Pattern Oriented Design

www.netobjectives.com

What We Are Going to Do

- This material is an excerpt from our Pattern Oriented Design courses
- Discuss a real world problem and solve it using ‘standard’ OO
- Learn what design patterns are
- Learn several design patterns
  - adapter, facade, bridge, abstract factory
- Get insights into Christopher Alexander’s thinking
- Learn to use patterns as a conceptual design tool -- “Thinking in Patterns”
- Revisit the problem using design patterns
- Demonstrate how principles drive good pattern design - Open-closed principle

Pattern Oriented Design

- Integrates five disciplines
  - Object-Oriented Principles
  - Design Patterns
  - Commonality-Variability Analysis
  - Analysis Patterns
  - Architectural Pattern Languages (a la Alexander)
- Is mostly about design, not implementation
- Attempts to change our paradigms of design

Pattern Oriented Design and Changing Requirements

- Pattern Oriented Design is about anticipating and handling changing requirements.
- It focuses on relationships more than specific entities and behaviors.
- Relationships vary more slowly than do our entities and the behaviors of these entities.
- Focusing on relationships allows us to disentangle the dependencies between things.

Pattern Oriented Design Vs Object-Oriented Design

- *Is* object-oriented design.
- Integrates advanced techniques into its approach.
- Goes beyond merely finding the nouns in your functional specifications and hoping insight and experience will take over.
- Uses relationships between the entities in your problem domain to identify the objects you need to define.
- Is iterative, so as you define more relationships, object definition becomes easier.

Pattern Oriented Design Vs OOD - cont'd

- Makes use of advanced design techniques espoused in Gang of Four Design Patterns book:
  - find what varies and encapsulate it
  - favor composition over inheritance
- However, use of design patterns makes this straightforward, not complicated.

Learning Pattern Oriented Design

- POD, is simple since it follows an intuitive approach.
- We must look at our design problem from a different perspective, however.
- It requires the same paradigmatic shift of looking at objects that object-oriented design requires.
- It also requires looking at relationships between these objects early in the design.

In the Old Days ...

- “Two things in life are certain -- death and taxes.”
  -- Ben Franklin

- In the information age, there is a third ...

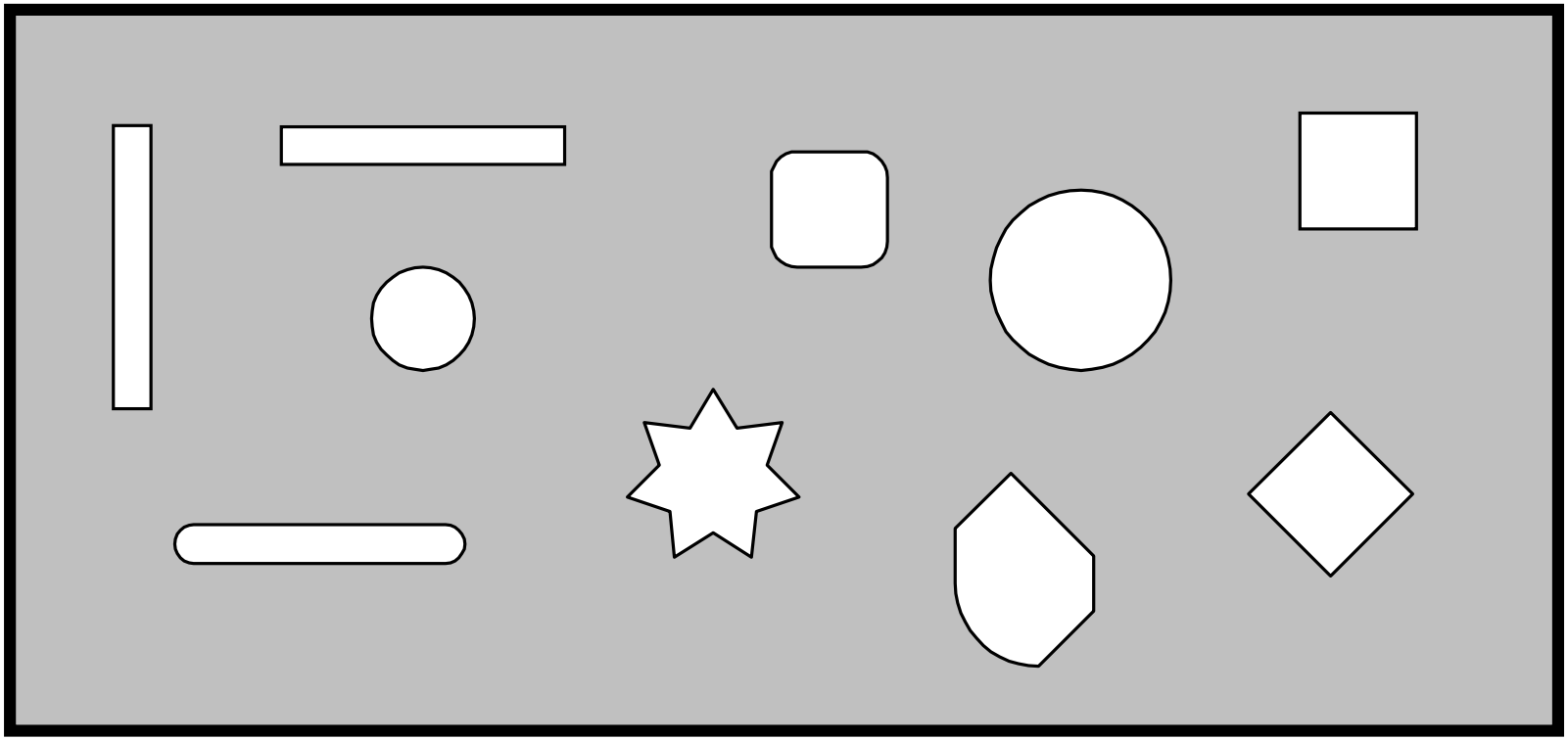
In the Information Age …

- Three things in life are certain --
  - death
  - taxes
  - requirements will change

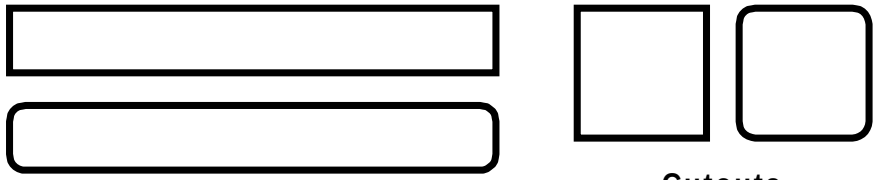
The Problem to Solve

- Start with CAD/CAM designs of sheet metal parts
- Need to be able to analyze the parts
- The strategy used to analyze the parts is dependent upon the equipment used to machine the parts
- This equipment will change more slowly than will the CAD/CAM systems used
- We therefore need to make our software independent of the CAD/CAM system in use

Example Piece of Sheet Metal

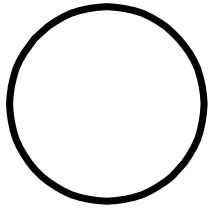


Sheet Metal Described as Combination of Features

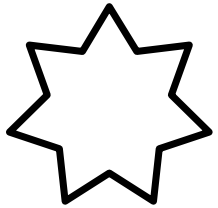


Slots

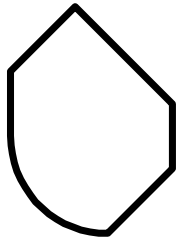
Cutouts



Hole



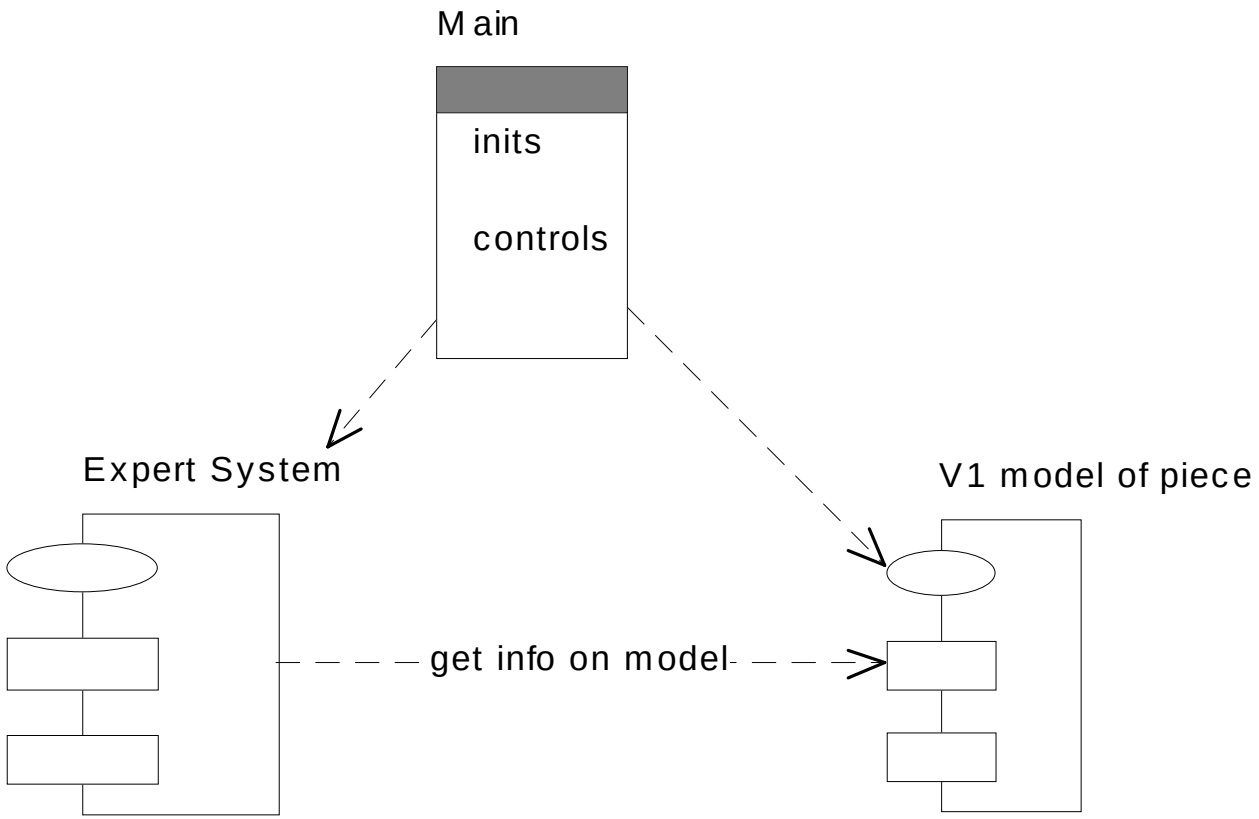
Special



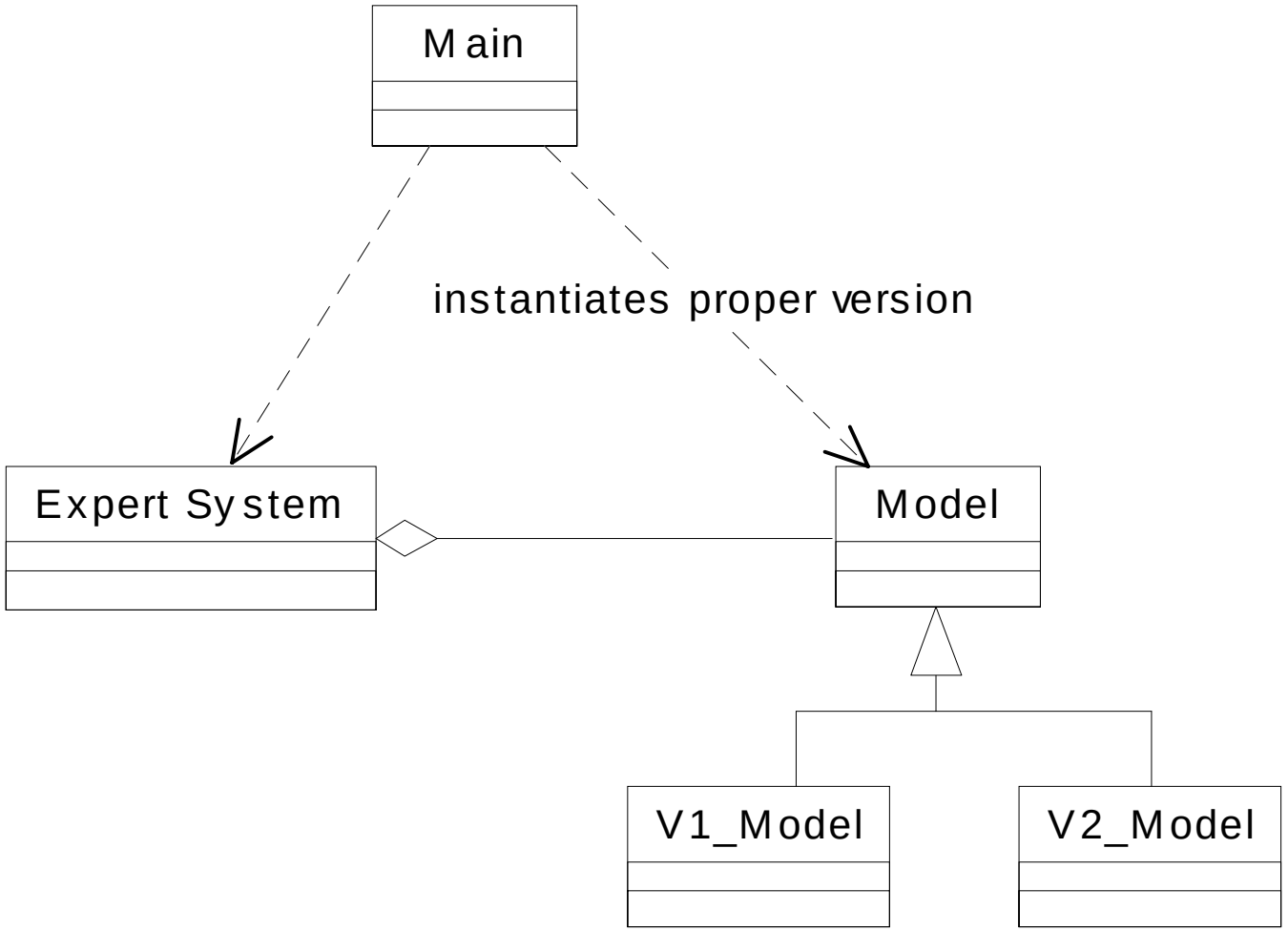
Irregular

TYPES OF FEATURES

High View of Desired Architecture

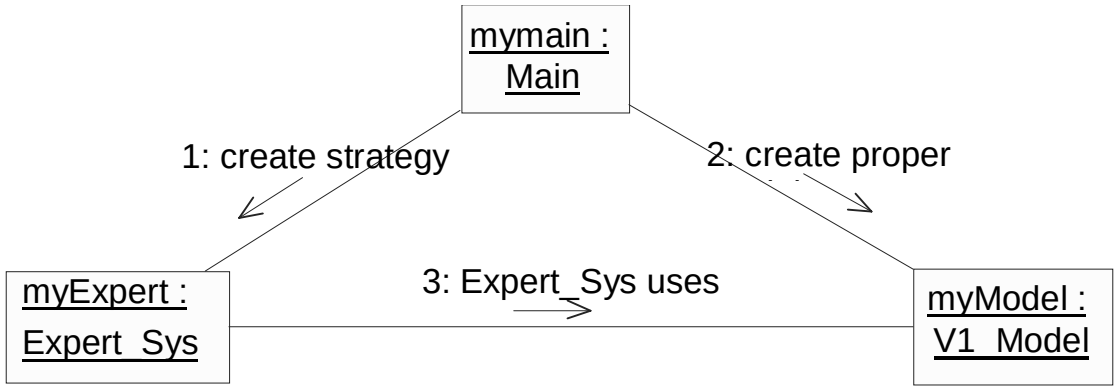


High Level Class Diagram

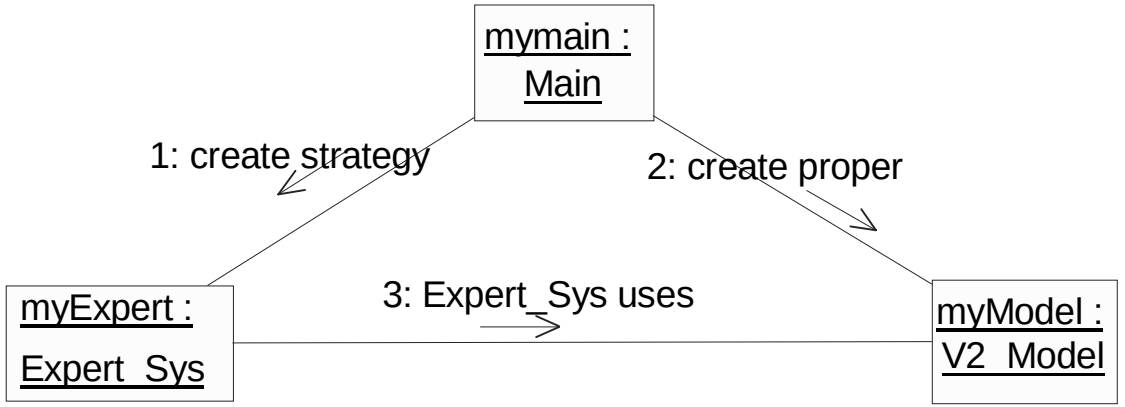


We Get One or The Other

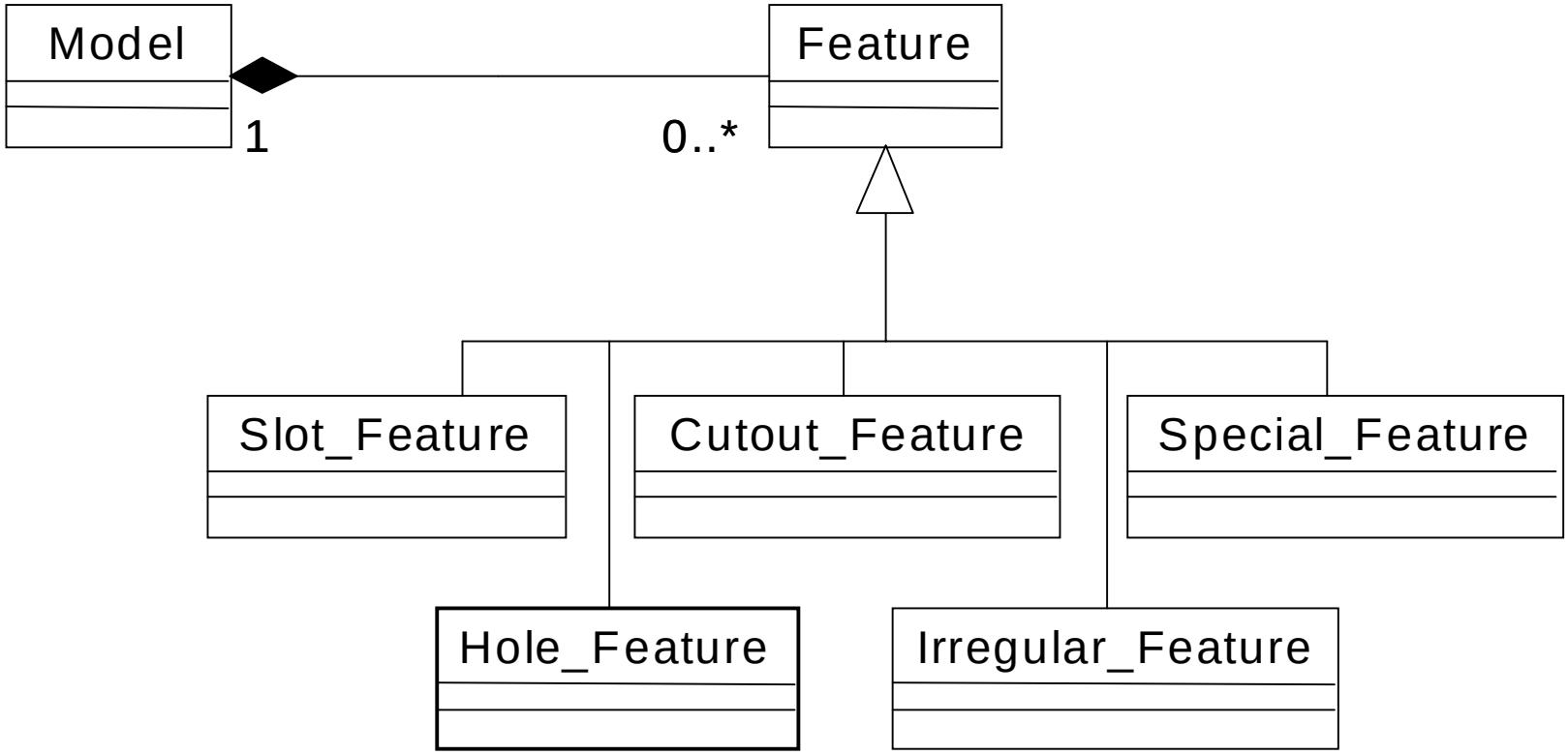
When we have model of type 1



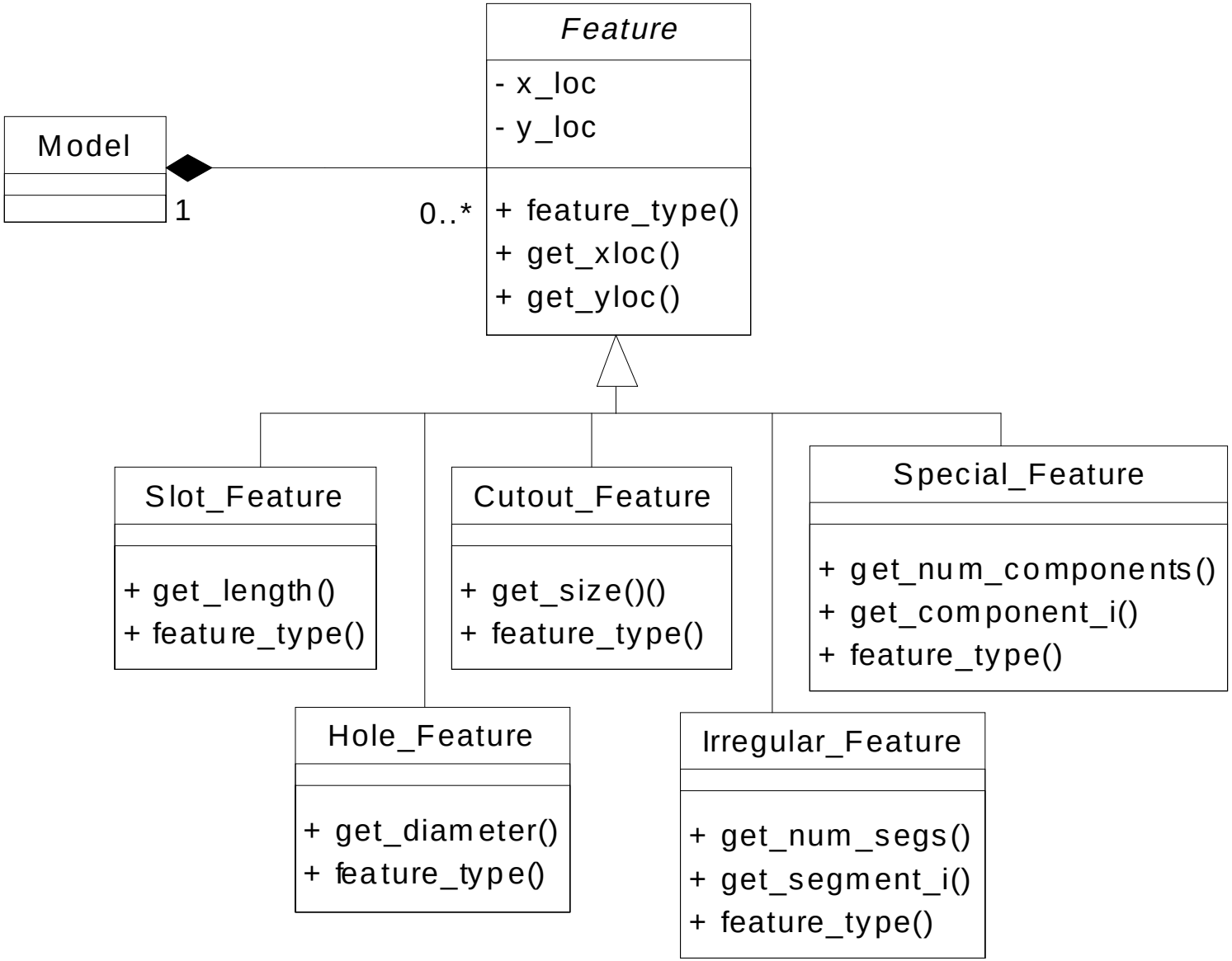
When we have model of type 2



The CAD/CAM Model



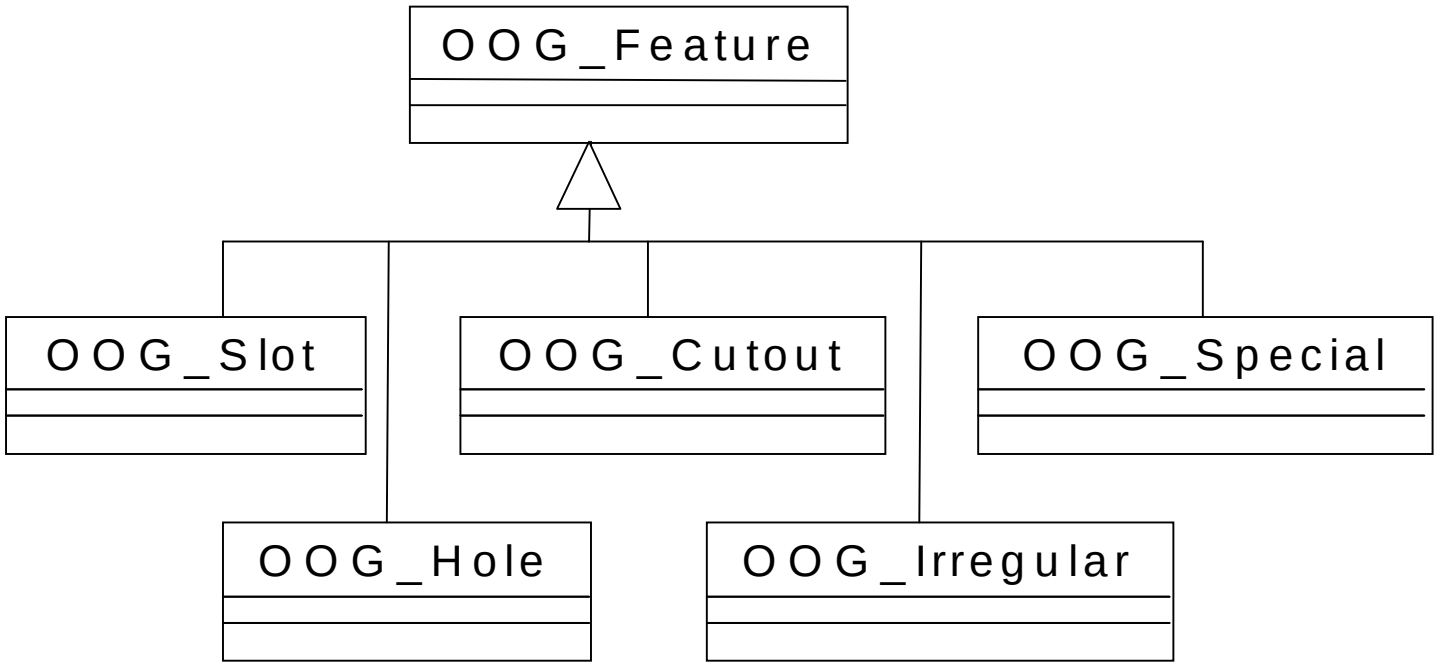
A Little More Detail



Our CAD/CAM Systems

- One system consists of a set of function calls to implement our features.
- The other system consists of a set of classes representing our features.

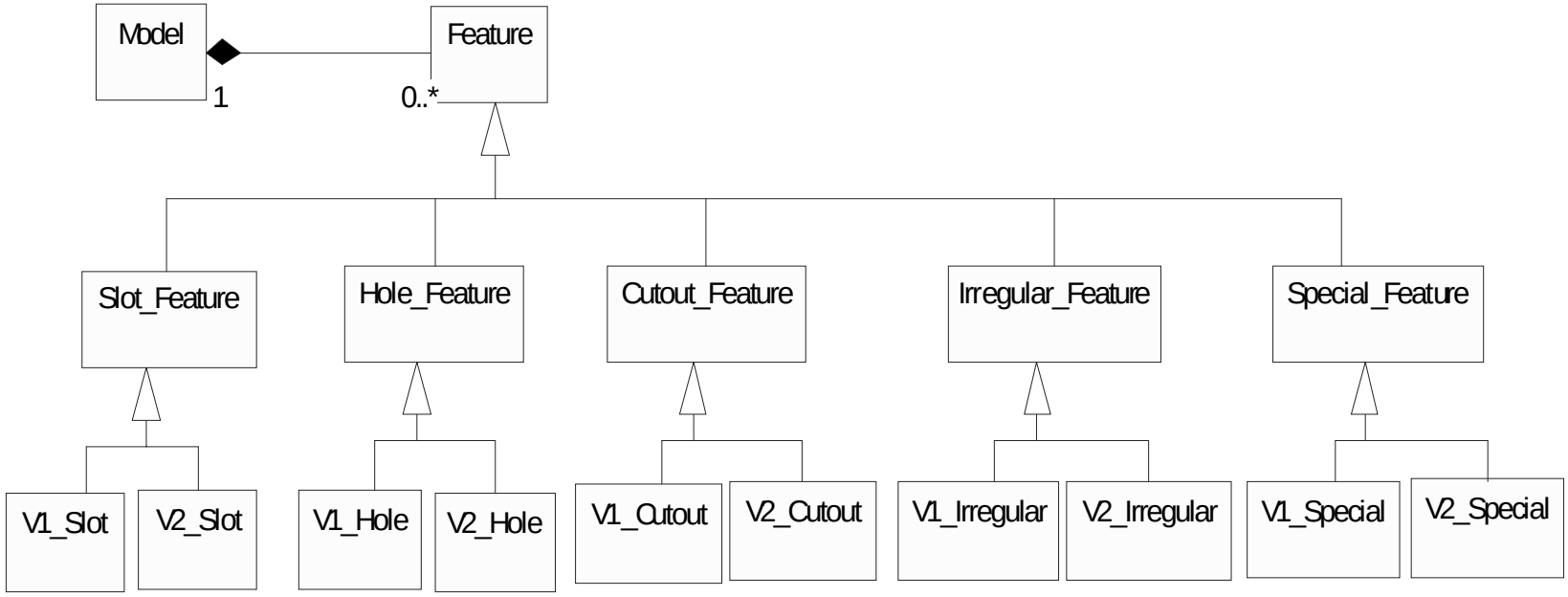
Our OO Based CAD/CAM System (Version 2)



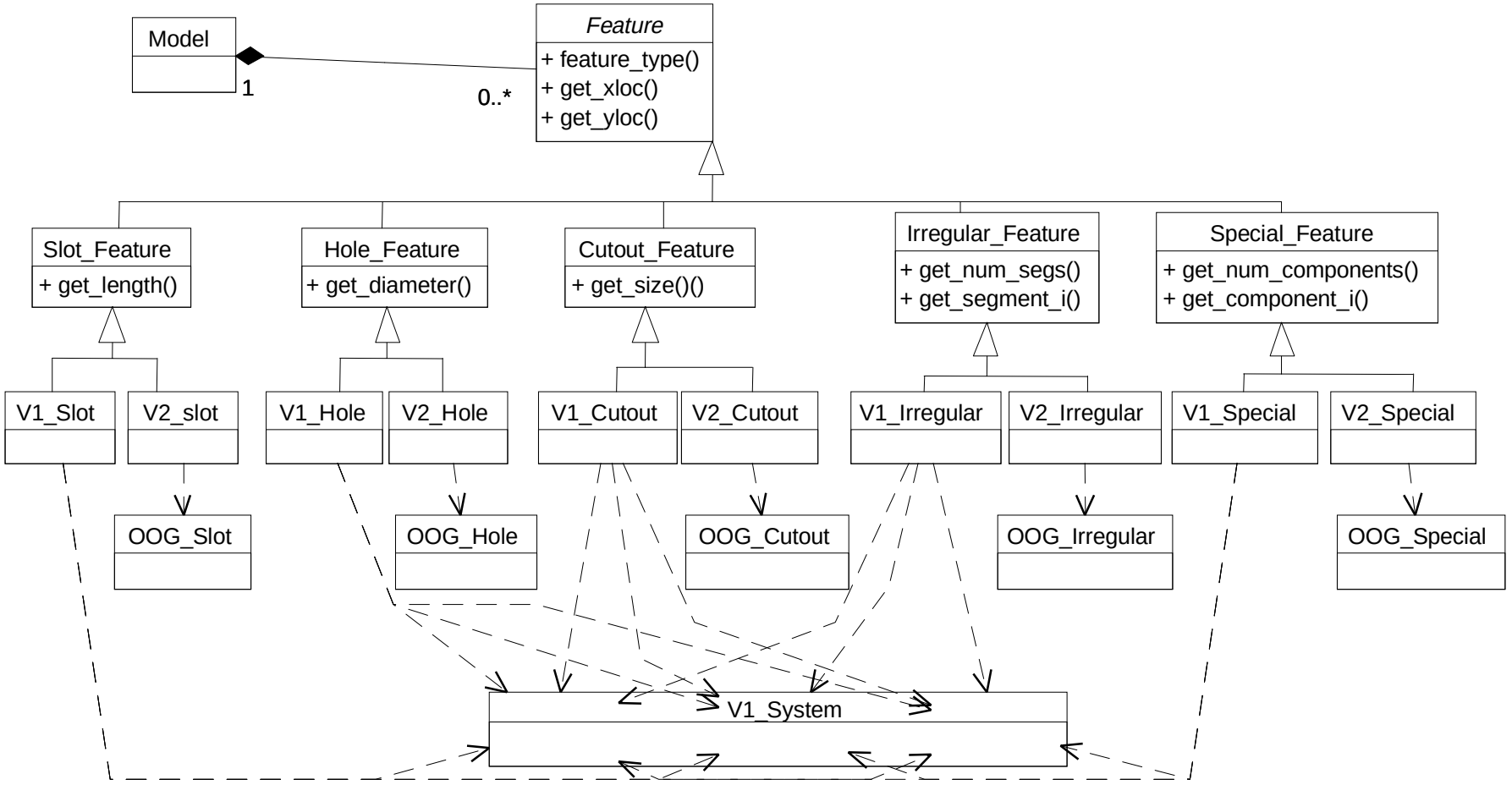
Finding a First Solution

- We know we will have different flavors of each feature
- When we have a V1 model, we will interface to the library routines
- When we have a V2 model, we will interface to objects that represent the features in our model

We Know We'll Specialize First



One Possible Solution



Three Levels of Perspective

- Conceptual Perspective
- Specification Perspective
- Implementation Perspective

- From Martin Fowler's UML Distilled

Conceptual Perspective

- Describes *what* you want
- *Not* how you'll get it
- Can be very detailed
- Example: want house with lots of bedrooms and baths should be light and airy, big kitchen

Specification Perspective

- We talk about *how* things behave
- What are the relationships between entities
- We have identified behaviors, but not implementations. That is, we have classes and interfaces, not code.
- For example: have 3BR, 2BA, and all bedrooms on top floor, kitchen overlooks garden, connected to dining room, ...

Implementation Perspective

- We have classes and how we are going to implement it.
- Methods (internal and external) are designed and laid out
- Language specific, actual modules, code
- For example: detailed blueprints from which you could build a house

Using Patterns to Learn Principles

- Throughout this process we will:
  - Review core object-oriented terms
  - Review some object-oriented design principles/strategies and see how patterns utilize them
- Practice has shown this to be the best way to learn OO concepts

What Are Patterns?

- "A pattern is a solution to a problem in a context" [1]

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." [1]

- [1] Christopher Alexander - A Pattern Language

Elements of Patterns

- **Pattern name.** Gives us a way to refer to the pattern. [2]
- **The problem.** A particular pattern is applicable to certain types of problems. Part and parcel of a pattern is a description of what types of problems for which it is useful. [2]
- **The solution.** Patterns define a particular conceptual solution to the problem. [2]
- **Consequences.** Implementation decisions have certain tradeoffs. The consequences of these decisions and the forces underlying the pattern are essential aspects of the pattern. [2]
- [2] Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides.

Common Reasons to Study Patterns

- Re-uses existing, quality solutions
- Use common terminology
- *Shift to new level of thinking*

The Courtyard Pattern - Christopher Alexander

In the same way, a courtyard which is properly formed, helps people come to life in it.

Consider the forces at work in a courtyard. Most fundamental of all, people seek some kind of private outdoor space, where they can sit under the sky, see the stars, enjoy the sun, perhaps plant flowers. This is obvious. But there are more subtle forces too. For instance, when a courtyard is too tightly enclosed, has no view out, people feel uncomfortable, and tend to stay away … they need to see out into some larger and more distant space. Or again, people are creatures of habit. If they pass in and out of the courtyard, every day, in the course of their normal lives, the courtyard becomes familiar, a natural place to go … and it is used. But a courtyard with only one way in, a place you only go when you "want" to go there, is an unfamiliar place, tends to stay unused … people go more often to places which are familiar. Or again, there is a certain abruptness about suddenly stepping out, from the inside, directly to the outside … it is subtle, but enough to inhibit you. If there is a

Courtyard cont'd

transitional space, a porch or a veranda, under cover, but open to the air, this is psychologically half way between indoors and outdoors, and makes it much easier, more simple, to take each of the smaller steps that brings you out into the courtyard …

When a courtyard has a view out to a larger space, has crossing paths from different rooms, and has a veranda or a porch, these forces can resolve themselves. The view out makes it comfortable, the crossing paths help generate a sense of habit there, the porch makes it easier to go out more often … and gradually the courtyard becomes a pleasant customary place to be.

What we've done:

Identified the pattern.

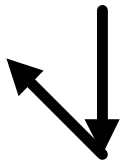
Discussed what we were trying to accomplish.

State how we could accomplish this.

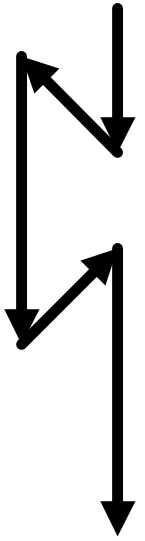
Talked about the forces that would hurt what we were trying to accomplish.

A Pattern in Carpentry

- Let's say two carpenter's are trying to decide on how to build a dresser.
- One says to the other: "should we build the joint by first cutting down, then cutting up at a 45 degree angle..."

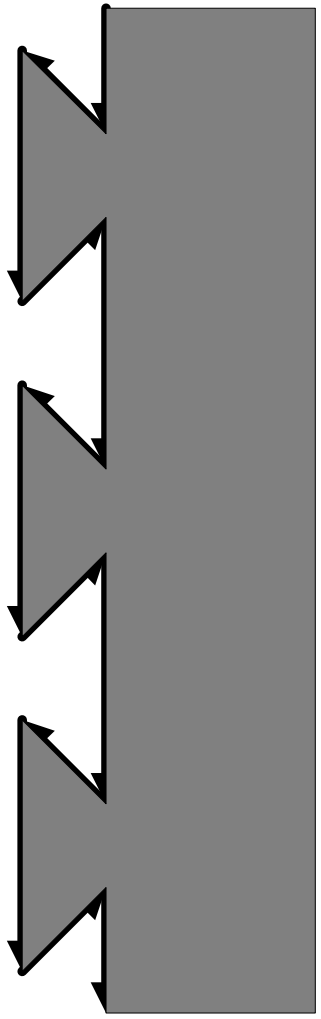


- "then back down and back up the other way, then back down"



We Are Describing a Dove-Tail Joint

- And keep this up until done.



Focusing on Detail Loses the Big Picture

- By having to describe how we implement the dove-tail joint, we can lose sight of why we might want to use it in the first place.
- Dove-tail joint Vs miter joint emphasizes:
  - do we want a strong, relatively expensive joint that is of high quality and will last forever
  - or do we want a weak, relatively inexpensive joint that is easy to make but not of high quality

Other Reasons to Study Patterns

- Common terminology assists learning across all levels
- Designs improved for modifiability (expand on good solutions)

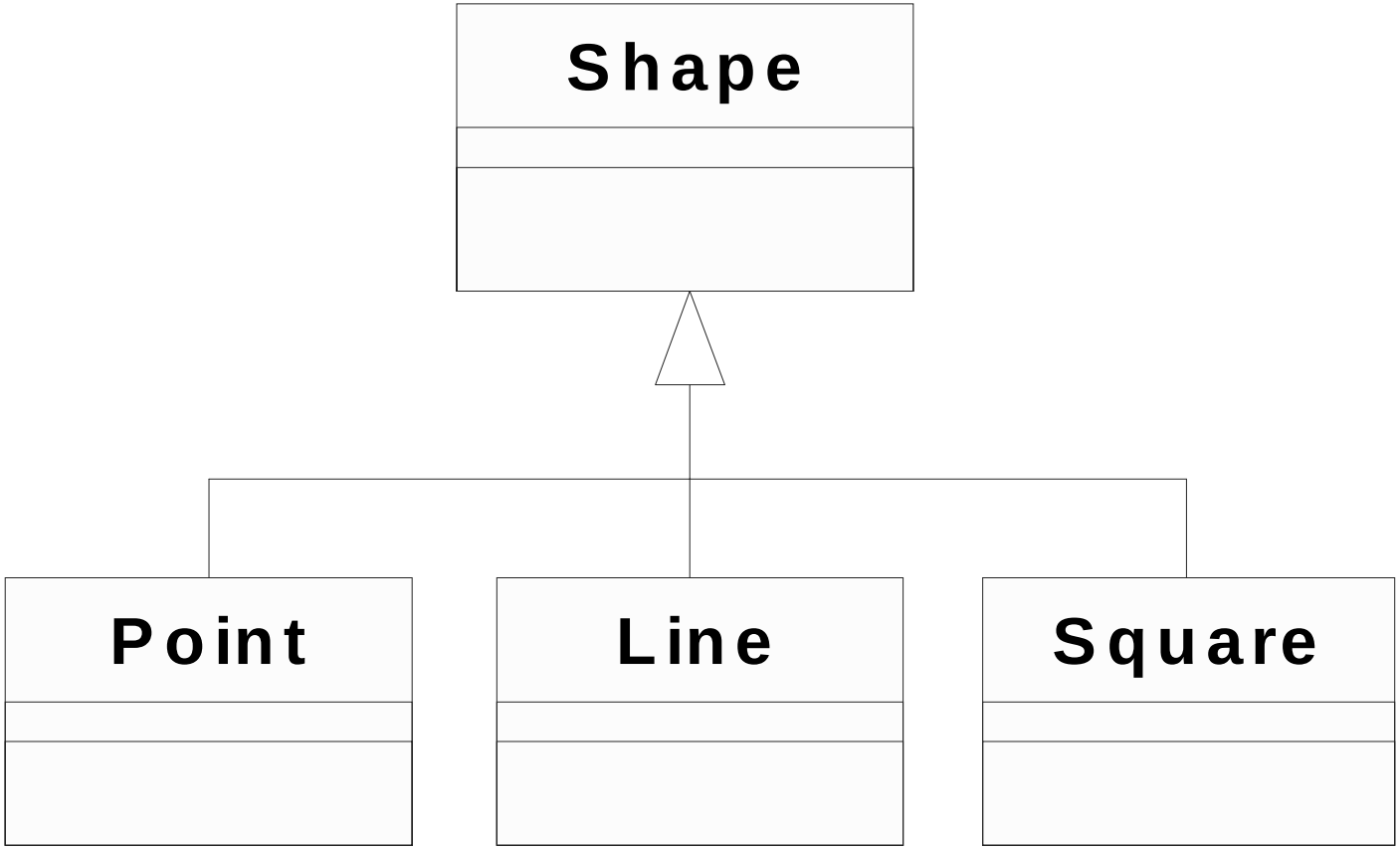
- We will find that design is not a process of synthesis, but one of differentiation.
- Design patterns can be used to help us build our designs better than normal methods.

The Adapter Pattern

- Intent: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. [3]

- [3] Design Patterns, Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides

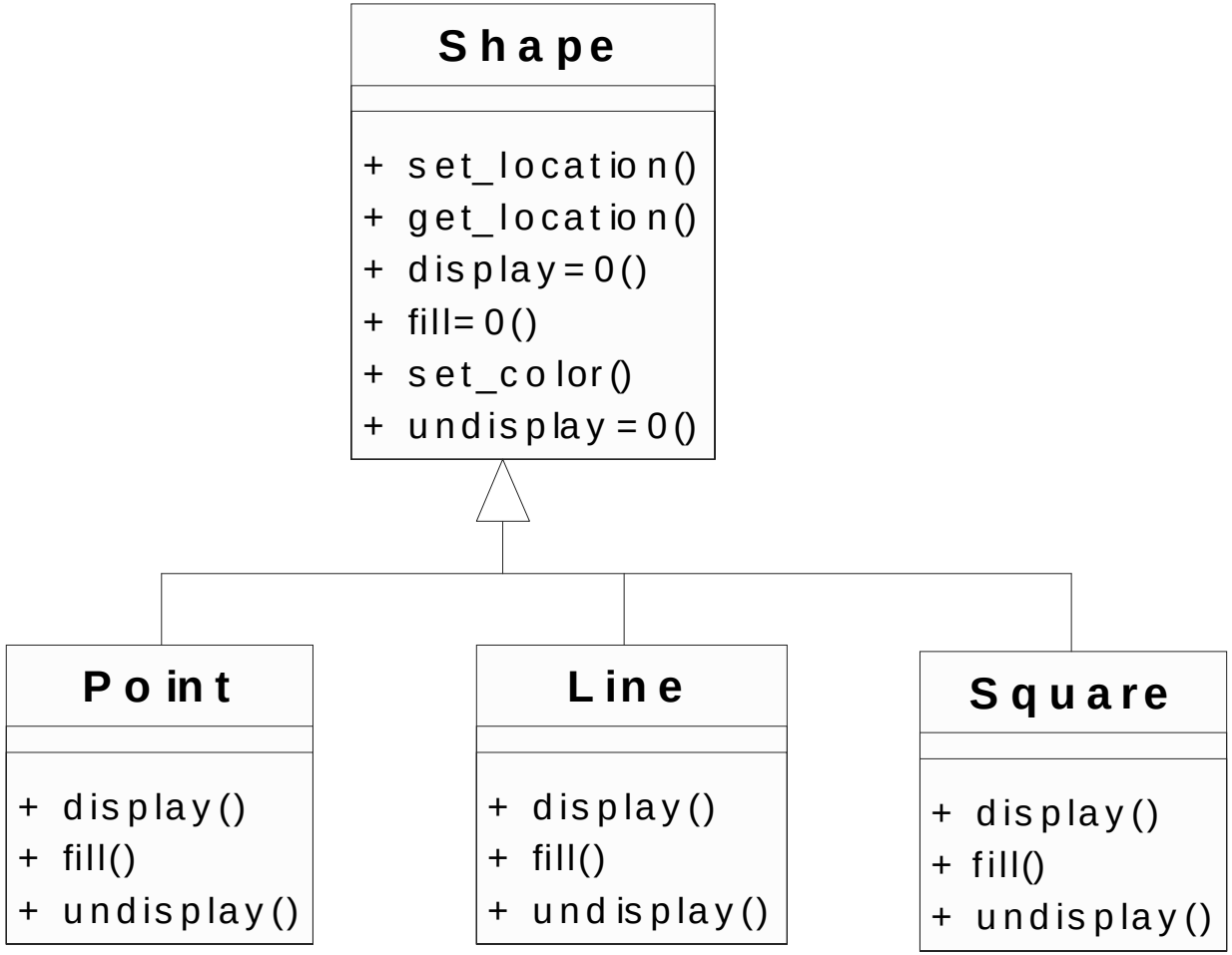
Our Problem



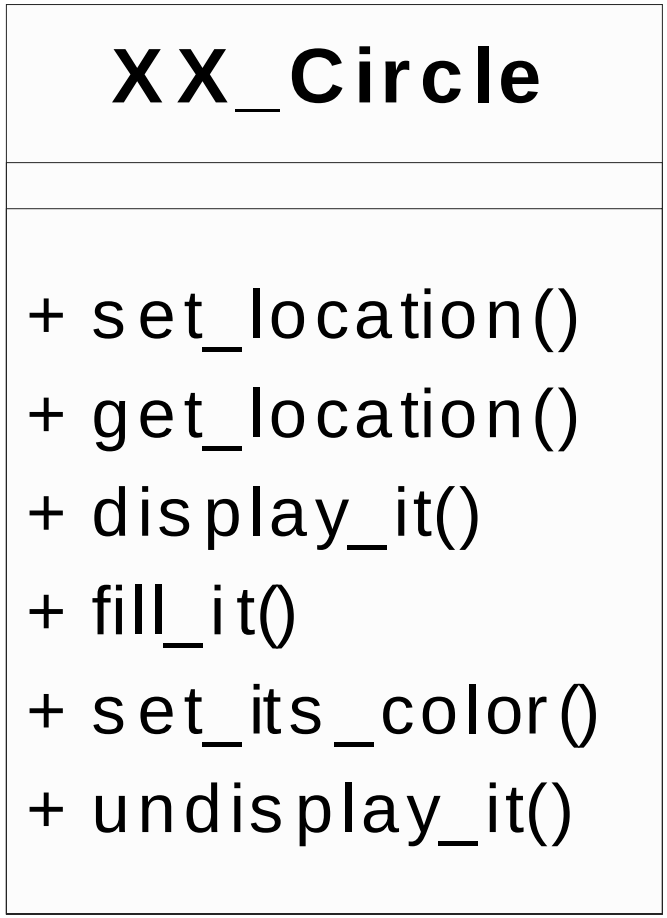
This Is an Example of Polymorphism

- Allows for different behavior
- Calling object does not need to know the exact type of object involved
- Calling object only needs to know the conceptual type of object
- In this case, anybody having *points*, *lines*, and *squares* only knows it has *shapes.*

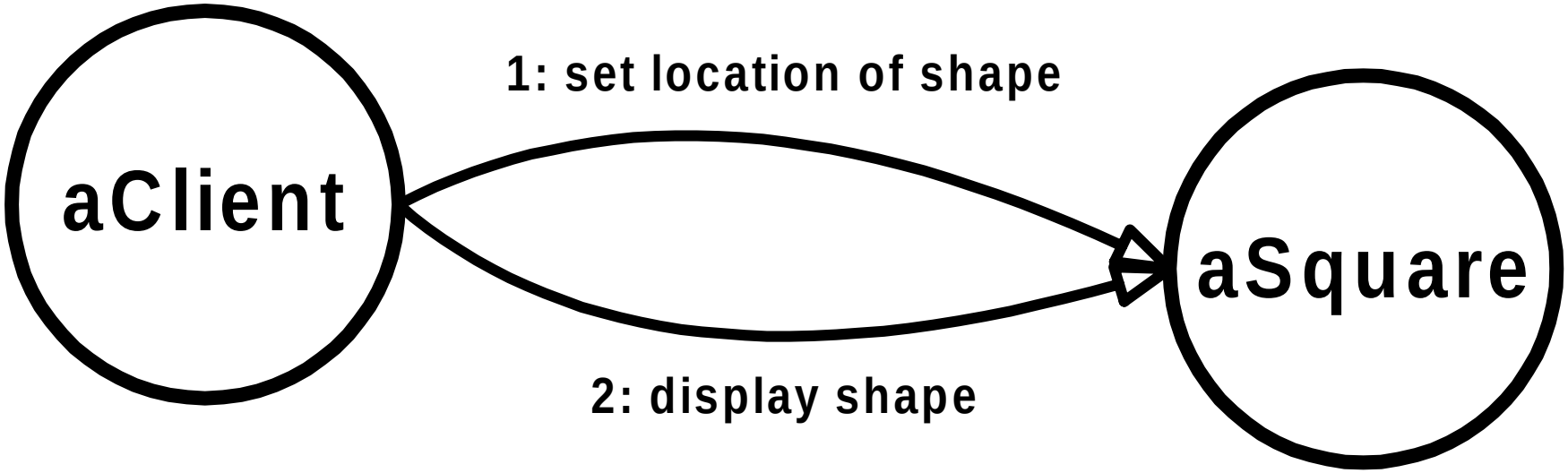
More Detail



What We Have

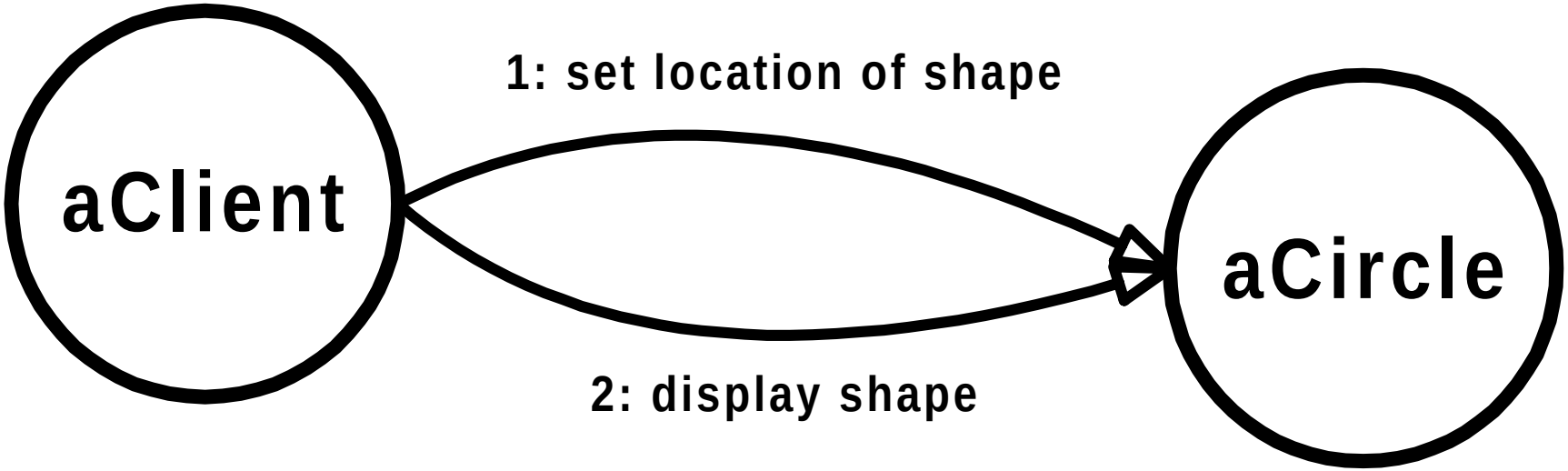


How Client Behaves With a Square



aClient **works with** ***aSquare*** **but only knows it's a** ***Shape***

What We Want



want *aClient* to work with *aCircle* in the same way

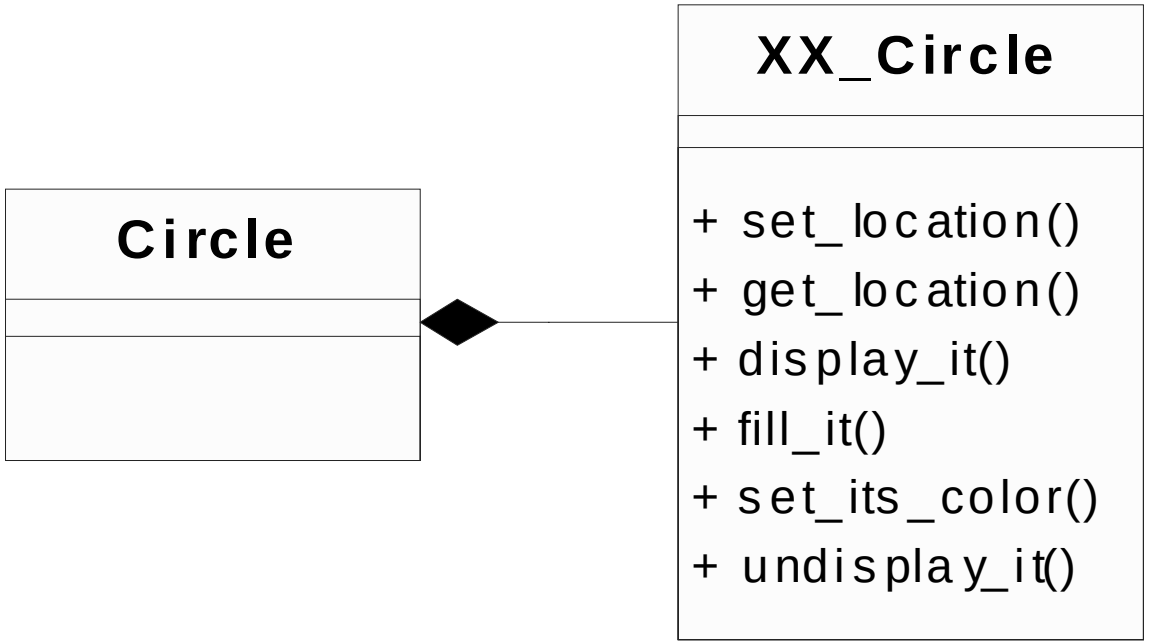
Our Problem

- Our client can't behave with our XX_Circle in the same way it does with the Square because XX_Circle is not a Shape
- Author of XX_Circle may not be willing or able to change XX_Circle's interface

Composition

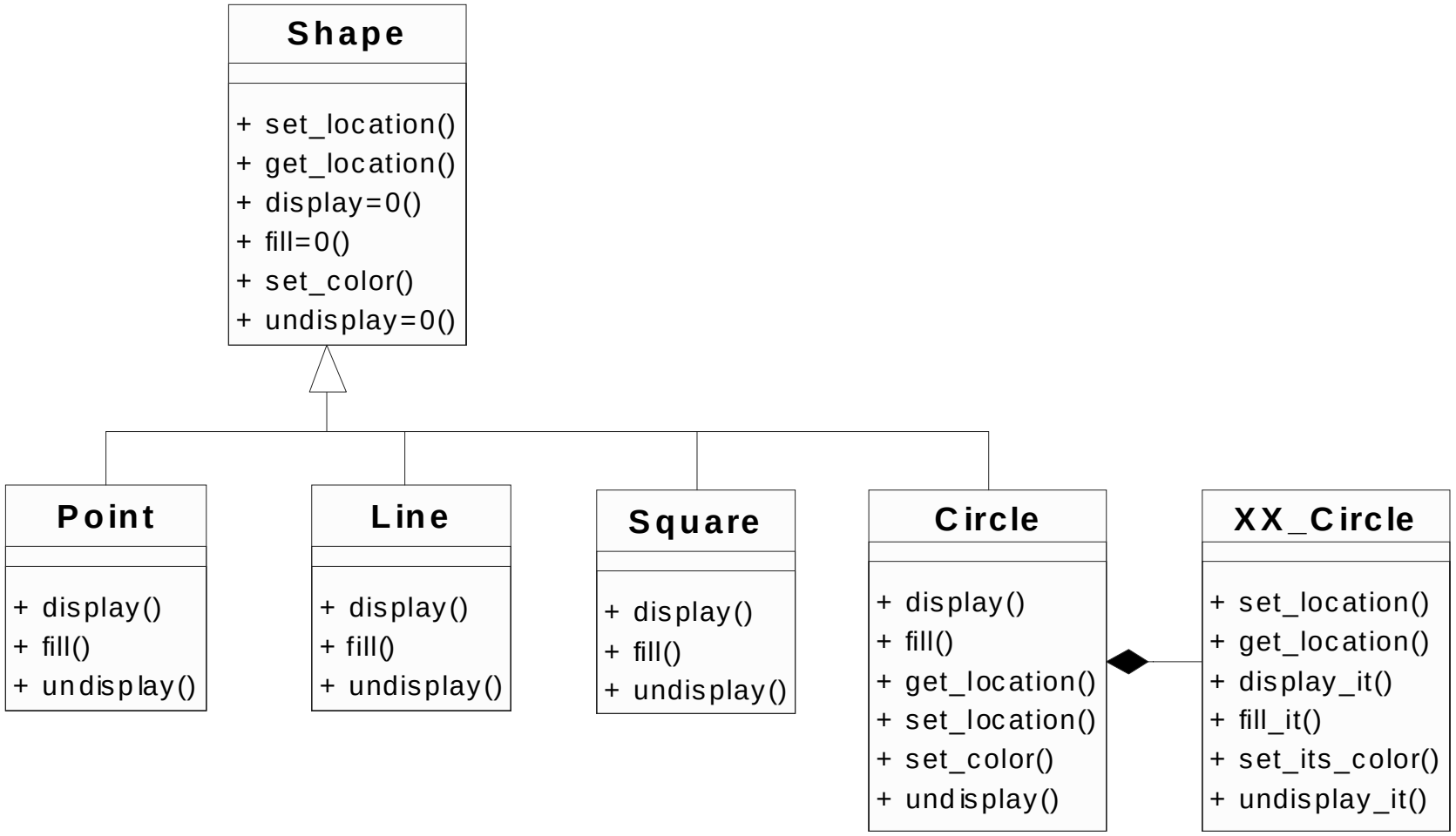
- When one object ‘contains’ another object.
- Like an engine in a car.

How to Implement the Pattern



If circle contained an XX_circle, it could handle the ‘communication’ with the client and let XX_circle handle the work

The Pattern in its Context



A C++ Example

IN HEADER

```
class circle : public shape
{  • • •
     private:
      XX_circle *pxc;
    • • •
}
```

IN CONSTRUCTOR

instantiate and initialize XX_circle
point to it with pxc

IN CODE

```
void circle::display ()
{
    pxc->display_it();
}
```

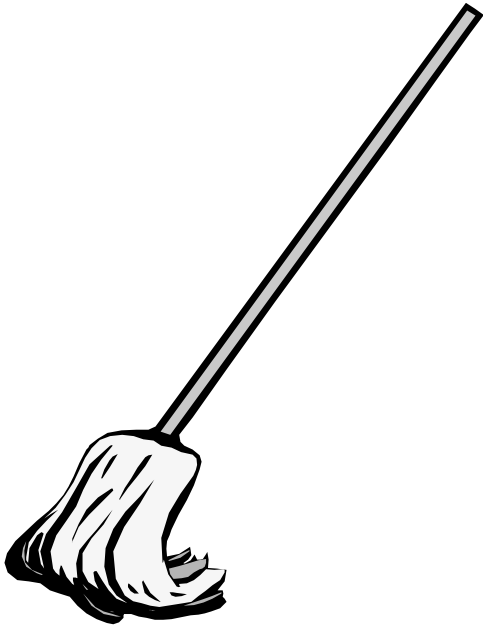
A Java Example

```
class circle extends shape {
   …
   private XX_circle pxc;
   …
   public shape () { pxc= new XX_circle(); }

   void public display() {
      pxc.display_it();
   }
}
```

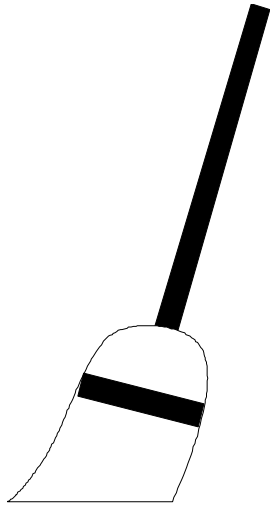
A Smalltalk Example

```
Shape subclass: #Circle
   instance Variables: ‘xCircle…’
   class Variables: ‘ ’
   pools: ‘’

display
   ^xCircle display_it
```

Are these the same??



Mop



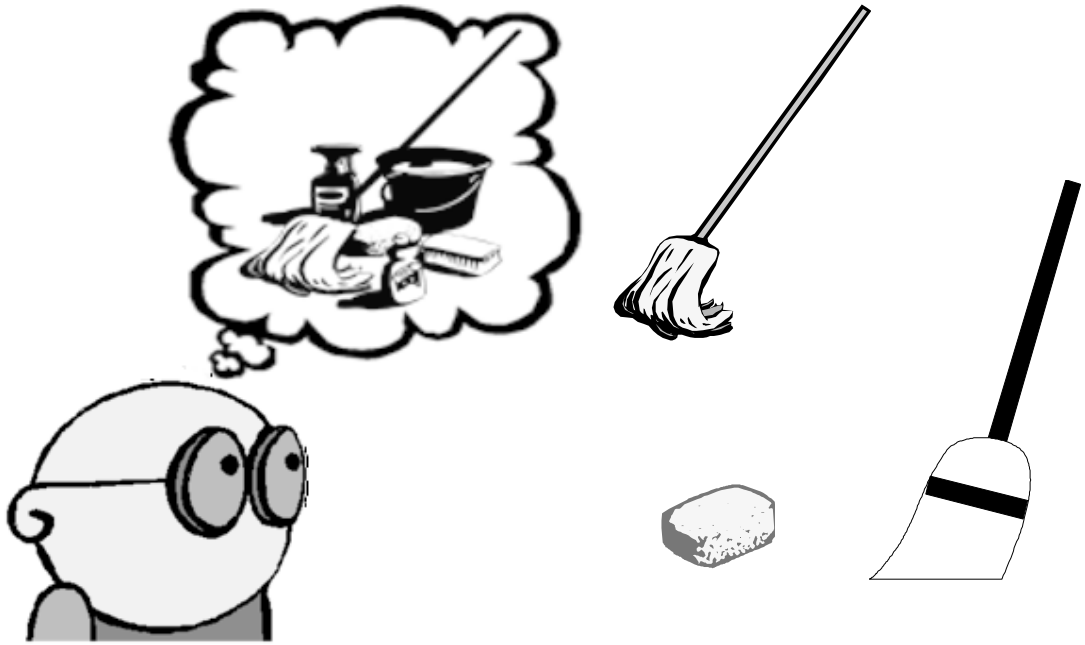
Broom

Sponge

As Any Good Consultant Will Tell You: "*IT DEPENDS!*"

- As specific objects: they are different
- As a concept: they are all Cleaning Utensils

Where does “Cleaning Utensil” exist?



Not in the real world, but in our thoughts as an abstraction classification!

A “Cleaning Utensil” does not exist, but specific kinds do!

Abstract Class

- A class that is never instantiated
- Used to define an interface for the real-world classes (i.e., used to define how to communicate with the objects that are derived from the abstraction)
- Represents a concept
- For example, a ‘cleaning utensil’ was an abstract class for ‘mops, ‘brooms’, and ‘sponges’.
- In the adapter pattern, a ‘shape’ was an abstract class for ‘points’, ‘lines’, ‘squares’, and ‘circles’.

This Is an Example of Polymorphism

- Allows for different specific behavior that is conceptually defined by the abstract class
- The calling object does not need to know the exact type of object involved - it only needs to know the conceptual type of object
- In our ‘shape’ example, anybody having *points*, *lines*, and *squares* only knows it has *shapes*.
- Both hides implementation details and allows for new types of implementations
- Allows for new occurrences of a type without having to change the software that uses it.

Open - Closed Principle

- Ivar Jacobson said: “All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version.
- Bertrand Meyer summarized this as: ***Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.***
- In English this means: *design modules so that they never change. When requirements change, add new modules to handle things.*
- (good article on Open-Closed Principle at www.oma.com under publications)

How Do We Implement the OCP?

- Abstraction is the key conceptually, abstract classes are the key implementationally.
- Abstract classes represent a fixed behavior definition with an unlimited number of possible implementations.

How'd We Do Re Principles?

- Did we follow the open-closed principle?
- Specifically:
  - were we open for extension? (i.e., were we able to make the changes we wanted to?)
  - were we closed for modification? (i.e., did we avoid changing anything we had?)

Encapsulation

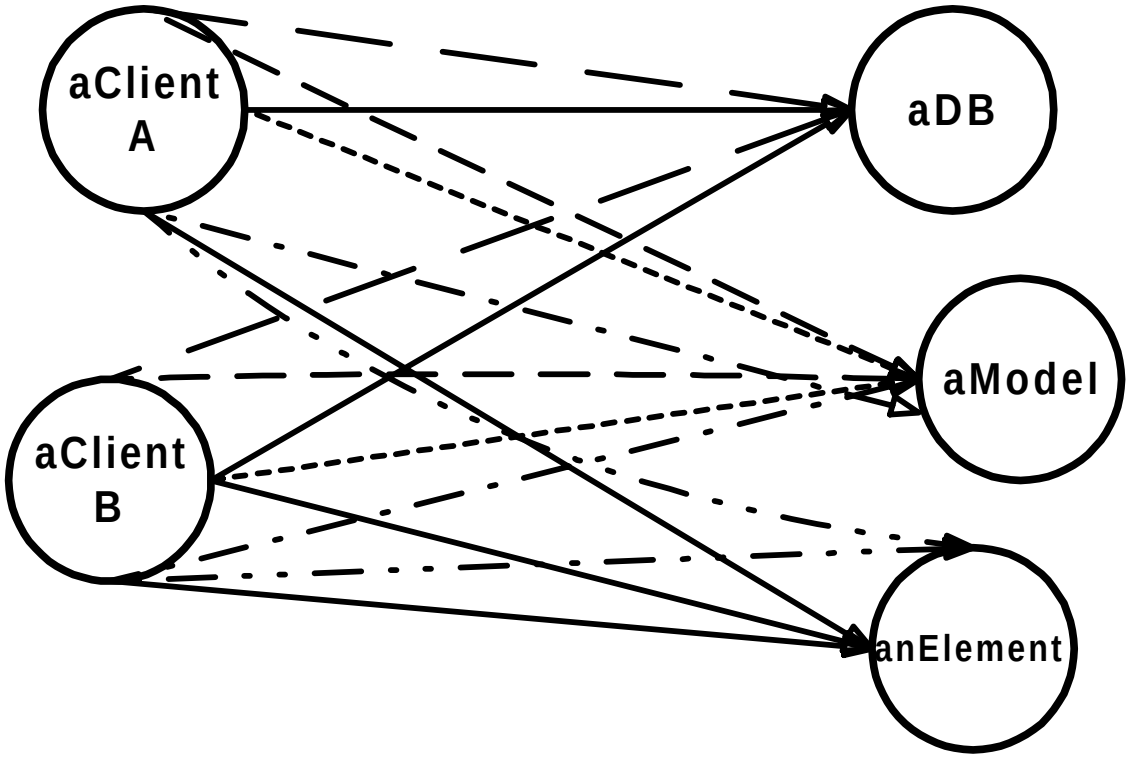
- Often called data-hiding
- Can also hide behavior
- Means can’t see what is being encapsulated

The Facade Pattern

- Intent: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. [4]

- [4] Design Patterns, Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides

Our Situation



aDB, aModel, anElement are all needed to get the job done
Many different methods are used, often in the same way.

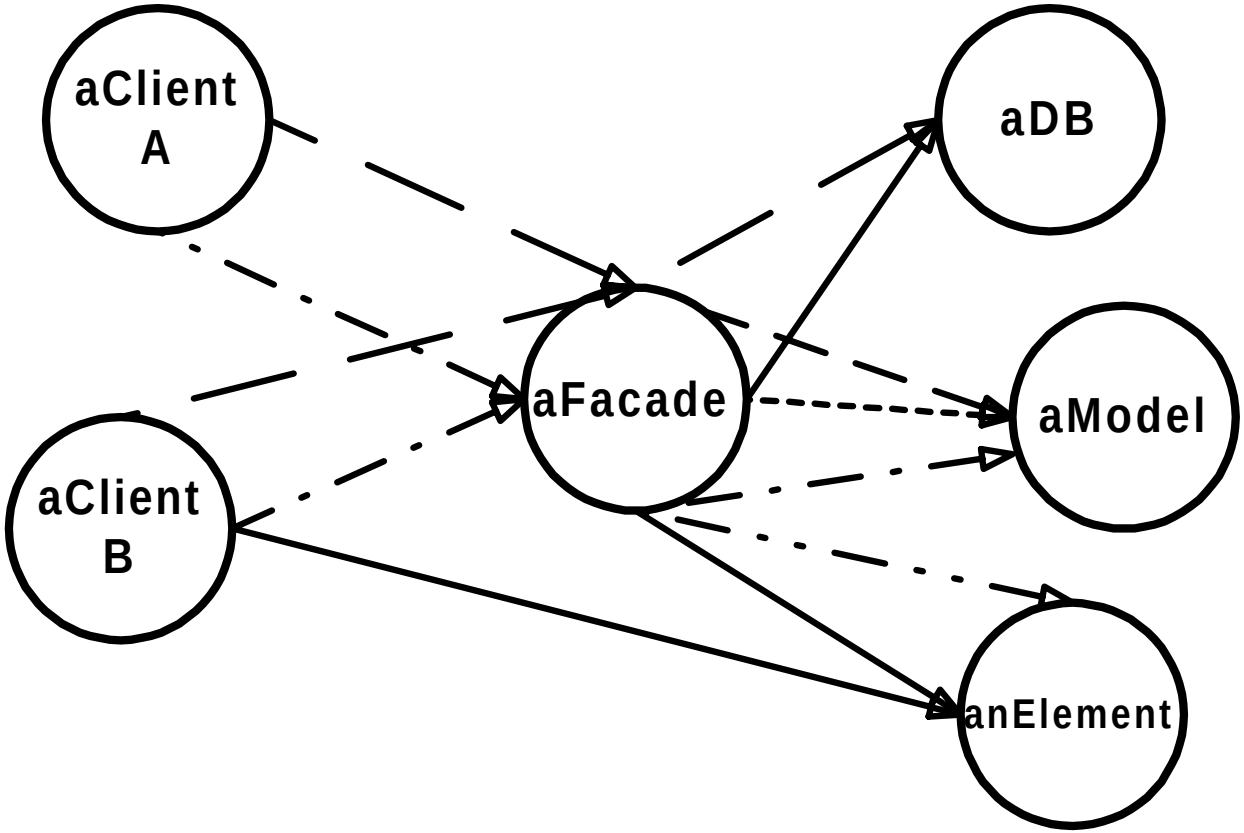
Our Problem

- Whoever writes the clients must learn pretty much the same things.
- Also, there is considerable duplication in the clients.
- We want to lower the learning curve and eliminate the duplication

Our Solution

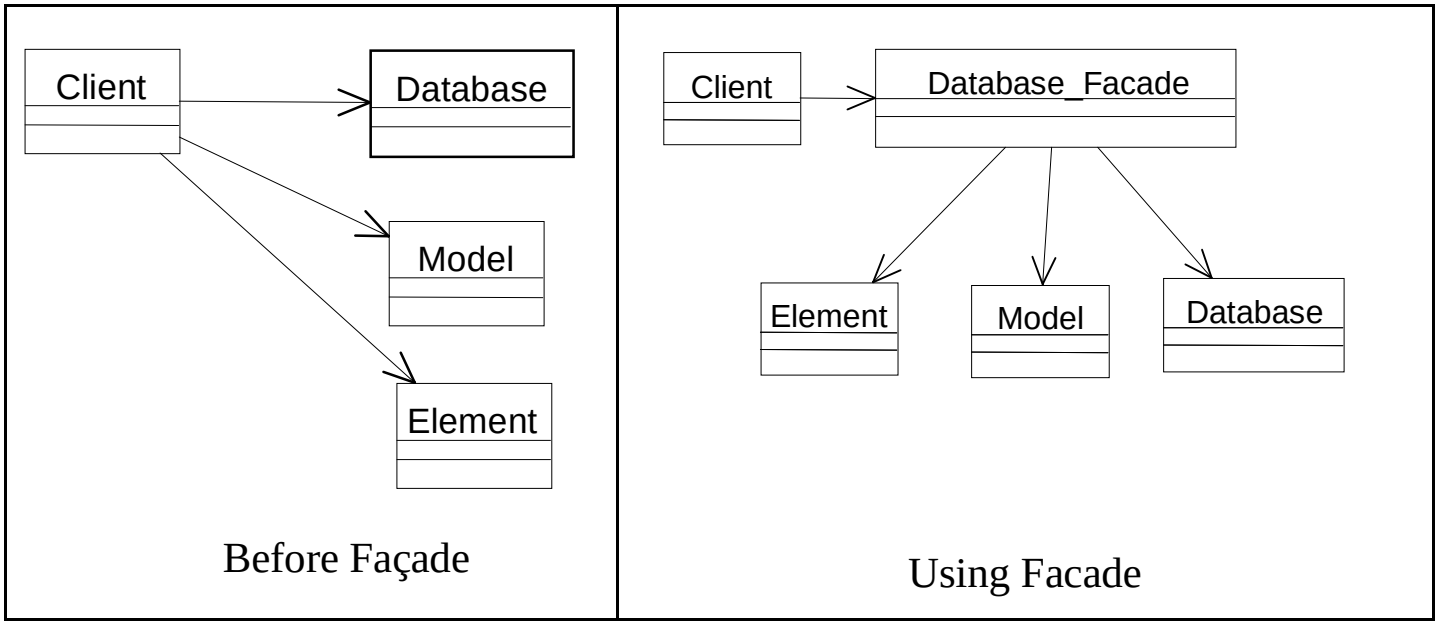
- Create a new class that both clients can use
- It handles the most common functions
- It allows clients to go directly to the underlying methods when needed

Our Solution

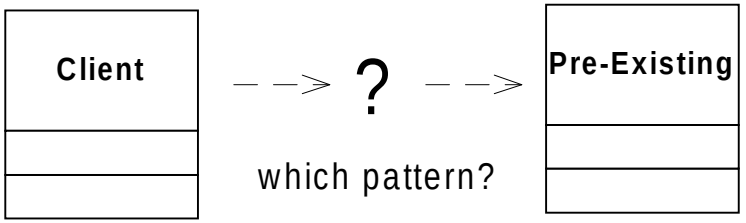


The Facade class creates a new interface for the commonly used classes that is easier to use.

Class Diagrams



Comparing Façade With Object Adapter



	Façade	Object Adapter
Have pre-existing classes?	Yes	Yes
Have an interface we must design to?	No	Yes
Have to make an object behave polymorphically?	No	Probably
Want to make a new interface to simplify things?	Yes	No

Commonality/Variability Analysis

- First find what is common across domain
- Identify where things vary
- Identify how things vary
- This is preparation for identifying potential classes

Find What Varies and Encapsulate It

- Identify varying behavior
- Define abstract class that allows for communicating with objects that have one case of this varying behavior

Favor Composition Over Inheritance

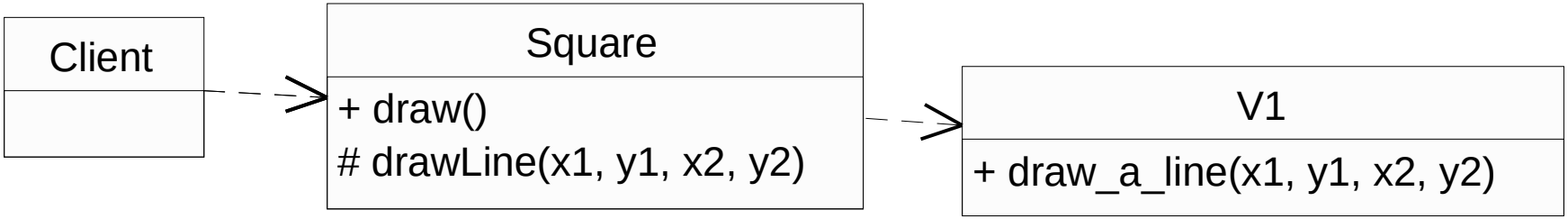
- If can define a class that encapsulates variation, contain (via composition) an instance of a concrete class of the abstract class defined earlier
- Allows for decoupling of concepts
- Allows for deferring decisions until runtime
- Small performance hit

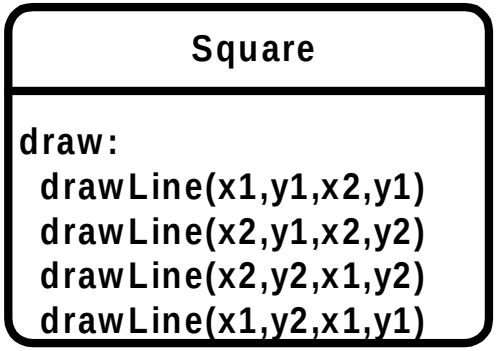
The Bridge Pattern

- Intent: De-couple an abstraction from its implementation so that the two can vary independently [3]

- [3] Design Patterns, Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides

Our Starting Situation



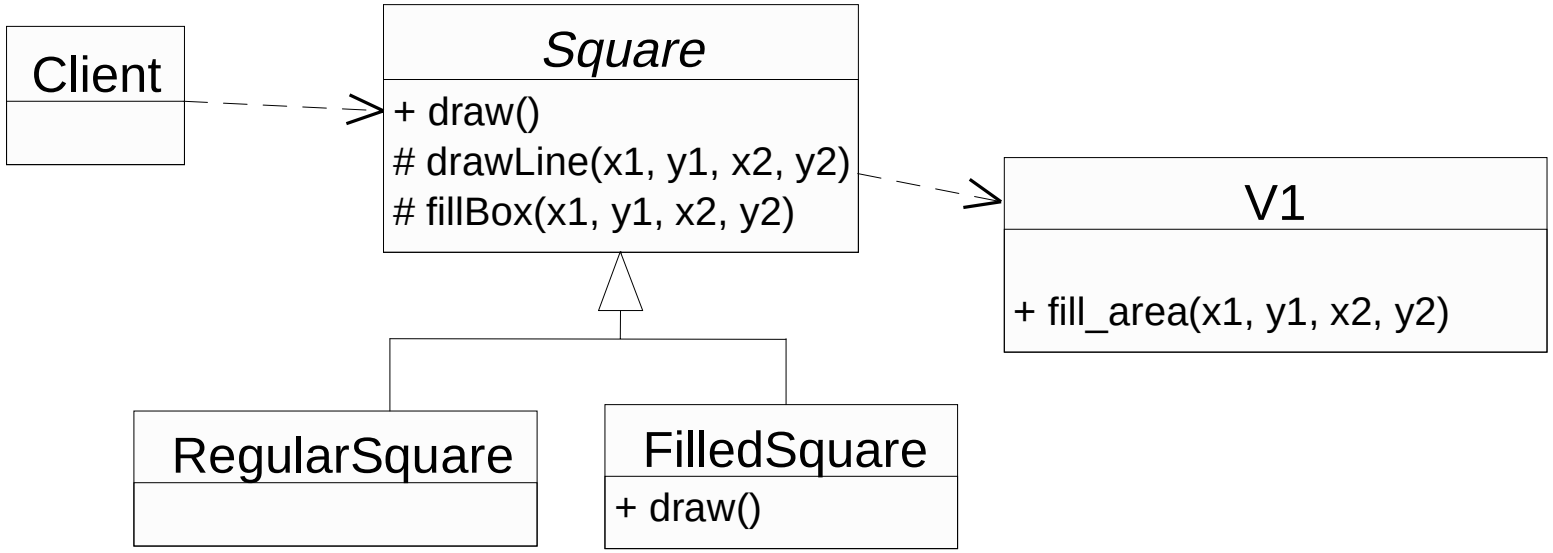


drawLine calls draw_a_line in V1

Notes on UML (Unified Modeling Language):
+ before methods means it is a public method
# before methods means it is a protected method
- before methods means it is a private method

Italics means class or method is abstract

Handling Requirement for New Class



RegularSquare

draw (from Square):
drawLine(x1,y1,x2,y1)
drawLine(x2,y1,x2,y2)
drawLine(x2,y2,x1,y2)
drawLine(x1,y2,x1,y1)

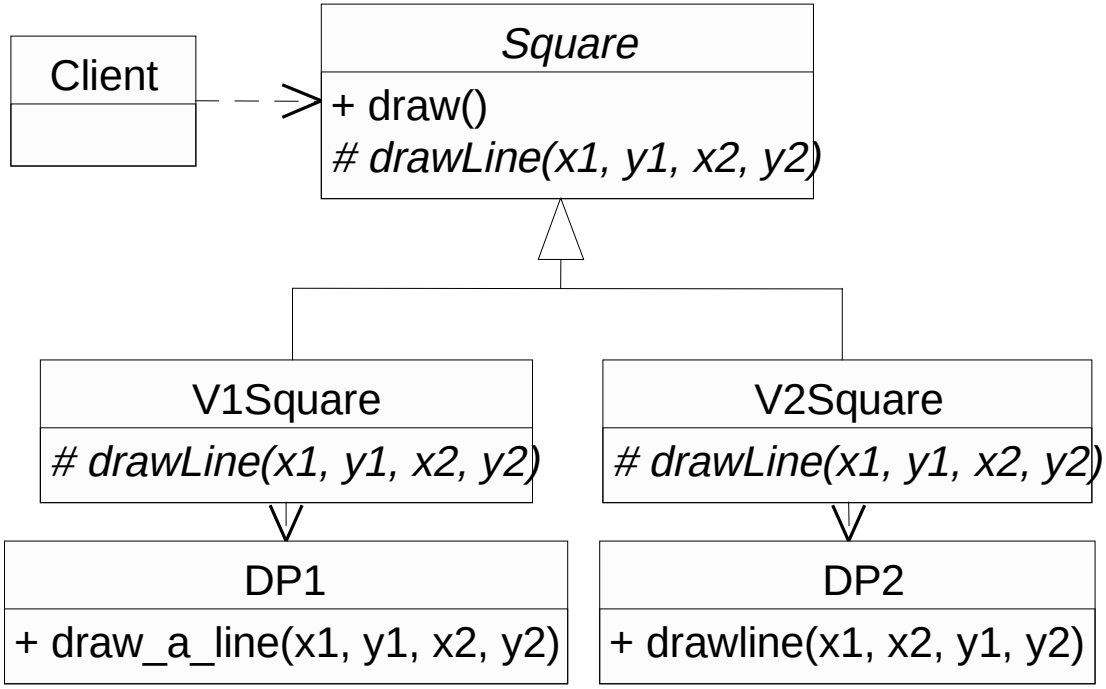
drawLine calls V1's draw_a_line as draw_a_line(x1,y1,x2,y2)

FilledSquare

draw:
Square::draw()
fillBox()

fillBox calls V1's fill_area as fill_area(x1,y1,x2,y2)

Handling Requirement for New Framework



Square

draw:
drawLine(x1,y1,x2,y1)
drawLine(x2,y1,x2,y2)
drawLine(x2,y2,x1,y2)
drawLine(x1,y2,x1,y1)

draw will be used by all Squares

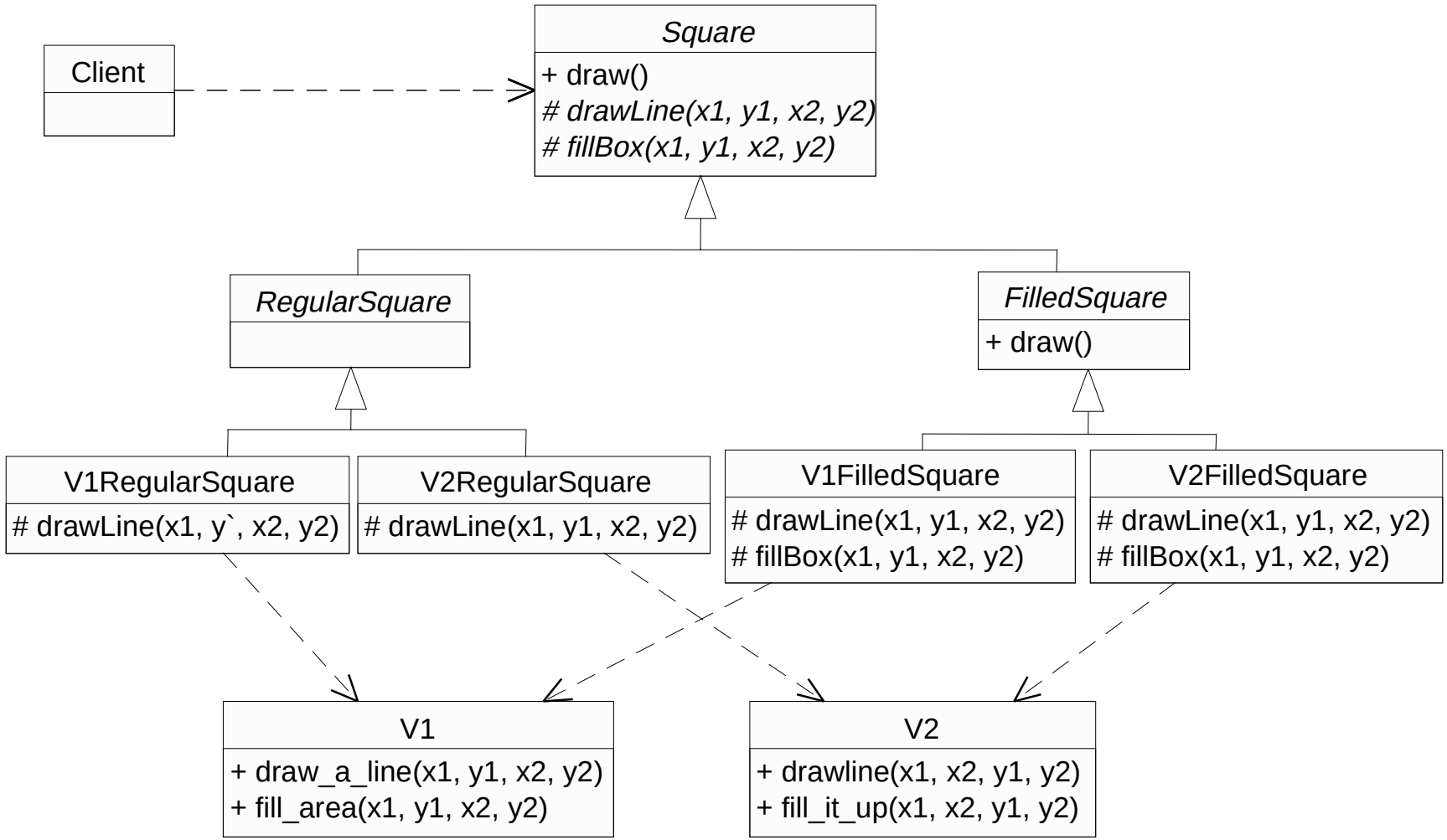
V1Square

drawLine:
call DP1's draw_a_line() passing parameters through

V2Square

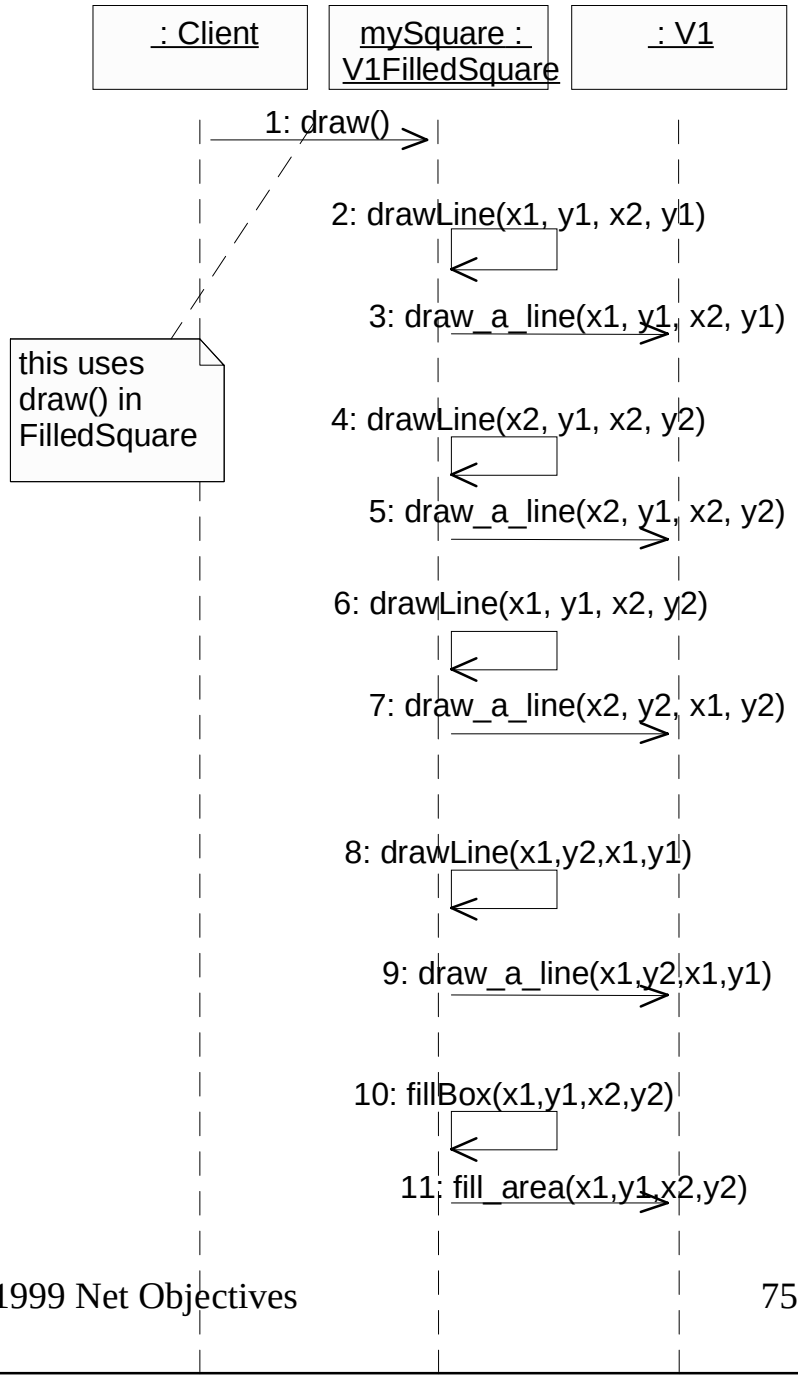
drawLine:
calls DP2's drawline() with parameters transposed

Handling Both New Requirements



How This Works When Running

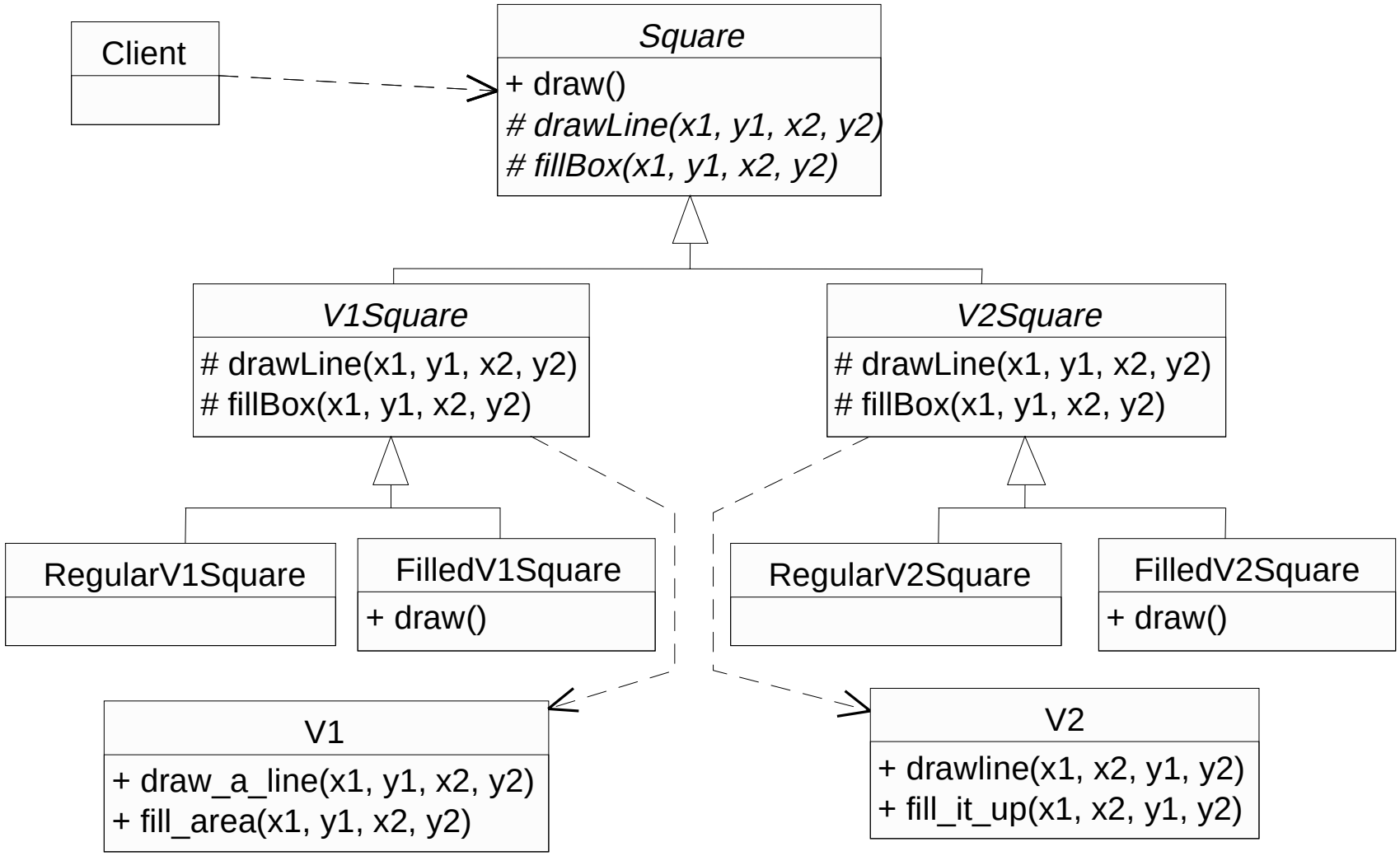
In this example we have a V1FilledSquare object called mySquare by Client. mySquare knows to use a V1 object. Note: Client only knows that mySquare is a Square, it doesn't know what type of Square it is.



Problems With This Approach

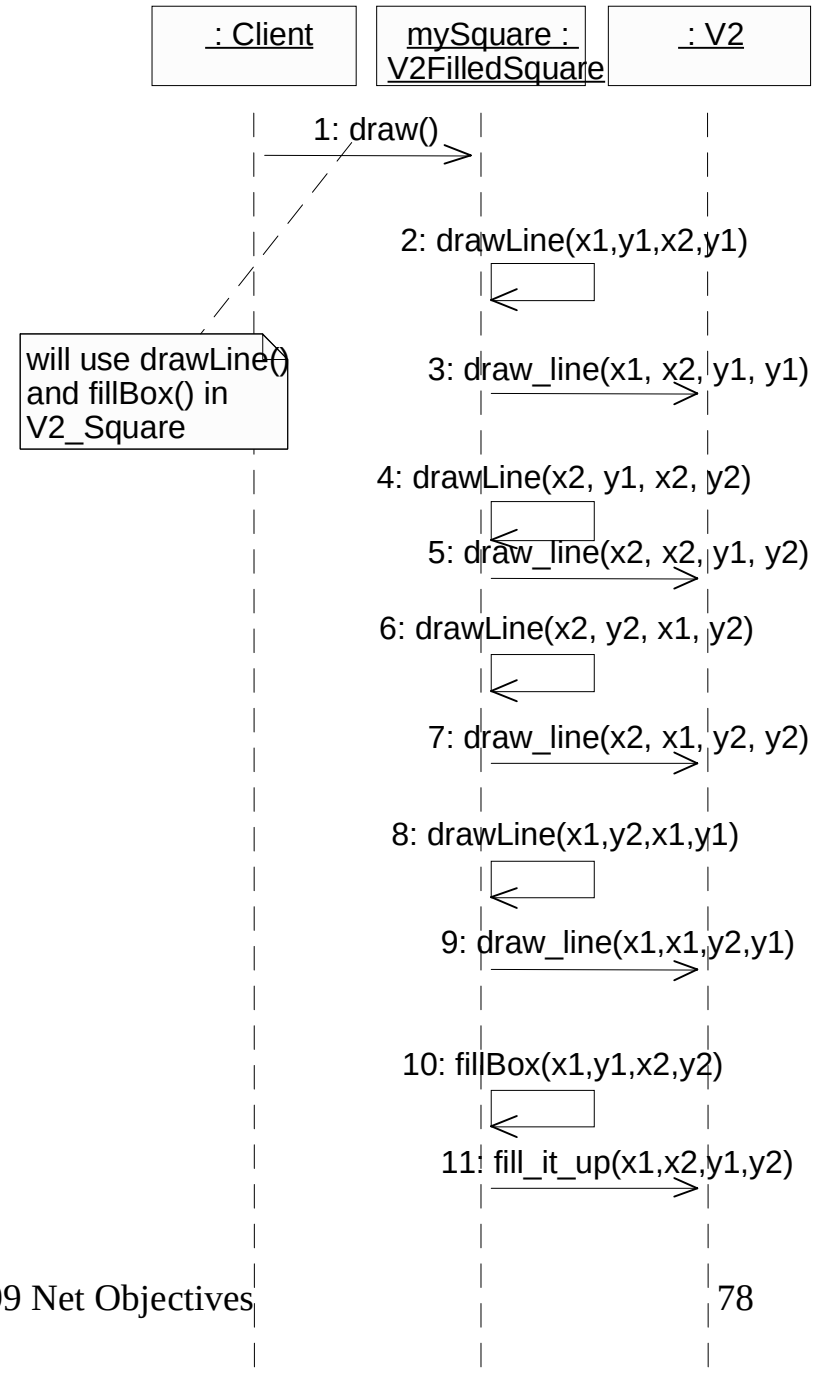
- Doesn't scale well
  - if add another type of Square, we have 6 implementations
  - number of implementations will equal # types of Squares times # of drawing programs
- Redundancy
  - clearly accessing each of the drawing programs across the different types of Squares will involve duplication or require greater modularization
- Confusing and complex

Handling Requirements for New Class and New Platform



How This Works When Running

In this example we have a V2FilledSquare object called mySquare by Client. mySquare knows to use a V2 object. Note: Client only knows that mySquare is a Square, it doesn't know what type of Square it is.



Data Abstraction

- A common term in object-oriented technology is data abstraction.
- Data abstractions are ways of thinking about our problem domain. The abstractions give us the keys to looking at our problem domain -- seeing what are the concepts upon which our entities/objects are based.
- ‘Data abstraction’ is usually used to refer to our entities - we need something broader
- We will use another more descriptive term- the differentiator.

The Differentiator

- A differentiator is an abstraction that creates a distinction in our problem domain.
- I prefer the term differentiator over data abstraction because it implies the concept creates distinctions in our way of looking at our problem.

What Are the Differentiators Present Here?

- In current example, we have two differentiators:
  - the concept of squares
  - different implementations
- It is best to state it in terms of our problem domain, not in terms of a solution.

Differentiators for Earlier Patterns

- Adapter: we have the right object, but the wrong interface.
- Facade: we have a set of objects that will do our job, but we need a simpler interface.

Discovering The Bridge Pattern

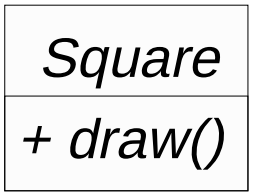
- In discovering the bridge pattern we'll follow the following approach
  - find what is common in the problem domain
  - find which of these commonalities vary, and how they vary
  - identify the concept of what is varying -- that is, what concept can incorporate all of the variation

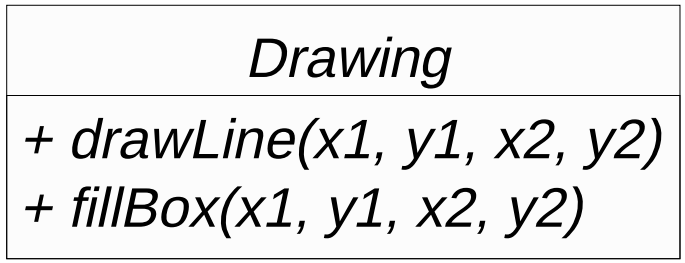
Discovering the Bridge Pattern cont'd

- define an abstract class to represent each common concept
- define concrete classes for each of these abstractions that represent a particular variation
- see how these different abstractions relate to each other

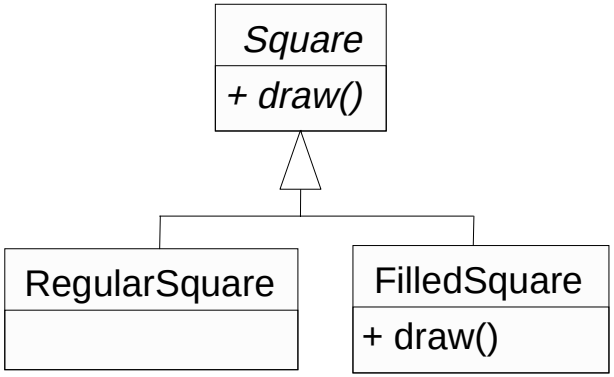
We Have the Following Commonalities

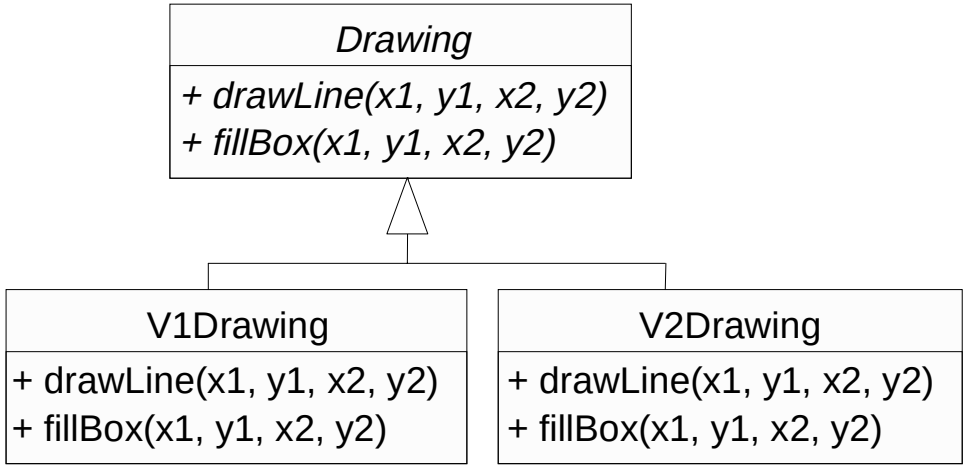
- Our square class represents our commonality of abstraction. In fact, this class is already abstracted and derived.
- We also have an implementation which varies.
- This gives us:



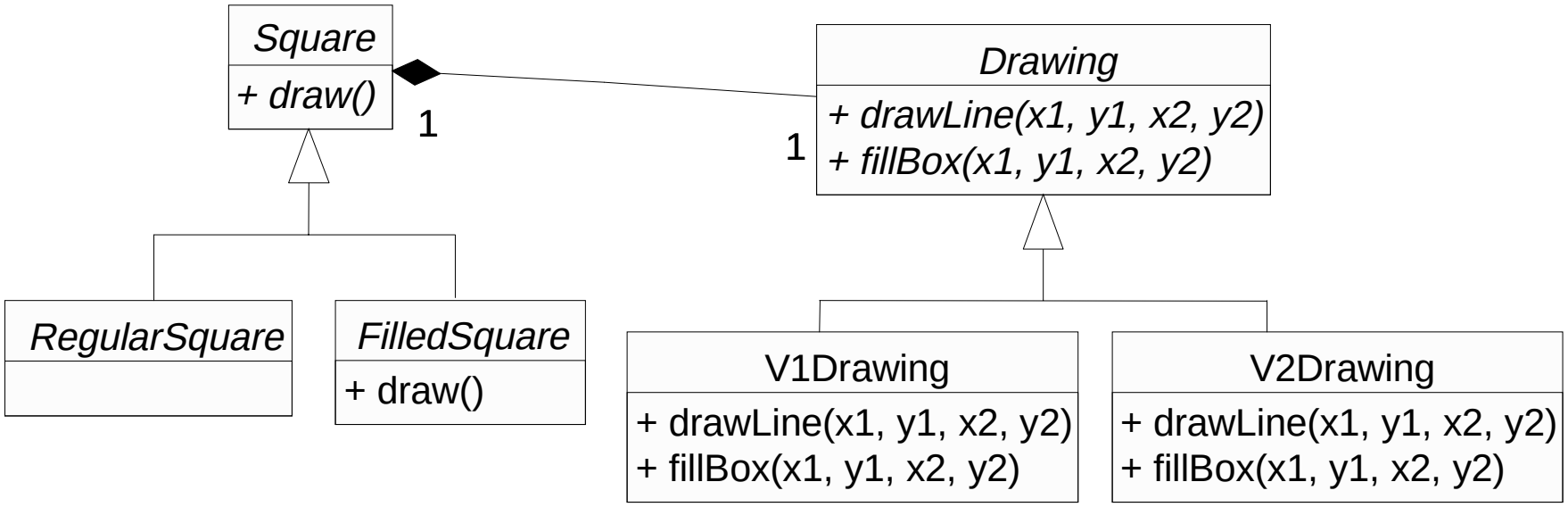


Derive Variations

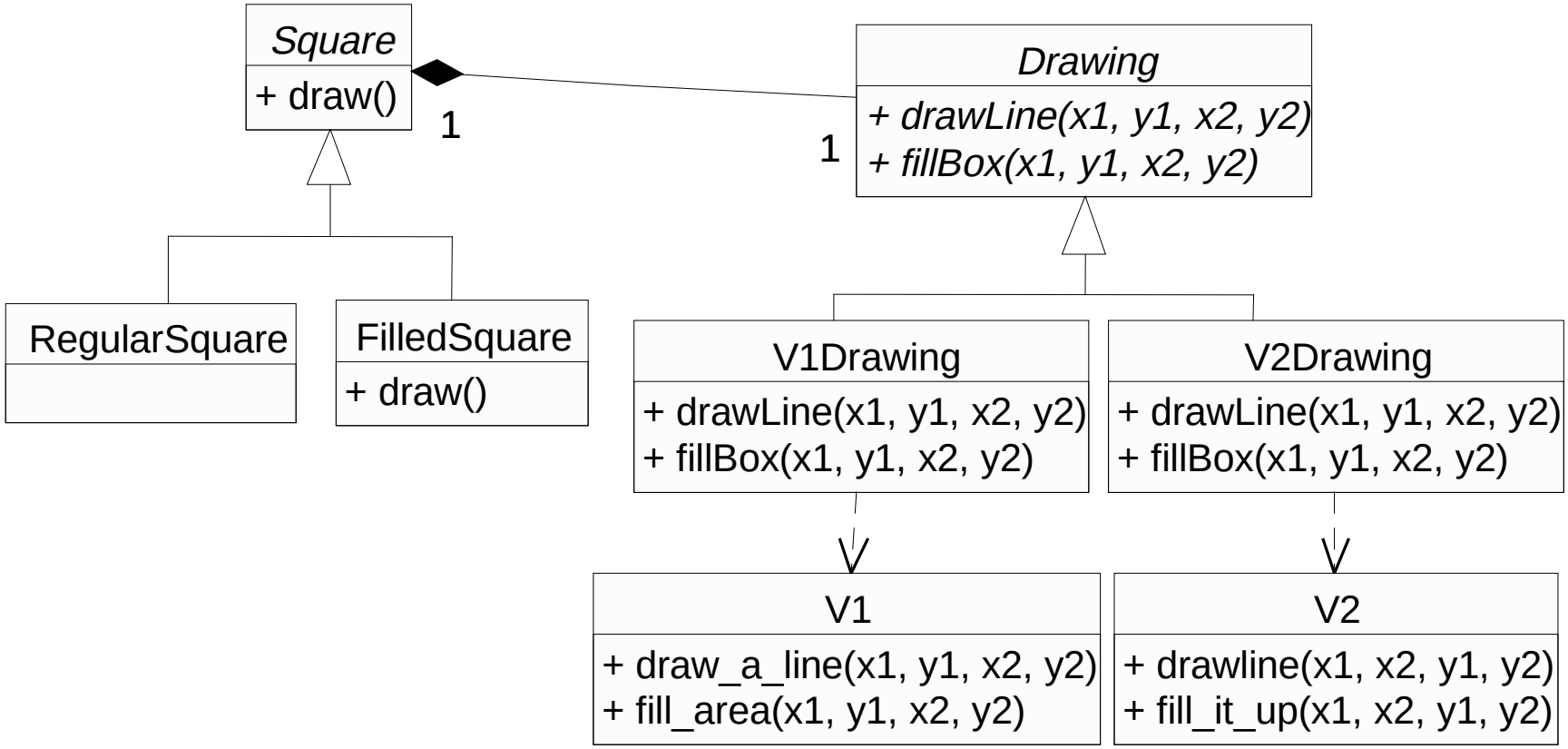




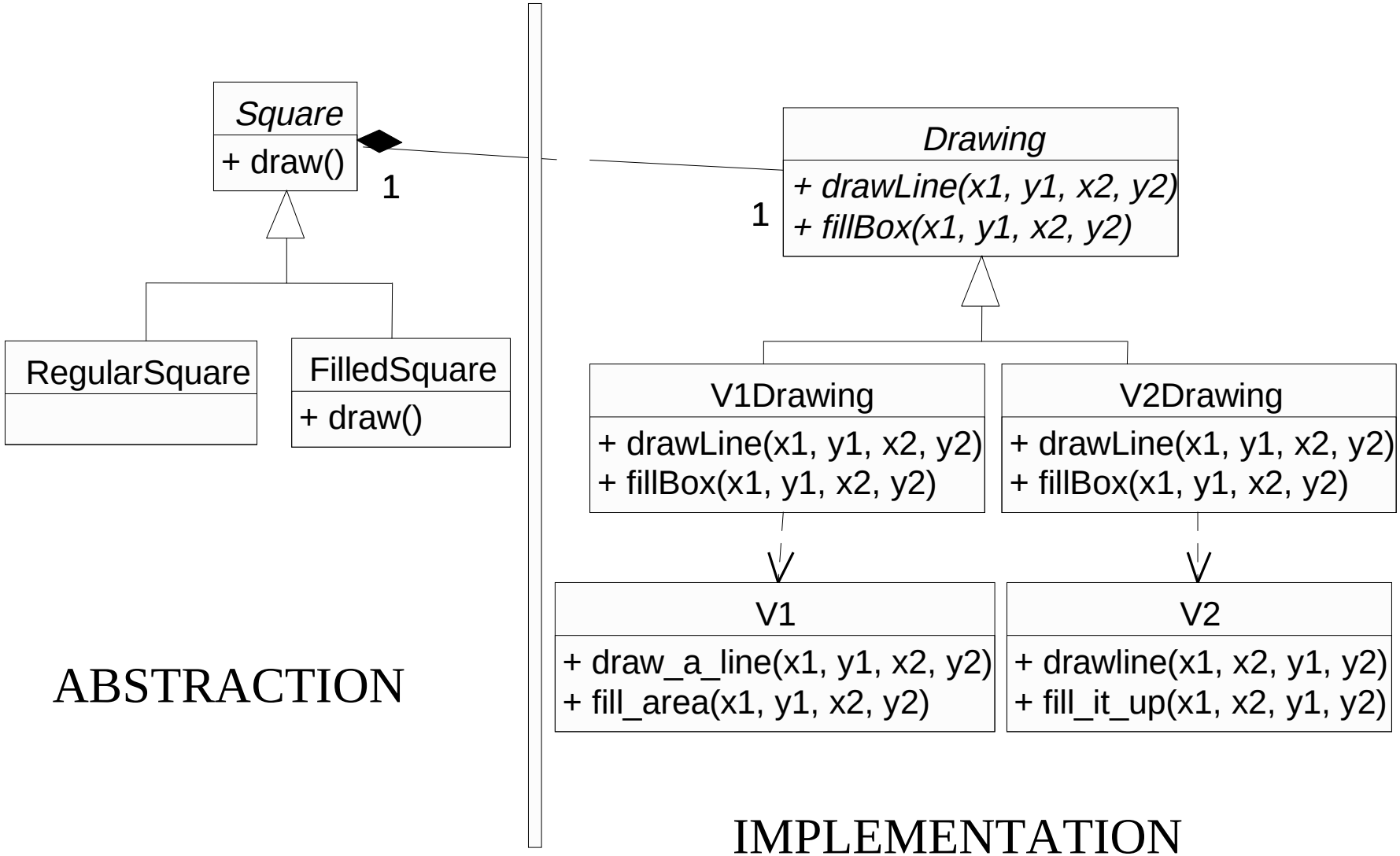
Show Relationship



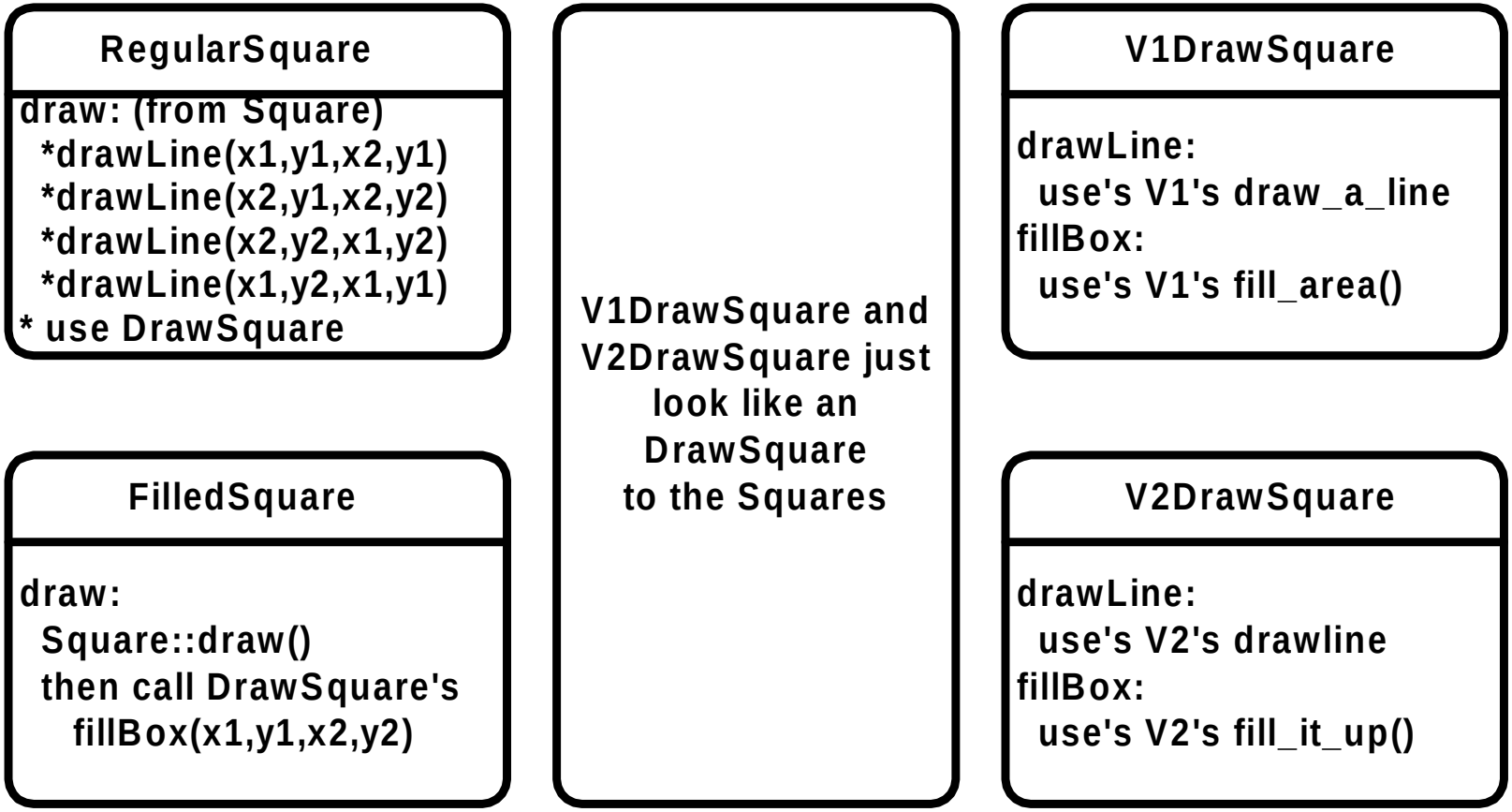
Finished Pattern



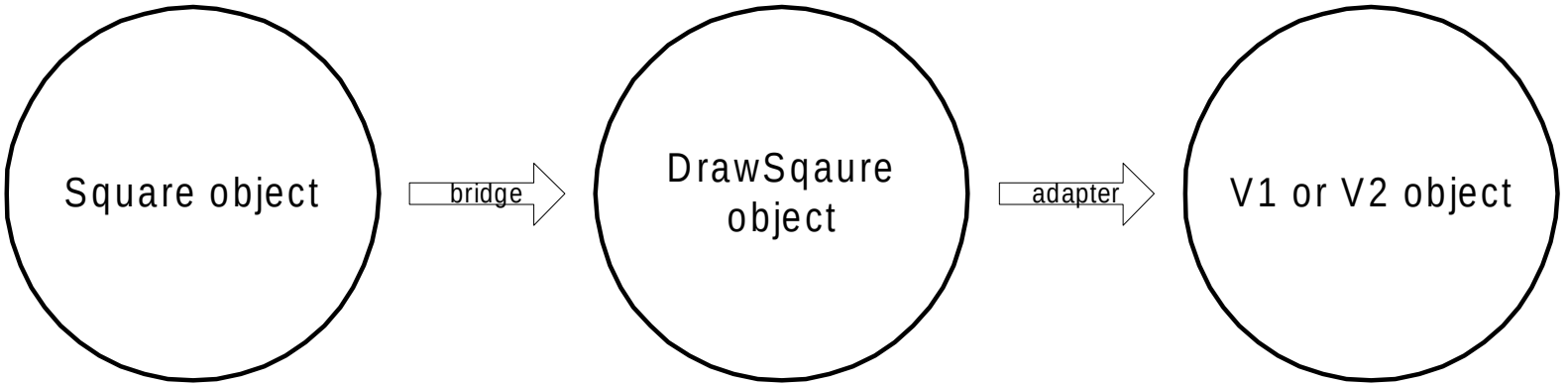
Illustrating the Separation



How It Works



Another Way To Look At It



This is actually a RegularSquare or a FilledSquare but the Client can't tell since they both act the same

This is actually a DrawSquareV1 or a DrawSquareV2 but the Square object can't tell since they both act the same

This must be the correct type of object, but the DrawSquare that uses it will know.

Example Implementation in C++

```
void main (String argv[]) {
  Square *s1;
  Square *s2;
  Drawing *dp;

  dp= new V1Drawing;
  s1= new RegularSquare( dp,1,1,2,2);

  dp= new V2Drawing;
  s2= new FilledSquare( dp,2,2,4,4);

  s1->draw();
  s2->draw();

  delete s1;
  delete s2;
}
```

NOTE: Memory management not tested.
Includes not shown.

```
class Square {
  public:
    Square (Drawing *dp, double x1, double y1, double x2, double y2);
    virtual void draw();
  private:
    double _x1, _y1, _x2, _y2;
    Drawing *_dp;
};
Square::Square (Drawing *dp, double x1, double y1, double x2, double y2) {
    _dp= dp;
    _x1= x1;
    _y1= y1;
    _x2= x2;
    _y2= y2;
}
void Square::draw () {
    drawLine( _x1, _y1, _x2, _y1);
    drawLine( _x2, _y1, _x2, _y2);
    drawLine( _x2, _y2, _x1, _y2);
    drawLine( _x1, _y2, _x1, _y1);
}  }
void Square::drawLine( double x1, double y1, double x2, double y2)
   _dp->drawLine( x1, y1, x2, y2);
}
```

C++ Example cont'd

```
class RegularSquare : public Square {
  public:
    RegularSquare (Drawing *dp, double x1, double
      y1, double x2, double y2);
};
RegularSquare::RegularSquare (Drawing *dp,
    double x1, double y1, double x2, double y2) :
    Square(dp, x1, y1, x2, y2) { }

class FilledSquare : public Square {
  public:
    FilledSquare ( Drawing *dp, double x1, double
      y1, double x2, double y2);
};
FilledSquare::FilledSquare (Drawing *dp, double x1,
      double y1, double x2, double y2) :
      Square(dp, x1, y1, x2, y2) { }

void FilledSquare::draw () {
    Square::draw();
    _dp->fillArea( _x1, _y1, _x2, _y2);
}
```

```
class Drawing {
  public:
    virtual void drawLine (double x1,double y1,double x2,double y2)=0;
    virtual void fillArea (double x1, double y1, double x2, double y2)=0;
};

class V1Drawing : public Drawing {
   public:  void drawLine (double x1, double y1, double x2, double y2);
            void fillArea( double x1, double y1, double x2, double y2);
};
void V1Drawing::drawLine (double x1,double y1,double x2,double y2){
  DP1.draw_a_line( x1, y1, x2, y2); }

void V1Drawing::fillArea (double x1, double y1, double x2, double y2){
  DP1.fill_area( x1, y1, x2, y2); }

class V2Drawing : public Drawing {
   public:  void drawLine (double x1, double y1, double x2, double y2);
            void fillArea( double x1, double y1, double x2, double y2);
};
void V2Drawing::drawLine (double x1,double y1,double x2,double y2){
  DP2.drawline( x1, x2, y1, y2); }

void V2Drawing::fillArea (double x1, double y1, double x2, double y2){
  DP2.fillitup( x1, x2, y1, y2); }
```

C++ Example cont'd

```
class DP1 {
  public:
    static void draw_a_line ( double x1, double y1, double x2, double y2);
    static void fill_area ( double x1, double y1, double x2, double y2);
};

class DP2 {
  public:
    static void drawline ( double x1, double x2, double y1, double y2);
    static void fillitup ( double x1, double x2, double y1, double y2);
};

// Implementations given to us
```

Memory Management in C++

- Sometimes each class representing the abstraction (the squares) will have its own implementation. In this case, it can be responsible for deleting its implementation.
- However, implementations can be shared across objects representing the abstractions. In this case, some higher level of control is needed.

Example Implementation in Java

```
class Client {

  public static void main (String argv[]) {
    Square s1;
    Square s2;
    Drawing dp;

    dp= new V1Drawing();
    s1= new RegularSquare( dp,1,1,2,2);

    dp= new V2Drawing ();
    s2= new FilledSquare( dp,2,2,4,4);

    s1.draw();
    s2.draw();
  }
}
```

NOTE: imports not included

```
abstract class Square {
  double _x1;
  double _y1;
  double _x2;
  double _y2;
  Drawing _dp;

  Square (Drawing dp, double x1, double y1, double x2,
    double y2) {
    _dp= dp;
    _x1= x1;
    _y1= y1;
    _x2= x2;
    _y2= y2;
  }
 public void draw () {
    drawLine( _x1, _y1, _x2, _y1);
    drawLine( _x2, _y1, _x2, _y2);
    drawLine( _x2, _y2, _x1, _y2);
    drawLine( _x1, _y2, _x1, _y1);
  }
 public void drawLine (double x1,double y1,double x2,double y2)
{
    _dp.drawLine( x1, y1, x2, y2);
}
```

Java Example cont'd

```
class RegularSquare extends Square {
  RegularSquare (Drawing dp, double x1, double y1,
      double x2, double y2) {
    super( dp, x1, y1, x2, y2);
  }
}

class FilledSquare extends Square {
  FilledSquare (Drawing dp, double x1, double y1,
    double x2, double y2) {
    super( dp, x1, y1, x2, y2);
  }

  public void draw () {
    super.draw();
    _dp.fillArea( _x1, _y1, _x2, _y2);
  }
}
```

```
abstract class Drawing {
  abstract void drawLine (double x1, double y1, double x2, double
      y2);
  abstract void fillArea (double x1, double y1, double x2, double
      y2);
}

class V1Drawing extends Drawing {
  void drawLine (double x1, double y1, double x2, double y2) {
    DP1.draw_a_line( x1, y1, x2, y2);
  }
  void fillArea (double x1, double y1, double x2, double y2) {
    DP1.fill_area( x1, y1, x2, y2);
  }
}

class V2Drawing extends Drawing {
  void drawLine (double x1, double y1, double x2, double y2) {
    DP2.drawline( x1, y1, x2, y2);
  }
  void fillArea (double x1, double y1, double x2, double y2) {
    DP2.fillitup( x1, y1, x2, y2);
  }
}
```

Java Example cont'd

```
class DP1 {
  static void draw_a_line ( double x1, double y1, double x2, double y2)
      {
    // draw_a_line implementation
  }
  static void fill_area ( double x1, double y1, double x2, double y2) {
    // fill up between point
  }
}

class DP2 {
  static void drawline ( double x1, double y1, double x2, double y2) {
    // draw_a_line implementation
  }
  static void fillitup ( double x1, double y1, double x2, double y2) {
    // fill up between points
  }
}
```

Defer Decisions Until Run-time

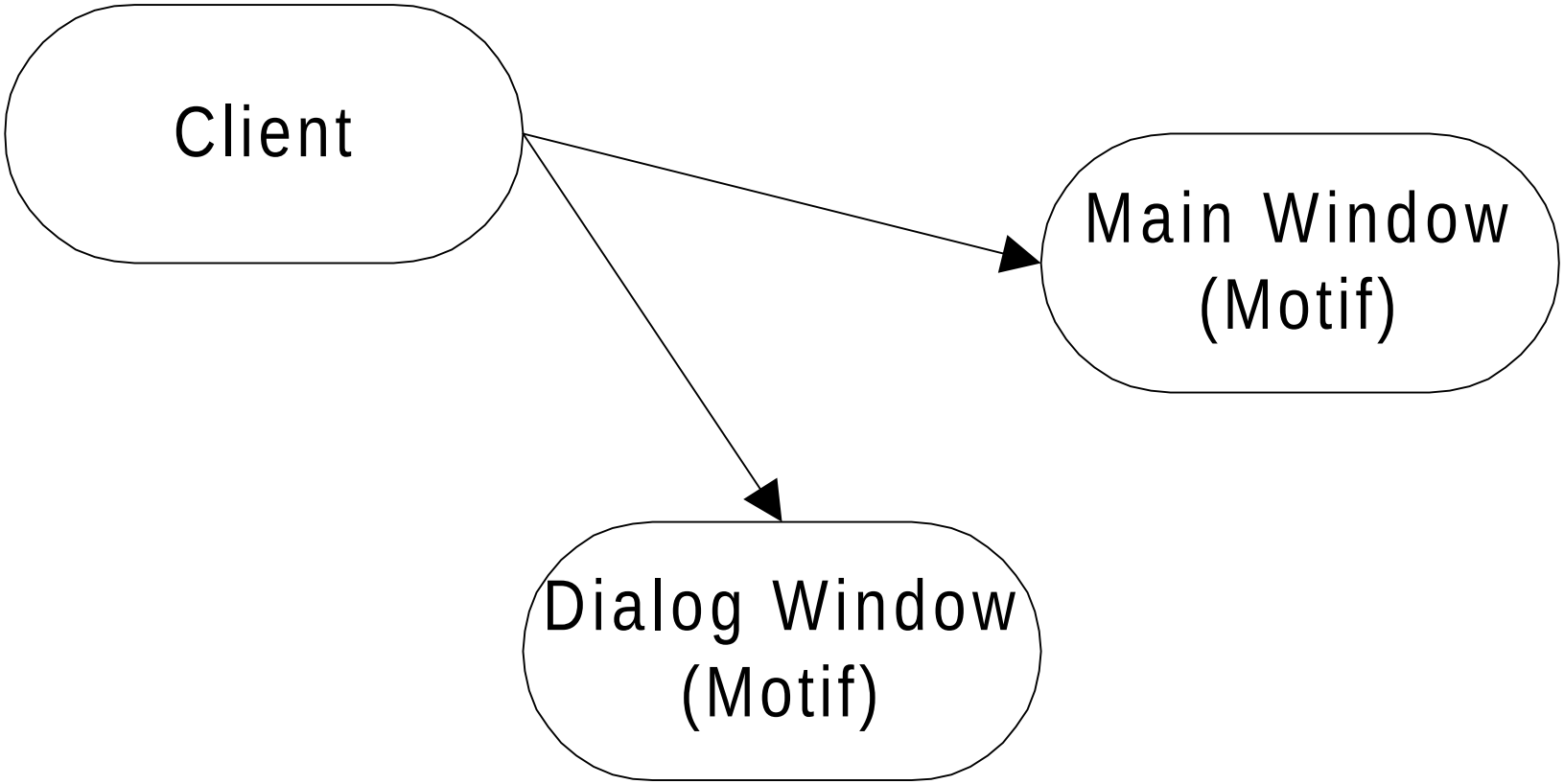
- Allows for adding new classes later
- Has minimal performance hit
- Increases flexibility

The Abstract Factory Pattern

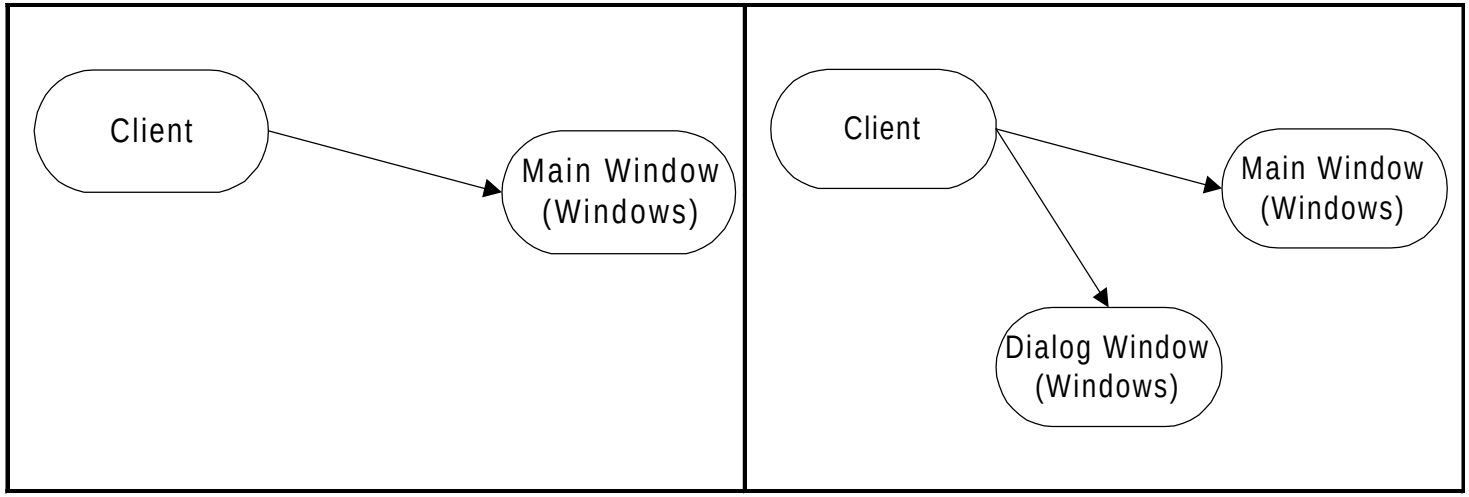
- Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes. [4]

- [4] Design Patterns, Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides

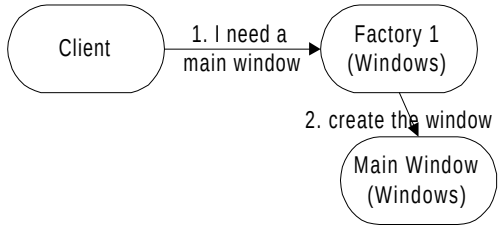
An Example of Where It Applies



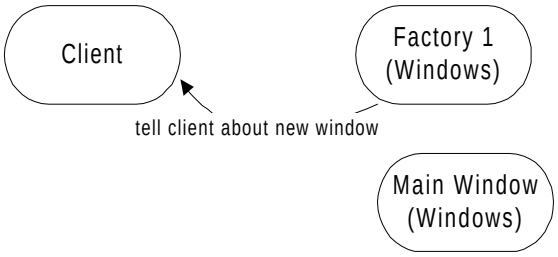
An Example of Where It Applies (cont'd)



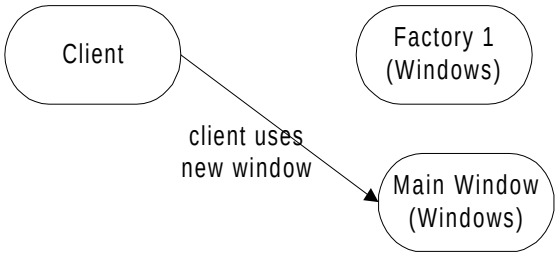
The Factory Object In Action



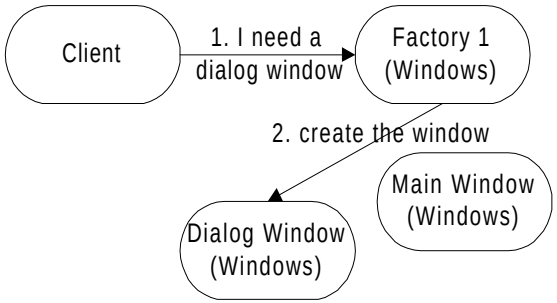
Step 1: Client asks for and creation object (factory 1) makes a main window



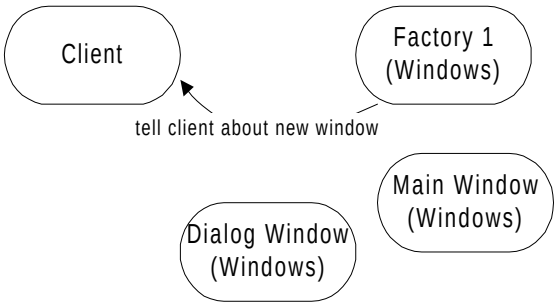
Step 2: Creation object tells client where this is



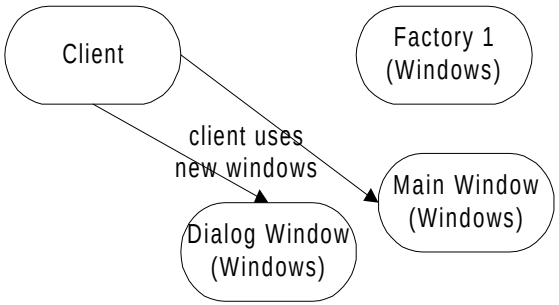
Step 3: Client uses the new object



Step 4: Client asks for and creation object makes a dialog window (making sure it is of the same type)

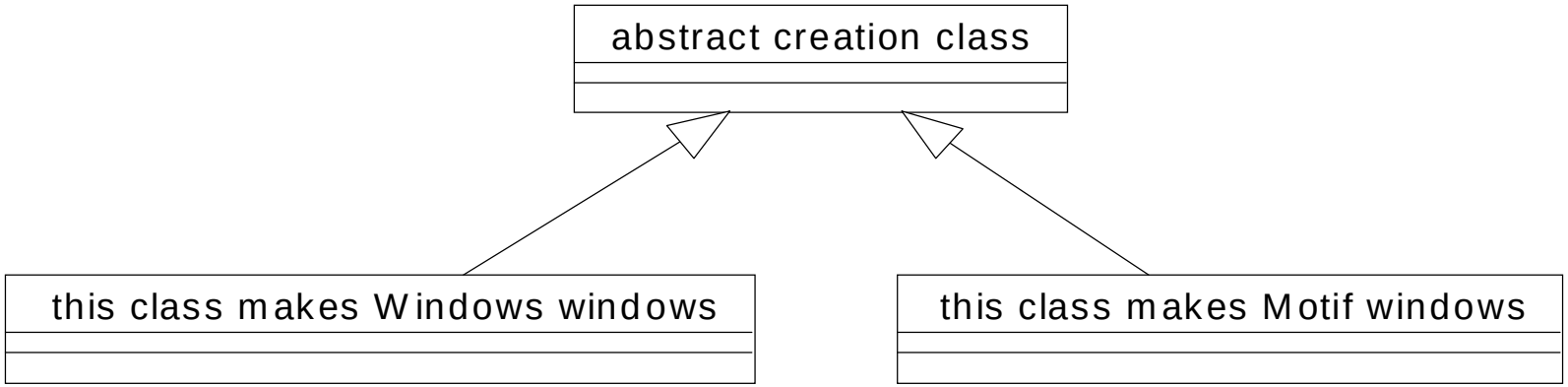


Step 5: Creation object tells client where this is

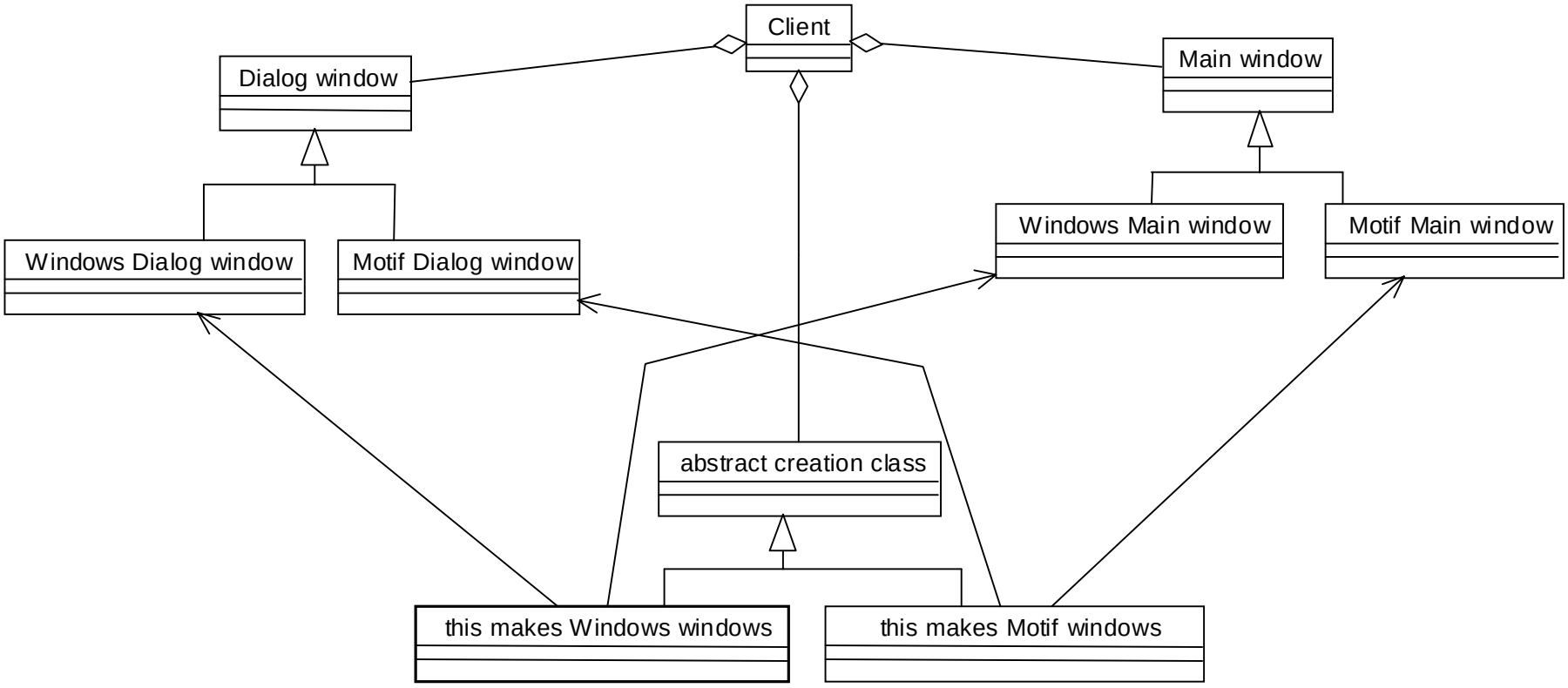


Step 6: Client uses newly created objects

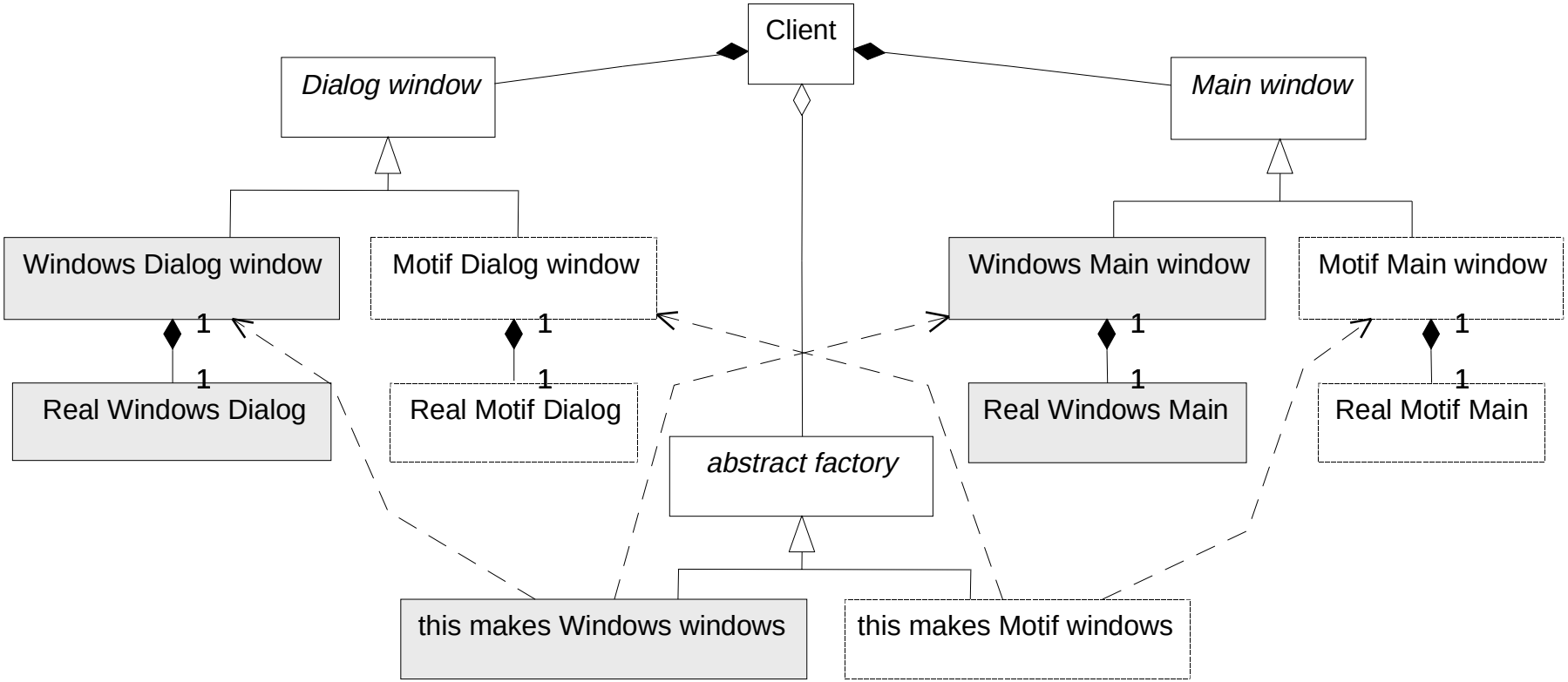
Classes of The Abstract Factory



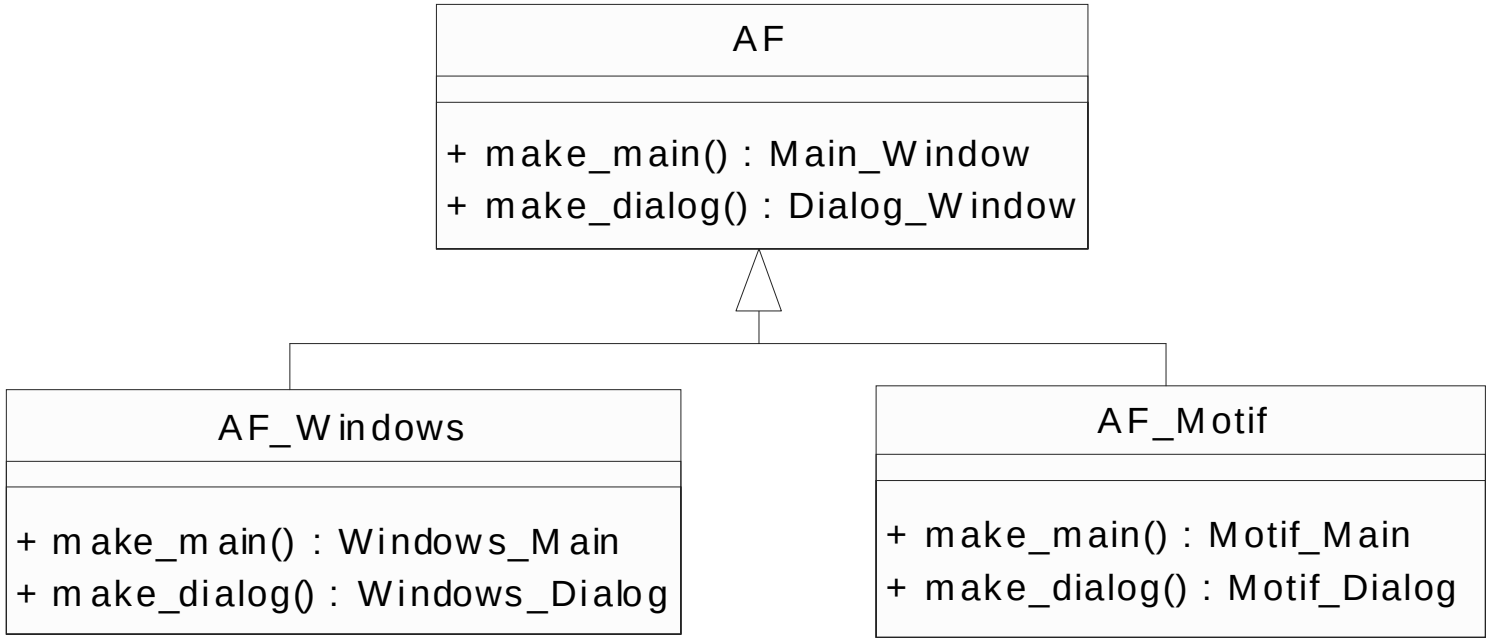
Classes of The Abstract Factory In Context of Their Use



A Little More Accurate



Class Diagram of Abstract Factory



C++ Example of The Abstract Factory

```
class AF_Windows : public AF;

Main_Window *
  AF_Windows::make_main() {
  return new Windows_Main;
}

Dialog_Window *
  AF_Windows::make_dialog() {
  return new Windows_Dialog;
}
```

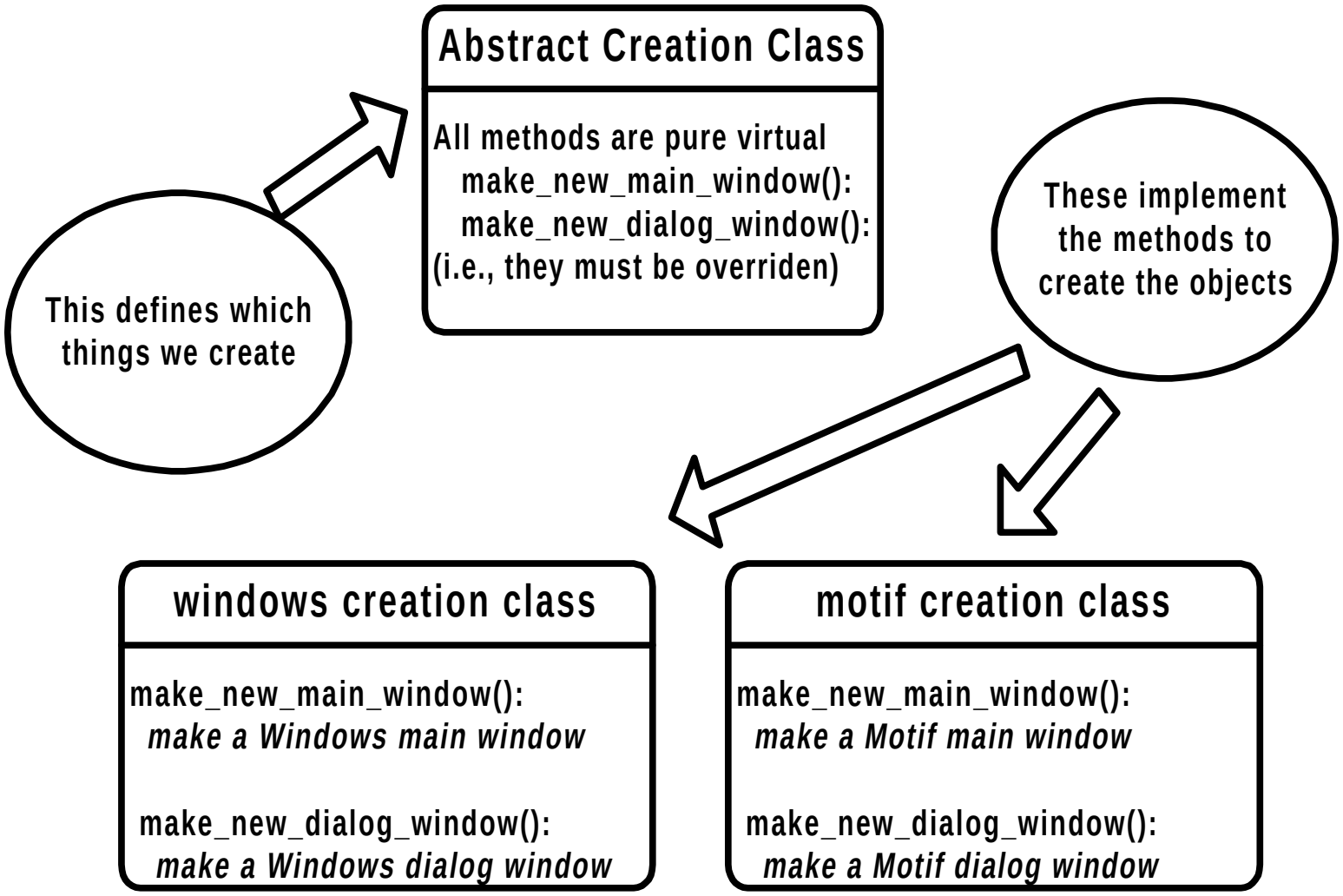
```
class AF_Motif : public AF;

Main_Window *
  AF_Motif::make_main() {
  return new Motif_Main;
}

Dialog_Window *
  AF_Motif::make_dialog() {
  return new Motif_Dialog;
}
```

Java Example of The Abstract Factory

```
class AF_Windows extends AF {
  Main_Window make_main() {
    return new Windows_Main();
  }

  Dialog_Window  make_dialog {
    return new Windows_Dialog();
  }
}
```

```
class AF_Motif extends AF {
  Main_Window make_main() {
    return new Motif_Main();
  }

  Dialog_Window make_dialog {
    return new Motif_Dialog();
  }
}
```

Abstract Factory Conceptually



Thinking in Patterns

We’ve completed the background:

- we’ve learned four patterns
- we’ve got a solution (albeit bad) without them
- we know there is a better solution

Re-Thinking Our Solution

- Our initial design focused on classes
- Our solution had several problems
- We knew of several patterns that were involved
- We used the patterns to solve local problems that existed in our class-based design

Design According To Christopher Alexander

- Design is often thought of as being a process of synthesis
- The best designs cannot be made this way
- The best designs are a process of differentiation

Excerpt from The Timeless Way of Building by Christopher Alexander (italics his).

Differentiating Space

- More from Alexander

- *… every individual act of building is a process in which space gets differentiated. It is not a process of addition, in which pre-formed parts are combined to create a whole: but a process of unfolding, like the evolution of an embryo, in which the whole precedes its parts, and actually gives birth to them, by splitting.*
- *Each part is slightly different, according to its position in the whole.*
- *Design is often thought of as a process of synthesis, a process of putting together things, a process of combination.*

Differentiating Space - Cont'd

- *According to this view, a whole is created by putting together parts. The parts come first: and the form of the whole comes second.*
- *But it is impossible to form anything which has the character of nature by adding preformed parts.*
- *When parts are modular and made before the whole, by definition then, they are identical, and it is impossible for every part to be unique, according to its position in the whole.*
- *It is only possible to make a place which is alive by a process in which each part is modified by its position in the whole.*

Differentiating Space - Cont'd

- *In short, each part is given its specific form by its existence in the context of the larger whole.*
- *This is a differentiating process.*
- *It views design as a sequence of acts of complexification; structure is injected into the whole by operating on the whole and crinkling it, not by adding little parts to one another. In the process of differentiation, the whole gives birth to its parts: the parts appear as folds in a cloth of three dimensional space which is gradually crinkled. The form of the whole, and the parts, come into being simultaneously.*
- *The image of the differentiating process is the growth of an embryo.*

Differentiating Space - Cont'd

- The unfolding of a design in the mind of its creator, under the influence of language, is just the same.
- Each pattern is an operator which differentiates space: that is, it creates distinctions where no distinction was before.
- And in the language the operations are arranged in sequence: so that, as they are done, one after another, gradually a complete thing is born, general in the sense that it shared its patterns with other comparable things; specific in the sense that it is unique, according to its circumstances.
- The language is a sequence of these operators, in which each one further differentiates the image which is the product of the previous differentiations.

Using Thinking in Patterns to Define Our Application Architecture

- Based on Christopher Alexander's philosophy of using a pattern [as] an operator which differentiates space: [creating] distinctions where no distinction was before.
- Start at highest conceptual level.
- Decide which patterns create the context for the other patterns.
- Apply patterns in this order, getting more detailed as you go.

Which Patterns Create Context for Others?

- **Object Adapter:** local - doesn’t create context for anything
- **Facade:** local - doesn’t create context for anything
- **Bridge:** creates context for both object adapter and facade
- **Abstract Factory:** involved in creating everything, but needs to see what to create before it can be implemented

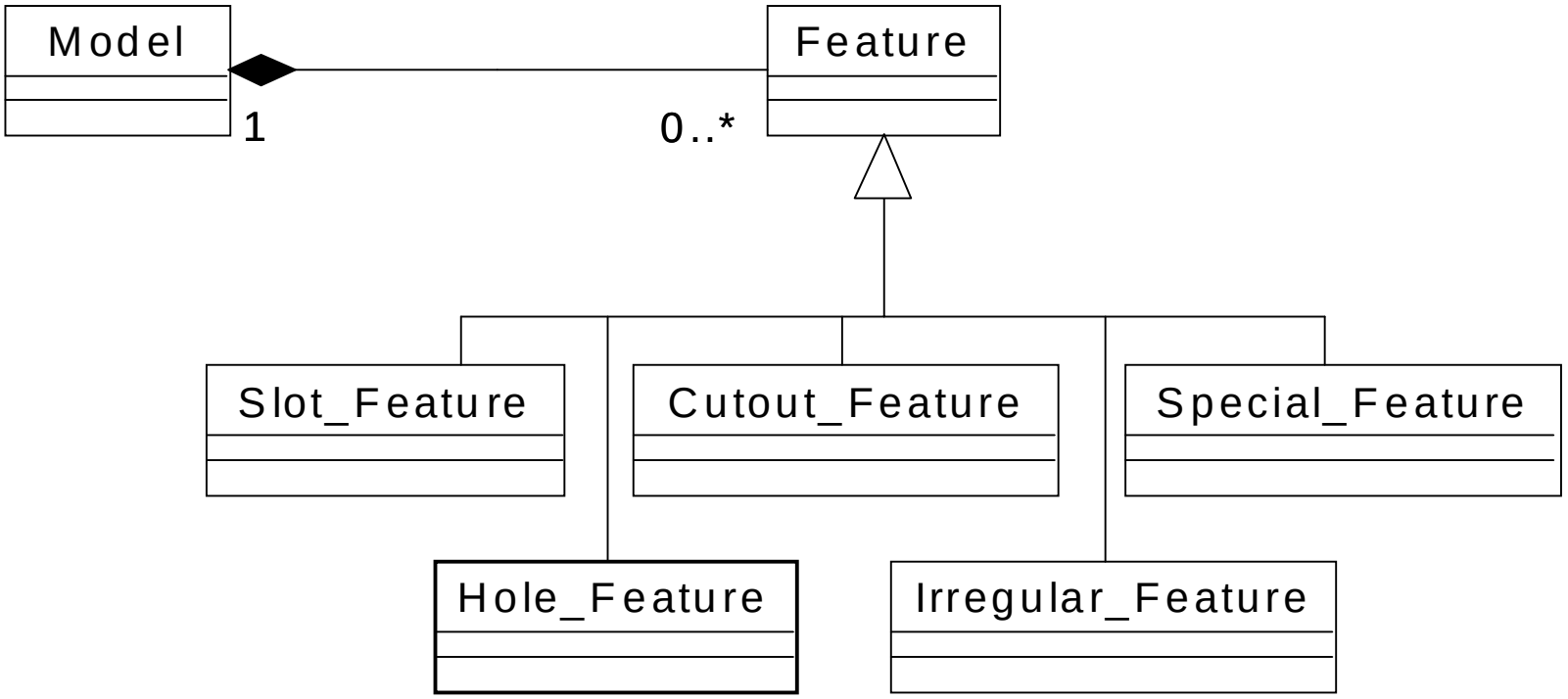
How To Determine Context

- Since patterns are interrelated, we can't simply ask which pattern, if applied, would affect another one?
- One pattern typically will define the context of the other.
- Try it both ways and see which makes more sense.
- Sometimes, it is not clear and the patterns are coupled.

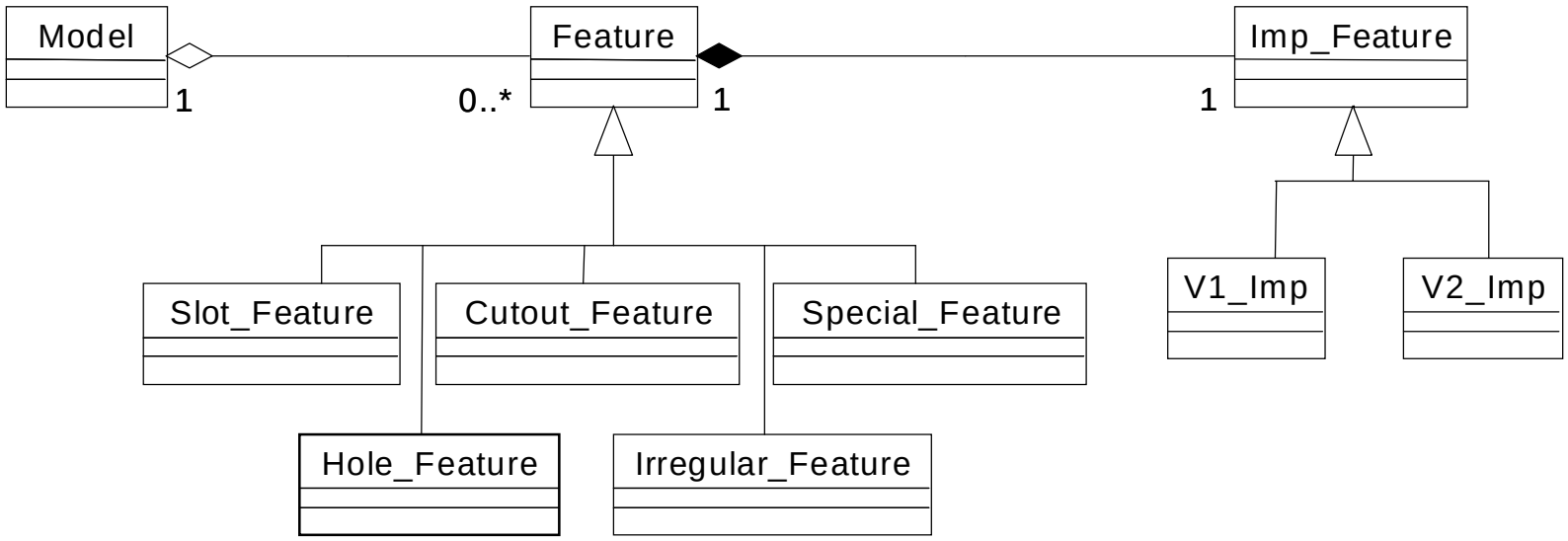
Decide What to do Before Deciding How to Instantiate Your Objects

- We should trust that after we've decided what our classes should be that we can instantiate them properly
- This simplifies the thinking process
- De-couples action from instantiation

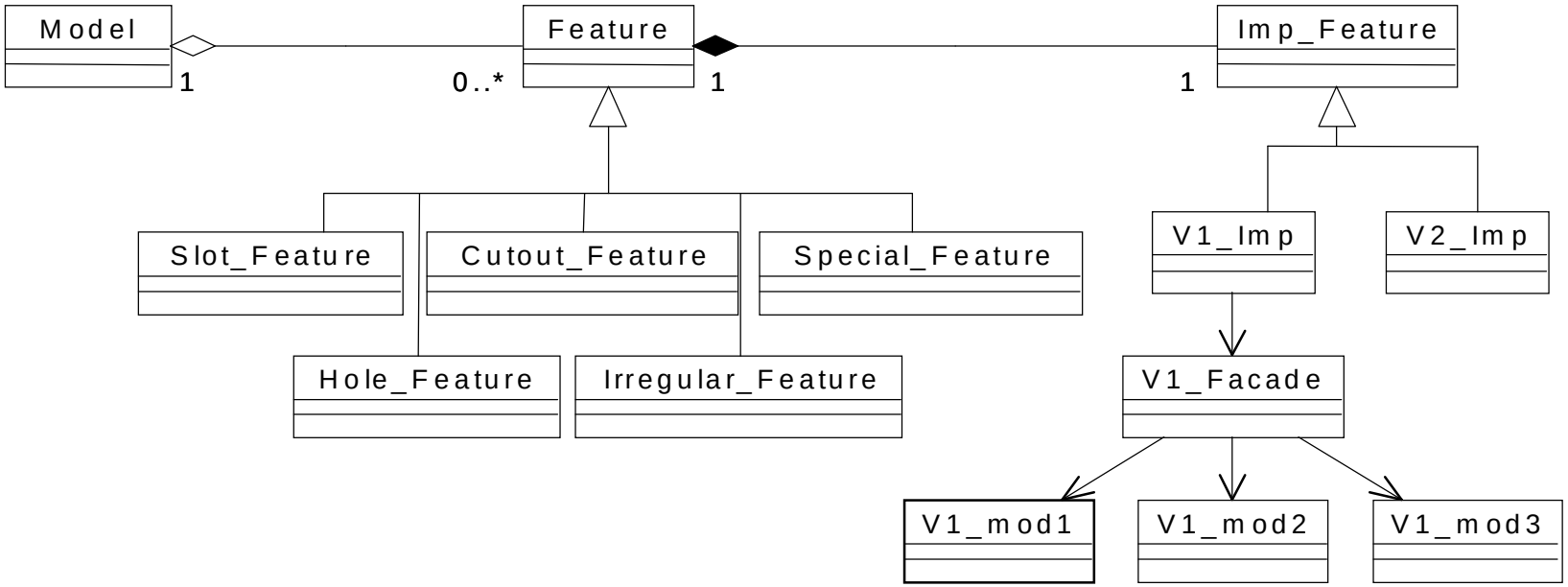
Start at the Highest Level



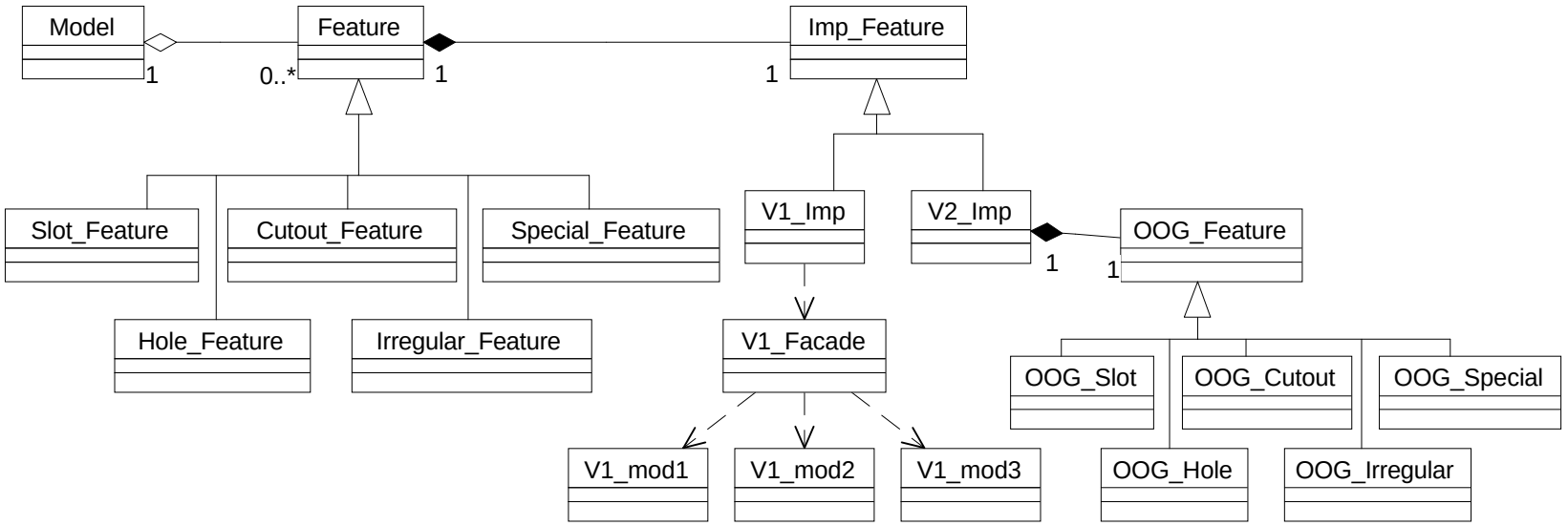
Apply Bridge Distinction



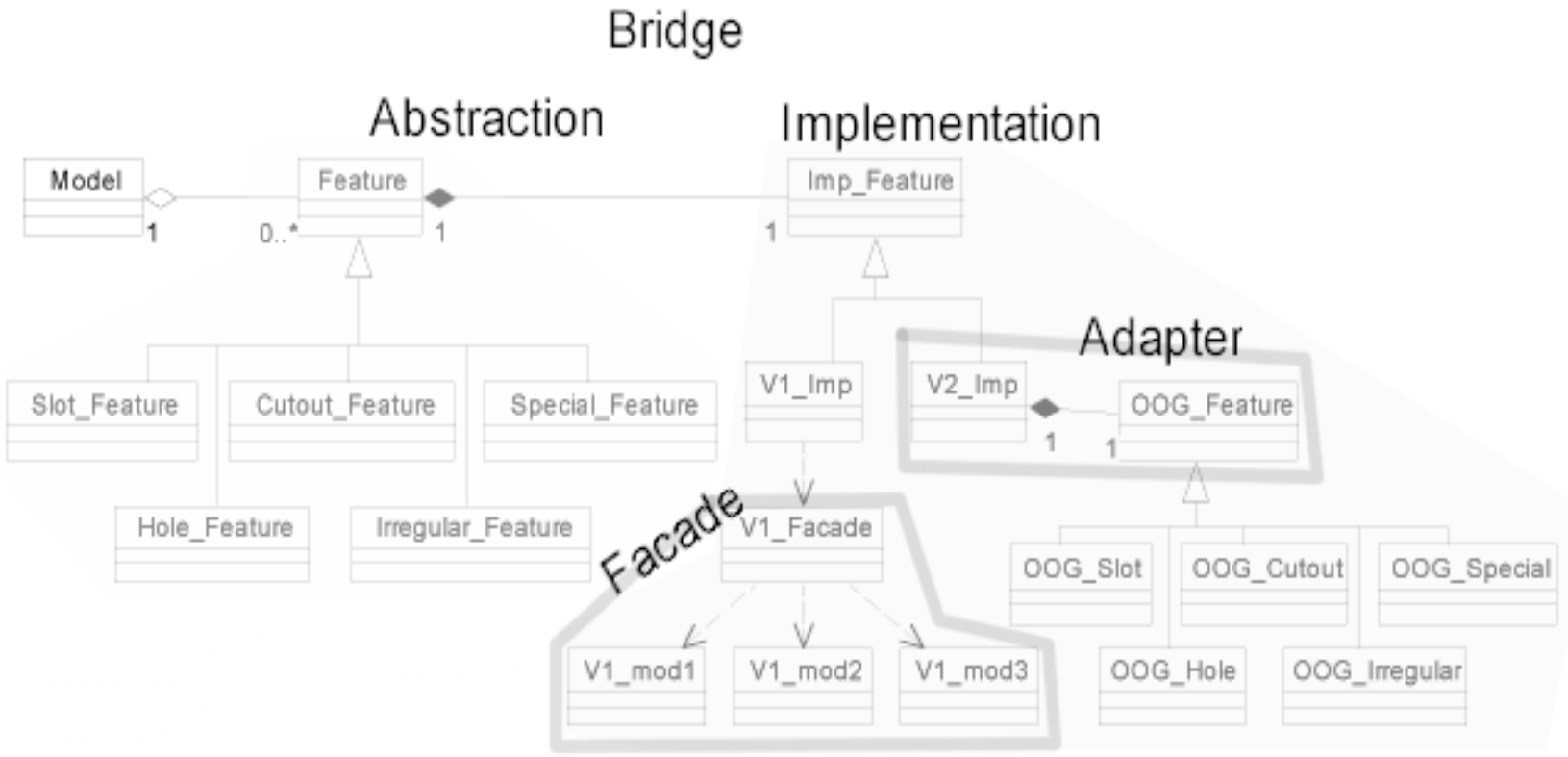
Apply Facade Distinction



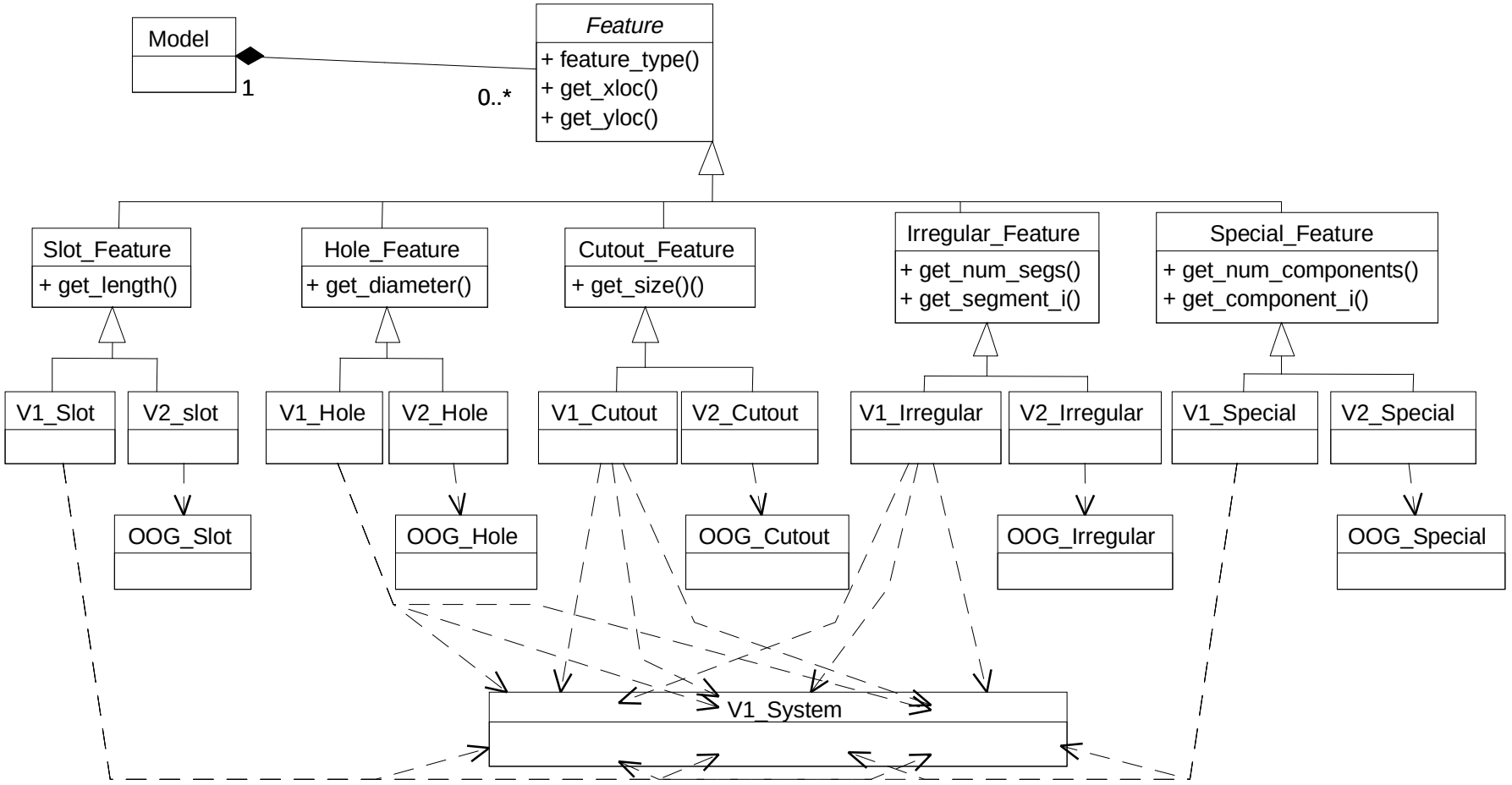
Apply Adapter Distinction



Solution with Patterns



The Original Solution



How To Do Pattern Oriented Design

- Study your problem domain
- Discover what patterns exist there
- See if the problem you are trying to solve can be discussed in terms of these patterns
- If so, you are ready to proceed to the next step, if not, continue to look for patterns, or describe the relationships present
- Find that pattern that defines the primary context
- Apply that, and continue this process

How To Learn More

- Net Objectives offers training in Pattern Oriented Design both for those new to object-oriented technology and those experienced in OO.
- Training also available in OOA, Java, C++ and CORBA.
- Mentoring and consulting can be provided.
- On-site courses available.

- www.netobjectives.com
- alshall@netobjectives.com -- 425-260-8754