Idioms & Design Patterns Structural

Patterns and idioms can be grouped roughly into:

- Creational Patterns and idioms

  Singleton, Factory Method, Abstract Factory

  Named constructor
- Structural patterns and idioms

  Composite, Facade, Adapter

  Compiler Firewall
- Behavioral patterns and idioms

  Observer, MVC - more

  Double-dispatch
- Design patterns come in two basic flavors:

  Using one or more base classes to hide details from the client
  - One thing about most of the design patterns: presented in a very general form

    E.g. normally everything has an abstract base class that defines the working interfaces between the parts of the pattern

    But an actuall implementation might not need this - the abstract base can be collapsed into a single concrete class
  - E.g. Factory

  Other clever ideas using encapsulation, interfaces, class responsibilities to hide details from the client.
  - e.g. Singleton

Key concepts for using design patterns:

- Take the class relationships and the details seriously!

  You aren't taking advantage of the design pattern just by having some classes with the "buzzword" names organized kinda like the pattern. The exact way in which the classes relate to each other, and how key details are handled, is where the real power of the pattern is!
- The goal of many of the patterns is to achieve easy extensibility, at the expense of some verbosity and some run-time overhead.

  This means that a special case solution to a design problem will take less code, and maybe run faster, but it will be much harder to generalize when new features and capabilities get added to the code.

  The need to extend a program is very common, even if it wasn't planned for.

  Either use a design pattern from the beginning, to allow for future extensibility, or be ready to refactor the special case solution to make use of a design pattern so that future extensions will go more smoothly.

  So the goal of the patterns is not short code, or fast code, but easy-to-extend code.

Structural Patterns

Idiom: Compiler Firewall: Separating interface from implementation

- interface is presented to

  other classes (public interface)

  derived classes (protected interface)

- implementation is how it gets done -

  private members, contents of fucntions

  also by non-public inheritance
  - adds capabilities without changing the interface

- separation of interface and implementation is critical

  expose only the minimum in the public interface

  do not provide direct access to private member data
  - // e.g. from "Dungeons & Dragons" sort of Role-playing game
  - class Player

```
private:
    multimap<string, Item *> knapsack; // container of items
public:
    // disaster waiting to happen
    multimap<string, Item *>& get_knapsack()
        {return knapsack;}
    // slightly better, but questionable - why is it needed?
    const multimap<string, Item *>& get_knapsack()
```

  C++ permits some of it directly
  - source files supply the implementation
  - header files provide access to the interface

    other module just includes it to use the class without being exposed to most of implementation details

  what has to be recompiled if what changes?
  - unfortunately, some of implementation detail is exposed in the class declaration - the private or protected members, what gets inherited.

    These affect the size of the object. If object declared as local or member var, or allocatd with new, this must be known.
  - means changes to *implementation* affect other modules ... force a recompile, even if public interface does not change

  Concept of **insulation - insulating part of a system to changes in another part**

- compiler firewall Idiom

  Provide insulation - set up classes so that change can't propagate past the "firewall".

  file Gizmo.h
  - class Gizmo_Impl; // incomplete declaration
  - class Gizmo {

```
public:
    Gizmo();
    ~Gizmo();
    void start();
    void stop();
private:
    Gizmo_Impl * pimpl;
};
```

This file never changes unless public interface changes

file Gizmo.cpp

- #include "Gizmo_Impl.h"
- Gizmo::Gizmo() : pimpl(new Gizmo_Impl) {}
- Gizmo::~Gizmo()
  {delete pimpl;}
- Gizmo::start()
  {pimpl->start();}
- Gizmo::stop()
  {pimpl->stop();}

this file will have to be recompiled if Gizmo_Impl.h changes, but change will not propagate further

file Gizmo_Impl.h

- class Gizmo_Impl {
- .....

file Gizmo_Impl.cpp

- .....

class implementer can change Gizmo_Impl freely, without having to worry about forcing users (= "client") of Gizmo to recompile for every little change in the private members of Gizmo's implementation

Wrapper Idiom

- Hide the details of a pre-existing facility inside a class, provide for initialization and deinitialization with constructors and destructors, and a better interface.
- Example: wrap C file I/O in a class:

class Output_file {

- private:
  FILE * file_ptr;
- public:
  Output_file(const string& filename, const string& mode); // open the file with fopen()
  ~Output_file(); // close the file
  operator bool(); // test the state of the file stream
  Output_file& operator<< (int); // output an integer
  Output_file& operator<< (const string&) // output a string
- };

in fact, early implementations of iostream and fstream were just wrappers around the C's stdio

Facade Pattern

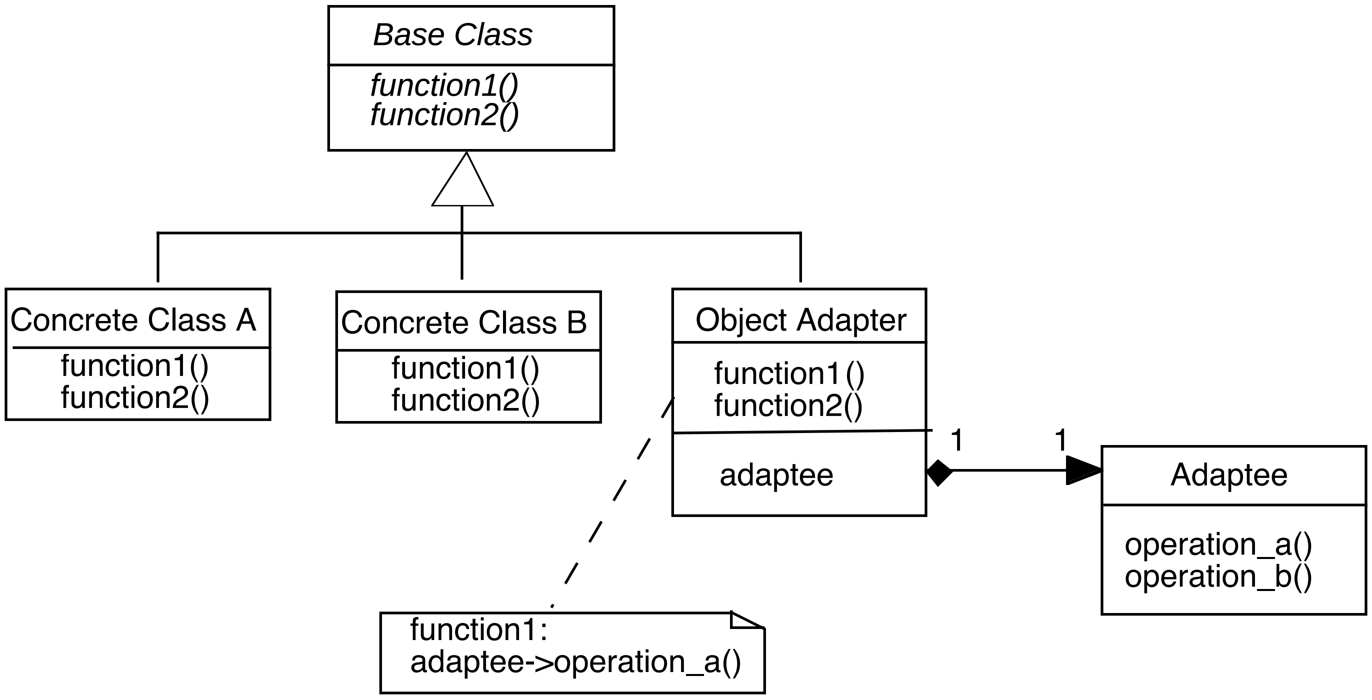
- A larger-scale version of the wrapper idea.
- Wrap an entire subsystem in a class, hiding the whole subsystem, and providing a simpler interface to the subsystem and hide all of its details. The Facade class constructor might build and configure the subsystem. The public interface might do additional computation and multiple calls into the subsystem components.
- My production-system example.

Basic interface is load, reset, and run, plus various mode/output settings.

Internals involve a several dozen classes, a whole collection of templates, very complicated processing - all hidden.

Adapter Pattern

- A wrapper that mates an odd-ball class or facility to a pre-defined interface.
- Used when you need everything to fit into a polymorphic class hierarchy.
- Adapter Pattern



- This is an object Adapter - works by delegating to a subobject (via composition).
- There is also a "class Adapter" - works by inheriting from an interface.

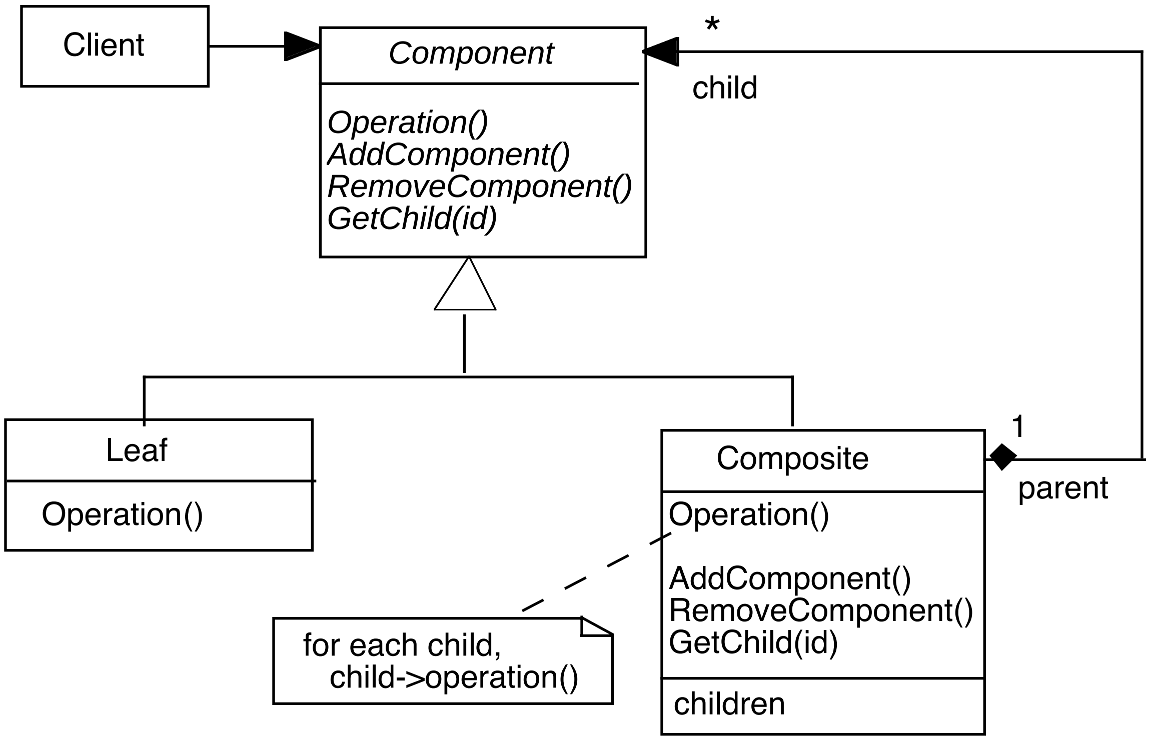
  e.g. see MVC example where a View base class is used to provide an interface for classes inheriting from GUI library, thereby allowing model to talk to GUI subclasses without becoming dependent on the GUI library details.

Composite Pattern

- Problem: You have a collection of basic objects that can be grouped in arbitrary ways. However, the client code needs to be able to operate on all of the objects the same way, even if they are grouped together.

  Common example - a drawing program - can group shapes together, have them drawn, resized, moved, etc as a group.But individal objects still exist.

- Solution: A base class represents a component of the collection. The component is either a basic leaf object, or is a composite object. A composite object contains a list of components (its children). A typical structure of objects will have a composite component at the top, whose children are leaf objects or other composite components. You can operate on a component; if it is a leaf object, it does the operation. If it is a composite object, it does the operation on each of its children.



There is a fat vs. thin interface issue in this pattern.

- What varies: Whether the child is a leaf or a composite, hidden under the component interface.
- Warning: It can be tempting to try to collapse two of the three classes (Component, Leaf, Composite) together into a single class. This is NOT a good idea - serious violations of the basic principle of inheritance can result, causes many bizarre problems in the design.

Example: Let leaf be an Agent in a game/simulation, in which the Client code controls the Agents (tells them what to do). Then Component represents the "controllable" things - the things that the client can control, by virtue of the Component interface. Composite then represents a group of controllable things. The controllable things are either groups of controllable things or individual Agents.

- Consider what happens if you combine some of these classes:
- If you make Leaf into the Component, you are saying that a *group* of Agents *IS-A* Agent. which can't be true. While a football team consists of football players, a team is not an individual person.
- If you make Composite into the Component, You are saying that an individual Agent *IS-A* group of Agents, which can't be true. The person playing quarterback is not a football team.

Keep the classes distinct: Component provides the common interface to both individuals (leaf) and groups (composite), and allows the client to communicate with either individuals or groups in the same way. A group contains a collection of Components, which in turn are either individual or groups.

Flyweight

- Use object-oriented techniques for a very large number of objects, without incurring storage overhead for a large number of objects.

  Abstracting what to do from the specifics.

  The Flyweight object is a package of functionality with no data.

- Example: display complex formatted text on the screen by having an object for each character and telling it to draw itself - It draws the appropriate pixel pattern with the appropriate shape and location.

  Drawing code doesn't have to know what character is being displayed.

  But number of objects is huge - if they contain their own data (e.g. location), storage becomes ridiculous.

- Pattern solution

  Separate intrinsic from extrinsic state; put extrinsic state elsewhere
  - Instrinsic state can be shared by all objects of that type.
  - Extrinsic state can be stored elsewhere or computed as needed.

  Create one shared object of each type needed, keep multiple pointers to multiple shared objects in appropriate containers.

  To accomplish something, call a virtual function for each pointed-to object, and supply the extrinsic information. Virtual function for that type of object does the appropriate thing with the supplied information.

- Example of complex text display:

  Create one sharable object (the Flyweight) for each ASCII character code.
  - Maximum is total number of distinct character codes.
  - Use a factory:
    If flyweight already exists, simply return pointer to it.
    If not, create it and add it to a Pool of flyweights.

  Each line of text represented by a container of pointers to shared objects.

  To display the line, compute the position of each character, and give that to each objects Drawself function.

  Fancy text editors have been built this way; worked well.

- Other applications for graphical programs are possible

  Especially in MVC -

  Data about a bunch of objects is often held independently of how they need to be drawn on the screen - the Model vs. the View

  Useful way to reduce the redundancy between the "real" objects in the Model data and the View objects.

Some others in Gang of Four book

- Decorator
- Proxy
- Bridge