Patrones de Diseño Creacionales

Fundamentos de Ingeniería de Software/Análisis y Diseño de Software

Pablo Cruz Navea-Gastón Márquez
Departamento de Informática
Universidad Técnica Federico Santa María

Patrones de Diseño [1]

- En el diseño de un sistema siempre encontramos problemas que son similares a otros antes ya vistos
- Es posible, entonces, generar un cuerpo de conocimiento que permita solucionar genéricamente grupos de problemas que son recurrentes en el diseño de software
- Los patrones de diseño constituyen este cuerpo de conocimiento:
  - Soluciones genéricas
  - Para problemas recurrentes en el diseño de software
- En general, los patrones de diseño se caracterizan por:
  - Descripción del problema
  - Descripción de la solución propuesta

Patrones de Diseño [2]

- Existen algunas clasificaciones de patrones de diseño:
  - Patrones de diseño creacionales:
    - Problemas de instanciación de clases (creación de objetos)
  - Patrones de diseño estructurales:
    - Problemas de “composición y herencia”
  - Patrones de diseño de comportamiento:
    - Problemas de comunicación entre objetos

Abstract Factory

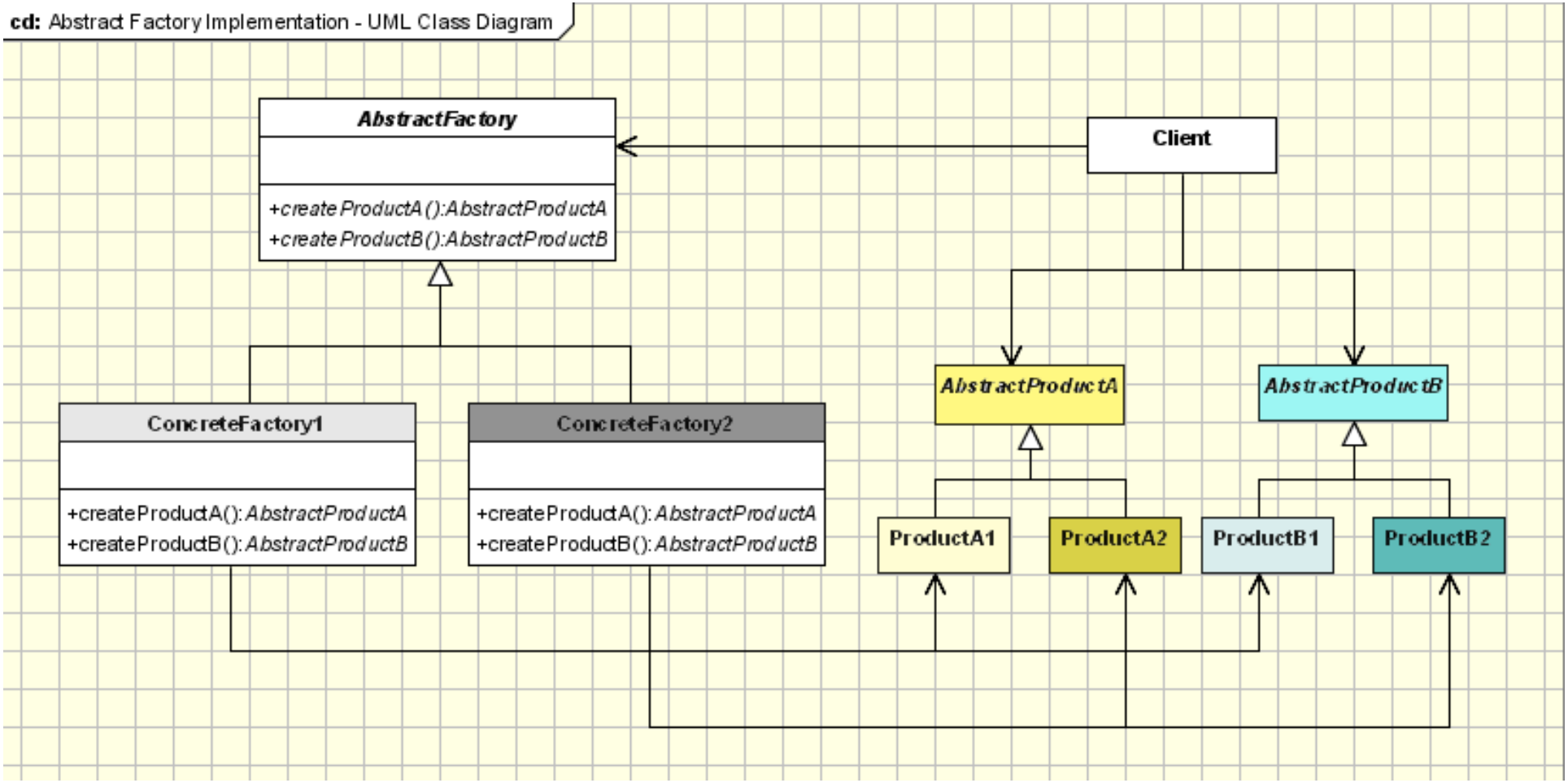
Abstract Factory [1]

- Patrón de diseño **creacional**
- Propósito: proveer una interfaz para crear familias de objetos dependientes sin especificar (al cliente) sus clases concretas
- Una clase abstracta provee los métodos (a modo de interfaz) que son comunes para todos los tipos de productos
- Varias clases (que representan los tipos de productos) heredan los métodos desde la clase abstracta

Abstract Factory [2]

- La instanciación de la clase puede ser entendida como “crear un producto en una fábrica”
  - Esto es, entenderemos al objeto como un producto
- **Idea central:** el cliente, en vez de crear directamente el objeto utilizando el operador adecuado (ej: en JAVA el operador `new`), entrega información a la **fábrica** sobre el tipo de **producto** que necesita
  - La fábrica instancia una nueva clase (crea el objeto/ producto) y lo devuelve al cliente
  - El cliente utiliza al objeto como un “producto abstracto” sin interesarse en la implementación concreta

Abstract Factory [3]



Abstract Factory [4]

- Las clases que participan en el patrón son las siguientes:
  - AbstractFactory: declara una interface para los operadores que crean productos abstractos
  - ConcreteFactory: implementa las operaciones para crear productos concretos
  - AbstractProduct: declara la interface para un tipo de producto objeto
  - Product: define un producto a ser creado por el correspondiente ConcreteFactory, es implementado por la interface AbstractProduct
  - Client: usa la interface declarada por la clases AbstractFactory y AbstractProduct

Abstract Factory [5]

- El hecho de que la fábrica devuelve una referencia abstracta al objeto creado significa que el cliente no tiene conocimiento del tipo de objeto.
- Una consecuencia de lo anterior es que cuando se necesitan nuevos tipos concretos de objetos, todo lo que tenemos que hacer es modificar el código de cliente y hacer uso de una fábrica diferente, que es mucho más fácil que crear instancias de un nuevo tipo.

Abstract Factory [6]

- La clase AbstractFactory es la que determina el tipo real del objeto concreto y lo crea, pero devuelve una referencia abstracta al objeto concreto recién creado.
- Esto determina el comportamiento del cliente que pide a la fábrica la creación de un objeto de un cierto tipo abstracto y para devolver la referencia a él, manteniendo el cliente de saber nada acerca de la creación real del objeto.

Abstract Factory [7]

```
abstract class AbstractProductA{
          public abstract void operationA1();
          public abstract void operationA2();
}

class ProductA1 extends AbstractProductA{
          ProductA1(String arg){
                    System.out.println("Hello "+arg);
          } // Implement the code here
          public void operationA1() { };
          public void operationA2() { };
}

class ProductA2 extends AbstractProductA{
          ProductA2(String arg){
                    System.out.println("Hello "+arg);
          } // Implement the code here
          public void operationA1() { };
          public void operationA2() { };
}

abstract class AbstractProductB{
          //public abstract void operationB1();
          //public abstract void operationB2();
};
```

Abstract Factory [7]

```
class ProductB1 extends AbstractProductB{
          ProductB1(String arg){
                    System.out.println("Hello "+arg);
          } // Implement the code here
}

class ProductB2 extends AbstractProductB{
          ProductB2(String arg){
                    System.out.println("Hello "+arg);
          } // Implement the code here
}

abstract class AbstractFactory{
          abstract AbstractProductA createProductA();
          abstract AbstractProductB createProductB();
}

class ConcreteFactory1 extends AbstractFactory{
          AbstractProductA createProductA(){
          return new ProductA1("ProductA1");
          }
          AbstractProductB createProductB(){
          return new ProductB1("ProductB1");
          }
}
```

Abstract Factory [7]

```
class ConcreteFactory2 extends AbstractFactory{
        AbstractProductA createProductA(){
                return new ProductA2("ProductA2");
        }
        AbstractProductB createProductB(){
                return new ProductB2("ProductB2");
        }
}

//Factory creator - an indirect way of instantiating the factories
class FactoryMaker{
        private static AbstractFactory pf=null;
        static AbstractFactory getFactory(String choice){
                if(choice.equals("a")){
                        pf=new ConcreteFactory1();
                }else if(choice.equals("b")){
                                pf=new ConcreteFactory2();
                        } return pf;
        }
}
```

Abstract Factory [7]

```
// Client
public class Client{
      public static void main(String args[]){
            AbstractFactory pf=FactoryMaker.getFactory("a");
            AbstractProductA product=pf.createProductA();
      //more function calls on product
      }
}
```

¿Cuándo ocupar Abstract Factory?

1. Cuando el proyecto en el cual estamos trabajando el sistema debe ser independiente a la forma en que los productos trabajan cuando son creados
2. Cuando el sistema debe ser configurado para trabajar con varias familias de productos
3. Cuando una familia de productos está diseñada para funcionar todos juntas
4. Cuando se necesita la creación de una librería de productos, donde es solo relevante la interface, pero no la implementación

Ejemplo Abstract Factory: Tienda de Pizza

- Un problema clásico de este patrón.
- Se necesita crear un sistema que separe los procesos de creación de una pizza del proceso de preparación/orden de una pizza

```
1  public class PizzaStore {
2
3      Pizza orderPizza(String type) {
4
5          Pizza pizza;
6
7          if (type.equals("cheese")) {
8              pizza = new CheesePizza();
9          } else if (type.equals("greek")) {
10             pizza = new GreekPizza();
11         } else if (type.equals("pepperoni")) {
12             pizza = new PepperoniPizza();
13         }
14
15         pizza.prepare();
16         pizza.bake();
17         pizza.cut();
18         pizza.box();
19
20         return pizza;
21     }
22
23 }
24
```

Creation

Creation code has all the same problems as the code earlier

Preparation

Note: excellent example of “coding to an interface”

```
1
2  public class NYPizzaStore extends PizzaStore {
3
4      Pizza createPizza(String item) {
5          if (item.equals("cheese")) {
6              return new NYStyleCheesePizza();
7          } else if (item.equals("veggie")) {
8              return new NYStyleVeggiePizza();
9          } else if (item.equals("clam")) {
10             return new NYStyleClamPizza();
11         } else if (item.equals("pepperoni")) {
12             return new NYStylePepperoniPizza();
13         } else return null;
14     }
15 }
16
```

```
1
2  public class ChicagoPizzaStore extends PizzaStore {
3
4      Pizza createPizza(String item) {
5              if (item.equals("cheese")) {
6                      return new ChicagoStyleCheesePizza();
7              } else if (item.equals("veggie")) {
8                      return new ChicagoStyleVeggiePizza();
9              } else if (item.equals("clam")) {
10                     return new ChicagoStyleClamPizza();
11             } else if (item.equals("pepperoni")) {
12                     return new ChicagoStylePepperoniPizza();
13             } else return null;
14     }
15 }
16
```

```
1
2  public class NYStylePepperoniPizza extends Pizza {
3
4      public NYStylePepperoniPizza() {
5          name = "NY Style Pepperoni Pizza";
6          dough = "Thin Crust Dough";
7          sauce = "Marinara Sauce";
8
9          toppings.add("Grated Reggiano Cheese");
10         toppings.add("Sliced Pepperoni");
11         toppings.add("Garlic");
12         toppings.add("Onion");
13         toppings.add("Mushrooms");
14         toppings.add("Red Pepper");
15     }
16 }
17
```

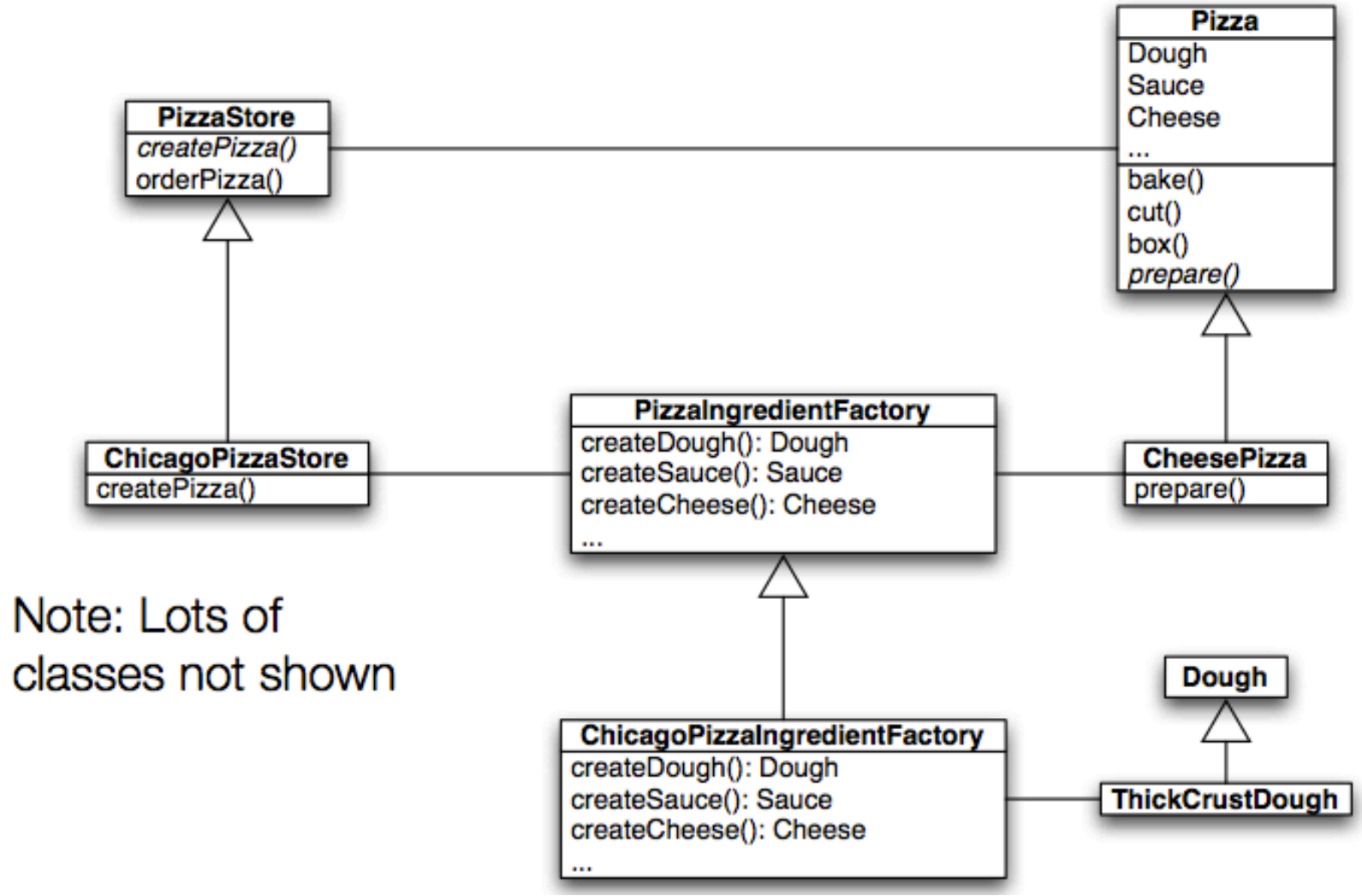
Un par de clases de productos concretos

```
1
2  public class ChicagoStylePepperoniPizza extends Pizza {
3
4      public ChicagoStylePepperoniPizza() {
5          name = "Chicago Style Pepperoni Pizza";
6          dough = "Extra Thick Crust Dough";
7          sauce = "Plum Tomato Sauce";
8
9          toppings.add("Shredded Mozzarella Cheese");
10         toppings.add("Black Olives");
11         toppings.add("Spinach");
12         toppings.add("Eggplant");
13         toppings.add("Sliced Pepperoni");
14     }
15
16     void cut() {
17         System.out.println("Cutting the pizza into square slices");
18     }
19 }
```

Ejemplo Abstract Factory: Tienda de Pizza

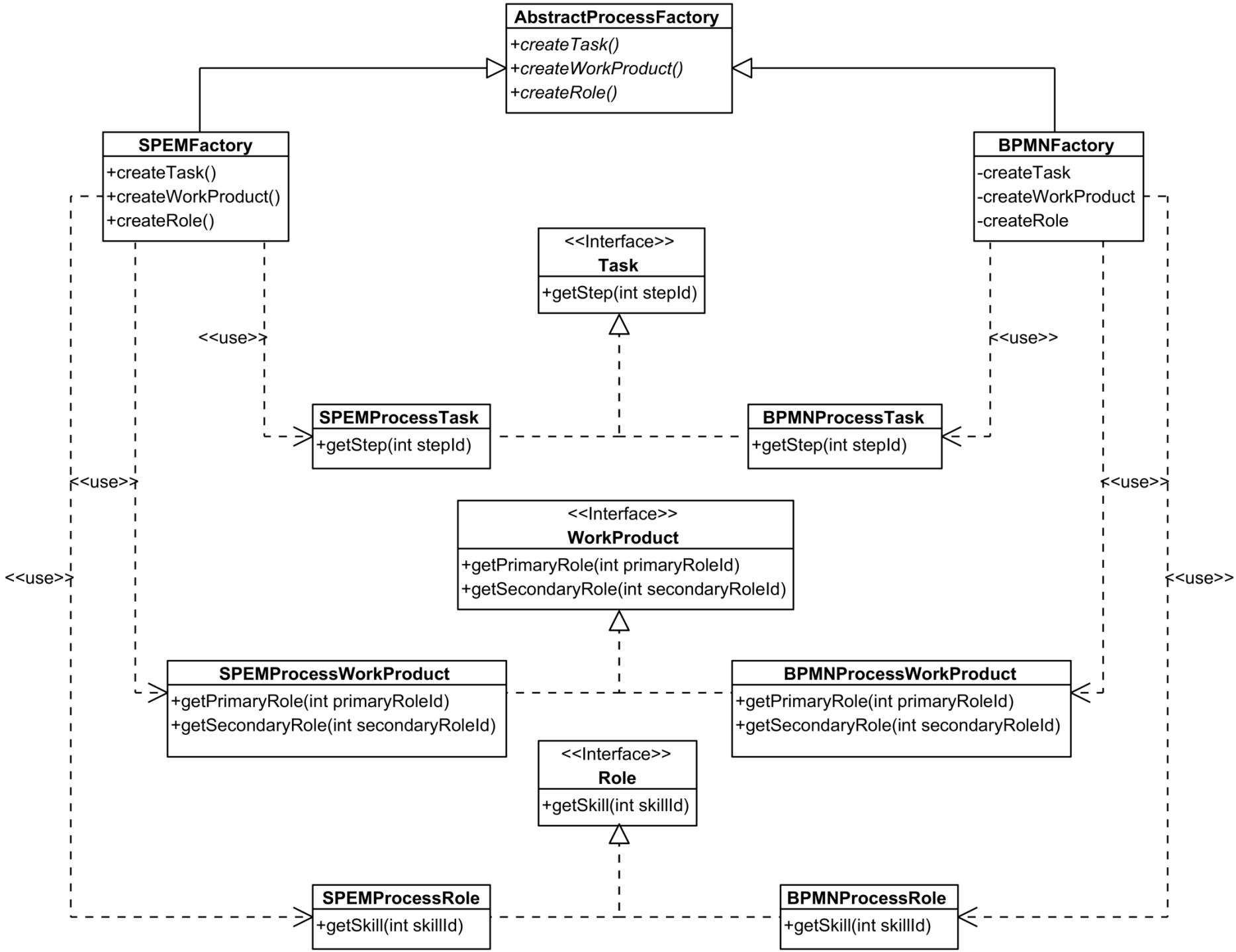
- Se debe cambiar el diseño tal que una fábrica suministre los ingredientes que se necesitan para la creación de la pizza
- De diferentes regiones se utilizan diferentes tipos de ingredientes
- Aseguramiento de la calidad
- Reducir costos

Ejemplo Abstract Factory: Tienda de Pizza



Ejemplo Abstract Factory: SPEM vs BPMN

- Problema: queremos diseñar un sistema que nos permita operar sobre los elementos básicos que constituyen procesos en SPEM y BPMN (notación para BPM)
  - Tareas, Productos de Trabajo, Roles
- Las implementaciones son distintas: SPEM Y BPMN son notaciones de la OMG, pero en términos concretos (implementación) son muy distintos
- Necesitamos el patrón Abstract Factory para crear "productos" SPEM y BPMN


AbstractProcessFactory
+createTask()
+createWorkProduct()
+createRole()
SPEMFactory
+createTask()
+createWorkProduct()
+createRole()
BPMNFactory
-createTask
-createWorkProduct
-createRole
<<Interface>>
Task
+getStep(int stepId)
<<use>>
<<use>>
SPEMProcessTask
+getStep(int stepId)
BPMNProcessTask
+getStep(int stepId)
<<use>>
<<use>>
<<Interface>>
WorkProduct
+getPrimaryRole(int primaryRoleId)
+getSecondaryRole(int secondaryRoleId)
<<use>>
<<use>>
SPEMProcessWorkProduct
+getPrimaryRole(int primaryRoleId)
+getSecondaryRole(int secondaryRoleId)
BPMNProcessWorkProduct
+getPrimaryRole(int primaryRoleId)
+getSecondaryRole(int secondaryRoleId)
<<Interface>>
Role
+getSkill(int skillId)
SPEMProcessRole
+getSkill(int skillId)
BPMNProcessRole
+getSkill(int skillId)

Interfaces: Task, WorkProduct, Role

```
public interface Task {
	public String getStep (int stepId);
}

public interface WorkProduct {
	public String getPrimaryRole (int primaryRoleId);
	public String getSecondaryRole (int secondaryRoleId);
}

public interface Role {
	public String getSkill (int skillId);
}
```

SPEM: clases Task, WorkProduct y Role

```
public SPEMProcessTask implements Task {
        public String getStep (int stepId) {
                return "SPEM Step";
        }
}

public SPEMProcessWorkProduct implements WorkProduct {
        public String getPrimaryRole (int primaryRoleId) {
                return "SPEM Primary Role";
        }
        public String getSecondaryRole (int secondaryRoleId) {
                return "SPEM Secondary Role";
}

public SPEMProcessRole implements Role {
        public String getSkill (int skillId) {
                return "SPEM Skill";
        }
}
```

BPMN: clases Task, WorkProduct y Role

```
public BPMNProcessTask implements Task {
        public String getStep (int stepId) {
                return "BPMN Step";
        }
}
public BPMNProcessWorkProduct implements WorkProduct {
        public String getPrimaryRole (int primaryRoleId) {
                return "BPMN Primary Role";
        }
        public String getSecondaryRole (int secondaryRoleId) {
                return "BPMN Secondary Role";
        }
}
public BPMNProcessRole implements Role {
        public String getSkill (int skillId) {
                return "BPMN Skill";
        }
}
```

Fábrica abstracta y fábrica para SPEM y BPMN [1]

```
public abstract class AbstractProcessFactory {
        public abstract Task createTask ();
        public abstract WorkProduct createWorkProduct ();
        public abstract Role createRole ();
}

public SPEMFactory extends AbstractProcessFactory {
        public Task createTask () {
                return new SPEMProcessTask ();
        }
        public WorkProduct createWorkProduct () {
                return new SPEMProcessWorkProduct ();
        }
        public Role createRole () {
                return new SPEMProcessRole ();
        }
}
```

Fábrica abstracta y fábrica para SPEM y BPMN [2]

```
public BPMNFactory extends AbstractProcessFactory {
        public Task createTask () {
                return new BPMNProcessTask ();
        }
        public WorkProduct createWorkProduct () {
                return new BPMNProcessWorkProduct ();
        }
        public Role createRole () {
                return new BPMNProcessRole ();
        }
}
```

Momento final: “crear y usar el producto”

```
AbstractProcessFactory factory = null;

// decidir lo que crearemos
if (SPEM) {
        factory = new SPEMFactory ();
}
if (BPMN) {
        factory = new BPMNFactory ();
}

// supongamos creamos fabrica para SPEM
Task mySPEMTask = factory.createTask ();
// luego imprimimos el paso 2 de una tarea
System.out.println (mySPEMTask.getStep (2));
```

Singleton

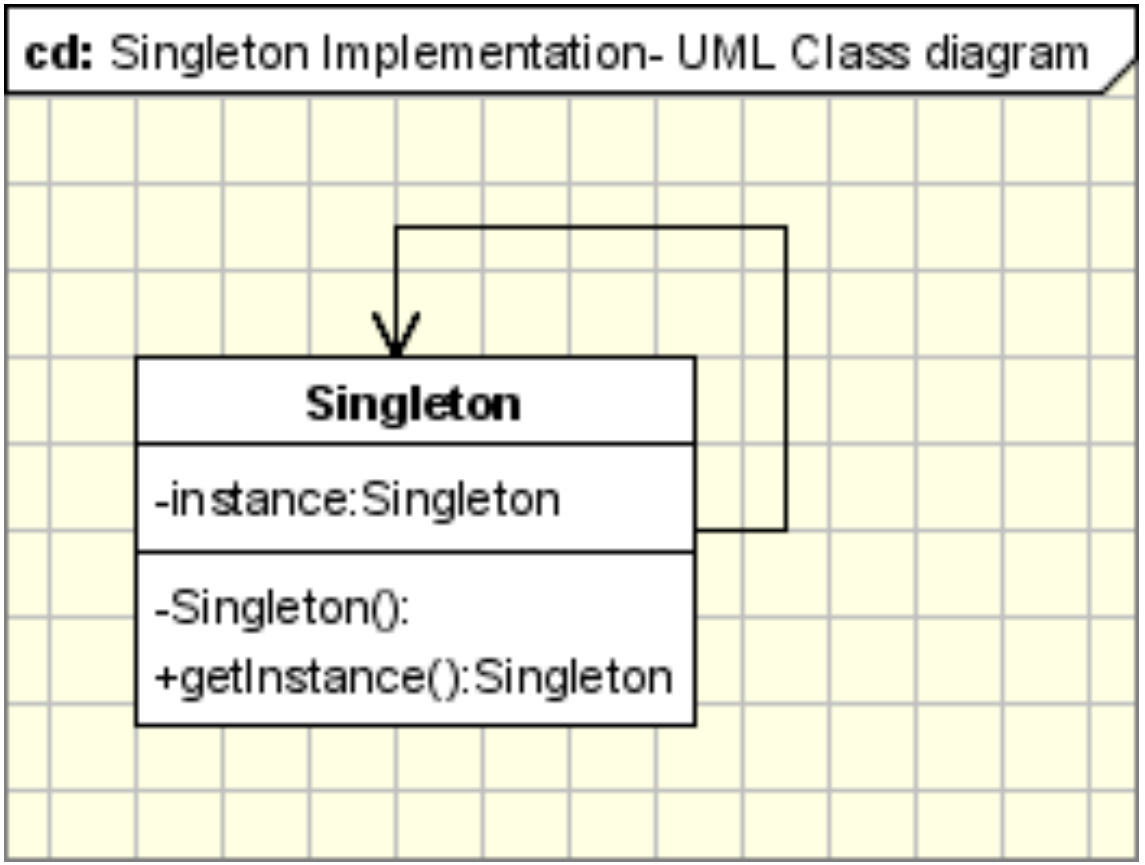
Singleton [1]

- Patrón de diseño **creacional**
- Propósito: asegurar que sólo una (y sólo una!) instancia de clase (objeto) pueda ser creada
- Con el patrón Singleton aseguramos también que exista sólo una forma de acceder al objeto
- Pregunta esencial: ¿cómo podemos detener la creación de nuevos objetos desde una misma clase?
  - Existen varias alternativas

Singleton [2]

- Primero, hacemos que el constructor de la clase sea privado
  - Con esto impediremos que alguien pueda crear una instancia de clase (objeto)
- Segundo, proveemos un *método estático* que retorna una instancia *estática* de la clase "singleton"
- Por definición, un método es estático cuando es declarado de forma tal que pueda ser accedido llamando directamente al objeto sin tener que pasar por el proceso de instanciación tradicional de una clase

Singleton [3]


cd: Singleton Implementation- UML Class diagram
Singleton
-instance:Singleton
-Singleton():
+getInstance():Singleton

Singleton [3]

```
class Singleton{

      private static Singleton instance;
      private Singleton(){
            ...
      }

      public static synchronized Singleton
getInstance(){
            if (instance == null)
                  instance = new Singleton();

            return instance;
      }
      ...
      public void doSomething(){
            ...
      }
}
```

Ejemplo Singleton: Fábrica de Chocolates

- Una fábrica de chocolate moderna tiene caldera de chocolate controlado por un computador.
- El trabajo de la caldera es tomar en el chocolate y la leche, llevarlos a ebullición, y luego pasarlos a la siguiente fase de hacer tabletas de chocolate.

Ejemplo Singleton: Fábrica de Chocolates

```
public class ChocolateBoiler{
  private boolean empty;
  private boolean boiled;

  public ChocolateBoiler() {
       empty = true;
       boiled = false;
  }
public void fill() {
      if (isEmpty()) {
      empty=false;
      boiled=false;
// Se llena el contenedor con
una mezcla de leche/chocolate
}
```

```
public void drain() {
if (!isEmpty() && isBoiled())
{
// drain the boiled milk and
      chocolate empty = true;
      }
}
public void boil() {
if (!isEmpty() && !
isBoiled()) {
// bring the contents to a
      boil boiled = true;
      }
}
public boolean isEmpty() {
      return empty;
}
public boolean isBoiled() {
      return boiled;
      }
}
```

Ejemplo Singleton: Fábrica de Chocolates

- Pero, si existen dos instancias de calderas de chocolate, algunas cosas pueden ocurrir, como por ejemplo, descontrol en la sincronización
- El patrón Singleton nos asegura que una clase tendrá solamente una instancia y provee un punto global de acceso a ella

Ejemplo Singleton: Fábrica de Chocolates (Solución [1])

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;
    private static ChocolateBoiler uniqueInstance;

    private ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public static ChocolateBoiler getInstance() {
        if (uniqueInstance == null) {
            System.out.println("Creating unique instance of Chocolate Boiler");
            uniqueInstance = new ChocolateBoiler();
        }
        System.out.println("Returning instance of Chocolate Boiler");
        return uniqueInstance;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }
```

Ejemplo Singleton: Fábrica de Chocolates (Solución [2])

```
    public void drain() {
        if (!isEmpty() && isBoiled()) {
            // drain the boiled milk and chocolate
            empty = true;
        }
    }

    public void boil() {
        if (!isEmpty() && !isBoiled()) {
            // bring the contents to a boil
            boiled = true;
        }
    }

    public boolean isEmpty() {
        return empty;
    }

    public boolean isBoiled() {
        return boiled;
    }
}
```

Ejemplo Singleton: Fábrica de Chocolates (Solución [3])

```
public class ChocolateController {
    public static void main(String args[]) {
        ChocolateBoiler boiler = ChocolateBoiler.getInstance();
        boiler.fill();
        boiler.boil();
        boiler.drain();

        // will return the existing instance
        ChocolateBoiler boiler2 = ChocolateBoiler.getInstance();
    }
}
```

Singleton [4]

- Un detalle, el patrón Singleton no está a salvo de problemas de concurrencia…
- ¿Cómo resolvemos esto?

Ejemplo Singleton tradicional: gestión de impresiones [1]

- Problema: tenemos una impresora de boletas modelo (hipotético) A80 que se “enoja” cuando recibe más de una petición para imprimir
- Utilizando el patrón de diseño Singleton podemos asegurar que nadie pueda crear más de una instancia de la clase gestora de impresiones
  - Por lo tanto, nadie podrá enviar a la impresora más de una solicitud de imprimir

Ejemplo Singleton tradicional: gestión de impresiones [2]

```
public class PrinterA80Manager {
        // instance es null al momento de crearse
        private static PrinterA80Manager instance;

        public synchronized static PrinterA80Manager getInstance () {
                if (instance == null) {
                        instance = new PrinterA80Manager ();
                }
                return instance;
        }
        // constructor es privado
        private PrinterA80Manager () {}

        public synchronized void Print (String text) {
                // imprimir de alguna forma
        }
}
```

Ejemplo Singleton tradicional: gestión de impresiones [3]

```
// instanciamos la clase; no existe otra forma de instanciar la clase
// por lo mismo, no podemos crear más de una instancia
PrinterA80Manager printManager = PrinterA80Manager.getInstance ();

printManager.Print ("fisw/a&dsw 2015");

printManager.print ("fisw/a&dsw 2015-2");
```

Singleton con “enum”

- Una alternativa al Singleton tradicional consiste en utilizar el tipo “enum” en lenguajes como JAVA

```
public enum PrinterA80Manager {
        // INSTANCE representa al Singleton
        INSTANCE;

        public synchronized void Print (String text) {
                // imprimir de alguna forma
        }
}

// imprimimos
PrinterA80Manager.INSTANCE.Print (“fisw/a&dsw 2015”);
```

¿Cuándo ocupar Singleton?

1. El patrón Singleton se utiliza en el diseño de las clases de registro (acceso a datos)
2. Para diseñar las clases que proporciona los valores de configuración de una aplicación
3. Se puede utilizar en el diseño de una aplicación que tiene que trabajar con puertos de serie

Builder

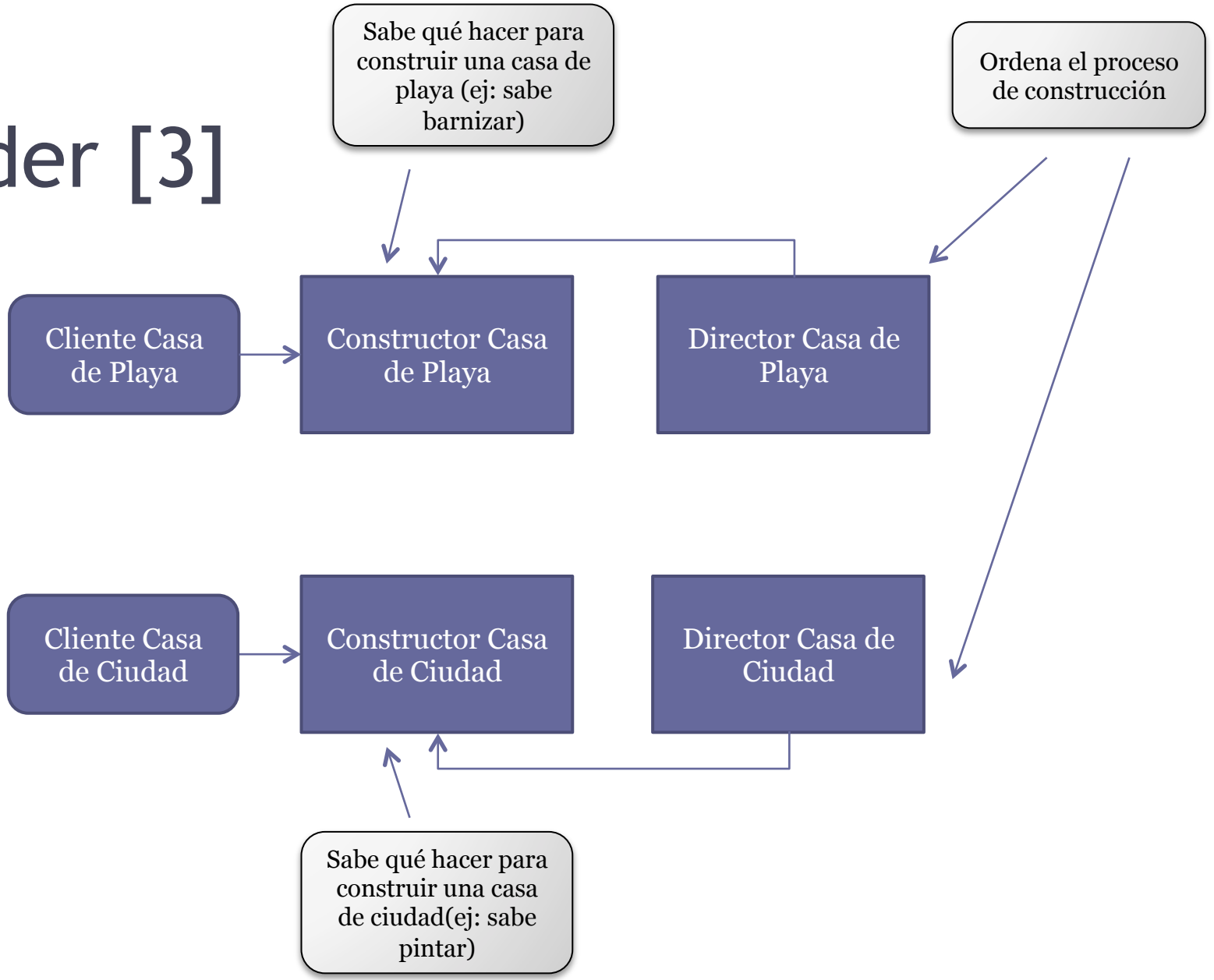
Builder [1]

- Patrón de diseño **creacional**
- Propósito: la construcción de un objeto complejo es separada de la representación. Un director de la construcción *vigila* el proceso de construcción
- Hace analogía a la construcción tradicional
  - Ejemplo: se utilizan dos procesos de construcción para distintas representaciones de casas (objeto complejo):
    - Casa de playa: de madera, requiere barnizado
    - Casa de ciudad: de hormigón, requiere pintado

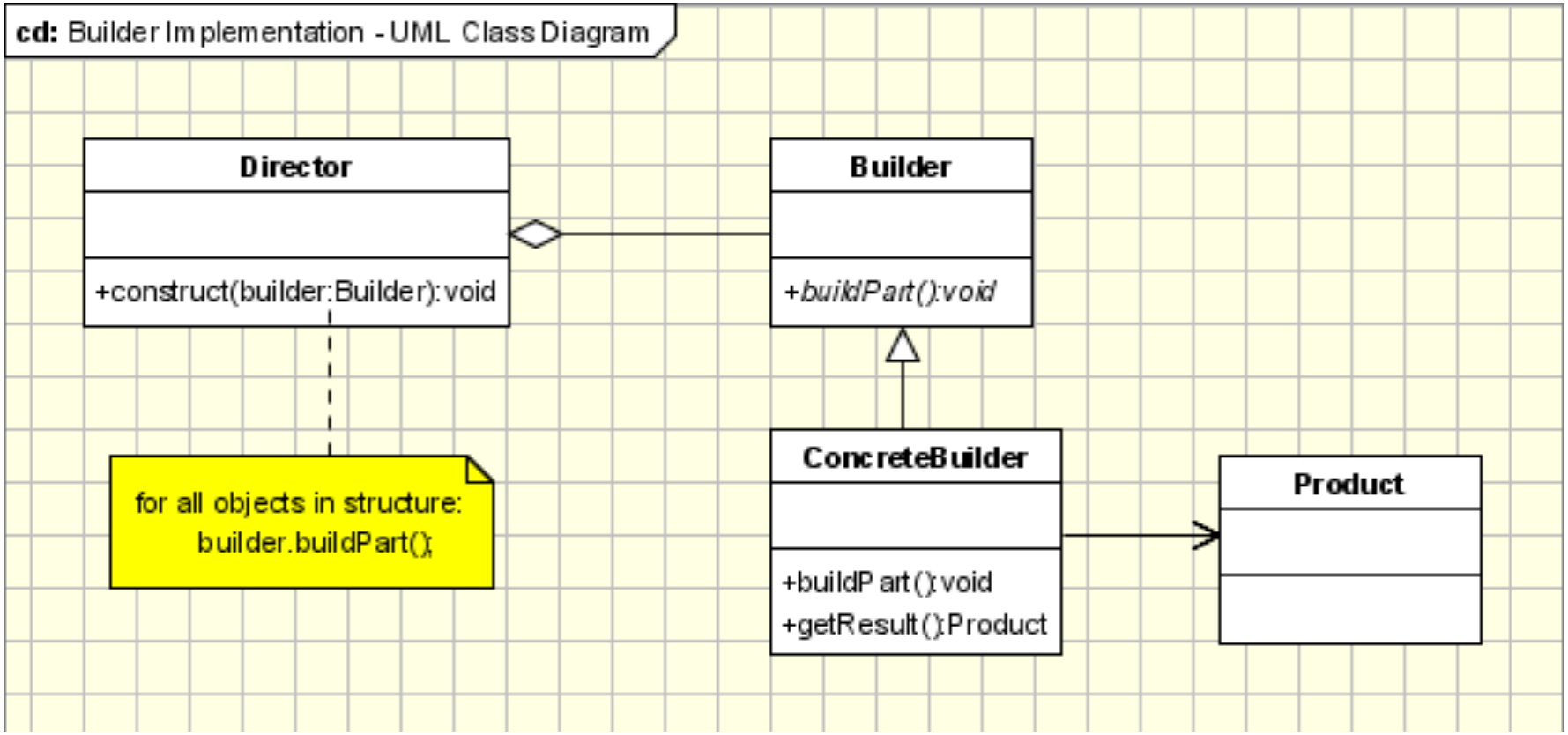
Builder [2]

- Aparecen dos nuevos tipos de clases
  - **Builder:** establece los métodos que se van a utilizar para construir una representación particular de objetos
    - Siguiendo la analogía, necesitamos dos "builders": uno para casa de playa y otro para casa de ciudad
  - **Director:** establece el orden de los pasos requeridos para la construcción del objeto
    - Siguiendo la analogía, los "directores" son los jefes de los proyectos "construir casa en la playa" y "construir casa en la ciudad"

Builder [3]


Sabe qué hacer para construir una casa de playa (ej: sabe barnizar)
Ordena el proceso de construcción
Cliente Casa de Playa
Constructor Casa de Playa
Director Casa de Playa
Cliente Casa de Ciudad
Constructor Casa de Ciudad
Director Casa de Ciudad
Sabe qué hacer para construir una casa de ciudad(ej: sabe pintar)

Builder [4]


cd: Builder Implementation - UML Class Diagram
Director
+construct(builder:Builder):void
Builder
+buildPart():void
for all objects in structure:
builder.buildPart();
ConcreteBuilder
+buildPart():void
+getResult():Product
Product

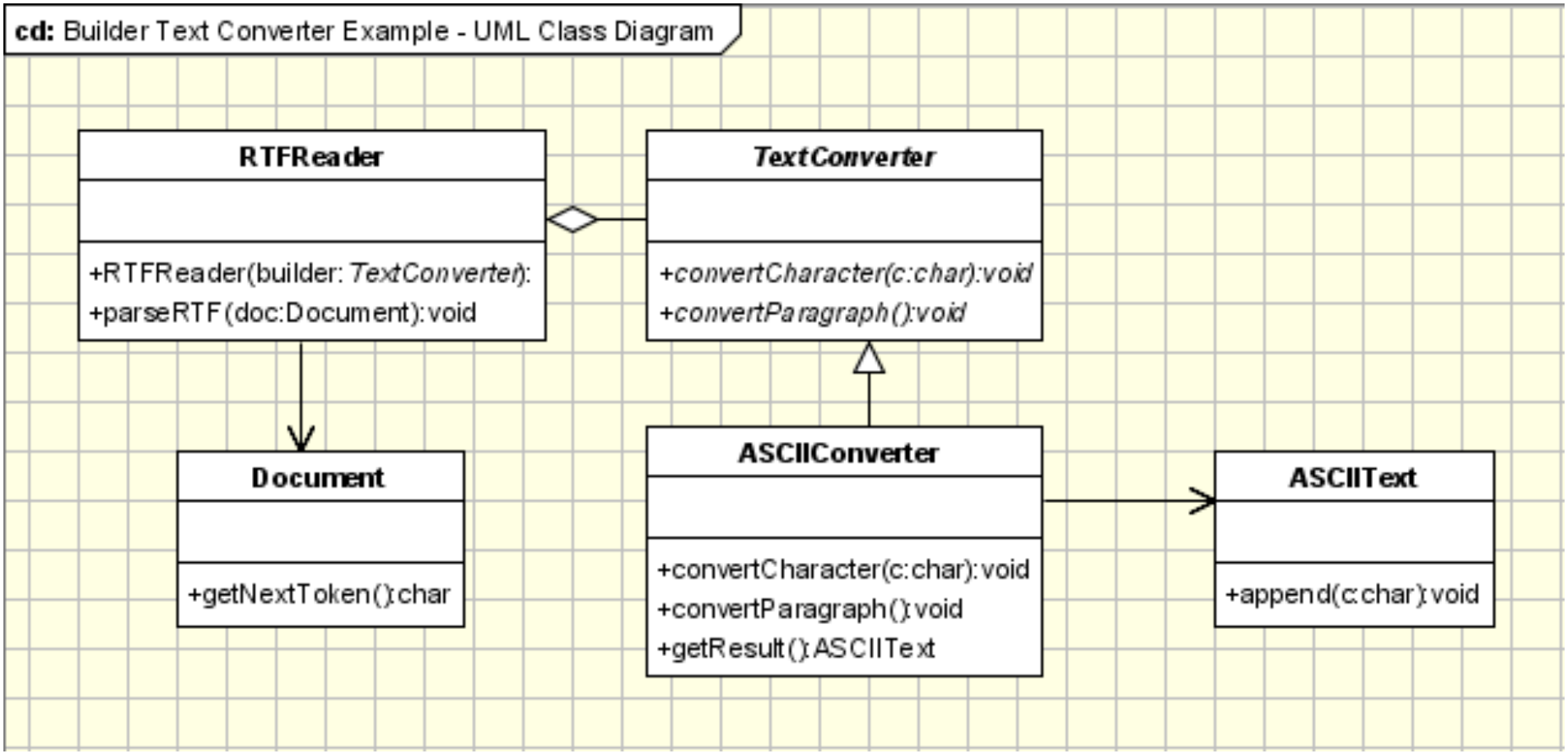
Builder [5]

- La clase Builder especifica una interface abstracta para crear partes de un objeto Product
- La clase ConcreteBuilder construye y junta las partes del producto que implementará la interface Builder
- La clase Director construye el objeto complejo usando la interface Builder
- La clase Product representa el objeto complejo que es necesario construir

Ejemplo Builder [1]

- Un cierto cliente necesita convertir un documento desde RTF hacia ASCII. Para solucionar el problema anterior, se aplicará el patrón Builder.
- Entonces, se llamará un método createASCIIText que toma como parámetro el documento que debe ser convertido. Este método llama a ASCIIConverter (ConcreteBuilder) que extiende al constructor (Builder), TextConverter, y sobre-escribe los dos métodos para convertir los caracteres a párrafos, y también el director, RTFReader analiza el documento y llama a los métodos constructores dependiendo del tipo de token encontrado.
- El producto, ASCIIText, es construido paso a paso, juntando los caracteres convertidos.

Ejemplo Builder [2]



Ejemplo Builder [3]

```
//Abstract Builder
class abstract class TextConverter{
        abstract void convertCharacter(char c);
        abstract void convertParagraph();
}

// Product
class ASCIIText{
        public void append(char c){ //Implement the code here }
}

//Concrete Builder
class ASCIIConverter extends TextConverter{
        ASCIIText asciiTextObj;//resulting product

        /*converts a character to target representation and appends to the resulting*/
        object void convertCharacter(char c){
                char asciiChar = new Character(c).charValue();
                        //gets the ascii character
                asciiTextObj.append(asciiChar);
        }
        void convertParagraph(){}
        ASCIIText getResult(){
                return asciiTextObj;
        }
}
```

Ejemplo Builder [3]

```
//This class abstracts the document object
class Document{
          static int value;
          char token;
          public char getNextToken(){
                    //Get the next token
                    return token;
          }
}

//Director
class RTFReader{
          private static final char EOF='0'; //Delimitor for End of File
          final char CHAR='c';
          final char PARA='p';
          char t;
          TextConverter builder;
          RTFReader(TextConverter obj){
                    builder=obj;
          }
          void parseRTF(Document doc){
                    while ((t=doc.getNextToken())!= EOF){
                              switch (t){
                                        case CHAR: builder.convertCharacter(t);
                                        case PARA: builder.convertParagraph();
                              }
                    }
          }


}
```

Ejemplo Builder [3]

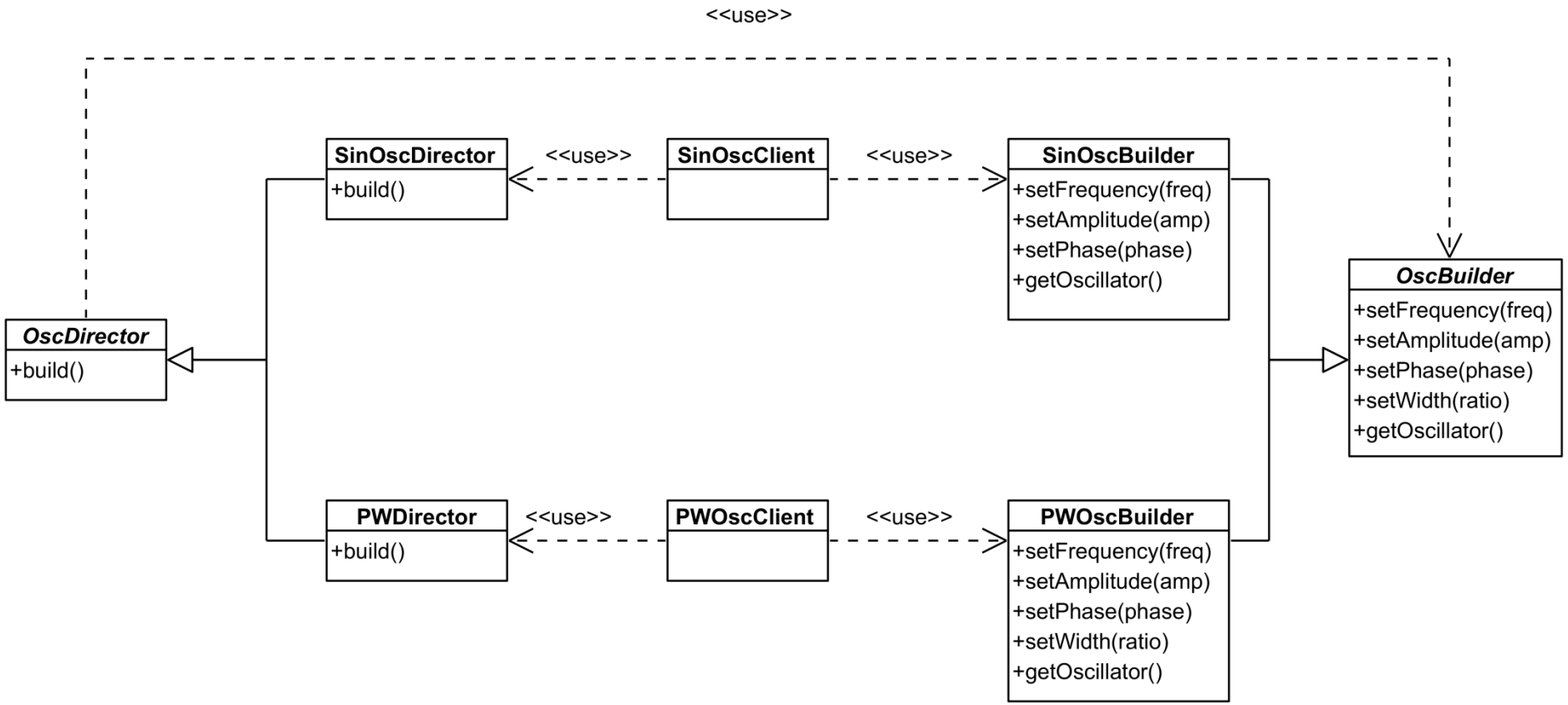
```
//Client
public class Client{
	void createASCIIText(Document doc){
		ASCIIConverter asciiBuilder = new ASCIIConverter();
		RTFReader rtfReader = new RTFReader(asciiBuilder);
		rtfReader.parseRTF(doc);
		ASCIIText asciiText = asciiBuilder.getResult();
	}
	public static void main(String args[]){
		Client client=new Client();
		Document doc=new Document();
		client.createASCIIText(doc);
		system.out.println("This is an example of Builder Pattern");

	}
}
```

Ejemplo: Builder con JAVA [1]

- Tenemos dos generadores de ondas:
  - Oscilador senoidal (sin)
  - Oscilador de ancho-de-pulsos (pw)
- Ambos generadores son del mismo tipo, pero se construyen de forma distinta
  - El oscilador senoidal requiere, como mínimo, los típicos parámetros: frecuencia, amplitud, fase
  - El oscilador de ancho-de-pulsos requiere, además, definir el ancho del pulso (usualmente como un ratio)
- Para el ejemplo, supondremos que contamos con una interfaz/clase de tipo “Osc” que representa a los posibles osciladores

Ejemplo: Builder con JAVA [2]



Ejemplo: Builder con JAVA [3]

```
// definimos TODOS los posible métodos para construir un
// oscilador (generador de onda)
public abstract class OscBuilder {
	public void setFrequency(float freq) {}
	public void setAmplitude(float amp) {}
	public void setPhase(float phase) {}
	public void setWidth(float ratio) {}

	public abstract Osc getOscillator();
}

// ahora es tarea de cada builder específico hacer el
// override necesario para los métodos que necesita
```

Ejemplo: Builder con JAVA [4]

```
// el constructor de onda senoidal conoce los métodos para
// construir uno de ellos
public class SinOscBuilder extends OscBuilder {
	// recibimos el objeto a construir o setear (constructor?)
	public void setFrequency(float freq) {
		// establecer frecuencia
	}
	public void setAmplitude(float amp) {
		// establecer amplitud
	}
	public void setPhase(float phase) {
		// establecer fase
	}
	public Osc getOscillator() {
		// retornamos el objeto seteado
		return sinOscInProgress;
	}
}
```

Ejemplo: Builder con JAVA [5]

```
// el constructor de ondas pwm sabe los métodos para construir un
// oscilador de este tipo
public class PWOscBuilder extends OscBuilder {
        // recibimos el objeto a construir o setear (constructor?)
        public void setFrequency(float freq) {
                // establecer frecuencia
        }
        public void setAmplitude(float amp) {
                // establecer amplitud
        }
        public void setPhase(float phase) {
                // establecer fase
        }
        public void setWidth(float ratio) {
                // establecer ratio
        }
        public Osc getOscillator() {
                // retornamos el objeto seteado
                return pwOscInProgress;
        }
}
```

Ejemplo: Builder con JAVA [6]

```
// cómo recibimos el objeto a construir?
// por ejemplo, queremos construir una onda de sonido senoidal
private SinOsc sinOscInProgress;

// usamos el constructor de los builders para asignar la onda de
// sonido al oscilador
public SinOscBuilder(SinOsc sinOsc) {
        sinOscInProgress = sinOsc;
}

// de manera análoga, recibimos el objeto en el constructor
// de ondas pulse-width
```

Ejemplo: Builder con JAVA [7]

```
// el director define el método build
public abstract class OscDirector() {
	public abstract Osc build(OscBuilder oscBuilder);
}
// los directores conocen el orden de construcción
public class SinOscDirector extends OscDirector() {
	public Osc build(OscBuilder oscBuilder){
		oscBuilder.setFrequency(…);
		oscBuilder.setAmplitude(…);
		oscBuilder.setPhase(…);

		return oscBuilder.getOscillator();
	}
}
```

Ejemplo: Builder con JAVA [8]

```
public class PWOscDirector extends OscDirector() {
        public Osc build(OscBuilder oscBuilder){
                oscBuilder.setFrequency(…);
                oscBuilder.setAmplitude(…);
                oscBuilder.setPhase(…);
                oscBuilder.setWidth(…);

                return oscBuilder.getOscillator();
        }
}
```

Ejemplo: Builder con JAVA [9]

```
// para construir un oscilador senoidal:
// primero, creamos un constructor de osciladores senoidales
// le entregamos como parámetro el objeto a construir o 'setear'
SinOsc sinOsc = new SinOsc();
OscBuilder oscBuilder = new SinOscBuilder(sinOsc);

// segundo: creamos al director y le pedimos que construya en el
// orden que sabe
OscDirector oscDirector = new SinOscDirector();
Osc newSinOscillator = oscDirector.build(oscBuilder);

// ya podemos utilizar el objeto 'newSinOscillator'
```

¿Cuándo ocupar Builder?

- Sistemas de manufactura
- Creación de certámenes
- Creación de algoritmos complejos
- Construcción de diferentes representaciones de objetos

Fin