



## Section 1

### Patterns in Spring

Factories - Overview  
Simple Factory  
Factory Method Pattern  
Strategy Pattern  
Command Pattern  
Template Method Pattern

Notes



## Section Objectives

- See how a ***Simple Factory*** works
- Gain an understanding of the ***Factory Method Pattern***

Notes



## Spring Patterns

### Factories - Overview

#### **Factories - Overview**

Simple Factory

Factory Method Pattern

Strategy Pattern

Command Pattern

Template Method Pattern

Notes



## Factories - Definition and Motivation

- Change happens
  - Introducing new concrete types means changing code
    - This can mean a lot of maintenance
  
- A **factory** is a piece of code that instantiates objects
  - The unit of code we usually speak of is the **factory method**
  
- **Factories separate instantiation from use**
  - This allows new subclasses to be introduced with no change to the code that uses the base class
    - Essentially, we're trying to rid client code of the use of **new**
  
- Code to the interface
  - We want code closed for modification

- The term *factory* usually resolves to a what we call a **factory method**, which resides (of course) in a class, referred to as the **factory class**.

Notes



## Factory Patterns

- Technically, i.e., according to the GoF, there are only two factory patterns:
  - **Factory Method Pattern**
    - *"Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses."* (GoF)
  - **Abstract Factory Pattern**
    - *"Provide an interface for creating families of related or dependent objects without specifying their concrete classes."* (GoF)

Notes



## Commonality and Variance

### Simple Factory

Factories - Overview  
**Simple Factory**  
Factory Method Pattern  
Strategy Pattern  
Command Pattern  
Template Method Pattern

Notes



## A Simple Factory - Overview

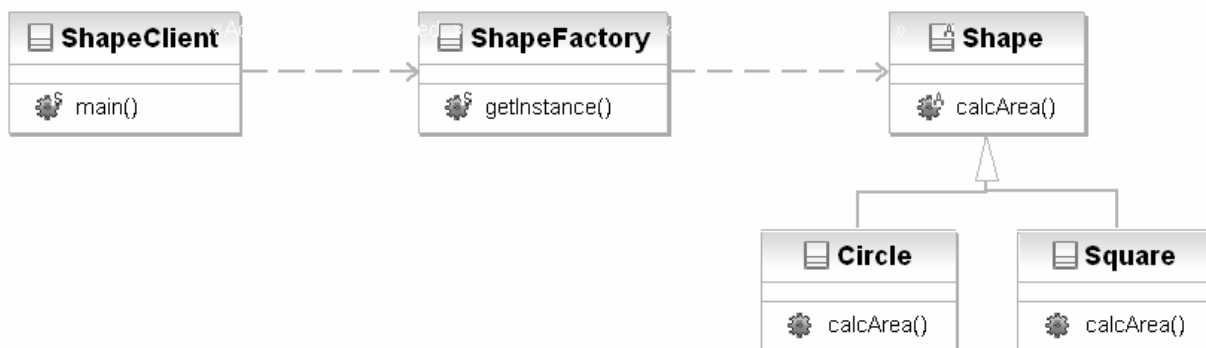
- Goal is to remove the use of **new** from client(s)
  
- Not "technically" a pattern
  
- Easy to learn
  - Helps us to grasp the principles and motivation quickly
  
- A natural bridge to the "real" factory patterns

Notes



## Simple Factory - Example

- We have a Shape abstract base class and a number of concrete subclasses
- We don't want client(s) to know the types of Shapes we have
  - This way, the client is not dependent on concrete types of Shape
  - The types of Shapes can vary without breaking the client



Notes



## Simple Factory - Example - Shape

- We've identified shapes as a point of variance and hid them behind an abstraction – the abstract base class Shape

```
public abstract class Shape
{
    public abstract void calcArea();
}
```

Notes



## Simple Factory - Example - Circle

```
public class Circle
extends Shape
{
    public void calcArea()
    {
        System.out.println("Circle calcArea");
    }
}
```

**Notes**



## Simple Factory - Example - Square

```
public class Square
extends Shape
{
    public void calcArea()
    {
        System.out.println("Square calcArea");
    }
}
```

**Notes**



## Simple Factory - Example - Factory Class

- The ShapeFactory class instantiates the actual objects
  - Why is this better?
  - Isn't there still maintenance when I add a new shape?

```
public class ShapeFactory
{
    // factory method
    public static Shape getInstance(String type)
    {
        if (type.equals("S"))
        {
            return new Square();
        }
        else
        {
            return new Circle();
        }
    }
}
```

Notes



## Simple Factory - Example - Client

- What is this client coupled to?
- What about change?

```
public class ShapeClient
{
    public static void main(String[] args)
    {
        // call the factory method, get a Shape back
        Shape s1 = ShapeFactory.getInstance("S"); // returns a Square
        s1.calcArea();
    }
}
```

- Although the client is working with a Square object, there is no occurrence of "new Square()" anywhere in the code.
  - Hence the client is decoupled from the specific type of Shape it is using.

### Notes



## Simple Factory - Example - Adding a New Shape

- We introduce a new, improved type called Triangle, that many of our customers want
  
- Impact of change:
  - Write a Triangle class
  - Modify ShapeFactory
  
- What about ShapeClient?
  - What if there were hundreds or thousands of clients?

Notes



## Simple Factory - Example - Triangle

- Triangle code – change in one place

```
public class Triangle
extends Shape
{
    public void calcArea()
    {
        System.out.println("Triangle calcArea");
    }
}
```

Notes



## Simple Factory - Example - Minimizing Changes

- ShapeFactory with Triangle – code changes here as well  
**BUT**
  - In how many total places did we change code?
  - How many clients changed?

```
public class ShapeFactory {  
    // factory method  
    public static Shape getInstance(String type) {  
        if (type.equals("S")) {  
            return new Square();  
        }  
        else if (type.equals("T")) {  
            return new Triangle();  
        }  
        else {  
            return new Circle();  
        }  
    }  
}
```

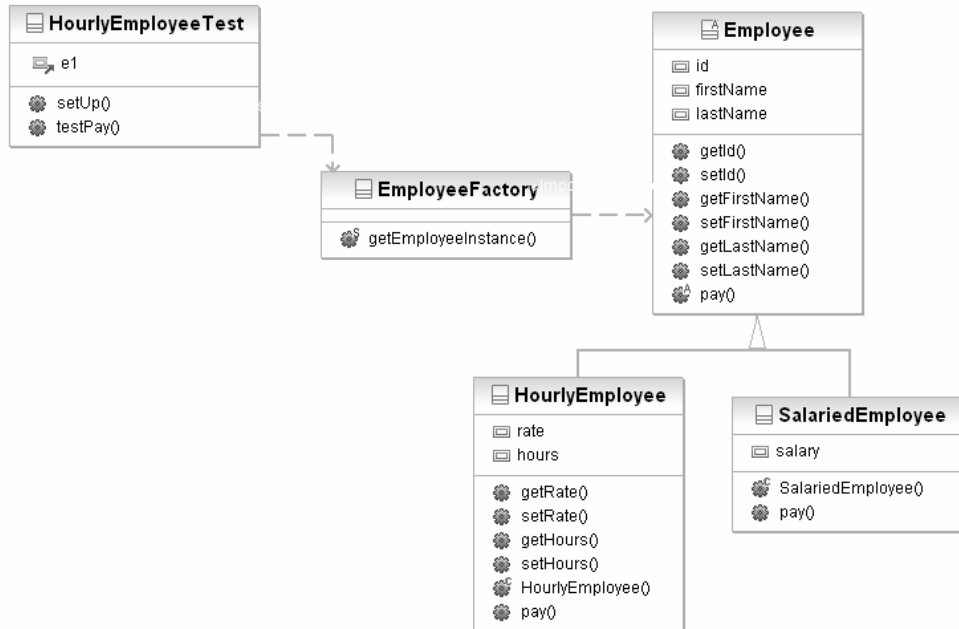
- We made two changes:
  - We added a Triangle class.
  - We modified the factory method.
- The key point is that we did **not** have to modify any client code.
  - This saves us from having to potentially change lots of code, both now and in the future (as new shapes are added).

**Notes**



## Patterns Are Best Practices

- Patterns aren't UML diagrams or even code
  - They're best practices – proven ways to solve problems
  - A given pattern's structure/design may look the same...



- The UML diagram above looks just like that of our shape example.

**Notes**



## Patterns Can Vary by Context

- ...but its implementation can be different in different contexts
  - There is a subtle change here, our method has parameters
    - One of the parameters represents variance (**hrs**)

```
public class EmployeeFactory
{
    // factory method - 0 hours for a salaried employee
    public static Employee getInstance(double pay, double hrs)
    {
        if (hours == 0)
        {
            return new SalariedEmployee(pay);
        }
        else
        {
            return new HourlyEmployee(pay, hrs);
        }
    }
}
```

- Model-View-Controller is an example of a pattern that can look quite different in the various contexts in which it's used.

**Notes**



## Spring Patterns

### Factory Method Pattern

Factories - Overview  
Simple Factory  
**Factory Method Pattern**  
Strategy Pattern  
Command Pattern  
Template Method Pattern

Notes



## The Factory Method Pattern - Overview

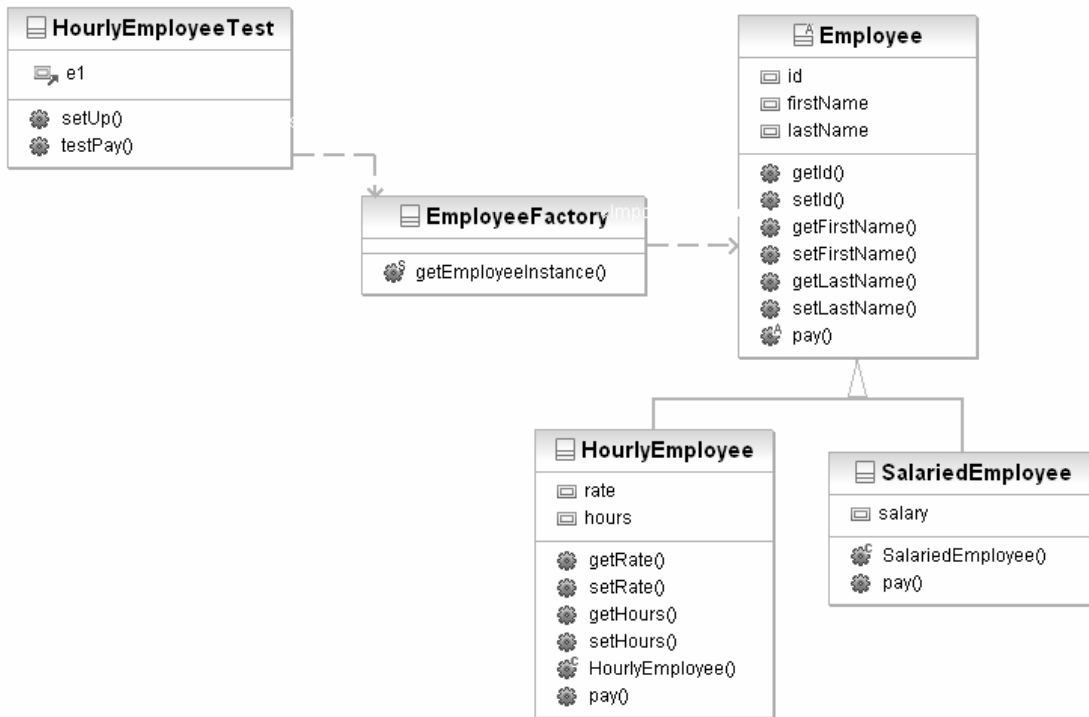
- Recall the GoF definition of the Factory Method Pattern:
  - *"Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses."* (GoF)
  
- When would I want to do that?
  - Frameworks
  - Parallel hierarchies
  - Any other time you need to defer instantiation to the subclass

Notes



# Factory Method Pattern - Example

- Let's revisit our HR example – pay is something that varies



Notes



## Factory Method Pattern - Example - Changes

- We get new requirements – employees can:
  - Buy life insurance
  - Give to charity
  - Contribute to their pension
  
- Your question:
  - For each new requirement, do all employees do this the same or differently?

Responsibility	Hourly Employee	Salaried Employee
pay	Varies	Varies
payLifePremium	Varies	Varies
giveToCharity	Same	Same
addToPension	Same	Same

- The table above is based on our HR example from Section 4.
  - Hourly employees and salaried employees are paid differently.
  - Hourly employees can buy up to \$100,000 of coverage. Salaried employees can buy up to \$250,000 of coverage.
  - All employees can add to their pensions the same way.
  - All employees can donate to charity in the same way.

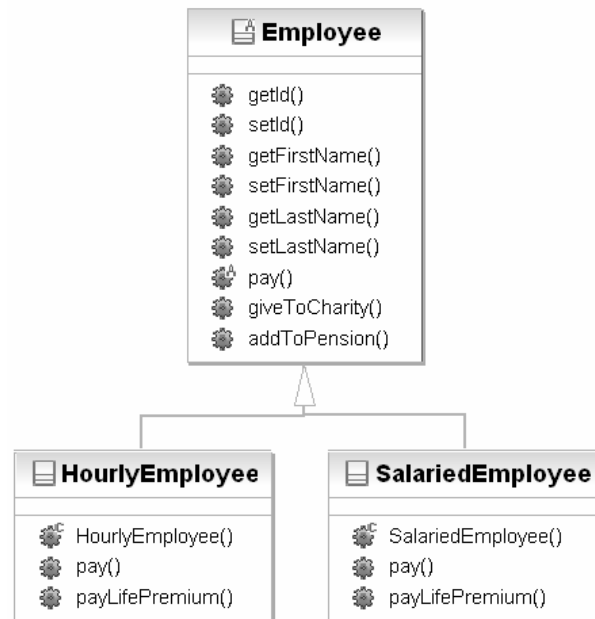
### Notes



## Factory Method Pattern - Example - Variation

### ■ Life insurance variation

- Hourly employees can get up to \$100,000 of coverage
- Salaried employees can get up to \$250,000 of coverage



Notes



## Factory Method Pattern - Example - More Types

- Until this point, we've handled the changes by encapsulating variation
  
- We just found out that our system will be used to handle employees who aren't citizens
  - We can have **citizens** and **visa workers**
  - Citizens can be hourly or salaried
  - Visa workers can be hourly or salaried
  
- Perhaps we need to do some CVA

Notes



## Factory Method Pattern - Example - CVA

- Employees working on a visa get reduced benefits
  - They can't purchase as much life insurance
    - \$50,000 max for hourly
    - \$75,000 max for salaried
  - More benefit reductions are being discussed

Responsibility	Hourly Employee (Citizen)	Hourly Employee (Visa)	Salaried Employee (Citizen)	Salaried Employee (Visa)
pay	rate*hours	?	salary	?
payLifePremium	<b>max 100k</b>	<b>max 50k</b>	<b>max 250k</b>	<b>max 75k</b>
giveToCharity	Same	Same	Same	Same
addToPension	Same	Same	Same	Same

Notes



## Factory Method Pattern - Example - CVA

- To make up for their reduced benefits, employees working on a visa get a pay differential
  - Additional 5% for hourly
  - Additional 7% for salaried

Responsibility	Hourly Employee (Citizen)	Hourly Employee (Visa)	Salaried Employee (Citizen)	Salaried Employee (Visa)
pay	rate*hours	rate*hours+5%	salary	salary+7%
payLifePremium	max 100k	max 50k	max 250k	max 75k
giveToCharity	Same	Same	Same	Same
addToPension	Same	Same	Same	Same

Notes



# Factory Method Pattern - Example - Subclasses

- Encapsulate what varies in the new subclasses



Notes



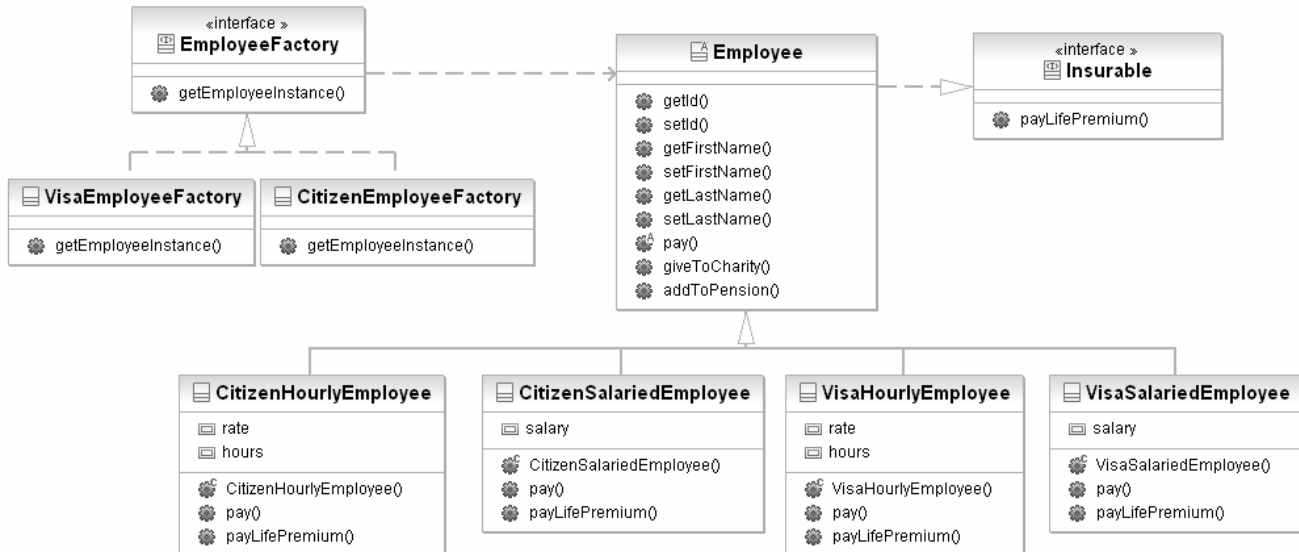
## Factory Method Pattern - Example - Which Type?

- From the client perspective, we will need to create hourly and salaried employees, of both visa and citizen types
  
- Let's assume that the client will know when it wants a visa- or citizen-type employee
  - However, **we don't want the client to be concerned with the details of hourly vs. salaried employees**

Notes

## Factory Method Pattern - Example - Solution

- We've deferred the decision of which concrete Employee class to instantiate to the appropriate factory subclass
  - VisaEmployeeFactory or CitizenEmployeeFactory



- In our example, the client knows whether it wants a visa- or citizen-type employee.
  - However, we don't want the client to be concerned with the details of hourly vs. salaried employees.
  - The client uses the appropriate factory – VisaEmployeeFactory or CitizenEmployeeFactory – and the factory takes care of the hourly/salaried details.

### Notes



## Factory Method Pattern - Example - The Factories

- Define the interface for creating Employee objects

```
public interface EmployeeFactory
{
    // factory method
    public Employee getInstance(double wage, double hours);
}
```

Notes



## Factory Method Pattern - Example - The Factories

- This factory creates **visa-type** Employees
  - And takes care of the hourly/salaried details

```
public class VisaEmployeeFactory
implements EmployeeFactory
{
    // factory method
    public Employee getInstance(double wage, double hours)
    {
        if (hours > 0)
        {
            return new VisaHourlyEmployee(wage, hours);
        }
        else
        {
            return new VisaSalariedEmployee(wage);
        }
    }
}
```

Notes



## Factory Method Pattern - Example - The Factories

- This factory creates **citizen-type** Employees
  - And takes care of the hourly/salaried details

```
public class CitizenEmployeeFactory
implements EmployeeFactory
{
    // factory method
    public Employee getInstance(double wage, double hours)
    {
        if (hours > 0)
        {
            return new CitizenHourlyEmployee(wage, hours);
        }
        else
        {
            return new CitizenSalariedEmployee(wage);
        }
    }
}
```

Notes



## Factory Method Pattern - Example - Client

- The client is not concerned with the hourly/salaried details
  - The resulting output is in the notes below

```
public class EmployeeClient
{
    public static void main(String[] args)
    {
        // use factory for visa-type employees
        EmployeeFactory visaFactory = new VisaEmployeeFactory();
        Employee e1 = visaFactory.getEmployeeInstance(45000, 0);
        e1.setFirstName("Jill");
        e1.setLastName("Salarygal");
        e1.pay();

        // use factory for citizen-type employees
        EmployeeFactory citizenFactory = new CitizenEmployeeFactory();
        Employee e2 = citizenFactory.getEmployeeInstance(13.45, 39);
        e2.setFirstName("Joe");
        e2.setLastName("Citizen");
        e2.pay();
    }
}
```

Your salaried pay is: \$48150.00  
Your total (hourly) pay is: \$524.55

### Notes

- Notice that the visa-type employee is salaried and is receiving the 7% pay differential. The citizen-type employee is hourly and is not receiving any additional pay.
  - The system works correctly and the client is unaware of these details. That is what we wanted.



## Section Review

- Factories reduce client dependencies
- Use the Factory Method Pattern when you want to defer instantiation to the subclasses
- Use the Abstract Factory Pattern when you want to create related "families" of objects
- Simple Factory isn't officially a pattern but it's good to know

Notes



## Spring Patterns

### Strategy Pattern

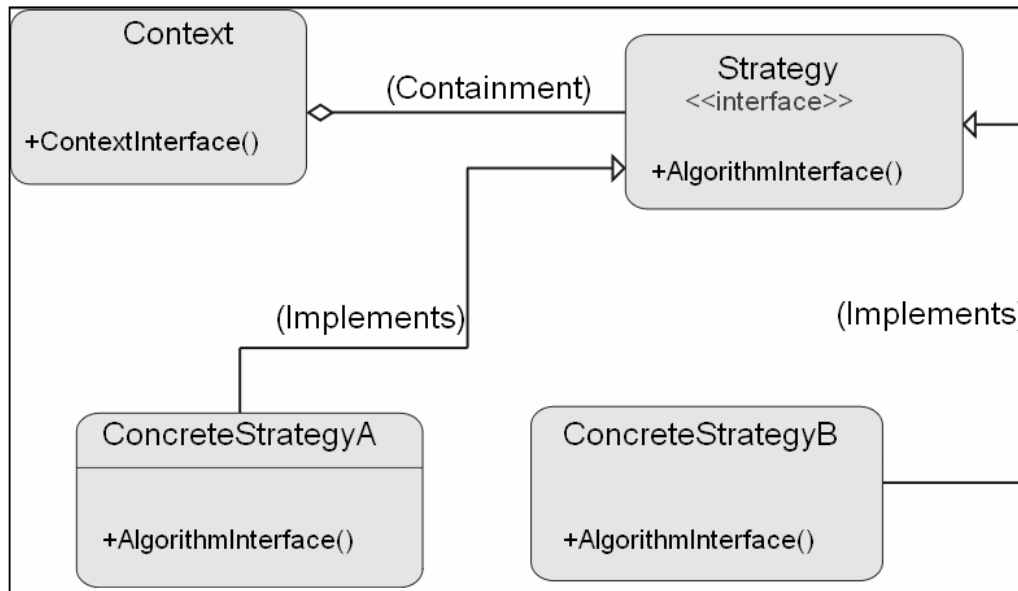
Factories - Overview  
Simple Factory  
Factory Method Pattern  
**Strategy Pattern**  
Command Pattern  
Template Method Pattern

Notes



## The Strategy Pattern

- The **Strategy Pattern** allows the selection of an algorithm to be made based upon the **context** in which it's used
  - Different clients can choose different implementations



- Situation: we must apply a business rule or use an algorithm. However, the exact implementation that makes sense varies by the context in which it's used.

**Notes**



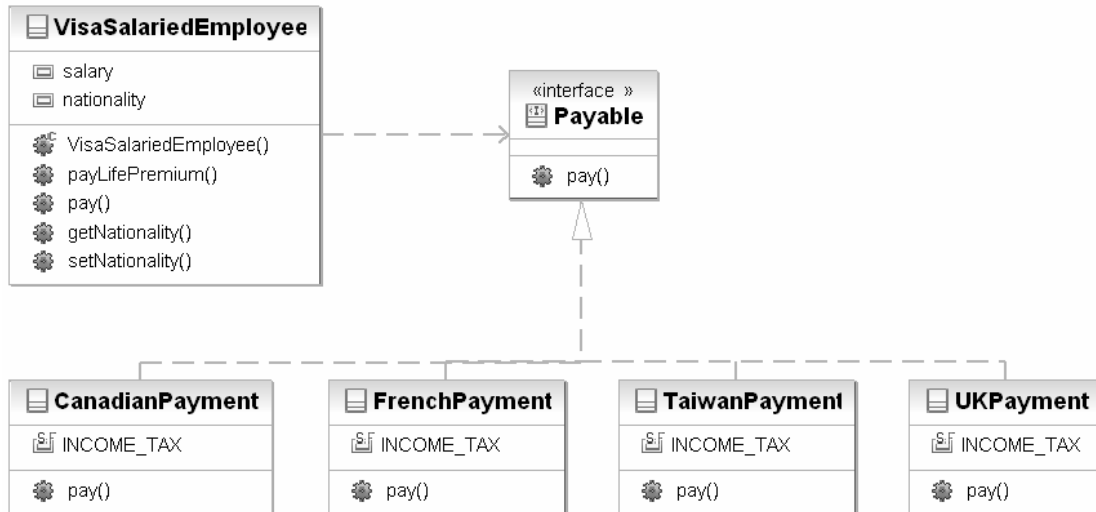
## Strategy Pattern - Example

- Our HR application has some new business requirements:
  - **Visa-type** employees want to be paid every two weeks; we've agreed to do it for **salaried** visa workers only
  - Due to the devaluation of the dollar, they also insist on being paid in their own currency
  
- There's another variance:
  - They all pay income tax differently
  
- So we have a business rule that must be executed, but **how** it's implemented varies
  - The implementation chosen is a function of the employee's nationality

Notes

## Strategy Pattern - Example - Our Approach

- How `VisaSalariedEmployee` are paid varies by **nationality**
  - The employee's nationality serves as the **context** in this example
- We choose an implementation of `Payable` accordingly



- The `Payable` interface encapsulates an algorithm that has several implementations – `CanadianPayment`, `FrenchPayment`, etc.
- The client in this example is `VisaSalariedEmployee`.
  - It uses a `Payable` in its `pay` method. It chooses the `Payable` implementation as a function of the employee's **nationality**, which serves as the **context**.
- Once again, we have delegation going on here, which will look something like this in `VisaSalariedEmployee`:
 

```

public double pay()
{
    Payable payable = // get Payable somehow
    double payment = payable.pay(salary * 1.07);
    return payment;
}
      
```

### Notes



## Strategy Pattern - Example - Payable

- The common interface to our implementations
  - Clients will reference the interface, not an implementation
    - This allows different implementations to be used in different contexts

```
public interface Payable
{
    public double pay(double salary);
}
```

- Do we have another option instead of using an interface here?

**Notes**



## Strategy Pattern - Example - Implementations

- The variables affecting payment are income tax, currency rate
  - These are specific to a nationality
- CanadianPayment and the others are written similarly

```
public class UKPayment implements Payable {
    public double pay(double salary) {
        // pay bimonthly and deduct taxes
        double net = (salary / 26) *
            ((100 - Nationality.INCOME_TAX_UK) / 100);

        // pay in local currency
        double payment = net * Nationality.CURRENCY_RATE_UK;

        // output results and return payment
        NumberFormat format =
            NumberFormat.getCurrencyInstance(Locale.UK);
        System.out.println("Your bimonthly salaried pay, " +
            "in local currency, is: " + format.format(payment));
        return payment;
    }
}
```

Notes



## Strategy Pattern - Example - Variances

- We encapsulate the variables into a **Nationality** type
  - This keeps all the variances in one place
- We also define the nationalities themselves here

```
public interface Nationality {
    public static final int NATIONAL_CANADA = 1;
    public static final int NATIONAL_FRENCH = 2;
    public static final int NATIONAL_TAIWAN = 3;
    public static final int NATIONAL_UK = 4;

    public static final double CURRENCY_RATE_CANADA = 1.029;
    public static final double CURRENCY_RATE_EURO = 0.694;
    public static final double CURRENCY_RATE_TAIWAN = 32.25;
    public static final double CURRENCY_RATE_UK = 0.51;

    public static final double INCOME_TAX_CANADA = 45;
    public static final double INCOME_TAX_FRANCE = 35;
    public static final double INCOME_TAX_TAIWAN = 18;
    public static final double INCOME_TAX_UK = 40;
}
```

- For each nation, we have:
  - A nationality constant, e.g., NATIONAL\_CANADA.
  - A currency rate, e.g., CURRENCY\_RATE\_CANADA.
  - An income tax rate, e.g., INCOME\_TAX\_CANADA.

**Notes**



## Spring Patterns

### Command Pattern

Factories - Overview  
Simple Factory  
Factory Method Pattern  
Strategy Pattern  
**Command Pattern**  
Template Method Pattern

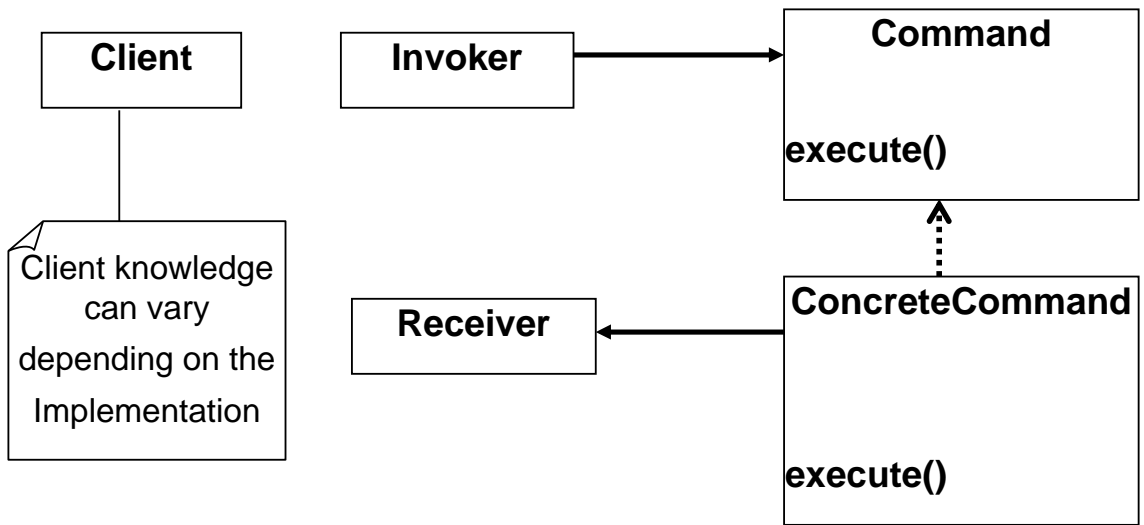
Notes



# Command Pattern

■ GoF tell us

- "Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations



Notes



## Command Pattern

- The classic example – ordering a meal in a restaurant
- Customer – sends message to create order
- Order
  - Encapsulates your command
  - Associated with a particular patron (you)
- Waiter – takes the order, goes to kitchen and places order
- Cook
  - Order and Cook are associated (Cook has the Order)
  - Food is then associated with the Order
    - Note that food item can change when command changes

Notes



## Command Pattern

- Client creates command object (customer)
  
- Command wraps receiver (Cook) of object
  - Provides one method like "orderUp" or "execute" that encapsulates real action
    - Action might be "makeBurger" or "addCustomer"
  
- Client calls Invoker (waitress) and passes in Command object
  
- Invoker calls execute
  
- Command calls real action it encapsulates
  - Cook.makeBurger
  - MyServlet.addCustomer

Notes

---



## Spring Patterns

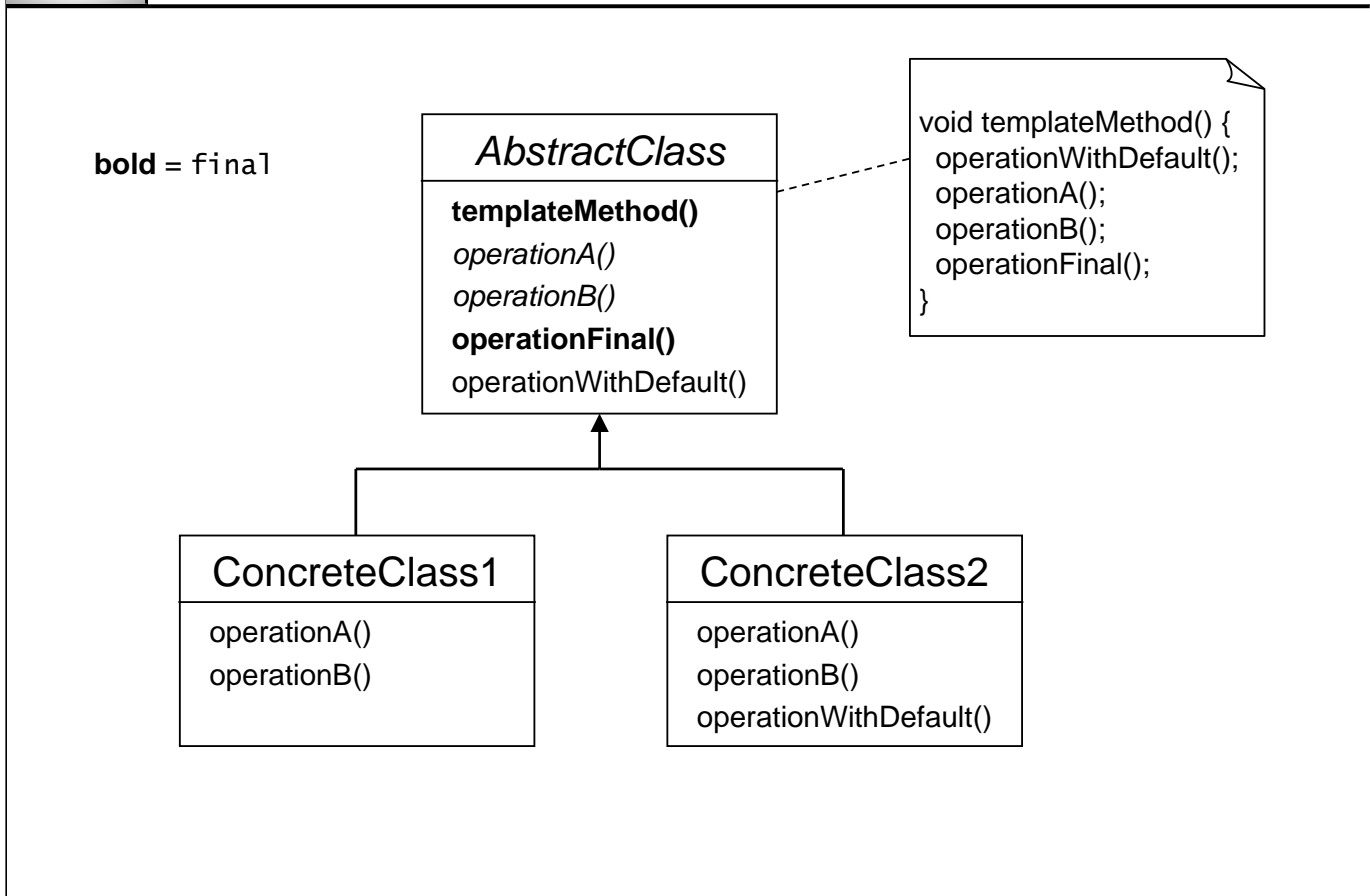
### Template Method Pattern

Factories - Overview  
Simple Factory  
Factory Method Pattern  
Strategy Pattern  
Command Pattern  
**Template Method Pattern**

Notes



## The Template Method Pattern - Class Diagram



- The methods shown in **bold** are marked as `final`.

### Notes

- The operation methods defined in the abstract base class are not **required** to be abstract. There is a great deal of flexibility and control given to the base class in regard to what subclasses **must do**, **cannot do**, and **may optionally do**.
  - In the diagram above, *operationA* and *operationB* are abstract and **must** be overridden by concrete subclasses.
  - The *operationFinal* method is marked `final` and therefore **cannot** be overridden by subclasses.
    - This allows the base class to provide some of the template method's implementation in an invariant way.
  - The *operationWithDefault* method is concrete and the base class provides a default implementation.
    - Subclasses have the **option** of overriding this method.
    - These methods are sometimes called "hooks."



## The Template Method Pattern - Diagram Details

- The `operationA` and `operationB` methods are **abstract**
  - Concrete subclasses **must override** them
    - They do so to provide variant behavior
  
- The `operationWithDefault` method is **concrete**
  - Subclasses can **optionally override** this method
  - Such a method is sometimes called a **"hook"**
    - A method declared in the abstract class and given a default or empty implementation
  
- The `operationFinal` method is **concrete** and **final**
  - Subclasses **cannot override** this method
  - This behavior is invariant and therefore it is implemented exclusively in the abstract base class

- Use **abstract** operation methods when you want to **require** subclasses to provide the behavior.
  - Such behavior is "purely variant" and cannot be implemented in the base class.
  
- Use **concrete** operation methods ("hooks") when you want to **allow, but not require**, subclasses to provide the behavior.
  - If the subclass does not override the hook, it uses the inherited implementation.
  
- Use **concrete and final** operation methods when you want to **prevent** subclasses from providing the behavior.
  - Such behavior is invariant and belongs only in the base class.

Notes



## The Template Method Pattern - Uses

- Separate commonality from variance
  - The **invariant** behavior is placed in the abstract base class, in the template method
  - Subclasses override the operation methods, providing the **variant** behavior
  
- Exert some control over the variance
  - The template method governs the selection and sequence of the operation methods
    - Allows the template method to stay in control of the big picture
    - Subclasses can override operations in the algorithm without changing its overall structure
  - The abstract class can also control at what point(s) overriding is allowed
    - By providing some of the operation methods itself (marked `final`)

Notes



## The Template Method Pattern - Benefits

- Avoid redundancy
  - Place the common code in the abstract base class
  - Implement only the variants in the subclasses
  
- Better cohesion
  - The general algorithm is saved in one place but the concrete steps are performed by the subclasses
  
- Ease of development
  - Subclasses can implement the operation methods without regard to the complexities of the overall algorithm

Notes



## Template Pattern - Example

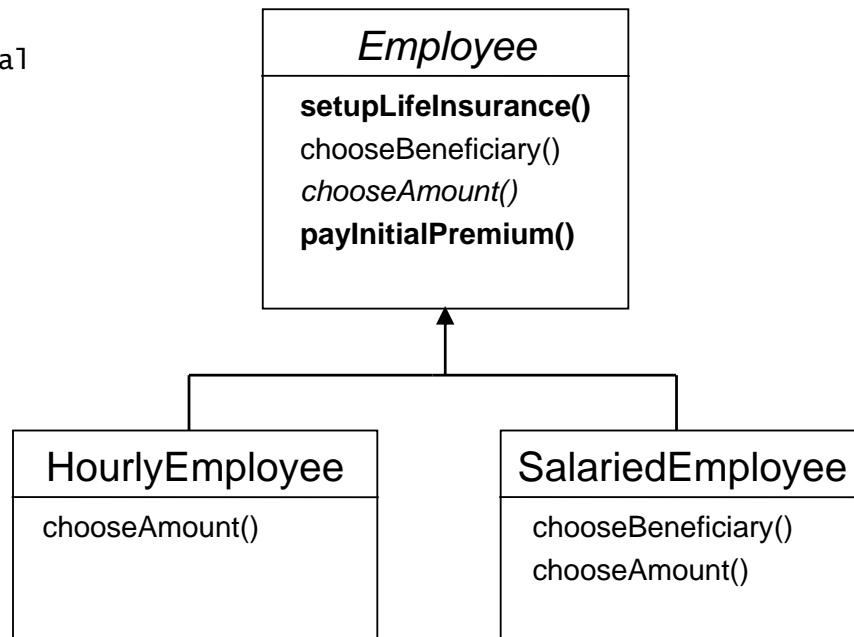
- We will use the template method pattern to define the steps needed for employees to purchase life insurance
  
- The required steps are, in this order:
  1. Choose coverage amount                      abstract
  2. Designate beneficiary                      concrete with default impl.
  3. Pay initial premium                      final
  
- The Employee base class contains the template method
  - It will also provide a default implementation for step (2) and the invariant implementation for step (3)
  
- Concrete subclasses **must** provide an implementation for step (1) and **may** override the default implementation for step (2)

Notes



## Template Pattern - Example - Class Diagram

**bold = final**



- The methods shown in **bold** are marked as `final`.
- The template method is `setupLifeInsurance`.
- Both concrete subclasses **must** override the `chooseAmount` method with an implementation.
- `HourlyEmployee` does not override `chooseBeneficiary`, so the default implementation in `Employee` will be used.
- `SalariedEmployee` overrides `chooseBeneficiary` with a custom implementation.
- Neither subclass can override the `payInitialPremium` method, because it is marked as `final`.
  - This functionality is invariant and therefore it is implemented exclusively in the base class.

**Notes**



## Template Pattern - Example - Abstract Class

```
public abstract class Employee {
    public final void setupLifeInsurance(int amount, String bene) {
        this.chooseAmount(amount);
        this.chooseBeneficiary(bene);
        this.payInitialPremium();
    }

    protected void chooseBeneficiary(String beneficiary) {
        if (beneficiary != null) {
            System.out.println("Your beneficiary is " + beneficiary);
        }
        else {
            System.out.println("No beneficiary selected at this time");
        }
    }

    protected abstract void chooseAmount(int amount);

    protected final void payInitialPremium() {
        System.out.println("Initial premium paid in cash");
    }
}
```

- Notice that the template method is `public`, whereas the operation methods are `protected`. While this is not required, it is common.
  - Clients only call the template method, not the individual operation methods.

### Notes



## Template Pattern - Example - Subclasses

```
public class HourlyEmployee
extends Employee {

    // override abstract operation method
    protected void chooseAmount(int amount) {
        System.out.println("You, a wage worker, have requested " +
            amount + " in coverage");
        System.out.println("Your maximum coverage amount is 100000");
        System.out.println("Your actual coverage amount will be " +
            String.valueOf(amount > 100000 ? 100000 : amount));
    }
}
```

Notes



## Template Pattern - Example - Subclasses

```
public class SalariedEmployee
extends Employee {

    // override abstract operation method
    protected void chooseAmount(int amount) {
        System.out.println("You, a salaried employee, have requested " +
            amount + " in coverage");
        System.out.println("Your maximum amount is 250000");
        System.out.println("Your actual coverage amount will be " +
            String.valueOf(amount > 250000 ? 250000 : amount));
    }

    // optionally override concrete operation method
    protected void chooseBeneficiary(String beneficiary) {
        if (beneficiary != null) {
            System.out.println("Your beneficiary is " + beneficiary);
        }
        else {
            System.out.println("Your beneficiary will be your estate");
        }
    }
}
```

Notes



## Template Pattern - Example - Client

```
public class EmployeeTest
{
    public static void main(String[] args)
    {
        Employee e1 = new HourlyEmployee();
        e1.setupLifeInsurance(500000, "Jack Beau");
        System.out.println();

        Employee e2 = new HourlyEmployee();
        e2.setupLifeInsurance(50000, null);
        System.out.println();

        Employee e3 = new SalariedEmployee();
        e3.setupLifeInsurance(500000, "Jill Gardner");
        System.out.println();

        Employee e4 = new SalariedEmployee();
        e4.setupLifeInsurance(50000, null);
        System.out.println();
    }
}
```

**Notes**



## Template Pattern - Example - the Results

You, **a wage worker**, have requested 500000 in coverage  
 Your maximum coverage amount is **100000**  
 Your actual coverage amount will be 100000  
 Your beneficiary is Jack Beau  
 Initial premium paid in cash

You, **a wage worker**, have requested 50000 in coverage  
 Your maximum coverage amount is **100000**  
 Your actual coverage amount will be 50000  
**No beneficiary selected at this time**  
 Initial premium paid in cash

You, **a salaried employee**, have requested 500000 in coverage  
 Your maximum coverage amount is **250000**  
 Your actual coverage amount will be 250000  
 Your beneficiary is Jill Gardner  
 Initial premium paid in cash

You, **a salaried employee**, have requested 50000 in coverage  
 Your maximum coverage amount is **250000**  
 Your actual coverage amount will be 50000  
**Your beneficiary will be your estate**  
 Initial premium paid in cash

- The behavior of the chooseAmount method varies, depending on what type of employee we have.
  - They have different output messages and different maximum coverages.
- The behavior of the chooseBeneficiary method also varies.
  - If an hourly employee does not specify a beneficiary, the output is "No beneficiary selected at this time," which is the default behavior provided in the base Employee class.
  - If a salaried employee does not specify a beneficiary, the output is "Your beneficiary will be your estate," which is the behavior provided in the SalariedEmployee subclass.
- The behavior of the payInitialPremium method is invariant, as it is provided exclusively in the base class, and cannot be overridden.

**Notes**



## Documentation of Template Methods

- Documentation of methods is always important – doubly so for the methods that comprise a Template Method Pattern
  
- Document what the abstract operation methods must do
  - And how they fit into the overall algorithm
  
- Document the hook methods (concrete operation methods)
  - What they should do if overridden
  - The behavior of the implementation in the base class
    - Whether it is empty, i.e., { }, or provides some default functionality
  - How the hook fits into the overall algorithm

Notes

---



## The Hollywood Principle

- An interesting thing about the Template Method Pattern is that the usual control structure is reversed
  - It is the parent class that calls the method(s) in the subclass
  
- This "inversion of control" is often referred to as the ***Hollywood Principle***
  - It gets its name from the saying, "Don't call us, we'll call you"
  
- The focus is on the subclass's operation methods, not on the control of those methods (which is done by the base class)
  - These methods are similar to callbacks that respond to events
  - Or lifecycle methods that get called by a JavaEE container
    - For example, the `init`, `service`, and `destroy` methods of a servlet

- In the Hollywood Principle paradigm, the `AbstractClass` is playing the part of the agent. It will call the `ConcreteClass` if and when the time comes. The `ConcreteClass` is playing the part of the actor, anxiously awaiting a call.

**Notes**



## Thoughts on the Template Method Pattern

- "Template methods are so fundamental that they can be found in almost every abstract class"
  - Design Patterns
  
- Template methods can and should be **designed** at the design stage
  
- Template methods can be found **after the fact**, as part of a refactoring to remove code duplication
  - Also very valuable

Notes

---