

# Software Development, Iterators and Iterables

COSC 1P03 – Lecture 09 (Spring 2024)

Maysara Al Jumaily

[amaysara@brocku.ca](mailto:amaysara@brocku.ca)

Brock University



Tuesday June 25, 2024

Total slides: 32

# Lecture Outline

---

- 01 Seven Phases of Software Development
- 02 Class Responsibility Collaborator (CRC) Cards
- 03 The Concept of Use Cases
- 04 Course Development Application Example
- 05 Implementing Traversals in Java
  - ▶ Iterators
  - ▶ Iterables
  - ▶ Iterators and Iterables

# Seven Phases of Software Development

- There are seven phases of software development, starting with the analysis of the project and ending with the maintenance of the project.



Figure: The seven phases of software development

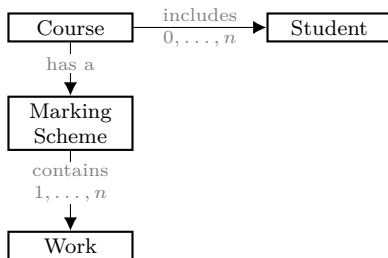
- A “good” project will have an excellent design. The analysis and design phases would either make the project or break it.
- The coding, testing and debugging are all based on the design.
- Maintaining the application is more critical than the initial coding of the application. An application will be coded once, but someone has to keep on maintaining/updating it by finding new bugs and adding more futures.
- Tip for life: design your assignments before coding them!

## ① Analysis – Phases of Software Development

- The *analysis* phase is defining what is the application. What is the problem statement (*i.e.*, the objective), the requirements needed, use cases and sample input and output.
- No one will write you a detailed specification of what the application, or what it needs to be implemented or how the components are integrated.
- You will have to figure this out. It requires meeting with the stakeholder (*i.e.*, person/group who wants to get the application developed) and understand what they are interested in having you develop.
- Don't rush and ask as many questions as needed during that stage. Understand what is needed before moving on to the next phase.
- A skeleton of the application should be formed here. The class names are chosen and the relationship of which class works with which ones is known.
- For instance, when creating a linked list, we need to create a `Node` class, and then ensure our `LinkedList` class *uses* that `Node` class.

## ① Analysis – Phases of Software Development (Cont.)

- Suppose we want to implement the notation of a university course. That course will have the following specifications:
  - A number of students between 0 and  $n$ . Since the range start with zero, we refer to this range as “includes”.
  - One and only one marking scheme. Single value only, known as: “has a”
  - The marking scheme has a number of work components (*e.g.*, assignments, tests, midterms, exams) that is between 1 and  $k$ . Since this range starts with one, we refer to this range as “contains”.
  - In short:  $0, \dots, n \rightarrow$  includes,  $1, \dots, k \rightarrow$  contains, Just 1  $\rightarrow$  has a



**Figure:** An analysis diagram of the relationship between a course, student, marking scheme and work.

## ② Design – Phases of Software Development

- The *design* phase is when the outline of classes and relationships between the components occurs.
- Here is when we choose the names of the classes, instance variables and methods (with their parameter names/type and return types) needed for each class.
- The blueprint of the entire application should be well-defined in this stage.
- Until now, no code has been written yet.
- In case ① Analysis and ② Design phases are done properly, then the coding portion needs to be straightforward.

### ③ Coding – Phases of Software Development

- The *coding* phase is when we write pseudocode and code the application.
- We already have the template of the classes and their objectives. We now start coding!
- This phase should be *much* shorter than the previous two phases.
- Remember to comment your code when writing and use appropriate documentation
- Use **class stubs** whenever needed. A *class stub* is a class that has methods to code but doesn't have the actual logic implemented, rather, has some code to ensure the program runs. The actual logic would be implemented later. For example, suppose we have a method

```
private int getMaxValue(int[] data){...}
```

that finds the largest value inside of an array. Instead of actually coding it, we could write

```
private int getMaxValue(int[] data){ return 0; }
```

and then code it later on.

## ④ Testing – Phases of Software Development

- The *testing* phase used to test the code previously implemented in the application via unit testing, integration testing and system testing.
- **Unit testing:** we test a single unit (*e.g.*, one class)
  - This is similar to coding a linked list with the following insertion (front/back) removal (front/back), printing, etc. and now wanting to actually test it and use the methods.
- **Integration testing:** occurs when several classes are related to each other and are working together to achieve some objective
- Write test harnesses<sup>1</sup> that tests the integration.
  - For example, suppose we have interfaces, stacks, queues, linked lists and arrays involved in one component of the application, then we have to perform integration testing to test all components with each other to ensure no mishaps exist.
  - Sample bug: suppose we have a linked list of `int`egers and want to get the current value (*i.e.*, `p.item`) to push it inside a stack of `String`s. We cannot do that because `int`  $\neq$  `String`. The linked list by itself works fine, and the stack by itself works fine too, but not when they are combined.
- **System testing:** ensures that the application works on different platforms and still have the same performance (more technical).

<sup>1</sup>A test harness is a class that has the `main` method and is created to test some code that has been implemented.



## ⑤ Debugging – Phases of Software Development

---

- The *debugging* phase relies on the testing phase. It is where bugs are fixed and further bugs are found and fixed.
- It is helpful to strategically place `System.out.println(...);` statements and see where the code fails.
- A debugger could also be used.
- Have a look at the end of this document for tips on debugging.

## ⑥ Production – Phases of Software Development

---

- The *production* phase occurs when the application is ready to be deployed to the user.
- Once deployed, we enter the next phase.

## ⑦ Maintenance – Phases of Software Development

- The *maintenance* phase ensure we have technical support hotline, email address for feedback and suggestions, etc.
- A mechanisms required to keep track of the bugs reported along with getting them to the developers of the application.
- After some time, a newer version gets released (previously it was V1.0, now, it is V2.0).
- With each release, ensure to keep track of the bugs encountered and the ones have been fixed. It is very Common to use [GitHub](#) for this sort of tasks.
- Note: Don't upload your assignments publicly on GitHub unless you are allowed.

## Class Responsibility Collaborator (CRC) Cards

- A Class Responsibility Collaborator (CRC) card is a diagram used to defines the class name, its responsibilities (what it needs to know from other classes and what it will do, *i.e.*, methods) and collaborators (other classes to communicate with).
- A sample diagram is shown next.

Class/Interface: <code>NameOfClass</code>	
Responsibilities	Collaborators
Knowing	■ The naumes of other classes it needs to communicate with
■ The data/values needed in order to complete its objective	
Doing	
■ What it will be doing in terms of logic and methods	

- Fill the knowing portion first, then the doing portion second and lastly, the collaborators portion.

# The Concept of Use Cases

- A “use case” is the notion of thinking of generating real-life scenarios that could be encountered within the application.
- A use case has a title, actor, goal and steps.
  - **Title:** title of the use case
  - **Actor:** the individual performing the task (*i.e.*, student, instructor, admin, etc.)
  - **Goal:** the objective that is trying to be achieved
  - **Steps:** actual step-by-step process to be followed
- We will mention some of the use cases later on in this document and denote it as the following block:

## Use case: Title

- **Actor:** ...
- **Goal:** ...
- **Steps:**
  1. Step one text
  2. Step two text
  - ⋮

## Course Development Application Example I

- Let us use the idea of CRC cards and use cases with the application of creating a course calculator.
- We will have the following program to implement:

### Objective

A program is needed to manage the marks for students registered in a course. Throughout the term students receive marks in various pieces of work. At the end of the term students are awarded a final grade computed from the marks in the pieces of work according to the marking scheme. A report (`.pdf`) containing the students' student number and final grade and the course average is sent to the Registrar's Office. The marking scheme defines for each piece of work its base mark and weight towards the final mark.

- Let us highlight the keywords to keep in mind in the next slide

### Objective

A program is needed to manage the **marks** for **students** registered in a **course**. Throughout the term students receive marks in various **pieces of work**. At the end of the term students are awarded a **final grade** computed from the marks in the pieces of work according to the **marking scheme**. A **report** containing the students' **student number** and **final grade** and the **course average** is sent to the Registrar's Office. The marking scheme defines for each piece of work its **base mark** and **weight** towards the final mark.

- The objects to create are:
  - A course containing a list of the students and course average (and indirectly, course name)
  - A student containing a student number and final grade and the work components (and indirectly, student name)
  - A work component that contains the base and weight (and indirectly, component name, such as assignment or midterm, etc.)
  - A marking scheme containing the work components that calculates the final grade of a single student

## Course Development Application Example III

- The CRC card for the course object could be the following

Class/Interface: <code>Course</code>	
Responsibilities	Collaborators
Knowing	■ <code>Student</code>
■ Course name ■ Number of students ■ Course average	
Doing	
■ Get student by id ■ Get next student ■ Compute course average	



## Course Development Application Example IV

- The CRC card for the marking scheme object could be the following

Class/Interface: <code>MarkingScheme</code>	
Responsibilities	Collaborators
Knowing	■ No collaborators
■ Number of work components ( <i>i.e.</i> , count)	
Doing	
■ Entering the marking scheme ■ Applying the marking scheme	

## Course Development Application Example V

- The CRC card for the student object could be the following

Class/Interface: <b>Student</b>	
Responsibilities	Collaborators
<b>Knowing</b>	■ <b>MarkingScheme</b>
<ul style="list-style-type: none"><li>■ Student number</li><li>■ Student name</li><li>■ Marks received in each work components</li><li>■ Final grade</li></ul>	
<b>Doing</b>	
<ul style="list-style-type: none"><li>■ Recording the mark received for each component</li><li>■ calculating the final grade</li></ul>	

## Course Development Application Example VI

- The CRC card for the a work component object could be the following

Class/Interface: <code>Work</code>	
Responsibilities	Collaborators
Knowing	■ No collaborators
■ Work name ■ base mark ■ weight	
Doing	
■ Nothing, just stores the name, base and weight	

- The use case of creating a course and computing final grades

### Use case: Creating a Course

- **Actor:** Instructor
- **Goal:** Prepare system for management of a course
- **Steps:**
  1. Instructor selects class list file (`.bin`)
  2. Instructor enters marking scheme information
  3. The program creates course file

### Use case: Compute Final Grades

- **Actor:** Instructor
- **Goal:** At the end of the term, the Instructor computes the final grades for submission to the Registrar's Office
- **Steps:**
  1. Instructor selects the course file
  2. The system calculates the final grades
  3. The system generates a report of the final grade

- The use case of entering grades

### Use case: Entering students' grades

- **Actor:** Marker
- **Goal:** Enter marks for a piece of work for students
- **Steps:**
  1. Marker selects class list file (`.bin`)
  2. The program presents student mark information for next student
  3. Marker enters mark in piece of work for that student
    - In case no mark is known yet, the marker skips the entry of mark
  4. Repeat steps 2 and 3 for next student(s)
    - In case the marker decides to stop entering marks (to continue later on), then the program doesn't repeat steps 2 and 3

- The use case of updating the students' grades

### Use case: Updating students' grades

- **Actor:** Instructor
- **Goal:** Instructor update mark(s) for piece(s) of work for student(s), or add the grades of missed work
- **Steps:**
  1. Instructor selects the class list file (`.bin`)
  2. Instructor enters student number
    - In case the student number entered is invalid, then go to step 2
  3. The program find the student and presents mark information for student
  4. Instructor updates mark in piece of work for student
    - In case the instructor decides to quit, then we don't repeat steps 2-4
    - In case the student number entered is invalid, then go to step 2
  5. Repeat steps 2 and 4 for next student(s)
    - In case the marker decides to stop entering marks (to continue later on), then the program doesn't repeat steps 2 and 3

## Course Development Application Example X

- The use case of creating a report of the current/final grades received in the course (to be sent out to the Registrar's office)

### Use case: Generating the current course grades report

- **Actor:** Instructor
- **Goal:** The instructor should generate a report of the current marks awarded to the students in the course at any time.
- **Steps:**
  1. Instructor selects the class list file (`.bin`)
  2. The program generates the report

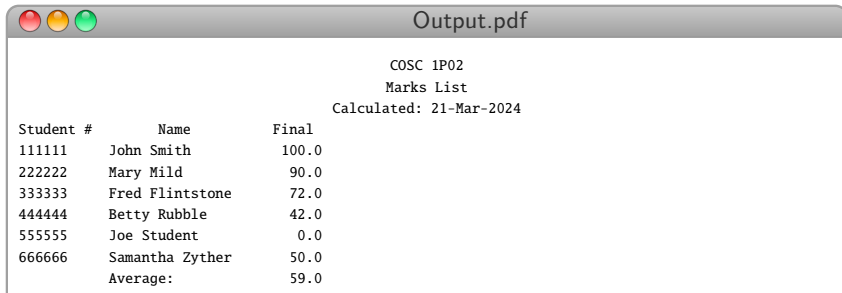
## Course Development Application Example XI

- Now that we have all of the use cases and know the classes to have in the program, let us discuss the implementation of the code.
- There will be four `interface`s and four `class`es (each class will `implements` one interface):
  - The interfaces are: `Course`, `MarkingScheme`, `Student` and `Work`
  - The classes are: `CourseImpl`, `MarkingSchemeImpl` (and `MarkingSchemeTest` for testing purposes), `StudentImpl` (but will have a class stub first named `StudentStub`) and `WorkImpl`
  - The `MarkingSchemeTest` shows how unit testing work.
  - The class stub is there to show how it is used. We will eventually code it and get rid of the class stub.
- These eight files contain the logic of the program, but not the UI.
- The UI will be done by the following classes: `CourseForm`, `MarkingSchemeForm` and `StudentForm`
- Furthermore, we will have the following classes as well: `CourseIterator`, `FinalGrades` and `FinalGradeReport`
- The executable classes (*i.e.*, the ones with the `main` method) are: `FinalGrades` and `MarkingSchemeTest`.



## Course Development Application Example XII

- To execute the program, we need to select the binary file `COSC1P02.bin` as the input, assuming that:
  - The package name is called `Student_Records` and is **NOT** inside some folder (*e.g.*, the package name `package Lecture09.Student_Records;` will not work), it must be `package Student_Records;`
- The input file `COSC1P02.bin` has the list of students, with their names, student IDs and grades. It will have to find the overall course average by calling the `calcFinalGrades()` found in the `Course` interface, which is implemented by the `CourseImpl` class.
- The program will generate a `.bin` file and the following `.pdf` file:



COSC 1P02		
Marks List		
Calculated: 21-Mar-2024		
Student #	Name	Final
111111	John Smith	100.0
222222	Mary Mild	90.0
333333	Fred Flintstone	72.0
444444	Betty Rubble	42.0
555555	Joe Student	0.0
666666	Samantha Zyther	50.0
	Average:	59.0

# Iterators I

- In COSC 1P02, we manipulated images by going through each pixel.
- We use a `while` loop and a `for`-each loop to looping through all of the pixels. Here are two code snippets showing each looping mechanism:

```
Picture pic = new Picture(); // select image
while(pic.hasNext()){
    Pixel p = pic.next(); // go to the next pixel
    int r = p.getRed(); // get the value of the red channel
}
```

---

```
Picture pic = new Picture(); // select image
for(Pixel p : pic){
    int r = p.getRed(); // get the value of the red channel
}
```

- How did the `while` loop know that we still have unvisited pixels? What does it mean to go to the “next” pixel? Do we go to the right of the current pixel? Left? Up? Down? What is the starting pixel? Top-left? Center? Bottom-right?
- How about the `for`-each loop, which pixel do we start with? How to go to the next pixel? etc.?

## Iterators II

- All of the logic and order of iteration has been programmed through the usage of an `Iterator`.
- An *iterator* is a parameterized `interface` that allows us to use the logic of `while(.hasNext()){ ... }` and `.next();`
- It requires three methods to be implemented:
  - `hasNext()`: a `boolean` method that returns `true` if there are more elements in the structure to visit, returns `false` if all elements in the structure has been visited.
  - `next()`: a `void` method that will move to the next available element in the structure. We will have to define and design what “next” means.
  - `remove()`: a `void` method that is supposed to remove an element from the structure. This method will be ignored in the course and will throw a Java exception named `UnsupportedOperationException`. This exception is included in Java’s standard library. The following line is sufficient for the method:

```
throw new UnsupportedOperationException();
```
  - There might be a method called `forEachRemaining` is generated when implementing the interface. Remove it! It is not a part of the course.
- Usually, a class that `implements` the `Iterator` interface will have an array or a linked list containing elements. That array or linked list will be traversed through the iterator mechanism.

## Iterators III

- Here is a simple example of a class that contains an array that needs to be iterated through. The two constructors are present here and the methods to implement are shown next:

```
public class DataIterator implements Iterator<Integer> {  
    private int[] data;  
    private int index; //points to current element  
  
    public DataIterator() { //creates 10 empty cells  
        index = 0;  
        data = new int[10];  
    }  
  
    public DataIterator(int[] values) {  
        index = 0;  
        data = values;  
    }  
    //The code continues in the next slide
```

```
//continuing the code from the previous slide:
@Override
public boolean hasNext() {
    if (index < data.length) { return true; }
    else { return false; }
}

@Override
public Integer next() {
    index = index + 1; //move index by 1 to point to next cell
    return data[index - 1]; //return previous cell
}

@Override
public void remove() {
    //we will not support removing elements from our data.
    throw new UnsupportedOperationException();
}
//Note: remove forEachRemaining method if generated by IntelliJ
} // end of the class
```

# Iterables I

- All of the logic and order of iteration has been programmed through the usage of an `Iterator`, discussed previously, allows the usage of a `while`-has-next.
- Something that is *iterable* is allowed to be used inside a `for`-each loop.
- To make something that is iterable, we must implement an interface.
- The `Iterable` interface is a parameterized `interface` that contains one and only one method to implement:
  - `iterator()`: returns an instance of a class that implements the iterator interface.
  - `forEach(...)`: Ignored method in this course.
  - `spliterator()`: Ignored method in this course.
- In other words, we need to `implements` the `Iterable` interface first, complete the logic, then return a new instance of the implementation class
- Continuing the code from the previous slides, the implementation of the iterable interface is found next slide.

## Iterables II

```
public class DataIterable implements Iterable<Integer> {
    private int[] data;

    public DataIterable() {
        data = new int[10];
    }

    public DataIterable(int[] values) {
        data = values;
    }

    @Override
    public Iterator<Integer> iterator() {
        Iterator result = new DataIterator(data);
        return result;
    }

    // public void forEach(Consumer<? super Integer> action) { ... }

    // public Spliterator<Integer> spliterator() { ... }
}
```

# Iterators and Iterables

- Iterators are used in `while`-has-next scenario, which implements the `Iterator` interface. The logic of iteration is implemented through the `hasNext()` and `next()` methods.
- Iterables are used in `for`-each loops and relies on the class(es) that implement the `Iterator` interface.
- Using a `while(hasNext()){ ... }` will allow us to iterate through the elements once and only once. This is because the flag that denotes whether we have more elements to visit is an instance variable. It will be available for the entire duration of execution. Trying to use another `while(hasNext()){ ... }` loop will not work.
- Use a `for`-each loop ensures that a local variable is returned. This means the its duration of execution is temporary (look at the `iterator` method from the `Iterable` interface). Creating another `for`-each loop creates yet another local instance.
- Hence, we could use as many `for`-each loops as we want, but only one `while(hasNext()){ ... }`.