Victoria University of Wellington

SWEN 225 Software Design

Group Project 2019: Chap's Challenge

Worth 20% of Overall Mark
Due Date: 11 October 2019

Integration Day Presentations: 23 - 27 September 2019 (scheduled and marked as Lab 5)

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. 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.

NOTE: You are not permitted to use external dependencies (i.e. libraries) for completing this project. Your final solution must run on a stock installation of Java 8 (e.g. Oracle or OpenJDK inc. JavaFX) without the need to download any external dependencies. This limits the scope of the project, and gives everyone a level playing field.

You should undertake this project in teams of 4-6 people. 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 6 September @ Midnight, 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 4-6 students. Anyone not in a team will be automatically added to a team after the team signup system has closed.

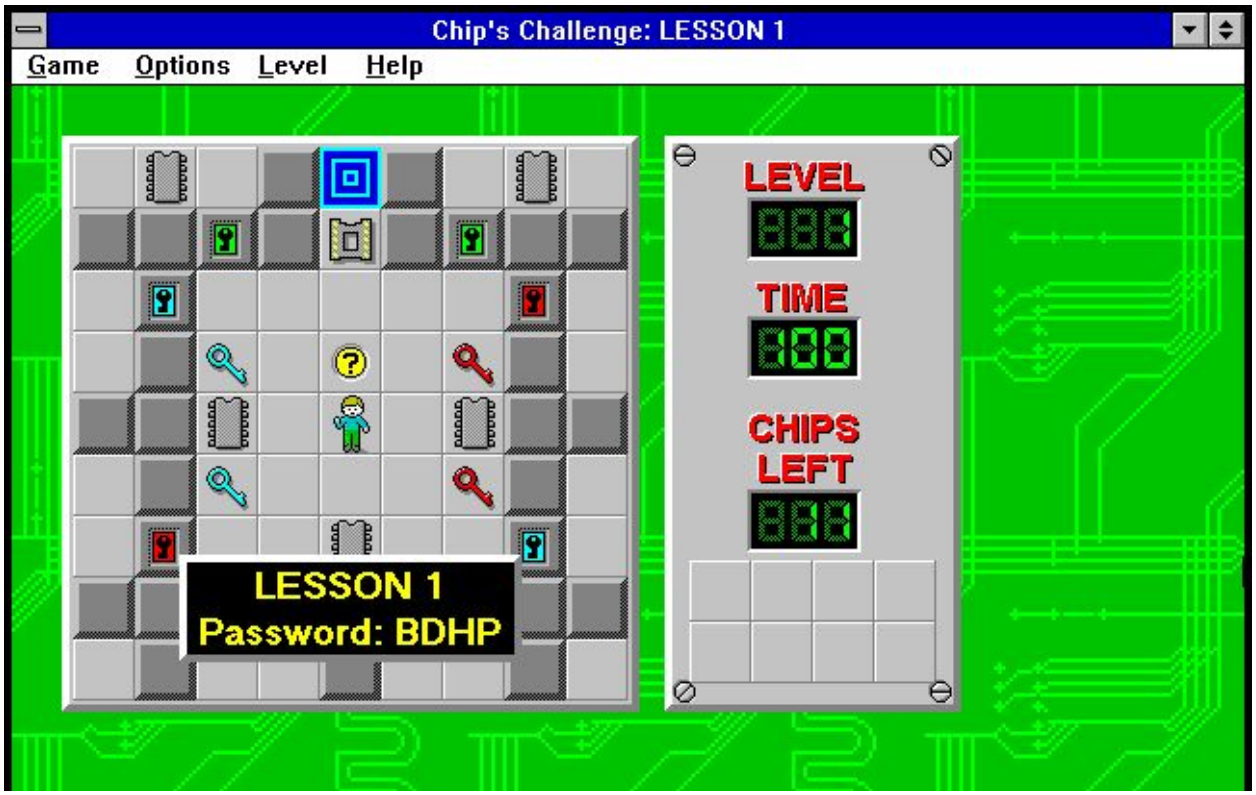
The Group Project will be conducted under the group work policy. We understand there are sometimes difficulties when working in teams, because of personality conflicts, and the pressures of time and workload. You will be able to resolve some such difficulties yourself, but we are willing to help. If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

Requirements

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

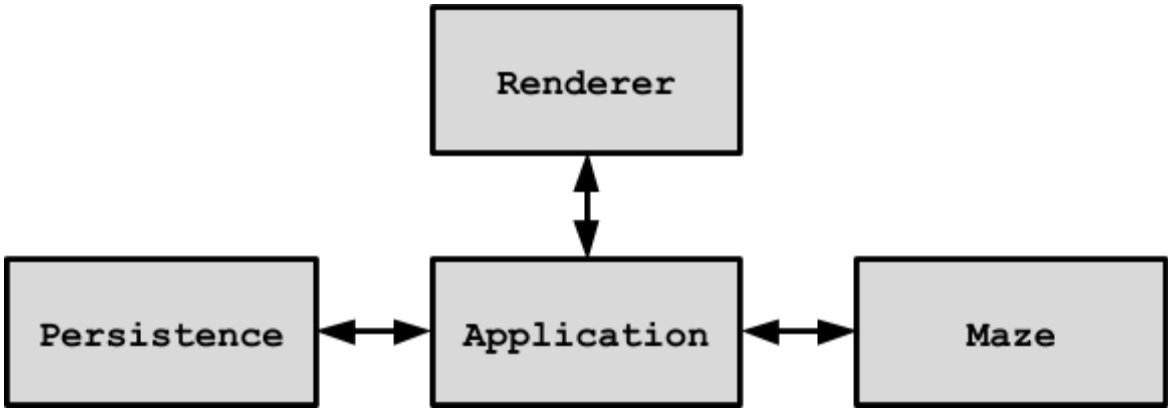
Chip and Chap

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 web-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:



Architecture

The game should adopt high-level architecture consisting of four core components, as illustrated in the following figure. The figure also shows component dependencies and interactions.



The four core components of the design are:

- The **Maze** package. This is responsible for maintaining the current state of the game, such as where the objects in the game currently are, and also for determining what actions are allowed within the game.
- The **Application** package. This 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).
- The **Renderer** package. This is responsible for drawing the maze onto a canvas and mapping key strokes on the canvas back to objects in the game world, etc.
- The **Persistence** package. This is responsible for reading map files and reading/writing files representing the current game state (in JSON format).

In addition to the above, there are two advanced components. Those components are more difficult to implement, and teams should carefully consider the resources they devote to the implementation of those features.

It is recommended that members of your team should choose one component to be responsible for. Each of these components will now be considered in more detail.

The Maze (package nz.ac.vuw.ecs.swen225.a3.maze)

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 has a unique maze. Level 1 should have all the tiles listed in the table above, and at least two different types (colours) of keys. Develop and use a creative custom look and feel (in particular, use your own icons). 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 below). 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, and are not directed by user input.

The maze package is 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 advances to the next level (if there is another level). Levels have a timeout, if the timeout is reached before Chap reaches the exit, the level stops, and the player gets the option to restart the game at the current level.

Application (package nz.ac.vuw.ecs.swen225.a3.application)

The application should provide a Graphical User Interface through which the player can see the maze and interact with it through the following key strokes:

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, 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 drawing of the actual maze is not the responsibility of the application component, but of the rendering component.

The application package must include an executable class **nz.ac.vuw.ecs.swen225.a3.application.Main** which starts the game.

Rendering (package nz.ac.vuw.ecs.swen225.a3.render)

The rendering window is responsible for providing a simple 2-dimensional view of the maze, 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. For larger boards, only a certain focus region of the board should be displayed, check the original Chip's Challenge game for an idea about the size of this focus area.

Persistence (package nz.ac.vuw.ecs.swen225.a3.persistence)

The persistence module saves the game state: the time left, and the position of Chap and other actors (if there are any), the treasures he holds, etc. Game state should be saved using the JSON format. Ensure (by means of testing or inspection) that the JSON format used is valid.

Advanced: Record and Replay Games (package nz.ac.vuw.ecs.swen225.a3.recnplay)

This is a design challenge suitable for teams aiming for A grades. The record and replay module adds functionality to record game play, and store the recorded game 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. You should also integrate this into testing: by creating integration tests as JUnit test **nz.ac.vuw.ecs.swen225.a3.recnplay.IntegrationTest** that executes recorded games, and check for the validity of game state (e.g., that Chap does not stand on a wall).

Advanced: Levels as Plugins (package nz.ac.vuw.ecs.swen225.a3.plugin)

This is a design challenge suitable for teams aiming for A grades. In this challenge, create a design where levels are self-contained plugins / dropins. Each level is completely defined by a file **level-k.zip** in a “**levels/”** folder, with k being the number of the level. The level-*.zip files contain all the resources and classes (tiles with their icons and classes implementing tile behaviour) required, and there are no dependencies between classes. This should be demonstrated by enabling level 2 by dropping level-2.zip into the **levels/** folder.

NOTE: this is similar to how Eclipse and other plugin models work. Java features (available in the standard class library) like **java.net.URLClassLoader** and **java.util.ServiceLoader** can be used to implement this.

Quality Assurance

We will measure the quality of the code using a range of well-known tools:

- Code Coverage. Code coverage will be measured using the EMMA tool and test cases you developed, with a target of 75% coverage being expected.
- 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].

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.

We also require that you use code contracts to ensure the correctness (in particular the integrity of the game state) of your code. The following constructs should be used to enforce contracts:

- conditional runtime exceptions for precondition checks. It is acceptable (but not required) to you create your own utility class for precondition checks in order to increase the readability of code. Check this blog post for ideas.
- assertions for postcondition and invariant checks.

NOTE: Assertion checking will be enabled when the project is being executed for marking.

[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

Submission and Assessment

There are several elements involved in the assessment of your project:

- Integration Day presentation.
- design documents.
- the actual program code.
- an individual report.

We now look at these in more detail.

Integration Day

On or before Integration Day (Friday 27th September @ Midnight) 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. All team members are expected to attend.

This program must demonstrate functionality from each of the main components of the system, however it does not need 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; and, finally, there should be some evidence of the game state and game logic.

Design Documents

The following documents should be generated and stored in the repository (the folder structure is with respect to the project root)

1. A UML class diagrams for the whole program (**docs/design/uml.pdf**)
2. The JavaDocs generated using the ydoc doclet (in **docs/javadoc/**)
3. A document **docs/plugins.pdf** or **docs/plugins.md**, with a description on how levels as plugins were implemented. This should include a discussion of design patterns and frameworks that have been used.
4. A document **docs/recnplay.pdf** or **docs/recnplay.md** describing the design of the record and play functionality. This should include a discussion of design patterns and frameworks that have been used.

NOTE: The respective commits are contributions which team members can list in their individual report.

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). It should be well-documented internally with the name and student ID of the author(s) clearly marked at the top of each Java file. This must be consistent with the commit history, i.e., there is an expectation that most commits will be by the team member responsible for the module.

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.a3.application.Main** from Eclipse" couldbe 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 markersmust be able to run your program easily on ECS machines without compatibility issues. The Java version supported should be Java 8.

Repository Use

A gitlab repository will be provided after the team signup phase. 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. It is highly recommended to also 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, and consist of:

1. A statement explaining your individual contribution to the project, using a table with the following three columns: module-name, description, commit-urls. List at most 3 commit URLs per row, and limit this to 5 rows.
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 4.
    d. Dave Dedicated daved@ecs.vuw.ac.nz 5
3. A short reflection essay (500-1000 words). This should discuss the following issues:
    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. What would I do differently if I had to do this project again?
    e. What should the team do differently if we had to do this project again?

NOTE: the ranking (self- and peer assessment) will not be used directly for marking other team members. However, the information provided may trigger a more detailed investigation into the contribution of each team member based on their respective commits, and this will affect the scaling factor applied to the group mark, for instance so that team members who suffered from lack of commitment by other members won't be penalised by proxy.

Submission

The following should be submitted (using separate 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.

Task	**Marks**	**remarks**
1. Group Mark	**90**	**A scaling factor will be applied if your contributions will be deemed as insufficient (based on the inspection of commits, and potential investigations triggered by peer- and self reviews).**
1.1 Design	10	UML model will be assessed, checking it for consistency and completeness, and compliance to the UML standard.
1.2 Basic Functionality	20	This will be assessed by playing the first two levels of the game, and establishing whether it satisfies the requirements.

1.3 Tests	10	Tests should succeed, have meaningful assertions and high coverage (at least 75%).
1.4 Quality - others	5	There should be no bugs reported by SpotBugs.
1.5. Documentation	5	The project readme should be precise and of satisfactory quality (grammar, spelling, pstyle), and code should have informative comments, and javadoc built using the ydoc doclet should be supplied.
1.6 Advanced Functionality: Levels as Plugins	10	The functionality will be assessed following the instructions supplied in the README. The design documentation is part of the assessment.
1.7 Advanced Functionality: Record and Replay	10	The functionality will be assessed following the instructions supplied in the README. The design documentation is part of the assessment.
1.8 Use of Git	10	Level of use will be assessed: **No use** -- only few commits of large chunks of code at the end of the project made by some team members. **Satisfactory** -- regular commits by all team members, most commits have meaningful comments. **Good** -- frequent commits by all team members, all commits have meaningful comments, tags are used to mark milestones **Excellent** -- “Good” level plus uses of branching and merging to add new features, issue tracking is used for project management.
2. Individual Mark	**10**	Reflection Essay.

Penalties

The following will be penalised:

1. Use of external libraries
2. Code cannot be compiled with the Java compiler version 8
3. Program cannot be started or crashes when executed using Java version 8

Additional Notes, Clarifications and Corrections

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

Date	**Note**