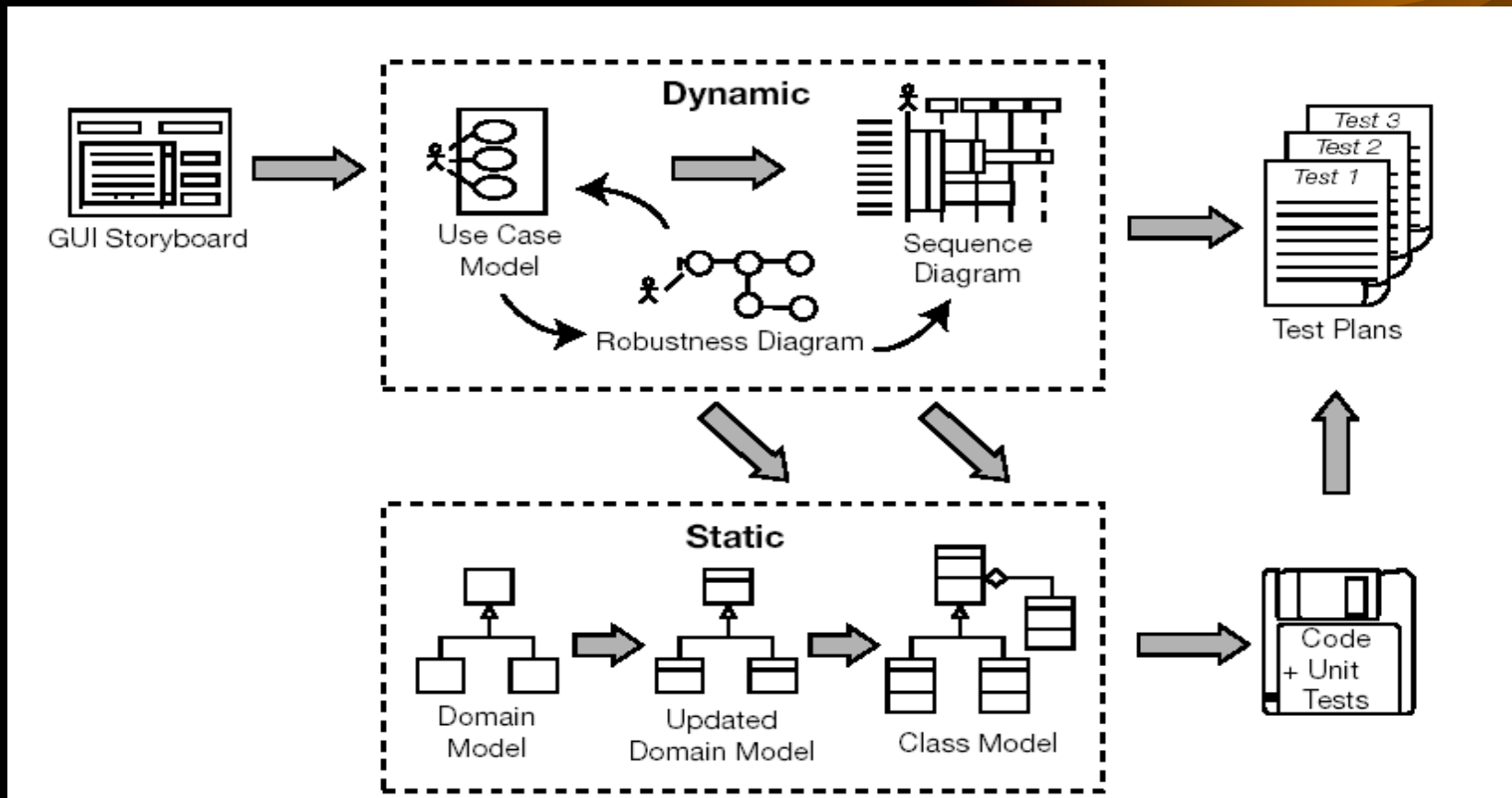
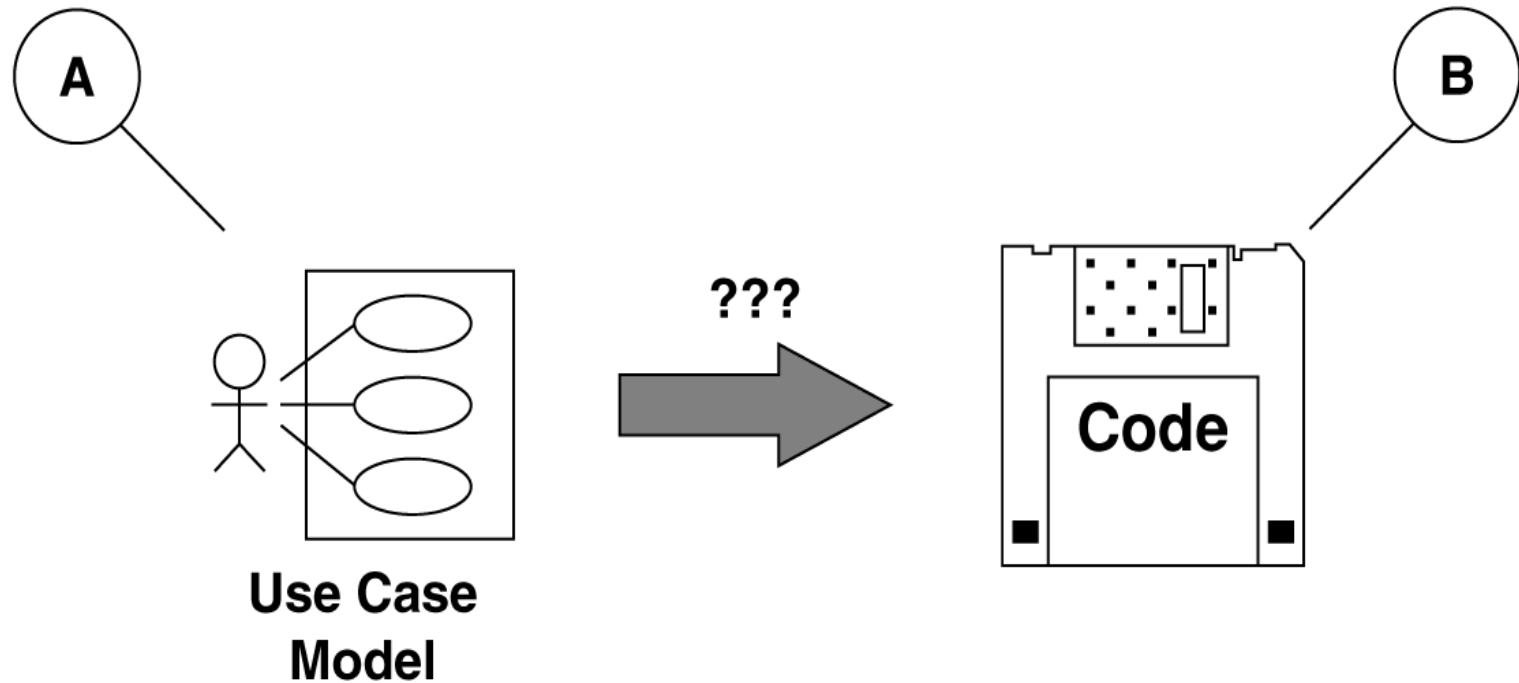


ICONIX Process: Use Case Driven Object Modeling



The goal. Driving a good O-O software design from use cases.



How do we get from use cases to code?

ICONIX Process: Use Case Driven Object Modeling



- Introduction
- The 10,000 foot view
- *ICONIX Process Roadmap*
- The 1000 foot view
- Summary

Introduction



- The difference between Theory and Practice
- Disambiguation – the key to use case driven development
- Getting from use cases to code
- The ICONIX UML core subset

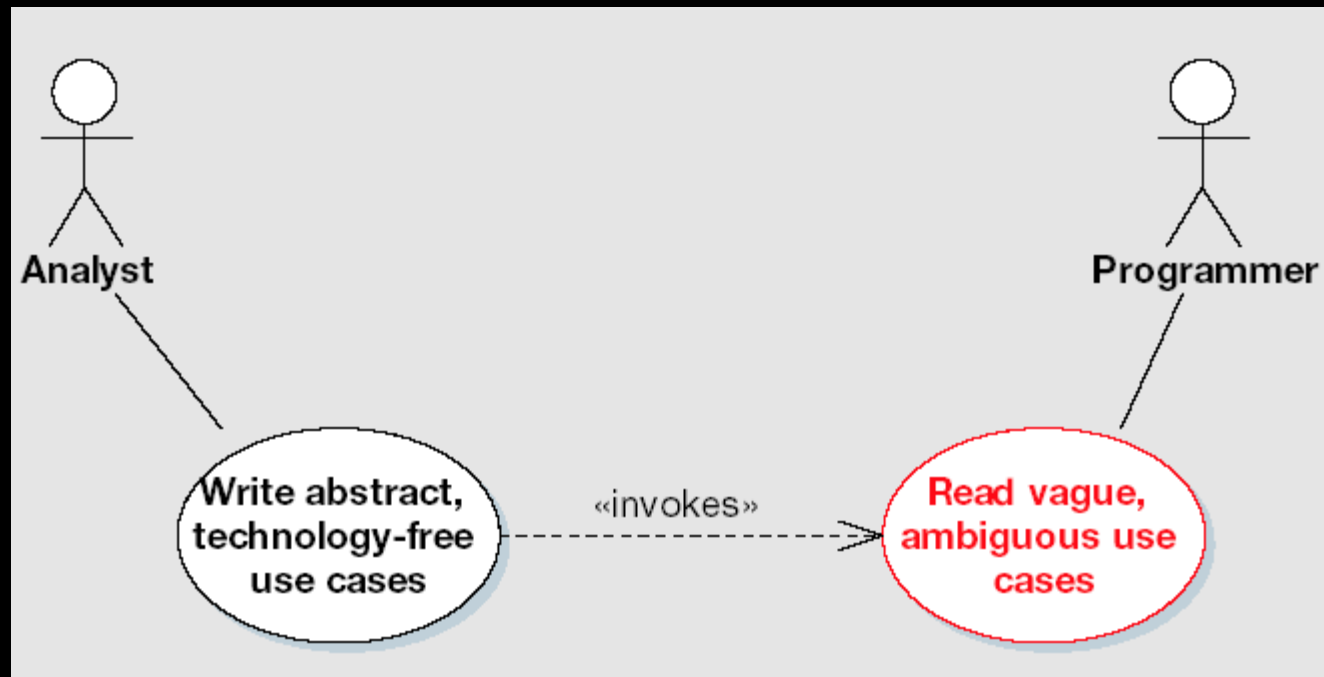
The difference between Theory and Practice...



- In theory, there is no difference between theory and practice. In practice there is.
- In practice, UML is TOO BIG.
- In practice, there's never enough time for modeling.
- ICONIX Process is a streamlined approach that helps you get from use cases to code quickly, using a UML core subset.

Disambiguation...the key to use case driven development

In theory, abstract use cases are good. In practice, they're vague and ambiguous.

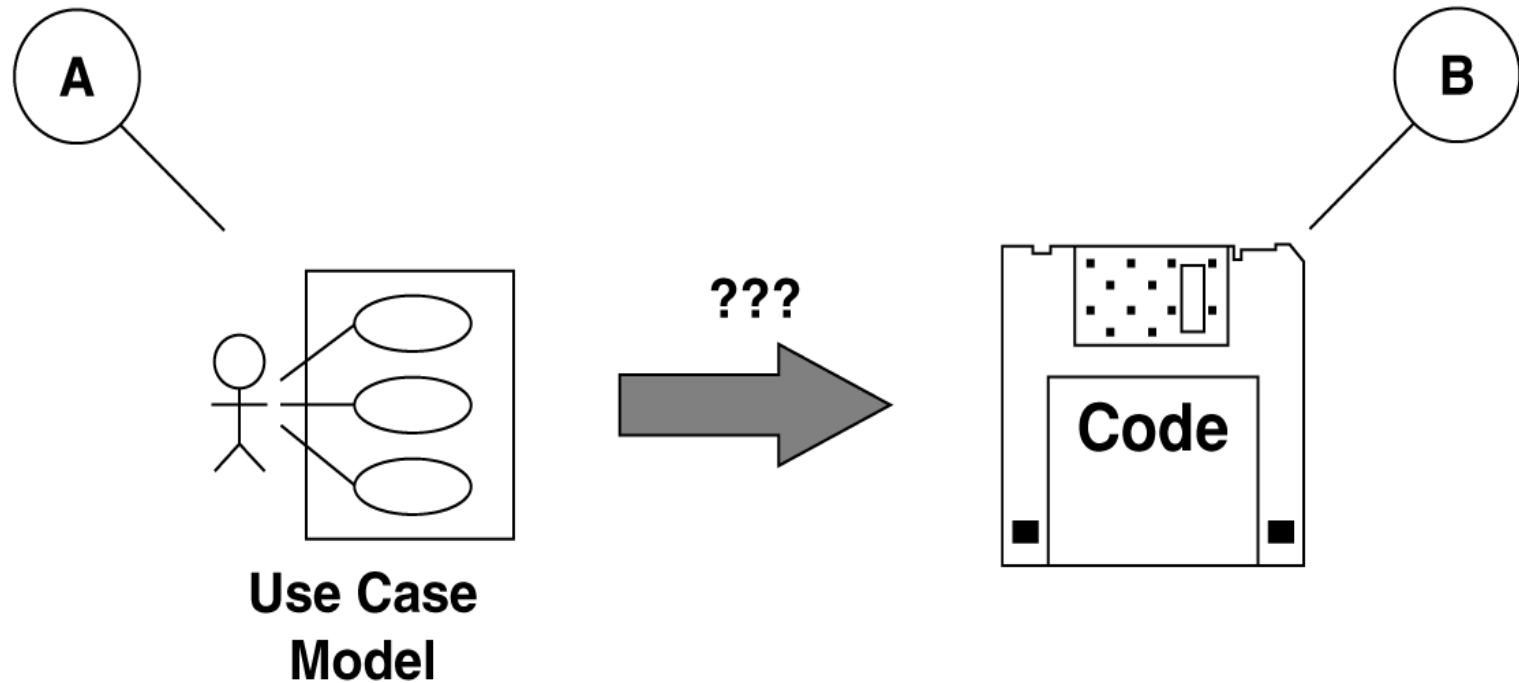


Key features of ICONIX Process



- Avoids analysis paralysis
- Core subset streamlines use of UML
- High degree of traceability

We'd like to get from Point A to Point B, as quickly as possible



How do we get from use cases to code?

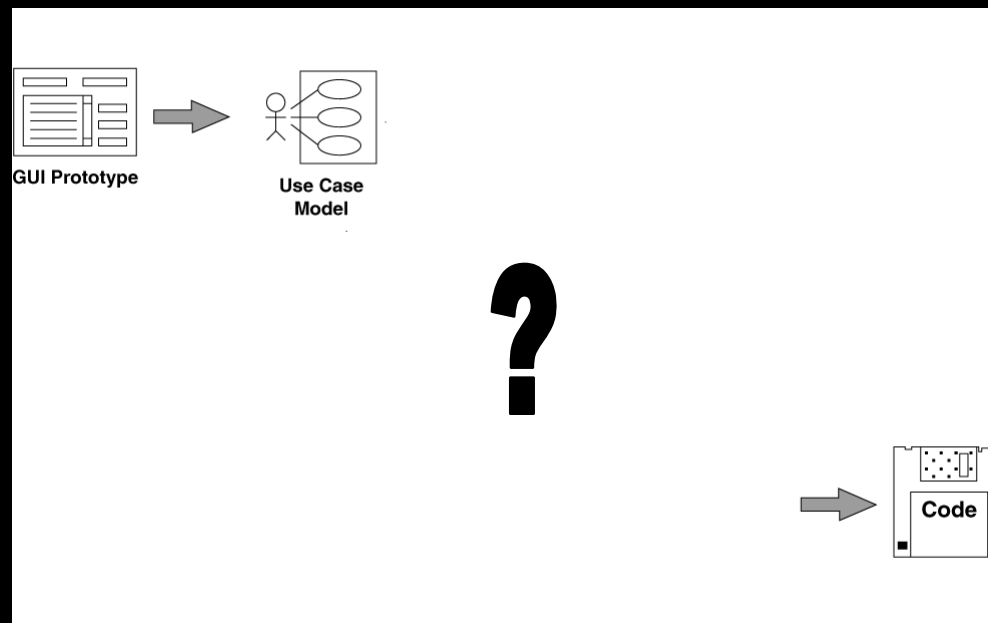
Defining a UML core subset

- The ICONIX Core Subset helps us to **avoid analysis paralysis**.
- We'll work backwards from code to determine which parts of the UML we really need.
- Then we'll **leave everything else out**.
- But feel free to use any other UML diagrams that you might have a specific need for.
- **Less is more**. 4 diagrams are better than 14.

Working backwards from code

Let's assume that we've done a little prototyping, and started to write some use cases.

But code is our desired destination.



OOAD, oversimplified



- 2 fundamental questions.
- **OBJECT DISCOVERY**: What are all the objects?
- **BEHAVIOR ALLOCATION**: How are the functions mapped across the objects?
- When we know what **classes** we need, what the **software functions** are, and have **mapped the functions onto the classes**, we're a long way towards understanding the design.

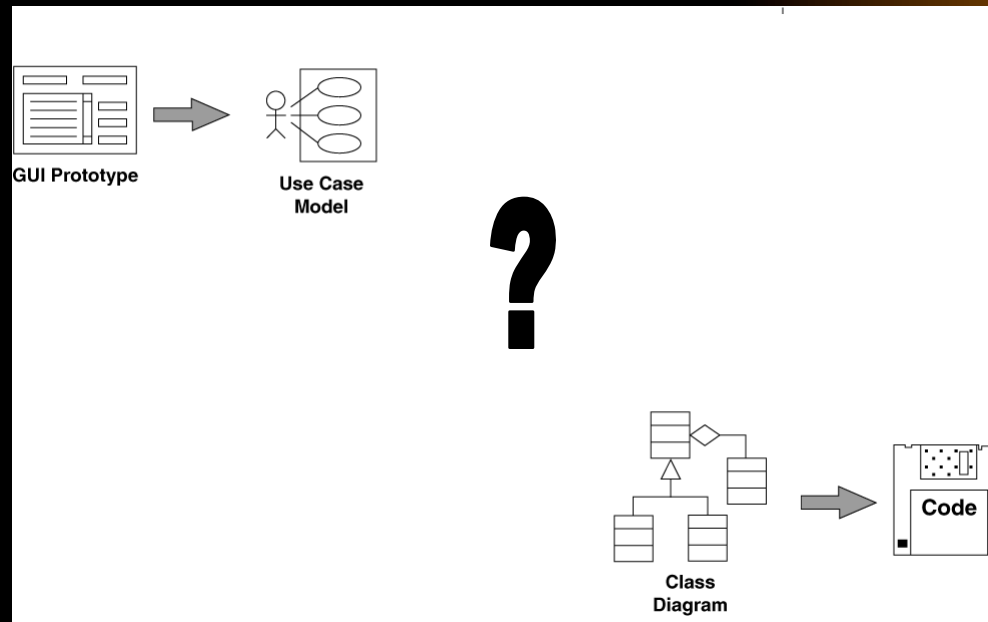
Before we get to code, though...



We need a complete set of classes, with accompanying attributes and methods.


We show this information on design-level class diagrams.

Design-Level Class Diagrams



Our design-level class diagrams define the structure of our code.

Before we have classes with attributes and methods, though...

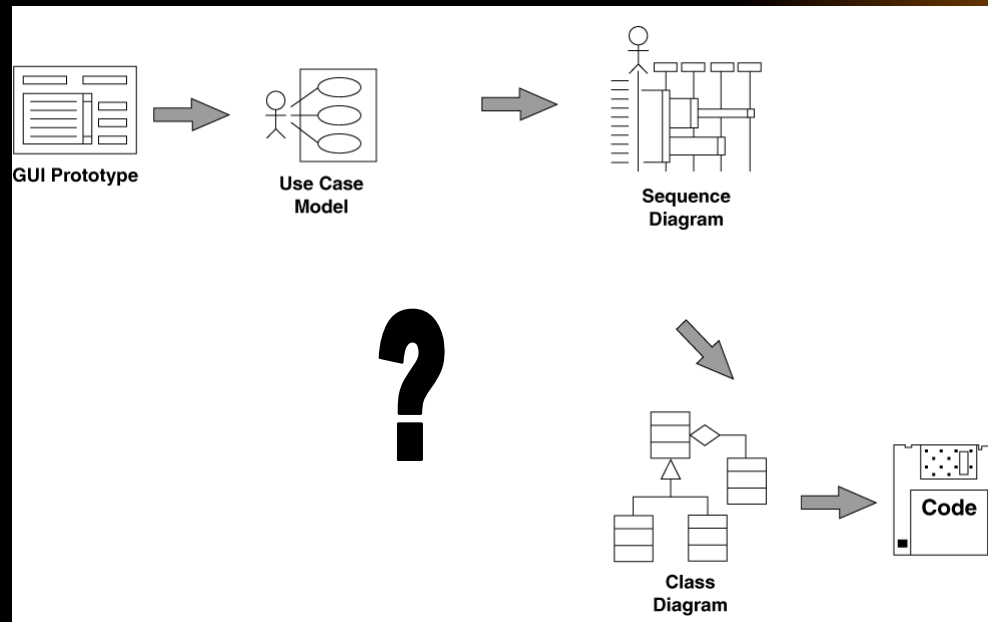


BEHAVIOR ALLOCATION:

We make decisions about **which classes are responsible for which methods** while we are drawing sequence diagrams.

So, we need to draw a sequence diagram for each use case.

Sequence Diagrams



We **allocate methods to classes** as we draw sequence diagrams.

Before we do sequence diagrams, though...

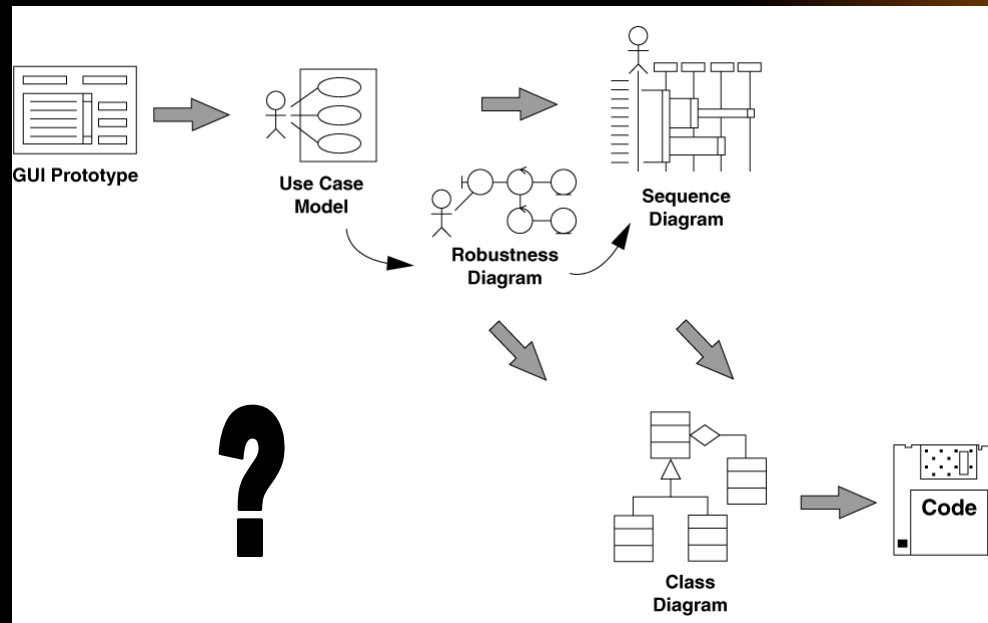


We need **unambiguous** use case text.

We need to have a good idea about **which** objects will be participating in each use case, and **what** functions the system will perform as a result of user actions.

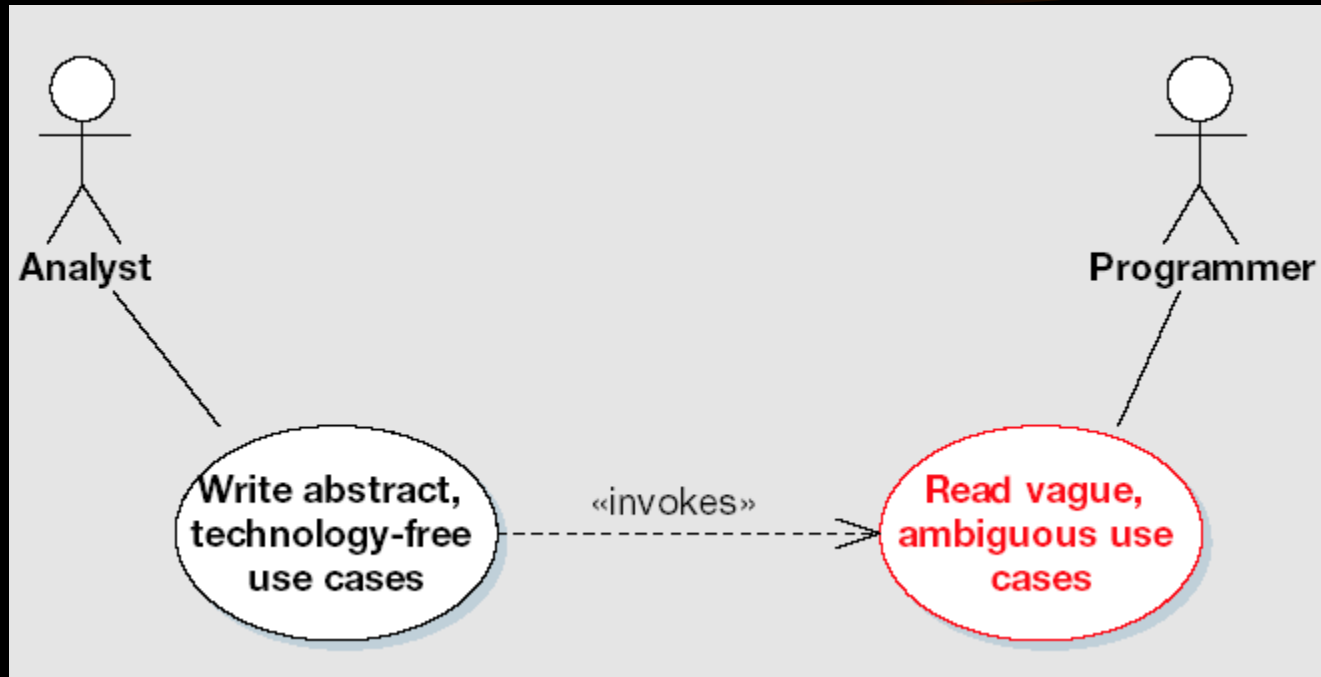
We identify participating objects, and the software functions that we need for a use case, during robustness analysis.

Robustness Diagrams



OBJECT DISCOVERY: We discover new objects, identify functions, and add attributes to classes, as we draw robustness diagrams. We also disambiguate the use cases.

Abstract, technology-free use cases are vague, ambiguous, incomplete



Robustness diagrams help us disambiguate the use cases

DISAMBIGUATION: We describe system usage *in the context of the object model*.

This means that *we don't write abstract, vague, ambiguous use cases* that we can't design from.

Instead, we need to write use case text that **references the names of objects in the problem domain**.

We also **reference the names of "boundary objects"** (screens) explicitly in the use case text.

First, though...

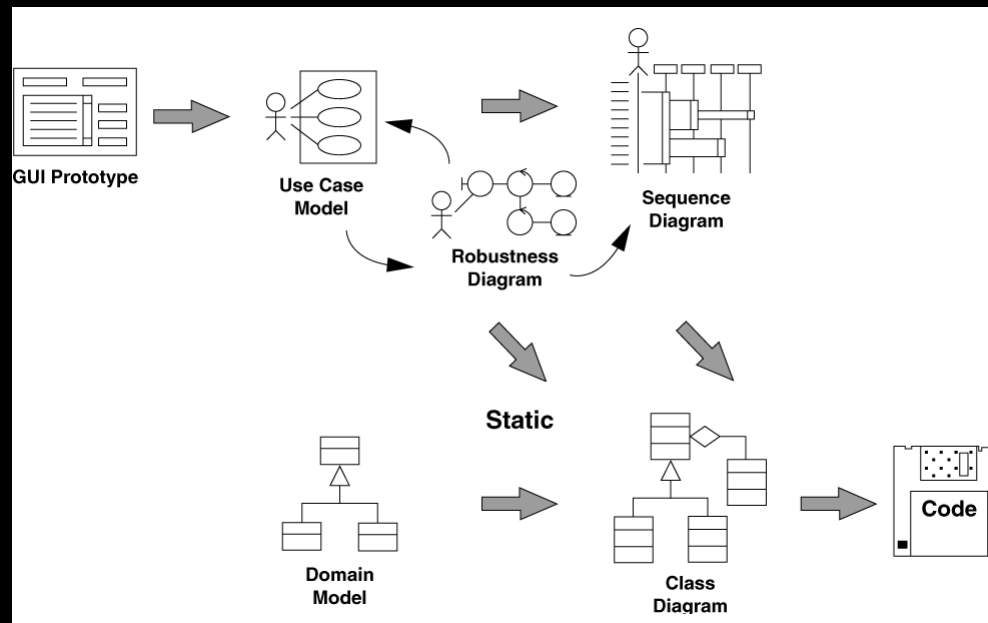


We need to identify the main abstractions that are present in the problem domain.

In other words, we need a domain model.

We show our domain model on class diagrams.

Domain Model



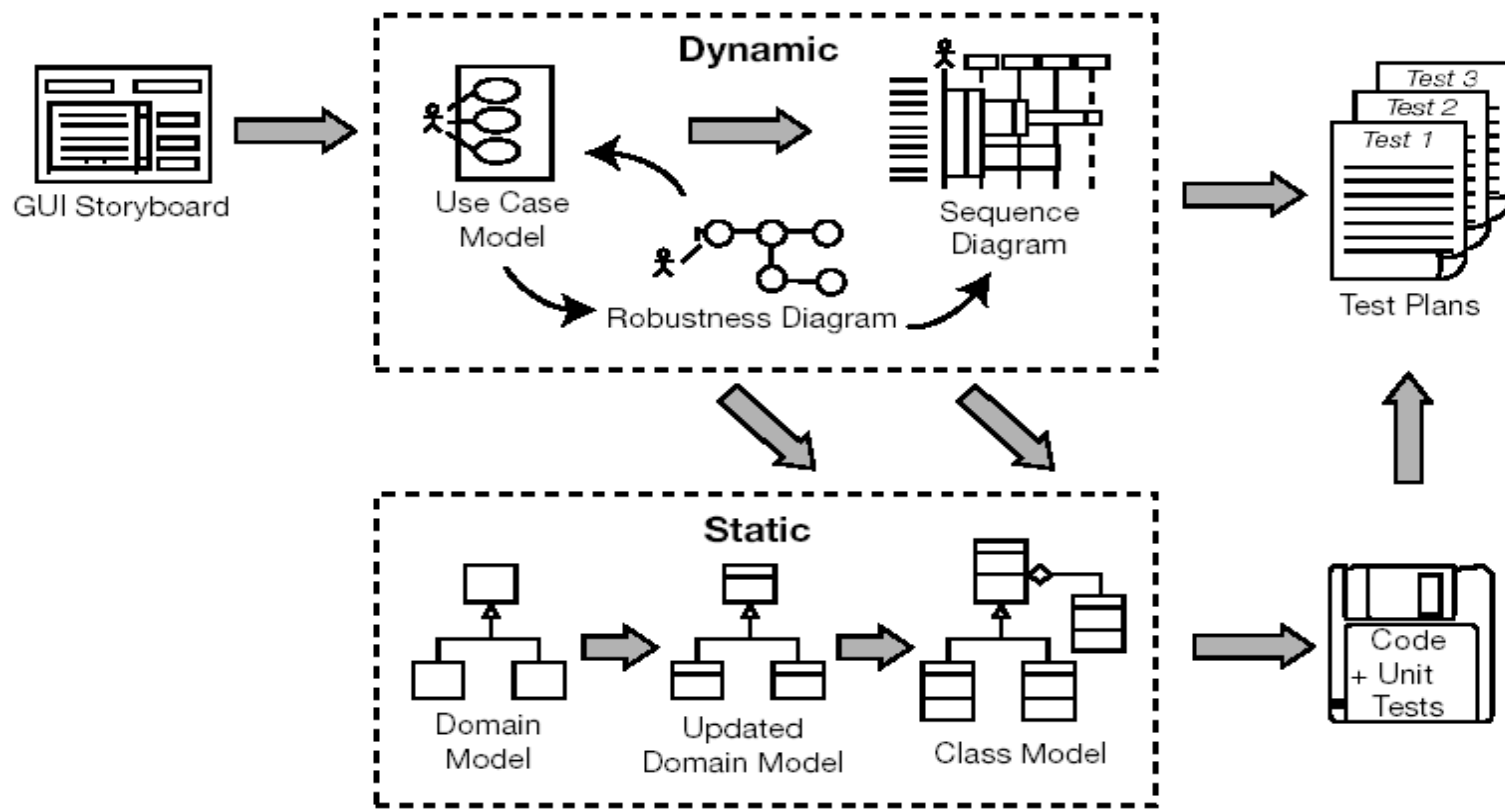
Refining our class diagrams



We'll **continuously refine** our analysis level class diagrams (our domain model) as we **explore the behavior** of the system in more and more detail during analysis and design.

This will ultimately result in our design-level class diagrams, which we can code from.

The ICONIX UML Core Subset



Break



- Good news: we don't have to cover the other 10 UML diagram types (state diagrams, activity diagrams, communication diagrams, component diagrams, deployment diagrams, timing diagrams....) before break
- Better news: we don't have to cover them after break, either

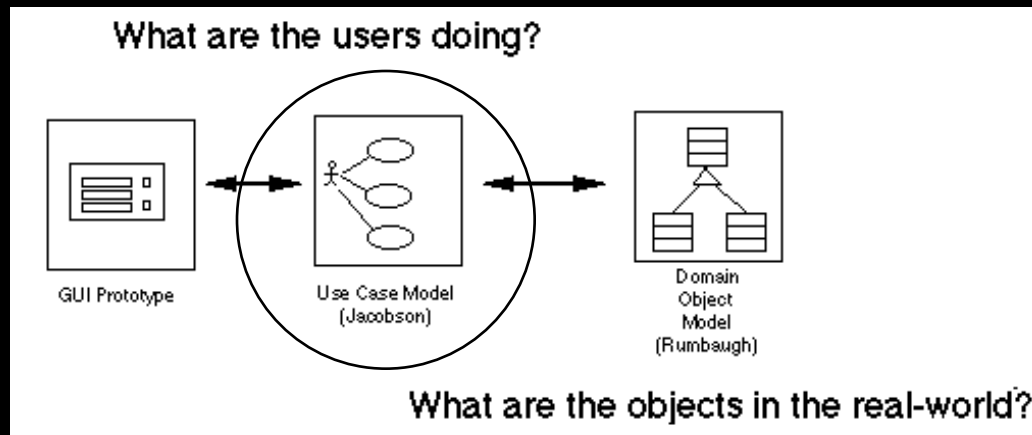
The 10,000 foot view



- *Each diagram answers a question.*
- Use Cases: **What are the users doing?**
- Domain Models: **What are the objects in the real world?**
- Robustness Diagrams: **What objects participate in each use case?**
- Sequence Diagrams: **How do the objects collaborate with each other?**

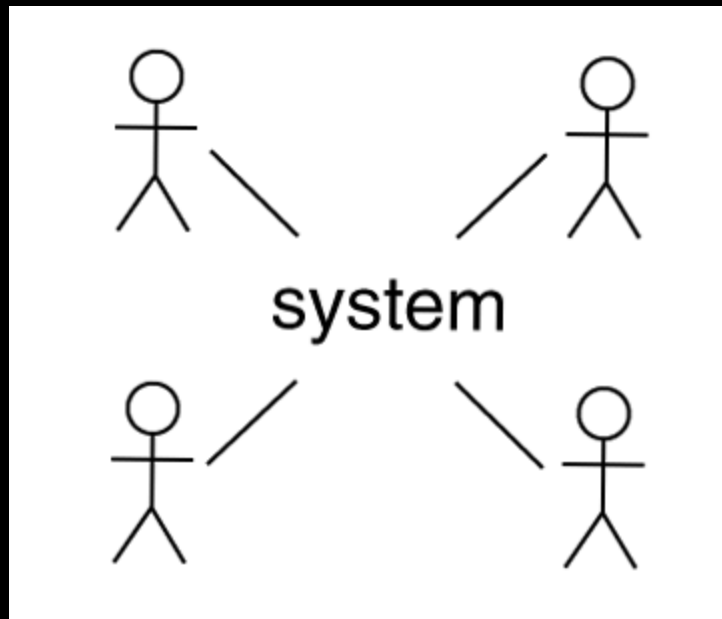
Use Cases: What are the users doing?

Use case driven means that user requirements are “king.” The use cases drive everything else within the approach.



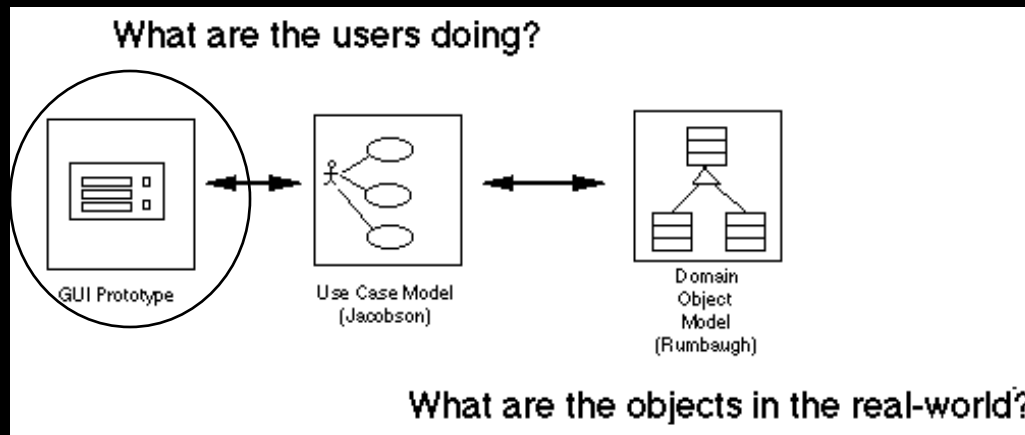
Working from the outside in

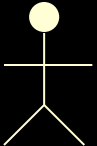

“Outside in” means working from the user requirements inward.



Elements of use case modeling

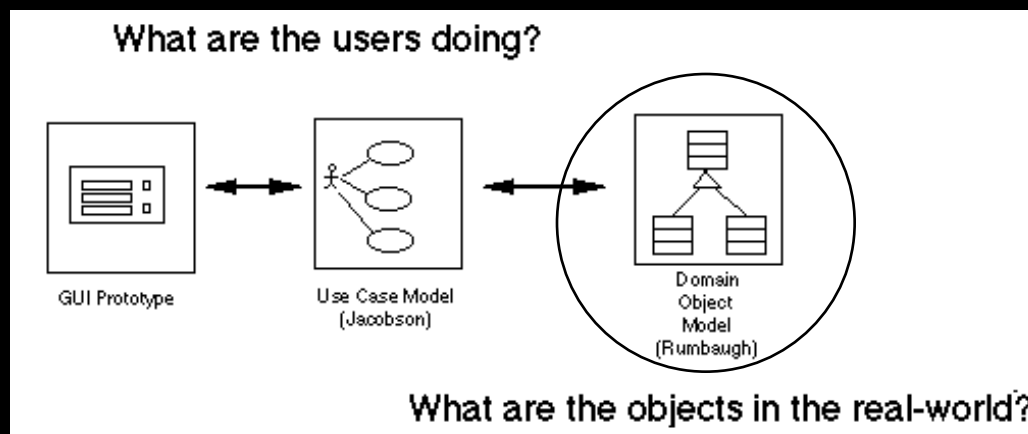
- Use *storyboards* to help define the use cases.



- Actors  and Use Cases  Name
- Basic and Alternate Courses of Action

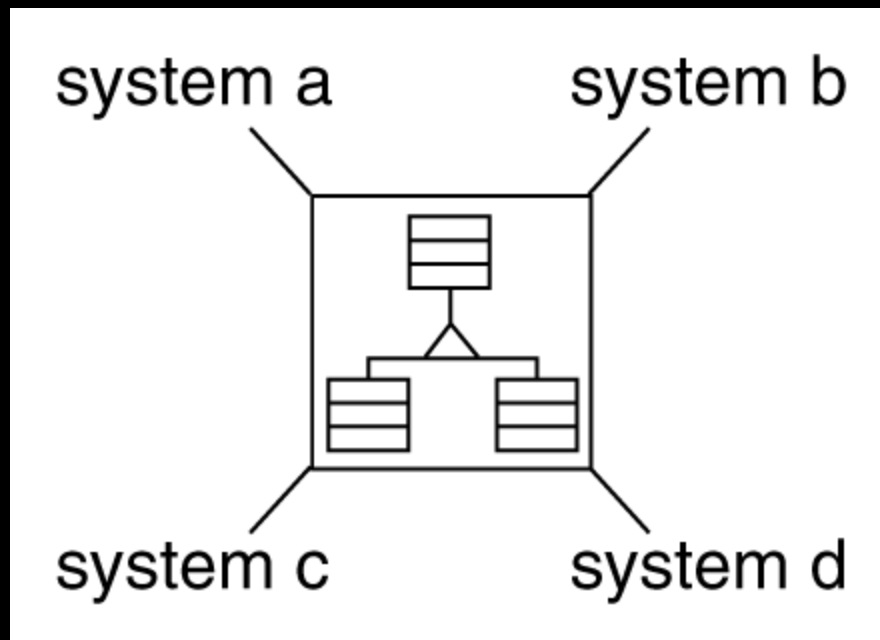
Domain Models: What are the objects in the real world?

- Identify real-world (problem domain) objects
- Identify “is” and “has” relationships
- This is where reuse comes from



Working from the inside out

“Inside out” means working from the domain objects outward.



Identify relationships between objects

- Associations



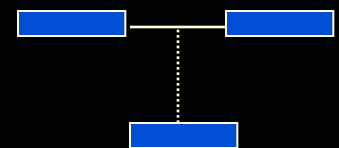
- Generalizations



- Aggregations

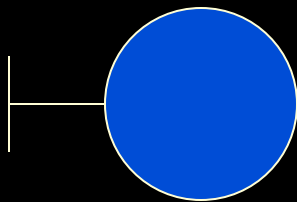


- Associations as classes (link classes)

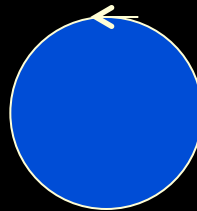


Robustness Analysis: What objects participate in each use case?

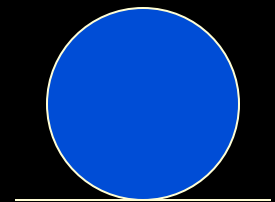
- Draw an “object picture” of the use case
- Use the **boundary/control/entity** stereotypes



Boundary class



Control class
(Controller)



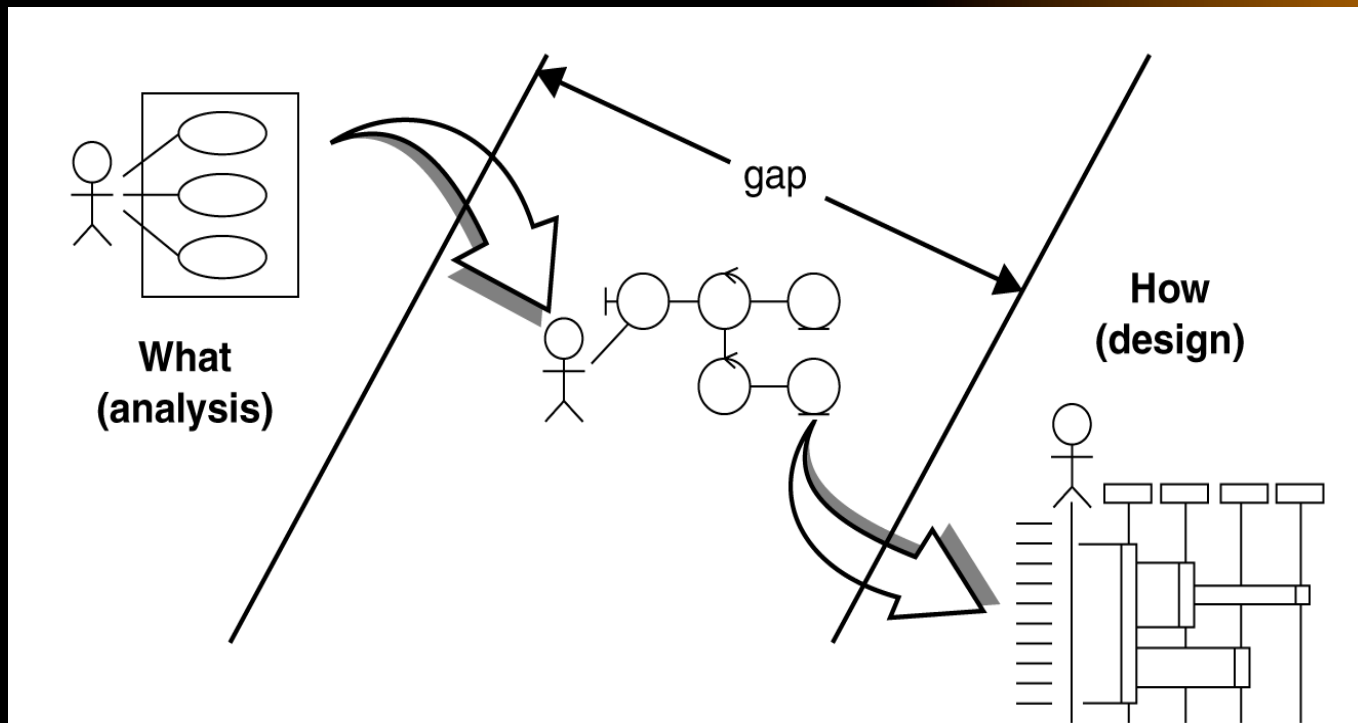
Entity class

Robustness Analysis



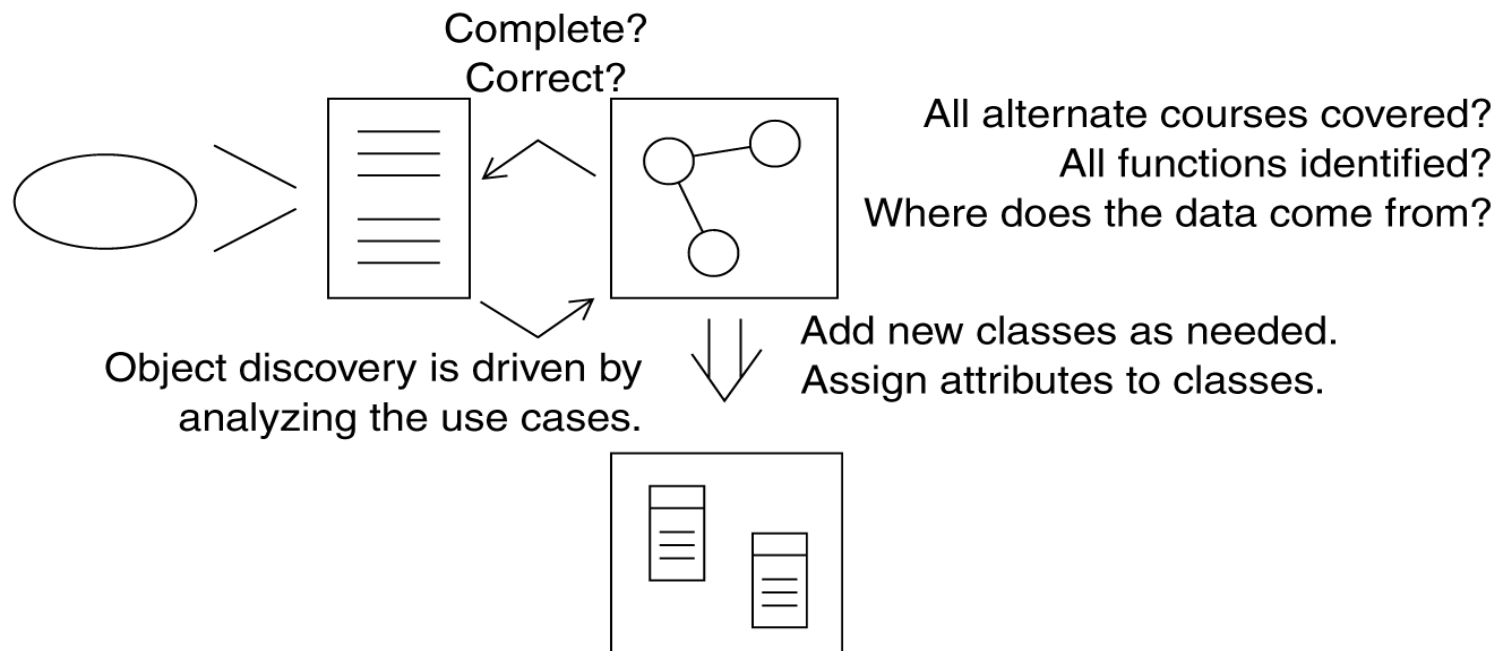
- Closes **gap** between “**what**” and “**how**”
- **Disambiguation + Object Discovery**
- Do a robustness diagram for each use case
- Client/server, N-tier, MVC

Closing the gap between “what” (analysis) and “how” (design)



Mind the gap: Many projects fail while trying to cross this what/how gap.

Disambiguation + Object Discovery



The domain model evolves into a detailed static model.
This evolution is driven by working through the use cases.

Do a robustness diagram for each use-case



- Directly traceable to user-approved prototypes and courses of action
- System performance characteristics defined by interactions between distributed objects
- Technical architecture important at this level


Client/Server, N-Tier, MVC

- Boundary/control/entity can be used to describe client/server architectures
 - GUI
 - Logic
 - Repository
-
- Also N-tier and MVC architectures.



The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may have been corrupted. Restart your computer, and then open the file again. If the red x still appears, you may have to delete the image and then insert it again.

Sequence Diagrams: How do the objects collaborate with each other?



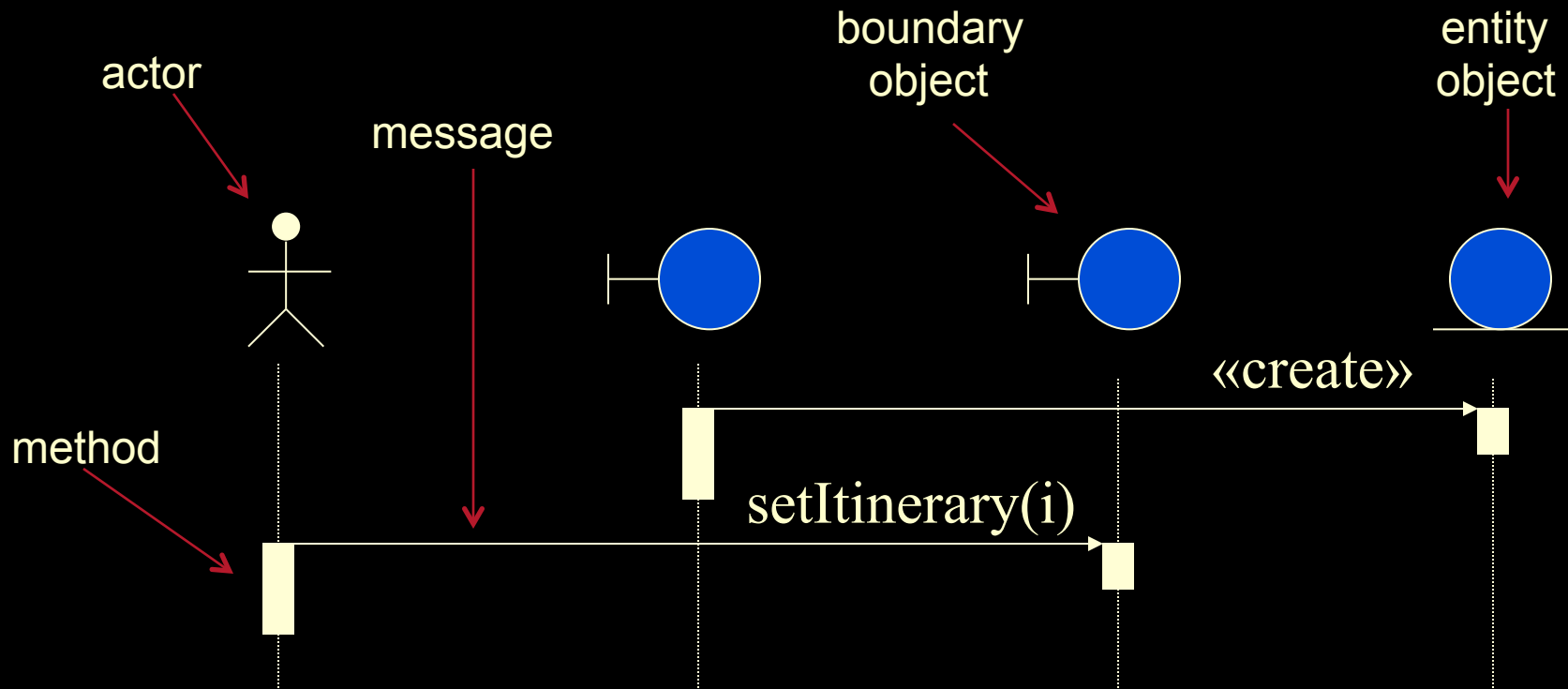
- Allocation of behavior to specific objects
- Message passing between objects
- Traceable back to the use case description

Behavior allocation is done on sequence diagrams

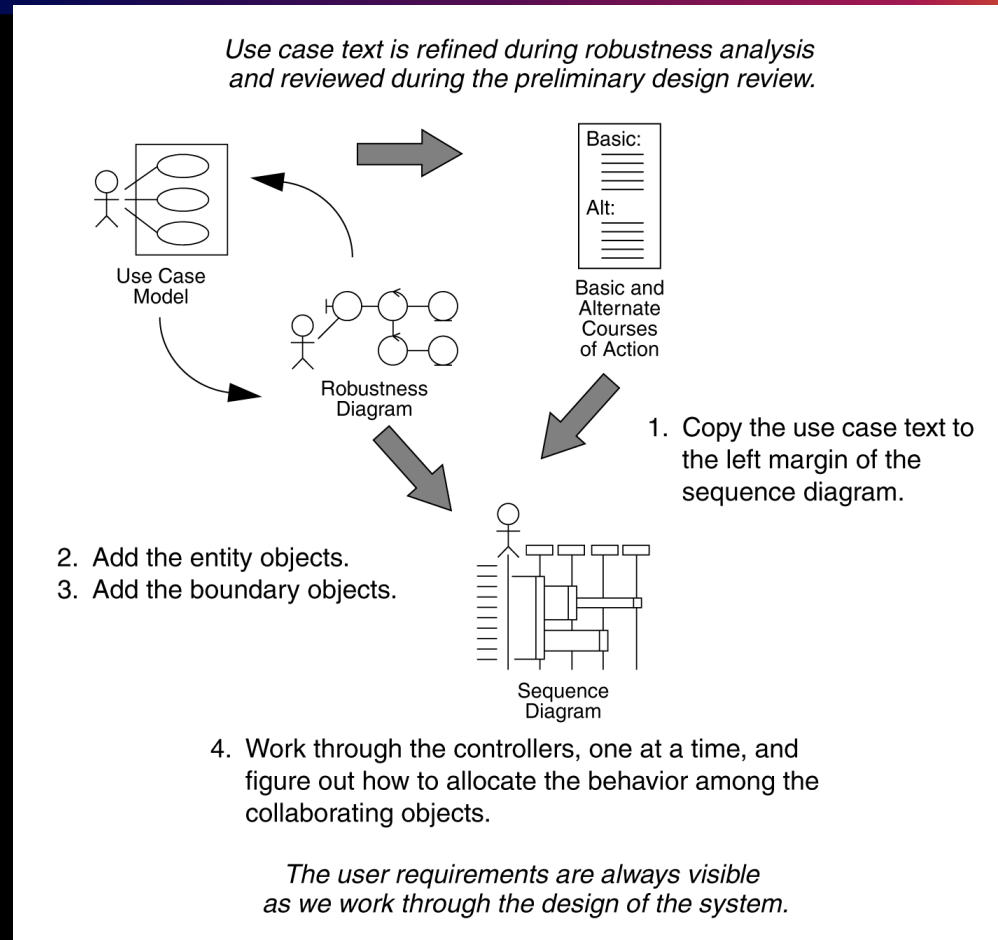


- Used to help allocate behavior among boundary, control, and entity objects--in other words, to answer the question: **Which class does an operation belong in?**
- Show detailed interactions between objects over time
- Object interactions follow the use case text

Sequence diagram notation



Building a sequence diagram is much easier when we've done robustness analysis



Break



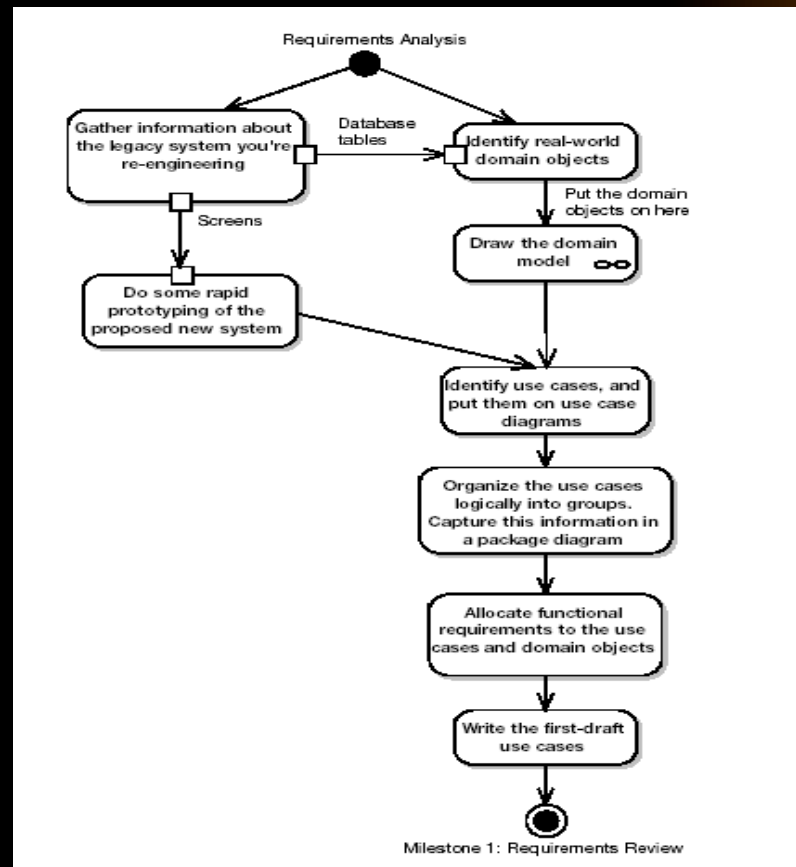
- Next, we'll introduce a step-by-step roadmap you can follow. The roadmap covers:
- Requirements Definition
- Analysis and Preliminary Design
- Detailed Design
- Implementation and testing

ICONIX Process Roadmap

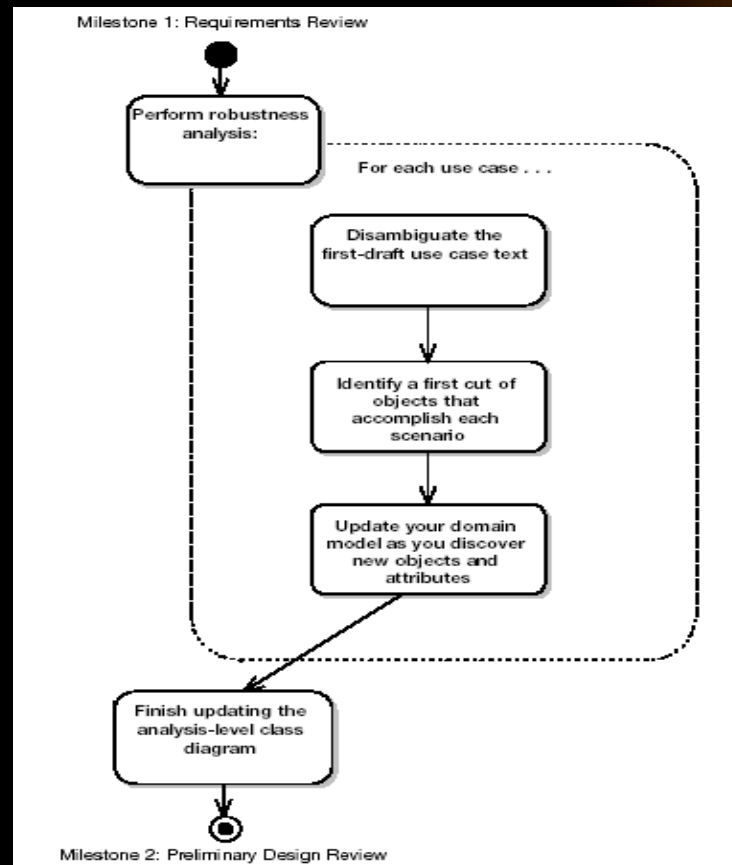


- Requirements Definition
- Analysis and Preliminary Design
- Detailed Design
- Implementation

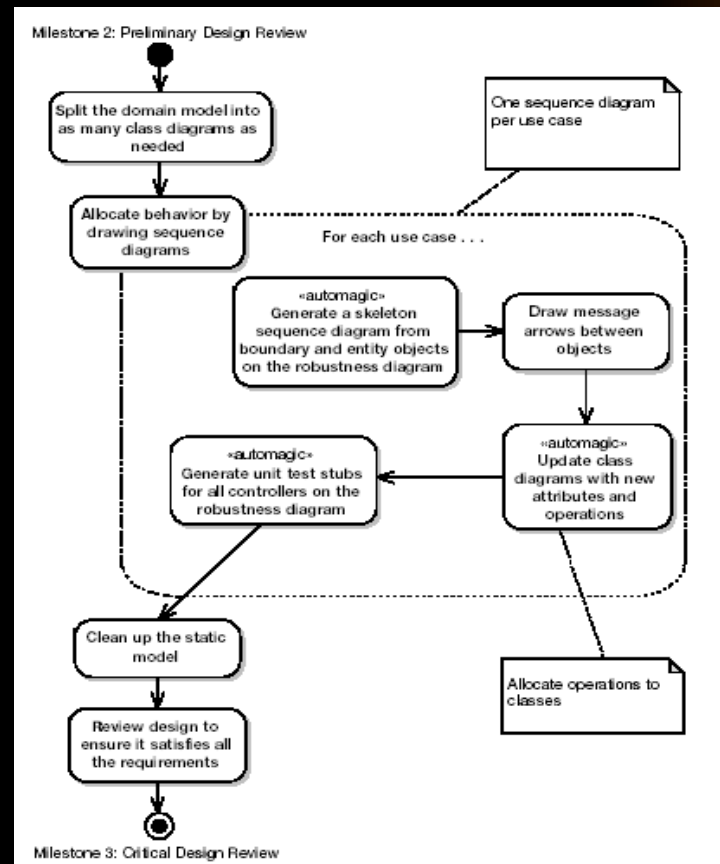
Requirements Definition (Collaborative Session + Lab 1)



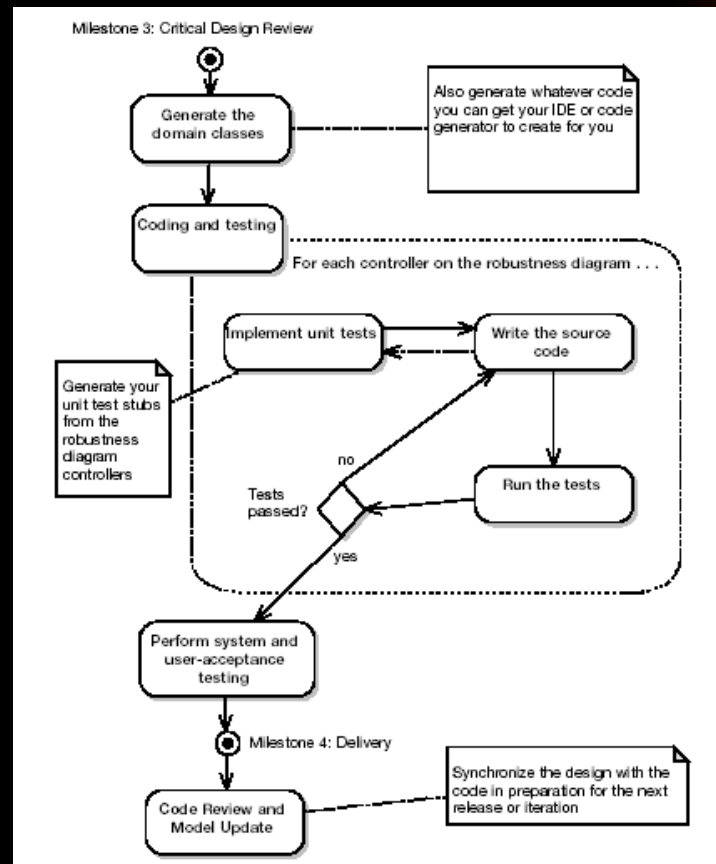
Analysis and Preliminary Design



Detailed Design



Implementation



The 1000 foot view



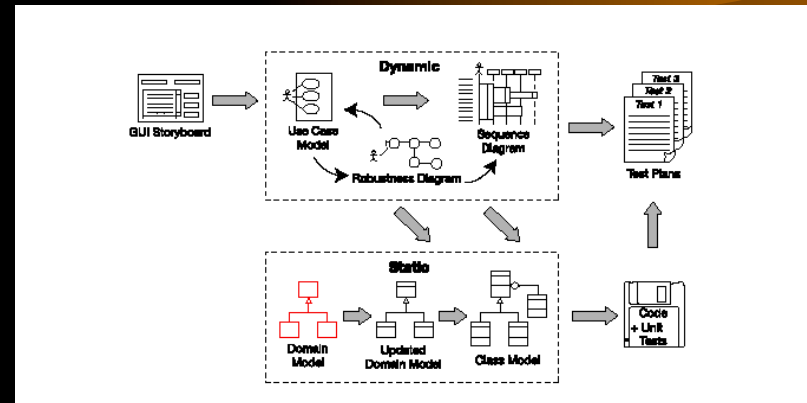
- Requirements Definition
- An in depth look at domain modeling
- An in depth look at use cases
- Requirements Review
- An in depth look at robustness analysis
- Preliminary Design Review
- An in depth look at sequence diagrams
- Critical Design Review
- Implementation

Requirements Definition – Top 10

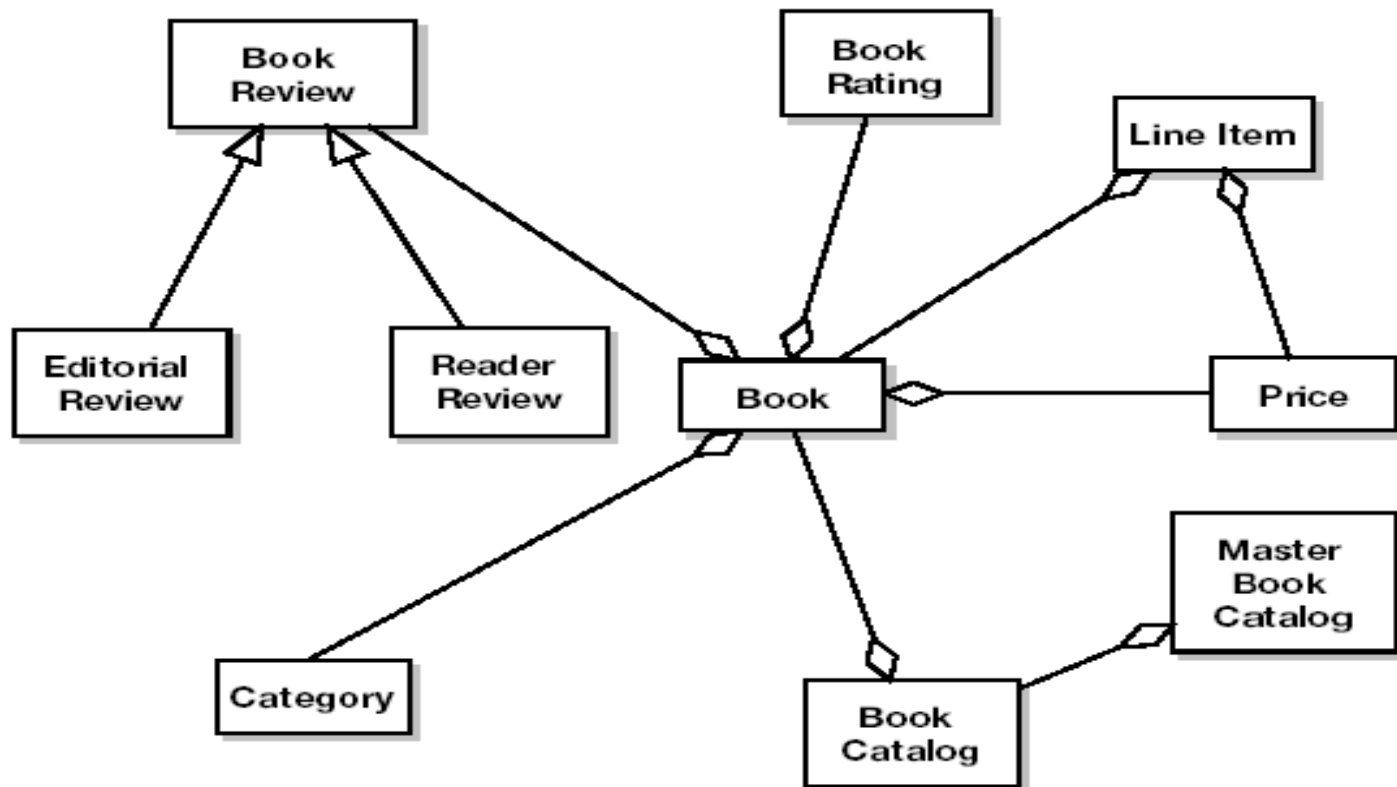
- 10. Use a modeling tool that supports **linkage and traceability** between requirements and use cases.
- 9. Link requirements to use cases by dragging and dropping.
- 8. **Avoid dysfunctional requirements** by separating functional details from your behavioral specification.
- 7. Write at least one test case for each requirement.
- 6. Treat requirements as first-class citizens in the model.
- 5. Distinguish between different types of requirements.
- 4. Avoid the “big monolithic document” syndrome.
- 3. Create estimates from the use case scenarios, not from the functional requirements.
- 2. Don't be afraid of examples when writing functional requirements.
- 1. Don't make your requirements a technical fashion statement.

An in depth look at domain modeling

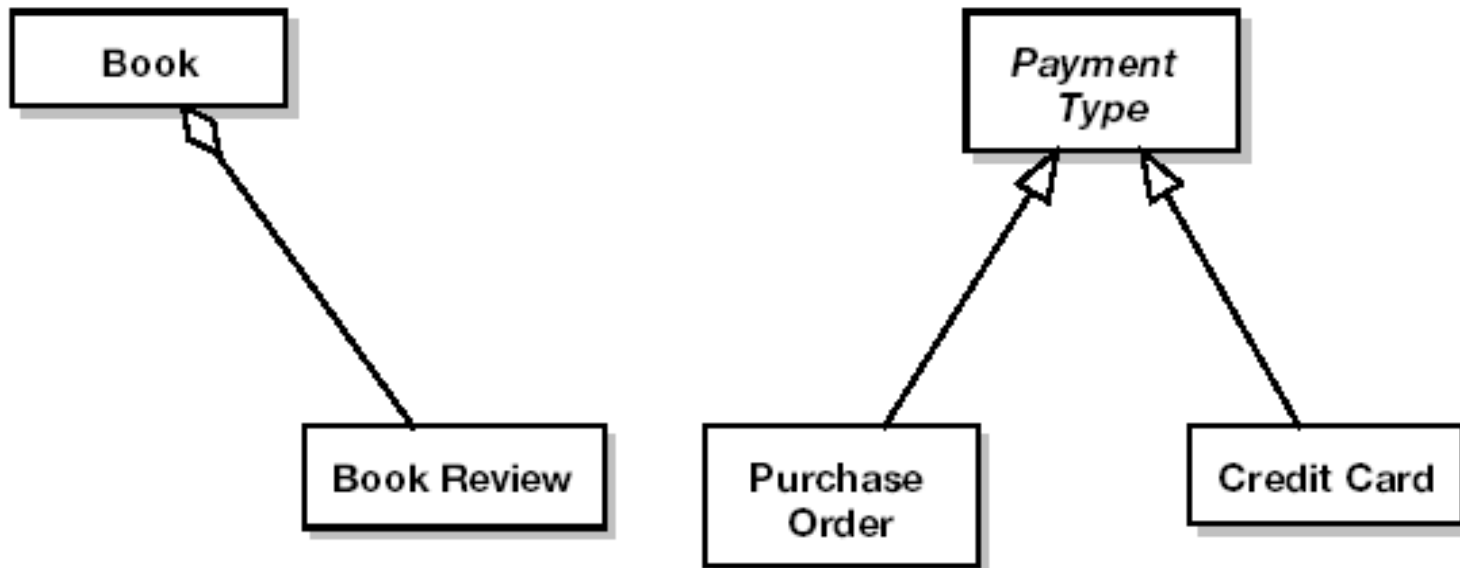
- An example domain model
- Aggregation and Generalization
- Finding domain objects by grammatical inspection.
- Domain classes aren't tables
- Domain Modeling Top 10



An example domain model



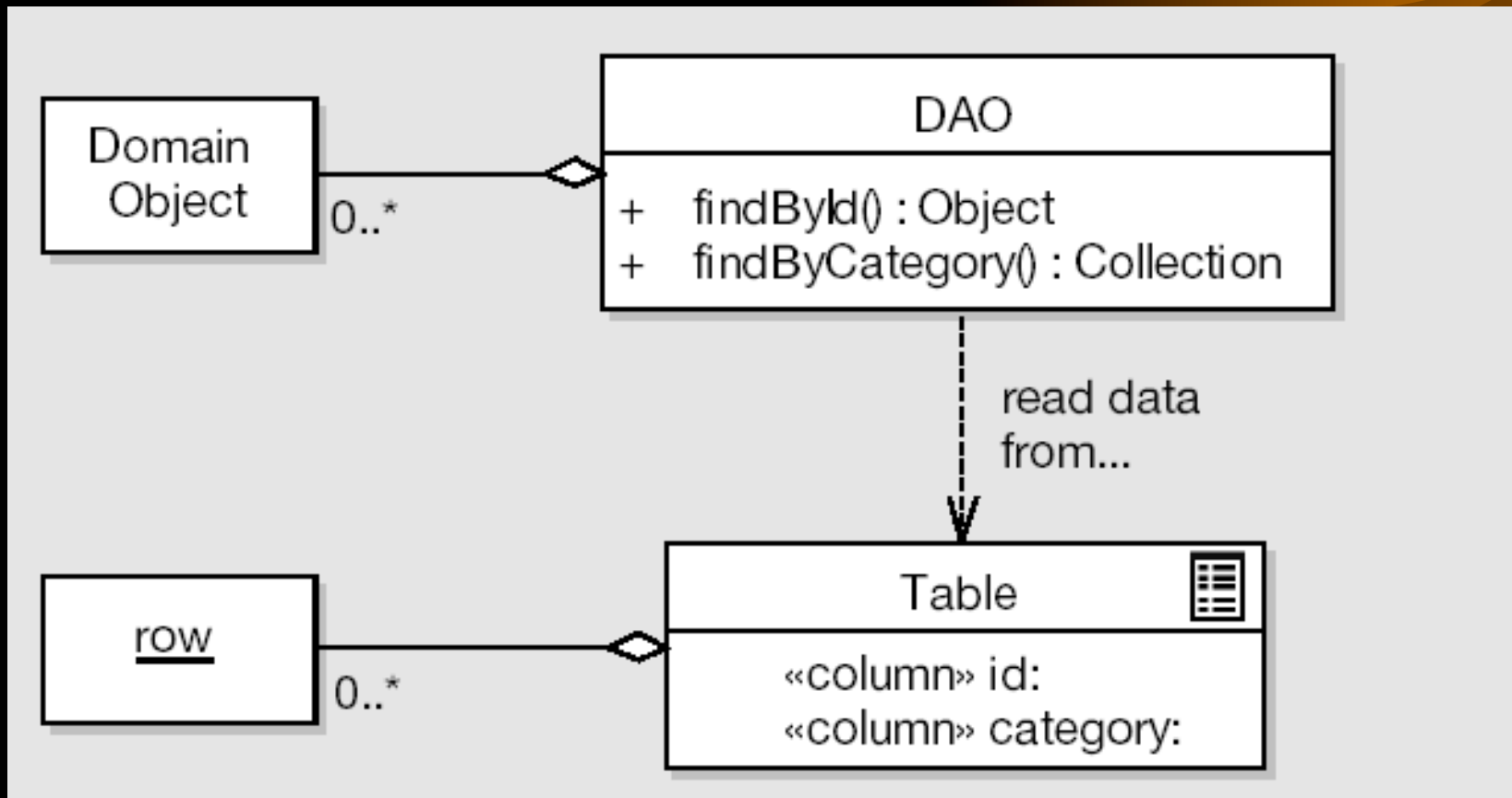
Aggregation (has) and Generalization (is)



Finding domain objects by grammatical inspection

- 1. The **bookstore** will be web based initially, but it must have a sufficiently flexible architecture that alternative front-ends may be developed (Swing/applets, web services, etc.)
- 2. The bookstore must be able to sell **books**, with **orders** accepted over the **Internet**.
- 3. The user must be able to add books into an online **shopping cart**, prior to **checkout**.
- a. Similarly, the user must be able to remove **items** from the shopping cart.
- 4. The user must be able to maintain **wish lists** of books that he or she wants to purchase later.
- 5. The user must be able to cancel orders before they've shipped.
- 6. The user must be able to pay by **credit card** or **purchase order**.

Domain classes aren't tables

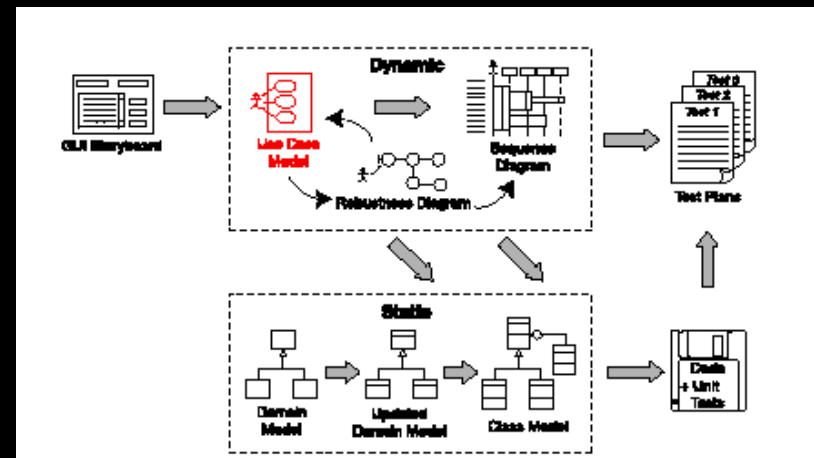


Domain Modeling – Top 10

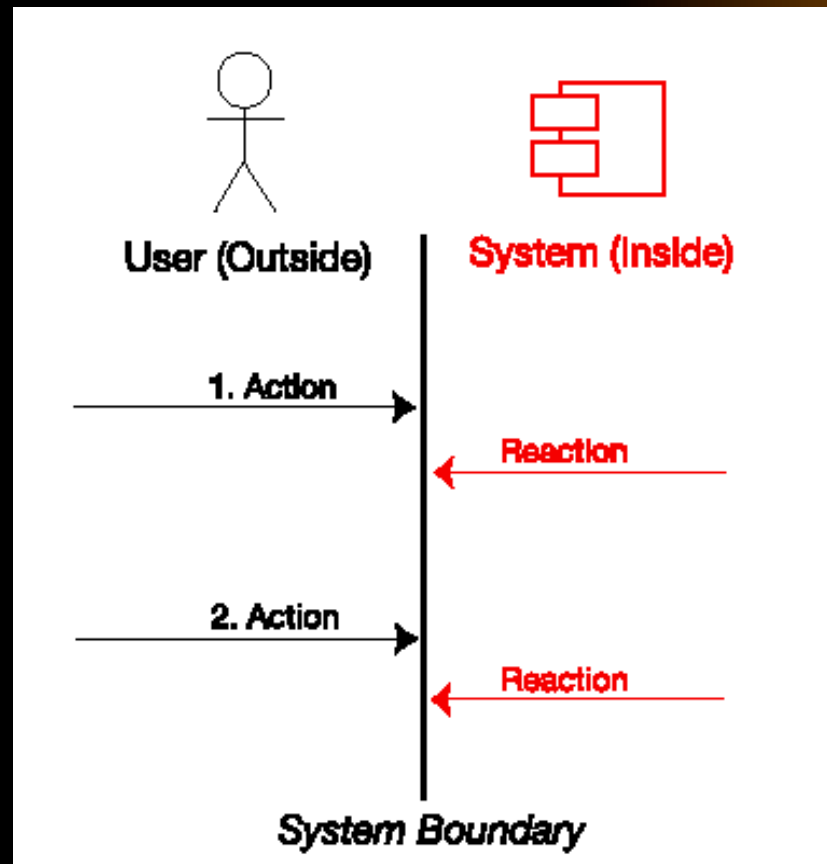
- 10. Focus on real-world (problem domain) objects.
- 9. Use generalization (is-a) and aggregation (has-a) relationships to show how the objects relate to each other.
- 8. Limit your initial domain modeling efforts to a couple of hours.
- 7. Organize your classes around key abstractions in the problem domain.
- 6. Don't mistake your domain model for a data model.
- 5. Don't confuse an object (which represents a single instance) with a database table (which contains a collection of things).
- 4. Use the domain model as a project glossary.
- 3. Do your initial domain model before you write your use cases, to avoid name ambiguity.
- 2. Don't expect your final class diagrams to precisely match your domain model, but there should be some resemblance between them.
- 1. Don't put screens and other GUI-specific classes on your domain model.

An in depth look at use cases

- Write an event/response description that covers both sides of the user/system dialogue
- Storyboard the UI
- Factor out commonality using <invokes> and <precedes>
- Use Cases vs. Algorithms
- 3 Magic Questions
- Do's and Don'ts
- Use Cases – Top 10



Write an event/response description that covers both sides of the user/system dialogue

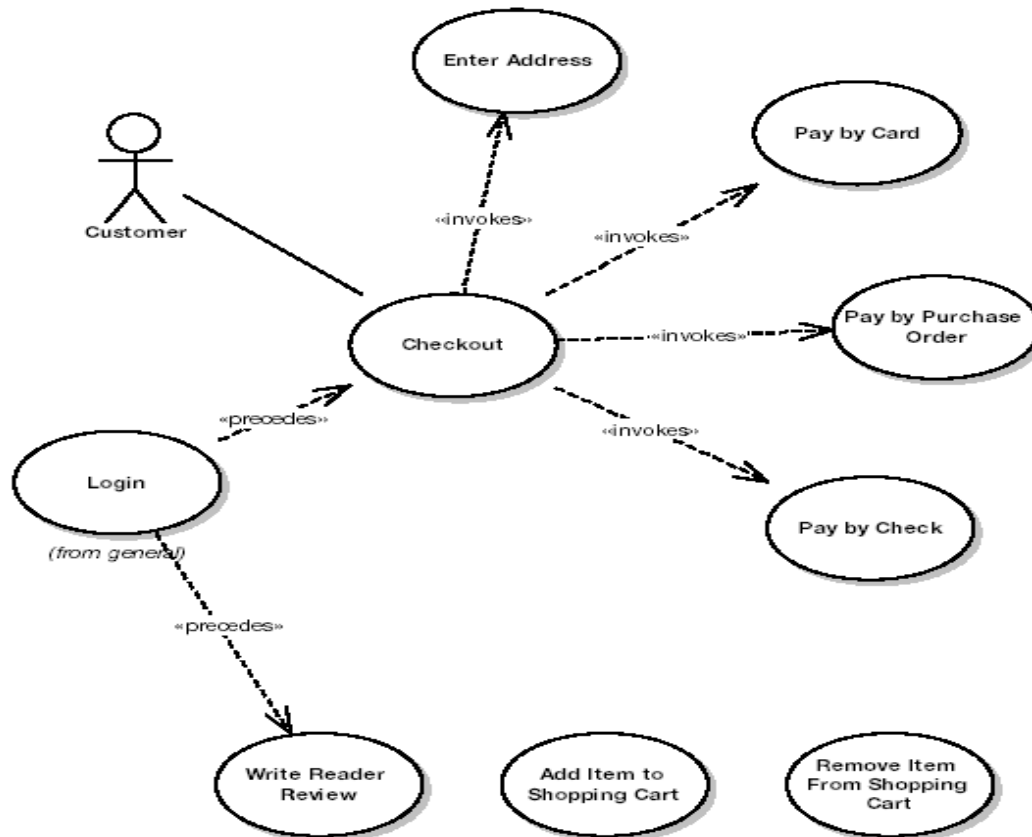


Storyboard the UI

Internet Bookstore - Edit Shopping Cart

Items in Your Shopping Cart	Price:	Qty:
Domain Driven Design	\$42.65	<input type="text" value="1"/>
Extreme Programming Refactored	\$29.65	<input type="text" value="1"/>

Factor out commonality using *<invokes>* and *<precedes>*



Use case diagram stereotypes as a cure for insomnia



The extending use case must define extension points where it may be extended, but the extended use cases must remain independent of the extending use cases, whereas including use cases do not define extension points even though their behavior is extended by the included use case, which is similar to use cases using generalization, which also do not define extension points but their behavior is not extended by the included use case per se—it's overridden, as opposed to preceding use cases, which must take place in their entirety before the child use case begins. The extends arrow MUST be drawn from the extended use case to the extending use case in an analogous fashion to a generalization arrow being drawn from the subclass toward its parent class, while an includes arrow is ALWAYS drawn from the including use case to the included use case in an analogous fashion to an aggregation arrow being drawn from the aggregate class to the classes being aggregated. You may consider both includes and extends to be subtypes of invokes (that is to say, invokes includes extends and invokes also includes includes, but includes does not include invokes, nor does extends include invokes), with the distinction being all that stuff about extension points and whether the extended use case knows about the use case that's extending it or not. If you think of a use case as a fragment of a user guide, as opposed to thinking of it as a classifier, which is, of course, how it is formally defined within the UML, you may discover that the difference between having extension points or simply including the use case into which you will be branching is not particularly significant, in which case a simple invokes may suffice and you don't have to worry about which way you should draw the arrow on the diagram since the extends arrow points in the opposite direction from the includes arrow, in a similar manner to how a generalization and an aggregation arrow are drawn in opposite directions, more specifically the extends arrow MUST be drawn from the extended use case to the extending use case in an analogous fashion to a generalization arrow being drawn from the subclass toward its parent class, while an includes arrow is drawn from the including use case to the included use case in an analogous fashion to an aggregation arrow being drawn from the aggregate class to the classes being aggregated. The bottom up-arrow convention for extends being analogous to generalization may cause confusion between generalization and extends, especially among Java programmers, to whom extends already means generalization. (Extends is a subtype of invokes, so you could say that extends extends invokes; but here we're using extends in the generalization sense, not in the UML extends sense, so it's an extension of the extended OO terminology; but to say that extends extends extends [but not a UML extends] would be extending the truth.) Additionally, when non-technical personnel are asked to review the use cases, they occasionally experience consternation while attempting to follow the arrows on the use case diagrams, since some arrows point from the invoking use case to the invoked use case, while others point from the invoked use case back toward the invoking use case. This problem is generally indicative of a lack of UML training among nontechnical personnel and is readily solved by forcing all users and marketing folks to attend a three-day "UML for nontechnical personnel" workshop, which will educate them on these subtle yet critically important features of UML. Precedes, on the other hand, is a somewhat different stereotype than includes, invokes, or extends, in that it simply indicates a temporal precedence; that is to say it is *occasionally* useful to indicate on a use case diagram that use case A needs to happen BEFORE use case B (i.e., there is temporal precedence in which A MUST OCCUR before B). Neither of the standard UML use case diagram stereotypes (i.e., neither includes nor extends) provides a convenient mechanism for expressing this concept of temporal precedence (despite the fact that showing temporal precedence is often more useful than showing whether the invoked use case has

Use cases vs. algorithms



Use Case

Dialogue between user and system
Event/response sequence
Basic/alternate courses
Multiple participating objects
User and System



Algorithm

“Atomic” computation
Series of steps
One step of a use case
Operation on a class
All System

Lunch



- After lunch we' ll do a collaborative session and develop the domain model, then create a use case package for each lab team, and draw the initial use case diagrams.

Collaborative Session



- Draw the domain model
- Create a use case package for each lab team
- Create use case diagrams for each team
- Assign team members

Use Case Do's and Don'ts



- Don't: write pseudocode
- Don't: write a flowchart in text form
- Don't: forget that non-technical people will have to understand your use case
- Do: write in the style of a user guide
- Do: name participating domain objects
- Do: name the screens

Use Cases – 3 Magic Questions



- What happens?
- And then what happens?
- What else might happen?

Use Cases – Top 10



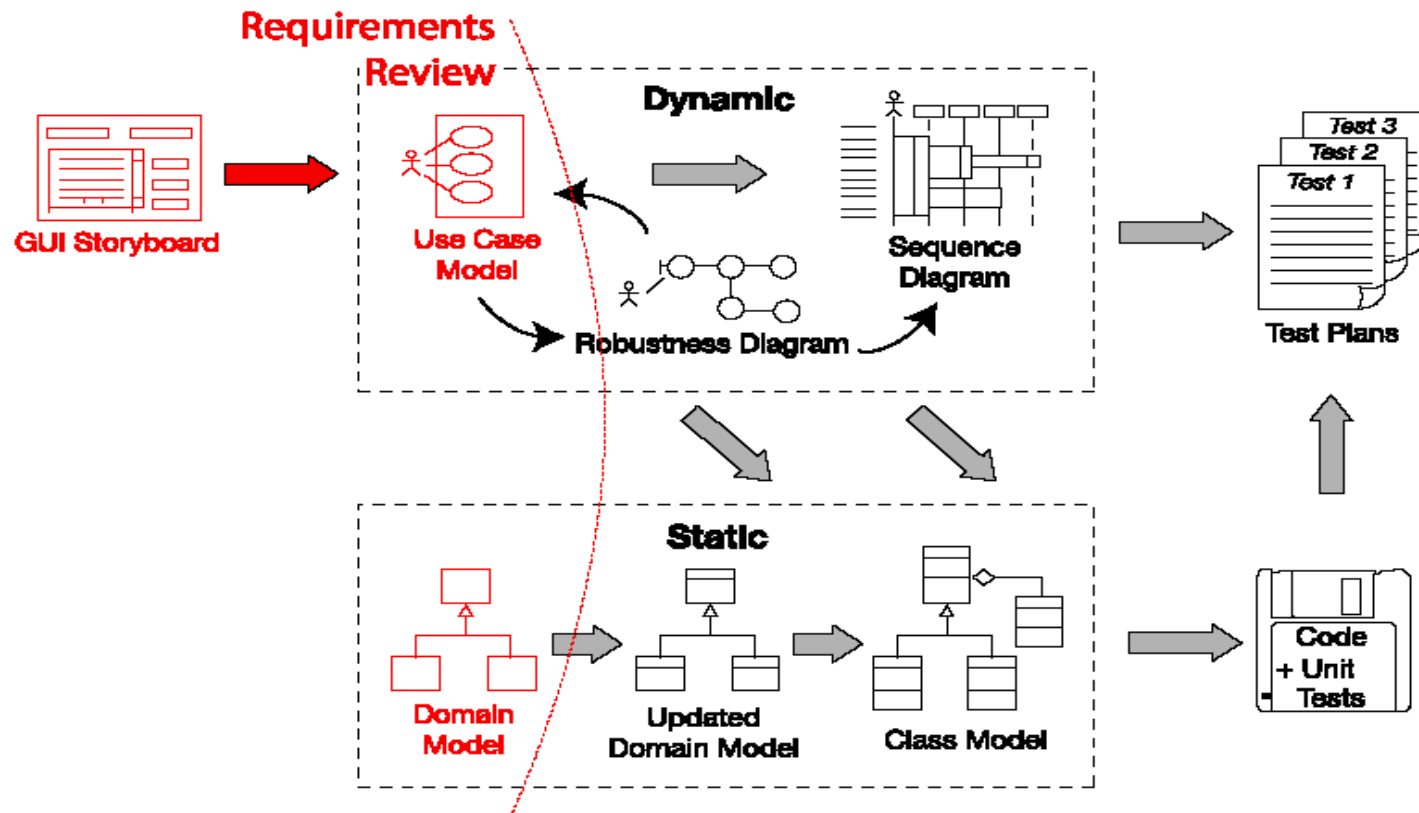
- 10. Follow the **two-paragraph** rule.
- 9. Organize your use cases with actors and use case diagrams.
- 8. Write your use cases in **active voice**.
- 7. Write your use case using an **event/response** flow, describing **both sides of the user/system dialogue**.
- 6. Use **GUI storyboards**, prototypes, screen mockups, etc.
- 5. Remember that your use case is really a runtime behavior specification.
- 4. Write the use case **in the context of the object model**.
- 3. Write your use cases using a **noun-verb-noun** sentence structure.
- 2. Reference **domain classes** by name.
- 1. Reference **boundary classes** (e.g., screens) by name.

Lab – writing use cases



- Write the first draft use cases
- Follow the Top 10 rules
- Expect them to be a little vague and ambiguous, but try your best to avoid it.
- The remaining ambiguity will be removed during the next lab (robustness analysis)

Requirements Review

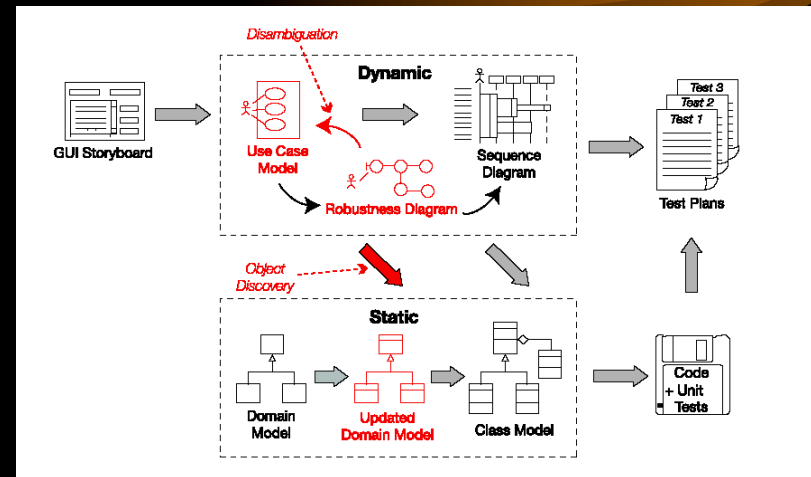


Requirements Review – Top 10

- **10.** Make sure your domain model describes at least 80% of the most important abstractions (i.e., real-world objects) from your problem domain, in nontechnical language that your end users can understand.
- **9.** Make sure your **domain model shows** the **is-a** (generalization) and **has-a** (aggregation) relationships between the domain objects.
- **8.** Make sure your use cases describe both **basic** and **alternate** courses of action, in active voice.
- **7.** If you have lists of functional requirements (i.e., “shall” statements), make sure these are **not absorbed into and “intermangled” with the active voice use case text.**
- **6.** Make sure you’ve organized your use cases into packages and that each package has at least one use case diagram.
- **5.** Make sure your use cases are written **in the context of the object model.**
- **4.** Put your use cases **in the context of the user interface.**
- **3.** Supplement your use case descriptions with some sort of **storyboard**, line drawing, screen mockup, or GUI prototype.
- **2.** Review the use cases, domain model, and screen mockups/GUI prototypes with end users, stakeholders, and marketing folks, in addition to more technical members of your staff.
- **1.** Structure the review around our “eight easy steps to a better use case”

An in depth look at robustness analysis

- An example robustness diagram
- Nouns are objects, verbs are functions
- Linking your use cases to the object model and to the GUI
- Rules for robustness analysis
- Don't forget the controllers
- Closing the gap between what and how
- Disambiguation + Object Discovery



An example robustness diagram

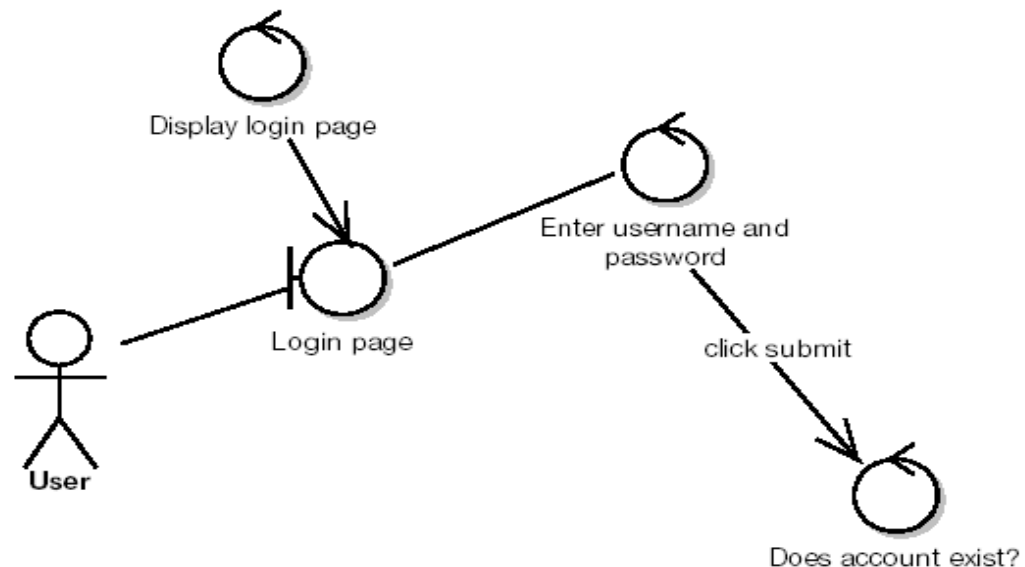
BASIC COURSE:

The user clicks the login link from any of a number of pages; the system displays the login page. The user enters their username and password and clicks Submit. The system checks the master account list to see if the user account exists. If it exists, the system then checks the password. The system retrieves the account information, starts an authenticated session, and redisplay the previous page with a welcome message.

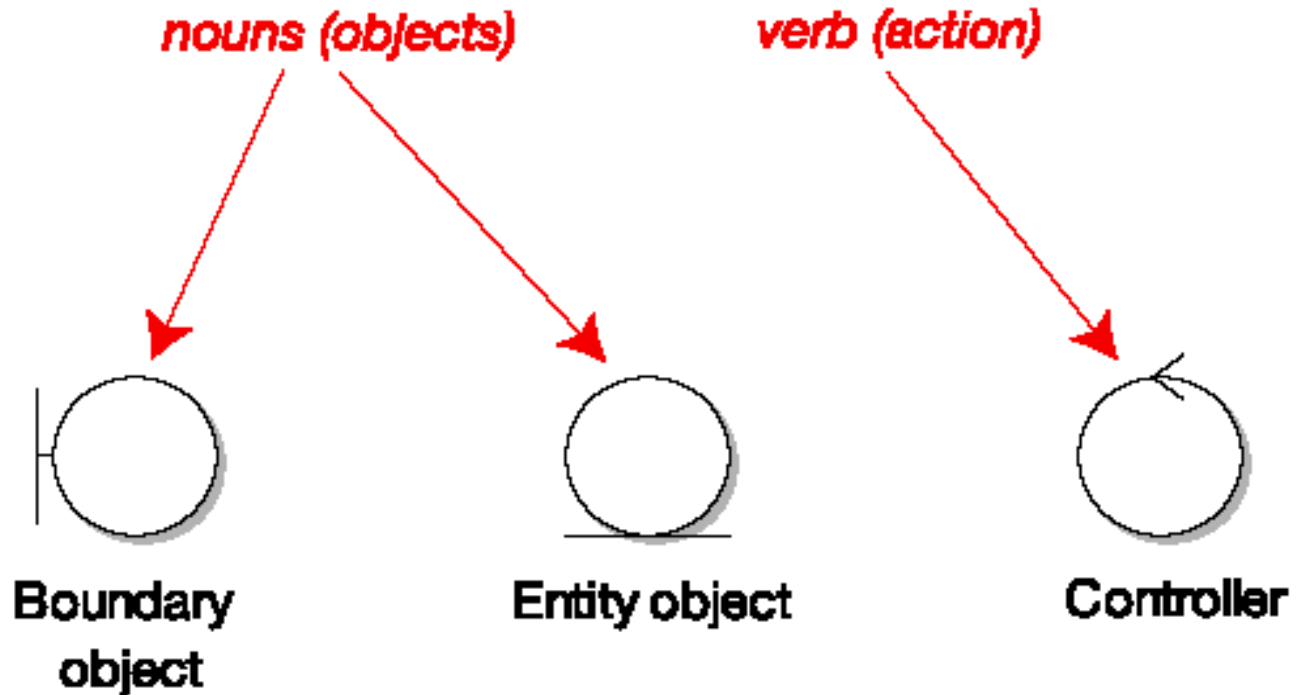
ALTERNATE COURSES:

User forgot the password: The user clicks the What's my Password? link. The system prompts the user for their username if not already entered, retrieves the account info, and emails the user their password.

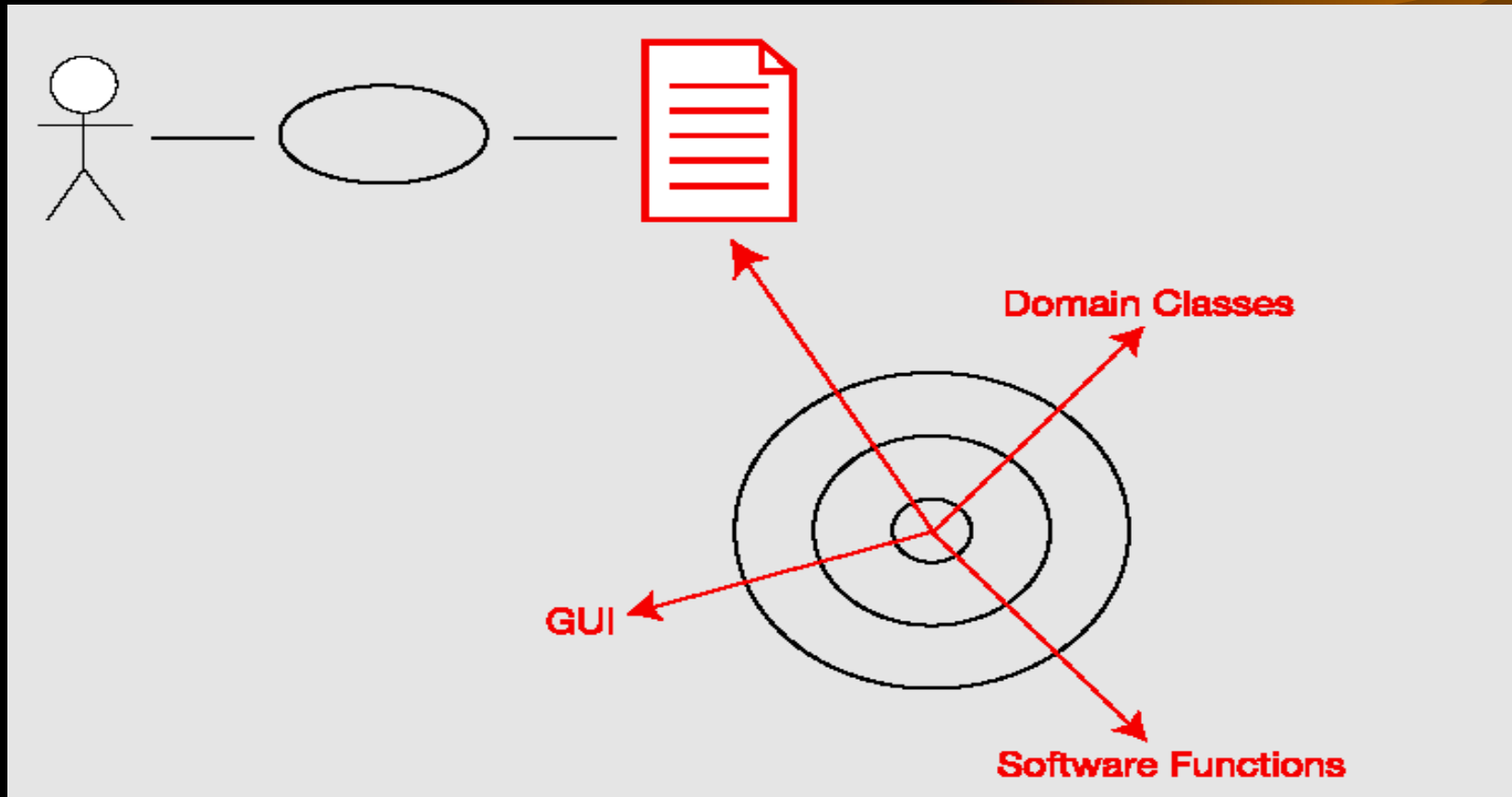
Invalid account: The system displays a message saying that



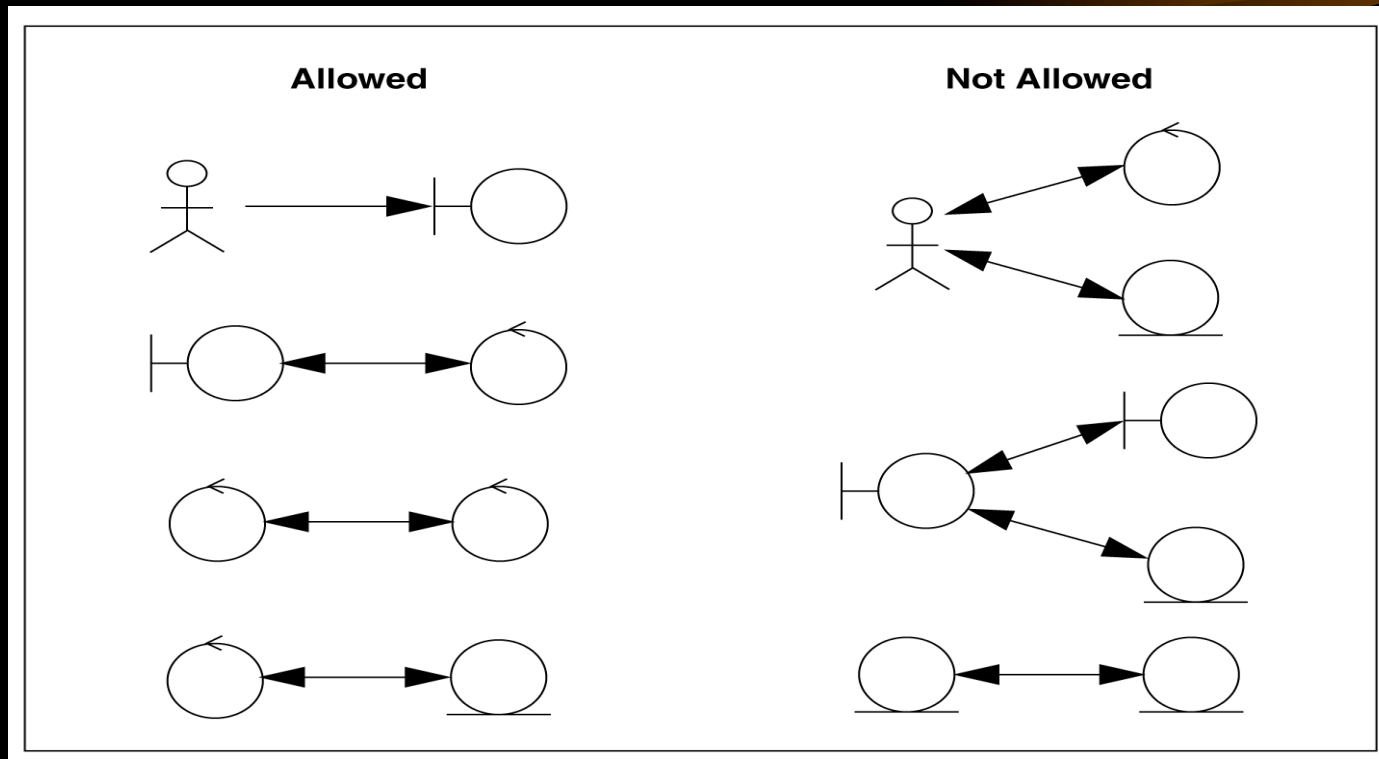
Nouns are objects, verbs are functions



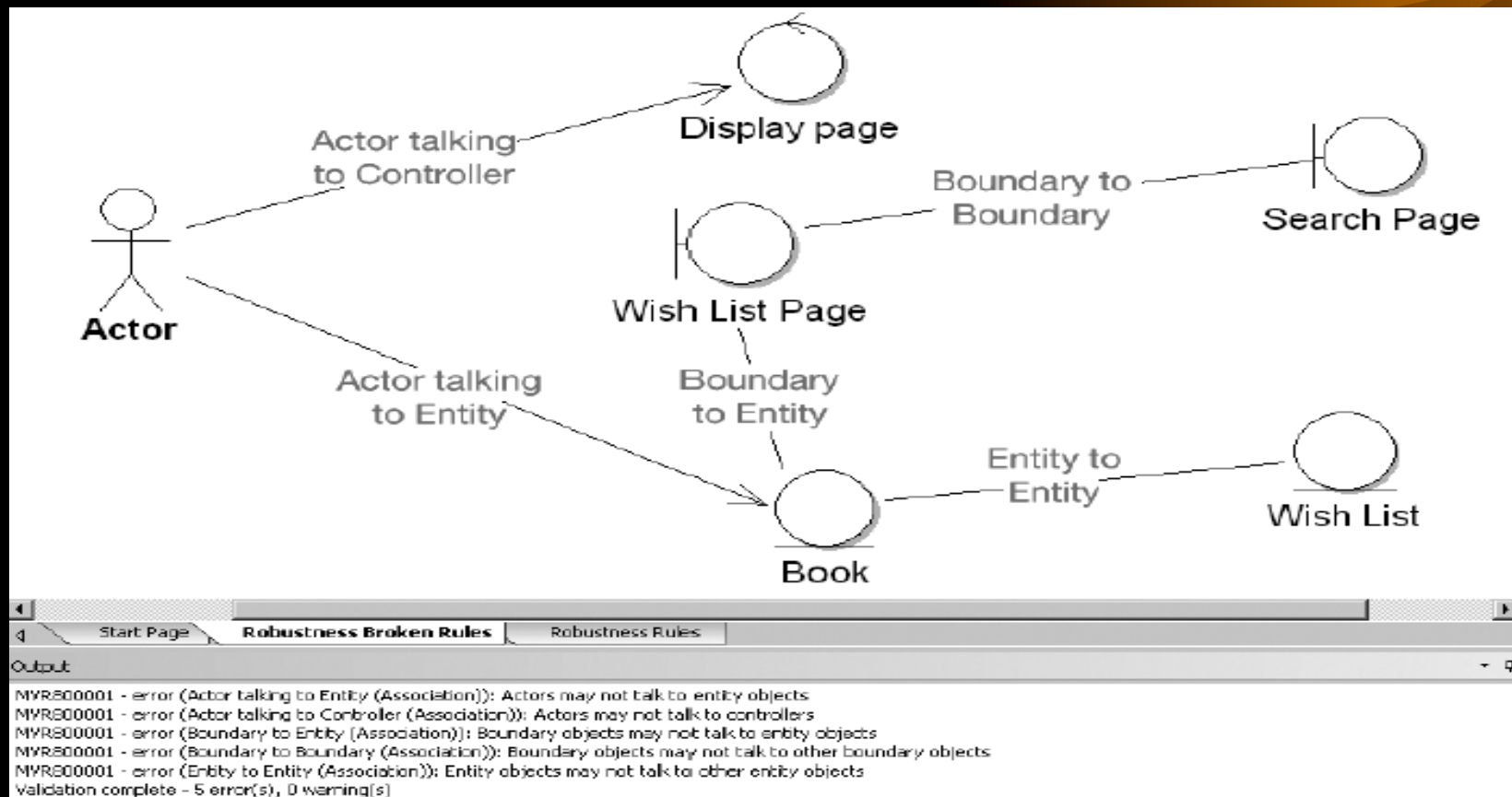
Linking your use cases to the object model, and to the GUI



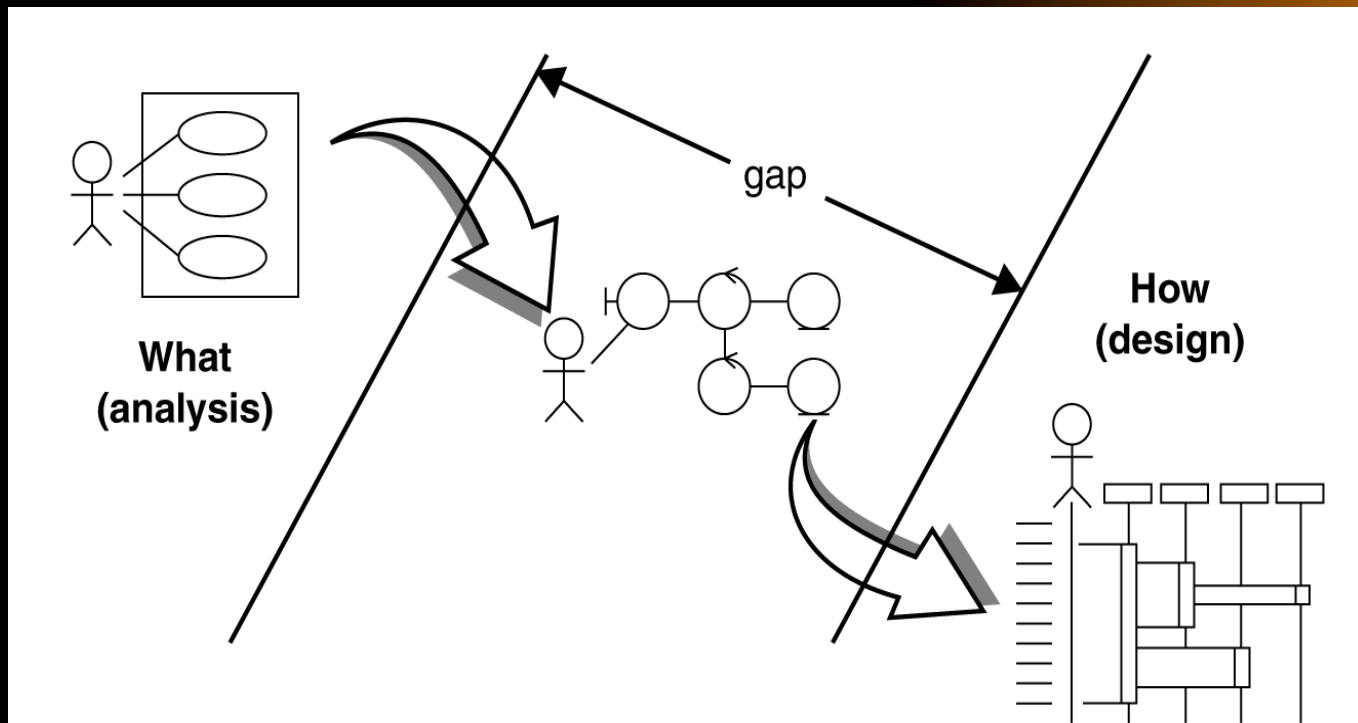
Rules for robustness analysis



Don't forget the controllers

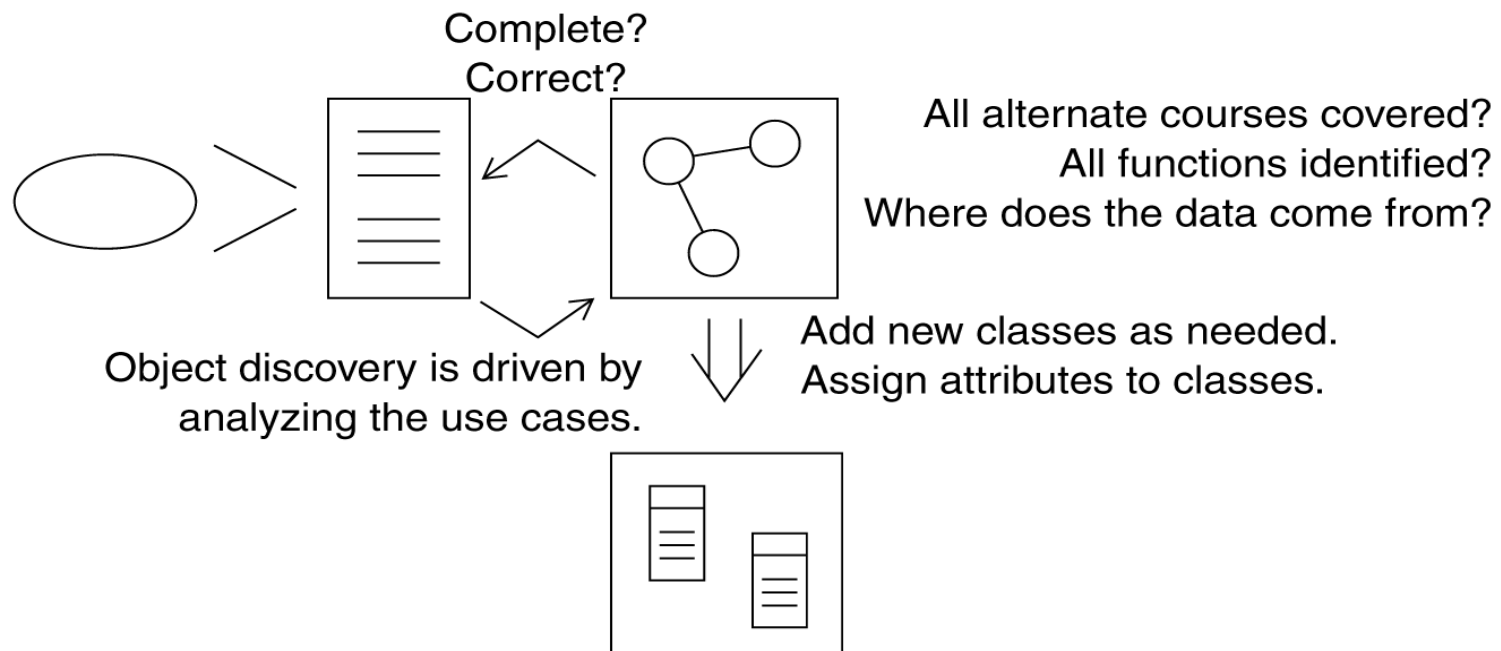


Closing the gap between “what” (analysis) and “how” (design)



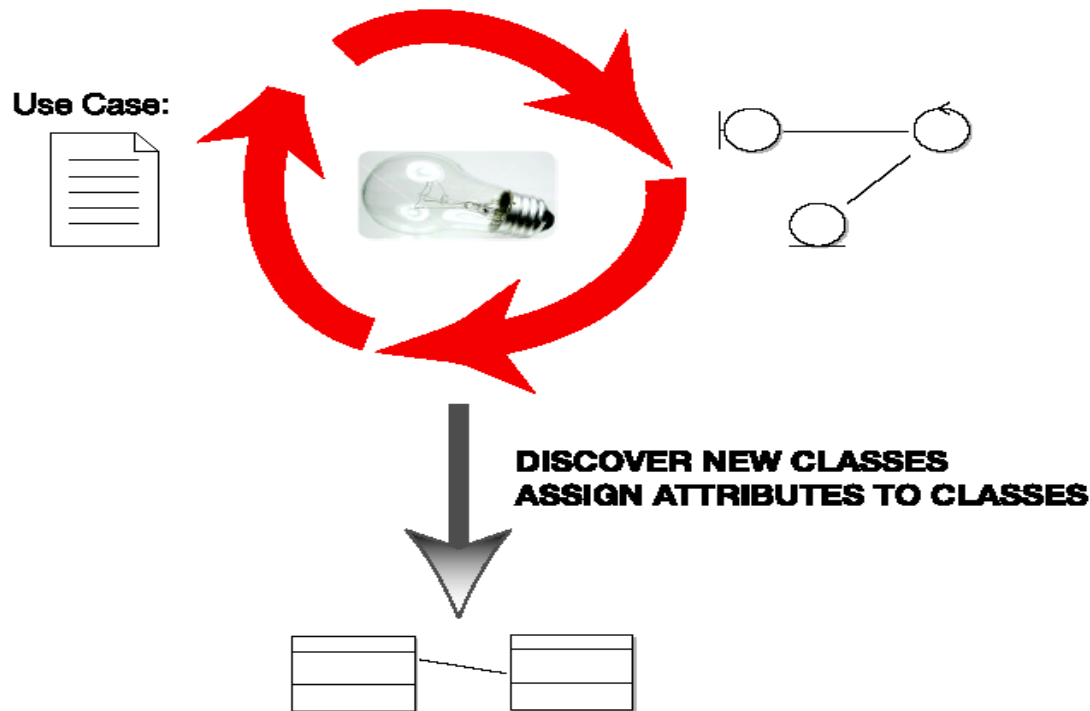
Mind the gap: Many projects fail while trying to cross this what/how gap.

Disambiguation + Object Discovery



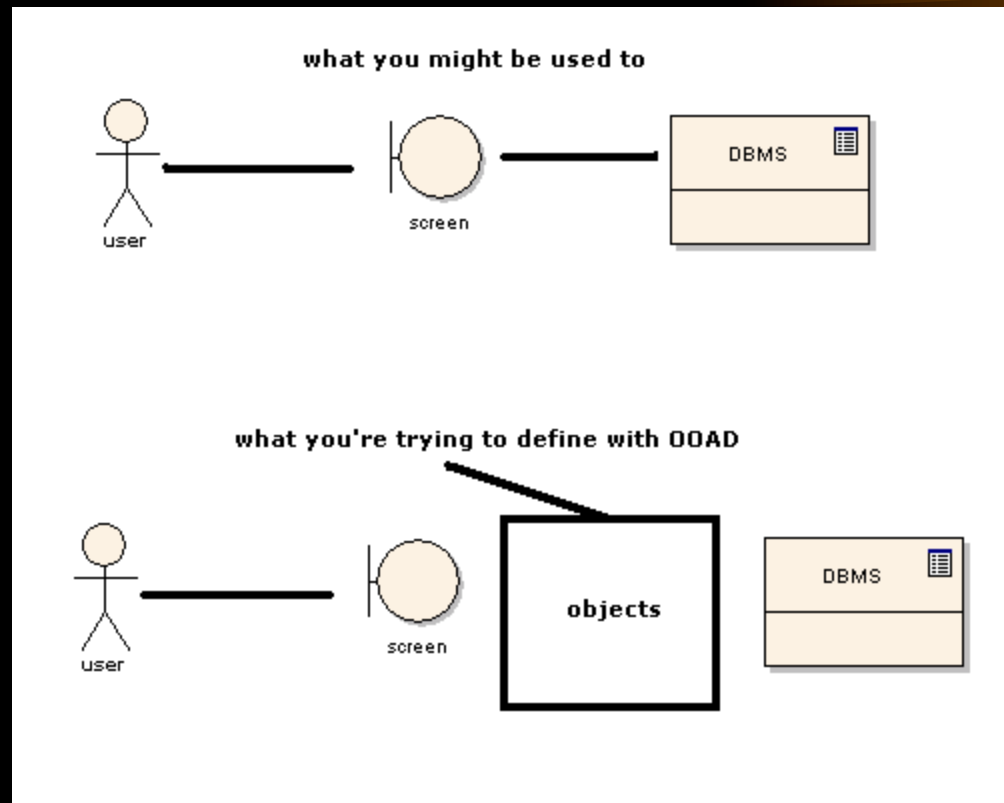
The domain model evolves into a detailed static model.
This evolution is driven by working through the use cases.

Object Discovery

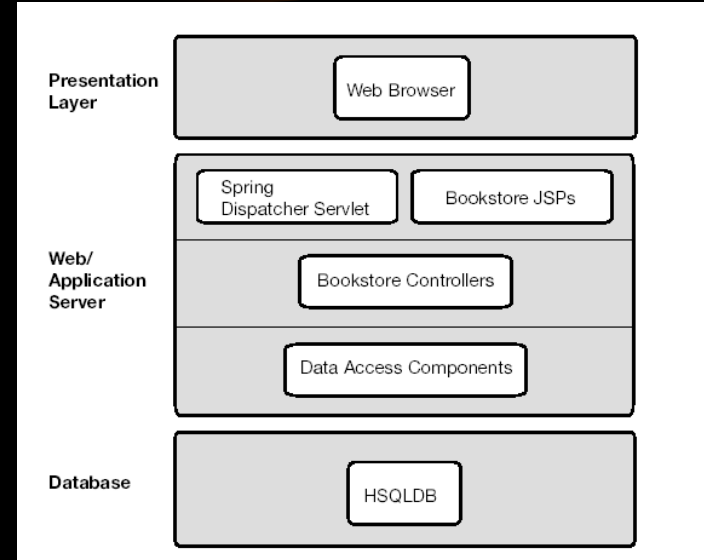
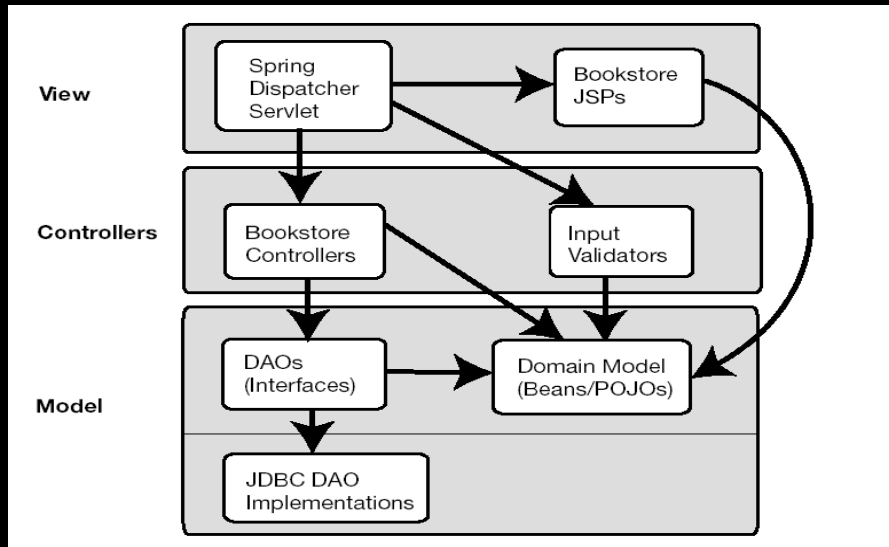


**Repeat for each use case, until the
Domain model has evolved into a static model**

Domain objects sit between the screens and the DBMS



Robustness diagrams should reflect Technical Architecture



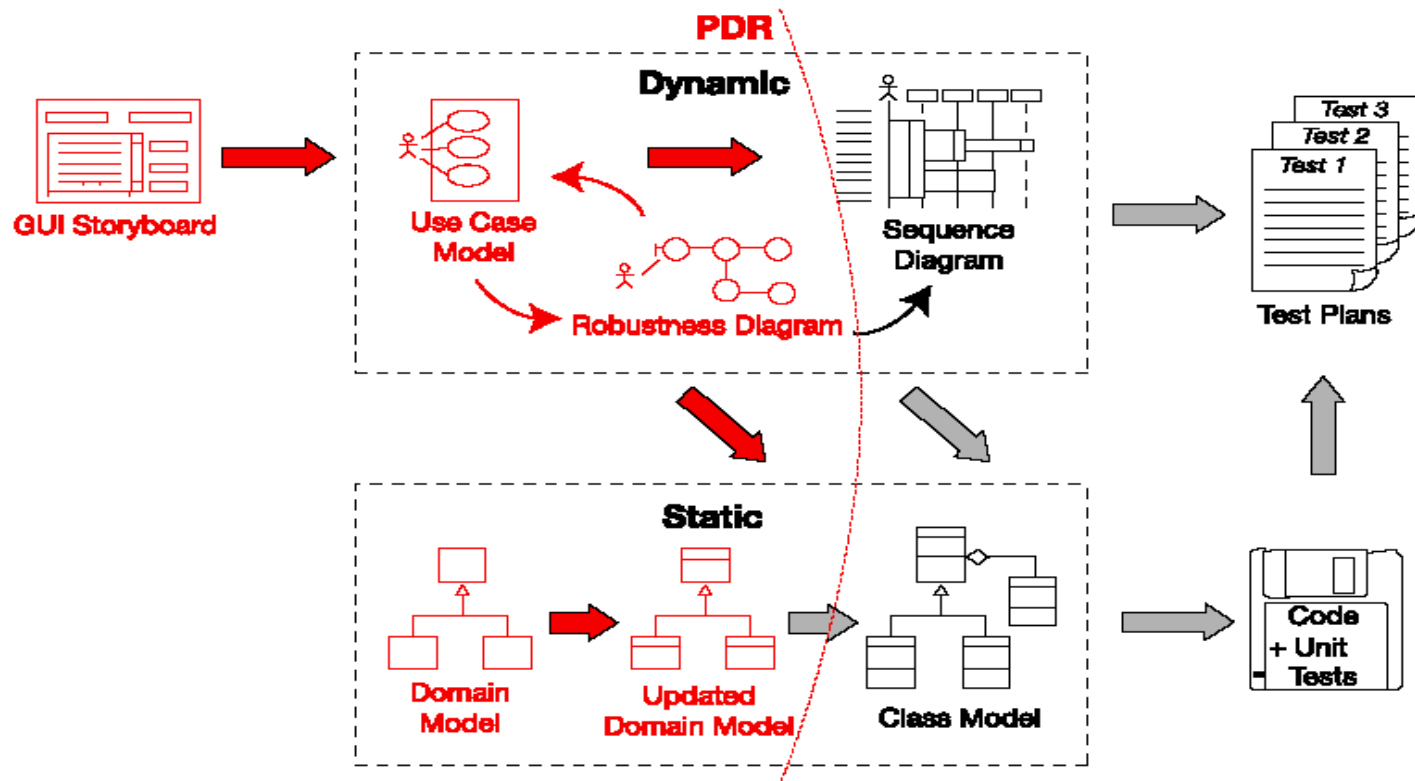
Robustness Analysis – Top 10

- 10. Paste the use case text directly onto your robustness diagram.
- 9. Take your entity classes from the domain model, and add any that are missing.
- 8. Expect to rewrite (*disambiguate*) your use case while drawing the robustness diagram.
- 7. Make a boundary object for each screen, and name your screens unambiguously.
- 6. Remember that controllers are only occasionally **real control objects**; they are typically **logical software functions**.
- 5. Don't worry about the direction of the arrows on a robustness diagram.
- 4. It's OK to drag a use case onto a robustness diagram if it's invoked from the parent use case.
- 3. The robustness diagram represents a *preliminary conceptual design* of a use case, not a literal detailed design.
- 2. Boundary and entity classes on a robustness diagram will generally become object instances on a sequence diagram, while controllers will become messages.
- 1. Remember that a robustness diagram is an “object picture” of a use case, whose purpose is to force refinement of both use case text and the object model.

Lab – robustness analysis

- Suggestion: draw the robustness diagrams on paper first, because you'll be fixing the use case text while you do this. After you've cleaned up the use cases, copy your robustness diagram into the tool.
- Don't forget to update the domain model with new objects and with attributes.
- **Always** drag entity classes from the domain model.

Preliminary Design Review

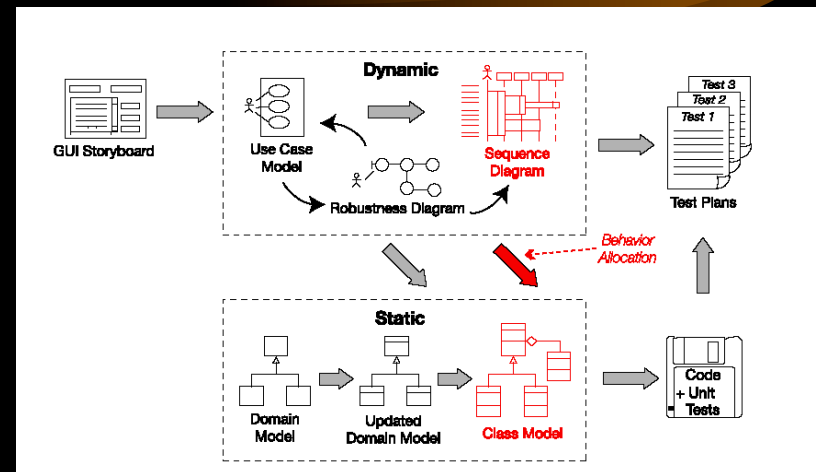


Preliminary Design Review – Top 10

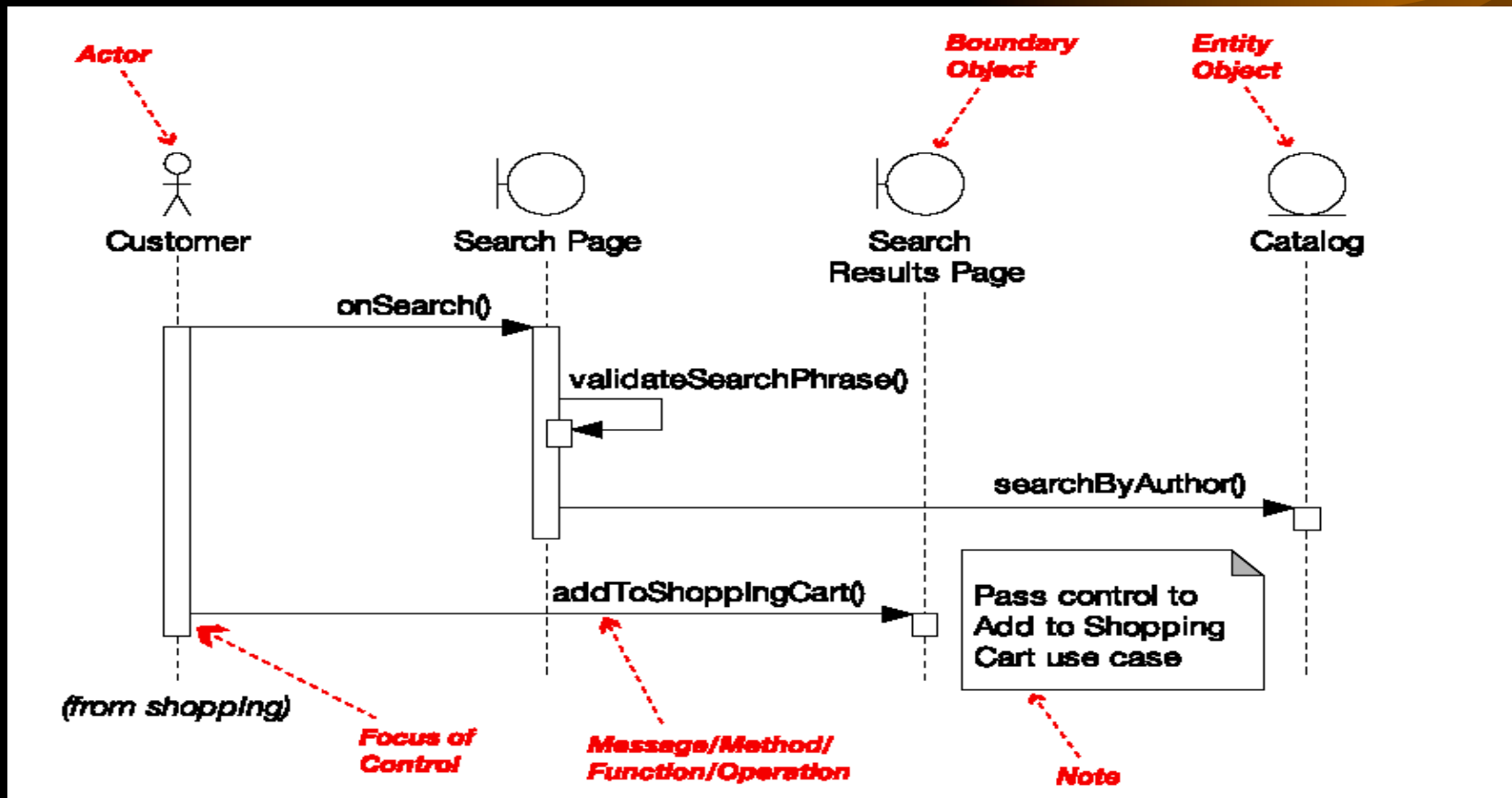
- **10.** For each use case, **make sure the use case text matches the robustness diagram**, using the highlighter test.
- **9.** Make sure that all the **entities** on all robustness diagrams appear within the updated **domain model**.
- **8.** Make sure that you can trace data flow between entity classes and screens.
- **7.** **Don't forget the alternate courses**, and don't forget to write behavior for each of them when you find them.
- **6.** Make sure each use case covers **both sides of the dialogue between user and system**.
- **5.** Make sure you haven't violated the syntax rules for robustness analysis,
- **4.** Make sure that this review includes both nontechnical (customer, marketing team, etc.) and technical folks (programmers).
- **3.** Make sure your use cases are **in the context of the object model and in the context of the GUI**.
- **2.** Make sure your robustness diagrams (and the corresponding use case text) don't attempt to show the same level of detail that will be shown on the sequence diagrams (i.e., **don't try to do detailed design yet**).
- **1.** Follow our “six easy steps” to a better preliminary design

An in-depth look at sequence diagrams

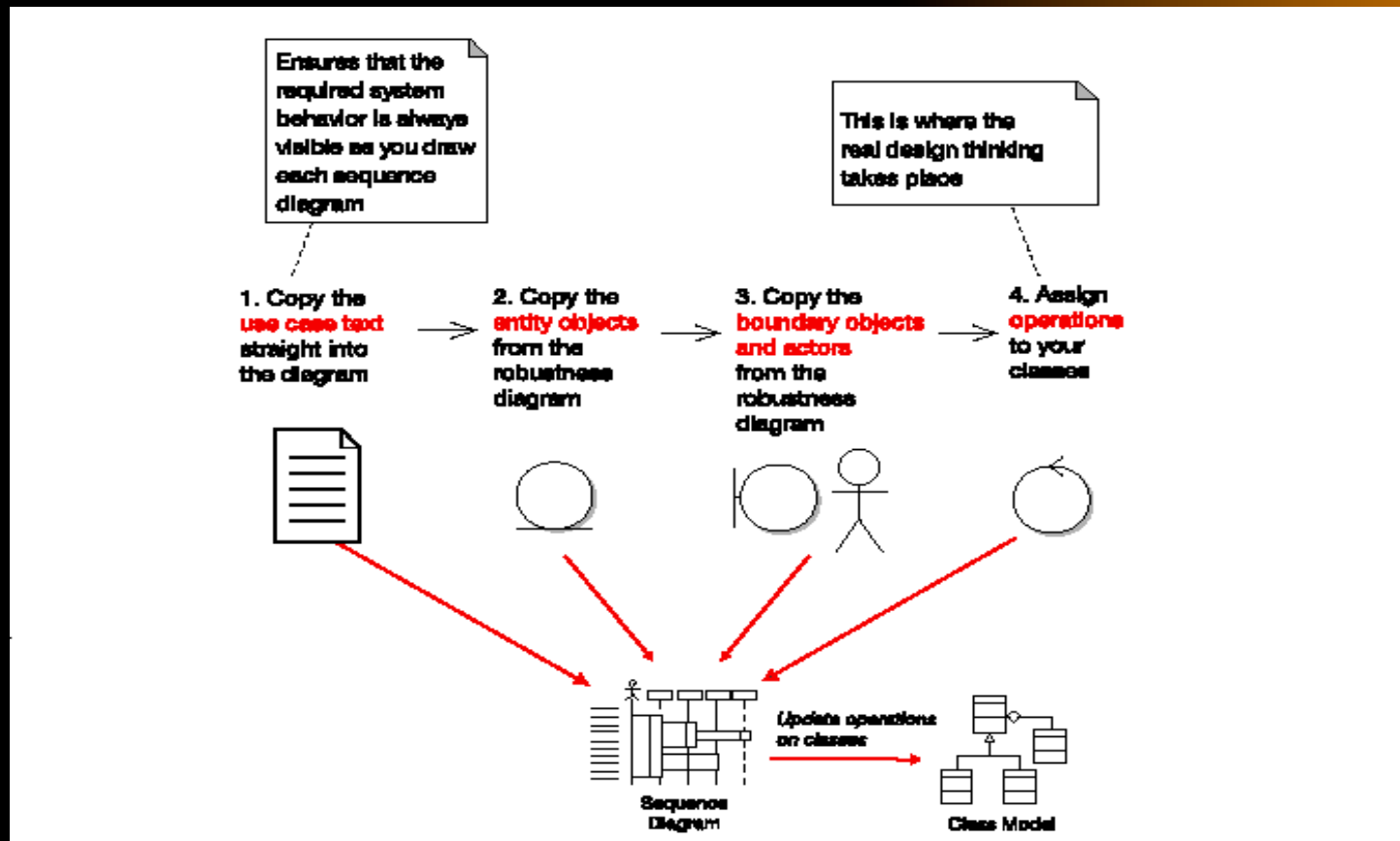
- An example Sequence Diagram
- Sequence diagramming: 4 steps
- Add behavior to the classes
- Boundaries become View classes
- Controllers usually become operations
- Assigning operations to classes
- Sequence diagrams – Top 10



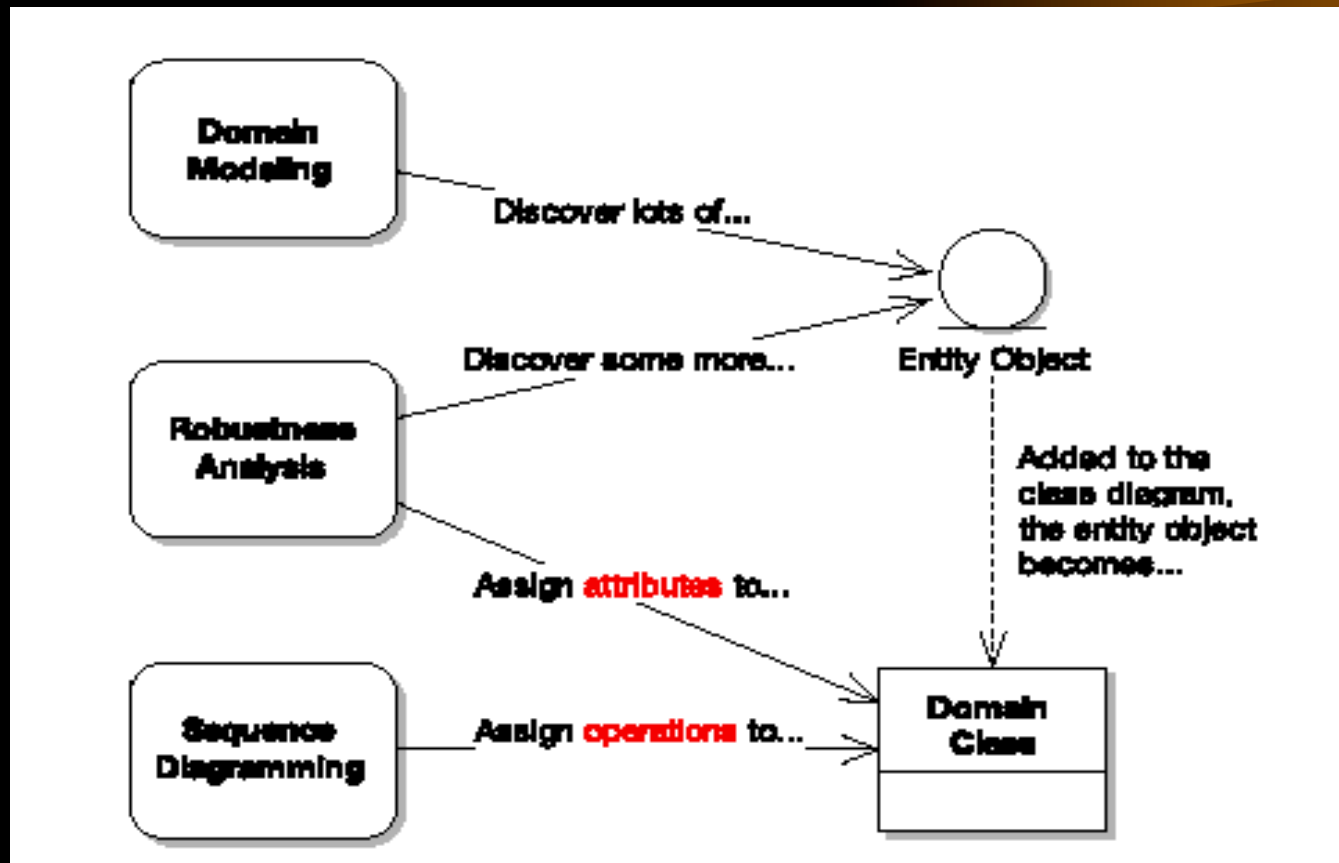
An example Sequence Diagram



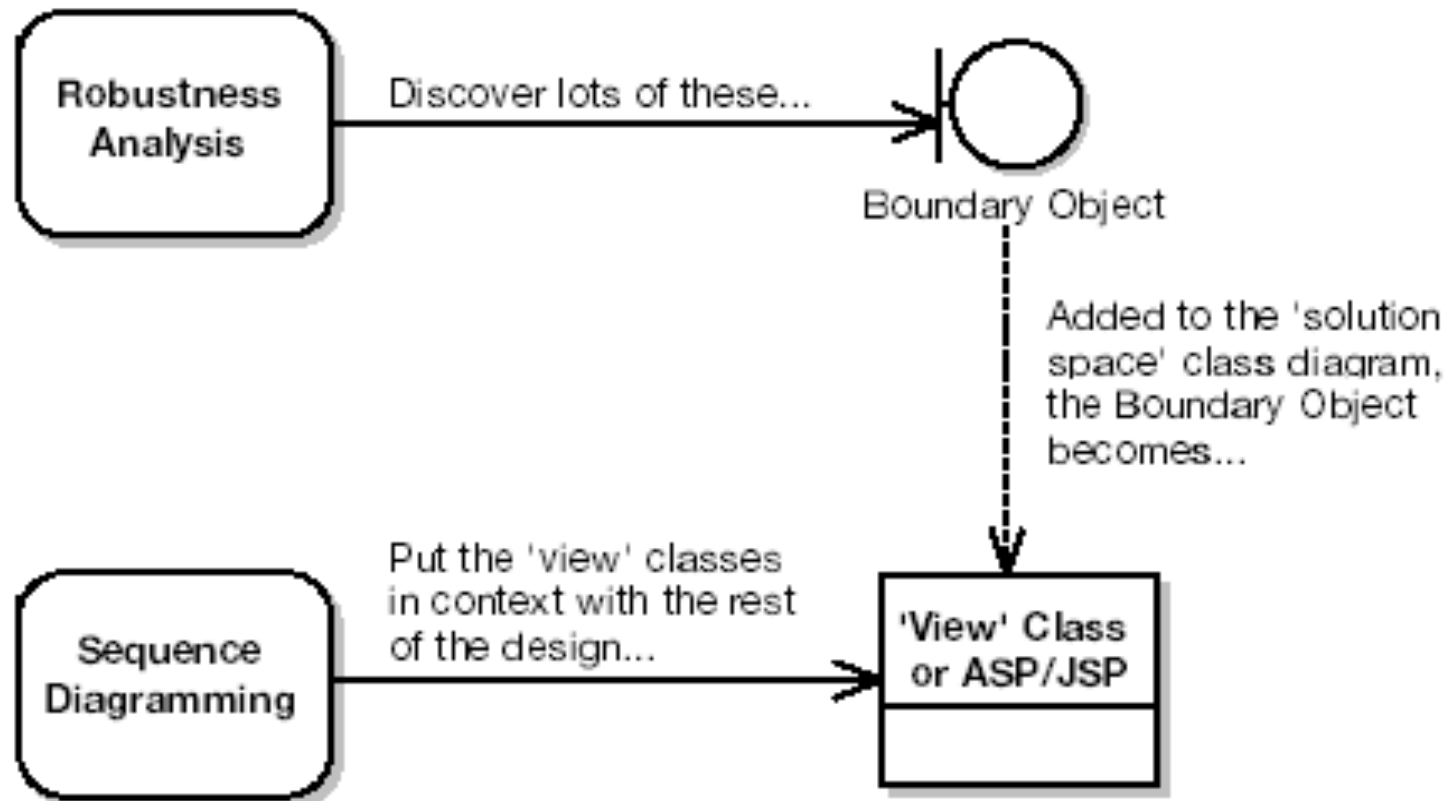
Sequence diagramming: 4 steps



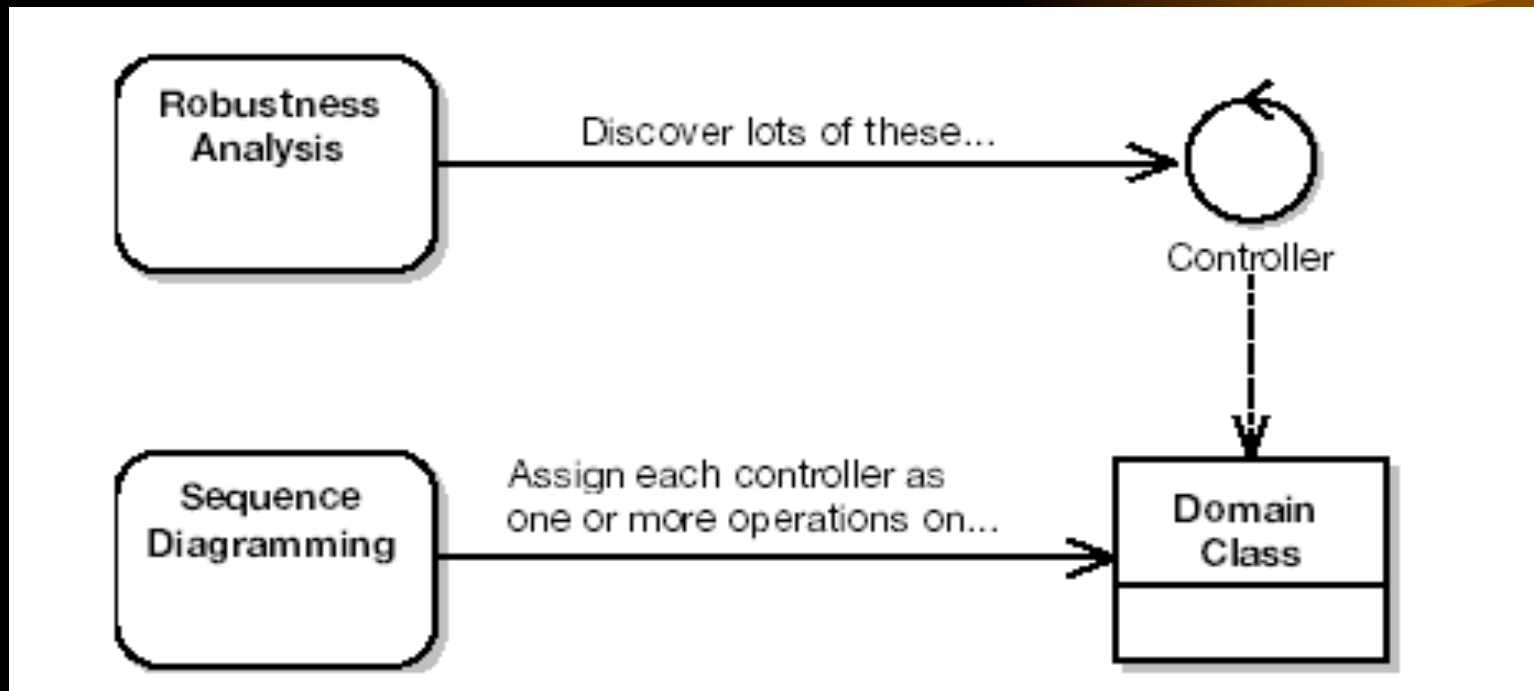
Add behavior to the classes



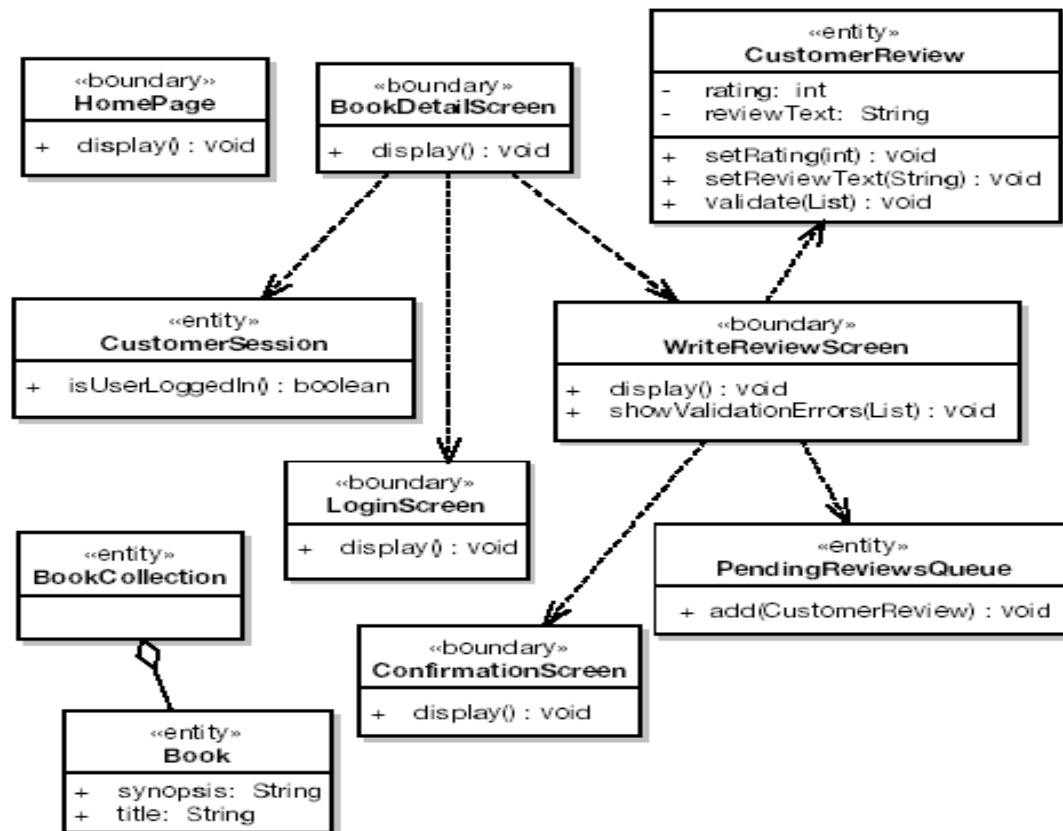
Boundaries become View classes



Controllers usually become operations



Assigning operations to classes



Sequence Diagrams – Top 10

- **10.** Understand **why** you're drawing a sequence diagram, to get the most out of it.
- **9.** Do a sequence diagram for every use case, with both basic and alternate courses on the same diagram.
- **8.** Start your sequence diagram from the boundary classes, entity classes, actors, and use case text that result from robustness analysis.
- **7.** Use the sequence diagram to **show how the behavior** of the use case (i.e., all the controllers from the robustness diagram) **is accomplished** by the objects.
- **6.** Make sure your use case text maps to the messages being passed on the sequence diagram. Try to line up the text and message arrows.
- **5.** Don't spend too much time worrying about focus of control.
- **4.** **Assign operations to classes while drawing messages.** Most visual modeling tools support this capability.
- **3.** Review your class diagrams frequently while you're assigning operations to classes, to make sure all the operations are on the appropriate classes.
- **2.** **Prefactor your design** on sequence diagrams before coding.
- **1.** Clean up the static model before proceeding to the CDR.

What is a “quality” class?



- Coupling: should be loosely coupled with other classes
- Cohesion: should be highly cohesive
- Sufficiency: does it do enough?
- Completeness: does it cover all the relevant a abstractions?
- Primitiveness: stick to basic operations

Parameterized and Instantiated Classes

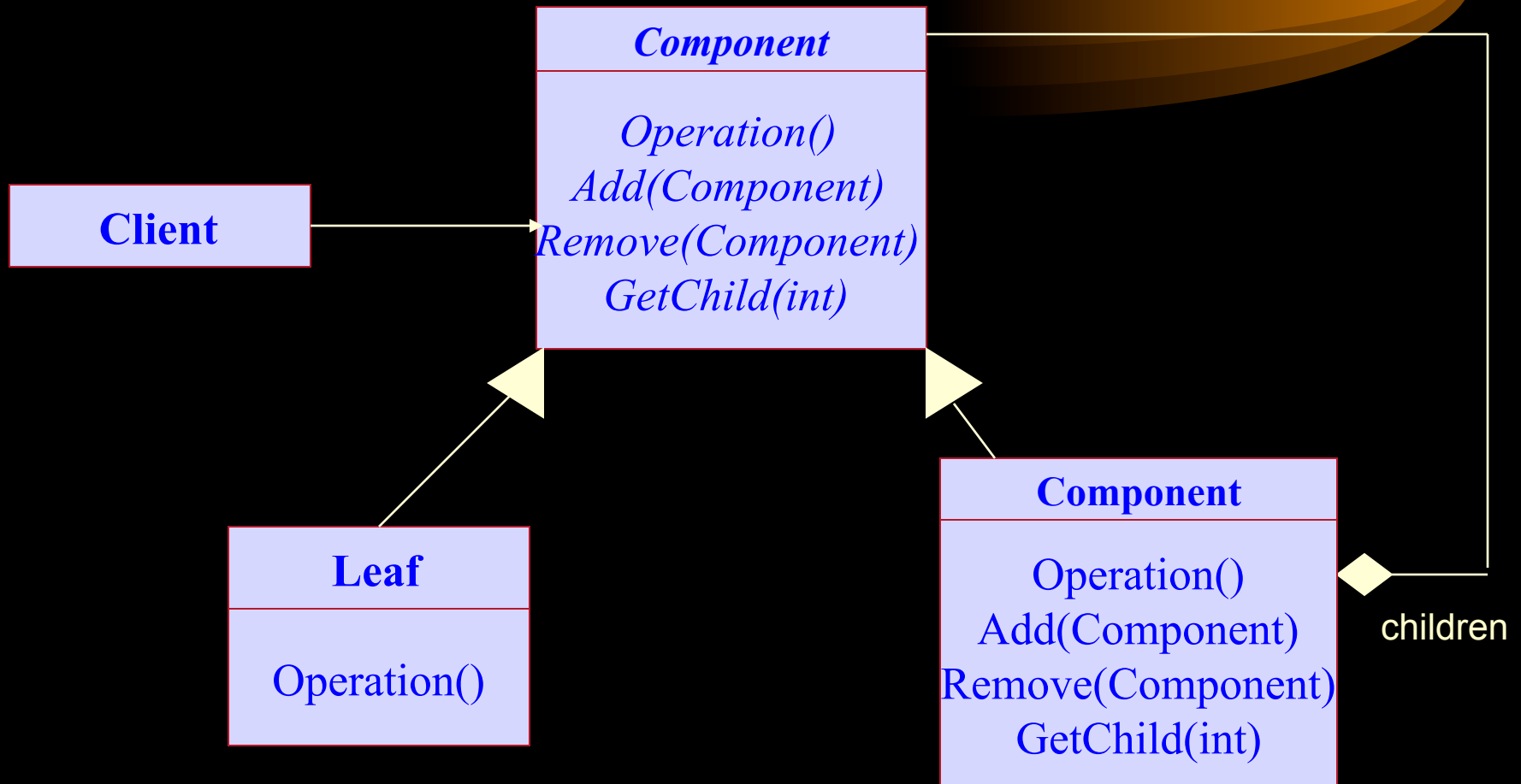


- Parameterized, or generic, classes serve as templates for other classes (C++ templates)
- Essential for application framework development; within UML, framework is architectural pattern that provides extensible template
- Used for container classes (sets, lists, trees)
- Instantiated classes represent specific instances of parameterized classes

An instantiated class is a specific instance of a parameterized class

```
// template Queue class with 2 instances
template <class Item> //Item is the formal parameter
class Queue {
..... }
Queue <int> intQueue; //Int is the actual parameter
Queue <displayItem*> ItemQueue;
```

Design patterns

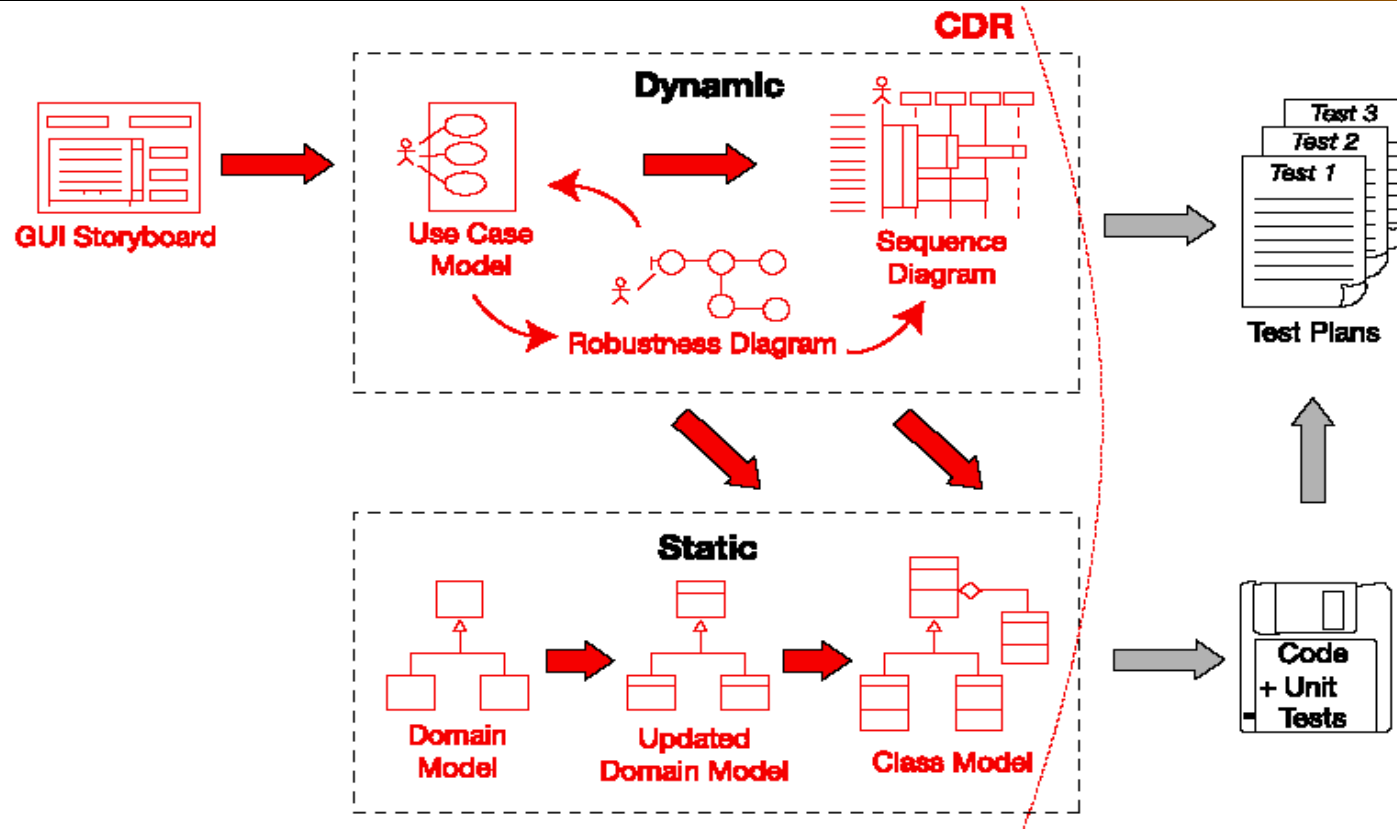


Lab – Sequence diagrams and detailed class diagrams



- Install the Agile/ICONIX add-in
- Use the add-in to generate skeleton sequence diagrams, and to generate test case diagrams
- Allocate behavior to classes while drawing the sequence diagrams, using a responsibility-driven approach
- Keep checking the class diagrams as you draw messages on the sequence diagrams.

Critical Design Review

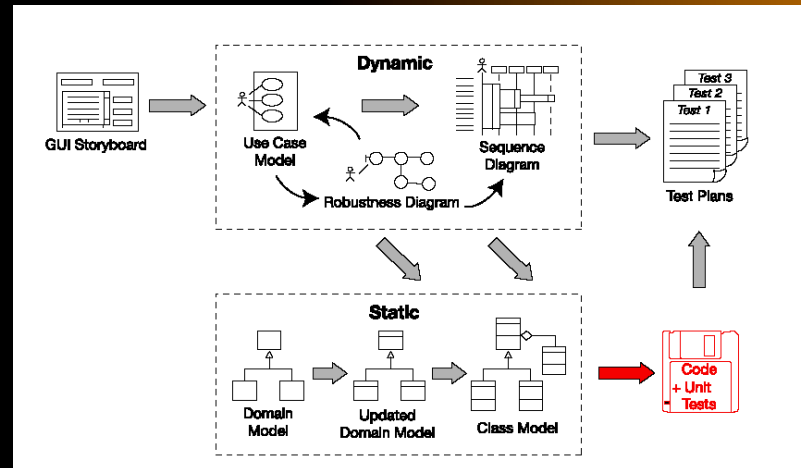


Critical Design Review

- 10. Make sure the sequence diagram matches the use case text.
- 9. Make sure (yes, again) that each sequence diagram accounts for both basic and alternate courses of action.
- 8. Make sure that operations have been allocated to classes appropriately.
- 7. Review the classes on your class diagrams to ensure they all have an appropriate set of attributes and operations.
- 6. If your design reflects the use of patterns or other detailed implementation constructs, check that these details are reflected on the sequence diagram.
- 5. Trace your functional (and nonfunctional) requirements to your use cases and classes to ensure you have covered them all.
- 4. Make sure your programmers “sanity check” the design and are confident that they can build it and that it will work as intended.
- 3. Make sure all your attributes are typed correctly, and that return values and parameter lists on your operations are complete and correct.
- 2. Generate the code headers for your classes, and inspect them closely.
- 1. Review the test plan for your release.

Implementation

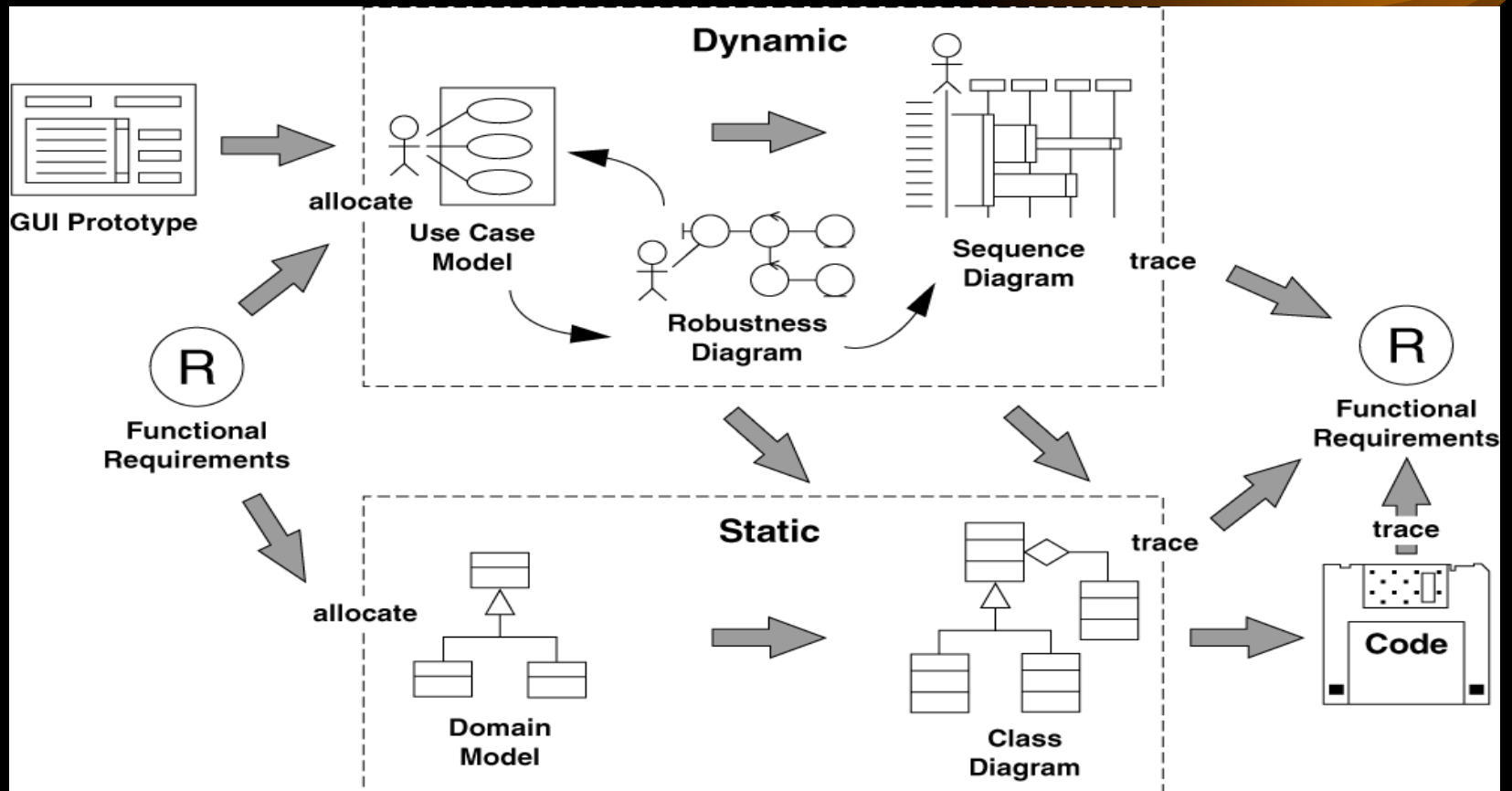
- Implementation
- Tracing Requirements
- Unit Test
- Code Review and Model Update



Implementation – Top 10

- 10. Be sure to drive the code directly from the design.
- 9. If coding reveals the design to be wrong in some way, change it. But also review the process.
- 8. Hold regular code inspections.
- 7. Always question the framework's design choices.
- 6. Don't let framework issues take over from business issues.
- 5. If the code starts to get out of control, hit the brakes and revisit the design.
- 4. **Keep the design and the code in sync.**
- 3. Focus on unit testing while implementing the code.
- 2. Don't overcomment your code (it makes your code less maintainable and more difficult to read).
- 1. **Remember to implement the alternate courses** as well as the basic courses.

Tracing requirements



Unit Testing – Top 10

- **10.** Adopt a “testing mind-set” wherein every bug found is a victory and not a defeat. If you find (and fix) the bug in testing, the users won’t find it in the released product.
- **9.** Understand the different kinds of testing, and when and why you’d use each one.
- **8.** When unit testing, **create one or more unit tests for each controller** on each robustness diagram.
- **7.** For real-time systems, use the elements on state diagrams as the basis for test cases.
- **6.** Do requirement-level verification (i.e., check that each requirement you have identified is accounted for).
- **5.** Use a **traceability matrix** to assist in requirement verification.
- **4.** Do **scenario-level acceptance testing** for each use case.
- **3.** Expand threads in your test scenarios to cover a complete path through the appropriate part of the basic course plus each alternate course in your scenario testing.
- **2.** Use a testing framework like JUnit to store and organize your unit tests.
- **1.** Keep your unit tests fine-grained.

Code Review and Model Update – Top 10

- **10.** Prepare for the review, and make sure all participants have read the relevant review material prior to the meeting.
- **9.** Create a high-level list of items to review, based on the use cases.
- **8.** If necessary, break down each item in the list into a smaller checklist.
- **7.** Review code at several different levels.
- **6.** Gather data during the review, and use it to accumulate boilerplate checklists for future reviews.
- **5.** Follow up the review with a list of action points e-mailed to all people involved.
- **4.** Try to focus on error detection during the review, not error correction.
- **3.** Use an **integrated code/model browser that hot-links your modeling tool to your code editor.**
- **2.** Keep it “just formal enough” with checklists and follow-up action lists, but don’t overdo the bureaucracy.
- **1.** Remember that it’s also a Model Update session, not just a Code Review.

Optional Lab

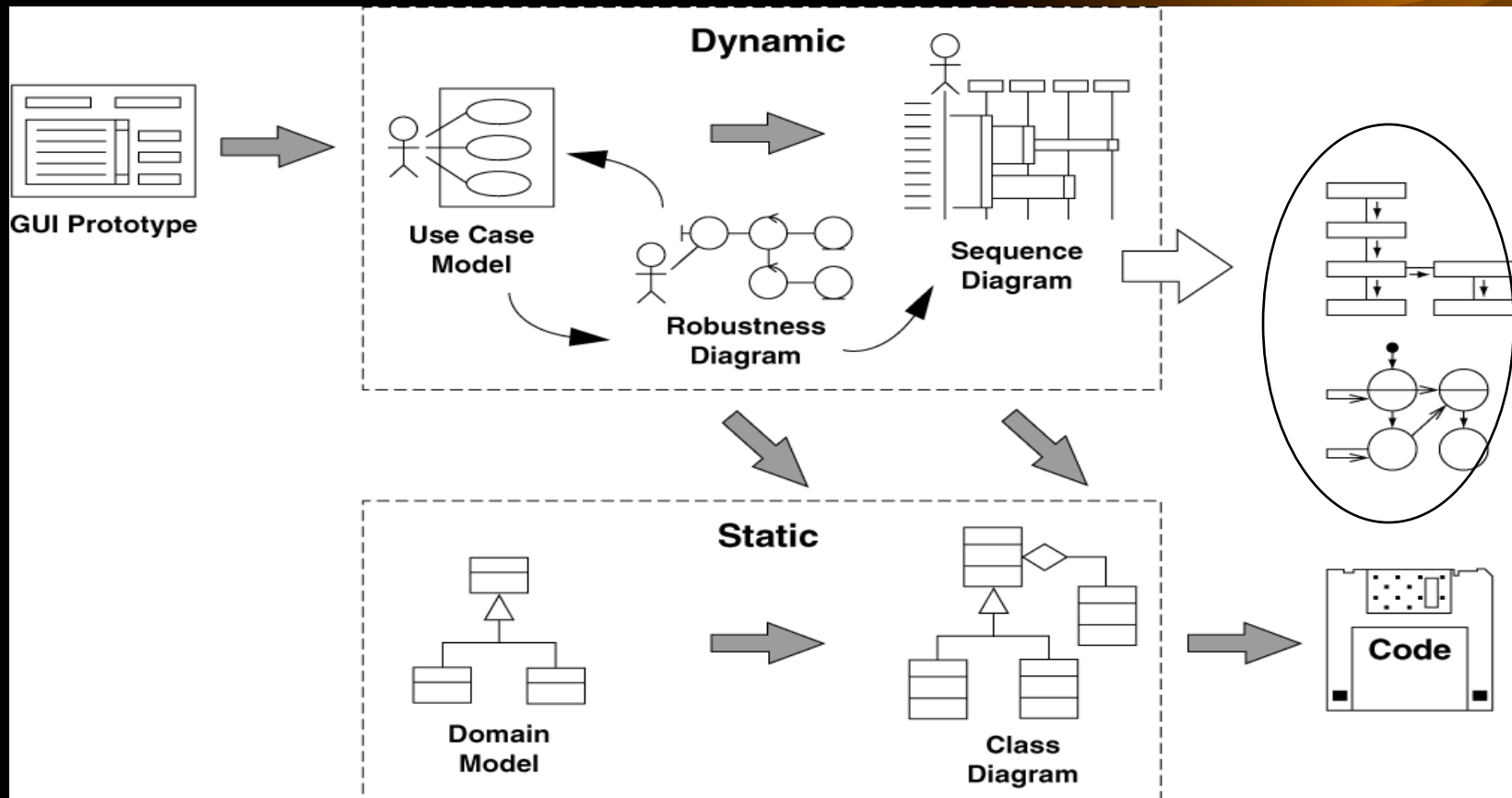


- Generate code
- Install MDG Integration for Visual Studio or Java
- Explore options for synchronizing model and code

Optional – Additional UML diagrams

- State, Activity and Collaboration Diagrams
- Component and Deployment Diagrams

State, Activity and Collaboration modeling



How do objects behave over time?



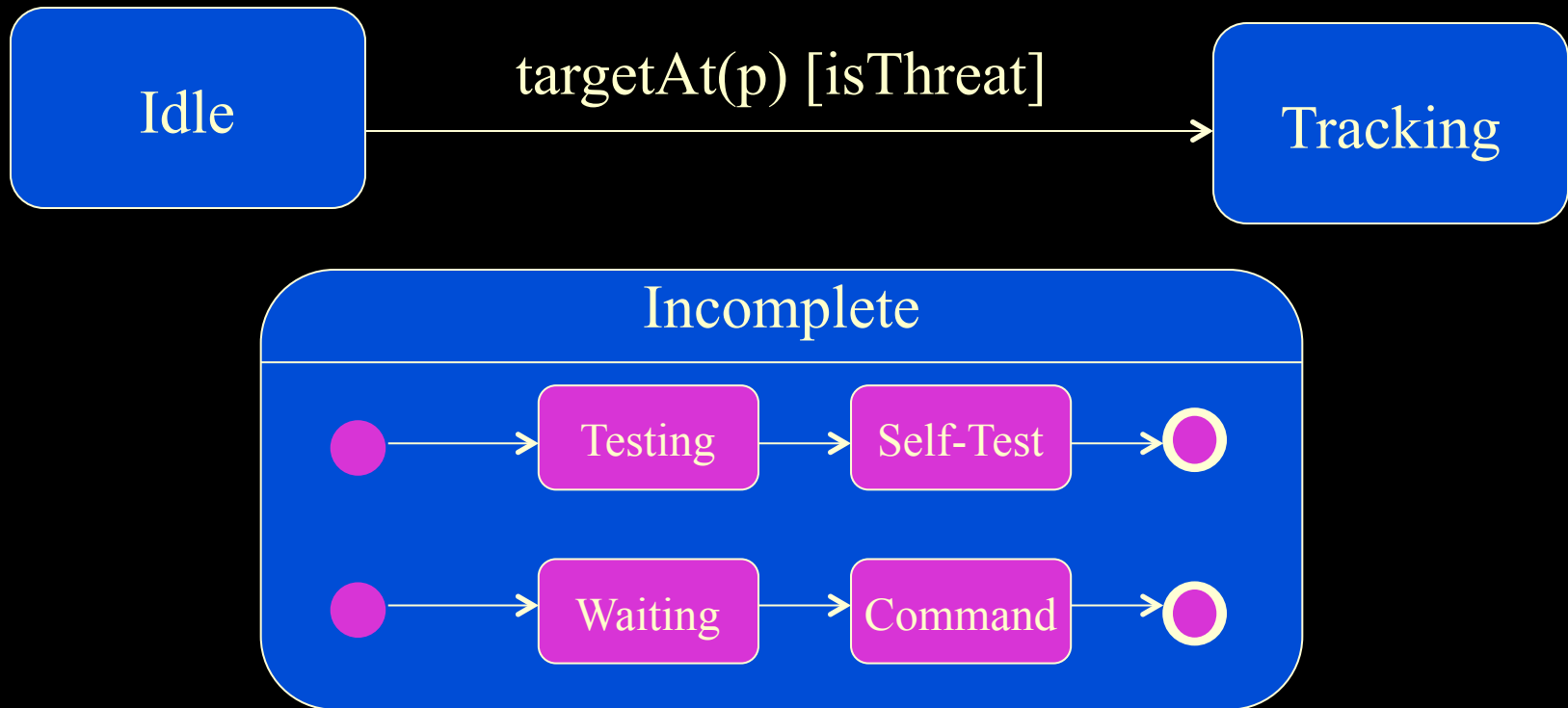
- Each object has a state machine which represents its behavior over time
- Objects make transitions between states
- State transitions are caused by events
- State transitions cause actions and/or activities to be executed
- State machines especially important for control objects (controllers)

State Diagrams



- Initial and final states
- Entry and exit actions
- Activities
- Guard conditions
- Sequential and concurrent substates

State diagram notation



Activity Diagrams



- Resemble flowcharts
- Useful for modeling workflows and details of operations
- Interaction diagram looks at the objects that pass messages; activity diagram looks at operations passed among objects

Carryovers from state diagrams



- activities
- actions
- transitions
- initial/final states
- guard conditions

Breaking up flows



alternate paths:

- branch
- merge



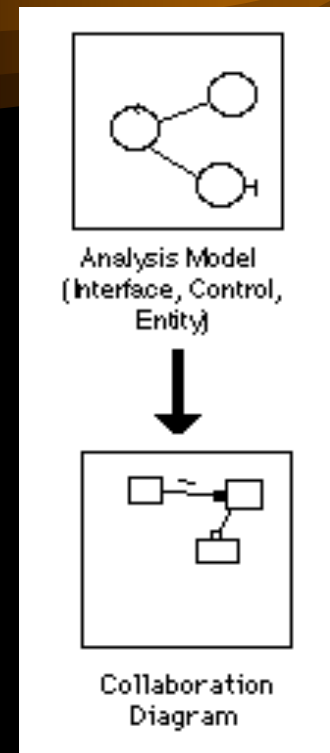
parallel flows:

- fork
- join



Collaboration Diagrams

- Collaboration diagrams show additional information, such as fields, parameters, and different kinds of messages
- Do one for each scenario (use case)
- 100% equivalent to sequence diagrams



What do collaboration diagrams address?



- How do objects communicate with each other?
- Object visibility
- Messages

How do objects communicate with each other?



- **Objects communicate with each other via messages**
- **Messages invoke methods on objects, which result in actions**
- **Objects recognize only a fixed, predefined set of messages**
- **Data contained in an object may only be modified by methods invoked by messages**
- **Loose coupling + high cohesion → good modularity**

Object Visibility



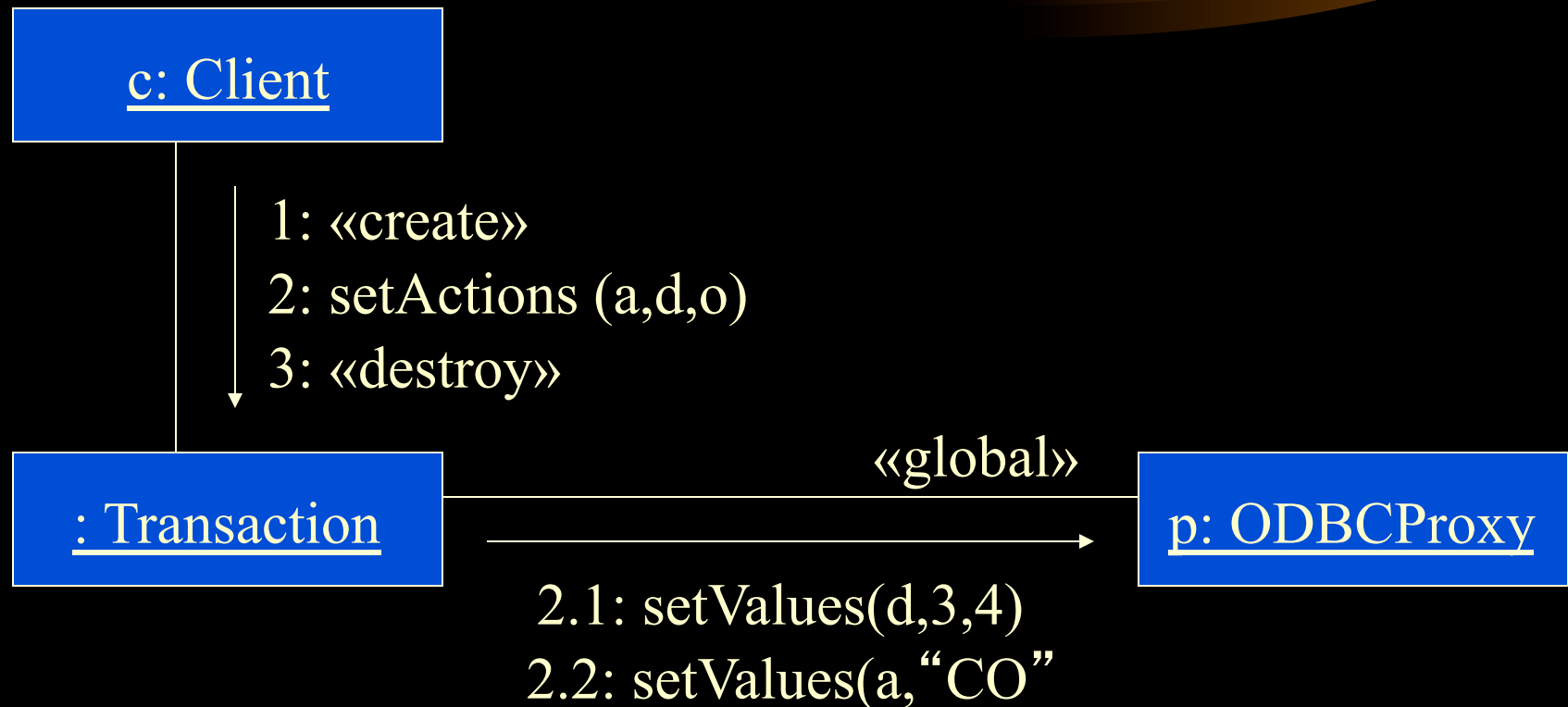
- **«self»** -- supplier object is part of client object (called “field” visibility by Booch)
- **«local»** -- supplier object is locally declared object in the scope of collaboration diagram
- **«global»** -- supplier object is global to client object
- **«parameter»** -- supplier object is parameter to some operation of client object
- **Objects can be shared (i.e., an object can play different roles within one collaboration)**

Messages



- Simple
- Synchronous
- Balking
- Timeout
- Asynchronous

Collaboration diagram notation



Component Diagrams

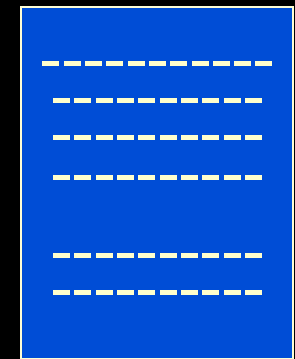
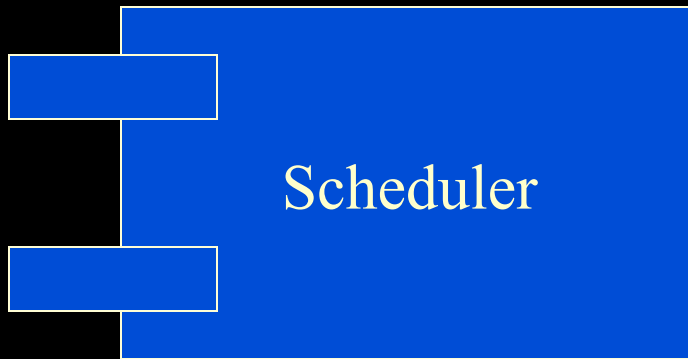


- Components are physical, replaceable parts of a system that conform to, and provide the realization of, interfaces.
- examples: dynamic link library (DLL), COM+ object, Enterprise Java Bean (EJB)
- Unlike classes, components are physical, not logical, and components have operations that are reachable only through their interfaces.

Uses of component diagrams

- modeling source code («file» stereotype)
- modeling executable releases («executable» stereotype)
- modeling physical databases («table» stereotype)
- modeling adaptable system ({location} tagged value)

Component diagram notation



signal.cpp

Deployment Diagrams

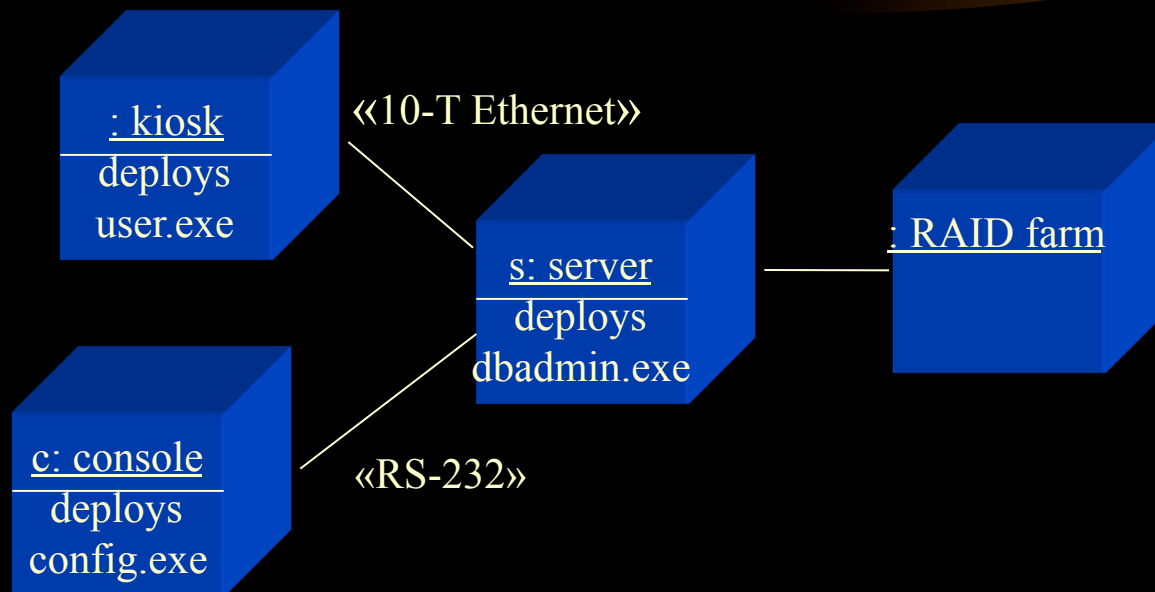


- A node is a physical element, which exists at run time, that represents some computation resource.
- This resource generally has at least some memory; it often has processing capability.
- Components are things that participate in the execution of a system; nodes are things that execute components.

Uses of deployment diagrams

- modeling embedded systems (to facilitate communication between hardware engineers and developers)
- modeling client/server systems (thin or fat client?)
- modeling fully distributed systems (Internet; CORBA/DCOM)

Deployment diagram notation



For further information



- EMAIL: doug@iconixsw.com
- WWW: <http://www.iconixsw.com>
- Phone: 310-474-8482