CS189A/172 - Winter 2010

Lectures 13 and 14: Design Patterns

What are Design Patterns?

- Think about the common data structures you learned
  - Trees, Stacks, Queues, etc.
- These data structures provide a set of tools on how to organize data
- Probably you implement them slightly differently in different projects
- Main concepts about these data structures, such as
  - how to store them
  - manipulation algorithms

  are well understood
- You can easily communicate these data structures to another software developer by just stating their name
- Knowing them helps you when you are dealing with data organization in your software projects
  - Better than re-inventing the wheel

What are Design Patterns?

- This is the question:
  - Are there common ideas in architectural design of software that we can learn (and give a name to) so that
    - We can communicate them to other software developers
    - We can use them in architectural design in a lot of different contexts (rather than re-inventing the wheel)

- The answer is yes according to E. Gamma, R. Helm, R. Johnson, J. Vlissides
  - They developed a catalog of design patterns that are common in object oriented software design

What are Design Patterns?

- Design patterns provide a mechanism for expressing common object-oriented design structures
- Design patterns identify, name and abstract common themes in object-oriented design
- Design patterns can be considered micro architectures that contribute to overall system architecture
- Design patterns are helpful
  - In developing a design
  - In communicating the design
  - In understanding a design

Origins of Design Patterns

- The origins of design patterns are in architecture (not in software architecture)
- Christopher Alexander, a professor of architecture at UC Berkeley, developed a pattern language for expressing common architectural patterns
- Work of Christopher Alexander inspired the work of Gamma et al.
- In explaining the patterns for architecture, Christopher Alexander says:

  “*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice*”
- These comments also apply to software design patterns

Resources for Design Patterns

- “Design Patterns: Abstraction and Reuse of Object-Oriented Design” by E. Gamma, R. Helm, R. Johnson, J. Vlissides
  - Original paper
- Later on same authors published a book which contains an extensive catalog of design patterns:
  - “Design Patterns: Elements of Reusable Object-Oriented Software”, by E. Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, ISBN 0-201-63361-2
  - This book is a catalog of design patterns. I recommend it.
- A more recent book
  - “Design Patterns Explained”, by A. Shalloway and J. R. Trott, Addison-Wesley, ISBN: 0-201-71594-5
- Design patterns resources at Doug Schmidt’s webpage (including tutorials):
  - http://www.cs.wustl.edu/~schmidt/patterns.html

Cataloging Design Patterns

- Gamma et al. present:
  - A way to describe design patterns
  - A way to organize design patterns by giving a classification system

- More importantly, in their book on design patterns, the authors give a catalog of design patterns
  - As a typical developer you can use patterns from this catalog
  - If you are a good developer you can contribute to the catalog by discovering and reporting new patterns

- The template for describing design patterns used by Gamma et al. is given in the next slide

Design Pattern Template

DESIGN PATTERN NAME the name should convey pattern's essence succinctly	**Jurisdiction Characterization** used for categorization

Intent

What particular design issue or problem does the design pattern address?

Motivation

A scenario in which the pattern is applicable. This will make it easier to understand the more abstract description that follows.

Applicability

What are the situations the design pattern can be applied?

Participants

Describe the classes and/or objects participating in the design pattern and their responsibilities.

Collaborations

Describe how the participants collaborate to carry out their responsibilities.

Diagram

A class diagram representation of the pattern (extended with pseudo-code).

Consequences

What are the trade-offs and results of using the pattern?

Implementation

What pitfalls, hints, or techniques should one be aware of when implementing the pattern?

Examples

Examples of applications of the pattern in real systems.

See Also

What are the related patterns and what are their differences?

Case Study: A Text Editor

- A case study from the Design Patterns book by Gamma et al.

- Use of design patterns in designing a
  - WYSIWYG document editor

Case Study: A Text Editor

- Issues to be addressed in designing the editor
  - Document structure. How should we store the document internally?
  - Formatting. How should we arrange text and graphics into lines and columns?
  - User interface includes scroll bars, borders, etc. and it should be extensible to include other embellishments.
  - The editor should support multiple look-and-feel standards.
  - The editor should work in multiple window systems.
  - There should be a uniform way to deal with user operations (and possibly undo them).
  - How to traverse the document for operations such as spell-checking?

Problem 1: Document Structure

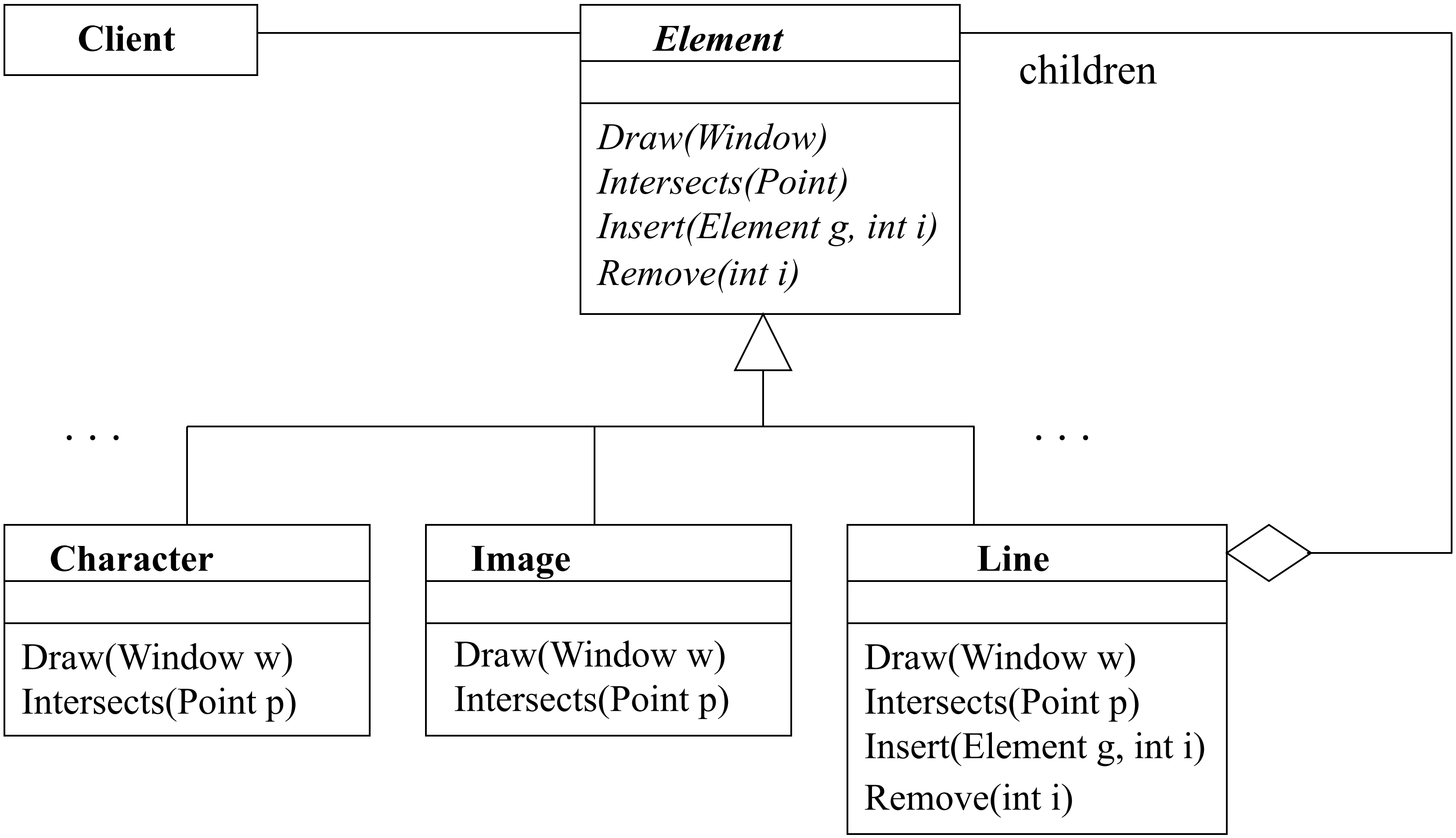
- The document contains:
  - Primitive elements which are not decomposable
    - such as characters and images
  - Composed elements
    - lines: a list of primitive elements
    - columns: a list of lines
    - pages: a list of columns
    - documents: a list of pages

- What class structure should we use to store these document elements?

Document Structure: Possible Solutions

- A solution:
  - Create a class for each element: Character, Image, Line, Column, Page, Document
  - Each composed element has a list of objects of some type for its parts
  - Problem: Not flexible, if we add a new kind of element we need to change other classes
  - Problem: There is no way to uniformly treat all the elements
- A better solution:
  - Use an abstract class for all elements
  - Each element is realized by a subclass of the abstract class
  - All elements have the same interface defined by the abstract class
    - How to draw, insert, etc.
  - Treats all elements uniformly
  - It is easy to extend the class structure with new elements

Class Diagram



Operations

- `Draw(Window w)`
  - Each primitive element draws itself to a window
  - Each compound element calls `Draw` method of each of its children to draw itself
- `Intersects(Point p)`
  - Each primitive element checks if it intersects a point
  - A compound element intersects a point if one of its elements intersect that point (i.e., a compound elements' `Intersects` method calls `Intersects` methods of its children)
- `Child(int i)`
  - Returns a child of a compound element
- `Insert(Element g, int i)`, `Remove(int i)`
  - Compound elements have `Insert` and `Remove` operations (to insert or remove their children) whereas basic elements do not

Operations

- Following operations are not supported by the primitive elements (like `Character` and `Image`) but are only supported by compound elements (like `Line`)
  - `Child(int i), Insert(Element g, int i), Remove(int i)`
- So if you make these operations part of the abstract base class `Element` (as shown in the previous slide), then at runtime you have to make sure that if one of these operations are called on primitive elements, either an exception is raised or the call has no effect (i.e., it is a no-op).
- Alternatively you can remove these operations from the abstract base class to make sure that they are never called on instances of primitive elements. However, this means that you lose some of the uniformity between the primitive and compound elements.

Recursive Composition: Composite Pattern

- The design pattern we used is called the **Composite Pattern**
  - aka recursive composition

- Composite Pattern is one of the entries in the design catalog

- It can be used in all cases where you have a hierarchical structure and leaves and internal nodes share the same functionality
  - For example my research group used it in our automated verification tool for storing logic formulas

- Let’s look at the catalog entry for Composite Pattern in the Design Patterns book

COMPOSITE **Compound Structural**

Intent

Composite lets clients treat individual objects and compositions of objects uniformly.

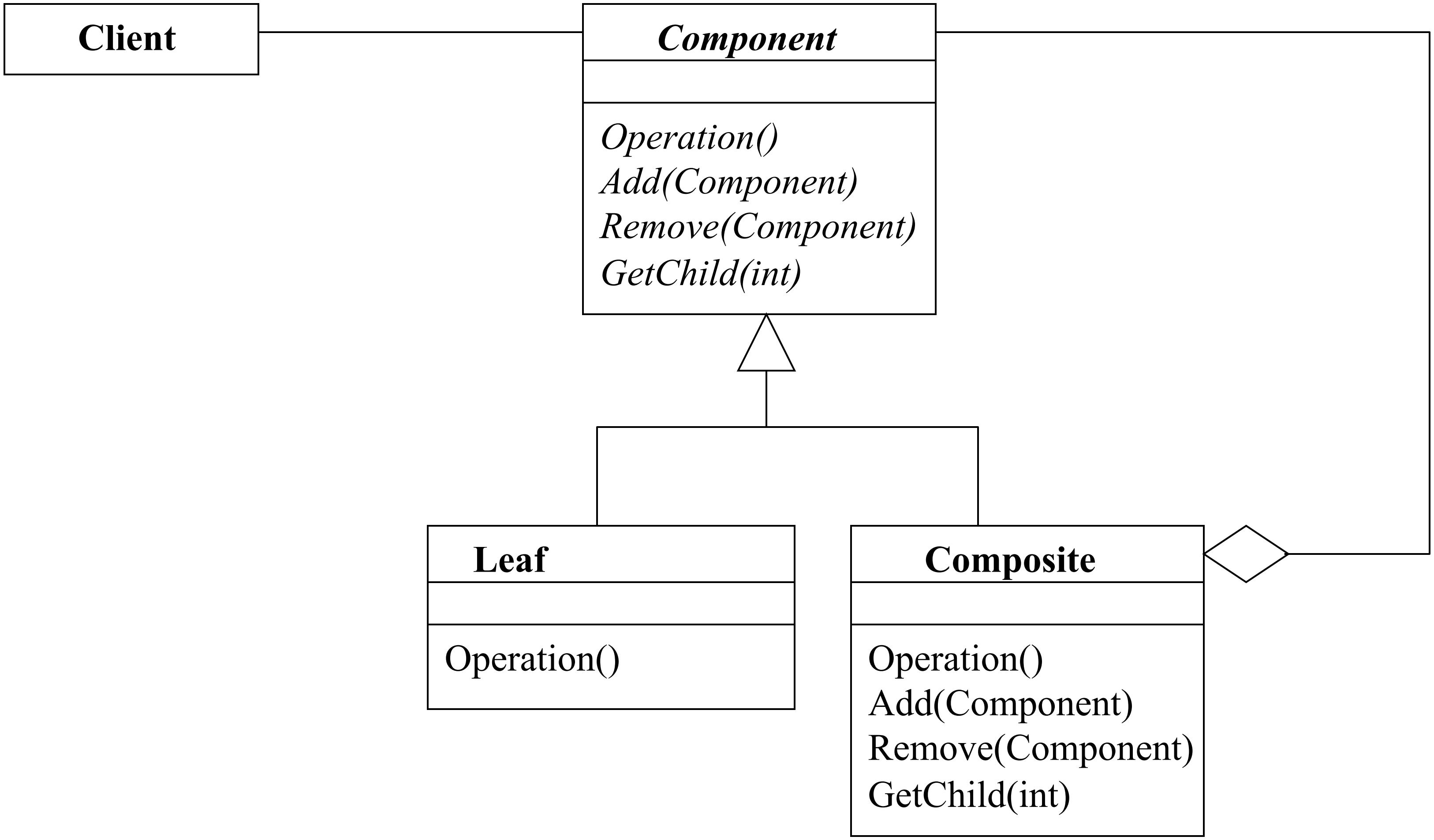
Motivation

Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of single components. The user can group components to form larger components.

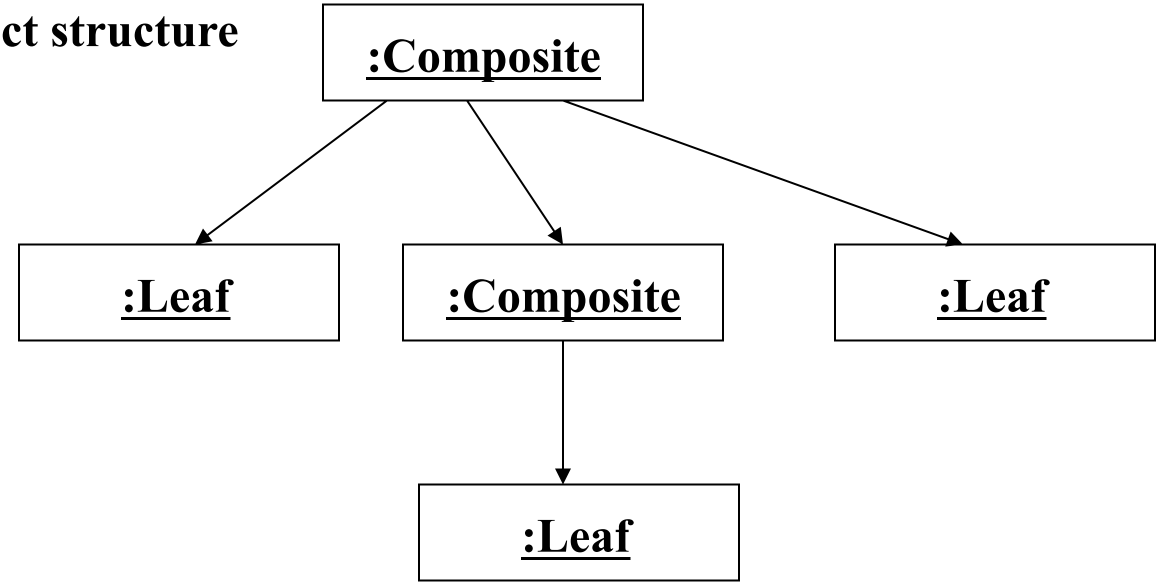
The code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically. Having to distinguish these objects makes the application more complex. Composite pattern describes how to use recursive composition so the clients do not have to make this distinction.

The key to Composite pattern is an abstract class that represents both primitives and their containers.

Class Diagram for the Composite Pattern



A Composite object structure



Digression on UML:

- The above diagram is an object diagram
- Each rectangle in an object diagram represents an object
- Objects in UML are written as <u>name:ClassName</u>
- If object name is not provided then we call it an anonymous object
- Attribute values of an object can be written in the rectangle representing the object
- The arcs in the diagram show the links between different objects (which are instantiations of the associations among different classes)

Applicability

Use Composite pattern when

- you want to represent part-whole hierarchies of objects
- you want client to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Participants

- **Component**
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all classes
  - declares an interface for accessing and managing child components
- **Leaf**
  - represent leaf objects, does not have any children
  - defines behavior of primitive objects in the composition
- **Composite**
  - defines behavior for components having children
  - stores child components
- **Client**
  - manipulates objects in the composition through Component interface

Another Case Study: Representing Expressions

- Assume that we would like to implement a set of classes for representing and manipulating expressions

- These classes can be used in a compiler implementation

- We need to store the expressions in some form (i.e., abstract syntax tree)

- We need to perform operations on the expressions such as
  - printing
  - type checking

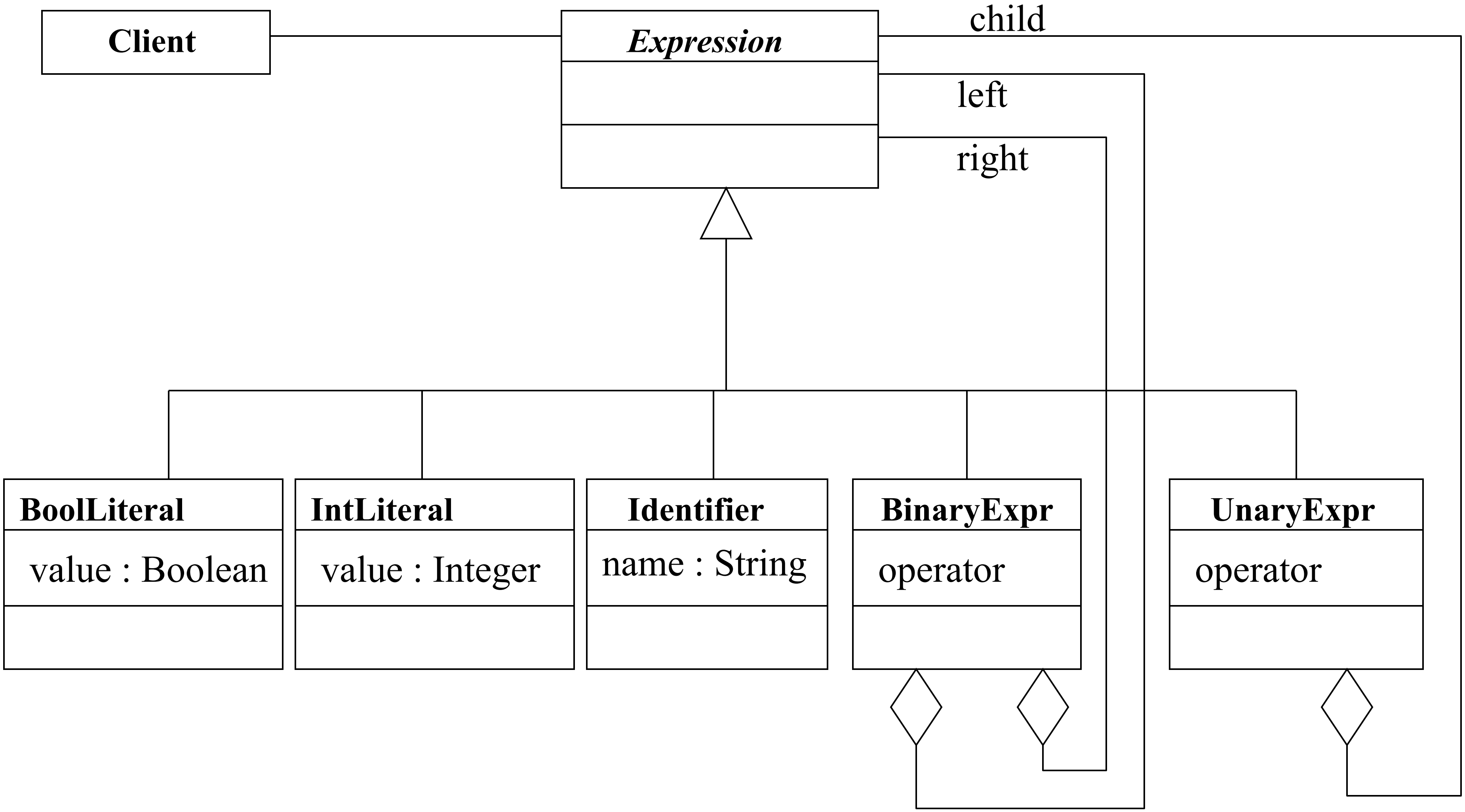
Problem 1: How To Represent Expressions?

- There are different types of expressions such as:
  - boolean literal, integer literal, identifier, binary expression, unary expression, etc.

- Different types of expressions have different attributes so it would make sense to have a different class for each expression type

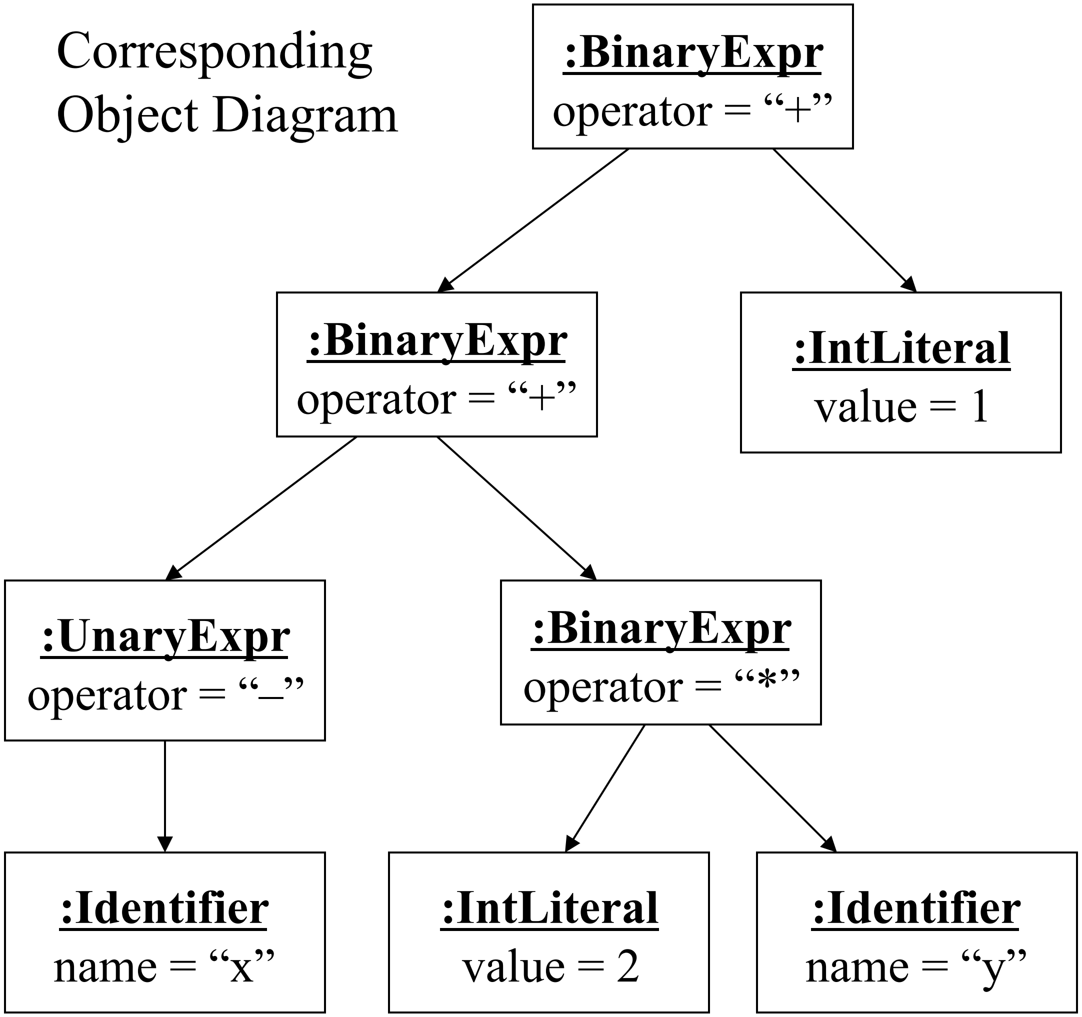
- However, we should be able to treat expressions uniformly
  - For example, children of binary expressions or unary expressions could be any type of expression

Class Diagram for Expressions



An Expression:
– x + 2 * y + 1

Corresponding
Object Diagram



:BinaryExpr
operator = “+”

:BinaryExpr
operator = “+”

:IntLiteral
value = 1

:UnaryExpr
operator = “–”

:BinaryExpr
operator = “*”

:Identifier
name = “x”

:IntLiteral
value = 2

:Identifier
name = “y”

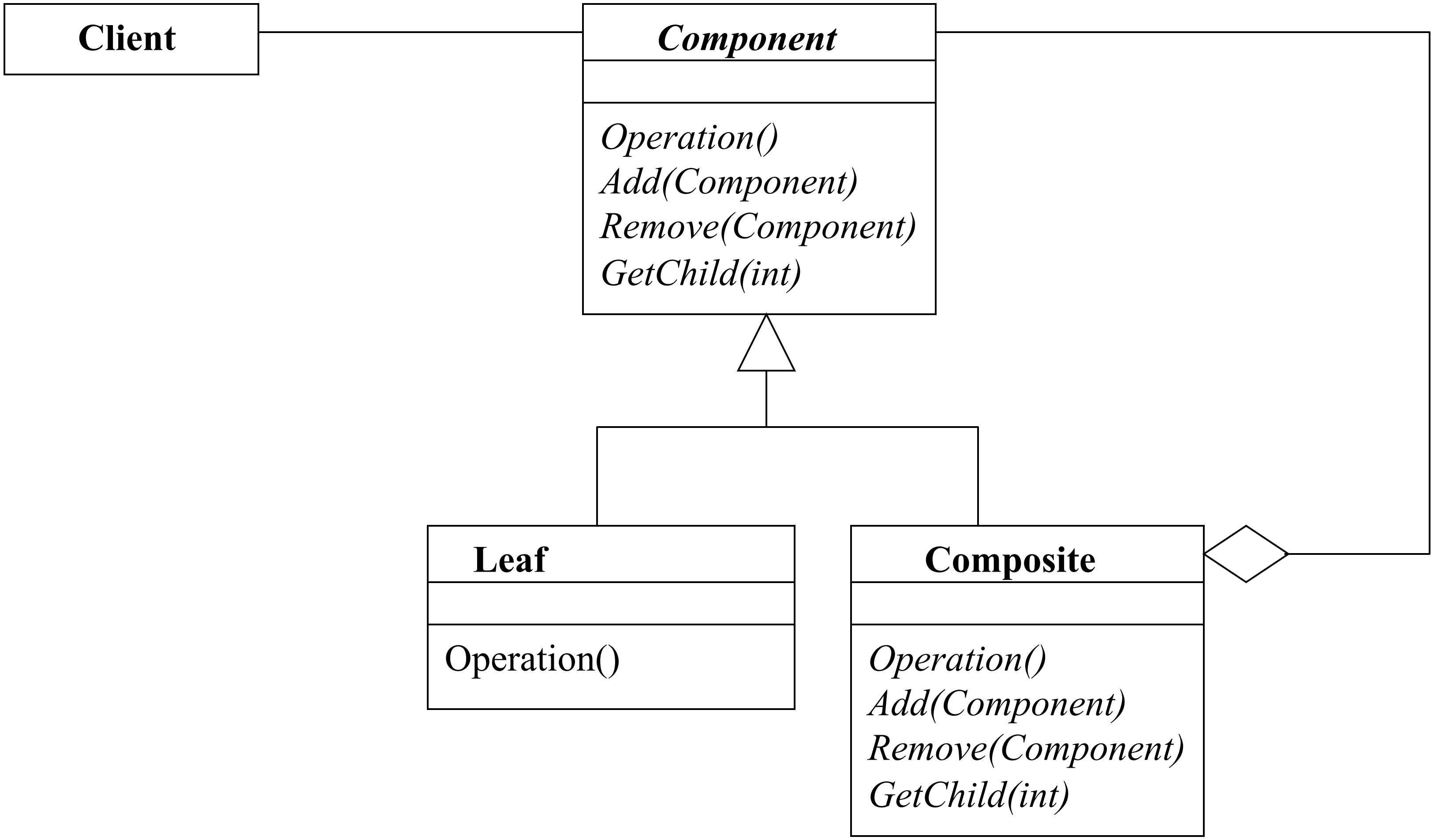
Using the Composite Pattern

- Using composite pattern enables us to treat all the expressions uniformly using the Expression interface

```
Expression e1 = new Identifier(...);
Expression e2 = new BoolLiteral(...);
Expression e3 = new BinaryExpr(e1, e2, ...);

...

printer.printSomeEpxression(e3);

...

public void printSomeExpression(Expression e) {
  e.print();
}
```

client code may not need to know what type of expression e is

We Used the Composite Pattern



Back to the Text Editor Case Study: Problem 2: Formatting

- Formatting decides the physical layout of the document
  - For example formatting decides on how to break a set of Elements to lines, or how to break a set of lines to columns, etc.

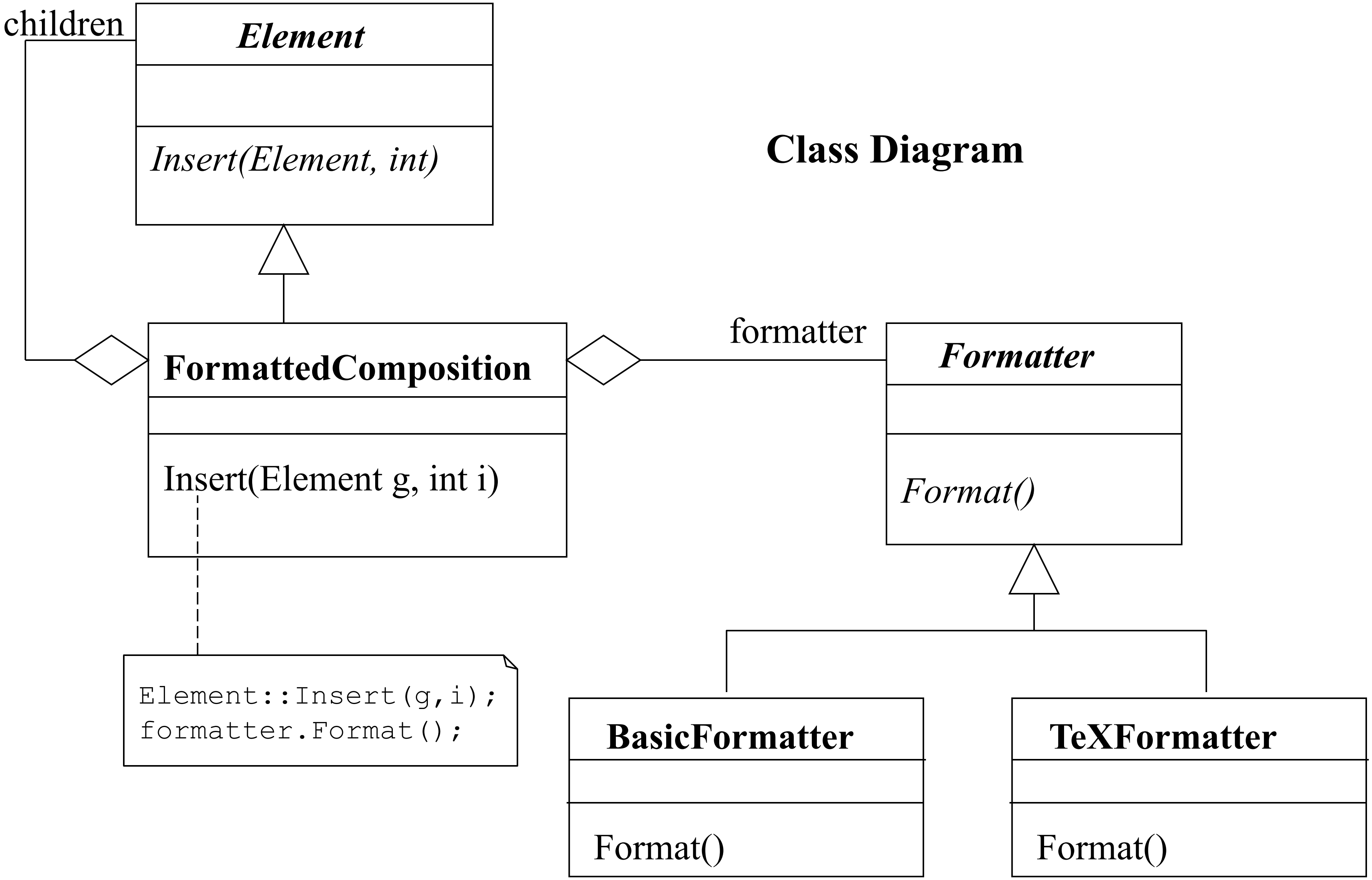
- Formatting is complex
  - There are various algorithms for formatting, not just a single best algorithm

Formatting: Possible Solutions

- A solution
  - Add a format method to each Element class
  - Problem: You have to modify the Element classes when you change the formatting algorithm
  - Problem: It is not easy to add new formatting algorithms

- A better solution
  - Encapsulate the formatting behind an interface
  - Create an abstract *Formatter* class and make different formatting techniques subclasses of this abstract class
  - We create a subclass of Element called FormattedComposition. This class represents composed text elements whose children are formatted by a subclass of *Formatter*

Class Diagram



```
Element::Insert(g,i);
formatter.Format();
```

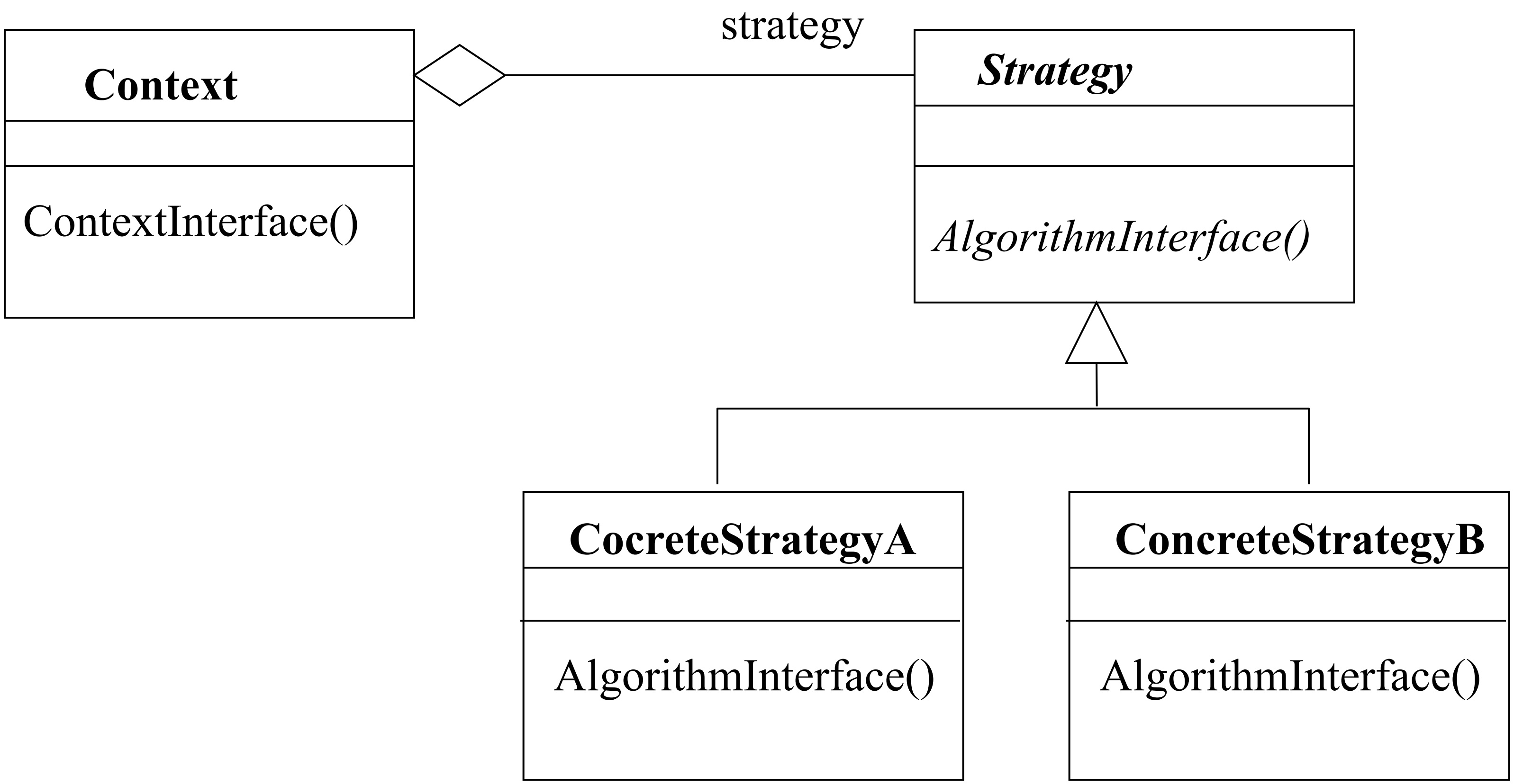
Encapsulating Algorithms: Strategy Pattern

- This design pattern is called the **Strategy** pattern

- You can use strategy pattern when
  - Many different variants of an algorithm is needed

- Strategy pattern
  - Declares an interface common to all supported algorithms as an abstract class
  - Concrete strategy classes are subclasses of the abstract class

Class Diagram for the Strategy Pattern



Back to the Expressions: Problem 2: Supporting Different Print Styles

- We want to be able to print the expressions in different format

- For example, given the expression: – x + 2 * y + 1

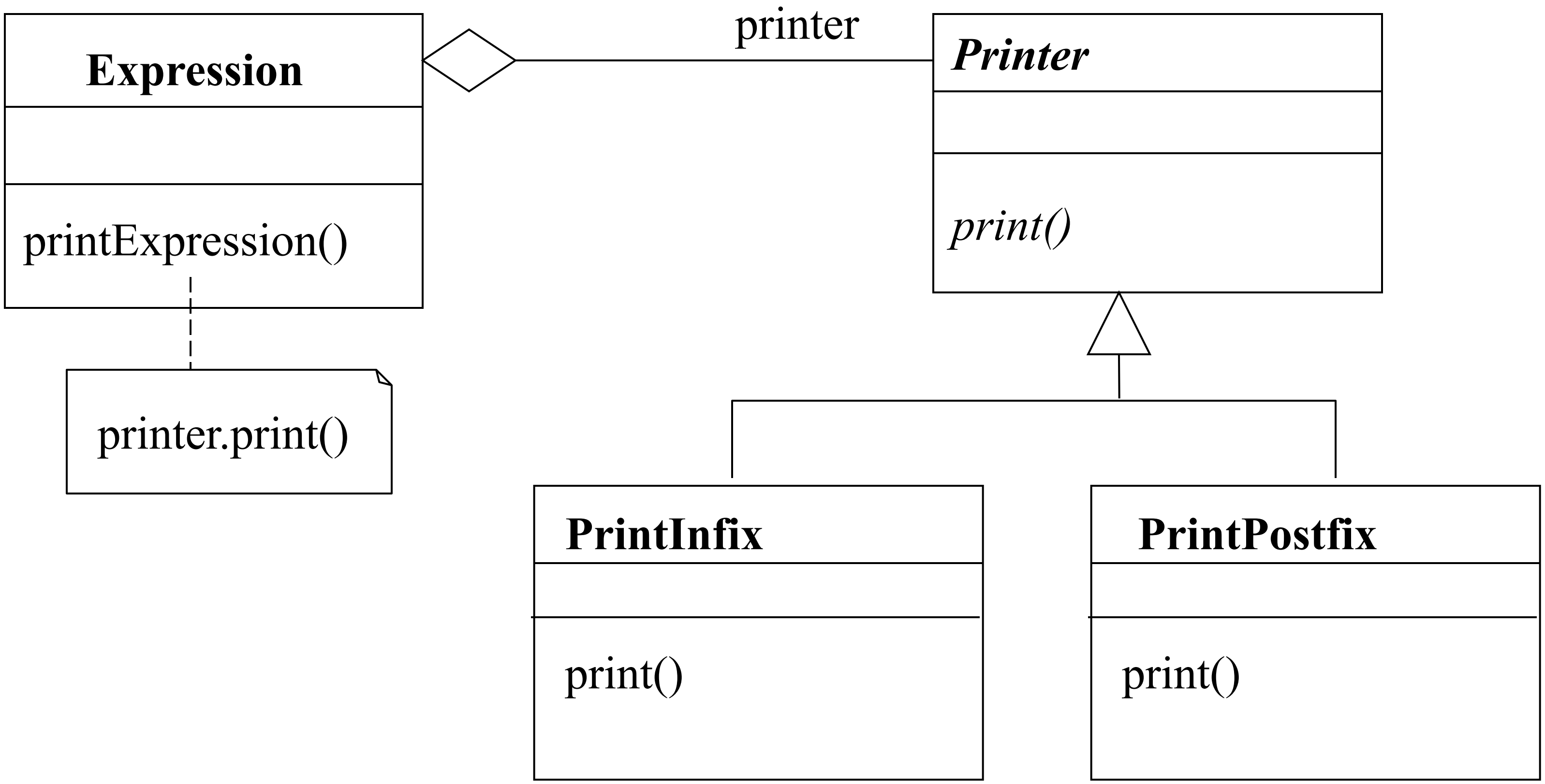
  - We may want to print it infix (fully parenthesized)

    (((– x) + (2 * y)) + 1)

  - or we may want to print it in postfix form:

    x–2 y * + 1 +

Printing Expressions



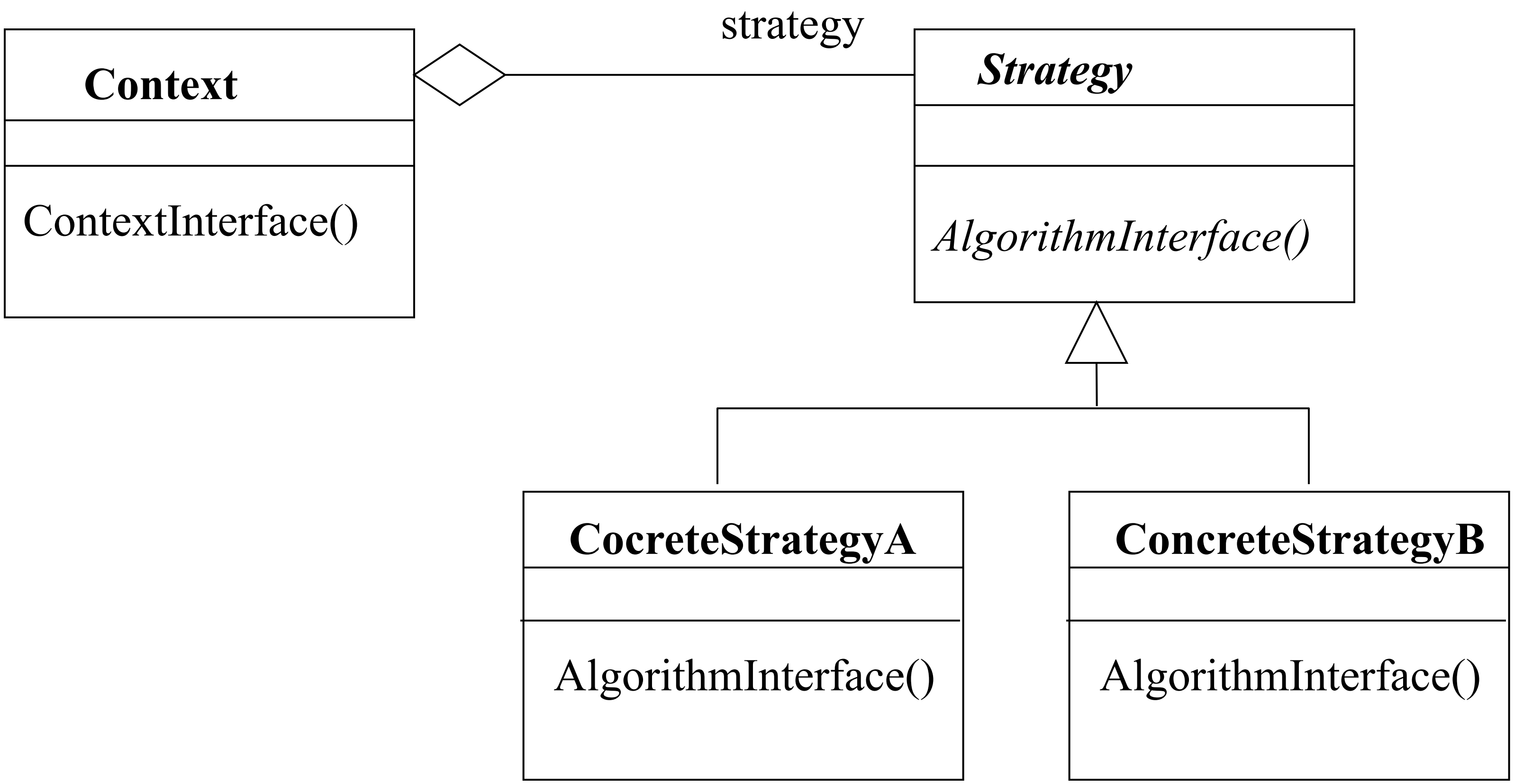
Printing Expressions

- Using strategy pattern enables us to encapsulate the printing algorithm behind a common interface.
- The client code does not have to know what type of printing strategy is being used
    - hence if the printing strategy changes we do not have to change the client code

```
Expression e = new Expression(...);
e.printer = new PrintPostfix(...);


...

e.printExpression();
```

client code

We Used the Strategy Pattern



Text Editor: Problem 3: Embellishing the User Interface

- We want to embellish the user interface by adding
  - Borders
  - Scrollbars, etc.

  when we draw a document

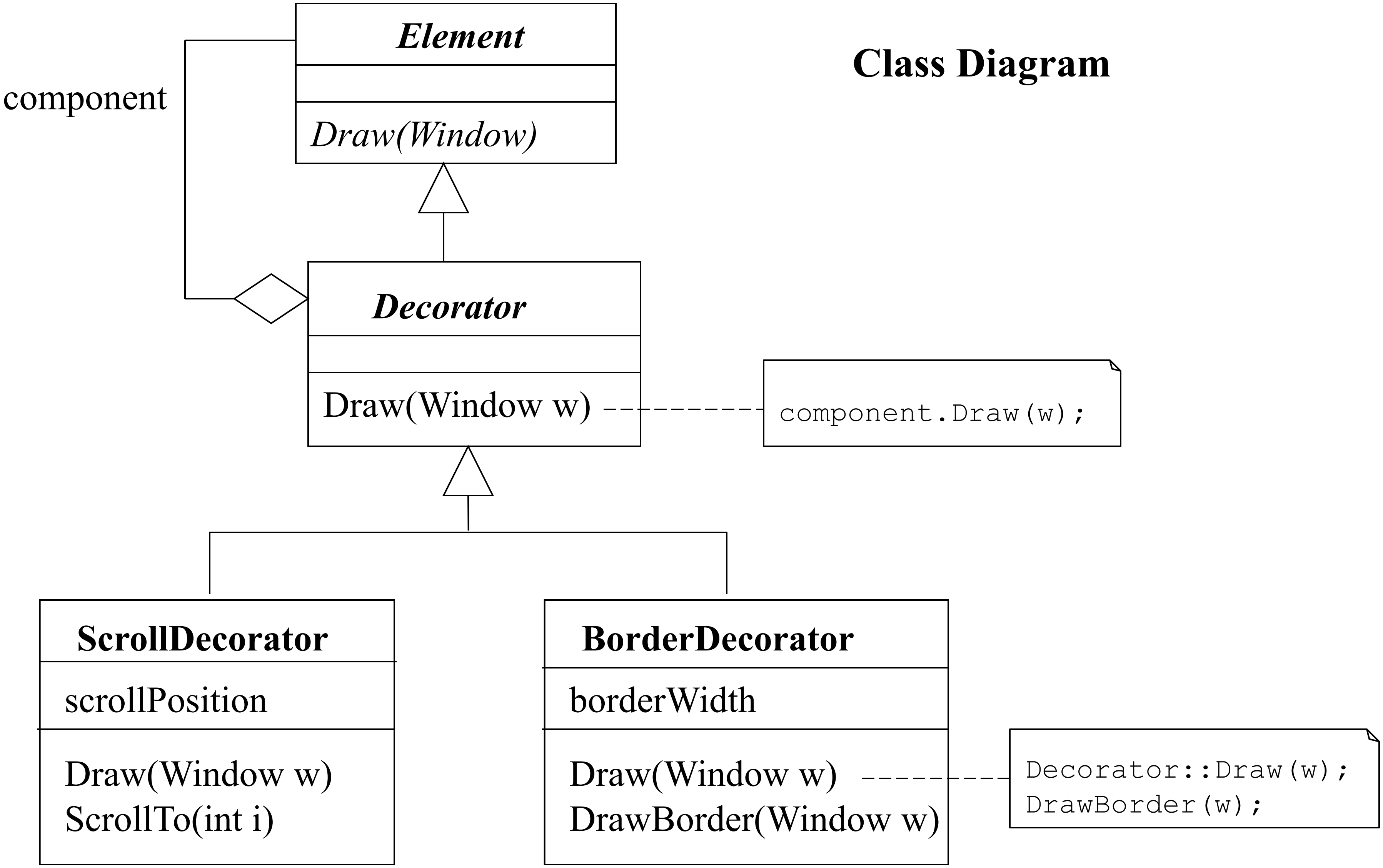
- How do we add this to the design structure?

Embellishing the User Interface

- A solution:
  - Subclass elements of Element
    - BorderedElement, ScrolledElement, BorderedandScrolledElement, etc.
  - Problem: Too many classes, hard to maintain

- A better solution:
  - Create an abstract class say *Decorator* which is a subclass of Element
  - Make all different embellishments subclasses of this new abstract class
  - BorderDecorator, ScrollDecorator, etc.
  - Each Decorator is a wrapper around a single Element

Class Diagram



component

Element

Draw(Window)

Decorator

Draw(Window w)

```
component.Draw(w);
```

ScrollDecorator

scrollPosition

Draw(Window w)
ScrollTo(int i)

BorderDecorator

borderWidth

Draw(Window w)
DrawBorder(Window w)

```
Decorator::Draw(w);
DrawBorder(w);
```

Transparent Enclosure: Decorator Pattern

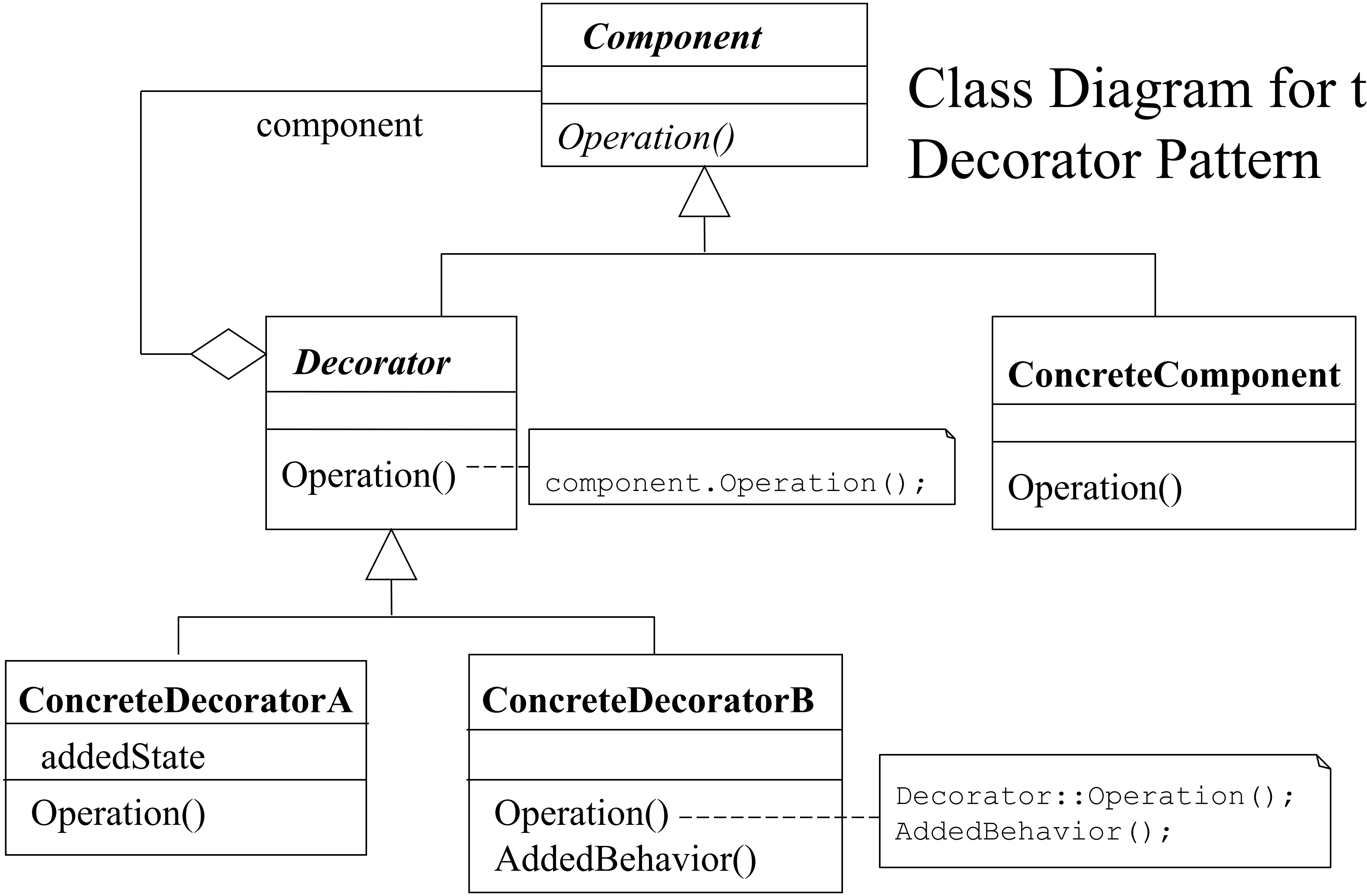
- This is called the **Decorator** pattern

- Note that component of a ScrollDecorator could be an instance of BorderDecorator
  - We can dynamically create all the combinations of decorators

- Decorator pattern is used to add responsibilities to individual objects dynamically and transparently without affecting other objects

- Decorator pattern is useful when extension by subclassing is impractical.
  - Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination

Class Diagram for the Decorator Pattern



Problem 4: Supporting multiple look-and-feel standards

- There are different look-and-feel standards
  - Look-and-feel standards determine the appearance of scrollbars, buttons, menus, etc.

- We want the editor to support multiple look-and-feel standards
  - Motif, Mac, etc.

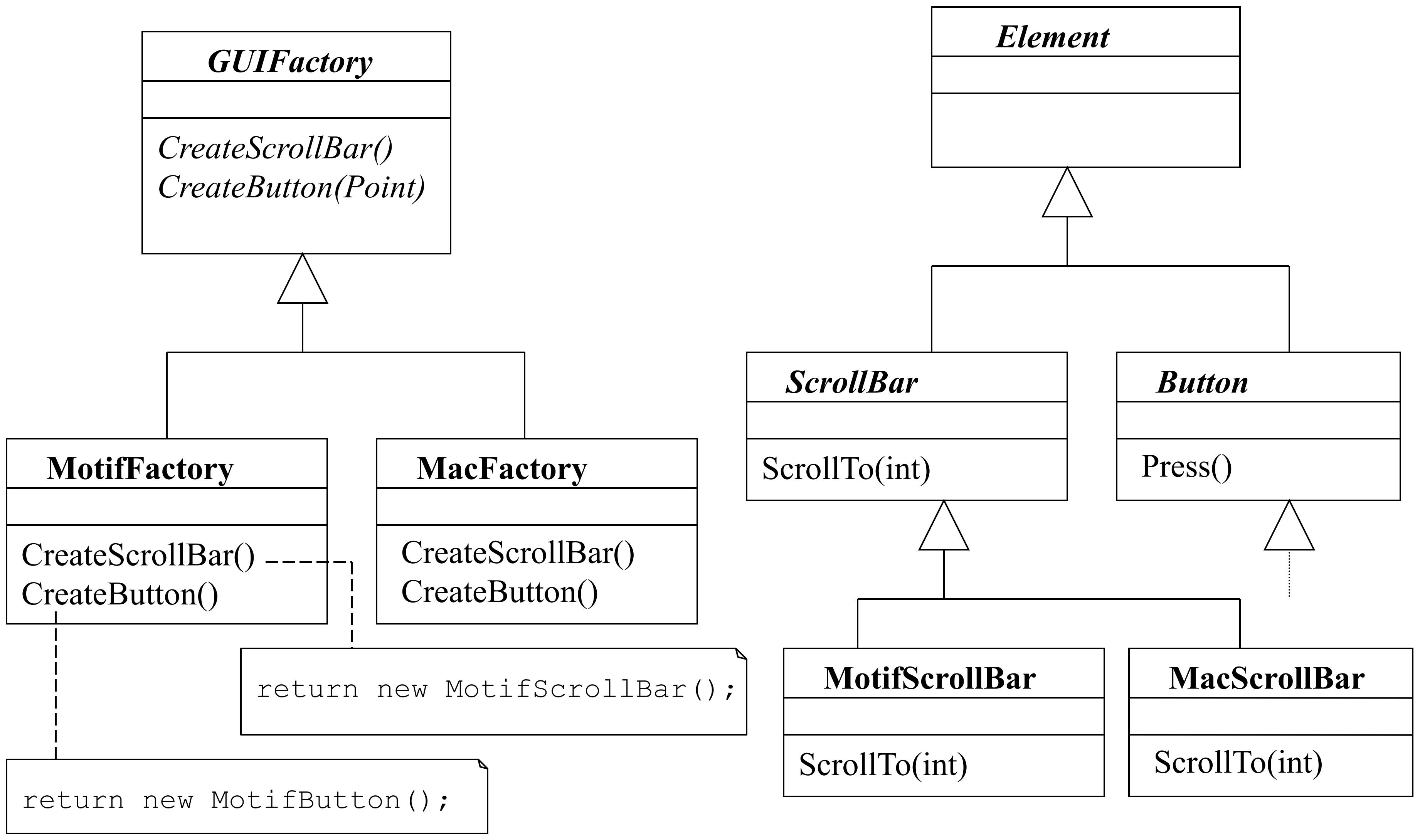
Supporting multiple look-and-feel standards

- A solution:
  - Use a lot of if statements
  - For example to create a ScrollBar:

```
ScrollBar sb;
if (style == MOTIF) then sb = new MotifScrollBar
else if (style == MAC) then sb =  ...
```

- A better solution:
  - Abstract object creation
  - Create a set of abstract subclasses of Element for each object class that will be influenced by the look-and-feel standards. Derive concrete subclasses for them for each look-and-feel standard.
  - Define an abstract *Factory* class. Each concrete subclass of this abstract class generates objects for one look-and-feel standard

Class Diagram



Abstracting Object Creation: Abstract Factory Pattern

- This design pattern is called **Abstract Factory**
  - Following slides show the catalog entry for the Abstract Factory pattern
- You can use it when
  - A system can be configured with one of multiple families of products
  - A family of related product objects is designed to be used together and this constraint is needed to be enforced

- Now, we can generate a scroll bar as follows

```
GUIFactory guiFactory;
if (style == MOTIF) then guiFactory = new MotifFactory;
else if ...

ScrollBar sb;
sb = guiFactory.CreateScrollBar();
```

ABSTRACT FACTORY Object Creational

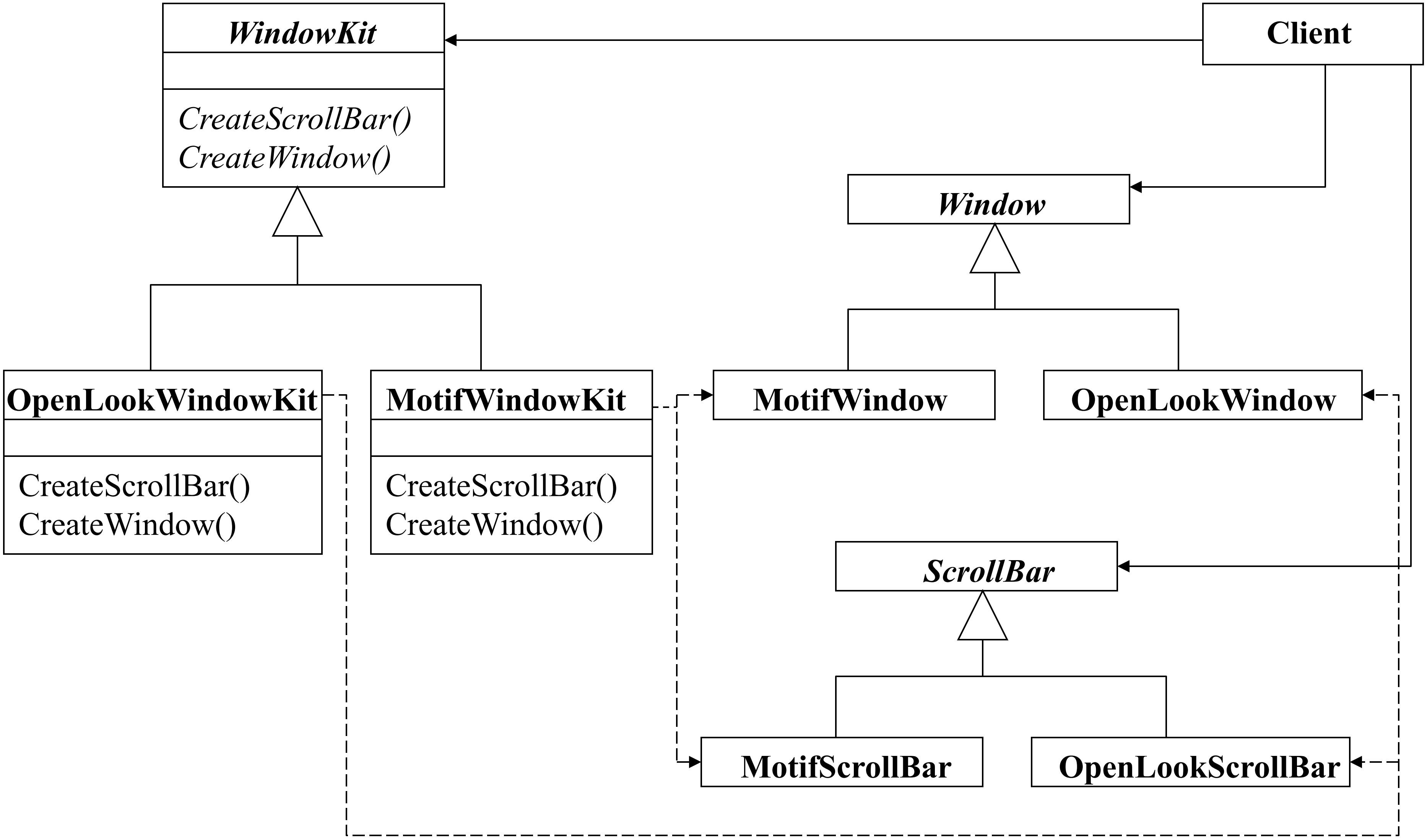
Intent

Provides an interface for creating generic product objects. It removes dependencies on concrete product classes from clients that create product objects.

Motivation

Consider a user interface toolkit that supports multiple standard look-and-feels, for example Motif and Open Look, and provides different scroll bars for each. It is undesirable to hard-code dependencies on either standard into the application, the choice of look-and-feel and hence scroll bar may be deferred until run-time.

When such a system is designed using Abstract Factory pattern an abstract base class WindowKit declares services for creating scroll bars and other controls. For each look-and-feel there is a concrete subclass of WindowKit that defines services to create the appropriate control. Clients access a specific kit through the interface declared by the WindowKit class.


WindowKit
CreateScrollBar()
CreateWindow()
Client
Window
OpenLookWindowKit
CreateScrollBar()
CreateWindow()
MotifWindowKit
CreateScrollBar()
CreateWindow()
MotifWindow
OpenLookWindow
ScrollBar
MotifScrollBar
OpenLookScrollBar

Applicability

When the classes of the product objects are variable, and dependencies on these classes must be removed from a client application.

When variations on the creation, composition, or representation of aggregate objects or subsystems must be removed from a client application. Clients do not explicitly create and configure the aggregate or subsystem but defer this responsibility to an AbstractFactory class.

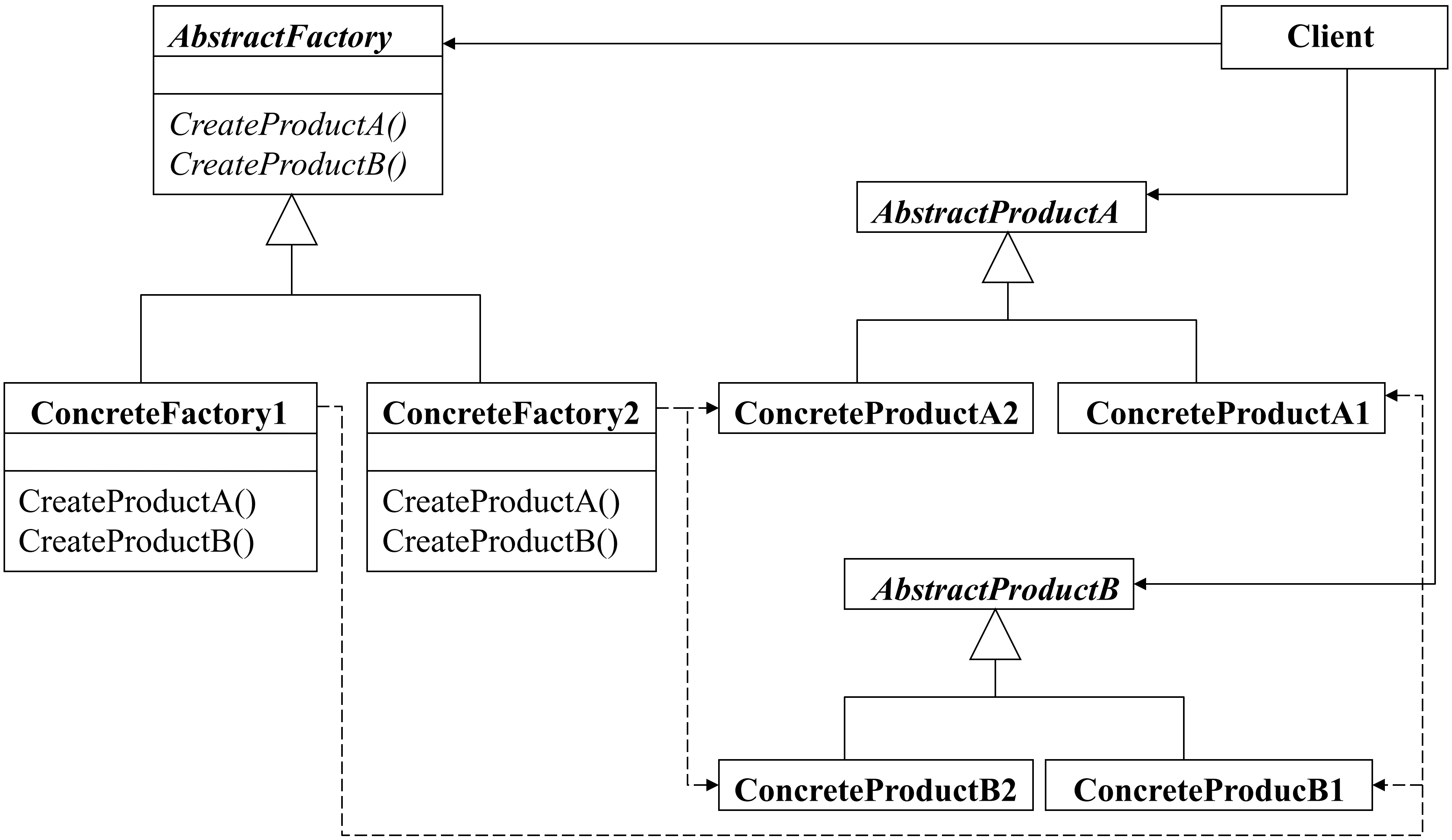
Participants

- **AbstractFactory**
  - declares a generic interface for operations that create generic product objects
- **ConcreteFactory**
  - defines the operation that create specific product objects
- **AbstractProduct**
  - declares a generic interface for product objects
- **ConcreteProduct**
  - defines a product object created by the corresponding concrete factory
  - all product classes must conform to the generic product interface

Collaborations

- Usually a single instance of ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation.
- AbstractFactory defers creation of product objects to its ConcreteFactoy subclasses

Class Diagram for the Abstract Factory Pattern



Consequences

When the classes of the product objects are variable, and dependencies on these classes must be removed from a client application.

AbstractFactory isolates clients from implementation classes, only generic interfaces are visible to clients. Implementation class names do not appear in the client code. Clients can be defined and implemented solely in terms of protocols instead of classes.

Examples

InterViews, ET++.

Implementation

AbstractFactory defines a different operation for each kind of product it can produce. The kinds of products are encoded in the operation signatures. Adding a new kind of product requires changing the AbstractFactory interface. A more flexible but less safe design is to add a parameter to operations that create objects which specifies what kind of object will be created. In this approach a single "Create" operation will be enough with a parameter defining the type of the object. However, now the client will use a generic base class to access the products and cannot make safe assumptions about the class of a product.

See Also Factory Method pattern

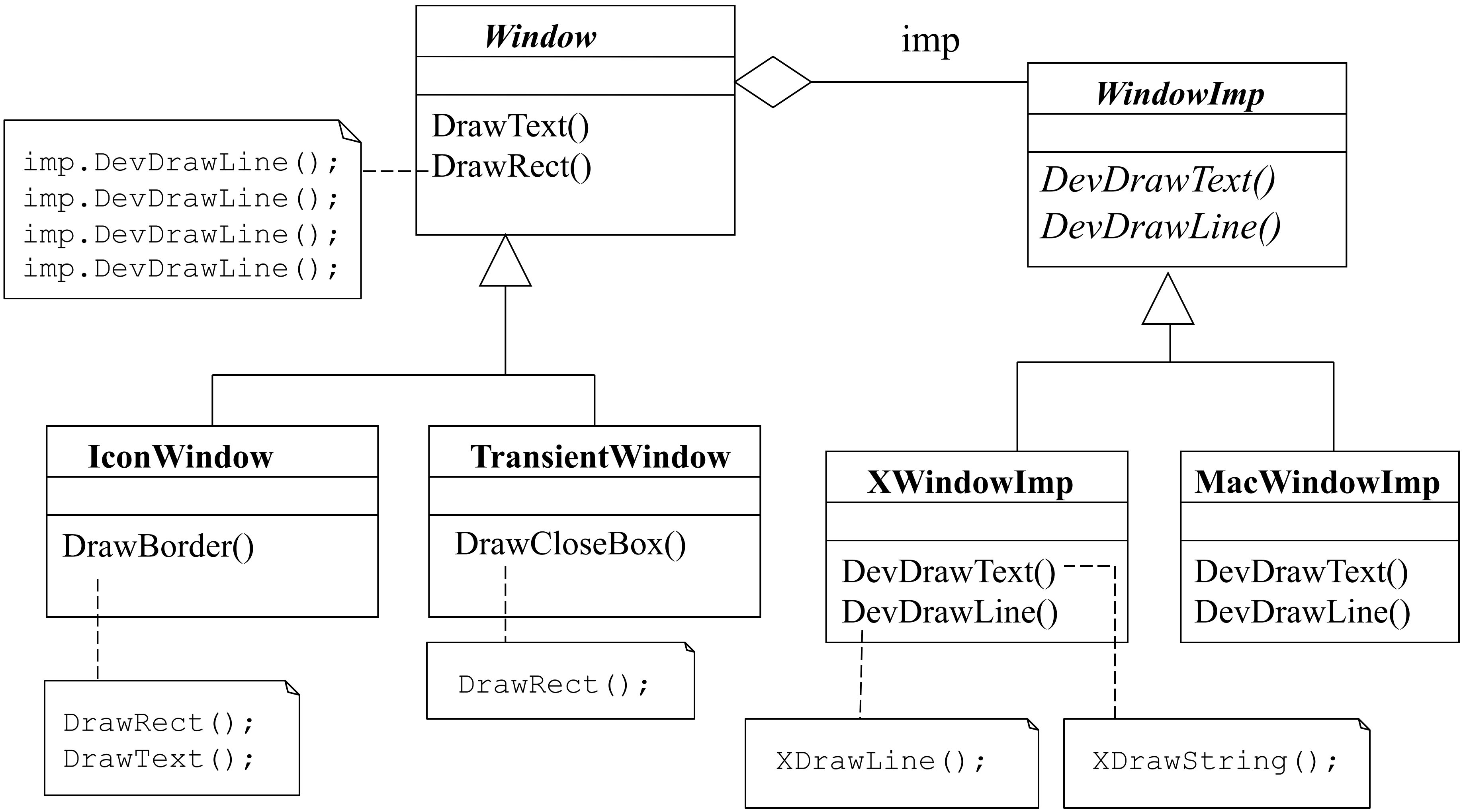
Back to the Case Study; Problem 5: Supporting Multiple Window Systems

- We want to support multiple window systems
- However there is a wide variation in standards
  - There are different models for window operations such as
    - resizing,
    - drawing,
    - raising, etc.
  - Different window systems provide different functionality
- Since different window systems may not be compatible we cannot use the Abstract Factory Pattern
  - Abstract Factory Pattern assumes that the class hierarchy is same for all the variations

Supporting Multiple Window Systems

- A solution
  - We can take an intersection of all the functionality
  - A feature is allowed in the window model only if it is in every window system
  - Problem: Intersection functionality may be too restrictive
- Another solution
  - Create an abstract window hierarchy
    - All the functionality required by the application is represented
  - Create a parallel hierarchy for window systems
    - All the functionality required by the application is represented
    - Requires methods to be written for the systems missing some functionality

Class Diagram



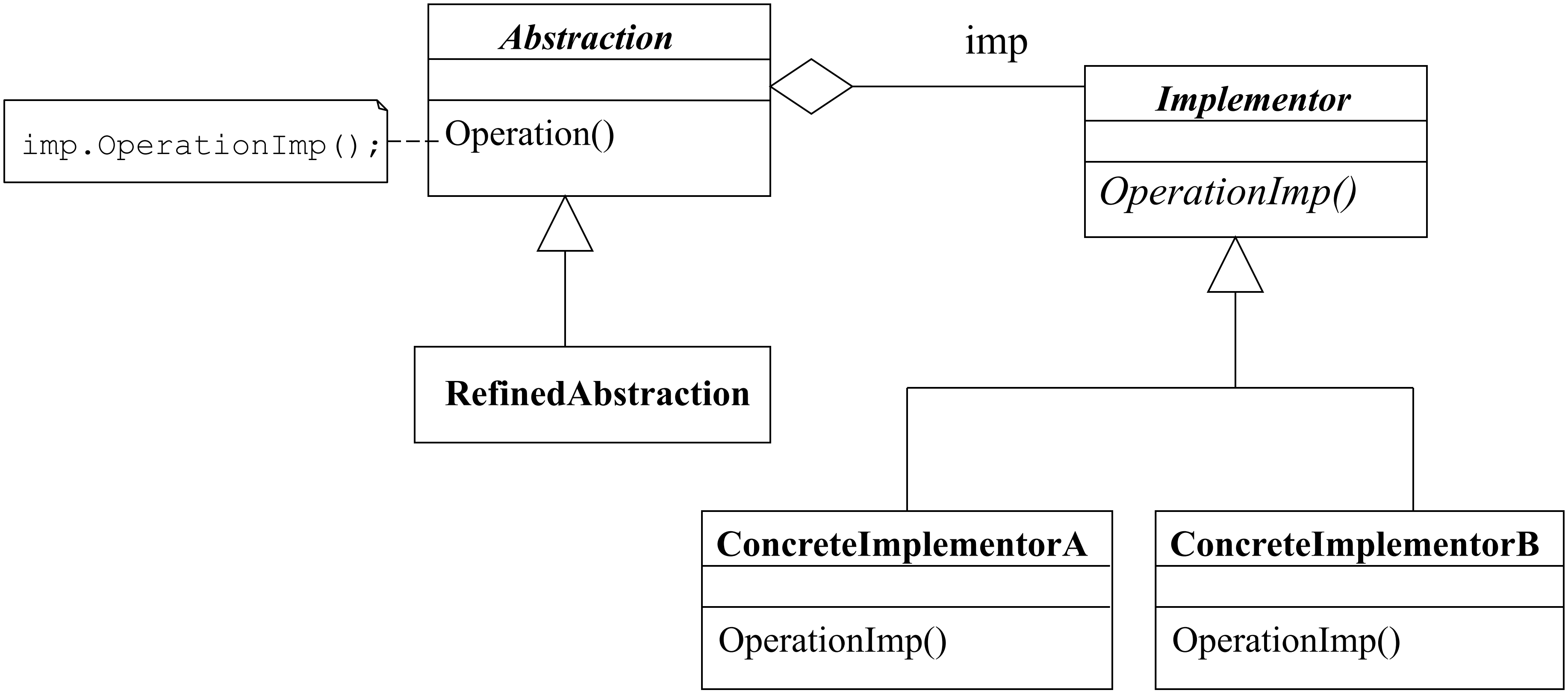
Encapsulating Implementation Dependencies: Bridge Pattern

- This is called the **Bridge** pattern

- There are two hierarchies
  - Abstraction: This is the abstract hierarchy showing a logical view
  - Implementation: This is the implementation hierarchy implementing the logical view

- Both hierarchies are extensible independently
- Implementation is hidden from the abstract hierarchy

Class Diagram for the Bridge Pattern



Encapsulating the Concept that Varies

- Note that there is a common theme in some of the patterns we are discussing (strategy, decorator, bridge patterns)
  - We are encapsulating the concept that varies
  - This varying part is accessed by the rest of the system through an abstract interface
- We are using two important software engineering principles: ***Anticipation of Change*** and ***Information Hiding***
  - We try to make it easy to change the parts we suspect will have a lot of variations
  - We try to isolate the rest of the system from the effects of these changes
  - We achieve this by hiding the part that varies behind an abstract interface

Problem 5: User Operations

- User has a set of operations such as
  - creating a new document
  - opening, saving, printing an existing document
  - changing the the font and style of the selected text
  - etc.
- There should be a uniform way to deal with user operations
  - and possibly undo them.

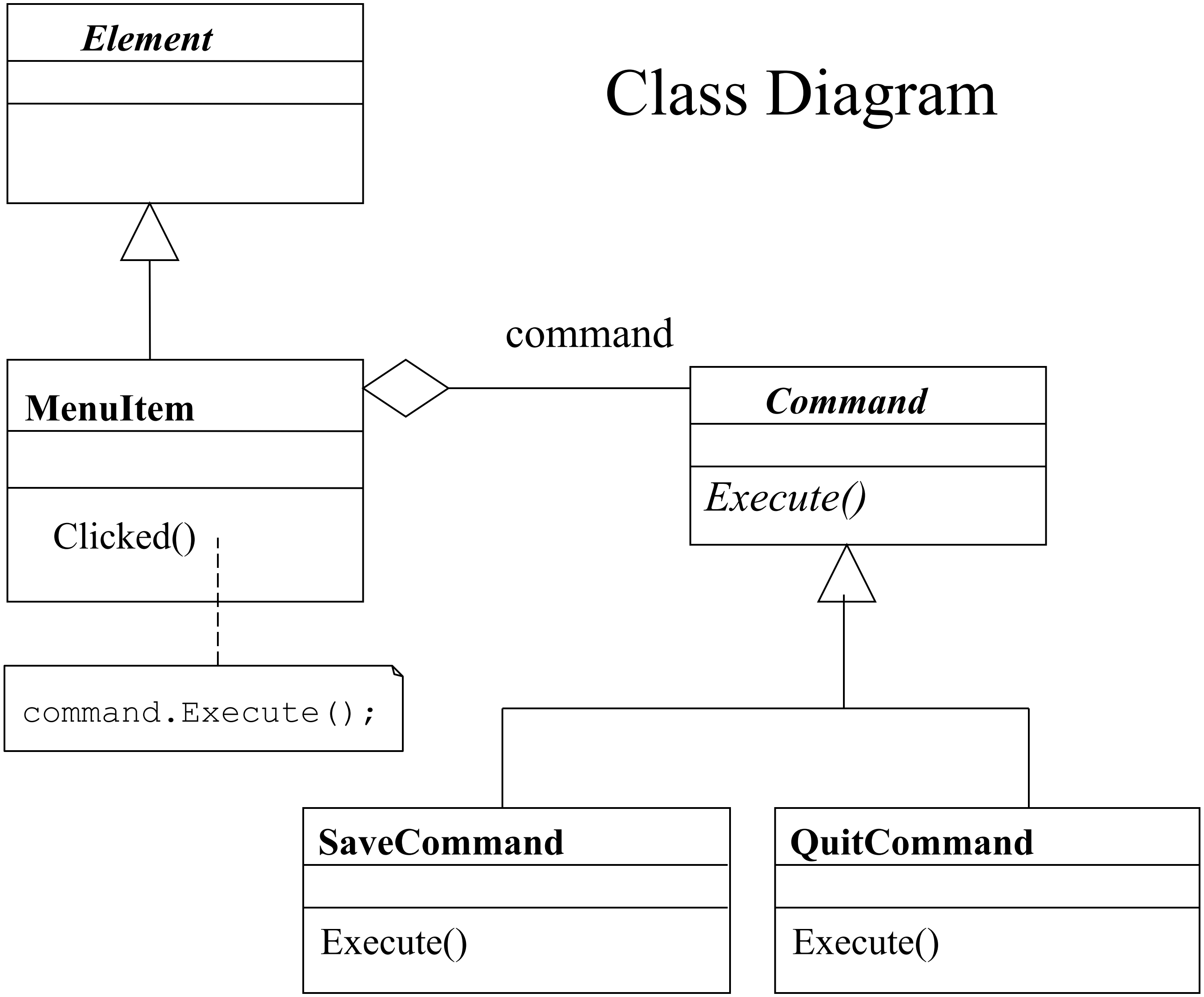
- How do represent user commands?

User Commands

- Define an abstract class for user operations
  - This class presents an interface common to all the operations
    - such as undo, redo

- Each operation is derived as a subclass of the abstract command class

Class Diagram



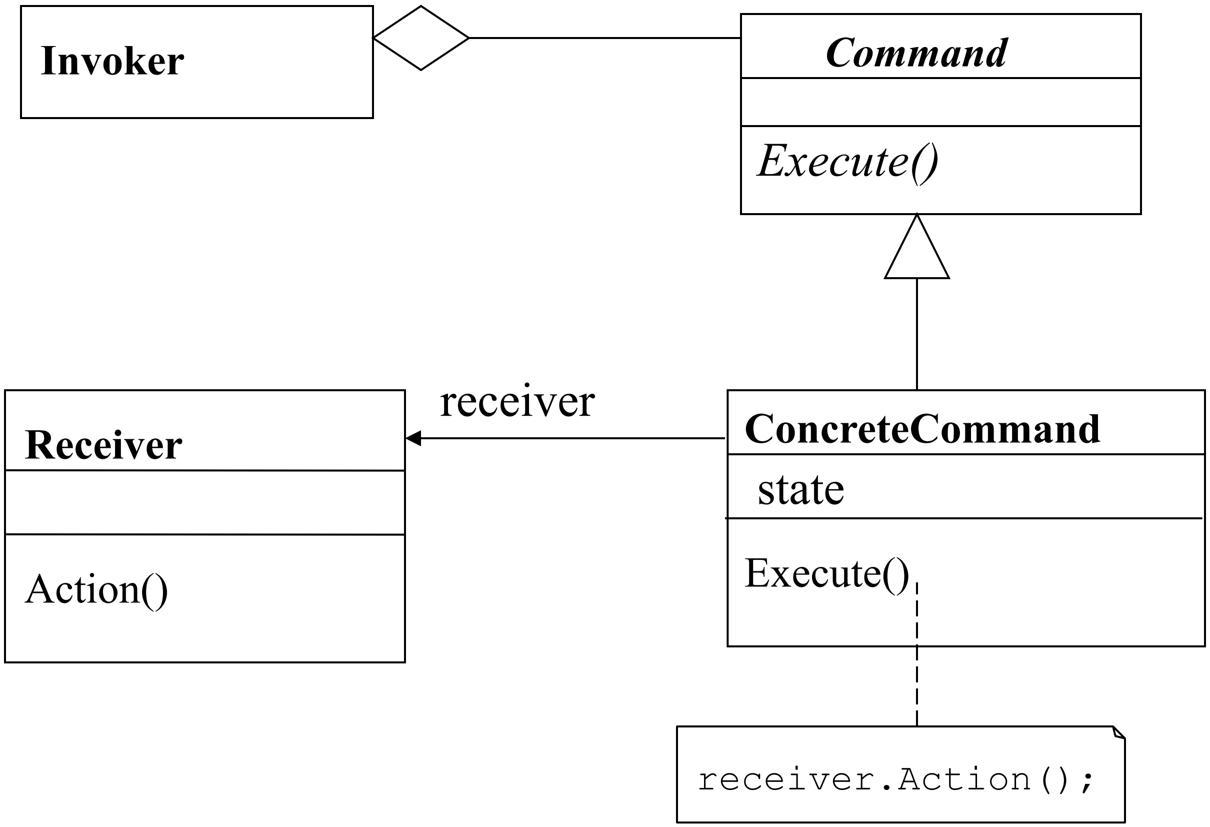
User Commands

- User may want to undo some commands
  - Add an abstract Unexecute operation to the command interface

- Command history
  - A command history is a list of commands that have been executed
  - Using a command history one can do arbitrary undo and redo operations

- Command pattern
  - Decouples command requester and requestee
  - Enables a uniform treatment of commands
    - command history
    - undo/redo

Encapsulating a Request: Command Pattern

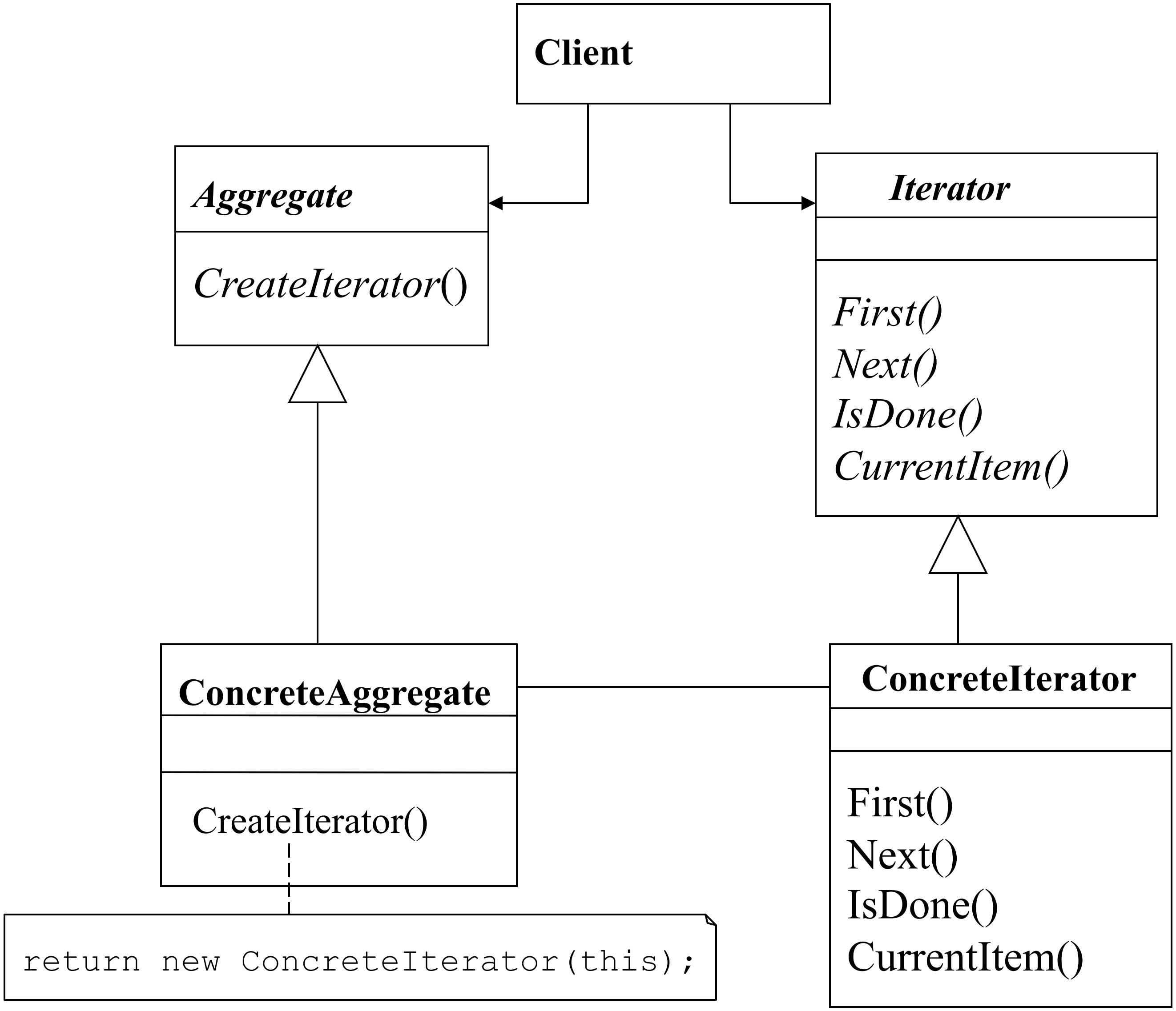


Problem 6: Spell Checking

- How to traverse the document for operations such as spell-checking
  - We need to traverse every Element in the document
  - There may be other analyses which require traversal

- A solution: Iterators
  - An Iterator hides the structure of the container from clients who want to iterate over the structure
  - An Iterator has a method for
    - Getting the first element
    - Getting the next element
    - Testing for termination

Encapsulating Access and Traversal: Iterator Pattern



Spell Checking Using Iterator Pattern

```
Element g;
Iterator i = g.CreateIterator();
for ( i = i.first() ; !(i.isdone()); i = i.next())
   {
     // spell check Element i.current()
   }
```

- Note that we can easily implement different traversal strategies (such as pre-order traversal, post-order traversal, etc.) by writing new concrete iterator classes.

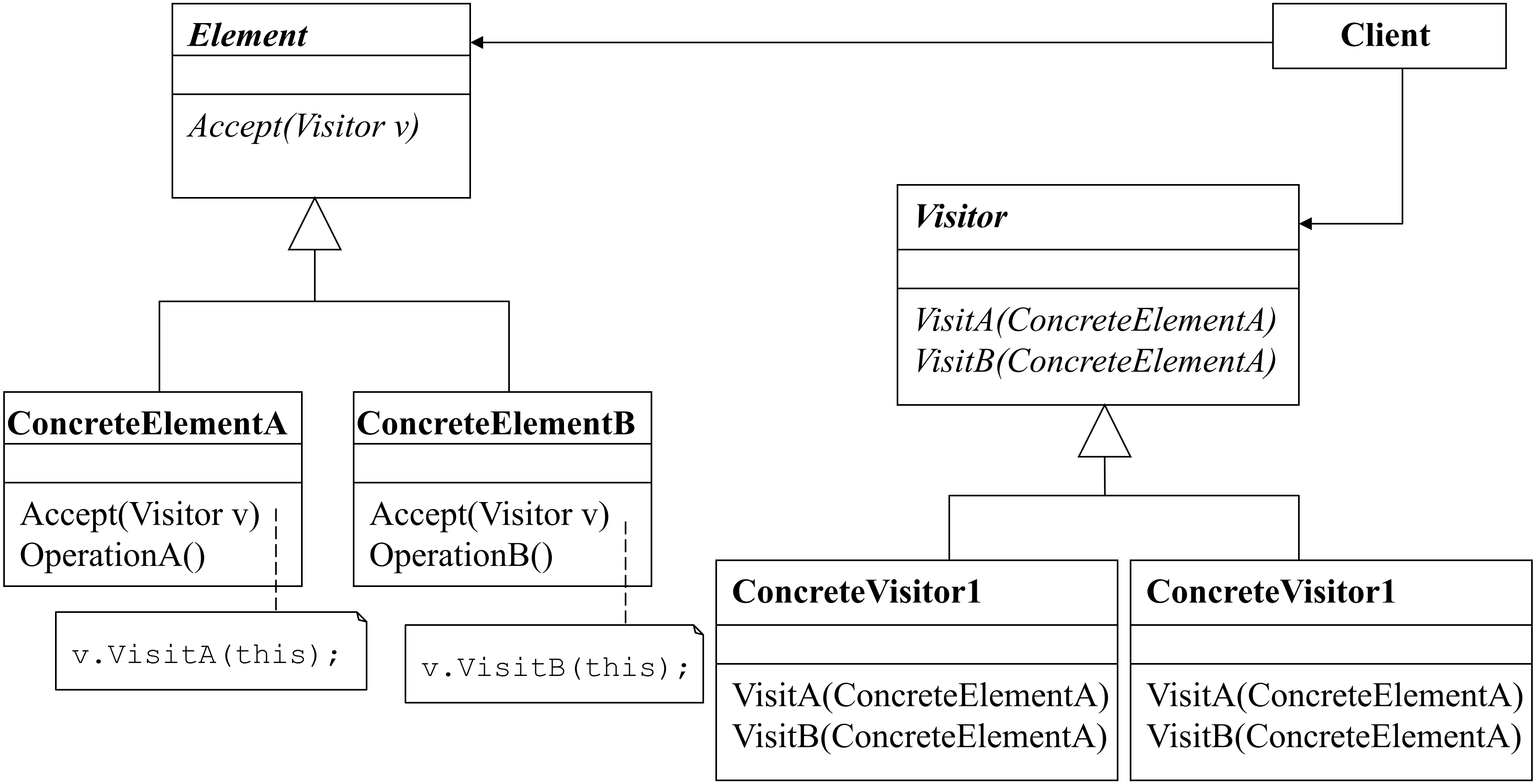
Problem 6: Generalizing the Analysis

- Iterator pattern provides traversal of containers

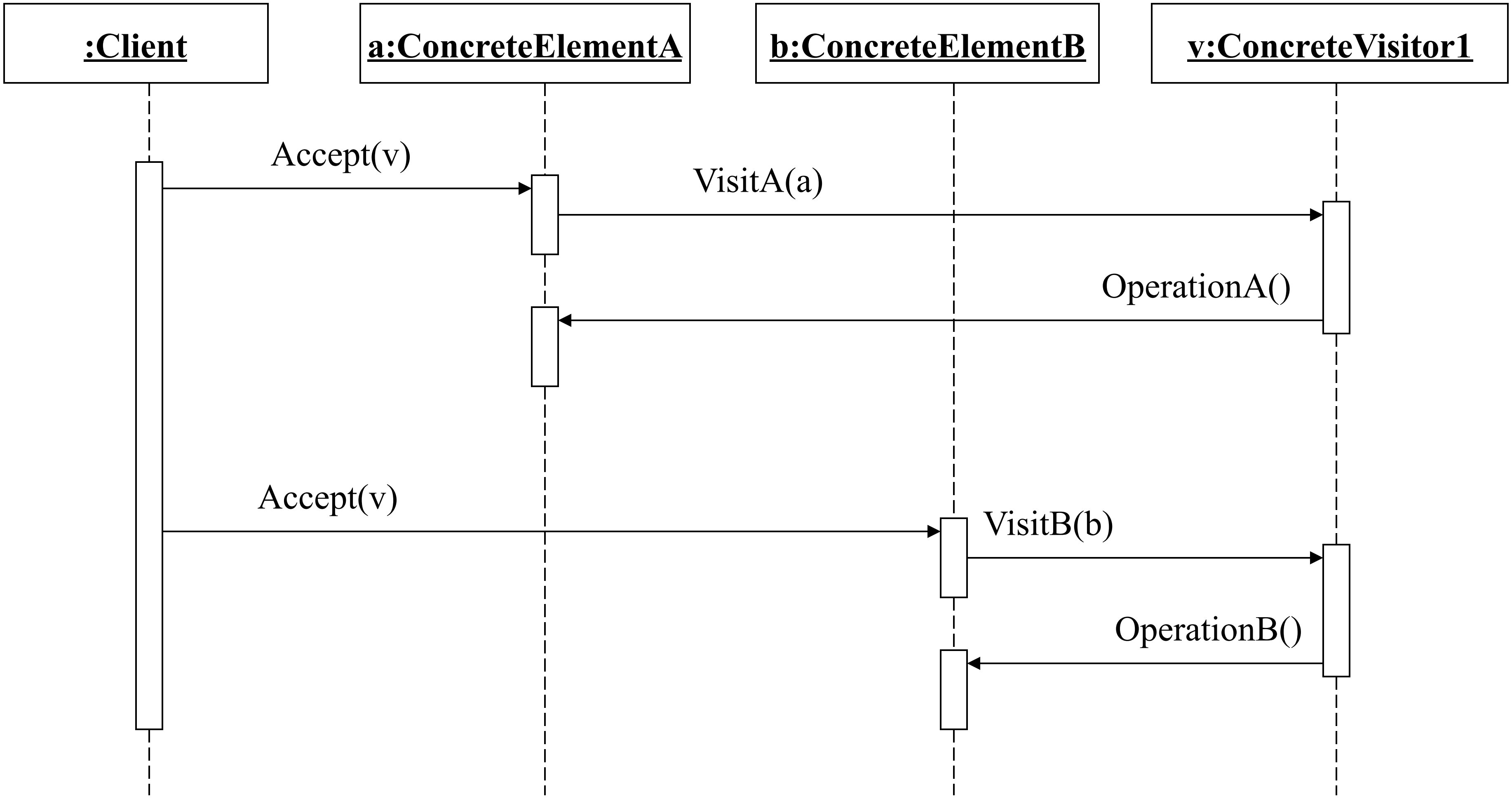
- We may also want to encapsulate the traversal with the analysis as follows:
  - Visit each item
  - Perform a type-specific action on each item
    - For example, spell check

- We can abstract recursive traversal in a class
  - Create a visit operation for each element that performs the analysis
  - The visitor can call the operations of the element while performing the analysis

Encapsulating Analysis: Visitor Pattern



Sequence Diagram for the Visitor Pattern

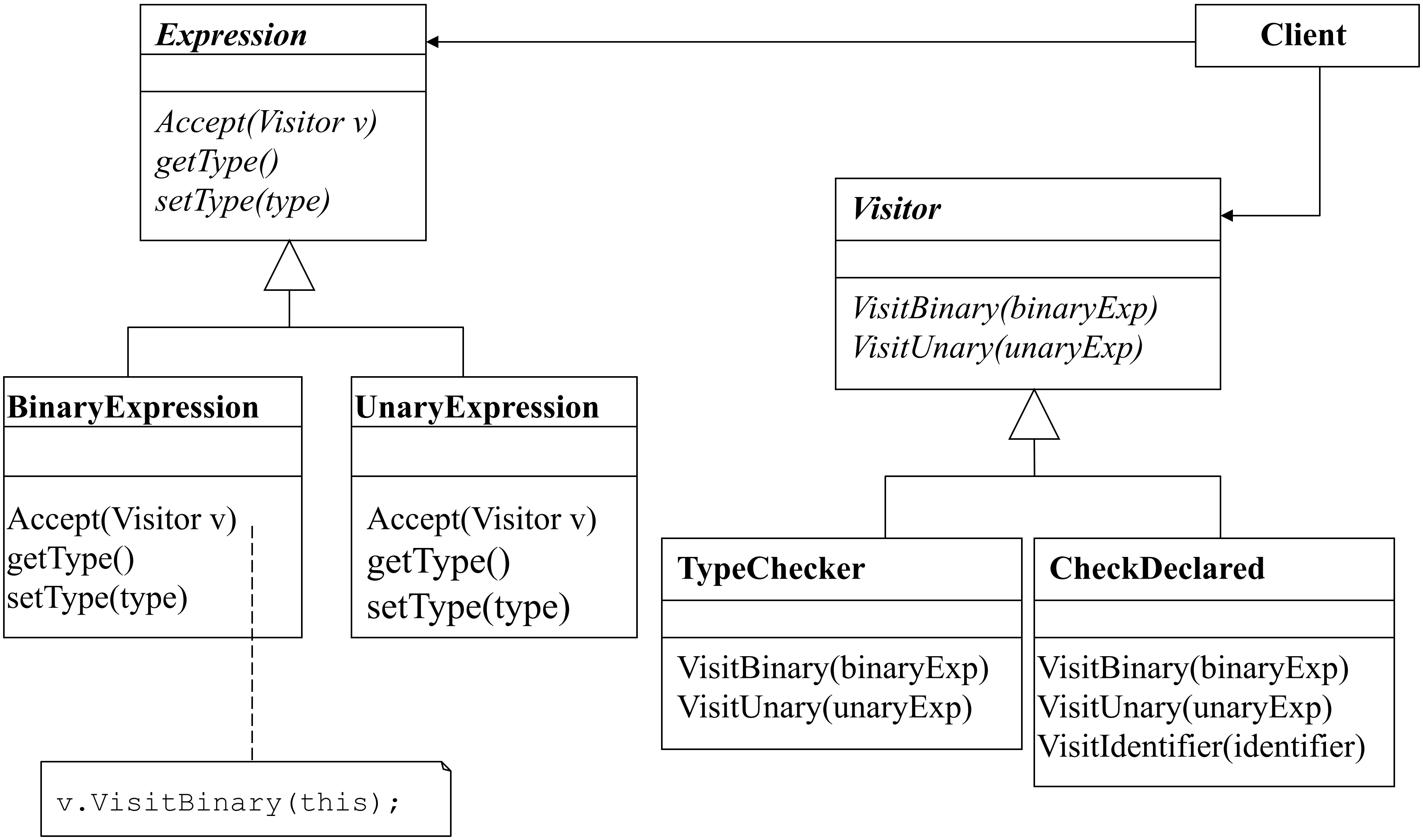


Back to Expressions: Problem 3: Checking Expressions

- We need to do type checking on expressions
  - For example, arguments of an addition operation should be integers; types of left and right children of an equality expression should match, etc.

- We may need to add other checks later on
  - For example, are all the identifiers used in the expression have been declared

Checking Expressions



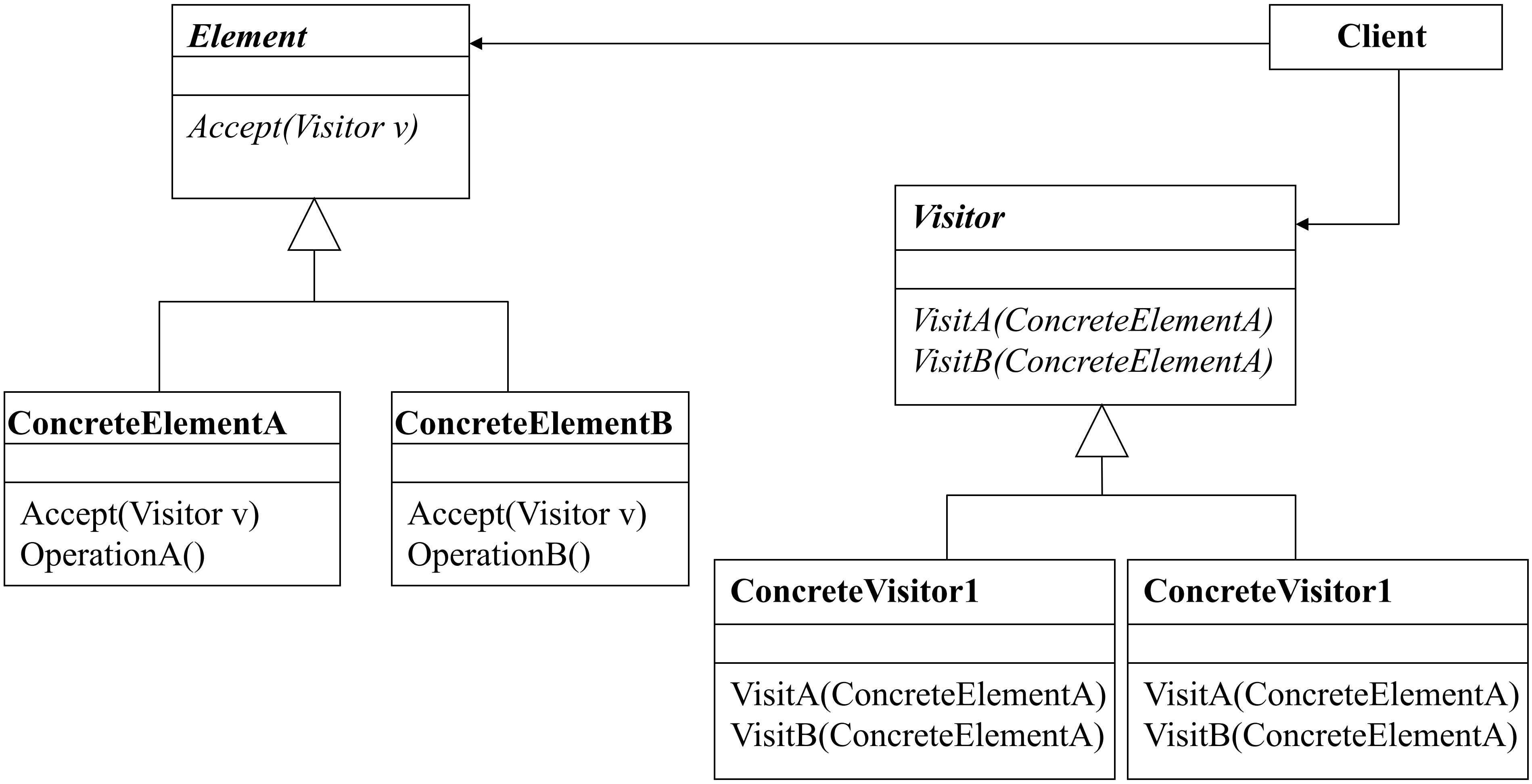
```
v.VisitBinary(this);
```

Checking Expressions

- The visitBinary method first calls the Accept methods of the left and right children and passes itself (type-checker) as the argument
  - This will type check all subexpressions recursively
- If the children have type errors or if the types of children do not match it sets its own type to type error

```
VisitBinary(binaryExp e) {
  e.left.Accept(this);
  e.right.Accept(this);
  if (e.left.getType() == type_error
       || e.right.getType() == type_error
       || e.left.getType() != e.right.getType())
     e.setType(type_error);
  else
     e.setType(...); // the argument here will depend
                      // on the type of the operator
}
```

We Used the Visitor Pattern



Benefits of Design Patterns

- Design patterns
  - provide a common vocabulary for designers to communicate, document and explore design alternatives
  - reduce system complexity by naming and defining abstractions that are above classes and instances
  - constitute a reusable base of experience for building software
  - act a building blocks for constructing more complex designs
  - reduce the learning time for a class library
  - provide a target for reorganization and refactoring of class hierarchies

Design Patterns

- A design pattern consists of three essential parts
  1. An abstract description of class or object collaboration and its structure
  2. The issue addressed by the design pattern, the circumstances in which it is applicable
  3. Consequences of using the design pattern

- Design patterns are micro-architectures
  - They can have several different realizations
  - They do not define a complete application or a library
  - To be useful they should be applicable to more than one problem

Categorizing Design Patterns

- Two orthogonal criteria can be used to categorize patterns
  - Jurisdiction
  - Characterization
- Jurisdiction
  - Class jurisdiction
    - Relationships between base classes and their subclasses, static semantics
  - Object jurisdiction
    - Relationships between peer objects
  - Compound jurisdiction
    - Deals with recursive object structures

Characterizing Design Patterns

- Characterization
  - Creational patterns
    - Deal with initializing and configuring classes or objects
  - Structural
    - Deal with composition of classes or objects, decoupling interface and implementation of classes or objects
  - Behavioral
    - Characterize the ways in which classes or objects interact and distribute responsibility, deal with dynamic interactions among classes or objects

Characterization

Jurisdiction

	Creational	**Structural**	**Behavioral**

Class	Factory Method	Adapter(class)
		Bridge(class)
			Template Method

Object	Abstract Factory Prototype Singleton	Adapter(object) Bridge(object) Flyweight Facade Proxy	Chain of Responsibility Command Iterator(object) Mediator Momento Observer State Strategy
Compound	Builder	Composite Decorator	Interpreter Iterator(compound) Walker

Creational Patterns

- Factory Method
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Abstract Factory
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- Builder
  - Separate the construction of a complex object from its representation so that the same construction can create different representations.
- Prototype
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Singleton
  - Ensure a class only has one instance, and provide a global point of access to it.

Structural Patterns

- Adapter
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.
- Bridge
  - Decouple an abstraction from its implementation so that the two can vary independently.
- Composite
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Decorator (aka Wrapper)
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality

Structural Patterns

- Facade (aka Glue)
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Flyweight
  - Use sharing to support large numbers of fine-grained object efficiently.
- Proxy
  - Provide a surrogate or placeholder for another object to control access to it

Behavioral Patterns

- Chain of Responsibility
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Command
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Interpreter
  - Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language

Behavioral Patterns

- Iterator
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Mediator
  - Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently
- Memento
  - Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later
- Observer
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updates accordingly.

Behavioral Patterns

- State
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Strategy
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Template Method
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure
- Visitor
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Case Study: Representing Expressions

- Assume that we would like to implement a set of classes for representing and manipulating expressions

- These classes can be used in a compiler implementation

- We need to store the expressions in some form (i.e., abstract syntax tree)

- We need to perform operations on the expressions such as
  - printing
  - type checking

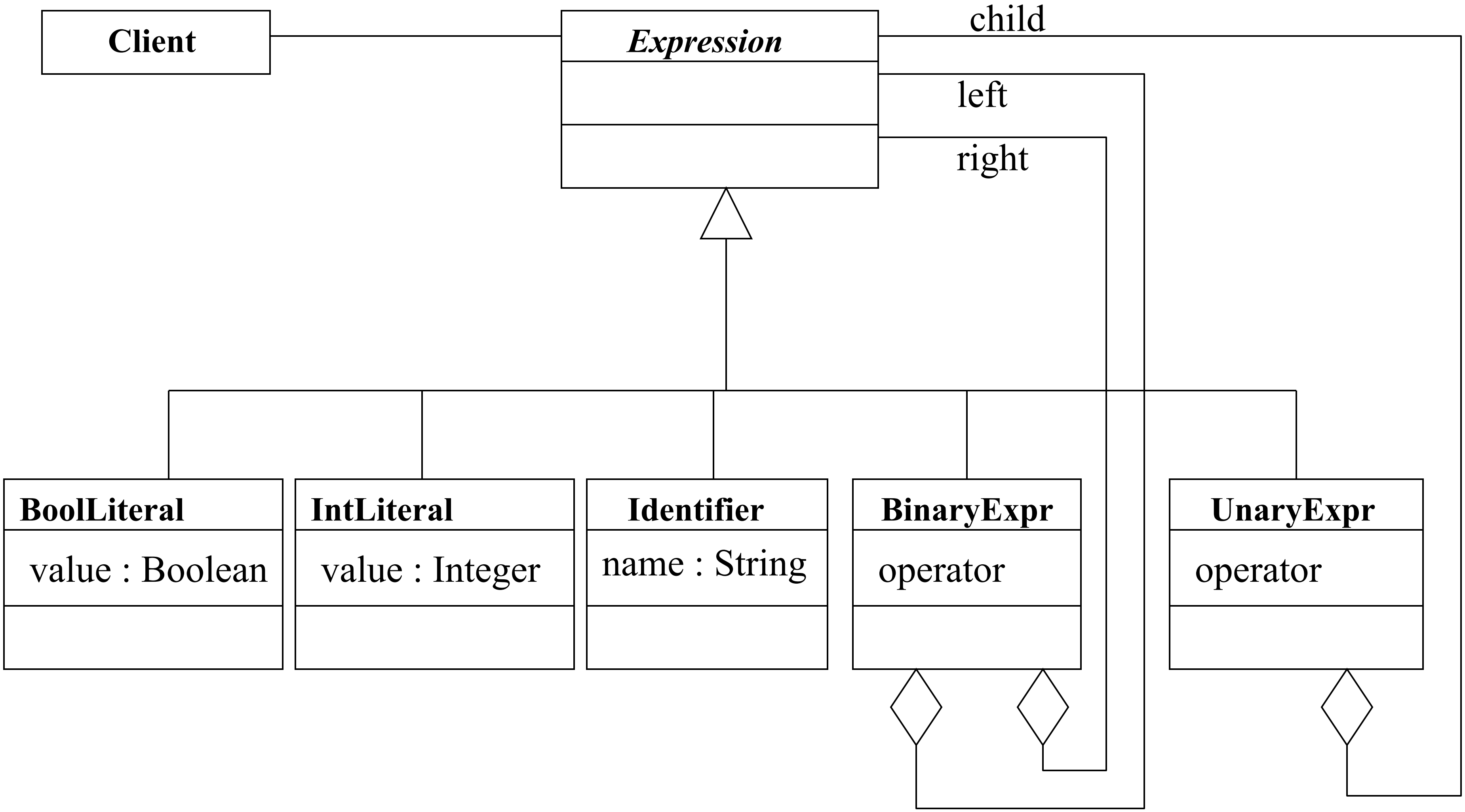
Problem 1: How To Represent Expressions?

- There are different types of expressions such as:
  - boolean literal, integer literal, identifier, binary expression, unary expression, etc.

- Different types of expressions have different attributes so it would make sense to have a different class for each expression type

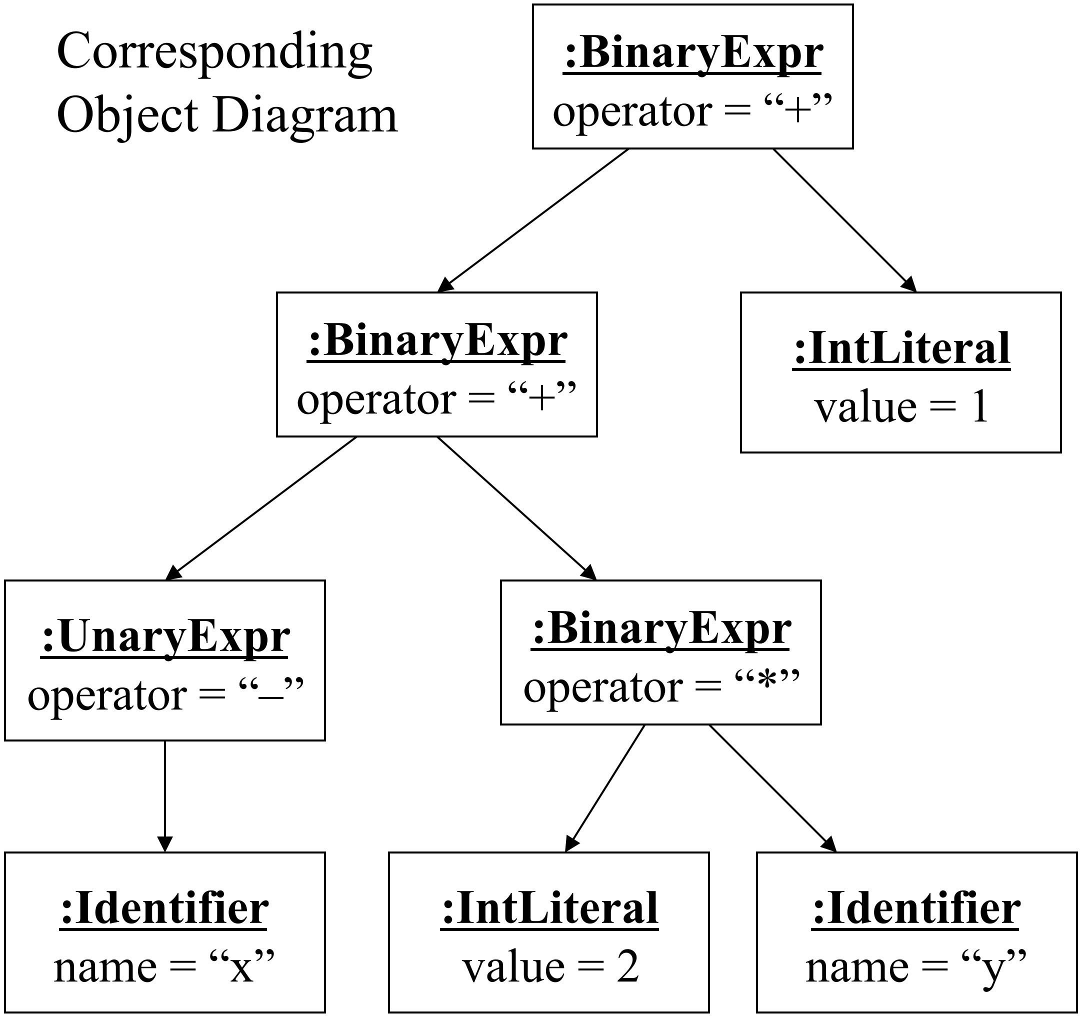
- However, we should be able to treat expressions uniformly
  - For example, children of binary expressions or unary expressions could be any type of expression

Class Diagram for Expressions



An Expression:
– x + 2 * y + 1

Corresponding Object Diagram



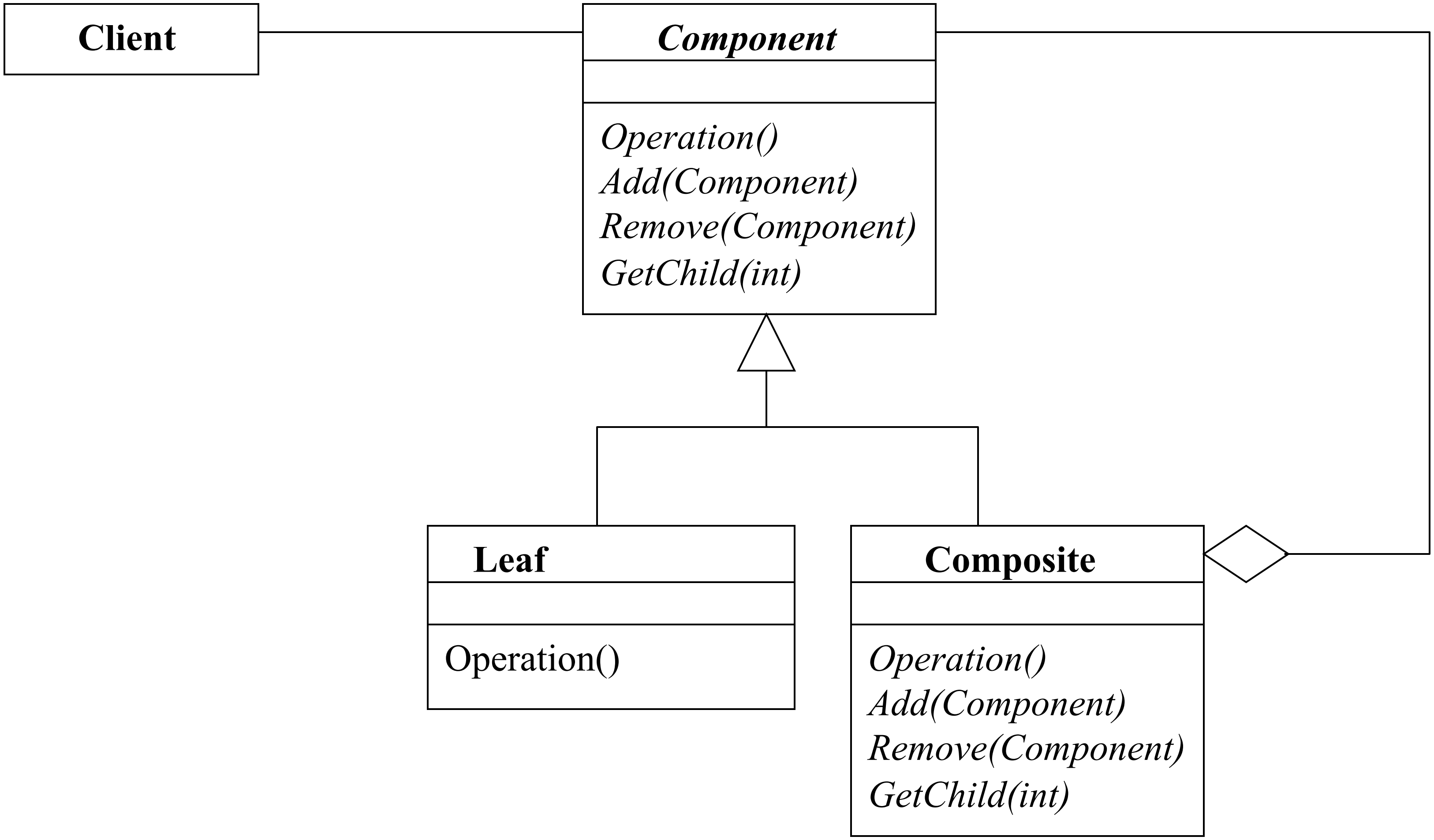
Using the Composite Pattern

- Using composite pattern enables us to treat all the expressions uniformly using the Expression interface

```
Expression e1 = new Identifier(...);
Expression e2 = new BoolLiteral(...);
Expression e3 = new BinaryExpr(e1, e2, ...);

...

printer.printSomeEpxression(e3);

...

public void printSomeExpression(Expression e) {
  e.print();
}
```

client code may not need to know what type of expression e is

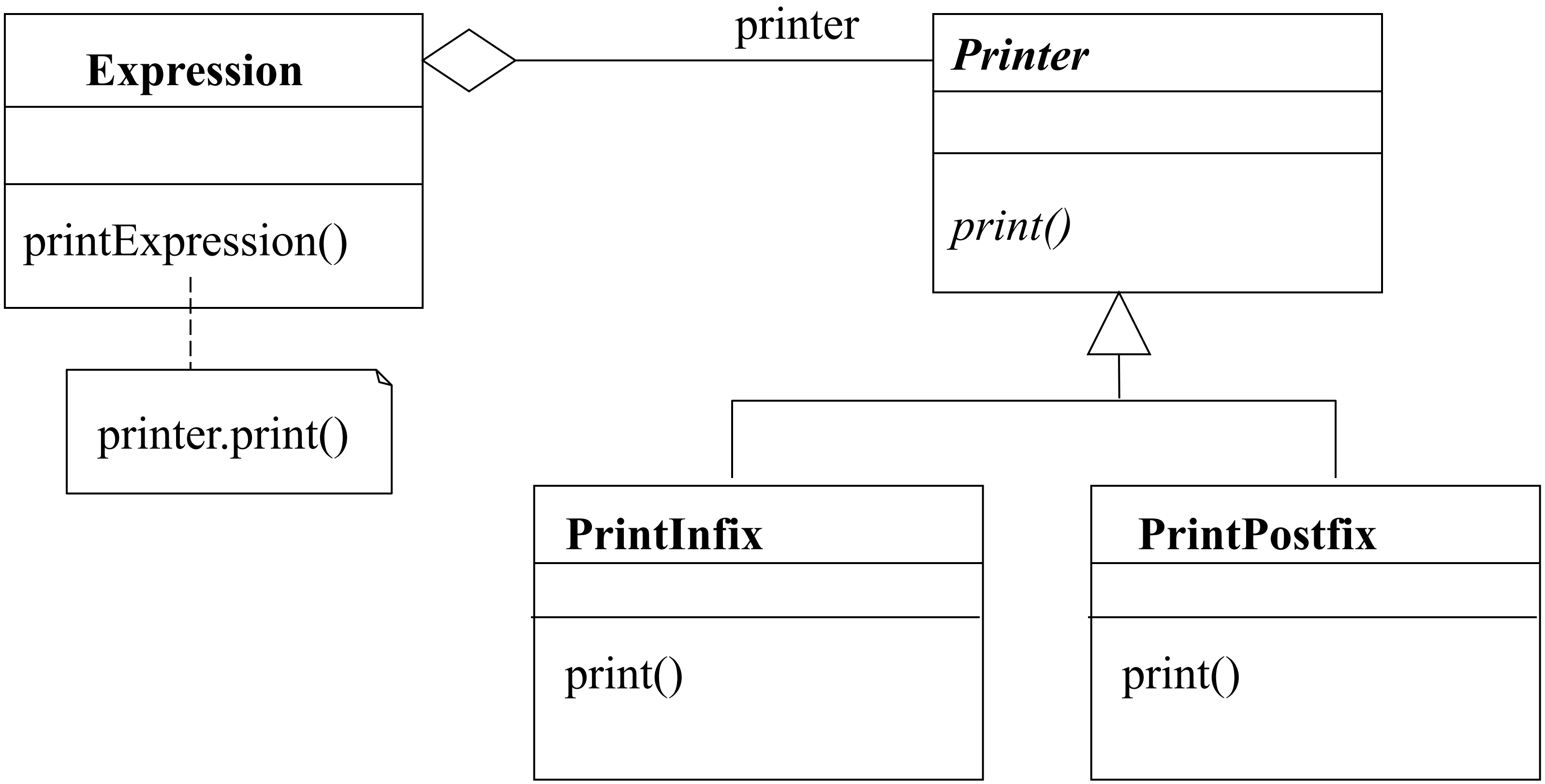
We Used the Composite Pattern



Problem 2: Supporting Different Print Styles

- We want to be able to print the expressions in different format
- For example, given the expression: – x + 2 * y + 1
  - We may want to print it infix (fully parenthesized)
    (((– x) + (2 * y)) + 1)
  - or we may want to print it in postfix form:
    x–2 y * + 1 +

Printing Expressions



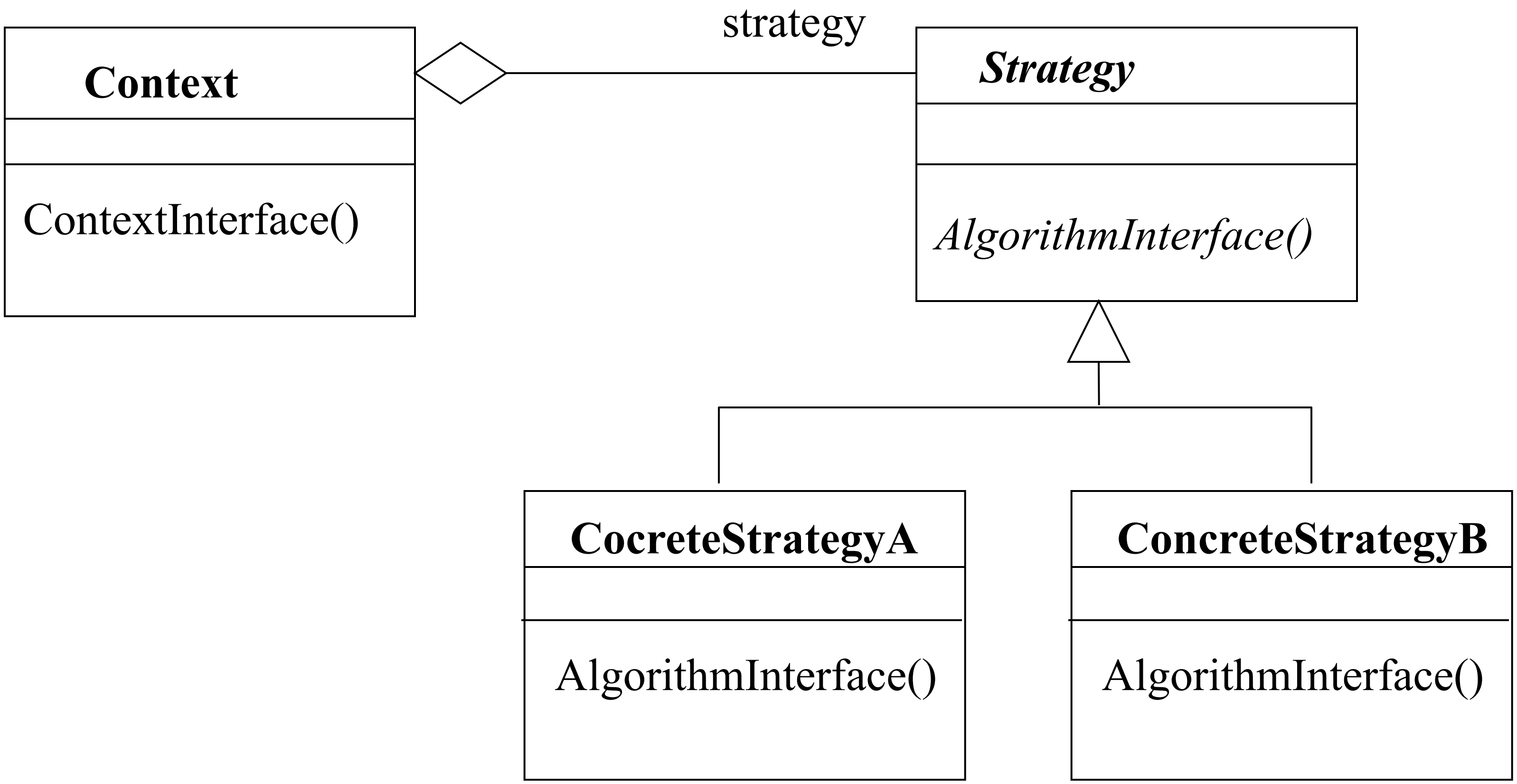
Printing Expressions

- Using strategy pattern enables us to encapsulate the printing algorithm behind a common interface.
- The client code does not have to know what type of printing strategy is being used
  - hence if the printing strategy changes we do not have to change the client code

```
Expression e = new Expression(...);
e.printer = new PrintPostfix(...);

...

e.printExpression();
```

client code

We Used the Strategy Pattern

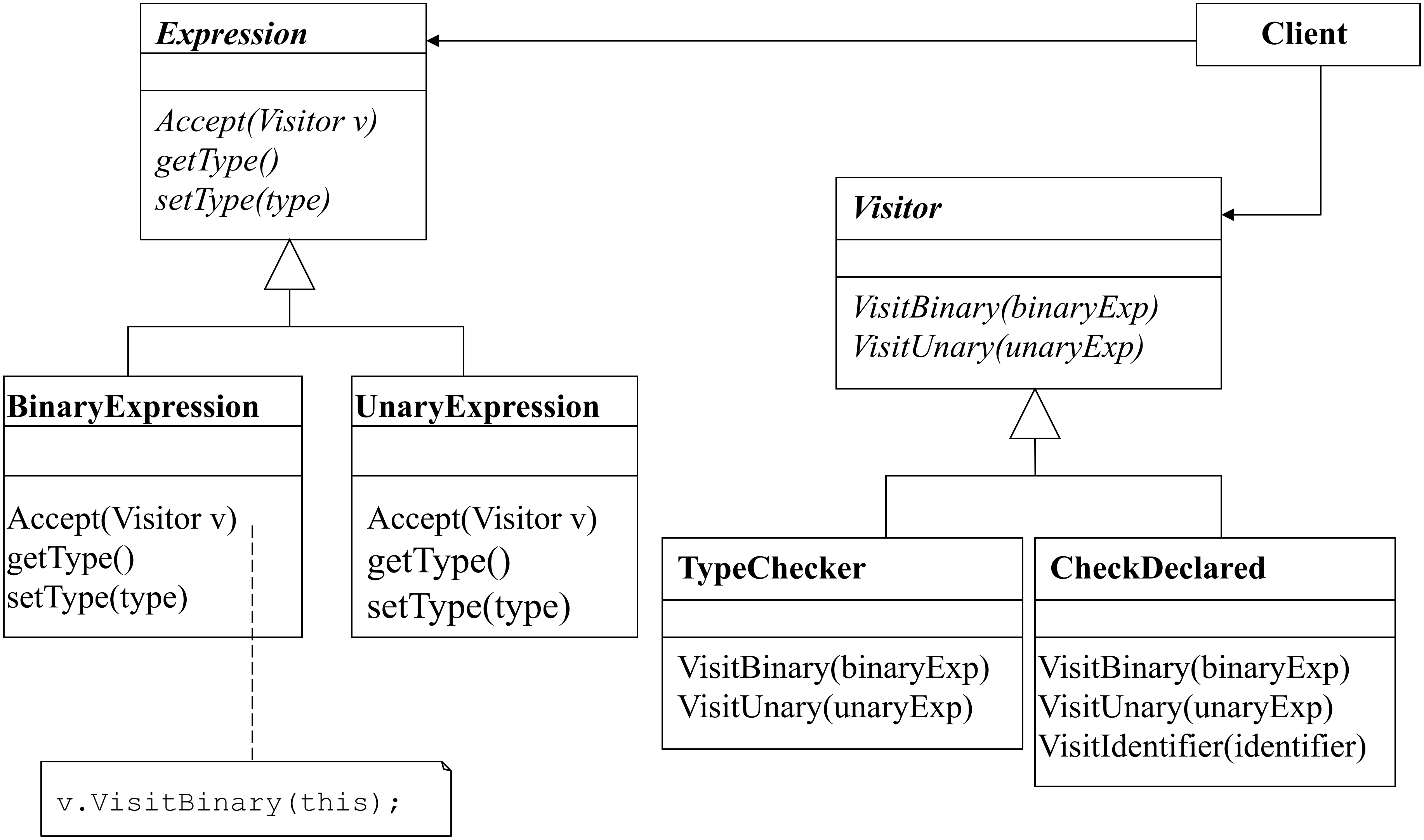


Problem 3: Checking Expressions

- We need to do type checking on expressions
  - For example, arguments of an addition operation should be integers; types of left and right children of an equality expression should match, etc.

- We may need to add other checks later on
  - For example, are all the identifiers used in the expression have been declared

Checking Expressions



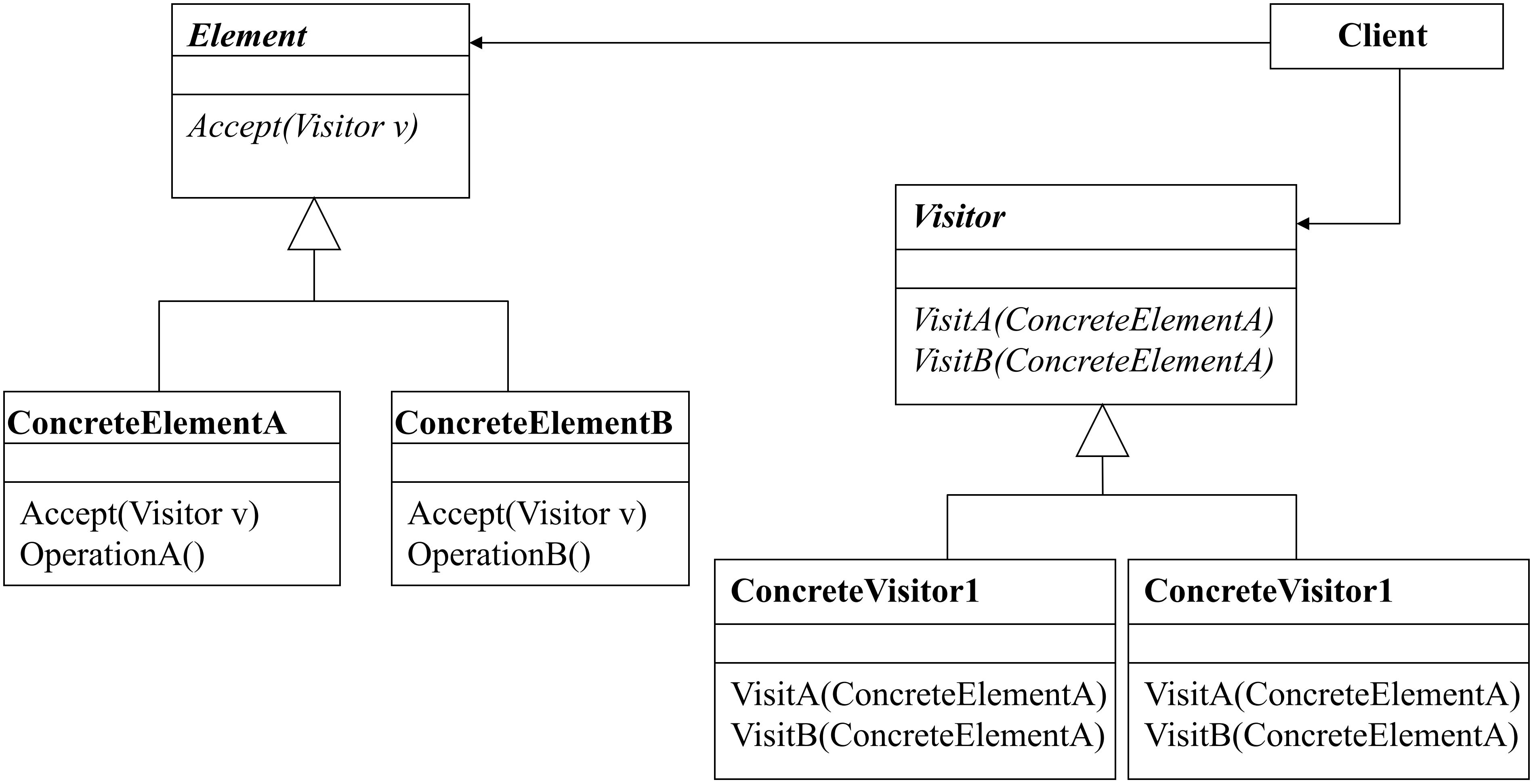
```
v.VisitBinary(this);
```

Checking Expressions

- The visitBinary method first calls the Accept methods of the left and right children and passes itself (type-checker) as the argument
  - This will type check all subexpressions recursively
- If the children have type errors or if the types of children do not match it sets its own type to type error

```
VisitBinary(binaryExp e) {
  e.left.Accept(this);
  e.right.Accept(this);
  if (e.left.getType() == type_error
       || e.right.getType() == type_error
       || e.left.getType() != e.right.getType())
     e.setType(type_error);
  else
     e.setType(...); // the argument here will depend
                     // on the type of the operator
}
```

We Used the Visitor Pattern



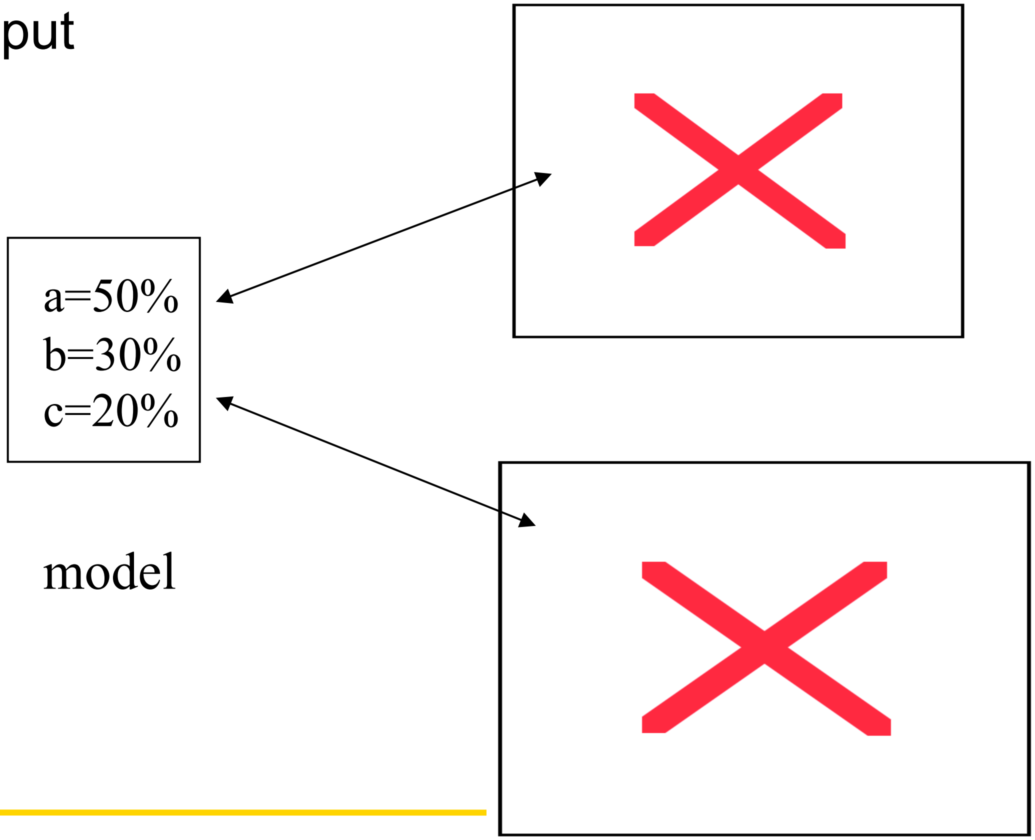
Observer Pattern

- Observer pattern is a pattern based on Model-View-Controller (MVC) architecture

- MVC is a design structure for separating representation from presentation using a subscribe/notify protocol

Model-View-Controller (MVC) Architecture

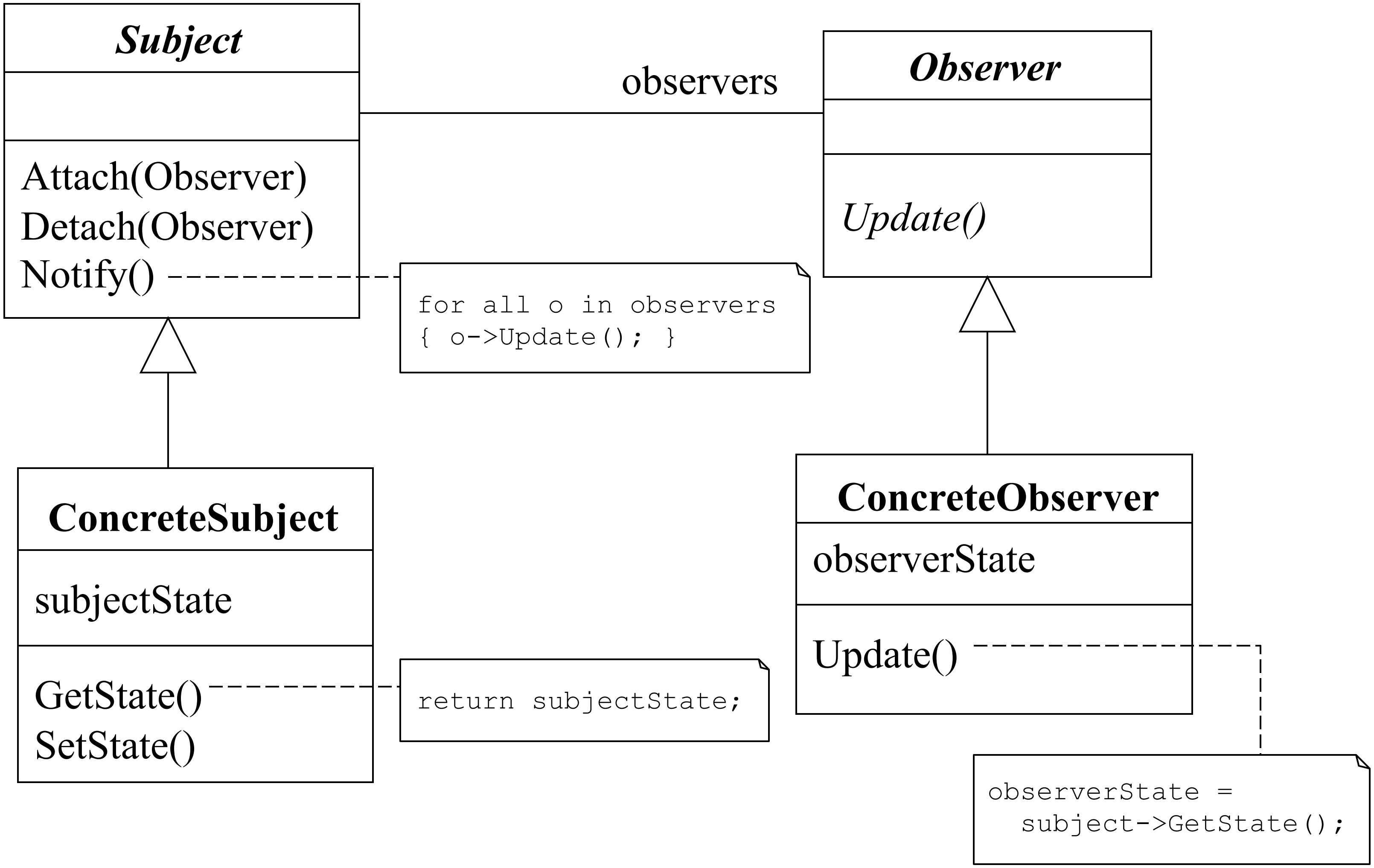
- MVC consists of three kinds of objects
  - Model is the application object
  - View is its screen presentation
  - Controller defines the way the user interface reacts to user input

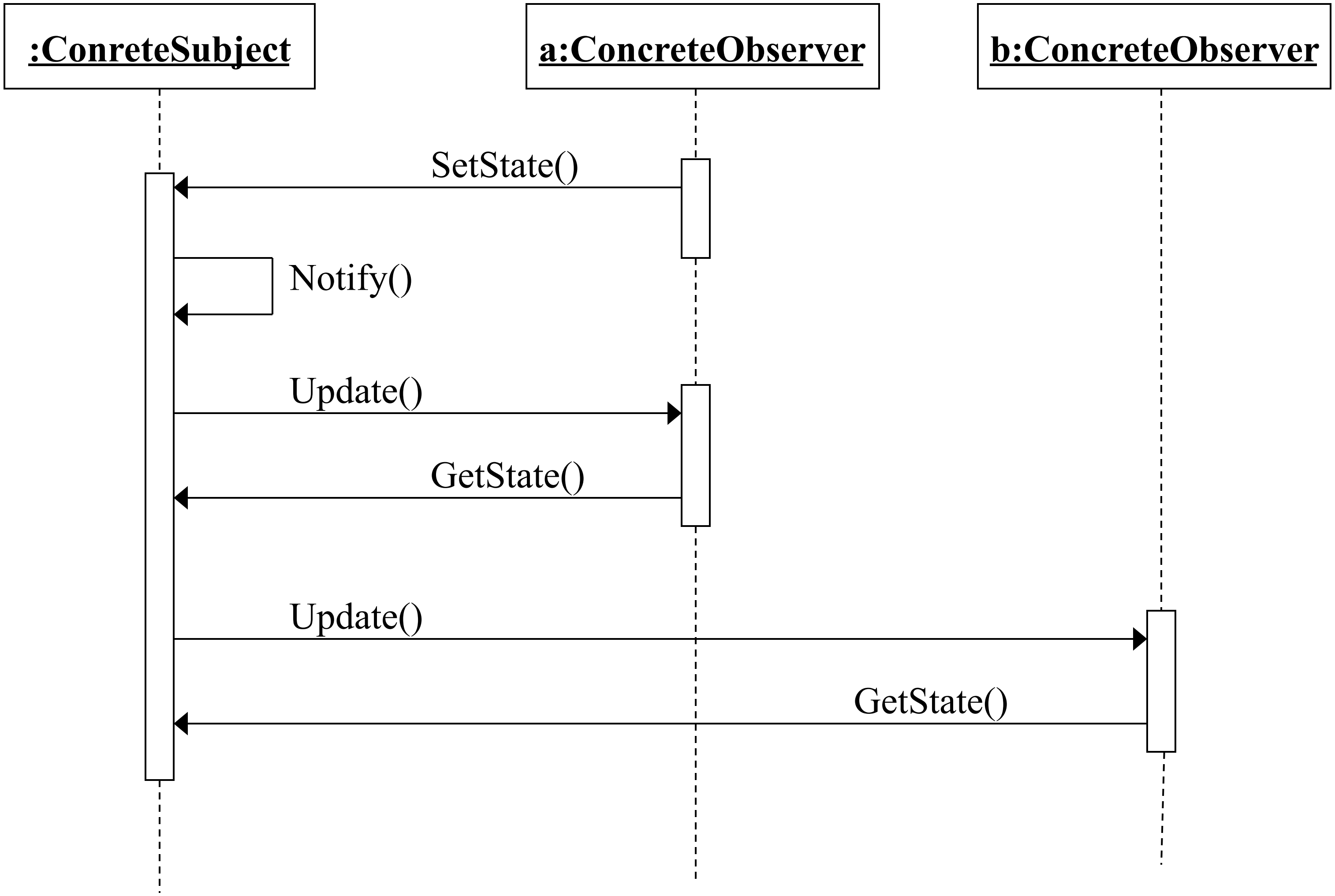


Model-View-Controller (MVC) Architecture

- MVC decouples views and models by establishing a subscribe/notify protocol between them
  - whenever model changes it notifies the views that depend on it
  - in response each view gets an opportunity to update itself
- This architecture allows you to attach multiple views to a model
  - it is possible to create new views for a model without rewriting it
- Taken at face value this may be seen as an architecture for user interface design
  - It is actually addresses a more general problem:
    - decoupling objects so that changes to one can affect any number of others without requiring the changed object to know the details of the others
  - This is called **Observer** pattern in the design patterns catalog

Class Diagram for the Observer Pattern




:ConreteSubject
a:ConcreteObserver
b:ConcreteObserver
SetState()
Notify()
Update()
GetState()
Update()
GetState()

A Case Study on Design Patterns

- A case study on design patterns for Communication Software:
  - “Using Design Patterns to Develop Reusable Object-Oriented Communication Software”, by D. C. Schmidt

- A case study on using design patterns to develop reusable object-oriented communication software

- Defines a new design pattern called Reactor

Conclusions from the case study

- Patterns enable widespread reuse of software architecture
  - Patterns explicitly capture knowledge that experienced developers already understand implicitly
  - Pattern descriptions explicitly record engineering trade-offs and design alternatives
  - The contexts where patterns apply and do not apply must be carefully documented

- Patterns improve communication within and across software development teams
  - Patterns facilitate training of new developers
  - Pattern names should be chosen carefully and used consistently
  - Successful pattern descriptions capture both structure and behavior

Conclusions from the case study

- Useful patterns arise from practical experience
  - Pattern authors should be directly involved with application developers and domain experts
  - Pattern descriptions should contain concrete examples
  - Patterns are validated by experience rather than by testing

- Everything should not be recast as a pattern
  - The focus should be on developing patterns that are strategic to the domain and reusing existing tactical patterns

- Integrating patterns into a software development process is a human-intensive activity
  - Rewards should be institutionalized for developing patterns
  - Patterns can be considered deliverables such as code

Conclusions from the case study

- Patterns help to transcend “programming language centric” viewpoints
  - However, implementing patterns efficiently requires careful selection of language features

- Managing expectations is crucial to using patterns effectively
  - Using patterns does not guarantee flexible and efficient software
  - Patterns may lead developers to think they know more about the solution to a problem than they actually do