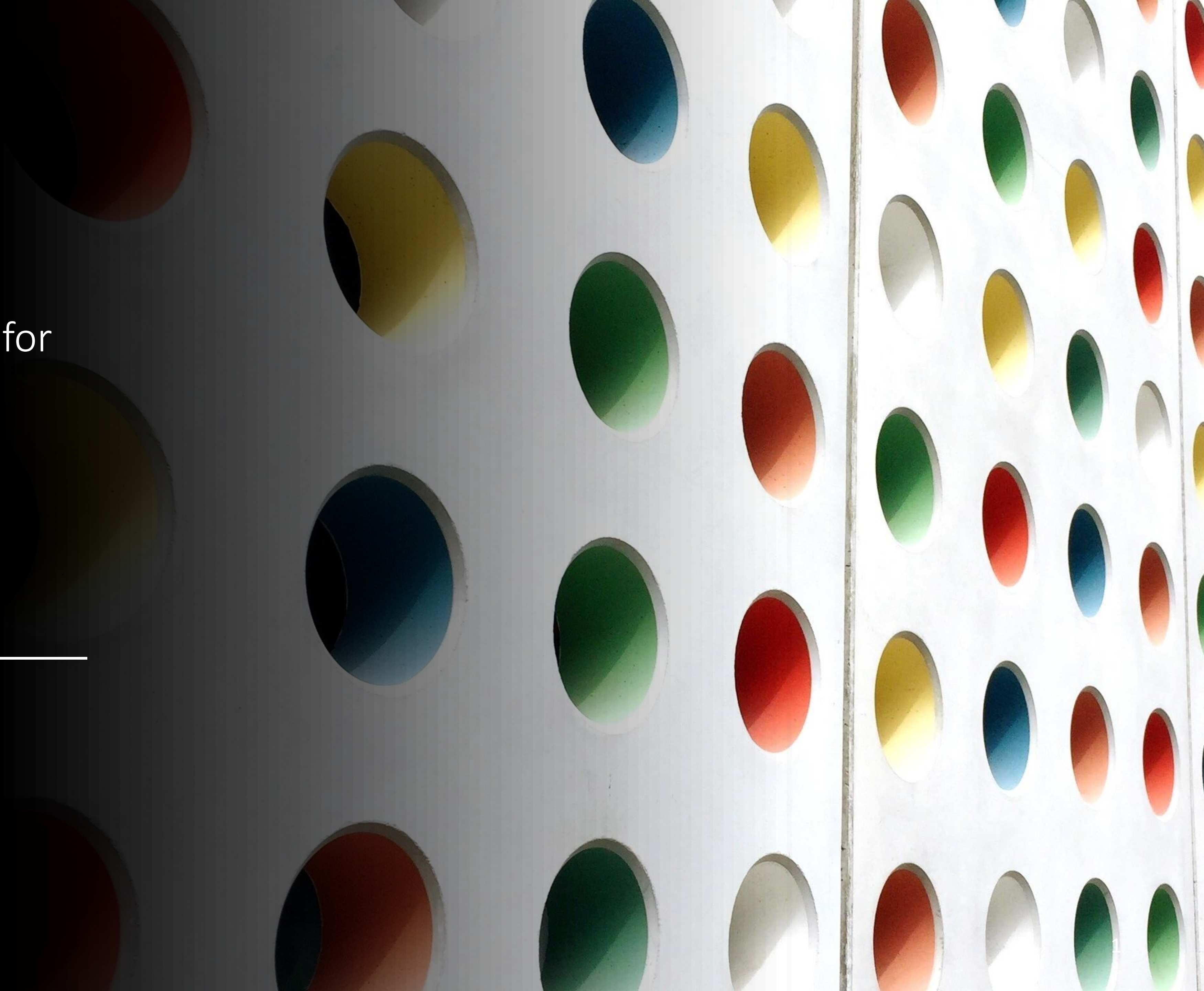
COMP8710 Advanced Java for Programmers

Lecture 7
Design Patterns

Yang He



Topics

- Overview
- Interpreter pattern
- Visitor pattern
- Decorator pattern
- Factory pattern
- Singleton pattern

Reusable software

- Goal of software engineering, especially OO software engineering
  - Factor objects into classes
  - Define class interfaces and inheritance hierarchies
  - Identify relationships
- Don’t solve every problem from first principles
  - Reuse solutions
  - Particularly patterns of classes and communicating objects

Design patterns

- Design patterns describe a commonly recurring problem and the core of a solution to that problem
- They make it easier to reuse successful designs and architectures
- They help enhance code
  - Flexibility
  - Maintainability
  - Scalability

How do you use design patterns?

- You have all used off-the-shelf libraries and frameworks
  - Take the code, configure it and use it
- Design patterns do not go straight into your code, they first go into your brain
  - First you need to learn what they are and how they work
  - Then you can identify problems that fit with those that a particular design pattern aims to solve
  - Then you can apply them in your design

Design Patterns (sceptical view)

- Like other paradigms, OO has some weaknesses w.r.t. software development, e.g.
  - Object creation is exposed to client
  - Heavy use of subclasses scatters code with a common functionality
  - Dependence on named interfaces makes it hard to create re-usable code
- Sometimes these weaknesses are addressed with new language features, and the idea of design patterns pre-dated some of those language extensions (for Java)
- Thus, often, design patterns are workarounds to tell you how to circumvent these weaknesses

Benefits of learning design patterns

- It helps you communicate with other developers
  - Using the name of the pattern to communicate your idea at a more abstract level
- It makes you a better designer to build reusable, extensible and maintainable software
- It helps you learn and use new frameworks faster
  - Design patterns are used in various frameworks and libraries

Classification of design patterns

- Creational patterns
  - Provide object creation mechanisms to increase flexibility and reuse of existing code
  - E.g. builder, factory, singleton
- Structural patterns
  - Explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient
  - E.g. decorator, façade, adapter
- Behavioural patterns
  - Provide effective communication and the assignment of responsibilities between objects
  - E.g. strategy, command, observer, interpreter, visitor

Essential elements of design patterns

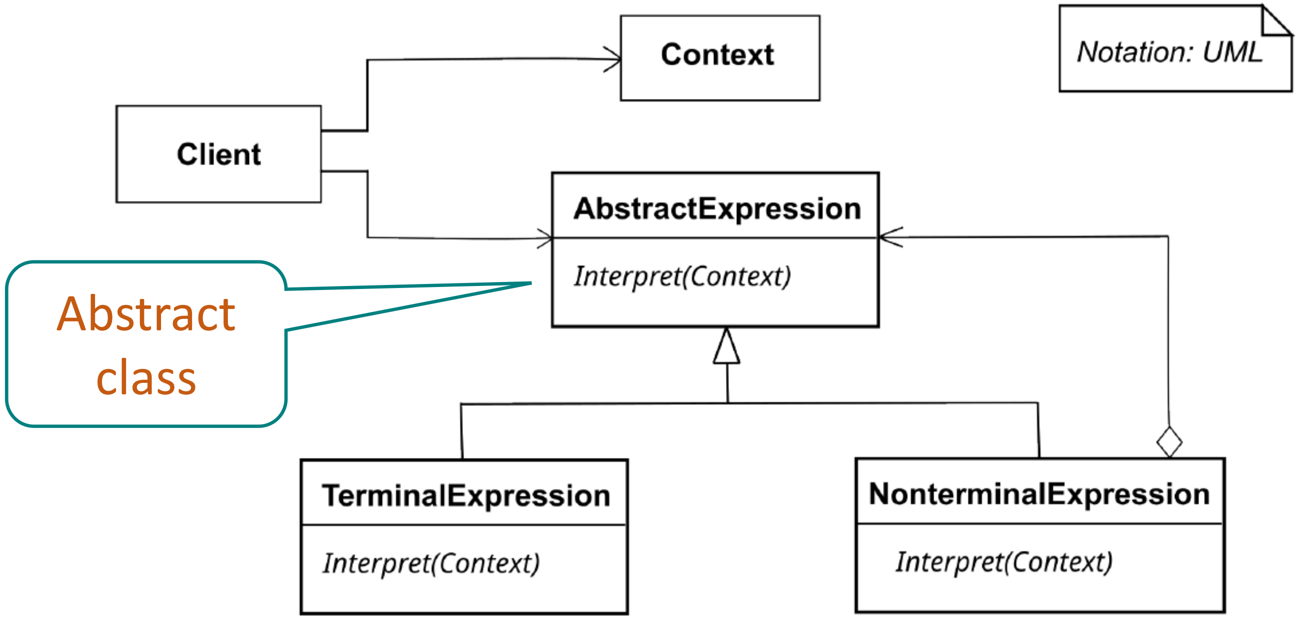
- Name
  - Increases vocabulary and allows design at a higher level, discussion, etc.
- Problem
  - When to apply the pattern, i.e. conditions that must be met, how to represent the algorithm as objects, etc.
- Solution
  - The elements that make up the design, their relationships, collaborations, responsibilities
  - A template that can be applied in many different situations
- Consequences
  - Results and trade-offs, impact on flexibility, extensibility, and portability

An example problem

- As part of a compiler project, we want to be able to handle simple arithmetical expressions like $3 * (4 + 5)$
- At the very least, we want to be able to:
  - Evaluate them, and
  - Pretty-print them
- In the future, we may wish to:
  - Apply new operations , e.g. cost metrics, common sub-expression elimination, etc.
  - Include new arithmetic operators
- We require:
  - A data representation, and
  - A way of interacting with it

Interpreter pattern (1)

- Intent
  - Define a representation for a language grammar, i.e. an abstract syntax tree (AST), and an interpreter for sentences in the language
- Pattern



Interpreter pattern (2)

- Participants
  - Classes for each node in the AST, derived from abstract class AbstractExpression
  - Concrete classes for each TerminalExpression, which implement interpret() for their terminal symbol
  - Classes for each NonterminalExpression in the AST
    - Given a rule $R ::= R_1R_2 \ldots R_n$ in the grammar, a NonterminalExpression will have an AbstractExpression attribute for each of the R_i
    - Typically interpret() calls itself recursively on each of these

Interpreter pattern (3)

- Context
  - Any information used by all the interpret() operations
- Client
  - Parses a sentence of the language and builds the AST from the TerminalExpression and NonterminalExpression classes
  - It then calls the interpret() method on the root of the AST

Interpreter pattern (4)

- An implementation

```
public abstract class SimpleNode {
   public abstract int eval();
}

public class ASTInteger extends SimpleNode {
   // value
   public int eval() { return getValue(); }
}

public class ASTAdd extends SimpleNode {
   // left and right
   public int eval() { return left().eval() + right().eval(); }
}

ASTStart ast = parser.start();   //parse the expression
int result = ast.eval();
```

Interpreter pattern (5)

- Applicability
  - Language to interpret; abstract syntax tree
  - Grammar is simple (otherwise use parser generator)
- Collaboration
  - Client builds the AST, initialises the context and invokes interpret on the root
    - Interpret uses context to access state of interpreter
- Consequences
  - Easy to change and extend the grammar
  - Easy to implement the grammar
- But
  - Large grammars hard to maintain – multiple classes
  - Hard to add new computations over the expression — need to change every class

Visitor Pattern (1)

- E.g.

```
public interface SimpleVisitor {
   int visit(ASTAdd node);
   int visit(ASTInteger node);
}

public class SimpleEvalVisitor implements SimpleVisitor {
   public int visit(ASTAdd node) {
      return node.left().accept(this) +
             node.right().accept(this);
   }
   public int visit(ASTInteger node) {
      return node.getValue();
   }
}
```

Interface methods are public and abstract

All "eval" code are now in one file

Visitor Pattern (2)

```
public abstract class SimpleNode {
   public int accept(SimpleVisitor visitor) {
      return visitor.visit(this);
   }
}


ASTStart ast = parser.start();
int result = ast.accept(new SimpleEvalVisitor() );
```

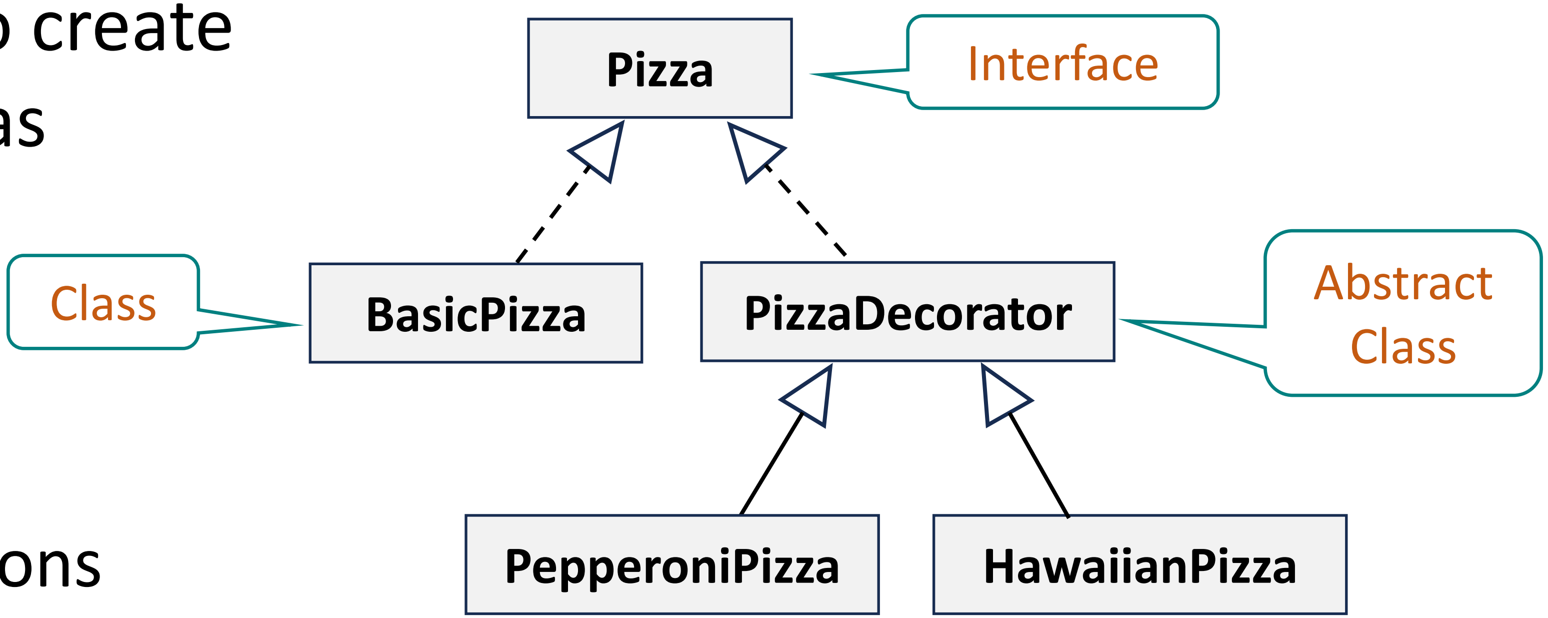
In more complex scenarios, where additional parameters are needed for visit, subclasses will often have to override the accept implementation

Visitor Pattern (3)

- Easy to add new computations over the expression
- Visitor gathers related computations and separates unrelated ones
- Can visit across class hierarchies
- Can accumulate state
- But
  - Hard to add new elements — need to change every visitor class
  - Breaks encapsulation — visitor reads internal state of objects it visits

Decorator pattern (1)

- Decorators are used to enhance or modify the behaviour of objects at runtime
- E.g. using decorator to create different kinds of pizzas
  - Basic pizza
  - Pepperoni pizza
  - Hawaiian pizza
  - Any other combinations



Decorator pattern (1)

- E.g. implement different kinds of pizzas

```
public interface Pizza {
   void decorate();
}

public class BasicPizza implements Pizza {

    @Override
    public void decorate() {
        System.out.println("Pizza base with tomato and cheese." );
    }
}
```

Decorator pattern (2)

```
public abstract class PizzaDecorator implements Pizza {
    private Pizza pizza;

    public PizzaDecorator(Pizza pizza) {
        this.pizza = pizza;
    }

    @Override
    public void decorate() {
        pizza.decorate();
        System.out.println("   Add toppings: " + getToppings());
    }

    public abstract Set<String> getToppings();
}
```

Decorator pattern (3)

```
public class PepperoniPizza extends PizzaDecorator {

    public PepperoniPizza(Pizza pizza) { super(pizza); }

    @Override
    public Set<String> getToppings() {
        return Set.of("Pepperoni");
    }
}

public class HawaiianPizza extends PizzaDecorator {

    public HawaiianPizza(Pizza pizza) { super(pizza); }

    @Override
    public Set<String> getToppings() {
        return Set.of("Pineapple", "Ham");
    }
}
```

Decorator pattern (4)

- In the main method:

```
System.out.println("\nBasic pizza:");
var basic = new BasicPizza();
basic.decorate();

System.out.println("\nPepperoni pizza:");
var pepperoni = new PepperoniPizza(basic);
pepperoni.decorate();

System.out.println("\nCombo pizza:");
var combo = new HawaiianPizza(pepperoni);
combo.decorate();
```

Output:

```
Basic pizza:
Pizza base with tomato and cheese.

Pepperoni pizza:
Pizza base with tomato and cheese.
   Add toppings: [Pepperoni]

Combo pizza:
Pizza base with tomato and cheese.
   Add toppings: [Pepperoni]
   Add toppings: [Pineapple, Ham]
```

Decorator pattern (5)

- Decorator pattern let us add extra features to objects *without* changing their core structure in runtime
  - More flexible
  - Easy to maintain and extend to more choices
- Decorator pattern is used a lot in Java IO classes
  - E.g. FileReader, BufferedReader etc.
- Java new features, e.g. lambda, make it a lot easier to implement some design patterns

Factory Pattern (1)

- Generally, it a good OO design to hide the implementation from a software interface
- Constructors break that to some extent, i.e. when calling a constructor of a class, it
    - Creates an object of the class (but not any of its subclasses)
    - The object created has the class fields as specified
- We may want to hide this from users. But how?

Factory Pattern (2)

- One way is to delegate the creation of objects to a factory class
- E.g.

```
public class PetFactory {
    public static Pet createPet(PetType type, String name) {
        switch (type) {
            case DOG:
                return new Dog(name);
            case CAT:
                return new Cat(name);
        }
        throw new IllegalArgumentException("Unknown pet type: ");
    }
}
```

```
var dog = PetFactory.createPet(PetType.DOG, "Spot");

var cat = PetFactory.createPet(PetType.CAT, "Molly");
```

Singleton pattern (1)

- The singleton pattern restricts the instantiation of a class, ensuring only one instance of the class exists
- E.g. only one single connection to a database

```
public class DatabaseConnection {
    private static DatabaseConnection db;
    private DatabaseConnection() {
        System.out.println("Create connection to database.");
    }
    public static DatabaseConnection getDBConnection() {
        if (db == null) {
            db = new DatabaseConnection();
        }
        return db;
    }
}
```

Singleton pattern (2)

- E.g.

```
var db1 = DatabaseConnection.getDBConnection();
var db2 = DatabaseConnection.getDBConnection();
System.out.println(db1 == db2);
```

Output:

```
Create connection to database.
true
```