Announcements

- We’ll code review Stars on Thursday.
- Volunteer your code by emailing me.
- Lab this week covers Ajax/Javascript. Interactive UIs.
- No lab (or lab hours) next week.
- Submit a group project idea on Piazza by Friday.
- Autocorrect Gear-up 6:30pm today here. (Metcalf)

Science Center’s Learning Exchange program is looking for student facilitators to teach this semester! LE helps middle school students develop their coding and math skills by using Scratch and Javascript to create their own games. As a facilitator, you’ll work one-on-one with students, guiding them in completing their projects and introducing them to coding. It’s a low-time commitment and a great way to make a difference off College Hill. If interested, please fill out this form: https://goo.gl/forms/eHLyu30SyRPfLAB53.
Design Patterns

John Jannotti

/course/cs0320/www/lectures/

Feb 13, 2018
Project Ideas

- League / tournament scheduler (round-robin, double elimination, matched strength)
- Anti-clickbait (or fake news) as a service and a browser plugin to use it.
- Analyze, find anomalies in large document dumps.
- Wait-in-line app (need to make it interesting — texting, history)
- “Good” Go, Chess, Scrabble, Crossword, Poker, Bridge players
- Real-time Strategy Game (Ikariam, Call of War)
- Desktop Tower Defense (Mix up a classic.)
- Teleconferencing, “burner numbers”, other voice app (w/ Twilio)
(Bad) Project Ideas

- Anything with network effects. (Not enough time.)
- Anything that requires specific data (that you don’t have).
- “Real” startup ideas (If simplistic implementation)
- Try to look outside the market of “Any Brown Student”. Find a niche.
- Don’t just build a C.R.U.D app!
Designers (of all kinds) work by recycling ideas.

- Experience provides more & better ideas.
- Knowing what ideas will work when.
- And how to combine them into something “new”

Patterns are a way of naming these ideas.

- The basic ideas themselves.
- The information for how & when to use them.
Patterns at multiple levels

- **Code patterns**
  - for-loop over array indices
  - Test a condition and acquire a lock atomically.

- **Data structure patterns**
  - Interface X, AbstractX class.
  - Provide iterator as inner class, Itr

- **Design patterns**
  - Organizing classes, methods, interfaces.

- **Architectural patterns**
  - Plugins, Thread-per-request, Client-Server, Peer-to-peer.
What is a Design Pattern?

- Problem to be addressed
  - What the pattern is trying to do
  - Motivation for using the pattern
  - Exactly what the pattern does for you

- Conditions
  - When the pattern can be applied
  - Strengths & Weaknesses of the pattern

- Implementation
  - What are the classes and methods?
  - How are the classes related?
  - How are the methods implemented?
Caveats

- These are solutions looking for problems.
- Understand & analyze the problem first.
  - *Then* consider suitable patterns.
  - The pattern’s problem description is vital.
- Experience will (eventually) tell what works.
- Beware of added/excess complexity.

A simple rule of thumb: Introduce a pattern (or, any complexity) when your code needs it, not in anticipation of needing it. YAGNI. But don’t use that as an excuse. Often, general code is *easier and clearer!*. Think about a SIZE constant, or n-dimensional vs 3-dimensions.
Standard Patterns

- Design Patterns, Gamma et al.
  - And lots of other books
  - Anti-patterns (code “smells”)
- Patterns can be classified by use
  - Factory patterns
  - Delegation patterns
  - Structural patterns
  - Control patterns
Factory Patterns

• How to create objects
  ▶ Without using new directly.
  ▶ Why?

• Various forms
  ▶ Factory class - Builders, hides multiple implementation classes
  ▶ Factory interface - callers provide factory to library
  ▶ Factory method - good for Fly-weight (below)

• Interesting variants
  ▶ Prototype: copy a sample object (But Java’s clone() has problems.)
  ▶ Singleton: allow only one instantiation (Some recommend an enum.)
  ▶ Fly-weight: shared use of immutable instances (maybe copy-on-write)
Sidebar: The need for Multimaps

You may have code that does this:

```java
Map<Clerk, List<Sale>> ledger = new HashMap<>();

public void makeSale(Clerk clerk, Sale sale) {
    List<Sale> sales = ledger.get(clerk);
    if (sales == null) {
        sales = new ArrayList<Sale>();
        ledger.put(clerk, sales);
    }
    sales.add(sale);
}
```
With Multimaps, you can say

```java
1 Multimap<Clerk, Sale> ledger =
2     new ArrayListMultimap<>();
3
4 public void makeSale(Clerk clerk, Sale sale) {
5     ledger.put(clerk, sale);
6 }
```

There’s Multiset, too. Boy, that’s handy when I need to count how often something (words, bigrams?) appears in a dataset (corpus?).
Factory use in Guava’s Multimaps

- Clients can choose iteration order / overhead.
  - ArrayListMultimap, HashMultimap, LinkedHashMultimap...
  - Clients can use `new` or Multimaps.newXXX()
  - Factory methods here are mostly historic, aesthetic.

- Immutable versions for safety
  - ImmutableListMultimap, ImmutableSetMultimap
  - Builder pattern is required. (two slides forward)

- Some are not even public classes
  - Multimaps.newListMultimap()
  - Multimaps.newSortedSetMultimap()
  - Factory methods hide details.
  - And these Factory methods take Factory objects!
Builder Pattern

Avoids confusing construction for “many fielded” objects.

```java
1 gc = new GraphicsContext(1, 30, null,
2 Color.RED, Color.Black,
3 . . . );
```

vs

```java
1 gc = new GraphicsContext.Builder()
2 .lineThickness(1)
3 .opacity(30)
4 .foreground(Color.RED)
5 .background(Color.BLACK)
6 .build();
```
Builder Pattern

Avoids confusing construction for “many fielded” objects.

```java
gc = new GraphicsContext(1, 30, null,
Color.RED, Color.Black,
...);
```

vs

```java
gc = new GraphicsContext.Builder()
  .lineThickness(1)
  .opacity(30)
  .foreground(Color.RED)
  .background(Color.BLACK)
  .build();
```

Make Builder a static inner class that calls the outer class’s private constructor.
Why must ImmutableMap offer a Builder class?

```java
Map<String, Integer> population =
    new ImmutableMap.Builder<String, Integer>()
    .put("AL", 4849377)
    .put("AK", 737732)
    ...
    .put("WY", 584153)
    .build();
```

You can't build an immutable map up bit by bit! (And you can't make a type safe varargs constructor that alternates types.)
Builder Pattern for Collections

Why must ImmutableMap offer a Builder class?

```java
Map<String, Integer> population =
    new ImmutableMap.Builder<String, Integer>()
    .put("AL", 4_849_377)
    .put("AK", 737_732)
    ...  
    .put("WY", 584_153)
    .build();
```

You can’t build an immutable map up bit by bit! (And you can’t make a type safe varargs constructor that alternates types.)
Delegation Patterns

- Separate implementation and interface
  - Multiple possible implementations
  - Might want to reuse existing implementation
  - Implementation might change later, be remote, be complex...

- Solutions
  - *Proxy* — real object is remote, protected, created on demand...
    - Early hint: This is great for objects in databases.
  - Object might be complex: *Facade*
  - Translate an interface: *Adapter*
  - Adding dynamic functionality: *Decorator*
ForwardingMultimap

See ForwardingMultimap.java. It forwards every single method to a designated Multimap.

```java
@Override
public void clear() {
    delegate().clear();
}

@Override
public Multiset<K> keys() {
    return delegate().keys();
}
```
Decorator allows more flexibility than hierarchy

```
class LoggingMultimap<K,V> extends ForwardingMultimap<K,V> {
    private Multimap<K,V> inner_;  
    LoggingMultimap(Multimap<K,V> mmap) {
        inner_ = mmap;
    }
    @Override public void delegate() { return inner_; }
    @Override public void clear() {
        super.clear();
        System.out.println("Cleared a multimap");
    }
}
```

You can replace a Multimap mm with new LoggingMultimap(mm) and carry on. And it works for any Multimap. (Guava provides at least 10 different implementations!)
Other Patterns

- **Structural patterns**
  - Composite — Group several of something into one of those things.
  - Command — Great for *undo*.

- **Control patterns**
  - Iterator — expose elements, but not container.
  - Strategy — embed algorithm in a class.
  - Template — algorithm with virtual hooks, a skeleton.
  - Visitor — apply operations to an object graph.

- **Algorithmic patterns**
  - Observer — publish-subscribe.
  - Momento — save/restore state.
Abstract public class Area {
    abstract public boolean contains(LatLong x);
}

public class JointArea extends Area {
    private List<Area> areas = new ArrayList<>();
    public JointArea(Area... as) {
        for (Area a : as)
            areas.add(a);
    }
    public boolean contains(LatLong x) {
        for (Area area : areas)
            if (area.contains(x))
                return true;
        return false;
    }
}
Strategy in Autocorrect

- Your program will need to
  - Create suggestions using a few different techniques.
  - Rank those suggestions a couple different ways.
- We suggest using the Strategy pattern.
  - Generator
  - Ranker
Correcter.java

```java
public class Correcter {
    Correcter(Generator g, Ranker r) {
    }
    String suggest(String previous, String last) {
        List<String> suggestions = g...(previous, last);
        Collections.sort(suggestions, r...);
    }
}
```

- Generator and Ranker are interfaces for Strategies
  - abstract classes if you have a good reason
- How can you turn on LED and word splitting at the same time?
- How will you make the Comparator?
- Maybe the Corpus belongs in the Ranker?
- Maybe you need to return a Suggestion/Bigram instead of a String?
- Maybe there’s an existing interface Ranker can implement?
Using Patterns

- When you run into a problem
  - The solution isn’t obvious
  - But the problem feels “common”
  - Might be encoded as a design pattern
  - Awareness of patterns helps you notice alternatives

- For describing a design
  - Experienced programmers have seen the same patterns
  - Stating a pattern simplifies description
    - Don’t get it wrong!