Java Drawing Tool Design Document

Valid for JDT framework version: 6 (Maven variant)

1 Introduction

This document describes briefly the most relevant aspects of the Java Drawing Tool (JDT) framework. This framework enables developers (students) to create concrete drawing applications in terms of instantiating this framework. The main purpose of the framework is of educational matter: Students shall apply the design principles and patterns taught during the lectures.

This document gives a broad overview only, does not attempt to explain all details. For particular aspects, the accompanying descriptions of exercises need to be considered. If that does not help then you have to resort to the source code.

The heart of the JDT framework consists of a set of framework interfaces, abstract classes, and, sometimes, default implementations for the interfaces. These artifacts can be found in package `ch.bfh.due1.jdt.framework`.

Credits:

- The framework is based on D. Gruntz's JDraw framework (www.gruntz.ch) which in turn is based on JHotDraw 6.

2 JDT Concepts

This is a (simple) framework for graphical editors. An e*ditor* supports several *views*. Every view has one associated *sheet*. A sheet is a drawing area into which *shapes* can be drawn. Shapes have *handles* for manipulation. Shapes can be created and manipulated by *tools*. Tools themselves are created by tool factories.

The editor's operations are carried out by *commands*. Commands are kept in a history list of the editor's *command handler*.

Figure 1 shows the domain model illustrating the core concept.

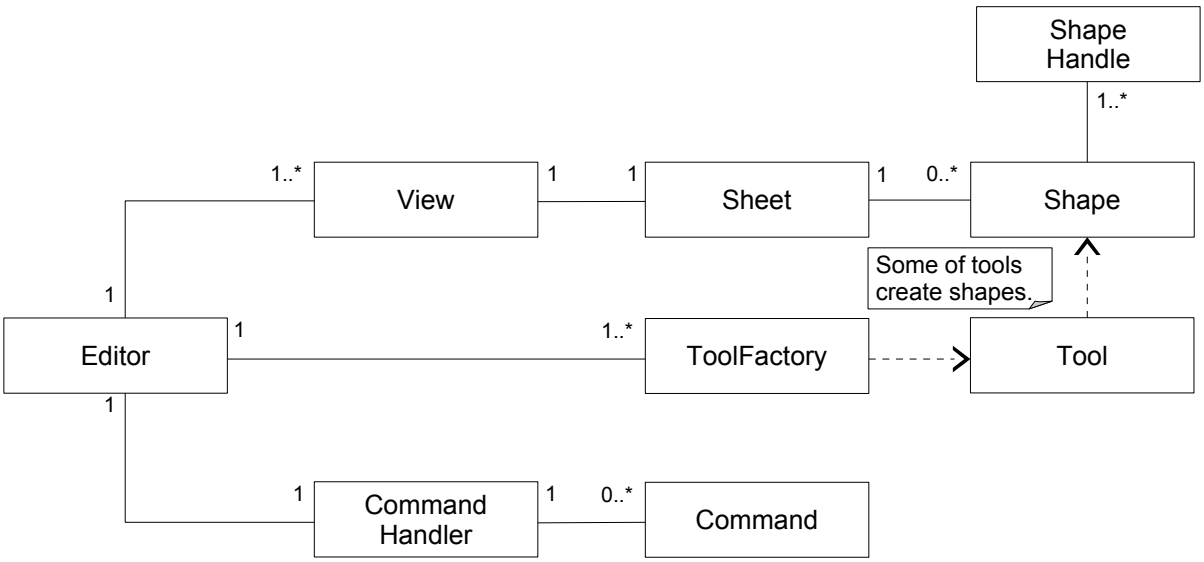

Figure 1: Concept interfaces/classes

3 JDT Core Interfaces and Classes

In the following sections the core interfaces and classes will be presented.

3.1 Editor, View, and Sheet

The editor maintains a list of views. Each view has an associated sheet. Responsibilities are:

- Editor:
  - maintains a list of views
  - maintains a list of tools
  - receives shapes to be delegated to the current view
  - receives selected shapes to be delegated to the current view
  - maintains a clip board
  - maintains a command handler
  - others
- View
  - receives shapes and delegates them to the associated sheet
  - receives and maintains selected shapes, returns selection handles of selected shapes
  - draws associated sheet
  - registers a listener at the sheet for sheet events
  - others
- Sheet
  - maintains the shapes including shape listeners, draws them if necessary
  - keeps a set of listeners for sheet events

3.2 Tool Factories, Tools, and Shapes

Tools are used to either manipulate shapes, or to create them. Let's consider tools for creating shapes. The editor is made aware about several tool `ToolFactory` instances. Upon calling a tool factory's `createTool()` method, the factory creates and returns the appropriate tool. The user's mouse interactions are delegated to the tool allowing it to create a particular shape, say, a box. Responsibilities are:

- Tool Factory:
  - creates the corresponding tool
  - returns the tool's icon and tool name (needed for the user interface)
- Tool
  - is activated prior to its use
  - receives the `mouseDown()`, `mouseDrag()`, `mouseUp()`, and `mouseOver()` events
  - upon creation, a shape is registered with the editor by the tool
  - is deactivated when no longer used (a tool can be reused, however)
- Abstract Shape class
  - provides the default properties (fill color, pen color, pen size)
  - provides basic infrastructure for its listeners
- Concrete Shapes
  - are created by a tool
  - different shapes are created by different tools

Figure 2 shows the relationship between these concepts, abstract class, and concrete classes.

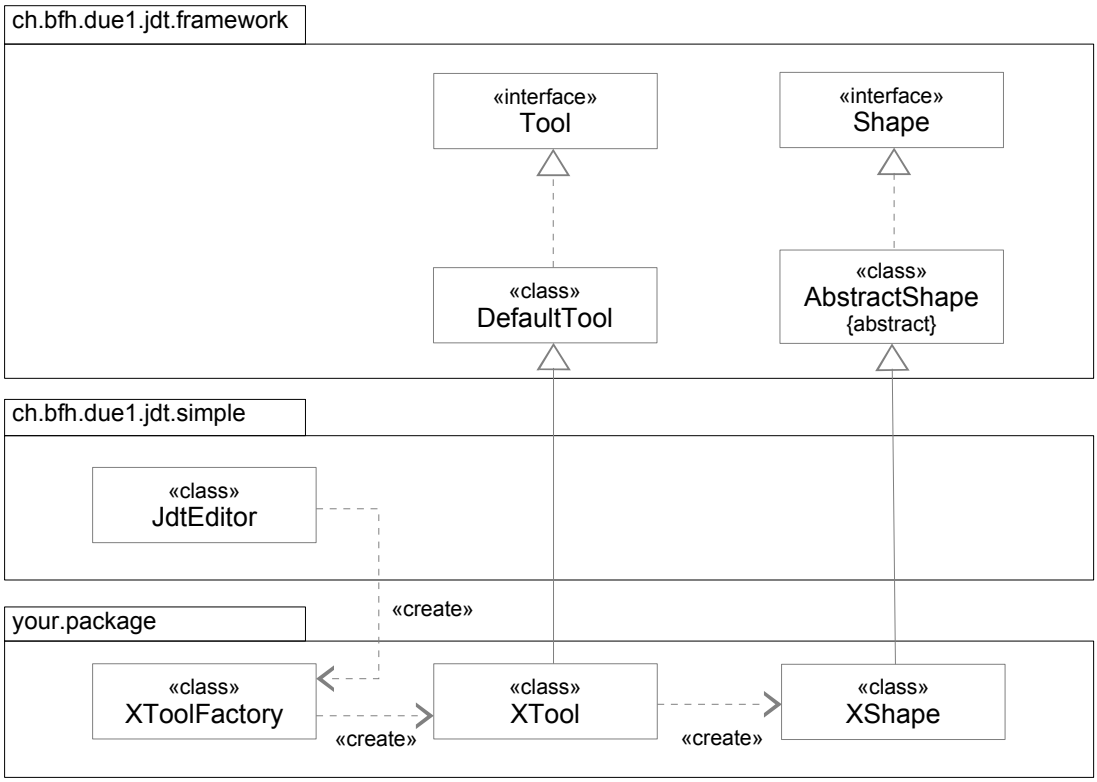

Figure 2: Relationship of tool factory, tools and shapes

3.3 Shapes and Shape Handles

Every shape hast at least one shape handle. A shape handle is a visible object related to a shape (or group of shapes). When the appropriate shape is selected then the group's shape handle are made visible on the screen. The user, then, can manipulate the shape by clicking and dragging on the shape handle.

Responsibilities are:

- Shape
  - when asked, returns the associated shape handles
- Abstract Shape Handle
  - defines the common form of all shape handles (form, fill and pen color, pen size)
  - has a reference to the shape the handle belongs to
  - provides the `draw()` method
  - provides empty method for mouse events; sub classes must override them appropriately
- Concrete Shape Handle
  - implements the concrete mouse event strategy action onto the associated shape

Figure 3 shows the relationship between shapes and shape handles.

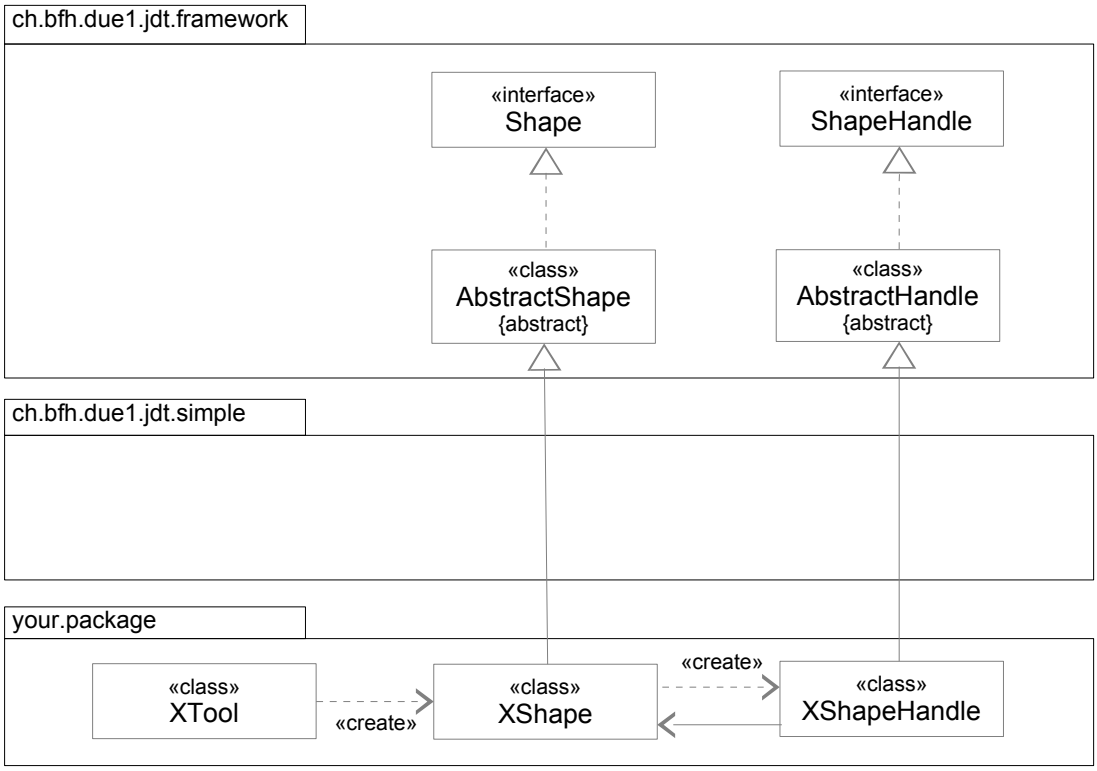

Figure 3: Shapes create shape handle instances; a shape handle operates on a shape

3.4 Commands, Undo- and Redo Operations

The editor supports the concept of undo-able operations. For this to work, the user's operations must be captured by `Command` objects. It is a `Command` object that actually executes the command. A `Command` object supports the following main operations:

- an operation `execute()` which performs the actual operation;
- an operation `undo()` which reverts the previously performed operation.

Command objects are stored in a history list provided by the command handler. Figure 4 shows the

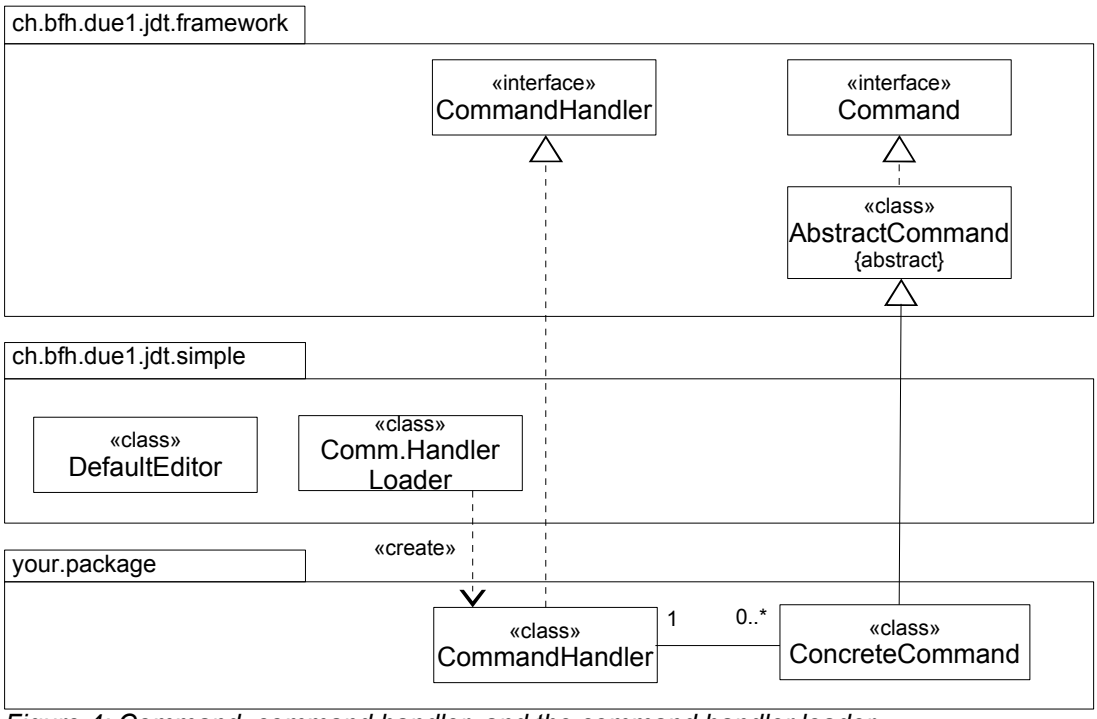

Figure 4: Command, command handler, and the command handler loader

relationship between the editor, the command handler, and the command objects.

3.5 Menu Items, Actions and Commands

Commands are invoked by *invokers*. Shape-creating *tools* can be invokers. Another class of invokers

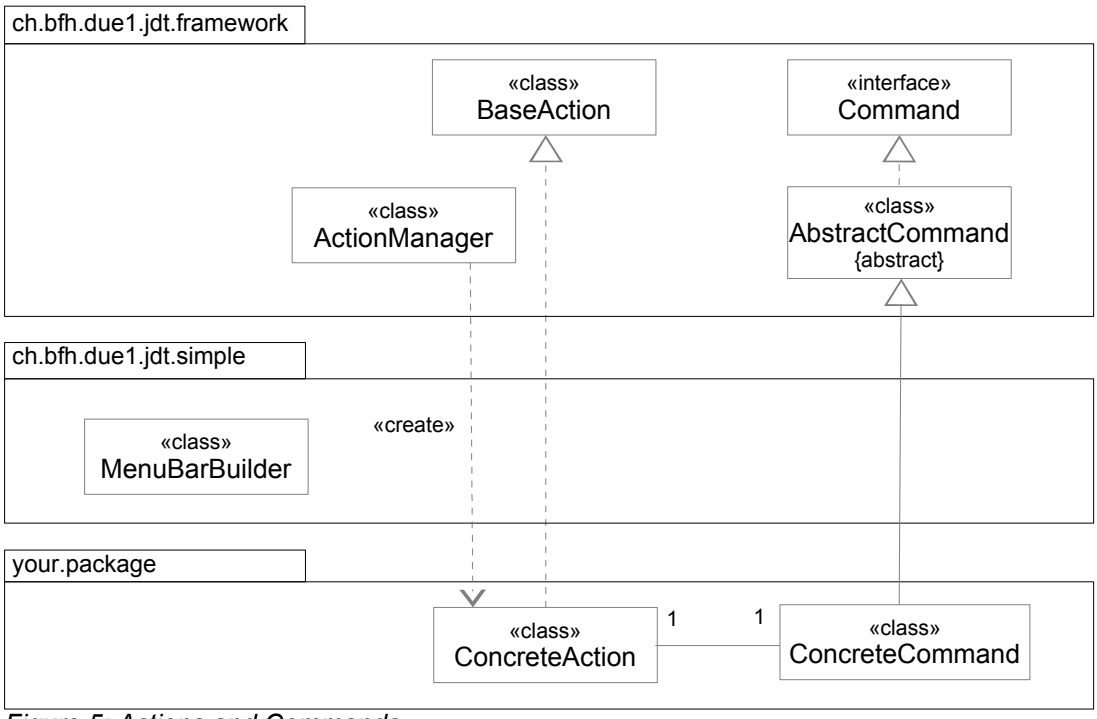

Figure 5: Actions and Commands

are *actions*. Actions implement Java's `javax.swing.Action` interface. However, your concrete action classes should extend the framework's `ch.bfh.due1.jdt.framework.BaseAction` class. See also Figure 5.

A sketch of a `ConcreteAction` looks like:

```
1.  public class ConcreteAction extends BaseAction {
2.      public CutAction() {
3.            setEnabled(false); // Set true if enabled from the beginning.
4.      }
5.
6.      @Override
7.      public void actionPerformed(ActionEvent ev) {
8.            // Perform preparatory work if necessary.
9.            // Create appropriate Command instance.
10.           // Execute command.
11.           // Get command handler from editor and register command.
12.           // Ask editor to check GUI state:
13.           getEditor().checkEditorState();
14.     }
15.
16.     @Override
17.     public void checkAction() {
18.           // Check condition if action should be enabled.
19.     }
20. }
```

Action classes are loaded and initialized by the framework's `ch.bfh.due1.jdt.framework.ActionManager` class. However, the action manager is directed by the simple application's `MenuBarBuilder` class. The menu bar builder reads the configuration file `jdt/menu/menu.properties`. An excerpt looks like:

```
1.  ##
2.  # File menu
3.  ...
4.  ...
5.
6.  ##
7.  # Edit menu.
8.
9.  jdt.menu.type.1=Regular
10. jdt.menu.name.1=Edit
11.
12. jdt.menu.item.type.1.0=Regular
13. jdt.menu.item.name.1.0=Undo
14. jdt.menu.item.description.1.0=Undo the last command
15. jdt.menu.item.accelerator.1.0=control Z
16. jdt.menu.item.iconpath.1.0=jdt/icon/Undo16.gif
17. jdt.menu.item.actionclass.1.0=<fully-qualified class name>
18.
19. ...
```

You need to declare your actions being called by menu items in the above configuration file.