se.sdmapeg.client.gui.views.tasks.PythonTask
se.sdmapeg.client
se.sdmapeg.client.gui.controllers.taskcreation
se.sdmapeg.client.gui.views.taskmanagement
se.sdmapeg.client.models
se.sdmapeg.client.gui.controllers.taskmanagement
se.sdmapeg.client.gui.models
se.sdmapeg.client.gui.views
se.sdmapeg.client.gui.listeners
se.sdmapeg.common
se.sdmapeg.serverclient

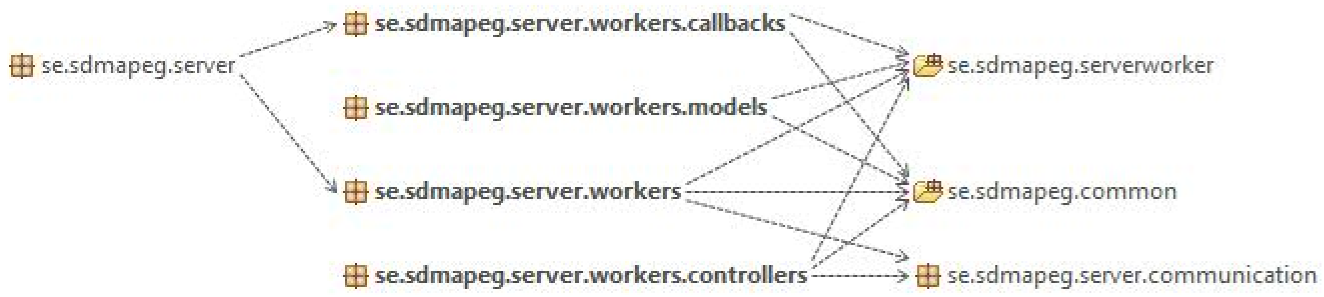
Package diagram: Client



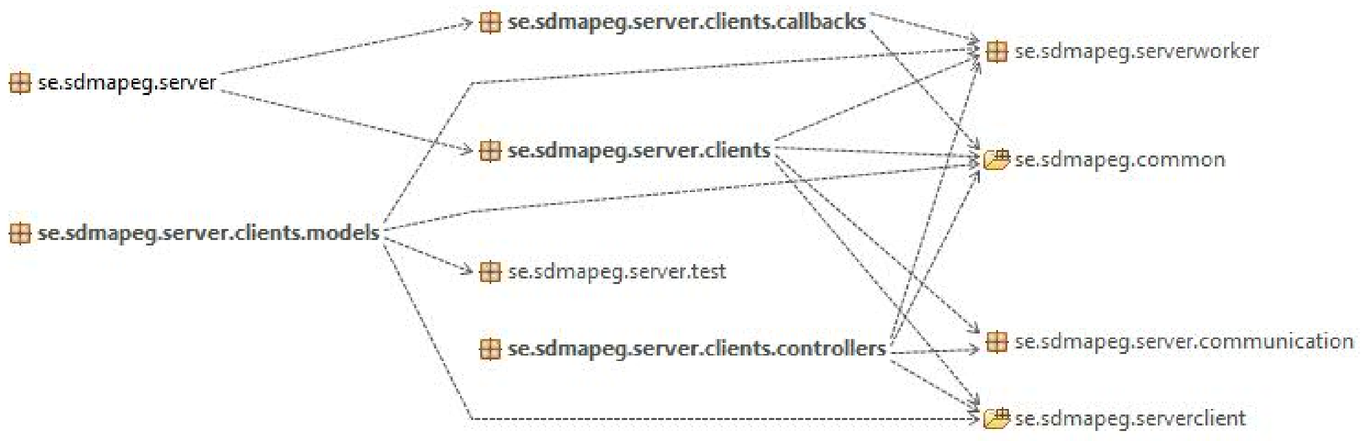
Package diagram: Worker



Package diagram: Server (main structure)


se.sdmapeg.server
se.sdmapeg.server.workers.callbacks
se.sdmapeg.server.workers.models
se.sdmapeg.server.workers
se.sdmapeg.server.workers.controllers
se.sdmapeg.serverworker
se.sdmapeg.common
se.sdmapeg.server.communication

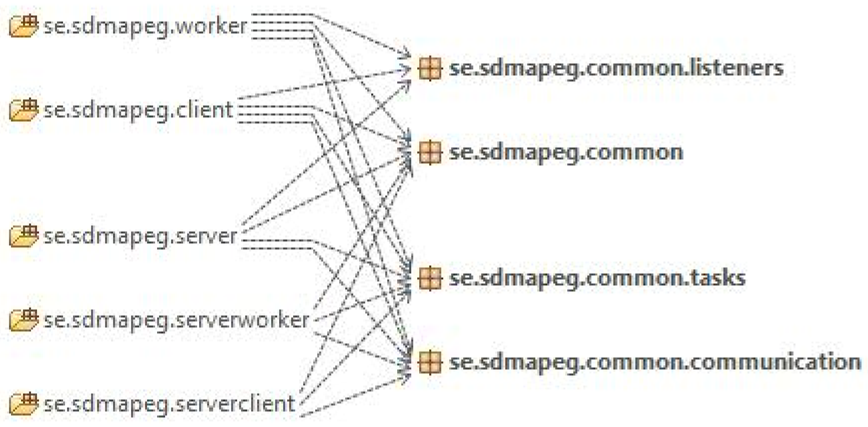
Package diagram: Server (Server.workers)


se.sdmapeg.server
se.sdmapeg.server.clients.models
se.sdmapeg.server.clients.callbacks
se.sdmapeg.server.clients
se.sdmapeg.server.test
se.sdmapeg.server.clients.controllers
se.sdmapeg.serverworker
se.sdmapeg.common
se.sdmapeg.server.communication
se.sdmapeg.serverclient

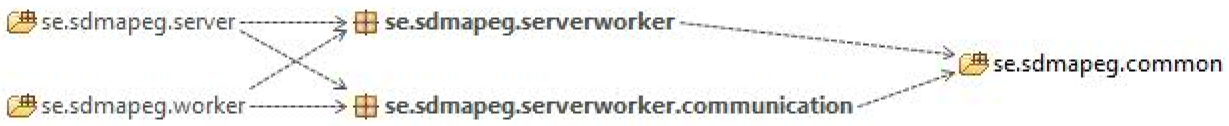
Package diagram: Server (Server.clients)

Package diagrams - communication modules and shared modules

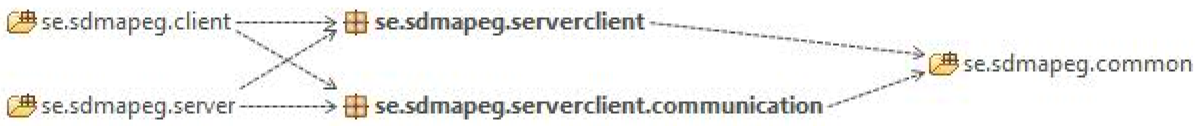
The diagrams below represents the structure of the remaining packages in the project. Those consist mainly of communication and exceptions, and are helper modules to the three main modules.



Package diagram: Common (classes used by all modules, for instance tasks)



Package diagram: Server-Worker module (Communication module server-worker)



Package diagram: Server-Client module (Communication module server-client)