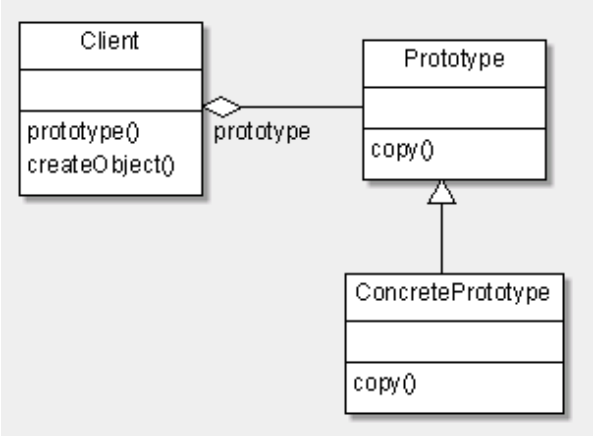
Creacion

Prototype

Con este patrón vamos a poder crear nuevas instancias de una clase a través de una instancia prototipo la cual clonaremos o copiaremos para crear nuevas instancias.

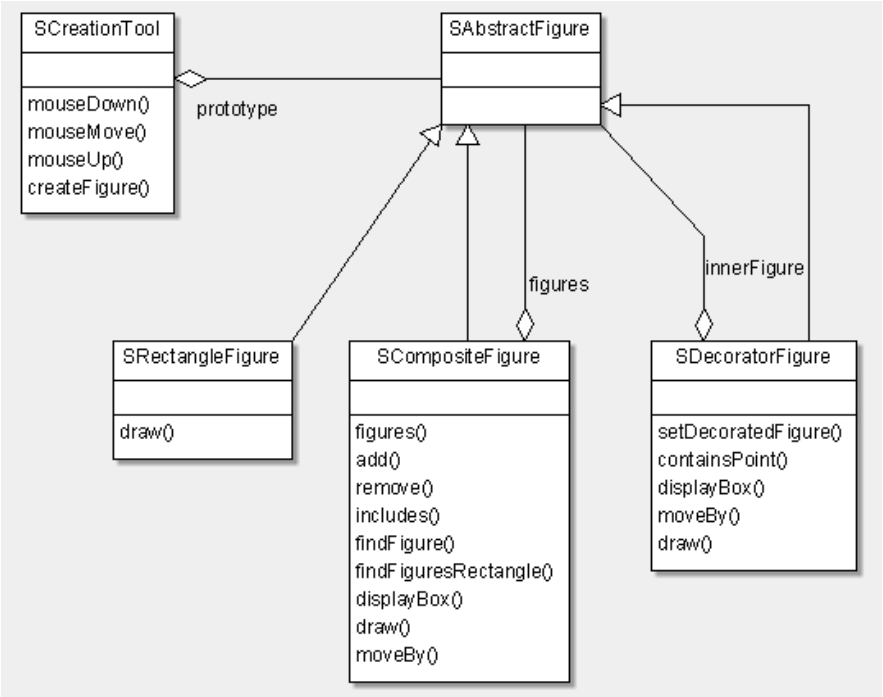


Invocando al método createObject se crearán nuevas instancias del tipo prototipo a través del método copy del objeto prototype agregado. Así el método constructor creará un objeto tal y como indica el siguiente código:

```
class Client{
 Component createObject(){
   return this.prototype.copy();
 }
}
```

En HotDraw: (Este nos permitirá crear cualquier figura solamente con una clase de creación)

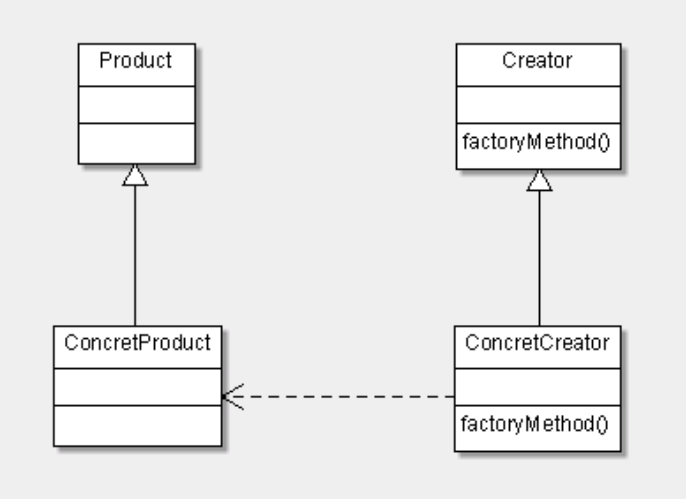
Se usa en las herramientas para no tener que crear una clase por cada botón de creación que necesitemos.



El Client es la clase uCreationTool y las clases Prototype serán la clase uAbstractFigure y las clases Concreteprototype serán todas las clases que heredan de uAbstractFigure. El método clone() de Java no funciona adecuadamente por lo que habrá que añadir el método clone() a la interfaz IFigure e implementarlo en todas las figuras que implementan dicha interfaz.

Factory Method

Define una interfaz, realmente suele ser un único método, para crear objetos, pero deja a las subclases la implementación de este método.

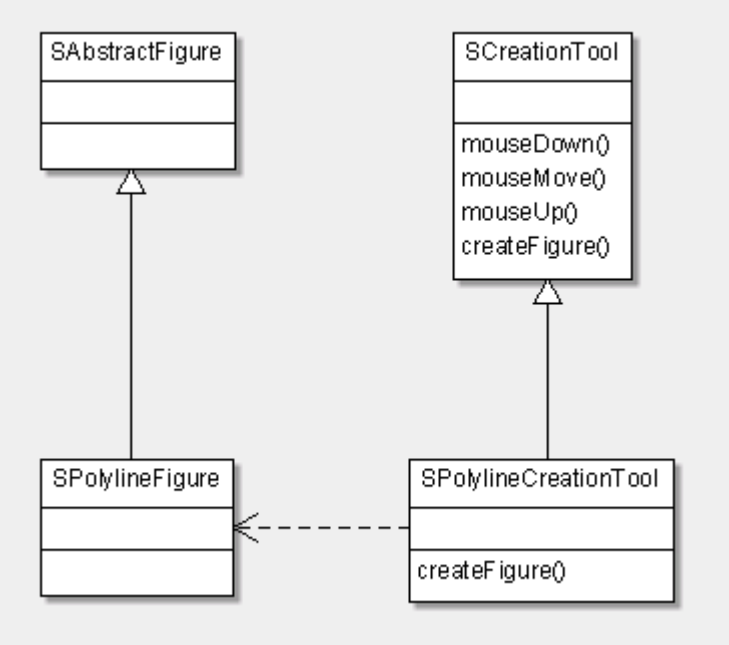


El meetodo factoryMethod() es el ideado como creador de objetos. En la clase Creator no se implementa, se hace en la clase ConcretCreator. Es posible que en la clase Creator haya métodos que invoquen a factoryMehtod y su ejecución dependerá de la subclase en la que se estén ejecutando. El método factoryMethod esta especializado en la creación de instancias de ConcretProduct, de ahí la relación de dependencia entre ambos.

```
class ConcretCreator{
 Product methodFactory(){
   return new ConcreteProduct();
 }
}
```

En HotDraw:

El metodo factoryMethod se denomina createFigure y se implementa en la clase uPolylineCreationTool que seria la clase ConcretCreator. La clase Creator seria la clase uCreationTool que no es abstracta como indica el patrón sino que es participe del patrón prototype.



implementación de createFigure de uPolylineCreationTool seria:

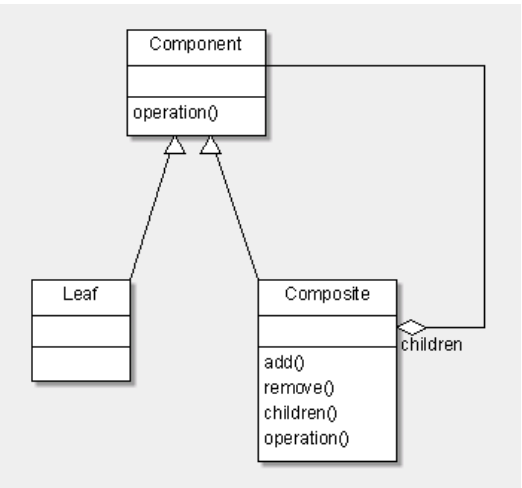
```
IFigure creationFigure ()
 return new uPolylineFigure()
```

Otros métodos de la clase Creator (uCreationTool) puedesn utilizar createFigure y esto ocurre en el método mouseUp, el cual puede crear instancias de clases tanto mediante prototipos como mediante clases que heredan de uCreationTool.

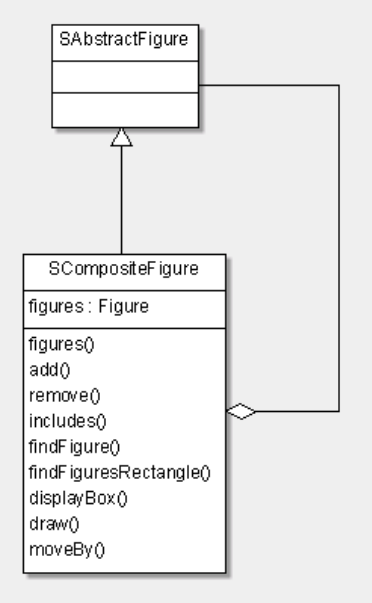
Estructurales

Composite

Permite crear componentes complejos a partir de componentes simples de manera recursiva.

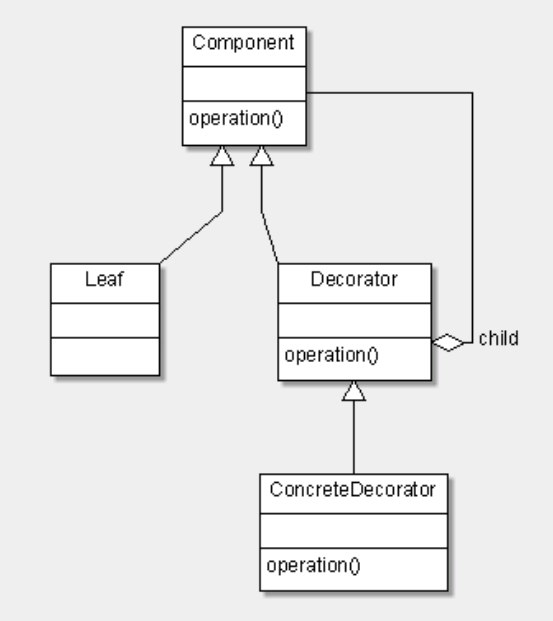


En HotDraw:(Para crear figuras compuestas a partir de figuras sencillas).
La clase Component será uAbstractFigure y la clase Composite será uCompositeFigure, la clase Leaf será el resto de clases que heredan de uAbstractFigure.



Decorator

Permite ampliar la funcionalidad de los objetos sin manipular los ya existentes.
Se pretende ampliar un objeto ya existente.



La clase Decorator agrega a una clase child que será la decorada o a la que queremos ampliar su funcionalidad, en este caso queremos ampliar la operación operation. Para ello debemos crear una clase hija de Decorator que contenga la ampliación necesaria en nuestro caso ConcreteDecorator. Así la implementación de operation() de Decorator seria:

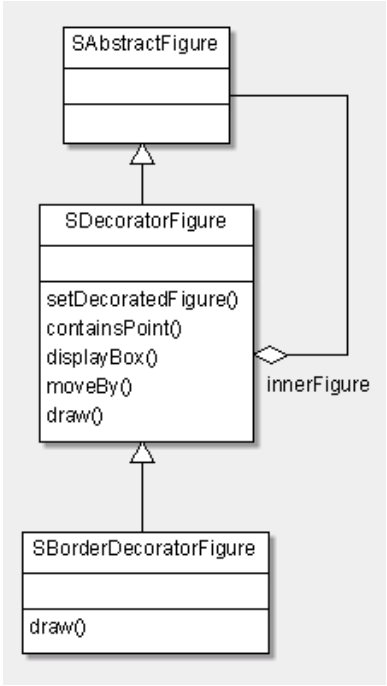
```
class Decorator{
 void operation(){
  child.operation();
 }
}
```

Y la implementación de operation de ConcreteDecorator seria:

```
class ConcreteDecorator{
 void operation(){
   super.operation();
   operationExtension();
 }
}
```

Donde vemos que lo primero que hacemos es utilizar la funcionalidad del objeto agregado y posteriormente invocar al método operationExtension() que ampliará la funcionalidad con las operaciones oportunas.

En HotDraw: (Poder ampliar la funcionalidad básica de una figura, pero de forma flexible).
La siguiente figura muestra la instanciación del patrón de diseño decorador utilizado para crear figuras con borde a través de la clase uBorderDecoratorFigure. Donde el componet es uAbstractFigure la clase decorador es uDecoretorFigure y la clase Concretedecorator es uBorderDecoratorFigure.
El funcionamiento de la clase uDecoratorFigure es simplemente envolver la clase decorada y reenviarle los métodos apropiados.
Mientra que la clase uBorderDecoratorFigure se redefinen los métodos que necesitemos para ampliar la funcionalidad, en este caso para ponerle un borde a la figura.



De esta manera la implementación del método draw en ambas clases seria:

```
class uDecoratorFigure{
 void draw(Graphics g){
  innerFigure.draw(g);
 }
}

class uBorderDecoratorFigure{
 void draw(Grapics g){
```

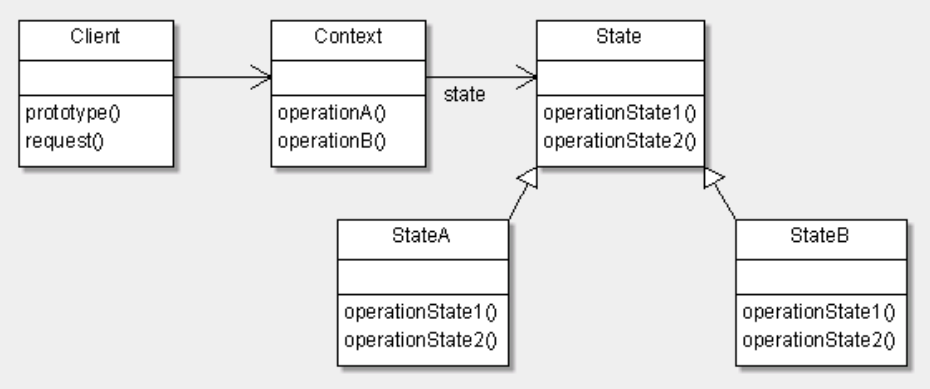
```
  super.draw(g);
  g.drawRect(innerFigure.displayBox());
 }
}
```

Comportamiento

State

Nos permite definir diferentes comportamientos en función del estado en el que se encuentre el objeto.

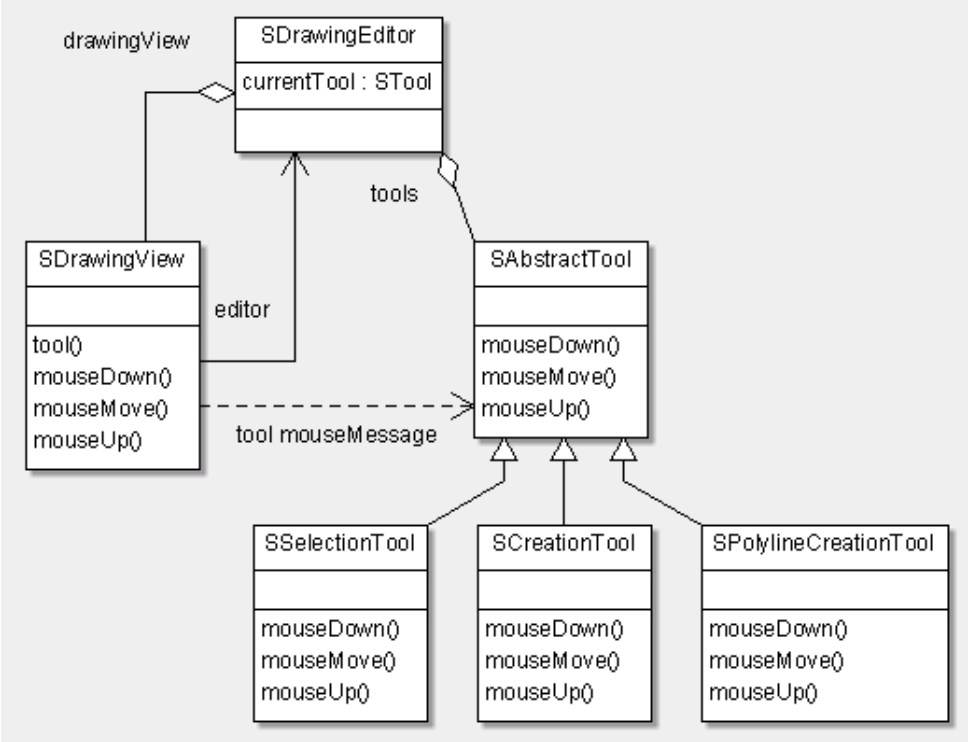
Vamos a tener una clase padre que define una serie de funcionalidades y para cada estado en el que puede estar los objetos de esa clase vamos a definir una clase hija que va a redefinir las funcionalidades.



En HotDraw:

Las herramientas es un estado en el que se encuentra la aplicación y en función del cual se responderá de manera diferente a los eventos de ratón.

Así según la figura siguiente la clase Context es uDrawingView, la clase State es uAbstractTool y las clases de estado StateA y StateB son las diferentes herramientas del programa como son uCreationTool, uSelectionTool y uPolylineCreationTool.



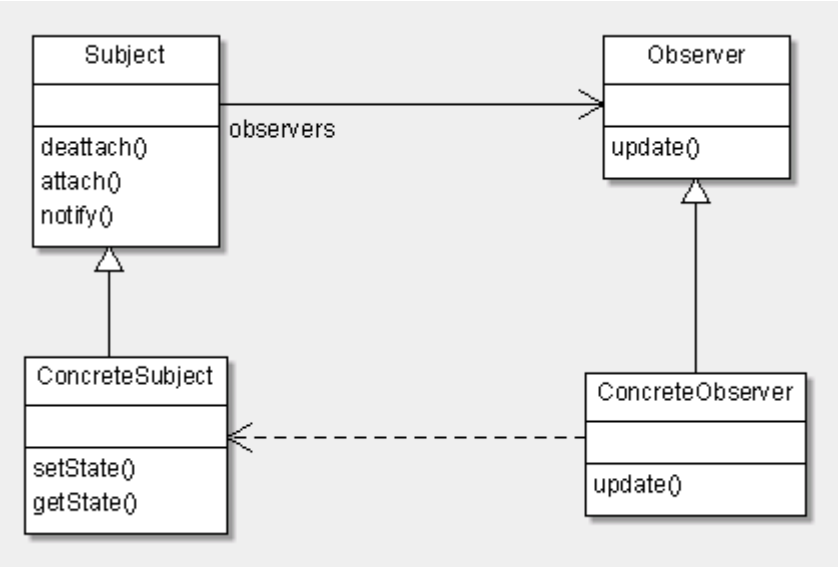
De esta forma necesitamos que la vista uDrawingView sepa en a que editor está agregado para luego solicitarle la herramienta seleccionada actualmente. Esto se consigue mediante el atributo editor y el método tool() cuya implementación seria:

```
class uDrawingView{
 ITool tool(){
  return editor.getCurrentTool();
 }
}
```

TAMBIEN SE USA PARA SELECCION

Observer

Permite que un observador sea notificado, cuando le sucede algún evento a otra clase observada. Para ello el observador debe suscribirse a la clase observada, para ello en el patrón se utiliza el método attach().
La clase observada deberá tener una colección de objetos observadores la cual recorrerá cuando desee informarles del evento ocurrido invocando un método que todas las clases observadoras deberán tener implementado, esto lo realiza el método notofy() de la clase Subject en el patrón, como se muestra en la figura siguiente.



Este patrón está muy extendido en el lenguaje Java mediante el uso de las interfaces Listener. Así tenemos la interfaz ActionListener que define la interfaz actionPerformed(). De esta forma en Java las clases observadas, deben tener una colección de objetos ActionListener y un método addActionListener(), esta último es equivalente al attach() del patrón Observer. Los observadores deben implementar el método actionPerformed() que será invocado cuando se genera una notificación. Todos los objetos de la interfaz de usuario JSwing, como JButton, JLabel, JPanel, etc,.. son objetos observados que notifican a los implementadotes de la interfaz ActionListener.

En HotDraw: Se usa cuando se añade una figura o se modifica el dibujo la vista que contiene el dibujo sea informada para que vuelva a repintar todo el dibujo y el usuario vea los cambios. En Java se usa con las interfaces Listener.
Se implementa dos veces, por un lado la vista OBSERVA al dibujo y si este cambia se actualiza y por otro el dibujo OBSERVA todas las figuras que contiene y si alguna cambia el dibujo cambia y la vista se vuelve a actualizar.

`public interface FigureListener`
`{`
`void figureChanged();`
`}`
	Que implementará la clase uDrawing para ser notificada de los cambios de las figuras que contiene.

`public interface DrawingListener`
`{`
`void drawingChanged();`
`}`
	Que implementara la clase uDarwingView para ser notificada de los cambios del dibujo que contiene.