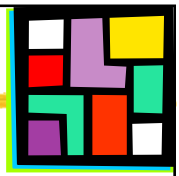
Design Patterns & Anti-Patterns

CSC 4700 Software Engineering

Patterns



- Provide solutions to recurring problems
- Balance sets of opposing forces
- Document well-proven design experience
- Abstraction above level of a single component
- Provide common vocabulary and understanding
- Are a means of documentation
- Support software devel with desirable properties

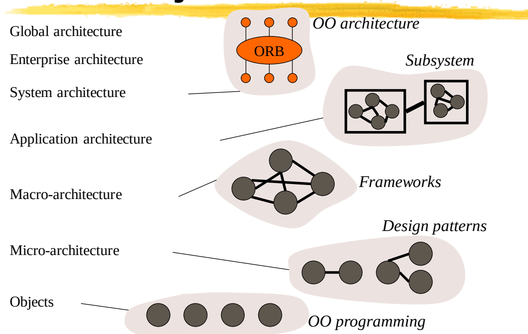
Purpose

- A design pattern captures *design expertise* – patterns are not created from thin air, but abstracted from *existing* design examples
- Using design patterns is *reuse* of design expertise
- Studying design patterns is a way of studying how the "experts" do design
- Design patterns provide a *vocabulary* for talking about design

Why design patterns in SA?

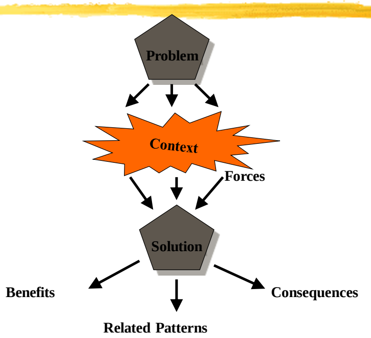
- If you’re a software engineer, you should know about them anyway
- Design Patterns help you *break out* of first-generation OO thought patterns

The seven layers of architecture*



* Mowbray and Malveau

How patterns arise



Structure of a pattern

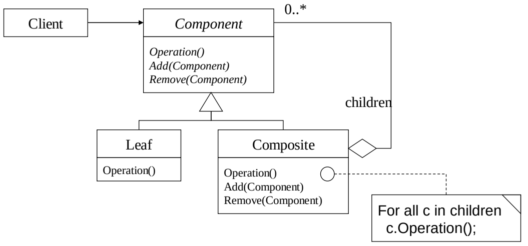
- Name
- Intent
- Motivation
- Applicability
- Structure
- Consequences
- Implementation
- Known Uses
- Related Patterns

Key patterns

- The following patterns are considered to be a good "basic" set of design patterns
- Competence in recognizing and applying these patterns *will* improve your design skills

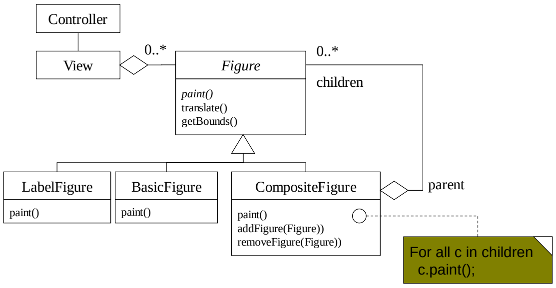
Composite

- Construct part-whole hierarchy
- Simplify client interface to leaves/composites
- Easier to add new kinds of components



Composite (2)

- Example: figures in a structured graphics toolkit



Adapter

- You have
  - legacy code
  - current client
- *Adapter* changes interface of legacy code so client can use it
- *Adapter* fills the gap b/w two interfaces
- No changes needed for either
  - legacy code, or
  - client

Adapter (2)

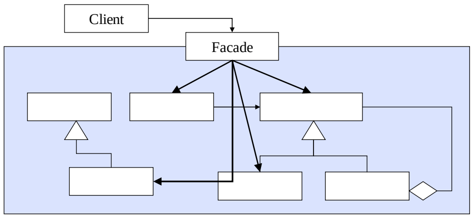
```
class NewTime
{
public:
int GetTime() {
        return m_oldtime.get_time() * 1000 + 8;
     }
private:
    OldTime m_oldtime;
};
```

Command

- You have commands that need to be
  - executed,
  - undone, or
  - queued
- *Command* design pattern separates
  - Receiver from Invoker from Commands
- All commands derive from *Command* and implement do(), undo(), and redo()

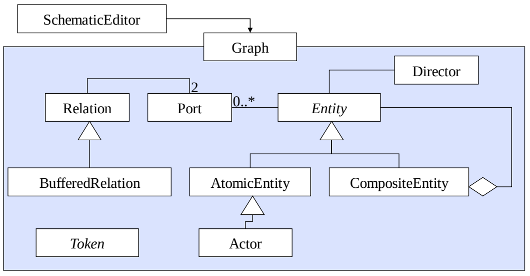
Facade

- Provide unified interface to interfaces within a subsystem
- Shield clients from subsystem components
- Promote weak coupling between client and subsystem components



Facade (2)

- Example: graph interface to a simulation engine

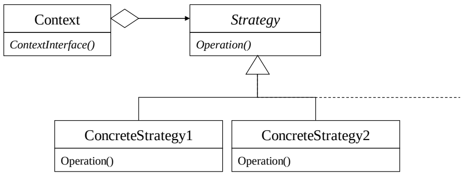


Proxy

- You want to
  - delay expensive computations,
  - use memory only when needed, or
  - check access before loading an object into memory
- *Proxy*
  - has same interface as Real object
  - stores subset of attributes
  - does lazy evaluation

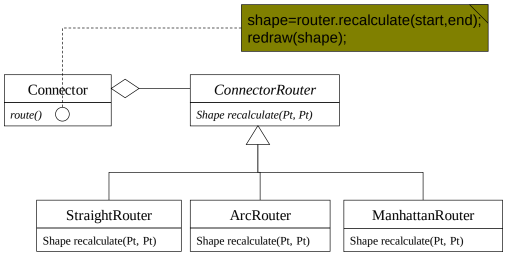
Strategy

- Make algorithms interchangeable---"changing the guts"
- Alternative to subclassing
- Choice of implementation at run-time
- Increases run-time complexity



Strategy (2)

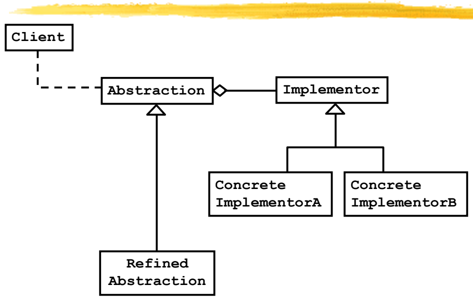
- Example: drawing different connector styles



Bridge

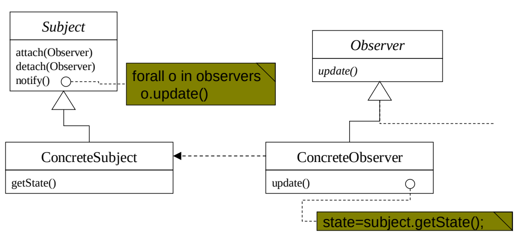
- You
  - have several different implementations
  - need to choose one, possibly at run time
- *Bridge*
  - decouples interface from implementation
  - shields client from implementations
  - Abstraction creates and initializes the ConcreteImplementations
  - Example: stub code, slow code, optimized code

Bridge (2)



Observer

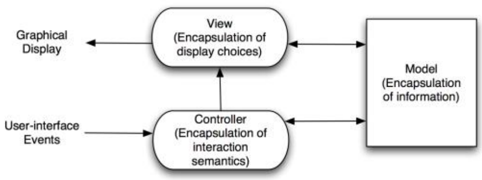
- Many-to-one dependency between objects
- Use when there are two or more views on the same "data"
- aka "Publish and subscribe" mechanism
- Choice of "push" or "pull" notification styles



Model-View-Controller (MVC)

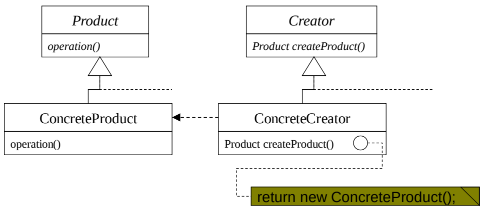
- Objective: Separation between information, presentation and user interaction.
- When a model object value changes, a notification is sent to the view and to the controller.
  - Thus, the view can update itself and the controller can modify the view if its logic so requires.
- When handling input from the user the windowing system sends the user event to the controller.
  - If a change is required, the controller updates the

Model-View-Controller



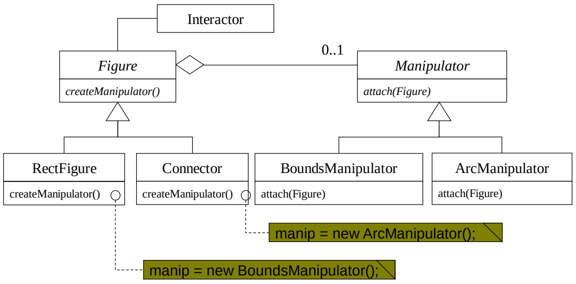
Factory Method

- Defer object instantiation to subclasses
- Eliminates binding of application-specific subclasses
- Connects parallel class hierarchies
- A related pattern is AbstractFactory



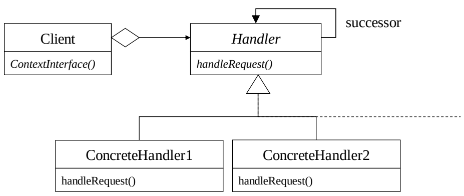
Factory Method (2)

- Example: creating manipulators on connectors



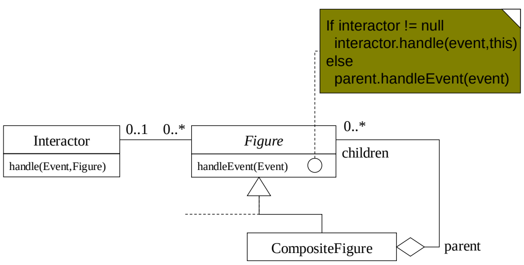
Chain of Responsibility

- Decouple sender of a request from receiver
- Give more than one object a chance to handle
- Flexibility in assigning responsibility
- Often applied with Composite



Chain of Responsibility (2)

- Example: handling events in a graphical hierarchy



Patterns vs "Design"

- Patterns *are* design
  - But: patterns transcend the "identify classes and associations" approach to design
  - Instead: learn to recognize patterns in the *problem* space and translate to the solution
- Patterns can capture OO design principles within a specific domain
- Patterns provide structure to "design"

Patterns vs Frameworks

- Patterns are lower-level than frameworks
- Frameworks typically employ many patterns:
  - Factory
  - Strategy
  - Composite
  - Observer
- Done well, patterns are the "plumbing" of a framework

Anti-Patterns

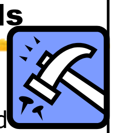
CSC 4700 Software Engineering

Auntie Patterns

Aunt "Patt"

Ant Tea Patterns

Anti-Patterns and Bad Smells



- Patterns describe desirable behavior
- Anti-patterns describe situations one had better avoid
- *Refactoring* is applied whenever an anti-pattern has been introduced
- Bad smells occur when something in your design seems "fishy"
  - They are not necessarily indications of problems

Example anti-patterns

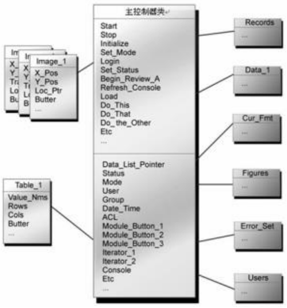
- *God class*: class that holds most responsibilities (also called *The Blob*)
- *Lava flow*: dead code
- *Poltergeist*: class with few responsibilities and a short life
- *Golden Hammer*: solution that does not fit the problem
- *Stovepipe*: (almost) identical solutions at different places
- *Swiss Army Knife*: excessively complex class interface

More example anti-patterns

- **Singletonitis** – over-use of the singleton pattern
- **Sequential coupling** – requires methods to be called in particular order
- **Object orgy** – failing to properly encapsulate objects permitting unrestricted access to their internals
- **Blind faith** – neglecting to test error returns from methods
- **Loop-switch sequence** – implementing sequential code as a loop statement, i.e. first time through do A, second time do B etc, rather than doA(); doB();
- **Magic numbers/strings** – unexplained number/string values in code

The Blob

The Blob



Golden Hammer

I have a hammer and everything else is a nail

Spaghetti Code

Cut-And-Paste Programming

"Hey, I thought you fixed that bug already, so why is it doing this again?"

"Man, you guys work fast. Over 400,000 lines of code in three weeks is outstanding progress!"