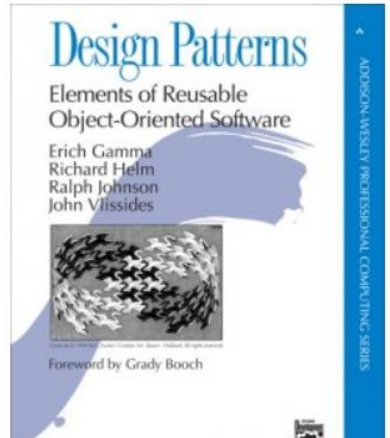
COSC 340: Software Engineering

Design Patterns

Audris Mockus: adapted from slides by Michael Jantz

Recommended text: *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides



Why Design Patterns?

- Designing object-oriented software is hard
  - Designing *reusable* object-oriented software is even harder
- Good designers *do not* solve every problem from first principles
  - When a good solution is found, reuse it over and over again
- Patterns make it easier to reuse successful designs and architectures
  - Pattern vocabulary makes systems more accessible to new developers
  - Help you choose reusable designs
  - Improve the documentation and maintainability of existing systems
  - Help you get the design 'right' faster

Elements of Design Patterns

- Pattern name
  - A one or two word handle for describing the pattern
- Problem
  - When to apply the pattern (the problem and its context)
- Solution
  - Elements of the design, their relationships, responsibilities and collaborations
- Consequences
  - Results and trade-offs of applying the pattern

What is a Design Pattern?

- Depends on your point of view
- Useful to concentrate on a certain level of abstraction
  - Do not consider objects that can be encoded and reused as is (e.g. linked lists and hash tables)
  - Do not consider complex, domain-specific designs for an entire system
- GoF Definition:
  - Design patterns are "*descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*"

Classifying Design Patterns

- Purpose (what does the pattern do?)
  - Creational (object creation)
  - Structural (composition of classes or objects)
  - Behavioral (characterize object interaction)
- Scope (where does it apply – classes or objects?)
  - Class patterns
    - Deal with relationships b/w classes and their subclasses
    - Static – fixed at compile time
  - Object patterns
    - Deal with object relationships
    - More dynamic – can be changed at runtime

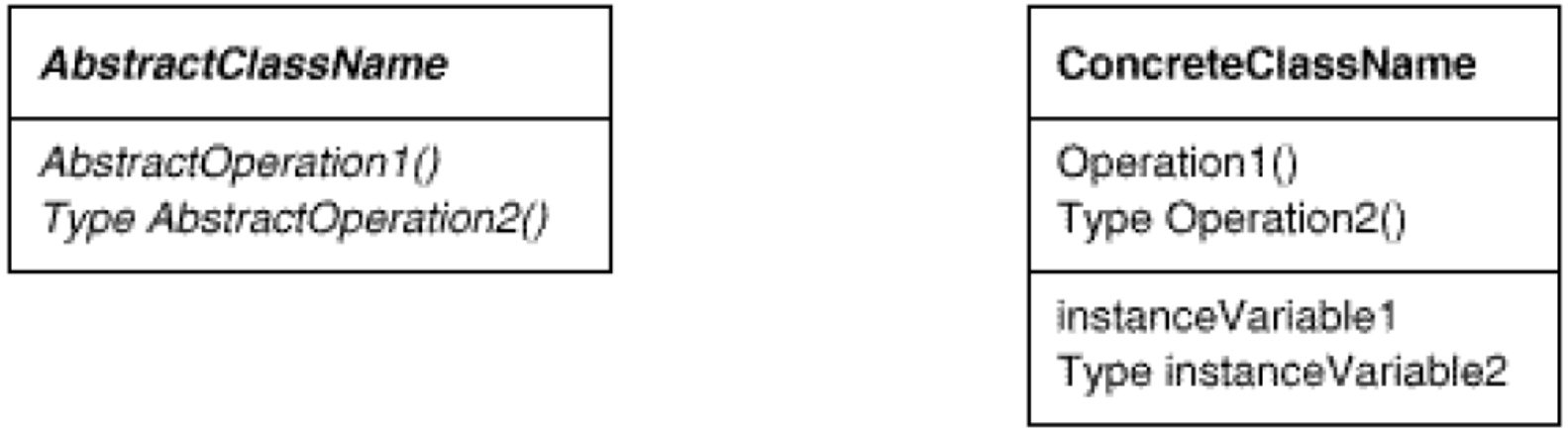
Organizing the Catalog

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

OMT Notation

- Three diagrammatic notations
  - Class diagram
    - depicts classes, their structure, and the static relationships between them
  - Object diagram
    - depicts a particular object structure at run-time
  - Interaction diagram
    - shows the flow of requests between objects

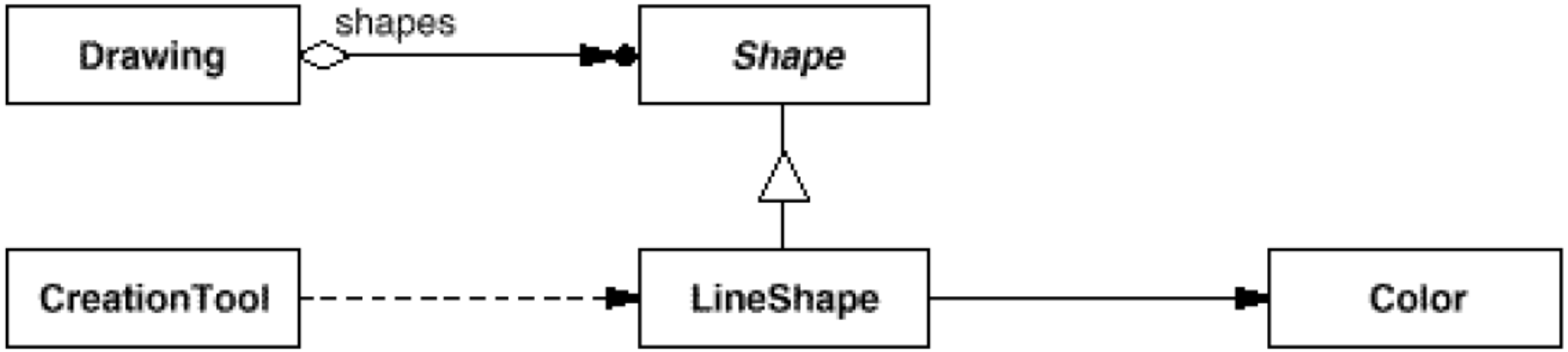
Class Diagrams



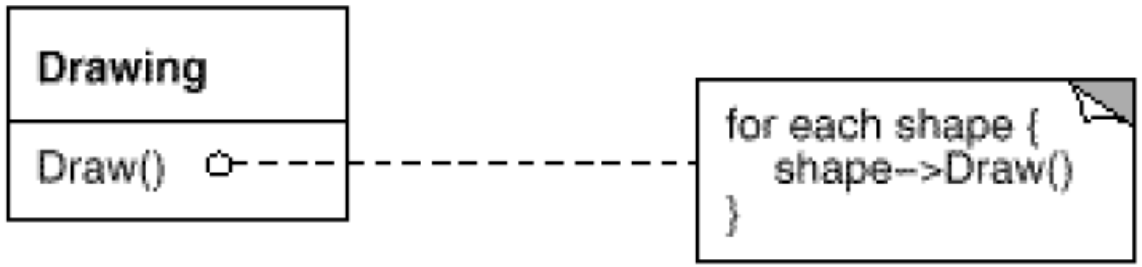
(a) Abstract and concrete classes

(b) Participant Client class (left) and implicit Client class (right)

Class Diagrams



(c) Class relationships



(d) Pseudocode annotation

Object Diagrams

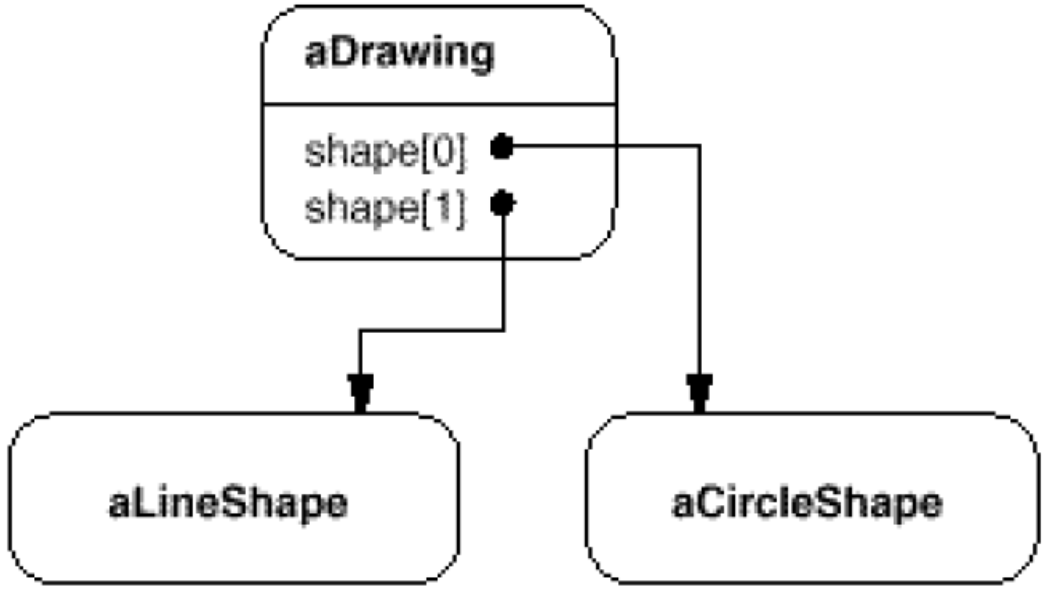


Figure B.2: Object diagram notation

Interaction Diagram

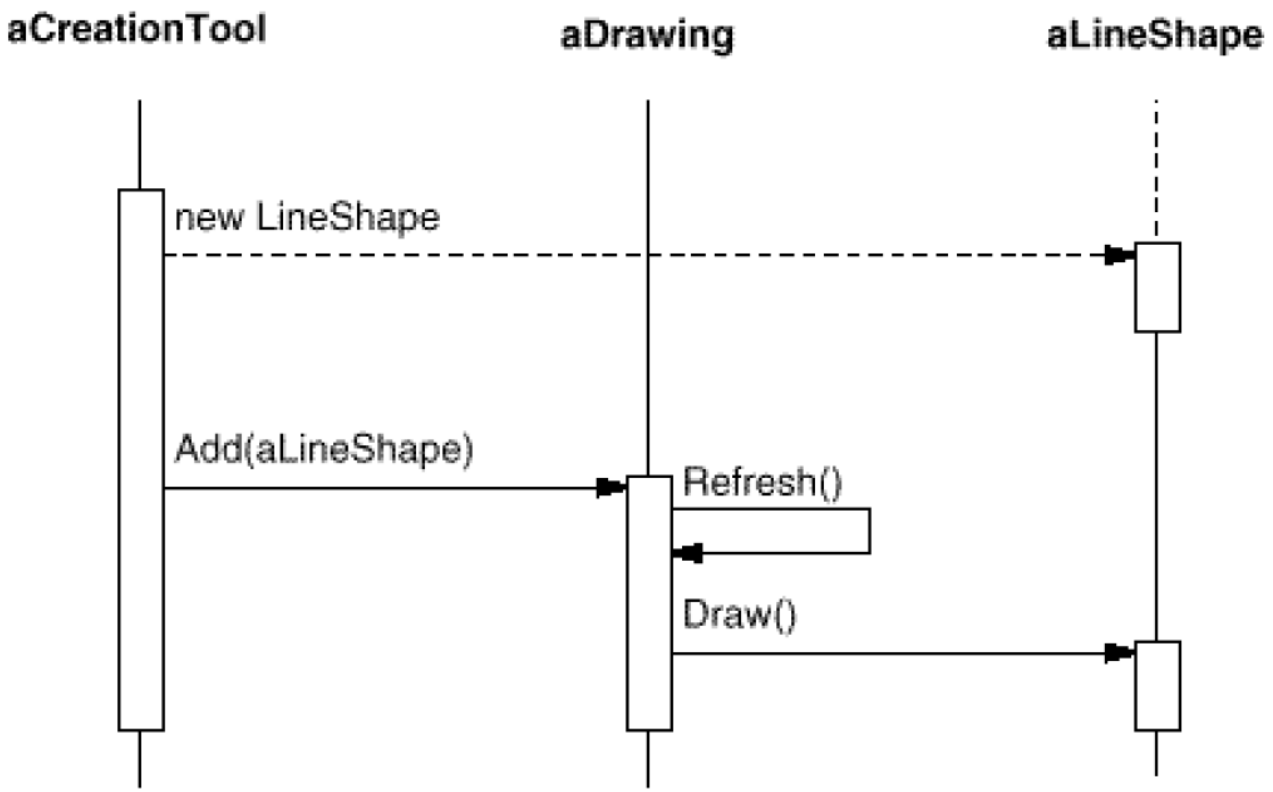


Figure B.3: Interaction diagram notation

Singleton Pattern

- Intent
  - Ensure a class only has one instance, and provide a global point of access to it
- Motivation
  - Often important for classes to have exactly one instance (e.g. we only want one file system or window manager)
  - Global variable does not prevent you from instantiating multiple objects
- Solution
  - Class itself is responsible for keeping track of the sole instance
  - The class ensures no other instances can be created, and provides access to the sole instance

Singleton Pattern Implementation:

Ensuring a Unique Instance

```
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};

Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance () {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

Singleton Pattern Implementation:

Ensuring a Unique Instance

```
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};

Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance () {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

- Access the singleton through the Instance function
- _instance variable initialized on first access
- Why not just define a global or static instance variable (initialize automatically)?
  - Cannot guarantee only one instance of the Singleton type
  - Runtime initialization allows you to define when the object is initialized

Singleton Pattern Example: MazeFactory

```
class MazeFactory {
public:
    static MazeFactory* Instance();

    // existing interface goes here
protected:
    MazeFactory();
private:
    static MazeFactory* _instance;
};
```

```
MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {

    if (_instance == 0) {
        _instance = new MazeFactory;
    }
    return _instance;
}
```

Singleton Pattern Example: MazeFactory

```
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        const char* mazeStyle = getenv("MAZESTYLE");

        if (strcmp(mazeStyle, "bombed") == 0) {
            _instance = new BombedMazeFactory;

        } else if (strcmp(mazeStyle, "enchanted") == 0) {
            _instance = new EnchantedMazeFactory;

        // ... other possible subclasses

        } else {        // default
            _instance = new MazeFactory;
        }
    }
    return _instance;
}
```

Singleton Pattern: Uses and Benefits

- Use Singleton when
  - There must be exactly one instance of a class, and it must be accessible from a well-known access point
  - The sole instance should be extensible by sub-classing, and you want to be able to use an extended instance without modifying other code
- Benefits
  - Controlled access to a single instance
  - Reduced name space (no global variables in the name space)
  - Singleton class can be sub-classed – and you can configure the application with an instance of the class you need at run-time
  - Can flexibly change the number of allowable instances

Behavioral Patterns

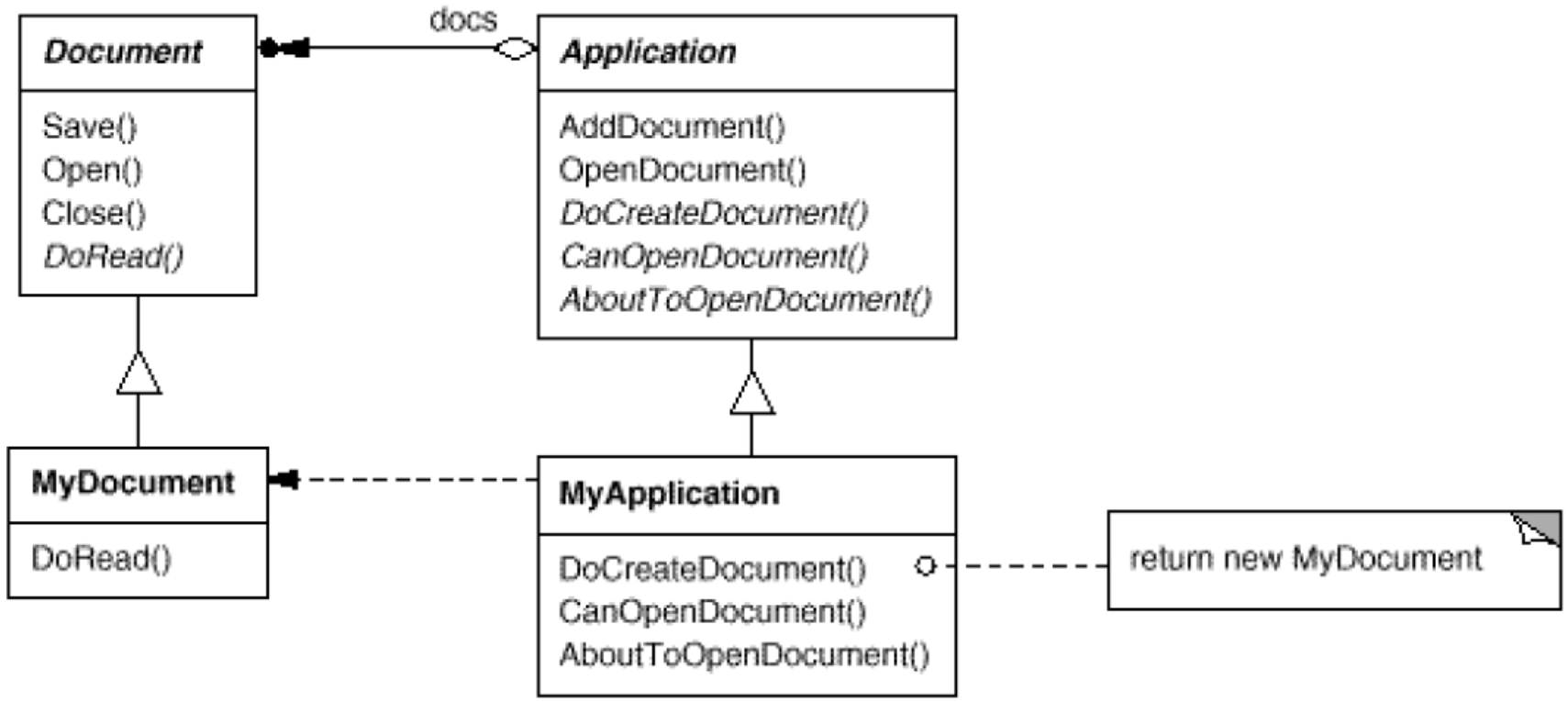
- Concerned with algorithms and the assignment of responsibilities between objects
- Two types
  - Behavioral class patterns
  - Behavioral object patterns
- Behavioral patterns we will look at
  - Template Method
  - Observer
  - Iterator

Template Method Pattern

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses
- Motivation
  - Often useful to allow subclasses to redefine parts of an algorithm without changing the algorithm's structure
- Solution
  - Define an algorithm in terms of abstract operations that subclasses override to provide concrete behavior
- Example: app framework with Application and Document classes

Template Method Pattern: Example

Application Framework with Application and Document classes



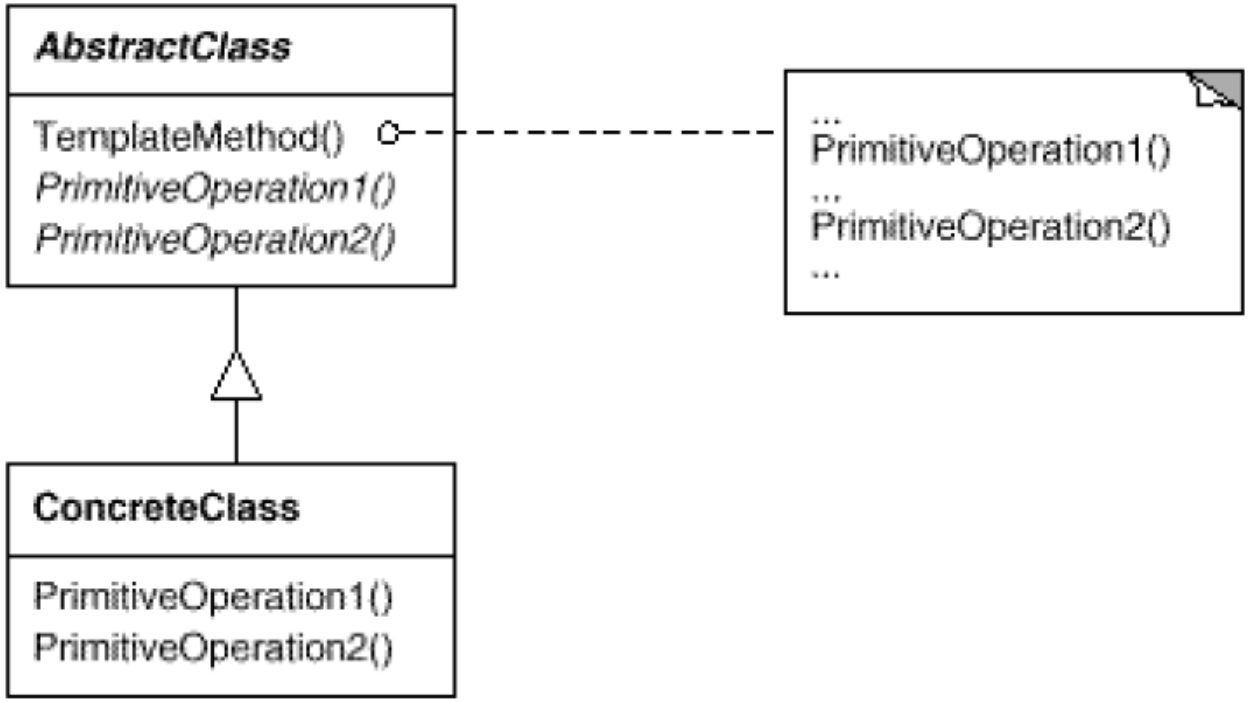
Template Method Pattern: Example

Application Framework with Application and Document classes

```
void Application::OpenDocument (const char* name) {
    if (!CanOpenDocument(name)) {
        // cannot handle this document
        return;
    }

    Document* doc = DoCreateDocument();

    if (doc) {
        _docs->AddDocument(doc);
        AboutToOpenDocument(doc);
        doc->Open();
        doc->DoRead();
    }
}
```

- TM fixes the ordering of the steps, but allows subclasses to vary what happens at each step

Template Method Pattern: Structure



Template Method Pattern: Participants

- **AbstractClass** (Application)
  - Defines abstract **primitive operations** that concrete subclasses define to implement steps of an algorithm
  - Implements a template method defining the skeleton of an algorithm
- **ConcreteClass** (MyApplication)
  - Implements the primitive operations to carry out subclass-specific steps of the algorithm

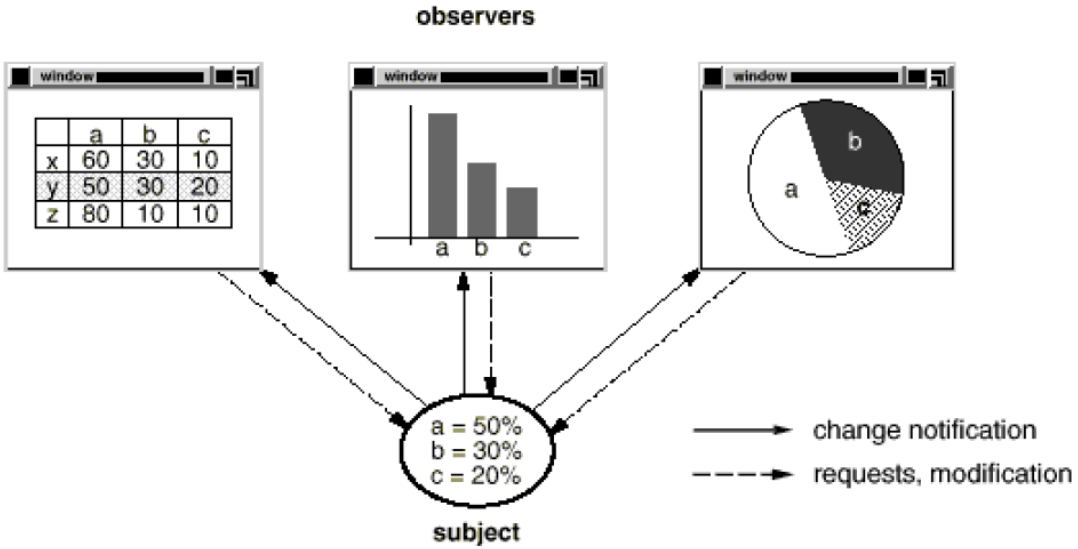
Observer Pattern

- Define a dependencies between objects so that when one object changes state, all its dependents are updated automatically
- Motivation
  - Often need to maintain consistency between related objects
  - But want to avoid making the classes tightly coupled
- Solution
  - Define a one-to-many relationship between subject and observers
  - Observers are notified whenever the subject changes state
  - In response, each observer queries the subject to synchronize its state with the subject's state

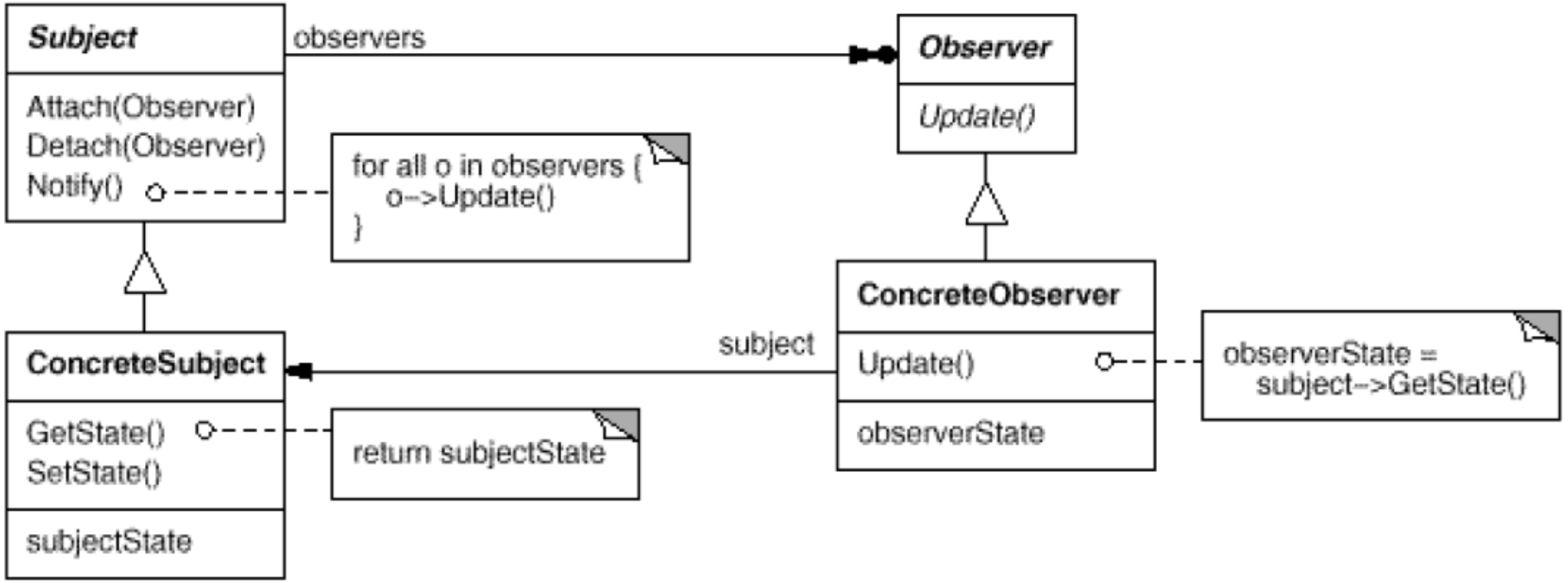
Observer Pattern: Example

Model and View Components

- View components (spreadsheet, bar chart, etc.) are dependent on the Model component and should be notified when the Model changes
- Observer pattern
  - Establishes a subject / observer relationship
  - Subject sends a message to the observers when the state has changed
- Also known as publish-subscribe
  - Subject publishes notifications
  - Any number of observers can subscribe



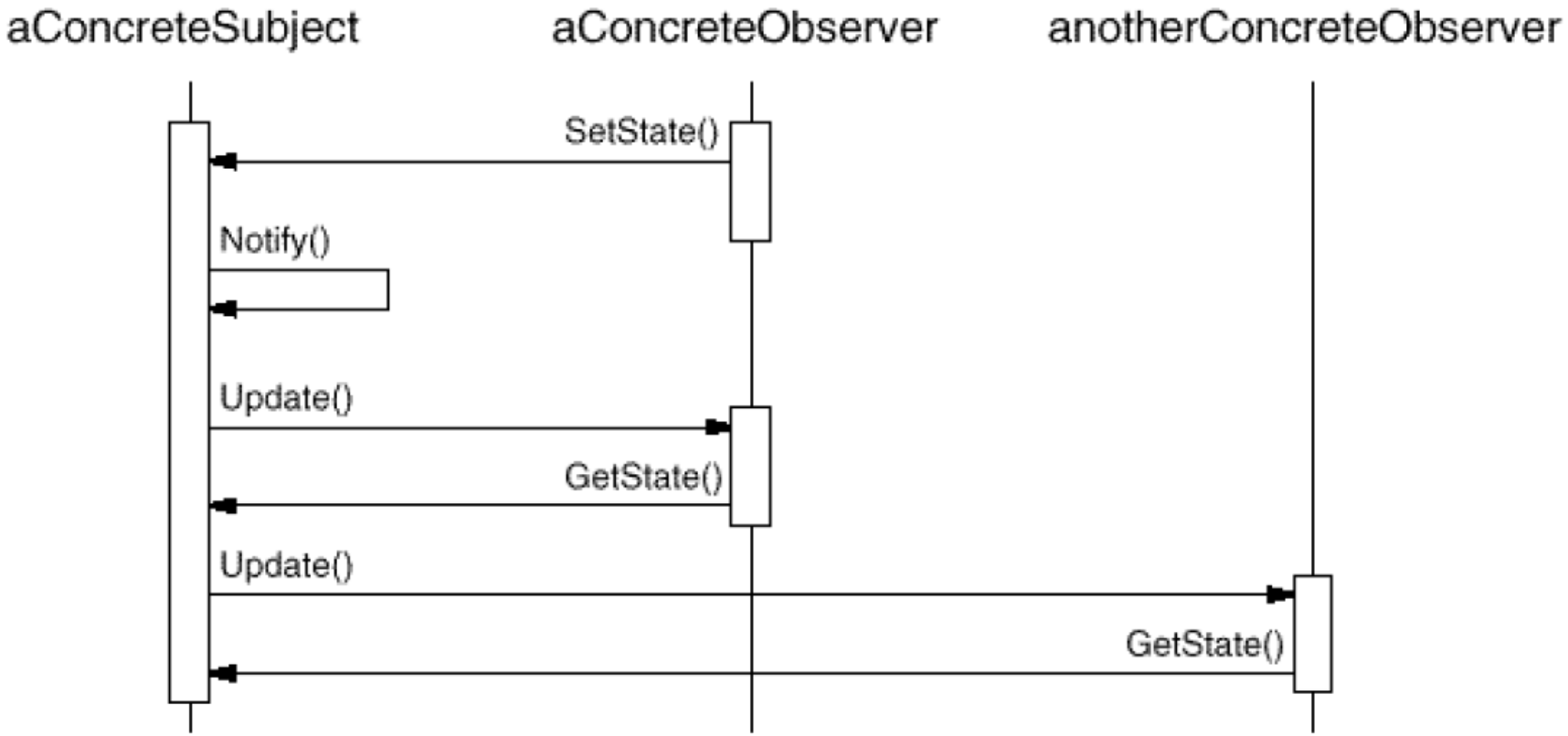
Observer Pattern: Structure



Observer Pattern: Participants

- **Subject**
  - Knows its observers. Any number of Observer objects may observe a subject
  - Provides an interface for attaching and detaching Observer objects
- **Observer**
  - Defines an updating interface for objects that should be notified of changes in a subject
- **ConcreteSubject**
  - Stores state of interest to ConcreteObserver objects
  - Sends a notification to its observers when its state changes
- **ConcreteObserver**
  - Maintains a reference to a ConcreteSubject object
  - Stores state that should stay consistent with the subject's
  - Implements the Observer updating interface to keep its state consistent with the subject's

Observer Pattern: Collaborations



Observer Pattern: Sample Code

```
class Subject;

class Observer {
public:
    virtual ~ Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
protected:
    Observer();
};
```

- Abstract class for the Observer interface
  - Supports multiple subjects for each observer

Observer Pattern: Sample Code

```
class Subject {
public:
    virtual ~Subject();

    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();
protected:
    Subject();
private:
    List<Observer*> *_observers;
};
```

```
void Subject::Attach (Observer* o) {
    _observers->Append(o);
}

void Subject::Detach (Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify () {
    ListIterator<Observer*> i(_observers);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}
```

- Abstract class for the Subject interface

Observer Pattern: Sample Code

```
class ClockTimer : public Subject {
public:
    ClockTimer();

    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();

    void Tick();
};
```

- ClockTimer is a concrete subject for maintaining the time of day

```
void ClockTimer::Tick () {
    // update internal time-keeping state
    // ...
    Notify();
}
```

- Tick is called by an internal timer at regular intervals

Observer Pattern: Sample Code

```
class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();

    virtual void Update(Subject*);
        // overrides Observer operation

    virtual void Draw();
        // overrides Widget operation;
        // defines how to draw the digital clock
private:
    ClockTimer* _subject;
};
```

- DigitalClock class for displaying the time

Observer Pattern: Sample Code

```
DigitalClock::DigitalClock (ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

DigitalClock:: DigitalClock () {
    _subject->Detach(this);
}

void DigitalClock::Update (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
    }
}

void DigitalClock::Draw () {
    // get the new values from the subject

    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
    // etc.

    // draw the digital clock
}
```

- Constructor and destructor attach and detach the DigitalClock from its subject

- On update, draw the digital clock with the updated hour and minute

Observer Pattern: Sample Code

```
ClockTimer* timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```

- Code for creating an AnalogClock and DigitalClock with the same subject
- Whenever the timer ticks, the clocks update and redisplay themselves

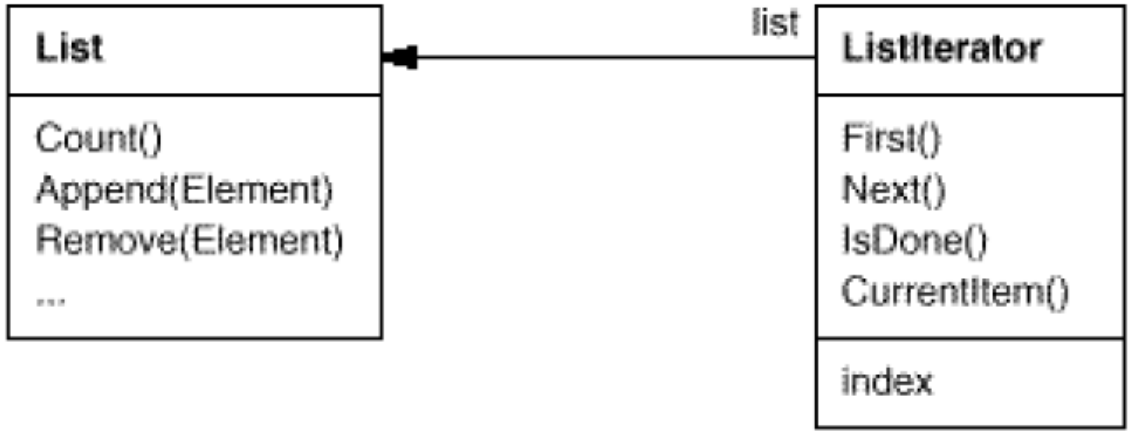
Iterator Pattern

- Provide a way to access elements of an aggregate object sequentially without exposing its underlying representation
- Motivation
  - Aggregate objects (e.g. lists, sets, containers) should have a way to access their elements without exposing their internal structures
  - Might also want to traverse the aggregate object in different ways
  - Or have more than one traversal pending at a time
- Solution
  - Responsibility for access and traversal is removed from the aggregate object and put into an iterator object
  - Iterator class defines an interface for accessing the object's elements
  - Each iterator object keeps track of the current element

Iterator Pattern: Example

List class with a ListIterator

- Creating a ListIterator instance requires a List to traverse
- ListIterator allows you to access List items sequentially
- ListIterator methods
  - CurrentItem: current element in the list
  - First: initializes current element to the first element
  - Next: advances the current element to the next element
  - IsDone: tests whether we've advanced past the last element



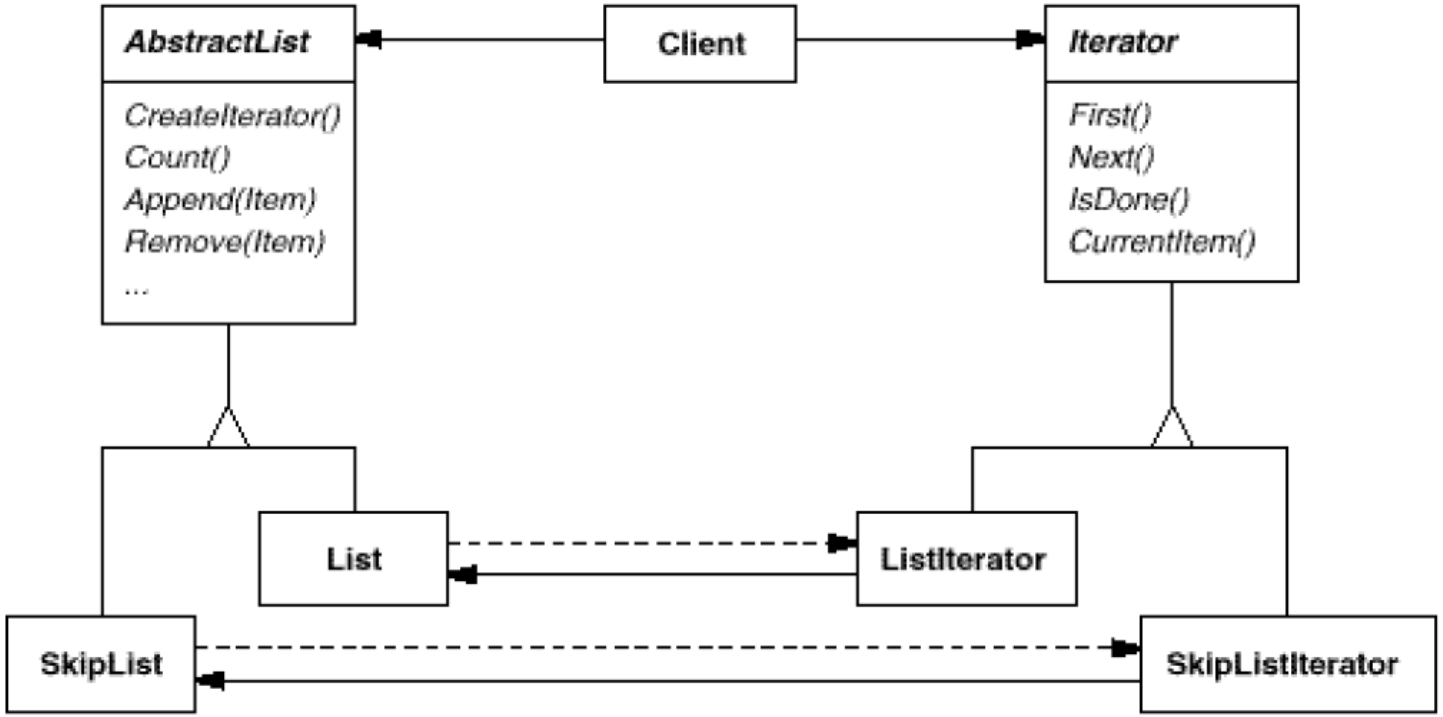
Iterator Pattern: Example

List class with a ListIterator

- Simple approach requires the client code to know that the object it wants to traverse is a List
- Instead, we can define an abstract Iterator class to support **polymorphic iteration**.

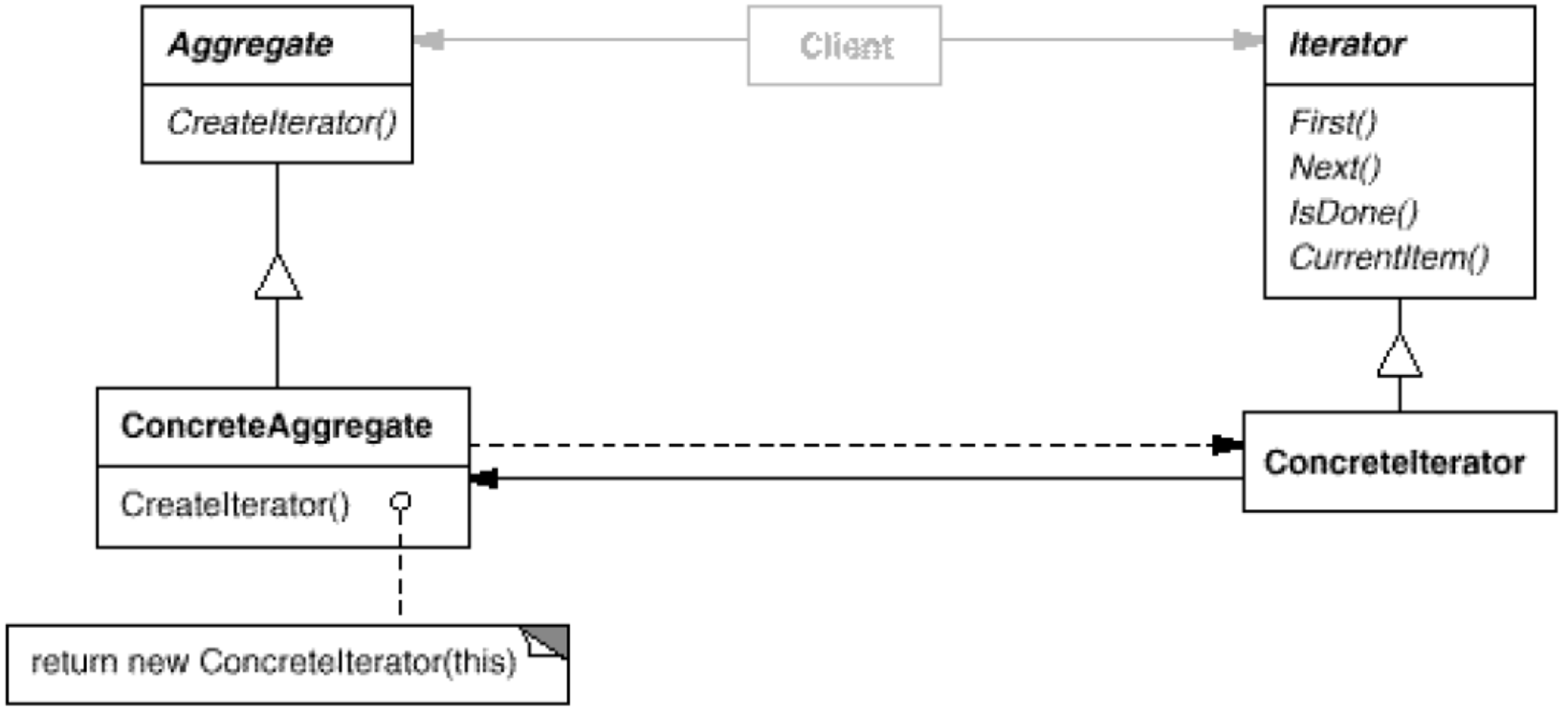
Iterator Pattern: Example

List class with a ListIterator



- Want to be able to write client code for both List and SkipList objects
- Define an abstract List class and abstract Iterator class

Iterator Pattern: Structure



Iterator Pattern: Participants

- **Iterator**
  - Defines an interface for accessing and traversing elements
- **ConcreteIterator**
  - Implements the Iterator interface
  - Keeps track of the current position in the traversal of the aggregate
- **Aggregate**
  - Defines an interface for creating an Iterator object
- **ConcreteAggregate**
- Implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

Iterator Pattern: Sample Code

Interface Definitions

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);

    long Count() const;
    Item& Get(long index) const;
    // ...
};
```

- List interface

```
template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```

- Iterator interface

Iterator Pattern: Sample Code

Concrete ListIterator Definition

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
    ListIterator(const List<Item>* aList);
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;

private:
    const List<Item>* _list;
    long _current;
};
```

```
template <class Item>
ListIterator<Item>::ListIterator (
    const List<Item>* aList
) : _list(aList), _current(0) {
}


template <class Item>
void ListIterator<Item>::First () {
    _current = 0;
}

template <class Item>
void ListIterator<Item>::Next () {
    _current++;
}
```

Iterator Pattern: Sample Code

Concrete ListIterator Definition

```
template <class Item>
bool ListIterator<Item>::IsDone () const {
    return _current >= _list->Count();
}
```

```
template <class Item>
Item ListIterator<Item>::CurrentItem () const {
    if (IsDone()) {
        throw IteratorOutOfBounds;
    }
    return _list->Get(_current);
}
```

Iterator Pattern: Sample Code

Using the Iterators

```
void PrintEmployees (Iterator<Employee*>& i) {
    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Print();
    }
}

List<Employee*>* employees;
// ...
ListIterator<Employee*> forward(employees);
ReverseListIterator<Employee*> backward(employees);

PrintEmployees(forward);
PrintEmployees(backward);
```

Structural Patterns

- Compose classes and objects to form larger structures
- Two types
  - Structural *class* patterns use inheritance to compose interfaces or implementations
  - Structural *object* patterns compose objects to realize new functionality
- Structural patterns we will cover
  - Composite
  - Decorator
  - Adapter

Composite Pattern

- Compose objects into tree structures that let you treat individual objects and compositions of objects uniformly
- Motivation
  - Having to distinguish between primitive objects and compositions of primitive objects can make the application more complex
  - Example: grouping components of a graphical application
- Solution
  - Use recursive composition of objects
  - Use an abstract class to represent both primitive objects and their containers

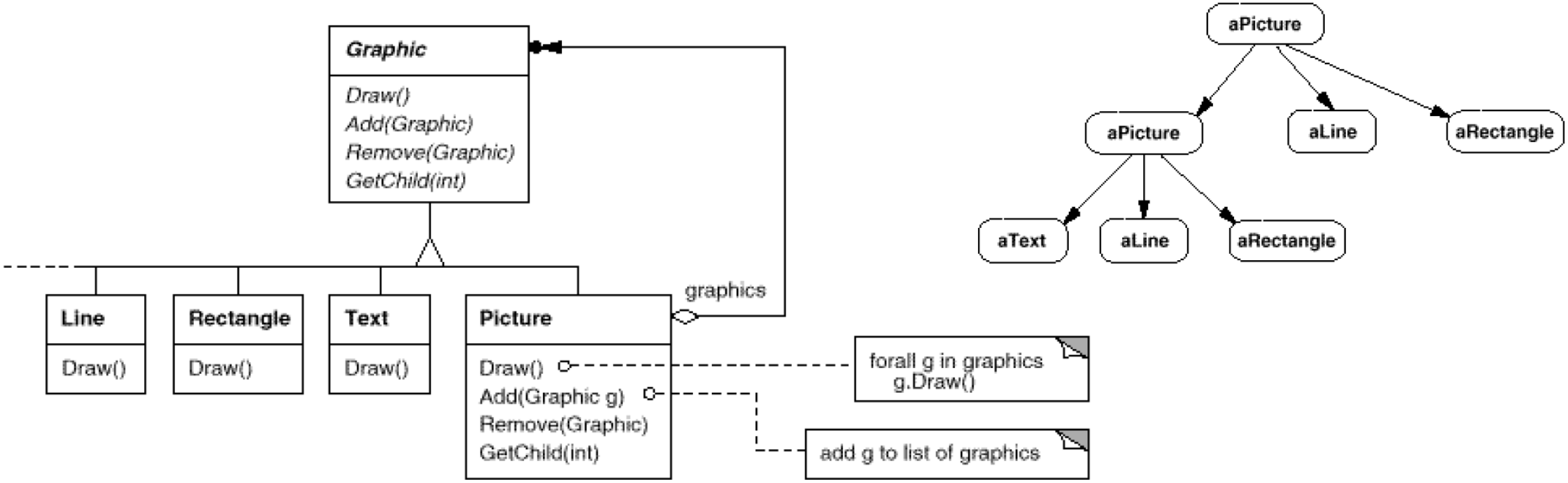
Composite Pattern: Motivation

Grouping Components in a Graphical Application

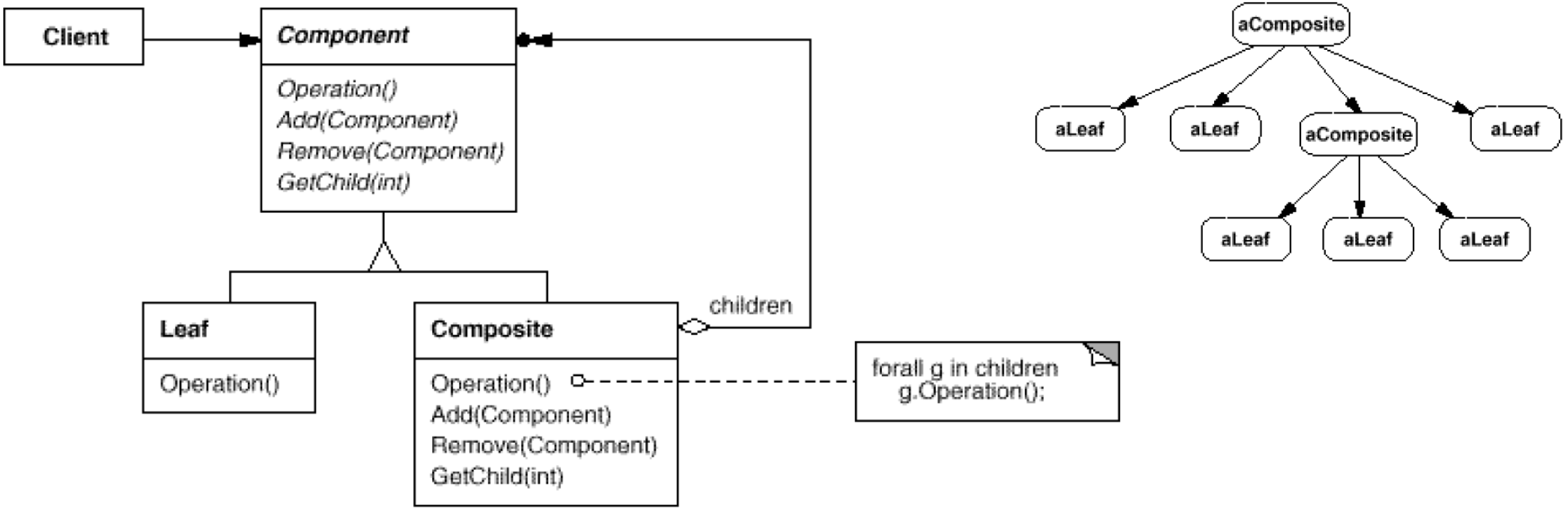
- Graphics applications (e.g. drawing editors) often build diagrams out of simple components
  - Users group components to form larger components
- Simple approach: classes for primitive objects (Line, Text) and classes for container objects
  - Problem: code must treat these objects differently – even if the user treats them identically
  - Desire a common interface for using and representing both primitive and container objects

Composite Pattern: Motivation

Grouping Components in a Graphical Application



Composite Pattern: Structure



Composite Pattern: Participants

- **Component** (Graphic)
  - Declares the interface for objects in the composition
  - Implements default behavior for the interface common to all classes, as appropriate
  - Declares an interface for accessing and managing its child components
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate
- **Leaf** (Rectangle, Line, Text, etc.)
  - Represents leaf objects in the composition (a leaf has no children)
  - Defines behavior for primitive objects in the composition
- **Composite** (Picture)
  - Defines behavior for components having children
  - Stores child components
  - Implements child-related operations in the Component
- **Client**
  - Manipulates objects in the composition through the Component interface.

Composite Pattern: Example

Electronic Equipment

```
class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator* CreateIterator();
protected:
    Equipment(const char*);
private:
    const char* _name;
};
```

- The Equipment class defines an interface for a hierarchy of electronic equipment

Composite Pattern: Example

Electronic Equipment

```
class FloppyDisk : public Equipment {
public:
    FloppyDisk(const char*);
    virtual ~FloppyDisk();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

- FloppyDisk is a Leaf class
- It defines the interface for one of the individual components in the hierarchy

Composite Pattern: Example

Electronic Equipment

```
class CompositeEquipment : public Equipment {
public:
    virtual ~CompositeEquipment();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator* CreateIterator();

protected:
    CompositeEquipment(const char*);
private:
    List _equipment;
};
```

- CompositeEquipment is a base class for equipment that contains other equipment

Composite Pattern: Example

Electronic Equipment

```
Currency CompositeEquipment::NetPrice () {
    Iterator* i = CreateIterator();
    Currency total = 0;

    for (i->First(); !i->IsDone(); i->Next()) {
        total += i->CurrentItem()->NetPrice();
    }
    delete i;
    return total;
}
```

- Operations for CompositeEquipment are defined in terms of their children

Composite Pattern: Example

Electronic Equipment

```
Cabinet* cabinet = new Cabinet("PC Cabinet");
Chassis* chassis = new Chassis("PC Chassis");

cabinet->Add(chassis);

Bus* bus = new Bus("MCA Bus");
bus->Add(new Card("16Mbs Token Ring"));

chassis->Add(bus);
chassis->Add(new FloppyDisk("3.5in Floppy"));

cout << "The net price is " << chassis->NetPrice() << endl;
```

- Client code treats Equipment and CompositeEquipment objects in a uniform way

Composite Pattern: Consequences

Benefits and Drawbacks

- Results in class hierarchies consisting of primitive objects and composite objects
- Benefits
  - Makes the client simpler
  - Makes it easier to add new types of components
- Drawbacks
  - Can make your design overly general

Adapter Pattern

- Convert (or "adapt") the interface of a class into another interface expected by the client code
- Also known as "wrapper"
- Motivation
  - A toolkit might not be usable only because its interface does not match the domain-specific interface expected by the application
  - Example: drawing editor program with text boxes
- Solution
  - Use multiple inheritance to adapt one interface to another
  - Use object composition to implement one interface in terms of another

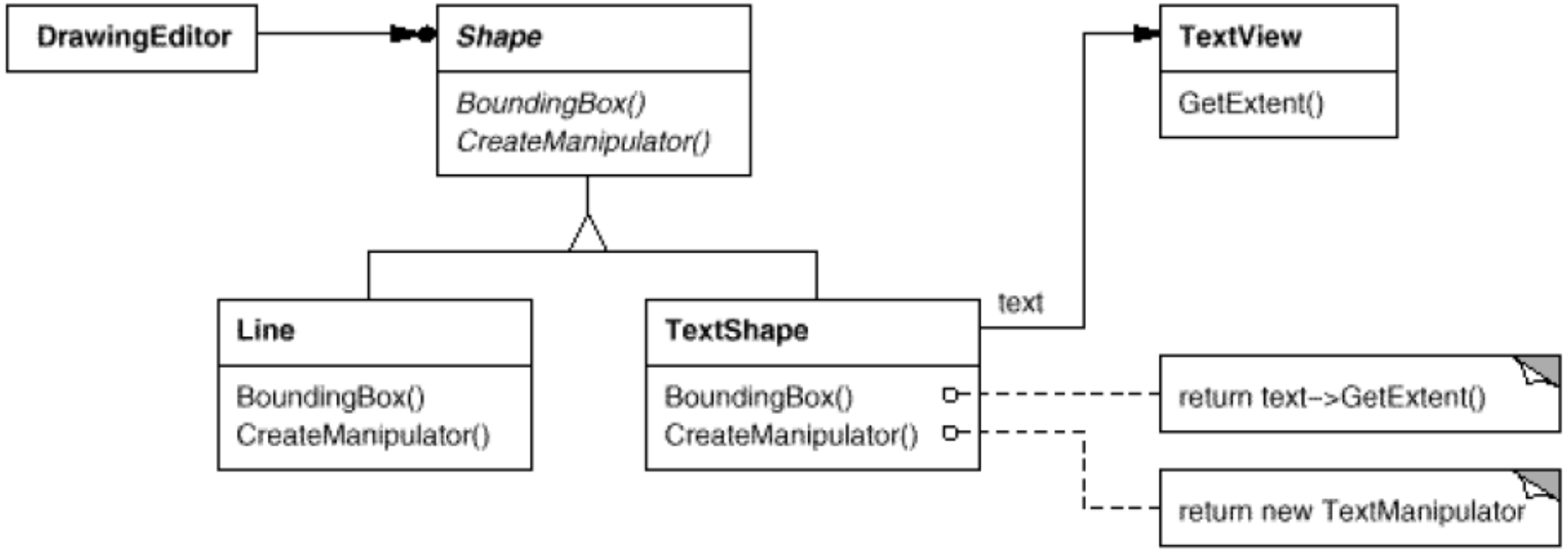
Adapter Pattern: Motivation

DrawingEditor Program

- Program allows users to draw and arrange graphical elements (lines, polygons, text) into pictures and diagrams
  - Shape class defines graphical objects (has an editable shape and can draw itself)
  - Each kind of graphical object is a subclass of shape (LineShape, PolygonShape, etc)
- TextShape is more difficult to implement
  - TextView class available from another toolkit, but does not fit the Shape interface
  - How to use TextView even though our drawing class must conform to a different and incompatible interface?
  - Solution: create an *adapter* class that adapts TextView's interface for use with the Drawing Editor program

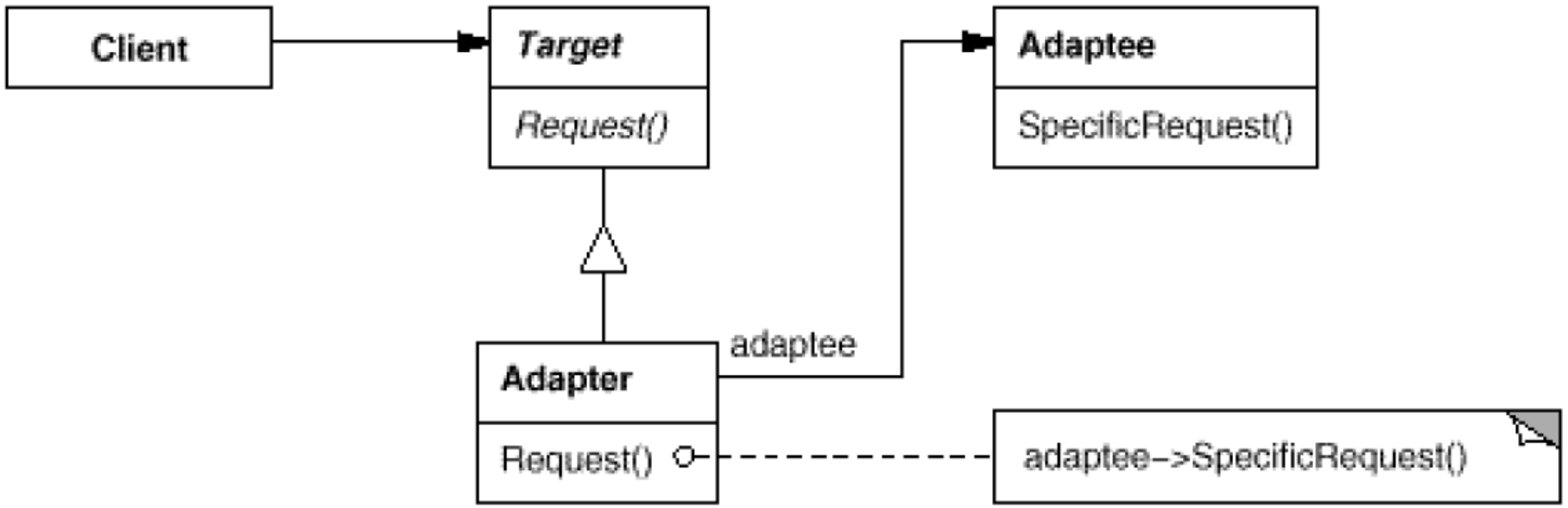
Adapter Pattern: Motivation

DrawingEditor Program



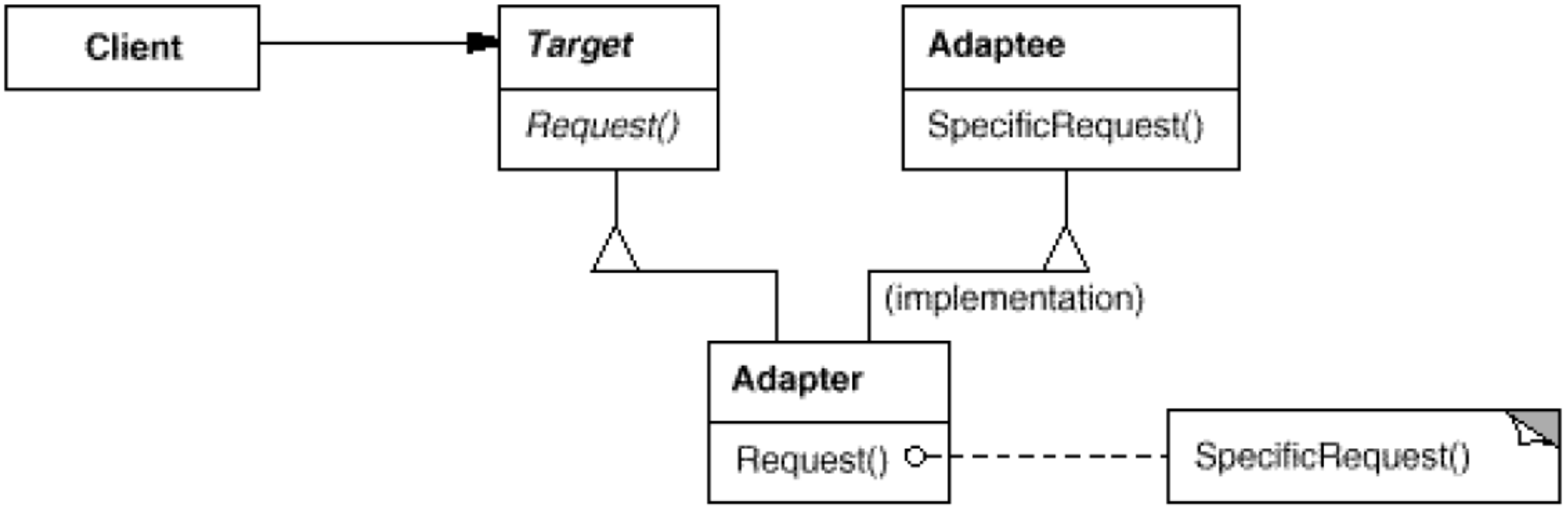
Adapter Pattern: Structure

Using Object Composition



Adapter Pattern: Structure

Using Multiple Inheritance



Adapter Pattern: Participants

- **Target** (Shape)
  - Defines the domain-specific interface that Client uses
- **Client** (DrawingEditor)
  - Collaborates with objects conforming to the Target interface
- **Adaptee** (TextView)
  - Defines an existing interface that needs adapting
- **Adapter** (TextShape)
  - Adapts the interface of Adaptee to the Target interface

Adapter Pattern: Sample Code

```
class Shape {
public:
    Shape();
    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual Manipulator* CreateManipulator() const;
};

class TextView {
public:
    TextView();
    void GetOrigin(Coord& x, Coord& y) const;
    void GetExtent(Coord& width, Coord& height) const;
    virtual bool IsEmpty() const;
};
```

Adapter Pattern: Sample Code

Using Multiple Inheritance

```
class TextShape : public Shape, private TextView {
public:
    TextShape();

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
};
```

Adapter Pattern: Sample Code

Using Multiple Inheritance

```
void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

    GetOrigin(bottom, left);
    GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}

bool TextShape::IsEmpty () const {
    return TextView::IsEmpty();
}

Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

Adapter Pattern: Sample Code

Using Object Composition

```
class TextShape : public Shape {
public:
    TextShape(TextView*);

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
private:
    TextView* _text;
};
```

Adapter Pattern: Sample Code

Using Object Composition

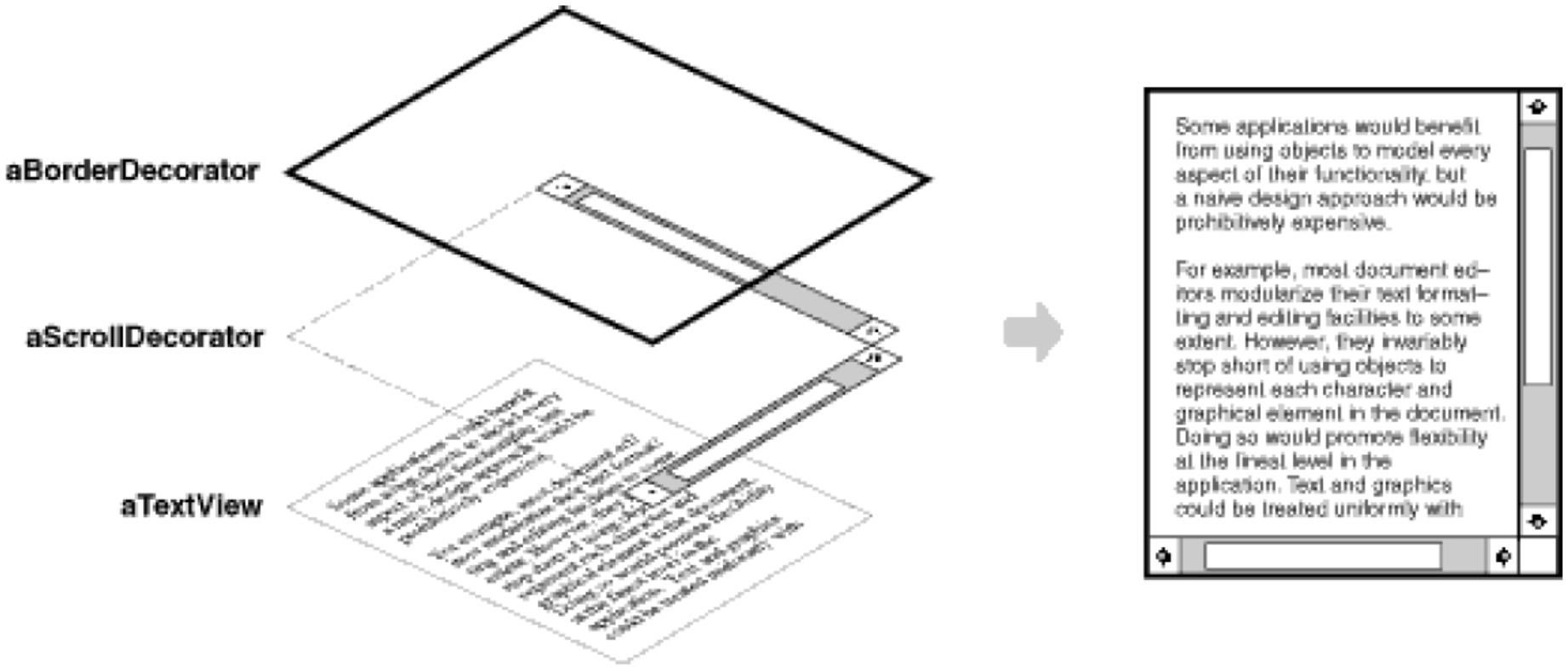
```
TextShape::TextShape (TextView* t) {
    _text = t;
}

void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

    _text->GetOrigin(bottom, left);
    _text->GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}

bool TextShape::IsEmpty () const {
    return _text->IsEmpty();
}
```

Decorator Pattern

- Attach additional responsibilities to an object dynamically
- Also known as "wrapper"
- Motivation
  - Often want to add responsibilities to individual objects, not an entire class
  - Class inheritance can be inflexible
- Solution
  - Enclose the target object in another object that adds the functionality
  - The enclosing object is called a **decorator**
  - The decorator forwards requests to the target and may perform additional actions before or after forwarding

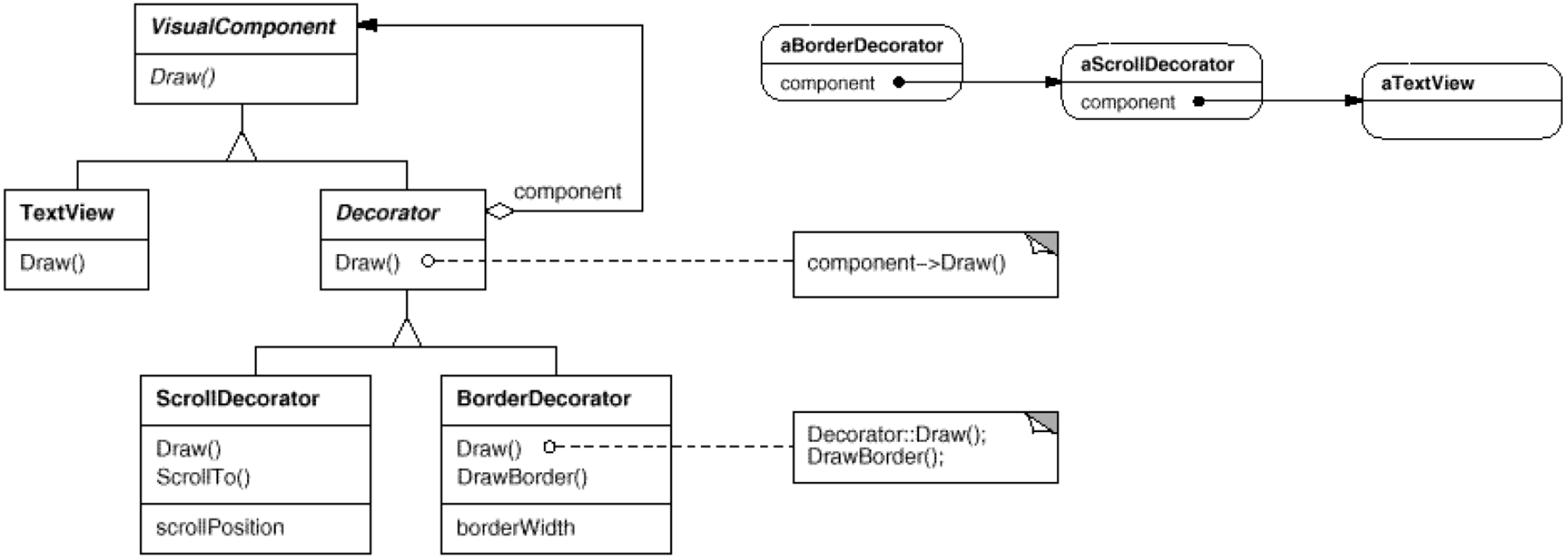
Decorator Pattern: Example

aTextView with aScrollDecorator and aBorderDecorator

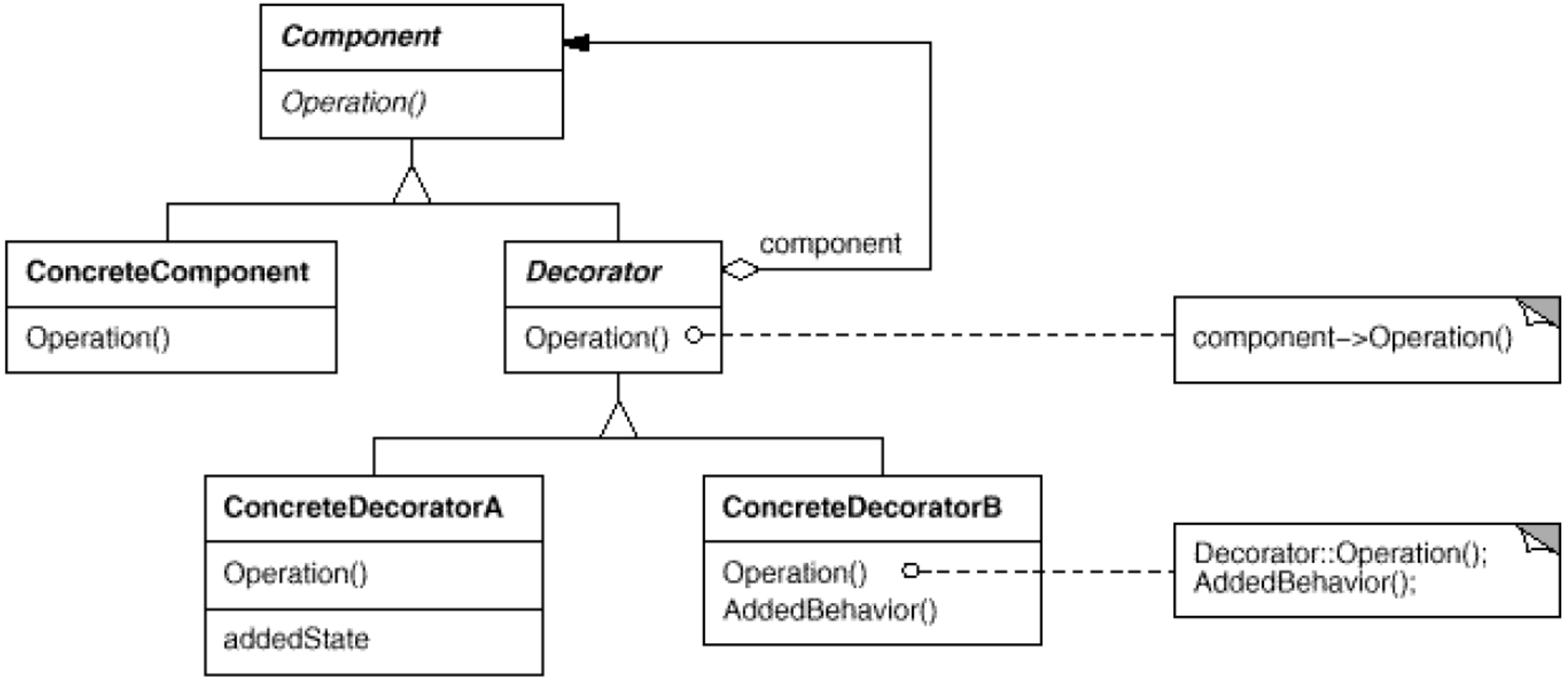


Decorator Pattern: Example

aTextView with aScrollDecorator and aBorderDecorator



Decorator Pattern: Structure



Decorator Pattern: Participants

- **Component** (VisualComponent)
  - Defines the interface for objects that can have responsibilities added to them dynamically
- **ConcreteComponent** (TextView)
  - Defines an object to which additional responsibilities can be attached
- **Decorator**
  - Maintains a reference to a Component object and defines an interface that conforms to Component's interface
- **ConcreteDecorator** (BorderDecorator, ScrollDecorator)
  - Adds responsibilities to the component

Decorator Pattern: Sample Code

```
class VisualComponent {
public:
    VisualComponent();

    virtual void Draw();
    virtual void Resize();
    // ...
};
```

- VisualComponent is the Component abstract base class

Decorator Pattern: Sample Code

```
class Decorator : public VisualComponent {
public:
    Decorator(VisualComponent*);

    virtual void Draw();
    virtual void Resize();
    // ...
private:
    VisualComponent* _component;
};
```

```
void Decorator::Draw () {
    _component->Draw();
}

void Decorator::Resize () {
    _component->Resize();
}
```

- Decorator subclass is used as a base class for creating new decorations
- For operations declared in the VisualComponent interface, Decorator passes the request to the _component object

Decorator Pattern: Sample Code

```
class BorderDecorator : public Decorator {
public:
    BorderDecorator(VisualComponent*, int borderWidth);

    virtual void Draw();
private:
    void DrawBorder(int);
private:
    int _width;
};

void BorderDecorator::Draw () {
    Decorator::Draw();
    DrawBorder(_width);
}
```

- Subclasses of Decorator define specific decorations

Decorator Pattern: Sample Code

```
void Window::SetContents (VisualComponent* contents) {
    // ...
}

Window* window = new Window;
TextView* textView = new TextView;
window->SetContents(textView);
```

- Placing a TextView into the window with no decoration

Decorator Pattern: Sample Code

```
window->SetContents(
    new BorderDecorator(
        new ScrollDecorator(textView), 1
    )
);
```

- Adding a TextView with border and scrolling capabilities

Decorator Pattern: Consequences

Benefits and Liabilities

- Benefits
  - More flexibility than static inheritance
    - Responsiblities added / removed at run-time
    - Inheritance requires a new class for each additional responsiblity
  - Avoids feature-laden classes high up in the hierarchy
- Liabilities
  - A decorator and its component are not identical
    - Complicates testing for identity
  - Lots of little objects