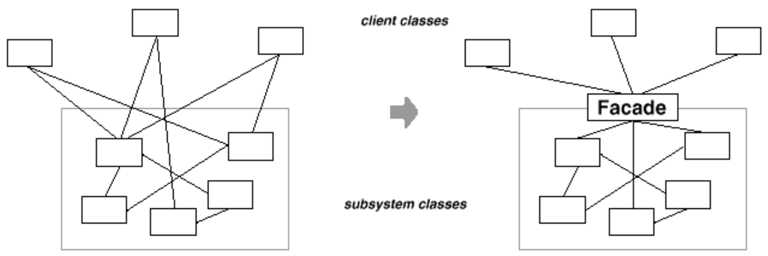
Structural Patterns

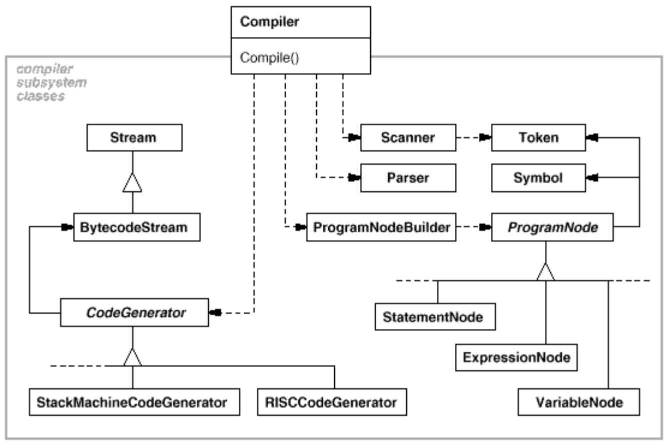
- concerned with how classes and objects are composed to form larger structures
  - ✓ Adapter
    - interface converter
  - ✓ Bridge
    - decouple abstraction from its implementation
  - ✓ Composite
    - compose objects into tree structures, treating all nodes uniformly
  - ✓ Decorator
    - attach additional responsibilities dynamically
  - – Façade
    - provide a unified interface to a subsystem
  - – Flyweight
    - using sharing to support a large number of fine-grained objects efficiently
  - – Proxy
    - provide a surrogate for another object to control access

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



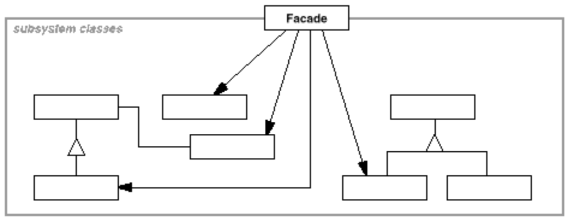
Façade



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

Structure



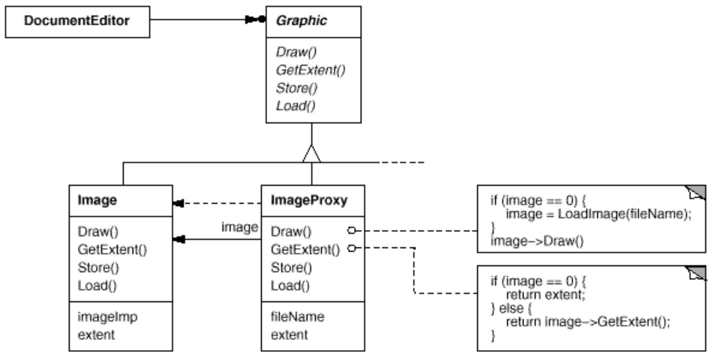
- Façade
  - knows which subsystem classes are responsible for a request
  - delegates client requests to appropriate subsystem objects
- subsystem classes
  - implement subsystem functionality
  - handle work assigned by the Façade object
  - have no knowledge of the façade

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



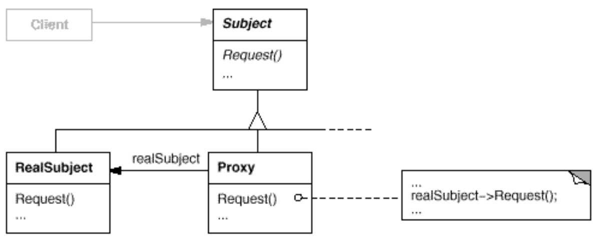
```
if (image == 0) {
    image = LoadImage(fileName);
}
image->Draw()
```

```
if (image == 0) {
    return extent;
} else {
    return image->GetExtent();
}
```

Applicability

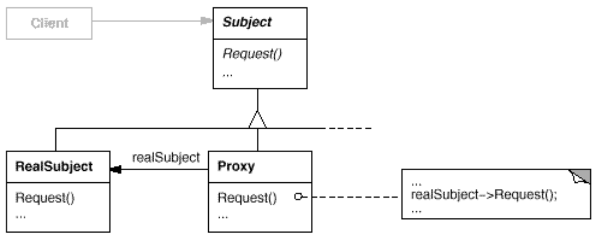
- whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer
  - A **remote proxy** provides a local representative for an object in a different address space
  - A **virtual proxy** creates expensive objects on demand
  - A **protection proxy** controls 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



- Subject
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected
- RealSubject
  - defines the real object that the proxy represents

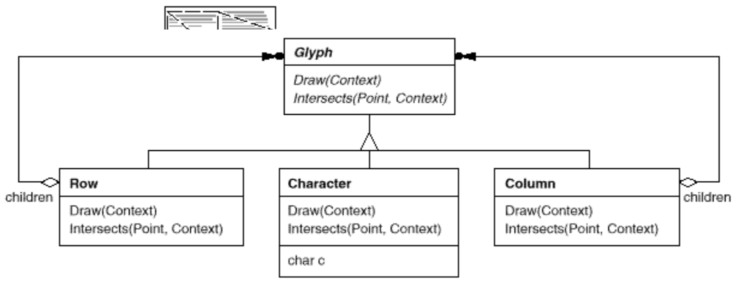
Structure



- 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

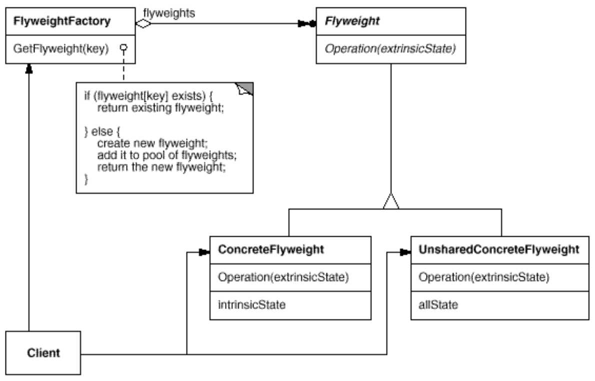
- Use sharing to support large numbers of fine-grained objects efficiently


Glyph
Draw(Context)
Intersects(Point, Context)
Row
Draw(Context)
Intersects(Point, Context)
children
Character
Draw(Context)
Intersects(Point, Context)
char c
Column
Draw(Context)
Intersects(Point, Context)
children

Applicability

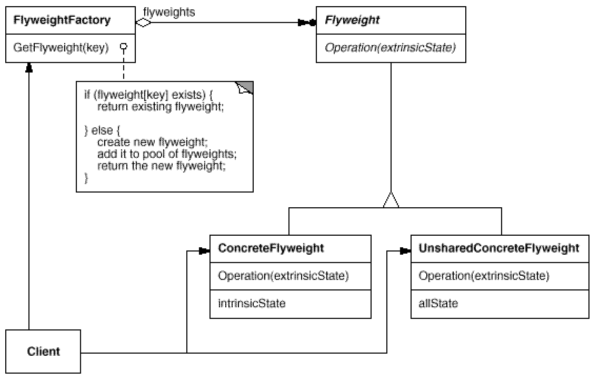
- Use when:
  - An application uses a large number of objects
  - Storage costs are high because of the sheer quantity of objects
  - Most object state can be made extrinsic
  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed
  - The application doesn't depend on object identity
    - Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects

Structure


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

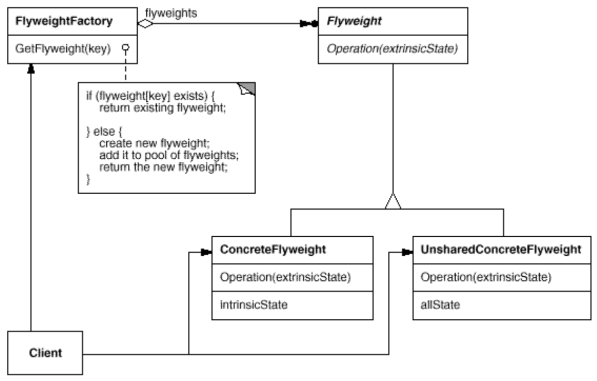
- Flyweight
  - declares an interface through which flyweights can receive and act on extrinsic state

Structure


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

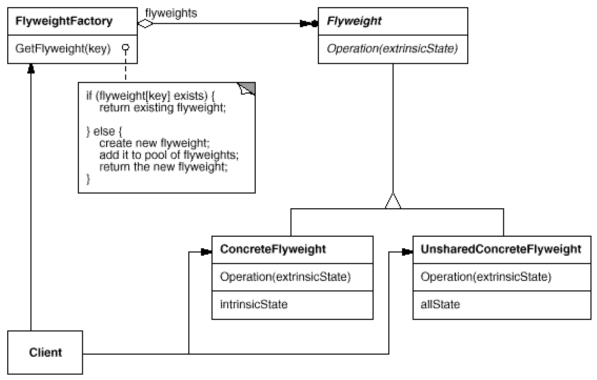
- ConcreteFlyweight
  - implements the Flyweight interface and adds storage for intrinsic state, if any
  - must be sharable
    - any state it stores must be intrinsic (independent of context)

Structure


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

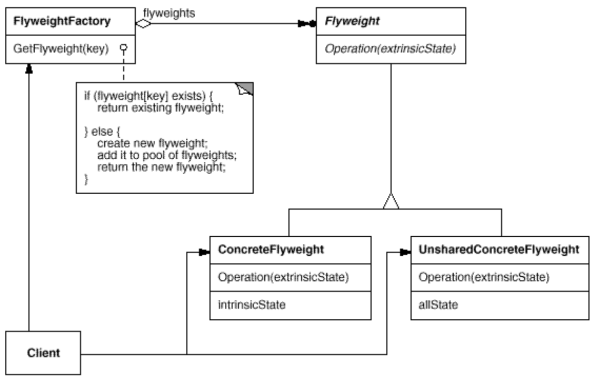
- UnsharedConcreteFlyweight
  - not all Flyweight subclasses need to be shared.
  - The Flyweight interface *enables* sharing; it doesn't enforce it

Structure


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

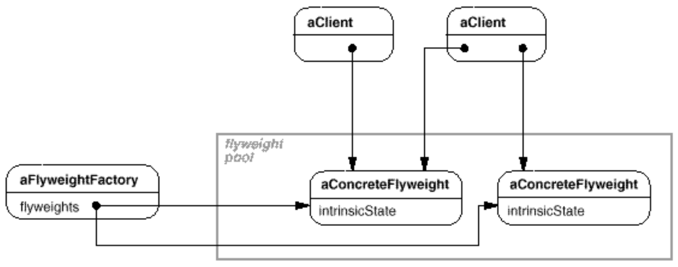
- FlyweightFactory
  - creates and manages flyweight objects
  - ensures that flyweights are shared properly
    - when a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists

Structure


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
    return existing flyweight;
} else {
    create new flyweight;
    add it to pool of flyweights;
    return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

- Client
  - maintains a reference to flyweights
  - computes or stores the extrinsic state of flyweights

Structure



- Clients should not instantiate ConcreteFlyweights directly.
- Clients must obtain ConcreteFlyweight objects exclusively 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

Implementation

- Extrinsic State *e.g.,* Document editor
  - character **font**, *type style,* and colour.
  - 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