# SOEN 6441 – Advanced Programming Practices – Winter 2021

# Warzone Project - Build 2 – Refactoring – Team 1

## Potential Refactoring Targets

The following list of refactoring targets have been taken mainly from the new requirements established in build 2, and based on pain points and inconsistencies encountered during the development of build 1. Also, several of the targets were taken from code analysis done using PMD Source Code Analyzer (https://pmd.github.io/).

1. Implement State pattern to phase processing in GameEngine.
2. Implement Observer pattern for message processing to console and log file.
3. Implement Command pattern for processing of Orders
4. Refactor all controller constructors to accept only ControllerFactory objects
5. Refactor all writes to 'System.out.print' to be done via Utl.consoleMessage (makes console output and logging centralized)
6. Rename class names: MessageType, OrderExecutionController, SoftwareFactoryController, UserMessageModel, OrderDeploy, SoftwareFactoryModel, SoftwareFactoryView, IUserMessageModel, MapEditorConsoleView, GameStartupConsoleView, GamePlayConsoleView, IOrderModel
7. Move order execution implementation from player class to respective order classes
8. Cleanup unused methods
9. Use AssertSame instead OfAssertTrue in unit tests
10. Use AssertNull instead of AssertTrue in unit tests
11. Use AssertEquals instead of AssertTrue in unit tests
12. Rename method(s): setPlayer in Order class
13. Remove unused variables
14. Remove unused formal parameter(s)
15. Use collection IsEmpty instead of size() == 0 (or size() != 0, size() > 0, size() < 1)

## Actual Refactoring Targets

The list of actual refactoring taken from the target list above were chosen mainly because of the new requirements established in build 2, and on the greatest pain points and inconsistencies encountered during the development of build 1.

### Implement State pattern to phase processing in GameEngine

This refactoring was chosen because before refactoring the GameEngine held all the phase change logic centralized in several big methods. This made it difficult to maintain, enhance and unit test the phase changes.

#### Before / After Refactoring:

Before refactoring the GameEngine and respective controllers held all the phase change logic centralized in several big methods. The following is an excerpt:

|  |
| --- |
| d\_view.displayGamePlayBanner();              // main game play loop              while (!d\_exit) {                  // assigning reinforcements phase                  d\_msg\_model.setMessage(MessageType.None,                          "\n\* round " + l\_round++ + " \*\n\n\* assigning reinforcements:");                  d\_gameplay\_model.assignReinforcements();                  // issue\_order phase                  if (!d\_exit) {                      d\_msg\_model.setMessage(MessageType.None, "\n\* issuing orders:");                      int l\_num\_orders = issueOrders();                      if (l\_num\_orders < 1) {                          d\_msg\_model.setMessage(MessageType.Warning, "no new orders - ending game");                          break;                      }                  }                  // execute phase                  if (!d\_exit) {                      d\_msg\_model.setMessage(MessageType.None, "\n\* executing orders:");                      d\_gameplay\_model.executeOrders();                  }              } |

*Before State pattern changes*

In addition each phase required it’s own logic similar to that shown above. This made it difficult to maintain, enhance and unit test the phase changes. After the refactoring using the State pattern the phase change logic was easier to enhance and test. The following is an excerpt:

|  |
| --- |
| GameEngine:              while( d\_phase != null ) {                  d\_phase.execPhase();              }  Each of the phases managed above included the following code to support processing of the next phase:  nextPhase(l\_next\_phase); |

*After State pattern changes*

Each of the classes supporting the different phases simply set the next phase and the GameEngine is responsible for invoking the phase objects.

#### Unit Tests:

Build 1 did not include any unit tests that checked specifically for phase changes. The following checks were added in build 2:

|  |
| --- |
| public void test\_processMapEditorCommand\_loadmap\_valid() throws Exception {  String l\_msg;  Phase next\_phase = d\_map\_editor\_controller.processMapEditorCommand...  l\_msg = d\_msg.getLastMessageAndClear().d\_message;  assertTrue(l\_msg.contains("loadmap processed successfully"));  assertTrue(next\_phase instanceof GameStartupController);  } |

The same check was also added to:

* test\_processGameStartupCommand\_assigncountries\_valid() in GameStartupControllerTest
* test\_processGamePlayCommand\_exit\_deploy\_valid() in IssueOrderControllerTest

### Implement Observer pattern to message processing

This refactoring was chosen not only because it is a requirement in build 2, but it makes the application easier to maintain, enhance and test.

#### Before / After Refactoring:

Before refactoring the application supported the Observer pattern for message processing; however, some refactoring was required in order to fully comply with the build 2 requirements.

Most of the changes involved:

* Renaming class UserMessageModel to LogEntryBuffer
* consolidating output to the console via a call to Utl.lprintln instead of a direct call to System.out.println
* adding output to the log file

#### Unit Tests:

Almost every unit test tests the last message in an assert. The following is one of many examples:

|  |
| --- |
| @Test  public void test\_processGamePlayCommand\_deploy\_invalid() throws Exception {  String l\_msg;  d\_gameplay\_controller.processGamePlayCommand("deploy", d\_player);  l\_msg = d\_msg.getLastMessageAndClear().d\_message;  assertTrue(l\_msg.contains("no options specified"));  d\_gameplay\_controller.processGamePlayCommand("deploy Canada", d\_player);  l\_msg = d\_msg.getLastMessageAndClear().d\_message;  assertTrue(l\_msg.contains("number of reinforcements not specified."));  d\_gameplay\_controller.processGamePlayCommand("deploy 5 Canada", d\_player);  l\_msg = d\_msg.getLastMessageAndClear().d\_message;  assertTrue(l\_msg.contains("Invalid deploy country name '5'"));  // build 1 requirement: Unit testing framework  // (4) player cannot deploy more armies that there is in their reinforcement  // pool.  d\_player.addPlayerCountry(d\_country);  d\_player.setReinforcements(3);  d\_gameplay\_controller.processGamePlayCommand("deploy Canada 5", d\_player);  l\_msg = d\_msg.getLastMessageAndClear().d\_message;  assertTrue(l\_msg.contains("does not have enough reinforcements (3) to deploy 5…  d\_gameplay\_controller.processGamePlayCommand("deploy Canada -1", d\_player);  l\_msg = d\_msg.getLastMessageAndClear().d\_message;  assertTrue(l\_msg.contains("Invalid number of deploy reinforcements '-1'"));  } |

### Refactor all controller constructors to accept only ControllerFactory objects

This refactoring was chosen because it makes the application easier to maintain, enhance and test.

#### Before / After Refactoring:

Before refactoring the application required that each dependency to the controllers be provided in the constructors. This created a problem every time a new dependency was added to the constructors (or deleted), as the change required many changes across the application, especially in the unit tests.

After refactoring adding new dependencies simply required adding them to the software factory classes.

Changes involved:

* Changing the controller constructors
* Fixing all calls to the constructors

#### Unit Tests:

Existing unit tests were able to cover the refactoring. Including but not limited to the following unit tests:

* GameStartupControllerTest (all tests within)
* IssueOrderControllerTest (all tests within)
* MapEditorControllerTest (all tests within)

### Rename class names

This refactoring was chosen because it makes the application code easier to read.

#### Before / After Refactoring:

Before refactoring the application code was difficult to read as some of the class names were too long. After refactoring the application code was easier to read.

Changes involved renaming of the following classes:

* MessageType, MsgType
* OrderExecutionController, OrderExecController
* SoftwareFactoryController, ControllerFactory
* UserMessageModel, AppMsg
* OrderDeploy, DeployOrder
* SoftwareFactoryModel, ModelFactory
* SoftwareFactoryView, ViewFactory
* IUserMessageModel, IAppMsg
* MapEditorConsoleView, MapEditorConsole
* GameStartupConsoleView, GameStartupConsole
* GamePlayConsoleView, GamePlayConsole
* IOrderModel, IOrder

#### Unit Tests:

Existing unit tests were able to cover the refactoring. Invalid or unchanged class names are caught by the compiler.

### Move order execution implementation from player class to respective order classes

This refactoring was chosen because it places the code that executes the orders closer to the order class keeping the order processing logic encapsulated in one place (ie class). This makes the application easier to read, maintain, enhance and test.

#### Before / After Refactoring:

Before refactoring the execution of orders was separated into 2 classes, 1) the respective IOrder class and 2) Player class. After refactoring the code was all contained in the respective IOrder class, making the code was easier to read, maintain, enhance and test.

#### Unit Tests:

Existing unit tests were able to cover the refactoring. Including but not limited to the following unit tests:

* deploy order:   
  test\_deploy\_1 -> OrderDeployTest  
  test\_processGamePlayCommand\_deploy\_invalid -> IssueOrderControllerTest