Structural Patterns

Bridge

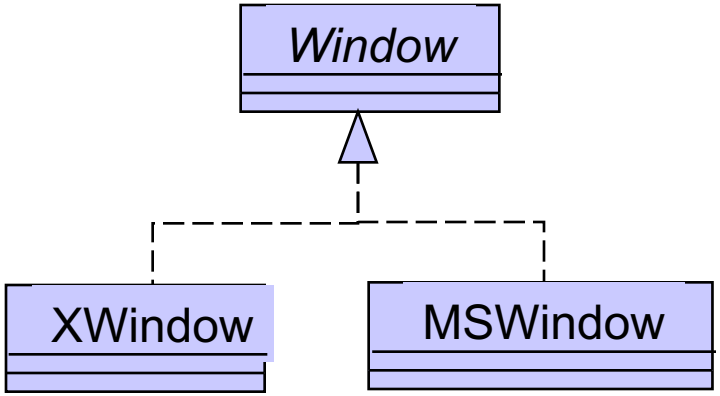
Flyweight

Bridge

- Intent
  - Decouple an abstraction from its implementation so that they can vary independently
- Employs 2 design guidelines
  - "Find what varies and encapsulate it“
  - "Favor aggregation over class inheritance"
- Abstraction and implementation can be changed independently
  - If you use inheritance, only implementation can change

Example

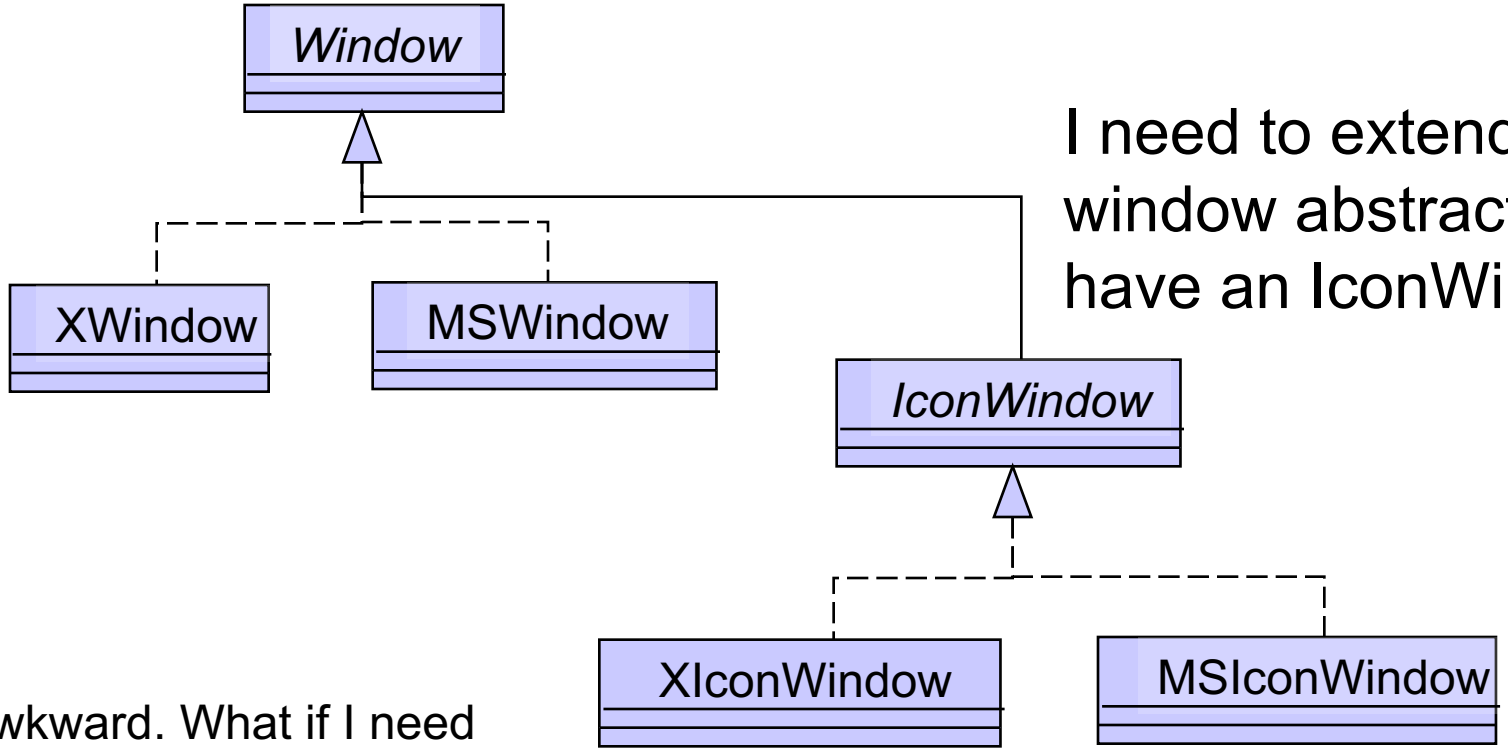
- When abstraction is bound to its implementation with inheritance



I need to extend the window abstraction and have an IconWindow

Example

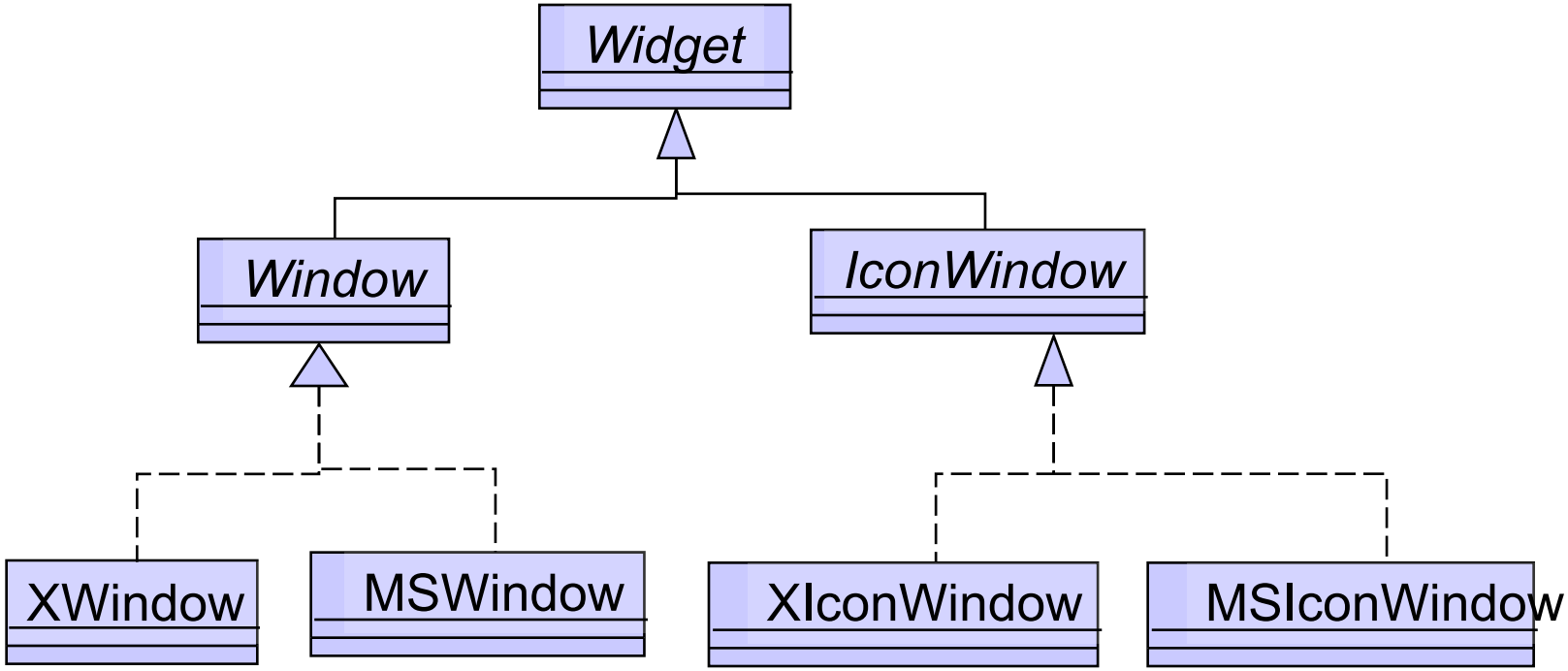
- When abstraction is bound to its implementation with inheritance



I need to extend the window abstraction and have an IconWindow

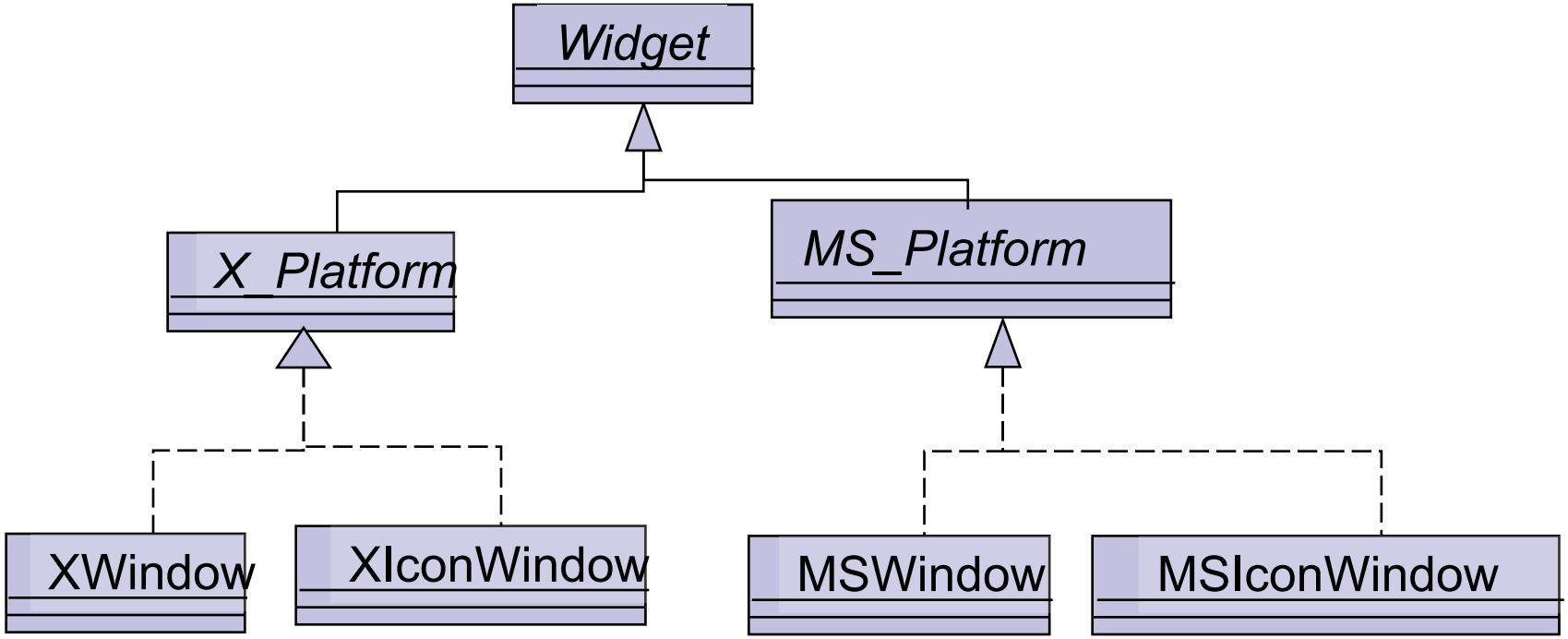
Awkward. What if I need another type of window?

Alternative 1



Still not good. Problem is more visible

Alternative 2

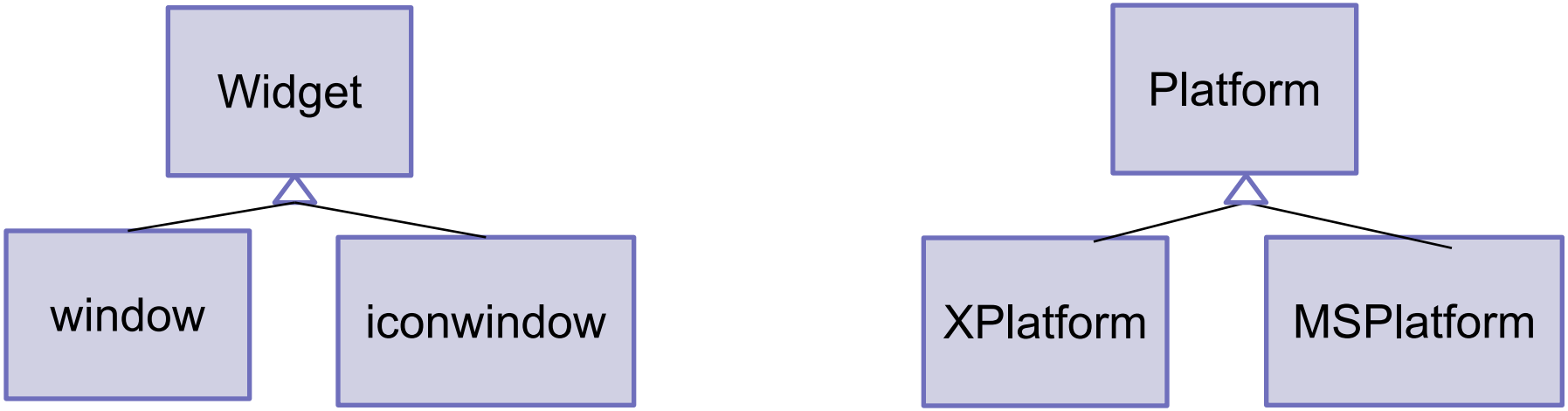


Still not good.

Toward solution

- What is varying?
  - Widget encapsulates window, iconwindow,etc
    - Varying: subtypes of windowing/widget
  - Platform encapsulates the OS specifics
    - Varying: X-platform or MS-platform

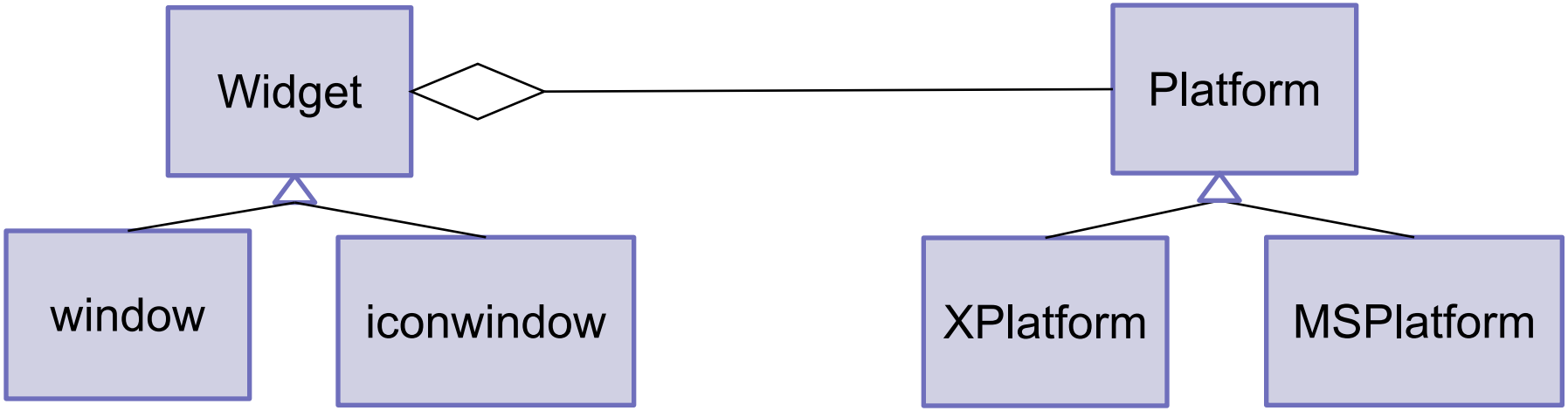
Towards Solution



Toward solution

- What is varying?
  - Widget encapsulates window, iconwindow,etc
    - Varying: subtypes of windowing/widget
  - Platform encapsulates the OS specifics
    - Varying: X-platform or MS-platform
- How to relate them without inheritance
  - does widget uses platform OR
  - platform uses widget?

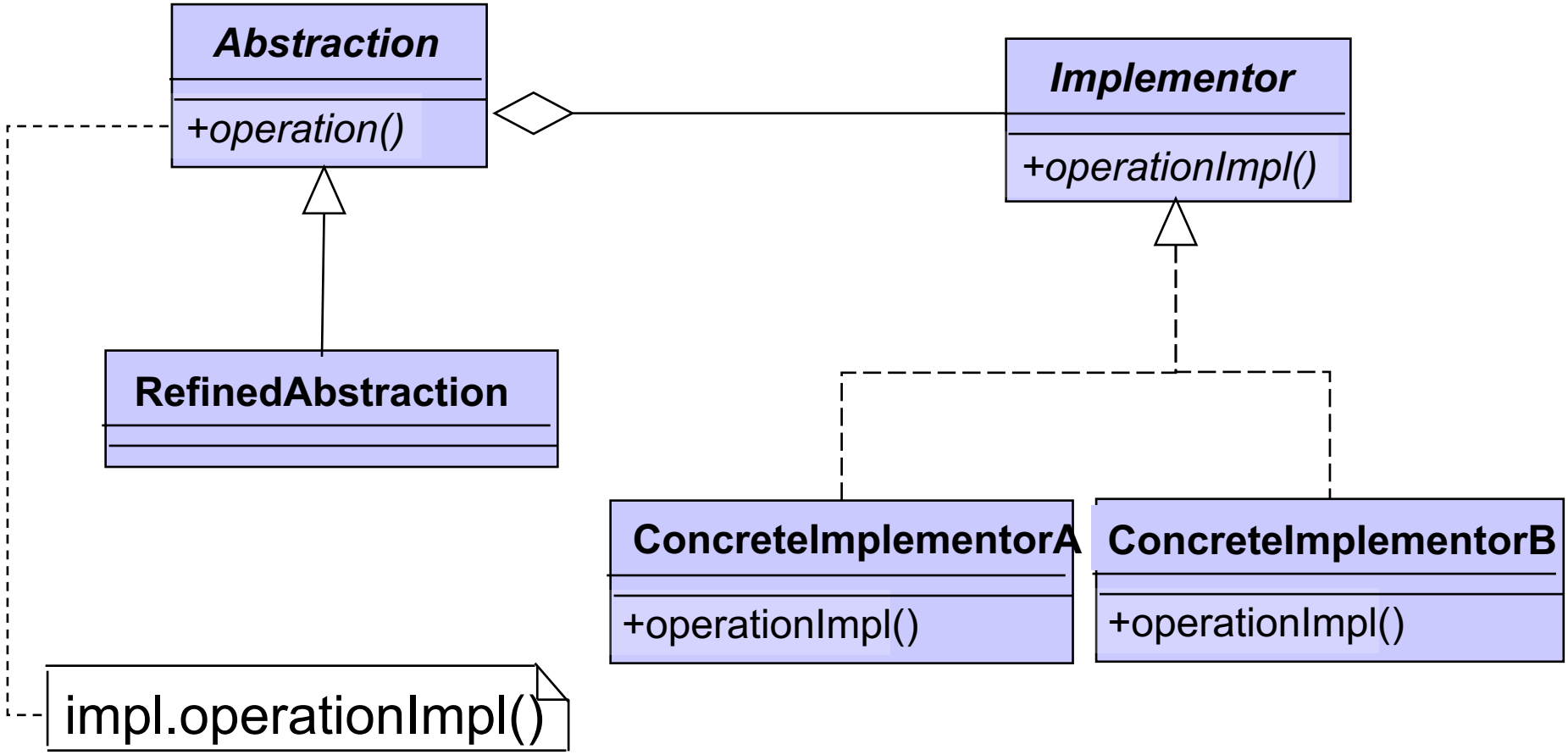
Solution




```
Widget:: expand{
    platfrom.drawRectangle(x,y,x2,y2);
    platform.color(white);
}
```

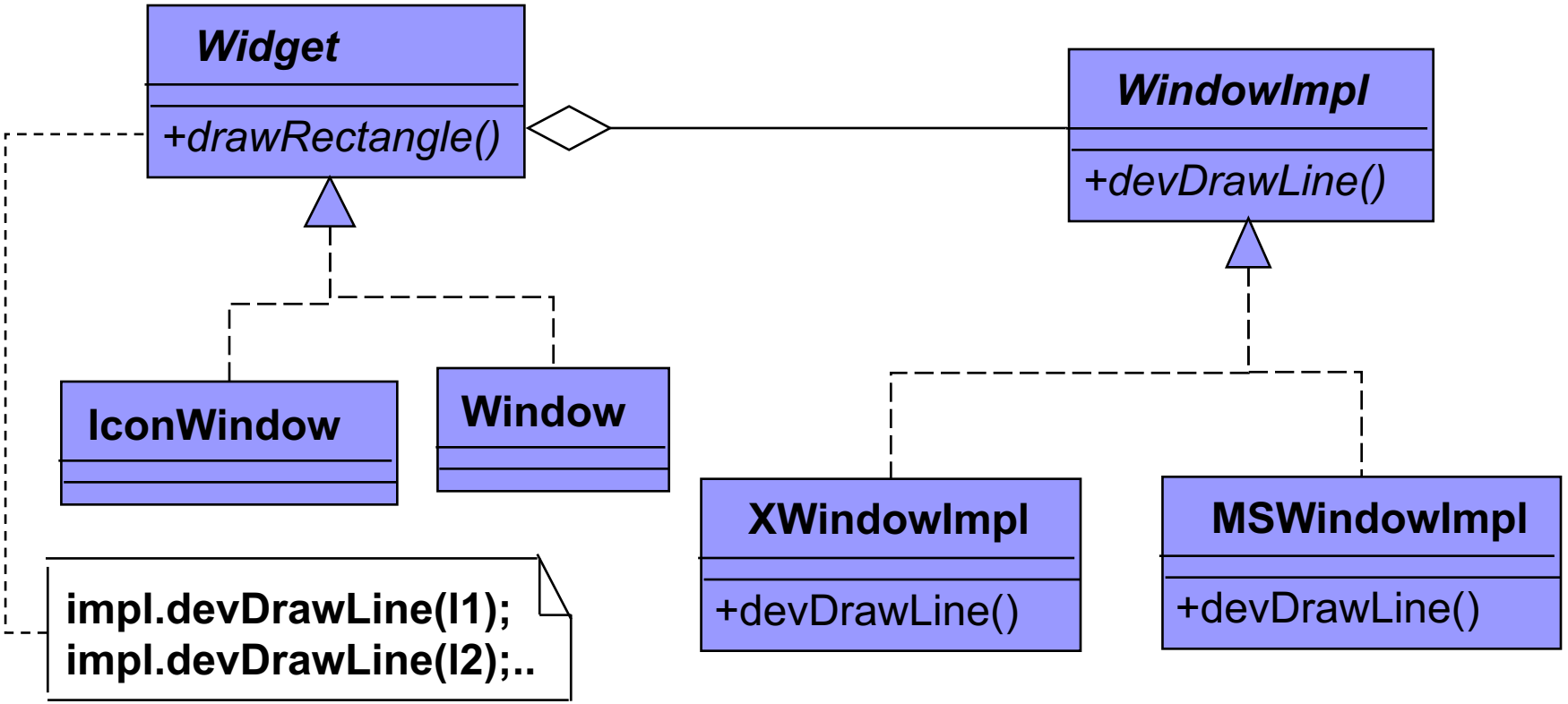
Bridge-Structure

Participants?



Collaboration: Abstraction forwards client requests to its Implementor object.

Using the Bridge



We can extend Window abstraction with more window types easily

We can add other platforms to implement a window

Implementor vs Abstraction

- Implementor interface does **not** have to correspond exactly to Abstraction's interface.
- The two interfaces can be quite different.
  - the Implementor interface provides only **primitive** operations,
    - drawLine, movePoint
  - Abstraction defines **higher-level** operations based on these primitives.
    - drawBorder, drawBoundingBox

Bridge-Applicability

- You want to select (or switch) implementation of an abstraction at runtime
  - **Inheritance** binds implementation to abstraction **permanently**
- Both the abstractions and their implementations should be extensible by subclassing.
  - Inheritance is not practical
- Changes on the implementation of an abstraction (RefinedAbstraction) should not require recompilation of client codes
- Want to hide the implementation of an abstraction completely from clients

Hide implementation from clients

- Using just an interface the client can cheat!

```
Abstraction widget = new ConcreteImplA();
widget.operation();
((ConcreteImplA) widget).concreteOperation();
//cheating
```

- In the Bridge pattern the client code cannot access the implementation
  - No inheritance → no casting

Hide implementation from clients

- Java uses Bridge to prevent programmer from accessing platform specific implementations of interface widgets, etc.
  - It is actually about decoupling for extensibility

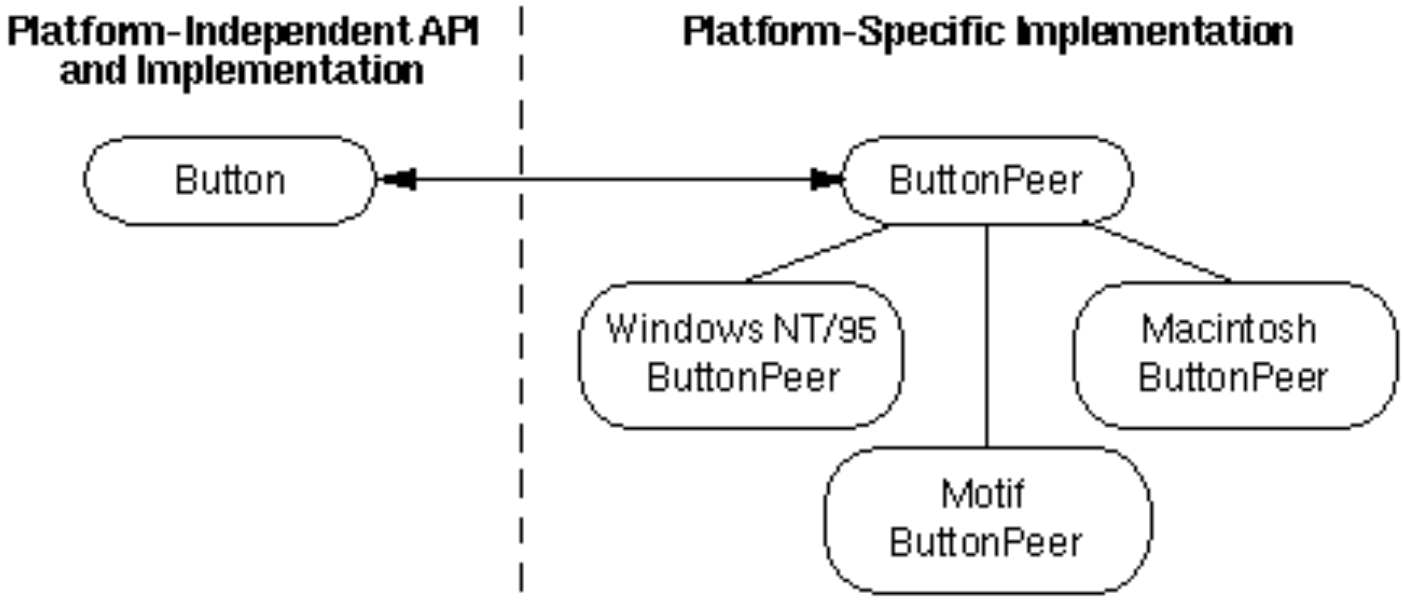
```
public synchronized void setCursor(Cursor cursor) {
        this.cursor = cursor;
        ComponentPeer peer = this.peer;
        if (peer != null) {
            //invoke implementor
            peer.setCursor(cursor);
            }
     }
```

(peer is the implementation)

No dependency injection

Bridge –Known uses

- Java runs on different platforms
  - Create components without committing a concrete implementation
  - Component and Component Peers decoupled



C++ idiom PIMPL

- **P**ointer to **Impl**ementation
- Hiding implementation!
  - Removes implementation details of a class from its object representation
  - i.e. separating implementation from the interface.

A **specific** application of the Bridge pattern to solve

- the problem of compile-time dependencies and
- hide private implementation details.

C++ idiom PIMPL

- **P**ointer to **Impl**ementation
  - Removes implementation details of a class from its object representation
  - i.e. separating implementation from the interface.
- Move the private data and member functions in a **separate class** and accessing them through a **pointer**
  - smart pointer to control the lifetime
- *Reduces build dependencies*
- *Hide implementation from clients*

Do you see the Bridge?

https://en.cppreference.com/w/cpp/language/pimpl
https://www.geeksforgeeks.org/pimpl-idiom-in-c-with-examples/

```
/* |INTERFACE| User.h file */
class User {
public:
    User(string name);
    ~User()=default;
    User(const User& other);
    User& operator=(User rhs);
    //operations
    int getSalary();
    void setSalary(int);
    void do2things();
private:
   // Internal Implementor
   class Impl;
   // pointer to the implementor
    unique_ptr<Impl> pimpl;
  };
```

No implementation detail in the header file.

No data structure info

```
/* |INTERFACE| User.h file */
class User {
public:
    User(string name);
    ~User()=default;
    User(const User& other);
    User& operator=(User rhs);
    //operations
    int getSalary();
    void setSalary(int);
    void do2things();
private:
   // Internal Implementor
   class Impl;
   // pointer to the implementor
    unique_ptr<Impl> pimpl;
  };
```

Do you see the Bridge?

```
class User::Impl { //implementor
   public:
   Impl(string name): name(move(name)){};
   ~Impl()=default;
    void doSth(){cout<<"impl do sth"<<endl;}
    int salary=-1;
   private: string name;
};
// Constructor connected with the Impl
User::User(string name)
            : pimpl(new Impl(move(name))) {}
User::User(const User& other)
       : pimpl(new Impl(*other.pimpl)){}
User& User::operator=(User rhs){
     swap(pimpl, rhs.pimpl); return *this;}
int User::getSalary(){ return pimpl->salary;}
void User::setSalary(int salary) {
            pimpl->salary = salary;}
void User::do2things(){
            pimpl->doSth(); pimpl->doSth();}
```

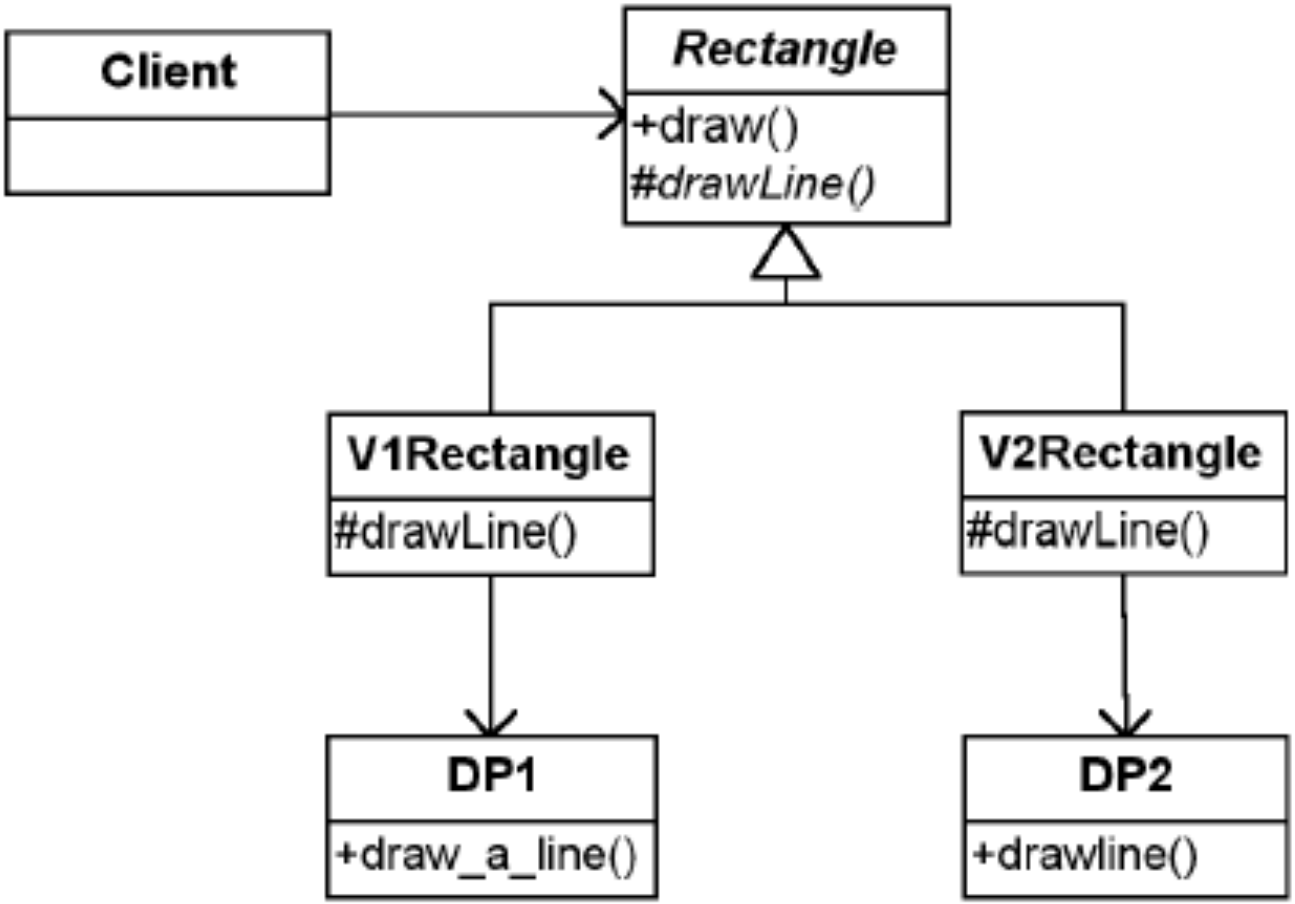
Hide implementation from clients: Sample scenario

Want to hide the implementation of an abstraction completely from clients

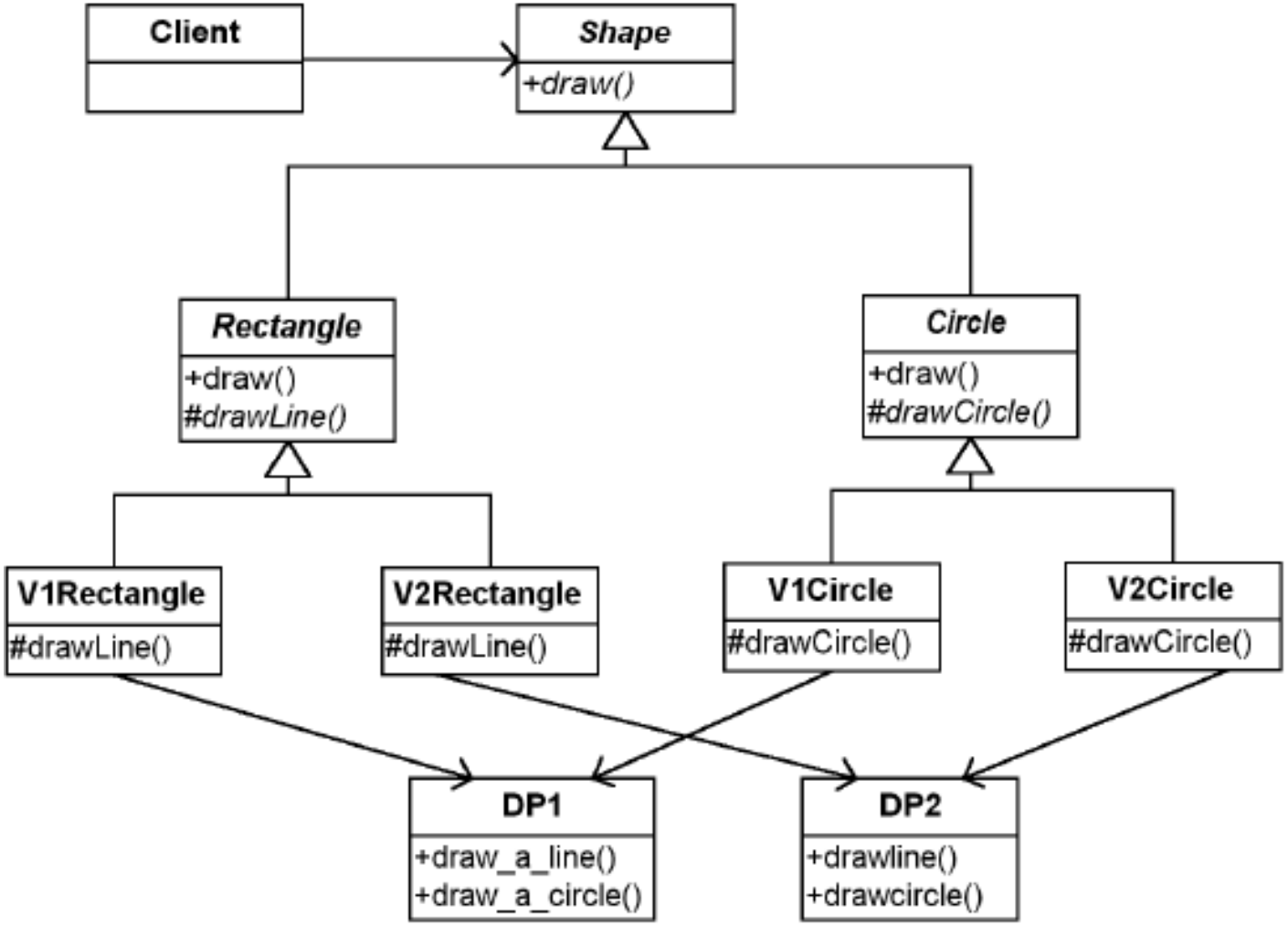
1) If a collection class supports multiple implementations, the decision can be based on the size of the collection.
   - A linked list implementation can be used for small collections and a hash table for larger ones.

2) If the collection grows bigger than a certain threshold, then it switches its implementation to one that's more appropriate for a large number of items.

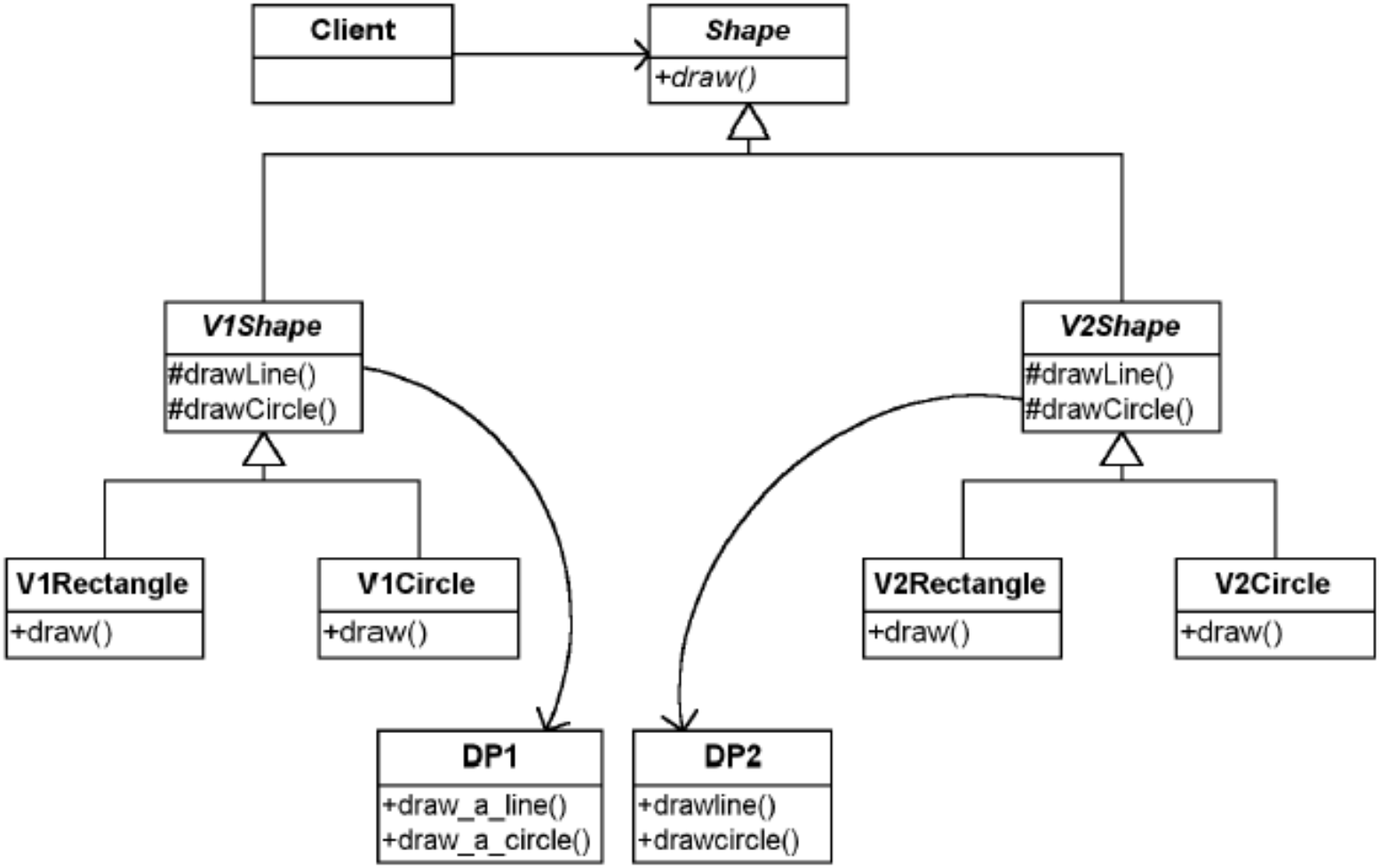
Another example



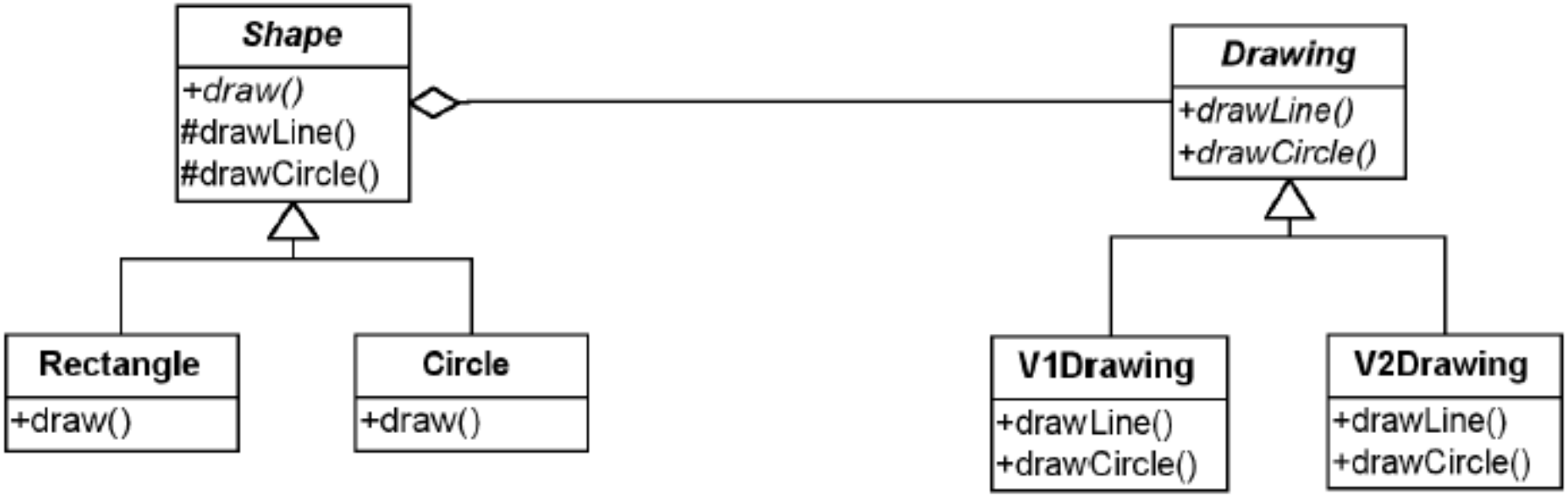
Adding a circle



Not a solution



Using Bridge



Bridge-Consequences

- Decoupling abstraction and implementation
  - Implementation is not bound permanently to a particular interface
  - Enables to switch implementations on-the-fly
  - Reduces compile-time dependencies
    - Changing an implementation class does not require recompilation of Abstraction classes and its clients
  - When *layering*, upper layer only knows Abstraction and Implementor

Bridge-Consequences

- Improved extensibility
  - Abstraction and Implementor hierarchies can be extended independently.
- Useful any time you need to vary an interface and an implementation in different ways.
  - e.g. Graphics that need to run over multiple platforms
- Hiding implementation details from clients.

Implementation issues-1

How, when, and where to instantiate which Implementor class when there's more than one?

- If Abstraction knows about all ConcreteImplementor classes, it can instantiate one of them in its constructor
  - decide between them based on parameters passed to it
- Choose a default implementation initially and change it later according to usage.
  - Delegate the decision to another object: factory
  - Abstraction is not coupled directly to any of the Implementor classes.

Implementation issues -2

- Use multiple inheritance in C++ to combine an interface with its implementation
  - e.g., a class can inherit publicly from Abstraction and privately from a ConcreteImplementor.
  - But it binds an implementation permanently to its interface.
    - You **cannot substitute** different implementors dynamically.
  - Therefore, we cannot implement a true Bridge with multiple inheritance—at least not in C++.

Multiple inheritance?

```
class MyClass : public Abstraction, private
ConcreteImplementor {
    // binds implementation directly
public:
    void operation() override {
        implementation(); // use the concrete method
    }
};
```

- **cannot substitute** different implementors dynamically.
- **Exposes** implementation in the class hierarchy.
- Does **not reduce** compile-time dependencies.

Implementation issues -2 (cont'd)

```
class Abstraction {
protected:
    Implementor* impl;
public:
    Abstraction(Implementor* i) : impl(i) {}
    virtual void operation() {
        impl->implementation();
    }
};
```

Enables substituting different implementors dynamically

- Bridge implementation in C++
- This is **not** Pimpl
  - Pimpl uses a **single concrete implementation** hidden behind a pointer.

Bridge vs Adapter

- Difference in intention
  - The Adapter pattern's intent is making unrelated classes work together
    - Make things work **after** they are designed
  - Bridge pattern intent's is to allows us to use multiple *implementations* with possibly multiple interfaces
    - Make things work **before** they are designed
- Both promotes flexibility by providing a level of indirection to another object

FLYWEIGHT

Flyweight

- Intent
  - Use sharing to support large numbers of fine-grained objects efficiently
- Some programs require a large number of objects whose states are the same.
  - Why not SHARE these objects instead of creating unique instances?
  - Shared object that can be used in multiple contexts simultaneously.

Flyweight: Example Scenario

- A scene drawing application
  - There are 10000 trees each of them have different location
  - All trees look the same. They are not the primary objective of this scene.
  - Memory problem… what can be shared?

```
public class Tree{
  private int height;
  private Point p;
  private Color color;
  public void display(Graphics g){ ..}
  public Tree(…)   {…}
}
```

```
class TreeRealistic{
private:
  Texture bark_;
  Texture leaves_;
  Vector position_;
  double height_;
  double thickness_;
  Color barkTint_;
  Color leafTint_;
};
```

Encapsulate what varies

```
class Tree{
  private int height;
  private Point p;
  private Color color;
  public void display(Graphics g) {
       ….g.setColor(color);
     g.fillOval(p.x - 5, p.y - 10, 10, 10);}
  public Tree(…){..}
}
class Forest { //client
 private Tree[ ] trees;
 void plantTree(){
     for (int i=0;i<10000; i++)
          trees[i]=new Tree(…);}
 void paint(Graphics g){
      for(var t:trees) t.display(g);}
}
```

```
class Tree{  //FLYWEIGHT
  private int height;
  private Color color;
  display(Point p,Graphics g) {
…g.fillOval(p.x - 5, p.y - 10, 10, 10);}
  public Tree(int h, Color c){…}
}
class Forest { //client
 private Point[ ] pos;
 private Tree[ ] trees;
 void plantTree(){
     for (int i=0;i<10000; i++)
        trees[i]=new Tree(100,green);}
 void paint(Graphics g){
     for (int i=0;i<1000; mi++)
          t.display(pos[i],g)}
}
```

Still not sharing

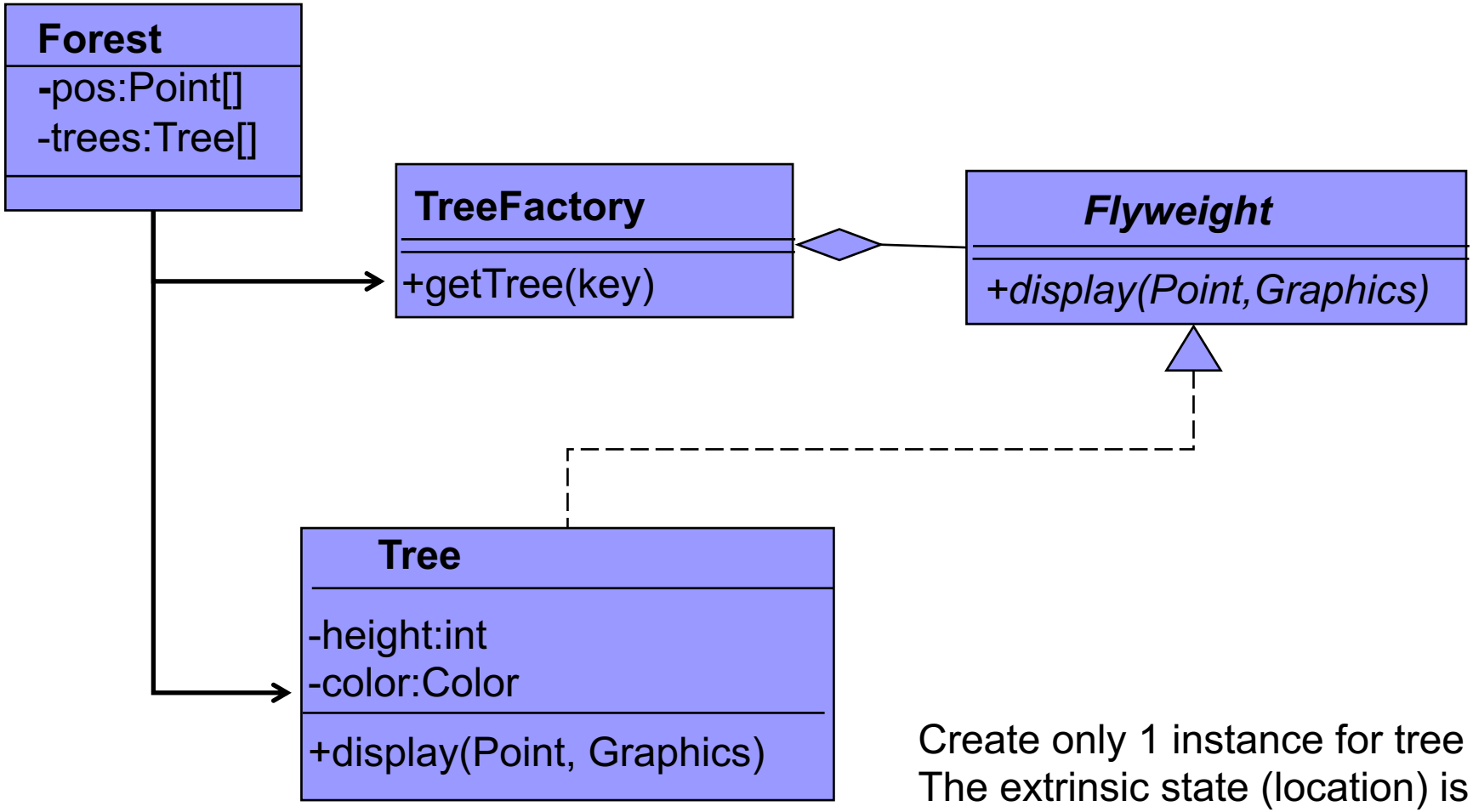
```
public class TreeFactory{
    Map<String,Tree> dict;
   public Tree getTree(String
name, int height, Color c){
    Tree result=dict.get(name);
     if(result==null)
        dict.put(name,
           new Tree(height,c));
    return result;
  }
  public TreeFactory(){
     dict=new HashMap<Tree>();
  }
 //make this class Singleton
}
```

```
class Tree{  //FLYWEIGHT
  private int height;
  private Color color;
  display(Point p,Graphics g) {
…g.fillOval(p.x - 5, p.y - 10, 10, 10);}
  public Tree(int h, Color c){…}
}
class Forest { //client
 private Point[ ] pos;
 private Tree[ ] trees;
 void plantTree(){
    for (int i=0;i<10000; i++)
        trees[i]=new Tree(100,green);}
 void paint(Graphics g){
    for (int i=0;i<1000; mi++)
        t.display(pos[i],g)}
}
```

```
public class TreeFactory{
    Map<String,Tree> dict;
  public Tree getTree(String key,
                int height, Color c){
    Tree result=dict.get(name);
     if(result==null)
         dict.put(name,
             new Tree(height,c));
    return result;
  }
  public TreeFactory(){
     dict=new HashMap<Tree>();
  }
 //make this class Singleton
}
```

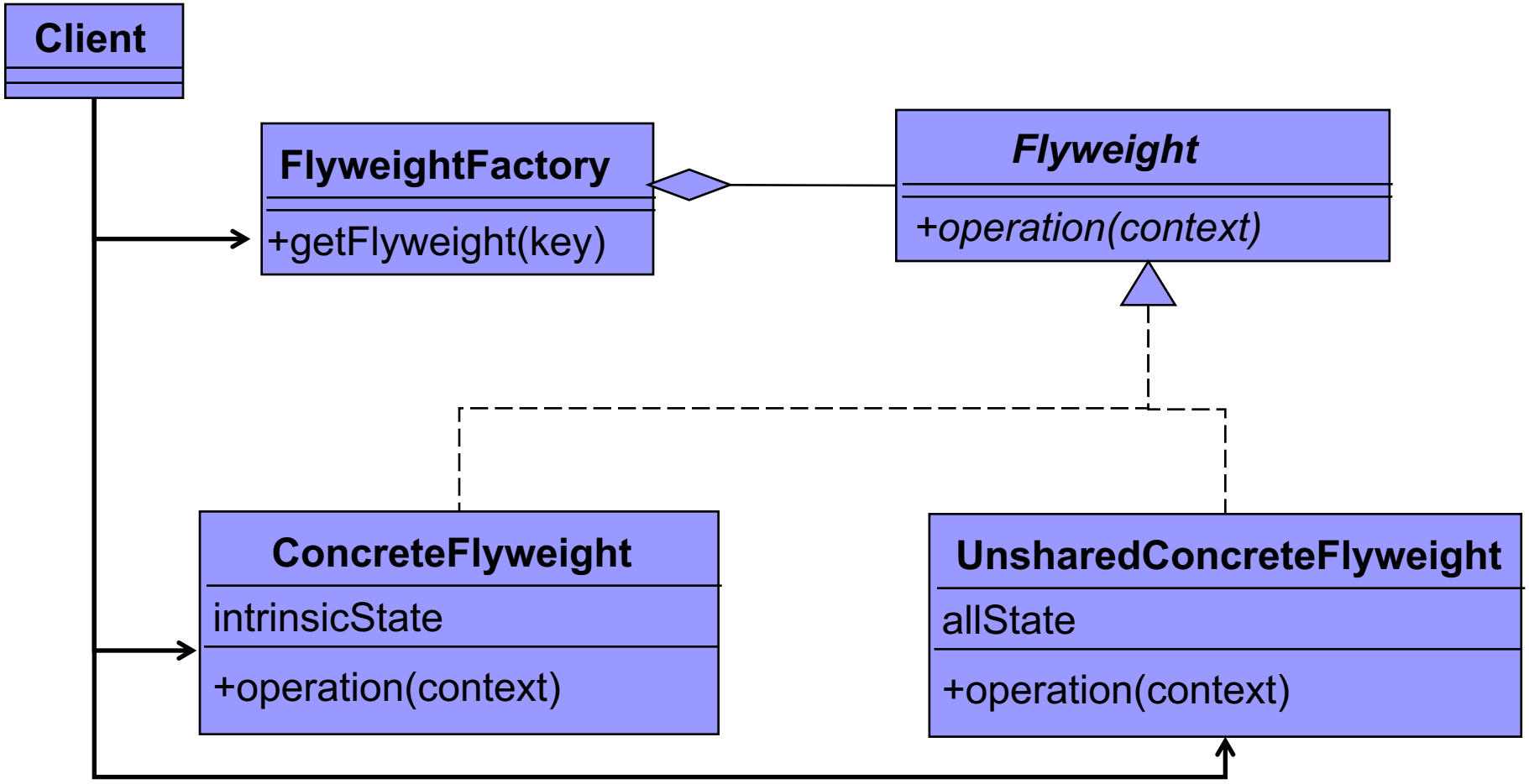
```
class Tree{  //FLYWEIGHT
  private int height;
  private Color color;
  display(Point p,Graphics g) {
…g.fillOval(p.x - 5, p.y - 10, 10, 10);}
  public Tree(int h, Color c){…}
}
class Forest { //client
 private Point[ ] pos;
 private Tree[ ] trees;
 void plantTree(){
     for (int i=0;i<10000; i++)
          trees[i]=
              factory.getTree(“y”,100,green);}
 void paint(Graphics g){
     for (int i=0;i<1000; mi++)
          t.display(pos[i],g)}
}
```

Structure of the Scene



Create only 1 instance for tree
The extrinsic state (location) is passed from outside when display is called

Flyweight -Structure

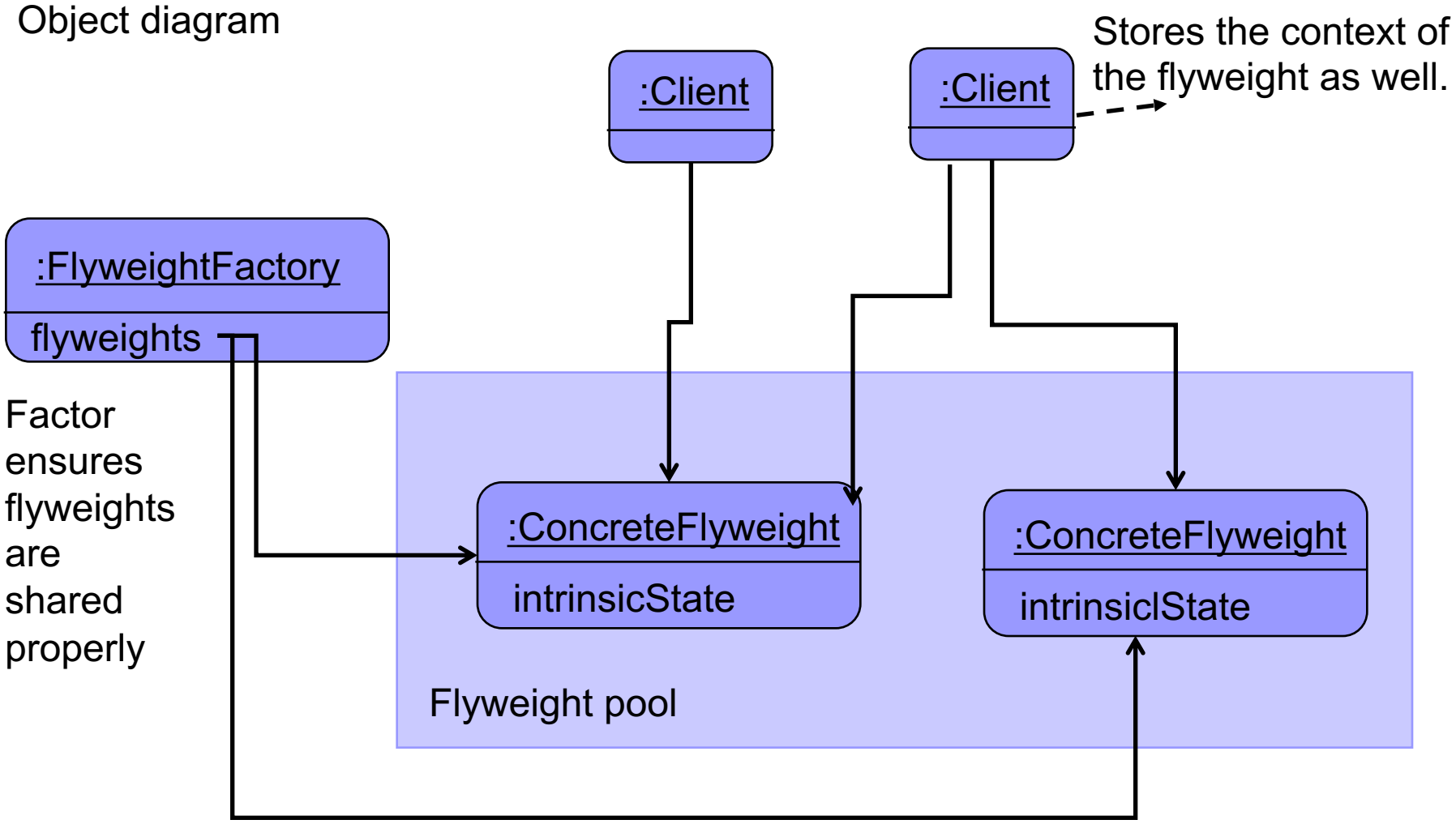


Intrinsic state=state that does not depend on the context

Flyweight participants

- **Flyweight** - Declares an interface through which flyweights can receive and act on extrinsic state.
- **FlyweightFactory** - The factory creates and manages flyweight objects. Ensures sharing of the flyweight objects. Maintains a pool of different flyweight objects.
- **Client** - A client maintains references to flyweights in addition to computing and maintaining extrinsic state (context)
- **ConcreteFlyweight** - Implements the Flyweight interface and stores intrinsic state. Sharable. Manipulates state that is extrinsic.

Flyweight - Structure



Client and Context

- Client stores or calculate values of the extrinsic state (context) to be able to call methods of flyweight objects.
  - Violates Single Responsibility
- Extract a Context class for cohesion
  - Move the extrinsic state along with the flyweight-referencing field to a separate context class.

Using a separate Context

```
public class Tree {
    private Point p;
    private TreeType type;

    public Tree(Point p,
                TreeType type) {
        this.p= p;
        this.type = type;
    }

    public void draw(Graphics g) {
        type.draw(p,g);
    }
}
```

Which one is Flyweight?
Which one is Context?
Is there another pattern here?

```
public class TreeType {
    private String name;
    private Color color;
    privare int height;

    public TreeType(String name,
                    Color color, int h) {
        this.name = name;
        this.color = color; height=h;
    }

    public void draw(Point p, Graphics g) {
        g.setColor(Color.BLACK);
        g.fillRect(p.x - 1, p.y, 3, 5);
        g.setColor(color);
        g.fillOval(p.x - 5, p.y - 10, 10, 10);
    }
}
```

https://refactoring.guru/design-patterns/flyweight/java/example

```
public class TreeFactory {
    static Map<String, TreeType> treeTypes = new HashMap<>();

    public static TreeType getTreeType(String name, Color color, int height) {
        TreeType result = treeTypes.get(name);
        if (result == null) {
            result = new TreeType(name, color, height);
            treeTypes.put(name, result);
        }
        return result; }
}
```

Which one is the Flyweight Factory?

```
public class Forest extends JFrame {
    public void plantTree(Point p, String name, Color color, int height) {
        TreeType type = TreeFactory.getTreeType(name, color, height);
        Tree tree = new Tree(p, type);
        trees.add(tree);
    }
    public void paint(Graphics graphics) {
        for (Tree tree : trees) {
            tree.draw(graphics);
        }
    }
    private List<Tree> trees = new ArrayList<>();
```

```
public class Demo {
    static int CanvasSIZE= 500;
    static int TREENUM = 1000000;
    static int TREE_TYPES = 2;

    public static void main(String[] args) {
        Forest forest = new Forest();
        for (int i = 0; i < Math.floor(TREENUM/ TREE_TYPES); i++) {

            forest.plantTree(random(0, CanvasSIZE),
                    "Summer Oak", Color.GREEN, "Oak texture stub");

            forest.plantTree(random(0, CANVAS_SIZE),
                    "Autumn Oak", Color.ORANGE, "Autumn Oak texture stub");
        }
        forest.setSize(CanvasSIZE, CanvasSIZE);
        forest.setVisible(true);
    }
    public static Point random(int min, int max) {……}
}
```

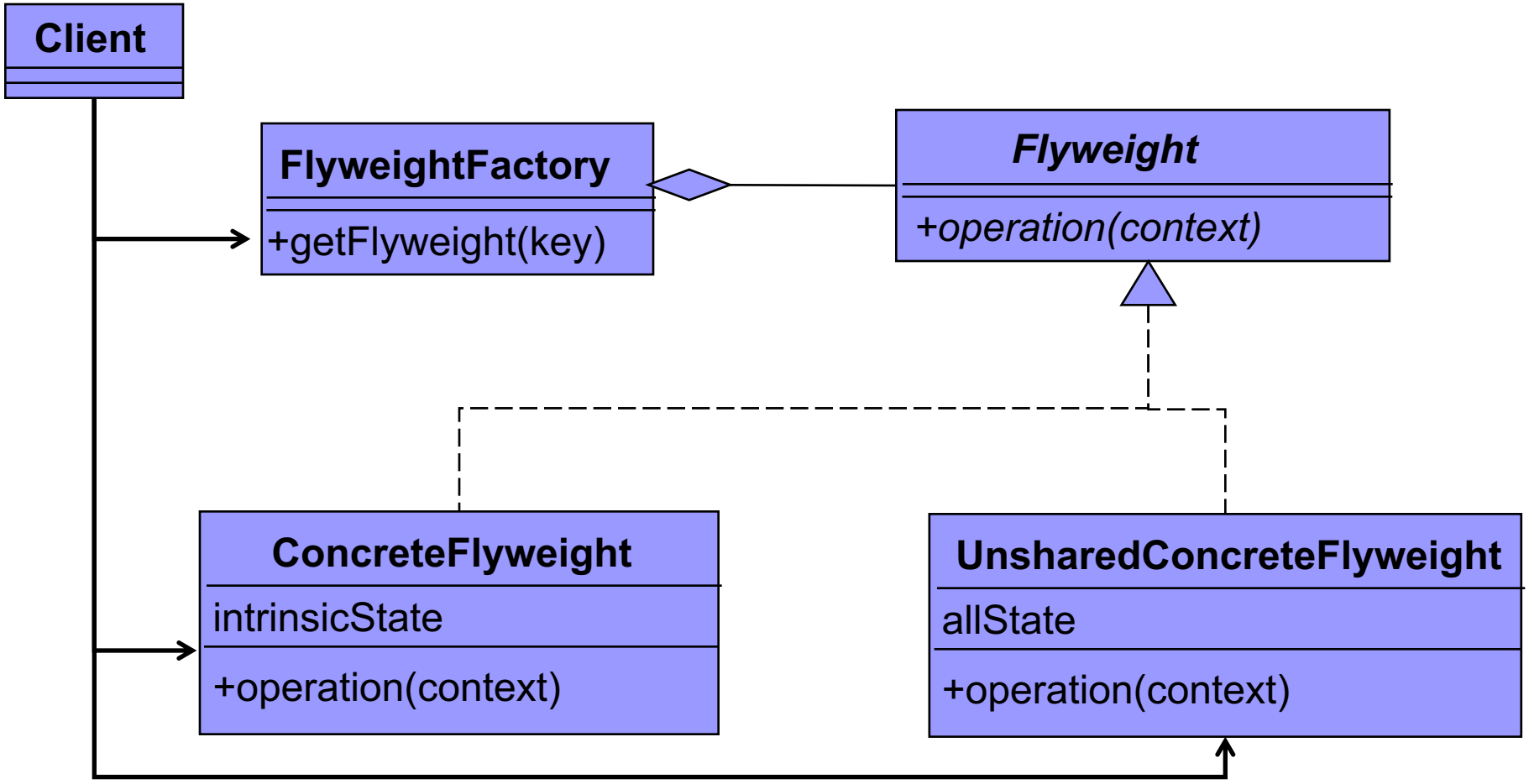
Draw UML diagram

Please see https://refactoring.guru/design-patterns/flyweight/java/example
used 7MB instead of 36MB

Flyweight

- Intent
  - Use sharing to support large numbers of fine-grained objects efficiently
- Applicability (All should apply)
  - Application uses large number of objects, and
  - Storage costs are high due to quantity of objects, and
  - Most object state can be made extrinsic (can be passed as parameter), and
  - Many groups of objects may be replaced by a few shared objects once the extrinsic state (context) is removed, and
  - Application does not depend on object identity
- Works good with immutable objects

Flyweight -Structure



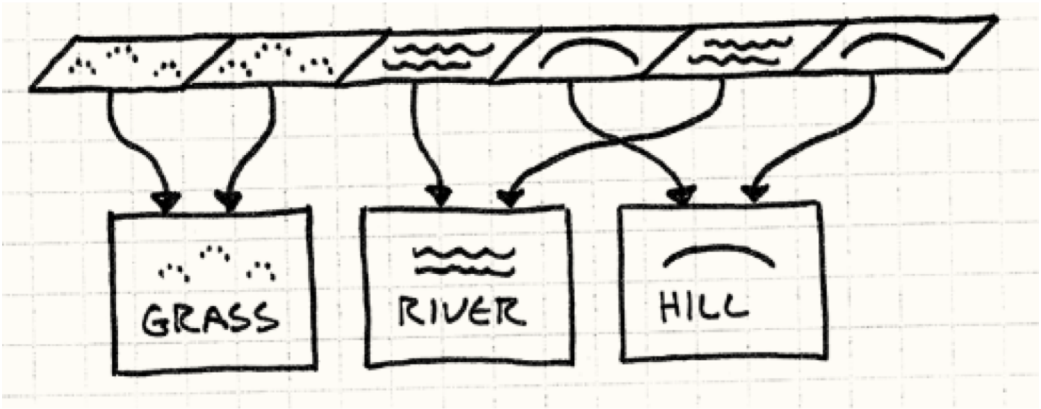
Clients should not instantiate Flyweights directly, since they are shared.

Flyweight

- Use flyweight when you need one instance of a class can be used to provide many *“virtual instances*”
- Support a large number of small objects efficiently by
  - Allowing them to be shared with essential values and
  - Storing them in a factory or manager
  - Shared objects modified with extrinsic values from their context

Use case: Game programming

- Game programming →optimization
- Coding with enum and lots of switch-case is efficient, but
  - No OOP advantage
  - No Open-Closed principle
- Could we use flyweights and get similar efficiency?



Please read the "Terrain" implementation in *Game Programming Patterns* book.

See https://gameprogrammingpatterns.com/flyweight.html

Flyweight: Example 2

- Would you have an object for each character in a document?
  - Each character object has the font and style information
- Have a limited character objects and a context that says where and with what attributes it exists in the document?
  - Shared Character flyweights with intrinsic state as character code
  - Character context that says character location and their formatting attributes (extrinsic states)
  - When using the flyweight operations, pass the extrinsic value to it without modifying the intrinsic state

Flyweight - Consequences

- Saves space by sacrificing performance
  - Reduces # of instances
  - Sacrificing time for transfer, find, and compute extrinsic state
- Space saving depends on
  - Reduction in total number of instances
  - Amount of essential state per object
  - Whether extrinsic state is stored or computed
- Centralizes state for many virtual objects into a single location
- Cannot perform identity test
- Instances cannot behave independently from other instances

Related patterns

- Often combined with the Composite when leaf nodes are shared objects
  - Tree turns into a directed-acyclic graph
  - No Parent pointer in the flyweight-- extrinsic state
- Implement State and Strategy objects as flyweights. (later)

Known use:

- java.lang.Integer#valueOf(int)