SEN1 practical task
State machine using enums testing using mocks

Pieter van den Hombergh

18th May 2016

Gum balls as Olifantys

Figure 1: Think Bigger?? Some would like their balls bigger.

Learning goals

This task has the following learning goals:

- Using enum as state pattern implementation. (enum)
- Use Mocking of input and out streams with ByteArrayOutputStream. (io streams)
- Use of mockito to monitor calls on collaborator.
- Optional: use of maven to resolve dependencies.

The idea of the state machine comes from the book Head First Patterns.

Design

Olifantys has gum ball machines, selling olifantys balls, the mother of all gumballs.
You will write tests and implement a state machine using enums. The state machine should support the following behavior:
The state machine starts in the filled state, filled with a number of balls determined by a constructor parameter.

Behavoir

The state model (state diagram) can be found in figure 2. We have four states:
SoldOut No balls.
NoCoin No coin in coin slot.
HasCoin A coin in coin slot.
Winner The customer is lucky and should get an extra ball.
(excluding the initial pseudo state) and three events:
refill New balls in machine, (maybe also remove collected coins).
insertCoin Put a coin in slot.
ejectCoin Get the coin back.
draw Pull to get a ball.
The implementation should be done with en enum and one enum constant (value) for each of the said states. The method all have as first parameter the context, which is the statemachine context. (The class whose behavior you are implementing).
The context has a few methods and fields which can be seen in the class diagram in figure 3

Test driven where applicable

Instead of a main program that is not very useful, write tests for the state enum, using the following concepts:
In the test packages, implement a mocked PrintStream, which instead of writing to an output channel or file, writes (appends) to a string. See earlier (Java 2) lesson demo for the idea. Then instead of using the State machine in the business package, create a mock using Mockito.

Hints and Tips

The state-enum implements the `State` interface as seen in figure 3.

State interface

```
interface State {
    void insertCoin(Context gbm);
    void ejectCoin(Context gbm);
    void draw(Context gbm);
    void refill(Context gbm);
}
```

You can have multiple approaches here:

- Leave all methods abstract en only implement them in the values.
- Implement all methods as normal methods and overwrite only those that are different per instance.
- A compromise between the two above, saving double work, or possible *copy and waste* pro-

gramming.

In this concrete case you do best with the second approach, because even empty implementations as normal methods are useful because effectively, the method will have no effect which is quite the same as ignoring the event. It can also be useful to let the default methods throw exceptions if for instance you want to make certain method calls in some states illegal. This is not what we use, but might be applicable in other situations.

State implemented as enum, fragments

```
public enum StateEnum implements State {
  NoCoin {
    @Override
      public void insertCoin( Context gbc ) {
      gbc.getOutput().println( "You inserted a ↩
          coin" );
      gbc.setState( HasCoin );
    }
    // enum values left out
  }
  // example default methods
  @Override
  public void draw( Context gbc ) {
    gbc.getOutput().println( reason() );
  }

  @Override
  public void insertCoin( Context gbc ) {
    gbc.getOutput().println( reason() );
  }

  @Override
  public void refill( Context gbm ) {
    gbm.getOutput().println(reason());
  }
}
```

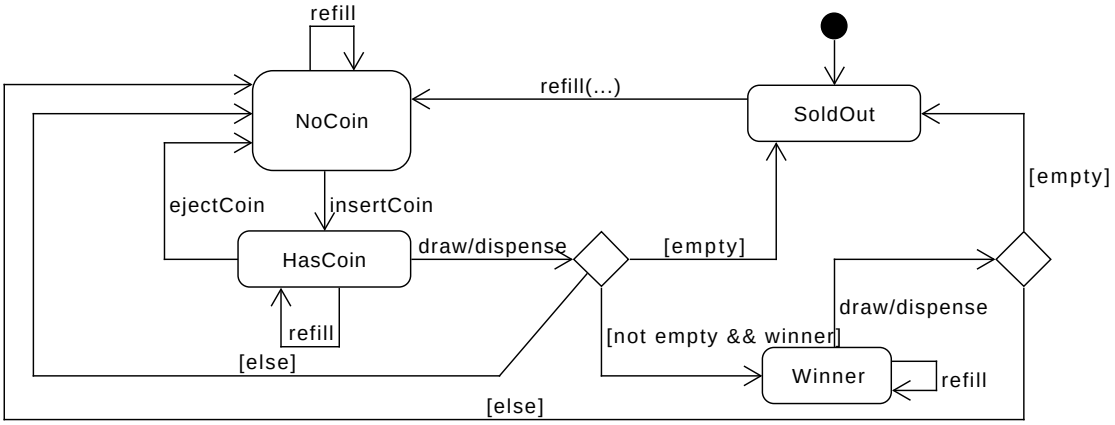

Figure 2: behavior

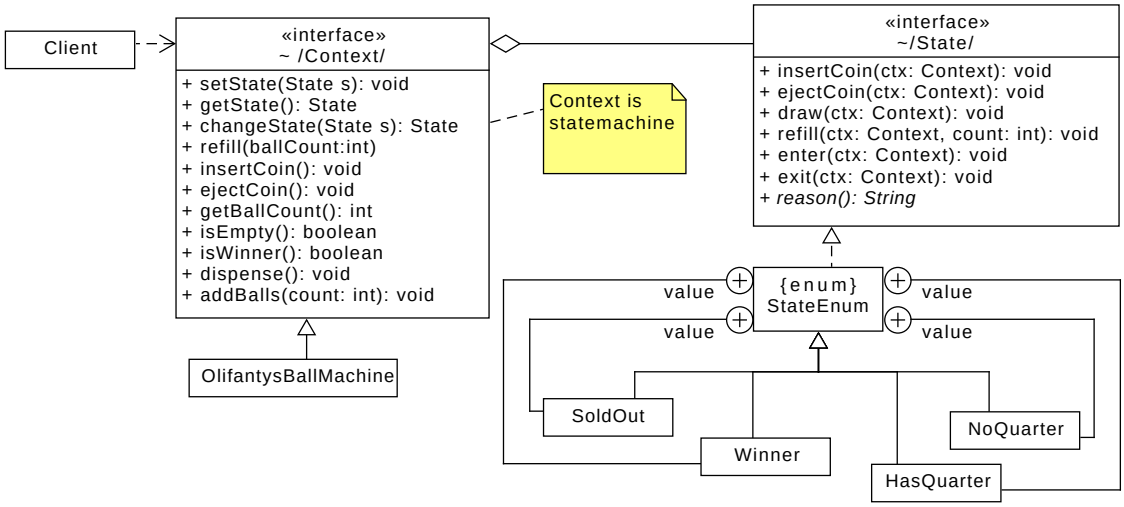

Figure 3: State classes and context