From Callbacks to Design Patterns

Tom Verhoeff
Software Engineering & Technology
Department of Mathematics & Computer Science
Eindhoven University of Technology (TU/e)
The Netherlands

Abstract

This note describes how the simple idea of function-as-parameter, also known as callback, leads to a number of well-known design patterns. In particular, the following design patterns appear: Strategy, Adapter, Composite, Decorator, and Observer. Conciseness of presentation was a major goal.

Each pattern emerges from explicit design considerations and is illustrated with UML class diagrams and (separately available) Java code. The relation to listeners in a Java GUI is explained. Also, push-pull and update-query variations are considered. Testing is addressed in a separate note.

The Observer design pattern emerges here in a way that I have not seen before. An observer is just a strategy, in the sense of the Strategy pattern. A Composite pattern is used to take observer management and notification distribution out of the observable subject. This has as side-benefit that observers can be organized hierarchically in separately manageable observer groups, which can be reused on other observable subjects as well. The resulting concrete subjects are simpler as well, because they need not duplicate code for observer management and notification distribution.

The Adapter pattern can be used to construct observers from other (non-observer) functionality, and the Decorator pattern is used to modify existing observers.

1 Introduction

Nowadays, there are numerous references for design patterns, for example, the original book that popularized design patterns [3], the funny explanation-through-example and consideration-filled book [2], the recent compact book [1], and, not to forget, Wikipedia [7].

This note shows how one can discover some design patterns starting from the notion of a callback [6]. The resulting Observer pattern deviates from standard presentations, offering more flexibility and less code duplication.

2 Callbacks

Suppose you have some (complicated, tested, and approved) functionality A, say a simulation or generation algorithm, and you want to incorporate some further functionality B, say to process data as it gets produced by A. Note that data items are produced one by one inside A as a stream, and not bundled in a single result when A returns. Processing could simply consist of printing or counting, or be more advanced, like collecting statistics, showing animated graphics, etc.

One way to accomplish this is to merge the code for the processing (B) into the code for the simulator or generator (A). This approach has major disadvantages:

1. While merging code, you may break functionality in A, or B.
2. It makes testing of A and B harder.
3. It makes reuse of A or B harder.
4. It will not be easy to choose or even vary B at run time.

In software engineering jargon, we say that this approach introduces a hard (strong) *coupling* between A and B.

Applying the technique of *Divide and Conquer*, the functionality in A and in B should be encapsulated as functions, or (if also some global variables are needed) as classes containing appropriate processing methods. We will name these functions doA and doB. That way, all we need to merge into doA is a call of doB (see Figure 1, left). Using a switch statement in B, we could even vary functionality in B at run time. This looks more decent, and offers some improvement, but basically has the same disadvantages, though testing and reusing B may be easier.

Ideally, we would like to generalize A by abstracting from the processing in B. That would loosen the coupling. This can be accomplished by parameterizing A, and passing B as parameter in the invocation of A. In languages like C/C++ and Object Pascal (Delphi), you can pass a **function as parameter**[1] to another function. A function passed as parameter is also known as a *callback*. The caller of A provides A with a function through which A can invoke externally defined functionality (B). In a sense, A calls back to the environment from which it was called.

3 Callbacks in Java

The Java[2] programming language does not support function parameters. But we can accomplish the same thing by putting the function as a method inside a class, and passing an object of that class as parameter. For even more flexibility, we typically define an *interface* that contains the signature of the callback method.

[1] Often, what is actually passed is a pointer to a function.

[2] Java 8 has *functional interfaces*, *lambda expressions*, and *method references* to remedy this; see Appendix B.

In our example, the interface for functionality B could be defined as follows[3].

```
1 public interface FunctionalityB {
2     void doB(String data);
3 }
```

Functionality A is now set up in the following way.

```
1 public class FunctionalityA {
2     public static void doA(int n, FunctionalityB b) {
3         for (int i = 0; i < n; ++ i) {
4             final String data; // data item produced
5             data = i + " produced by A"; // "complex" computation
6             b.doB(data);
7         }
8     }
9 }
```

Functionality B can be defined in different ways, for instance, to print the data:

```
1 public class DataPrinter implements FunctionalityB {
2     private final String name;
3     public DataPrinter(String name) {
4         this.name = name;
5     }
6     public void doB(String data) {
7         System.out.println(name + " processed " + data);
8     }
9 }
```

The main environment couples functionality A with functionality B at runtime:

```
1 public class Main {
2     public static void main(String[] args) {
3         final int n = Integer.parseInt(args[1]);
4         final FunctionalityB printer = new DataPrinter(args[2]);
5         FunctionalityA.doA(n, printer);
6     }
7 }
```

In DrJava[4], you can invoke the example by typing in the Interactions panel

```
> run Main a 3 B1
B1 processed 0 produced by A
B1 processed 1 produced by A
B1 processed 2 produced by A
```

[3] We violated some good coding standards to show all the code on a single page.

[4] See `www.drjava.org`. Try out the code while studying the examples; see Exercise 1.

Invoke it with different parameters for functionality A and B by typing

```
> run Main a 5 B2
B2 processed 0 produced by A
B2 processed 1 produced by A
B2 processed 2 produced by A
B2 processed 3 produced by A
B2 processed 4 produced by A
```

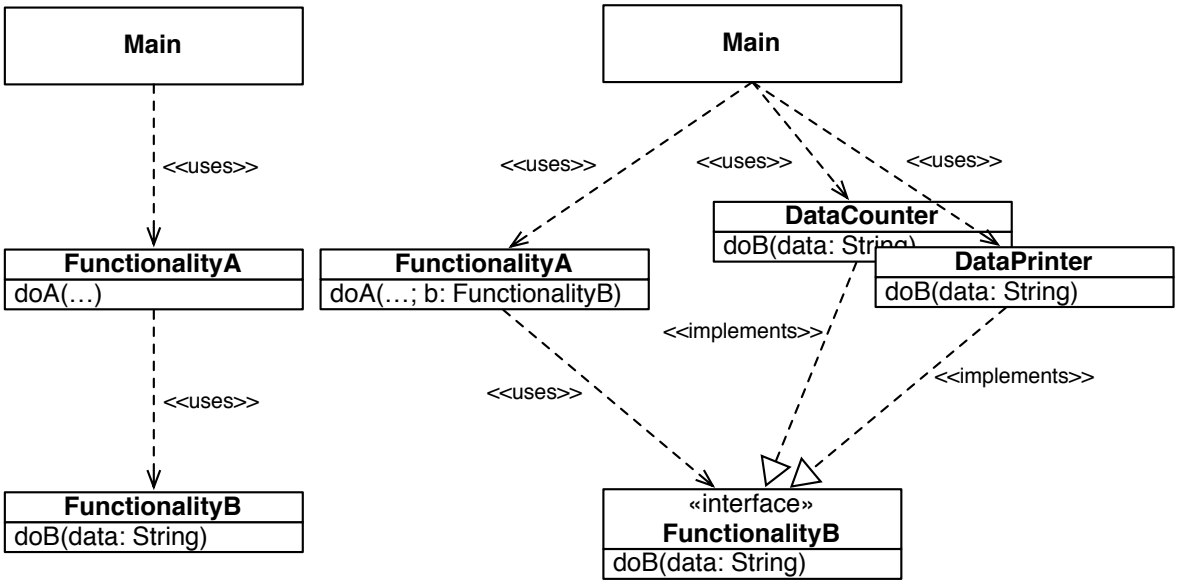

Figure 1: UML class diagram of hard coupling (left) and the basic callback (right)

Figure 1 (right) shows a UML class diagram of this approach with callbacks. The dashed arrows represent static dependencies. That is, $X \dashrightarrow Y$ means that X depends on Y, and X cannot be compiled/deployed without Y.

Note that functionality A and concrete functionality B are independent: neither depends on the other. Both depend on the, quite general, interface `FunctionalityB`. This is an example of the **Dependency Inversion Principle** (DIP): depend on abstractions (interfaces), not on concrete implementations.

At run time, functionality A gets coupled to specific functionality B by `Main`. However, A only requires that this B implements `FunctionalityB`. A does not need to know more details. Hence, it can invoke only operation `doB` on `b`, and nothing else, even if `b` has other methods. This run-time configuring of functionality A is known as **Dependency Injection** (DI).

It is also an example of the *Strategy* Design Pattern. Each way of processing the data items (functionality B) is referred to as a strategy. This design pattern makes it easy to vary the strategy used by functionality A at run time.

4 Defining an Anonymous Callback on the Fly

You can use an *anonymous* callback, that is, a nameless class that implements `FunctionalityB`. Here is an example that counts out the data items on standard output.

```
FunctionalityA.doA(n, new FunctionalityB() {
    int count;
    public void doB(String data) {
        ++ count;
        System.out.println("Item " + count + " counted");
    }
});
```

There is no way to retrieve the count (that is why we print each count). However, it is advisable at least to assign the anonymous callback to a local variable:

```
FunctionalityB adhocCounter = new FunctionalityB() {
    ...
};
FunctionalityA.doA(n, adhocCounter);
```

Even though the callback itself now has a name, the class of which it is an instance still does not have a name. Consequently, it is still impossible to retrieve the count afterwards. The type of `adhocCounter` is `FunctionalityB`, and that only provides access to `doB` and not to `count`.

It might be clearer to give that ad hoc class a name, in a local class definition, nested inside the main class. You can then also retrieve the count afterwards:

```
class AdhocCounter implements FunctionalityB {
    ...
};
AdhocCounter adhocCounter = new AdhocCounter();
FunctionalityA.doA(n, adhocCounter);
System.out.println(adhocCounter.count + " items seen");
```

Note that we now typed `adhocCounter` as an `AdhocCounter`, to allow access to the (non-private) instance variable `count`.

5 Wrapping a Class to Provide a Callback

Suppose we already have a nice Abstract Data Type for counting:

```
1 public class Counter {
2     private long count;
3     public long getCount() {
4         return count;
5     }
6     public void count() {
7         ++ count;
8     }
9 }
```

Unfortunately, this class does not implement interface `FunctionalityB`, so it cannot be passed to functionality A. A bad solution is to 'hack' the `Counter` and turn it (also) into `FunctionalityB`. A better solution is to *wrap* it, so that it becomes a `FunctionalityB` (an example of the *Adapter* Design Pattern):

```
1 class DataCounter extends Counter implements FunctionalityB {
2     public void doB(String data) {
3         count();
4     }
5 }
```

The main environment can use this wrapped version as follows.

```
1         final DataCounter counter = new DataCounter();
2         FunctionalityA.doA(n, counter);
3         System.out.println("Counter received " + counter.getCount()
4                 + " items from A");
```

Figure 2 shows a UML class diagram for the wrapping technique. A `DataCounter` is (behaves as) both a `Counter` and a `FunctionalityB`.

For more flexibility, keep the counter external to the adapter, and use composition instead of inheritance (see Figure 3). The main application then uses the adapter as callback, and uses the separate counter to retrieve its state. The adapter's constructor stores a reference to the counter to be used (also see Exercise 1d):

```
1 class CounterAdapter implements FunctionalityB {
2     private final Counter counter;
3     public CounterAdapter(Counter counter) {
4         this.counter = counter;
5     }
6     public void doB(String data) {
7         counter.count();
8     }
9 }
```

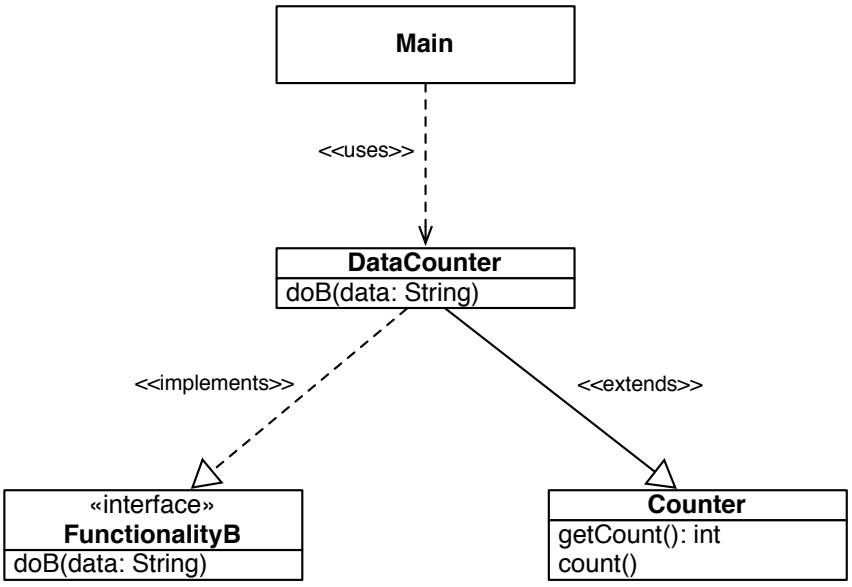

Figure 2: UML class diagram for wrapping to define a callback, with inheritance

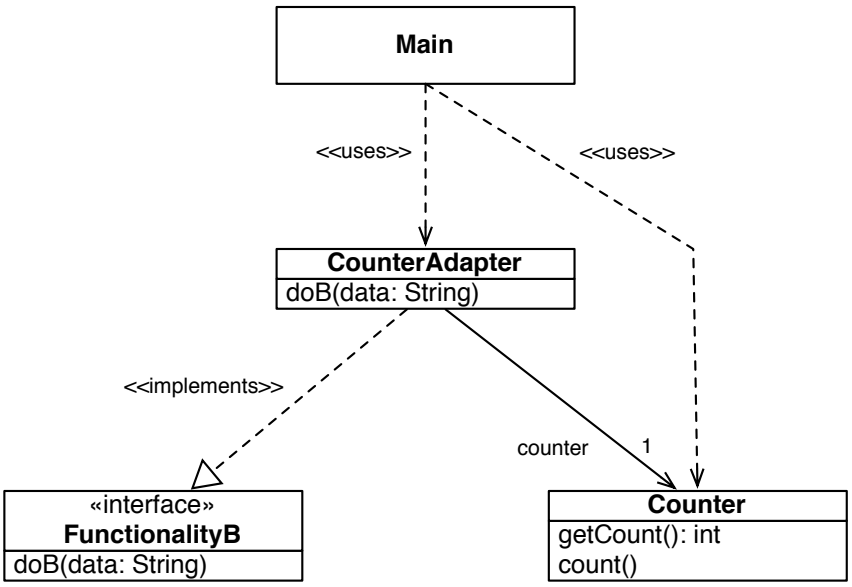

Figure 3: UML class diagram for adapter to define a callback, with composition

6 Distributing a Callback to Other Callbacks

What if you want to print the data items and also count them? It would be nice if you could combine the existing data processors, without modifying them (you do not want to break them, now that they have been tested and are working well).

You could combine them in an ad hoc (hard-coded) way, by defining a class that implements `FunctionalityB` and passes the data on to a data printer and a data counter.

Instead of this rather inflexible solution, it is nicer to define a generic data distributor that distributes one callback to multiple other callbacks:

```
1  public class CompositeFunctionalityB implements FunctionalityB {
2      private final List<FunctionalityB> parts;
3      public CompositeFunctionalityB() {
4          parts = new ArrayList<FunctionalityB>();
5      }
6      public void add(FunctionalityB b) {
7          parts.add(b);
8      }
9      public void doB(String data) {
10         for (FunctionalityB b : parts) {
11             b.doB(data);
12         }
13     }
14 }
```

The main environment can configure a distributor (the 'plumbing') at run time as follows, to print and count in a single run of functionality A.

```
1          final FunctionalityB printer = new DataPrinter(args[2]);
2          final DataCounter counter = new DataCounter();
3          final CompositeFunctionalityB printer_counter
4                  = new CompositeFunctionalityB();
5          printer_counter.add(printer);
6          printer_counter.add(counter);
7          FunctionalityA.doA(n, printer_counter);
8          System.out.println("Counter received " + counter.getCount()
9                  + " items from A");
```

By extending `CompositeFunctionalityB` with a method to unregister callbacks, you obtain a very flexible infrastructure:

```
1      public void remove(FunctionalityB b) {
2          parts.remove(b);
3      }
```

You can register a `CompositeFunctionalityB` with another `CompositeFunctionalityB` to create a hierarchic distribution network (also see the *Composite* Design Pattern).

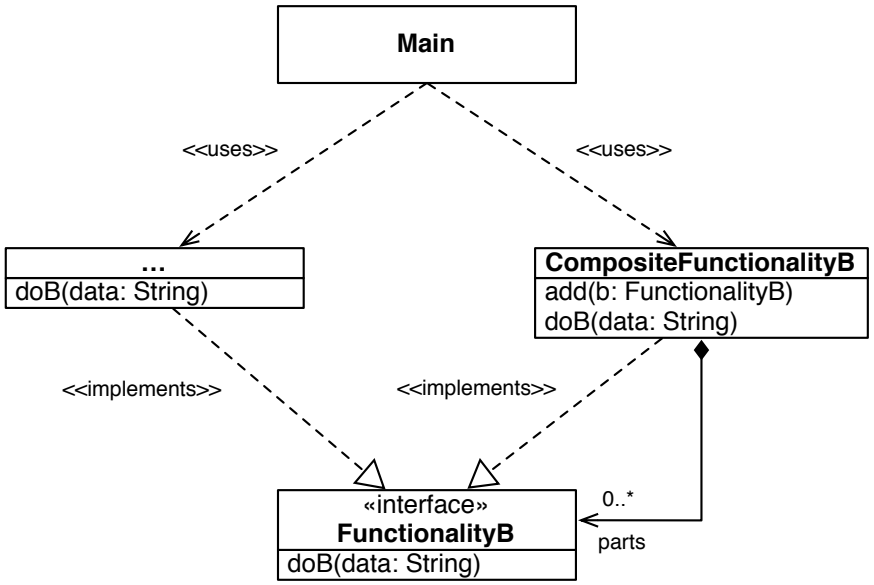

Figure 4: UML class diagram for a composite callback

Figure 4 shows a UML class diagram for the composite callback. An instance of a `CompositeFunctionalityB` is (behaves as) a `FunctionalityB`, and also contains a set of `FunctionalityB`s. Each of these parts could be a specific callback for functionality B, such as a `DataPrinter` or `DataCounter`, or in turn another composite callback.

Why would you use a hierarchy, rather than a flat structure? Because certain combinations of callbacks are also used in other places, and can be set up once, to be coupled later as one group to various functionalities A. If such a group needs to be combined with another callback or group of callbacks, you get a hierarchy. Furthermore, you can unregister such a group as a whole in one action, without affecting other registered callbacks.

7 Modifying and Filtering Data as It Is Transferred

When there is a need to vary the processing of the data items in functionality B, you can modify an existing callback or copy-and-edit it. Alternatively, you can extend (subclass) an existing callback.

Neither of these solutions is flexible. Imagine that you want other variations, and also of other callbacks. The number of possible combinations grows fast: m variations of n callbacks would give rise to $m \times n$ new callbacks. And then we do not even consider the desire of combining multiple variations applied to a callback.

A better approach is to put each variation in a separate method, that can be applied to data items before they are passed on to the callback to be varied. So, the variation behaves like a `FunctionalityB`, that passes modified data on to another `FunctionalityB`.

A variation in our example could be to put quotes around the data item. This is accomplished with a `DataQuoter`:

```
1 class DataQuoter implements FunctionalityB {
2     private FunctionalityB destination;
3     public DataQuoter(FunctionalityB destination) {
4         this.destination = destination;
5     }
6     public void doB(String data) {
7         destination.doB('"' + data + '"');
8     }
9 }
```

You might even suppress some calls depending on a condition, as in this `DataFilter`:

```
1  class DataFilter implements FunctionalityB {
2      private FunctionalityB destination;
3      private String pattern;
4      public DataFilter(FunctionalityB destination, String pattern) {
5          this.destination = destination;
6          this.pattern = pattern;
7      }
8      public void doB(String data) {
9          if (data.contains(pattern)) {
10             destination.doB(data);
11         }
12     }
13 }
```

We now only need $m + n$ callbacks, and can also combine variations easily:

```
new DataFilter(new DataQuoter(new DataPrinter("PQF")), "1")
```

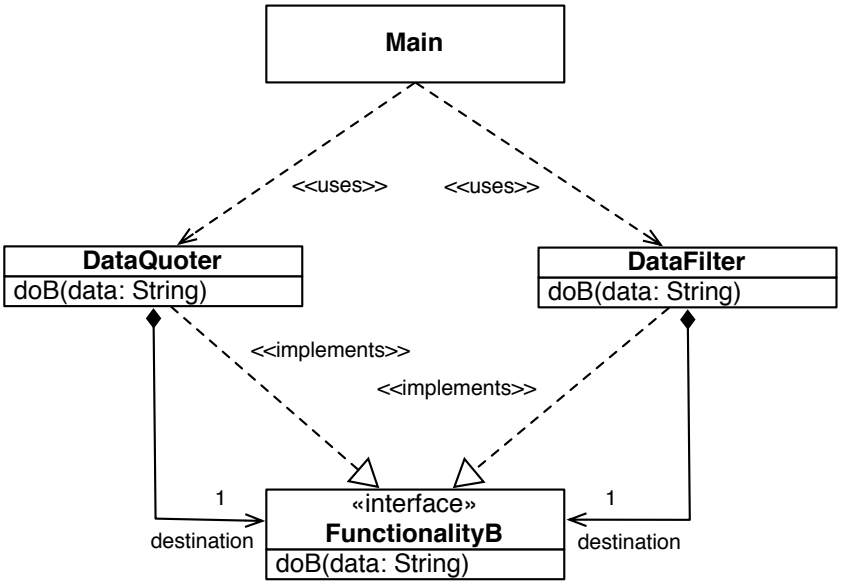

Figure 5: UML class diagram for a callback modifier or filter

This is an example of the *Decorator* Design Pattern. Figure 5 shows a UML class diagram for the callback modifier or filter. An instance of a `DataFilter` is (behaves as) a `FunctionalityB`, but also uses another `FunctionalityB` to complete the processing. In case of multiple modified callbacks, the duplicate code of the various decorators can be moved to a common base class (see Figure 6).

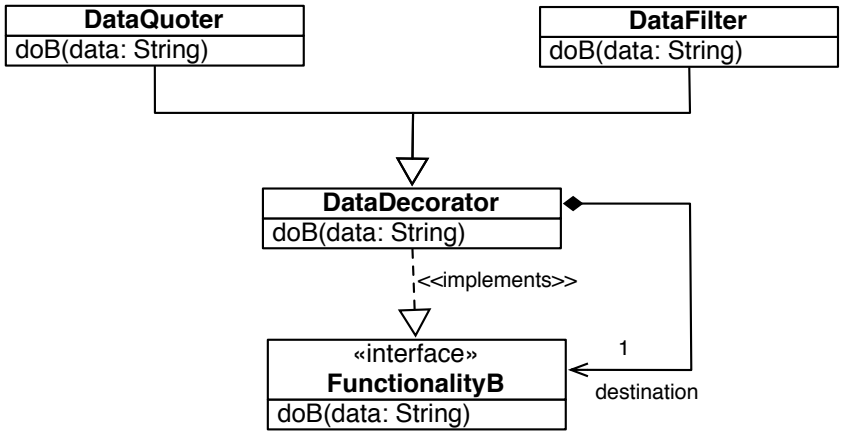

Figure 6: UML class diagram for a callback decorator base class

8 Data Processing Toolkit and Trade-offs

With all these techniques (callback interface, composite, adapter, decorator) you can construct an elegant and flexible toolkit for the processing of data items in callbacks. But this flexibility does have a price:

- there is a performance penalty because of all indirections and data transfers;
- the configuration needs to be programmed out and is created at run time.

The latter invites the introduction of a *Domain Specific Language* to describe configurations (but that is a more advanced topic).

Some further variations are discussed in Sections 10 through 12. Also see Exercises 3 and 6.

9 Listeners

You can also look at all this in a slightly different way. And that is exactly what is done in Java when coupling a graphical user interface (GUI) to the actual functionality of an application.

The graphical user interface is driven by the user via the hardware (keyboard, mouse, touch screen, microphone, etc.). The user serves as functionality A; the hardware details are hidden from the application. The actual processing can be compared to functionality B, which is all you need to program for the application.

Each user action is the *source* of an *event*, produced in the (hidden) *main event loop* (functionality A). The occurrence of an event results in the call of a method that *handles* the event. A class that provides such event handler methods is referred to as an *event listener*, or *listener* for short (functionality B).

Thus, the applications becomes a collection of event listeners driven from the (invisible) main event loop. Event handling methods are nothing more than callbacks of the application.

The signatures of event handling methods are defined in appropriate interfaces. For example[5]:

```
ActionListener
KeyListener
MouseListener
MouseMotionListener
```

In the `MouseListener` interface, there is, among others, a method

```
void mouseClicked(MouseEvent e)
```

[5] See `docs.oracle.com/javase/tutorial/uiswing/events/api.html`

which is called in response to the user clicking the mouse button. The parameter `MouseEvent e` communicates information about the location of the pointer and which mouse button was clicked, and how often it was clicked.

Any source of events and corresponding event handlers can be coupled by callbacks. The terminology is different, but the technique is the same.

10 Observers

Yet another —more general way— to describe this, involves the following terminology. We have a *subject* or *observable* (compare this to functionality A) of which certain state changes require external processing, done by *observers* (functionality B). To that end, the subject *notifies* or *updates* the observers, by calling an agreed callback. Different terminology, same technique.

For performance reasons, it can be useful to avoid passing the observer (callback) to the subject (functionality A) with every invocation of `doA`. Moreover, it can be useful to have the ability to change the observer dynamically (at run time). This can be achieved by storing the observer in the subject in a private instance variable, which can be set in the constructor, and/or via a separate setter method. To support multiple observers, we could use a composite observer:

```
1     CompositeObserverB cob = new CompositeObserverB();
2     cob.add(new PrintingObserverB("B1"));
3     cob.add(new PrintingObserverB("B2"));
4     SubjectA generator1 = new SubjectA("A1");
5     generator1.setObserver(cob);
6     generator1.doA(3);
```

In practice, observer management is often built into the subject. This is less flexible, because you cannot reuse a distribution network across multiple subjects. And it gives rise to unnecessary code duplication. Therefore, we recommend composite observers as external observer managers.

The terminology for managing the collection of observers varies:

- add, remove;
- attach, detach;
- register, unregister;
- subscribe, unsubscribe;
- connect, disconnect.

This view on callbacks is also known as the *Observer* Design Pattern.

11 Push to Observer, or Pull from Subject, or Both

So far, we let the subject **push** data to the observer directly. However, not every observer has the same interests. For instance, the data counters in Sections 4 and 5 ignored the content of the data completely.

Rather than letting the subject decide what data to communicate to observers, that decision can be left to the individual observers. The subject only has to **notify** the observer and can leave it to each observer to **pull** relevant data from the subject (using other methods), if so needed.

However, if all observers basically need the same data, then the pulling strategy introduces unnecessary duplication of work, because all observers will query the subject in the same way. In the push strategy, there is one query, executed by the subject itself, and pushed to all observers.

Here is an example where the subject notifies, and the observers pull.

The subject with functionality A implements the interface `ObservableB`, so that it can be observed by observers with functionality B.

```
1  public interface ObservableB {
2      void setObserver(ObserverB observer);
3      String getData();
4  }
```

An observer with functionality B implements interface `ObserverB`, so that it can be notified by subjects:

```
1  public interface ObserverB {
2      void notifyB(ObservableB subject);
3  }
```

The observer needs to know the identity of the subject that sent the notification, in order to pull data from that subject. Therefore, the notifying subject is a parameter to `notifyB`. This parameter can also be useful when observing multiple subjects.

Here is a concrete subject that produces `n` data items:

```
1  public class SubjectA implements ObservableB {
2      private final String name;
3      private ObserverB observer;
4      private String data;
5      public SubjectA(String name) {
6          this.name = name;
7      }
8      public void setObserver(ObserverB observer) {
9          this.observer = observer;
10     }
11     public String getData() {
```

```
12          return data;
13      }
14      public void doA(int n) {
15          for (int i = 0; i < n; ++ i) {
16              data = i + " produced by " + name; // "complex" computation
17              observer.notifyB(this);
18          }
19      }
20  }
```

Note that the observer should not be **null** when `doA` is called with `n > 0`. This is a precondition to be ensured by proper configuration of `SubjectA`. If you want to execute `doA` without observer, you need to set the observer, for example, to an empty `CompositeObserverB` or to `EmptyObserverB`:

```
1  public class EmptyObserverB implements ObserverB {
2      public void notifyB(ObservableB subject) {
3          // empty implementation, doing nothing
4      }
5  }
```

A concrete observer that retrieves the data and prints it is defined as follows:

```
1  public class PrintingObserverB implements ObserverB {
2      private final String name;
3      public PrintingObserverB(String name) {
4          this.name = name;
5      }
6      public void notifyB(ObservableB subject) {
7          final String data = subject.getData();
8          System.out.println(name + " processed " + data);
9      }
10 }
```

Figure 7 shows a UML class diagram for the example above with pulling observers. `EmptyObserverB` is not shown.

It is also possible to combine push and pull.

In this example, we have used data of type `String`. It is possible to define the observer structure generically, and abstract from the specific data involved.

12 Update or Query the Observers, or Both

Pushing of data to observers is also said to **update** the observers. Instead of updating the observer, the subject might want to **query** the observer and pull data from it. Or, the subject could notify the observer, which then can decide to push

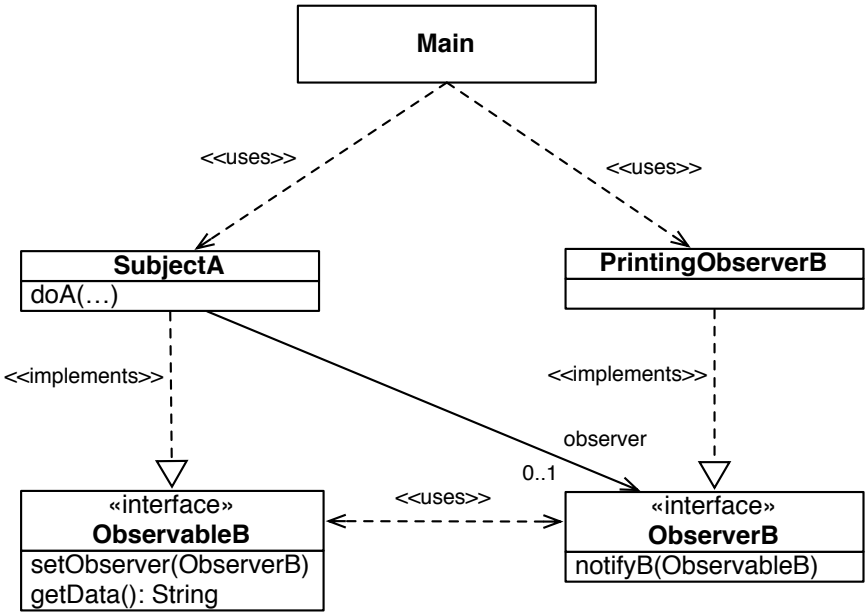

Figure 7: UML class diagram for observers that pull

some data to the subject (using other methods). In case of querying, 'observer' is a misnomer, since it now acts as a source.

Again, it is also possible to have a combination of communicating data to and from observers. With a little more thought, you can turn this into a generic multi-party communication facility (a bit like the internet, with addressable parties and data packages).

13 Testing

In a full-blown application, all these facilities need to be tested. Testing of callbacks, listeners, and observers, and related adapters, composites, and decorators, requires some ingenuity. Such testing is beyond the scope of this note, and will be presented in another note. Because of the separation of concerns, all these facilities are independently testable.

14 Concluding Remarks

All the mechanisms discussed here, that is, callbacks, functions-as-parameter, listeners, and observers, aim to accomplish the same thing. It is not about how to name it, who takes the initiative, or in what direction the data travels. In the end,

it is all about *decoupling*, that is, reducing dependencies, and about avoiding *code duplication*. Decoupling makes it easier to test and to change things, for instance, when fixing defects, when creating a new version or variant, or when re-configuring a system at run time.

Which approach is to be favored depends on what data is needed when and where. Design patterns need to be applied with consideration, taking the specific problem context into account.

References

[1] Eddie Burris, *Programming in the Large with Design Patterns*. Pretty Print Press, 2012.

[2] Eric Freeman, Elisabeth Freeman. *Head First Design Patterns*. O'Reilly Media, 2004.

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. Also known as the Gang-of-Four Book, or GOF Book.

[4] Oracle. *The Java™ Tutorials: Lambda Expressions*. `docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`

[5] Oracle. *The Java™ Tutorials: Method References*. `docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html`

[6] Wikipedia, *Callback*, `en.wikipedia.org/wiki/Callback_(computer_programming)` (accessed 11 Nov. 2012).

[7] Wikipedia, *Software Design Pattern*, `en.wikipedia.org/wiki/Software_design_pattern` (accessed 30 Oct. 2012).

A `EventListener` Interface

In Java, callback signatures are often defined as extension of the predefined interface `EventListener`. This interface is known as a *marker interface*, that is, an empty interface used to mark callbacks. That way, they can easily be recognized as such, for example by the compiler or by using `instanceOf()`. Since Java 5, marking can also be done through *annotations* (also see Appendix B).

B Java 8

Java 8 has improved support for functional programming, including functions as parameters (callbacks). In a way, it works through some abbreviations. An inter-

face is called a **functional interface** when it defines exactly one abstract method. Such a functional interface serves as a type for 'pure' functions, viz. functions that implement its (one and only) abstract method. So, `FunctionalityB` is a functional interface, and it is a type of **`void`** functions that apply to one `String` argument.

A **lambda expression** [4] defines an anonymous 'pure' function using the syntax

```
(param_1, param_2, ..., param_N) -> body
```

where each `param_i` defines a typed and named positional parameter (as with traditional methods), and `body` is either a statement (for a **`void`** function) or an expression of some type `T` (for a function that returns a result of type `T`). The parameter types often can be omitted, since they can be inferred from the context of usage in the body. When there is exactly one parameter, the surrounding parentheses can be omitted as well (in that case the type must be omitted).

Wherever an object that implements a functional interface `F` is expected, one can substitute a lambda expression that matches `F`. The lambda expression above matches any functional interace that defines an abstract method of the form

```
T m(param_1, param_2, ..., param_N);
```

where `T` is the type of expression `body`, or **`void`** if `body` is a statement. If the abstract method of `F` is named `m`, then the lambda expression above is (almost) equivalent to the following anonymous (local) class instantiation:

```
1  new F() {
2      @Override
3      public void m(param_1, param_2, ..., param_N) {
4          body;  // when body is a statement
5      }
6  }
```

or

```
1  new F() {
2      @Override
3      public T m(param_1, param_2, ..., param_N) {
4          return body;  // when body is an expression
5      }
6  }
```

where the parameter types and return type are inferred from `F` as needed. Thus, we can invoke `FunctionalityA.doA()` with a lambda expression as callback:

```
1  doA(n, (String data) -> System.out.println(data));
2  doA(n, (data) -> System.out.println(data));
3  doA(n, data -> System.out.println(data));
```

A lambda expression is not fully equivalent to an anonymous class instantiation. For instance, there is a difference in scope of local variables. A lambda expression does not introduce a new scope local to that expression; instead, it 'lives' in the enclosing scope. This also affects the interpretation of the keyword **this**.

In Java 8, a functional interface can be marked by a new predefined annotation `@FunctionalInterface`, rather than by extending the `EventListener` interface (Appendx A). With such an annotation, the compiler will check that the annotated interface indeed qualifies as a functional interface.

The new package `java.util.function` defines several generic functional interfaces, including `Consumer<T>`, `Supplier<T>`, `Function<T, R>` and `Predicate<T>`. In essence, these are defined as

```
@FunctionalInterface
public interface Consumer<T> {
    public void accept(T t);
}

@FunctionalInterface
public interface Supplier<T> {
    public T get();
}

@FunctionalInterface
public interface Function<T, R> {
    public R apply(T t);
}

@FunctionalInterface
public interface Predicate<T> {
    public boolean test(T t);
}
```

Thus, the callback in functionality `A` does not need the application-specific type `FunctionalityB`, but could be using the type `Consumer<String>`.

Any implemented method or constructor can serve as a 'pure' function via a **method reference** [5]. The general syntax is

```
qualifier :: methodName
```

where the qualifier is either a class type or an expression evaluating to an object, and where the keyword **new** serves as 'method name' for a constructor. You can also view this syntax as an abbreviation for some common lambda expressions. Given the class `C` and object `obj` defined by

```
public class C {
    public C(U arg, ...) { ... } // constructor
```

```
3       public static T s(U arg, ...) { ... } // static method
4       public T m(U arg, ...) { ... } // instance method
5   }
6
7   C obj = new C(...);
```

we have these relationships:

Method reference	Lambda expression
`C :: s`	`(U arg, ...) -> C.s(arg, ...)`
`obj :: m`	`(U arg, ...) -> obj.m(arg, ...)`
`C :: m`	`(C o, U arg, ...) -> o.m(arg, ...)`
`C :: new`	`(U arg, ...) -> C(arg, ...)`
`int[] :: new`	`n -> new int[n]`

In Java 8, the interface `Iterable<T>` now has a default method `forEach()`, whose definition is equivalent to:

```
1       void forEach(Consumer<? super T> action) {
2           for (T t : this) {
3               action.accept(t);
4           }
5       }
```

For example, if we have a variable `Set<String> set`, then

```
set.forEach( s -> System.out.print(s + " ") );
```

will print all elements in the set. And if you just want to print the items, you can use a method reference:

```
set.forEach( System.out :: println );
```

Finally, the new package `java.util.stream` provides support for **streams**[6] of elements with numerous predefined functional operators for sequential and parallel aggregation. For instance, a `Collection` can be turned into a stream by the new method `stream()`. Here is code that takes the set of `String` from above, converting it into a stream, keeping strings whose length exceeds 2, converting it into a stream of **`int`**, and finally summing this stream:

```
1       int result = set.stream()
2           .filter( s -> s.length() > 2 )
3           .mapToInt( String :: length )
4           .sum();
```

A stream builder can be used to convert data items, that are generated one by one via a `Consumer<T>`, into a `Stream<T>` as follows:

[6] In functional programming better known as lists or sequences.

```
1     Stream.Builder<String> builder = Stream.builder(); // step 1
2     FunctionalityA.doA(n, builder :: accept); // step 2
3     // doA(n, builder), if FunctionalityB = Consumer<String>
4     Stream<String> stream = builder.build(); // step 3
5     stream.filter( s -> s.contains(pattern) )
6             .map( s -> '"' + s + '"' )
7             .forEach( System.out :: println );
```

Notes:

- A stream builder operates in four sequential steps:
  1. An empty builder is created via `Stream.builder()`.
  2. The builder is filled via zero or more calls of `accept(t)`, or `add(t)`.
  3. The builder is converted to a stream via `build()`.
- A stream can be processed only once, and cannot be reused.

C Exercises

1. Using the example code from `CallbackExamples.zip`, run the various examples to understand better how the proposed solutions work. The following entities are relevant, ordered from abstract to concrete:
   - **`interface`** `FunctionalityB` with method `doB`
   - **`class`** `FunctionalityA` with method `doA`
   - **`class`** `DataPrinter` **`implements`** `FunctionalityB`
   - **`class`** `Counter` with methods `count` and `getCount`
   - **`class`** `DataCounter` **`extends`** `Counter` **`implements`** `FunctionalityB`
   - **`class`** `CounterAdapter` **`implements`** `FunctionalityB` using a `Counter`
   - **`class`** `CompositeFunctionalityB` **`implements`** `FunctionalityB` with `List<FunctionalityB> parts` and method `add`
   - **`class`** `DataQuoter` **`implements`** `FunctionalityB`
   - **`class`** `DataFilter` **`implements`** `FunctionalityB`
   - **`class`** `Main` with method `main`

   The following experiments are supported in `Main.main()`.

   (a) Stateless callback in §3: `run Main a 3 B1`

   (b) Anonymous stateless callback in §4: `run Main b 3`

   (c) Callback with state, using an adapter via inheritance in §5: `run Main c 3`

(d) Callback with state, using an adapter via composition in §5: `run Main d 3`

(e) Composite callback in §6: `run Main e 3 B2`

(f) Decorator callback in §7: `run Main f 3`

(g) Decorator callback that filters in §7: `run Main g 12 1`

You can also try other parameters in the experiments.

2. Incorporate the code of the ad hoc counter `AdHocCounter` from §4 in a new method `Main.main2()`, and run it.

3. Implement the callback distribution network of Figure 8 in a new method `Main.main3()`, and run it.

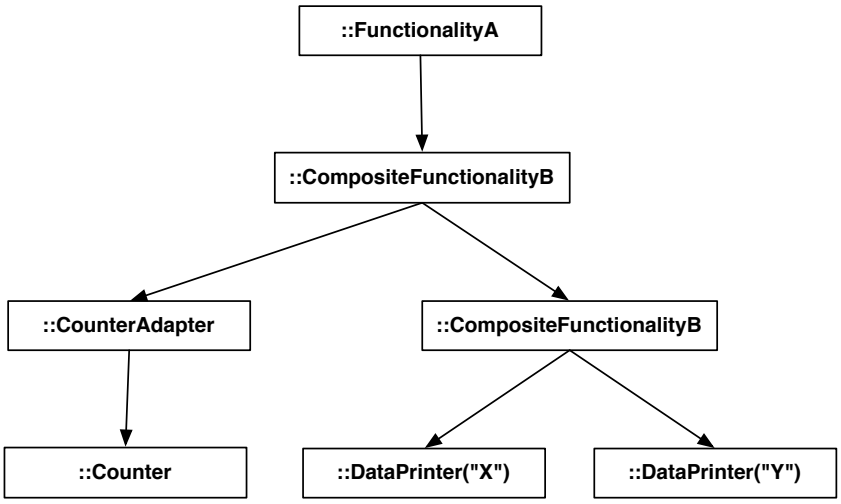

Figure 8: Callback network with multiple composite callbacks

4. Incorporate the code with the combined decorators of §7 in a new method `Main.main4()`, and run it:

```
new DataFilter(new DataQuoter(new DataPrinter("PQF")), "1")
```

5. Move the duplicate code in `DataQuoter` and `DataFilter` to a common base class `DataDecorator` (also see Figure 6).

6. Implement the callback distribution network of Figure 9 in a new method `Main.main5()`, and run it.

7. Implement and use a new adapter `StatisticsAdapter` for given non-callback class `Statistics` with a method `void update(int)`. Use the approach via composition. Let `FunctionalityA` produce a given number of random numbers from a given range, and obtain the mean and standard deviation of the generated numbers through the adapted `Statistics`.

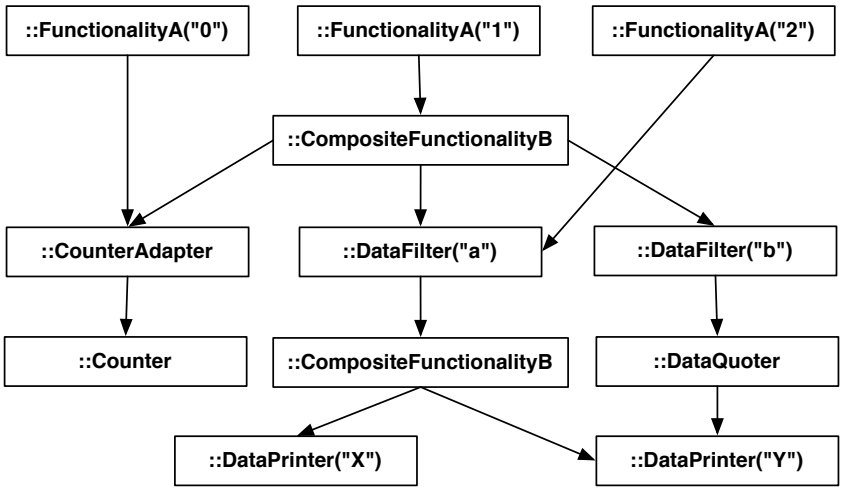

Figure 9: Callback network with multiple composite callbacks and sources

8. Implement and use a new callback decorator `RepeatFunctionalityB` that repeats each received call a given number of times. The repeat count is set in the constructor of the decorator.

9. Implement and use a variant of functionality A (a subject) with a stored and settable callback (observer).

10. Implement code to use `SubjectA`, which is a variant of `FunctionalityA`, where callbacks (observers) *pull* data. That is, using an `ObserverB` interface. See §11. Note

    - that `SubjectA` offers a method to pull data by implementing the interface `ObservableB`.
    - that callbacks need to know the actual subject; the identity is passed as parameter in the callback.

Index