Prototype

Escenario

Un juego de rol interactivo tiene un apetito insaciable por los monstruos. A medida que sus héroes hacen su viaje a través de un paisaje creado dinámicamente, se encuentran con una cadena interminable de enemigos que deben ser sometidos.

Escenario

Te gustaría que las características del monstruo evolucionen con el paisaje cambiante. No tiene mucho sentido para los monstruos parecidos a las aves seguir a tus personajes en los reinos debajo del mar.

Escenario

Finalmente, le gustaría permitir que los jugadores avanzados creen sus propios monstruos personalizados.

Estudiante 1

Sería mucho más limpio si pudiéramos desacoplar el código que maneja los detalles de la creación de los monstruos desde el código que realmente necesita crear las instancias sobre la marcha.

Estudiante 2

El solo hecho de crear todos estos diferentes tipos de instancias de monstruos es cada vez más complicado ...

Poner todo tipo de detalles de estado en los constructores no parece ser muy cohesivo. Sería genial si hubiera un solo lugar donde todos los detalles de la instanciación pudieran ser encapsulados ...

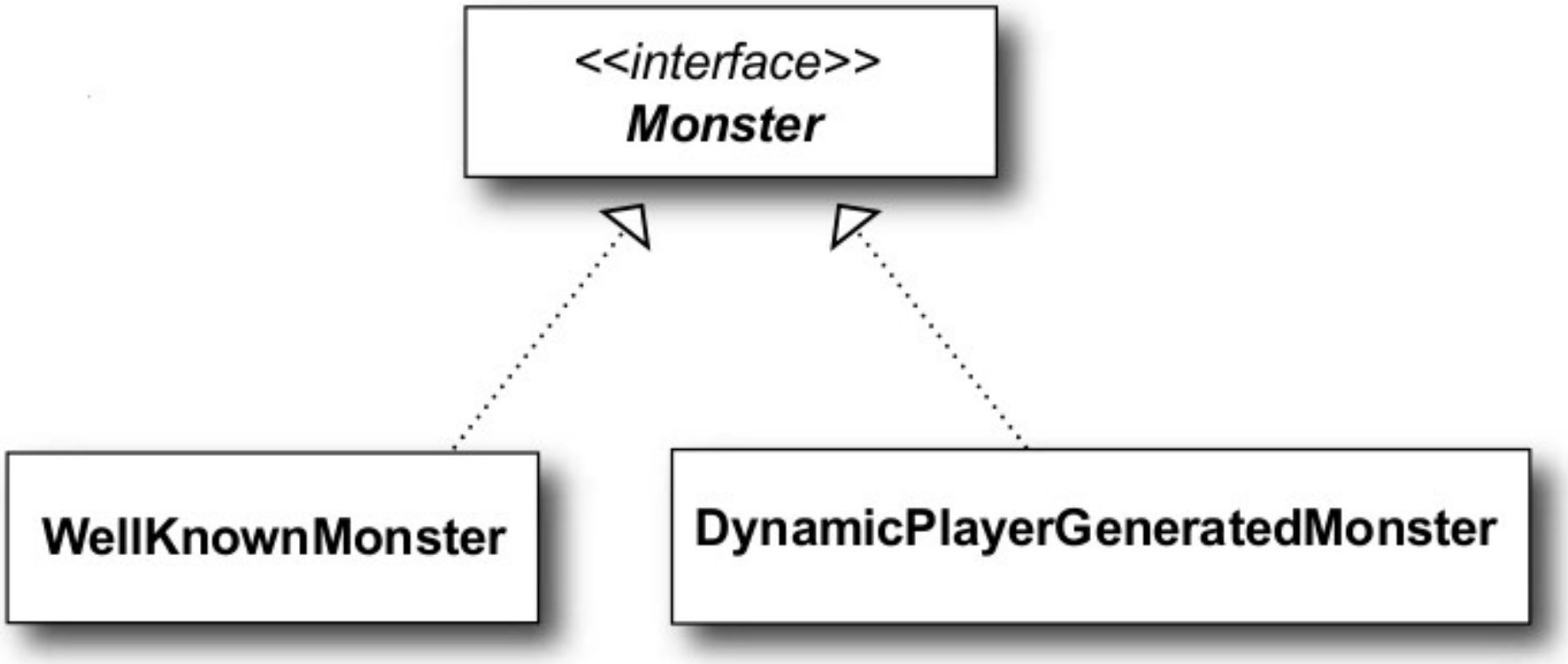
Y ahora ¿quien podrá ayudarnos?

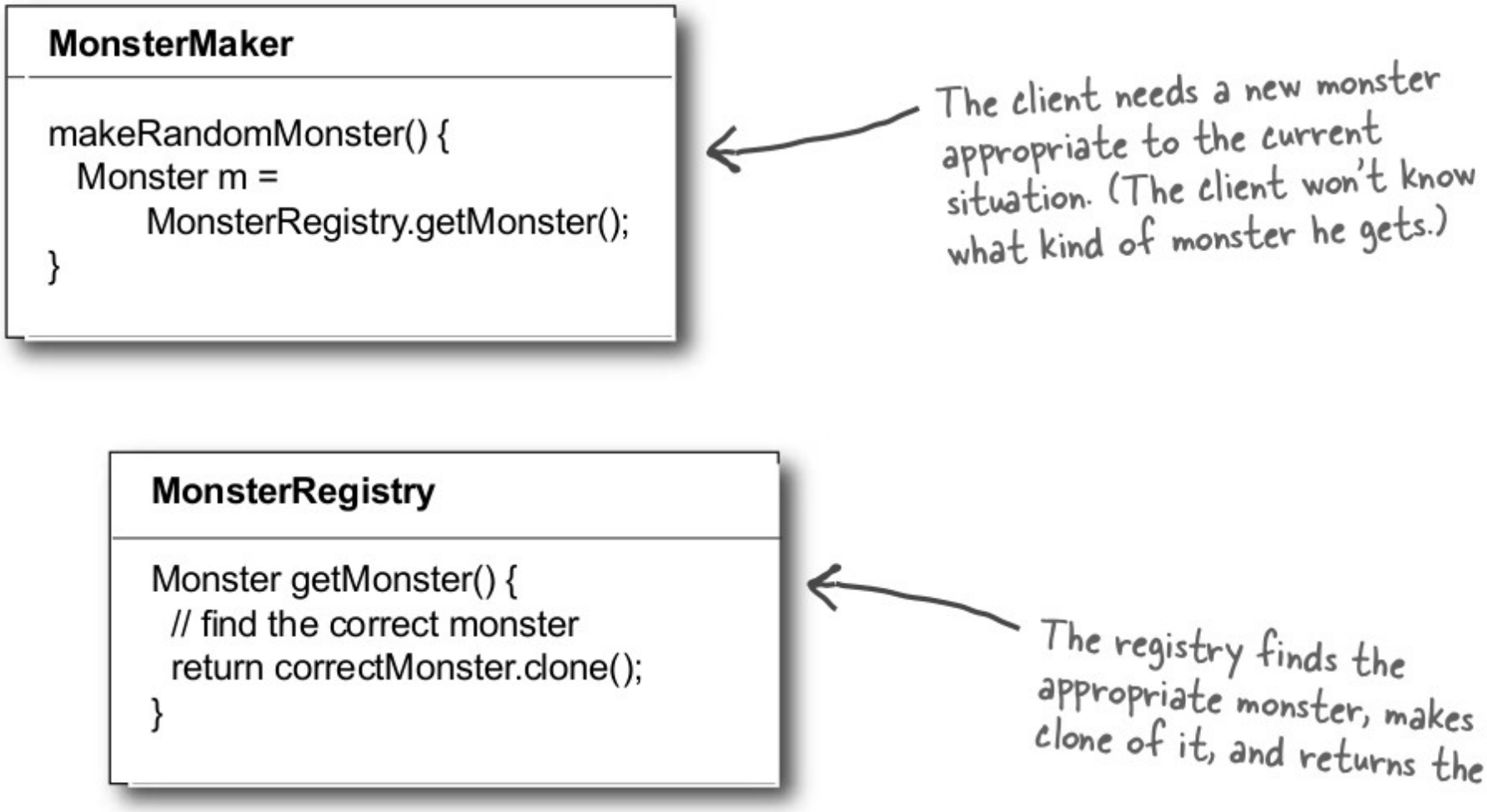
Prototype

Usa el patrón de prototipo cuando crear una instancia de una clase determinada es costosa o complicada

En algunos casos, el coste de crear un objeto nuevo desde 0 es muy elevado, y más aún si luego hay que establecer una gran colección de atributos. En éste contexto sería más conveniente clonar un objeto predeterminado que actúe de prototipo y modificar los valores necesarios para que se ajuste a su nuevo propósito.

El patrón Prototype le permite hacer nuevas instancias copiando instancias existentes. (En Java, esto generalmente significa usar el método clone() o deserialización cuando necesita copias en profundidad.) Un aspecto clave de este patrón es que el código del cliente puede crear nuevas instancias sin saber qué clase específica se está instanciando.


<<interface>>
Monster
WellKnownMonster
DynamicPlayerGeneratedMonster


MonsterMaker
makeRandomMonster() {
Monster m =
MonsterRegistry.getMonster();
}
The client needs a new monster appropriate to the current situation. (The client won't know what kind of monster he gets.)
MonsterRegistry
Monster getMonster() {
// find the correct monster
return correctMonster.clone();
}
The registry finds the appropriate monster, makes a clone of it, and returns the clone.

La solución consistirá en **definir una interfaz que expone el método necesario** para realizar la clonación del objeto. Las clases que pueden ser clonadas implementarán esta interfaz, mientras que las clases que deseen clonar deberán utilizar el método definido en la interfaz.

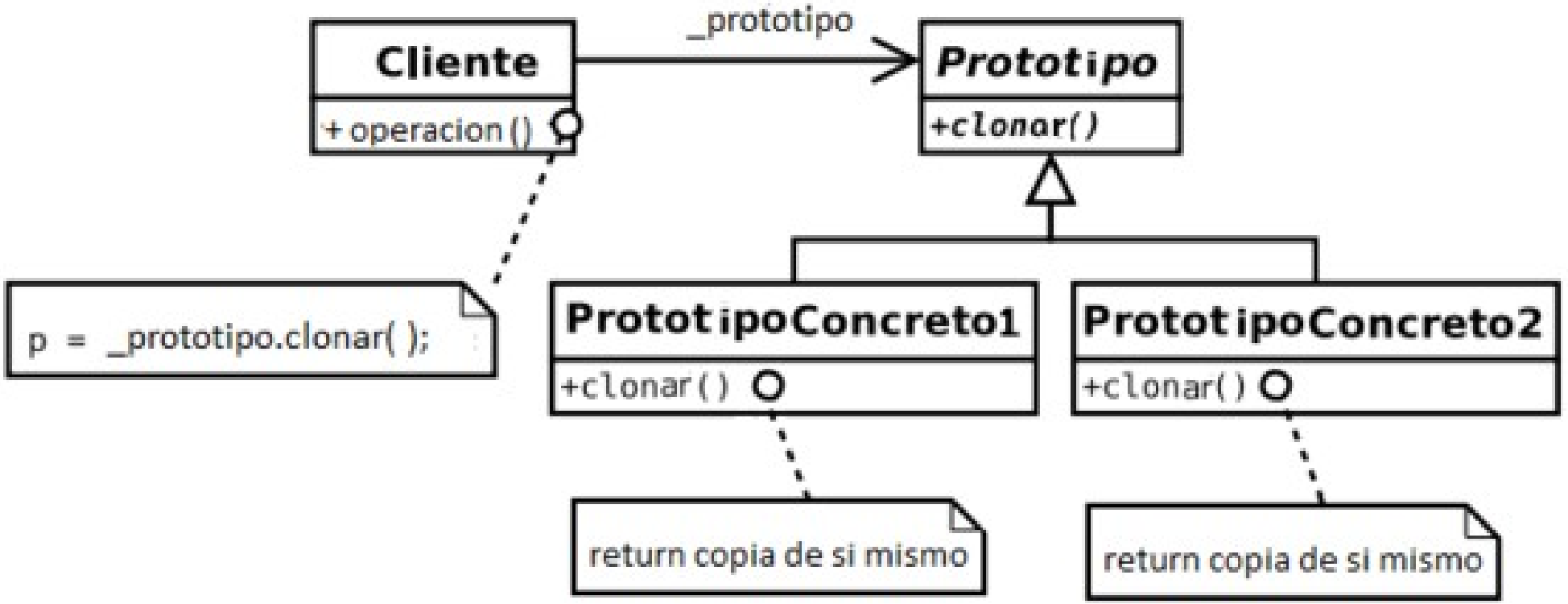
Además, existen 2 tipos de clonación: la clonación profunda y la clonación superficial. En la **clonación superficial modificar las referencias a terceros objetos hace que los originales cambien**, ya que los terceros objetos son en realidad apuntadores. En la **clonación profunda se clonan los terceros objetos dando lugar a nuevas referencias** independientes.

Participantes

Cliente: es el encargado de solicitar la creación de los nuevos objetos a partir de los prototipos.

Prototipo: declara una interfaz para clonarse, a la que accede el cliente.

Prototipo Concreto: posee características concretas que serán reproducidas para nuevos objetos e implementa una operación para clonarse.


Cliente
+ operacion ()
_prototipo
Prototipo
+clonar()
p = _prototipo.clonar( );
PrototipoConcreto1
+clonar()
PrototipoConcreto2
+clonar()
return copia de si mismo
return copia de si mismo

Aterricemos el patrón con un ejemplo más sencillo

```java
public abstract class Shape implements Cloneable {

   private String id;
   protected String type;

   abstract void draw();

   public String getType(){
      return type;
   }

   public String getId() {
      return id;
   }

   public void setId(String id) {
      this.id = id;
   }

   public Object clone() {
      Object clone = null;

      try {
         clone = super.clone();

      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }

      return clone;
   }
}
```

```java
public class Rectangle extends Shape {

   public Rectangle(){
     type = "Rectangle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

```java
public class Circle extends Shape {

   public Circle(){
     type = "Circle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

```java
import java.util.Hashtable;

public class ShapeCache {
    |
   private static Hashtable<String, Shape> shapeMap  = new Hashtable<String, Shape>();

   public static Shape getShape(String shapeId) {
      Shape cachedShape = shapeMap.get(shapeId);
      return (Shape) cachedShape.clone();
   }

   // for each shape run database query and create shape
   // shapeMap.put(shapeKey, shape);
   // for example, we are adding three shapes

   public static void loadCache() {
      Circle circle = new Circle();
      circle.setId("1");
      shapeMap.put(circle.getId(),circle);

      Square square = new Square();
      square.setId("2");
      shapeMap.put(square.getId(),square);

      Rectangle rectangle = new Rectangle();
      rectangle.setId("3");
      shapeMap.put(rectangle.getId(), rectangle);
   }
}
```

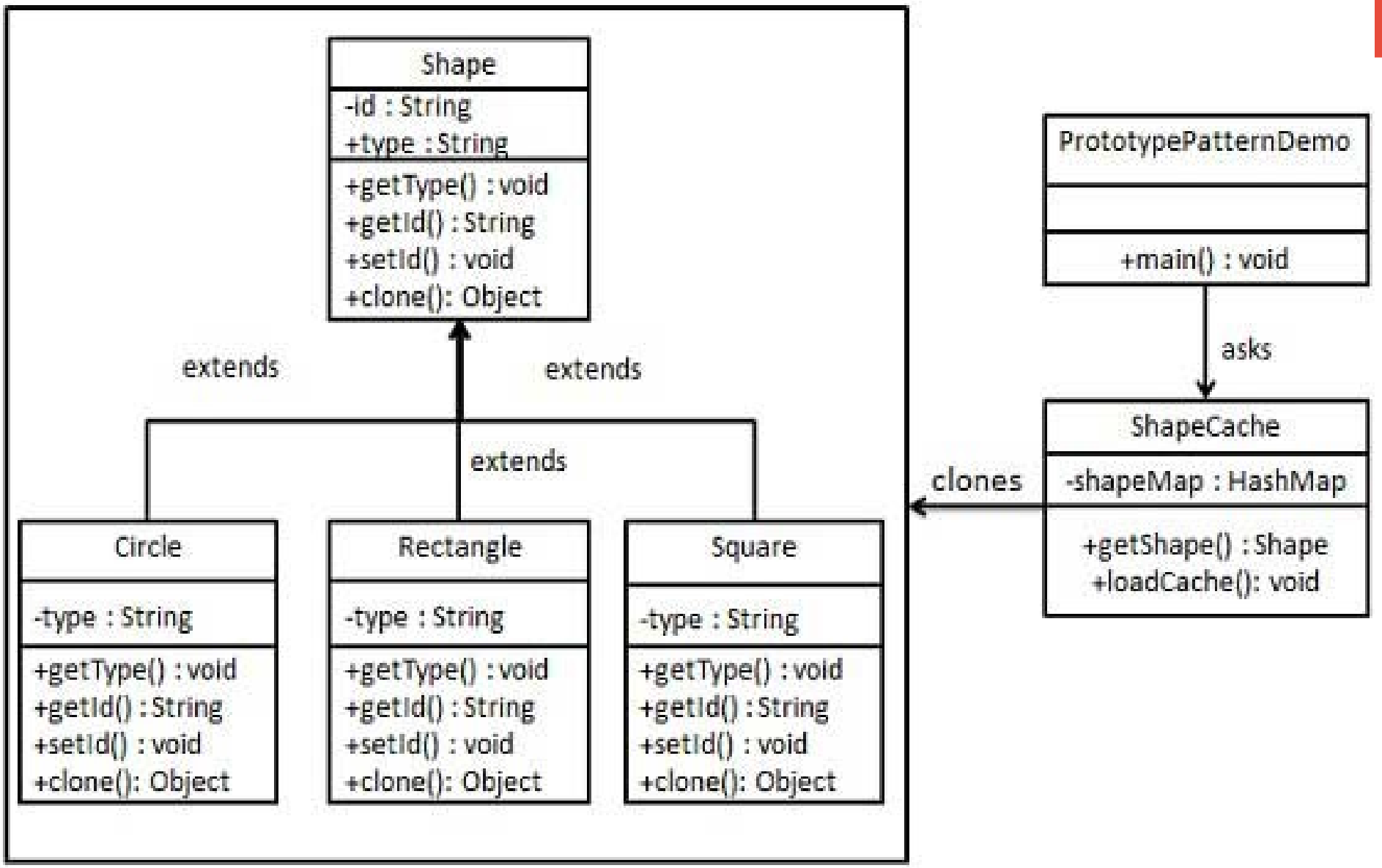
```java
public class PrototypePatternDemo {
   public static void main(String[] args) {
      ShapeCache.loadCache();

      Shape clonedShape = (Shape) ShapeCache.getShape("1");
      System.out.println("Shape : " + clonedShape.getType());

      Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
      System.out.println("Shape : " + clonedShape2.getType());

      Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
      System.out.println("Shape : " + clonedShape3.getType());
   }
}
```

```
Shape : Circle
Shape : Square
Shape : Rectangle
```


Shape
-id : String
+type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
extends
extends
extends
Circle
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
Rectangle
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
Square
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
clones
PrototypePatternDemo
+main() : void
asks
ShapeCache
-shapeMap : HashMap
+getShape() : Shape
+loadCache(): void

Beneficios de prototype

- **Oculta las complejidades de crear nuevas instancias del cliente, es decir el cliente no debe conocer los detalles de cómo construir los objetos prototipo.**
- **Proporciona la opción para que el cliente genere objetos cuyo tipo no se conoce.**
- **En algunas circunstancias, copiar un objeto puede ser más eficiente que crear un nuevo objeto.**

- **Clonar un objeto es mucho más rápido que crearlo.**
- **Un programa puede añadir y borrar dinámicamente objetos prototipo en tiempo de ejecución.**


PLANTS VS. ZOMBIES 2
Intenta conseguir 2 columnas de productoras solares.
© 2021 Electronic Arts Inc.


500
2330
Antiguo Egipto - Día 6
50
100
50
100
0
175
800
1200
1000

Usos e inconveniente de prototype

- **La creación de nuevos objetos acarrea un coste computacional elevado.**
- **Los objetos a crear tienen o suelen tener atributos que repiten su valor.**
- **El prototipo debe considerarse cuando un sistema debe crear nuevos objetos de muchos tipos en una jerarquía de clases complejas.**

Usos e inconveniente de prototype

- **Una desventaja de usar el Prototipo es que hacer una copia de un objeto a veces puede ser complicado.**

- **Dado que en objetos muy complejos, implementar la interfaz Prototype puede ser muy complicada.**

Repaso de Patrones que hemos visto

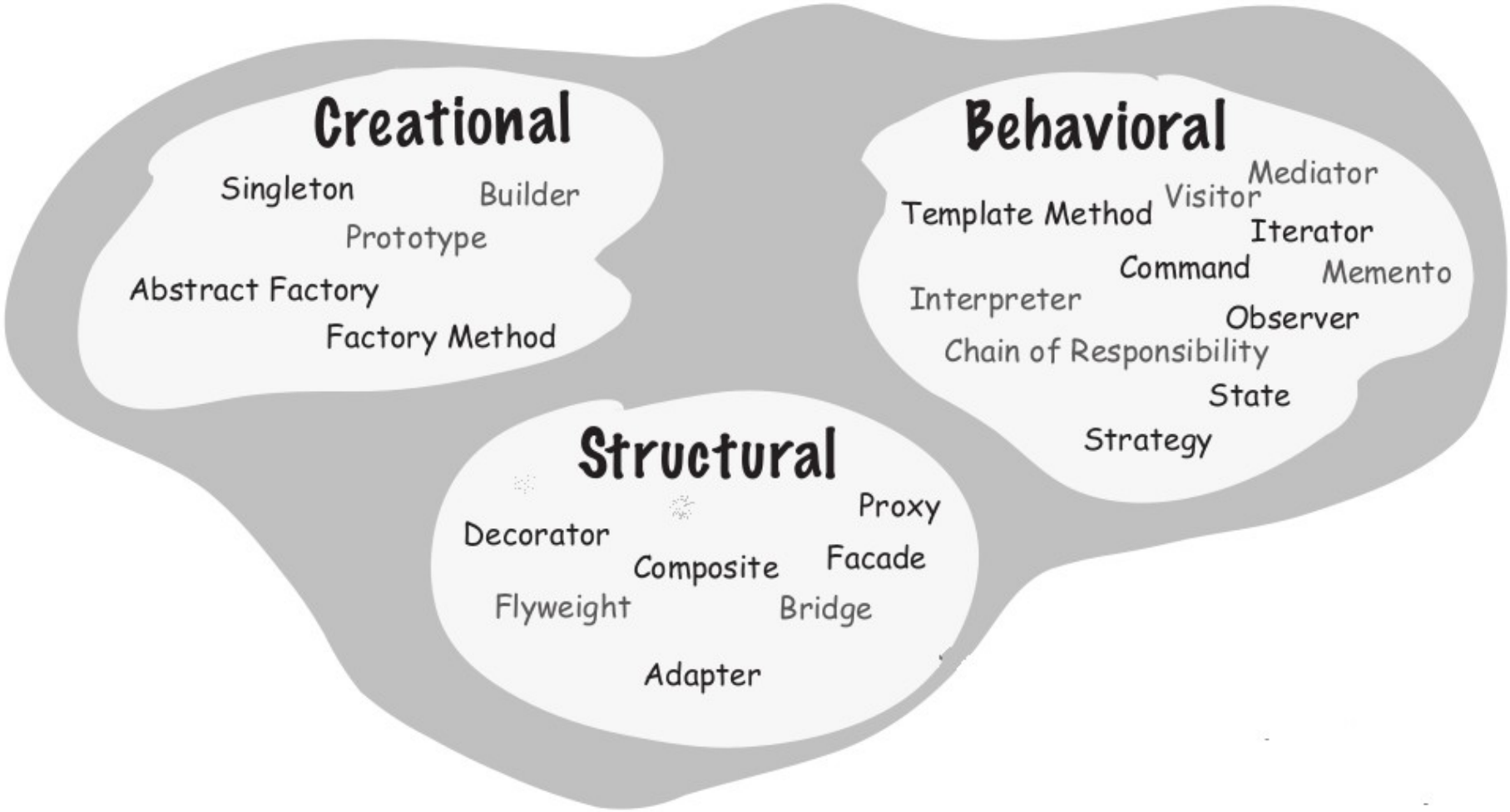
Repaso de patrones

Cualquier patrón que sea un patrón de comportamiento se refiere a cómo las clases y los objetos interactúan y distribuyen la responsabilidad.

Los patrones estructurales le permiten componer clases u objetos en estructuras más grandes.

Repaso de patrones

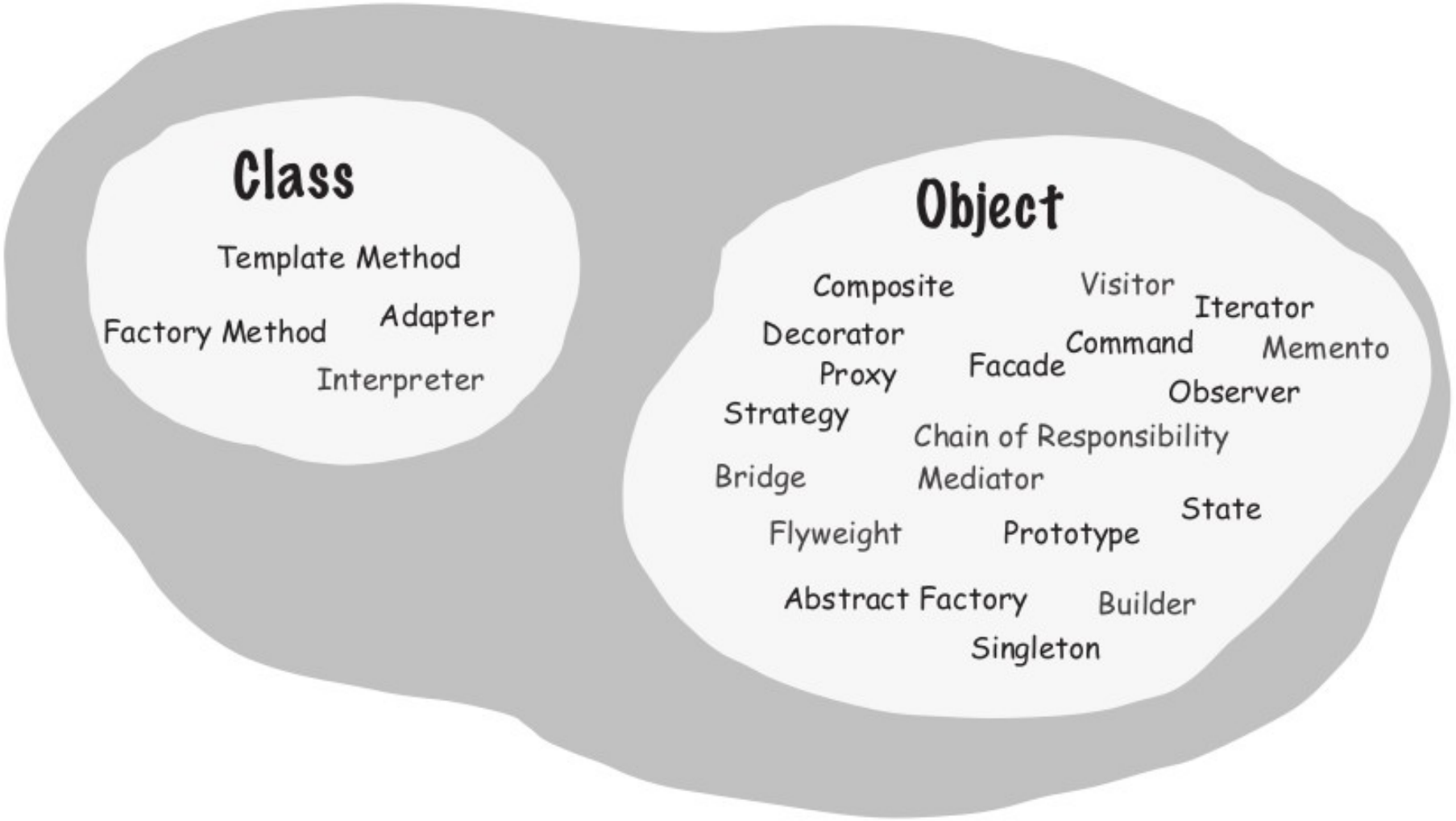
Los patrones de creación implican creación de instancias de objetos y todos proporcionan una forma de desacoplar a un cliente de los objetos que necesita para crear instancias.


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

Otra separación

Los patrones de clase describen cómo las relaciones entre clases se definen a través de la herencia. Las relaciones en los patrones de clase se establecen en tiempo de compilación.

Los patrones de objetos describen las relaciones entre los objetos y se definen principalmente por la composición.
Las relaciones en los patrones de objeto se suelen crear en tiempo de ejecución y son más dinámicas y flexibles.


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