2IPC0 Programming Methods

From Small to Large Programs

Loek Cleophas
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

http://canvas.tue.nl/courses/473

Overview

- GUI organization in Java
- Replacing functionality at run-time
- Observer Design Pattern: *listeners*

Textual versus Graphical User Interface

Textual User Interface: The program controls the user

Graphical User Interface (GUI): The user controls the program:

- The user generates events through keyboard, mouse, etc.
- The main event loop dispatches events, by calling appropriate event handling methods (event handlers).
- Each event handler responds to events.

This way, the control flow in the program is partly invisible, hidden in the main event loop: the program consists of initialization code and a bunch of event handlers, called by the main event loop.

Graphical User Interfaces in Java

Java Foundations Classes:

- Abstract Window Toolkit (AWT): translates to native GUI. Offers the *Look & Feel* of the host operating system.

  Package: `java.awt`

- Swing: a 'light-weight' all-Java GUI library. Offers run-time selectable platform-independent *Look & Feel*.

  Package: `javax.swing`

- 2D Graphics and Imaging

- Drag-and-Drop, multi-threading (concurrency), ...

- Accessibility, Internationalization, ...

GUI Organization

- Components are the building blocks of a GUI: frames (windows), panels, labels, buttons, check boxes, text fields, text areas, . . .

- Components hava a *hierarchical* (tree-like) organization:
  Container components can contain other components.

- A component can generate various events.

  To handle such events, a listener needs to implement the relevant listener interface(s) and must be registered with the component.

- Each component *paints* itself.

  The default paint behavior can be overridden to provide application-specific graphics.

Designing the Look of a GUI in NetBeans

- For each window, create a new `JFrame` descendant via

  File > New File... > Swing GUI Forms > JFrame Form

  and put it in the `gui` package.

  This adds a `.java` and related `.form` file to the project.

- In the IDE, select

  Source   to work on the code view of the window frame

  NetBeans controls the `Generated Code`.

  Design   to work on the graphical view of the window frame

  Drag components from the *Palette* to the frame, and set their properties in the *Properties* panel.

Designing the Feel of a GUI: Handling Button Events

- To handle button events, use `addActionListener()` to register a listener that implements interface

`ActionListener`
`actionPerformed(ActionEvent e)`

- A click on the button results in a call to `actionPerformed()` of each registered listener (callback: handler = strategy).

- In NetBeans, double clicking the button in Design mode will add the relevant Java code, where the body of `actionPerformed()` calls an *event handler* in the form that contains the button.

  You can supply a body for that event handler.

- Example: `InterestCompounder`

A (recursive) generator of combinatorial objects

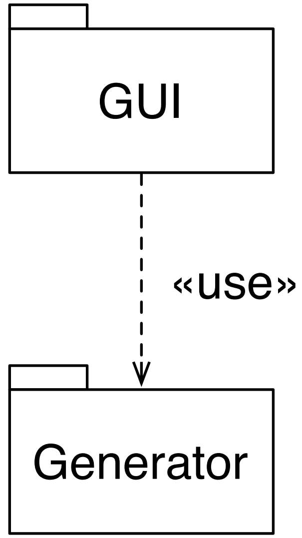
```
/**
 * @pre   0 <= k <= n
 * @post  each bit pattern with prefix s and n additional bits,
 *    of which k 1's, has been printed exactly once
 * @bound n
 */
public static void generate(String s, int n, int k) {
    if (n == 0) {  // k == 0, base case
        System.out.println(s);
    } else {  // 0 < n, inductive step
        for (int b = 0; b <= 1; ++ b) {
            if (0 <= k - b && k - b <= n - 1) {
                generate(s + b, n - 1, k - b);
            }
        }
    }
}
```

Example run of generator

In DrJava, `BitPatterns.generate("", 4, 2)` produces the output

```
0011
0101
0110
1001
1010
1100
```

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 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 needed data is hard-coded.

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

?

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 0 for Replacing Functionality at Run-time: List

- Let the generator return a list of all generated objects.

  Client code can process that list independently of the generator.

- In recursive generators:

  - Needs extra parameter or global variable to accumulate result.

- Disdvantages:

  - Needs enough memory to store entire result.

    Even if items will be processed individually.

  - Prohibits concurrent generation and processing.

    E.g.: User-generated events must be processed immediately.

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; cf. Python generators using `yield`)

  - Can only supply functionality that handles each computed item.

Solution 2: Template Method Pattern

Apply Template Method Pattern:

- Define the generator algorithm as a *template method*
- Define **`void`** `process(String s)` as a *hook method*

  Either **`abstract`**, or with a default implementation
- In client, subclass the generator and override `process` with desired functionality
- Advantage: can insert several steps in different places in generator
- Disadvantage: client code can only be used with specific generator

Solution 3: Strategy Pattern

Apply Strategy Pattern:

- Specify the method **void** `process(String s)` in an (abstract) class or interface
- Implement it by overriding `process()` in a subclass
- Pass an object of this subclass to the generator

  Also known as *Dependency Injection* (DI)
- Advantage: client code is less coupled to generator
- Disadvantage: there is more overhead (viz. to configure)

Solution 4: 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 pre 8. (N.B. This is possible in Java 8 using lambda expressions.)

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

  Cf. Strategy Pattern, and *From Callbacks to Design Patterns*

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.

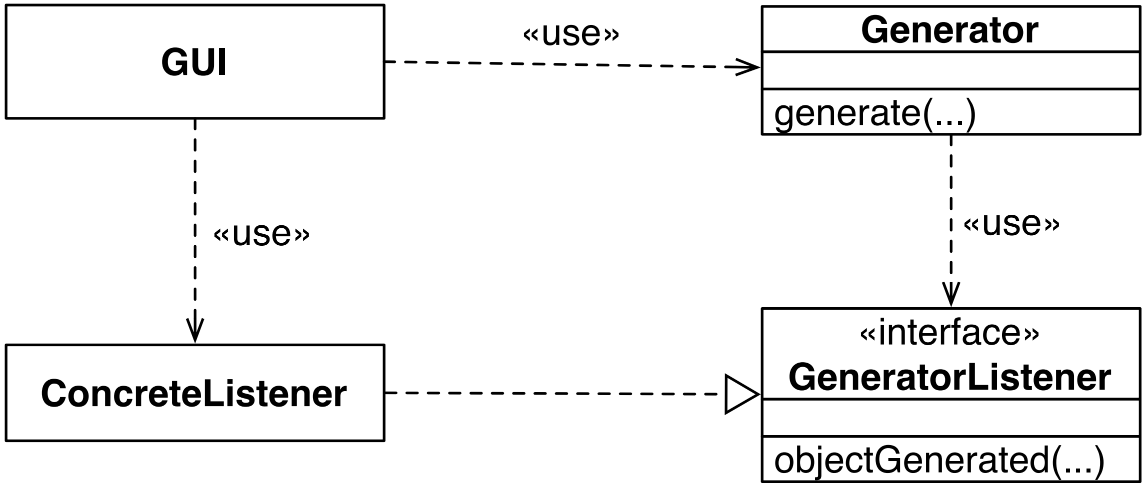
- Listener = (client-supplied) object implementing event methods.

- 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

```java
import java.util.EventListener;

/**
 * Interface with events that can be triggered by a generator.
 */
public interface GeneratorListener extends EventListener {

    /**
     * Reports a combinatorial object.
     *
     * @param object  the combinatorial object
     */
    void objectGenerated(String object);

}
```

Defining a Stateless Listener for Generators

```java
/**
 * Listener that prints the objects generated.
 */
static class Printer implements GeneratorListener {

    public void objectGenerated(String object) {
        System.out.println(object);
    }
}
```

Defining a Generator Listener with State

```java
/**
 * Listener that counts the objects generated.
 */
static class Counter implements GeneratorListener {
    private int count = 0; // number of objects generated

    public void objectGenerated(String object) {
        ++ count;
    }

    public int getCount() {
        return count;
    }
}
```

Bit Pattern Generator with Listener Parameter

```java
public class GeneratorPlain {
    public static void generate(String s, int n, int k,
            GeneratorListener listener) {
        if (n == 0) {  // k == 0, base case
            if (listener != null) {
                listener.objectGenerated(s);
            }
        } else {  // 0 < n, inductive step
            for (int b = 0; b <= 1; ++b) {
                if (0 <= k - b && k - b <= n - 1) {
                    generate(s + b, n - 1, k - b, listener);
                }
            }
        }
    }
    // N.B. listener is passed down the chain of recursive calls
```

Invoking a Generator with Listener Parameter

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

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

Invoking a Genarator with Anonymous Inner Class

```java
    public static void exampleAnonymous(int n, int k) {
        System.out.println("Anonymous class: n, k == " + n + ", " + k);

        GeneratorPlain.generate("", n, k, new GeneratorListener() {
            public void objectGenerated(String object) {
                System.out.print(".");
            }
        });

        System.out.println();
    }
```

Generator with Stored Listener, Given in Constructor

```java
/**
 * Generator for combinatorial objects of n bits with k 1s,
 * with one fixed listener set in constructor.
 */
public class GeneratorWithFixedListener {

    /** A listener, or null */
    private GeneratorListener listener;

    /**
     * Constructs generator with given listener for generator events.
     */
    public GeneratorWithFixedListener(GeneratorListener listener) {
        this.listener = listener;
    }
```

Generator with Stored Listener

```java
public void generate(int n, int k) {
    generate("", n, k);
}

void generate(String s, int n, int k) {
    if (n == 0) {  // k == 0, base case
        if (listener != null) {
            listener.objectGenerated(s);
        }
    } else {  // 0 < n, inductive step
        for (int b = 0; b <= 1; ++b) {
            if (0 <= k - b && k - b <= n - 1) {
                generate(s + b, n - 1, k - b);
            }
        }
    }
}
```

Generator with Settable Listener

```java
/**
 * Generator for combinatorial objects of n bits with k 1s,
 * with one settable listener.
 */
public class GeneratorWithSettableListener {

    /** A listener, or null */
    private GeneratorListener listener;

    /**
     * Sets listener for generator events.
     */
    public void setListener(GeneratorListener listener) {
        this.listener = listener;
    }
```

Invoking a Generator with Settable Listener

```java
    public static void exampleSettable(int n, int k) {
        System.out.println("One settable: n, k == " + n + ", " + k);

        final GeneratorWithSettableListener g =
            new GeneratorWithSettableListener();

        g.setListener(new Printer());
        g.generate(n, k);

        final Counter counter = new Counter();
        g.setListener(counter);
        g.generate(n, k);
        final int count = counter.getCount();
        System.out.println("# combinations = " + count);
    }
```

Generator with Multiple Listeners

```java
import java.util.List;
import java.util.ArrayList;

/**
 * Generator for combinatorial objects of n bits with k 1s,
 * with multiple listeners.
 */
public class ObservableGenerator {

    /** The registered listeners */
    private final List<GeneratorListener> listeners;

    /**
     * Constructs a generator without listeners.
     */
    public ObservableGenerator() {
        this.listeners = new ArrayList<GeneratorListener>();
    }
```

Generator with Multiple Listeners

```java
/**
 * Adds a listener for generator events.
 */
public void addListener(GeneratorListener listener) {
    listeners.add(listener);
}

/**
 * Notifies all registered listeners.
 */
void notifyListeners(String s) {
    for (GeneratorListener listener : listeners) {
        listener.objectGenerated(s);
    }
}
```

Generator with Multiple Listeners

```java
public void generate(int n, int k) {
    generate("", n, k);
}

void generate(String s, int n, int k) {
    if (n == 0) {  // k == 0, base case
        notifyListeners(s);
    } else {  // 0 < n, inductive step
        for (int b = 0; b <= 1; ++b) {
            if (0 <= k - b && k - b <= n - 1) {
                generate(s + b, n - 1, k - b);
            }
        }
    }
}
```

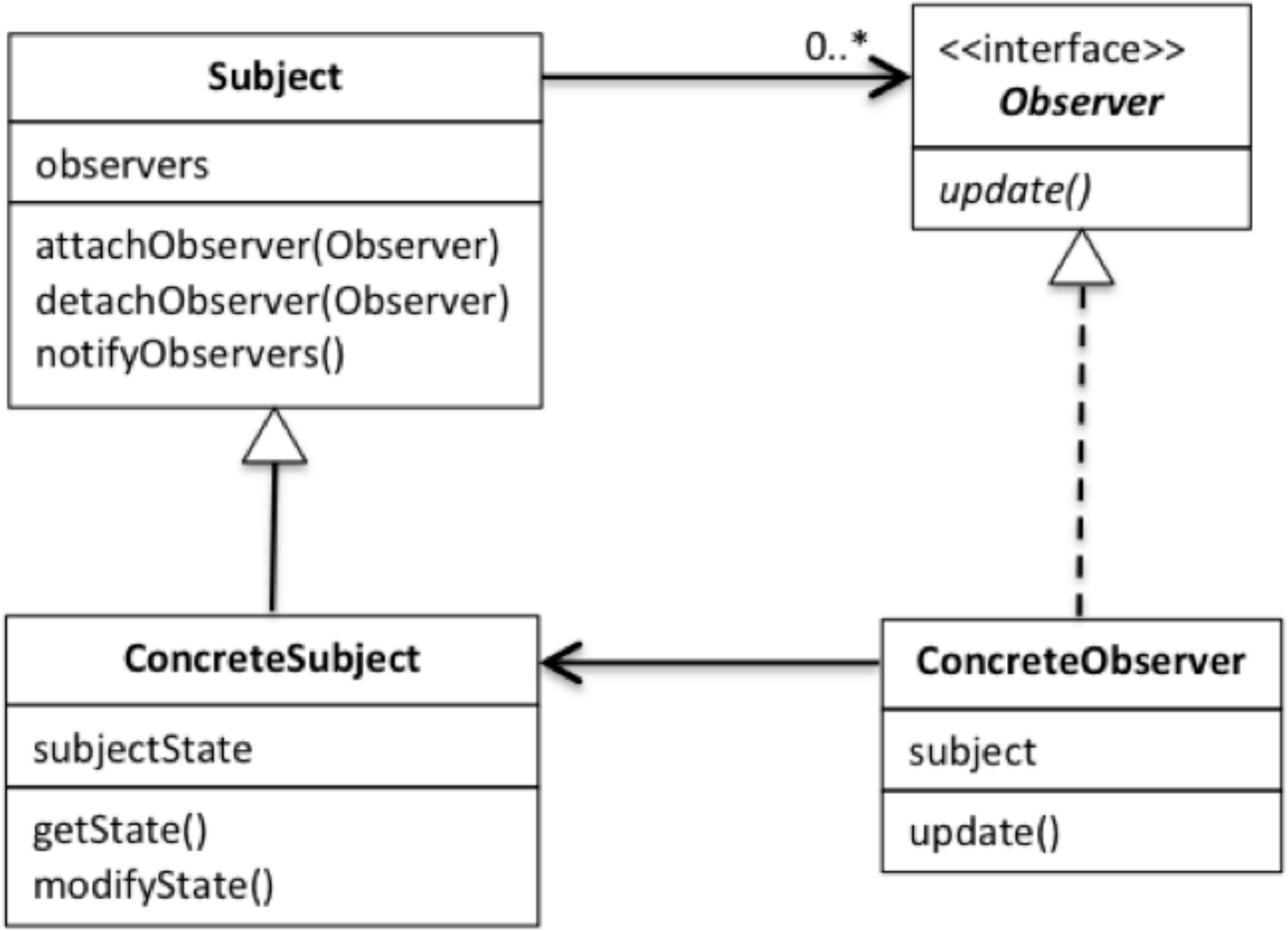
Invoking a Generator with Multiple Listeners

```java
public static void exampleMulti(int n, int k) {
    System.out.println("Multi addable: n, k == " + n + ", " + k);

    final ObservableGenerator g = new ObservableGenerator();

    g.addListener(new Printer());
    final Counter counter = new Counter();
    g.addListener(counter);

    g.generate(n, k);
    final int count = counter.getCount();
    System.out.println("# combinations = " + count);
}
```

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)



Homework Series 5

- Count Kakuro combinations (recursively), with listener
- Simple GUI for Kakuro combinations
- Modify *Simple Kakuro Helper* to use Composite pattern and pulling

Summary

- Processing can be decoupled from a generator via an iterator. But this is not so general. Alternative: Strategy, Template Method
- 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.
- Objects can 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.