Software Design and Architecture

Flyweight and Proxy Patterns

Design principles

High-level principles

- **S**ingle Responsibility
- **O**pen/Closed
- **L**iskov Substitution Principle
- **I**nterface Segregation
- **D**ependency Inversion

Low-level principles

- Encapsulate what varies
- Program to interfaces, not implementations
- Favor composition over inheritance
- Strive for loose coupling

Flyweight

Behavioral Patterns
- observer
- strategy
- command
- template
- null object
- state
- iterator

Creational Patterns
- factory method
- abstract factory
- singleton
- builder

Structural Patterns
- decorator
- adapter
- façade
- composite
- bridge
- flyweight

Problem

You are to display a sequence of symbols that represent some ancient text.

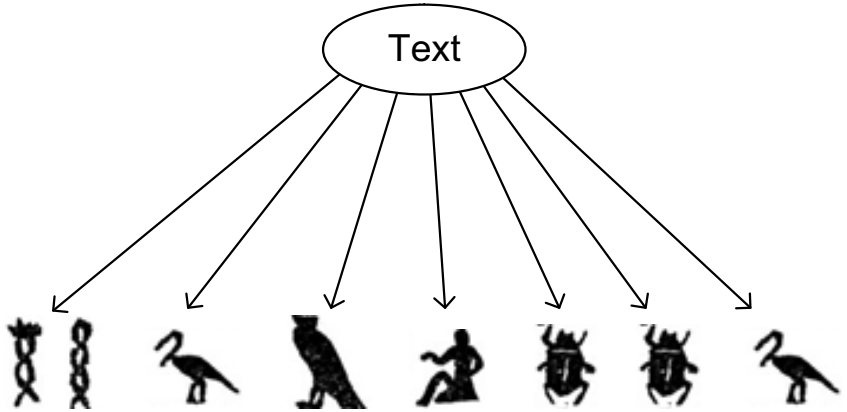
Issue: each symbol of the alphabet is *very* big not Unicode character.

Text

Alphabet

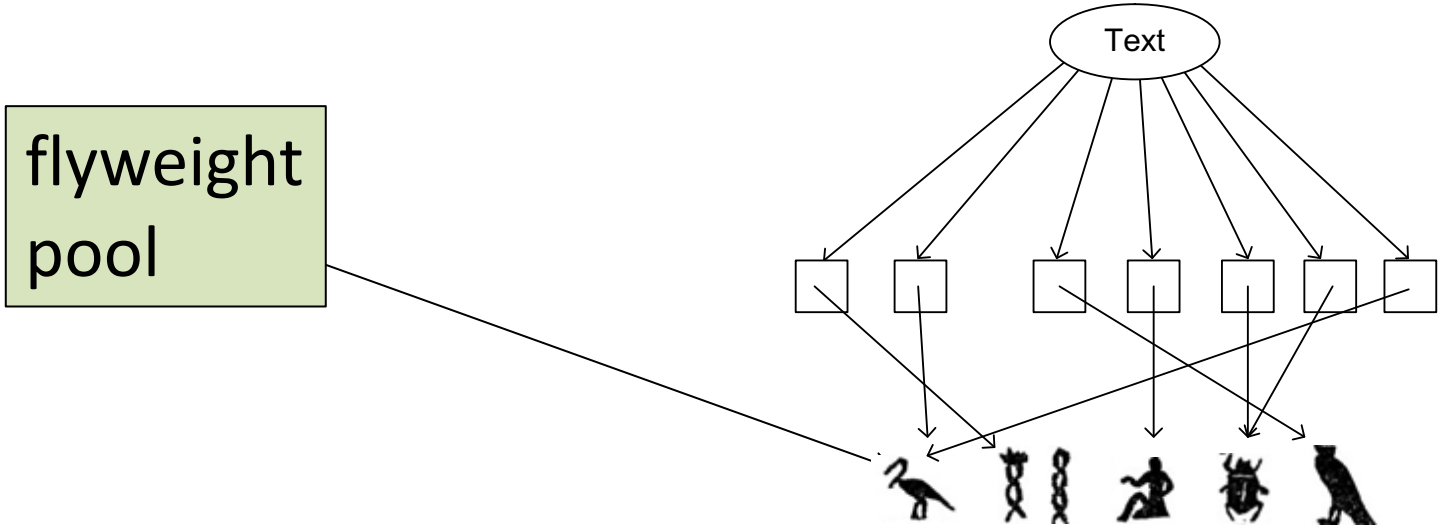
Problem

Conceptually, a text is a sequence of symbols.



Solution

However, you must implement it differently than a typical sequence **for efficiency**



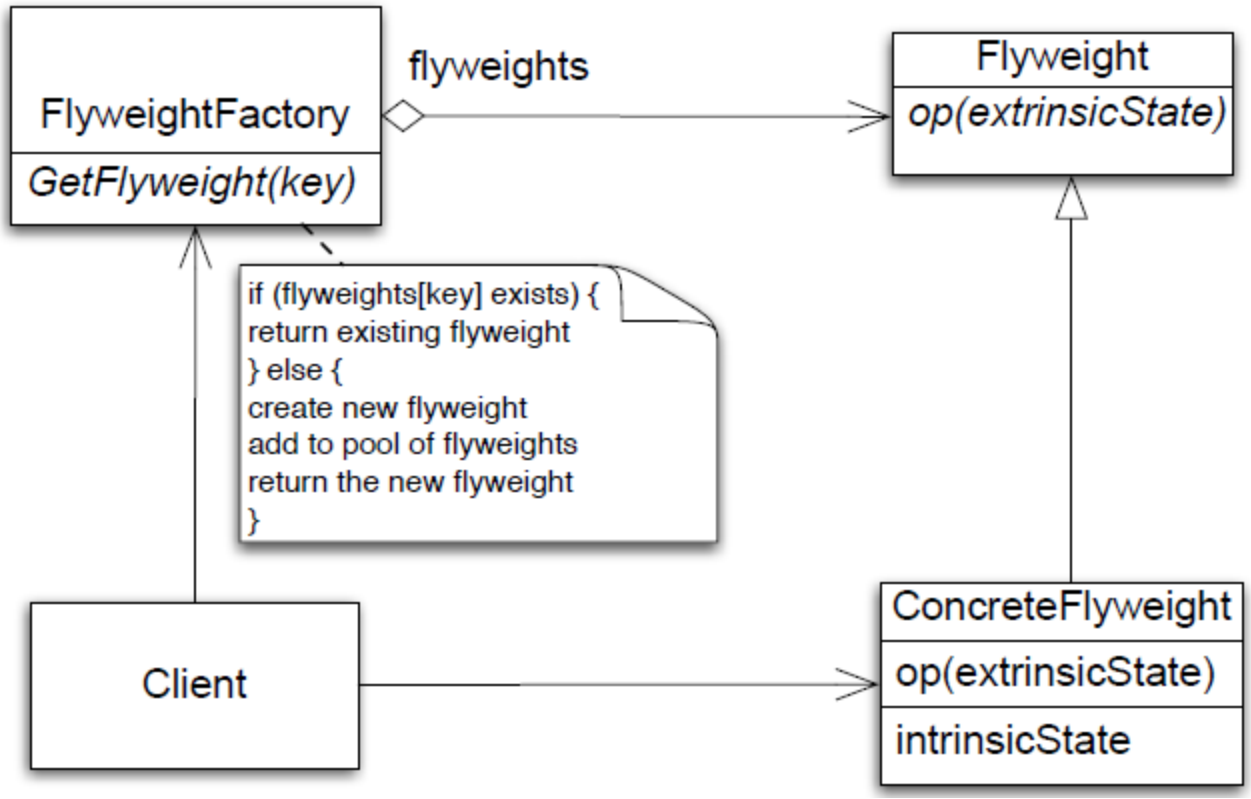
Idea of Flyweight

- Each object has **intrinsic** data and **extrinsic**
  - ex: characters in a String:
    - » extrinsic = index position
    - » intrinsic = value
- Idea of Flyweight is to separate intrinsic data from extrinsic
  - 2 classes now: Extrinsic + Intrinsic
  - extrinsic data can be supplied via calculation
  - intrinsic data is typically found in a file or database
- *Optionally, Intrinsic and Extrinsic objects are pooled*
  - *allocated from a pool and returned when no longer used*

Flyweight Pattern

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

Class Diagram



Flyweight Participants

Flyweight

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

ConcreteFlyweight

- implements Flyweight interface and adds storage for **intrinsic state**

FlyweightFactory

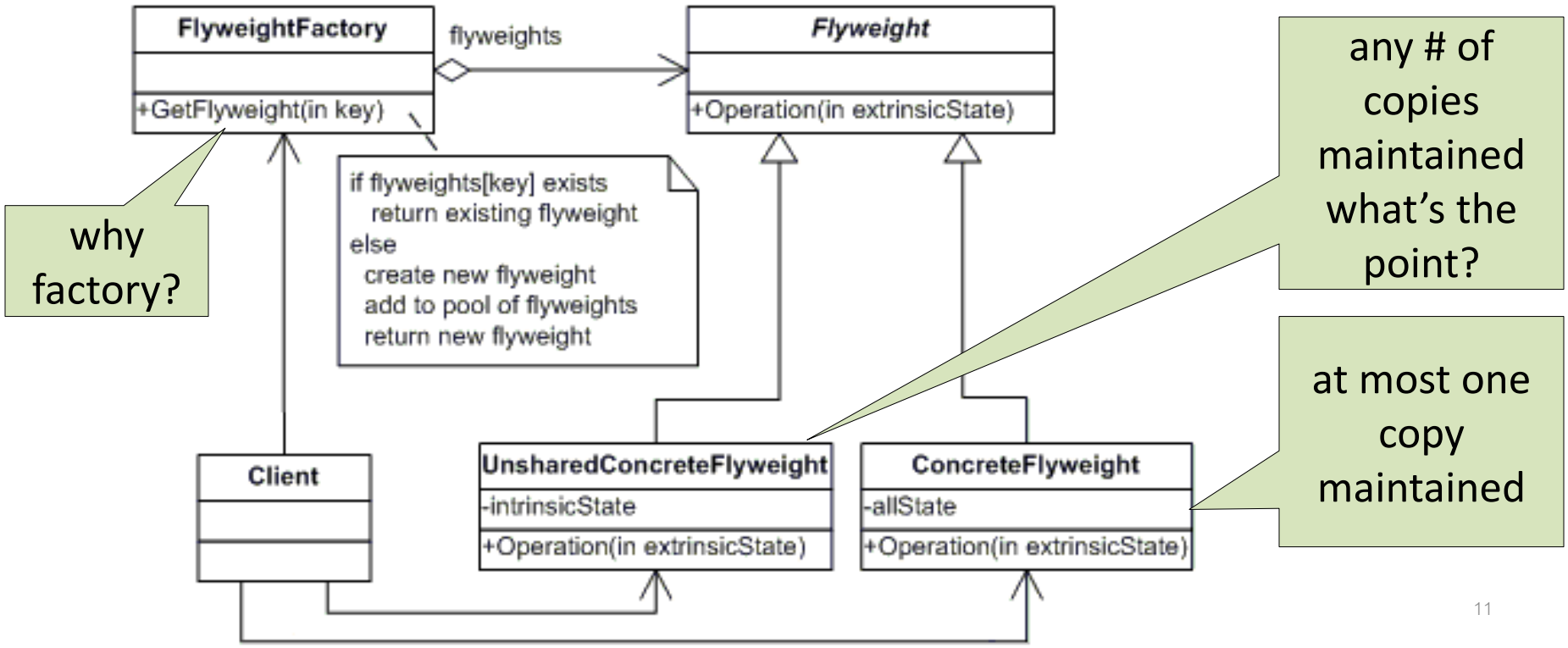
- creates and manages flyweight objects

Client

- maintains extrinsic state and stores references to flyweights

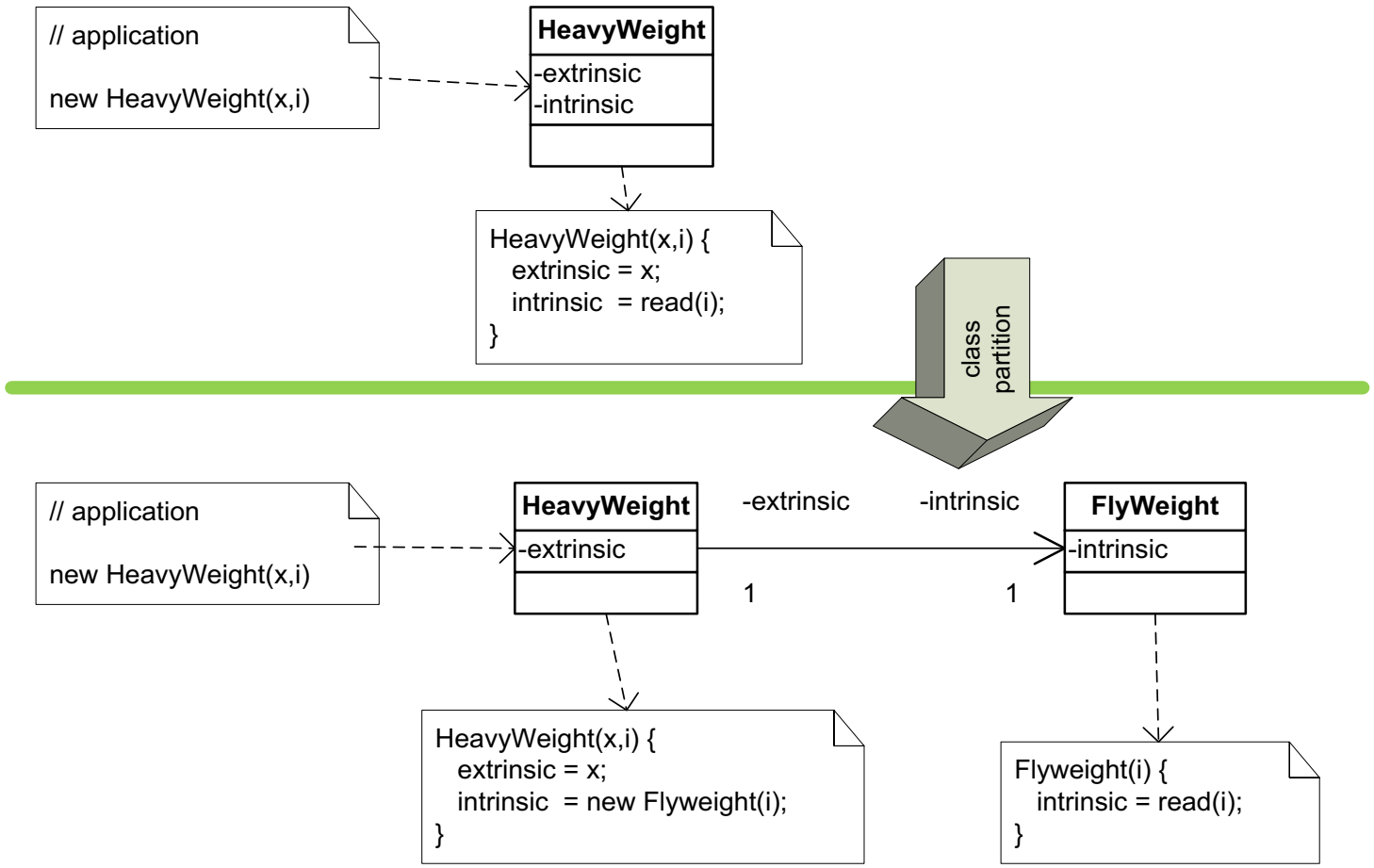
Flyweight Pattern

- There can be huge numbers of objects that share the same data
- **Idea:** split objects into sharable and non-sharable parts
- Design below looks awfully complicated. Where does it come from?



Derivation

- Flyweight class is partitioned away from an Original class



Example

We want to create a system to represent all of the cars in a city. You need to store the details about each car (model, and year) and the details about each car's ownership (owner name, tag number, last registration date).

```
class CarHeavyWeight {
        public  Car ( model, year, owner, tag, renewDate) {
                this.model = model;
                this.year = year;
                this.owner = owner;
                this.tag = tag;
                this.renewDate = renewDate;
         }

         //-------------------------------------------------------------
}
```

Example

Categorizing an object's data as intrinsic or extrinsic: the physical car data (model, year) is intrinsic, and the owner data (owner name, tag number, last registration date) is extrinsic. This means that only one car object is needed for each combination of model, and year.

```
/* Car class, optimized as a flyweight. */
class Car  {
      public Car ( model, year) {
            this.model = model;
            this.year = year;
      }

       //---------------------------------
}
```

Example

Instantiation Using a Factory: Factory ensures that only a single copy of each unique intrinsic state is created:

```
/* CarFactory singleton. */
class CarFactory {
        private Car createdCars [];
        public  Car  createCar( model, year) {
                // Check to see if this particular combination has been created
                // before. otherwise create a new instance and save it.
                if(! createdCars[ model + '-' + year]) {
                        var car = new Car(model, year);
                        createdCars[ model + '-' + year] = car;
                }
                 return createdCars[model + '-' + year];
        }
        //---------------------------------------------------------------
}
```

Example

Extrinsic State Encapsulated in a Manager : All of the data that are not included in the Car objects has to be stored somewhere (CarRecord); you use a Singleton as a manager to encapsulate that data.

```
/* CarRecordManager singleton. */
class CarRecordManager {
        private  CarRecord  carRecordDatabase [];
        // Add a new car record into the city's system.
        public  void addCarRecord
                                (model, year, owner, tag, renewDate) {
                car = CarFactory.createCar( model, year);
                carRecordDatabase[tag] = {
                        owner: owner,
                        renewDate: renewDate,
                        car: car
                }
        }
        //-------------------------------------------------------
}
```

Flyweight Participants

Collaborations

- Data that a flyweight needs to process must be classified as intrinsic or extrinsic
  - Intrinsic is stored with flyweight; Extrinsic is stored with client
- Clients should not instantiate ConcreteFlyweights directly

Consequences

- Storage savings is a tradeoff between total reduction in number of objects verses the amount of intrinsic state per flyweight and whether or not extrinsic state is computed or stored
  - greatest savings occur when extrinsic state is computed

When to Use Flyweight

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 does not depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects

Proxy

Behavioral Patterns
- observer
- strategy
- command
- template
- state
- iterator

Creational Patterns
- factory method
- abstract factory
- singleton
- builder

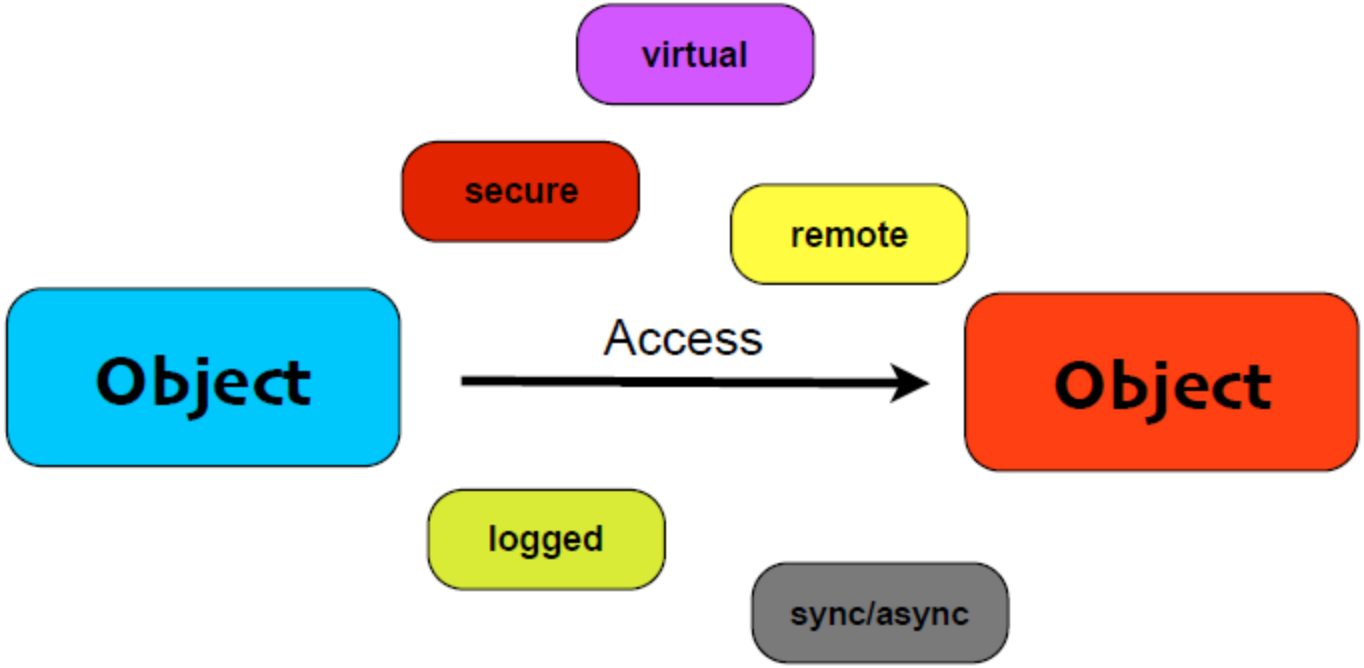
Structural Patterns
- decorator
- adapter
- façade
- composite
- bridge
- flyweight
- **proxy**

Problem

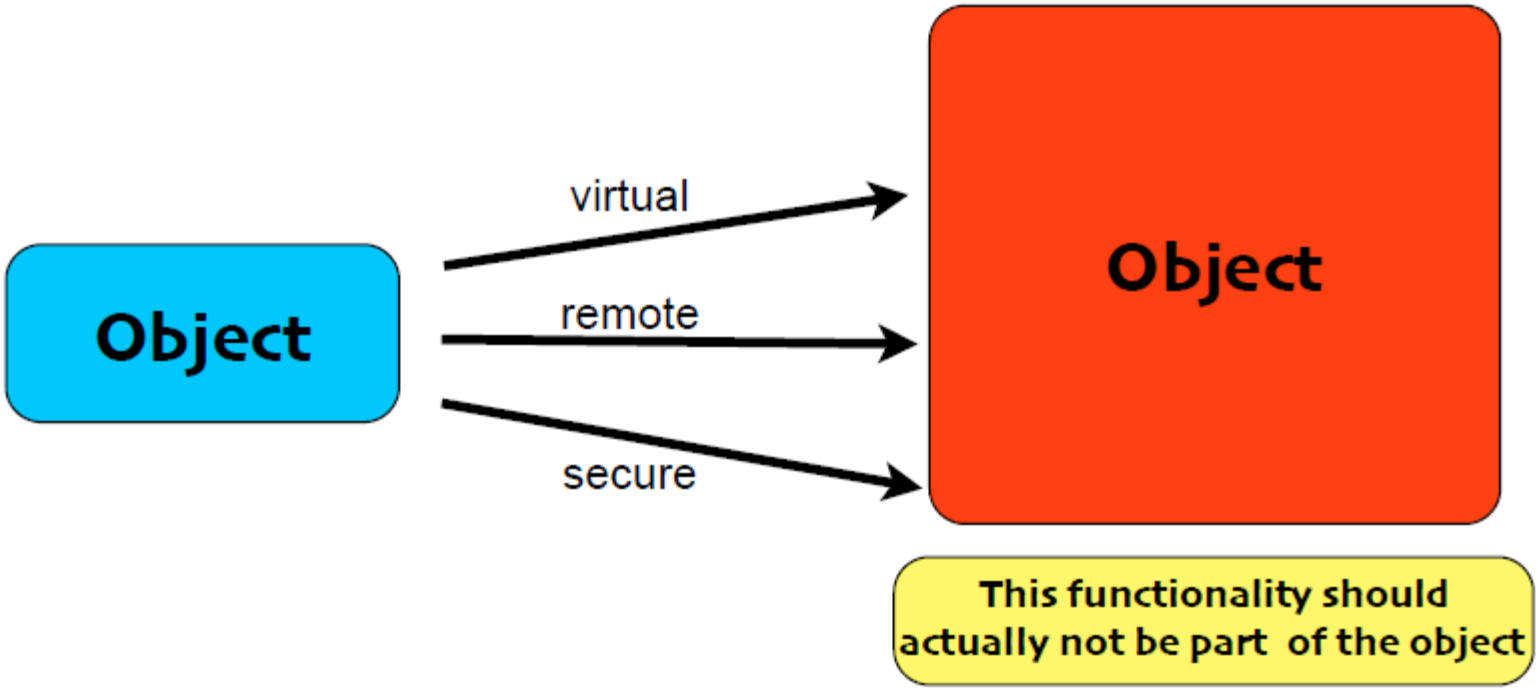
Objects talk to each other using an interface that has been overburdened with the needs of networking, security, access coherence, or historic versions of the interface.

an object demands too much of another object

Object Access Control



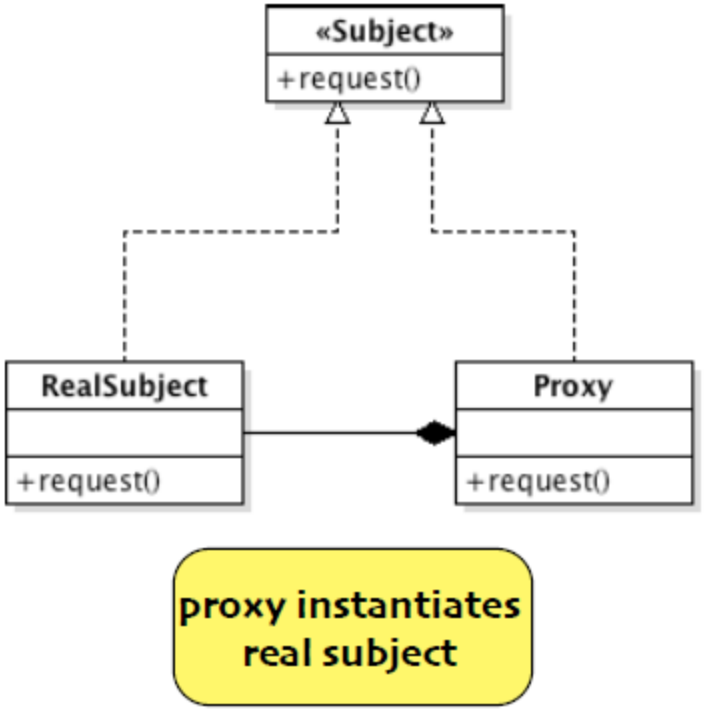
Object Access Control



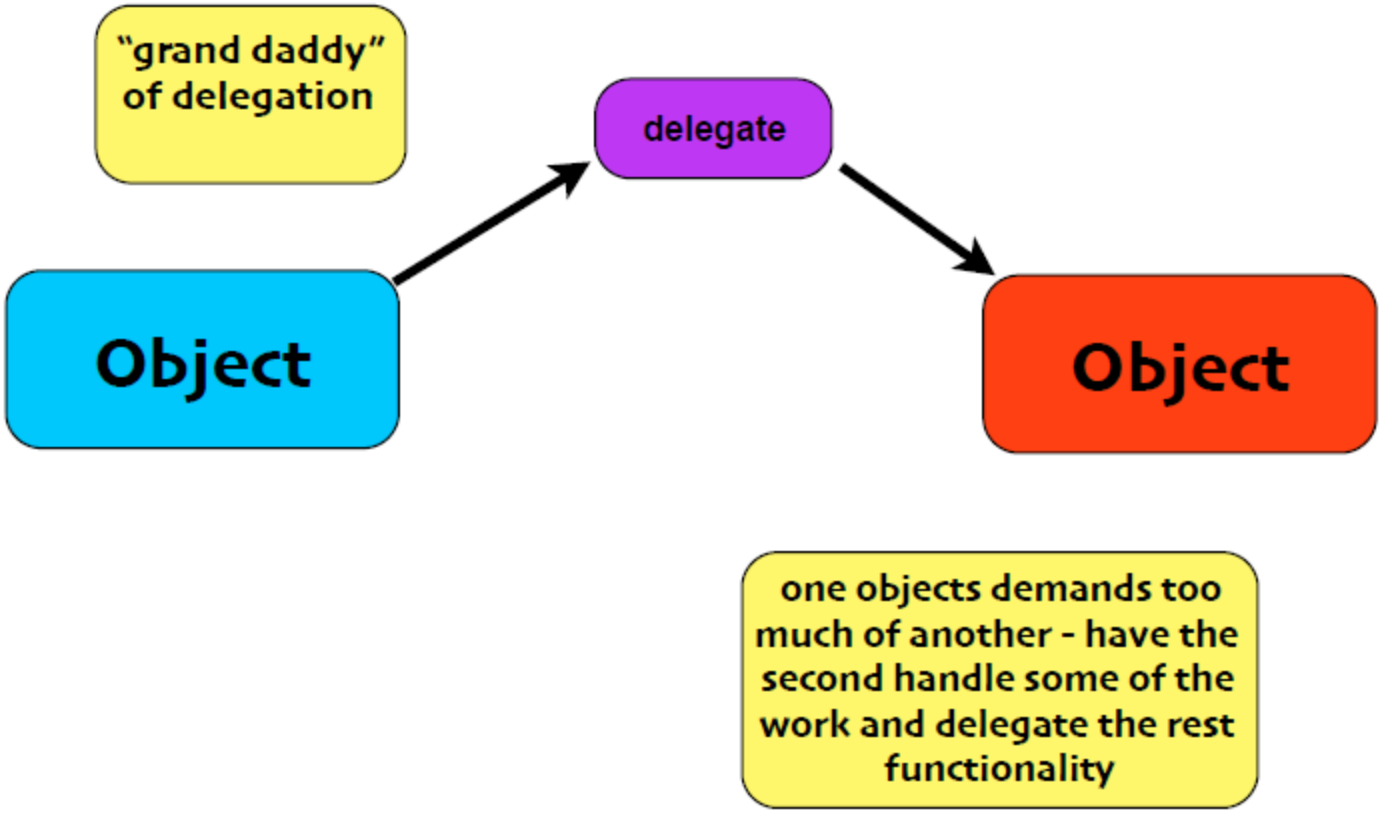
The Proxy Pattern

The Proxy Pattern provides a surrogate or placeholder for another object to control access to it.

Class Diagram



The Proxy Pattern



Types of proxies

- **Remote Proxy -** Provides a reference to an object located in a different address space
- **Virtual Proxy -** Allows the creation of a memory intensive object on demand
- **Protection Proxy -** provides different clients with different levels of access
- **Cache Proxy -** Temporary stores results of expensive target operations
- **Fire Wall Proxy -** protects targets from bad clients
- **Synchronization proxy -** provides multiple access to a target object
- **Smart reference Proxy -** counting the number of references to an object

Proxy Pattern: Structural Code

```
/**
 * Test driver for the pattern.
 */
public class Test {
            public static void main( String arg[] ) {
                 Subject real = new RealSubject();
                 Proxy proxy = new Proxy();
                 proxy.setRealSubject( real );
                 proxy.request();
                 }
}

/**
 * Defines the common interface for RealSubject and Proxy so that
 * a Proxy can be used anywhere a RealSubject is expected.
 */
public interface Subject {
            void request();
}

/**
 * Defines the real object that the proxy represents.
 */
public class RealSubject implements Subject {
            public void request() {
                        // Do something based on the interface.
            }
}
```

```
/**
 * 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 by substituted for the real subject.
 */
public class Proxy implements Subject {
            private Subject realSubject;
            public void setRealSubject( Subject subject ) {
                        realSubject = subject;
            }
            public Subject getRealSubject() {
                        // This may not be possible if the
                        // proxy is communicating over a network.
                        return realSubject;
            }
            public void request() {
                        // This is very simplified.
                        //
                        // This could actually be a call to a different
                        // run-time environment locally, a machine over a
                        // TCP socket, or something completely different.
                        realSubject.request();
            }
}
```

Proxy vs Decorator

- Proxy is structurally similar to Decorator, but the two differ in their purpose.
- The Decorator Pattern adds behavior to an object, while a proxy controls access to it.
- Like any wrapper, proxies will increase the number of classes and objects in your design.