# Introduction to Patterns and Frameworks

**Douglas C. Schmidt**

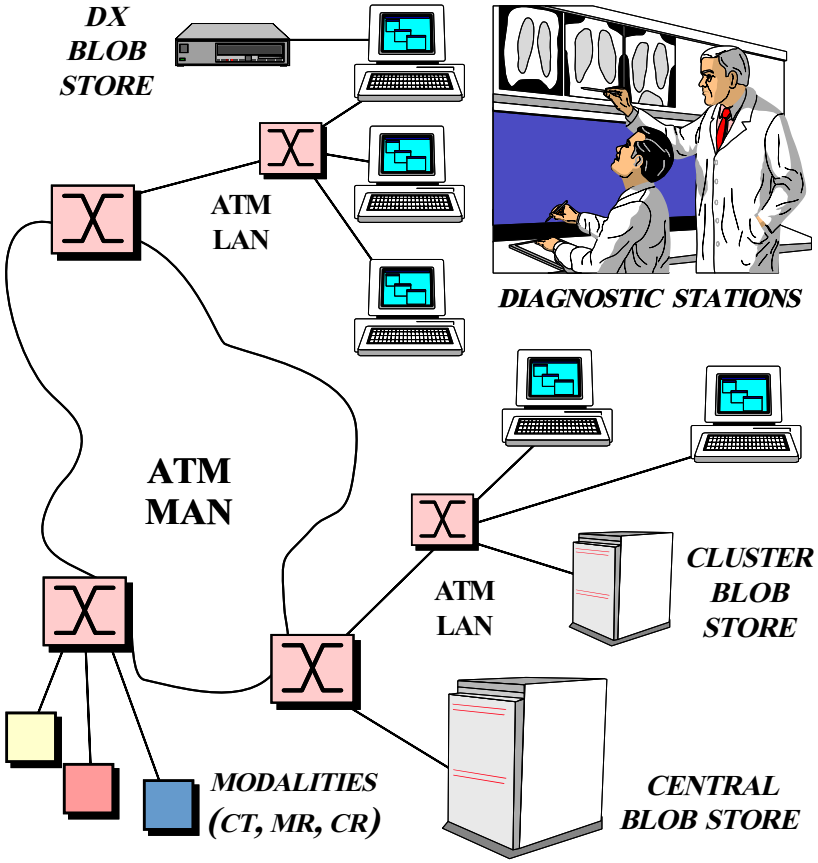
Professor
d.schmidt@vanderbilt.edu
www.cs.wustl.edu/∼schmidt/

Department of EECS
Vanderbilt University
(615) 343-8197

# Patterns and Frameworks

- Motivation for Patterns and Frameworks
- What is a Pattern? A Framework?
- Pattern Categories
- Pattern Examples

# Motivation for Patterns and Frameworks




- Developing software is hard
- Developing reusable software is even harder
- Proven solutions include *patterns* and *frameworks*
- www.cs.wustl.edu/∼schmidt/patterns.html

# Overview of Patterns and Frameworks

- Patterns support reuse of software *architecture* and *design*
  - *Patterns capture the static and dynamic structures and collaborations of successful solutions to problems that arise when building applications in a particular domain*
- Frameworks support reuse of *detailed design* and *code*
  - *A framework is an integrated set of components that collaborate to provide a reusable architecture for a family of related applications*
- Together, *design patterns* and *frameworks* help to improve software quality and reduce development time
  - *e.g.*, reuse, extensibility, modularity, performance

# Patterns of Learning

- Successful solutions to many areas of human endeavor are deeply rooted in patterns
  - In fact, an important goal of education is transmitting *patterns of learning* from generation to generation
- In a moment, we'll explore how patterns are used to learn chess
- Learning to develop good software is similar to learning to play good chess
  - Though the consequences of failure are often far less dramatic!

# Becoming a Chess Master

- ***First learn the rules***
  - *e.g.*, names of pieces, legal movements, chess board geometry and orientation, *etc.*
- ***Then learn the principles***
  - *e.g.*, relative value of certain pieces, strategic value of center squares, power of a threat, *etc.*
- ***However, to become a master of chess, one must study the games of other masters***
  - These games contain *patterns* that must be understood, memorized, and applied repeatedly
- ***There are hundreds of these patterns***

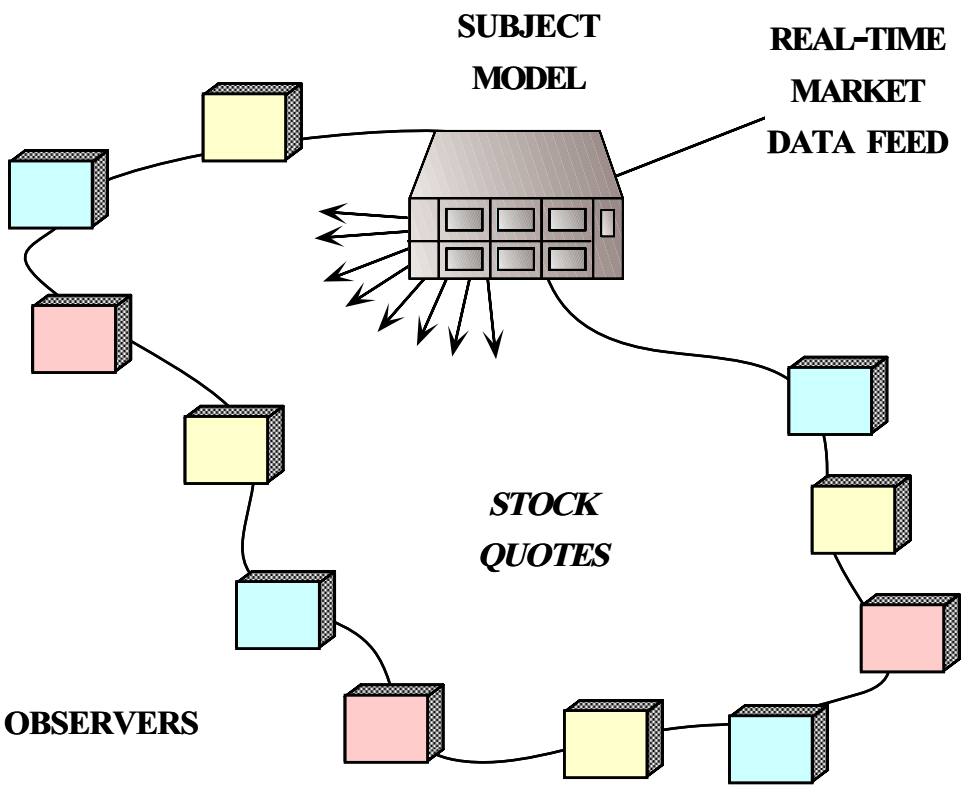
# Becoming a Software Design Master

- ***First learn the rules***
  - *e.g.*, the algorithms, data structures and languages of software
- ***Then learn the principles***
  - *e.g.*, structured programming, modular programming, object oriented programming, generic programming, *etc.*
- ***However, to become a master of software design, one must study the designs of other masters***
  - These designs contain *patterns* that must be understood, memorized, and applied repeatedly
- ***There are hundreds of these patterns***

# Design Patterns

- Design patterns represent *solutions* to *problems* that arise when developing software within a particular *context*
  - *i.e.*, "Pattern == problem/solution pair in a context"
- Patterns capture the static and dynamic *structure* and *collaboration* among key *participants* in software designs
  - They are particularly useful for articulating how and why to resolve *non-functional forces*
- Patterns facilitate reuse of successful *software architectures* and *designs*

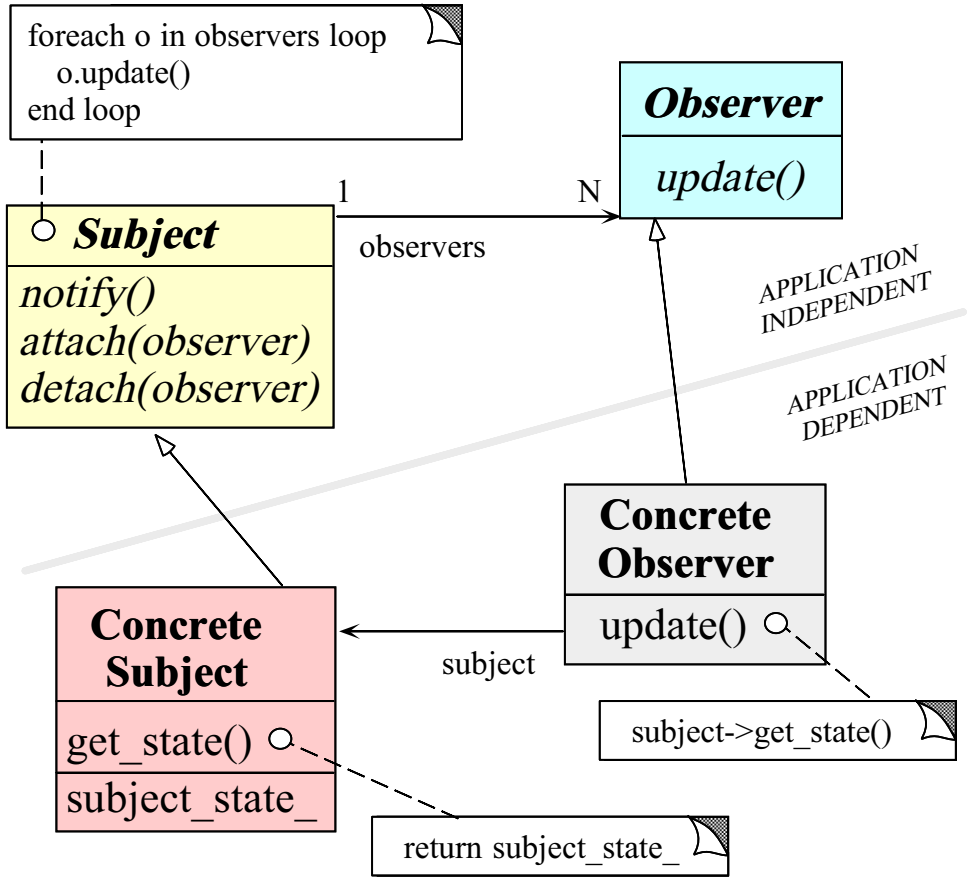
# Example: Stock Quote Service




## Key Forces

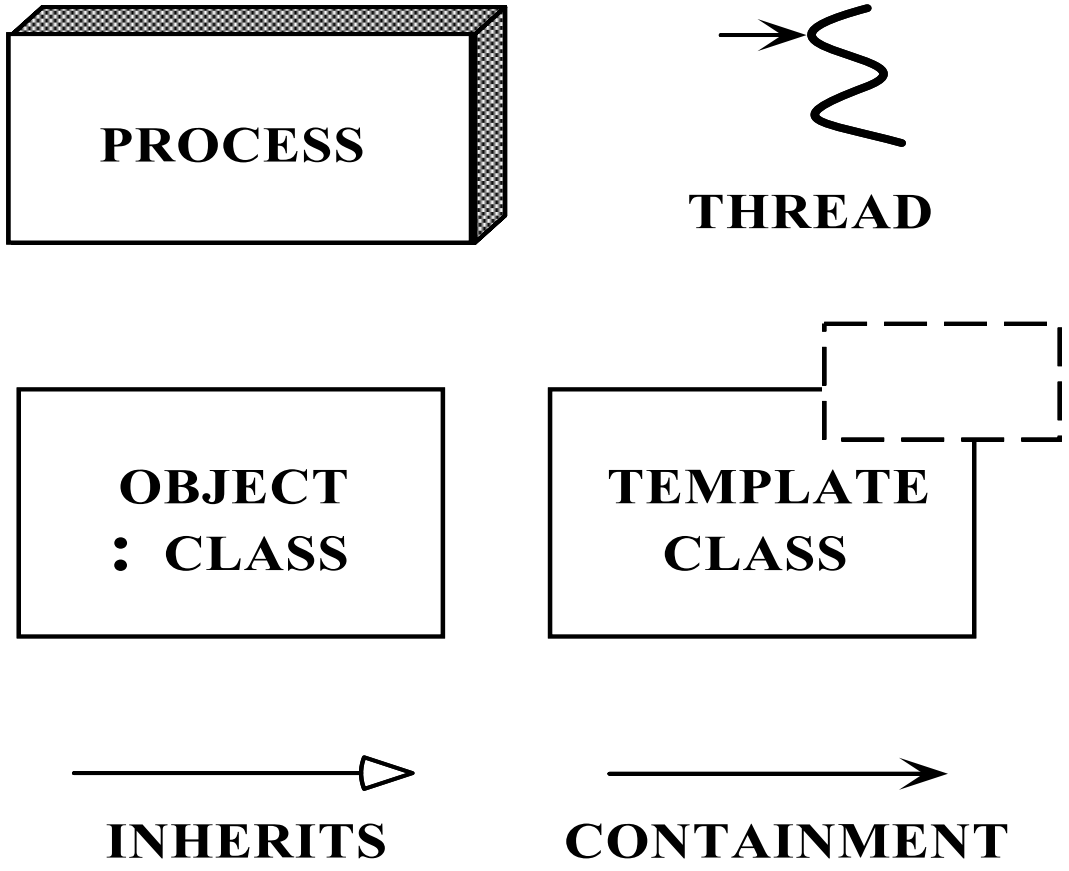
1. There may be many observers
2. Each observer may react differently to the same notification
3. The subject should be as decoupled as possible from the observers
   - *i.e.*, allow observers to change independently of the subject

# Structure of the Observer Pattern

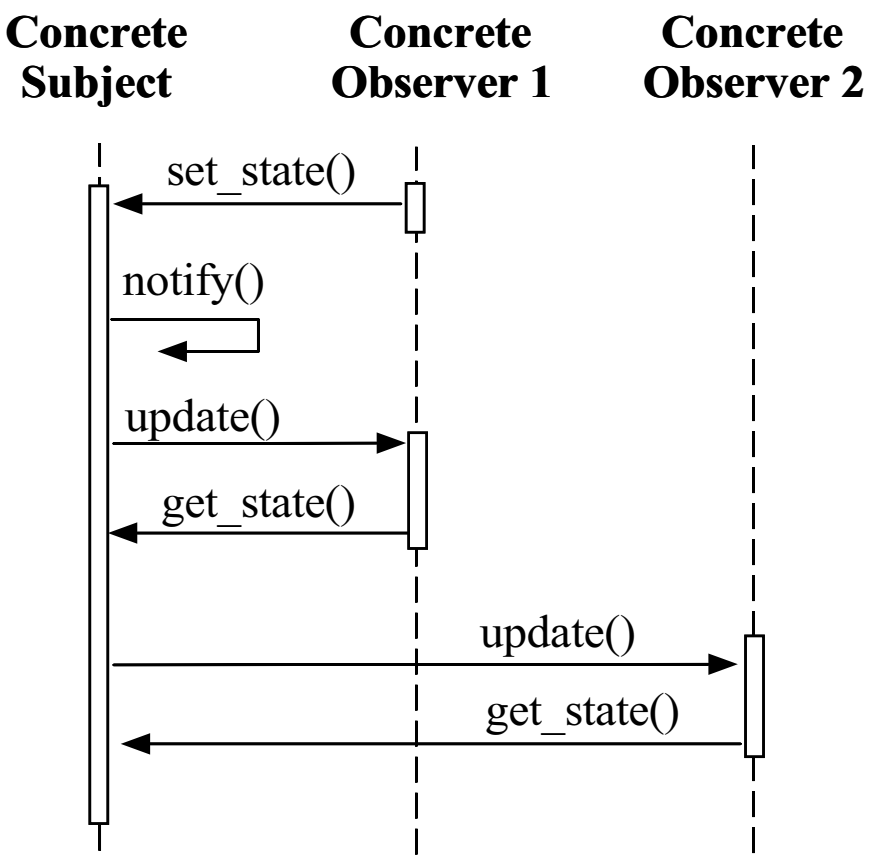



- **Intent**
  - *Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.*

# Graphical Notation



# Collaboration in the Observer Pattern




## Variations

- "Push" architectures combine control flow and data flow
- "Pull" architectures separate control flow from data flow

# Design Pattern Descriptions

Main parts

1. *Name and intent*
2. *Problem and context*
3. *Force(s) addressed*
4. *Abstract description of structure and collaborations in solution*
5. *Positive and negative consequence(s) of use*
6. *Implementation guidelines and sample code*
7. *Known uses and related patterns*

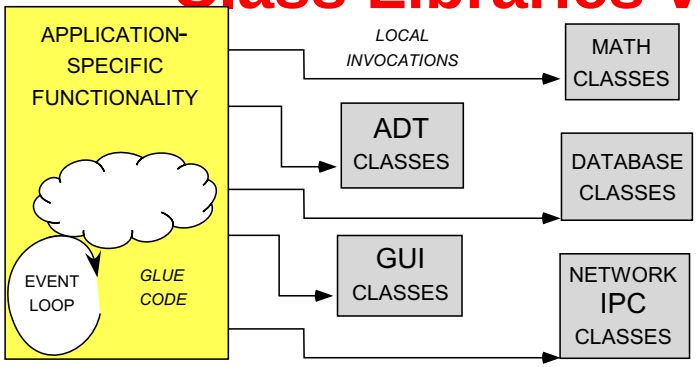
Pattern descriptions are often independent of programming language or implementation details

- Contrast with frameworks

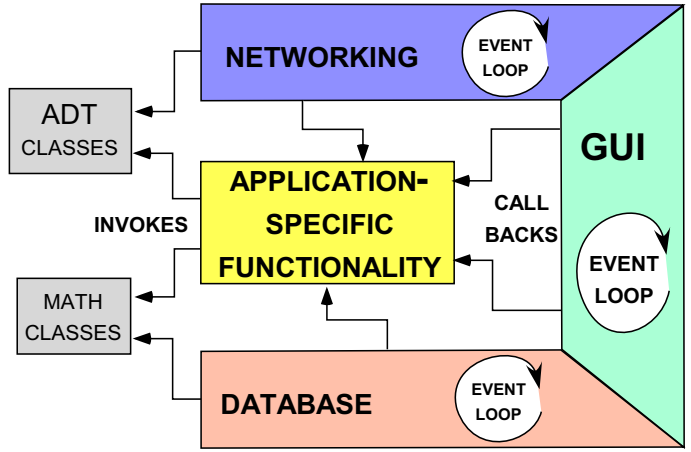
# Frameworks

1. ***Frameworks are semi-complete applications***
   - Complete applications are developed by *inheriting* from, and *instantiating* parameterized framework components
2. ***Frameworks provide domain-specific functionality***
   - *e.g.*, business applications, telecommunication applications, window systems, databases, distributed applications, OS kernels
3. ***Frameworks exhibit inversion of control at run-time***
   - *i.e.*, the framework determines which objects and methods to invoke in response to events

# Class Libraries vs. Frameworks vs. Patterns




(A) CLASS LIBRARY ARCHITECTURE

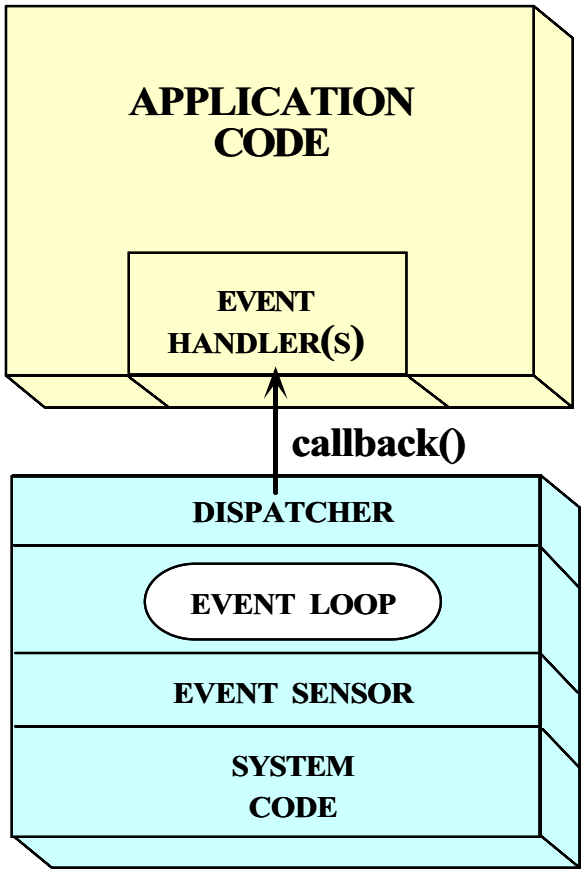



(B) FRAMEWORK ARCHITECTURE

## Definition

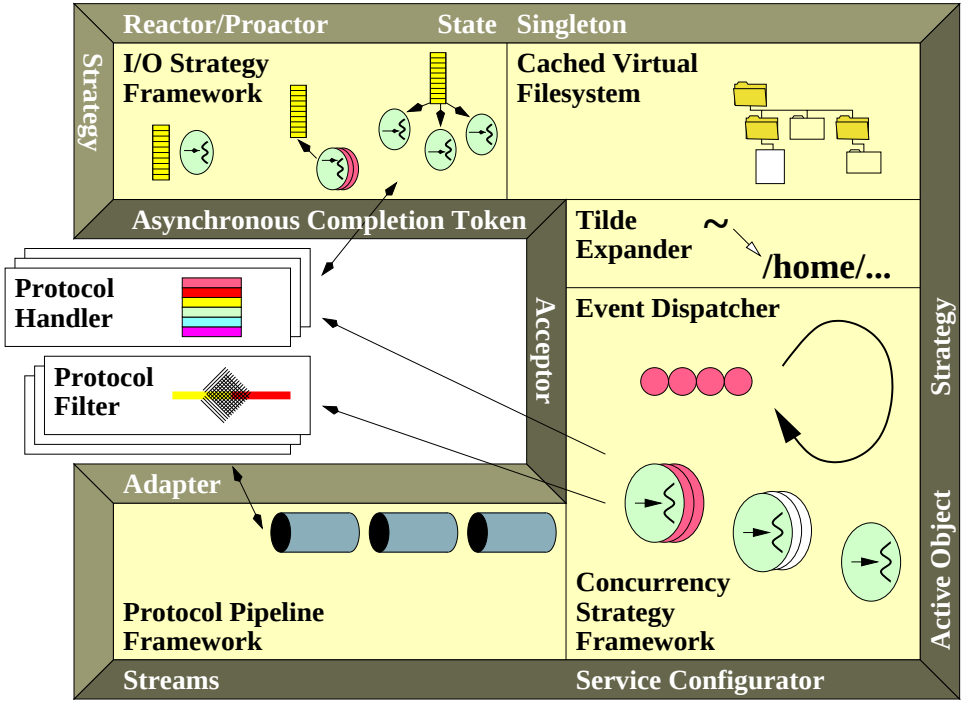
- *Class libraries*
  - Self-contained, "pluggable" ADTs
- *Frameworks*
  - Reusable, "semi-complete" applications
- *Patterns*
  - Problem, solution, context

# Component Integration in Frameworks




- Framework components are loosely coupled via *callbacks*
- Callbacks allow independently developed software components to be connected together
- Callbacks provide a connection-point where generic framework objects can communicate with application objects
  - The framework provides the common *template methods* and the application provides the variant *hook methods*

# Comparing Patterns and Frameworks




- Patterns and frameworks are highly synergistic
  - *i.e.*, neither is subordinate
- Patterns can be characterized as more abstract descriptions of frameworks, which are implemented in a particular language

In general, sophisticated frameworks embody dozens of patterns and patterns are often used to document frameworks

# Design Pattern Space

- *Creational patterns*
  - Deal with initializing and configuring classes and objects
- *Structural patterns*
  - Deal with decoupling interface and implementation of classes and objects
- *Behavioral patterns*
  - Deal with dynamic interactions among societies of classes and objects

# Creational Patterns

- *Factory Method*
  - Method in a derived class creates associates
- *Abstract Factory*
  - Factory for building related objects
- *Builder*
  - Factory for building complex objects incrementally
- *Prototype*
  - Factory for cloning new instances from a prototype
- *Singleton*
  - Factory for a singular (sole) instance

# Structural Patterns

- *Adapter*
  - Translator adapts a server interface for a client
- *Bridge*
  - Abstraction for binding one of many implementations
- *Composite*
  - Structure for building recursive aggregations
- *Decorator*
  - Decorator extends an object transparently

# Structural Patterns (cont'd)

- *Facade*
  - Facade simplifies the interface for a subsystem
- *Flyweight*
  - Many fine-grained objects shared efficiently
- *Proxy*
  - One object approximates another

# Behavioral Patterns

- *Chain of Responsibility*
  - Request delegated to the responsible service provider
- *Command*
  - Request as first-class object
- *Interpreter*
  - Language interpreter for a small grammar
- *Iterator*
  - Aggregate elements are accessed sequentially

# Behavioral Patterns (cont'd)

- *Mediator*
  - Mediator coordinates interactions between its associates
- *Memento*
  - Snapshot captures and restores object states privately
- *Observer*
  - Dependents update automatically when a subject changes
- *State*
  - Object whose behavior depends on its state

# Behavioral Patterns (cont'd)

- *Strategy*
  - Abstraction for selecting one of many algorithms
- *Template Method*
  - Algorithm with some steps supplied by a derived class
- *Visitor*
  - Operations applied to elements of an heterogeneous object structure

# When to Use Patterns

1. *Solutions to problems that recur with variations*
   - No need for reuse if the problem only arises in one context
2. *Solutions that require several steps*
   - Not all problems need all steps
   - Patterns can be overkill if solution is simple linear set of instructions
3. *Solutions where the solver is more interested in the existence of the solution than its complete derivation*
   - Patterns leave out too much to be useful to someone who really wants to understand
     - They can be a temporary bridge, however

# What Makes a Pattern a Pattern?

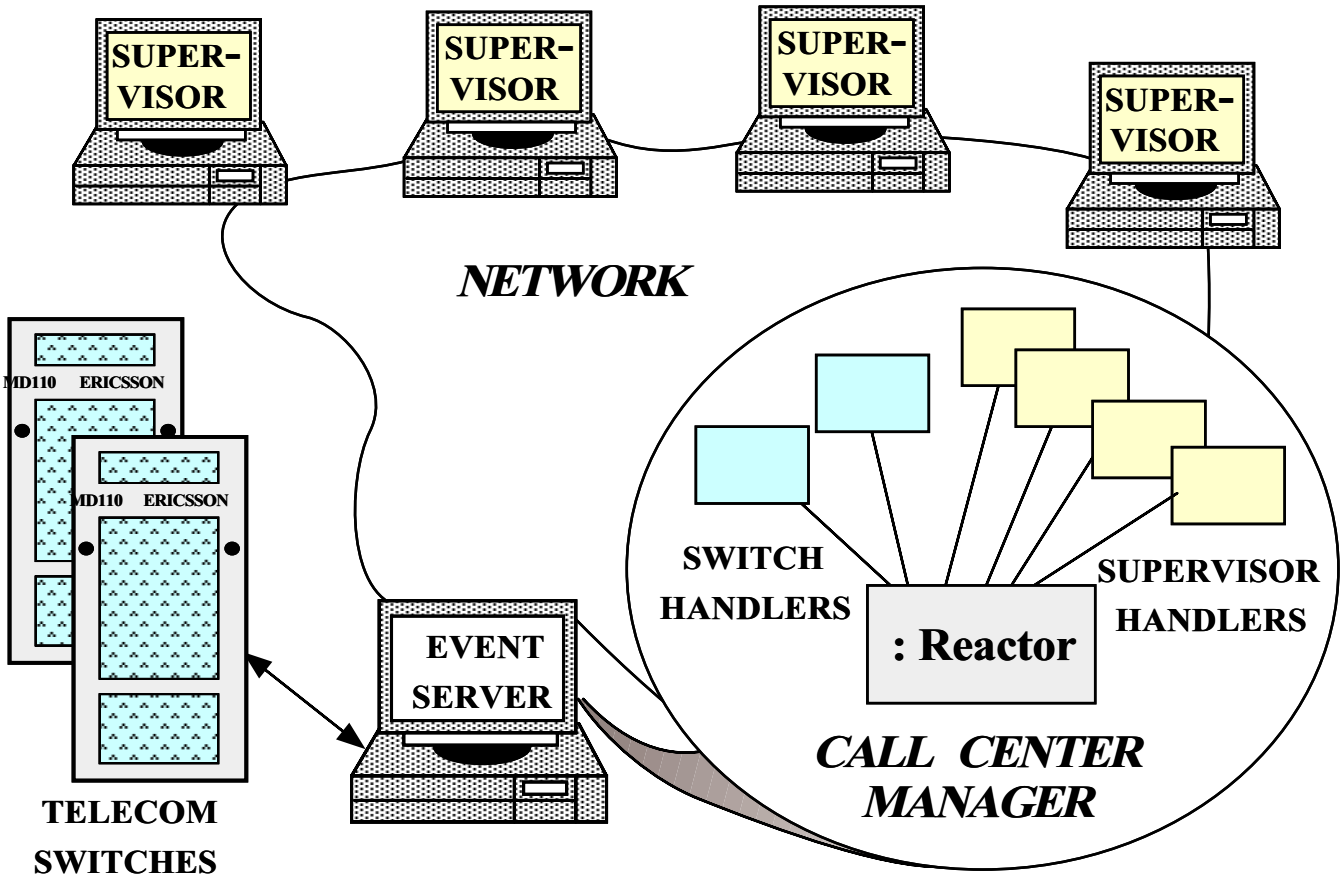
A *pattern* must:

- ***Solve a problem***,
  - *i.e.*, it must be useful!
- ***Have a context***,
  - It must describe where the solution can be used
- ***Recur***,
  - It must be relevant in other situations
- ***Teach***
  - It must provide sufficient understanding to tailor the solution
- ***Have a name***
  - It must be referred to consistently

# Case Study: A Reusable Object-Oriented Communication Software Framework

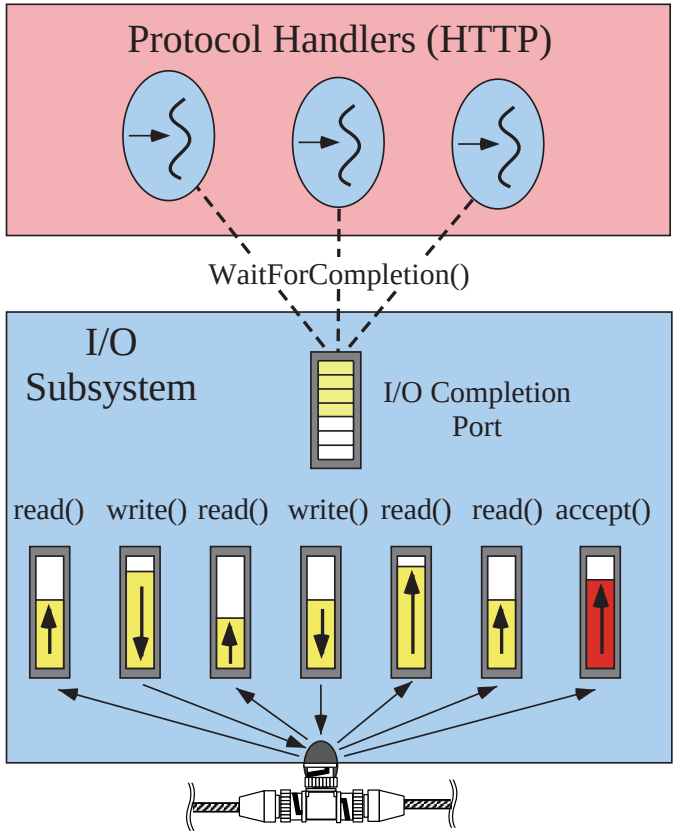
- Developing portable, reusable, and efficient communication software is hard
- OS platforms are often fundamentally incompatible
  - *e.g.*, different concurrency and I/O models
- Thus, it may be impractical to directly reuse:
  - *Algorithms*
  - *Detailed designs*
  - *Interfaces*
  - *Implementations*

# System Overview



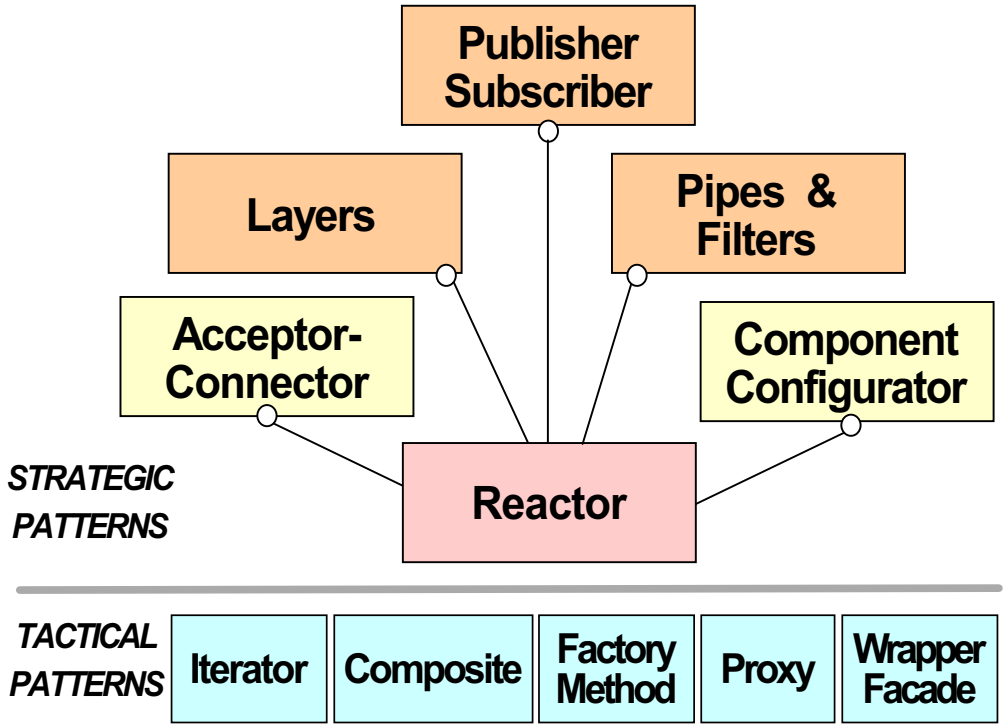

- OO framework for Call Center Management
- www.cs.wustl.edu/~schmidt/PDF/ECOOP-95.pdf
- www.cs.wustl.edu/~schmidt/PDF/DSEJ-94.pdf

# Problem: Cross-platform Reuse



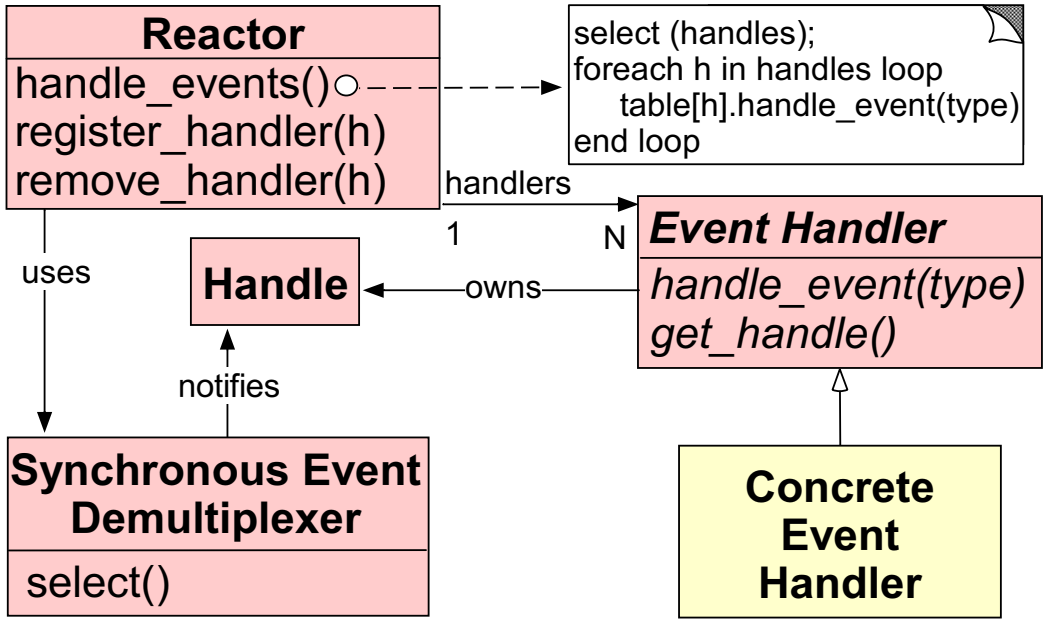

- OO framework was first developed on UNIX and later ported to Windows NT 3.51 in 1993
- UNIX and Windows NT have fundamentally different I/O models
  - *i.e.*, synchronous vs. asynchronous
- Thus, direct reuse of original framework was infeasible
  - Later solved by ACE and Windows NT 4.0

# Solution: Reuse Design Patterns




- Patterns support reuse of *software architecture*
- Patterns embody successful *solutions* to *problems* that arise when developing software in a particular *context*
- Patterns reduced project risk by leveraging proven design expertise

# The Reactor Pattern




www.cs.wustl.edu/∼schmidt/POSA/

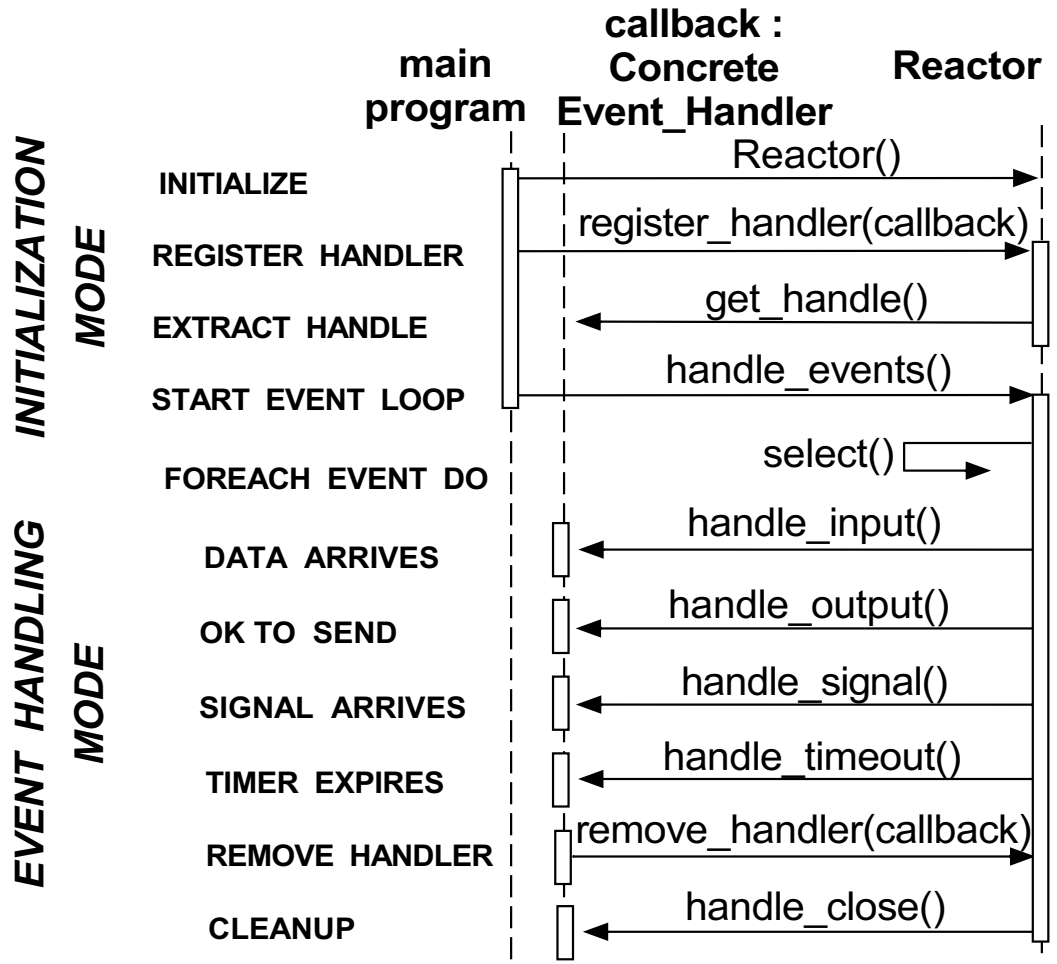
## Intent

- *Decouples synchronous event demuxing & dispatching from event handling*

## Forces Resolved

- Efficiently demux events *synchronously* within one thread
- Extending applications without changing demux infrastructure

# Collaboration in the Reactor Pattern




- Note *inversion of control*
- Also note how long-running event handler callbacks can degrade quality of service

# Using ACE's Reactor Pattern Implementation

```
#include "ace/Reactor.h"
class My_Event_Handler : public ACE_Event_Handler {
public:
  virtual int handle_input (ACE_HANDLE h) {
    cout << "input on handle " << h << endl;
    return 0; }
  virtual int handle_signal (int signum,
                             siginfo_t *,
                             ucontext_t *) {
    cout << "signal " << signum << endl;
    return 0; }
  virtual ACE_HANDLE get_handle (void) const {
    return ACE_STDIN; }
};
```

# Using ACE's Reactor Pattern Implementation (cont'd)

```
int main (int argc, char *argv[])
{
  My_Event_Handler eh;
  ACE_Reactor reactor;

  reactor.register_handler
    (&eh, ACE_Event_Handler::READ_MASK);

  reactor.register_handler
    (SIGINT, &eh);

  for (;;)
    reactor.handle_events ();

  /* NOTREACHED */
  return 0;
}
```

# Differences Between UNIX and Windows NT

- *Reactive* vs. *Proactive* I/O
  - Reactive I/O is synchronous
  - Proactive I/O is asynchronous
    - Requires additional interfaces to “arm” the I/O mechanism
  - See *Proactor* pattern
    - www.cs.wustl.edu/∼schmidt/POSA/
- Other differences include
  - *Resource limitations*
    - *e.g.*, Windows **`WaitForMultipleObjects()`** limits HANDLEs per-thread to 64
  - *Demultiplexing fairness*
    - *e.g.*, **`WaitForMultipleObjects`** always returns the lowest active HANDLE

# Lessons Learned from Case Study

- Real-world constraints of OS platforms can preclude direct reuse of communication software
  - *e.g.*, must often use non-portable features for performance
- Reuse of design patterns may be the only viable means to leverage previous development expertise
- Design patterns are useful, but are no panacea
  - Managing expectations is crucial
  - Deep knowledge of platforms, systems, and protocols is also very important

# Key Principles

- Successful patterns and frameworks can be boiled down to a few key principles:

  1. Separate interface from implementation
  2. Determine what is *common* and what is *variable* with an interface and an implementation
     - Common == stable
  3. Allow substitution of *variable* implementations via a *common* interface

- Dividing *commonality* from *variability* should be goal-oriented rather than exhaustive

# Planning for Change

- Often, aspects of a design “seem” constant until they are examined in the light of the dependency structure of an application
  - At this point, it becomes necessary to refactor the framework or pattern to account for the variation
- Frameworks often represent the distinction between commonality and variability via *template methods* and *hook methods,* respectively

# The Open/Closed Principle

- Determining common vs. variable components is important
  - Insufficient variation makes it hard for users to customize framework components
  - Conversely, insufficient commonality makes it hard for users to comprehend and depend upon the framework's behavior
- In general, dependency should always be in the direction of stability
  - *i.e.*, a software component should not depend on any component that is less stable than itself
- The "Open/Closed" principle
  - This principle allows the most stable component to be extensible

# The Open/Closed Principle (cont'd)

- Components should be:
  - open for extension
  - closed for modification
- Impacts
  - Abstraction is good
  - Inheritance and polymorphism are good
  - Public data members and global data are bad
  - Run-time type identification can be bad

# Violation of Open/Closed Principle

```
struct Shape { enum Type { CIRCLE, SQUARE }
                     shape_type;
               /* . . . */ };
void draw_square (const Square &);
void draw_circle (const Circle &);

void draw_shape (const Shape &shape) {
  switch (shape.shape_type) {
    case SQUARE:
      draw_square ((const Square &) shape);
      break;
    case CIRCLE:
      draw_circle ((const Circle &) shape);
      break;
    // etc.
```

# Application of Open/Closed Principle

```
class Shape {
public:
  virtual void draw () const = 0;
};

class Square : public Shape { /* . . . */ };
class Circle : public Shape { /* . . . */ };

typedef vector<Shape> Shape_Vector;

void draw_all (const Shape_Vector &shapes) {
  for (Shape_Vector::iterator i = shapes.begin();
       i != shapes.end ();
       i++)
    (*iterator).draw ();
}
```

# Benefits of Design Patterns

- ***Design patterns enable large-scale reuse of software architectures***
  - They also help document systems to enhance understanding
- ***Patterns explicitly capture expert knowledge and design tradeoffs, and make this expertise more widely available***
- ***Patterns help improve developer communication***
  - Pattern names form a vocabulary
- ***Patterns help ease the transition to object-oriented technology***

# Drawbacks to Design Patterns

- ***Patterns do not lead to direct code reuse***
- ***Patterns are deceptively simple***
- ***Teams may suffer from pattern overload***
- ***Patterns are validated by experience and discussion rather than by automated testing***
- ***Integrating patterns into a software development process is a human-intensive activity***

# Tips for Using Patterns Effectively

- ***Do not recast everything as a pattern.***
  - Instead, develop strategic domain patterns and reuse existing tactical patterns
- ***Institutionalize rewards for developing patterns***
- ***Directly involve pattern authors with application developers and domain experts***
- ***Clearly document when patterns apply and do not apply***
- ***Manage expectations carefully***

# Lessons Learned using OO Frameworks

- ***Benefits of frameworks***
  - Enable direct reuse of code
  - Facilitate larger amounts of reuse than stand-alone functions or individual classes
- ***Drawbacks of frameworks***
  - High initial learning curve
    - Many classes, many levels of abstraction
  - The flow of control for reactive dispatching is non-intuitive
  - Verification and validation of generic components is hard

# Patterns and Framework Literature

- **Books**
  - Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software* AW, '94
  - *Pattern Languages of Program Design* series by AW, '95-'99.
  - Siemens & Schmidt, *Pattern-Oriented Software Architecture*, Wiley, volumes '96 & '00 (`www.posa.uci.edu`)
  - Schmidt & Huston, *C++ Network Programming: Mastering Complexity with ACE and Patterns*, AW, '02 (`www.cs.wustl.edu/~schmidt/ACE/book1/`)
  - Schmidt & Huston, *C++ Network Programming: Systematic Reuse with ACE and Frameworks*, AW, '03 (`www.cs.wustl.edu/~schmidt/ACE/book2/`)

# Conferences and Workshops on Patterns

- Pattern Language of Programs Conferences
  - September 8-12, 2003, Monticello, Illinois, USA
  - http://hillside.net/conferences/plop.htm
- The European Pattern Languages of Programming conference
  - June 25-29, 2003, Kloster Irsee, Germany
  - http://hillside.net/conferences/europlop.htm
- Middleware 2003
  - June 16-20, 2003, Rio, Brazil
  - www.cs.wustl.edu/ schmidt/activities-chair.html

# Summary

- Mature engineering disciplines have handbooks that describe successful solutions to known problems
  - *e.g.*, automobile designers don't design cars using the laws of physics, they adapt adequate solutions from the handbook known to work well enough
  - The extra few percent of performance available by starting from scratch typically isn't worth the cost
- Patterns can form the basis for the handbook of software engineering
  - If software is to become an engineering discipline, successful practices must be systematically documented and widely disseminated