Comet Pinball

Design Model

Team Members:
Patrick Haring
Christian Bürgi

Client:
Jean-Pierre Caillot

Revision hash: 2371af6

Commit time: 2013-01-21 02:28:07 +0100

https://github.com/boskoop/comet-pinball/

Contents

List of Figures

1 Module overview

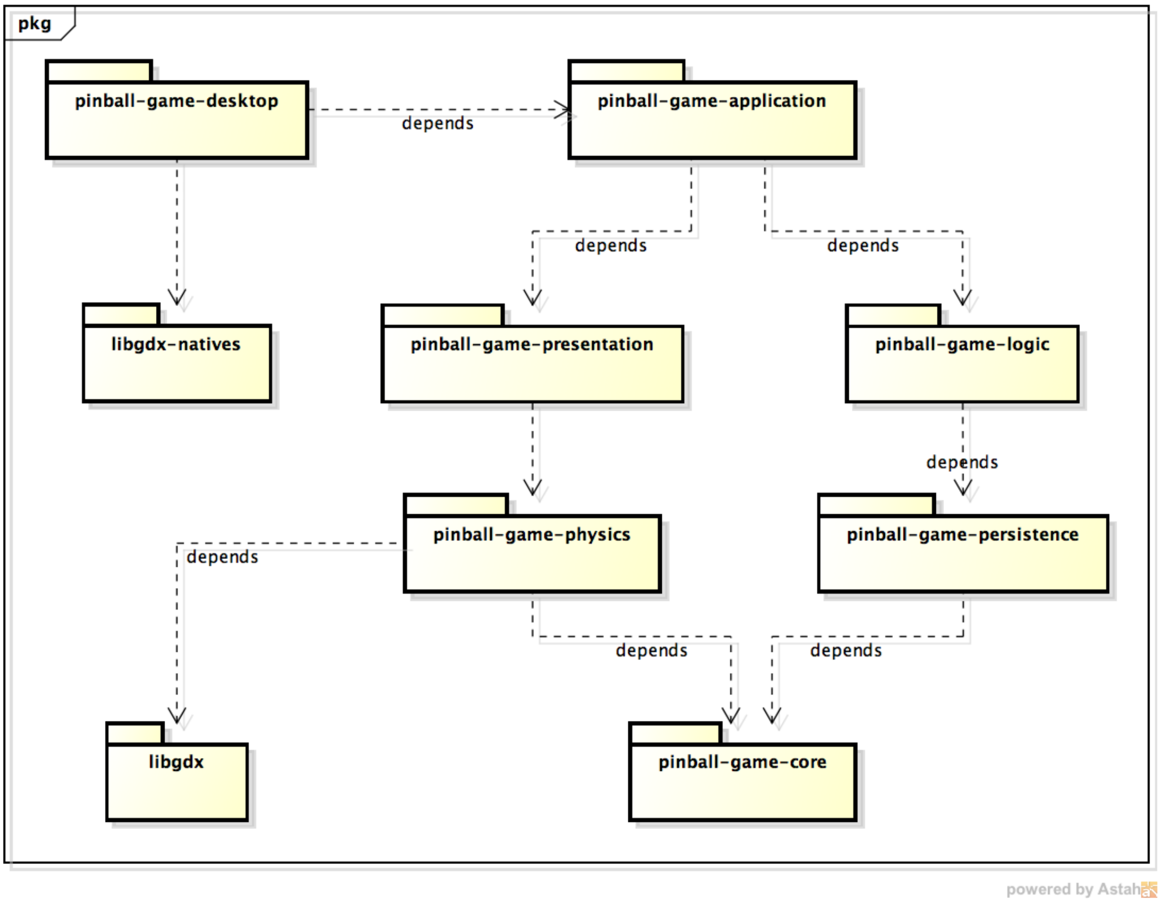

Figure 1: Module overview

Figure 1 shows the modules in the pinball application. For better understanding, it contains also the modules *libgdx* and *libgdx-natives*.

The main modules are divided into a presentation- and a logic-branch.

1.1 pinball-game-desktop

This module contains the bootstrapping code (main method) for the desktop version of the application. The dependency to the *libgdx-natives* includes the necessary native code in order to run on a desktop. The other parts of the application only depends on the *libgdx* module, which is basically a java api for all the native code in *libgdx-natives*.

1.2 pinball-game-application

This module contains the platform-independent implementation of the initialising and configuration of the application. It wires the presentation together with the logic in the dependency injection container.

1.3 pinball-game-core

This module is the base for all other modules. It defines the communication between the presentation and logic via interfaces.

1.4 pinball-game-presentation

This module's main responsibility is to render the screens in the OpenGL main loop.

1.5 pinball-game-physics

This module's main responsibility is to calculate the physics in the game.

1.6 pinball-game-logic

This module's main responsibility is to implement the game's logic an control the process.

1.7 pinball-game-persistence

This module's main responsibility is to handle access to persistent data, such as the play fields and saved simulations.

2 Persistence

2.1 Play field

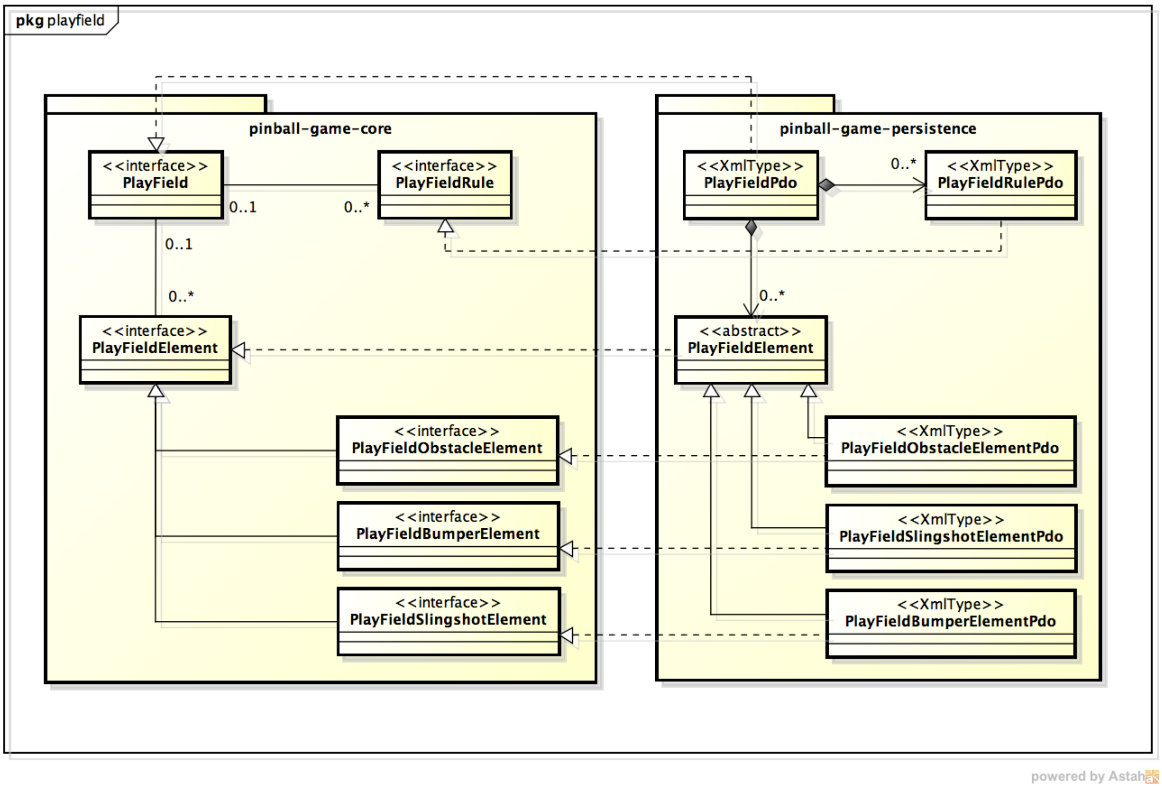

Figure 2: Play field persistence

Figure 2 shows the model for play fields. The play field is loaded from the XML file playfields.xml using JAXB. In the module *pinball-game-core* are the interfaces used to describe the entities which are crucial for the persistence of the data concerning a play field.

A `PlayField` can have multiple `Rules` and it can have multiple `PlayFieldElements`. A `PlayFieldElement` is either a `PlayFieldObstacleElement`, a `PlayFieldBumperElement` or a `PlayFieldSlingshotElement`.

In the module *pinball-game-persistence* are the concrete implementations of these interfaces which are XMLTypes and have JAXB annotations which helps JAXB to insert the data of the XML file at the right place.

2.1.1 Load playfields via JAXB

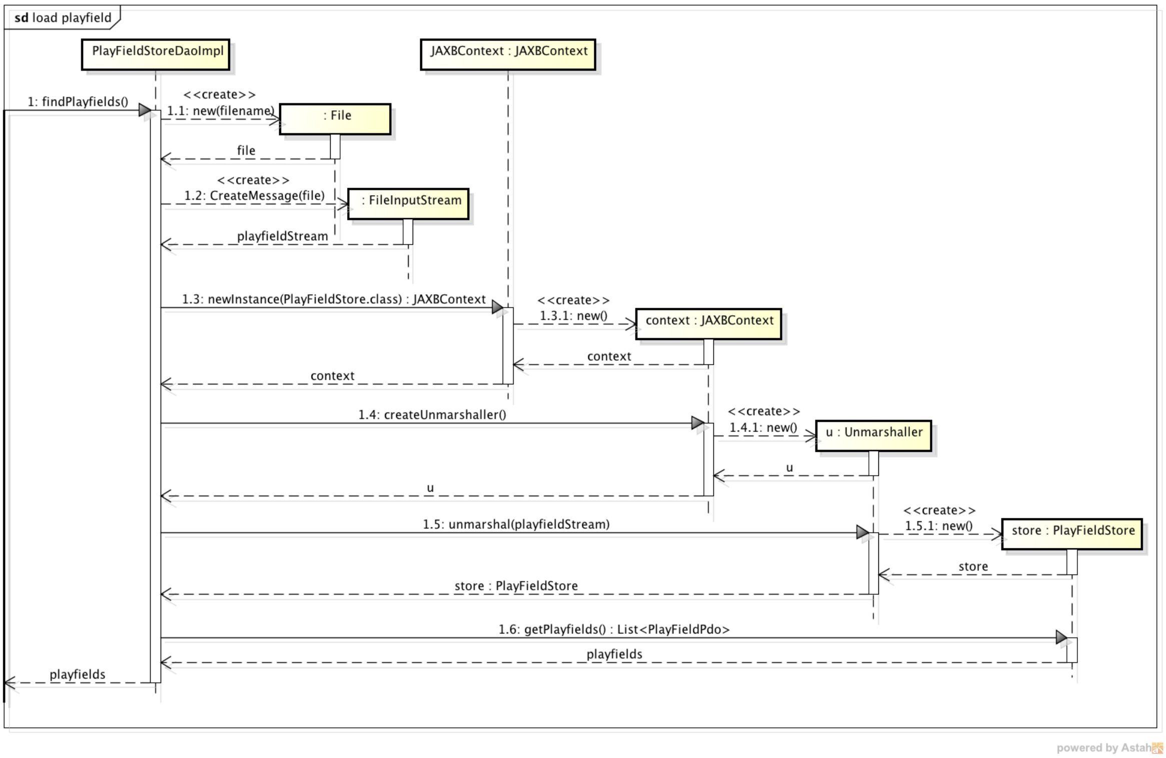

Figure 3: Load play field with JAXB

Figure 3 shows how the *playfields.xml* is loaded and the `PlayFieldPdos` are generated.

We need to bind the `JAXBContext` to the class we want to unmarshal. Then we are able to create a new `Unmarshaller` which can unmarshal the *playfields.xml* into an object of type `PlayFieldStore`. This object has a method getPlayfields which returns a List of `PlayFieldPdo`s.

2.1.2 XML Schema

The following XML schema describes the playfields.xml and is used to validate the XML file and intelligent XML editors may help the user with code completion if a XML schema is given. So the users is automatically warned if some inputs were incorrect.

Listing 1: playfield.xsd

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" targetNamespace="http://comet.m02.ch/pinball/
    playfield" xmlns:tns="http://comet.m02.ch/pinball/playfield" xmlns:xs="
    http://www.w3.org/2001/XMLSchema">
```

```
<xs:element name="configuration" type="tns:configuration"/>

<xs:complexType name="configuration">
  <xs:sequence>
    <xs:element name="playfields" form="qualified">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="playfield" type="tns:playfield" form="
              qualified" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="playfield">
  <xs:sequence>
    <xs:element name="name" type="xs:string" form="qualified"/>
    <xs:element name="elements" form="qualified">
      <xs:complexType>
        <xs:sequence>
          <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element name="bumper" type="tns:bumper" form="qualified"
                />
            <xs:element name="slingshot" type="tns:slingshot" form="
                qualified"/>
            <xs:element name="obstacle" type="tns:obstacle" form="
                qualified"/>
          </xs:choice>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="rules" form="qualified">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="rule" type="tns:rule" form="qualified"
              maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="bumper">
  <xs:complexContent>
    <xs:extension base="tns:element">
      <xs:sequence>
        <xs:element name="radius" type="xs:float" form="qualified"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
```

```
</xs:complexType>

<xs:complexType name="element" abstract="true">
  <xs:sequence>
    <xs:element name="id" type="xs:int" form="qualified"/>
    <xs:element name="position" type="tns:vector" form="qualified"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="vector">
  <xs:sequence>
    <xs:element name="x" type="xs:float" form="qualified"/>
    <xs:element name="y" type="xs:float" form="qualified"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="slingshot">
  <xs:complexContent>
    <xs:extension base="tns:element">
      <xs:sequence>
        <xs:element name="corner.a" type="tns:vector" form="qualified"/>
        <xs:element name="corner.b" type="tns:vector" form="qualified"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="obstacle">
  <xs:complexContent>
    <xs:extension base="tns:element">
      <xs:sequence>
        <xs:element name="vertices" form="qualified">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="vertice" type="tns:vector" form="
                  qualified" maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="rule">
  <xs:sequence>
    <xs:element name="class" type="xs:string" form="qualified"/>
    <xs:element name="parameters" form="qualified">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="parameter" type="xs:int" form="qualified"
              maxOccurs="unbounded"/>
```

```
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
</xs:schema>
```

2.2 Simulation

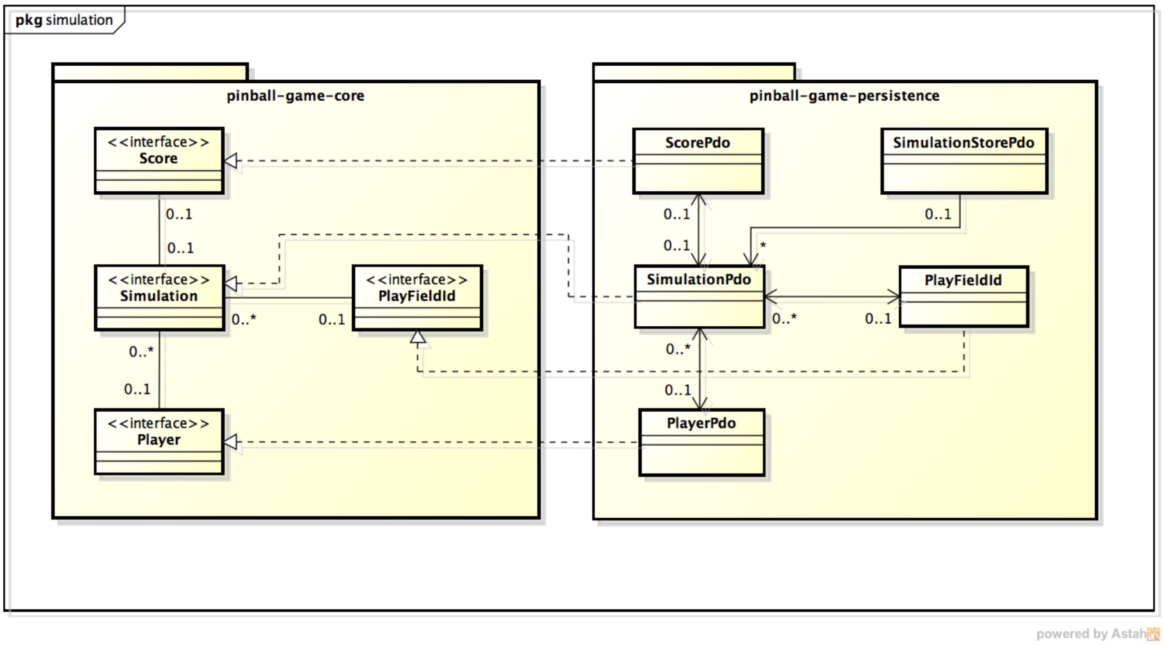

Figure 4: Simulation persistence

Figure 4 shows the data model of the data which is generated during a simulation. This data is stored in the easiest way java offers because it's not necessary that humans ever directly read the data from the generated file.

In the module *pinball-game-core* are the interfaces of the concerned entities. Each `Simulation` can have a `Player`, a `Score` and a `PlayFiedId`. The `PlayFieldId` is used to reference the play field the simulation was situated on. The implementation of these interfaces are situated in the module *pinball-game-persistence*.

3 Dependency injection

3.1 Why dependency injection?

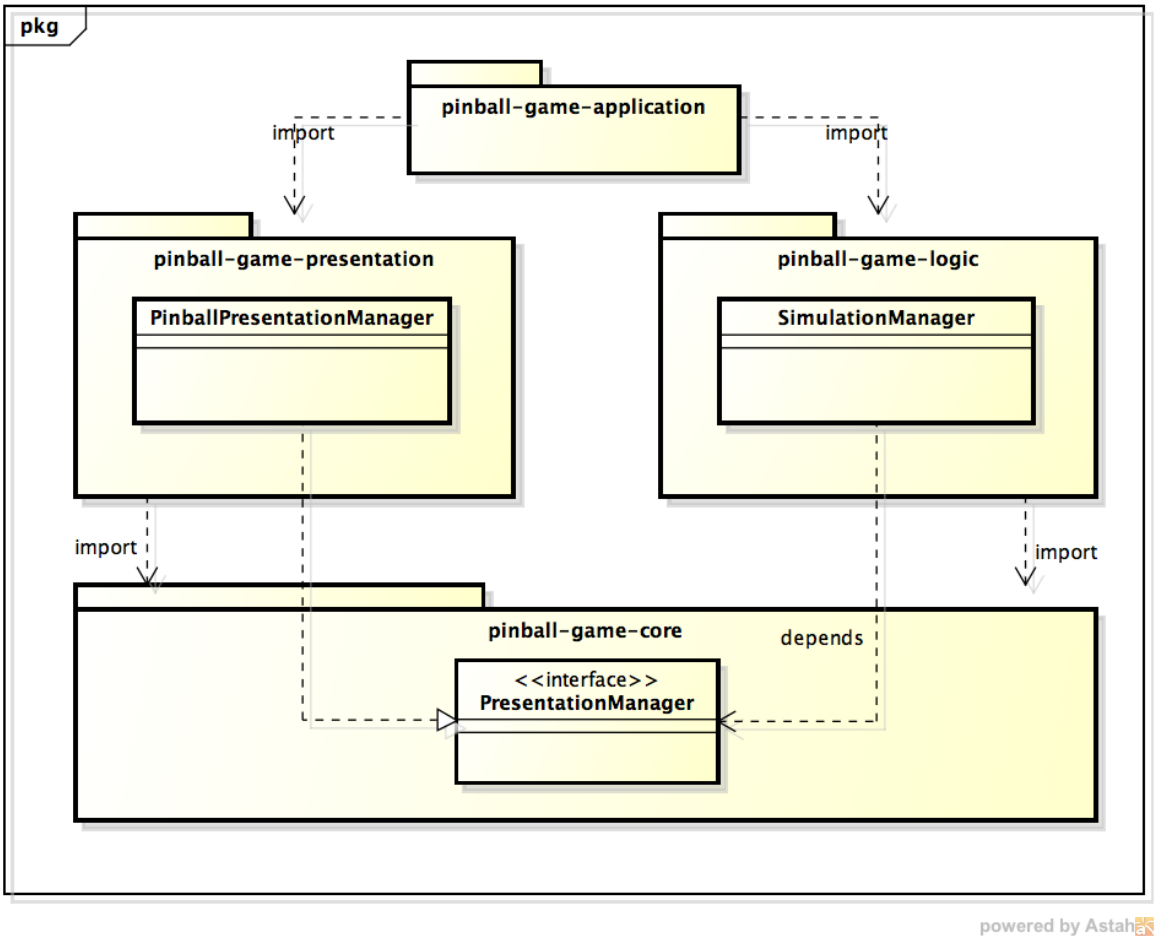

Figure 5: System design using interfaces

Figure 5 shows how interfaces are used to reference dependencies between *logic* and *presentation* code. The problem here is the following: How to get a fully initialised instance of `SimulationManager`? Only code in the package *pinball-game-application* can instantiate instances of both `SimulationManager` and `PinballPresentationManager`.

Listing 2: no dependency injection

```
public class Pinball {
  // other code...

  public void needsSimulationManager() {
    PresentationManager presentation = new PinballPresentationManager();
```

```
    SimulationManager simulation = new SimulationManager(presentation);

    // do things with SimulationManager
  }
}
```

3.2 Component wiring

In order to resolve this, we used the dependency injection framework *PicoContainer*, which takes care of the lifecycle and instantiation of components in the application which have dependencies across modules.

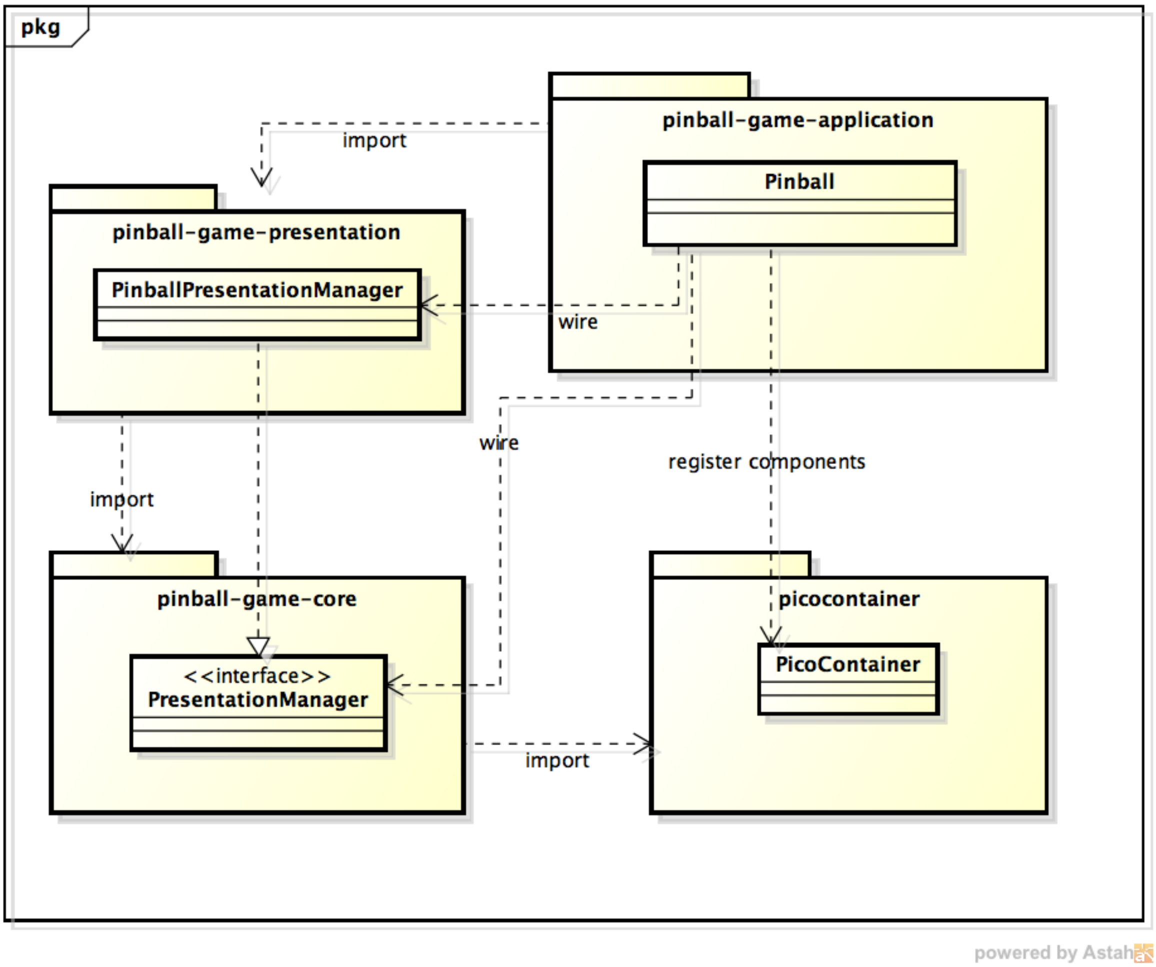

Figure 6: Component wiring

Figure 6 shows how we wire `PresentationManager` and `PinballPresentationManager` together and register them in a `PicoContainer` instance. This is done using the following code snipped:

Listing 3: dependency injection

```
public class Pinball {
  private MutablePicoContainer container;

  public Pinball(MainApplication application) {
    this.application = application;
    container = new DefaultPicoContainer();
    registerComponents();
  }

  private void registerComponents() {
    log.debug("Registering pico components");
    container.addComponent(PresentationManager.class,
        PinballPresentationManager.class);
    container.addComponent(SimulationManager.class);
    // and so on
  }
}
```

Thereby it is afterwards possible to get an initialised instance of `SimulationManager` using the following code:

Listing 4: get initialised component

```
public class AnyWhere {

  @Inject
  PicoContainer context;

  public void needsSimulationManager() {
    // PicoContainer injects PresentationManager into SimulationManager
    SimulationManager s = context.getComponent(SimulationManager.class);

    // do things with SimulationManager
  }
}
```

Figure 7 shows what happens in a sequence diagram.

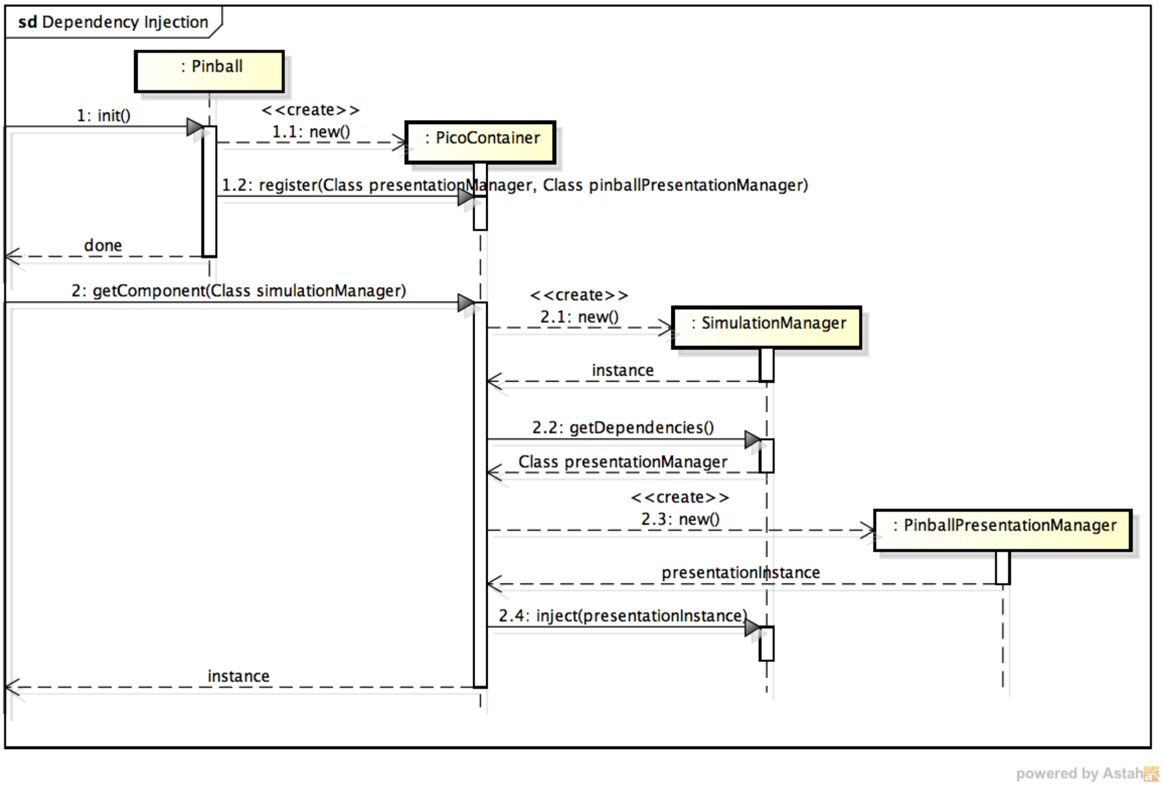

Figure 7: Dependency injection

4 Communication between modules

4.1 Command pattern

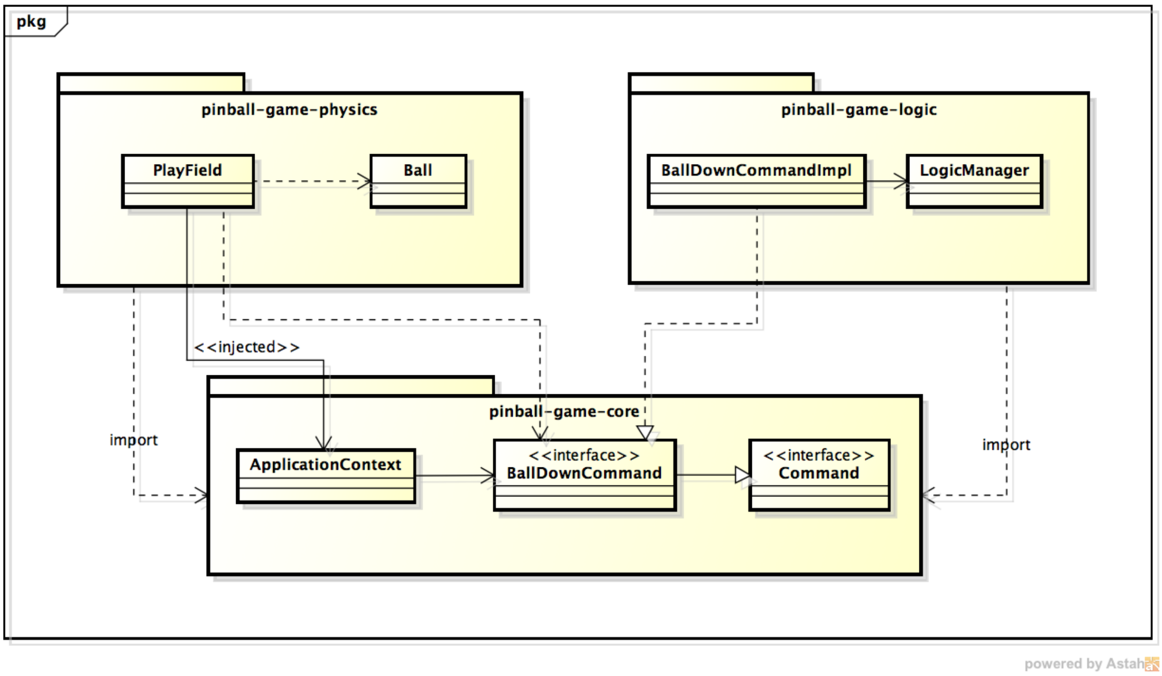

Figure 8: Command pattern

This diagram shows how the communication between the modules is implemented with a command pattern. If the physics wants to trigger an event in the game logic, it lets the *pico container* create a new implementation of the concerning command interface. The *pico container* injects the *LogicManager* so the only thing left to to in the physics module is calling the method execute of the generated command object.

5 State machine

5.1 States and transitions

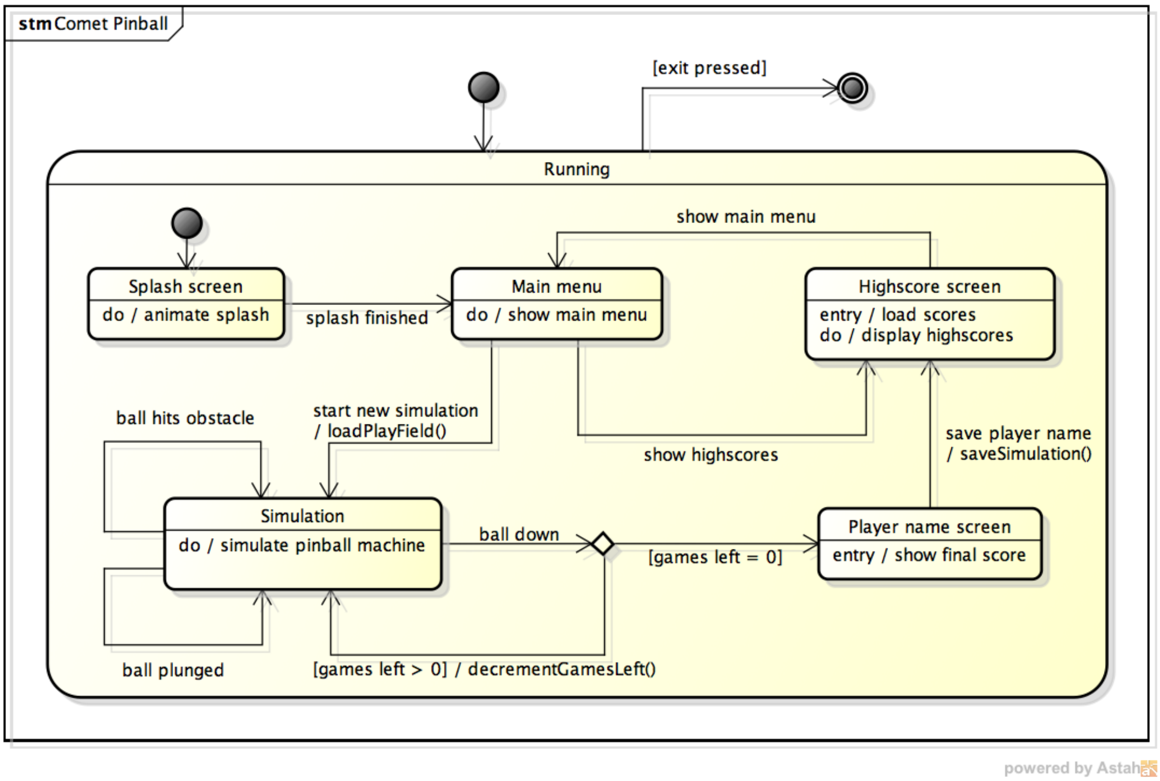

Figure 9: Pinball state machine

Figure 9 shows the states the game can be in and the transitions in between. Each *State* represents a screen in the application and each transition is triggered by an user interaction. This model is easily convertible to a state pattern implementation and very modular and thereby easy to extend.

5.2 State pattern

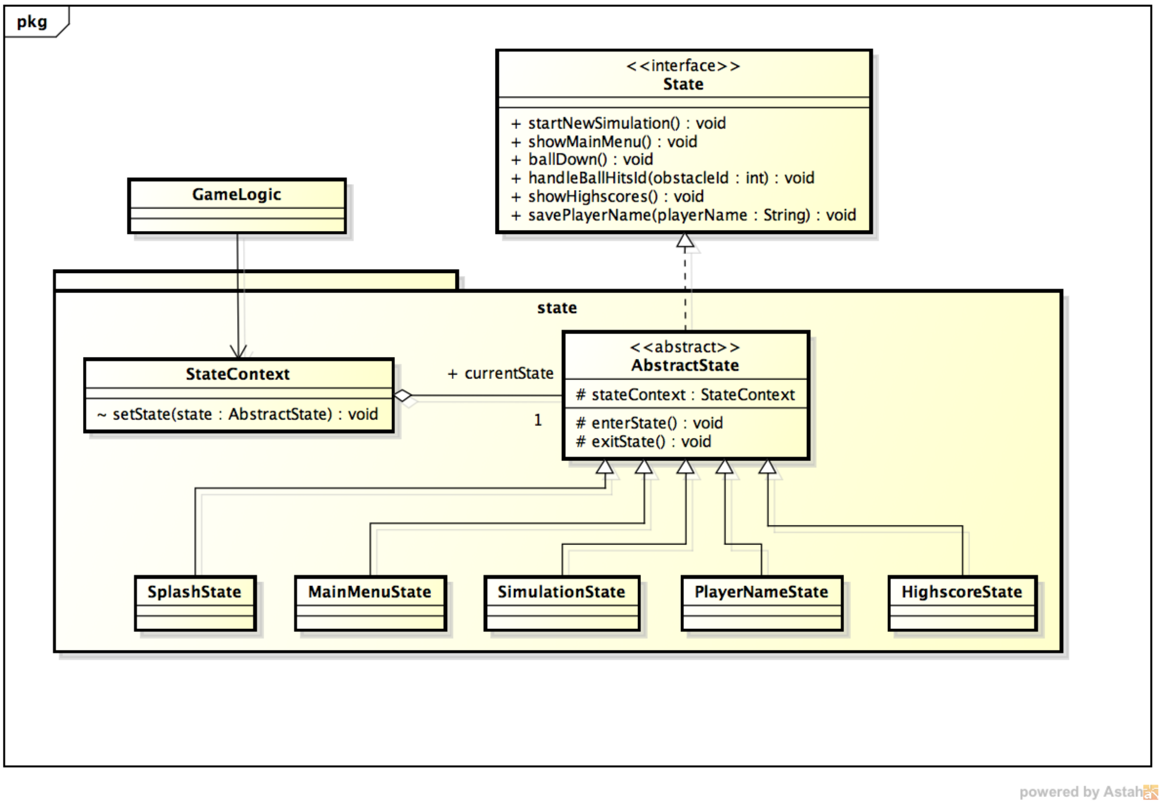

Figure 10: State pattern

Figure 10 shows how the state pattern is implemented. The interface `State` is implemented by each `State` implementation. The abstract class `AbstractState` is just for comfort: each method of the interface is implemented so it does nothing in this class. In consequence, the concrete implementation of the interface `State` does only need to implement these methods which are intended to do something. The methods `enterState()` and `exitState()` when a transition occurs. The `StateContext` is unique and handles the transition via the call of the `setState()` method.