REin
REliable, INtelligent & Scalable Systems

互联网应用开发技术

Web Application Development

第19课
设计模式 – 创建型模式

Episode Nineteen

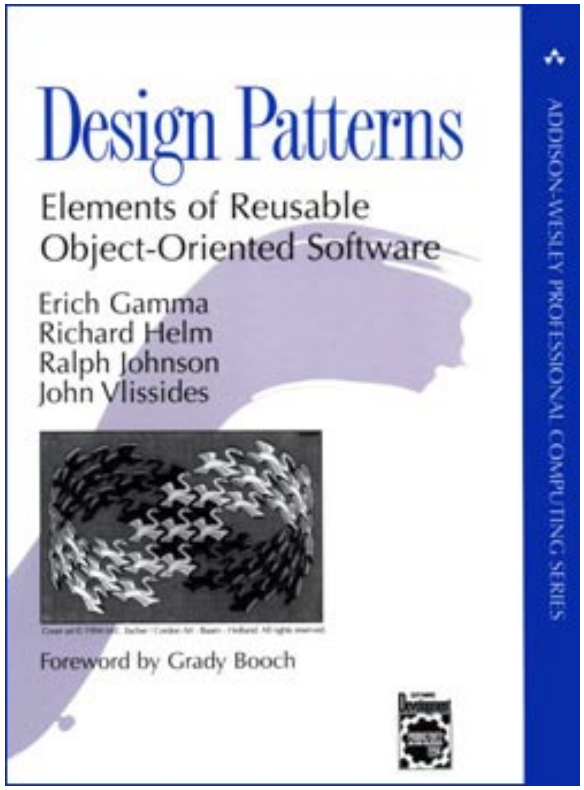
Creational Design Patterns

陈昊鹏

chen-hp@sjtu.edu.cn

Agenda

- What is a Design Pattern?
- Design Patterns Categories
- GoF Design Patterns
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns
- Software Framework



What is a Design Pattern ?

- A design pattern is a solution to a common design problem.
  - Describes a common design problem
  - Describes the solution to the problem
  - Discusses the results and trade-offs of applying the pattern
- Design patterns provide the capability to reuse successful designs.

Pattern and Framework

- Pattern
  - Provides a common solution to a common problem in a context
- Analysis/Design pattern
  - Provides a solution to a narrowly-scoped technical problem
  - Provides a fragment of a solution, or a piece of the puzzle
- Framework
  - Defines the general approach to solving the problem
  - Provides a skeletal solution, whose details may be Analysis/Design patterns

Design Patterns Categories

REin
REliable, INtelligent & Scalable Systems

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy Flyweight	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Creational Patterns

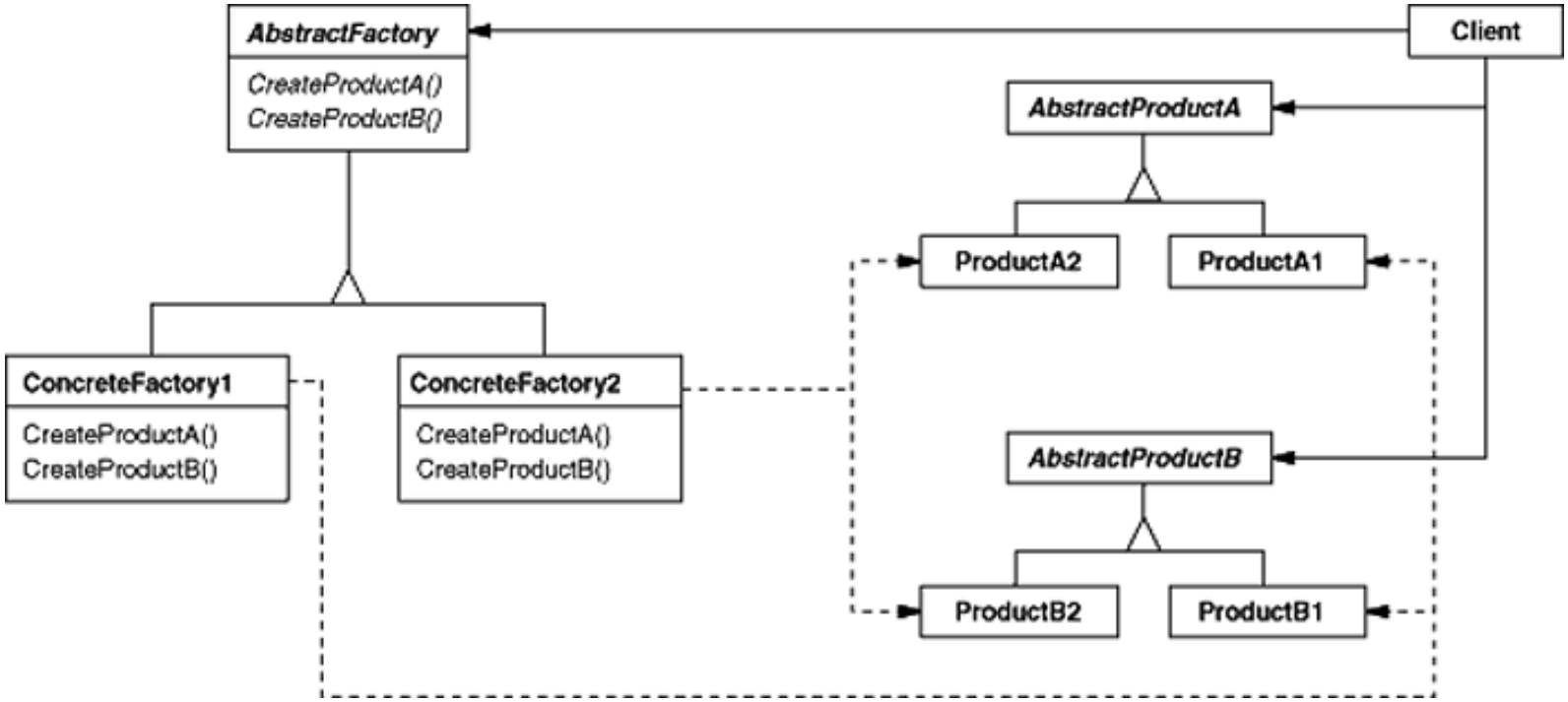
- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Abstract Factory

- Intent
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Abstract Factory

- Structure



Abstract Factory

- Applicability
- Use the Abstract Factory pattern when
  - a system should be independent of how its products are created, composed, and represented.
  - a system should be configured with one of multiple families of products.
  - a family of related product objects is designed to be used together, and you need to enforce this constraint.
  - you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Abstract Factory

- Consequences

- The Abstract Factory pattern has the following benefits and liabilities:
  - It isolates concrete classes.
  - It makes exchanging product families easy.
  - It promotes consistency among products.
  - Supporting new kinds of products is difficult.

Abstract Factory

- Implementation

- Here are some useful techniques for implementing the Abstract Factory pattern.
  - Factories as singletons.
  - Creating the products.
  - Defining extensible factories.

Abstract Factory

- AbstractSoupFactory.java //an Abstract Factory

```
abstract class AbstractSoupFactory {
 String factoryLocation;
 public String getFactoryLocation() {return factoryLocation;}
 public ChickenSoup makeChickenSoup() {return new ChickenSoup();}
 public ClamChowder makeClamChowder() {return new ClamChowder();}
 public FishChowder makeFishChowder() {return new FishChowder();}
 public Minnestrone makeMinnestrone() {return new Minnestrone();}
 public Pastafazul makePastafazul() {return new Pastafazul();}
 public TofuSoup makeTofuSoup() {return new TofuSoup();}
 public VegetableSoup makeVegetableSoup() {return new VegetableSoup();}
}
```

Abstract Factory

- Soup.java //helper classes

```
import java.util.ArrayList;
import java.util.ListIterator;

abstract class Soup {
 ArrayList soupIngredients = new ArrayList();
 String soupName;
 public String getSoupName() { return soupName; }
 public String toString() {
  StringBuffer stringOfIngredients = new StringBuffer(soupName);
  stringOfIngredients.append(" Ingredients: ");
  ListIterator soupIterator = soupIngredients.listIterator();
  while (soupIterator.hasNext())
   { stringOfIngredients.append((String)soupIterator.next()); }
  return stringOfIngredients.toString();
 }
}
```

Abstract Factory


```
class ChickenSoup extends Soup {
 public ChickenSoup() {
  soupName = "ChickenSoup";
  soupIngredients.add("1 Pound diced chicken");
  soupIngredients.add("1/2 cup rice");
  soupIngredients.add("1 cup bullion");
  soupIngredients.add("1/16 cup butter");
  soupIngredients.add("1/4 cup diced carrots");
 }
}
class ClamChowder extends Soup {
 public ClamChowder() {
  soupName = "ClamChowder";
  soupIngredients.add("1 Pound Fresh Clams");
  soupIngredients.add("1 cup fruit or vegetables");
  soupIngredients.add("1/2 cup milk");
  soupIngredients.add("1/4 cup butter");
  soupIngredients.add("1/4 cup chips");
 }
}
```

Abstract Factory


```
class FishChowder extends Soup {
 public FishChowder() {
  soupName = "FishChowder";
  soupIngredients.add("1 Pound Fresh fish");
  soupIngredients.add("1 cup fruit or vegetables");
  soupIngredients.add("1/2 cup milk");
  soupIngredients.add("1/4 cup butter");
  soupIngredients.add("1/4 cup chips");
 }
}

class Minnestrone extends Soup {
 public Minnestrone() {
  soupName = "Minestrone";
  soupIngredients.add("1 Pound tomatos");
  soupIngredients.add("1/2 cup pasta");
  soupIngredients.add("1 cup tomato juice");
 }
}
```

Abstract Factory


```
class Pastafazul extends Soup {
 public Pastafazul() {
  soupName = "Pasta fazul";
  soupIngredients.add("1 Pound tomatos");
  soupIngredients.add("1/2 cup pasta");
  soupIngredients.add("1/2 cup diced carrots");
  soupIngredients.add("1 cup tomato juice");
 }
}

class TofuSoup extends Soup {
 public TofuSoup() {
  soupName = "Tofu Soup";
  soupIngredients.add("1 Pound tofu");
  soupIngredients.add("1 cup carrot juice");
  soupIngredients.add("1/4 cup spirolena");
 }
}
```

Abstract Factory


```
class VegetableSoup extends Soup {
  public VegetableSoup() {
    soupName = "Vegetable Soup";
    soupIngredients.add("1 cup bullion");
    soupIngredients.add("1/4 cup carrots");
    soupIngredients.add("1/4 cup potatoes");
  }
}
```

Abstract Factory

- BostonConcreteSoupFactory.java

```
// One of Two concrete factories extending the abstract factory
class BostonConcreteSoupFactory extends AbstractSoupFactory {
 public BostonConcreteSoupFactory() {factoryLocation = "Boston";}
 public ClamChowder makeClamChowder() {return new BostonClamChowder();}
 public FishChowder makeFishChowder() {return new BostonFishChowder();}
}

class BostonClamChowder extends ClamChowder {
 public BostonClamChowder() {
   soupName = "QuahogChowder";
   soupIngredients.clear();
   soupIngredients.add("1 Pound Fresh Quahogs");
   soupIngredients.add("1 cup corn");
   soupIngredients.add("1/2 cup heavy cream");
```

Abstract Factory


```
  soupIngredients.add("1/4 cup butter");
  soupIngredients.add("1/4 cup potato chips");
 }
}
class BostonFishChowder extends FishChowder {
 public BostonFishChowder() {
  soupName = "ScrodFishChowder";
  soupIngredients.clear();
  soupIngredients.add("1 Pound Fresh Scrod");
  soupIngredients.add("1 cup corn");
  soupIngredients.add("1/2 cup heavy cream");
  soupIngredients.add("1/4 cup butter");
  soupIngredients.add("1/4 cup potato chips");
 }
}
```

Abstract Factory

- HonoluluConcreteSoupFactory.java

```
// Two of Two concrete factories extending the abstract factory
class HonoluluConcreteSoupFactory extends AbstractSoupFactory {
 public HonoluluConcreteSoupFactory() {factoryLocation = "Honolulu";}
 public ClamChowder makeClamChowder() {return new HonoluluClamChowder();}
 public FishChowder makeFishChowder() {return new HonoluluFishChowder();}
}

class HonoluluClamChowder extends ClamChowder {
 public HonoluluClamChowder() {
   soupName = "PacificClamChowder";
   soupIngredients.clear();
   soupIngredients.add("1 Pound Fresh Pacific Clams");
   soupIngredients.add("1 cup pineapple chunks");
   soupIngredients.add("1/2 cup coconut milk");
```

Abstract Factory


```
  soupIngredients.add("1/4 cup SPAM");
  soupIngredients.add("1/4 cup taro chips");
 }
}

class HonoluluFishChowder extends FishChowder {
 public HonoluluFishChowder() {
  soupName = "OpakapakaFishChowder";
  soupIngredients.clear();
  soupIngredients.add("1 Pound Fresh Opakapaka Fish");
  soupIngredients.add("1 cup pineapple chunks");
  soupIngredients.add("1/2 cup coconut milk");
  soupIngredients.add("1/4 cup SPAM");
  soupIngredients.add("1/4 cup taro chips");
 }
}
```

Abstract Factory

- TestAbstractSoupFactory.java //testing the abstract factory

```
import java.util.Calendar;
class TestAbstractSoupFactory {
 public static Soup MakeSoupOfTheDay(AbstractSoupFactory concreteSoupFactory)
 {
   Calendar todayCalendar = Calendar.getInstance();
   int dayOfWeek = todayCalendar.get(Calendar.DAY_OF_WEEK);

   if (dayOfWeek == Calendar.MONDAY) {
     return concreteSoupFactory.makeChickenSoup();}
   else if (dayOfWeek == Calendar.TUESDAY) {
     return concreteSoupFactory.makeClamChowder();}
   else if (dayOfWeek == Calendar.WEDNESDAY) {
     return concreteSoupFactory.makeFishChowder();}
   else if (dayOfWeek == Calendar.THURSDAY) {
     return concreteSoupFactory.makeMinnestrone();}
```

Abstract Factory

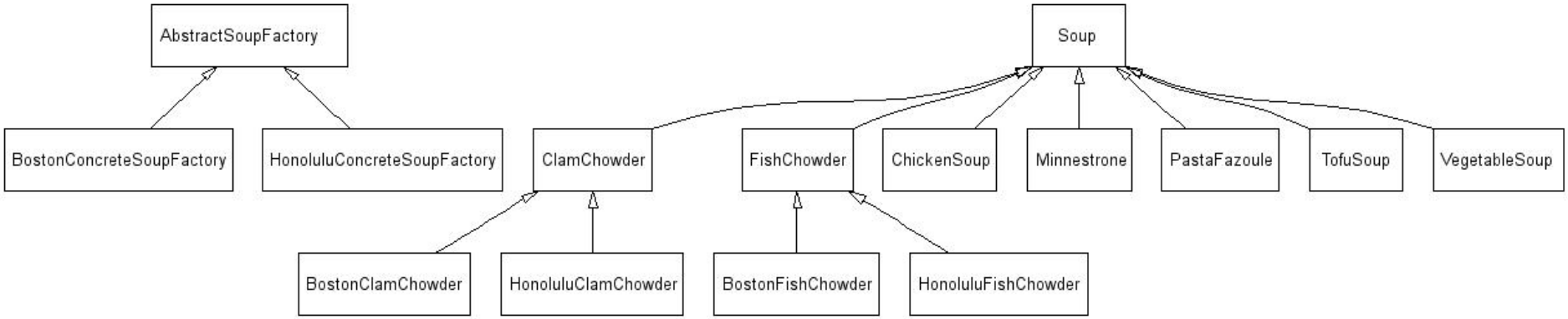

```
    else if (dayOfWeek == Calendar.TUESDAY) {
      return concreteSoupFactory.makePastafazul();}
    else if (dayOfWeek == Calendar.WEDNESDAY) {
      return concreteSoupFactory.makeTofuSoup();}
    else {
      return concreteSoupFactory.makeVegetableSoup();}
  }
  public static void main(String[] args) {
    AbstractSoupFactory concreteSoupFactory = new BostonConcreteSoupFactory();
    Soup soupOfTheDay = MakeSoupOfTheDay(concreteSoupFactory);
    System.out.println("The Soup of the day in " + concreteSoupFactory.getFactoryLocation() + " is " +
          soupOfTheDay.getSoupName());
    concreteSoupFactory = new HonoluluConcreteSoupFactory();
    soupOfTheDay = MakeSoupOfTheDay(concreteSoupFactory);
    System.out.println("The Soup of the day in " + concreteSoupFactory.getFactoryLocation() + " is " +
          soupOfTheDay.getSoupName());
  }
}
```

Abstract Factory

- Test Results (if run on a Tuesday)

The Soup of the day in Boston is QuahogChowder

The Soup of the day in Honolulu is PacificClamChowder

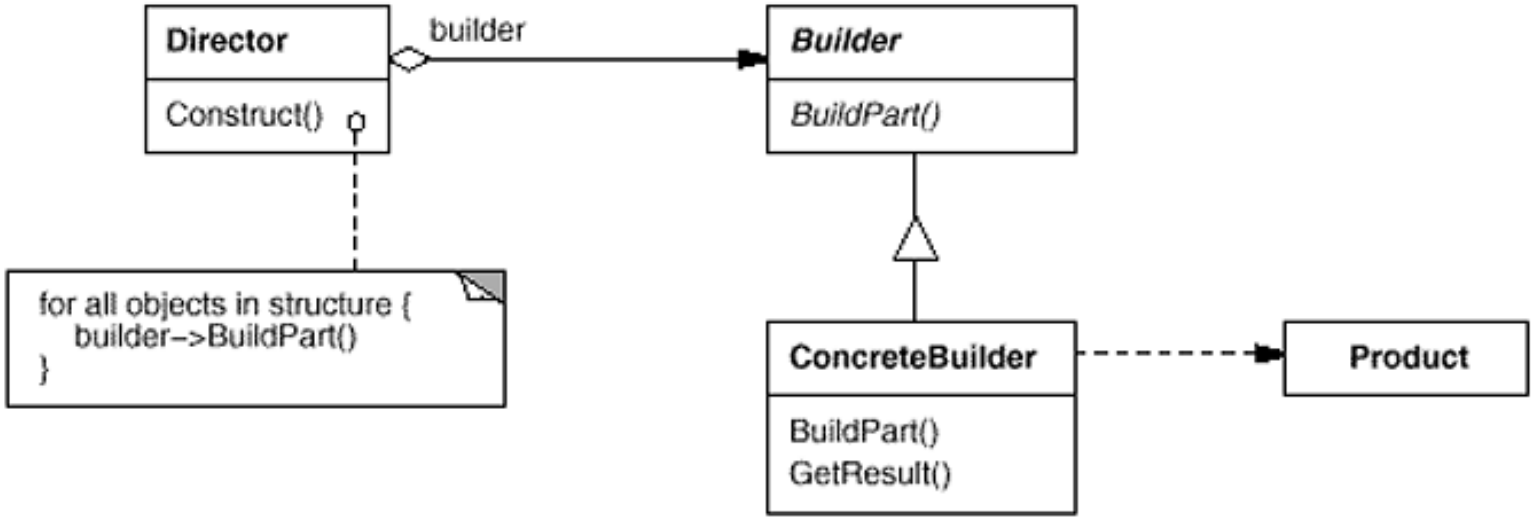


Builder

- Intent
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Builder

- Structure



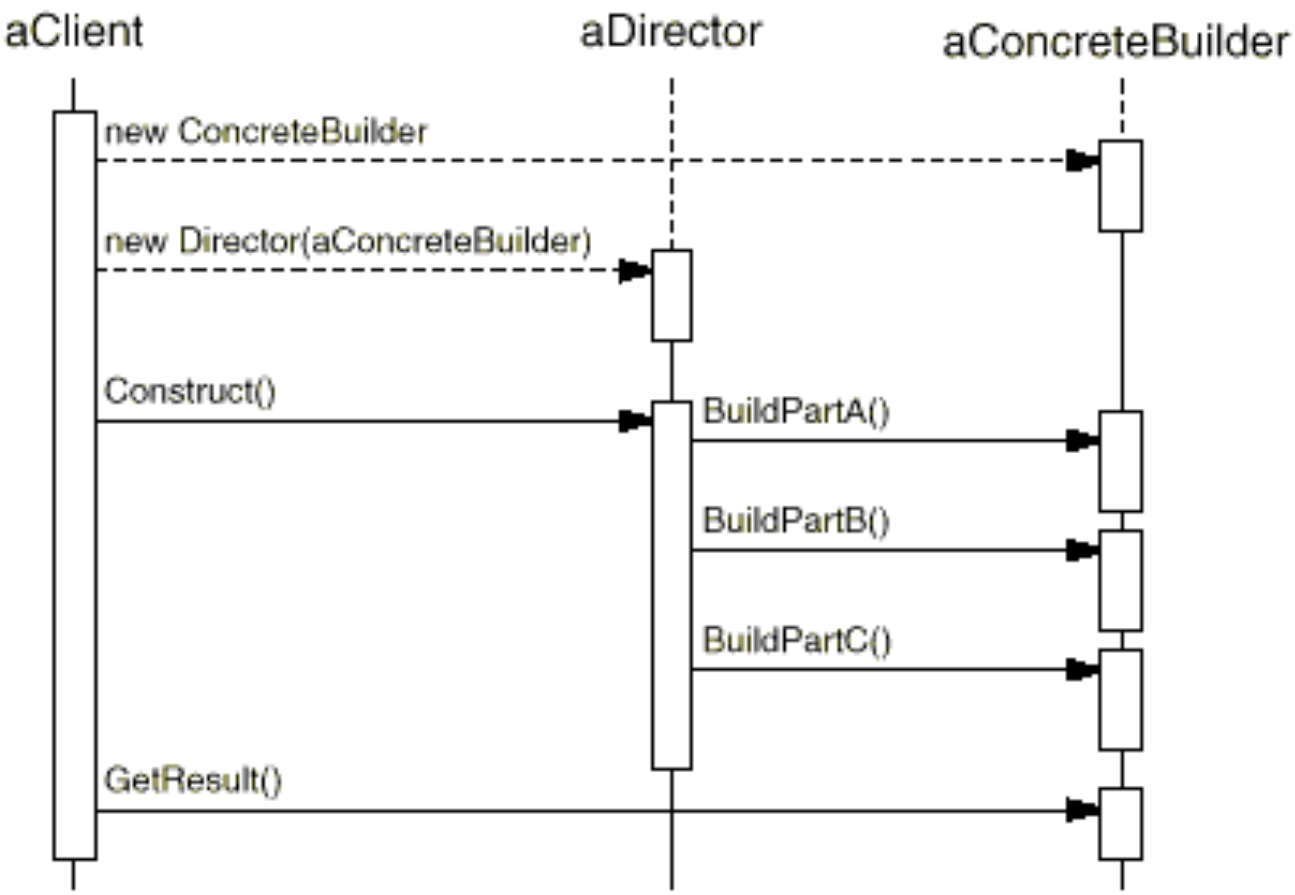
Builder

- Applicability

- Use the Builder pattern when
  - the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
  - the construction process must allow different representations for the object that's constructed.

Builder

- Collaborations



Builder

- Consequences

- Here are key consequences of the Builder pattern:
  - It lets you vary a product's internal representation.
  - It isolates code for construction and representation.
  - It gives you finer control over the construction process.

Builder

- Implementation
  - Typically there's an abstract Builder class that defines an operation for each component that a director may ask it to create.
  - A ConcreteBuilder class overrides operations for components it's interested in creating.

- Here are other implementation issues to consider:
  - Assembly and construction interface.
  - Why no abstract class for products?
  - Empty methods as default in Builder.

Builder

- SoupBuffet.java //helper class

```
class SoupBuffet {
  String soupBuffetName;
  ChickenSoup chickenSoup;
  ClamChowder clamChowder;
  FishChowder fishChowder;
  Minnestrone minnestrone;
  Pastafazul pastafazul;
  TofuSoup tofuSoup;
  VegetableSoup vegetableSoup;

  public String getSoupBuffetName() { return soupBuffetName; }
  public String getTodaysSoups() {
    StringBuffer stringOfSoups = new StringBuffer();
    stringOfSoups.append(" Today's Soups! ");
    stringOfSoups.append(" Chicken Soup: ");
    stringOfSoups.append(this.chickenSoup.getSoupName());
    stringOfSoups.append(" Clam Chowder: ");
```

Builder


```
    stringOfSoups.append(this.clamChowder.getSoupName());
    stringOfSoups.append(" Fish Chowder: ");
    stringOfSoups.append(this.fishChowder.getSoupName());
    stringOfSoups.append(" Minnestrone: ");
    stringOfSoups.append(this.minnestrone.getSoupName());
    stringOfSoups.append(" Pasta fazul: ");
    stringOfSoups.append(this.pastafazul.getSoupName());
    stringOfSoups.append(" Tofu Soup: ");
    stringOfSoups.append(this.tofuSoup.getSoupName());
    stringOfSoups.append(" Vegetable Soup: ");
    stringOfSoups.append(this.vegetableSoup.getSoupName());
    return stringOfSoups.toString();
  }
}
```

Builder

- SoupBuffetBuilder.java

```
//a Builder
abstract class SoupBuffetBuilder {
 SoupBuffet soupBuffet;
 public SoupBuffet getSoupBuffet() {return soupBuffet;}
 public void buildSoupBuffet() {soupBuffet = new SoupBuffet();}
 public abstract void setSoupBuffetName();
 public void buildChickenSoup() { soupBuffet.chickenSoup = new ChickenSoup(); }
 public void buildClamChowder() { soupBuffet.clamChowder = new ClamChowder(); }
 public void buildFishChowder() { soupBuffet.fishChowder = new FishChowder(); }
 public void buildMinnestrone() { soupBuffet.minnestrone = new Minnestrone(); }
 public void buildPastafazul() { soupBuffet.pastafazul = new Pastafazul(); }
 public void buildTofuSoup() { soupBuffet.tofuSoup = new TofuSoup(); }
 public void buildVegetableSoup() { soupBuffet.vegetableSoup = new VegetableSoup(); }
}
```

Builder

- BostonSoupBuffetBuilder.java
  //One of Two Builder Subclasses

```
class BostonSoupBuffetBuilder extends SoupBuffetBuilder {
 public void buildClamChowder() {
  soupBuffet.clamChowder = new BostonClamChowder();
 }
 public void buildFishChowder() {
  soupBuffet.fishChowder = new BostonFishChowder();
 }
 public void setSoupBuffetName() {
  soupBuffet.soupBuffetName = "Boston Soup Buffet";
 }
}
```

Builder

- HonoluluSoupBuffetBuilder.java

```
//Two of Two Builder Subclasses
class HonoluluSoupBuffetBuilder extends SoupBuffetBuilder {
 public void buildClamChowder() {
  soupBuffet.clamChowder = new HonoluluClamChowder();
 }
 public void buildFishChowder() {
  soupBuffet.fishChowder = new HonoluluFishChowder();
 }
 public void setSoupBuffetName() {
  soupBuffet.soupBuffetName = "Honolulu Soup Buffet";
 }
}
```

Builder

- TestSoupBuffetBuilder.java
  //testing the builder

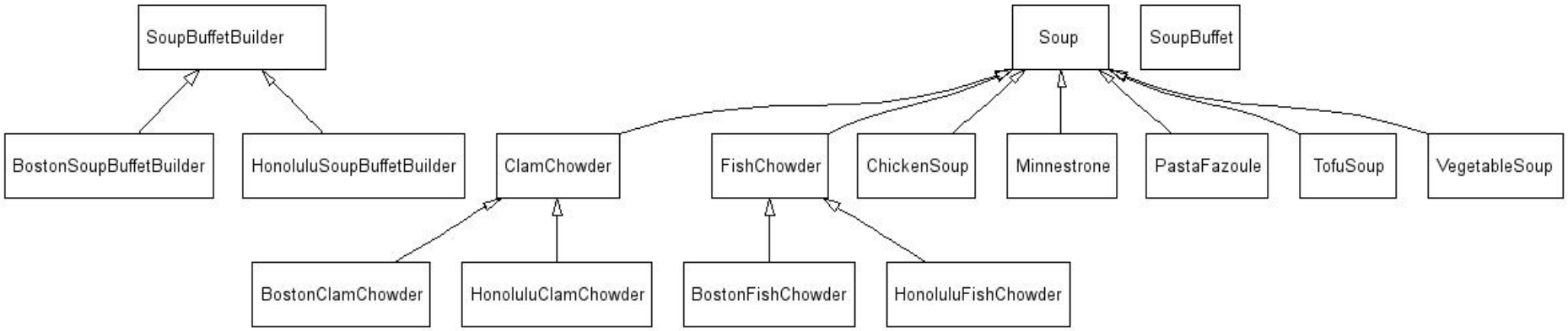
```
class TestSoupBuffetBuilder {
  public static SoupBuffet CreateSoupBuffet(SoupBuffetBuilder soupBuffetBuilder) {
    soupBuffetBuilder.buildSoupBuffet();
    soupBuffetBuilder.setSoupBuffetName();
    soupBuffetBuilder.buildChickenSoup();
    soupBuffetBuilder.buildClamChowder();
    soupBuffetBuilder.buildFishChowder();
    soupBuffetBuilder.buildMinnestrone();
    soupBuffetBuilder.buildPastafazul();
    soupBuffetBuilder.buildTofuSoup();
    soupBuffetBuilder.buildVegetableSoup();
    return soupBuffetBuilder.getSoupBuffet();
  }
```

Builder


```
public static void main(String[] args) {
    SoupBuffet bostonSoupBuffet = CreateSoupBuffet(new
                                                    BostonSoupBuffetBuilder());
    System.out.println("At the " + bostonSoupBuffet.getSoupBuffetName() +
                        bostonSoupBuffet.getTodaysSoups());
    SoupBuffet honoluluSoupBuffet = CreateSoupBuffet(new
                                                    HonoluluSoupBuffetBuilder());
    System.out.println("At the " + honoluluSoupBuffet.getSoupBuffetName() +
                        honoluluSoupBuffet.getTodaysSoups());
}
}
```

- Test Results
- At the Boston Soup Buffet Today's Soups! Chicken Soup: ChickenSoup Clam Chowder: QuahogChowder Fish Chowder: ScrodFishChowder Minnestrone: Minestrone Pasta fazul: Pasta fazul Tofu Soup: Tofu Soup Vegetable Soup: Vegetable Soup
- At the Honolulu Soup Buffet Today's Soups! Chicken Soup: ChickenSoup Clam Chowder: PacificClamChowder Fish Chowder: OpakapakaFishChowder Minnestrone: Minestrone Pasta fazul: Pasta fazul Tofu Soup: Tofu Soup Vegetable Soup: Vegetable Soup

Builder

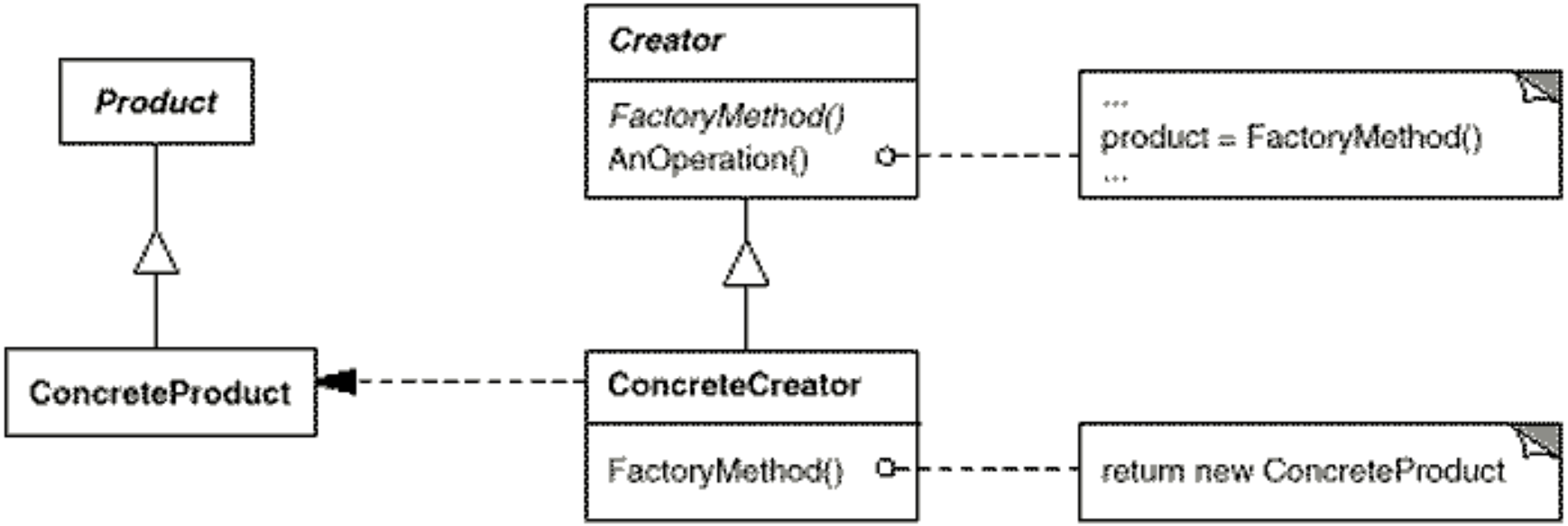

SoupBuffetBuilder
Soup
SoupBuffet
BostonSoupBuffetBuilder
HonoluluSoupBuffetBuilder
ClamChowder
FishChowder
ChickenSoup
Minnestrone
PastaFazoule
TofuSoup
VegetableSoup
BostonClamChowder
HonoluluClamChowder
BostonFishChowder
HonoluluFishChowder

Factory Method

- Intent
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Factory Method

- Structure



Factory Method

- Applicability

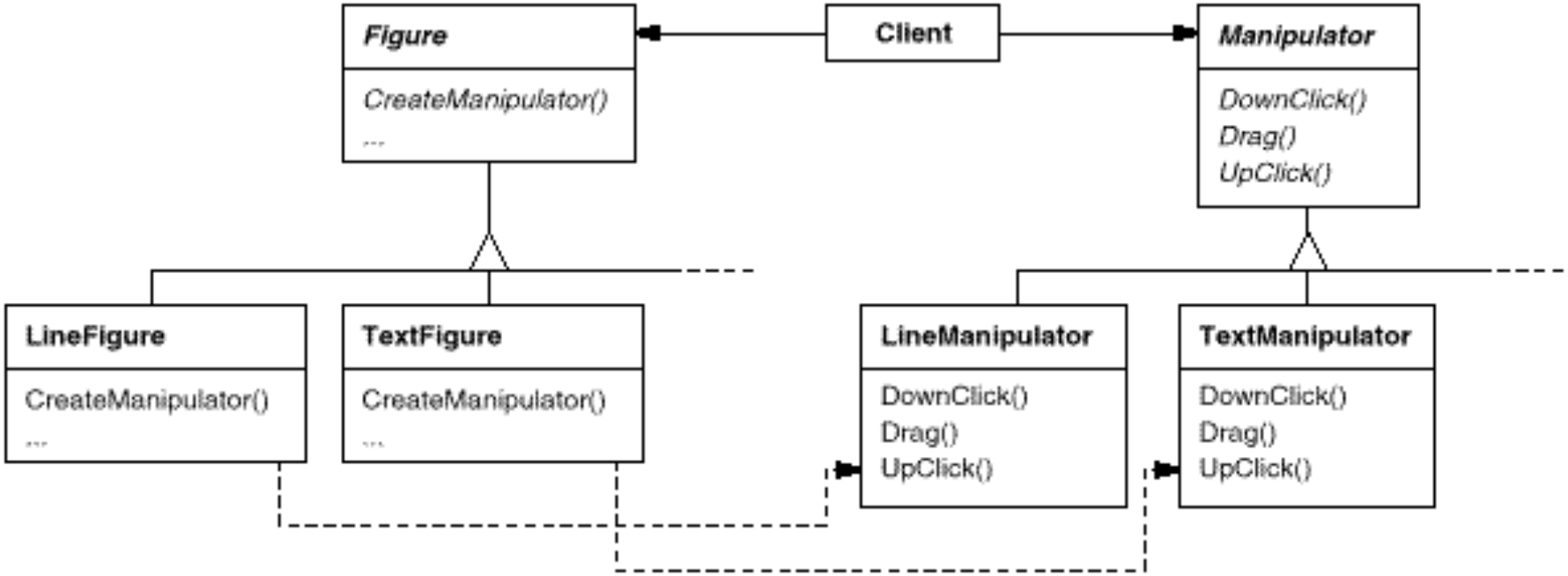
- Use the Factory Method pattern when
  - a class can't anticipate the class of objects it must create.
  - a class wants its subclasses to specify the objects it creates.
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Factory Method

- Consequences
  - Factory methods eliminate the need to bind application-specific classes into your code.
  - A potential disadvantage of factory methods is that clients might have to subclass the Creator class just to create a particular ConcreteProduct object.

- Here are two additional consequences of the Factory Method pattern:
  - Provides hooks for subclasses.
  - Connects parallel class hierarchies.

Factory Method



Factory Method

- Implementation

- Consider the following issues when applying the Factory Method pattern:
  - Two major varieties.
  - Parameterized factory methods.
  - Language-specific variants and issues.
    - Different languages lend themselves to other interesting variations and caveats.
  - Using templates to avoid subclassing.
  - Naming conventions.

Factory Method

- SoupFactoryMethod.java
  //a Factory Method

```
class SoupFactoryMethod {
 public SoupFactoryMethod() {}
 public SoupBuffet makeSoupBuffet() {return new SoupBuffet();}
 public ChickenSoup makeChickenSoup() {return new ChickenSoup();}
 public ClamChowder makeClamChowder() {return new ClamChowder();}
 public FishChowder makeFishChowder() {return new FishChowder();}
 public Minnestrone makeMinnestrone() {return new Minnestrone();}
 public Pastafazul makePastafazul() {return new Pastafazul();}
 public TofuSoup makeTofuSoup() {return new TofuSoup();}
 public VegetableSoup makeVegetableSoup() {return new VegetableSoup();}
 public String makeBuffetName() {return "Soup Buffet";}
}
```

Factory Method

- BostonSoupFactoryMethodSubclass.java
  //One of Two Subclass Factory Methods

```
class BostonSoupFactoryMethodSubclass extends SoupFactoryMethod {
 public String makeBuffetName() {return "Boston Soup Buffet";}
 public ClamChowder makeClamChowder() {return new BostonClamChowder();}
 public FishChowder makeFishChowder() {return new BostonFishChowder();}
}
```

- HonoluluSoupFactoryMethodSubclass.java
  //Two of Two Subclass Factory Methods

```
class HonoluluSoupFactoryMethodSubclass extends SoupFactoryMethod {
 public String makeBuffetName() {return "Honolulu Soup Buffet";}
 public ClamChowder makeClamChowder() {return new HonoluluClamChowder();}
 public FishChowder makeFishChowder() {return new HonoluluFishChowder();}
}
```

Factory Method

- TestAbstractSoupFactory.java
  testing the abstract factory

```
class TestSoupFactoryMethod {
 public static void main(String[] args) {
   SoupFactoryMethod soupFactoryMethod = new SoupFactoryMethod();
   SoupBuffet soupBuffet = soupFactoryMethod.makeSoupBuffet();
   soupBuffet.setSoupBuffetName(soupFactoryMethod.makeBuffetName());
   soupBuffet.setChickenSoup(soupFactoryMethod.makeChickenSoup());
   soupBuffet.setClamChowder(soupFactoryMethod.makeClamChowder());
   soupBuffet.setFishChowder(soupFactoryMethod.makeFishChowder());
   soupBuffet.setMinnestrone(soupFactoryMethod.makeMinnestrone());
   soupBuffet.setPastafazul(soupFactoryMethod.makePastafazul());
   soupBuffet.setTofuSoup(soupFactoryMethod.makeTofuSoup());
   soupBuffet.setVegetableSoup(soupFactoryMethod.makeVegetableSoup());
   System.out.println("At the " + soupBuffet.getSoupBuffetName() +
     soupBuffet.getTodaysSoups());
```

Factory Method

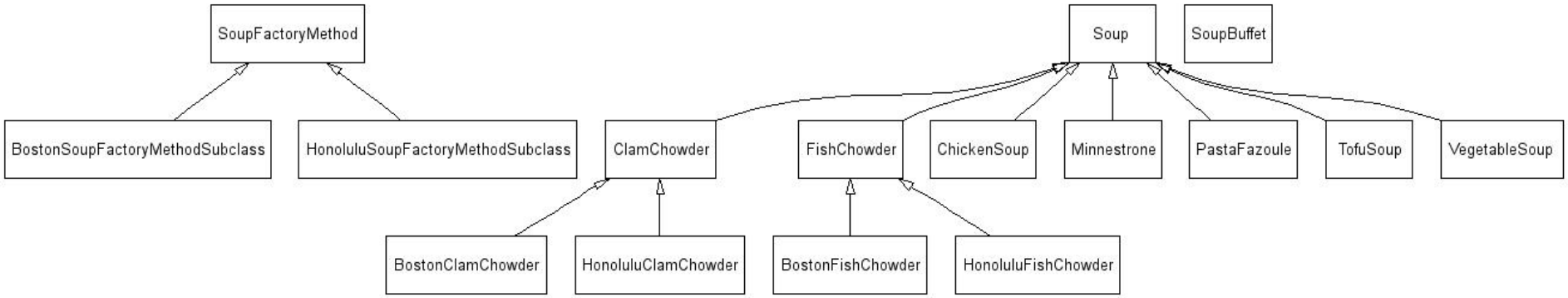

```
SoupFactoryMethod bostonSoupFactoryMethod = new BostonSoupFactoryMethodSubclass();
SoupBuffet bostonSoupBuffet = bostonSoupFactoryMethod.makeSoupBuffet();
bostonSoupBuffet.setSoupBuffetName(bostonSoupFactoryMethod.makeBuffetName());
bostonSoupBuffet.setChickenSoup(bostonSoupFactoryMethod.makeChickenSoup());
bostonSoupBuffet.setClamChowder(bostonSoupFactoryMethod.makeClamChowder());
bostonSoupBuffet.setFishChowder(bostonSoupFactoryMethod.makeFishChowder());
bostonSoupBuffet.setMinnestrone(bostonSoupFactoryMethod.makeMinnestrone());
bostonSoupBuffet.setPastafazul(bostonSoupFactoryMethod.makePastafazul());
bostonSoupBuffet.setTofuSoup(bostonSoupFactoryMethod.makeTofuSoup());
bostonSoupBuffet.setVegetableSoup(bostonSoupFactoryMethod.makeVegetableSoup());
System.out.println("At the " + bostonSoupBuffet.getSoupBuffetName() +
    bostonSoupBuffet.getTodaysSoups());

SoupFactoryMethod honoluluSoupFactoryMethod = new HonoluluSoupFactoryMethodSubclass();
SoupBuffet honoluluSoupBuffet = honoluluSoupFactoryMethod.makeSoupBuffet();
```

Factory Method


```
honoluluSoupBuffet.setSoupBuffetName(honoluluSoupFactoryMethod.makeBuffetName());
honoluluSoupBuffet.setChickenSoup(honoluluSoupFactoryMethod.makeChickenSoup());
honoluluSoupBuffet.setClamChowder(honoluluSoupFactoryMethod.makeClamChowder());
honoluluSoupBuffet.setFishChowder(honoluluSoupFactoryMethod.makeFishChowder());
honoluluSoupBuffet.setMinnestrone(honoluluSoupFactoryMethod.makeMinnestrone());
honoluluSoupBuffet.setPastafazul(honoluluSoupFactoryMethod.makePastafazul());
honoluluSoupBuffet.setTofuSoup(honoluluSoupFactoryMethod.makeTofuSoup());
honoluluSoupBuffet.setVegetableSoup
            (honoluluSoupFactoryMethod.makeVegetableSoup());
System.out.println("At the " + honoluluSoupBuffet.getSoupBuffetName() +
    honoluluSoupBuffet.getTodaysSoups());
}
}
```

- Test Results
  - ……

Factory Method

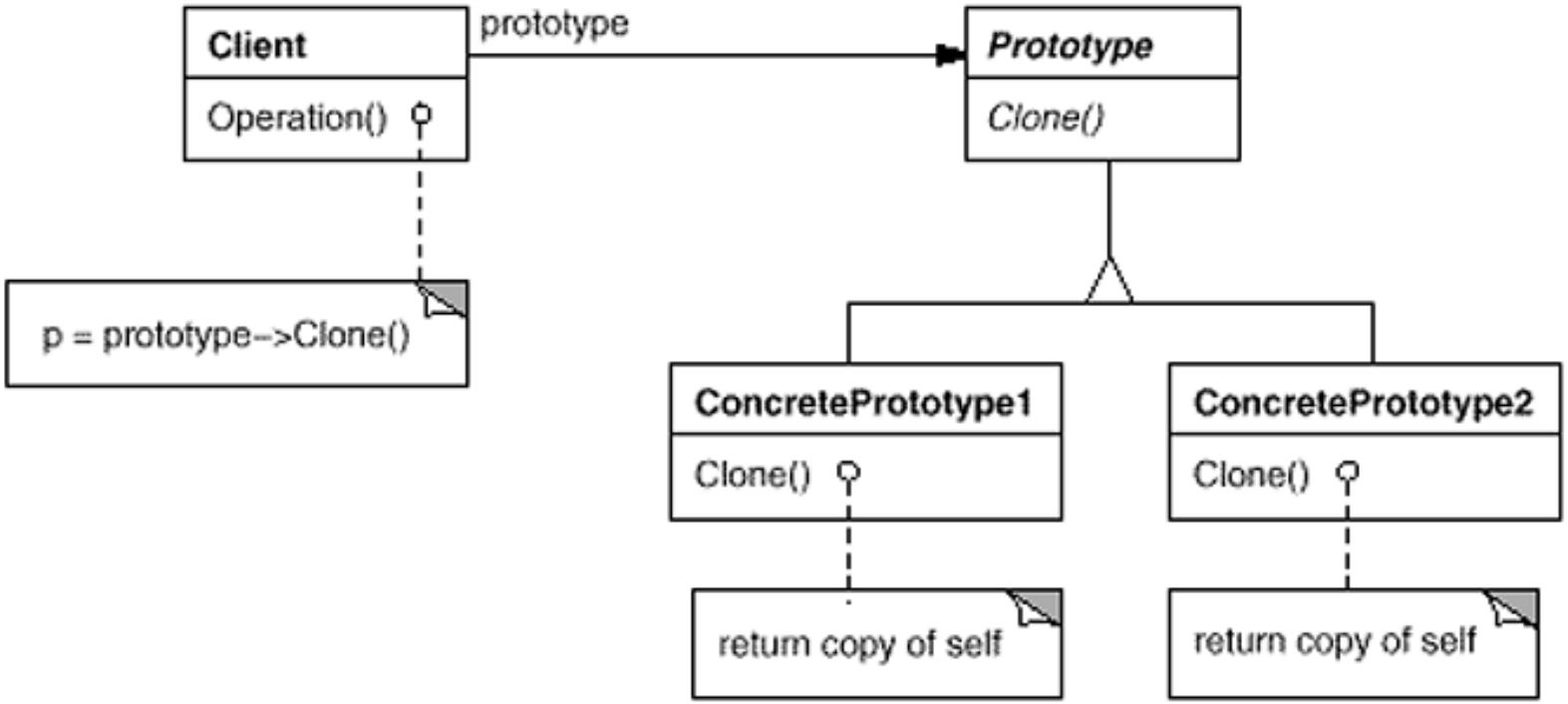


Prototype

- Intent
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Prototype

- Structure



Prototype

- Applicability

- Use the Prototype pattern
  - when a system should be independent of how its products are created, composed, and represented; *and*
  - when the classes to instantiate are specified at run-time, for example, by dynamic loading; *or*
  - to avoid building a class hierarchy of factories that parallels the class hierarchy of products; *or*
  - when instances of a class can have one of only a few different combinations of state..

Prototype

- Consequences

- Prototype
  - Hiding the concrete product classes from the client, thereby reducing the number of names clients know about.
  - Adding and removing products at run-time.
  - Specifying new objects by varying values.
  - Specifying new objects by varying structure.
  - Reduced subclassing.
  - Configuring an application with classes dynamically.

Prototype

- Implementation

- Consider the following issues when implementing prototypes:
  - Using a prototype manager.
  - Implementing the Clone operation.
  - Initializing clones.

Prototype

- PrototypeFactory.java

```
//a Factory for Prototypes
public class PrototypeFactory {
 AbstractSpoon prototypeSpoon;
 AbstractFork prototypeFork;
 public PrototypeFactory(AbstractSpoon spoon, AbstractFork fork)
 {
    prototypeSpoon = spoon;
    prototypeFork = fork;
 }
 public AbstractSpoon makeSpoon() {
    return (AbstractSpoon)prototypeSpoon.clone();
 }
 public AbstractFork makeFork() {
    return (AbstractFork)prototypeFork.clone();
 }
}
```

Prototype

- AbstractSpoon.java

```
//One of Two Prototypes
public abstract class AbstractSpoon implements Cloneable {
 String spoonName;
 public void setSpoonName(String spoonName) {
  this.spoonName = spoonName;
 }
 public String getSpoonName() {
  return this.spoonName;
 }
 public Object clone() {
  Object object = null;
  try {
    object = super.clone();
  }catch (CloneNotSupportedException exception)
  { System.err.println("AbstractSpoon is not Cloneable"); }
  return object;
 }
}
```

Prototype

- AbstractFork.java

```
//Two of Two Prototypes
public abstract class AbstractFork implements Cloneable {
 String forkName;
 public void setForkName(String forkName) {
  this.forkName = forkName;
 }
 public String getForkName() {
  return this.forkName;
 }
 public Object clone() {
  Object object = null;
  try {
    object = super.clone();
  }catch (CloneNotSupportedException exception)
  { System.err.println("AbstractFork is not Cloneable"); }
  return object;
 }
}
```

Prototype

- SoupSpoon.java

```
//One of Two Concrete Prototypes extending the AbstractSpoon Prototype
public class SoupSpoon extends AbstractSpoon {
 public SoupSpoon() { setSpoonName("Soup Spoon"); }
}
```

- SaladSpoon.java

```
//Two of Two Concrete Prototypes extending the AbstractSpoon Prototype
public class SaladSpoon extends AbstractSpoon {
 public SaladSpoon() { setSpoonName("Salad Spoon"); }
}
```

- SaladFork.java

```
//The Concrete Prototype extending the AbstractFork Prototype
public class SaladFork extends AbstractFork {
 public SaladFork() { setForkName("Salad Fork"); }
}
```

Prototype

- TestPrototype.java

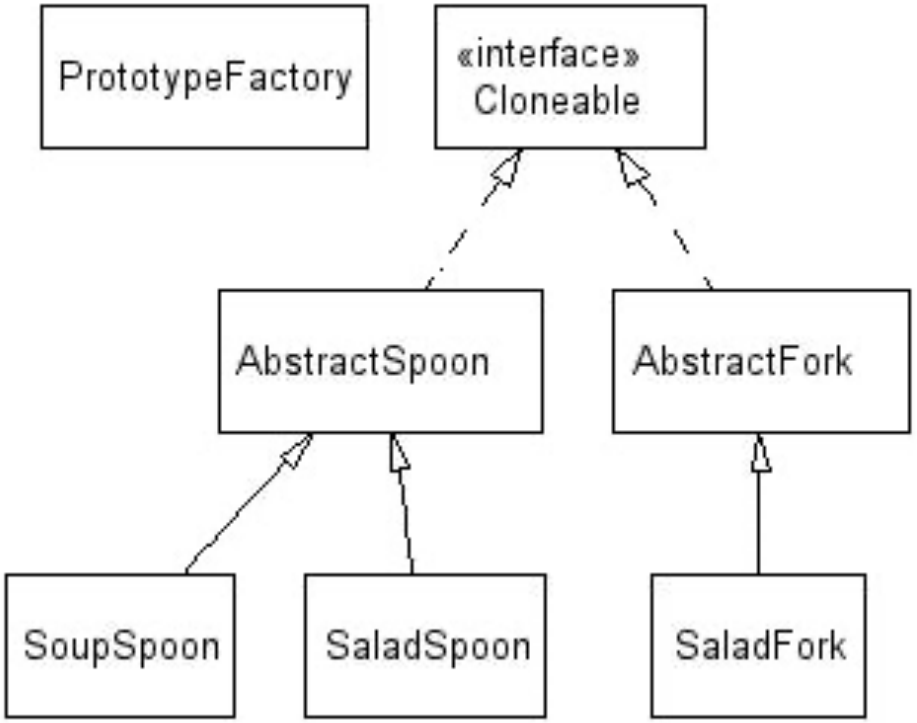
```
//testing the Prototype
class TestPrototype {
 public static void main(String[] args) {
  System.out.println( "Creating a Prototype Factory with a SoupSpoon and a SaladFork");
  PrototypeFactory prototypeFactory = new PrototypeFactory(new SoupSpoon(), new SaladFork());
  AbstractSpoon spoon = prototypeFactory.makeSpoon();
  AbstractFork fork = prototypeFactory.makeFork();
  System.out.println("Getting the Spoon and Fork name:");
  System.out.println("Spoon: " + spoon.getSpoonName() + ", Fork: " + fork.getForkName());
  System.out.println(" ");

  System.out.println( "Creating a Prototype Factory with a SaladSpoon and a SaladFork");
  prototypeFactory = new PrototypeFactory(new SaladSpoon(), new SaladFork());
```

Prototype


```
  spoon = prototypeFactory.makeSpoon();
  fork = prototypeFactory.makeFork();
  System.out.println("Getting the Spoon and Fork name:");
  System.out.println( "Spoon: " + spoon.getSpoonName() + ", Fork: " +
      fork.getForkName());
 }
}
```

- Test Results
- Creating a Prototype Factory with a SoupSpoon and a SaladFork
- Getting the Spoon and Fork name:
- Spoon: Soup Spoon, Fork: Salad Fork
- Creating a Prototype Factory with a SaladSpoon and a SaladFork
- Getting the Spoon and Fork name:
- Spoon: Salad Spoon, Fork: Salad Fork

Prototype

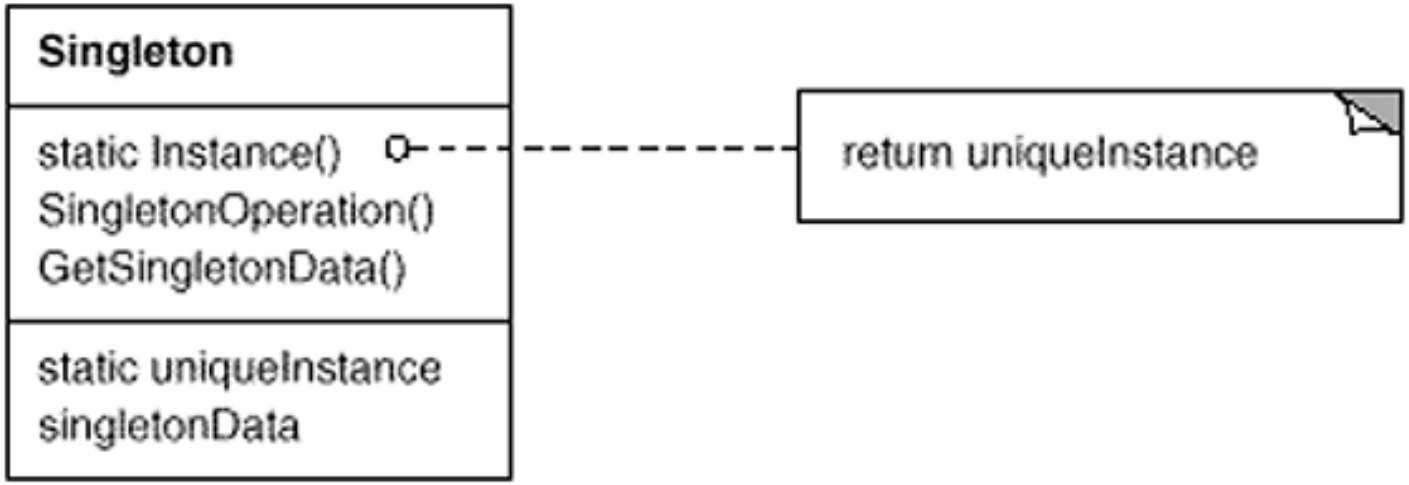

PrototypeFactory
«interface»
Cloneable
AbstractSpoon
AbstractFork
SoupSpoon
SaladSpoon
SaladFork

Singleton

- Intent
  - Ensure a class only has one instance, and provide a global point of access to it.

Singleton

- Structure



Singleton

- Applicability

- Use the Singleton pattern when
  - there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
  - when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Singleton

- Consequences

- The Singleton pattern has several benefits:
  - Controlled access to sole instance.
  - Reduced name space.
  - Permits refinement of operations and representation.
  - Permits a variable number of instances.
  - More flexible than class operations.

Singleton

- Implementation

- Here are implementation issues to consider when using the Singleton pattern:
  - Ensuring a unique instance.
  - Subclassing the Singleton class.
    - A more flexible approach uses a registry of singletons.

Singleton

- SingleSpoon.java //a Singleton

```
public class SingleSpoon {
 private Soup soupLastUsedWith;
 private static SingleSpoon theSpoon;
 private static boolean theSpoonIsAvailable = true;

 private SingleSpoon() {}

 public static SingleSpoon getTheSpoon() {
  if (theSpoonIsAvailable) {
    if (theSpoon == null) { theSpoon = new SingleSpoon(); }
    theSpoonIsAvailable = false;
    return theSpoon;
   } else {
    return null;
   }
 }
```

Singleton


```
public String toString() { return "Behold the glorious single spoon!"; }

public static void returnTheSpoon() {
  theSpoon.cleanSpoon();
  theSpoonIsAvailable = true;
}

public Soup getSoupLastUsedWith() { return this.soupLastUsedWith; }

public void setSoupLastUsedWith(Soup soup) { this.soupLastUsedWith = soup; }

public void cleanSpoon() { this.setSoupLastUsedWith(null); }

}
```

Singleton

- TestSingleSpoon.java： //testing the singleton

```
class TestSingleSpoon {
  public static void main(String[] args) {
    System.out.println("First person getting the spoon");
    SingleSpoon spoonForFirstPerson =
        SingleSpoon.getTheSpoon();
    if (spoonForFirstPerson != null)
      System.out.println(spoonForFirstPerson);
    else
      System.out.println("No spoon was available");
    System.out.println("");

    System.out.println("Second person getting a spoon");
    SingleSpoon spoonForSecondPerson = SingleSpoon.getTheSpoon();
    if (spoonForSecondPerson != null)
      System.out.println(spoonForSecondPerson);
    else
      System.out.println("No spoon was available");
    System.out.println("");
```

Singleton


```
    System.out.println("First person returning the spoon");
    spoonForFirstPerson.returnTheSpoon();
    spoonForFirstPerson = null;
    System.out.println("The spoon was returned");
    System.out.println("");

    System.out.println("Second person getting a spoon");
    spoonForSecondPerson = SingleSpoon.getTheSpoon();
    if (spoonForSecondPerson != null)
       System.out.println(spoonForSecondPerson);
    else
       System.out.println("No spoon was available");
  }
}
```

Singleton

- Test Results

First person getting the spoon
Behold the glorious single spoon!

Second person getting a spoon
No spoon was available

First person returning the spoon
The spoon was returned

Second person getting a spoon
Behold the glorious single spoon!

- *Web*开发技术
- *Web Application Development*

Thank You!