



## Design Patterns

- Design Patterns
- Composite Pattern
- Observer Pattern



## Job Annoucement

- CVAP has part-time positions for students
  - Programming
  - Maintaining software & hardware
  - Flexible working hours
  - Up to 10 hours per week
  - Contact: Frank Hoffmann or Anders Orebäck



## DIA

- DIA is a diagram creation program from Linköping University  
<http://www.lysator.liu.se/~alla/dia/dia.html>
- It can be used to draw many different kinds of diagrams
  - Entity relationship diagrams
  - UML diagrams
  - Flowcharts
  - Network diagrams
  - Simple circuits



## Design Patterns

- Literature:
  - Design Patterns
  - Elements of Reusable Object-Oriented Software
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
  - Addison-Wesley, Professional Computing Series



## Design Patterns

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Christopher Alexander



## Design Patterns

- Each design pattern names, explains and evaluates an important and recurring design in object-oriented systems.
- A design pattern makes it easier to reuse successful designs and architectures.
- Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability.



## Design Patterns

- The *pattern name* is a handle to describe the design problem, its solutions and consequences.
- The *problem* describes when to apply the pattern. It explains the problem and its context.
- The *solution* describes the elements (classes and objects) that make up the design, their relationships, responsibilities and collaborations.
- The *consequences* are the results and trade-offs of applying the pattern. They may address language and implementation issues as well.



## Object-Oriented Design

- The hard part about object-oriented design is decomposing a system into objects because many factors come into play:
  - Encapsulation
  - Granularity
  - Dependency
  - Flexibility
  - Performance
  - Evolution
  - Reusability

## Program to an Interface not an Implementation

- All classes derived from an abstract class share its interface.
- Manipulating objects solely in terms of their interface defined by the abstract class offers two major benefits.
- Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.
- Clients remain unaware of the classes that implement these objects. Clients only know about the abstract classes defining the interface.
- *Program to an interface not an implementation.*

## Inheritance vs. Composition

- The two most common techniques for reusing functionality in object-oriented systems are *class inheritance* and *object composition*
- Class inheritance defines the implementation of one class in terms of another's implementation. With inheritance the internals of parent classes are often visible to sub-classes (*white box*).
- In object composition new functionality is obtained by assembling or composing objects to get more complex functionality. Internal details of objects are not visible, objects appear as *black boxes*.

## Pros and Cons of Inheritance

- Pros: Class inheritance is defined statically at compile-time and is straightforward to use, since it's supported directly by the programming language. Class inheritance makes it easier to modify the implementation being reused.
- Cons: You can not change the implementations being inherited at run-time. Inheritance exposes as subclass to details of its parent's implementation. Any change in the parent's implementation will force the subclass to change. One cure is to only inherit from abstract classes since they provide little or no implementation.

## Pros and Cons of Composition

- Composition is defined at run-time through objects acquiring references to other objects.
- Composition requires objects to respect each other's interface. Because objects are accessed solely through their interfaces we don't break encapsulation. Any object can be replaced at run-time by another as long as it has the same type.
- Because an object's implementation is written in terms of object interfaces, there are substantially fewer implementation dependencies.

## Inheritance vs. Object Comp.

- Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task.
- Classes and class hierarchies remain small and manageable.
- A design based on object composition has more objects (if fewer classes) and the system behavior depends on their interrelationships instead of being defined in one class.
- *Favor object composition over class inheritance.*

## Composite Pattern

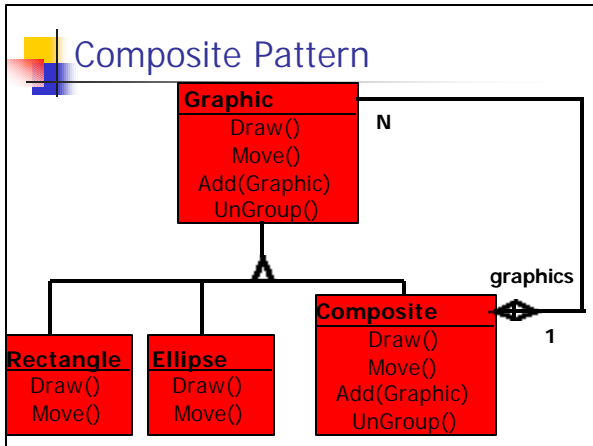
- Intent: Compose Objects into tree structures to represent part-whole hierarchies. Composite let clients treat individual objects and compositions of objects uniformly.
- Motivation: Graphics applications like drawing editors let users build complex diagrams out of simple components. The user can group components to form larger components, which in turn can be grouped to form still larger components.

## Composite Pattern

- A simple implementation could define classes for graphical primitives such as Rectangles and Ellipses plus other classes that act as containers for these primitives.
- The problem is that code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically.
- Having to distinguish these objects makes the application more complex. The Composite pattern describes how to use recursive composition so that clients don't have to make that distinction.

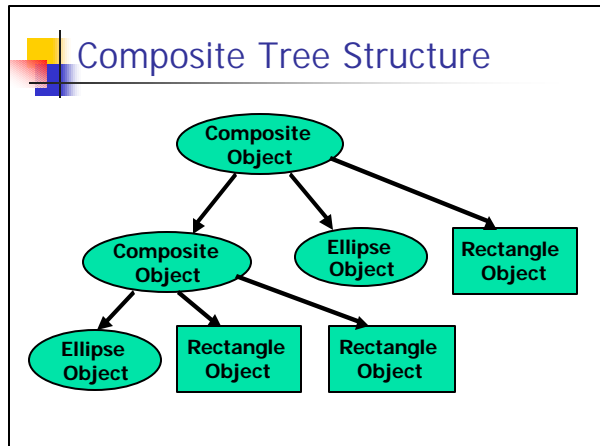
## Composite Pattern

- The key to the Composite pattern is an abstract class that represents both primitives and their containers.
- For the graphic system, the abstract class Graphic declares operations like Draw, Move, Delete that are specific to graphical objects.
- It also declares operations that all composite objects share, such as operations for accessing and managing its children, like Add, UnGroup.



- ### Composite Pattern
- The subclasses Ellipse, Rectangle etc. Define primitive graphic objects.
  - These classes implement Draw to draw rectangles, ellipses respectively.
  - Since primitive graphics have no child graphics, none of these subclasses implements child related operations such as Add or UnGroup.

- ### Composite Pattern
- The Composite class defines an aggregate of Graphic objects.
  - Composite implements Draw to call Draw on its children, and it implements child-related operations accordingly.
  - Because the Composite interface conforms to the Graphic interface, Composite can compose other Composites recursively.



## Abstract Base Class Graphic

```
class Graphic
{
public:
    virtual void Draw(Window* w)=0; // pure virtual
    virtual void Move(int dx, int dy)=0;
    virtual void Add(Graphic* g)=0;
    virtual list<Graphic*> Ungroup()=0;
    virtual void Delete()=0;
};
```

## Subclass Composite

```
class Composite : public Graphic
{
private:
    list<Graphic*> graphics;
public:
    virtual void Draw(Window* w);
    virtual void Move(int dx, int dy);
    virtual void Add(Graphic* g);
    virtual list<Graphic*> Ungroup();
    virtual void Delete();
};
```

## Subclass Composite

```
virtual void Composite::Draw(Window* w)
{
    for (list<Graphic*>::iterator iter=graphics.begin();
         iter!=graphics.end(); iter++)
        (*iter)->Draw(w); // delegate Draw() to children
}

virtual void Composite::Add(Graphic* g)
{
    graphics.push_back(g); // add graphic to list
}
```

## Subclass Rectangle

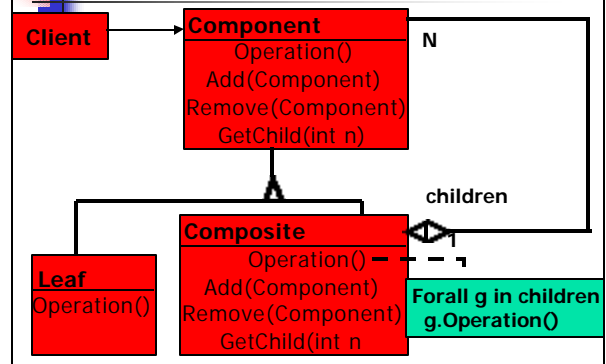
```
class Rectangle : public Graphic
{
private:
    int x, y;
    int width, height;
public:
    virtual void Draw(Window* w);
    virtual void Move(int dx, int dy);
    virtual void Add(Graphic* g);
    virtual list<Graphic*> Ungroup();
    virtual void Delete();
    class GraphicError{}; // exception class for Graphic
};
```

## Subclass Rectangle

```

virtual void Rectangle::Draw(Window* w)
{
    w->draw_rectangle(...); // call graphic routine
}
virtual void Rectangle::Move(int dx, int dy)
{
    x+=dx; // move relative by displacement (dx,dy)
    y+=dy;
}
virtual void Rectangle::Add(Graphic* g)
{
    throw GraphicError(); // throw an exception
}
    
```

## Composite Pattern General

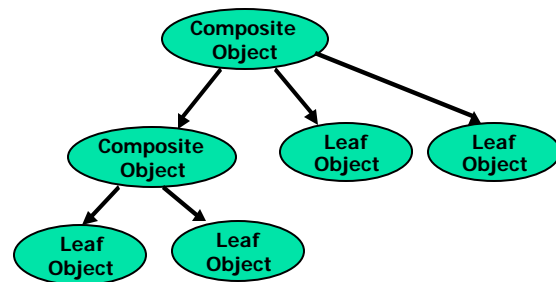


## Applicability of Composite

Use the Composite pattern when

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

## Composite Tree Structure





## Participants of Composite

- Component
  - Declares the interface for objects in the composition
  - Implements default behavior for the interface common to all classes, as appropriate
  - Declares an interface for accessing and managing its child components
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that is appropriate



## Participants of Composite

- Leaf
  - Represents leaf objects in the composition. A leaf has no children.
  - Defines behavior for primitive objects in the composition



## Participants of Composite

- Composite
  - Defines behavior for components having children
  - Stores child components
  - Implements child-related operations in the Component interface
- Client
  - Manipulates objects in the composition through the Component interface



## Collaborations of Composite

- Client uses the Component class interface to interact with objects in the composite structure.
- If the recipient is a Leaf, then the request is handled directly.
- If the recipient is a Composite, then it is usually forwards requests to its child components, possibly performing additional operations before or/and after forwarding.

## Consequences of Composite

- Makes the client simple: Clients can treat composite structures and individual objects uniformly. Clients normally don't know and should not care whether they are dealing with a leaf or composite component.
- Makes it easier to add new kinds of components. Newly defined Composite or Leaf classes work automatically with existing structures and client code.
- Can make your design overly general. The disadvantage of making it easy to add new components is that it makes it hard to restrict the components of a composite.

## Implementation Issues

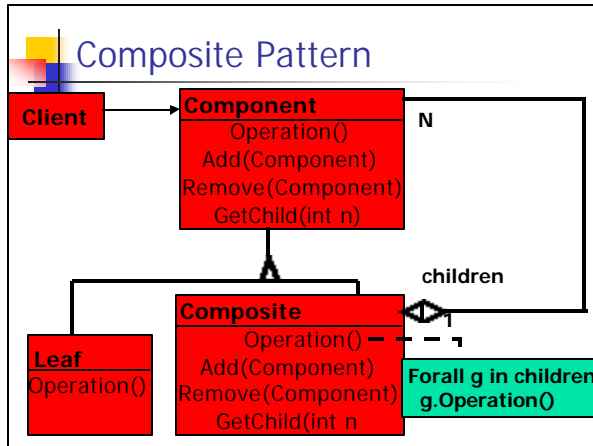
- Explicit parent references
  - Maintaining references from child components to their parent can simplify the traversal and management of a composite structure.
  - The usual place to define the parent reference is in the Component class.
  - Leaf and Composite can inherit the reference and the operations that manage it.
  - It is necessary to maintain the invariant that children of a composite have as their parent the composite that in turn has them as children.

## Implementation Issues

```
class Component
{
protected:
    Component* parent;
    ...
};
void Composite::Add(Component* c)
{
    c->parent=this;
    ...
}
```

## Implementation Issues

- Declaring the child management operations
  - Defining the child management interface at the root of the class hierarchy gives you transparency, because you can treat all components uniformly. It costs you safety, however, because clients may try to do meaningless things like add objects to leaves.
  - Defining child management in the Composite class gives you safety because any attempt to add objects to leaves will be caught at compile-time. But you lose transparency, because leaves and composites have different interfaces.



### Child Management in Base Class

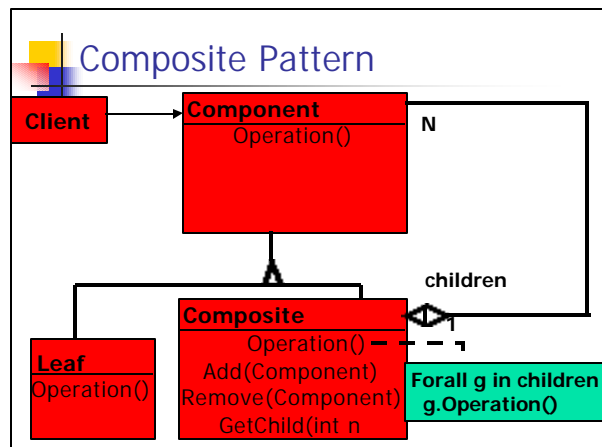
```

class Component
{
public:
    virtual void Operation=0; // pure virtual
    virtual void Add(Component* c)=0;
    // child management functions declared in base class
};
class Composite : public Component
{
public:
    virtual void Operation();
    virtual void Add(Component* c); // implements Add()
};
  
```

### Safety vs. Transparency

```

vector<Component*> components;
components.push_back(new Leaf());
components.push_back(new Leaf());
components.push_back(new Composite());
components[2]->Add(components[0]); // transparent
// works because Add() is member function of
// base class Component
components[1]->Add(components[0]); // unsafe
// fails as it tries to add a Leaf to a Leaf ,
// possible solution throw an exception for Leaf::Add()
  
```



## Child Management in Subclass

```
class Component
{
public:
    virtual void Operation=0; // pure virtual
};
class Composite : public Component
{
public:
    virtual void Operation();
    void Add(Component* c);
    // child management functions declared in subclass
};
```

## Safety vs. Transparency

```
vector<Component*> components;
components.push_back(new Leaf());
components.push_back(new Leaf());
components.push_back(new Composite());
components[2]>Add(components[0]); // non-transparent
// fails because Add() is not a member function of
// base class Component
Composite* c= (Composite*) components[2];
// explicit type cast from Component* to Composite*
c->Add(components[0]);
components[1]>Add(components[0]); // safe
// will issue compiler error because Add() is not a member
// function of base class Component
```

## Observer Pattern

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Example: Graphical editor with multiple views of the same graphic objects. If a graphical object is manipulated in one view, it notifies the other views to display the changes made to it.

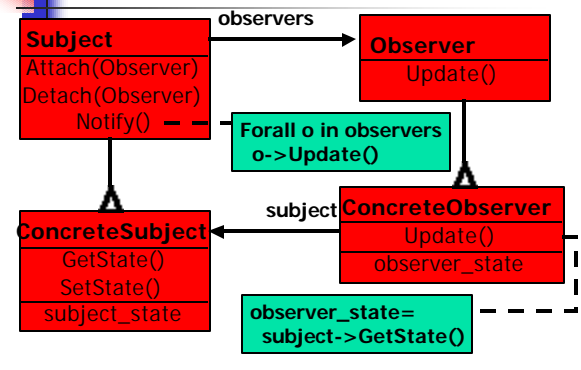
## Observer Pattern

- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
- The key objects in the Observer pattern are Subject and Observer.
- A subject may have any number of dependent observers.
- All observers are notified whenever the subject undergoes a change in state.
- In response, each observer will query the subject to synchronize its state with the subject's state.

## Observer Applicability

- Use the Observer pattern in any of the following situations
  - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - When a change to one object requires changing others, and you do not know how many objects need to be changed.
  - When an object should be able to notify other objects without making assumptions about who these objects are.

## Observer Structure



## Observer Participants

- Subject
  - Knows its observers. Any number of Observer objects may observe a subject.
  - Provides an interface for attaching and detaching Observer Objects.
- Observer
  - Defines an updating interface for objects that should be notified of changes in a subject

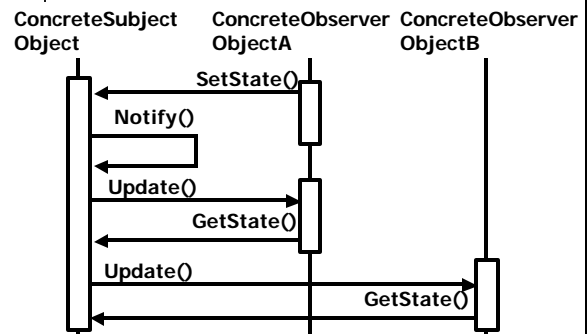
## Observer Participants

- ConcreteSubject
  - Stores a state of interest to ConcreteObserver objects.
  - Sends a notification to its observers when its state changes.
- ConcreteObserver
  - Maintains a reference to a ConcreteSubject object
  - Stores state that should stay consistent with the subject state.
  - Implements the Observer updating interface to keep its state consistent with the subject state.

## Observer Collaborations

- ConcreteSubject notifies its observers whenever a change occurs that could make its observer's state inconsistent with its own.
- After being informed of a change in the ConcreteSubject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the object.

## Observer Sequence Diagram



## Observer Sequence Diagram

- Note that the Observer object that initiates the change request with SetState() postpones its update until it gets a notification from the subject.
- In this scenario Notify() is called by the subject, but it can be called by an observer or by another kind of object (see implementation issues)

## Observer Consequences

- The Observer pattern lets you vary subjects and observers independently. You can reuse subjects without reusing observers, and vice versa. It lets you add observers without modifying the subject or other observers.



## Observer Consequences

- Abstract coupling between Subject and Object
  - All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class. The subject does not know the concrete class of any observer.
- Support for broadcast communication
  - Unlike an ordinary request, the notification that a subject sends need not specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it.



## Observer Consequences

- Unexpected Updates
  - Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocuous operation to the subject may cause a cascade of updates to observers and their dependent objects.
  - This problem is aggravated by the fact that the simple update protocol provides no details on *what* changed in the subject. Without additional protocol observers have to query the entire state of the subject to deduce the changes.



## Observer Implementation

- Mapping subjects to their observers
  - The simplest way for a subject to keep track of the observers it should notify is to store references to them explicitly in the subject. An alternative is to use an associative look-up (multimap) to maintain the subject-observer mapping.



## Observer Implementation

- Observing more than one subject
  - It might make sense in some situations for an observer to depend on more than one subject. It is necessary to extend the Update interface in such cases to let the observer know *which* subject is sending the notification. The subject can simply pass itself as a parameter in the Update operation, thereby letting the observer know which subject to examine.



## Observer Implementation

- Who triggers the update
  - The subject and its observers rely on the notification mechanism to stay consistent. But what object actually calls Notify() to trigger the update? There are two options.
  - Have state-setting operations on the Subject call Notify after they change the subject's state
  - Make clients responsible for calling Notify at the right time.



## Observer Implementation

- Subject calls Notify
  - The advantage of this approach is that clients do not have to remember to call Notify on the subject. The disadvantage is that several consecutive operations will cause several consecutive updates, which may be inefficient
- Clients calls Notify
  - The advantage here is that the client can wait to trigger the update until after a series of state changes has been made. The disadvantage is that clients have an added responsibility to trigger the update, causing possible errors.