University of Nottingham
UK | CHINA | MALAYSIA

COMP2059
Developing Maintainable Software

LECTURE 10 – REVISION

Boon Giin Lee (Bryan) and Heng Yu

University of Nottingham
UK | CHINA | MALAYSIA

Today's Session

- Exam Revision.
- Exam Formalities.
- Coursework.
- Module Conclusion.

University of Nottingham
UK | CHINA | MALAYSIA

Exam Revision Part 1

IMPORTANT TOPICS FOR LECTURES (EXCEPT LECTURE 04)

BY BRYAN

Developing Maintainable Software

- What is it?
- Why is it important?
- Different types?
- What are the challenges?

Maintainability is not an afterthought but should be addressed from the very beginning of a development project.

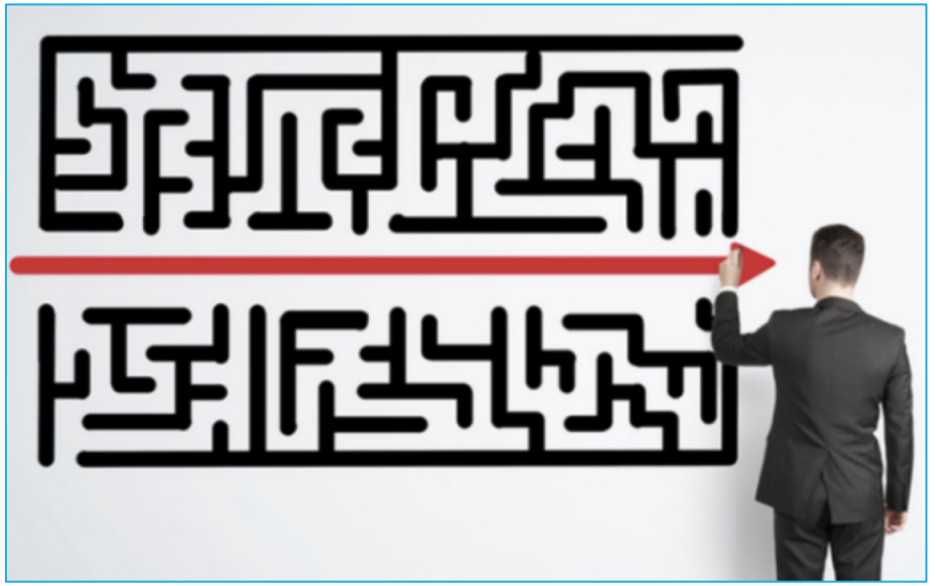
Developing Maintainable Software

- What is it?
  - "Modification of software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."

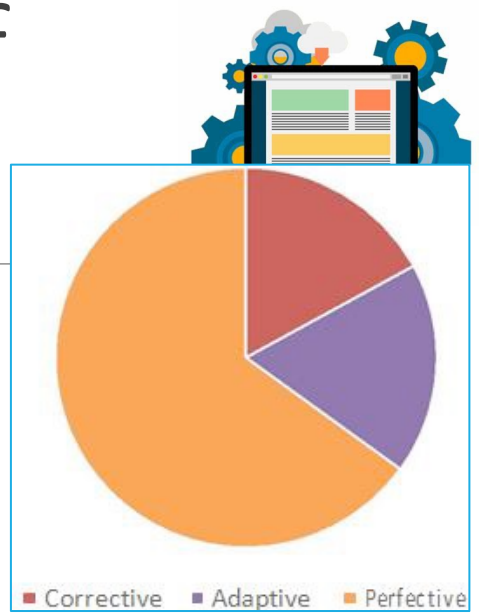
- Why is important?
  - http://blog.lookfar.com/blog/2016/10/21/software-maintenance-understanding-and-estimating-costs/
  - Harder than writing new software.

Maintainable Software

- Overview of generic maintainability guidelines.
  - Write short units (constructors/methods) of code.
  - Write simple units of code.
  - Write code once.
  - Keep unit interfaces small.
  - Separate concerns in modules (classes).
  - Couple architecture components loosely.
  - Keep your codebase small.
  - Automate development pipeline and tests.
  - Write clean code.



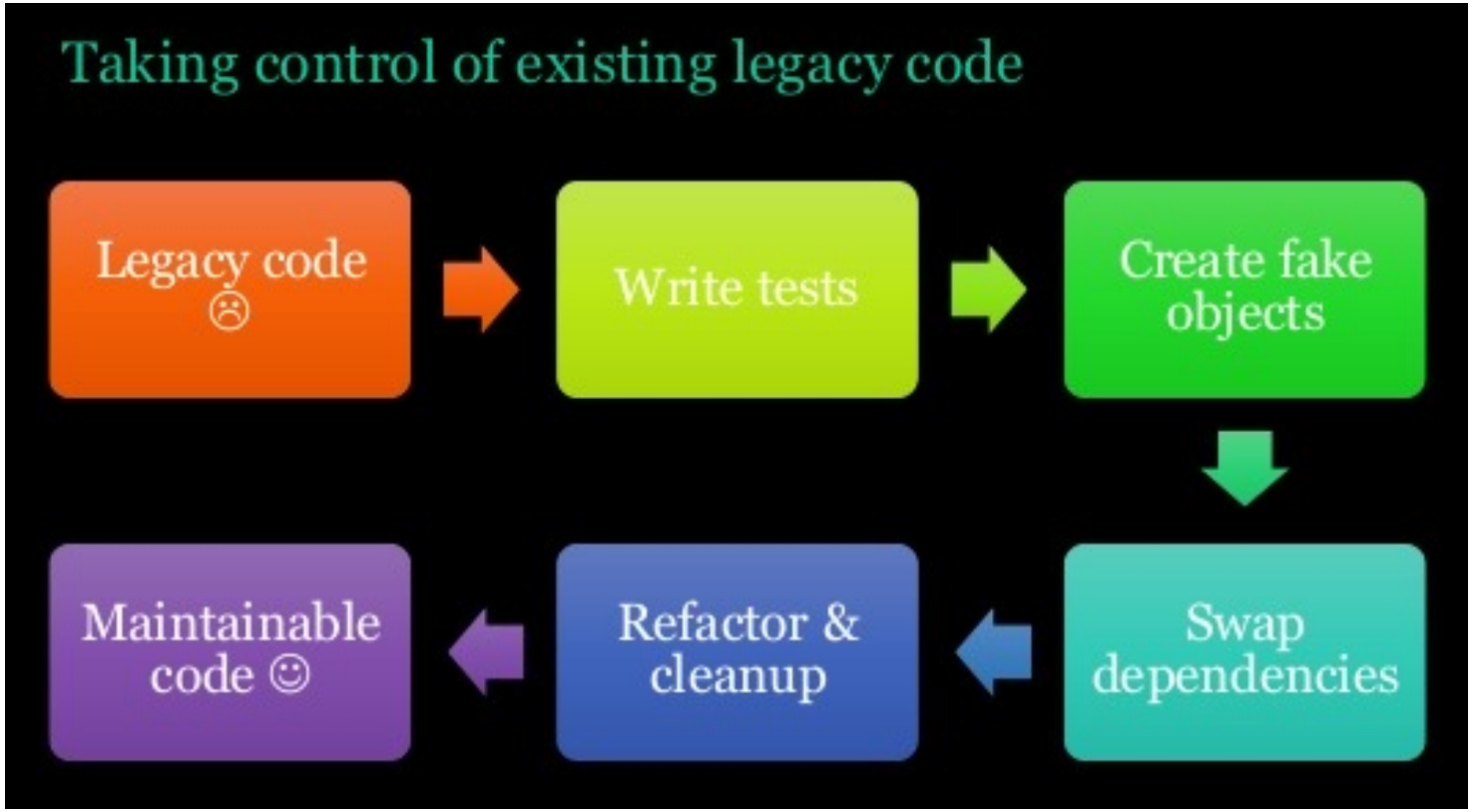
Three (or four, depending on authors) Main Categories of Maintenance



- Corrective Maintenance
  - Finding and fixing errors in the system.
    - e.g., bugs.
- Adaptive Maintenance
  - The system must be adapted to changes in the environment in which it operates.
    - e.g., VAT change, bank offers new mortgage product.
- Perfective + Preventive Maintenance
  - Users of the system (and/or other stakeholders) have new or changed requirements.
  - Ways are identified to increase quality or prevent future bugs from occurring.

Managing Legacy Code

(Lecture 03)

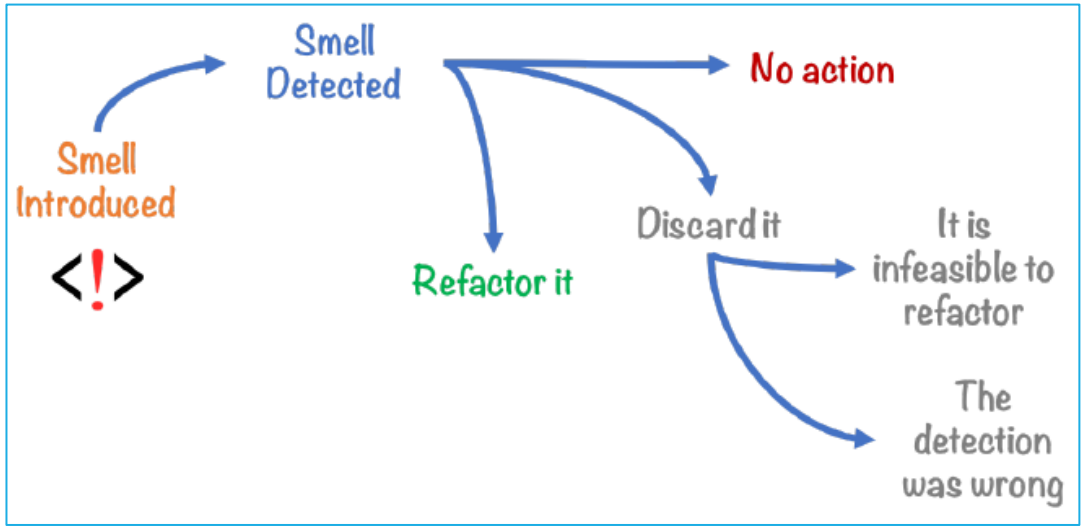


Refactoring

(Lecture 03)

```
/**
* Code Readability
*/
if (readable()) {
    be_happy();
} else {
    refactor();
}
```

- Identify code smells.
- Construct test harness.
- Perform small almost imperceptible changes.
- Run tests until code passes tests.
- Repeat …



Types of Refactoring

(Lecture 03)

- Composing Methods: Refactoring within a method or within an existing class.
- Moving Features Between Objects: Refactoring changing the responsibility of a class.
- Organizing Data: Refactoring improve data structures and object linking.
- Simplifying Conditional Expressions: Encapsulation of conditions by replacing with polymorphism.
- Making Method Calls Simpler: Refactoring that make interfaces simpler.
- Dealing with Generalization: Moving methods up and down the hierarchy of inheritance by (often) applying the Factory or Template design pattern.
- Big Refactoring: Architectural refactoring to promote loose coupling or to realise a redesign via object orientation (NOTE: this is extremely difficult for most projects).

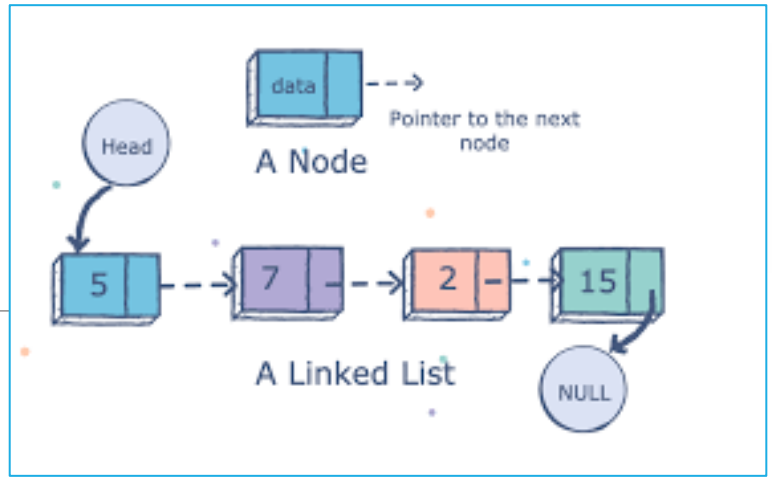
Multi-User Management with Git

(Lecture 05)

- In the old days “three ways merge”.
- You can your peers want to create a commit which includes the changes of both parties.
- If you have edited different parts of the file, Git will know what to do, usually.
- If you have edited the SAME part of the file, a good tool will show you both modifications and then you choose which one to go with.
- Forking and branches …
- Interesting features like “cherry picking” version.

Branching

(Lecture 05)



- Each commit is a node in a linked list on disk.
- Branch is the pointer to that node.
  - If you want to force the branch back, you can lose the old node.
- History tree is preserved by the data structure retaining its linked list integrity.
- Server have their own copies of branches.
  - Pushing or pulling can result in moving your own or someone-else's
- Best thing:
  - Tiny itty-bitty branches so to avoid egomaniacs working on the master branch.
  - Our server is configured to provide protection of the master (can be turned off).

Coding Tools for DMS

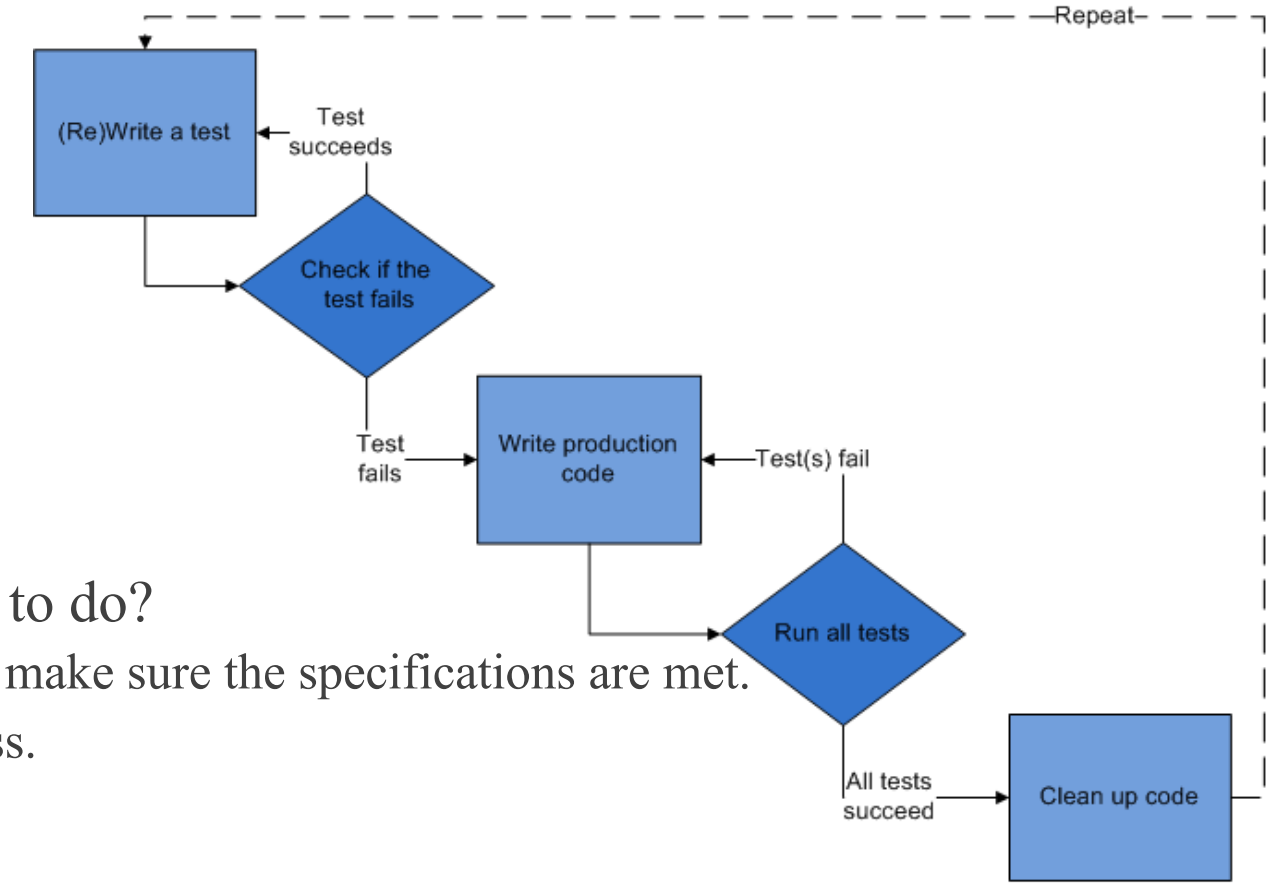
(Lecture 05)

- Documenting code
  - Javadocs – make sure to remember the common block tags (e.g., `@param`, `@return`).
- Build files (build scripts)
  - Common tools (e.g., Maven, Gradle).
  - Understanding of simple Maven/Gradle dependencies.
    - Ant vs Maven vs Gradle: https://www.baeldung.com/ant-maven-gradle
  - Why build files are useful?
  - Typical tasks accomplished with build files.
- Testing
  - Unit and Regression Testing – what's the difference?
  - Check out the unit test code examples from the lab.
  - Test-Driven Development (TDD)

Test-Driven Development

(Lecture 05)

- How does TDD work?
  - Write a test.
  - Check if test fails.
  - If test fails, write production code.
  - Run all tests.
  - If all tests succeed, clean up code.



- What does write the test first allow us to do?
  - Test can be derived from requirements, to make sure the specifications are met.
  - Write the minimum amount of code to pass.
  - Write maintainable code.

Test-Driven Development

(Lecture 05)

Best Practices

When should tests be written?

Tests should be written before the code. Test-first programming is practiced by only writing new code when an automated test is failing.

Good tests tell you how to best design the system for its intended use. They effectively communicate in an executable format how to use the software. They also prevent tendencies to over-build the system based on speculation. When all the tests pass, you know you're done!

Whenever a customer test fails or a bug is reported, first write the necessary unit test(s) to expose the bug(s), *then* fix them. This makes it almost impossible for that particular bug to resurface later.

Test-driven development is a lot more fun than writing tests after the code seems to be working. Give it a try!

[top]

https://junit.org/junit4/faq.html#best_1

Repository Tools for DMS

(Lecture 05)

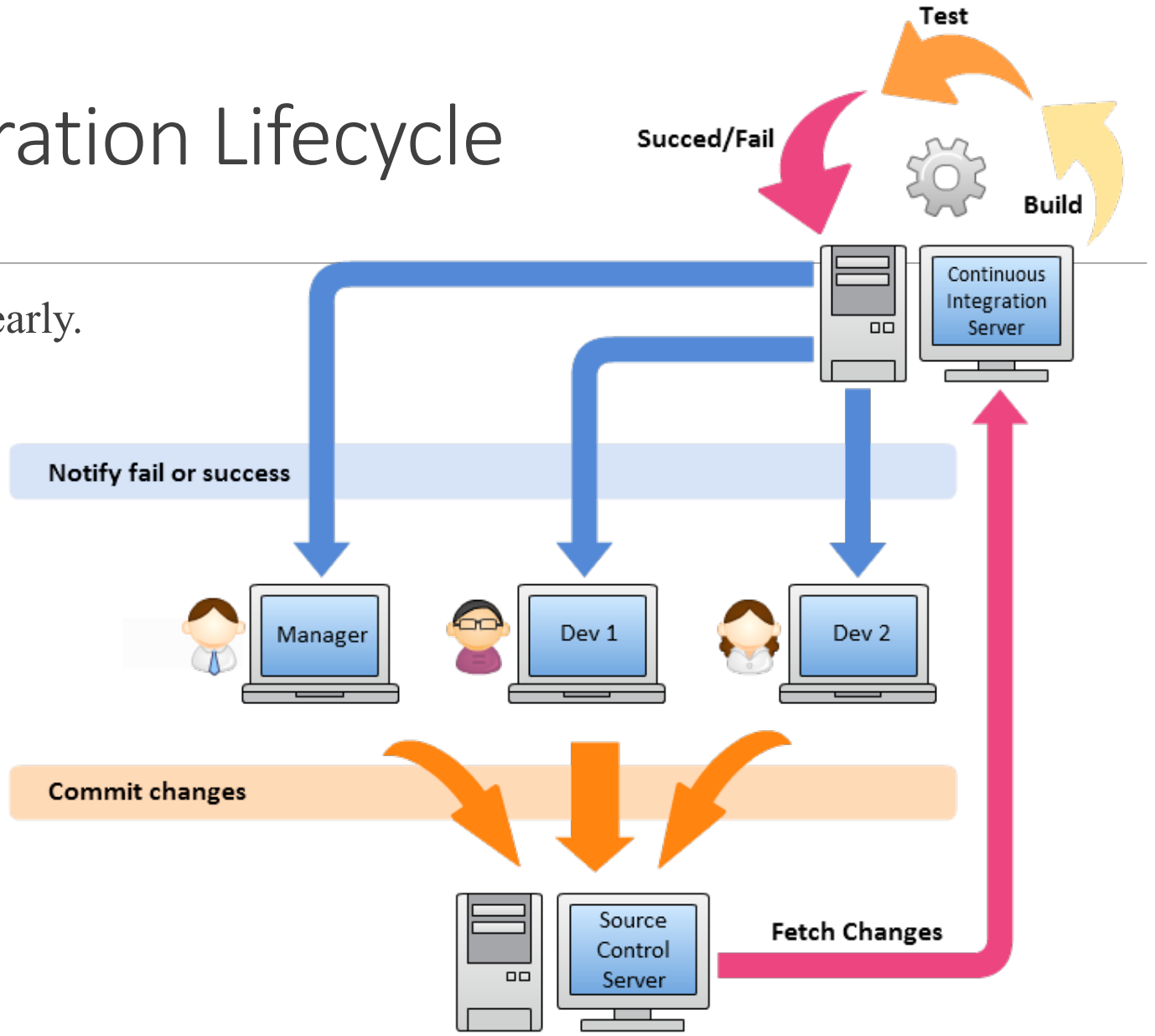
- Sharing a codebase with others.
  - The challenges of working on code together.
  - The three Cs of Team-Based Software Development.
- Managing collaboration.
  - Team and self organisation (e.g., using Trello).
- Collaborative source code management.
  - Awareness of different version control systems (e.g., Git, Apache Subversion (SVN)).
  - Git process in detail.
    - Merging and conflict in Git (e.g., 3-way merge, edit collisions)
    - Use of branches.
- Continuous integration (CI) vs. continuous delivery (CD) vs. continuous deployment.
  - What they are? https://stackify.com/continuous-delivery-vs-continuous-deployment-vs-continuous-integration/

- Communication
- Conflict
- Consistency

Continuous Integration Lifecycle

https://code-maze.com/what-is-continuous-integration/
(Lecture 05)

- Integrate new code often and early.



Maintainable GUI Development

(Lecture 06 + 07)

- GUIs in Java.
  - Swing vs JavaFX.
  - Concept of FXML and SceneBuilder.
  - Event driven programming in JavaFX.
  - Simple usage of CSS to style interfaces.
  - MVC design pattern.
- Good user interface design considerations.
  - Guidelines provided in Lecture 06 and 07.
- Threading principles.
  - Stop GUI becomes unresponsive.
  - The idea behind why to do this.

Properties & Binding

(Lecture 06 + 07)

- JavaFX properties are often used in conjunction with binding, a powerful mechanism for expressing direct relationships between variables.
- When objects participate in bindings, changes made to one object will automatically be reflected in another object.
- Can also add a change listener to be notified when the property's value has changed.

See "Using JavaFX Properties and Binding": https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm

See http://tutorials.jenkov.com/java/lambda-expressions.html for in-depth introduction to lambda expressions.

Lambda Expressions

(Lecture 06 + 07)

- It is to be noticed that it is rather laborious to wire up events to buttons.

```
btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Bye!");
        System.exit( status: 0);
    }
});
```

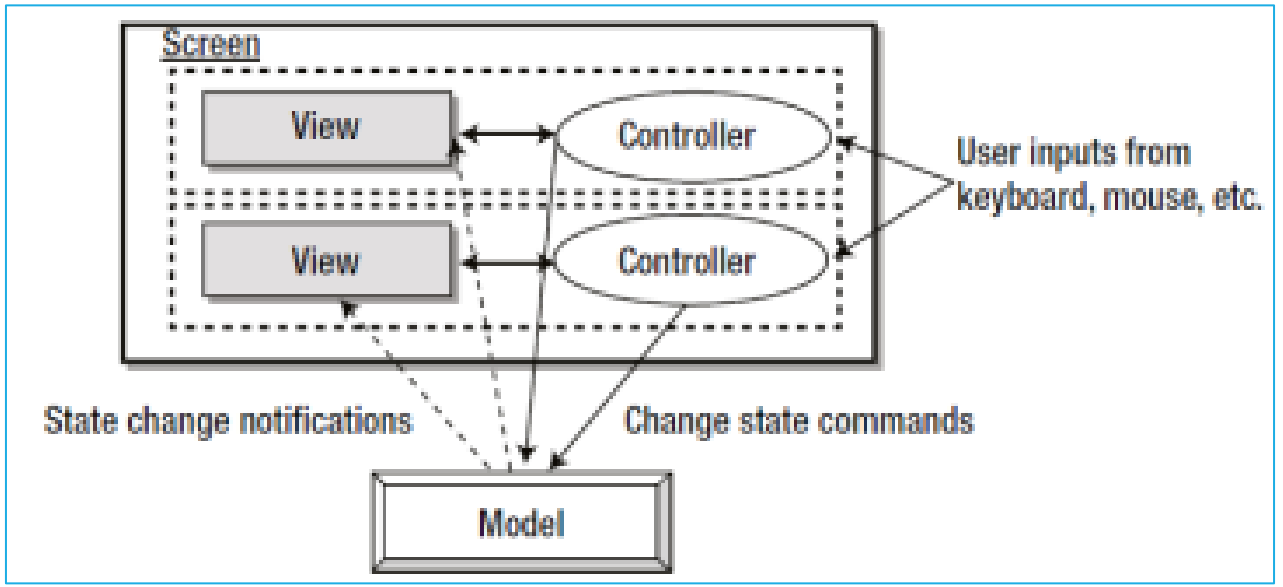
- Is there a better way?
  - YES! Java 8 introduced **Lambda expressions**.

```
btn.setOnAction(event -> {
    System.out.println("Bye!");
    System.exit( status: 0);
});
```

MVC Design Pattern

(Lecture 06 + 07)

- The model provides a way for views to subscribe to its state change notifications.
- Any interested views subscribe to the model to receive state change notifications.
- The model notifies all views that had subscribed whenever a model's state changes.



Threading: Stop GUIs Becomes Unresponsive

(Lecture 06 + 07)

- JavaFX launches the UI on a JavaFX Application thread.
  - This thread should be left handling the UI interaction.
  - Heavy computation should be done elsewhere to prevent freezing!

- JavaFX provides a solution: The "`javafx.concurrent`" package.
  - A way of creating and communicating with other threads with a JavaFX interface.

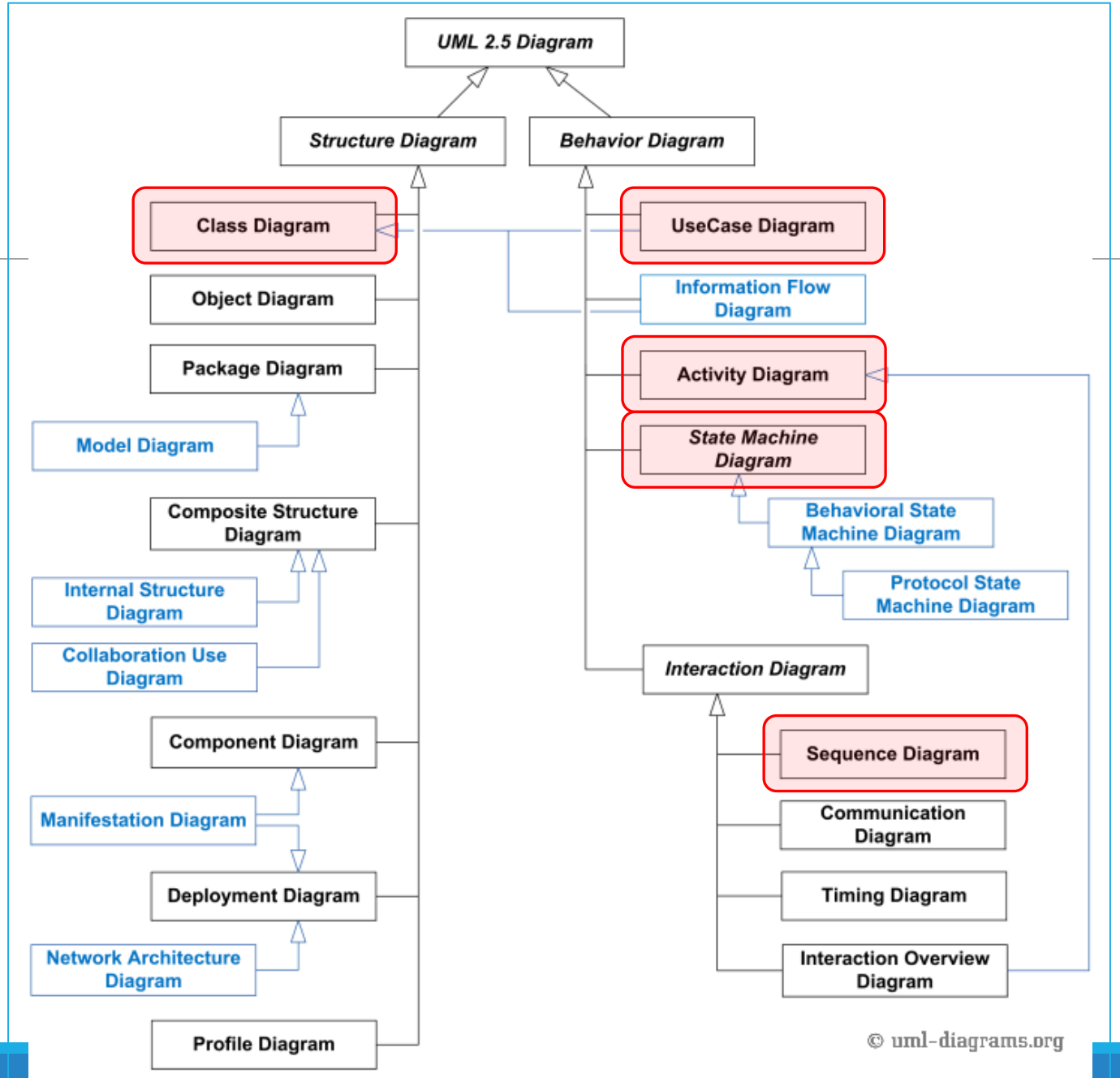
University of Nottingham
UK | CHINA | MALAYSIA

UML and Its Java Implementation

(Lecture 08)

- UML diagrams
  - Only need to consider the ones we discussed in the lecture.
  - What are those diagrams used for?
    - Use case, sequence, activity, class, state machine diagrams.

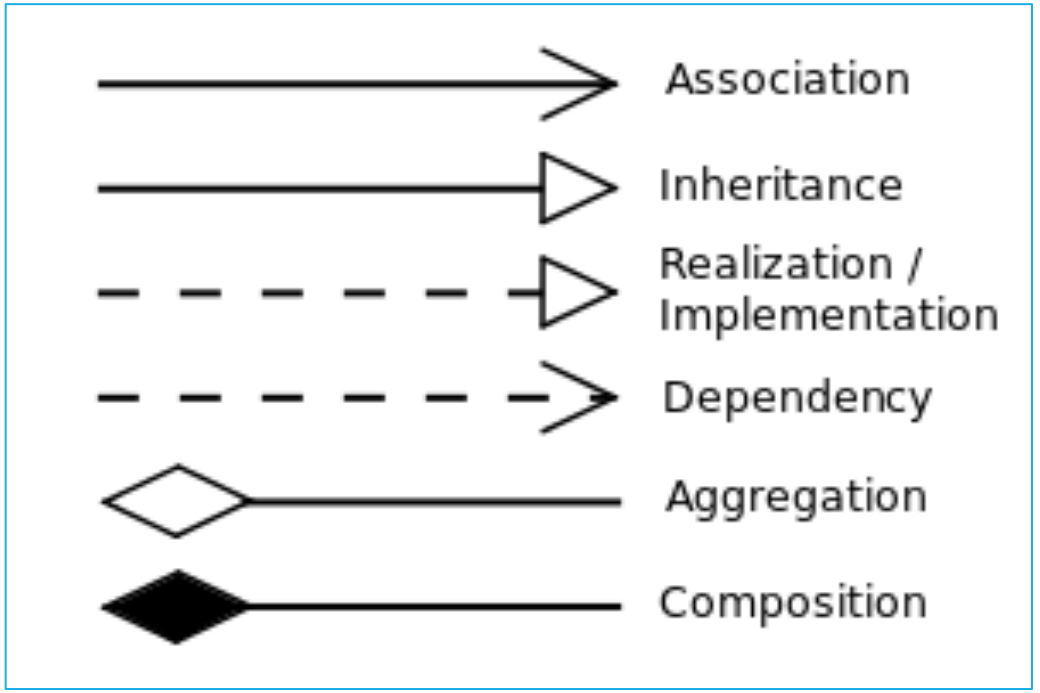
- Class diagrams
  - Relationship types and multiplicity.
  - Design.
  - Implementation.


UML 2.5 Diagram
Structure Diagram
Behavior Diagram
Class Diagram
Object Diagram
Package Diagram
Model Diagram
Composite Structure Diagram
Internal Structure Diagram
Collaboration Use Diagram
Component Diagram
Manifestation Diagram
Deployment Diagram
Network Architecture Diagram
Profile Diagram
UseCase Diagram
Information Flow Diagram
Activity Diagram
State Machine Diagram
Behavioral State Machine Diagram
Protocol State Machine Diagram
Interaction Diagram
Sequence Diagram
Communication Diagram
Timing Diagram
Interaction Overview Diagram
© uml-diagrams.org

Class Diagram

(Lecture 08)

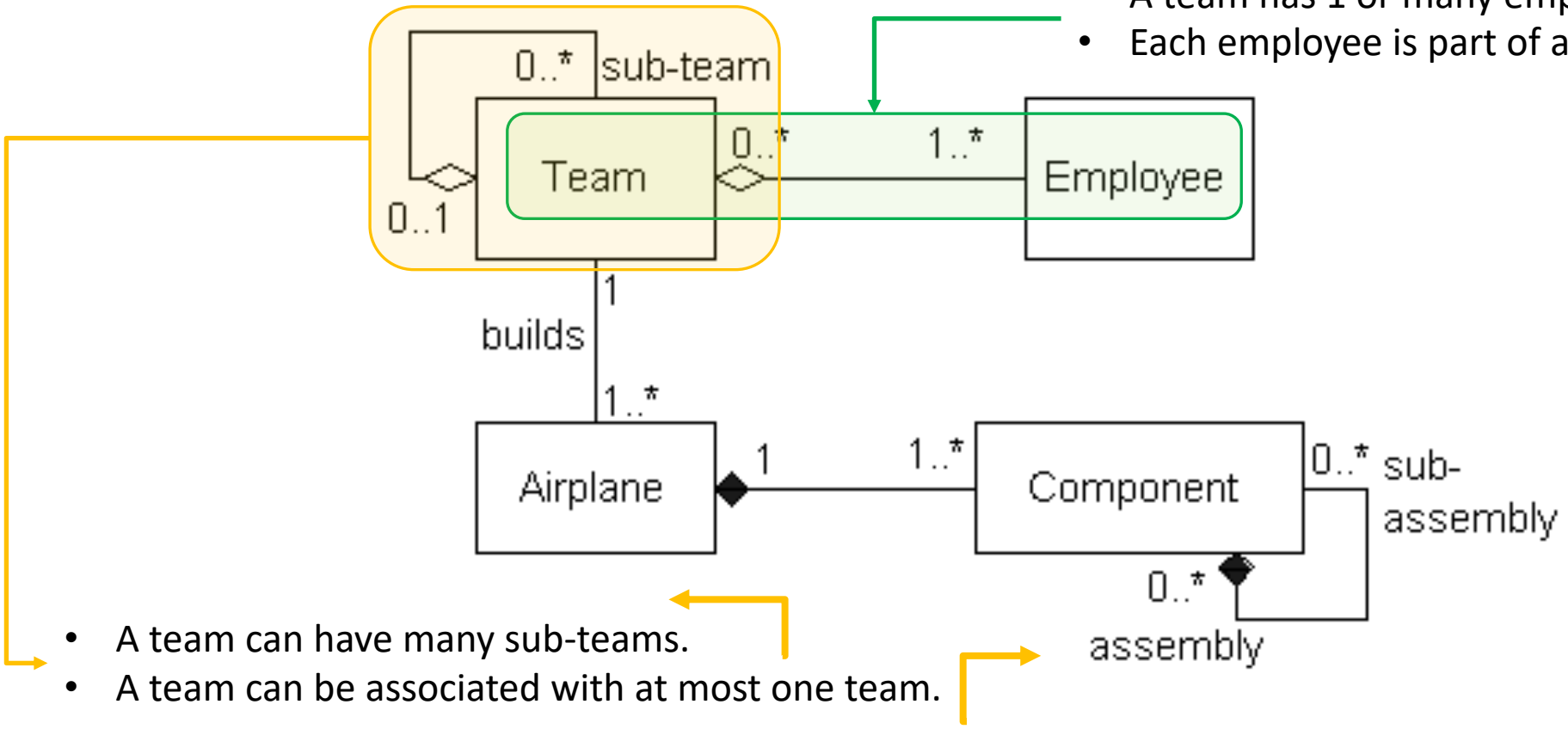
- Class diagrams show the existence of **classes**, their **structures** and **relationships** in the logical view of a system.
- Class diagram components:
  - Classes (structure and behaviour).
  - Class relationships.
    - Association.
    - Dependency.
    - Aggregation.
    - Composition.
    - Realisation.
    - Generalisation / Inheritance.
  - Multiplicity and navigation indicators.



Class Diagram

(Lecture 08)

- A team has 1 or many employees.
- Each employee is part of at most 1 team.



- A team can have many sub-teams.
- A team can be associated with at most one team.

Class Diagram

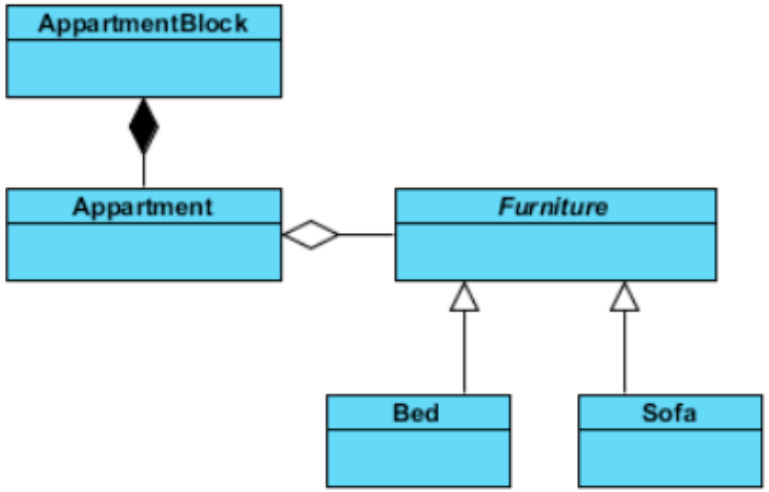
(Lecture 08)

- How can we tell if a reference means **aggregation or association**?
  - Well, we can't.
  - The difference is only logical – whether one of the objects is part of the other or not.

Class Diagram

(Lecture 08)

- Composition vs Aggregation
  - Composition
    - A relationship where a child cannot exist independent of the parent, e.g., an apartment cannot exist outside of the `ApartmentBlock`.
    - Implies that the internal objects are not seen from the outside.
  - Aggregation
    - Implies that the child can exist without the parent, e.g., the bed can exist outside of the apartment.
    - Obtaining new functionality by assembling objects.
      - This allows reuse as aggregated objects have a separate existence.
    - Aggregated objects might be directly accessed.



Open Source, Libraries, Communal Software Development

(Lecture 09)

- Software deployment through libraries.
  - Example: Java JRE - *.jar file + API + License.
- Open-source software development.
  - Why go open source?
  - Principle of open-source development.
    - Open-source software.
  - Common licenses / types.
    - Berkeley Software Distribution (BSD), GNU etc.
    - Permissive, CopyLeft etc.
  - Tracking issues and bugs.
    - Mantis and Bugzilla.

Open-Source Software

(Lecture 09)

- Why Go Open Source?
  - Customisable.
  - Improvable.
  - Collaborative bug finding/fixing.
  - Redistributable.
  - Transparency.
  - Free.

  - Freedoms 0: Run it
  - Freedoms 1: Change/modify.
  - Freedoms 2: Redistribute – free or sell.
  - Freedoms 3: Redistribute with charges.

Which Open-Source License is Best?

(Lecture 09)

- Copyleft licenses generally impose more restrictions — potentially offering less liability — compared to permissive licenses.
  - If to maximize code reusability and sharing: Permissive license
  - Developing software intended for use over a network: Affero General Public License (AGPL)
- The GNU General Public License (GPL) comes in two main versions: GPLv2 and GPLv3.
  - GPLv3 addresses issues not covered in GPLv2, such as patents, and enhances compatibility with other open-source licenses like the Apache License v2.
  - GPLv2 and GPLv3 are not compatible with each other.
- MIT licenses enjoy widespread usage, benefiting from their recognition and common understanding.
  - Software licensed under MIT entails no restrictions on redistribution or monetization.
  - Moreover, MIT licenses are compatible with many other open-source licenses.

License: Overview

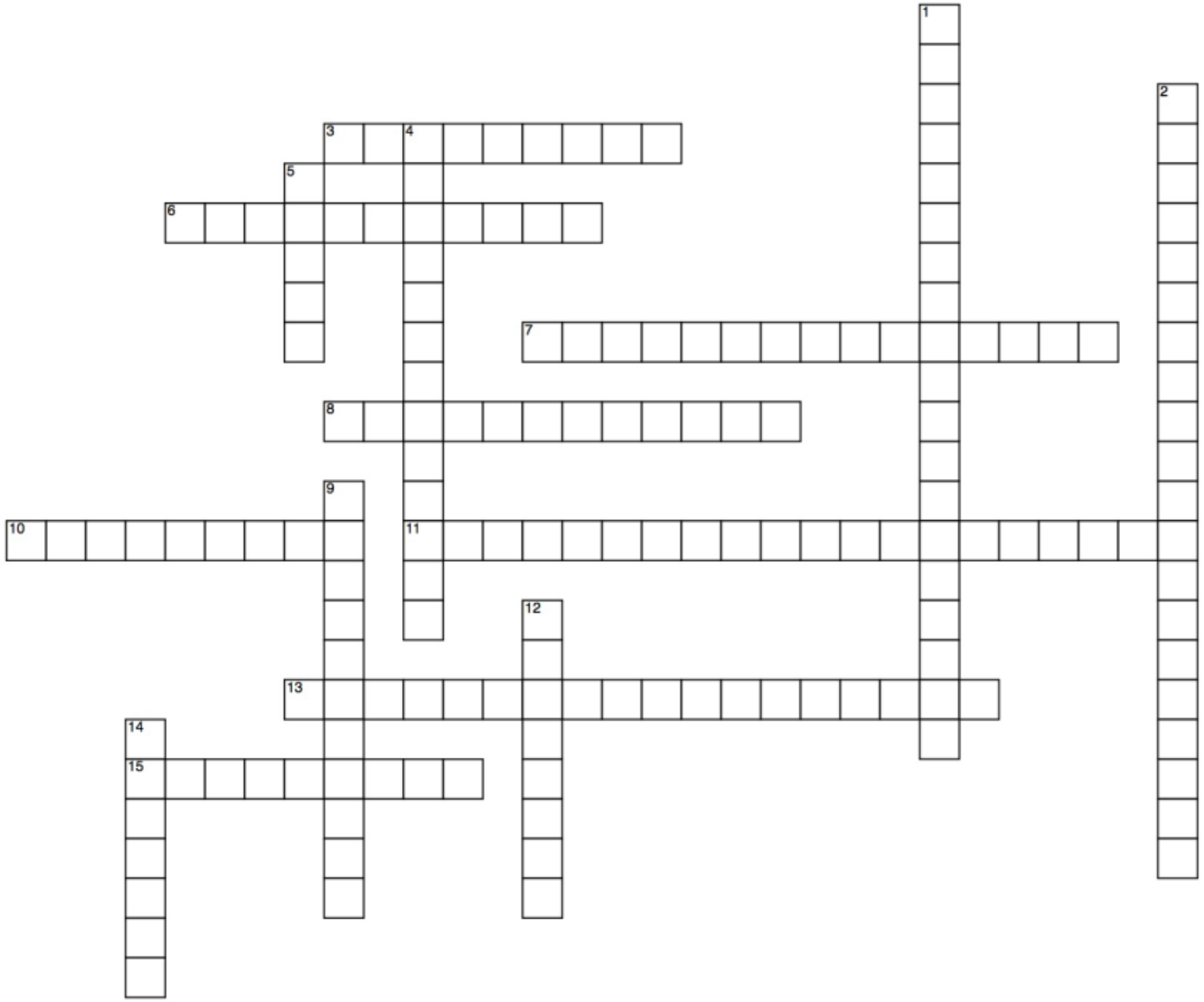
(Lecture 09)

- Ultimately, the selection of a licence will hinge on the specific needs and requirements of your project.
- By comprehending the advantages of each licence, you can make an informed decision that aligns with your goals and contributes to the open-source community.
- An open-source license grants users the rights:
  - The right to access and use the software's open-source code.
  - The right to modify the software's source code.
  - The right to distribute the software's source code and any modified version.
  - These rights are often subject to certain conditions, such as giving credit to the original author or contributing any modifications back to the community.

Cross Word Puzzle

SWM Design Principles and Patterns

Object orientation, SOLID and GOF Patterns Puzzle



ACROSS

3 Describes the use of plugins and APIs
6 Expose essential features while hiding irrelevant detail
7 Uninstantiated creational pattern
8 Adaptation to a specific usage
10 For creating one and only one instance
11 Clients should not be forced to depend on methods that they themselves do not use
13 If it quacks like a duck and swims like a duck but needs batteries you are probably breaking this principle
15 Flexible alternative to using inheritance

DOWN

1 High and low level modules should depend on abstractions
2 One class should do one thing
4 To modularise class functionality
5 To allow object behaviour to change at run time
9 Passing on behaviours from super classes to subclasses
12 One-to-many behavioural pattern
14 Is a wrapper for interfaces

Finally …

- What have you learned in the module and how does it help you in the future?
- Last word for the revision:
  - Lecture recordings are not the most reliable source of information.
  - You should use them to complement other more reliable resources.
  - There are many additional **resources links** in most of the slides, please make use of these resources which we have painstakingly watched in the preparation for this module.

Sample Exam Questions and Solution - Part 1

BY BRYANS

2022-23: Question 2

- a) Explain the Open-Closed principle in OOP. **[6 marks]**

- Sample Solution:
  - **Open for Extension**: The behavior of the module can be extended. [2 marks]
  - **Closed for Modification**: The source code of a module is inviolate, or no one is allowed to make changes to the source code. [2 marks]
  - "**Abstraction**" is the key in Open-Closed Principle. [2 marks]

2022-23: Question 2

- b) Use the Open-Closed principle to refactor the Java code below. **[8 marks]**

```
public double calculateSalary(Employee employee) {
  double salary = 0.0;
  if (employee is Developer) {
    Developer developer = (Developer) employee;
    salary += employee.workingHours * employee.hourlyRate;
  }
  else if (employee is TeamLeader) {
    TeamLeader teamLeader = (TeamLeader) employee;
    salary += teamLeader.workingHours * teamLeader.hourlyRate * teamLeader.bonus;
  }
  else if (employee is Recruiter) {
    Recruiter recruiter = (Recruiter) recruiter;
    salary += recruiter.workingHours * recruiter.hourlyRate * recruiter.bonus * 1.5;
  }

  return salary;
}
```

2022-23: Question 2

- Sample Solution:

```
public abstract class Employee {
  public abstract double calculateSalary();              [2 marks]
}
public class Developer : Employee {
  public double workingHours, hourlyRate;
  public double calculateSalary() {                      [2 marks]
      return workingHours * hourlyRate;
  }
}
public class TeamLeader : Employee {
  public double workingHours, hourlyRate, bonus;
  public double calculateSalary() {                      [2 marks]
      return workingHours * hourlyRate * bonus;
  }
}
public class Recruiter : Employee {
  public double workingHours, hourlyRate, bonus;
  public double calculateSalary() {                      [2 marks]
      return workingHours * hourlyRate * bonus * 1.5;
   }
}
```

2021-22: Question 3

- Animation is the basic class in JavaFX to define high-level animation. Animation is not restricted to movement and changing the background of a node over time is also considered an animation.

  a) Describe **FOUR** key concepts involved in JavaFX timeline animation. **[8 marks]**

  b) A client has requested you to code an animated square with the following conditions: **[11 marks]**

     i. The square has the width of 120 pixels, starting at the position (25, 175).

     ii. The final position of the square is at (145, 40) for a complete cycle of animation with ease in interpolator.

     iii. The square should start with an opacity value of 1 and end with the value of 0.2.

     iv. The total duration of the animation is 13 seconds.

     v. The number of repeats for animation is 5 times.

2021-22: Question 3

a) Describe **FOUR** key concepts involved in JavaFX timeline animation. **[8 marks]**

- First, look at the keywords: "Describe", "FOUR" and the marks.
- This explains each concept is 2 marks which you should provide the concept [1 mark] and its description [1 mark].
- Sample solution:
  - **Timeline**: Denotes the progression of time during animation with an associated key frame at a given instance.
  - **Key Frame**: Represents the state of the node being animated at a specific instant on the timeline.
  - **Key Value**: Represents the value of a property of a node along with an interpolator to be used.
  - **Transition**: Denotes the built-in transition animation to perform common animated effects.

2021-22: Question 3

b) A client has requested you to code an animated square with the following conditions: **[11 marks]**

i. The square has the width of 120 pixels, starting at the position (25, 175).

ii. The final position of the square is at (145, 40) for a complete cycle of animation with ease in interpolator.

iii. The square should start with an opacity value of 1 and end with the value of 0.2.

iv. The total duration of the animation is 13 seconds.

v. The number of repeats for animation is 5 times.

2021-22: Question 3

b) A client has requested you to code an animated square with the following conditions: **[11 marks]**

i. The square has the width of 120 pixels, starting at the position (25, 175). **[2 marks]**

```
Rectangle rectangle = new Rectangle(25, 175, 120, 120);
```

ii. The final position of the square is at (145, 40) for a complete cycle of animation with ease in interpolator. **[2 marks]**

```
KeyValue xValue = new KeyValue(rectangle.xProperty(), 145 - 25, Interpolator.EASE_IN);
KeyValue yValue = new KeyValue(rectangle.yProperty(), 40 - 175, Interpolator.EASE_IN);
```

ii. The square should start with an opacity value of 1 and end with the value of 0.2. **[2 marks]**

```
KeyValue oValue = new KeyValue(rectangle.opacityProperty(), 0.2);
```

2021-22: Question 3

b) A client has requested you to code an animated square with the following conditions: **[11 marks]**

i. The total duration of the animation is 13 seconds. **[2 marks]**

```
KeyFrame keyFrame = new KeyFrame(Duration.millis(13000), xValue, yValue, oValue);
```

iv. The number of repeats for animation is 5 times. **[3 marks]**

```
Timeline timeline = new Timeline();
timeline.setCycleCount(5);
timeline.getKeyFrames().addAll(keyFrame);
timeline.play();
```

2020-21: Question 2

- There are six relationships in class diagrams where classes are interrelated to each other in specific ways. Provide Java sample code and discuss **TWO** differences between "Aggregation" and "Composition". **[6 marks]**
- Sample solution:
- Tip:
  - Explain with code.

```java
public class CarFactory {
    Car car;

    private void manufactureCar(Car car) {
        this.car = car;
    }
}
```

```java
public class CarFactory {
    Car car;

    private void manufactureCar() {
        this.car = new Car();
    }
}
```

Aggregation [2 marks]	Composition [2 marks]
Weaker Association.	Stronger association. Creating an object of a class `Car` inside class `CarFactory`.
The object `car` still exist even the class `CarFactory` is deleted.	The object `car` will not exist if the class `CarFactory` is deleted.

2020-21: Question 3

- Explain the differences between continuous integration (CI), continuous delivery (CD) and continuous deployment. **[6 marks]**
- Sample solution:
  - CI is a software engineering practice in which developers integrate code into a shared repository several times a day to obtain rapid feedback of the feasibility of that code (and/or to avoid conflict) **[1 mark]**. CI enables automated build and testing so that teams can rapidly work on a single project together **[1 mark]**.
  - CD is a software engineering practice in which teams develop, build, test, and release software in short cycles **[1 mark]**. It depends on automation at every stage so that cycles can be both quick and reliable **[1 mark]**.
  - Continuous Deployment is the process by which qualified changes in software code or architecture are deployed to production as soon as they are ready [1 mark] and without human intervention [1 mark].
- Tip:
  - You need to **write a full sentence** although keywords are the key to the marks.

2019-20: Question 3

- Code refactoring refers to changes made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviors.

  a) Identify **3 issues** in the code snippets below that **requires code refactoring**, assuming codes are commented and documented appropriately. **[4 marks]**

  b) Provide **solutions to reflect the changes of above issues** based on code refactoring methods you have learnt. **[4 marks]**

2019-20: Question 3

```
public class GalleryApp {
     public static void main(String[] args) {
          Art a1 = new Art();
          a1.setArtName("Mona Lisa");
          a1.setArtPrice(300000000);
          a1.setArtSizeX(77);
          a1.setArtSizeY(53);
          a1.setArtArtist("Leonardo Da Vinci");
          Gallery g = new Gallery();
          g.addArt(a1);
          g.displayArts();
     }
}
```

```
public class Gallery {
     private Art[] arts;
     private int count;
     private ArrayList<String> employeeName;
     private ArrayList<Integer> employeeId;

     public Gallery() {
          arts = new Art[20];
          count = 0;
          employeeName = new ArrayList<>();
          employeeId = new ArrayList<>();
     }
     public void addArt(Art art) { … }
     public void displayArts() { … }
     public void addEmployeeName(String name) { … }
     public void addEmployeeId(int id) { … }
     public void printAllEmployeeInfo() { … }
}
```

2019-20: Question 3

```
public class Art {
      public String n, a;
      public double p;
      public int x, y;
      public static int numOfArt;

      public Art() { … }
      public void setArtName(String n) { … }
      public void setArtPrice(double p) { … }
      public void setArtSizeX(int x) { … }
      public void setArtSizeY(int y) { … }
      public void setArtArtist(String a) { … }
      public String getArtInfo() { … }
}
```

2019-20: Question 3

a) Identify **3 issues** in the code snippets below that **requires code refactoring**, assuming codes are commented and documented appropriately. **[4 marks]**

- Sample solution:
  1) The class Gallery has poor cohesion, all employee related attributes and method should be separated into another class.
  2) Instance variables are public, encapsulation (information hiding) concept should be applied.
  3) Naming convention issue in class Art.

2019-20: Question 3

b) Provide **solutions to reflect the changes of above issues** based on code refactoring methods you have learnt. **[4 marks]**

- Sample solution:

  1) The class Gallery has poor cohesion, all employee related attributes and method should be separated into another class.

```
public class Employee {
      private String name;
      private int id;

      public Employee(String name, int id) {
            this.name = name;
            this.id = id;
      }
}
```

2019-20: Question 3

b) Provide **solutions to reflect the changes of above issues** based on code refactoring methods you have learnt. **[4 marks]**

- Sample solution:
  1) Instance variables are public, encapsulation (information hiding) concept should be applied.
  2) Naming convention issue in class Art.

```
private String name, artist;
private double price;
private int sizeX, sizeY;
```

```
public class Art {
    public String n, a;
    public double p;
    public int x, y;
    …
}
```

University of Nottingham
UK | CHINA | MALAYSIA

Exam Revision Part 2

MORE ON DESIGN PATTERNS, AND DESIGN PRINCIPLES

BY HENG YU

Format of Exam

- Questions for design patterns, design principles, and general refactoring skills.
  - Concepts of design principles and design patterns.
  - Review their examples provided in lectures and labs.
  - Short text answers + Java coding.

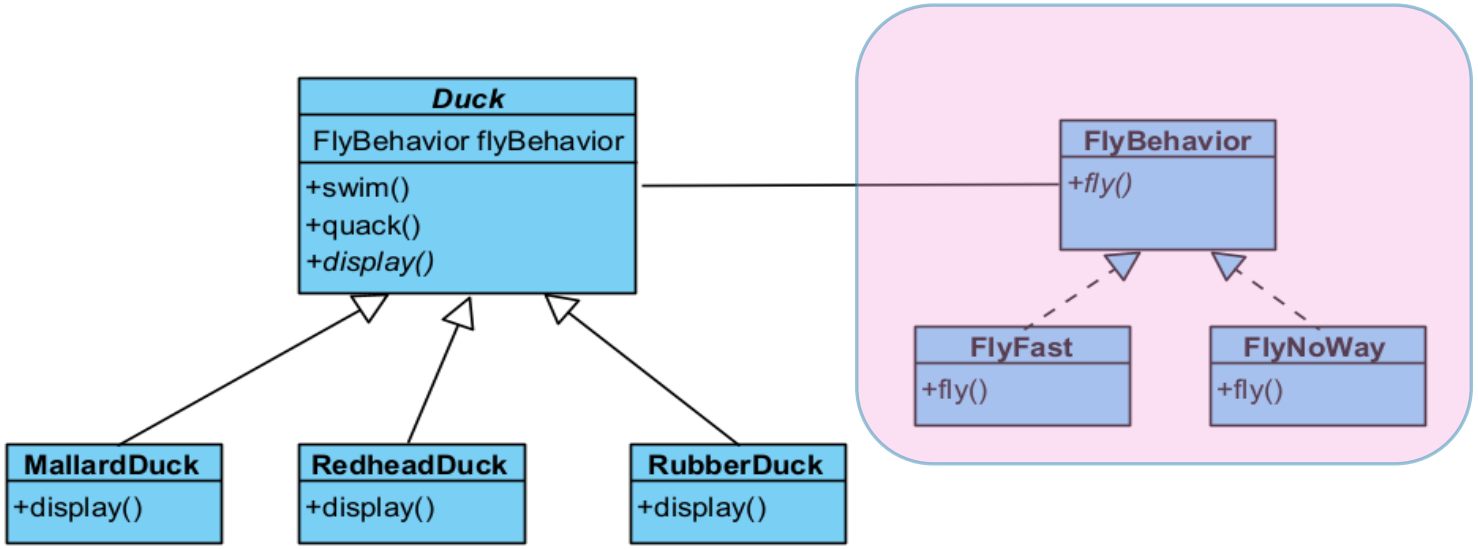
Principles and Patterns

- SOLID and other Principles.
  - You should understand and be able to explain the idea behind the principles.

- Design Patterns.
  - You should understand the ones we discussed in the lectures.
    - Strategy; Decorator; Singleton; Factory Method; Adapter; Observer; State; MVC (see Lecture 07).
  - You should be able to identify and describe core design patterns in class diagrams.
    - Only the ones we have gone through.

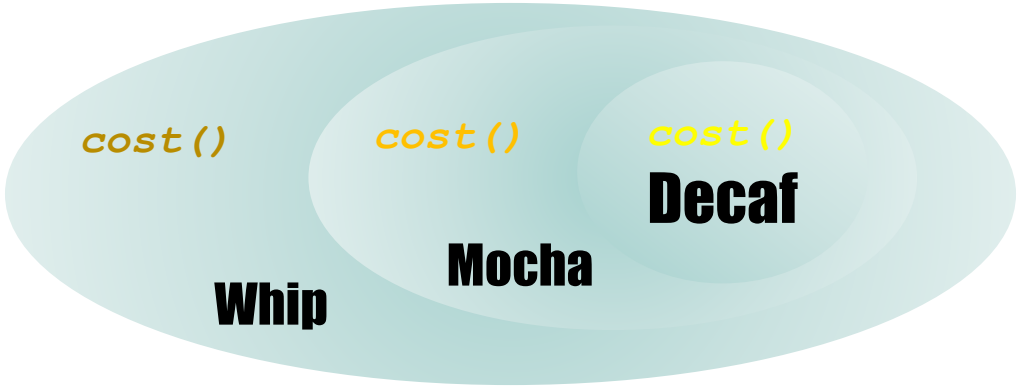
Tips on Strategy Pattern

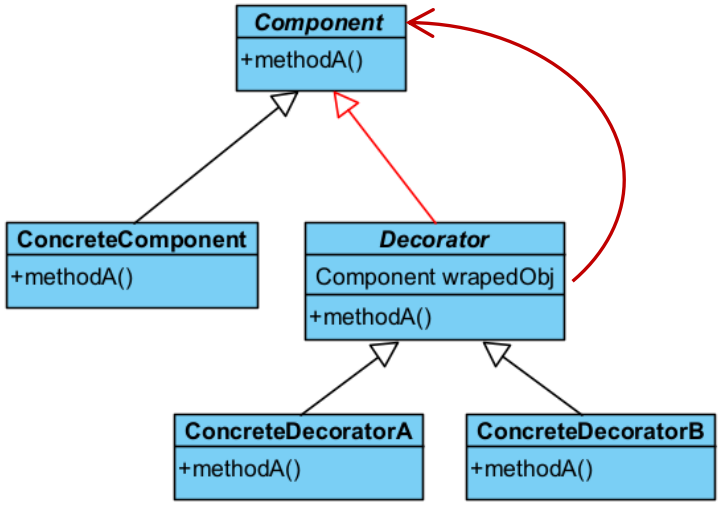
- Used when subclasses may exhibit various behaviors.
- To implement a strategy pattern, create an algorithm interface (FlyBehaviors) whose various implementing classes (behaviors) are adopted by various 'Duck' subclasses that follows different algorithms (fly behaviors).

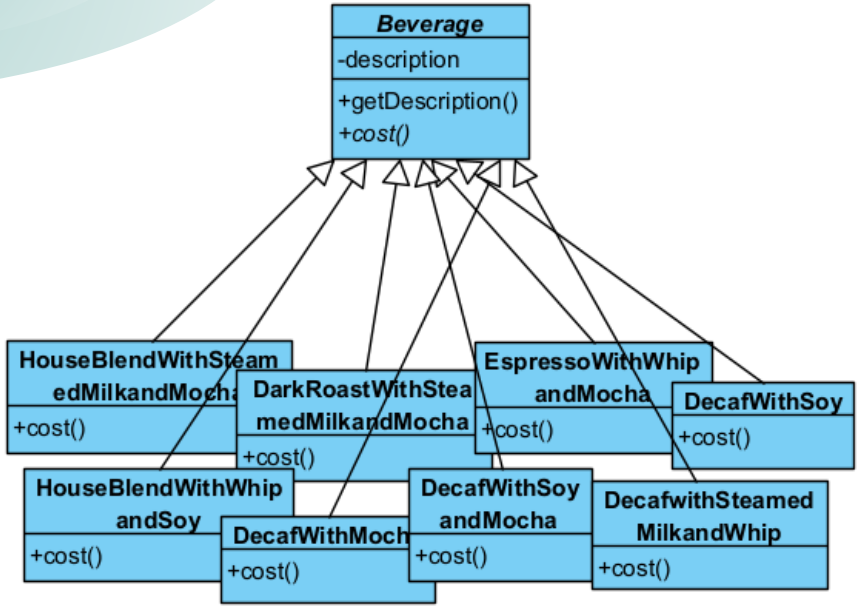


Tips on Decorator Pattern

- Applied under combinatorial explosion of subclasses. Decorators provide a flexible alternative to subclassing for extending functionality
- To implement a decorator pattern, the decorator should be a subclass of the component to be decorated, should contain the component inside, and should implement the a same method as the component's behavior.

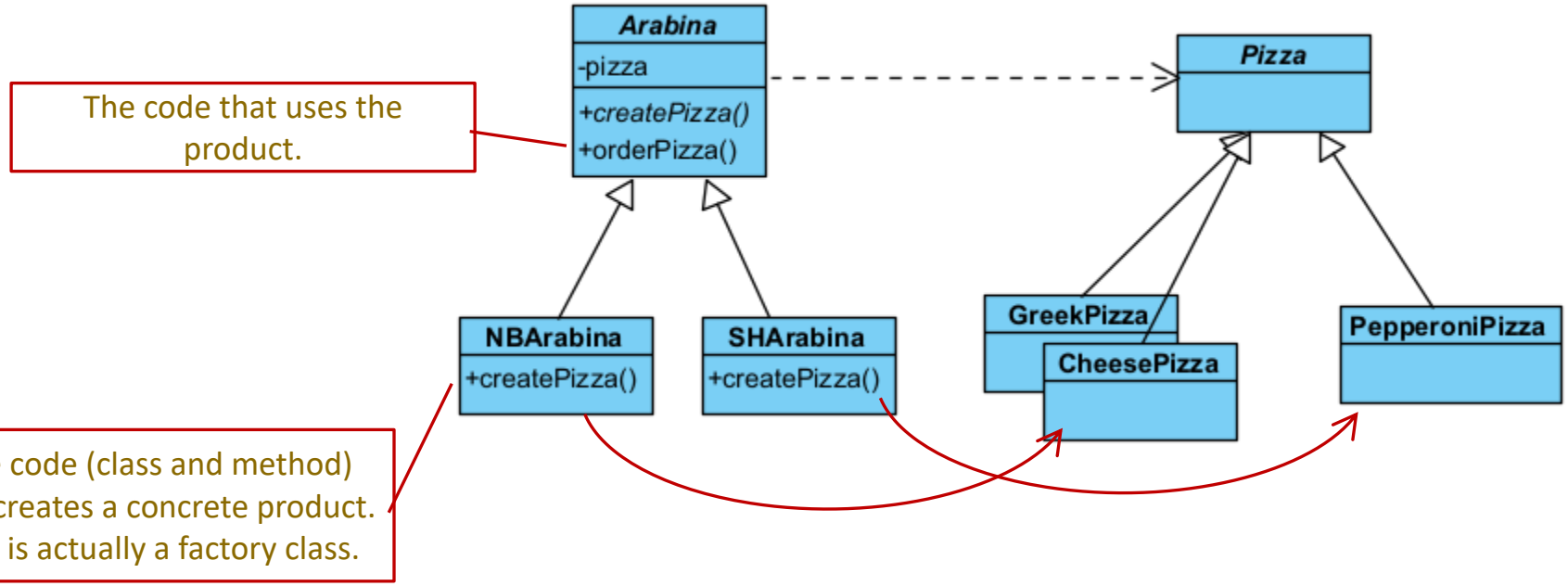






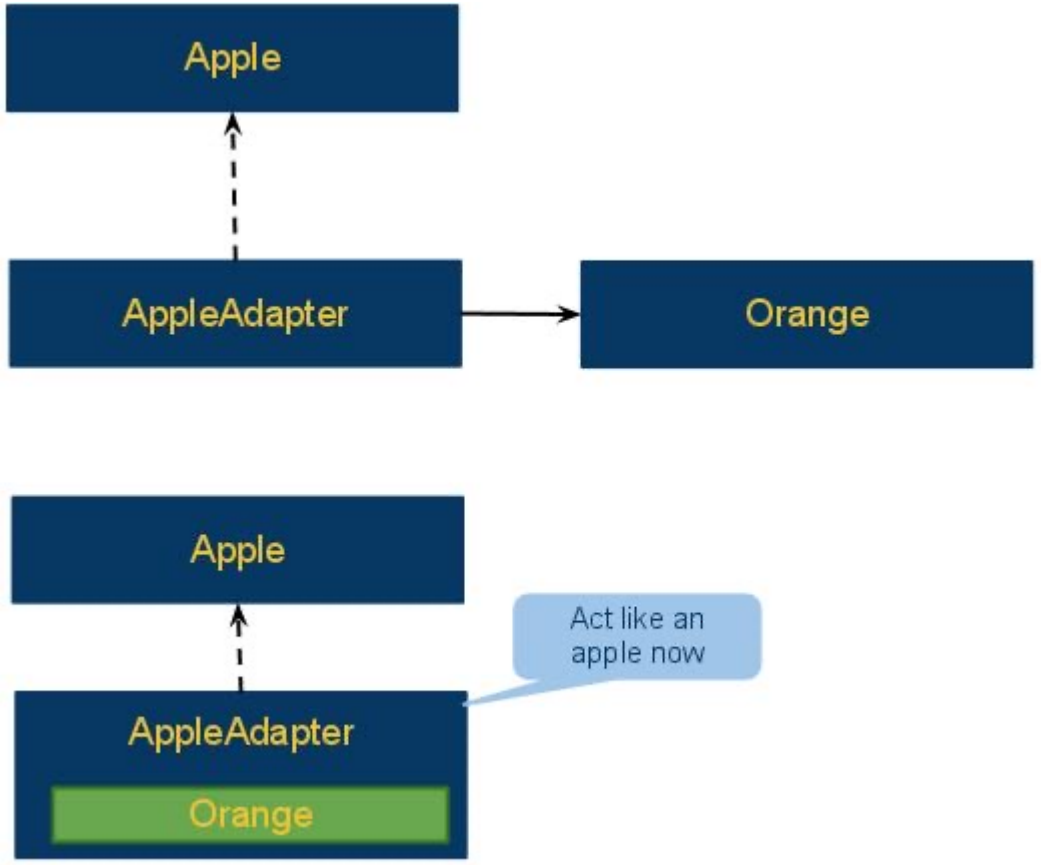
Tips on Factory Method Pattern

- Usage: hide the creation logic of the classes from the caller.
- The Factory Method Pattern defines an *interface* for creating objects, but lets *subclasses* decide which class to instantiate.



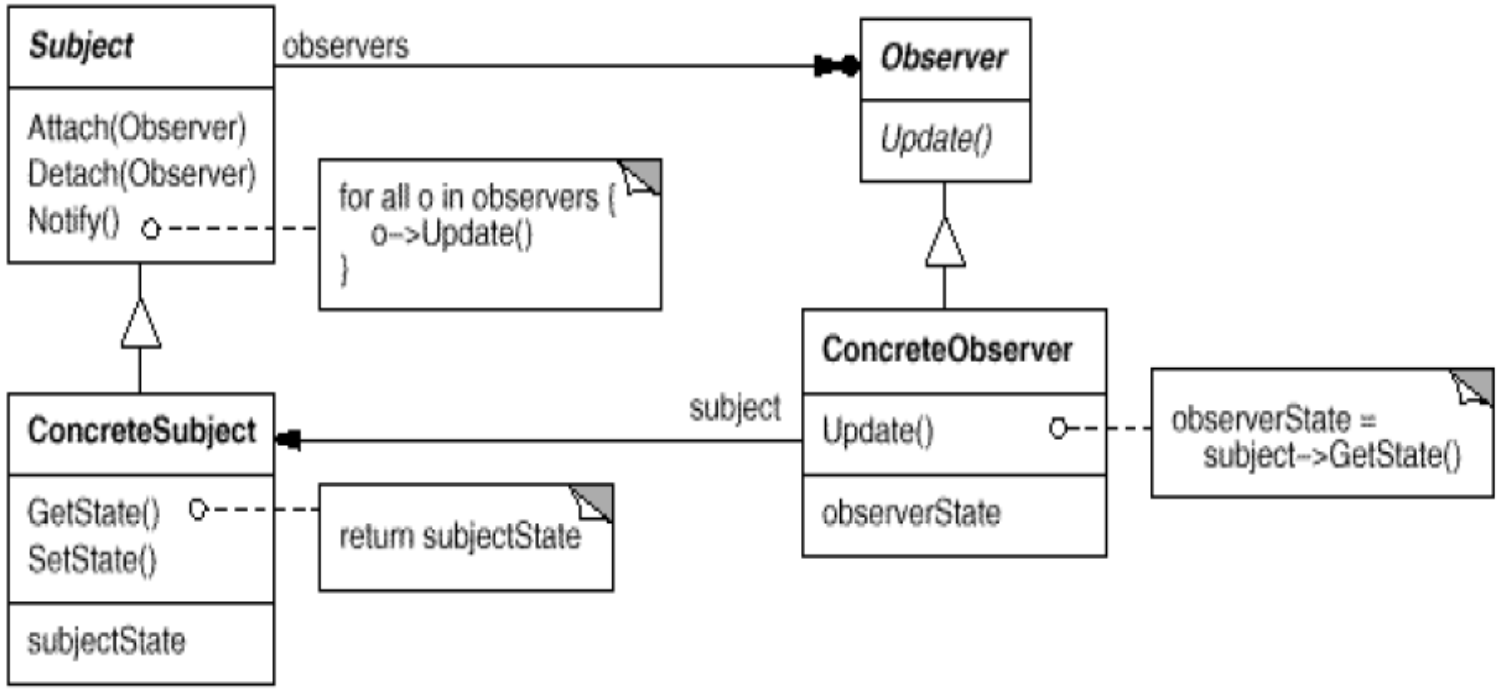
Tips on Adapter Pattern

- Used when to merge the functionalities of two unrelated products/classes.
- If merge O to A, create an AAdapter, which is a kind of A.
- This AAdapter should contain an O object to behave O's functionality.
- Then, put the AAdapter into the field of A, such that A can use the O object in the AAdapter to behave O's functionality.



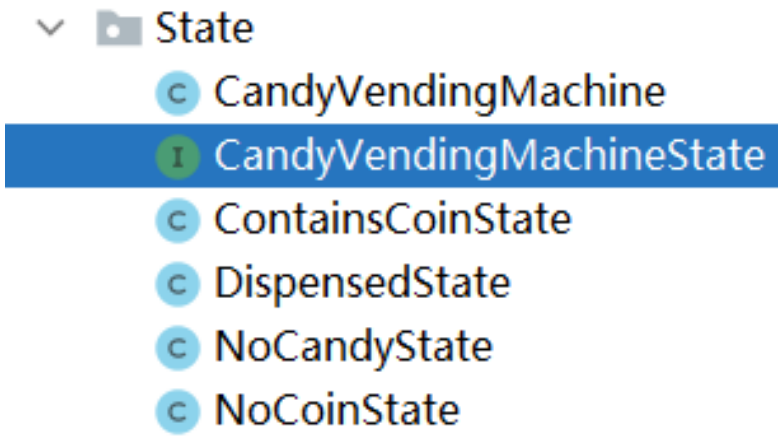
Tips on Observer Pattern

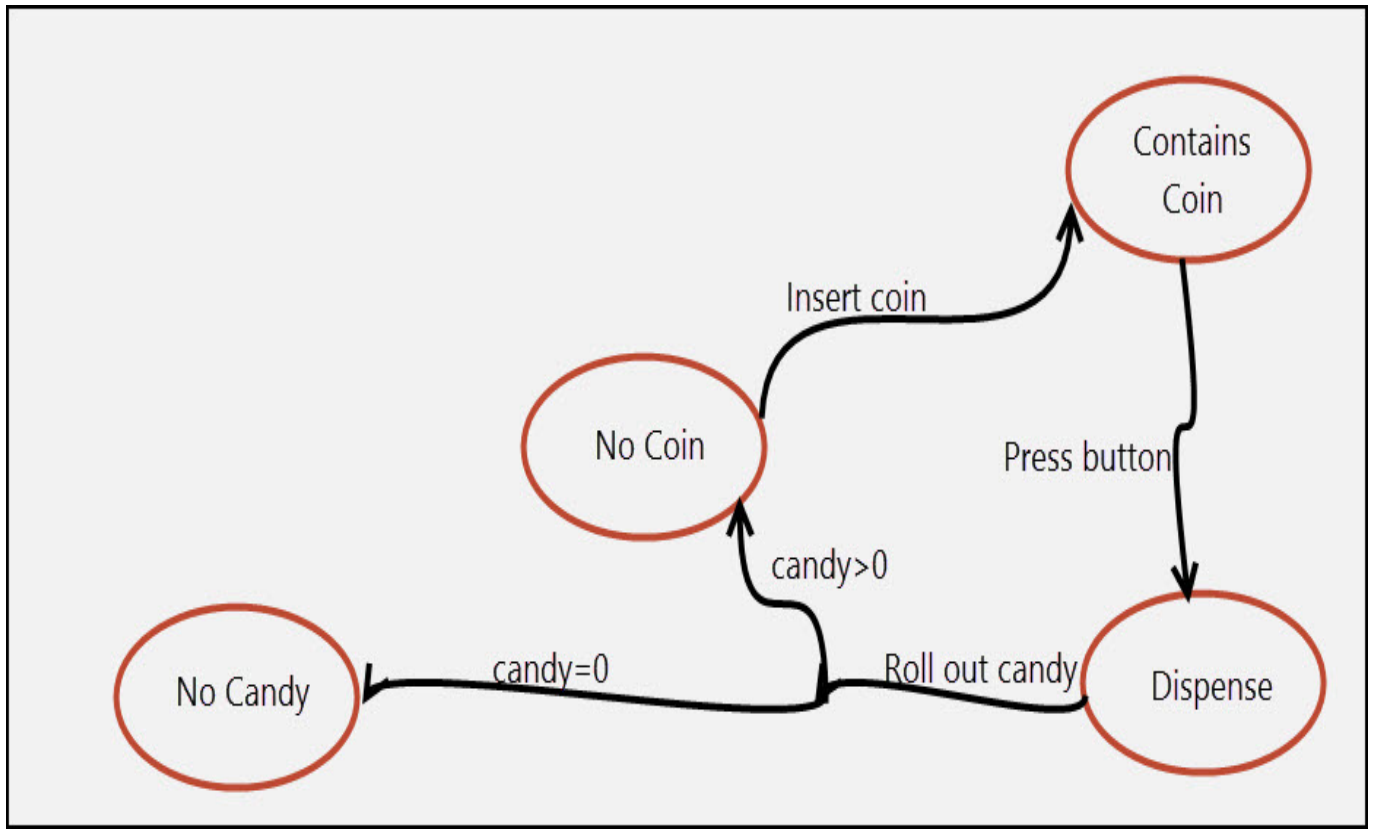
- Used when any changes of a state/value in the Subject need to be immediately known by the Observers.
- Subject should contain a list of observers, attach and detach methods, and a `notify()` method.
- Observer should contain an `update()` method to handle the state/value change, and a reference of the subject to provide the changed state/value.



Tips on State Pattern

- State diagram contains states and transitions.
- Each individual state should be defined as an individual state-class.





Tips on State Pattern

- Each transition (edge) should be implemented by all state-classes, as methods.
- Each method implemented in each state should reflect its behavior specific to this state.
- E.g., in ContainsCoinState, if a button is pressed, then enters the DispenseState.

```java
public interface CandyVendingMachineState {
    void insertCoin();
    void pressButton();
    void dispense();
}
```

```java
public class ContainsCoinState implements CandyVendingMachineState{
    CandyVendingMachine machine;
    @Override
    public void insertCoin() {
        System.out.println("Coin already inserted");
    }
    @Override
    public void pressButton() {
        machine.setState(machine.getDispensedState());
    }
    @Override
    public void dispense() {
        System.out.println("Press button to dispense");
    }
    @Override
    public String toString(){
        return "ContainsCoinState";
    }
```

Tips on Singleton Pattern

- Used when the project logic requires a class to have only one instance at any time.
- Make the constructor private.
- In the class, define a private static object, named `instance`.
- Define a `getInstance()` method, for the outside world to access the instance.

```
public class SingleObject {
    //create an object of SingleObject
    private static SingleObject instance = new SingleObject();

    //make the constructor private so that this class cannot be
    //instantiated
    private SingleObject(){}

    //Get the only object available
    public static SingleObject getInstance(){
        return instance;
    }
```

How to Prepare

- Understand the lecture slides on design principles & patterns, and refactoring including code smells.
- Understand concepts and examples in lab manuals.
- Exercise on the past year exam papers (We provide sample solutions to some questions).
- Go through all the external links that appear in lecture slides and lab manuals.

University of Nottingham
UK | CHINA | MALAYSIA

Coursework

HIGH LEVEL FEEDBACK

Coursework Feedback

- Feedback will be given through Moodle page under “Coursework” section.
- An appeal period of one week will be provided after the coursework marks are released.
  - A link to submit appeals via MS Form will be made available.

Exam Formalities

25% OF THE MODULE

Format of Exam

- Exam worth 25%
  - Location: Siyuan Auditorium
  - Date & Time: 8th January 2024, 14:00 – 15:00
- Format
  - Text answers; code snippet writing; etc.
  - You may be asked to draw class diagrams and write the OO Java code accordingly. Minor syntax errors can be tolerated, but largely mistakes or pseudo code format are not acceptable.
  - Mostly testing knowledge and application.

SET/SEM

- Start: 18th November 2024
- End: 13th December 2024

- https://bluecastle-cn-surveys.nottingham.ac.uk

Put Your Mind in Maintenance Mode