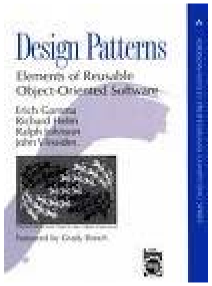
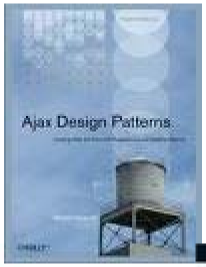
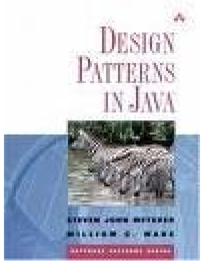
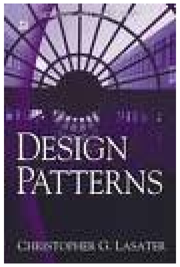
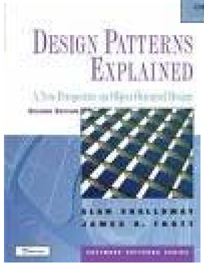
Design Patterns

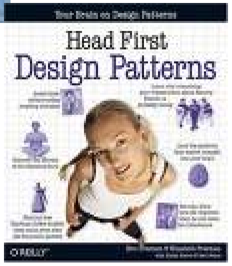












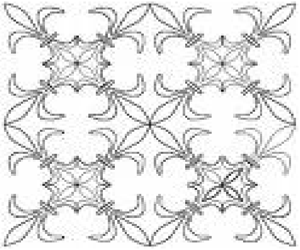
CSE 403, Spring 2008, Alverson

With material from Marty Stepp 403 lectures.

Design challenges

- Designing software is hard! One must find:
    - a good problem decomposition
    - a design with flexibility, modularity and elegance
- Designs often emerge from trial and error
- Successful designs do exist
    - two designs they are almost never identical
    - they exhibit some recurring characteristics

Design patterns



A *design pattern* is a time-tested solution to a common software problem

- Patterns enable a common design vocabulary, improving communication, easing implementation and documentation
- Patterns capture design expertise and allow that expertise to be transferred

Online Readings

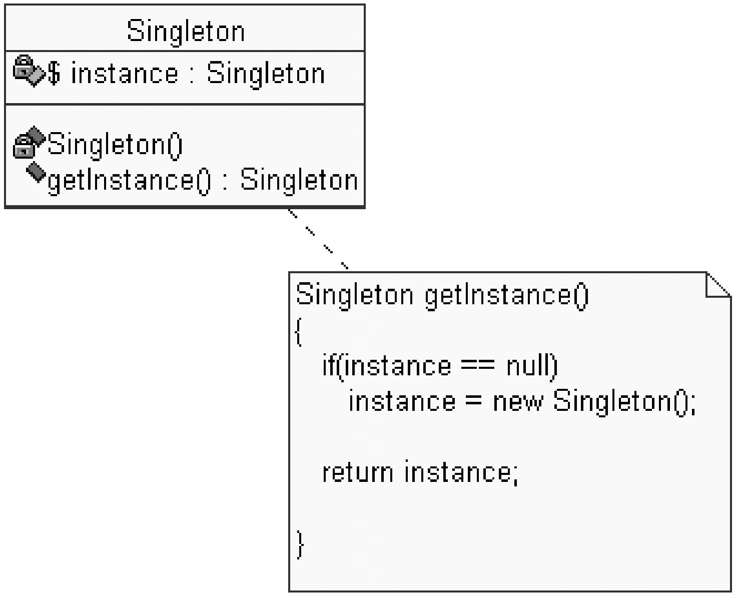
- My latest favorite survey of common patterns: http://sourcemaking.com/design_patterns
- [Optional] See the “References” link on the class web page for a number of others

Gang of Four (GoF) patterns

- Creational Patterns (abstract object instantiation)
  Abstract Factory, Factory, Builder, Prototype
  Singleton
- Structural Patterns (combine objects)
  Adapter, Bridge, Composite, Decorator, Façade,
  Flyweight, Proxy
- Behavioral Patterns (communication btwn objects)
  Chain of responsibility, Command, Interpreter,
  Iterator, Mediator, Memento, Observer, State,
  Strategy, Template Method, Visitor

Pattern: Singleton

a class that has only one instance



```
Singleton getInstance()
{
   if(instance == null)
      instance = new Singleton();

   return instance;

}
```

Restricting object creation

- Problem: Sometimes we will really only ever need one instance of a particular class.
    - We'd like to make it illegal to have more than one
    - Examples: keyboard reader, printer spooler, gradebook

- Why we care:
    - Creating lots of objects can take a lot of time
    - Extra objects take up memory
    - It is a maintenance headache to deal with different objects floating around if they are the same

Singleton pattern

- singleton: an object that is the only object of its type
  - Ensures that a class has at most one instance
  - Provides a global access point to that instance

  - Takes responsibility of managing that instance away from the programmer (illegal to construct more instances)

  - Provide accessor method that allows users to see the (one and only) instance

  - Possibly the most known / popular design pattern!

Singleton pattern

- singleton: an object that is the only object of its type
  - Ensures that a class has at most one instance
  - Provides a global access point to that instance

  - Takes responsibility of managing that instance away from the programmer (illegal to construct more instances)

  - Provide accessor method that allows users to see the (one and only) instance

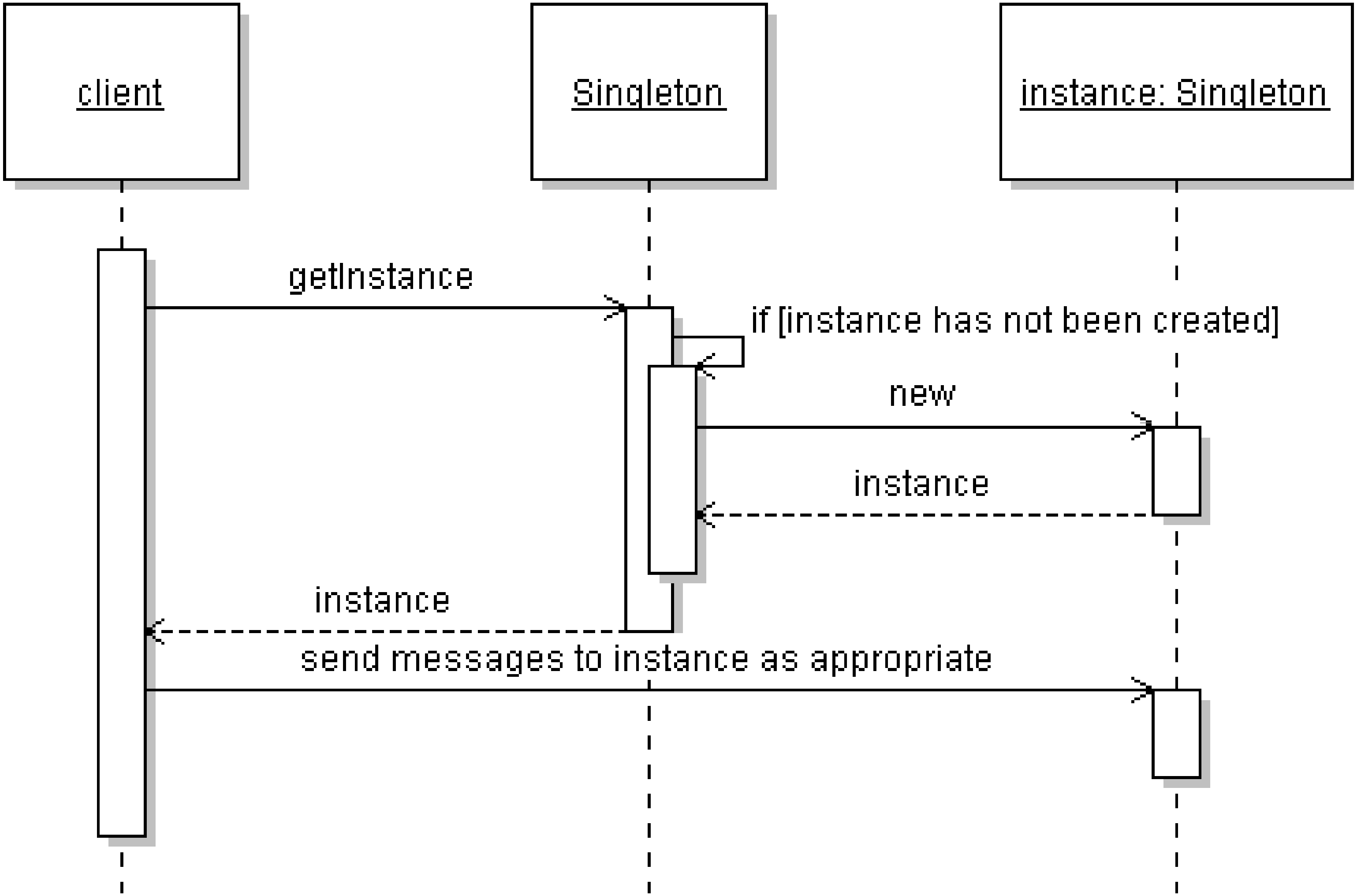
  - Possibly the most known / popular design pattern!

How is this different from a global variable?

Implementing singleton (one instantiation of the pattern…)

- Make constructor(s) **`private`** so that they can not be called from outside
- Declare a single **`static private`** instance of the class
- Write a public **`getInstance()`** or similar method that allows access to the single instance
  - possibly protect / synchronize this method to ensure that it will work in a multi-threaded program

Singleton sequence diagram



Singleton example

- Consider a singleton class RandomGenerator that generates random numbers

```
public class RandomGenerator {
 private static RandomGenerator gen = new RandomGenerator();

   public static RandomGenerator getInstance() {
      return gen;
   }

   private RandomGenerator() {}
     ...
}
```

- Is there a problem with this class?

Singleton example 2

- Variation: don't create the instance until needed

```
public class RandomGenerator {
    private static RandomGenerator gen = null;

    public static RandomGenerator getInstance() {
        if (gen == null) {
            gen = new RandomGenerator();
        }
        return gen;
    }

    ...
}
```

- What could go wrong with this version?

Singleton example 3

- Variation: solve concurrency issue by locking

```
public class RandomGenerator {
    private static RandomGenerator gen = null;

    public static synchronized RandomGenerator
                                                  getInstance() {
        if (gen == null) {
            gen = new RandomGenerator();
        }
        return gen;
    }

    ...
}
```

- Is anything wrong with this version?

Singleton example 4

- Variation: solve concurrency issue without unnecessary locking

```
public class RandomGenerator {
   private static RandomGenerator gen = null;

   public static RandomGenerator getInstance() {
      if (gen == null) {
         synchronized (RandomGenerator.class) {
            // must test again -- can you see why?
            // sometimes called test-and-test-and-set
            if (gen == null) {
               gen = new RandomGenerator();
            }
         }
      }
      return gen;
} }
```

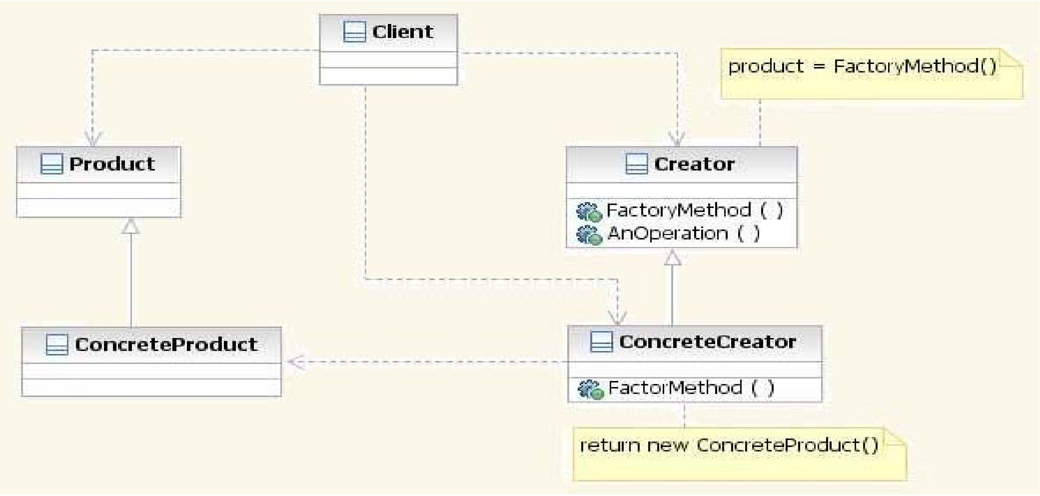
Singleton exercise

- Consider your projects. What classes could be a singleton in this system?

Pattern: Factory

(a variation of Factory Method, Abstract Factory)

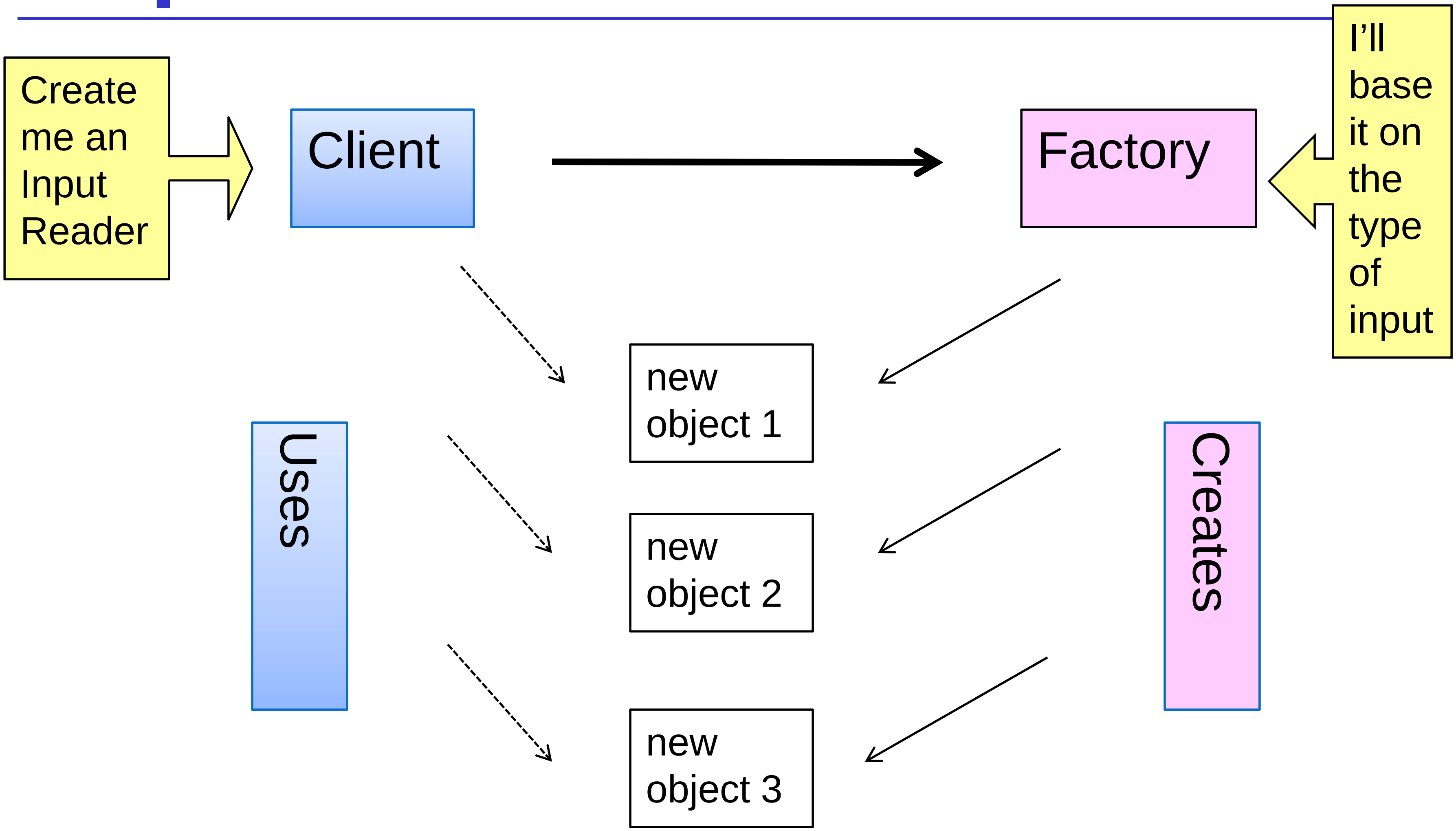
a class or method used to create objects easily



Factory pattern

- factory: a class whose job is to easily create and return instances of other classes
  - Instead of calling a constructor, use a static method in a "factory" class to set up the object
  - Allows you to separate the construction information from the usage information (improve cohesion, loosen coupling), making creation and management of objects easier
  - Allows you to defer instantiation of the subclass

Separate creation from use



Create me an Input Reader

Client

Factory

I'll base it on the type of input

new object 1

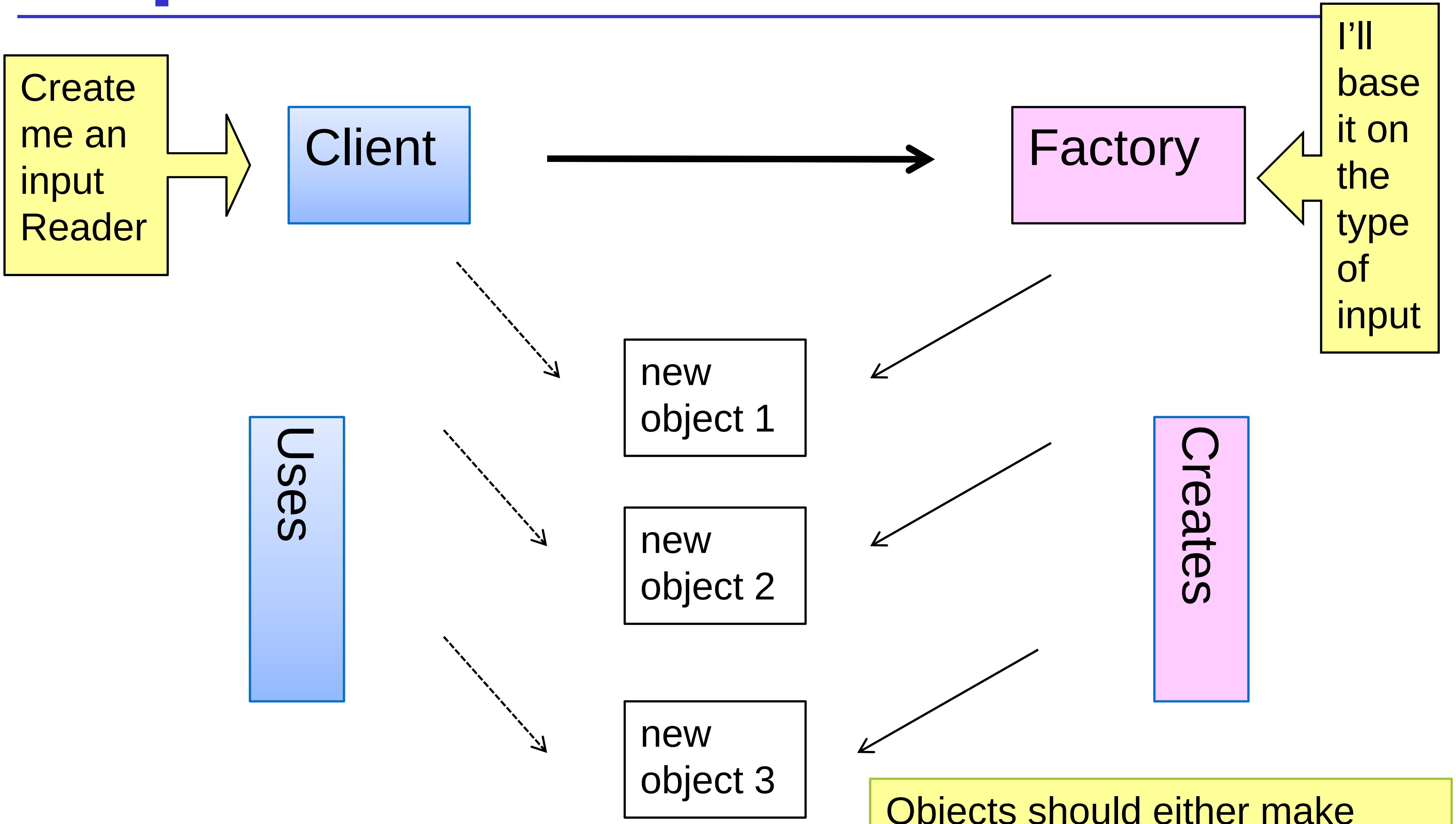
new object 2

new object 3

Uses

Creates

Separate creation from use



Create me an input Reader

Client

Factory

I'll base it on the type of input

new object 1

new object 2

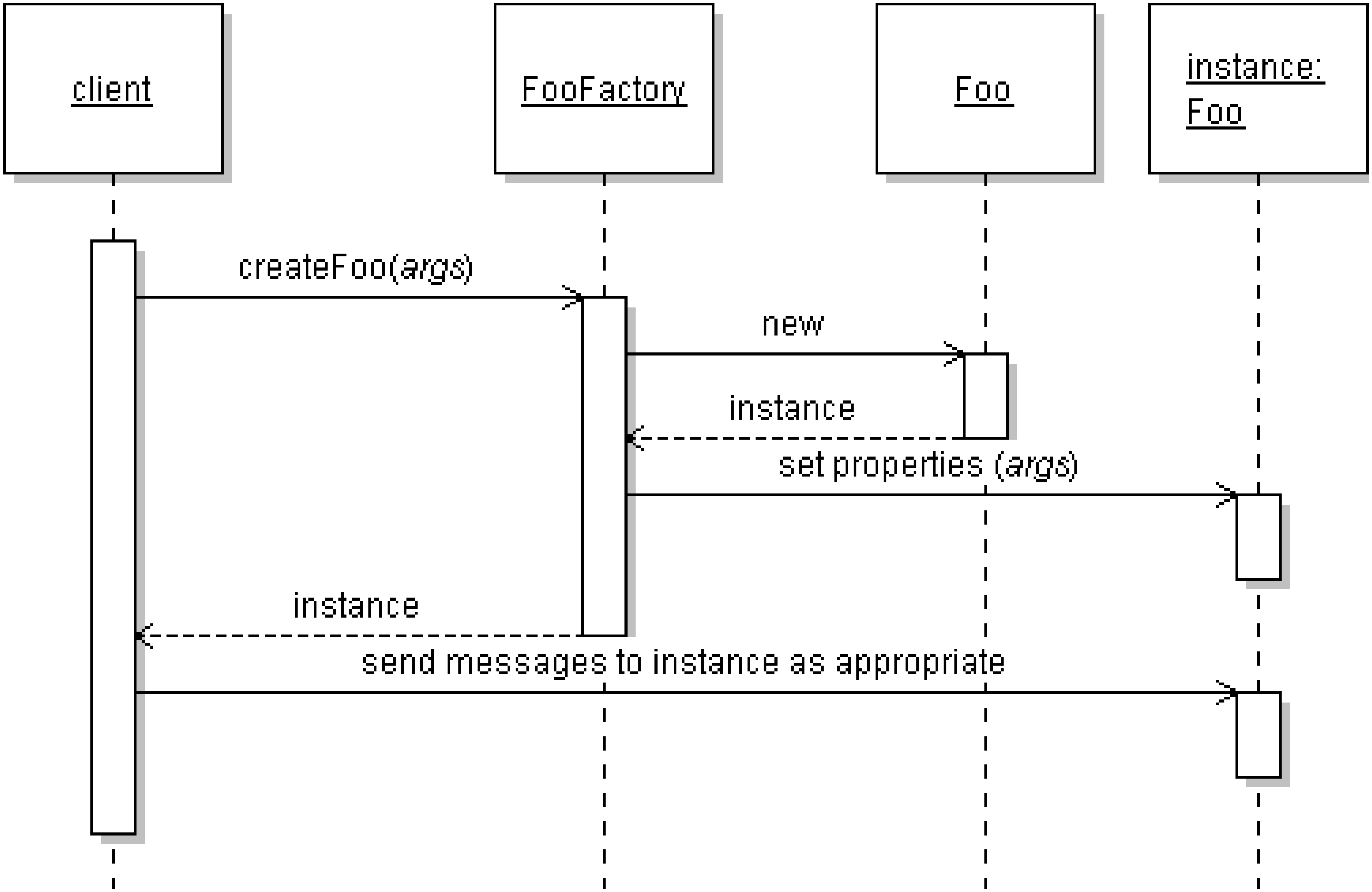
new object 3

Uses

Creates

Objects should either make other objects or use other objects but never both.

Factory sequence diagram



Factory implementation

When implementing a factory of your own, here's one scheme:

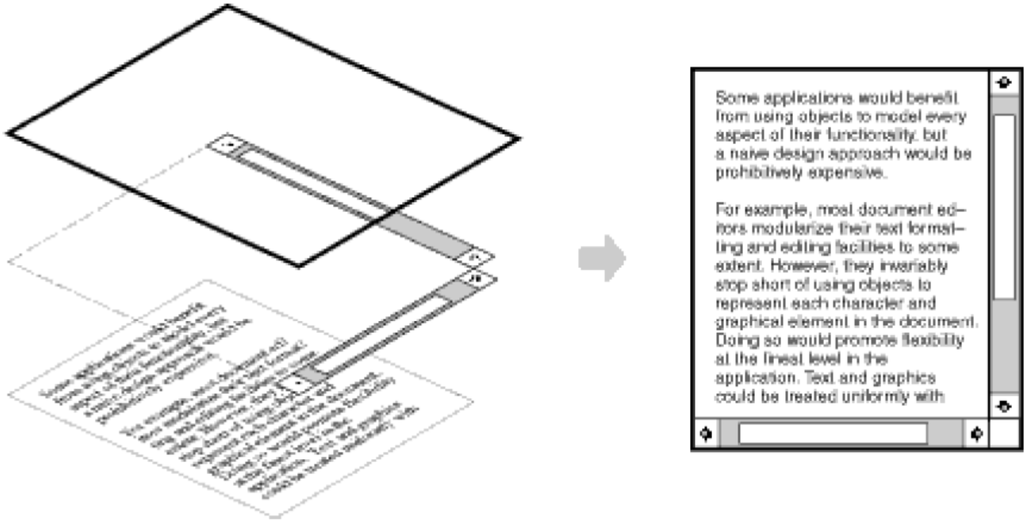
- The factory itself should not be instantiated
  - make constructor `private`
- The factory uses `static` methods to construct components
- The factory should offer as simple an interface to client code as possible

Factory example

```
public class ImageReaderFactory
{
    public static ImageReader createImageReader(
    InputStream is ) {
     int imageType = figureOutImageType( is );

     switch( imageType ) {
           case ImageReaderFactory.GIF:
                      return new GifReader( is );
           case ImageReaderFactory.JPEG:
                      return new JpegReader( is ); // etc.
     }
     }
}
```

Pattern: Decorator

objects that wrap around other objects to add useful features



Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with

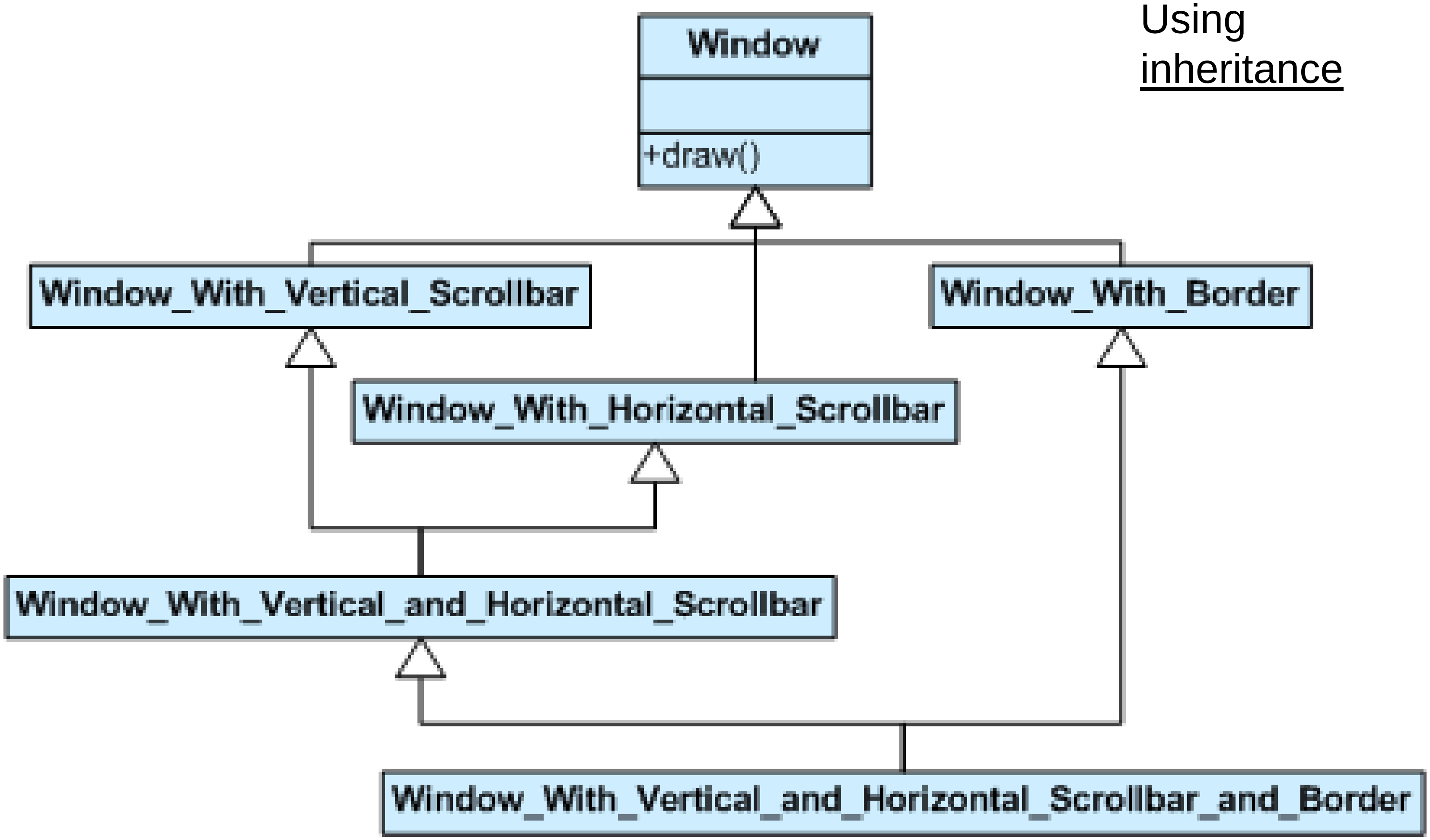
+ scrollbars

Decorator pattern

- decorator: an object that modifies behavior of, or adds features to, another object
  - Adds additional responsibilities to an object dynamically
  - The object being "decorated" usually does not explicitly know about the decorator
  - Decorator must maintain the common interface of the object it wraps up

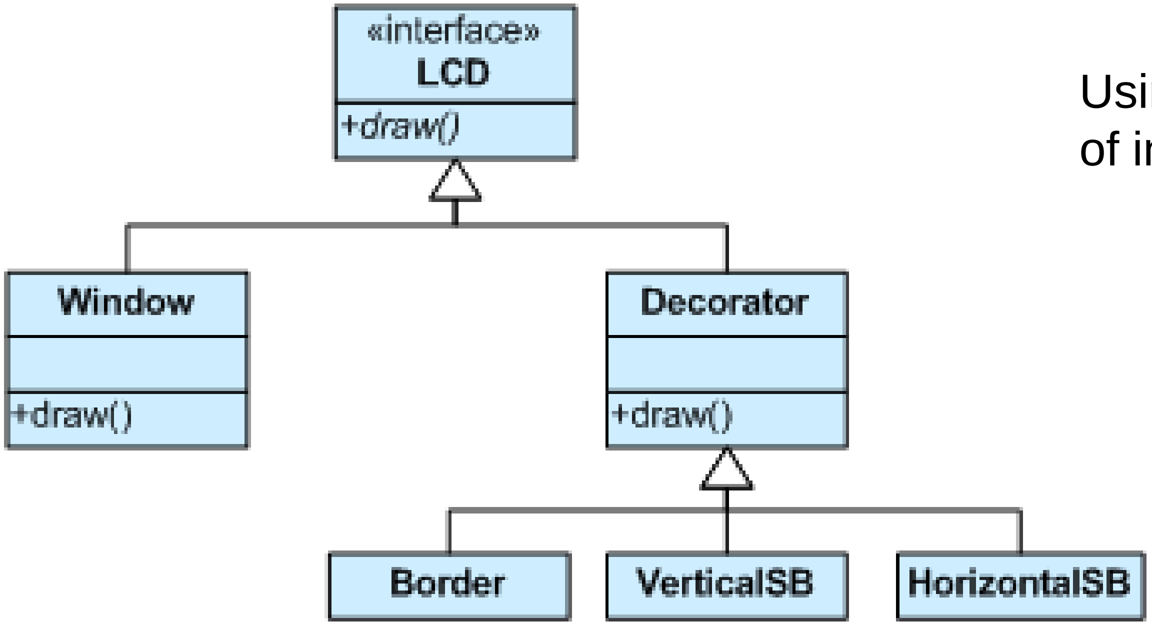
What are two ways in which this differs from inheritance?

Decorator example: GUI



Using
<u>inheritance</u>

Using decorator objects



Using aggregation instead of inheritance

```
Widget* aWidget = new BorderDecorator(
  new HorizontalScrollBarDecorator(
  new VerticalScrollBarDecorator(
    new Window(80, 24))));
aWidget->draw();
```

Another decorator example: I/O

- InputStream class has only public int read() method to read one letter at a time

- Decorators such as BufferedReader add additional functionality to read the stream more easily

```
InputStream in = new FileInputStream("hardcode.txt");
InputStreamReader isr = new InputStreamReader(in);
BufferedReader br = new BufferedReader(isr);
// InputStream only provides public int read()
String wholeLine = br.readLine();
```

Pattern: Facade

provide a uniform interface to a set of other (alternative) interfaces
or
wrap a complicated interface with a simpler one

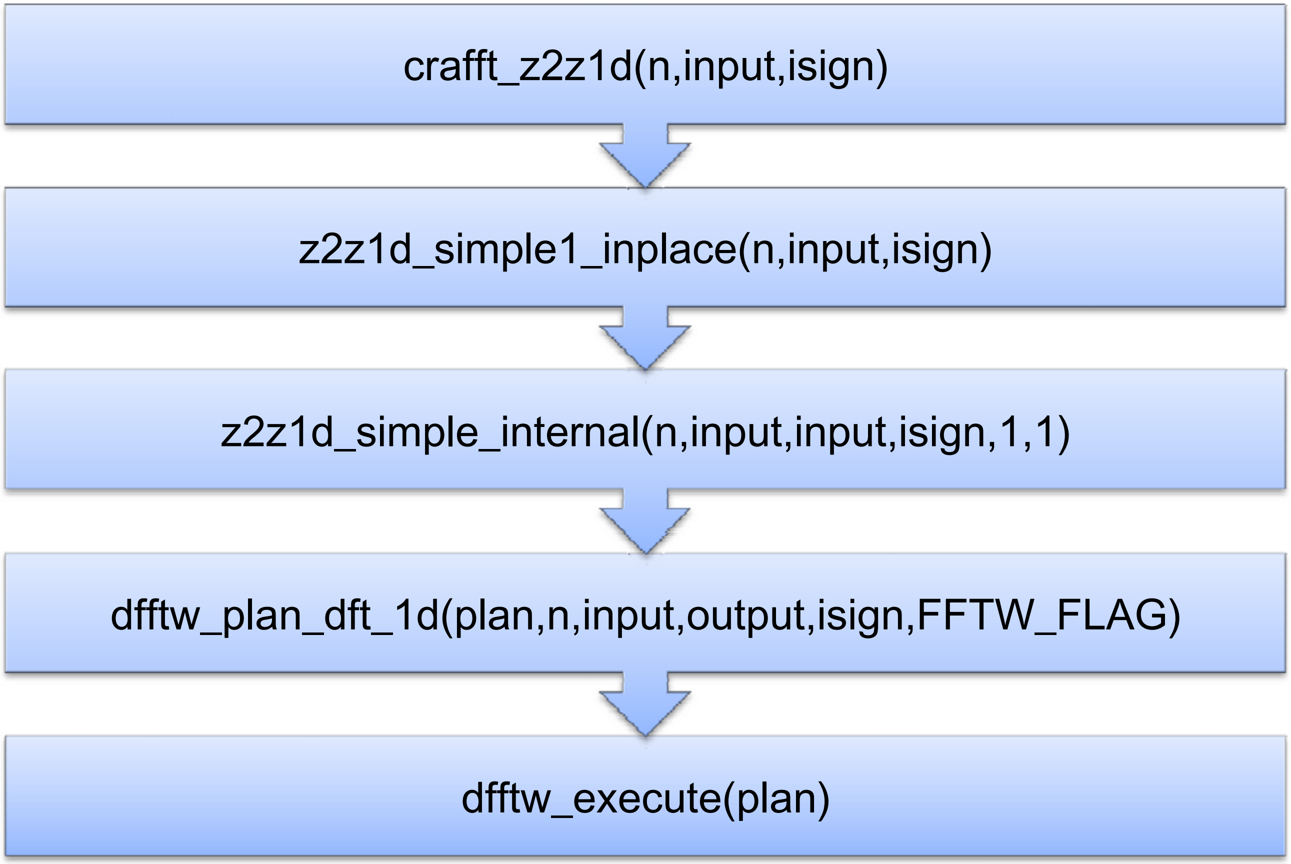
Facade pattern

- Problem: a current interface is too complicated to easily use OR there are several choices to use for a subsystem; we want to allow the use of either

- facade: *objects that provide a uniform interface to a complicated or set of other alternative interfaces*

Examples from Cray:
MySQL package or PostgreSQL package
FFT math library from Fast FFT or FFTW or …

Cray CRAFFT library example



crafft_z2z1d(n,input,isign)

z2z1d_simple1_inplace(n,input,isign)

z2z1d_simple_internal(n,input,input,isign,1,1)

dfftw_plan_dft_1d(plan,n,input,output,isign,FFTW_FLAG)

dfftw_execute(plan)

Cray CRAFFT library example

crafft_z2z2d(n1,n2,input,ld_in,output,ld_out,isign,work)

z2z2d_adv1(n1,n2,input,ld_in,output,ld_out,isign,work)

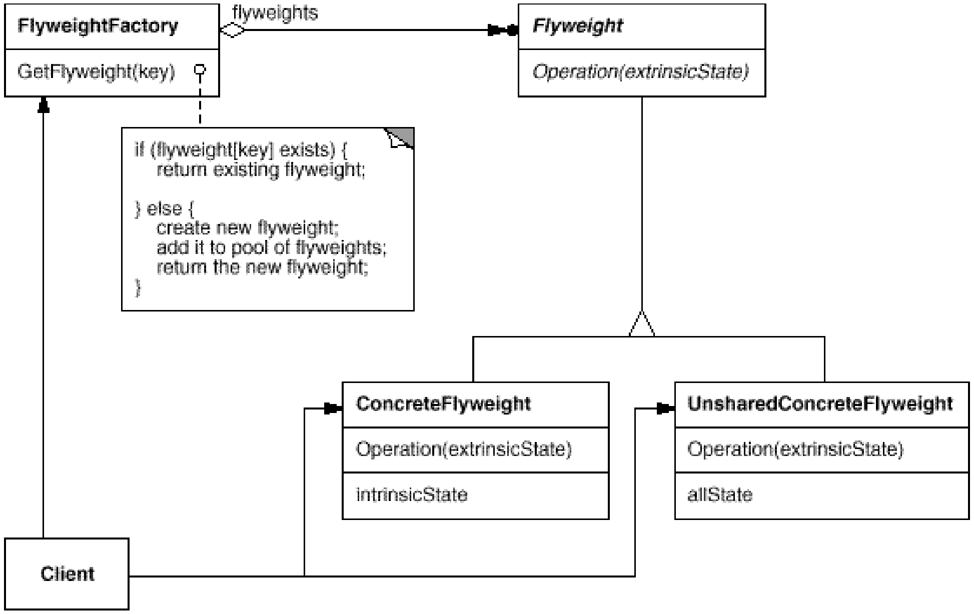
z2z2d_adv_internal(n1,n2,input,ld_in,output,ld_out,isign,1,1,work)

dfftw_plan_many_dft(plan,rank,n,howmany,input,inembed,istride,idist,output,onembed,ostride,odist,isign,FFTW_FLAG)

dfftw_execute(plan)

Pattern: Flyweight

a class that has only one instance for each unique state



Problem of redundant objects

- Problem: redundant objects can be inefficient
  - Many objects have same state

  - Example: string/text structures used by document editors, error messages

  - Example: File objects that represent the same file on disk
    - **new File("notes.txt")**
    - **new File("notes.txt")**
    - **new File("notes.txt")**

    Or point objects that represent points on a grid
    - **new Point(x,y)**
    - **new Point(5.23432423, 3.14)**

Why can't this be solved by using a const?

Flyweight pattern

- flyweight: an assurance that no more than one instance of a class will have identical state
  - Achieved by caching identical instances of objects to reduce object construction
  - Similar to singleton, but has many instances, one for each unique-state object
  - Useful for cases when there are many instances of a type but many are the same

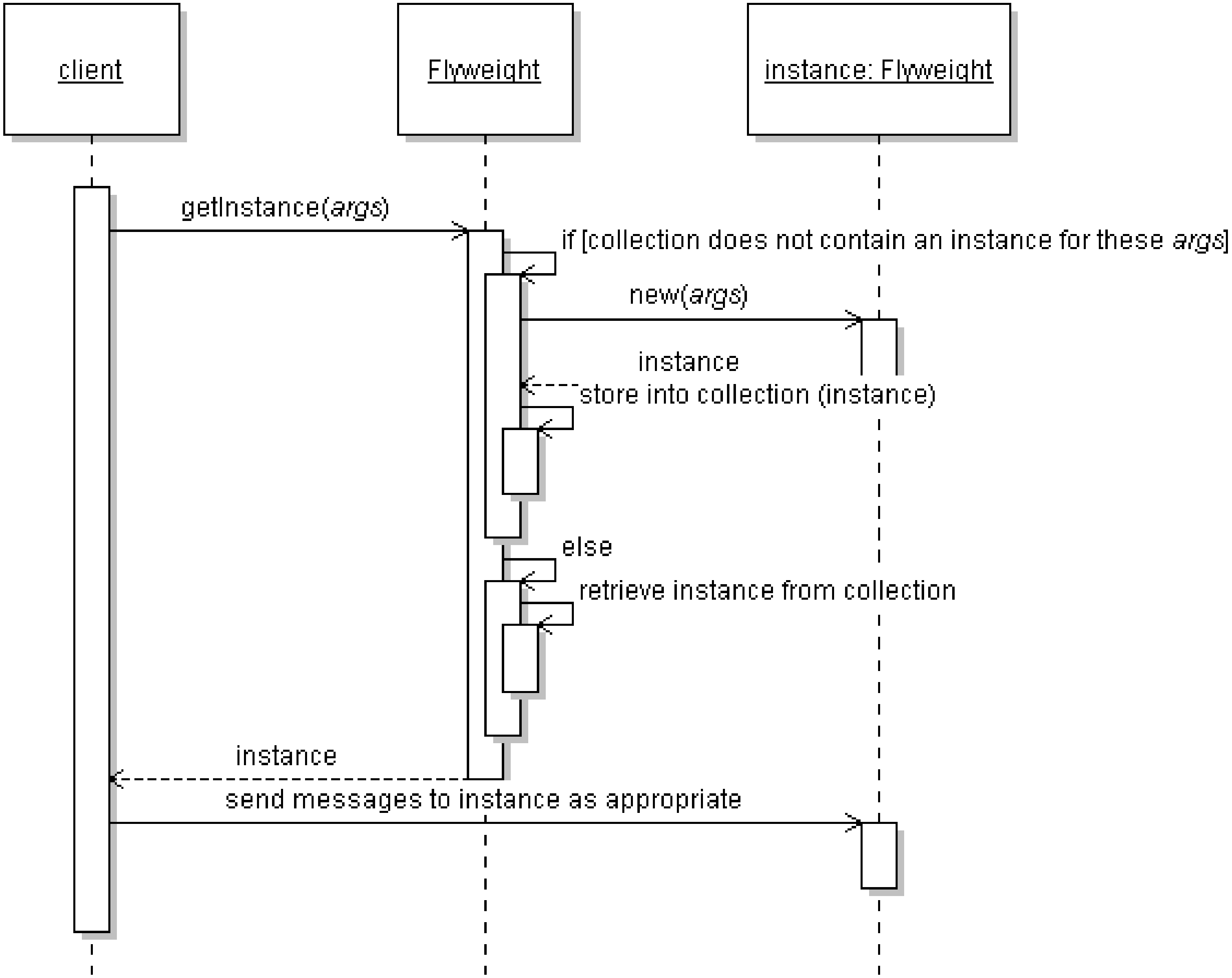
Implementing a Flyweight

- Flyweighting works best on *immutable* objects

pseudo-code:

```
public class Flyweighted {
    o static collection (list) of instances
    o private constructor
    o static method to get an instance:
        ☐ if (we have created this kind of instance before),
          get it from the collection and return it
        ☐ else,
          create a new instance, store it in the collection and
          return it
}
```

Flyweight sequence diagram



Implementing a Flyweight

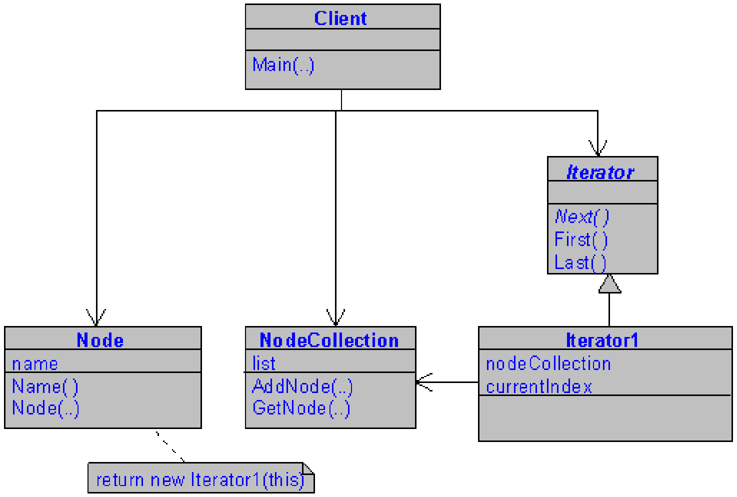
```
public class Flyweighted {
    private static Map instances;

    private Flyweighted() {}

    public static synchronized Flyweighted
                          getInstance(Object key) {
       if (!instances.contains(key)) {
           Flyweighted fw = new Flyweighted(key);
           instances.put(key, fw);
           return fw;
       } else {
           return instances.get(key);
       }
   }
}
```

Flyweight exercise

- Consider your projects. Is there an opportunity to use a flyweight?

Pattern: Iterator

objects that traverse collections



Iterator pattern

- iterator: an object that provides a standard way to examine all elements of any collection
- Benefits:

Iterators in Java

- All Java collections have a method `iterator` that returns an iterator for the elements of the collection
- Can be used to look through the elements of any kind of collection (an alternative to for loop)

```
List<Account> list = new ArrayList<Account>();
// ... add some elements ...

for (Iterator<Account> itr = list.iterator(); itr.hasNext(); ) {
   Account a = itr.next();
   System.out.println(a);
}
```

Adding your own iterators

- When implementing your own collections, it can be convenient to use iterators.

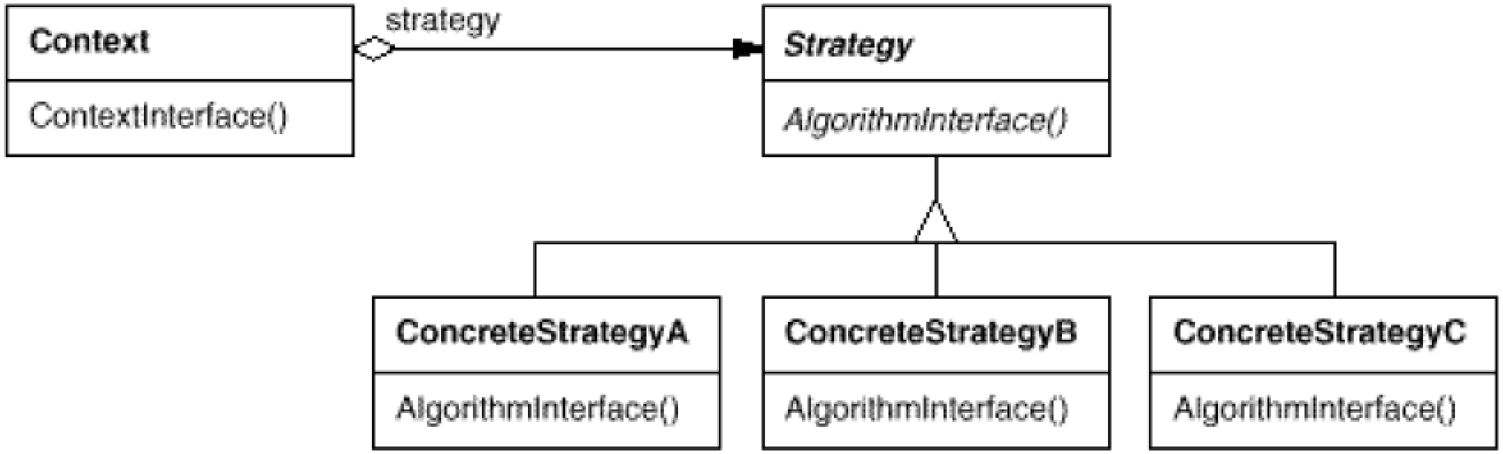
```
class List {
        public:
                  int size() {…}
       boolean isEmpty() {…}
                  ListElement* get(int index) {…}
   }


  public class ListIterator {
                  int currentIndex;
        public:
                  boolean hasNext() {…}
                  ListElement* first() {…}
                  ListElement* next() {…}
                  ListElement* current() {…}
      }
```

What do you need to know to write next()?

Can there be different iteration strategies?

Pattern: Strategy

objects that hold alternate algorithms to solve a problem



Strategy pattern

- strategy: an algorithm separated from the object that uses it, and encapsulated as its own object
  - Each strategy implements one behavior, one implementation of how to solve the same problem
  - Separates algorithm for behavior from object that wants to act
  - Allows changing an object's behavior dynamically without extending / changing the object itself
- Examples?

Strategy example: Card player

```
// Strategy hierarchy parent
// (an interface or abstract class)
public interface Strategy {
  public Card getMove();
}

// setting a strategy
player1.setStrategy(new SmartStrategy());
// using a strategy
Card p1move = player1.move();  // uses strategy
```

All strategies must declare (the same) interface common to all supported algorithms

Selecting a design pattern

- Consider how design patterns solve design problems
  - You’ll need to get familiar with them first

- Consider design patterns of similar purpose to select the one that best fits your situation
  - Creational
  - Structural
  - Behavioral

- Consider the aspects of your system most likely to change, evolve, be reused

Think of an example of where you could apply a pattern to your project.