Design Patterns

Patterns

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”

– Christopher Alexander
  (building architect)

Object Technology Patterns

• A pattern is a named description of a problem and solution that can be applied in new contexts.
• Patterns codify the principles and idioms that guide experienced developers in their creation of software.
• What is and isn’t a pattern depends on your point of view.
Object Technology Patterns (2)

- Design patterns are not about structures such as vectors that can be encoded in classes and reused as is, nor are they complex domain specific designs.
- Ideally, patterns should include guidance for how they can be applied in novel situations.
- Many patterns provide guidance for how responsibilities should be assigned to objects.

Responsibilities

- Doing
  - Doing something yourself
  - Initiating action in others
  - Controlling and coordinating action in others
- Knowing
  - About private encapsulated data
  - About related objects
  - About things you can calculate or derive

Pattern Names

Naming facilitates communication:

Fred: “Where do you think we should place the responsibility for printing a Sale? I think Model-View Separation would work well – how about a SaleReportView?”

Wilma: “I think Expert is a better choice since it is a simple print and the Sale has all the data required in the printout – let the Sale do it.”

Fred: “OK, I agree.”
Responsibility Patterns

- Expert
- Creator
- High Cohesion
- Low Coupling
- Controller
- Polymorphism
- Pure fabrication
- Indirection
- Don’t talk to strangers

Expert

- Solution: Assign a responsibility to the information expert – the class that has the information necessary to fulfill the responsibility.
- Problem: What is the basic principle by which responsibilities are assigned in object-oriented design?
- Example: Who should be responsible for knowing the grand total of a sale? According to the expert pattern, we look for the class that has the information needed to determine the total.

Expert (2)

Who has all the information needed for the grand total?
Who has the information for the line item subtotals?
**Expert (3)**

Responsibilities for knowing total, subtotal, and price.

**Expert (4)**

- Most used pattern for assignment of responsibility.
- Leads to designs where software objects *do* operations that are normally *done to* the corresponding real world object.
- When information is spread around, leads to collaboration among partial experts.
- Encourages more cohesive “lightweight” classes due to distributed behavior – high cohesion.
- Encapsulation is maintained – low coupling.

**Creator**

- **Solution:** Assign class B the responsibility to create an instance of class A if one of the following is true:
  - B aggregates or contains A objects
  - B records instances of A objects
  - B closely uses A objects
  - B has the initializing data for A objects (B is an Expert with respect to creating A objects).
- **Problem:** Who should be responsible for creating a new instance of some class? Goals are to support low coupling, increased clarity, encapsulation, and reusability.
Creator (2)

- **Example**: Who should be responsible for creating a `LineItem` of a sale?

  Since a `Sale` aggregates `LineItem` objects, the Creator pattern suggests that `Sale` is a good candidate to have the responsibility of creating `LineItem` instances.

```
makeUpc(upc, quantity) : Sale
create(quantity) : LineItem
```

Creator (3)

- Find a creator which needs to be connected to the created object anyway.
- Supports low coupling. The class of the created object is likely to be already visible to the creator class.
- If the creator class is chosen based on having the initializing data, this is an example of the Expert pattern.

Low Coupling

- **Solution**: Assign a responsibility so that coupling remains low.
- **Problem**: How to support low dependency and increased reuse? A class with too many strong connections to other classes:
  - May need change when related classes change
  - Is harder to understand in isolation
  - Is harder to reuse because it requires the presence of classes on which it is dependent
- **Example**: Who should be responsible for creating a `Payment` object and associating it with a `Sale`?
Low Coupling (2)

In the real world, POST “records” a Payment, so Creator suggests:

```
1: makePayment
   ✖️→ Payment
2: makeSale
   ✖️→ Sale
```

Low Coupling (3)

But Low Coupling suggests:

```
1: makePayment
   ✖️→ Payment
1.1: create
   ✖️→ Payment
2: makeSale
   ✖️→ Sale
```

Low Coupling (4)

- Low Coupling is an evaluative pattern applied by a designer while evaluating designs.
- In OO languages, common forms of coupling include:
  - Class X has a method that references an instance of Y (parameter, local variable, return value)
  - Class X has an attribute that refers to an instance of Y
  - Class X is a subclass of Y
  - Class X is an implementation of interface Y
High Cohesion

- **Solution:** Assign a responsibility so that cohesion remains high.

- **Problem:** How to keep complexity manageable? A class that has too many responsibilities or unrelated responsibilities is:
  - Hard to comprehend
  - Hard to reuse
  - Hard to maintain
  - Fragile, constantly affected by change

Low cohesion classes often represent a very "large-grain" of abstraction, or have taken on responsibilities that should have been delegated to other classes.

---

High Cohesion (2)

- **Example:**

  ![Diagram of classes and responsibilities]

  - The first design assigns responsibility for Payment creation to POST.
  - The second design delegates responsibility to Sale, which supports greater cohesion in POST.

---

High Cohesion (3)

- **Assignment of responsibility for Payment creation to POST** may be acceptable, but if a trend of assigning system operations to POST develops it may become a bloated and incohesive class.

- The second design delegates responsibility to Sale, which supports greater cohesion in POST.
Controller
• Solution: Assign the responsibility for handling a system event message to a class representing one of the following choices:
  – Represents the overall “system” (facade controller)
  – Represents the overall business or organization (facade controller)
  – Represents something active in the real world that might be involved in the task (role controller)
  – Represents an artificial handler of all system events of a use case (use case controller)

Controller (2)
• Problem: Who should be responsible for handling a system event?
  – System events are high level events generated by external actors.
  – Controllers are non-user interface objects responsible for handling a system event. They define methods for system operations.

Controller (3)
• Example: During system behavior analysis, system operations were assigned to the class System:

<table>
<thead>
<tr>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>receive()</td>
</tr>
<tr>
<td>enterItem()</td>
</tr>
<tr>
<td>makePayment()</td>
</tr>
</tbody>
</table>

Who should control system events such as enterItem?
Controller (4)

- The Controller pattern suggests one of four possibilities:
  1. POST – represents the overall system
  2. Store – represents the overall business
  3. Cashier – represents something active in the real world that might be involved in the task
  4. BuyItemHandler – represents an artificial handler of all system events of a use case

- The choice is influenced by other factors such as cohesion and coupling.

Controller (5)

- System events should be controlled in the domain layer of objects, not in the interface or presentation layer.
- The same controller class should be used for all system events of a given use case.
- Controllers should normally delegate the work to other objects while coordinating the activity.

Controller (6)

- Facade controllers are suitable when there are only a few system events, or when it is impossible to redirect event messages to multiple controllers.
- Use case controllers are suitable when the other choices lead to low cohesion or high coupling:
  - When an existing controller becomes bloated
  - When there are many system events across different processes – factor into separate manageable classes.
Controller (7)

- Role controllers are acceptable when the designer is careful to avoid creating an incohesive controller that does not delegate.
  - Don’t create “person-like” objects to do all the work.
- Message handling applications may combine the facade and command patterns
  - For each message type there is a corresponding command class with an execute method

Key Points

- Patterns capture the essential elements of the best existing designs.
- Patterns should include guidance for how they can be applied in novel situations.
- Patterns that provide guidance for how responsibilities should be assigned to objects include:
  - Expert, Creator, High Cohesion, Low Coupling, and Controller