Patterns of Software Development

Report on a Bohnanza Implementation

Jeroen Meijer (s0166111)

October 14, 2013

1 Design goals

1.1 Criteria

We offer our system open source and free of charge online[1]. We assume the application is a highly popular one and is used among many people. We identify the most important quality criteria for this environment as *extensibility* and *reusability*. The most affected stakeholders are the *developers*, *maintainers*, *testers* and *users*.

Extensibility is an external criteria; it is most important for the user. Reusability is more important for developers etc. Both criteria are also covered under the term modularity. A key feature of bohnanza is to add extensions to the standard game. Modularity allow the user to easily use one of the many extensions, such as High Bohn or spin-offs, such as Al Cabohne. Also, for developers modularity allows for easy implementations of these extensions.

A natural way of decomposing the application into modules to achieve modularity is to provide one base module which contains basic bohnanza logic. Another module containing the std bohnanza game and the third module containing the high bohn extension. Using these modules allows us to compile and test them separately. Also, to make an extension for bohnanza we can easily extend the base module. Every module is a distinct Eclipse project.

1.2 Flexibility

Creating flexible software is not easy. Flexibility may introduce risks in software, but it allows for faster adaptation of software when external changes are desired. The basic bohnanza module should allow for creating an extension easily. However one should not over-engineer software to much. That is, by trying to anticipate what a future release of the software should support. The way we support flexibility in the bohnanza game

[1] https://github.com/Meijuh/psd

is only for creating extensions of the standard bohnanza game. These extensions can be created by subtyping certain classes in the base module. An alternative would be to supply hooks in our base module. But one can only guess if this approach works well with extensions we have not implemented, such as el Cabohne.

One can not simply override any method if behaviour needs to be changed, because this is bad practice. As Checkstyle[2] suggests a method should either be final or abstract. Therefore if one needs to override a method a final method should be made abstract and (parts of it) should be implemented in a subtype. We find this closely relates to the *open-closed principle* of modularity. The base module is *open* because no classes are final and can thus be extended. But it is *closed* – thus harder to extend – because methods are final. In our base module behaviour is hard to *modify* but behaviour can however be easily *added*, since classes are not final. The trade-off here is that it is harder to modify behaviour of the base module and thus one can not easily break the way we intent bohnanza to work. On the other hand, it is unforeseeable if for some extension the base module needs to be changed. If this is the case one has to change the accessibility modifiers described earlier.

2 Design

Our design is split up in three modules.

bohnanza Mostly an abstract base module that contains logic for both the std and hb game.

bohnanza-std A concrete module that uses the *bohnanza* module in order to play the std bohnanza game.

bohnanza-hb A concrete module that uses the *bohnanza* module in order to play the hb bohnanza game.

The reason why these elements are composed this way and called modules is a result of the dependency injection (Guice[3]) framework we use. This framework supplies abstract module classes which need to be implemented. Such a module class contains a set of dependencies. For example the bohnanza module uses an abstract Player class. Both the bohnanza- and bohnanza-hb module use an concrete implementation hereof. Another reason why these modules are composed this way is due to the fact that we chose an extension (and the std game) should extend classes of the base game and not implement hooks.

Every module is designed according to the MVC pattern. The main classes that are involved in the MVC pattern are shown in Figure 1.

[2] http://checkstyle.sourceforge.net/
[3] https://code.google.com/p/google-guice/

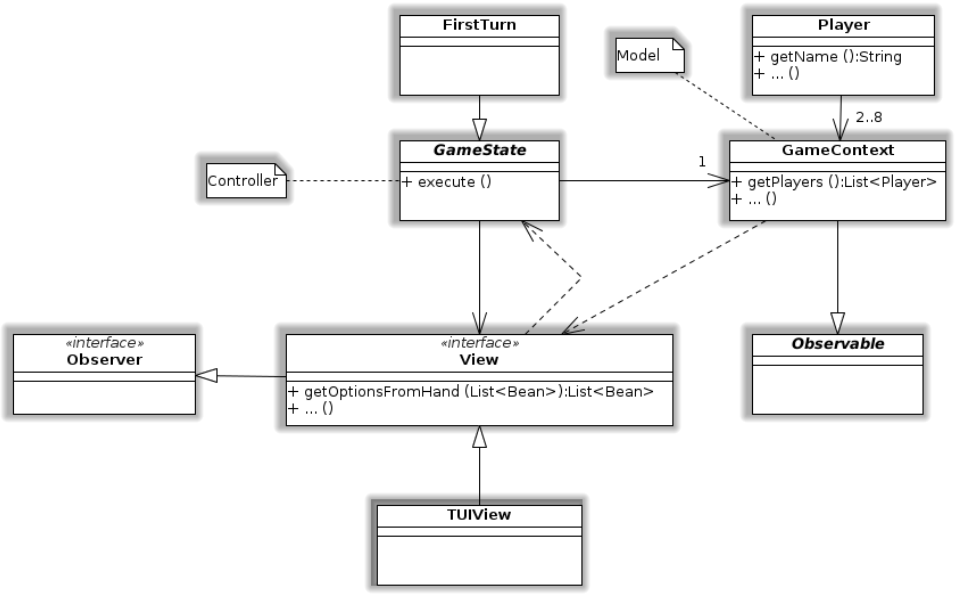

Figure 1: Classes involved in MVC

The `GameContext` is the main model class. It contains many assocations to other model classes such as the `Player`. Many model classes are `Observable` by `Observer`s such as the `View` class; this is how the view knows the status of the model. We could also have given the `GameContext` to the `View`.

The `GameState` is the controller which updates the view and the model and is part of a *state* pattern. The controller issues method calls such as `getOptionsFromHand()` in the view and the method returns a sub set of `Bean` cards. An alternative to the MVC pattern is the Model View Presenter pattern which could have been applied in this context. We chose to keep things simple and focus more on the extensibility of bohnanza rather than on implementing other views such as a graphical one. We think that the classic MVC pattern suffices here.

2.1 Model

Three important parts of the model involve the hierarchy of cards (Figure 2), the creation of cards (Figure 3) using a *factory method* pattern and the relation between different sets of cards and the players (Figure 4).

The hierarchy of the cards here is convenient, namely when creating extensions of bohnanza. When new cards are introduced to the game creating a new factory i.e. subtyping `CardCreator` or `BeanCreator` is easy because those factories expect to create a subtype of a `Card`. Instead of using a factory method pattern we could have approached

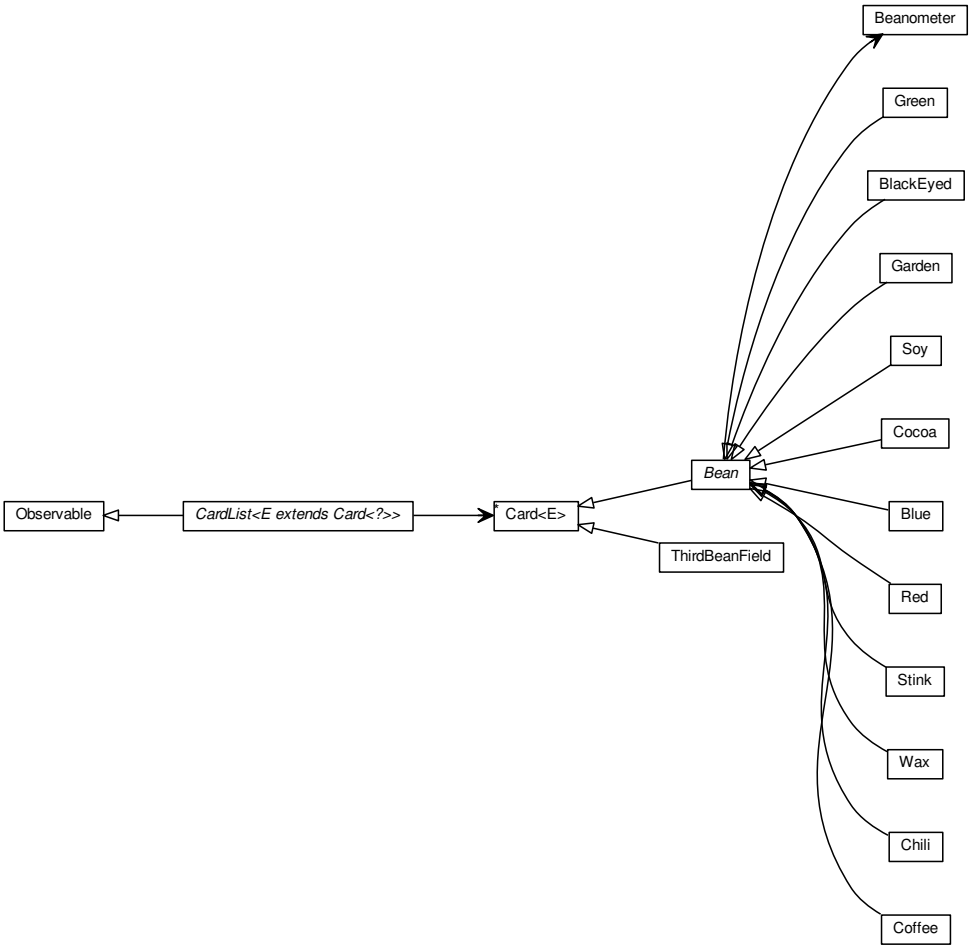

Figure 2: Class diagram of the cards

this a little different. Guice allows creating multiple instances of a class by means of a `Provider` class. Using this alternative would have resulted in cleaner code. The reason why we did not choose this alternative is because we introduced Guice into our project after we used the factory method pattern and did not want to restructure our code. We did however use the provider method for creating `Player`s as shown in Figure 6, because this was the only way to inject these objects into an `Bohnanza` object.

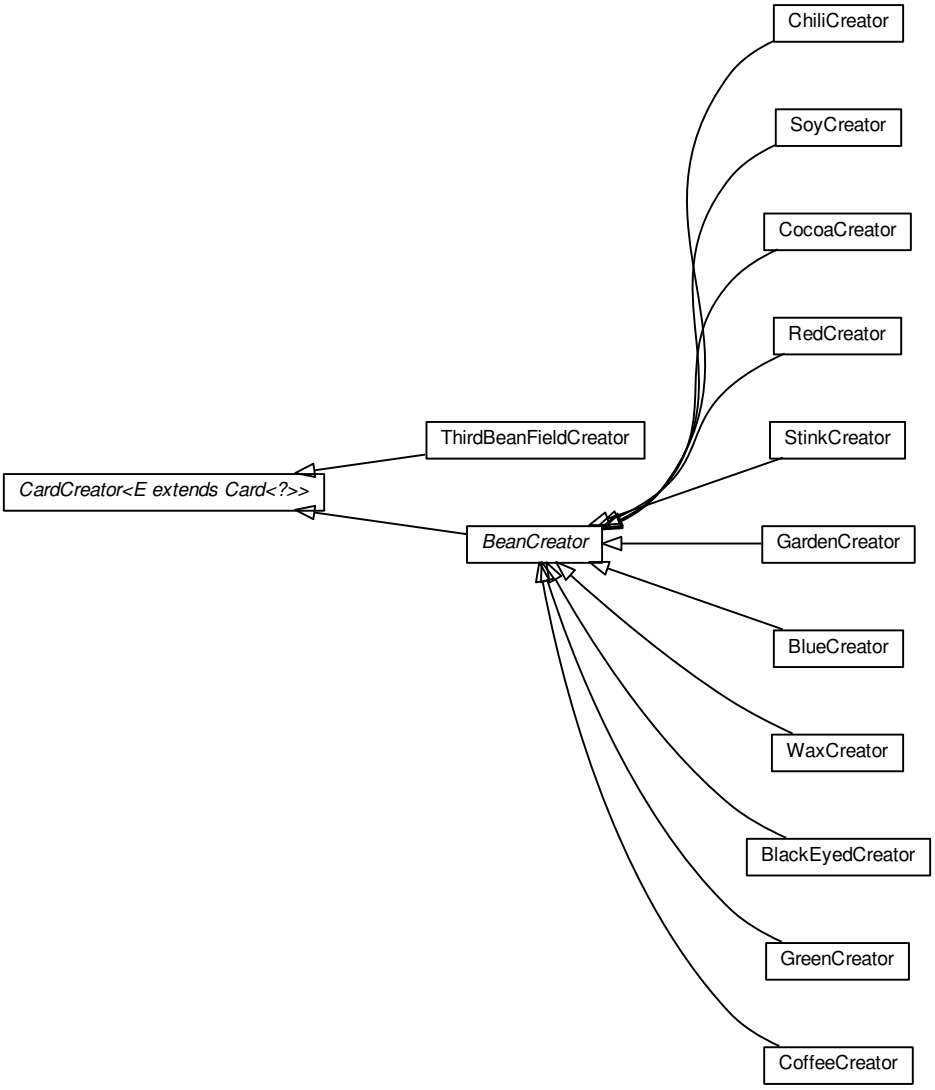

Figure 3: Class diagram for the creation of cards

2.2 Controller

The controller of the game is implemented using a *state* design pattern and is shown in Figure 5. In bohnanza the order of states is rather fixed, the order only varies depending on how often the deck is shuffled or in case of exeception behaviour. In case exceptional

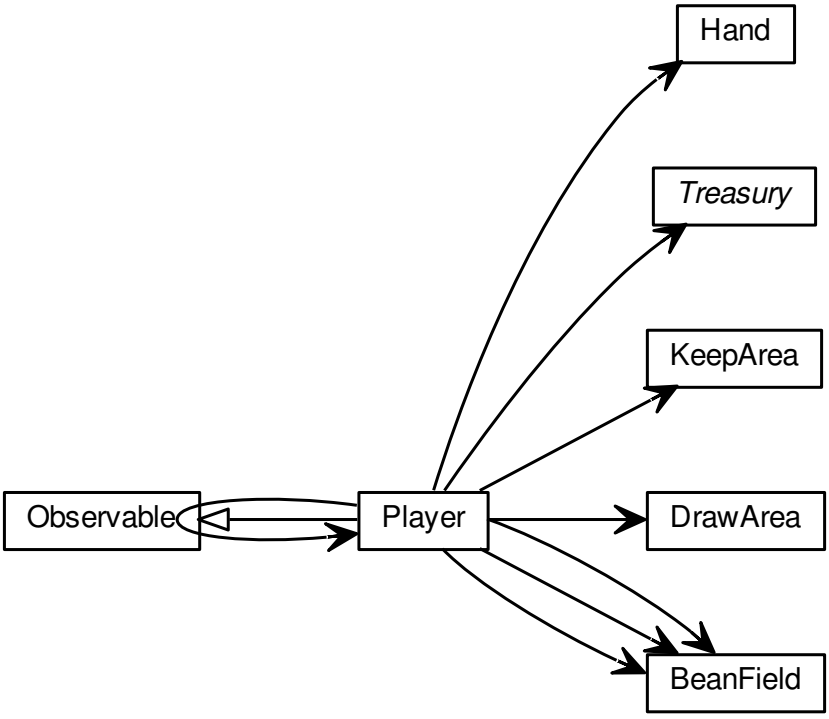

Figure 4: Class diagram for the player and association to particular sets of cards

behaviour occurs in our application the next state is always the `Fail` state. In general the order of states that are executed are `Prepare` → `FirstTurn` → `SecondTurn` → `ThirdTurn` → `FourthTurn` → `FirstTurn` → ... → `End`. A typical implementation of the `GameState.execute` method is shown from `FirstTurn` in Listing 1. Note how the next state is set by getting the singleton instance on Line 19. The application continues until a `GameState` sets the next state to `null`, i.e. when `GameContext.getState() == null`.

Listing 1: FirstTurn.execute()

```
1  @Override
2  public void execute(GameContext context) {
3
4      if (context.getCurrentPlayer().getHand().size() > 0) {
```

```
5
6           int beanFieldNumber = context.getView().mustPlant(context.getCurrentPlayer());
7
8           try {
9
10              context.getCurrentPlayer().plantHand(beanFieldNumber);
11
12              if (context.getCurrentPlayer().getHand().size() > 0) {
13                  beanFieldNumber = context.getView().mayPlant(context.getCurrentPlayer());
14
15                  context.getCurrentPlayer().plantHand(beanFieldNumber);
16
17              }
18
19              context.changeState(SecondTurn.getInstance());
20
21          } catch (BohnanzaException e) {
22              context.changeState(Fail.getInstance());
23              Fail.getInstance().setException(e);
24          }
25      }
26  }
```

As concrete state classes we could have merged states such as the first turn and second turn. But we split the states the way we did, because they form units which can conveniently be tested in isolation. An entirely alternative design pattern for the controller we could use here is the *strategy* pattern. The strategy pattern is more applicable in situations where strategies are chosen at runtime. But more importantly the strategy pattern is used in situations where the result of different strategies is the same, of course in bohnanza this is not the case.

2.3 View

Currently we support only a textual user interface. The `TUIView` class is an implementation of a `View` interface. Creating a new interface like a graphical or a network would involve implementing the `View` interface.

2.4 Packages

Our modules contain the following packages.

bohnanza contains the main class.

bohnanza.game contain some models for bohnanza, such as a `Card`.

bohnanza.game.factory contains classes w.r.t. the creation of cards.

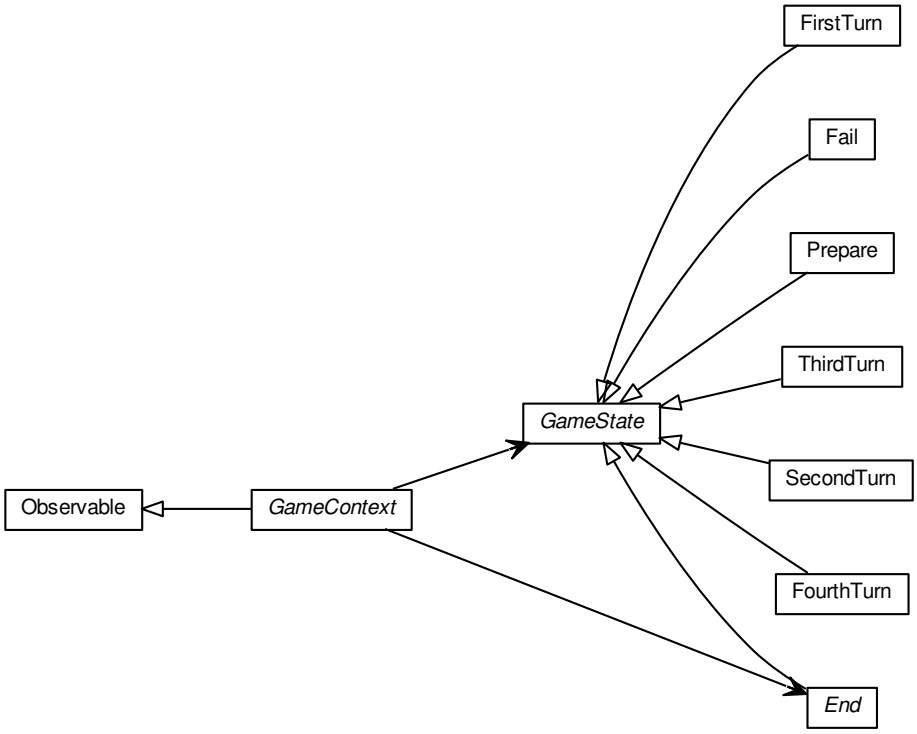

Figure 5: Class diagram for controlling the state of the game

bohnanza.game.player contains model classes w.r.t. the player.

bohnanza.game.shared contains model classes which may be accessed by all players.

bohnanza.game.gameplay contains controller classes w.r.t. the discussed state pattern.

bohnanza.module contains classes w.r.t. dependency injection of Guice.

bohnanza.umlgraph contains classes that model UML graphs used in this report.

bohnanza.view contains the view classes.

The packages are rather self explanatory. It can be argued however that some packages such as `bohnanza.game.player` and `bohnanza.game.shared` may be merged. We did not do this because in an early stage of the development we chose to make some classes *package private* such that for example only a `Player` could construct a `Hand`. This

would make sure there can only be as many `Hand`s in the game as there are `Player`s. This design choice was one of the reasons our code was not testable and we had to change the visibility parameters to use dependency injection.

2.5 Variations

2.5.1 User

The user can specify during the start up of the application how many players there are using a command line option. For example `java bohnanza.BohnanzaStd -p=3` starts the BohnanzaStd module with three players. Parsing command line options is done using the Apache Commons CLI library.

2.5.2 Developer

Variability for the developer depends heavily on notion of *dependency injection*. Dependency injection is a design pattern that basically involves moving dependency resolution from a particular class to a framework dedicated to dependency injection. Classes involved with dependency injection are shown in Figure 6. If a developer wants another variation of the application the developer can simply extend the `BohnanzaModule` class and define other classes to inject, note that BohnanzaModule extends the `AbstractModule` class from Guice. For example if a `Hand` object needs to behave differently for a particular extension of bohnanza one can extend the `Hand` class and define the extension in the extension of BohnanzaModule.

BohnanzaModule itself is an abstract class and this class only knows the full list of required dependencies. A concrete class such as BohnanzaStdModule implements some methods that return concrete classes as dependencies. This relates closely to the *single choice principle*. Listing 2 shows how one can change dependencies using an extension of the AbstractModule class. Note how the singleton pattern and the provider incorporate nicely in the configure method.

Listing 2: BohnanzaModule.java

```
1  public abstract class BohnanzaModule extends AbstractModule {
2
3      @Override
4      protected void configure() {
5          bind(View.class).to(getViewClass());
6          bind(DrawDeck.class).in(Singleton.class);
7          bind(Player.class).toProvider(getPlayerProviderClass());
8          ...
9      }
10
11     protected abstract Class<? extends Provider<? extends Player>> getPlayerProviderClass();
```

```
12      protected abstract Class<? extends DrawDeck> getDrawDeckClass();
13      protected abstract Class<? extends View> getViewClass();
14      ...
15  }
```

We have not really considered an alternative to dependency injection, because to our knowledge there is no real alternative for it. There is however the service locater pattern, but both these patterns solve a problem which is not that hard, that is injecting instances of some classes in another object instead of constructing these instances in that particular object. We found that Guice was a good enough framework, so we did not consider an alternate framework.

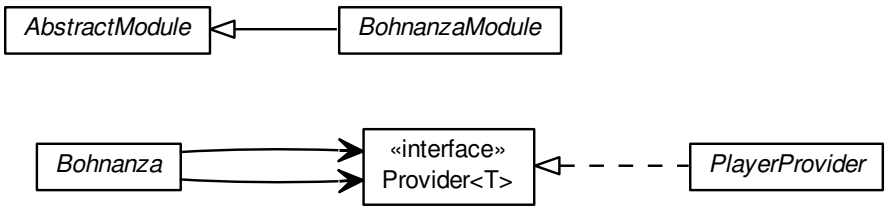

Figure 6: Class diagram of classes involving dependency injection

3 testing

The main goal of our testing approach is to test the more complex parts of the game such as trading. One of our goals is not to achieve 100 percent statement coverage, but to test the complex parts in a nice way.

Our design is well suited for testing. Since our design adheres to the dependency injection pattern injecting mocked objects of classes is easy. For mocking we use the Mockito[4] framework. Since we made our design work with Guice making our design work with Mockito is easy. Mocking namely requires dependency injection to inject mocks into the object under test. Interesting to see is that both Guice and Mockito use annotations to inject objects, these annotations are *@Inject* and *@InjectMocks* respectively.

Eclipse does not generate unit tests well from existing classes. So we installed an Eclipse plugin named moreunit, which is able to generate unit tests including necessary mocks.

[4]https://code.google.com/p/mockito/

Up until installing moreunit we were not aware of any mocking frameworks. Moreunit suggested to use Mockito and we found that this framework is the most mature one and beats mocking by hand.

For testing complicated algorithms one can use methods like *equivalence partitioning* and *boundary value analysis*. We have not used these methods due to timing constraints, but they would have been well suited for testing algorithms such as the trading part in bohnanza. Using these methods would result in far better coverage and even more likely increase multi condition coverage.

Each of the three modules can be compiled and tested separately. However the *bohnanza* module is rather abstract and thus testing some abstract classes do not really make sense. We test more concrete classes in either the standard module or the high bohn module. In the bohanza module we mainly tested whether the `Beanometer` and `CardList` works correctly. This is shown in the coverage report in the package `bohnanza.game`. Since a game can not really be played in the bohnanza module the game-play is tested in a concrete implementation of bohnanza, namely bohnanza-std. This is shown in the package `bohnanza.gameplay`.

The statement coverage we achieved is shown in Figure 7 and 8, both figures are generated using EclEmma[5].

Figure 7: Coverage of the *bohnanza* module

[5]http://www.eclemma.org/

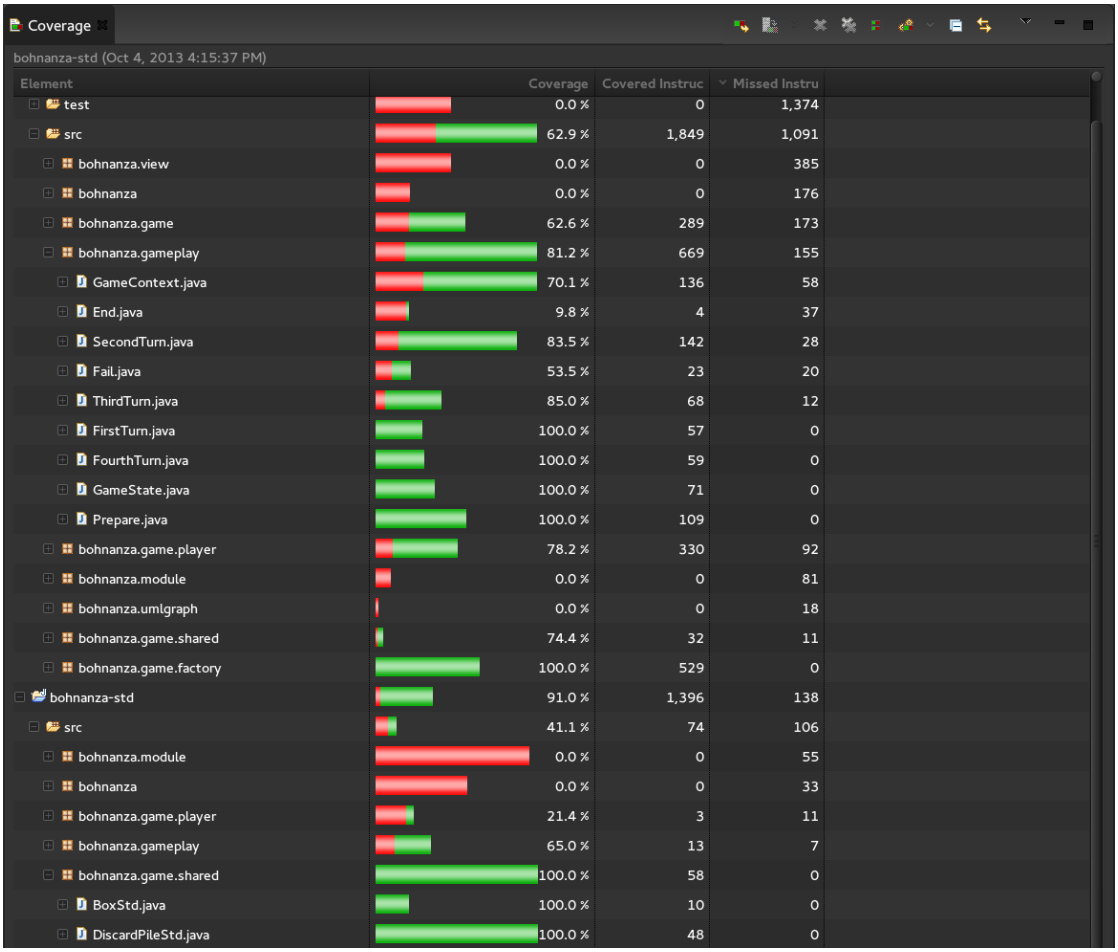

Figure 8: Coverage of the *bohanza-std* module

4 development process

First we drew a class diagram using Software Ideas Modeler[6] including methods and attributes. Using this diagram we generated Java class skeletons. Soon we came to the conclusion that many parts of our application could not be tested well, because we tried to protect it from incorrect behaviour to much by minimizing the amount of collaborations.

After standard bohnanza was implemented we began working on high bohn. We decided we could use the dependency injection framework to create the three modules. At this moment in time we saw that in order to have both the standard and high bohn implementation we needed three instead of two modules. The main reason is that you should design for extension as suggested by Checkstyle (see subsection 1.2).

[6] http://www.softwareideas.net/

After we were done with the implementations we used UmlGraph [7] to generate necessary class diagrams and write this report.

Our design phase was rather short; we drew only a class diagram. In a bachelor course we learned proper software design with several phases such as domain analysis, domain models, high level designs, detailed designs and more. The class diagram we drew for bohnanza is most reasonably a detailed design. We find that the method learned during the bachelor course is well suited for creating complex software systems, but since this approach does not fit well in the required report for this assignment (it should focus on the detailed design) we decided to focus more on explaining design decisions and applied techniques.

4.1 lessons learned

Only rather late in the development process we came to the conclusion that our implementation was hard to test. Should we have used a Test Driven Development approach we would have come to the conclusion to use dependency injection much earlier. So on the principle of making testable code we might have failed, but we learned. We know that dependency injection is key to making testable code, however we do not think TDD key to software development; sometimes implementing first and then generate a unit test to some degree can be just as convenient.

Minimizing the amount of classes with which a class colaborates is a good design goal (heuristic *Uses Relationship one* (UR1) from lecture two). One example where we tried to apply this was by introducing classes in the model which do have a role in the game such as a player area but no real behaviour. A class representing this role (the player area) would have methods to perform atomic operations making it harder to allow incorrect behaviour. The class would have bean fields, a draw area, a keep area etcetera, but not the discard pile. An atomic operation would be to plant a card from the draw area into a bean field. An operation a player area is not capable of is to move a card from the bean field to the discard pile. This is an advantage because it prevents incorrect behaviour. However we chose not to implement these kinds of classes because it makes it hard to test and proxy methods like `Player.getPlayerArea().getFarm().getBeanField()` make the source code unreadable. Instead we implemented methods directly in the `Player`, such that `Player.getBeanField()` is available. We changed this by refactoring in Eclipse; it allowed us to move methods and then remove classes such as the player area. One major reason why applying heuristic UR1 is not practical is that Mockito does not support injecting mocked objects into other mocked objects – it is considered bad practise. For example if we want to test if a keep area has certain cards we have to mock both the player area and the keep area and then inject the keep area in the player area. We therefore make the observation that the heuristic is not applicable where mocking is necessary. Our solution has of course one

[7] http://www.umlgraph.org/doc/indexw.html

major disadvantage, that is any `CardList` such as a `BeanField` exposes a list of cards it contains. Exposing this list allows to change the order of cards and even remove them. That is why every `CardList` exposes only a method that returns a (new) unmodifiable list of cards.

Some good principles we have learned came from using tools. After we installed moreunit it suggested to use the Mockito framework. This allowed us to easily mock classes such as the view. Instead of subtyping a view class we could simply invoke `View view = mock(View.class);` and `when(view.getOptionsFromHand()) .thenReturn(new ArrayList<Bean>());`. But more importantly what we learned is that one should not do partial mocking. That is either one should mock everything of a class or nothing. Partial mocking is done by mocking one method and invoke the real implementation of another method. Mockito warns the developer when he/she uses partial mocking.

Because mocking does not allow assigning values to private fields of mocked objects one has to *verify* behaviour. In Mockito this can be done with the `verify` method, e.g. `verify(discardPile, atLeastOnce()).add(new BlackEyed());`. For verifying advanced argument captors can be used and asserting that such a captor contains a certain value.

Some developers circumvent the need for partial mocking by using a tool called PowerMock[8]. PowerMock injects bytecode at runtime to test certain things. From what we have seen if one has to use PowerMock, either the design is flawed or one is dependent on external libraries.

Another good principle we have learned came from using Checkstyle. Checkstyle warned us about the fact that a method should either be final or abstract, which we discussed earlier.

Furthermore we have learned there is only one good free and open source tool for generating class diagrams. This tool is called UMLGraph and uses Java classes for specifying – with visibility parameters – certain *view*s on your design. To generate these diagrams one has to pass a custom doclet to the javadoc command. Additionally, Visual Paradigm[9] is a closed source and costly alternative which also supports round-tripping which is very convenient for larger projects.

[8] https://code.google.com/p/powermock/
[9] http://www.visual-paradigm.com/