Class decoupling

• Consider the following interface

```java
public interface CreditCardProcessor {
    void process(PizzaOrder order);
}
```

• Usage

```java
public class BillingService {
    private CreditCardProcessor p = new VisaProcessor();
    public boolean chargeOrder(PizzaOrder o) {
        p.process(o);
        // other logic...
    }
}
```
Class decoupling (cont.)

- Although we use interfaces, the implementation is highly coupled with VisaProcessor.
- `new` is a static method, and static methods always mean coupling.
- It is much harder to test using real dependencies.
  - VisaProcessor can be a very slow class.
  - It can have unwanted side effects (do we want to be billed for every test?)
  - How can we replace it with mocks?
- It is not easily configurable.

Solution

- Pass collaborators (dependencies) to the object.
- This is called Dependency Injection.
Dependency Injection

• The current supplier can behave differently based on the client configuration
  • Can easily replace VisaProcessor with PaypalProcessor
  • Can inject MockProcessor for testing

• Dependency objects can be reused elsewhere

• The default behavior can still exist if we desire it
  • Implementation of the default behavior differs depending on the type of injection used
Using inheritance?

• An abstract class implements the main algorithmic skeleton
• Some logic is deferred to template methods
• **Default** implementation can be enabled by making the method **non-abstract**

```java
public abstract class BillingService {
    protected abstract void chargeOrderAux(PizzaOrder o);
    public void chargeOrder(PizzaOrder o) {
        chargeOrderaux(o);
        // other logic...
    }
}

public class CreditCardBillingService extends BillingService {
    private final CreditCardProcessor p = new VisaProcessor();
    @Override protected void chargeOrderAux(PizzaOrder o) {
        p.process(o);
    }
}
```
Inheritance – problems

• Testing the parent class with mocks is pretty **verbose**
  • Have to override the template method with a new **class**
• We may be forced to create many classes, causing **visual clutter**, and **namespace pollution**
• What do we do when we have two or more, **orthogonal** dependencies? (Combinatorial explosion)
• It's hard to inject the **same** dependencies in **different** places
  • What if we want to use **VisaProcessor** in **all** of our classes?
• Generally speaking: inheritance doesn't **compose**
• This **can** be **good enough** in some **special** case
  • For example, when you have a **small finite, pre-determined** number of possible implementation
  • But it can be pretty hard to know that **in advance**!
Setters?

• We can use setters to set the value after object creation

```java
public class BillingService {
    private CreditCardProcessor processor;
    public void setCreditCardProcessor(CreditCardProcessor p) {
        this.processor = p;
    }
}
```

• We can implement default behavior by settings the fields in the default constructor
Setters – Pros & Cons

• Pros
  • Easy to create **multiple, orthogonal** defaults
  • Easy to match parameter and argument

• Cons
  • There’s a window in which an object has been **created** and is **not yet ready** for use
    • What happens if we **forget** to call a setter?
  • Members cannot be final
    • loss of **write-safety**; what happens if we call set more than once?
    • class has to be **mutable**, which is something we should **always** strive to avoid
  • **Client** code is more **verbose** (more lines of code)
Constructors

• We can pass the object at the construction of the object

```java
public class BillingService {
    private final CreditCardProcessor processor;
    public BillingService(CreditCardProcessor p) {
        processor = p;
    }
}
```

• We can implement default behavior using a default constructor
• Pros and cons are the **opposite** of the setters solution
• Pros:
  • Dependencies are injected upon object **instantiation**, so object is always in a correct state
  • Class can be **immutable**
  • Less lines of code
• Cons:
  • Multiple defaults lead to combinatorial explosion
  • Hard to match primitive parameters with arguments
Best of both worlds: Builder Objects

```java
Foo foo = new FooBuilder()
    .setX(x)
    .setY(y)
    .build(); // passes x and y to Foo's constructor
```

- **Builder** objects combine setters with constructors
- Usually use fluent APIs, i.e., setters return this at the end of the setter method
- Easy to provide multiple defaults
- Easy to match parameters with arguments
- Can check invariants either in the `build()` function (missing fields) or in the constructor (invalid fields)
- Class be immutable (builder can be but usually isn't)
- Frameworks help us avoid **boiler plate** configuration of builders
  - E.g., [FreeBuilder](#) (also provides free `toString`, `hashCode` and `equals`!)
Builders: implementation example

class Person {
    private final String name;
    private final String address;
    private final boolean isStudent;
    // ctor code omitted
}

// Oops, mixed up name and address
// What does true mean again?
new Person("Haifa, Technion", "John smith", true);

class PersonBuilder() {
    private String name;
    public PersonBuilder setName(String name) {
        this.name = name;
        return this;
    }
    // other setters and fields look the same
    public Person build() {
        return new Person(name, address, isStudent);
    }
}

// isStudent can default to false (why couldn't we do it inline before?)
new PersonBuilder().setName("John Smith").setAddress("Technion").build();
Factory Method pattern

• What if we actually want to create more than one object in our class?
  • For example, we need to create a new processor to handle every request

• We could use a `createProcessor()` method that will be in charge of creating & configuring our Processor
  • This method is called a factory method

• This solution is completely analog to the template method pattern, with its pros and cons
  • Most importantly, doesn't compose
• We saw a similar solution when we talked about OOP in test
• This is more declarative and less error-prone than the first inheritance example we saw (why?)
Abstract Factory

• We would like to **separate** between an object’s **creation** and its **configuration**
  • The point (in the program) of configuration is often different than that of instantiation
• We will **favour composition over inheritance**
• We will **delegate** the job of object creation to **factory objects**
Abstract Factory Pattern

Processor

process(PizzaOrder, CreditCard)

implements

VisaProcessor

process(PizzaOrder, CreditCard)

create(): VisaProcessor

ProcessorFactory

create(): Processor

implements
Abstract Factory – Client

• Interface definition:

```java
// or Supplier<Processor> in the standard library
interface ProcessorFactory {
    Processor create();
}
```

• Client:

```java
class BillingService {
    private final ProcessorFactory pf;
    public BillingService(ProcessorFactory pf) {
        this.pf = pf;
    }
    public void chargeOrder(PizzaOrder o) {
        Processor p = pf.create();
        p.process(o);
        // other logic...
    }
}
```
Abstract Factory – Participants

• To change the dependencies, simply send a new factory:

```java
public class VisaProcessorFactory implements ProcessorFactory {
    @Override public VisaProcessor create() {
        return new VisaProcessor();
    }
}
```

```java
public class MockFactory implements ProcessorFactory {
    @Override public Processor create() {
        Processor $ = mock(Processor.class);
        // ...mock configuration code...
        return $;
    }
}
```

• We can use lambdas if the factory is a functional interface:

```java
ProcessorFactory paypalFactory = () -> new PaypalProcessor();
```
Abstract Factory vs. Factory Method – Names are important!

**Factory Method**
- Creational logic is handled with *methods* and *inheritance*
- Modifications is handled via *overriding* the *factory method*
- Use when you want to *couple* the creational and business logic
- Doesn't compose
- Usually can't use lambdas

**Abstract Factory**
- Creational logic is *delegated* to an *external* object
  - more objects are created/passed around
  - But factories can be reused
- Use when you want to *decouple* the creational and business logic
- Composes
- Can be replaced with lambda
Factories – summary

• The create method can also accept parameters
  • It usually mirrors the constructors signature
  • But it can be simpler or more complex as needed

• Know when to use constructor dependency injection, and when you need a factory
  • Need a single object? Inject it in the constructor
  • Need to create objects? That's a factory

• Remember, the creational logic needs to be somewhere
  • Be careful of creating FactoryFactory!

• These our first design patterns!
When should you inject dependencies? (Deja-vu)

• **You should** inject dependencies when:
  • You don’t want to be tied down to a **single** implementation
  • You want **compositional polymorphism**
    • As opposed to inheritance polymorphism
  • You want to enable mocking during tests

• **You don’t** want to inject:
  • **Dependable dependencies**: standard / external library classes
  • For the sake of injecting
Dependency injection: Cons

Dependency injection makes the code harder to follow

- Flexibility / polymorphism always increases complexity
- Client code now needs to configure its dependencies
  - Even if you provide defaults, the option is still there
- When debugging, we have to find the place of injection, which could be very far from the usage location
  - Reduced code locality
  - Need to keep track of several code modules
- Managing deep or parallel dependencies is a real pain
  - Next week, we will learn how to fix that using Guice