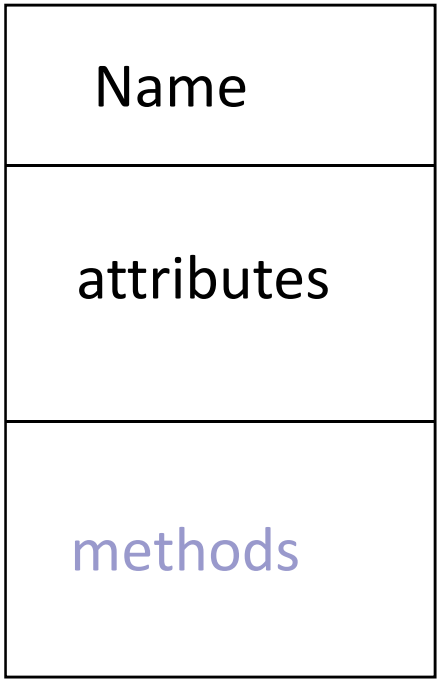
Design Patterns…

Exactly 3 slides on objects, notation, diagrammatic representation

Object Interface

Class diagram



Every method in the class has its own unique **signature**

return_type name (parameter types)

The set of all signatures defined by an object's methods or operations is called the object interface.

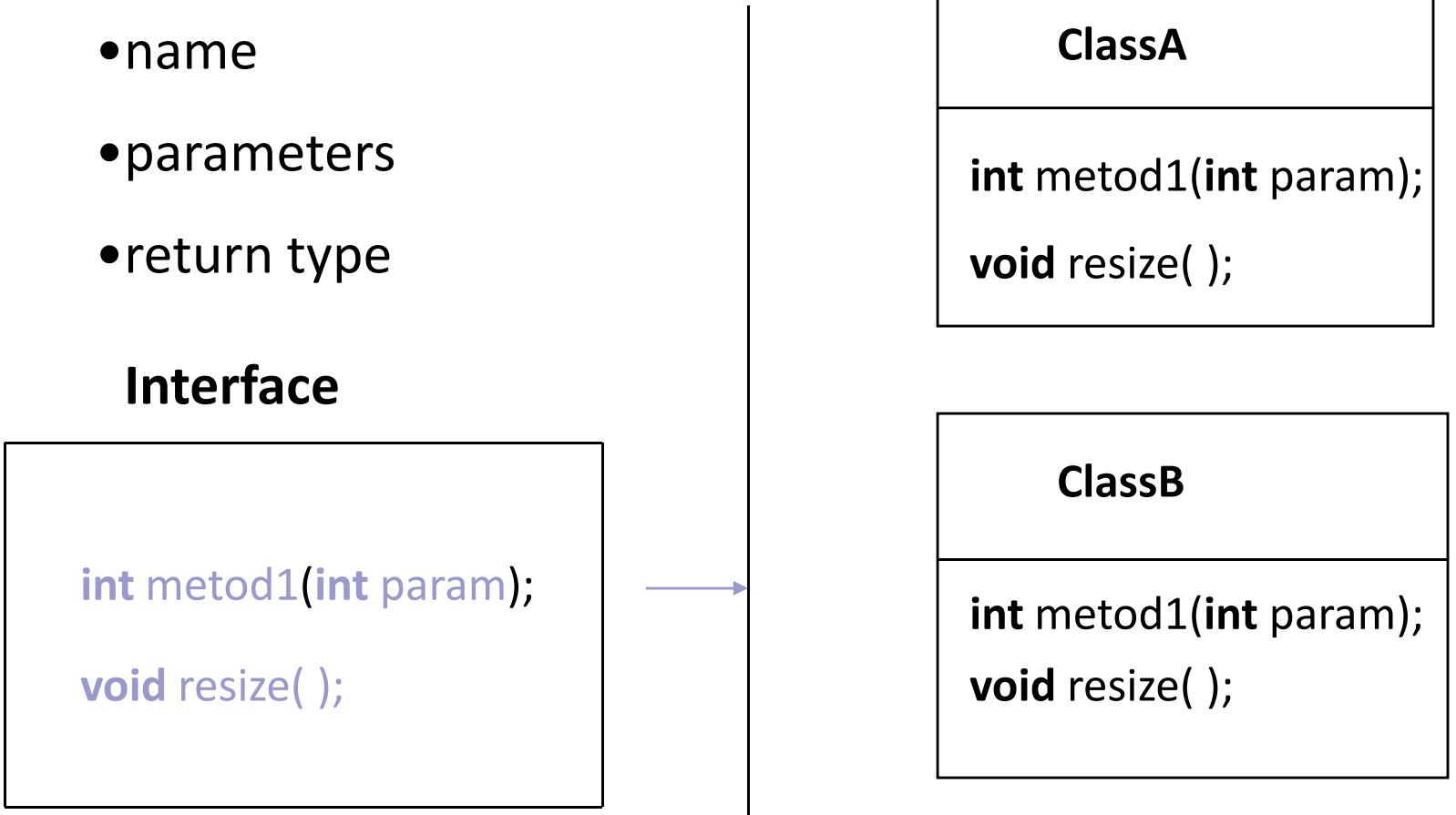
This interface characterizes the complete set of requests that can be sent to the object.

Specifying Object Interfaces

Signature of a method:

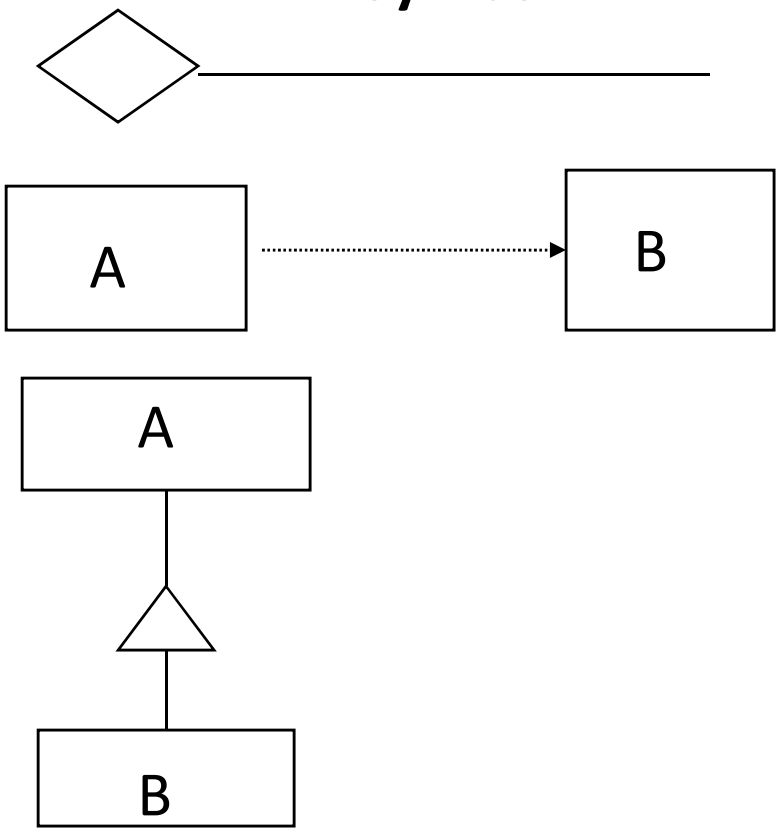
- name
- parameters
- return type

Interface



Possible message recipients

Notation


Symbol
A
B
A
B
ClassName
class method
Meaning
Aggregation
Object A instantiates B
B is a subclass of A
Comment box
Implementation pseudocode

Few Design Principles

- **Principle of least astonishment**

  (don’t be astonishing).
- **Make common things easy, and rare things possible**
- **Consistency**

  The more random rules you pile onto the programmer, rules that have nothing to do with solving the problem at hand, the slower the programmer can produce. And this does not appear to be a linear factor, but an exponential one.

- **Law of Demeter:** a.k.a. “Don’t talk to strangers.”

  An object should only reference itself, its attributes, and the arguments of its methods.
- **Subtraction:**

  A design is finished when you cannot take anything else away.
- **Simplicity before generality:**

  (A variation of *Occam’s Razor*, which says “the simplest solution is the best”). A common problem we find in frameworks is that they are designed to be general purpose without reference to actual systems. This leads to a dizzying array of options that are often unused, misused or just not useful. However, most developers work on specific systems, and the quest for generality does not always serve them well. The best route to generality is through understanding well-defined specific examples. So, this principle acts as the tie breaker between otherwise equally viable design alternatives. Of course, it is entirely possible that the simpler solution is the more general one.

- **Reflexivity**

  One abstraction per class, one class per abstraction. Might also be called **Isomorphism**.
- **Independence** or **Orthogonality**

  Express independent ideas independently. This complements Separation, Encapsulation and Variation, and is part of the Low-Coupling-High-Cohesion message.
- **Once and once only**

  Avoid duplication of logic and structure where the duplication is not accidental, ie where both pieces of code express the same intent for the same reason.

Few OO Principles

- Encapsulate what varies
- Favor composition over inheritance
- Program to interfaces, not to implementations
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension but closed for modification
- Depend on abstractions. Do not depend on concrete classes

Patterns v/s Frameworks v/s Toolkit

- A *toolkit* is a library of reusable classes designed to provide useful, general-purpose functionality.
  - E.g. Routine util classes
- An *application framework* is a specific set of classes that cooperate closely with each other and together embody a reusable design for a category of problems.
  - E.g., MFC, JFC, etc.
- A *design pattern* describes a general recurring problem in different domains, a solution, when to apply the solution, and its consequences

- A *framework* embodies a complete design of an application, while a *pattern* is an outline of a solution to a class of problems.
- A *framework* dictates the architecture of an application and can be customized to get an application. (E.g., .Net Web Services, *Java Applets*)
- When one uses a *framework*, one reuses the main body of the framework and writes the code it calls.
- When one uses a *toolkit*, one writes the main body of the application that calls the code in the toolkit

Definition

- … a fully realized form, original, or model accepted or proposed for imitation…[dictionary]
- ... 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 [Alexander]
- … the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts [Riehle]
- …both a thing and the instructions for making the thing [Coplien]
- ...a literary format for capturing the wisdom and experience of expert designers, and communicating it to novices
- …represent solutions to problems that arise when developing software within a particular context

Patterns = Problem/Solution pair in Context

Pattern origins and history

- writings of architect Christopher Alexander
  (coined this use of the term "pattern" ca. 1977-1979)
- Kent Beck and Ward Cunningham, Textronix, OOPSLA'87
  (used Alexander's "pattern" ideas for Smalltalk GUI design)
- Erich Gamma, Ph. D. thesis, 1988-1991
- James Coplien, *Advanced C++ Idioms book*, 1989-1991
- Gamma, Helm, Johnson, Vlissides ("Gang of Four“ - GoF)
  Design Patterns: Elements of Reusable Object-Oriented Software, 1991-1994
- PLoP Conferences and books, 1994-present
- Buschmann, Meunier, Rohnert, Sommerland, Stal, *Pattern -Oriented Software Architecture: A System of Patterns* (“POSA book”)

Quality Without a Name: The Goal

… there is a central quality, which is the root criterion of life and spirit
in a man, a town, a building, or a wilderness.
This quality is objective and precise, but it cannot be named.

… the search, which we make for this quality, in our own lives,
is the central search of any person,
and the crux of any individual person's story.
It is the search of ***those moments when we are most alive****.*

C.Alexander – Timeless Way of Building

Properties

Patterns do...

- provide common vocabulary
- provide “shorthand” for effectively communicating complex principles
- help document software architecture
- capture essential parts of a design in compact form
- show more than one solution
- describe software abstractions

Patterns do not...

- provide an exact solution
- solve all design problems
- only apply for object-oriented design

Reuse Rather Than Rebuild

- Libraries are predefined classes you can reuse
  - components, like architect's windows
  - examples: **`cs015.prj, wheels, Demos.Cars`**
  - like components, no indication on *how* to use them in program

- *Patterns* are more general than *libraries*
  - specifies some relationships between classes
  - one pattern may represent many interacting classes
  - general, so must be applied to specific problem

- Patterns *name, abstract, and identify key aspects of design's structure*
  - name: Colonial Revival
  - abstract: any structure
  - key aspects: typically white or gray wood, facade shows symmetrically balanced windows centered on door; windows have double-hung sashes and panes

You've Already Seen Patterns

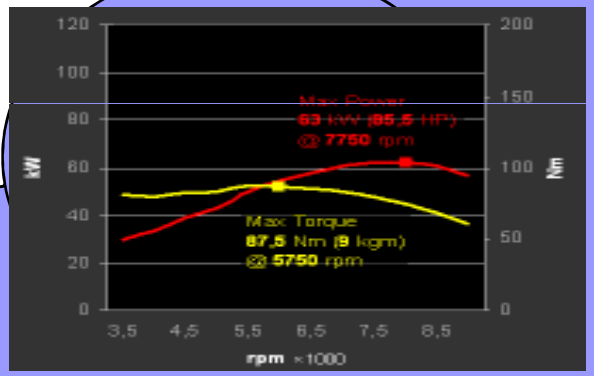
- Note *containment* pattern uses *initialization* and *encapsulation* patterns
- Constructors
  - **pattern name:** initialization
  - **abstract:** way to ensure objects have proper internal state before use
  - **key aspects:** method that first calls **base()** and then sets up instance's own instance variables
- Properties
  - **pattern name:** encapsulation
  - **abstract:** keeps other objects from changing an object's internal state improperly
  - **key aspects:** make instance variables **private** or **protected** and provide **public** accessor and mutator methods as appropriate
- Instance variables
  - **pattern name:** containment
  - **abstract:** models objects that are composed of other objects
  - **key aspect:** store components as instance variables, initialize them in the constructor and provide access protection through encapsulation

Purpose

- A design pattern captures *design expertise* –patterns are not created from thin air, but abstracted from *existing* design examples
- Using design patterns is *reuse* of design expertise
- Studying design patterns is a way of studying how the “experts” do design
- Design patterns provide a *vocabulary* for talking about design

On vocabulary

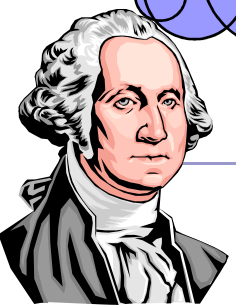
"Longitudinally-mounted 90-degree V-twin"



"Torque"

"Growl"

"Perfect primary balance"



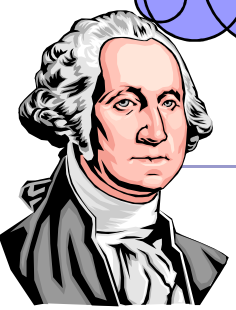
Images from http://www.ducati.com

On vocabulary (2)

"Transverse inline fo[illegible]"

"Power peak"

"Screamer"



Images from http://www.tntperformancedyno.com, http://dsr.racer.net and http://www.motorcycle.com

Why design patterns in SA?

- If you're developing/designing/managing software, you should know about them anyway
- There are many architectural patterns published, and the GoF Design Patterns is a prerequisite to understanding these:
  - Mowbray and Malveau – CORBA Design Patterns
  - Schmidt et al – Pattern-Oriented Software Architecture
- Design Patterns help you *break out* of

The seven layers of architecture*

Global architecture

Enterprise architecture

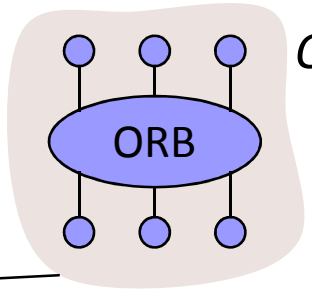
System architecture

Application architecture

Macro-architecture

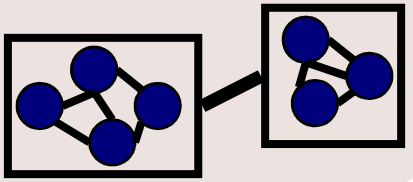
Micro-architecture

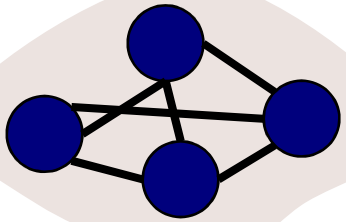
Objects



OO architecture

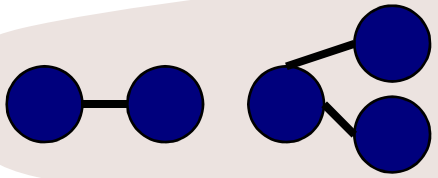
Subsystem

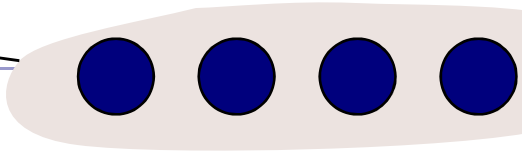




Frameworks

Design patterns

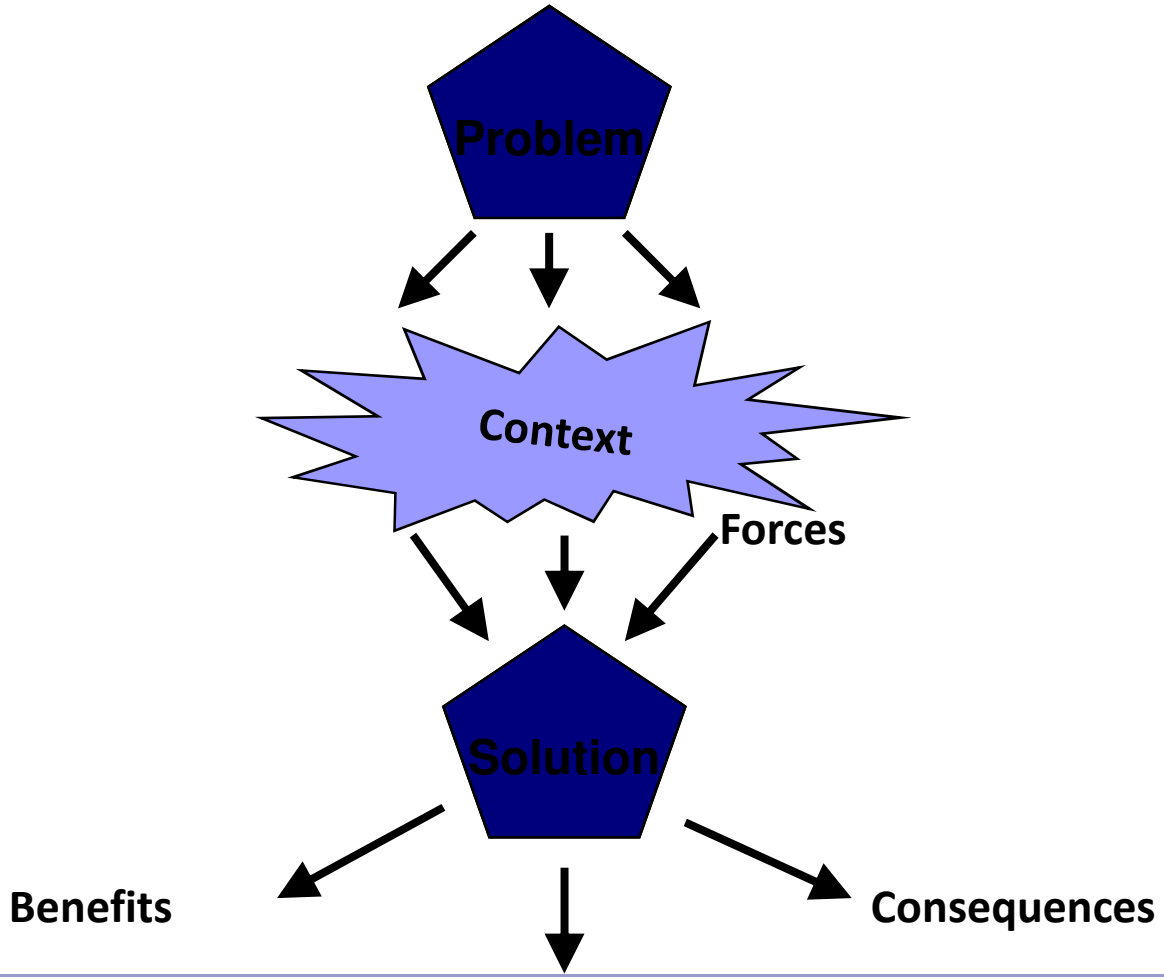




OO programming

* Mowbray and Malveau

How patterns arise



Goals

- Codify good design
  - Distil and disseminate experience
  - Aid to novices and experts alike
  - Abstract how to think about design
- Give design structures explicit names
  - Common vocabulary
  - Reduced complexity
  - Greater expressiveness
- Capture and preserve design information
  - Articulate design decisions succinctly
  - Improve documentation
- Facilitate restructuring/refactoring
  - Patterns are interrelated
  - Additional flexibility

Design patterns can be subjective.

- One person's pattern may be another person's primitive building block.

The focus of the selected design patterns are:

- Object and class communication
- Customized to solve a general design problem
- Solution is context specific

Why catalog/learn design patterns?

- Patterns help you learn from other's successes, instead of your own failures
- Provides (an application-independent) vocabulary to communicate, document, and explore design alternatives.
- Captures the experience of an expert (especially the rationale behind a design and the trade-offs involved) and codifies it in a form that is potentially "reusable".
  - *What, why, how, …*
  - *Example* is not another way to teach, it is the *only* way to teach. -- *Albert Einstein*
    - (*Cf.* Vince Lombardi Quote)

Design Patterns - Why?

- Designing OO software is hard
- Designing *reusable* OO software – harder
- Experienced OO designers make good design
- New designers tend to fall back on non-OO techniques used before
- Experienced designers know something – what is it?

Design Patterns - Why?

- Expert designers know *not* to solve every problem from first principles
- They *reuse* solutions
- These patterns make OO designs more flexible, elegant, and ultimately reusable

Analogy II – **Telenovela**!!

Analogy I - Novelists and playwrights:
"**Tragically Flawed Hero**" (Macbeth, Hamlet...)
"**The Romantic Novel**"

Key Mechanisms in Design Patterns

Class vs. Interface Inheritance

- Class – defines an implementation
- Type – defines only the interface
  - the set of requests that an object can respond to
- Relation between Class and Type
  - the class implies the type
- On class, many types. Many classes, same type
- Class Inheritance
  - one implementation in terms of another
- Type Inheritance

GoF Design Principle #1

Program to an interface, not an implementation

- Use interfaces to define common interfaces
  - and/or abstract classes in C++/C#
- Declare variables to be instances of the abstract class
  - not instances of particular classes
- Use *Creational patterns*
  - to associate interfaces with implementations

Contd…

Benefits

- Greatly *reduces the implementation dependencies*
  - Client objects remain unaware of the classes that **implement** the objects they use.
  - Clients know only about the abstract classes (or interfaces) that define the interface.

Class Inheritance vs. Composition

- Mechanisms of reuse
  - White-box vs. Black-box
- Class Inheritance
  - easy to use; easy to modify
    - implementation being reused;
  - language-supported
  - static bound ⇒ can't change at run-time;
  - mixture of physical data representation ⇒ breaks encapsulation
    - change in parent ⇒ change in subclass
- Object Composition
  - objects are accessed solely through their interfaces

Design Principle #2

Favor composition over class inheritance

- Keeps classes focused on one task
- Inheritance and Composition Work Together!
  - ideally no need to create new components to achieve reuse
  - this is rarely the case!
  - reuse by inheritance makes it easier to make new components
    - modifying old components
- Tendency to overuse inheritance as code-reuse technique
- Designs – more reusable by depending more on object composition

Types of software patterns

- design patterns **(software design) [Buschmann-POSA]**
  - architectural **(systems design)**
  - design **(micro-architectures) [Gamma-GoF]**
  - idioms **(low level)**

- analysis patterns **(recurring & reusable analysis models) [Flower]**
- organization patterns **(structure of organizations/projects)**
- process patterns **(software process design)**
- domain-specific patterns

Alexandrian form (canonical form)

Name
: meaningful name

Problem
: the statement of the problem

Context
: a situation giving rise to a problem

Forces
: a description of relevant forces and constraints

Solution
: proven solution to the problem

Examples
: sample applications of the pattern

Resulting context (force resolution)
: the state of the system after pattern has been applied

Rationale
: explanation of steps or rules in the pattern

Related patterns
: static and dynamic relationship

Known use
: occurrence of the pattern and its application within existing system

GoF format

Pattern name and classification

Intent

what does pattern do / when the solution works

Also known as

other known names of pattern (if any)

Motivation

the design problem / how class and object structures solve the problem

Applicability

situations where pattern can be applied

Structure

a graphical representation of classes in the pattern

Participants

the classes/objects participating and their responsibilities

Collaborations

of the participants to carry out responsibilities

GoF format

Consequences
trade-offs, concerns

Implementation
hints, techniques

Sample code
code fragment showing possible implementation

Known uses
patterns found in real systems

Related patterns
closely related patterns

Pattern templates

[PATTERN-NAME]
Author
[YOUR-NAME] ([YOU@YOUR.ADDR]).
Last updated on [TODAY'S-DATE]
Context
[PARAG-1]
[PARAG-2]
Problem
[ONE-ASPECT]
[ANOTHER-ASPECT]
Examples
Forces
1.[FORCE-1]
2.[FORCE-2]
Design
[SOME-DESIGN]
[PARAG-2]
An Implementation
[SOME-CODE]
Examples
Variants
[VARIANT]
[ANOTHER-VARIANT]
See Also
[ANOTHER-REF]

IF you find yourself in **CONTEXT**
for example **EXAMPLES**,
with **PROBLEM**,
entailing **FORCES**
THEN for some **REASONS**,
apply **DESIGN FORM AND/OR RULE**
to construct **SOLUTION**
leading to **NEW CONTEXT and OTHER PATTERNS**

http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html

http://hillside.net/patterns/Writing/Lea.html

More pattern templates:

- http://hillside.net/patterns/template.html
- http://www.paterndepot.com/pages (Templates)

Pattern language

[Coplien]

- …is a structured collection of patterns that build on each other to transform needs and constraints into an architecture
  [Software Design Patterns: Common Questions and Answers]

- …defines collection of patterns and rules to combine them into an architectural style…describe software frameworks or

Pattern catalogs and systems

[Buschmann, POSA]

- pattern catalog

  …a collection of related patterns, where patterns are subdivided into small number of broad categories…

- pattern system

  …a cohesive set of related patterns, which work together to support the construction and evolution of whole architectures

Classification of Design Patterns

- Creational Patterns
  - deal with initializing and configuring classes and objects
  - *how am I going to create my objects?*
- Structural Patterns
  - deal with decoupling the interface and implementation of classes and objects
  - *how classes and objects are composed to build larger structures*
- Behavioral Patterns
  - deal with dynamic interactions among societies of classes and objects
  - *how to manage complex control flows*

Design pattern catalog - GoF

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	• **Factory Method**	• **Adapter**	• **Interperter**
	Object	• **Abstract Factory** • **Builder** • **Prototype** • **Singleton**	• **Adapter** • **Bridge** • **Composite** • **Decorator** • **Facade** • **Flyweight** • **Proxy**	• **Chain of Responsibility** • **Command** • **Iterator** • **Mediator** • **Momento** • **Observer** • **State** • **Strategy** • **Vistor**

Design Pattern Elements

1. Pattern Name

   Handle used to describe the design problem

   Increases vocabulary

   Eases design discussions

   Evaluation without implementation details

Design Pattern Elements

2. Problem

   Describes when to apply a pattern

   May include conditions for the pattern to be applicable

   Symptoms of an inflexible design or limitation

Design Pattern Elements

3. Solution

   Describes elements for the design

   Includes relationships, responsibilities, and collaborations

   Does not describe concrete designs or implementations

   A pattern is more of a template

Design Pattern Elements

4. Consequences

   Results and Trade Offs

   Critical for design pattern evaluation

   Often space and time trade offs

   Language strengths and limitations

   (Broken into benefits and drawbacks for this discussion)

Let us write a pattern!

Creational Patterns

Creational Patterns

- Abstracts instantiation process
- Makes system independent of how its objects are
  - created
  - composed
  - represented
- Encapsulates knowledge about which concrete classes the system uses
- Hides how instances of these classes are created and put together

Singleton

Singleton Pattern

Ensure a class only has one instance, and provide a global point of access to it

Example that would benefit from Singleton Pattern

An application uses a Database Manager. It needs only one Database Manager object. It must not allow the creation of more than one object of this class.

Scenario: Audio Clip

- Imagine that your application played an audio clip
- Based on user action, a different audio clip may begin playing
- You want only one audio clip to play at a time

Goal:

You want to control all references to the AudioClip, and reference at one time only one AudioClip – In sum, you'll want to manage the AudioClip.

(I borrowed this example from Patterns in Java)

How would you design this?

- Not implement the functionality, or
- Create a class with only static methods, or
- Create a global variable to reference the clip, or
- Create only one instance of the object that plays the clip; reference only that object to play/stop/etc.

Problems with Static Functions/Global Var

- Data not protected from errant behavior; can be changed by any other part of application
- Decrease your chances of writing modular code (refactor or enhance)
- Objects no longer hold state and behavior

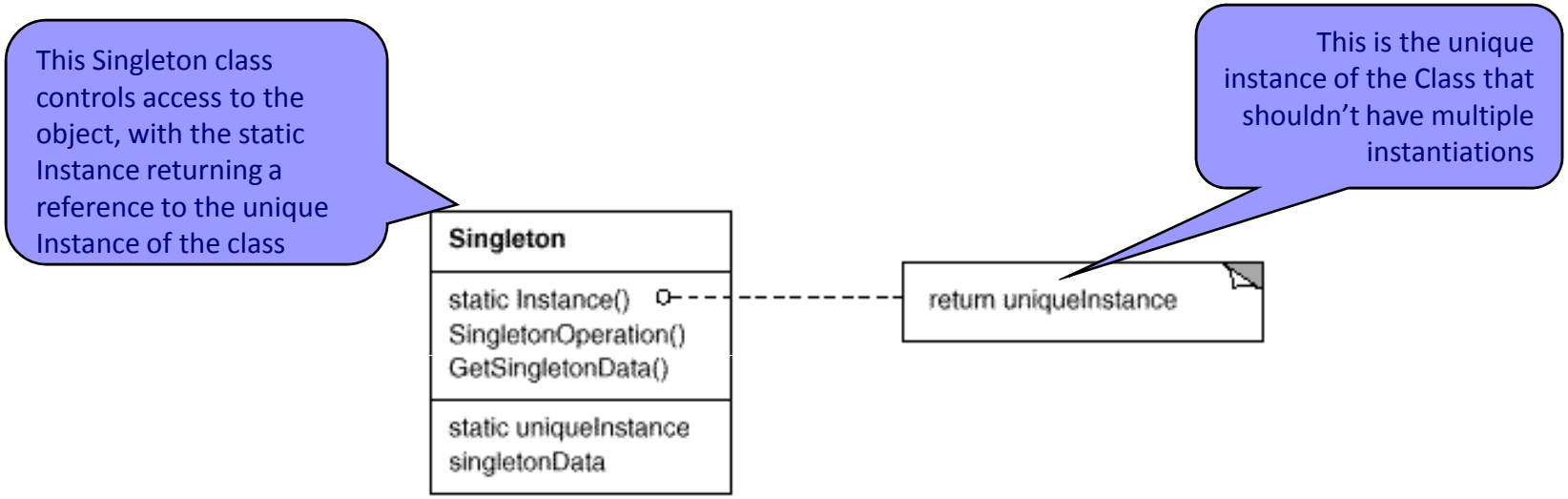
Creational Patterns

Singleton

Definition: Job of the Singleton is to enforce the existence of a maximum of one object of the same type at any given time. It does this by providing a global point of access to the object.

Singleton belongs to the family of Creational design patterns, those patterns that govern the instantiation process.

GofF UML Diagram



Scenario: MessageLogManager

Imagine that you want all messages, warnings, etc., displayed to the user to also be written to a log file. You don't want to create multiple LogWriters, nor do you want to get into the trouble of trying to write to the log file two different messages at the same time.

LogManager UML Diagram

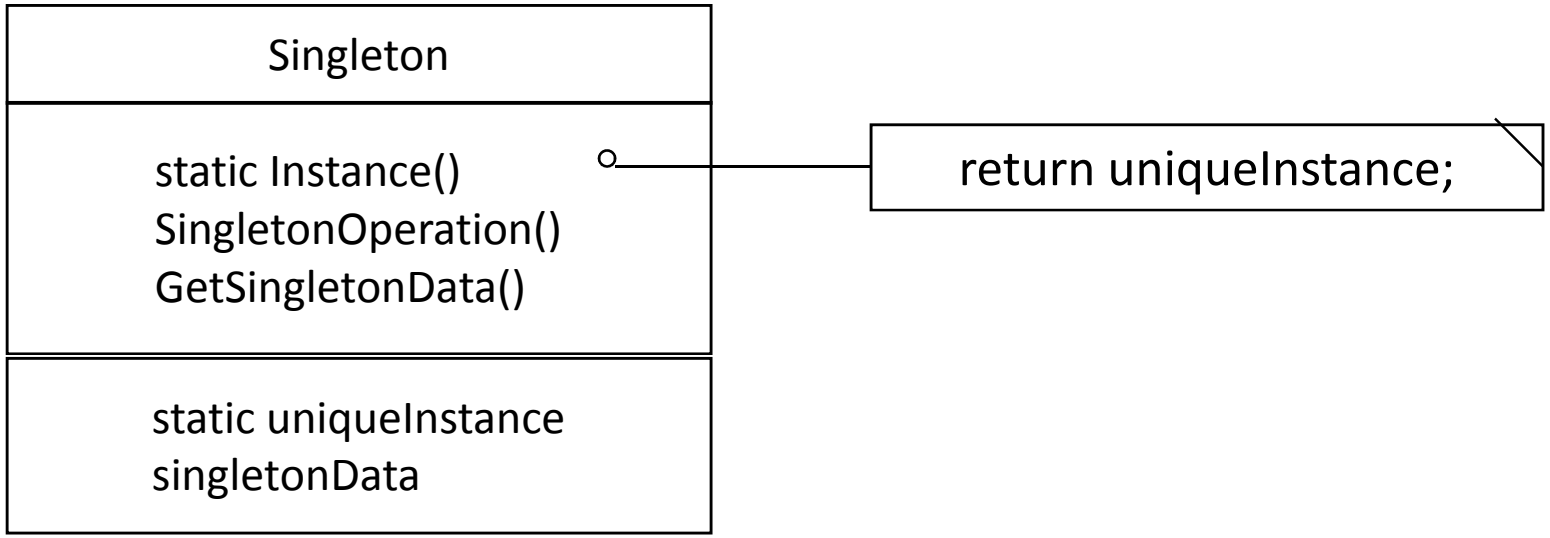
Possible Pitfalls

1. Depending on your implementation, your Singleton class – and all of its data – might be freed or garbage collected. If you cannot ensure a live reference to your Singleton class, you will need to force one.
2. The beauty of the Singleton pattern is NOT because the class is defined as a static object, but because the method or function is static, and the constructor is private. GofF: "A client that tries to instantiate Singleton directly will get an error at compile-time. This ensures that only one instance can ever get created."
3. Subclassing: Place the getInstance() method in your subclass, and not in the parent class (parent class could be an interface?). Registry of Singletons (beyond the scope of this presenter).
4. Multiple threads will also need to be carefully examined so a. one instance is created, and b. thread synchronization stays simple.

When to use Singleton Pattern

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
- when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

Structure


Singleton
static Instance()
SingletonOperation()
GetSingletonData()
static uniqueInstance
singletonData
return uniqueInstance;

Consequences of using Singleton

- Controlled access to sole instance
- Reduced name space
- Permits refinement of operations & representation
- Permits a variable number of instances
- More flexible than class operations

Singleton Vs. Other Patterns

- Several patterns are implemented using Singleton Pattern. For instance AbstractFactory needs a singleton pattern for single instance of the factory

Factory Method

Factory Method

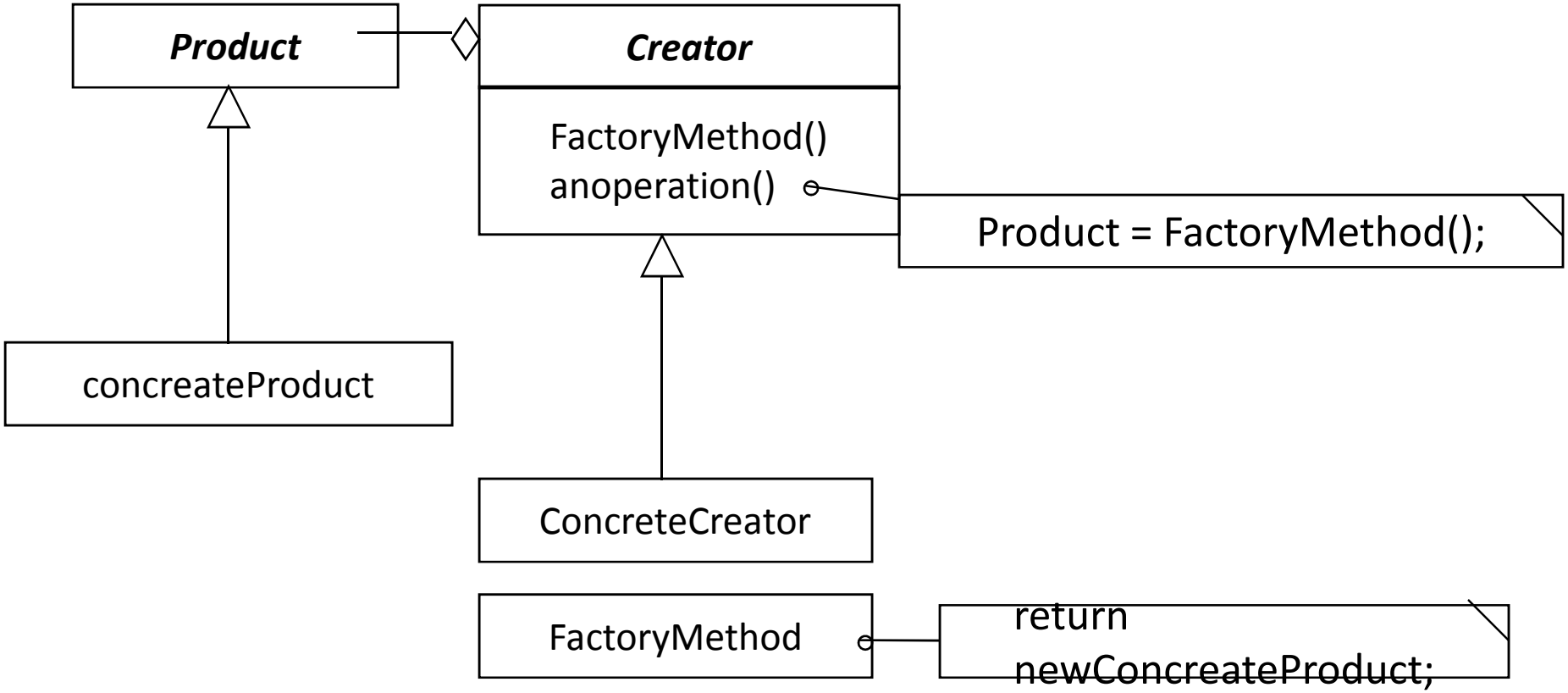
Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Also known as Virtual Constructor

When to use Factory Method?

- A class can't anticipate the class of objects it must create
- a class wants its subclasses to specify the objects it creates
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate

Structure


Product
Creator
FactoryMethod()
anoperation()
Product = FactoryMethod();
concreateProduct
ConcreteCreator
FactoryMethod
return
newConcreateProduct;

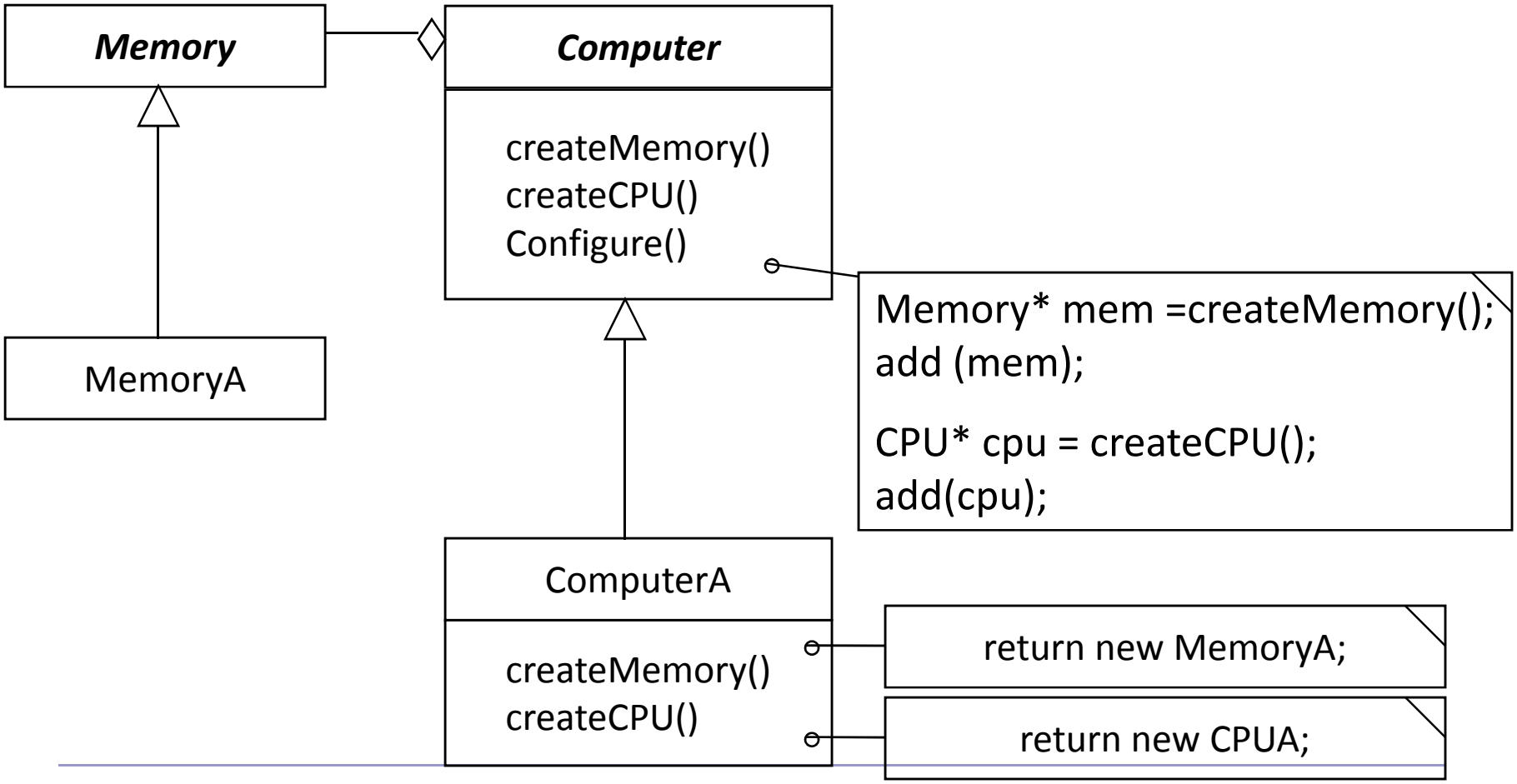
Consequences of using Factory Method

- Provides hooks for subclasses
- Connects parallel class hierarchies

Example that would benefit from Factory Method

We want to develop a framework of a Computer that has memory, CPU and Modem. The actual memory, CPU, and Modem that is used depends on the actual computer model being used. We want to provide a configure function that will configure any computer with appropriate parts. This function must be written such that it does not depend on the specifics of a computer model or the components.

Example using Factory Method



Factory Method Vs. Other Pattern

- Abstract Factory often implemented with Factory Method
- Factory Methods usually called within Template Methods
- Prototypes don't require sub-classing the Creator. However, they often require initialize operation on the Product class. Factory Method doesn't require such an operation

- Proliferation of subclasses

Abstract Factory

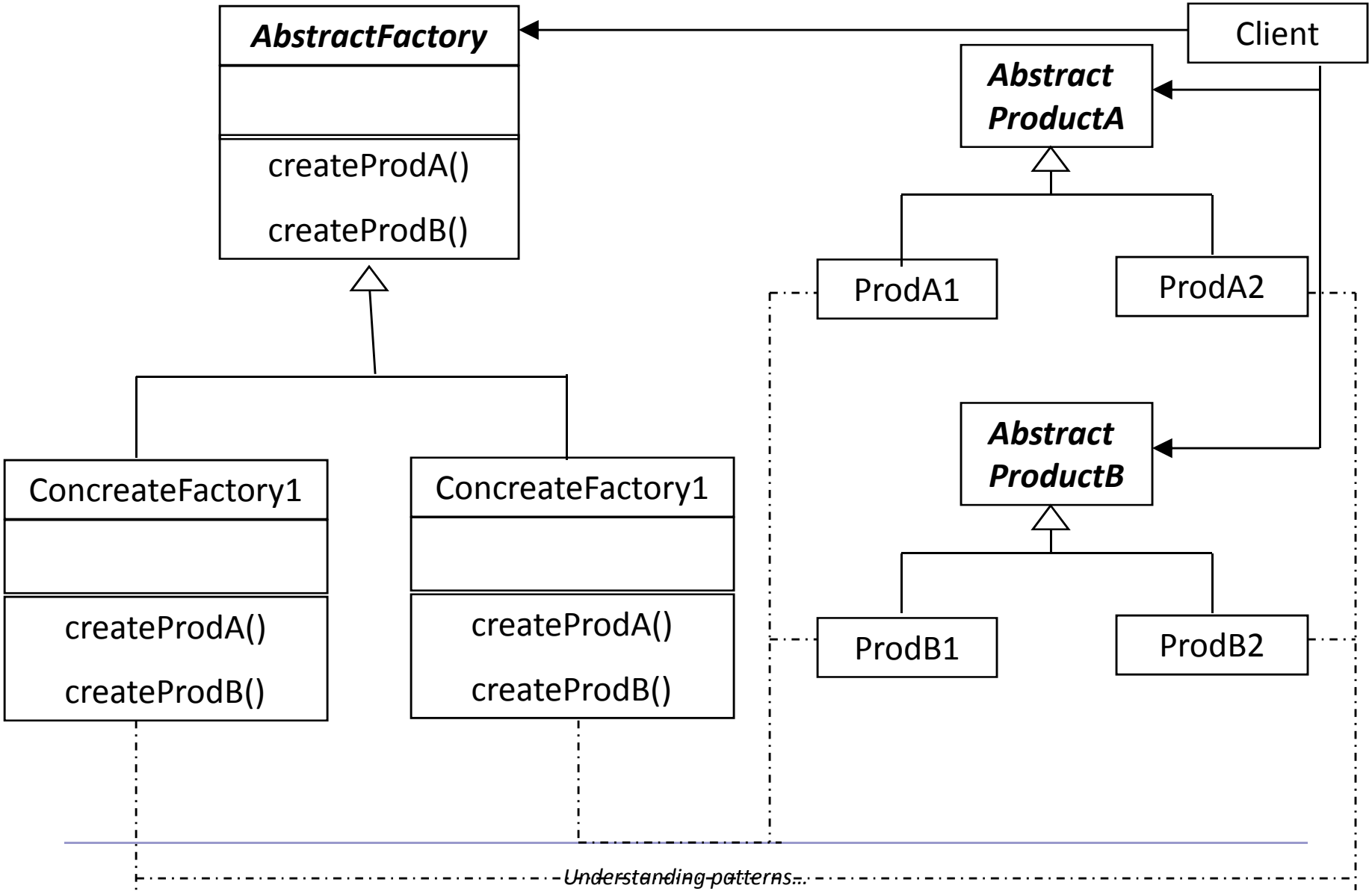
Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes

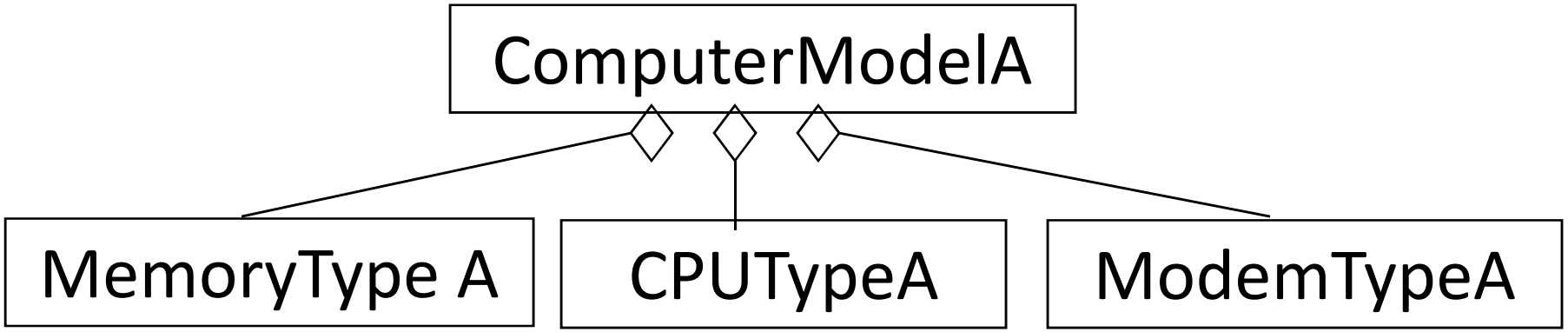
When to use Abstract Factory?

- Use Abstract Factory when:
  - system should be independent of how its products are created, composed and represented
  - system should be configured with one of multiple families of products
  - a family of related product objects must be used together and this constraint need to be enforced
  - you want to reveal only the interface, and not the implementation, of a class library of products

Structure


AbstractFactory
createProdA()
createProdB()
Client
Abstract ProductA
ProdA1
ProdA2
Abstract ProductB
ProdB1
ProdB2
ConcreateFactory1
createProdA()
createProdB()
ConcreateFactory1
createProdA()
createProdB()

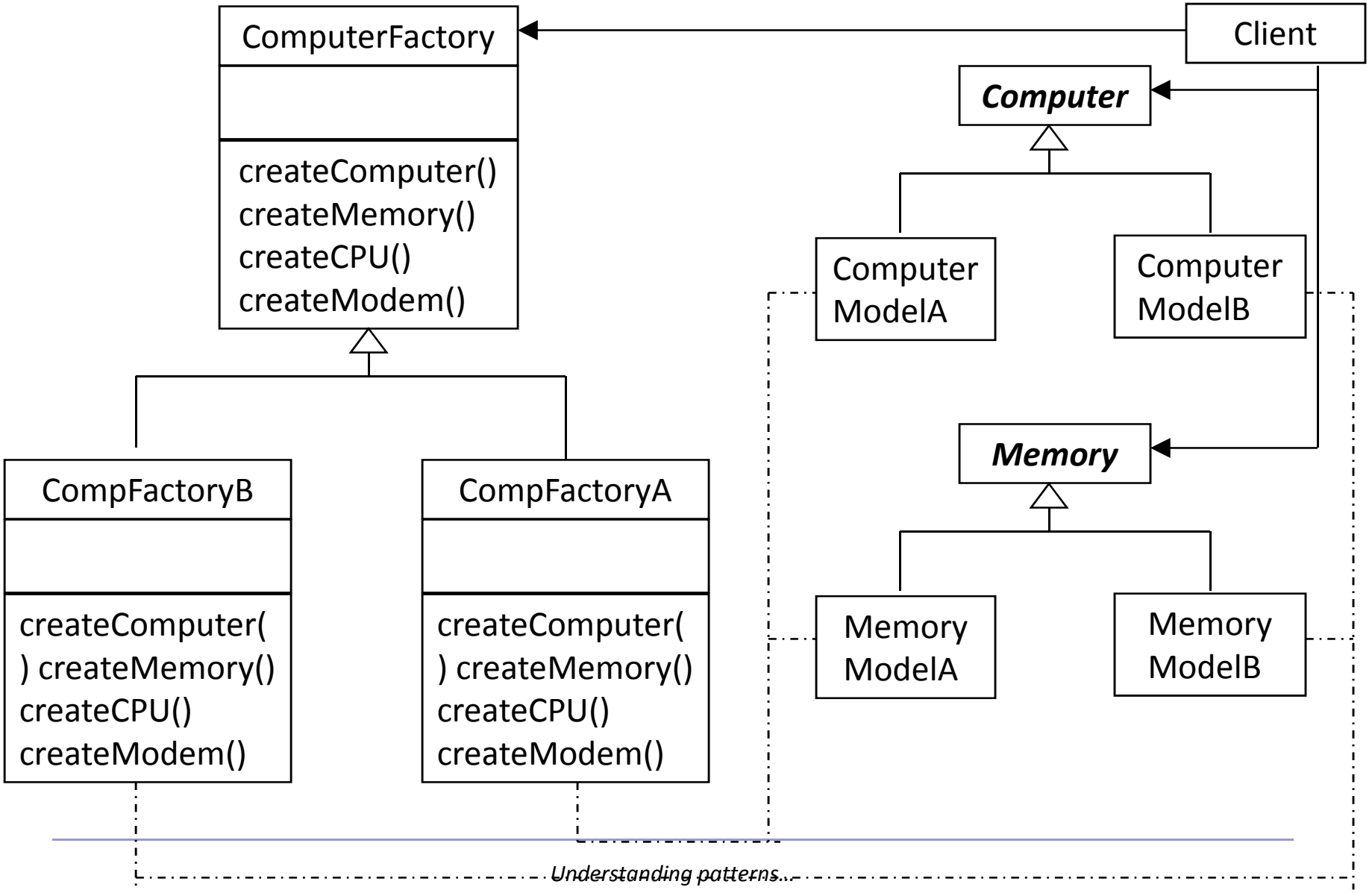
Example that would benefit from Abstract Factory




```
BuildComputer(ComputerModelA& comp)
{  comp.Add(new MemoryTypeA);
   comp.Add(new CPUTypeA);
   comp.Add(new ModemTypeA); }
```

What if I want to build a Computer of Model B with Model B Memory,CPU and Modem?

Using Abstract Factory



Using Abstract Factory…

```
BuildComputer(Computer& comp,
   ComputerFactory&       compFactory)
{
   comp.Add(compFactory.createMemory());
   comp.Add(compFactory.createCPU());
   comp.Add(compFactory.createModem());
}
```

onsequences of using bstract Factory

- Isolates concrete classes
- Makes exchanging products families easy
- Promotes consistency among products
- Supporting new kinds of products is difficult

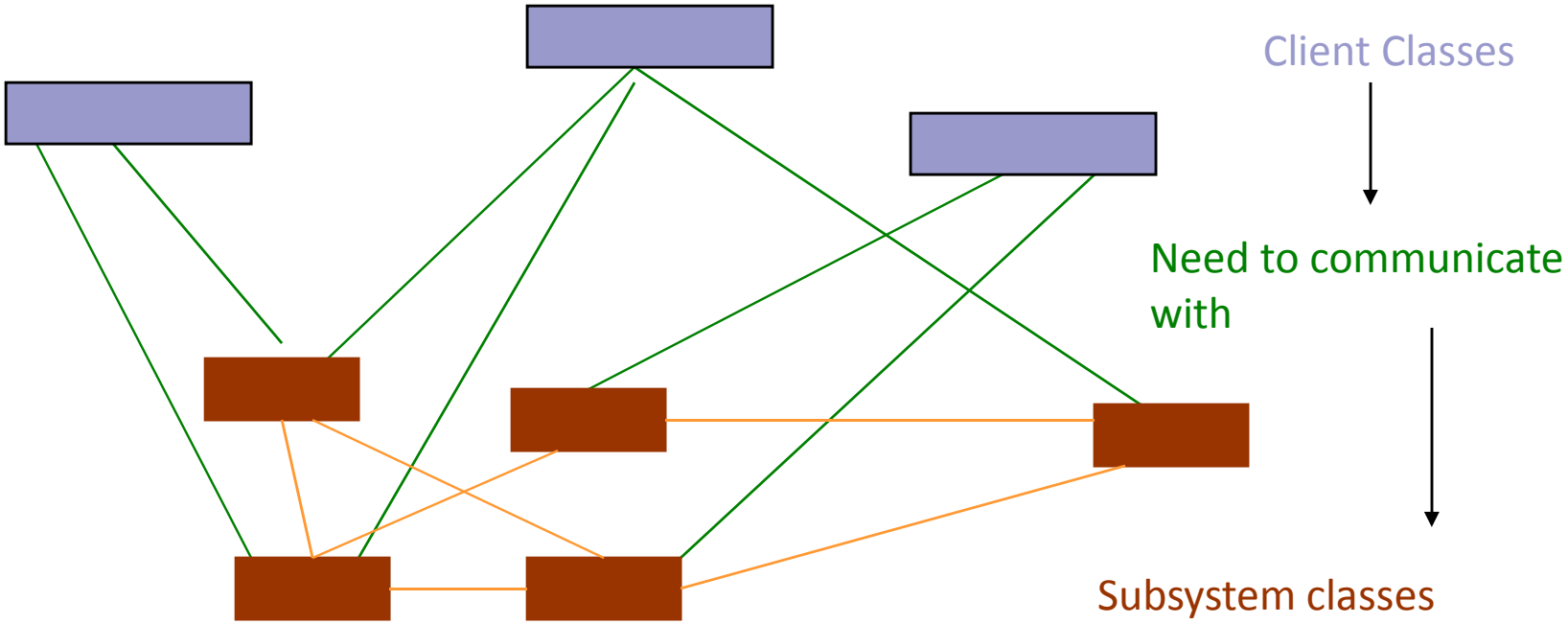
Structural Patterns

Structural Patterns

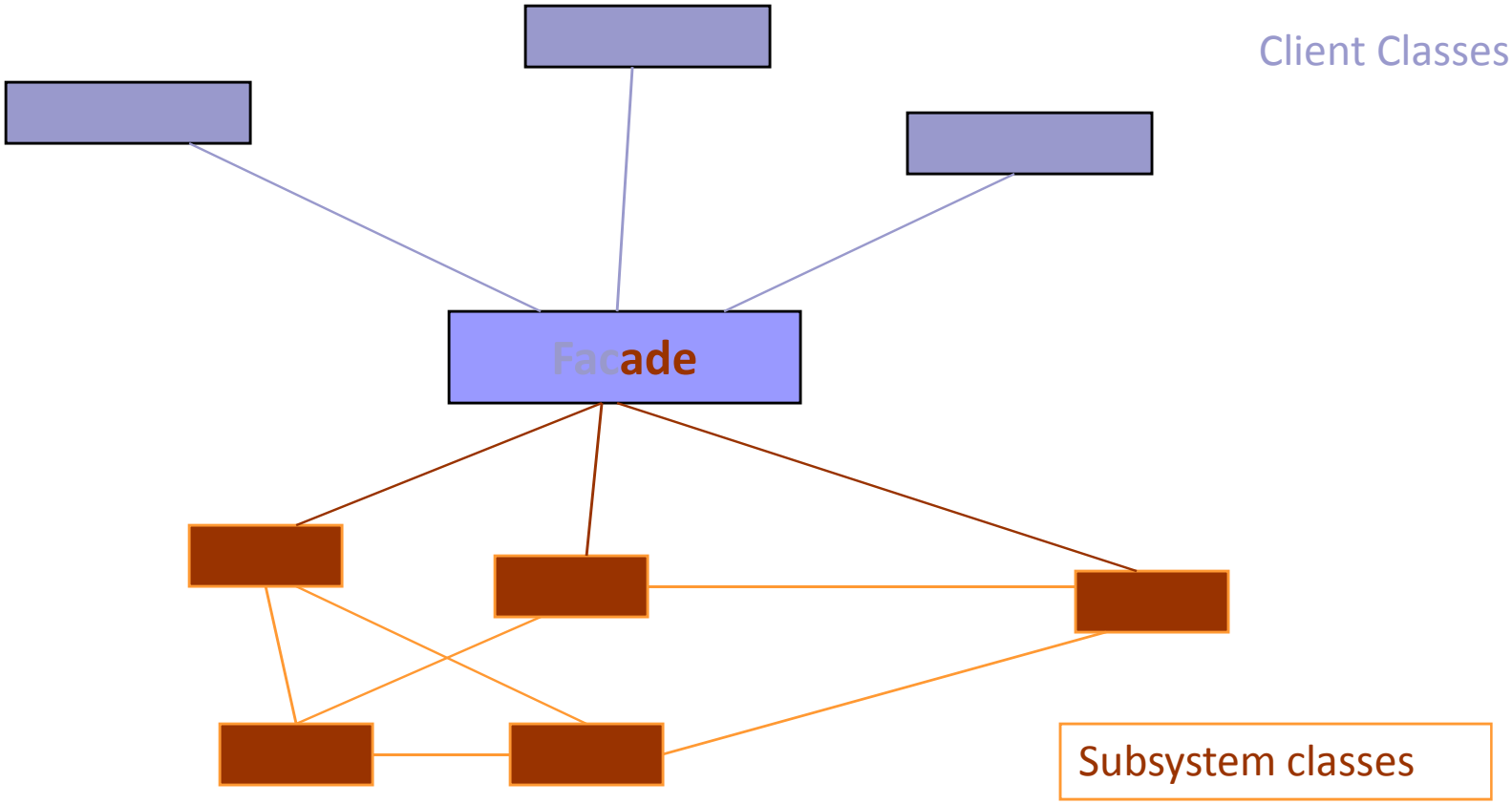
- Describe how classes and objects are composed to form larger structures
- Structural **class** patterns use inheritance to compose interfaces or implementations
- Structural **object** patterns describe ways to compose objects to realize new functionality

Façade

Facade Pattern: Problem



Facade Pattern: Solution



Facade Pattern: Why and What?

- Subsystems often get complex as they evolve.

- Need to provide a simple interface to many, often small, classes. *But not necessarily to ALL classes of the subsystem.*
- Façade provides a simple default view good enough for most clients.
- Facade decouples a subsystem from its clients.
- A façade can be a single entry point to each subsystem level. This allows layering.

Facade Pattern: Participants and Communication

- Participants: Façade and subsystem classes

- Clients communicate with subsystem classes by sending requests to façade.
- Façade forwards requests to the appropriate subsystem classes.
- Clients do not have direct access to subsystem classes.

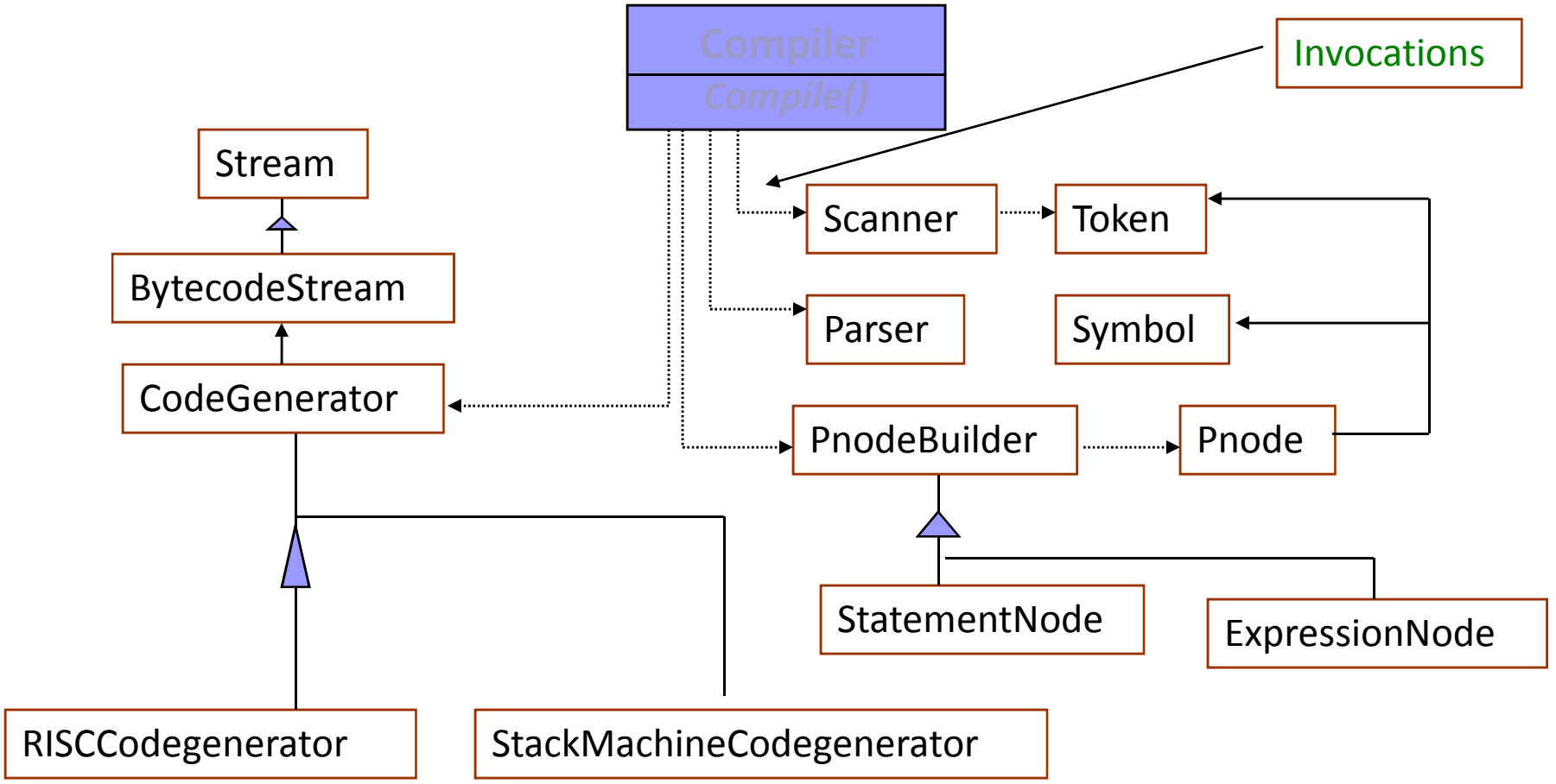
Facade Pattern: Benefits

- Shields clients from subsystem classes; reduces the number of objects that clients deal with.

- Promotes weak coupling between subsystem and its clients.

- Helps in layering the system. Helps eliminate circular dependencies.

Example: A compiler



Façade Pattern: Example

```
// Façade. Offers a simple interface to compile and
// Generate code.
public:
class Compiler {
        Compiler();
        virtual void Compile (istream&, BytecodeStream&);
    }
void Compiler:: Compile (istream& input, BytecodeStream& output) {
        Scanner scanner (input);
        PnodeBuilder builder;
        Parser parser;
        parser.Parse (scanner, builder);
        RISCCodeGenerator generator (output);
        Pnode* parseTree = builder.GetRootNode();
        parseTree→Traverse (generator);
}
```

Could also take a CodeGenerator Parameter for increased generality.

Facade Pattern: Another Example from POS [1]

- Assume that rules are desired to invalidate an action:
  - Suppose that when a new Sale is created, it will be paid by a gift certificate
    - Only one item can be purchased using a gift certificate.
  - Hence, subsequent enterItem operations must be invalidated in some cases. (Which ones?)

How does a designer factor out the handling of such rules?

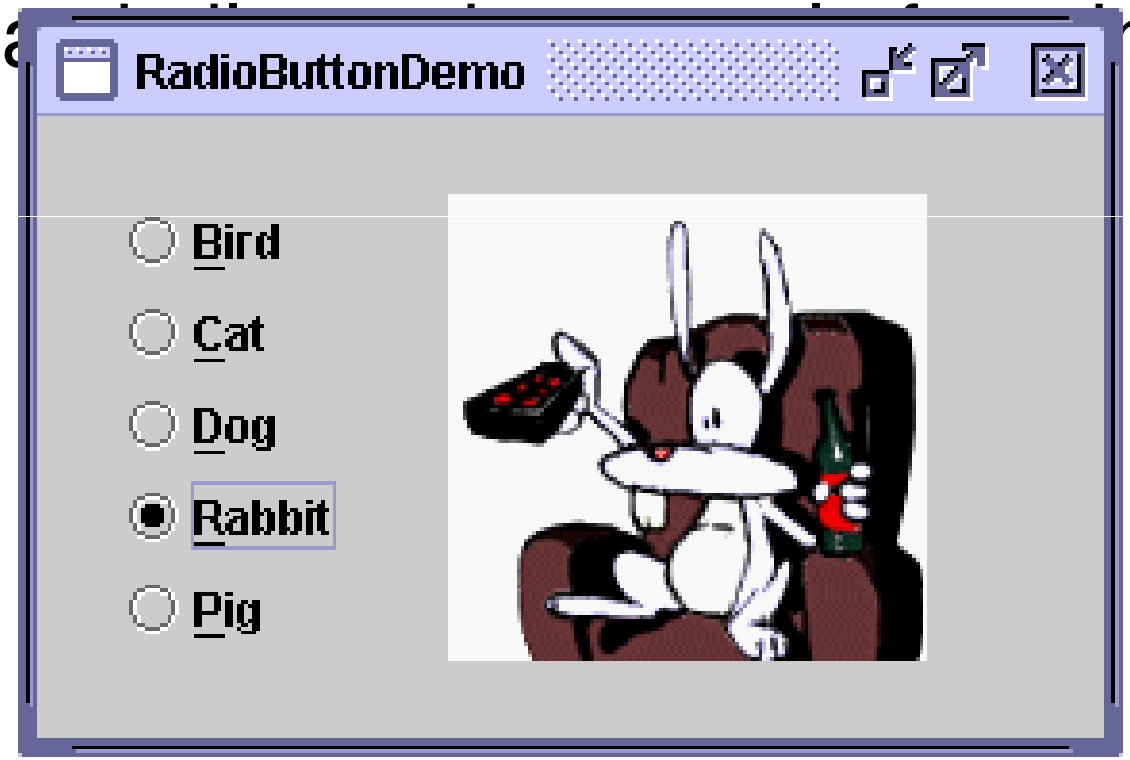
Decorator

An example: a coffee machine

- You are requested to do an implementation for the GUI menu for the coffee machine
- Return value: the option chosen by the user

An easy way

Hack a [illegible] nternet

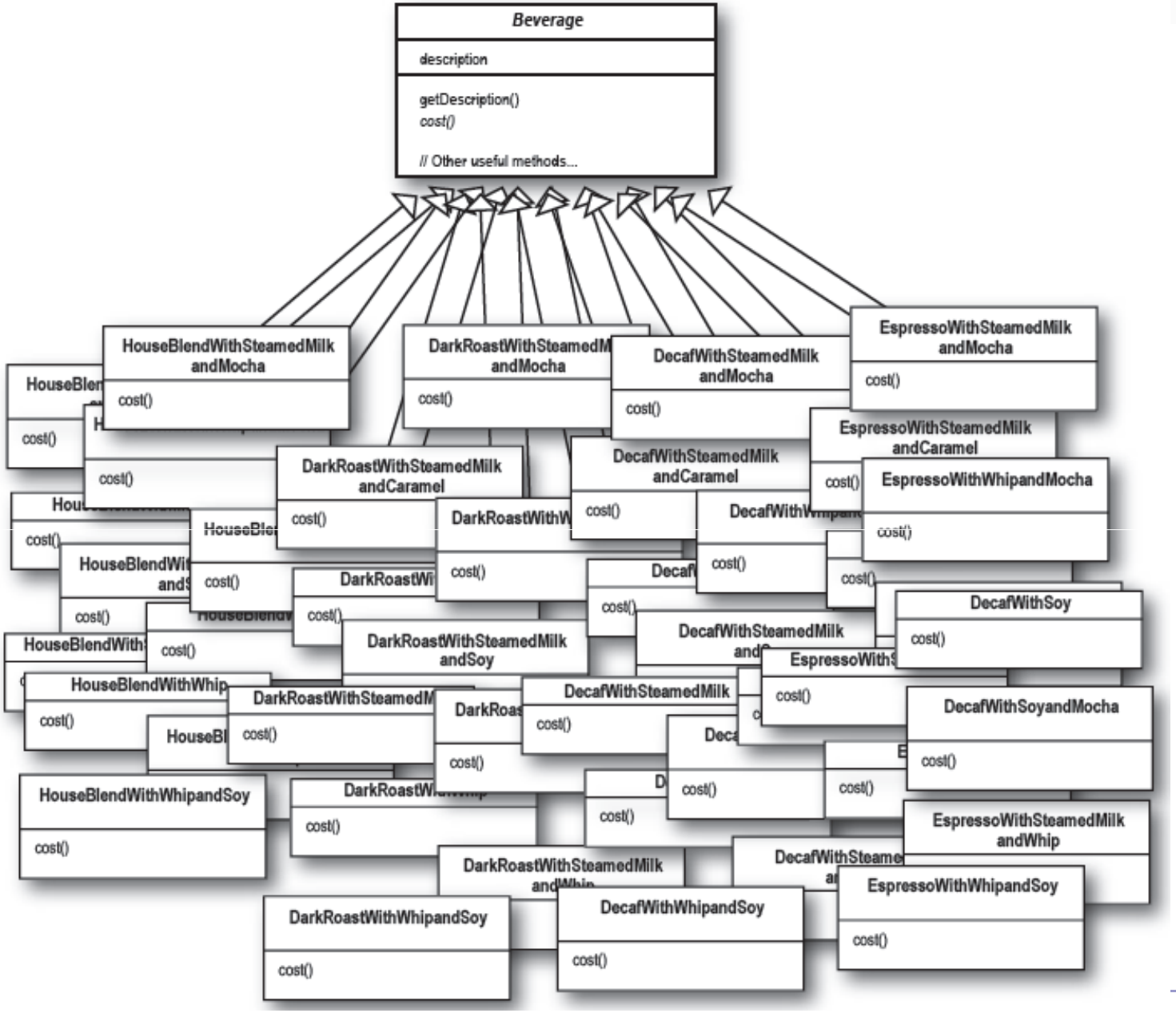


C# example

- Run modified h
  WinForms

Adding more options

- There are 4 types of coffee where the user can add something more
- Soy, sugar, whipped milk, cinnamon, coffee without caffeine, strong coffee, etc..
- Each set of options have different prices
- Easy solution: one class for each combination of options


Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
cost()
DarkRoastWithSteamedMilk andMocha
cost()
DecafWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andCaramel
cost()
DarkRoastWithSteamedMilk andCaramel
cost()
DecafWithSteamedMilk andCaramel
cost()
EspressoWithWhipandMocha
cost()
DecafWithSoy
cost()
DarkRoastWithSteamedMilk andSoy
DecafWithSteamedMilk
cost()
DecafWithSoyandMocha
cost()
HouseBlendWithWhipandSoy
cost()
EspressoWithSteamedMilk andWhip
DarkRoastWithSteamedMilk
DecafWithWhipandSoy
cost()
DarkRoastWithWhipandSoy
cost()
EspressoWithWhipandSoy
cost()

Maintenance

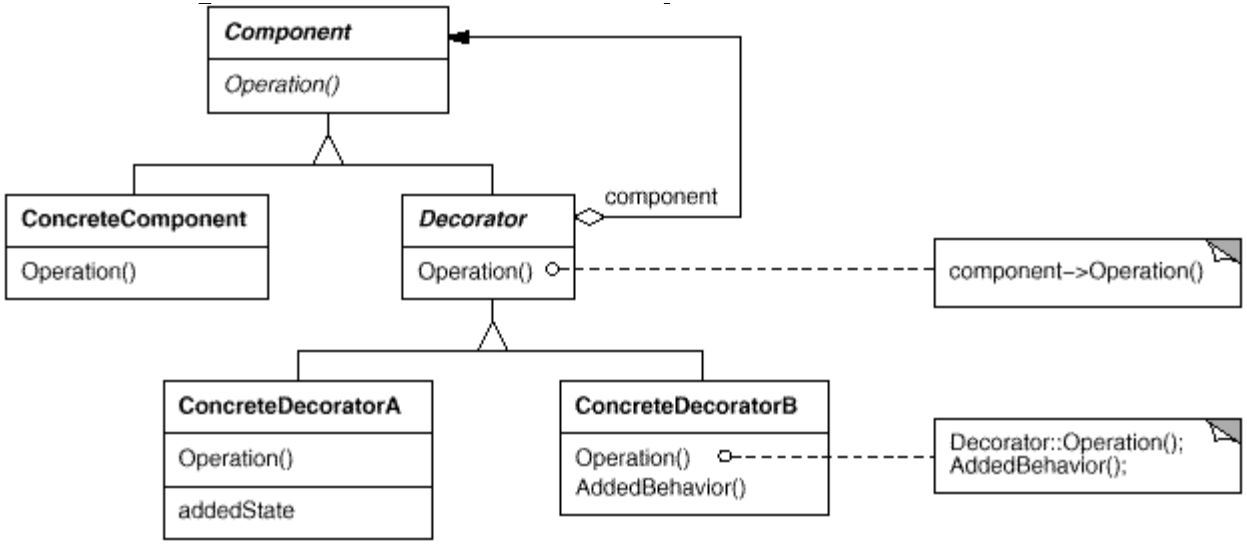
- In the design, you should predict the future: what if the client want to add more options or change the price?
- Maintenance: how easy will it be to do a change?

The Decorator

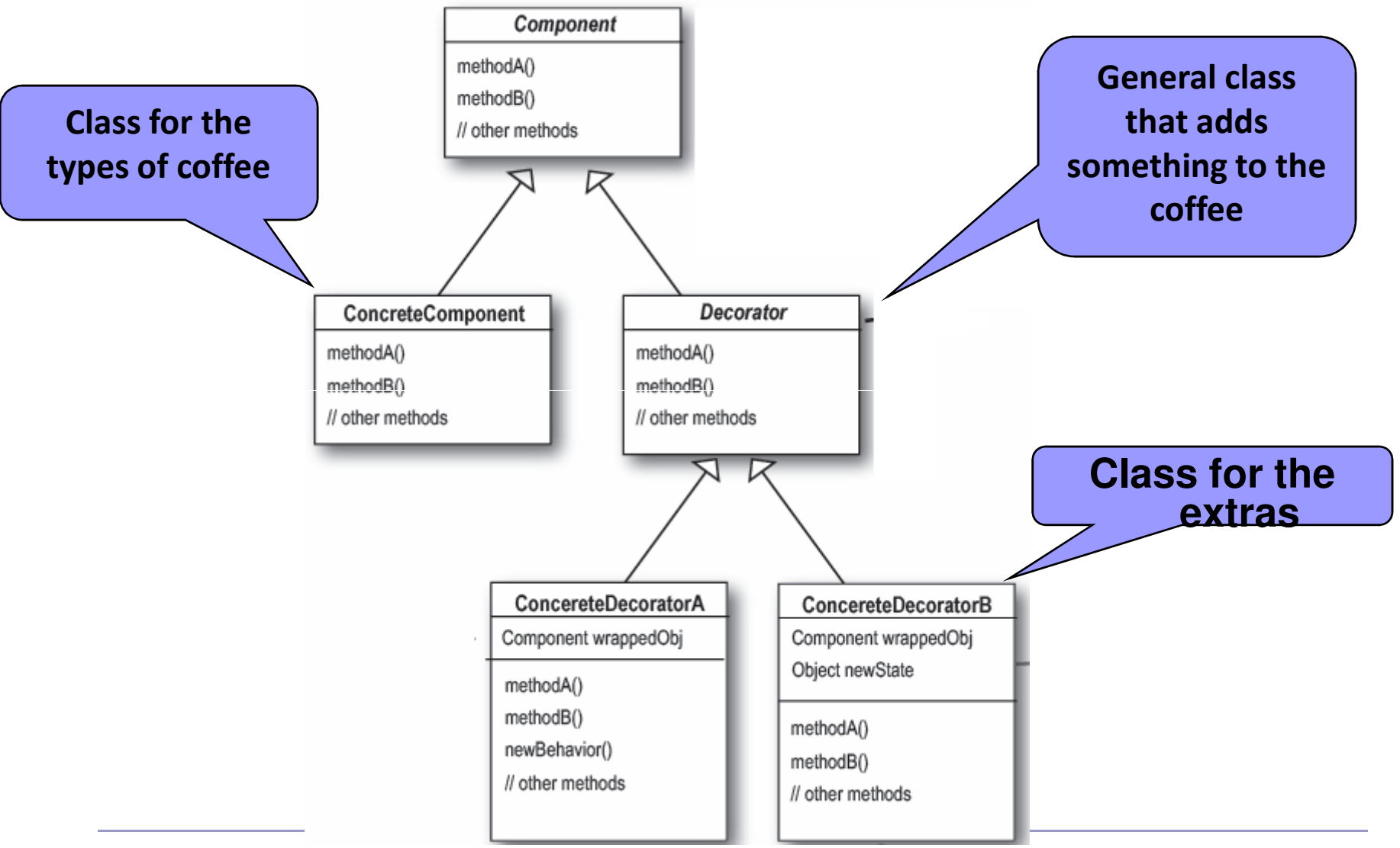
- Also called wrapper
- Use it to add (or decorate with) more functionalities to an object/class

Decorator

- Intent
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alter
    func



Structure


Component
methodA()
methodB()
// other methods
Class for the types of coffee
General class that adds something to the coffee
ConcreteComponent
methodA()
methodB()
// other methods
Decorator
methodA()
methodB()
// other methods
Class for the extras
ConcereteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
// other methods
ConcereteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
// other methods

Consequences

- *More flexibility than static inheritance.* The Decorator pattern provides a more flexible way to add responsibilities to objects than can be had with static (multiple) inheritance. With decorators, responsibilities can be added and removed at run-time simply by attaching and detaching them. In contrast, inheritance requires creating a new class for each additional responsibility. This gives rise to many classes and increases the complexity of a system.
- Furthermore, providing different Decorator classes for a specific Component class lets you mix and match responsibilities. Decorators also make it easy to add a property twice. For example, to give a TextView a double border, simply attach two BorderDecorators. Inheriting from a Border class twice is error-prone at best.

Consequences

- *Avoids feature-laden classes high up in the hierarchy.* Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects. Functionality can be composed from simple pieces. As a result, an application needn't pay for features it doesn't use. It's also easy to define new kinds of Decorators independently from the classes of objects they extend, even for unforeseen extensions. Extending a complex class tends to expose details unrelated to the responsibilities you're adding.

- *A decorator and its component aren't identical.* A decorator acts as a transparent enclosure. But from an object identity point of view, a decorated component is not identical to the component itself. Hence you shouldn't rely on object identity when you use decorators.

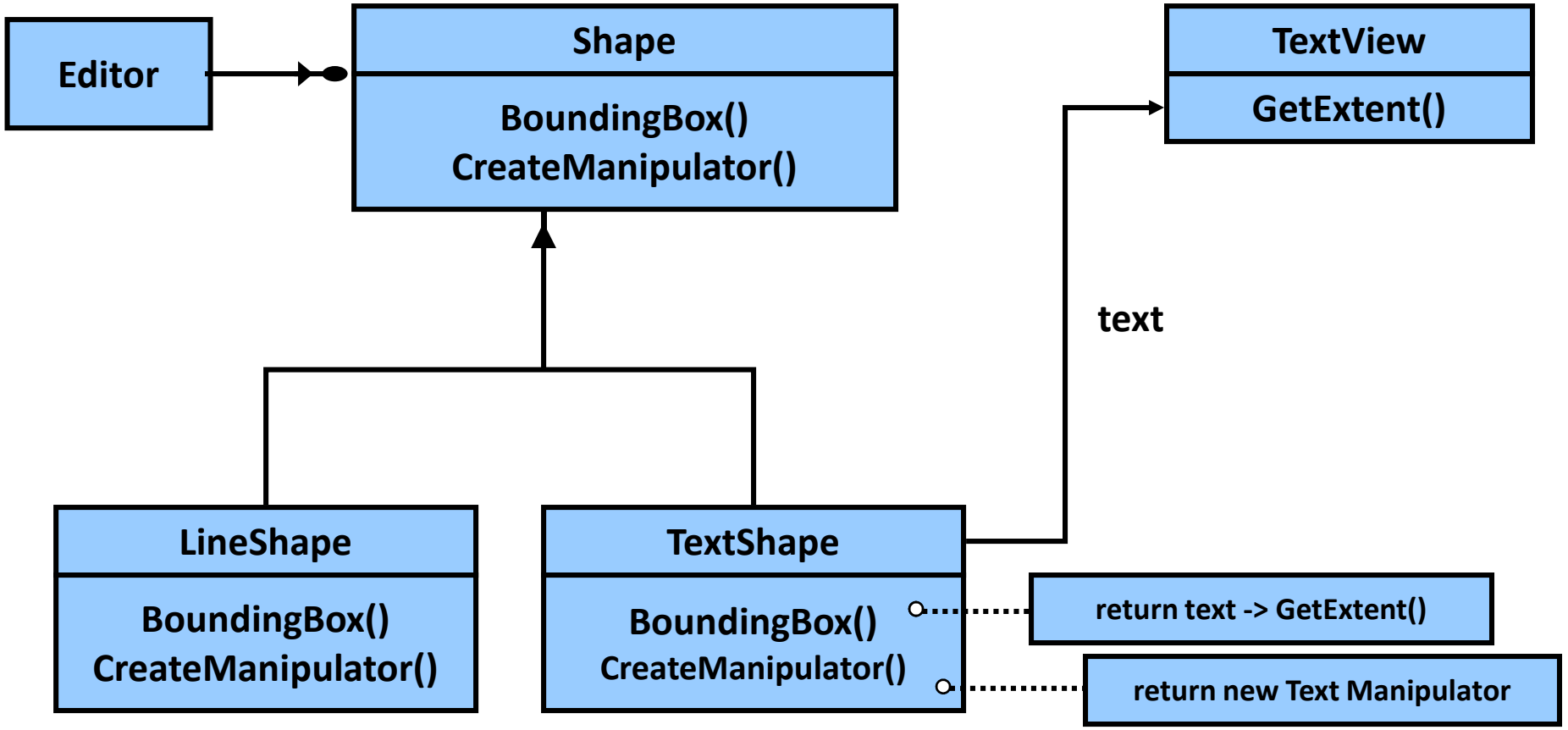
- *Lots of little objects.* A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected, not in their class or in the value of their variables. Although these systems are easy to customize by those who understand them, they can be hard to learn and debug.

Adapter

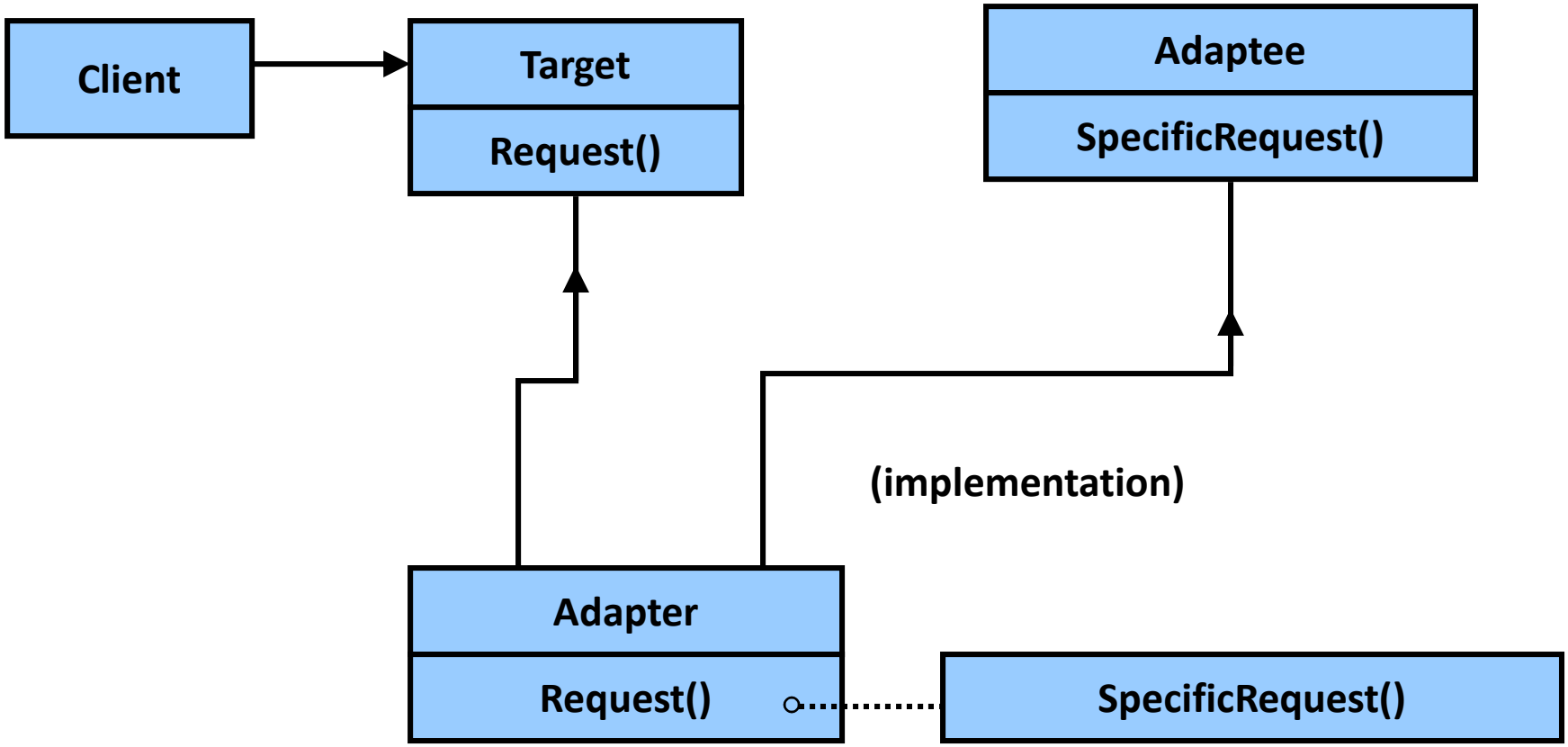
The Adapter Pattern

- **Intent:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Motivation:** When we want to reuse classes in an application that expects classes with a different interface, we do not want (and often cannot) to change the reusable classes to suit our application.

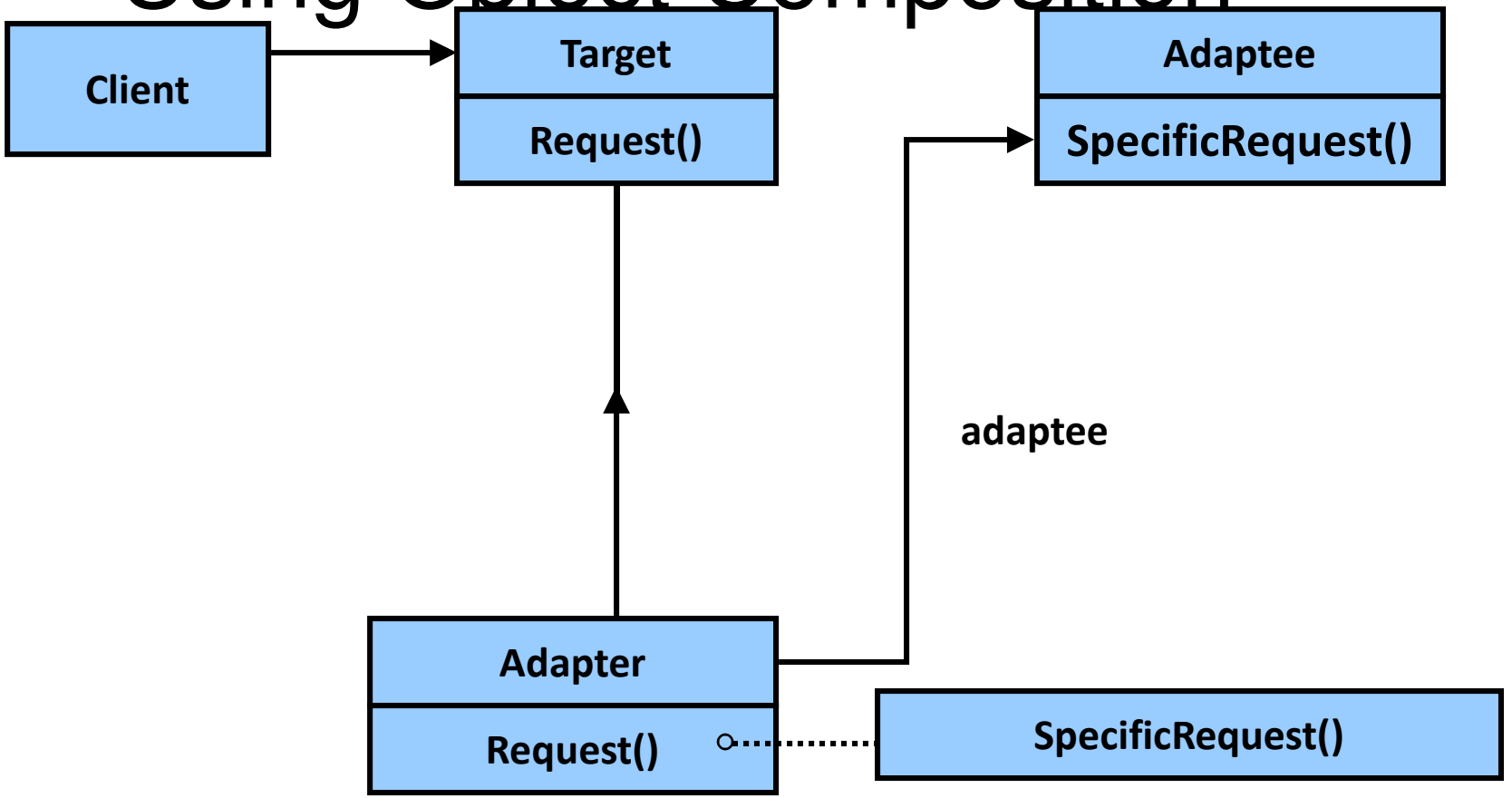
Example of the Adapter Pattern



Structure of the Adapter Pattern Using Multiple Inheritance



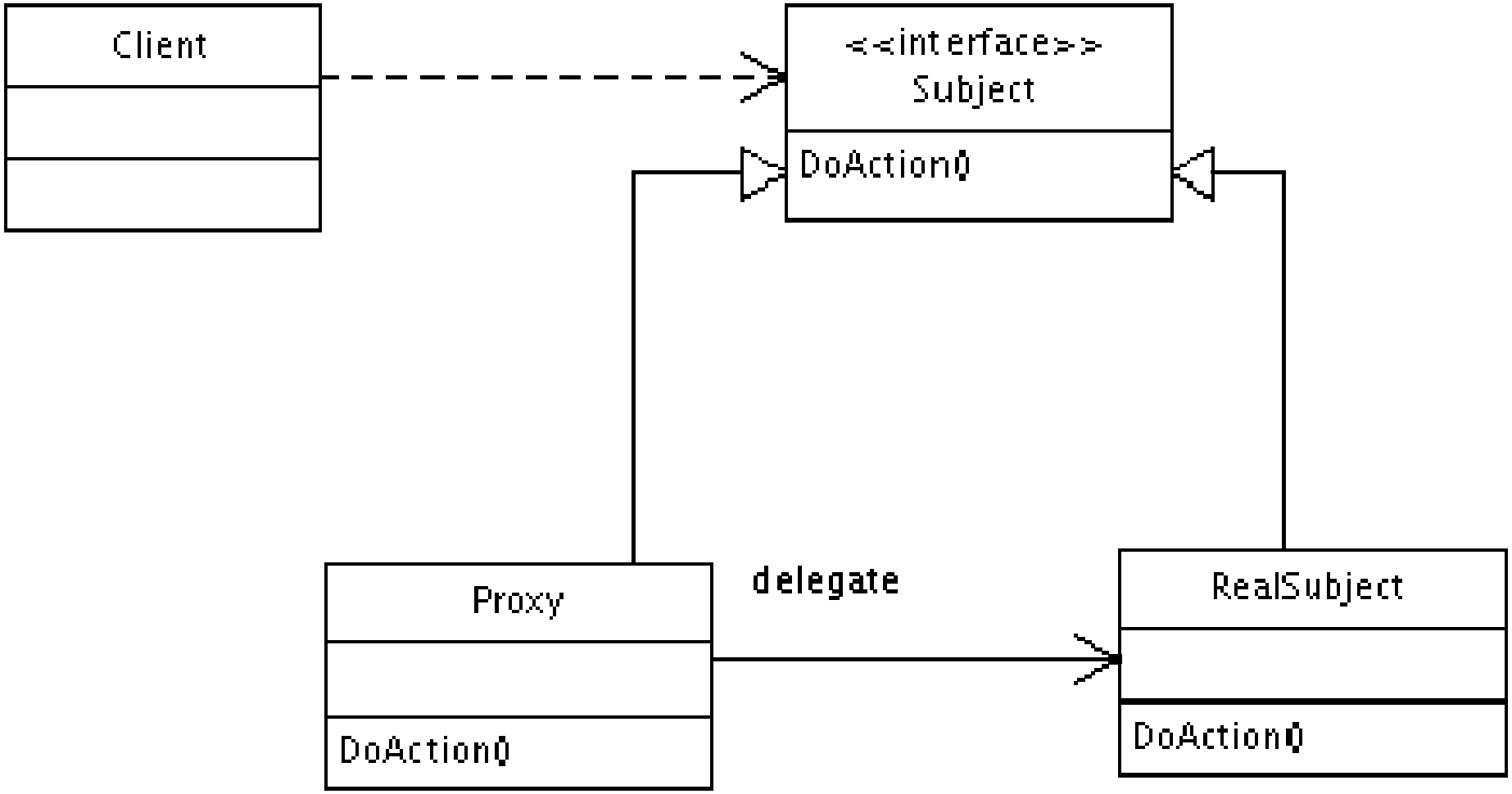
Structure of the Adapter Pattern Using Object Composition



Participants of the Adapter Pattern

- **Target:** Defines the application-specific interface that clients use.
- **Client:** Collaborates with objects conforming to the target interface.
- **Adaptee:** Defines an existing interface that needs adapting.
- **Adapter:** Adapts the interface of the adaptee to the target interface.

- Provide a surrogate or placeholder for another object to control access to it.
- In its most general form, is a class functioning as an interface to another thing. The other thing could be anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate.


Client
<<interface>>
Subject
DoAction()
Proxy
DoAction()
delegate
RealSubject
DoAction()

Participants

- The classes and/or objects participating in this pattern are:
- **Proxy (MathProxy)**
  - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - provides an interface identical to Subject's so that a proxy can be substituted for for the real subject.
  - controls access to the real subject and may be responsible for creating and deleting it.
  - other responsibilites depend on the kind of proxy:
    - *remote proxies* are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
    - *virtual proxies* may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real images's extent.
    - *protection proxies* check that the caller has the access permissions required to perform a request.
- **Subject (IMath)**
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject (Math)**
  - defines the real object that the proxy represents.

- In situations where multiple copies of a complex object must exist the proxy pattern can be adapted to incorporate the Flyweight Pattern in order to reduce the application's memory footprint. Typically one instance of the complex object is created, and multiple proxy objects are created, all of which contain a reference to the single original complex object. Any operations performed on the proxies are forwarded to the original object. Once all instances of the proxy are out of scope, the complex object's memory

Proxy examples

- Types of proxy pattern include:
- **Remote proxy**: Provides a reference to an object located in a different address space on the same or different machine.
- **Virtual proxy**: Allows the creation of a memory intensive object on demand. The object will not be created until it is really needed. (See also Lazy evaluation.)
- **Copy-on-write proxy**: Defers copying (cloning) a target object until required by client actions. This is really a special case of the "virtual proxy" pattern.
- **Protection (access) proxy**: Provides different clients with different levels of access to a target object.
- **Cache proxy**: Provides temporary storage of the results of expensive target operations so that multiple clients can share the results. (See also Memoization.)
- **Firewall proxy**: Protects targets from bad clients (or vice versa).
- **Synchronization proxy**: Provides concurrency control over an unsynchronized target object.
- **Smart reference proxy**: Provides additional actions whenever a target object is referenced, such as counting the number of references to the object.

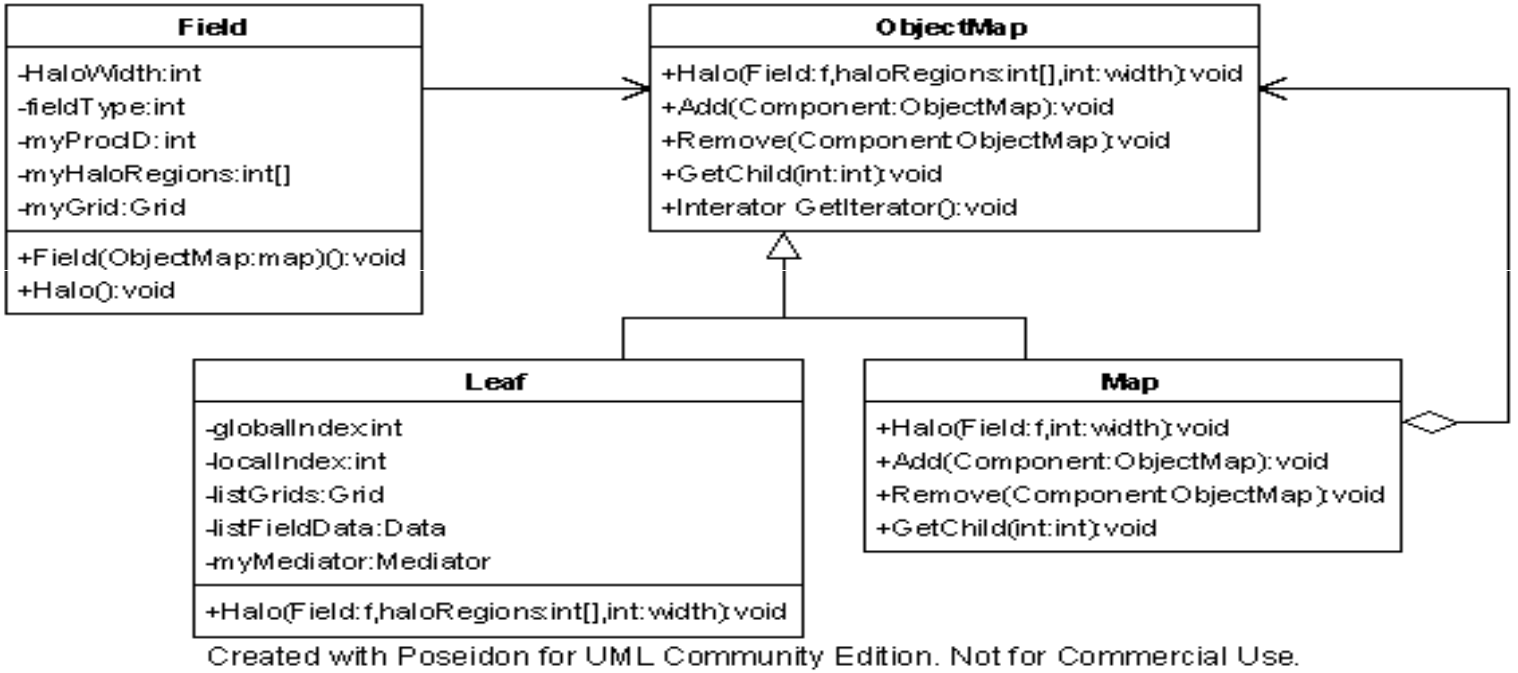
Composite

Composite pattern

- **composite:** an object that is either an individual item or a collection of many items
- composite objects can be composed of individual items or of other composites
- recursive definition: objects that can hold themselves
- often leads to a tree structure of leaves and nodes:
  - <node> ::= <leafnode> | <compositenode>
  - <compositenode> ::= <node>*

- examples in Java:
  - collections (a List of Lists)
  - GUI layout (panels containing panels containing buttons, etc.)

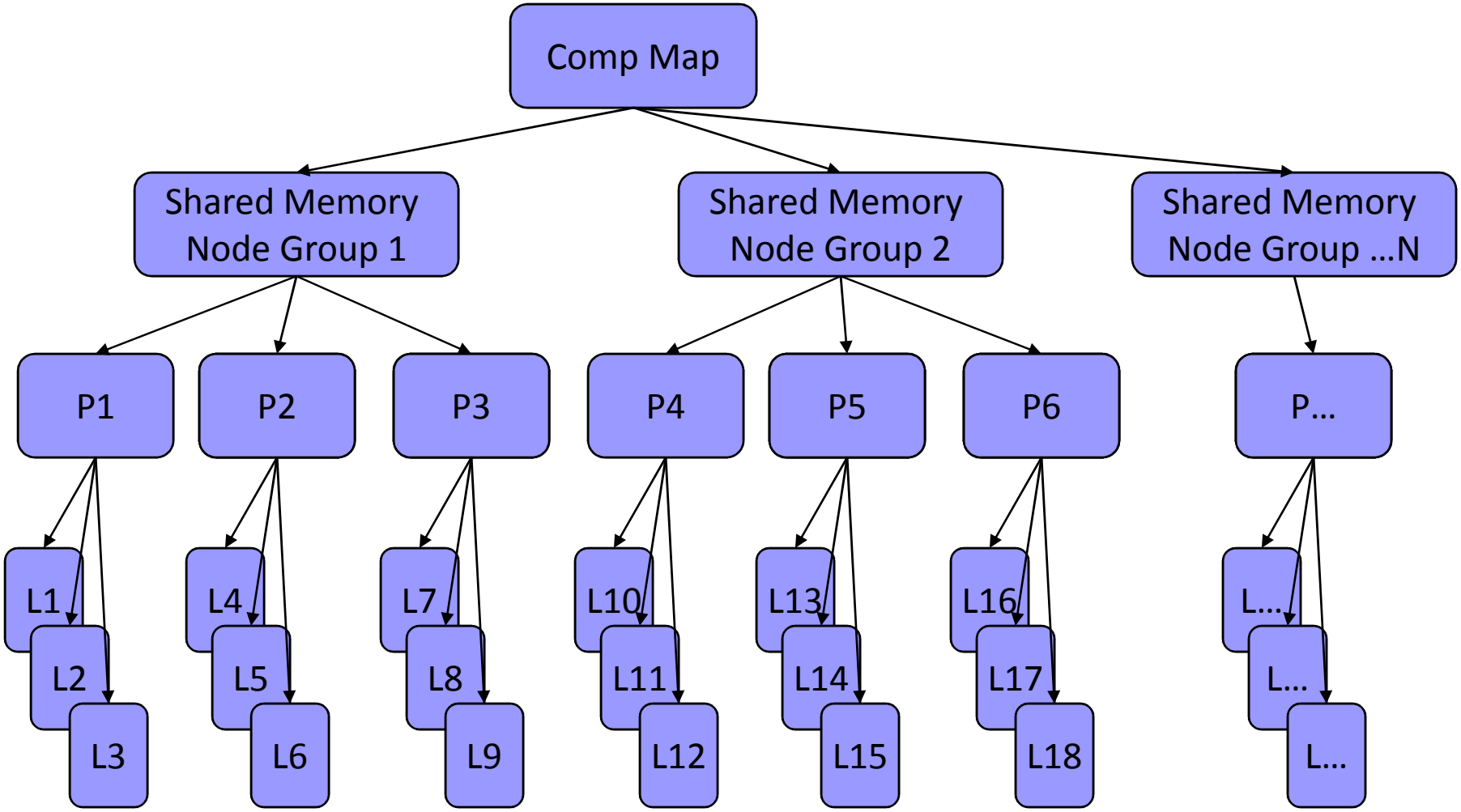
GoF Composite Pattern



GoF Composite Pattern

- Allows us to maintain a unified picture of subdomains whether we want to think of them in:
  - a global setting, the problem as a whole
  - a local setting, the problem as local pieces.
- Methods associated with the composite provide a common interface at each level of the decomposition and allow the software developer to move from one view of the problem to another in a consistent manner.

Composite Tree Structure



Composite features

- Application code is written to expect a block of data from a leaf. It shouldn't assume any specific :
  - index order
  - data layout
  - or decomposition
- Blocks of data are assigned to processors by the object map structure
  - migration of work is done by simply moving one or more leaves to a different location within the composite
- Pieces of work (leaves) can be adaptively refined(AMR splitting/combining of sub elements) by replacing a leaf with a composite containing the newly refined leaves or the reverse operation

Behavioral Patterns

Behavioral design patterns identify common

communication patterns between objects and

realize these patterns. By doing so, these patterns

increase flexibility in carrying out this communication.

Observer

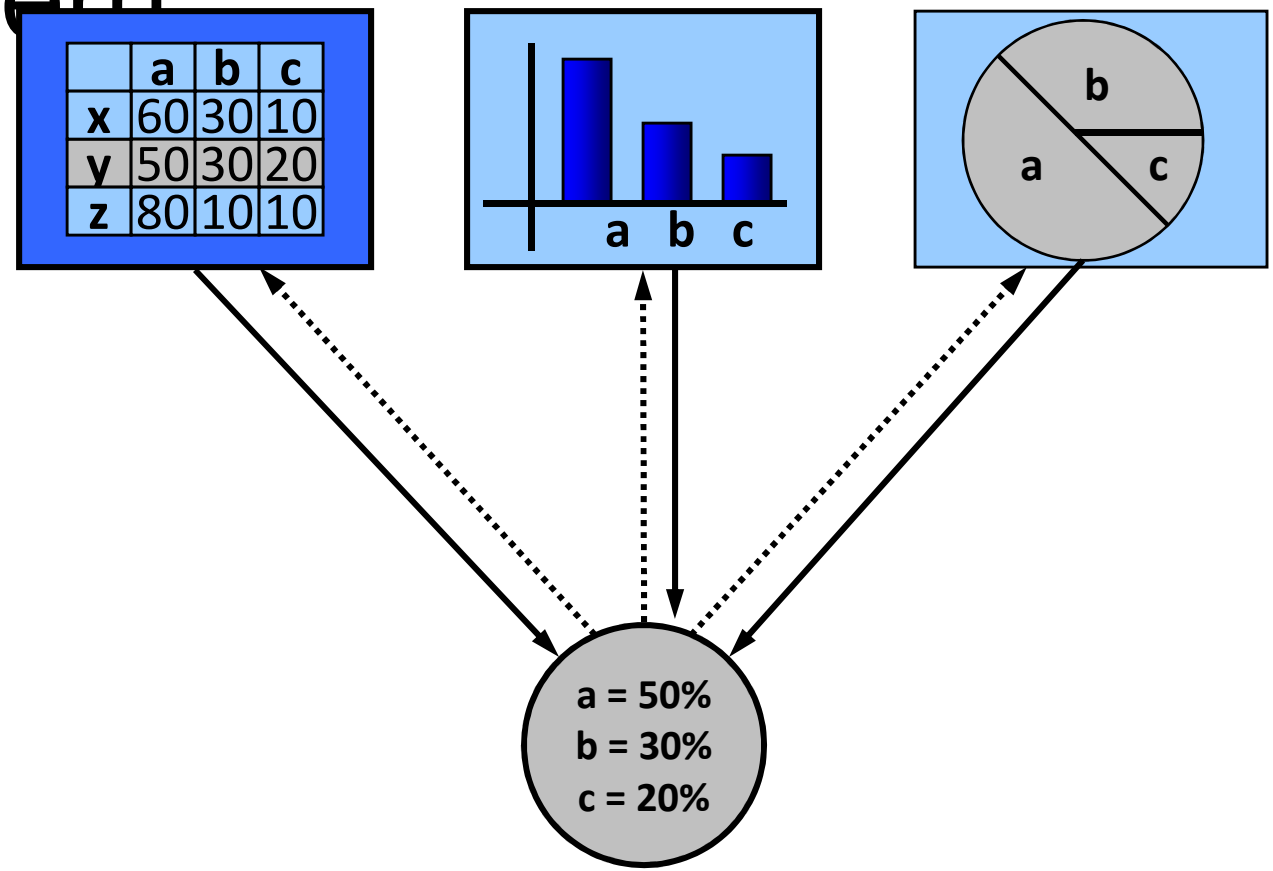
The Observer Pattern (Intent)

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

The Observer Pattern (Motivation)

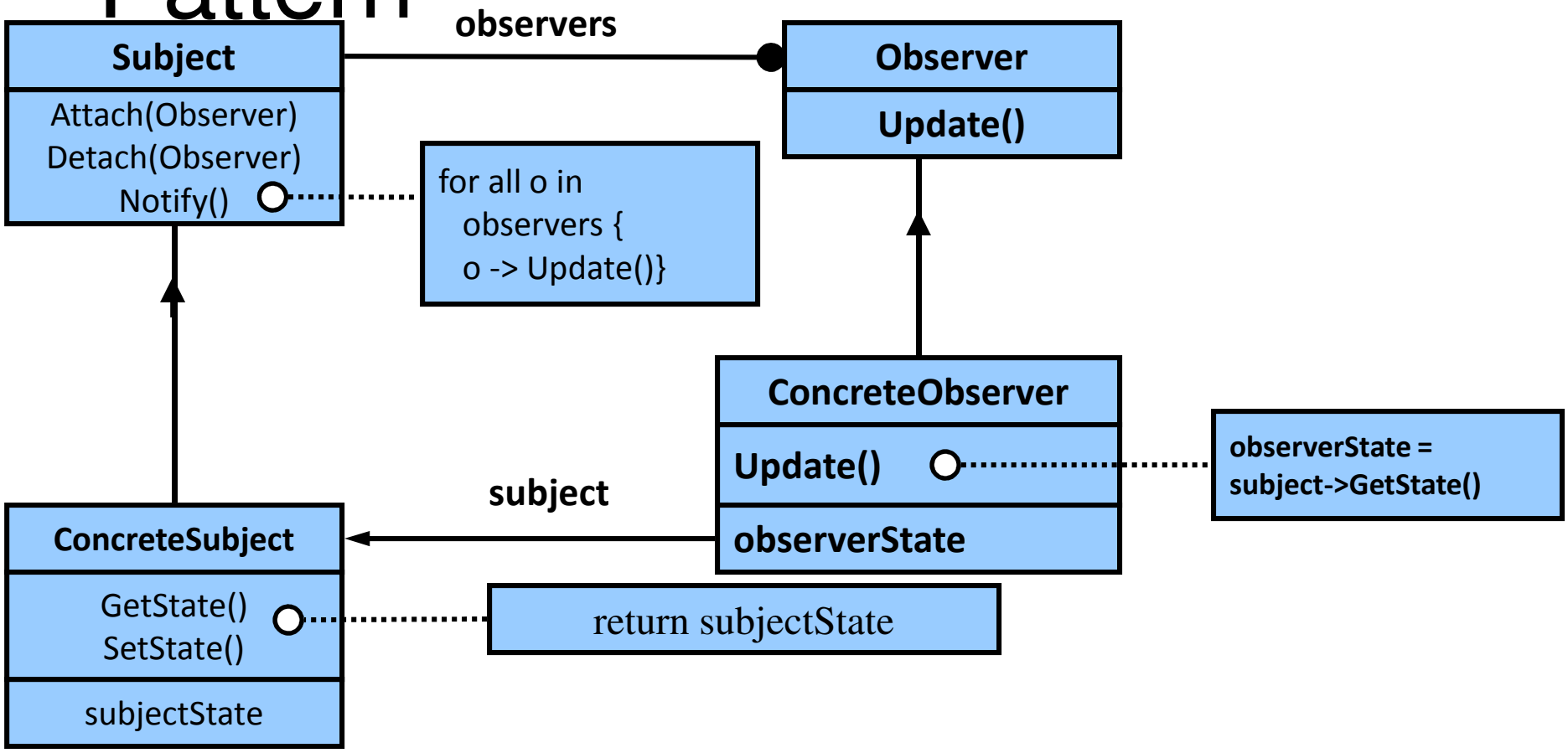
- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
- You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

Example of the Observer Pattern



Understanding patterns…

Structure of the Observer Pattern



Structure of the Observer Pattern

- The key objects in this pattern are **subject** and **observer**.
  - A subject may have any number of dependent observers.
  - All observers are notified whenever the subject undergoes a change in state.

Participants of the Observer Pattern

- **Subject:**
  - Knows its numerous observers.
  - Provides an interface for attaching and detaching observer objects.
  - Sends a notification to its observers when its state changes.
- **Observer:**
  - Defines an updating interface for concrete observers.

Participants of the Observer Pattern (Cont'd)

- **Concrete Subject:**
  - Stores state of interest to concrete observers.
- **Concrete Observer:**
  - Maintains a reference to a concrete subject object.
  - Stores state that should stay consistent with the subject's.
  - Implements the updating interface.

Strategy

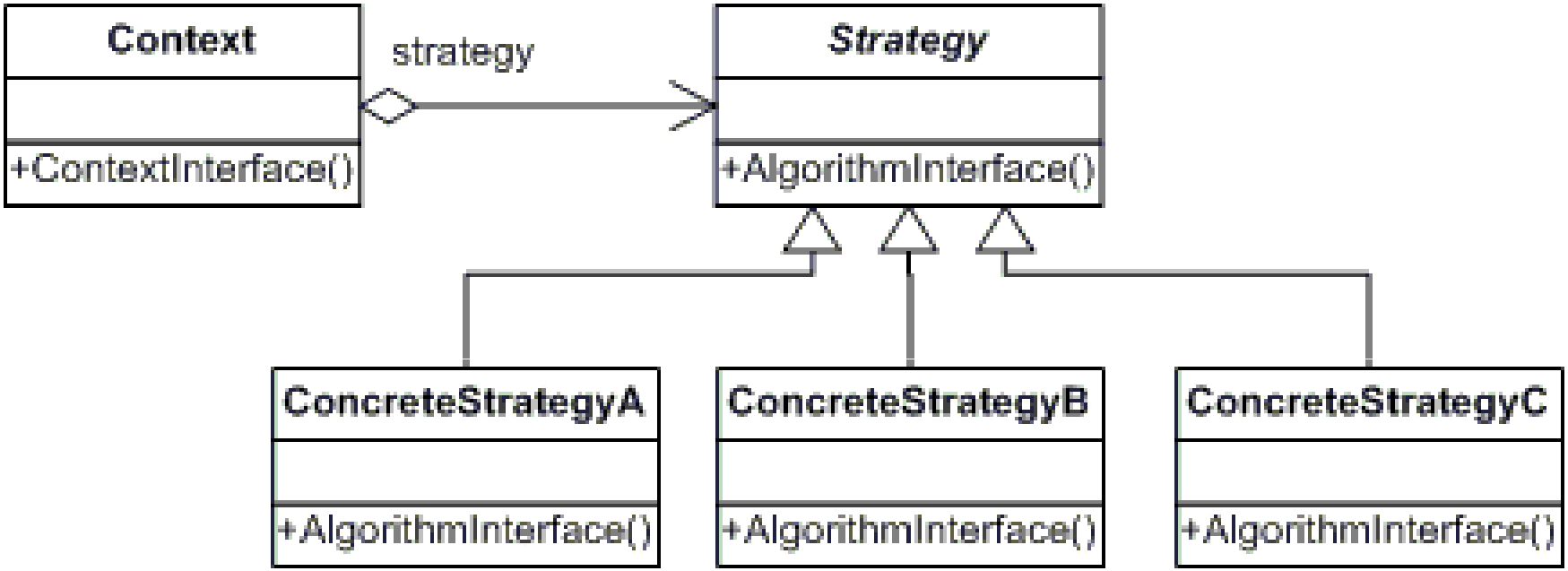
Intent

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Strategy pattern

- pulling an algorithm out from the object that contains it, and encapsulating the algorithm (the "strategy") as an object
- each strategy implements one behavior, one implementation of how to solve the same problem
  - how is this different from **Command** pattern?
- separates algorithm for behavior from object that wants to act
- allows changing an object's behavior dynamically without extending / changing the object itself

- **examples**:


Context
strategy
Strategy
+ContextInterface()
+AlgorithmInterface()
ConcreteStrategyA
+AlgorithmInterface()
ConcreteStrategyB
+AlgorithmInterface()
ConcreteStrategyC
+AlgorithmInterface()

Participants

- The classes and/or objects participating in this pattern are:
- **Strategy (SortStrategy)**
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy
- **ConcreteStrategy (QuickSort, ShellSort, MergeSort)**
  - implements the algorithm using the Strategy interface
- **Context (SortedList)**
  - is configured with a ConcreteStrategy object
  - maintains a reference to a Strategy object

participants

Strategy versus Bridge

- The UML class diagram for the Strategy pattern is the same as the diagram for the Bridge pattern. However, these two design patterns aren't the same in their *intent*. While the Strategy pattern is meant for *behavior*, the Bridge pattern is meant for *structure*.
- The coupling between the context and the strategies is tighter than the coupling between the abstraction and the implementation in the Bridge pattern.

- Difference lies in the creator of the concrete classes
  - Bridge
    - *Abstraction* creates and initializes the *ConcreteImplementations*
    - Done at initialization time
  - Strategy
    - *Client* creates the *ConcreteStrategy* objects and configures the *Context*
    - Done at run time – several times …

Bridge and Strategy look similar…

Iterator

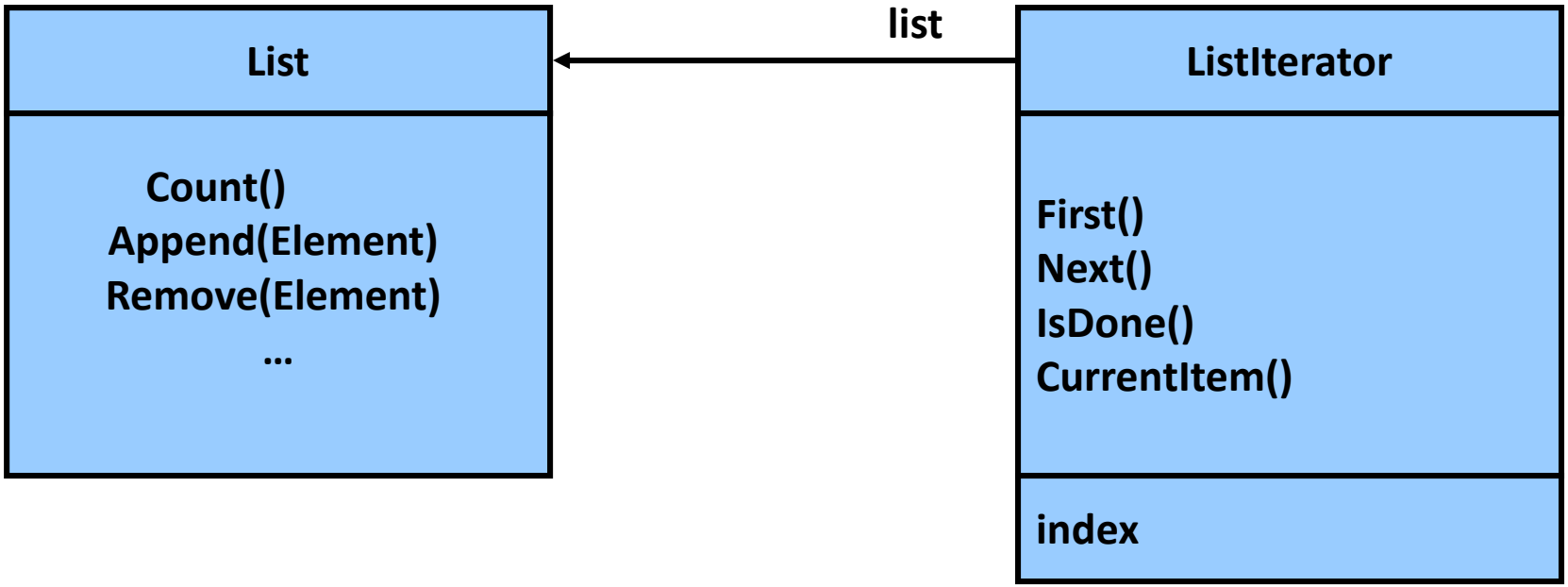
The Iterator Pattern (Intent)

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Move the responsibility for access and traversal from the aggregate object to the iterator object.

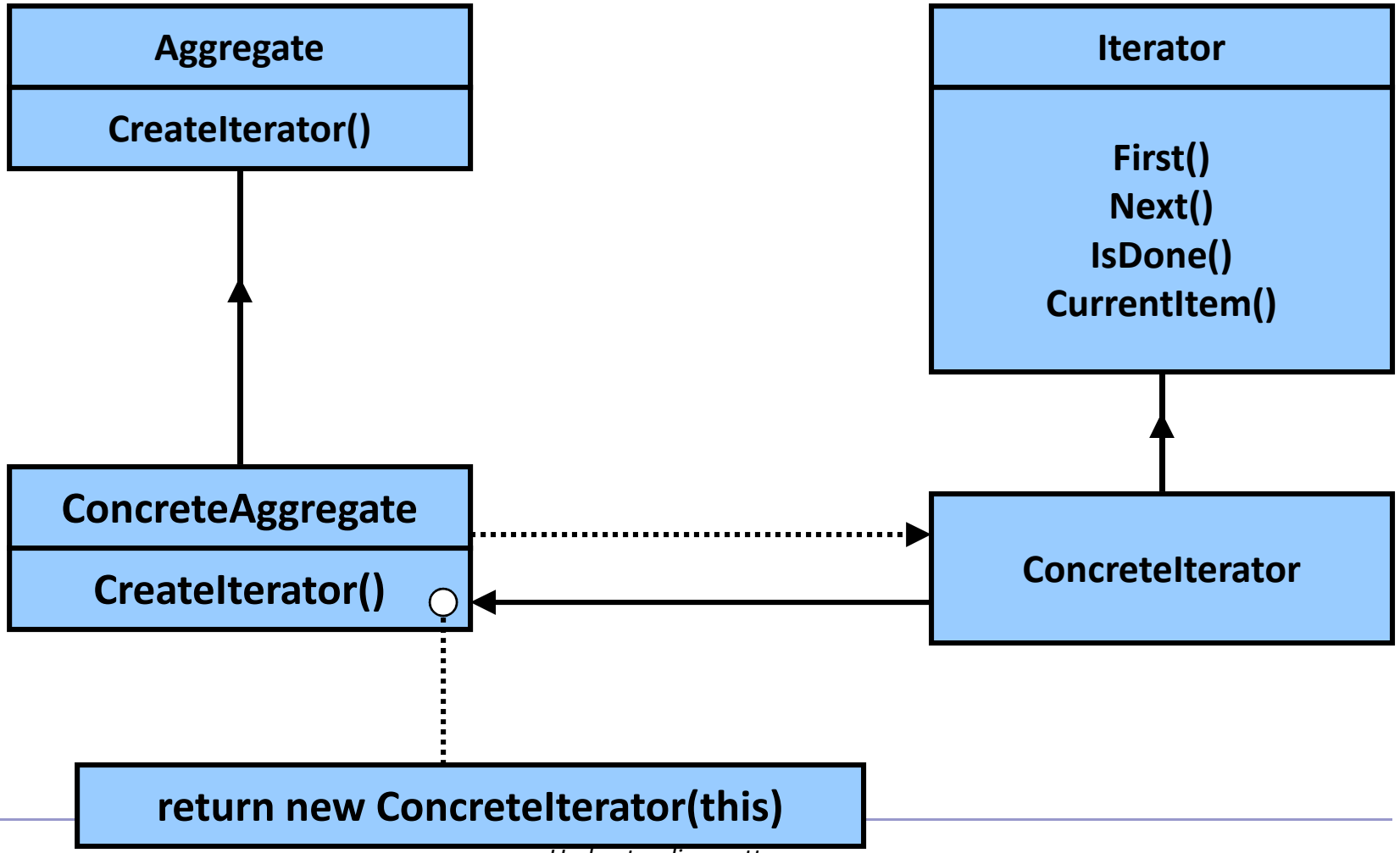
The Iterator Pattern (Motivation)

- One might want to traverse an aggregate object in different ways.
- One might want to have more than one traversal pending on the same aggregate object.
- Not all types of traversals can be anticipated a priori.
- One should not bloat the interface of the aggregate object with all these traversals.

Example of the Iterator Pattern



Structure of the Iterator Pattern



Participants of the Iterator Pattern

- **Iterator:** Defines an interface for accessing and traversing elements.
- **Concrete Iterator:** Implements an iterator interface and keeps track of the current position in the traversal of the aggregate.
- **Aggregate:** Defines an interface for creating an iterator object.
- **Concrete Aggregate:** Implements the iterator creation interface to return an instance of the proper concrete iterator.

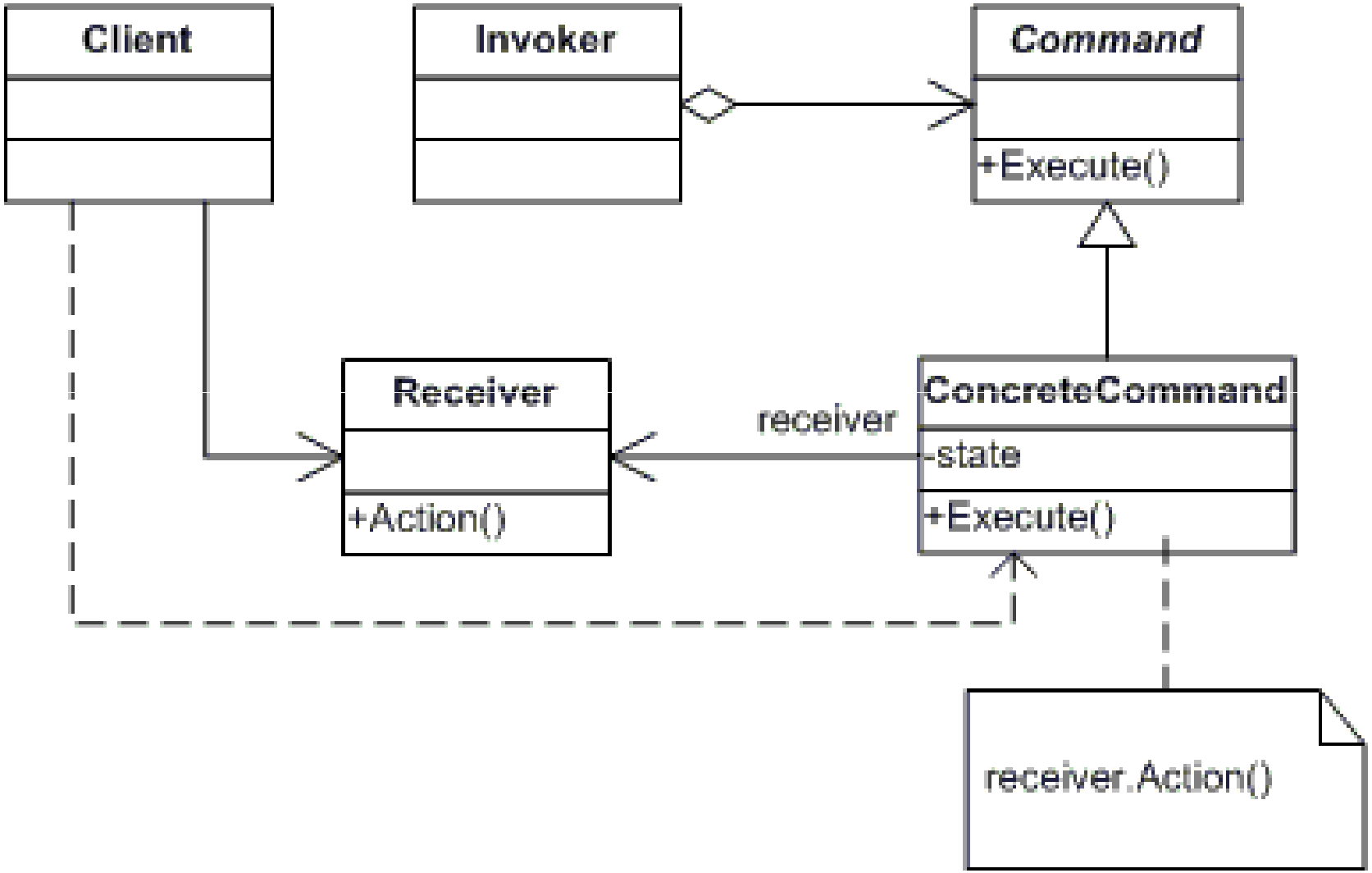
Command

Command Pattern

Intent

Encapsulate a request as an object letting you:

- Parameterize clients with different requests
- Queue or log requests
- Support undo operations


Client
Invoker
Command
+Execute()
Receiver
+Action()
receiver
ConcreteCommand
-state
+Execute()
receiver.Action()

Participants

- Command
  -- declares an interface for executing an operation
- ConcreteCommand
  -- defines a binding between a **Receiver** object and an action
  --implements execute() by invoking the corresponding operation(s) on **Receiver**
- Client (Application)
  -- creates a ConcreteCommand object and sets its receiver
- Invoker (Menuitem)
  --asks the command to carry out the request
- Receiver (Document)
  --knows how to carry out the request. (Any class may serve as Receiver)

Consequences

1. Command decouples the object that invokes the operation from the one that knows how to perform it.
2. Commands are first-class objects. They can be manipulated and extended like any other object.
3. Its easy to add new Commands, because you don't have to change existing classes.
4. Commands can be assembled into a composite command.

Uses

- Multi-level undo
  - If all user actions in a program are implemented as command objects, the program can keep a stack of the most recently executed commands. When the user wants to undo a command, the program simply pops the most recent command object and executes its undo() method.
- Transactional behavior
  - Undo is perhaps even more essential when it's called *rollback* and happens automatically when an operation fails partway through. Installers need this and so do databases. Command objects can also be used to implement two-phase commit.
- Progress bars
  - Suppose a program has a sequence of commands that it executes in order. If each command object has a getEstimatedDuration() method, the program can easily estimate the total duration. It can show a progress bar that meaningfully reflects how close the program is to completing all the tasks.
- Wizards
  - Often a wizard presents several pages of configuration for a single action that happens only when the user clicks the "Finish" button on the last page. In these cases, a natural way to separate user interface code from application code is to implement the wizard using a command object. The command object is created when the wizard is first displayed. Each wizard page stores its GUI changes in the command object, so the object is populated as the user progresses. "Finish" simply triggers a call to execute(). This way, the command class contains no user

Uses contd

- GUI buttons and menu items
  - In Swing programming, an Action is a command object. In addition to the ability to perform the desired command, an Action may have an associated icon, keyboard shortcut, tooltip text, and so on. A toolbar button or menu item component may be completely initialized using only the Action object.
- Thread pools
  - A typical, general-purpose thread pool class might have a public addTask() method that adds a work item to an internal queue of tasks waiting to be done. It maintains a pool of threads that execute commands from the queue. The items in the queue are command objects. Typically these objects implement a common interface such as java.lang.Runnable that allows the thread pool to execute the command even though the thread pool class itself was written without any knowledge of the specific tasks for which it would be used.
- Macro recording
  - If all user actions are represented by command objects, a program can record a sequence of actions simply by keeping a list of the command objects as they are executed. It can then "play back" the same actions by executing the same command objects again in sequence. If the program embeds a scripting engine, each command object can implement a toScript() method, and user actions can then be easily recorded as scripts.
- Networking
  - It is possible to send whole command objects across the network to be executed on the other machines, for example player actions in computer games.

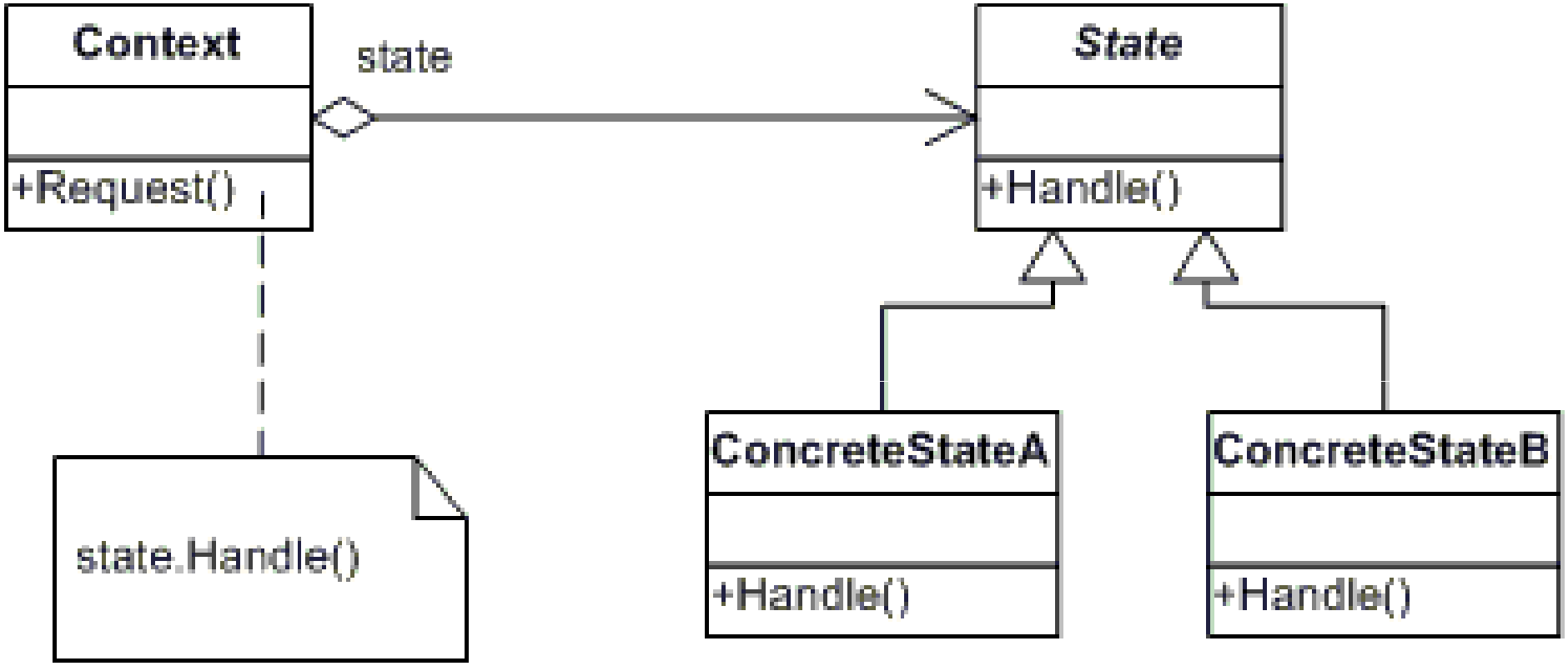
General Description

- A type of Behavioral pattern.
- Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Uses Polymorphism to define different behaviors for different states of an object.

When to use STATE pattern ?

- State pattern is useful when there is an object that can be in one of several states, with different behavior in each state.
- To simplify operations that have large conditional statements that depend on the object's state.

```
if (myself = happy) then
{
    eatIceCream();
    ….
    }
else if (myself = sad) then
{
    goToPub();
    ….
  }
else if (myself = ecstatic) then
{
    ….
```


Context
state
State
+Request()
+Handle()
state.Handle()
ConcreteStateA
+Handle()
ConcreteStateB
+Handle()

Participants

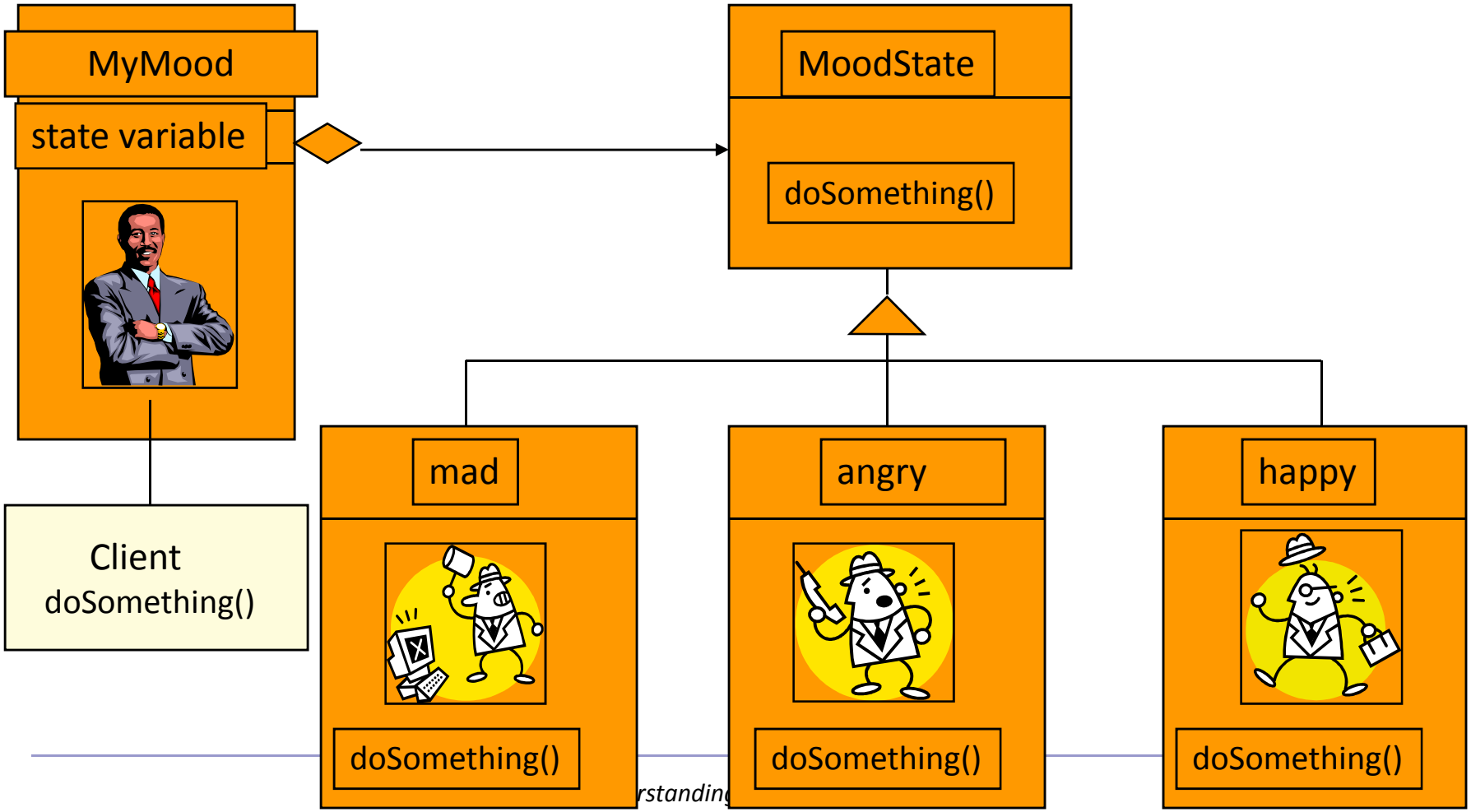
- The classes and/or objects participating in this pattern are:
- **Context (Account)**
  - defines the interface of interest to clients
  - maintains an instance of a ConcreteState subclass that defines the current state.
- **State (State)**
  - defines an interface for encapsulating the behavior associated with a particular state of the Context.
- **Concrete State (RedState, SilverState, GoldState)**
  - each subclass implements a behavior associated with

How is STATE pattern implemented ?

- "Context" class:
  Represents the interface to the outside world.
- "State" abstract class:
  Base class which defines the different states of the "state machine".
- "Derived" classes from the State class:
  Defines the true nature of the state that the state machine can be in.

Context class maintains a pointer to the current state. To change the state of the state machine, the pointer needs to be changed.

Example II


MyMood
state variable
MoodState
doSomething()
Client
doSomething()
mad
doSomething()
angry
doSomething()
happy
doSomething()

Benefits

- **Localizes all behavior associated with a particular state into one object.**
  - New state and transitions can be added easily by defining new subclasses.
  - Simplifies maintenance.
- **It makes state transitions explicit.**
  - Separate objects for separate states makes transition explicit rather than using internal data values to define transitions in one combined object.
- **State objects can be shared.**
  - Context can share State objects if there are no instance variables.

Food for thought…

- **To have a monolithic single class or many subclasses ?**
  - Increases the number of classes and is less compact.
  - Avoids large conditional statements.
- **Where to define the state transitions ?**
  - If criteria is fixed, transition can be defined in the context.
  - More flexible if transition is specified in the State subclass.
  - Introduces dependencies between subclasses.
- **Whether to create State objects as and when required or to create-them-once-and-use-many-times ?**
  - First is desirable if the context changes state infrequently.
  - Later is desirable if the context changes state frequently.

Template Method

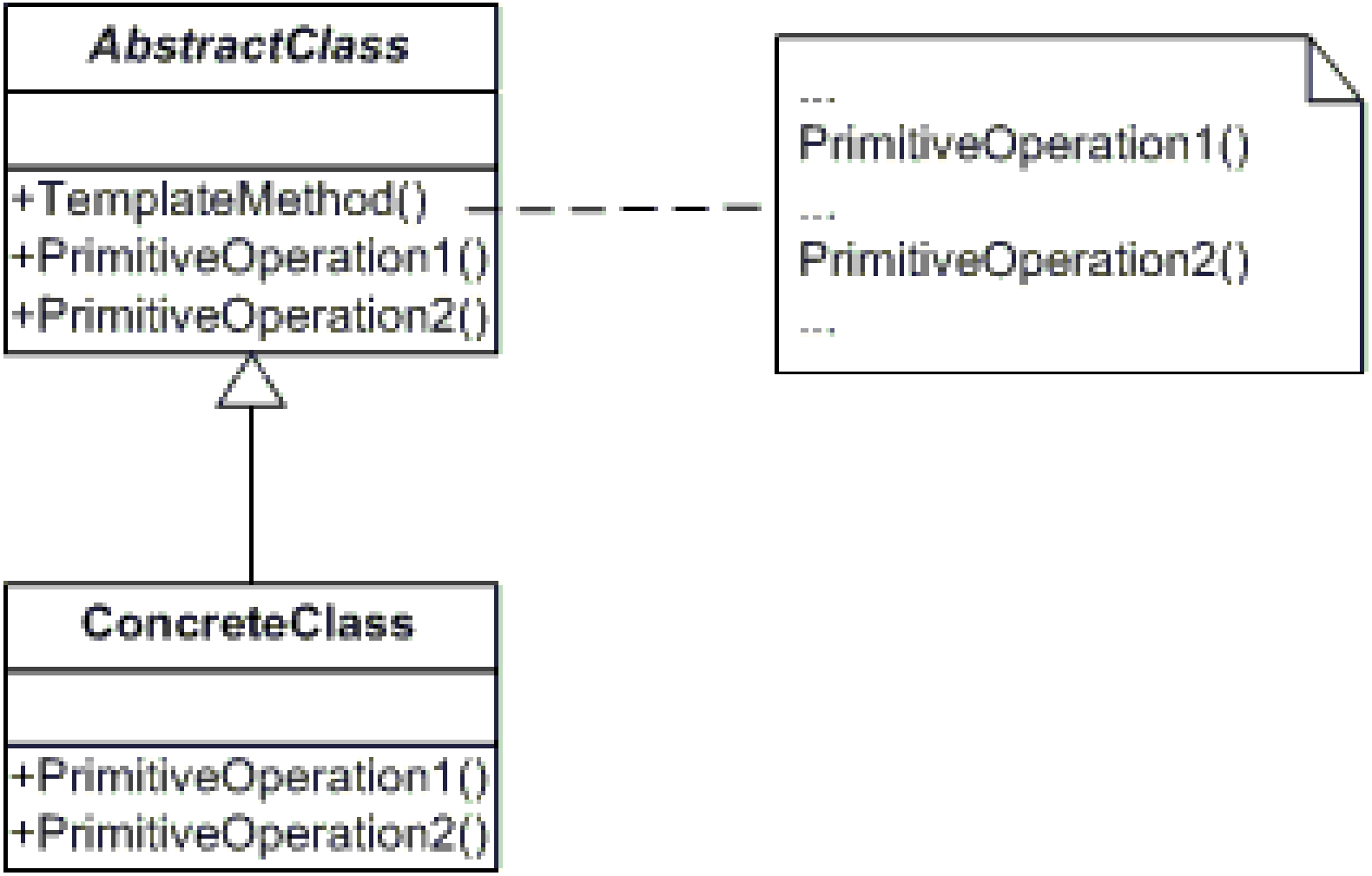
The Template Pattern (Intent)

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- The *Template Method* lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

The Template Pattern (Motivation)

- By defining some of the steps of an algorithm, using abstract operations, the template method fixes their ordering.

Structure of the Template



Structure of the Template Pattern

- **Abstract Class:**
  - Defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
  - Implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in Abstract Class or those of other objects.

Structure of the Template Pattern (Cont'd)

- **Concrete Class:** Implements the primitive operations to carry out subclass-specific steps to the algorithm.

The template method is used to:

- let subclasses implement behaviour that can vary
- avoid duplication in the code: you look for the general code in the algorithm, and implement the variants in the subclasses
- to control at what point(s) subclassing is allowed.