Overview of Design Patterns

CS356 Object-Oriented Design and Programming
http://cs356.yusun.io
October 24, 2014

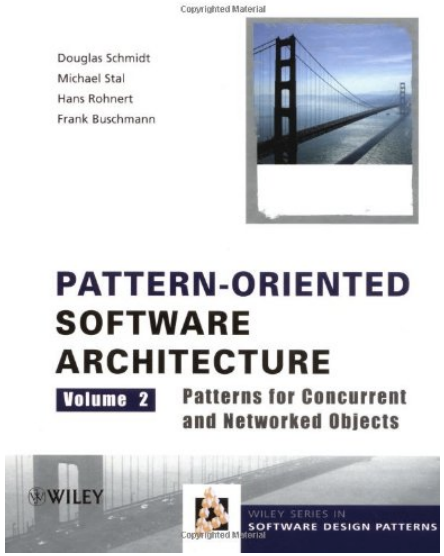
Yu Sun, Ph.D.
http://yusun.io
yusun@csupomona.edu

CAL POLY POMONA

Acknowledgement

- Part of the presentation is based on Prof. Douglas Schmidt's lecture materials on patterns and software design

- http://www.dre.vanderbilt.edu/~schmidt/

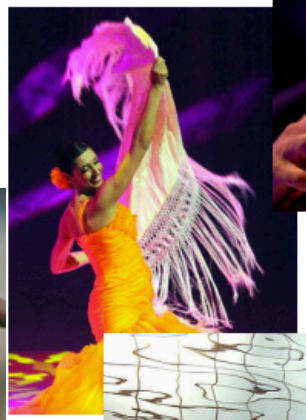


Overview

- Motivate the importance of design experience & leveraging recurring design structure in becoming a master software developer

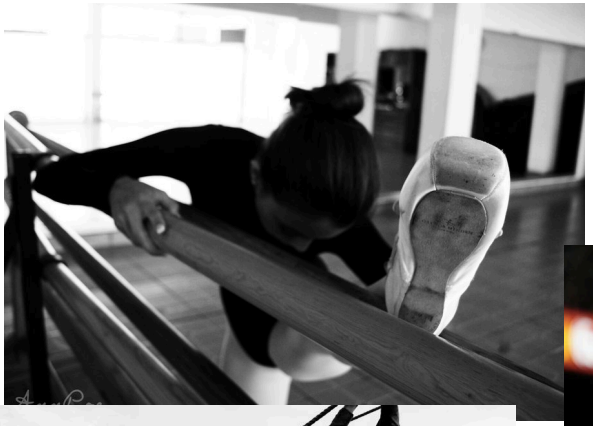
Becoming a Master

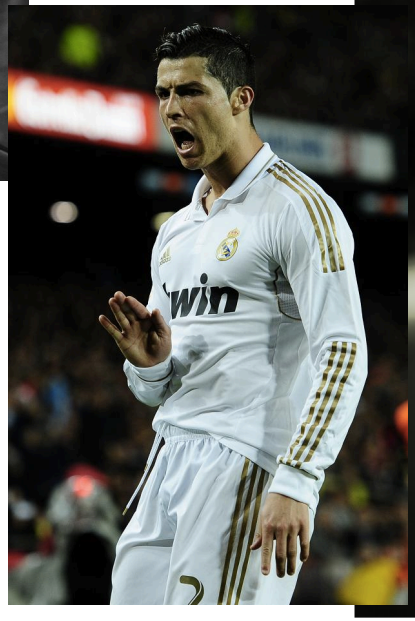
- Experts perform differently than beginners
  - Unlike novices, professional athletes, musicians & dancers move fluidly & effortlessly, without focusing on each individual movement



Becoming a Master

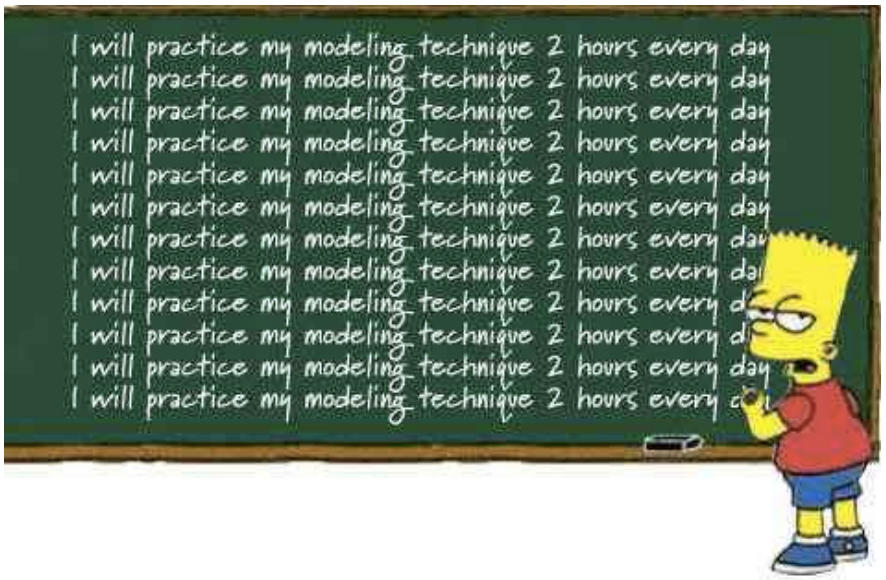
- When watching experts perform, it's easy to forget how much effort they've put into reaching high levels of achievement





Becoming a Master

- Continuous repetition & practice are crucial to success



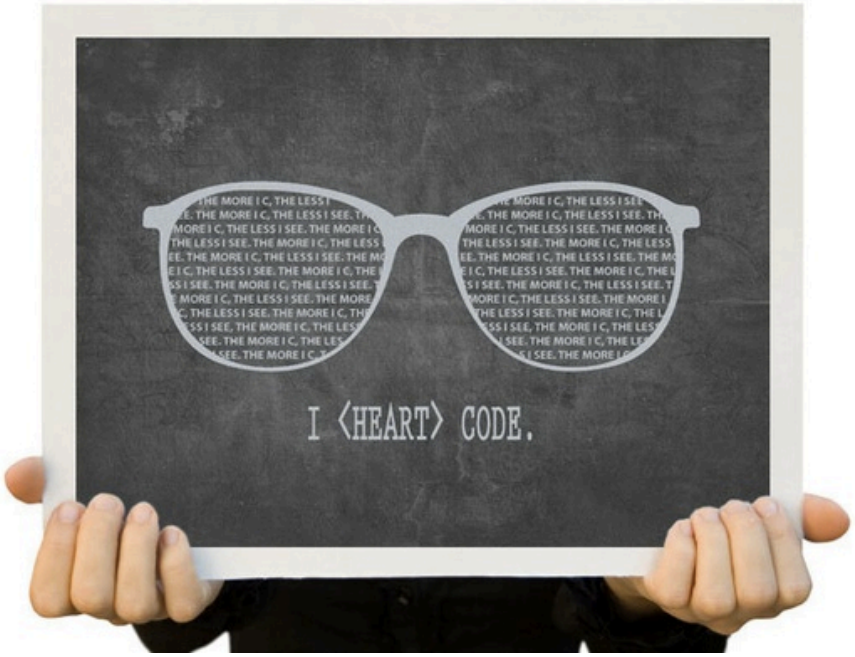
Ted Talk: **The Skill of Self Confidence**
Dr. Ivan Joseph

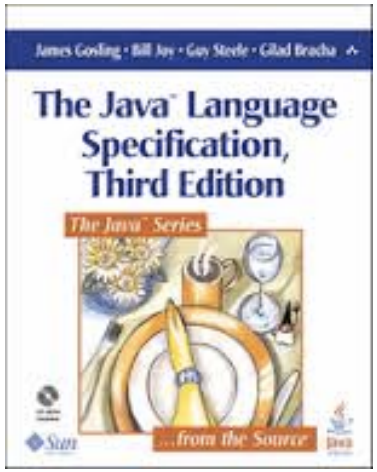
Becoming a Master

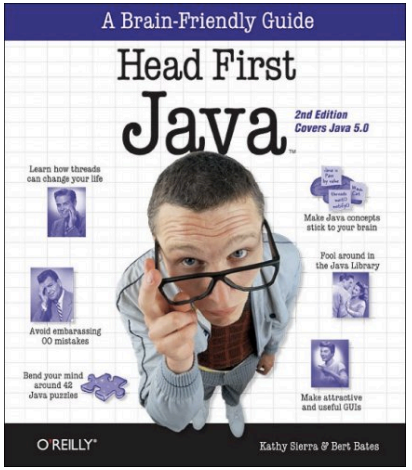
- Mentoring from other experts is also essential to becoming a master

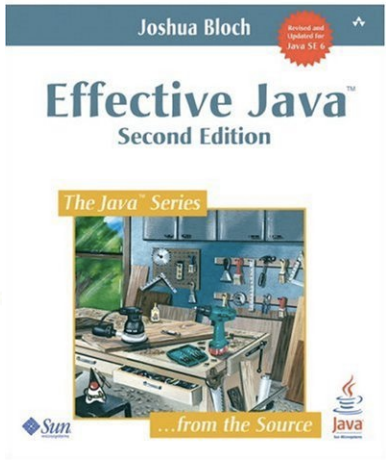
Becoming a Master Software Developer

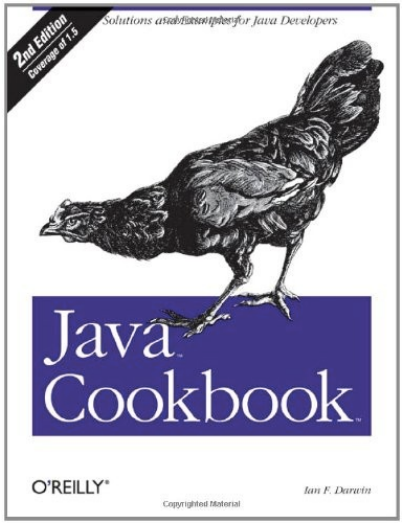
- Knowledge of programming languages is necessary, but not sufficient









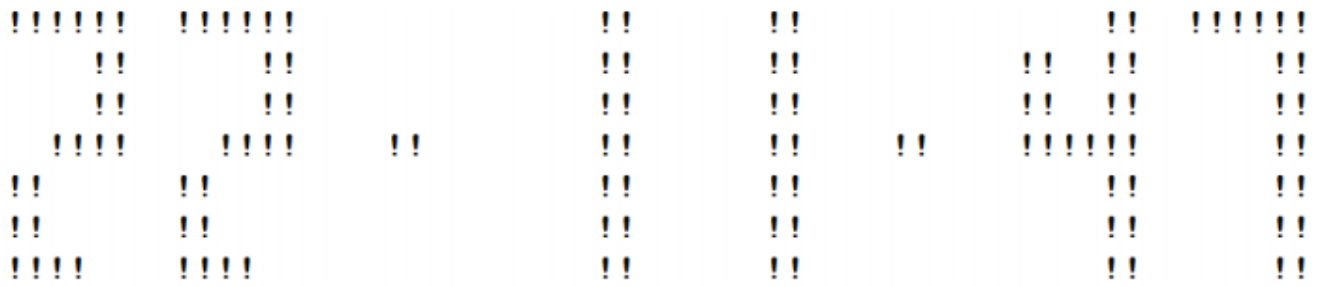


Becoming a Master Software Developer

- Knowledge of programming languages is necessary, but not sufficient
  - e.g., "Best one-liner" from 2006 "Obfuscated C Code" contest

```
main(_){_^448&&main(-~_);putchar(--_%64?32|-~7[
__TIME__-_/8%8]["">'txiZ^(~z?"-48]>>";;;====~$::199"
[_*2&8|_/64]/(_&2?1:8)%8&1:10);}
```

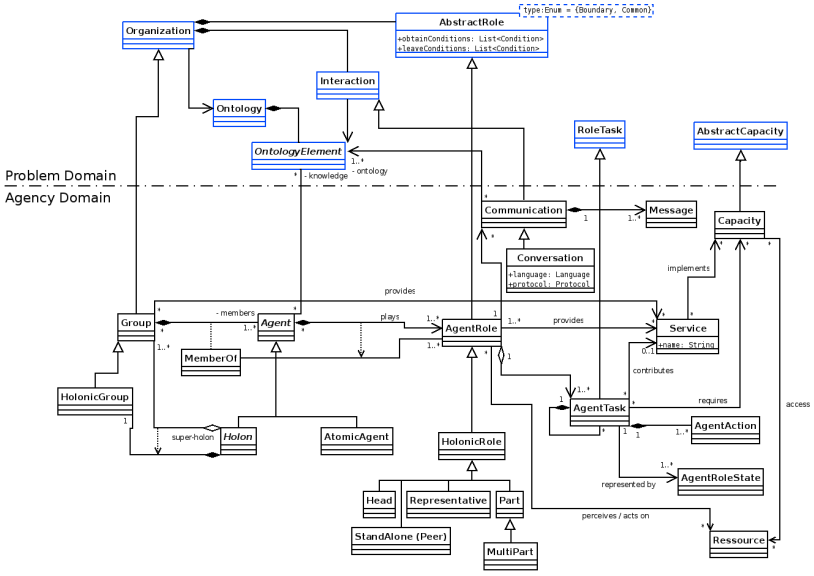
  - This program pints out the time when it was compiled!

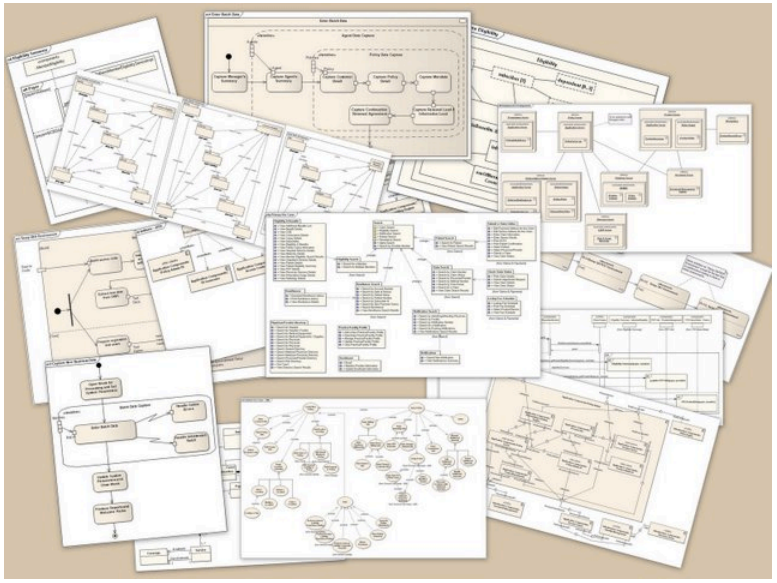


- http://www.ioccc.org/

Becoming a Master Software Developer

- Software methods emphasize design notations, such as UML
  - Fine for specification & documentation
  - e.g., omits mundane implementation details & focuses on relationships between key design entities



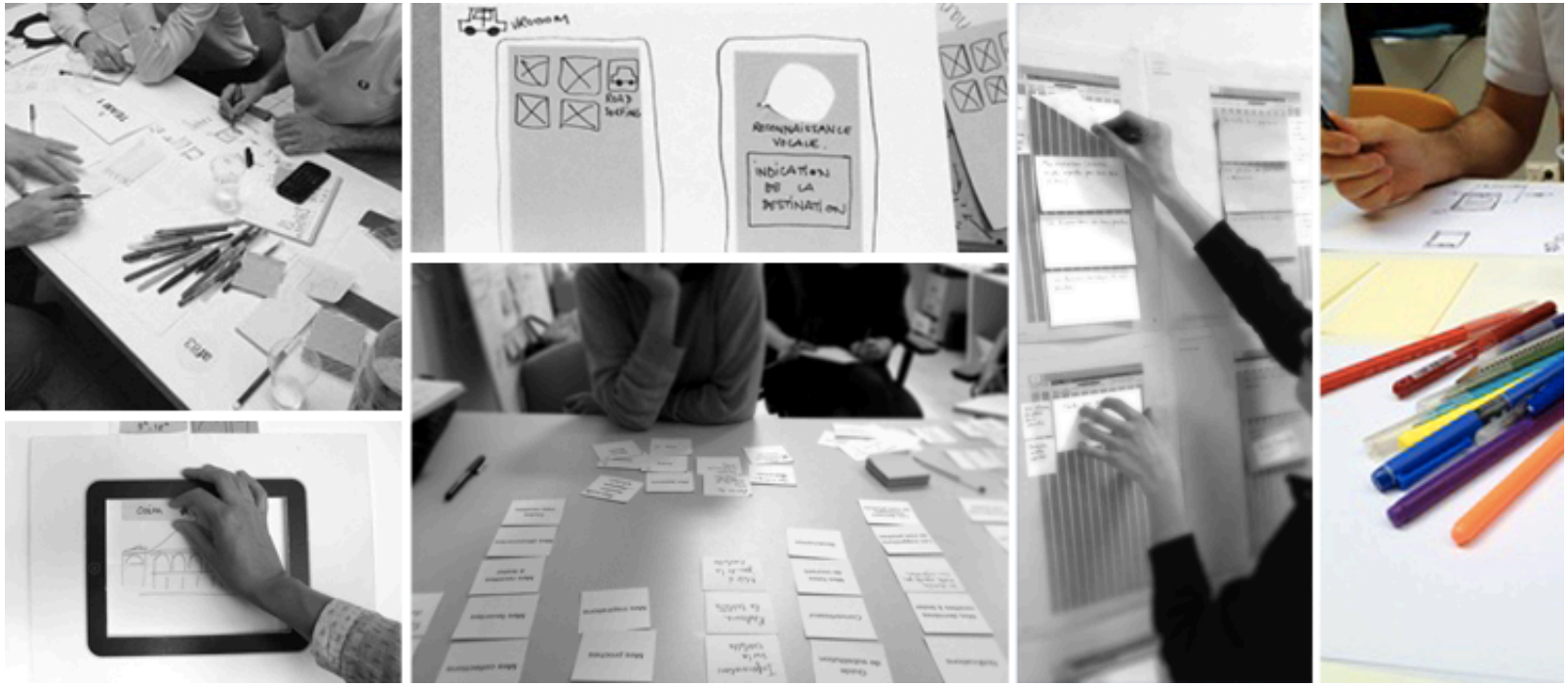


Becoming a Master Software Developer

- But good software design is more than drawing diagrams
  - Good draftsmen/artists are not necessarily good architects!

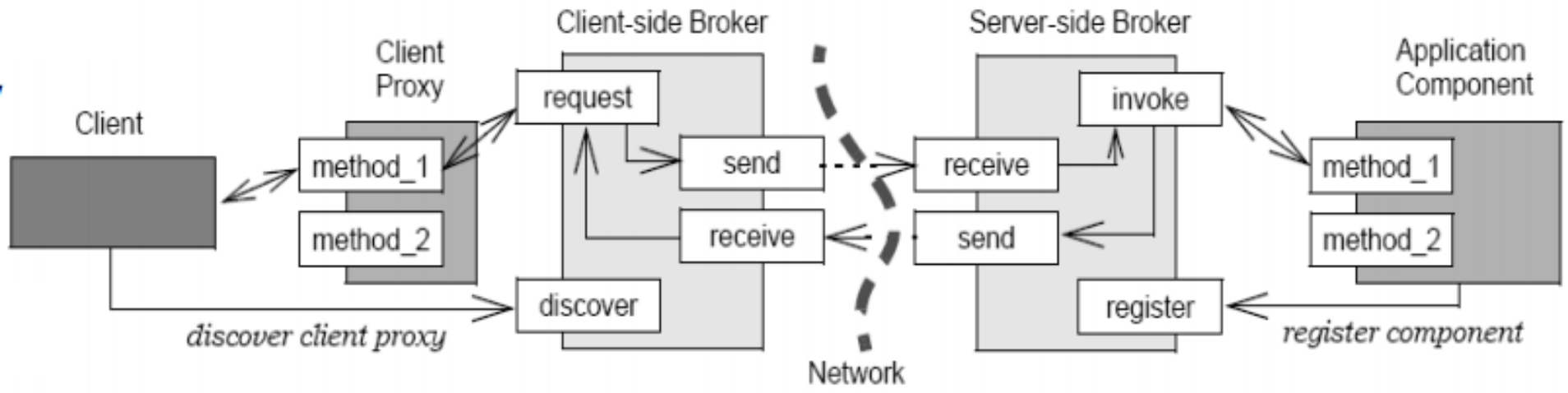
Becoming a Master Software Developer

- Bottom-line: Master software developers rely on design experiences



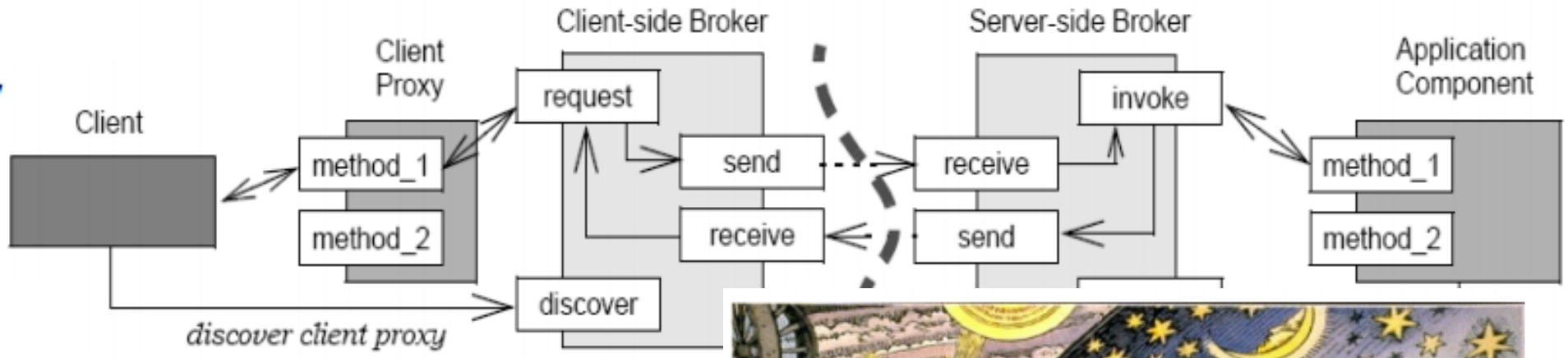
Where should design experience reside?

- Well-designed software exhibits recurring structures & behaviors that promote
  - Abstraction
  - Flexibility
  - Reuse
  - Quality
  - Modularity



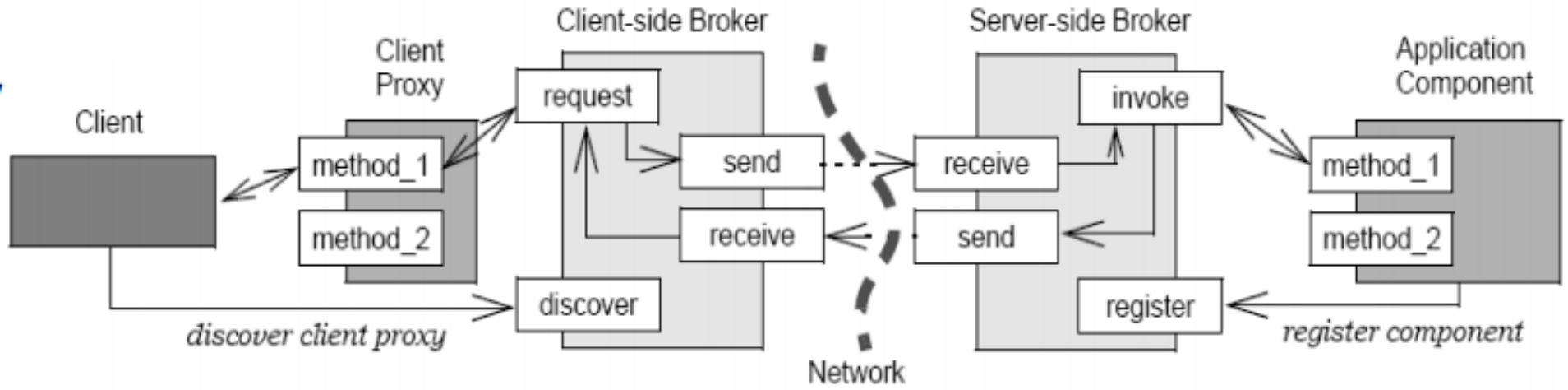
Where should design experience reside?

- Well-designed software exhibits recurring structures & behaviors that promote

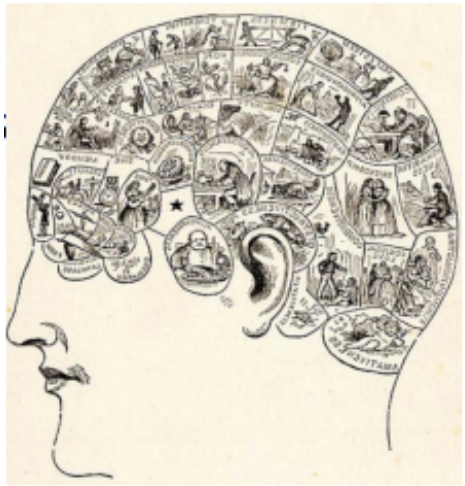


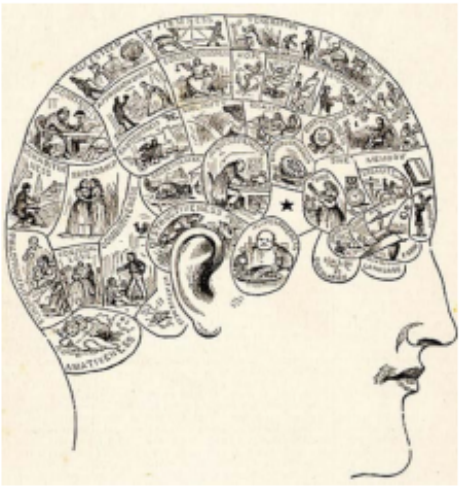
- Therein lies valuable design knowledge

Where should design experience reside?

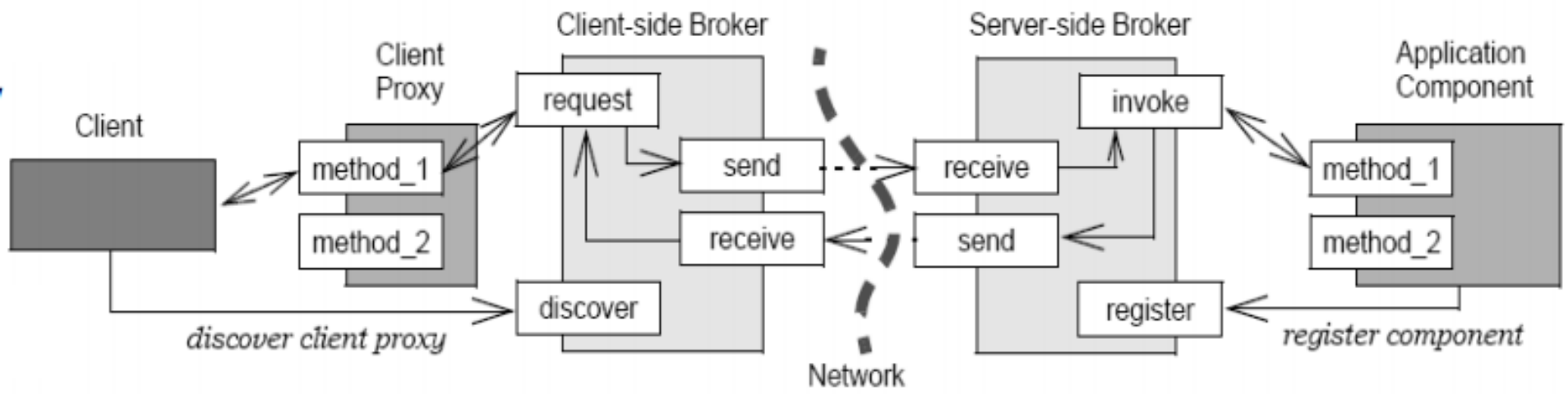


- Unfortunately, this design knowledge is typically located in:
  - The heads of the experts





Where should design experience reside?



- Unfortunately, this design knowledge is typically located in:
  - The bowels of the source code

```
public class KeyGeneratorImpl extends Service {
    private Set<UUID> keys = new HashSet<UUID>();
    private final KeyGenerator.Stub binder = new KeyGenerator.Stub() {
            public void setCallback (final KeyGeneratorCallback callback) {
                UUID id;
                synchronized (keys) {
                    do { id = UUID.randomUUID(); } while (keys.contains(id));
                    keys.add(id);
                }
                final String key = id.toString();
                try {
                    Log.d(getClass().getName(), "sending key" + key);
                    callback.sendKey(key);
                } catch (RemoteException e) { e.printStackTrace(); }
            }
    };
    public IBinder onBind(Intent intent) { return this.binder; }
}
```

Where should design experience reside?

- Unfortunately, this design knowledge is typically located in:
  - The heads of the experts
  - The bowels of the source code
- Both locations are fraught with danger!

Where should design experience reside?

- What we need is a means of extracting, documenting, conveying, applying, & preserving this design knowledge without undue time, effort, & risk!

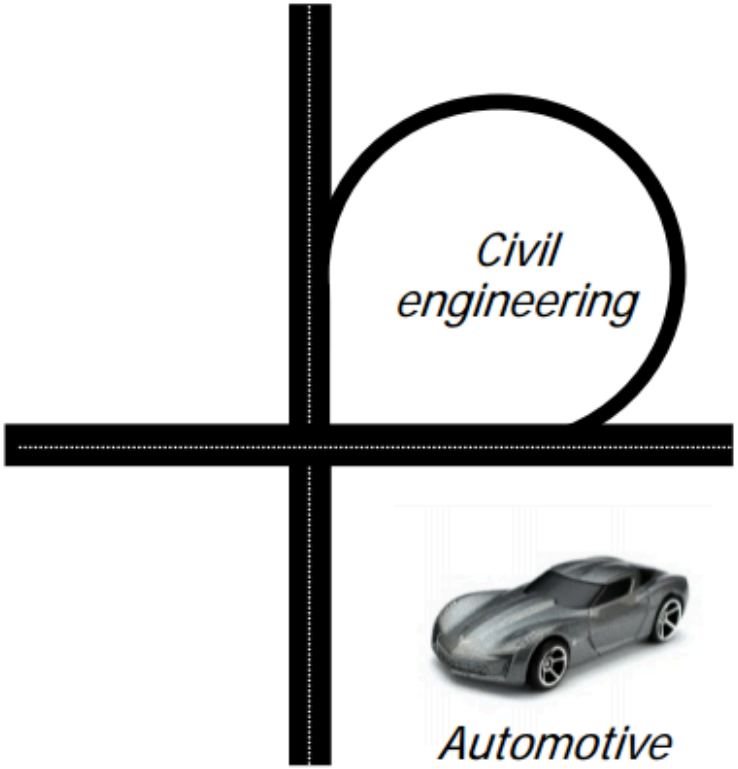
Key to Mastery: Knowledge of Software Patterns

- Describe a **solution** to a common **problem** arising within a **context**

Mobile devices

Aerospace

Electronic Trading



Automotive

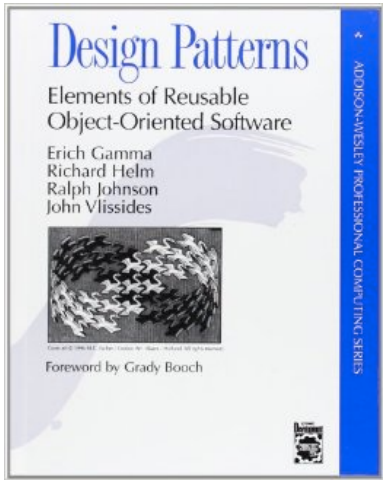
What is a Pattern? The "Alexandrian" Definition

Each pattern describes a problem
which occurs over and over again in our environment,
and then describes
the core of the solution to that problem,
in such a way that
you can use this solution a million times over,
without ever doing it the same way twice

C. Alexander, "*The Timeless Way of Building*", 1979

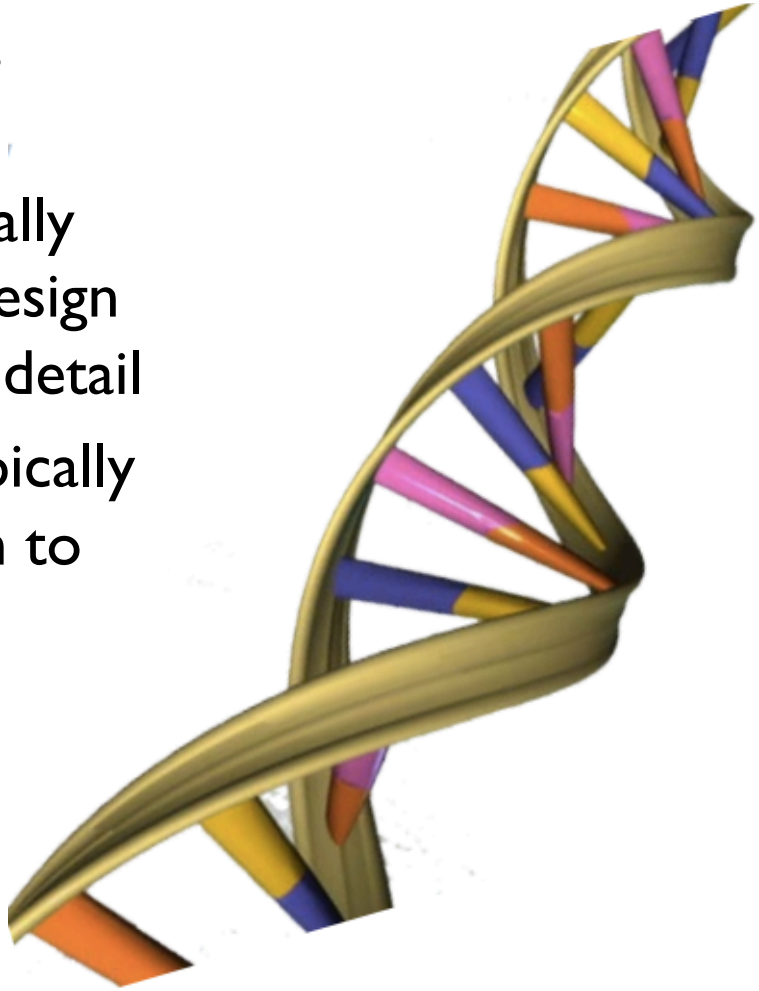
Design Patterns

- "A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context." – [GoF]



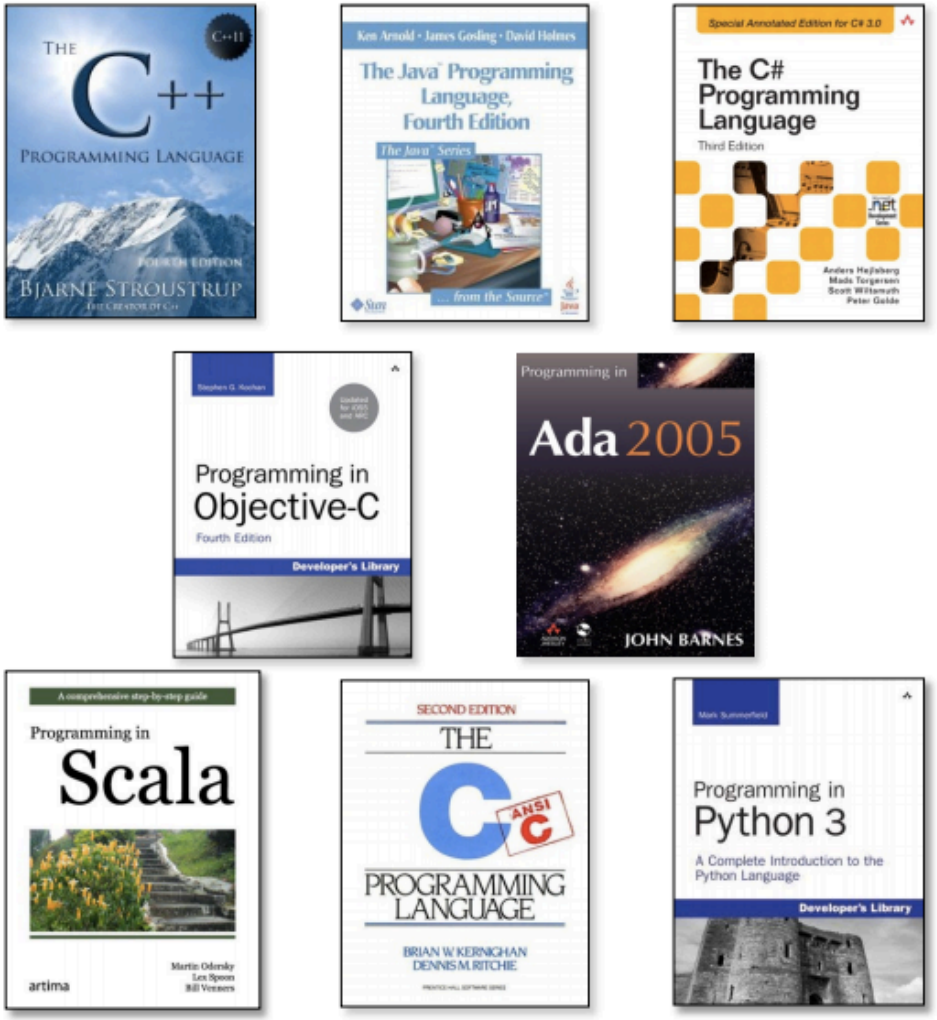
Common Characteristics of Patterns

- They describe both a thing & a process
  - The "thing" (the "what") typically means a particular high-level design outline or description of code detail
  - The "process" (the "how") typically describes the steps to perform to create the "thing"



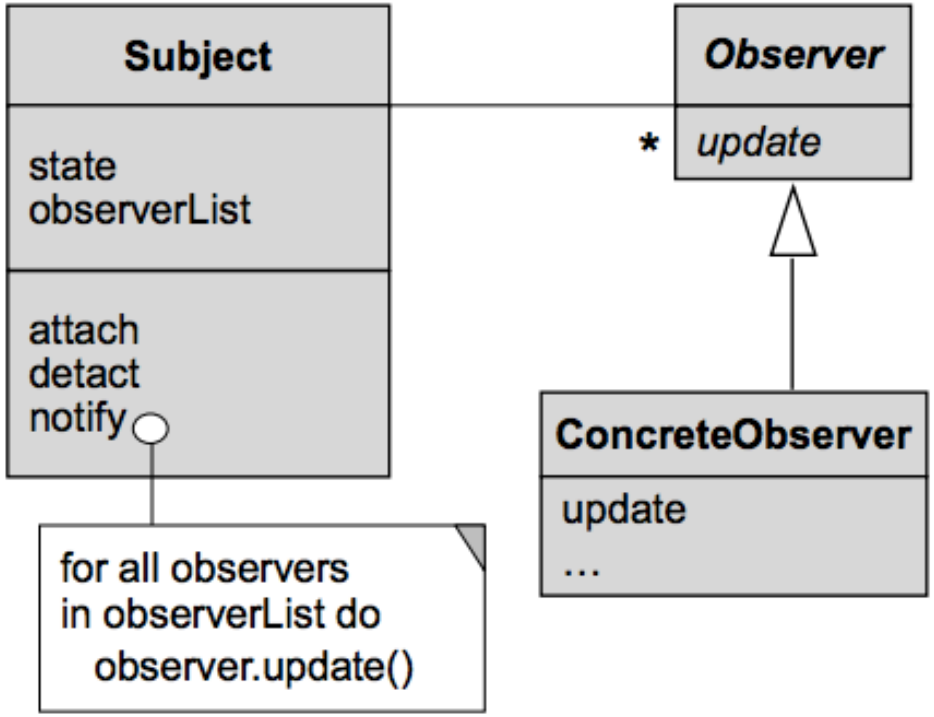
Common Characteristics of Patterns

- They can be independent of programming languages & implementation techniques



Common Characteristics of Patterns

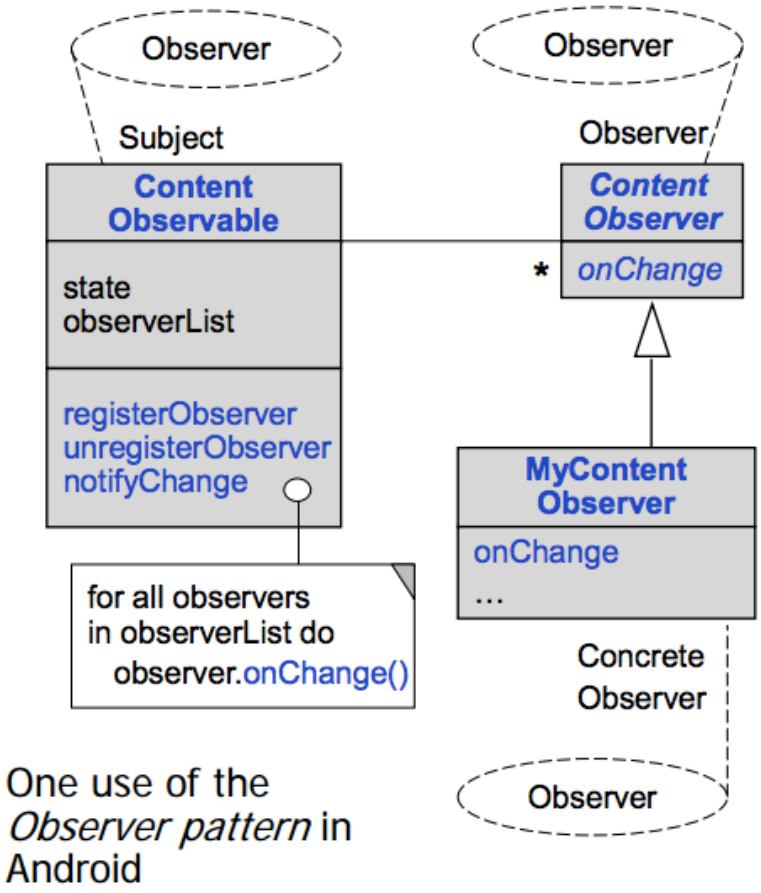
- They define "micro-architectures"
  - recurring design structure



Observer pattern

Common Characteristics of Patterns

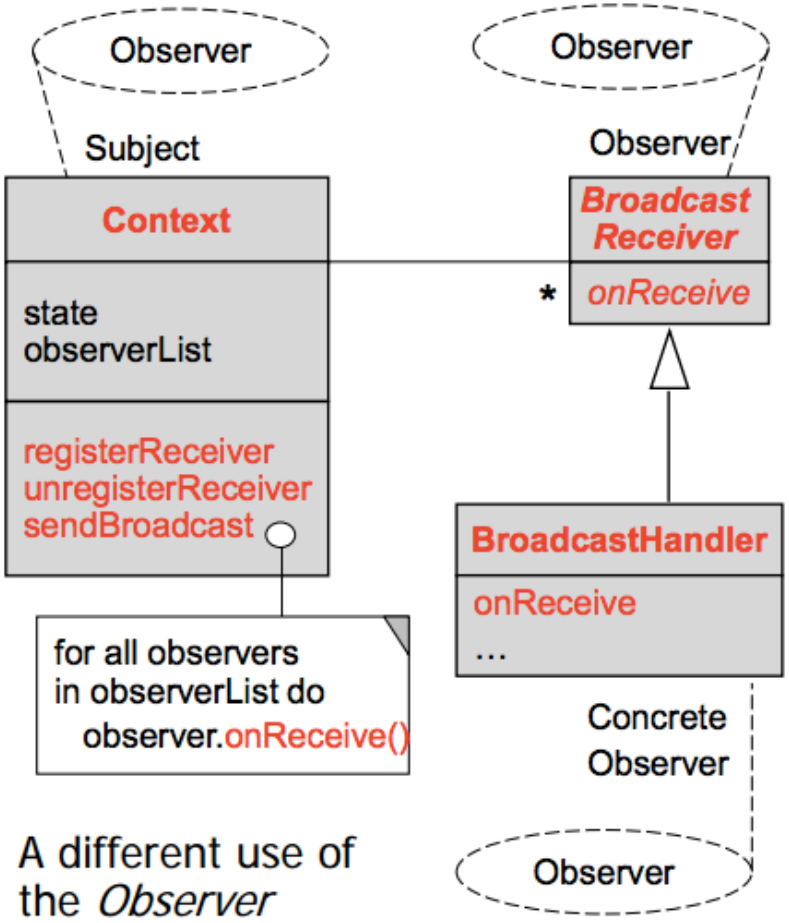
- They define "micro-architectures"
  - Certain properties may be modified for particular contexts



One use of the *Observer pattern* in Android

Common Characteristics of Patterns

- They define "micro-architectures"
  - Certain properties may be modified for particular contexts



A different use of the *Observer pattern* in Android

Common Characteristics of Patterns

- They aren't code or concrete designs, so they must be reified and applied in particular languages

```
public class EventHandler
    extends Observer {
  public void update(Observable o,
                     Object arg)
  { /*…*/ }
    …

public class EventSource
    extends Observable,
    implements Runnable {
  public void run()
  { /*…*/ notifyObservers(/*…*/); }
    …

EventSource eventSource =
        new EventSource();
EventHandler eventHandler =
        new EventHandler();
eventSource.addObserver(eventHandler);
Thread thread
            = new Thread(eventSource);
thread.start();
```

- Observer pattern in Java

Common Characteristics of Patterns

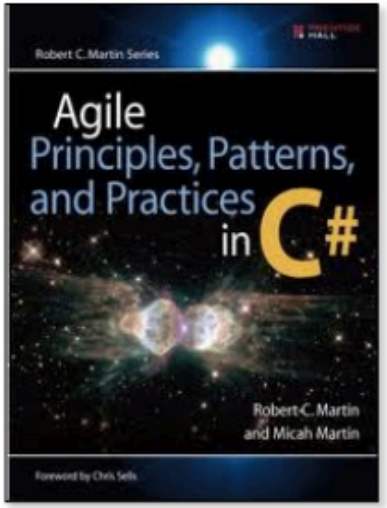
- They aren't code or concrete designs, so they must be reified and applied in particular languages

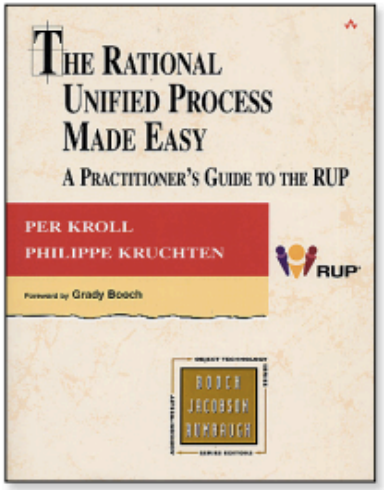
```
class Event_Handler
    : public Observer {
public:
  virtual void update(Observable o,
                      Object arg)
  { /* … */ }
    …
class Event_Source
    : public Observable,
      public ACE_Task_Base {
public:
  virtual void svc()
  { /*…*/ notify_observers(/*…*/); }
    …

Event_Source event_source;
Event_Handler event_handler;
event_source->add_observer
                    (event_handler);
Event_Task task (event_source);
task->activate();
```

- Observer pattern in C++

Common Characteristics of Patterns

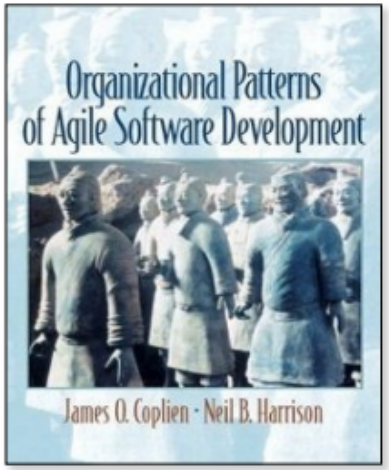
- They are not methods but can be used as an adjunct to methods
  - Rational Unified Process
  - Agile
  - Others

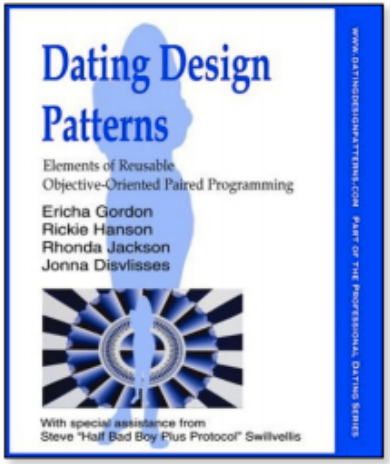




Common Characteristics of Patterns

- There are also patterns for organizing effective software development teams and navigating other complex settings





What Makes it a Pattern? A pattern must…

- …solve a problem
  - It must be useful
- …have a context
  - It must describe where the solution can be used
- …recur
  - Must be relevant in other situations; rule of three
- ... teach
  - Provide sufficient understanding to tailor the solution
- ... have a name
  - Referred consistently

GoF Form of a Design Pattern

- Pattern name and classification
- Intent
  - What does pattern do
- Also known as
  - Other known names of pattern (if any)
- Motivation
  - The design problem
- Applicability
  - Situations where pattern can be applied
- Structure
  - A graphical representation of classes in the pattern

GoF Form of a Design Pattern

- Participants
  - The classes/objects participating and their responsibilities
- Collaborations
  - Of the participants to carry out responsibilities
- Consequences
  - Trade-offs, concerns
- Implementation
  - Hints, techniques
- Sample code
  - Code fragment showing possible implementation

GoF Form of a Design Pattern

- Known uses
  - Patterns found in real systems
- Related patterns
  - Closely related patterns

Why are Patterns Important?

- "Patterns provide an incredibly dense means of efficient and effective communication between those who know the language." – [Nate Kirby]
- "Human communication is the bottleneck in software development. If patterns can help developers communicate with their clients, their customers, and each other, then patterns help fill a crucial need in our industry." – [Jim Coplien]
- "Patterns don't give you code you can drop into your application, they give you experience you can drop into your head." – [Patrick Logan]
- "Giving someone a piece of code is like giving him a fish; giving him a pattern is like teaching him to fish." – [Don Dwiggins]

Reuse Benefits

- Mature engineering disciplines have handbooks of solutions to recurring problems
  - All certified professional engineers in these fields have been trained in the contents of these handbooks
- In an experiment, teams of leading çfrom five New England medical centers observed one another's operating room practices and exchanged ideas about their most effective techniques. The result?
  - A 24% drop in their overall mortality rate for coronary bypass surgery = 74 fewer deaths than predicted

Patterns to help with design changes…

Designing for Change – Causes for Redesign (I)

- Creating an object by specifying a class explicitly
  - Commits to a particular implementation instead of an interface
  - Can complicate future changes
  - Create objects indirectly
  - Patterns: Abstract Factory, Factory Method, Prototype
- Dependence on specific operations
  - Commits to one way of satisfying a request
  - Compile-time and runtime modifications to request handling can be simplified by avoiding hard-coded requests
  - Patterns: Chain of Responsibility, Command

Causes for Redesign (II)

- Dependence on hardware and software platform
  - External OS-APIs vary
  - Design system to limit platform dependencies
  - Patterns: Abstract Factory, Bridge
- Dependence on object representations or implementations
  - Clients that know how an object is represented, stored, located, or implemented might need to be changed when object changes
  - Hide information from clients to avoid cascading changes
  - Patterns: Abstract Factory, Bridge, Memento, Proxy

Causes for Redesign (III)

- Algorithmic dependencies
  - Algorithms are often extended, optimized, and replaced during development and reuses
  - Algorithms that are likely to change should be isolated
  - Patterns: Builder, Iterator, Strategy, Template Method, Visitor
- Tight coupling
  - Leads to monolithic systems
  - Tightly coupled classes are hard to reuse in isolation
  - Patterns: Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer

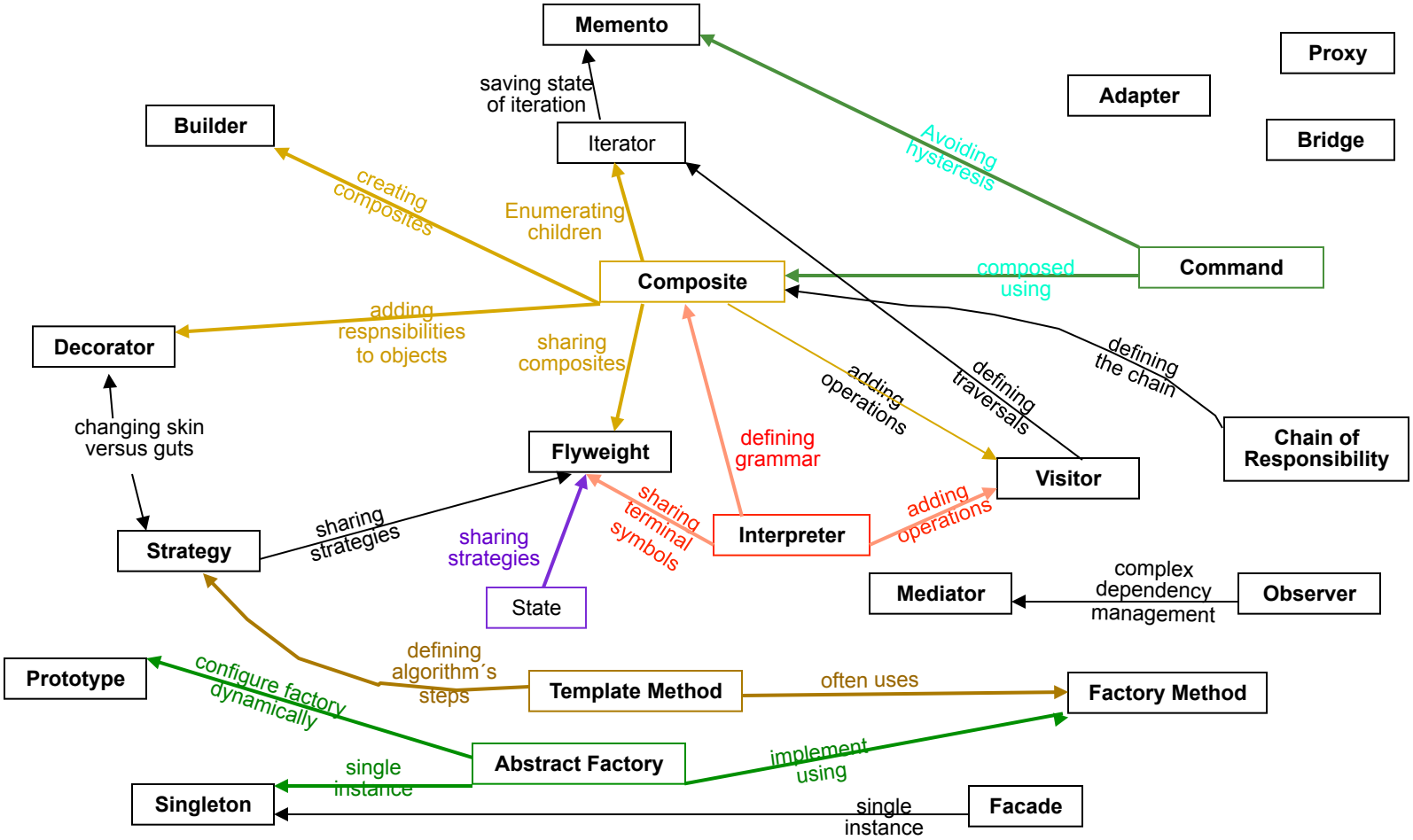
Causes for Redesign (IV)

- Extending functionality by subclassing (can be bad)
  - Requires in-depth understanding of the parent class
  - Overriding one operation might require overriding another
  - Can lead to an explosion of classes (for simple extensions)
  - Patterns: Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy
- Inability to alter classes conveniently
  - Sources not available
  - Change might require modifying lots of existing classes
  - Patterns: Adapter, Decorator, Visitor

Design Pattern Space

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Relations among Design Patterns



Drawbacks of Design Patterns

- Patterns do not lead to direct code reuse (rather, they enable *experiential* reuse)
- Patterns are deceptively simple
- Integrating patterns into a software development process is a human-intensive activity
- Teams may suffer from patterns overload

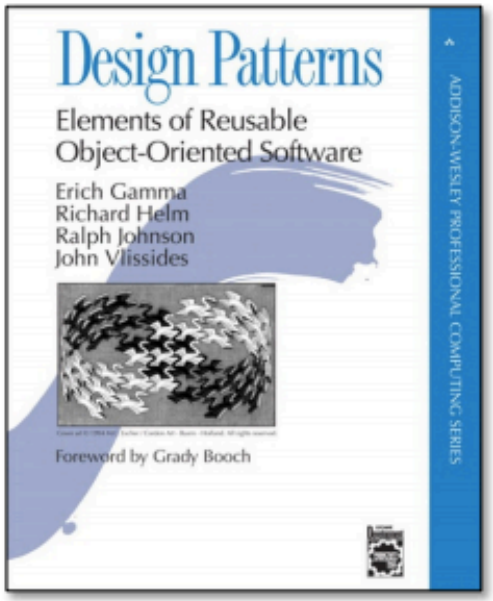
When your only tool is a hammer…

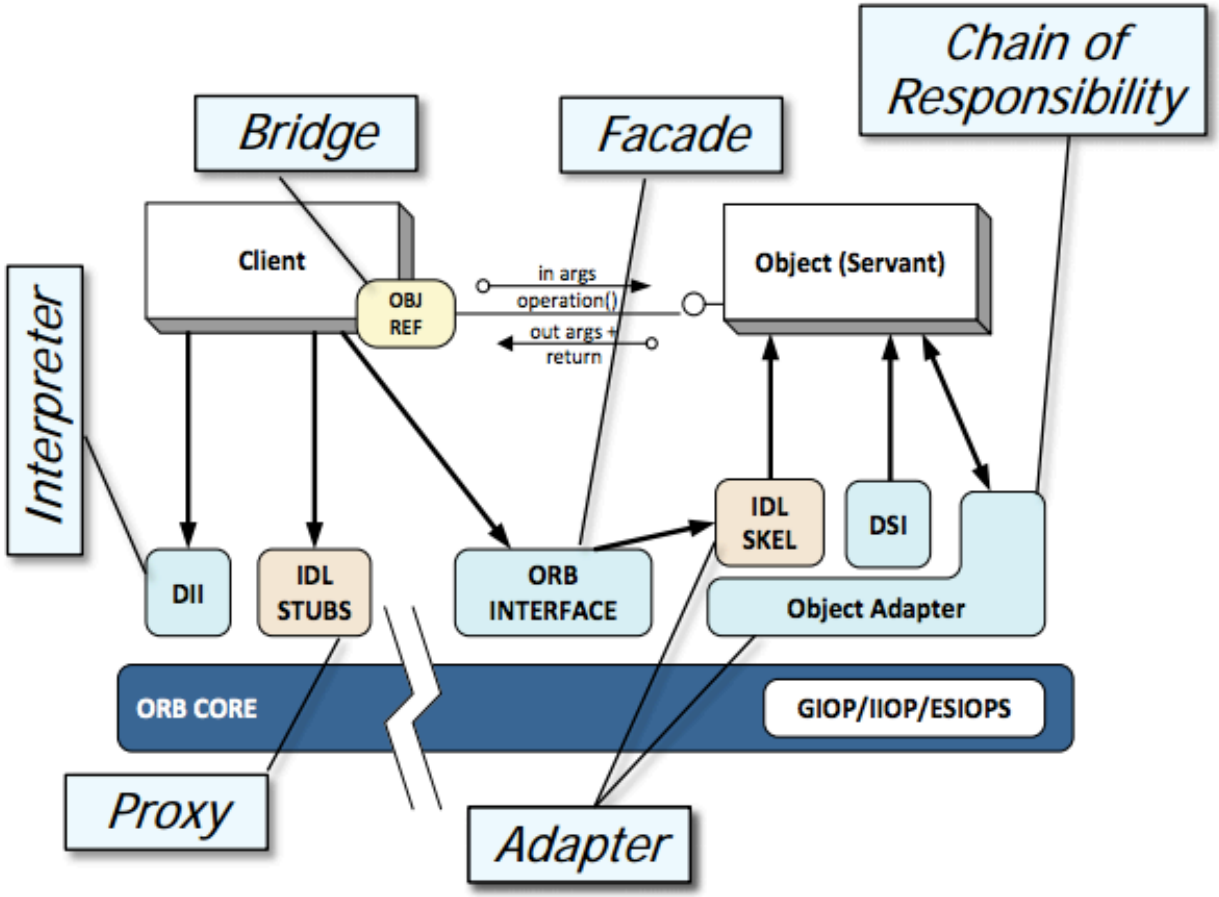
- …all the problems look like a nail

- When first learning patterns, all problems begin to look like the problem under consideration – try to avoid this!
  - Similar to someone just learning to play chess and using the same strategy everywhere – eventually you will get burned!

Variation-oriented Process for Applying Patterns

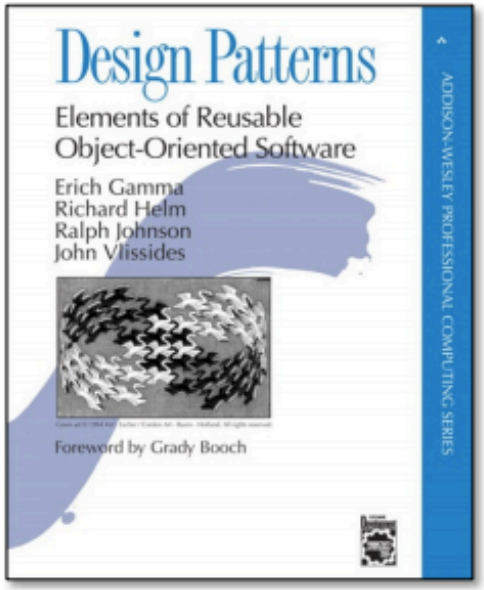
- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domains

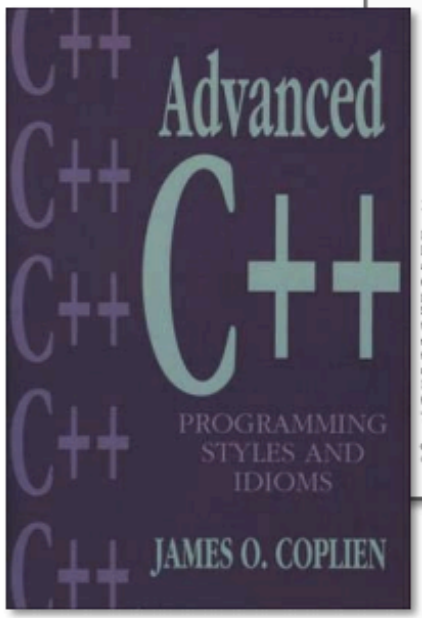


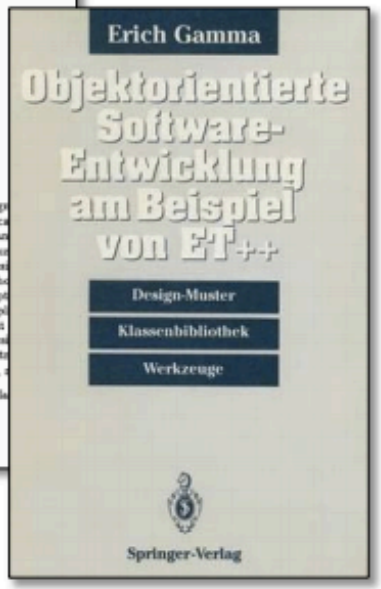


Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domains

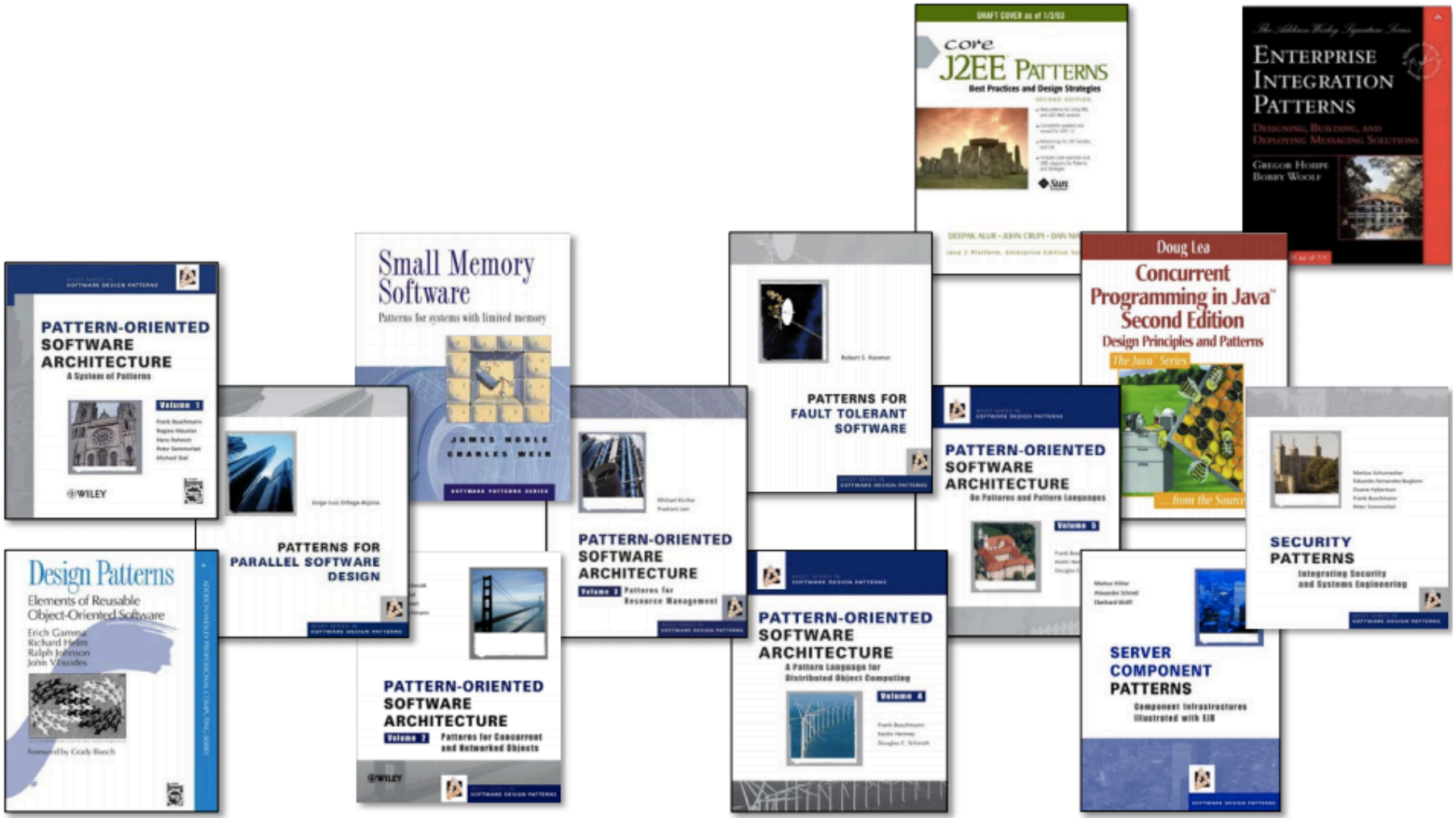






Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domains

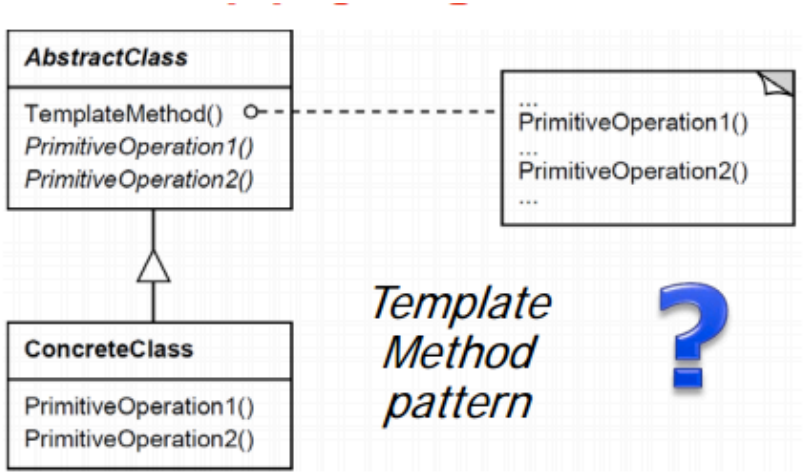


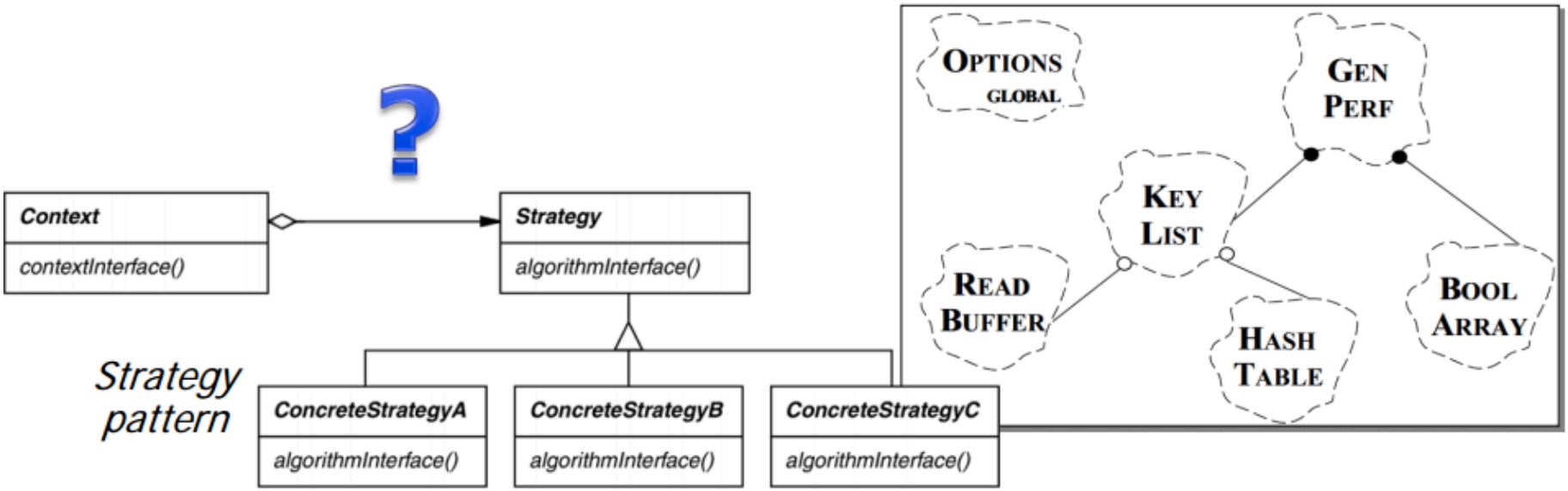
Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Evaluate trade-offs & impact of using certain patterns in their software

Variation-oriented Process for Applying Patterns

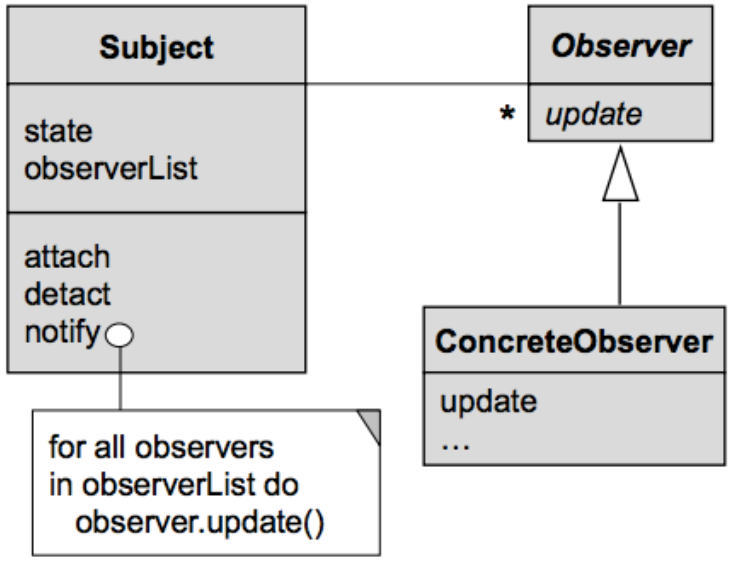
- To apply patterns successfully, software developers need to:
  - Evaluate trade-offs & impact of using certain patterns in their software





Variation-oriented Process for Applying Patterns

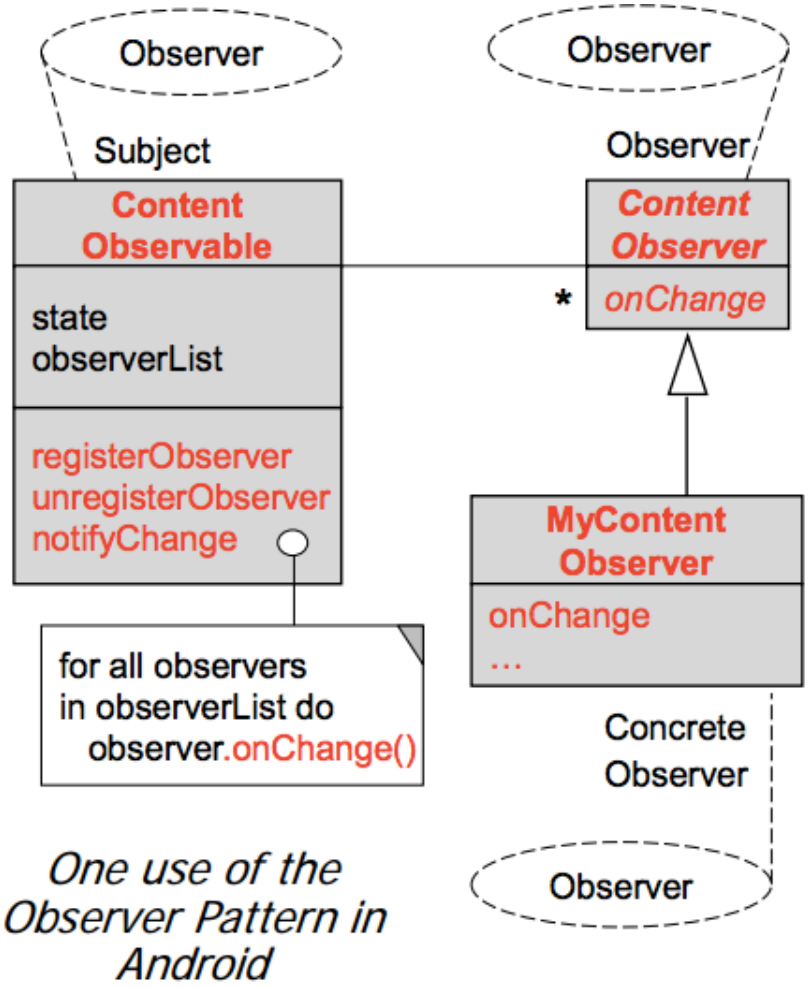
- To apply patterns successfully, software developers need to:
  - Make design and implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts



The Observer Pattern

Variation-oriented Process for Applying Patterns

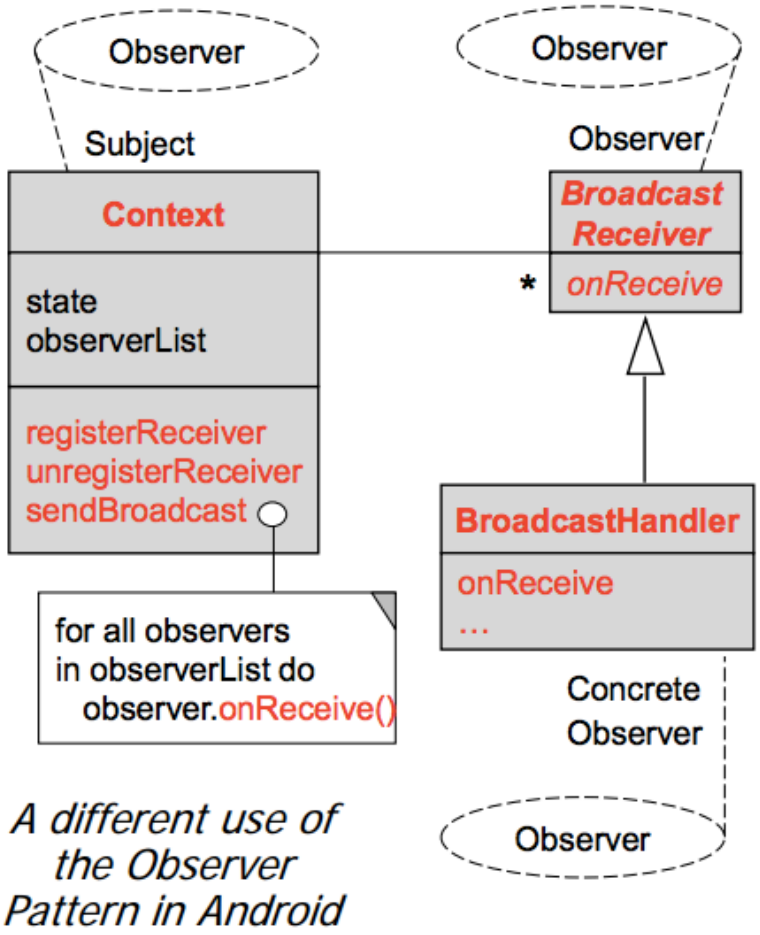
- To apply patterns successfully, software developers need to:
  - Make design and implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts



One use of the Observer Pattern in Android

Variation-oriented Process for Applying Patterns

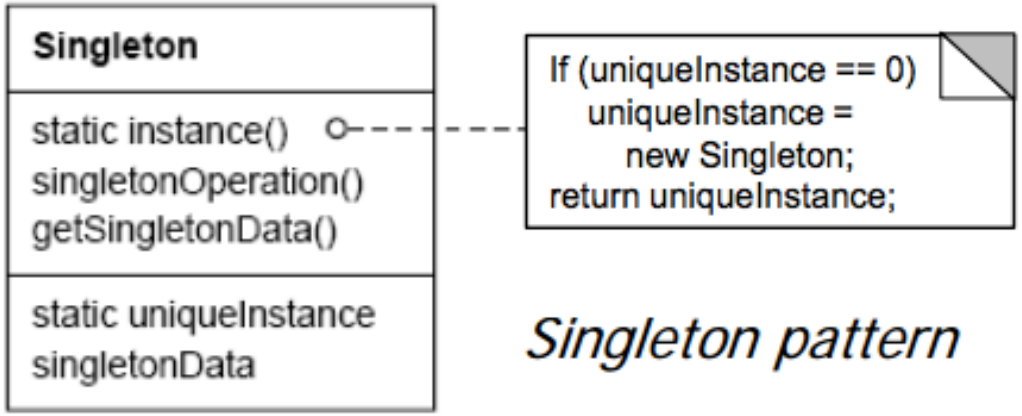
- To apply patterns successfully, software developers need to:
  - Make design and implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts



A different use of the Observer Pattern in Android

Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Make design and implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts



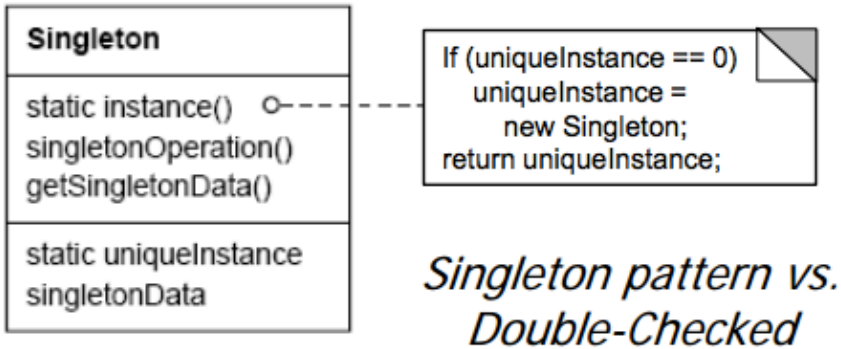
Singleton pattern

· John Vlissides, “To kill a singleton”

· sourcemaking.com/design_patterns/to_kill_a_singleton

Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Make design and implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts

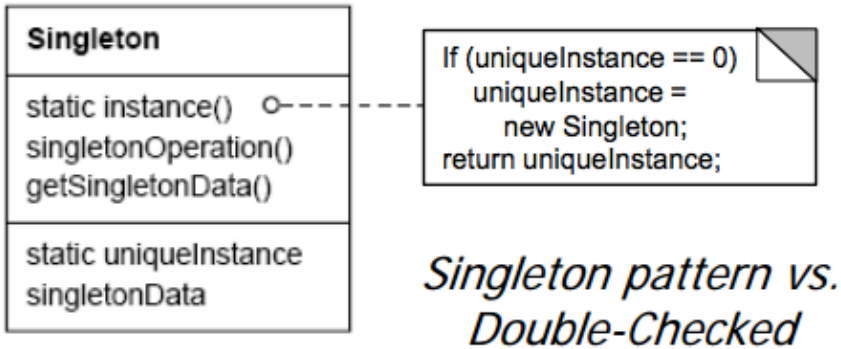


Singleton pattern vs. Double-Checked Locking Pattern

```
class Singleton {
  private static Singleton inst = null;
  public static Singleton instance() {
    Singleton result = inst;
    if (result == null) {
      inst = result = new Singleton();
    }
    return result;
  }
  ...
```

Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Make design and implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts



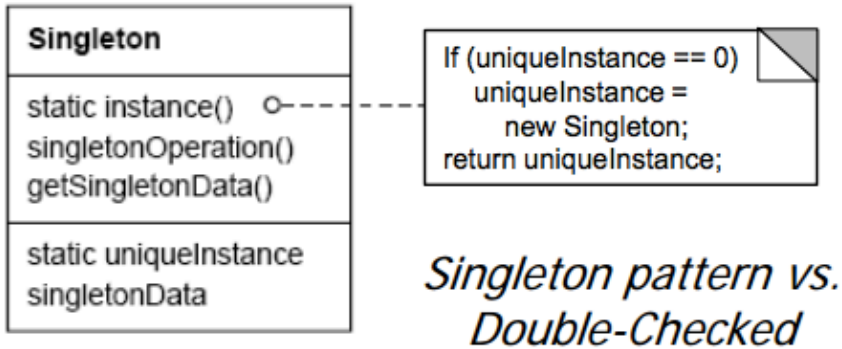
Singleton pattern vs. Double-Checked Locking Pattern

Too little synchronization

```
class Singleton {
  private static Singleton inst = null;
  public static Singleton instance() {
    Singleton result = inst;
    if (result == null) {
      inst = result = new Singleton();
    }
    return result;
  }
  ...
```

Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Make design and implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts

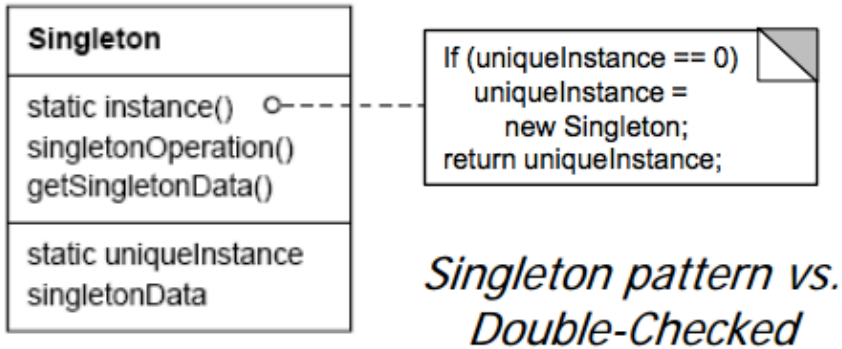


Singleton pattern vs. Double-Checked Locking Pattern

```
class Singleton {
  private static Singleton inst = null;
  public static Singleton instance() {
    synchronized(Singleton.class) {
      Singleton result = inst;
      if (result == null) {
        inst = result = new Singleton();
      }
    }
    return result;
  }
  ...
```

Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Make design and implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts



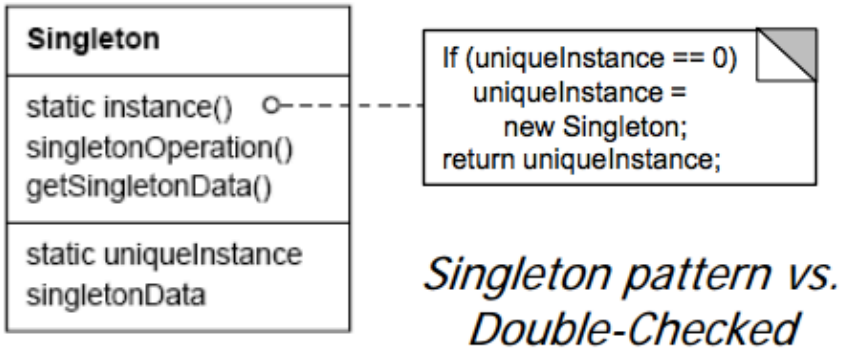
Singleton pattern vs. Double-Checked Locking Pattern

Too much synchronization

```
class Singleton {
  private static Singleton inst = null;
  public static Singleton instance() {
    synchronized(Singleton.class) {
      Singleton result = inst;
      if (result == null) {
        inst = result = new Singleton();
      }
    }
    return result;
  }
  ...
```

Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Make design and implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts



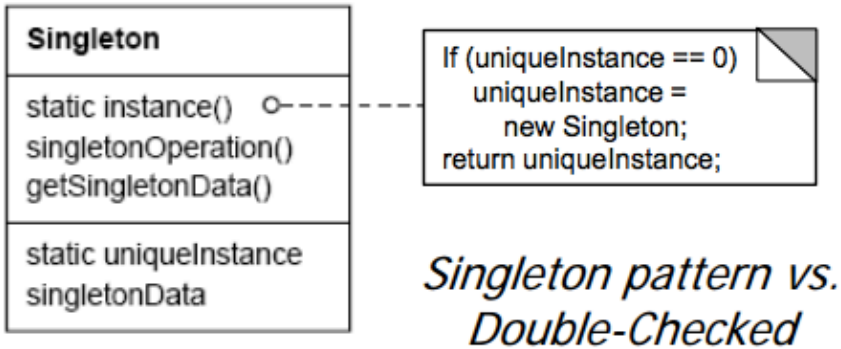
Singleton pattern vs. Double-Checked Locking Pattern

```
class Singleton {
  private static volatile Singleton
                                  inst = null;
  public static Singleton instance() {
    Singleton result = inst;
    if (result == null) {
      synchronized(Singleton.class) {
        result = inst;
        if (result == null)
        { inst = result = new Singleton(); }
      }
    }
    return result;
...
```

Just right amount of synchronization

Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Make design and implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts



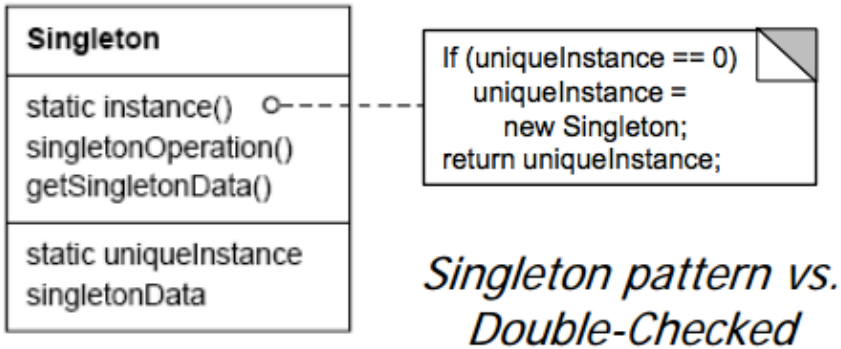
Singleton pattern vs. Double-Checked Locking Pattern

```
class Singleton {
  private static volatile Singleton
                                    inst = null;
  public static Singleton instance() {
    Singleton result = inst;
    if (result == null) {
      synchronized(Singleton.class) {
        result = inst;
        if (result == null)
        { inst = result = new Singleton(); }
      }
    }
    return result;
...
```

Only synchronizes when inst is null

Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Make design and implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts



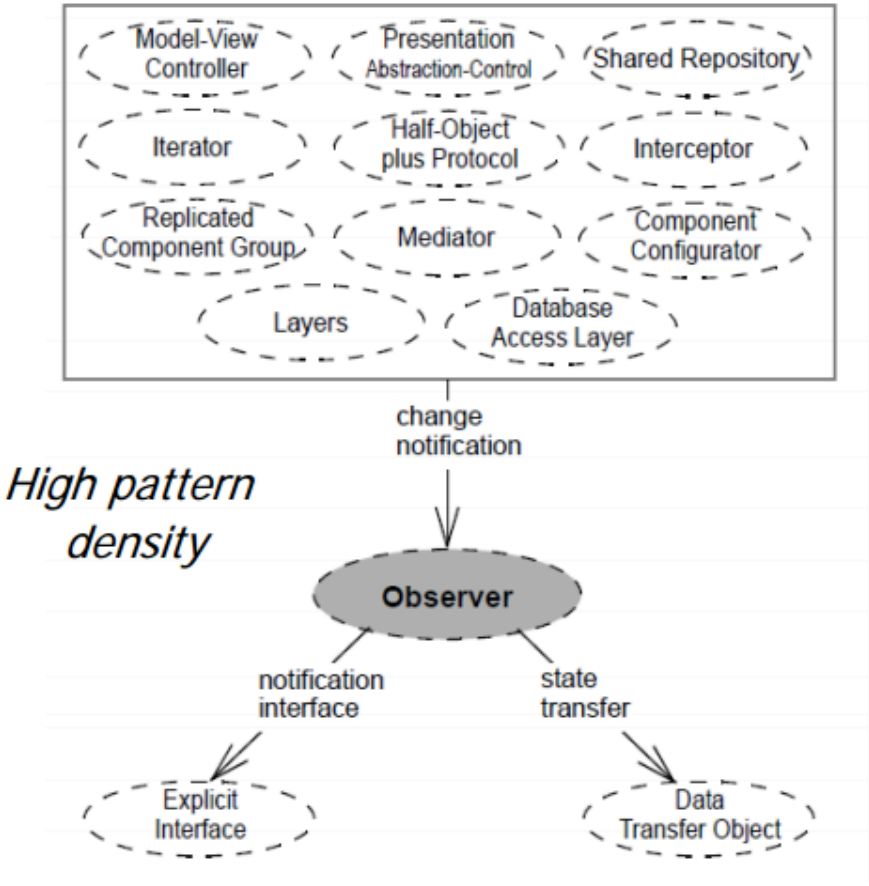
Singleton pattern vs. Double-Checked Locking Pattern

```
class Singleton {
  private static volatile Singleton
                                    inst = null;
  public static Singleton instance() {
    Singleton result = inst;
    if (result == null) {
      synchronized(Singleton.class) {
        result = inst;
        if (result == null)
        { inst = result = new Singleton(); }
      }
    }
    return result;
...
```

No synchronization after inst is created

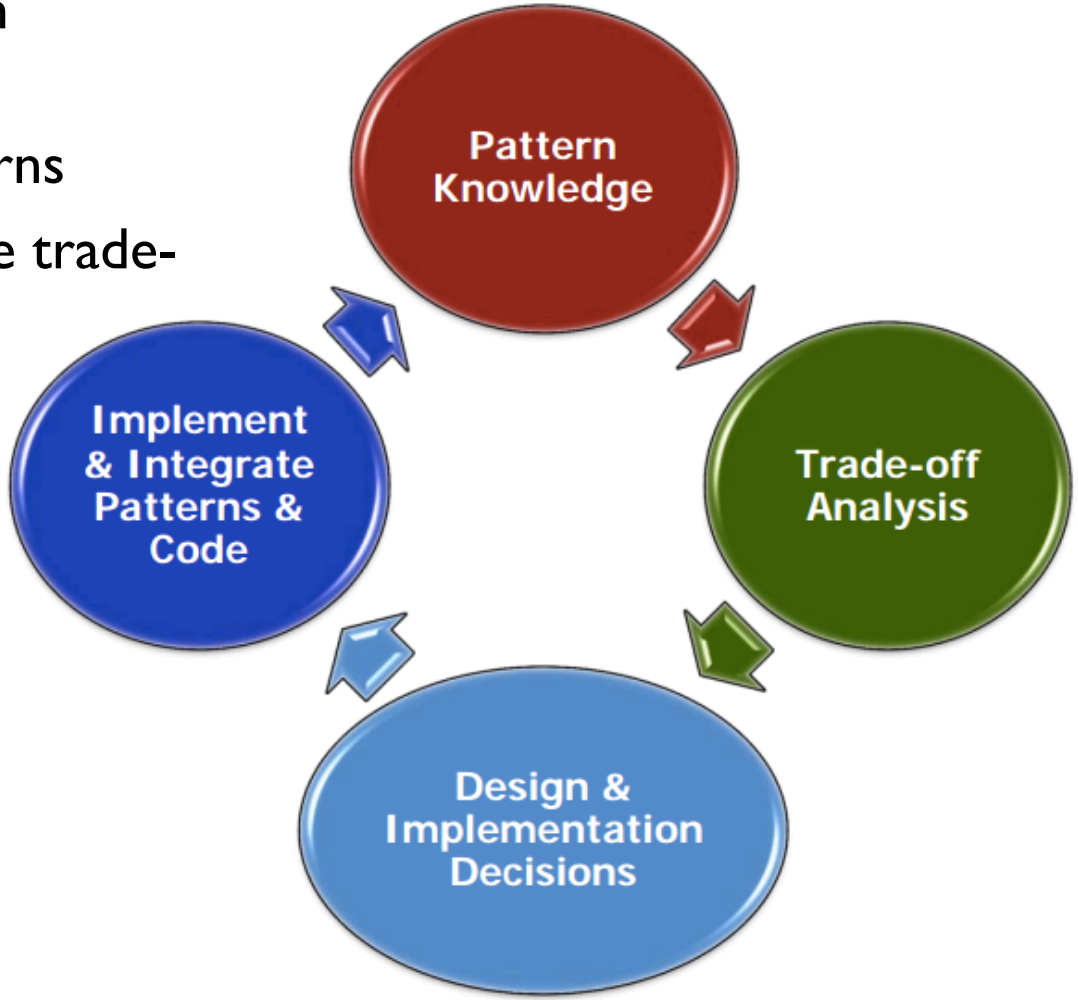
Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Combine with other patterns & implement/integrate with code



Summary

- Patterns support a variation-oriented design process
  - Determine which design elements can vary
  - Identify applicable patterns
  - Vary patterns & evaluate trade-offs
  - Repeat …



Summary

- Seek generality, but don't brand everything as a pattern

Summary

- Articulate specific benefits and demonstrate general applicability
  - Find three different existing examples from code other than yours!

Rule of Three

More Pattern Information

- Robert C. Martin's Chess Analogy
  - http://www.cs.wustl.edu/~schmidt/cs242/learning.html
- John Vlissides' "Top 10 Misconceptions"
  - http://www.research.ibm.com/designpatterns/pubs/top10misc.html
- Seven Habits of Successful Pattern Writers
  - http://www.research.ibm.com/designpatterns/pubs/7habits.html
- Brad Appleton's "Patterns in a Nutshell"
  - http://www.cmcrossroads.com/bradapp/docs/patterns-nutshell.html
- Mike Duell's non-software examples
  - http://www.cours.polymtl.ca/inf3700/divers/nonSoftwareExample/patexamples.html