Interactr

Iteration 3

Jelle de Coninck, Hannes De Smet, Bruno Vandekerkhove, Shani Vanlerberghe May 24, 2018

KULeuven

Table of contents

- 1. Design
- 2. Extensibility
- 3. Testing
- 4. Project Management

Design

Domain Model - Overview

- Low representational gap
- Main classes : Diagram, Message, Party
- Business logic completely separate from UI

Domain Model - UML

Put diagram of domain model here

Domain Model - Diagram Visitors

Interface DiagramVisitor can be implemented by various visitors in other subsystems.

Every DiagramComponent can accept such a visitor.

ightarrow makes it easy to add functionality to the domain layer without having to add code to the classes themselves (like UI-related code)

Example of visitors used in our UI layer : DialogCreator (makes dialog for a given diagram component without resorting to the use of instanceof)

Domain Model - Diagram Observers

Interface DiagramObserver represents an observer of a diagram.

A diagram has a list of such observers that it notifies when a party is added, a label is edited, ...

We currently implement this interface in our classes representing dialog windows and diagram views, for synchronisation of, say, labels.

Entry Point - Controller

Everything starts in Controller :

- → creates a Window (extends CanvasWindow)
- → associates the window with PaintBoard (draw events)
- ightarrow associates the window with EventHandler (key/mouse events)

Both support Protected Variations. Both encapsulate awt package.

Entry Point - Controller

Put diagram of Controller, PaintBoard and EventHandler and Window + CanvasWindow here

Controller - Underlying Structure

The Controller keeps track of a list of SubWindows.

A SubWindow can be a DiagramWindow or a DialogBox.

Each SubWindow has a frame.

Each SubWindow is 'activated' by putting it at the front of the window list.

SubWindows - Diagram Windows & Dialog Boxes/Windows

A DiagramWindow has two DiagramViews. A DialogWindow aggregates Models representing *controls*.

A Diagram is directly altered by DiagramViews and DialogWindows.

Synchronization is either done with the Observer pattern, or *lazily* (update upon drawing).

Since Diagram knows when it changes, it keeps a list of DiagramObservers and notifies them (Information Expert).

→ disadvantage : coordinates aren't synchronised (solved by 'making room' upon addition of a diagram component to a 'DiagramView')

Event Handling - How it all Starts

- ightarrow Every mouse or key event goes from Window ightarrow EventHandler.
- ightarrow EventHandler interprets the events and transforms them into Command instances.
- \rightarrow Command is forwarded to Controller.

Note: if event is a *mouse press*, EventHandler asks the Controller to activate the SubWindow at that coordinate first. This very window will be the first receiver for the command.

Event Handling - Commands

Put diagram of Command etc. here

Event Handling - Executing Commands

Precondition: SubWindow is active

- ightarrow Every Command can be handled by a CommandHandler.
- ightarrow DiagramWindow, DiagramView and DialogBox extend the CommandHandler class.
- → The first CommandHandler is the active SubWindow.
- → Every CommandHandler either deals with the Command or (on failure) passes it to the next CommandHandler in its chain.

Patterns used : Command, Visitor and (some sort of) Chain of Responsibility.

Event Handling - Example

Vid. Sequence diagram for Add Party.

Drawing - Paint Board

The PaintBoard encapsulates the *awt* package. It is passed along to any class responsible for drawing. Drawing is done 'on' the paint board.

- manages clip rect (to prevent overflowing)
- allows color changes

- ..

ightarrow it is a Facade

Drawing - Displaying Elements

Each class that displays something draws itself (Information Expert):

- SubWindow draws title bar, close button, frame.
- DiagramView draws diagram with messages, parties, ...
 - \rightarrow use of Visitor pattern to generate figures (representations) for these diagram components.
 - \rightarrow these figures are Flyweights.
 - \rightarrow separation of UI / domain logic without use of type checking (i.e. instanceof).
- Class hierarchy for Model (Composite pattern) used for representation/drawing of a variety of elements in the system (parties, messages, controls, ...).

Extensibility

Extensibility - Diagram Visitors

As stated before, DiagramVisitor interface makes it easy to add functionality to the domain model without 'polluting' it with, say, UI logic.

Extensibility - Commands

Command pattern eases undoing, allows delayed processing, use by menu items, ... When using new types of events (eg. speech recognition) only EventHandler has to change.

Disadvantages;

- many new types of CommandHandler?
 - → bloated Command abstract class (Visitor pattern discussion)
- if Chain of Responsibility is long
 - ightarrow wasted processing (command goes down the whole chain before realising that it can't be handled)
- $\bullet\,$ if many new system operations for many different types of windows
 - ightarrow redesign probably warranted.

Extensibility - Model

Representation of various elements in the system can be done with the Model class.

→ new elements in domain model and elsewhere is a matter of creating subclasses of Model, whenever necessary.

More complex Models can be made (and recycled) by composing them out of 'primitives' (Lines, Ovals, ...).

Testing

Testing - Coverage

90% classes, 87% lines covered in 'all classes in scope'		
Element	Class,%	Method, %
interactr.cs.kuleuven.be.domain	90% (9/10)	84% (78/92)
interactr.cs.kuleuven.be.exceptions	78% (11/14)	100% (3/3)
interactr.cs.kuleuven.be.ui.command	88% (16/18)	85% (34/40)
interactr.cs.kuleuven.be.ui.control	92% (13/14)	82% (164/200)
interactr.cs.kuleuven.be.ui.design	100% (10/10)	96% (55/57)
interactr.cs.kuleuven.be.ui.geometry	100% (2/2)	90% (19/21)
interactr.cs.kuleuven.be.ui.Controller	100% (1/1)	94% (18/19)
c interactr.cs.kuleuven.be.ui.EventHandler	100% (1/1)	100% (13/13)
c interactr.cs.kuleuven.be.ui.PaintBoard	100% (1/1)	90% (19/21)
c interactr.cs.kuleuven.be.ui.Window	100% (1/1)	100% (10/10)

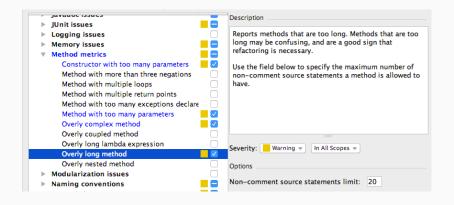
Project Management

Refactoring

- No duplicated code.
- Moved methods here and there.
- Addressed long methods with extract method

 → not the best metric ... but all methods (easily) 'fit on the screen'
 (max. ≈ 25 lines).
- If appropriate a temporary variable was replaced with getters.
- Avoided message chains.
- Virtually no switch statements (in EventHandler).
- Comments never really necessary (maybe when drawing messages in SequenceView).

Refactoring - Tools



Time Management

- weekly meeting with assistant
- +- 40 hours per person
- designed, refactored, tested, redesigned, ... in no particular order

Demonstration