Topics in Object-Oriented Design Patterns

† Material mainly from the book *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides; slides originally by Spiros Mancoridis; modified by Ian Davis and Grant Weddell.
Outline

- Terminology and motivation.

- Reusable object-oriented design patterns.
  - creational patterns
  - structural patterns
  - behavior patterns
Design patterns

- Good designers know not to solve every problem from first principles. They reuse solutions.

- Practitioners do not do a good job of recording experience in software design for others to use.
Design patterns (cont’d)

- A design pattern systematically names, explains, and evaluates an important and recurring design.
- We describe a set of well-engineered design patterns that practitioners can apply when crafting their applications.
Becoming a skilled designer

We first learn the basics.
- algorithms
- data structures
- languages

Then, “programming in the medium.”
- structured programming
- modular programming
- object-oriented programming
Becoming a skilled designer (cont’d)

Ultimately, a designer needs to become familiar with the good designs and design ideas of others.

- Design *patterns* must be understood, memorized, and applied.

- There are thousands of existing design patterns.
Reusable object-oriented design patterns
Creational patterns

- Factory
- Abstract factory
- Builder
- Prototype
- Singleton
Factory: intent

- Introduces a layer of software which allows runtime creation of objects that are identified by parametric value rather than by compile time name.

- May use polymorphism to create different objects in distinct subclasses sharing common factory method.
**Factory: motivation**

- May wish to restrict conditions under which objects returned by factory may be created.
- The classes employed within software may not be known until run-time; e.g., OLE pictures etc. may be inserted into documents.
- May wish to identify classes by GUID’s, and to register them.
**Abstract factory: intent**

- Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

- Allows entire suites of classes to optionally be interchanged easily at run time.
Abstract factory: behavior

- Sometimes we have systems that support different representations depending on external factors.

- There is an abstract factory that provides an interface for the client. In this way the client can obtain a specific object through this abstract interface.
Abstract factory: example
Abstract factory: structure

AbstractFactory

CreateProductA()
CreateProductB()

ConcreteFactory1

CreateProductA()
CreateProductB()

ConcreteFactory2

CreateProductA()
CreateProductB()

AbstractProductA

ProductA1

ProductA2

AbstractProductB

ProductB1

ProductB2
Abstract factory: participants

**Abstract Factory:**
- Declares an interface for operations that create abstract product objects.

**Concrete Factory:**
- Implements the operations to create concrete product objects.
Abstract factory: participants (cont’d)

Abstract Product:
- Declares an interface for a type of product object.

Concrete Product:
- Defines a product object to be declared by the corresponding concrete factory. (Implements the Abstract Product interface).

Client:
- Uses only interfaces declared by Abstract Factory and Abstract Product classes.
**Builder**

- Methods of a builder class provide an interface which allows the builder class to receive in a pre-determined manner the material from which something is to be built, produced or used.

- The builder also provides a method to return the result of construction if appropriate.
**Builder: motivation**

- Want to divorce the issue of what is used to build a result from how the result is built.
- May want to build many different things from the same input.
- Easy to create and introduce into system new builder classes, if they all have the same external behaviour.
- Essentially, the builder pattern reinforces the importance of polymorphism.
**Builder: example**

- SGML parser identifies types of data token traversed within document.

- Want to tell application what has been seen while traversing document.

- Don’t want to tell application what to do with this data. Application may format this data, build electronic indices from it, etc.

- May want to view document in many concurrent windows and styles.
Prototype

- Don’t want to explicitly remember the knowledge of exactly what class, and with what parameters a class should be created when requested.
- Instead encapsulate this information by holding a dummy instance of the class to be created following a given request.
- Clone copies of this dummy instance as needed.
Prototype: examples

- Visio has vast numbers of visual symbols stored in template sheets.
- The symbols can be identified by GUID.
- The GUID allows the class supporting the representation of that symbol to be loaded.
- Once loaded the visual symbol can be cloned multiple times.
- It may appear multiple times in a drawing.
**Singleton**

- Provides global access to single instances of a class transparently.
- Centralizes access to this single instance.
- Embeds appropriate concurrency control within the singleton pattern.
- Avoids potential misuse of global objects.
Singleton: example

- Threads may wish to emit error messages.
- How error messages are handled is not the threads problem.
- May want to direct error messages to a single shared error message area.
- Need to avoid conflict when multiple threads concurrently report errors.
- Easy to change so that each thread has their own error message area if desired.
Structural patterns

- Adapter
- Facade
- Composite
**Adapter**

**Intent:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

**Motivation:** When we want to reuse classes in an application that expects a different interface, we do not want to (and often cannot) change the reusable classes to suit our application.
Adapter: example

```
Editor -> Shape
  BoundingBox()
  CreateManipulator()

Shape -> TextView
  GetExtent()

LineShape
  BoundingBox()
  CreateManipulator()

TextShape
  BoundingBox()
  CreateManipulator()

return text -> GetExtent()

return new TextManipulator
```
Adapter: structure with multiple inheritance

Client \rightarrow \text{Target} \begin{align*}
& \text{Request()} \\
\end{align*}

\text{Target} \rightarrow \text{Adaptee} \begin{align*}
& \text{SpecificRequest()} \\
\end{align*}

\text{Adaptee} \rightarrow \text{Adapter} \begin{align*}
& \text{Request()} \\
& \text{SpecificRequest()} \\
\end{align*}

(implementation)
*Adapter*: structure using object composition
**Adapter: participants**

**Target:** Defines the application-specific interface that clients use.

**Client:** Collaborates with objects conforming to the target interface.

**Adaptee:** Defines an existing interface that needs adapting.

**Adapter:** Adapts the interface of the adaptee to the target interface.
Facade: intent

- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Facade: motivation

- Structuring a system into subsystems helps reduce complexity.
- A common design goal is to minimize the communication and dependencies between subsystems.
- Use a facade object to provide a single simplified interface to the more general facilities of a subsystem.
Facade: example
Facade: structure
Facade: participants

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.
Composite: intent

- Compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.
Composite: motivation

If the composite pattern is not used, client code must treat primitive and container classes differently, making the application more complex than is necessary.
**Composite**: example
Composite: structure

Client

Component

Operation()
Add(Component)
Remove(Component)
GetChild(int)

Leaf

Operation()

Composite

Operation()
Add(Component)
Remove(Component)
GetChild(int)

forall g in children
  g.Operation()
Composite: participants

Component:

- Declares the interface for objects in the composition.
- Implements default behavior for the interface common to all classes.
- Declares an interface for accessing and managing its child components.
- Defines an interface for accessing a component's parent in the recursive structure (optional).
Composite: participants (cont’d)

Leaf:

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

Composite:

- Defines behavior for components having children.
- Stores child components.
- Implements child-related operations in the component interface.
Composite: participants (cont’d)

Client:

- Manipulates objects in the composition through the component interface.
Behavioral patterns

- Command
- Interpreter
- Iterator
- Template
- Observer
- Master-slave
**Command: intent**

- Encapsulate a request as an object, thereby letting one parameterize clients with different requests, queue or log requests, and support undoable operations.
Command: motivation

- Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.
**Command**: example

![Diagram showing the command pattern example]
Command: structure

![Diagram showing the structure of Command]

- Document
  - Open()
  - Close()
  - Cut()
  - Copy()
  - Paste()

- Command
  - Execute()

- PasteCommand
  - Execute()

- document
  - document->Paste()
Command: structure (cont’d)

Application
Add(Document) → application

OpenCommand
Execute()
AskUser()

name = AskUser()
doc = new Document(name)
application->Add(doc)
doc->Open()
Command: participants

Command:
- Declares an interface for executing an operation.

Concrete Command:
- Defines a binding between a Receiver object and an action.
- Implements Execute by invoking the corresponding operations(s) on Receiver.
Command: participants (cont’d)

Client:
- Creates a *Concrete Command* object and sets its receiver.

Invoker:
- Asks the command to carry out the request.

Receiver:
- Knows how to perform the operations associated with carrying out a request. Any class may serve as a *Receiver*.
Interpreter: intent and motivation

- Want to compute something capable of being expressed in a language.
- The computation is performed on structural representation of the language operations to be performed.
- Structural objects representing operations have built in knowledge of how to perform their own operation.
Interpreter: example

- A language statement may be converted to an expression tree of operations to be performed.
- Each operation takes as its input the results returned by it’s child expression trees.
- Such an expression tree is often called a plan.
**Iterator: intent**

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Move the responsibility for access and traversal from the aggregate object to the iterator object.
- Potentially preserves state information which speeds traversal.
Iterate: motivation

- One might want to traverse an aggregate object in different ways.
- One might want to have more than one traversal pending on the same aggregate object.
- Not all types of traversals can be anticipated a priori.
- One should not bloat the interface of the aggregate object with all these traversals.
### Iterator: example

<table>
<thead>
<tr>
<th>List</th>
<th>ListIterator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count()</td>
<td>First()</td>
</tr>
<tr>
<td>Append(Element)</td>
<td>Next()</td>
</tr>
<tr>
<td>Remove(Element)</td>
<td>IsDone()</td>
</tr>
<tr>
<td>...</td>
<td>CurrentItem()</td>
</tr>
</tbody>
</table>

```
index
```
**Iterator**: structure

```
Aggregate
  CreateIterator()

ConcreteAggregate
  CreateIterator()

Iterator
  First()
  Next()
  IsDone()
  CurrentItem()

return new ConcreteIterator(this)
```

CS646: Software Design and Architecture
**Iterator**: participants

- **Iterator**: Defines an interface for accessing and traversing elements.

- **Concrete Iterator**: Implements an iterator interface and keeps track of the current position in the traversal of the aggregate.

- **Aggregate**: Defines an interface for creating an iterator object.

- **Concrete Aggregate**: Implements the iterator creation interface to return an instance of the proper concrete iterator.
**Template: intent**

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.

- The *Template Method* lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
By defining some of the steps of an algorithm using abstract operations, the template method fixes their ordering.
Template: structure

AbstractClass

TemplateMethod()
PrimitiveOp1()
PrimitiveOp2()

ConcreteClass

PrimitiveOp1()
PrimitiveOp2()

... PrimitiveOp1()
... PrimitiveOp2()
...
Template: participants

Abstract Class:

- Defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
- Implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in Abstract Class or those of other objects.
Template: participants (cont’d)

Concrete Class:

- Implements the primitive operations to carry out subclass-specific steps to the algorithm.
**Observer**: intent

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Observer: motivation

- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
- You don't want to achieve consistency by making the classes tightly coupled because that reduces their reusability.
Observer: example

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>30</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>30</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

\( a = 50\% \)
\( b = 30\% \)
\( c = 20\% \)

change notification

requests, modifications
Observer: structure

Subject
- Attach(Observer)
- Detach(Observer)
- Notify()

ConcreteSubject
- GetState()
- SetState()
- subjectState

Observer
- Update()

ConcreteObserver
- Update()

observerState = subject->GetState()

for all o in observers {
    o -> Update()
}
**Observer: structure (cont’d)**

The key objects in this 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.
Observer: participants

Subject:
- Knows its numerous observers.
- Provides an interface for attaching and detaching observer objects.
- Sends a notification to its observers when its state changes.

Observer:
- Defines an updating interface for concrete observers.
Observer: participants (cont’d)

Concrete Subject:
- Stores state of interest to concrete observers.

Concrete Observer:
- Maintains a reference to a concrete subject object.
- Stores state that should stay consistent with the subject's.
- Implements the updating interface.
Master-slave: intent

- Handles the computation of replicated services within a software system to achieve fault tolerance and robustness.

- Independent components providing the same service; slaves are separated from a component master responsible for invoking them and for selecting a particular result from the results returned by the slaves.
Master-slave: motivation

- Fault tolerance is a critical factor in many systems.
- Replication of services and delegation of the same task to several independent suppliers is a common strategy to handle such cases.
Master-slave: example

NuclearPP
acceptableRL()

Voter
RadLevel()

return max(
  slave1 -> RadLevel(),
  slave2 -> RadLevel(),
  slave3 -> RadLevel())

Slave1
RadLevel()

Slave2
RadLevel()

Slave3
RadLevel()
**Master-slave: structure**

![Diagram of master-slave architecture]

- **Client**
  - `Compute()` request
  - request service

- **Master**
  - `service()` request
  - forward request

- **Slave1**
  - `ServiceImpl1()`

- **Slave2**
  - `ServiceImpl2()`

- **Slave3**
  - `ServiceImpl3()`
Master-slave: participants

**Slave:**
- Implements a service.

**Master:**
- Organizes the invocation of replicated services.
- Decides which of the results returned by its slaves is to be passed to its clients.

**Client:**
- Requires a certain service in order to solve its own task.