Problem Domain & Initial Design Plus More on Design and UML

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 5 — 09/06/2011

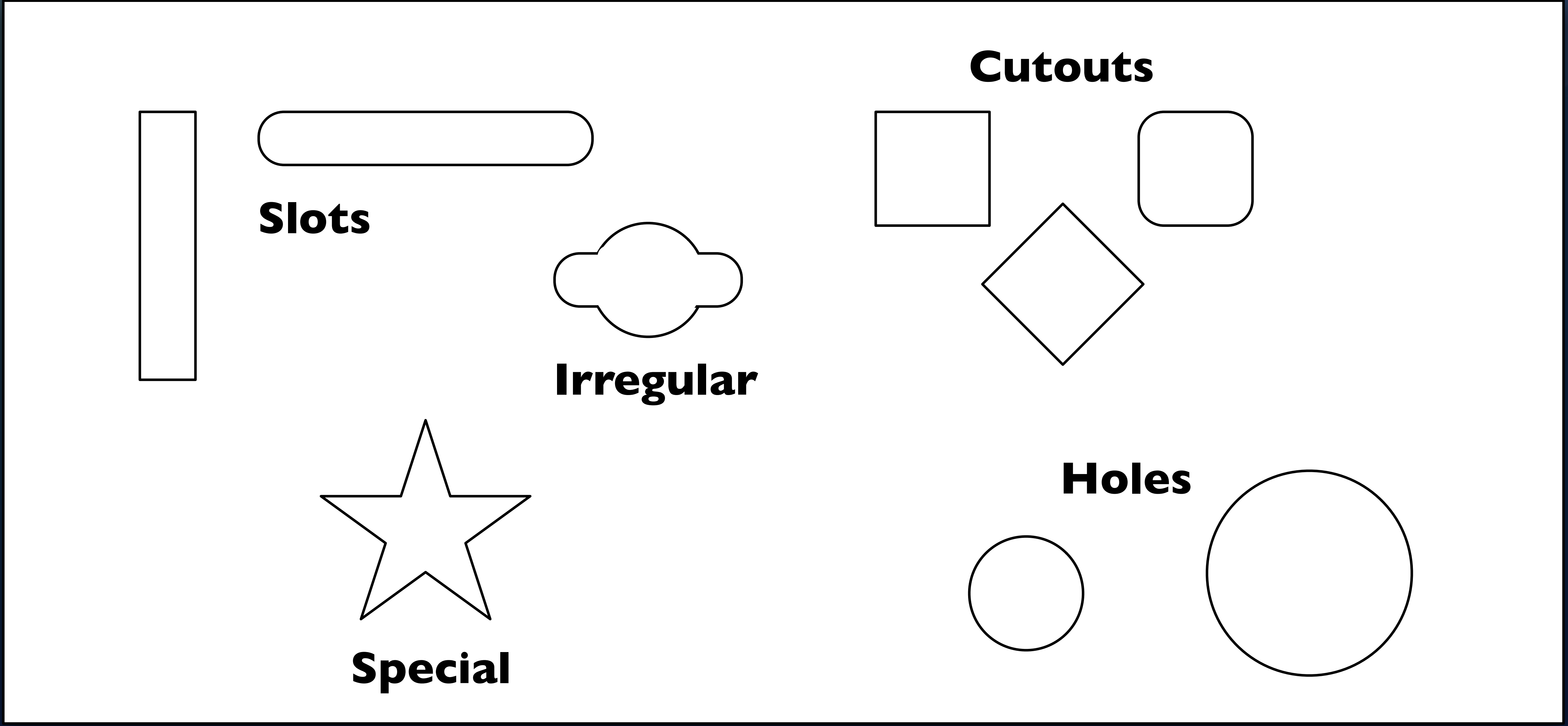
Goals of the Lecture

- Introduce and reflect on the problem domain of the book's running example
- Present an initial design to the problem domain
  - Highlight its strengths (if any) and weaknesses
- Then switch to an overview of the analysis phase
  - Use cases and other UML diagrams
  - How these diagrams work together

The Problem Domain

- A company provides software that
  - allows engineers to create models for parts made out of sheet metal
  - generates the instructions needed by a computer-controlled cutting tool to actually make the part specified by the models

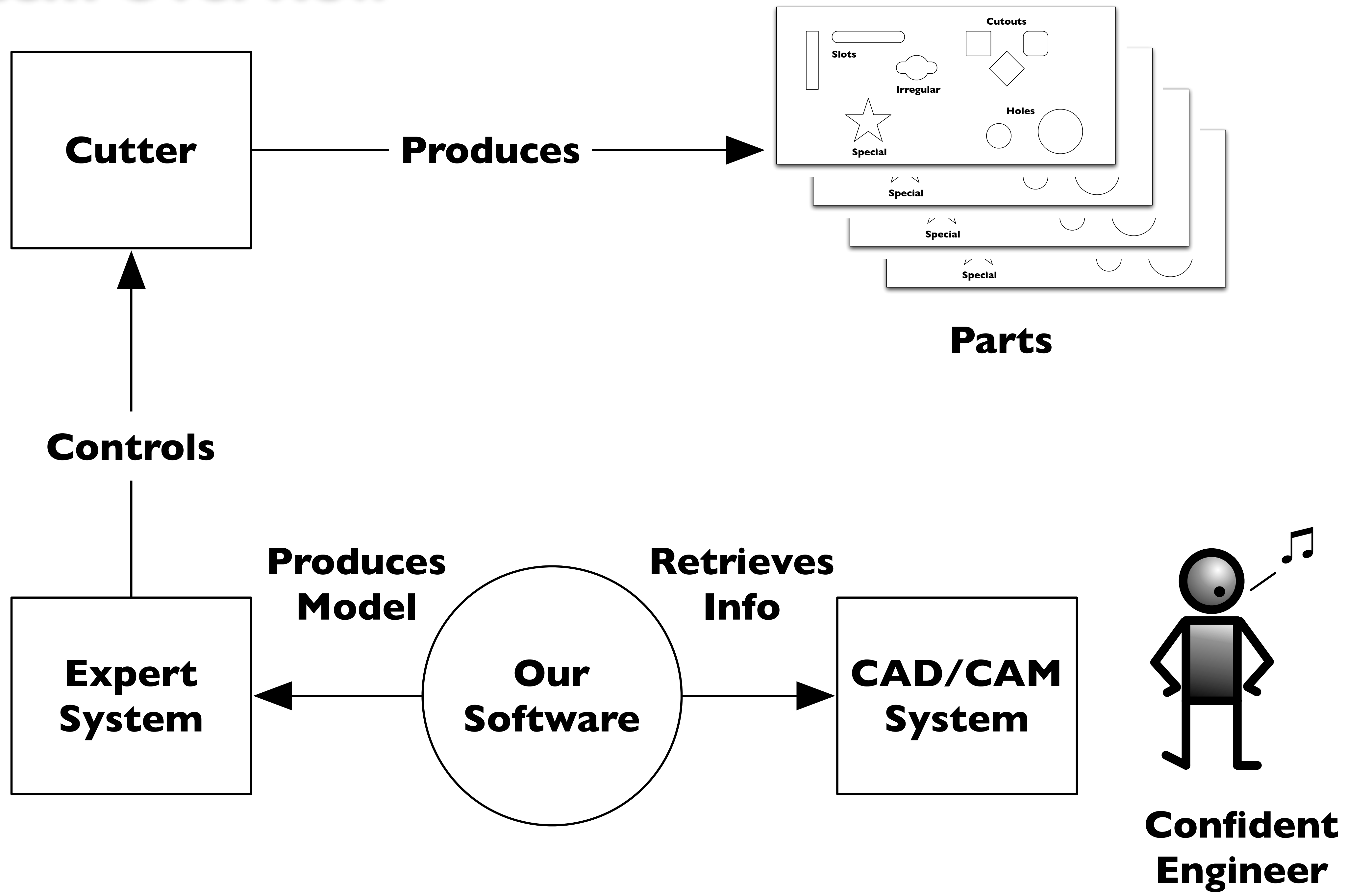
An example part with all 5 features



Feature?

- We have a terminology overlap
  - In previous lectures, I referred to an object's attributes and methods collectively as "features"
  - In this problem domain, a "feature" is a type of shape that can automatically be cut into a piece of sheet metal
  - **Terminology overlaps like this are common** when doing analysis and design
    - For this system, "Feature" is a domain concept and will eventually appear as a class in our design

System Overview



Nice system!

- The engineers get to use familiar tools when designing new parts
- The expert system encodes all the rules about how the cutter is used to create parts out of features
- Our software simply acts as the "glue" between these two major components
  - extracting information and converting it into a format that the expert system understands

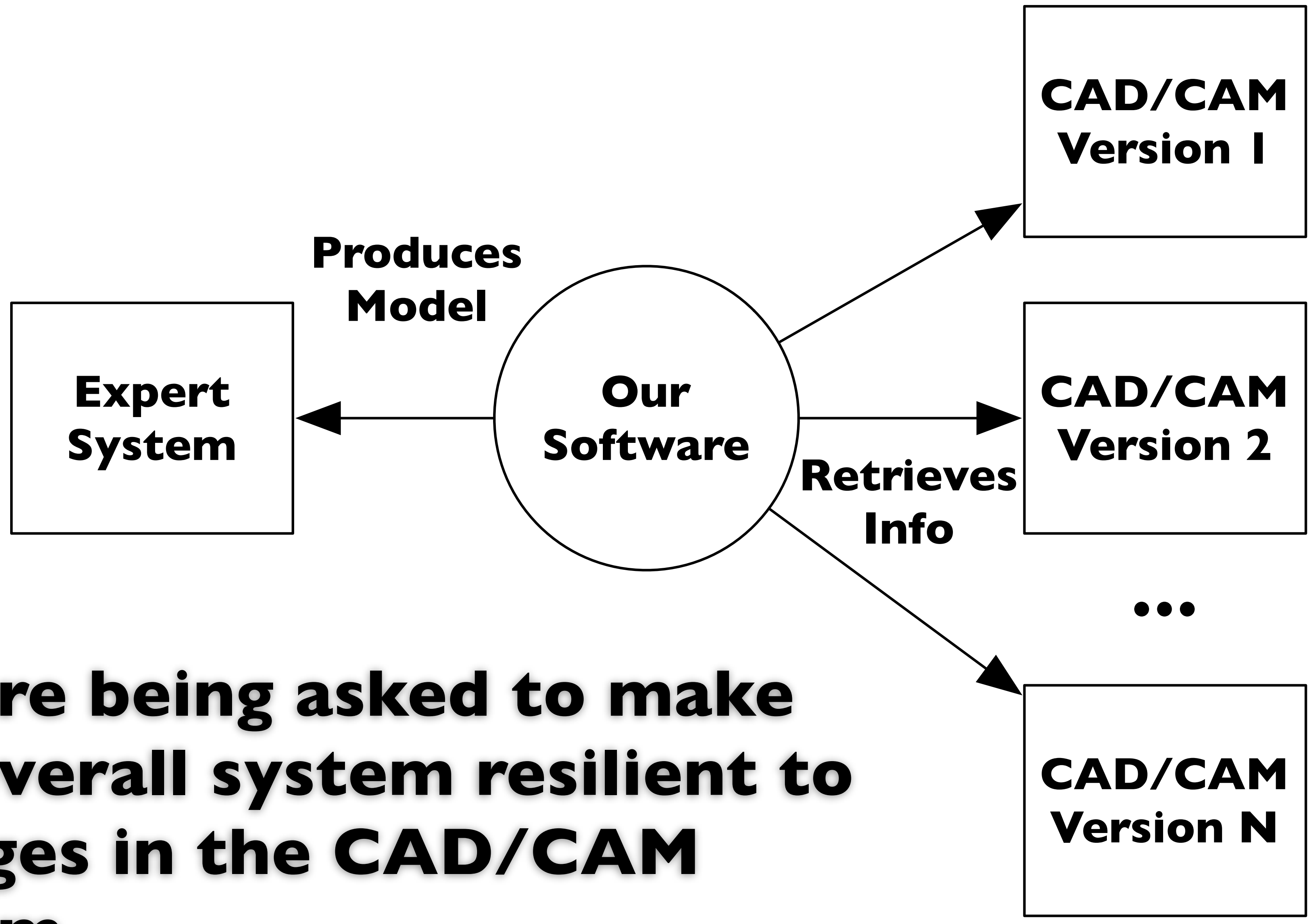
Discussion

- The use of existing CAD software was a good decision
  - Imagine if the original development team had been infected with **Not Invented Here** syndrome and had decided they needed to build a modeling tool
    - It would have increased expense and complexity
      - Plus their tool would likely have been non-standard
    - Sometimes, "buy" is the best option of a "buy vs. build" decision; be sure to leverage standards

So, What's the Problem?

- So far, all I've presented is information about the application domain
  - What we are missing is details concerning what the problem might be
- Don't confuse supplemental information or domain information for a problem statement
  - As designers, we need to know what the problem is

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…

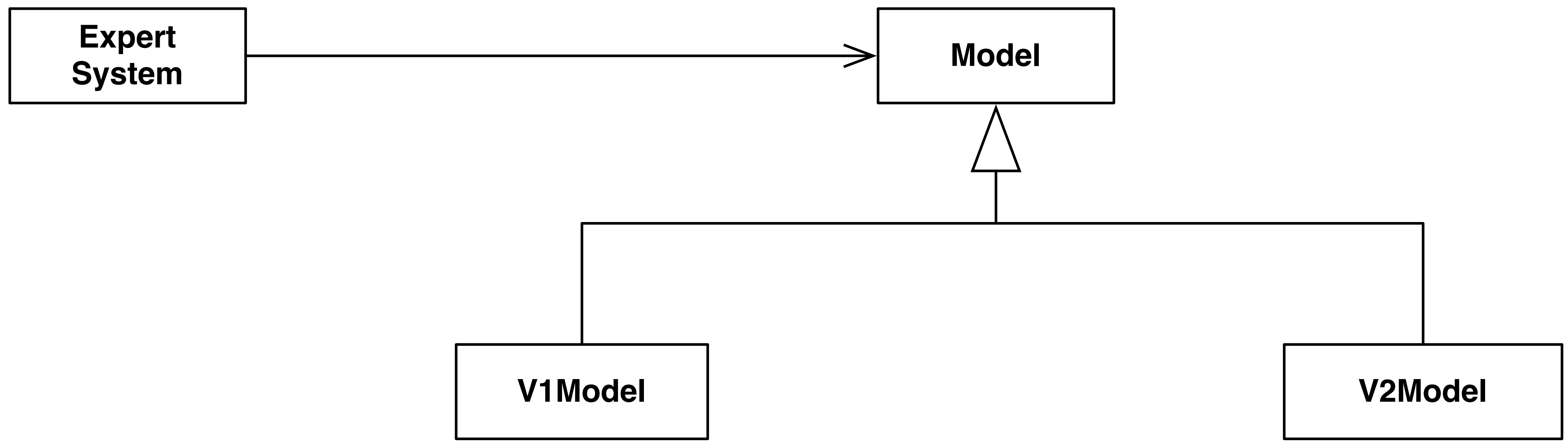
Discussion (I)

- Our problem is to allow the expert system to work with multiple CAD systems
  - currently different versions of the existing CAD system or (possibly) CAD systems from different vendors

Discussion (II)

- Why not replace the expert system?
  - It was an expensive piece of software to develop and embodies a significant amount of domain knowledge
    - Translating models into commands for the cutter is non trivial
      - punching features in the wrong order produces defective parts
- This type of legacy system is **common**; you just have to incorporate it into your design

Our Approach



We want to provide the expert system with a single model that it understands; we will subclass this model to integrate the different versions of the CAD system

Understanding the Challenges

- The API of version 1 of the CAD system is NOT object-oriented
  - It is accessed via a set of library routines
    - (think standard C library)
- The API of version 2 of the CAD system is object-oriented
  - It provides an OO framework of classes to describe its models

Example of Version 1 API

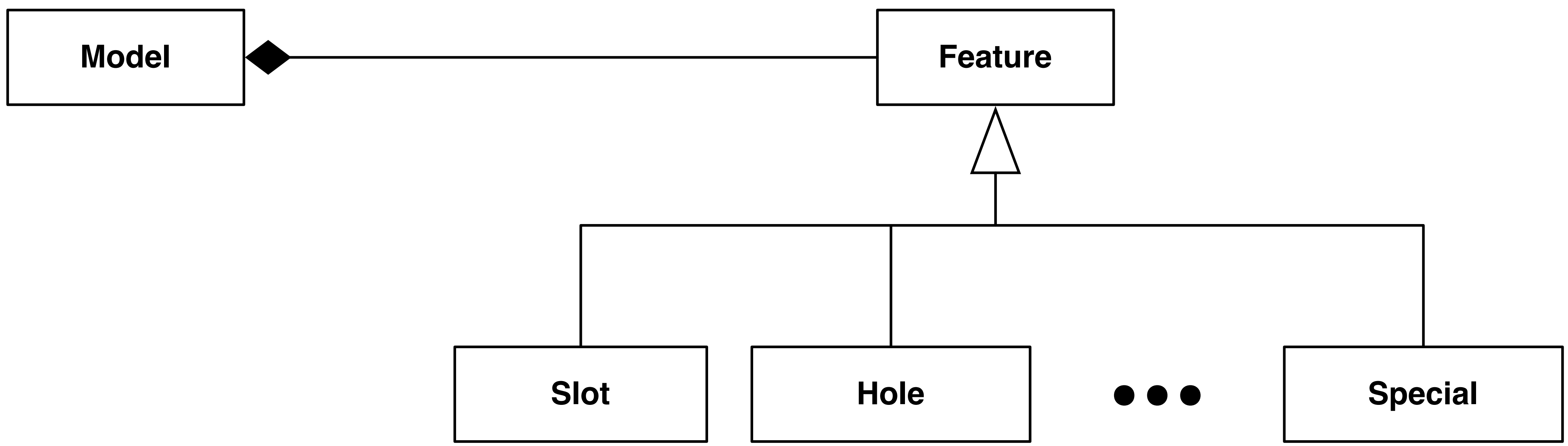
- `model_t *get_model(char *name);`
- `int number_of_features(model_t *model);`
- `int get_id_of_ith_feature(model_t *model, int index);`
- `feature_type get_feature_type(model_t *model, int id);`
- `int get_x_coord_of_slot(model_t *model, int id);`

- Gosh, I miss programming in C! ☺

Accessing the API

- To get the x coordinate of a feature, I need to do something like
  - `model_t *model = get_model(“part XYZ”);`
  - `int num = number_of_features(model);`
  - `for (int i = 0; i < num; i++) {`
    - `int id = get_id_of_ith_feature(model, i);`
    - `switch (get_feature_type(model, id)) {`
      - `case SLOT:`
        - `int x = get_x_coord_of_slot(model, id);`
        - …

Version 2's API



Much Better!

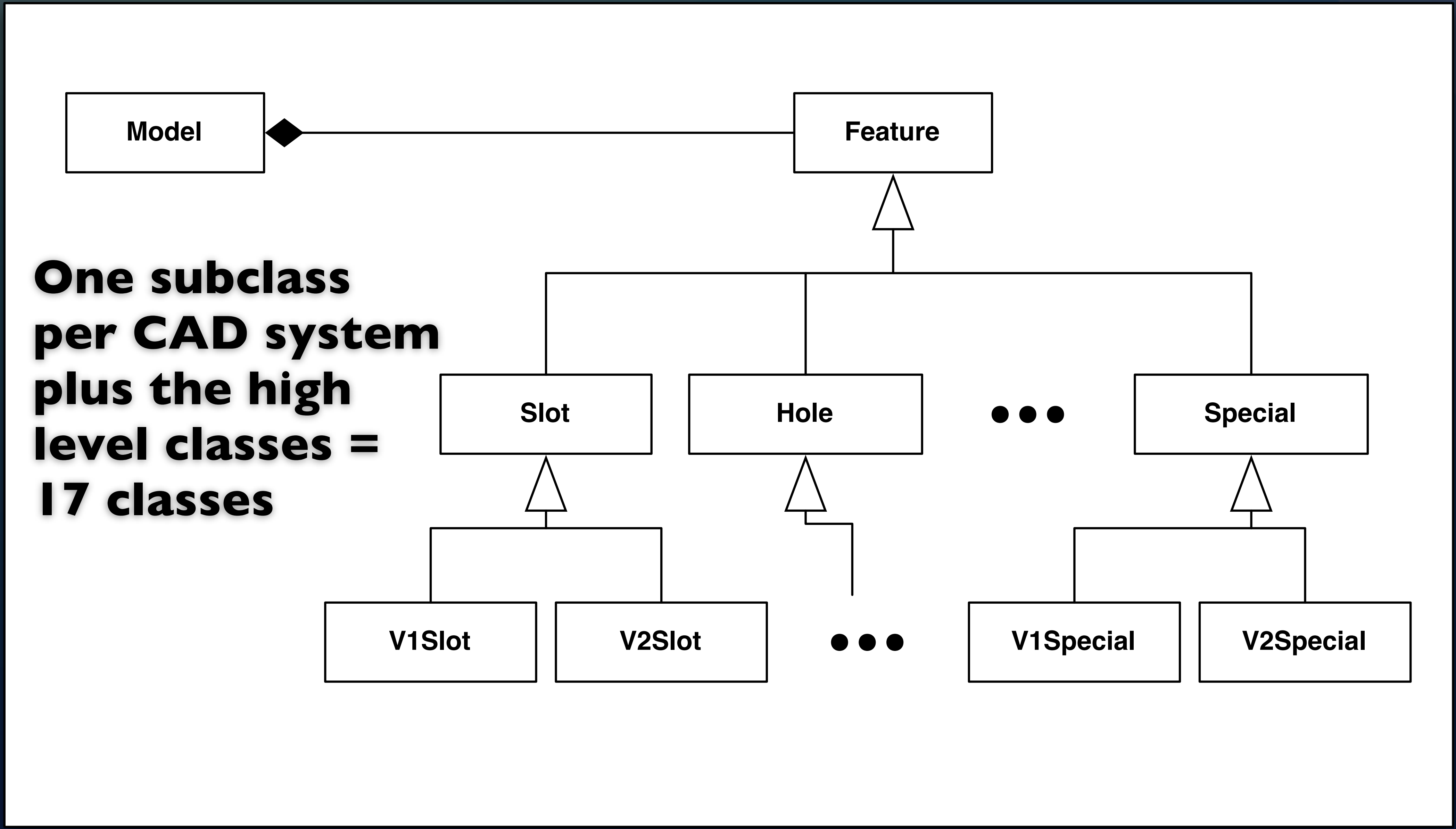
Discussion: The Challenge is Clear

- We want to give the expert system an OO API
  - Version 2 provides us with a nice OO model, so our system will need to "wrap" those classes in some way
  - Version 1 provides only library routines, so our system will need to "hide" the non-OO API from the expert system
- If we do this right, we will be able to write robust, polymorphic code for the expert system that doesn't change when support for a new CAD system is added to our system

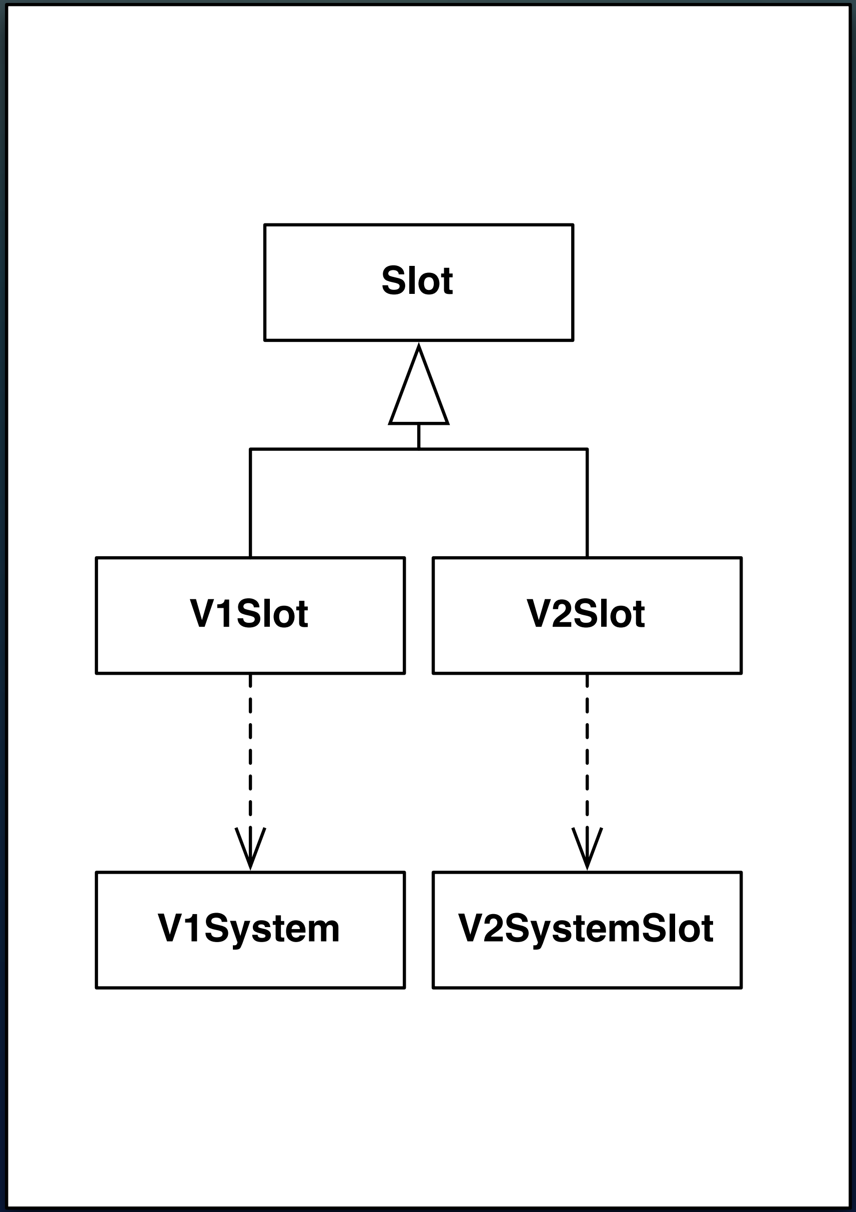
First Attempt: Not so Great

- In Chapter 4, an initial attempt to solve the problem is presented
  - "It is not a great solution, but it is a solution that would work."
- The idea is to present an obvious elaboration of the approach outlined so far
  - and then highlight some obvious problems it has
  - these problems will be dealt with later in the book

The Basic Approach (I)



The Basic Approach (II)



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"

Note on Polymorphism (I)

- The authors comment that their goal is not to achieve polymorphism across Features
  - In their design, they assign different sets of methods to different feature subclasses rather than trying to define all of the methods in the top level Feature class
    - The expert system needs to know the types of features it is dealing with
    - abstracting those details away will prevent it from doing its job

Note on Polymorphism (II)

- This means they are **not** striving to support client code like this
  - for (Feature f : features) {
    - f.doSomething();
  - }
- The expert system needs to differentiate among the various feature types; the design does achieve polymorphism across the V1* and V2* subclasses
  - Slot s = <retrieve a slot>; s.getLength(); // polymorphic across V1 and V2 subclasses

Problems with the Design (I)

- The design has four problems that the authors highlight
  - 1. Redundancy among methods
    - Lots of duplicated code or highly similar code is likely across V1 subclasses
      - OO designers hate duplicated code!
  - 2. "Messy", "Ill structured", "Cumbersome"
    - something doesn't feel quite right about the design

Problems with the Design (II)

- The design has four problems that the authors highlight
  - 3. Tight coupling
    - The design is tightly coupled to the different CAD systems; A lot of code will need to be changed or produced if a new CAD system is added or an existing one is changed
  - 4. Weak cohesion
    - core functionality is too widely dispersed across the various classes; Model is too simple a class

Potential for Class Growth

- The final problem is that the design does not scale nicely
  - (# of features * # of CAD systems) + 7 core classes
  - 5 features, 2 systems = 17 classes
  - 25 features, 10 systems = 257 classes (!!)
- especially if something else about the system suddenly started to vary, even the "worst case" of "# of expert systems"

Switching Gears

- Let's look at analysis and design more generically
- During analysis and design, we will
  - capture requirements,
  - brainstorm candidate objects and roles,
  - consider trade-offs and design alternatives,
  - and make decisions
- We will capture these decisions in UML diagrams and use cases

User Perspective (I)

- In analysis, as much as possible, we want to write our artifacts from the standpoint of a user
  - We will make frequent and consistent use of domain-related vocabulary and concepts
  - We will talk about the software system as a "black box"
  - We can describe **its inputs and its expected outputs** but we try to avoid discussing how the system will process or produce this information

User Perspective (II)

- **Use cases** are a technique for maintaining this perspective
  - we identify the **different types of users** for our system
  - we then **develop tasks** for each of the different types of user

Actors

- More formally, a user is represented by an **actor**
  - Each use case can have one or more actors involved
    - An actor can be either a **human user or a software system**
- Actors have two defining characteristics
  - They are **external to the system under design**
  - They take initiative and interact with our system
  - During a use case, they have a goal they are trying to achieve

Use Cases

- Each use case describes a single task for a particular actor
  - The description typically includes **one "success" case** and a **number of extensions** that document "exceptional" conditions
- Use cases are used to capture functional requirements
  - They can be annotated to also describe non-functional requirements but typically the focus is on functional requirements only

Example Use Case

What the Door Does

1. Fido barks to be let out.
2. Todd or Gina hears Fido barking.
3. Todd or Gina presses the button on the remote control.
4. The dog door opens.
5. Fido goes outside.
6. Fido does his business.
    6.1 The door shuts automatically
    6.2 Fido barks to be let back inside.
    6.3 Todd or Gina hears Fido barking (again).
    6.4 Todd or Gina presses the button on the remote control.
    6.5 The dog door opens (again).
7. Fido goes back inside.

If something goes wrong with step 6, then 6.1-6.5 kicks in to handle it

Goes hand in hand with requirements

Requirements List

1. The dog door opening must be at least 12” tall.

2. A button on the remote control toggles the state of the door: it opens the door if closed, and closes the door if open.

3. Once the dog door has opened, it should close automatically after a short delay

How are they related?

Requirements List

1. The dog door opening must be at least 12” tall.

2. A button on the remote control toggles the state of the door: it opens the door if closed, and closes the door if open.

3. Once the dog door has opened, it should close automatically after a short delay

What the Door Does

1. Fido barks to be let out.
2. Todd or Gina hears Fido barking.
3. Todd or Gina presses the button on the remote control.
4. The dog door opens.
5. Fido goes outside.
6. Fido does his business.
    6.1 The door shuts automatically
    6.2 Fido barks to be let back inside.
    6.3 Todd or Gina hears Fido barking (again).
    6.4 Todd or Gina presses the button on the remote control.
    6.5 The dog door opens (again).
7. Fido goes back inside.

Use cases contain scenarios

- Important concept
  - A complete path through a use case from the first step to the last is called a scenario
  - Most use cases have **multiple scenarios** but a **single user goal**
    - All paths try to achieve victory

Iterative Process

- Once you have written
  - requirements and use cases to fulfill them
    - and you've discussed the use cases with clients to determine the various alternate paths
  - You're ready to start creating class diagrams, activity diagrams, state diagrams and sequence diagrams
    - using information in the use cases as inspiration

What are Activity & State Diagrams?

- They represent alternate ways to record/capture design information about your system. They can help you identify new classes and methods
- They are typically used in the following places in analysis and design
  - After use case creation: create an activity diagram for the use case
  - For each activity in the diagram: draw a sequence diagram
    - Add a class for each object in the sequence diagrams to your class diagram, add methods in sequence diagrams to relevant classes

What are Activity & State Diagrams?

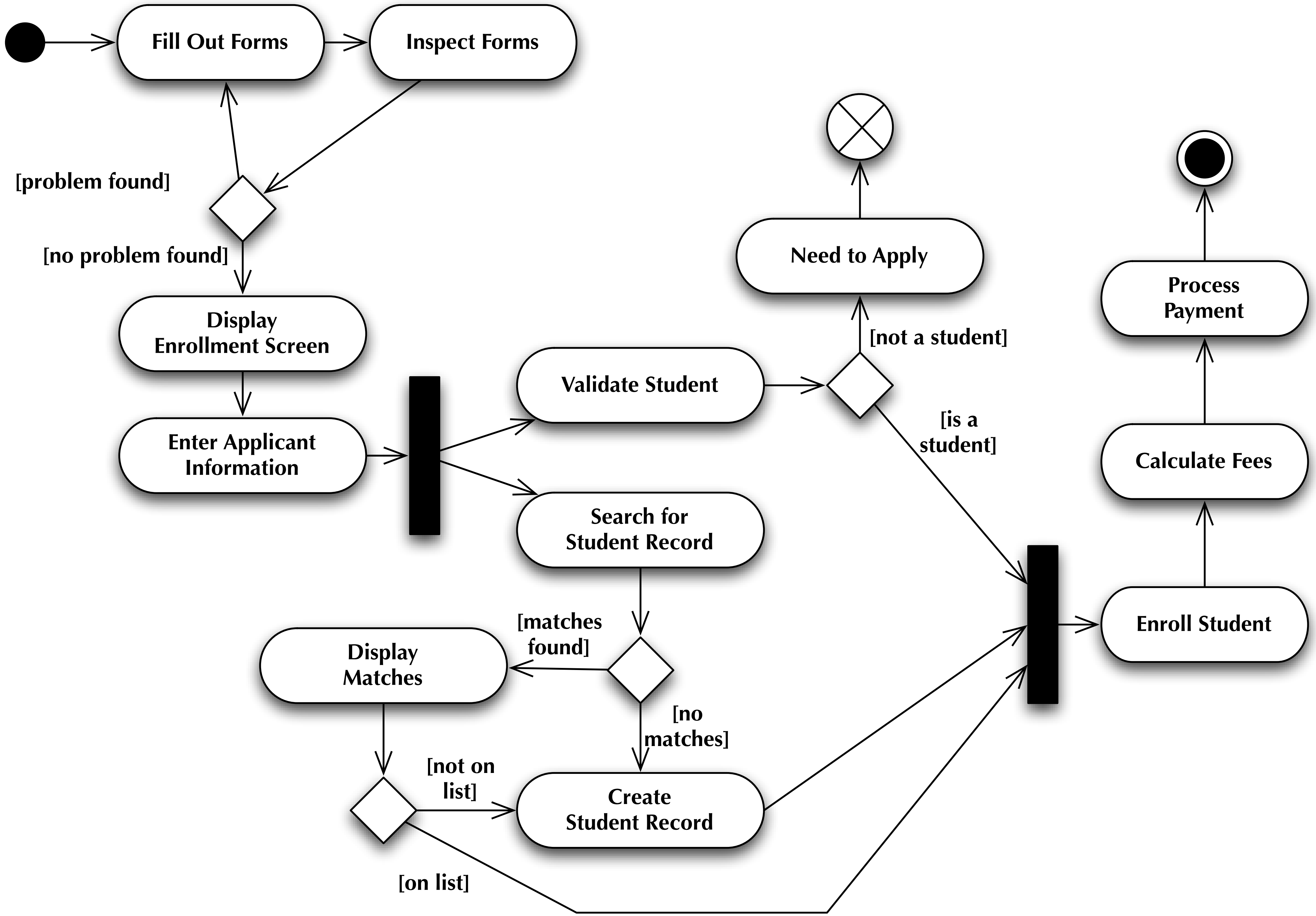
- Based on the information in the activity and sequence diagrams, see if you can partition an object's behavior into various categories (initializing, acquiring info, performing calcs, …)
  - Create a state diagram for the object that documents these states and the transitions between them (transitions typically map to method calls)

Activity Diagrams (I)

- Think "Flow Chart on Steroids"
  - Able to model complex, parallel processes with multiple ending conditions
- Notation
  - Initial Node (circle)/Final Node (circle in circle)/Early Termination Node (circle with x through it)
  - Activity: Rounded Rectangle indication an action of some sort either by a system or by a user

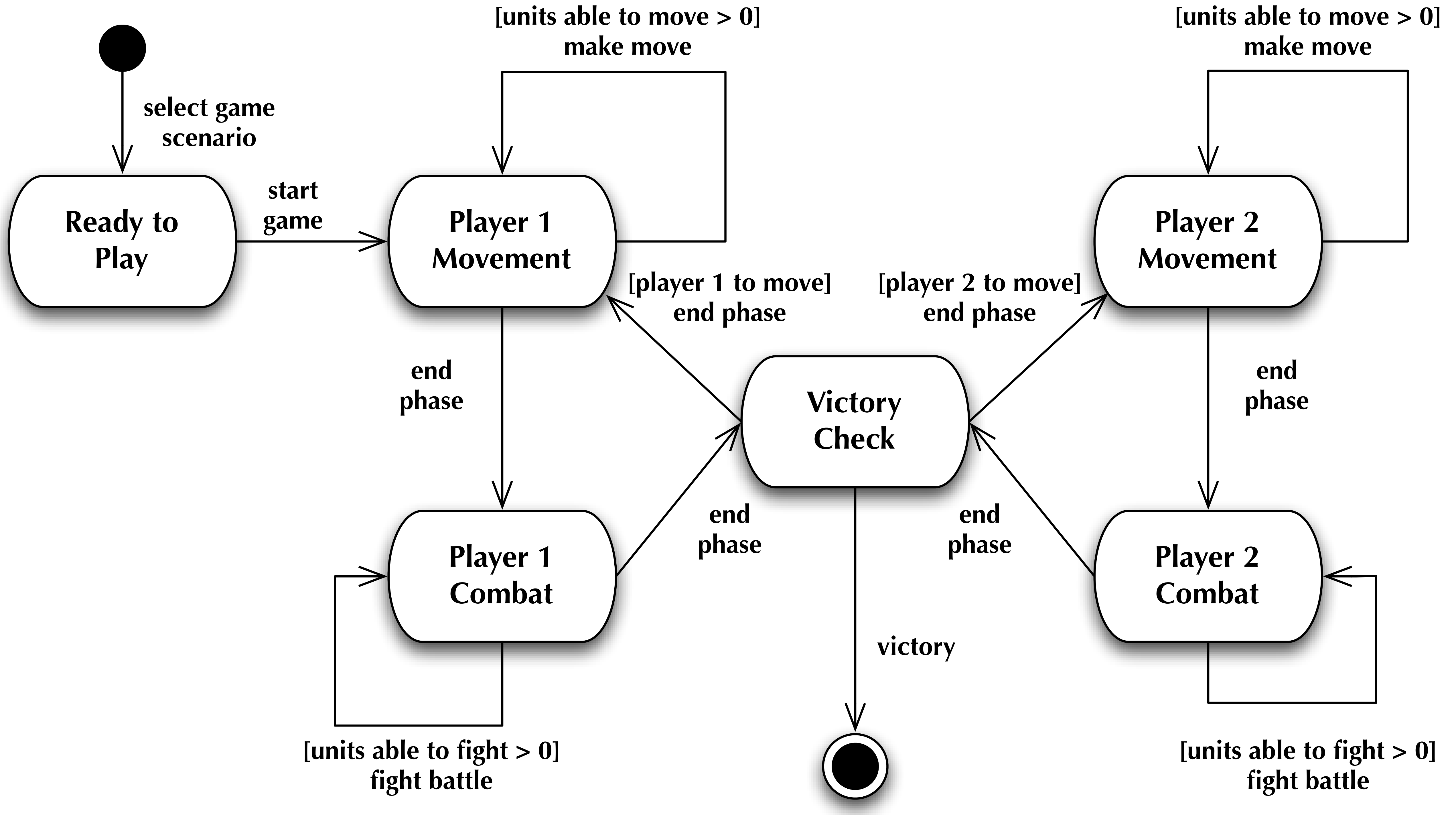
Activity Diagrams (II)

- Notation
  - Flow: directed lines between activities and/or other constructs. Flows can be annotated with guards "[student on list]" that restrict its use
  - Fork/Join: Black bars that indicate activities that happen in parallel
  - Decision/Merge: Diamonds used to indicate conditional logic.

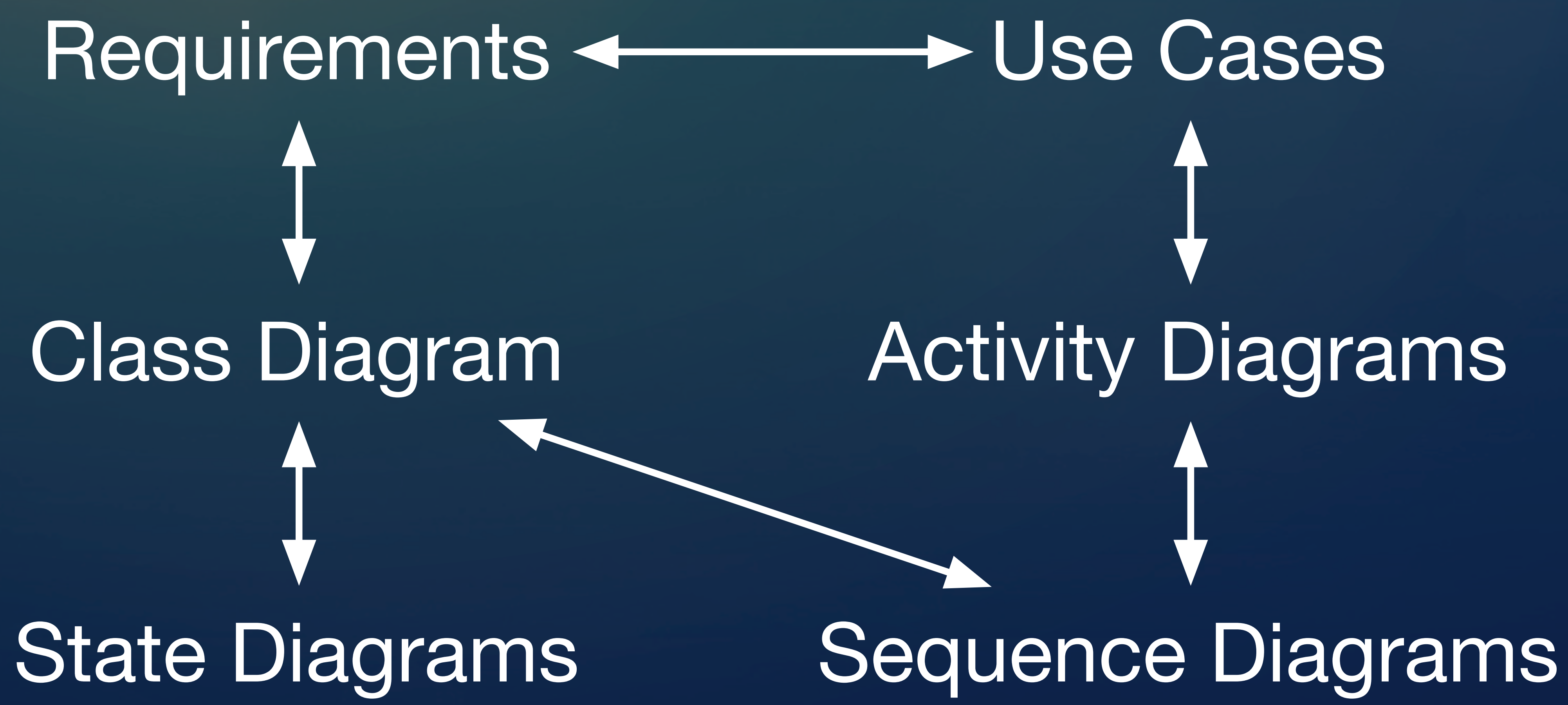

Fill Out Forms
Inspect Forms
[problem found]
[no problem found]
Display Enrollment Screen
Enter Applicant Information
Validate Student
Search for Student Record
Need to Apply
[not a student]
[is a student]
Process Payment
Calculate Fees
Enroll Student
[matches found]
Display Matches
[no matches]
[not on list]
Create Student Record
[on list]

State Diagrams

- Shows the major states of an object or system
  - Each state appears as a rounded rectangle
  - Arrows indicate state transitions
    - Each transition has a name that indicates what triggers the transition (often times, this name corresponds to a method name)
    - Each transition may optionally have a guard that indicates a condition that must be true before the transition can be followed
  - A state diagram also has a start state and an end state


[units able to move > 0]
make move
[units able to move > 0]
make move
select game
scenario
Ready to
Play
start
game
Player 1
Movement
Player 2
Movement
[player 1 to move]
end phase
[player 2 to move]
end phase
end
phase
end
phase
Victory
Check
end
phase
end
phase
Player 1
Combat
Player 2
Combat
victory
[units able to fight > 0]
fight battle
[units able to fight > 0]
fight battle

Relationships between OO A&D Software Artifacts



Wrapping Up

- We've seen an application domain with a specific problem
  - We've seen an initial (poor) OO design to solve it
- We then took a step back and looked at some of the activities in OO A&D that our book doesn't focus on
  - including the creation of use cases and new UML diagrams our book doesn't discuss
- Finally, we looked at how all our diagram types support an iterative approach to analysis and design

Coming Up Next

- Homework 2 Due Friday
- Lecture 6: Introduction to Design Patterns
  - Read Chapter 5 of the Textbook
- Homework 3 Assigned on Friday
- Lecture 7: Facade and Adapter
  - Read Chapters 6 and 7 of the Textbook