Design patterns

- So far, we have concentrated on **reusable software**.
- A **design pattern** is a **reusable practice** for designing **software**.
- Simply stated, a pattern is a **proven way of solving some problem**.
- So far, we've concentrated on **principles that guide practice**.
- A design pattern is a **practice based upon principles**.
Naïve concept of a pattern

Description with blanks in it.
You "fill in the blanks" to get software.
There is documentation on
  What kinds of things fit in the blanks.
  (preconditions)
  What you can expect if you put them there. (postconditions)
A pattern is a **three-part rule** that expresses a relationship between:
- a **context**: environment in which a problem occurs.
- a **problem**: some issue to be addressed in the context.
- a **solution**: a suggested and proven way of solving the problem.
Design pattern intentions

Monday, November 09, 2009
2:13 PM

Design pattern goals:

Avoid "re-inventing the wheel".
Leverage proven solutions.
Gain proven benefits.
Limit to known risks.
The basic motivation for patterns is economy of scale.  
• If lots of people have successfully used an approach, then it is most likely a good approach.  
• If a few people have successfully used it, it is less attractive and less proven.

More specifically,  
• If 1/2 of the known internet developers use a specific library or toolkit, then  
  ▪ the way they use it is likely to be free of bugs, and  
  ▪ that particular way is unlikely to become deprecated.

Pattern implementations gain validity by adoption.

This is a "500-lb gorilla" approach to choosing technological solutions.
The above logic does not apply if:
  ○ You are doing something quite different than the other people.
  ○ You are using a library in an unusual way.

You don't want to be the only developer using a library in a given way! Then:
  ○ You're the only tester!
  ○ Your use of the library can be deprecated!
Pattern essentials
A clear **mission** or purpose ("value proposition")
A clear **method** to be applied.
**Proof** of successful application.
If a large number of people have used a solution, and they critically analyzed and understood what it did, then you can use it and know what you'll get out of it.
What patterns do:

- **Exploit economy of scale** and the large number of developers working in a similar domain.
- Utilize **global wisdom** rather than local knowledge.
- Expose **known risks** (and limit unknown risks).
Example: encapsulation as a pattern

○ The context: web applications
○ The problem: database interaction with web pages.
○ The solution: **object-relational modeling (ORM)**
  ▪ Define primitive operations on a database as functions.
  ▪ Functions are methods of a "database object" that makes the appropriate changes to the database.
  ▪ Enabling libraries instantiate common operations:
    □ **Create**
    □ **Read**
    □ **Update**
    □ **Delete**
  ▪ This is the so-called **CRUD model** of database interaction.
○ Attributes of the solution: encapsulates database operations, increases inherent portability (and ease of porting) of solution.
Example 2: algorithm as a pattern

The context: web applications
The problem: communication between local JavaScript and remote server (Web 2.0).
The solution: **JavaScript Object Notation (JSON)**. Code information as JavaScript structures. Web client uses **client library** to encode structures as text data.
Text data is sent to server.
Server uses **server library** to decode data, and encode answer.
Client receives response from server in kind.
Attributes of the solution: leverages existing debugged libraries for server-client interaction, including **tested and reliable encoding and decoding**. One doesn't have to write encoding and decoding "again".
Example: architecture as a pattern

The context: writing a web application
The problem: use an architecture that streamlines development
The solution: **Model-View-Controller (MVC)** architecture

- **The model**: database interactions (an ORM).
- **The view**: how that appears to the user.
- **The controller**: manages changes of state in response to user input (uses the model).

Attributes of the solution: separates a web application into three parts that are **usually orthogonal**, eases parallel development and maintenance of M, V, and C.
Example: protocol as a pattern.
The context: making an application out of several geographically distributed services.
The problem: communication between remote web services.
The solution: **Service-Oriented Architecture (SOA)**
Express service requests in SOAP or REST.
Express answers as XML.
Define preconditions and postconditions as Xschemas.
Attributes of the solution: easier to design, decouples parts of solution for parallel development and maintenance, makes resulting code easier to "push into the cloud" for economic reasons.
Kinds of design patterns:

**Algorithmic:** a proven method for solving an algorithmic problem, e.g., greedy heuristic for traveling salesman.

**Abstraction:** allow decoupling of parts of a problem to allow independent development.

**Architectural:** a proven architecture for a kind of solution, e.g., MVC for web applications; ORM for database interactions.

**Protocol:** a proven method for communication between asynchronous components.

... and many more ...
A short list of available patterns

○ Creational
  ▪ Factories: customize an object for a specific purpose.
  ▪ Builder: separate object construction from use to allow different constructions as necessary.
  ▪ Prototype: pre-instantiate parts of an object to reduce factory cost.

○ Structural
  ▪ Adapter pattern: create "middleware" that adapts an existing object for use by a new kind of client.
  ▪ Bridge pattern: decouple an abstraction from its implementation.
  ▪ Container pattern: hold a set of objects to manipulate it.
  ▪ Proxy pattern: a class that functions as an interface to something that might not be an object.
  ▪ Pipes and filters: constructing a dataflow process from primitive dataflow components.

○ Behavioral
  ▪ Command: encapsulate a command and its parameters.
  ▪ Event listener: objects "listen" for events and only respond to those that they recognize.
  ▪ Interpreter: use a specialized computer language to solve a set of problems.
  ▪ Iterator: iterate over a container of objects without reference to how the container is designed.
  ▪ Mediator: provide a unified interface to a set of interfaces within a subsystem.
  ▪ Visitor: separate an algorithm from an object by
defining what is done when one "visits" a node.
Pattern limits

**Known limits**: patterns have limitations. **Square peg; round hole**: patterns can't be applied to just any problem without thinking about it. **Not a pattern**: a pattern isn't a pattern until it's proven by practice. A "personal pattern" is an oxymoron.
Not a pattern

Good idea but not proven.
Unclear specification of problem context or problem to be solved.
Not enough information to apply the pattern.

Example: interface "layering"
When is a pattern inappropriate?
SOA: not applicable when all interaction is local (too expensive to encode and decode)
ORM: not applicable when all database interaction is custom (e.g., user-built queries)
MVC: not applicable when application is too simple to gain from separation.

In general, a pattern has
A cost of adoption
A benefit from adoption
Cost must be less than value!
Example of pattern inapplicability
Pattern: MVC
Problem: new kind of database interaction, e.g., Amazon S3 (scalable storage service).
Difficulty: MVC tied to existing ORM software.
"Square peg in round hole"
Two approaches:
   ignore MVC and proceed with classical web development
   craft a new ORM to plug into existing MVC.
Agile solution: both.
How to specify a pattern:

**Problem:** the problem it solves.

**Motivation:** what you expect adopters to gain.

**Context:** describes the application context to which the pattern applies.

**Forces:** limitations and constraints that should be considered in adopting.

**Solution:** a detailed description of what adopters should do.

**Collaborations:** how other patterns contribute to the solution.

**Consequences:** tradeoffs that must be made when considering adoption.

**Implementation:** special issues that must be considered when implementing the pattern.

**Known uses:** examples of successful adoption.

**Related patterns:** cross-reference to patterns with similar goals.
Aspect-oriented programming

**Problem:** modification of specific data elements often requires modification of related elements, e.g., a display or sum or other function of the elements.

**Motivation:** can forget about side-effects and program changes in data elements without considering them.

**Context:** serial programming in typical high-level languages: C++, C#, Java, J#.

**Forces:** only really effective in decoupling parameter changes from side-effects if the language supports it transparently (C#/J#).

**Solution:** program side-effects of setting a parameter via get and set methods that are transparently used whenever anyone sets or accesses the parameter.

**Collaborations:** none.

**Consequences:** any time one changes a get or set method, one must potentially retest every component that uses it.

**Implementation:** only truly effective when language supports it.

**Known uses:** displaying the value of a changing parameter as a side-effect of changing it. Keeping running sums and statistics of several parameters transparently. Thousands of C# and J# apps.

**Related patterns:** none.
Problem: manage interaction between components of a distributed architecture.

Motivation: easy relocation of service components and cloud-sourcing of appropriate services to exploit economy of scale and elasticity.

Context: composite services ("mashups") composed of many smaller services.

Forces: Does not work well to split up a single atomic service; encoding and decoding time is significant for simpler problems.

Solution: Decompose problem into autonomous services, request services via SOAP or REST, receive responses via XML. Verify requests and responses via Xschemas.

Collaborations: XML, Xschema, SOAP, REST.

Consequences: Overhead for encoding and decoding requests is unacceptably high if components are too simple. Problem of "ontology" makes it difficult to locate relevant services because use of names for items is not universal.
**Implementation:** Use libraries for SOAP on client and server.

**Known uses:** google maps, location-aware services.

**Related patterns:** Remote Procedure Calls (RPC).
What are patterns, really?
A rigorous depiction of what is really "common sense". Not really any different from the way we approach any coding problem. The basic principle: if it worked for others, it'll work for you, unless something about your problem is really different!
Is software engineering creative?
    My answer: **No! It is engineering.**
    Engineering is about using the **lessons from the past**
    to engineer reliable **solutions for the future**.
    "Sometimes the last thing you need is a new idea."

In other words,
    Reliable, quickly implementable, risk-free solutions are
    always preferable to creative, risky ones with unknown
    risk factors.
    Leave the risky approaches to **research!**