10. Structural Pattern

Structural Patterns

- Concerned with how classes and objects are composed to form large structures
- ***Class*** Patterns use ***inheritance*** to compose interfaces or implementation
- ***Object*** Patterns describe ways to ***compose*** objects to realize new functionality

Adapter Pattern

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

Example that would benefit from Adapter Pattern

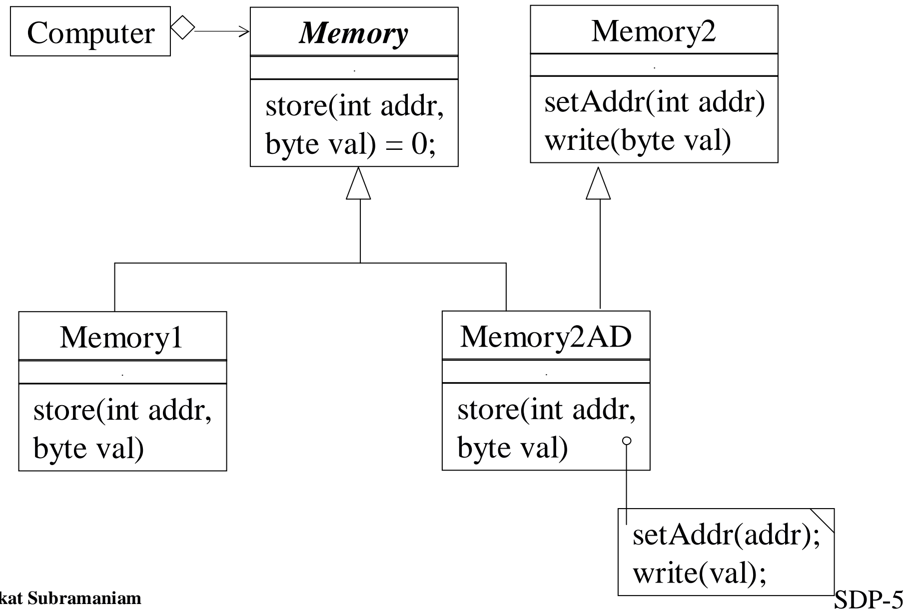
- A Computer class uses an abstract Memory that supports the given interface:

```
class Memory {virtual void store(int addr, byte value) = 0;};
```

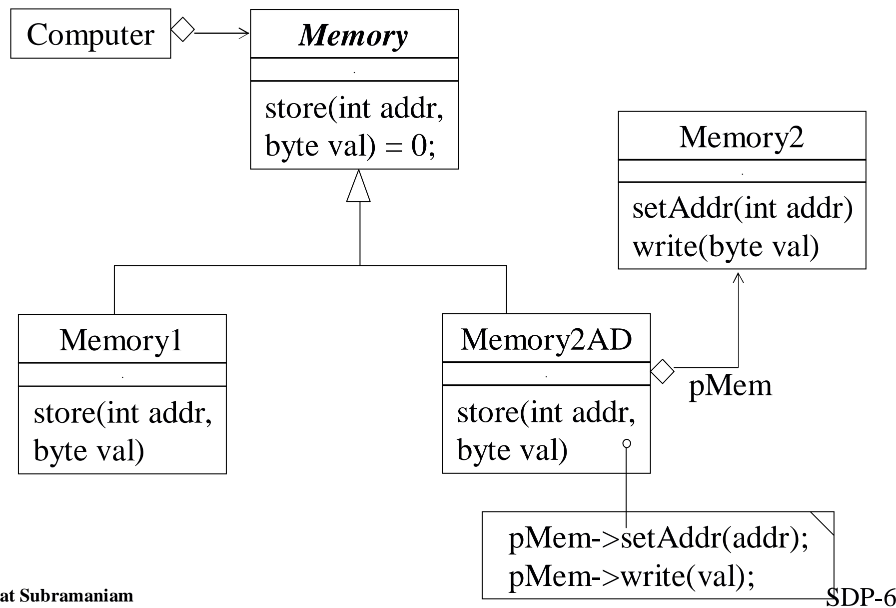
A Memory1 supports this interface. Another department has a class Memory2 that I would like to use. However, this model does not support the above interface.

```
class Memory2 {…
   virtual void setAddr (int address); // Set addr to access next
   virtual void write(byte value);
};
```

Example using Class Adapter



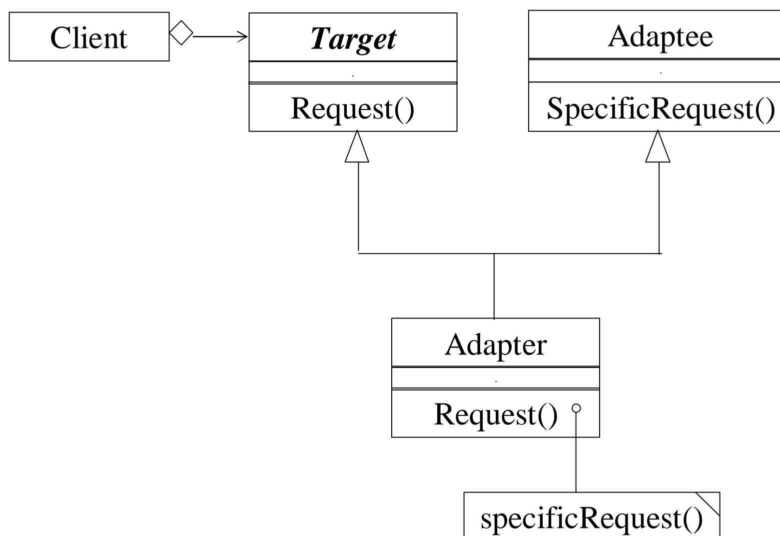
Example using Object Adapter



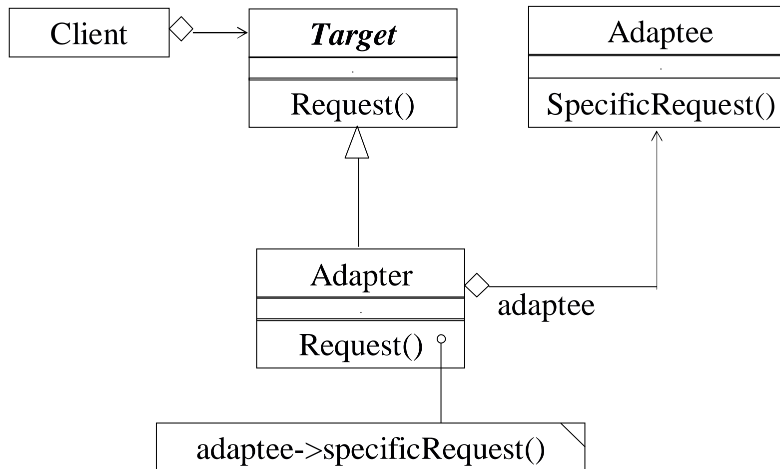
When to use Adapter Pattern

- You want to use existing class, and its interface does not match the one you need
- You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces
- (object adapter) you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class

Structure - Class Adapter



Structure - Object Adapter



Consequences of Using Adapter

Class Adapter:

- Adapts Adaptee to Target by committing to a concrete Adapter class
- Can't adapt a class and all its subclasses
- Lets Adapter override some of Adaptee's behavior
- No extra objects due to adapter usage

Object Adapter:

- Lets a single Adapter work with many Adaptees - a class and its subclasses
- Can add functionality to all Adaptees
- Harder to override Adaptee behavior

Adapter Vs. Other Patterns

- Structure similar to Bridge, different intent
  - Bridge: separate interface from implementation
  - Adapter: change the interface
- Decorator enhances another object without changing interface

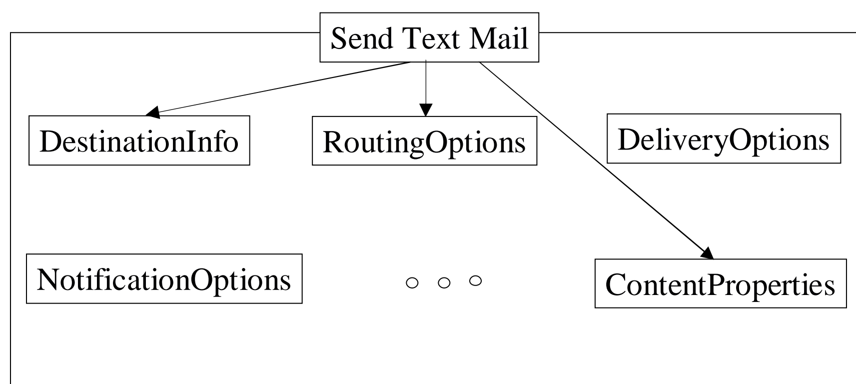
Façade Pattern

"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."

Example that would benefit from Façade Pattern

- A hi-tech mailer subsystem has several classes to specify the destination of mail, routing preference - shortest, most reliable, most economical, etc. and delivery options like urgent, important, confidential, encrypted and return notification options.
- While some clients of the subsystem may be interested in these finer classes, others may just be interested in a subset that provides basic functionality

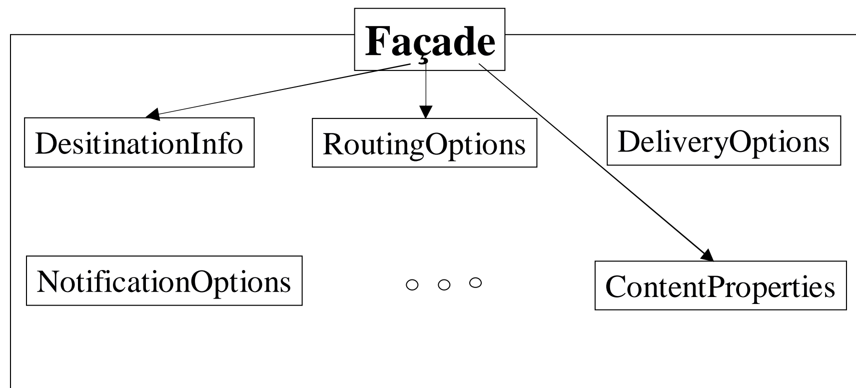
Example using Façade Pattern



When to use Façade Pattern

- You want to provide simple interface to a complex subsystem
  - Clients that need no customized features do not look beyond Façade
- You want to decouple subsystem from clients
- You want to layer the subsystem to reduce dependency between various levels of the subsystem

Structure



Consequences of using Façade

- Shields clients from subsystem component
- Makes subsystem easier to use
- Promotes week coupling between client & subsystem
- Helps layer subsystem
- Reduced coupling helps build time for large systems
- No limit on clients looking beyond Façade

Façade Vs. Other Patterns

- Abstract Factory may be used with Façade
  - interface for creating subsystem objects in subsystem independent way
- Abstract Factory may be an alternate to Façade to hide platform-specific classes
- Mediator similar to Façade
  - In Mediator, Colleague objects know Mediator
  - In Façade subsystem classes do not see Façade
- Usually Singletons

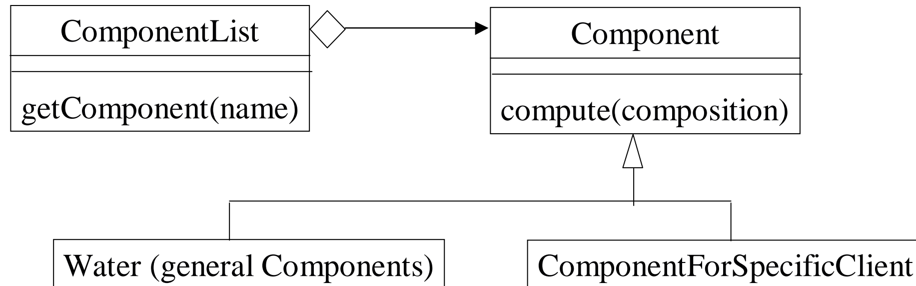
Flyweight Pattern

"Use sharing to support large number of fine-grained objects efficiently"

Example that would benefit from Flyweight Pattern

- An Engineering application uses several Chemical components
- Each component has properties like its molecular weight, density, etc.
- The components are used all over the application with additional information about composition quantities, unit specifications, etc.
- Storing the components with all its properties is prohibitively expensive

Example using Flyweight Pattern



When to use Flyweight Pattern

- An application uses large number of objects
- Impractical to have that many objects
- State can be made extrinsic
- Groups of objects may be shared with intrinsic properties upon removal of extrinsic properties
- Object identify is not an issue - not used for comparison purposes

Consequences of using Flyweight

- Storage saving
- Intrinsic state information grouped together
- Flexibility to introduce new variations of objects
- Run-time overhead for finding objects, computing extrinsic state

Flyweight Vs. Other Patterns

- Combined with Composite
- State and Strategy Patterns often implemented as flyweights

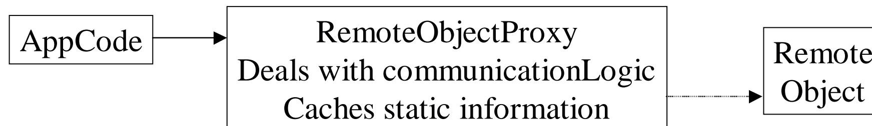
Proxy Pattern

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

Example that would benefit from Proxy Pattern

- An application needs to communicate with a remote subsystem to obtain some critical information
- The application may have code all over that deals with communication logic and data
- How to minimize the code for communication logic?
- What if not all data on the remote subsystem is changing dynamically?

Example using Proxy Pattern



When to use Proxy Pattern

- A more sophisticated reference than a simple pointer is needed
- Remote proxy provides local representative for object in different address space
- Virtual proxy creates expensive objects on demand
- Protection proxy controls access to original object
- Copy-on-modify proxy guts of data not copied until if and when data modification starts
- Smart pointers are needed to
  - manage object lifetime
  - loading object into memory when first referenced
  - lock management for synchronization

Consequences of using Proxy

- Introduces a level of indirection in accessing objects
- Indirection provides flexibility
- Incurred cost on additional object and computations

Proxy Vs. Other Patterns

- Adapter changes interface, Proxy provides the same interface
  - protection proxy implements subset of interface - may deny access to certain functions
- Decorator adds functionality, proxy controls functionality

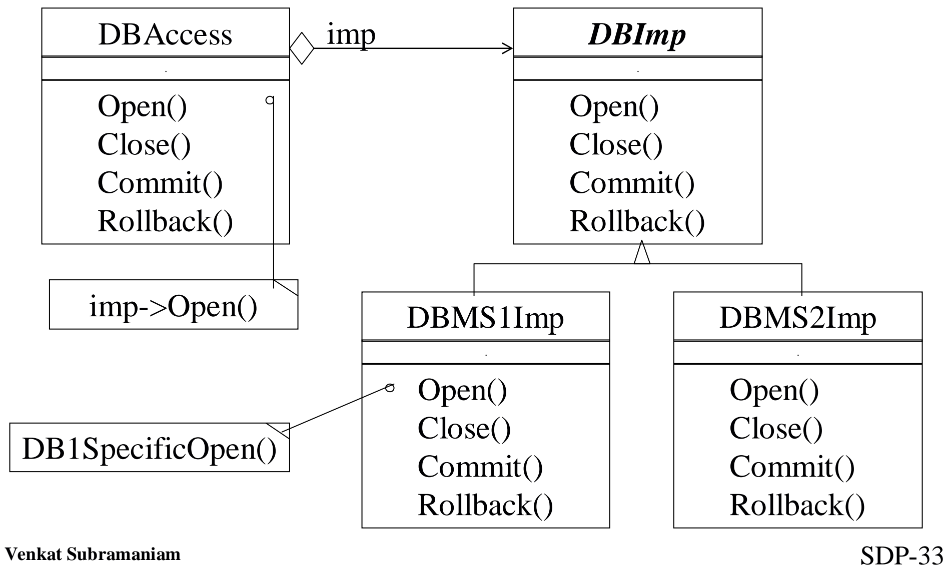
Bridge Pattern

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

Example that would benefit from Bridge Pattern

- An application wants to be able to use one of several databases available. However, each database has different API. How to write one set of code such that the code is not affected by which database is used or when new database is considered?

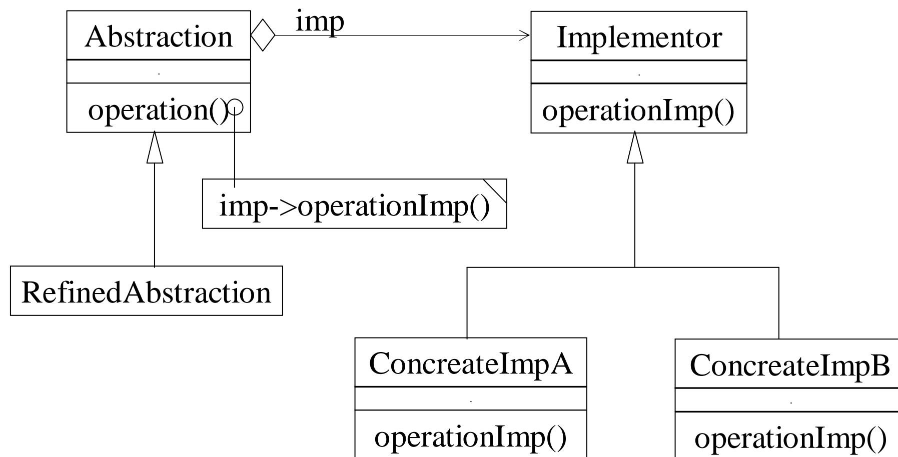
Example using Bridge Pattern



When to use Bridge Pattern

- You want to avoid a permanent binding between an abstraction and its implementation - esp. when implementation may be selected or switched at runtime
- Both the abstractions and their implementations should be extensible by subclassing.
- Change in the implementation of an abstraction should not impact the clients - no recompilation of client code
- In C++, you want to hide the implementation from the .h file
- Avoids proliferation of classes

Structure



Consequences of using Bridge

- Decoupling interface and implementation - may be configured at runtime - may even be changed
- Eliminates compile time dependency on implementation
- Encourages layering - resulting in better system
- Improved extensibility
- Shields clients from implementation details

Bridge Vs. Other Patterns

- Abstract Factory can create and configure a particular Bridge
- Different from Adapter Pattern:
  - Adapter - making unrelated classes work together - usually applied to systems after redesign
  - Bridge: lets abstraction and implementation vary independently - used up-front in design

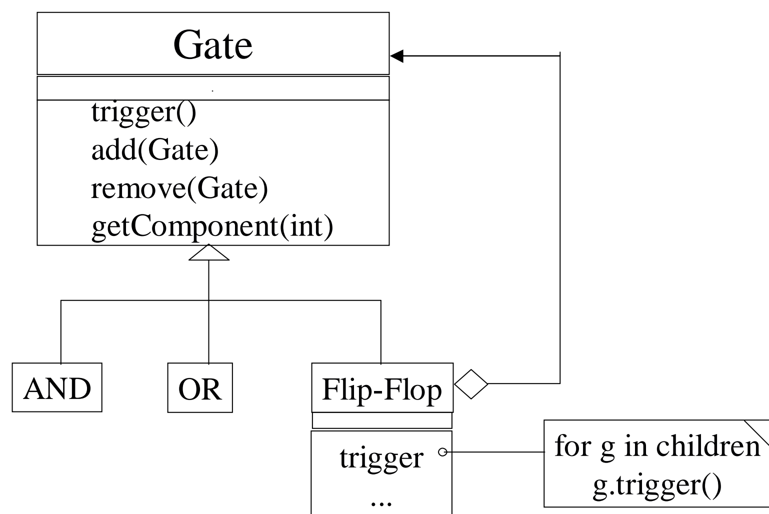
Composite Pattern

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

Example that would benefit from Composite Pattern

- An application wants to be able to use several types of Gates. Gates like AND, OR are primitive, while Gates like Flip-Flops are Containers of other gates. Define the hierarchy of these classes such that the client can treat all the types of Gate classes uniformly

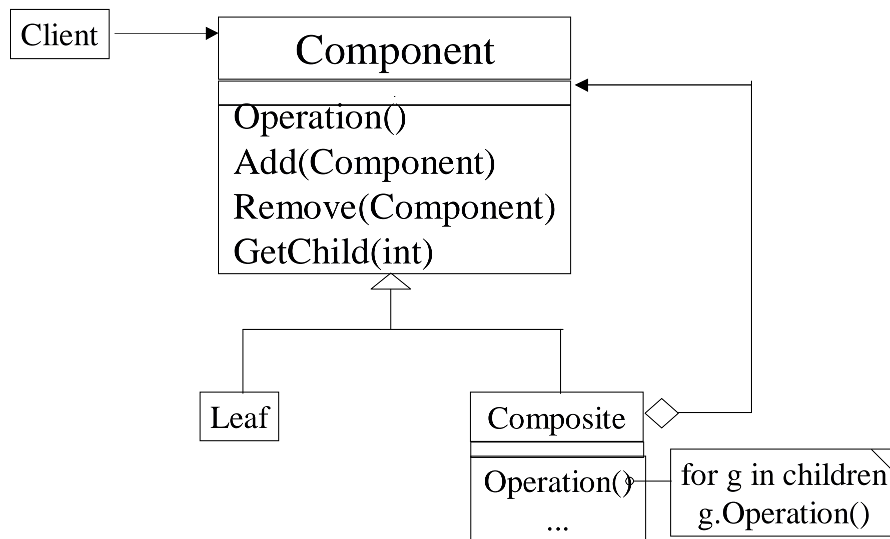
Example using Composite Pattern



When to use Composite Pattern

- You want to represent part-whole hierarchies of objects
- You want the clients to be able to ignore the differences between the compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly

Structure



Consequences of using Composite

- Primitive objects recursively composed int more complex objects
- Where ever client code expects primitive object, it can also take a composite
- Simplifies Client code - can treat composite & individual objects uniformly
- Easier to add new kinds of Components
- Makes design overly general
- Harder to restrict the components of a composite

Composite Vs. Other Patterns

- Used for Chain Of Responsibility
- Decorator often used with Composite
- Flyweight lets you share components but they no longer refer to their parents
- Iterator can be used to traverse Composites
- Visitor localizes operations & behavior that would otherwise be distributed across composite and leaf classes

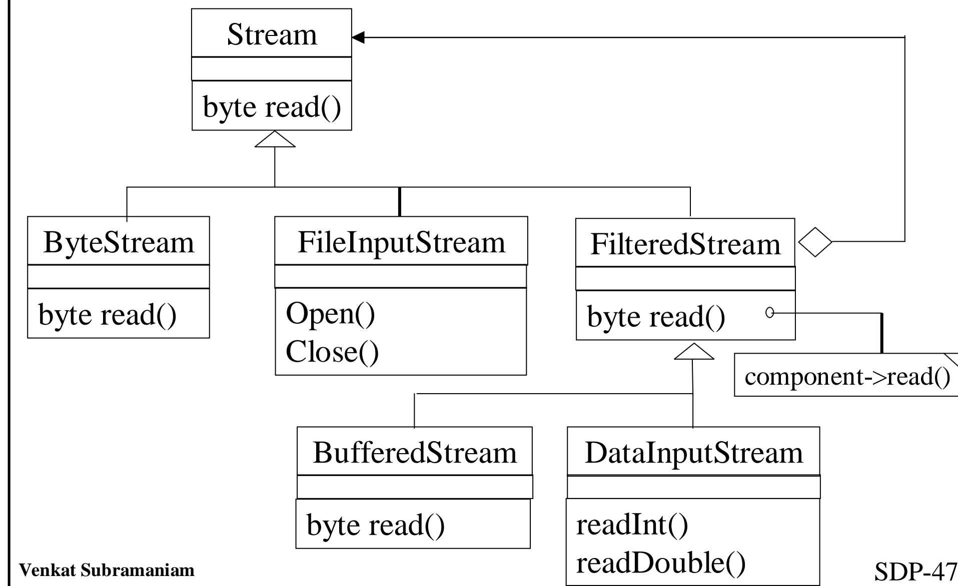
Decorator Pattern

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."

Example that would benefit from Decorator Pattern

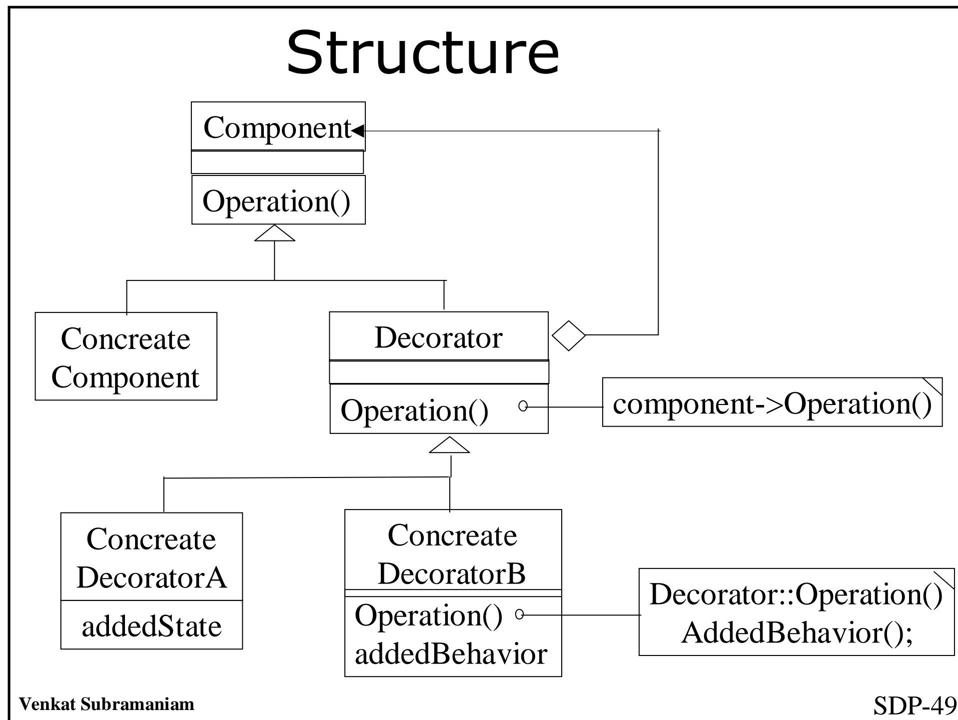
- A stream of bytes may be read as raw data. It may also be read from a file. It may be read as a buffered stream. We may also read integer, double, etc. from the stream. A user may use one or a combination of the above features.
- We don't want to create several classes with a combination of these features

Example using Decorator Pattern



When to use Decorator Pattern

- To add responsibilities to individual objects dynamically and transparently, without affecting other objects
- Responsibilities may be withdrawn
- Extension by subclassing is impractical



Consequences of using Decorator

- Flexibility compare to static inheritance
  - Functionality may be added / removed at runtime
- You only get features you ask for
- Decorator acts as transparent enclosure
- Can't rely on Object identify
- Lots of little objects

Decorator Vs. Other Patterns

- Adapter changes objects interface. Decorator changes only its responsibilities
- Decorator is not intended for object aggregation like Composite
- Strategy and Decorator are used to change an object - Strategy lets you change the guts of an object - Decorator its skin