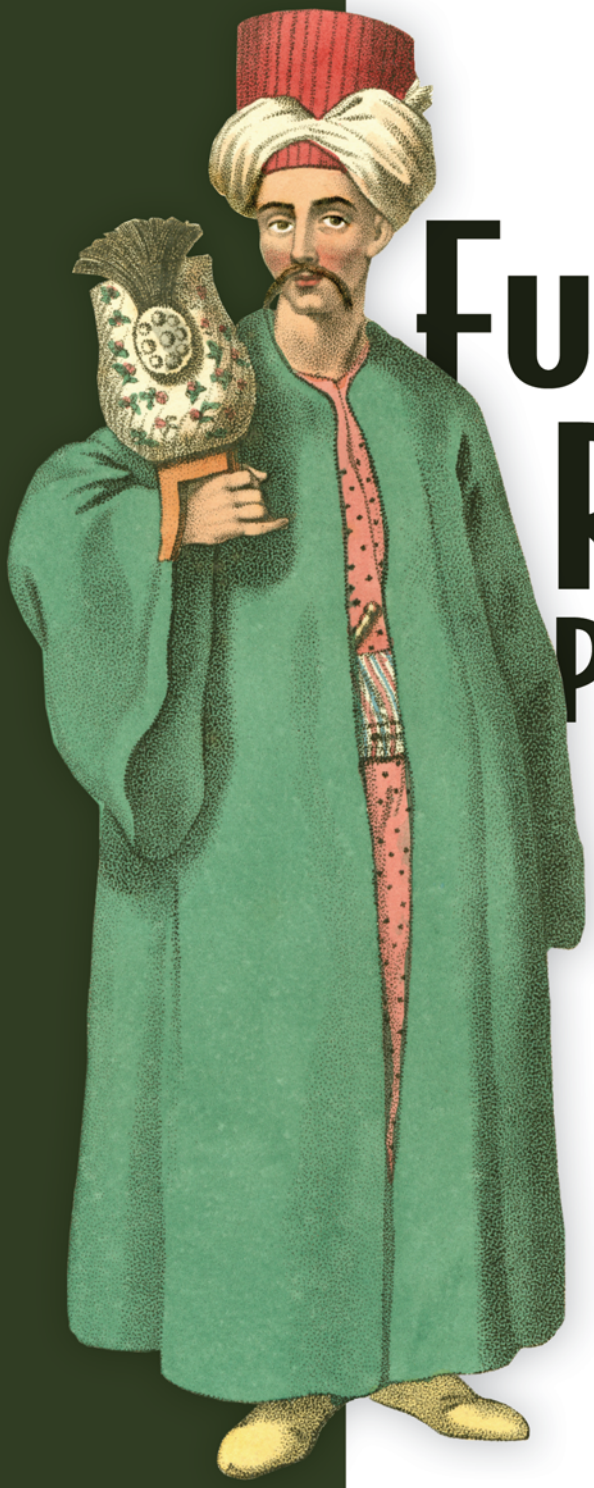
SAMPLE CHAPTER



Functional Reactive Programming

Stephen Blackheath
Anthony Jones

FOREWORD BY Heinrich Apfelmus

MANNING

Functional Reactive Programming

by Stephen Blackheath, Anthony Jones

Sample Chapter 13

brief contents

13 Refactoring

This chapter covers

- A drag-and-drop example
- Adding some features
- Contrasting refactoring between OOP and FRP code

In your job as a programmer, you'll often add a feature or fix a bug by adding extra code to a class or method. As the code gets longer, it can get messier. In this chapter, we'll illustrate that process by example.

As the Agile software development methodology emphasizes, when you start to smell that "code smell" of untidy code, it's usually a good idea to refactor the code by breaking the class or method into smaller pieces. This is important because messy code is complex code, and we've argued that complexity can compound. We've also argued that this should be less of an issue in FRP due to its compositionality, but refactoring is still important. Fortunately, refactoring with FRP is as easy as falling off a log.

13.1 To refactor or not to refactor?

If you add extra state and logic to an existing class, this is the question you must ask yourself. Latent problems in code usually manifest when you make modifications.

Often, deep in your heart, you'll hear a little voice calling, "Refactor me!" But do you always listen?

The complexity you added gives you an uneasy feeling. That's because you know at some point you'll need to split things up. "It's only a few small changes," you reply to yourself. But the longer you put off refactoring, the more work it will eventually be, as you can see in figure 13.1.

Sometimes it's difficult to see a neat way to do it. Sometimes you don't want to incur hours of testing and be blamed for refactoring breakage in someone else's code; adding an extra variable and a couple of lines of logic seems infinitely preferable. Yes, sometimes short-term considerations win out. This is exactly the process by which Frankenstein created his famous monster.

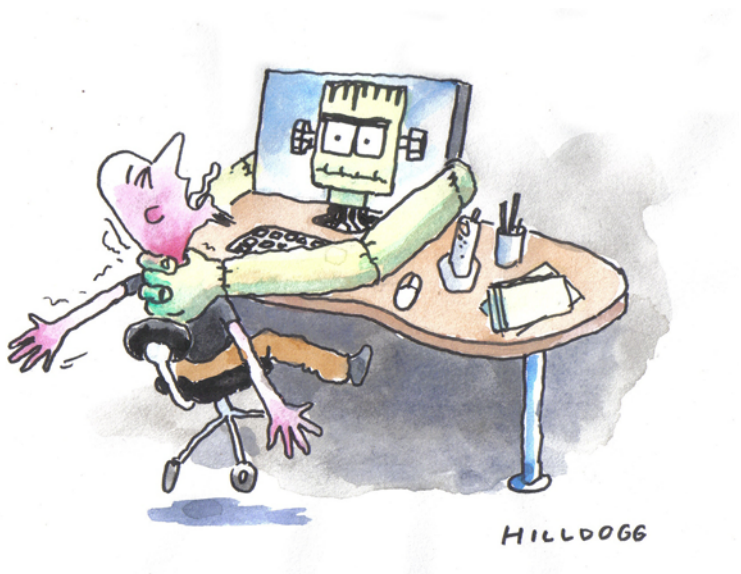


Figure 13.1 Dale regrets having put off refactoring.

13.2 A drag-and-drop example

To show FRP refactoring in action, we'll use a variation on the drag-and-drop examples developed in chapters 7 and 10. Recall from chapter 7 that there are three types of mouse event, each associated with an (x, y) position in the window:

- *Mouse down*—The mouse button is pressed down.
- *Mouse move*—The mouse position changes, but there is no change to the buttons.
- *Mouse up*—The mouse button is released.

This implementation doesn't use `switch` because we'll be drawing diagrams and we haven't figured out a way to diagram the dynamic changes of a `switch`.

13.2.1 Coding it the traditional way

Let's first look at how you write this in a traditional object-oriented / listener / state machine style. You typically write a class called `DragAndDrop` that does the following:

- Registers listeners on the input events
- Has fields for the state

To keep things tidy, you'll use two container classes. `Dragging` holds the state you need to keep while dragging. Instead of updating the diagram for each mouse move, you'll draw the element separately as it's dragged and update the document only at the end. You add a helper method to give a representation of this in a second class, `FloatingElement`. This information is used in the `paint` method to draw the floating element:

Omitting boilerplate

Original mouse position of the drag

Container class for the drag state used during drag

Element you're dragging

Helper method that returns a representation of the floating element

New position = original position + distance traveled

Selected element and the position to draw it at while floating

```
class Dragging {
    Dragging(Element elt, Point origMousePos) { ... }
    Element elt;
    Point origMousePos;

    FloatingElement floatingElement(Point curMousePos) {
        Vector moveBy = curMousePos.subtract(origMousePos);

        return new FloatingElement(elt.getPosition().add(moveBy), elt);
    }
}

class FloatingElement {
    FloatingElement(Point position, Element elt) { ... }
    Point position;
    Element elt;
}
```

Before we get to the rest of the code, we'll sketch out the logic in a simplified version of the diagram style used in chapter 2. It uses

- Round corners for things that output *streams*
- Square corners for state (cells)

We'll keep it simple and leave out the `mouseMove` event handling, so for now the floating element won't be drawn as you drag.

In figure 13.2, the top rounded-corner box (logic) is activated when a `mouseDown` event comes in. It snapshots from the `document` (note the arrow from `document`) and asks if an element exists at the mouse position. If it does, it updates `dragging` with a value of `new Dragging(elt, pos)`. You're now dragging `elt`.

The rounded-corner box at left is activated by the `mouseUp` event. If you're dragging (that is, the `dragging` variable has a non-`null` value), you'll end the drag. You produce an event labeled `drop`, and if you follow the arrows, you can see that it causes three things to happen:

1. `null` is written into the `dragging` variable, which puts you back in the idle state (not dragging).

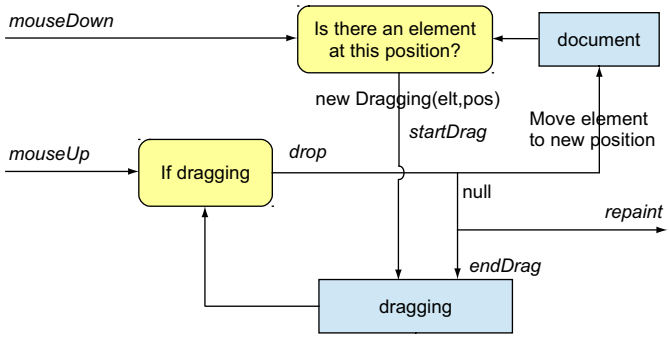


Figure 13.2 Minimal drag-and-drop logic

2 You update the document with the new position for the element.
3 You repaint the window.

The Java pseudocode is shown in the following listing. Shortly we'll contrast it against the equivalent FRP.

Listing 13.1 Pseudo Java for drag-and-drop logic, traditional object-oriented style

```
class Dragging {
    Dragging(Element elt, Point origMousePos) { ... }
    Element elt;
    Point origMousePos;

    FloatingElement floatingElement(Point curMousePos) {
        Vector moveBy = curMousePos.subtract(origMousePos);

        return new FloatingElement(elt.getPosition().add(moveBy), elt);
    }
}

class FloatingElement {
    FloatingElement(Point position, Element elt) { ... }
    Point position;
    Element elt;
}

class DragAndDrop implements MouseListener
{
    Document doc;
    Window window;
    Dragging dragging = null;

    DragAndDrop(Document doc, Window window) {
        ...
        window.addMouseListener(this);
    }
```

Container class for the drag state used during drag

Omitting boilerplate

Element you're dragging

Original mouse position of the drag

Helper method that returns a representation of the floating element

New position = original position + distance traveled

Selected element and the position to draw it at while floating

Asks the window to call you back with mouse events

```
void mouseDown(Point mousePos) {
    Element elt = doc.lookupByPosition(mousePos);
    if (elt != null)
        dragging = new Dragging(elt, mousePos);
}
void mouseMove(Point mousePos) {
}
void mouseUp(Point mousePos) {
    if (dragging != null) {
        FloatingElement flt = dragging.floatingElement(mousePos);
        doc.moveTo(flt.elt, flt.position);
        dragging = null;
        window.repaint();
    }
}
}
```

Starts dragging if you press down on a document element

If you're dragging…

…moves the document element to its floating position

Repaint: assumes the paint() method reads from the document directly

13.2.2 The FRP way: diagrams to code

As we said at the beginning of the book, FRP code directly reflects a box-and-arrows diagram. Let's translate our diagram into code.

In figure 13.3, we put the diagram side-by-side with the equivalent FRP pseudocode. The structure of FRP code is fundamentally the same as the structure of the diagram.

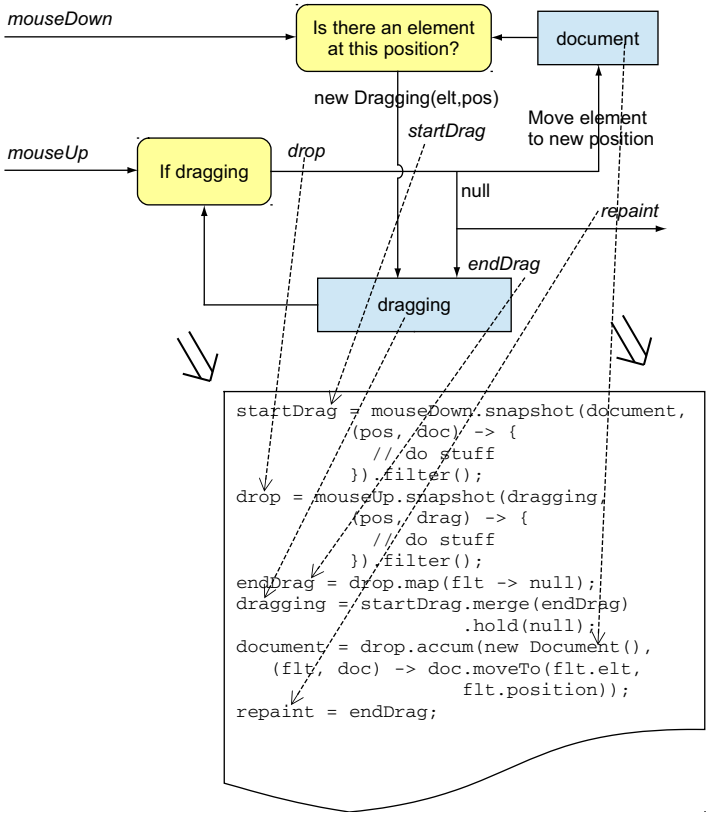


Figure 13.3 The structure of FRP code corresponds closely to a "boxes and arrows" diagram.

Observe the following:

- For brevity, we've left out the types in the variable assignments.
- Each variable (rectangular box) and each italicized label we've added to an arrow corresponds to a statement in the FRP code. These statements are written as assignments to named variables.
- Whenever a statement references a variable declared elsewhere, there's a corresponding arrow in the diagram.

13.3 Adding a feature: drawing the floating element

As it is, the user doesn't get any visual feedback when they drag an element. You'd like to draw the element floating as the user drags it. The traditional way would be to make the `mouseMove()` method cause a repaint if you're dragging. The `Window` instance's `paint()` method, which does the real work, will use `document` directly to draw the document, and it will call the `Dragging` class's `floatingElement()` method to find out where and how to draw the floating element (we won't show the code for `paint()`):

```
FloatingElement floatElt = null;

void mouseMove(Point mousePos) {
    if (dragging != null)
        floatElt = dragging.floatingElement(mousePos);
    else
        floatElt = null;
    window.repaint();
}
FloatingElement floatingElement() {
    return floatElt;
}
```

Records floating element information

Requests the window to be repainted

Window's paint() calls this to ask about the floating element and its position.

Now let's add this `mouseMove` handling to the diagram. The additions are shown in bold; see figure 13.4.

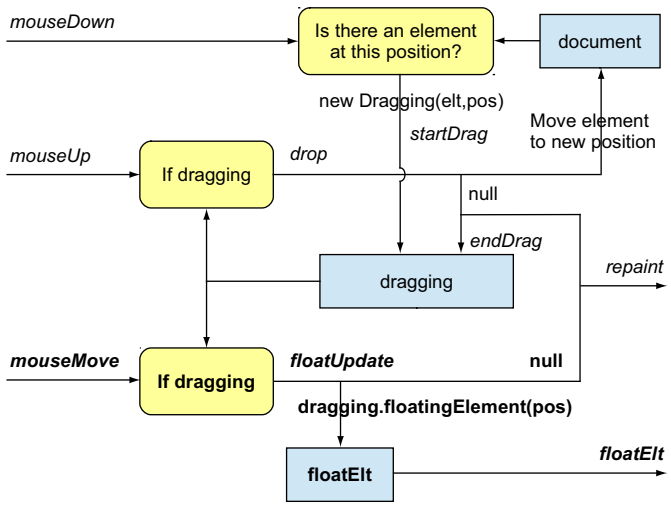


Figure 13.4 Add repaint on `mouseMove` so you see the element floating as you drag it. Additions are in bold.

13.4 Fixing a bug: clicks are being treated as drags

This code has an undocumented feature that annoys the user: clicks are being misinterpreted as drags, so when the user clicks an object, it often gets moved slightly. Let's fix that by having two phases:

- *Pending*—The mouse button has been pressed down, but you haven't started moving yet.
- *Dragging*—You've detected mouse motion while in the pending phase, so the element is now really being dragged.

You'll detect mouse motion if the mouse has moved five pixels or more from the point where the mouse button was pressed. See figure 13.5: it's not until you get outside a five-pixel radius of the drag origin that the drag starts, at event 3.

Figure 13.5 Start the drag only when you move outside a five-pixel radius of the drag origin.

The next listing shows the changed lines in bold to add the extra pending phase. Note that you have to be careful to use the right state variable in the right place.

Listing 13.2 Code changes to add a pending phase before drag, traditional-style

```
class DragAndDrop implements MouseListener
{
    Document doc;
    Window window;
    Dragging pending = null;
    Dragging dragging = null;
    FloatingElement floatElt = null;

    DragAndDrop(Document doc, Window window) {
        ...
        window.addMouseListener(this);
    }
    void mouseDown(Point mousePos) {
        Element elt = doc.lookupByPosition(mousePos);
        if (elt != null)
            pending = new Dragging(elt, mousePos);
    }
    void mouseMove(Point mousePos) {
        if (pending != null &&
                mousePos.distance(pending.origMousePos) >= 5)
            dragging = pending;
        if (dragging != null)
            floatElt = dragging.floatingElement(mousePos);
        else
            floatElt = null;
        window.repaint();
    }
    FloatingElement floatingElement() {
        return floatElt;
    }
```

```
    void mouseUp(Point mousePos) {
        if (dragging != null) {
            FloatingElement flt = dragging.floatingElement(mousePos);
            doc.moveTo(flt.elt, flt.position);
            dragging = null;
            window.repaint();
        }
        pending = null;
    }
}
```

Figure 13.6 adds this logic to the previous FRP diagram (which was figure 1.7). Additions are again in bold.

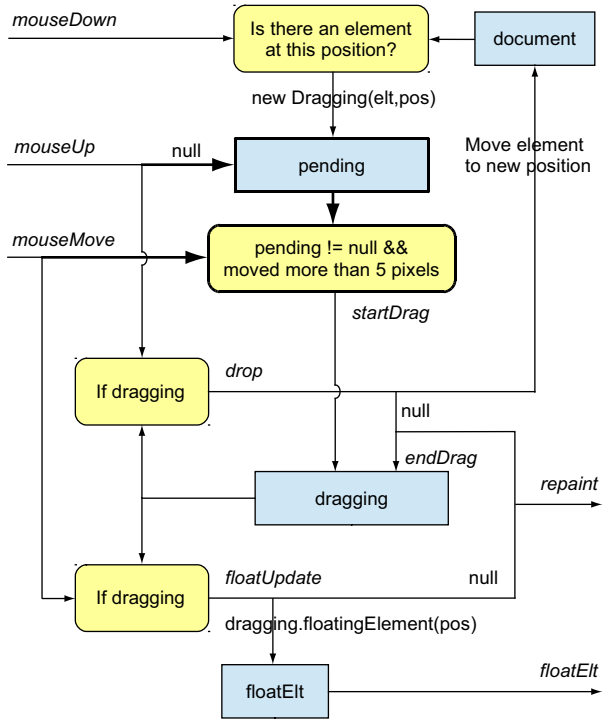


Figure 13.6 Add a pending phase before the drag starts. Additions are in bold.

13.5 FRP: refactoring is a breeze

The code in the previous example has a problem: it's getting messy. In the traditional-style code presented, it would be easy to make a mistake and mix up `pending` and `dragging`.

You can improve this by refactoring each variable into a separate class to limit the scope so one part of the logic sees only `pending` and the other sees only `dragging`, and

the interface between the classes is clearly delineated. You'll separate this logic into three classes:

- `DragPending`—Manages `pending` state
- `Dragging`—Manages `dragging` state
- `DrawFloating`—Manages `floatElt` state

Figure 13.7 shows a typical refactoring in the traditional programming style. You move the bits of code relating to each state into classes of their own. After that, you'd neaten up the interfaces between them; for example, `DragPending` would call a new `startDrag()` method on `Dragging` to set its `dragging` state.

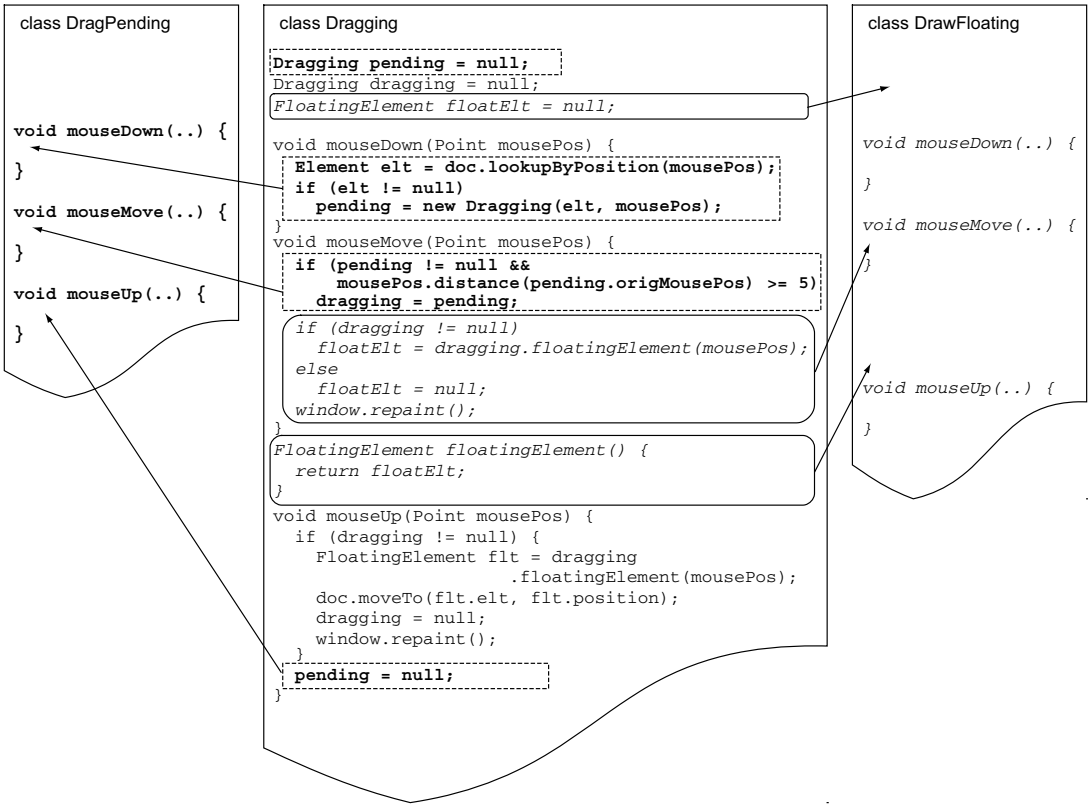


Figure 13.7 Traditional refactoring: separate the code into three classes.

Figure 13.8 shows how you refactor in FRP. In practice, you'd do this directly with code, but we're using a diagram to get the concept across.

We've "drawn" circles around groups of boxes that are conceptually related, so that the number of incoming and outgoing arrows is small, and we've given the circles

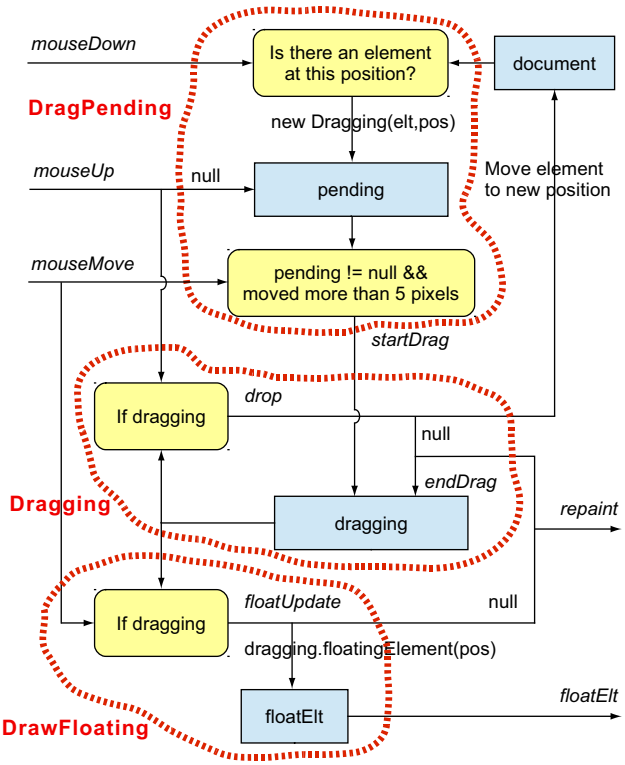


Figure 13.8 FRP refactoring: just "draw" circles around the modules you want, and label them.

names. Each box corresponds to a statement in the FRP code. You just move those statements into new classes and fix up the variable references.

You'd create a structure like the following. We arrived at this by going round the edge of the lines in the diagram. Each incoming arrow becomes a constructor argument, and each outgoing arrow becomes a field. In practice, you'd look at each statement and its dependencies:

```
class DragPending {
    DragPending(Stream<MouseEvent> sMouseDown, Stream<MouseEvent> sMouseUp,
                Stream<MouseEvent> sMouseMove, Cell<Document> document) {}
    Stream<Unit> sStartDrag;
}
class Dragging {
    Dragging(Stream<Unit> sStartDrag, Stream<MouseEvent> sMouseUp) {}
    Cell<Dragging> dragging;
    Stream<DocumentUpdate> sDrop;
    Stream<Unit> sRepaint;
}
class DrawFloating {
    DrawFloating(Stream<MouseEvent> sMouseMove, Cell<Dragging> dragging) {}
    Stream<Unit> sRepaint;
    Cell<Optional<FloatingElement>> floatElt;
}
```

Then you paste the existing FRP statements into the new constructors. Done.

You don't have to tease the code apart as you did with the traditional code. Furthermore, the compiler will do a good job of making sure you don't make mistakes: if something is out of scope or you paste a statement into the wrong constructor, the code won't compile. If you get the order of arguments wrong, the compiler is likely (but not certain) to complain about a type mismatch.

We'll repeat what we said in section 11.4: unit tests are normally used to protect against code breakage. FRP has so much built-in checking that it generally isn't necessary to use tests to protect the code against refactoring breakage. But of course, tests can never hurt. As a result of all this, FRP programmers don't experience the same "Should I? Shouldn't I?" refactoring dilemma.

13.6 Summary

- Sometimes the difficulty and risk of refactoring lead us to favor short-term considerations. Refactoring gets put off, and code gets messy.
- FRP code doesn't get messy as easily due to its compositionality.
- FRP code is automatically safe and easy to refactor.
- FRP protects against refactoring breakage so well that unit tests aren't necessary for this purpose.
- The dilemma of short-term versus long-term considerations largely disappears with FRP.

PROGRAMMING

Functional Reactive Programming

Blackheath • Jones

Today's software is shifting to more asynchronous, event-based solutions. For decades, the Observer pattern has been the go-to event infrastructure, but it is known to be bug-prone. Functional reactive programming (FRP) replaces Observer, radically improving the quality of event-based code.

Functional Reactive Programming teaches you how FRP works and how to use it. You'll begin by gaining an understanding of what FRP is and why it's so powerful. Then, you'll work through greenfield and legacy code as you learn to apply FRP to practical use cases. You'll find examples in this book from many application domains using both Java and JavaScript. When you're finished, you'll be able to use the FRP approach in the systems you build and spend less time fixing problems.

What's Inside

- Think differently about data and events
- FRP techniques for Java and JavaScript
- Eliminate Observer one listener at a time
- Explore Sodium, RxJS, and Kefir.js FRP systems

Readers need intermediate Java or JavaScript skills. No experience with functional programming or FRP required.

Stephen Blackheath and **Anthony Jones** are experienced software developers and the creators of the Sodium FRP library for multiple languages.

Illustrated by Duncan Hill

"A gentle introduction to the necessary concepts of FRP."
—From the Foreword by Heinrich Apfelmus, author of the Reactive-banana FRP library

"Highly topical and brilliantly written, with great examples."
—Ron Cranston, Sky UK

"A comprehensive reference and tutorial, covering both theory and practice."
—Jean-François Morin
Laval University

"Your guide to using the merger of functional and reactive programming paradigms to create modern software applications."
—William E. Wheeler
West Corporation

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
manning.com/books/functional-reactive-programming

ISBN-13: 978-1-63343-010-5
ISBN-10: 1-63343-010-3
54999
9 781633 430105

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]