

CS 2110

Software Design Principles I

## Recap from last time

- We were talking about the class hierarchy and inheritance of methods
- Basic idea was to have a parent class that implements some very general functionality and then a child class that specializes it.
- A parent class can also standardize an “interface” shared by child classes. For example: classes that support the same interface as an Array
  - $X[i] = 17 + Y[j,k] / X[i]$ ;      // Is X really an array?

## Array vs ArrayList vs HashMap (latter two from java.util)

- |   |   |
|---|---|
| <ul style="list-style-type: none"> <li>□ Array           <ul style="list-style-type: none"> <li>■ Storage is allocated when array created; cannot change</li> <li>■ Extremely fast lookups</li> </ul> </li> <li>□ ArrayList (in java.util)           <ul style="list-style-type: none"> <li>■ An “extensible” array</li> <li>■ Can append or insert elements, access <math>i</math>th element, reset to 0 length</li> <li>■ Lookup is slower than an array</li> </ul> </li> </ul> | <ul style="list-style-type: none"> <li>□ HashMap (in java.util)           <ul style="list-style-type: none"> <li>■ Save data indexed by keys</li> <li>■ Can look up data by its key</li> <li>■ Can get an iteration of the keys or values</li> <li>■ Storage allocated as needed but works best if you can anticipate need and tell it at creation time.</li> </ul> </li> </ul> |
|---|---|

## HashMap Example

- Create a HashMap of numbers, using the names of the numbers as keys:
 

```
Map<String, Integer> numbers
    = new HashMap<String, Integer>();
numbers.put("one", new Integer(1));
numbers["two"] = new Integer(2);
numbers.put("three", new Integer(3));
```
- To retrieve a number:
 

```
Integer n = numbers.get("two"); // Explicit method call
Integer n = numbers["two"];    // Array notation
```
- Returns null if the HashMap doesn't contain key
  - Can use numbers.containsKey(key) to check this

## Generics and Autoboxing

- Is a number like 71 an Integer (an object) or a base type (an “int”)?
- How do I create an array with an object, not a base type, in the entries?
- Java automatically “autoboxes” and also lets you use types as a kind of parameter

```
Map<String, Integer> numbers = new HashMap<String, Integer>();
numbers.put("one", 1);      // Autobox converts 1 to new Integer(1);
int s = numbers.get("one");
```

## What do these tell us?

- There is a great deal of power in “abstraction”
  - Here we're seeing examples in which the abstract type is an array, but the values and even the index can be arbitrary objects!
- In Java we often also need special-purpose objects that add functionality, properties etc
  - For example, to make an array extensible, or to ensure that lookup will use a very fast method even if the index type isn't an integer

## Our challenge?

- We need to look at a computing problem, such as building software for cyclists, and learn to
  - ▣ See the most general abstractions, where they arise. For example “gosh, these are graphs”
  - ▣ Build powerful, general purpose solutions, such as a graph class supporting graph operations
  - ▣ But then also see how to map that general abstraction back to the real world
    - For example, creating bike routes that have GPS locations and times and other bike-specific properties

## Mapping goes two ways

- You look at a problem and say “I see a more basic, general idea here”
  - ▣ These bike routes look like graphs to me
  - ▣ So I'll build a graph class, and then I'll specialize it to support graphs of bike data
- But sometimes you have an existing powerful class and think the opposite way
  - ▣ I already have a graph package. I'll use it to implement bike routes

## So how do people do this?

- One of the hardest questions in computing centers on finding the right abstractions
- We want them to be powerful, yet efficient
- We want ways to specialize them that seem as natural as possible

## A journey of a thousand miles...

- ... starts with a single step
- Most developers develop code partly by experimentation
- Don't be afraid to experiment by writing little code fragments and seeing if they compile and what they do.
- **But don't write random code hoping that it might work by some miracle.**



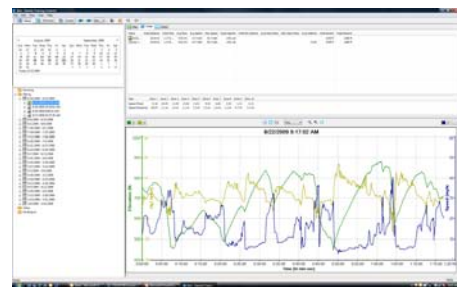
## Mistakes will happen!



- We call them bugs...
- To debug code, we need to think hard...
  - ▣ **Do not just make random changes, hoping something will work. This never works.**
  - ▣ Think about what could cause the observed behavior
  - ▣ Isolate the bug. Focus on the first thing that goes wrong.
- An IDE helps by providing a *Debugging Mode*
  - ▣ Can set breakpoints, step through the program while watching chosen variables
  - ▣ When program pauses at breakpoint, or dies, can look at values of variables it was using

## So let's look at how all this works

- Garmin GPS unit tracks your bike ride

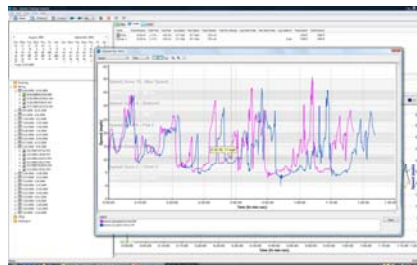


... displays match one graph (of the ride) with another graph (the map)



... comparisons also match two graphs

- Which parts of my ride gained time versus last time? Which lost time?



Actual data is an XML document containing a list of "track points"

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<TrainingCenterDatabase xmlns="http://www.garmin.com/xmlschemas/TrainingCenterDatabase2" xmlns:sec="http://www.garmin.com/xmlschemas/TrainingCenterDatabase2" xmlns:location="http://www.garmin.com/xmlschemas/TrainingCenterDatabase2" />
  <Activity xmlns="http://www.garmin.com/xmlschemas/TrainingCenterDatabase2" />
    <Id>2009-08-22T13:17:02Z</Id>
    <Name>2009-08-22T13:17:02Z</Name>
    <Start Time>2009-08-22T13:17:02Z</Start Time>
    <TotalTimeSeconds>4625.000000</TotalTimeSeconds>
    <DistanceMeters>3213.270000</DistanceMeters>
    <MaximumSpeed>17.700000</MaximumSpeed>
    <Calories>145</Calories>
    <Intensity></Intensity>
    <Cadence></Cadence>
    <EngineMetricName></EngineMetricName>
    <Tracks>
      <Track>
        <Time>2009-08-22T13:17:02Z</Time>
        <LatitudeDegrees>42.5619367</LatitudeDegrees>
        <LongitudeDegrees>-76.4460288</LongitudeDegrees>
        <AltitudeMeters>225.4117432</AltitudeMeters>
        <SpeedMetersPerSecond>3.214454</SpeedMetersPerSecond>
        <SensorStatus></SensorStatus>
      </Track>
      <Track>
        <Time>2009-08-22T13:17:05Z</Time>
        <LatitudeDegrees>42.5619367</LatitudeDegrees>
        <LongitudeDegrees>-76.4460288</LongitudeDegrees>
        <AltitudeMeters>227.4891927</AltitudeMeters>
        <SpeedMetersPerSecond>3.5653191</SpeedMetersPerSecond>
        <SensorStatus></SensorStatus>
      </Track>
    </Tracks>
  </Activity>
</TrainingCenterDatabase>
```

But the data didn't "start" as a graph

- Applications like this often need to get their data from some other format
- For example, the Garmin bike device actually creates a file in which it records the GPS data
- The analysis program runs separately on my PC

Each ride is in a separate file

- Sort of like a set of documents
- I want to find the ones that "describe" the same route – the same list of roads in the same order, turns at the same place, etc
- But the GPS unit won't have collected snapshots at identical spots

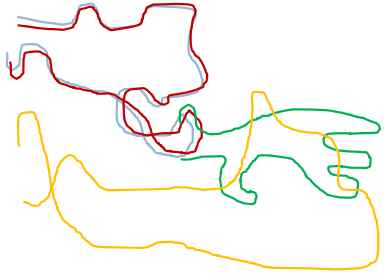
So suppose we want to compare two "rides"



- We've been thinking of each ride as a graph
- If we also consider the GPS data the ride is a curve in 3-D "space" (nodes are GPS data, edges link successive points)
- If two rides were on the same route, then these curves should match closely, provided we ignore the timestamp
  - After all, my rides weren't at identical speeds, which is my reason for wanting to compare them

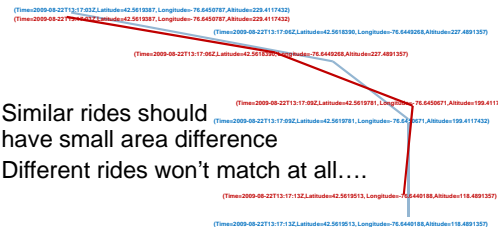


## Which rides were similar?



## Which rides were similar?

- Could match the curves “edge by edge” and compute area between them....



- Similar rides should have small area difference
- Different rides won't match at all...

## What makes it tricky?

- Lance and Pantani didn't follow the identical route (they were on the same road, but obviously didn't exactly follow each other)
- They may have been separated in time here and there, even if at the end of the day they were side by side on the climb
- Sometimes Lance was faster, sometimes Pantani was faster

## The idea of abstraction

- Our goal is to learn to think very abstractly
  - A “ride” that followed some “route”
  - The ride may differ (faster, slower, paused to wait for a car to pass) and yet the “route” is essentially the same
  - Yet even the route won't be identical (depends on how you define identical...)

## Software Engineering



- The art by which we start with a problem statement and gradually evolve a solution
- There are whole books on this topic and most companies try to use a fairly uniform approach that all employees are expected to follow
- The IDE can help by standardizing the steps

## The software design cycle

- Some ways of turning a problem statement into a program that we can debug and run
  - Top-Down, Bottom-Up Design
  - Software Process (briefly)
  - Modularity
  - Information Hiding, Encapsulation
  - Principles of Least Astonishment and “DRY”
  - Refactoring

## Top-Down Design

- Garmin GPS software

- Refine the design at each step
- Decomposition** / "Divide and Conquer"

## Not a perfect, pretty picture

- Boxes at lower levels are "more concrete" and contain things like GPS records, actual strings
- Boxes at higher levels are more abstract and closer to dealing with the user
- In between are "worker bees" that do things like file storage and waking up Google Earth
- But don't take the hierarchy too seriously
  - Most things don't fit perfectly into trees

## Bottom-Up Design

- Just the opposite: start with parts

- Composition**
- Build-It-Yourself (e.g. IKEA furniture)

## Top-Down vs. Bottom-Up

- Is one of these ways better? Not really!
  - It's sometimes good to alternate
  - By coming to a problem from multiple angles you might notice something you had previously overlooked
  - Not the only ways to go about it
- With **Top-Down** it's **harder** to **test early** because parts needed may not have been designed yet
- With **Bottom-Up**, you may end up **needing** things **different** from how you built them

## Software Process

- For simple programs, a simple process...

"Waterfall"

- But to use this process, you need to be sure that the **requirements are fixed** and **well understood!**
  - Many software problems are not like that
  - Often customer refines the requirements when you try to deliver the initial solution!

## Incremental & Iterative

- Deliver **versions of the system** in several **small cycles**

- Recognizes that for some settings, software development is like gardening
- You plant seeds... see what does well... then replace the plants that did poorly

## Modularity

- Module: component of a system with a well-defined interface. Examples:
  - Tires in a car (standard size, many vendors)
  - Cable adaptor for TV (standard input/output)
  - External storage for computer
  - ...
- Often includes more than one class
- Modules “hide information” behind their interfaces

## A module isn't just an object

- *We're using the term to capture what could be one object, but will often be a larger component constructed using many objects*
- In fact Java has a module subsystem for this reason (we won't use it in cs2110)
  - A module implements some “abstraction”
  - You think of the whole module as a kind of big object

## Information Hiding

- What “information” do modules hide? “Internal” design decisions.

```
class Set {
    ...
    public void add(Object o) ...
    public boolean contains(Object o) ...
    public int size() ...
}
```

- A class's **interface** is everything in it that is **externally accessible**

## Encapsulation

- By hiding code and data behind its interface, a class encapsulates its “inner workings”
- Why is that good?
  - Lets us change the implementation later without invalidating the code that uses the class

```
class LineSegment {
    private Point2D _p1, _p2;
    ...
    public double length() {
        return _p1.distance(_p2);
    }
}
```

```
class LineSegment {
    private Point2D _p;
    private double _length;
    private double _phi;
    ...
    public double length() {
        return _length;
    }
}
```

## Encapsulation

- Why is that good? *(continued)*
  - Sometimes, we want a few different classes to implement some shared functionality
  - For example, recall the “iterator” construct we saw in connection with collections:

```
Iterator it = collection.iterator();
while (it.hasNext()) {
    Object next = it.next();
    doSomething(next);
}
for (String s: args) {
    System.out.println("Argument "+s);
}
```

- To support iteration, a class simply needs to implement the Iterable interface

## Degenerate Interfaces

- Public fields are usually a **Bad Thing**:

```
class Set {
    public int _count = 0;
    public void add(Object o) ...
    public boolean contains(Object o) ...
    public int size() ...
}
```

- Anybody can change them; the class has no control

## Interfaces vs. Implementations

- This says “I need this specific implementation”:

```
public void doSomething(LinkedList items) ...
```

- This says “I can operate on anything that supports the Iterable interface”

```
public void doSomething(Iterable items) ...
```

- Interfaces represent higher levels of abstraction (they focus on “what” and leave out the “how”)

## Use of interfaces?

- When a team builds a solution, interfaces can be very valuable!
  - ▣ Rebecca agrees to implement the code to extract GPS data from files
  - ▣ Tom will implement the logic to compare bike routes
  - ▣ Willy is responsible for the GUI
- By agreeing on the interfaces between their respective modules, they can all work on the program simultaneously

## Principle of Least Astonishment

- A interface should “hint” at its behavior

```
Bad:
public int product(int a, int b) {
    return a*b > 0 ? a*b : -a*b;
}
```

```
Better:
public int absProduct(int a, int b) {
    return a*b > 0 ? a*b : -a*b;
}
```

- Names and comments matter!

## Principle of Least Astonishment

- Unexpected **side effects** are a Bad Thing

```
class Integer {
    private int _value;
    ...
    public Integer times(int factor) {
        _value *= factor;
        return new Integer(_value);
    }
}
Integer i = new Integer(1);
Integer j = i.times(10);
```

Developer was trying to be clever. But what does this code do to i?

## Duplication

- It is very common to find some chunk of working code, make a replica, and then edit the replica
- But this makes your software fragile: later, when the code you copied needs to be revised, either
  - The person doing that changes all instances, or
  - some become inconsistent
- Duplication can arise in many ways:
  - constants (repeated “magic numbers”)
  - code vs. comment
  - within an object’s state
  - ...

## “DRY” Principle

- Don't Repeat Yourself
- A nice goal is to to have each piece of knowledge live in one place
- But don't go crazy over it
  - ▣ DRYing up at any cost can increase dependencies between code
  - ▣ “3 strikes and you refactor” (i.e., clean up)

