SENG201 – 2014
Assignment

PLEASE READ THE INSTRUCTIONS CAREFULLY (including submission and marking guidelines)

General remarks

- This is an **individual assignment**. We encourage you to discuss your assignment in general terms with others, but the work you submit for marking must be your own.
- You may not use material from other sources **without** appropriate attribution. You should assume that plagiarism detection systems will be used. If in doubt, please ask for guidance.
- This assignment is somewhat **open-ended**. We did this on purpose to give you an opportunity to demonstrate the knowledge you have gained from lectures, labs and private study. If you have been keeping up with the lecture and lab material then you should be ready to begin right now. Most new software projects seem complex at first (in fact, if they don't then we probably have missed something). We know what SENG201 students are capable of and we have to mark the assignment so we are confident that you can all do a reasonable job and that there's something to challenge those who want to add extras.
- This assignment includes **one deliverable** which includes your source code, UML diagrams, a brief report, etc. Details about the deliverable are provided in the instructions below. This deliverable is **due at 5:00pm on Monday, May 26, 2014.** The department's standard policy for drop dead dates (May 29, 2014) with 15% penalty applies to the deliverable.
- **Each student will be asked to demo his / her program** (in the week of May 26 or June 2). There is no drop dead date for the demo.
- This assignment contributes **25%** to your **final grade**.

1. Introduction

This assignment is based on scenarios found in a wide range of applications. We will be building software which represents places (e.g., rooms, forests), things (e.g., torches, food) and actors (e.g., people, small furry animals). Items such as these are found at the core of many applications. For example, simulations are a situation where visual representations of an evolving system are useful (e.g., queuing theory provides examples such as telephone exchanges and bank tellers). Games are another common application area.

2. Model-View-Controller (MVC)

Scenarios based on graphical or textual representations of *state* have appeared in a wide range of games, simulations, educational and other applications including “dungeon” style games such as Doom or less deadly applications such as navigating to the baby changing station at the mall.

An effective way to model applications of this type is with the Model-View-Controller (MVC) pattern. MVC allows clean separation of concerns – an important software engineering principle which allows key elements to be designed, implemented and evolved independently. The underlying domain logic specific to the application is decoupled from the (user) interface. The key elements of MVC are:

- **Model:** Core classes which encapsulate the key elements of *state* and *behaviour*. Key model elements in the game / simulation scenarios described above are places, things and actors – plus the environments they inhabit. We will utilize the MVC to enable us to write some model classes which can be used in a range of situations and for different applications.
- **View:** Provides a representation of the model (i.e., it renders the current state of the model), usually as GUI or text interface. More than one view may be available for a single model. When the model state changes state (e.g., because the controller asks it to) the view will be updated (or the view may ask the model for information about its state).
- **Controller:** Handles (user) input events and passes requests to the model and view as appropriate.

There are a number of variations of the basic MVC pattern (see for example Brett D. McLaughlin, Garry Pollice, and David West. Head First Object-Oriented Analysis and Design). Patterns will be discussed in much more detail in SENG301.

3. Process

We have mentioned different software development processes in the lecture, some of which are more flexible than others. We have mentioned agile processes in class and will be using something with an agile flavour for this project. The key idea is that the customer gives us specifications for an increment of functionality which we then implement and test. The process repeats until time runs out or the customer decides to terminate the project.

4. The scenario – part 1

To begin with, we will create some model classes in this iteration. Later on we will add some more features but initially they will be quite simple. Below are some more details about the scenarios we will be modelling:

- We need to be able to have a number of rooms. We might be modelling a building, a cave system or some other world we can make out of rooms.
    - We don’t know much about rooms. Each room has a `label`, if it represents an office then that might be what goes on the door
    - Each room has a brief `description`.

    - We assume that rooms are rectangular (i.e., has `width` and `depth`).
    - We don't care how high rooms are but we need to know what `level` they are on. We use zero for ground level and negative values for basement rooms.
- Rooms may contain things, like furniture, equipment, food. Sometimes weapons. Each thing is in a room.
- People (or other actors) might come along and take things. If a person comes into a room she might pick up something, like a torch, and take it with her. This means, people can move from room to room and collect various items as they go.
- We need some kind of reporting system that can tell us about people, who they are, where they are and what they are carrying. We will use the same sort of reporting system for rooms to tell us about their details, together with their contents.

5. What to do – part 1

- Obtain the following files from the assignment resources archive:
    - Javadoc (folder `doc`) describing the public API's of some relevant classes (`Person`, `Room`, `Thing`, `World`, `Reporter`) and an interface (*`Reportable`*).
    - Java source code for class `Reporter`, which provides a simple text view of a world and some idea of the overall feel of the system. This class implements an interface (*`Reportable`*) and uses a mix of custom code and `toString` calls in `println()`(remember that `println()` implicitly prints whatever is returned by `toString`).
    - A sample output file `demoworldreport.txt` from `Reporter`.
- Using the description above and the supplied Javadoc, design your own `Person`, `Room`, `Thing`, and `World` classes.
    - In a first step, sketch a UML class diagram to record your design.
    - Note that Javadoc does not include private members (properties or methods) so you will probably need to complete some parts of the design.
    - Note in Javadoc that classes are placed in packages in order to simplify subsequent evolution of the system as the requirements become more detailed. Your code should also be structured in this way.
    - Write the code for `Person`, `Room`, `Thing`, and `World` classes.
- Write JUnit tests for your classes.
- Write the *`Reportable`* interface.
- Confirm that your classes confirm to the expected behaviour by running the `Reporter` class supplied. You have the source for this class and this may give you some ideas for how to write your own classes.

6. The scenario – part 2

In part 1 you developed some model classes suitable for use in a range of MVC scenarios. In this part you will extend these and add some new classes. Key model elements in the game / simulation scenarios described in part 1 are places, things and actors, plus the environments they inhabit.

The `Reporter` class simply examined the model and reported the state of the world. In a more general MVC context it would be desirable to have view and controllers too – these are subjects of this part of the assignment. Below is a conversation between the customer (Charles) and the analyst of your software (Alice):

Alice: *So, what do you think of the software we've delivered so far?*

Charles: *Very nice – we should be able to do some good stuff with that.*

Alice: *So what would you like next?*

Charles: *Well, there are so many things we could do – there's not much time before the next convention and we'll need a new version to release there.*

Alice: *Ok, tell me about some of the high priority features.*

Charles: *When you first told me about MVC, I got the impression that the model held all the scenario state.*

Alice: *That's right.*

Charles: *So we can do better than the Reporter you wrote me?*

Alice: *Sure! That's really just to help us check that the model is being built correctly - it just queries the model state and reports what it finds. We could make something that would respond whenever a change occurs in the model and update the info in a display then.*

Charles: *Sounds good. Let's call it a* `WorldView`*. Could you work out the details with your team?*

Alice: *Yes, that sounds like the V in MVC. Will you want a controller too?*

Charles: *Absolutely, our games and simulators are controlled in a range of different ways - keyboard, GUIs, specialised game control units, mobile phones, tablets, etc.*

Alice: *Oh. So where should we start?*

Charles: *Well, I liked the idea of a* `WorldView`*. Could we make a simple* `WorldControl` *that would provide some basic functionality for manipulating worlds?*

Alice: *That sounds like a good way to start. I'm thinking of a simple GUI that allows people to move from room to room, picking up and dropping things as they go.*

Charles: *That would be nice. Maybe we could add some editing facilities so that new rooms, people and things can be added while the application is running?*

Alice: *That should be possible. What else is on your wish list?*

Charles: *Lots of things! It would be nice if the people could remember where they have been as they move around. Could they each keep a history of the places they have visited?*

Maybe we could eventually have them do things like building up maps and finding their way out of mazes?

Alice: *Cool! Will you need different kinds of actors, places and things? We could allow for that in our design.*

Charles: *So I could have Person, Animal and Robot that would all be kinds of Actor?*

Alice: *That's right. We could have different kinds of Thing –maybe some are edible, some can be operated.*

Charles: *Hmmm... sounds interesting.*

Alice: *OK, that's all I need for now.*

Charles: *Great—I'm looking forward to seeing what your team builds me this time!*

Alice uses her notes from the discussion with Charles to produce a prototype view which observes the world (`WorldView`) and controller that can interact with the world (`WorldControl`). Both have a main method so that they can be run on their own. Alice tries them out after making necessary changes to her model classes to make them observable. The `WorldView` simply notes the various updates made to the world and reports them in a textual form. The `WorldControl` controller supports some basic functions, including the following:

- Displaying the world's contents. Its simple GUI has lists of places, people, things in places and things people have taken.
- Modifying the world. Buttons support moving a selected person to a new location, allowing a person to take something from the room they are in and allowing a person to drop something from their inventory (in which case it will remain in the person's current location).

7. What to do – part 2

Please use your work from part 1 as a starting point for working on part 2 and evolve it as you go through the following tasks. We recommend that you backup part 1 and evolve a separate version of part 1 in part 2 so that you can always roll back to a partial solution of this assignment.

- Obtain from the following files from the assignment resources archive:
    - o Java source code for `WorldView` and `WorldControl`.
    - o An executable `.jar` file `WorldDemo.jar` which demonstrates the basic functionality provided by the prototypes.
- Refactor your own `Person`, `Room`, `Thing` and `World` classes to make them observable. Check that they work with the view and the controller provided.
- Add to `WorldControl` operations which provide the ability to add new `Person`, `Room` and `Thing` objects to a `World` and configure them appropriately.

- `WorldControl` sets up a separate `WorldView` but does not really make much use of it. Consider whether it would be better for the controller to respond to updates happening in the view rather than those happening in the model as at present and make any appropriate changes to your programme.
- The GUI of `WorldControl` is rather crude. Maybe the selected layout manager is not the best choice. As the size of the lists increases the interface may become unwieldy. Make any appropriate changes to improve the GUI.
- Consider the pre-conditions and post-conditions of operations and make sure that something suitable happens if inappropriate operations are attempted. For example, should it be possible to go to somewhere you already are?
- Implement appropriate functionality to allow `Person` to collect a history of where they have been, together with a way to display this in a view or controller.
- Use inheritance to generalise places, actors and things. For example, `Room` may be a concrete subclass of an abstract `place.Place`.
- As you go through the above listed tasks, please write and (re-)run test cases frequently, in particular for Model classes.

8. How to get started

1. Read the description. Carefully. Several times. Some of it will be relatively minor detail, but there will also be critical information which will inform the decisions you make. Don't code anything unless you can explain (to yourself or others) why you are doing it. Feel free to consult your tutor or ask for help during the lab sessions.

2. Summarise the most important information. Make notes. Underline candidate classes and methods. You will find that sketching UML is helpful. At this point you'll have a high level picture of what's in the system, what's not and how the pieces are related and interact. You won't know everything yet and you aren't ready to write much code either. You are likely to have questions and the answers may change your mind about design decisions (e.g. is X a class or just a property of other classes?). If you have had to choose between competing designs then note the details (Javadoc comments might be a good place to do this) in case you need to modify your decisions as the program evolves.

3. Set up an eclipse project, decide what the initial package structure will be and create the required packages.

4. Update your UML so it is always in synch with your code.

5. It's likely that there will be things which you can't yet implement fully. In the meantime, it is OK to "fake it" so that your tests will pass. For example, a method might return 42 irrespective of its arguments but will eventually perform some more sophisticated processing.

6. Code your unit tests as JUnit tests wherever practicable and run them often. These tests exercise the public operations provided by your classes so they won't call private methods directly. We expect unit tests for all Model classes.

7. Don't spend too much time initially thinking about the design and implementation of the GUI. If you have covered the Model-View-Controller (MVC) idea then you can think of focusing initially on the model. The classes you write will provide methods which will eventually be wired up to buttons etc. but for a start can just be called from a JUnit test or other test.

8. The term 2 lectures will cover some of the material that you will need to complete the assignment. In the meantime, a simple solution that you can extend later is likely to be a good way to go.

9. Report

Write a clear and concise report describing your work.

- Your report should be limited to 1 – 2 pages (single-sided) of (no smaller than 10 point) text.
- Your report must be in PDF.
- Where collections have been used provide a rationale for your choice.
- Describe the structure and significant features of any components that you have designed.
- Provide, on a separate sheet and a separate file (see below), a UML class diagram for your system. Please add a footnote to indicate classes for which you do not include getters, setters explicitly.

10. Submission and marking

Since we will have to process a large number of assignment submissions, we introduce and enforce several strict submission rules (i.e., we will deduct marks for not complying with these instructions). Your assignment should be submitted as a **.zip** archive via Learn. The following items must be submitted (please use this list as a checklist before you submit your assignment):

- Your program exported as an Eclipse project, including
  - all source code,
  - all unit tests, and
  - Javadoc
- UML class diagrams (as **one** PDF)
- README file with instructions for building and running your software
- Brief report (as **one** PDF)
- Other relevant documents

Naming conventions:

- .zip archive: seng201_2014_<*your last name*>_<*your first name*>.zip
- Exported Eclipse project: seng201_2014_<*your last name*>_<*your first name*>
- PDF with UML class diagram(s): seng201_2014_uml_<*your last name*>_<*your first name*>.pdf

- README: seng201_2014_README_<*your last name*>_<*your first name*>.txt
- Report: seng201_2014_report_<*your last name*>_<*your first name*>.pdf
- <*your last name*> should be your last name in small letters (without the "<" and ">").
- <*your first name*> should be your first name in small letters(without the "<" and ">").

Further notes:

- For your Eclipse project, please check that
  - it compiles,
  - it actually can be imported into Eclipse,
  - it can be run from Eclipse, and
  - it can be run from outside Eclipse
- The Eclipse project, the PDF with UML diagrams and the README should be in one (and the same) directory with no subdirectories (the Eclipse project will have subdirectories though).

Important note: Missing any of the above mentioned items and / or not complying with the submission and / or naming conventions will result in a deduction of marks.

Marking will take into account the professionalism of your approach, the quality of the work you have done, the degree of success you have achieved and the extent to which you have recorded your results and communicated them to the marker.

Major categories for which marks will be allocated include the following:

- **Demo:** You will be required to demo your project. The weight of the demo for the grade is 30%. Therefore, missing the demo will result in a penalty of 30% of the grade for the assignment. Also, changes made to your program and your submitted assignment after the demo (e.g., bug fixing, further testing, new functionality) will not be considered during the marking process.

- **Functionality:** The extent to which the code you have written meets the requirements.

- **Code quality:** This includes layout, naming conventions and commenting (including Javadoc).

- **Tests:** Your own tests, including JUnit tests for code you have written, will be evaluated. The marker will also have additional tests.

- **Analysis and design:** Your analysis, specification and design will be evaluated.

- **Report:** This includes the content, including the design decisions that you made, and style. Basics of technical writing (e.g., structure, grammar, spelling) are important. Employers place a high value on communication skills.