Determining responsibilities

What object should be responsible for what behavior?

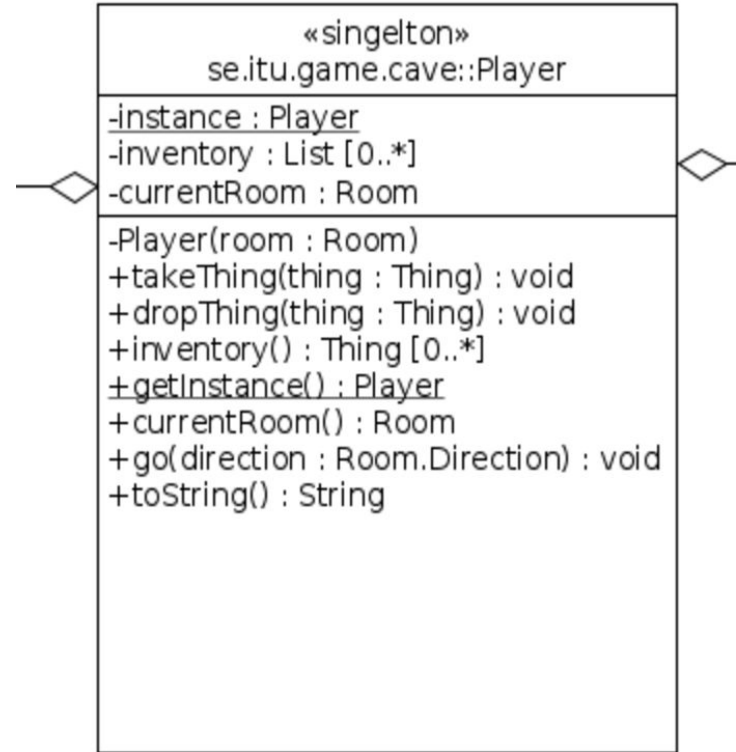
This is a design discussion for Sprint 2 of the Cave game

Analysing and designing classes

In Sprint 1, we designed the Player class like this:

The behavior/responsibilities of the Player was:

- takeThing
- dropThing
- reveal its inventory
- reveal its (current) Room
- go in some direction



Player’s responsibilities - impact on client code

We’re going to look at how this design impacts the GUI code.

The GUI needs to display the Player’s current Room’s description, as well as the Player’s current Room’s things.

Look at that phrase: “Player’s current Room’s description”

- The GUI must know what a Room is
- The GUI must get the Room from the Player
- Then GUI must get the description from the Room

This is a level of indirection - GUI -> Player -> Room -> Description

Player’s responsibilities - impact on client code

Is it intuitive for us that a Player is able to relay a whole Room?

We could argue: “If a Player is able to pass on an entire Room, the Player surely could pass on what he sees in the Room?”

The Player object isn’t blind! Of course the Player object can view the current Room’s description and list of Thing:s! So why not let the Player be responsible for keeping track of not only the current Room, but also that Room’s description and Thing:s?

Player’s responsibilities - impact on client code

Let’s make the Player more competent, and see what impact that has on the GUI code!

Player’s new responsibilities

- Describe the Room currently standing in
- Relay the Thing:s of the Room currently standing in

UML syntax:

```
+describeCurrentRoom() : String

+thingsInCurrentRoom() : Thing [*]
```

Player’s new responsibilities

- Describe the Room currently standing in
- Relay the Thing:s of the Room currently standing in

Java syntax:

```
public String describeCurrentRoom() {
  return currentRoom.description();
}

public List<Thing> thingsInCurrentRoom() {
  return currentRoom.things();
}
```

Player’s new responsibilities - Impact on GUI code

For the GUI (or any kind of UI) to get the String with the Player’s current Room’s description or the Thing:s in the Room, the syntax now becomes:

```
String currentRoomDescription =
    Player.getInstance().describeCurrentRoom();

List<Thing> currentRoomThings =
    Player.getInstance().thingsInCurrentRoom();
```

What happened here, is that the (G)UI code is no longer dependant on the Room class!

What about the possible directions from a Room?

We decided earlier that the (G)UI should check legal directions for navigation and disable the buttons for directions which don’t have any connecting Room, so that the user playing the game can’t try to go in a direction without a Room.

So, how should the (G)UI discover what directions are possible, then?

If the (G)UI don’t know about the Room class, this too becomes a responsibility of the Player class.

What about the possible directions from a Room?

Isn't it feasible to say that the Player not only can see and relay the description and Thing:s of the Room, but also see what directions have exits?

Let's add this new responsibility to the Player. UML:

```
+canSeeDoorIn(direction : Room.Direction) : boolean
```

Java:

```
public boolean canSeeDoorIn(Room.Direction direction) {
  return currentRoom.getRoom(direction) != null;
}
```

Impact on client code (UI code)

```
private Map<Room.Direction, JButton> buttonMap;
……..

/* A map with directions as keys and the nav buttons as values
 * used for looping through the buttons and enable/disable them
 */
buttonMap = new HashMap<>();
buttonMap.put(Direction.NORTH, northButton);
buttonMap.put(Direction.SOUTH, southButton);
buttonMap.put(Direction.EAST, eastButton);
buttonMap.put(Direction.WEST, westButton);
```

Impact on client code (UI code)

The syntax for determining whether to enable or disable a navigation button now becomes (if we have a Map<Room.Direction, JButton>):

```
private void updateButtons() {
  for (Direction dir : Direction.values()) {
    buttonMap.get(dir)
        .setEnabled(Player.getInstance()
                           .canSeeDoorIn(dir));
  }
}
```

UML for the new responsibilities

```
+describeCurrentRoom() : String

+thingsInCurrentRoom() : Thing [*]

+canSeeDoorIn(direction : Room.Direction) : boolean
```

Benefits from this new Design

We have insulated the (G)UI class from changes in the Room class.

Since we have decoupled the (G)UI from the Room, the (G)UI is not affected by changes in the design (or implementation) in the Room class.

Instead of having to know how to query a room for its things, description and connecting rooms, the (G)UI now relies only on the Player to handle or “know” this information.

The syntax becomes more concise as a bonus side-effect.

Contrasting syntax before and after

Before

```
String currentRoomDescription =
   Player.getInstance()
         .getCurrentRoom()
         .description();
```

After

```
String currentRoomDescription =
   Player.getInstance()
         .describeCurrentRoom();
```

Contrasting syntax before and after

Before

```
List<Thing> roomThings =
    Player.getInstance()
          .currentRoom()
          .things();
```

After

```
List<Thing> roomThings =
    Player.getInstance()
          .thingsInCurrentRoom();
```

Contrasting syntax before and after

Before

```
boolean hasConnectingRoom =
    (Player.getInstance()
           .currentRoom()
           .getRoom(NORTH))
    != null;
```

After

```
boolean hasConnectingRoom =
    Player.getInstance()
          .canSeeRoomIn(NORTH);
```

Discussion and questions

- Do you have any questions regarding the new responsibilities?
- Do you agree with the suggested re-design?
- Which design lead to the cleanest (shortest and/or easiest to read) code?
- Shall we agree on the suggested re-design?
    - If not, what do you propose instead?

Name-dropping time

We’ve actually followed the Law of Demeter, which states:

- *Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.*
- *Each unit should only talk to its friends; don't talk to strangers.*
- *Only talk to your immediate friends.*

A method m of an object O may only invoke the methods of the following kinds of objects:

- *O itself*
- *m's parameters*
- *Any objects created/instantiated within m*
- *O's direct component objects*
- *A global variable, accessible by O, in the scope of m*

What does the legal system say about violations?

The Academic courts of theoretical programming will have to follow the Swedish law which says about violations of the law of demeter:

Framkallar någon allmän fara för djur eller växter genom att öka coupling mellan klasser eller genom att sprida kod utan UML eller på annat dylikt sätt, dömes för förgöring till böter eller fängelse i högst två år.

Är brottet grovt, skall dömas till fängelse, lägst sex månader och högst sex år. Vid bedömande huruvida brottet är grovt skall särskilt beaktas, om det skett med uppsåt att skada eller om egendom av betydande värde utsatts för fara.

Name-dropping time

We’ve achieved “Lower coupling” between objects and classes.