Problem Domain & Initial Design Plus More on Design and UML

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 6 — 02/04/2019

Task number one

- If you have your note card from the first class, please place it in front of you…
- If not…

- Get a marker and a piece of card stock
- Fold it in half to make a little tent
- Write the name you’d like to be called on that
- Face the name towards me
- Try to remember to bring that to class for the next several weeks

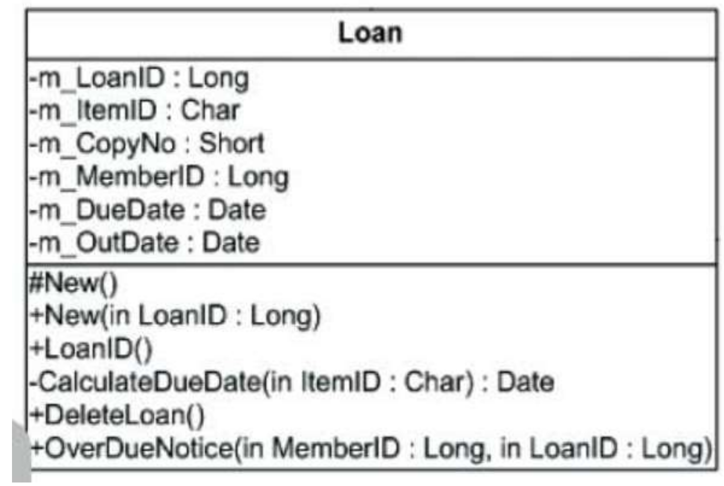
Acknowledgement & Materials Copyright

- Dr. Ken Anderson is a Professor of the Department of Computer Science and the Associate Dean for Education for the College of Engineering & Applied Science
- Ken taught this OOAD class on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

Clarification on Class Diagrams and Data/Method Accessibility

You can use UML to notate which accessibility you want each member to have. The three most common types of accessibility available in most object-oriented languages are as follows:

- **Public**—Notated with a plus sign (+). This means all objects can access this data or method.
- **Protected**—Notated with a pound sign (#). This means only this class and all of its subclasses (i.e. derivations) can access this data or method.
- **Private**—Notated with a minus sign (–). This means that only methods of this class can access this data or method.
- There are others – package, derived, static – expect to see variations in this by language!



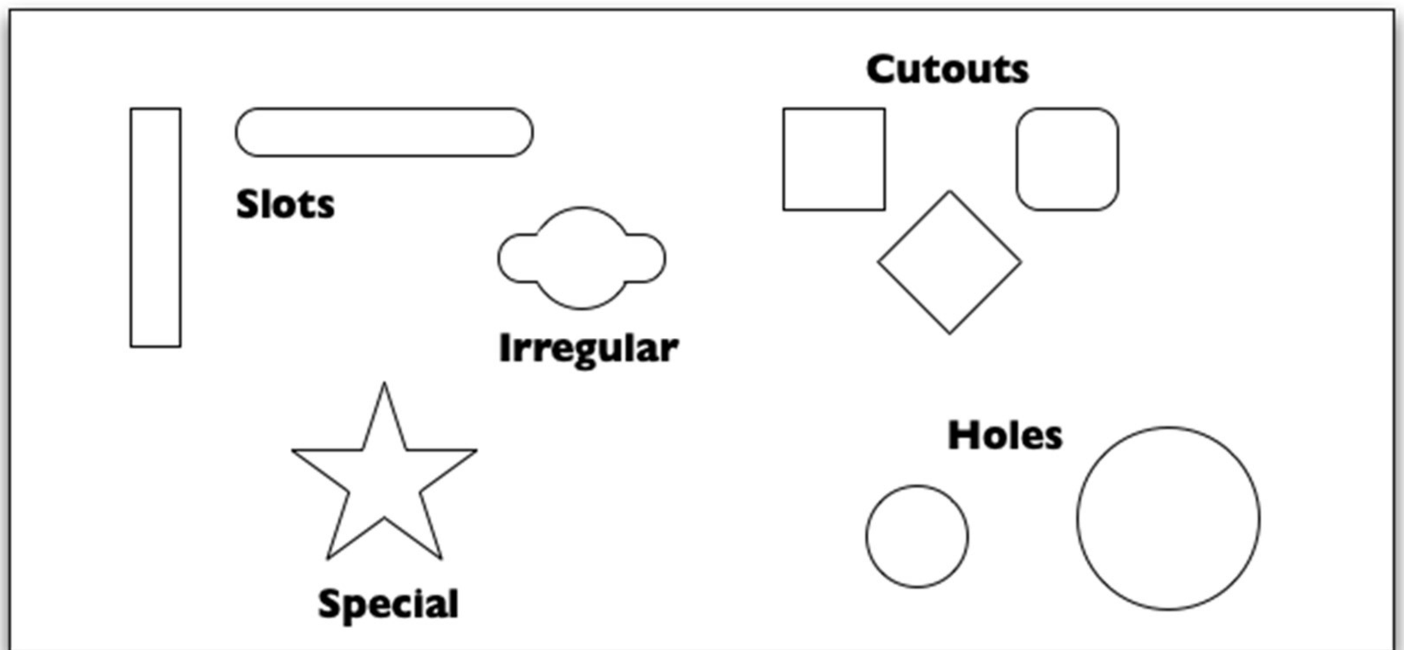
http://www2.sys-con.com/itsg/virtualcd/dotnet/archives/0105/clark/index.html

Goals of the Lecture

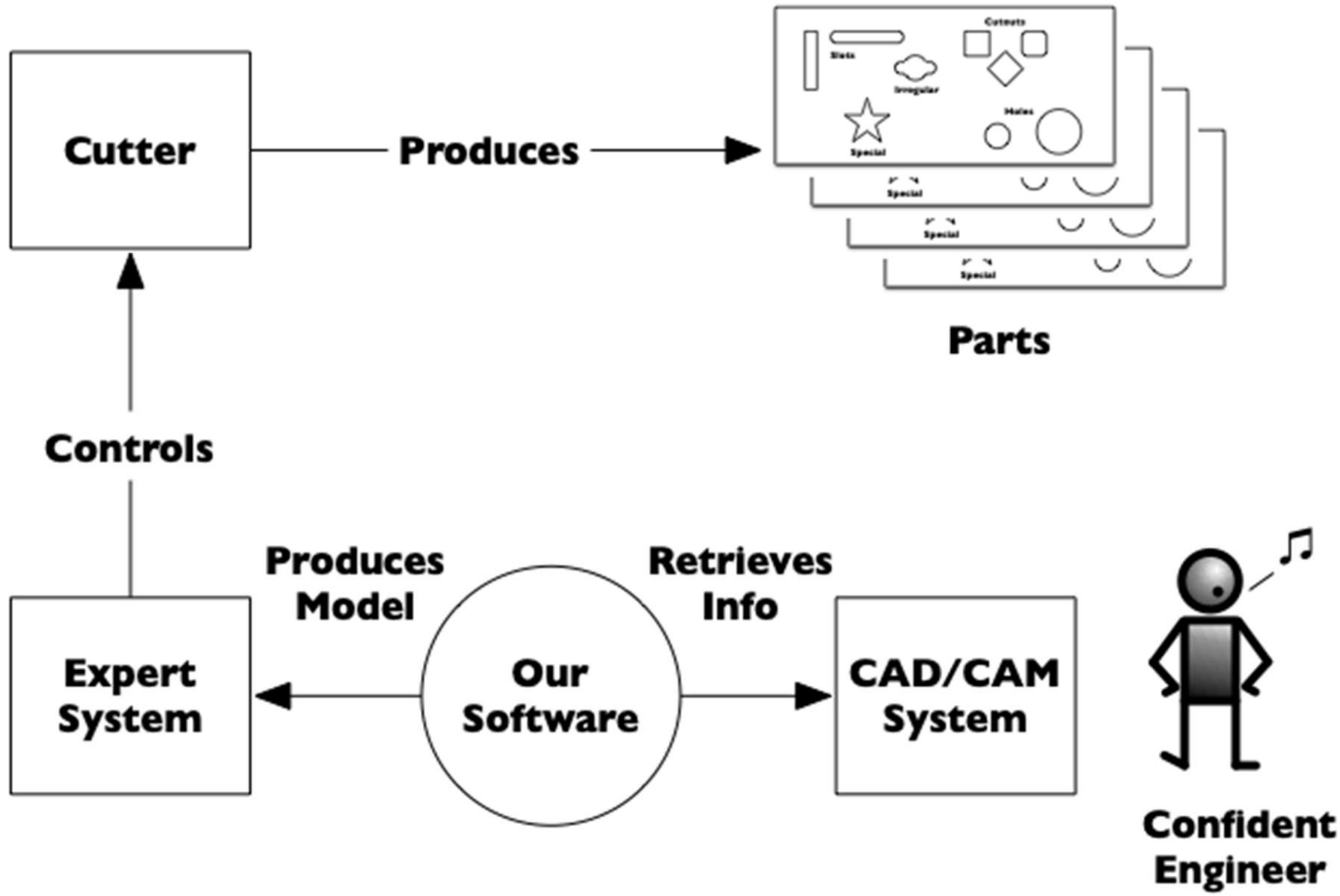
- Introduce and reflect on the problem domain of the book's running example
- Present an initial design to the problem domain
  - Highlight its strengths (if any) and weaknesses
- Then switch to an overview of the analysis phase
  - Use cases and other UML diagrams
  - How these diagrams work together

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
- Example part with five “feature” types



System Overview



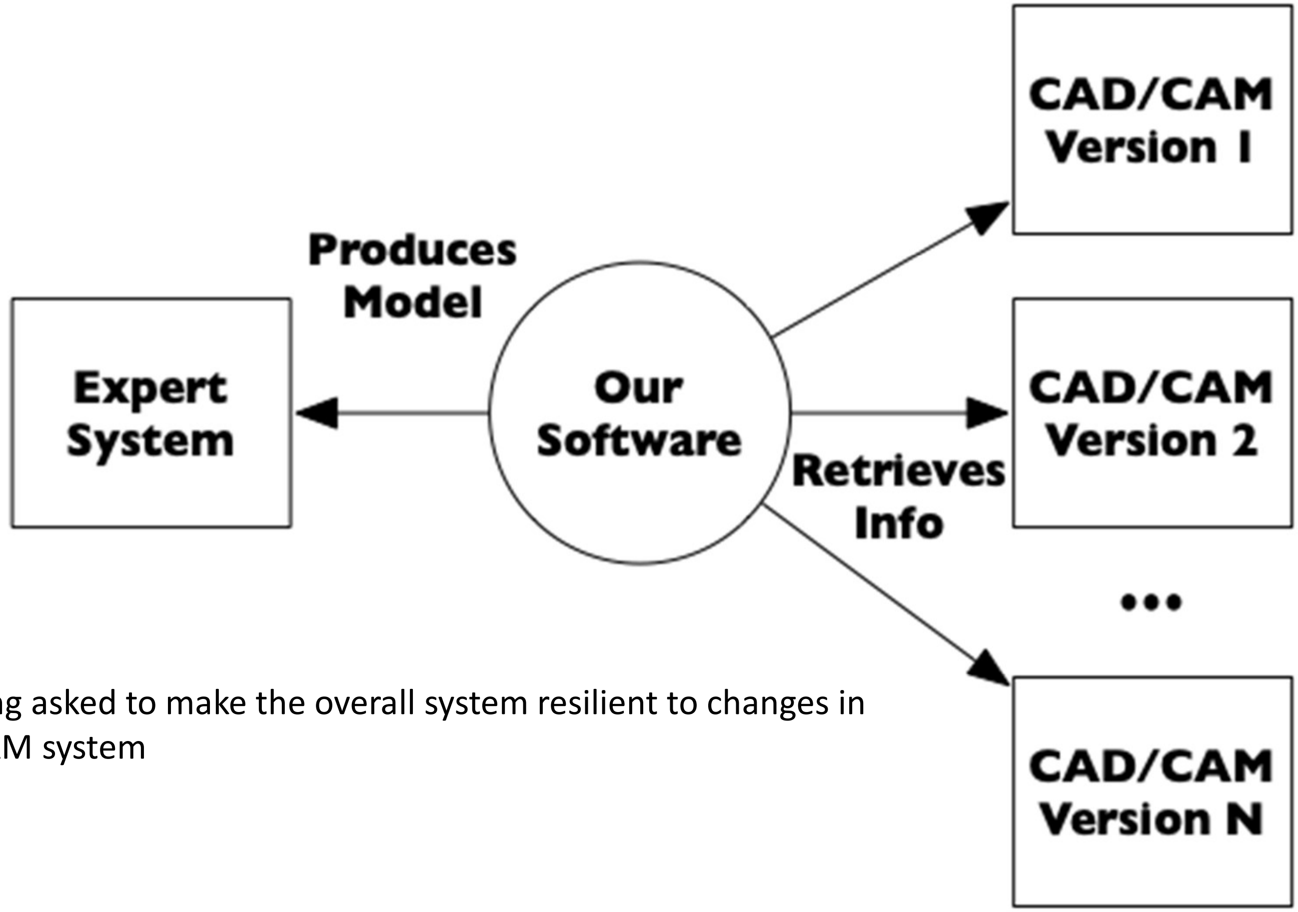
Nice system!

- 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
- 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 in your system designs

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/domain information for a problem statement**
  - As designers, we need to know **what the problem is**

Here's the Problem



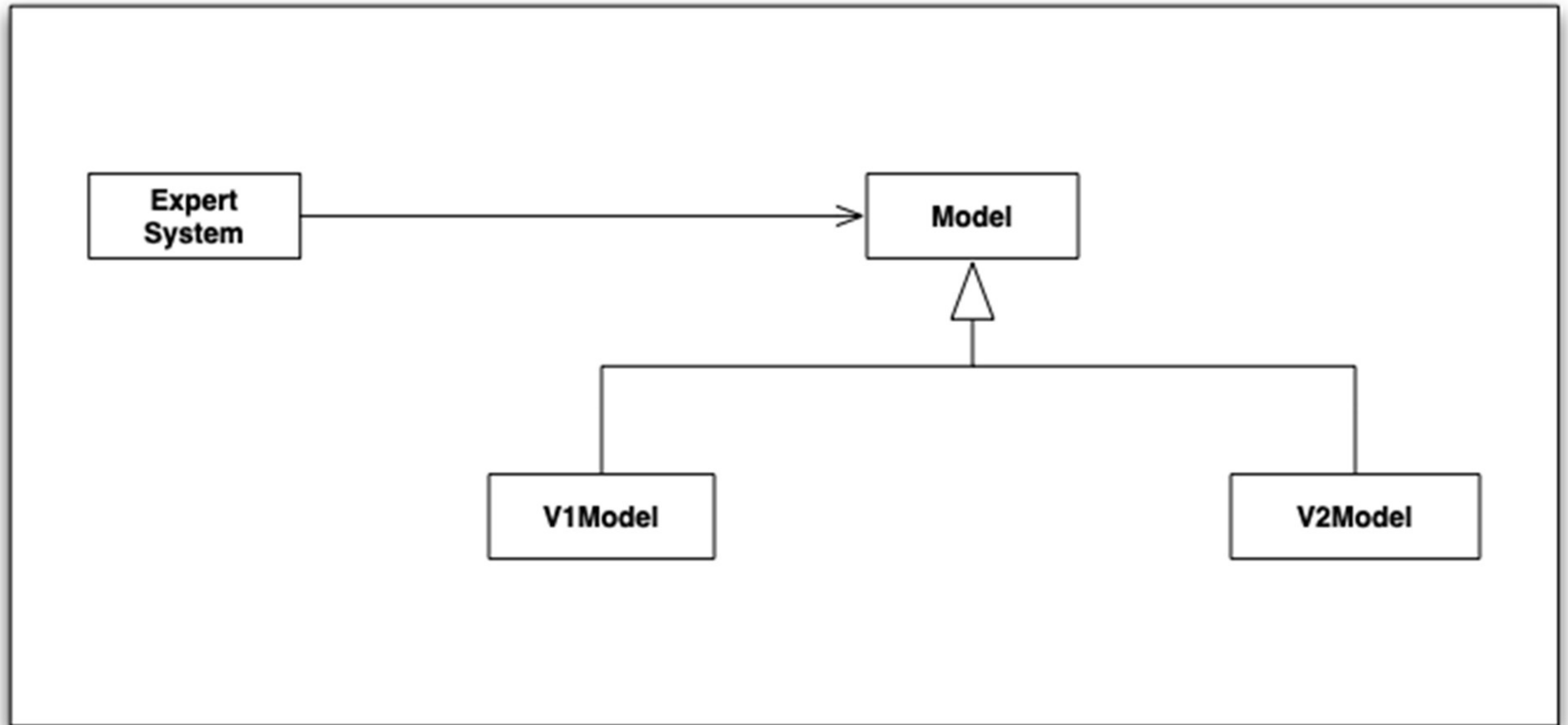
We are being asked to make the overall system resilient to changes in the CAD/CAM system

Example of encapsulation via software architecture…

Discussion

- 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
- 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

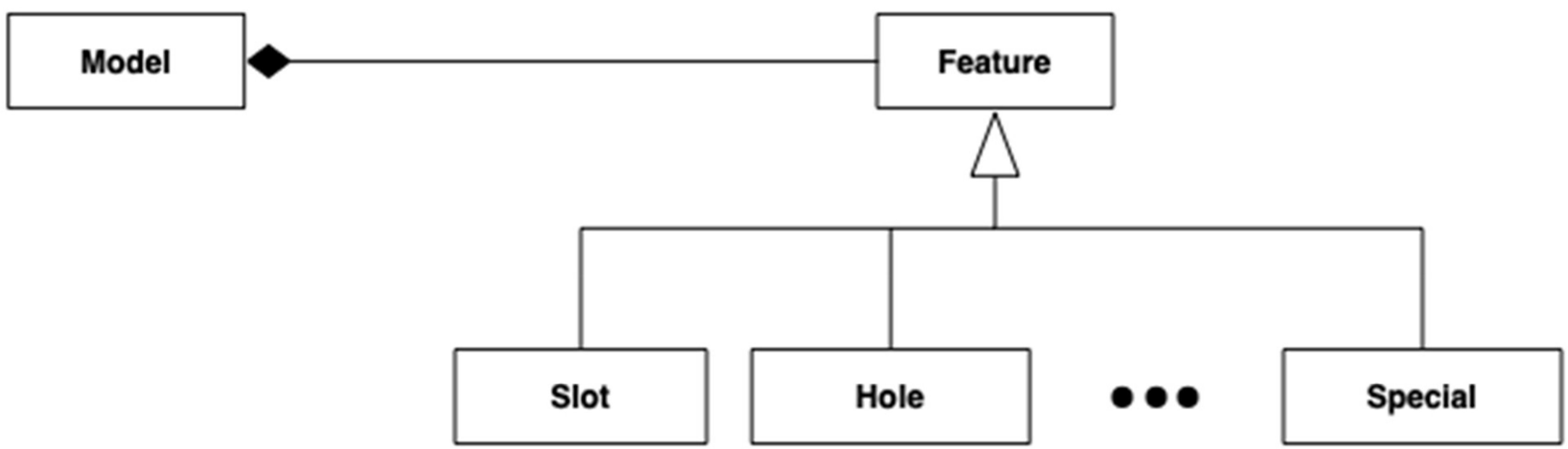
- The API of version 1 of the CAD system is NOT object-oriented
  - It is accessed via a set of library routines
    - (think C API)
- The API of version 2 of the CAD system is object-oriented
  - It provides an OO framework of classes to describe its models

Accessing the Version 1 API

- V1 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);
- (Lovely C code, I almost remember how it works.)
- 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

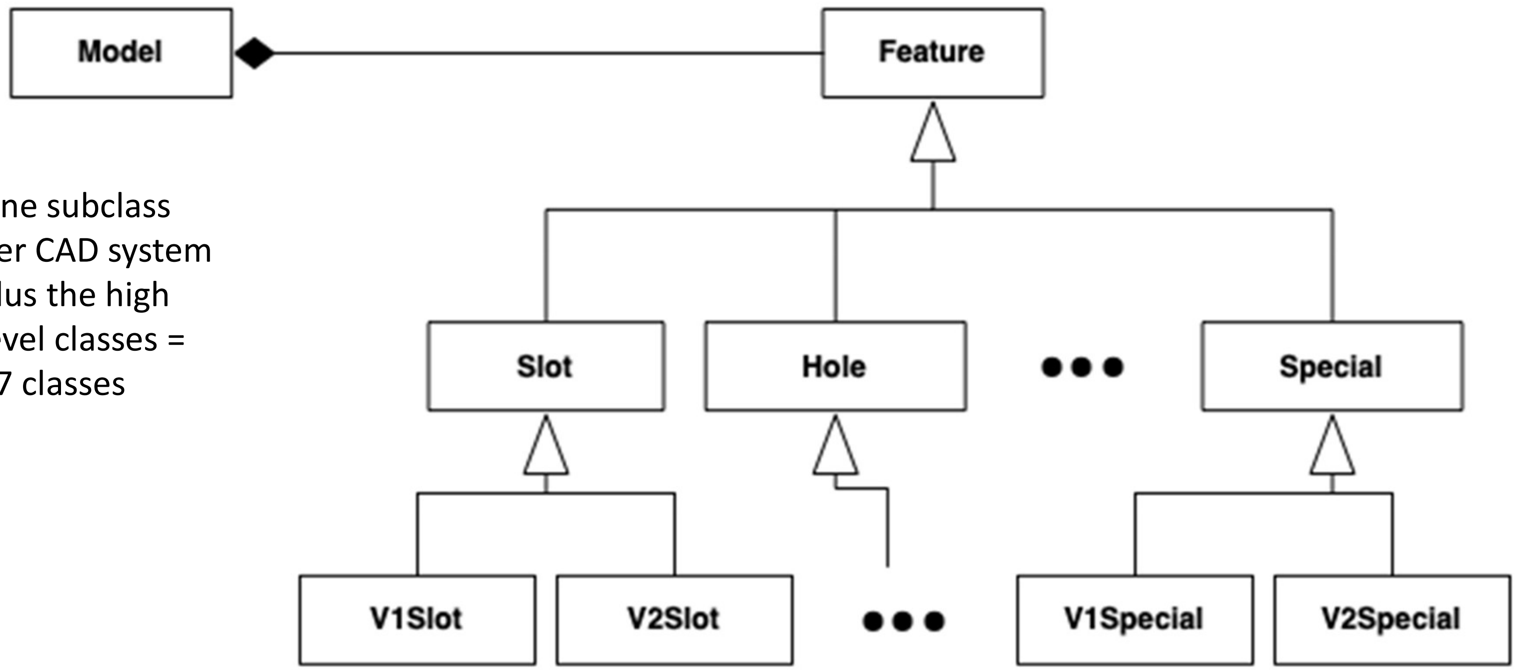
- We want to give the expert system an OO API
  - Version 2 provides us with a nice OO model, so our system will need to “wrap” those classes in some way
  - Version 1 provides only library routines, so our system will need to “hide” the non-OO API from the expert system
- If we do this right, we will 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

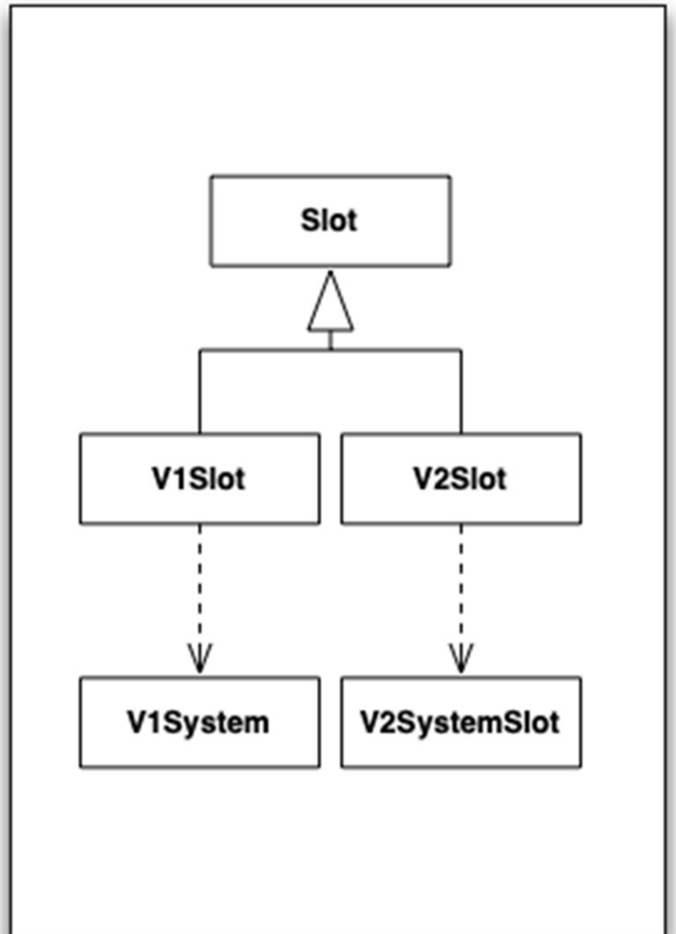
- In Chapter 4 of the textbook, an initial attempt to solve the problem is presented
  - “It is not a great solution, but it is a solution that would work.”
- The idea is to present an obvious elaboration of the approach outlined so far
  - and then highlight some obvious problems it has
  - these problems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

The Basic Approach (I)

One subclass
per CAD system
plus the high
level classes =
17 classes



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

- 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
  - This situation is less than ideal (it would be nice to put the knowledge of what to do for a particular feature inside of it) but
    - Here we're in a situation where "figuring out what to do" cannot be isolated inside a single class; different combinations of features require different strategies
- 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

- 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
  - 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"

- Will the OO design patterns help us with these problems?…

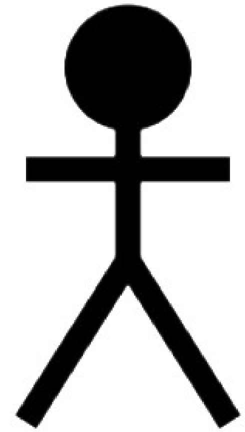
Switching Gears

- 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 either text-based or UML use cases

User Perspective and Use Cases

- 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
- In UX oriented workflows, understanding the user and their tasks are key
  - A typical UX development process might include
    - Analysis and Planning
    - User and Task Research (<- Use cases)
    - Interface and Interaction Design
    - Verification and Validation
- Use cases help maintain the user perspective
  - We identify the different types of users for our system – “who”
  - We then develop tasks for each of the different types of user – “what”
- 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

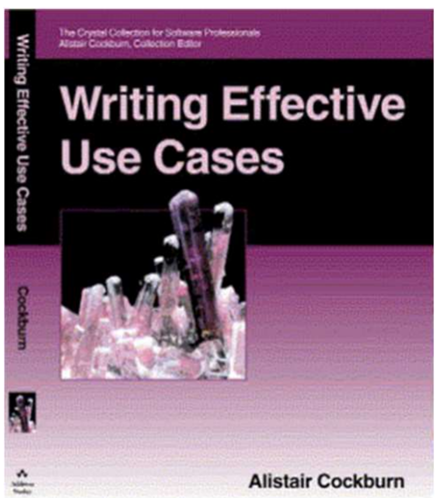
Actors



- 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
- Each use case describes a task or tasks for a particular actor
  - The description typically includes one "success" case and a number of extensions that document "exceptional" conditions

Text-based Use Cases

- From a presentation by Alistair Cockburn, author of Writing Effective Use Cases
  - Presentation is: Agile Use Cases
  - http://alistair.cockburn.us/get/2231
  - What is and isn't a use case good for:



Good use cases

are	aren't
Text	UML use case diagrams
No GUI	describing the GUI
No data formats	describing data formats
3 - 9 steps in main scenario	multiple-page main scenario
Easy to read	complicated to read
At user's goal level	at program-feature level
Record of decisions made	tutorial on the domain

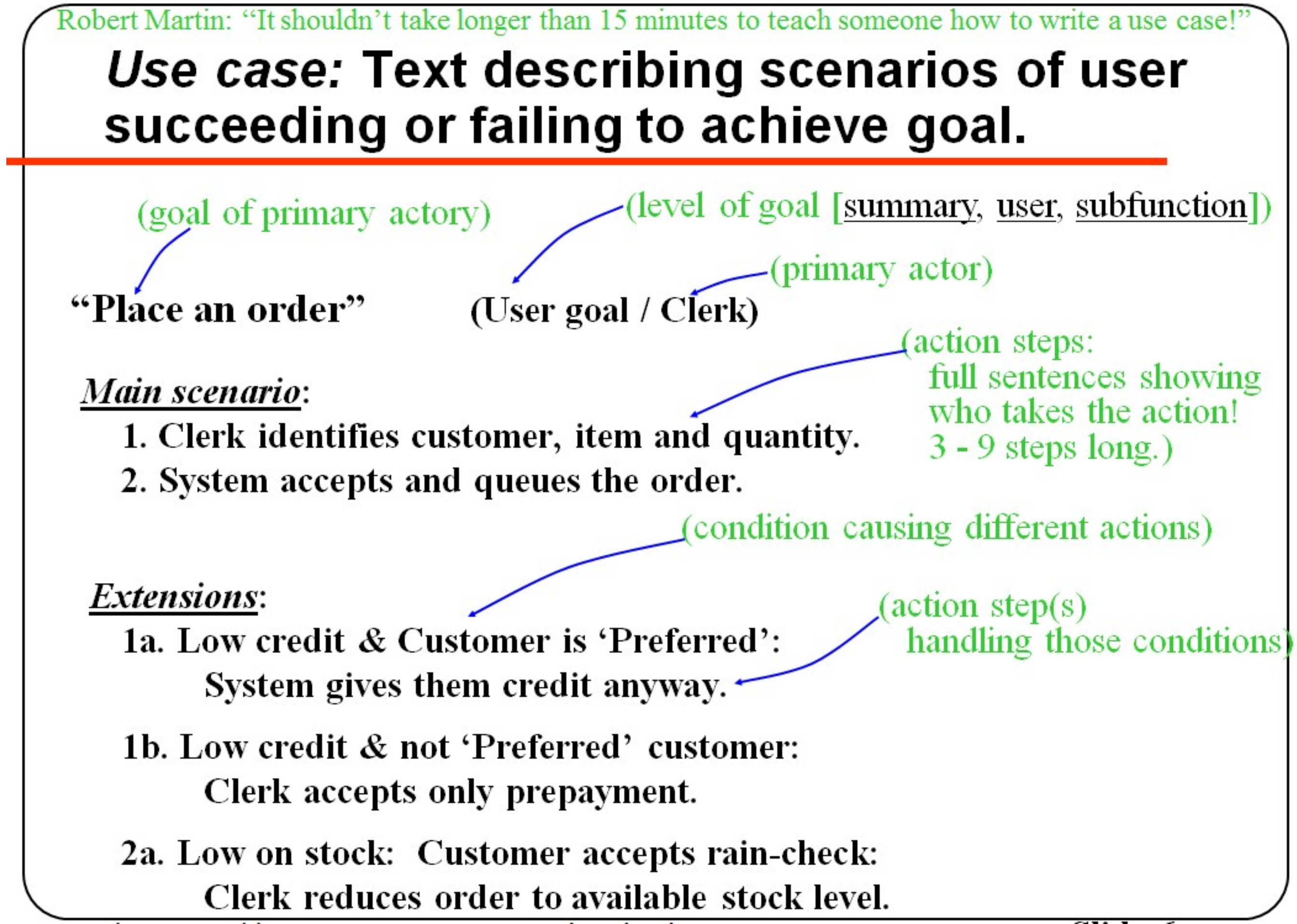
Use cases *can be* written --

all up front --or-- just-in-time

each to completion --or-- in (usable) increments

Text-based Use Cases

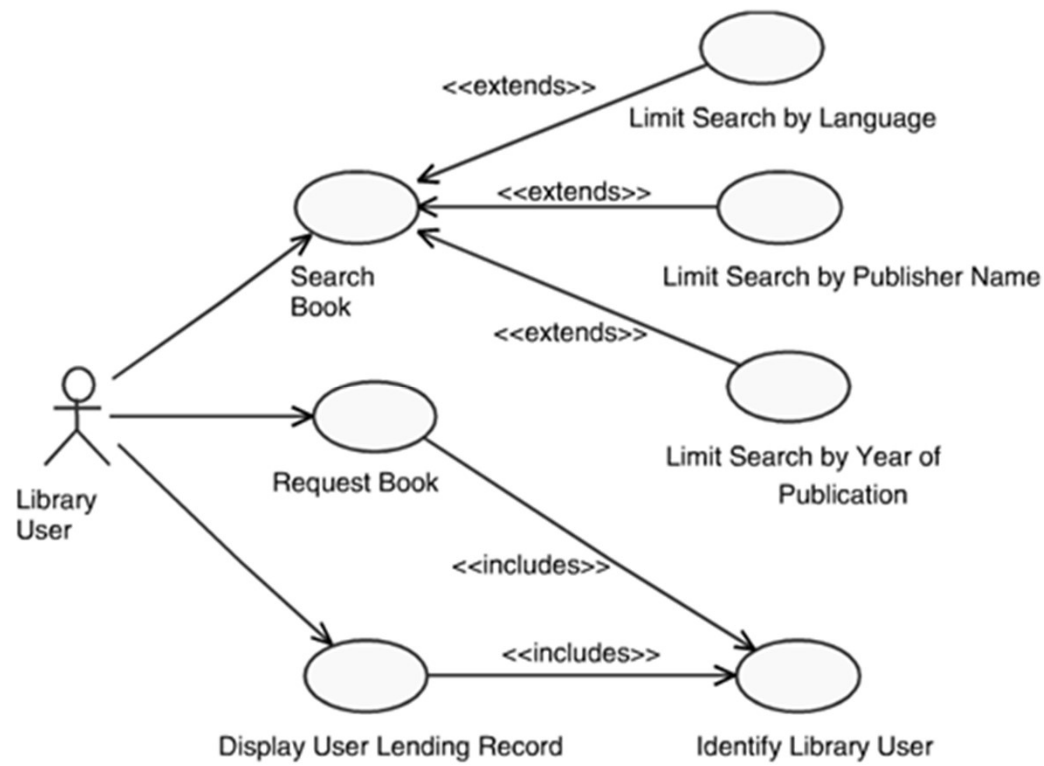
- Four benefits:
  - Short summary of system goals
  - Main success scenario (system responsibility)
  - Extension conditions (things to watch for or consider)
  - Extension handling (decisions on policy)
- From a presentation by Alistair Cockburn
  - Agile Use Cases
  - http://alistair.cockburn.us/get/2231



- Use Cases contain scenarios
  - A complete path through a use case from the first step to the last is called a **scenario** 27
  - Some use cases have multiple scenarios but a single user goal
    - All paths try to achieve victory

UML Use Cases – Best Practices

- Always design use cases from the actor's point of view
- Model the entire flow of a given operation
- For most systems, use cases should number in the tens, not hundreds
- <include> cases: not optional, base use case not complete without it, not conditional, and doesn't change the base use case behavior
- <extend> cases: Can be optional, not part of base use case, can be conditional or change behavior



WAVE Test for Use Cases (from Maksimchuk)
W: Use case describes WHAT to do, not how
A: ACTOR'S point of view
V: Has VALUE for actor
E: Use case models ENTIRE scenario

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 after use case creation: for instance, create an activity diagram for a given use case scenario
- For each activity in the diagram, (you might) follow-on and 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
  - Remember – sequence diagrams may not needed for simple logic

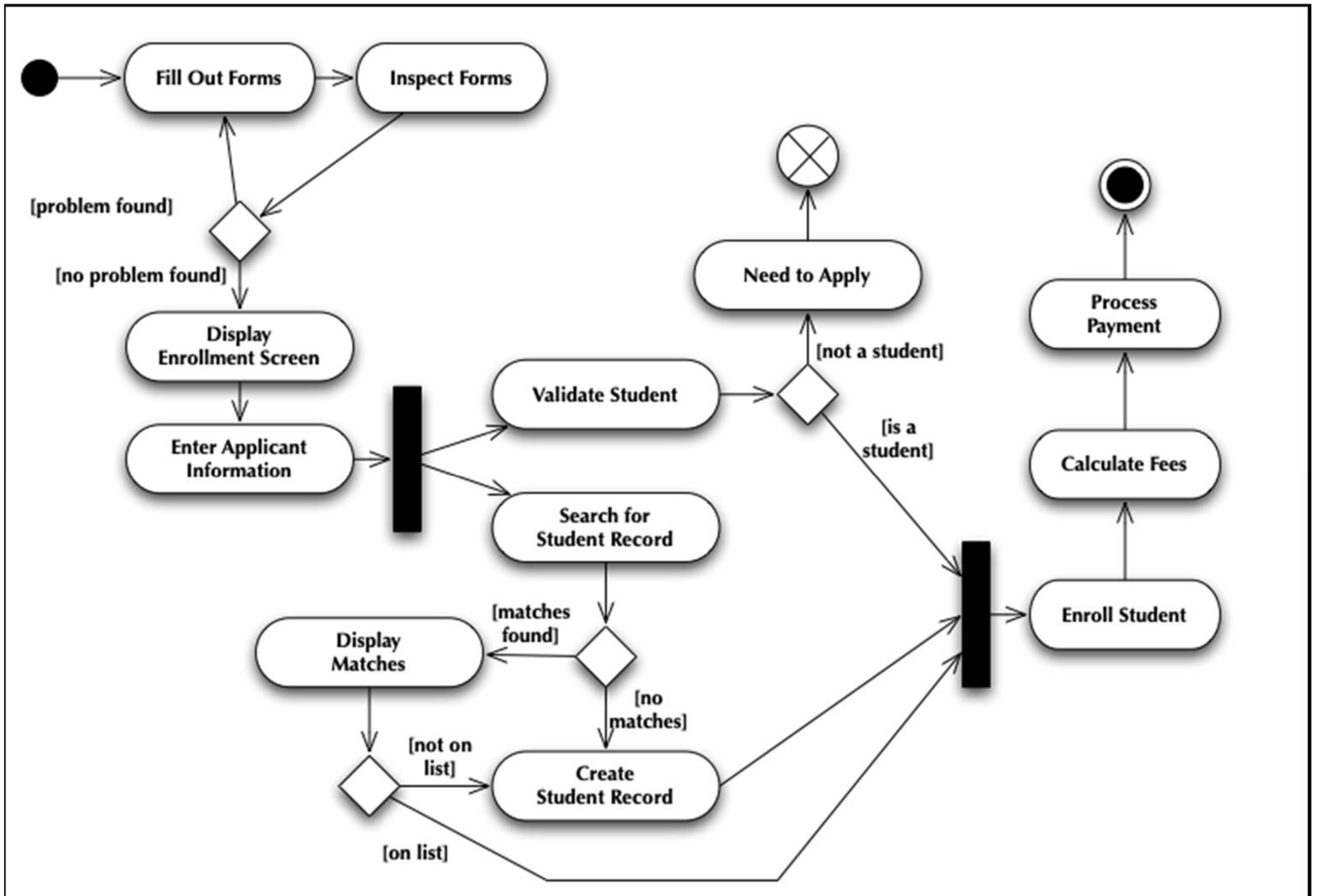
What are Activity & State Diagrams?

- Activity Diagram
  - Think “Flow Chart on Steroids”
  - Able to model complex, parallel processes with multiple ending conditions
- State Diagram
  - Shows the major states of an object or system
  - partition an object’s behavior into various categories (initializing, acquiring info, performing calcs, …)
  - documents these states and the transitions between them (transitions typically map to method calls)

Activity Diagrams

Notation

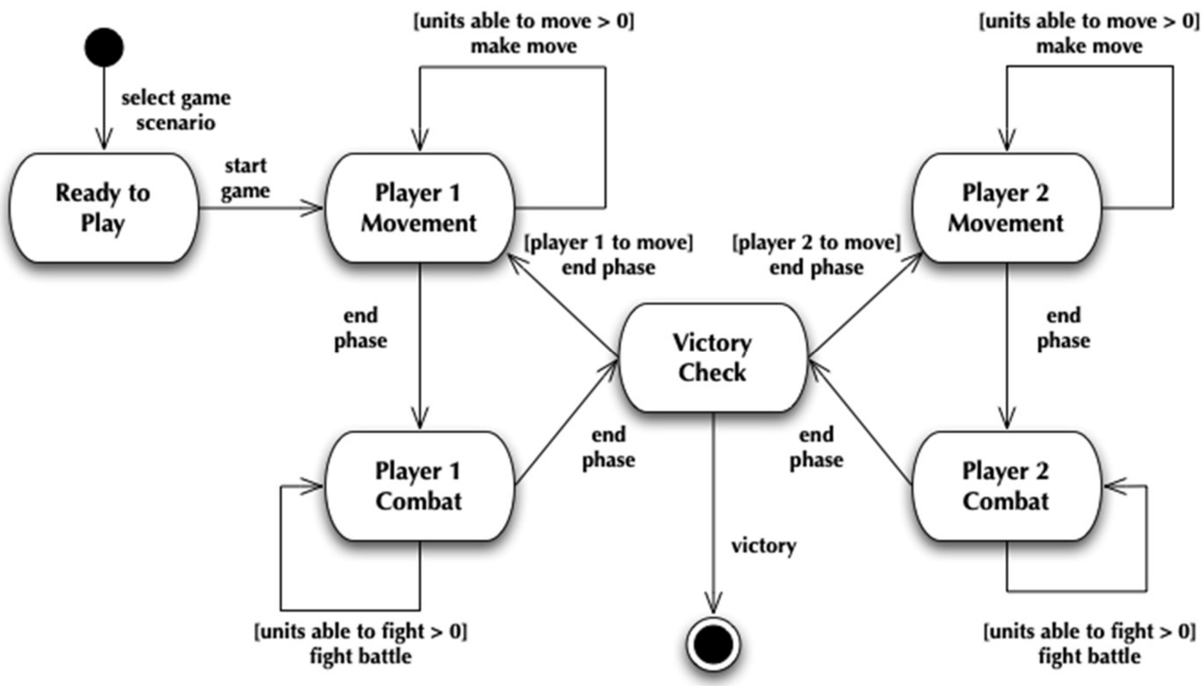
- Initial Node (circle)/Final Node (circle in circle)/Early Termination Node (circle with x through it)
- Activity: Rounded Rectangle indication an action of some sort either by a system or by a user
- Flow: directed lines between activities and/or other constructs. Flows can be annotated with guards "[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.



Example adapted from <http://www.agilemodeling.com/artifacts/activityDiagram.htm>. Copyright © 2003-2006 Scott W. Ambler

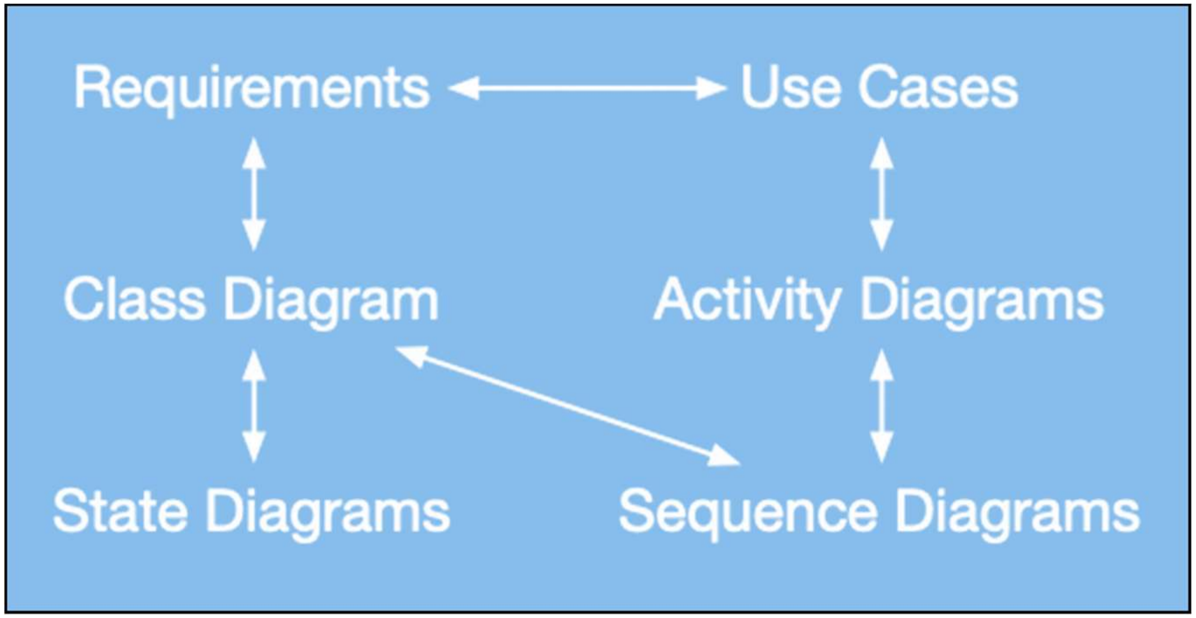
State Diagrams

- 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



Iterative Development Process

- Once you have written requirements and use cases to fulfill them
  - and you've reviewed 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
  - Details are developed in iterative change and review
- Relationships between OO A&D Software Artifacts



Next Steps

- Optional additional material
  - Dr. Anderson's lecture on the Problem Domain and these UML elements
    - You can find it on the class Canvas site under Media Gallery starting in mid-lecture 6
    - Again, very similar material as I'm using versions of his slides…
- This week
  - Wednesday 2/6 – Recitation session with Manjunath (optional)
  - Reading for Friday – Chapter 5 of Textbook
  - Friday 2/8 – Lecture – Intro to Design Patterns!
- Things that are due
  - Quiz 2 is up on Canvas, will be due before Wednesday recitation
  - Graduate Presentation topic e-mail is due Friday 2/8/19 11 AM
    - Topic list so far is posted on-line in Canvas under Files
  - Homework 2 is due 2/15 at 11 AM
  - Class Semester Project topic e-mail is due Friday 3/1/19 11 AM