7. Framework Architectures

1. Motivation and Fundamental Concepts
2. Revisiting Object-Oriented Analysis, Design, and Implementation
3. Design Patterns
4. Pipes & Filter Architectures
5. Event-based Architectures
7. Framework Architectures
   7.1 Example: Making the Java Collections Framework Type Safe
   7.2 Example: The JHotDraw Framework
   7.3 The Template Method Pattern for Extending Algorithms
8. Component Architectures
Frameworks

Frameworks are software components designed for reuse. They
- offer prefabricated components as building blocks and design patterns as blueprints for the architecture,
- capture architectural and implementation artifacts that do not vary,
- defer the variant parts to application-specific logic,
- manage the control-flow.

Frameworks usually have reversed control-flow (the Hollywood Principle: “Don’t call us, we’ll call you”). This makes them difficult for beginners to understand.

Frameworks: Examples

Examples of frameworks:
- ET++ (portable application framework and class library for C++)
- Java Swing (GUI Components)
- HotDraw / JHotDraw (Drawing Framework in Smalltalk / Java)
- Java Collections Framework (Java 2): Lists, Maps, Sets, ...
- Java Activation Framework (used to assign application components to open / edit MIME contents; used by JavaMail)
- WAM (Werkzeug, Automaten, Material) - Framework (=> SWT Univ. Hamburg)
- ...

Class libraries are sometimes also considered as frameworks:
- Microsoft Foundation Classes (MFC)
- Borland Object Window Library (OWL)
- C++ Standard Template Library (STL)
Frameworks: Interaction with Application Logic

Frameworks interact with application logic through defined connection points:

- **Events**: Application-specific logic can register an interest in events generated by the framework. When an event occurs, the application logic is notified.
  
  Example: Application logic registers for `MouseMotion` events generated by an applet.

- **Hooks**: Application-specific classes can be attached to the framework’s hooks. When the control flow reaches a certain point in the framework, the application-specific logic is invoked.
  
  Example (Java Runtime Environment: Java VM hook): Application logic that shall be invoked when the VM stops (`System.exit()`) can be registered at the JRE with `System.addShutdownHook(Thread t);`

- **Subclassing**: An application-specific class replaces a generic framework class. Usually, the framework must use the `Factory Method Pattern` to defer instantiation of the class to allow this.
  
  Example: Application class inherits from Swing class (`JFrame`) and adds behavior and components (buttons, text areas).

Note: Events and hooks are quite similar concepts.

Frameworks: Benefits and Liabilities

**Benefits of frameworks:**

- Frameworks help create an application that is developed in a timely manner and customized to the user’s requirements, and still benefits from the framework’s maturity with regard to robustness and stability.

**Liabilities:**

- Cost of learning and understanding framework abstractions, interactions and limitations. Most frameworks are rather complex pieces of software at a high level of abstraction.

- Frameworks offer customization facilities, but they can impose restrictions and may require special programming techniques, especially if the application needs to perform functionality slightly out of the framework’s defined scope.

[Kaiser01]
Frameworks: Evaluation

Before using a framework, it should be evaluated as a whole:

- Target applications it addresses
- Components and structure
- Fundamental design patterns
- Programming techniques
- Control-flow within the framework

Further Reading

[GHJV95]  E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements Of Reusable Object-Oriented Software, Addison-Wesley, 1995.


Chapter 7

7. Framework Architectures

7.1 Example: Making the Java Collections Framework type safe

7.2 Example: The JHotDraw Framework

7.3 The Template Method Pattern for Extending Algorithms

The Java Collection Framework

These interfaces are defined by the Java 2 Collections Framework. A large number of classes implement these interfaces.

Extending a Framework:
It would not at all make sense to change the code of the existing Java classes:
- Code is owned by Sun
- Changed code is not supported
- After bug fixes / new JDK releases changes have to be re-applied
Making Java Collection Typesafe (1)

**Problem:** Java collections are heterogeneous, but are not type-safe (no type parameters) in Java versions prior to 5.0:

It is possible to add a Banana to a collection which shall only contain Students. The error is encountered when the objects are retrieved from the collection and a type-cast exception is thrown.

```java
List list = new ArrayList();  // intended as list of Students
Student s = new MasterStudent();
list.add(s);
Banana b = new Chiquita();
list.add(b);
...
Student lastStudent = (Student)list.get(1);  // does not raise an error, although the error occurs here
```

Better with Java Platform 5 which offers parameterized classes. But: for compatibility reasons, collection types can still be used the old way.

Making Java Collection Typesafe (2)

Solution through framework extension: using the decorator pattern.

Build decorators for Collections, Lists, Maps and Sets that receive a type parameter and check all method parameters for type conformity.

```java
public class SafeList implements List {
    private List list;
    private Class type;
    public SafeList (List list, Class type) {
        this.list = list;
        this.type = type;
    } // constructor
    ... (continued on next slide)
```
Making Java Collection Typesafe (3)

```java
... protected void checkType (Object o) {
    if (! type.isInstance (o))
        throw new ClassCastException (type.getName () + " expected");
} // checkType
public Object set (int index, Object element) {
    checkType (element);
    return list.set (index, element);
} // set
} // class SafeList
```

7. Framework Architectures

7. Framework Architectures

7.1 Example: Making the Java Collections Framework type safe

7.2 Example: The JHotDraw Framework

7.3 The Template Method Pattern for Extending Algorithms
Example: The JHotDraw Framework

The JHotDraw framework is targeted at applications for drawing technical and structured graphics – such as network layouts and Gantt diagrams.

Originally developed in Smalltalk by Kent Beck and Ward Cunningham, JHotDraw was one of the first software development projects explicitly designed for reuse and labelled a framework.

Understanding JHotDraw

JHotDraw Relevant Classes:

- Frame
- Panel
- DrawApplication
- StandardDrawingView
- Tool
- Figure
- Handle
- Java
- AWT classes
- JHotDraw classes

[Kaiser01]

[www.iam.unibe.ch01]
Understanding JHotDraw (2)

Using the Framework

The framework is used by extending its functionality by domain-specific / application-specific classes.
Design Patterns in JHotDraw

JHotDraw makes extensive use of Design Patterns [GHJV95]:
- Model-View-Controller
- Composite
- Strategy
- State
- Template Method
- Decorator
- Factory Method
- Prototype

General observation: Extensibility of frameworks calls for extensive use of design patterns.

JHotDraw Patterns: Model View Controller (1)

Model View Controller Pattern (see first lesson)
- Model
  - Figures (and their attributes: FillColor, Position)
  - Drawing (figure container)
- View
  - DrawingView (clipping view of a window)
  - DrawingWindow
- Controller
  - Tools to manipulate the model
JHotDraw Patterns: Model View Controller (2)

JHotDraw Patterns: Composite

A figure is composed of several figures. Composite figure as well as child figures are treated uniformly. Operations performed on either of the figures will exert a common behavior on all figures (scale, rotation, move).

Example: A composite figure consisting of 5 subordinate figures - the famous “Das-ist-das-Haus-vom-Nikolaus”.

composition  scaling rotation
Composite Pattern

Contributions:
- Allow nested structures of arbitrary depth.
- Uniformly handle all objects forming a composite.

<table>
<thead>
<tr>
<th>Figure</th>
<th>RectangleFigure</th>
<th>LineFigure</th>
<th>GraphicalCompositeFigure</th>
</tr>
</thead>
<tbody>
<tr>
<td>draw () : void</td>
<td>shape</td>
<td>...</td>
<td>delegate methods to all contained Shapes</td>
</tr>
</tbody>
</table>

JHotDraw Patterns: Strategy

It has advantages to separate layout algorithms from objects to be laid out (see also Java AWT, Swing). Here, the Strategy Pattern is used.

Every layouter (Java Swing / AWT: layout manager) can be attached to a composite figure rendering it.
JHotDraw Patterns: State

**Goal:** Externalize the state of a tool.

A tool can subdivide its state. This is necessary to provide means to make a tool operate in different modes (states).

Example: A drawing tool's operation may differ whether the tool has already been used before: This is obvious for toggling tools:
- zoom tool: toggle between “zoom in” and “zoom out”
- selection tool: toggle between “select border” and “select text”

E.g., text field:
- set the cursor with 1 click
- select a word with 2 clicks
- select the whole line with 3 clicks

JHotDraw Patterns: Template Method

**Goal:** Define the skeleton of an algorithm in the framework, deferring application-specific steps or additions to the application-specific classes.

From the framework’s point of view, connecting figures via a line is always the same operation. From the application’s point of view, it is quite a difference depending on the semantics of the connection.

To join these two views, the Template Method pattern is used. The framework’s algorithm for connecting figures (its invariant parts) are placed in one class (LineConnection), exposing the variant parts of the algorithm in empty methods that subclasses can overwrite.

Examples:

- Petri net connections
- Class diagram connections
JHotDraw Patterns: Decorator

Goal: Add specific visualization to generic visualization.

Example: Add borders and shadows to a figure.

Note: A problem of the decorator pattern is that the type of the decorated object is difficult to obtain if

- it is not made public by access methods like `getDecoratedType()` (the Java `instanceof` operator does not work) and
- no separate list of all figures (including the decorated) exists.

JHotDraw Patterns: Factory Method

Goal: Abstract from the concrete classes to be instantiated, which helps creation of customized components.

This is used in JHotDraw to keep menus and tool structure flexible. Factory methods in the `DrawApplication` class are called `createMenus()` and `createTools()`. The customized application class (MyDrawApplication) inherits from `DrawApplication` and overwrites the `createMenus()` and `createTools()` methods.
JHotDraw Patterns: Prototype

Each tool is initialized with an instance (a prototype) of the figure it is meant to create. When creating a new figure, the tool clones the prototype.

7. Framework Architectures

7.1 Example: Making the Java Collections Framework Type Safe
7.2 Example: The JHotDraw Framework
7.3 The Template Method Pattern for Extending Algorithms
Template Method Pattern

The Template Method Pattern is used for structuring adaptable algorithms. It is quite comparable to frameworks with respect to:

- Control flow ("Hollywood Principle")
- Extending functionality at definite points

The algorithm's control flow is structured in an abstract class. Subclasses define / alter the variable parts of the algorithm, although subclasses can alter the variant parts of an algorithm, they cannot change the control flow in general and other invariant parts [Vli98].

Example: Template Method Pattern (1)

Scenario:
You are in a team developing a word processor. When the user saves a document, the word processor opens a file, writes document header information and the document's content and then closes the file.

This is simple. The algorithms is as follows:

```java
public void write (String filename) throws Exception {
    OutputStream os = new FileOutputStream (filename);
    PrintWriter out = new PrintWriter (os);
    out.write (getStandardEncoding ().getBytes ());
    out.write (getHeader ().getBytes ());
    out.write (doc.getContent ().getBytes ());
    out.close ();
} // write
```
Example: Word Processor Writer

Now your word processor shall support several formats: BigWord95, BadWord97, WurdForNonspellers2000. These formats all contain a document header and the document's content, but the representation format is different.

"... The quick brown fox jumps over the lazy dog..."

BigWord95 format  BadWord97 format  WurdForNonspellers 2000 format  WurdForNonspellers XP format

Word Processor Writer: Bad Solution (1)

**Bad solution:** You define a new write() method for each format (writeBigWord95(), ...) and then switch between the methods.

```java
public void write (String filename, String format) {
    if ("BigWord95".equals (format))
        writeBigWord95 (filename) ;
    else if ("BadWord97".equals (format))
        writeBadWord97 (filename) ;
    else if ("WurdForNonspellers2000".equals (format))
        writeWurdForNonspellers2000 (filename) ;
    else
        ; // what to do?
}

public void writeBigWord95 (String filename) { ... }
public void writeBadWord97 (String filename) { ... }
public void writeWurdForNonspellers2000 (String filename) { ... }
```
Word Processor Writer: Bad Solution (2)

Why is it a bad solution?

- You will find that all `writeXXX()` methods basically have the same structure (open file, write header, write content, close file).
- You cannot anticipate all formats. Therefore, every time a new format is to be supported, you have to change the class.

Word Processor Writer: Good Solution

Better solution:

Instead, use a template method to extract the invariant part (the algorithm’s structure) to an abstract class and move the variant parts into subclasses.

One will probably use another creational pattern (factory method, prototype) to add flexibility to the creation of the concrete writer.
Template Method Pattern Applied (1)

```
public interface FormatWriter {
    public void write (Document doc, String filename) throws Exception;
}
// interface FormatWriter

public abstract class AbstractWriter implements FormatWriter {
    public void write (Document doc, String filename) throws Exception {
        OutputStream os = new FileOutputStream (filename);
        PrintWriter out = new PrintWriter (os);
        out.write (getStandardEncoding ().getBytes ());
        doWriteHeader (doc, out);
        doWriteContent (doc, out);
        out.close ();
    }
    // write
    protected abstract void doWriteHeader (Document doc, PrintWriter out);
    protected abstract void doWriteContent (Document doc, PrintWriter out);
}
// abstract class AbstractWriter
```

control flow defined here
format-specific algorithmic parts defined by subclasses

Template Method Pattern Applied (2)

```
public class BigWord95Writer extends AbstractWriter {
    protected void doWriteHeader (Document doc, PrintWriter out) {
        ...
    }
    // doWriteHeader

    protected void doWriteContent (Document doc, PrintWriter out) {
        ...
    }
    // doWriteContent
}
// class BigWord95Writer
```

format-specific implementation
format-specific implementation
Implications of Template Method Pattern (1)

Identified problem:
The specific writer class should not be bound to (named) in the generic code for saving a document.

```java
FormatWriter writer = null;
if ("BigWord95".equals (format))
    writer = new BigWord95Writer () ;
else if ("BadWord97".equals (format))
    writer = new BadWord97Writer () ;
else ...
writer.write (doc, filename) ;
...
```

Somewhere in the code, the specific writer class has to be chosen.
This part has to change whenever a new FormatWriter is added!

Implications of Template Method Pattern (2)

Solutions:
Remove binding to specific classes by either
- applying the Factory Method pattern
  - moves details into sub-classes
- or by mapping formats to classes textually and use dynamic class loading
  - moves details into configuration files
Implications of Template Method Pattern (3)

Factory Method pattern:

- **FormatWriter**
  - `+write() : void`

- **AbstractWriterCreator**
  - `+getWriter(formatName : String) : FormatWriter`

- **BigWord95Writer**
  - `+write() : void`

- **WriterCreator**
  - `+getWriter(formatName : String) : FormatWriter`

provided by framework

provided by the application

Implications of Template Method Pattern (4)

Factory Method pattern (2):

```java
abstract WriterCreator class

public abstract class AbstractWriterCreator {
    public abstract FormatWriter getWriter (String formatName) ;
}

class AbstractWriterCreator

public class WriterCreator extends AbstractWriterCreator {
    public FormatWriter getWriter (String formatName) throws Exception {
        if (formatName.equals("BigWord95"))
            return new BigWord95Writer () ;
        throw new Exception ("Format not identified.");
    }

class WriterCreator
```

abstract WriterCreator class

specific WriterCreator class
Implications of Template Method Pattern (5)

Dynamic lookup: A mapping of format names to java classes is defined in a (textual) configuration:

```java
Hashtable formatWriters = new Hashtable();
formatWriters.put("bigWord95", softarch.Bigword95Writer.class);
formatWriters.put("badWord97", software.BadWord97Writer.class);
...
public FormatWriter getWriter(String formatName)
    throws Exception
{
    Class writerClass = (Class)formatWriters.get(formatName);
    FormatWriter writer = (FormatWriter)writerClass.newInstance();
    return writer;
} // getWriter
```

APPENDIX: Exploring JHotDraw

The following slides’ content is mainly taken from an article by Wolfram Kaiser [Kaiser01].

The Design Patterns’ class diagrams are taken from the book “Design Patterns: Elements of Reusable Object-Oriented Software” [GHJV95].
JHotDraw: Package Organization

All JHotDraw classes and interfaces are organized in packages according to their functionality.

- The package `CH.ifa.draw.framework` contains mostly interface definitions of core component requirements – their responsibility, functionality, and interoperation.
- One can find a standard implementation of these interfaces in `CH.ifa.draw.standard`.
- Additional functionality is located in `CH.ifa.draw.figures` and `CH.ifa.draw.contrib`.
- A skeleton of an application or applet is defined in `CH.ifa.draw.application` (or `CH.ifa.draw.applet` to create an applet).

[Kaiser01]

Development Process Using JHotDraw (1)

1. Create your own graphical figures and symbols for your application.

A lot of predefined figures are already available:

- `CH.ifa.draw.figures.RectangleFigure`
- `CH.ifa.draw.figures.LineFigure`
- `CH.ifa.draw.figures.LineConnection`
- `CH.ifa.draw.figures.ElbowConnection`
- `CH.ifa.draw.contrib.GraphicalCompositeFigure`
- `CH.ifa.draw.contrib.DiamondFigure`
- `CH.ifa.draw.contrib.PolygonFigure`

See `CH.ifa.draw.figures` and `CH.ifa.draw.contrib` packages for more figures.

One can refine their behavior by subclassing them and overriding some methods, such as `draw()`, to customize the graphical representation in the diagram. Typically, the graphical figures should correspond and somehow relate to the objects used in your application.

[Kaiser01]
Development Process Using JHotDraw (2)

2. Develop your own tools to create figures and manipulate them according to application requirements.

JHotDraw offers tools:

- CreationTool
- ConnectionTool
- CH.ifa.draw.figures.ScribbleTool (to create and manipulate a PolyLine)
- SelectionTool (to select figures)
- CH.ifa.draw.figures.TextTool (to create and manipulate a TextFigure)

Subclassing those tools and overriding methods like `mouseUp()` and `mouseDown()` allows you to specify your own application interaction and perform the tasks your application needs – such as manipulating the object defined by your application.

[Kaiser01]

Development Process Using JHotDraw (3)

3. Create the actual GUI and integrate it into your application.

Unsurprisingly, JHotDraw already includes a basic application skeleton: Either a basic DrawApplication, a MDI_DrawApplication with support for several internal frames, or a DrawApplet.

You can define your own menus by refining `createMenus()`, `createFileMenu()`, and so on, and plug in new tools by overriding the `createTools()` method in a subclass.

A complete GUI is created when you instantiate your application at runtime and call the `open()` method.

⇒ Class to extend:

- CH.ifa.draw.application.DrawApplication (simple window) or
- CH.ifa.draw.contrib.MDI_DrawApplication (multiple windows).

[Kaiser01]
Creating a new DrawApplication

```java
package softarch;
import javax.swing.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.contrib.*;
import CH.ifa.draw.util.*;
public class MyDrawApplication extends MDI_DrawApplication {
    public MyDrawApplication () {
        super ("My Draw Application");
        // constructor
    }
    public static void main (String [] args) {
        MyDrawApplication app = new MyDrawApplication () ;
        app.open () ;
        // main
    }
}
```

Adding Rectangles & Tool (1)

The rectangle figure is provided by JHotDraw: RectangleFigure

The tool that creates and manipulates spatial figures (rectangles, circles, ...) is the CreationTool. The tool takes an instance of the figure it is meant to create and manipulate (see Prototype pattern).

```
Tool tool = new CreationTool (view(), new RectangleFigure());
```

Prototype Pattern

![Prototype Pattern Diagram](image-url)
Adding Rectangles & Tool (2)

In order to add the tool to the application, overwrite the DrawApplication's createTools() method (see Factory Method Pattern):

```java
protected void createTools (JToolBar palette) {
    super.createTools (palette) ;
    Tool tool = new CreationTool (view (), new RectangleFigure ()) ;
    palette.add (createToolButton (IMAGES + "RECT", "Rectangle", tool)) ;
} // createTools
```

Adding Line Connections & Tool

The line connection figure (LineConnection) is already provided by JHotDraw. The tool to create connections is the ConnectionTool. For a line connection, add to the createTools() method:

```java
... tool = new ConnectionTool (view (), new LineConnection ()) ;
    palette.add (createToolButton (IMAGES + "CONN", "Line Connection", tool)) ;
... 
```

For an elbow connection, use the ElbowConnection instead:

```java
... tool = new ConnectionTool (view (), new ElbowConnection ()) ;
    palette.add (createToolButton (IMAGES + "OCONN", "Elbow Connection", tool)) ;
... 
```
Changing a Connection’s Appearance (1)

Create a subclass of the existing class `LineConnection` called `AssociationConnection`.

Assign `LineDecorators` to `startDecoration` and `endDecoration`. `ArrowTip` is an existing `LineDecorator`.

```java
package softarch;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.figures.*;
import java.awt.Color;

public class AssociationConnection extends LineConnection {
    public AssociationConnection () {
        setStartDecoration (null);
        ArrowTip arrowtip = new ArrowTip (0.35, 20.0, 20.0);
        arrowtip.setFillColor (Color.white);
        arrowtip.setBorderColor (Color.black);
        setEndDecoration (arrowtip);
    }
    // constructor
    } // AssociationConnection
```

Changing a Connection’s Appearance (2)

**Note:** The Decorator pattern does not apply here. Why not?

- The decorator does not implement the component's interfaces (`Line` and `LineDecorator` are distinct interfaces). It is an Aggregation relationship (a `Line` has two `LineDecorators` and not an inheritance relationship).
- Decorators cannot be nested (only a single decorator can be set).

### Decorator Pattern

- Component
- Operation() (Decorator)
- ConcreteComponent
  - Operation() (ConcreteDecoratorA)
  - addedState
  - Component
- ConcreteDecoratorB
  - Operation() (ConcreteDecoratorB)
  - AddedBehavior() (Decorator)
  - Component

Software Architectures: Chapter 7 – Frameworks
Changing a Connection’s Appearance (3)

The method `createTools()` of the class `DrawApplication` must reflect the new connection class:

```java
... tool = new ConnectionTool (view (),
    new AssociationConnection () );
palette.add (createToolButton (IMAGES + "LINE",
    "Line Connection", tool));
...```

Grouping Figures

Figures can be grouped using the `GraphicalCompositeFigure` (see Composite Pattern). A `GraphicalCompositeFigure` is usually rendered as a rectangle.

Example: create class `GroupedFigure`, a figure that consists of two `GraphicalCompositeFigure`s, of which the upper one is colored yellow and contains a `TextFigure`, and the lower one is colored red.

The `TextFigure` will be assigned a text `Figure X` with `X` starting from 1.
Grouping Figures  (3)

```java
package softarch;
import java.awt.*;
import java.util.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.contrib.*;
import CH.ifa.draw.util.*;

public class GroupedFigure extends GraphicalCompositeFigure {
    private GraphicalCompositeFigure firstRectangle;
    private GraphicalCompositeFigure secondRectangle;

    public GroupedFigure() {
        super(new RectangleFigure());
    }

    // constructor

    static int counter = 1;

    public void initialize() {
        removeAll();
        TextFigure nameFigure = new TextFigure() {
            public void setText(String newText) {
                super.setText(newText);
                update();
            }
        };

        nameFigure.setText("Figure " + (counter++));
        nameFigure.setFont(new Font("Helvetica", Font.BOLD, 12));
    }
}
```

Grouping Figures  (4)

```java
// hook method called to initialize a figure
protected void initialize() {
    // start with an empty Composite
    removeAll();

    // create a TextFigure responsible for the name
    // if the name is changed, the size of the figure is updated
    TextFigure nameFigure = new TextFigure() {
        public void setText(String newText) {
            super.setText(newText);
            update();
        }
    };

    // set the figure name initially to "Figure X"
    nameFigure.setText("Figure " + (counter++));
    nameFigure.setFont(new Font("Helvetica", Font.BOLD, 12));
}
```
Grouping Figures (5)

```java
// add the TextFigure to the first rectangle
firstRectangle = new GraphicalCompositeFigure();
firstRectangle.add(nameFigure);
firstRectangle.getLayouter().setInsets(new Insets(0, 4, 0, 0));
firstRectangle.setAttribute("FillColor", Color.yellow);
add(firstRectangle);

// add a second rectangle
secondRectangle = new GraphicalCompositeFigure();
secondRectangle.getLayouter().setInsets(new Insets(4, 4, 4, 0));
secondRectangle.setAttribute("FillColor", Color.red);
add(secondRectangle);

// chain initialize()
super.initialize();
} // initialize
```

Grouping Figures (6)

```java
// return default handles on all four edges for this figure
public Vector handles () {
    Vector handles = new Vector();
    handles.addElement(new NullHandle(getPresentationFigure(), RelativeLocator.northWest()));
    handles.addElement(new NullHandle(getPresentationFigure(), RelativeLocator.northEast()));
    handles.addElement(new NullHandle(getPresentationFigure(), RelativeLocator.southWest()));
    handles.addElement(new NullHandle(getPresentationFigure(), RelativeLocator.southEast()));
    return handles;
} // handles
```
Grouping Figures (7)

Add the tool to create the GroupedFigures:

```java
... Tool tool = new CreationTool (view (), new GroupedFigure ()) ;
palette.add (createToolButton (IMAGES + "RECT",
                                "GroupedFigure", tool)) ;
...```

Changing Texts with a New Selection Tool (1)

The selection tool can be replaced by a refined selection tool that activates a TextTool on double-click on a TextFigure. The TextTool then edits the TextFigure's content.

```java
package softarch ;
import CH.ifa.draw.framework.* ;
import CH.ifa.draw.contrib.* ;
import CH.ifa.draw.figures.* ;
import java.awt.event.* ;
import java.awt.* ;
// Delegate mouse selection to a specific TextTool if the figure selected
// inside a CompositeFigure is a TextFigure
public class DelegationSelectionTool extends CustomSelectionTool {
    private TextTool textTool ;
    public DelegationSelectionTool (DrawingView view) {
        super (view) ;
        textTool = new TextTool (view, new TextFigure ()) ;
    } // constructor
```
Changing Texts with a New Selection Tool (2)

```java
protected void handleMouseDoubleClick (MouseEvent e, int x, int y) {
    Figure figure = drawing ().findFigureInside (e.getX (), e.getY ()) ;
    if ((figure != null) && (figure instanceof TextFigure)) {
        textTool.activate () ;
        textTool.mouseDown (e, x, y) ;
    }
}
```

```java
protected void handleMouseClick (MouseEvent e, int x, int y) {
    deactivate () ;
}
```

```java
public void deactivate () {
    super.deactivate () ;
    if (textTool.isActivated ())
        textTool.deactivate () ;
}
```

```java
} // class DelegationSelectionTool
```

Changing Texts with a New Selection Tool (3)

The change of selection tool must be reflected in the DrawApplication. The DrawApplication class provides a factory method createSelectionTool() that must be overwritten by the MyDrawApplication class (see Factory Method Pattern before).

```java

defined Tool createSelectionTool () {
    return new DelegationSelectionTool (view ()) ;
}
```

...
Adding Menus and Commands (1)

Add new menu with menu items can be added to the menu bar by overwriting the `DrawApplication`'s `createMenus()` method. The menu includes commands to be executed on the selected figures (see Command Pattern).

```java
protected void createMenus (JMenuBar mb) {
    super.createMenus (mb) ;
    mb.add (createMyMenu ()) ;
} // createMenus

protected JMenu createMyMenu () {
    CommandMenu menu = new CommandMenu ("Change") ;
    menu.add (new PaintItBlackCommand ("PaintItBlack", view ())) ;
    return menu ;
} // createMyMenu
```

Adding Menus and Commands (2)

Add new menu with menu items can be added to the menu bar by overwriting the `DrawApplication`'s `createMenus()` method. The menu includes commands to be executed on the selected figures (see Command Pattern).

**Command Pattern**

- **Client**
- **Invoker**
- **Command**
- **ConcreteCommand**
- **Receiver**
- **ConcreteReceiver**

**Commands and Menus**

- **Application**
- **Menu**
- **Menuitem**
- **Document**

**Execute()** will perform an action on the current document.
Adding Menus and Commands (2)

```java
package softarch;
import java.util.*;
import java.awt.Color;
import CH.ifa.draw.util.Command;
import CH.ifa.draw.framework.*;

public class PaintItBlackCommand extends Command {
    private DrawingView fView;

    public PaintItBlackCommand (String name, DrawingView view) {
        super (name);
        fView = view;
    }

    public void execute () {
        Iterator figures = fView.selection ().iterator () ;
        while (figures.hasNext ()) {
            Figure f = (Figure)figures.next () ;
            f.setAttribute ("FillColor", Color.black) ;
        }
        fView.checkDamage () ;
    }

    public boolean isExecutable () {
        return fView.selectionCount () > 0 ;
    }
}
```

Adding Menus and Commands (3)

```java
public class PaintItBlackCommand {

    public void execute () {
        Iterator figures = fView.selection ().iterator () ;
        while (figures.hasNext () ) {
            Figure f = (Figure)figures.next () ;
            f.setAttribute ("FillColor", Color.black) ;
        }
        fView.checkDamage () ;
    }

    public boolean isExecutable () {
        return fView.selectionCount () > 0 ;
    }
}
```