Appendix B

Design Patterns Summaries

Abstract Factory

- Intent

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

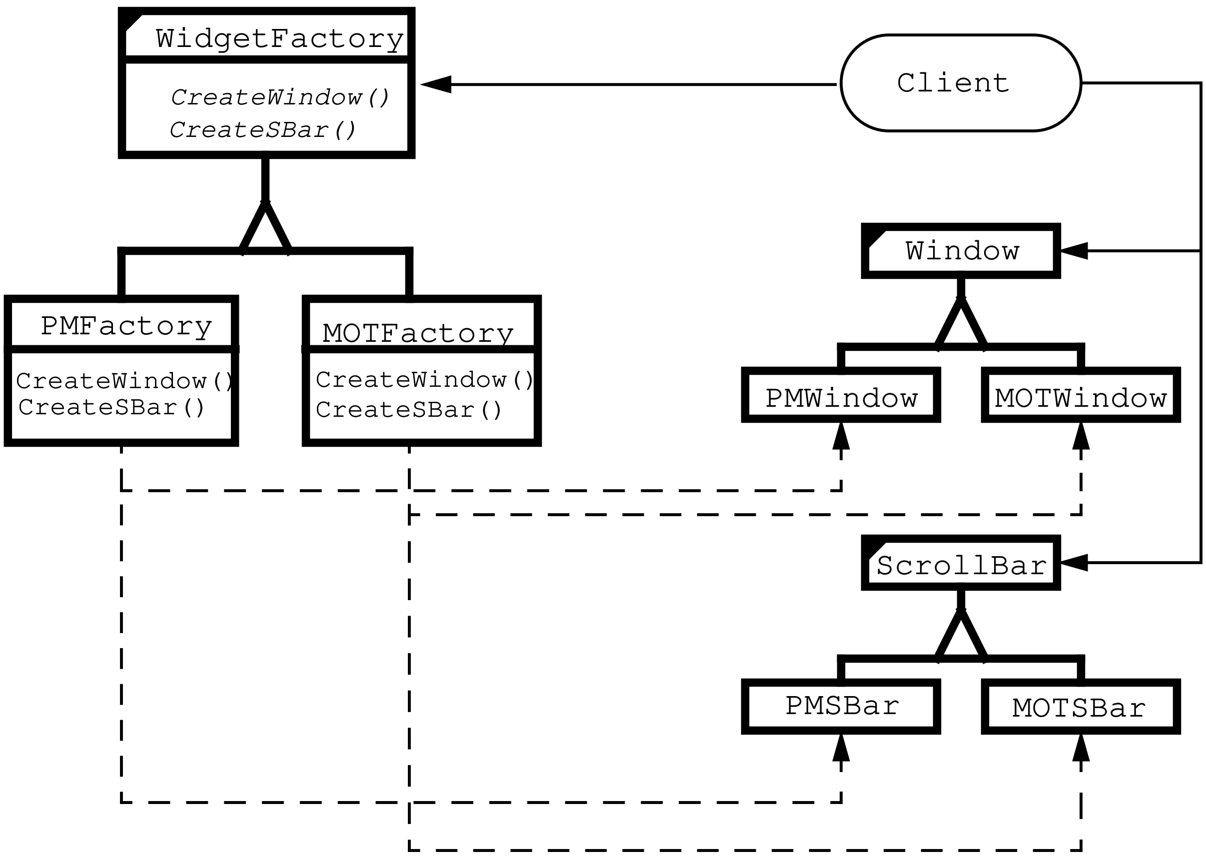
- Motivation

  Consider a user interface toolkit that supports multiple look-and-feel standards such as Motif and Presentation Manager. To be portable across look-and-feel standards, an application should not hard-code its widgets (windows, scroll bars,...) for a particular look and feel.

  We can solve this problem by defining an abstract WidgetFactory class that declares the interfaces for creating each kind of widget. We also use abstract classes for each kind of widget with concrete subclasses to actually implement the various look-and-feel standards

Abstract Factory

- Specific implementation

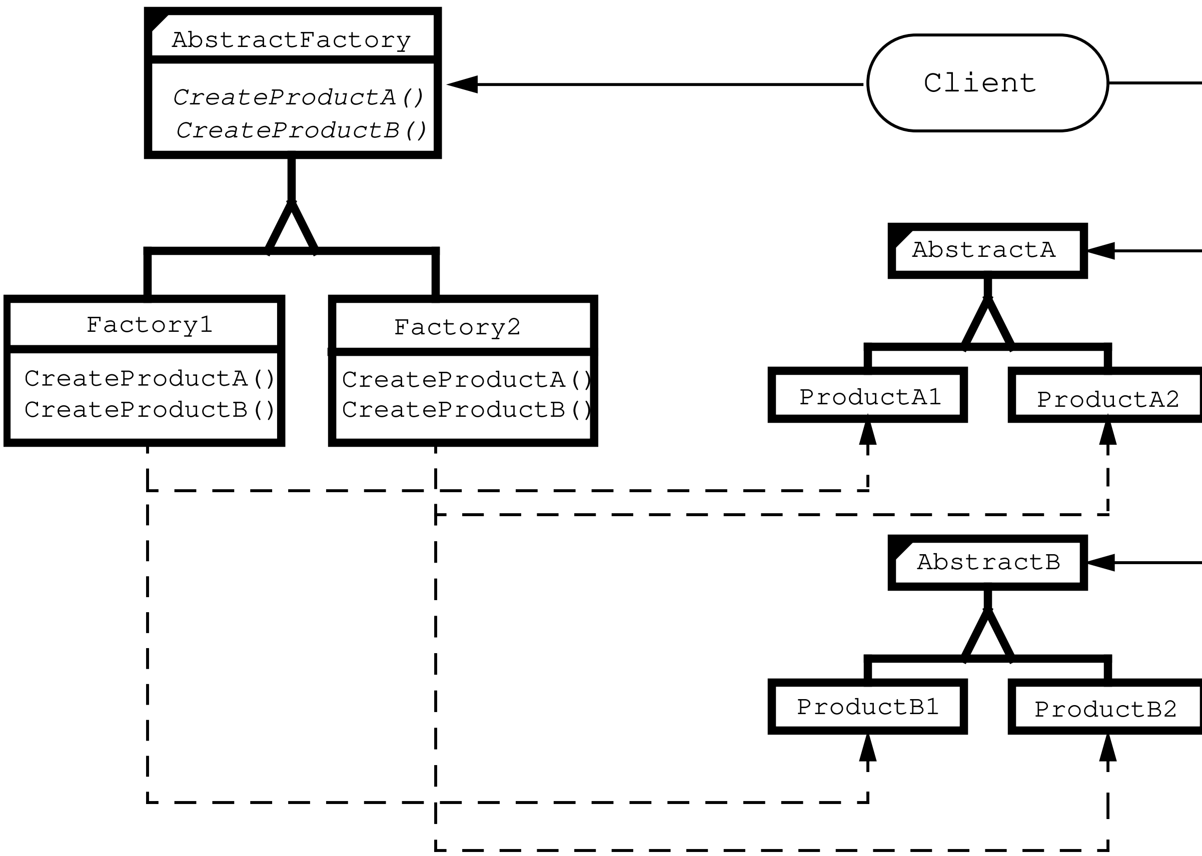


Abstract Factory

- Applicability
  - When a system should be independent of how its products are created, composed, and represented
  - When a system should be configured as one of a multiple set of possibilities
  - When a family of related classes should be used together and you need to enforce this constraint
  - When you want to reveal interfaces to, but not the implementations of, classes

Abstract Factory

- General structure



Abstract Factory

- C++ Implementation

```
class AbstractFactory {
public:
  virtual CreateProductA()=0;
  virtual CreateProductB()=0;
};

class Factory1: public AbstractFactory {
public:
  CreateProductA() {…}
  CreateProductB() {…}
};
class Factory2: public AbstractFactory {
public:
  CreateProductA() {…}
  CreateProductB() {…}
};

class AbstractA{};

class ProductA1: public AbstractA{};

class ProductA2: public AbstractA{};


class AbstractB{};

class ProductB1: public AbstractB{};

class ProductB2: public AbstractB{};
```

Abstract Factory

- Consequences

  + The Abstract Factory pattern isolates client classes from the implementation classes by creating and using abstract interface classes

  + This pattern makes changing products easy because the concrete factory name appears only once, where it is instantiated

  + It guarantees consistency among the various product classes

  - Supporting new products may require extending the Abstract Factory interface which would require a retrofit of all the ConcreteFactory classes

Composite

- Intent

  Compose objects into tree structures to represent whole-part hierarchies. Composite lets client treat individual objects and compositions of objects uniformly

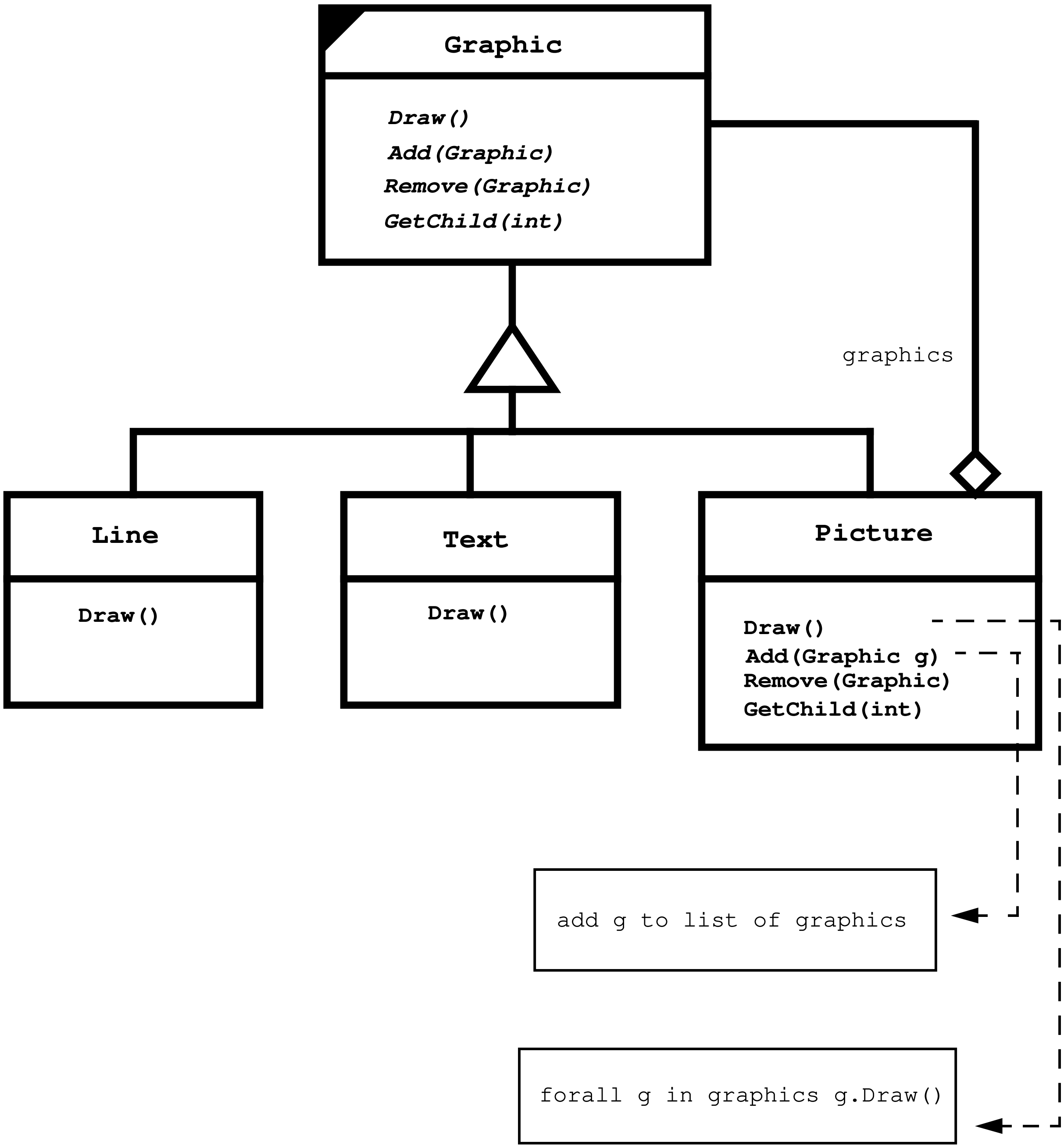
- Motivation

  Graphics applications like drawing editors let users build complex diagrams out of simple components. The user can group components to form larger components, which can be grouped to form larger components.

  A simple implementation defines classes for primitives such as Text and Line plus other classes that act as containers for these primitives. There is a problem with this approach: Code that references these classes must treat primitive and container objects differently even though users would like to treat them the same. Having to distinguish between these different kinds of classes makes applications more difficult. The Composite pattern uses recursive composition to simplify processing

Composite

- Specific implementation

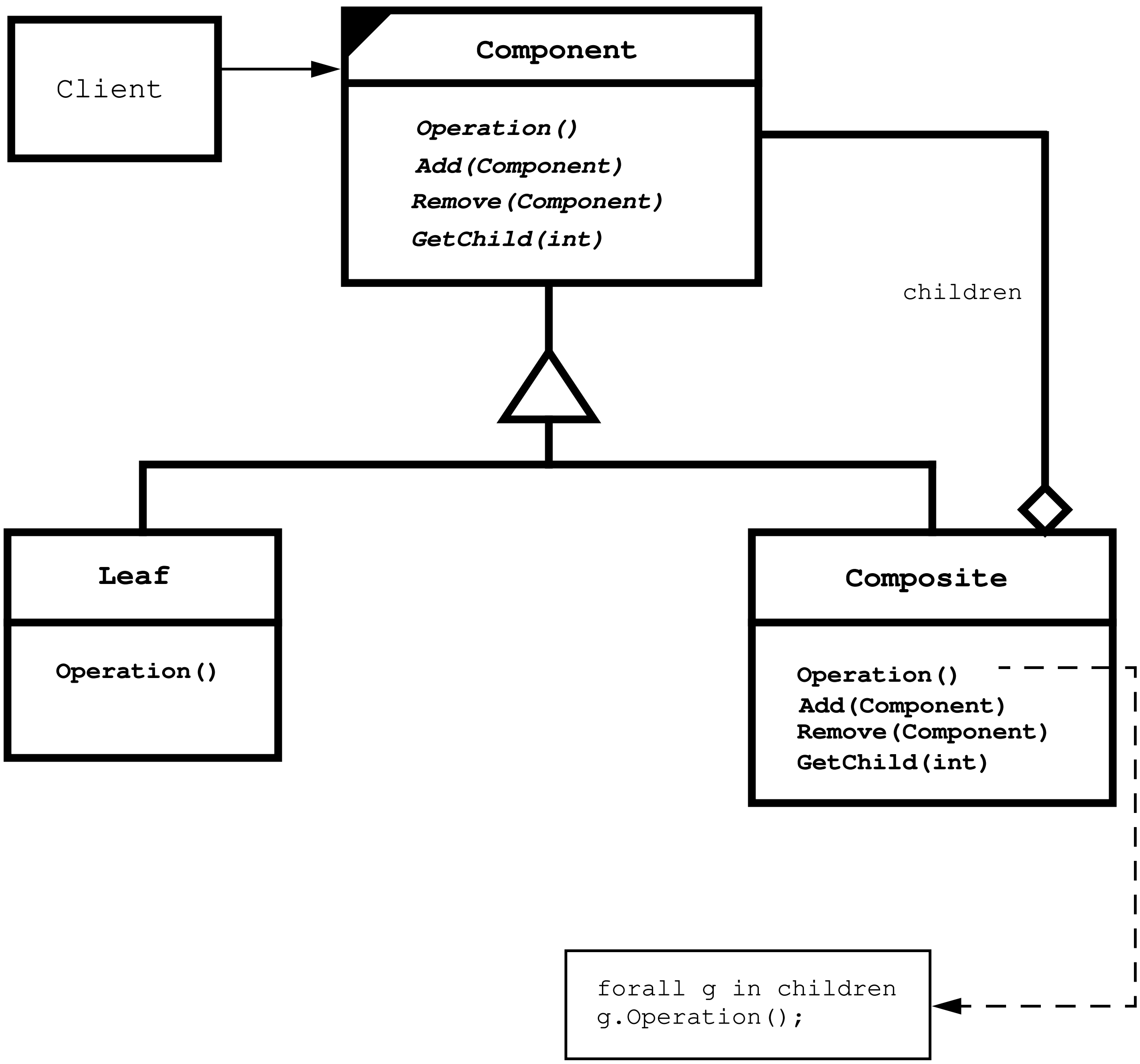


Composite

- Applicability
  - You want to represent part-whole hierarchies of objects and you want clients to be able to ignore the differences between the individual objects and the containers of these objects

Composite

- General structure



Composite

- C++ Implementation

```
class Component{
public:
  virtual Operation()=0;
  virtual Add()=0;
};

class Leaf: public Component{
public:
  Operation{…}
  Add(){???}
};

class Composite: public Component{
public:
  Operation(){"foreach child in children" child.Opera-
tion()}
  Add(){…}
private:
  List<Component> children;
};
```

Composite

- Consequences

  + Allows primitive objects to be composed into more complex objects which can be composed into more complex objects

  + Makes client programs simple. Clients can treat composite structures and their components uniformly. Clients don't know (and shouldn't care) whether they are dealing with leaf or composite components

  + Makes it easier to add new kinds of components. Newly defined Composite or Leaf subclasses work automatically with existing structures and client code

  - Can make design too general. Sometimes you want a composite to have only certain components. With the Composite pattern you can't rely on the type system to enforce those constraints. You will have to do run-time type checking to enforce the constraints

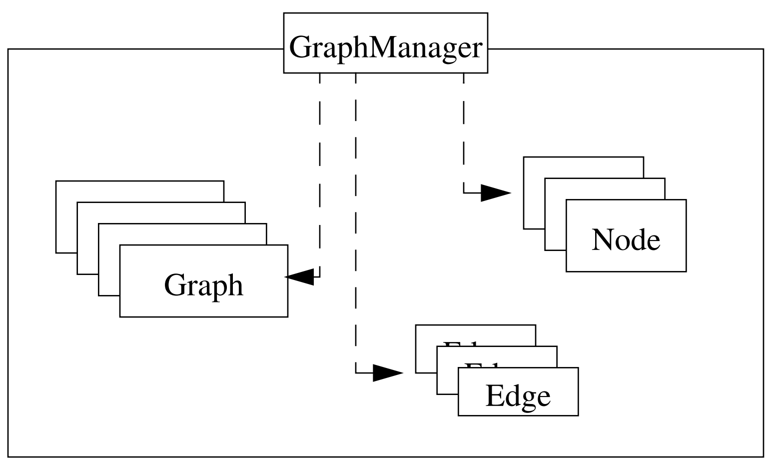
Facade

- ## Intent

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

- ## Motivation

  Decomposing any entity into small pieces reduces its structural complexity, but it can increase its psychological complexity. That is, the single entity with a single well-defined interface is easier to understand but can have a very complex interna structure. By decomposing into smaller pieces, the internal structure is simplified; however, there are now many interfaces to understand and coordinate. The **facade** provides a single simple interface to themore general services of the pieces.



Facade

- Applicability

Use the Facade when you want to provide a simple interface to a complex subsystem.

Use a Facade when there is much interaction between the clients and the classes that form the subsystem encapsulated by the facade.

Use a Facade to layer the system.

- Participants

Facade - delegates client messages to the appropriate subsystem class.

Subsystem classes - these implement the subsystem behavior.

- Consequences

Benefits

Reduces complexity

Reduces coupling

Risks

Delegation introduces an increased number of messages that may not be able to be inlined.

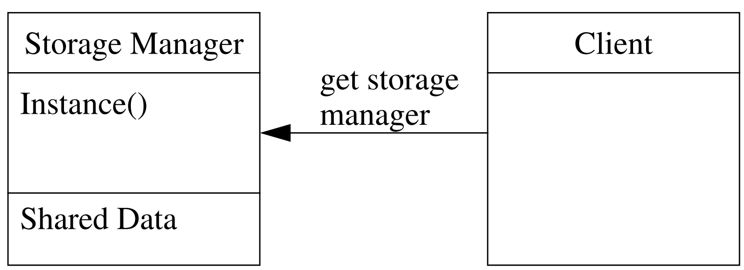
Singleton

- Intent

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

- Motivation

  It is sometimes critical that there only be one instance of a class. Certain managers, such as storage managers, that coordinate access to shared resources must be the exclusive agent in the system.



Singleton

- Applicability

  Use the Singleton pattern when there must be exactly one instance of a class, and it must be accessible from a single point of access.

  Use the Singleton pattern when the instance must be extensible by subclassing.

- Participants

  The Singleton class that provides a method that returns the single instance.

  The Singleton class produces the instance.

- Consequences

  Benefits

  Guarantee of single instance or any specific number of instances

  Avoids global variables

  Risks

  An additional small class that must be managed

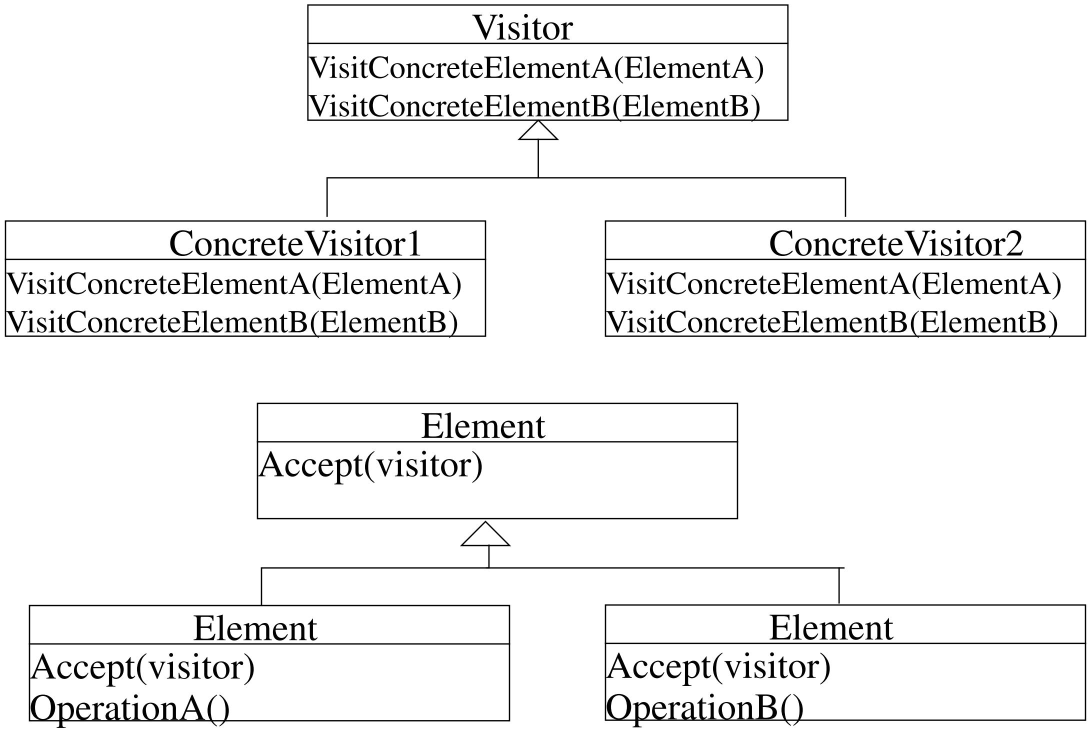
Visitor

- **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.
- **Motivation**

  Often a single family of operations, such as collisions, must be spread over a number of classes, such as the ArcadeGamePiece classes. Adding a new operation to the family requires recompiling all of the classes.
  A separate object, called a **visitor**, can be defined that encodes a single operation for all of the different types of classes it will visit. In this way, a new operation can be added by defining a single additional class.
- **Applicability**

  Use Visitor when
  - an object structure contains many classes of objects with differing interfaces, the Arcade-GamePieces and you want to perform operations on these objects that depend on their concrete classes.
  - the classes defining the object structure rarely change, but you want ot be able to define new operations over the structure.
- Structure

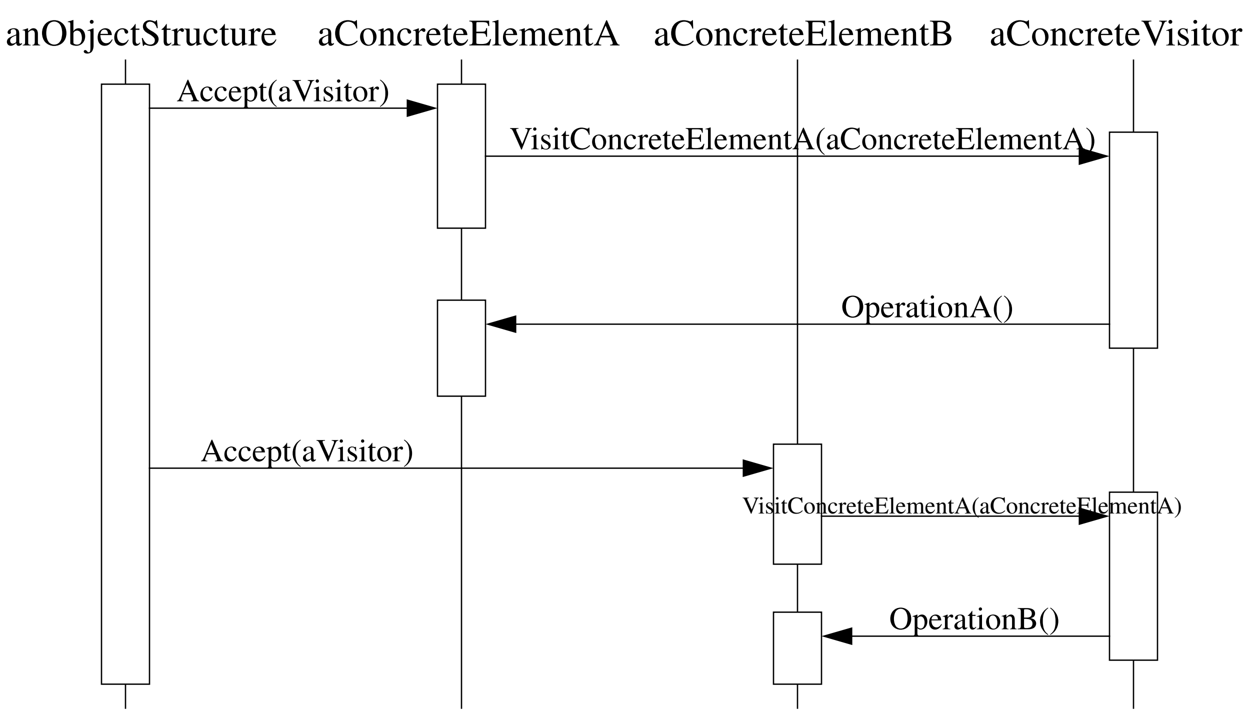

Visitor
VisitConcreteElementA(ElementA)
VisitConcreteElementB(ElementB)
ConcreteVisitor1
VisitConcreteElementA(ElementA)
VisitConcreteElementB(ElementB)
ConcreteVisitor2
VisitConcreteElementA(ElementA)
VisitConcreteElementB(ElementB)
Element
Accept(visitor)
Element
Accept(visitor)
OperationA()
Element
Accept(visitor)
OperationB()

Visitor

- **Participants**

  Visitor - defines a set of operations on all of the classes to be visited, the signature of each operation identifies the class that sends the Visit request to the visitor.
  Concrete Visitor - implements each operation defined by Visitor. ConcreteVisitor stores the local state for the algorithm represented by the methods in the class.
  Element - defines the top level Accept operation that takes a visitor as an argument.
  ConcreteElement - implements the Accept operation defined at the Element level.
  ObjectStructure - can enumerate its elements. Provides a means of visiting individual elements.

- **Collaborations**



- **Consequences**

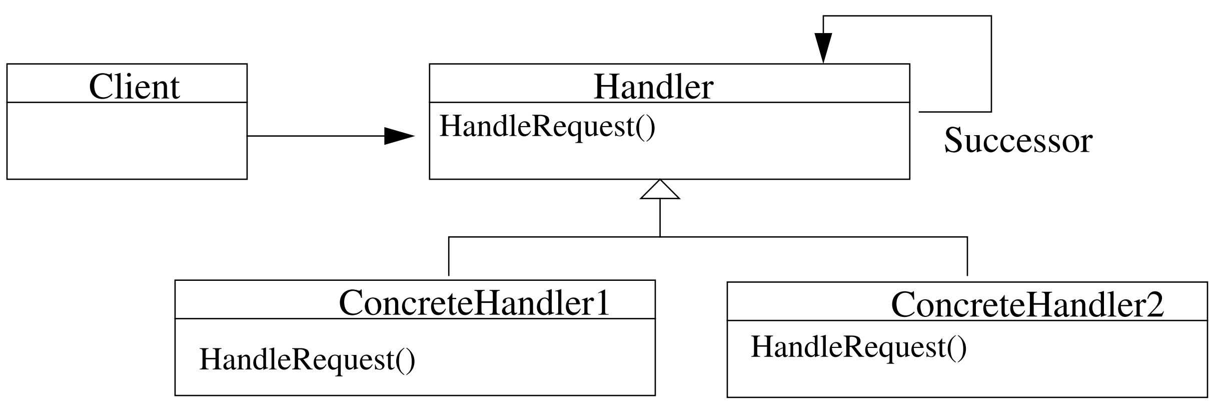
  Visitor makes adding new operations easier.
  A visitor gathers related operations and separates unrelated ones.

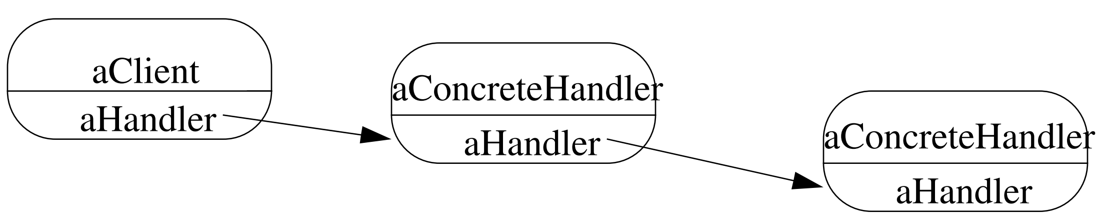
Chain of Responsibility

- **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.
- **Motivation**

  At times more than one object will be able to handle a request. These objects can be chained together and the request can be handed to each object in some type of priority ordering or it can be broadcast to all of the objects.
- **Structure**





Hardware-in-Software

Intent

The intent of the hardware-in-software pattern is to provide an interface between a piece of hardware and the software with which it interacts. This is often accomplished via a software object that acts as a surrogate for the piece of hardware.

Motivation

A traffic intersection utilizes sensors to determine the next valid configuration of green lights. The software controlling the intersection's lights must "read" each sensor to determine the feasible patterns. There are many types of sensors that are intermixed in traffic systems with different types of sensors even being used in the same intersection. A software sensor object decouples the intersection's controller from the particular type of hardware sensor being utilized. The same software system must also interface with the set of traffic signals that control the traffic in the intersection.

Applicability

This pattern is applicable whenever a piece of hardware is being referenced by a software system. Using this pattern simplifies the rest of the design by eliminating the need to consider the mechanisms by which real-time hardware information is captured for system use. In particular it can eliminate the type of design that makes the event handling system an integral part of the entire application.

Participants

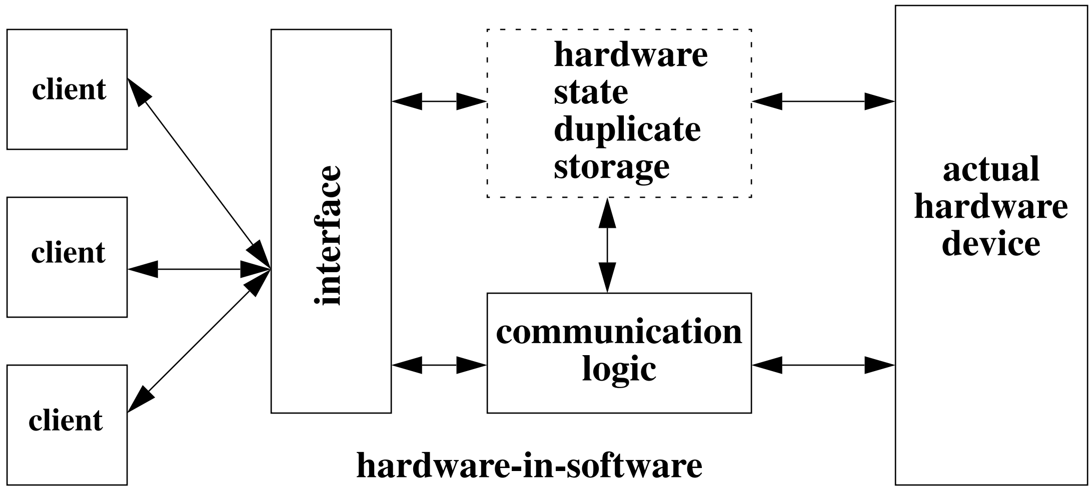
The classes that participate in the pattern include a class that provides the public interface of the hardware to the rest of the software system, the communications logic, and the optional duplicate state storage. The class providing the public interface decides whether to query the duplicate storage class for information to return to the client or to use the communications logic to query the hardware. The communications logic class provides the capabilities to interact with specific ports. The information received from the hardware may be returned directly to the interface class for immdiate use, handed to the duplicate state storage for retention and later use, or both. The state storage class is an optional part of the pattern. The rate of change of state of some hardware may be such that no duplicate state storage is allowed. However, where possible, this can eliminate the need for repeated query of the hardware, leaving it free for other operations. There is a potential connection between the state storage and the hardware directly to allow for direct memory access by the hardware.

Each of the three major class may well rely on objects from other classes such as the classes associated with notification between the hardware and software, communications devices such as ports and the classes that are needed to model the piece of hardware.

ollaborations

The pattern collaborates with two other entities. The first is the actual hardware itself. The hardware-in-software realization interfaces with the hardware through low-level communications devices such as a serial port. It is this interaction that is being hidden from the remainder of the application. The hardware-in-software pattern interacts with all clients of the hardware device. It provides the services of the hardware to these clients. The interaction with the client is assumed to be synchronous. For multiple concurrent clients, external synchronization is assumed.

Diagram



Consequences

The use of this pattern isolates the details of communication and control of the hardware device in one place. This allows similar devices to be substituted with minimal effort. This includes the ability to replace an actual piece of hardware with a simulator for that hardware. This can greatly ease the introduction of software into the field by using simulation testing prior to actual field testing. It also allows the algorithm for communication to be changed.

Implementation

This pattern will typically be realized by the cooperative efforts of a cluster of classes rather than a single class. The interface to the hardware-in-software pattern is the interface of the hardware and, as such, will vary widely depending upon the hardware being interfaced. Therefore it is impossible to define a single abstract representation of the interface. For a specific type of hardware, the collaborating peices of the pattern might be encapsulated within a single class and provided with a common interface. For example, in the traffic intersection example, a traffic signal abstract class could be defined to provide a high-level view of the pattern. This abstract class could be used to build classes for specific types of traffic light objects or pedestrian crosswalk signals. This class would, however, look quite different from the abstract class for the various types of sensors used in intersections.

There are several potential approaches to implementing the communication logic. These include interrupt-driven, event-driven, and polling arrangements. There are reasons to use each of these and we will detail those at a later time. The overall strategy is to define two interfaces: one to the actual hardware and one to the users of the pattern. The hardware interface will use one of the approaches listed above to communicate with the piece of hardware. The second interface will list all of the services that are associated with that piece of hardware. Some of the state of the hardware may be maintained in the hardware device itself. For example, the contents of the register that counts how far the mouse has traveled since the last click of a button is best stored in the hardware register of the mouse. Other values might be stored in software or even duplicated in both hardware and software for reasons of efficiency. For example, the current status of a button might be updated inthe software periodically by querying the mouse for the contents of its button register. In between updates, the software value could be used to answer the numerous queries resulting from a rubber band line drawing algorithm.

Examples

An example is the traffic control problem mentioned in the Motivation section. The devices in this case are relatively simple and change relatively slowly. The duplicate state storage would be useful in this case to speed the process of selecting the next phase, pattern of green lights. Once a sensor is "set", its state will not change until the intersection has moved to the next phase and all of the sensors have been reset. Therefore duplicate state storage would be easy to maintain. The system may wish to query several times while determining th enext phase to use for the intersection. This reduces the amount of interaction needed with the individual sensors.

A second example are the many hardware devices associated with a virtual reality system. The hardware-in-software patterns for the sensors on individual joints of individual fingers would be hierarchically related to ultimately provide an interface to the complete hand. In order to capture small movements of finger tips, no duplicate storage would be allowed.

See Also

Depending upon the set of patterns available, Event and Port patterns might be of use here. The Strategy pattern from Gamma, Helm, Johnson, and Villisides might be used as a high-level pattern for the communication logic.