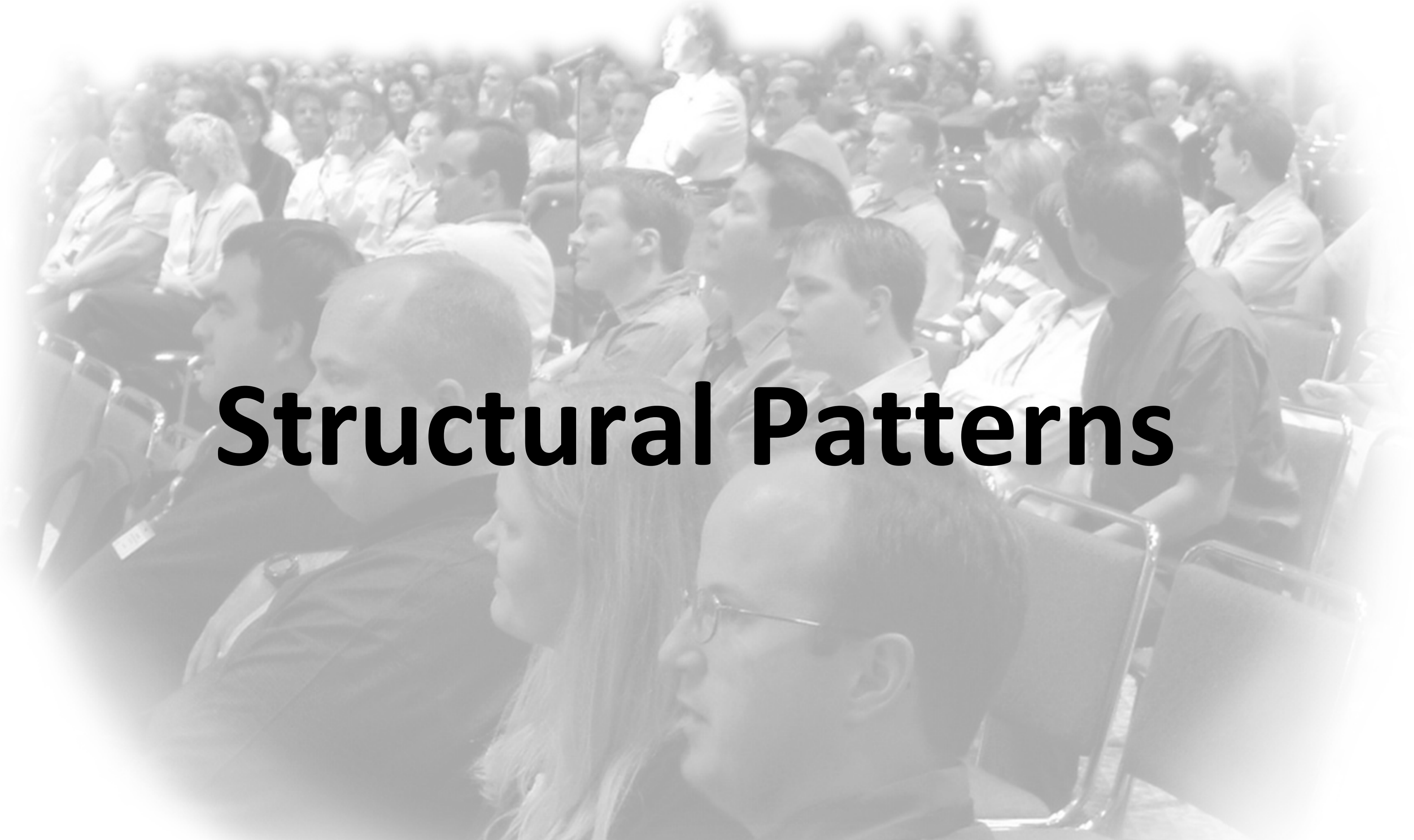


Structural Patterns

Agenda

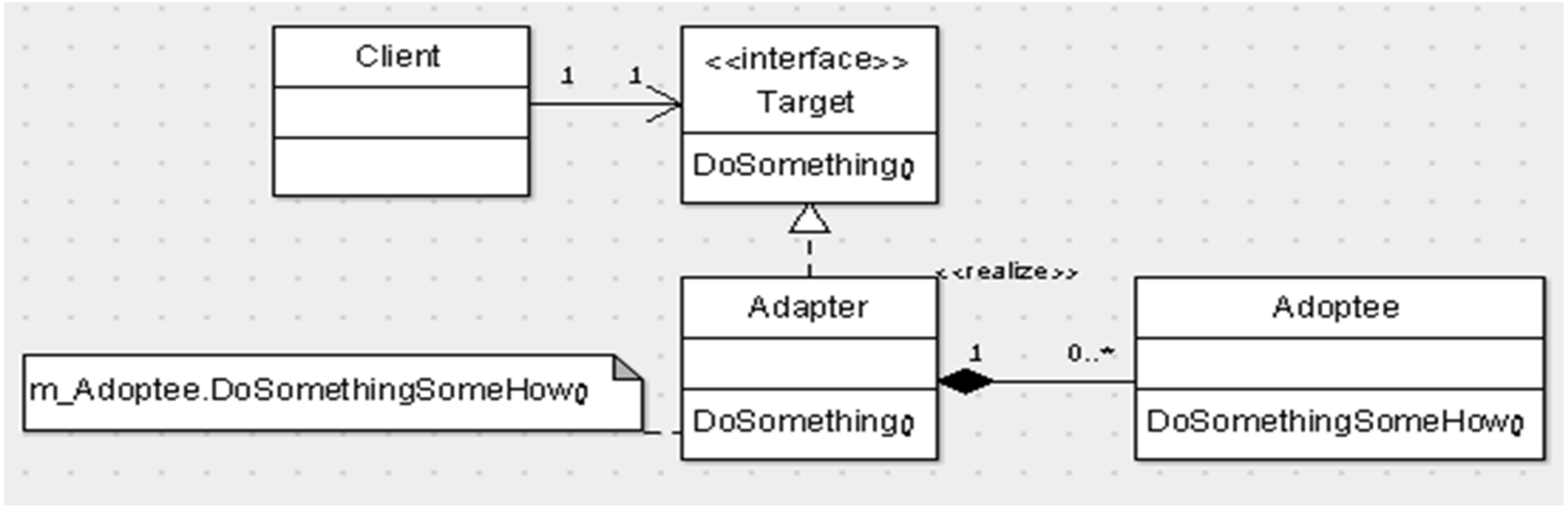
- Adapter
- Proxy
- Façade
- Composite
- Flyweight
- Bridge
- Decorator

Adapter

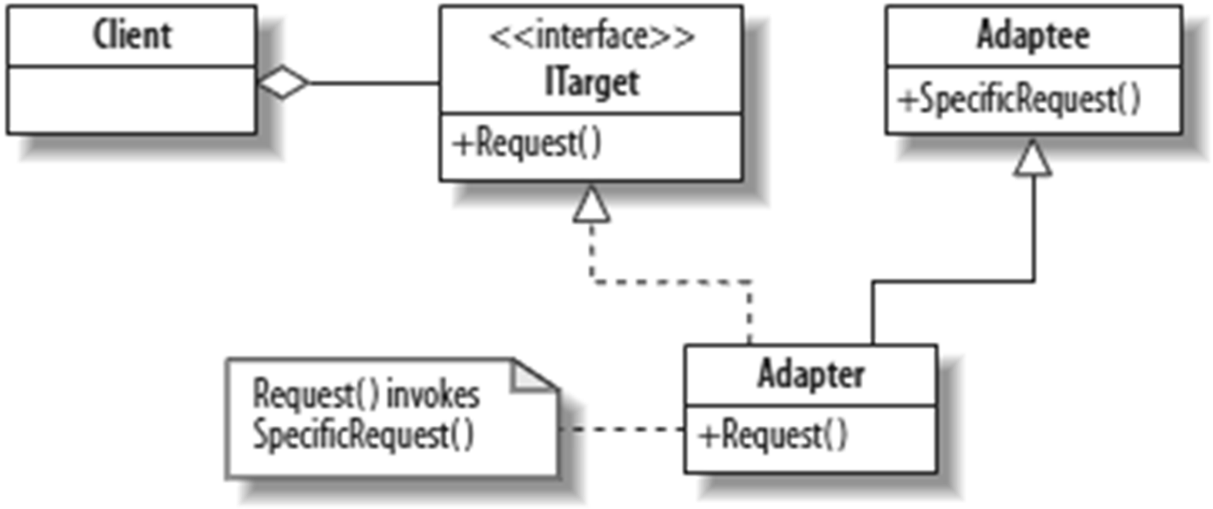
- Definition:
  Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Participant
  - **Target** – Defines the domain-specific interface that the **Client** uses
  - **Adaptee** – The existing object that needs to be adapted
  - **Adapter** – Adapts the object **Adaptee** to the **Target** interface
  - **Client**– Collaborates with objects conforming to the **Target** interfaces
- Motivation
  - We have a system that is used to use a certain type of object. We have an object that implements what the system needs, but with a different interface that the system knows. Both sides can not be changed. We can create an adapter, which implements the same interface the system knows, and invokes the given object accordingly
- Variations:
  - Class, Object, Two-Way, Pluggable

Adapter – Class Diagrams

- Gang-Of-4:
  (Object Adapter)



- Alternative:
  (Class Adapter)



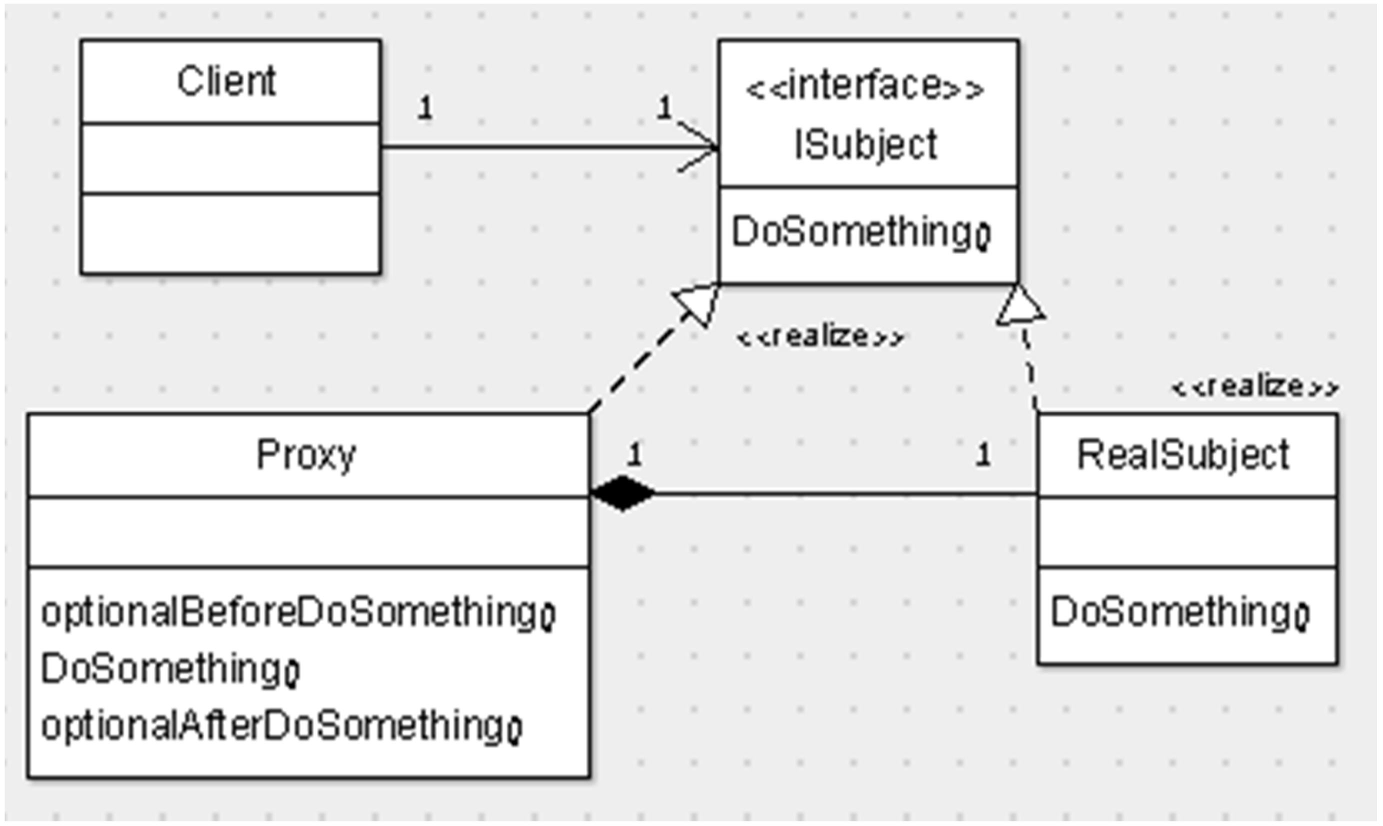
Adapter

- Adapter –Adoptee Scenarios
  - Re-Routing
  - Type conversions
  - Less to More
  - More to Less
- C# 3.0 features
  - Automatic Properties
  - Object Initializer

Proxy

- Intent:
  Provide a surrogate or placeholder for another object to control access to it.
- Motivation
  - Use an extra level of indirection to support distributed, controlled, or intelligent access.
  - Add a wrapper and delegation to protect the real component from undue complexity
  - **Virtual Proxy:**
    You want to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.
  - **Remote Proxy:**
    You want to hide the pluming and the complexity of working with a remote object
  - **Protective Proxy:**
    You want to control the access to a sensitive master object and to ensure that the caller has the access permissions required prior to forwarding the request
  - **Smart Proxy:**
    You want to encapsulate extra logic on an object like thread safety, lazy loading of persisted data, reference counting, caching

Proxy – Class Diagrams



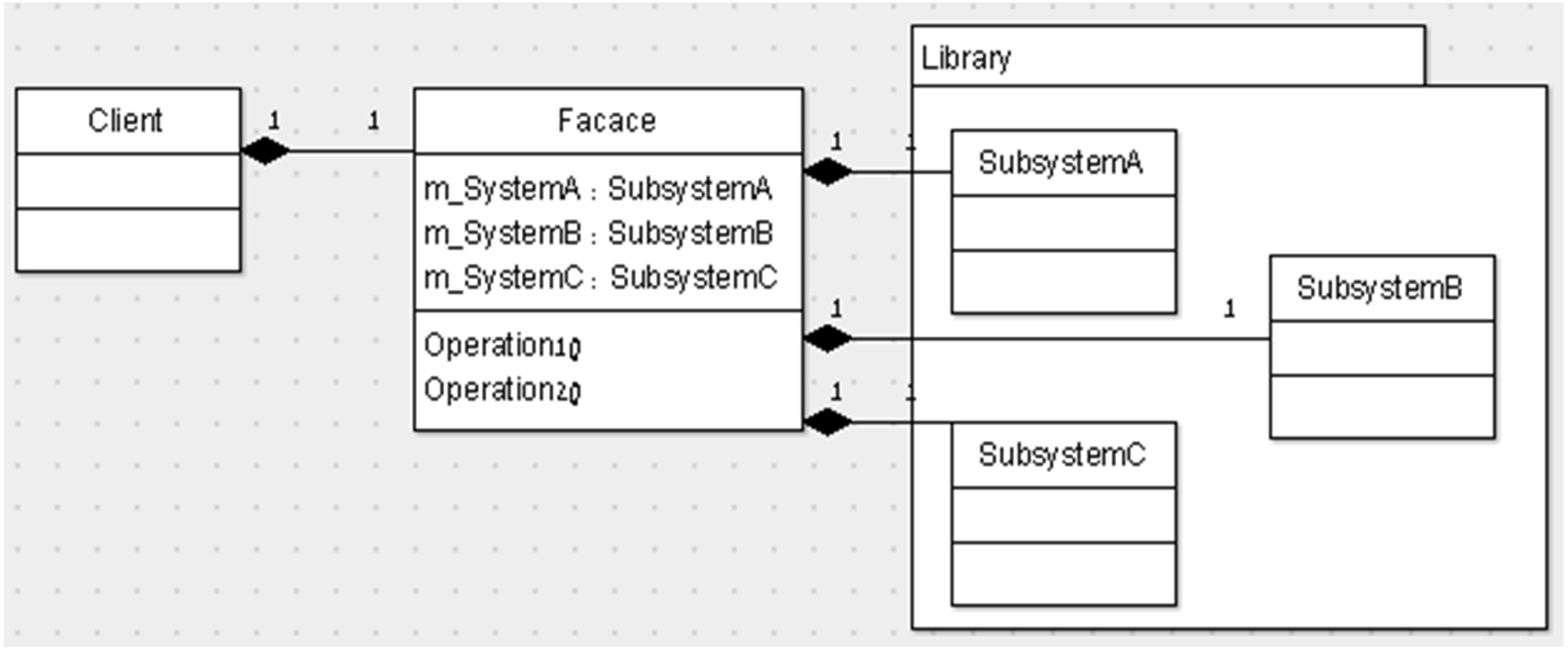
Proxy

- Participants
  - **Proxy**
    - Maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same
    - Provides an interface identical to Subject's so that a proxy can be substituted for the real subject
    - controls access to the real subject and may be responsible for creating and deleting it
  - **Subject**
    Defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected
  - **RealSubject**
    defines the real object that the proxy represents
- Variations:
  - Virtual Proxy
  - Authentication Proxy
  - Remote Proxy
  - Smart Proxy

Façade

- Definition:
  Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.
- Participant
  - **Sub-system Classes** – implement subsystem functionality
  - **Façade** – knows which **subsystem** classes are responsible for a request
  - **Client**– Uses the **façade** to easily operate the aggregated sub-systems
- Motivation
  - A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem
  - Reduces the learning curve necessary to successfully leverage the subsystem
  - Promotes decoupling the subsystem from its potentially many clients
- Variations:
  - Transparent Façade, Static Façade, Singleton Façade

Façade – Class Diagrams

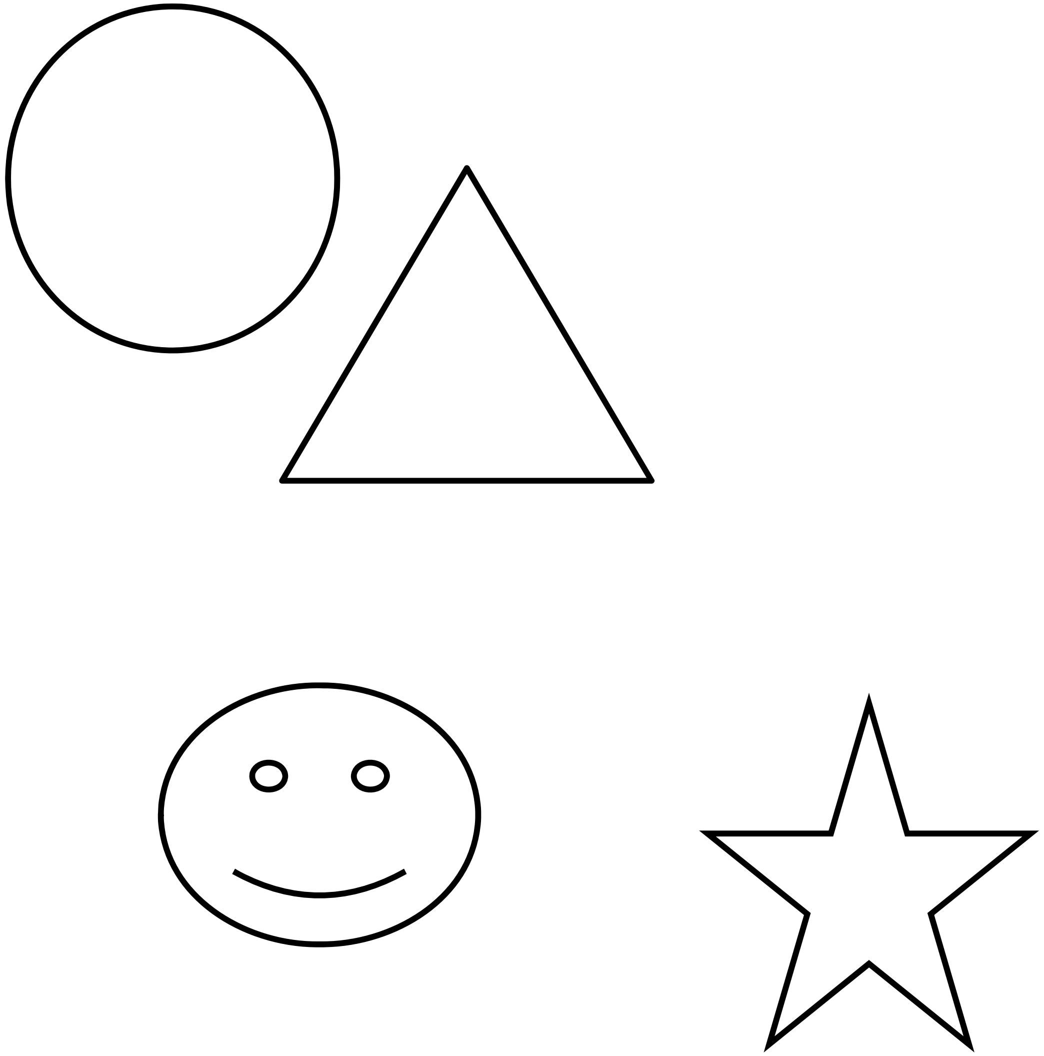


Composite

- **Intent**

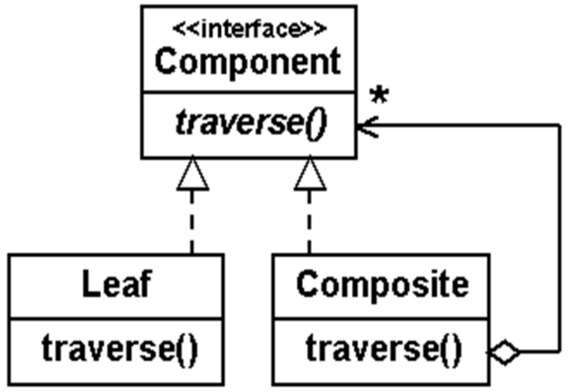
  Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly

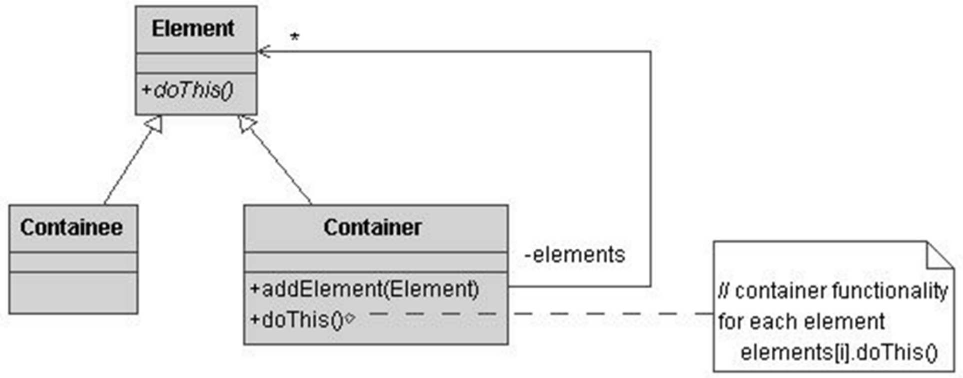
- **Motivation**
  - Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable
  - Applicability
    - To represent a part-whole hierarchies of objects
    - To make the difference between individual objects and compositions transparent to the client who uses them, and to make it treat all objects in the composite structure uniformly
  - Examples
    - Graphical objects in a graphics application (mspaint, powerpoint)
    - File system application (Directory/File)
    - Windows controls (Button/Panel)

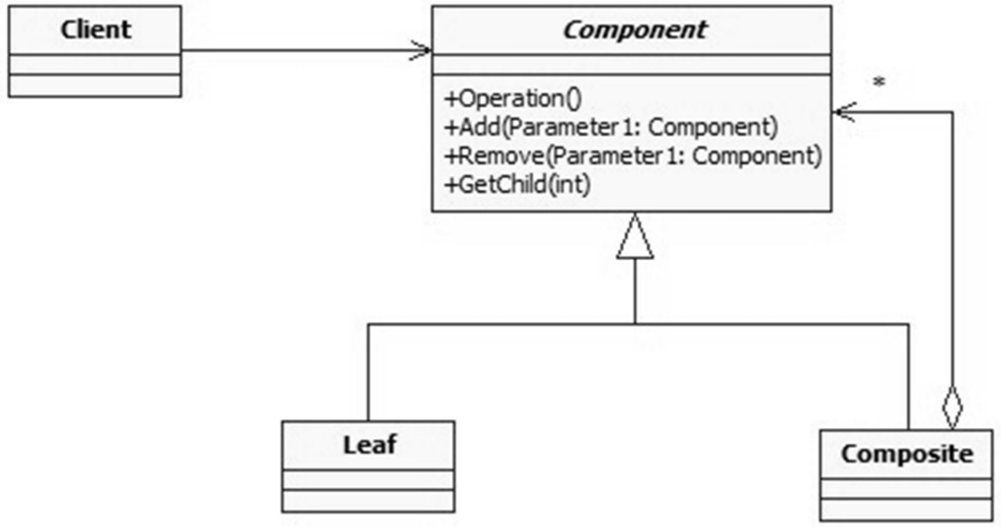


Composite

- **Structures**







- **Structure Basics**
  - Identify if your domain is: Collections that contain elements, each of which could be a collection
  - Define a base class that makes collection objects and element objects interchangable
  - Define the collection and element classes as derived classes
  - Define a one-to-many recursive relationship from the collection

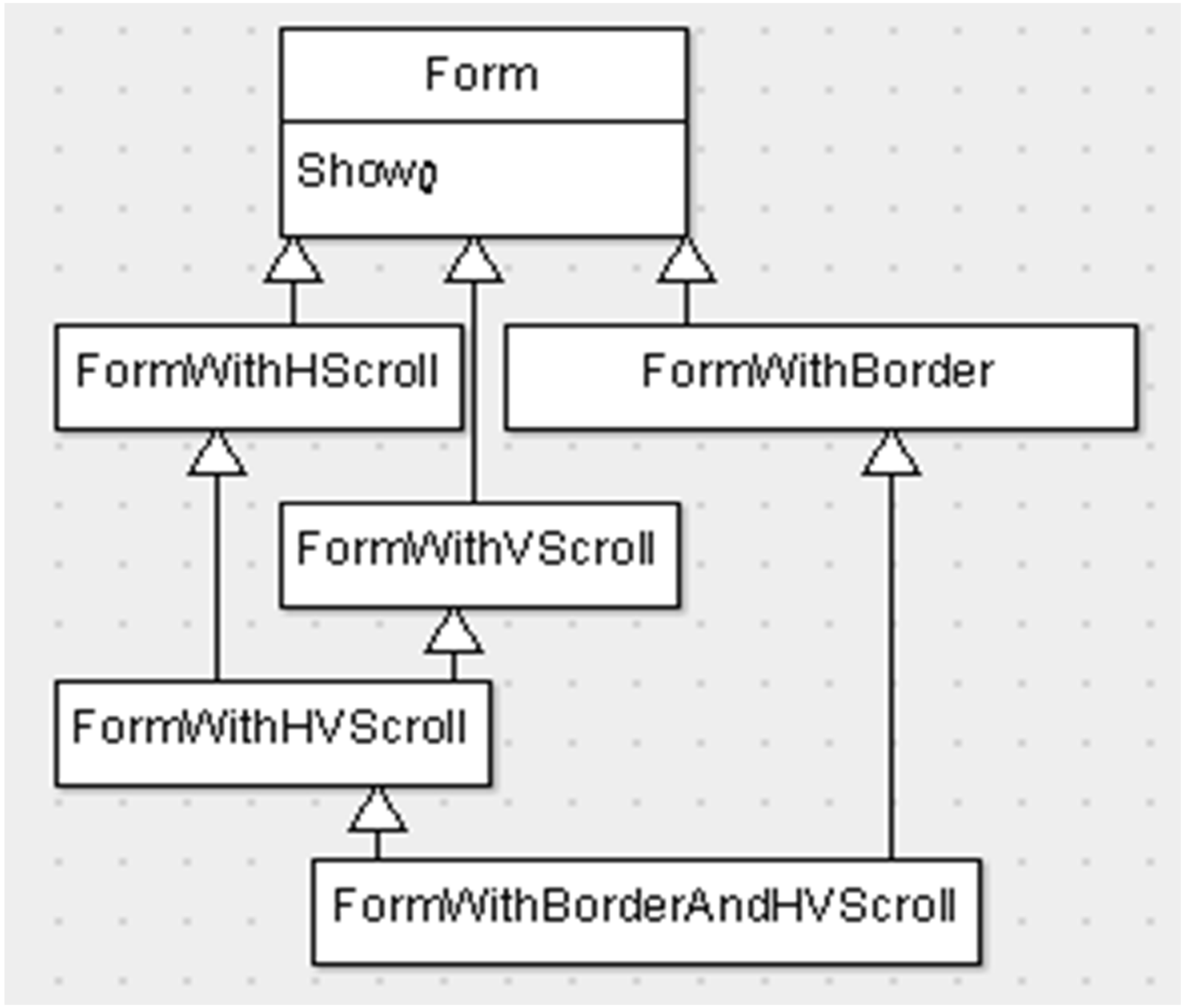
Composite

- **Participants**
  - Component (Windows.Forms.Control)
    - Declares the interface for objects in the composition (Size, Location, Visible..)
    - Implement default behavior for the interface common to all classes (don't paint if I am not visible)
    - Optional:
      Declares an interface for accessing and managing its child components
    - Optional:
      Declares an interfaces for accessing a component's parent in the recursive structure, and implements it if that's appropriate (m_Button.Parent)
  - Leaf (TextBox, Button, Label)
    - Defines behavior for primitive objects in the composition
  - Composite (Panel, GroupBox)
    - Stores child components (optionally in the base component class)
    - Implement child related operations declared in the Component interface

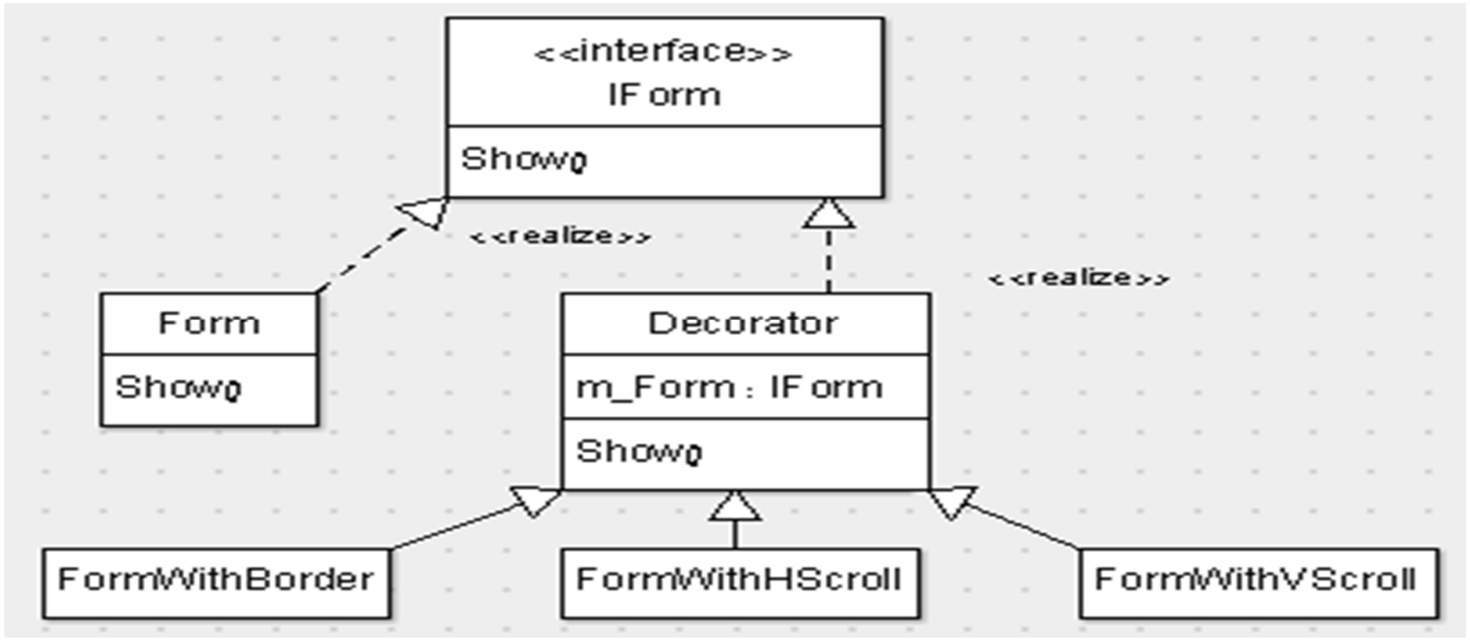
Composite

- **Implementations**
  - Explicit parent references
    - Good for back-tracking and caching (relates to the observer pattern)
    - Maintaining the invariant
  - Sharing Components
    - Good for reducing storage
    - Problematic when an component can not have more than one parent (relates to the Flyweight pattern)
  - Maximizing the Component Interface
    - Good for transparency and abstraction
    - Bad for code safety
  - Declaring the child management operations (safety vs. transparency)
    - In this pattern – Transparency over Safety is emphasized.
    - Icomposite vs. GetComposite
    - Throwing exceptions in leaf implementations
    - Change the meaning in leaf implementations
  - Implementing the list of components in the Component base class
  - Child ordering
  - Caching in the composite
  - Disposing the child components
  - Data structures for holding the child components

Decorator


Form
Show()
FormWithHScroll
FormWithBorder
FormWithVScroll
FormWithHVScroll
FormWithBorderAndHVScroll

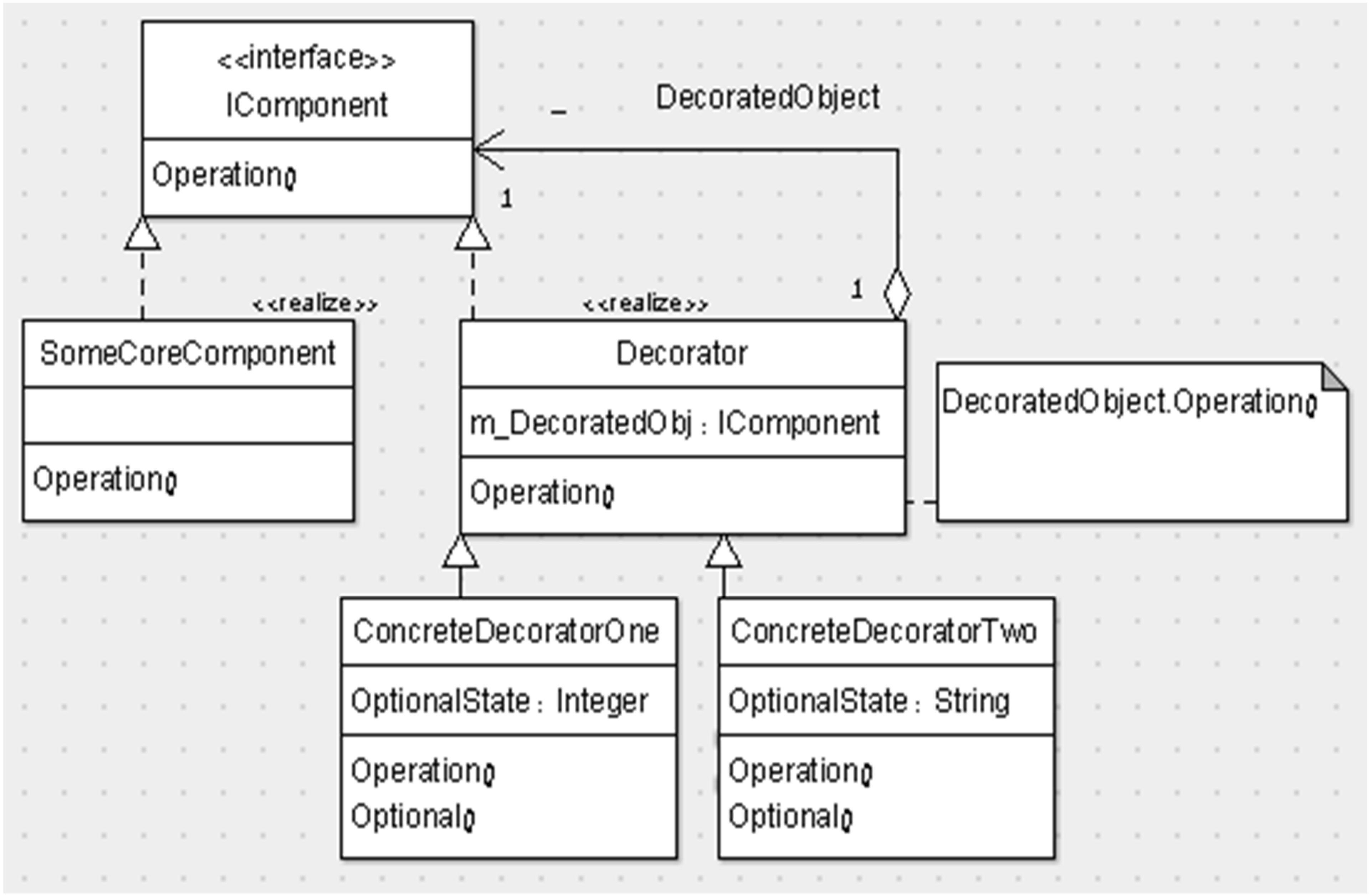
Decorator



```
IForm form = new BorderDecorator(
             new HorizontalScrollBarDecorator(
                   new VerticalScrollBarDecorator(
                         new Form( 80, 24 ))));
form ->Show();
```

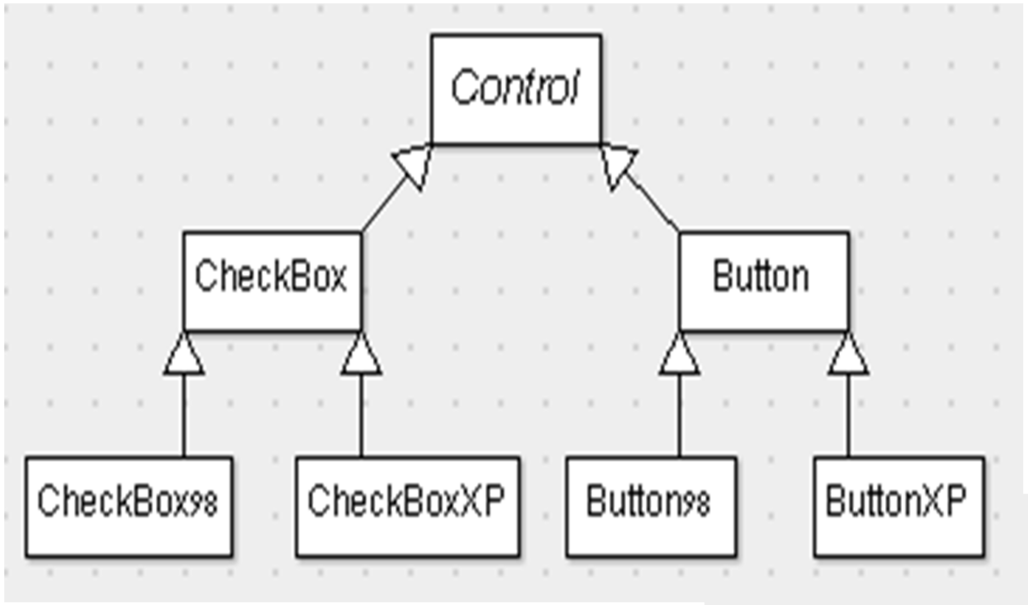
```
Stream stream = new CompressingStream(
                     new ASCII7Stream(
                           new FileStream("fileName.dat")));

stream->AddString("Hello world");
```

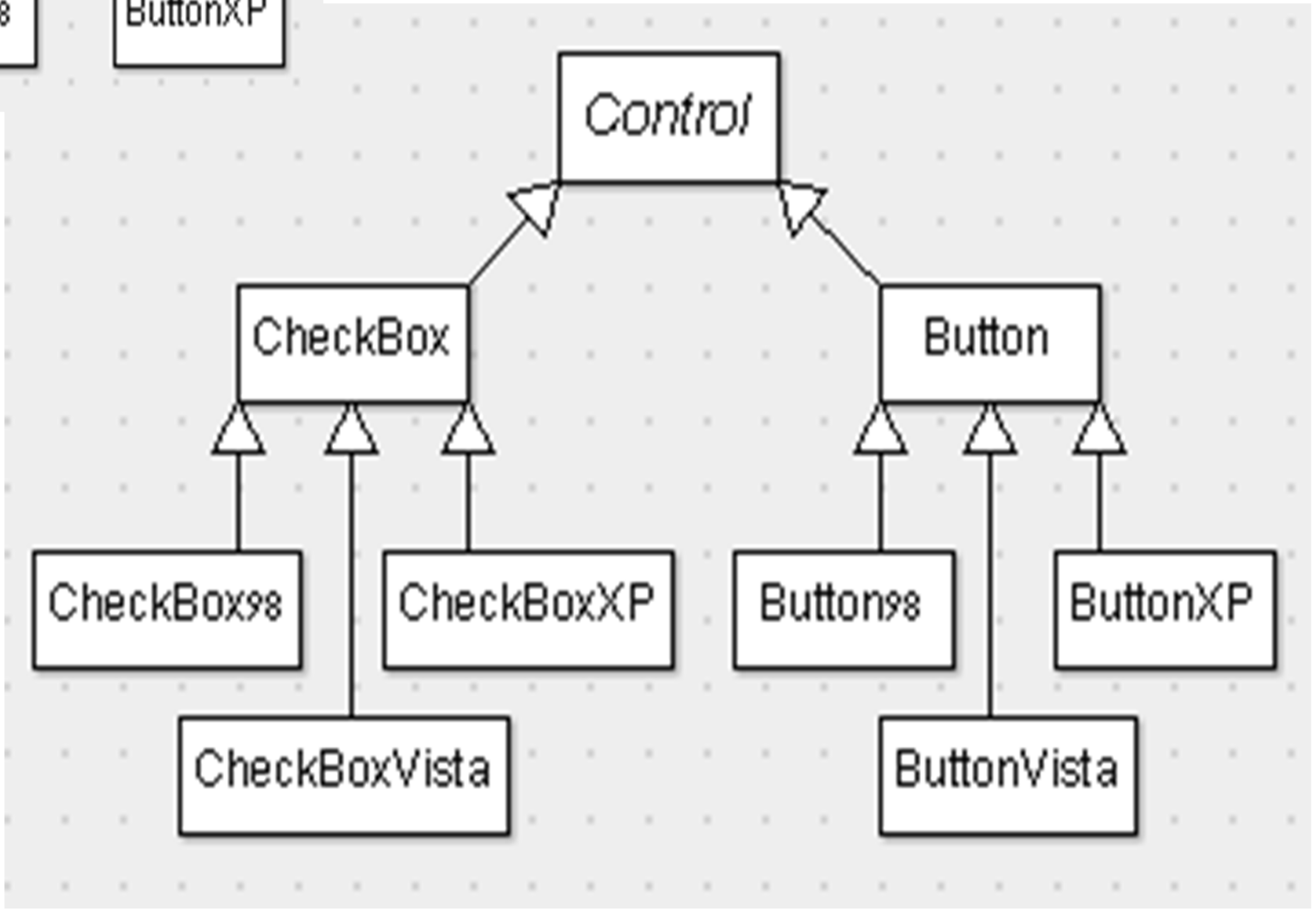
Decorator



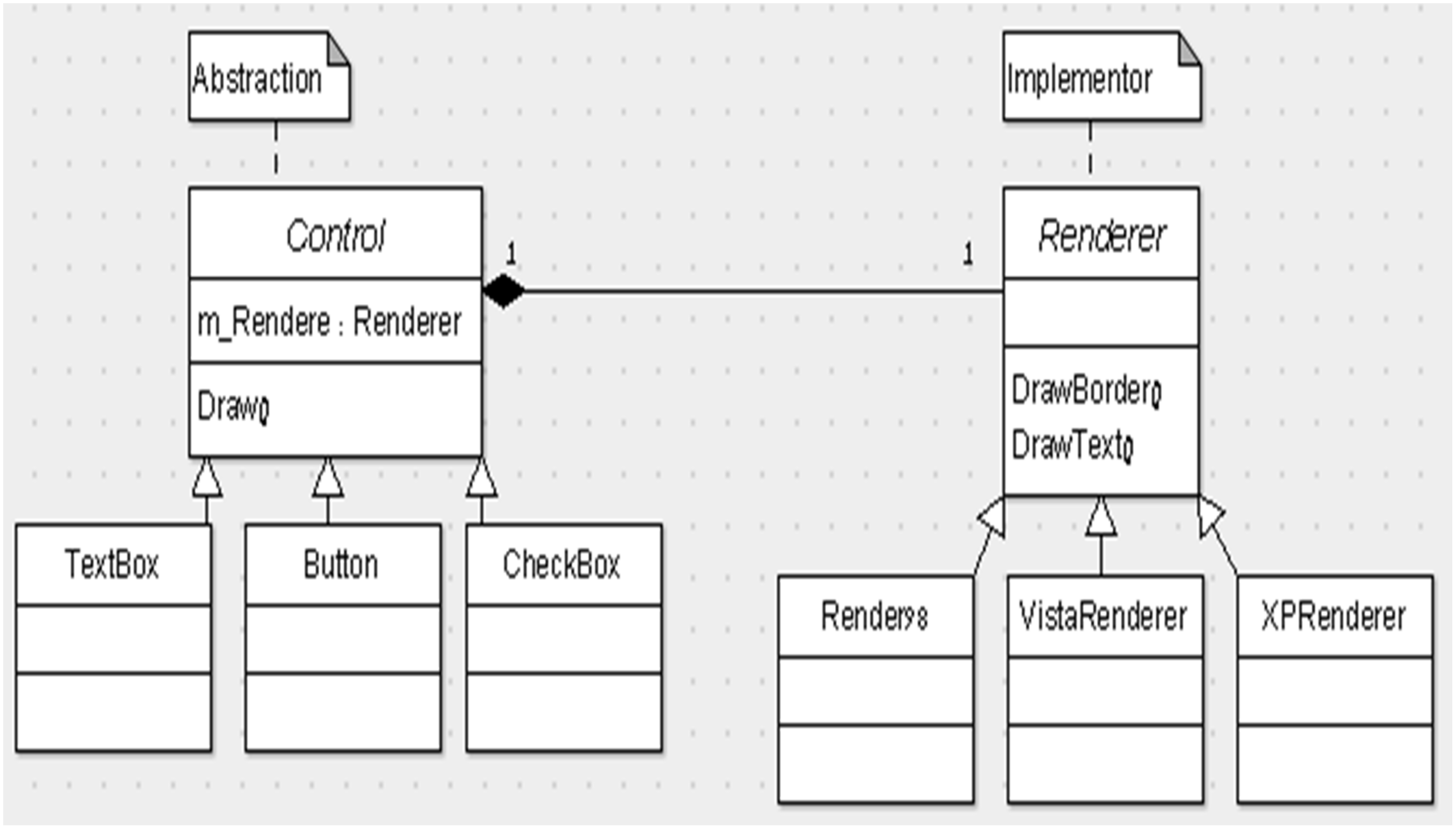
<<interface>>
IComponent
Operation()
DecoratedObject
1
1
<<realize>>
<<realize>>
SomeCoreComponent
Operation()
Decorator
m_DecoratedObj : IComponent
Operation()
DecoratedObject.Operation()
ConcreteDecoratorOne
OptionalState : Integer
Operation()
Optional()
ConcreteDecoratorTwo
OptionalState : String
Operation()
Optional()

Bridge


Control
CheckBox
Button
CheckBox98
CheckBoxXP
Button98
ButtonXP

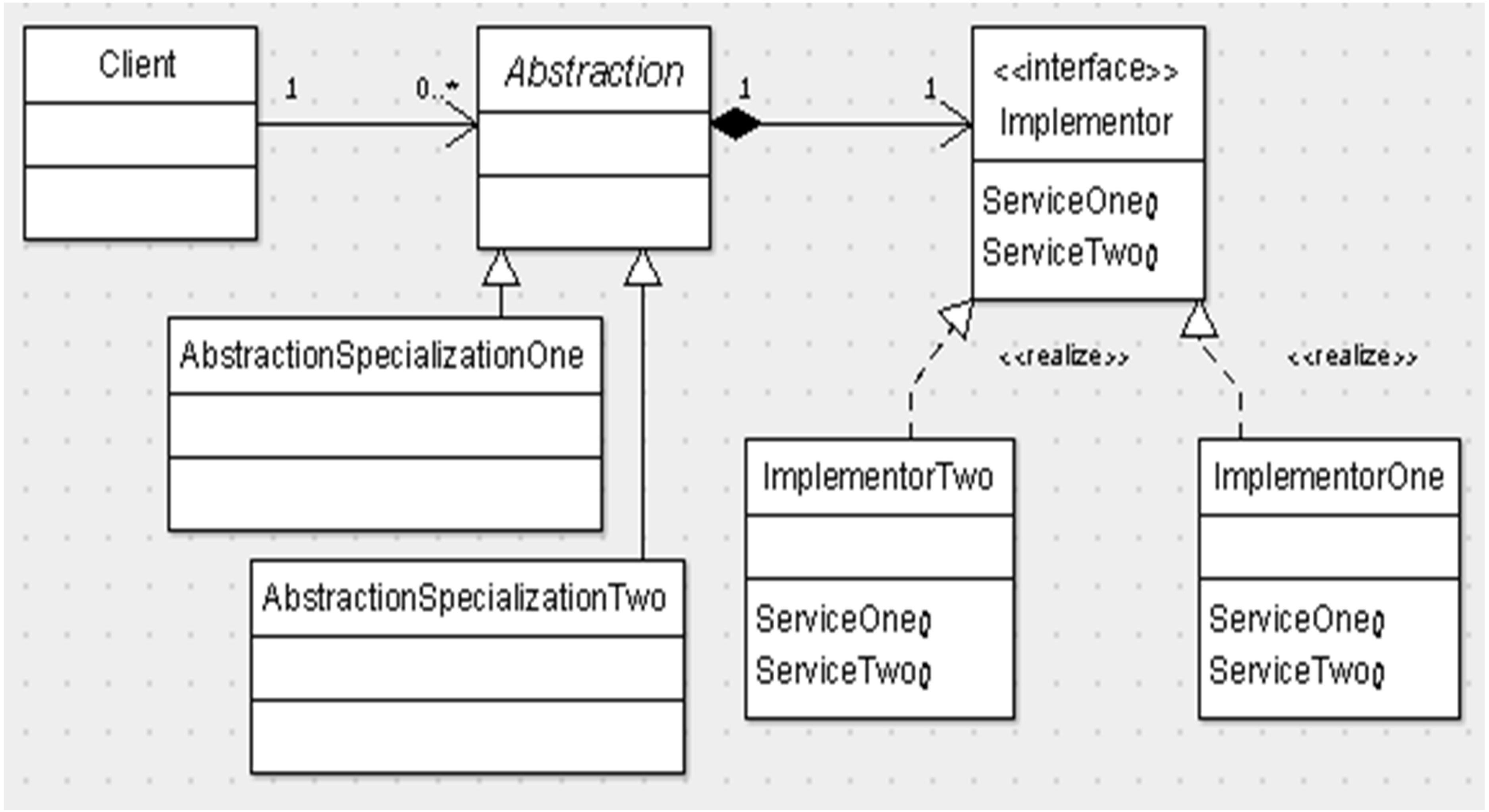

Control
CheckBox
Button
CheckBox98
CheckBoxXP
Button98
ButtonXP
CheckBoxVista
ButtonVista

Bridge


Abstraction
Implementor
Control
m_Rendere : Renderer
Draw()
1
1
Renderer
DrawBorder()
DrawText()
TextBox
Button
CheckBox
Render98
VistaRenderer
XPRenderer

Bridge

Structure
The Client doesn't want to deal with platform-dependent details. The Bridge pattern encapsulates this complexity behind an abstraction "wrapper".


Client
1
0..*
Abstraction
1
1
<<interface>>
Implementor
ServiceOne()
ServiceTwo()
AbstractionSpecializationOne
AbstractionSpecializationTwo
<<realize>>
<<realize>>
ImplementorTwo
ServiceOne()
ServiceTwo()
ImplementorOne
ServiceOne()
ServiceTwo()