Big programs

CS 2110

Fall 2016

Announcements

- optional final (details soon)

Programs in the real world

Programs are big
- A5 solution: 2k loc
- All solution: 4k loc
- fabric: 130k loc
- linux: 15000k (15m) loc
- http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

Programs also last a long time
- “small” programs: years
- “large” programs: decades (so far!)

Corollary: code will be read by many different people

Your only hope: forget almost everything

- need to be able to focus on one thing at a time
- need to be able to “zoom in and out”
- trees have depth log n

OOP tools help you forget

- type checking
  - let the compiler remember things for you!
- access control (public/private/protected)
  - implementor can ignore everyone else
  - everyone else can ignore implementation
- interfaces
  - by using a narrow interface, all irrelevant details hidden (e.g. Set<E> or Collection<E> or Iterable<E>)

How to forget:

- Write down specifications
- Write down specifications (really!)
- Program as generically as possible
  - Use the most general interface possible
  - Generic code is easier to write
- Reuse existing abstractions
  - Especially the standard library
- Organize code
  - Put things you need to think about together together;
  - Separate things that can be thought about separately
- Reduce dependencies (decouple)
  - The fewer names a reader needs to know while reading your code, the better.
Design patterns

- Ideas for structuring code to help organize and decouple code
- Read “Gang of Four book” (Design Patterns by Gamma, Helm, Johnson, Vlissides)
- Examples: Observer, MVC, Visitor, ...

Code organization example: Visitor pattern

- Recall expression trees:
  ```java
  interface Expr {
    int evaluate();
    String infix();
    String postfix();
    Expr simplify();
    void generateCode(CompilerState c);
    void typeCheck(Type t);
    void transmit(NetworkConnection conn);
  }
  ```

- What might we do with them?
  - evaluate, print infix, print postfix
  - simplify
  - generate code, type check, optimize
  - transmit over network

- Put these all in Expr?

  ```java
  package interp;
  class Evaluator implements Visitor {
    int visitSum(Sum s) {
      ...
    }
    int visitProduct(Product p) {
      ...
    }
    int visitNumber(Number n) {
      ...
    }
  }
  ```

- Yuck
  - Dependencies
  - Have to think about everything in one place

Let’s make a common interface

  ```java
  package interp;
  class Evaluator implements Visitor {
    int visitSum(Sum s) {
      ...
    }
    int visitProduct(Product p) {
      ...
    }
    int visitNumber(Number n) {
      ...
    }
  }
  ```

We can use generics to make the return types match

  ```java
  package interp;
  class Evaluator implements Visitor<
  package net;
  class Transmitter implements Visitor<
```
Code organization example: Visitor pattern

The implementation of each visitor is easy:

```java
class Evaluator implements Visitor<Integer> {
    Integer visitSum(Sum s) {
        return s.left.visit(this) + s.right.visit(this);
    }
    Integer visitProd(Prod p) {
        return s.left.visit(this) * s.right.visit(this);
    }
    Integer visitNumber(Number n) { return n.value; }
}
```

Using the visitors is easy too:

```java
int result = expr.visit(new Evaluator());
```

With these ideas, one can write down the right interfaces:

```java
interface Visitor<R> {
    R visitSum(Sum s);
    R visitProduct(Product p);
    R visitNumber(Number n);
}
```

```java
interface Expr {
    R visit(Visitor<R> v);
}
```

The implementation of visit is trivial:

```java
class Sum implements Expr {
    Expr left;
    Expr right;
    <R> R visit(Visitor<R> v) {
        return v.visitSum(this);
    }
}
```

Summary:

- To first order of approximation, your code will last forever
- Write code to be read and extended
- Think about what a reader needs to think about to understand your code.
- Design patterns can help