Electronics and
Computer Science

UNIVERSITY OF
Southampton

Coursework Specification

Module:	COMP1202: Programming 1		
Assignment:	Programming coursework	**Weighting:**	45%
Lecturers:	Jian Shi, Heather Packer, Thai Son Hoang		
Deadline:	17/Dec/2021	**Feedback:**	28/Jan/2022

Coursework Aims

This coursework allows you to demonstrate that you:

- Understand how to construct simple classes and implement methods.
- Are able to take simple pseudo-code descriptions and construct working Java code from them.
- Can write a working program to perform a complex task.
- Have an understanding of object-oriented programming.
- Can correctly use polymorphism and I/O.
- Can write code that is understandable and conforms to good coding practice.

Contacts

General programming queries should be made to your demonstrator in the timetabled labs.

Queries about the coursework specification should be made to **Son Hoang** (T.S.Hoang@soton.ac.uk).

Any issues that may affect your ability to complete the coursework should be made known to **Jian Shi** (Jian.Shi@soton.ac.uk) or **Son Hoang** (T.S.Hoang@ecs.soton.ac.uk), ideally before the submission deadline.

Instructions

Late submissions will be penalised at 10% per working day.
No work can be accepted after feedback has been given.
You should expect to spend up to 50 hours on this assignment.
Please note the University regulations regarding academic integrity.

ECS Building Defence (v1.0.0)

Specification

The aim of this coursework is to construct a simple "tower-defence" simulation (https://en.wikipedia.org/wiki/Tower_defense). The purpose of the simulation is to develop a team of ECS students to battle different waves of bugs to defence the ECS Building.

You are not required to have prior knowledge in gaming to complete this coursework.

Your task is to implement the specification as written.

In some cases, specific names are defined or specific properties given for classes. This is mainly to ensure that we can easily read your code, and find the parts that are most relevant to the marking scheme. You will not gain marks for deviating from the specification in order to increase the realism of the simulation. In fact, it may cost you marks and make it harder to assess what you have written. In some cases, we wish to know that you can use specific Java constructs, so if it specifies an `ArrayList` then we want to see you use an `ArrayList`. There might be other ways of implementing the same feature but we are trying to assess your ability to use specific constructs. Where the specification is less explicit in the way something is implemented (for example, by omitted what data types to use) you are free to make your own choices but will be assessed on how sensible those choices are. Comments in your code should explain the choices you have made.

An FAQ (https://secure.ecs.soton.ac.uk/student/wiki/w/COMP1202/Coursework) will be kept on the notes pages of answers that we give to questions about the coursework. If issues are found with the specification it will be revised if necessary. If questions arise as to how certain aspects might be implemented then suggestions of approaches may be made but these will be suggestions only and not defined parts of the specification.

How the ECS Building Defence works

For this coursework you are expected to follow the specification of the students, teams, and the lab as set out below. This will not correspond exactly to a real student, teams, and definitely not bugs nor battle arena in reality but we have chosen particular aspects for you to model **that help you to demonstrate your Java programming skills**.

The ECS Building Defence simulates the team of students to defence against waves of (programming) bugs. There are a number of concepts, people, and procedures that contribute to this simulation. For our purposes these include:

Students: The different students that join the Team and defence the building against the bugs.

Bugs: The (software) bugs that attack the building with the purpose of getting to the top of the building.

Teams: The team of students to defence against the bugs.

Buildings: The buildings for which the student team defence against the bugs. Each building has several floors.

A list of students participated in our simulation are listed in

Table 2. Base attributes for different bugs

. The columns list the base attributes for students, i.e., attack and delay of the different students.

- **Attack** – the student's attacker power. The higher a student's attack power, the more damage that the student will deal to the bugs.
- **Delay** – the student's delay (cool-down) duration. A student with a lower delay duration will use the special ability more often.

We will explain how the base attributes contribute to the students' ability later.

Students	Attack	Delay
Computer Science (CS)	6	6
Computer Science with Artificial Intelligence (AI)	7	7
Computer Science with Cyber Security (Cyber)	7	8
Software Engineering (SE)	5	6

Table 1. Base attributes for different students

The list of programming bugs to be considered in this coursework is in Table 2. The columns list the base attributes for bugs, i.e., their hit points (HP) and the number of steps for the different bugs.

- **HP** – the bug's hit points. A bug's is *alive* if and only if their hit points is positive. We will often use HP as a short-hand for hit points.
- **Steps** – the number of steps required for a bug to move to the next floor. The smaller the number of steps, the faster the bug to go up the building.

Bugs	HP	Steps
ConcurrentModificationBug	20	4
NoneTerminationBug	200	6
NullPointerBug	10	2

Table 2. Base attributes for different bugs

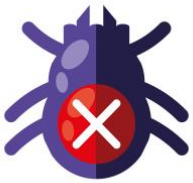
The next sections will take you through the construction of the various required concepts. You are recommended to follow this sequence of development as it will allow you to slowly add more functionality and complexity to your final simulation. The first few steps will provide more detailed information to get you started. It is important that you conform to the specification given. If properties and methods are written in `red` then they are expected to appear that way for the test harness and our marking harnesses to work properly.

You must use Google Java Style (https://google.github.io/styleguide/javaguide.html) for this coursework to gain the mark for code style. We recommend you to use IntelliJ with setting up for Google Java Style for this coursework.

Part 1 – The Bug classes

Create a package called `bugs`. All classes developed in this part must be in this package.

Bugs



(Image from vecteezy.com)

The first class you will need to create is a class that represents *(software) Bugs.* The properties for **the** `Bug` **class** (in the `bugs` package) that you will need to define are:

- `name` – this is the unique name of the bug,
- `baseHp`, `baseSteps` – this is the base hit points, and the base steps of the bug.
- `level` – the level of the bug. The higher the level, the more hit points that the bug can have.

Define these as you think appropriate, and create a constructor that initialises them with the following signature.

```
Bug(String name, int baseHp, int baseSteps, int level)
```

Next, define the accessor methods:

- `getBaseSteps()` – to return the base steps of the bug.
- `getLevel()` – to return the level of the bug.

Now define the following additional properties:

- `currentHp` – the current hit points of the bug. Initially, is compute according to the bug base hit points and the level of the bug.

  Round(baseHP * (level^1.5))
- `currentSteps` – the current steps required for the bug to move to the next floor. If this is 0, the bug will move to the next location when it moves.
- `currentFloor` – the current location of the bug. Initially, the bug is at Floor `-1`, indicating that it is waiting to entering the building.

Define these as you think appropriate, and create *another* constructor that initialises them and other variables with the following signature.

```
Bug(String name, int baseHp, int baseSteps, int level, int initialSteps)
```

In particular, the parameter `initialSteps` must be used as to initialise `currentSteps`.

Now define the accessor methods:

- `getCurrentHp()`, `getCurrentSteps()`, `getCurrentFloor()` – return the current HP, current steps, and current floor of the bug accordingly.

Then define the method `move()` for the bug to move `1` step.

- If the current steps for the bug is positive, decreases the steps by `1`.
- Otherwise, the bug is move to the next floor, i.e., increases the current floor by `1`. The current steps are reset to the `baseSteps - 1`.

Finally, implement the following methods with the suggested signature and specification.

- `damage(int amount)` – decreases the current HP of the bug by the input `amount`. You must ensure that the HP of the bug is always non-negative.
- `slowDown(int steps)` – slow down the bug by adding the input `steps` to the current number of steps.

ConcurrentModificationBugs, NoneTerminatationBugs, NullPointerBugs

To model different types of bugs, you will use inheritance. The base attributes for the different bugs are in Table 2. Create three classes (in the `bugs` package), namely `ConcurrentModificationBug`, `NoneTerminationBug`, `NullPointerBug`, all of which inherit from the `Bug` class. For each of these new class, create a constructor that have parameters for the bug's name, the bug's level, and the bug's initial steps, e.g. `ConcurrentModificationBug(String name, int level, int initialSteps)`.

You can have any additional properties and methods for your classes as you wish.

BY THIS STAGE you should have a package contains a `Bug` class, and the subclasses of `Bug`: `NullPointerBug`, `ConcurrentModificationBug`, and `NoneTermination`. Figure 2 shows the relationship between the different classes that we have at the moment.

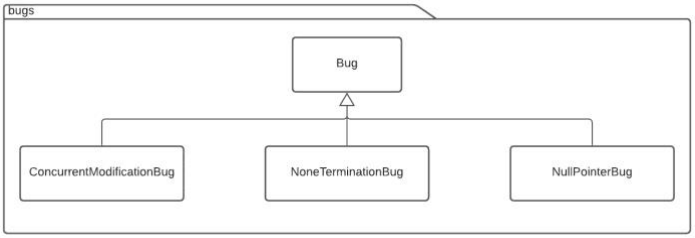


Figure 1. The hierarchy of bugs

You can now test your `Bug` and its subclasses by creating a main method and to create some objects of these classes. You can call `move()`, `damage(int)`, and `slowDown(int)` methods on the `Bug` objects and check how their status changes.

Part 2 – The Building class

Create a package called `building`. All classes developed in this part must be in this package.

(Image from ecs.soton.ac.uk)

The first step in this part is to create is a class that represents a *Building*. The properties for **the `Building` class** (in the `building` package) that you will need to define are:

- `constructionPoints` – the construction points of the building. The higher the construction points of the building, the more tolerant the building against the bugs.
- `topFloor` – the top floor of the building. When a bug reaches the top floor, the bug will make some damage to the building. The building is considered defeated when its construction points reach `0`. Depending on the type of the bugs, the damage to the building is different. For `ConcurrentModificationBug` the damage is `2`, for `NoneTerminationBug` the damage is `4`, and for `NullPointerBug` the damage is `1`.
- `bugs` – the collection of bugs currently attacking the building.

Define `constructionPoints` and `topFloor` as you think appropriate, and use an `ArrayList` to represent the collection of bugs. Create a constructor `Building(int topFloor, int constructionPoints)` that initialises them. Implement the following getter and setter methods

- `getTopFloor()` – to return the top floor of the building.
- `getConstructionPoints()` – to return the current construction points of the building.
- `Bug[] getAllBugs()` – to return the current alive bugs in the building as *an array of* `Bug`s. A bug at Floor `-1` is NOT considered to be in the building (i.e., cannot be attacked). The bugs must be sorted as follows in a descending order to how close the bug to reach the top floor, that is
  - If a bug is on a higher floor than another bug, the former is closer to the top than the latter.

- If two bugs are on the same floor and a bug has less steps to complete the floor than the other, the former is closer to the top than the latter.

- `addBug(Bug bug)` – add a new bug to the building. The method returns an `int` indicating the status of the operation as follows.
  - If the bug is already in the building, returns `-1`.
  - Otherwise, return the number of bugs in the building after the bug has been added.
- `removeBug(Bug bug)` – remove a bug from the building.

Finally, implement the following `bugsMove()` with the specification.

- For each bug in the building, call the `move()` method.
- If the bug reaches the top floor, decreases the construction points of the building accordingly and remove the bug from the building.
- Stop when the construction points of the building reach `0` or when all the bugs have moved.

You can have any additional properties and methods for your classes as you wish.

BY THIS STAGE you should have the `Building` class connected to the `Bug` class. Figure 2 shows the relationship between the different classes that we have at the moment.

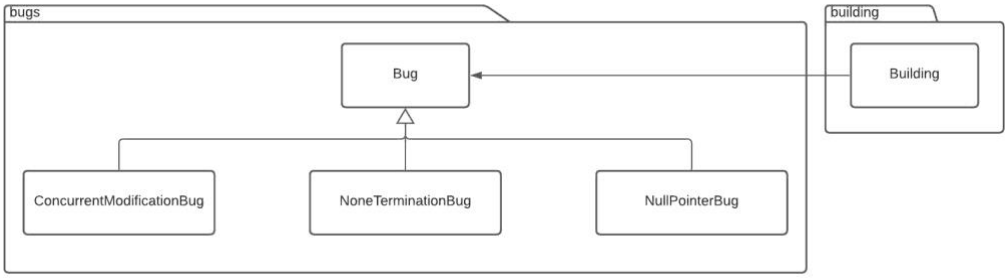
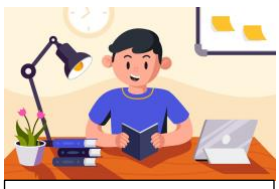


Figure 2. Bug and Building classes

You can now test your new `Building` class by creating a main method and to create some a Building objects. Create some `Bug` objects and add them to the building. Call the `bugsMove()` method and observes the changes to the state of the system.

Part 3 – The Students classes

Create a package called `students`. All classes developed in this part must be in this package.



(Image from vecteezy.com)

Students

The first step in this part is to create is an interface that represents a *Student*. **The `Student` interface** (in the `students` package) will be the basis for the students in different programmes. The base attributes for the different types of students are show in Table 1. The interface must contain the following methods:

- `getLevel` – method to return the level of the student. The higher the level of the student, the more damage that it will deal to the bugs.
- `upgradeCost` – method to return the cost (in knowledge points) for upgrading the student to the next level. The cost of upgrading a student depends on the student level according to the following formula.

  100 * (2 ^ level)
- `defence` – method for the student to defence a building and return the knowledge points gain for the team by removing any bugs from the building. The knowledge points gain for removing a bug is according to the level of the bug as follows.

bugLevel * 20

Define the signature of the methods as appropriated. As a part of defending a building, every student will use either a normal attack or a special ability. The student will use the special attack when their delay is over. For example, for a CsStudent, with the base delay of 6, their patterns of attacking bug (by calling the `defence` method) will be: *normal, normal, normal, normal, normal, Special, normal, normal, normal, normal, normal, Special.*

For a normal attack, the student will attack the first bug (the one closest to the top of the building, see description for `getAllBugs()` in the previous section), and causes a damage to the bug according to the following formula.

Round(baseAtk * (level^1.2))

We describe the special ability for each type of students below.

AiStudents, CsStudents, CyberStudents, SeStudents

To model students in different programmes, you will use inheritance. The base attributes for the different students are in Table 1. Create four classes (in the `students` package), namely `AiStudent, CsStudent, CyberStudent, SeStudent`, all of which implement the `Student` interface. For each of these new class, create a constructor that have one parameter for the student's level, e.g. `AiStudent(int level)`. Implement the methods of the `Student` interface accordingly. Hint: You can introduce other classes/interfaces as necessary.

Each type of students has their own special attack as described below.

- *AiStudent* – The Artificial Intelligence students' special ability of using machine learning and will cause damage to the first 3 bugs (those closest to the top of the building).
- *CsStudent* – The Computer Science students' special ability of pair programming and will cause 4 times the normal damage.
- *CyberStudent* – The Cyber Security students' special ability of setting up antivirus software with a chance of instantly remove the first bug. The probability of instant removal is "level + 20", where level is the current level of the student, but cannot be more than 50. In the case where the student cannot remove the bug instantly, it causes double the normal damage to the bug.
- *SeStudent* – The Software Engineering students' special ability of group working and will cause slow down the first 5 bugs by 2 steps.

You can implement the probabilities with the Toolbox class that was provided with the labs, or by using the methods in the java.util.Random class directly.

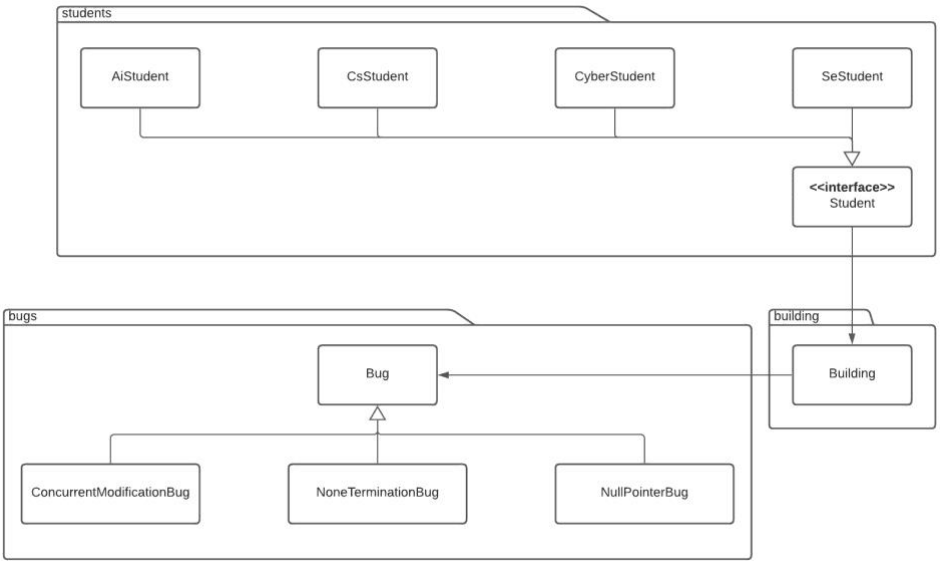


Figure 3. Inheritance Hierarchy of Students and relationship with Building

You can have any additional properties and methods for your classes as you wish. You can introduce new classes or interfaces if necessary.

BY THIS STAGE you should have the first interface `Student` and the implementation of `Student`: `AiStudent`, `CsStudent`, `CyberStudent`, and `SeStudent`. Figure 3 shows the inheritance hierarchy of Students and the relationship with other classes that we have at the moment. (Note: This does not include any additional class that you create).

You can now test your new `Student` and its implementation classes by creating extending your current test from previous section, e.g., create few students and let them attack the bugs in the building.

Part 4 – The Team class

The team manages a group of students and manage the knowledge points gain during the defence of the building. Create a `Team` class (in the `students` package) and add a constructor with an appropriate parameter representing the initial knowledge points for the team. Implement a getter method `getKnowledgePoints` to return the current the team's current knowledge points. Define appropriate properties to store the collection of students in the team and a getter method `getStudents` to return the students as an array of students.

Now define a method for a student team to acts to defence a building, i.e., `studentsAct(Building building)`. The method will cause every student in the team to defence the building in turn. Remember to add the knowledge points gain by the student to the team's knowledge points.

There are two ways to spend the team's knowledge points to improve the team.

1. Recruiting a new student – Recruiting a new student requires some knowledge points. The cost of recruiting students starts from `100` and increase by `10` each time a student is recruited. Define appropriate properties for the class and implement the following methods.
   - `getNewStudentCost` – return the cost for recruiting a new student.
   - `recruitNewStudent` – recruit a new student for the team. The method will throw an `Exception` if the team does not have sufficient knowledge points to recruit a student. The type of the newly recruited students will be determined probabilistically and equally for each type of

students (i.e., 25% for each type). The cost of recruiting a new student is deducted from the team's knowledge points and increased accordingly.

2. Upgrading an existing student – Implement the `upgrade(Student)` method to upgrade a student. The method will throw an `Exception` if the team does not have sufficient knowledge points to upgrade the student. The cost of upgrading the student is deducted from the team's knowledge points. Extend the class `Student` accordingly to increase the student's level by `1`.

You can have any additional properties and methods for your classes as you wish. You can introduce new classes or interfaces if necessary.

BY THIS STAGE you should have implemented the `Team` class. The relationships amongst the existing classes can be seen in Figure 4. (Note: This does not include any additional class that you create).

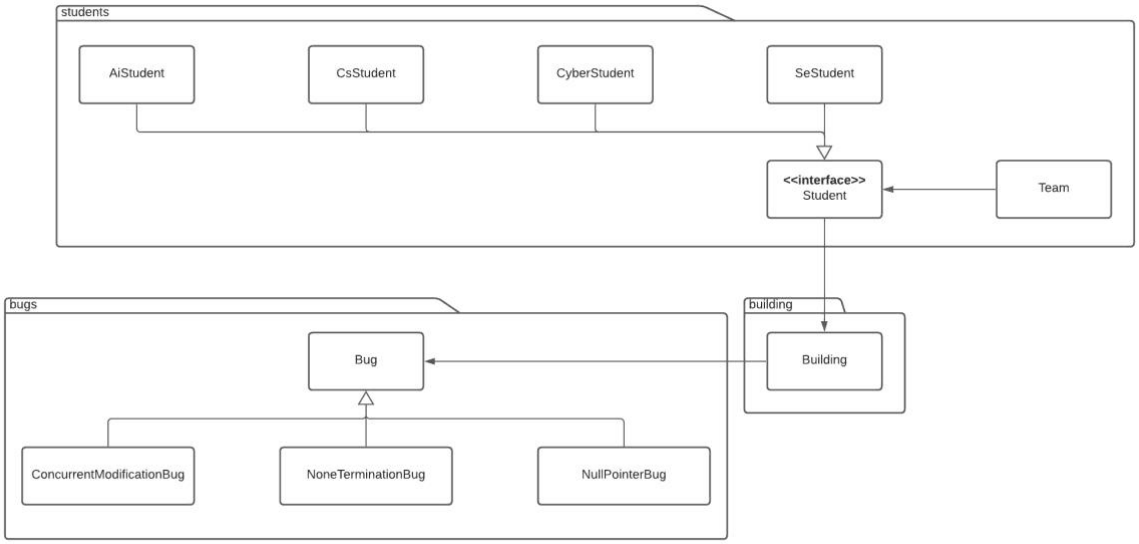


Figure 4. Classes relationships after Part 4

You can now test your newly created classes by extending the existing test from the previous section to create a team, add some students to it, and let the team defence the building to gain knowledge points. Subsequently, you can test to upgrade some students or to recruit some new student. You are advised to get all of this working before attempting to extend your functionality.

Part 5 – Battle

Our simulation will contain a battle between the student team and waves of bugs attacking the building. You should create a new class called `Battle` (do not use package for this class). It should have a `Team` and a `Building` instance, which should be initialised from a constructor `Battle(Team, Building)`.

Implement method `manageTeam()` that makes decision for improving the teams, e.g., by upgrading some existing students or recruiting some new students, given sufficient knowledge points (this method should change the state of the battle this way, i.e., does not allow directly attack the bugs or create/upgrade the students without consuming knowledge points). This method is the intelligence of the game and you can program your strategy for student team. Your implementation should NOT take any input from user but makes decision programmatically based on the current state of the battle, i.e., the status of the student team and of the building. Moreover, the method should not throw any exceptions.

The battle contains several steps, in each step, the following events occur:

- Manage the team using `manageTeam` method.
- The bugs attacking the building moves
- The students in the team act to defence the building.
- Print the information about the team (knowledge points, recruiting cost, students and their information) and the building (the current construction points, the bugs and their information).

Implement the method `step()` to simulate one step of the simulation. You can use print statements to record what is happening in your simulation, including the characters moves. To slow things down a little, the following code will help you to pause for half a second between rounds if you choose to include it.

```
try
{
      Thread.sleep(500);
      // Code for a round
      ...
}
catch (InterruptedException e)
{
}
```

You can have any additional properties and methods for your classes as you wish. You can introduce new classes or interfaces if necessary.

BY THIS STAGE you should have implemented `Battle` class, connected to a `Team` and a `Building`. The relationships amongst the existing classes can be seen in Figure 5. (Note: This does not include any additional class that you create).

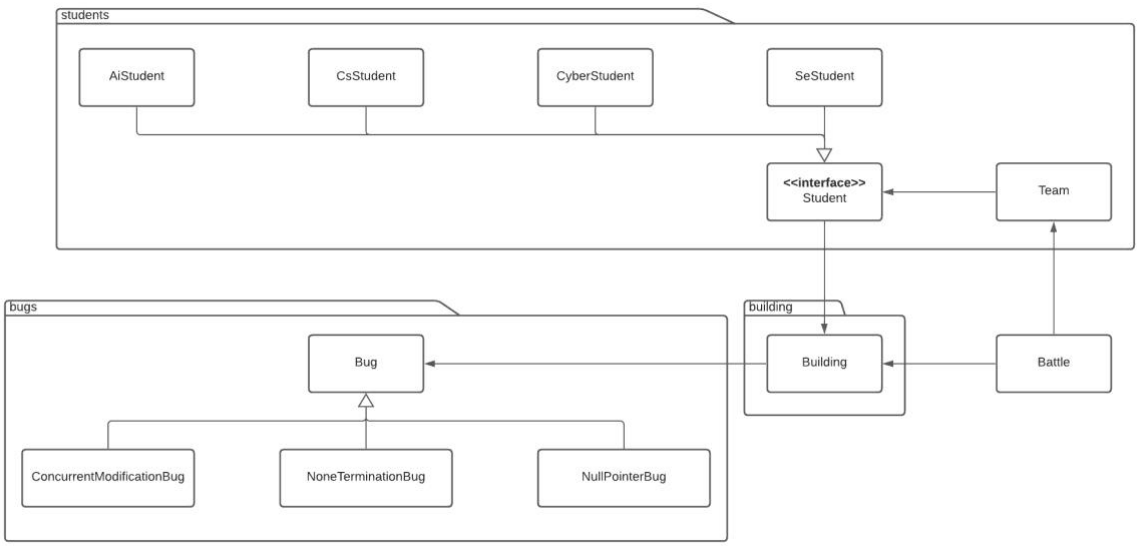


Figure 5. Classes relationships after Part 5

You can now test your new `Battle` class by extending the test from the previous section and connect a Team and a Building. Subsequently you can the corresponding `step()` repeatedly to observe how the state of the battle changes.

Part 6 – ECS Building Defence – Reading a simulation configuration file

In this part we implement the game where to defence ECS building against the bugs. You will need to use your file handling methods as well as split strings into different component parts. Create a battle between a Team and a Building. The parameters for the Building (i.e., the top floor and the construction points) will be instantiate from the program arguments.

Our basic configuration file will look like the example below. You may choose to extend this for your extensions, but for testing purposes your code should accept and use configuration files in this form. Each line gives information about a team of monsters.

```
Ardyn(CMB,1,1);Bahamut(NPB,2,2);Gilgamesh(NTB,1,3);Necron(CMB,1,4);Ultimecia(NTB,1,5)
Ardyn(CMB,2,1);Bahamut(NPB,2,2);Gilgamesh(NTB,1,3);Necron(CMB,1,4);Ultimecia(NTB,1,5)
Ardyn(CMB,2,1);Bahamut(NPB,3,2);Gilgamesh(NTB,2,3);Necron(CMB,1,4);Ultimecia(NTB,2,5)
Ardyn(CMB,3,1);Bahamut(NPB,3,2);Gilgamesh(NTB,2,3);Necron(CMB,2,4);Ultimecia(NTB,2,5)
Ardyn(CMB,3,1);Bahamut(NPB,4,2);Gilgamesh(NTB,2,3);Necron(CMB,2,4);Ultimecia(NTB,3,5)
Ardyn(CMB,4,1);Bahamut(NPB,4,2);Gilgamesh(NTB,3,3);Necron(CMB,2,4);Ultimecia(NTB,3,5)
```

Each line is a list of bugs separated by "`;`". Each bug has the following format.

`Name(type, level, steps)`

Some example simulation files of varying challenge level will be placed on the WIKI.

Create a class `EcsBuildingDefence` with the `main()` method so that it can take `4` arguments: the top floor of the building, the construction points of the building, the name of a file on the command line, and the initial knowledge points for the student team. For example, the following command

`java EcsBuildingDefence 4 20 bugs.txt 100`

will start a simulation for a building with the top floor is `4`, and has `20` construction points. The waves of bugs are taken from the files called `bugs.txt`. The initial knowledge points for the student team is `100`.

When the `EcsBuildingDefence` receives the configuration, it should read the file a line at a time. For each line the *file* will need to create a wave of bugs corresponding to the line and add them to the building. You may find you need to create specific methods or indeed a helper class, to parse the configuration file and extract the information that the `EcsBuildingDefence` needs.

For each wave of bugs, the battle should run `8*topFloor` (8 times the top floor) steps before the next wave of bugs arrive.

You can have any additional properties and methods for your classes as you wish.

You should now have a simulator that runs with a `EcsBuildingDefence` game, well done.

Exceptions

You should be trying to use exceptions in the construction of your simulator where possible. You should be catching appropriate I/O exceptions but also might consider the use of exceptions to correctly manage:

- The input of a configuration file that does not conform to the specified file format.
- …

Submitting the Basic Part

Submit **all Java source files** you have written for this coursework in a zip file **`coursework.zip`**. Do not submit class files. As a minimum this zip will include the files with the following folder structure.

```
coursework
--> bugs
```

```
----> Bug.java
----> ConcurrentModificationBug.java
----> NoneTerminationBug.java
----> NullPointerBug.java
--> building
----> Building.java
--> students
----> AiStudent.java
----> CsStudent.java
----> CyberStudent.java
----> SeStudent.java
----> Student.java
----> Team.java
Battle.java
EcsBuildingDefence.java
```

Submit your files by **Friday 17th December 2021 16:00** to: https://handin.ecs.soton.ac.uk. Note that only 85% of the coursework marks are allocated for the basic part. The other 15% of marks are allocated for the extensions.

Extensions

Have you archived your basic part of the coursework? Please do so before continue.

You are free to extend your code beyond the basic simulator as described above. You are advised that your extensions should not alter the basic structures and methods described above as marking will be looking to see that you have met the specification as we have described it. Please include your extended version in a separate zip file.

Some extensions that we would heartily recommend include:

- Extend to include students from other programmes in ECS (from ELEC sides, there are 6 different programmes, see https://www.ecs.soton.ac.uk/undergraduate/find_a_course), including their abilities. Make sure that you extend your strategy to include these new students.
- You might want to allow the simulation to save out the current state of a simulation to a file so that it can be reloaded and restarted at another time. Define the format of the file to save the state of the simulation appropriately.
- Add (at least 3) new kind of bugs. For each bug (including the old and new ones), adding some special ability which they can use when moving to a new floor, for example, some healing or speeding up abilities.

15% of the marks are available for implementing an extension. Any of the above examples would be suitable, but feel free to come up with one of your own if you like. It is not necessary to attempt multiple extensions in order to gain these marks. However, any extension that you proposed must be comparable in to the above examples in order to gain all the marks. Please describe your extension clearly in the readme.txt file that you submit with your code.

Space Cadets

You might want to add a GUI to your simulator so that you can visualise the state of the *School* at any given moment. **No marks are available for a GUI**, we put the suggestion forward simply for the challenge of it, although you can reasonably expect praise and glory for your efforts.

Submitting the Extensions

Please submit **all Java source files** you have written for extension in a zip file **`extension.zip`** using similar structure as the basic part, i.e., having a folder `extension` containing the source files. Do not submit class files.

Also include a text file **`readme.txt`** that contains a brief listing of the parts of the coursework you have attempted, describes how to run your code including command line parameters, and finally a description of the extensions you have attempted if any. Submit your files by **Friday 17th December 2021 16:00** to: https://handin.ecs.soton.ac.uk

Relevant Learning Outcomes

1. Simple object-oriented terminology, including classes, objects, inheritance and methods.
2. Basic programming constructs including sequence, selection and iteration, the use of identifiers, variables and expressions, and a range of data types.
3. Good programming style
4. Analyse a problem in a systematic manner and model in an object-oriented approach

Marking Scheme

Criterion	Description	Outcomes	Total
Compilation	Applications (containing all parts) that compile and run.	1	15
Specification	Meeting the specification properly.	1,2,4	60
Extensions	Completion of one or more extensions.	1,2,4	15
Style	Good coding style.	3	10