Software Engineering for Industry: Analysis of JMVA Codebase

Yufei Wang (yw6312)
Paul Gribleyuk (pg1312)
Yawei Li (yl8012)
Jun He (jh1212)
Xiaoxing Yang (xy212)

Imperial College London

Deciphering the Codebase

- Initial Steps
  - We downloaded the project from SVN and imported it into IntelliJ
  - We fixe a text encoding (UTF-8 vs ISO-8859-1) in the files.
  - Next, we located the `main` method in `GraphStartScreen` and ran it to determine relationship between JMT and JMVA.
  - This helped us find the entry point into the JMVA application: the `ExactWizard` class
  - We observed how the application behaves by tracing the execution of button clicks (`JButton` and `Jpanel` actions)
- Functionality of JMVA
  - JMVA produces solution to Mean-Value-Analysis problem from queuing theory
  - The network model is defined by classes, stations, service time, etc.
  - What-if analysis can be performed by means of different values of parameters
  - The analysis of network performance will be represented in terms of throughput, utilization, system response time, etc.
  - What-if solutions are displayed in graphical or textual form

How to use JMVA

- Defining a model of queuing network
  - From an empty model:
  1. Parameters (classes, stations, service demands) are entered using 5 or 6 tabs in Wizard
  2. select control parameter in what-if tab if models need to change values in selected range
  - From an existing model:
  1. load the model through 'File' menu
- Model Solution
  1. use 'Solve' to solve the model
  2. performance (throughput, residence times, utilizations, synopsis panel) is shown on separate window
- Models can be saved into .jmva file

Graphing the Structure

- Deeper Analysis
  - Use of JDepend was not helpful because of the size of the project and large number of circular dependencies
  - IntelliJ Pro Dependency Matrix of JMVA (Figure 1) shows the coupling between packages (i.e. `analytical` package depends on `common` package in 74 places, etc.)
  - Strong circular link between `gui` and `analytical` (`gui -> analytical`, 282 times, `analytical -> gui` 262 times)
  - Specifically, `ExactModel` is the primary culprit as it is referenced in `analytical` package extensively

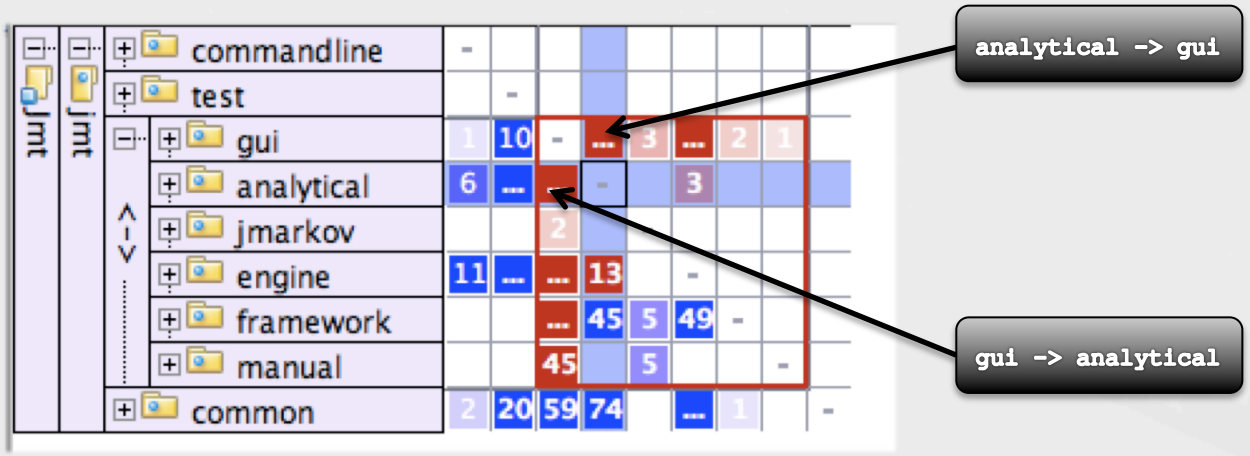
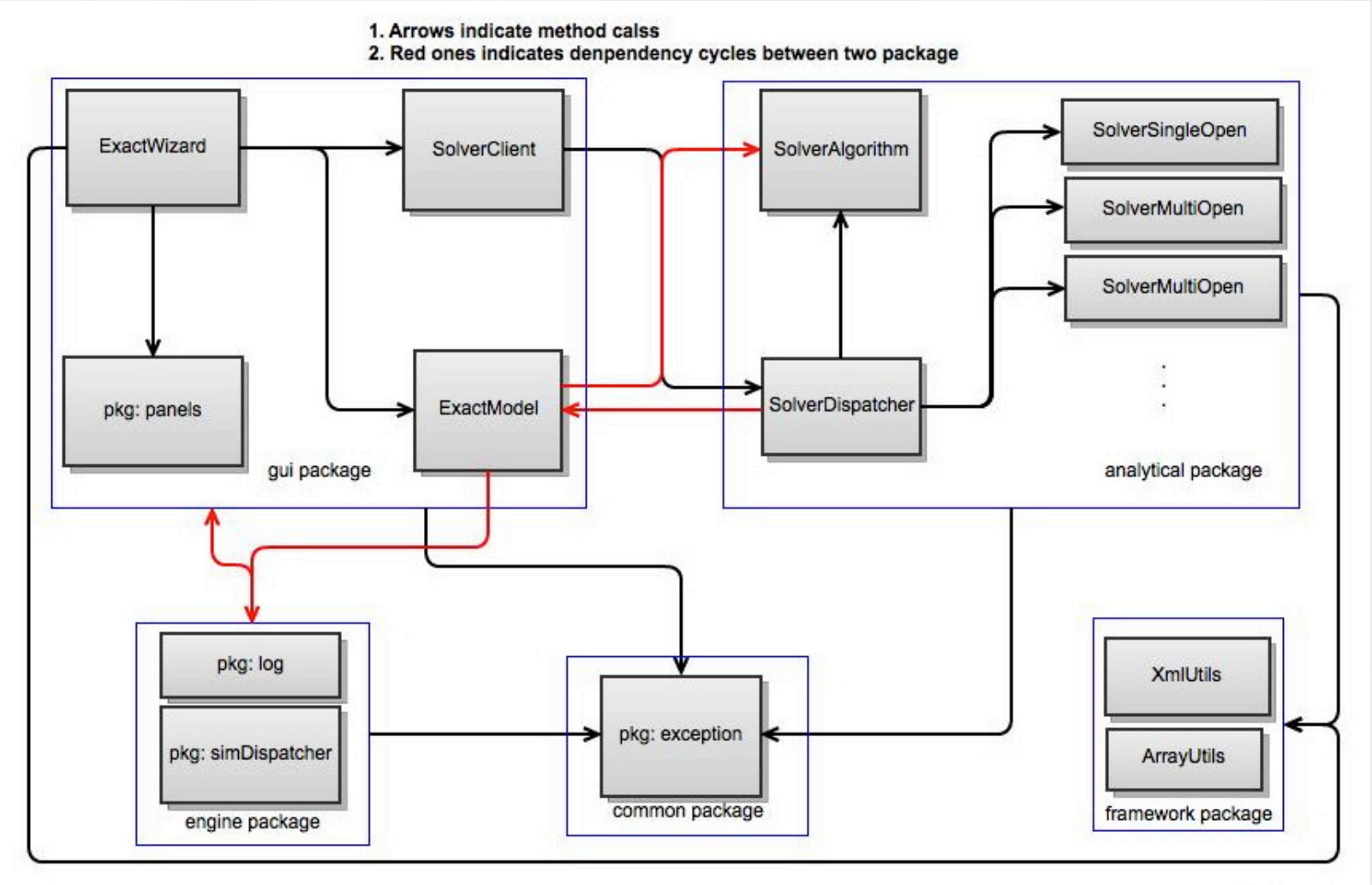


Figure 1

Critical Components



Deciphering the Codebase

- Use of STAN Tool:
  - Used to visualize layers of the code and find where changes will have greatest effect
  - Allowed us to quickly modify package structure to create more top-down structure by moving the `ExactModel` to the `analytical` package (Fig. 2a and 2b)
    - Ideally, `ExactModel` should exist in a separate package
    - The >2800 LOC file should be split into a hierarchical inheritance structure rather than attempting to describe every model in one class
  - Code metrics showed marked decrease in "entangled-ness" of project (7.49% to 1.45%). Further improvement possible.
- Tests
  - JMVA contains regression tests for different solvers (`TestAnalytical`), for different model inputs (`TestAMVA`), threads (`PauseThreadTest`), and others
  - These tests tend to measure performance rather than guaranteeing behaviour or state of objects

Improved Layout

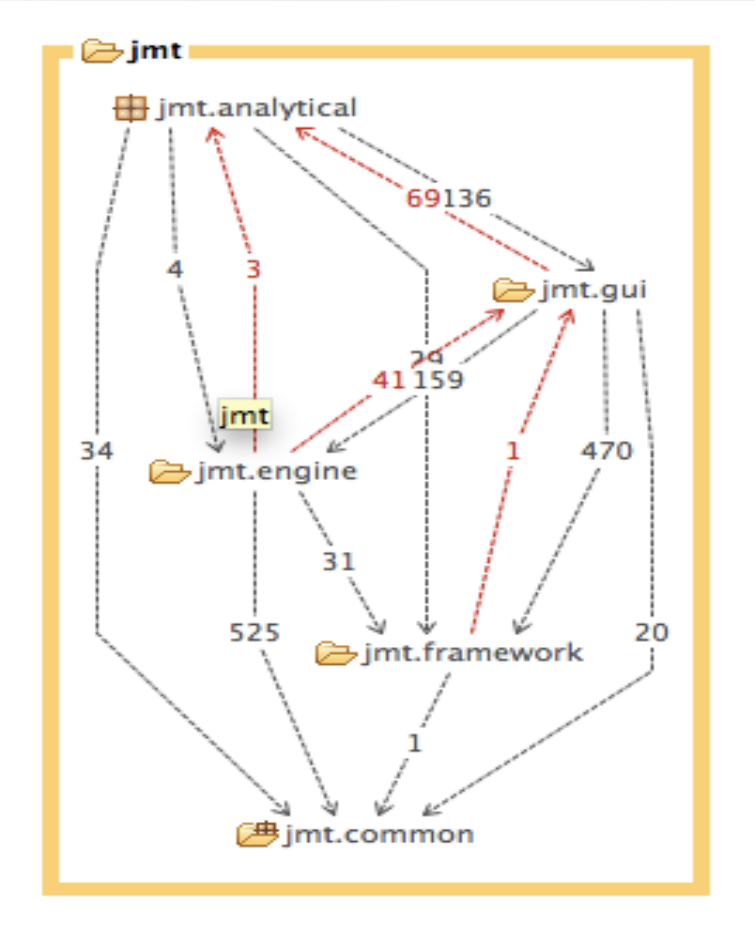


Figure 2a. Before refactoring

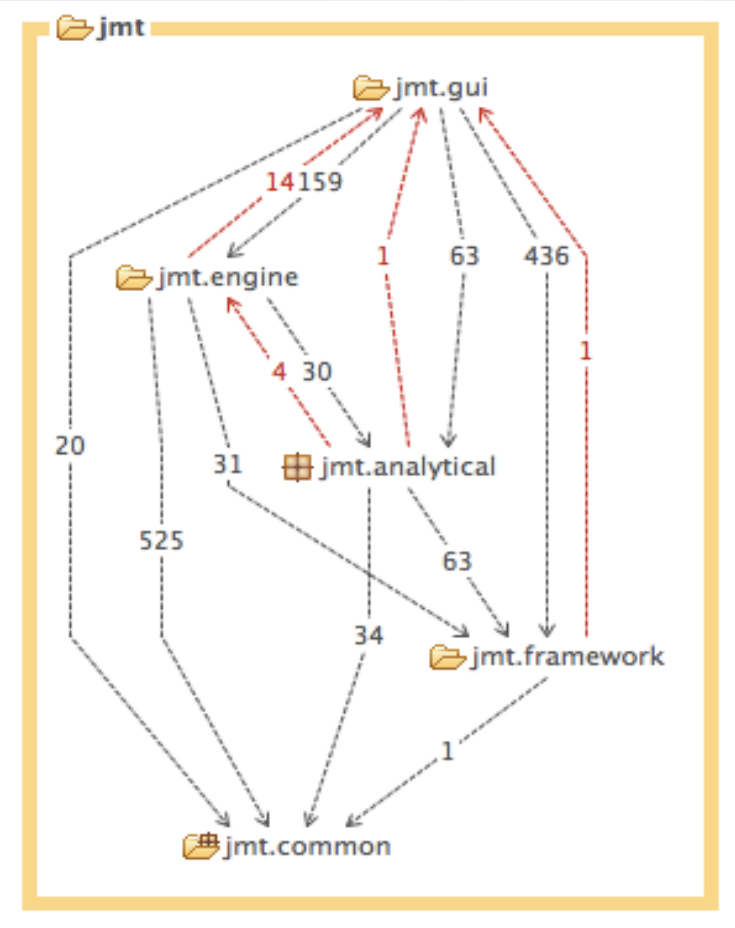


Figure 2b. After refactoring

Critique

- Program architecture
  - We recommend an interface layer between `SolverClient` and `SolverDispatcher`
  - We recommend moving classes not used by JMVA out of the `analytical` package
  - We recommend further refactoring of packages to reinforce a stricter layered structure between GUI components, Model controllers, and Analytics Engines
  - Cyclical dependencies between different layers make understanding and maintaining project very difficult
- Code
  - Comments are sparse and in Italian make it difficult to understand flow
  - Names of classes and methods obfuscate their responsibilities and actions (`Solver` doesn't solve, `solve()` method delegates responsibility rather than performing analysis, etc.)
  - The test package contains tests for main functional classes in analytical and engine package. The tests' output is either empty if succeed or error message printed to screen if failed. There is a lack of end-to-end tests and unit tests in other classes.

Critique (cont.)

- Components are often linked by direct call or reference (e.g. `DirectModel` class calls `Solver` classes in `analytical` package without any interface). This dependency on concretion set barriers for doing unit test with mock objects.
- Class cohesion is low (e.g. `ExactModel, SolverDispatcher`).
- `ExactWizard` breaks inheritance model by overriding superclass methods with empty methods (e.g. `finish()`); prefer to use interface to encapsulate commonality between classes in this case.

- Bugs
  - Application shows inconsistencies between execution and documentation (e.g. naming of tabs in solutions frame)