Structural Patterns

- How classes and objects are composed to form larger structures
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy

Façade

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

Week 9b Nov 6/03 - Structural

Applicability

- you want a simple interface to a complex subsystem
  - Subsystems often get more complex as they evolve
    - this makes the subsystem more reusable and easier to customize,
    - but it also becomes harder to use for clients that don’t need to customize it
  - A façade can provide a simple default view of the subsystem that is good enough for most clients
    - Only clients needing more customizability will need to look beyond the façade
- there are many dependencies between clients and the implementation classes of an abstraction
  - Introduce a façade to decouple the subsystem from clients and other subsystems
- you want to layer your subsystems
  - Use a façade to define an entry point to each subsystem level
**Structural**

- **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 façade

**Consequences**

- shields clients from subsystem components
  - reduces the # of objects clients see
  - easier to use subsystem
- promotes weak coupling between the subsystem and its client
  - can vary the components of a subsystem without affecting clients
  - reduces compilation dependencies
- doesn't prevent applications from using subsystem classes if they need to.
  - you can choose between ease of use and generality

**Proxy**

- Provide a surrogate or placeholder for another object to control access to it
  - e.g., on-demand image loading
    - so that opening a document is fast (since screen res much lower than print res)

**Proxy - CORBA remote proxy**

Clients send messages to local proxy.

Server deserializes params and transmits to server

CorBA server

Class
Applicability

- whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer
  - A **remote proxy** provides a local representative for an object in a different address space
    - One of main ideas behind “distributed objects”.
  - A **virtual proxy** creates expensive objects on demand
  - A **protection proxy** controls access to the original object.
    - Protection proxies are useful when objects should have different access rights
  - A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed
    - counting the number of references to the real object (**smart pointer**) 
    - loading a persistent object into memory when it's first referenced 
    - checking that the real object is locked before it's accessed to ensure that no other object can change it
  - COW (copy-on-write)

Structure

- **Proxy**
  - maintains a reference that lets the proxy access the real subject
  - provides an interface identical to Subject's so that a proxy can by substituted for the real subject
  - controls access to the real subject and may be responsible for creating and deleting it
    - *remote proxies* are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space
    - *virtual proxies* may cache additional information about the real subject so that they can postpone accessing it
    - *protection proxies* check that the caller has the access permissions required to perform a request

Flyweight

- Use sharing to support large numbers of fine-grained objects efficiently
- Reduce the space consumed by many objects by reusing a reasonably sized pool of them many times.
- We have been over and over this example. It’s simply the best one.
- Next up is one of those pictures that is worth a thousand words..
Applicability

- Use when:
  - An application uses a large number of objects
  - Storage costs are high because of the sheer quantity of objects
  - Most object state can be made extrinsic
  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
    - e.g. the letter “Z”.
  - The application doesn't depend on object identity
    - Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects
  - This sounds a lot easier than it is.
    - At least a few Stanford PhD’s written out as Linton et al worked out how to build interactive apps this way.
    - Trick appears to be to find an abstraction that allows most Flyweights to collaborate without storing any private state.

Flyweight

- declares an interface through which flyweights can receive and act on extrinsic state
- This may well be the hard part!
• **ConcreteFlyweight**
  - implements the Flyweight interface and adds storage for intrinsic state, if any
  - must be sharable
    • Any state it stores must be intrinsic (independent of context)

• **FlyweightFactory**
  - creates and manages flyweight objects
  - ensures that flyweights are shared properly
    • when a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists

• **UnsharedConcreteFlyweight**
  - not all Flyweight subclasses need to be shared.
  - The Flyweight interface enables sharing; it doesn't enforce it

• **Client**
  - maintains a reference to flyweights
  - computes or stores the extrinsic state of flyweights
  - computation is likely delegated to the flyweights.
**Structure**

- Clients typically should not instantiate ConcreteFlyweights directly for fear of needlessly duplicating instances.
- Clients should obtain ConcreteFlyweight objects from the FlyweightFactory object to ensure they are shared properly.

**Consequences**

- Flyweights introduce run-time costs associated with transferring, finding, and/or computing extrinsic state.
- Costs are offset by space savings
  - (which also save run-time costs)
  - depends on
    - the reduction in the total number of instances that comes from sharing
    - the amount of intrinsic state per object
    - whether extrinsic state is computed or stored
- Often coupled with Composite to represent a hierarchical structure as a graph with shared leaf nodes
  - flyweight leaf nodes cannot store a pointer to their parent
  - parent pointer is passed to the flyweight as part of its extrinsic state
    - profound effect on object collaboration
    - probably limits the domains for which flyweight is appropriate.

**Implementation**

- Extrinsic State *e.g.*, Document editor
  - character *font*, *type style*, and *colour*.
  - try to use containment when possible.
    - *e.g.* All children of this node are bold.
  - store a map that keeps track of runs of characters with the same typographic attributes
- Shared Objects
  - FlyweightFactory can use an associative array to find existing instances.
  - need reference counting for garbage collection (in C++)