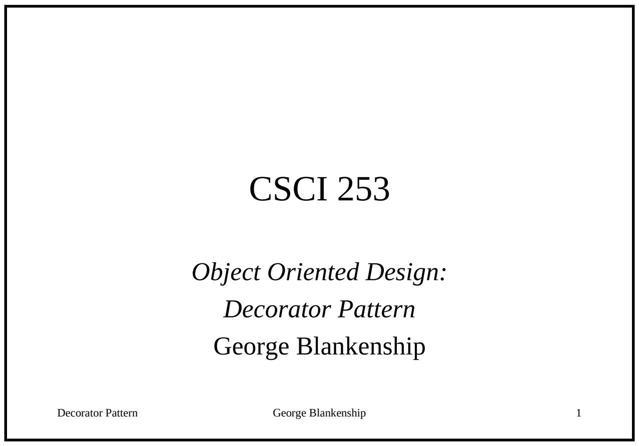
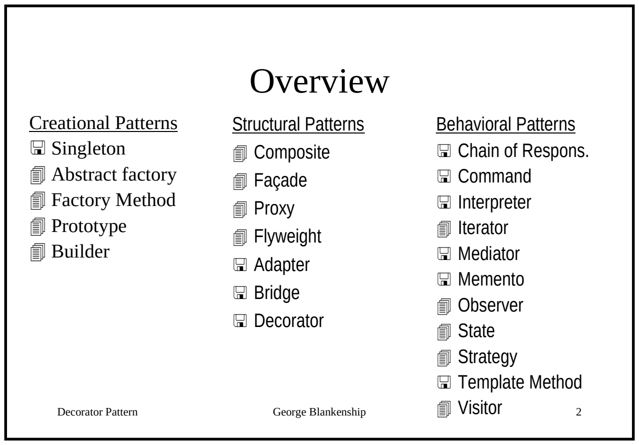


CSCI 253

Object Oriented Design:

Decorator Pattern

George Blankenship

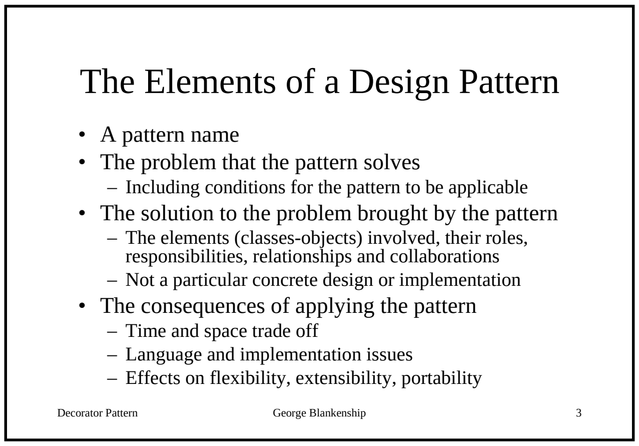


Overview

Creational Patterns
- 🖫 Singleton
- 🗐 Abstract factory
- 🗐 Factory Method
- 🗐 Prototype
- 🗐 Builder

Structural Patterns
- 🗐 Composite
- 🗐 Façade
- 🗐 Proxy
- 🗐 Flyweight
- 🖫 Adapter
- 🖫 Bridge
- 🖫 Decorator

Behavioral Patterns
- 🖫 Chain of Respons.
- 🖫 Command
- 🖫 Interpreter
- 🗐 Iterator
- 🖫 Mediator
- 🖫 Memento
- 🗐 Observer
- 🗐 State
- 🗐 Strategy
- 🖫 Template Method
- 🗐 Visitor

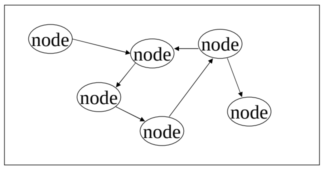


The Elements of a Design Pattern

- A pattern name
- The problem that the pattern solves
  - Including conditions for the pattern to be applicable
- The solution to the problem brought by the pattern
  - The elements (classes-objects) involved, their roles, responsibilities, relationships and collaborations
  - Not a particular concrete design or implementation
- The consequences of applying the pattern
  - Time and space trade off
  - Language and implementation issues
  - Effects on flexibility, extensibility, portability

The Decorator Pattern: The Problem

You want to add responsibilities to individual objects dynamically and transparently without affecting other objects



-Discrete simulator (FSM) describes simulation space as a map

-Each node of the map is a place (processing moves from place to place)

-Type of node defines processing

Also known as : wrapper

Node

- A node of the simulation map
- Nodes are linked together to define paths that an object might follow
- Display() display information about the node
- Enter(object) place object on enter queue
- Exit(object) place object on exit queue
- Visit(object) place object on visit queue

GenerateVisitor

- The generator node creates objects that visit other nodes
- public GenerateVisitor(String n, int i, NodeRoute r, QueueDescriptor q)
- public void display()
- public boolean epilogue(int time) // create new objects
- public void destroy()

Replicate

- The replicator node duplicate objects that visit other nodes
- public Replicate(String n, int i, NodeRoute r, QueueDescriptor q)
- public void display()
- public boolean processObject(MobileObject template, int time) // clone objects
- public void destroy()

Decorator Pattern George Blankenship 7

Route

- The route change node changes the route of an object
- public Route(String n, int i, NodeRoute r, QueueDescriptor q)
- public void display()
- public boolean processObject(MobileObject o, int time) // change the route
- public void destroy()

Decorator Pattern George Blankenship 8

StateMachine States

- currentState =
  – new StateVariable(getName()+" current state");
- initialState =
  – new StateVariable(getName()+" initial state");
- endState =
  – new StateVariable(getName()+" end state");
- errorState =
  – new StateVariable(getName()+" error state");

Decorator Pattern George Blankenship 9

EchoConnection

- public void completed()
  - Process ending states (normal/error)
  - Close connection
- public boolean execute()
  - Execute state machine (state & input)
  - processInitialState()
  - processWaitDSRState()
  - processWaitCTSState()
  - processWaitXONState()
  - processWaitInputState()

Decorator Pattern George Blankenship 10

DialConnection

- public void completed()
  - Process ending states (normal/error)
  - Close connection
- public boolean execute()
  - Execute state machine (state & input)
  - processInitialState()
  - processWaitDSRState()
  - processWaitCTSState()
  - processWaitXONState()
  - processResetStringSendState()
  - processResetStringSentState()
  - processInitStringSendState()
  - processInitStringSentState()
  - processDialStringSendState()
  - processDialStringSentState()

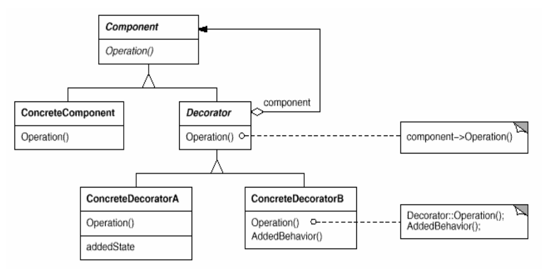
Decorator Pattern George Blankenship 11

AnswerConnection

- public void completed()
  - Process ending states (normal/error)
  - Close connection
- public boolean execute()
  - Execute state machine (state & input)
  - processInitialState()
  - processWaitDSRState()
  - processWaitCTSState()
  - processWaitXONState()
  - processResetStringSendState()
  - processResetStringSentState()
  - processInitStringSendState()
  - processInitStringSentState()
  - processWaitCDState()
  - processWaitInputState()
  - processEndStringSendState()
  - processEndStringSentState()

Decorator Pattern George Blankenship 12

The Decorator Pattern: Structure

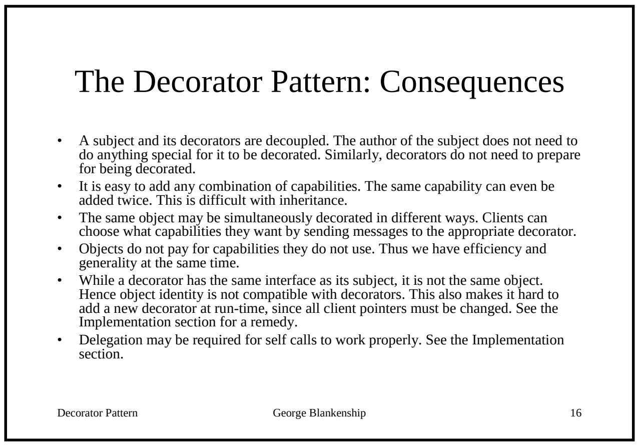


The Decorator Pattern: Participants

- Component - Each component can be used on its own or wrapped by a decorator component.
- ConcrecteComponent - the object we are going to dynamically add new behavior to. It extends the Component.
- Decorator - Each decorator HAS_A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.
- ConcreteDecorator - The ConcreteDecorator has an instance variable for the thing it decorates (the Component the Decorator wraps)

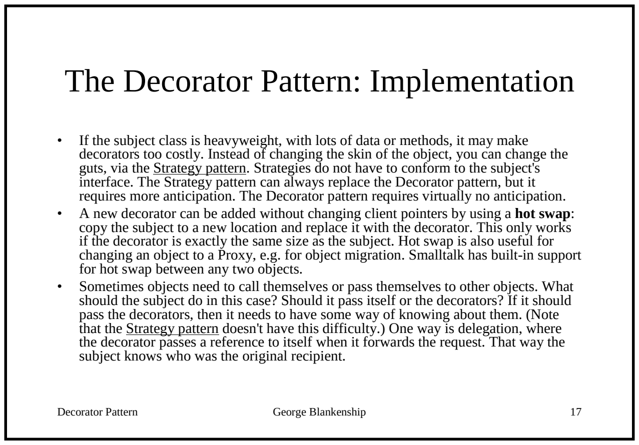
The Decorator Pattern: Collaboration

- Enclose the subject in another object, the decorator object, which conforms to the same interface. This makes the decorator transparent to clients.
- The decorator forwards requests to the subject while performing additional actions before and after forwarding.



The Decorator Pattern: Consequences

- A subject and its decorators are decoupled. The author of the subject does not need to do anything special for it to be decorated. Similarly, decorators do not need to prepare for being decorated.
- It is easy to add any combination of capabilities. The same capability can even be added twice. This is difficult with inheritance.
- The same object may be simultaneously decorated in different ways. Clients can choose what capabilities they want by sending messages to the appropriate decorator.
- Objects do not pay for capabilities they do not use. Thus we have efficiency and generality at the same time.
- While a decorator has the same interface as its subject, it is not the same object. Hence object identity is not compatible with decorators. This also makes it hard to add a new decorator at run-time, since all client pointers must be changed. See the Implementation section for a remedy.
- Delegation may be required for self calls to work properly. See the Implementation section.



The Decorator Pattern: Implementation

- If the subject class is heavyweight, with lots of data or methods, it may make decorators too costly. Instead of changing the skin of the object, you can change the guts, via the Strategy pattern. Strategies do not have to conform to the subject's interface. The Strategy pattern can always replace the Decorator pattern, but it requires more anticipation. The Decorator pattern requires virtually no anticipation.
- A new decorator can be added without changing client pointers by using a **hot swap**: copy the subject to a new location and replace it with the decorator. This only works if the decorator is exactly the same size as the subject. Hot swap is also useful for changing an object to a Proxy, e.g. for object migration. Smalltalk has built-in support for hot swap between any two objects.
- Sometimes objects need to call themselves or pass themselves to other objects. What should the subject do in this case? Should it pass itself or the decorators? If it should pass the decorators, then it needs to have some way of knowing about them. (Note that the Strategy pattern doesn't have this difficulty.) One way is delegation, where the decorator passes a reference to itself when it forwards the request. That way the subject knows who was the original recipient.