Big programs

CS 2110

Fall 2016
Announcements

- optional final (details soon)
Programs in the real world

Programs are big

- A5 solution: 2k loc
- A8 solution: 4k loc
Programs in the real world

Programs are big

- A5 solution: 2k loc
- A8 solution: 4k loc
- fabric: 130k loc
Programs in the real world

Programs are big

- A5 solution: 2k loc
- A8 solution: 4k loc
- fabric: 130k loc
- linux: 15000k (15m) loc
Programs in the real world

Programs are big
- A5 solution: 2k loc
- A8 solution: 4k loc
- fabric: 130k loc
- linux: 15000k (15m) loc
- [http://www-informationisbeautiful-net/visualizations/million-lines-of-code/]
Programs in the real world

Programs are big

- A5 solution: 2k loc
- A8 solution: 4k loc
- fabric: 130k loc
- linux: 15000k (15m) loc
- [http://www.informationisbeautiful.net/visualizations/million-lines-of-code/](http://www.informationisbeautiful.net/visualizations/million-lines-of-code/)

Programs also last a long time

- "small" programs: years
- "large" programs: decades (so far!)
Programs in the real world

Programs are big

- A5 solution: 2k loc
- A8 solution: 4k loc
- fabric: 130k loc
- linux: 15000k (15m) loc
- [http://www.informationisbeautiful.net/visualizations/million-lines-of-code/](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
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"
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>`)
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
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

- 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

- 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.
How to forget:

- Write down specifications
- Write down specifications (really!)
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.
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
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
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, ...
Recall expression trees:

```java
interface Expr {
    ...
}

class Sum implements Expr {
    Expr left; Expr right; ...
}

class Product implements Expr {
    Expr left; Expr right; ...
}

class Number implements Expr {
    int value; ...
}
```

What might we do with them?
- evaluate, print infix, print postfix
Code organization example: Visitor pattern

- Recall expression trees:

```java
interface Expr {
  ...
}

class Sum implements Expr {
  Expr left; Expr right; ...
}

class Product implements Expr {
  Expr left; Expr right; ...
}

class Number implements Expr {
  int value; ...
}
```

- What might we do with them?
  - evaluate, print infix, print postfix
  - simplify
  - generate code, type check, optimize
  - transmit over network
Code organization example: Visitor pattern

- Recall expression trees:

```java
interface Expr {
    ...
}

class Sum implements Expr {
    Expr left; Expr right; ...
}

class Product implements Expr {
    Expr left; Expr right; ...
}

class Number implements Expr {
    int value; ...
}
```

- What might we do with them?
  - evaluate, print infix, print postfix
  - simplify
  - generate code, type check, optimize
Code organization example: Visitor pattern

- Recall expression trees:

```java
interface Expr {
    ...
}

class Sum implements Expr {
    Expr left; Expr right; ...
}

class Product implements Expr {
    Expr left; Expr right; ...
}

class Number implements Expr {
    int value; ...
}
```

- What might we do with them?
  - evaluate, print infix, print postfix
  - simplify
  - generate code, type check, optimize
  - transmit over network
Code organization example: Visitor pattern

▶ put these all in Expr?

```
interface Expr {
    int evaluate();
    String infix();
    String postfix();
    Expr simplify();
    void generateCode(CompilerState c);
    void typeCheck(Type t);
    void transmit(NetworkConnection conn);
}
```
Code organization example: Visitor pattern

- put these all in Expr?

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

- Yuck
  - Dependencies
  - Have to think about everything in one place
Code organization example: Visitor pattern

What would we like?
What would we like?

```java
package interp;
class Evaluator {
    int evaluateSum(Sum s) { ... }
    int evaluateProduct(Product p) { ... }
    int evaluateNumber(Number n) { ... }
}
```

```java
package net;
class Transmitter {
    void transmitSum(Sum s) { ... }
    void transmitProduct(Product p) { ... }
    void transmitNumber(Number n) { ... }
}
```
Code organization example: Visitor pattern

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) { ... }
}
```

```java
package net;

class Transmitter implements Visitor {
    void visitSum(Sum s) { ... }
    void visitProduct (Product p) { ... }
    void visitNumber(Number n) { ... }
}
```
Code organization example: Visitor pattern

We can use generics to make the return types match

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

```java
package net;
class Transmitter implements Visitor<Void> {
    Void visitSum(Sum s) { ... }
    Void visitProduct(Product p) { ... }
    Void visitNumber(Number n) { ... }
}
```
Code organization example: Visitor pattern

The implementation of each visitor is easy:

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:

// instead of "int result = expr.eval();"
int result = expr.visit(new Evaluator());

// instead of "expr.transmit();"
expr.visit(new Transmitter());
Code organization example: Visitor pattern

With these ideas, can write down the right interfaces:

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

interface Expr {
    <R> 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