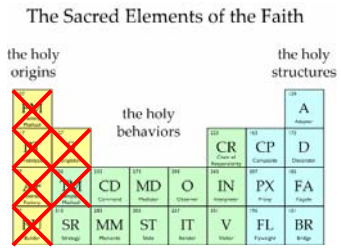
ECE450 – Software Engineering II

Today: **Design Patterns IV**
Structural Patterns

Changing gears: Structural patterns

- So far we have discussed almost exclusively creational patterns
  - Factory method
  - Abstract factory
  - Builder
  - Prototype
  - Singleton
- ...and a couple of behavioral patterns
  - Template method
  - Null object
- We'll now discuss *structural patterns*, that is, those that solve design problems through particular arrangements of class structures

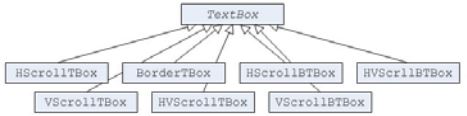


Adding behaviour to a text box...

- Our application uses a TextBox class that displays texts in a box
- Sometimes these text boxes need additions:
  - Borders (of different kinds)
  - Vertical scrollbars
  - Horizontal scrollbars
  - ...
- How can we design the classes of the application so that they address this problem?

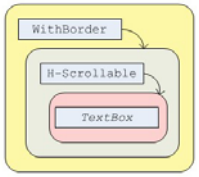
Just subclass everything!

- We can simply subclass everything
  - But given enough added behaviors, this solution will go nuts quickly...



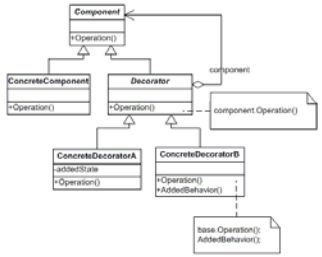
“Decorate” the original TextBox

- A more flexible approach is to enclose the component in another object that adds behaviour/functionality
  - Think of it as “wrapping” the original object
    - The “wrapped” object does what the original did, but has additional features
  - This “wrapper”, or **decorator**, conforms to the interface of the component it decorates
    - Its presence is transparent to the component’s clients
  - We can nest decorators recursively, allowing a potentially unlimited number of added responsibilities



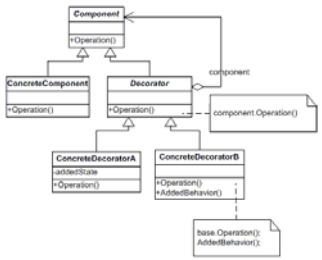
The *Decorator* pattern

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



Participants

- Component
  - Defines the interface for objects that can have responsibilities added to them dynamically
- ConcreteComponent
  - Defines an object to which additional responsibilities can be attached
- Decorator
  - Maintains a reference to a Component object and defines an interface that conforms to Component’s interface
- ConcreteDecorator
  - Adds responsibilities to the component



Applicability

- Use Decorator…
  - To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects
  - For responsibilities that can be withdrawn
  - When extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination.
- How does it work?
  - Decorator forwards requests to its Component object.
  - It may optionally perform additional operations before and after forwarding the request.

Consequences

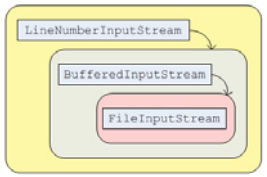
- (+) More flexibility than static inheritance
  - Responsibilities can be added and removed at run-time simply by attaching and detaching them.
  - It's also easy to add a property twice
- (+) Avoids feature-laden classes high up in the hierarchy
  - Offers a pay-as-you-go approach to adding responsibilities
  - Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects.
- (-) A decorator and its component aren't identical
  - From an object identity point of view, a decorated component is not identical to the component itself
- (-) Lots of little objects
  - Systems composed of lots of little objects that all look alike
  - Easy to customize, but hard to learn and debug

Implementation

- Interface conformance
  - A decorator object's interface must conform to the interface of the object it decorates
- Omitting the abstract Decorator class
  - There's no need to define an abstract Decorator class when you only need to add one responsibility.
- Keeping Component classes lightweight
  - To ensure a conforming interface, components and decorators must descend from a common Component class
  - It's important to keep this class lightweight
    - Definition of data representation should be deferred to subclasses
    - Otherwise the complexity of the Component class might make the decorators too heavyweight to use in quantity
- Changing the skin of an object vs. changing its guts
  - Think of a decorator as a skin over an object that changes its behaviour
  - If you want to change the guts instead you can use *Strategy*
    - We'll check it out later…

Java I/O streams are Decorators!

- There is an abstract Component…
  - *InputStream*
- …a few concrete Components…
  - FileInputStream
  - StringBufferInputStream
  - ByteArrayInputStream
  - …
- …an abstract Decorator
  - FilterInputStream
- …and a bunch of concrete Decorators
  - PushbackInputStream
  - BufferedInputStream
  - DataInputStream
  - LineNumberInputStream
  - …



Moving on: Incompatibility problems

- We arrive at Europe with our laptop's battery almost exhausted. We want to recharge the battery and…

Moving on: Incompatibility problems

- ...but you're smart engineers and knew you were going to deal with this...



Incompatibility problems: They happen with software too

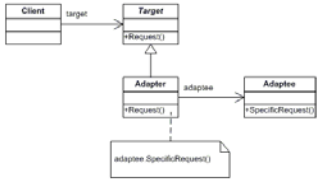
- We may have an application that needs to use libraries/a different application/you-name-it, but the thing we want to call has a different interface than our caller
  - Alternative 1: Re-write the caller
    - Ugly, messy, error-prone
    - Equivalent to changing the power cable in our previous example
  - Alternative 2: Re-write the called libraries/classes
    - May not have the source code
    - As ugly and error-prone as Alternative 1
  - Alternative 3: Write an adapter
    - The adapter converts all requests to a language the adaptee understands

The *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*
- Applicability: Use when...
  - You want to use an 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

Structure

- Target
  - Defines the domain-specific interface that Client uses.
- Client
  - Collaborates with objects conforming to the Target interface
- Adaptee
  - Defines an existing interface that needs adapting
- Adapter
  - Adapts the interface of Adaptee to the Target interface

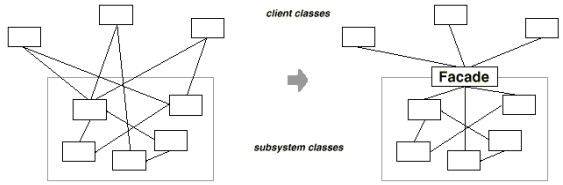


Consequences and Implementation

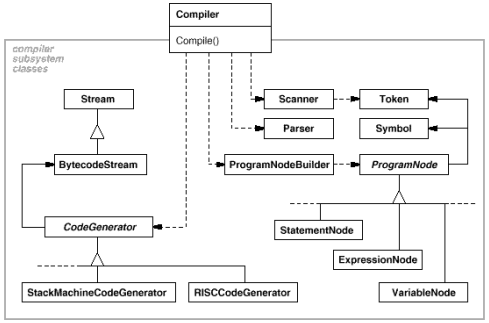
- It's a very straightforward pattern
  - Makes things work after they are designed
- Amount of work it takes to implement Adapter depends on complexity of interface to adapt
- Comparison with Decorator
  - Adapter is meant to change the interface of an existing object
  - Decorator enhances another object without changing its interface
  - Decorator is thus more transparent to the application than Adapter is
  - Decorator supports recursive composition, which isn't possible with pure Adapters

Facade pattern

- *Provide a unified interface to a set of interfaces in a subsystem*
  - Defines a higher-level interface that makes the subsystem easier to use
- Structuring a system into subsystems helps reduce complexity
  - A common design goal is to minimize the communication and dependencies between subsystems:



Example of a *Facade*

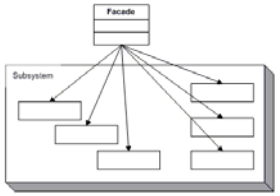


Applicability

- Use it when…
  - You want to provide a simple interface to a complex subsystem
    - Subsystems often get more complex as they evolve
    - A facade can provide a simple default view of the subsystem that is good enough for most clients
    - Only clients needing more customizability will need to look beyond the facade
  - There are many dependencies between clients and the implementation classes of an abstraction
    - A facade decouples the subsystem from clients and other subsystems, promoting subsystem independence and portability
  - You want to layer your subsystems
    - Use a facade to define an entry point to each subsystem level
    - If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades

Structure

- Facade
  - Knows which subsystem classes are responsible for a request
  - Delegates client requests to appropriate subsystem objects
- Subsystem classes
  - Implement subsystem functionality
  - Handle work assigned by the Facade object
  - Have no knowledge of the facade; that is, they keep no references to it



Consequences

- Shields clients from subsystem components
  - Reduces number of objects that clients deal with
  - Makes it easier to use the subsystem
- Promotes weak coupling between the subsystem and its clients
  - Can vary the components of a subsystem without affecting clients
  - Reduces compilation dependencies
- Doesn't prevent applications from using subsystem classes if they need to
  - You can choose between ease of use and generality