Victoria University of Wellington

SWEN 225 Software Design

Group Project 2020: Chap’s Challenge

Worth 45% of Overall Mark

Important Dates	
Team Signup Deadline:	11 Sept 2020
Integration Day Presentations:	28 Sept - 2 Oct 2020 (scheduled and marked as Lab5)
Submission Due Date:	21 Oct 2020
Blackboard Quizz:	23 Oct 2020 (will be open for 24h)

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 11 (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 Friday 11 September @ Midnight 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 or 3 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.

The Group Project will be conducted under the group work policy. 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.

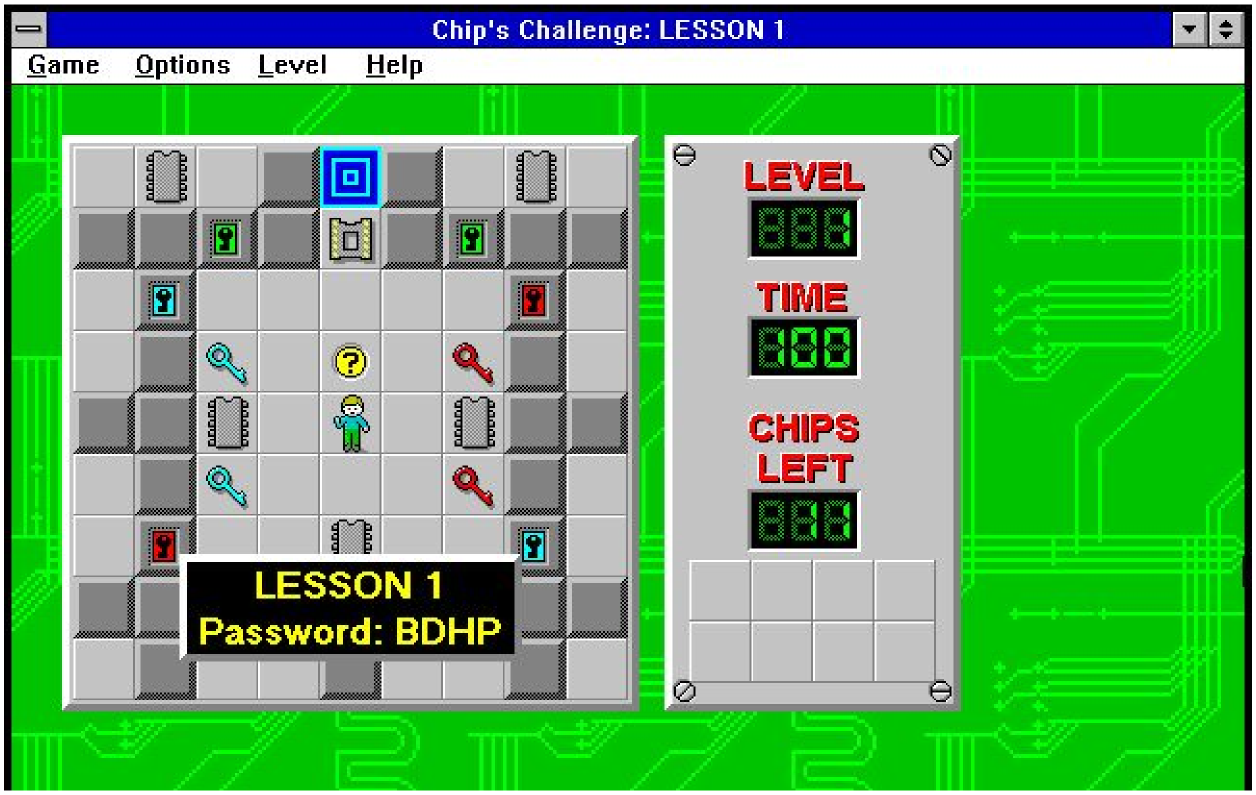
NOTE: While this is a team project, we will mainly assess individual contributions.

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

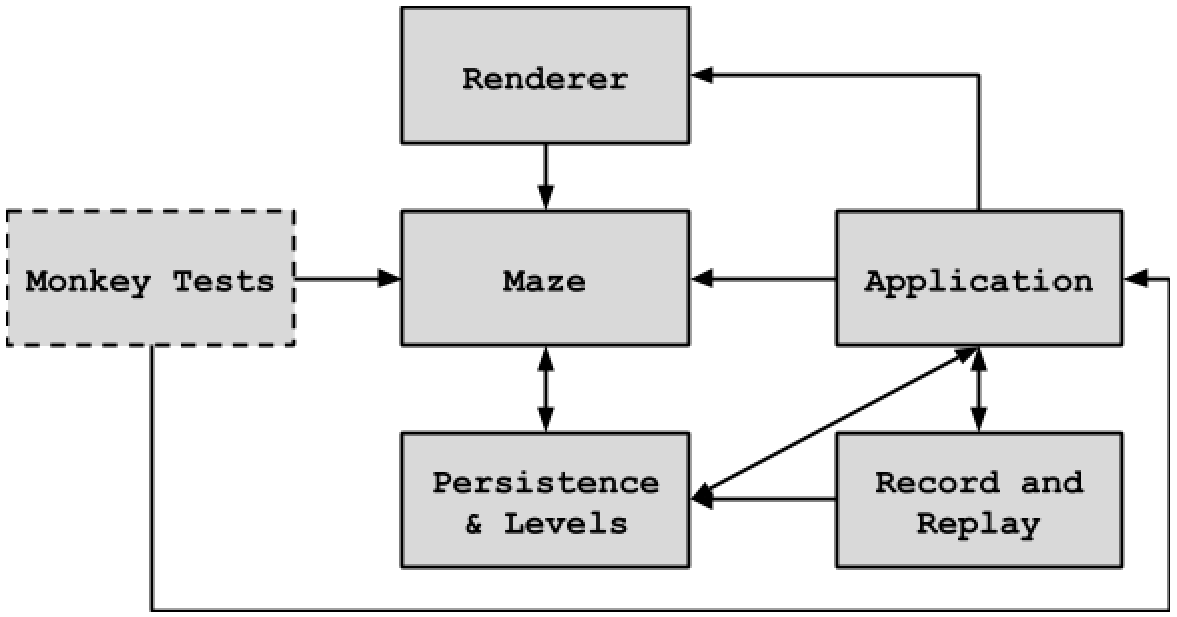
Chap's challenge is a ***creative*** clone of the (first level of the) 1989 Atari game Chips Challenge. To learn more about Chip's Challenge, please read the Wikipedia article, watch a playthrough on youtube, or even try to get hold of the actual game (there are some free browser-based versions, and 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 four core 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 **Maze** module. This module is responsible for representing and maintaining the state of the game, such as what type of objects are on the maze, where these objects are, and which actions are allowed to change the state of those objects.
2. The **Application** 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). Use input in 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, including transitions, animations, focus areas and sound.
4. The **Persistence** module. This module is responsible for reading map files and reading/writing files representing the current game state (in JSON format) in order for the player to resume games.
5. The **Record and Replay** module. This module records games, and allows a user to load and replay games.

6. The **Monkey Test** module. This module can load a game level, and play the game using randomly generated input. The purpose of this module is to detect violations of the game logic. **This component only has to be implemented by teams of six.**

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** in the root folder of the repository that contains a table with the following structure:

Module	**Team Member Name**	**Team Member gitlab account**
maze	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”.

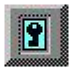
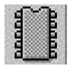
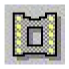
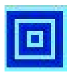
Each of these components will now be considered in more detail.

The Maze Component (package nz.ac.vuw.ecs.swen225.gp20.maze)

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, he 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, and Chap keeps the key.
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, he 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 persistence 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 he reaches the exit and then advances to the next level (if there is another level).
This module should make extensive use of contracts to ensure the integrity of the maze. Use explicite pre- and postconditions to check for state invariants such as:

- Chap can never stand on a tile
- 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 he must have collected the correct key.
- etc

You must use Guava Preconditions API for preconditions, and Java asserts for postconditions and invariant checks.

Unit tests (using JUnit 4 or 5) for this package are to be provided in the package test.nz.ac.vuw.ecs.swen225.gp20.maze.

Application (package nz.ac.vuw.ecs.swen225.gp20.application)

The application module provides a Graphical User Interface 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
4. CTRL-P - start a new game at the last unfinished level
5. CTRL-1 - start a new game at level 1
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 (such as 1 min), and the once the countdown reaches zero, the game terminates with a group dialog informing the user, and resetting the game to replay the level.

The application package must include an executable class **nz.ac.vuw.ecs.swen225.gp20.application.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.gp20.render)

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 the game board looks smooth. Develop and use a creative 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 should 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 and Levels (package nz.ac.vuw.ecs.swen225.gp20.persistence)

The persistence module loads game levels from JSON files. The files defining levels are located in the **levels/** folder and follow the naming convention **levels/level1.json**, **levels/level2.json** etc. You are free to design the actual JSON 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.

Unit tests (using JUnit 4 or 5) for this package are to be provided in the package test.nz.ac.vuw.ecs.swen225.gp20.persistence.

Record and Replay Games (package nz.ac.vuw.ecs.swen225.gp20.recnplay)

The record and replay module adds functionality to record game play, and stores the recorded games in a file (in JSON format). It also adds the dual functionality to load a recorded game, and to replay it. The user should have controls for replay: step-by-step, auto-reply, 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.

Monkey Testing (package test.nz.ac.vuw.ecs.swen225.gp20.monkey)

This module consists of (junit4 or junit5) tests only. The tests will generate some random input to call public methods in nz.ac.vuw.ecs.swen225.gp20.application and/or nz.ac.vuw.ecs.swen225.gp20.maze. In particular, those tests will **automatically (randomly, preferably enhanced with some intelligence)** play the game in “headless mode” (i.e., the display of the board is not required). Those tests do not need explicit JUnit assertions (i.e., invocations of JUnit assert* methods), instead, their purpose is to trigger exceptions or errors in nz.ac.vuw.ecs.swen225.gp20.application, such as:

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

Whenever a general programming error or postcondition or invariant violation is detected, a new gitlab issue should be opened (not necessary for precondition violations) and assigned to the student responsible for the nz.ac.vuw.ecs.swen225.gp20.application package.

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 will be measured using the EMMA tool and test cases you developed.
- SpotBugs. 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, with the minimum confidence to report set to medium. You should ensure all bug categories are enabled in SpotBugs[1].
- JavaDoc. JavaDoc violations will be measured using the Eclipse JavaDoc analysis, and marks will be deducted for issues raised. You should set the severity level for malformed/missing Javadoc comments and tags to Error in Eclipse, and validate all standard tags[2].
- IDE Reference Browser. 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 maze 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 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 best 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
[2] In Eclipse, you can configure JavaDoc checks in the preferences: Java > Compiler > Javadoc

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 particulate challenge is that most parts of the application depend on the maze 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 persistence 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.

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), timeout not implemented, the entire game is rendered (no focus area), etc. The focus is on the demonstrable 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 should comply to 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.gp20.application.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 11.

Repository Use

A gitlab group will be provided after the team signup phase, and the group can then create the repository to be used. The use of ***this* git 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.

You should 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 at most 3 commit URLs per row, and limit this to 5 rows. The commits should be for the module assigned to you.
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

The following should be submitted using different submission sites:

1. The individual report **individual-report.pdf**.
2. A file **checkout.txt** stating the URL of the repository and the name of the tag to checkout for marking. If no tag is provided, the HEAD is used, and late submission penalties will be applied if the timestamp of the last commit is after the submission deadline.

For instance, if the **checkout.txt** had the following content:

```
repo url: https://gitlab.ecs.vuw.ac.nz/foo
tag to mark: SUBMITTED
```

Then we would acquire the project for marking by running the following commands:

```
git clone https://gitlab.ecs.vuw.ac.nz/foo
git checkout tags/SUBMITTED
```

NOTE: it is your responsibility to keep **checkout.txt** submissions consistent across team members. If checkout.txt files submitted by different team members differ, we will use ***any*** for marking. This is important if you decide to update your submission.

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 in 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	4	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.
Group Mark: documentation	2	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 .
Individual Mark: use of Git (all team members)	8	Level of use will be assessed: *poor* -- only few commits (<4, or all within one week) of large chunks of code *satisfactory* -- several commits (<8 over at least two weeks), most commits have meaningful comments. *good* -- frequent commits (>=8 over several weeks), all commits have meaningful comments. *excellent* - "good" level plus uses of branching and merging to add new features.
Individual Contributions -- Quality of Code - Module Specific	5	See below for details
Individual Contributions -- Quality of Code - Generic	5	Absence of bugs reported by spotbugs in the module. Absence of non-permitted outgoing dependencies in a module. Classes and public methods are documented using the Javadoc standard.
Blackboard Test	8	A quiz with questions related to the assignment, in week 13.

Reflection Essay	10	Individual Reflection Essay

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**
maze	● test coverage of maze package (running tests in this module only), expected to be at least 80% ● usage or pre- and post condition checks
application	● visual appeal (modern, consistent UI) ● timeout functionality
renderer	● originality (more than just a clone of the existing game) ● visual appeal ● smoothness of transitions ● sound effects
monkey	● test coverage of maze package (running tests in this module only) ● use of randomness and strategy to generate input, expected to be at least 60% ● bonus: reported issues with bugs discovered
persistence	● use of standard json ● performance ● test coverage of persistence package (running tests in this module only), expected to be at least 80%
record and replay	● features implemented (manual tested): ● step-by-step ● auto-reply ● 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://repo1.maven.org/maven2/javax/json/javax.json-api/1.1.4/javax.json-api-1.1.4.jar and libraries it depends on. See https://stackoverflow.com/questions/47022653/provider-org-glassfish-json-jsonprovideri mpl-not-found-at-javax-json-spi-jsonpro/47035781 for details.
2. https://repo1.maven.org/maven2/com/google/code/gson/gson/2.8.6/gson-2.8.6.jar
3. https://repo1.maven.org/maven2/com/fasterxml/jackson/core/jackson-databind/2.11.2/jackson-databind-2.11.2.jar and libraries it depends on.
4. https://mvnrepository.com/artifact/com.google.guava/guava/29.0-jre and libraries it depends on. This library is to be used for enforcing pre-conditions, but also provides a wide range of useful utilities and data structures.

Penalties

The following will be penalised:

1. Use of non-whitelisted external libraries.
2. Code cannot be compiled with the Java compiler version 11
3. Program cannot be started or crashes when executed using Java version 11

Additional Notes, Clarifications and Corrections

You are expected to check those notes, and forum posts frequently.

Date	**Note**
7 Sept 20	Changes Java 8 requirement to Java 11 to make this consistent with default JDK version in ECS Labs.
7 Sept 20	Added note to marking guide how to generated UML models.
9 Sept 20	Clarified packages / modules monkeytests can depend on.

10 Sept 20	Clarified coverage expectations for persistence module
11 Sept 20	Added note to clarify that coverage means instruction coverage.
16 Sept 20	Added clarification that after signup we will create the group, but groups have to create the actual repository.
21 Sept 20	Added permitted dependency, added FAQ list (see below)
22 Sept 20	In table describing team.md, fixed typo: github > gitlab

Frequently Asked Questions

Q: What is the reason to impose dependency constraints between modules ?

A: Without dependency constraints, software quickly evolves into a "big ball of mud" with a large amount of interdependencies and coupling that makes future changes expensive or even impossible. For instance, you might want to replace the renderer or application modules with modules to run the game on a mobile device. This should be possible without changing the maze and persistence modules.

Q: How can I break a dependency between two modules A and B?

I use the application (APP) and the record and replay (RRP) module as examples. In the assignment, they can depend on each other, but it is actually possible to avoid this. APP needs to display some controls (buttons) for the recording and the replay functions. So the actions triggered by clicking those buttons invoke code in RRP, adding a dependency APP -> RRP. But you could avoid this and generally minimise dependencies using the following design:

APP accepts "plugins". A plugin is a provider of actions (a list of actions, instances of some Action type you provide). Those actions have some functionality (a method performed representing the action, such as "run"), and some information such as a name, an icon, a help desk, perhaps some state whether they are available or not.

When the application starts, it looks for available plugins (for instance, using some registry, using a static variable is a simple way to make this work), and when the main application window is built, buttons are added for the actions of each available plugin. This design avoids a dependency APP -> RRP. There is still a dependency from RRP -> APP as RRP uses the Action interface (that would be defined in APP). This could be avoided by creating some independent (and minimalistic by design) “commons” module containing Action other modules depend on (instead of depending on each other).

Some ideas for further reading: Eclipse plugins use a design very similar to what was just described. JDBC (the Java framework to connect to relational databases) also uses a similar design, with “drivers” being the plugins. A particular interesting aspect is how the “plugin registry” is designed (java.sql.DriverManager, it is also worth looking into older JDBC versions that used a different design).

Q: What kind of functionality can be in the commons module / package mentioned in the last answer ?

Examples include:

1. Common data structure, such as a Pair class that was suggested by a team, see also https://www.baeldung.com/java-pairs for a discussion
2. An enum containing defining constants for possible moves, this could then be used by multiple other modules

Q: The handout indicates the board should be drawn onto a canvas. Does this mean we have to specifically use the Canvas class or can we use something like a JPanel to display it instead?

You dont have to use a canvas class — a JPanel (for instance, with a grid layout of some other swing components) is also ok.