Callbacks, Events, and Event-Driven Programs
The limits of scaling

What prevents us from building huge, intricate structures that work perfectly and indefinitely?

– No friction
– No gravity
– No wear-and-tear

... it’s the difficulty of understanding them

So we split designs into sensible parts and reduce interaction among the parts

– More cohesion within parts
– Less coupling across parts
Design exercise

We will extend and modify this example throughout this lecture
  – Provided code shows *skeletal versions that compile*

Our application has various *styled words*
  – A mutable word with a color (and font, size, weight, …)
  – Some styled words are spell-checked against a dictionary
  – Some styled words forbid the letter ‘Q’ [toy example 😊]

Want good coupling, cohesion, and reuse
Available libraries

To set up the example, we assume we have:

1. **StringBuffer** to hold mutable text (in standard library)
   - Methods `insert`, `delete`, and much more

2. A **Dictionary** class with a static method providing dictionaries for available languages
   
   ```java
   class Dictionary {
       public static Dictionary findDictionary(String lang){...}
       public boolean contains(String s){...}
   }
   ...
   ```

3. Classes for all the styling of words
   - Skeletal code just assumes a **Color** class
     - E.g., `new Color("red")`
A direct approach

Version 1 (see v1.java)

Three new classes:

- **StyledWord**
  - Contains a `StringBuffer` and a `Color`

- **SpellCheckedStyledWord**
  - Contains a `StyledWord` and a `Dictionary`

- **NoQsStyledWord**
  - Contains a `StyledWord`
Module dependency diagram (MDD)
What’s wrong with v1?

*Cohesion:* Seems fine – each class has 1 purpose

*Reuse:* So-so

  – Subclassing would avoid all those *forwarding methods*
    • but *SpellCheckedStyledWord / NoQsStyledWord* might not be true subtypes
  – No way to spell-check *and* forbid ‘Q’
    • important if we want *StyledWord* to be a *public library*

*Coupling:* Problematic…
“When the text changes”

class SpellcheckedStyledWord {
  ...
  private void performSpellcheck() {...}
  public void addLetter(char c, int pos) {
    word.addLetter(c, position);
    performSpellcheck();
  }
}

SpellCheckedStyledWord and NoQsStyledWord need to know whenever the text changes
  - addLetter and deleteLetter
  - Hopefully no other ones we forgot!
  - But concept of “text changed” is something we want to leave to StyledWord
  - To avoid this coupling, want the “text changed” event to be managed by StyledWord
Moving “when the text changes”

Version 2 (see v2.java)
  - (Not good but a stepping-stone to version 3)

Let’s make StyledWord responsible for any necessary spell-checking or Q-removal
  - A StyledWord’s state now includes:
    • A Spellchecker if there is one
    • A QRemover if there is one
  - When the word changes, pass this to the spell-checker and/or Q-remover
What is right in v2?

*Reuse*: solves the problems with v1

*Coupling*:
- removes *some* dependence of SpellChecker / Q-Remover on the details of StyledWord
- but on the other hand…
Version 2 MDD

- StyledWord
  - SpellChecker
  - QRemover
What’s wrong with v2?

**Reuse**: A bit better, but work-in progress
- No more forwarding methods
- Can spell-check or Q-remove or both
- But what if there’s a third (or fourth or…) thing we want to do later when some words change

**Cohesion**: Worse: *StyledWord* shouldn’t be directly tracking what needs spell-checking or Q-removal

**Coupling**: Solved our V1 coupling problem, but made our MDD worse
V2 uses callbacks

class StyledWord {
    ...
    private void afterWordChange() {
        if (spellchecker != null)
            spellchecker.performSpellcheck(this);
        if (qremover != null)
            qremover.removeQs(this);
    }

    • performSpellcheck & removeQs passed to the constructor
    • All the StyledWord does with those objects is call performSpellcheck(this) or removeQs(this)
    • performSpellcheck and removeQs are callbacks – code passed in for the purpose of being called some time later
Callbacks

Callback: “Code” provided by client to be used by library
  • In Java, pass an object with the “code” in a method

*Synchronous* callbacks:
  • Examples: `HashMap` calls its client’s `hashCode, equals`
  • Useful when library needs the callback result immediately

*Asynchronous* callbacks:
  • *Register* to indicate interest and where to call back
  • Useful when the callback should be performed later, when some interesting event occurs
  • UIs, servers, etc.
The key decoupling insight

• **StyledWord** depends on **Spellchecker** and **Qremover** in v2, but does *not* need to know *anything* about what these classes do
  – Just needs to call the call-backs when an event occurs (the text changes)

• Weaken the dependency by introducing a much weaker specification in the form of an interface or abstract class
  – The interface implemented by things that can be *notified* when the text changes

```java
interface WordChangeListener {
    public void onWordChange(StyledWord w);
}
```
class StyledWord {
    
    private List<WordChangeListener> listeners;
    public StyledWord(Collection<WordChangeListener> ls) {
        this.listeners = new ArrayList<>(ls);
    }

    public void addLetter(char c, int position) {
        text.insert(position,c);
        afterWordChange();
    }

    private void afterWordChange() {
        for (WordChangeListener listener : listeners)
            listener.onWordChange(this);
    }
}
v3: implement `WordChangeListener`

class Spellchecker implements WordChangeListener {
    ...
    public void onWordChange(StyledWord word) {
        performSpellcheck(word); // as before
    }
}

class QRemover implements WordChangeListener {
    ...
    public void onWordChange(StyledWord word) {
        removeQs(word); // as before
    }
}
A better MDD

- `WordChangeListener` is simple and weak
Judging v3

Cohesion: Good!

Coupling: Good!

Reuse: Good!

- Better than v2: can use any `WordChangeListener` -- no need for to know what they are
  - See `ChangeCounter` in `v3.java`
Achievement unlocked: Observer Pattern

- v3 allows any number of listeners

- Cohesion: **StyledWord** handles styled text while supporting listeners; each listener does its thing

- Coupling: Only via the weakly specified listener interface

This is the **observer pattern**
- Words can be **observed** via **observers/listeners** that are **notified** via **callbacks** when an **event** (of interest) occurs
- **Pattern**: Something used over-and-over in software, worth recognizing when appropriate and using common terms
- Widely used in public libraries
Could be further improved...

- **StyledWord** v3 is reusable enough to be a public library

- But it is not as easy to use as it could be:
  - listeners are only notified that a change has occurred
  - it is up to them to figure out what changed
  - (listener could do this by keeping a copy of the last version for comparison, but that is hard work)

- Easy solution: **StyledWord** should pass a description of what changed to listeners
### Improved `WordChangeEventListener`

```java
interface WordChangeEventListener {
    public void onWordChange(WordChangeEvent e);
}
```

```java
class WordChangeEvent {
    public final StyledWord target;
    public int position; // where it changed
    public String textAdded;
    public String textRemoved;
}
```

Allows even more flexibility for `StyledWord` without any changes needed for listeners (e.g., remove and add text in one operation).
Final version of StyledWord

- Observable with events is **widely** used by important libraries
  - network & file I/O libraries on servers
  - user interface libraries on clients

- In fact, the fundamental structure of these programs is built around processing events & notifying listeners
  - the “main” of these programs is a loop that waits for events and, when they arrive, notifies the appropriate listeners
**Event-driven programming**

An *event-driven* program is designed to wait for events:
- program initializes then enters the *event loop*
- abstractly:
  ```java
do {
    e = getNextEvent();
    process event e;
  } while (e != quit);
```

Contrast with most programs we have written so far
- they perform specified steps in order and then exit
- that style is still used, just not as frequently
  - example: computing Page Rank or other Big Data work
Server Programming

• Servers sit around waiting for events like:
  – new client connections
  – new data from the client (large scale servers)

• Simple version (normal scale):

```java
while (true) {
    wait for a client to connect
    process the request; send a response back
}
```

  – (might want to use a new thread for processing)
  – web servers usually look like this (easiest solution)
Advanced Server Programming

• Large scale servers usually do not have one thread per client
  – it would be hard to scale that past hundreds of clients
  – (need a more complex solution to scale)

• Instead, they have a small number (1?) of threads that simultaneously wait on events from all sockets
  – new connections on the server socket
  – new data to read on any client socket
  – finish writing to any client socket
    • (can then write more)
    – handlers do not make any calls that might wait for something

• These servers look much more like GUI clients…
GUI Client Programming

• Clients sit around waiting for events like:
  – mouse move/drag/click, button press, button release
  – keyboard: key press or release, sometimes with modifiers like shift/control/alt/etc.
  – finger tap or drag on a touchscreen
  – window resize/minimize/restore/close
  – timer interrupt (including animations)
  – network activity or file I/O (start, done, error)
    • (we will see an example of this shortly)
Events in Java AWT/Swing/Android

AWT & Swing are the native Java libraries for writing GUIs
Android apps are also GUIs and written in Java

Most of the GUI widgets can generate events
  – button clicks, menu picks, key press, etc.
Events are handled using the Observer Pattern:
  – objects wishing to handle events register as observers with the objects that generate them
  – when an event happens, appropriate method in each observer is called
  – as expected, multiple observers can watch for and be notified of an event generated by an object

Likewise, advanced servers register handlers on each socket
Event listeners / handlers

**Event listeners** must implement the proper interface. AWT/Swing:

- `KeyListener` – handle key press
- `ActionListener` – handle button press
- `MouseListener` – handle mouse clicks
- `MouseMotionListener` – handle mouse move/drag

When an event occurs
- the appropriate method specified in the interface is called: `actionPerformed`, `keyPressed`, `mouseClicked`, `mouseDragged`, ...
- an event object is passed to the listener method

Interfaces are different in Android but all conceptually the same
Event objects

GUI event is represented by an event object
  – passes information often needed by the handler

In AWT/Swing, the superclass is AWTEvent. Some subclasses are:
  ActionEvent – GUI-button press
  KeyEvent – keyboard
  MouseEvent – mouse move/drag/click/button

In Android, the superclass is InputEvent.

Event objects contain
  – UI object that triggered the event
  – other information depending on event. Examples:
    ActionEvent – text string from a button
    MouseEvent – mouse coordinates
Example: button

Create a JButton and add it to a window
  – (we will talk about windows next time)

Create an object that implements ActionListener
  – contains an actionPerformed method

Add the listener object to the button’s listeners
  – then it will be called when the button is pressed

ButtonDemo1.java
Listener classes

ButtonDemo1.java defines a class that is used only once to create a listener for a single button.

Not ideal in a couple of respects:
- listener code is far away from where it’s used
  - that makes it a little harder to understand
- it’s a lot of code for just one listener
  - imagine doing this in a UI with thousands of components

A more convenient shortcut: lambdas
- in Java 8+, you can use lambdas to create anonymous methods instead of creating a class that only exists to house one method.
Example: button

ButtonDemo2.java
Android similarities

- Events and listeners work in the same manner
- Here is code that listens for a button click:

```java
Button btn = ...;
btn.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Log.d("My Button", "You pressed it");
    }
});
```

- Many of the same widgets as in AWT/Swing
UI Thread

• Where is the event loop in these Swing programs?

• The library creates a separate thread that runs that event loop
  – the “UI thread”
  – created when the JFrame is made visible
  – application does not exit until this thread also finishes
    • that happens automatically when the window is closed
CHAT APP + SERVER
Sockets

- Each client connection is represented by a “socket”

- A socket is like a file
  - can be read from and written to
  - (in Unix, sockets and files are nearly identical)

- Client and server each have “half” of the socket
  - what the client writes is read by the server
  - what the server writes is read by the client
Example: Chat Server

ChatServer.java

(warning: some unfamiliar APIs… )
Example: GUI + sockets

Most modern client applications have to both
– display a GUI
– communicate with one or more servers
– (doing both creates additional difficulties…)

We can make an example by writing a GUI chat client

ChatClientGUI.java
UI thread

• The event loop of a GUI program is run on the “UI thread”

• Often have need of additional threads
  – example: chat UI needed one to listen for new messages
  – any work that may take > 200ms should be done elsewhere

• Warning: most UI frameworks are not multi-thread safe
  – this will not be an issue in this class but will be IRL
  – very few UI API methods can be called from other threads
  – instead, they provide ways to push work onto the UI thread
    • pass a callback to be called from the UI thread
    • then perform the UI changes you need there