How Do Experts Design?

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 21 — 03/29/2011

Goals of the Lecture

- Cover the material in Chapters 12 & 13 of our textbook
  - How do experts design?
  - Solving the CAD/CAM Problem

- But first…

Objective-C 2.0 Class Extensions

- Interesting encapsulation mechanism
  - Allows the header file of a class to include only details about the class's public interface
  - Allows everything that should be private to remain unseen by the developers who make use of your class

Example: Set-Up

```
@interface Roster : NSObject {
@private
  NSMutableArray *roster;
}
- (void) appendPlayer: (Player *)p;
- (Player *) playerAtIndex:(NSUInteger) index;
@private
- (void) updatePlayerAtIndex:...
@end
```

Discussion (I)

- The Roster class using Objective-C 1.x techniques tells developers that it is using an array internally to store the players of a team
- The array is declared private, but Objective-C is a dynamic language and its possible to get around these restrictions if you really want to
- As a result, some developers could violate the abstraction of "Roster of Players" and instead come to depend on the implementation "Array of Players"

Example: Properties Don't Help

```
@interface Roster : NSObject {
}
@property (retain) NSMutableArray *roster;
- (void) appendPlayer: (Player *)p;
- (Player *) playerAtIndex:(NSUInteger) index;
@private
- (void) updatePlayerAtIndex:...
@end
```

Discussion (II)

- Not declaring the instance variable and just declaring a property won't help
  - in fact, it's worse
- In the previous example, it was a bit more work to access the array… with a property we are making it easy to access the array
  - Indeed, properties should only be used to declare attributes that are part of the public interface
    - data that you WANT to share with clients

Discussion (III)

- However, it WOULD be nice to have a property defined for the internal array
  - So, in our own internal code we can write things like
    - self.roster = …
  - and have getters and setters invoked automatically for us without making those properties publicly available
- In addition, it would be nice to declare private helper methods that can be used anywhere in the implementation section of a class but, again, hide these from external users of the class

Class Extensions

- Objective-C 2.0 class extensions allow you to do all of this
  - It builds off the category mechanism I discussed in Lecture 20
    - categories: ability to add methods to existing classes

- A class extension occurs at the top of a class's .m file just after the inclusion of the class's .h file
  - @interface Roster ()
    - <internal properties and method defs>
  - @end

Note: category name is left blank

Example: Public Interface becomes…

```
@interface Roster : NSObject {
}
- (void) appendPlayer: (Player *)p;
- (Player *) playerAtIndex:(NSUInteger) index;
@end
```

Example: Class extension is…

```
@interface Roster ()

@property (retain) NSMutableArray *roster;

- (void) updatePlayerAtIndex:...

@end


@implementation Roster

@end
```

No need to declare anything private this is all in the .m file and available only to the class implementation below

From OO A&D perspective…

- Interesting encapsulation mechanism that
  - allows public interface to be expressed without clutter
  - allows developers to specify internal interface cleanly
    - including specifying additional instance variables that are hidden from external users
- If you distribute just a .h file and a framework, external developers are much less likely to become dependent on the implementation of your class

Back to the Book: Overview

- Chapter 12 talks about applying the lessons that Christopher Alexander developed for designing cities—architectural design patterns—to software engineering and to the design of software systems
  - there is not a one-to-one mapping
  - but there are important lessons to learn
- It was NOT enough for Alexander to specify individual design patterns… it was about using design patterns to transform the nature of the design process

Important for all designers

- Alexander's approach has had such impact on the software design community because his work describes an approach to design that is valid to ANY designer
- There are several aspects to his work that are not intuitive; indeed, it is our intuitive notion of design that can often lead us into trouble by oversimplifying
  - Analysis: What's the Problem?
  - Design: What's the Solution?
    - Both of these are oversimplifications of really complex tasks

Intuitive Notion of Design

- Build by fitting things together: "build from pieces"
  - Indeed, this is the whole point of functional decomposition
    - decompose the problem into small pieces and then build up from there
  - And OO follows this with classes and objects
- But Alexander indicates that this is NOT a good way to design

The Set-Up

- Alexander says:
  - Design is often thought of as a process of synthesis, a process of putting together things, a process of combination. According to this view, a whole is created by putting together parts. The parts come first: and the form of the whole comes second.

The Problem

- Alexander continues
  - 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. Even more important, it simply is not possible for any combination of modular parts to contain the number of patterns which must be present simultaneously in a place which is alive.
- "Cookie cutter" designs do not produce high quality results
  - Think european city center vs. southern california suburb

Possible Confusions (I)

- In software engineering, we talk about producing modular, generic classes that are reusable across multiple contexts
  - Now Alexander is seemingly telling us not to do this!
    - but remember: he was talking about architecture…
- Alexander's "modular" refers to identical parts that can be snapped together to produce a structure that was designed independent of its final location
  - Software Engineering's "modular" refers to separation of concerns: "I deal with persistence in this module."

Possible Confusions (II)

- Alexander talks about a "place which is alive"
  - What does that mean for software systems?
- Alexander says
  - "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."
- The context of a part influences the design and characteristics of that part…

Possible Confusions (III)

- Our authors indicate that the counterpart to
  - "a place which is alive" in software design is
    - robust and flexible software systems
    - systems whose parts have been tweaked by context to reach a state in which the system is
      - extensible, maintainable, flexible, etc.
      - i.e. resilient or **lifelike**

The Goal

- So, interpreting what we've seen so far, the goal of design becomes
  - Design pieces—classes and objects—within the context in which they must live in order to create robust and flexible systems
- How?
  - Alexander's answer is a bit mystifying at first
    - ☺

Alexander's Answer: "Complexification"

- 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 form of the whole, and its parts, come into being simultaneously. The image of the differentiating process is the growth of an embryo.

Translation

- Design is a process that starts by looking at a problem in its simplest term, giving a unified whole
  - it is refined by making decisions, adding information (and thus, complexity), making distinctions between elements in the design where none existed before
  - but the distinctions are made within a larger context, guided by the whole, and the elements added to the domain by the decision can be guided by patterns

Example

- Think about a planning (designing) an academic conference
  - Multi-day event held at a hotel with sessions and a reception organized by a set of people
    - sessions: workshops, keynotes, papers, posters, etc.
    - people: conference chair, program chair, program committee, conference committee
      - conference committee: publicity chair, proceedings chair, local events chair, etc.

Alexander: The Role of Patterns

- Each pattern is an operator that differentiates space: that is, it creates distinctions where no distinction was before.
- … the operations are arranged in sequence: so that, as they are done, one after the other, 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
- … each [pattern] further differentiates the [whole], which is the product of previous differentiations.

Design for Everyone

- The interesting thing about this stance is that design is something that can be learned by anyone
  - A design that follows well-established patterns will produce good, solid results; it should not be surprising that quality solutions for similar problems appear very much alike
    - For instance, following Model-View-Controller brings benefits to even the most novice of designers…
- Creativity comes in understanding how to adapt the patterns to the context you find yourself in

The Steps

- Within the context of a design
  - Identify patterns that can add information to the design
    - ones that define useful relationships between entities of the design (or suggest entities and relationships not currently present that would benefit the design)
  - Add them to the design, thus updating the context
  - Repeat, until no more entities and relationships are needed to solve the problem

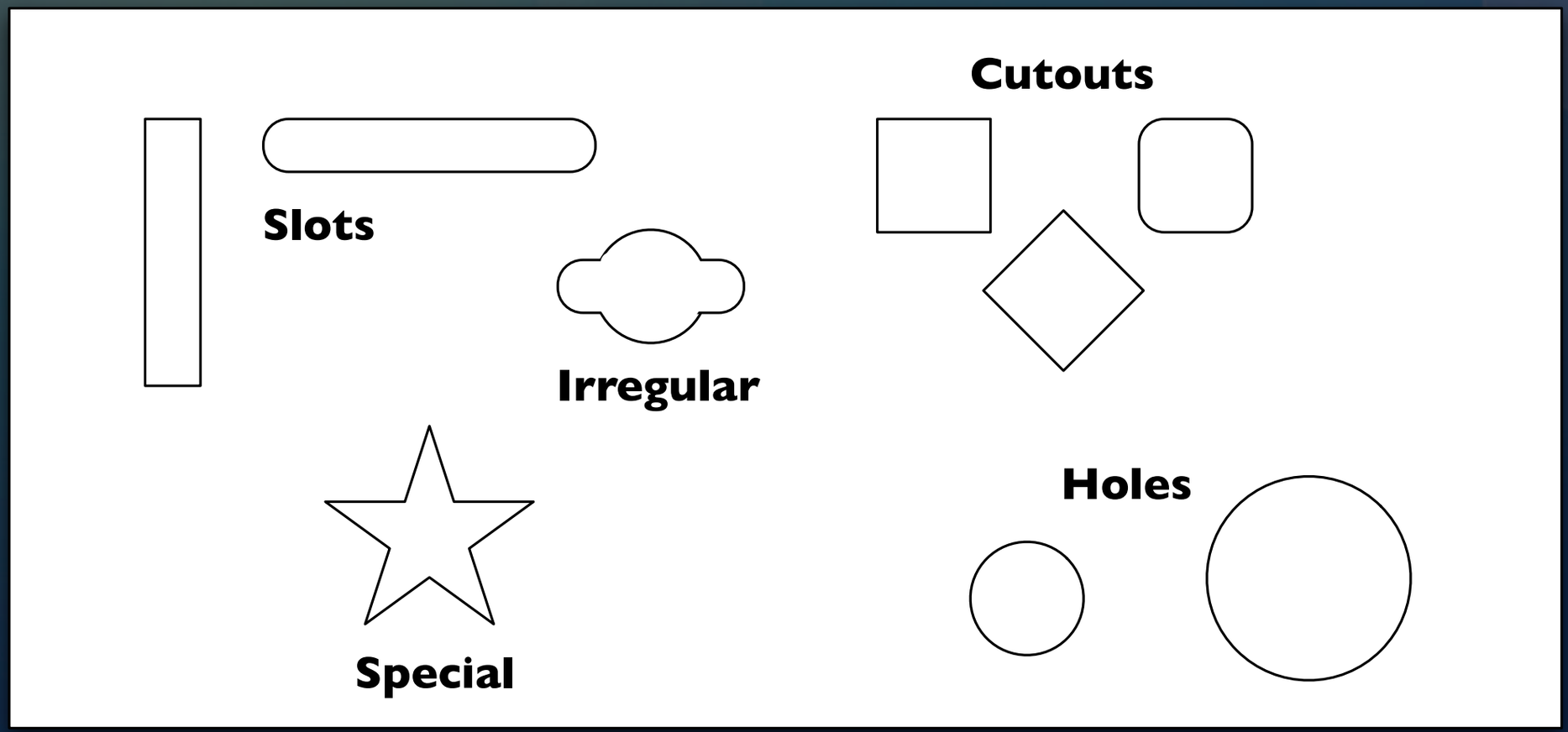
Example

- Design a system that aids a geologist in assigning ages to rock samples collected from the field
  - Context Pattern: Desktop Application
    - Leads to: Model-View-Controller
      - Model Leads to: Database of Rock Samples
      - View Leads to: Collection Browser and Operations
      - Controller: Set of "glue" objects that invoke operations on selected samples, updates database, displays results

Limitations

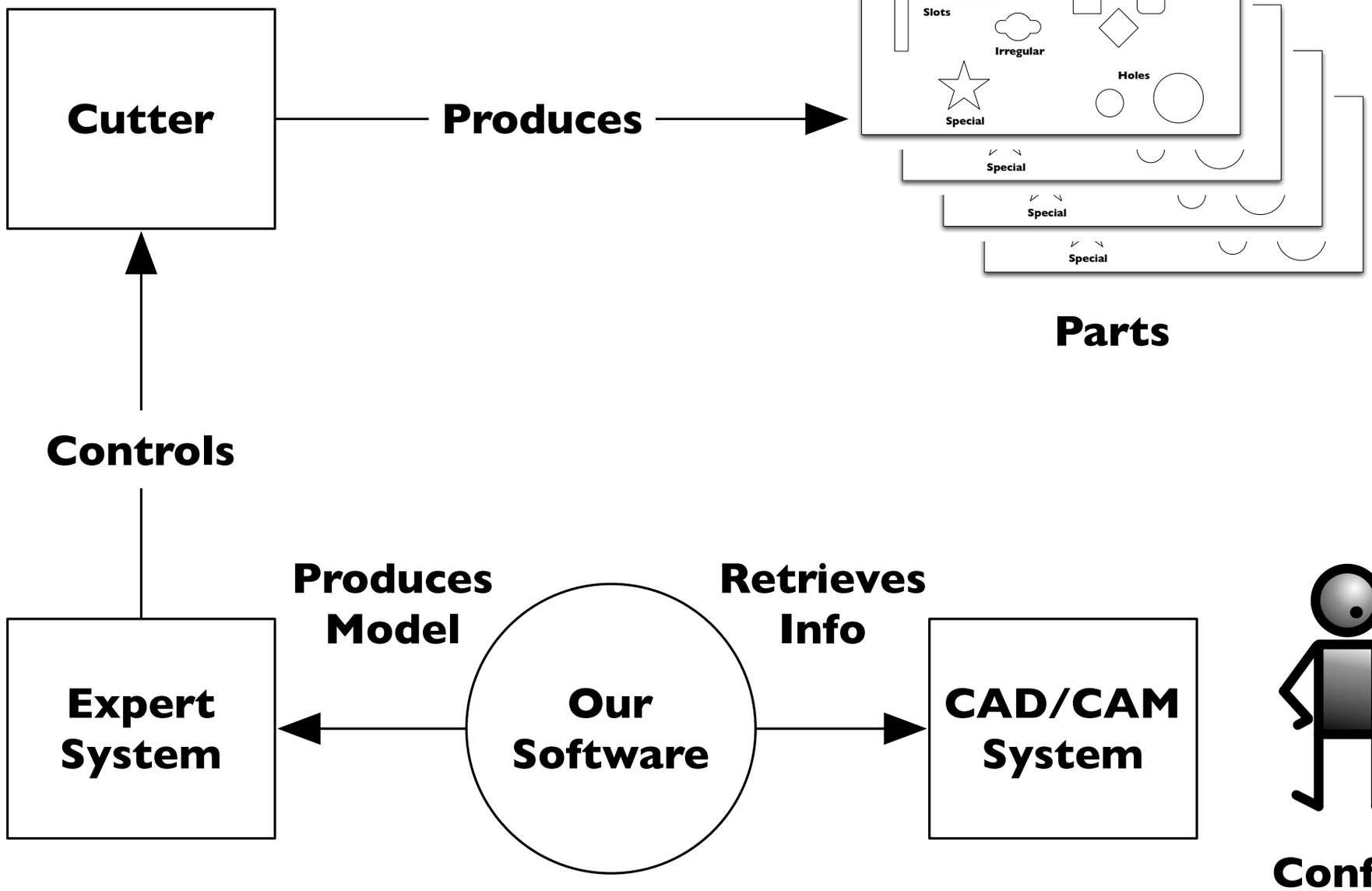
- The authors of our book caution that Alexander's approach does not directly translate to software design
  - Well-defined patterns do not exist for all problem domains
  - Context to Pattern to New Context to Pattern chains may not be that deep
  - How to customize a pattern to a particular context may be non-obvious
- The principles behind Alexander's techniques ARE important

Review of CAD/CAM Problem

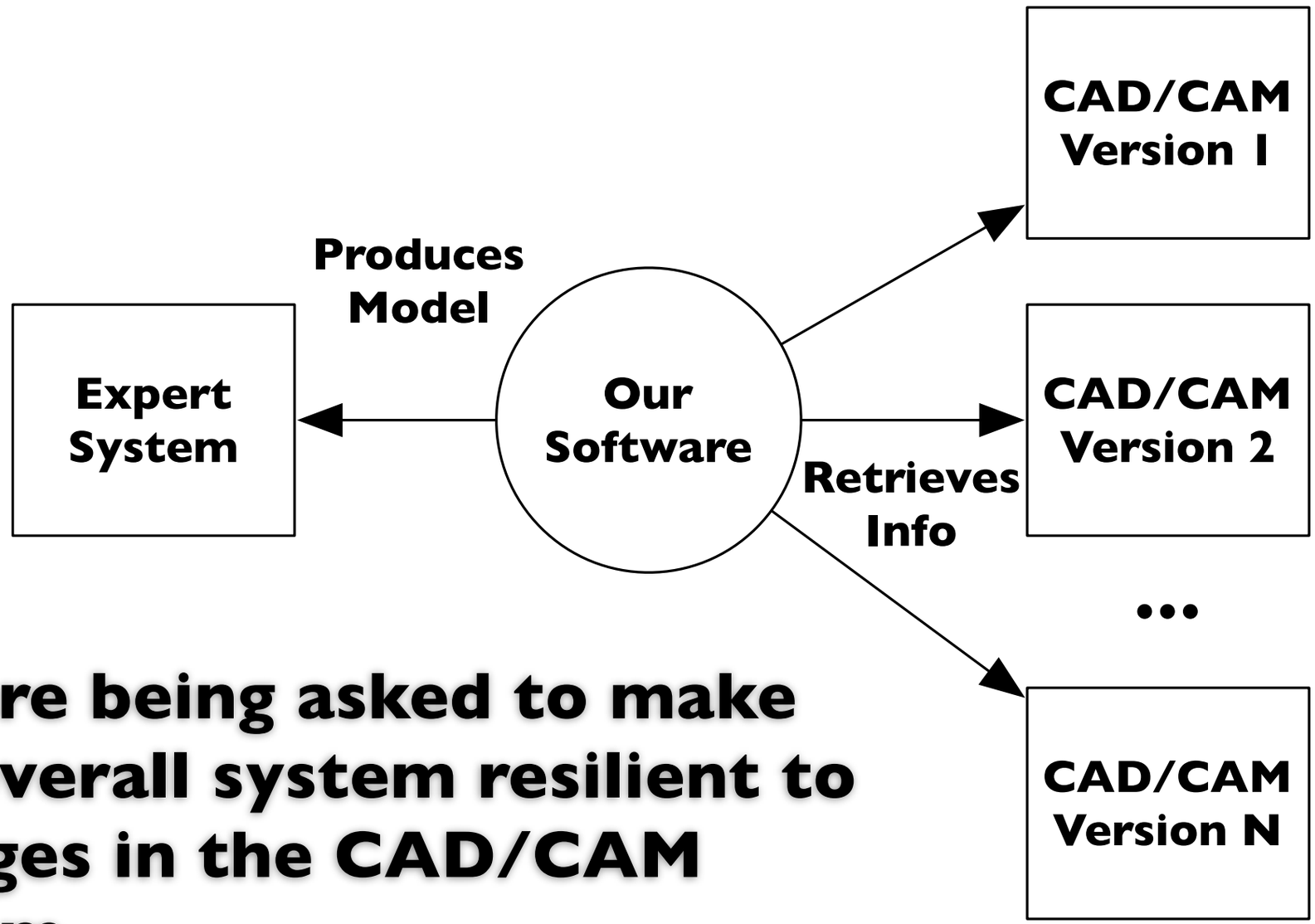


Design software that translates CAD designs that use the parts above into instructions for a machine that punches the actual part out of sheet metal

System Overview



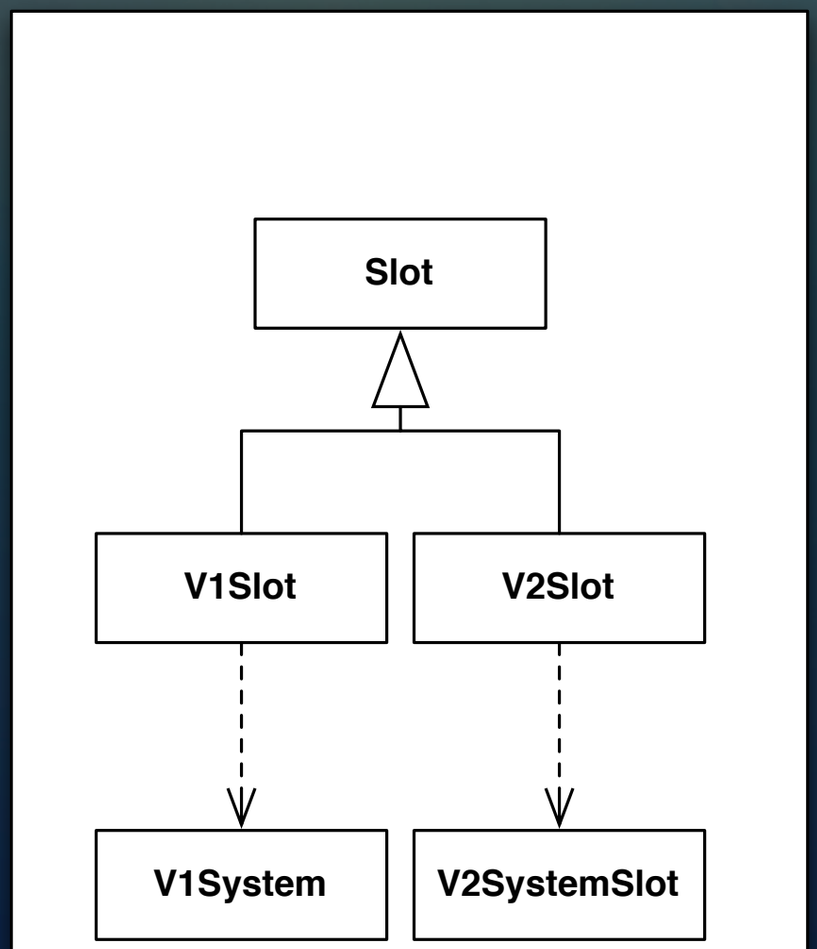
Here's the Problem



We are being asked to make the overall system resilient to changes in the CAD/CAM system

Example of encapsulation via software architecture…

The Original Solution (I)

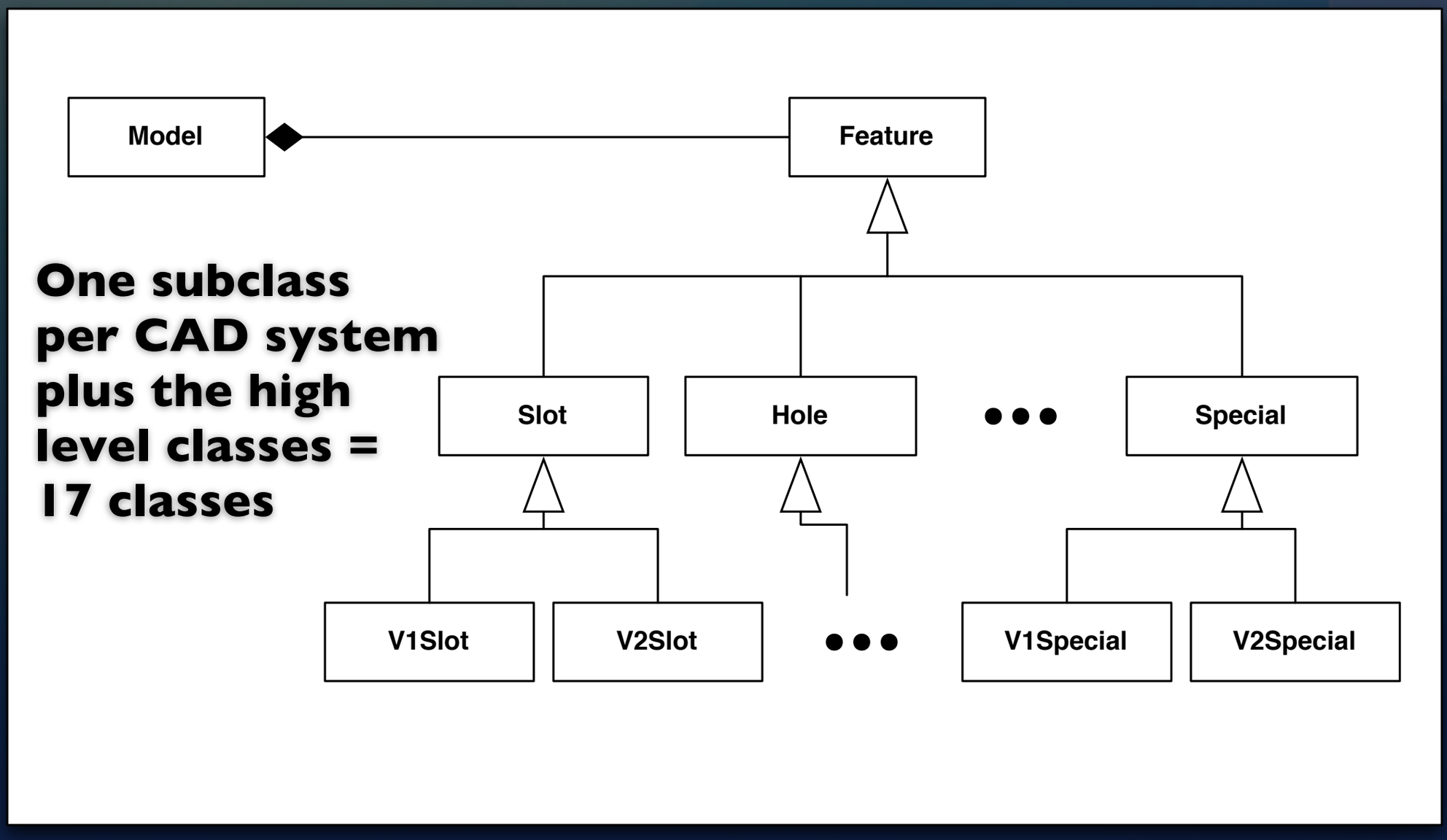


For each Feature class, the version 1 variation will have attributes that link to the version 1 model id and the feature id; it will then call the V1 library routines directly

The version 2 variation will simply wrap the Feature class that comes from the CAD system

The arrow with dashed line means "uses"

The Original Solution (II)



WWAD? (What Would Alexander Do?)

- To develop a better solution to this problem, let's think in terms of patterns
  - What are the essential concepts of the problem domain and what relationships exist between them?
    - This can help us identify patterns that can be applied
- This doesn't always work (design is hard) but patterns can often get you started moving in a particular direction

Thinking in Patterns (big picture view)

- Step 1: Identify the Patterns
- Step 2: Analyze and apply the patterns
  - 2a. Order the patterns by context creation
  - 2b. Select pattern and expand design
  - 2c. Identify additional patterns, add them to the set
  - 2d. Repeat
- Step 3: Add detail

Step 1: Identify the Patterns

- For the CAD/CAM Domain, the authors identified
  - Abstract Factory: Create parts for a particular CAD system
  - Adapter: Adapt new CAD systems to the target interface
  - Bridge: Implement the abstractions of the domain by "bridging" to a particular CAD system
  - Facade: keep the complexities of the CAD system hidden from the expert system

Step 2a: Which pattern provides context for the others?

- Look through all possible parings of the identified patterns
  - Does x provide context for y?
  - Does abstract factory provide a context for bridge?
    - Look back at our Pizza shop example for inspiration
  - To help with these decisions look at the patterns conceptually…

Step 2a (II)

- Abstract factory creates sets of related objects
- Adapter adapts existing class A to the interface needed by a client class B
- Bridge allows for different implementations to be used by a set of related client objects
- Facade simplifies an existing system A for a client class B

Step 2a (III)

- Abstract factory's context is the structure of the objects its creating
  - Pizza is made of dough, sauce, toppings, etc.
- It does not provide context for other patterns
  - This is true of most "creational patterns"
  - So, scratch it off the list
- This leaves
  - Adapter ↔ Bridge; Bridge ↔ Facade; Facade ↔ Adapter

Step 2a (IV)

- Bridge ↔ Adapter
  - Adapter will allow the expert system to access the OO interface of the new CAD system by making it conform to Feature and its subclasses
  - Bridge will ensure that Feature and its subclasses can access version 1 and 2 of the CAD system
    - Bridge provides context for Adapter

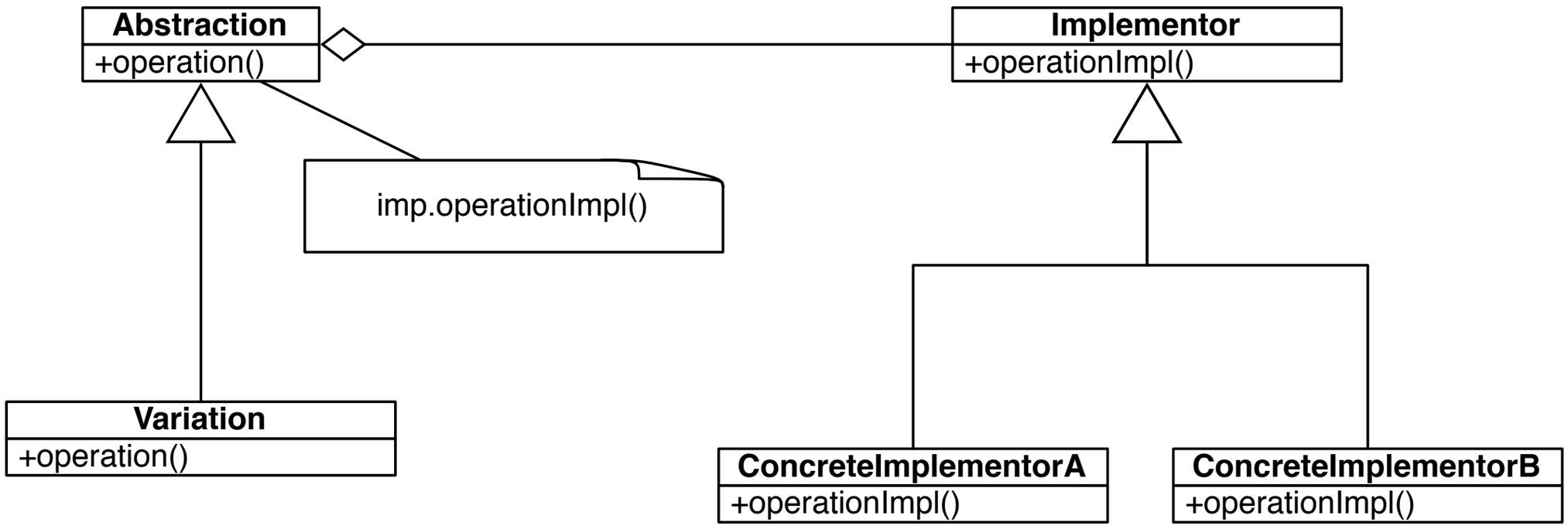
Step 2a (V)

- Bridge ↔ Facade
  - Facade will simplify the complex interface of the first CAD system
    - Bridge will ensure that Feature and its subclasses can access version 1 and 2 of the CAD system
      - which means making use of the Facade
- Bridge provides context for Facade (in this system)
  - Since Bridge "wins" twice, its the outermost pattern

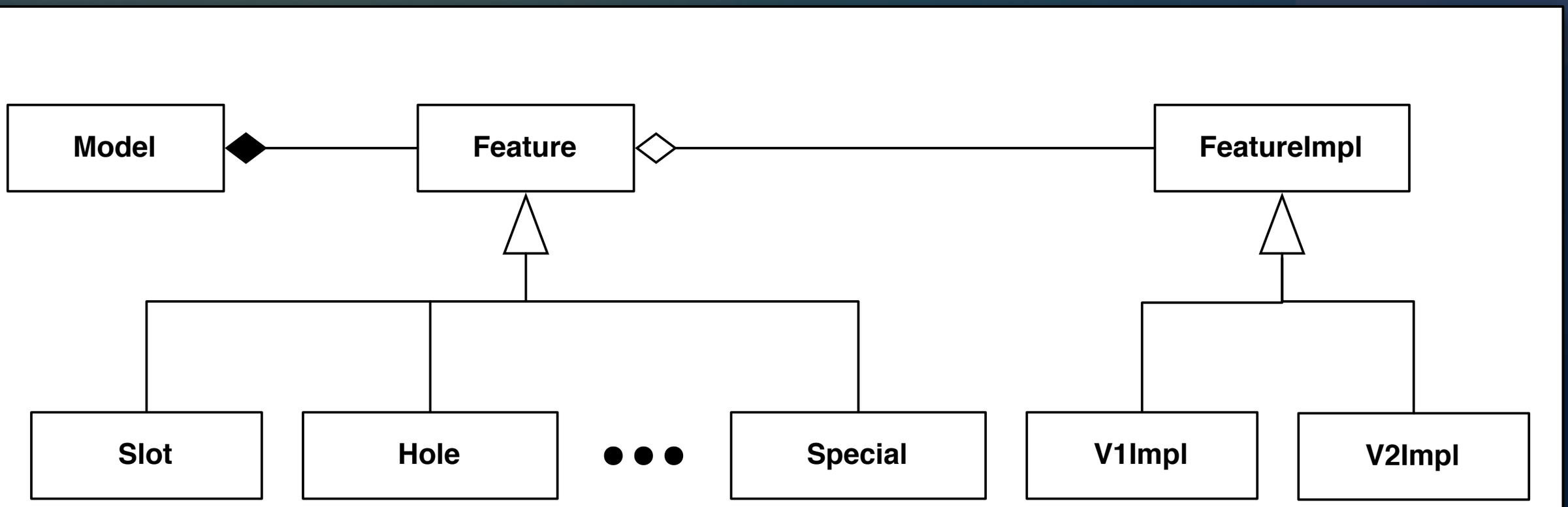
Step 2b: Select Pattern and Expand Design

- How does Bridge fit into the conceptual whole of the design?
  - What, exactly, provides a context for the Bridge pattern?
    - The elements of the problem domain!
      - Expert System uses Model
      - Model aggregates Features (abstractions)
      - Different CAD systems provide different types of features (implementations)
      - Bingo! The Bridge pattern

Bridge Structure Diagram



Bridge in Context

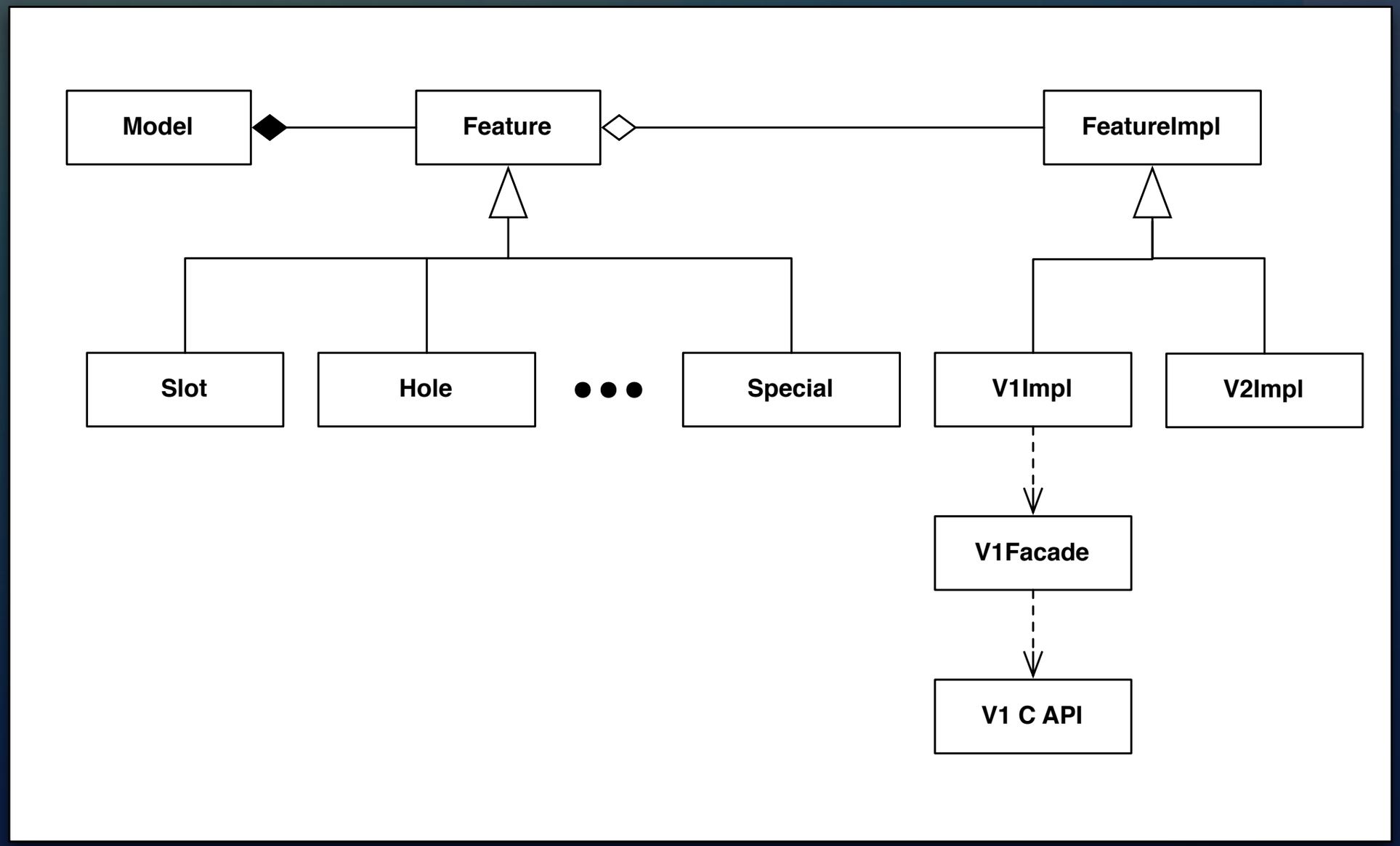


Assumes that Feature has a public interface that provides all of the information needed by expert system

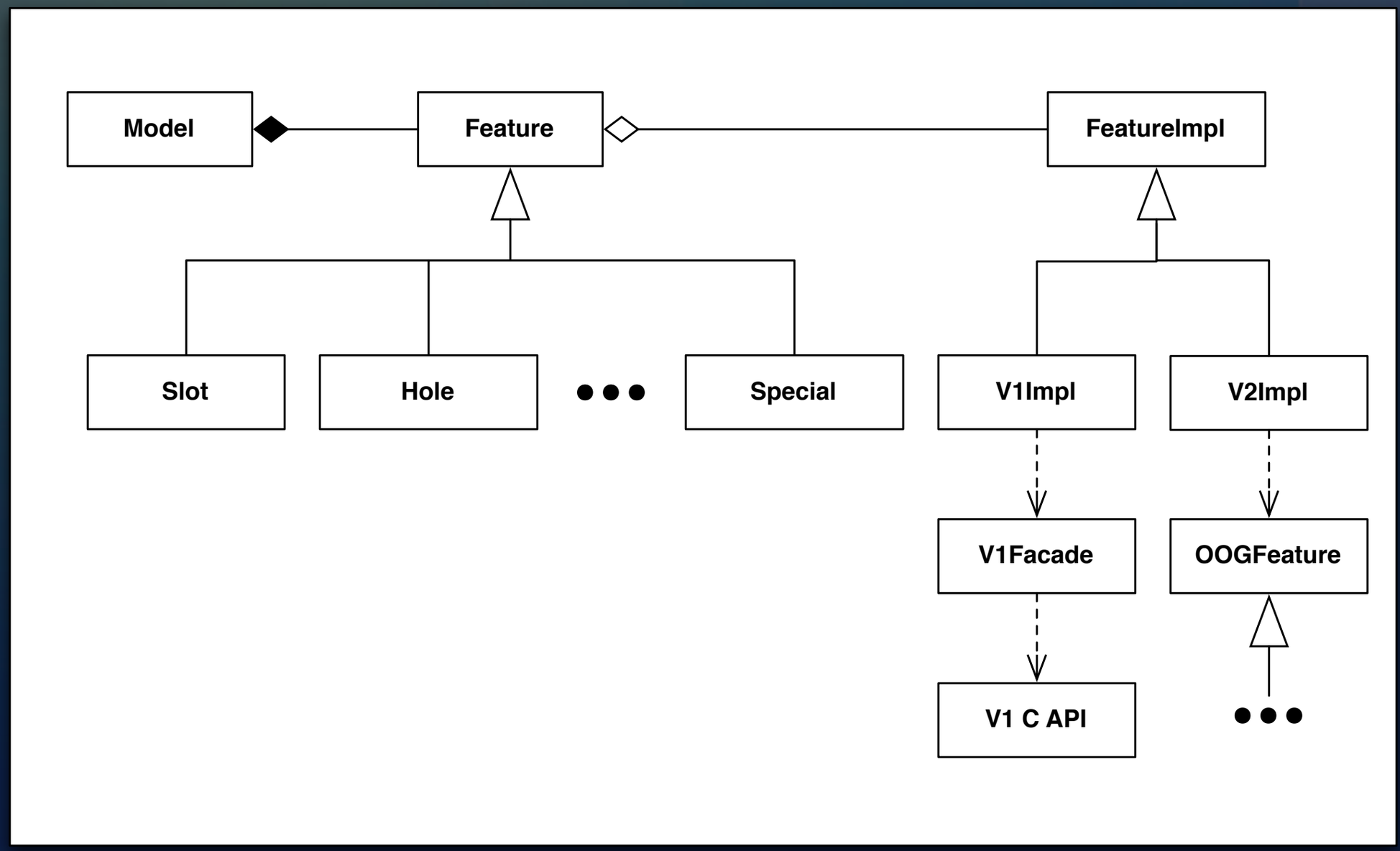
2c: Identify additional patterns

- All that is left in this particular system is to attach the V1 and V2 systems to the design
  - Adapter and Facade will do that for us, so no additional patterns are needed
- Looping back, we know that Adapter and Facade are independent of each other in this design
  - They can be applied in any order

Context for Facade



Context for Adapter (& Final Design)



Step 3: Add Detail

- At this point, we would start to add detail
  - What exactly is the public interface of Feature and FeatureImpl
  - How will each subclass of Feature implement that public interface by calling operations on FeatureImpl?

Is it better?

- Is the new design better?
  - The book suggests talking through each design
    - "Read the UML diagram"
      - The new design sounds simpler (especially because it can be explained using design patterns)
- Now consider, what happens when V3 of the CAD system comes along…
  - 6 new subclasses in 1st design; 2 new classes in the 2nd

Class Focus vs. Pattern Focus

- In the first design, we got to a state that works but it wasn't that maintainable
  - it had a class-based focus that stuck parts together from the bottom up, creating a whole
- In the second design, we started with the big picture, found the most suitable pattern and worked down, adding patterns that worked with the first one
  - the patterns then deliver on good software qualities because that's what they are all about!

Wrapping Up

- Went deeper into the pattern-based approach to software design by looking at Christopher Alexander's work more closely
  - Start with a conceptual understanding of a problem domain; identify patterns that highlight coarse-grained elements and relationships in the domain; use those patterns as context to implement additional more-refined patterns; repeat until problem is solved
- Saw an example of this approach applied to the CAD/CAM problem discussed earlier this semester

Coming Up Next

- Lecture 22: Advanced Design (chapters 14-16)
- Homework 6 due on Friday
- Homework 7 assigned on Friday
- Lecture 23: Decorator, Observer, Template Method