Object-Oriented Programming
Conclusions

CSIE Department, NTUT
Woei-Kae Chen
Discussion of Behavior Patterns

- **Encapsulating Variation (with inheritance; p345)**
  - encapsulate an aspect that changes frequently
    - new object: encapsulates the aspect
    - existing object: use the new object
  - **Example**
    - Strategy $\rightarrow$ algorithm
    - State $\rightarrow$ state-dependent behavior
    - Iterator $\rightarrow$ traversal
    - Abstract Factory, Builder $\rightarrow$ object creation
    - Command $\rightarrow$ command
    - Visitor $\rightarrow$ operation
  - **Advantage**
    - new objects can be changed at run-time
    - vary new and existing objects independently
    - easy to add new objects
    - easy to maintain and reuse
  - **Disadvantage**
    - have to open interfaces for new objects to access existing objects
Discussion of Behavior Patterns

- **Encapsulating Variation (without inheritance)**
  - define an **object** to encapsulate an aspect that **changes frequently**
    - new object: encapsulates the aspect
    - existing object: use the new object
  - Example
    - Mediator encapsulates the protocol between objects
    - Prototype encapsulates object creation
    - Decorator encapsulates responsibility added to an object
    - Bridge encapsulates an abstraction from its implementation
Discussion of Behavior Patterns

- Object as arguments
  - Visitor
    - is the argument to the `accept` operation
  - Command
    - act as tokens to be passed around and invoked later
    - the token represents a request
  - Memento
    - act as tokens to be passed around and invoked later
    - the token represents an internal state at a particular time
Discussion of Behavior Patterns

- Should communication be encapsulated or distributed?
  - Mediator vs. Observer
    - competing patterns
  - Mediator
    - encapsulates communication between objects
    - centralizes communications
    - maintains a communication constraint in the mediator
    - difficult to make reusable Mediators
    - easier to understand the flow of communication
  - Observer
    - distributes communication by introducing Observer and Subject objects
    - observer and subject cooperate to maintain a constraint
    - easier to make reusable observers and subjects
    - difficult to understand the flow of communication
Discussion of Behavior Patterns

- **Decoupling Senders and Receivers (1)**
  - Related patterns
    - Command
    - Observer
    - Mediator
    - Chain of Responsibility
  - Command
    - decouples invokers and receivers by command objects
    - allow senders to work with different receivers
    - a subclass for a sender-receiver connection

```
anInvoker (sender)        aCommand        aReceiver (receiver)
                     |                     |
                     v                     v
  Execute()            Action()             `-->
```


Discussion of Behavior Patterns

- Decoupling Senders and Receivers (2)
  - Observer
  - decouples subjects and observers by an interface for signaling changes
  - a subject may have multiple observers
  - the number of observers can vary at run-time
  - best for decoupling objects with data dependencies

```
<table>
<thead>
<tr>
<th>aSubject (sender)</th>
<th>anObserver (receiver)</th>
<th>anObserver (receiver)</th>
<th>anObserver (receiver)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update()</td>
<td>Update()</td>
<td>Update()</td>
<td>Update()</td>
</tr>
</tbody>
</table>
```

Update()
Discussion of Behavior Patterns

- Decoupling Senders and Receivers (3)
  - Mediator
    - decouple objects (Colleagues) through a Mediator object
    - routes requests between Colleague objects
    - centralizes communication between Colleague objects
Discussion of Behavior Patterns

- Decoupling Senders and Receivers (4)
  - Chain of Responsibility
    - decouple client and handler by passing the request along a chain
    - when the chain is part of the system’s structure
    - when one of several objects may be in a position to handle the request

```
aClient (sender)
```
```
aHandler (receiver)
HandleHelp()
aHandler (receiver)
HandleHelp()
aHandler (receiver)
HandleHelp()
aHandler (receiver)
```
Principle of OO Design

- Program to an interface, not an implementation
  - commit only to an abstract class interface
  - how to instantiate concrete classes?
    - Abstract Factory, Builder, Factory Method, Prototype, and Singleton.

- Favor object composition over class inheritance
  - subclassing: white-box reuse
    - inheritance breaks encapsulation
    - change parent classes → change subclasses
    - unmanageable monster classes and class hierarchies
  - object composition: black-box reuse
    - requires carefully designed interface
    - keep each class encapsulated
    - classes and class hierarchies remain small
Design patterns: Conclusion

- Documents existing designs
  - used by expert object-oriented designers
- What to expect from design patterns (p351)
  - A Common Design Vocabulary
    - to communicate, document, and explore design alternatives
    - make a system seem less complex talking about it in a higher level
  - A Documentation and Learning Aid
    - most large OO system use these design patterns
    - describing a system in terms of the design patterns make it a lot easier to understand (e.g., patterns generate architectures)
    - make you a better designer
Designing for Change (1)

- Design patterns help to ensure that a system can change in specific ways \( \rightarrow \) easier to change.
- Common cause of redesign (p24)
  - Creating an object by specifying a class explicitly
    - complicates future changes
    - design patterns: Abstract Factory, Factory Method, Prototype
  - Dependence on specific operations
    - avoid hard-coded requests
    - design patterns: Chain of Responsibility, Command
  - Dependence on hardware and software platform
    - harder to port to other platforms
    - harder to keep it up to date on its native platform
    - design patterns: Abstract Factory, Bridge
Designing for Change (2)

- Dependence on object representation or implementation
  - knows how objects are represented, stored, or implemented → when object changes → client must change
  - design patterns: Abstract Factory, Bridge, Memento, Proxy

- Algorithmic dependencies
  - algorithm change → object change → algorithm that are likely to change should be isolated
  - design patterns: Builder, Iterator, Strategy, Template Method, Visitor

- Tight coupling
  - tightly coupled classes are hard to reuse, learn, port, and maintain
  - design patterns use abstract coupling and layering to promote loose coupling
  - design patterns: Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer
Designing for Change (3)

- **Extending functionality by subclassing**
  - subclassing can lead to explosion of subclasses
  - object composition and delegation provide flexible alternative to inheritance for combining behavior
  - design patterns: Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy

- **Inability to alter classes conveniently**
  - modifying a class that cannot be modified conveniently
    - no source code
    - any change would require modifying lots of existing subclasses
  - design patterns: Adapter, Decorator, Visitor
Design patterns: Conclusion

- An Adjunct to Existing Methods
  - is an important missing piece from OO design methods
  - show how to use primitive techniques (e.g., objects, inheritance, and polymorphism)
  - provide a way to describe more of the “why” → not just record the results of decisions.
  - useful in turning an analysis model into an implementation model

- A Target for Refactoring
  - OO software lifecycle
    - prototyping phase
    - expansionary phase
    - consolidating phase

- OO software lifecycle
  - prototyping phase
  - expansionary phase
  - consolidating phase

- Prototyping to consolidation
  - more requirements
  - more reuse
Design patterns: Conclusion

- **Expansionary phase**
  - add new classes and operations
  - new requirements are satisfied, but ...
    - becomes too inflexible for further change
    - classes define many unrelated operations and instance variables

- **Consolidating phase**
  - moving operations up or down the class hierarchy
  - rationalizing the interface of classes
  - tearing apart classes into special- and general-purpose components
    - produce many new kinds of objects
    - decomposing existing objects using object composition instead of inheritance
    - black box reuse replace white box reuse.
  - software becomes more general

- **Design patterns**
  - capture many structures results from refactoring
  - provide targets for refactorings.

more requirements  more reuse
prototyping  consolidation
expansion