Lecture 24:
Design patterns,
Demo advice,
Wrapup

Software methodology and snake oil

- programming is hard
  - programs are very expensive to create
  - full of errors
  - hard to maintain
- how can we design and program better?
- a fruitful area for people selling "methodologies"
  - for at least 50 years
- each methodology has the germ of a useful idea
- each claims to solve major programming problems
- some are promoted with religious fervor

- in fact most don't seem to work well
- or don't seem to apply to all programs
- or can't be taught to others

- a few are genuinely useful and should be in everyone's repertoire

Examples of methodologies ...

- modularity, information hiding, coupling, cohesion
- structured programming (programming without goto's)
  - top-down development, successive refinement
  - chief programmer teams, egoless programming
  - structured design, analysis, requirements, specification, walkthroughs...
- CASE tools (Computer Aided Software Engineering)
  - UML (Unified Modeling Language), message sequence charts, state diagrams
- formal methods
  - verification, validation, correctness proofs, model checking
- object-oriented programming
  - CRC cards (Class, Responsibilities, and Collaborators)
  - object-oriented design, analysis, requirements, specification, walkthroughs, ...
- RAD (Rapid Application Development)
  - components, COTS (Components off the Shelf)
  - 4th generation languages, automatic programming, X by example, graphical programming
- design patterns
  - patterns of everything

More recent examples...

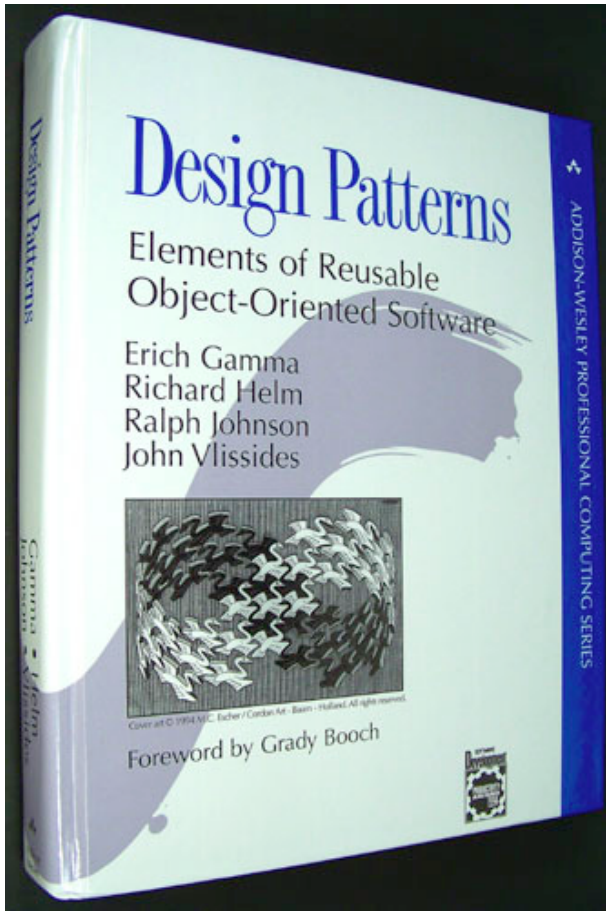
- extreme programming, refactoring, agile methods
- test-driven design
- pair programming
- aspect-oriented programming
- Scrum
- Kanban
- Continuous Integration: CircleCI, Travis CI, Jenkins
- "X as a Service" (for all X)

Design patterns

- "Design patterns ... describe simple and elegant solutions to specific problems in object-oriented software design."
  - *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma, Helm, Johnson, Vlissides (the "Gang of Four"), 1995

- "idioms for design" or program structure
  - successful among broad group of programmers
  - widely used to describe software structure

- three basic categories:
  - creational: making things
  - structural: organizing things
  - behavioral: operating things

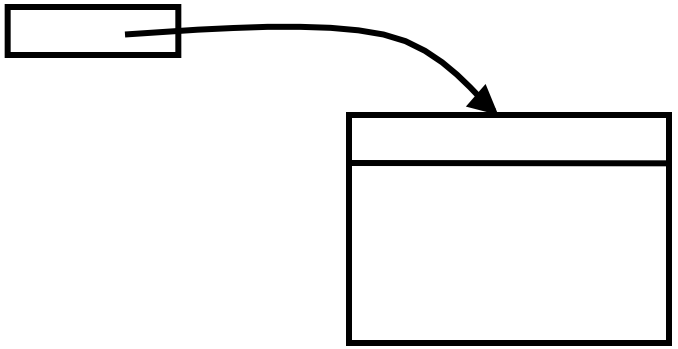


Bridge (or "handle/body") pattern

- "Decouple an abstraction from its implementation so that the two can vary independently"
- C++ string class: separate handle from body
  - implementation can be changed without changing abstraction of "string"

```
class String {
   private:
        Srep *p;
   public:
        ...

};
class Srep {
        char *sp;   // data
        int   n;    // ref count
        ...
};
```



- similar examples:
  - FILE * in C stdio, RE * in regexpr interface, connection in MySQL interface
- change of implementation has no effect on client
  - can even switch implementation at run time
- (in C and C++) hides implementation completely
  - C: hidden behind opaque type; C++: implementation class is invisible
- can share implementation among multiple objects without revealing the sharing
  - e.g., reference counting, sharing of open files in FILE*

Adapter (or Wrapper) pattern

- "Convert the interface of one class into another interface that clients expect"

- maps one interface into another
  - more or less at the same level

- e.g., in the C stdio package:

  fread(buf, objsize, nobj, stream)

  fwrite(buf, objsize, nobj, stream)

  are wrappers around

  read(fd, buf, size)

  write(fd, buf, size)

- e.g., Java box types like Integer, Double, etc.

Decorator pattern

- "Attach additional responsibilities to an object dynamically"

- decorator conforms to interface it decorates
  - transparent to clients
  - may forward some requests
  - usually does some actions of its own before or after

- example: Java buffered I/O streams
  - responsibility for buffering is attached dynamically
  - interface remains unchanged

```
FileInputStream fin = new FileInputStream(args[0]);
FileOutputStream fout = new FileOutputStream(args[1]);

BufferedInputStream bin = new BufferedInputStream(fin);
BufferedOutputStream bout = new BufferedOutputStream(fout);
```

Creational patterns

- Abstract Factory: "Provide an interface for creating families of related or dependent objects." (also Builder and Factory)
  - DOM and SAX builder factories
- Singleton: "Ensure a class only has one instance"
  - Java System, Runtime, Math classes
- Prototype: "Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype."
  - Javascript objects

Behavorial patterns

- Observer: "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically"

- e.g, Javascript onWhatever() events:

```
<form>
<input type=button value="Start" onClick='newgame()'>
</form>
```

  – called when Click event occurs on button

Behavorial patterns (2)

- Iterator: "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation"
  - C++ STL, Java Collections classes. SQL databases

```
Map hs = new TreeMap();
for (Iterator it : hs.keySet()) {
      String n = (String) it.next();
      Integer v = (Integer) hs.get(n);
      ...
```

- Visitor: "Represent an operation to be performed on the elements of an object structure"
  - almost any tree walk that does some evaluation at each node
  - draw() where one kind of "Shape" is an entire picture made of Shapes

- Memento: "Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later"
  - JSON, Java serialization, tar file, ...

Behavioral patterns (3)

- Interpreter: "Given a language, define a representation for its grammar along with an interpreter that uses the presentation to interpret sentences in the language"

- regular expression processors
- eval(…) or execute(…) in many languages
- printf format strings

- domain-specific / application-oriented languages