Design Patterns II, Documentation and Serialisation

Alma Rahat

SOFT251: Object Oriented Programming
SOFT252: Object oriented Software Engineering with Design Patterns

1 November 2018

COMPUTING WITH PLYMOUTH UNIVERSITY

What did we cover last week?

1. Poll results!
2. Recap.
3. Introduction to design patterns.
4. Observer pattern.
5. Model-View-Controller architecture.

Outline

Housekeeping

- Are the design patterns relevant to SOFT251?
  $\Rightarrow$ Yes, patterns are fundamental to object oriented design and programming. You will be assessed on these. The only difference between SOFT251 and SOFT252 is: the students in SOFT251 will be asked to develop a web application while the students in SOFT252 will be asked to develop a standalone desktop application.

Housekeeping

- Are the design patterns relevant to SOFT251?
  $\Rightarrow$ Yes, patterns are fundamental to object oriented design and programming. You will be assessed on these. The only difference between SOFT251 and SOFT252 is: the students in SOFT251 will be asked to develop a web application while the students in SOFT252 will be asked to develop a standalone desktop application.
- The date for in-class test: **29 November 2018**.
  - You will be informed where to go for the test nearer the time.
  - There will be no lecture on the day!

Housekeeping

- Are the design patterns relevant to SOFT251?
  $\Rightarrow$ Yes, patterns are fundamental to object oriented design and programming. You will be assessed on these. The only difference between SOFT251 and SOFT252 is: the students in SOFT251 will be asked to develop a web application while the students in SOFT252 will be asked to develop a standalone desktop application.
- The date for in-class test: **29 November 2018**.
  - You will be informed where to go for the test nearer the time.
  - There will be no lecture on the day!
  - What to expect?
    - 30% of the total module marks.
    - Multiple choice questions.
    - Short answer questions.

Strategy Pattern

Scenario

Original image

120kB

compression

16-colour bitmap

gif

80kB

70kB

I am interested in compressing an image to reduce its size for storage and web communication.

A naive implementation

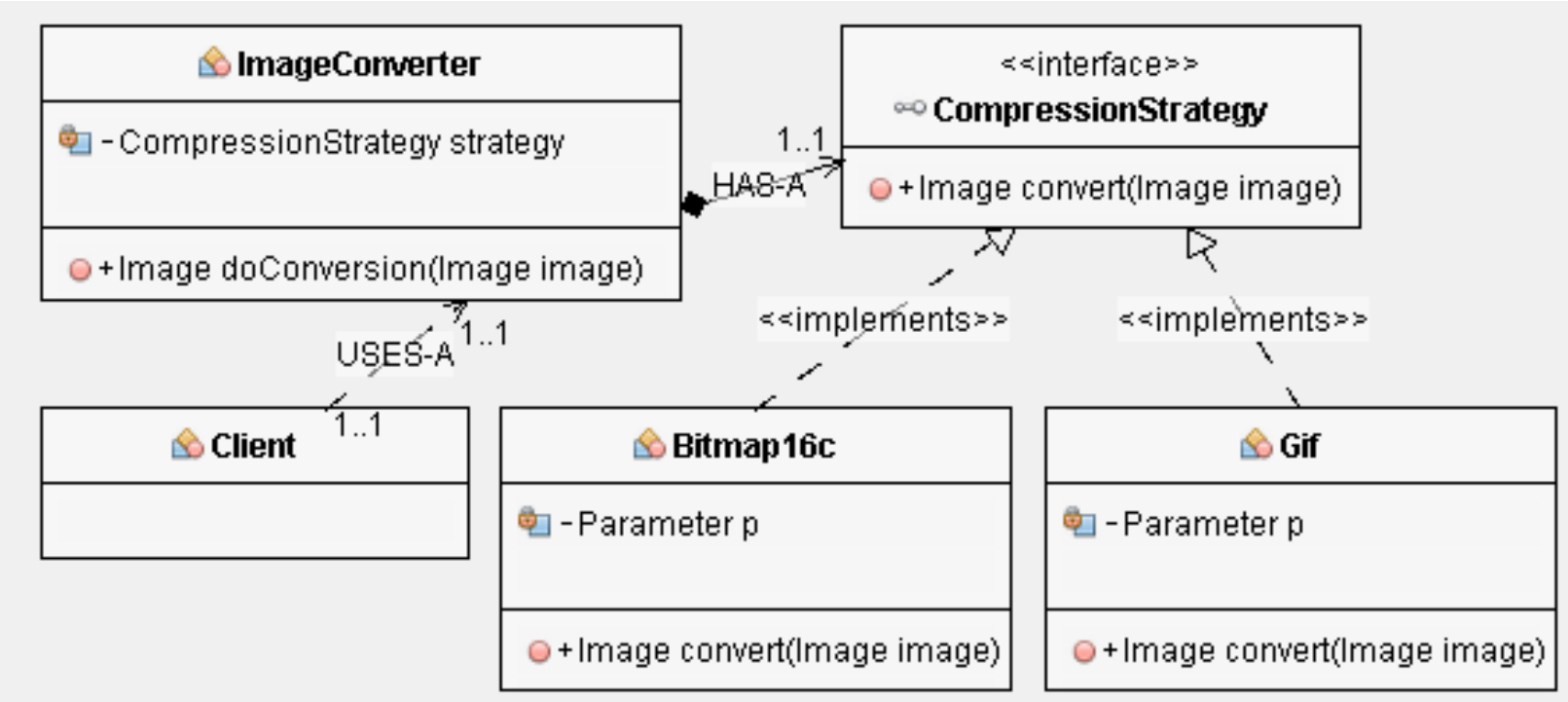
```
public class Converter{
   public void convert(String condition){
        switch(condition){
          case "bitmap":
            // convert to bitmap
            break;
          case "gif":
            // convert to gif
            break;
          default:
            System.out.println("Invalid format")
        }
   }
}
```

A naive implementation

```
public class Converter{
   public void convert(String condition){
        switch(condition){
          case "bitmap":
            // convert to bitmap
            break;
          case "gif":
            // convert to gif
            break;
          default:
            System.out.println("Invalid format")
        }
   }
}
```

What if I would like to add a new method for conversion?

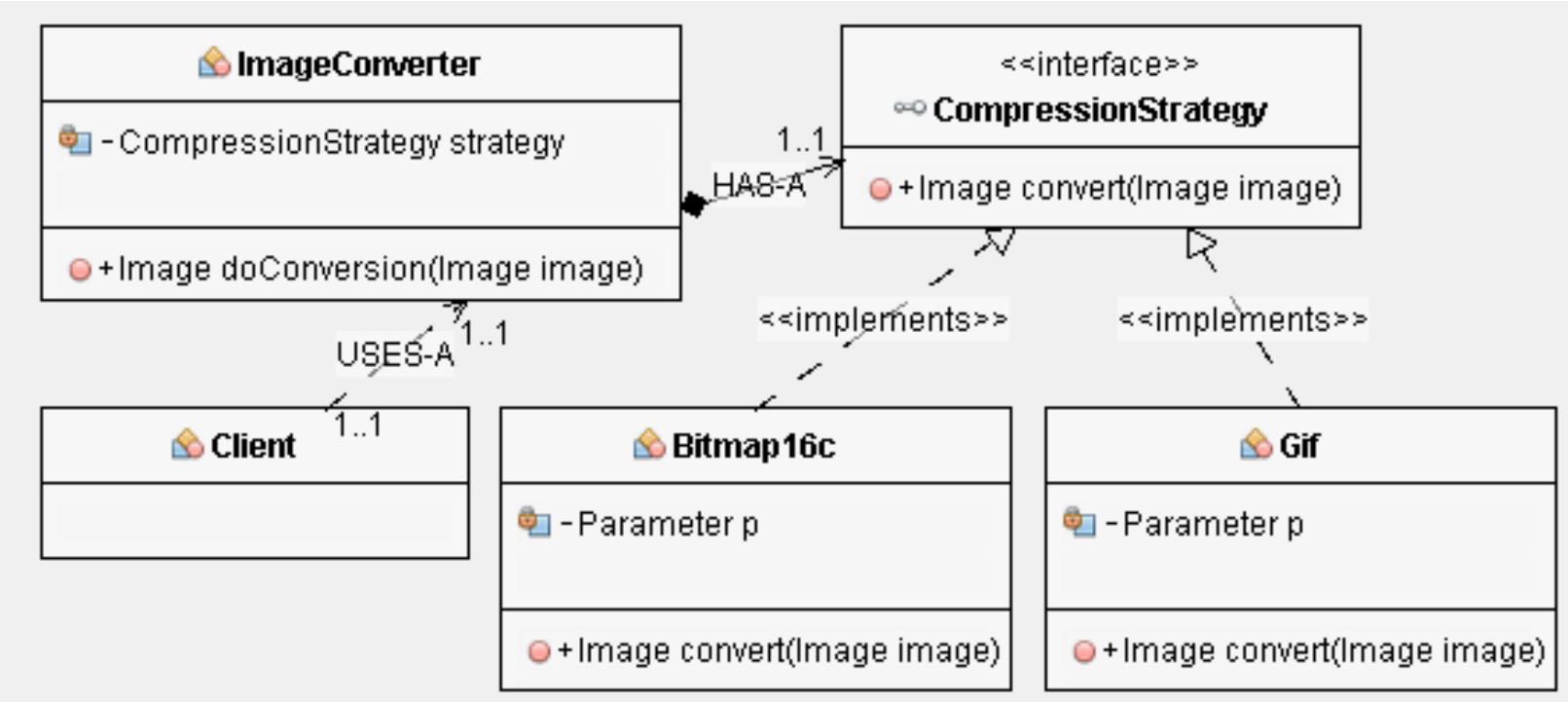
A better implementation



A better implementation

Context

Strategy interface



Concrete strategies

- Coding to interfaces not to concrete implementations.
- Open-closed principle.
- Can easily add new methods for conversion.

A better implementation

```
public class ImageConverter{
  private CompressionStrategy strategy;
  ImageConverter(Parameters p){
    strategy = new CompressionStrategy(p);
  }
  public boolean doConversion(Image image){
    Image retImage = strategy.convert(image);
    retImage.save();
    return true;
  }
}
public interface CompressionStrategy{
  public Image convert(Image image);
}
public class Bitmap16c implements CompressionStrategy{
  private Parameter parameter;
  Bitmap16c(Parameter p){
    parameter = p;
  }
  public Image convert(Image image){
    // do something with image
    return image;
  }
}
```

Strategy pattern

Type Behavioural

Pattern name Strategy (a.k.a. policy)

Problem When there is a family of related and interchangeable algorithms, and it is necessary to choose one during runtime.

Solution Create a single interface encapsulating the common behaviour, and create a specific concrete class for each behaviour implementing the interface.

Consequences Advantages and disadvantages of using the pattern:

- Add new algorithms as and when available without altering the existing classes.
- Eliminates conditional statements (e.g. using *switch*).
- An alternative to subclassing.
- Clients must be aware of different strategies.

Known uses Implementing multiple sorting algorithms.

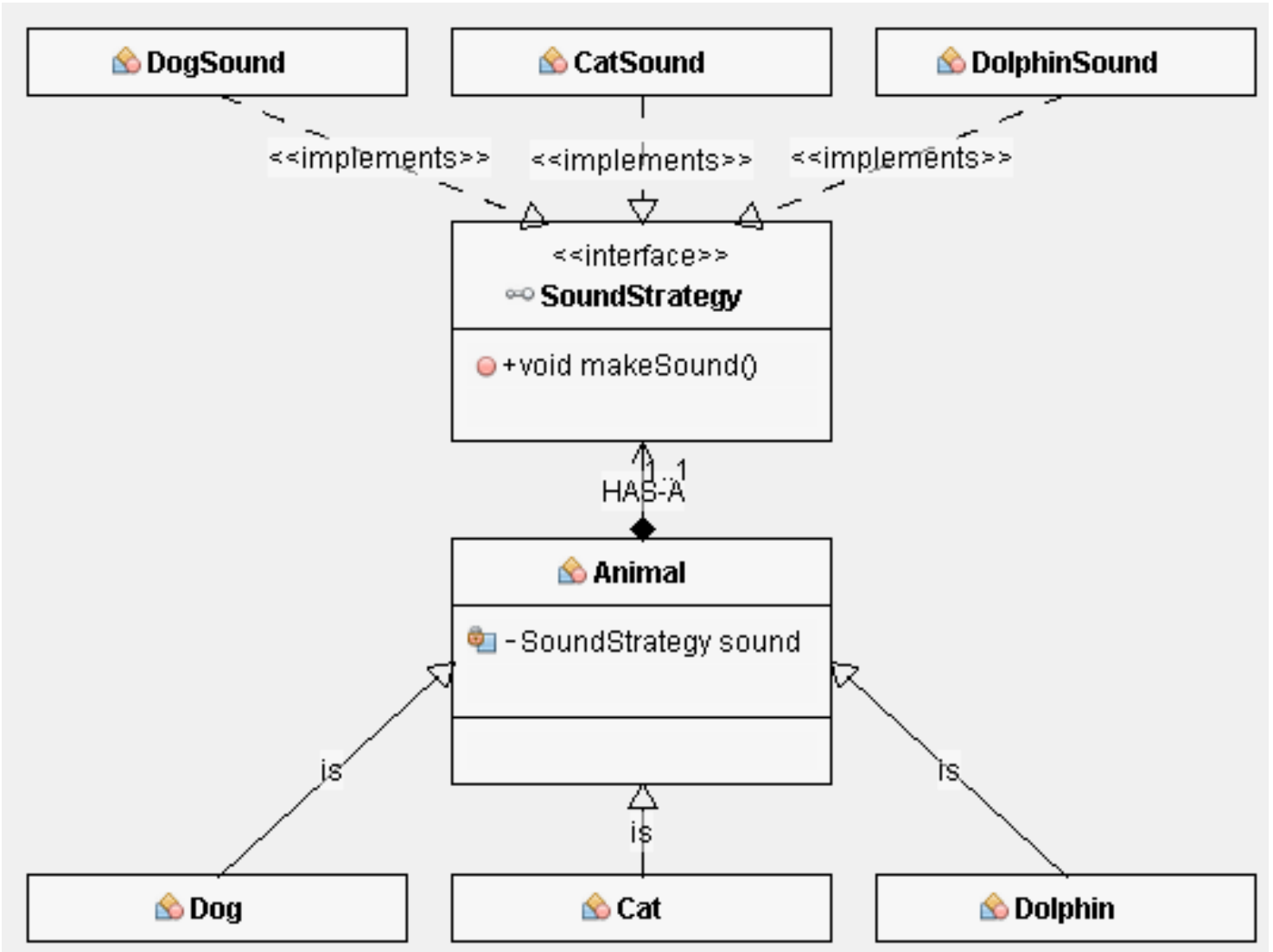
Any questions?

Exercise I

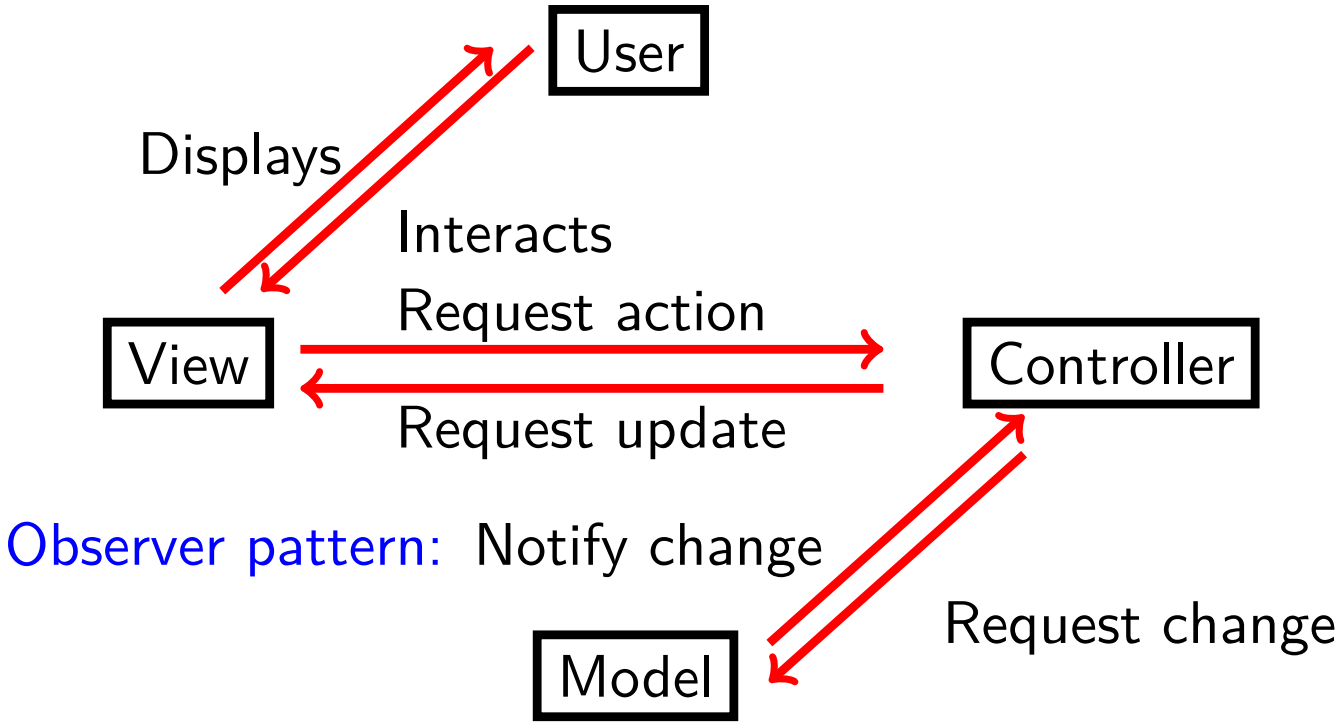
A company has approached you to develop an animal sound simulator. The software will be used to teach young children about different sounds that various animals make. The company insists that the simulator must have dogs, cats, and dolphins.

Can you apply the Strategy pattern in this case? If so, sketch a UML diagram (now), and write a Java program adhering to your design (homework).

Exercise I


DogSound
CatSound
DolphinSound
<<implements>>
<<implements>>
<<implements>>
<<interface>>
SoundStrategy
+void makeSound()
1..1
HAS-A
Animal
- SoundStrategy sound
is
is
is
Dog
Cat
Dolphin

Patterns in MVC

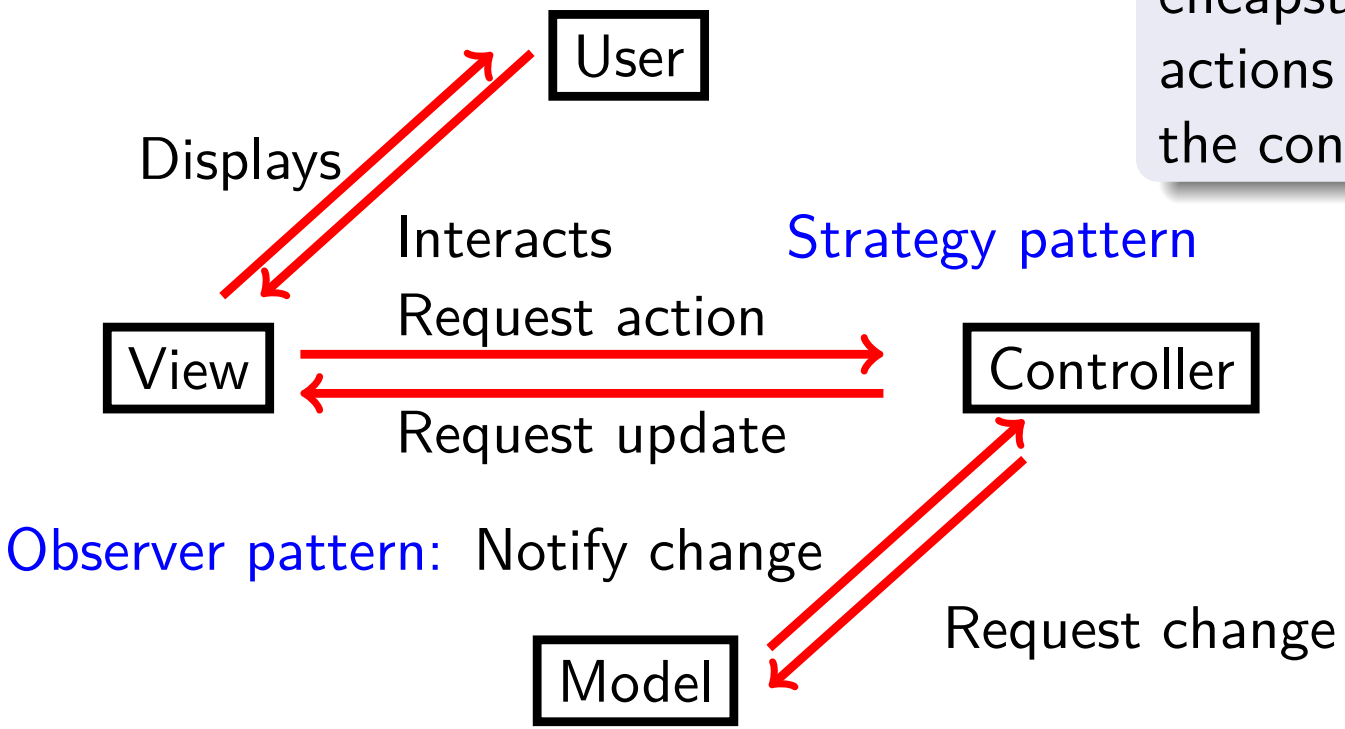


Data, logic and rules of the application

Do you notice an opportunity to use the Strategy pattern?

Patterns in MVC

Strategy pattern can encapsulate the actions performed by the controller.



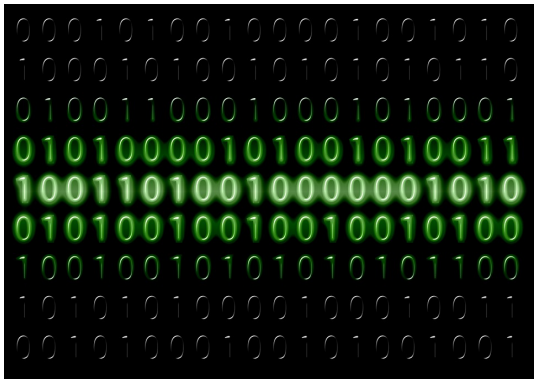
Data, logic and rules of the application

Do you notice an opportunity to use the Strategy pattern?

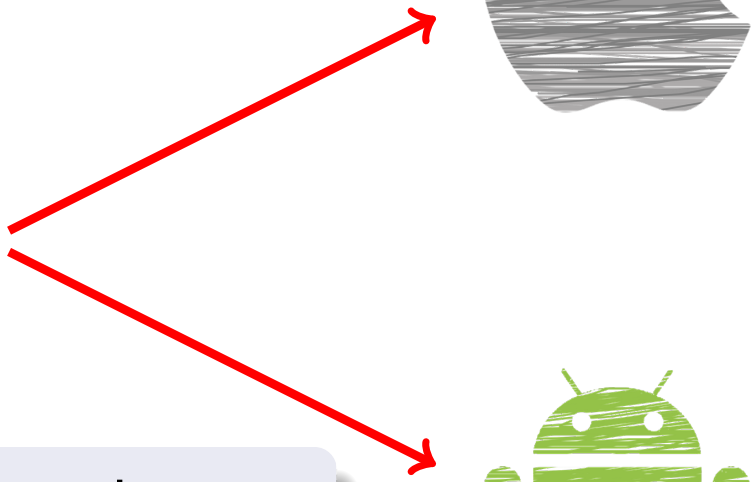
Template Method Pattern

Scenario

Code



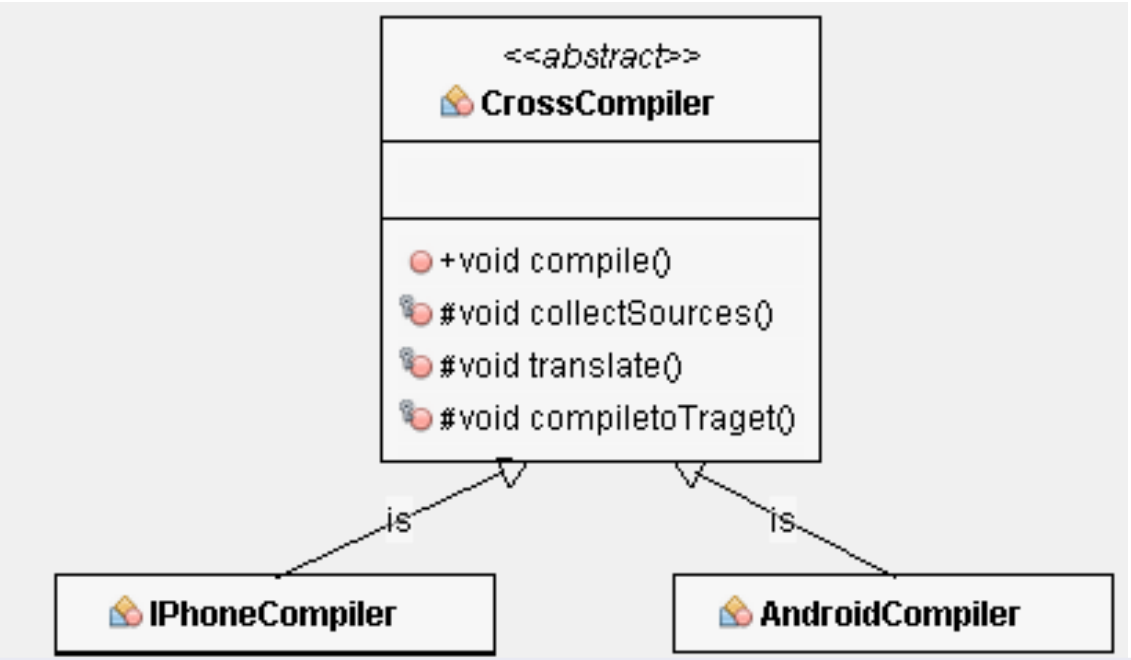
Compilation



We have developed a new programming language, and now we are developing a compiler for various platforms. Compilation for this can be performed in three steps:

- collect sources
- translate to intermediary language
- compile to target

A good implementation



- Template method – the method that contains the skeleton algorithm – compile (public method).
- Template method should be *final*: no subclass should be able to override it.
- A common method here is **translate**. This may be overridden: this will then be called a *hook*. (protected method)
- Other methods are abstract (and protected).

A good implementation

```
public abstract class CrossCompiler{
  public final void compile(){
    collectSources();
    translate();
    compileToTarget();
  }
  protected void translate(){
    // code to translate
  }
  protected abstract void collectSources();
  protected abstract void compiletoTarget();
}
public class IPhoneCompiler extends CrossCompiler{
  protected void collectSources(){
    // code for platform specific collection
  }
  protected void compileToTarget(){
    // code for platform specific compilation
  }
}
```

Template method pattern

Type Behavioural

Pattern name Template Method

Problem When we want the same algorithm to vary in steps.

Solution Define outline or skeleton of an algorithm in a superclass and ask subclasses to provide concrete implementations of the steps that vary.

Consequences Advantages and disadvantages of using the pattern:

- Uses abstract classes, inheritance and method overriding: promotes code reuse.
- Eliminates conditional statements (e.g. using *switch*).
- Dependency structure may be complex.
- It may be difficult to understand program flow.

Known uses Implementing document parser.

Distinction between Strategy and Template

Strategy Pattern	Template Method Pattern
Used when there is a family of interchangeable algorithms.	Used when there are variations in steps of a particular algorithm.
Uses composition.	Uses inheritance.
More flexible.	More efficient.

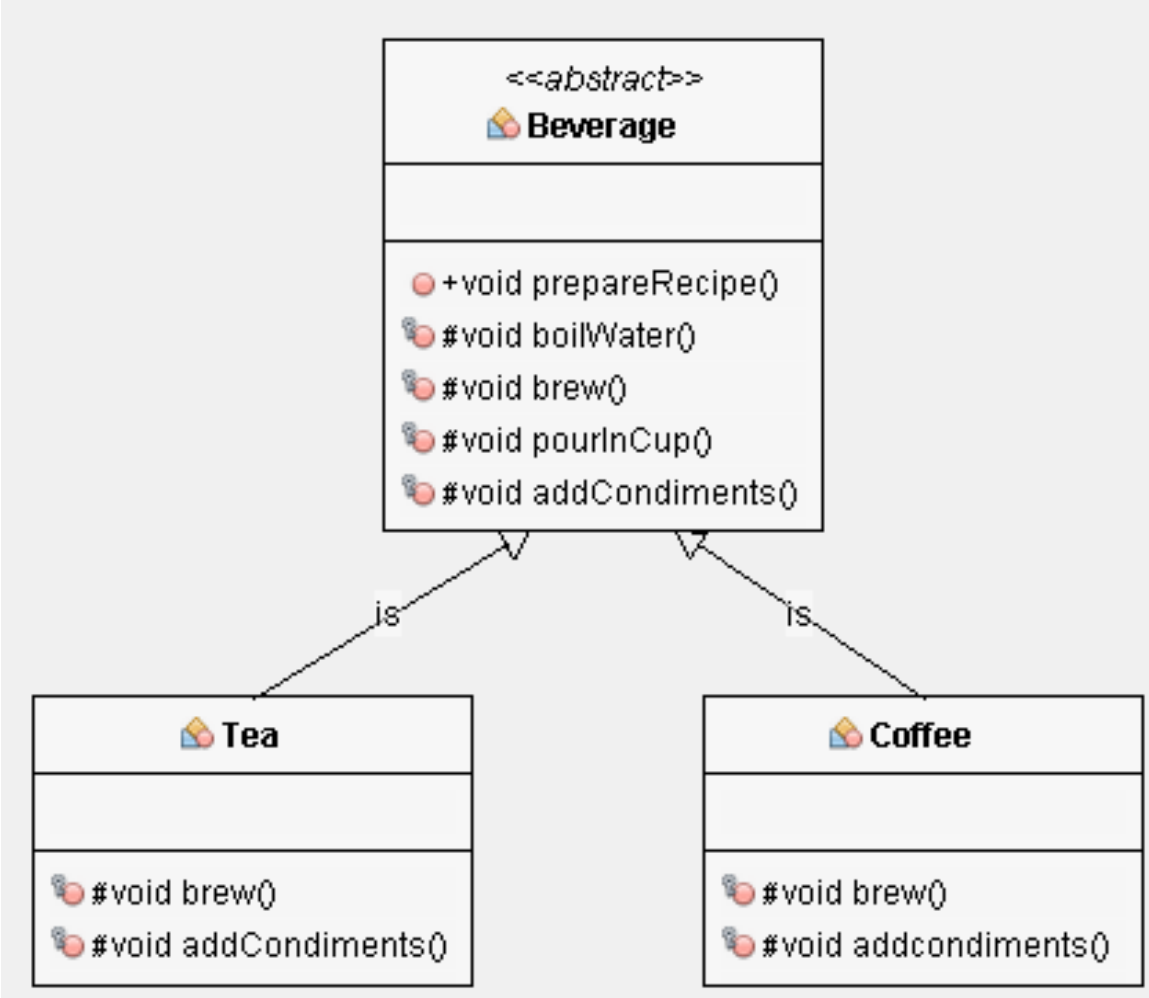
Prefer Strategy over Template (if it comes to it)!

Any questions?

Exercise II

A café in Plymouth serves special tea and coffee to customers. They have distinct recipes for each. The steps to make a cup of tea are: boiling some water, steep special tea bag in the water, pour tea in a cup, add a bit of lemon juice and slices of ginger. On the other hand, the steps to make a cup of coffee include boiling some water, grinding some coffee beans, brew the ground coffee, pour coffee in a cup, add milk and sugar. If we were to design a program to emulate the tea and coffee making procedures, what would be an appropriate pattern to use? Draw a UML diagram of your design.

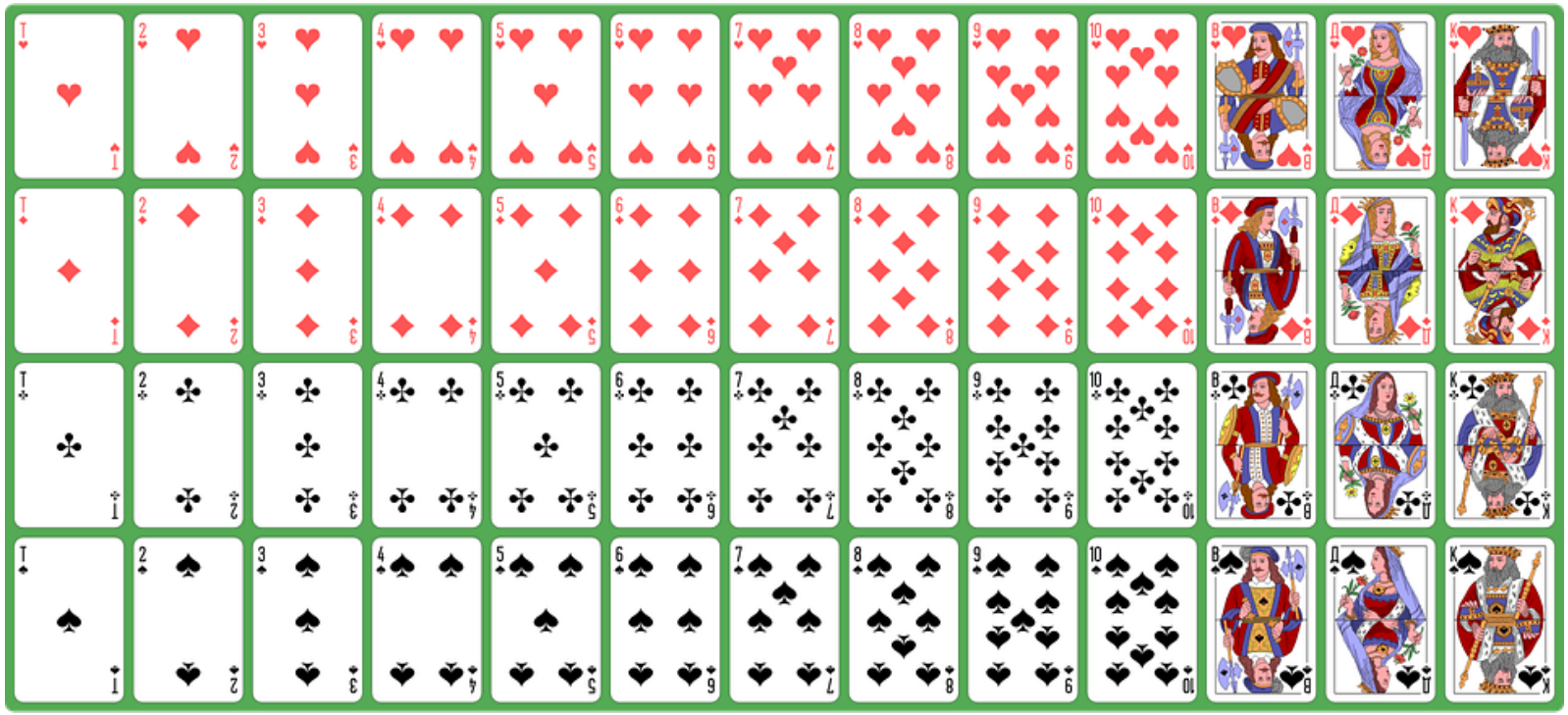
Exercise II



- Template method: prepareRecipe (it is final – we don't want it to be overridden).
- prepareRecipe:
  1. boilWater
  2. brew
  3. pourInCup
  4. addCondiments
- boiling water and pouring in cup are common methods – no need to implement in subclasses.
- brewing and adding condiments are beverage specific – implement in subclasses.

Singleton Pattern

Scenario



We have a deck of playing cards. There are four suits, each with 13 unique ids. We want to develop a program where we will take a deck, pick the top card and display the card picked. We only have one deck of cards, i.e. during the lifetime of the program we want just a single instance of the Deck class that may be shared between multiple players!

Live Demonstration

Singleton have three major parts:

- a static class variable holding the instance.
- a private constructor.
- a public class variable accessor.

Code available on the DLE page.
Can you draw a general UML for a singleton class?

Singleton pattern

Type Creational

Pattern name Singleton

Problem When we want exactly one instance of a class and provide a global point of access.

Solution Define private constructor and populate the instance with an accessor method for the instance.

Consequences Advantages and disadvantages of using the pattern:

- Controlled access to sole instance.
- Difficult to test programs with global state.
- Breaks the single responsibility principle. It has two responsibilities: create a unique instance and perform the duties.
- Difficult to use in multi-threaded application!

Known uses Logging – a single access point to a log file.

Any questions?

Javadoc

Documentation and comments

Why is it necessary?

- Internal users.
  - What does the code do?
  - How does it do it?
- External users.
  - How do we use the code?

Rules.

- What does each method do?
- What are the input and output parameters?
- Are there enough comments to understand the procedures?

- https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#exampleresult
- https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html
- https://google.github.io/styleguide/javaguide.html

Warning: Please do not do this!

"Now! ... *That* should clear up a few things around here!"

https://medium.com/@andrewgoldis/how-to-document-source-code-responsibly-2b2f303aa525

Javadoc

- Javadoc is a document generator for generating API documentation in HTML format from Java source code.
- NetBeans can help with auto-generated documentation.

Example

```
/**
 * Calculate the areas of a rectangle.
 * <p>
 * A very useful method.
 *
 * @param width The width of the rectangle
 * @param height The height of the rectangle
 * @return float The area of the rectangle
 */
public float calcArea(float width, float height){
  // compute and return area
}
```

A comprehensive example with documentation is on DLE.

Example

```
/**
 * Calculate the areas of a rectangle.
 * <p>
 * A very useful method.
 *
 * @param width The width of the rectangle
 * @param height The height of the rectangle
 * @return float The area of the rectangle
 */
public float calcArea(float width, float height){
  // compute and return area
}
```

"/**" marks the beginning of a new java doc comment.

A comprehensive example with documentation is on DLE.

Example

```
/**
 * Calculate the areas of a rectangle.
 * <p>
 * A very useful method.
 *
 *     Stylistic choice – put them in!
 * @param width The width of the rectangle
 * @param height The height of the rectangle
 * @return float The area of the rectangle
 */
public float calcArea(float width, float height){
  // compute and return area
}
```

A comprehensive example with documentation is on DLE.

Example

Description of the method. Can use HTML tags.

```
/**
 * Calculate the areas of a rectangle.
 * <p>
 * A very useful method.
 *
 * @param width The width of the rectangle
 * @param height The height of the rectangle
 * @return float The area of the rectangle
 */
public float calcArea(float width, float height){
  // compute and return area
}
```

A comprehensive example with documentation is on DLE.

Example

```
/**
 * Calculate the areas of a rectangle.
 * <p>
 * A very useful method.
 *
 * @param width The width of the rectangle
 * @param height The height of the rectangle
 * @return float The area of the rectangle
 */
public float calcArea(float width, float height){
  // compute and return area
}
```

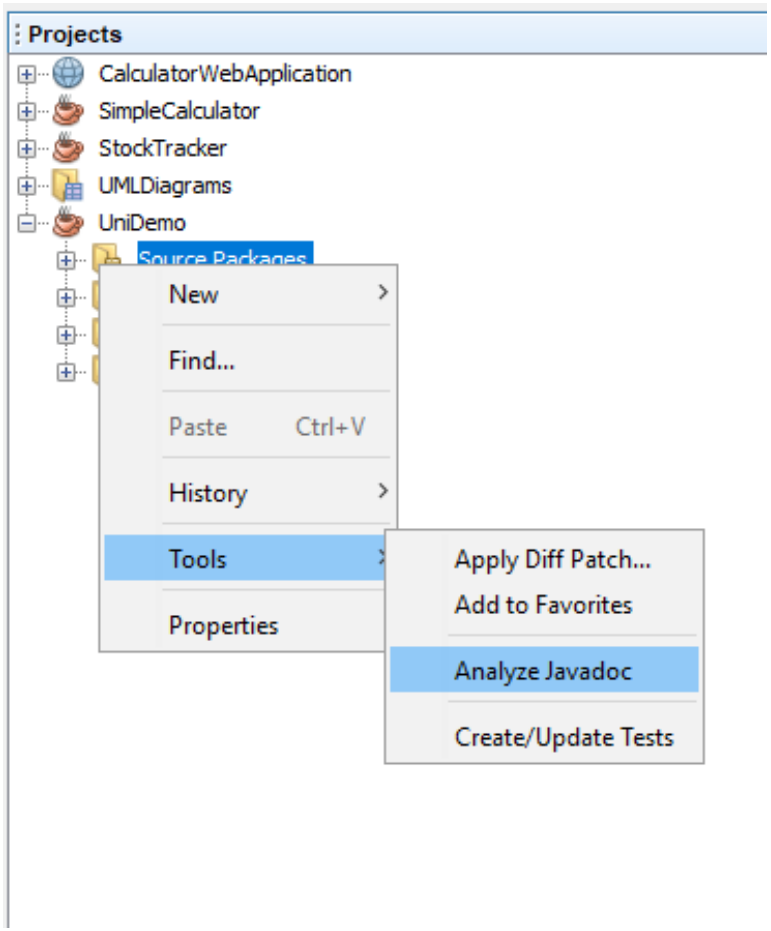
Block tags: tells the user about inputs (@param) and output (@return).

A comprehensive example with documentation is on DLE.

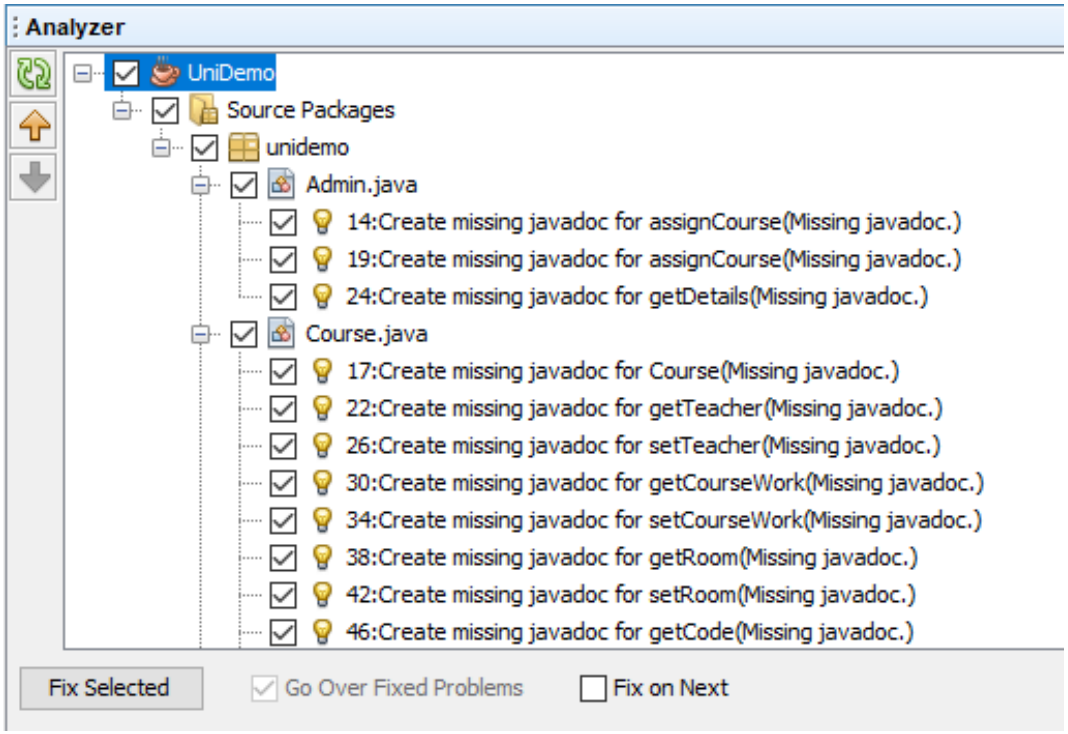
Common tags

@author	Describes the authors.
@version	Provides software version entry. Maximum one per class or interface.
@param	Describes method input parameters.
@return	Describes the return value.
@throws	Describes an exception that may be thrown from the method.

Using NetBeans



Using NetBeans



Press *Fix Selected*

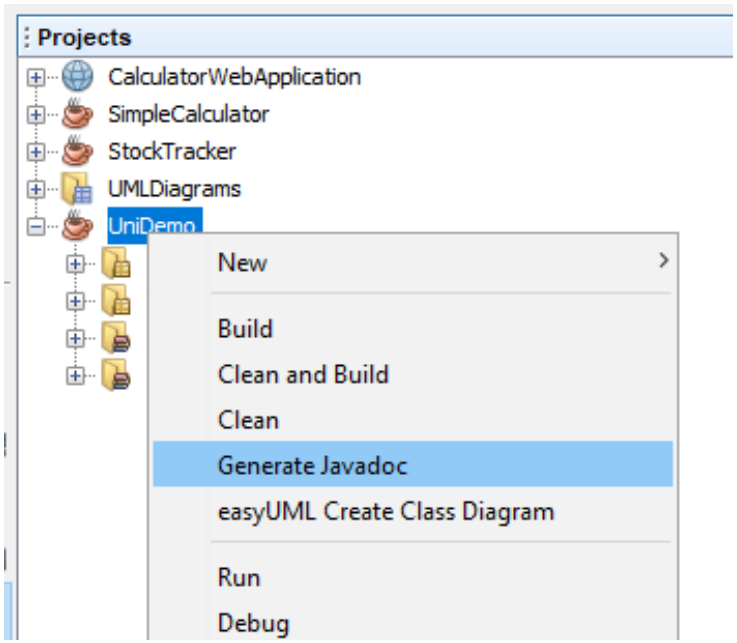
Using NetBeans

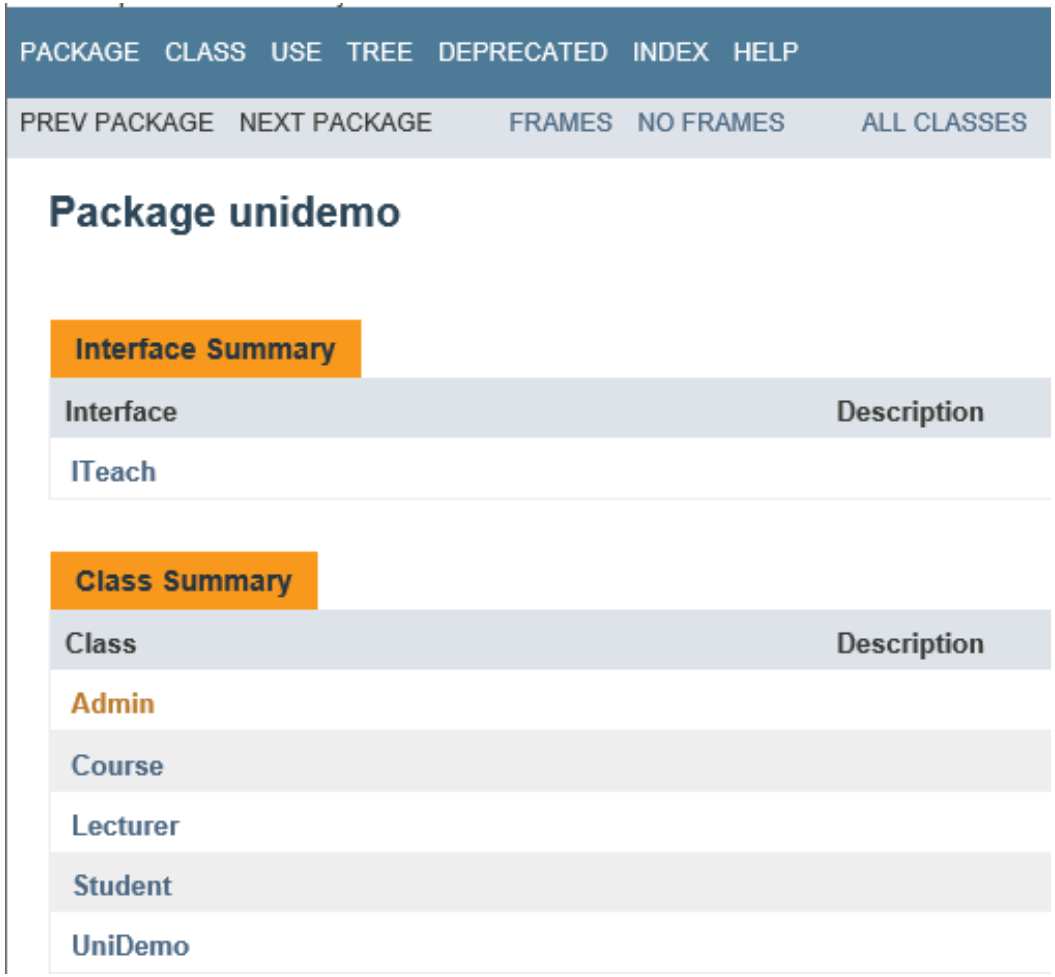
```
public static void assignCourse(Lecturer teacher, Course course) {
    teacher.setCourse(course);
    course.setTeacher(teacher);
}
```

```
/**
 *
 * @param teacher
 * @param course
 */
public static void assignCourse(Lecturer teacher, Course course) {
    teacher.setCourse(course);
    course.setTeacher(teacher);
}
```

Tags for inputs and output are generated – you have to populate the details.

Using NetBeans





Serialisation

saving and loading data

Serialisation

- What happens to the object's state when a program terminates?

Serialisation

- What happens to the object's state when a program terminates?
  $\Rightarrow$ The state is lost!

Serialisation

- What happens to the object's state when a program terminates?
  $\Rightarrow$ The state is lost!
- We may want *persistence*: a state outlives the process that created it!

Serialisation

- What happens to the object's state when a program terminates?
  $\Rightarrow$ The state is lost!
- We may want *persistence*: a state outlives the process that created it!
- In Java, we achieve this using object serialisation: convert an object into a series of bytes that can be written to a stream, and read from a stream to create a live object in the state it was written.

Serialisation

- What happens to the object's state when a program terminates?
  $\Rightarrow$ The state is lost!
- We may want *persistence*: a state outlives the process that created it!
- In Java, we achieve this using object serialisation: convert an object into a series of bytes that can be written to a stream, and read from a stream to create a live object in the state it was written.

To make an object serialisable, its class must implement the interface **Serializable** – it has no methods.

`https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html`

Required Java.io classes

FileInputStream Read bytes from a file.

FileOutputStream Write bytes to a file.

ObjectOutputStream Knows how to convert – flatten – an object to a byte stream.

ObjectInputStream Knows how to revert – unflatten – a byte stream to an object.

Java byte streams are used to perform input and output of 8-bit bytes.

Example implementation and demonstration

```
import java.io.*
public class Card implements Serializable{
  private String id;
  private String suit;
  Card(String s, String i){
    suit = s;
    id = i;
  }
  public void setSuit(String s){
        suit = s;
  }
  public String getSuit(){
        return suit;
  }
  public void setID(String i){
        id = i;
  }
  public String getID(){
        return id;
  }
  public void print(){
        System.out.println("Card: "+ suit + " " + id);
  }
}
```

Implement Serializable interface from java.io.

Code on DLE page.

Remember

- Only an object's state is saved: when regenerating it you will need to access the class file for definition.
  - This means that different version may hamper loading data back! **InvalidClassException** will be raised!
- If the state of the object A has an instance of object B in its state, the class defining object B must be serialisable!
- We will not use databases in this course. So, use serialisation to save object states, if you are asked to do so in the coursework.

Any questions?

Design Exercise

Design a chess playing game!

Summary

1. Design Patterns II
    1. Strategy
    2. Template Method
    3. Singleton
2. Javadoc
3. Serialisation

Next in Line

- More design patterns.
- GUI.