# CSE 331

Design Patterns

Kevin Zatloukal

Administrivia

- **HW8: level 4?**
  - state in both the client and server
  - lots of debugging

- **HW9 will be released tomorrow**

- **Monday is a holiday**
  - no lecture
  - no OHs unless announced otherwise (on Ed)

Design Patterns

- **Introduced in the book of that name**
  - written by the "Gang of Four"
    - Gamma, Helm, Johnson, Vlissides
  - worked in C++ and SmallTalk

- **Found that they independently developed many of the same solutions to recurring problems**
  - wrote a book about them
  - required at least three real-world uses to be included
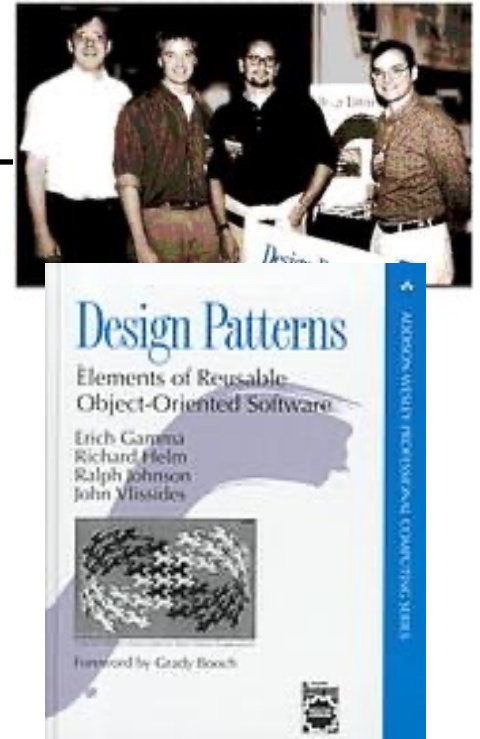
Reasons for Design Patterns

- ## Many are solutions to problems with OO languages
  - authors worked in C++ and SmallTalk

- ## Some are techniques for increasing changeability
  - albeit it a cost in terms of abstraction & complexity

    more abstraction will likely make debugging harder

    do not over-use design patterns!

- ## Terminology itself is often useful
  - shorthand description of a design
  - high-level programming idiom

Parts of a Design Patterns

Each pattern in the book includes

- **Problem** to be solved

- **Description** of the solution

- **Name** of the pattern

Java Example: Iterator

- **Java Collections use the Iterator Design Pattern**
  - **enumerate a collection while hiding data structure details**
  - **return another ADT that outputs the items**
    - that object knows how to walk through the data structure
    - operations for retrieving the current item and moving on to the next one

- **Clever idea that is now used everywhere**
  - **I remember when C++ introduced iterators**
  - **huge improvement over code we were writing before**

Categories of Design Patterns

The book has three categories of patterns

- **Creational**: factory function, factory object, builder, prototype, singleton, ...

- **Structural**: adapter, bridge, composite, decorator, façade, flyweight, proxy

- **Behavioral**: command, interpreter, iterator, mediator, observer, state, strategy, visitor, ...

  – we will not cover all, just some highlights

Categories of Design Patterns

The book has three categories of patterns

- **Creational**:   factory function, factory object, builder, prototype, singleton, ...

- **Structural**:   adapter, bridge, composite, decorator, façade, flyweight, proxy

- **Behavioral**:   command, interpreter, iterator, mediator, observer, state, strategy, visitor, ...

  - green = mentioned already

Creational Patterns

- One third of the patterns deal with object creation

- We saw why last time: constructors suck
  - surprisingly error-prone
  - several important limitations
    1. Cannot return an existing object
    2. Cannot return a different class
    3. Does not have a name!

- Already saw factory functions and singleton
  - yet we still need more!

Creational Pattern: Builder

- **Object that helps with creation of another object**
  - constructor / factory requires you to give info all at once
  - builder lets you describe what you want bit by bit

- **Java Example:** `StringBuilder`

```java
StringBuilder buf = new StringBuilder();
buf.append("Total distance: ");
buf.append(distance);
buf.append(" meters.");
return buf.toString();
```

  - each call adds more text / number to the final string
  - we can't do this with strings because strings are *immutable*

Creational Pattern: Builder

- A Builder helps with creation of another class
  - constructor requires you to give info all at once
  - builder lets you describe what you want bit by bit

- Good pairing: mutable Builder for an immutable type
  - must avoid aliasing with the mutable builder
    - e.g., never use it as a key in a BST or Map
  - immutable object can be shared arbitrarily
    - no worries about aliasing
  - only need to be extra careful with the mutable part

Creational Pattern: Builder

- **Builder** is often written like this:

```java
class FooBuilder {
   …
   public FooBuilder setX(int x) {
      this.x = x;
      return this;
   }
   …
   public Foo build() { … }
}
```

 – can then use them like this

```java
Foo f = new FooBuilder().setX(1).setY(2).build();
```

avoids worries about argument order

Structural Pattern: Adapter

- **Mentioned of these in lecture 3: adapter**

- **In Java, these two classes are not interoperable:**

```java
interface Duration {
  int getMinutes();
  int getSeconds();
}

interface AmountOfTime {
  int getMinutes();
  int getSeconds();
}
```

  – cannot pass one where the other is expected

Structural Pattern: Adapter

- **Mentioned of these in lecture 3: adapter**

- **Get around this by creating an adapter**

```
class DurationAdapter implements AmountOfTime {

    private Duration d;

    int getMinutes() { return d.getMinutes(); }
    int getSeconds() { return d.getSeconds(); }
}
```

- **makes a** `Duration` **into an** `AmountOfTime`

Structural Pattern: Adapter

- **Adapters are often needed with nominal typing**
  - design pattern working around a language issue

- **With structural typing, these two interoperate:**

```
type Duration = {min: number, sec: number};

type AmountOfTime = {min: number, sec: number};
```
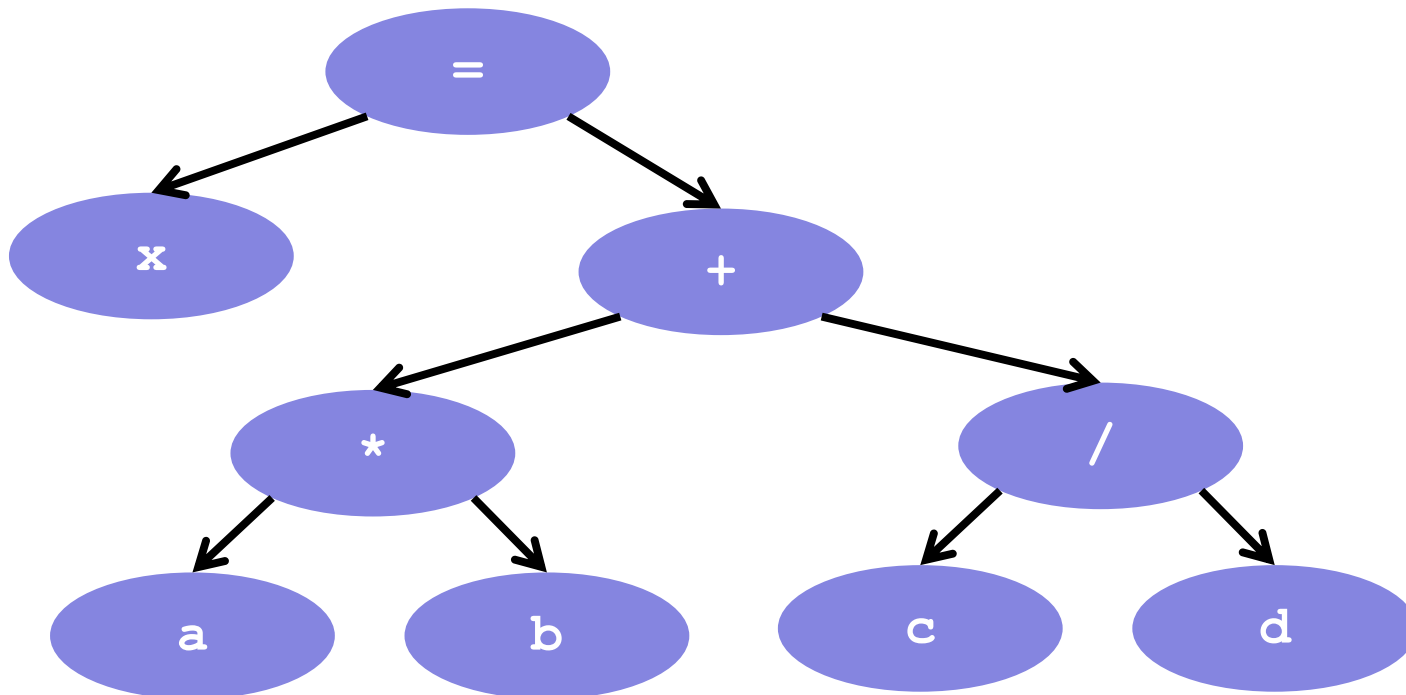
  - **can pass either where the other is expected**
  - **not an issue of concrete vs abstract**

    still interoperable if we have `getMinutes` and `getSeconds` methods

Trees

- **Trees are inductive data types**
  - anything with a constructor that has **2+ recursive arguments**
    HW8 tree (Square) has 4 recursive arguments

- **They arise frequently in practice**
  - HTML: used to describe UI
  - JSON: used for client/server communication
  - parse trees: represent code

Parse Tree

- **Output of parsing is a tree**
  - encodes the order of operations

- **Example: parse of** "x = a * b + c / d"

Parse Tree

- Output of parsing is a tree
  - records the order of operations

- Parse tree is an inductive data type

$$\textbf{type } \text{Expression} := \text{variable}(\text{name: } \mathbb{S}^*)$$
$$| \quad \text{constant}(\text{val} : \mathbb{Z})$$
$$| \quad \text{plus}(\text{left} : \text{Expr}, \text{right} : \text{Expr})$$
$$| \quad \text{times}(\text{left} : \text{Expr}, \text{right} : \text{Expr})$$
$$| \quad \text{divide}(\text{left} : \text{Expr}, \text{right} : \text{Expr})$$
$$| \quad \text{assign}(\text{name} : \mathbb{S}^*, \text{value} : \text{Expr})$$

  - **parse of** "`x = a * b + c / d`"

$$\text{assign}(\text{"x"}, \text{plus}(\text{times}(\text{variable}(\text{"a"}), \text{variable}(\text{"b"})),$$
$$\text{divide}(\text{variable}(\text{"c"}), \text{variable}(\text{"d"})))$$

Operations on Parse Trees

- Compilers perform various operations on expressions
  - type check
  - evaluate
  - generate code

- Each operation defined for each type of expression

Type of Expr

	Variable	Plus	Times
type check			
eval			
code gen			

Operation

Operations on Parse Trees

- Need to write code for each box
  - each case is slightly different

- Two reasonable ways to organize into files
  - file per expression type: Interpreter pattern
  - file per operation: Procedural pattern

Type of Expr

	Variable	Plus	Times
type check			
eval			
code gen			

Operation

Interpreter **Pattern**

```
interface Expr {
  typeCheck = (c: Context) => Type,
  evaluate = (c: Context) => number | undefined,
  generate = (c: Context) => List<Instruction>
}

class Variable implements Expr {
  name: string;
  typeCheck = (c: Context): Type => {
    return c.get(this.name);
  }
  evaluate = (c: Context): number | undefined => {
    return undefined;
  }
  …
}
```
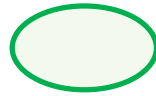
- **Each type of expression is a class**

Interpreter **Pattern**

```
interface Expr {
    typeCheck = (c: Context) => Type,
    evaluate = (c: Context) => number | undefined,
    generate = (c: Context) => List<Instruction>
}
```

- **Easy to add new types of expression**
  - new subtype of `Expr`
  - goes into its own file

- **Hard to add new operations**
  - new method of `Expr`
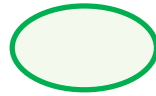 - changes *every* file

Procedural Pattern

```
interface Procedure<R> {
  processVar = (v: Variable, c: Context) => R,
  processConst = (n: Constant, c: Context) => R,
  …
}

class TypeChecker implements Procedure<boolean> {
  processVar = (v: Variable, c: Context): boolean => {
    return c.has(v.name);
  }
  processConst = (n: Constant, c: Context): boolean => {
    return true;
  }
  …
}
```

- **Each type of procedure is a class**
  - one method for each type of expression

Procedural Pattern

```
interface Procedure<R> {
  processVar = (v: Variable, c: Context) => R,
  processConst = (n: Constant, c: Context) => R,
  …
}
```

- **Easy to add new types of operations**
  - **new subtype of** `Procedure`
  - **goes into its own file**

- **Hard to add new expressions**
  - **new method of** `Procedure`
  - **changes** *every* **file**

Interpreter vs Procedural Pattern

- **Both patterns are reasonable**
  - **best choice is problem-dependent**

      for a compiler, I prefer the procedural pattern

- **But there is a problem with Procedural in OO**
  - **suppose `e` is an `Expr` but we don't know which one**
  - **how do we call the right method?**

      could be `processVar`, `processConst`, `processPlus`, …

Problems with Procedural Pattern in OO

```
function process(p: Procedure, e: Expr, c: Context) {
  if (e instanceof Variable) {
    p.processVar(e, c);
  } else if (e instanceof Constant) {
    p.processConst(e, c);
  } else if (e instanceof Plus) {
    p.processPlus(e, c);
  } else …
}
```

- ## Not great, Bob!
  - code is slow
  - will call it enough times that this will matter

- ## There is a solution, but… buckle up!

Dynamic Dispatch (good case in Java)

```java
interface Expr {
  boolean typeCheck(Context c);
}

class Variable implements Expr {
  public boolean typeCheck(Context c) { … }
}

class Constant implements Expr {
  public boolean typeCheck(Context c) { … }
}
```

- Java / TypeScript (or any OO) makes this case easy

```java
Expr e = …
e.typeCheck(c);          // e could be any Expr
```

- automatically "dispatches" to the right method

Dynamic Dispatch (bad case in Java)

```java
interface Procedure<R> {
  R process(Variable v, Context c);
  R process(Constant n, Context c);
  …
}

class TypeChecker implements Procedure<Boolean> {
  Boolean process(Variable v, Context c) { … }
  Boolean process(Constant c, Context c) { … }
  …
}
```

overloading

- **This is impossible in Java:**

```java
Process p = new TypeChecker();
Expr e = …
p.process(e, c);          // e could be any Expr
```

Dynamic Dispatch (bad case in Java)

- **This is impossible in Java:**

```
Process p = new TypeChecker();
Expr e = …
p.process(e, c);          // e could be any Expr
```

- **Need to put "e" before "." to get dynamic dispatch**
  - here's how we do that... (gulp)

Double Dispatch

```
interface Procedure<R> {
  R process(Variable v, Context c);
  R process(Constant n, Context c);
  …
}

interface Expr {
  R perform(Procedure<R> p, Context c);
}

class Variable implements Expr {
  public R perform(Procedure<R> p, Context c) {
    p.process(this, c);
  }                         calls process(Variable, Context)
}

class Constant implements Expr {
  public R perform(Procedure<R> p, Context c) {
    p.process(this, c);
  }                         calls process(Constant, Context)
}
```

Double Dispatch

```java
interface Procedure<R> {
  R process(Variable v, Context c);
  R process(Constant n, Context c);
  …
}

interface Expr {
  R perform(Procedure<R> p, Context c);
}
```

- ## We can now do this

```java
Process p = new TypeChecker();
Expr e = …
e.perform(p, c);          // e could be any Expr
```

  – calls `Expr.perform`, which calls `TypeChecker.process`
  – two function calls is still faster than all the "`if`"s

Double Dispatch

- **This works, but... why so hard?**

- **Other languages just let you do this:**

```
Process p = new TypeChecker();
Expr e = …
p.process(e, c);              // e could be any Expr
```

  – or even more general "multiple dispatch" cases
  – use a better language?

Look at this. Look at what they make you give.

Traversing Trees

- ## Same idea is used to traverse trees

$$\textbf{type}\ \text{Expression} := \text{variable}(\text{name}: \mathbb{S}^*)$$
$$|\quad \text{constant}(\text{val} : \mathbb{Z})$$
$$|\quad \text{plus}(\text{left} : \text{Expr}, \text{right} : \text{Expr})$$
$$|\quad \text{times}(\text{left} : \text{Expr}, \text{right} : \text{Expr})$$
$$|\quad \text{divide}(\text{left} : \text{Expr}, \text{right} : \text{Expr})$$
$$|\quad \text{assign}(\text{name} : \mathbb{S}^*, \text{value} : \text{Expr})$$

- **parse of "`x = a * b + c / d`"**

assign("x", plus(times(variable("a"), variable("b")),
divide(variable("c"), variable("d"))))

- **would like to process ("visit") each node in this tree**

Visitor Pattern

```typescript
interface ExprVisitor {
  visitVariable = (v: Variable) => void,
  visitConstant = (n: Constant) => void,
  visitPlus = (p: Plus) => void,
  …
}

interface Expr {
  // Visits this node and all its children.
  accept = (v: ExprVisitor) => void
}

class Variable implements Expr {
  name: string;
  accept = (v: ExprVisitor): void => {
    v.visitVariable(this);
  }
}

…
```

Visitor Pattern

- **Combines double dispatch with tree traversal**

```
class Plus implements Expr {
  left: Expr;
  right: Expr;

  accept = (v: ExprVisitor): void => {
    left.accept(v);
    right.accept(v);
    v.visitVariable(this);
  }
}
```

  – traverses children before visiting parent

Visitor Pattern

```
p.accept(v)
  t.accept(v)
    a.accept(v)
      v.visitVariable(a)
    b.accept(v)
      v.visitConstant(2)
    v.visitTimes(t)
  d.accept(v)
    …
    v.visitDivide(p)
  v.visitPlus(p)
```