Design Patterns

Walter F. Tichy
Karlsruhe Institute of Technology

Contents

Definition

A Software Design Pattern describes

a family of solutions to a software design problem.

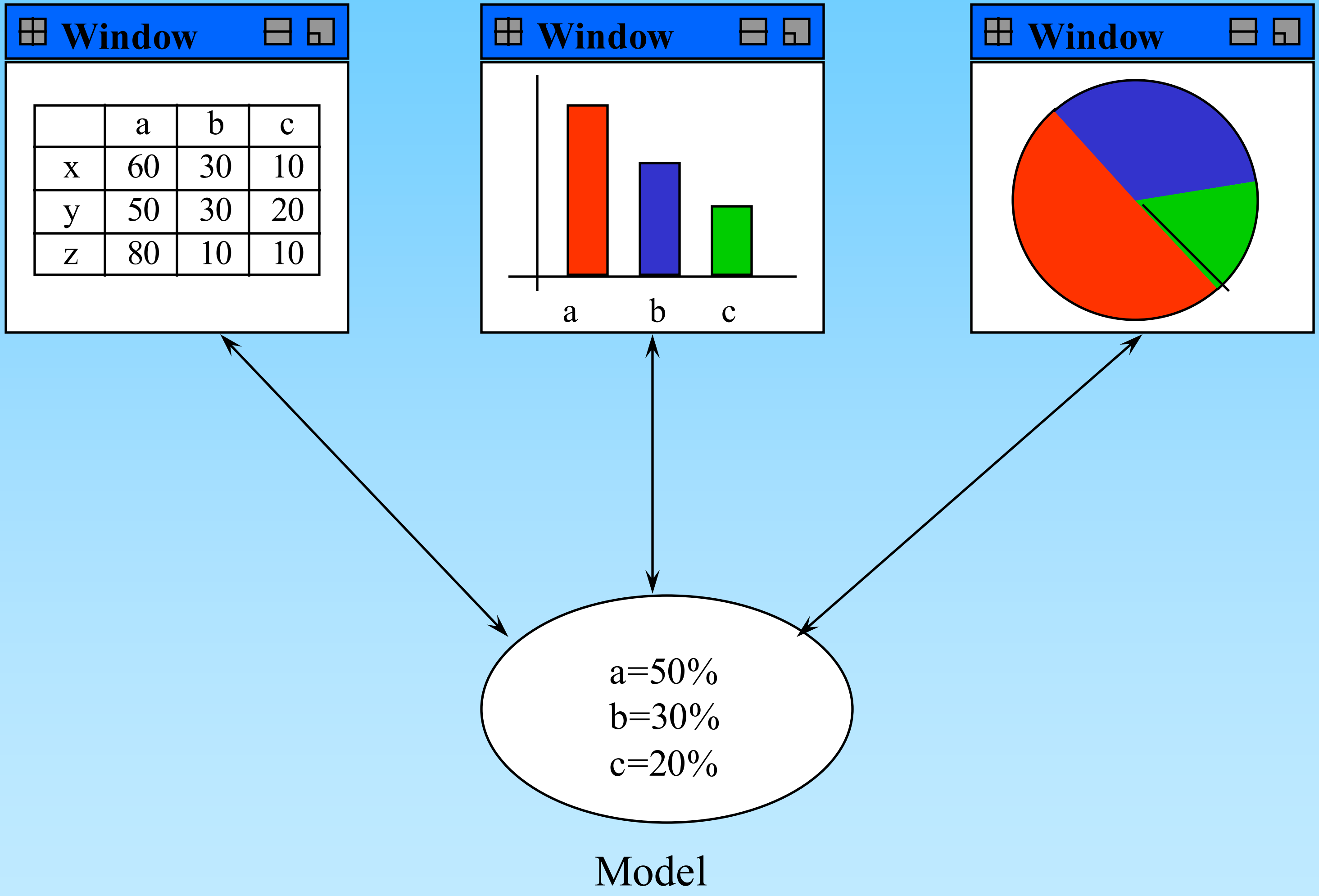
A software design pattern consists of

- design elements such as interfaces, classes, objects, methods, threads, etc.,
- relationships among the elements such as association, inheritance, delegation, creation, invocation, etc.,
- and a behavioral description.

The description is intended to facilitate reuse of the solution.

Design patterns are to design what algorithms are to programming.

Pattern 1 in MVC: Observer (Views)



	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

Example: Model/View/Controller (MVC)

MVC is a set of classes for building user interfaces.

- *Model*: application data structure
- *View*: screen presentation of model (possibly several)
- *Controller*: defines reaction of user interface to input

MVC is a compound design pattern that combines several other patterns: *Observer*, *Composite*, and *Strategy*.

Pattern 1 in MVC: Observer (Views)

Pattern 1: Model & views constitute the *Observer* pattern.

- There is a subscribe/notify protocol between the model (the “subject”) and the views (the “observers.”)
- Whenever the model’s data changes, it notifies subscribed views. In response, each view accesses the model’s data and updates itself.
- The views do not know anything about each other. The model does not know how the views use its data (decoupling).

Pattern 2 in MVC: Composite

Pattern 2: Views can be nested.

A composite view is an example of the *Composite* pattern.

- For example, an object inspector can consist of nested views and may itself be contained in the user interface of a debugger.
- A composite view is a subclass of the **`view`** class. A composite view object can be used just like a view object, but it also contains and manages nested views.

Pattern 3 in MVC: Strategy

Pattern 3: The view-controller relationship is an example of the *Strategy* pattern.

- There exist several controller subclasses that implement different response strategies. For instance, they may treat keyboard input differently or use a pop-up menu instead of command keys.
- A view chooses a particular controller subclass that implements a particular response strategy; it might even choose them dynamically (for instance for ignoring input in a disabled state.)

What Are Design Patterns Good For?

1) Improve team communication
2) Capture design essentials in concrete form
3) Record and encourage “best practices”
4) Improve quality of code, productivity of developers

1) Improved team communication

- Design patterns provide an effective “shorthand” or terminology and shared problem/solution understanding for communicating complex concepts effectively between designers

2) They capture the essentials of a design in concrete form

- Patterns help understand designs,
- document designs concisely,
- avoid architectural drift,
- make design knowledge explicit.

3) Patterns can record and encourage "best practices"

- Patterns help and teach less experienced designers.
- A pattern is not a fixed rule to be followed blindly, but a guide and a set of alternatives for solving a design problem.

4) Patterns can improve the quality/quantity of code produced

- Patterns encourage good design and good code by presenting positive examples.

Good patterns are difficult and time-consuming to invent

- Difficult to spot and extract.
- Recording them is an iterative process requiring application and refinement.
- Practice helps!

Patterns are not necessarily object-oriented!

What does a Design Pattern Look Like?

- A name (plus synonyms)
- A problem statement including its context
- A description of the solution
  - Structure (components, connections)
  - Tradeoffs and consequences
  - Implementation
  - Sample code

The Observer Pattern

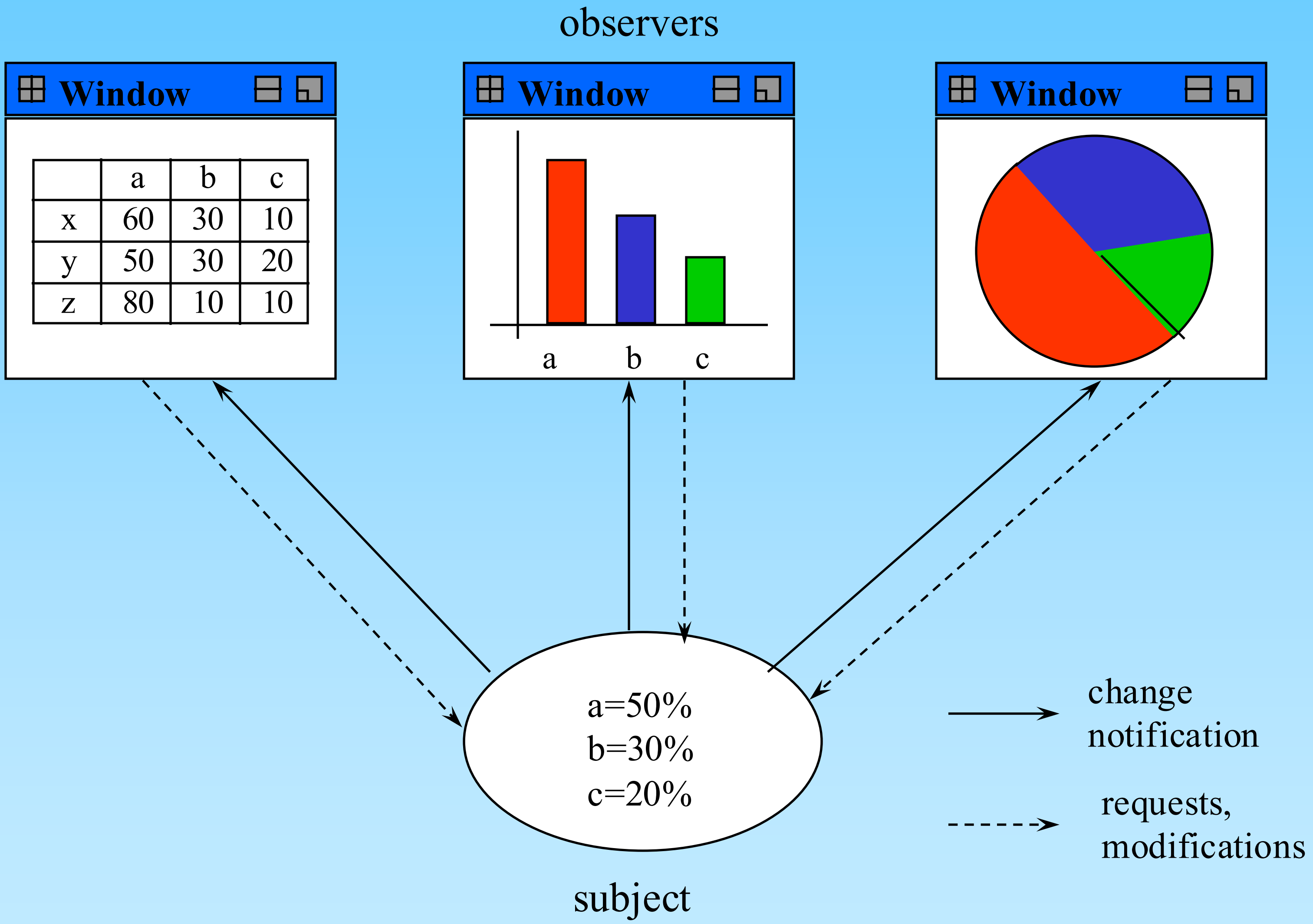
Intent

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

Also known as

Dependents, Publish-Subscribe, Subject-Observer, Listener

The Observer Pattern



	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

The Observer Pattern

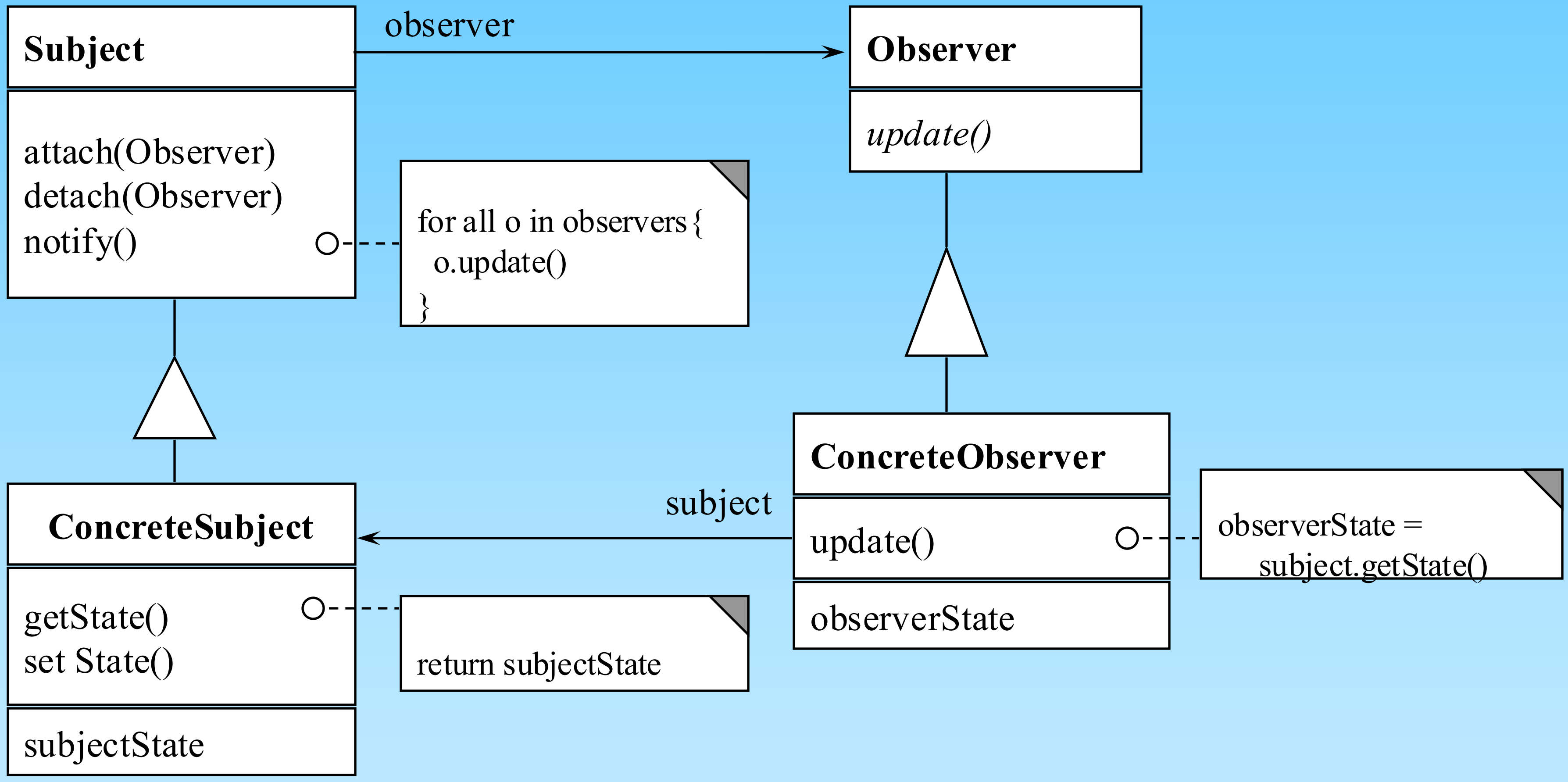
Motivation

When partitioning a system into a set of cooperating classes, consistency between related objects must be maintained.

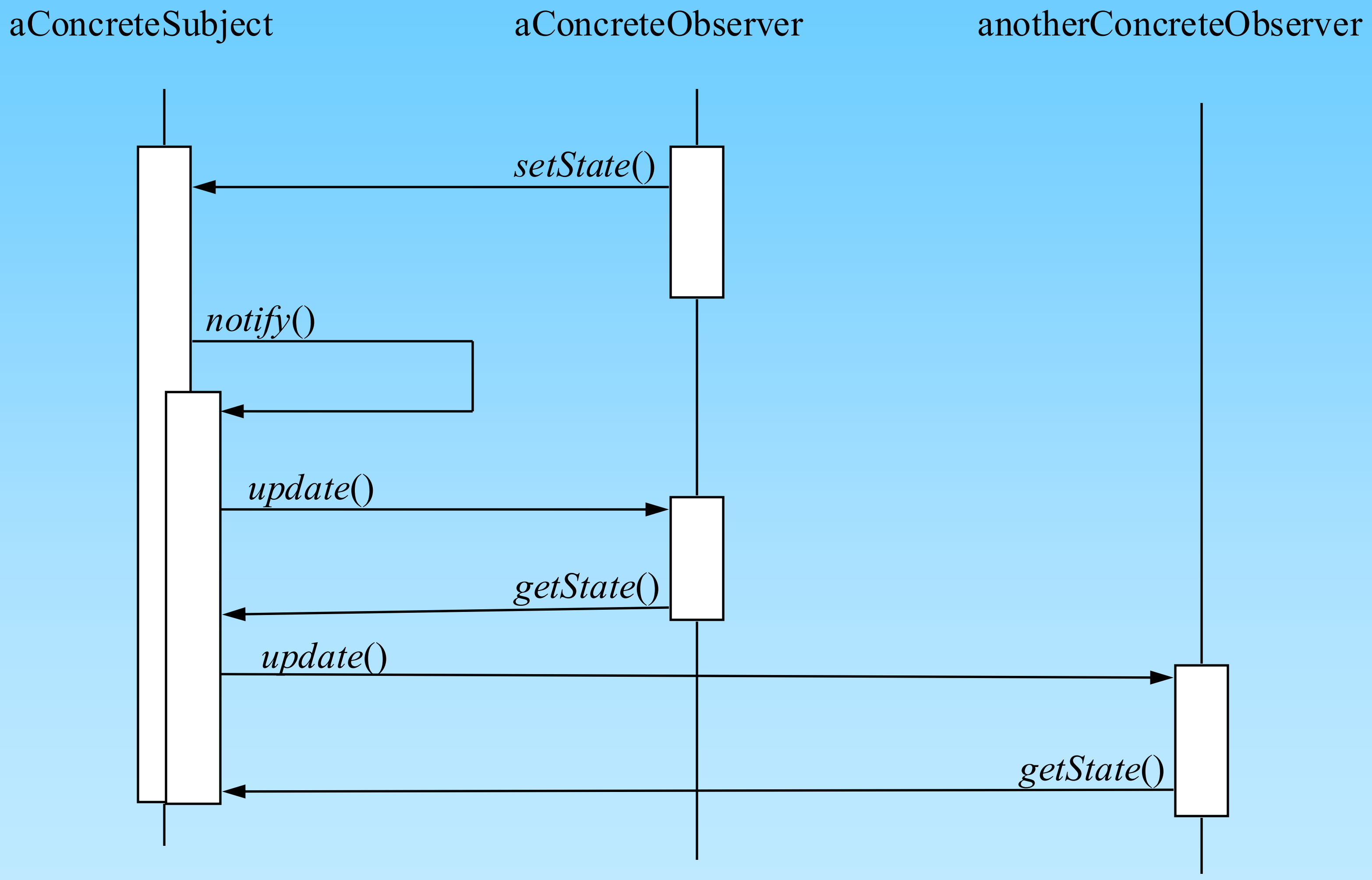
It is not desirable to couple the classes tightly (every class knowing which other classes to update and how), because then the classes can't be reused individually.

In the MVC example, the spreadsheet view and the bar chart view don't know about each other, so they can be reused independently. A change to the spreadsheet is immediately reflected in the bar chart and vice versa.

Structure of Observer



Collaborations of Observer



The Observer Pattern

Applicability

- When a change to an object requires changing others, and you don’t know how many or which ones.
- When an object needs to notify others but can’t make assumptions about these objects.
- When an abstraction has two aspects, one depending on the other. Encapsulating these aspects in separate objects allows independent reuse.

The Observer Pattern

<u>Consequences</u>

- Subjects and observers can be reused independently.
- Observers can be added/removed without modifying the subject or other observers.
- Abstract coupling between subject and observer is achieved via the notify interface. Subject and observers can therefore belong to different layers in the USES-hierarchy, without introducing cycles (Subject does not use Observer in the sense of the USES-relation, because the subject's integrity does not depend on the presence or correct operation of observers).

The Observer Pattern

Consequences (cont.)

- Automatic broadcast of changes.
- Observers can decide whether to ignore a notification.
- Cost of an update may be hidden
  - A single notification may cascade to many observers and their dependent objects.
  - Notify provides no hint on *what* changed. Additional protocol may be necessary to help observers detect what changed (Subject does not use Observer).

The Observer Pattern

<u>Implementation</u>

1. Observing more than one subject: Send subject as parameter in ***`notify(Subject s)`***.
2. Triggering update:
   - part of ***`setState()`***,
   - clients call ***`notify()`*** explicitly.
3. Deleting an observer: first unsubscribe it at its subjects.

The Observer Pattern

Implementation (cont.)

4. Subject must be self-consistent before notification. When a subject subclass calls inherited operations, it may unintentionally trigger a notify from the superclass before the subclass object is completely consistent.

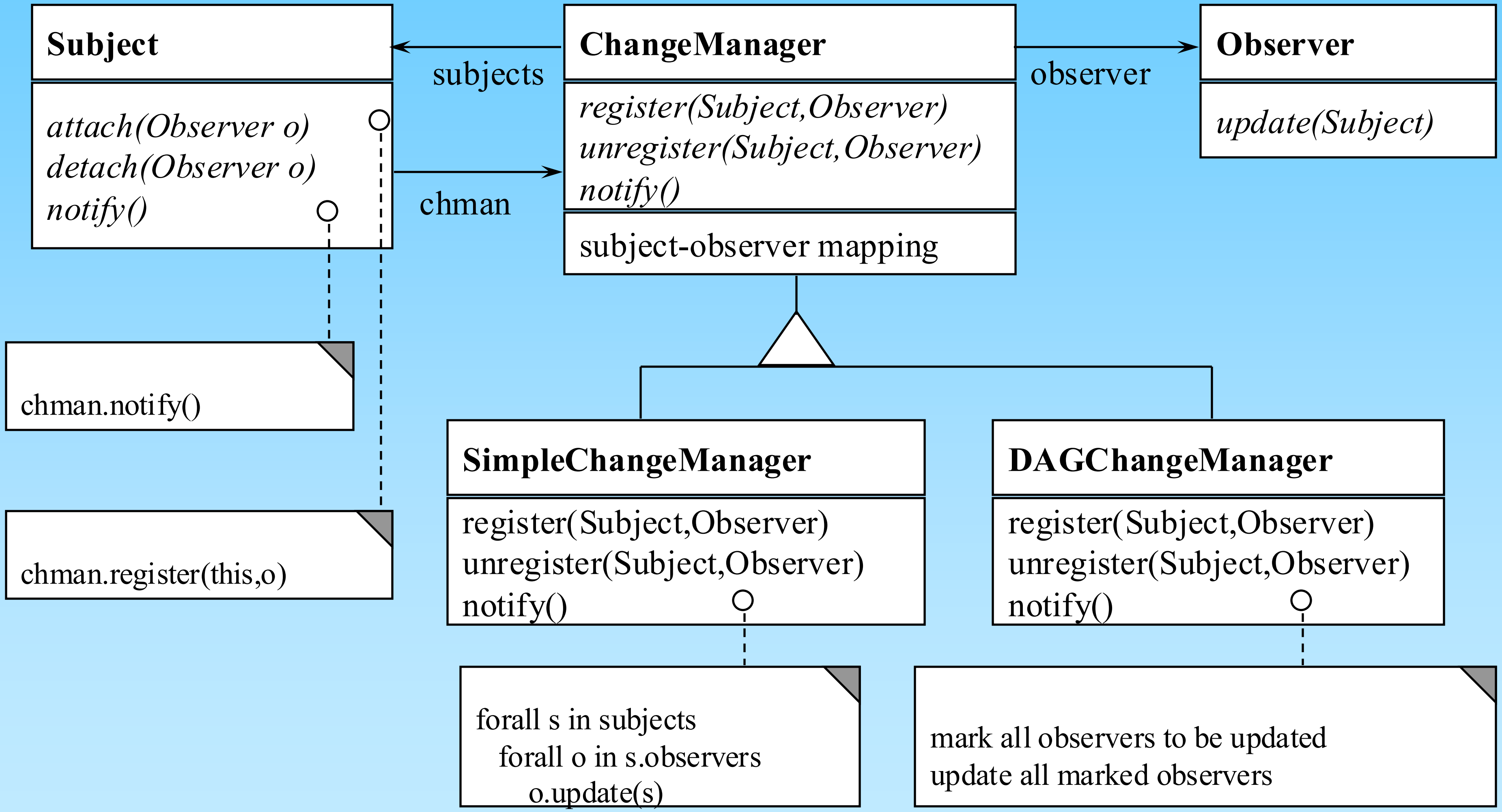
5. Update protocols: pull and push model
   - Pull model: Observer fetches all data from subject directly (may be inefficient)
   - Push model: Subject passes change data to observers in ***`notify()`*** (may reduce reusability)

The Observer Pattern

Implementation (cont.)

6. Use Change Manager between subject and its observers:
   - maintains a mapping of subjects to observers,
   - updates all dependent observers at the request of a subject,
   - avoids multiple updates and infinite recursion,
   - encapsulates complex update semantics.

Observer with ChangeManager



The Composite Pattern

Intent

Represents part-whole hierarchies (aggregates). Composite lets clients treat individual objects and composites of objects uniformly.

The Composite Pattern

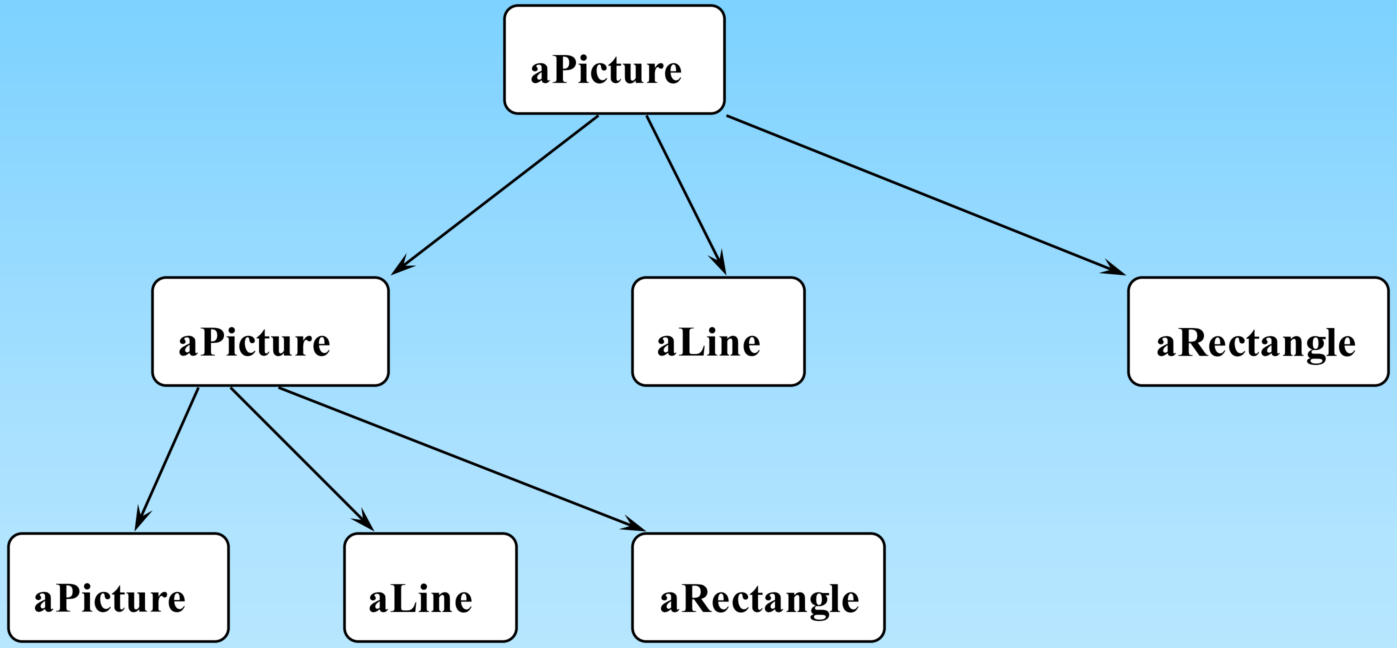
Motivation

Part-whole hierarchies occur whenever complex objects are modeled, e.g. file systems, graphics applications, word processing, CAD, CIM, ... In these applications, primitive objects are combined into groups which may again be combined to form still larger groups.

Often, client programs wish to treat primitive and composite objects (aggregates) uniformly. The composite pattern factors out the common aspects of both primitives and aggregates into a common superclass.

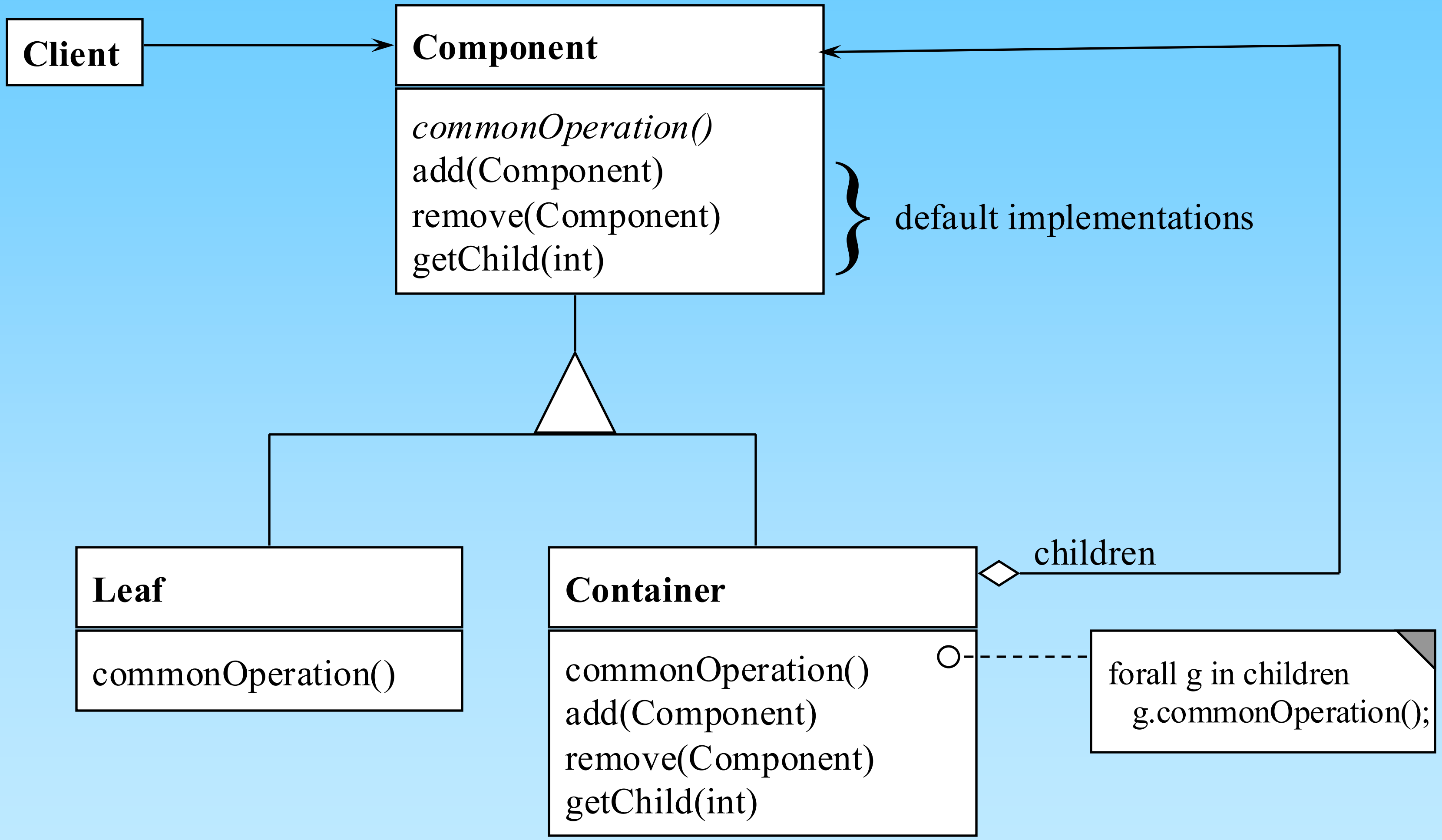
Example of Composite

composition of graphic objects:

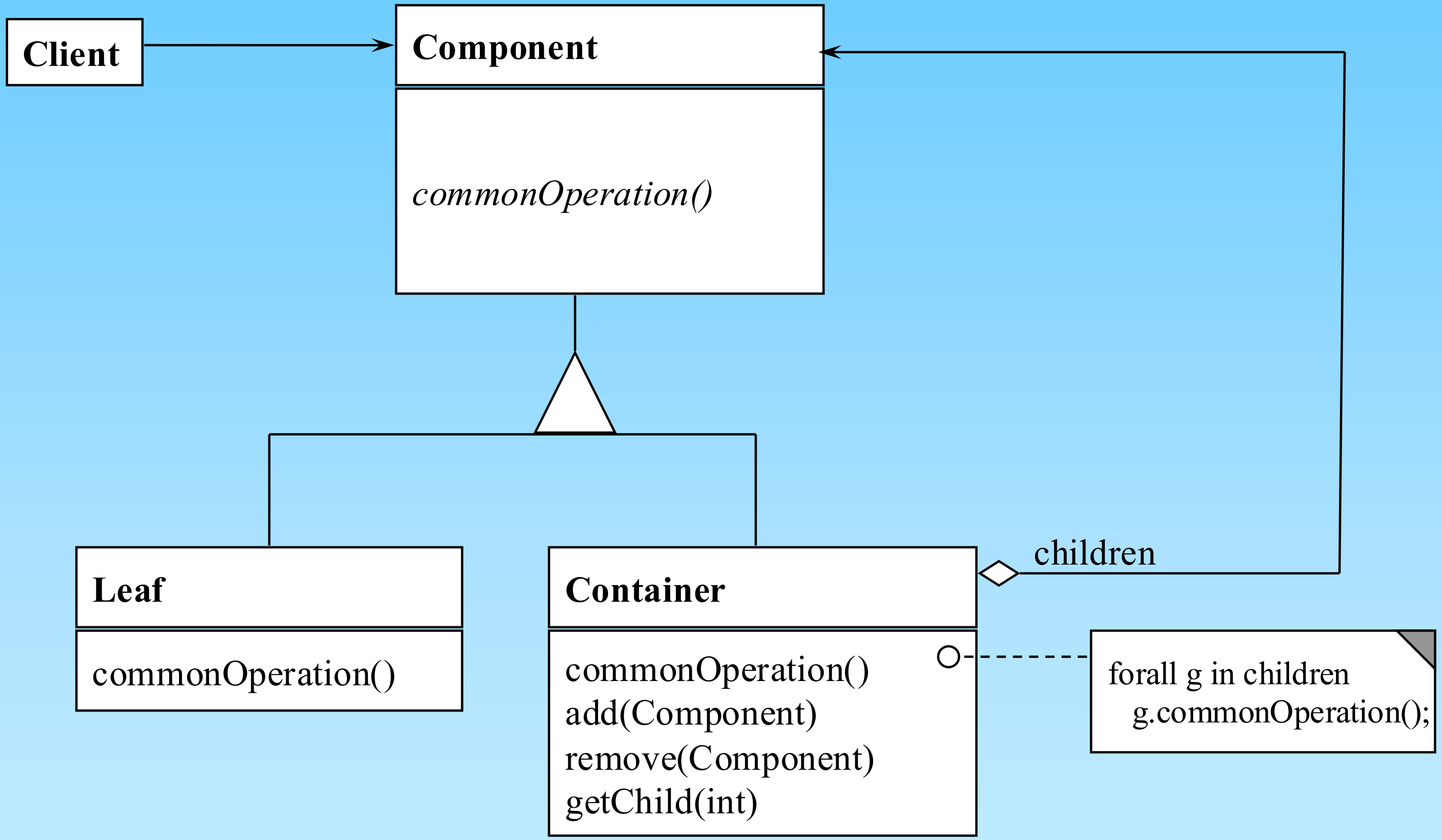


common operations: draw(), move(), delete()

Structure of Composite1: Container Operations in Component



Structure of Composite 2 :
Container Operations in Container only



The Composite Pattern

Consequences

- The composite pattern defines hierarchies of both primitive objects (leaves) and composite objects (containers).
- Simplifies clients: the same code can handle leaves and containers objects.
  - Container operations for leaves and containers: apply common and container methods to leaves and containers transparently.
  - Container operations for containers only: test for container instance before applying container methods; common methods apply to all.
- New kinds of components (both leaves and containers) can be added without changing client code.
- Must use runtime checks to restrict the components of a container (can´t use type checking).

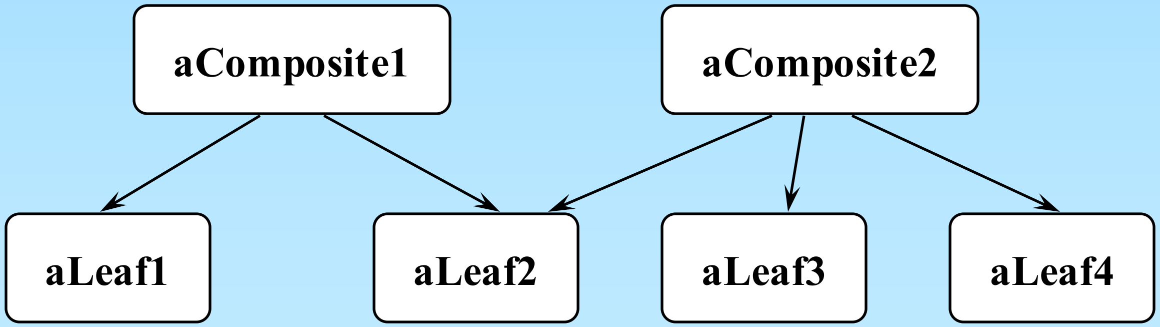
The Composite Pattern

Implementation

1. It is often useful to maintain a parent reference in each component (simplifies traversal).

   Of course, the parent reference must point to the true parent. This invariant can be maintained by the ***add*** and ***remove*** operation for composites.

   Sharing components causes multiple parents (ambiguity).



The Composite Pattern

Implementation (cont.)

2. Maximize the Component Interface

The component interface should define as many common methods of containers and leaves as possible, to ensure transparency.

When container methods are defined for all components:

`getChild` should return nothing for leaves - this can be the default implementation in component.

`addChild` and ***`removeChild`*** should fail for leaves (return an error or generate an exception).

The Composite Pattern

<u>Implementation (cont.)</u>

3. Storing children:

arrays, lists, hashtables - choice depends on efficiency.

For fixed set of children, use explicit variables (and specialized ***`add`*/*`remove`*/*`getChild`*** operations).

Children may also have to be maintained in a specific order (for instance, for layout managers).

The Strategy Pattern

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Algorithms can vary independent of clients that use them.

Also known as

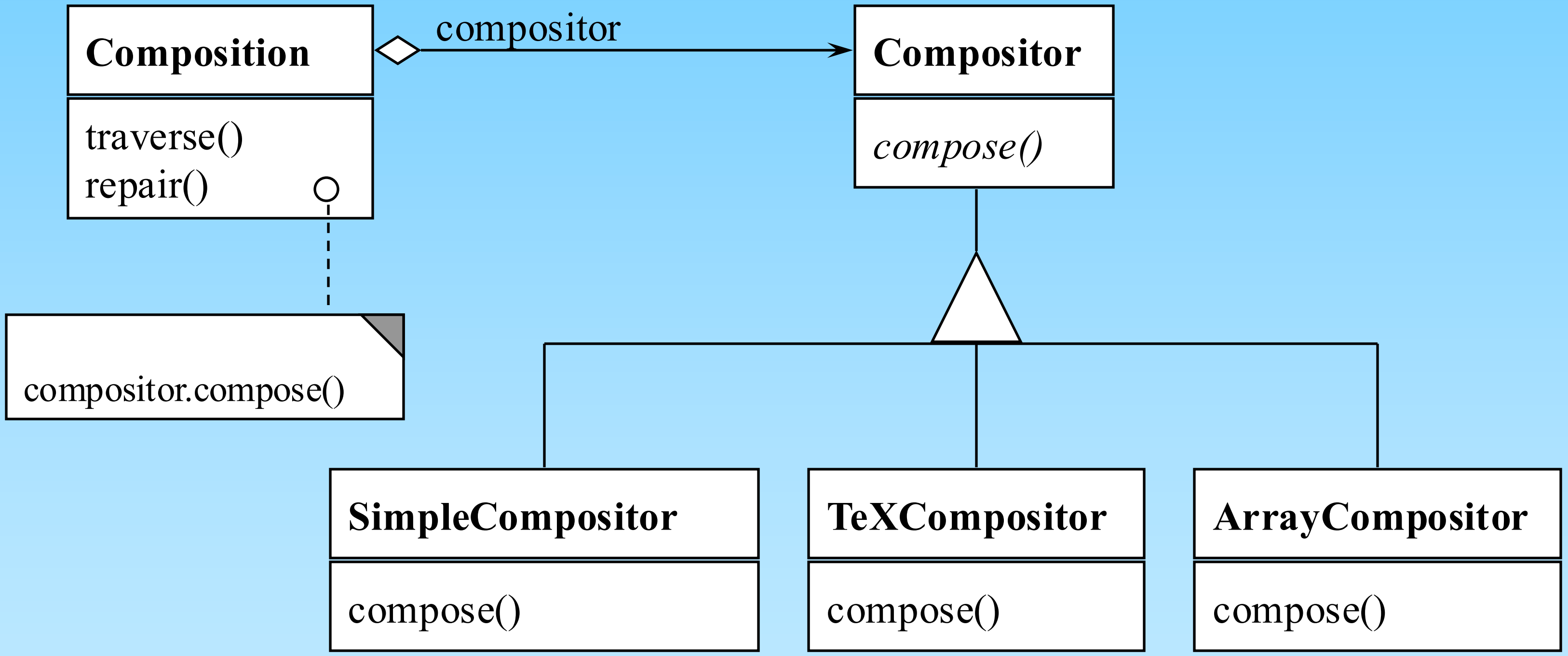
Policy

Motivation

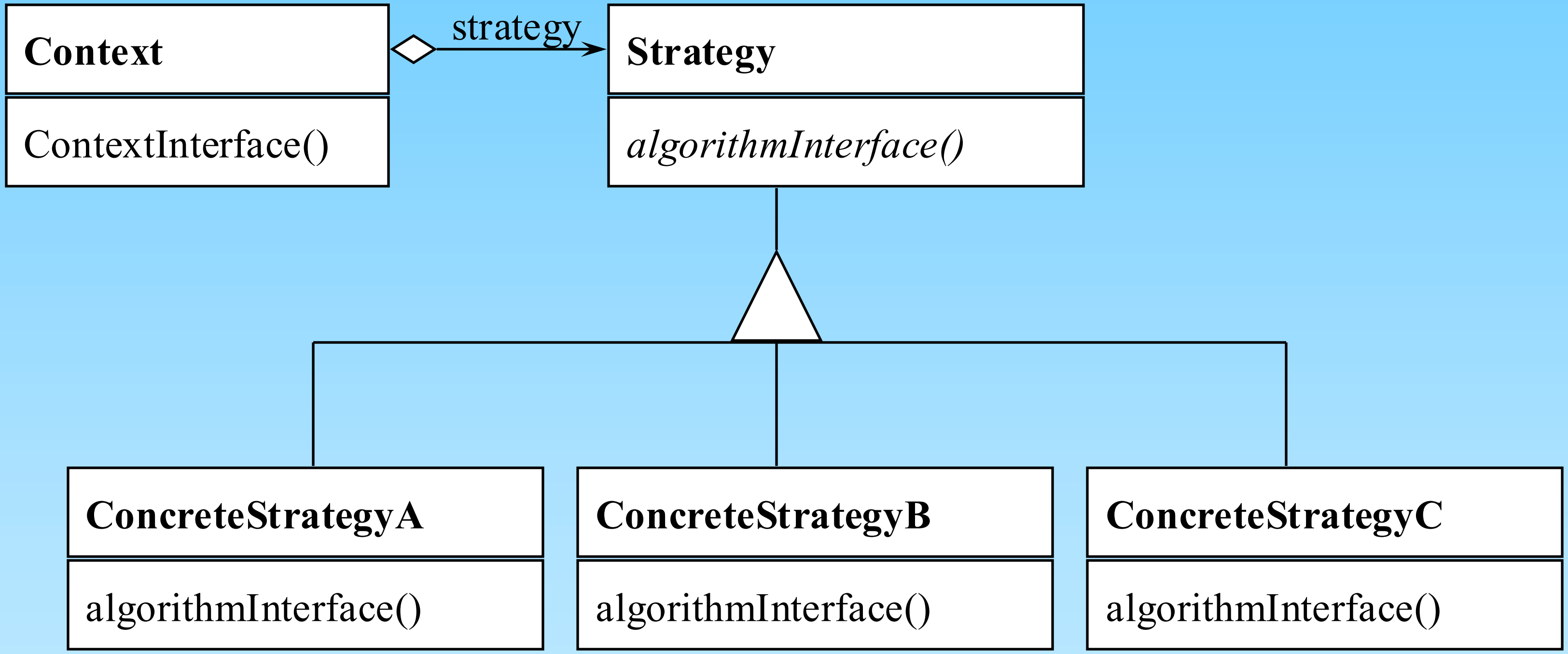
Sometimes algorithms need to vary, depending on size of data, performance required, or type of data.

Example of Strategy

Encapsulation of line breaking algorithms in classes



Structure of Strategy



How to Use Strategy

```
Strategy S;    // variable of type Strategy
               // assume Strategy has methods m1(), m2()

S = new ConcreteStrategyA();   // choose a strategy

S.m1(); S.m2();    // use ConcreteStrategyA

S = new ConcreteStrategyB();   // substitute strategy

S.m1(); S.m2();    // use ConcreteStrategyB with same code


// Thanks, Barbara Liskov, for the substitution principle!
```

The Strategy Pattern

Applicability

- When several classes differ only in their behavior. Strategies can configure a class with one of many behaviors.
- When different variants of algorithms are needed, e.g. reflecting different space/time/quality tradeoffs.
- When algorithm-specific data structures can be hidden.
- Switchless programming: When a class defines many behaviors as multiple conditional statements or switches. Move related branches into their own strategy class. Then the code for every single case is contained in a single class, which results in a cleaner design. Also, the classes for the cases can be worked on independently.

Switch-less Programming

```
m1() {
  switch(…) {
    case 1: code m1.1;
     ⋮
    case i: code m1.i;
     ⋮
    case n: code m1.n;
}}
m2() {
  switch(…) {
    case 1: code m2.1;
     ⋮
    case i: code m2.i;
     ⋮
    case n: code m2.n;
}}
m3() {
  switch(…) {
    case 1: code m3.1;
     ⋮
    case i: code m3.i;
     ⋮
    case n: code m3.n;
}}   ⋮
```

- Can be transformed into n strategies with methods `m1`, `m2`, `m3`

```
class Strategy_i extends Strategy {
  m1() { code m1.i }
  m2() { code m2.i }
  m3() { code m3.i }
}
```

- Context contains a single switch-statement:

```
Strategy suitableStrategy;
switch(…) {
   ⋮
  case i: suitableStrategy = new Strategy_i();
   ⋮
}

// Use
suitableStrategy.m1()
   ⋮
suitableStrategy.m2()
   ⋮
//etc.
```

The Strategy Pattern

Applicability (cont.)

- Alternative to strategy: subclass the class Context: One can subclass Context directly to obtain different behaviors.
- This results in many classes that only differ in behavior and the behavior is fixed for each class.
- Strategy, on the other hand, factors out only the behavior, and allows dynamic change of behavior.

Where can I find More Design Patterns?

- “Design Patterns”, Gamma et al, Addison Wesley,1995. Deutsche Ausgabe: “Entwurfsmuster”, Riehle.
- “A System of Patterns”, Buschmann et al, Wiley, 1996
- “Pattern Languages of Program Design”, conference proceedings, Addison-Wesley, 1995, 1996, 1998, ...
- “Pattern Hatching”, John Vlissides, Addison-Wesley, 1998. Deutsch: “Entwurfsmuster awenden”, 1999.
- “Patterns in Java”, Mark Grand, Wiley, 1998.
- “Pattern-Oriented Software Architecture”, Schmidt et al, Wiley, 2000.

Summary

Design Patterns

- provide vocabulary and shared understanding for efficient communication among designers,
- encapsulate best design practices,
- document designs,
- make novices into good designers,
- make good designers better,
- improve quality and productivity,
- are difficult to find and describe.

Design Pattern Categories

0. Architectural Patterns
1. Decoupling Patterns
2. Variant Management Patterns
3. State Handling Patterns
4. Control Patterns
5. Virtual Machines
6. Convenience Patterns
7. Compound Patterns
8. (Concurrency Patterns — not covered)
9. (Distribution Patterns — not covered)

1. Decoupling

Definition

Decoupling Patterns divide a software system into several independent parts in such a way that they can be built, changed, and replaced independently as well as reused and recombined in unforeseen combinations. An important advantage of decoupling is local change, i.e., a system consisting of decoupled parts can be adapted and extended by modifying or adding a single or only a few parts, instead of modifying everything.

Examples

Module, Abstract Data Type

2. Variant Management

Definition

Variant management patterns treat different but related objects uniformly by factoring out their commonality. All of the variant patterns rely on features found in object-oriented programming languages.

Examples

Superclass, Strategy, Composite

3. State Handling

Definition

State Handling patterns manipulate the state of objects generically. This means that these patterns work on the state of any object or class, independent of the actual purpose of these objects/classes.

Examples

Memento, Prototype, Singleton

4. Control

Definition

Control patterns deal with control of execution and selecting the right methods at the right time.

Examples

Blackboard, Command, Chain of Responsibility

5. Virtual Machines

Definition

A virtual machine simulates a processor. It is software that interprets a program written in a specific language.

Examples

Interpreter, Rule-based Interpreter

6. Convenience Patterns

Definition

Convenience patterns simplify code by shortening method invocations or concentrating often-repeated code sequences in one place.

Examples

Convenience Method, Null Object

7. Compound Patterns

Definition

Compound patterns are composed from others, with the original patterns visible to the client programs. Most patterns are actually composed of others, but if a particular combination takes on a different purpose, then it is categorized according to this purpose. Because compound patterns consist of several visible sub-patterns, compound patterns can usually be classified into several categories at once. To avoid the duplication, they are placed in this category.

Examples

Model/View/Controller, Bureaucracy

1. Decoupling

- Module
- Abstract Datatype
  - Repository
    - Client/Server
  - Manager
  - Iterator
  - Collection
    - Set
    - Bag
    - Sequence
- Layers
  - Sandwich
  - Facade
  - Mediator
  - Bridge
  - Adapter (Wrapper)
  - Facet (Extension Object)
  - Proxy
    - Decorator
    - Buffer Proxy
    - Logging (Counting) Proxy
    - Firewall
    - Synchronization Proxy
    - Remote Access Proxy

1. Decoupling(cont.)

- Pipeline
- Event Notification
  - Event Handler (Catch and Throw)
  - Callback
  - Event Loop
  - Event Channel
  - Propagator
    - Strict Propagator with/without failure
    - Lazy Propagator
    - Adaptive Propagator
    - Observer
- Framework

2. Variant Management

- Superclass (Family)
  - Strategy
  - Composite
  - Bridge
- Visitor
  - Default Visitor
  - Extrinsic Visitor
  - Acyclic Visitor
- Template Method
  - Factory Method
  - Builder
- Abstract Factory

3. State Handling

- Memento
- Prototype (Cloneable)
- Flyweight
- Singleton

4. Control

- Blackboard
- Command
- Chain of Responsibility
- Strategy
- Control State
- Master/Worker
- Process Control
  - Open-loop system
  - Closed-loop system
    - Feedback control
    - Feedforward control

5. Virtual Machines

- Interpreter
  - Emulator
- Rule-based Interpreter

6. Convenience Patterns

- Convenience Method
- Convenience Class
- Default Class
- Null Object (Stub)

7. Compound Patterns

Model/View/Controller

Bureaucracy

The Design Pattern Catalogue

The following slides discuss the main patterns in the catalogue.

1. Decoupling

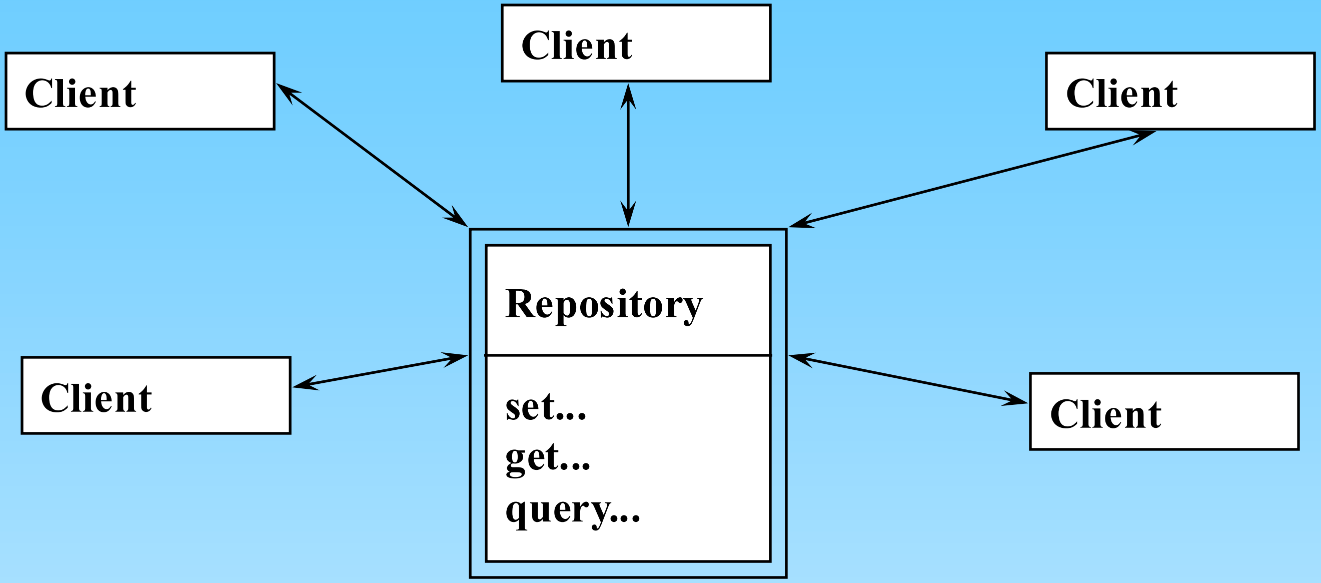
- Module
- Abstract Datatype
  - Repository
    - Client/Server
  - Manager
  - Iterator
  - Collection
    - Set
    - Bag
    - Sequence
- Layers
  - Sandwich
  - Facade
  - Mediator
  - Bridge
  - Adapter (Wrapper)
  - Facet (Extension Object)
  - Proxy
    - Decorator
    - Buffer Proxy
    - Logging (Counting) Proxy
    - Firewall
    - Synchronization Proxy
    - Remote Access Proxy

Repository

Intent

A set of independent clients communicate with a central data structure, into which they put, or from which they get elements, or ask queries.

Structure of Repository



Examples for complex Repositories:
Data bases, Hypertext systems.
Additional features compared to simple repositories:
persistence, access control, transaction mechanism

Iterator

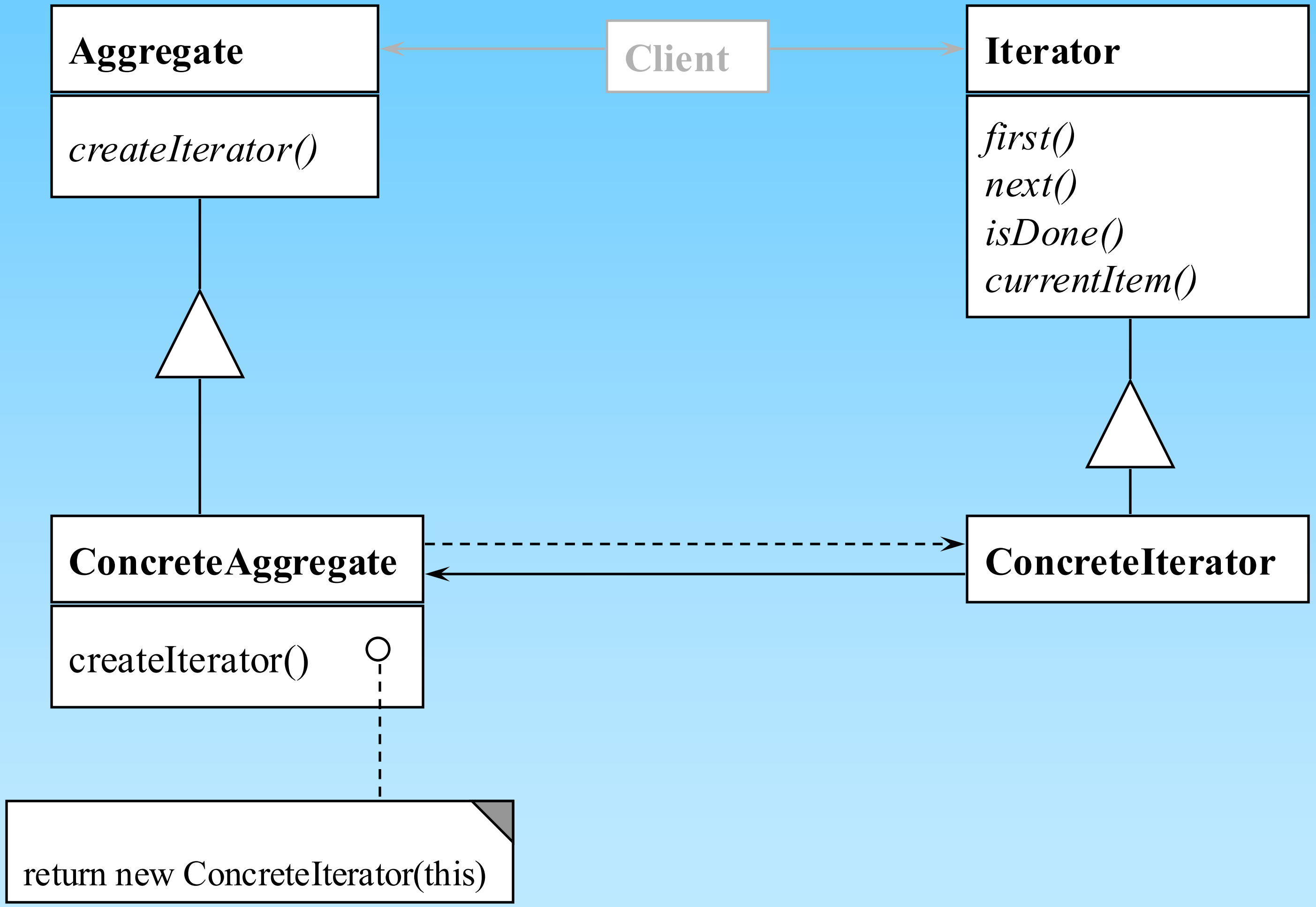
<u>**Intent**</u>

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

<u>**Also Known As**</u>

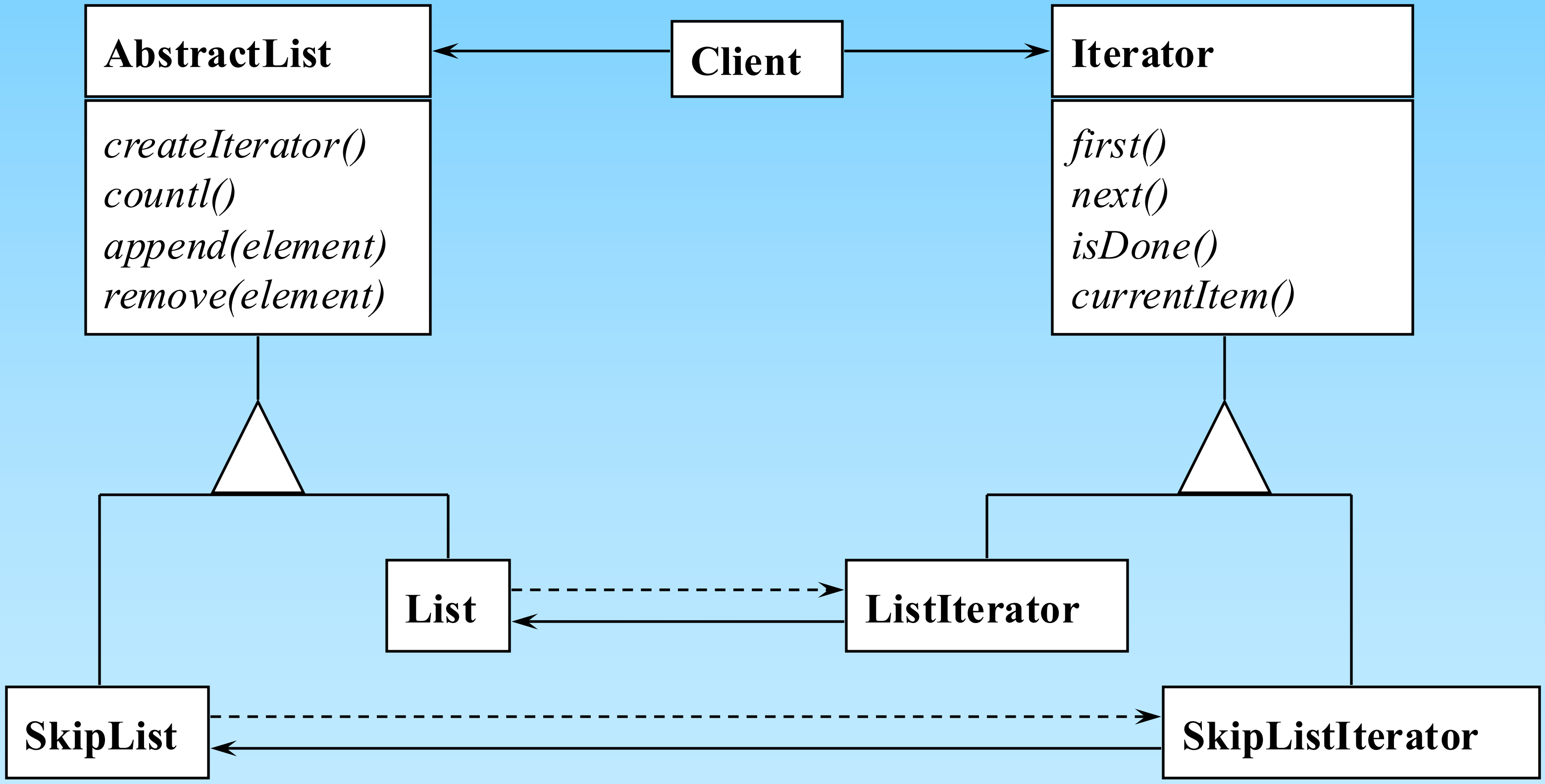
Robust Iterator, Enumeration

Structure of Iterator



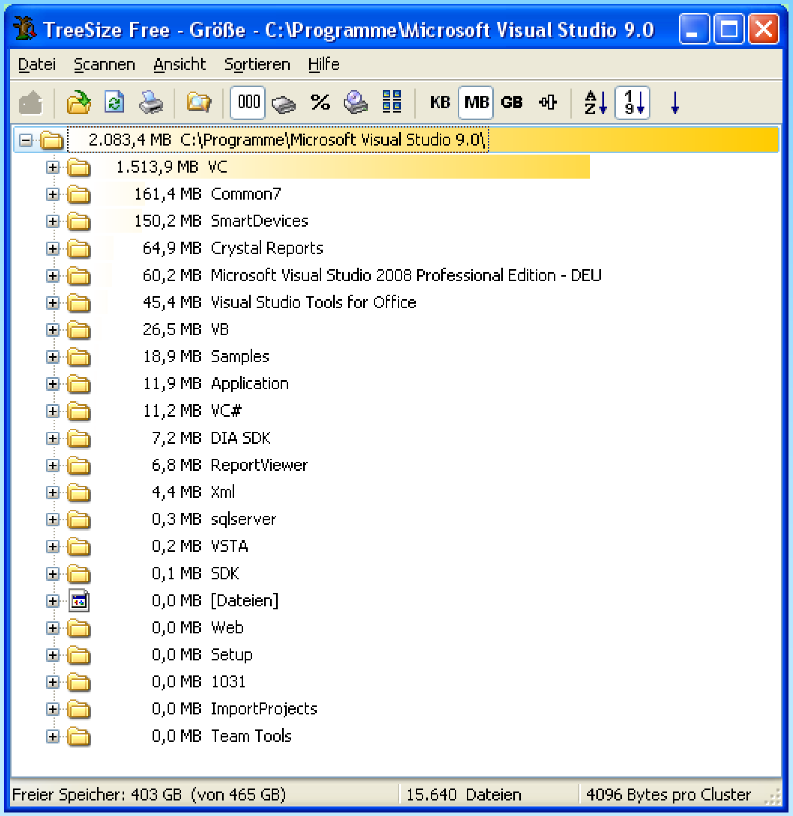
Example of Iterator

Separation of list implementation
and list traversal implementation



Iterator: Example Application

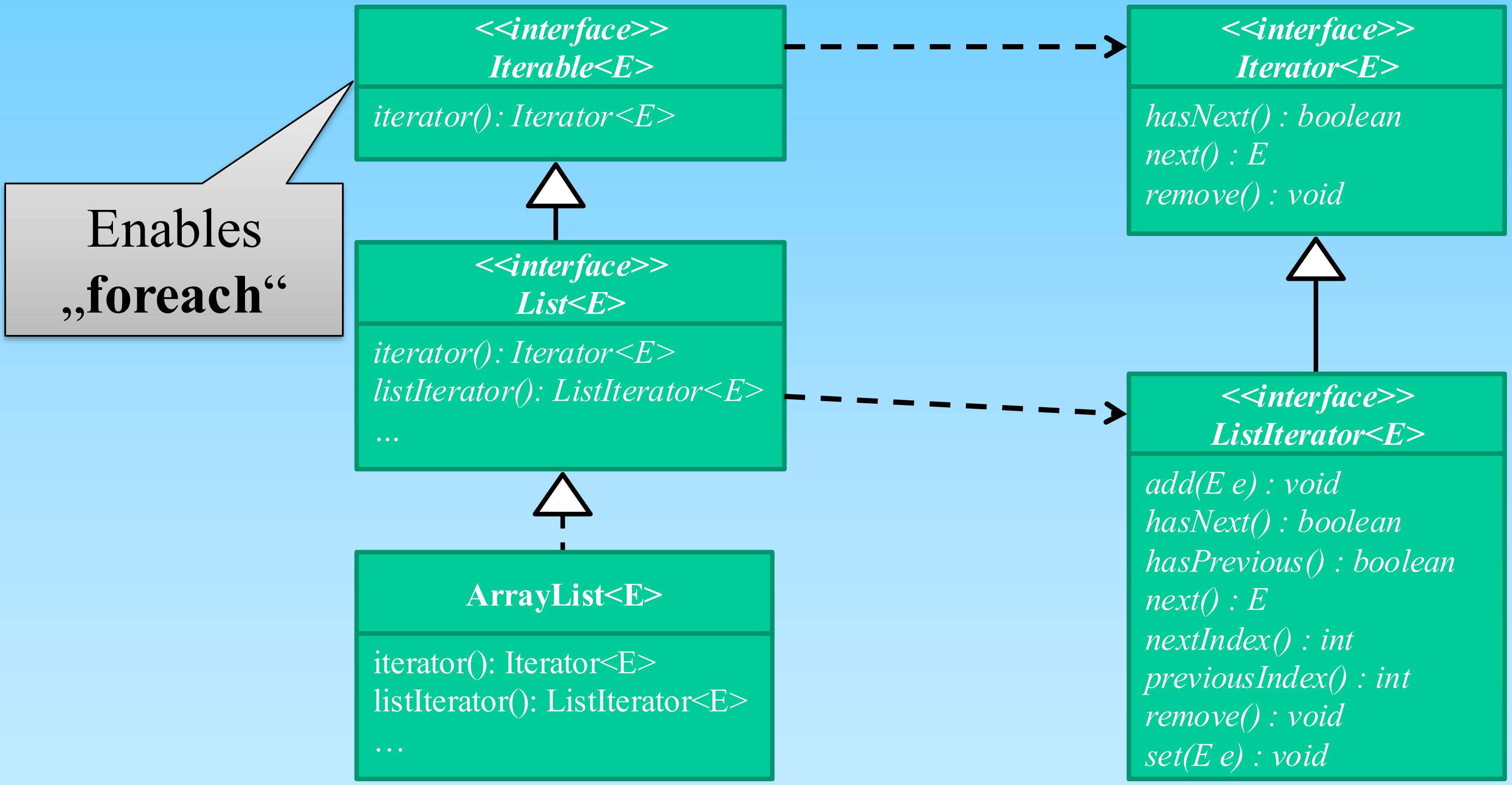
Compute storage size of directory


TreeSize Free - Größe - C:\Programme\Microsoft Visual Studio 9.0
Datei
Scannen
Ansicht
Sortieren
Hilfe
KB
MB
GB

```
2.083,4 MB  C:\Programme\Microsoft Visual Studio 9.0\
   1.513,9 MB  VC
     161,4 MB  Common7
     150,2 MB  SmartDevices
      64,9 MB  Crystal Reports
      60,2 MB  Microsoft Visual Studio 2008 Professional Edition - DEU
      45,4 MB  Visual Studio Tools for Office
      26,5 MB  VB
      18,9 MB  Samples
      11,9 MB  Application
      11,2 MB  VC#
       7,2 MB  DIA SDK
       6,8 MB  ReportViewer
       4,4 MB  Xml
       0,3 MB  sqlserver
       0,2 MB  VSTA
       0,1 MB  SDK
       0,0 MB  [Dateien]
       0,0 MB  Web
       0,0 MB  Setup
       0,0 MB  1031
       0,0 MB  ImportProjects
       0,0 MB  Team Tools
Freier Speicher: 403 GB  (von 465 GB)   15.640 Dateien   4096 Bytes pro Cluster
```

- Use an iterator that descends the directory hierarchy in some order (depth or breadth first)
- `currentItem()` returns files and directories; client program determines their sizes and adds them up.

Iterator: Implementation in Java (1)



Iterator: Implementation in Java (2)

Variant 1:

```
ArrayList<String>
stringArrayList =
  new ArrayList<String>();

Iterator<String> iter =
  stringArrayList.iterator();

while(iter.hasNext()) {
  System.out.println(
    iter.next()
  );
}
```

Variant 2:

```
ArrayList<String>
stringArrayList =
  new ArrayList<String>();

for(String str: stringArrayList)
{
  System.out.println(str);
}
```

Allowed, because `ArrayList` implements interface `Iterable`

Iterator

Applicability

- To access an aggregate object´s contents without exposing its internal representation.
- To support multiple traversals of aggregate objects (a “robust” iterator allows multiple traversals)
- To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

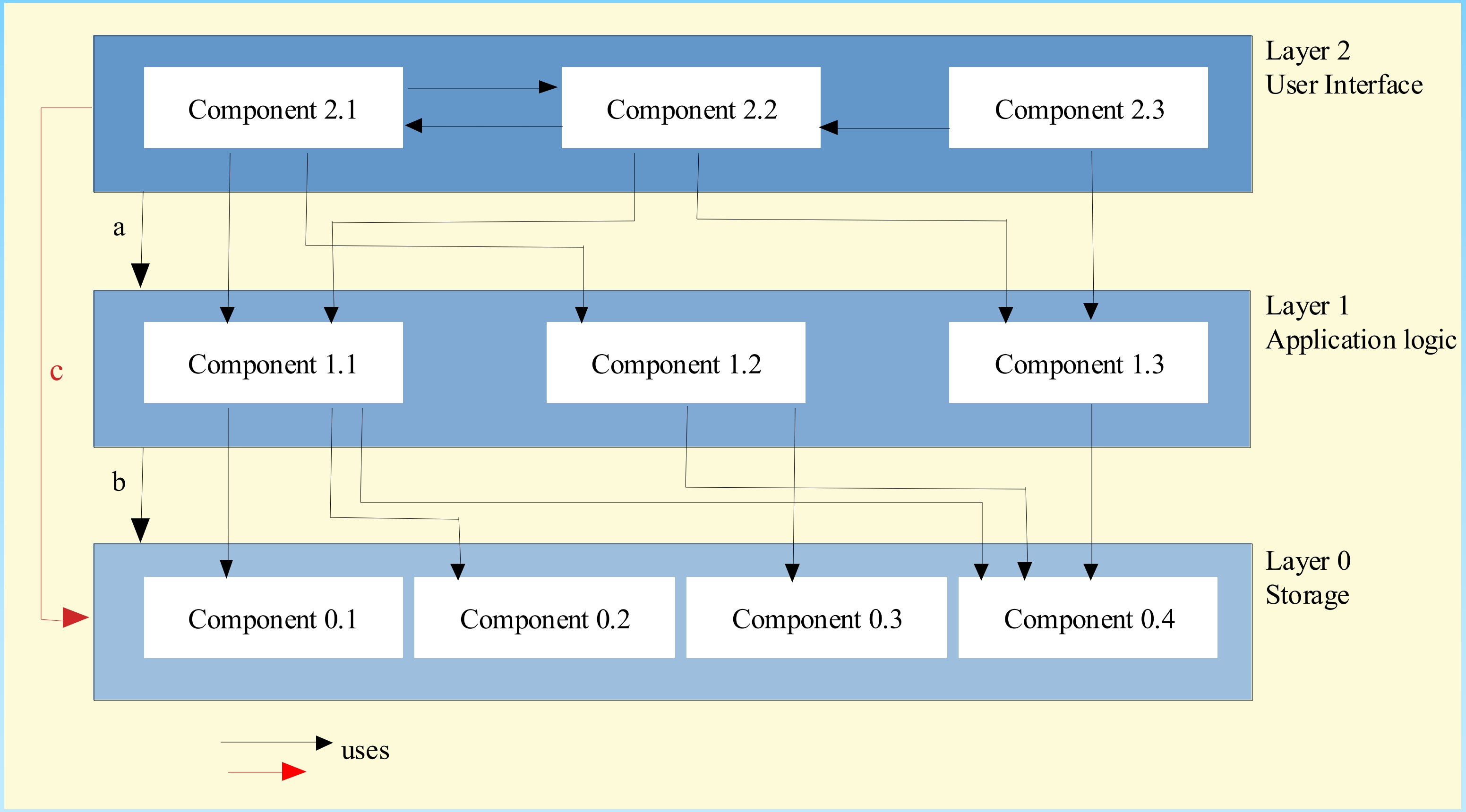
Layers

Intent

Compose a system as a hierarchically ordered set of layers. A layer is a set of software components with a well-defined interface, which uses lower layers as a client and acts as server for upper layers.

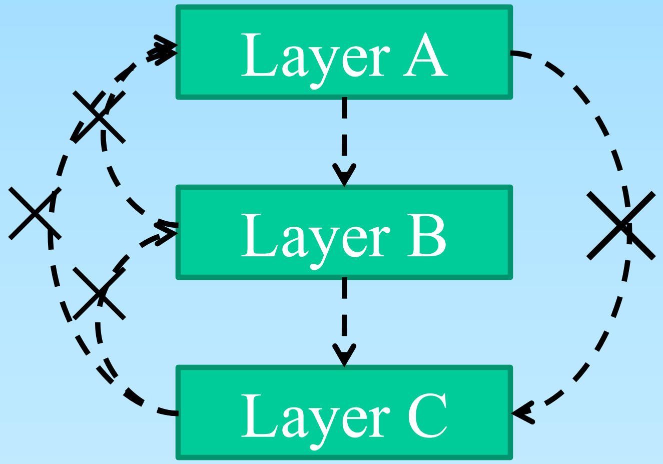
Layers

- Example of a three-layer architecture



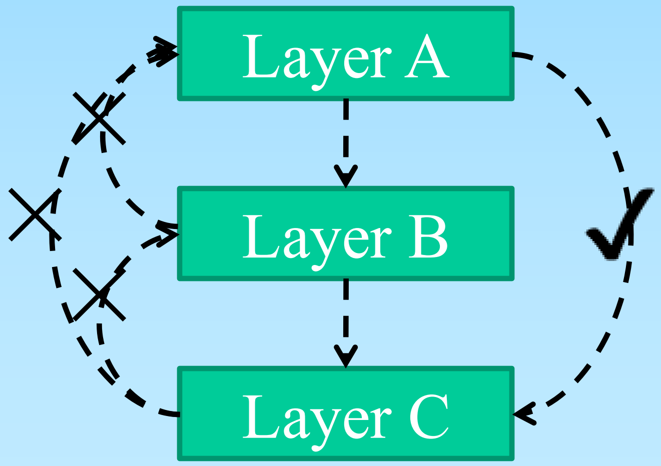
Opaque Layers

- In a layered architecture with opaque layers, each layer can only use the layer directly below it.



Transparent Layers

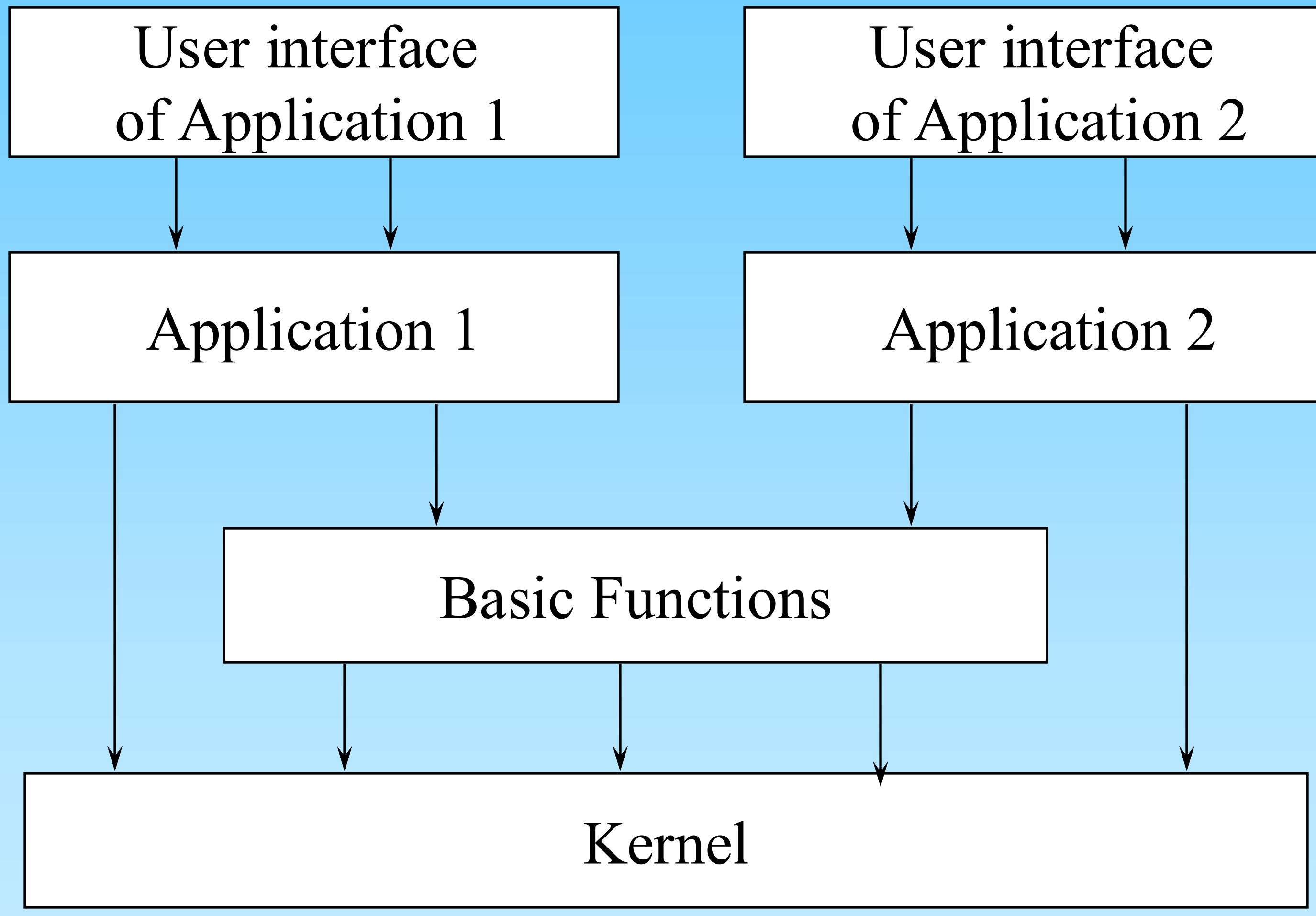
- In a layered architecture with transparent layers, each layer can use **all the layers below** it.



3-Tier Architecture

- A 3-tier architecture is a 3-layer architecture with some of the layers being on different machines.
  - For example, the user interface layer runs on the client machine, the application and database layers on the server.

Structure of Layers


User interface of Application 1
User interface of Application 2
Application 1
Application 2
Basic Functions
Kernel

Layers

Examples

- microkernels (operating systems)
- protocol stacks (networking)
- business applications (3 layers: data base, application kernel, user interface

In some systems usage of the next lower layer only is allowed. Another variation is that every layer exports only a subset of components of the next lower layer besides its own components.

Layers

Applicability

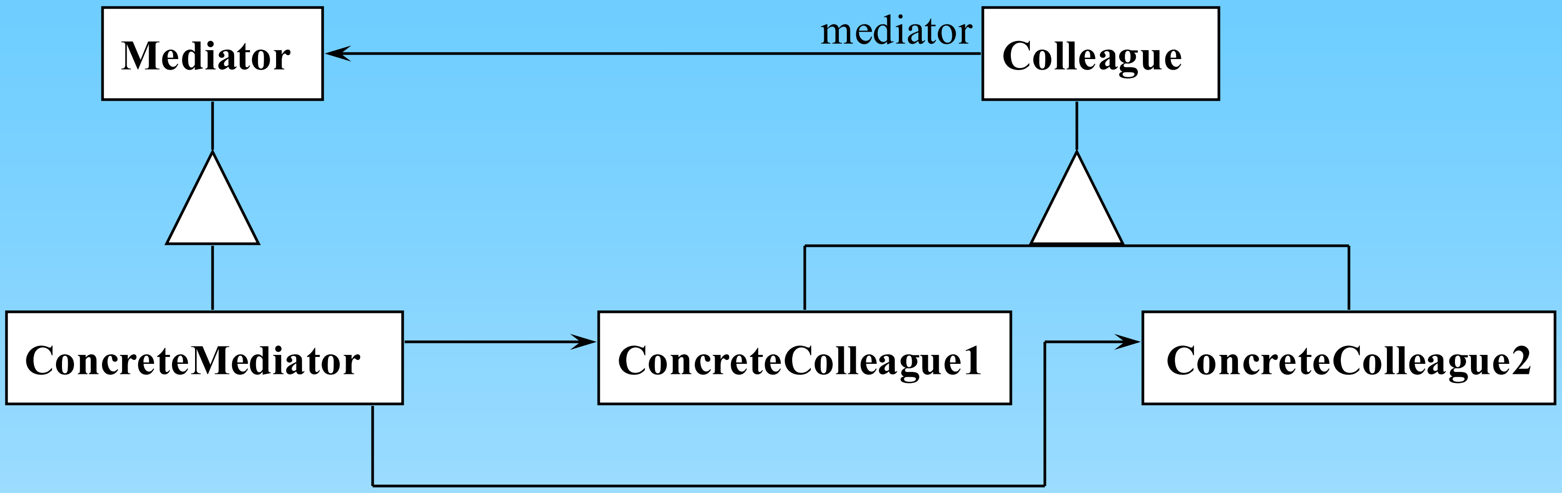
- Clear structuring of a system in levels of abstraction or virtual machines,
- No restrictions on the structure within layers,
- Independent development, testing, and replacement of layers,
- Step-by-step development and step-by-step testing,
- Reuse of lower layers in other configurations,
- To provide a simplified interface for a complex layer (or several), one can use a Façade (see later).

Mediator

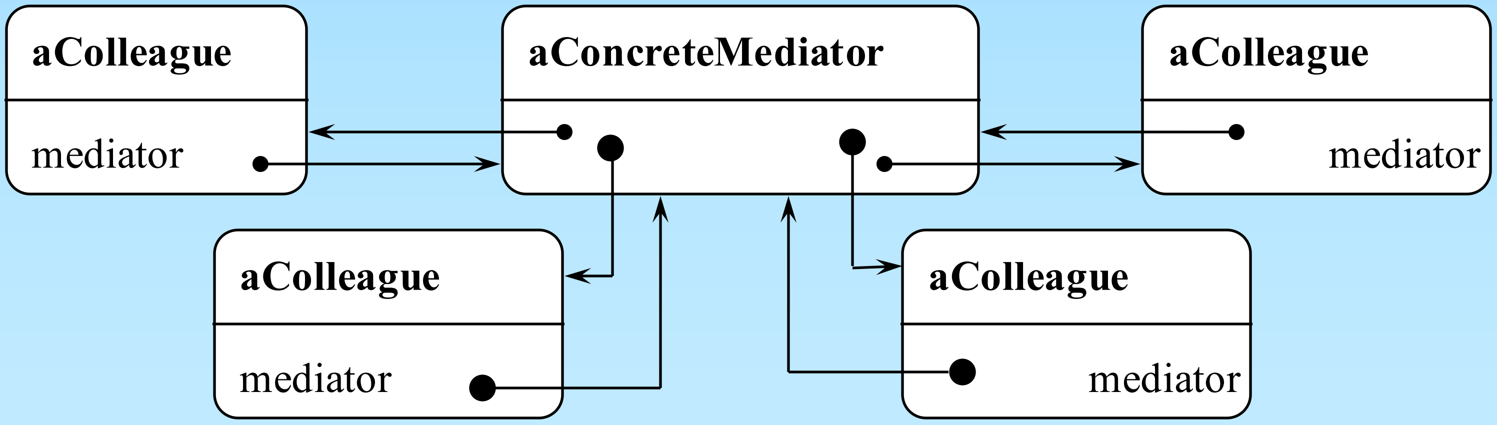
Intent

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Structure of Mediator



typical object structure:



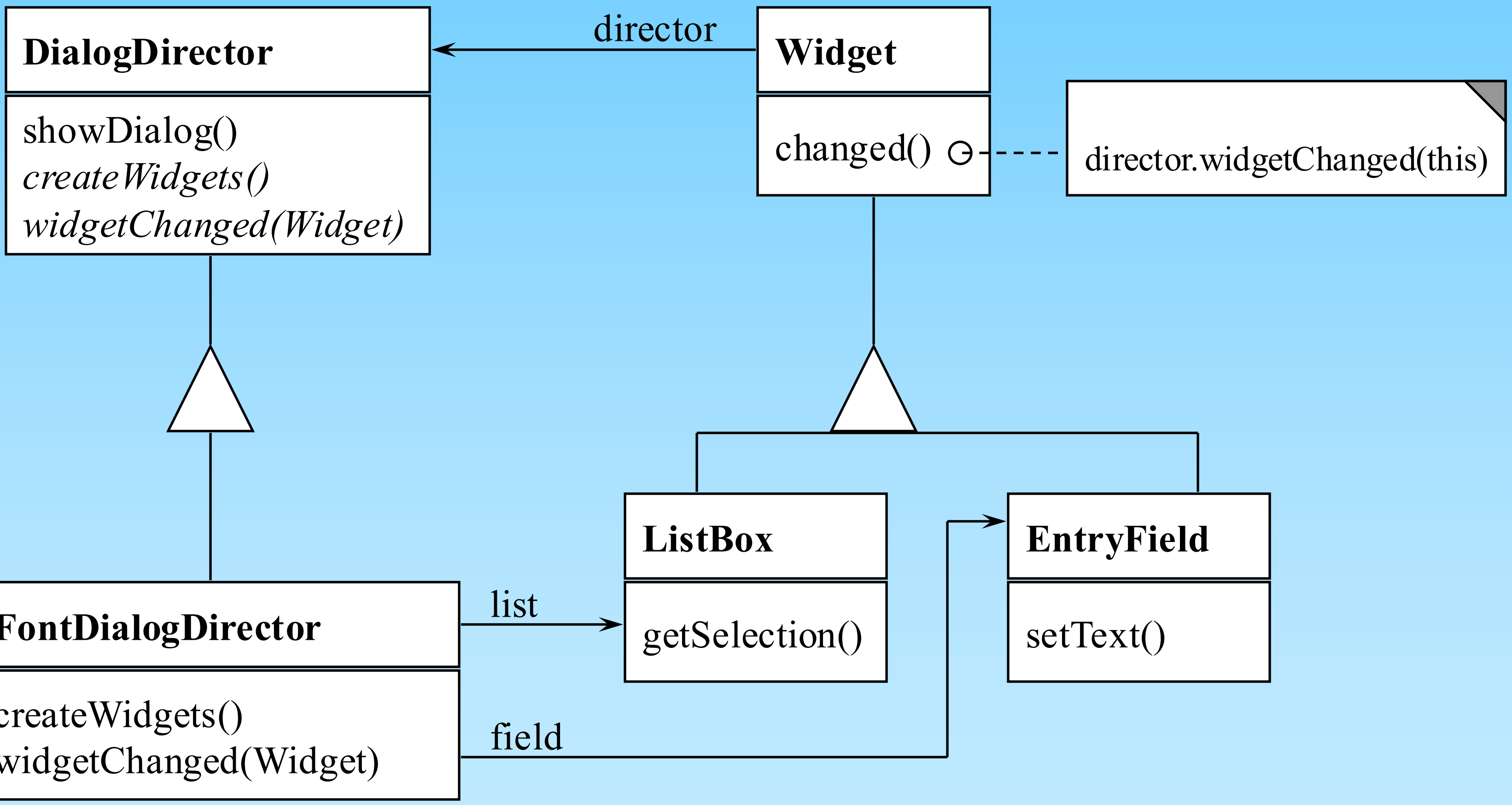
Example of Mediator

- Often there are dependencies between the widgets (buttons, menus, entry fields, etc.) in a dialog box. For example, a button gets disabled when a certain entry field is empty.
- Different dialog boxes will have different dependencies between widgets.
- Customizing them individually in subclasses will be tedious, since many classes are involved.

⇨ encapsulate collective behavior in a separate mediator object

Example of Mediator

Window with Font Dialog



Mediator

Applicability

- When a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- When reusing an object is difficult because it refers to and communicates with many other objects.
- When a behavior that´s distributed between several classes should be customizable without a lot of subclassing.

Bridge

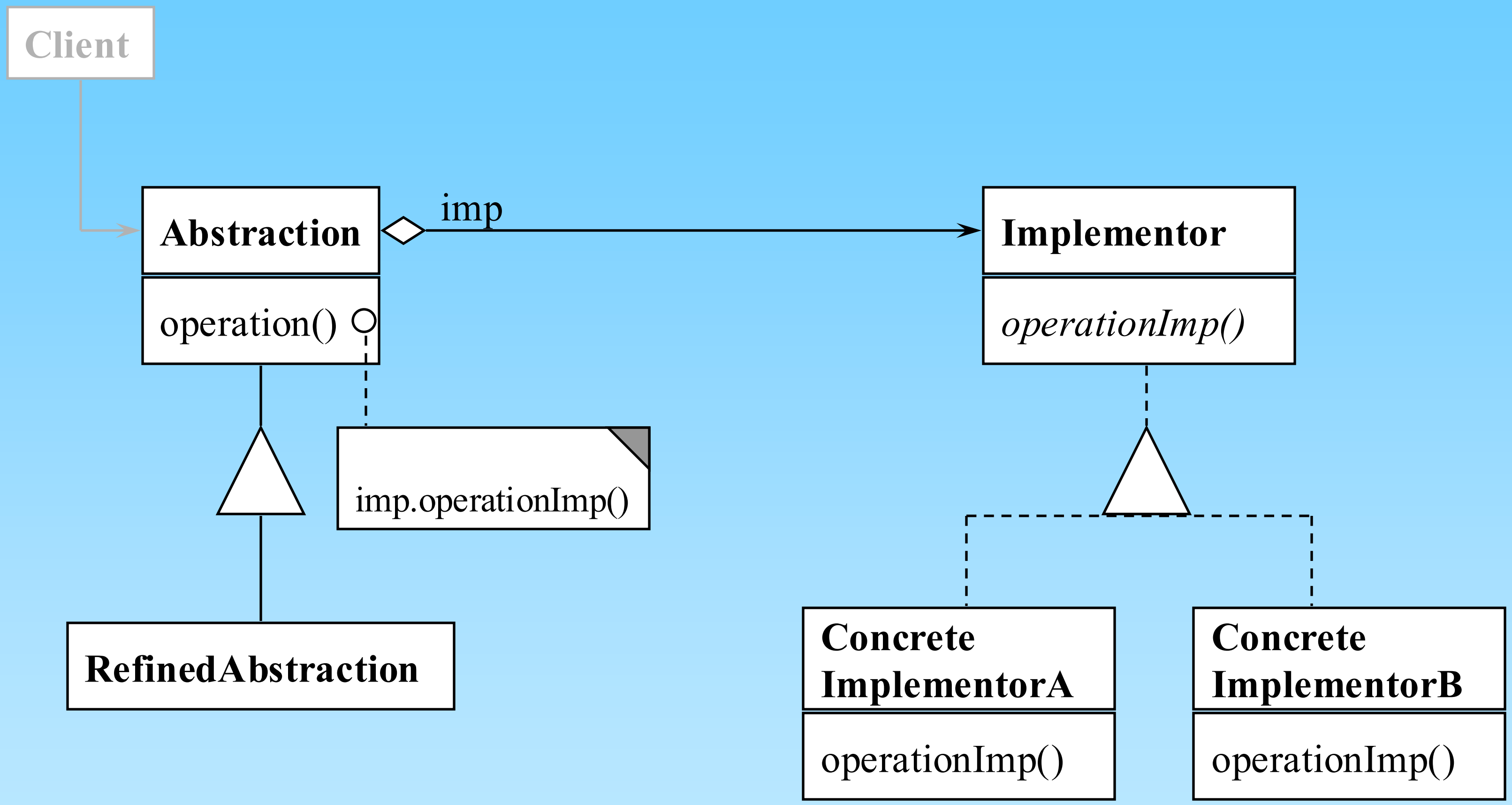
Intent

Decouple an abstraction from its implementation so that the two can vary independently.

Also known as

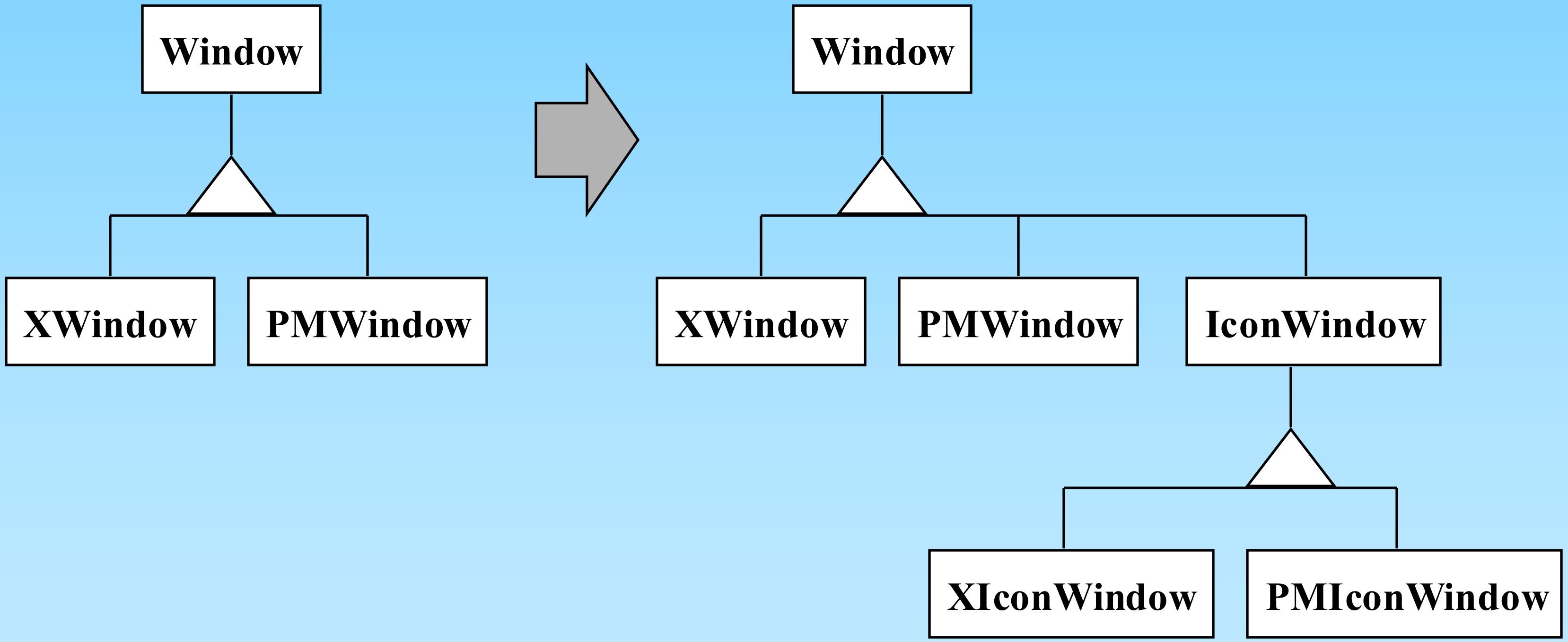
Handle/Body

Structure of Bridge

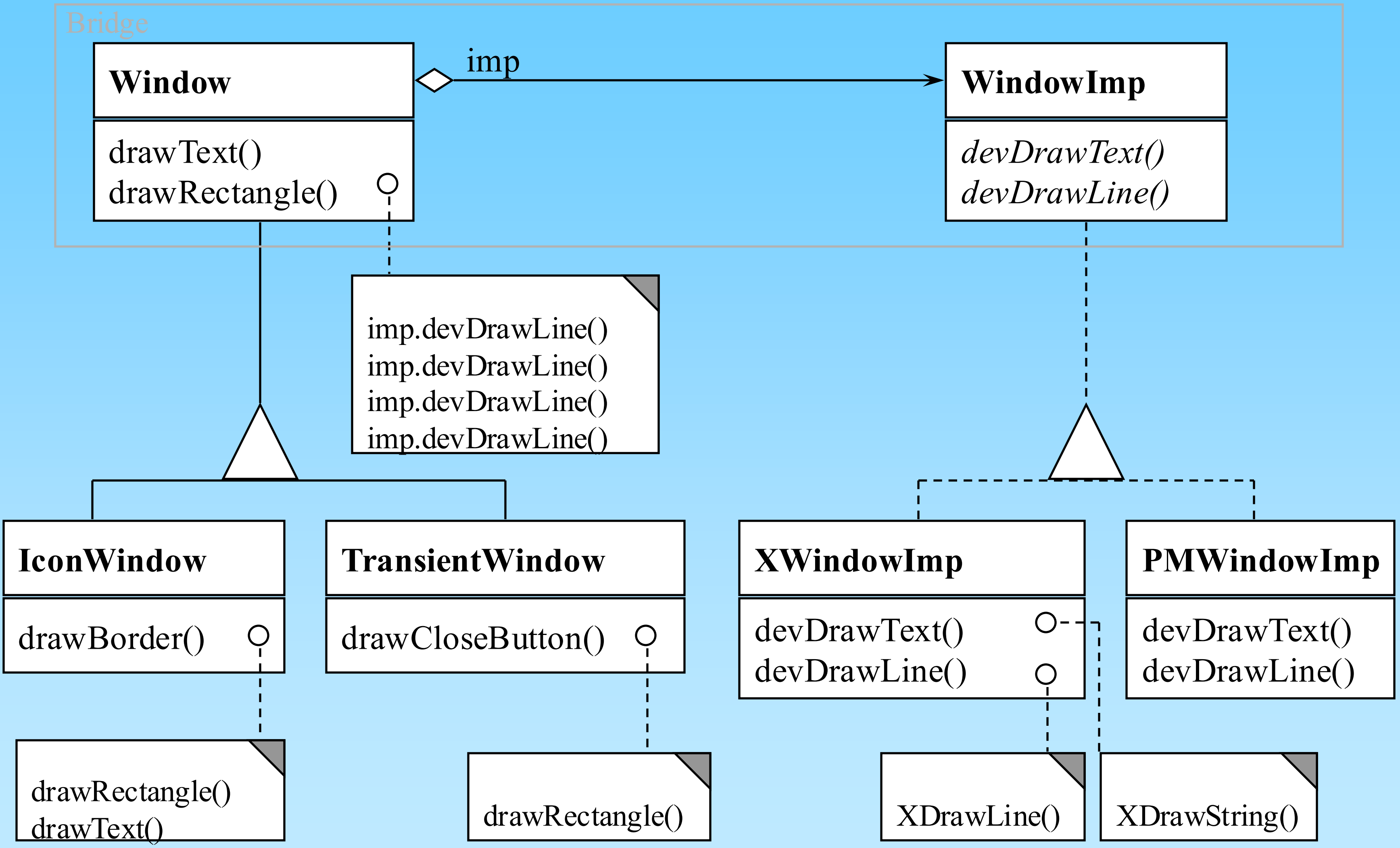


Example of Bridge

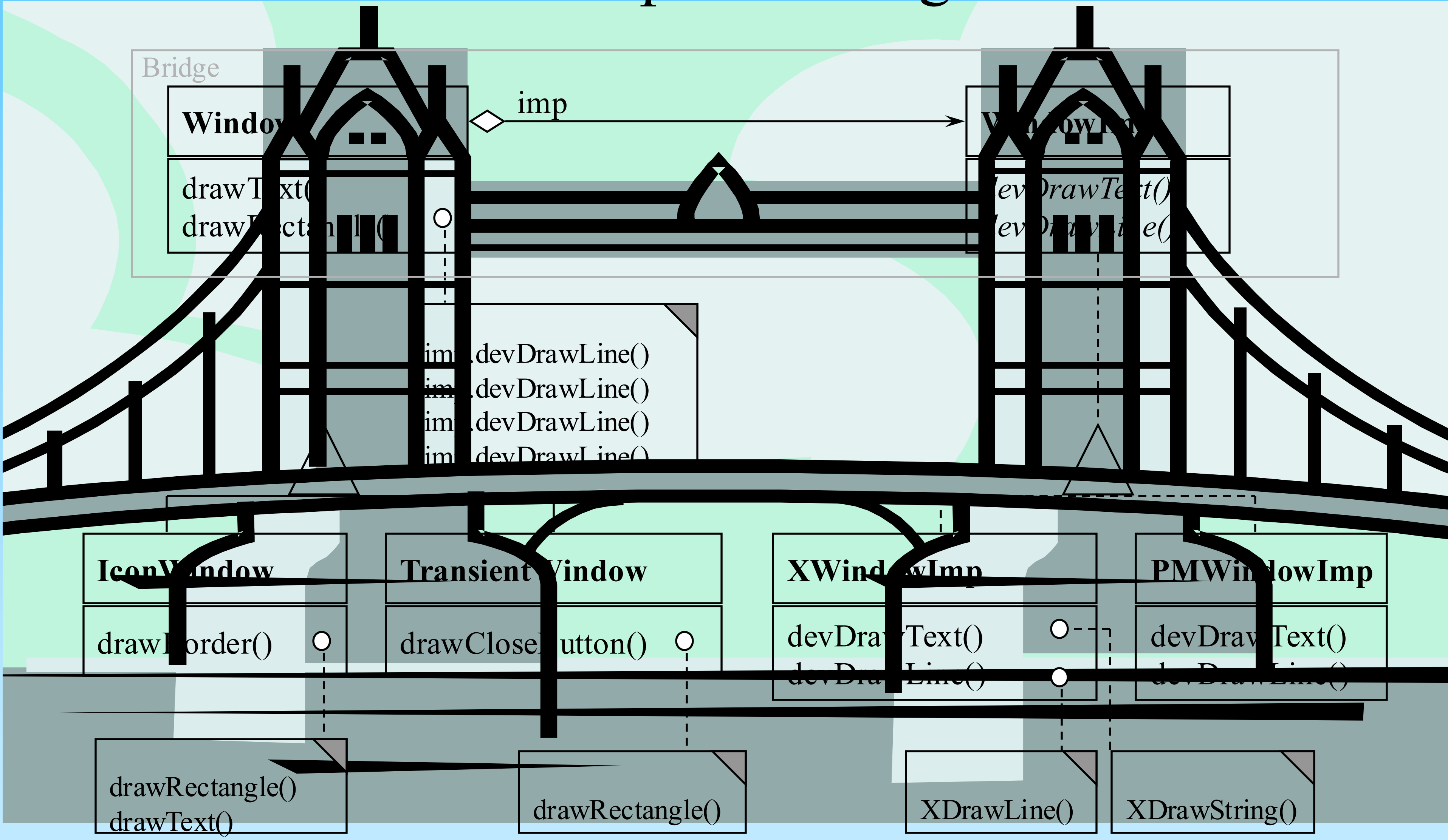
Both implementation and abstraction of user interfaces: subclassing leads to many classes.



Example of Bridge



Example of Bridge



Bridge

Applicability

- When a permanent binding between abstraction and its implementation should be avoided.
- When both abstraction and implementation should be extensible by subclassing.
- When changes in the implementation should have no impact on clients.
- When the implementation should be hidden completely from clients (C++: representation visible in class interface)
- When a large number of classes should be avoided (see example).
- When an implementation should be shared among multiple objects.
- (two-levels with subclassing on both levels)

Adapter

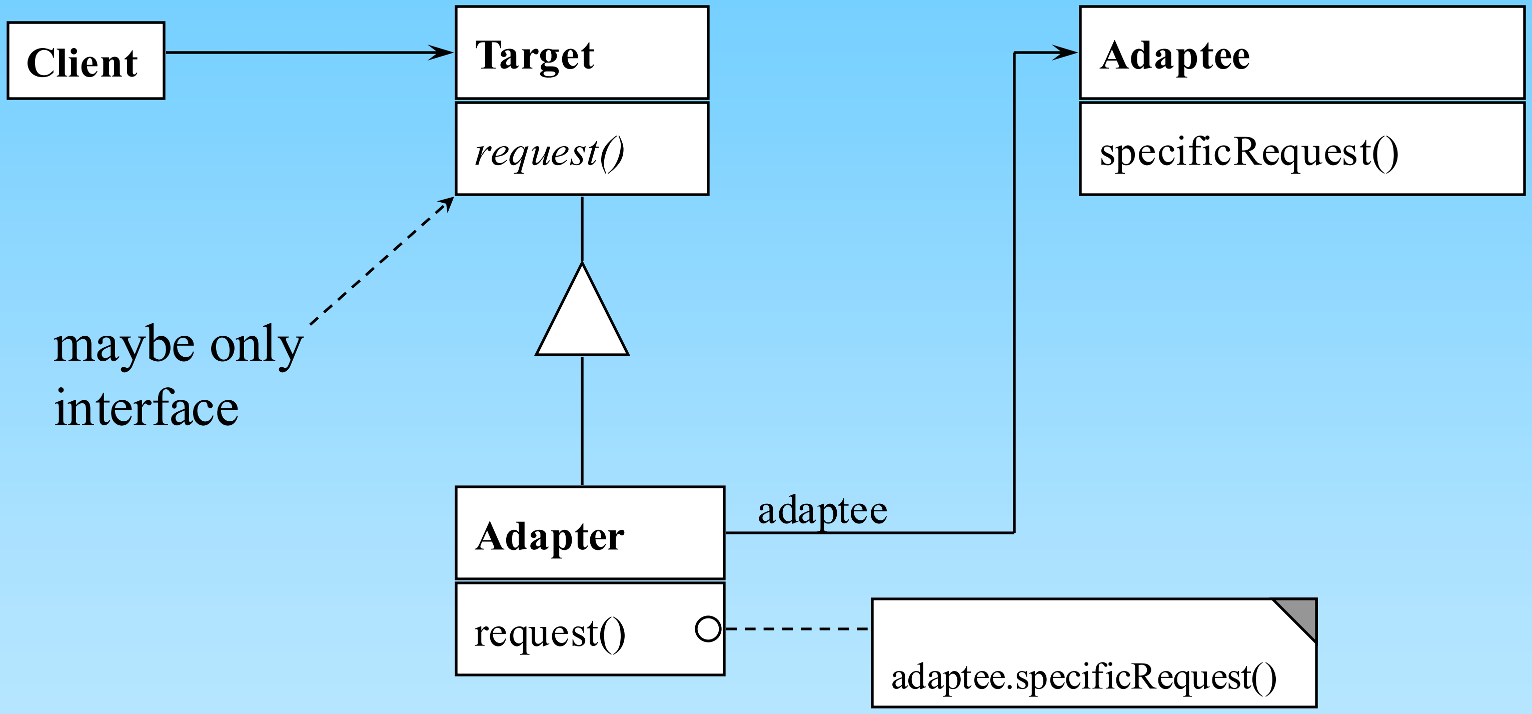
Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Also known as

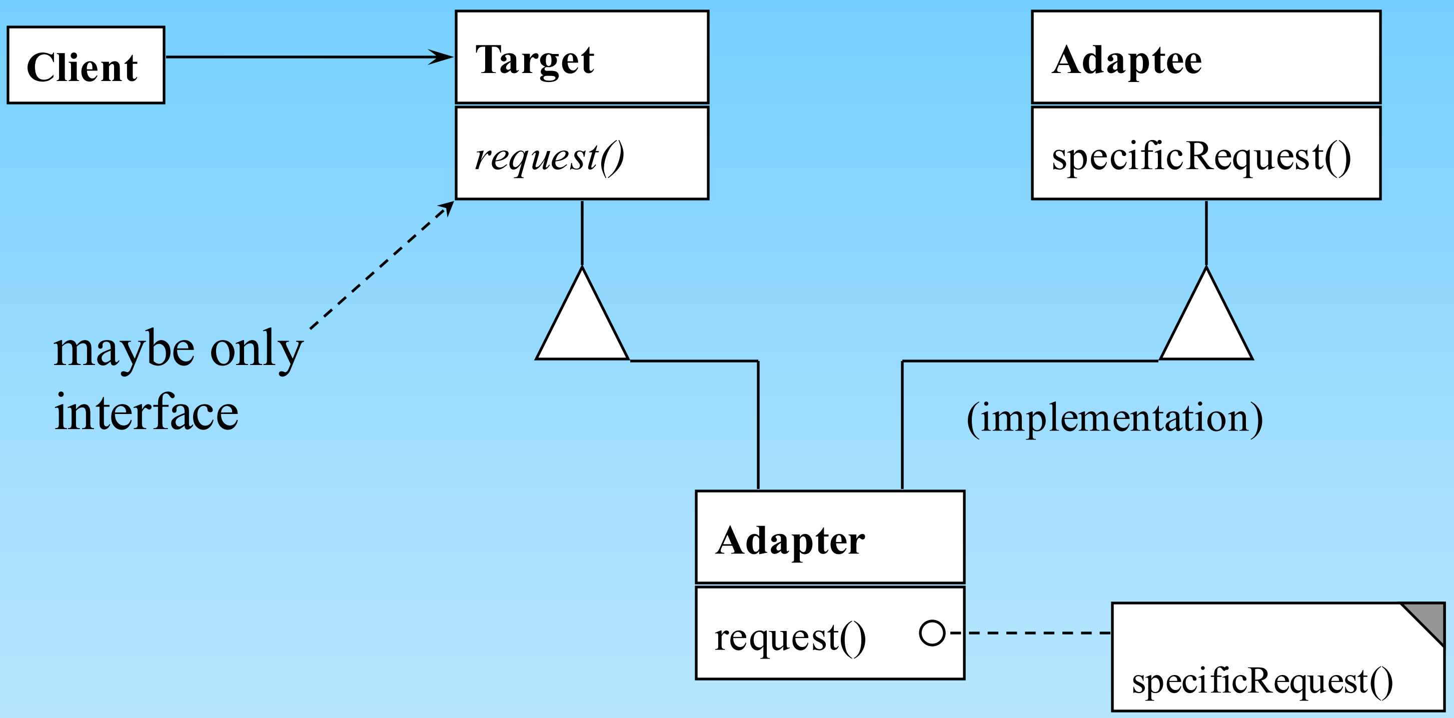
Wrapper

Structure of Adapter (1)



without multiple inheritance (object adapter)

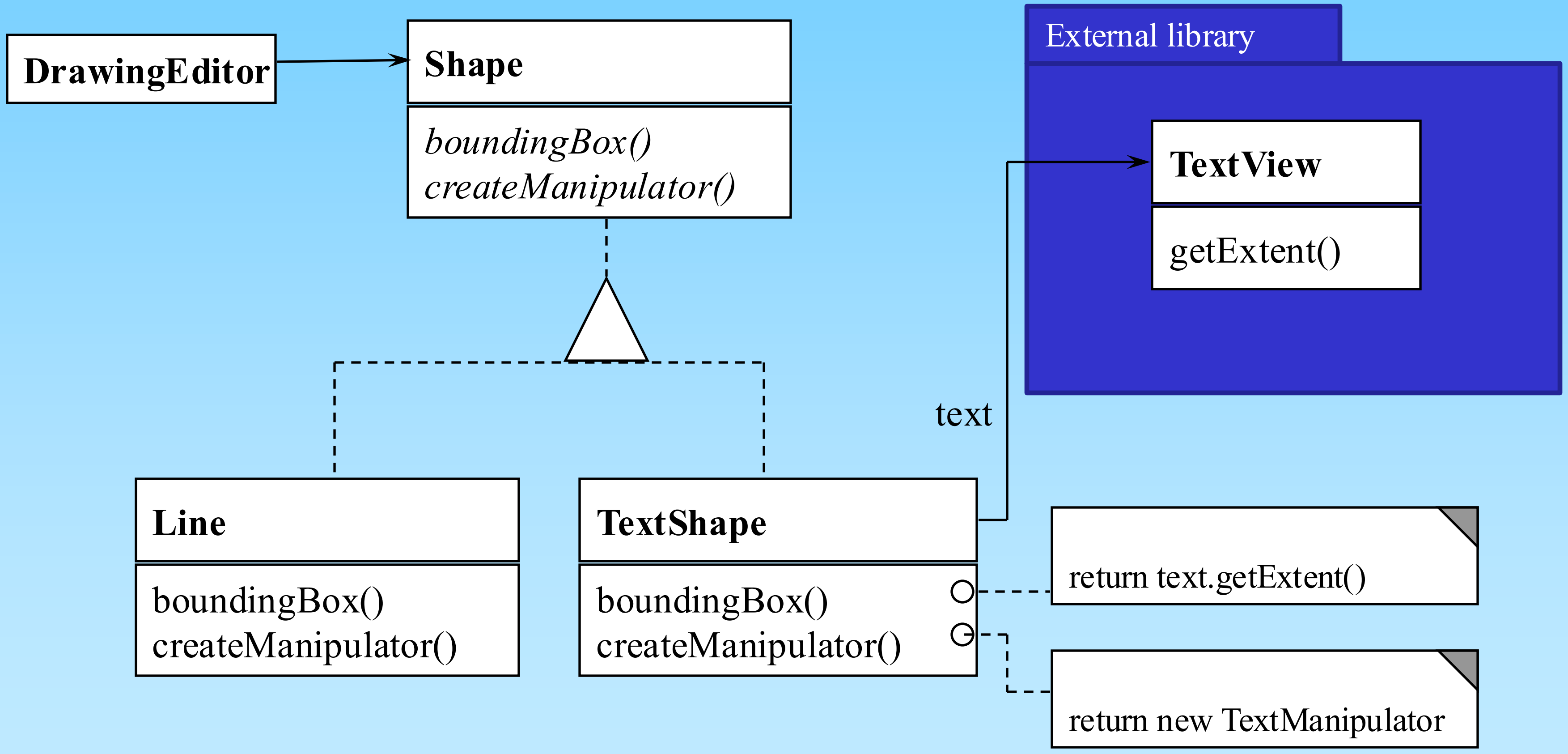
Structure of Adapter (2)



with multiple inheritance (class adapter)

Example of Adapter

Use of external class library
to display texts in a drawing editor.



Adapter

Applicability

- When an existing class should be used, and its interface does not match the one you need.
- When a reusable class should be created that cooperates with unrelated or unforeseen classes, that is, classes that don´t necessarily have compatible interfaces.
- (object adapter only) When several existing subclasses should be used, but it’s impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Proxy

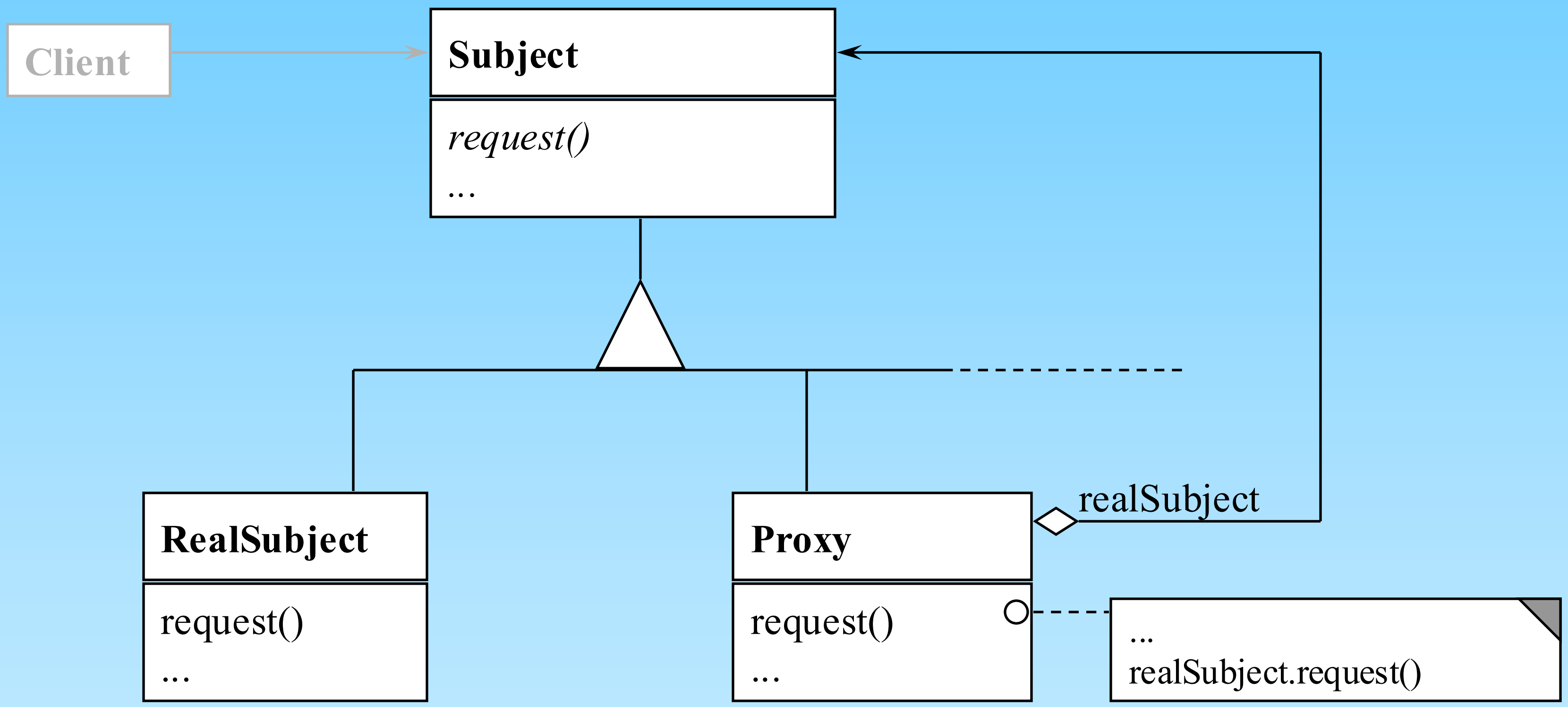
Intent

Provide a surrogate or placeholder for another object to control access to it.

Also known as

Surrogate

Structure of Proxy



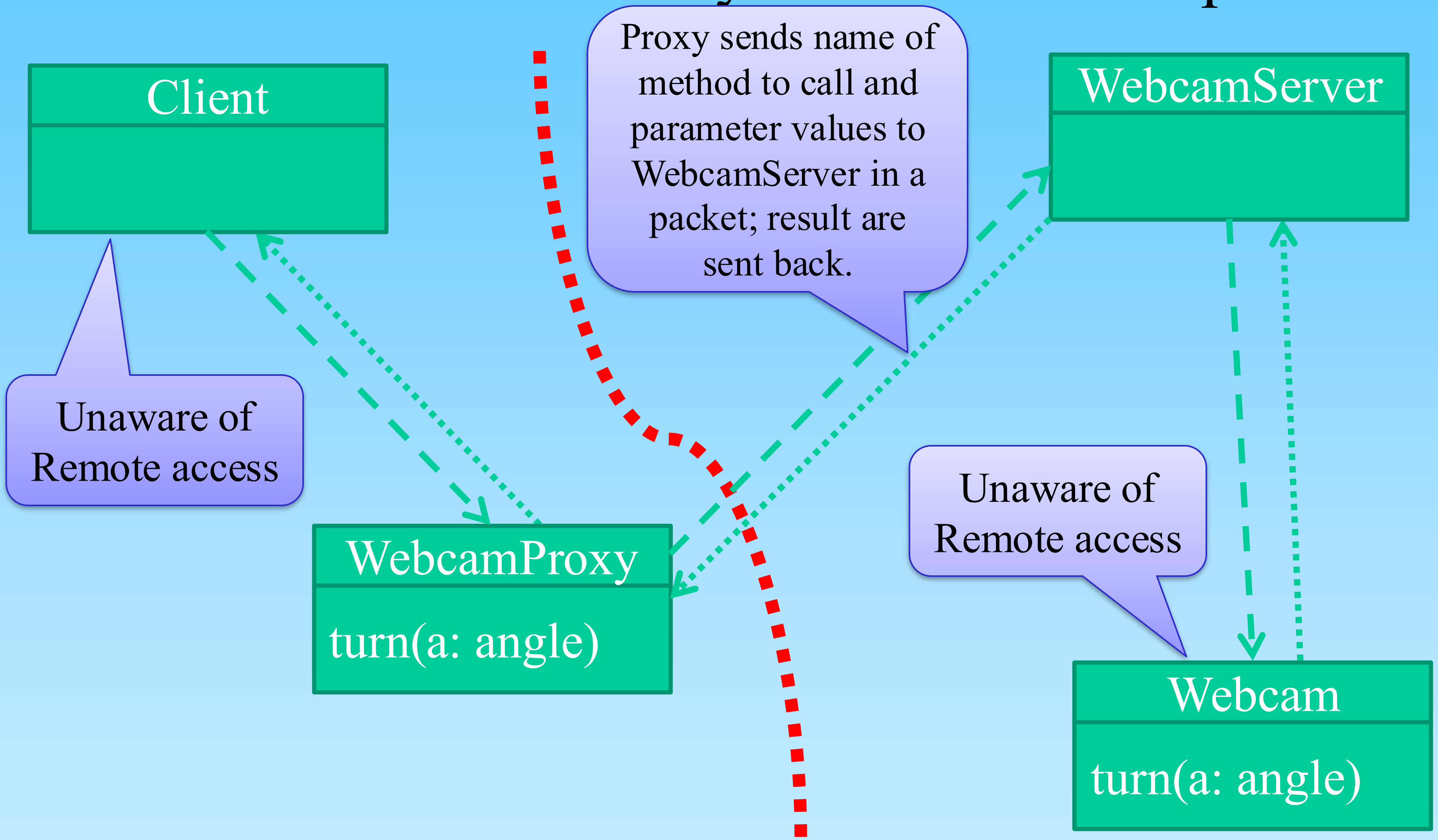
Proxy

Applicability

Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the proxy pattern is applicable:

1. A **logging proxy** (counting proxy) counts the number of references to the real object so that it can be freed automatically when there are no more references. It also can log access information.
2. A **remote access proxy** provides a local representative for an object in a different address space.

Remote Access Proxy: Webcam-Example

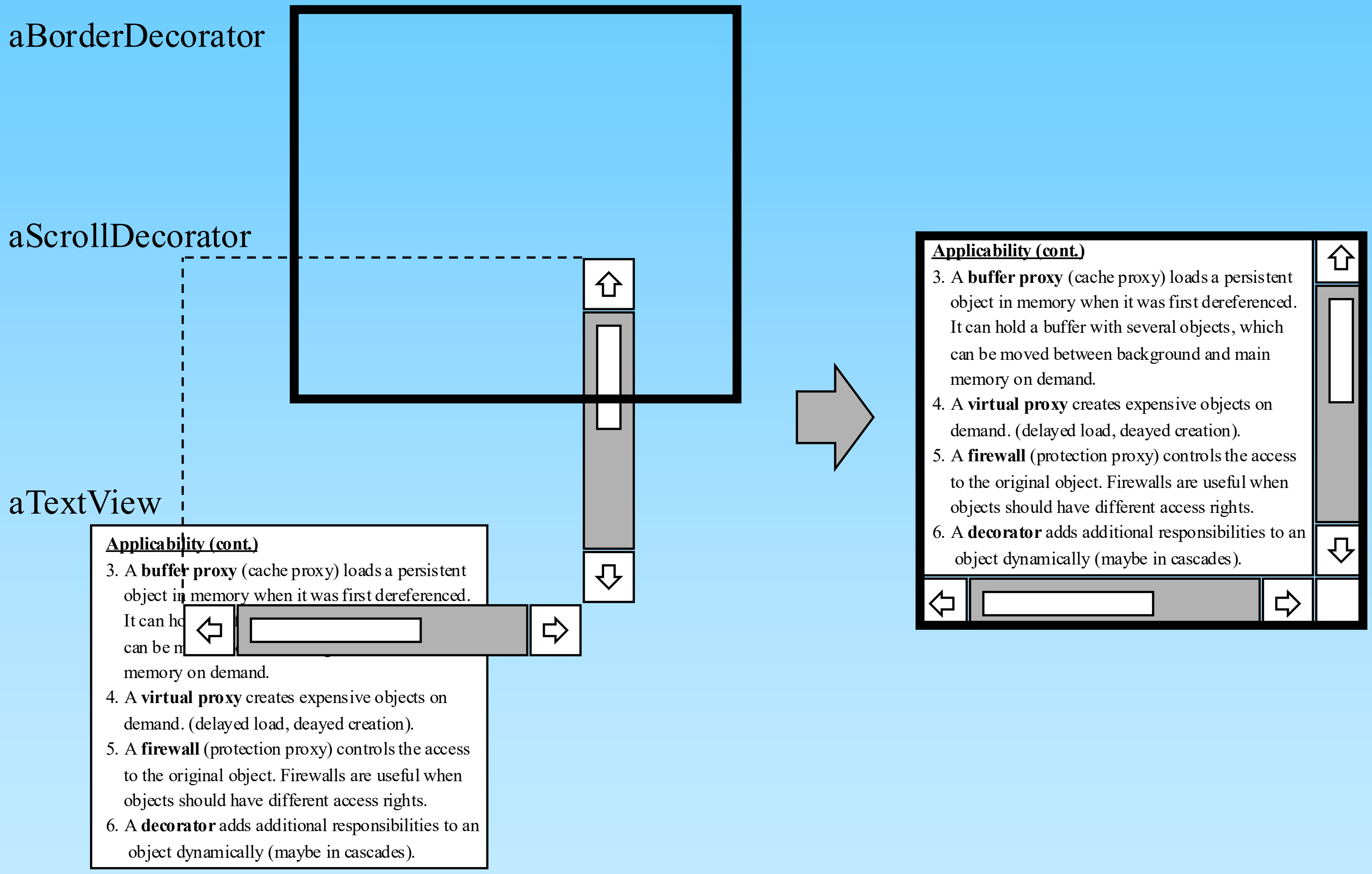


Proxy

Applicability (cont.)

3. A **buffer proxy** (cache proxy) loads a persistent object in memory when it was first dereferenced. It can hold a buffer with several objects, which can be moved between background and main memory on demand.
4. A **virtual proxy** creates expensive objects on demand. (delayed load, delayed creation).
5. A **firewall** (protection proxy) controls the access to the original object. Firewalls are useful when objects should have different access rights.
6. A **decorator** adds additional responsibilities to an object dynamically (maybe in cascades).

Example of Proxy: Decorator



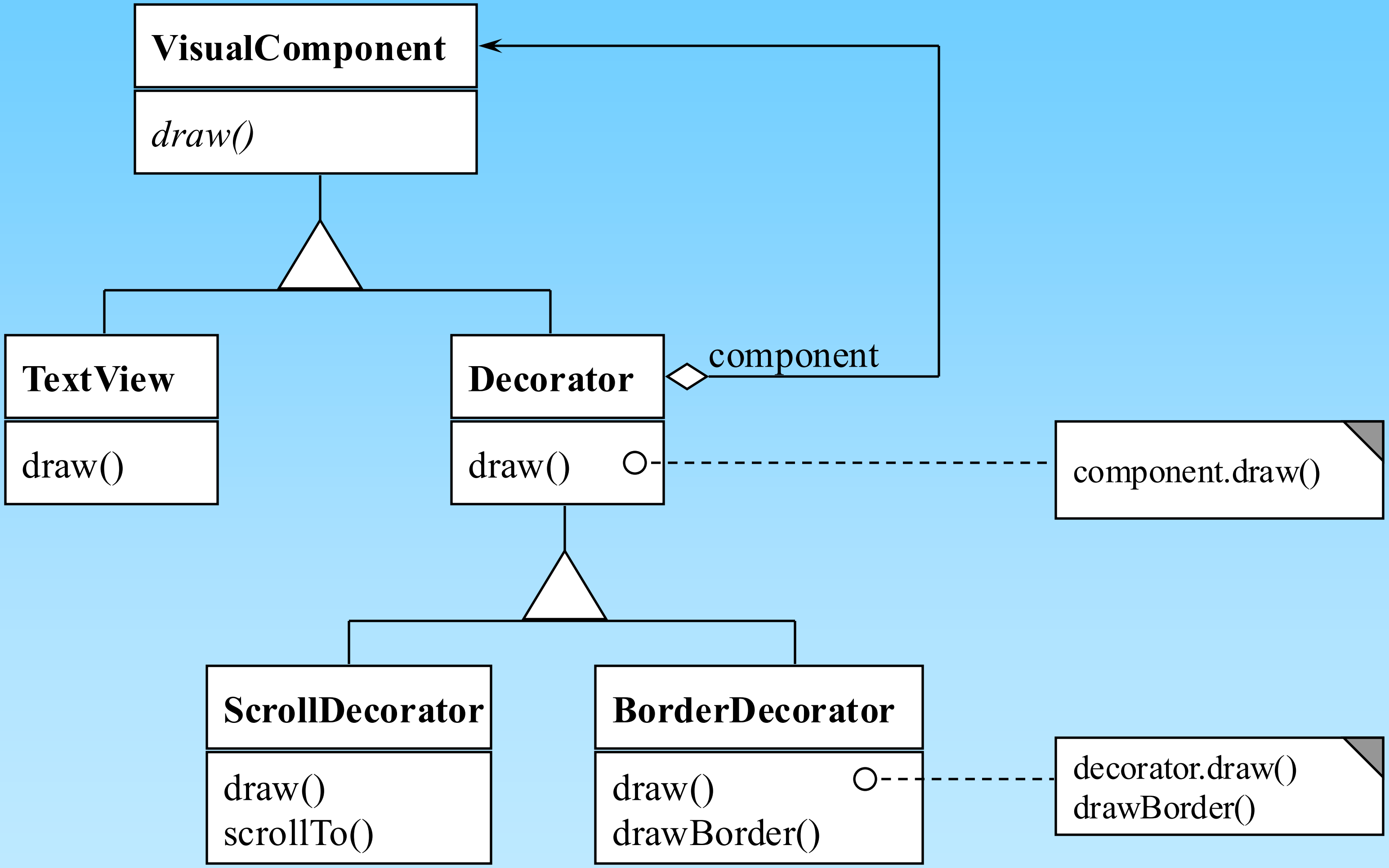
Applicability (cont.)

3. A **buffer proxy** (cache proxy) loads a persistent object in memory when it was first dereferenced. It can hold a buffer with several objects, which can be moved between background and main memory on demand.
4. A **virtual proxy** creates expensive objects on demand. (delayed load, deayed creation).
5. A **firewall** (protection proxy) controls the access to the original object. Firewalls are useful when objects should have different access rights.
6. A **decorator** adds additional responsibilities to an object dynamically (maybe in cascades).

Applicability (cont.)

3. A **buffer proxy** (cache proxy) loads a persistent object in memory when it was first dereferenced. It can ho can be m memory on demand.
4. A **virtual proxy** creates expensive objects on demand. (delayed load, deayed creation).
5. A **firewall** (protection proxy) controls the access to the original object. Firewalls are useful when objects should have different access rights.
6. A **decorator** adds additional responsibilities to an object dynamically (maybe in cascades).

Example of Proxy: Decorator



Pipeline

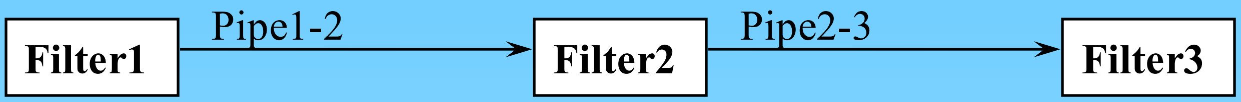
Intent

Provide a structure for systems that operate on date flows. Every operation step is encapsulated in a *filter* component. Data is transmitted via *pipes*. To build new systems create a new combination of filters.

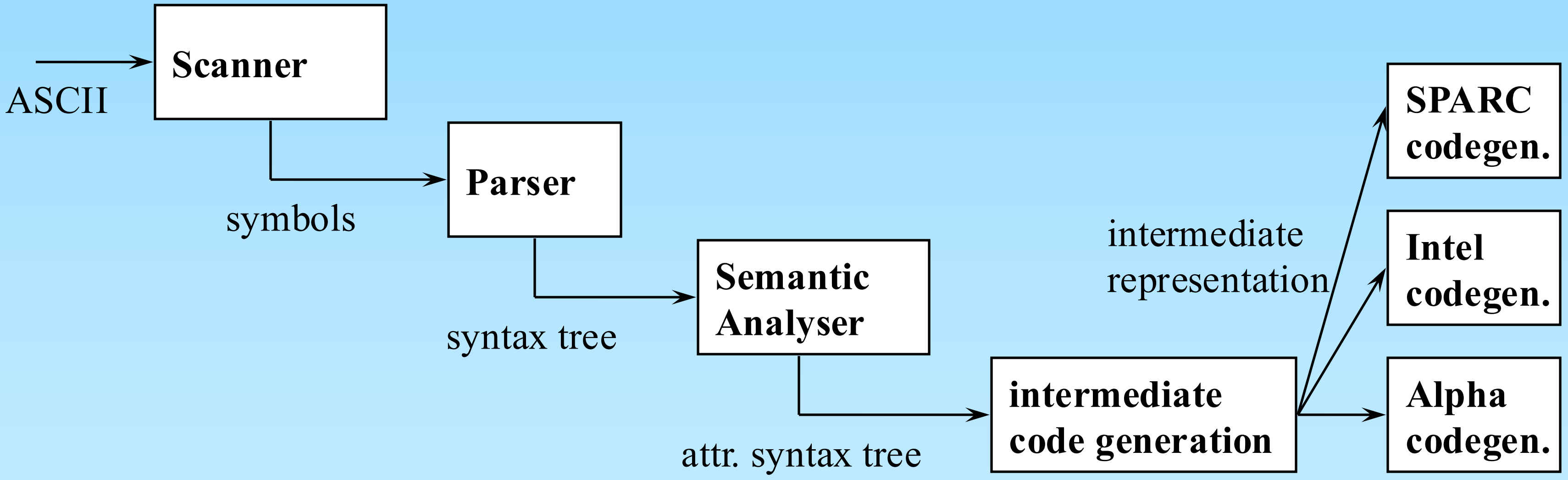
Also known as

Data Flow, Pipes and Filters

Pipeline



Example: Compiler



Pipeline

Applicability

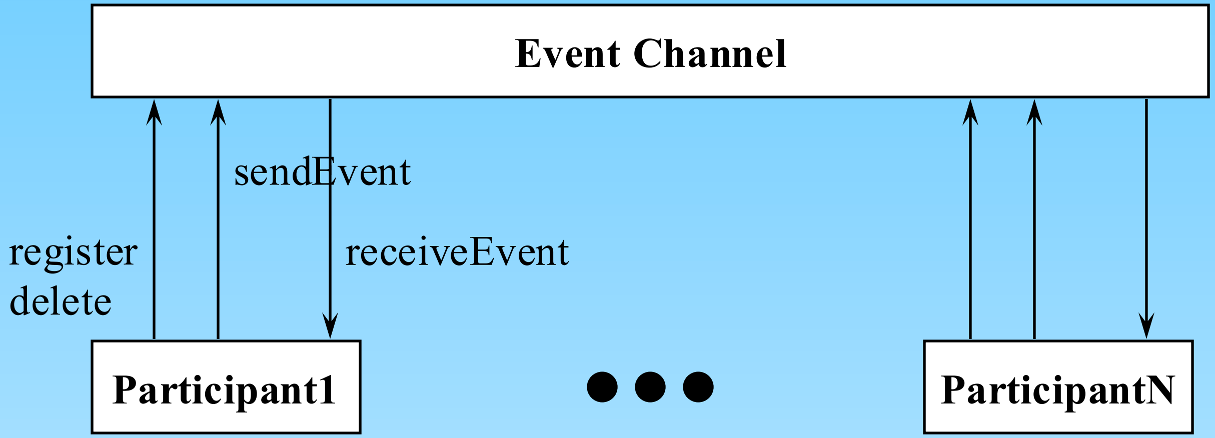
- When a system operates on, or transforms, data flows and a system of a single component is not customizable.
- When in future development some components will be replaced or the operational order will be changed.
- When smaller components can be reused in other systems.
- When some components should run in parallel or quasi-parallel.

Event Channel

Intent

Decouple participants of a system completely so that they can work independently, and do not know anything about the existence or number of other participants. Interactions are transmitted as events.

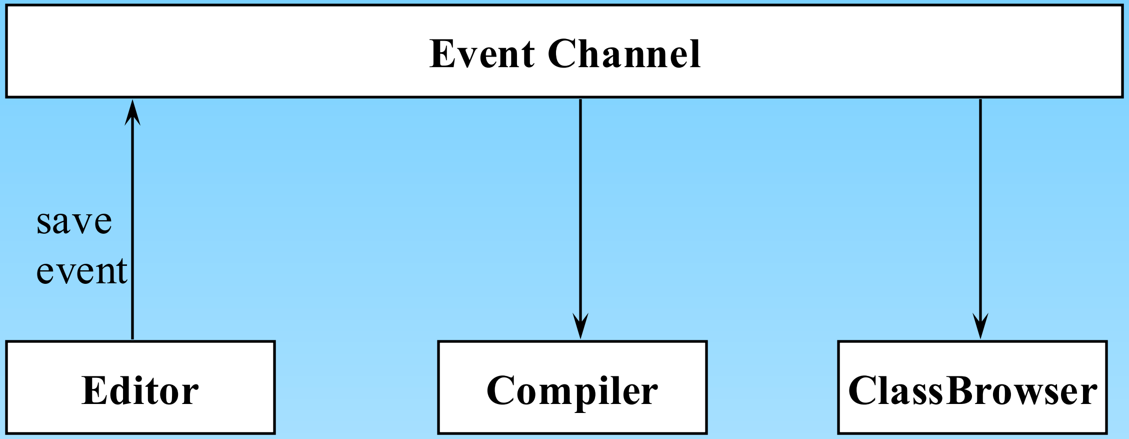
Structure of Event Channel



The participants register at the event channel about which events they want to be informed. When a participant sends an event to the event channel, it resends this event to registered participants.

Example of Event Channel

Programming Environment



When the editor saves a file, it generates an event which causes the compiler to start and the class browser to update its display of classes.

Framework

Intent

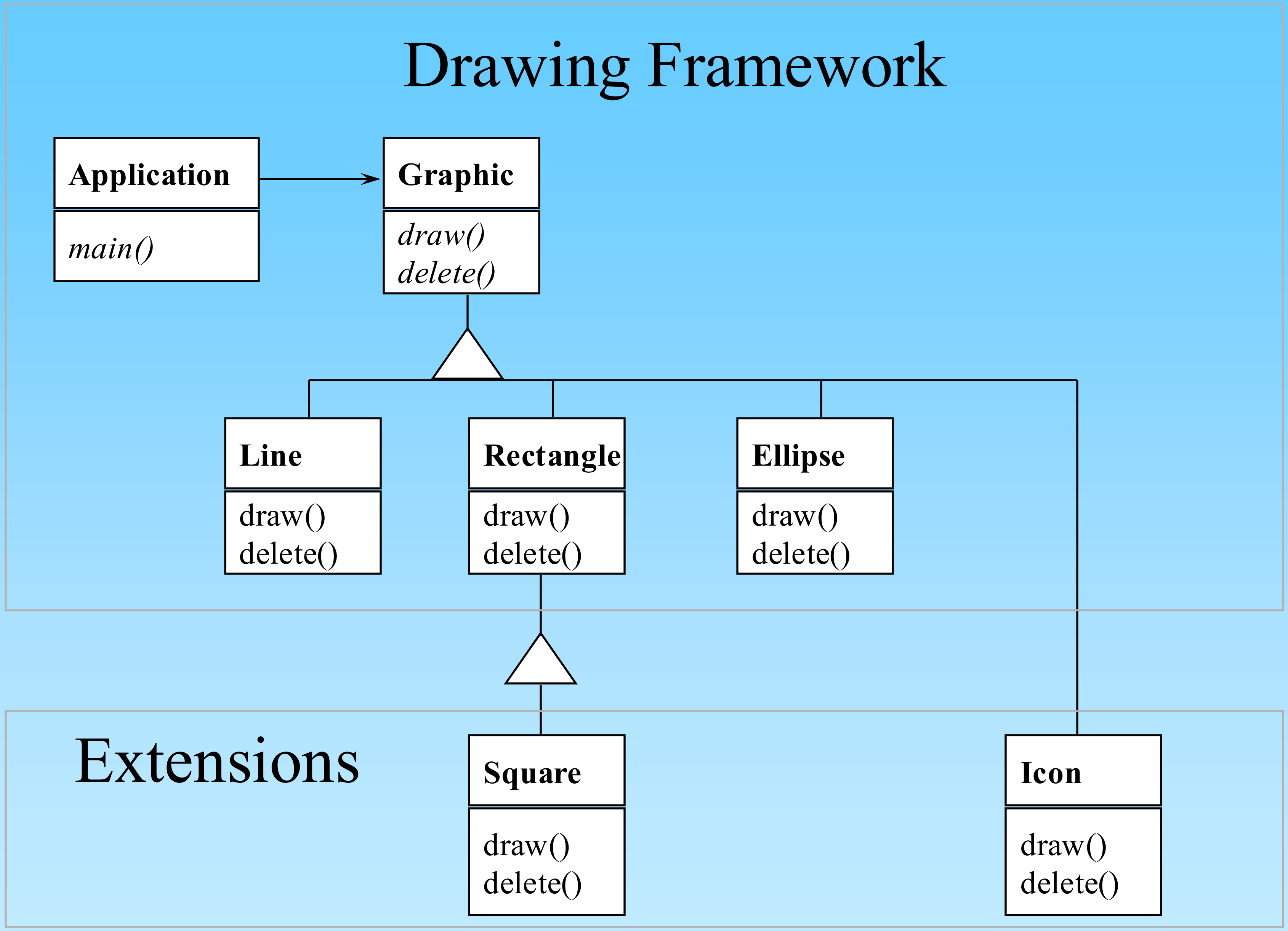
Provide a (nearly) complete program that contains planed "empty spaces", which can be extended to form a complete program. It contains the complete application's logic, mostly the main program. The user can build subclasses from some classes, overwrite certain methods or implement pre-defined abstract methods. The framework is designed so that it calls the user-defined extensions correctly.

Example of Framework

A drawing framework has a class *Graphic* and some subclasses *Line*, *Rectangle*, *Ellipse*, etc. The user may add new subclasses, e.g. *Square* or *Icon*, but he must define a **`draw`** method in this new subclass.

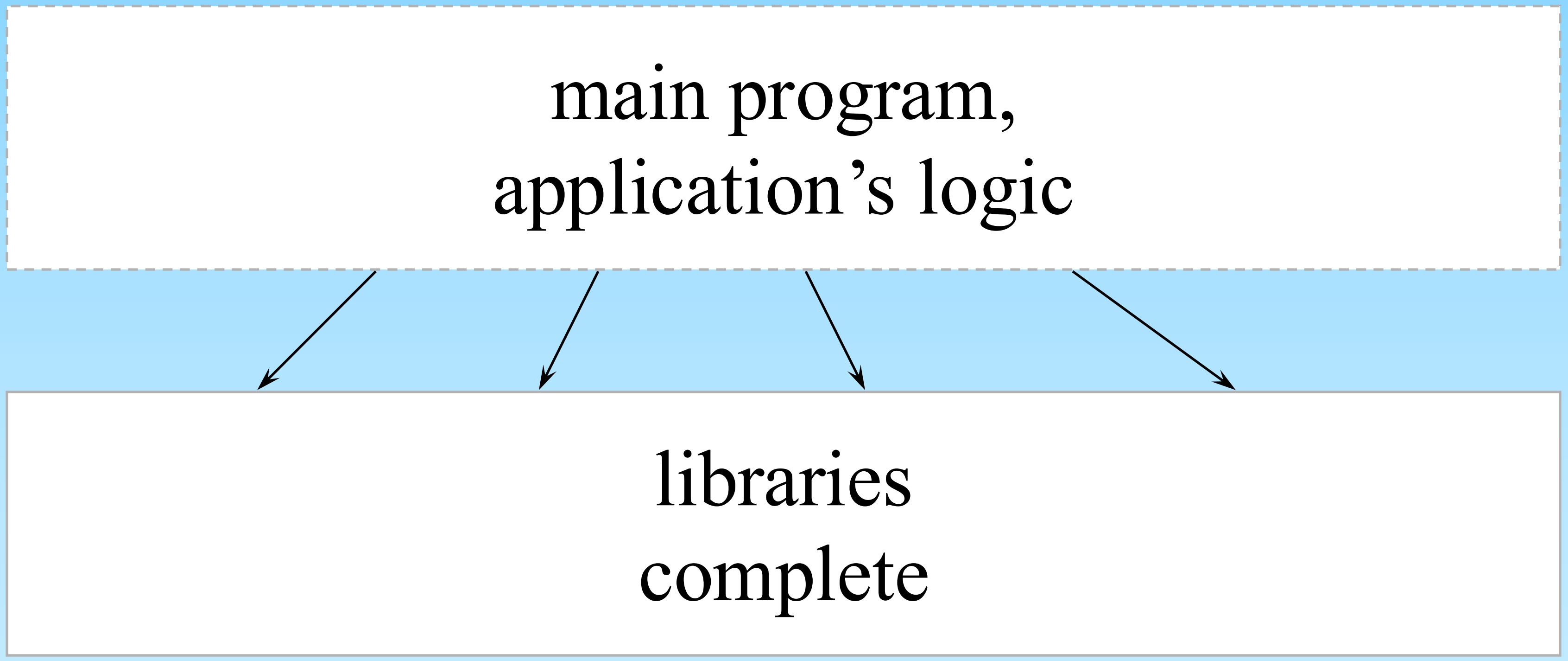
The framework ensures the correct creation, positioning, moving, saving, etc. of the objects of the new classes.

Drawing Framework



Conventional System Structure

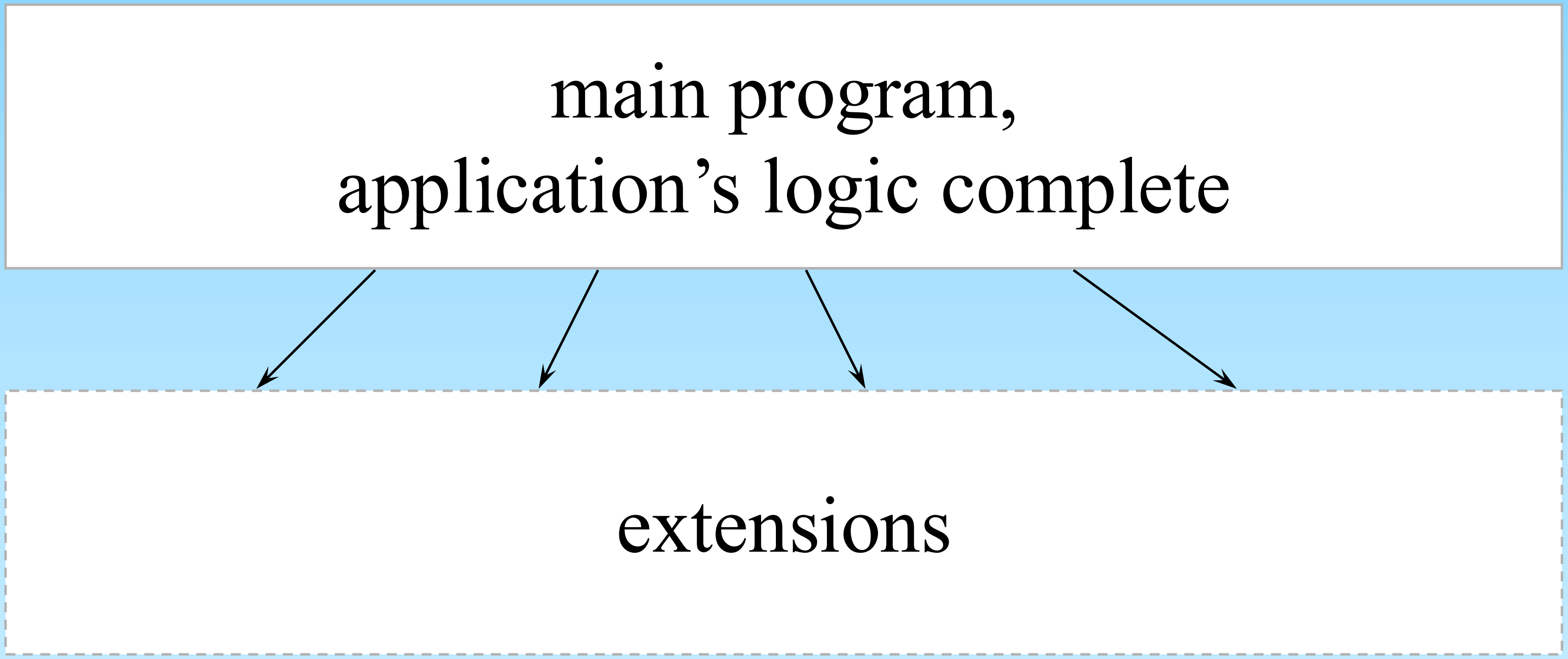
- Vendor provides libraries,
- Programmer writes main program and application´s logic.



Framework

A Framework follows the “Hollywood-Principle”:

“Don’t call us - we’ll call you”. The main program already exists and calls programmer-defined extensions.



Framework

Applicability

- When a base version of the application should work without extensions.
- When extensions should be possible with consistent behavior (application’s logic in framework).
- When a new implementation of the application’s logic should be avoided.

The patterns Factory Method, Abstract Factory, and Template Method are often used in Frameworks.

2. Variant Management

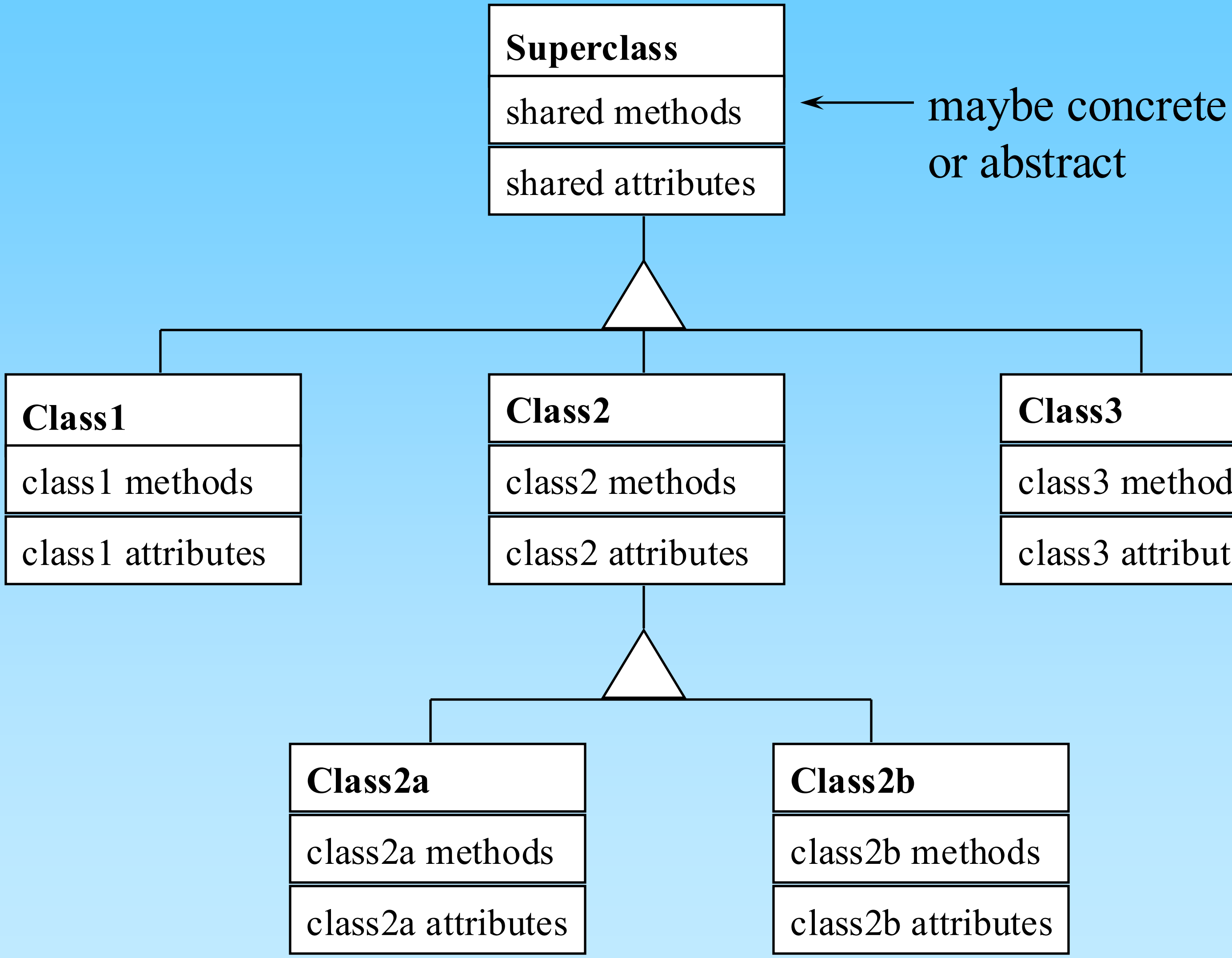
- Superclass (Family)
  - Strategy
  - Composite
  - (Bridge)
- Visitor
  - Default Visitor
  - Extrinsic Visitor
  - Acyclic Visitor
- Template Method
  - Factory Method
  - Builder
- Abstract Factory

Superclass

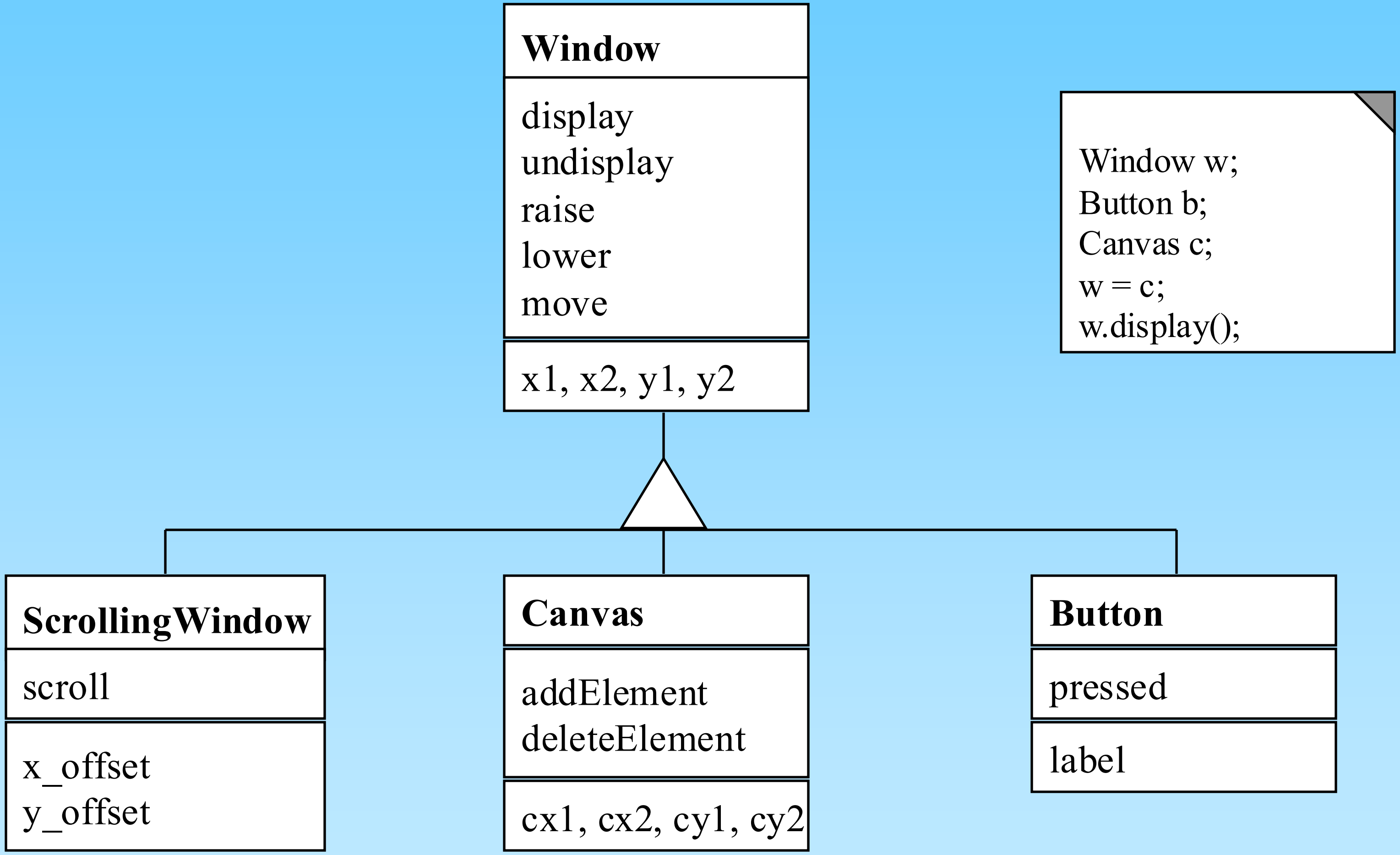
Intent

Uniform treatment of objects that belong to different classes but share common attributes or methods.

Structure of Superclass



Example of Superclass



```
Window w;
Button b;
Canvas c;
w = c;
w.display();
```

Superclass

Applicability

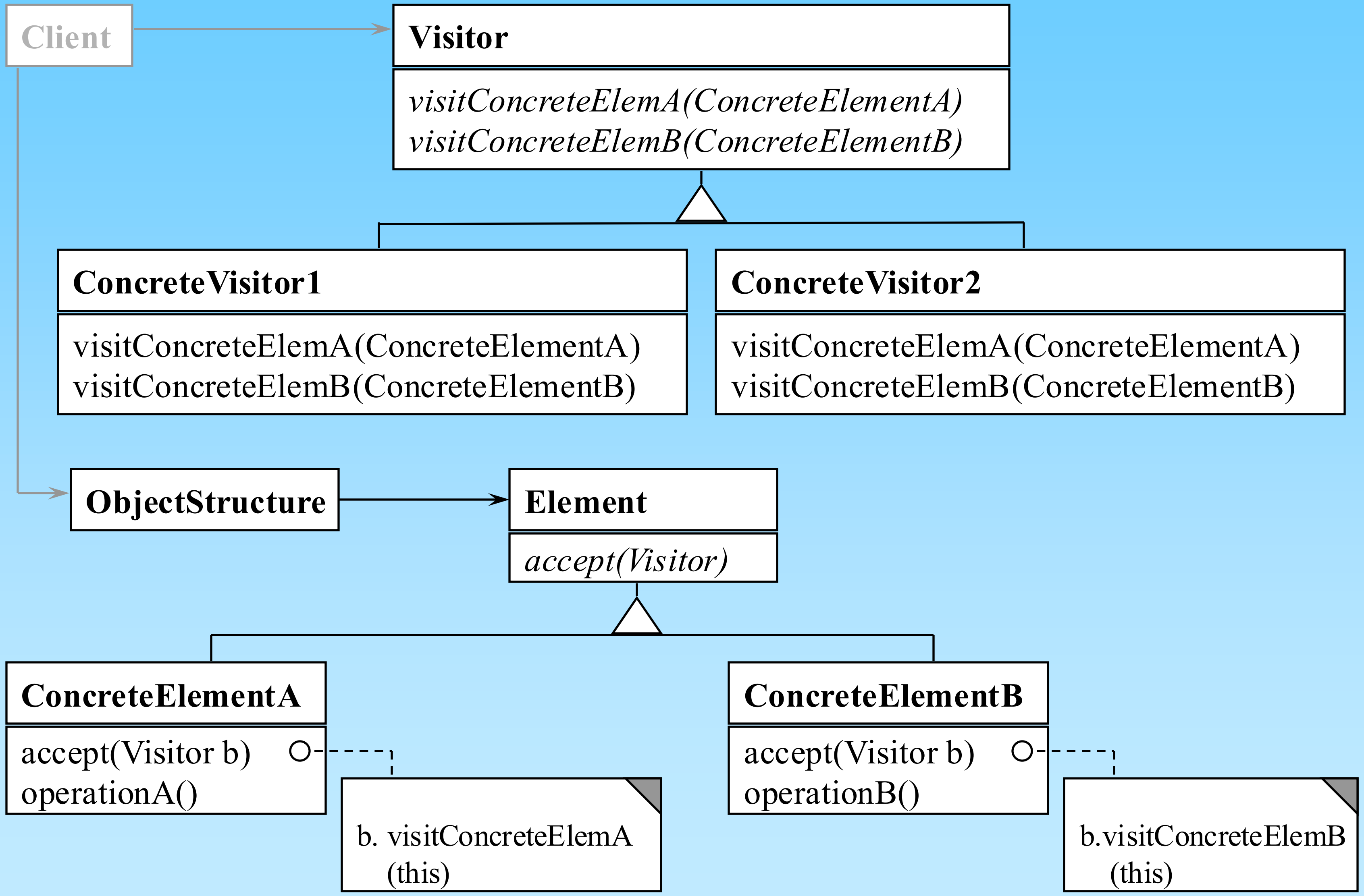
- When objects of several different classes share attributes or methods or interface elements.
- When objects of different classes need to be manipulated uniformly in one program.
- When extension with additional classes should be possible without changing existing code.
- Variant: Interface and implementations.

Visitor

Intent

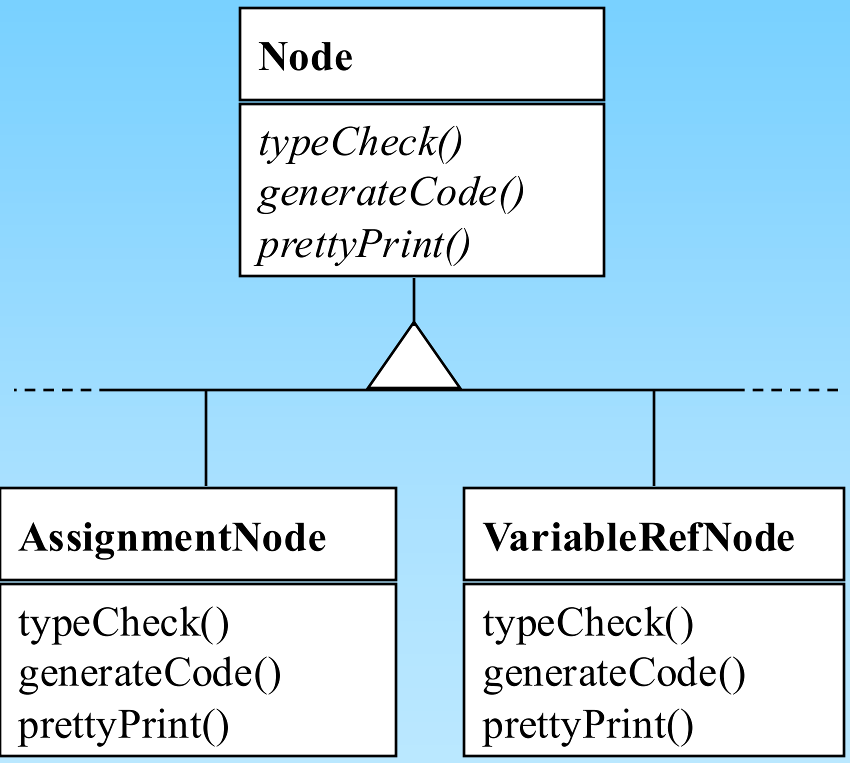
Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Structure of Visitor


Client
Visitor
visitConcreteElemA(ConcreteElementA)
visitConcreteElemB(ConcreteElementB)
ConcreteVisitor1
visitConcreteElemA(ConcreteElementA)
visitConcreteElemB(ConcreteElementB)
ConcreteVisitor2
visitConcreteElemA(ConcreteElementA)
visitConcreteElemB(ConcreteElementB)
ObjectStructure
Element
accept(Visitor)
ConcreteElementA
accept(Visitor b)
operationA()
b. visitConcreteElemA (this)
ConcreteElementB
accept(Visitor b)
operationB()
b.visitConcreteElemB (this)

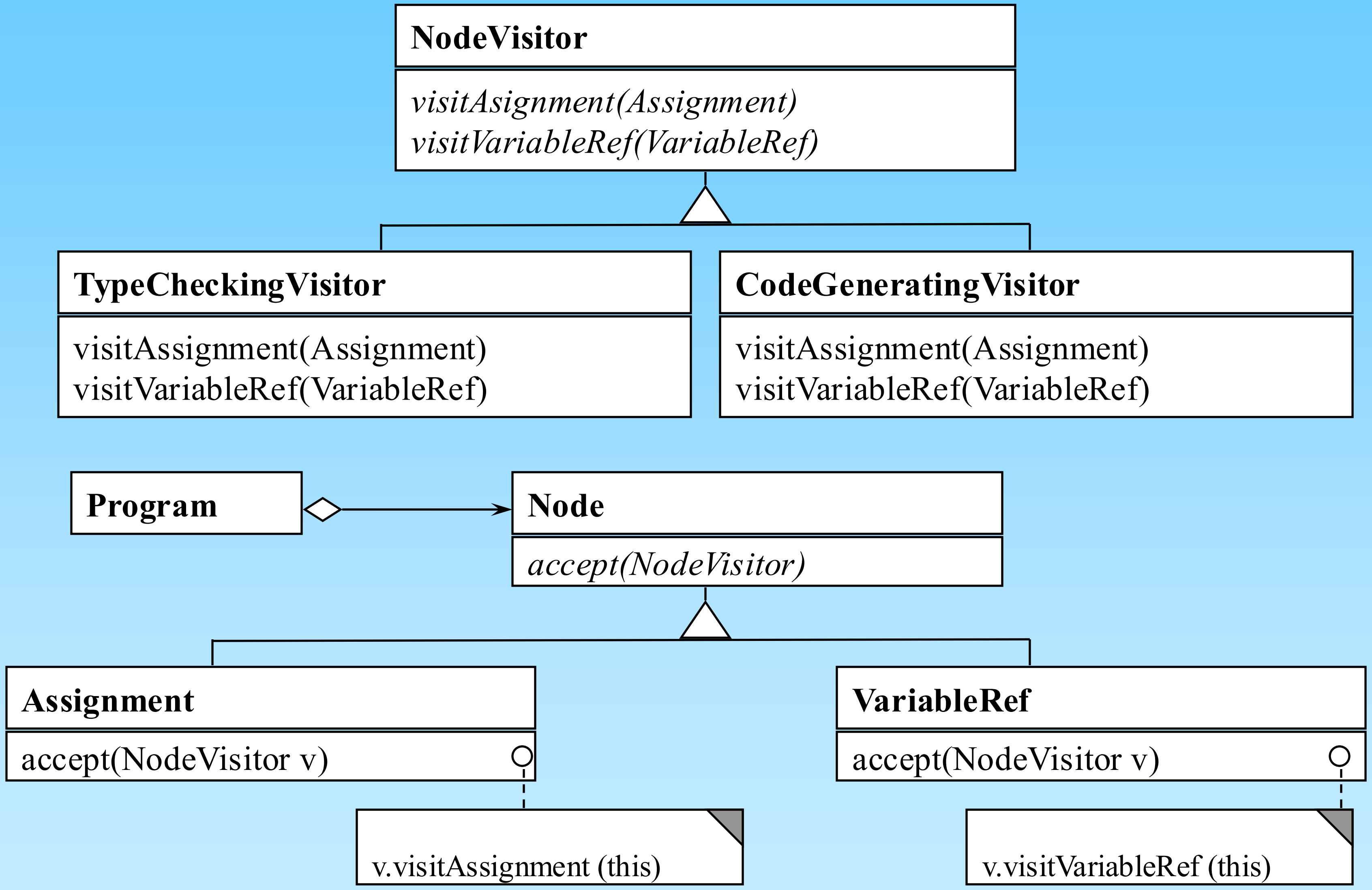
Example of Visitor

abstract syntax tree in compilers



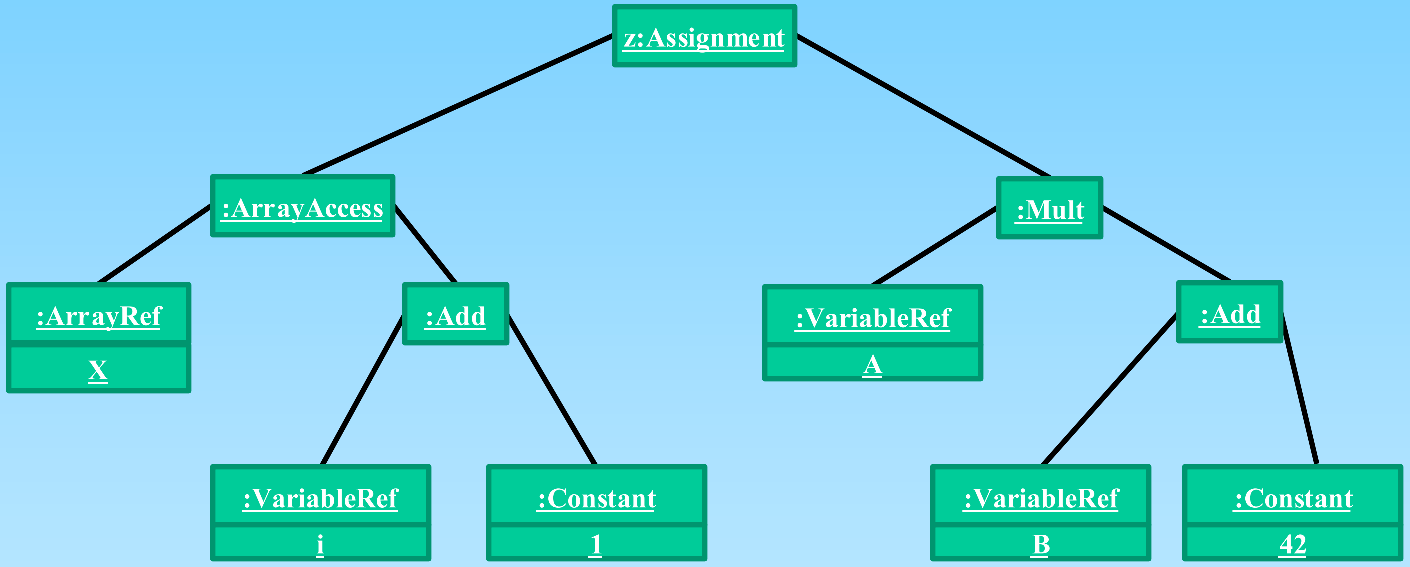
The operations are distributed among several classes. If introducing a new operation all these classes must be changed.

Example of Visitor


NodeVisitor
visitAsignment(Assignment)
visitVariableRef(VariableRef)
TypeCheckingVisitor
visitAssignment(Assignment)
visitVariableRef(VariableRef)
CodeGeneratingVisitor
visitAssignment(Assignment)
visitVariableRef(VariableRef)
Program
Node
accept(NodeVisitor)
Assignment
accept(NodeVisitor v)
v.visitAssignment (this)
VariableRef
accept(NodeVisitor v)
v.visitVariableRef (this)

Example for an abstract syntax tree

```
X[i+1] = A * (B + 42)
```



What happens in the call `z.accept(new CodeGeneratingVisitor())` ?

Visitor

Applicability

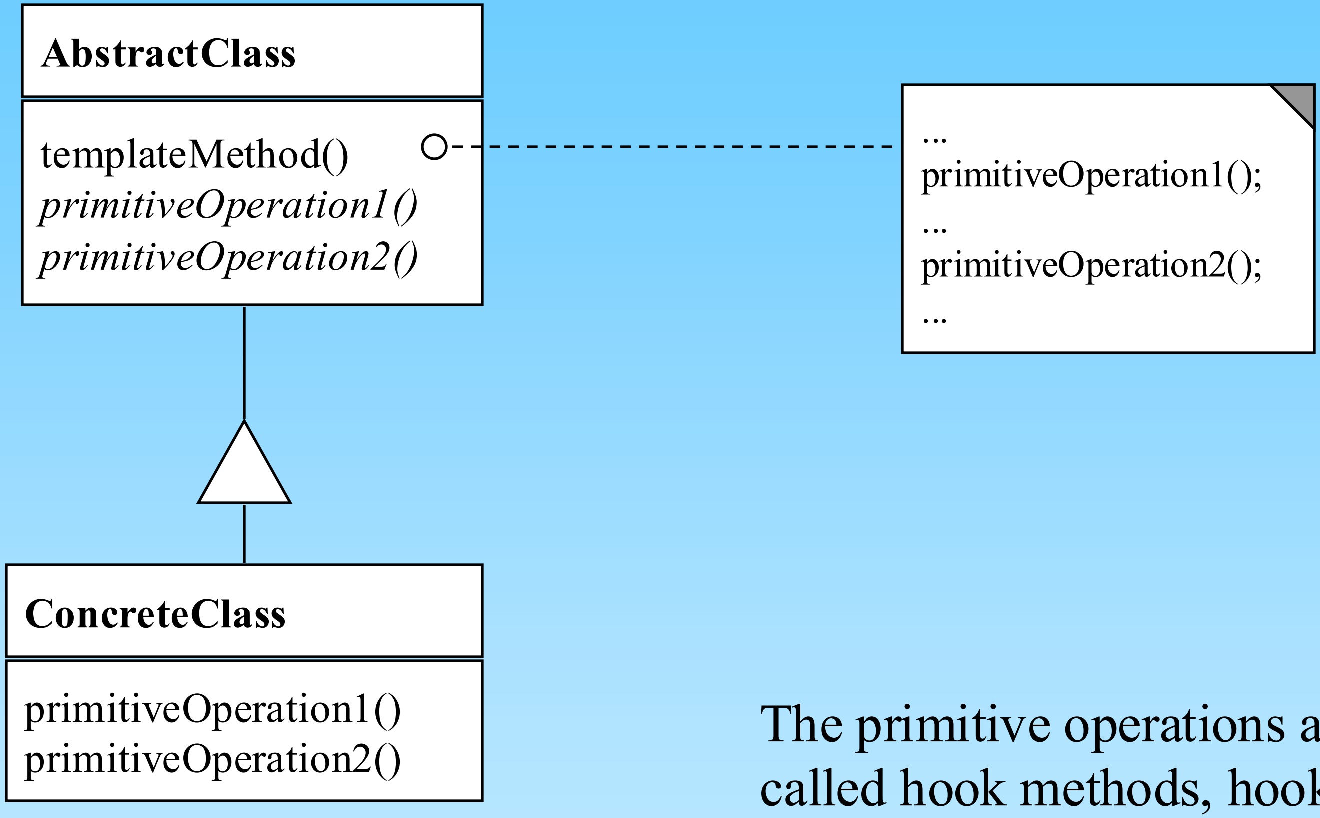
- When an object structure contains many classes of objects with differing interfaces, and operations on these objects that depend on their concrete classes should be performed.
- When many distinct and unrelated operations need to be performed on objects in an object structure, and "polluting" their classes with these operations should be avoided.
- When classes defining the object structure rarely change, but often new operations over the structure are defined.
- When several developers wish to extend the object structure simultaneously with several sets of operations.

Template Method

Intent

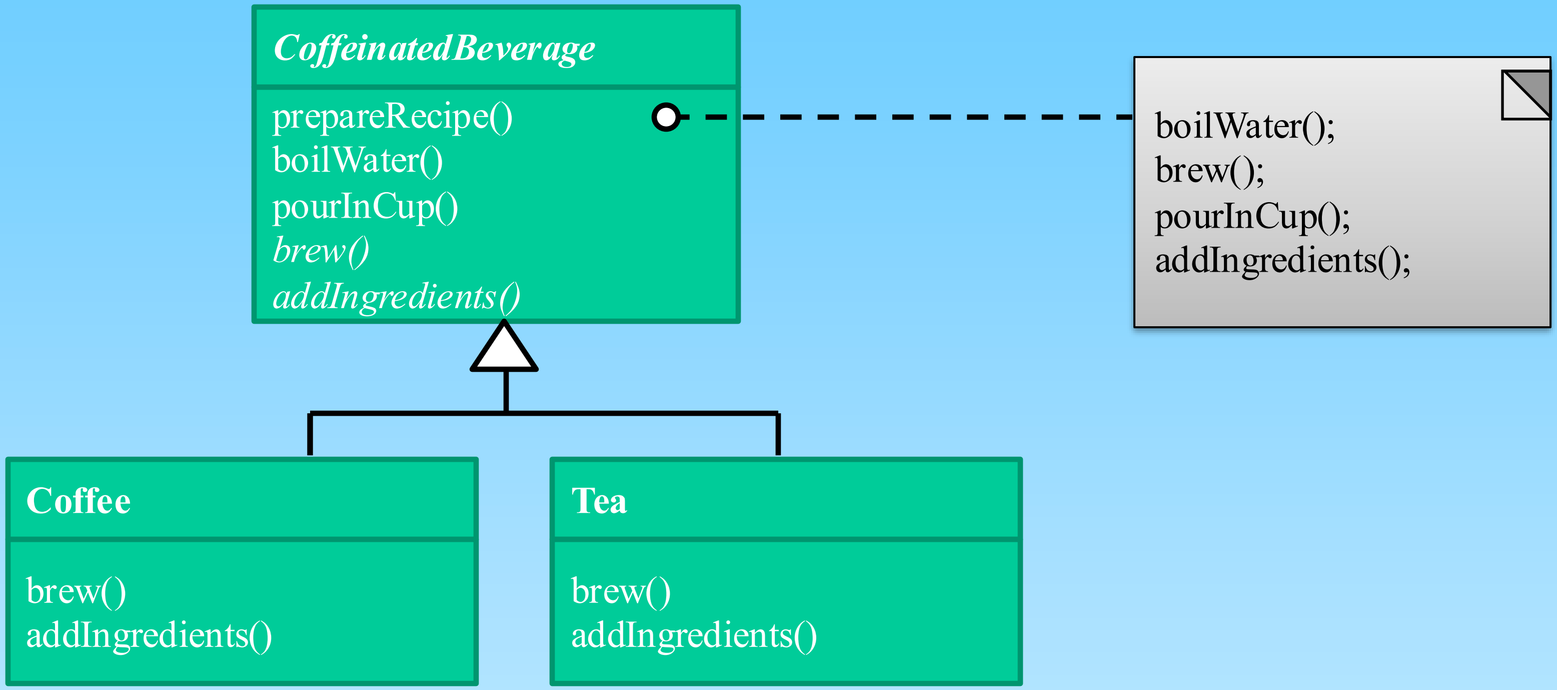
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm´s structure.

Structure of Template Method



The primitive operations are also called hook methods, hooks, or insertion methods

Example for Template Method



Source: Head First Design Patterns, page 280

Template Method

<u>Applicability</u>

- To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- When common behavior among subclasses should be factored and localized in a common class to avoid code duplication.
- To control subclasses extensions. For example, define a template method that calls “hook” operations at specific points, thereby permitting extensions only at those points.

Factory Method

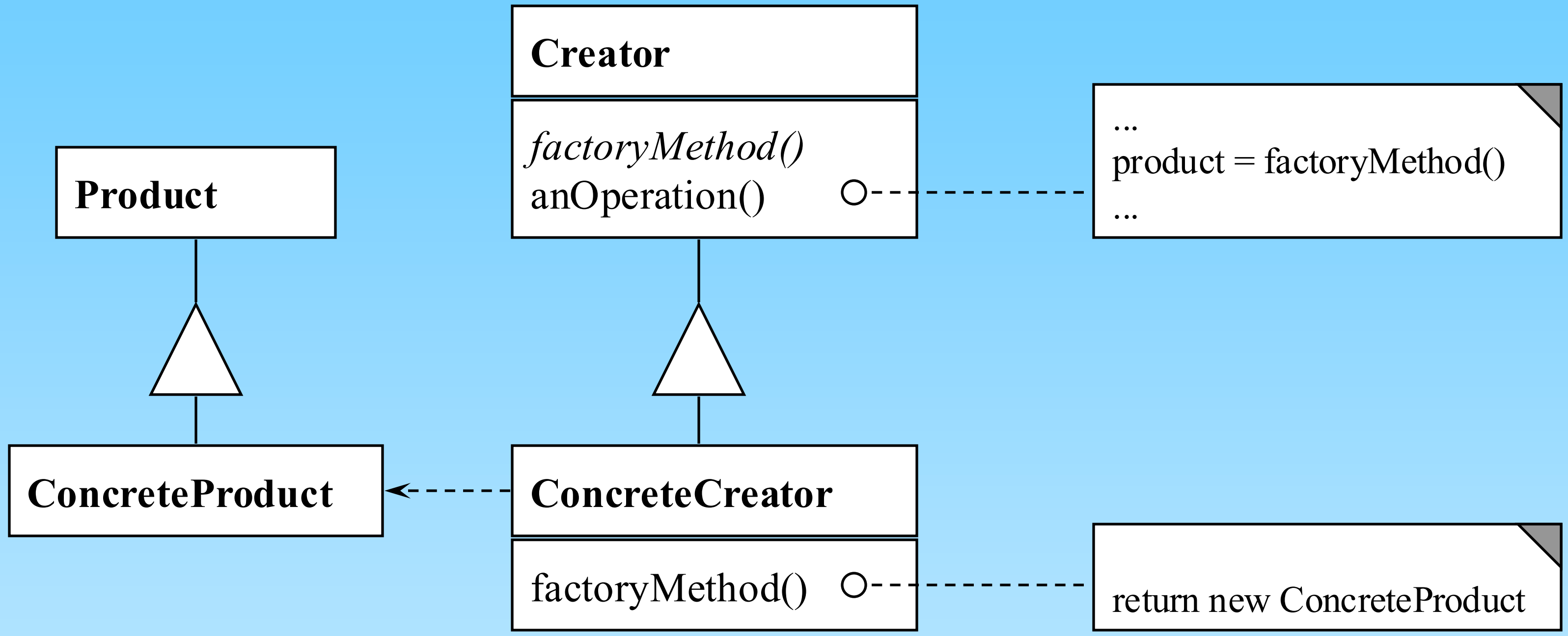
Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. Alterantively, the factory method decides which subclass to instantiate.

Also known as

Virtual Constructor

Structure of Factory Method



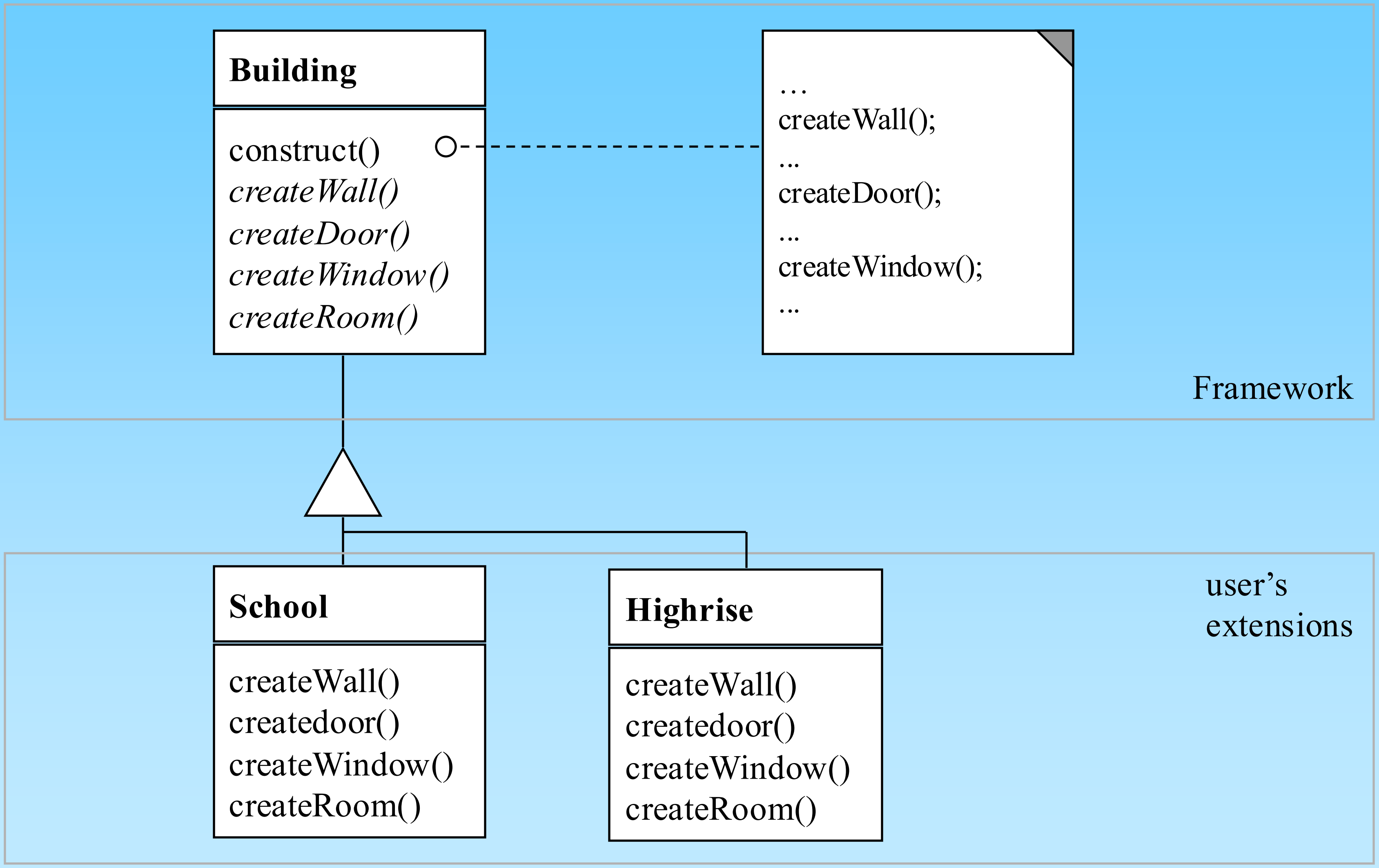
Example of Factory Method

Assume, Archi is a framework for planning buildings. It contains a class *Building*, which contains a ***construct*** method. This method is an interactive program to create a plan for a new building.

It is allowed for users to implement subclasses of Building. These subclasses must implement the abstract methods ***createWall***, ***createRoom***, ***createDoor***, ***createWindow.***

Problem: How does the user tell the name of the subclasses to the Framework?

Example of Factory Method



Example of Factory Method

Solution 1: Pass the class name *School* as character string and transform it to the corresponding class.

in Java: ***`Class.forName(String name)`***

```
construct() {
    String className = fetchClassName()
    // fetchClassName reads from a  manifest
    // or uses a parameter, or asks interactively
    bldg = Class.forName(className).newInstance();
    //above is deprecated because of exceptions. Better:
    Class.forName(className).getDeclaredConstructor().newInstance()
    bldg.construct();
    ...
}
```

Example of Factory Method

Solution 2: Subclasses of *Building* are pre-determined in the Framework; parameterize **`construct()`** accordingly.

```
construct (BuildingType t) {
    switch (t) {
        case SCHOOL:  bldg = (Building) new School(); break;
        case BANK:    bldg = (Building) new Bank(); break;
        case HOME:    bldg = (Building) new Home(); break;
    }
    bldg.construct();
}
```

Disadvantage: subclasses are fixed.

Factory Method

Applicability

- When a class can´t anticipate the class of objects it must create.
- When a class wants its subclasses to specify the objects it creates.
- When classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

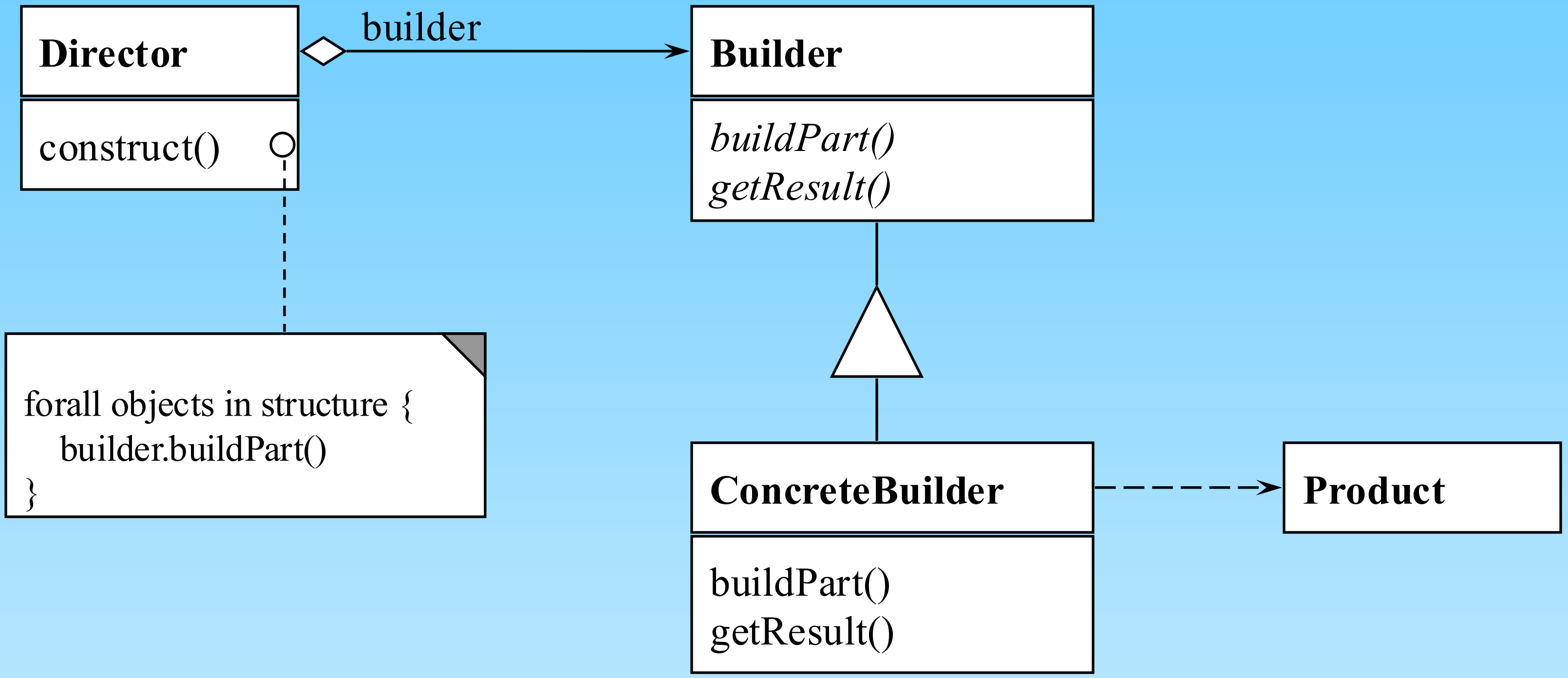
Factory Method is Template Method applied to object creation (some confusion in terminology: in Factory Method, the primitive operations are called factory methods. In Template Method, the primitive operations are called just that.)

Builder

Intent

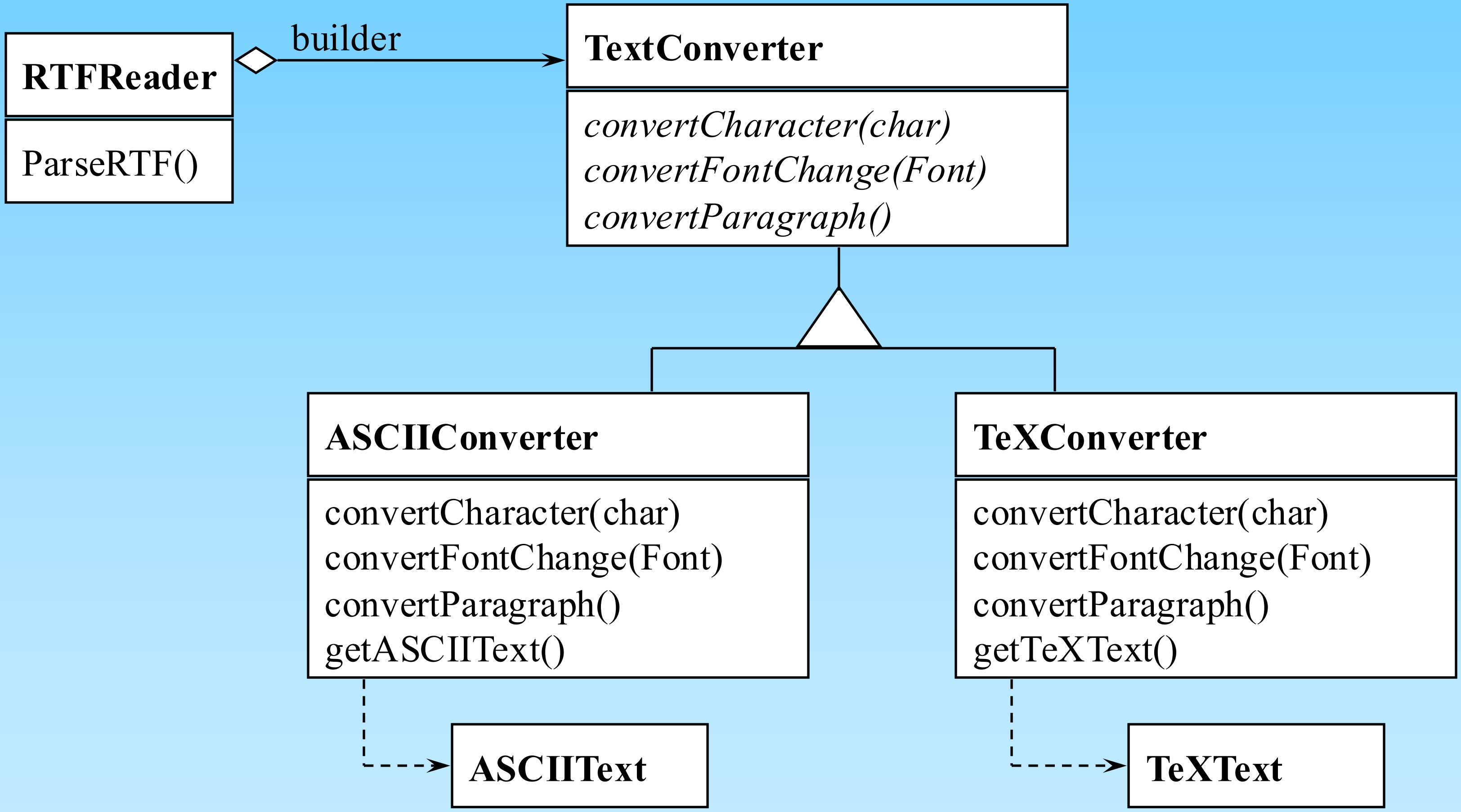
Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Structure of Builder



Example of Builder

Transformation between text formats



Builder

Applicability

- When the algorithm for creating a complex object should be independent of the parts that make up the object and how they´re assembled.
- When the construction process must allow different representations for the object that´s constructed.

Builder

Collaborations

- The client creates the Director object and configures it with the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

Builder vs. Factory Method

Builder separates the construction algorithm and the interface for building the individual parts (Director and Builder classes). Thus, it is possible to vary the construction algorithm dynamically.

Abstract Factory

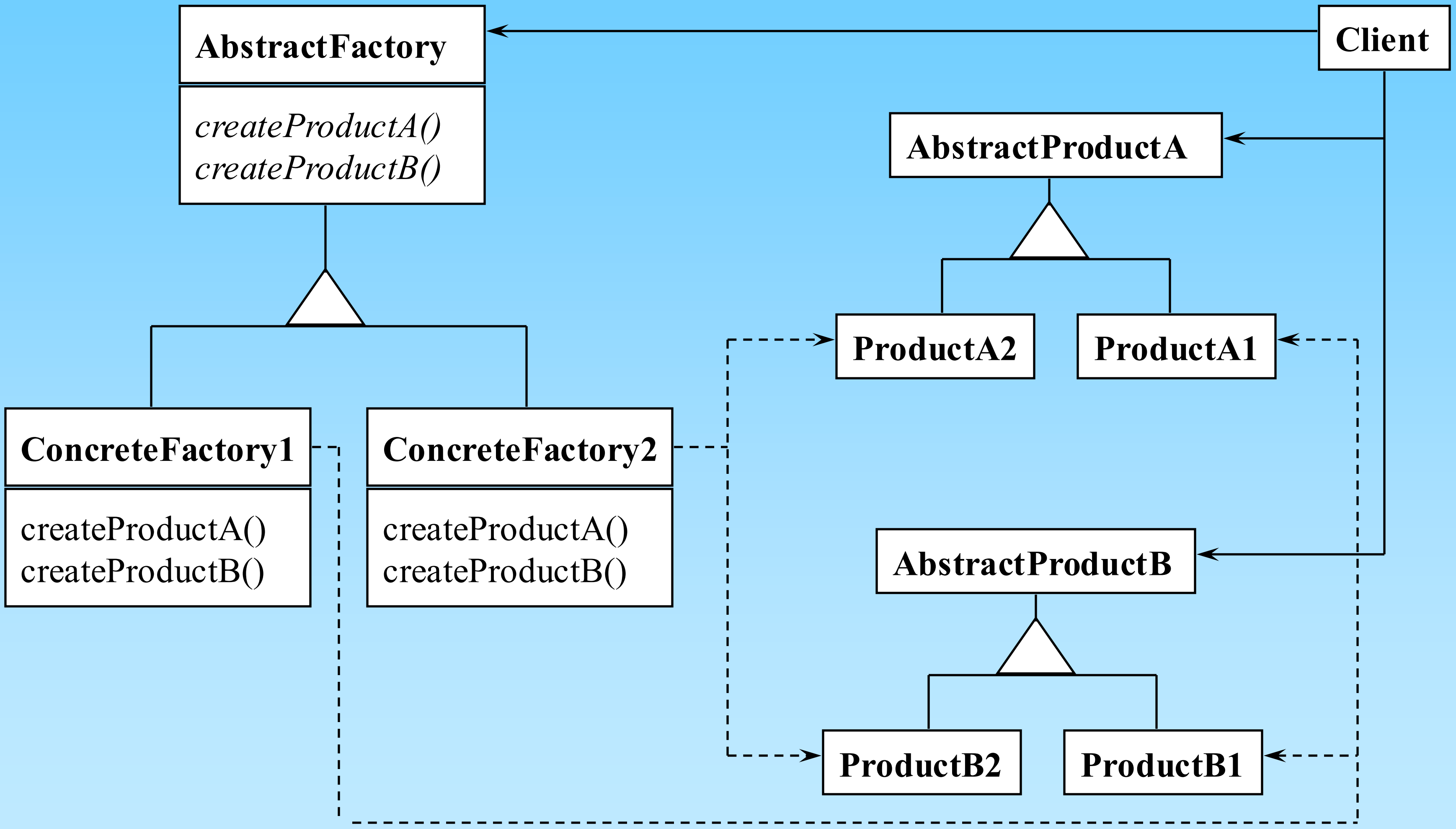
<u>Intent</u>

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

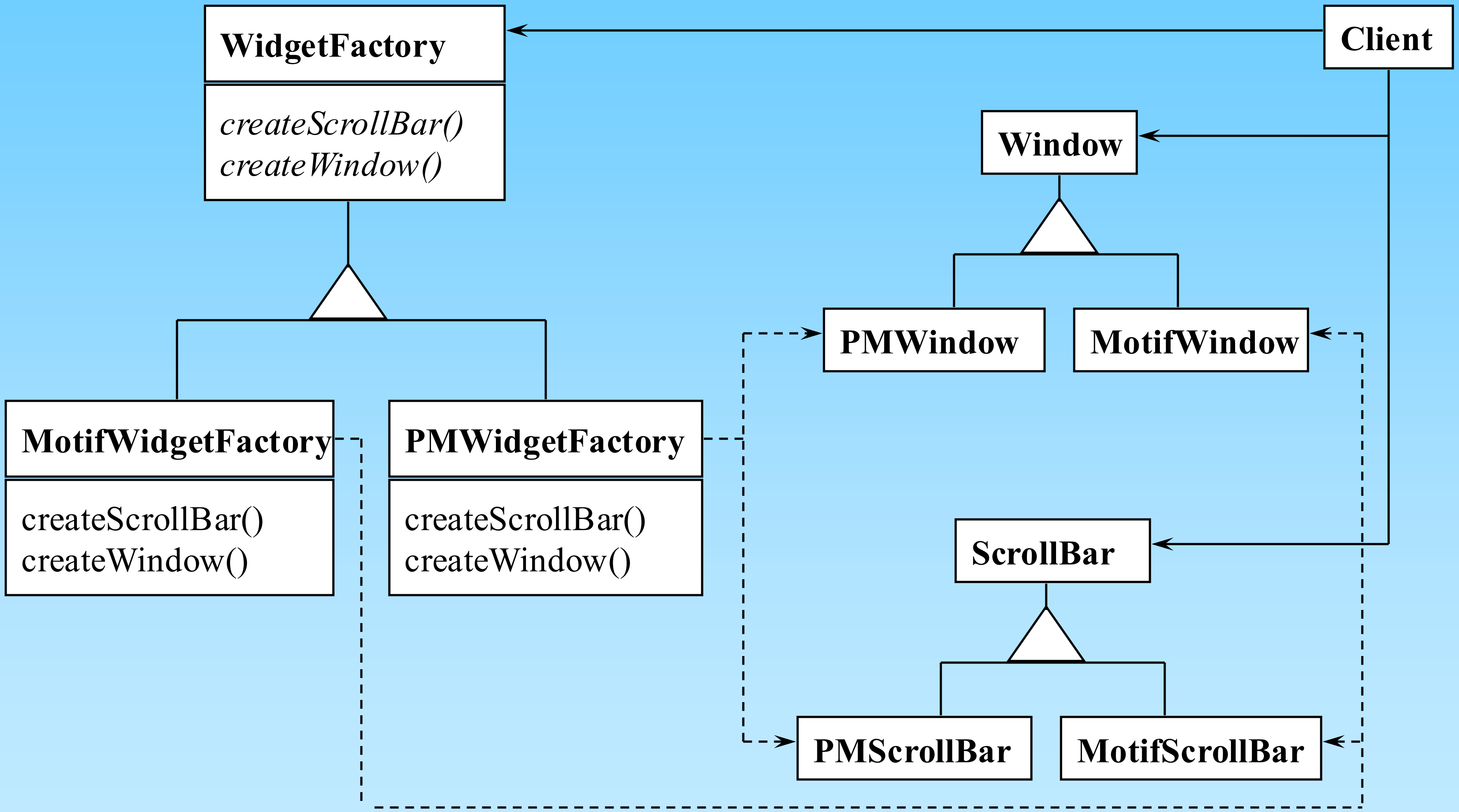
<u>Also known as</u>

Kit

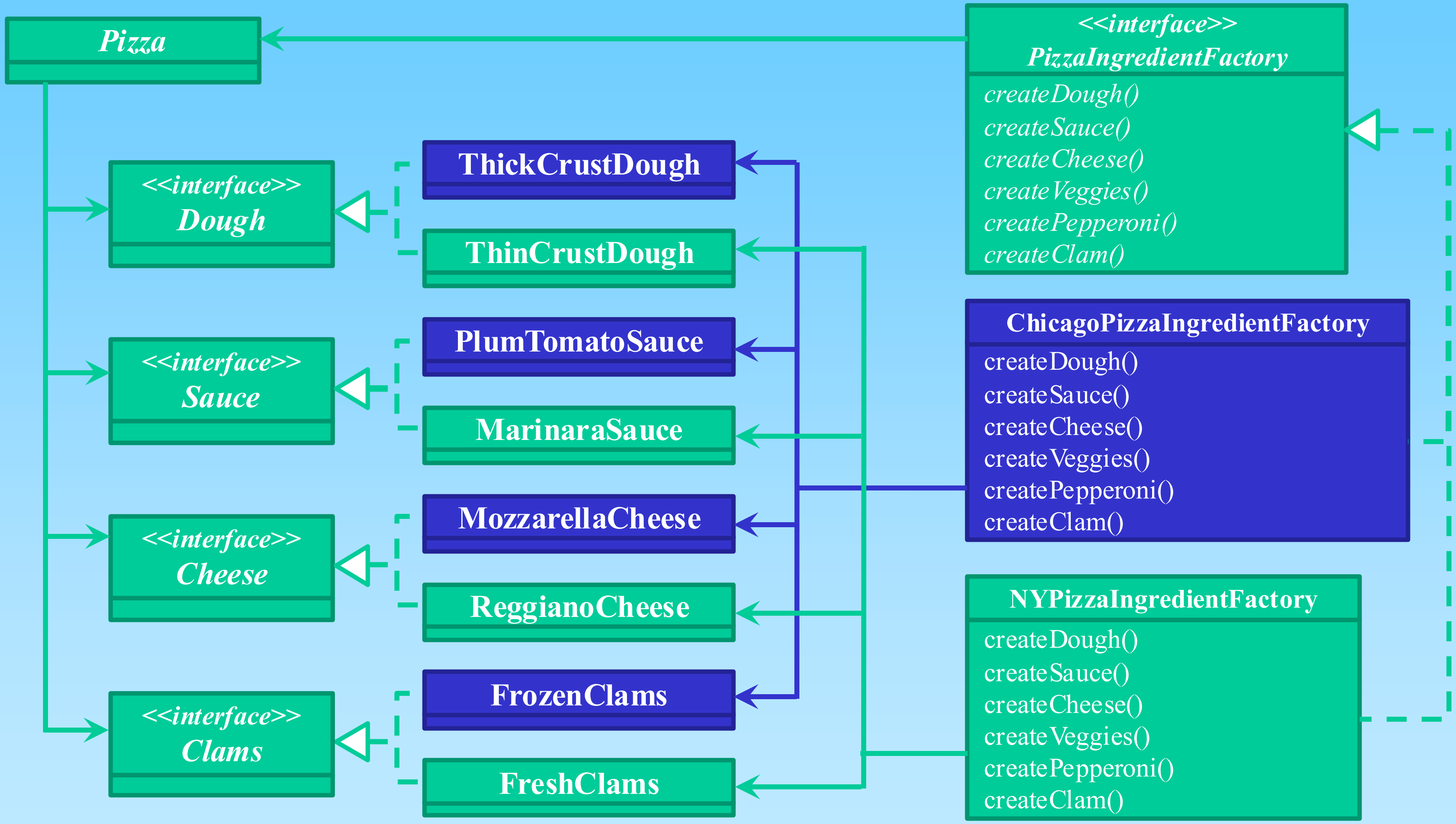
Structure of Abstract Factory



Example of Abstract Factory



Example for Abstract Factory



Source: Head First Design Patterns, page 157

Abstract Factory

Applicability

- When a system should be independent of how its products are created, composed, and represented.
- When a system should be configured with one of multiple families of products.
- When a family of related product objects is designed to be used together, and there is a need to enforce this constraint.
- When a class library of products should be provided, and just their interfaces should be revealed, not their implementations.

Builder vs. Abstract Factory

Abstract Factory is similar to Builder in that it too may construct complex objects. The primary difference is that the Builder pattern focuses on constructing a complex object step by step. Abstract Factory´s emphasis is on families of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory pattern is concerned, the product gets returned immediately.

3. State Handling

- Memento
- Prototype (Cloneable)
- Flyweight
- Singleton

Memento

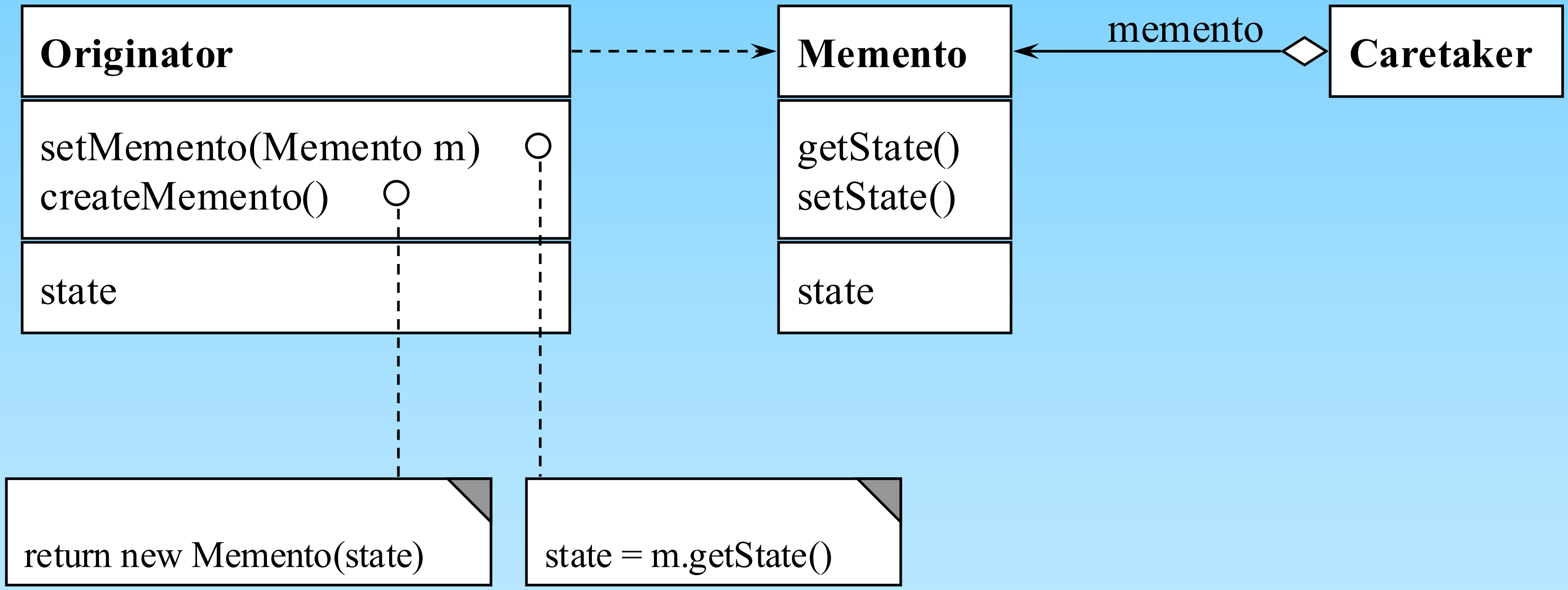
Intent

Without violating encapsulation, capture and externalize an objects´s internal state so that the object can be restored to this state later.

Also known as

Token

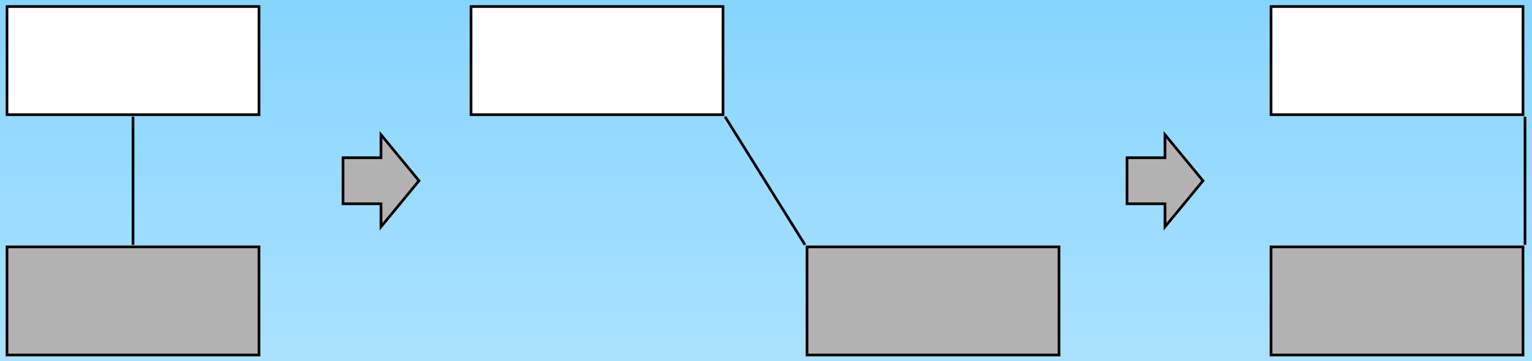
Structure of Memento



Example of Memento

complex checkpoints and undo mechanisms
e.g. in a graphical editor:

(Connectivity functionality is encapsulated in a ConstraintSolver object)



Rectangles are still connected, if one rectangle is moved.

A possible wrong result after an undo, if only the distance to original rectangle was saved.

Memento

Applicability

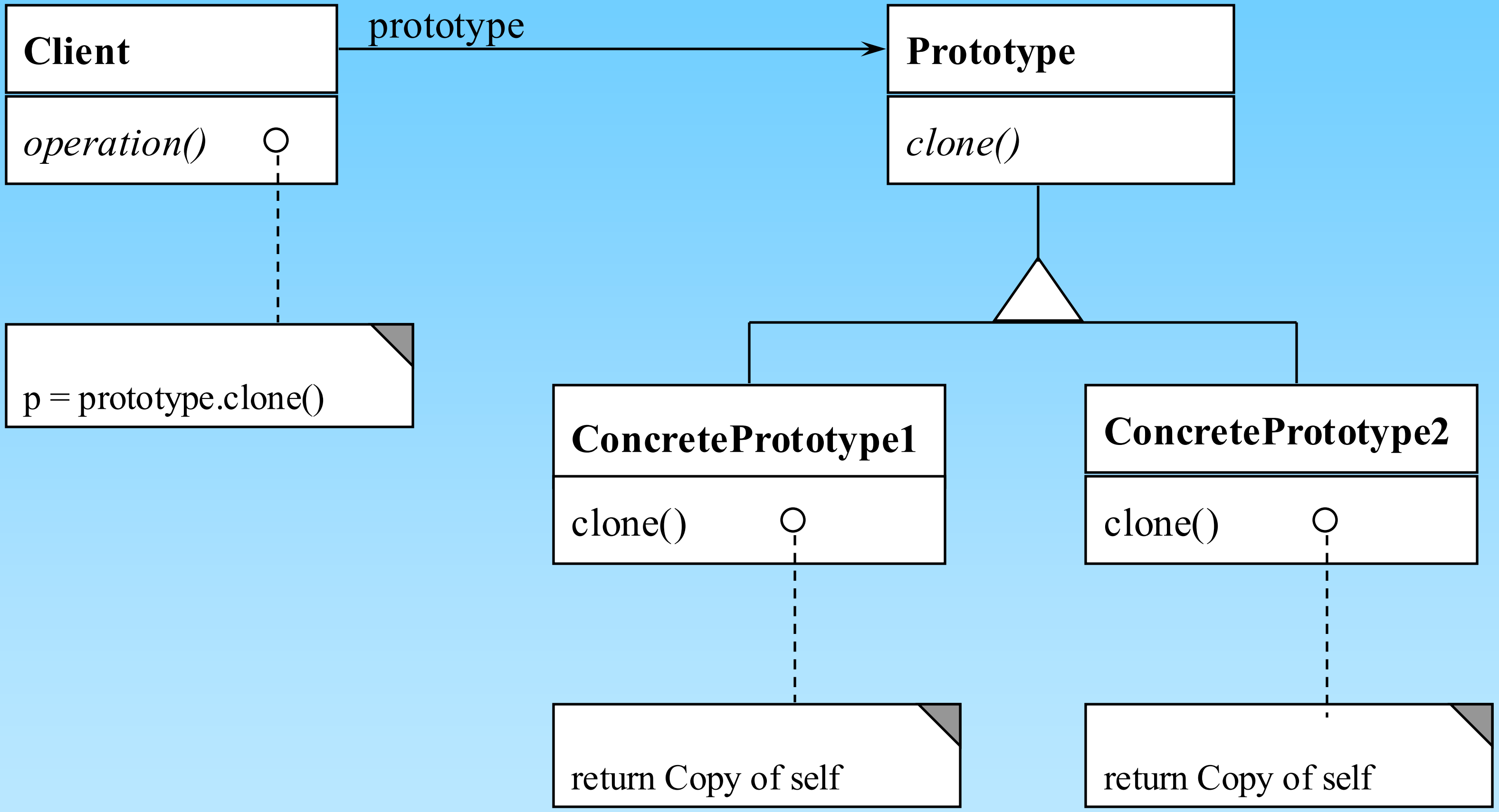
- When a snapshot (some portion of) an objects's state must be saved so that it can be restored to that state later, *and*
- when a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

Prototype

Intent

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

Structure of Prototype



Prototype

Applicability

The Prototype pattern is used when a system should be independent of how its products are created, composed, and represented; *and*

- when the classes to instantiate are specified at run-time, for example, by dynamic loading; *or*
- to avoid building a class hierarchy of factories that parallels the class hierarchy of products; *or*
- when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class, each time with the appropriate state.

Prototype vs. Factory Method

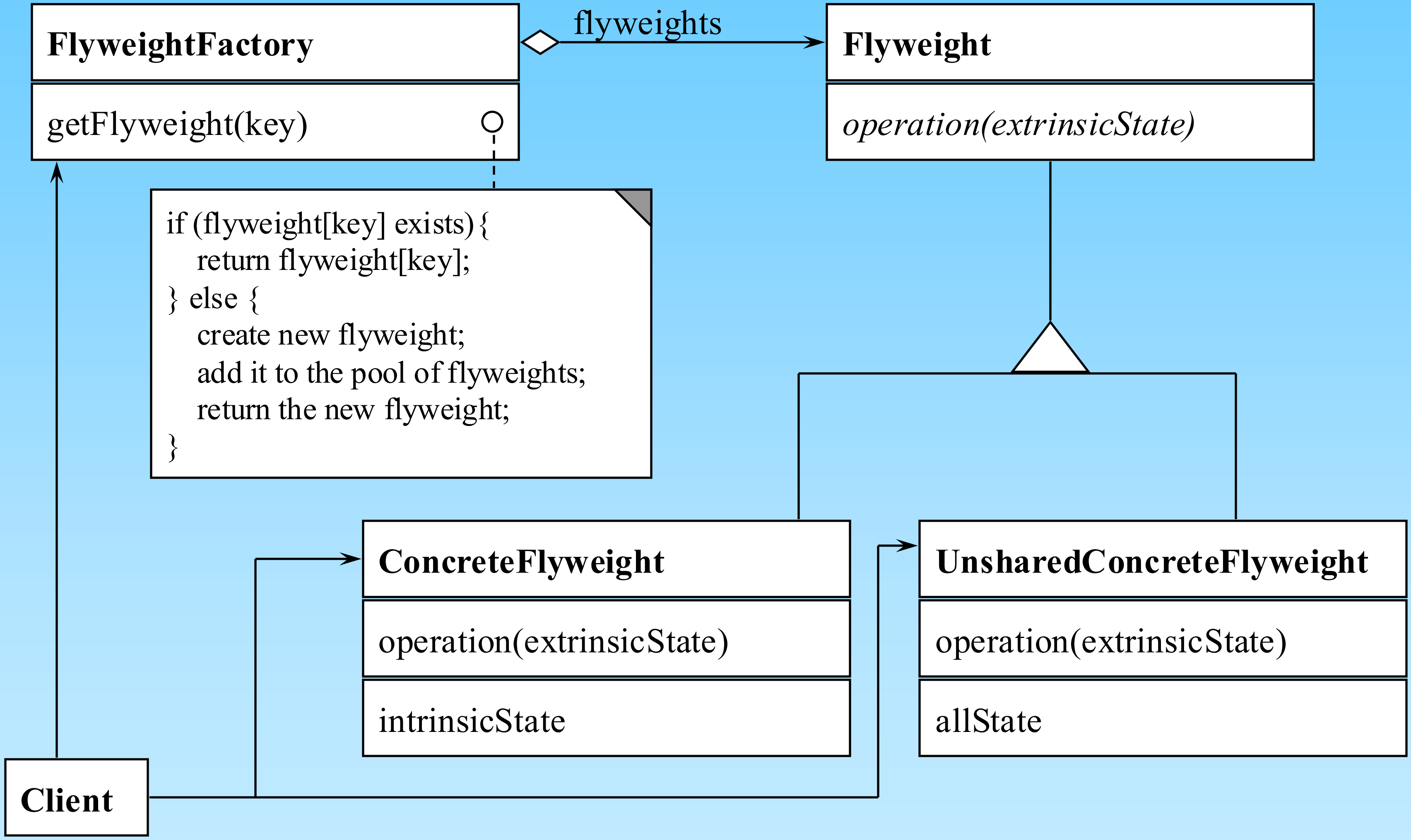
- Use Prototype, when creating an object needs much more time than creating a copy.
- Don´t use Prototype when the prototypical objects are too big or too numerous and thus use up too much memory.

Flyweight

Intent

Use sharing to support large numbers of fine-grained objects efficiently.

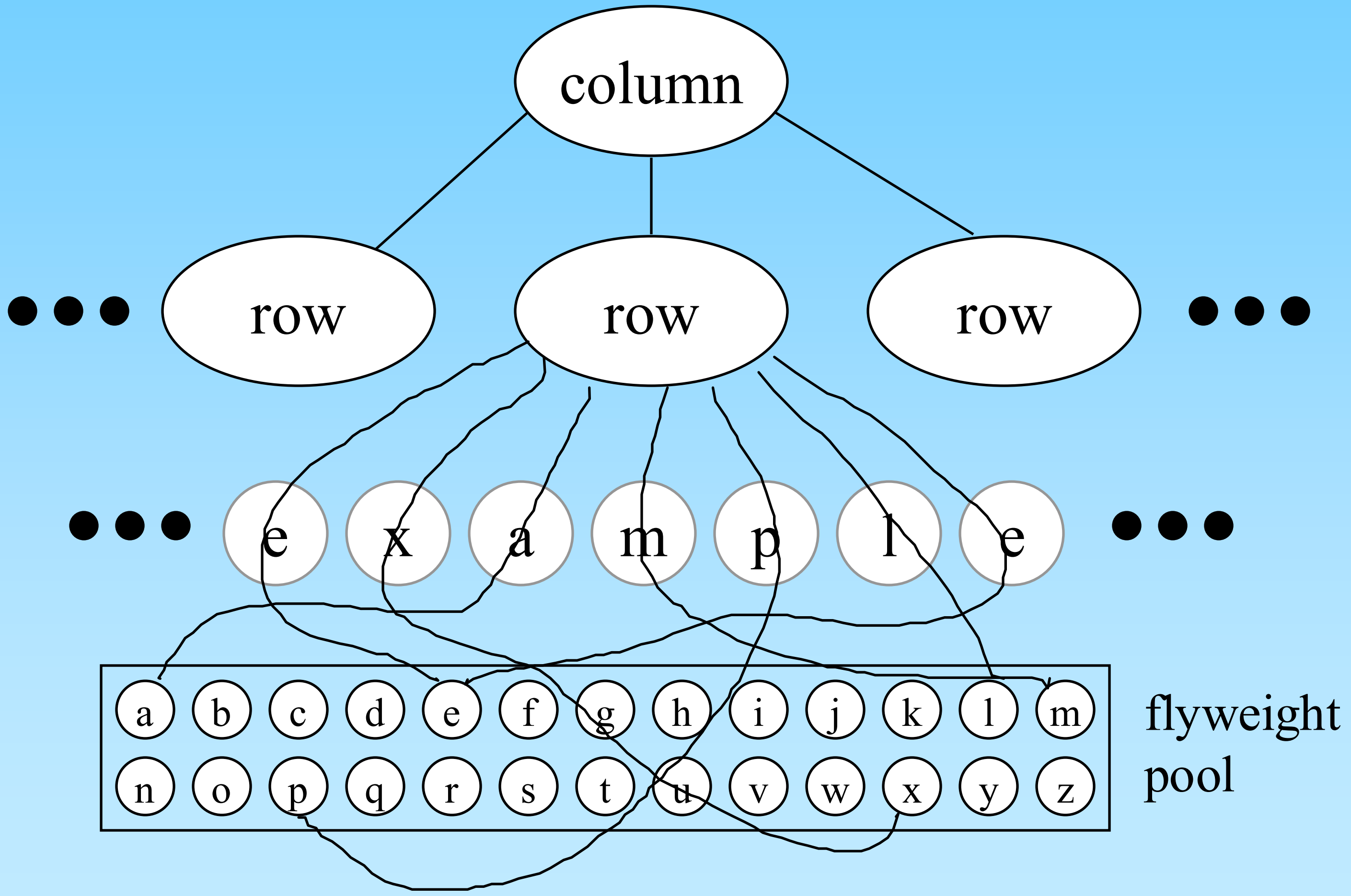
Structure of Flyweight


FlyweightFactory
getFlyweight(key)
flyweights
Flyweight
operation(extrinsicState)
ConcreteFlyweight
operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
operation(extrinsicState)
allState
Client

```
if (flyweight[key] exists) {
    return flyweight[key];
} else {
    create new flyweight;
    add it to the pool of flyweights;
    return the new flyweight;
}
```

Example of Flyweight

Object modeling down to characters in a text editor:



Example of Flyweight

- The characters themselves can be represented by a character code (intrinsic state).
- The information about font, size, and position can be externalized (extrinsic state) and placed into the row or column object, or subsequences of characters. Font and size need not be stored in every character object but can be looked up in their container. The position of each character can be computed from the position of the container, as necessary.

Flyweight

Applicability

The Flyweight pattern's effectiveness depends heavily on how and where it is used. Apply the Flyweight pattern when all the following are true:

- An application uses many objects.
- Storage costs are high because of the sheer quantity of objects.
- Most object state can be made extrinsic and applies to many objects at once, or intrinsic state is heavy-weight and can be reused.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
- The application doesn't depend on the identity of flyweights.

Singleton

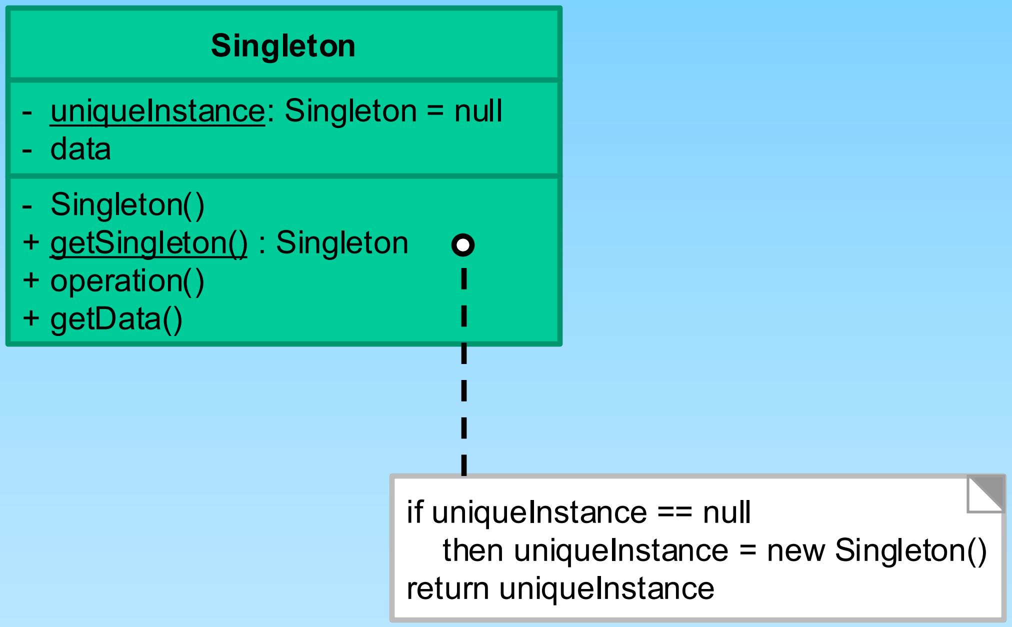
Intent

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

Motivation

The class itself is responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance.

Structure of Singleton


Singleton
- uniqueInstance: Singleton = null
- data
- Singleton()
+ getSingleton() : Singleton
+ operation()
+ getData()

```
if uniqueInstance == null
    then uniqueInstance = new Singleton()
return uniqueInstance
```

Singleton

Applicability

- When there must be exactly once instance of a class, and it must be accessible to clients from a well-known access point.
- When it is impractical or impossible to determine which part of an application creates the first instance.
- When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

4. Control

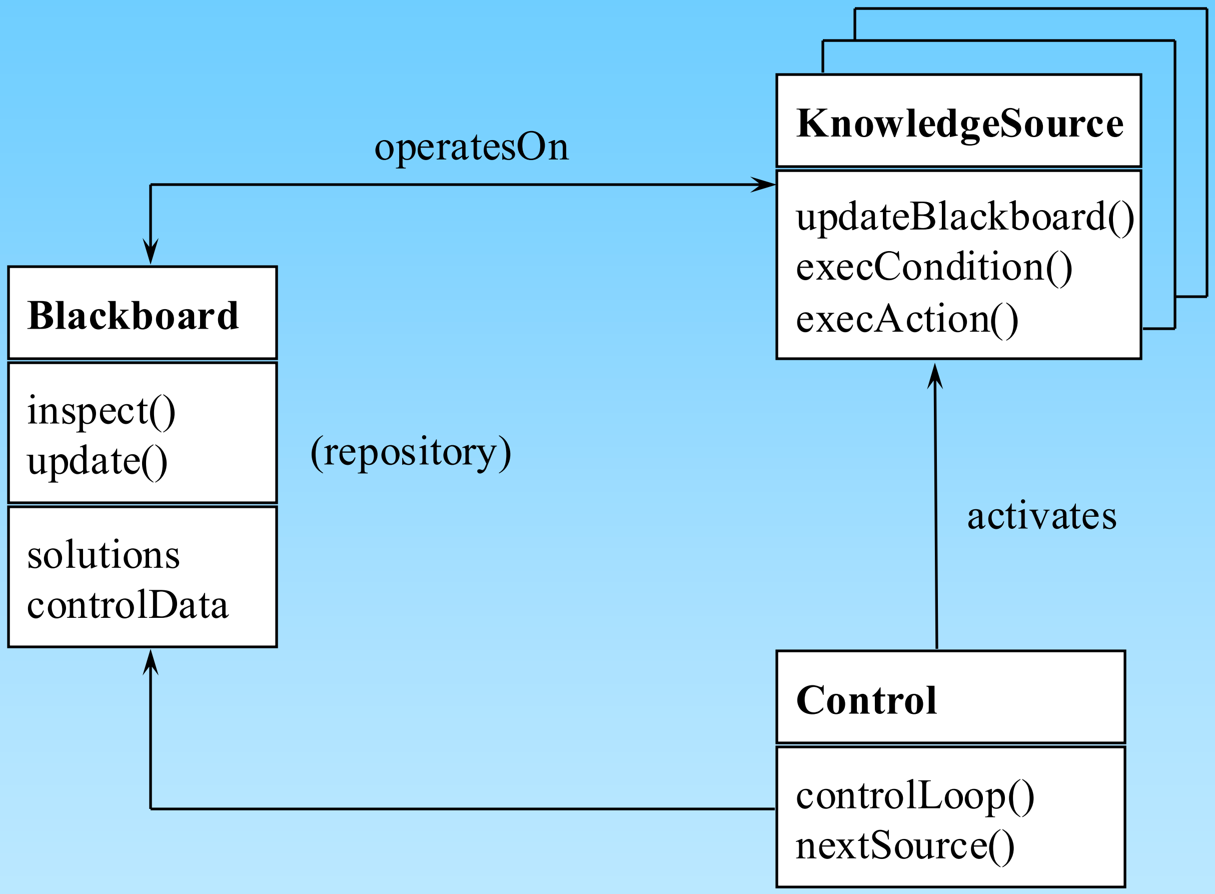
- Blackboard
- Command
- Chain of Responsibility
- Strategy
- Control State
- Master/Worker
- Process Control
  - Open-loop system
  - Closed-loop system
    - Feedback control
    - Feedforward control

Blackboard

Intent

The Blackboard architectural pattern is useful for problems for which no deterministic solution strategies are known. In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

Structure of Blackboard

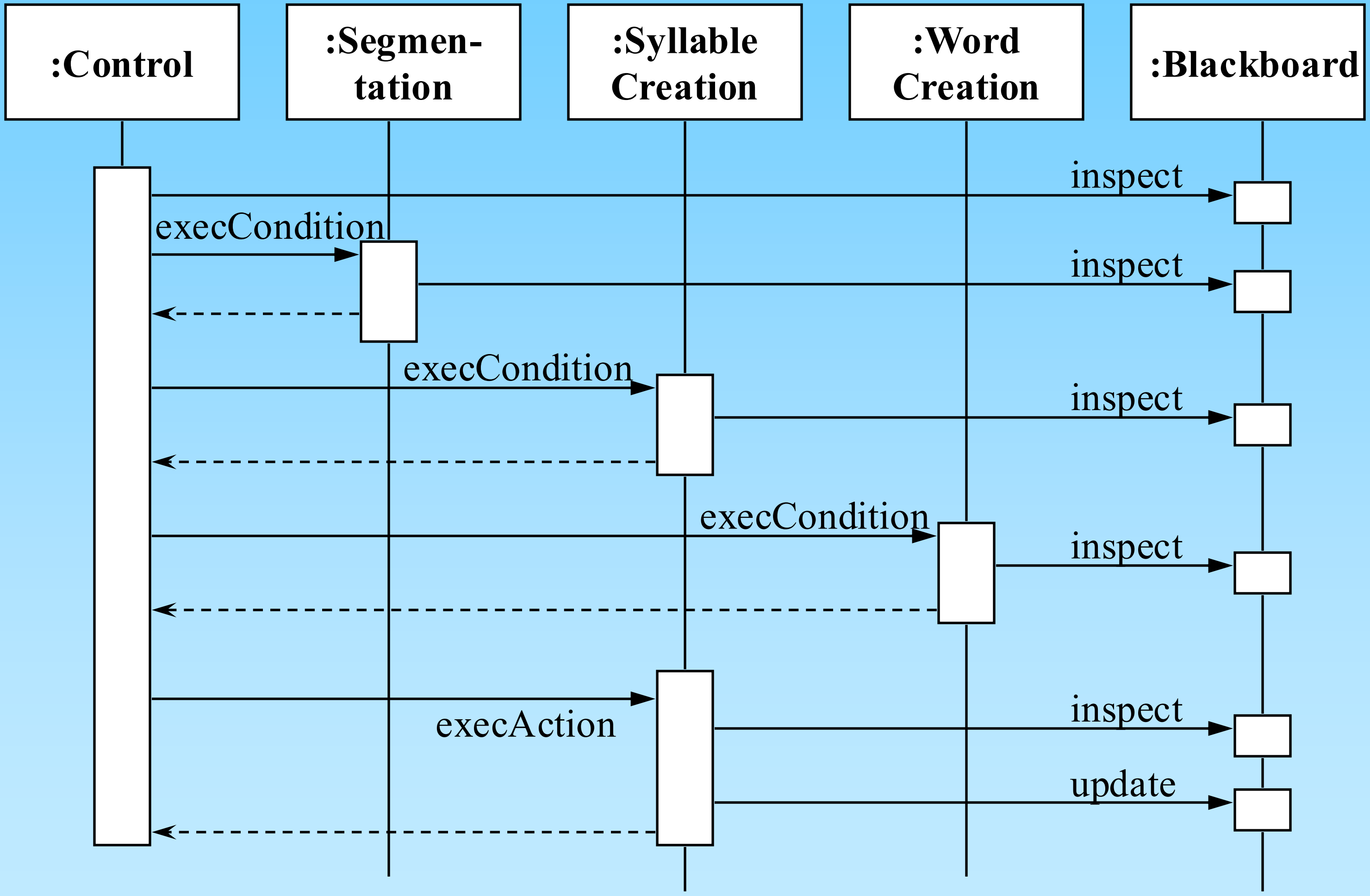


Blackboard

- Simple Form: Control activates all knowledge sources sequentially.
- Complex Form: Checks a set of applicable knowledge sources (with **`execCondition()`**), chooses one, and executes it.

Example of Blackboard

Speech Recognition



Blackboard

Applicability

- When several transformations (“knowledge sources”) operate on a common data structure (“blackboard”).
- When triggering the transformations is controlled by the contents of the data structure.
- When the selection of the applicable transformation should be controlled.

Command

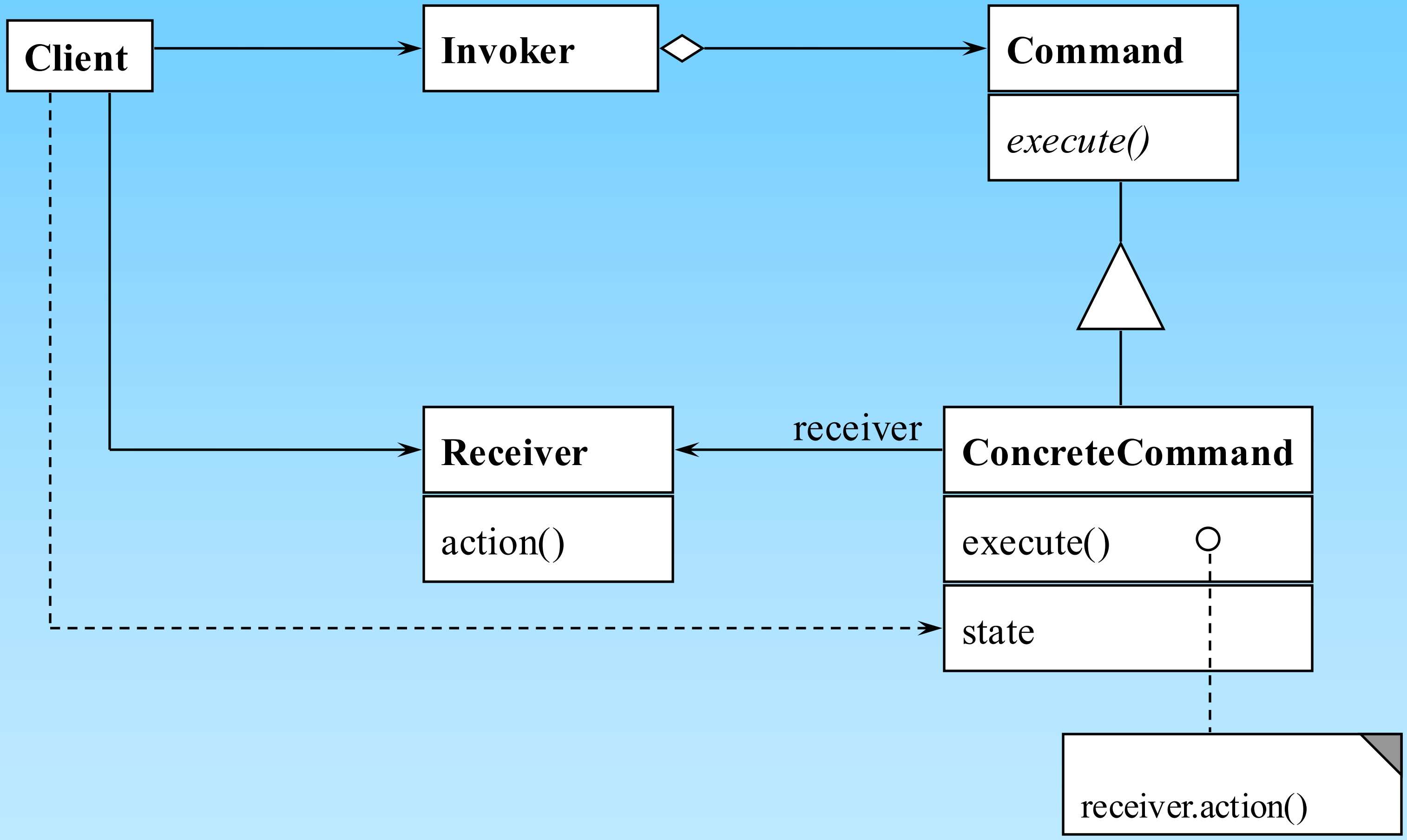
Intent

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Also known as

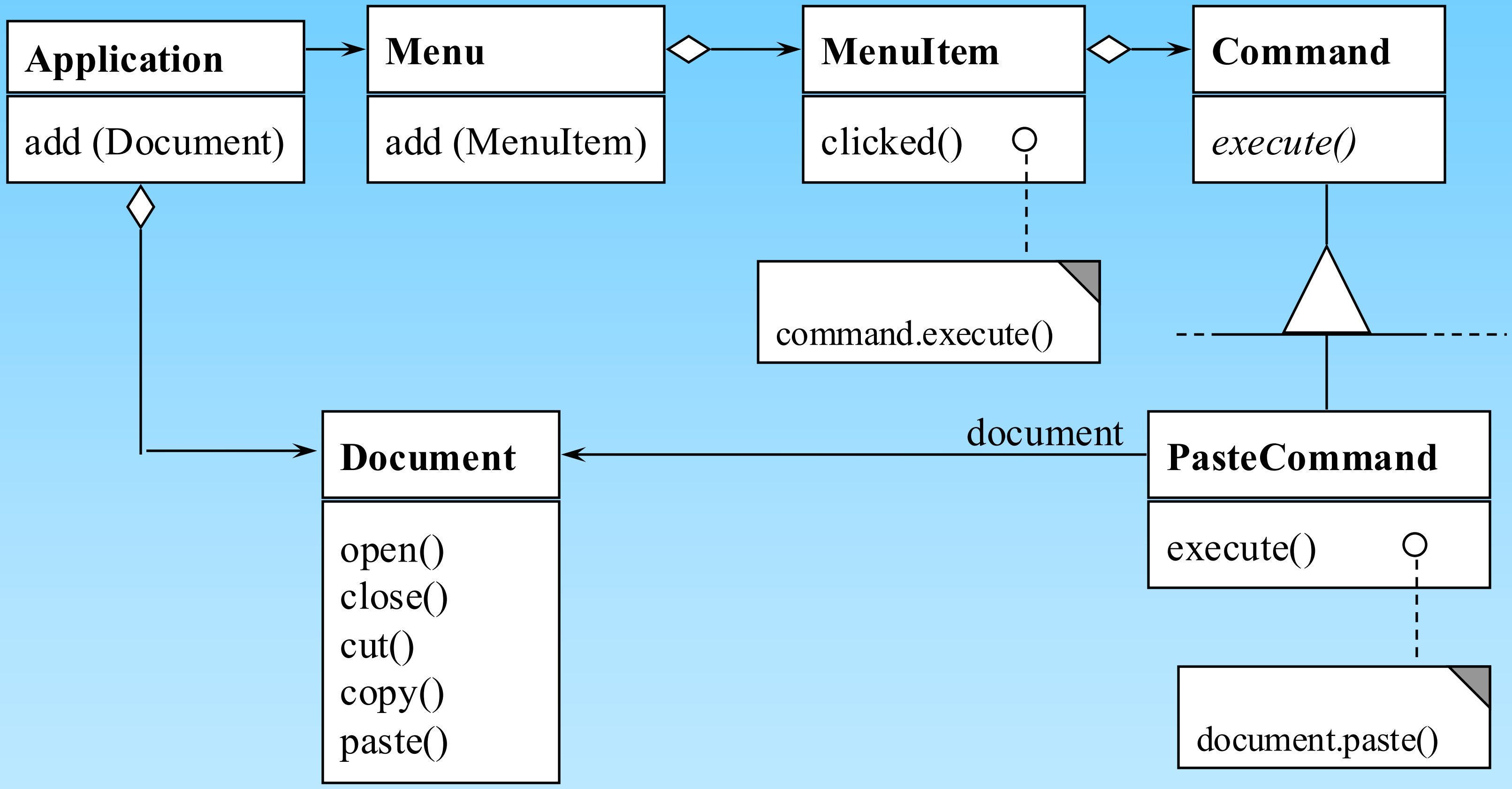
Action, Transaction

Structure of Command



Example of Command

Menus of user interfaces

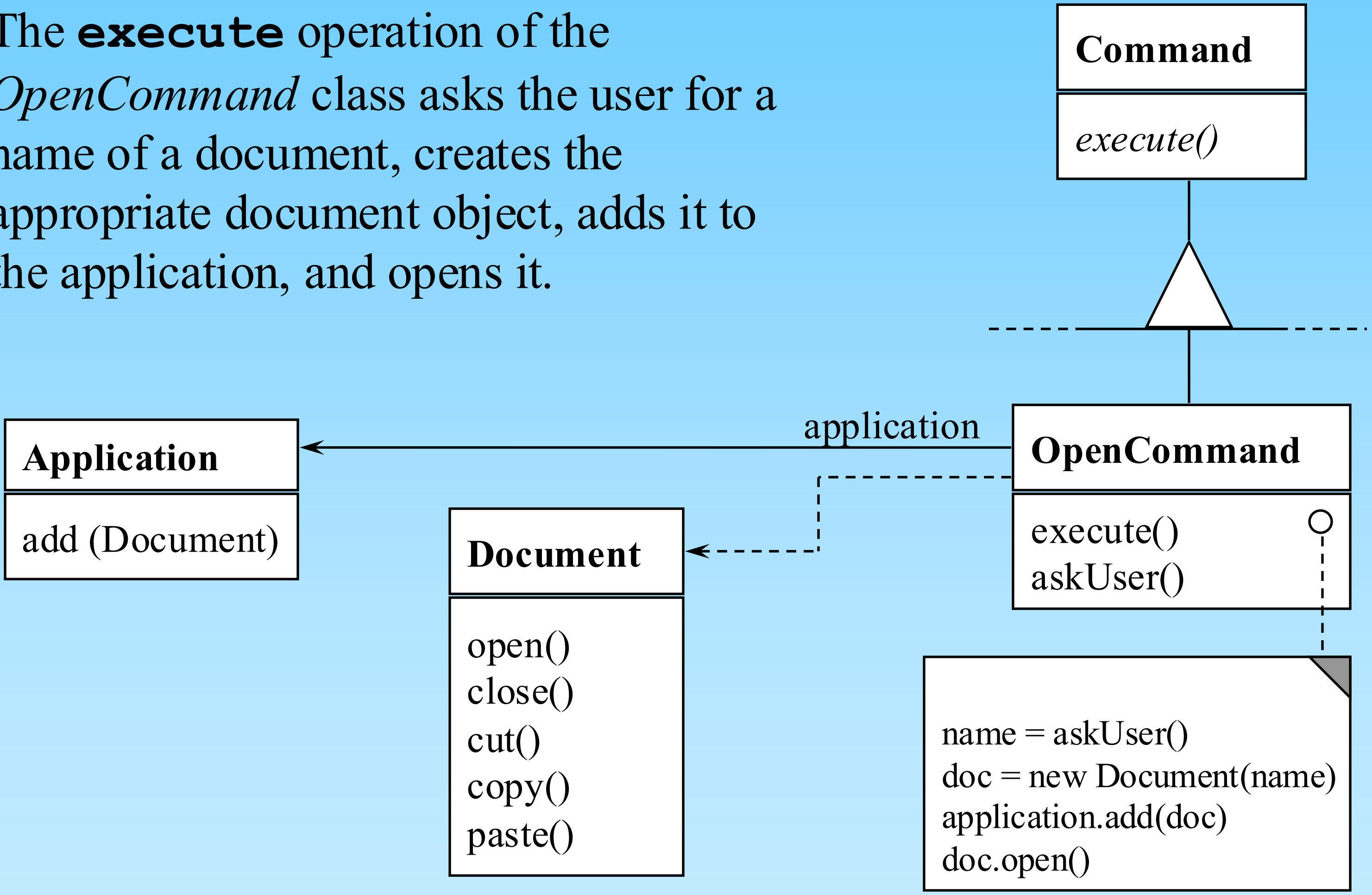


Command has an **`execute`** method and stores the object on which the operation is performed.

Example of Command

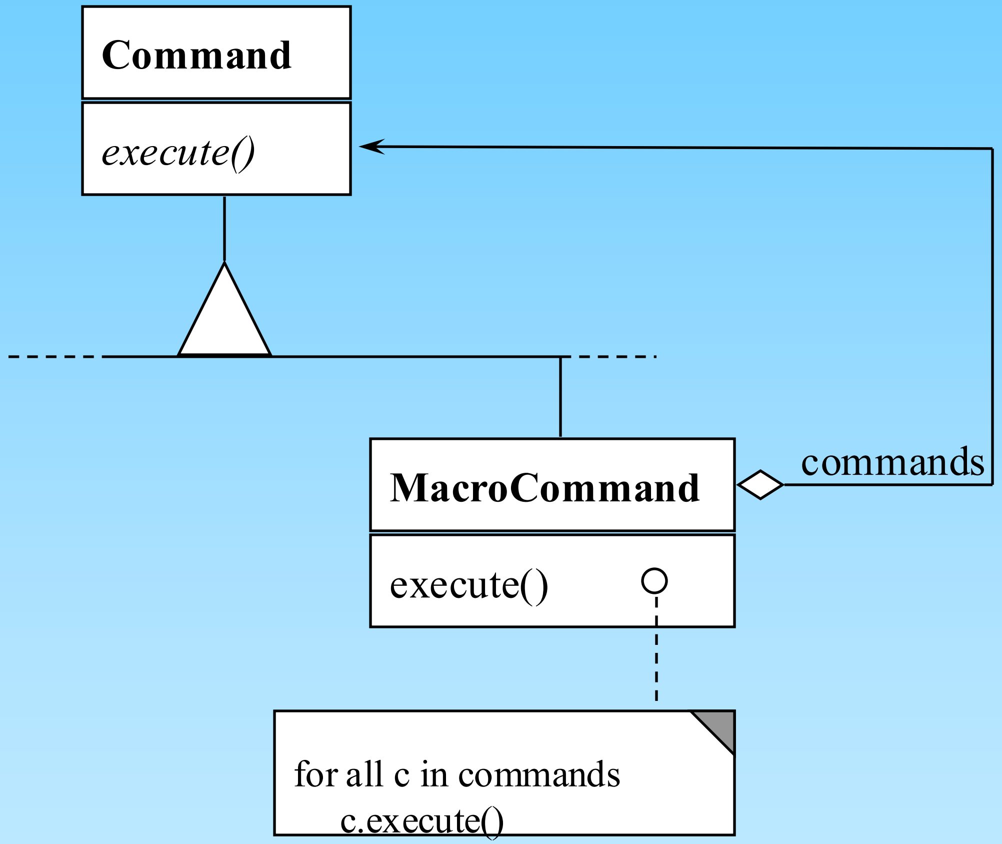
Command to open a document

The **`execute`** operation of the *OpenCommand* class asks the user for a name of a document, creates the appropriate document object, adds it to the application, and opens it.



Macro Command is a Composite Command

Sequence of Commands (MacroCommand)



Command

Applicability

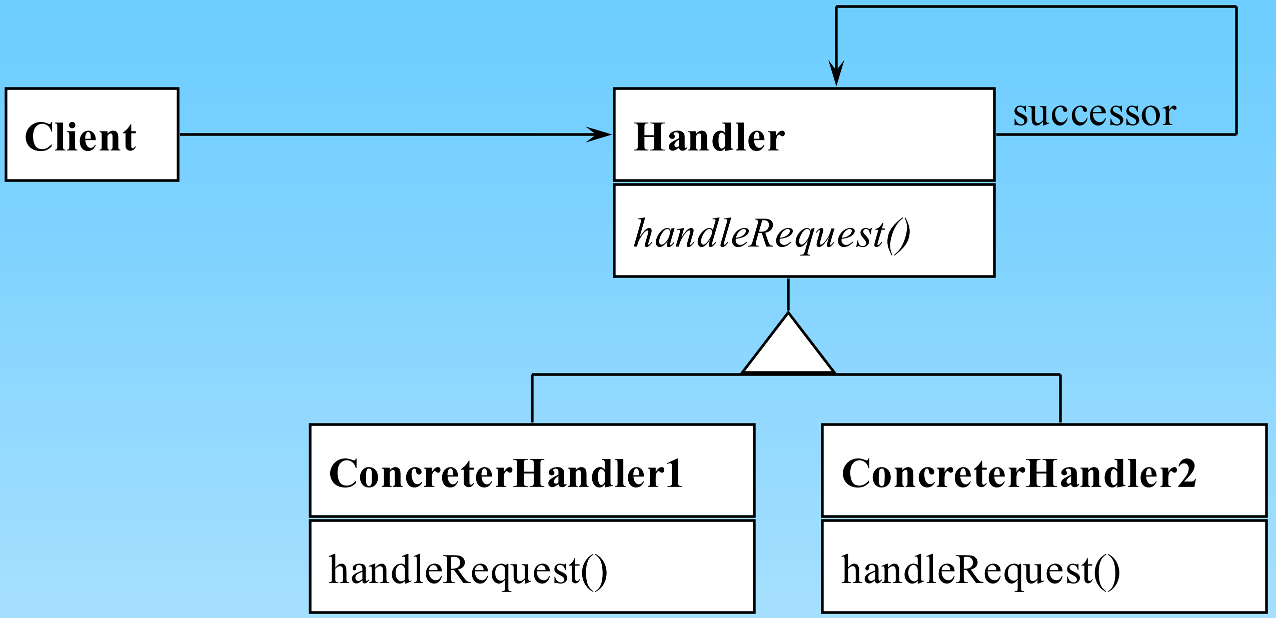
- When objects should be parameterized by an action to perform, as MenuItem objects did above.
- When requests should be specified, requested, and executed at different times.
- When an undo should be supported.
- When changes should be logged, so that they can be reapplied in case of a system crash.
- When a system should be structured around high-level operations built on primitive operations (macro command).

Chain of Responsibility

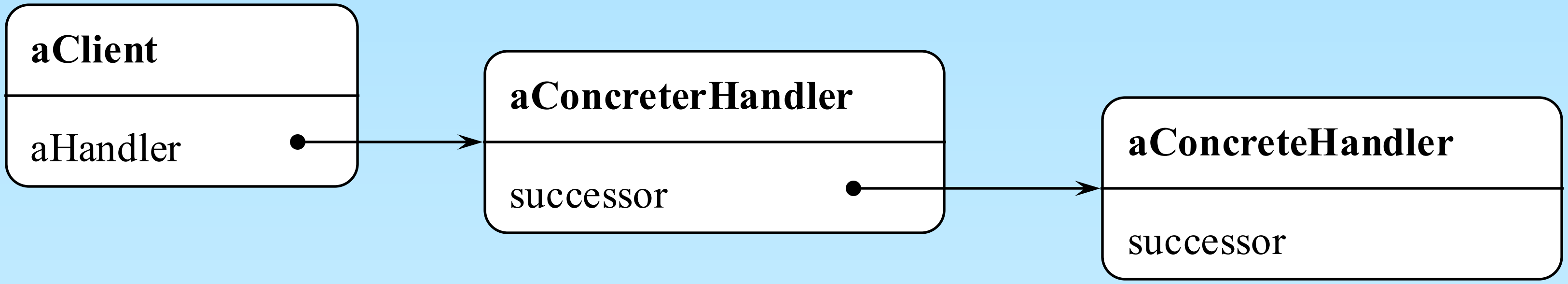
Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Structure of Chain of Responsibility

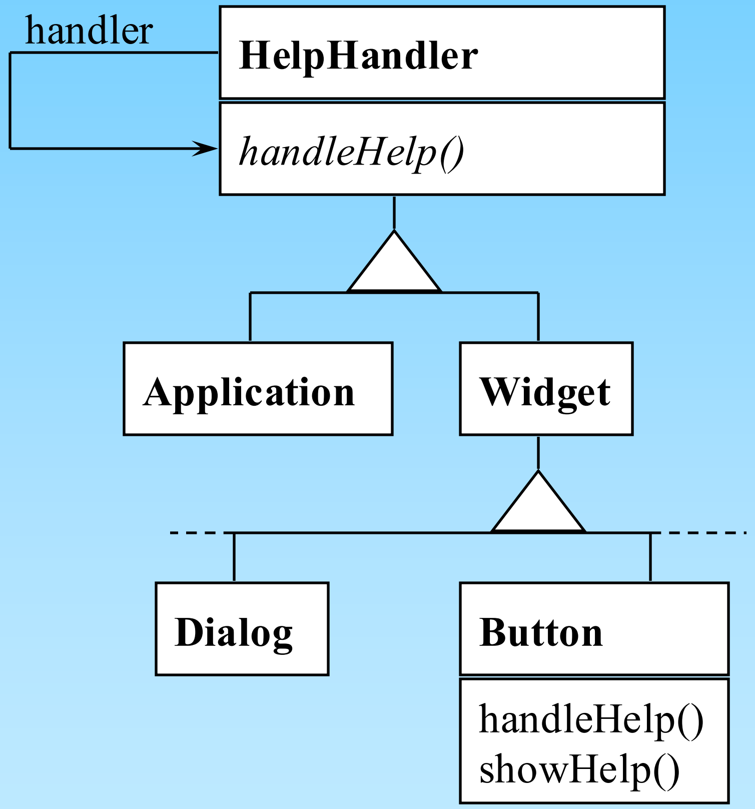


typical object structure:

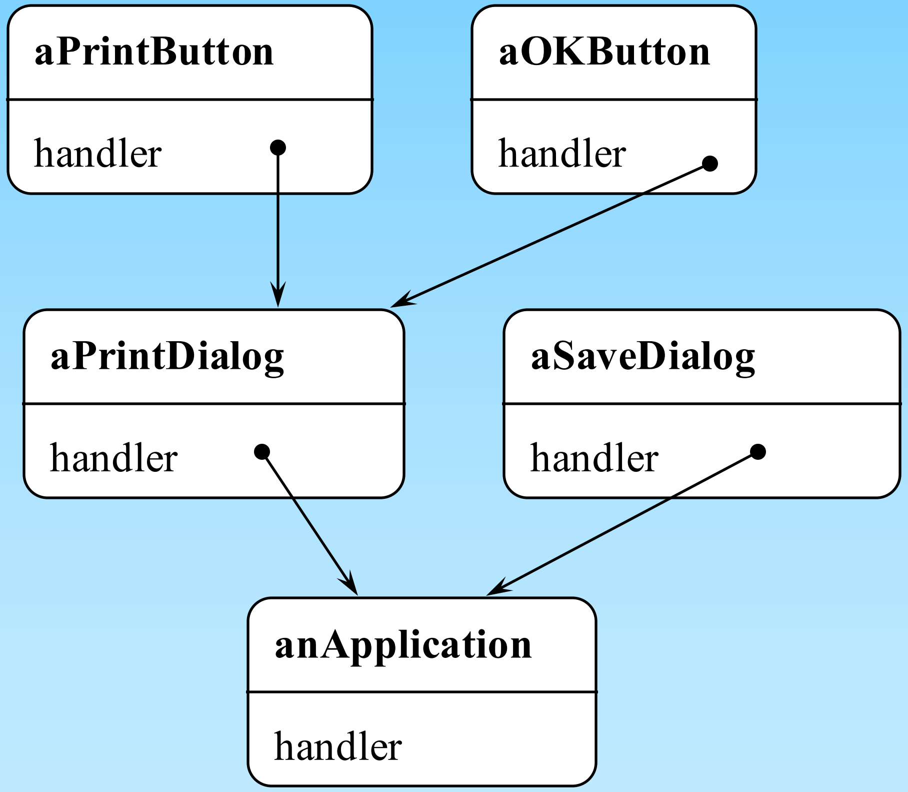


Example of Chain of Responsibility

help handler of a graphical user interface



object structure:



Chain of Responsibility

Applicability

- When more than one object my handle a request, and the handler is not known *a priori*. The handler should be ascertained automatically.
- When a request should be issued to one of several objects without specifying the receiver explicitly.
- When the set of objects that can handle a request should be specified dynamically.

Master/Worker

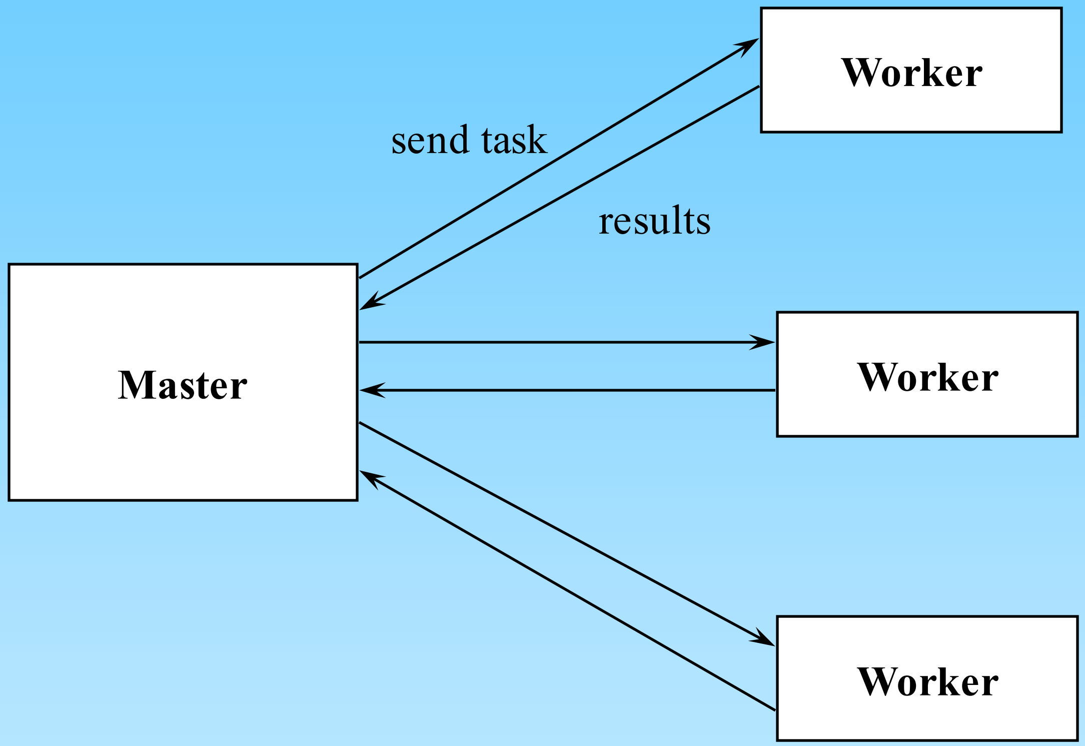
Intent

The Master/Worker design pattern supports fault tolerance, parallel computation, and computational accuracy. A master component distributes work to identical worker components and computes a final result from the results these workers return.

Also Known as

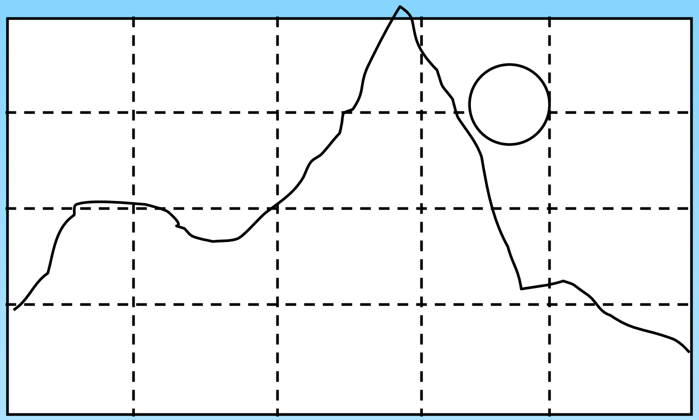
Master/Slave

Structure of Master/Worker



Example of Master/Worker

Parallel Computation of Ray-traced Image



Master hands out rectangular regions of the image to free Workers.

Master/Worker

<u>Applicability</u>

- When there are several tasks that can be completed independently.
- When there are several processors available to work in parallel.
- When the load of the Workers must be balanced.

Process Control

Intent

Regulate a physical (continuous) process.

- Open-Loop System
- Closed-Loop System
  - Feedback Control System
  - Feedforward Control System

Process Control: Terminology

Process Variables: Properties of the process that can be measured; several specific kinds are often distiguished.

Controlled variable: Process variable whose value the system is intended to control

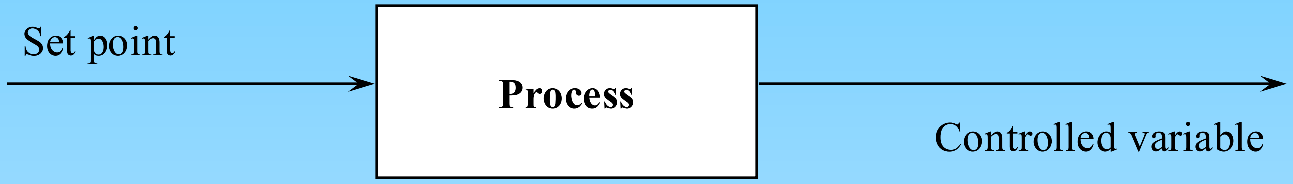
Input variables: Process variable that measures an input to the process

Manipulated variable: Process variable whose value can be changed by the controller

Set point: The desired value for a controlled variable

Open-Loop System

System in which information about process variables is not used to adjust the system.



Example Heating:

- constant burner
- use a timer to turn burner off and on at fixed intervals

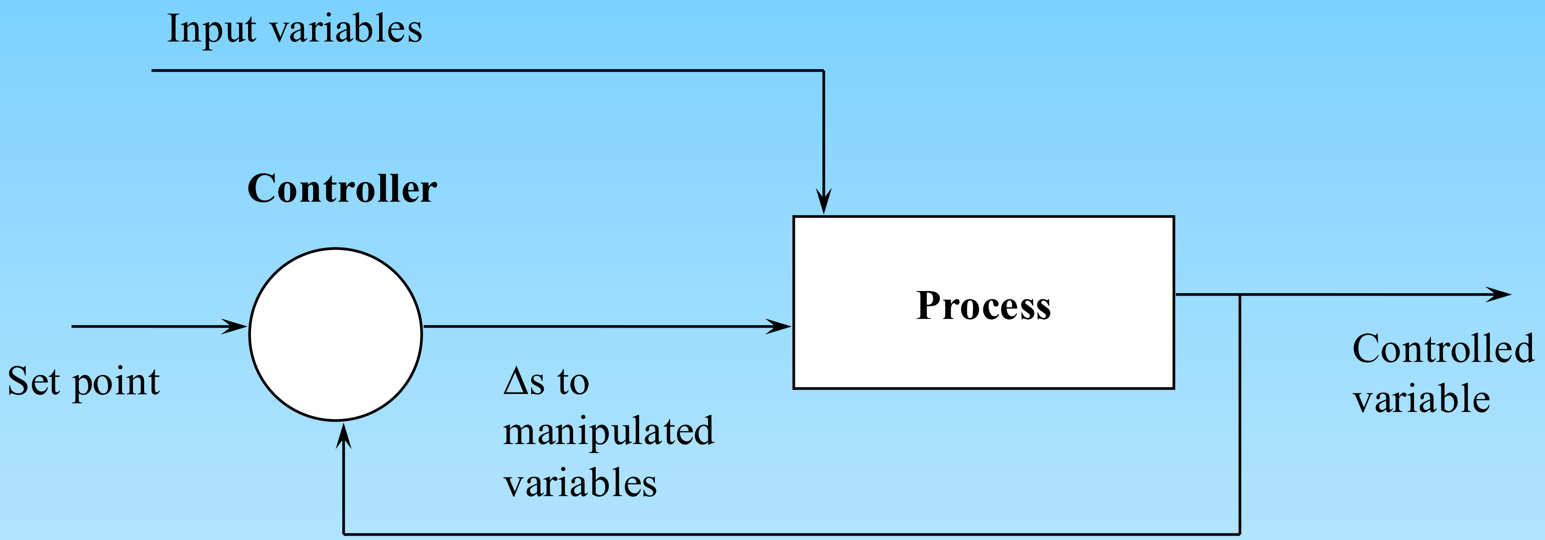
Closed-Loop System

System in which information about process variables is used to manipulate a process variable to compensate for variations in process variables and operating conditions.

- Feedback Control System
- Feedforward Control System

Feedback Control System

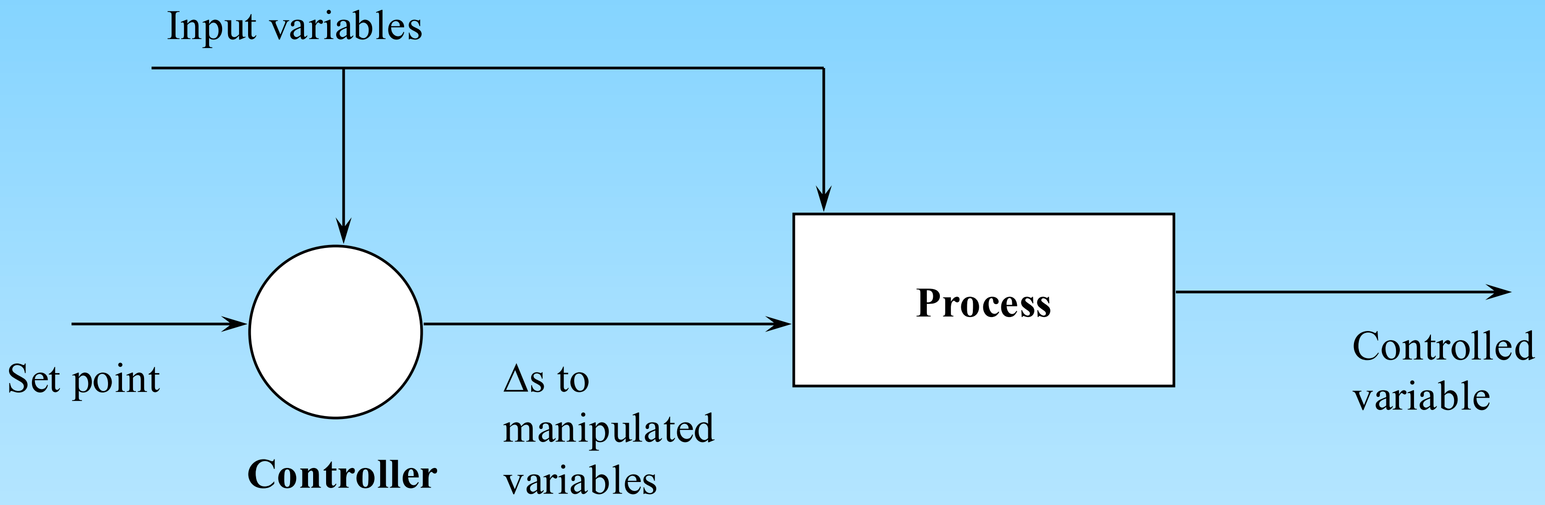
The controlled variable is measured, and the result is used to manipulate one or more of the process variables.



Example Heating:
use a thermostat, burner is turned off and on as necessary to maintain the desired temperature (the set point)

Feedforward Control System

Some of the process variables are measured, and anticipated disturbances are compensated for without waiting for changes in the controlled variable to be visible.



Example Heating:
measure the outside temperature (i.e., before the heating process), select a timer interval dependent on this temperature

5. Virtual Machines

- Interpreter
  - Emulator
- Rule-based Interpreter

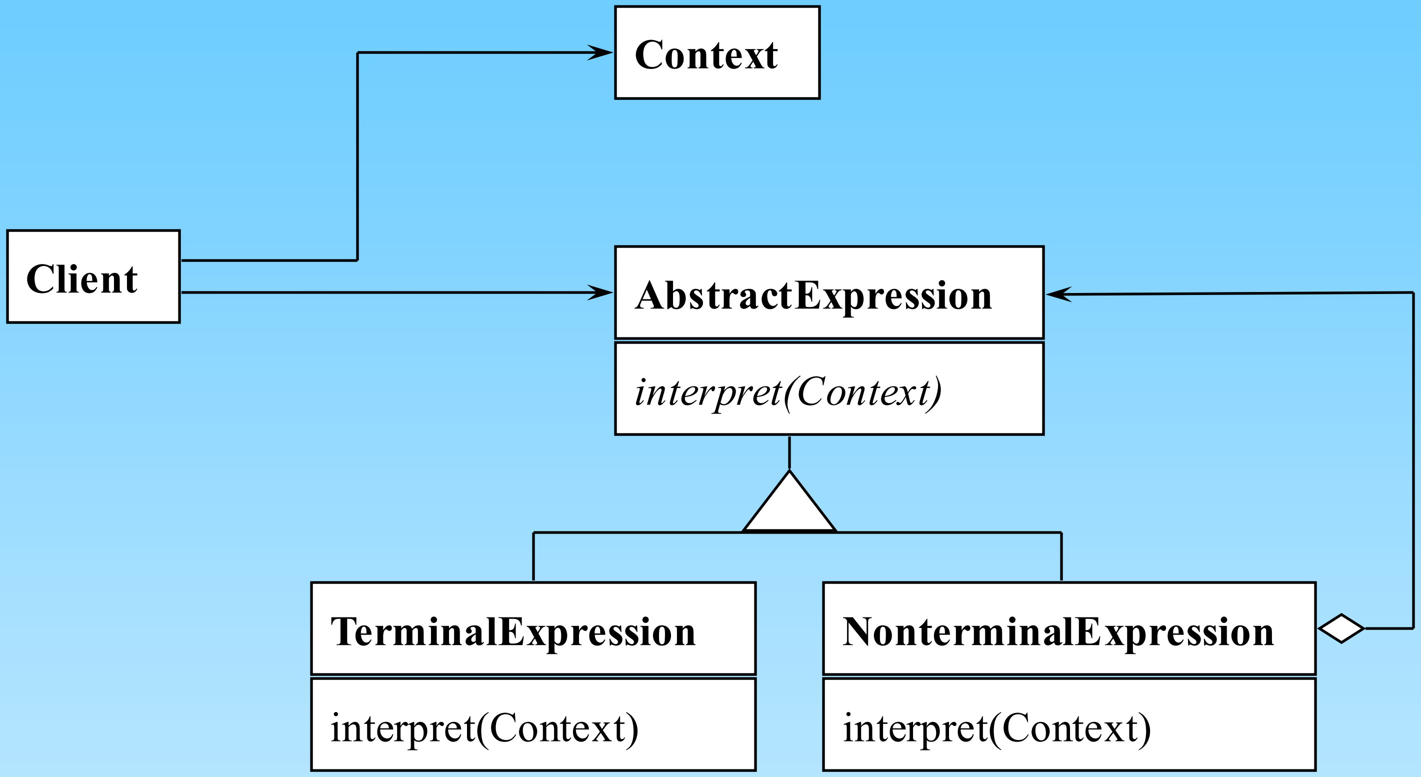
Interpreter

Intent

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

A context provides values for variables or other information needed for interpretation.

Structure of Interpreter



Interpreter

Applicability

When there is a language to interpret, and phrases in the language can be represented in the language as abstract syntax trees. The interpreter pattern works best when

- the grammar is simple. For complex grammars, the class hierarchy for the grammar becomes large and unmanageable. Tools such as parser generators are a better alternative in this case.
- efficiency is not a critical concern. The most efficient interpreters do *not* interpret the input directly but first translate it into a more compact form that is faster to traverse. The translation can also be done by an interpreter.

Interpreter vs. Composite vs. Visitor

- Interpreter and composite share the same structure. One speaks of an interpreter, when sentences in a language are represented and evaluated. In particular, the method *interpret()* usually returns a value that is the result of evaluating and combining all sub-components. One does not normally consider a parts hierarchy as a sentence in a language to be interpreted. Thus, the interpreter can be viewed as a special case of composite.
- Rather than placing the method *interpret()* into every subclass, one can collect them into a visitor.

Rule-Based Interpreter

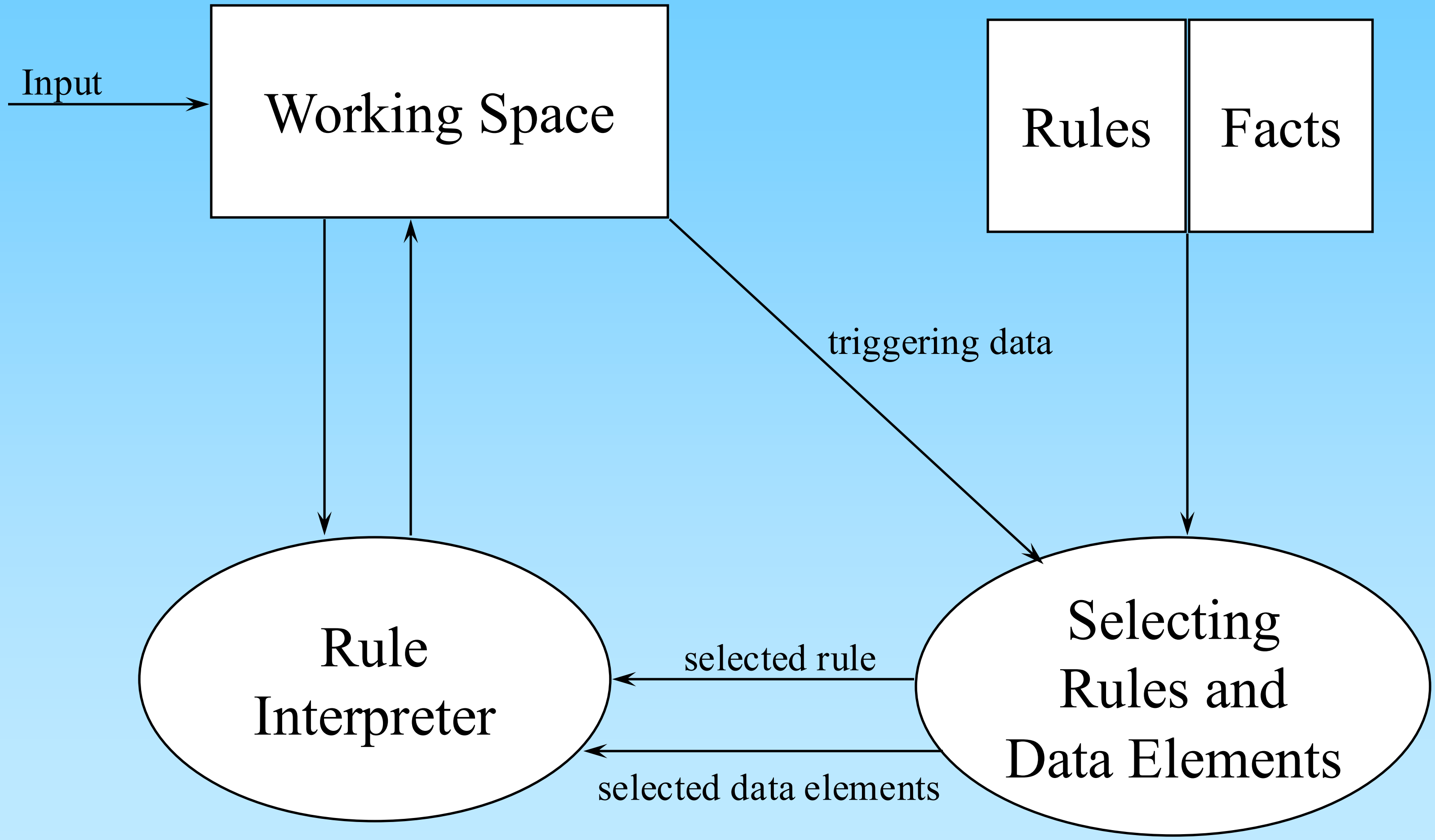
Intent

Solve a Problem using a set of rules. A rule consists of a condition part and an action part. If the condition evaluated on a set of data elements in the working space returns true, then the action part can be executed. The action part replaces or removes data elements which have been selected in the condition part, or adds new data elements.

Also known as

Rule-Based (Expert-) System

Rule-Based Interpreter



Rule-Based Interpreter: Selecting Rules

If more than one rule can be applied, the selection of a rule is done by the the following steps:

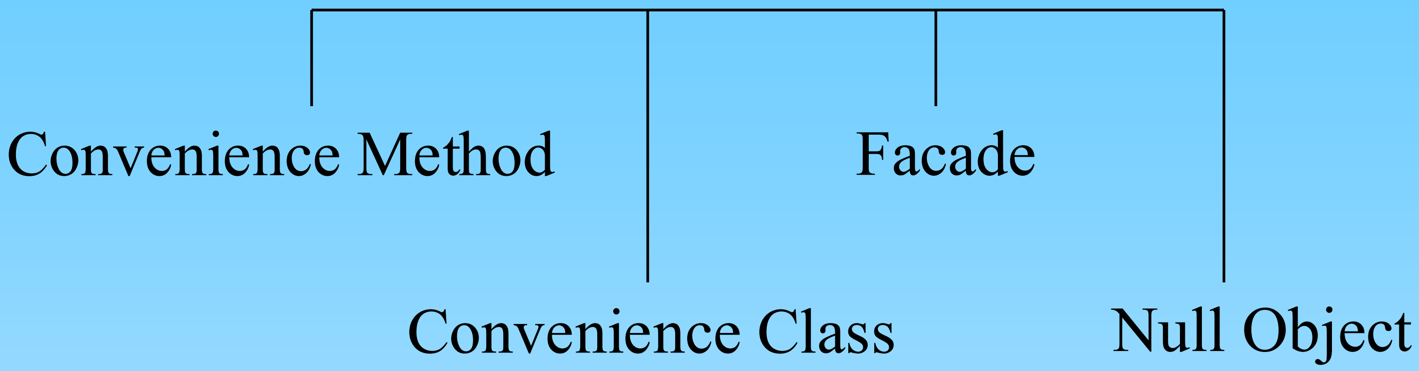
1. A rule may only be applied once to a given data element in the working space.
2. If several rule-data element combinations are applicable, then select those that operate on the youngest data elements.
3. Select the rules with the highest "specificity", i.e., those whose conditions need the most elements of the working space.
4. Select a rule a) in order of specification or b) randomly.

Rule-Based Interpreter

Applicability

- When the solution of the problem is best formulated as a set of condition-action-rules (for example, diagnosis problems, matching problems).
- When the action part only executes simple operations on the data elements (no loops, no recursion, no procedure calls to modify the working space).

6. Convenience Patterns



Convenience Method

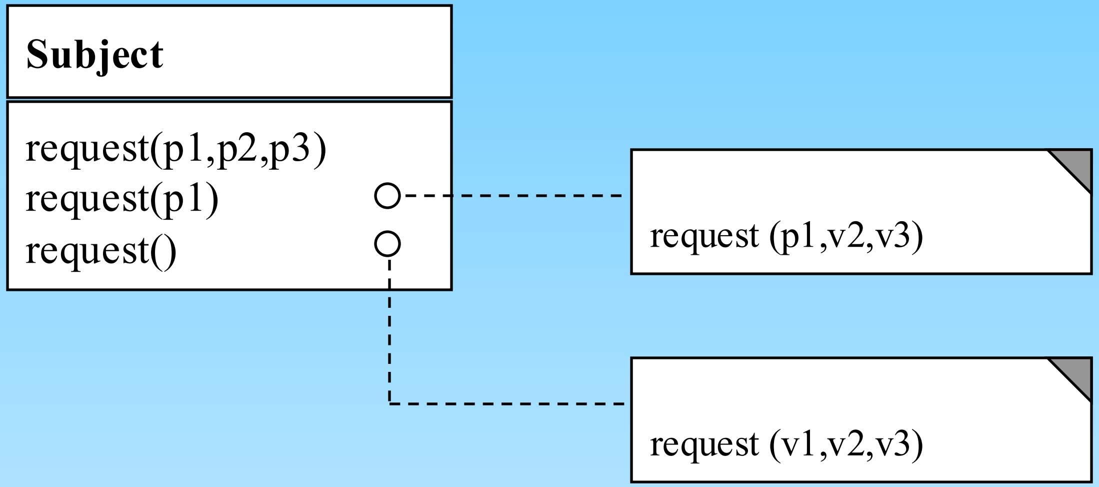
Intent

Simplify method invocation by supplying often-used parameter combinations through additional methods (overloading).

Applicability

When method calls with the same parameters appear in many invocations.

Convenience Method



Convenience methods are not needed in prog. languages with default parameters. The defaults are supplied in the signature, like this:
request(p1 = v1, p2 = v2, p3 = v3) {…}
v1, v2, and v3 are the default parameters. Calls like
request() or request(x) or request(p3 = y) will be completed with the defaults for the missing parameters.

Convenience Class

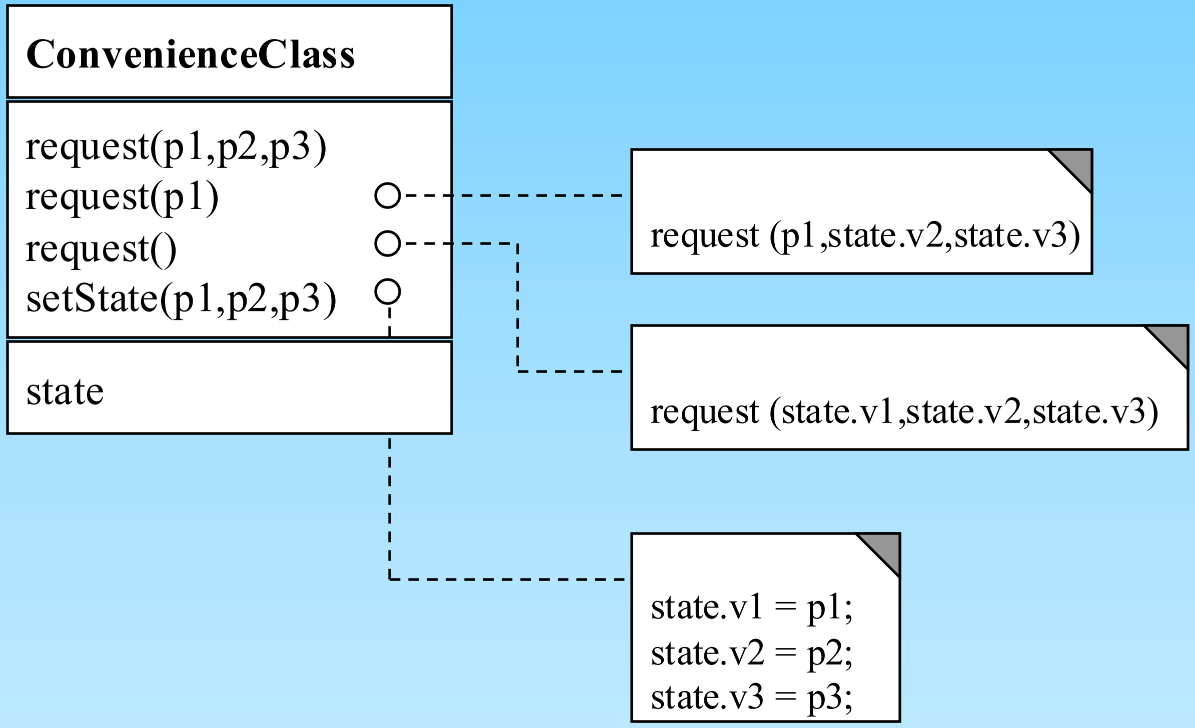
Intent

Simplify method invocation by maintaining parameter values in a special class.

Applicability

When methods with parameters are shared among many invocations but change occasionally.

Convenience Class

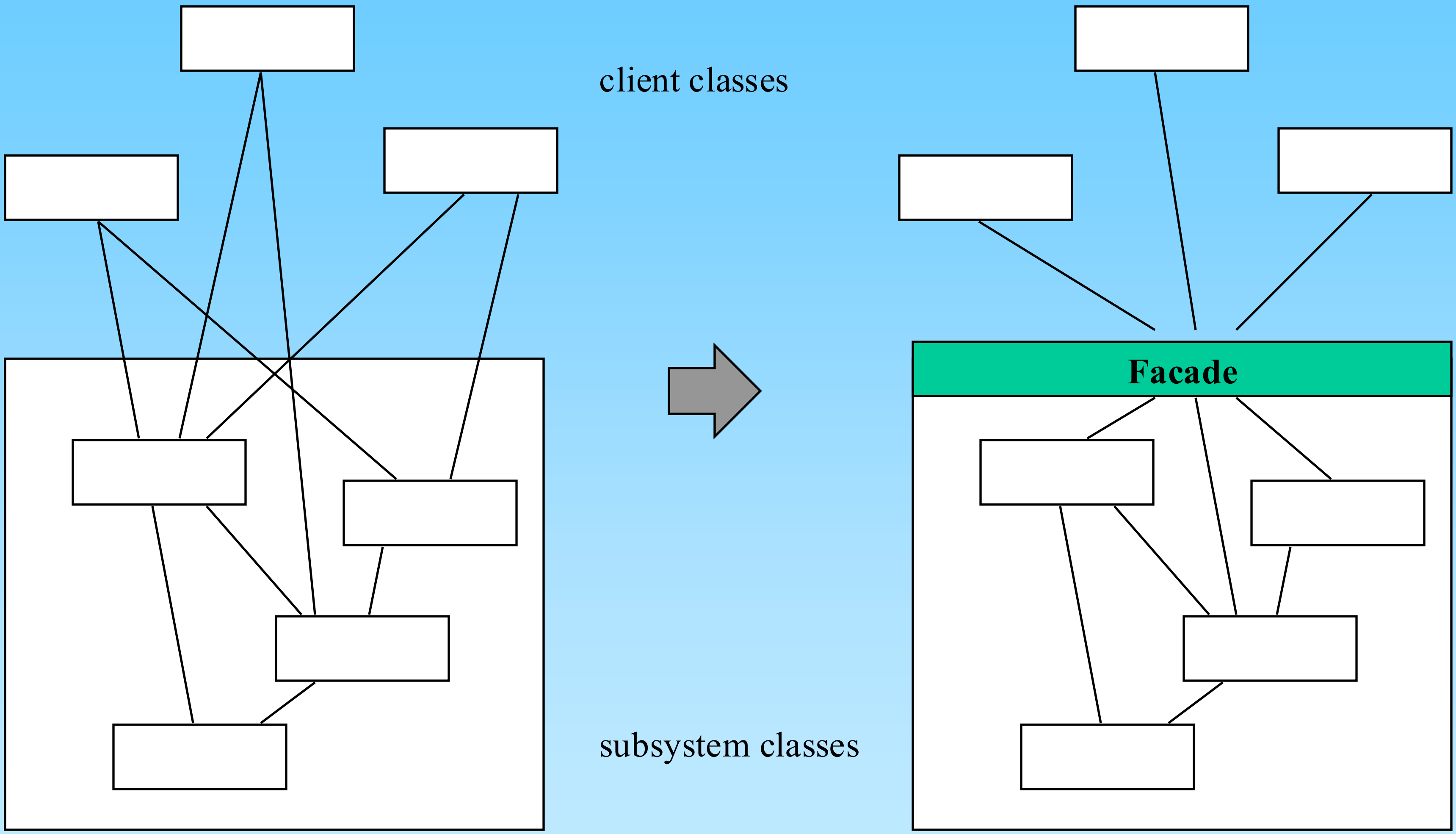


Facade

Intent

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Facade



Facade

Applicability

- When you want to provide a simple interface to a complex subsystem. A facade can provide a simple default view of the subsystem that is good enough for most clients.
- When there are many dependencies between clients and the implementation classes of an abstraction. The introduction of a facade decouples the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
- When you want to layer your subsystems. The facade is used to define an entry point to each subsystem level.

Null Object

Intent

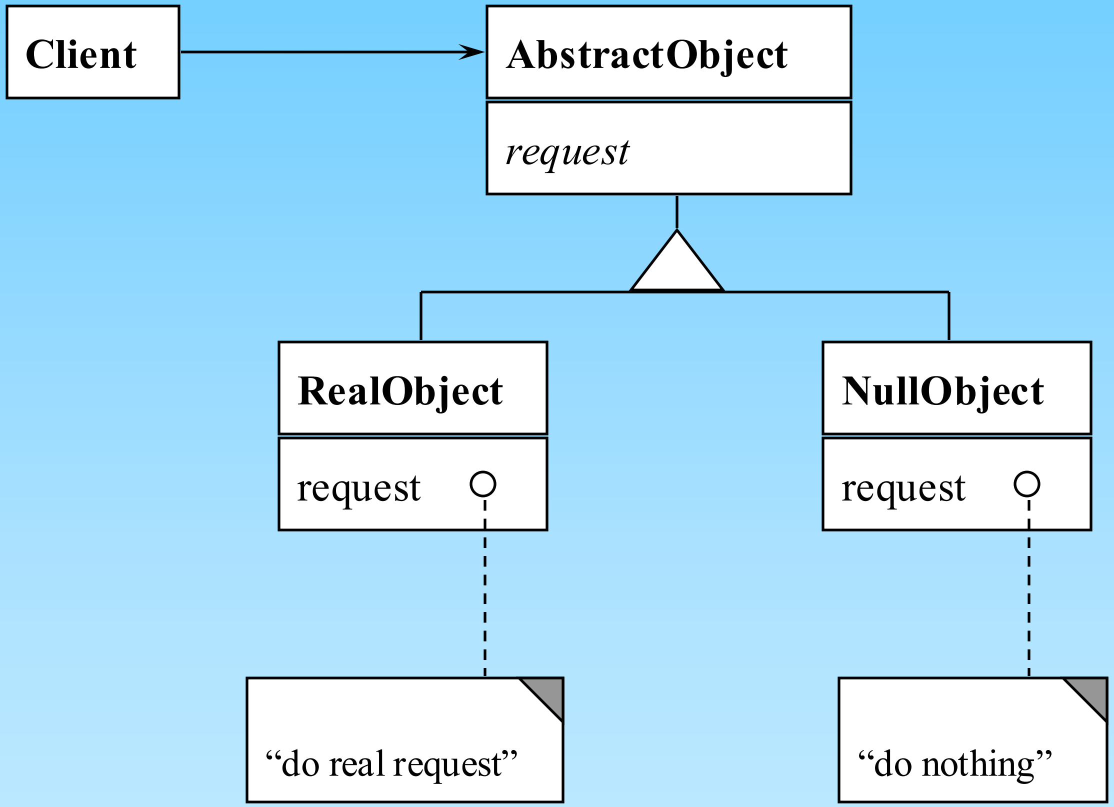
Provide a surrogate for another object that shares the same interface but does nothing. The Null Object encapsulates the implementation decisions of how to “do nothing” and hides those details from its collaborators.

Motivation

Avoid cluttering up code with tests for null values like

```
if (thisCall.callingParty != null)
  thisCall.callingParty.action()
```

Structure of Null Object



Example

```
class NullAction implements … {
    public void action (){/* do nothing*/}
}

thisCall.callingParty = new NullAction();

…
// somewhere else
~~if (thisCall.callingParty != null)~~
thisCall.callingParty.action()
```

Null Object

Applicability

- When an object requires collaborators and one or more of them should do nothing.
- When clients should be able to ignore the difference between a collaborator which provides real behavior and that which does nothing.
- When the do-nothing behavior should be reused so that various clients which need this behavior will consistently work the same way.

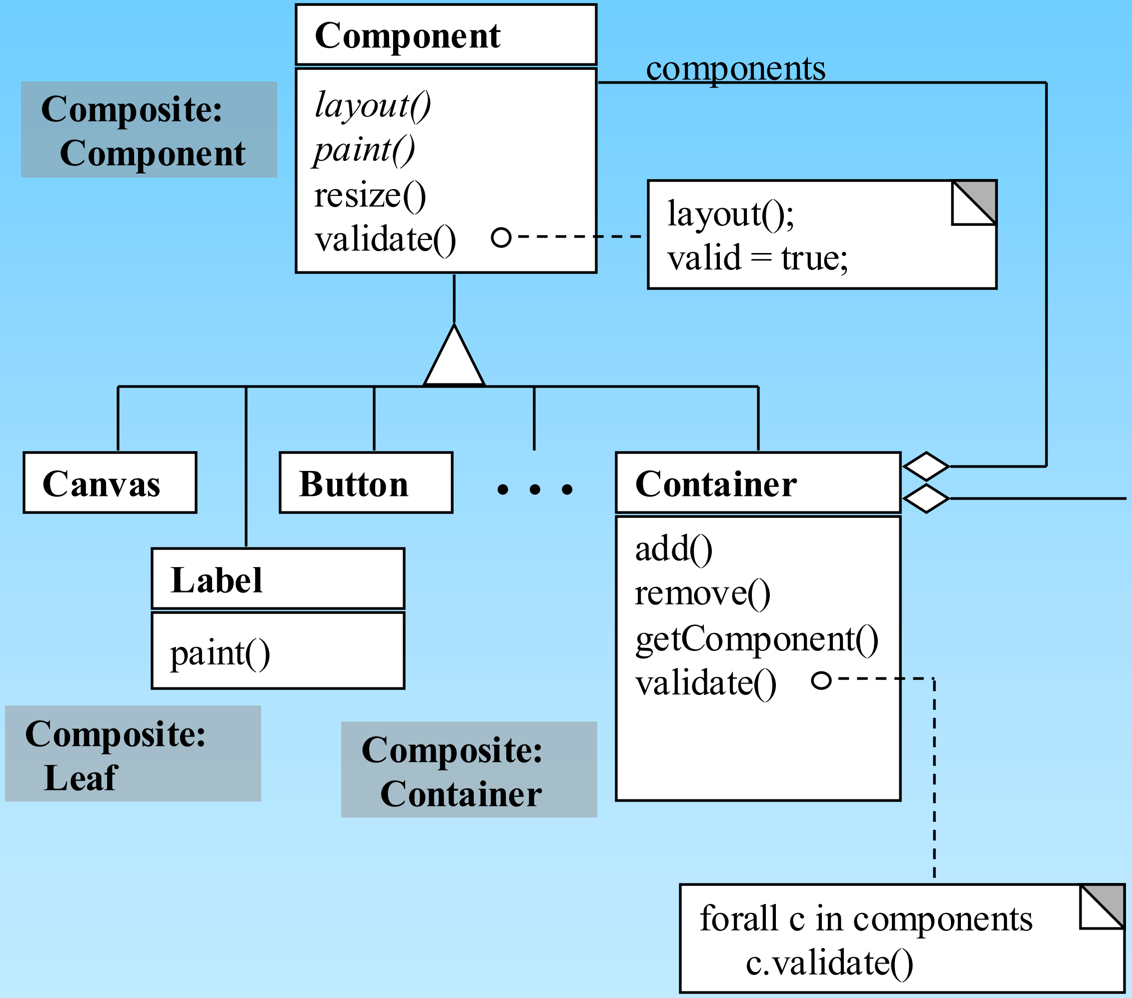
Design Patterns within AWT 1.0

- AWT (Abstract Window Toolkit) is a class library that
  - offers a uniform GUI builder
  - for several operating systems and GUI styles.
- AWT is implemented in Java and thus portable.

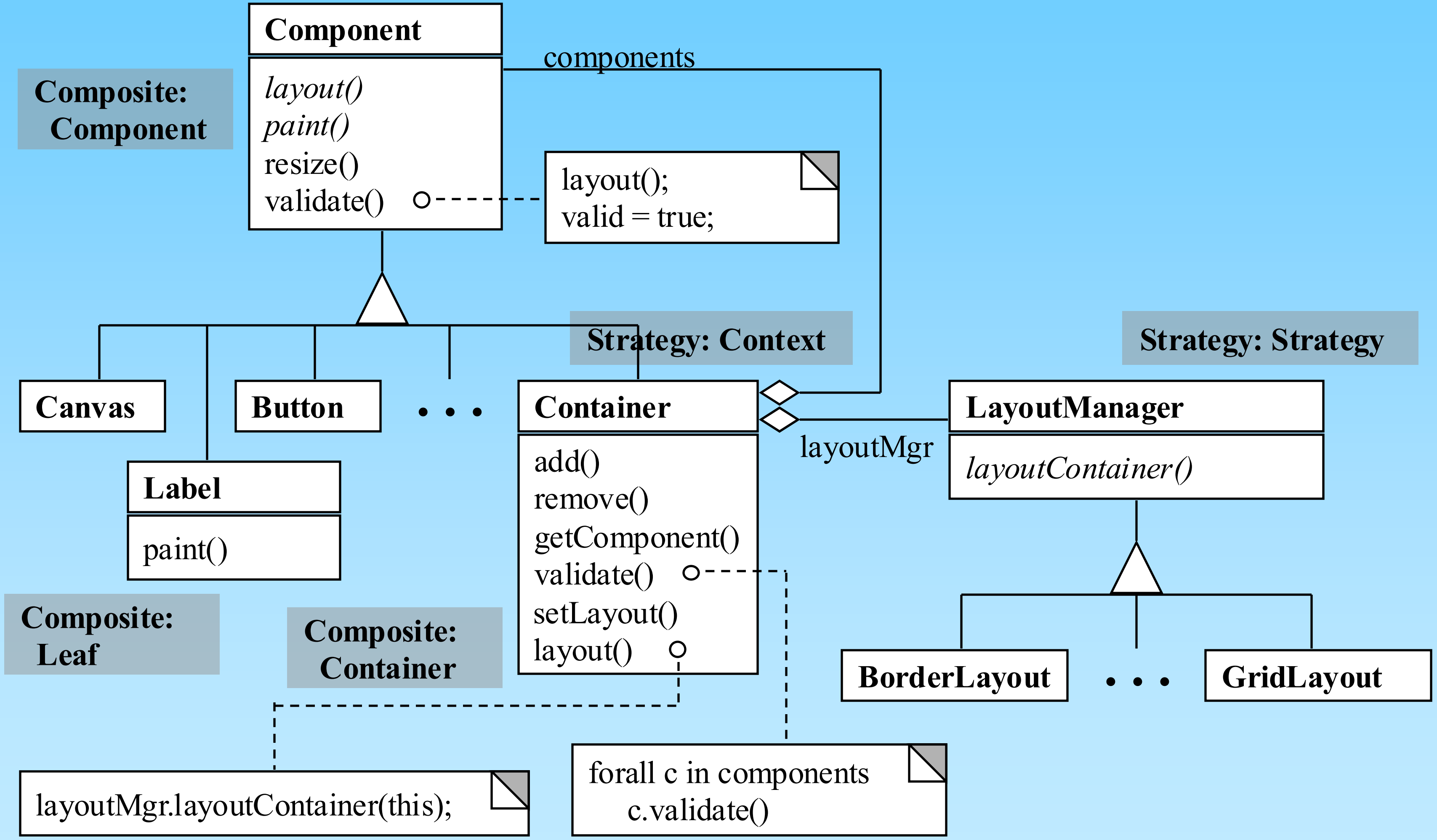
Composite, Strategy, Observer

- Composite
  - There are widgets that contain other widgets.
  - Composites can be treated like single elements.
- Strategy
  - The (generic) composites such as *window* and *panel* can be used in different layout contexts.
  - The programmer can choose a layout algorithm.
- Observers
  - Listeners (= observers) react to events (mouse or key clicks, etc.); several may be registered.

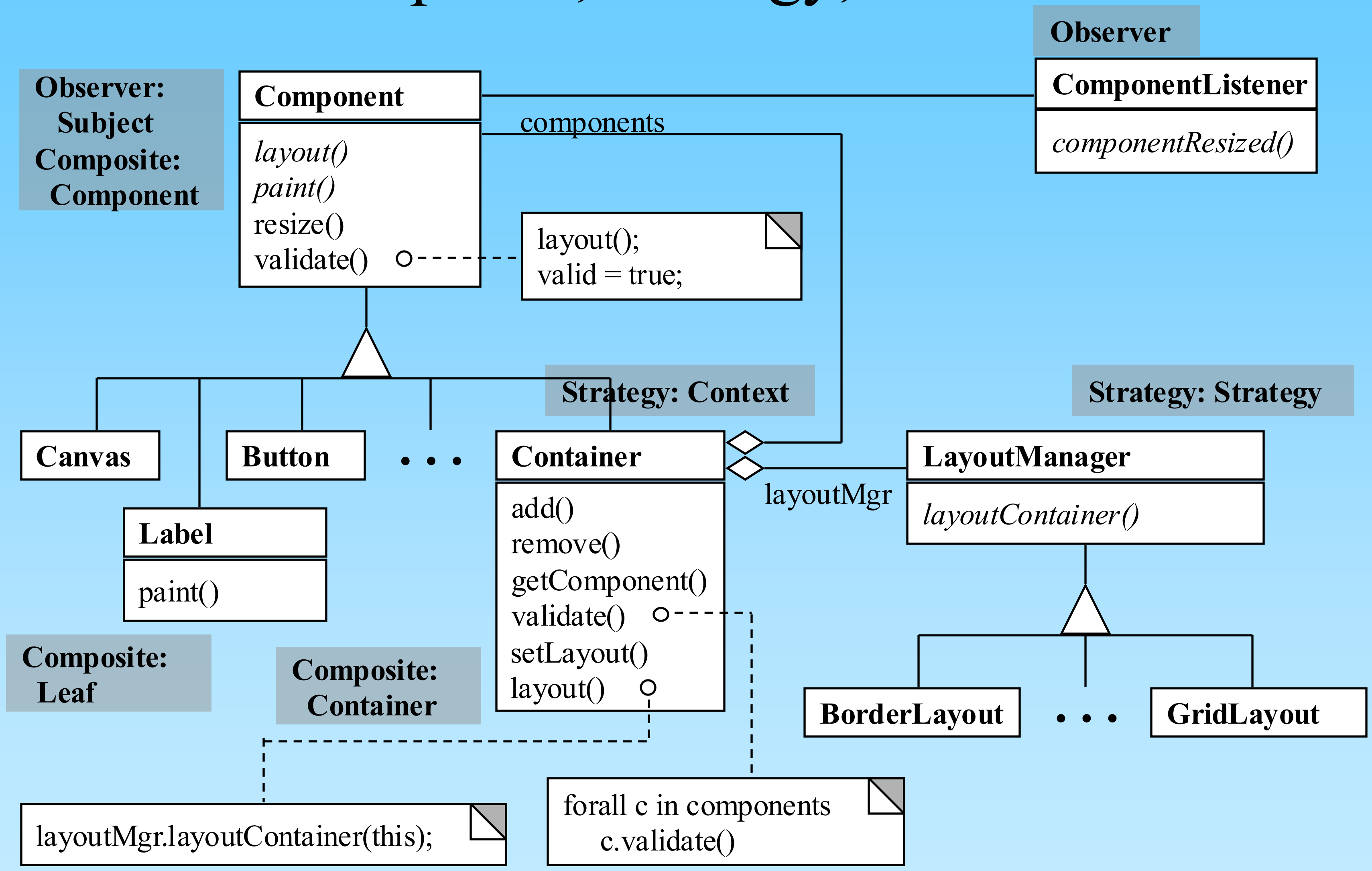
Composite


Component
layout()
paint()
resize()
validate()
components
layout();
valid = true;
Composite: Component
Canvas
Button
. . .
Container
add()
remove()
getComponent()
validate()
Label
paint()
Composite: Leaf
Composite: Container
forall c in components
c.validate()

Composite and Strategy



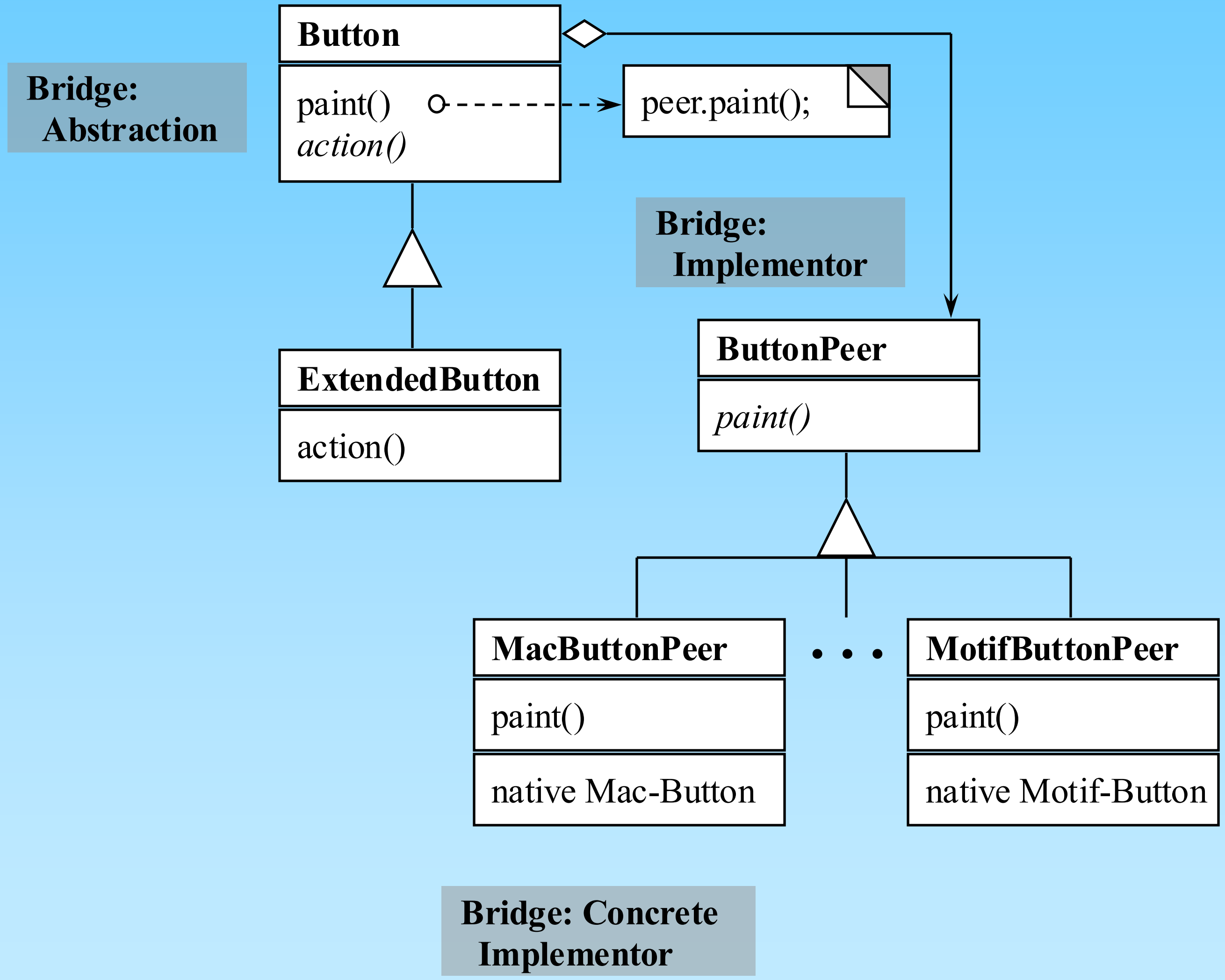
Composite, Strategy, Observer



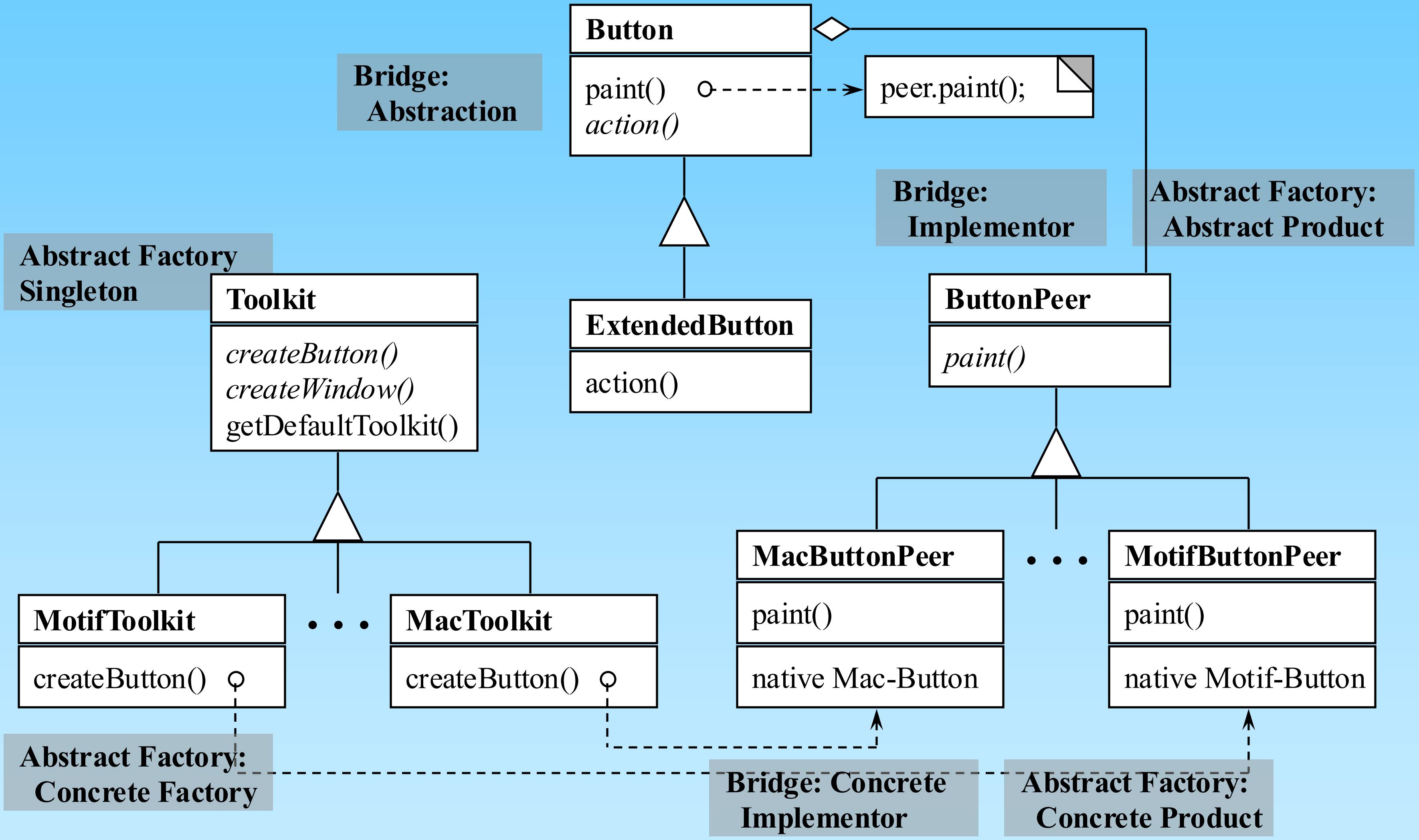
Bridge, Singleton, and Abstract Factory

- Bridge
  - Widgets like *button* abstract from concrete shape and implement only behavior.
  - Drawing etc. is done by a GUI-depended implementation.
- Abstract Factory
  - Widgets can be created platform-independently.
- Singleton
  - guarantees that only one instance of the *toolkit* class can be created.

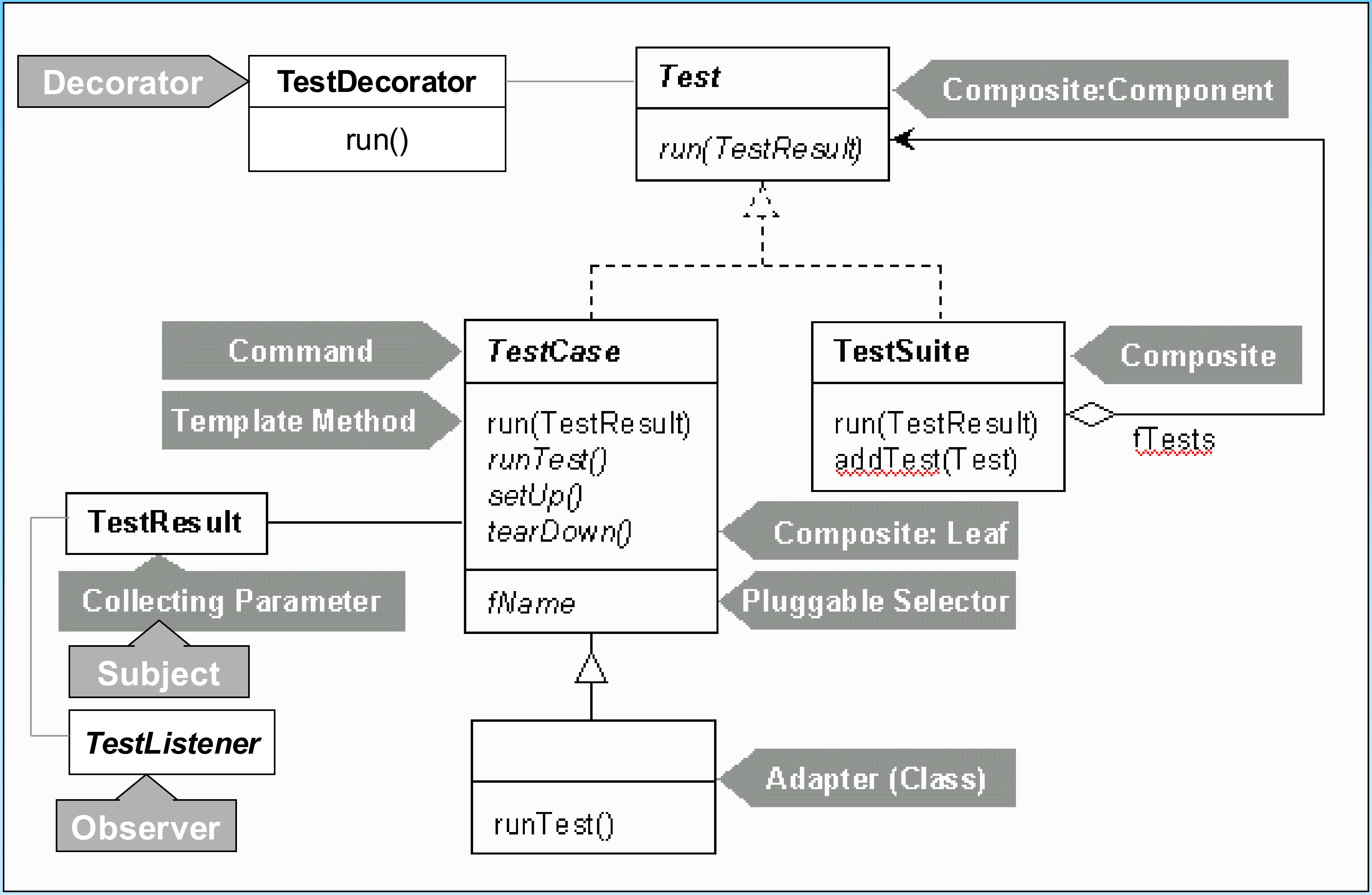
Bridge


Button
paint()
action()
peer.paint();
Bridge: Abstraction
Bridge: Implementor
ExtendedButton
action()
ButtonPeer
paint()
MacButtonPeer
paint()
native Mac-Button
. . .
MotifButtonPeer
paint()
native Motif-Button
Bridge: Concrete Implementor

Bridge, Singleton, and Abstract Factory



Architecture of JUnit, a Tool for Regression Testing (superseded).



Summary

Design Patterns

- record proven solutions to design problems,
- provide vocabulary for efficient communication among designers and implementers,
- encapsulate best design practices,
- document designs,
- improve design quality and designer productivity,
- can be organized according to their purpose.