CS 2110

Based on a slide set by Juan Altmayer Pizzorno
port25.com

Software Design Principles I

Overview

- Today: Design Concepts & Principles
  - Top-Down, Bottom-Up Design
  - Software Process (briefly)
  - Modularity
  - Information Hiding, Encapsulation
  - Principles of Least Astonishment and "DRY"
  - Refactoring (if there's time)
- Next week: Test-Driven Development

Our Challenge

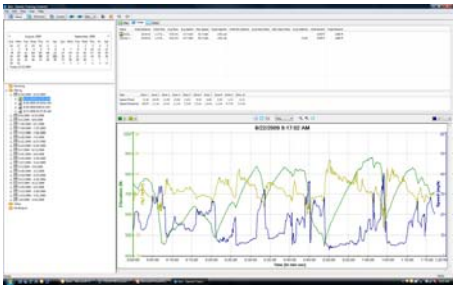
- For simple applications, writing code is "linear"
  - You pin down the problem
    - Example: "search files and list lines that contain the string SnortBlat"
  - You make minor decisions, such as where the list of files will come from, and whether SnortBlat will be a constant or an input to the program
  - And then you write the code
- Easy as pie!

But it isn't always so easy!

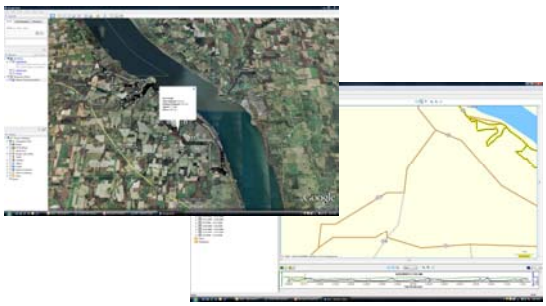
- Interesting applications are often challenging in ways that simple ones aren't
  - Data sets may be enormous
  - The thing being computed may be complex
  - The amount of code required to do it in the most obvious way may seem huge (and perhaps also, repetitious)

A more complex example

- Garmin GPS unit tracks your bike ride

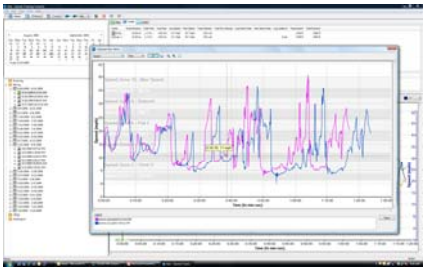


… making display easy



… or comparisons

- How did I do today compared to the last time I rode this same route?



… or comparisons

- Suppose we wanted to automate “finding previous rides on the same route”
  - A rides is a long list of location points and won’t be identical each time
  - How could we search a list of “rides” to see which ones were rides on the same route?
- Is this problem similar to search files for the word SnortBlat, or different?

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


```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<TrainingCenterDatabase xmlns="http://www.garmin.com/xmlschemas/TrainingCenterDatabase/v2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.garmin.com/xmlschemas/TrainingCenterDatabase/v2 http://www.garmin.com/xmlschemas/TrainingCenterDatabasev2.xsd">

 <Activities>
  <Activity Sport="Biking">
   <Id>2009-08-22T13:17:02Z</Id>
   <Lap StartTime="2009-08-22T13:17:02Z">
    <TotalTimeSeconds>4625.0800000</TotalTimeSeconds>
    <DistanceMeters>30319.2753906</DistanceMeters>
    <MaximumSpeed>17.7600002</MaximumSpeed>
    <Calories>1451</Calories>
    <Intensity>Active</Intensity>
    <Cadence>0</Cadence>
    <TriggerMethod>Manual</TriggerMethod>
    <Track>
     <Trackpoint>
      <Time>2009-08-22T13:17:03Z</Time>
      <Position>
       <LatitudeDegrees>42.5619387</LatitudeDegrees>
       <LongitudeDegrees>-76.6450787</LongitudeDegrees>
      </Position>
      <AltitudeMeters>229.4117432</AltitudeMeters>
      <DistanceMeters>9.2514459</DistanceMeters>
      <SensorState>Absent</SensorState>
     </Trackpoint>
     <Trackpoint>
      <Time>2009-08-22T13:17:06Z</Time>
      <Position>
       <LatitudeDegrees>42.5618390</LatitudeDegrees>
       <LongitudeDegrees>-76.6449268</LongitudeDegrees>
      </Position>
      <AltitudeMeters>227.4891357</AltitudeMeters>
      <DistanceMeters>26.0653191</DistanceMeters>
      <SensorState>Absent</SensorState>
     </Trackpoint>
     ...
    </Track>
    ...
```


(Time=2009-08-22T13:17:03Z,Latitude=42.5619387, Longitude=-76.6450787,Altitude=229.4117432)

(Time=2009-08-22T13:17:06Z,Latitude=42.5618390, Longitude=-76.6449268,Altitude=227.4891357)

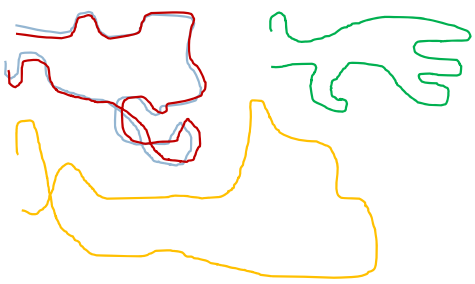
Each ride is in a separate file

- 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

An idea!

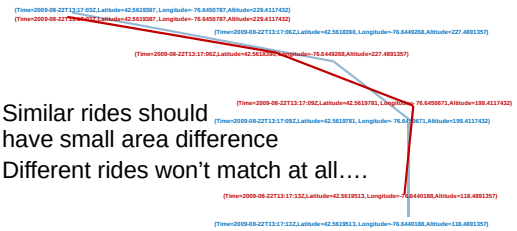
- Think of each ride as a curve that we represent as a graph (nodes are GPS data, edges link successive points)
- If two rides were on the same route, then these curves should overlap closely, provided we *ignore the timestamp*
  - (This is because the rides were different days and perhaps different speeds)

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….

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 c… the sa…
  - Yet ev… how you define identical…)
- Finding the "same route" is like "searching files for some term" yet all the details are different!

This gets at the idea of abstraction. In some sense we can use the idea of searching files to think about the search for rides on the same route!

Software Engineering

- Actually, more like an "art"!
  - Elegantly expresses the necessary logic
  - Built with minimum effort… "obviously" correct…
  - Self-testing
  - Flexible and extensible: can be understood and maintained by someone years after you retire
- The go-it-alone style is fine too, but even if a program is for your own use, quality matters

Applied to bike riding

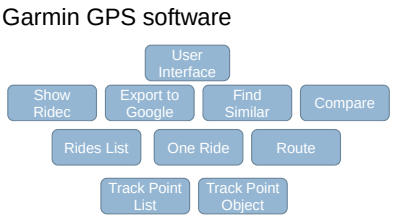
- We have an idea for how we might find similar bike rides in a set of bike rides
- But to turn this into software we need to decide
  - What Java classes to create and what methods to support in each
  - We want this to "fit" with our mental image of how the comparison algorithm needs to work
  - There are many ways to do this, but some might be awkward to implement or might be confusing to actually work with…

Software Engineering: History

- In early days, wasn't recognized as a real need
- Then studies revealed that the best computer scientists were often 10x or even 100x more productive than the average code hacker!
  - Moreover, some languages seemed to encourage better code quality
  - Java was one of them…
- Software Engineering emerged from the effort to reduce this to a kind of science

Top-Down Design

- Garmin GPS software



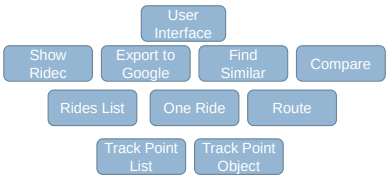
- Refine the design at each step
- Decomposition / "Divide and Conquer"

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

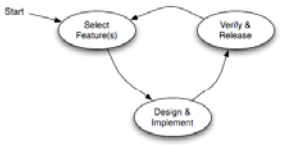
- For simple programs, a simple process…

“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
  - ...
- 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")

Principle of Least Astonishment

- Have your designs work how a user would expect

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
  - ...

Duplication in Comments

```
public double totalArea() {
  ...
  // now add the circle
  area += PI * pow(radius,2);
  ...
}
```

```
public double totalArea() {
  ...
  area += circleArea(radius);
  ...
}

private double circleArea(double radius) {
  return PI * pow(radius, 2);
}
```

- Some people use comments when they should be creating a new method to do the thing the comment "says" you are doing!
  - If the thing you are doing captures some idea (here, it involves the formula for the area of a circle), create a method and use that
  - The code itself becomes "self documenting"

Duplication of State

- Duplication of state can lead to inconsistency within an object:

```
class LineSegment {
  private Point2D _p1, _p2;
  private double _length;

  ...
  public double length() {
    return _length;
  }
}
```

```
class LineSegment {
  private Point2D _p1, _p2;

  ...
  public double length() {
    return _p1.distance(_p2);
  }
}
```

- Can you see how this code could evolve later and accidentally become inconsistent?
- Duplication may be desirable for performance, but don't optimize too soon

"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)

Refactoring

- Refactor: to improve code's internal structure without changing its external behavior
- Most of the time we're modifying existing software
- "Improving the design after it has been written"
- Refactoring steps can be very simple:

```
public double weight(double mass) {
  return mass * 9.80665;
}
```

```
static final double GRAVITY = 9.80665;

public double weight(double mass) {
  return mass * GRAVITY;
}
```

- Other examples: renaming variables, methods, classes

Why is refactoring good?

- If your application later gets used as part of a Nasa mission to Mars, it won't make mistakes
- Every place that the gravitational constant shows up in your program a reader will realize that this is what she is looking at
- The compiler may actually produce better code

Extract Method

- A comment explaining what is being done usually indicates the need to extract a method

```
public double totalArea() {
  ...
  // now add the circle
  area += PI * pow(radius,2);
  ...
}
```

```
public double totalArea() {
  ...
  area += circleArea(radius);
  ...
}

private double circleArea(double radius) {
  return PI * pow(radius, 2);
}
```

- One of the most common refactorings

Extract Method

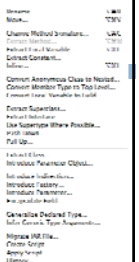
- Simplifying conditionals with Extract Method

```
before
    if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
      charge = quantity * _winterRate + _winterServiceCharge;
    }
    else {
      charge = quantity * _summerRate;
    }
```

```
after
    if (isSummer(date)) {
      charge = summerCharge(quantity);
    }
    else {
      charge = winterCharge(quantity);
    }
```

Refactoring & Tests

- Eclipse supports various refactorings
- You can refactor manually
  - Automated tests are essential to ensure external behavior doesn't change
  - Don't refactor manually without retesting to make sure you didn't break the code you were "improving"!
- More about tests and how to drive development with tests next week



Back to the future...

- All of this leads back to solving problems like our bike-ride comparisons
- In fact we'll solve a similar problem for homework!
  - Animals with DNA containing genes
  - Comparing genes is like comparing bike routes
  - Comparing animals is like finding people who rode similar sets of bike routes
- But solution will need some "tricks of the trade" that we'll cover in the next few weeks of classes…