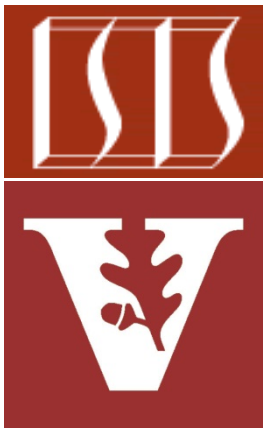
Overview of Patterns

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

Institute for Software
Integrated Systems

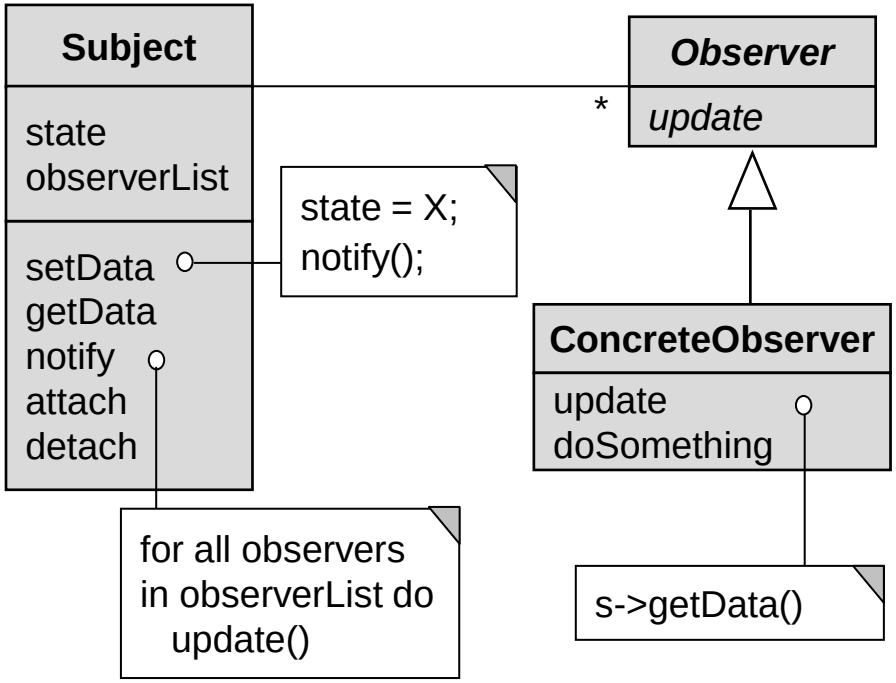
Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Module

- Motivate the importance of design experience & leveraging recurring design structure to become a master software developer

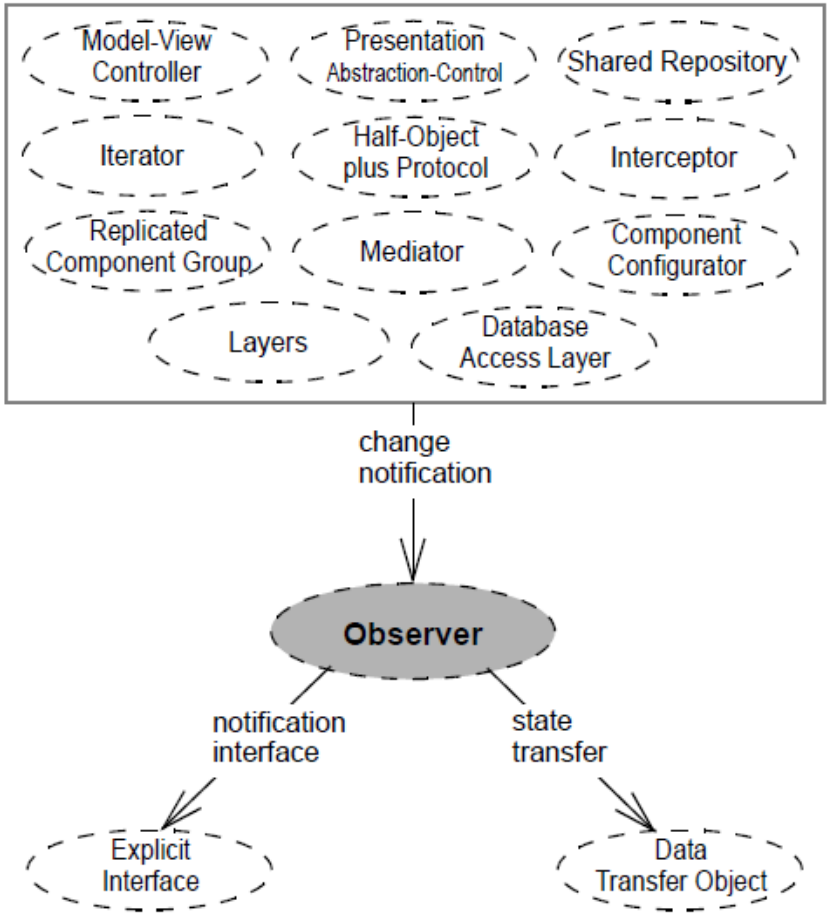
Topics Covered in this Module

- Motivate the importance of design experience & leveraging recurring design structure to become a master software developer
- Introduce patterns as a means of capturing & applying proven design experience that makes software more robust to change



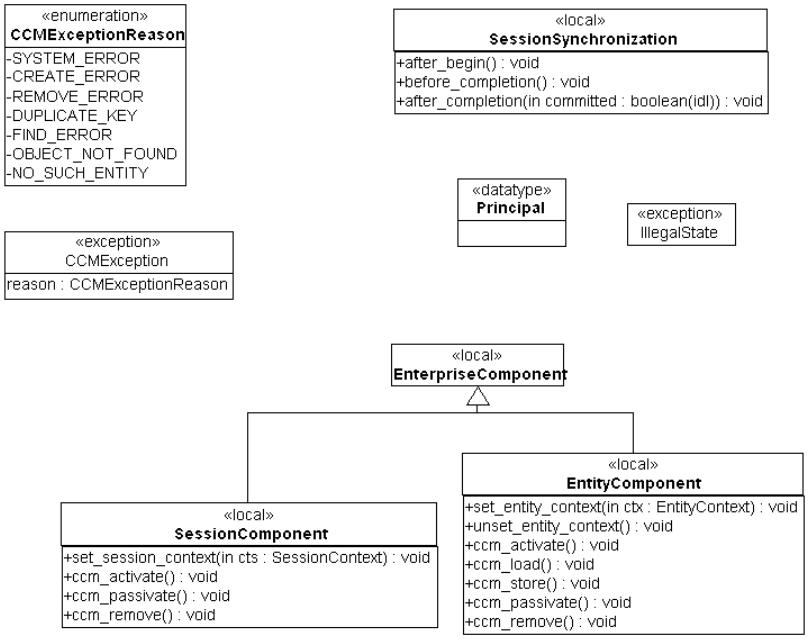
Topics Covered in this Module

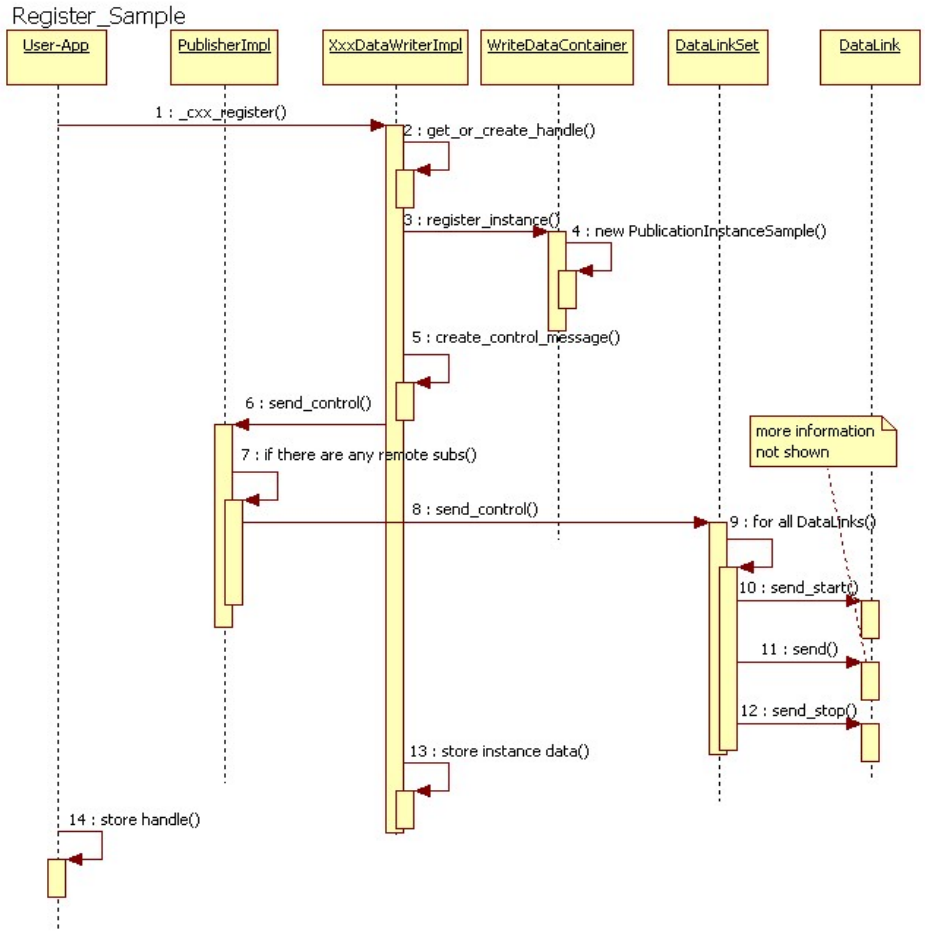
- Motivate the importance of design experience & leveraging recurring design structure to become a master software developer
- Introduce patterns as a means of capturing & applying proven design experience that makes software more robust to change
- Describe a process for successfully applying patterns to software development projects



Becoming a Master Software Developer

- Software methods emphasize design notations, such as UML
  - Fine for specification & documentation



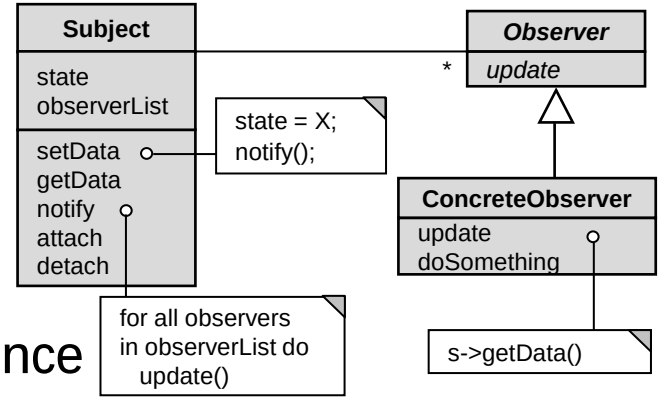


Becoming a Master Software Developer

- Software methods emphasize design notations, such as UML
  - Fine for specification & documentation
- But software is more than drawing diagrams
  - Good draftsmen are not necessarily good architects!

Becoming a Master Software Developer

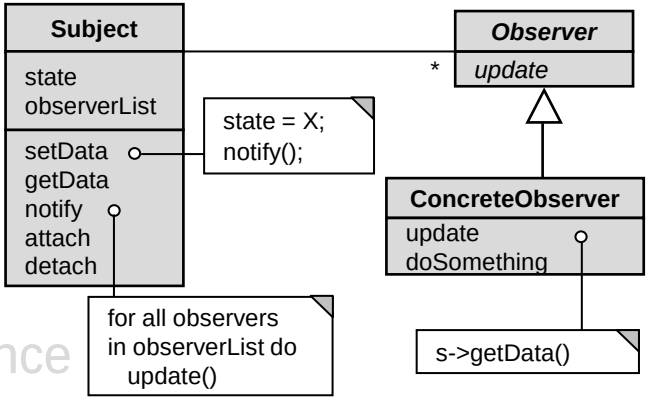
- Software methods emphasize design notations, such as UML
  - Fine for specification & documentation
- But software is more than drawing diagrams
  - Good draftsmen are not necessarily good architects!



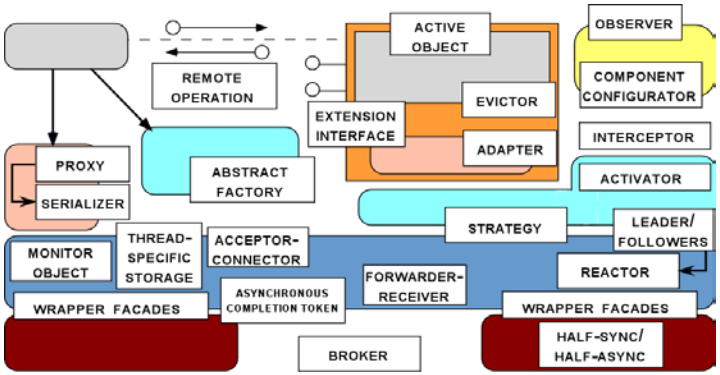
- Good software developers rely on design experience
  - At least as important as knowledge of programming languages

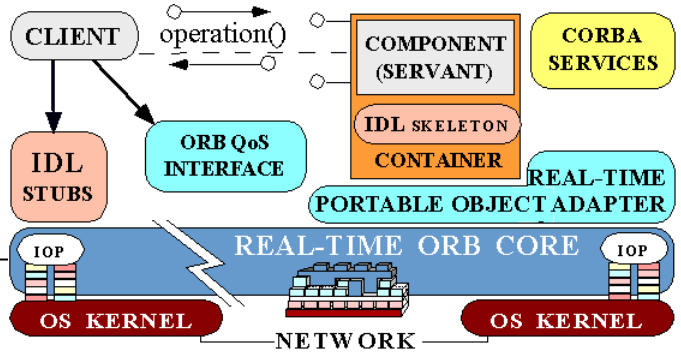
Becoming a Master Software Developer

- Software methods emphasize design notations, such as UML
  - Fine for specification & documentation
- But software is more than drawing diagrams
  - Good draftsmen are not necessarily good architects!
- Good software developers rely on design experience
  - At least as important as knowledge of programming languages



- Design experience can be codified via *design & code* reuse
  - **Design reuse**: Match problem(s) to design experience & best practices
  - **Code reuse**: Reify proven designs within a particular set of domains & development environments

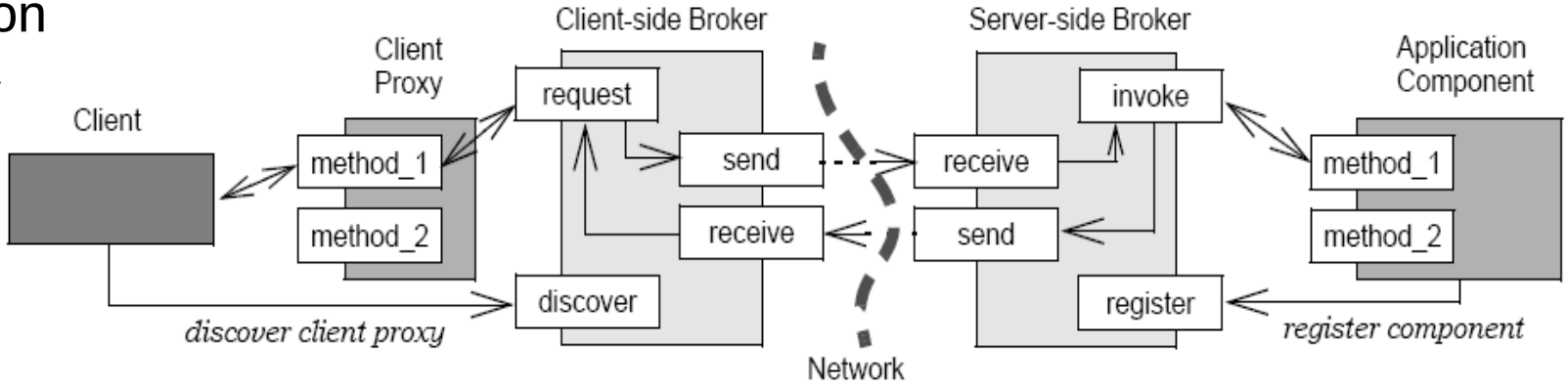




Leveraging Recurring Design Structures

Well-designed software systems exhibit recurring structures that promote

- Abstraction
- Flexibility
- Reuse
- Quality
- Elegance
- Modularity



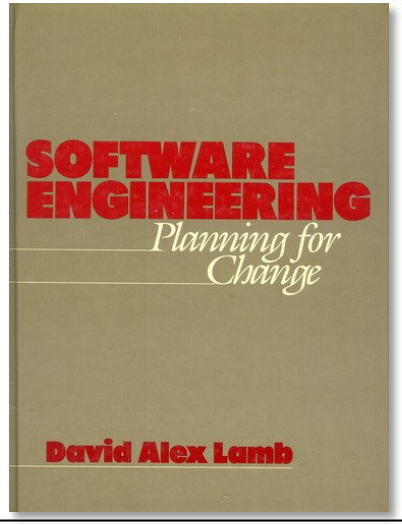
Therein lies valuable design knowledge

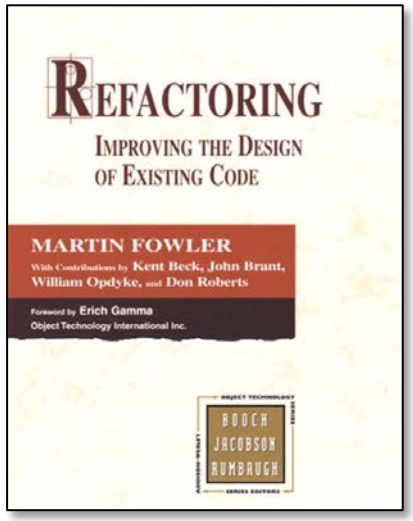
Challenge: extracting, documenting, communicating, applying, & preserving this knowledge without undue time, effort, & risk *in the face of continual change to the software!*

Making Software that's Robust to Changes

- Change is intrinsic to software development as requirements, use-cases, technologies, platforms, & quality goals evolve
- Robustness to change means that software can be modified locally without endangering overall structure
  - It is a quality that reflects ease of evolution & maintenance costs

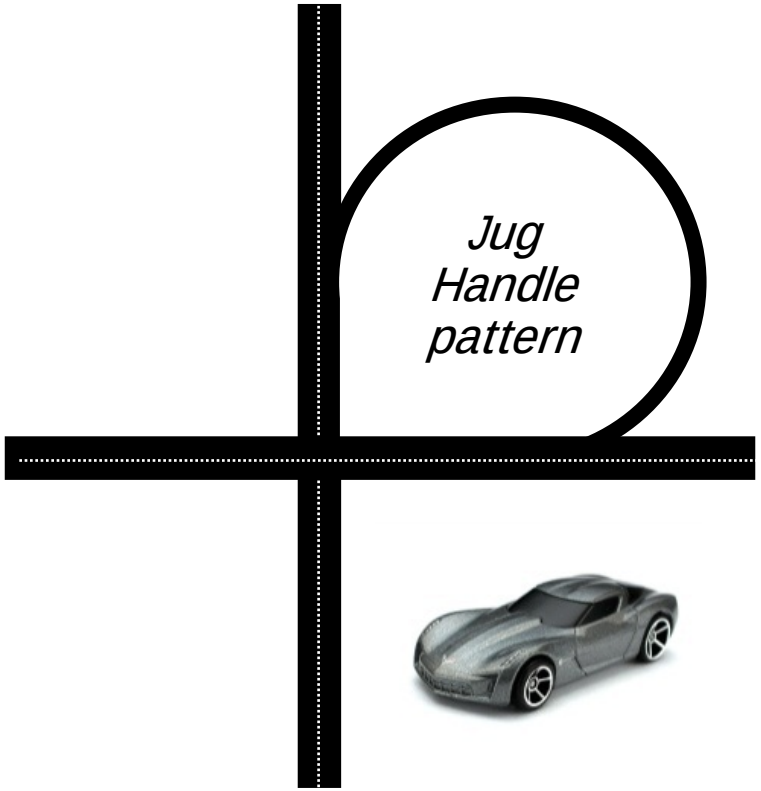
What is needed is a means to address particular design aspects of software & allow controlled variation & evolution of these aspects





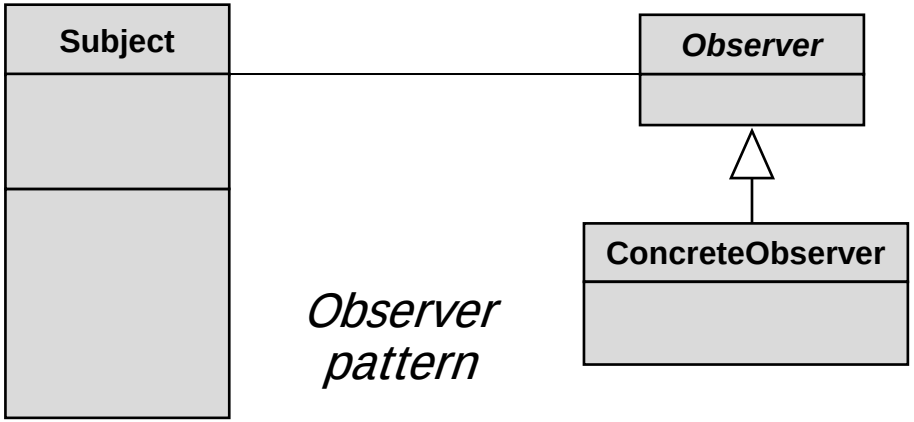
Key to Mastery: *Knowledge of Software Patterns*

- A patterns describes solution(s) to common problem(s) arising within a context by
  - Naming a recurring design structure



Key to Mastery: *Knowledge of Software Patterns*

- A patterns describes solution(s) to common problem(s) arising within a context by
  - Naming a recurring design structure

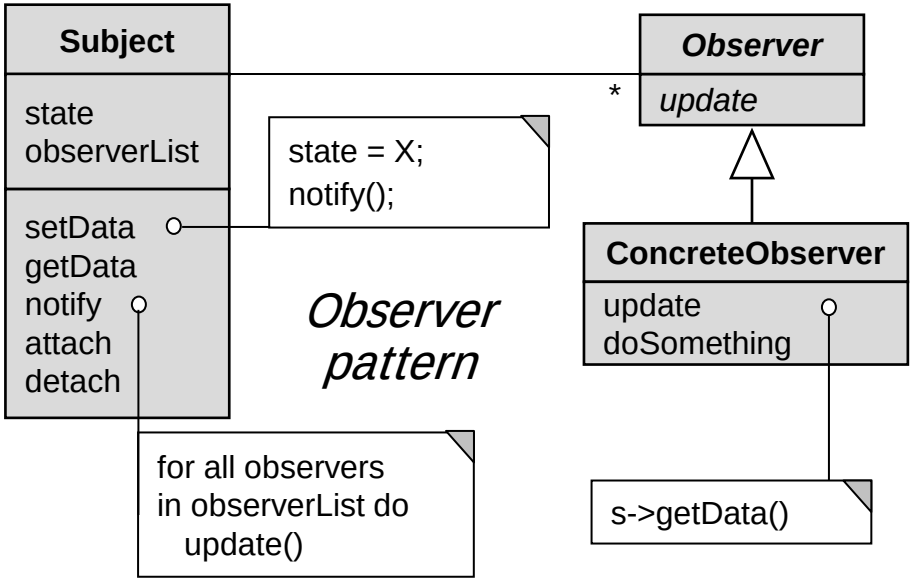


Observer pattern

"define a one-to-many dependency between objects so that when one object changes state, all dependents are notified & updated"

Key to Mastery: *Knowledge of Software Patterns*

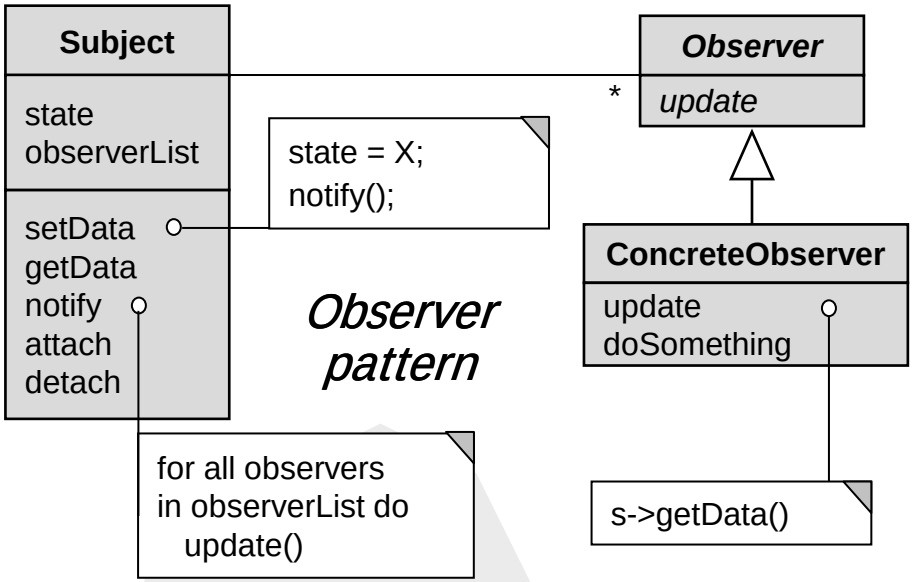
- A patterns describes solution(s) to common problem(s) arising within a context by
  - Naming a recurring design structure
  - Specifying design structure explicitly by identifying key classes/objects*
    - Roles & relationships
    - Dependencies
    - Interactions
    - Conventions



*Interpret “class” & “object” loosely: patterns are for more than OO languages!

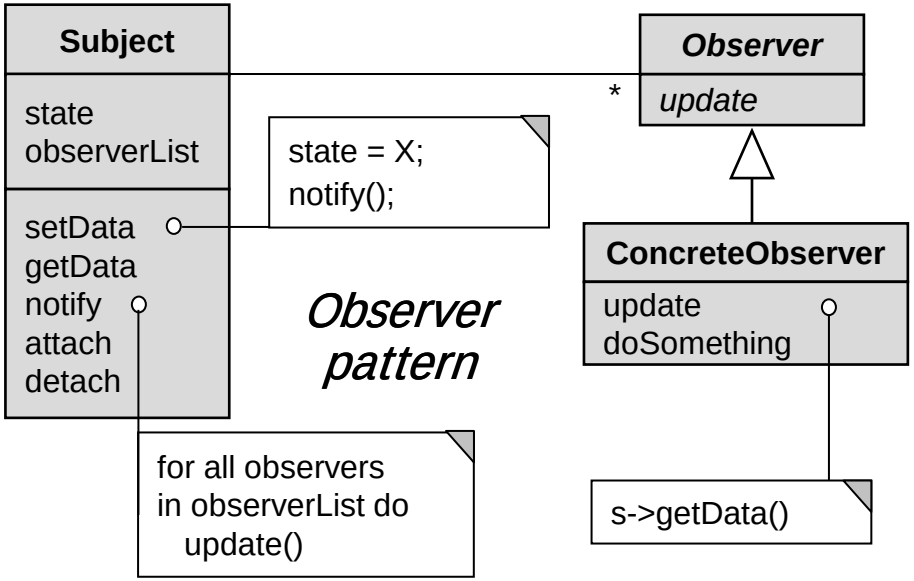
Key to Mastery: *Knowledge of Software Patterns*

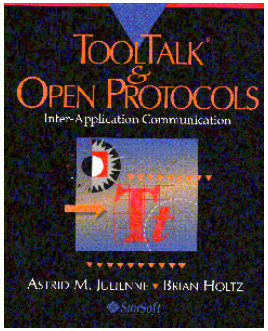
- A patterns describes solution(s) to common problem(s) arising within a context by
  - Naming a recurring design structure
  - Specifying design structure explicitly by identifying key classes/objects
    - Roles & relationships
    - Dependencies
    - Interactions
    - Conventions
- Abstracting from concrete design elements, e.g., problem domain, programming language, vendor, etc.

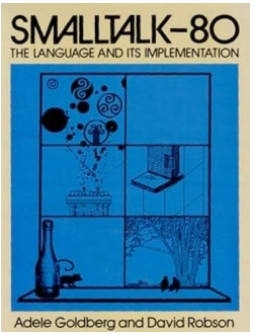


Key to Mastery: *Knowledge of Software Patterns*

- A patterns describes solution(s) to common problem(s) arising within a context by
  - Naming a recurring design structure
  - Specifying design structure explicitly by identifying key classes/objects
    - Dependencies
    - Roles & Relationships
    - Interactions
    - Conventions
  - Abstracting from concrete design elements, e.g., problem domain, programming language, vendor, etc.
- Distilling & codifying knowledge gleaned from successful design experience

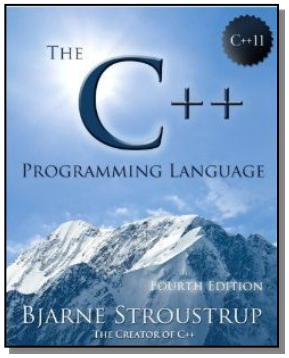


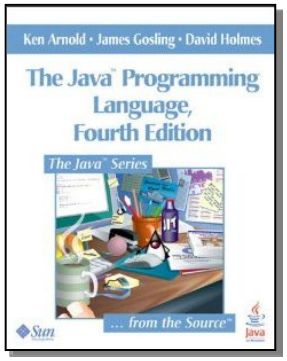


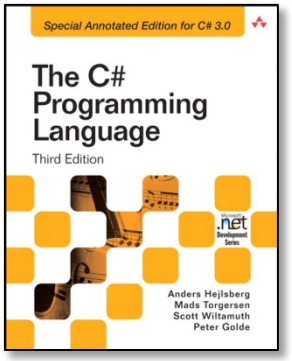


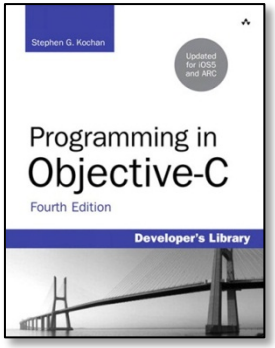
Common Characteristics of Patterns

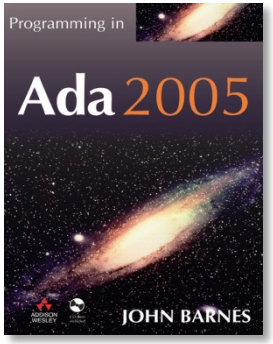
- They are independent of programming languages & implementation techniques

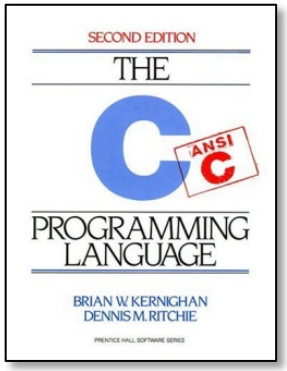






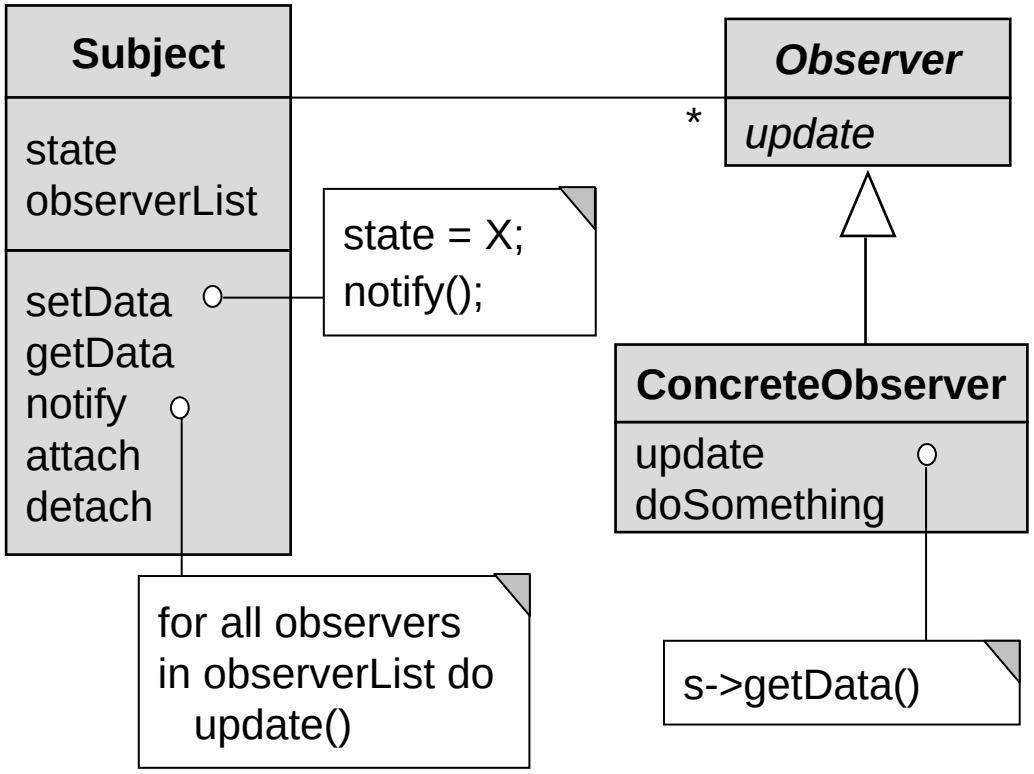






Common Characteristics of Patterns

- They are independent of programming languages & implementation techniques
- They define "micro-architectures"
  - i.e., a "society of objects"



Common Characteristics of Patterns

- They are independent of programming languages & implementation techniques
- They define "micro-architectures"
  - i.e., a "society of objects"
- They aren't code or (concrete) designs, so they must be reified & applied in particular languages

Observer pattern in Java

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

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

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

Common Characteristics of Patterns

- They are independent of programming languages & implementation techniques
- They define "micro-architectures"
  - i.e., a "society of objects"
- They aren't code or (concrete) designs, so they must be reified & applied in particular languages

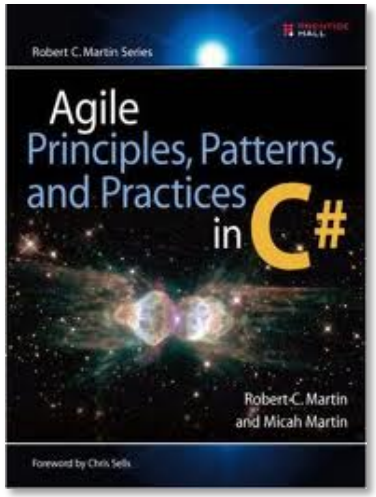
Observer pattern in C++

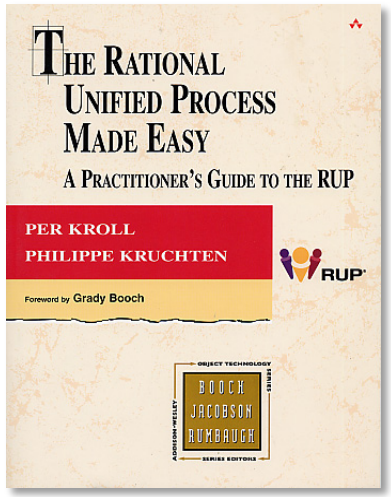
(uses the GoF Bridge pattern with reference counting to simplify memory management & ensure exception-safe semantics)

```
class Event_Handler : public Observer { public:
  virtual void update(Observable obj,
                      Object arg)
  { /* ... */ }
  ...

class Event_Source : public Observable,
                     public Runnable { public:
  virtual void run()
  { /* ... */ notify_observers(/* ... */); }
  ...

Event_Source event_source =
        new Event_Source_Impl;
Event_Handler event_handler =
        new Event_Handler_Impl;
event_source->add_observer(event_handler);
Thread thread = new Thread(event_source);
thread->start();
...
```

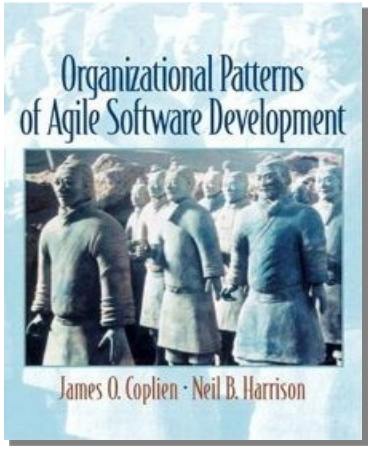
Common Characteristics of Patterns

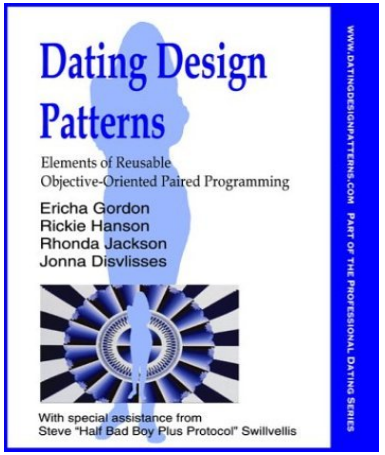
- They are independent of programming languages & implementation techniques
- They define "micro-architectures"
  - i.e., a "society of objects"
- They aren't code or (concrete) designs, so they must be reified & applied in particular languages





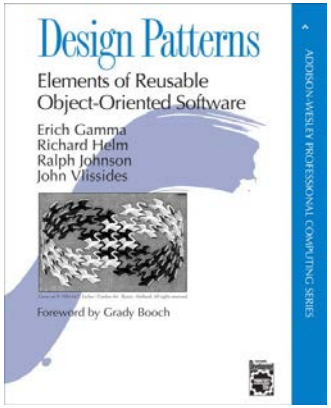
- They are not methods, but can be used an adjunct to other methods
  - e.g., Rational Unified Process, Agile, etc.
  - There are also patterns for organizing effective software development teams & navigating other complex settings

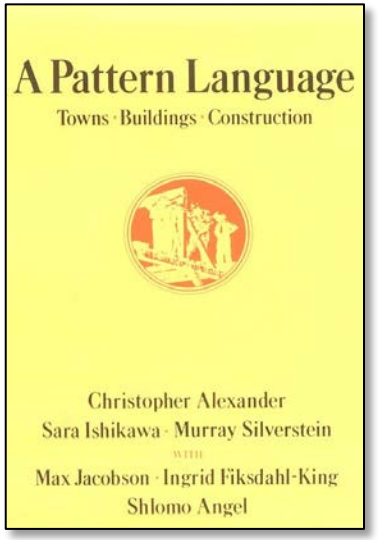


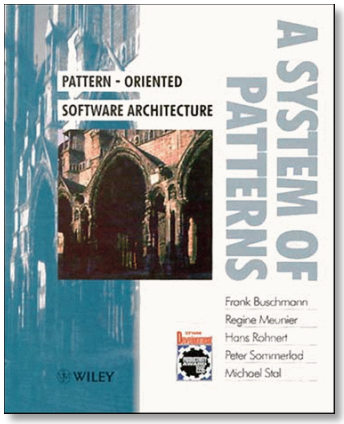


Common Parts of a Pattern Description

- **Name** & statement of pattern **intent**
- **Problem** addressed by pattern
  - Including “forces” & “applicability”
- **Solution**
  - Visual & textual descriptions of pattern structure & dynamics
- **Consequences**
  - Pros & cons of applying the pattern
- **Implementation** guidance
  - May include source code examples
- **Known uses**
  - “rule of three”
- **Related patterns**
  - Tradeoffs between alternative patterns







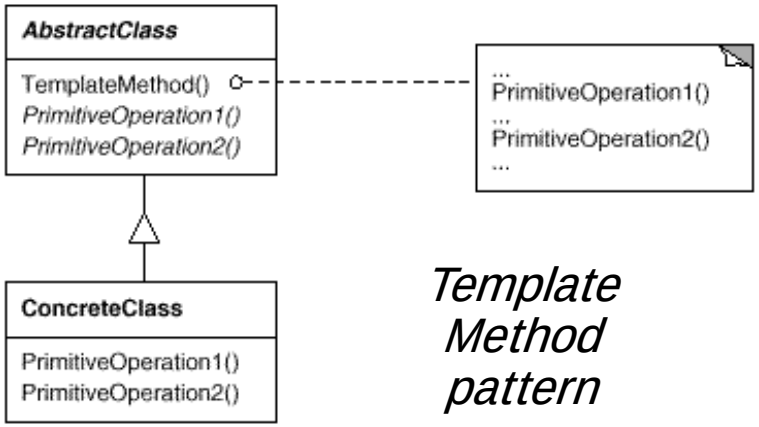
See c2.com/cgi/wiki?PatternForms for more info on pattern forms

Process for Applying Patterns

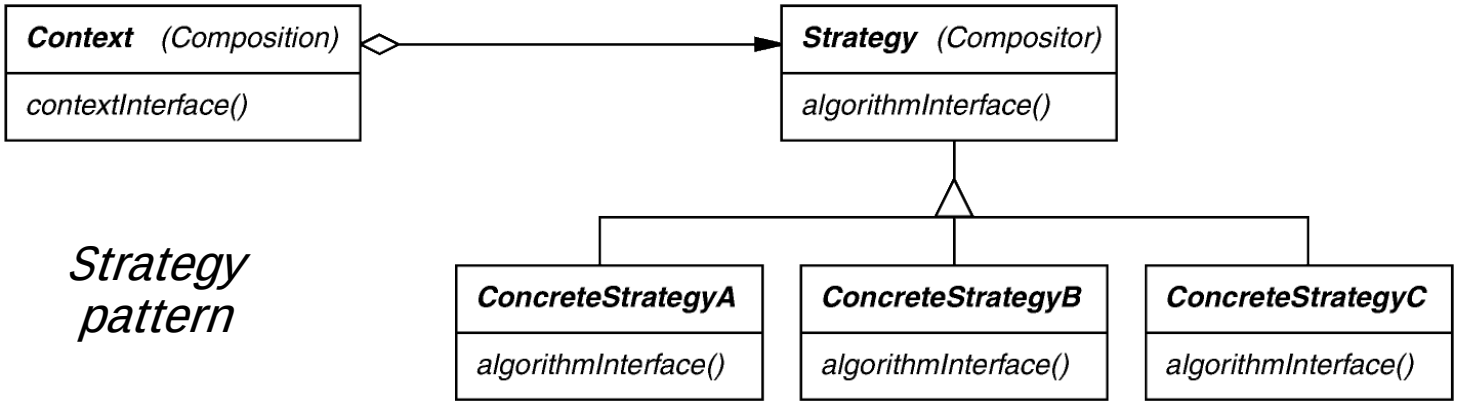
- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)

Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software



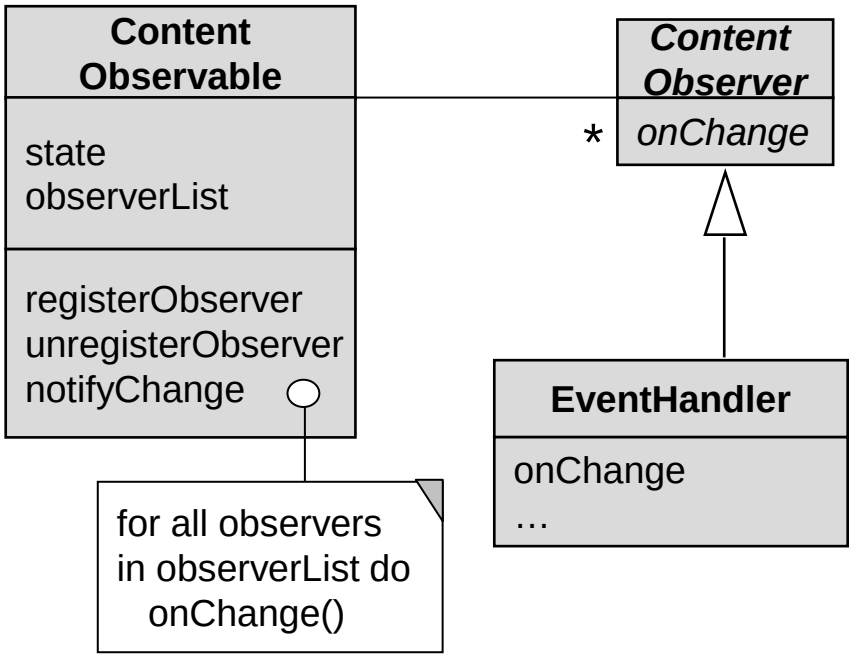
Template Method pattern



Strategy pattern

Process for Applying Patterns

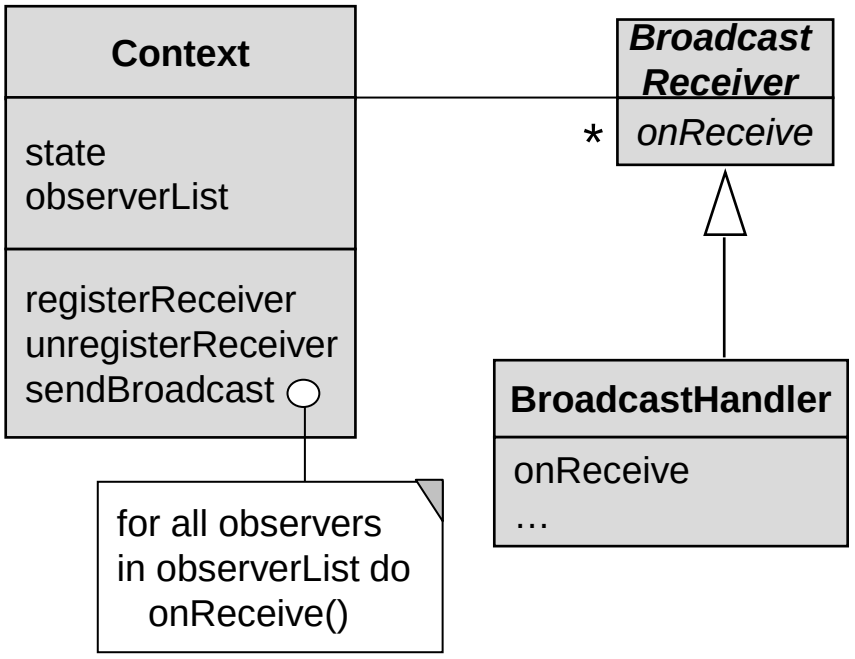
- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software
  - Make design & implementation decisions about how best to apply the selected patterns
    - Patterns may require slight modifications for particular contexts



One use of the Observer Pattern in Android

Process for Applying Patterns

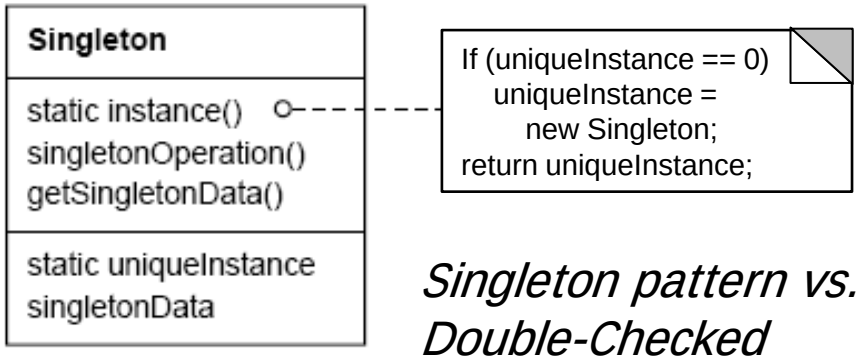
- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software
  - Make design & implementation decisions about how best to apply the selected patterns
    - Patterns may require modifications for particular contexts



A different use of the Observer Pattern in Android

Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software
  - Make design & implementation decisions about how best to apply the selected patterns
    - Patterns may require modifications for particular contexts

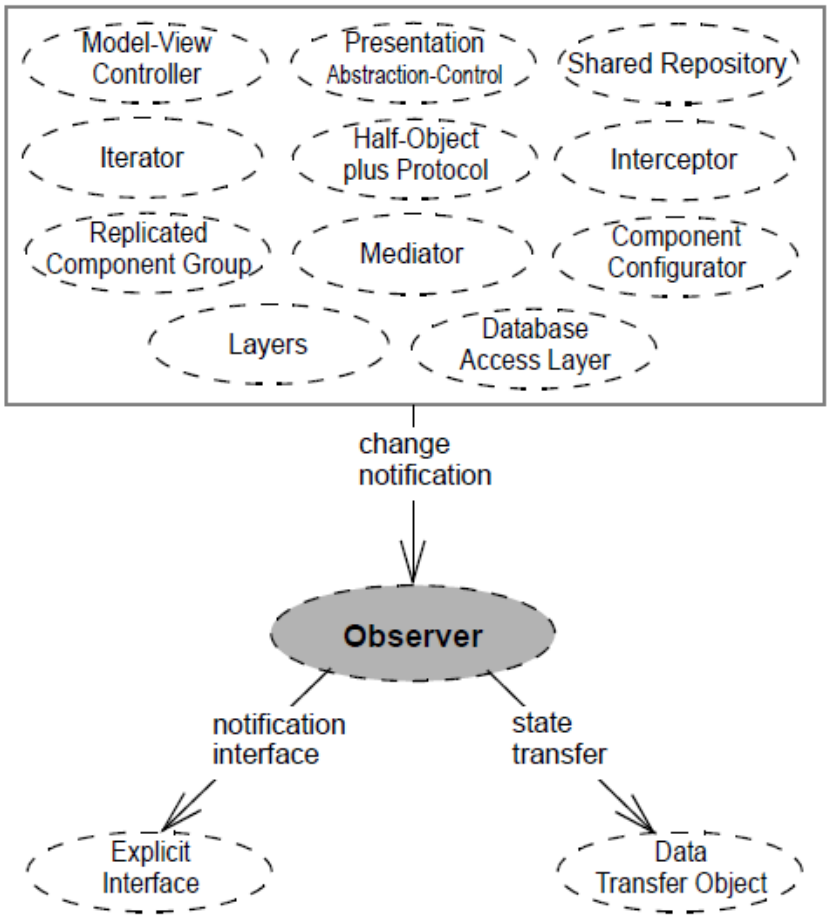


Singleton pattern vs. Double-Checked Locking Optimization Pattern

```
class Singleton {
public:
  static Singleton *instance () {
  // First check
  if (instance_ == 0) {
    Guard<Thread_Mutex> g(lock_);
      if (instance_ == 0) // Double check
      instance_ = new Singleton;
    }
    return instance_;
  }
private:
  static Singleton *instance_;
  static Thread_Mutex lock_;
};
```

Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software
  - Make design & implementation decisions about how best to apply the selected patterns
    - Patterns may require modifications for particular contexts
  - Combine with other patterns & implement/integrate with code



Summary

- Patterns support
  - Design at a more abstract level
    - Treat many class/object interactions as a conceptual unit
    - Emphasize design *qua* design, not (obscure) language features
    - Provide ideal targets for design refactoring
  - Variation-oriented design process
    1. Determine which design elements can vary
    2. Identify applicable pattern(s)
    3. Vary patterns & evaluate trade-offs
    4. Repeat...
- Patterns can be applied in all software lifecycle phases
  - Analysis, design, & reviews
  - Implementation & documentation
  - Testing & optimization
  - Reuse & refactoring
- Resist urge to brand everything as a pattern
  - Articulate specific benefits & demonstrate general applicability
    - e.g., find three different existing examples from code other than your own!

Patterns often equated with OO languages, but can apply to non-OO languages