CSE 331
Software Design & Implementation

Hal Perkins

Winter 2023

Callbacks, Events and Listeners/Observers

(original slides by Dan Grossman)

Administrivia

- Final goal for our project is to add a graphical user interface (GUI) to our hw7 map…
- …which means (these days) making a web app
- So need to learn just enough JS/TS/React for this
  - Done: last Fri. JS/TS overview; weekend TS video; this week React overview and examples
  - Added to course web resources page: React tips
- HW8 – React map warmup – due next Thur.
- Today: a broader view of callbacks, events, and listeners – key design idea in interactive apps
- Next week: HW9: connect the HW8 React GUI to HW7 pathfinder data

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

- Six versions, each making a point ☺
- Provided code shows *skeletal versions that compile*
- Slides won’t make sense without the code and vice versa!!

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. A **`Dictionary`** class with a static method providing dictionaries for available languages

```
class Dictionary {
  public static Dictionary findDictionary(String lang){…}
  public boolean contains(String s){…}
  …
}
```

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

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`**)

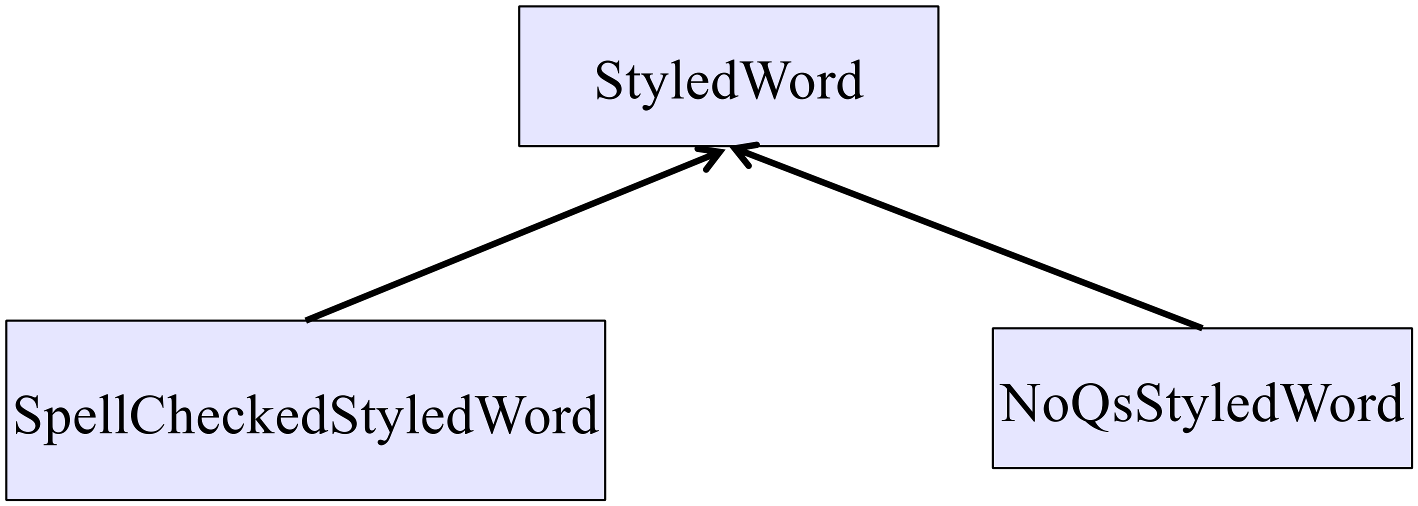
Three new classes:

- **`StyledWord`**
  - Contains a **`StringBuffer`** and a **`Color`**
- **`SpellCheckedStyledWord`**
  - Contains a **`StyledWord`** and a **`Dictionary`**
- **`NoQsStyledWord`**
  - Contains a **`StyledWord`**

Module dependency diagram (MDD)

An arrow in a module dependency diagram (MDD) indicates "depends on" or "knows about"

- Simplistically: "any name mentioned in the source code"
- *Not* just fields, though we emphasize that here



What's wrong with v1?

Cohesion: Seems fine – each class has 1 purpose

Reuse: So-so

- Subclassing would avoid all those *forwarding methods*
- But is **`SpellCheckedStyledWord`** or **`NoQsStyledWord`** a true subtype of **`StyledWord`** ?
  - Depends on spec of **`StyledWord`** (likely not)
- Another reuse issue we will return to: No way to spell-check *and* forbid 'Q'

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`**)

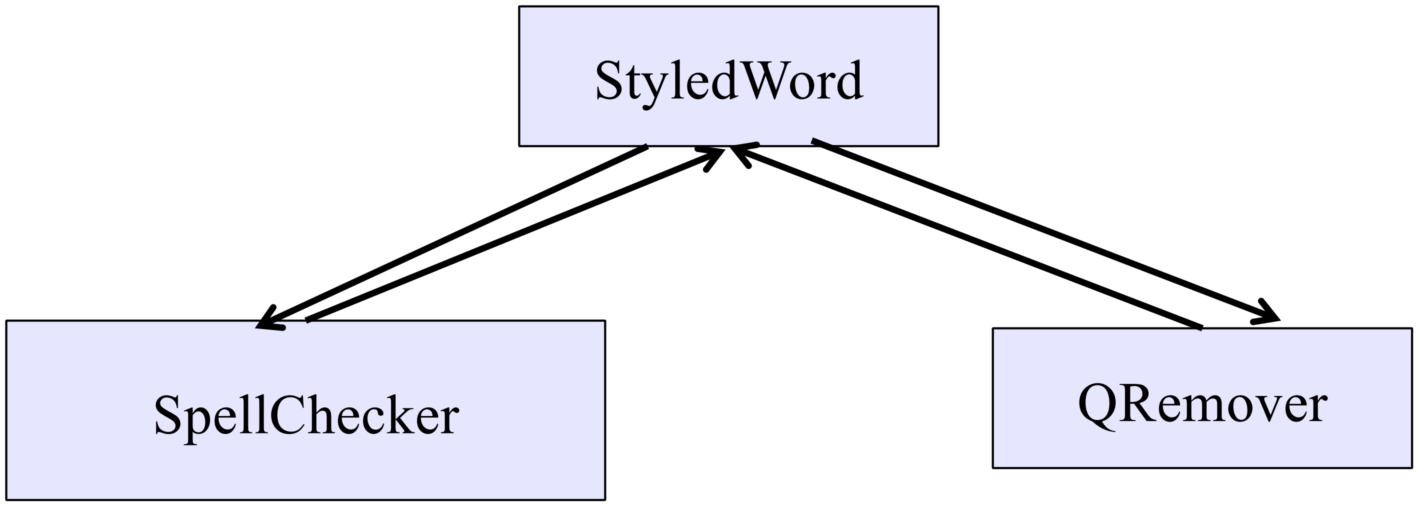
- (Not good but a stepping-stone to version 3)

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

Version 2 MDD

Hmm, more dependencies, but less coupling via the dependencies we had…



V2 uses callbacks

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

- Why do we pass a **`Spellchecker`** or **`Qremover`** to the **`StyledWord`** 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:

- Examples: v2-6; GUI listeners (upcoming homework)
- *Register* to indicate interest and where to call back
- Useful when the callback should be performed later, when some interesting event occurs

What's wrong with v2?

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

Reuse: 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

Coupling: Solved our V1 coupling problem, but made our MDD worse

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

```
interface WordChangeListener {
    public void onWordChange(StyledWord w);
}
```

v3: take a **`WordChangeListener`**

```
class StyledWord {
    private StringBuffer text  = new StringBuffer();
    private Color        color = new Color("black");
    private WordChangeListener listener;
    public StyledWord(WordChangeListener l) {
        listener = l;
    }
    private void afterWordChange() {
        listener.onWordChange(this);
    }
    public void addLetter(char c, int position) {
        text.insert(position,c);
        afterWordChange();
    }
```

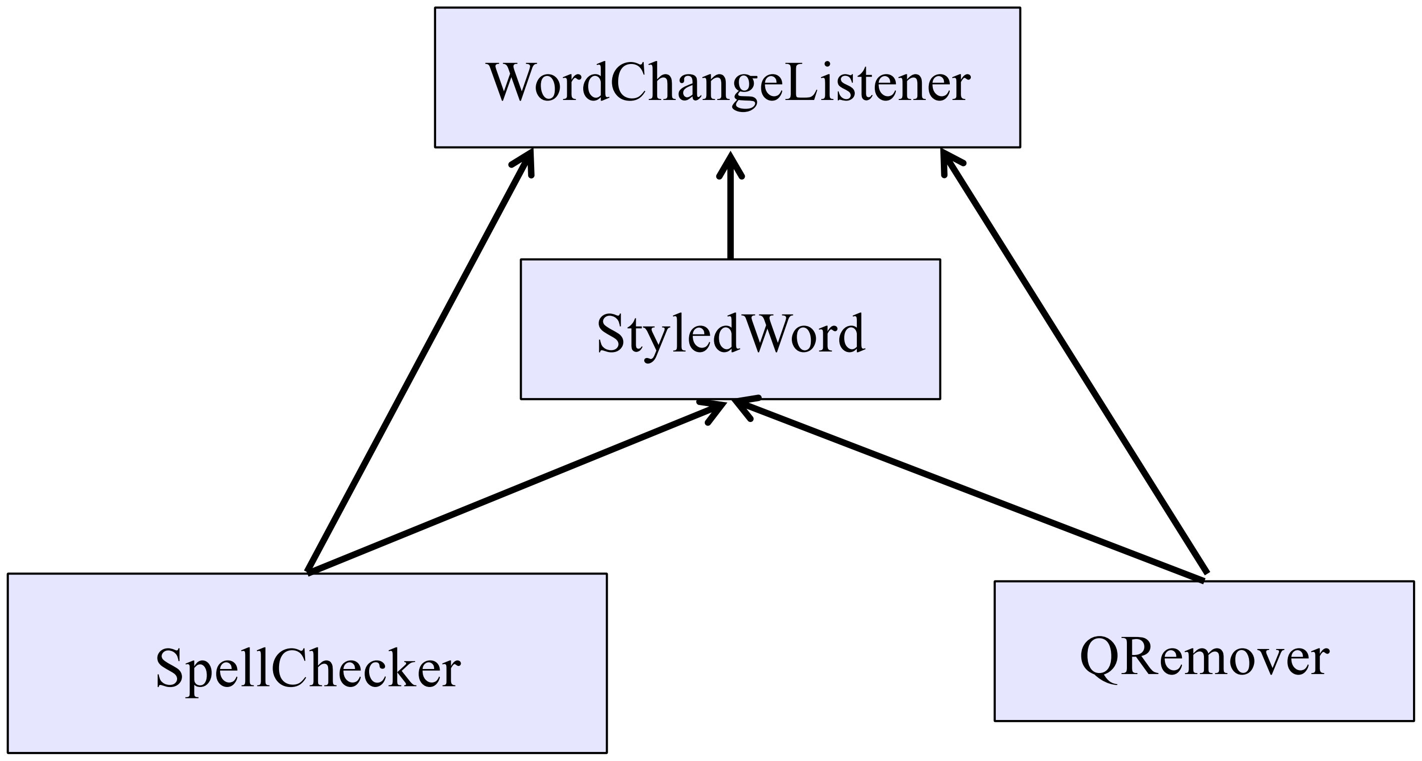
v3: implement `WordChangeListener`

```
class Spellchecker implements WordChangeListener {
    …
    public void onWordChange(StyledWord word) {
        performSpellcheck(word);
    }
}

class QRemover implements WordChangeListener {
    …
    public void onWordChange(StyledWord word) {
        removeQs(word);
    }
}
```

A better MDD

- **`WordChangeListener`** is simple and weak



Judging v3

Cohesion: Good!

Coupling: Good!

Reuse: Better!

- Better than v2: Can use *any* **`WordChangeListener --`** no need for to know what they are
  - See **`ChangeCounter`** in **`v3.java`**
- Worse than v2: Back to allowing only one listener/callback for any particular **`StyledWord`**
  - Hence v4, an “easy fix”

v4: allow multiple listeners

```
class StyledWord {
    …
    private List<WordChangeListener> listeners =
        new ArrayList<WordChangeListener>();
    public StyledWord() { }
    public StyledWord(WordChangeListener l) {
        listeners.add(l);
    }
    public StyledWord(Collection<? extends
                                 WordChangeListener> c) {
        listeners.addAll(c);
    }
    private void afterWordChange() {
        for(WordChangeListener listener : listeners) {
            listener.onWordChange(this);
        }
    }
```

Achievement unlocked: Observer Pattern

- v4 has all the advantages of v3 and 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

v5: dynamic addition/deletion

- No good reason for `StyledWord` to require the listeners to be fixed at object-creation time
  - It “doesn’t care” what the listeners are; just responsible for notifying them when the text changes

- Clients may wish to add and/or remove listeners
  - Example: Change language for spell-checking
  - Example: Start counting changes at some point

- Version 5 does this and is the common approach
  - Mutator methods that add/remove listeners
  - More flexible for clients; up to them to use it wisely

v5: final version of **`StyledWord`**

```
class StyledWord {
    …
    private List<WordChangeListener> listeners =
         new ArrayList<WordChangeListener>();

    public StyledWord() { }
    public void addListener(WordChangeListener l) {
        listeners.add(l);
    }
    public void removeListener(WordChangeListener l) {
        listeners.remove(l);
    }
    private void afterWordChange() {
        for(WordChangeListener listener : listeners) {
            listener.onWordChange(this);
        }
    }
```

A meta-lesson

- We could have just showed you v5 and told you to parrot it and recognize it in industry
- A powerful idiom refined by decades of wisdom, unlikely to be reinvented this well by a relative novice
- But better to *appreciate its good design* in contrast to earlier versions
  - And start to develop the ability to judge a design and identify approaches to improve it
  - And don't be afraid to redesign

Bonus version: v6

- Actually, v1-v5 all contain another “classic” design weakness:
  - *Don’t mix appearance and content*

- This method has poor cohesion, by “hard-wiring” specific colors – or even that coloring is the output – into the actual spell-check method:

```
public void performSpellcheck(StyledWord word) {
  if(dictionary.contains(word.getText()))
     word.setColor(new Color("black"));
  else
     word.setColor(new Color("red"));
}
```

v6 improves this

- Make the spell-checker parameterized over a color-choice
  - Even better would be an arbitrary text-restyling

- Separate “does it spell-check” from “what to do if it does/doesn’t”

- Both lead to better cohesion

- See the code
  - Not directly related to callbacks/events/listeners
  - But helps show why graphical applications tend to have lots of parameters and levels of abstraction

A note on React

- In React an observer is a function that we want to have called when an event happens

- The observer function is passed down to the component that can generate the event as an element of the generating component’s props
  - This is React’s version of *registering* a listener

- When an event happens, the generating component calls the function that was part of its props
  - This is the callback