Programming in the Large (CSSE2002)
Assignment 2 — Semester 1, 2024
School of EECS
The University of Queensland

Due May 13th 13:00 AEST

> You can, if you want, rewrite forever.
> — Neil Simon

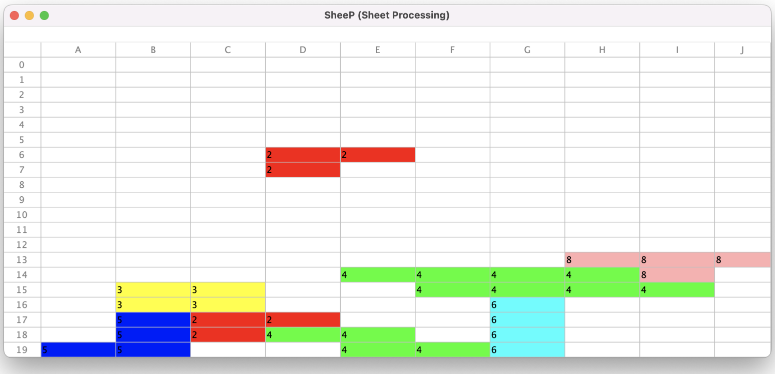
Do not distribute. Revision 1.0.0

Overview This assignment delivers experience working with an existing software project. You are provided with a small extension to the SheeP spreadsheet program. The extension has introduced bugs and is poorly written. Fortunately, the extension includes a suite of JUnit tests.

You will be evaluated on your ability to:

- find and fix errors in provided code,
- extend the functionality of provided code, and
- and meaningfully refactor supplied code to improve its code quality.

Task Your program, SheeP, was a huge success. However, a few users felt it was a bit boring. You decide to make the product more exciting by introducing new games into the spreadsheet program. The course staff have tried to add one mini-game, but have made a mess of it. Your task is to fix their mistakes and finish the implementation according to the specification. You must also refactor the code to improve its quality.



SheeP (Sheet Processing)

	A	B	C	D	E	F	G	H	I	J
0										
1										
2										
3										
4										
5										
6				2	2					
7				2						
8										
9										
10										
11										
12										
13								8	8	8
14					4	4	4	4	8	
15		3	3			4	4	4	4	
16		3	3				6			
17		5	2	2			6			
18		5	2	4	4		6			
19	5	5			4	4	6			

Plagiarism As per assignment one.

Generative Artificial Intelligence As per assignment one.

Interviews As per assignment one.

Software Design In contrast to assignment 1, you are not given specification at the method level. Instead, you are given a broad specification of each component and how the component must be integrated with the spreadsheet program. The rest of the implementation design is up to you. You should use the software design principles (such as coupling, cohesion, information hiding, SOLID, etc.) that are taught in class to help your design.

The design of your software is part of the assessment. Please be aware that:

1. Discussing the design of your classes in detail with your peers may constitute collusion. Please discuss general design principles (cohesion, coupling, etc) but do not discuss your specific approach to this assignment.

2. Course staff will provide minimal assistance with design questions to avoid influencing your approach. You are encouraged to ask general software design questions.

Bug Fixing Most tests fail initially and this may be overwhelming. Find the smallest failing test — do not start with a complex test as there are multiple bugs that may intersect. Develop some theories about what may be wrong with the implementation, play testing the software may be helpful. You may find it helpful to construct some smaller test scenarios. There are 5 bugs, each can be fixed by modifying a single line in `Tetros.java`. You will need to have a clear understanding about what the bug is before attempting to fix it. You may find manually writing the test cases out (i.e. drawing the tetros board) helpful.

Components

This assignment has 4 components. Each component should be well styled such that it is readable and understandable and should be well designed such that it is extensible and maintainable.

- The first three components are implementation components, that is, you are given a specification and asked to develop an appropriate component that satisfies the specification.
- The last component is a refactoring component, you are given a quite poorly designed implementation and must refactor the implementation to improve the design while preserving its functionality.

Component #1: File Load & Saving

You must implement two new features, one for saving a spreadsheet to a file and one for loading a spreadsheet from a file. The file format is *not* specified. You must design an appropriate file format that can store the state of a sheet.

1. The loading and saving features **must** be compatible, i.e. saving a sheet to a file then loading that file should restore the sheet to its original state (even after closing the spreadsheet application or moving the file).

2. You must *not* utilize Java Serialization.

3. You must modify the *current Sheet instance*, rather than constructing a new instance (Hint: see `Sheet.updateDimensions(int, int)`).

4. You must first clear the sheet (See `Sheet.clear()`), then each cell must be updated starting from the top and populating each row from left to right (row by row).

The `features.files.FileLoading` and `features.files.FileSaving` classes must both implement the supplied `Feature` interface. Both classes must have a constructor that accepts a `Sheet` instance.

1. Within the `FileLoading.register(UI)` method, a feature to load a spreadsheet from a file must be bound to the identifier `"load-file"`.

   See `UI.addFeature`.

2. Within the `FileSaving.register(UI)` method, a feature to save a spreadsheet to a file must be bound to the identifier `"save-file"`.

   See `UI.addFeature`.

When either feature is activated, the user must be prompted for a file path relative to the current directory (See `Prompt.ask(String)`). If any unexpected exception occurs when saving or loading a file (e.g. file not found, file cannot be read, file is the wrong format, etc.), the user must be informed (See `Prompt.message(String)`).

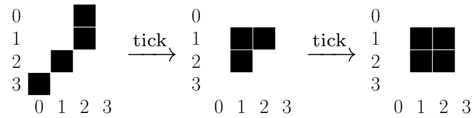
Component #2: Game of Life

Conway's game of life is a popular cellular automaton. The simulation occurs in a grid of tiles. Each tile is either *on* or *off*. At each step a simple set of rules are applied to each tile to determine the state (on or off) of the tile in the next step.

The rules are as follows where a neighbour is any tile one step away horizontally, vertically, or diagonally (giving each tile not on a boundary 8 possible neighbours):

1. Any *on* cell with fewer than two *on* neighbours turns *off*.
2. Any *on* cell with two or three *on* neighbours stays *on*.
3. Any *on* cell with more than three *on* neighbours turns *off*.
4. Any *off* cell with exactly three *on* neighbours turns *on*, otherwise it stays *off*.

For example:



In the above example, moving from the first state to the second state:

- (3, 0) has one *on* neighbour, so turns *off* by (1).
- (2, 1) has two *on* neighbours, so stays *on* by (2).
- (1, 2) has two *on* neighbours, so stays *on* by (2).
- (1, 1) has three *on* neighbours, so turns *on* by (4).
- (0, 2) has one *on* neighbours, so turns *off* by (1).

Moving from the second state to the third state, (2, 2) has three *on* neighbours, so turns *on* by (4).

The `games.life.Life` class must implement the provided `Feature` interface. `Life` must have a constructor that takes a `Sheet` instance. Within the `Life.register(UI)` method:

1. A feature to start the game of life simulation must be bound to the identifier `"gol-start"`.

   See `UI.addFeature`.

2. A feature to stop the simulation must be bound to the identifier `"gol-end"`.

   See `UI.addFeature`.

3. An appropriate tick callback must be registered such that when the simulation is running, the sheet is updated according to the above rules.

   See `UI.onTick`.

Within a spreadsheet, a cell that renders as `"1"` (without quotes) is considered to be in the *on* state. All other cells are considered to be in the *off* state. When turning a cell *off*, the sheet should be updated to insert `""` (without quotes) at the cell.

Component #3: Snake

Snake is another classic game that is played on a grid. The snake is a chain of tiles. When the game starts the chain is just one tile. At each step (tick) the head of the snake moves one tile in the direction it is currently heading. The tail will also move to catch up to the head.

Initially, the snake should be heading south, i.e. increase the row. The spreadsheet user may use the shortcuts, `w`, `a`, `s`, and `d` to alter the direction of the snake such that at the next tick, it moves in the new direction.

`w` Change direction to up/north (decrease row).

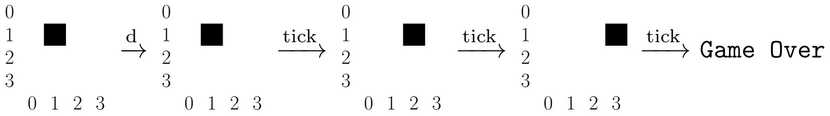
`a` Change direction to left/west (decrease column).

`s` Change direction to down/south (increase row).

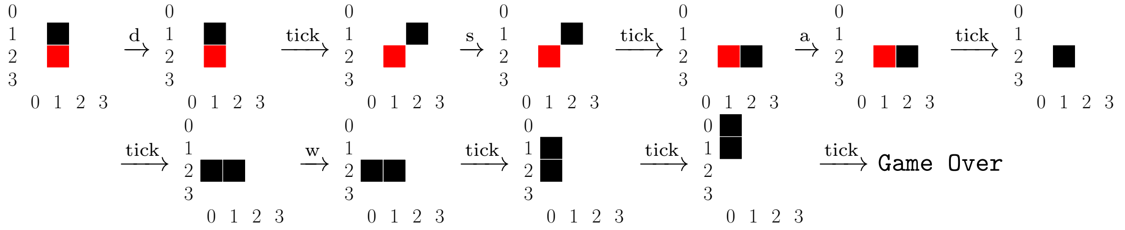
`d` Change direction to right/east (increase column).

If the snake hits a wall (goes beyond the bounds of the grid) or collides with itself, the game is over.

Example 1: Wall Crash



Example 2: Eating



The `games.snake.Snake` class must implement the `Feature` interface. `Snake` must have a constructor that accepts a `Sheet` instance and a `RandomCell` instance. Within the `Snake.register(UI)` method:

1. An action to start a game of snake must be bound to the identifier `"snake"`. The snake must start at the position passed into the `perform` method.
2. An appropriate tick callback must be registered to move the snake at each tick.
3. A key binding must be registered for each direction (`w`, `a`, `s`, and `d`).

When the game ends, this must be indicated to the player via the `Prompt.message(String)` method of the instance passed into the tick callback. The message must say exactly `"Game Over!"` (without quotes).

You may assume that:

1. Every cell that renders as empty, i.e. `""` (without quotes) is a cell the snake can safely move to.
2. Every cell that renders as `"1"` (without quotes) is the snake itself.
3. All other non-empty cells are food that the snake may eat.

In the tick that food is consumed, you must call the `RandomCell.pick()` method and place a new food item with value `"2"` at the returned location.

Component #4: Tetros

The supplied code comes with an implementation of Tetris, called Tetros. This implementation is not a faithful implementation, i.e. it diverges from real tetris in major ways, you might call these bugs. However, this is intentional. The course staff have decided that they prefer this variation of the classic game.

You must maintain the existing functionality, i.e. do not make a proper implementation of Tetris. To ensure that your implementation maintains the original functionality, JUnit tests have been developed. Your implementation should always pass these tests. To make this easy on yourself, ensure that you run the tests after each modification that may cause tests to fail.

Note that the provided tests are not a good demonstration of unit testing. As we want you to have a high degree of flexibility in how you implement your design, the tests are not granular and may be considered closer to integration tests.

Hint A good refactoring of the Tetros program should make it easy to provide a correct implementation of Tetris via dependency inversion. It should also aim to make it easy to change the behaviour of mechanisms such as the rotation system, piece spawning system, piece variations, etc.

Tasks

1. Download the assignment `.zip` archive from Blackboard.
   - Import the project into IntelliJ or your preferred IDE.
   - Ensure that the project compiles and runs (including running JUnit tests).
2. Identify and fix the errors in the provided code.
   - The code contains several bugs, causing the provided JUnit tests to fail.
   - You must fix all errors in the code to implement the specification and pass all the tests.
   - You must not change the provided tests in any way.
3. Complete the implementation of the provided code.
   - There are three components (file saving/loading, game of life, and snake) that have not been implemented.
   - Create appropriate packages and classes for these components as specified above.
   - You must implement these components according to the specification above.
   - You are encouraged to create any additional classes that aid your implementation.
   - You are encouraged to write additional JUnit tests to test your new features.
4. Refactor the provided code.
   - The provided tetros implementation is poorly implemented.
   - You must refactor the provided code to improve its quality.
   - You must not modify the behaviour of the original game while refactoring.

Marking

The assignment is marked out of 100. The marks are divided into four categories: bug fixes (B), extension functionality (F), code quality (Q), and style (S).

	Weight	*Description*
B	10	The errors in the provided code have been fixed.
F	40	The provided code has been extended to include the specified new components and those components function as expected.
Q	40	The new components have good code quality and the provided code has been refactored to improve its quality.
S	10	Code style conforms to course style guides.

The overall assignment mark is defined as

$$A_2 = (10 \times B) + (40 \times F) + (40 \times Q) + (10 \times S)$$

Bug Fixes The provided code includes JUnit tests that fail, indicating an incorrect implementation. You will be awarded marks for modifying the implementation such that the provided JUnit tests pass.

Your mark is based on the number of bugs you fix. The number of bugs you fix is determined by the number of unit tests you pass. For example, assume that the project has 40 unit tests, when given to you, 10 pass. After you have fixed the bugs, 25 tests pass. Then you have fixed 15 out of 30 bugs.[1] Your mark is then

$$\frac{25 - 10}{30} = \frac{15}{30} = 0.5 \ (50\%)$$

In general, let p_0 and f_0 be the number of unit tests that pass and fail in the provided code respectively. If p is the number of unit tests that pass when you submit, then your mark is

$$B = \max\left(\frac{p - p_0}{f_0}, 0\right)$$

Functionality Each class has a number of unit tests associated with it. Your mark for functionality is based on the percentage of unit tests you pass. Assume that you are provided with 10 unit tests for a class, if you pass 8 of these tests, then you earn 80% of the marks for that class. Classes may be weighted differently depending on their complexity. Your mark for the functionality, F, is then the weighted average of the marks for each of the n classes,

$$F = \frac{\sum_{i=1}^{n} w_i \cdot \frac{p_i}{t_i}}{\sum_{i=1}^{n} w_i}$$

where n is the number of classes, w_i is the weight of class i, p_i is the number of tests that pass on class i, and t_i is the total number of tests for class i.

Code Quality The code quality of new features and refactored existing features will be manually marked by course staff.

To do well in this category of the marking criteria, you should consider the software design topics covered in this course. For example, consider the cohesion and coupling of your classes. Ensure that all classes appropriately document their invariants and pre/post-conditions. Consider whether SOLID principles can be applied to your software.

An implementation with high code quality is one that is readable, understandable, maintainable, and extensible. The rubric on the following page details the criteria your implementation will be marked against. Ensure that you read the criteria prior to starting your implementation and read it again close to submission to ensure you meet the criteria.

[1] You do not get any marks for the unit tests that pass before you start and you cannot end up with a negative mark. If your modifications cause a test that originally passed to fail, that no longer counts as a pass.

Readability (40%)

Criteria	Standard				
	Advanced (100%)	**Functional (75%)**	**Developing (50%)**	**Little Evidence (25%)**	**No Evidence (0%)**
Method Decomposition (10%)	All functionality has been decomposed into small coherent methods that have singular clear purposes.	Most functionality has been decomposed into small coherent methods, however, some methods attempt to take responsibility for too much functionality.	The functionality of some methods is not clear as they are either extraneous or perform multiple tasks.	Methods are occasionally decomposed appropriately, however, on the whole most methods perform too many tasks.	Almost no attempt has been made to decompose methods.
Descriptive Naming (10%)	All classes, members, and local variables have clear names that clarify their purpose.	Most classes, members, and local variables have clear names that clarify their purpose.	Some classes, members, or local variables are named poorly and harm the readability of the code.	Minimal attempts have been made to create names that are clarifying.	Almost no attempt has been made to create descriptive names for identifiers.
Documentation (10%)	All classes and public members have clear Javadoc comments that explain how to utilize the classes and members. Documentation may include usage examples.	All classes and public members have Javadoc that attempts to explain how classes and members should be utilized. However, the documentation does not make the purpose and/or expected usage of the software obvious.	All classes and public members have Javadoc comments but they occasionally do not help to explain how to utilize the class in a meaningful way.	Not all classes and public members have Javadoc comments or documentation is not helpful.	Minimal evidence or no evidence of Javadoc documentation throughout the submission.
Program Structure (10%)	The structure of code within methods is clear. Vertical spacing is utilized to separate any logical blocks of code. The control structures are suitable for the task. Any complex blocks or lines of code have been minimized and are appropriately documented.	The structure of code within methods is appropriate but not clear. Vertical spacing is mostly utilized to separate any logical blocks of code. Control structures are somewhat convoluted but mostly appropriate. Any complex blocks or lines of code have been minimized or are appropriately documented.	The structure of code within methods occasionally makes it difficult to understand the intention of the code. An attempt has been made to structure the code but falls short of being well structured.	The structure of code within methods makes the intention of the code or functionality of the method difficult to understand.	The structure of code within methods is poorly structured and hard to read.

Design (60%)

Criteria	Standard				
	Advanced (100%)	**Functional (75%)**	**Developing (50%)**	**Little Evidence (25%)**	**No Evidence (0%)**
Information Hiding (15%)	All classes hide and protect their internal representation. References to internal state is protected by appropriate copying. An appropriate abstraction has been created around the internal representation that would make it feasible to later change implementations.	All classes attempt to hide and protect their internal representation, however, some leakage of internal state may occur. Consideration has been made towards creating an appropriate data abstraction.	Most classes have attempted to protect their internal representation. Some methods that should be private are public, damaging the data abstraction.	Minimal attempts have been made to protect the internal representation of classes. Methods that should be private are public or member variables have been made public.	Member variables/methods are public in a way that encourages tight coupling. No or minimal evidence of data abstraction.

Criteria	**Standard**				
	Advanced (100%)	**Functional (75%)**	**Developing (50%)**	**Little Evidence (25%)**	**No Evidence (0%)**
Dependency Inversion (15%)	The logic of the program is inverted in a way that makes most classes highly configurable. The domain logic of the program is passed down allowing easy future modification.	It is possible to alter some of the program logic via dependency injection. However, non-trival changes would mostly still require modification.	An attempt has been made to parameterise classes by abstractions of domain logic. However, the abstractions are not at an appropriate level.	No attempt has been made to enable changing the program logic by dependency inversion. Some attempt has been made to parameterise classes but program logic is rigid.	The software is highly rigid, any alterations of the program logic would require modification of low-level program components.
Cohesion (10%)	All classes are highly cohesive with each serving a clear single-minded purpose.	Most classes are highly cohesive, serving a clear purpose.	There are no “God classes” but not all classes are highly cohesive.	There is at least one “God class” that assumes too many responsibilities.	No/minimal attempt has been made to decompose classes such that each is cohesive.
Polymorphism (10%)	Polymorphism is used to enhance the flexibility of the program. No subclasses duplicate the functionality of the parent. No classes violate the substitution principle.	Polymorphism is used and no subclasses duplicate the functionality of their parent. No classes violate the substitution principle.	Polymorphism is used and some subclasses duplicate the functionality of their parent. No classes violate the substitution principle.	Polymorphism is used but subclasses duplicate some functionality of their parent or violate the substitution principle.	Polymorphism has not been used.
Contract Programming (10%)	Where appropriate, classes include documented class invariants and pre/post-conditions on public members.	Where appropriate, most classes include documented class invariants and pre/post-conditions on public members.	Some classes have documented their class invariants and pre/post-conditions on public members.	Some classes lack important documentation of their class invariants and/or pre/post-conditions on public members.	Minimal or no attempt has been made to document the invariants/pre/post-conditions on public members.

Code Style The Code Style category is marked starting with a mark of 10. Every occurrence of a style violation in your solution, as detected by *Checkstyle* using the course-provided configuration[2], results in a 1 mark deduction, down to a minimum of 0. For example, if your code has 2 checkstyle violations, then your mark for code quality is 8. Note that multiple style violations of the same type will each result in a 1 mark deduction.

$$S = max(0, 10 - \text{Number of style violations})$$

Note: There is a plug-in available for *IntelliJ* which will highlight style violations in your code. Instructions for installing this plug-in are available in the Java Programming Style Guide on Blackboard (Learning Resources → Guides). If you correctly use the plug-in and follow the style requirements, it should be relatively straightforward to get high marks for this section.

Electronic Marking

Marking will be carried out automatically in a Linux environment. The environment will not be running Windows, and neither IntelliJ nor Eclipse (or any other IDE) will be involved. OpenJDK 21 with the JUnit 4 library will be used to compile and execute your code and tests. When uploading your assignment to Gradescope, ensure that Gradescope says that your submission was compiled successfully.

Your code must compile.
If your submission does not compile, **you will receive zero marks**.

Submission

Submission is via Gradescope. Submit your code to Gradescope *early and often*. Gradescope will give you some feedback on your code, but it is not a substitute for testing your code yourself.

You must submit your code *before* the deadline. Code that is submitted after the deadline will **not** be marked (1 nanosecond late is still late). See Assessment Policy.

You may submit your assignment to Gradescope as many times as you wish before the due date. Your last submission made before the due date will be marked.

What to Submit Your submission must include at least the following directories:

```
src/sheep/features
src/sheep/games
```

Do not include the provided code outside of these packages. If you create additional packages, include them in the submission.

Ensure that your classes and interfaces correctly declare the package they are within. For example, `Snake.java` should declare `package sheep.games.snake;`.

Provided tests A small number of the unit tests used for assessing Functionality (F) are provided in Gradescope, which can be used to test your submission against.
The purpose of this is to provide you with an opportunity to receive feedback on whether the basic functionality of your classes and tests is correct or not. Passing all the provided unit tests does *not* guarantee that you will pass all the tests used for functionality marking.

[2] The latest version of the course *Checkstyle* configuration can be found at `http://csse2002.uqcloud.net/checkstyle.xml`. See the Style Guide for instructions.

Assessment Policy

Late Submission Any submission made after the grace period (of one hour) will not be marked. Your last submission before the deadline will be marked.

Do not wait until the last minute to submit the final version of your assignment. A submission that starts before the end of the grace period but finishes after will not be marked.

Extensions If an unavoidable disruption occurs (e.g. illness, family crisis, etc.) you should consider applying for an extension. Please refer to the following page for further information:

`http://uq.mu/rl551`

All requests for extensions must be made via my.UQ. Do not email your course coordinator or the demonstrators to request an extension.

Remarking If an *administrative error* has been made in the marking of your assignment (e.g. marks were incorrectly added up), please contact the course coordinator (csse2002@uq.edu.au) to request this be fixed.

For all other cases, please refer to the following page for further information:

`http://uq.mu/rl552`

Change Log

Revision: 1.0.0

If it becomes necessary to correct or clarify the task sheet or Javadoc, a new version will be issued and an announcement will be made on the Blackboard course site. All changes will be listed in this section of the task sheet.