What’s Next? Other Patterns.

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

Outcomes

After today's lecture you will be able to:

- Understand what other patterns are available outside what was covered in this course
- Understand that patterns and pattern concepts go well beyond the domain of design

Inspiration

“Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius - and a lot of courage - to move in the opposite direction.” – Albert Einstein

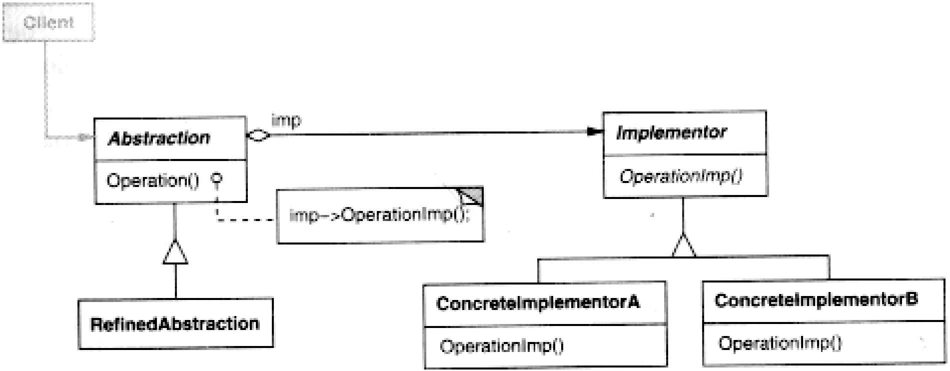
Other GoF Patterns

These patterns are used relatively infrequently used and we are only going to quickly cover them. They are summarized in Chapter 14 of the HFDPs book.

Bridge

Go4 description: Decouple an abstraction from its implementation so that the two can vary independently.

Go4 Diagram:



Bridge

Bridge is an apt name, because it forms a bridge between two inheritance hierarchies.

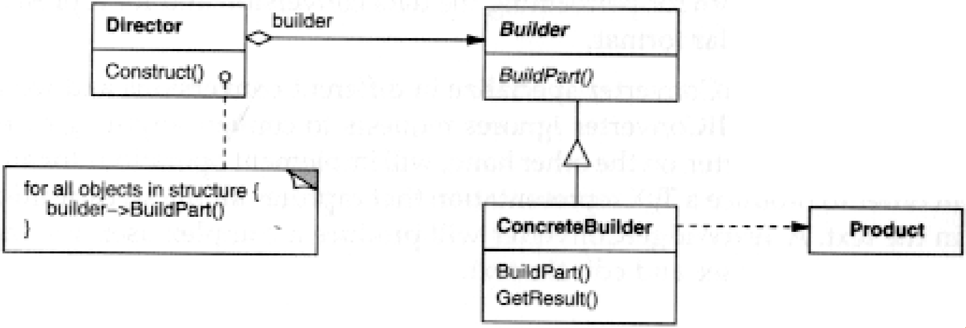
Bridge emerges as a result of a refactoring which introduces delegation:

- Suppose you have a complex inheritance tree (example: `Window` with subclasses `IconWindow`, `Dialog`, etc)
- Suppose there are also different modalities of implementation of the whole tree, for example PC, UNIX, and Mac-specialized implementations are needed.
- Refactor this mess into two trees, an abstraction hierarchy which is the original `Window/IconWindow/Dialog` tree with the PC/Mac/UNIX implementation bits removed, and an implementation tree which has an abstract Implementation class and each concrete `ImplementationMac`, `ImplementationPC`, `ImplementationUNIX` as subclasses.
- The abstract tree then delegates to its implementation object for the low-level code

Builder

Go4 description: Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Go4 Diagram:



Builder

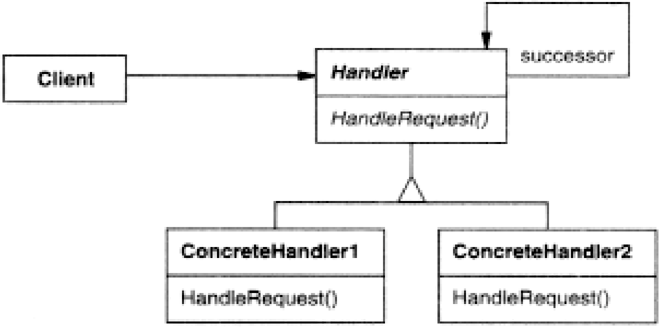
- the `Director` needs to create many different kinds of parts to make the full product. In the stupid method, the director has a whole pile of different classes he has to `new`.
- In the smart method above, he has a uniform pile (an `Array` say) of `Builder`'s, and by invoking `BuildPart()` on each one in a loop, he gets all his parts made with minimal code fuss.
- the `ConcreteBuilder` is a particular concrete `Builder`, a factory class, designed to create `Product`'s.

Idaho State University Computer Science

Chain of Responsibility

Go4 description: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle a request. Chain the receiving objects and pass the request along the chain until an object handles it.

Go4 Diagram:



ROAR

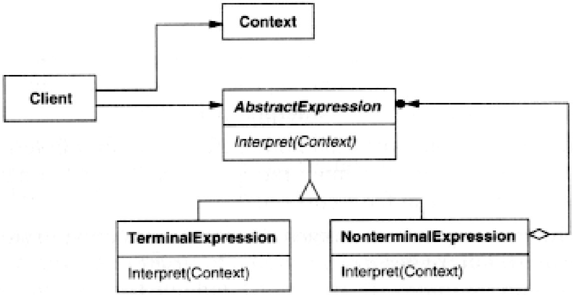
Chain of Responsibility

- This pattern shares some structural similarities with Java exceptions: either they are handled or passed on. But with exceptions they are implicitly passed on if they are not handled; here the passing on is explicit
- This pattern is useful for hierarchical structures where a request can be handled at multiple layers.
- **Example:** GUI event handling can be done hierarchically. If a contained view doesn't want to handle an event it can delegate it to its container, etc up the chain to the window. Java doesn't use this event model however.

Interpreter

Go4 description: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Go4 Diagram:



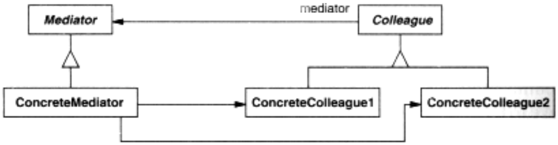
- This is a very specific pattern to represent language syntax.
- It is a variation on **Composite**. In general it is also like **Composite** showing how **union** types are encoded via this recursive diagram structure.

Mediator

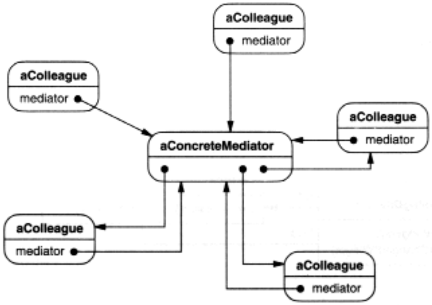
Put someone in charge (a mediator) of an interaction between two classes.

Go4 description: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets one vary their interaction independently.

Go4 Diagram:



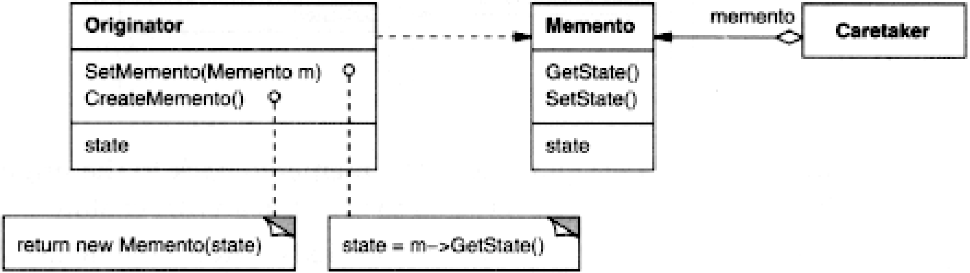
A typical object structure might look like this:



Memento

Go4 description: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Go4 Diagram:



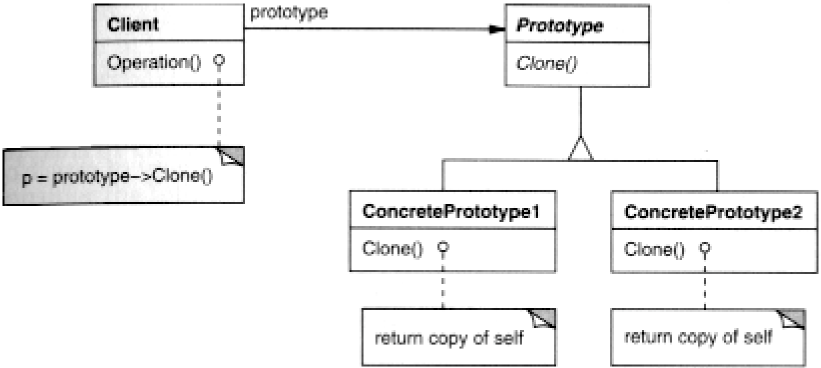
- This pattern can be one useful way to interact with a database in an object-oriented fashion: keep mementos around of all objects

Idaho State University Computer Science

Prototype

Go4 description: Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype.

Go4 Diagram:



ROAR

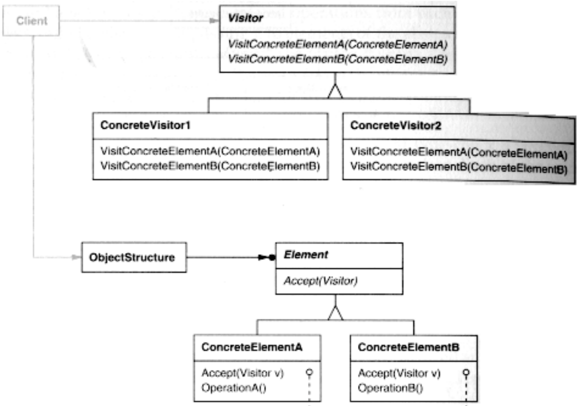
Prototype

- When you want new objects, copy from a prototype instead of creating directly from a class.
- Useful when its a significant effort to create object structure from scratch.
- **Example:** To create a new `Deck` of 52 playing cards, cards could be copied from a static variable in `Deck` which was originally initialized when the class was loaded to hold a “fresh” `Deck` of `Card` objects rather than making cards all over again.

Visitor

Go4 description: Represent an operation to be performed on the elements of an object structure. Visitor allows one to define a new operation without changing the classes of the elements on which it operates.

Go4 Diagram:



Visitor

- Suppose you have an `Element` object in a variable and need to perform a `switch` on what concrete subclass of `Element` we in fact have.
- Note that this is an incredibly common C programming pattern on `union` types – you are casing on which branch of the union you are in (the C analogy of inheritance is union).
- The problem is this notion does not fit well with O-O, the union is treated as passive in this switch; you are also casing at run-time on what class an object is, a brittle programming pattern.
- Alternative 1: add a method to each class in the union to do the walk-through
  - Big Advantage: we kept things highly O-O!
  - Big Disadvantage: this is shotgun surgery – each time we want to do such a switch we have to add a method to all the classes in the tree. Code gets all spread out.

Visitor

- Alternative 2: **Visitor**
  - Add an intermediary class, the visitor, which holds all the cases
  - The classes in the original inheritance hierarchy gets a new method `Accept` to help “walk” the visitor through the union
  - ... this is a compromise, we are not completely violating O-O and we avoid shotgun surgery when adding an operation over the tree, but it adds complexity to the design.
  - Note that if we add a new concrete element type we on the other hand have to do surgery on all visitors. But, we have localized the surgery to just the visitors.
- This pattern is another pattern that is useful to get rid of switch statements.

How the Visitor works:

- Abstract superclass `Visitor` is the superclass of all visitors
- `ConcreteVisitor1` is a concrete visitor (e.g. we make a class `GetHealthRating` for `getHealthRating()` in the menu example); we make new `ConcreteVisitorX` for each different switch we wanted to do over the union.
- `ConcreteVisitor1` has a method `visitConcreteElementA` etc for each kind of node A/B/.. in the original union structure - this is where the code in the original switch for the case it is **A/B**.. goes.
- `anElement.accept(aVisitor)` starts the visiting process
- This method in each inheritance class `ConcreteElementA` etc in turn calls the correct "case" to be performed on it, e.g. `ConcreteElementA` calls `visitConcreteElementA(this)` which will run the correct case of the switch..

The place where **Visitor** really shines is using it together with **Composite** to visit a tree structure.

Are there any questions?

ROAR