

JHotDraw

**Aplicación de patrones a la construcción de Frameworks**



JHotDraw

## Introducción



# JHotDraw

- **Orígenes**
  - *JHotDraw se ha inspirado en:*
    - HotDraw. Framework en Smalltalk. Desarrollado por Kent Beck y Ward Cunningham
    - ET++. Framework y librería de clases portable para C++. Desarrollado por Andre Weinand y Erich Gamma
  - *JHotDraw ha sido desarrollado por Thomas Eggenschwiler y Erich Gamma y fue presentado en el OOPSLA97*
  - *Fue creado para un seminario como ejemplo de la aplicación de patrones en la creación de Frameworks, pero sus ideas son directamente aplicables a las aplicaciones profesionales*
  - *Han colaborado también personas como Uwe Steinmueller, Doug Lea, Kent Beck, y Ward Cunningham.*

# JHotDraw

- **Objetivo**
  - *Framework para editores de manipulación directa*
- **Característica**
  - *Distintos tipos de figuras*
  - *Manipulación directa de las figuras*
  - *Conexión de figuras*
  - *Herramientas*
  - *Actualización en pantalla eficiente*
  - *Applets y aplicaciones*

# JHotDraw

- **Framework**

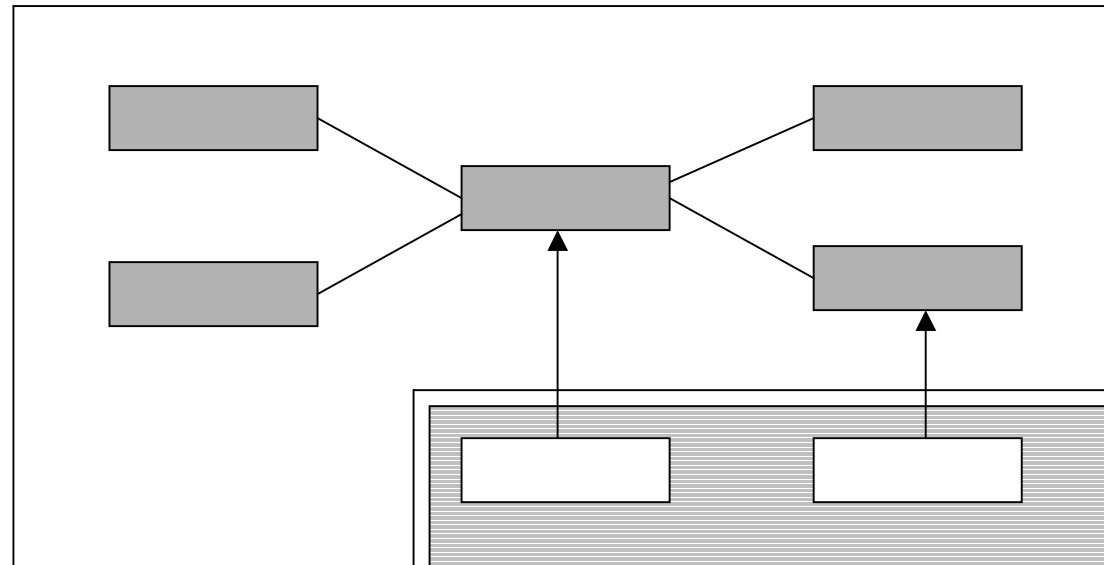
- *Colaboración de clases adaptables que definen una solución para un problema dado*

- Definen las abstracciones fundamentales y sus interfaces
- Establecen las interacciones entre los objetos
- Adaptación: redefinición
- Soluciones por defecto

- *Reutilización*

- Arquitectura y diseño
- Código

# JHotDraw



# JHotDraw

- **Proceso**
  - ***Desarrollo de un modelo del dominio (OOD)***
    - Identificar abstracciones fundamentales y sus responsabilidades (CRC, Use Cases)
  - ***Diseño flexible y extensible con patrones***
    - Flexibilidad: Strategy, State, Decorator...
    - Independencia: Abstract Factory, Bridge...
  - ***Integrar en una infraestructura existente***
    - Adapter, Bridge
  - ***Implementación***
    - Coding Patterns
    - Idiomas específicos del lenguaje

# JHotDraw

- **Problemas de Diseño**
  - *Abstracciones fundamentales*
  - *Herramientas*
  - *Actualización de pantalla*
  - *Manipulación de Figuras*
  - *Conexiones*
  - *Entorno de Ejecución*

JHotDraw

## **Abstracciones fundamentales**



# Abstracciones fundamentales

- **Problemas de Diseño**
  - *Abstracciones fundamentales*
  - *Herramientas*
  - *Actualización de pantalla*
  - *Manipulación de Figuras*
  - *Conexiones*
  - *Entorno de ejecución*

# Abstracciones fundamentales

- **Figuras**
  - *Simples y complejas*
  - *Pueden tener un número indeterminado de atributos*
- **Responsabilidades**
  - *dibujarse*
  - *saber su posición y tamaño*
  - *moverse*
- **La abstracción de las figuras debe ser independiente de cualquier detalle de implementación**

# Abstracciones Fundamentales

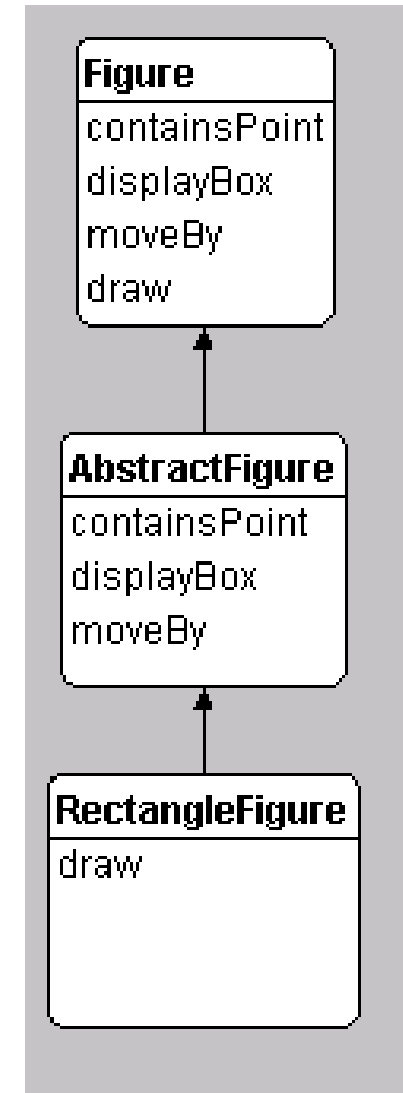
```
interface Figure
{
    displayBox
    containsPoint
    draw
    moveBy
    ...
}
```

# Abstracciones Fundamentales

- **Desarrollo de Frameworks**
  - ***Separación de diseño y código:***
    - “We believe that interface design and functional factoring constitute de key intellectual content of software and that are far more difficult to create or re-create than code”. Peter Deutsch
  - ***Por tanto separar diseño e implementación con interfaces***

# Abstracciones Fundamentales

- **Figura define un interface**
- **La clase abstracta AbstractFigure ofrece un comportamiento por defecto**
  - *template methods*
  - *implementaciones por defecto*



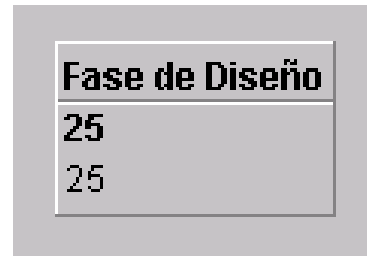
# Abstracciones Fundamentales

- **Asignación de nombres a las abstracciones**
  - *Nombre simple para la superclase (SBPP)*
    - Una palabra que exprese su propósito
    - Nombres compuestos para las subclases
- **En HotDraw**
  - *Interface: nombre simple*
  - *implementaciones por defecto*
    - **Abstract<X>**: Necesita ser redefinido
    - **Standard<X>**: Puede ser reusado directamente

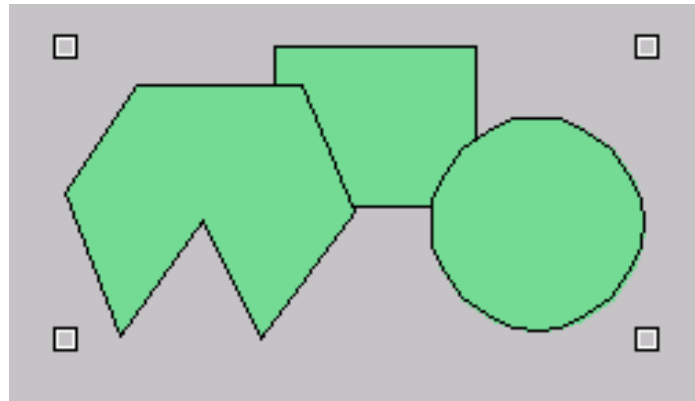
# Abstracciones Fundamentales

- Figuras compuestas

- *PertFigure*



- *GroupFigure*



- *¿Patrón?*

# Composite

---

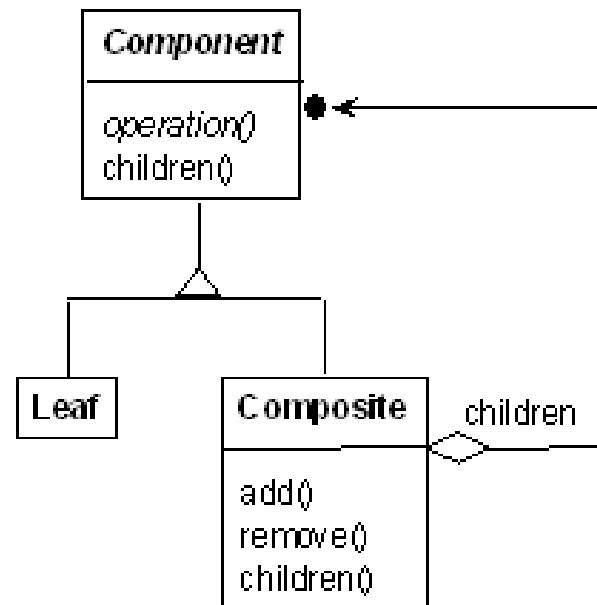
- **Intent**

- Enable to assemble complex objects out of primitive objects, recursive composition

- **Applicability**

- need to assemble objects out of primitive objects
- represent part-whole hierarchies

- **Structure**



# Abstracciones Fundamentales

- **Composite en JHotDraw**

```
interface Figure {  
    Enumeration figures()  
    includes(Figure)  
    ...  
}
```

```
abstract class CompositeFigure extends  
    AbstractFigure {  
    ...  
    add(Figure)  
    remove(Figure)
```

```
    moveBy() { for each figure do figure.moveBy() }  
    draw() { for each figure do figure.draw() }  
}
```

# Abstracciones Fundamentales

- Ejemplos

```
class PertFigure extends CompositeFigure {
    PertFigure() {
        ...
        TextFigure name = new TextFigure();
        name.setText("Task");
        add(name);

        NumberTextFigure duration = new NumberTextFigure();
        duration.setValue(0);
        add(duration);

        NumberTextFigure end = new NumberTextFigure();
        end.setValue(0);
        end.setReadOnly(true);
        add(end);
    }
}
```

# Abstracciones Fundamentales

- **Atributos**
  - *Puede haber mucho atributos (demasiados métodos de acceso)*
  - *Muchos son específicos de un tipo de figura (tipo de letra, ancho de la línea...)*
  - *Algunos son específicos de una figura concreta (URL...)*
- **¿Patrón?**

# Abstracciones Fundamentales

- **Variable State Pattern (SBPP)**

- *Define un interface genérico de acceso a pares {clave, valor}*

```
void setAttribute(String name, Object value)  
Object getAttribute(String name)
```

- **Implementación**

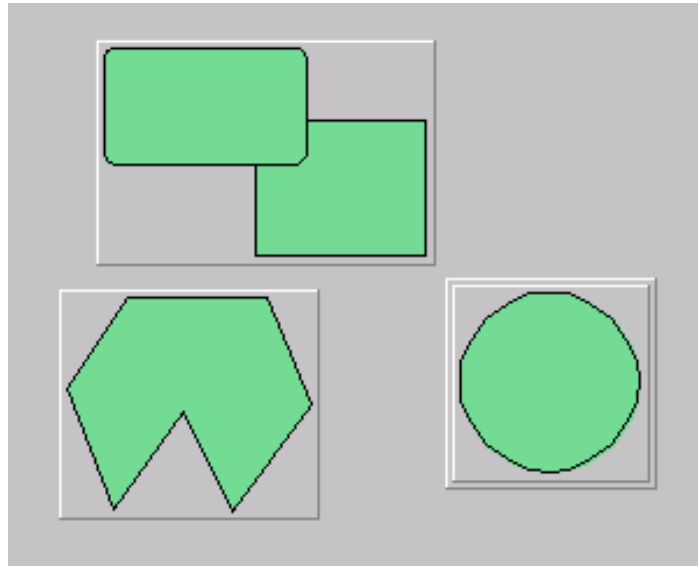
- *Atributos*
- *Diccionario*

- **La clase AttributeFigure da la implementación de un figura cuyo estado se almacena en un diccionario**

# Abstracciones Fundamentales

- **Problema**

- *Añadir comportamiento a las figuras (bordes, sombras...)*



- *Extensión no intrusiva de la figura*
- *Filtro de los mensajes*
- *¿Patrón?*

## Decorator (AKA Wrapper)

---

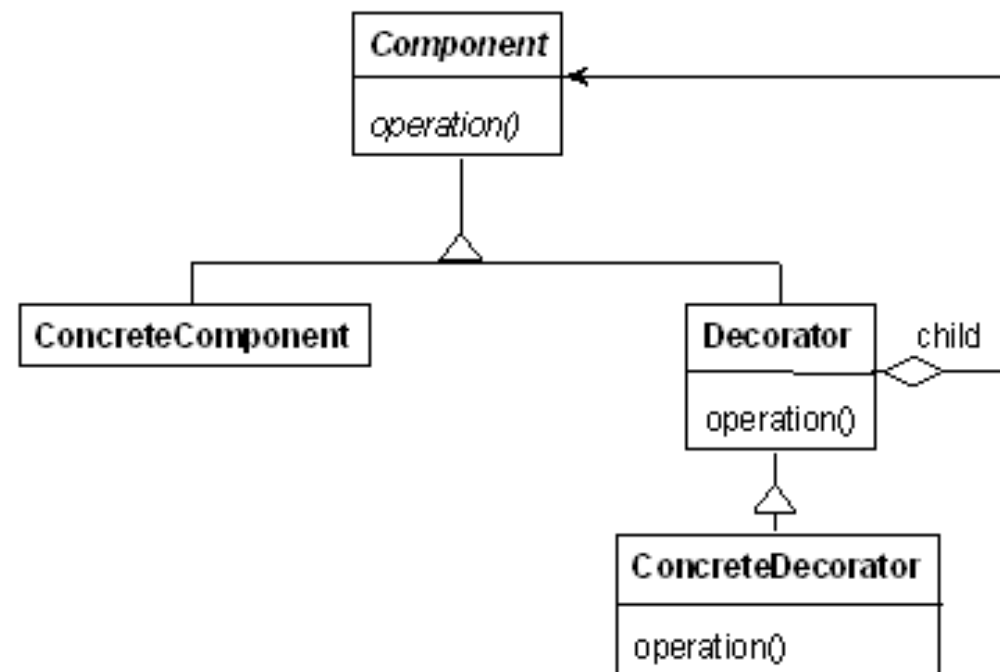
- **Intent**

Augment objects with new responsibilities

- **Applicability**

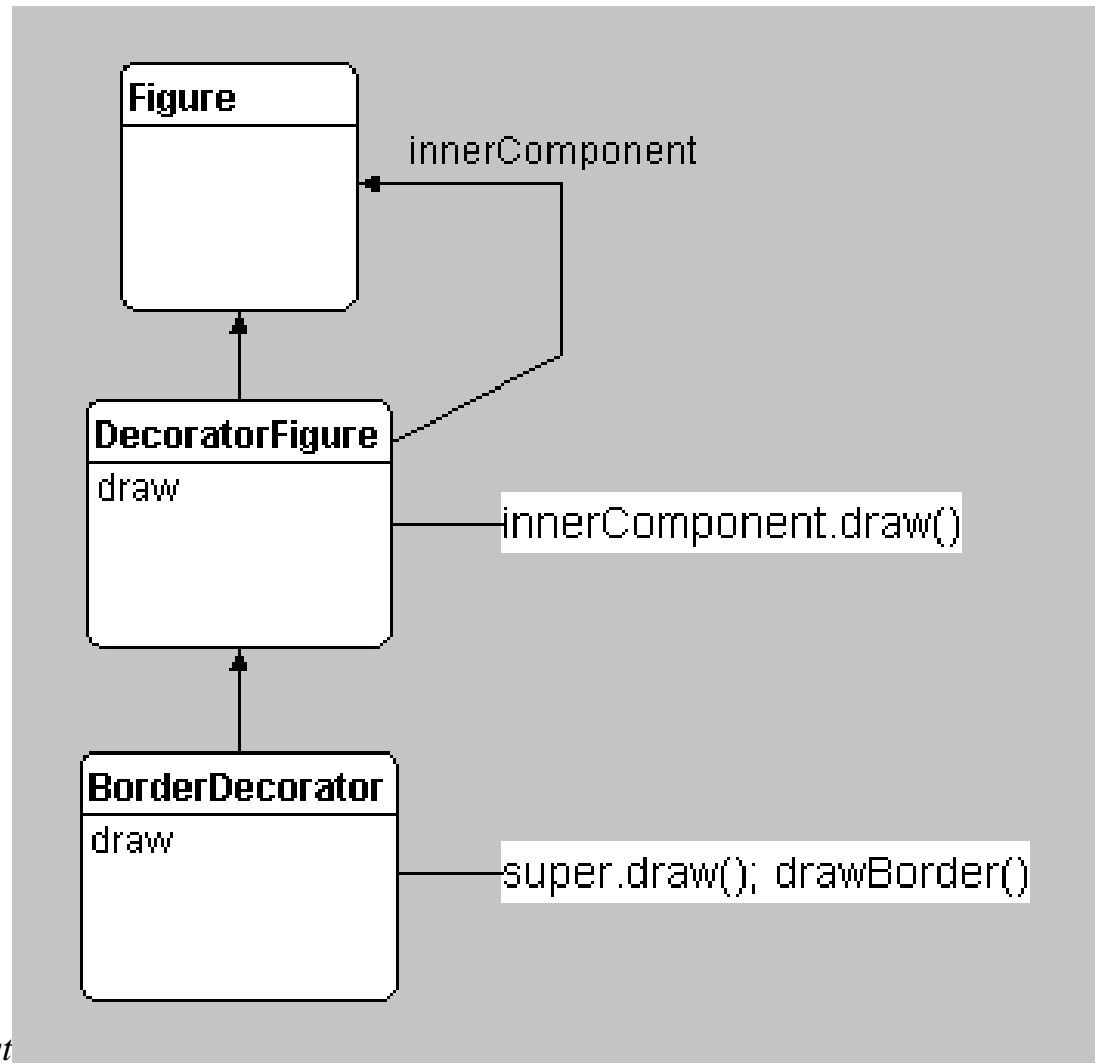
- when extension by subclassing is impractical
- when base class should be extensible with additional properties

- **Structure**



# Abstracciones Fundamentales

- Decorator en JHotDraw



# Abstracciones Fundamentales

- **¿Qué es un dibujo?**

```
class Drawing {  
    add(Figure)  
    remove(Figure)  
  
    draw(Graphics)  
  
    figures()  
    figuresReverse()  
    findFigure(int, int)  
    findFigure(Rectangle)  
  
    sendToBack(Figure)  
    bringToFront(Figure)  
}
```

# Abstracciones Fundamentales

- **Un Drawing es un conjunto de Figures**
- **Una CompositeFigure es un conjunto de Figures**
- **¿Es un Drawing un CompositeFigure?**

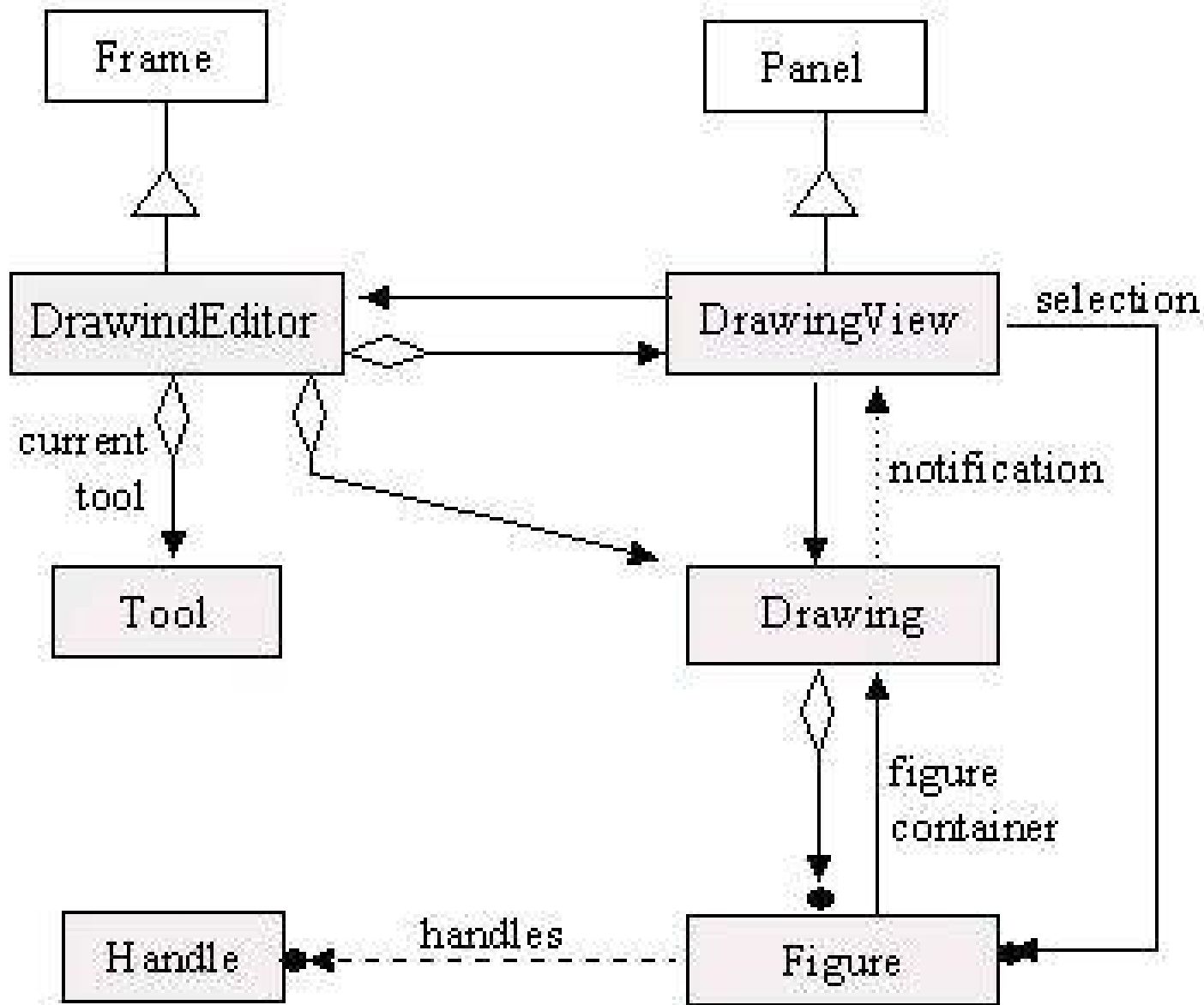
# Abstracciones Fundamentales

- ¿Dónde se dibujan los dibujos (Drawings)?
- ¿Donde está la selección actual?

```
interface DrawingView {  
    setDrawing(Drawing)  
    drawing()  
  
    drawAll(Graphics)  
  
    addToSelection(Figure)  
    selection()  
    selectionCount()  
    toggleSelection(Figure)  
    removeFromSelection(Figure)  
    clearSelection()  
  
    findHandle(int, int)  
    ...  
}
```

## Abstracciones Fundamentales

- **La clase StandardDrawingView es un Panel sobre el que se dibuja el Drawing**
- **Además obtiene los eventos de teclado y ratón y se los pasa al editor para que los maneje**



JHotDraw

**Herramientas**



# JHotDraw

- **Problemas de Diseño**
  - *Abstracciones fundamentales*
  - *Herramientas*
  - *Actualización de pantalla*
  - *Manipulación de Figuras*
  - *Conexiones*
  - *Entorno de ejecución*

# Herramientas

- **DrawingView recibe los eventos del interface de usuario**
- **La respuesta a estos eventos depende de la herramienta activa en ese momento**
- **Objetivos**
  - *Diferentes tipos de editores pueden requerir distintos tipos de herramientas: no debe limitarse su funcionalidad*
  - *Debe ser sencillo definir nuevas herramientas*

# Herramientas

- **Primera aproximación**

```
class DrawingView {
    void mouseDown(...) {
        witch(tool) {
            case SELECTION: ...
                break;
            case CIRCLE: ...
                break;
        }
    }
    void mouseDrag(...) {
        witch(tool) {
            case SELECTION: ...
                break;
            case CIRCLE: ...
                break;
        }
    }
}
```

# Herramientas

- **Es tedioso añadir nuevas herramientas y además se impide su reutilización**
- **La reacción del editor antes los eventos depende del estado en el que se encuentre el editor. ¿Patrón?**

## State

---

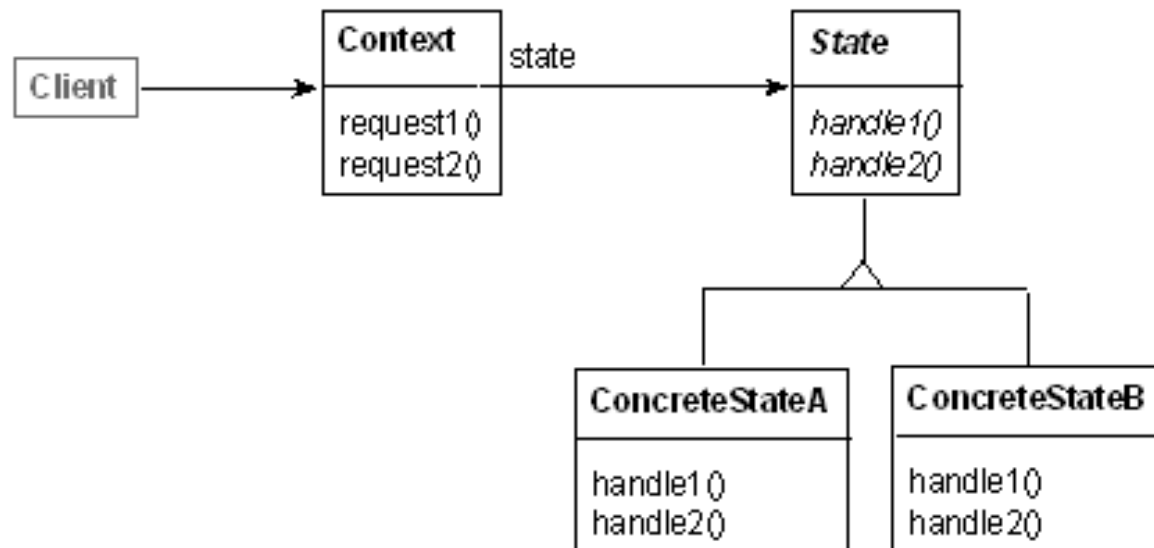
- **Intent**

Allow an object to alter its behavior when its internal state changes

- **Applicability**

- an object's behavior depends on its state
- operations have conditional statements that depend on the object's state

- **Structure**



# Herramientas

- Patrón State en JHotDraw

```
interface Tool
{
    activate()
    deactivate()

    mouseDown(MouseEvent, int, int)
    mouseDrag(MouseEvent, int, int)
    mouseMove(MouseEvent, int, int)
    mouseUp(MouseEvent, int, int)

    keyDown(KeyEvent, int)
}
```

# Herramientas

```
class StandardDrawingView {  
    public void mousePressed(MouseEvent e) {  
        ...  
        tool().mouseDown(e, p.x, p.y);  
    }  
  
    public void mouseDragged(MouseEvent e) {  
        ...  
        tool().mouseDrag(e, p.x, p.y);  
    }  
  
    public void mouseReleased(MouseEvent e) {  
        ...  
        tool().mouseUp(e, p.x, p.y);  
    }  
    ...  
}
```

# Herramientas

- **Herramientas de creación**
  - *Muchas de ellas se diferencian únicamente en la figura a crear*
- **¿Patrones de creación?**

## Factory Method

---

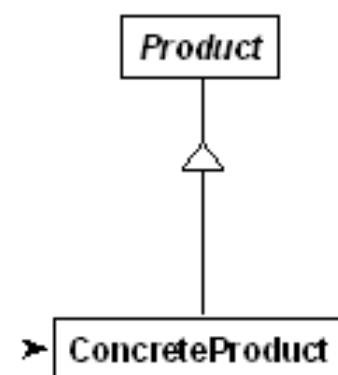
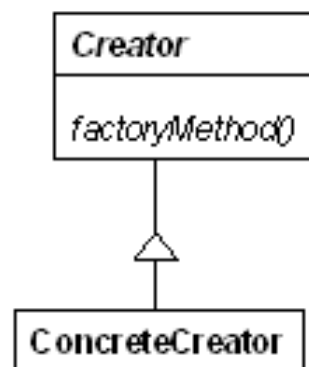
- **Intent**

- define an interface for creating an object, but leave it up to subclasses to decide which object to create

- **Applicability**

- a class can't anticipate the class of the objects it must create
- to connect "parallel" class hierarchies

- **Structure**



# Prototype

---

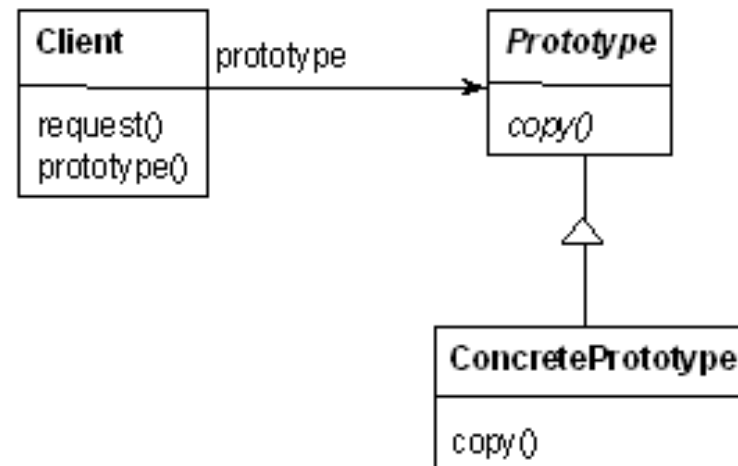
- **Intent**

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

- **Applicability**

- to avoid building a class hierarchy of factories that parallels the class hierarchy of products
- when the classes to instantiate are specified at run-time
- objects need to be created in a preconfigured state

- **Structure**



# Herramientas

- **Herramientas de creación en JHotDraw**
  - *Ventaja Prototype*
    - No hace falta una clase Herramienta por cada figura. Una sola clase puede valer para todas
  - *Ventaja FactoryMethod*
    - La Figura no necesita saber clonarse
- **¿Cual se implementa en JHotDraw?**

# Herramientas

```
public class CreationTool extends AbstractTool
{

private Figure  fPrototype;
...
public CreationTool(DrawingView view, Figure prototype) {
    super(view);
    fPrototype = prototype;
}

protected CreationTool(DrawingView view) {
    super(view);
    fPrototype = null;
}

<sigue>
```

```

public void mouseDown(MouseEvent e, int x, int y) {
    fAnchorPoint = new Point(x,y);
    fCreatedFigure = createFigure();
    fCreatedFigure.displayBox(fAnchorPoint, fAnchorPoint);
    view().add(fCreatedFigure);
}

protected Figure createFigure() {
    if (fPrototype == null)
        throw new HJDError("No protoype defined");
    return (Figure) fPrototype.clone();
}

public void mouseDrag(MouseEvent e, int x, int y) {
    fCreatedFigure.displayBox(fAnchorPoint, new Point(x,y));
}

public void mouseUp(MouseEvent e, int x, int y) {
    if (fCreatedFigure.isEmpty())
        drawing().remove(fCreatedFigure);
    fCreatedFigure = null;
    editor().toolDone();
}
}
}

```

# Herramientas

- **Nuevas herramientas en JHotDraw**
  - *Una vez hecha una nueva figura se desea obtener su herramienta*
  - *Opción 1*
    - **Dar una semántica adecuada al método clone para que sirva como prototipo (generalmente vale con la implementación por defecto)**
    - **De esta manera no hace falta implementar ninguna herramienta nueva**

```
tool = new CreationTool(view(), new RectangleFigure());  
tool = new CreationTool(view(), new EllipseFigure());  
tool = new CreationTool(view(), new CircleFigure());
```

# Herramientas

- **Nuevas herramientas en JHotDraw**

- *Opción 2*

- **Factory Method**

```
class PertFigureCreationTool extends CreationTool {  
  
    public PertFigureCreationTool(DrawingView view) {  
        super(view);  
    }  
  
    protected Figure createFigure() {  
        return new PertFigure();  
    }  
}
```

- **Uso**

```
tool = new PertFigureCreationTool(view());
```

# Herramientas

- **Herramienta de Selección**
  - *Selecciona figuras pulsando un botón*
  - *Selecciona las figuras incluidas en un área*
  - *Mueve las figuras*
  - *Las cambia de tamaño*

# Herramientas

- **Primera aproximación**

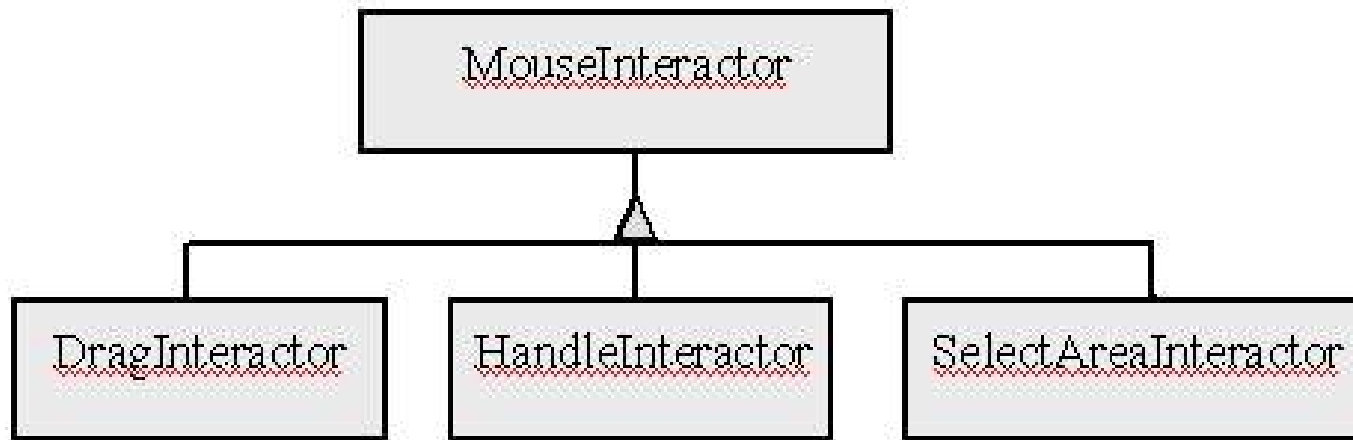
```
class SelectionTool implements Tool {
    public void mouseDrag(MouseEvent, x, y) {
        if (estabaArrastrandoHandle) {
            <código cambiar tamaño>
        } else if (estabaMoviendoFigura) {
            <código mover figura>
        } else if (estabaSeleccionandoArea) {
            <hacer rubberbanding>
        } else ...
    }

    public void mouseUp(MouseEvent, x, y) {
        if (estabaArrastrandoHandle) {
            <código cambiar tamaño>
        } else if (estabaMoviendoFigura) {
            <código mover figura>
        } else if (estabaSeleccionandoArea) {
            <hacer rubberbanding>
        } else ...
    }
}
```

# Herramientas

- **Problemas**
  - *Demasiado código mezclado*
  - *Dificultad de implementación y depuración*
- **La acción a realizar por la herramienta de selección depende del estado en el que se encuentre**
  - *Moviendo*
  - *Cambiando el tamaño*
  - *Seleccionando un área*
- **¿Patrón?**

# Herramientas



```

public class SelectionTool extends AbstractTool
{
private Tool fChild = null;

public void mouseDown(MouseEvent e, int x, int y) {
    Handle handle = findHandle(e.getX(), e.getY());
    if (handle != null)
        fChild = new HandleTracker(view, handle);
    else {
        Figure figure = findFigure(e.getX(), e.getY());
        if (figure != null)
            fChild = new DragTracker(view, f);
        else
            fChild = new SelectAreaTracker(view);
    }
    fChild.mouseDown(e, x, y);
}

public void mouseDrag(MouseEvent e, int x, int y) {
    fChild.mouseDrag(e, x, y);
}

public void mouseUp(MouseEvent e, int x, int y) {
    fChild.mouseUp(e, x, y);
}
}
}

```

JHotDraw

**Actualización en Pantalla**

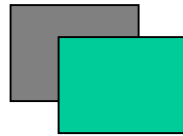


# JHotDraw

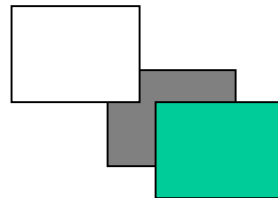
- **Problemas de Diseño**
  - *Abstracciones fundamentales*
  - *Herramientas*
  - *Actualización de pantalla*
  - *Manipulación de Figuras*
  - *Conexiones*
  - *Entorno de ejecución*

## Actualización de pantalla

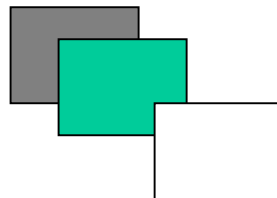
- **Problemas. Supóngase que se mueve una figura desde A hasta B (moveBy)**



- *La figura no puede limpiar A. No sabe lo que hay debajo*



- *La figura no debe dibujarse en B. Puede que haya otra figura encima*



## Actualización de pantalla

- **El DrawingView es el que debe realizar la actualización**
  - *Representa al lugar donde se dibujan las figuras*
  - *Sabe cual es su color de fondo*
  - *Sabe donde están el resto de las figuras y que partes se solapan*
- **Cuando una herramienta manipula una figura ¿Cómo se consigue que el DrawingView la actualice en pantalla?**

## Observer

---

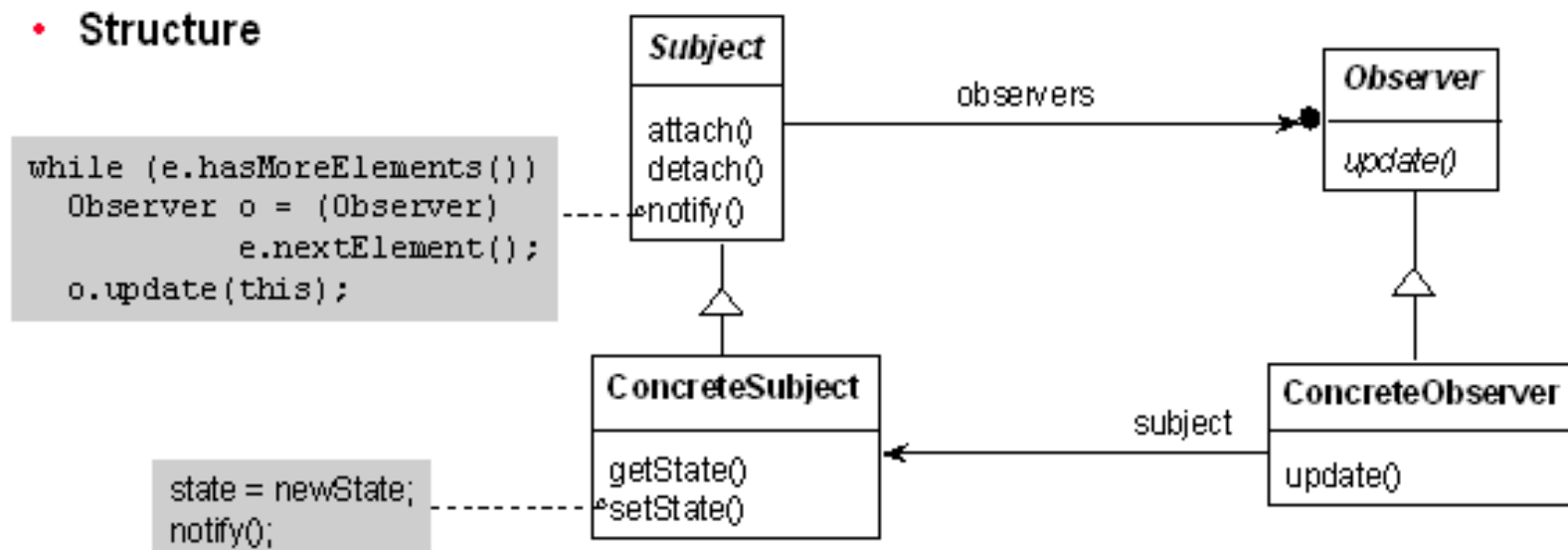
- **Intent**

Define a dependency between objects so that when one object changes state then all its dependents are notified

- **Applicability**

- When a change to one object requires changing others
- Decouple notifier from other objects

- **Structure**



# Actualización en Pantalla

- **Observer en JHotDraw**

```
public interface FigureChangeListener
{
    figureInvalidated(FigureChangeEvent)
    figureChanged(FigureChangeEvent)

    figureRemoved(FigureChangeEvent)
    figureRequestRemove(FigureChangeEvent)
    figureRequestUpdate(FigureChangeEvent)
}
```

# Actualización en Pantalla

- **La figura avisa cuando se necesite repintar un área**

```
class AbstractFigure implements Figure
{
    ...
    addFigureChangeListener(FigureChangeListener fcl) { ...}

    public void moveBy(int dx, int dy) {
        invalidate();
        // Mover la figura...
        changed();
    }

    public void changed() {
        invalidate();
        fListener.figureChanged(new FigureChangeEvent(this));
    }

    public void invalidate() {
        fListener.figureInvalidated(new FigureChangeEvent(this,
            r));
    }
}
```

# Actualización en Pantalla

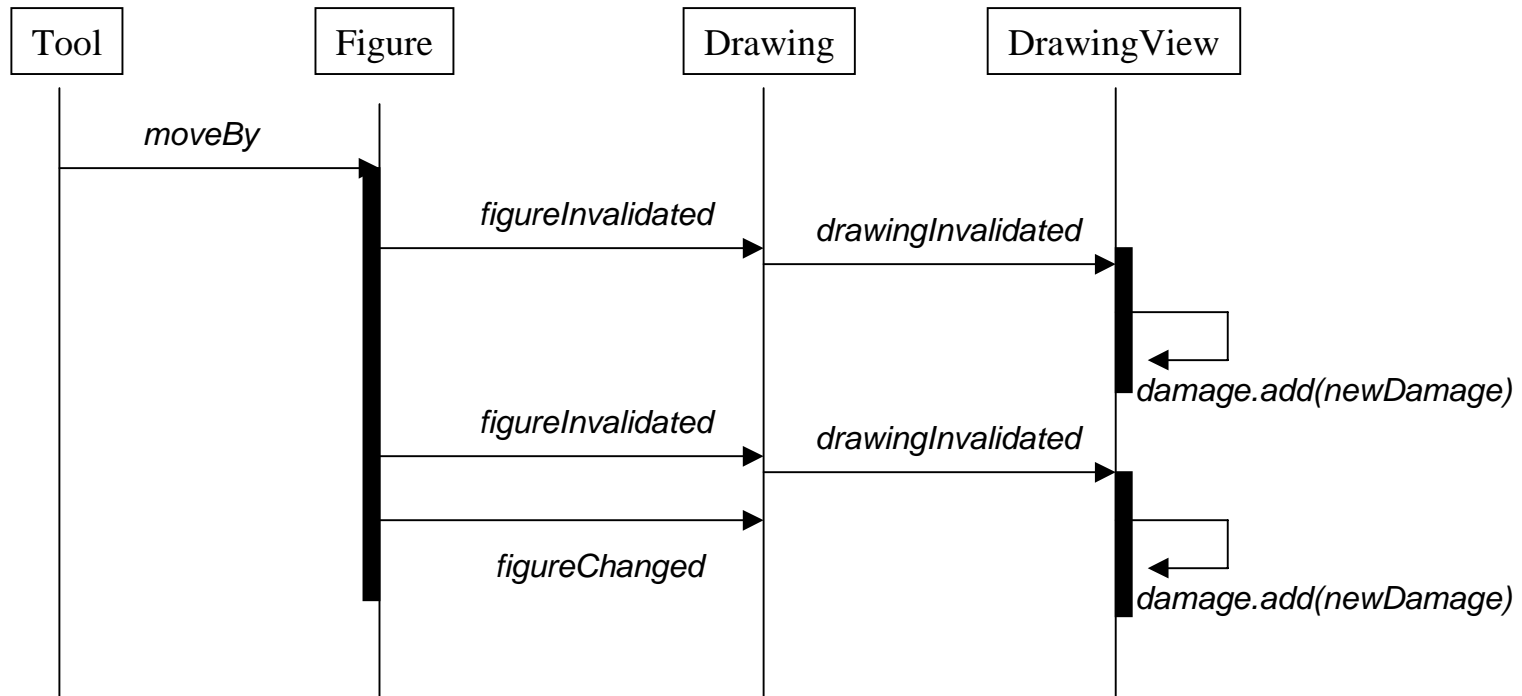
- **Observer en JHotDraw**

- *El Listener de una Figure es su Drawing. Este a su vez utiliza el patrón Observer para notificar a sus DrawingView*

```
public interface DrawingChangeListener
{
    drawingInvalidated(DrawingChangeEvent e);
    drawingRequestUpdate(DrawingChangeEvent e);
}
```

- *El DrawingView se limita a calcular la unión de las áreas a repintar*

# Actualización en Pantalla



## Actualización en Pantalla

- **Cualquier método de la clase figura que necesite un reflejo en pantalla necesita llamar a `invalidate` (`move`, `displayBox`, etc.)**
- **Cuando una nueva figura redefine alguno de estos métodos deberá recordar invocar a *invalidate* y *changed* en la nueva implementación...**
- **¿Solución?**

# Template Method

---

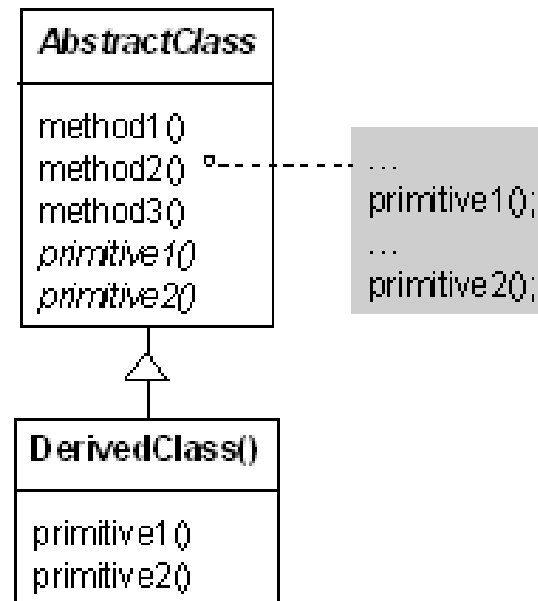
- **Intent**

define skeleton of an algorithm in an operation, deferring some steps to subclasses

- **Applicability**

- implement invariant parts of a design
- define skeleton of an algorithm

- **Structure**



# Actualización en Pantalla

- **Template Method en JHotDraw**

```
public void displayBox(Point origin, Point corner) {  
    willChange();  
    basicDisplayBox(origin, corner);  
    changed();  
}
```

```
public abstract void basicDisplayBox(Point origin, Point  
    corner);
```

```
public void moveBy(int dx, int dy) {  
    willChange();  
    basicMoveBy(dx, dy);  
    changed();  
}
```

```
protected abstract void basicMoveBy(int dx, int dy);
```

# Actualización en Pantalla

- El DrawingView va acumulando áreas que necesitan actualización
- ¿Cuándo hay que actualizarlas en pantalla?
  - *Si se actualizan muy frecuentemente baja el rendimiento*
  - *Si se actualizan con poca frecuencia el usuario percibirá desajustes*
- JHotDraw actualiza a la “velocidad del usuario”
  - *Después de cada evento de usuario actualiza todas las áreas de una sola vez*

```
public void mouseDragged(MouseEvent e) {  
    tool().mouseDrag(e, p.x, p.y);  
    checkDamage();  
}
```
  - *Imprime todas las figuras dentro del área dañada de abajo hacia arriba*

# Actualización de Pantalla

- **Actualización directa o con buffer**
  - ***Con buffer***
    - No requiere borrar el fondo
    - Evita parpadeos
  - ***Sin buffer***
    - Sencillo
    - Consume menos recursos
- **¿Cual utilizar en JHotDraw? Cualquiera**

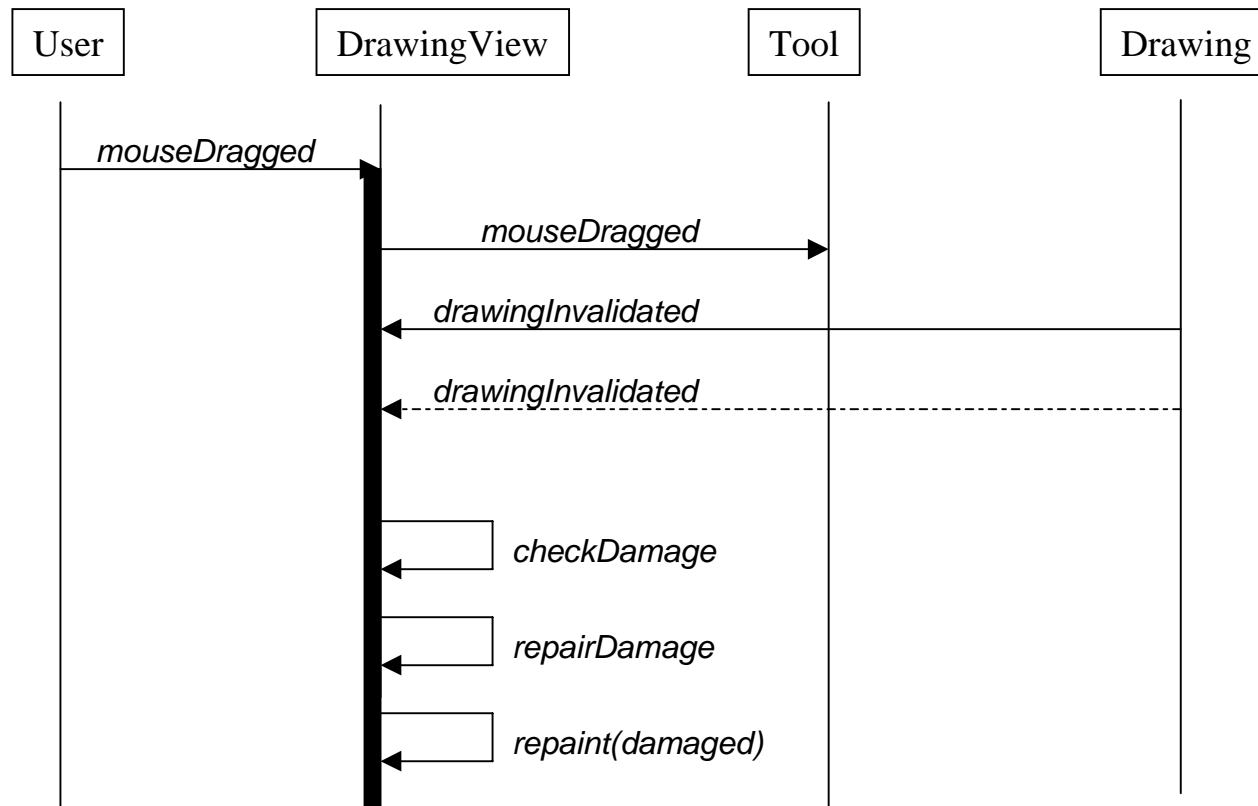
# Actualización de Pantalla

```
public interface Painter {  
    public void draw(Graphics g, DrawingView view);  
}
```

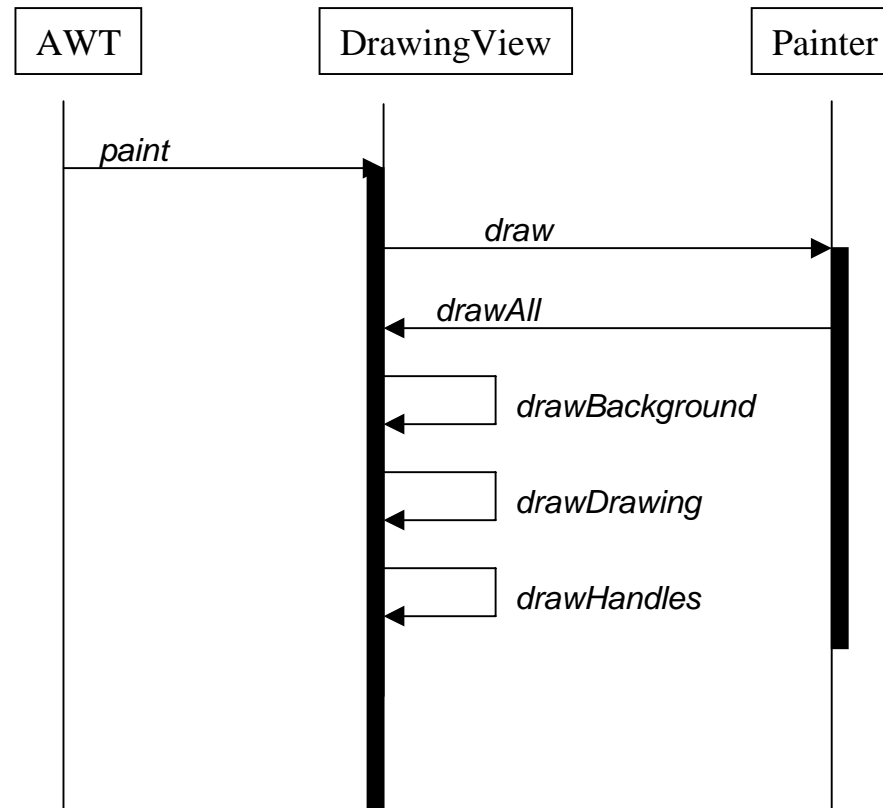
```
public class StandardDrawingView ...  
{  
    private Painter fUpdateStrategy;  
  
    public void setDisplayUpdate(Painter painter) {  
        fUpdateStrategy = painter;  
    }  
  
    public void paint(Graphics g) {  
        fUpdateStrategy.draw(g, this);  
    }  
}
```

...

# Actualización en Pantalla



# Actualización en Pantalla



JHotDraw

## **Manipulación de las figuras**



# JHotDraw

- **Problemas de Diseño**
  - *Abstracciones fundamentales*
  - *Herramientas*
  - *Actualización de pantalla*
  - *Manipulación de Figuras*
  - *Conexiones*
  - *Entorno de ejecución*

# Manipulación de Figuras

- **Requisitos**
  - ***Cambio de tamaño***
  - ***Rotaciones***
  - ***Manipulación de vértices***
  - ***Inicio de conexiones***
  - ***Cambio de inicio y final de la conexión***
  - ***Cualquier otra manipulación específica de una nueva figura...***

# Manipulación de Figuras

- **Problema**

- *Cada figura tendrá una forma distinta de ser manipulada*

- displayBox(x, y, width, height)
- rotate(angle)
- movePoint(index, newX, newY)
- new Connection()
- setConnectionStart (...) / setConnectionEnd(...)
- ???

# Manipulación de Figuras

- **Primera aproximación**

```
class HandleTracker {  
  
    void mouseDrag(x, y)  
    {  
        type = fig.handleType(x, y);  
        if (type == ROTATION) {  
            fig.rotate(...)  
        } else if (type == MOVE) {  
            fig.move(...)  
        } else if (type == CONNECTION) {  
            line.setEnd(...)  
        } else ...  
    }  
}
```

# Manipulación de Figuras

- Otra aproximación

```
class Figure {  
  
    int getHandle(x, y) { ... }  
  
    void invokeStart(int handleIndex, x, y) {...}  
    void invokeStep(int handleIndex, x, y) {...}  
    void invokeEnd(int handleIndex, x, y) {...}  
  
    void drawHandles() {...}  
  
}
```

# Manipulación de Figuras

- **Patrones**
  - *Se desea que la misma herramienta de selección pueda manipular figuras que tienen distintos interfaces*

# Adapter

---

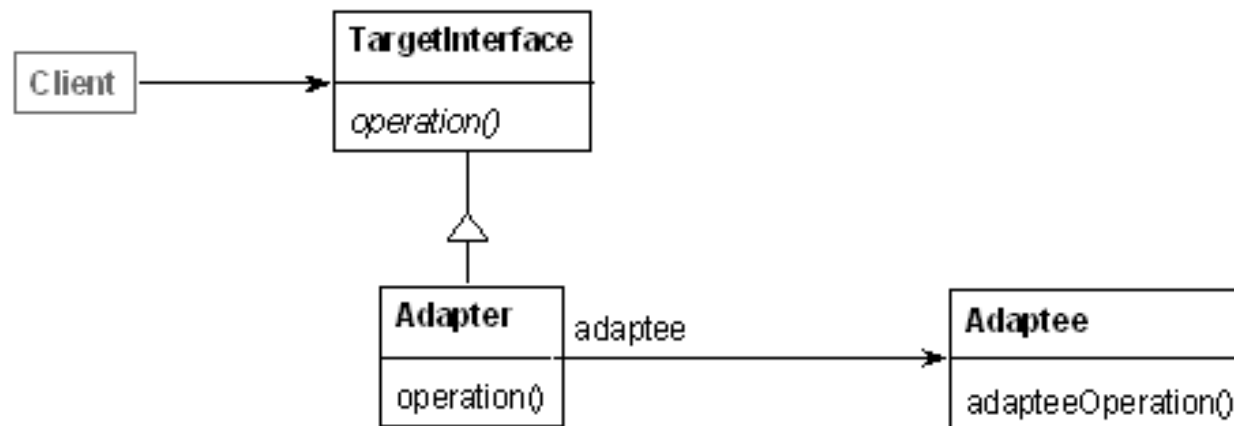
- **Intent**

Convert the interface of a class into another interface.

- **Applicability**

- existing interface doesn't match
- provide reusable abstraction that can work with unrelated classes
- decouple objects and make them more reusable

- **Structure**



# Manipulación de Figuras

- **Clase Handle**

```
interface Handle {  
    invokeStart(int, int, DrawingView)  
    invokeStep(int, int, int, int, DrawingView)  
    invokeEnd(int, int, int, int, DrawingView)  
  
    Figure owner()  
  
    boolean containsPoint(int, int)  
    Point locate()  
    Rectangle displayBox()  
  
    draw(Graphics)  
}
```

```
public class PolyLineHandle extends AbstractHandle {  
    private int fIndex;  
    public PolygonHandle(PolygonFigure owner, int index) {  
        super(owner);  
        fIndex = index;  
    }  
  
    public void invokeStep (int x, int y, int anchorX, int  
        anchorY, DrawingView view) {  
        myOwner().setPointAt(new Point(x, y), fIndex);  
    }  
  
    public Point locate() {  
        return myOwner().pointAt(fIndex);  
    }  
  
    ...  
}
```

```

class TriangleRotationHandle extends AbstractHandle {

public TriangleRotationHandle(TriangleFigure owner) {
    super(owner);
}

public void invokeStart(int x, int y, Drawing drawing)
{
    fCenter = owner().center();
    fOrigin = getOrigin();
}

public void invokeStep (int dx, int dy, Drawing drawing)
{
    double angle = Math.atan2(fOrigin.y + dy - fCenter.y,
                             fOrigin.x + dx - fCenter.x);
    ((TriangleFigure)(owner())).rotate(angle);
}

public void invokeEnd (int dx, int dy, Drawing drawing)
{
    fOrigin = null;
    fCenter = null;
}
}

```

```

class NorthHandle extends LocatorHandle {
public void invokeStep (int x, int y, int anchorX, int
    anchorY, DrawingView view) {
    Rectangle r = owner().displayBox();
    owner().displayBox(
        new Point(r.x, Math.min(r.y + r.height, y)),
        new Point(r.x + r.width, r.y + r.height)
    );
    }
    ...
}

```

```

class WestHandle extends LocatorHandle {
public void invokeStep (int x, int y, int anchorX, int
    anchorY, DrawingView view) {
    Rectangle r = owner().displayBox();
    owner().displayBox(
        new Point(Math.min(r.x + r.width, x), r.y),
        new Point(r.x + r.width, r.y + r.height)
    );
    }
    ...
}

```

# Manipulación de Figuras

- Para obtener los handles de una Figura

```
class Figure {  
    public abstract Vector handles()  
    ...  
}
```

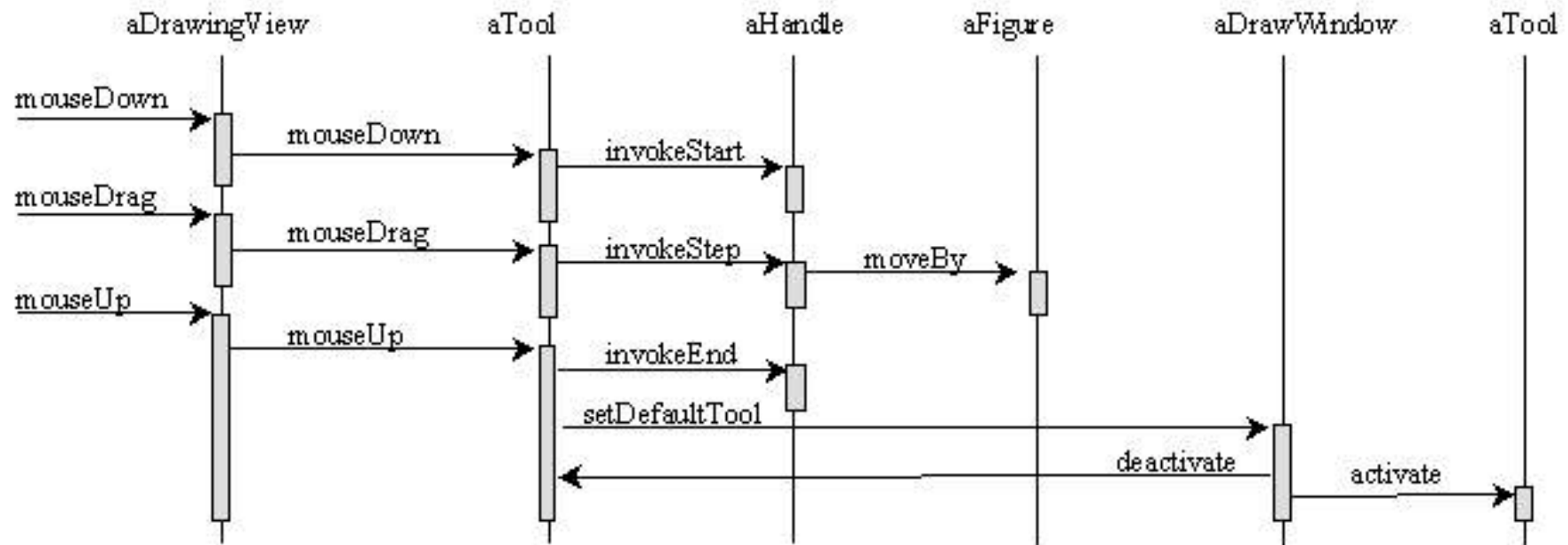
- Al implementar una nueva figura hay que redefinir el método *handles* para devolver los objetos que se desea que manipulen la figura

```
class PolyLineFigure extends AbstractFigure {  
  
    public Vector handles() {  
        Vector v = new Vector(fPoints.size());  
        for (int i = 0; i < fPoints.size(); i++)  
            v.addElement(new PolyLineHandle(this, i));  
        return v;  
    }  
    ...  
}
```

```
class SelectionTool extends AbstractTool {  
  
public void mouseDown(MouseEvent e, int x, int y)  
{  
    Handle handle =  
    view().findHandle(e.getX(),e.getY());  
  
    if (handle != null) {  
        fChild = createHandleTracker(view(), handle);  
    }  
    else {  
        ...  
    }  
  
    fChild.mouseDown(e, x, y);  
}  
}
```

```
public class HandleTracker extends AbstractTool {  
    private Handle fAnchorHandle;  
  
    public HandleTracker(DrawingView view, Handle anchorHandle)  
    {  
        super(view);  
        fAnchorHandle = anchorHandle;  
    }  
  
    public void mouseDown(MouseEvent e, int x, int y) {  
        fAnchorHandle.invokeStart(x, y, view());  
    }  
  
    public void mouseDrag(MouseEvent e, int x, int y) {  
        fAnchorHandle.invokeStep(x, y, fAnchorX, fAnchorY,  
        view());  
    }  
  
    public void mouseUp(MouseEvent e, int x, int y) {  
        fAnchorHandle.invokeEnd(x, y, fAnchorX, fAnchorY,  
        view());  
    }  
}
```

# Manipulación de Figuras



# Manipulación de Figuras

- **Problema**

- *Un handle es responsable de manipular la figura y de saber su propia posición*
- *Por cada handle con acción y posición distinta habría una nueva clase*
- *Problema: handles con misma acción pero distinta posición también requieren distintas clases: NullHandle, ConnectionHandle, GroupHandle, FontSizeHandle, etc.*

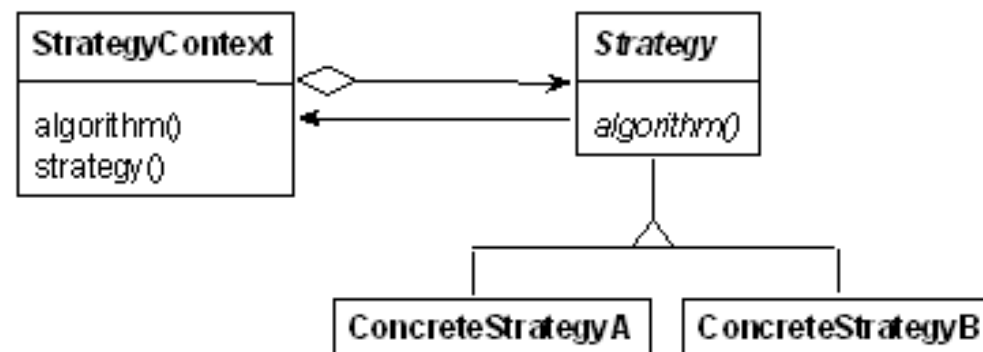
# Manipulación de Figuras

- **Problema**
  - *Hay que separar las dos responsabilidades del handle:*
    - Manipular la figura
    - Saber su colocación en la figura

## Strategy

---

- AKA Policy
- **Intent**
  - define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- **Applicability**
  - when object should be configurable with algorithm to be used
  - need to dynamically reconfigure
- **Structure**



# Manipulación de Figuras

- **Clase Locator**

```
interface Locator {  
    public Point locate(Figure owner);  
}
```

- **Clase LocatorHandle**

```
class LocatorHandle extends AbstractHandle {  
    private Locator fLocator;  
  
    public LocatorHandle(Figure owner, Locator l) {  
        super(owner);  
        fLocator = l;  
    }  
  
    public Point locate() {  
        return fLocator.locate(owner());  
    }  
}
```

# Manipulación de Figuras

- **El Handle delega en el Locator la responsabilidad de saber su posición**
- **Cada Locator representa a una posición de la figura**
- **Todos los Handle que tengan la misma posición (incluso en distintas figuras) podrán compartir el mismo Locator**

```
public class CenterLocatorExample
{
    public Point locate(Figure owner)
    {
        Rectangle r = owner.displayBox();
        return new Point(r.x + r.width / 2,
            r.y + r.height / 2);
    }
}
```

# Manipulación de Figuras

- **Clase RelativeLocator**

```
public class RelativeLocator extends AbstractLocator
{
    double fRelativeX, fRelativeY;

    public RelativeLocator(double relativeX, double
relativeY) {
        fRelativeX = relativeX;
        fRelativeY = relativeY;
    }

    public Point locate(Figure owner) {
        Rectangle r = owner.displayBox();
        return new Point(r.x + (int)(r.width*fRelativeX),
            r.y + (int)(r.height*fRelativeY));
    }
    ...
}
```

# Manipulación de Figuras

```
static public Locator east() {
    return new RelativeLocator(1.0, 0.5);
}
static public Locator north() {
    return new RelativeLocator(0.5, 0.0);
}
static public Locator west() {
    return new RelativeLocator(0.0, 0.5);
}
static public Locator northEast() {
    return new RelativeLocator(1.0, 0.0);
}
static public Locator northWest() {
    return new RelativeLocator(0.0, 0.0);
}
static public Locator south() {
    return new RelativeLocator(0.5, 1.0);
}
static public Locator southEast() {
    return new RelativeLocator(1.0, 1.0);
}
static public Locator center() {
    return new RelativeLocator(0.5, 0.5);
}
```

# Manipulación de Figuras

```
public class PertFigure extends CompositeFigure {  
  
    public Vector handles()  
    {  
        Vector handles = new Vector();  
        handles.addElement(new NullHandle(this,  
            RelativeLocator.northWest()));  
  
        handles.addElement(new NullHandle(this,  
            RelativeLocator.northEast()));  
  
        handles.addElement(new NullHandle(this,  
            RelativeLocator.southWest()));  
  
        handles.addElement(new NullHandle(this,  
            RelativeLocator.southEast()));  
  
        handles.addElement(new ConnectionHandle(this,  
            RelativeLocator.east(), new PertDependency()));  
  
        return handles;  
    }  
}
```

# Manipulación de Figuras

- Los Handles de redimensionado si tienen distintas acciones en distintas posiciones: una clase por Handle
- Pero no hay que rehacerlas para cada nueva figura: reutilización

```
Clase BoxHandleKit {  
    static Handle east(Figure)  
    static Handle north(Figure)  
    static Handle northEast(Figure)  
    static Handle northWest(Figure)  
    static Handle south(Figure)  
    static Handle southEast(Figure)  
    static Handle southWest(Figure)  
    static Handle west(Figure)  
  
    static void addCornerHandles(Figure, Vector)  
    static void addHandles(Figure, Vector)  
}
```

# Manipulación de Figuras

```
public class BoxHandleKit {
    static public Handle south(Figure owner) {
        return new SouthHandle(owner);
    }
    ...
}

class SouthHandle extends LocatorHandle {
    SouthHandle(Figure owner) {
        super(owner, RelativeLocator.south());
    }

    public void invokeStep (int x, int y, int anchorX, int
    anchorY, DrawingView view) {
        Rectangle r = owner().displayBox();
        owner().displayBox(new Point(r.x, r.y),
            new Point(r.x + r.width, Math.max(r.y, y)));
    }
}
```

# Manipulación de Figuras

- **Uso de BoxHandleKit**

```
public class RoundRectangleFigure {
    public Vector handles() {
        Vector handles = new Vector();
        BoxHandleKit.addHandles(this, handles);

        handles.addElement(new RadiusHandle(this));

        return handles;
    }
    ...
}
```

# Manipulación de Figuras

- **Extensión de JHotDraw: incorporación de Handles a una nueva figura**
  - *Redefinir el método handles*
  - *Si se desean los Handles de redimensionado usar la clase **BoxHandleKit***
  - *Si la acción y la localización del nuevo Handle ya la contempla alguna clase: combinarlos*

```
handles.addElement(new NullHandle(this,
    RelativeLocator.northWest()));
```
  - *Si la acción es específica de la figura redefinir **LocatorHandle***
  - *Si la localización es específica de la figura redefinir **AbstractHandle***

JHotDraw

## Conexión de Figuras

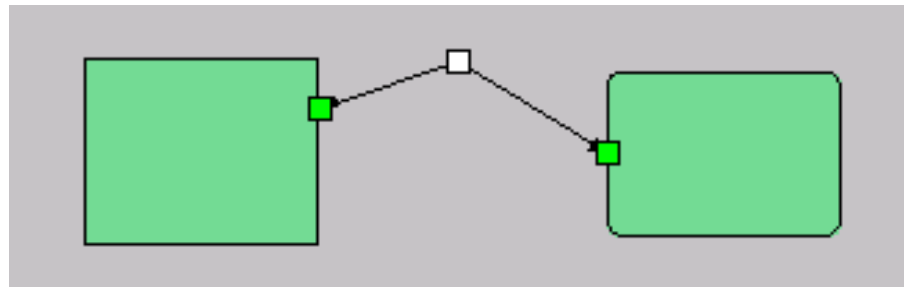


# JHotDraw

- **Problemas de Diseño**
  - *Abstracciones fundamentales*
  - *Herramientas*
  - *Actualización de pantalla*
  - *Manipulación de Figuras*
  - *Conexiones*
  - *Entorno de Ejecución*

# Conexión

- **Requisitos**
  - *Independencia de la implementación de las Figuras*
  - *Control de conexiones inválidas*
  - *Conexión en cualquier punto de la figura o en puntos concretos*
  - *Reacción ante los cambios en las figuras conectadas*
  - *Distintos tipos de conexiones (recta, segmentos, arcos...)*
  - *Las conexiones deben poder manipularse (handles)*



# Conexión

- Una ConnectionFigure es una Figure que relaciona dos figuras

```
interface ConnectionFigure extends Figure
{
    endPoint(int, int)
    endPoint()
    startPoint(int, int)
    startPoint()

    canConnect(Figure, Figure)
    ...
}
```

# Conexión

- Una ConnectionFigure puede tener varios segmentos

```
...
splitSegment(int, int)
joinSegments(int, int)

setPointAt(Point, int)
Point pointAt(int)

pointCount()
...
```

# Conexión

- **Posicionamiento Relativo**
  - ***Al crear una conexión entre dos figuras ¿Cuales son las coordenadas de sus extremos?***
    - El centro de cada figura
    - La mínima distancia entre sus áreas
    - Los puntos de corte de la recta que una los centros y el displayBox
    - Los puntos cardinales de la figura...
  - ***Al mover alguna de las figuras ¿Cuales son los nuevos puntos de conexión? ¿Los mismos de antes?***
  - ***En definitiva ¿qué estrategia se sigue para localizar los extremos de la conexión?***

# Conexión

- **Un Connector es un punto de conexión de la figura**
- **Las ConnectionFigure están ancladas a un Connector en cada uno de sus extremos**
- **Un Connector sabe cuales son las coordenadas en las cuales debe realizarse la conexión con la Figura**

```
public interface Connector {  
    boolean containsPoint(int, int);  
    Rectangle displayBox();  
  
    Point findStart(ConnectionFigure connection)  
    Point findEnd(ConnectionFigure connection)  
  
    void draw(Graphics);  
    Figure owner();  
}
```

# Conexión

- En la **ConnectionFigure** en vez de dar coordenadas absolutas (**startPoint, endPoint**) es más flexible dar un **Connector** a cada extremo

```
class ConnectionFigure
{
    connectEnd(Connector)
    connectStart(Connector)

    Connector end()
    Connector start()
    ...
}
```

# Conexión

- **La clase Figura es la que sabe en que puntos puede realizarse una conexión**

```
interface Figure
{
    ...
    boolean canConnect();
    Connector connectorAt(int x, int y);

    void connectorVisibility(boolean isVisible);
}
```

# Conexión

- Ejemplo

```
class AbstractFigure implements Figure
{
    ...
    Connector connectorAt(int x, int y)
    {
        return new ChopBoxConnector(this);
    }
}
```

# Conexión

- **Ejemplo: ChopBoxConnector**

```
class ChopBoxConnector extends AbstractConnector
{
public Point findStart(ConnectionFigure connection)
{
    Rectangle r = connection.end().displayBox();
    Point middle = new Point(r.x + r.width/2,
        r.y + r.height/2);

    r = owner().displayBox();

    // Calcula la intersección
    return Geom.angleToPoint(r, (Geom.pointToAngle(r,
        middle)));
}
...
}
```

# Conexión

- **Un connector debe saber su posición, tamaño y dibujo**
- **Un mismo tipo de connector situado en distintas partes de la figura ¿debe dar lugar a distintas clases?**

# Conexión

```
class LocatorConnector extends AbstractConnector
{
private Locator fLocator;

public LocatorConnector(Figure owner, Locator l) {
    super(owner);
    fLocator = l;
}

protected Point locate(ConnectionFigure connection) {
    return fLocator.locate(owner());
}

public Rectangle displayBox() {
    Point p = fLocator.locate(owner());
    return new Rectangle(p.x - SIZE / 2, p.y - SIZE / 2,
        SIZE, SIZE);
}

public boolean containsPoint(int x, int y) {
    return displayBox().contains(x, y);
}
```

# Conexión

```
public class NodeFigure extends TextFigure {  
  
    private Vector connectors() {  
        if (fConnectors == null)  
            createConnectors();  
        return fConnectors;  
    }  
    private void createConnectors() {  
        fConnectors = new Vector(4);  
        fConnectors.addElement(new LocatorConnector(this,  
            RelativeLocator.north()) );  
  
        fConnectors.addElement(new LocatorConnector(this,  
            RelativeLocator.south()) );  
  
        fConnectors.addElement(new LocatorConnector(this,  
            RelativeLocator.west()) );  
  
        fConnectors.addElement(new LocatorConnector(this,  
            RelativeLocator.east()) );  
    }  
}
```

# Conexión

```
<NodeFigure>
...
private Connector findConnector(int x, int y) {
    // return closest connector
    long min = Long.MAX_VALUE;
    Connector closest = null;
    Enumeration e = connectors().elements();
    while (e.hasMoreElements()) {
        Connector c = (Connector)e.nextElement();
        Point p2 = Geom.center(c.displayBox());
        long d = Geom.length2(x, y, p2.x, p2.y);
        if (d < min) {
            min = d;
            closest = c;
        }
    }
    return closest;
}
...
```

# Conexión

- **Resumen**

- *Al implementar una nueva figura*

- Si no se redefine `conectorAt` se obtiene la conexión estándar mediante un `ChopBoxConnector` (`AbstractFigure`)
    - Si se desean indicar puntos de conexión fijos se debe redefinir `conectorAt` y devolver el conector adecuado (`NodeFigure`)
    - Si se desea cambiar la apariencia de los conectores se deberá derivar de `LocatorConnector` y redefinir `draw`
    - Si se desea cambiar además el posicionamiento habrá que derivar del interface `Locator`

## Conexión

- **Al mover una de las dos figuras que participan en la conexión ¿cómo se actualiza la ConnectionFigure?**

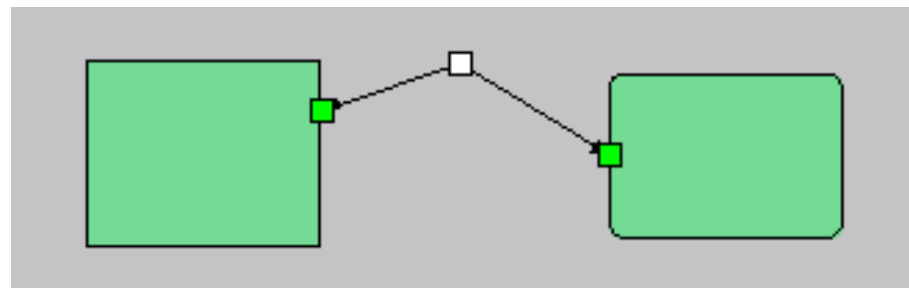
# Conexión

- **Implementaciones standard de JHotDraw relacionadas con la conexión de figuras**
  - *LineConnection*
  - *ElboxConnection*
  - *ConnectionTool*

# Conexión

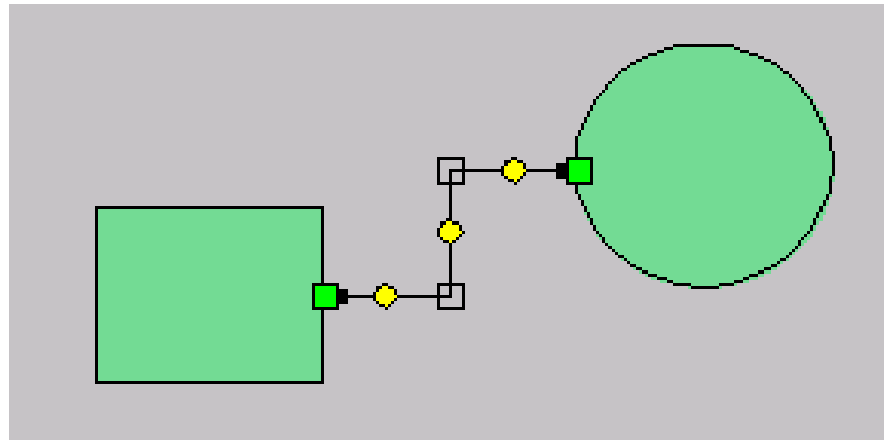
- **LineConnection** es la figura de conexión más habitual
- Se implementa mediante una **PolyLineFigure**
- Permite indicar los adornos de extremos de la conexión

```
public interface LineDecoration
{
    public abstract void draw(Graphics g, int x1, int y1,
        int x2, int y2);
}
```



# Conexión

- **ElbowConnection** implementa una **FigureConnection** formada a base de segmentos verticales y horizontales



## Conexión

- **ConnectionTool** es la herramienta que permite crear las conexiones
- Vale para cualquier **ConnectionFigure**. No es necesario crear una herramienta distinta para cada tipo de conexión

```
tool = new ConnectionTool(view(), new  
    LineConnection());
```

```
tool = new ConnectionTool(view(), new  
    ElbowConnection());
```

# Conexión

- **Resumen. Creación de nuevos tipos de conexiones**
  - ***Si simplemente se desean conectar figuras sin ningún tipo de semántica usar `CreationTool` con el tipo de conexión deseada (`LineConnection`, `ElbowConnection`, ...)***
  - ***Si se desea algún tipo de acción asociada a la conexión (validación, conexión, desconexión, etc.) derivar del tipo de conexión deseado y redefinir los métodos adecuados***

```
public class PertDependency extends LineConnection {
    public boolean canConnect(Figure start, Figure end) {
        return (start instanceof PertFigure && end instanceof
            PertFigure);
    }
    public void handleConnect(Figure source, Figure target) {
        if (source.hasCycle(target))
            setAttribute("FrameColor", Color.red);
        else {
            target.addPreTask(source);
            source.addPostTask(target);
            source.notifyPostTasks();
        }
    }
}
```

# Conexión

- **Resumen. Creación de nuevos tipos de conexiones**
  - *Si se desea crear un nuevo tipo de conexión (arcos) se deberá implementar partiendo de `ConnectionFigure`*
  - *En este caso se podrá seguir usando `ConnectionTool` si la nueva conexión se crea como las habituales (pinchar, arrastrar, conectar)*
  - *En caso contrario habrá que crear su propia `ConnectionTool` (derivando de `AbstractTool`)*

JHotDraw

**Entorno de Ejecución**



# JHotDraw

- **Problemas de Diseño**
  - *Abstracciones fundamentales*
  - *Herramientas*
  - *Actualización de pantalla*
  - *Manipulación de Figuras*
  - *Conexiones*
  - *Entorno de Ejecución*

## Entorno de Ejecución

- **Un editor JHotDraw consiste en**
  - *DrawingView*
  - *Drawing*
  - *Tools*
  - *Menus*
  - *Botones (paletas)*
- **Se desea un coordinador de todos estos objetos**
- **Este coordinador debe valer tanto para aplicaciones como para Applets**
- **¿Patrón?**

# Mediator

---

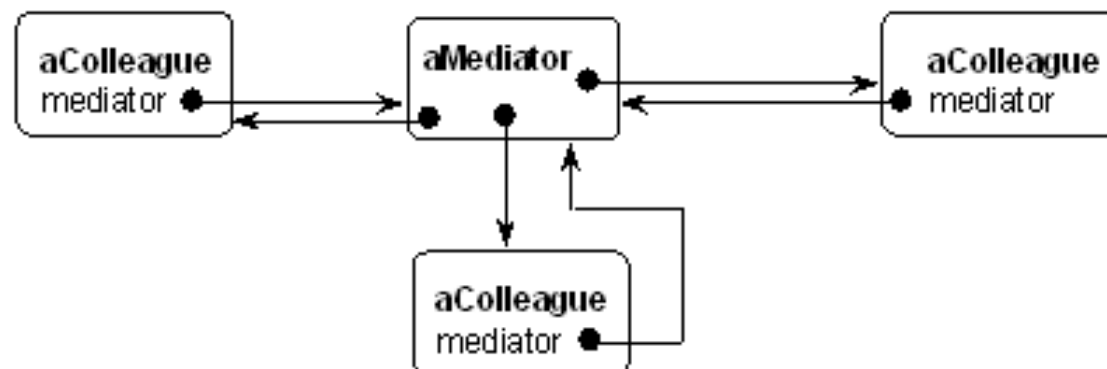
- **Intent**

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly.

- **Applicability**

- reusing an object is difficult because it refers and communicates with many other objects
- a set of objects communicate in well-defined but complex ways
- a behavior that is distributed between several classes should be customizable without a lot of subclassing

- **Structure (object)**



## Entorno de Ejecución

- **El editor se basa en DrawingEditor independientemente de si el programa principal es un aplicación o Applet**

```
public interface DrawingEditor
{
    DrawingView view();
    Drawing      drawing();
    Tool         tool();
    void         toolDone();

    void         selectionChanged(DrawingView view);
    void         showStatus(String string);
}
```

## Entorno de Ejecución

- **JHotDraw ya viene con una implementación por defecto del DrawingEditor tanto para aplicaciones como para Applets**
  - *DrawApplication*
  - *DrawApplet*
- **Se encargan de crear las herramientas, menús y paletas mas comunes**
- **Problema: el usuario quiere crear un nuevo editor adaptando alguno de los elementos anteriores. ¿Patrón?**

# Entorno de Ejecución

- **Factory Method**
- **Las dos clases definen varios factory methods con comportamiento por defecto que puede ser redefinido**
  - createEditMenu()
  - createFileMenu()
  - createFontMenu()
  - createFontSizeMenu()
  - createFontStyleMenu()
  - createMenus(MenuBar)
  - createSelectionTool()
  - createStatusLine()
  - createToolButton(String, String, Tool)
  - createToolPalette()
  - createTools(Panel)
  - ...

```

public class PertApplication extends DrawApplication {
    static private final String PERTIMAGES = "/images/";

protected void createTools(Panel palette) {
    super.createTools(palette);

    Tool tool = new TextTool(view(), new TextFigure());
    palette.add(createToolButton(IMAGES+"TEXT", "Text Tool",
    tool));

    tool = new PertFigureCreationTool(view());
    palette.add(createToolButton(PERTIMAGES+"PERT", "Task
    Tool", tool));

    tool = new ConnectionTool(view(), new PertDependency());
    palette.add(createToolButton(IMAGES+"CONN", "Dependency
    Tool", tool));

    tool = new CreationTool(view(), new LineFigure());
    palette.add(createToolButton(IMAGES+"Line", "Line Tool",
    tool));
}

public static void main(String[] args) {
    PertApplication pert = new PertApplication();
    pert.open();
}

```

## Entorno de Ejecución

- **Muchas operaciones pueden invocarse desde varios sitios**
  - *Menús*
  - *Atajos de teclado*
  - *Botones*
- **¿Duplicar el código?**
- **Muchas operaciones son las mismas para todos los editores (cut, paste, delete, align, order,...)**
- **¿Repetir en cada nuevo editor?**
- **¿Solución?**

# Command

---

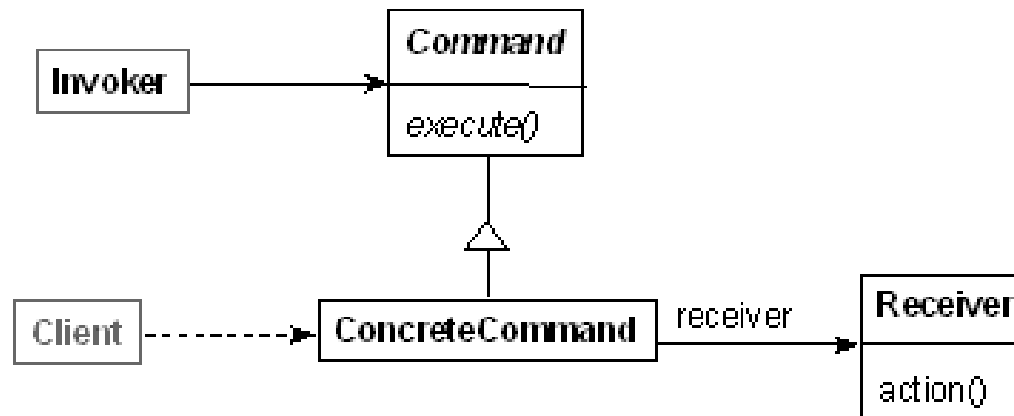
- **Intent**

Encapsulate a request as an object, thereby letting you parameterize objects with different requests

- **Applicability**

- parameterize objects by action to be performed
- queue or log requests
- support undo

- **Structure**



# Entorno de Ejecución

- **Command en JHotDraw**

```
public abstract class Command
{
    public abstract void execute();

    public boolean isExecutable()
    {
        return true;
    }
}
```

- **¿Implementa JHotDraw el Undo?**

## Entorno de Ejecución

- De esta manera las operaciones más habituales se pueden reutilizar directamente
  - *CutCommand, CopyCommand, PasteCommand*
  - *DeleteCommand, DuplicateCommand*
  - *GroupCommand, UngroupCommand*
  - *ChangeAttributeCommand*
  - *AlignCommand*
  - *etc.*

# Entorno de Ejecución

- **JHotDraw incluye componentes que ejecutan Commands (CommandMenu, CommandChoice, ...)**

```
protected Menu createEditMenu() {
    CommandMenu menu = new CommandMenu("Edit");
    menu.add(new CutCommand("Cut", fView), new
    MenuShortcut('x'));
    menu.add(new CopyCommand("Copy", fView), new
    MenuShortcut('c'));
    menu.add(new PasteCommand("Paste", fView), new
    MenuShortcut('v'));
    menu.addSeparator();
    menu.add(new DeleteCommand("Delete", fView));
    menu.addSeparator();
    menu.add(new GroupCommand("Group", fView));
    menu.add(new UngroupCommand("Ungroup", fView));
    menu.addSeparator();
    menu.add(new SendToBackCommand("Send to Back", fView));
    menu.add(new BringToFrontCommand("Bring to Front",
    fView));
    return menu;
}
```

JHotDraw

**Conclusiones**



## Conclusiones

- **Los patrones establecieron la estructura del diseño**
- **No siempre es inmediato saber que patrón utilizar**
  - *A veces varios son aplicables (prototype, Factory method)*
  - *Implementar los patrones es fácil; lo difícil es saber cuando y dónde hay que implementarlos*
- **El diseño sigue siendo iterativo**
- **Los patrones ayudan en la documentación (pattlets). Para explicar la clase locator basta con decir que es un Strategy**

## Conclusiones

- **HotDraw es similar en SmallTalk y Java**
- **Los patrones de diseño y Java son una combinación muy potente**
  - *“...pero ningún lenguaje puede reducir la importancia del diseño”. Kent Beck y Erich Gamma*