Section 1

Patterns in Spring

Patterns Used In Spring
Factories - Overview
Simple Factory
Strategy Pattern
Command Pattern
Template Method Pattern

Patterns found in Spring

- Factory
- Builder
- Singleton
- Prototype
- Proxy
- Dynamic Proxy
- Adapter
- Decorator
- Façade
- Observer
- Strategy
- Template Method
- Command
- Front Controller
- MVC

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

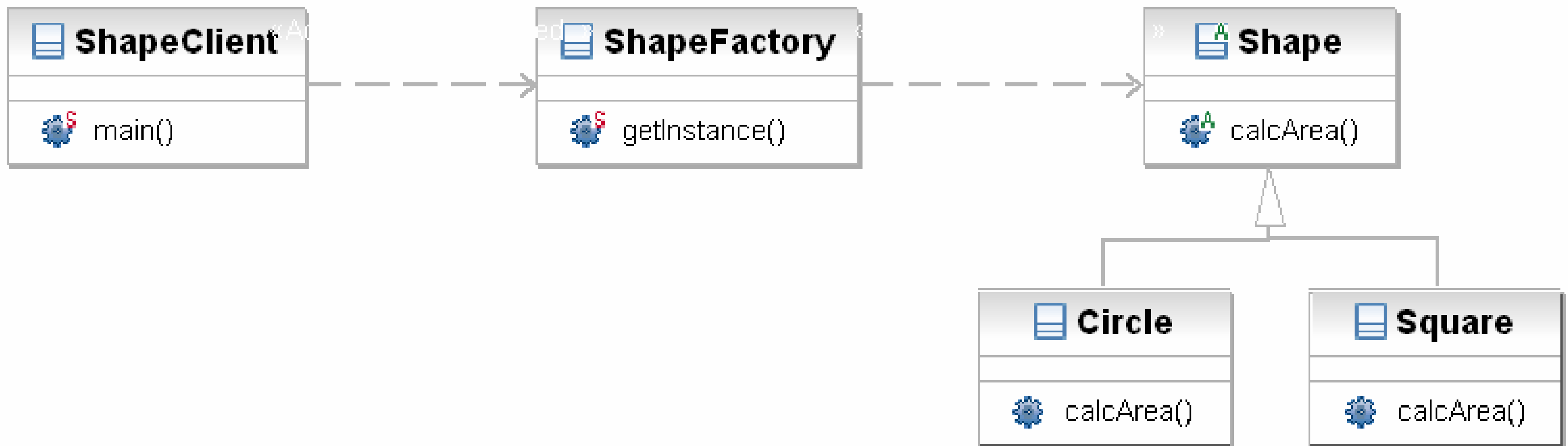
A Simple Factory - Overview

- 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

Simple Factory - Example

- We have a Shape abstract base class and a number of concrete subclasses

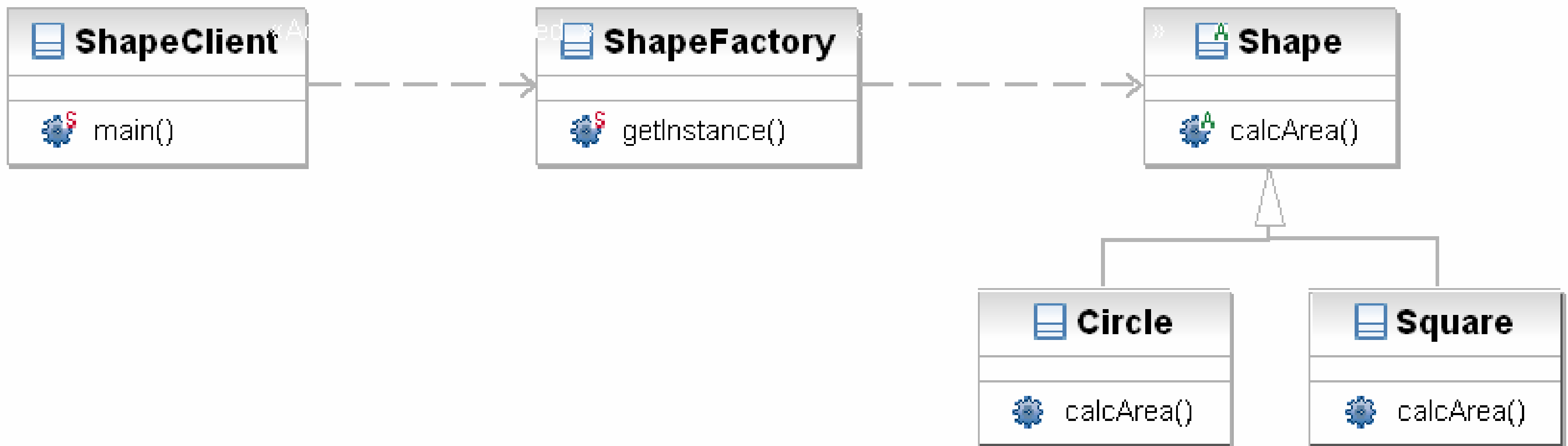
- We don't want client(s) to know the types of Shapes we have
  - This way, the client is not dependent on concrete types of Shape
  - The types of Shapes can vary without breaking the client



Separating instantiation from implementation

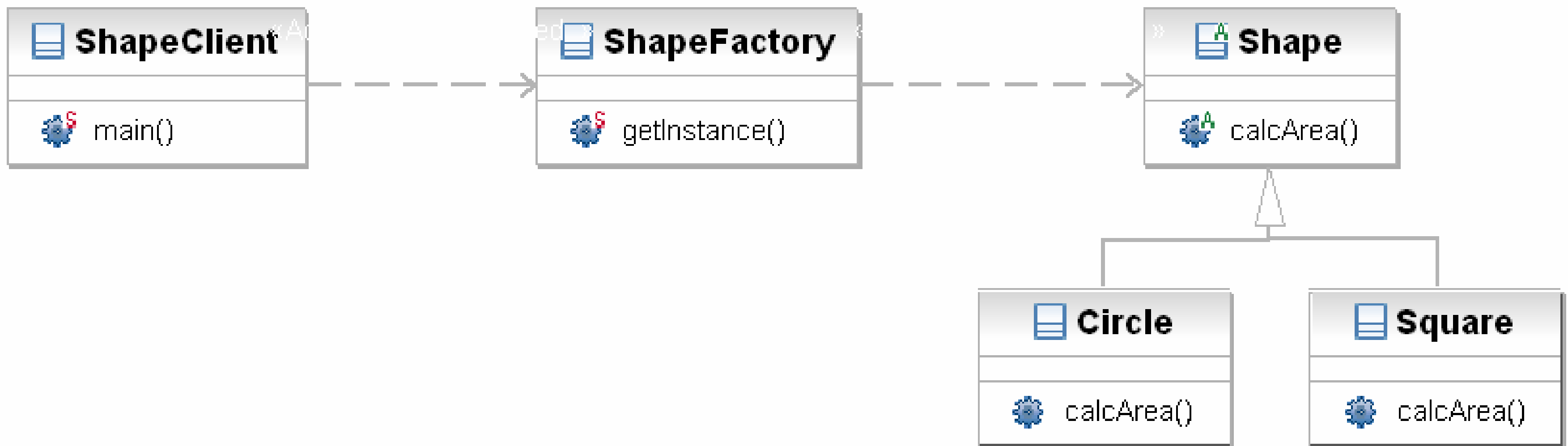
- Before the factory the client "asked" for a shape type by invoking new. The client was dependent on implementation

- After the factory the client asks the ShapeFactory for a Shape. The client is coupled (dependent) to the ShapeFactory but not to a particular shape type.



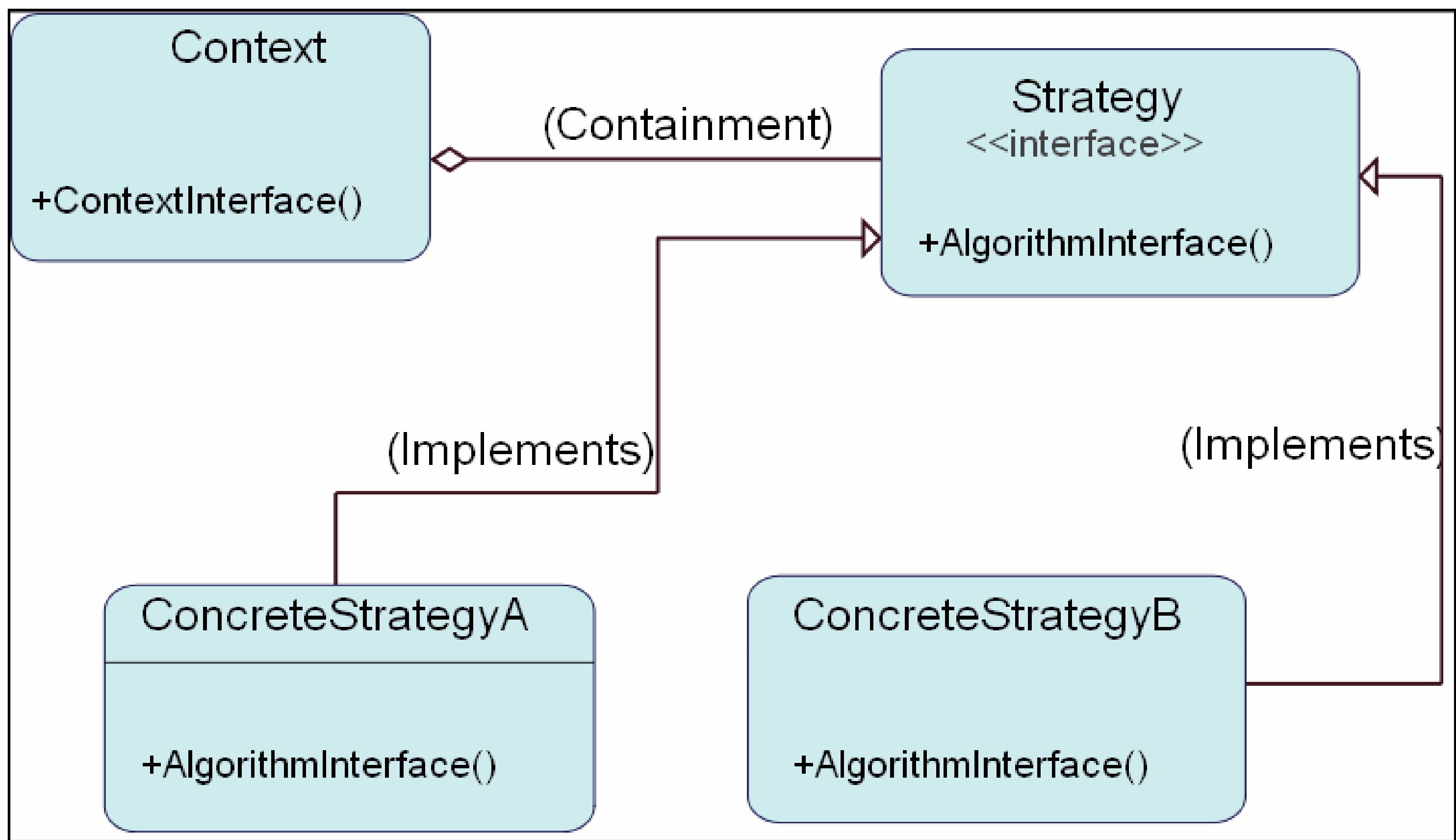
Dependency Injection

- What if Spring gave the right type of Shape during runtime?
- We remove the dependencies from code (even a factory) and inject them into the right client during runtime



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



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

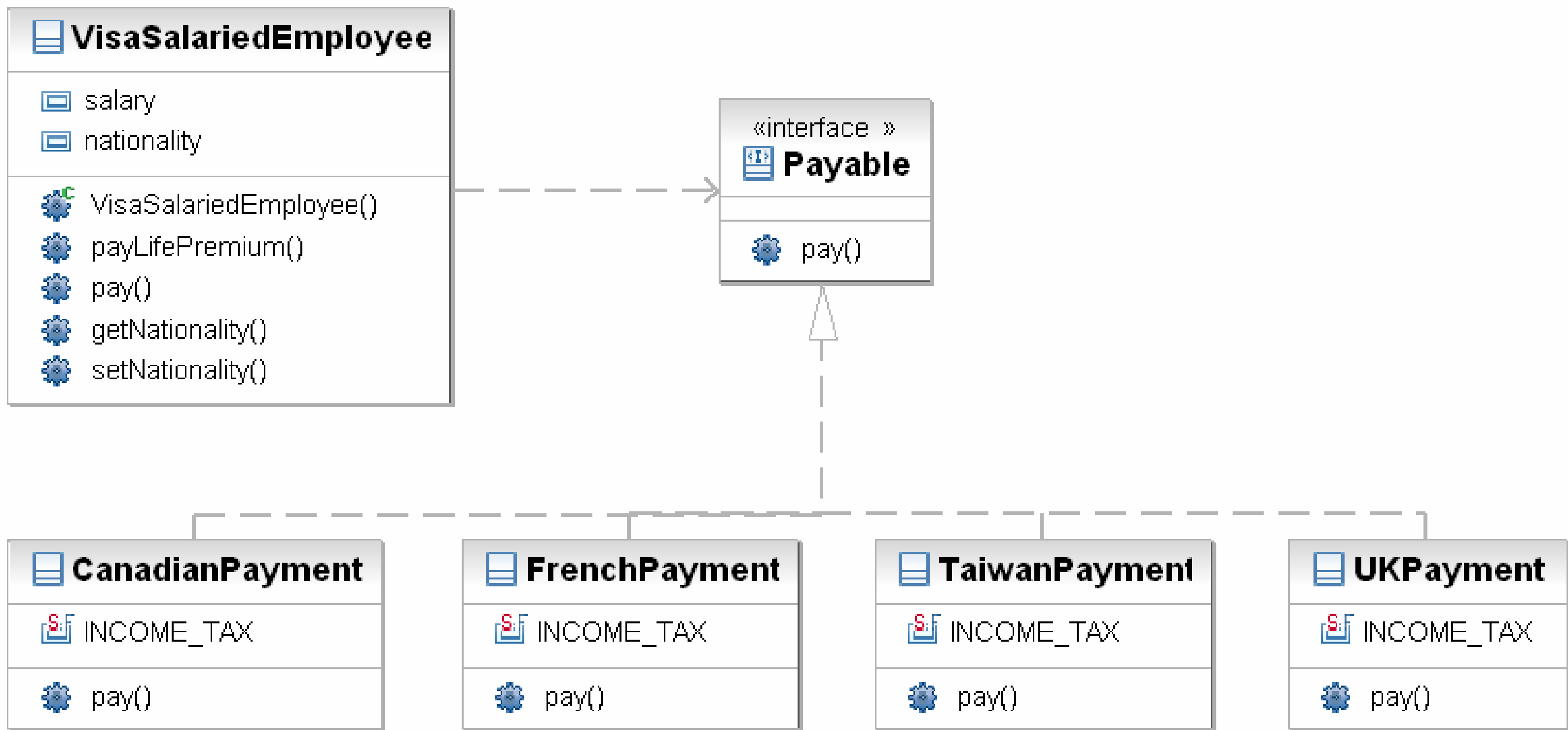
- There's another variance:
  - They all pay income tax differently

- 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

Strategy Pattern - Example - Our Approach

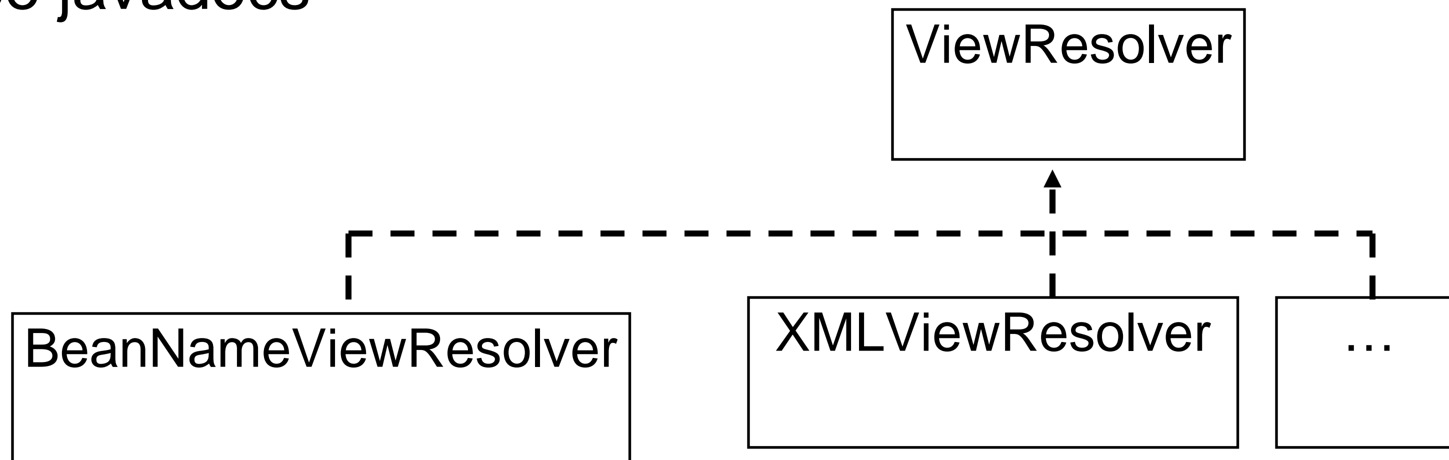
- **How** `VisaSalariedEmployees` are paid varies by **nationality**
  - The employee's nationality serves as the **context** in this example

- We choose an implementation of `Payable` accordingly



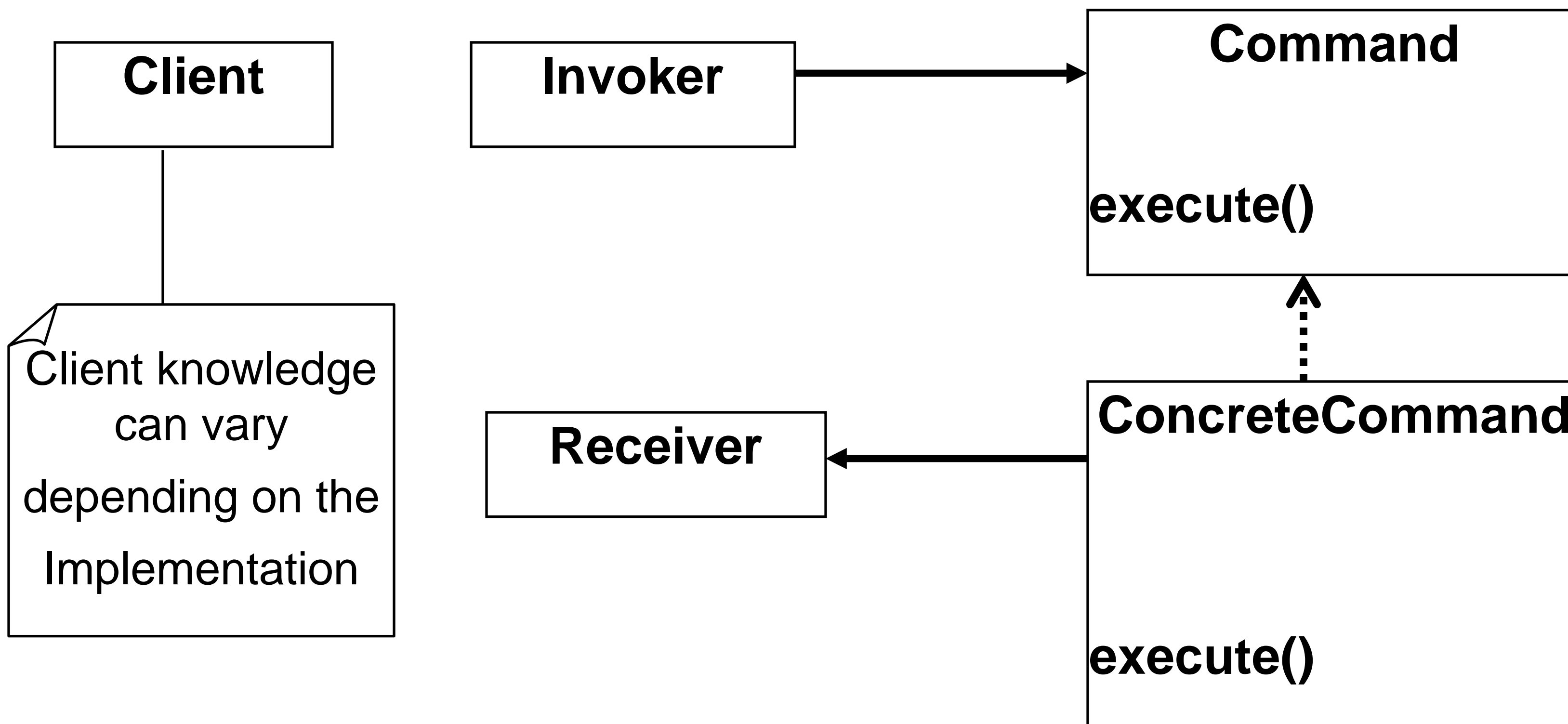
Strategies in Spring

- Handler Mapping
- Handler
- View
- ViewResolver
- See javadocs



Command Pattern

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



Copyright © 2007-8 nTier Training, Inc. All rights reserved.

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

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

Command Pattern in Spring

- So what do you do when you have a whole “menu” (think restaurant) of items (“commands”) you want to send from the presentation layer?
- AbstractCommandController –
  – Autopopulates a command bean from the request.
  – For command validation, a validator (property inherited from BaseCommandController) can be used.
- AbstractFormController –
  – Form controller that auto-populates a form bean from the request

The Template Method Pattern - Class Diagram

bold = `final`

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

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`)

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

ntier TRAINING

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

Template Method Pattern in Spring

- JDBCTemplate
- HibernateTemplate
- HibernateDAOSupport
  - Implements InitializingBean interface, one method afterPropertiesSet. Note – final method
  - Developers can provide more init functionality by overriding the initDAO method

- Extra credit
  - Servlets…think about doPost, doGet…and the service method
  - Who invokes service?
  - What does service do?