COMP2511

Week 9 Tutorial

Last Tutorial

- Generics
- Iterator Pattern
- Singleton Pattern
- Decorator Pattern

This Tutorial

- Visitor Pattern
- Builder Pattern
- Revision (if time permits)

Notices

- If you are completing Assignment 3 and would like to work solo or with a different partner, please notify me via. Teams or email ASAP.
- The lab computers during next week's lab will be running a sample exam environment.
  - The main goal of this is to get you comfortable with the **setup** of the exam.
  - There will be practice questions for you to complete, and the sample exam will also be released on GitLab at a later date.
  - You can still have Lab 8 manually marked next week if required.

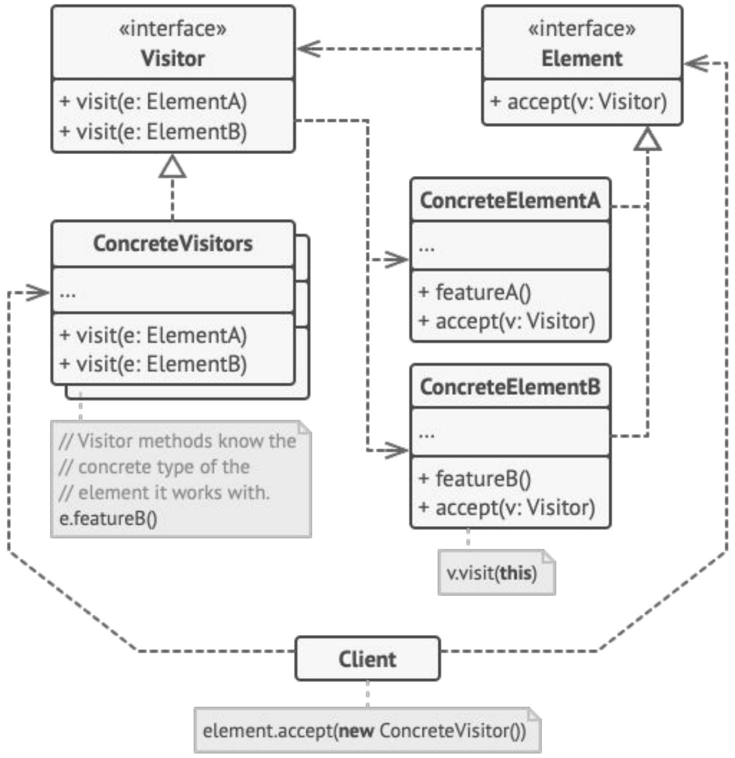
Visitor Pattern

- The **Visitor** pattern is a behavioural design pattern that allows for new operations to be defined for classes without changing the classes (to a significant degree).
- It achieves this by separating the new behaviour/algorithm into a separate class called a **visitor**, which contains all of the required implementations (and all possible variations if the implementation differs across different classes).

Visitor Pattern

- From the previous slide, the main goal of the visitor pattern is to extend upon the functionality upon a class. Why not just use inheritance or just change each of the classes?
    - First point: Some classes are **final** and hence cannot be extended. Also, the functionality that you are trying to add on may be fairly disconnected from the rest of what the class is trying to do, and hence may not necessarily make the most sense as a method belonging to that class (for example, turning an object into a format like JSON or XML).
    - Second point: The open-closed principle; making extensions to the system should modify as little of the existing code as possible.

Visitor Pattern Implementation

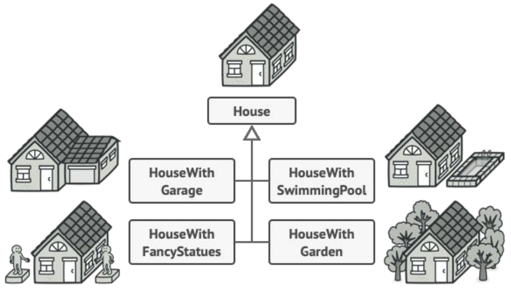


Live Example

src/visitor AND src/computer
(details in README.md)

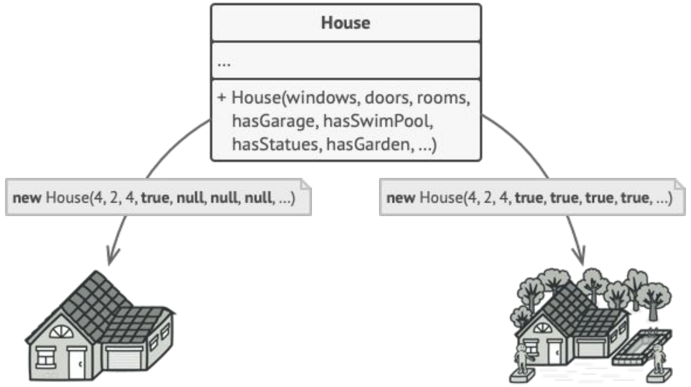
Builder Pattern

- Imagine you were creating classes to represent a House - the house could have a variable number of doors and windows, could potentially have a pool, garden, multiple storeys etc.
- Would it be a feasible solution to create separate classes for each variation of house? For example, making House, HouseWithPool, HouseWithPoolAndGarden classes?
    - No! You should be convinced that this would require way too many classes.



Builder Pattern

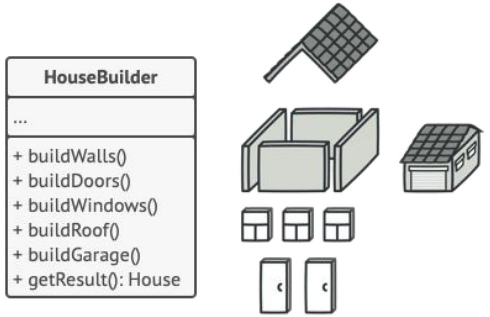
- How about this potential solution?


House
...
+ House(windows, doors, rooms, hasGarage, hasSwimPool, hasStatues, hasGarden, ...)
new House(4, 2, 4, true, null, null, null, ...)
new House(4, 2, 4, true, true, true, true, ...)

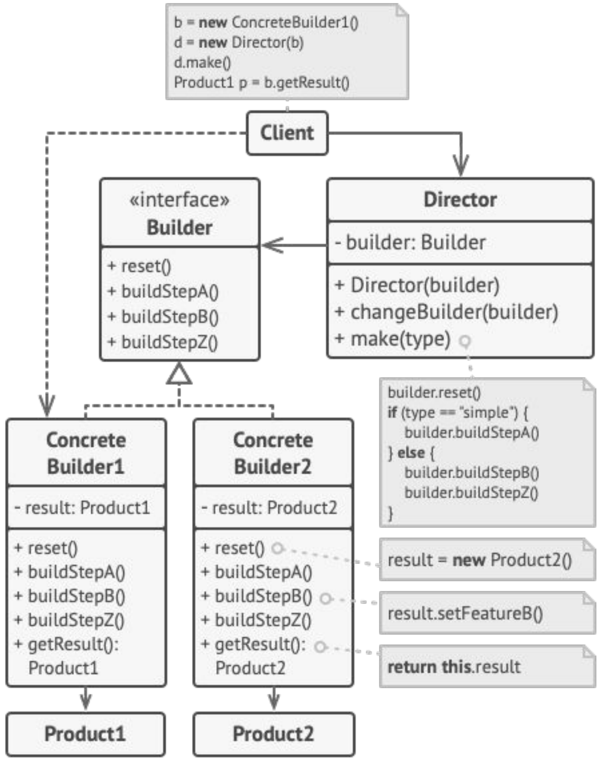
- Rather than exploding the number of classes, this explodes the number of fields and parameters. This approach still isn't desirable.

Builder Pattern

- The **Builder** pattern is a creational design pattern that separates the construction of a complex object (e.g. objects that have many different components that can be mix and matched) into individual steps.
- It achieves this by defining a Builder class, containing methods that each add some sort of component to the complex structure you are building; once all components have been added, it returns the finished product.



Builder Pattern Implementation



```
b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
```

```
builder.reset()
if (type == "simple") {
    builder.buildStepA()
} else {
    builder.buildStepB()
    builder.buildStepZ()
}
```

```
result = new Product2()
```

```
result.setFeatureB()
```

```
return this.result
```

Live Example

src/trains (details in README.md)

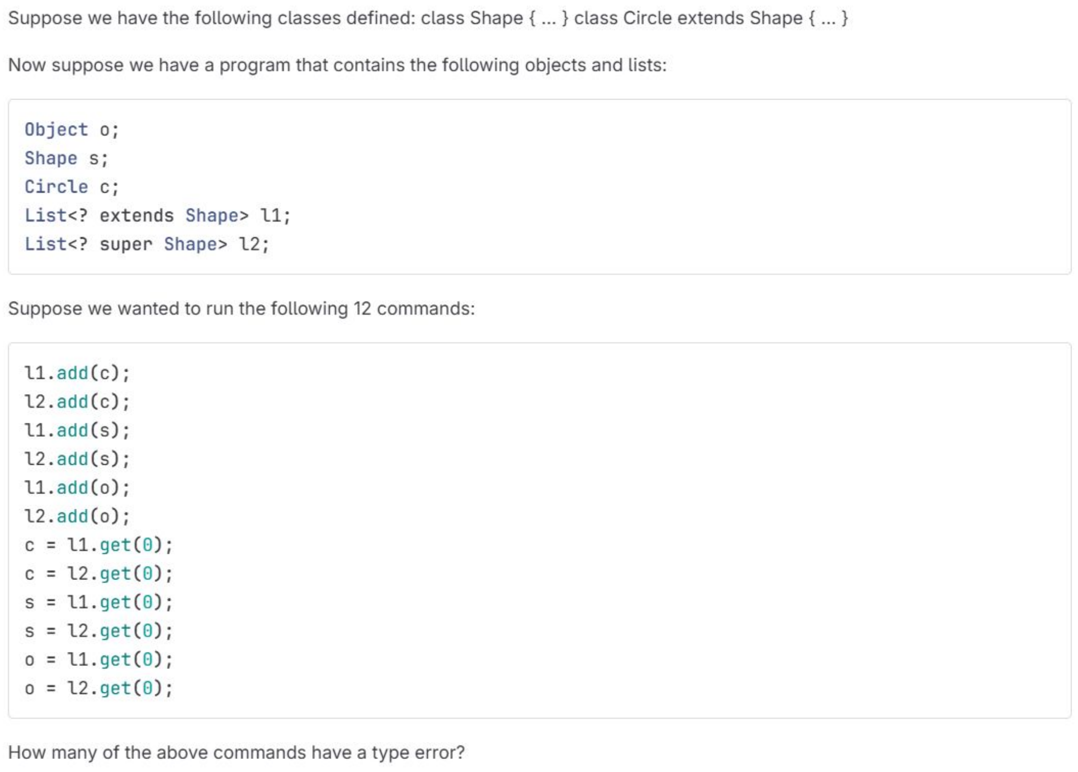
More Patterns

- That’s all the tutorial content for the term!
- Some patterns have not been covered in tutorials. The **theory** of these patterns is assessable in the exam, but it is *very* unlikely that they will ask you to code these up:
    - Template Pattern
    - Command Pattern
    - Facade Pattern
    - Adapter Pattern

Revision Question

- For each of the following scenarios, what pattern(s) are most applicable as potential solutions?
    - Sorting collections of records in different orders
        - **Answer: Strategy**
    - Listing the contents of a file system.
        - **Answer: Composite, potentially Visitor**
    - Traversing through a linked list without knowing the underlying representation.
        - **Answer: Iterator**
    - Updating a UI component when the state of a program changes.
        - **Answer: Observer**
    - A frozen yogurt shop model which alters the cost and weight of a bowl of frozen yogurt based on the toppings that customers choose to add before checkout.
        - **Answer(s): Decorator, Builder**

Revision Question



Suppose we have the following classes defined: class Shape { ... } class Circle extends Shape { ... }

Now suppose we have a program that contains the following objects and lists:

```java
Object o;
Shape s;
Circle c;
List<? extends Shape> l1;
List<? super Shape> l2;
```

Suppose we wanted to run the following 12 commands:

```java
l1.add(c);
l2.add(c);
l1.add(s);
l2.add(s);
l1.add(o);
l2.add(o);
c = l1.get(0);
c = l2.get(0);
s = l1.get(0);
s = l2.get(0);
o = l1.get(0);
o = l2.get(0);
```

How many of the above commands have a type error?

Revision Question

- **Answer: 7 type errors**
  - A List<? extends Shape> can theoretically store *any* subclass of Shape. The problem is that this is not known at compile time, so you are not guaranteed to be able to add something like Circle, since it could potentially be a List<BetterCircle> where BetterCircle is a subclass of Circle.
    - Lines 1, 3, 5 are invalid (we cannot guarantee what we are adding is valid).
    - Line 7 is invalid (it may store a subclass of Shape that isn't Circle)
  - A List<? super Object> can theoretically store *any* superclass of Shape. But what if Shape had some other non-Object superclass? Then we wouldn't be able to store an Object.
    - Line 6 is invalid (we cannot guarantee what we are adding is valid).
    - Lines 8, 10 are invalid (we cannot guarantee that we can treat some superclass as a Shape or Circle, e.g. if it was a List<Object>).

Revision Question

- Are all code smells emblematic of a design problem?
  - **Answer:** No! Code smells are *indicators* of design problems, not guarantees. If a code smell is present, it’s worth investigating and assessing whether the code smell is a ‘necessary evil’ - i.e. it’s the best available solution to your problem.