Implementing Rules

Java rules ok

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

What types of Rule:s exist in the Game?

After analysing the old specification, it seems that there are two main types of Rule:s in the game:

- ThingRule:s - Rules about picking up Thing:s
- RoomRule:s - Rules about what happens with the Room

ThingRule:s identified

Bird:

- Affected by the Rod (you must put it down in order to pick up the Bird)
- Can’t be picked up, unless we have the Cage in our inventory

Pirate Chest

- Can’t be picked up (opened) unless we have the four keys

RoomRule:s identified

Snake room:

- Snake means no SOUTH exit
- Snake disappears if we drop down Cage and Bird
- Without the Snake, SOUTH room is unblocked (and sets to a new Room)

Dragon room:

- Dragon keeps Glass Key from appearing
- Dragon disappears if we drop down Gold, Jewels, Diamonds and Silver
- Without the Dragon, WEST exit disappears

Bird room: The bird gets scared if we have the Rod in our inventory

Game over Rooms

Two Room:s will “kill the Player” (Game Over)

We suggest that we make these Room:s special, when reading them from the Database - The description of theses Room:s should describe that the Player dies and it’s Game Over.

This could be accomplished as simply as setting all exits from this Room to null (meaning there’s no way out of there) - The User playing the game can’t continue, since all navigation buttons are disabled.

ThingRule

What behavior is affected by the ThingRules?

Look at the descriptions again. The verb we’re after is “pick up”.

ThingRule

What behavior is affected by the ThingRules?

Look at the descriptions again. The verb we’re after is “pick up”.

The behavior affected is the Player’s takeThing(thing) method.

ThingRule

So, we need a lot of IF-statements in the takeThing(thing) method?

ThingRule

So, we need a lot of IF-statements in the takeThing(thing) method?

Not necessarily. What we need, is to check the ThingRule before completing the operation of taking a thing.

If the ThingRule decides it is allowed to pick up the Thing, we'll procede as usual (add to inventory, remove from room). Otherwise we'll throw an exception (so that the GUI, and therefore the User playing, notices that a rule was violated.

ThingRule

How can we implement the ThingRule?

Requirements/dependencies:

- There are more than one ThingRule
- The ThingRule depends on the Thing to be picked up and the Player’s inventory
- We don’t want to hard code rules using if statements
- The Player “doesn’t know the Rules”, they have to be looked up

Possible solution - The Things carry the rules

One way of implementing this would be that the Thing:s have a rule connected to them.

Feels intuitive?

Possible solution - The Player carries the rules

Another way of implementing this would be that the Player have a rules for the Thing:s.

Feels intuitive? When Playing the game, the Player can’t know the Rules - they constitute part of what makes the game harder to solve.

Possible solution - The (G)UI carries the rules

Another way of implementing this would be that the (G)UI have a rules for the Thing:s.

Feels intuitive? But what if we have more than one UI for the game?

Should each type and instance of a UI have a list of ThingRule:s?

Possible solution - There’s some kind of rule book

Another way of implementing this would be that there’s some kind of RuleBook to query when picking up Thing:s.

This could be implemented as a static method which returns some kind of result with permit/prevent and a message.

Another way could be that the RuleBook returns a ThingRule for a particular Thing (provided as an argument when querying).

Investigating - There's some kind of rule book

Let's play with the idea that the RuleBook exists and can return a ThingRule provided we give it a Thing:

+getRuleFor(thing : Thing) : ThingRule

As you see (underline!), the method is a static method.

The RuleBook must have some kind of internal map between Thing and ThingRule, so that if we do this:

```
ThingRule rule = RuleBook.getRuleFor(thing);
```

then we get the correct rule for the provided thing.

Investigating - There's some kind of rule book

Let's play with the idea that the RuleBook exists and can return a ThingRule provided we give it a Thing:

+getRuleFor(thing : Thing) : ThingRule

But most things don't have any rules for picking them up. What would the method return for e.g. a Key?

Well, a rule which permits picking it up, of course!

Investigating - There's some kind of rule book

Let's play with the idea that the RuleBook exists and can return a ThingRule provided we give it a Thing:

+getRuleFor(thing : Thing) : ThingRule

So, what is a ThingRule, then?

Investigating - There's some kind of rule book

Let's play with the idea that the RuleBook exists and can return a ThingRule provided we give it a Thing:

+getRuleFor(thing : Thing) : ThingRule

So, what is a ThingRule, then?

An object which can either permit picking the Thing up, or convey some kind of disruption with a message if it doesn't permit it.

Investigating - There's some kind of rule book

The behavior of the ThingRule objects could be this:

```
+apply() : void { exceptions=RuleViolationException }
```

If RuleViolationException is a checked exception, the Player's takeThing() method must handle it (or pass it on the the UI).

Investigating - There's some kind of rule book

The behavior of the ThingRule objects could be this:

+apply() : void { exceptions=RuleViolationException }

So how do we create all the ThingRule objects, then, so that they have different behavior (rules) for different Thing:s?

Let's look at ThingRule from the Player's perspective!

Investigating - ThingRule from the Player's view

The Player, in the takeThing(Thing thing) method, must query the RuleBook if the Thing to be taken is allowed. If not, an Exception will be thrown.

The method doesn't know what the Thing is (it is not hard coded, but rather just a parameter to takeThing(Thing thing).

The Player doesn't want to or need to know what actual class the ThingRule is, just how to query it for applying the rule for the Thing.

How can we write code which operates on a number of classes sharing a common method?

Investigating - ThingRule from the Player's view

The ThingRule could be a Java Interface, declaring just one simple method:

`+apply() : void {exceptions=RuleViolationException}`

(note the *italics* - what do you think it means?)

Now, the `+getRuleFor(thing : Thing) : ThingRule` method will simply return *some object* of *some class implementing ThingRule*.

This object could be created as a local inner anonymous class or a Java 8 lambda expression.

Investigating - ThingRule from the Player's view

The ThingRule could be a Java Interface, declaring just one simple method:

+apply() : void {exceptions=RuleViolationException}

Knowing that ThingRule is just an interface with just the above method, the takeThing(Thing thing) method could do this:

```
public void takeThing(Thing thing) throws RuleViolationException {
  RuleBook.getRuleFor(thing).apply(); // here an exception might happen
  currentRoom.removeThing(thing);
  inventory.add(thing);
}
```

Investigating - ThingRule from the UI's view

The ThingRule could be a Java Interface, declaring just one simple method:

+apply() : void {exceptions=RuleViolationException}

The (G)UI is of course mandated by the checked exception to handle it:

```
Thing thing = ((JList<Thing>)event.getSource())
    .getSelectedValue();
try {
  Player.getInstance().takeThing(thing);
  updateModels(); // make the UI discover the change and
                  // fix the UI lists (inventory and room things)
} catch (RuleViolationException e) {
  messages.setText(e.getMessage()); // notify the User of the denial
}
```

RuleViolationException

Inherit Exception and create a constructor which takes a String message, call super(message) in the constructor;

RuleBook

Create a `static Map<Thing, ThingRule>`. We'll populate the map from the `CaveInitializer` code, so that all the `Thing`:s with rules are in the map.

Make a method, `static public ThingRule getRuleFor(Thing thing)` which

- Looks up the `Thing` in the map and
  - if it exists, return the corresponding `ThingRule`
  - otherwise return a new `ThingRule` which doesn't throw an exception

Alternative approach for apply()

We could make the apply() method of ThingRule return a boolean, to make the code in Player more clear:

```
public void takeThing(Thing thing) throws RuleViolationException {
  if(RuleBook.getRuleFor(thing).apply()) {
    currentRoom.removeThing(thing);
    inventory.add(thing);
  }
}
// ThingRule interface:
+apply() : boolean {exceptions=RuleViolationException}
```

Alternative approach for apply()

Our code, which populates the map in RuleBook could create a rule for a Thing with a rule (bird requires cage) like this:

```
RuleBook.addThingRule(thing, () ->
    {
      if(! Player.getInstance()
                .inventory()
                .contains(Things.get("Cage")) ) {

        throw new RuleViolationException("Bird requires a Cage");
      }
      return true;
    });
```

Alternative approach for apply()

Our code, which populates the map in RuleBook could create a rule for a Thing without a rule like this:

```
RuleBook.addThingRule(thing, () ->
    {
      return true;
    });
```

When applying that rule, apply() simply returns true and no exception is thrown.

You don’t have to worry about this code, we’ll provide it in CaveInitializer.

Things without rules

Or, better still...

Let the getRuleFor(thing) method of RuleBook return a rule which just allows if it can't find any rule for the thing asked for:

```
public static getRuleFor(Thing thing) {
  ThingRule rule = rules.get(thing);
  if(rule == null) {
    rule = () -> { return true; };
  }
  return rule;
}
```

Alternative approach for apply()

Let’s go for the **boolean** apply() throws RuleViolation approach.

UML updates

`Player` gets a different notation for the takeThing(Thing thing) method:

`+takeThing(thing : Thing) : void` **`{exceptions=RuleViolationException}`**

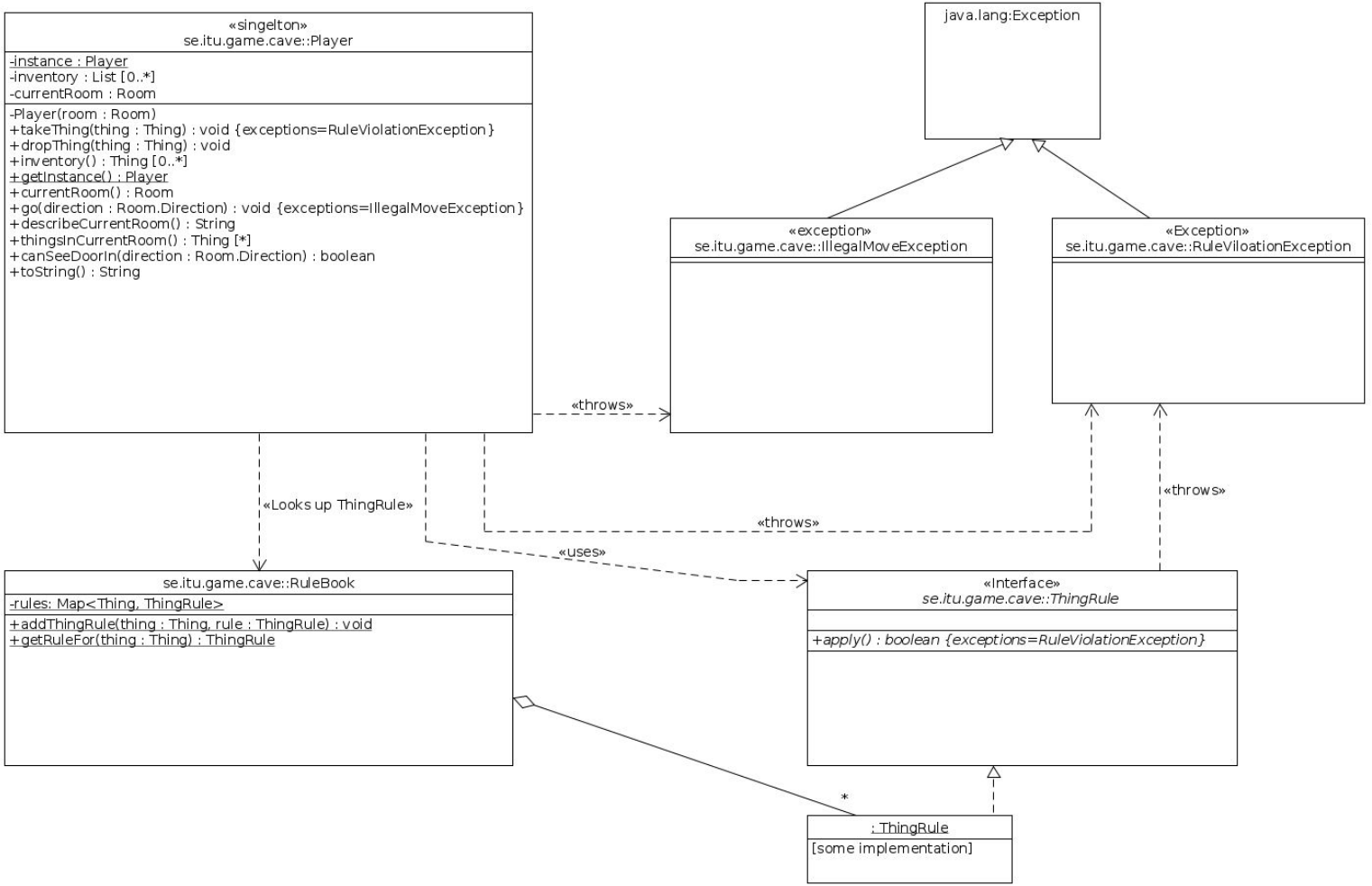
The interface ThingRule has the following:

`+apply() : boolean {exceptions=RuleViolationException}`

RuleBook:

`+getRuleFor(thing : Thing) : ThingRule`

`RuleViolationException` just extends `java.lang.Exception` and is thus a checked exception.


«singelton»
se.itu.game.cave::Player
-instance : Player
-inventory : List [0..*]
-currentRoom : Room
-Player(room : Room)
+takeThing(thing : Thing) : void {exceptions=RuleViolationException}
+dropThing(thing : Thing) : void
+inventory() : Thing [0..*]
+getInstance() : Player
+currentRoom() : Room
+go(direction : Room.Direction) : void {exceptions=IllegalMoveException}
+describeCurrentRoom() : String
+thingsInCurrentRoom() : Thing [*]
+canSeeDoorIn(direction : Room.Direction) : boolean
+toString() : String
java.lang:Exception
«exception»
se.itu.game.cave::IllegalMoveException
«Exception»
se.itu.game.cave::RuleViloationException
«throws»
«Looks up ThingRule»
«throws»
«throws»
«uses»
se.itu.game.cave::RuleBook
-rules: Map<Thing, ThingRule>
+addThingRule(thing : Thing, rule : ThingRule) : void
+getRuleFor(thing : Thing) : ThingRule
«Interface»
se.itu.game.cave::ThingRule
+apply() : boolean {exceptions=RuleViolationException}
*
: ThingRule
[some implementation]

Next sprint - RoomRule:s!

Next, we'll have to implement the RoomRule:s. We'll do that in the next sprint!