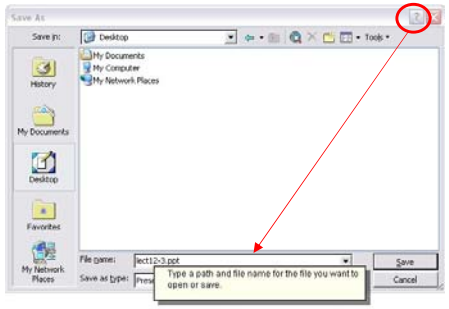
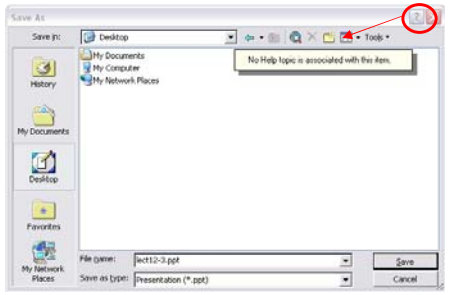
ECE450 – Software Engineering II

Today: **Design Patterns VIII**
Chain of Responsibility, Strategy, State

Implementing GUI context-sensitive help...



Implementing GUI context-sensitive help...

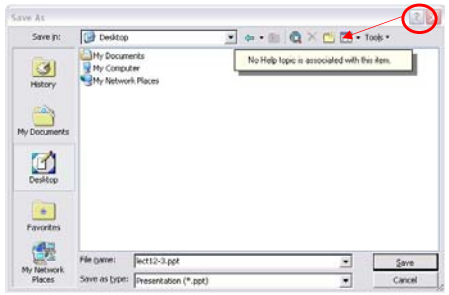


Context-sensitive help

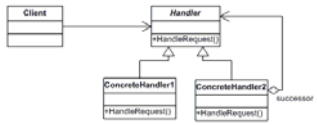
- The user can obtain help information on any part of the interface just by clicking on it
- The help that is provided depends on the part of the interface that is selected and its context
  - A button widget in a dialog box might have different help information than a similar button in the main window
  - If no specific help information exists for that part of the interface, then the help system should display a more general help message
- It's therefore natural to organize help information according to its generality –from the most specific to the most general
- The problem is that the object that ultimately *provides* the help is not known explicitly to the object that *initiates* the help request
  - We need a way to decouple the button that initiates the help request from the objects that provide help information

Chain of Responsibility

- *Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.*



Structure and participants



- Handler
  - Defines an interface for handling requests
  - (Optionally) implement the successor link
- ConcreteHandler
  - Handles requests it is responsible for
  - Can access its successor
  - If the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor
- Client
  - Initiates the request to a ConcreteHandler object on the chain

Applicability

- Use Chain of Responsibility when…
  - More than one object may handle a request, and the handler isn't known *a priori*. The handler should be ascertained automatically.
  - You want to issue a request to one of several objects without specifying the receiver explicitly
  - The set of objects that can handle a request should be specified dynamically
- Question: Is throwing exceptions the same as using the Chain of Responsibility pattern?
- Question: Can an object tell its successor exactly what operation to perform? If so, how?
  - Hint: Remember *Command*?
- Question: Would Chain of Responsibility work in conjunction with Composite?

Consequences

- Reduced coupling
  - The pattern frees an object from knowing which other object handles a request
  - As a result, instead of objects maintaining references to all candidate receivers, they keep a single reference to their successor
- Added flexibility in assigning responsibilities to objects
  - You can add or change responsibilities for handling a request by changing the chain at run-time
- Receipt isn't guaranteed
  - Since a request has no explicit receiver, there's no *guarantee* it will be handled – the request can fall off the end of the chain without ever being handled!

Implementing Sort()

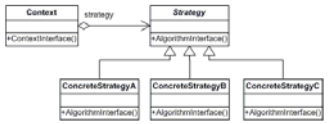
- Often we need to implement a feature, but:
  - There are several algorithms that could do the job, and we want to defer the decision of which to use…
  - …or maybe we want to use a different algorithm depending on the characteristics of the object that will run it
- Think of Sort()
  - Given the characteristics of your data structure, a sorting algorithm may be more convenient than another
  - …so it would be good if we avoid getting stuck with one algorithm in particular

The *Strategy* pattern

- *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.*

Structure and Participants

- Strategy
  - Declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy
- ConcreteStrategy
  - Implements the algorithm using the Strategy interface
- Context
  - Is configured with a ConcreteStrategy object
  - Maintains a reference to a Strategy object
  - May define an interface that lets Strategy access its data



Applicability

- Use the Strategy pattern when…
  - Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors
  - You need different variants of an algorithm
  - A class defines many behaviors, and these appear as multiple conditional statements in its operations
    - Instead of many conditionals, move related conditional branches into their own Strategy class
  - Remember that Decorator modifies classes' behaviors by changing their **skin**, while Strategy modifies their behaviors by changing their **guts**.

Consequences

- Strategies eliminate conditional statements
  - When different behaviors are lumped into one class, it's hard to avoid using conditional statements to select the right behavior
  - Encapsulating the behavior in separate Strategy classes eliminates these conditional statements
- A choice of implementations
  - Strategies can provide different implementations of the *same* behavior
  - The client can choose among strategies with different time and space trade-offs
- Increased number of objects
  - Strategies increase the number of objects in an application
  - Sometimes you can reduce this overhead by implementing strategies as stateless objects that contexts can share

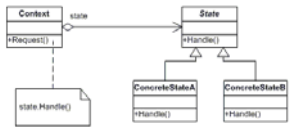
Objects may behave differently depending on their **state**

Handling states

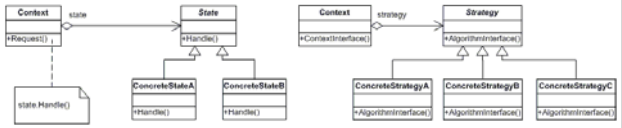
- Many programs rely heavily on tracking and changing the state of their objects
  - Network connections
  - Vending machines
  - Finite state machines
  - …
- For these programs, the **state** of the object affects their behavior
  - E.g., a vending machine only dispenses a product if its user has given it enough cash
  - The naive approach to handle states is to implement a lot of conditionals in one class
- Introducing the *State* pattern:
  - *Allow an object to alter its behavior when its internal state changes. The object will appear to change its class*

Participants and structure

- Context
  - Defines the interface of interest to clients
  - Maintains an instance of a ConcreteState subclass that defines the current state
- State
  - Defines an interface for encapsulating the behavior associated with a particular state of the Context
- ConcreteState subclasses
  - Each subclass implements a behavior associated with a state of the Context



Wait a minute...



State pattern **Strategy pattern**

- They are the same diagram!!
  - Yes, but the patterns differ in **intent:**
    - In the **State** pattern we have a set of behaviors encapsulated in state objects; at any time the context delegates to one of those states.
    - Over time, the current state changes across the set of state objects to reflect the internal state of the context, so the context's behavior changes over time as well.
    - In **Strategy**, the client usually specifies the strategy object that the context is composed with.
    - Although you can change Strategy objects, often there is one that is most appropriate for a context object
    - In general, think of Strategy as a flexible alternative to subclassing…
    - …while State is an alternative to putting lots of conditionals in your Context

Applicability

- Use the State pattern in either of the following cases:
  - An object's behavior depends on its state, and it must change its behavior at run-time depending on that state
  - Operations have large, multipart conditional statements that depend on the object's state.
    - This state is usually represented by one or more enumerated constants
    - Often, several operations will contain this same conditional structure
    - The State pattern puts each branch of the conditional in a separate class

Consequences

- *State* localizes state-specific behavior and partitions behavior for different states
  - Because all state-specific code lives in a State subclass, new states and transitions can be added easily by defining new subclasses
- It makes state transitions explicit
  - When an object defines its current state solely in terms of internal data values, its state transitions have no explicit representation
    - They only show up as assignments to some variables
  - Introducing separate objects for different states makes the transitions more explicit