2IP15 Programming Methods

From Small to Large Programs

Tom Verhoeff
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

www.win.tue.nl/~wstomv/edu/2ip15

Overview

- Replacing functionality at run-time
- Observer Design Pattern: *listeners*

Design Patterns

- A Design Pattern is an *outline* of a *general* solution to a design problem, *resuable* in *multiple, diverse, specific* contexts.
- It is *not* a solution in the form of source code that can be reused "as is".
- Example: Strategy, Iterator, Observer
- The term *Design Pattern* comes from architecture and was made popular for software by the book

  Design Patterns: Elements of Reusable Object-Oriented Software

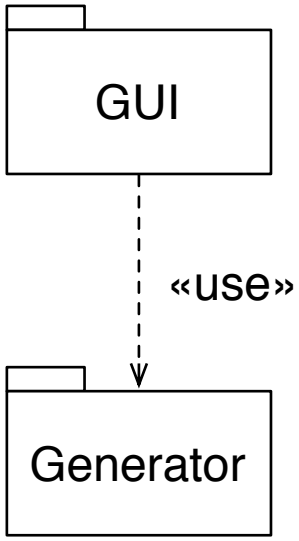
  Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1995.

  The authors are commonly referred to as the Gang of Four (GoF).

A (recursive) generator of combinatorial objects

```
1  /**
2   * @pre   0 <= k <= n
3   * @post  each bit pattern with prefix s and n additional bits,
4   *   of which k 1's, has been printed exactly once
5   * @bound n
6   */
7  public static void generate(String s, int n, int k) {
8      if (n == 0) {  // k == 0, base case
9          System.out.println(s);
10     } else {  // 0 < n, inductive step
11         for (int b = 0; b <= 1; ++ b) {
12             if (0 <= k - b && k - b <= n - 1) {
13                 generate(s + b, n - 1, k - b);
14             }
15         }
16     }
17 }
```

A GUI on top of the generator



Package `GUI` has checkboxes to select item processing and a button to activate the `Generator`.

Package `Generator` has method `generate(...)`.

`GUI` is a client of `Generator`.

The Problem of Replacing Functionality at Run-time

- Consider a module doing some (complicated) computation.

  E.g. generating combinatorial objects, or solving a binary puzzle.

- Part of the computation should be replaceable at run-time.

  E.g. how to process generated items: count, display, write to file.

- That is, the computation involves client-supplied functionality.

  This functionality may change at run-time, e.g. selected via a GUI.

- Hard-coding that functionality in separate methods does not help:

  It cannot be replaced easily at run time.

Generator with tweakable functionality: not so good solutions

- Could replace `System.out.println(s)` by method call `process(s)`
- Where to define method `process(String)`?
  - In client environment: creates a cyclic dependency
  - Inside computation module: harder to tweak functionality
- How to tweak functionality at run-time?
  - Via inspection of conditions in client environment: **`if`** `(...)`
- *Reuse* and *maintenance* are harder,
  when link to actual `process()` or data it needs is hard-coded.

Intermezzo: Why Reuse via Copy-Edit Is a Bad Idea

- When the processing of items is interwoven with the generator, it is hard to change the generator or the processing independently.
- When the processing is done in a separate method, such changes are easier.
- Reusing the generator in another program by copying it and editing the processing method to suit the new application can be made to work (especially if processing is done in a separate method).
- However, it now becomes inconvenient to change (enhance, improve) the generator, because there are multiple copies to be updated.
- Hence, we seek a looser form of coupling for the generator and the processing.

Solution 1 for Replacing Functionality at Run-time: Iterator

- Offer the computation as an *iterator*,
  doing the client-part of the computation outside the generator:

```
for (String s : generator) {
    process(s);
}
```

- That way, the core of the computation resides inside the iterator;
  the calling environment iterates while doing its own computation.

- Limitations:

  – Harder to do for recursive computations.
    (Beyond scope)

  – Can only supply functionality that handles each computed item.

Solution 2 for Replacing Functionality at Run-time: Callbacks

- Provide one or more methods as *parameter* to the computation:

  ```
  void generate(String s, int n, int k, _method_ p(String))
  ```

  Parameter `p()` is known as a *callback* method.

- In principle, parameterization provides ideal decoupling:

  Client defines the processing to be done in a callback method, and tells generator at call-time which method to call for processing.

- Possible in C, C++, Object Pascal, Python, but not in Java.

- In Java, need to pass an object as carrier for callback method.

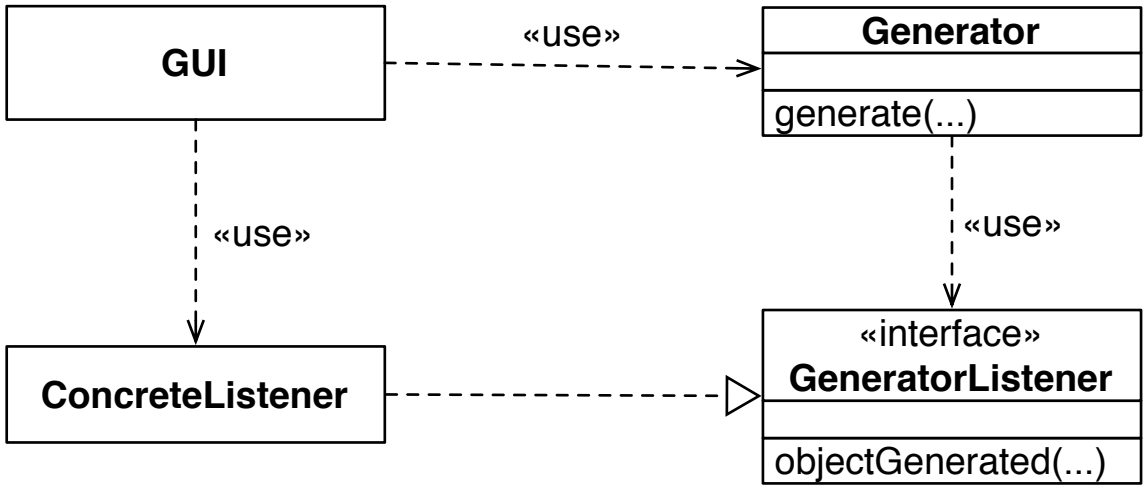
Events and Listeners

- Report events during the computation by calling appropriate (callback) methods of a listener object.
- Event = call of method introduced for this purpose.

  An event can have (event-dependent) parameters.
- Listener = (client-supplied) object implementing event methods.
- Listener interface defines signatures of event methods.
- Listener can have *local state*.

  E.g. count number of events.

A GUI for the generator with listener interface



`GUI` provides listener object to `Generator`.

Listener object implements event method `objectGenerated`.

`Generator` calls event method of listener object.

There is no cyclic dependency at compile-time.

Listener Interface for Generators

```
1  import java.util.EventListener;
2
3  /**
4   * Interface with events that can be triggered by a generator.
5   */
6  public interface GeneratorListener extends EventListener {
7
8      /**
9       * Reports a combinatorial object.
10      *
11      * @param object  the combinatorial object
12      */
13     void objectGenerated(String object);
14
15 }
```

Defining a Stateless Listener for Generators

```
1       /**
2        * Listener that prints the objects generated.
3        */
4       static class Printer implements GeneratorListener {
5
6           public void objectGenerated(String object) {
7               System.out.println(object);
8           }
9       }
```

Defining a Generator Listener with State

```
1     /**
2      * Listener that counts the objects generated.
3      */
4     static class Counter implements GeneratorListener {
5         private int count = 0; // number of objects generated
6
7         public void objectGenerated(String object) {
8             ++ count;
9         }
10
11        public int getCount() {
12            return count;
13        }
14    }
```

Bit Pattern Generator with Listener Parameter

```
 1 public class GeneratorPlain {
 2     public static void generate(String s, int n, int k,
 3             GeneratorListener listener) {
 4         if (n == 0) {  // k == 0, base case
 5             if (listener != null) {
 6                 listener.objectGenerated(s);
 7             }
 8         } else {  // 0 < n, inductive step
 9             for (int b = 0; b <= 1; ++b) {
10                 if (0 <= k - b && k - b <= n - 1) {
11                     generate(s + b, n - 1, k - b, listener);
12                 }
13             }
14         }
15     }
16     // N.B. listener is passed down the chain of recursive calls
```

Invoking a Generator with Listener Parameter

```
1     public static void examplePlain(int n, int k) {
2         System.out.println("As parameter: n, k == " + n + ", " + k);
3         GeneratorPlain.generate("", n, k, new Printer());
4         final Counter counter = new Counter();
5         GeneratorPlain.generate("", n, k, counter);
6         final int count = counter.getCount();
7         System.out.println("# combinations = " + count);
8     }
```

```
n, k == 4, 2
0011
0101
0110
1001
1010
1100
# combinations = 6
```

Invoking a Genarator with Anonymous Inner Class

```
1     public static void exampleAnonymous(int n, int k) {
2         System.out.println("Anonymous class: n, k == " + n + ", " + k);
3
4         GeneratorPlain.generate("", n, k, new GeneratorListener() {
5             public void objectGenerated(String object) {
6                 System.out.print(".");
7             }
8         });
9
10        System.out.println();
11    }
```

Generator with Stored Parameters and Settable Listener

```
 1 /**
 2  * Generator for combinatorial objects of n bits with k 1s,
 3  * with one settable listener.
 4  */
 5 public class Generator {
 6
 7     /** A listener, or null */
 8     private GeneratorListener listener;
 9
10     /**
11      * Sets listener for generator events.
12      */
13     public void setListener(GeneratorListener listener) {
14         this.listener = listener;
15     }
```

Generator with Stored Parameters and Settable Listener

```
1     public void generate(int n, int k) {
2         generate("", n, k);
3     }
```

```
1     void generate(String s, int n, int k) {
2         if (n == 0) {  // k == 0, base case
3             if (listener != null) {
4                 listener.objectGenerated(s);
5             }
6         } else {  // 0 < n, inductive step
7             for (int b = 0; b <= 1; ++b) {
8                 if (0 <= k - b && k - b <= n - 1) {
9                     generate(s + b, n - 1, k - b);
10                }
11            }
12        }
13    }
```

Invoking a Generator with Settable Listener

```
1     public static void exampleSettable(int n, int k) {
2         System.out.println("One settable: n, k == " + n + ", " + k);
3
4         final Generator g = new Generator();
5
6         g.setListener(new Printer());
7         g.generate(n, k);
8
9         final Counter counter = new Counter();
10        g.setListener(counter);
11        g.generate(n, k);
12        final int count = counter.getCount();
13        System.out.println("# combinations = " + count);
14    }
```

Generator with Multiple Listeners

```
 1 import java.util.List;
 2 import java.util.ArrayList;
 3
 4 /**
 5  * Generator for combinatorial objects of n bits with k 1s,
 6  * with multiple listeners.
 7  */
 8 public class ObservableGenerator {
 9
10     /** The registered listeners */
11     private final List<GeneratorListener> listeners;
12
13     /**
14      * Constructs a generator without listeners.
15      */
16     public ObservableGenerator() {
17         this.listeners = new ArrayList<GeneratorListener>();
18     }
```

Generator with Multiple Listeners

```
1     /**
2      * Adds a listener for generator events.
3      */
4     public void addListener(GeneratorListener listener) {
5         listeners.add(listener);
6     }
7
8     /**
9      * Notifies all registered listeners.
10     */
11    void notifyListeners(String s) {
12        for (GeneratorListener listener : listeners) {
13            listener.objectGenerated(s);
14        }
15    }
```

Generator with Multiple Listeners

```
1     public void generate(int n, int k) {
2         generate("", n, k);
3     }
```

```
1     void generate(String s, int n, int k) {
2         if (n == 0) {  // k == 0, base case
3             notifyListeners(s);
4         } else {  // 0 < n, inductive step
5             for (int b = 0; b <= 1; ++b) {
6                 if (0 <= k - b && k - b <= n - 1) {
7                     generate(s + b, n - 1, k - b);
8                 }
9             }
10        }
11    }
```

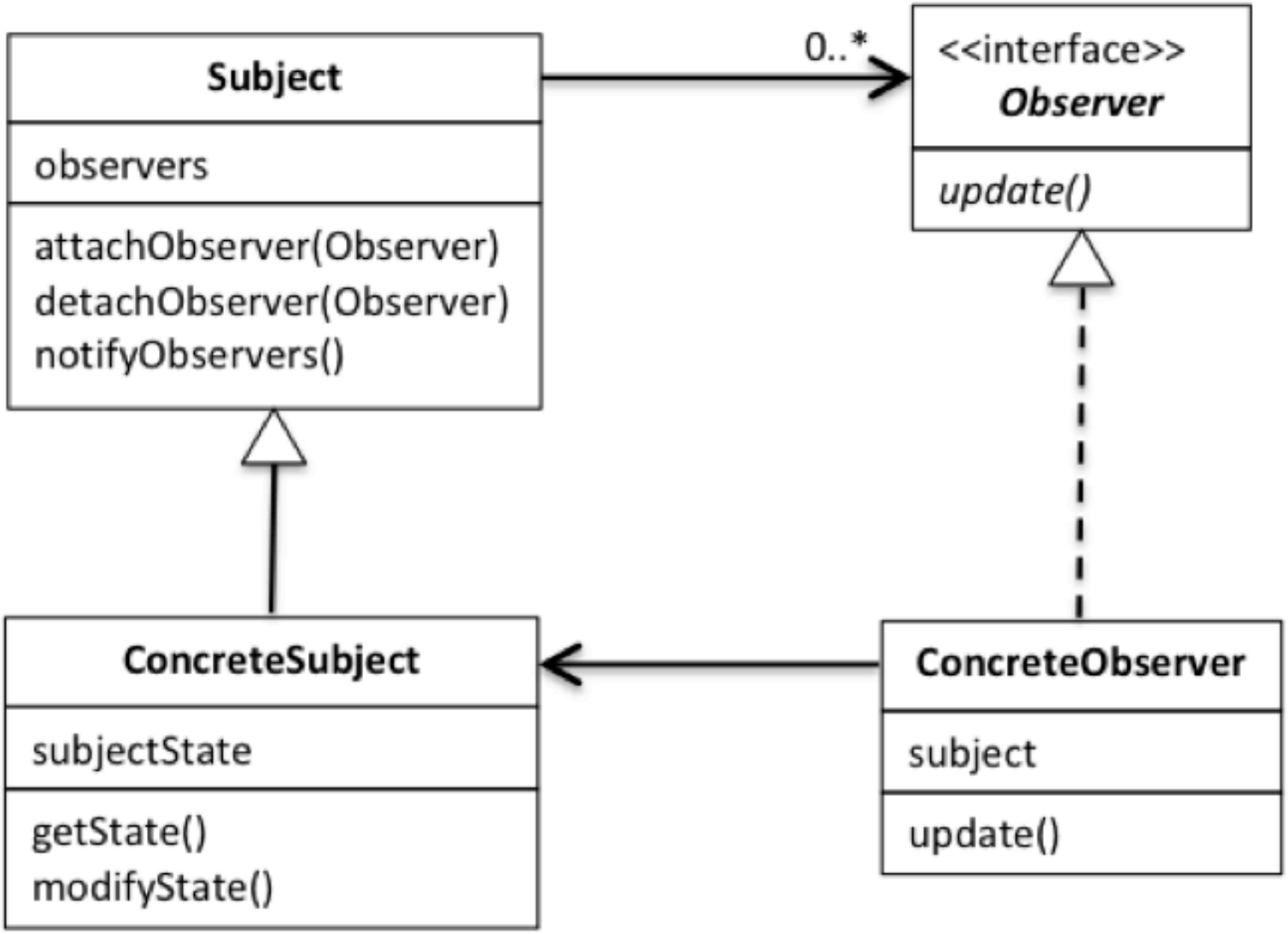
Invoking a Generator with Multiple Listeners

```
1     public static void exampleMulti(int n, int k) {
2         System.out.println("Multi addable: n, k == " + n + ", " + k);
3
4         final ObservableGenerator g = new ObservableGenerator();
5
6         g.addListener(new Printer());
7         final Counter counter = new Counter();
8         g.addListener(counter);
9
10        g.generate(n, k);
11        final int count = counter.getCount();
12        System.out.println("# combinations = " + count);
13    }
```

Observer Design Pattern: Roles

- `Subject`: defines interface for observables (to register, etc.)
- `ConcreteSubject`: implements `Subject` interface
- `Observer`: defines interface for events that can be observed
- `ConcreteObserver`: implements `Observer` interface to handle the observable events
- `Client`: configures concrete subject with concrete observers

Observer Pattern: Class Diagram (from Burris)



Summary

- Processing can be decoupled from a generator via an iterator. But this is not so general.
- Recursion can be eliminated from a generator via an iterator. (This falls outside the scope of this course.)
- Functionality can be decoupled through events and listeners.
- A class could have multiple listeners, reconfigurable by client at run-time.
- There can be multiple events in a listener interface.
- Read Ch.9 in Burris: Observer Design Pattern.