FIT2099 S1 2022

Online Forum (9/5/2022)

The association between Learning outcomes and assessments:

No.	Learning outcomes	Assessments
1	LO1: Design object-oriented solutions for small to medium-size systems using standard software engineering notations such as UML diagrams.	Bootcamps, EdLessons (10%) Assignment 1 (20%), Assignment 2(20%), Assignment 3 (20%) Examination (30%).
2	LO2: Develop object-oriented designs in an object-oriented programming language such as Java, using object-oriented programming constructs such as classes, inheritance, abstract classes, and generics.	Bootcamps, EdLessons (10%) Assignment 2(20%), Assignment 3 (20%) Examination (30%).
3	LO3: Apply available language tools, such as debuggers and profilers, and good programming practice to debug their implementations systematically and efficiently.	Bootcamps, EdLessons (10%) Assignment 2(20%), Assignment 3 (20%)
4	LO4: Evaluate the quality of object-oriented software designs both in terms of meeting user requirements and in terms of good design principles, using appropriate domain vocabulary.	Bootcamps, EdLessons (10%) Assignment 1 (20%), Assignment 2(20%), Assignment 3 (20%) Examination (30%).

The association between Learning outcomes and Learning materials:

No.	Learning outcomes	Materials
1	LO1: Design object-oriented solutions for small to medium-size systems using standard software engineering notations such as UML diagrams.	Moodle • Week 2 – UML class diagrams • Week 4 – Dynamic diagrams, sequence diagrams and communication diagrams EdLessons • Week 1 – UML and Basic Dependency • Week 4 – Sequence diagrams
2	LO2: Develop object-oriented designs in an object-oriented programming language such as Java, using object-oriented programming constructs such as classes, inheritance, abstract classes, and generics.	Moodle • Week 1 – Abstraction and separation of concerns, Classes and Objects • Week 2 – Inheritance • Week 3 – Encapsulation in Java, polymorphism, abstract classes, packages • Week 4 – Interfaces • Week 8 – Review of abstraction Ed Lessons • Week 0 – Java for Beginners (Part 1) • Week 1 – Java for Beginners (Part 2) • Week 2 – Modifiers and Encapsulation, Inheritance and Abstraction • Week 3 – Interface
3	LO3: Apply available language tools, such as debuggers and profilers, and good programming practice to debug their implementations systematically and efficiently.	Moodle Code along video clips • Week 1 – classes and objects • Week 2 – Code smells • Week 3 – Statics in Java
4	LO4: Evaluate the quality of object-oriented software designs both in terms of meeting user requirements and in terms of good design principles, using appropriate domain vocabulary.	Moodle • Week 2 – three core design principles • Week 5 – Design by contract • Week 6 – SOLID principles • Week 7 – Connascence • Week 8 – Dependency injection • Week 9 – Code smells • Week 10 – Refactoring Ed Lessons • Week 5 – SOLID Principles

Week 10

No.	Learning outcomes	Examination (30%)
1	LO1: Design object-oriented solutions for small to medium-size systems using standard software engineering notations such as UML diagrams.	UML diagrams. Arrows indicate the type of relationship between classes +, - access modifier.
2	LO2: Develop object-oriented designs in an object-oriented programming language such as Java, using object-oriented programming constructs such as classes, inheritance, abstract classes, and generics.	Write/implement codes. Classes, inheritance, abstract classes, interfaces
3	LO3: Apply available language tools, such as debuggers and profilers, and good programming practice to debug their implementations systematically and efficiently.	Trace code errors.
4	LO4: Evaluate the quality of object-oriented software designs both in terms of meeting user requirements and in terms of good design principles, using appropriate domain vocabulary.	Good design principles. DRY, Privacy leaks, Encapsulation. SOLID principles Single responsibility principle Open-closed principle Liskov's substitution principle Interface segregation principle Dependency inversion principle Connascence (9 different levels). Dependency Injection **Constructor injection** **Setter injection** **Interface injection** Code smells

Week 9-10 Code and design smell

Code smells

- is usually a **design problem**

Refactoring

- is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

Code smells were introduced to students started from Week 2. (Refer to Lesson 1.4 Code smells in the IOT example – code along)

Code smells (source: https://refactoring.guru/)

1. Bloaters are code, methods and classes that have increased to such hugh proportions that they are hard to work with. Usually these smells do not crop up right away, rather they accumulate over time as the program evolves.

Long Method: A method contains too many lines of code.	**Solution**: Extract method. Move this code to a separate new method (or function) and replace the old code with a call to the method. Use Replace Method with Method Object
Large Class: A class contains many fields/methods/lines of code. Classes usually start small. But over time, they get bloated as the program grows.	**Solution**: Extract class, Extract subclass, Extract interface.
Long Parameter List: More than three or four parameters for a method.	**Solution:** Introduce Parameter Object.

2. Change Preventers: These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too.

<table>
<tr>
<td>Divergent change: You find yourself having to change many unrelated methods when you make changes to a class. For example, when adding a new product type you have to change the methods for finding, displaying, and ordering products. (Hints: there are two distinct classes there).</td>
<td>Solution: Extract class, Extract superclass/subclass (Inheritance).

```
public class Hero {
    private Integer stamina;
    private Integer health;
    private Armour armour;

    public void defense(){
        // implementation code here
    }

    public void attack(){
        // implementation code here
    }
}

public class Armour {
    private Integer health;
    private Integer status;
    private String rarity;

    public Integer getHealth() {
        return health;
    }
}
```
</td>
</tr>
<tr>
<td>Shotgun surgery: Making any modifications requires that you make many small changes to many different classes.</td>
<td>Solution: Use Move Method and Move Field to move existing class behaviors into a single class.

```
public class SavingsAccount {
    private double balance;

    public void withdraw(double amount) {
        if(isAccountUnderMinimum()){
            this.notifyAccountHolder();
            return;
        }
        // implementation code here
    }
    public void transfer(double amount) {
        if(isAccountUnderMinimum()){
            this.notifyAccountHolder();
            return;
        }
        // implementation code here
    }
    public void processFees(double fee) {
        this.balance = this.balance - fee;
        if(isAccountUnderMinimum()){
            this.notifyAccountHolder();
        }
        // implementation code here
    }
    private bool isAccountUnderMinimum(){
        return this.balance < MINIMUM_BALANCE;
    }
}
```
</td>
</tr>
</table>

3. Couplers: All the smells in this group contribute to excessive coupling between classes.

Feature envy: A method accesses the data of another object more than its own data.	**Solution:** use Move method, Extract Method, Move the data to the calling class.
	`public class Phone {`
	`  private final String thePhoneNumber;`
	
	`  public String getAreaCode() {`
	`  return thePhoneNumber.substring(0, 3);`
	`  }`
	
	`  public String getPrefix() {`
	`  return thePhoneNumber.substring(3, 6);`
	`  }`
	
	`  public String getNumber() {`
	`  return thePhoneNumber.substring(6, 10);`
	`  }`
	
	`  public String toFormattedString() {`
	`  return "(" + getAreaCode() + ") " + getPrefix() + "-" + getNumber();`
	`  }`
	`}`

4. Procedural programming smell (Bloaters/Object-oriented abusers/couplers)

Primitive obsession: Use of primitives instead of small objects for simple tasks.	**Solution:** Logically group some of the primitives into their own class. Try Replace Data Value with Object.
	`public class SavingsAccount {`
	`  private double balance;`
	`  private int accountNumber;`
	`  private String accountName;`
	`  private Address address;`
	`  private MedicareInfo medicare;`
	`}`

Data clumps: Sometimes different parts of the code contain identical groups of variables
(data items that always appear together should probably be attributes of an object).
	Solution: Extract class, Introduce parameter object.
	`class Date {`
	`  int year;`
	`  int month;`
	`  int day;`
	`}`
	
	`class DateUtil {`
	`  boolean isAfter(Date date1, Date date2) {`
	`  // implementation code here`
	`  }`
	
	`  int differenceInDays(Date date1, Date date2) {`
	`  // implementation code here`
	`  }`
	`}`

Switch statements: especially switching on type information.	**Solution**: Replace Conditional with Polymorphism. Extract the switch into the right class.

	`public abstract class Pet {`
	`  abstract String makeSoundInSpanish();`
	`}`
	
	`public class Cat extends Pet {`
	`  String makeSoundInSpanish() {`
	`  return "miau miau";`
	`  }`
	`}`
	
	`public class Dog extends Pet {`
	`  String makeSoundInSpanish() {`
	`  return "guau guau";`
	`  }`
	`}`

Data class: classes that have no logic in them.	**Solution:** extract the code from the method that uses the data and put it in the data class.
	
	`class Date {`
	`  int year;`
	`  int month;`
	`  int day;`
	
	`  boolean isAfter(Date date1, Date date2) {`
	`  // implementation code here`
	`  }`
	
	`  int differenceInDays(Date date1, Date date2) {`
	`  // implementation code here`
	`  }`
	`}`

Message chain: It occurs when a client requests another object, that object requests yet another one, and so on. (this is a special case)	**Solution:** Reduce delegates. Move or extract methods to the beginning of the chain.

Middle man: If a class performs only one action, delegating work to another class.	**Solution:** Remove middle man.
	
	`aFred.doThing()`
	
	`class Fred {`
	`  private Worker worker;`
	`  public void doThing() {`
	`  worker.doActualThing();`
	`  }`

An Example:

Code smell 4 marks

Here is a simplified class “Client” that is part of a system.

Current code:

```
public class Client {
    private ContactDetails _contact = new ContactDetails ();
    public Client(ContactDetails contact) {
    _contact = contact;
}

public String getFullAddress() {
    String address = _contact.getStreetName();
     address += _contact.getStreetNumber() + “ , “;
    address += _contact.getSuburb() + “ , “;
     address += _contact.getPostCode() + “ , “;
     address += _contact.getCountry();
     return (address);
}
}
```

A colleague mentioned that there is something not right with this code and that small changes would contribute to improve the design of the system.

1) What type of code smell is your colleague hinting? Briefly explain how this is a code smell for this specific case (in 1 or 2 lines maximum). (2 marks)

   Feature envy.

   The method getFullAddress is extensively making use of the class ContactDetails.

2) Briefly explain how you would address the actually reduce the connascence in this case to avoid this issue to accidentally occur again in the future (use maximum 3-4 lines). (2 marks)

   move the method getFullAddress to the class ContactDetails since all it does is to use ContactDetails methods

Assignment 3

Group of 2 students: implement three (3) fixed requirements and one flexible requirements.

REQ1,2,3 and REQ 4 or 5.

OR

REQ1,2,3 and ONE complex feature.

Group of 3 students: implement three (3) fixed requirements and two flexible requirements.

REQ1,2,3 4 and 5.

OR

REQ1,2,3 and TWO complex features.

Exam Revision guidelines:

- Revision Learning materials (EdLessons & bootcamps & assignments) based on the learning outcomes.
- Listen to the “video recording”, if needed.
- Concentrate on understanding and applying the concepts.
- Try out the sample exam questions available in Scheduled Final Assessments tab, Moodle. (The sample solutions and marking rubric are given)
- Attend consultations.

Ureview comments received from the student representative

Forum: Need more explanation on the concepts and have lots of videos. Materials given are not prepared well. Video provided for the unit are bit too vague.

Bootcamp: Instruction are very vague and make student confuse. Led to tutors having to explain certain misunderstanding of the questions. Much constructive feedback should be given.

Assignment: Hard to understand, need better formatted and structure specs. Too less time to complete.

Suggested improvement: Solution of each BootCamp should be released.

Recommendations:

Forum: This unit has pre-class activities which are weekly EdLessons and pre-recorded videos in the Moodle before the week. Forum is conducted to revisit the key concepts. During the forum, students are allowed to ask questions related to the weekly concepts or unclear specifications. Multiple choice questions (Flux.qa) are given at the end of the session to perform sanity check. There is no much questions from students related to unclear concepts during the forum.

Action: Students are advised to attend PASS OR consultations OR send questions to Ed Discussion regarding unclear concepts.

Bootcamp: Students have to discuss with tutors to answer open/optional requirements in bootcamps. Students are advised to choose alternative solutions with justification. The number of bootcamps for this cohort is reduced (4) as compared to previous cohort (5).

Assignment: There are pre-recorded videos and explanation in EdLessons to assist students for the assignments. Queries via EdDiscussion/forum/emails are addressed regularly. Samples/guidelines are provided to students during the forum. There is no late submission for Assignment 1. Each assignment is given more than 2 weeks to complete, which would be sufficient.

The teaching team does not provide solutions. Please join consultation for a tutor to help students to answer the bootcamps.