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 30 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 part of everyone's repertoire**

Examples...

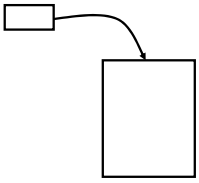
- **modularity, information hiding (Parnas)**
  - coupling, cohesion (Constantine)
- **structured programming (programming without goto's)**
  - top-down development, successive refinement
  - structured everything
    design, analysis, requirements, specification, walkthroughs...
  - chief programmer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

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")
- **successful among broad group of programmers**
- **increasingly used to describe software structure**

Bridge 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
        ...
};
```

- **sometimes called "Handle / Body"**
- **similar examples:**
  - FILE * in C stdio
  - RE * in regexpr interface
  - connection in MySQL interface

Bridge pattern, continued

- **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
  - e.g., sharing of open files in FILE*

Adapter 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)
- **also known as "wrapper" pattern**
- **real-world examples:**
  - electrical plugs, various other connectors

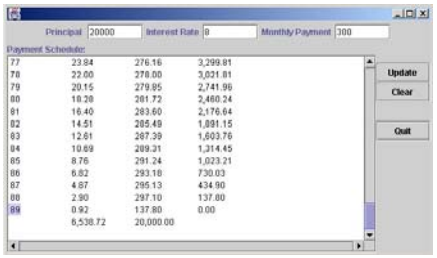
Decorator pattern

- **"Attach additional responsibilities to an object dynamically"**

- **decorator conforms to interface it decorates**
  - transparent to clients
  - forwards some requests
  - usually does some actions of its own before or after
- **e.g., Java Swing JScrollPane class**

```
JTextArea tpay = new JTextArea(15, 45);

JScrollPane jsp = new JScrollPane(tpay,
   JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
   JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
```



Other structural patterns

- **Facade: "Provide a unified interface to a set of interfaces in a subsystem."**
  - provides a higher-level interface to something underneath that remains visible and accessible
  - Perl CGI package (and others)
  - simplified socket package (Perl and others)
  - graphics interfaces
    (X widgets -> X toolkits -> X intrinsics -> Xlib)
  - …

- **Proxy: "Provide a surrogate or placeholder for another object to control access to it."**
  - smart pointers
  - implicit initialization
  - load on demand (lazy evaluation)
  - …

- **how do we tell all of these patterns apart?**
  - distinctions are not always clear

Iterator

- "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation"

- in Java, iterators and tokenizers

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

- the basis of algorithms in C++ STL

```
template <class InputIterator,
          class OutputIterator>
OutputIterator mycopy(InputIterator first,
   InputIterator last, OutputIterator result)
{
    ...
}
```

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 processor
  - variations of grep

```
int match(char *regexp, char *text) ...
```

- eval(…) or execute(…) in many languages

- printf format strings?

Observer (/observable)

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

- **Java ActionListener mechanism:**

  `button.addActionListener(this)`
  - tells **button** to notify **this** container when event happens
  - usually called by container that contains object that will get the event
  - can have more than one listener

  `void actionPerformed(ActionEvent e) { … }`
  - called when event occurs
  - determines type or instance that caused event
  - handles it

Others...

- **Abstract Factory: "Provide an interface for creating families of related or dependent objects." (also Factory)**

- **Singleton: "Ensure a class only has one instance"**
  - Java System, Runtime, Math classes

- **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"**
  - Java serialization