Software Design Patterns

Greg Butler

Computer Science and Software Engineering
Concordia University, Montreal, Canada

Email: gregb@cs.concordia.ca

Software Design Patterns

"Gang of Four" Book 1994

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

What is a Design Pattern

A *design pattern* describes

a commonly-recurring structure of communicating components

that solves

a general design problem within a particular context.

What is a Design Pattern

An example of *"best practice"* in OO design.

What is a Design Pattern

Design patterns are a *"vocabulary"* for designers

to better communicate design ideas.

Description of a Design Pattern

Essentials for Design Pattern Description

problem that the pattern addresses

context in which the pattern is used

the suggested solution

consequences of choosing that solution, e.g. strength and weakness

Template for Gang of Four Book

Name, Motivation, Context, Problem

Pattern Name (Scope, Purpose)
Intent
Also Known As
Motivation
Applicability

Solution

Structure
Participants
Collaborations
Consequences
Implementation

Other

Sample Code and Usage
Known Uses
Related Patterns

Name, Motivation, Context, Problem

Pattern Name (Scope, Purpose)
The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.

Intent
A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Also Known As
Other well-known names for the pattern, if any.

Motivation
A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

Applicability
What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

Solution

Structure

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

Collaborations
How the participants collaborate to carry out their responsibilities.

Consequences
How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

Implementation
What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

Other

Sample Code and Usage
Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

Known Uses
Examples of the pattern found in real systems. We include at least two examples from different domains.

Related Patterns
What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

Organization of Design Patterns

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

WhatVaries

Design Pattern	What Can Vary
Abstract Factory	Families of product objects
Adapter	Interface to an object
Bridge	Implementation of an object
Builder	How a composite object gets created
Chain fo Responsibility	Object that can fill a request
Command	When and how a request is fulfilled
Composite	Structure and composition of an object
Decorator	Responsibilities of an object without subclassing
Facade	Interface to a subsystem
Factory Method	Subclass of object that is instantiated
Flyweight	Storage cost of objects
Interpreter	Grammar and interpretation of a language
Iterator	How an aggregate's elements are accessed, traversed
Mediator	How and which objects interact with each other
Memento	What priavte information is stored outside an object, and when
Observer	Number of objects that depend on another object; how the dependent objects stay up to date
Prototype	Class of object that is instantiated
Proxy	How an object is accessed; its location
Singleton	The sole instance of a class
State	States of an Object
Strategy	An algorithm
Template	Steps of an algorithm
Visitor	Operations that can be applied to objects(s) without changing their class(es)

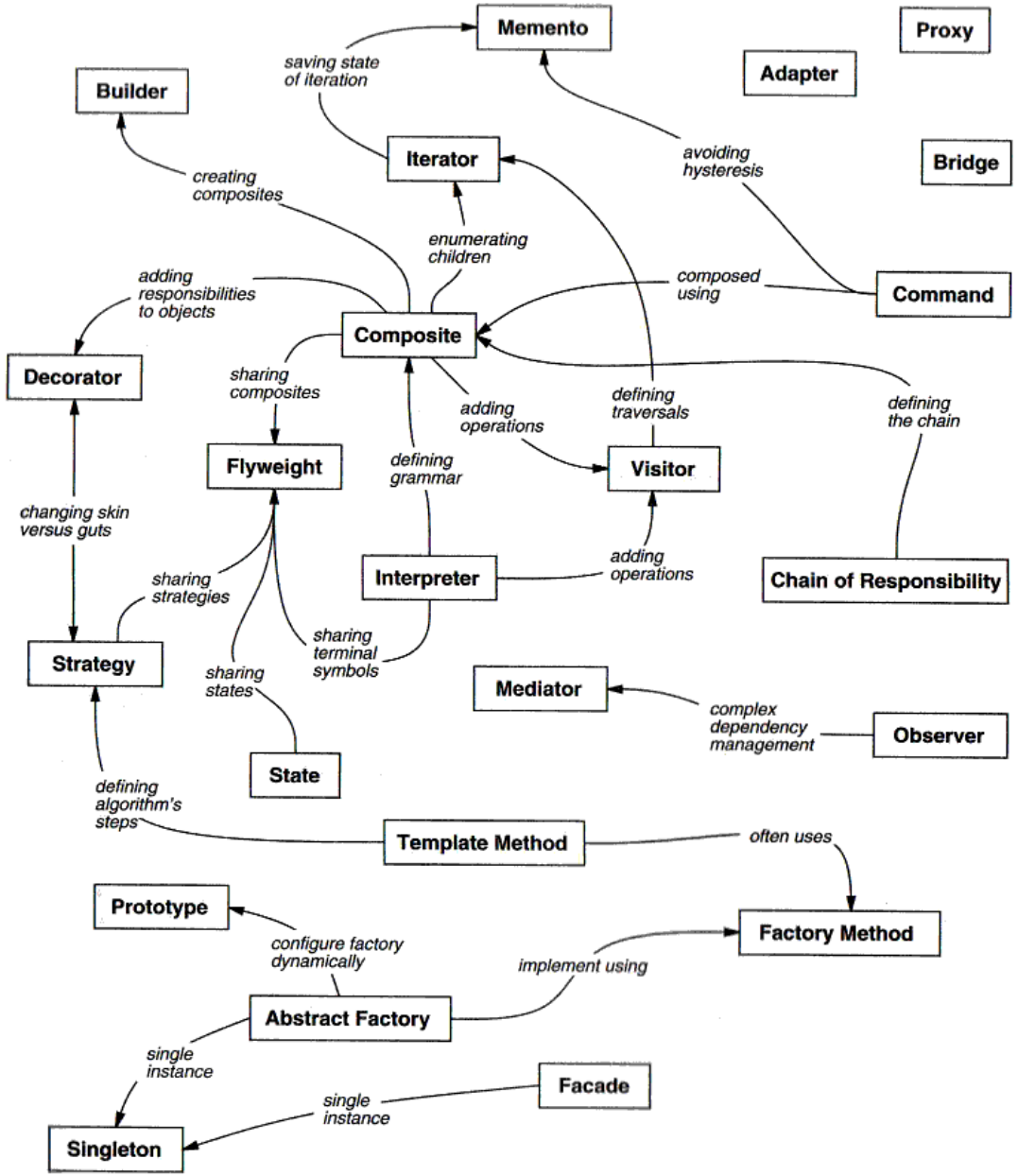
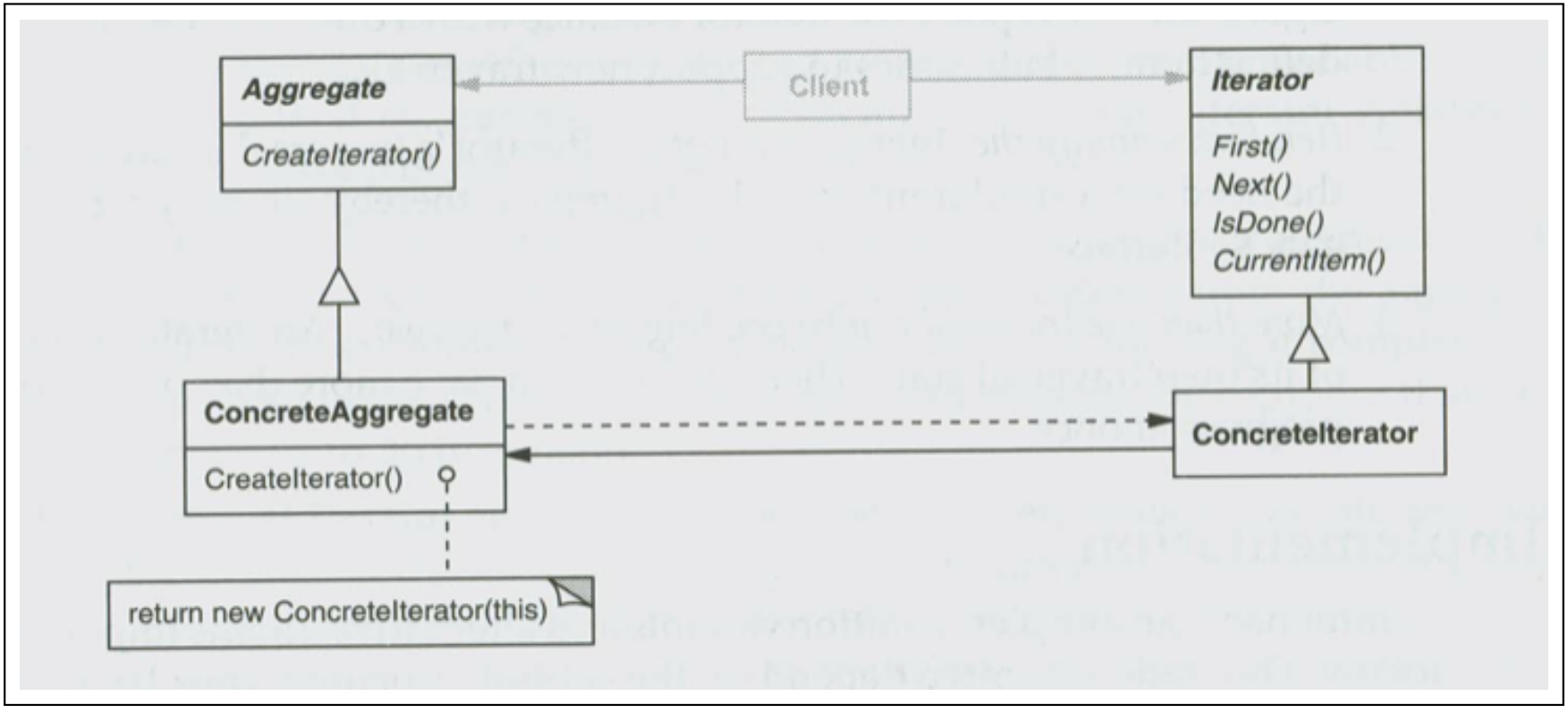


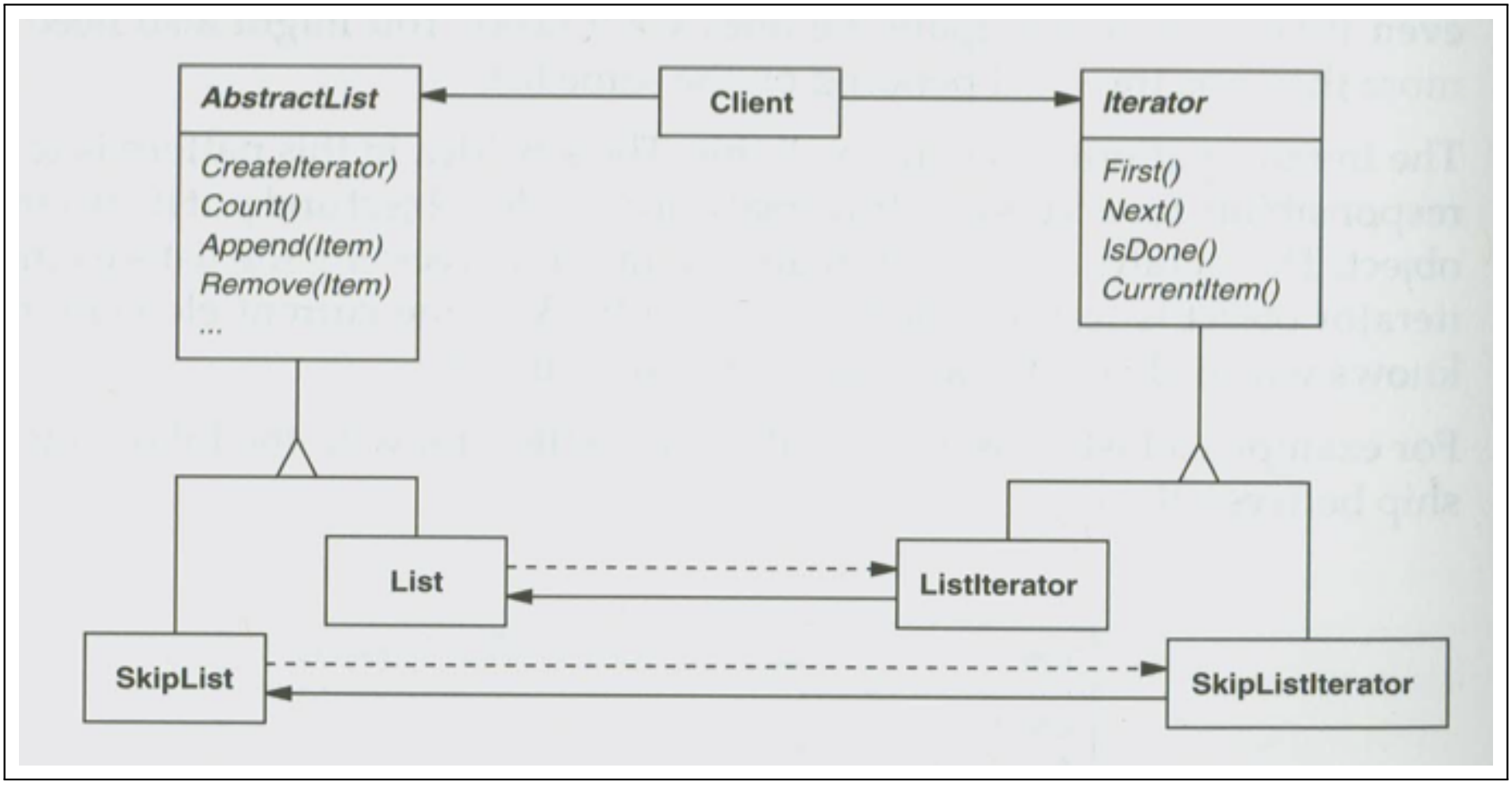
Figure 1.1: Design pattern relationships

Iterator Pattern

Intent: An Iterator provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

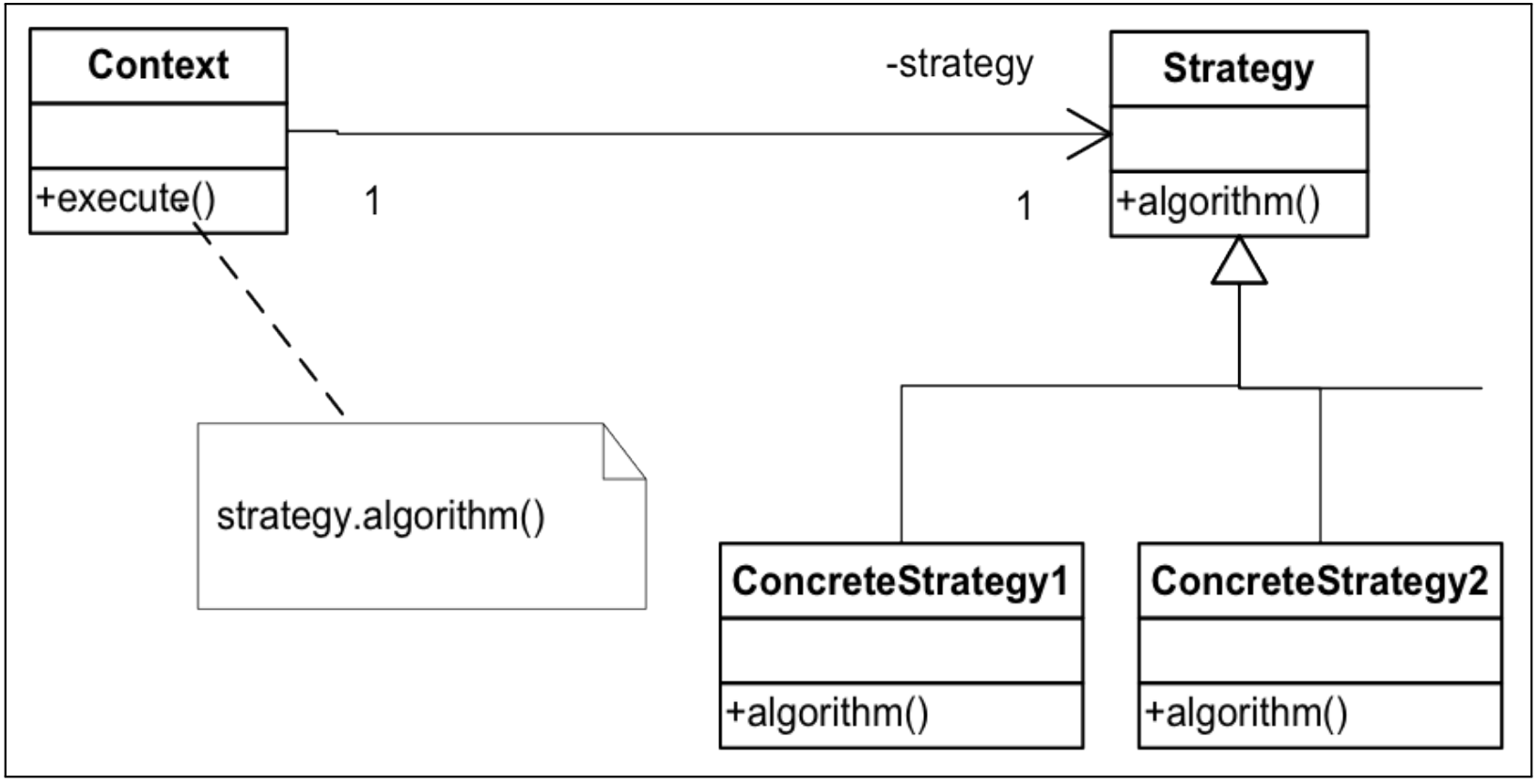


Example — Iterator Pattern

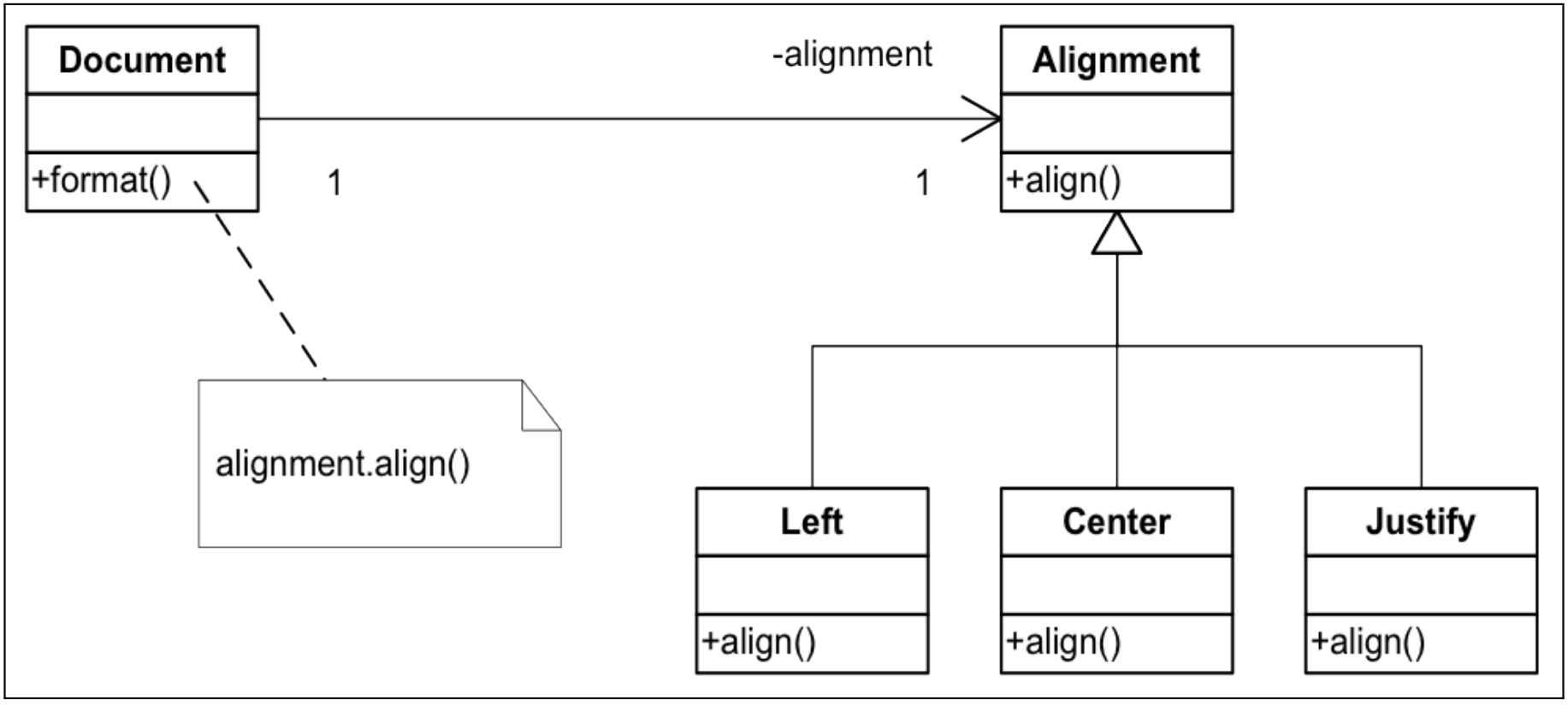


Strategy Pattern

Intent: A strategy object encapsulates an algorithm, so that the algorithm can vary independently from the clients that use it.

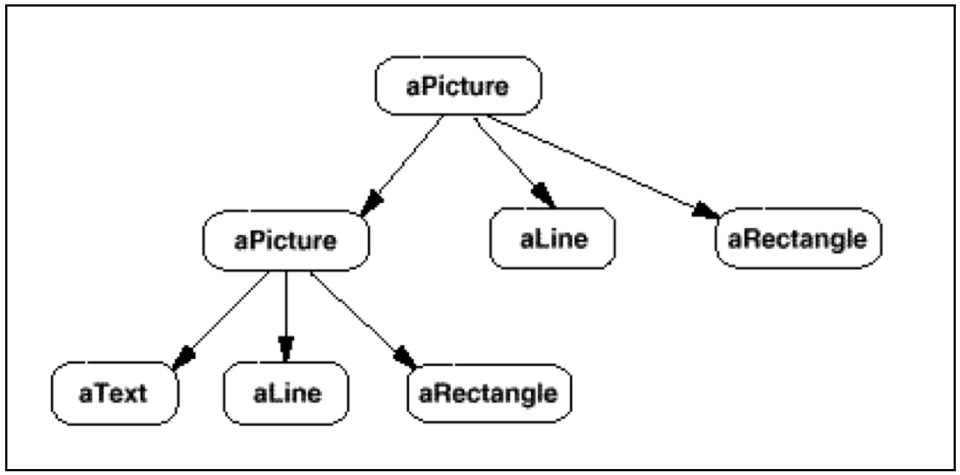


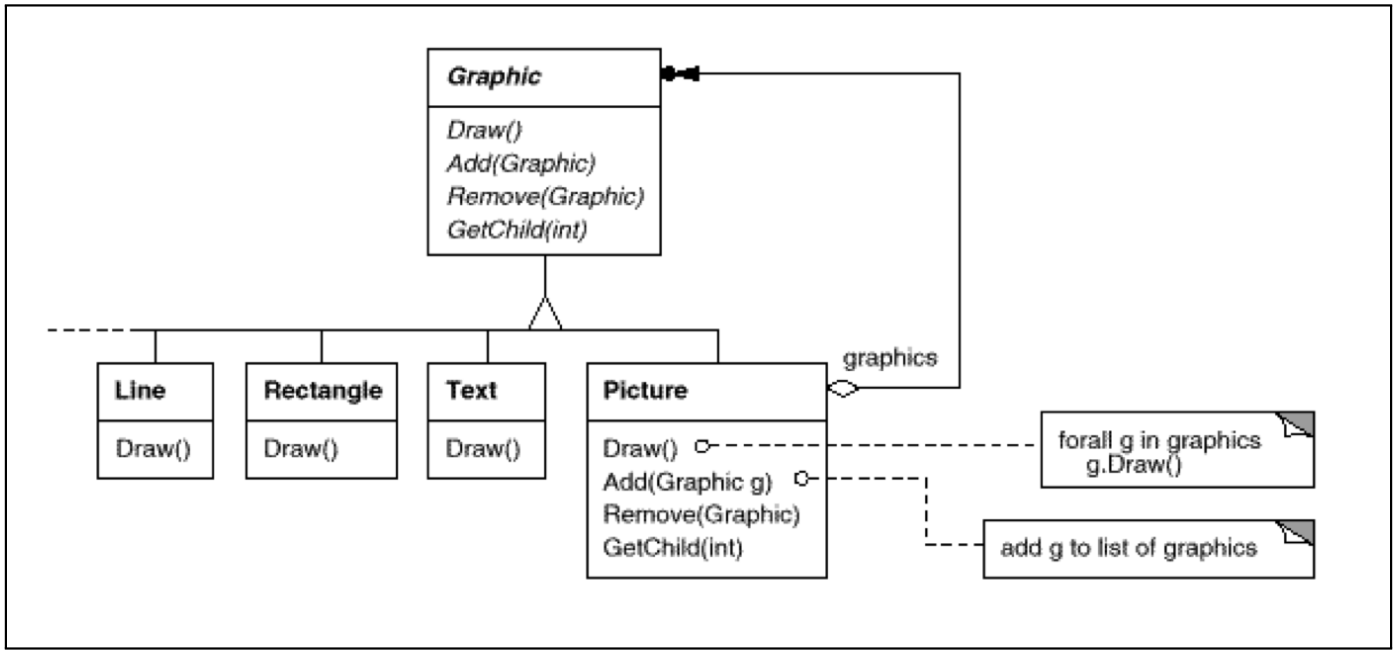
Strategy Pattern — Example



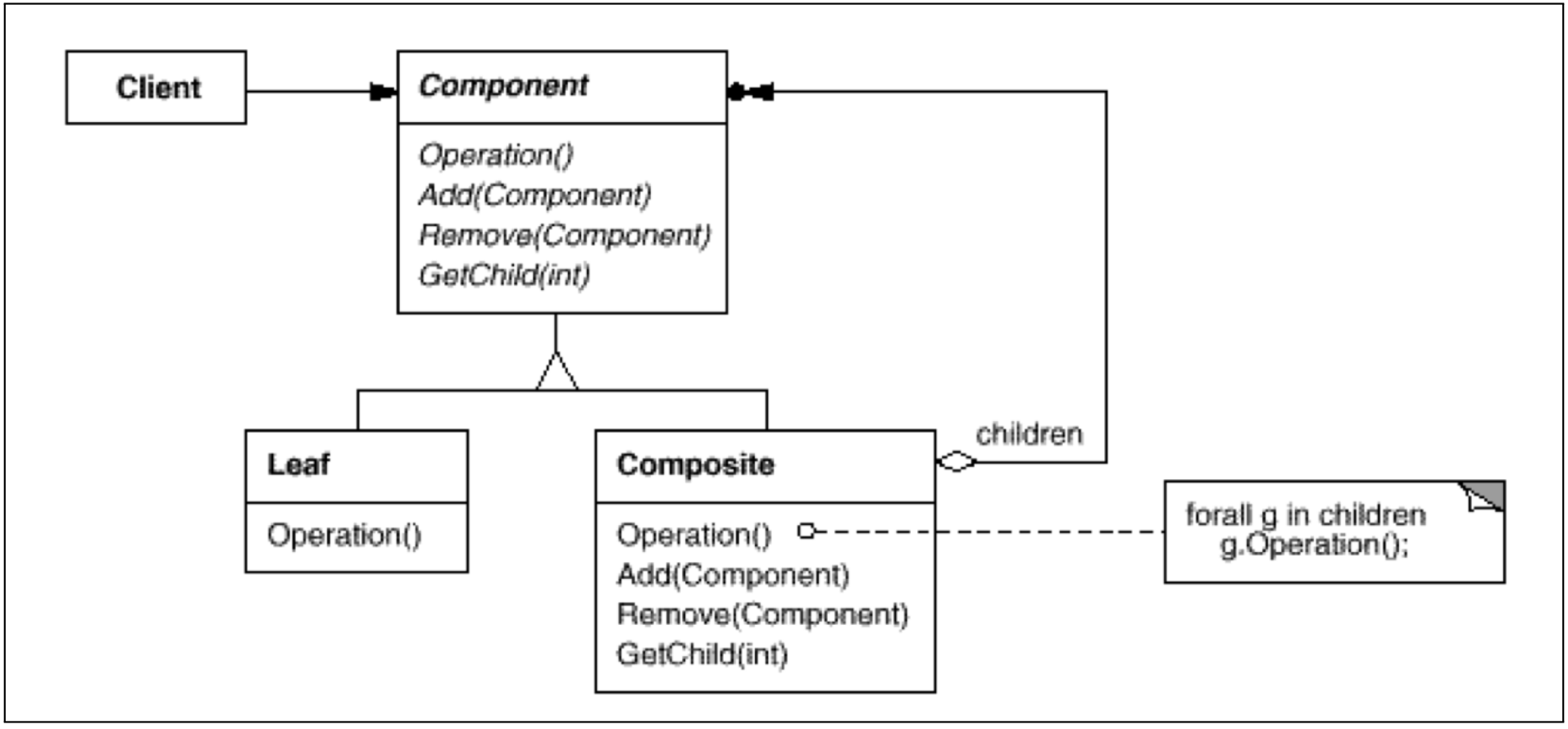
Composite Pattern

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



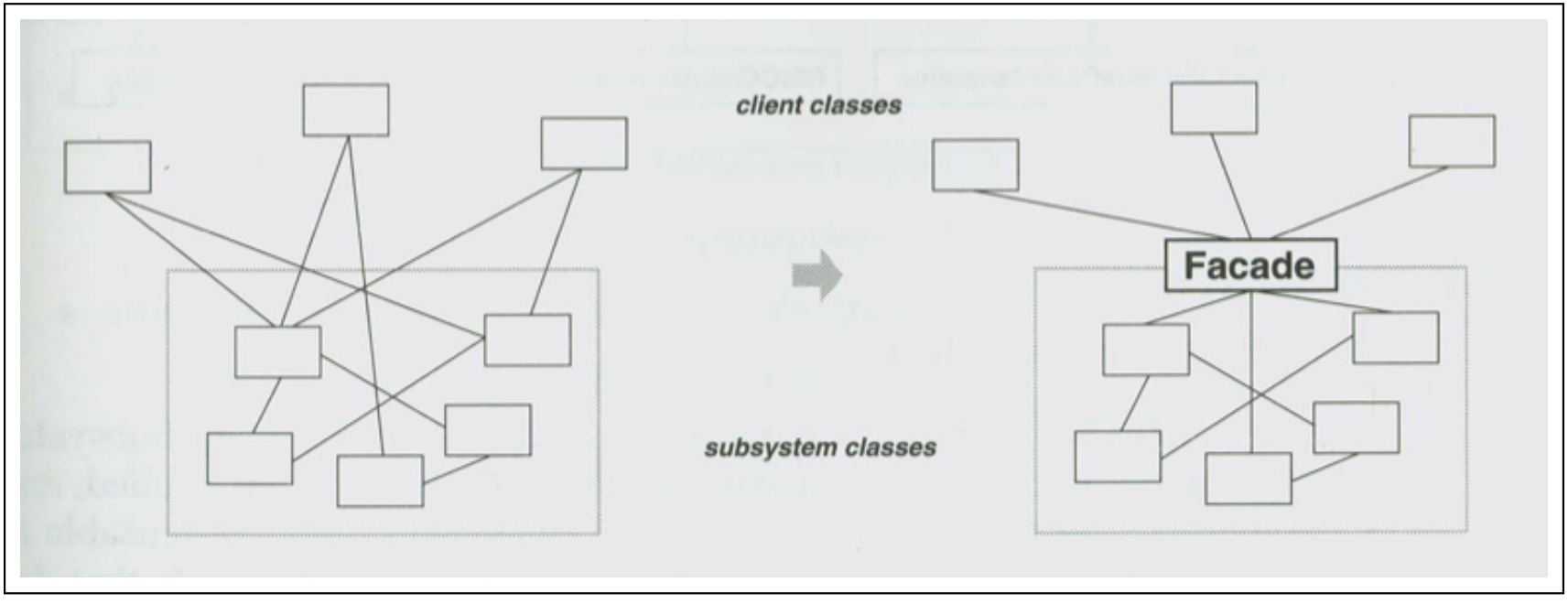


Structure — Composite Pattern



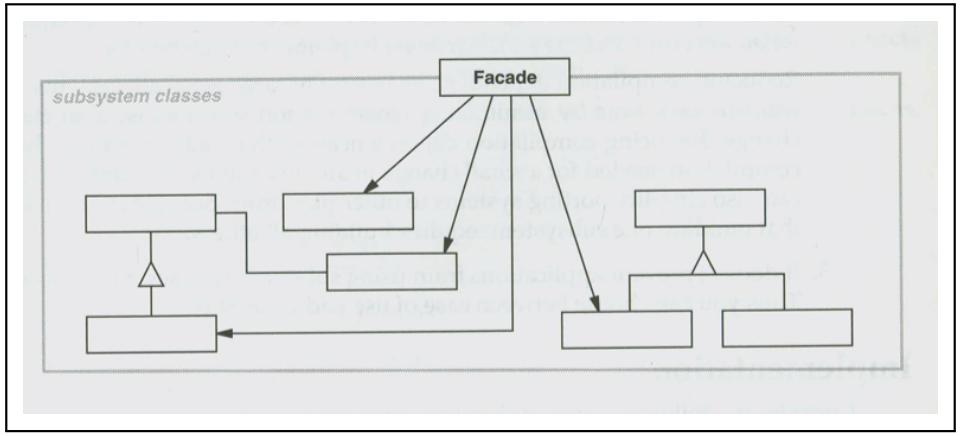
Facade Pattern

Intent: A facade provides a uniform interface to a set of interfaces in a subsystem. The Facade defines a higher-level interface that makes the subsystem easier to use.

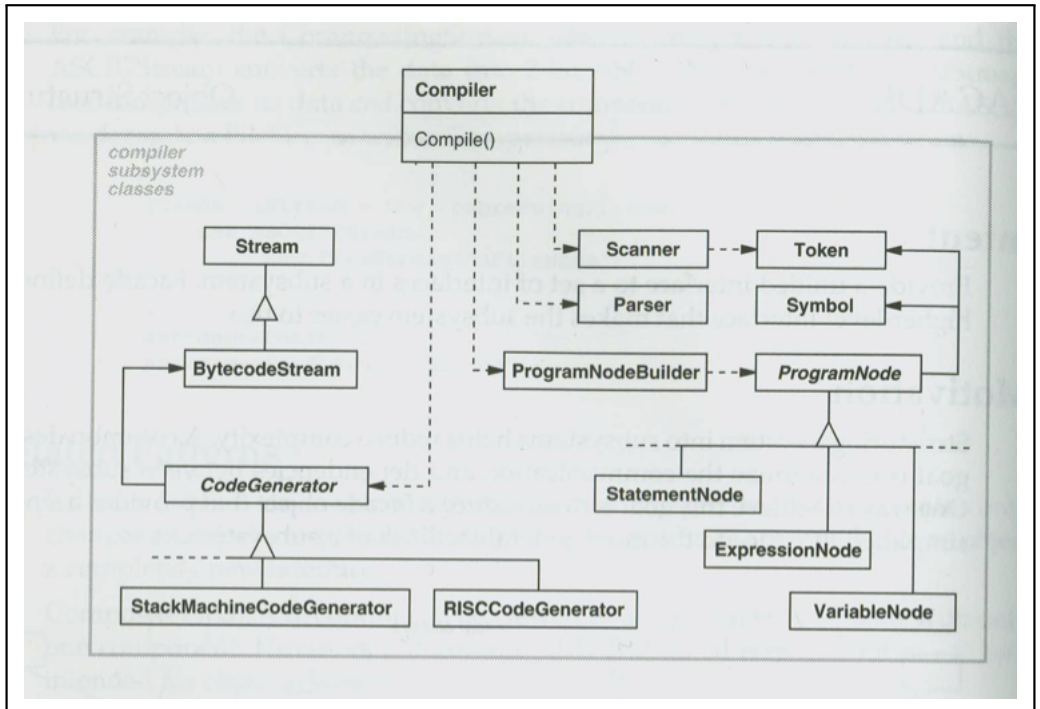


Facade Pattern

Structure

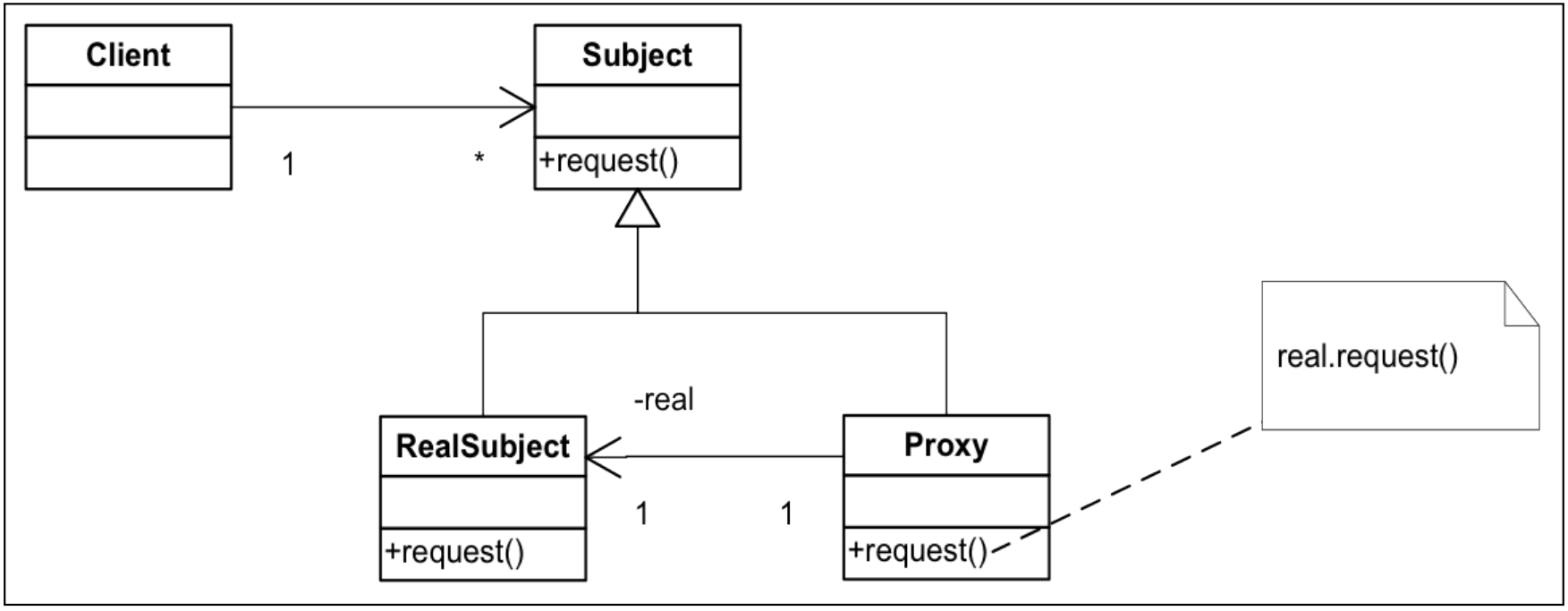


Compiler Example

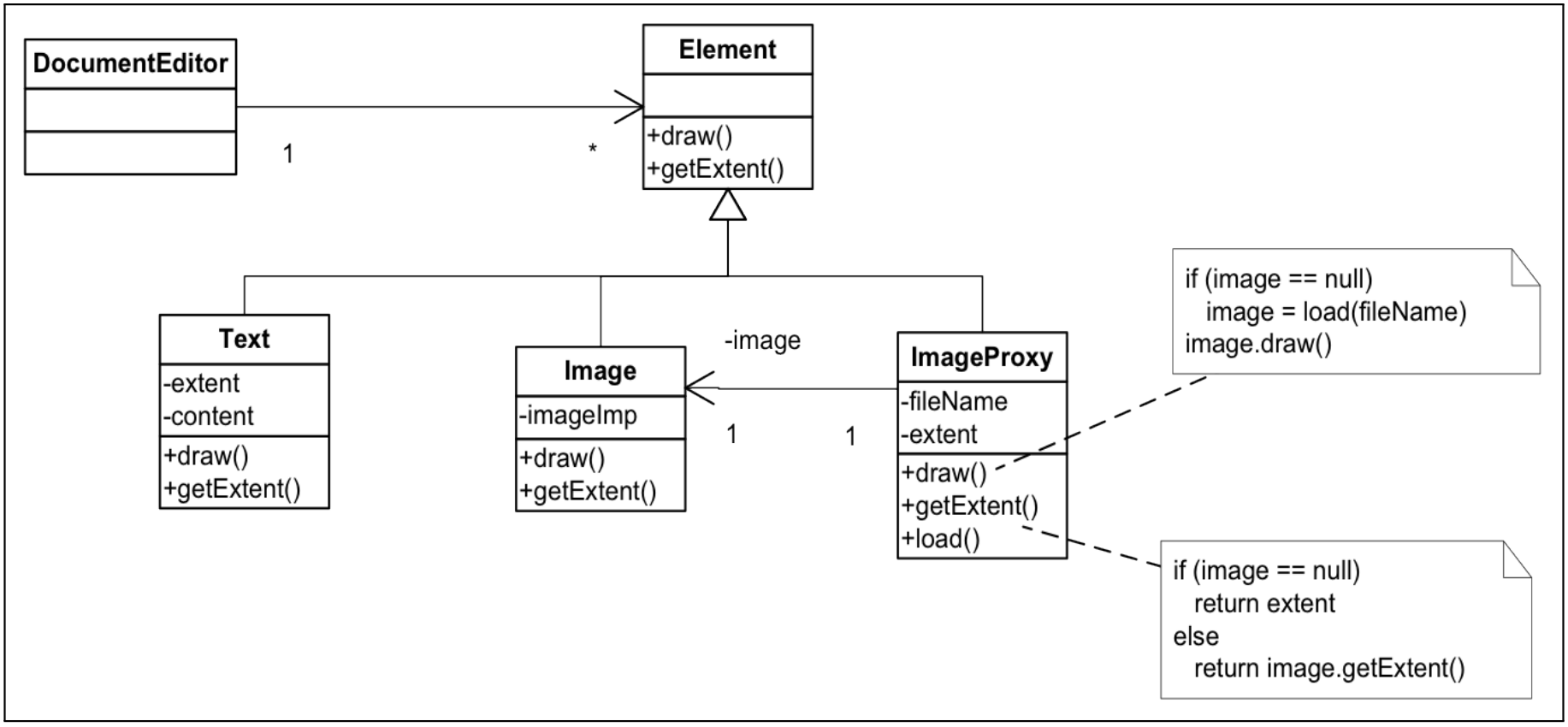


Proxy Pattern

Intent: A proxy is an object used as a substitute or placeholder for an object in order to control access to it.

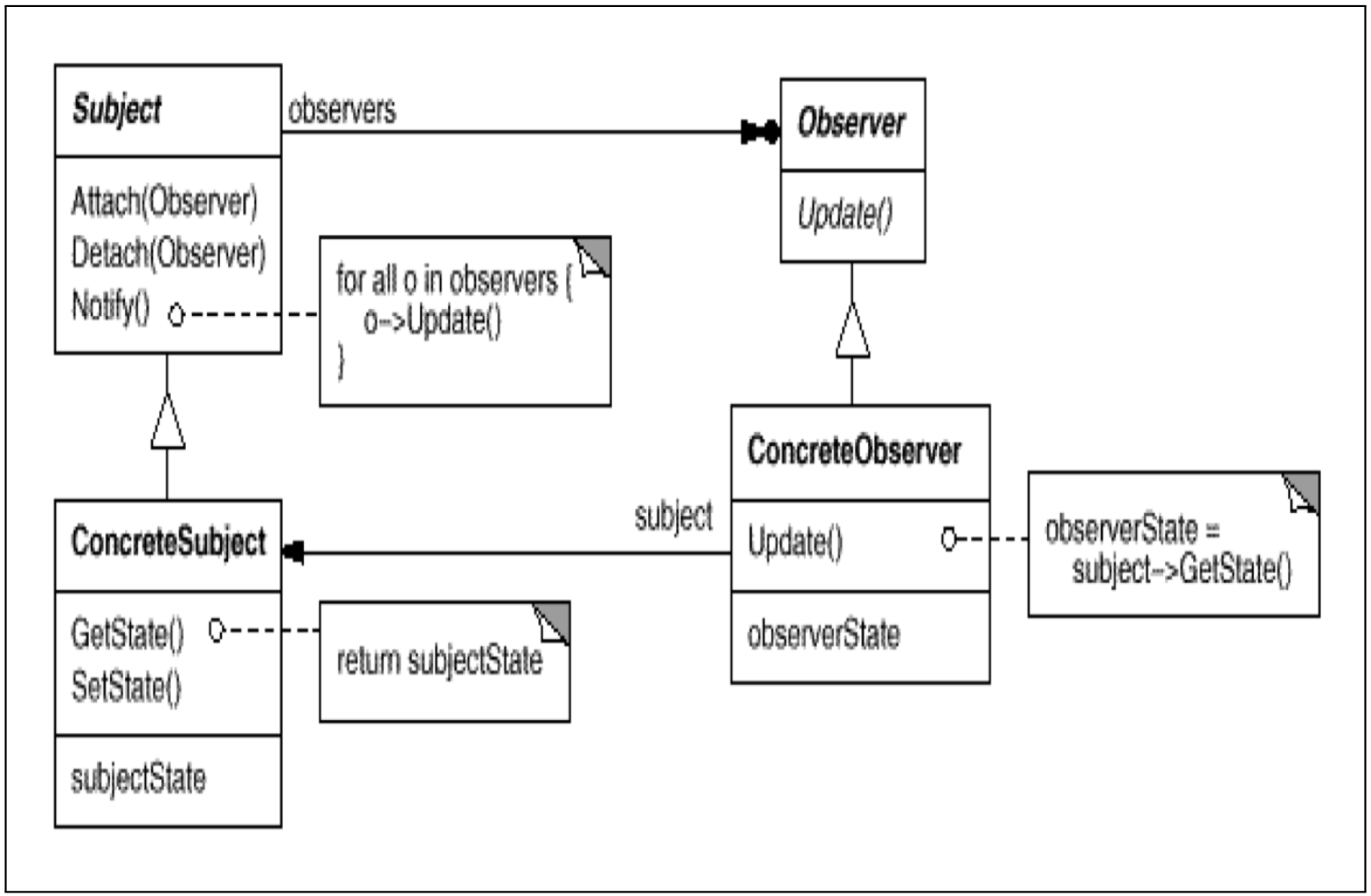


Proxy Pattern — Example



Observer Pattern

Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
that is, a subscription service, or event notification service.



Command Pattern

Intent: A Command object encapsulates a request as an object, thereby allowing you to parameterize clients with different requests, queue or log requests, and support undoable operations.