Designing the RoomRule:s

Arriving at a design

RoomRule

- What is a RoomRule?
- Where are RoomRule:s stored
- When are RoomRule:s applied/checked?
- What is the effect of applying a RoomRule?
- How and where are RoomRule:s created?

RoomRule

- What is a RoomRule?

A ***room*** can have a special creature (or some ***creature description***) like a Snake or a Dragon. The creature can be bribed with stuff from the Player's inventory (put down the cage and the bird, and the snake disappears), which should change the ***creature description***.

We need instance variables for `room` and `creatureDescription`.

RoomRules should be able to +a`pply() : void`

RoomRule

- Where are RoomRule:s stored?

Let’s use the good old RuleBook (sounds right?). How do we store a RoomRule?

RoomRule - RuleBook changes

```
+getRuleFor(room : Room) : RoomRule

// What should this method return for rooms without rules?

// Some kind of default rule:
// no description
// no effect when applied
```

RoomRule - RuleBook changes

- Where are RoomRule:s stored?

Let’s use the good old RuleBook (sounds right?). How do we store a RoomRule?

We could store it together with the Room it applies to. So the RuleBook needs methods for `put`ting and `get`ting a rule for a room. Suggestion:

```
Map<Room, RoomRule> roomRules…
+getRuleFor(room : Room) : RoomRule
+addRoomRule(room : Room, rule : RoomRule) : void
```

RoomRule

- When are RoomRule:s applied/checked?

We need to see the information about e.g. the Snake when the room is described. So one place is 1. In Player's `describeCurrentRoom()` method.

Another place to apply the rule is when the player puts down things (trying to bribe the creature for instance). So another place is 2. In Player's `dropThing(thing : Thing)` method.

Strategy here: in 1 : get the rule, get the `creatureDescription()`. In 2: get the rule, and `apply()` it after putting down the thing.

RoomRule

We already have identified:

```
---------------------------
          RoomRule
---------------------------
room : Room
creatureDescription : String
---------------------------
apply() : void
creatureDescription() : String
// Do we need a mutator for the description?
---------------------------
```

RoomRule

What access modifiers should we use?

```
---------------------------
          RoomRule
---------------------------
room : Room
creatureDescription : String
---------------------------
apply() : void
creatureDescription() : String
// Do we need a mutator for the description?
---------------------------
```

RoomRule

Is this an interface, or a class, or some kind of class?

```
---------------------------
         RoomRule
---------------------------
room : Room
creatureDescription : String
---------------------------
apply() : void
creatureDescription() : String
// Do we need a mutator for the description?
---------------------------
```

RoomRule

The behavior changes for each rule - so the methods need to be overridden.

```
---------------------------
         RoomRule
---------------------------
room : Room
creatureDescription : String
---------------------------
apply() : void      <- this one changes, make it abstract
creatureDescription() : String
// Do we need a mutator for the description?
---------------------------
```

RoomRule

Make the class abstract

```
---------------------------
         RoomRule
---------------------------
room : Room
creatureDescription : String
---------------------------
apply() : void
creatureDescription() : String
// Do we need a mutator for the description?
---------------------------
```

RoomRule

Make the class abstract with protected variables and a constructor

```
---------------------------
         RoomRule
---------------------------
#room : Room
#creatureDescription : String
---------------------------
+RoomRule(room : Room, creatureDescription : String)
apply() : void
creatureDescription() : String
// Do we need a mutator for the description?
---------------------------
```

RoomRule

- What is the effect of applying a RoomRule?

Running apply() should check if the Room contains the Thing:s needed to bribe the creature or trigger the effect. When the creature is bribed, it goes away. Sometimes a new exit is opened in the Room. Sometimes a Key magically appears in the Room.

So, apply() should use its Room reference and check it and manipulate the Room if the rule is triggered.

RoomRule

- How and where are RoomRule:s created?

The CaveInitializer takes care of this for you. The RoomRule class is abstract, so we need to extend it and override apply(). The Room and creatureDescription should be supplied when calling the constructor (via new).

You'll also need to create some kind of default room rule for most rooms, since they lack rules - in the RuleBook's getRuleFor(room) method.

RoomRule - creation

We can use an anonymous inner class for this:

```
RuleBook.addRoomRule(someRoom,
    new RoomRule(someRoom, someDescription) {
      @Override
      public void apply() {
          //use the room instance variable to check
          //use the room inst. var. to manipulate
          //use the creatureDescription inst. variable
          //   and change it if needed the snake has gone
      }
    });
```

RoomRule - creation

```
new RoomRule(someRoom, someDescription) {
  @Override
  public void apply() {
    if(room.things().contains(Things.get("Silver"))) {
      creatureDescription = "Alvsilver looks happy";
      room.setConnectingRoom(NORTH, someOtherRoom);
    }
  }
}
```

RoomRule - creation

```
new RoomRule(someRoom, someDescription) {
  @Override
  public void apply() {
    if(room.things().contains(Things.get("Silver"))) {
      creatureDescription = "Alvsilver looks happy";
      room.setConnectingRoom(NORTH, someOtherRoom);
    }
  }
}
```

Using the inherited protected variables room and creatureDescription!

RoomRule - creation - Default room rule

```
class DefaultRoomRule extends RoomRule {
  public DefaultRoomRule() {
    // call super with uninteresting values
  }
  @Override
  public void apply() { /* do nothing */ }
}

// In RuleBook, you could create a static instance of the
// above class and return it for calls to getRuleFor(room)
// which can't find any rule - e.g. rooms without rules.
```