

Victoria University of Wellington

SWEN 225 Software Design

Group Project 2022: Chap's Challenge

Worth 30% of Overall Mark

Important Dates	
Team Signup Deadline:	24 August (Mid-Trimester break)
Integration Day Presentations:	26 - 30 September (Week 10)
Submission Due Date (group work):	7 October (Week 11)
Submission Due Date (individual work):	7 October (Week 11)

Introduction	2
Requirements	3
Chip vs Chap	3
Architecture	4
The Domain Component (package nz.ac.vuw.ecs.swen225.gp22.domain)	5
App (package nz.ac.vuw.ecs.swen225.gp22.app)	8
Rendering (package nz.ac.vuw.ecs.swen225.gp22.renderer)	9
Persistency (package nz.ac.vuw.ecs.swen225.gp22.persistency)	9
Recorder (package nz.ac.vuw.ecs.swen225.gp22.recorder)	9
Fuzz Testing (package test.nz.ac.vuw.ecs.swen225.gp22.fuzz)	10
Quality Assurance	11
Submission and Assessment	11
Integration Day	11
Program Code	12
Repository Use	13
Individual Report	13
Submission	14
Group Project Common Parts	14

Group Project Individual Parts	15
Reflection Essay	15
Late Penalties	15
Marking Guide	15
Module-Specific Quality Criteria	17
Use of External Libraries	18
Penalties	18
Additional Notes, Clarifications and Corrections	18

Introduction

You are to design and implement a program for a single-player graphical adventure game using only the tools available in the Java standard library, and selected libraries from a whitelist provided in this document. The objective of the game is to explore an imaginary world, collecting objects, solving puzzles, and performing actions to complete the game. This project will bring together all of the techniques you have been taught thus far.

Your final solution must run on a stock installation of Java 17 (e.g. Oracle or OpenJDK). This limits the scope of the project, and gives everyone a level playing field.

You should undertake this project in teams of 5-6 students. We want you to work in teams for two different reasons: firstly, to experience what it is like to work as part of a team on a software development project; secondly, to be involved in a larger project without you having to do all the work yourself. Your team must be registered using the online team signup system:

<http://www.ecs.vuw.ac.nz/cgi-bin/teamsignup> . **The team signup system will close on Wednesday 24 August @ 2359 sharp**, no late requests will be considered, and you must ensure your team is registered by then.

NOTE: You may register incomplete teams (e.g. 2 - 4 people) who wish to work together. All incomplete teams will be combined together to form complete teams of 5-6 students. Anyone not in a team will be automatically added to a team after the team signup system has closed.

Working in Teams. We understand that there are sometimes difficulties when working in teams, because of personality conflicts, and the pressures of time and workload. You will be able to resolve some such difficulties yourself, but we are willing to help. If your team is facing problems that you cannot resolve, then please seek our assistance.

NOTE: While this is a team project, we will mainly assess individual contributions.

Requirements

What follows are the main requirements for the game. They are neither extremely detailed, complete and may even be inconsistent: your team must make reasonable assumptions where necessary or ask for clarification (preferably on the course forum).

Chip vs Chap

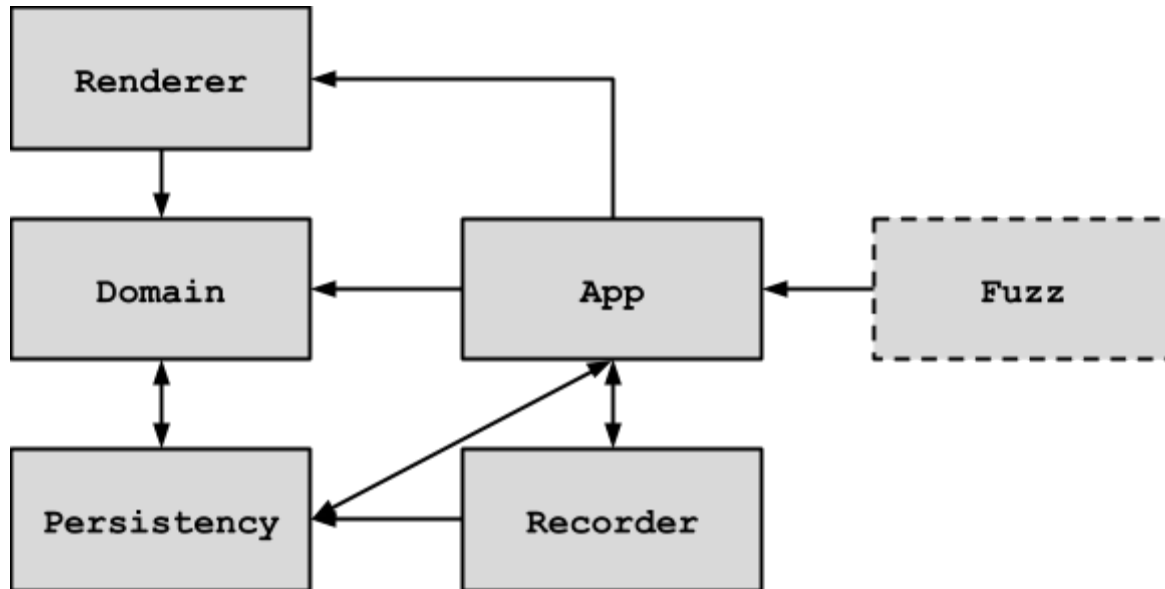
Chap's challenge is a **creative** clone of the (first level of the) 1989 Atari game Chips Challenge. To learn more about Chip's Challenge, please read the [Wikipedia article](#), watch a [playthrough on youtube](#), or even try to get hold of the actual game (there are some free browser-based versions, and it is [available on Steam](#)). In Chips Challenge, an actor moves through a maze directed by the keystrokes of the user, collects treasures and searches for the exit that leads to the next level. Here is a screenshot of the Chip's Challenge board:



We have been given permission by Chuck Summerville, the developer of CC, to reimplement this game. Thank you Chuck !

Architecture

The game should adopt high-level architecture consisting of modules, as illustrated in the following figure. The figure also shows module dependencies and interactions. Each module is associated with exactly one Java package containing the code. See below for details.



The core components of the design are:

1. The **Domain** module. This module is responsible for representing and maintaining the state of the game, such as what types of objects there are and which actions are allowed to change the state of those objects. This module is not concerned with how the game is displayed or stored.
2. The **App** module. This module is responsible for managing the functionality of the game (e.g. starting new games, loading/saving games, moving the player, managing the application window(s), etc). User input is managed by this module, this includes mapping key strokes back to objects in the game world, etc.
3. The **Renderer** module. This module is responsible for drawing the maze onto a canvas or panel, including transitions, animations, focus areas and sound.
4. The **Persistency** module. This module is responsible for reading map files and reading/writing files representing the current game state (in XML format) in order for the player to resume games.
5. The **Recorder** module. This module records games, and allows a user to load and replay games.
6. The **Fuzz** module. This module can load a game level, and play a game using randomly generated input. This means that the fuzz event generates events that would usually be

generated by the user (key strokes, mouse clicks, etc). The purpose of this module is to detect violations of the game logic.

It is required that members of your team should choose one module to be responsible for. The assignment must happen before integration day, and must be defined in a text file **team.md** located in the root folder of the repository that contains a table with the following structure:

Module	Team Member Name	Team Member gitlab account
Domain	Jonathan Jones	jonejo
..




It is expected that the majority of the commits for each module are made by the person assigned to this module. A permitted exception is code refactoring, when some changes are made simultaneously to multiple modules. Such commit messages should contain the word “refactoring”.







Each of these components will now be considered in more detail.

The Domain Component (package `nz.ac.vuw.ecs.swen225.gp22.domain`)

This is the domain model that forms the core of the application, it conceptualises the maze, its elements and their interaction. Because it is so central, it is now discussed in more detail. All team members must have a thorough understanding of this module.

The maze is made up of tiles. The tiles are:

Name	Corresponding icon In Chip's Challenge	Description
Wall tile		Part of a wall, actors cannot move onto those tiles.
Free tile		Actors can freely move onto those tiles.
Key		Actors can move onto those tiles. If Chap moves onto such a tile, it picks up the key with this colour, once this is done, the tile

		turns into a free tile.
Locked door		Chap can only move onto those tiles if they have the key with the matching colour -- this unlocks the door. After unlocking the door, the locked door turns into a free tile.
Info field		Like a free tile, but when Chap steps on this field, a help text will be displayed.
Treasure		If Chap steps onto the tile, the treasure (chip) is picked up and added to the treasure chest. Then the tile turns into a free tile.
Exit lock		Behaves like a wall time for Chap as long as there are still uncollected treasures. Once the treasure chest is full (all treasures have been collected), Chap can pass through the lock.
Exit		Once Chap reaches this tile, the game level is finished.
Chap		The hero of the game. Chap can be moved by key strokes (up-right-down-left), his movement is restricted by the nature of the tiles (for instance, it cannot move into walls). Note that the icon may depend on the current direction of movement.

The game has levels, each level defines a unique maze, the actual levels are *defined in the persistency module*. Level 1 should have all the tiles listed in the table above, and at least two different types (colours) of keys. Keep it simple so that the level can be easily played and finished *within one minute*. You should also implement a second level with some different tiles. The original Chip's Challenge game will provide some ideas, but you should invent your own tiles (see renderer component description). You should also use a second type of actor in level 2 -- an actor is a game character that moves around, like Chap, and interacts with Chap (for instance, by exploding and eating Chap or robbing him). Unlike Chap, actors will move around on their own (randomly, or following some pattern), and are not directed by user input.

The classes in this module (package) are responsible for maintaining the game state and implementing the game logic. The game state is primarily made up of the maze itself, the current location of Chap on the maze, the treasure chest and other items Chap has collected, such as keys. The game logic controls what events may, or may not happen in the game world (e.g. "Can Chap go through this door?", "Can Chap pick up this object?", "Does this key open that door?", etc.).

The core logic of the game is that the player moves Chap around the maze until it reaches the exit and then advances to the next level (if there is one). This module should implement extensive checks ensure the integrity of the game. For instance, ensure that:

- Chap can never stand on a wall
- the total number of treasures (collected and uncollected) is constant during the play of a given level
- the total number of treasures is non-negative
- if a treasure has been collected, the number of treasures still on the board is reduced by one
- if Chap stands on a locked door, then it must have collected the correct key.
- etc

You must use **IllegalArgumentException** and **IllegalStateException** to enforce preconditions, and Java asserts for postconditions checks. Here are some examples how to do this (the methods are hypothetical):

```
// check a condition before a state change (precondition)
public void moveChap(Direction direction) {
    Point nextPosition = .. ;
    // precondition check
    if (isWall(nextPosition)) {
        throw new IllegalArgumentException("chap cannot be moved into a wall");
    }
    ...
}
```

```
// check a condition before and after a state change (pre- and postcondition)
public void pickupTreasure() {
    Point currentPosition = .. ;
    // precondition check
    if (!hasTreasure(currentPosition)) {
        throw new IllegalStateException("there is no treasure here: " + currentPosition);
    }
    int uncollectedTreasureCount = .. ;
    ... // pick up treasure here
    int uncollectedTreasureCount2 = .. ;
    // postcondition check
    assert uncollectedTreasureCount2 == uncollectedTreasureCount-1;
```

```
}
```

Unit tests (using JUnit 4 or 5) for this package are to be provided in the package `test.nz.ac.vuw.ecs.swen225.gp22.domain`.

App (package `nz.ac.vuw.ecs.swen225.gp22.app`)

The application module provides a Graphical User Interface implemented using swing through which the player can see the maze and interact with it through the following keystrokes:

1. CTRL-X - exit the game, the current game state will be lost, the next time the game is started, it will resume from the last unfinished level
2. CTRL-S - exit the game, saves the game state, game will resume next time the application will be started
3. CTRL-R - resume a saved game -- this will pop up a file selector to select a saved game to be loaded
4. CTRL-1 - start a new game at level 1
5. CTRL-2 - start a new game at level 2
6. SPACE - pause the game and display a "game is paused" dialog
7. ESC - close the "game is paused" dialog and resume the game
8. UP, DOWN, LEFT, RIGHT ARROWS -- move Chap within the maze

The application window should display the time left to play, the current level, keys collected, and the number of treasures that still need to be collected. It should also offer buttons and menu items to pause and exit the game, to save the game state and to resume a saved game, and to display a help page with game rules.

Note that the actual drawing of the maze is **not the responsibility** of the application module, but of the rendering module.

This module also manages a countdown -- each level has a maximum time associated with it (level 1 -- 1 min), and once the countdown reaches zero, the game terminates with a message informing the user, and then resetting the game to replay the current level.

The application package must include an executable class **`nz.ac.vuw.ecs.swen225.gp22.app.Main`** which starts the game.

Graphical user interfaces are notoriously difficult to test, so no unit tests are expected for this package. Testing is to be done manually.

Rendering (package `nz.ac.vuw.ecs.swen225.gp22.renderer`)

The rendering module is responsible for providing a simple 2-dimensional view of the maze, to be embedded in the application. It is updated after each move in order to display the current game play. If you use “actors” that move around freely (like bugs trying to eat Chap), then the renderer needs to update the maze view more often. Rendering should ensure that the movement of actors on the game board looks smooth. Develop and use a creative and consistent custom look and feel (in particular, use your own icons and sound effects).

Only a certain focus region of the maze should be displayed, check the original Chip's Challenge game for an idea about the size of this focus area. Rendering must also include sound effects.

Graphical user interfaces are notoriously difficult to test, so no unit tests are expected for this package. Testing is to be done manually.

Persistence (package `nz.ac.vuw.ecs.swen225.gp22.persistence`)

The persistence module loads game levels from XML files. The files defining levels are located in the **levels/** folder and follow the naming convention **levels/level1.xml**, **levels/level2.xml** etc. You are free to design the actual XML format (schema) to be used.

The game should have two levels, and level 2 should introduce a new actor (for instance, a bug that moves through the maze and tries to eat Chap). The logic of this actor, and how it is rendered is defined in classes and resources to be stored in jar files names **levels/level<some number>.jar** .

HINT: Utilities like **java.util.ServiceLoader** or **java.net.URLClassLoader** can be used to implement access to custom actors using a plugin-based design.

This module is also responsible for the save / resume functionality.

Unit tests (using JUnit 4 or 5) for this package are to be provided in the package `test.nz.ac.vuw.ecs.swen225.gp22.persistence`.

Recorder (package `nz.ac.vuw.ecs.swen225.gp22.recorder`)

The recorder module adds functionality to record game play, and stores the recorded games in a file (in xml format). It also adds the dual functionality to load a recorded game, and to replay it.

The user interface should have controls for replay: step-by-step, auto-replay, set replay speed. Note that this is different from the persistence module: here, not just the current game state is saved, but also its history (i.e., each turn or Chap and any other actors).

No unit tests are required for this module.

Fuzz Testing (package `test.nz.ac.vuw.ecs.swen225.gp22.fuzz`)

This module consists of (junit4 or junit5) a single test only, to be implemented in two methods `test.nz.ac.vuw.ecs.swen225.gp22.fuzz::FuzzTest::test1` and `test.nz.ac.vuw.ecs.swen225.gp22.fuzz::FuzzTest::test2`. These tests will generate some random input to invoke methods in `nz.ac.vuw.ecs.swen225.gp22.app`. In particular, those tests will **automatically (randomly, preferably enhanced with some intelligence)** play level 1 (test1) and level 2 (test2) of the game. Those tests do not need explicit JUnit assertions (i.e., invocations of JUnit `assert*` methods), instead, their purpose is to trigger exceptions or errors such as:

1. Precondition violations, such as `IllegalArgumentException` and `IllegalStateException`
2. General programming errors, such as `NullPointerException`
3. Postcondition or Invariant violations caused by asserts, i.e. `AssertionErrors`

Whenever such an error or exception is detected which crashes the application, a new gitlab issue should be opened (not necessary for precondition violations) and assigned to the student responsible for the module that causes the issue. Use a [label](#) `#detectedByFuzzer` to report such issues.

Both tests should run for 1 min max, the `timeout` annotation or `assertTimeout` functions in junit4 / junit5 can be used to enforce this. When running the tests with coverage, at least 60% line coverage in the domain module should be reached after both tests have completed.

Quality Assurance

We will measure the quality of the code using a range of well-known tools, which you should also use during development:

- Code Coverage. Code coverage means line coverage, it will be measured using the EMMA / Jacoco tool (plugins for Eclipse and IntelliJ are available) and test cases you have developed.
- SpotBugs (plugins for Eclipse and IntelliJ are available). Code smells will be measured using the SpotBugs tool, with marks deducted for issues raised. The threshold for issues is rank 15 across all bug categories (i.e. consider scary and troubling, but ignore of concern), with the minimum confidence to report set to medium. You should ensure all bug categories are enabled in SpotBugs¹.
- IDE Reference Browser / Dependency Analysis. We will check module / package dependencies using the Eclipse reference browser (for a given package, go to References > Project in the context menu, IntelliJ provides a similar *Find Usage* utility). The architecture overview figure above indicates permitted and prohibited dependencies using the direction of arrows representing dependencies. For instance, the renderer component depends on the domain component, but not vice versa. Dependencies here means compile-time dependency, i.e., if A depends on B, A cannot be compiled without B.

NOTE: You should enable the above tools in Eclipse or IntelliJ so that they are run continuously on your code base. Otherwise, there will be a lot of work left to do at the end. Optionally, you could invest some time to set up build scripts using ant, maven or gradle, and configure those tools as plugins. This is the most efficient approach, but is not expected for SWEN225.

Submission and Assessment

There are several group and individual elements involved in the assessment of your project. We now look at these in more detail.

¹ In Eclipse, you can configure the SpotBugs plugin in the preferences: Java > SpotBugs

Integration Day

On or before Integration Day your team must demonstrate a working program to one of the lab tutors during the lab slots. This is Lab 5, the lab times scheduled for Lab5 will be used, and this will be marked as Lab5. We will inform teams about their presentation slot the week before.

The program must demonstrate functionality from each of the main modules of the system, however, it does not need to be complete. A particular challenge is that most parts of the application depend on the domain module providing the domain model for the application. Initially, this module does not have to be complete. There should be an application window containing some buttons and the rendering window; the rendering window should be able to draw the game world view to some degree; the persistency package should be capable of reading files in the basic file format given for level 1; and, finally, there should be some evidence of the game state and game logic. The fuzz module should have at least one test that randomly calls methods in the application module, and the recorder module should at least record moves made in a file (but not necessarily be able to replay them).

Other possible simplifications acceptable for the integration day version include: no automated test cases, simplified game play (e.g., logic and rendering for locked doors is missing, treasures are not being picked up), no contract checks (pre- , postconditions), timeouts not implemented, the entire game is rendered (no focus area), etc. The focus is on the demonstrable integration and interaction between modules.

In other words, we want to check whether [the marshmallow is on top](#).

Program Code

The program code and JUnit tests should be stored in the repository, the source code should be in the **src/** folder. You may (but don't have to) use a separate **src-test/** folder to separate test from production code. Test classes should have names ending with "**Test**".

Packaging. Your program code must comply with the package structure stipulated above (packages corresponding to modules).

Documentation Code should be well-documented internally with the name and student ID of the author(s) clearly marked at the top of each Java file using JavaDoc conventions. At least all classes and public methods and fields should have meaningful comments describing their functionality.

Instructions. Your program code should include a **README.md** (in the root folder of the eclipse project) which details exactly how to start up your game and provides any necessary information on how to play the game (this information is essential for the markers). The instructions should be kept simple, for instance, a single sentence “Start the game by running **nz.ac.vuw.ecs.swen225.gp22.app.Main** from Eclipse” could be sufficient.

Compatibility. Your code must work on the ECS lab machines and you risk being heavily penalised if this is not the case. In particular, the markers must be able to run your program easily on ECS machines without compatibility issues. The Java version supported should be Java 17.

Repository Use

A gitlab repository will be created and provided after the team signup phase. **The use of this (and only this) gitlab repository is mandatory**, projects not using **this** repository will not be marked and all team members will receive zero marks for this assignment. See the marking guidelines for levels of expected repository usage. **A penalty will be applied if repositories related to this project are detected on GitHub, bitbucket or similar code sharing platforms.**

It is expected and will be assessed that you use the issue tracking system to track bugs and plan features.

Individual Report

Your individual report must be submitted by the deadline. The report itself should be private — you should not consult with your team members over the report. The report must be submitted as a PDF using the submission system, and consist of:

1. A statement explaining your individual contribution to the project, using a table with the following two columns: commit-urls, description. List 3 commit URLs. The commits should be for the module assigned to you. These commits should contain your most substantial/important contributions.
2. A ranking of the contribution of each person in the team. To do this, list the name and ECS email address of each team member (**including yourself**) and give them a score out of 5 (5 is best, 1 means no contribution) for their contribution. For example:
 - a. Sue Student sams@ecs.vuw.ac.nz 3
 - b. Lazy Larry larry@ecs.vuw.ac.nz 1
 - c. Busy Bernice berniceb@ecs.vuw.ac.nz 5
 - d. Dave Dedicated daved@ecs.vuw.ac.nz 5

3. A short reflection essay (1-2 pages). This should discuss the following issues (use those questions as section headers):
 - a. What knowledge and/or experience have I gained?
 - b. What were the major challenges, and how did we solve them?
 - c. Which technologies and methods worked for me and the team, and which didn't, and why?
 - d. Discuss how you used one particular design pattern or code contract in your module. What were the pros and cons of using the design pattern or contract in the context of this project ?
 - e. What would I do differently if I had to do this project again?
 - f. What should the team do differently if we had to do this project again? =

Submission

Group Project

Where to submit: https://apps.ecs.vuw.ac.nz/submit/SWEN225/Group_Project

What to submit: GitLab Group Project (from SWEN225/Group space)

Group Project Individual

Where to submit: https://apps.ecs.vuw.ac.nz/submit/SWEN225/Group_Project_Individual

What to submit: GitLab Group Project (from SWEN225/Assignments space)

1. A file **top-commits.txt** containing the URLs of the three top commits you have made. If this file is missing, your three most recent commits will be used to assess your work. See discussion in the **Individual Report** section -- those should be the same commits.
2. A file **contribution-rankings.txt** containing a ranking of the contributions of each person in the team.
3. The individual Reflection report **individual-report.pdf**. PDF only. Other formats like doc (MSWord) will be ignored and marked zero!!

Late Penalties

The lectures and assignments will have given you the opportunity to develop experience in design and programming, and the deadline has been set to give you time to complete this project. Accordingly, you should have no problem completing this project on time. Furthermore, as the final deadline is already as late as possible and arrangements for marking are very tight, extensions can only be given in exceptional circumstances. If you do not submit on time without prior consent you should expect to receive no marks for the project. **Note that it is not possible to use any late days for the group submission.**

Marking Guide

What follows is a detailed marking guide for the main components of the project.

Marking Component	Mark	Notes
Group Mark: playability	9	The program is playable. Marks will be deducted for minor faults.
Group Mark: gitlab	3	The GitLab issue tracking system is used to plan projects, and assign tasks, including bugfixes. A Gource (https://gource.io/) video showing files and group member commits
Group Mark: documentation	3	The program contains up-to-date javadocs with embedded generated UML programs in the docs/ folder. These UML models can be generated with https://www.yworks.com/products/ydoc or https://github.com/talsma-ict/umldoclet (the latter will work better if you use Java 17 language features).
Individual Mark: use of Git (all team members)	4	Level of use will be assessed: <i>poor</i> -- only few commits (<4, or all within one week) of large chunks of code <i>satisfactory</i> -- several commits (<8 over at least two weeks), most commits have meaningful comments. <i>good/excellent</i> -- frequent commits (>=8 over several weeks), all commits have meaningful comments.
Individual Contributions -- Quality of Code - Module Specific	4	See below for details
Individual Contributions -- Quality of Code - Generic	4	Absence of bugs reported by spotbugs in the module. Absence of non-permitted outgoing dependencies in a module (see dependency structure above). Classes and public methods are documented using the Javadoc standard.
Individual Reflection Report	3	Individual Reflection Report

NOTE: We reserve the right to apply a scaling factor to the group mark for individual team members if the peer- and self review indicates that a team member has not sufficiently contributed. We also reserve the right to reduce individual marks if other team members have committed significant contributions to modules not assigned to them. Penalties may be applied to both the module owner, and the committer. In any case, such commits should be avoided, and if necessary (e.g., when refactoring code), the commit message should contain a justification.

Module-Specific Quality Criteria

Module	Check
domain	<ul style="list-style-type: none">• test coverage of domain package (running tests in this module only), expected to be at least 80%• usage or pre- and post condition checks
app	<ul style="list-style-type: none">• visual appeal (modern, consistent UI)• timeout functionality• consistency
renderer	<ul style="list-style-type: none">• originality (more than just a clone of the existing game)• visual appeal• consistency• smoothness of transitions• sound effects
fuzz	<ul style="list-style-type: none">• test coverage of domain package (running tests in this module only)• use of randomness and strategy to generate input, coverage after 1 min should reach 60% (using a standard ECS computer) for any test run• bonus: reported issues with bugs discovered
persistence	<ul style="list-style-type: none">• use of xml• satisfactory performance or load / save (< 1s)• test coverage of persistence package (running tests in this module only), expected to be at least 80%
recorder	<ul style="list-style-type: none">• features implemented (manual tested):• step-by-step• auto-replay• set replay speed

Note: the coverage metric to be used is instruction coverage. This is the Emma default. In the coverage report settings, you can select the coverage criteria to be used.

Use of External Libraries

Only the following libraries may be used. They must be kept in the project **lib/** folder.

1. <https://mvnrepository.com/artifact/com.google.guava/guava/30.1.1-jre> and any library it depends on (for general utilities)
2. <https://mvnrepository.com/artifact/commons-io/commons-io/2.11.0> and any library it depends on (for general utilities)
3. <https://mvnrepository.com/artifact/org.jdom/jdom2/2.0.6> and any library it depends on (for XML parsing and generating)
4. <https://mvnrepository.com/artifact/org.dom4j/dom4j/2.1.3> and any library it depends on (for XML parsing and generating)
5. JUnit4 or JUnit5 (junit3 is not acceptable) -- support for JUnit is available within IDEs , but you can also use libraries from these groups:
<https://mvnrepository.com/artifact/org.junit.jupiter> (JUnit5) or
<https://mvnrepository.com/artifact/junit/junit> (JUnit4)
6. <https://mvnrepository.com/artifact/org.mockito> -- any library within this group (for testing)
7. <https://mvnrepository.com/artifact/org.json/json> - for JSON processing (but note that XML should be used!)
8. <https://mvnrepository.com/artifact/com.google.code.gson/gson> - for JSON processing (but note that XML should be used!)
9. <https://mvnrepository.com/artifact/com.fasterxml.jackson> (any in this group) - for JSON processing (but note that XML should be used!)
10. <https://mvnrepository.com/artifact/com.alibaba/fastjson> - for JSON processing (but note that XML should be used!)

This is a whitelist. Please seek permission to use a particular library, and we will consider extending this list.

Penalties

The following will be penalised:

1. Use of non-whitelisted external libraries.
2. Code cannot be compiled with the Java compiler version 17
3. Program cannot be started or crashes when executed using Java version 17
4. Use of code from other projects (we will use [Moss](#) to detect clones)

In extreme cases (such as: project not compilable, and there is no obvious way of fixing this to get the game running) this can lead to all team members getting zero marks.

Frequently Asked Questions

What does smooth mean here: “.. movement of actors on the game board looks smooth”?

This does not mean that movement needs to be animated with a high frame rate. But what is expected here is that there is some basic animation, and the board is updated after each keystroke ahead of new input, i.e. the game should not get stuck for an observable amount of time, compromising the player experience.

My commits show up under different names, and I am concerned that my contributions are incorrectly assessed, what can I do to avoid this?

Please use only a single name, and please don't use cryptic aliases. In a git client, you can configure the name to be used. You can test this with a dummy commit (use something like “dummy commit to test user name” as commit message). Note that a different name might be used if you make changes directly on the gitlab webpage. It is highly recommended to avoid this anyway for another reason -- this is likely to create conflicting changes with other team members manipulating the same file in a local copy which you later have to resolve.

In the fuzzing module, what does “.. preferably enhanced with some intelligence” mean?

An example would be to guide this by coverage. Say you are in a certain position and could go left, right, top and down. You keep a record of which position you have already explored, and you did not go up yet. So intelligence would just mean here to prioritise choosing up as your

next move. By the way, coverage-driven fuzzers like American fuzzing lop (AFL) are based on this overall idea.

Additional Notes, Clarifications and Corrections

You are expected to check the following notes, and Discord group project channel posts frequently. Use Discord to ask questions and seek clarifications.

[illegible]