# Object-Oriented Patterns & Frameworks


Dr. Douglas C. Schmidt

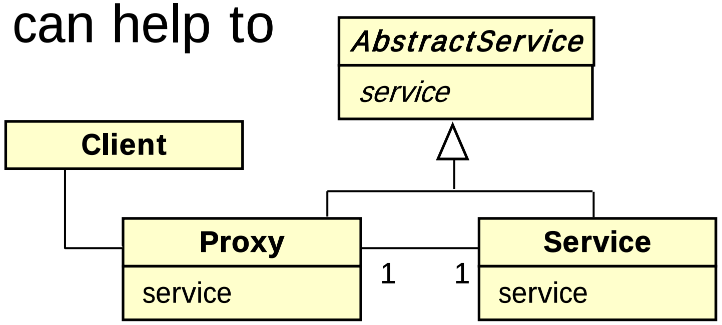
d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of EECS
Vanderbilt University
Nashville, Tennessee


# Goals of this Presentation

Show *by example* how patterns & frameworks can help to




- *Codify good OO software design & implementation practices*
  – distill & generalize experience
  – aid to novices & experts alike
- *Give design structures explicit names*
  – common vocabulary
  – reduced complexity
  – greater expressivity
- *Capture & preserve design & implementation knowledge*
  – articulate key decisions succinctly
  – improve documentation
- *Facilitate restructuring/refactoring*
  – patterns & frameworks are interrelated
  – enhance flexibility, reuse, & productivity

```
class Reactor {
public:
  /// Singleton access point.
  static Reactor *instance (void);

  /// Run event loop.
  void run_event_loop (void);

  /// End event loop.
  void end_event_loop (void);

  /// Register @a event_handler
  /// for input events.
  void register_input_handler
    (Event_Handler *eh);

  /// Remove @a event_handler
  /// for input events.
  void remove_input_handler
    (Event_Handler *eh);
```

# Tutorial Overview

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

## Part I: Motivation & Concepts

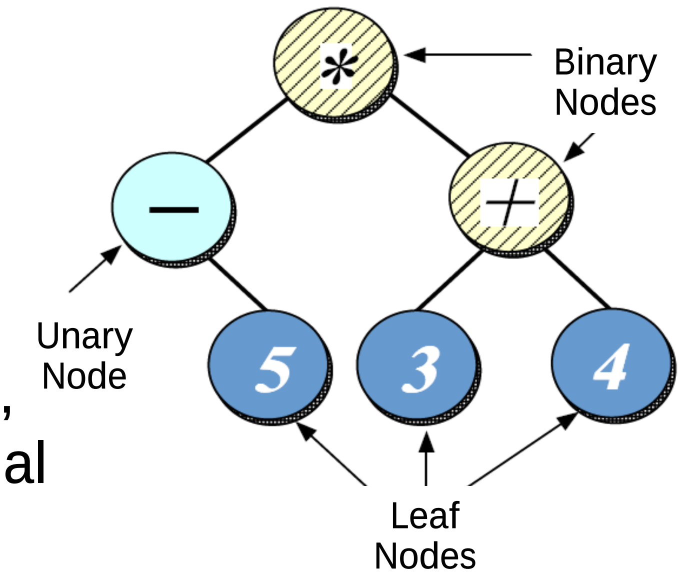
- The issue
- What patterns & frameworks are
- What they're good for
- How we develop/categorize them

## Part II: Case Study

- Use patterns & frameworks to build an expression tree application
- Demonstrate usage & benefits

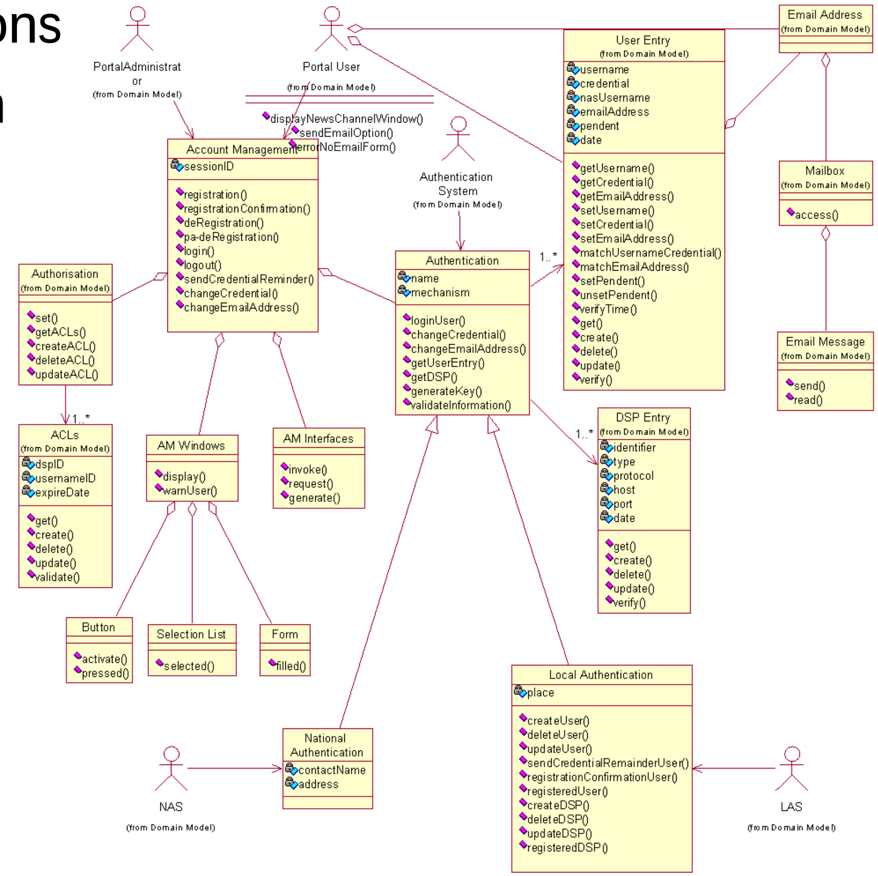
## Part III: Wrap-Up

- Life beyond the GoF book, observations, caveats, concluding remarks, & additional references



# Part I: Motivation & Concepts

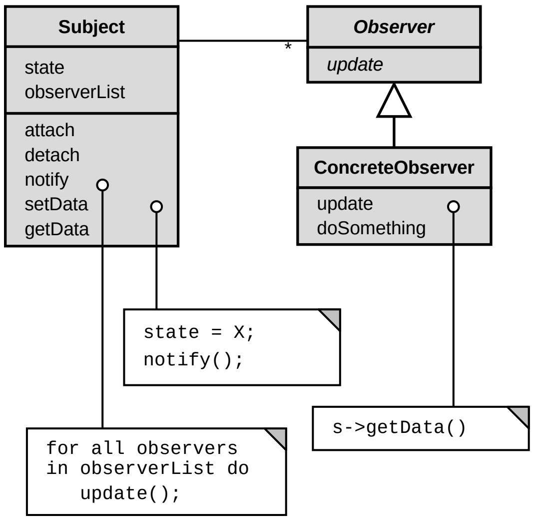
- OOD methods emphasize design notations
  - Fine for specification & documentation



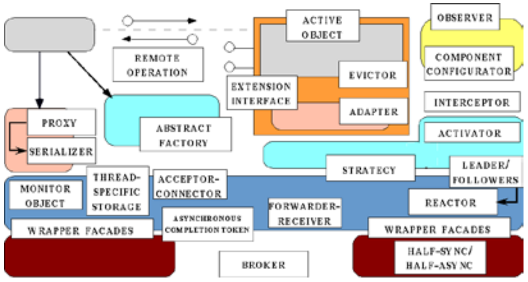
# Part I: Motivation & Concepts

- OOD methods emphasize design notations
  - Fine for specification & documentation
- But OOD is more than just drawing diagrams
  - Good draftsmen are not necessarily good architects!

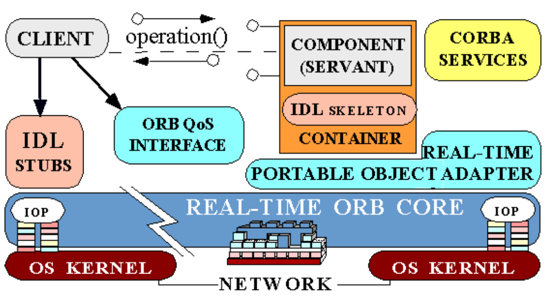
# Part I: Motivation & Concepts




- OOD methods emphasize design notations
  - Fine for specification & documentation
- But OOD is more than just drawing diagrams
  - Good draftsmen are not necessarily good architects!
- Good OO designers rely on lots of experience
  - At least as important as syntax
- Most powerful reuse combines *design & code* reuse
  - *Patterns*: Match problem to design experience



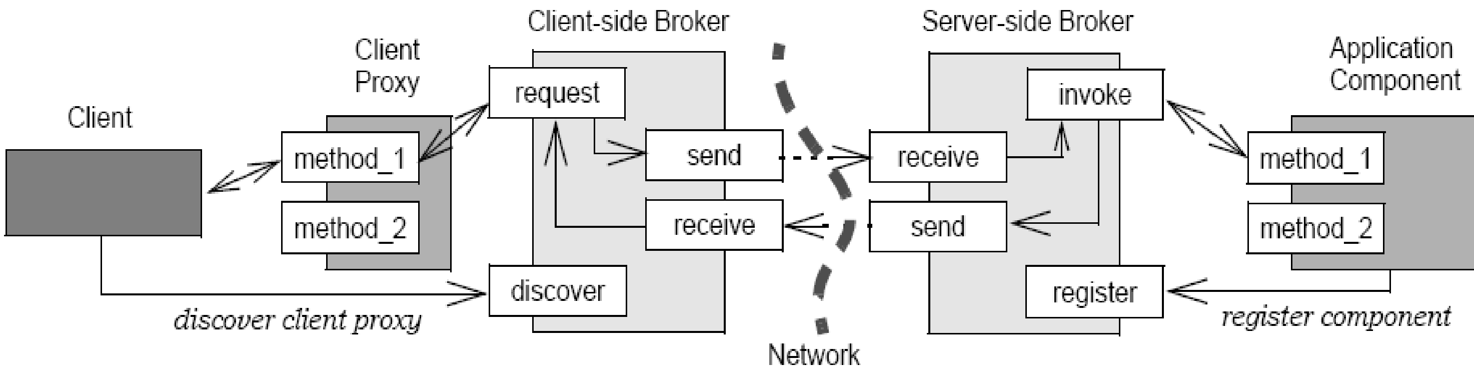

  - *Frameworks*: Reify patterns within a domain context



# Recurring Design Structures

Well-designed OO systems exhibit recurring structures that promote

– Abstraction
– Flexibility
– Modularity
– Elegance

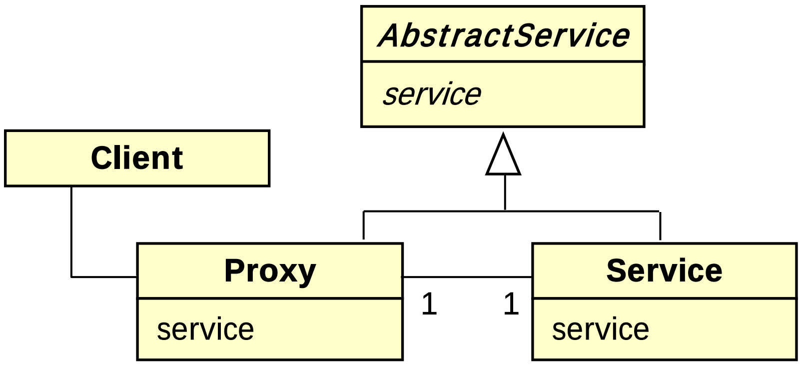



*Therein lies valuable design knowledge*

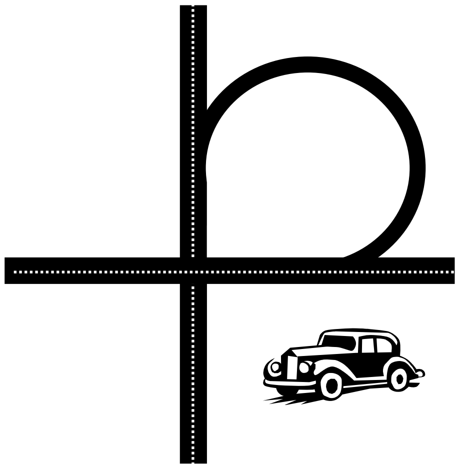

Problem: capturing, communicating, applying, & preserving this knowledge without undue time, effort, & risk

# A *Pattern*...

- Abstracts & names a recurring design structure
- Comprises class and/or object
  - Dependencies
  - Structures
  - Interactions
  - Conventions
- Specifies the design structure explicitly
- Is distilled from actual design experience



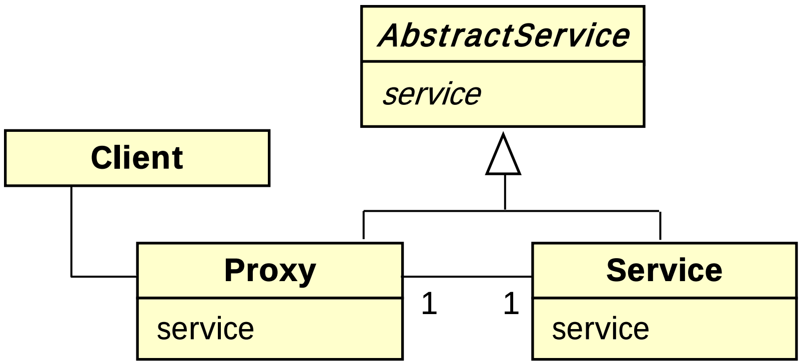

*The Proxy Pattern*



Presents solution(s) to common (software) problem(s) arising within a context

# Four Basic Parts of a Pattern

1. Name
2. Problem (including "forces" & "applicability")
3. Solution (both visual & textual descriptions)
4. Consequences & trade-offs of applying the pattern

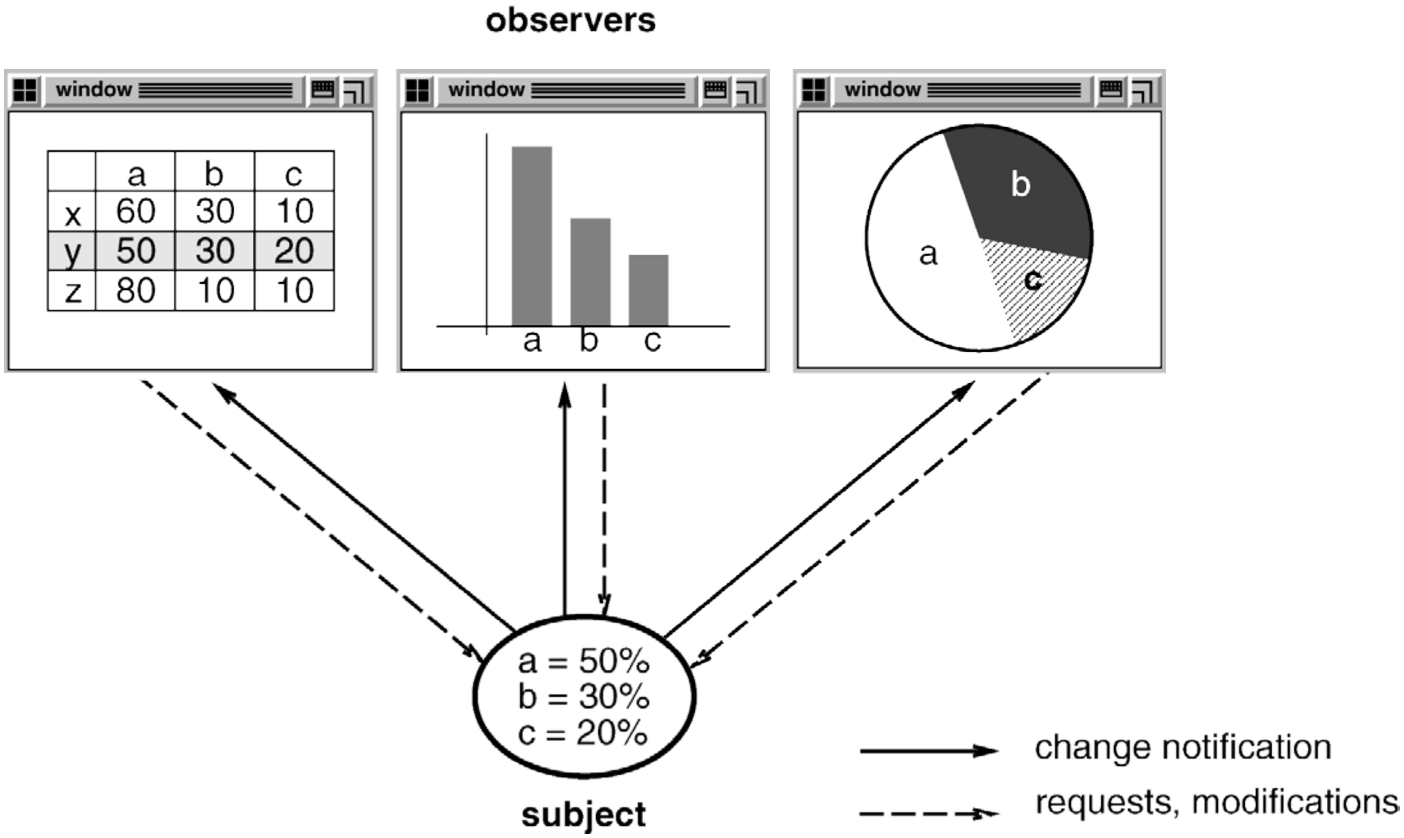


*The Proxy Pattern*

Key characteristics of patterns include:

- Language- & implementation-independent
- "Micro-architecture"
  - i.e., "society of objects"
- Adjunct to existing methodologies
  - e.g., RUP, Fusion, SCRUM, etc.

# Example: Observer



# Observer object behavioral

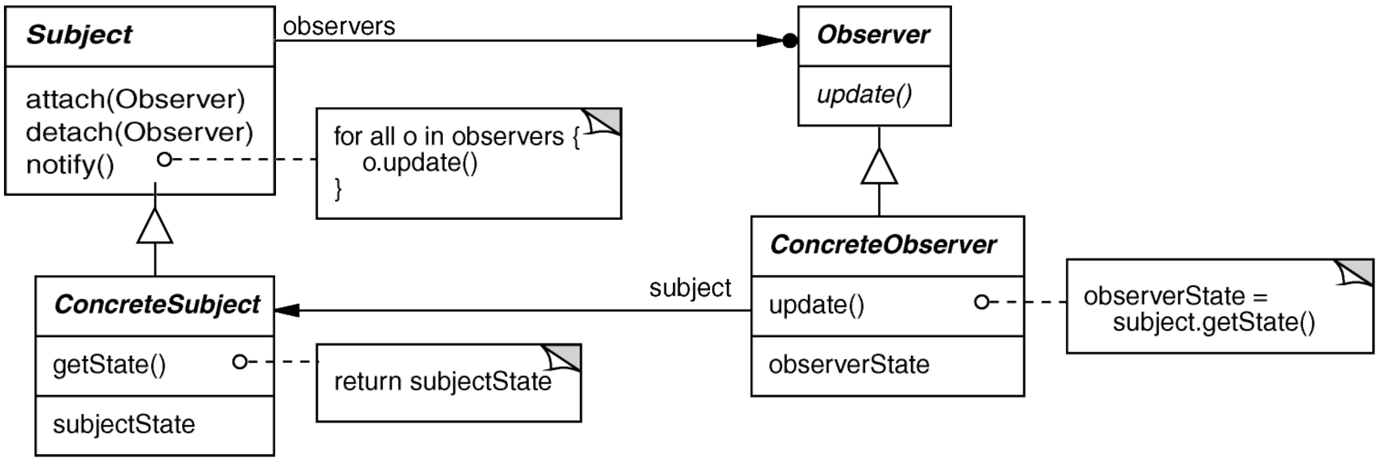
## Intent

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

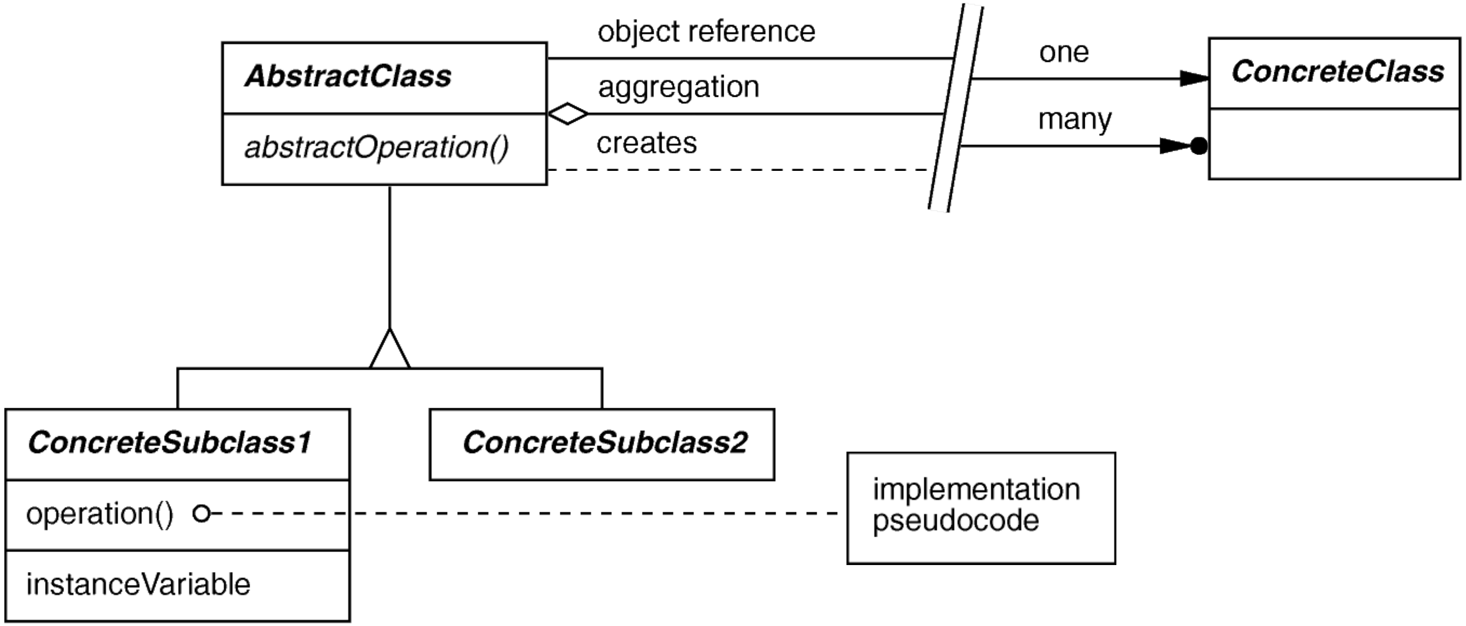
## Applicability

– an abstraction has two aspects, one dependent on the other
– a change to one object requires changing untold others
– an object should notify unknown other objects

## Structure

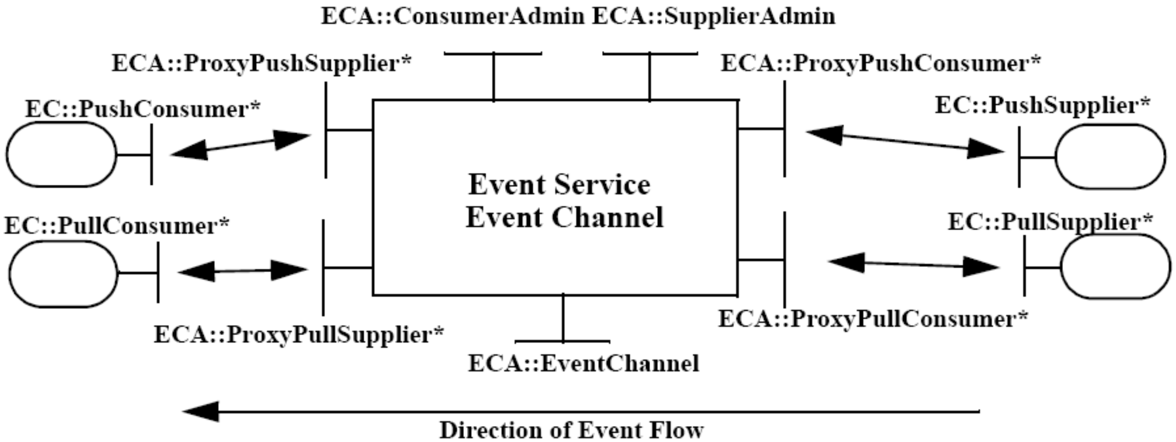


# Modified UML/OMT Notation



# Observer object behavioral

```
class ProxyPushConsumer : public // …
  virtual void push (const CORBA::Any &event) {
    for (std::vector<PushConsumer>::iterator i
          (consumers.begin ()); i != consumers.end (); i++)
      (*i).push (event);
  }

class MyPushConsumer : public // ….
  virtual void push
   (const CORBA::Any &event) { /* consume the event. */ }
```




CORBA Notification Service example using C++ Standard Template Library (STL) iterators (which is an example of the Iterator pattern from GoF)

# Observer object behavioral

## Consequences

+ modularity: subject & observers may vary independently
+ extensibility: can define & add any number of observers
+ customizability: different observers offer different views of subject

– unexpected updates: observers don't know about each other

– update overhead: might need hints or filtering

## Implementation

– subject-observer mapping

– dangling references

– update protocols: the push & pull models

– registering modifications of interest explicitly

## Known Uses

– Smalltalk Model-View-Controller (MVC)

– InterViews (Subjects & Views, Observer/Observable)

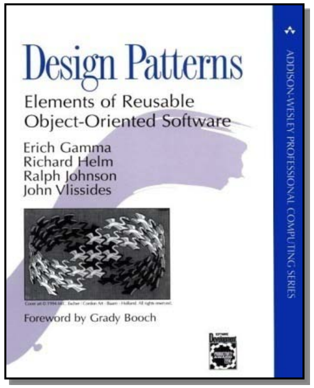
– Andrew (Data Objects & Views)

– Smart phone event frameworks (e.g., Symbian, Android, iPhone)

– Pub/sub middleware (e.g., CORBA Notification Service, Java Message Service, DDS)

– Mailing lists

# Design Space for GoF Patterns




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

**Scope**: domain over which a pattern applies

**Purpose**: reflects what a pattern does

# GoF Pattern Template (1st half)

**Intent**
- short description of the pattern & its purpose

**Also Known As**
- Any aliases this pattern is known by

**Motivation**
- motivating scenario demonstrating pattern's use

**Applicability**
- circumstances in which pattern applies

**Structure**
- graphical representation of pattern using modified UML notation

**Participants**
- participating classes and/or objects & their responsibilities

# GoF Pattern Template (2nd half)

...

**Collaborations**

how participants cooperate to carry out their responsibilities

**Consequences**

the results of application, benefits, liabilities

**Implementation**

pitfalls, hints, techniques, plus language-dependent issues

**Sample Code**

sample implementations in C++, Java, C#, Python, Smalltalk, C, etc.

**Known Uses**

examples drawn from existing systems

**Related Patterns**

discussion of other patterns that relate to this one

# Benefits & Limitations of Patterns

***Benefits***

- *Design* reuse
- Uniform design vocabulary
- Enhance understanding, restructuring, & team communication
- Basis for automation
- Transcends language-centric biases/myopia
- Abstracts away from many unimportant details

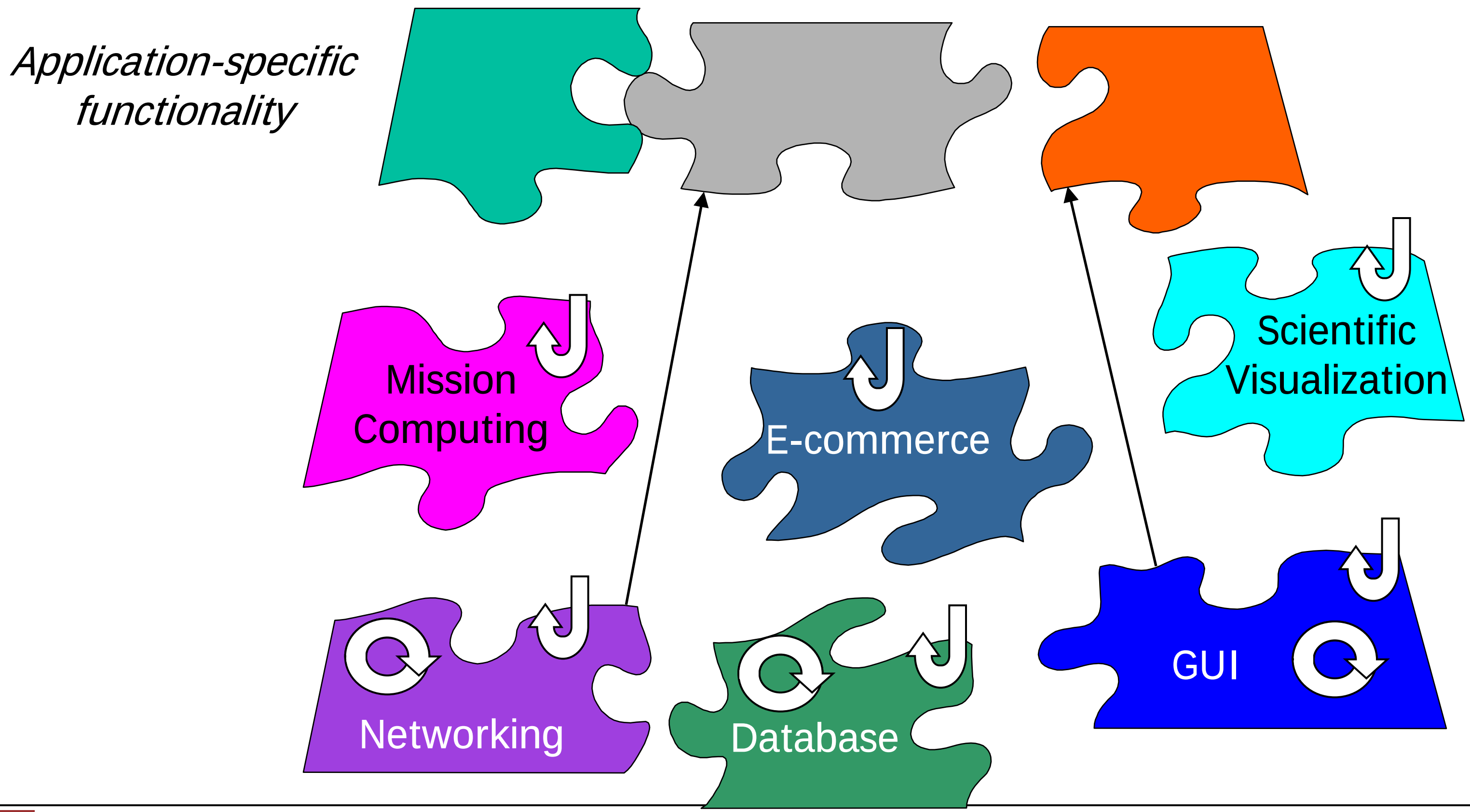
***Limitations***

- Require significant tedious & error-prone human effort to handcraft pattern implementations *design* reuse
- Can be deceptively simple uniform design vocabulary
- May limit design options
- Leaves important (implementation) details unresolved

Addressing the limitations of patterns requires more than just *design* reuse

# Overview of Frameworks

- Frameworks exhibit "inversion of control" at runtime via callbacks
- Frameworks provide integrated domain-specific structures & functionality
- Frameworks are "semi-complete" applications



# Motivation for Frameworks







*Proprietary & Stovepiped Application & Infrastructure Software*




*Standard/COTS Hardware & Networks*

Legacy embedded systems have historically been:

- Stovepiped
- Proprietary
- Brittle & non-adaptive
- Expensive
- Vulnerable

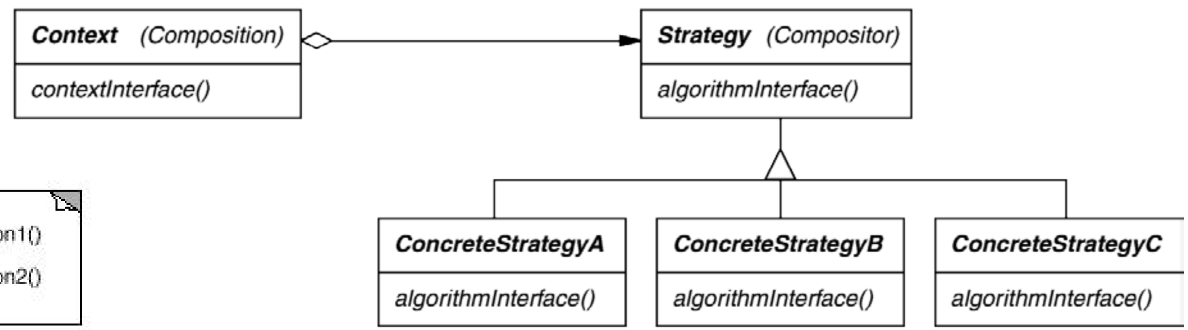
*Consequence: Small HW/SW changes have big (negative) impact on system QoS & maintenance*

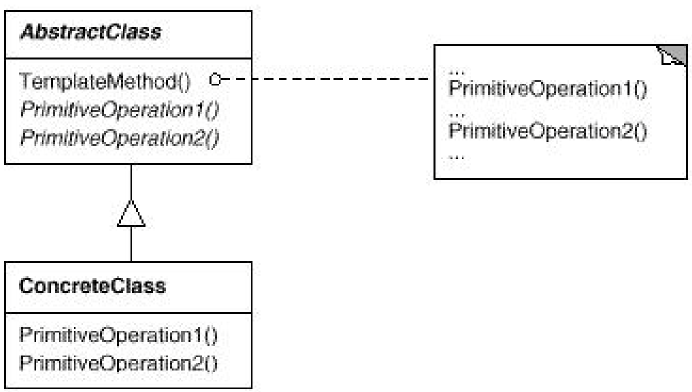
# Motivation for Frameworks


- ***Frameworks*** factors out many reusable general-purpose & domain-specific services from traditional DRE application responsibility
- Essential for ***product-line architectures (PLAs)***
- Product-lines & frameworks offer many configuration opportunities
  - e.g., component distribution/deployment, OS, protocols, algorithms, etc.

# Categories of OO Frameworks

- *White-box frameworks* are reused by subclassing, which usually requires understanding the implementation of the framework to some degree
- *Black-box framework* is reused by parameterizing & assembling framework objects, thereby hiding their implementation from users
- Each category of OO framework uses different sets of patterns, e.g.:




  – *White-box frameworks* rely heavily on inheritance-based patterns, such as Template Method & State

  – *Black-box frameworks* reply heavily on object composition patterns, such as Strategy & Decorator

Many frameworks fall in between white-box & black-box categories

# Commonality & Variability Analysis in Frameworks

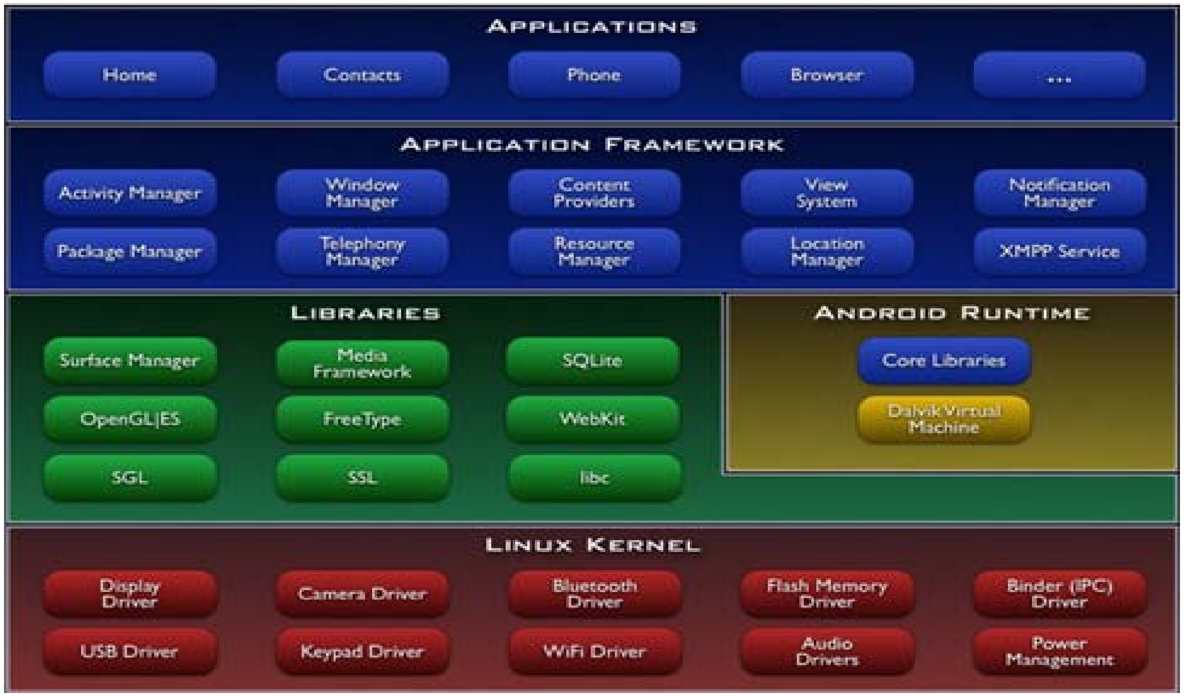
- Framework characteristics are captured via *Scope, Commonalities, & Variabilities (SCV) analysis*
  - This process can be applied to identify commonalities & variabilities in a domain to guide development of a framework
- Applying SCV to Android smartphones
  - Scope defines the domain & context of the framework
    - Component architecture, object-oriented application frameworks, & associated components, e.g., GPS, Network, & Display

Reusable Application Components

Reusable Architecture Framework

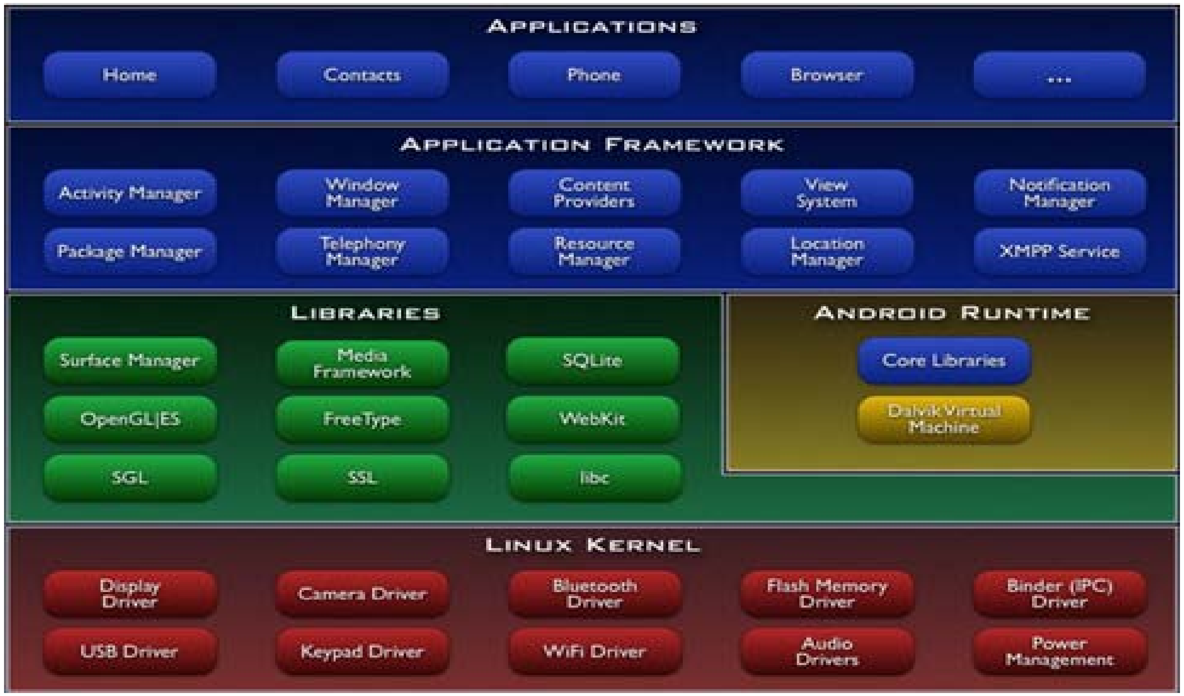
# Applying SCV to an Android Framework

- *Commonalities* describe the attributes that are common across all members of the framework
  - Common object-oriented frameworks & set of component types
    - e.g., Activities, Services, Content Providers, & Display components
  - Common middleware infrastructure
    - e.g., Intents framework, Binder, etc.

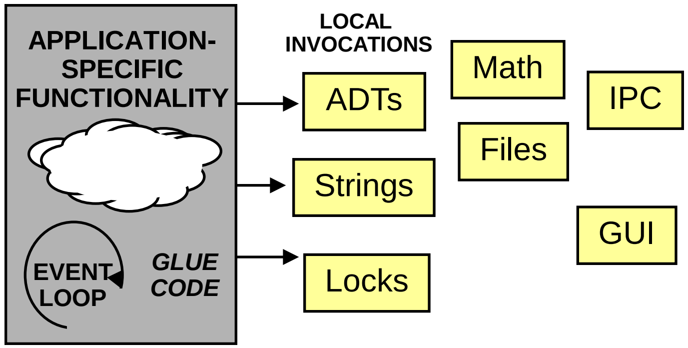


# Applying SCV to an Android Framework

- ***Variabilities*** describe the attributes unique to the different members of the framework
  - Product-dependent component implementations, e.g., Motorola, HTC, Samsug
  - Product-dependent component connections
  - Product-dependent component assemblies (e.g., CDMA vs. GSM in different countries)
  - Different hardware, OS, & network/bus configurations

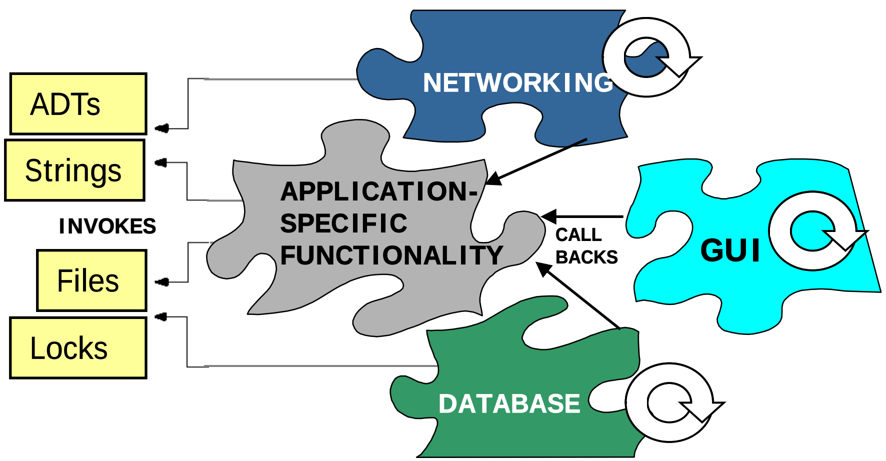


# Comparing Reuse Techniques



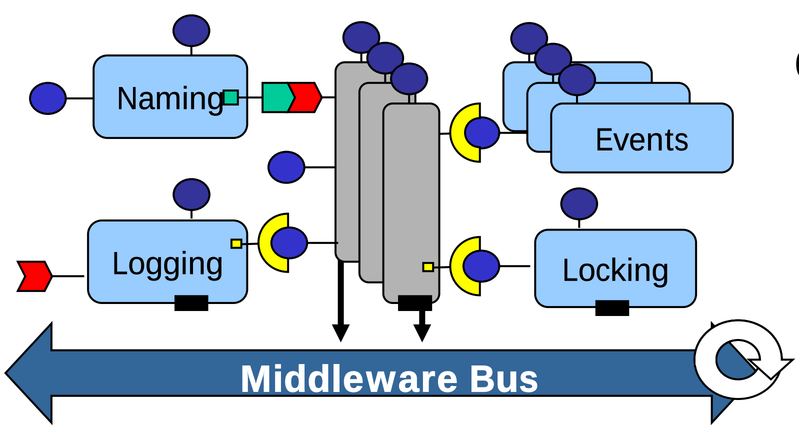

## Class Library (& STL) Architecture

- A ***class*** is an implementation unit in an OO programming language, i.e., a reusable ***type*** that often implements ***patterns***
- Classes in class libraries are typically *passive*




## Framework Architecture

- A ***framework*** is an integrated set of classes that collaborate to form a reusable architecture for a family of applications
- Frameworks implement ***pattern languages***




## Component & Service-Oriented Architecture

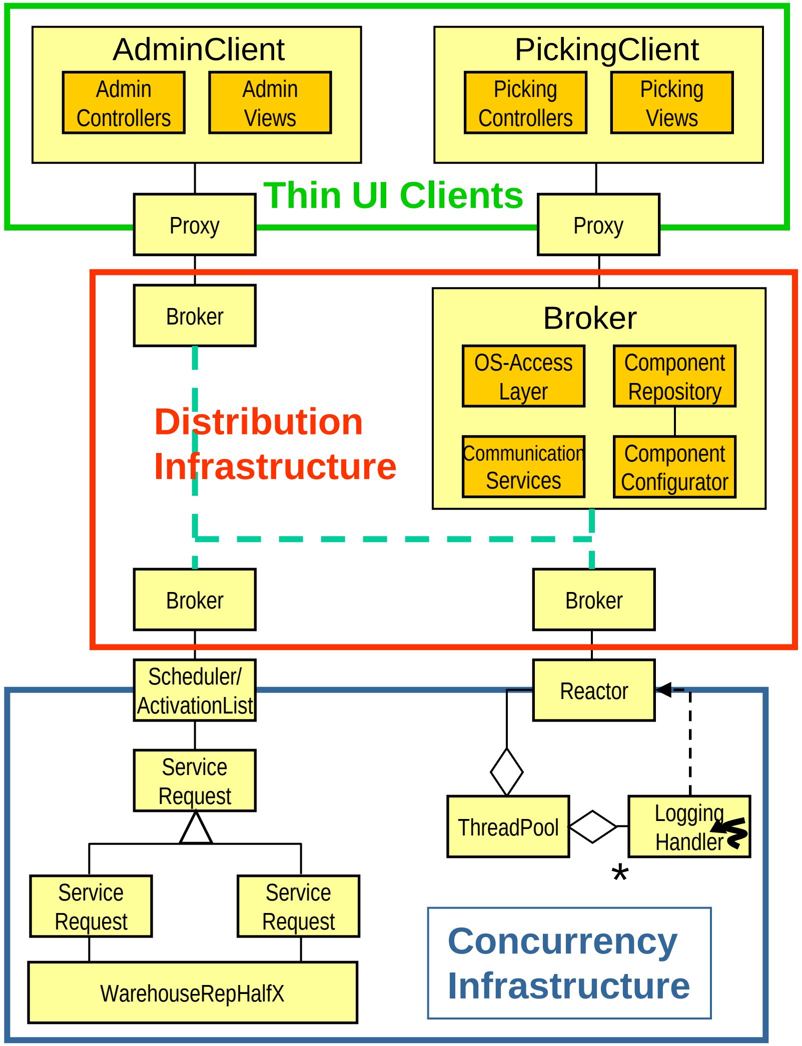
- A ***component*** is an encapsulation unit with one or more interfaces that provide clients with access to its services
- Components can be deployed & configured via ***assemblies***

# Taxonomy of Reuse Techniques

| Class Libraries | Frameworks | Components |
|---|---|---|
| Micro-level | Meso-level | Macro-level |
| Stand-alone language entities | "Semi-complete" applications | Stand-alone composition entities |
| Domain-independent | Domain-specific | Domain-specific or Domain-independent |
| Borrow caller's thread | Inversion of control | Borrow caller's thread |

# Benefits of Frameworks

- **Design reuse**
  - e.g., by guiding application developers through the steps necessary to ensure successful creation & deployment of software



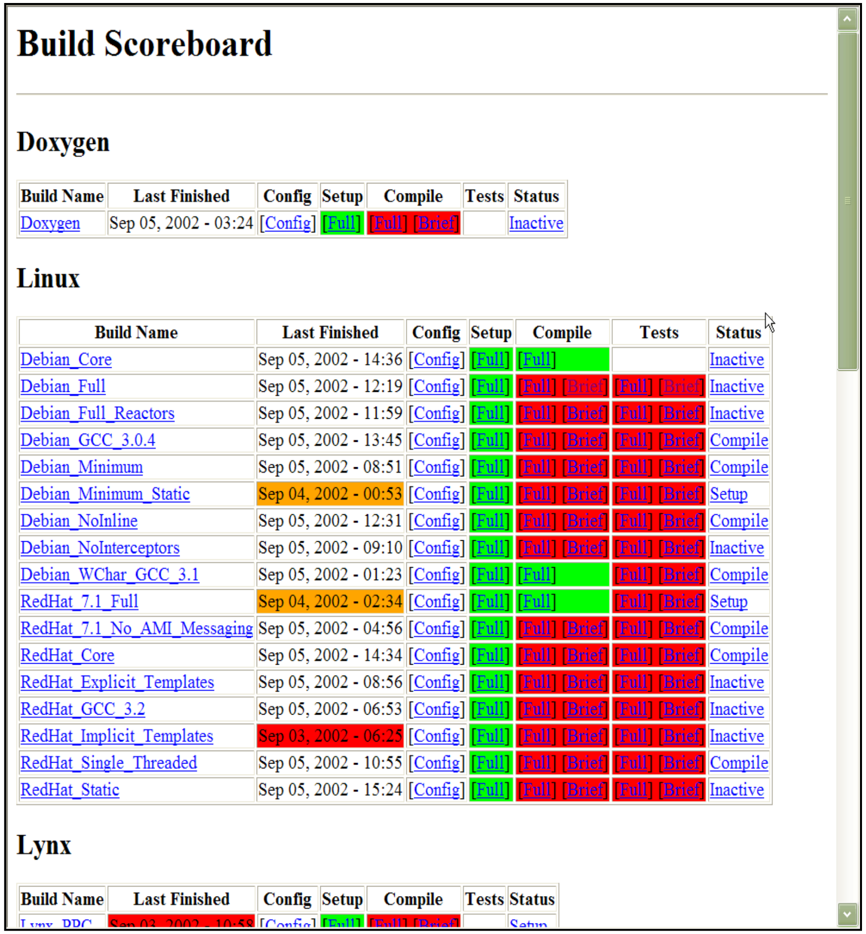
# Benefits of Frameworks

- **Design reuse**
  - e.g., by guiding application developers through the steps necessary to ensure successful creation & deployment of software
- **Implementation reuse**
  - e.g., by amortizing software lifecycle costs & leveraging previous development & optimization efforts

```
package org.apache.tomcat.session;

import org.apache.tomcat.core.*;
import org.apache.tomcat.util.StringManager;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * Core implementation of a server session
 *
 * @author James Duncan Davidson [duncan@eng.sun.com]
 * @author James Todd [gonzo@eng.sun.com]
 */

public class ServerSession {

    private StringManager sm =
        StringManager.getManager("org.apache.tomcat.session");
    private Hashtable values = new Hashtable();
    private Hashtable appSessions = new Hashtable();
    private String id;
    private long creationTime = System.currentTimeMillis();;
    private long thisAccessTime = creationTime;
private int inactiveInterval = -1;

    ServerSession(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }

    public long getCreationTime() {
        return creationTime;
    }

    public ApplicationSession getApplicationSession(Context context,
        boolean create) {
        ApplicationSession appSession =
            (ApplicationSession)appSessions.get(context);

        if (appSession == null && create) {

            // XXX
            // sync to ensure valid?

            appSession = new ApplicationSession(id, this, context);
            appSessions.put(context, appSession);
        }

        // XXX
        // make sure that we haven't gone over the end of our
        // inactive interval -- if so, invalidate & create
        // a new appSession

        return appSession;
    }

    void removeApplicationSession(Context context) {
        appSessions.remove(context);
    }
```

# Benefits of Frameworks

- **Design reuse**
  - e.g., by guiding application developers through the steps necessary to ensure successful creation & deployment of software
- **Implementation reuse**
  - e.g., by amortizing software lifecycle costs & leveraging previous development & optimization efforts
- **Validation reuse**
  - e.g., by amortizing the efforts of validating application- & platform-independent portions of software, thereby enhancing software reliability & scalability



## Build Scoreboard

### Doxygen

| Build Name | Last Finished | Config | Setup | Compile | Tests | Status |
|---|---|---|---|---|---|---|
| Doxygen | Sep 05, 2002 - 03:24 | [Config] | [Full] | [Full] [Brief] | | Inactive |

### Linux

| Build Name | Last Finished | Config | Setup | Compile | Tests | Status |
|---|---|---|---|---|---|---|
| Debian_Core | Sep 05, 2002 - 14:36 | [Config] | [Full] | [Full] | | Inactive |
| Debian_Full | Sep 05, 2002 - 12:19 | [Config] | [Full] | [Full] [Brief] | [Full] [Brief] | Inactive |
| Debian_Full_Reactors | Sep 05, 2002 - 11:59 | [Config] | [Full] | [Full] [Brief] | [Full] [Brief] | Inactive |
| Debian_GCC_3.0.4 | Sep 05, 2002 - 13:45 | [Config] | [Full] | [Full] [Brief] | [Full] [Brief] | Compile |
| Debian_Minimum | Sep 05, 2002 - 08:51 | [Config] | [Full] | [Full] [Brief] | [Full] [Brief] | Compile |
| Debian_Minimum_Static | Sep 04, 2002 - 00:53 | [Config] | [Full] | [Full] [Brief] | [Full] [Brief] | Setup |
| Debian_NoInline | Sep 05, 2002 - 12:31 | [Config] | [Full] | [Full] [Brief] | [Full] [Brief] | Compile |
| Debian_NoInterceptors | Sep 05, 2002 - 09:10 | [Config] | [Full] | [Full] [Brief] | [Full] [Brief] | Inactive |
| Debian_WChar_GCC_3.1 | Sep 05, 2002 - 01:23 | [Config] | [Full] | [Full] | [Full] [Brief] | Compile |
| RedHat_7.1_Full | Sep 04, 2002 - 02:34 | [Config] | [Full] | [Full] | [Full] [Brief] | Setup |
| RedHat_7.1_No_AMI_Messaging | Sep 05, 2002 - 04:56 | [Config] | [Full] | [Full] [Brief] | [Full] [Brief] | Compile |
| RedHat_Core | Sep 05, 2002 - 14:34 | [Config] | [Full] | [Full] [Brief] | [Full] [Brief] | Compile |
| RedHat_Explicit_Templates | Sep 05, 2002 - 08:56 | [Config] | [Full] | [Full] [Brief] | [Full] [Brief] | Inactive |
| RedHat_GCC_3.2 | Sep 05, 2002 - 06:53 | [Config] | [Full] | [Full] [Brief] | [Full] [Brief] | Inactive |
| RedHat_Implicit_Templates | Sep 03, 2002 - 06:25 | [Config] | [Full] | [Full] [Brief] | [Full] [Brief] | Inactive |
| RedHat_Single_Threaded | Sep 05, 2002 - 10:55 | [Config] | [Full] | [Full] [Brief] | [Full] [Brief] | Compile |
| RedHat_Static | Sep 05, 2002 - 15:24 | [Config] | [Full] | [Full] [Brief] | [Full] [Brief] | Inactive |

### Lynx

| Build Name | Last Finished | Config | Setup | Compile | Tests | Status |
|---|---|---|---|---|---|---|
| Lynx_PPC | Sep 03, 2002 - 10:58 | [Config] | [Full] | [Full] [Brief] | | Setup |

# Limitations of Frameworks

- Frameworks are powerful, but can be hard to use effectively (& even harder to create) for many application developers
  - Commonality & variability analysis requires significant domain knowledge & OO design/implementation expertise
- Significant time required to evaluate applicability & quality of a framework for a particular domain
- Debugging is tricky due to inversion of control
- V&V is tricky due to "late binding"
- May incur performance degradations due to extra (unnecessary) levels of indirection

www.cs.wustl.edu/ ~schmidt/PDF/Queue-04.pdf

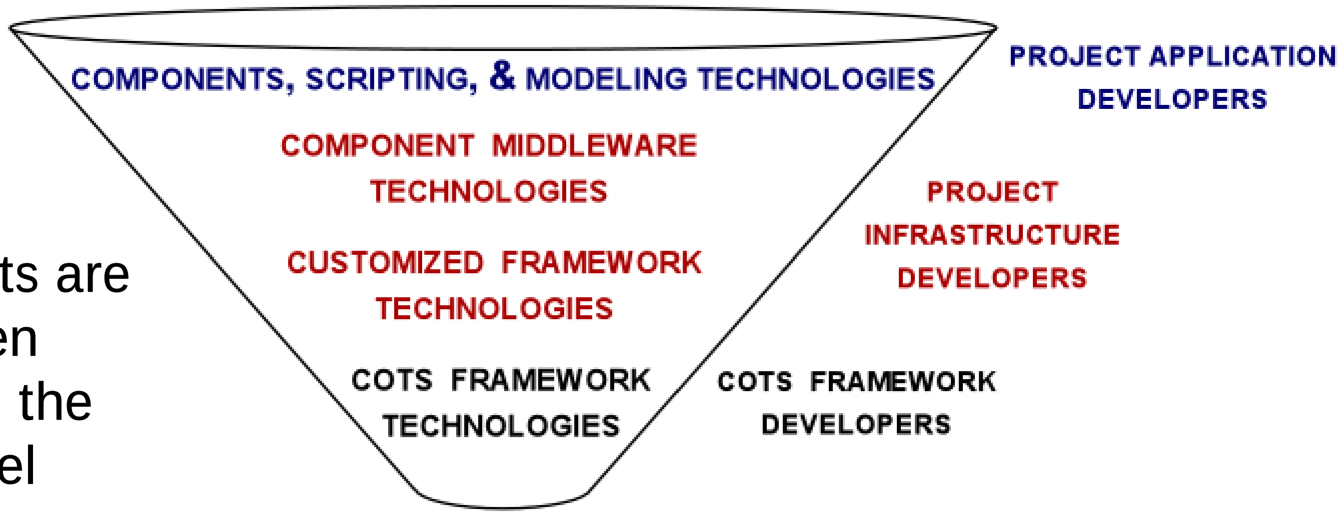
Many frameworks limitations can be addressed with knowledge of patterns!

# Using Frameworks Effectively

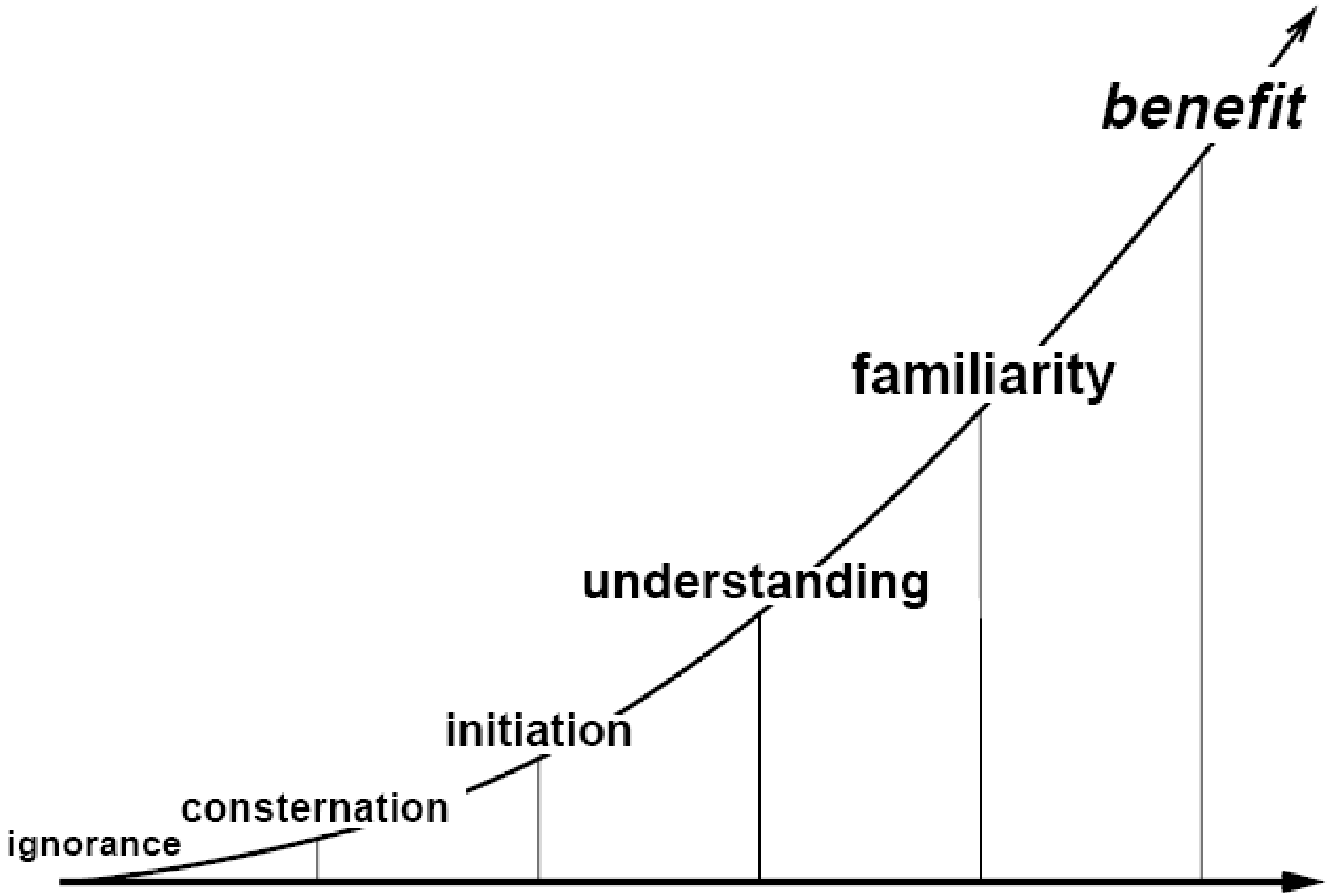
**Observations**

- Since frameworks are powerful—but but hard to develop & use effectively by application developers—it's often better to use & customize COTS frameworks than to develop in-house frameworks
- Classes/components/services are easier for application developers to use, but aren't as powerful or flexible as frameworks

Successful projects are therefore often organized using the "funnel" model



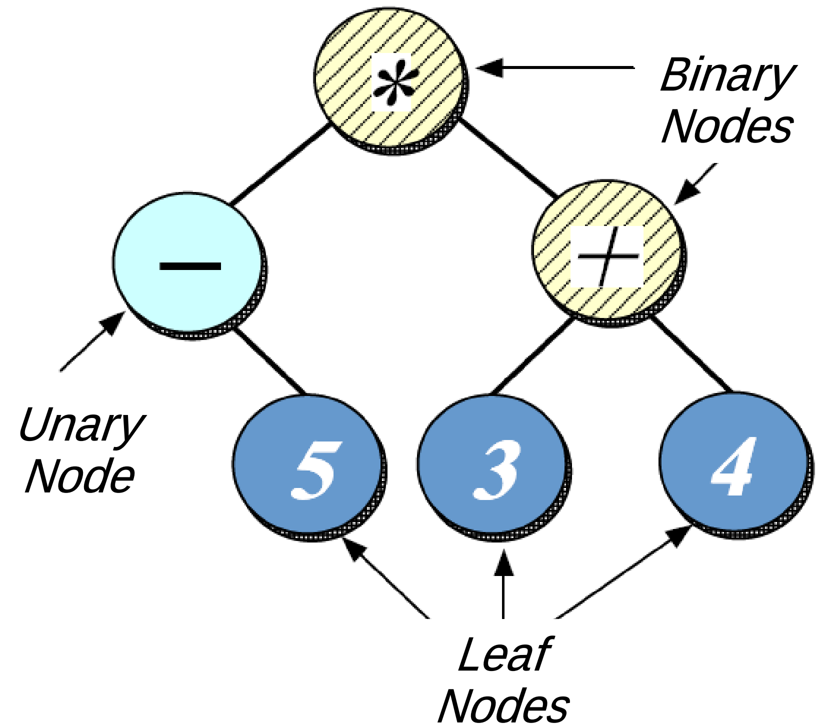
# Stages of Pattern & Framework Awareness



# Part II: Case Study: Expression Tree Application

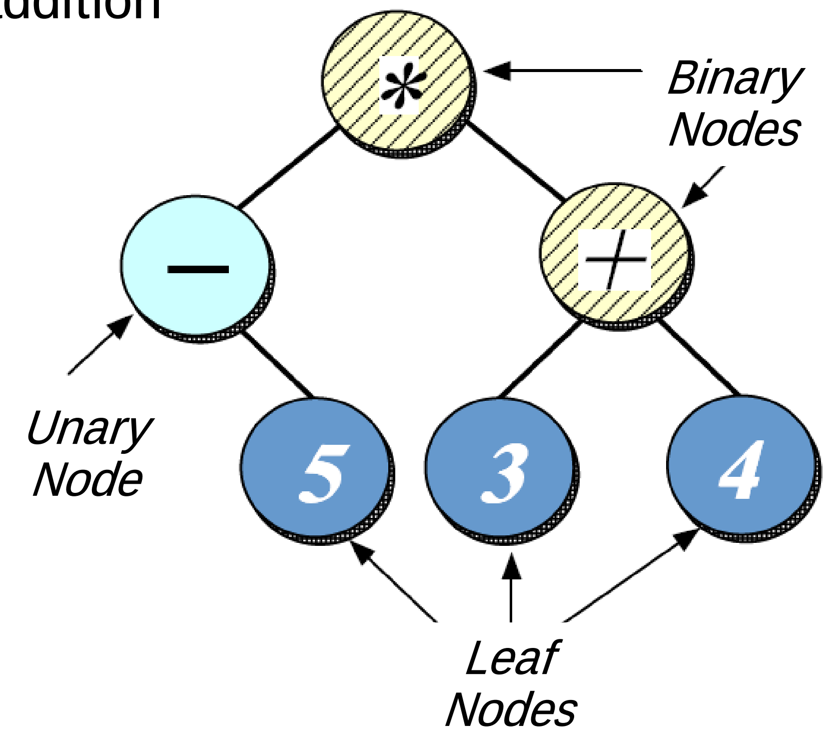
## Goals

- Develop an object-oriented expression tree evaluator program using *patterns* & *frameworks*
- Demonstrate commonality/variability analysis in the context of a concrete application example
- Illustrate how OO frameworks can be combined with the generic programming features of C++ & STL
- Compare/contrast OO & non-OO approaches



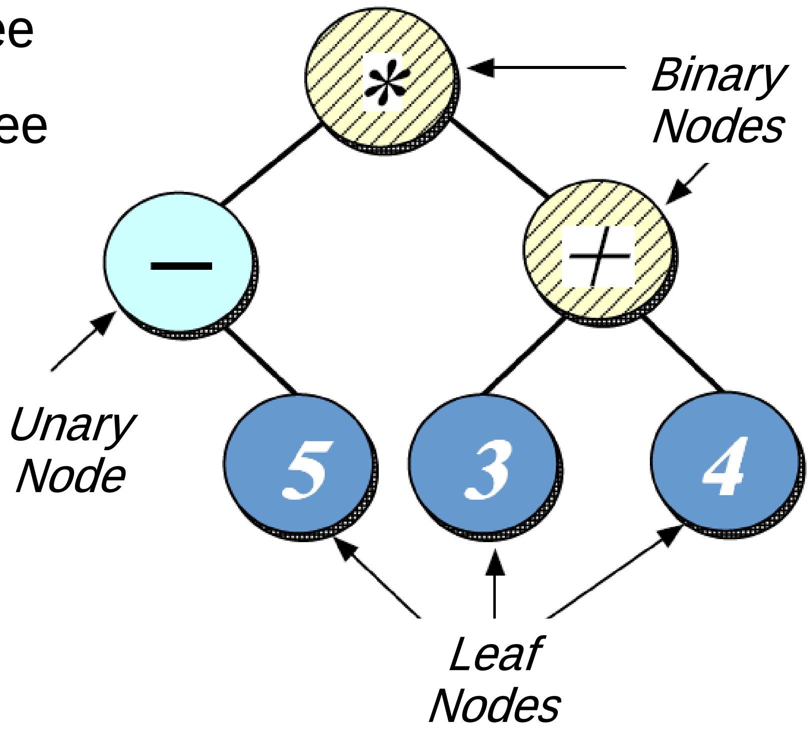
# Overview of Expression Tree Application

- Expression trees consist of nodes containing *operators* & *operands*
- Operators have different precedence levels, different associativities, & different arities, e.g.:
  - Multiplication takes precedence over addition
  - The multiplication operator has two arguments, whereas unary minus operator has only one
- Operands can be integers, doubles, variables, etc.
  - We'll just handle integers in this application
  - Application can be extended easily



# Overview of Expression Tree Application

- Trees may be "evaluated" via different traversal orders
  - e.g., in-order, post-order, pre-order, level-order
- The evaluation step may perform various operations, e.g.:
  - Print the contents of the expression tree
  - Return the "value" of the expression tree
  - Generate code
  - Perform semantic analysis & optimization
  - *etc.*



# Using the Expression Tree Application

- By default, the expression tree application can run in “succinct mode,” e.g.:

```
% tree-traversal
> 1+4*3/2
7
> (8/4) * 3 + 1
7
^D
```

- You can also run the expression tree application in “verbose mode,” e.g.:

```
% tree-traversal -v
format [in-order]
expr [expression]
print [in-order|pre-order|post-order|level-order]
eval [post-order]
quit
> format in-order
> expr 1+4*3/2
> eval post-order
7
> quit
```

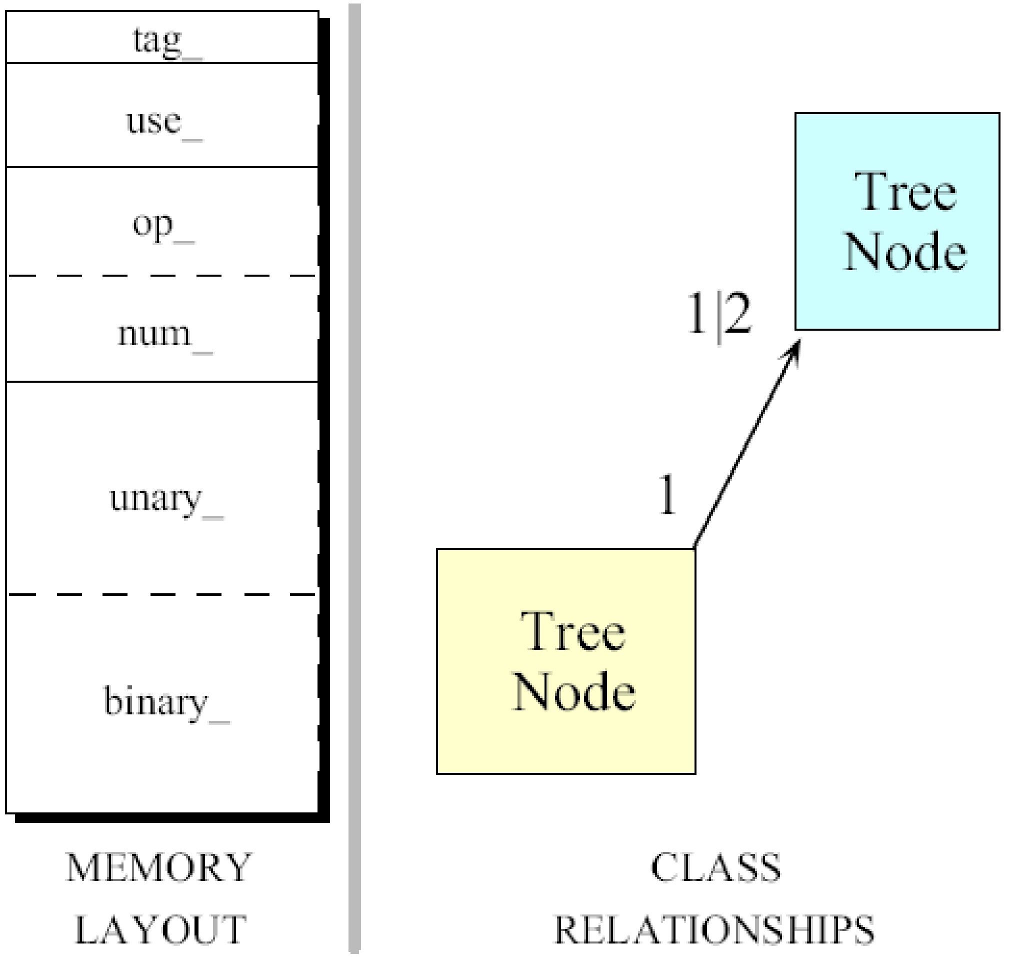
# How *Not* to Design an Expression Tree Application

A typical algorithmic-based solution for implementing expression trees uses a C struct/union to represent the main data structure

```
typedef struct Tree_Node {
  enum { NUM, UNARY, BINARY } tag_;
  short use_; /* reference count */
  union {
    char op_[2];
    int num_;
  } o;
#define num_ o.num_
#define op_  o.op_
  union {
    struct Tree_Node *unary_;
    struct { struct Tree_Node *l_, *r_;} binary_;
  } c;
#define unary_ c.unary_
#define binary_ c.binary_
} Tree_Node;
```

# How *Not* to Design an Expression Tree Application

Here's the memory layout & class diagram for a **`struct Tree_Node`**:



# How *Not* to Design an Expression Tree Application

A typical algorithmic implementation uses a switch statement & a recursive function to build & evaluate a tree, e.g.:

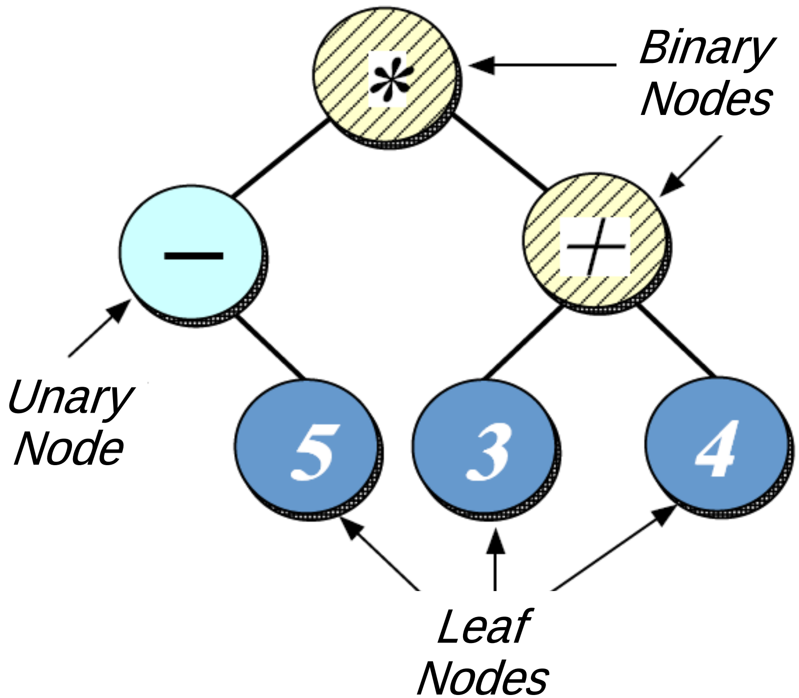
```
void print_tree (Tree_Node *root) {
  switch (root->tag_)
  case NUM: printf (“%d”, root->num_); break;
  case UNARY:
    printf ("(%s”, root->op_[0]);
    print_tree (root->unary_);
    printf (")"); break;
  case BINARY:
    printf ("(");
    print_tree (root->binary_.l_); // Recursive call
    printf (“%s”, root->op_[0]);
    print_tree (root->binary_.r_); // Recursive call
    printf (")"); break;
  default:
    printf ("error, unknown type ");
}
```

# Limitations with the Algorithmic Approach

- Little or no use of encapsulation: implementation details available to clients
- Incomplete modeling of the application domain, which results in
  - Tight coupling between nodes/edges in union representation
  - Complexity being in algorithms rather than the data structures
    - e.g., switch statements are used to select between various types of nodes in the expression trees
- Data structures are "passive" functions that do their work explicitly
- The program organization makes it hard to extend
  - e.g., small changes will ripple through the entire design & implementation
- Easy to make mistakes switching on type tags
- Wastes space by making worst-case assumptions wrt structs & unions

# An OO Alternative Using Patterns & Frameworks

- Start with OO modeling of the "expression tree" application domain



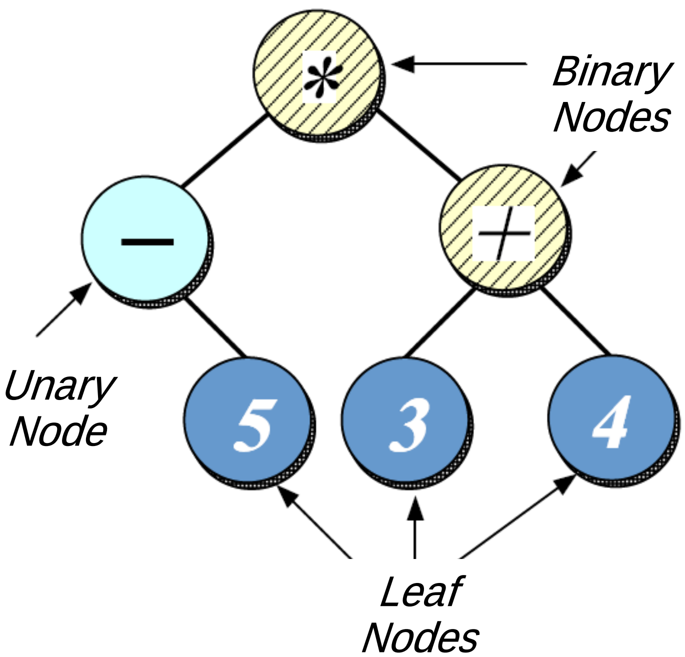

- Model a *tree* as a collection of *nodes*
- *Nodes* are represented in an inheritance hierarchy that captures the particular properties of each node
  - e.g., precedence levels, different associativities, & different arities

Application-dependent steps

- Conduct *commonality/variability analysis* (CVA) to determine stable interfaces & points of variability
- Apply patterns to guide design/implementation of framework
- Integrate w/C++ STL algorithms/containers when appropriate

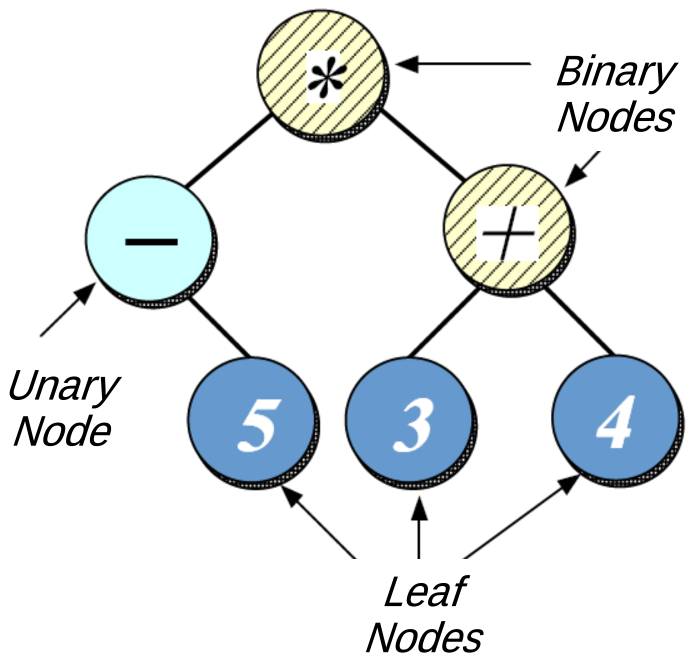
Application-independent steps

# Design Problems & Pattern-Oriented Solutions




| Design Problem | Pattern(s) |
|---|---|
| Expression tree structure | Composite |
| Encapsulating variability & simplifying memory management | Bridge |
| Tree printing & evaluation | Iterator & Visitor |
| Consolidating user operations | Command |
| Ensuring correct protocol for commands | State |
| Consolidating creation of Variabilities | Abstract Factory & Factory Method |
| Parsing expressions & creating expression tree | Interpreter & Builder |

# Design Problems & Pattern-Oriented Solutions




| Design Problem | Pattern(s) |
| --- | --- |
| Driving the application event flow | Reactor |
| Supporting multiple operation modes | Template Method & Strategy |
| Centralizing global objects effectively | Singleton |
| Implementing STL iterator semantics | Prototype |
| Eliminating loops via the STL `std::for_each()` algorithm | Adapter |
| Provide no-op commands | Null Object |

None of these patterns are restricted to expression tree applications...

# Managing Global Objects Effectively

**Goals:**

- Centralize access to objects that should be visible globally, e.g.:
  - command-line options that parameterize the behavior of the program
  - The object (Reactor) that drives the main event loop

**Constraints/forces:**

- Only need one instance of the command-line options & Reactor
- Global variables are problematic in C++

Verbose mode

```
% tree-traversal -v
format [in-order]
expr [expression]
print [in-order|pre-order|post-order|level-order]
eval [post-order]
quit
> format in-order
> expr 1+4*3/2
> eval post-order
7
> quit
```

Succinct mode

```
% tree-traversal
> 1+4*3/2
7
```

# Solution: Centralize Access to Global Instances

Rather than using global variables, create a central access point to global instances, e.g.:

```
int main (int argc, char *argv[])
{
  // Parse the command-line options.
  if (!Options::instance ()->parse_args (argc, argv))
    return 0;

  // Dynamically allocate the appropriate event handler
  // based on the command-line options.
  Expression_Tree_Event_Handler *tree_event_handler =
    Expression_Tree_Event_Handler::make_handler
      (Options::instance ()->verbose ());

  // Register event handler with the reactor.
  Reactor::instance ()->register_input_handler
    (tree_event_handler);
  // ...
```

# Singleton object creational

## Intent

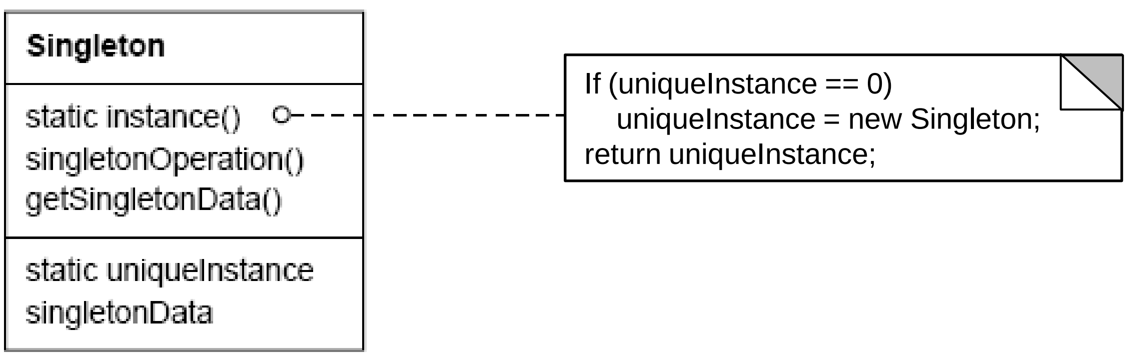
ensure a class only ever has one instance & provide a global point of access

## Applicability

– when there must be exactly one instance of a class & it must be accessible from a well-known access point

– when the sole instance should be extensible by subclassing & clients should be able to use an extended instance without modifying their code

## Structure



# Singleton object creational

### Consequences

+ reduces namespace pollution
+ makes it easy to change your mind & allow more than one instance
+ allow extension by subclassing

– same drawbacks of a global if misused
– implementation may be less efficient than a global
– concurrency/cache pitfalls & communication overhead

### Implementation

– static instance operation
– registering the singleton instance
– deleting singletons

### Known Uses

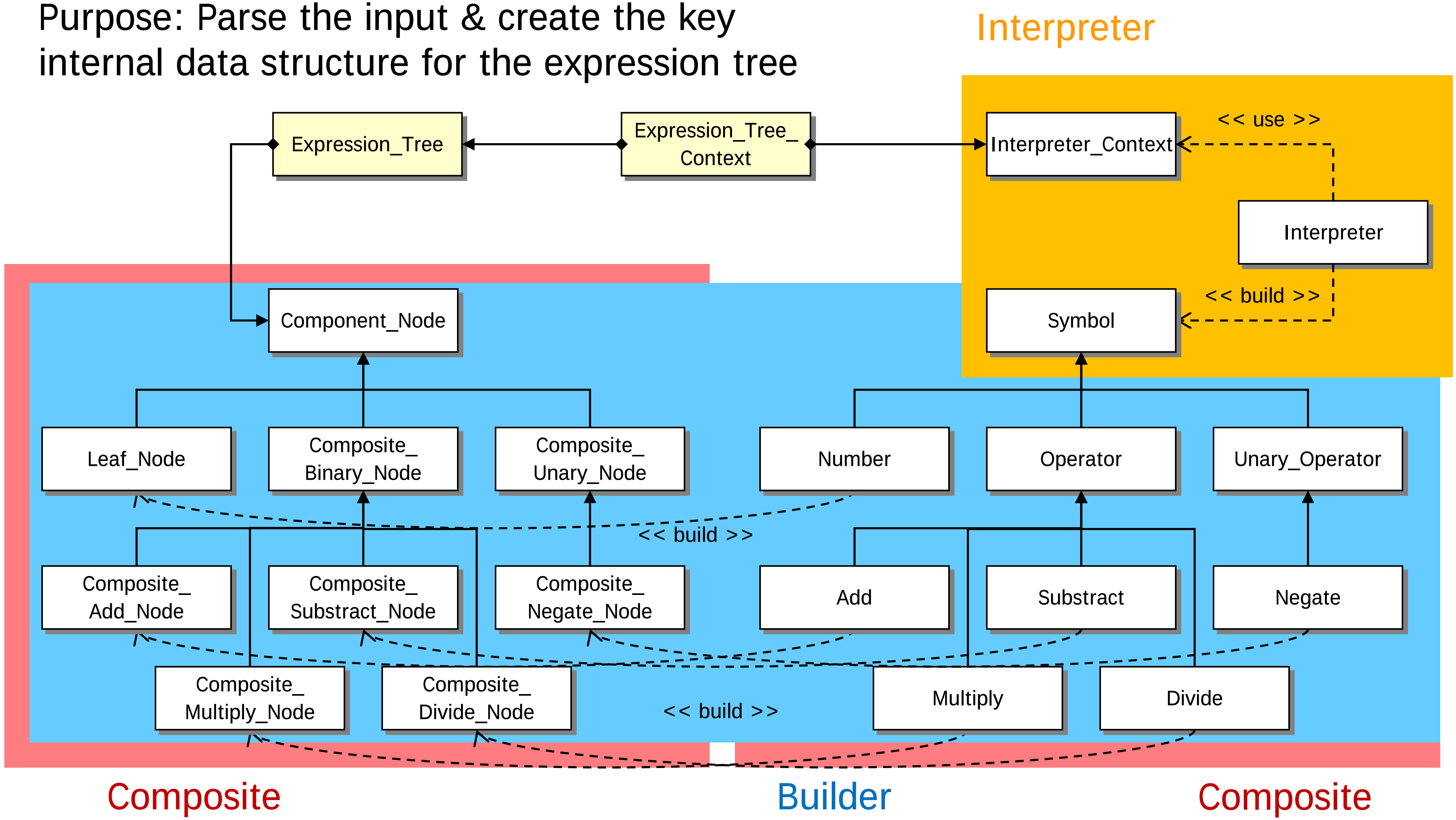
– Unidraw's Unidraw object
– Smalltalk-80 ChangeSet, the set of changes to code
– InterViews Session object

### See Also

– Double-Checked Locking Optimization pattern from POSA2
– "To Kill a Singleton" **`www.research.ibm.com/designpatterns/pubs/ph-jun96.txt`**

# Overview of Tree Structure & Creation Patterns

Purpose: Parse the input & create the key internal data structure for the expression tree




Patterns (& pattern combos) help focus on “forests” rather than (just) “trees”

# Expression Tree Structure

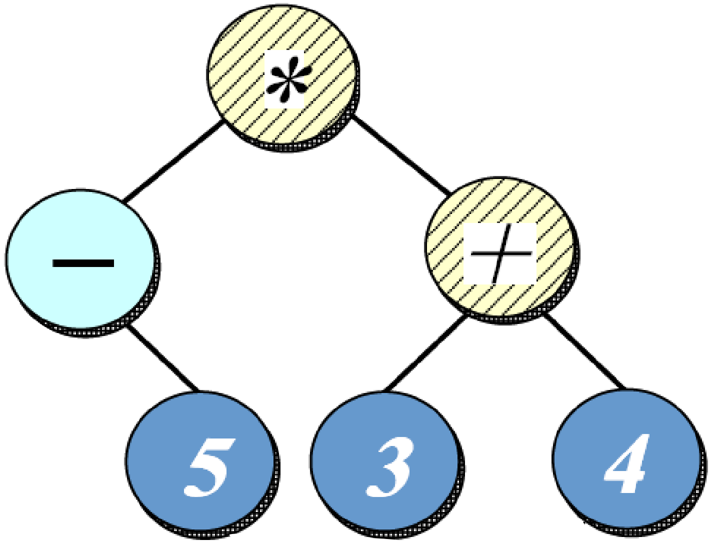
**Goals:**

– Support "physical" structure of expression tree

- e.g., binary/unary operators & operators

– Provide "hook" for enabling arbitrary operations on tree nodes

- Via Visitor pattern (described later)

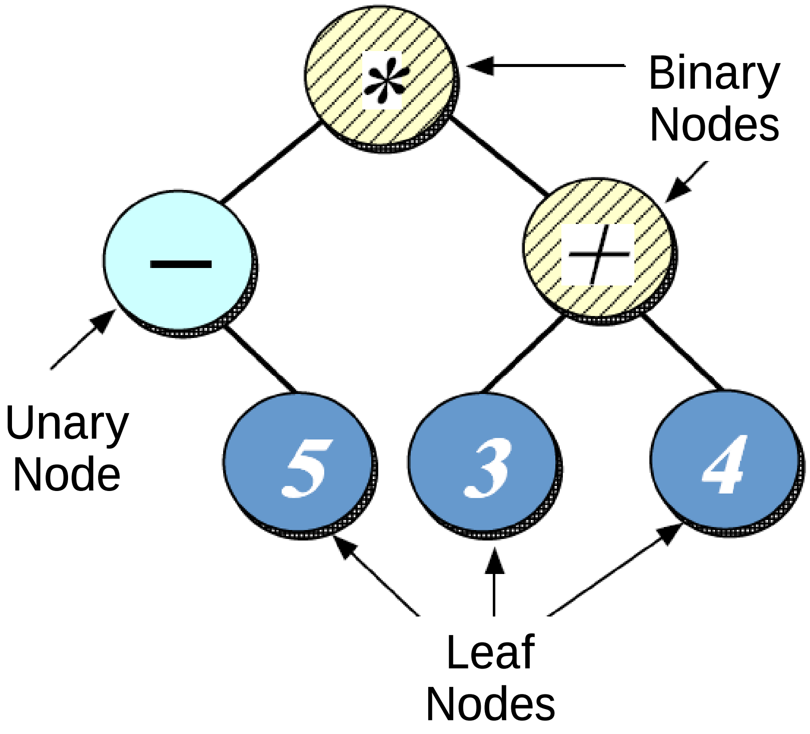



**Constraints/forces:**

– Treat operators & operands uniformly

– No distinction between one vs. many to avoid/minimize special cases

# Solution: Recursive Structure




- Model a *tree* as a recursive collection of *nodes*
- *Nodes* are represented in an inheritance hierarchy that captures the particular properties of each node
  - e.g., precedence levels, different associativities, & different arities
- Binary nodes recursively contain two other child nodes; unary nodes recursively contain one other child node

# Component_Node

Abstract base class for composable expression tree node objects

**Interface**:

virtual ~Component_Node (void)=0
virtual int item (void) const
virtual Component_Node * left (void) const
virtual Component_Node * right (void) const
virtual void accept (Visitor &visitor) const

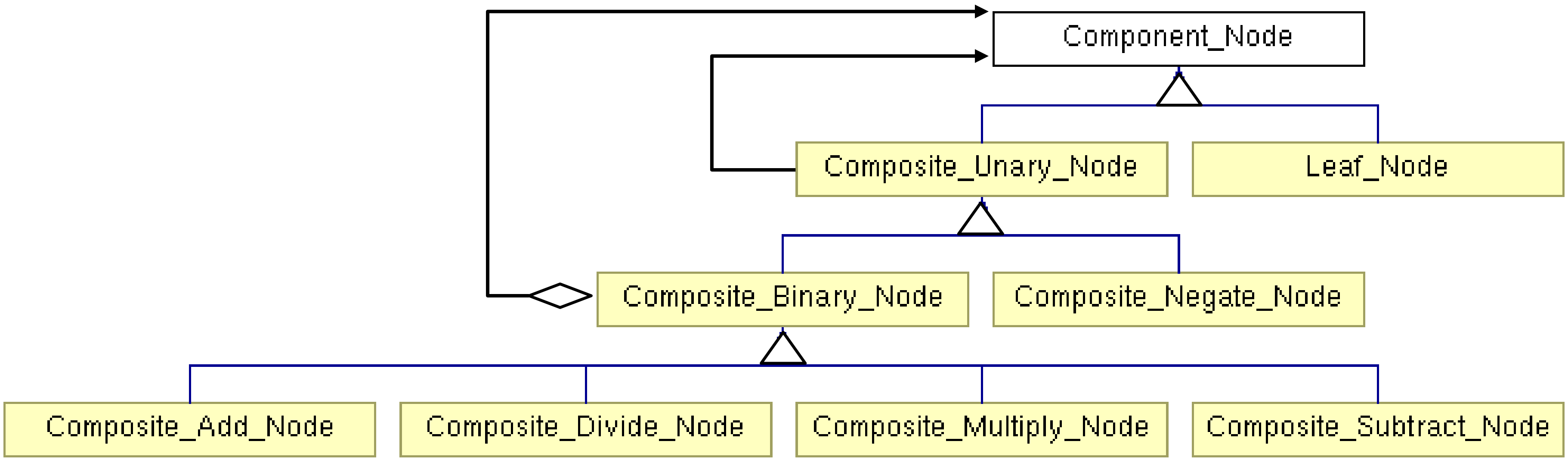
**Subclasses**:

**Leaf_Node**, **Composite_Unary_Node**, **Composite_Binary_Node**, etc.

**Commonality**: base class interface is used by all nodes in an expression tree

**Variability**: each subclass defines state & method implementations that are specific for the various types of nodes

# Component_Node Hierarchy




Note the inherent recursion in this hierarchy

- i.e., a **Composite_Binary_Node** *is a* **Component_Node** & a **Composite_Binary_Node** also *has* **Component_Node**s!

# Composite object structural

**Intent**

treat individual objects & multiple, recursively-composed objects uniformly

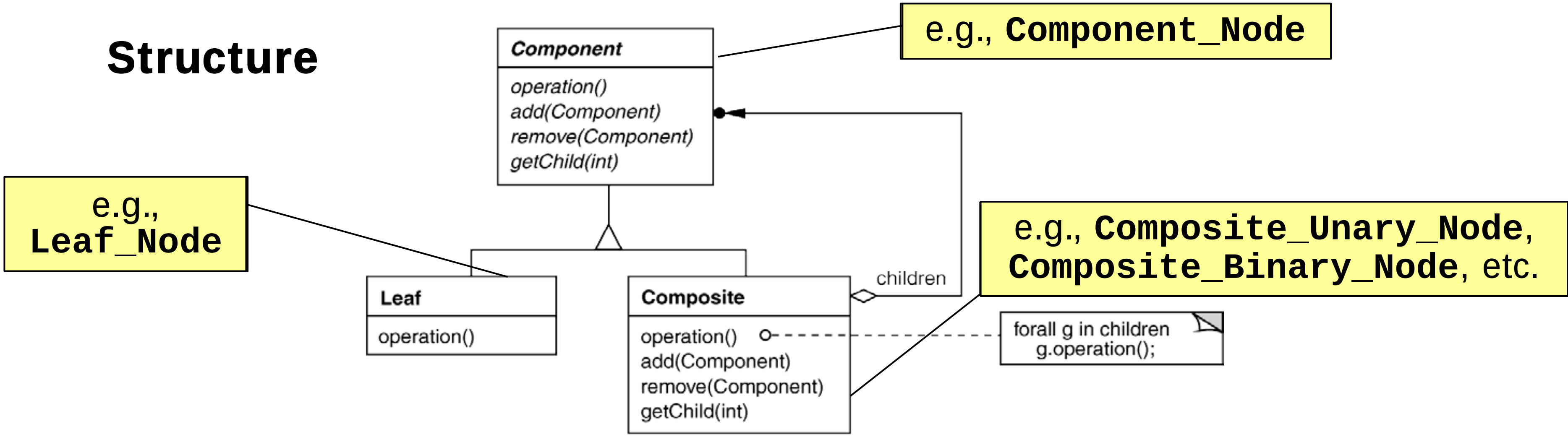
**Applicability**

objects must be composed recursively,

*and* no distinction between individual & composed elements,

*and* objects in structure can be treated uniformly

**Structure**



# Composite object structural

**Consequences**

+ uniformity: treat components the same regardless of complexity
+ extensibility: new Component subclasses work wherever old ones do
– overhead: might need prohibitive numbers of objects
– Awkward designs: may need to treat leaves as lobotomized composites in some cases

**Implementation**

– do Components know their parents?
– uniform interface for both leaves & composites?
– don't allocate storage for children in Component base class
– responsibility for deleting children

**Known Uses**

– ET++ Vobjects
– InterViews Glyphs, Styles
– Unidraw Components, MacroCommands
– Directory structures on UNIX & Windows
– Naming Contexts in CORBA
– MIME types in SOAP

# Parsing Expressions & Creating Expression Tree

**Goals:**

– Simplify & centralize the creation of all nodes in the composite expression tree

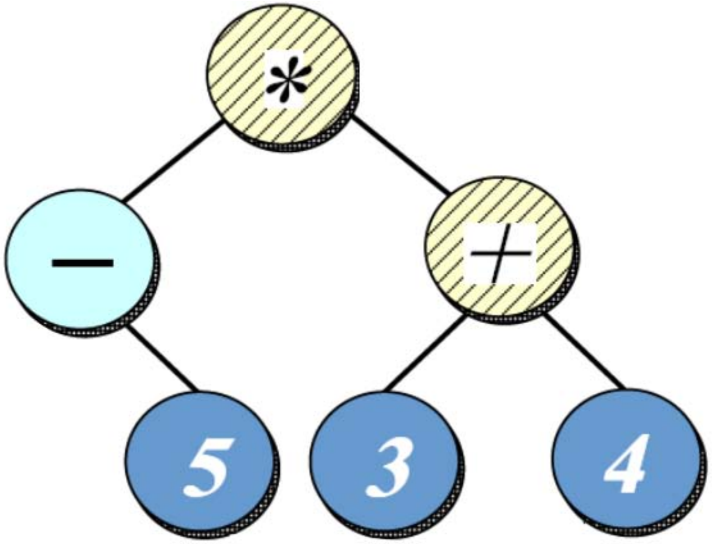
– Extensible for future types of expression orderings & optimizations

**Constraints/forces:**

– Don't recode existing clients when changes occur

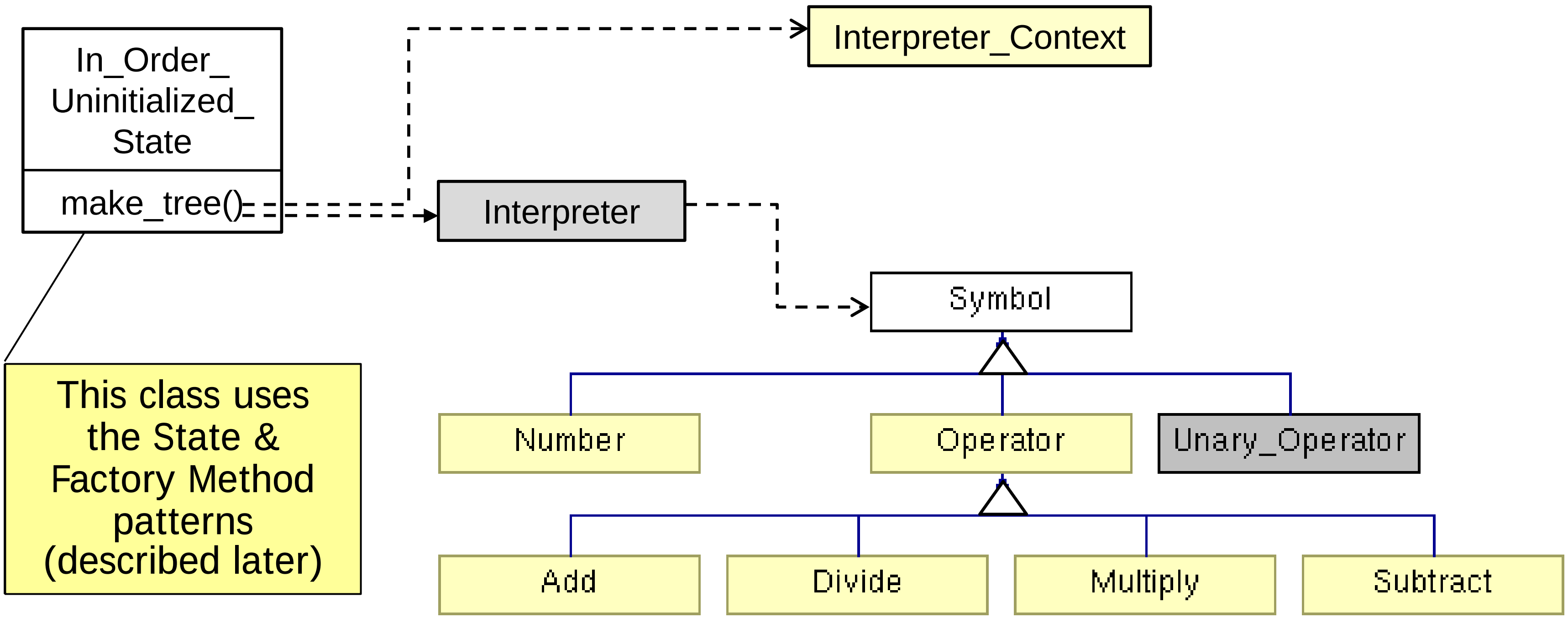
– Add new expressions without recompiling existing code

"in-order" expression = **-5*(3+4)**
"pre-order" expression = ***-5+34**
"post-order" expression = **5-34+***
"level-order" expression = ***-+534**



# Solution: Build Parse Tree Using Interpreter

- Use an *interpreter* to create a parse tree corresponding to input expression
- This parse tree is then traversed to build the appropriate type of nodes in the corresponding expression tree
  - We create the entire parse tree to enable future optimizations



# Interpreter

Parses expressions into parse tree & generate corresponding expression tree

**Interface:**

```
                      Interpreter (void)
              virtual ~Interpreter (void)
Expression_Tree interpret (Interpreter_Context &context,
                           const std::string &input)
```

uses

```
        Interpreter_Context (void)
       ~Interpreter_Context (void)
   int get (std::string variable)
  void set (std::string variable, int value)
  void print (void)
  void reset (void)
```

creates

```
                          Symbol (Symbol *left,
                                  Symbol *right)
                  virtual ~Symbol (void)
              virtual int precedence (void)=0
virtual Component_Node * build (void)=0
```

**Commonality**: Provides a common interface for parsing expression input & building parse trees & expression trees

**Variability**: The structure of the parse trees & expression trees can vary depending on the format & contents of the expression input

# Interpreter class behavioral

## Intent

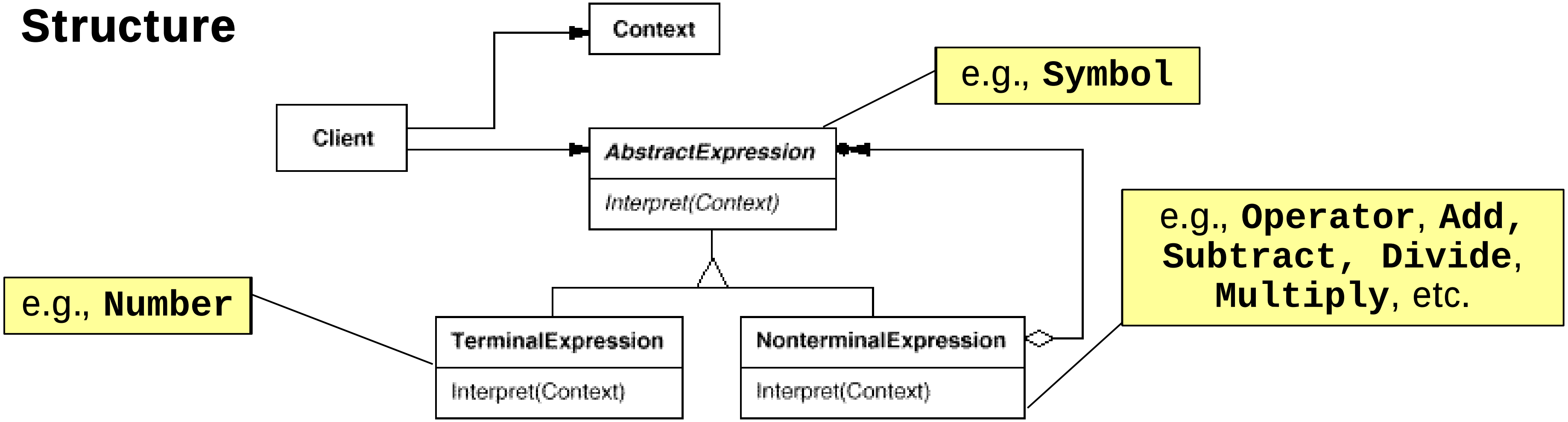
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language

## Applicability

– When the grammar is simple & relatively stable

– Efficiency is not a critical concern

## Structure



# Interpreter class behavioral

## Consequences

+ Simple grammars are easy to change & extend, e.g., all rules represented by distinct classes in an orderly manner

+ Adding another rule adds another class

– Complex grammars are hard to implement & maintain, e.g., more interdependent rules yield more interdependent classes

## Implementation

- Express the language rules, one per class
- Alternations, repetitions, or sequences expressed as *nonterminal expresssions*
- Literal translations expressed as *terminal expressions*
- Create interpret method to lead the context through the interpretation classes

## Known Uses

- Text editors &Web browsers use Interpreter to lay out documents & check spelling
- For example, an equation in TeX is represented as a tree where internal nodes are operators, e.g. square root, & leaves are variables

# Builder object creational

## Intent

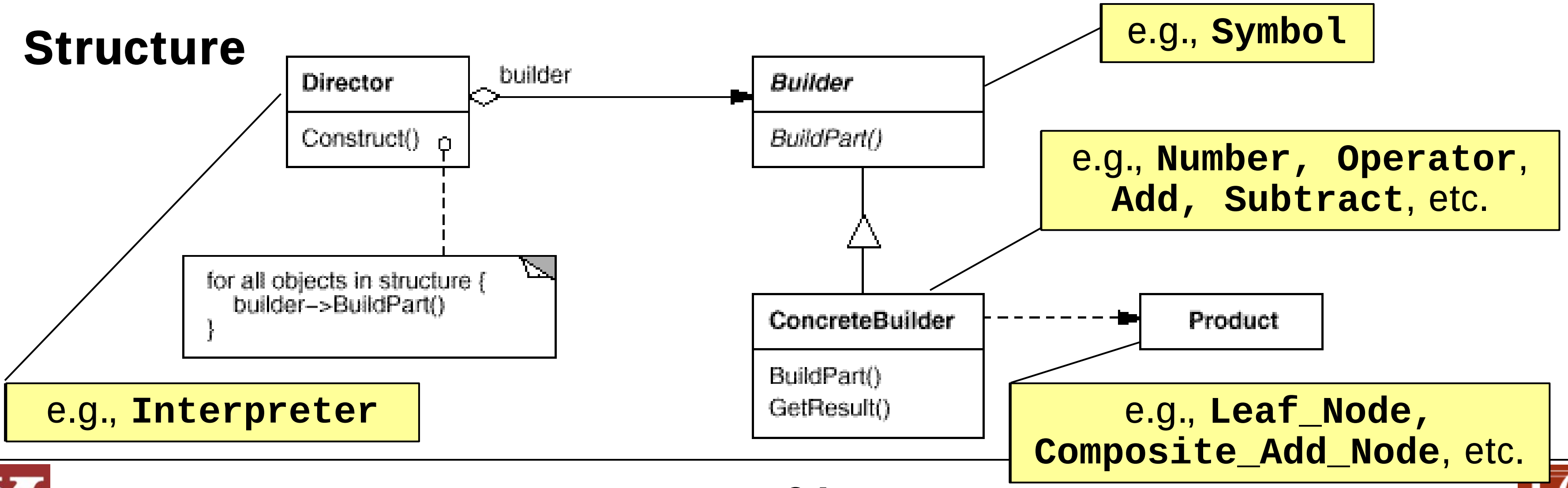
Separate the construction of a complex object from its representation so that the same construction process can create different representations

## Applicability

– Need to isolate knowledge of the creation of a complex object from its parts

– Need to allow different implementations/interfaces of an object's parts

## Structure



# Builder object creational

## Consequences

+ Can vary a product's internal representation
+ Isolates code for construction & representation
+ Finer control over the construction process
– May involve a lot of classes

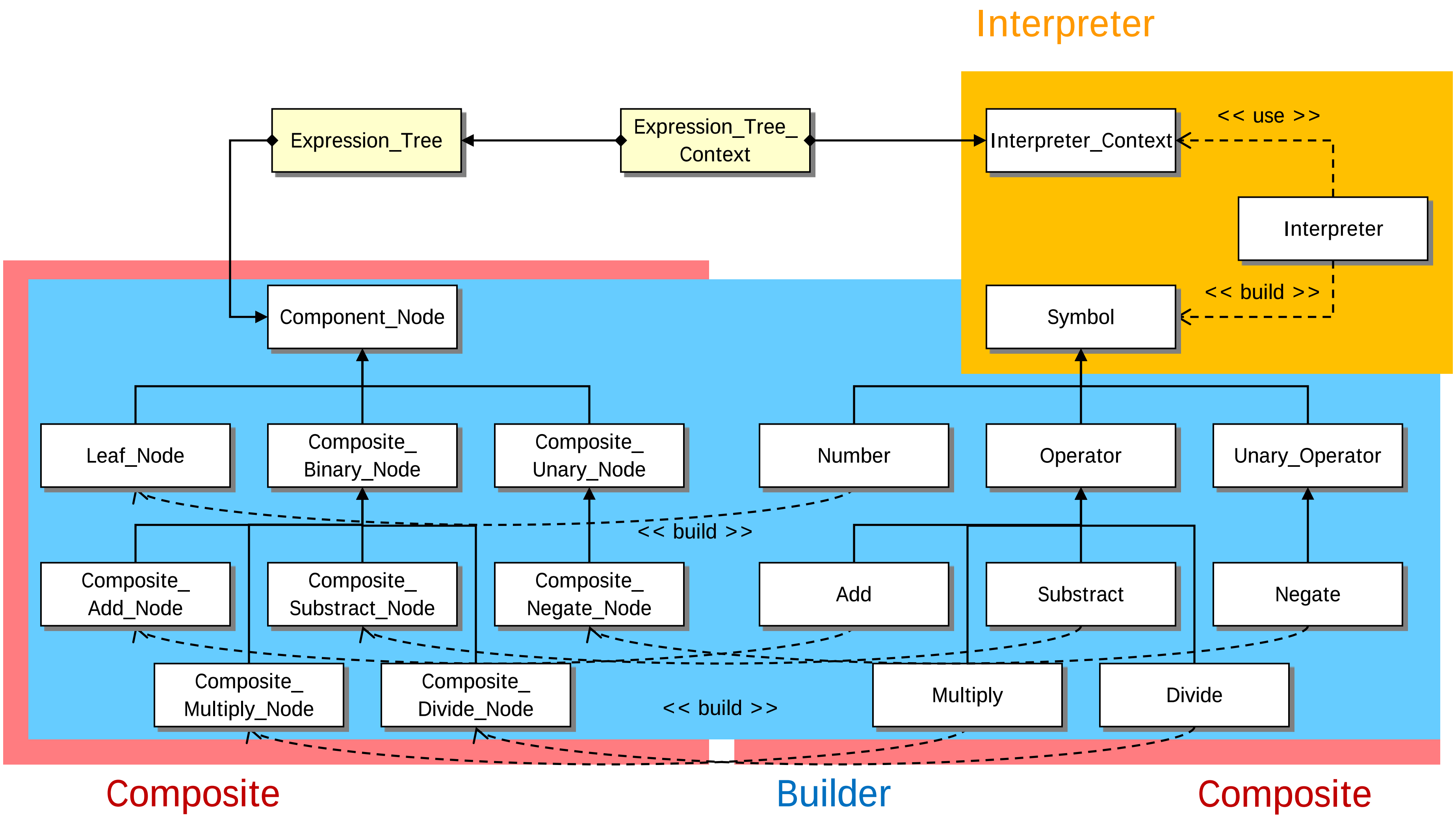
## Implementation

- The Builder pattern is basically a Factory pattern with a mission
- A Builder pattern implementation exposes itself as a factory, but goes beyond the factory implementation in that various implementations are wired together

## Known Uses

- ACE Service Configurator framework

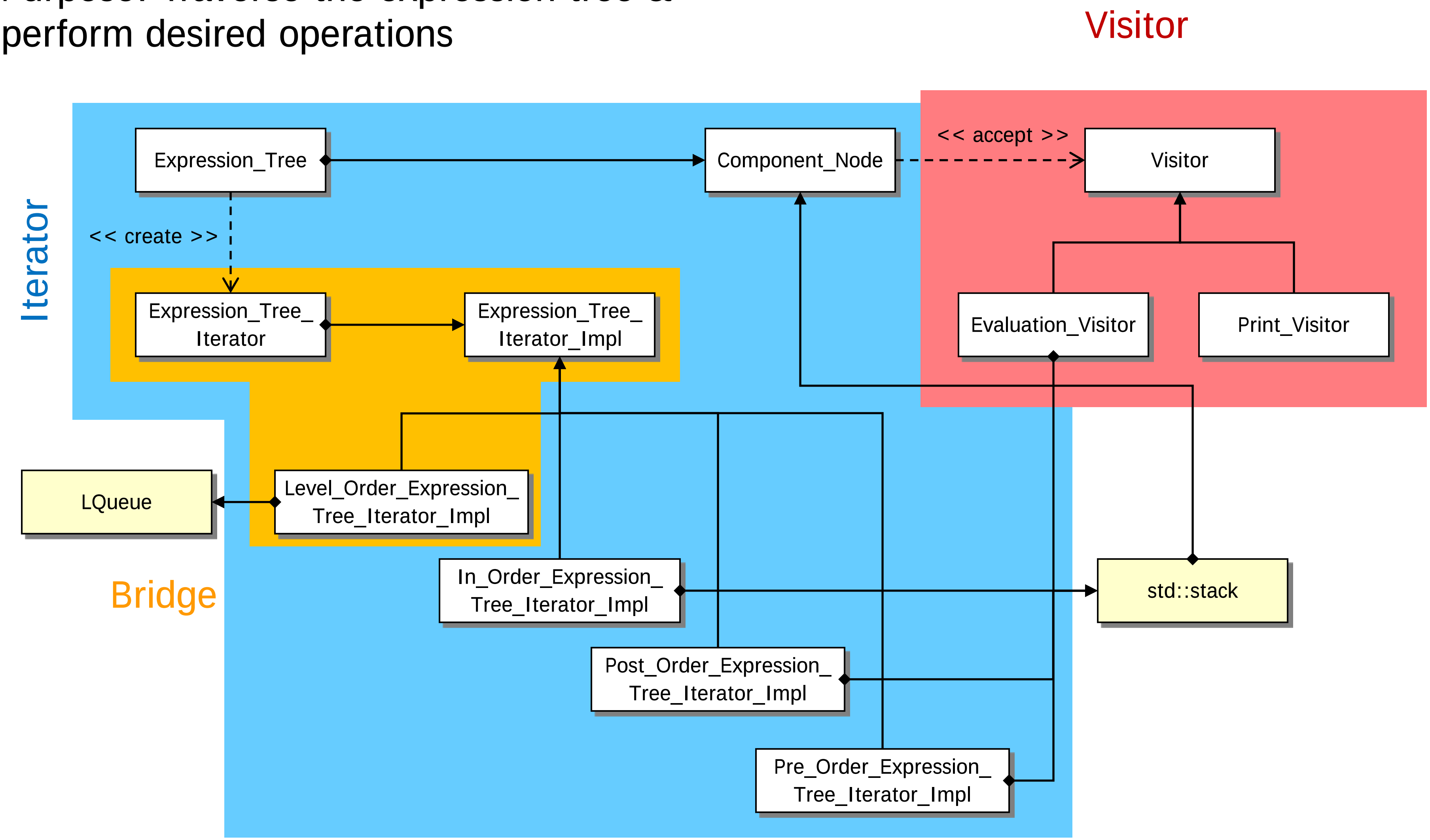
# Summary of Tree Structure & Creation Patterns




There are many classes, but only a handful of patterns involved in this design

# Overview of Tree Traversal Patterns

Purpose: Traverse the expression tree & perform desired operations




Patterns extensibly decouple structure of expression tree from operations on it

# Encapsulating Variability & Simplifying Memory Managment

**Goals**

– Hide many sources of variability in expression tree construction & use

– Simplify C++ memory management, i.e., minimize use of new/delete in application code

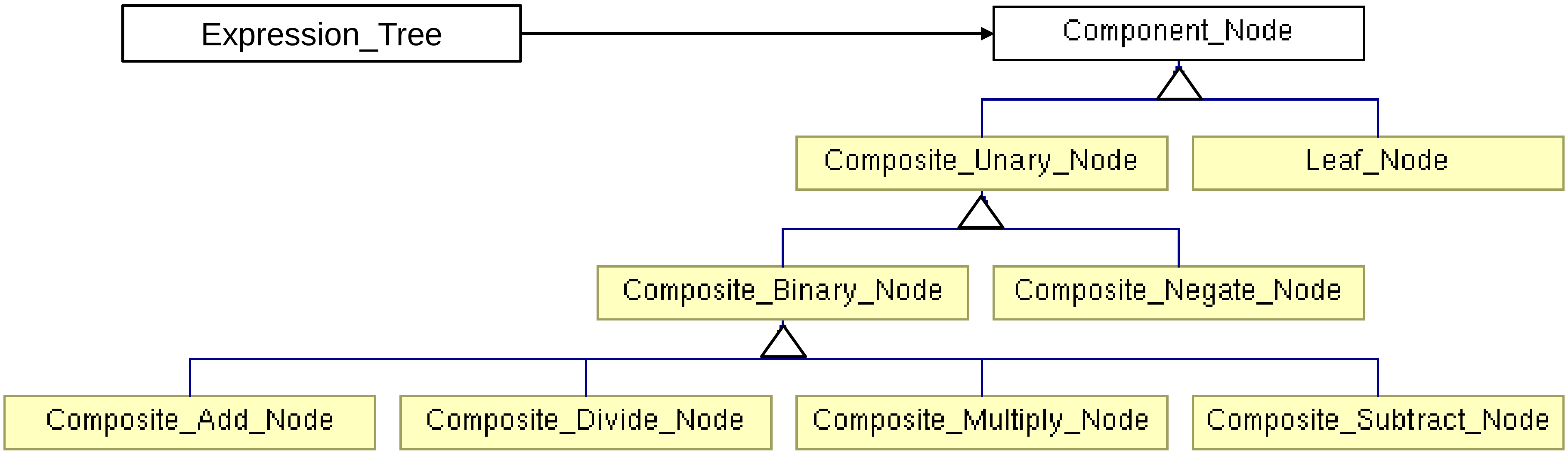
**Constraints/forces:**

– Must account for the fact that STL algorithms & iterators have "value semantics"

```
for (Expression_Tree::iterator iter = tree.begin ();
     iter != tree.end ();
     ++iter)
  (*iter).accept (print_visitor);
```

– Must ensure that exceptions don't cause memory leaks

# Solution: Decouple Interface & Implementation(s)




- Create a public interface class (**`Expression_Tree`**) used by clients & a private implementation hierarchy (rooted at **`Component_Node`**) that encapsulates variability
  - The public interface class can perform reference counting of implementation object(s) to automate memory management
- An Abstract Factory can produce the right implementation (as seen later)

# Expression_Tree

Interface for Composite pattern used to contain all nodes in expression tree

**Interface:**

```
                    Expression_Tree (void)
                    Expression_Tree (Component_Node *root)
                    Expression_Tree (const Expression_Tree &t)
               void operator= (const Expression_Tree &t)
                    ~Expression_Tree (void)
   Component_Node * get_root (void)
               bool is_null (void) const
          const int item (void) const
    Expression_Tree left (void)
    Expression_Tree right (void)
           iterator begin (const std::string &traversal_order)
           iterator end (const std::string &traversal_order)
     const_iterator begin (const std::string &traversal_order) const
     const_iterator end (const std::string &traversal_order) const
```

**Commonality**: Provides a common interface for expression tree operations

**Variability**: The contents of the expression tree nodes can vary depending on the expression

# Bridge object structural

## Intent

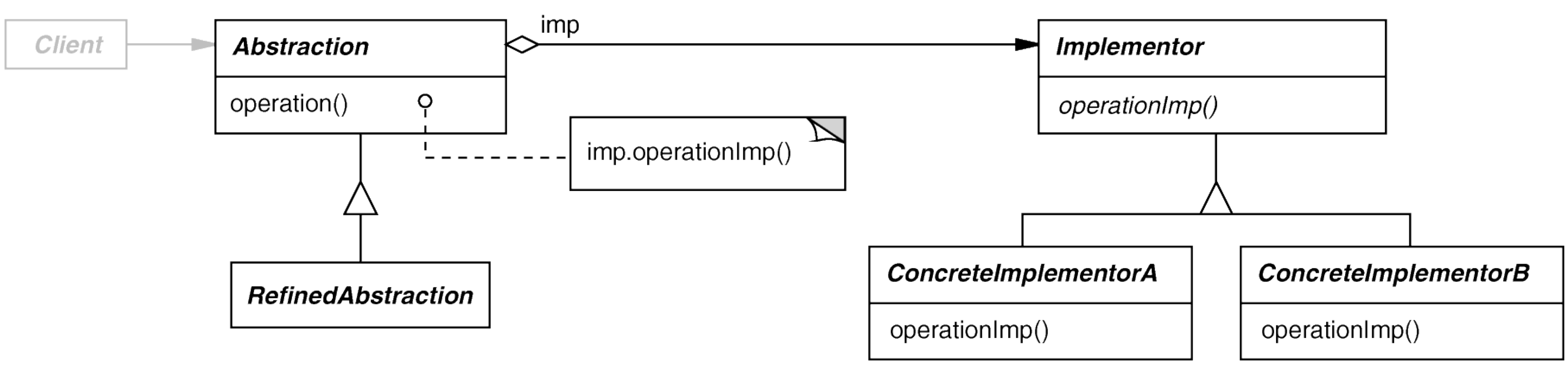
Separate a (logical) abstraction interface from its (physical) implementation(s)

## Applicability

– When interface & implementation should vary independently

– Require a uniform interface to interchangeable class hierarchies

## Structure



# Bridge object structural

## Consequences

+ abstraction interface & implementation are independent
+ implementations can vary dynamically
+ Can be used transparently with STL algorithms & containers

– one-size-fits-all Abstraction & Implementor interfaces

## Implementation

- sharing Implementors & reference counting
  - See reusable **`Refcounter`** template class (based on STL/boost **`shared_pointer`**)
- creating the right Implementor (often use factories)

## Known Uses

- ET++ Window/WindowPort
- libg++ Set/{LinkedList, HashTable}
- AWT Component/ComponentPeer

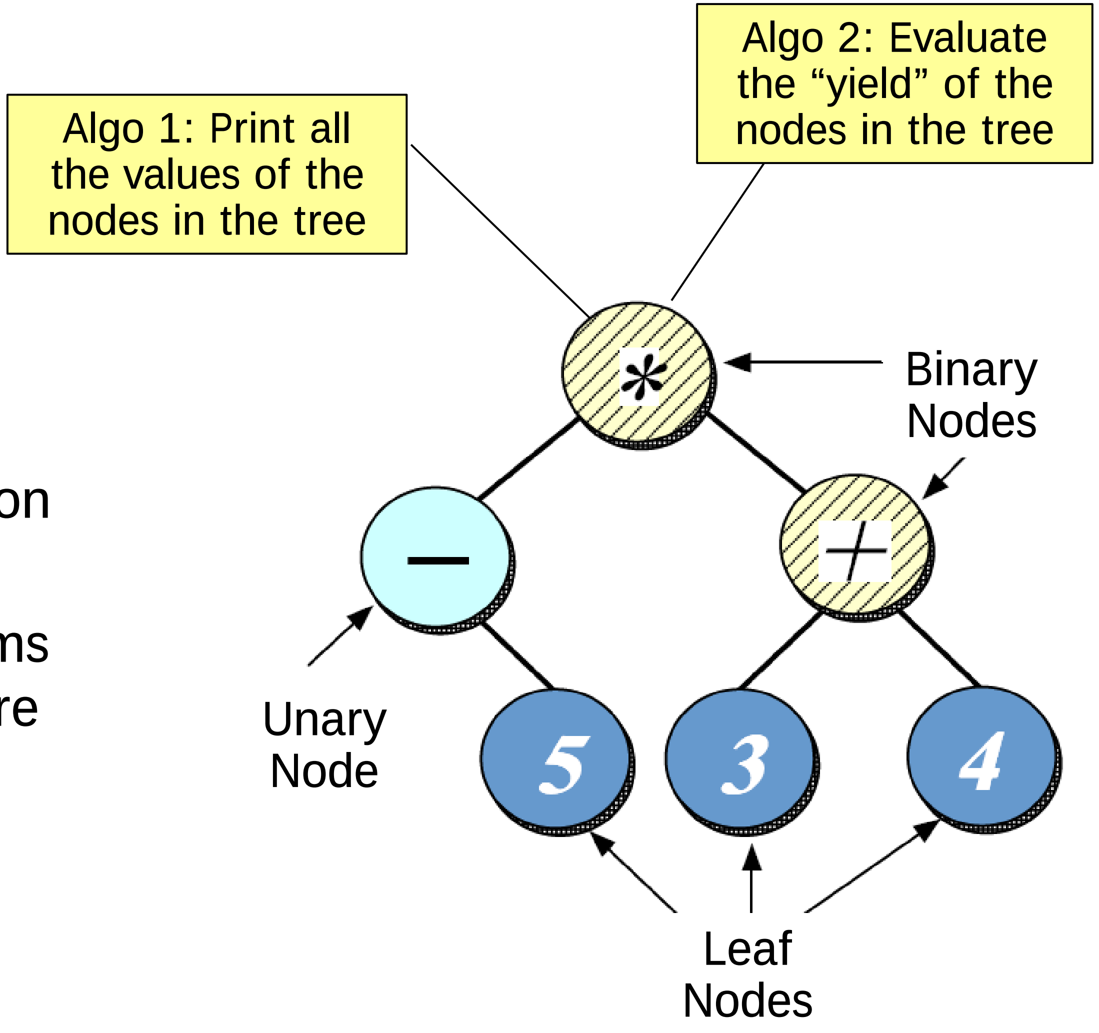
# Tree Printing & Evaluation

**Goals:**

- create a framework for performing algorithms that affect nodes in a tree

**Constraints/forces:**

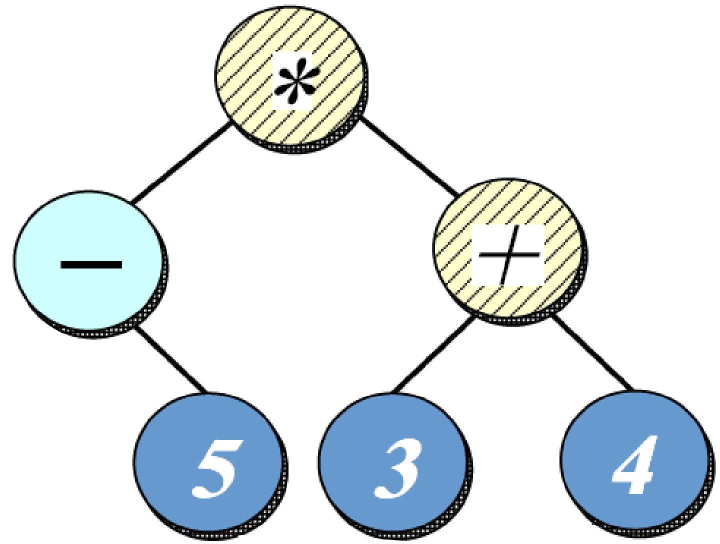
- support multiple algorithms that can act on the expression tree
- don't tightly couple algorithms with expression tree structure
  - e.g., don't have "print" & "evaluate" methods in the node classes

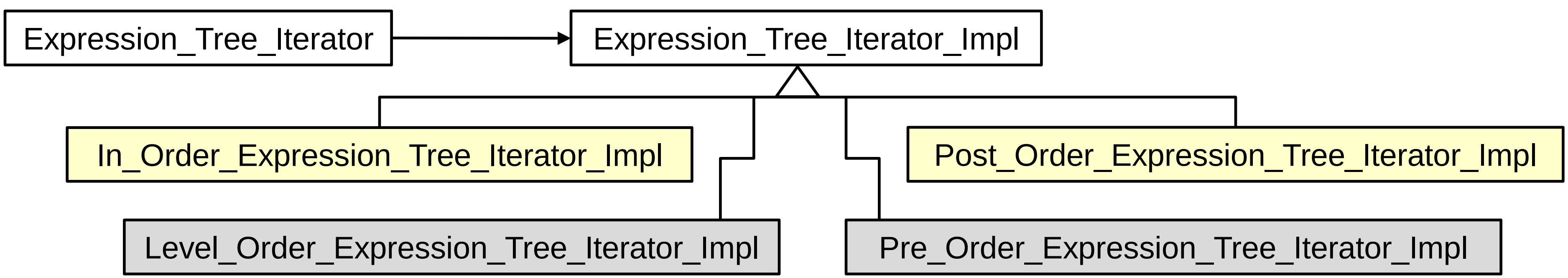


# Solution: Encapsulate Traversal

**Iterator**

- encapsulates a traversal algorithm without exposing representation details to callers
- e.g.,
  - “in-order iterator” = **`-5*(3+4)`**
  - “pre-order iterator” = **`*-5+34`**
  - “post-order iterator” = **`5-34+*`**
  - “level-order iterator” = **`*-+534`**




Note use of the Bridge pattern to encapsulate variability

# Expression_Tree_Iterator

Interface for Iterator pattern that traverses all nodes in tree expression

**Interface:**

```
Expression_Tree_Iterator (const Expression_Tree_Iterator &)
Expression_Tree_Iterator (Expression_Tree_Iterator_Impl *)
Expression_Tree operator * (void)
const Expression_Tree operator * (void) const
Expression_Tree_Iterator & operator++ (void)
Expression_Tree_Iterator operator++ (int)
bool operator== (const Expression_Tree_Iterator &rhs)
bool operator!= (const Expression_Tree_Iterator &rhs)
```

**Commonality**: Provides a common interface for expression tree iterators that conforms to the standard STL iterator interface

**Variability**: Can be configured with specific expression tree iterator algorithms via the Bridge & Abstract Factory patterns

See Expression_Tree_State.cpp for example usage

# Expression_Tree_Iterator_Impl

Implementation of the Iterator pattern that is used to define the various iterations algorithms that can be performed to traverse the expression tree

**Interface:**

```
Expression_Tree_Iterator_Impl (const
        Expression_Tree &tree)
virtual ~Expression_Tree_Iterator_Impl (void)
virtual Expression_Tree operator * (void) =0
virtual const Expression_Tree operator * (void) const =0
virtual void operator++ (void)=0
virtual bool operator== (const
        Expression_Tree_Iterator_Impl &rhs) const =0
virtual bool operator!= (const
        Expression_Tree_Iterator_Impl &rhs) const =0
virtual Expression_Tree_Iterator_Impl * clone (void)=0
```

**Commonality**: Provides a common interface for implementing expression tree iterators that conforms to the standard STL iterator interface

**Variability**: Can be subclasses to define various algorithms for accessing nodes in the expression trees in a particular traversal order

# Iterator object behavioral

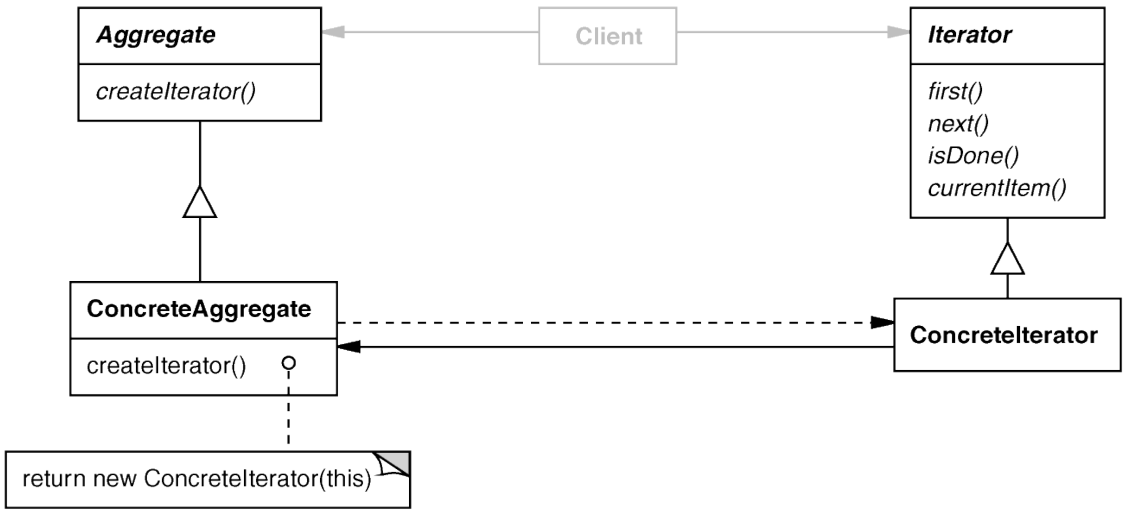
## Intent

access elements of a aggregate (container) without exposing its representation

## Applicability

- require multiple traversal algorithms over an aggregate
- require a uniform traversal interface over different aggregates
- when aggregate classes & traversal algorithm must vary independently

## Structure



# Comparing STL Iterators with GoF Iterators

STL iterators have “value-semantics”, e.g.:

```
for (Expression_Tree::iterator iter = tree.begin ("Level Order");
     iter != tree.end ("Level Order");
     ++iter)
  (*iter).accept (print_visitor);
```

In contrast, “GoF iterators have “pointer semantics”, e.g.:

```
Iterator *iter;

for (iter = tree.createIterator ("Level Order");
     iter->done () == false;
     iter->advance ())
  (iter->currentElement ())->accept (print_visitor);

delete iter;
```

The Bridge pattern simplifies use of STL iterators in expression tree application

# Iterator object behavioral

## Consequences

+ flexibility: aggregate & traversal are independent
+ multiple iterators & multiple traversal algorithms
– additional communication overhead between iterator & aggregate
  – This is particularly problematic for iterators in concurrent or distributed systems

## Implementation

- internal versus external iterators
- violating the object structure's encapsulation
- robust iterators
- synchronization overhead in multi-threaded programs
- batching in distributed & concurrent programs

## Known Uses

- C++ STL iterators
- JDK Enumeration, Iterator
- Unidraw Iterator

# Visitor

- Defines action(s) at each step of traversal & avoids wiring action(s) in nodes
- Iterator calls nodes's **`accept(Visitor)`** at each node, e.g.:

```
void Leaf_Node::accept (Visitor &v) { v.visit (*this); }
```

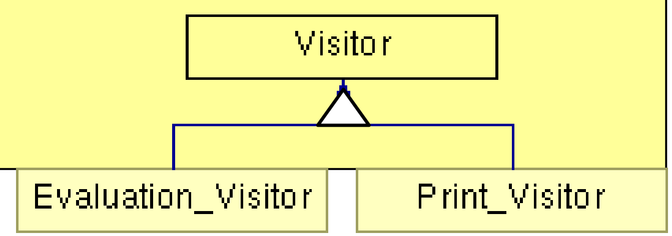
- **`accept()`** calls back on visitor using "static polymorphism"

**Interface:**

virtual void visit (const Leaf_Node &node)=0
virtual void visit (const Composite_Negate_Node &node)=0
virtual void visit (const Composite_Add_Node &node)=0
virtual void visit (const Composite_Subtract_Node &node)=0
virtual void visit (const Composite_Divide_Node &node)=0
virtual void visit (const Composite_Multiply_Node &node)=0

**Commonality**: Provides a common **`accept()`** method for all expression tree nodes & common **`visit()`** method for all visitor subclasses

**Variability**: Can be subclassed to define specific behaviors for the visitors & nodes



# Print_Visitor

- Prints character code or value for each node

```
class Print_Visitor : public Visitor {
public:
    virtual void visit (const Leaf_Node &);
    virtual void visit (const Add_Node &);
    virtual void visit (const Divide_Node &);
    // etc. for all relevant Component_Node subclasses
};
```

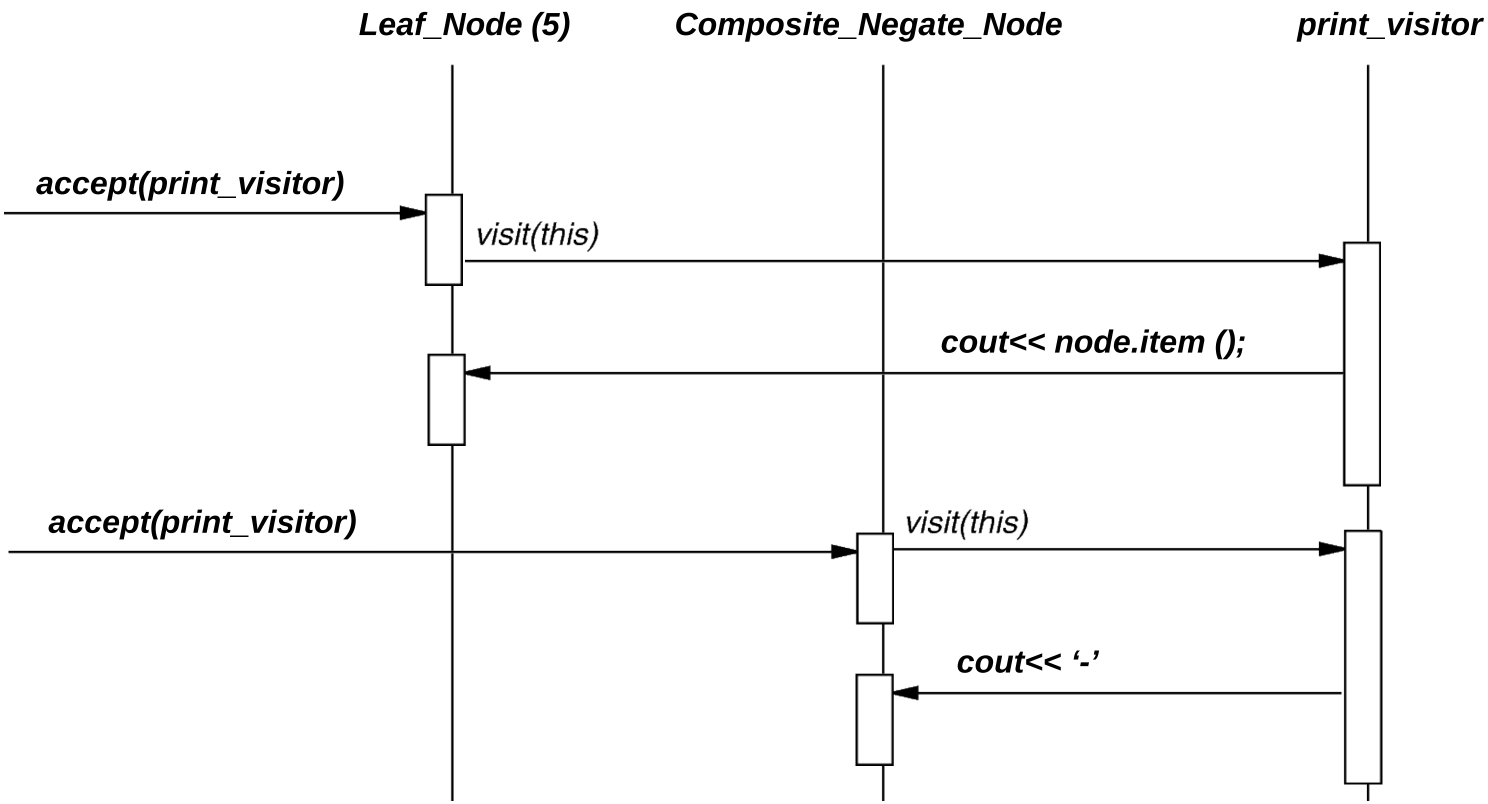
- Can be combined with any traversal algorithm, e.g.:

```
Print_Visitor print_visitor;
for (Expression_Tree::iterator iter =
       tree.begin ("post-order");
     iter != tree.end ("post-order");
     ++iter)
  (*iter).accept (print_visitor); // calls visit (*this);
```

See Expression_Tree_State.cpp for example usage

# Print_Visitor Interaction Diagram

- The iterator controls the order in which **`accept()`** is called on each node in the composition
- **`accept()`** then "visits" the node to perform the desired *print* action

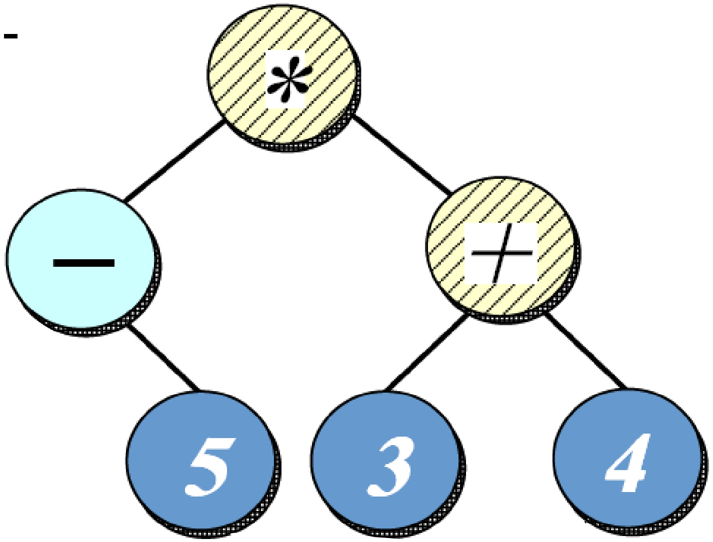


# Evaluation_Visitor

- This class serves as a visitor for evaluating nodes in an expression tree that is being traversed using a post-order iterator
  - e.g., **5-34+***

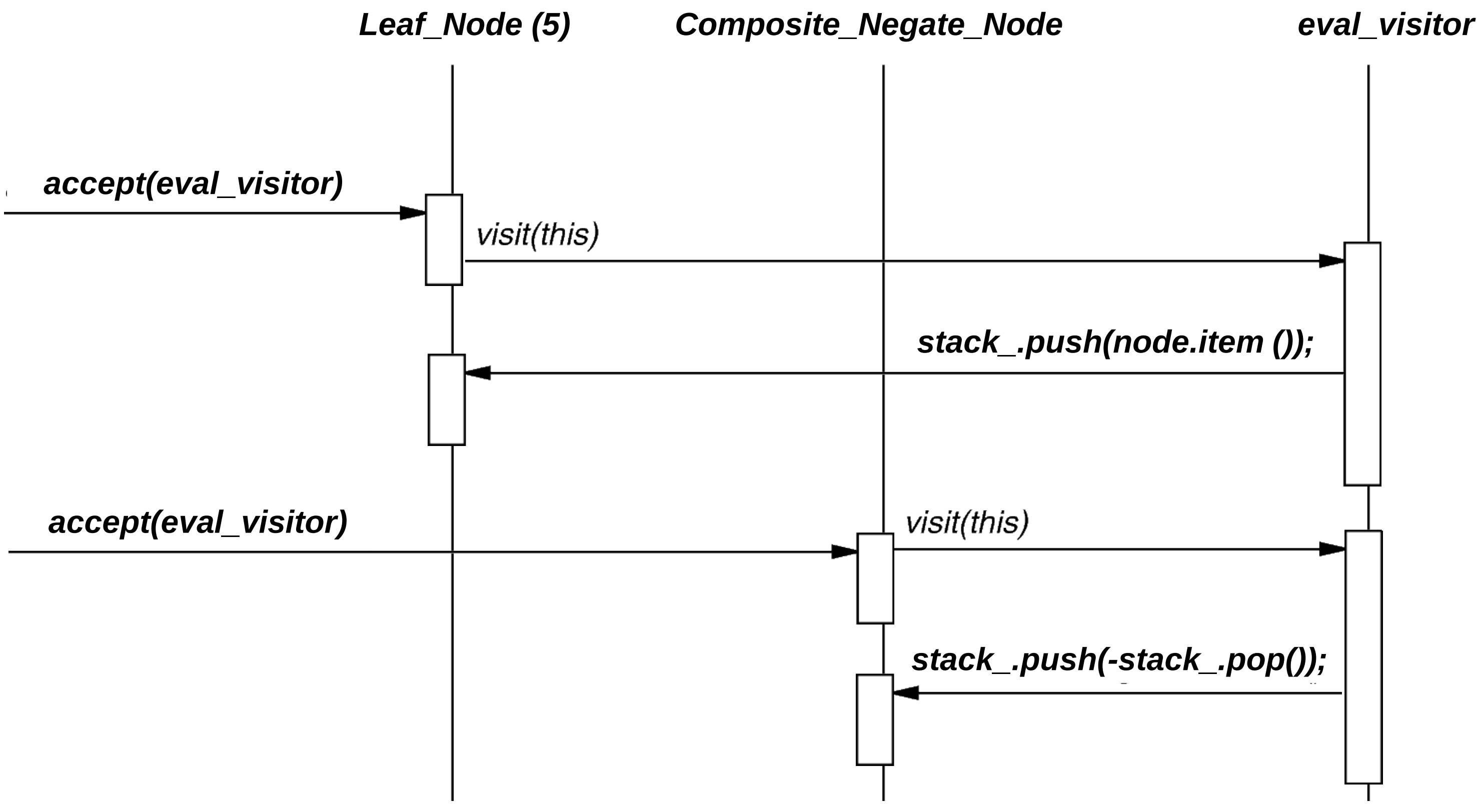
```
class Evaluation_Visitor :
   public Visitor { /* ... */ };
```

- It uses a stack to keep track of the post-order expression tree value that has been processed thus far during the iteration traversal, e.g.:
  1. S = [5] **push(node.item())**
  2. S = [-5] **push(-pop())**
  3. S = [-5, 3] **push(node.item())**
  4. S = [-5, 3, 4] **push(node.item())**
  5. S = [-5, 7] **push(pop()+pop())**
  6. S = [-35] **push(pop()*pop())**



# Evaluation_Visitor Interaction Diagram

- The iterator controls the order in which **`accept()`** is called on each node in the composition
- **`accept()`** then "visits" the node to perform the desired *evaluation* action



# Visitor object behavioral

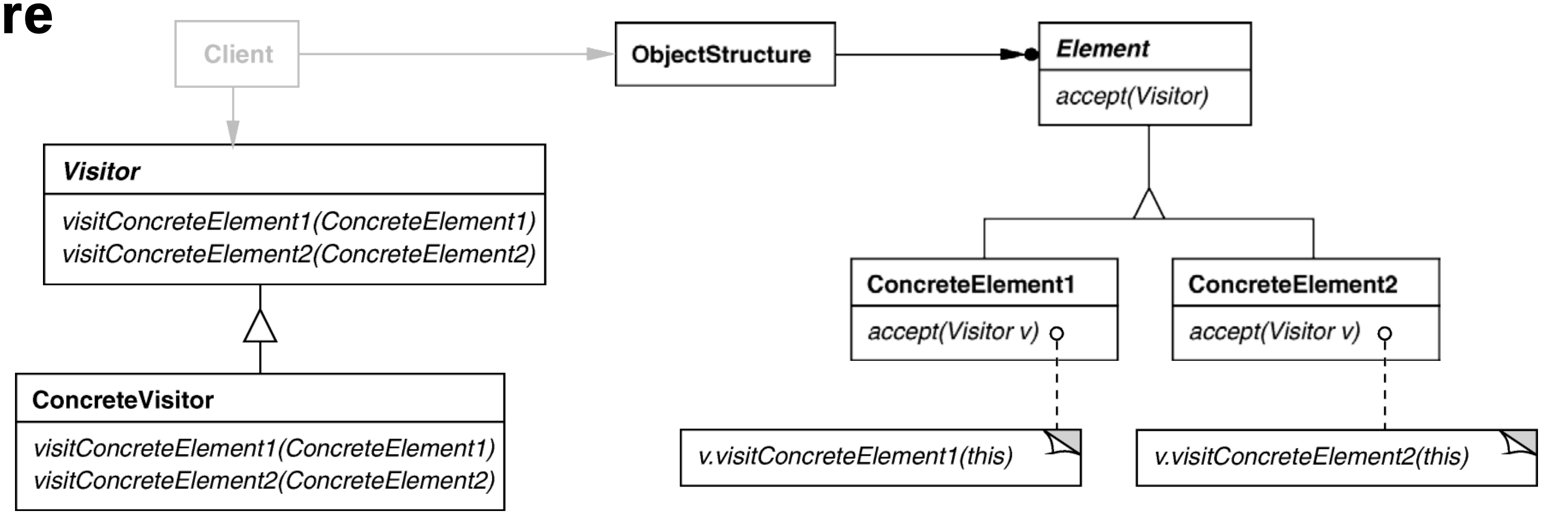
## Intent

Centralize operations on an object structure so that they can vary independently but still behave polymorphically

## Applicability

– when classes define many unrelated operations
– class relationships of objects in the structure rarely change, but the operations on them change often
– algorithms keep state that's updated during traversal

## Structure




Note "static polymorphism" based on method overloading by type

# Visitor object behavioral

## Consequences

+ flexibility: visitor algorithm(s) & object structure are independent
+ localized functionality in the visitor subclass instance
– circular dependency between Visitor & Element interfaces
– Visitor brittle to new ConcreteElement classes

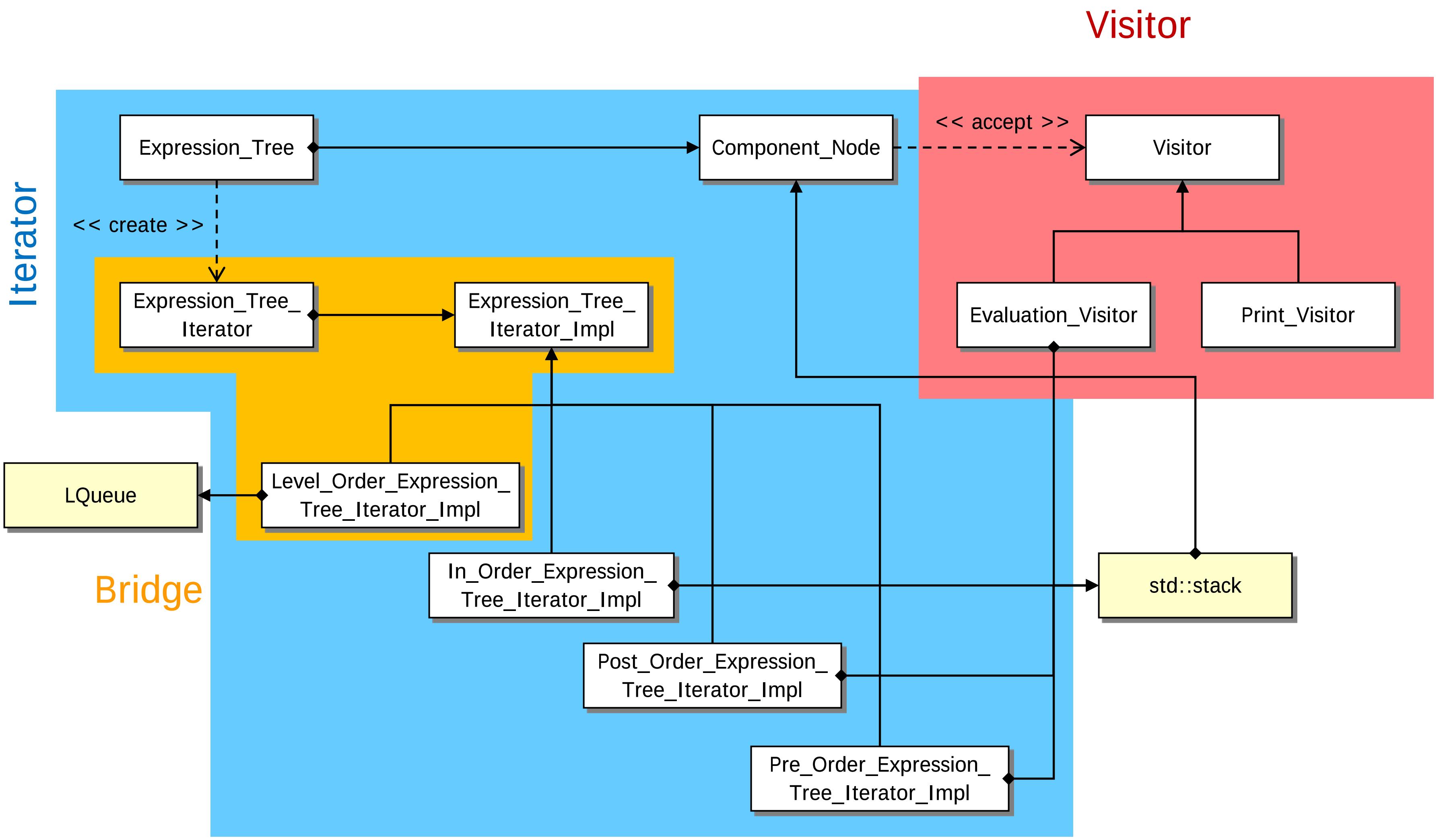
## Implementation

- double dispatch
- general interface to elements of object structure

## Known Uses

- ProgramNodeEnumerator in Smalltalk-80 compiler
- IRIS Inventor scene rendering
- TAO IDL compiler to handle different backends

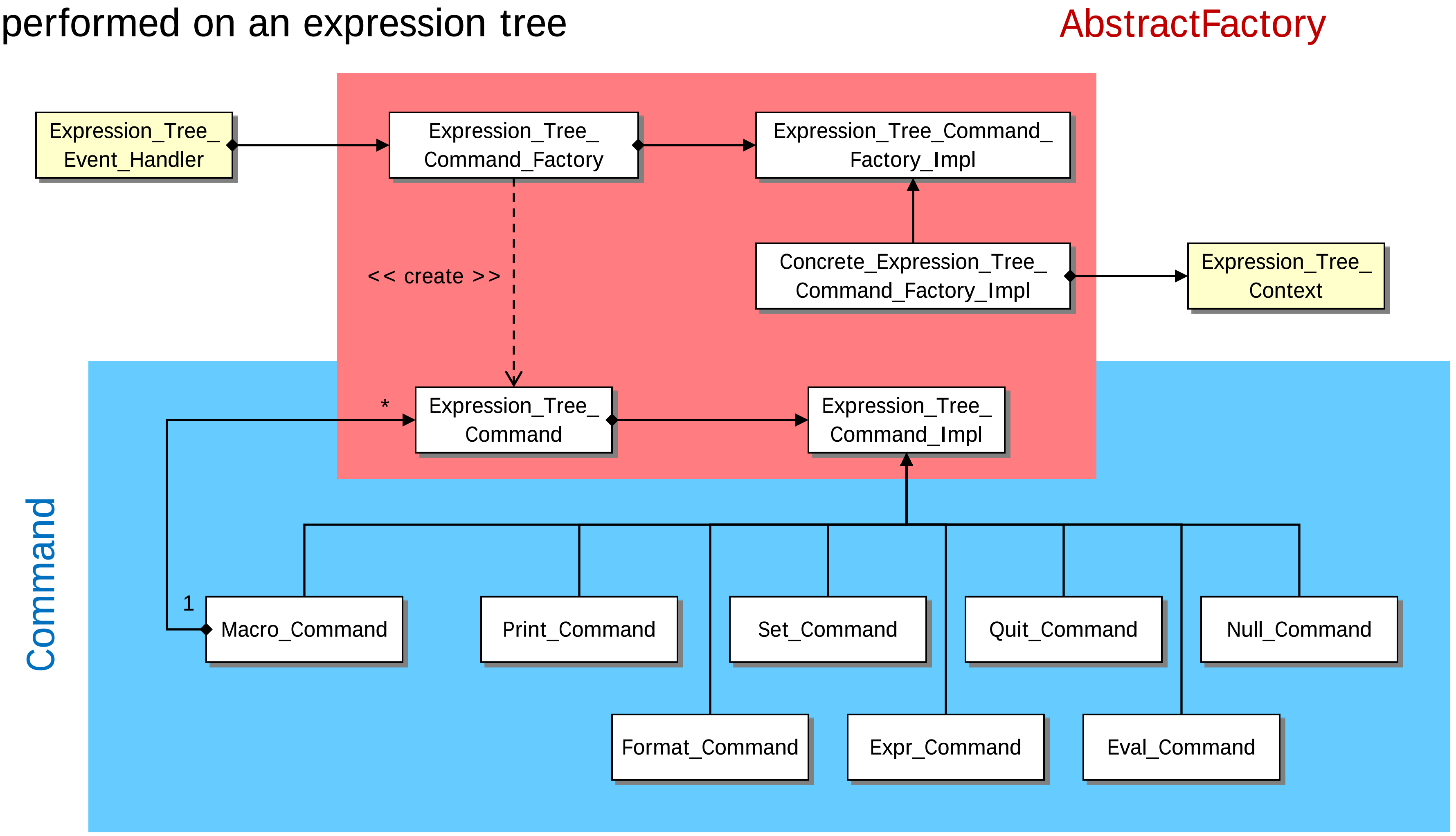
# Summary of Tree Traversal Patterns




Patterns allow adding new operations extensibly without affecting structure

# Overview of Command & Factory Patterns

Purpose: Define operations that can be performed on an expression tree




These patterns decouple creation from use & provide a uniform command API

# Consolidating User Operations

**Goals:**

- support execution of user operations
- support macro commands
- support undo/redo

**Constraints/forces:**

- scattered operation implementations
- Consistent memory management

```
% tree-traversal -v
format [in-order]
expr [expression]
print [in-order|pre-order|post-order|level-order]
eval [post-order]
quit
> format in-order
> expr 1+2*3/2
> print in-order
1+2*3/2
> print pre-order
+1/*232
> eval post-order
4
> quit
```

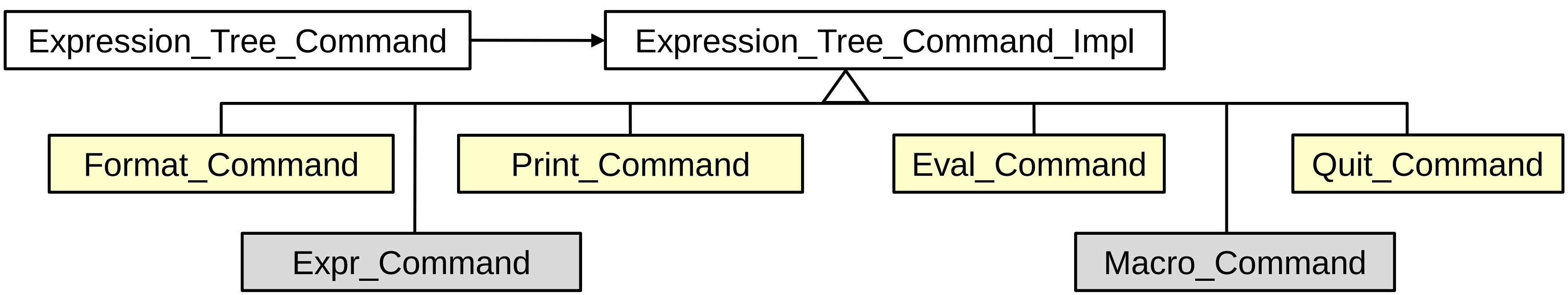
# Solution: Encapsulate Each Request w/Command

A **Command** encapsulates

- an operation (**`execute()`**)
- an inverse operation (**`unexecute()`**)
- a operation for testing reversibility (**`boolean reversible()`**)
- state for (un)doing the operation

Command may

- implement the operations itself, *or*
- forward them to other object(s)




Note use of Bridge pattern to encapsulate variability & simplify memory management

# Expression_Tree_Command

Interface for Command pattern used to define a command that performs an operation on the expression tree when executed

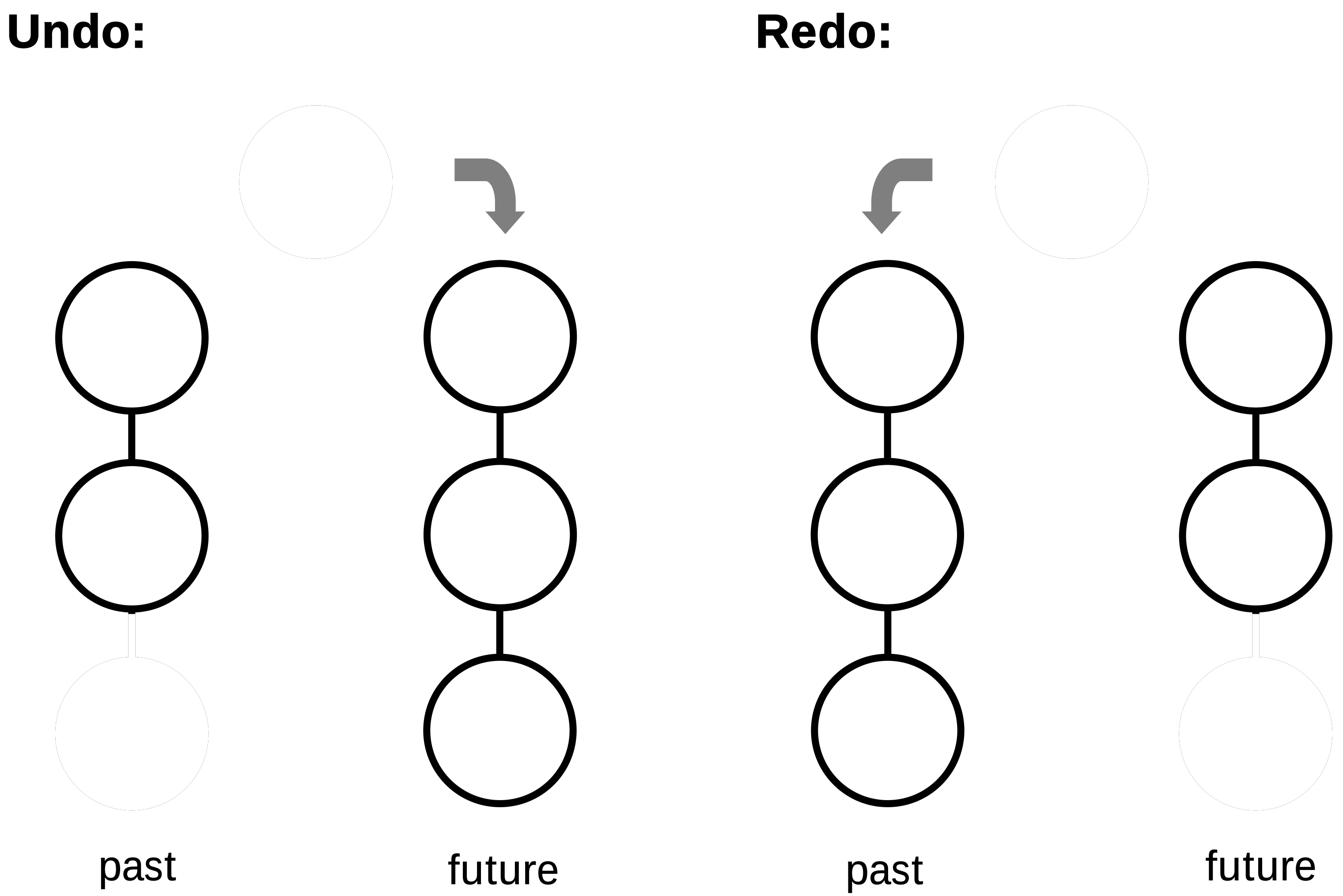
**Interface:**

```
                      Expression_Tree_Command (Expression_Tree_Command_Impl *=0)
                      Expression_Tree_Command (const Expression_Tree_Command &)
Expression_Tree_Command & operator= (const Expression_Tree_Command &)
                      ~Expression_Tree_Command (void)
                 bool execute (void)
                 bool unexecute (void)
```

**Commonality**: Provides a common interface for expression tree commands

**Variability**: The implementations of the expression tree commands can vary depending on the operations requested by user input

# List of Commands = Execution History



# Command object behavioral

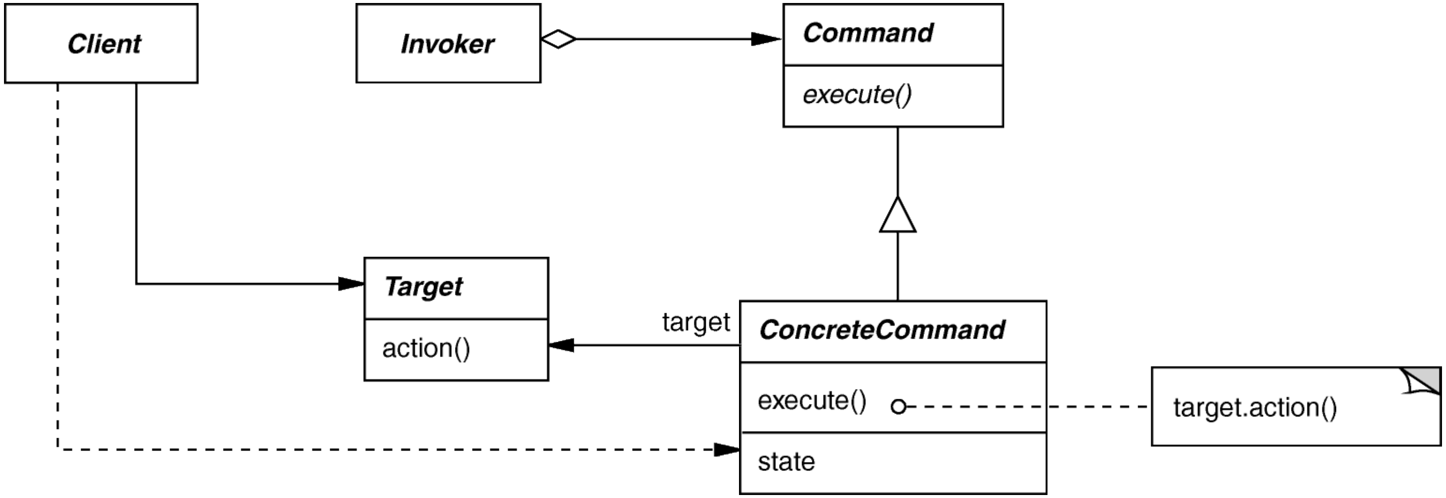
## Intent

Encapsulate the request for a service

## Applicability

- to parameterize objects with an action to perform
- to specify, queue, & execute requests at different times
- for multilevel undo/redo

## Structure



# Command object behavioral

## Consequences

+ abstracts executor of a service
+ supports arbitrary-level undo-redo
+ composition yields macro-commands
– might result in lots of trivial command subclasses
– excessive memory may be needed to support undo/redo operations

## Implementation

- copying a command before putting it on a history list
- handling hysteresis
- supporting transactions

## Known Uses

- InterViews Actions
- MacApp, Unidraw Commands
- JDK's UndoableEdit, AccessibleAction
- Emacs
- Microsoft Office tools

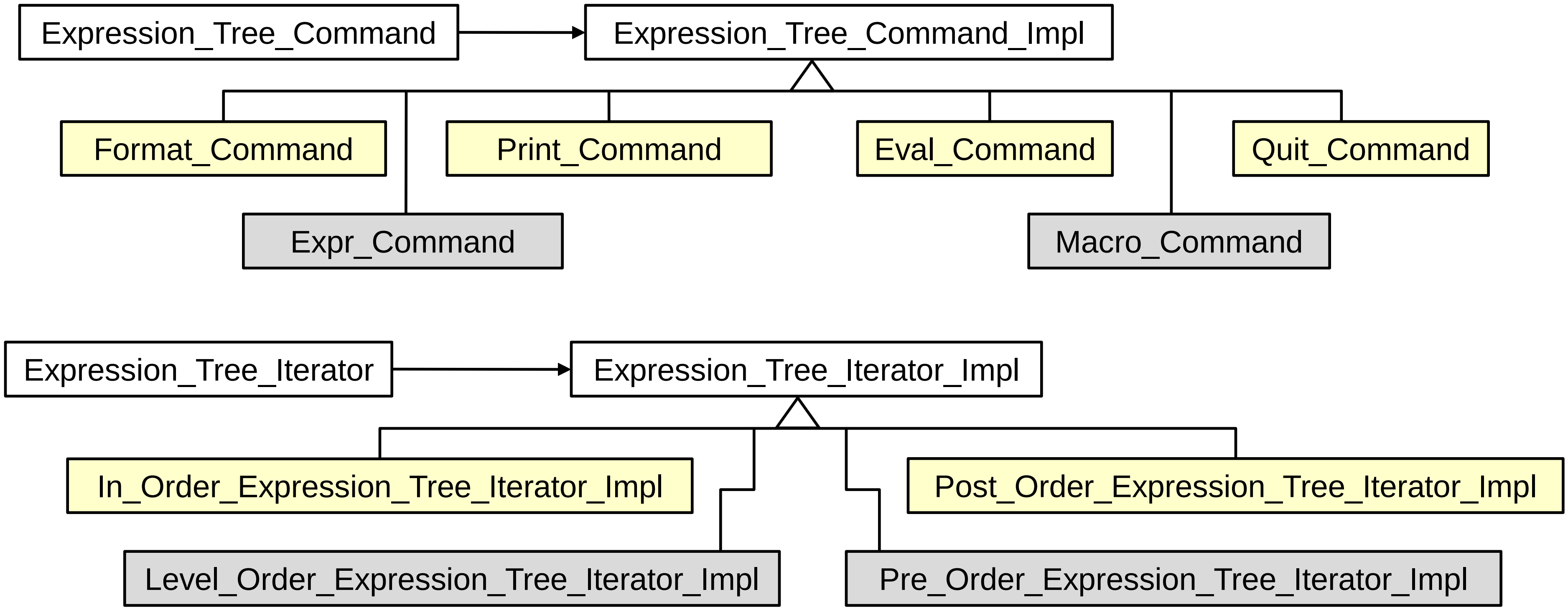
# Consolidating Creation of Variabilities

**Goals:**

- Simplify & centralize the creation of all variabilities in the expression tree application to ensure semantic compatibility
- Be extensible for future variabilities

**Constraints/forces:**

- Don't recode existing clients
- Add new variabilities without recompiling



# Solution: Abstract Object Creation

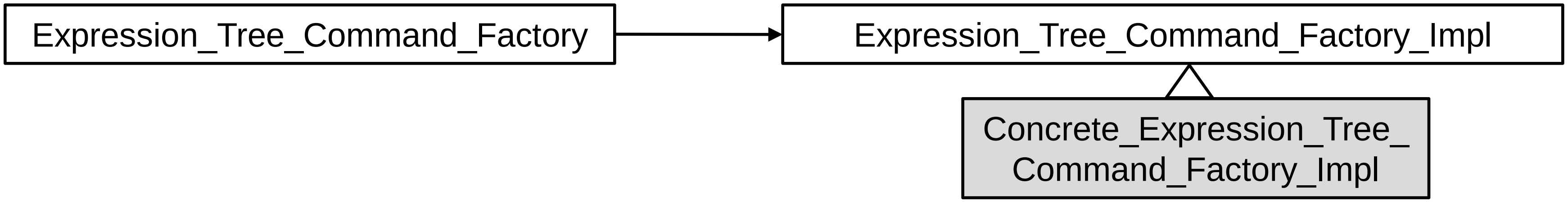
Instead of

```
Expression_Tree_Command command
  = new Print_Command ();
```

Use

```
Expression_Tree_Command command
  = command_factory.make_command ("print");
```

where **command_factory** is an instance of **Expression_Tree_Command_Factory** or anything else that makes sense wrt our goals



# Expression_Tree_Command_Factory

Interface for Abstract Factory pattern used to create appropriate command based on string supplied by caller

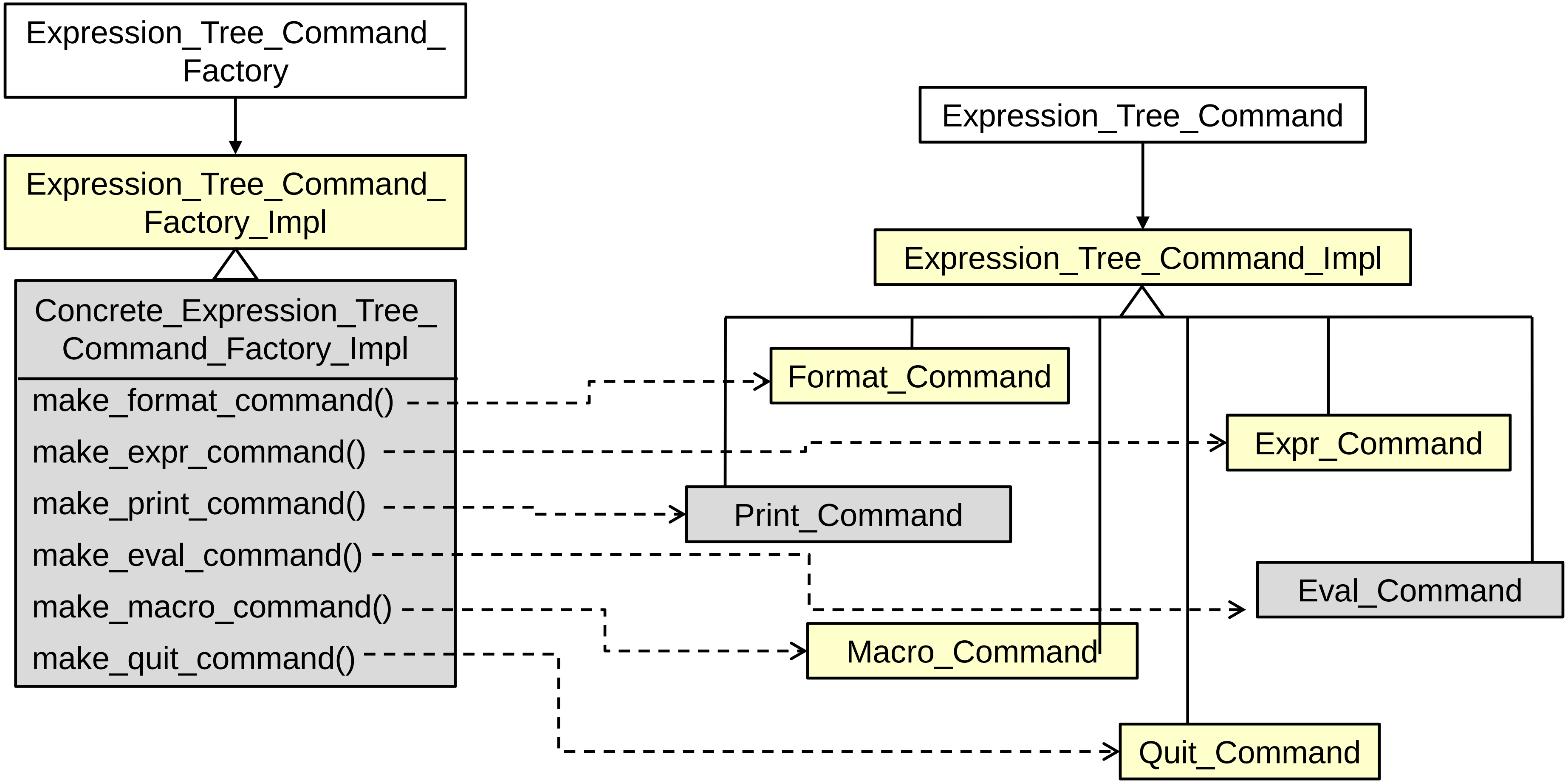
**Interface:**

```
                              Expression_Tree_Command_Factory
                                        (Expression_Tree_Context &tree_context)
                              Expression_Tree_Command_Factory
                                        (const Expression_Tree_Command_Factory &f)
                         void operator= (const Expression_Tree_Command_Factory &f)
                              ~Expression_Tree_Command_Factory (void)
Expression_Tree_Command make_command (const std::string &s)
Expression_Tree_Command make_format_command (const std::string &)
Expression_Tree_Command make_expr_command (const std::string &)
Expression_Tree_Command make_print_command (const std::string &)
Expression_Tree_Command make_eval_command (const std::string &)
Expression_Tree_Command make_quit_command (const std::string &)
Expression_Tree_Command make_macro_command (const std::string &)
```

**Commonality**: Provides a common interface to create commands

**Variability**: The implementations of the expression tree command factory methods can vary depending on the requested commands

# Factory Structure




Note use of Bridge pattern to encapsulate variability & simplify memory management

# Factory Method class creational

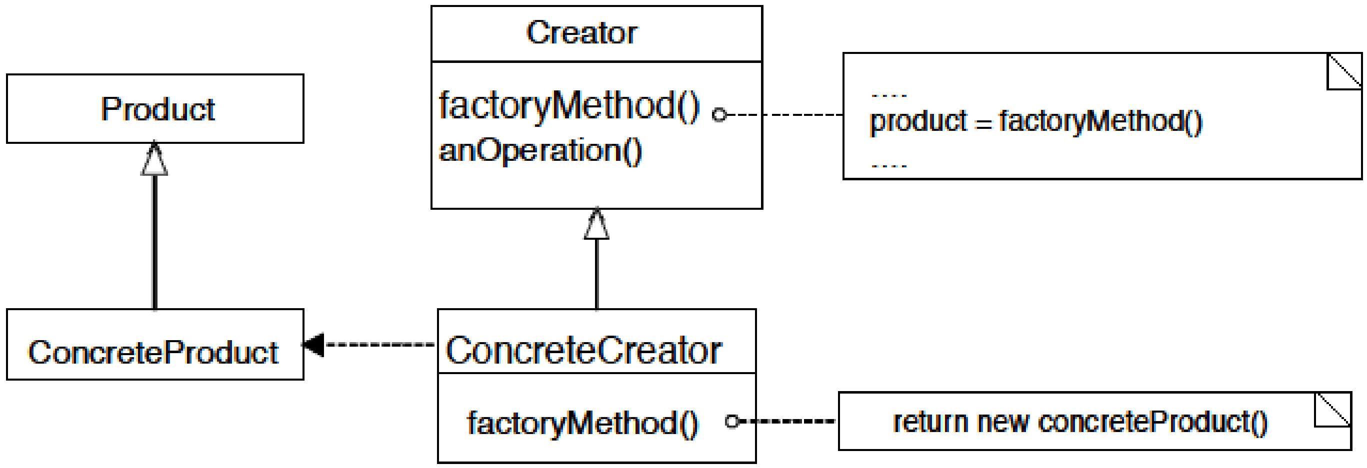
## Intent

Provide an interface for creating an object, but leave choice of object's concrete type to a subclass

## Applicability

when a class cannot anticipate the objects it must create or a class wants its subclasses to specify the objects it creates

## Structure



# Factory Method class creational

**Consequences**

+By avoiding to specify the class name of the concrete class &the details of its creation the client code has become more flexible

+The client is only dependent on the interface

- Construction of objects requires one additional class in some cases

**Implementation**

- There are two choices here
  - The creator class is abstract & does not implement creation methods (then it *must be subclassed)*
  - The creator class is concrete & provides a default implementation (then it *can be subclassed)*
- Should a factory method be able to create different variants? If so the method must be equipped with a parameter

**Known Uses**

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- The ACE ORB (TAO)
- BREW
- UNIX open() syscall

# Abstract Factory object creational

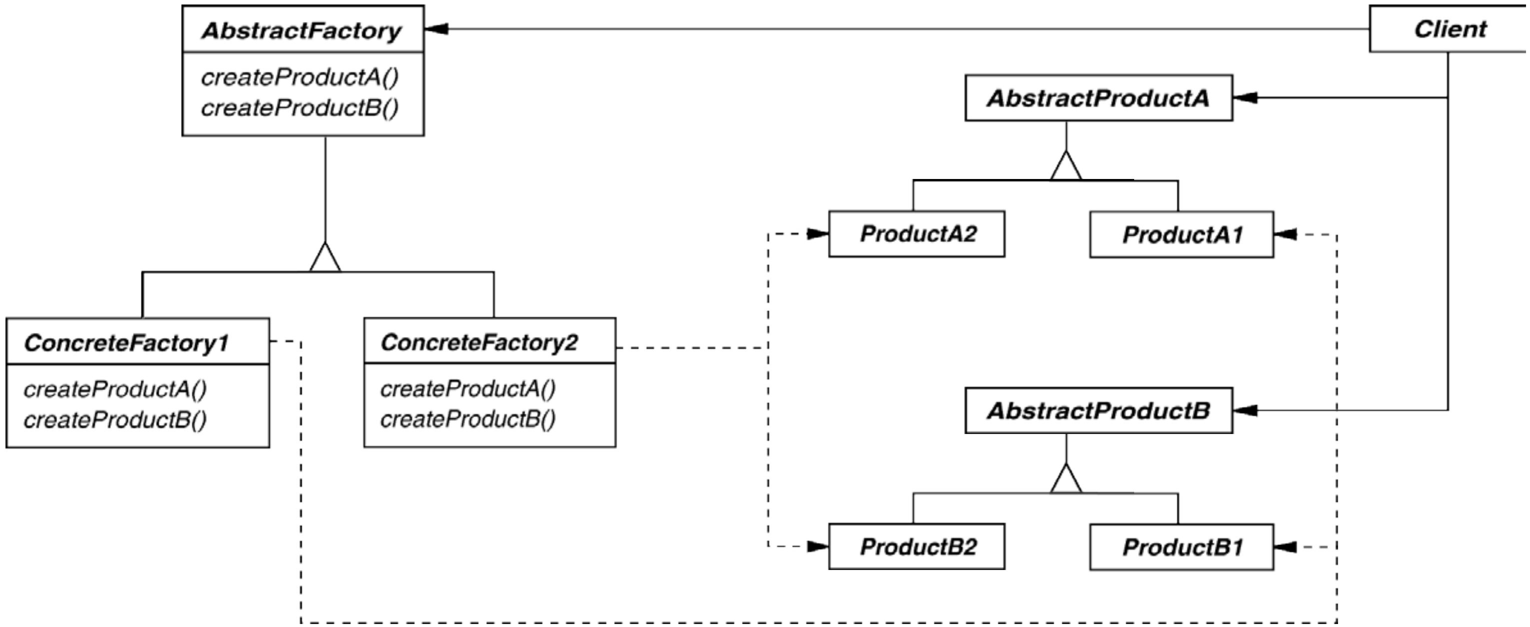
## Intent

create families of related objects without specifying subclass names

## Applicability

when clients cannot anticipate groups of classes to instantiate

## Structure




See **`Uninitialized_State_Factory`** & **`Expression_Tree_Event_Handler`** for Factory pattern variants

# Abstract Factory object creational

## Consequences

+ flexibility: removes type (i.e., subclass) dependencies from clients
+ abstraction & semantic checking: hides product's composition
– hard to extend factory interface to create new products

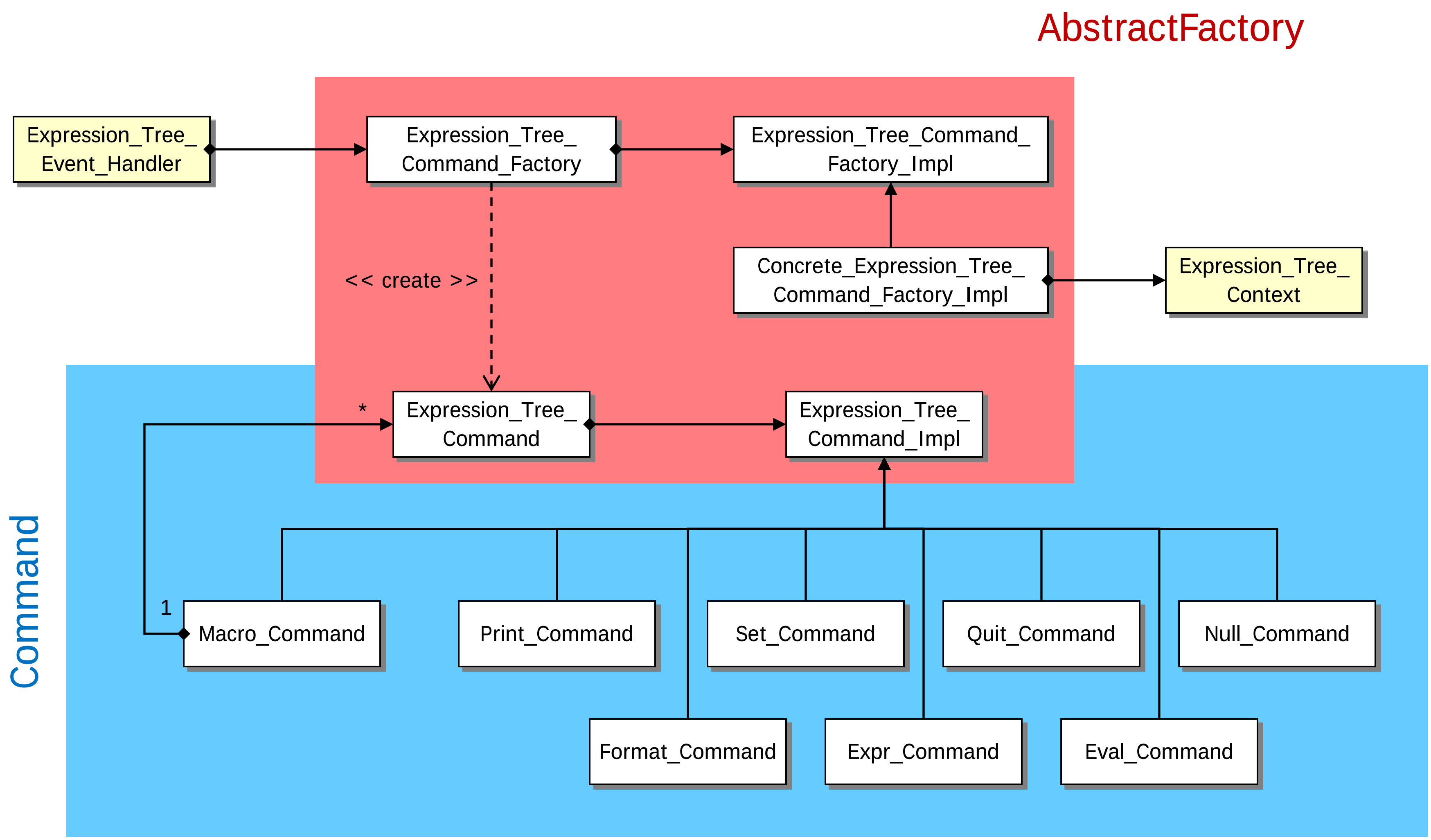
## Known Uses

– InterViews Kits
– ET++ WindowSystem
– AWT Toolkit
– The ACE ORB (TAO)

## Implementation

- parameterization as a way of controlling interface size
- configuration with Prototypes, i.e., determines who creates the factories
- abstract factories are essentially groups of factory methods

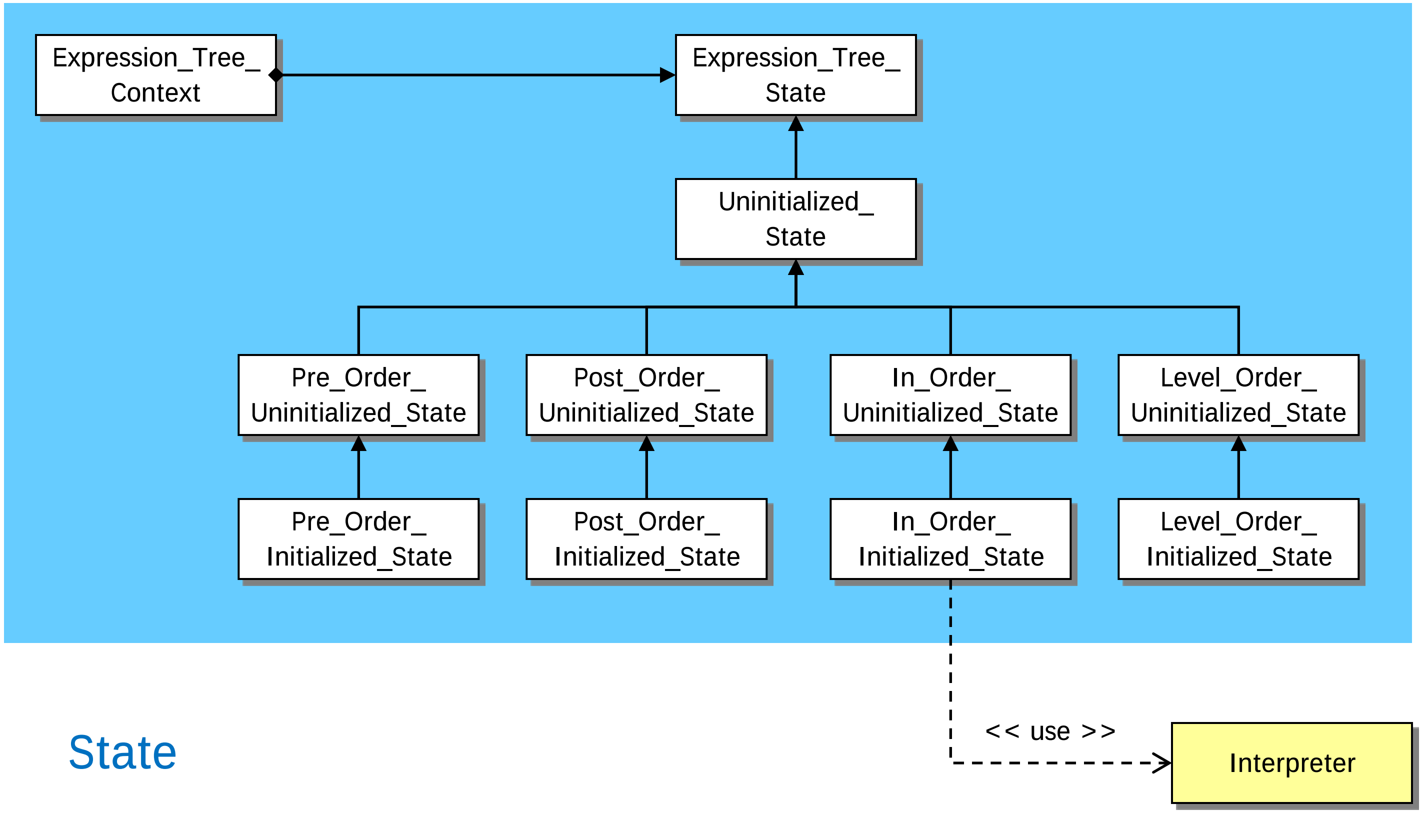
# Summary of Command & Factory Patterns




Patterns enable new commands extensibly by adding new factory methods

# Overview of State Pattern

Purpose: Ensure user commands are performed in the correct order




State

Pattern organizes the correct processing of user commands

# Ensuring Correct Protocol for Commands

**Goals:**

– Ensure that users follow the correct protocol when entering commands

**Constraints/forces:**

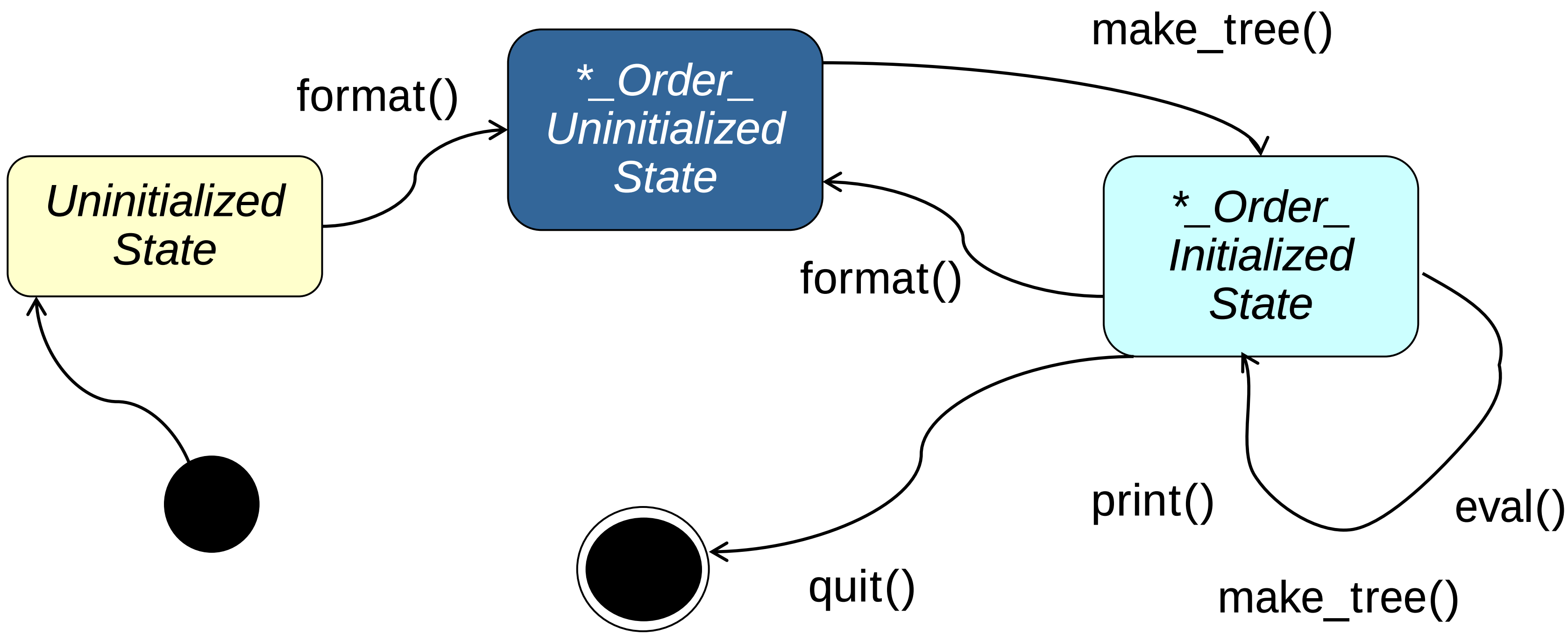
– Must consider context of previous commands to determine protocol conformance, e.g.,
  - **`format`** must be called first
  - **`expr`** must be called before **`print`** or **`eval`**
  - **`Print`** & **`eval`** can be called in any order

```
% tree-traversal -v
format [in-order]
expr [expression]
print [in-order|pre-order|post-order|level-order]
eval [post-order]
quit
> format in-order
> print in-order
Error: Expression_Tree_State::print called
in invalid state
```

Protocol violation

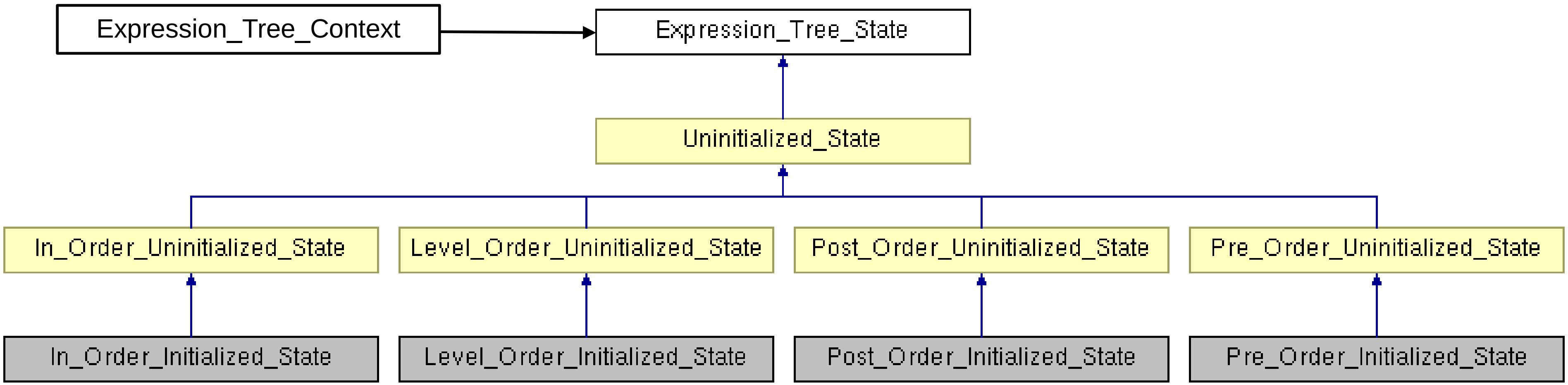
# Solution: Encapsulate Command History as States

- The handling of a user command depends on the history of prior commands
- This history can be represented as a state machine



# Solution: Encapsulate Command History as States

- The state machine can be encoded using various subclasses that enforce the correct protocol for user commands




Note use of Bridge pattern to encapsulate variability & simplify memory management

# Expression_Tree_Context

Interface for State pattern used to ensure that commands are invoked according to the correct protocol

**Interface:**

```
void format (const std::string &new_format)
void make_tree (const std::string &expression)
void print (const std::string &format)
void evaluate (const std::string &format)
Expression_Tree_State * state (void) const
void state (Expression_Tree_State *new_state)
Expression_Tree & tree (void)
void tree (const Expression_Tree &new_tree)
```

**Commonality**: Provides a common interface for ensuring that expression tree commands are invoked according to the correct protocol

**Variability**: The implementations—& correct functioning—of the expression tree commands can vary depending on the requested operations & the current state

# Expression_Tree_State

Implementation of the State pattern that is used to define the various states that affect how users operations are processed

**Interface:**

```
virtual void format (Expression_Tree_Context &context,
                     const std::string &new_format)
virtual void make_tree (Expression_Tree_Context &context,
                        const std::string &expression)
virtual void print (Expression_Tree_Context &context,
                    const std::string &format)
virtual void evaluate (Expression_Tree_Context &context,
                       const std::string &format)
```

**Commonality**: Provides a common interface for ensuring that expression tree commands are invoked according to the correct protocol

**Variability**: The implementations—& correct functioning—of the expression tree commands can vary depending on the requested operations & the current state

# State object behavioral

## Intent

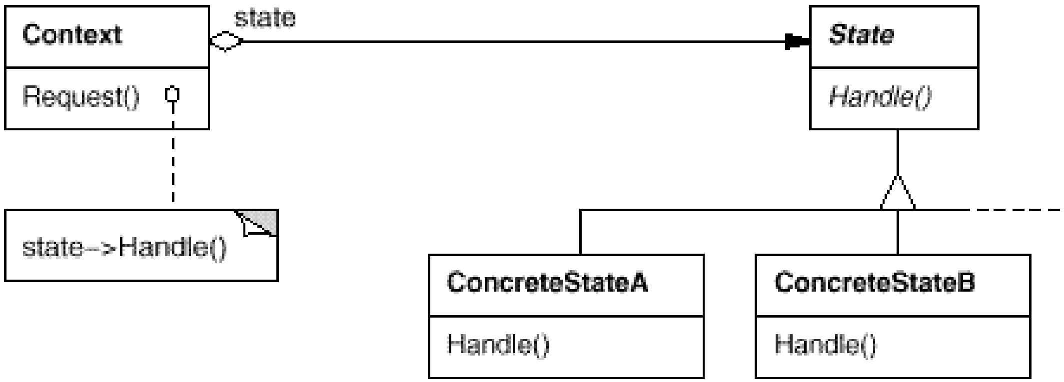
Allow an object to alter its behavior when its internal state changes—the object will appear to change its class

## Applicability

– When an object must change its behavior at run-time depending on which state it is in

– When several operations have the same large multipart conditional structure that depends on the object's state

## Structure



# State object behavioral

## Consequences

+ It localizes state-specific behavior & partitions behavior for different states
+ It makes state transitions explicit
+ State objects can be shared

– Can result in lots of subclasses that are hard to understand

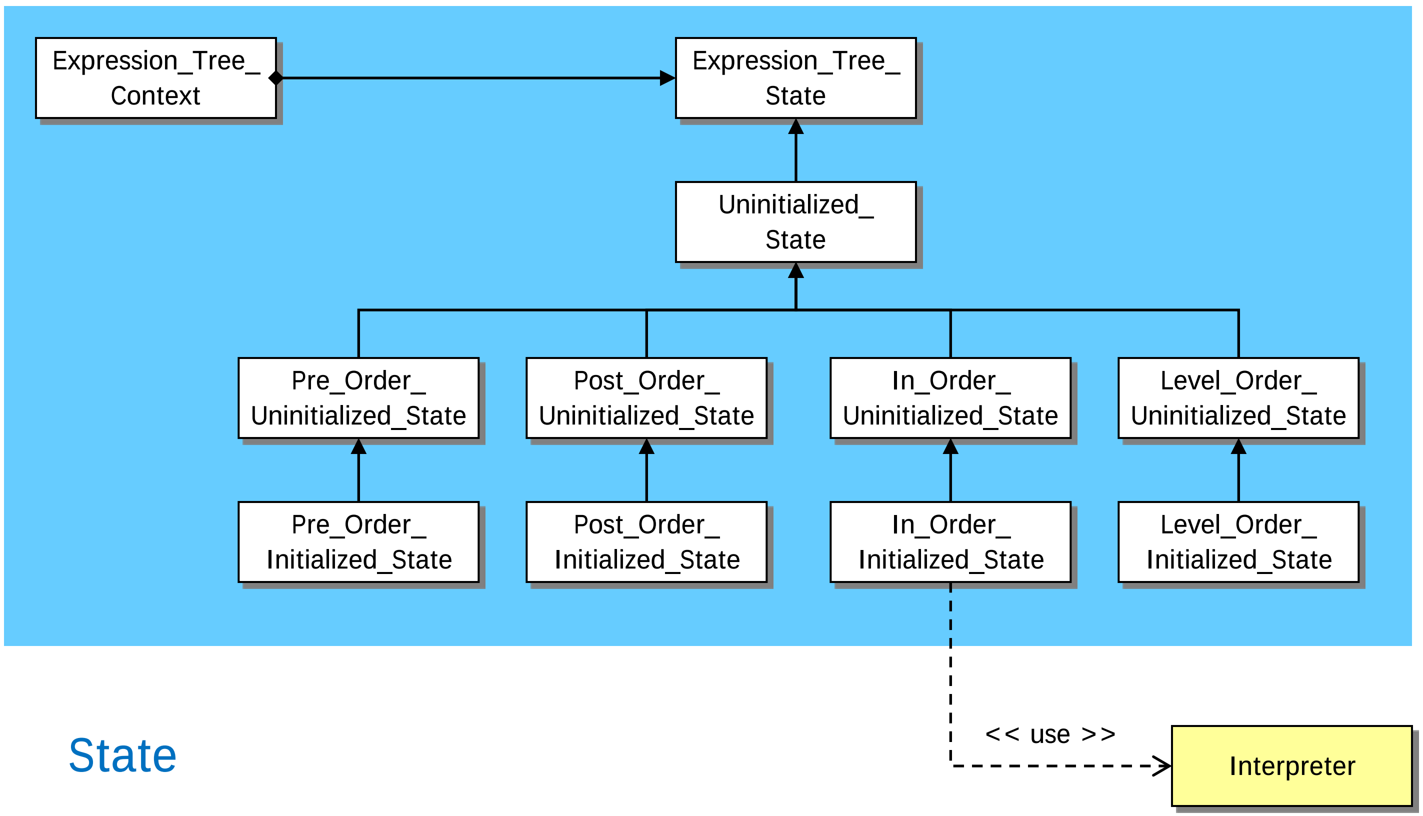
## Implementation

- Who defines state transitions?
- Consider using table-based alternatives
- Creating & destroying state objects

## Known Uses

- The State pattern & its application to TCP connection protocols are characterized in: Johnson, R.E. & J. Zweig. "Delegation in C++. Journal of Object-Oriented Programming," 4(11):22-35, November 1991
- Unidraw & Hotdraw drawing tools

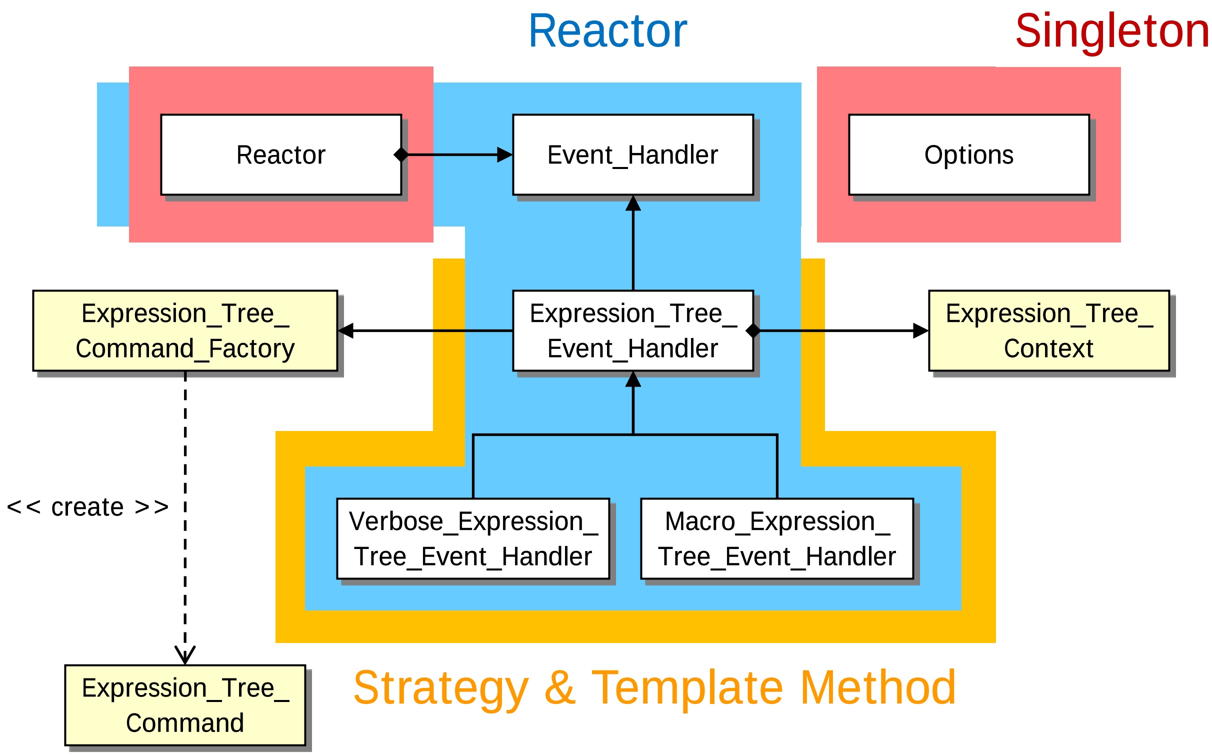
# Summary of State Pattern




Pattern makes the correct ordering of user commands explicit in the design

# Overview of Application Structure Patterns

Purpose: Provide the overall structure of the event-driven application



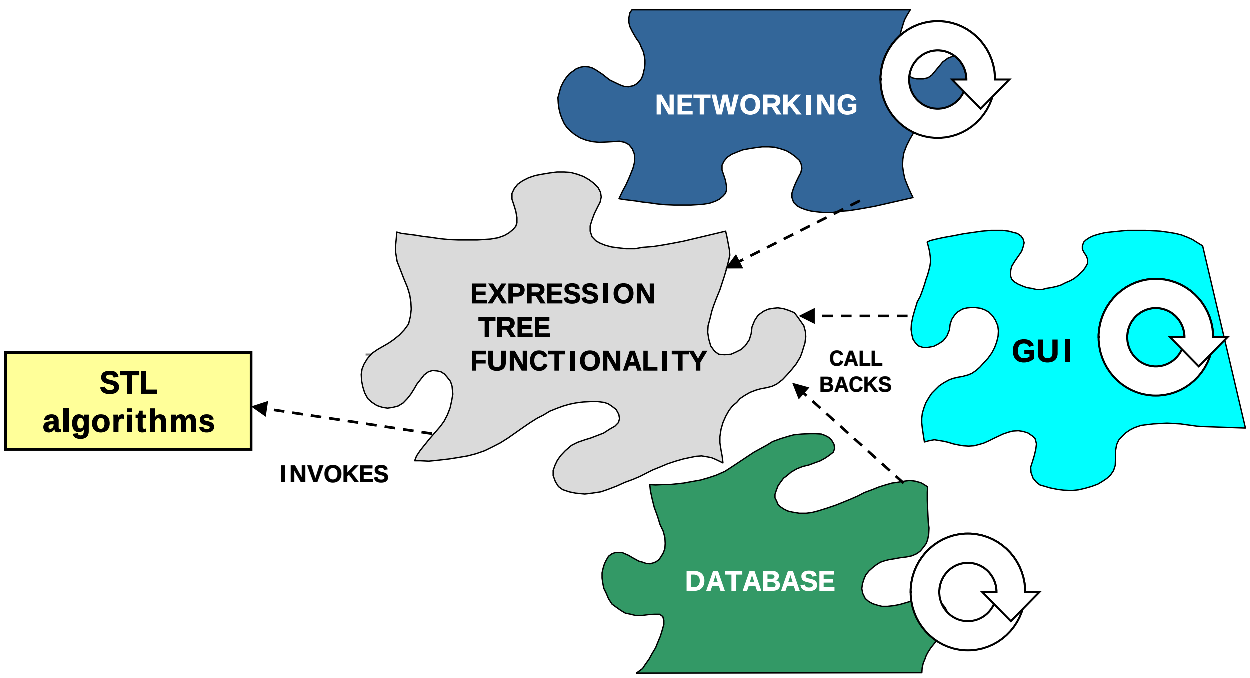
# Driving the Application Event Flow

**Goals:**

- Decouple expression tree application from the context in which it runs
- Support inversion of control

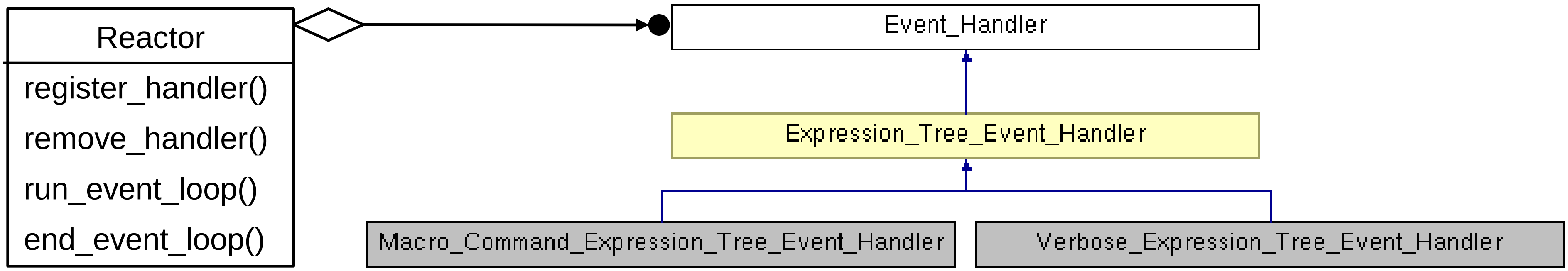
**Constraints/forces:**

- Don't recode existing clients
- Add new event handles without recompiling



# Solution: Separate Event Handling from Event Infrastructure

- Create a reactor to detect input on various sources of events & then demux & dispatch the events to the appropriate event handlers
- Create concrete event handlers that perform the various operational modes of the expression tree application
- Register the concrete event handlers with the reactor
- Run the reactor's event loop to drive the application event flow



# Reactor & Event Handler

An object-oriented event demultiplexor & dispatcher of event handler callback methods in response to various types of events

**Interface:**

virtual void ~Event_Handler (void) =0
virtual void delete_this (void)
virtual void handle_input (void)=0

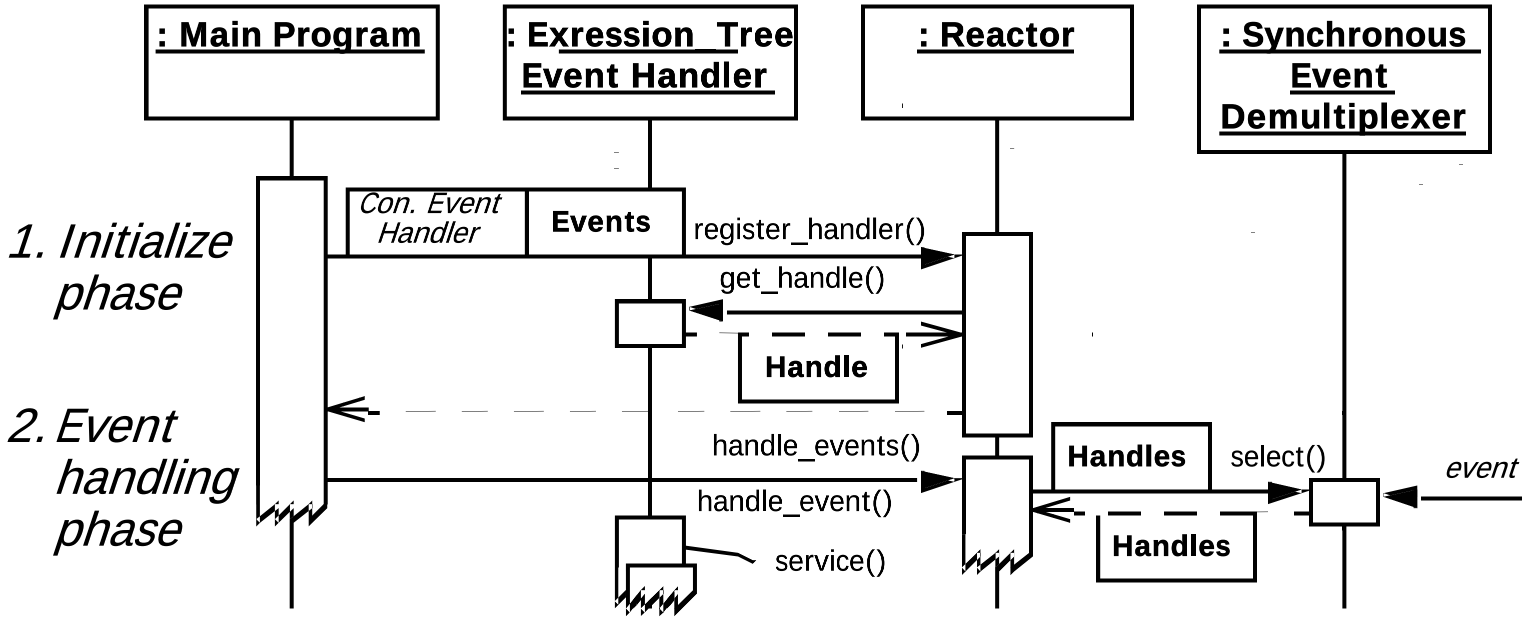
uses

~Reactor (void)
void run_event_loop (void)
void end_event_loop (void)
void register_input_handler (Event_Handler *event_handler)
void remove_input_handler (Event_Handler *event_handler)
static Reactor *instance (void)

**Commonality**: Provides a common interface for managing & processing events via callbacks to abstract event handlers

**Variability**: Concrete implementations of the Reactor & Event_Handlers can be tailored to a wide range of OS demuxing mechanisms & application-specific concrete event handling behaviors

# Reactor Interactions




## Observations

- Note inversion of control
- Also note how long-running event handlers can degrade the QoS since callbacks steal the reactor's thread!

See main.cpp for example of using Reactor to drive event loop

# Reactor object behavioral

**Intent**

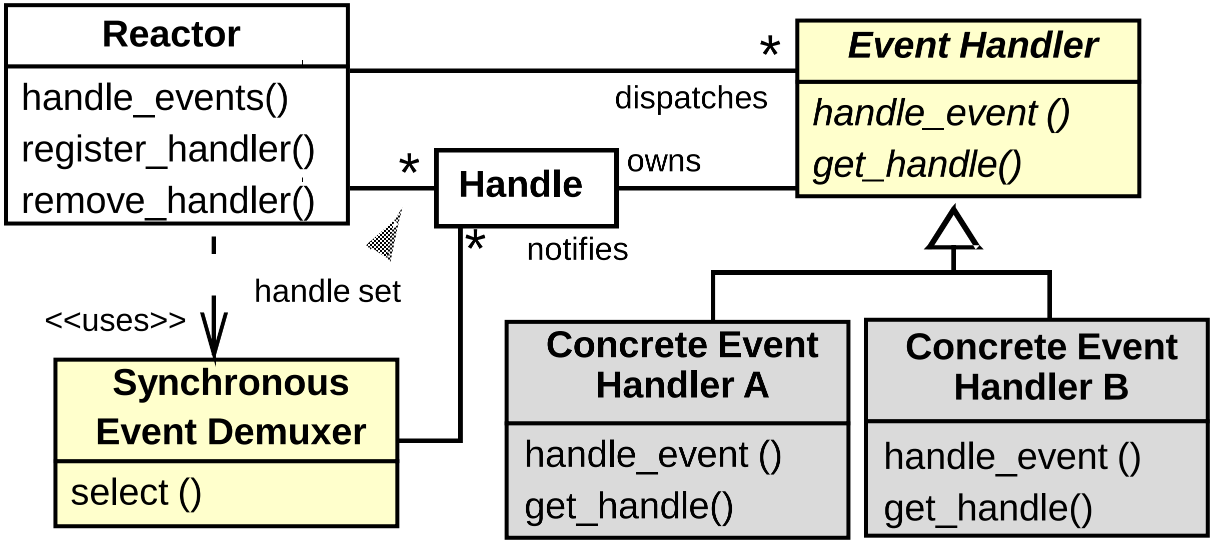
allows event-driven applications to demultiplex & dispatch service requests that are delivered to an application from one or more clients

**Applicability**

– Need to decouple event handling from event detecting/demuxing/dispatching

– When multiple sources of events must be handled in a single thread

**Structure**



# Reactor — object behavioral

**Consequences**

+ Separation of concerns & portability
+ Simplify concurrency control
– Non-preemptive

**Implementation**

- Decouple event demuxing mechanisms from event dispatching
- Handle many different types of events, e.g., input/output events, signals, timers, etc.

**Known Uses**

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- X Windows Xt
- ACE & The ACE ORB (TAO)

# Supporting Multiple Operation Modes

**Goals:**

- Minimize effort required to support multiple modes of operation
  - e.g., verbose & succinct

**Constraints/forces:**

- support multiple operational modes
- don't tightly couple the operational modes with the program structure to enable future enhancements

Verbose mode

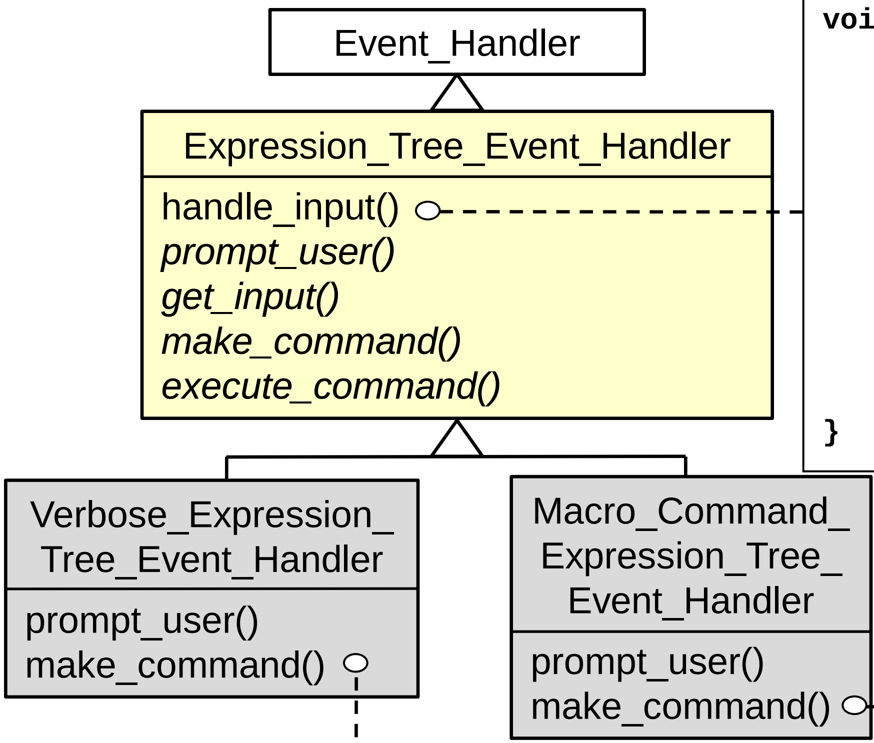
```
% tree-traversal -v
format [in-order]
expr [expression]
print [in-order|pre-order|post-order|level-order]
eval [post-order]
quit
> format in-order
> expr 1+4*3/2
> eval post-order
7
> quit
```

Succinct mode

```
% tree-traversal
> 1+4*3/2
7
```

# Solution: Encapsulate Algorithm Variability

Implement algorithm once in base class & let subclasses define variant parts




```
void handle_input (void) { // template method
   prompt_user (); // hook method

   std::string input;

   if (get_input (input) == false) // hook method
     Reactor::instance ()->end_event_loop ();

   Expression_Tree_Command command
     = make_command (input); // hook method

   if (!execute_command (command)) // hook method
     Reactor::instance ()->end_event_loop ();
}
```

```
Expression_Tree_Command make_command
             (const std::string &input) {
  return
    command_factory_.make_macro_command (input);
}
```

```
Expression_Tree_Command make_command
             (const std::string  &input) {
  return
     command_factory_.make_command (input);
}
```

# Expression_Tree_Event_Handler

Provides an abstract interface for handling input events associated with the expression tree application

**Interface:**

```
virtual void handle_input (void)
static Expression_Tree_Event_Handler * make_handler (bool verbose)
virtual void prompt_user (void)=0
virtual bool get_input (std::string &)
virtual Expression_Tree_Command make_command
                                (const std::string &input)=0
virtual bool execute_command
                                (Expression_Tree_Command &)
```

**Commonality**: Provides a common interface for handling user input events & commands

**Variability**: Subclasses implement various operational modes, e.g., verbose vs. succinct mode

Note **`make_handler()`** factory method variant

# Template Method class behavioral

## Intent

Provide a skeleton of an algorithm in a method, deferring some steps to subclasses

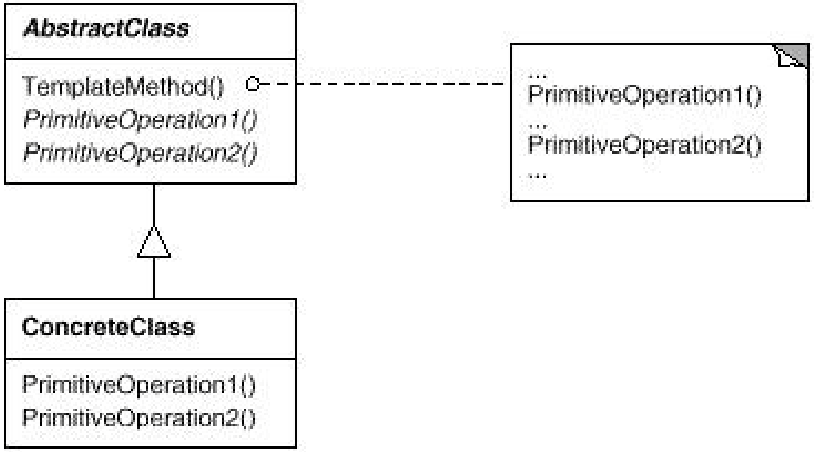
## Applicability

– Implement invariant aspects of an algorithm once & let subclasses define variant parts

– Localize common behavior in a class to increase code reuse

– Control subclass extensions

## Structure



# Template Method class behavioral

## Consequences

+ Leads to inversion of control (“Hollywood principle”: don't call us – we'll call you)
+ Promotes code reuse
+ Lets you enforce overriding rules
– Must subclass to specialize behavior (*cf.* Strategy pattern)

## Implementation

- Virtual vs. non-virtual template method
- Few vs. lots of primitive operations (hook method)
- Naming conventions (do_*() prefix)

## Known Uses

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- ACE & The ACE ORB (TAO)

# Strategy object behavioral

## Intent

define a family of algorithms, encapsulate each one, & make them interchangeable to let clients & algorithms vary independently

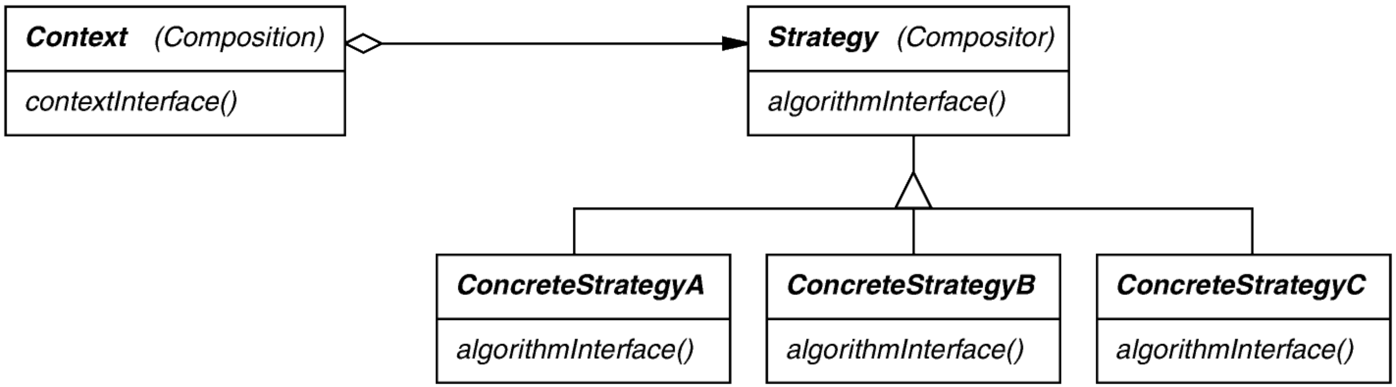
## Applicability

– when an object should be configurable with one of many algorithms,

– *and* all algorithms can be encapsulated,

– *and* one interface covers all encapsulations

## Structure



# Strategy object behavioral

## Consequences

+ greater flexibility, reuse
+ can change algorithms dynamically
– strategy creation & communication overhead
– inflexible Strategy interface
– semantic incompatibility of multiple strategies used together

## Implementation

- exchanging information between a Strategy & its context
- static strategy selection via parameterized types

## Known Uses

- InterViews text formatting
- RTL register allocation & scheduling strategies
- ET++SwapsManager calculation engines
- The ACE ORB (TAO) Real-time CORBA middleware

## See Also

- Bridge pattern (object structural)

# Comparing Strategy with Template Method

**Strategy**

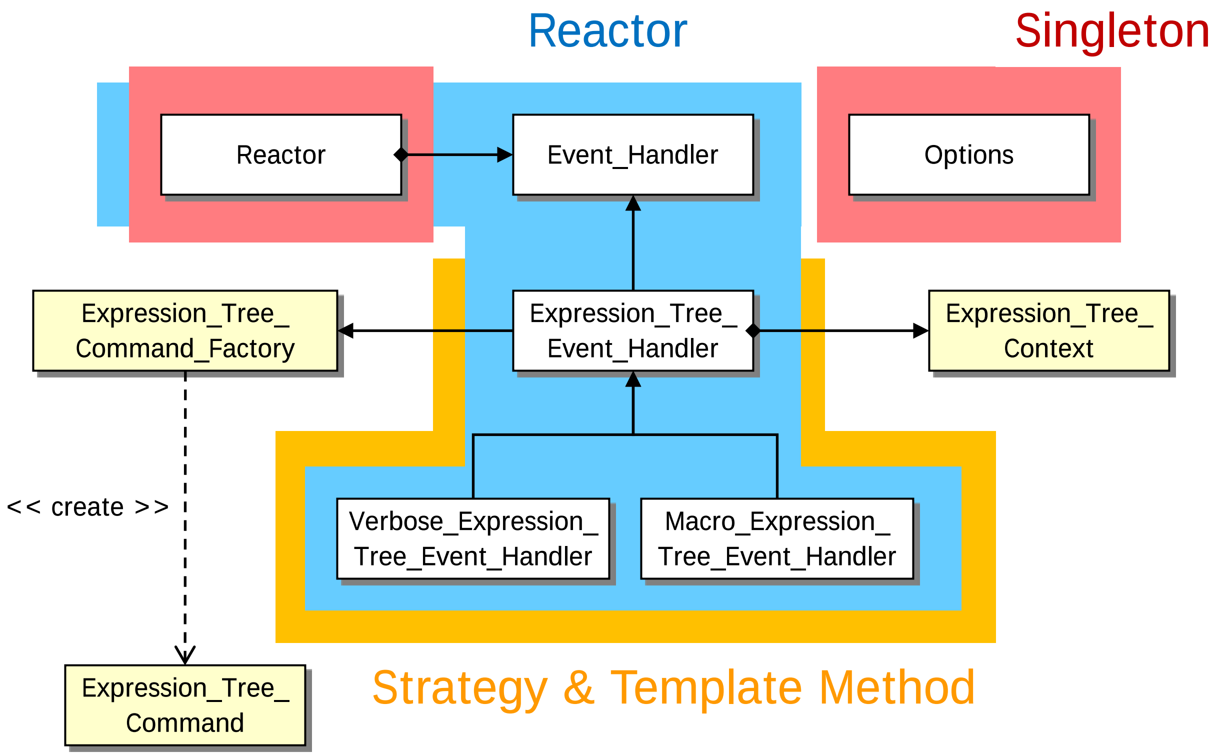
- + Provides for clean separation between components through interfaces
- + Allows for dynamic composition
- + Allows for flexible mixing & matching of features
- – Has the overhead of forwarding
- – Suffers from the identity crisis
- – Leads to more fragmentation

**Template Method**

- + No explicit forwarding necessary
- – Close coupling between subclass(es) & base class
- – Inheritance hierarchies are static & cannot be reconfigured at runtime
- – Adding features through subclassing may lead to a combinatorial explosion
- – Beware of overusing inheritance– inheritance is not always the best choice
- – Deep inheritance hierarchy (6 levels & more) in your application is a red flag

Strategy is commonly used for blackbox frameworks
Template Method is commonly used for whitebox frameworks

# Summary of Application Structure Patterns




Patterns make it easier to support various modes of operations

# Implementing STL Iterator Semantics

**Goals:**

– Ensure the proper semantics of post-increment operations for STL-based **`Expression_Tree_Iterator`** objects

**Constraints/forces:**

– STL pre-increment operations are easy to implement since they simply increment the value & return *this, e.g.,

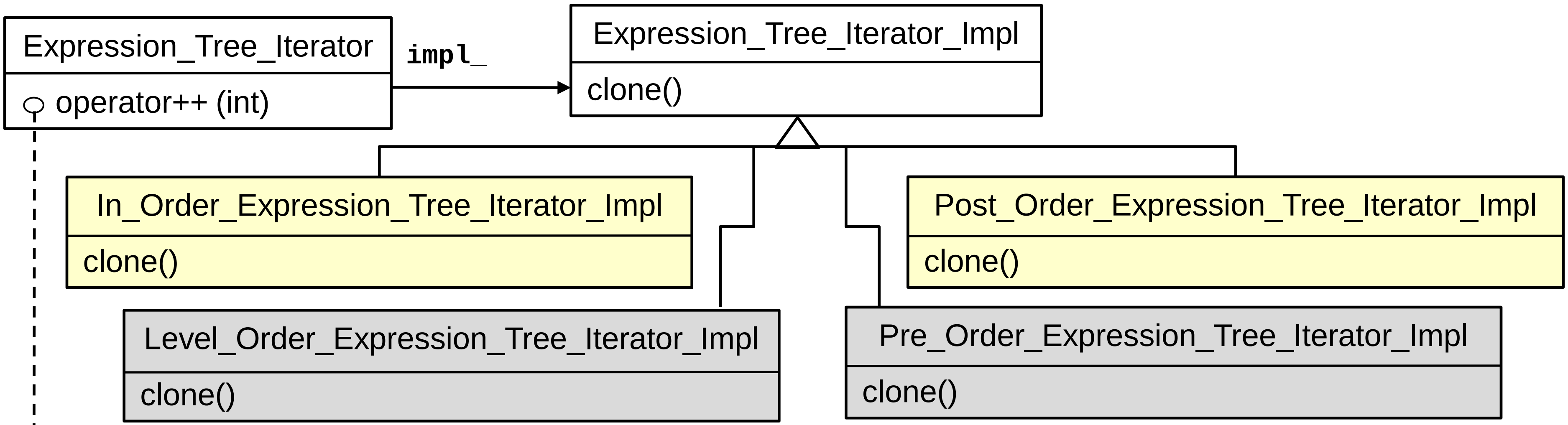
```
iterator &operator++ (void) { ++...; return *this; }
```

– STL post-increment operations are more complicated, however, since must make/return a copy of the existing value of the iterator *before* incrementing its value, e.g.,

```
iterator operator++ (int) {
  iterator temp = copy_*this; ++...; return temp;
}
```

– Since our **`Expression_Tree_Iterator`** objects use the Bridge pattern it is tricky to implement the "***`copy_*this`***" step above in a generic way

# Solution: Clone a New Instance From a Prototypical Instance




```
iterator
Expression_Tree_Iterator::operator++ (int)
{
  iterator temp (impl_->clone ());
  ++(*impl_);
  return temp;
}
```

Note use of Bridge pattern to encapsulate variability & simplify memory management

# Expression_Tree_Iterator_Impl

Implementation of Iterator pattern used to define various iterations algorithms that can be performed to traverse an expression tree

**Interface:**

```
Expression_Tree_Iterator_Impl
    (const Expression_Tree &tree)
virtual Component_Node * operator * (void)=0
void operator++ (void)=0
virtual bool operator== (const Expression_Tree_
    Iterator_Impl &) const=0
virtual bool operator!= (const Expression_Tree_
    Iterator_Impl &) const=0
virtual Expression_Tree_Iterator_Impl * clone (void)=0
```

**Commonality**: Provides a common interface for expression tree iterator implementations

**Variability**: Each subclass implements the **`clone()`** method to return a deep copy of itself

# Prototype object creational

## Intent

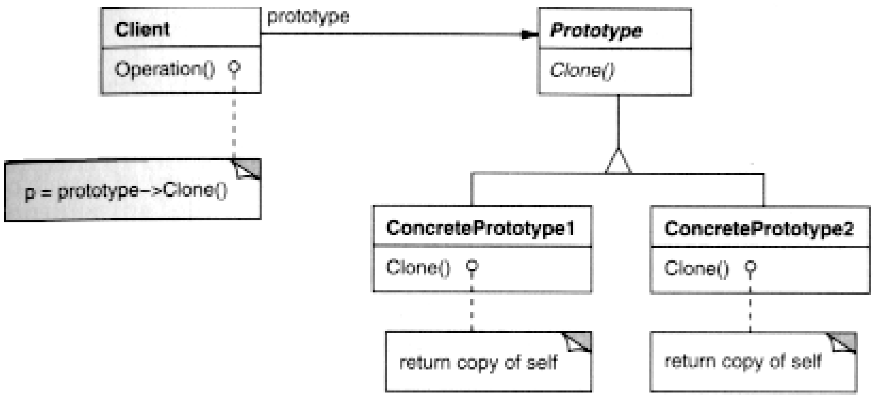
Specify the kinds of objects to create using a prototypical instance & create new objects by copying this prototype

## Applicability

– when a system should be independent of how its products are created, composed, & represented

– when the classes to instantiate are specified at run-time; or

## Structure



# Prototype object creational

## Consequences

+ can add & remove classes at runtime by cloning them as needed
+ reduced subclassing minimizes/eliminates need for lexical dependencies at run-time
– every class that used as a prototype must itself be instantiated
– classes that have circular references to other classes cannot really be cloned

## Implementation

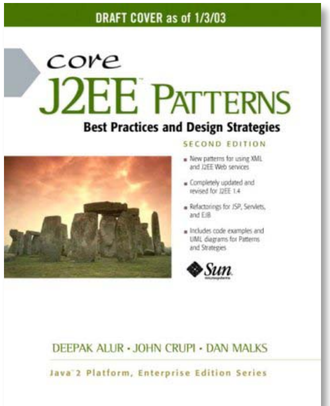
– Use prototype manager
– Shallow vs. deep copies
– Initializing clone internal state within a uniform interface

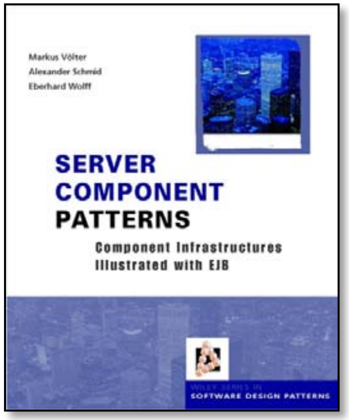
## Known Uses

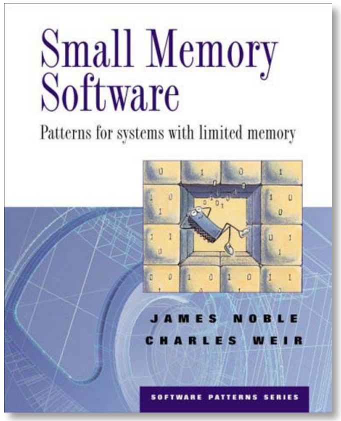
– The first widely known application of the Prototype pattern in an object-oriented language was in ThingLab
– Coplien describes idioms related to the Prototype pattern for C++ & gives many examples & variations
– Etgdb debugger for ET++
– The music editor example is based on the Unidraw drawing framework

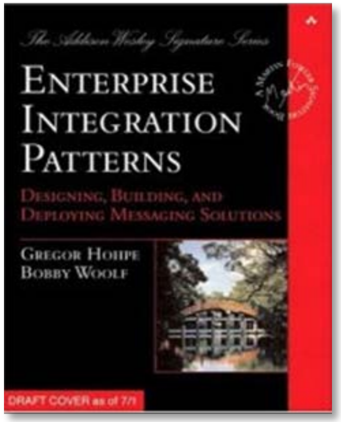
# Part III: Wrap-Up

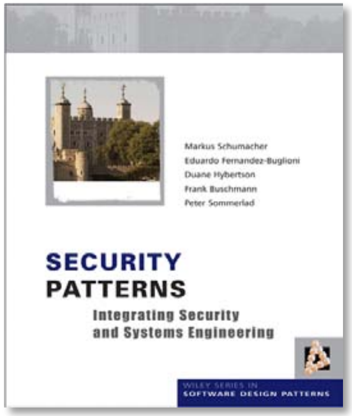
# Life Beyond GoF Patterns

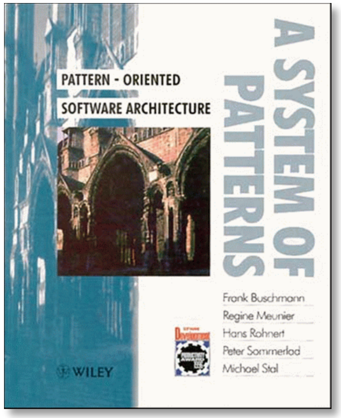


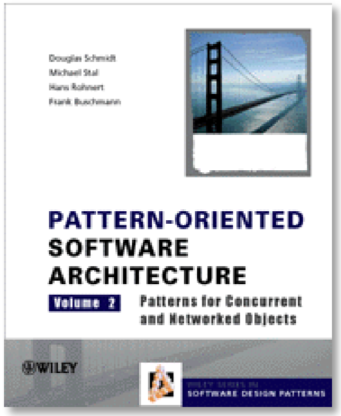



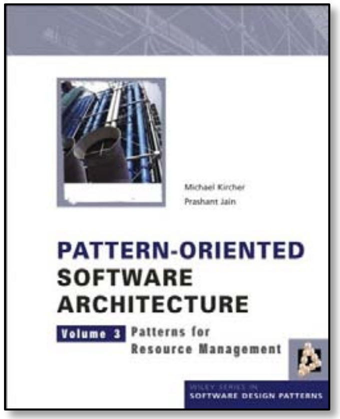



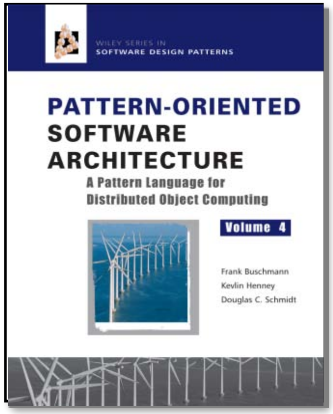



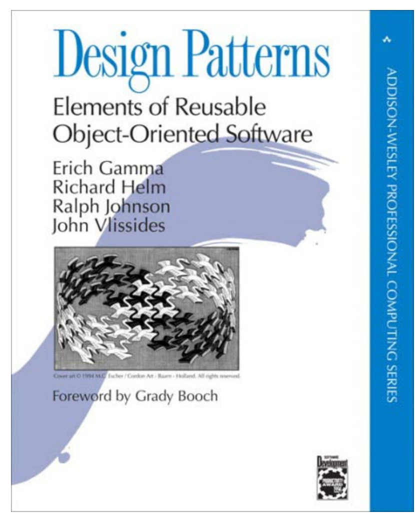


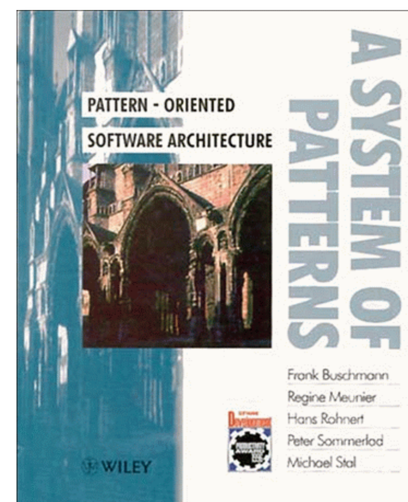
www.cs.wustl.edu/~schmidt/PDF/ieee-patterns.pdf

# Overview of Pattern Collections

- Stand-alone patterns are point solutions to relatively bounded problems that arise within specific contexts
  - e.g., see the PLoPD books


- Any significant software design inevitably includes many patterns, however, which means that a stand-alone pattern unusual in practice
- A common presentation of multiple patterns is in the form of a *pattern collection*
  - e.g., the GoF & POSA1 books



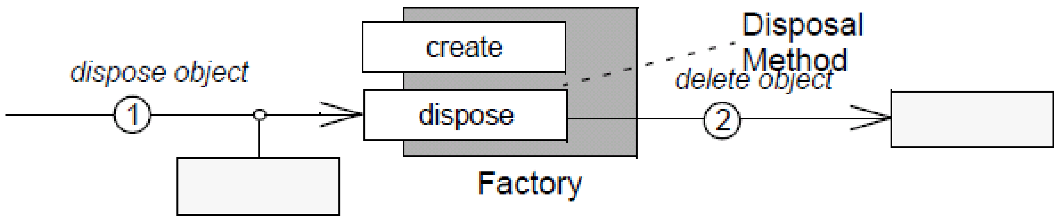
# Overview of Pattern Relationships

- Patterns representing the foci for discussion, point solutions, or localized design ideas can be used in isolation with some success
- Patterns are generally gregarious, however, in that they form relationships with other patterns
- Four of the most common types of pattern relationships include:

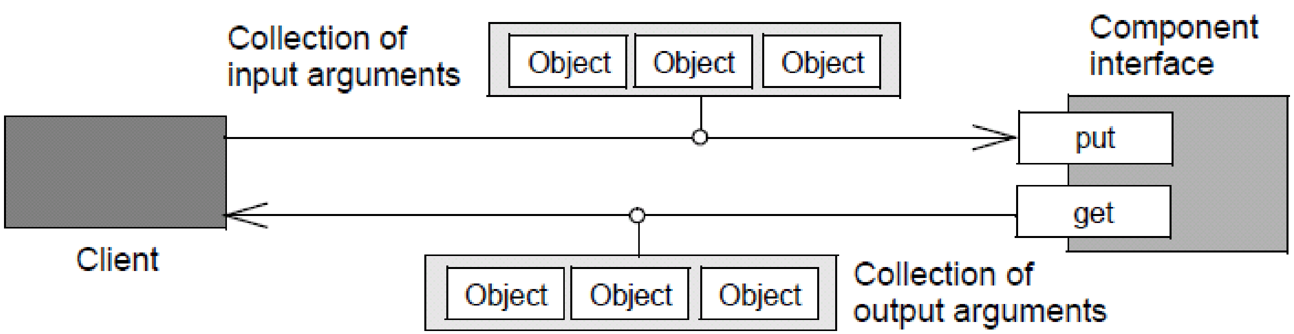
1. ***Patterns complements***, where one pattern provides the missing ingredient needed by another or where one pattern contrasts with another by providing an alternative solution to a related problem
2. ***Pattern compounds*** capture recurring subcommunities of patterns that are common & identifiable enough that they can be treated as a single decision in response to a recurring problem
3. ***Pattern sequences*** generalize the progression of patterns & the way a design can be established by joining predecessor patterns to form part of the context of each successive pattern
4. ***Pattern languages*** define a vocabulary for talking about software development problems & provide a process for the orderly resolution of these problems

# Overview of Pattern Complements

- One pattern provides missing ingredient needed by another—or where one pattern competes with another by providing an alternative solution to a related problem—to make resulting designs more complete & balanced, e.g.:
  - *Disposal Method* complements *Factory Method* by addressing object destruction & creation, respectively, in the same design

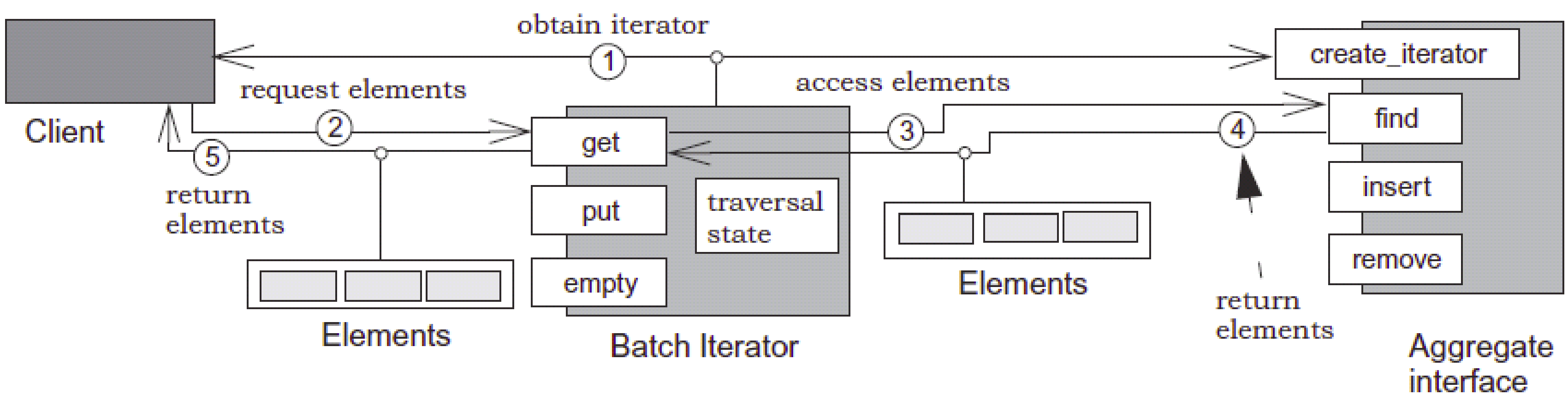



  - *Batch Method* competes with *Iterator* by accessing the elements of an aggregate in bulk, reducing roundtrip network costs



# Overview of Pattern Compounds

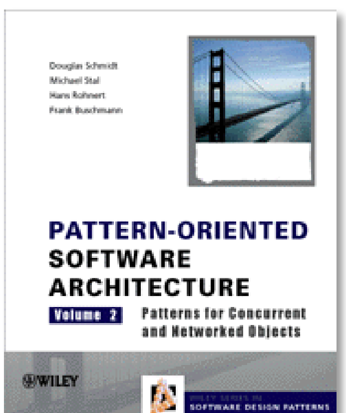
- Pattern compounds capture recurring subcommunities of patterns that are common & identifiable enough that they can be treated as a single decision in response to a recurring problem
- For example, *Batch Iterator* brings together two complementary patterns, *Iterator* & *Batch Method*, to address the problem of remotely accessing the elements of aggregates with large numbers of elements
- A *Batch Iterator* refines the position-based traversal of an *Iterator* with a *Batch Method* for bulk access of many, but not all, elements



# Overview of Pattern Sequences

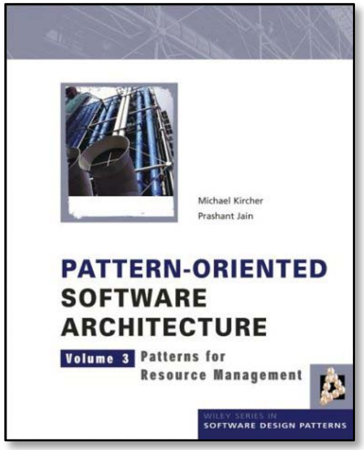
- *Pattern sequences* generalize the progression of patterns & the way a design can be established by joining predecessor patterns to form the context of each successive pattern
- A pattern sequence captures the unfolding of a design or situation, pattern-by-pattern
- e.g., POSA2 & POSA4 present pattern sequences for communication middleware

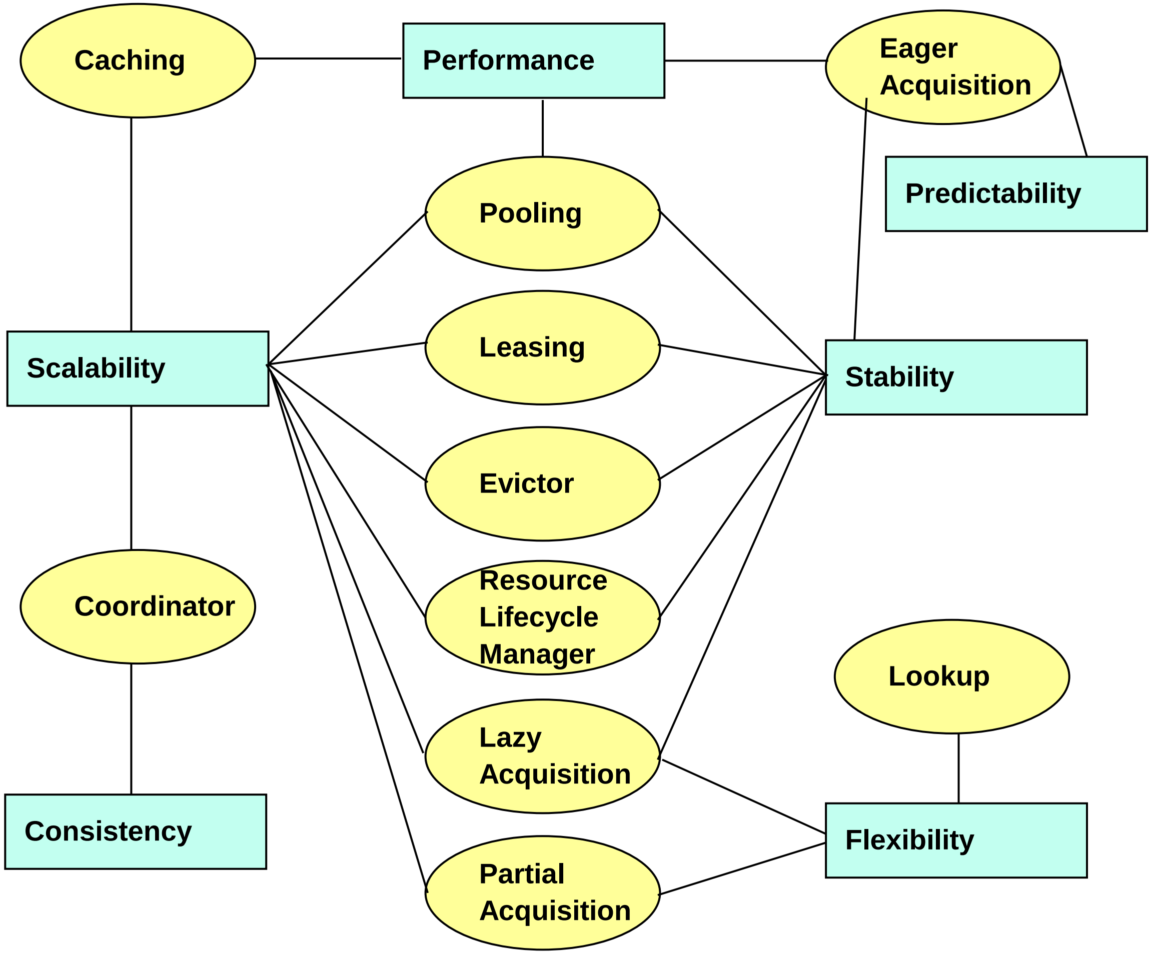
| Pattern | Challenges |
|---|---|
| Broker | Defining the ORB's base-line architecture |
| Layers | Structuring ORB internal design to enable reuse and clean separation of concerns |
| Wrapper Facade | Encapsulating low-level system functions to enhance portability |
| Reactor | Demultiplexing ORB Core events effectively |
| Acceptor-Connector | Managing ORB connections effectively |
| Half-Sync/Half-Async | Enhancing ORB scalability by processing requests concurrently |
| Monitor Object | Efficiently synchronize the Half-Sync/Half-Async request queue |
| Strategy | Interchanging internal ORB mechanisms transparently |
| Abstract Factory | Consolidating ORB mechanisms into groups of semantically compatible strategies |
| Component Configurator | Configuring consolidated ORB strategies dynamically |




# Overview of Pattern Sequences

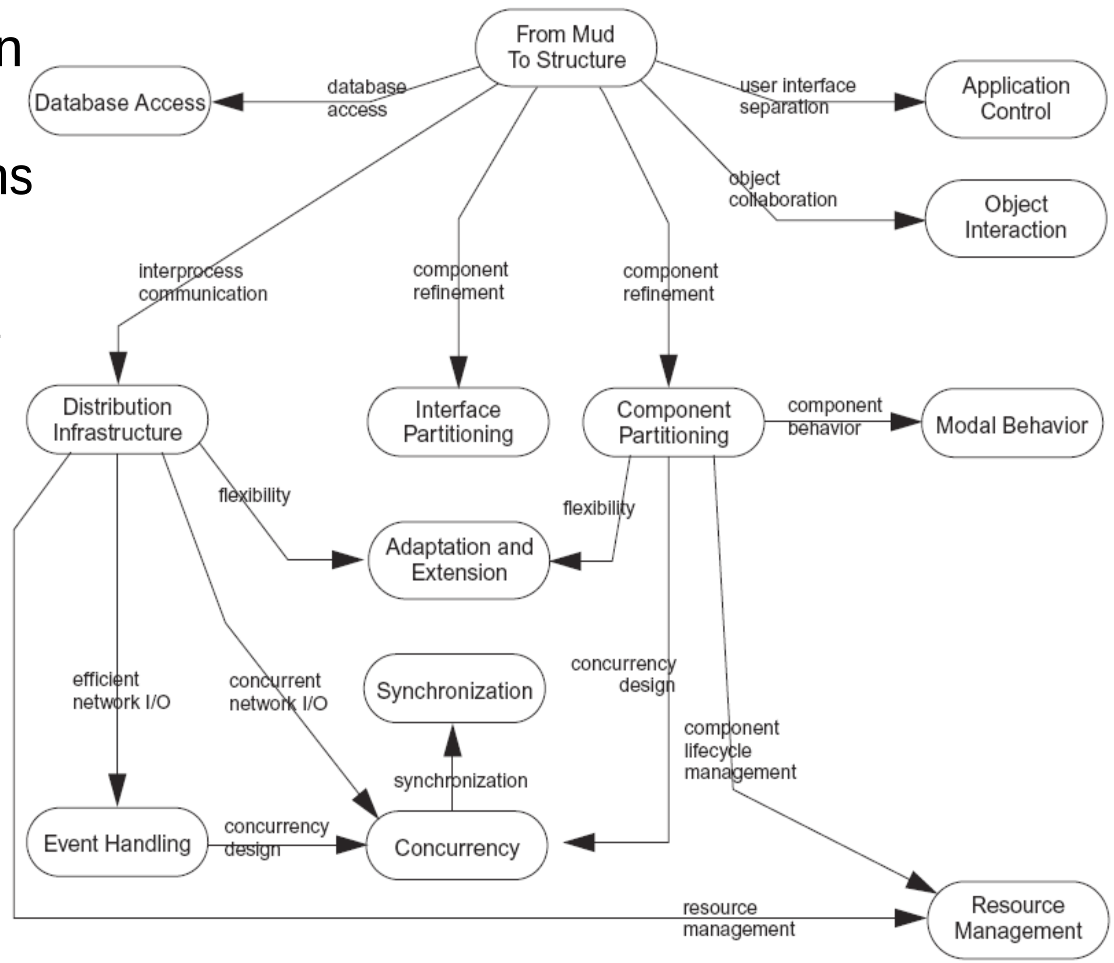
- *Pattern sequences* generalize the progression of patterns & the way a design can be established by joining predecessor patterns to form the context of each successive pattern
- A pattern sequence captures the unfolding of a design or situation, pattern-by-pattern
- e.g., POSA3 presents pattern sequences for resource management



# Overview of Pattern Languages

- *Pattern languages* define a vocabulary for talking about software development problems & provide a process for the orderly resolution of these problems
- For example, the POSA4 pattern language for distributed computing includes 114 patterns grouped into 13 problem areas
- Each problem area addresses a specific technical topic related to building distributed systems
- POSA5 describes key concepts of pattern languages




# Observations on Applying Patterns & Frameworks

Patterns & frameworks support

- design/implementation at a more abstract level
  - treat many class/object interactions as a unit
  - often beneficial *after* initial design
  - targets for class refactorings
- Variation-oriented design/implementation
  - consider what design aspects are variable
  - identify applicable pattern(s)
  - vary patterns to evaluate tradeoffs
  - repeat

Patterns are applicable in all stages of the OO lifecycle

- analysis, design, & reviews
- realization & documentation
- reuse & refactoring

Patterns often equated with OO languages, but many also apply to C

# Caveats to Keep in Mind

Don’t apply patterns & frameworks blindly

- Added indirection can yield increased complexity, cost
- Understand patterns to learn how to better develop/use frameworks

Resist branding everything a pattern

- Articulate specific benefits
- Demonstrate wide applicability
- Find at least *three* existing examples from code other than your own!

Pattern & framework design even harder than OO design!

# Concluding Remarks

- Patterns & frameworks promote
  - *Integrated* design & implementation reuse
  - uniform design vocabulary
  - understanding, restructuring, & team communication
  - a basis for automation
  - a "new" way to think about software design & implementation

There's much more to patterns than just the GoF, however!!!!

# Pattern References

**Books**

*Timeless Way of Building*, Alexander, ISBN 0-19-502402-8

*A Pattern Language*, Alexander, 0-19-501-919-9

*Design Patterns*, Gamma, et al., 0-201-63361-2 CD version 0-201-63498-8

*Pattern-Oriented Software Architecture*, *Vol. 1*, Buschmann, et al., 0-471-95869-7

*Pattern-Oriented Software Architecture, Vol. 2*, Schmidt, et al., 0-471-60695-2

*Pattern-Oriented Software Architecture, Vol. 3*, Jain & Kircher, 0-470-84525-2

*Pattern-Oriented Software Architecture, Vol. 4*, Buschmann, et al., 0-470-05902-8

*Pattern-Oriented Software Architecture, Vol. 5*, Buschmann, et al., 0-471-48648-5

# Pattern References (cont'd)


**More Books**

*Analysis Patterns*, Fowler; 0-201-89542-0

*Concurrent Programming in Java, 2nd ed.*, Lea, 0-201-31009-0

*Pattern Languages of Program Design*
Vol. 1, Coplien, et al., eds., ISBN 0-201-60734-4
Vol. 2, Vlissides, et al., eds., 0-201-89527-7
Vol. 3, Martin, et al., eds., 0-201-31011-2
Vol. 4, Harrison, et al., eds., 0-201-43304-4

Vol. 5, Manolescu, et al., eds., 0-321-32194-4

*AntiPatterns*, Brown, et al., 0-471-19713-0

*Applying UML & Patterns, 2nd ed.*, Larman, 0-13-092569-1

*Pattern Hatching*, Vlissides, 0-201-43293-5

*The Pattern Almanac 2000*, Rising, 0-201-61567-3

# Pattern References (cont'd)

**Even More Books**

*Small Memory Software*, Noble & Weir, 0-201-59607-5

*Microsoft Visual Basic Design Patterns*, Stamatakis, 1-572-31957-7

*Smalltalk Best Practice Patterns*, Beck; 0-13-476904-X

*The Design Patterns Smalltalk Companion*, Alpert, et al., 0-201-18462-1

*Modern C++ Design*, Alexandrescu, ISBN 0-201-70431-5

*Building Parsers with Java*, Metsker, 0-201-71962-2

*Core J2EE Patterns*, Alur, et al., 0-130-64884-1

*Design Patterns Explained*, Shalloway & Trott, 0-201-71594-5

*The Joy of Patterns*, Goldfedder, 0-201-65759-7

*The Manager Pool*, Olson & Stimmel, 0-201-72583-5

# Pattern References (cont'd)

## Early Papers

“Object-Oriented Patterns,” P. Coad; Comm. of the ACM, 9/92

“Documenting Frameworks using Patterns,” R. Johnson; OOPSLA ’92

“Design Patterns: Abstraction & Reuse of Object-Oriented Design,” Gamma, Helm, Johnson, Vlissides, ECOOP ’93

## Articles

Java Report, Java Pro, JOOP, Dr. Dobb’s Journal, Java Developers Journal, C++ Report

## How to Study Patterns

http://www.industriallogic.com/papers/learning.html

# Pattern-Oriented Conferences

**PLoP 2011**: Pattern Languages of Programs
October 2011, Collocated with SPLASH (formerly OOPSLA)

**EuroPLoP 2012**, July 2012, Kloster Irsee, Germany

**...**

See hillside.net/conferences/ for
up-to-the-minute info

# Mailing Lists

**patterns@cs.uiuc.edu:** present & refine patterns

**patterns-discussion@cs.uiuc.edu:** general discussion

**gang-of-4-patterns@cs.uiuc.edu:** discussion on *Design Patterns*

**siemens-patterns@cs.uiuc.edu:** discussion on *Pattern-Oriented Software Architecture*

**ui-patterns@cs.uiuc.edu:** discussion on user interface patterns

**business-patterns@cs.uiuc.edu:** discussion on patterns for business processes

**ipc-patterns@cs.uiuc.edu:** discussion on patterns for distributed systems

See http://hillside.net/patterns/mailing.htm for an up-to-date list.