Dungeon Engineering Lab

Contents

1. Organizational Matters

Since you have already completed the exercise phase, the practical part of the Software Engineering Lab now begins. The practical part is divided into a group phase (∼4 weeks) and a subsequent individual phase (∼2 weeks).

In the group phase, you will design, implement, and test a game together with your fellow students in a team of 5–7 persons. Group assignment will be done by the chair on the first day of the group phase. You will be added to a team for the group phase in MS-Teams, which will serve as your virtual workspace. A tutor will be available to assist you and will be in regular contact with your team.

If you are assigned to an on-site group, your group will have access to a workroom at the university for the duration of the group phase. Your tutor will let you know which room is available for you.

You will spend the first one and a half weeks (approximately) of the group phase for the design, the following two and a half weeks (approximately) for the implementation (incl. testing). You will present your designs to a member of the Chair of Software Engineering and receive feedback. The implementation includes the game server with the game logic. In addition, you will need to develop an extensive test suite that tests all special cases that may arise, if possible.

Like customers in real life, the chair is very changeable in its requirements for the game to be implemented. Therefore, there will be a change of the game rules after the design review. Your design must be kept flexible enough to accommodate the changes in the task without much effort. The specific changes will then be announced in a separate document after the design reviews.

After the group phase, you will receive a new task in the individual phase, which must be worked on by all participants alone. You will receive more information about the individual phase after the end of the group phase. The prerequisite for the individual phase is the successful completion of the group phase.

Please note:

For the schedule of the group phase, please additionally note the information from the document on Organizational Matters, which has already been provided to you in the CMS at the beginning of the SE Lab.

1.1. Design

Regarding the design, your group must complete the following tasks:

1. You must create a UML class diagram of your system. The class diagram must contain all relevant classes of your implementation, as well as their most important fields and methods. This should show which data and which functionality you encapsulate in each class. The relations of the classes among themselves must be modeled correctly. In the class diagram, all classes as well as all non-trivial methods together with their parameters must be specified. Trivial methods include, in particular, standard getter and setter methods, as well as `hashCode`, `equals`, and `toString`.

2. You need to model the game in a state diagram to better understand the game flow and to ensure that you also fully consider this game flow in your design. Create a state diagram that is beginning at the start of the game (i.e., the configuration file is not yet loaded) and models the game itself up to the end of the game. Try to make the diagram as fine-granular and detailed as possible.

   Note: Represent only states of the game server – abstract and without concrete snapshots of the game and its dungeons etc.

3. You need to demonstrate the interaction of your classes using UML sequence diagrams for the following 2 scenarios:

   - Initialization of the game.
   - The last player (`playerID=4`) has chosen the option for his last bid. After that, all bids are be placed on the *Bidding Square*. The first bid of the first player is successfully placed on `FOOD`.

   In the requirements change after the design review, we will announce a third scenario for which you will need to demonstrate the interaction of your classes using another UML sequence diagram.

4. You must draw up a detailed time and work plan for the implementation including testing. This plan specifies who will be responsible for which parts of the implementation and who will be responsible for testing which components. In this plan, pay particular attention to the dependencies between different components and the order of the next

milestones. Do not forget to include writing of the integration and system tests in the plan as well.

Design Review & Design Defense

In the design review, you will receive feedback on your design from a member of the chair. In the design defense, your design is approved by a member of the chair. Both appointments are mandatory face-to-face appointments on-site at the campus (even for online groups).

After the design reviews, we will make changes to the specification. The changes will be based on the original game. Try to keep your design modular enough so that any changes can be easily incorporated into your design. However, you should not try to incorporate all possible changes into the design in advance.

For both appointments, design review and design defense, note that the current state of your draft that you will present to the chair member must already be uploaded in our *GitLab* repository 1 hour before your assigned review or defense appointment (see also Submissions).

1.2. Implementation

In this phase, you will complete the actual implementation of the game simulator in your group. You will also write unit tests, integration tests, and system tests. Your game implementation must meet the following requirements:

1. Your implementation must pass the system tests we create. We run our system tests on your implementation regularly during the implementation phase, and the test results will be provided to you at regular intervals in due course.
2. Your implementation must pass the unit tests, integration tests, and system tests that you create. The unit tests are there to help you debug and correct errors. If you have unit tests that your implementation does not pass, you either made a mistake when creating the tests or your implementation is still buggy. Your tests are intended to achieve the highest possible code coverage while testing reasonable scenarios, which is ensured by finding mutants[1].
3. We will examine your implementation with the code analysis tools *PMD* and *SpotBugs*. This happens automatically; your project needs to *build* and pass the code analysis tools on your `main` branch before our tests are run on your code. We expect to find no problems in this process. (If there are some rules of the code analysis tools which appear to you to be absurd in a specific situation, please contact your tutor. We expect you to be able to provide sufficient justification why you think that the rule is absurd in this situation.)

[1] https://en.wikipedia.org/wiki/Mutation_testing

With the build script we have created for you, you can run these tools yourself from the beginning of the implementation phase to check if your code still has problems.

Simply passing the tests is not enough, your code must also be well structured and follow the concepts of the lecture. In particular, each individual group member must contribute a significant amount to your group's code. (Be sure here to have *git* properly configured on your computer so that your commits are also assigned to you in our *GitLab*).

Code Review

In the code review, a member of the chair will look at your code together with you, point out weak points, and give concrete suggestions for improvement. The code review is also a mandatory face-to-face meeting.

1.3. Tools Needed

To participate in the SE Lab, you need to bring your own laptop. You need to install the following tools on your device:

- *Java 17* (our reference platform uses *OpenJDK 17.0.4*)[2]
- Version-control system *git*[3]
- IDE of your choice (we recommend *IntelliJ*[4])

The build system *Gradle*, which we use, does not need to be explicitly installed at your end, because we provide you with a project in which a *Gradle wrapper* is already included, which takes care of the installation by itself. All other tools you need do not need to be installed, but are already preconfigured in the project by us.

All libraries included in the *gradle* configuration file may be used. Other libraries must not be used.

1.4. Submissions

We will provide you with a *git* repository for the group phase in our *GitLab*[5]. To be able to clone or push to the provided repository, you need an *SSH* key. You have already learned how to generate a *SSH* key and which other technical requirements are necessary for using the repository in the practical tutorial. In the group phase, the entire group will work on one repository.

[2] https://jdk.java.net/17/
[3] https://git-scm.com/
[4] https://www.jetbrains.com/idea/
[5] https://sopra.se.cs.uni-saarland.de/

A separate branch (not `main`) is provided for submitting your design. All design documents and associated diagrams must be pushed to the repository. You can upload photos of your diagrams on paper etc. for this purpose, as long as the resolution is high enough and details are visible. The final version of your design must be pushed and marked with the tag[6] `design_defense`, which must be attached to the submitted revision. This also holds already for the preliminary version of your design, which you have to submit prior to the design review: Use the tag `design_review` in this case. To create a tag for the current commit, use the *git* command `git tag <tagname>`. Note that the submission of your design must be pushed to the repository, at latest, one hour before your assigned appointment for the design defense.

For your implementation, you must use the `main` branch after the design defense. There you will be provided with a project with a minimal code framework at the start of your implementation phase. We will not consider other branches than `main` for the code submission.

For the final submission of your implementation, the last commit on the `main` branch made to the repository no later than 23:59 CEST on the day of code submission counts. It is your responsibility to verify that the push has arrived in the repository and that everything is up to date in a timely manner. Your submissions must be executable on our reference platform (*Debian 11* with *OpenJDK 17*).

1.5. Interaction with Tutors regarding Task Specification

Inaccuracies in the task specification, which follows in the next chapters, are unavoidable. As in real software projects, there may be situations which are not completely specified in the specification text. Often this holds, for instance, for specific corner cases. In such a case, interaction with the customer is necessary. That is, you need to interact with the tutors (e.g., in the forum) to clarify unspecified details.

[6] https://git-scm.com/book/en/v2/Git-Basics-Tagging

2. The Game

Welcome to the DE Lab (...the ~~Software~~ Dungeon Engineering Lab)!

You've probably played a few dungeon crawlers in your life. Funny little games where you manage a team of adventurers trying to conquer dungeons. During their journey they fight against various monsters, avoid traps, and loot everything they find.

But have you ever thought about what you are doing? What financial damage your funny little games do to the dungeon operator? How much it costs to build such a dungeon? How much it costs to hire all these creatures? The many hours that helpers have to spend digging these tunnels? No? Then maybe now is the time for you to get to know the other side.

2.1. Introduction

This year, you have to deal with all the headaches associated with the construction and maintenance of such a dungeon. Luckily, instead of being alone, you have three loyal, hard-working minions to help you achieve your goals. They will take your orders to buy food or imps, hire new monsters, or send the imps to mine gold/tunnels. If you tell them to improve your reputation, they will do their best to do so.

Nevertheless, you might underestimate how long it takes to build such a dungeon. It might take several years to build your dungeon to excellence. Optimally, your employees work seasonally. This means you buy the material for your workers at the beginning of each of the four seasons of a year. The problem is that you are not the only one who dreams of owning a dungeon. So, you can't just buy what you want, you have to bid to get materials, new imps, monsters, etc. Sometimes, some of you will be empty handed. Some actions don't cost gold or food, but rather take a toll on your reputation, e.g., your *evilness* increases when you get to hire a monster. This might be bad for your reputation, but it also gives you a better chance to become the dungeon master at the end!

If you somehow managed to make it to the end of the year without going bankrupt and you think you can rest, we're sorry to disappoint you. Now these pesky adventurers pay you a visit and try to loot your dungeon.

At the end of the day, we hope that after playing this game, you'll never want to touch those pesky dungeon crawlers again and the Dungeon Lords can finally build their dungeons in peace!

2.2. Game Details

2.2.1. Dungeon Lord

The dungeon lord's goal is to build the dungeon and hire monsters in the *Building Phase* to make it very hard for the adventurers to conquer and destroy their dungeon during the *Combat Phase*.

Every dungeon lord must make three bids per season in the *Building Phase*, without a guarantee that these bids will be successful. Whether a bid will be successful depends on how many other dungeon lords make bids on the same option in this season.

All details on how and where these bids are placed on the *Bidding Square* is described in Section Bidding Square.

In general, there are six different options from which you can choose:

- stack up on food
- work on your reputation
- let imps mine tunnel tiles
- let imps mine gold
- acquire imps
- hire monsters

> **IMPORTANT:**
> **The option of your second and third bid will be blocked for the next season**, because everyone should have somewhat equal chances to get resources and make bids on the different options.
>
> You cannot bid on these two options in the next season, so you can—in practice—only choose from four options per season.

To have some resources to use in the first season, the dungeon lords start with three imps, three coins of gold, and three pieces of food.

Every player has one dungeon lord character in the game, so we use the term "player" and "dungeon lord" interchangeably throughout the rest of the specification.

2.2.2. Evil-O-Meter

The reputation of a dungeon lord is measured by the *Evil-O-Meter* with range 0 (*nicest*) to 15 (*most evil*). All dungeon lords start at level 5 and can move up or down one or more levels on the Evil-O-Meter, depending on the cost of the options they take. You can, e.g., get food in exchange for worsening your reputation instead of paying gold. The higher the ranking on the Evil-O-Meter, the worse your reputation is.

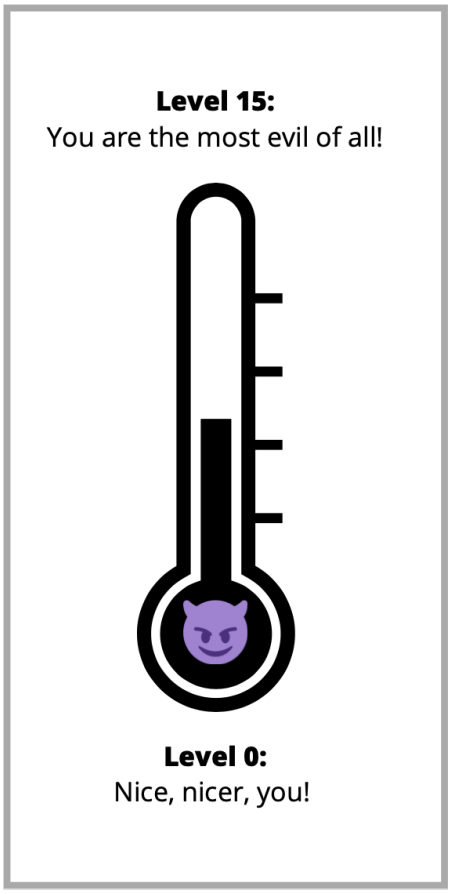

Figure 1: The Evil-O-Meter

The following statements are equivalent:

- Improve your reputation.
- Moving down on the Evil-O-Meter.
- Increase your *niceness*.
- Decrease your *evilness*.

The following statements are equivalent:

- Worsen your reputation.
- Moving up on the Evil-O-Meter.
- Decrease your *niceness*.
- Increase your *evilness*.

If fulfilling a bid would require you to move up beyond the maximum level of evilness (15 on the Evil-O-Meter), or move down below the minimum level of niceness, the bid has no effect and just expires.

2.2.3. Imps

Your imps can mine gold or tunnels, where one imp mines one tile of tunnel, or one coin of gold per season. The option "mine gold" on the *Bidding Square* allows for a maximal use of five imps at the time (corresponding to Slot 3, see Bidding Square) that mine gold even if you have more than that.

However, more than three imps cannot go without one extra imp who is supervising the rest of the group. The supervisor imps do not mine gold or tunnels themselves then (because they are busy supervising, obviously).

Imps that are placed in the dungeon to mine tunnels/gold (including the supervising imps, if necessary), are unavailable to do other tasks in this season.

After the season is finished, the imps stop working and rest until the next bid to let them work is successful.

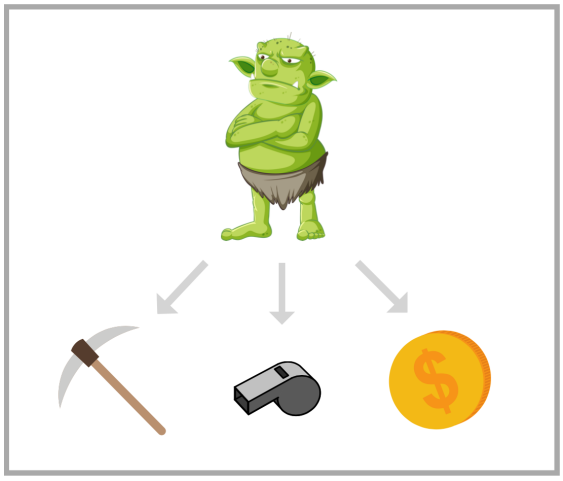

Figure 2: Imps can mine tunnels, supervise, or mine gold

In Section Bidding Square it is specified how many imps you can potentially have mining tunnels (or gold) per season.

FAQ: I have four imps and hope to get the second slot to mine gold. What happens if I land on the third?
If you land on the third slot, where four imps can mine gold, you still need one extra imp to supervise them, so five in total. Since you only have four imps, your bid expires and you cannot mine any gold in this round.

2.2.4. Dungeon

The dungeon you build is a linear tunnel, that can be extended in *tiles per imp*. Each dungeon lord starts with a one-tile long tunnel.

In front of the dungeon, there is a queue with space for three adventurers that will line up there during the year.

The first adventurer arrives at the *entrance* of the dungeon, the ones that follow simply line up further. There is one exception to this rule, which is described in Section Adventurers.

Via the entrance, the adventurers step together into the dungeon when the *Combat Phase* begins.

Figure 3: Adventurers ready for combat in front of your dungeon

2.2.5. Adventurers

Adventurers want to conquer and destroy your dungeon in the end year's combat. They move through the dungeon together as a group, so they will always stay on the same tunnel tile during the combat phase once they are inside the dungeon.

Of course, conquering the dungeon is not easy-peasy, since the dungeon lords try to protect their dungeons and attack the adventurers with their monsters. So adventurers will have to deal with *damage*, and depending on their *healthPoints*, they can deal with more or less damage before they are imprisoned. When the adventurer becomes too weak (i.e., the damage exceeds the *healthPoints*) he will be imprisoned in the dungeon.

There are different kinds of adventurers:

- normal adventurers, who only have a certain *strength* level
- the *priest* who can heal adventurer's damage
- the *warrior* who is always placed in front of the other already lined-up adventurers (who may also be of type *warrior* already).

The priest can heal damage of adventurers, including himself. The healing always starts at the adventurer who stands at the entrance of the dungeon. If the priest can heal more than the damage of the first adventurer, the second adventurer will start healing as well, and so on.

Attributes that are specified in the config file:

- `int id`: The identifier of the adventurer.
- `int difficulty`: The strength of the adventurer determines to which dungeon lord they will be assigned.
- `int healthPoints`: The health points indicate how much damage the adventurer can get before they become too weak to conquer the dungeon further.
- `int healValue (>= 0)`: The heal value indicates whether an adventurer is a priest ($healValue > 0$) or a normal adventurer ($healValue = 0$).
- `boolean charge`: If `charge` $= true$, the adventurer is a *warrior*, else a normal adventurer.

Figure 4: Configurable Attributes of Adventurers

At the beginning of each season, there are as many new adventurers arriving in town as there are players in the game. At the end of each season, those adventurer's strength will be matched to the dungeon lord's evilness to decide which adventurer will be assigned to which player. The weakest adventurer will go to the nicest dungeon lord, the second weakest to the second nicest, and so on. If there are multiple dungeon lords on the same level on the Evil-O-Meter, the weaker adventurers will go to players with the lower `playerID`s.

2.2.6. Monsters

The dungeon lords can hire monsters on the *Bidding Square* to use them in combat against the adventurers.

Per season, three new monsters are looking for employment on the *Bidding Square*. Monsters have configurable attributes and require different costs in terms of food you need, and evilness that will increase upon obtaining the monster.

Every monster you have employed can be used once every year.

Attributes that are specified per config file:

- `int id:` The identifier of the monster. This is needed when the player chooses a monster, so that the server can process this request correspondingly.
- `int hunger`: The food you have to feed upon hiring the monster.
- `int evilness:` This is how many levels you move up the Evil-O-Meter when you employ this monster.
- `int damage:` This is the damage it can cause one or more adventurers depending on its *attackStrategy*
- `String attackStrategy:` The attack of a monster can proceed differently:
  - `BASIC`: The monster attacks only the first adventurer. All the damage is applied to the first adventurer, even if it exceeds the `healthPoints` of the adventurer.
  - `MULTI`: All adventurers get the indicated amount of damage from the monsters' attack.
  - `TARGETED`: The player can choose which adventurer should get the damage. It then follows the same principle as the `BASIC` attack.

Figure 5: Configurable Attributes of Monsters

2.2.7. Bidding Square

The *Bidding Square* is the central space where all bids of all players per season are placed in a more or less auction-like manner.

The bids of all players are placed *round-robin* always with ascending `playerID` and starting at the start player of this season.

First, all first bids of all players are placed, then the second bids of all players, and finally the third bids of all players.

Figure 6 shows the structure of the *Bidding Square*, where you can bid on getting food, niceness, imps, monsters, or wanting to send your imps mining tunnels or gold.

There are only three slots per option available on the *Bidding Square*, so only the first three players are lucky. Since every player has *only one possible bid per option per season*, the

number of slots corresponds to the number of players that have a chance to get or buy the item of interest in one season.

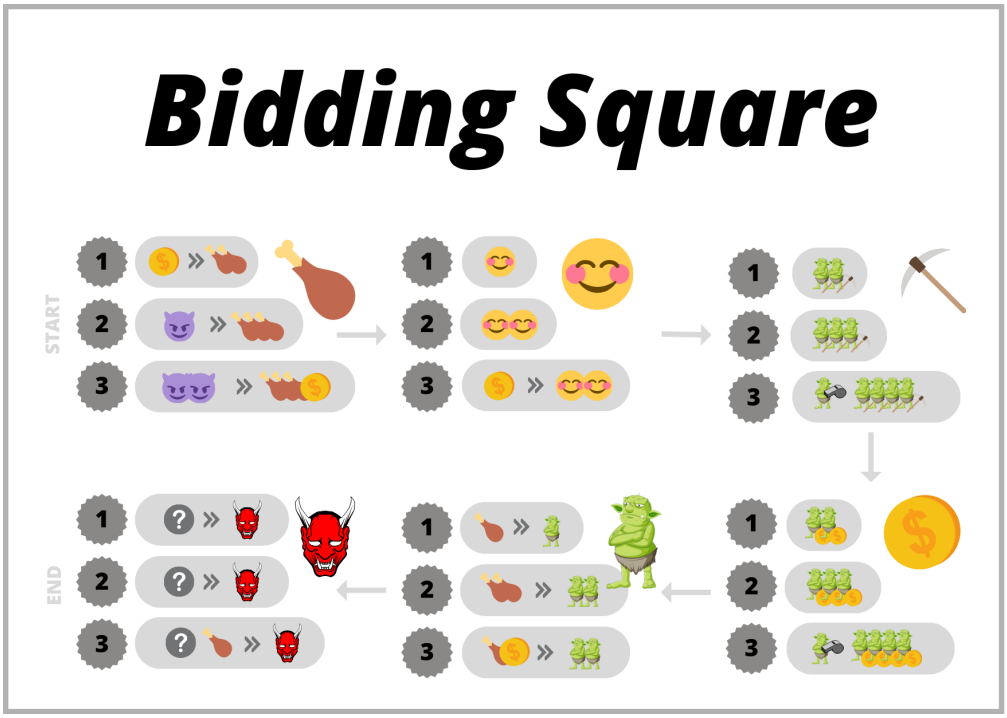

Figure 6: Structure of the *Bidding Square*: Options, Slots and Consequences

Once all bids are placed on the *Bidding Square*, the evaluation starts. The dungeon lords have to pay the full cost like indicated on the slot their bid was placed.

If you have the resources to fulfill the bid, then you cannot choose to opt-out, even if it's not the slot you hoped to get. This might be the case when you, e.g., hoped to get the cheapest slot (which is not always equal to the first one!), but it's taken already when it's time for your bid to be placed.

If you cannot pay in full, your bid simply expires and you get nothing. It is not possible to pay less to receive less, or to pay more to receive more.

Determination of the start player:

- All players have a unique `int playerID` (see Section Registration and Preparation Phase).
- The player with `playerID = 0` is the start player at the start of the game.
- The role of the start player is moved to the player with the next larger `playerID` after each season of the *Building Phase*.

Description of the *Bidding Square* Slots:

1. `FOOD`: You can either get two pieces of food, three pieces of food, or three pieces of food AND one coin of gold.
    - Slot 1: Pay 1 coin of gold, get two pieces of food.
    - Slot 2: Move up one level on the Evil-O-Meter, get three pieces of food.
    - Slot 3: Move up two levels on the Evil-O-Meter, get three pieces of food AND one coin of gold.
2. `NICENESS`: You can either move down one or two levels on the Evil-O-Meter.
    - Slot 1: No extra cost, just move down one level on the Evil-O-Meter.
    - Slot 2: No extra cost, just move down two levels on the Evil-O-Meter.
    - Slot 3: Pay 1 gold to move up two levels on the Evil-O-Meter.

3. `TUNNEL`: You can have your imps mine two, three, or four tunnels.
    - Slot 1: Place two imps in your dungeon to mine two tunnels.
    - Slot 2: Place three imps in your dungeon to mine three tunnels.
    - Slot 3: Place four imps AND one supervisor imp to mine four tunnels.

4. `GOLD`: You can have your imps mine two, three, or four coins of gold.
    - Slot 1: Place two imps in your dungeon to mine two coins of gold.
    - Slot 2: Place three imps in your dungeon to mine three coins of gold.
    - Slot 3: Place four imps AND one supervisor imp to mine four coins of gold.

5. `IMPS`: You can buy one or two new imps.
    - Slot 1: Pay one piece of food to get one imp.
    - Slot 2: Pay two pieces of food to get two imps.
    - Slot 3: Pay one piece of food AND one coin of gold to get two imps.

6. `MONSTER`: You can choose one of the monsters that are up for sale.
    - Slot 1: Feed the monster you choose according to its hunger value, and move up the Evil-O-Meter based on the evilness value of the monster.
    - Slot 2: Same as Slot 1.
    - Slot 3: Same as Slot 1 AND feed one piece of food extra.

2.2.8. Final Score of an Individual Dungeon Lord

After the combat phase at the end of the last year, all players are awarded points based on how well their dungeons were managed. The following criteria are checked at the very moment of evaluation:

- +1 point per employed monster
- -2 points per conquered tunnel tile
- +2 points per imprisoned adventurer
- +3 points per *title* (if there is a tie, the players get +2 points each):

  The Lord of Dark Deeds: the dungeon lord who ranks highest on the Evil-O-Meter

  The Tunnel Lord: the dungeon lord who has the longest tunnel

  The Monster Lord: the dungeon lord with the most monsters

  The Lord of Imps: the dungeon lord with the most imps

  The Lord of Riches: the dungeon lord with the most food and gold left

  The Battlelord: the dungeon lord with the most unconquered tiles

2.3. Game Phases

You are in the *Building Phase* for one year, which is divided into four seasons (or rounds). At the end of each year, there is a battle in the *Combat Phase*, where the adventurers try to take over your dungeon. This phase also lasts four rounds.

The amount of years the game lasts is configurable.

2.3.1. Building Phase Steps

After all players are registered for the game, the game starts in the first year with the first round of the *Building Phase*.

What happens per round:

- *The new adventurers arrive in town**:
  Draw as many new adventurers as there are players in the game.
- *The new unemployed monsters arrive at the Bidding Square as well*:
  Draw three new monsters, discard the old ones (i.e. remove them from the game).
- *Choose options to bid on*:
  Each player decides which options they want to bid on this season. The game can only progress when all players have chosen their three bids.

- *Place and Evaluate Bids*:
  The bids are placed round-robin on the *Bidding Square*, like described in Section Bidding Square. Afterwards, the bids are always evaluated in the same order (and from Slot 1 to Slot 3), i.e., `FOOD`, `NICENESS`, `TUNNEL`, `GOLD`, `IMPS`, `MONSTER`.
- *Lock options*:
  Lock the option types of each player's second and third bid from this round. Unlock the old ones.
- *Return Imps*:
  If the bid for mining gold or tunnels was successful, your imps stop working.
- *Spread Adventurers**:
  At the end of the season, the adventurers choose which player they will fight at the end of the year, based on their current strength-evilness match as described in Section Adventurers.

* = **Only in the first three rounds!** There are no new adventurers coming in the last round, because the combat it happening right after the last round of building.

2.3.2. Combat Phase

When all four seasons have passed, the *Combat Phase* of the year begins, which also lasts four rounds. The adventurers try to conquer your dungeon. You can fight them with monsters you hired during the *Building Phase* to save as much of your dungeon as possible ...

Combat is happening for each player individually, sorted by `playerID`.

What happens per round:

- *Defense*:
  The player can place a monster in the dungeon to attack the adventurers.
- *Damage*:
  Each adventurer gets a default fatigue damage of two each round, and they also get damage from the monsters that have attacked them, depending on which `attackStrategy` was used.

  When the damage exceeds an adventurer's *healthpoints*, he gets imprisoned.
- *Conquer*:
  All adventurers step into the tunnel at the beginning of the first round, and conquer one tile per round. The order of adventurers inside the group is not changed, as this is important for who is damaged or healed first.

  For each conquered tile, the dungeon lord has to move down one level on the Evil-O-Meter.

If all tiles have been conquered already at this point, then one imprisoned adventurer flees instead. (Fled adventurers are just removed from the game.)

If no adventurer is imprisoned and all tiles are conquered, the combat is over for this dungeon lord for this year.

- *Heal damage*:
  If a priests is part of the group of adventurers, and he is not imprisoned yet, he heals the damage of the group according to the details described in Section 2.2.5.

When the *Combat Phase* is over, the adventurers are removed from the tunnel, the remaining adventurers that have not been imprisoned are removed from the tunnel and discarded as they are not needed anymore. The conquered tiles are irreversibly damaged. In the next year's *Combat Phase*, the new adventurer group will start at the first unconquered tile.

3. Technical Details

In this chapter, we explain more technical details—from the build script, to code analysis tools, and, finally, to technical implementation details. Here we also explain the framework that is provided by us and must be used by you.

3.1. Client-Server Communication

The game is controlled by a server (written by you), which communicates with the clients (players) using **Actions & Events**. We provide you with a *CommLib* (Communication Library) that handles part of the communication infrastructure between server and clients already.

Actions are sent by the clients to the server to tell it what the player wants to do. The server in turn uses *Events* to tell the clients what has changed in the game.

Figure 7 shows the interaction of your server and a client. The yellow objects in the diagram are the parts that you need to implement, i.e., the game server and the `ActionFactory`.

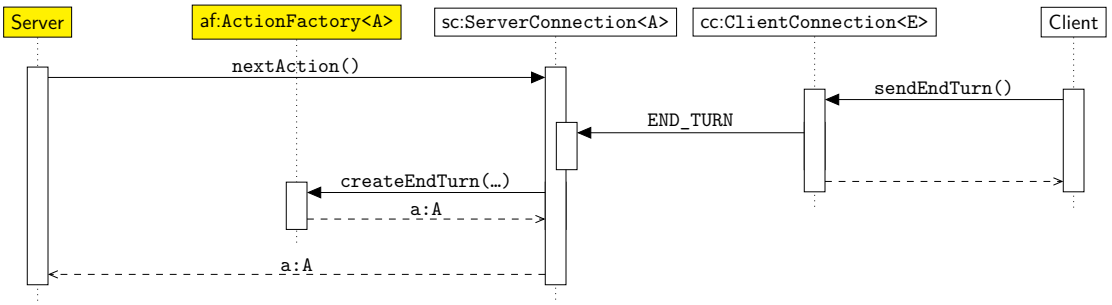

Figure 7: Server side interaction of the `ActionFactory` with the `ServerConnection`.

Type `A` represents your concrete implementation of an `Action` superclass. You need to implement the provided `ActionFactory` interface, that you instantiate with your `Action` type. Accessing this `ActionFactory` interface is possible via the `ServerConnection` class, which is part of the CommLib.

Type `E` represents the type of an event, which we are not concerned with as this only happens on the client side, not on the server's side. You do NOT need to worry about the `ClientConnection` or `EventFactory`, Figure 8 should only serve to deepen your understanding of the interactions between the different objects.

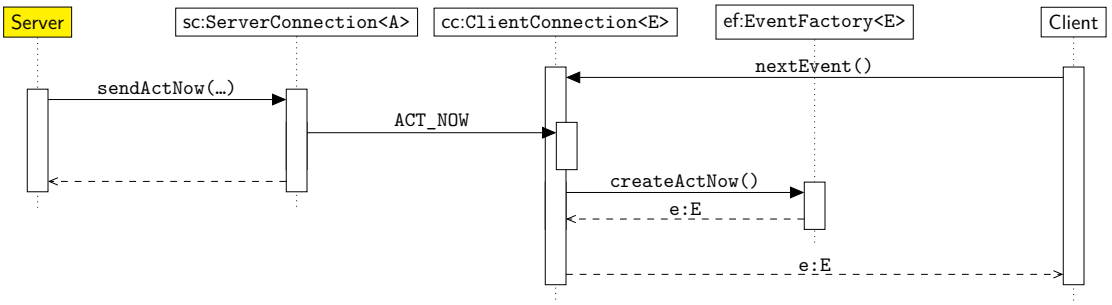

Figure 8: Client side interaction of the `EventFactory` with the `ClientConnection`.

We distinguish between *individual events* and *broadcast events*. Individual events are only sent to the player it concerns, while broadcast events have to be sent to all players.

The CommLib sends events only to one player at a time, so your server has to make sure that broadcast events are actually sent to all players. To do this, each player has (in addition to its ID) a `commID`. This is automatically assigned by the CommLib and must be mapped to the `playerID`. All actions get the `commID` of the executing player as parameter. All actions sent by non-registered clients are ignored during the game.

3.2. Registration and Preparation Phase

The game is divided into several phases. It begins with the registration phase, which is followed by the preparation phase and, finally, the actual game. The individual phases are described in detail below.

After the game details have been parsed from a provided Configuration File, your game server is started.

In the registration phase, the server waits for all players who want to participate in the game to register. During this process, each player is assigned a `playerID`, which starts at 0 and is always incremented by 1. An already registered player can start the game, even if the maximum number of players is not yet registered.

The preparation phase starts after either the maximum number of players is reached, or when a registered player starts the game.

The server first notifies all clients (i.e., the registered players) that the game has started, followed by the information to each client which other players are registered.

Now everything is set up; the first year starts with the first round of the *Building Phase*.

All details on the order of the events that your server needs to send is described in the appendix.

The server should terminate in the following cases:

- When the server times out while waiting for players to register.
- When all players have left the game (i.e., have sent the `Leave` action).
- When the server times out while waiting for the bids of the players, i.e., we can assume that there are no active players left.
- When all rounds (years) have been played. In this case, the game ends normally.

3.3. Random Number Generator

To keep our game deterministic despite random computations, we use the *java.util.Random* random number generator, which we initialize with a given `seed`.

> **Why is this important?** *"If the seed parameter is passed, the random number generator initializes its internal counter with this value, and the subsequently generated sequence of random numbers is reproducible. On the other hand, if the parameterless constructor is called, it initializes the random number generator based on the current system time. In this case, the sequence of random numbers is not reproducible."* [a]
>
> [a] https://dbs.cs.uni-duesseldorf.de/lehre/docs/java/javabuch/html/k100105.html

When initializing the game, we use the `java.util.Random random` object to shuffle the lists of objects you have read in via the configuration file.

We shuffle these lists using *Collections.shuffle()* in this order:

```
Collections.shuffle(<list of all monsters>, random);
Collections.shuffle(<list of all adventurers>, random);
```

Whenever an object is drawn from a shuffled list during the game, it's always the first object at index 0 which is drawn. Drawn objects are never put back, but always discarded if they did not come to use.

3.4. Configuration File

Instead of setting all game parameters during the implementation, we will instead read in a configuration file at the start of the game server, which models the game to a large extent. This will be used, for example, to specify the attributes of different characters and other game parameters.

As notation we use JSON[1]. Additionally, we specify schemas[2] that determine the structure and, if necessary, value ranges for fields in the configuration file.

It is useful to deal with these formats in detail, as a number of aspects of the game are already defined here.

Requirements for a valid configuration file:

- The maximal allowed number of players must be given (`int maxPlayers`).
- The number of years the game is played must be given (`int years`).
- There have to be enough monsters. Each season, there are three new monsters drawn:
  $\#Monsters \geq \#drawnMonsters * 4 * years$
- There have to be enough adventurers:
  $\#Adventurers \geq maxPlayers * 3 * years$
- The `ids` of monster and adventurers are unique among each other with `id` ≥ 0.
- The `hunger` of a monster is ≥ 0 and by default 0.
- The `evilness` of a monster is ≥ 0 and by default 0.
- The `damage` of a monster is ≥ 1.
- The `attackStrategy` of a monster has to be set.
- The `difficulty` of an adventurer is ≥ 0.
- The `healthPoints` of an adventurer is ≥ 1.
- The `healValue/defuseValue` of an adventurer is ≥ 0 and by default 0.

Note: Configuration file requirements aren't necessarily corresponding to game state requirements in general.

We will provide an example of a configuration file in the CMS.

[1] https://www.json.org
[2] https://json-schema.org

3.5. Command-Line Interface

The server is started with these command line parameters:

`--config <path>` The path to the file from which to load the configuration in *JSON* format.

`--port <int>` The port on which the client can communicate with the server.

`--seed <long>` The seed used to initialize the game's random number generator (see Random Number Generator).

`--timeout <int>` The server's timeout in seconds (the maximum time the server will wait for Actions from the client).

3.6. Build Script

We provide you with a build script. A build script is responsible for managing the dependencies of a software project and creating a finished executable program from the project files. As a build tool, we use *gradle*[3].

The build script can be found in the `build.gradle` file. Changes to your build script will be rolled back from the test server for execution. You can (and should) build the project yourself by either running the `./gradlew build` command or running the *gradle* task `build` directly in an IDE (e.g. *IntelliJ*). We use *Java 17*.

In the build script there are dependencies on various libraries. We recommend that you look at these libraries and consider whether you can use them. Other libraries that are not entered in the build script must not be used. For security reasons, we do not allow reflection, network connections, or other connections to the outside world except for the used *CommLib* (see Client-Server Communication). Also, with the exception of loading configuration files, you may not read, create, or modify files in the filesystem. For loading configuration files, you may only access the already existing `resources` directory in the project repository.

3.7. Code Quality

Good code quality helps software to remain readable and to better avoid errors. In the SE Lab, you will learn about automated tools that help you write good and (hopefully) bug-free code. You can also use these tools with *gradle*.

CheckStyle This tool checks if you follow the style guide that is specified. We provide you with a code style config, which you can customize or use as is.

PMD is a static analysis tool, which means it doesn't need to run your code to find errors. PMD examines the source code for certain patterns (not *design patterns* ;D) that are

[3] https://gradle.org/

known to cause errors. Examples of such patterns are unreachable code (you probably forgot to call a method) or comparing two strings with == (in Java, strings must be checked for equality with `equals()`). The report can be opened with a web browser and lists all problem locations and links to the exact problem descriptions.

You will notice that *PMD* does not allow `System.(out|err).println` to be used. Such console output should only be used during debugging. You can use the logging framework *SLF4J*[4] instead, which is more configurable.

SpotBugs SpotBugs is also a static analysis tool, but it works on the compiled Java bytecode rather than directly on the source code. This allows it to find problematic code locations missed by *PMD*, such as when certain expressions always have the value `null`. Again, there is a bug report pointing out the problem.

[4] https://www.slf4j.org/manual.html

4. Tests

In the group phase we expect system, unit, and integration tests from you. You can find the requirements, which are demanded from your application, in the specifications from the previous chapters.

Try to achieve the highest possible statement and branch coverage in the implementation phase. This means that in your code as many statements and branches as possible should be covered by the tests. The coverage can be calculated, e.g., with *JaCoCo*[1], also *IntelliJ* has already integrated tools for this.

4.1. Unit Tests

For unit tests, use *JUnit 5.9.0*[2] and *Mockito*[3]. Various tutorials for using these frameworks can be found on the Internet[4]. We will provide you with a sample test at the beginning of the implementation phase.

The unit tests can be run with `./gradlew test` (or `gradlew.bat test`).

Hint for implementation

Unit tests belong in the given directory `src/test/java` in the project folder.

4.2. Integration Tests

It is your job to also write integration tests to test how the modules work together. In integration tests, you access individual methods similar to unit tests. You can write and execute these tests like Unit Tests.

[1] https://www.jacoco.org/
[2] https://junit.org
[3] https://site.mockito.org
[4] https://junit.org/junit5/docs/current/user-guide/

4.3. System Tests

You also need to write system tests that sufficiently test the game server for all functionality. This time, the framework provides the option for different system tests.

We provide a framework that you will receive at the beginning of the implementation phase together with the *CommLib*. The framework takes care of the communication with the server.

Your system tests will also run on the reference implementation, so that you can determine if your assumption about the server's behavior is correct.

Note that your system tests are also run on faulty game servers (mutants), so your tests must find faulty implementations.

Hint for implementation

System tests belong in the given directory `src/systemtest/java` in the project folder.

A. Appendix

A.1. Actions and Events

Please note that this table should give you an overview of possible actions and events, how they look like and when they will be used. With the change in game rules, there might be new Actions/Events, or existing ones adapted or even not needed anymore. So please keep that in mind when designing your game.

Actions

EndTurn()
Using this action, a client ends their turn.

HireMonster(`int` monster)
Using this action, a client can hire an unemployed monster. If the monster does not exist, or if the player cannot feed the monster or move enough levels up on the Evil-O-Meter, an "actionFailed" will be sent.

Leave()
Using this action, a client can leave. The client will be removed from the game, all their hired monsters will be unemployed again and all their imprisoned adventurers escape.

Monster(`int` monster)
Using this action, a client can attack the adventurers with one of their monsters. If the monster's `attackStrategy` is not `BASIC` or `MULTI`, or if the client which sent the action does not own the monster, an "actionFailed" will be sent.

Actions
MonsterTargeted(`int` monster, `int` position) Using this action, a client can attack an arbitrary adventurer with one of their monsters. If the client which sent the action does not own the monster, or the monster does not have the `attackStrategy TARGETED`, an "actionFailed" will be sent.
PlaceBid(`BidType` bid, `int` number) Using this action, a client sends out a minion to place a bid. The second parameter `number` refers to the first, second, or third bid of a player. If the specific type of bid is not available for the player, or the player has already placed three bids, or has already send a bid with the same number, an "actionFailed" will be sent.
Register(`String` playerName) Using this action, a client can register to the server. If an already registered client uses this action an "actionFailed" is sent.
StartGame() Using this action, a client can tell the server that the game should start before the maximal number of players has been reached. If this action is used when the game has already started an "actionFailed" is sent.

Table 1: Actions

Events

ActNow()
With this event the server tells the client that it is their turn and a new action is expected.

ActionFailed(`String` message)
With this event the server informs the client that their least recently used action failed to execute. The message hereby provides the reason for the failure. *Hint: We do not explicitly test the content of the message, yet it is not allowed to be empty and be very helpful when debugging with system tests.*

AdventurerArrived(`int` adventurer, `int` player)
With this event the server informs the client that the provided adventurer arrived at the given players dungeon.

AdventurerDamaged(`int` adventurer, `int` amount, `int` player)
This event notifies a client that the given adventurer has been damaged.

AdventurerDrawn(`int` adventurer)
This event notifies a client that an adventurer has been drawn from the game's stack of adventurers.

AdventurerFled(`int` adventurer)
This event notifies a client that the given adventurer fled the prison.

AdventurerHealed(`int` amount, `int` priest, `int` target)
This event notifies a client that the given adventurer has been healed.

AdventurerImprisoned(`int` adventurer, `int` player)
This event notifies a client that the given adventurer has been imprisoned.

Events

BiddingStarted()

This event notifies a client that the bidding has started, followed by an via an "actNow" (i.e. all players are asked to place their bids).

Config(`String` config)

With this event the server sends the configuration file to the client as a JSON string. This configuration will be sent right after the 'Registration'-action.

DefendYourself()

With this event the server notifies the client it's time to place monsters to defend the dungeon against the adventurers.

DigTunnel()

With this event the server notifies the client that it's time to dig a tunnel.

EvilnessChanged(`int` amount, `int` player)

With this event the server informs the client that the provided player's evilness changed. With $amount > 0$, the player gets more evil, with $amount < 0$, the player gets nicer.

FoodChanged(`int` amount, `int` player)

With this event the server informs the client that the provided player's food changed.

GameEnd(`int` player, `int` points)

With this event the server informs the client that the game has ended. The parameter 'player' indicates the playerID of the player who won the game, as well as the respective 'points'.

GameStarted()

With this event the server informs the client that the game started.

Events

GoldChanged(`int` amount, `int` player)
With this event the server informs the client that the provided player's gold changed.

SelectMonster()
With this event the server informs the client that they are now allowed to select a monster from the available ones.

ImpsChanged(`int` amount, `int` player)
With this event the server informs the client that the provided player's amount of available imps changed.

Left(`int` player)
With this event the server informs the client that player with the given id has left the game.

MonsterDrawn(`int` monster)
With this event the server notifies a client that a monster has been drawn and is available to be hired this round.

MonsterHired(`int` monster, `int` player)
With this event the server informs the client that the provided player hired the given monster to their army.

MonsterPlaced(`int` monster, `int` player)
This event notifies the client that a player placed a monster in their dungeon.

NextRound(`int` round)
This event notifies the client that a new round started and provides the number of the round. The first round is always 1.

Events
NextYear(`int` year) This event notifies the client that a new year started and provides the number of the year. The first year is always 1.
BidPlaced(`BidType` bid, `int` player, `int` slot) With this event the server informs the client that a player placed their bid on a given slot.
BidRetrieved(`BidType` bid, `int` player) With this event the server informs a client an bid is now available again to a player.
Player(`String` name, `int` player) After registration is done the server sends this event to inform all players which players will be participating in the game. The events will be sent in ascending bid of 'player's.
RegistrationAborted() This event notifies the client that the server is aborting the registration phase because it received an invalid action. Invalid actions are all which is not the 'StartGame'-action or 'Register'-action.
TunnelConquered(`int` adventurer) With this event the server informs a client that the provided adventurer conquered a tunnel of the dungeon.
TunnelDug(`int` player) With this event the server informs the client that the provided player dug one tunnel tile.

Table 2: Events