4. Software architecture design and refactoring

Modifiability

Providing a low-coupling and high-cohesion code is one of the best ways to increase the modifiability of an application. In order to achieve this goal, the new architecture includes a lot of new classes, files and packages. The following package diagram presents the global architecture. Each layer depends on one or more lower layers. As there is no cyclic dependency, the coupling between different packages is rather low.

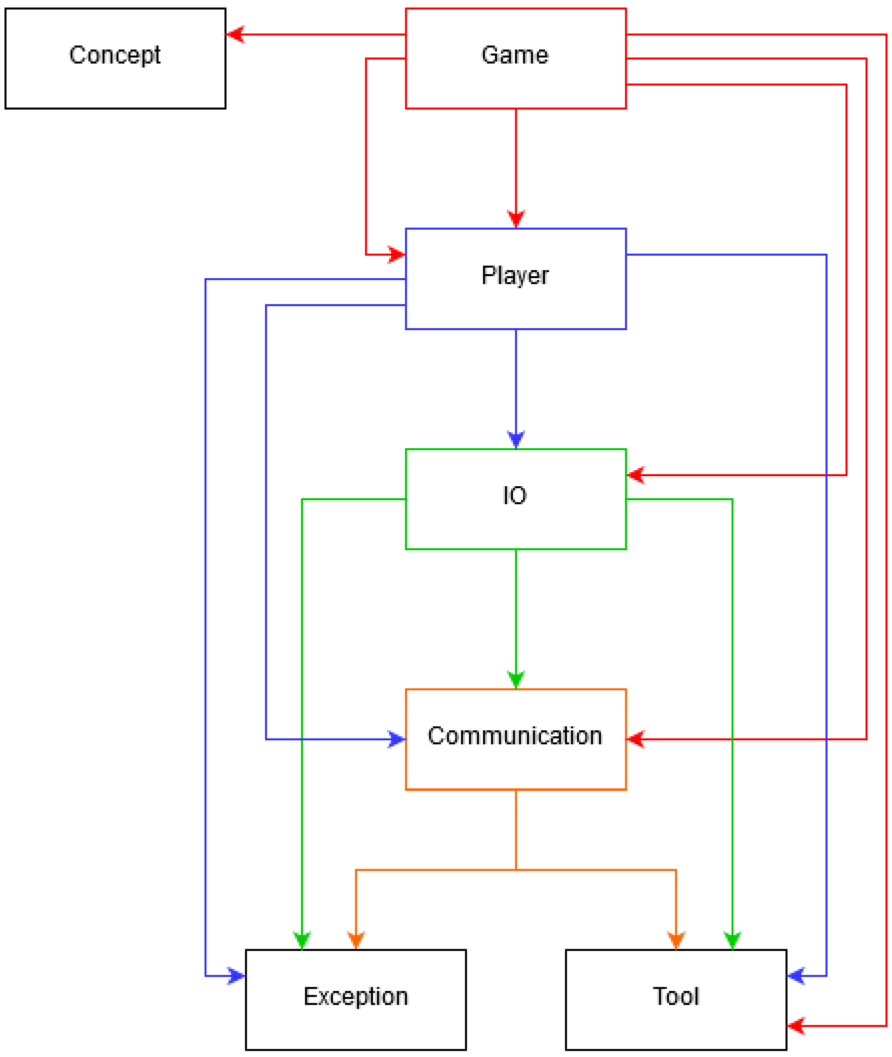


Diagram 1- Package diagram

Extensibility

In order to provide extensibility, a few mechanisms have been used.

First, monsters' characteristics, cards attributes (cost, name, …), literal strings are stored as text files (*JSON* for data and *properties* for string literals). It is therefore possible to add, remove or edit cards or monsters without modifying the code. Moreover, it provides a way to translate the whole game if a future requirement specifies it. That's probably the most important extensibility mechanism. The following excerpt gives an example of how a store card is represented.

```json
{
  "cost": 5,
  "name": "Alpha Monster",
  "description": "Gain 1 STAR when you attack.",
  "effects": [
    {
      "effect": "GainStars",
      "value": 1,
      "targets": [
        "Self"
      ],
      "triggeringActions": [
        "DealDamages"
      ]
    }
  ],
  "type": "Keep"
}
```

Representation 1- Store card

Second, the *IO* package provides easy-to-use interfaces to work with inputs and outputs. It would then be easy to add new human-computer interactions, such as a graphical interface. It currently supports a console input/output, a remote input/output and an *InputString* which is basically a list of inputs that replaces manual inputs.

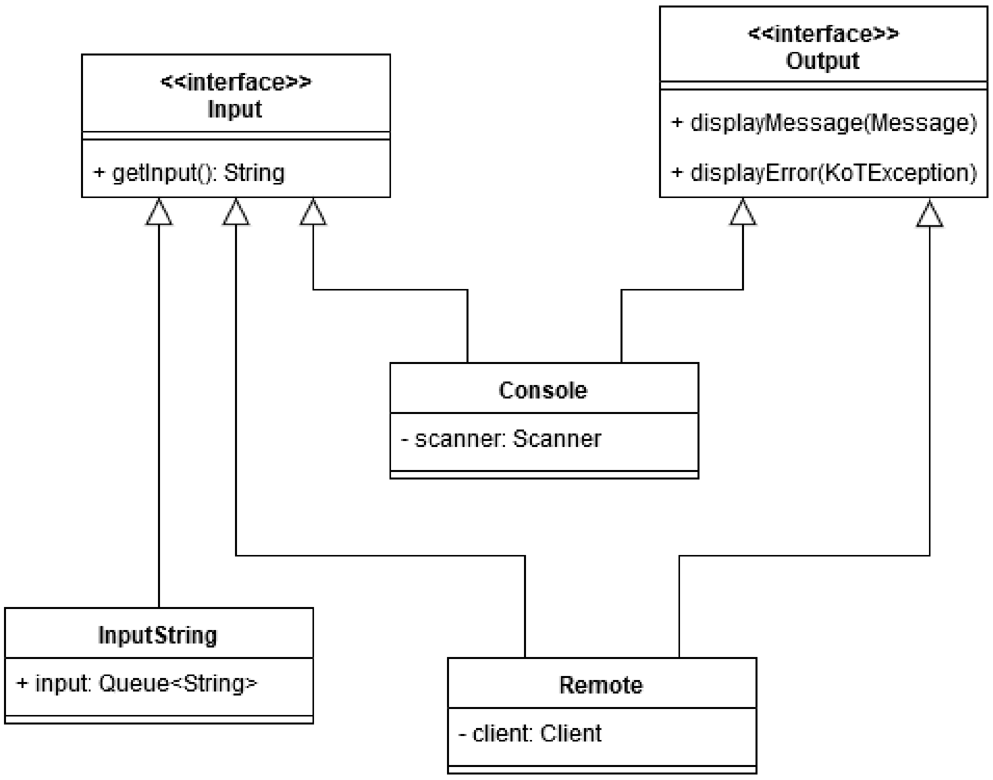


Diagram 2- IO class diagram

Third, the *communication* package holds a *message* subpackage. This package provides an abstract class *Message* used to communicate between the server and the clients. The *Message* class is inherited in 3 types of message:

- *Information*: any text that doesn't need any reply;
- *Reply*: any reply to a question;
- *Question*: a question to the player, itself inherited:
  - *ClosedQuestion*: question with a limited amount of possible choices;
  - *OpenQuestion*: question which accepts any literal choice.

In order to work with these classes, a Factory design pattern has been used: it enables to create different messages without worrying about the actual return type. This mechanism provides an extensible way to communicate between remote clients.

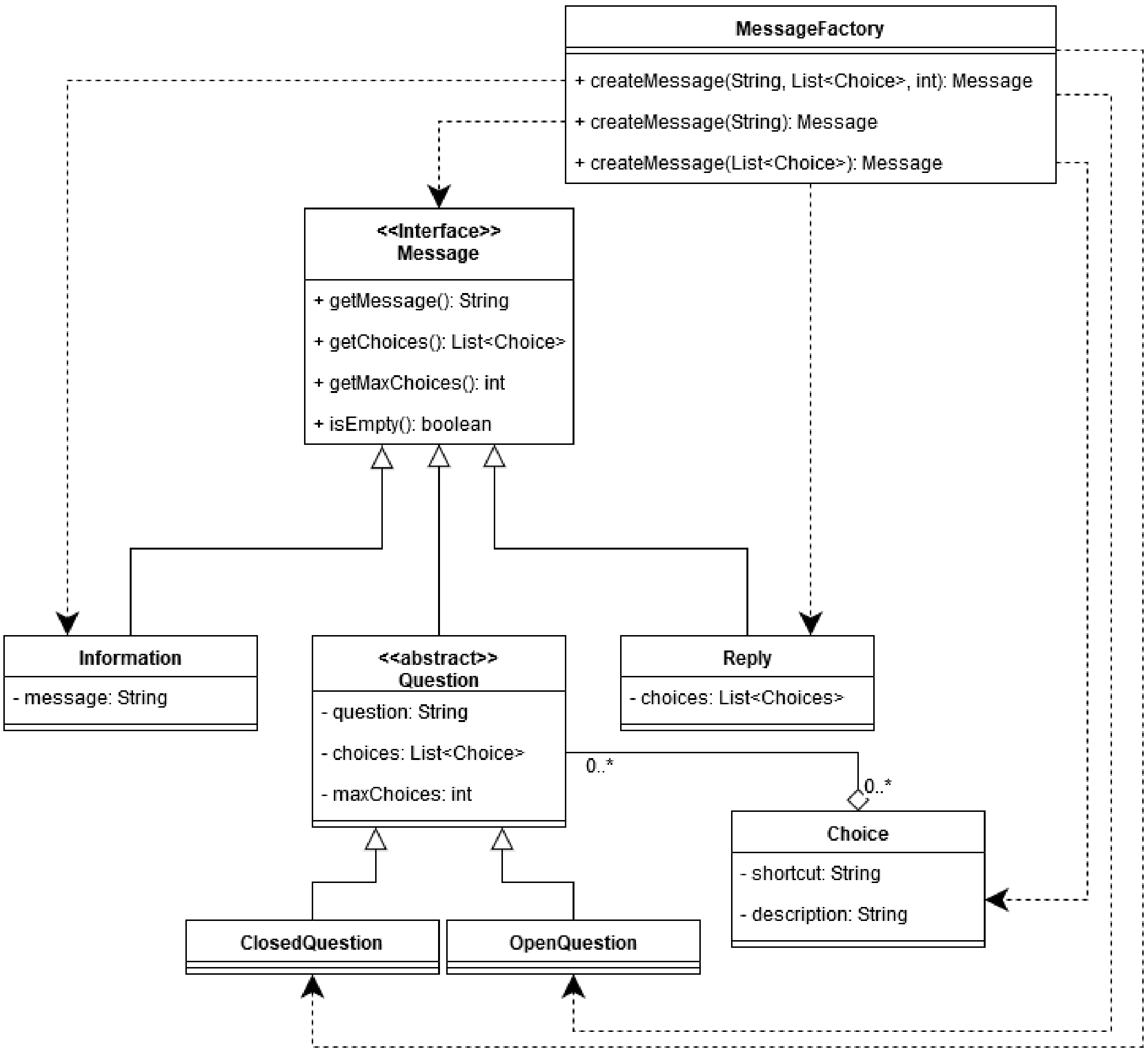


Diagram 3- Message class diagram

Fourth, a *State* pattern has been used to model the game loop: this loop has been split in different classes that accepts a *GameState* object, representing all of the game values - monsters' states (HP, stars, energy, …), players, revealed cards, … - at a given time. A *Phase* object can be played and computes the next phase when it ends. *Phase* objects can check if effects are triggered, too (e.g. give stars on attack) and stop an action if an effect tells so (e.g. prevent a monster from gaining a star when starting in Tokyo).

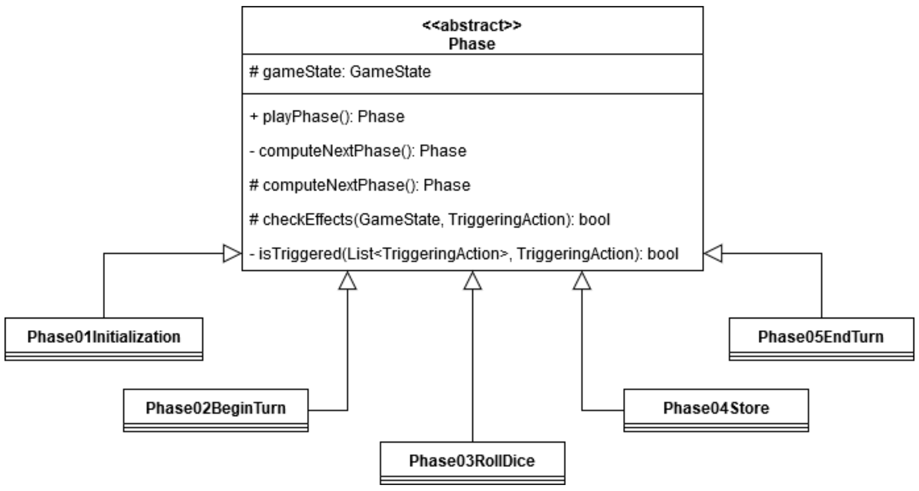

<<abstract>>
Phase
# gameState: GameState
+ playPhase(): Phase
- computeNextPhase(): Phase
# computeNextPhase(): Phase
# checkEffects(GameState, TriggeringAction): bool
- isTriggered(List<TriggeringAction>, TriggeringAction): bool
Phase01Initialization
Phase02BeginTurn
Phase03RollDice
Phase04Store
Phase05EndTurn

Diagram 4- Phase class diagram

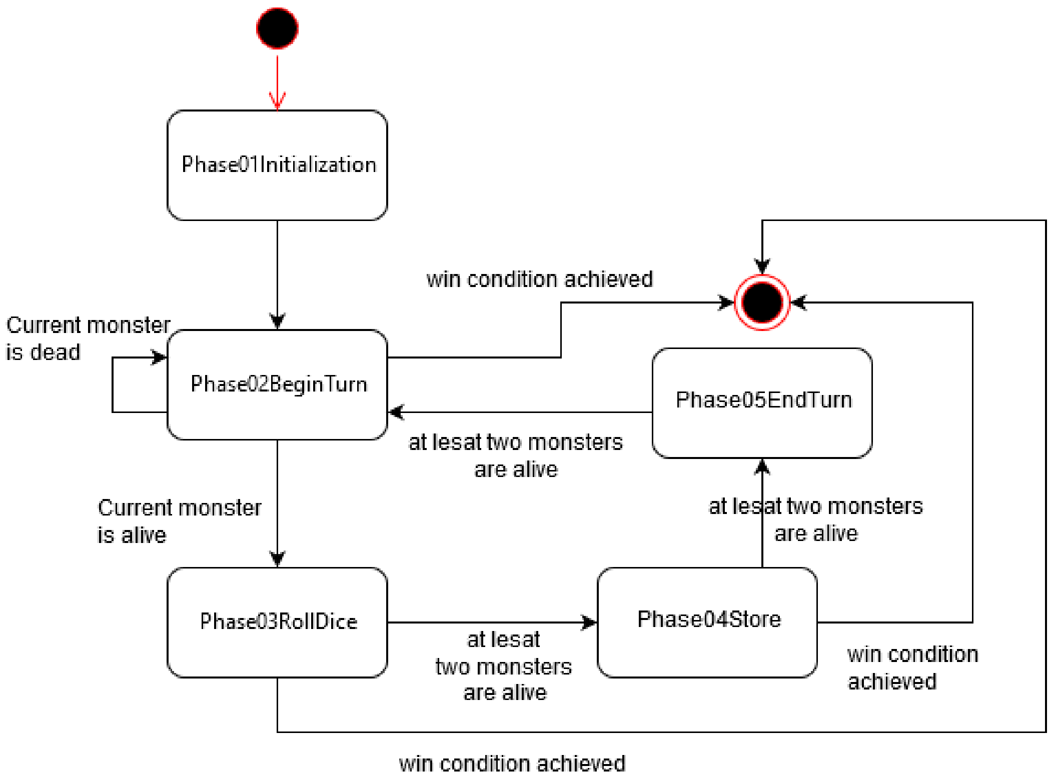


Diagram 5- Phase statechart diagram

Fifth, a *CommonAction* enumeration provides different effects that could be used by different cards (e.g. “gain stars”, “deal damages”, “increase armor”, …). These effects can be triggered at different phases and target different monsters. A *TriggeringAction* enumeration lists every action that can trigger an effect (e.g. “when a monster gains HP”, “when a monster stars in Tokyo”, …). This enumeration can be enhanced to add new triggering actions. You can find examples of those effects and triggering actions in *Representation 1- Store card*.

Testability

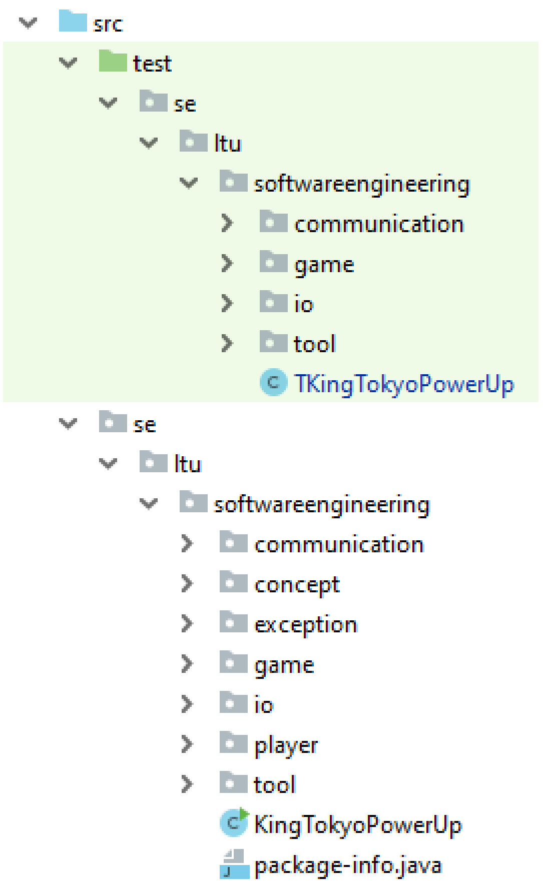
Some mechanisms have been introduced to provide testability. Here they are.

First, thanks to the *State* pattern presented in a former paragraph each phase of the game is testable independently from the others. It is the exact opposite of the current implementation wherein a single loop rules the whole game.

Second, a singleton *Randomizer* class provides a controlled randomness. It can be provided with a list of values that will be consumed later when methods need random values. When there are no more provided values, a fallback to a real random value is used. During a real game, no value should be provided. Thus, every call to the *Randomizer* returns a really random value. During a test game, the tester should provide test values to this *Randomizer*. Shuffles and dice rolls can then be controlled, and results known and tested.

Third, as shown in *Diagram 2- IO class diagram*, an *InputString* class provides a way to test the game against constant lists of inputs.

Fourth and last, the project file structure has been built in order for test classes to be in the same package as regular classes. It enables them to access to protected and package-private methods and test them.


```
src
  test
    se
      ltu
        softwareengineering
          communication
          game
          io
          tool
          TKingTokyoPowerUp
  se
    ltu
      softwareengineering
        communication
        concept
        exception
        game
        io
        player
        tool
        KingTokyoPowerUp
        package-info.java
```

Representation 2- File structure

5. Quality Attributes, Design Patterns, Documentation, Re-engineering, Testing

Prerequisites to build and run

The easiest way to build the project is to use Apache Ant. A *build.xml* file is provided within the project. It describes every useful step (directory creation, resources copying, compilation, running, cleaning). This file should not be modified.

Java 11 or higher is required.

Build and run

The project can be build using the following command:

ant compile

It will create a *build* folder containing every compiled class, resources and libraries required to run the game.

The game can then be run using the following command:

ant run

However, output messages are ugly. The preferred way to run the game is to compile (using Ant) then to run this command:

java -cp "build;build\lib\gson-2.8.6.jar" se.ltu.softwareengineering.KingTokyoPowerUp

Testing

The project can be tested using the following command:

ant runTests

It displays test classes that encountered one or more errors during the tests.

A coverage report can be generated using Jacoco. The following command should put the report in the *report* folder:

ant createReport