CSCI3100: Software Engineering

Assignment 2

March 7, 2025

```
Due date: 19 Mar 2025 Total marks: 67
```

1 Questions

Let's try to apply our OOP knowledge to model the white blood cells in the Japanese manga series <Cells at Work!>!

White blood cell is a class of cells in our body that can attack the invaders, be it bacteria, virus, or even cancer cells. It can be classified into a few types, including Neutrophils and Lymphocytes. Lymphocytes can be further divided into NK cells, B cells, T cells. NK cells can detect and kill enemies even if the enemies have not been seen before, whereas the other types of white blood cells likely need to rely on some previously collected intelligence.

The enemies can be identified by their antigens. After analyzing the antigens of an enemy, it is possible to characterize it by a list of floating point numbers. B cells can make use of such information to create antibody to attack the enemies, either on their own or by working together with some T cells. An antibody is effective against only one particular type of enemy.

A newbie programmer has come up with the following design. Everything in the design is defined theoretically without adhering to any particular programming language:

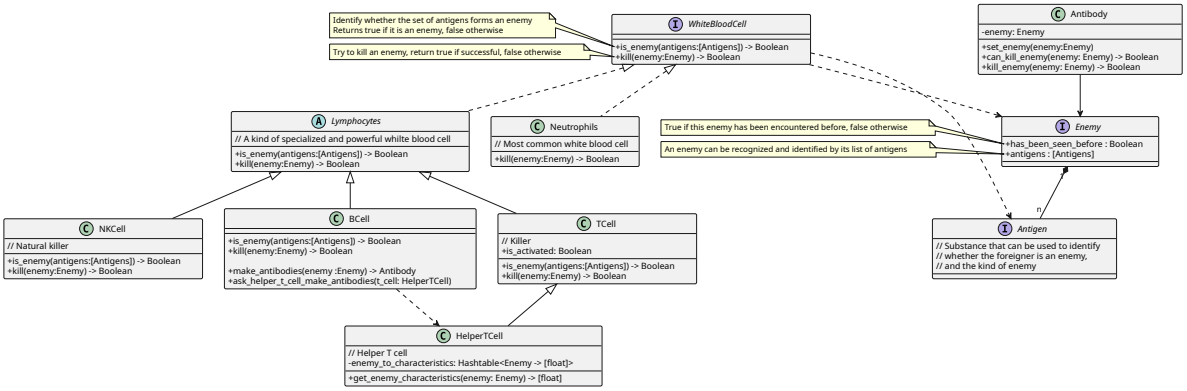

Figure 1: OOP classes related to white blood cells

Notations:

- Struct type:
  - C — Class
  - I — Interface
  - A — Abstract class
- Functions and variables:
  - Functions have the form `func_name(parameter_1, parameter_2, ..., parameter_n) [-> output type]`; the output is optional
  - Variables, and parameters have the form `name: type`
- Naming conventions:
  - Structs are named with the CamelCase
  - Functions and variables are named with the snake_style
- Visibility:
  - `-` for private
  - `+` for public
- Method implementation/overriding:
  - If a method is not listed under the struct, it is either not implemented or not overridden
- Relationships between two structs:
  - There can be arrows in the strong/weak associations indicating who knows whom. If the edge has no arrow, the two structs know each other

– There can be a multiplicity number in composition and aggregation at each end of the edge

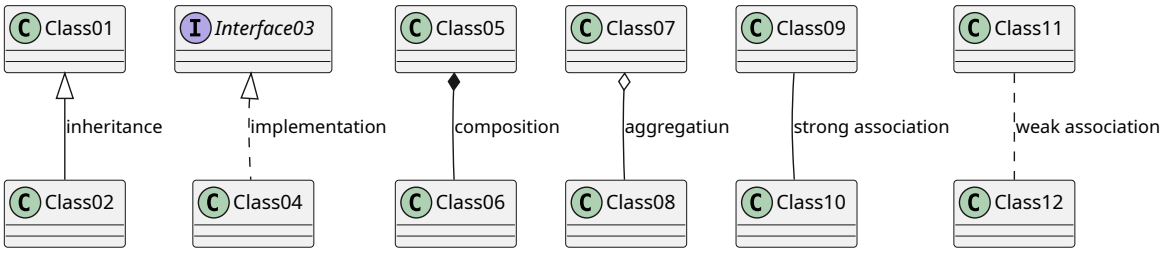

Figure 2: Classes relationships

1. Is it better to apply only OOP, or only pure procedural programming (e.g. C programming language), or a mix of the two programming paradigms together, for the whole software implementation? Why? [5 pt]
2. There are two critical errors in the UML diagram, making it inconsistent with the intended design. Could you list them and give the reasons? [10 pt]
3. Please identify and list the "four pillars of OOP"' observable in the design. [10 pt]
4. Based on the observed convention in the class diagram, please state the differences between `strong association` and `weak association` in the implementation. [5 pt]
5. In the current design, every instance of Antibody stores a copy of the Enemy instance for reference. Its function `can_kill_enemy(...)` will match the argument against the stored Enemy instance copy, and will return true if matched, or false if unmatched. Can Antibody store a list of floating point numbers representing an Enemy instead? What are the advantages and disadvantages? [10 pt]
6. Will Antigen and Enemy be destroyed together, or not? [2 pt]
7. (Not discussed in the lecture) In OOP, every instance has a destructor method that is called when the instance is to be deallocated from the memory. A child class will have the destructor the same as that of its parent class if the programmer does not write that explicitly. If the programmer writes the destructor for the child class explicitly, he may need to call the parent class's destructor within the child class's destructor in some programming language (e.g. Python); in some other programming languages (e.g. C++) the destructors will be called automatically in the reverse hierarchical order: from the child class up the the parent class.

   In the design, `HelperTCell` inherits from `TCell`. In the implementation, is it necessary for `HelperTCell` to override the destructor of `TCell` if they have different resources to be freed? [5 pt]

8. Implement the design in Python after fixing the two critical errors. You can put all classes in the same file named `white_blood_cells.py`. Just use `pass` for the function implementations. [10 pt]

9. Re-design: get rid of all inheritance (**is-a**), and use interface and **has-a** instead. Please provide the UML class diagram. [10 pt]

   For your convenience, the code for generating the original class diagram in PlantUML syntax is listed below:

```
@startuml
skinparam classAttributeIconSize 0

'----------------------------------------

interface Enemy {
+has_been_seen_before : Boolean
+antigens : [Antigens]
}

note left of Enemy::has_been_seen_before {
True if this enemy has been encountered before, false otherwise
}
note left of Enemy::antigens {
An enemy can be recognized and identified by its list of antigens
}
'----------------------------------------
interface Antigen {
// Substance that can be used to identify
// whether the foreigner is an enemy,
// and the kind of enemy
}

'----------------------------------------

interface WhiteBloodCell {
+is_enemy(antigens:[Antigens]) -> Boolean
+kill(enemy:Enemy) -> Boolean
}
note left of WhiteBloodCell::is_enemy {
Identify whether the set of antigens forms an enemy
Returns true if it is an enemy, false otherwise
}
```

```
note left of WhiteBloodCell::kill {
 Try to kill an enemy, return true if successful, false otherwise
}

'----------------------------------------
class Neutrophils {
// Most common white blood cell
+kill(enemy:Enemy) -> Boolean
}
WhiteBloodCell <|.. Neutrophils
'----------------------------------------

abstract Lymphocytes {
// A kind of specialized and powerful whilte blood cell
+is_enemy(antigens:[Antigens]) -> Boolean
+kill(enemy:Enemy) -> Boolean
}

'----------------------------------------
class NKCell {
// Natural killer
+is_enemy(antigens:[Antigens]) -> Boolean
+kill(enemy:Enemy) -> Boolean
}

'----------------------------------------
class BCell {
+is_enemy(antigens:[Antigens]) -> Boolean
+kill(enemy:Enemy) -> Boolean

+make_antibodies(enemy :Enemy) -> Antibody
+ask_helper_t_cell_make_antibodies(t_cell: HelperTCell)
}

'----------------------------------------
class TCell {
// Killer
+is_activated: Boolean

+is_enemy(antigens:[Antigens]) -> Boolean
+kill(enemy:Enemy) -> Boolean
}
```

```
'----------------------------------------
class HelperTCell {
// Helper T cell
-enemy_to_characteristics: Hashtable<Enemy -> [float]>
+get_enemy_characteristics(enemy: Enemy) -> [float]
}

'----------------------------------------
class Antibody {
-enemy: Enemy
+set_enemy(enemy:Enemy)
+can_kill_enemy(enemy: Enemy) -> Boolean
+kill_enemy(enemy: Enemy) -> Boolean
}

'----------------------------------------
WhiteBloodCell <|.. Lymphocytes
Lymphocytes <|-- NKCell
Lymphocytes <|-- BCell
Lymphocytes <|-- TCell
TCell <|-- HelperTCell
BCell ..> HelperTCell
WhiteBloodCell ..> Enemy
WhiteBloodCell ..> Antigen
Enemy "1" *-- "n" Antigen
Antibody --> Enemy

'----------------------------------------
@enduml
```

2 Submission

1. What to submit:
    1. Your answers written in a Word or PDF document
    2. The associated source code files
2. Pack everything in a zip file named ".zip". E.g. if your student ID is `1234567890`, name the zip file as `1234567890.zip`
3. Submit the zip file to Blackboard