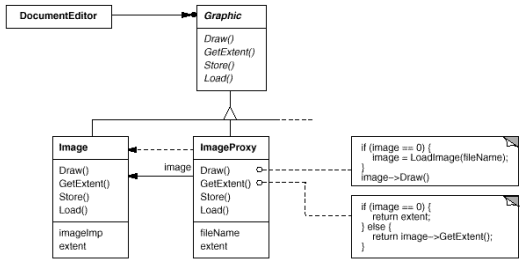
ECE450 – Software Engineering II

Today: **Design Patterns V**
More Structural Patterns

Difficulties opening large documents

- Consider a document editor that can embed graphical objects in a document
  - Some graphical objects (say, large images) can be expensive to create…
  - …but opening a document should be fast!
    - We shouldn't create all the expensive objects at once when the document is opened
  - …and we need data from every image (e.g. height and width) from the start!
    - So we have a dilemma: we don't want to create all expensive objects at once, but we need their information anyway!
- Alternatives?

The *Proxy* pattern

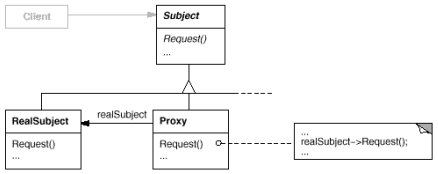


- ***Proxy****: Provide a surrogate or placeholder for another object to control access to it*
  - In our example, the DocumentEditor loads ImageProxy instead –and if we need to Draw() the image, the ImageProxy will load it.

Applicability

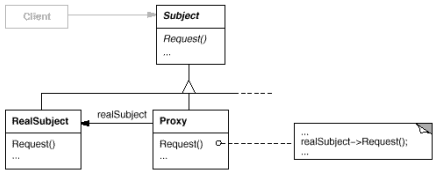
- Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer.
- Several possible scenarios:
  - **Remote Proxy:** Provide a local representative for an object in a different address space
  - **Virtual Proxy:** Create expensive objects on demand (as in our example)
  - **Protection Proxy:** Control access to the original object (useful when objects should have different access rights)
  - **Smart Reference:** A replacement for "bare" pointers that perform additional actions when objects are accessed:
    - Counting the number of references to the real object so that it can be freed automatically when there are no more references
    - Checking that the real object is locked before it's accessed to ensure that no other object can change it
    - …

Participants



- Proxy
  - Maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same
  - Provides an interface identical to Subject's so that a proxy can be substituted for the real subject
  - Controls access to the real subject and perhaps creates/deletes it
  - Other responsibilities, depending on the kind of proxy…
    - 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

Participants (cont)



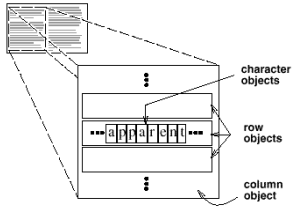
- Subject
  - Defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected
- RealSubject
  - Defines the real object that the proxy represents

Consequences

- Depending on the type of Proxy, the level of indirection that it introduces leads to different advantages:
  - A remote proxy can hide the fact that an object resides in a different address space
  - A virtual proxy can perform optimizations such as creating an object on demand
  - Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed
- Another optimization that the Proxy pattern can hide from the client is **copy-on-write**:
  - Copying a large object might be expensive, so when the client asks for a copy, we can just copy a proxy.
  - We postpone the actual copy of the object until the copy is modified
- Note that although decorators can have similar implementations as proxies, decorators have a different purpose. Decorators add one or more responsibilities to an object, whereas a proxy controls access to an object

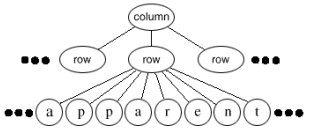
Next problem, but still with document editors…

- Object-oriented document editors typically use objects to represent embedded elements like tables and figures.
  - However, they stop short of using an object for each character in the document
    - Even though doing so would promote flexibility at the finest levels in the application



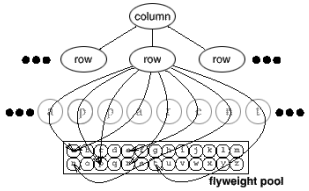
Lots of little objects

- Is it practical to make each character an object?
  - We could easily have hundreds of thousands of character objects!
    - Each of them representing a character, its location, font, size, style, etc.



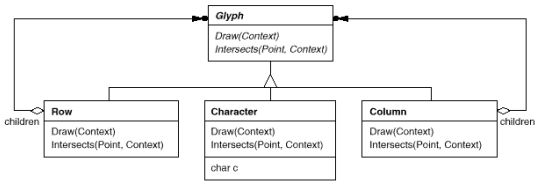
But…

- If we **share** objects from a pool, handling each character as an object becomes practical
  - Each "flyweight" object stores a character code, but its coordinate position in the document and its typographic style can be determined from the text layout algorithms
    - The character code is **intrinsic state**, while the other information is **extrinsic**
    - **Intrinsic** state consists of information that is independent of the object's context, thereby making it shareable
    - **Extrinsic** state depends on and varies with the context and therefore can't be shared



Class structure of example

- This is how the class structure of our example looks like
  - Glyph is the abstract class for graphical objects, some of which may be "flyweights"
  - A "flyweight" representing the letter "a" only stores the corresponding character code; it doesn't need to store its location or font
    - Clients supply the context-dependent information that the flyweight needs to draw itself

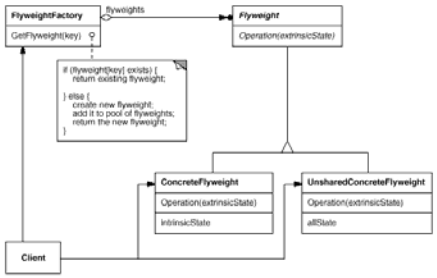


The *Flyweight* pattern

- *Use sharing to support large numbers of fine-grained objects efficiently*
- Applicability: Use when *all* of the following are true:
  - 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)

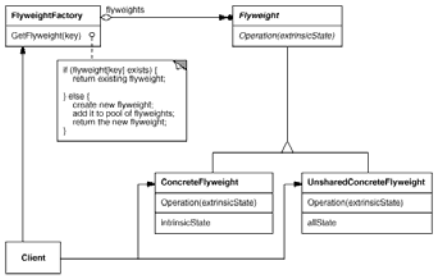
Structure and participants

- Flyweight
  - Declares an interface through which flyweights can receive and act on extrinsic state
- ConcreteFlyweight
  - Implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight must be shareable.



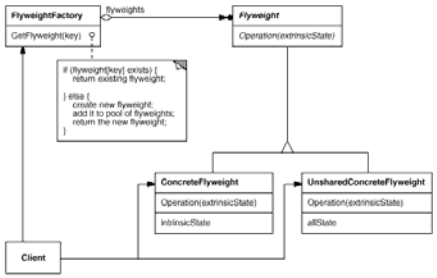
Structure and participants (cont)

- UnsharedConcreteFlyweight
  - Not all Flyweight subclasses need to be shared (e.g. rows and columns). The Flyweight interface *enables* sharing, it does not enforce it.
- FlyweightFactory
  - Creates and manages flyweight objects



Structure and participants (cont)

- Client
  - Maintains a reference to flyweights
  - Computes or stores the extrinsic state of flyweights
  - Should not instantiate ConcreteFlyweights directly. They must ubtain ConcreteFlyweight objects *exclusively* from the FlyweightFactory object to ensure they are shared properly



Consequences

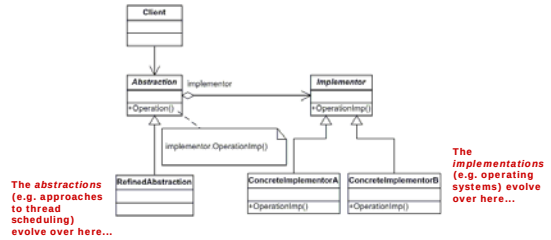
- Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state
  - However, such costs are offset by space savings, which increase as more flyweights are shared
- Storage savings are a function of several factors:
  - The reduction in the total number of instances that comes from sharing
  - The amount of intrinsic state per object
  - Whether extrinsic state is computed or stored

One more problem

- Consider the domain of thread scheduling. We have two types of thread schedulers (preemptive and time sliced), and two operating systems (Unix and Windows)
  - We define a class for each permutation of these two dimensions; we end up with four classes (preemptive-Unix, preemptive-Windows, etc.)
  - Adding platforms or approaches to thread scheduling increases considerably the number of classes we would need to code (and debug, and maintain…)
- The *Bridge* design pattern refactors this explosive inheritance into two orthogonal hierarchies:
  - One for platform-independent *abstractions*
  - One for platform-dependent *implementations*

The *Bridge* pattern

- *Decouple an abstraction from its implementation so that the two can vary independently*

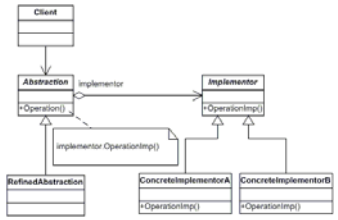


Applicability

- Use the Bridge pattern when…
  - You want to avoid a permanent binding between an abstraction and its implementation
  - Both the abstractions and their implementations should be extensible by subclassing (in this case, the pattern lets you extend them independently)
  - You have a proliferation of classes as shown in our example

Structure and participants

- Abstraction
  - Defines the abstraction's interface
  - Maintains a reference to an object of type Implementor
- RefinedAbstraction
  - Extends the interface defined by Abstraction
- Implementor
  - Defines the interface for implementation classes. The interface does not have to correspond exactly to Abstraction's interface; in fact the two can be quite different
- ConcreteImplementor
  - Implements the Implementor interface



Consequences

- Decoupling interface and implementation
  - An implementation is not bound permanently to an interface
  - The implementation of an abstraction can be configured at run-time
  - An object could even change its implementation at run-time!
- Improved extensibility
  - You can extend the Abstraction and Implementor hierarchies independently
- Hiding implementation details from clients