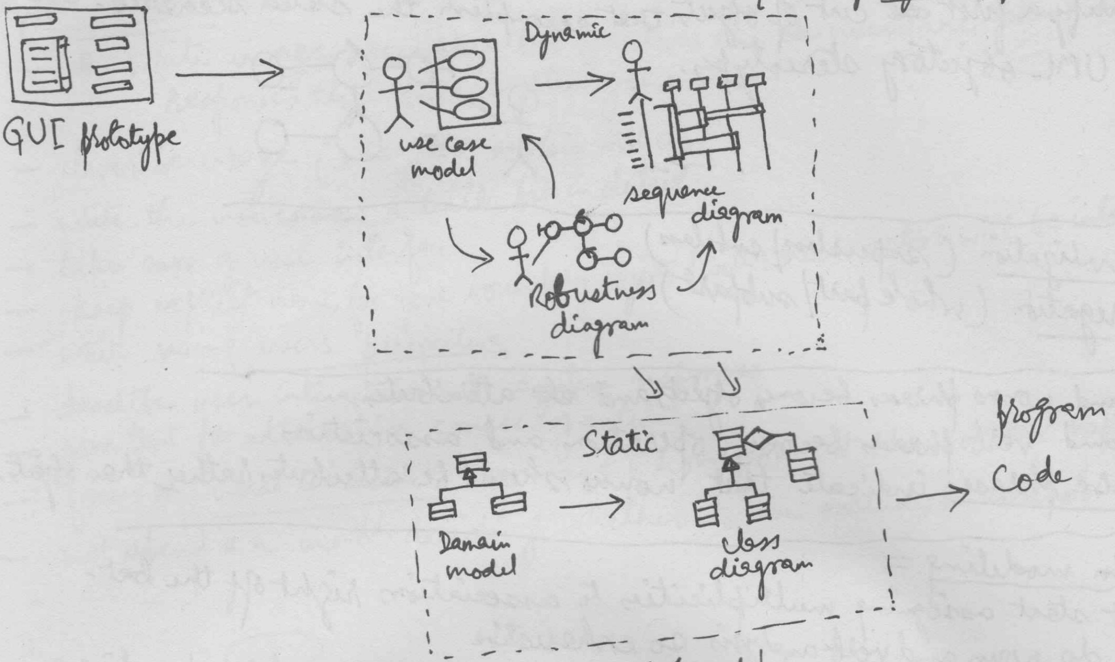
Applying Use Case Driven Object Modeling with UML: An annotated e-commerce Example

[redacted] [redacted]

* Describe system usage in the context of object model.
- domain objects (e.g., catalog, purchase order, ...)
- boundary objects (e.g., screens of the system, ...)



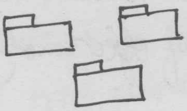
(streamlined approach to S/W development)

Process fundamentals:

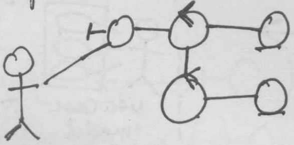
milestones for an OO process should include:
- team has identified and described all the usage scenarios for the system to build
- " " taken hard look for reusable abstractions (classes) that participate in multiple scenarios.
- " " thought about problem domain and has identified belonging classes
- " " varified all functional requirements of system
- " " carefully thought about system behavior's allocation to identified abstractions, taking into consideration good design principles (minimizing coupling, maximizing cohesion, generality, and sufficiency, and so forth)

There are four fundamental requirements of a process:
- flexible ~~to~~ for styles and kinds of problems
- support the way people really work (prototyping, iterative/incremental development, ...)
- ~~so~~ needs to serve as guide for inexperienced members
- " " expose precode products of a development effort to management

* organize the use cases into groups. Capture this organization in a package diagram.



* perform robustness analysis. For each use case:
  - identify a first ~~cut~~ cut of objects that accomplish the stated scenario. Use the UML objectory stereotypes.



---

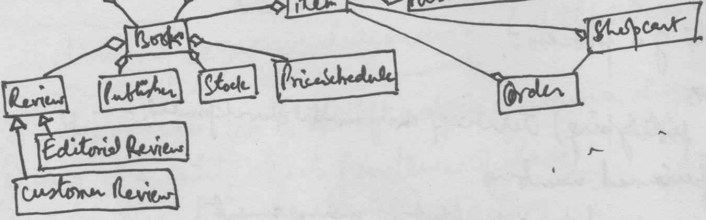
* generalization (superclass/subclass)
* aggregation (whole part/subpart)

---

* nouns and nouns phrases become objects and attributes.
* verbs and verb phrases become operation and associations.
* possessive phrases indicate that nouns should be attributes rather than objects.

---

* Domain modeling =
  - donot start assigning multiplicities to associations right off the bat.
  - " " do noun and verb analysis so exhaustive
  - " " assign operations to classes without exploring use cases and sequence diagrams.
  - " " optimize your code for reusability before making sure you have satisfied user's requirement.
  - prefer to use simple aggregation (has by reference) relation than composition (has by value)
  - ~~presume~~ donot presume specific implementation strategy without modeling the problem space
  - donot use hard-to-understand names for your classes.
  - donot jump directly to implementation constructs, such as friend relationships and parametrized class.
  - " " create one-for-one mapping between domain classes and ~~relationship~~ RDBMS tables.
  - " " perform "premature patternization", which involves building cool solutions, from patterns, that have little or no connection to user problems.

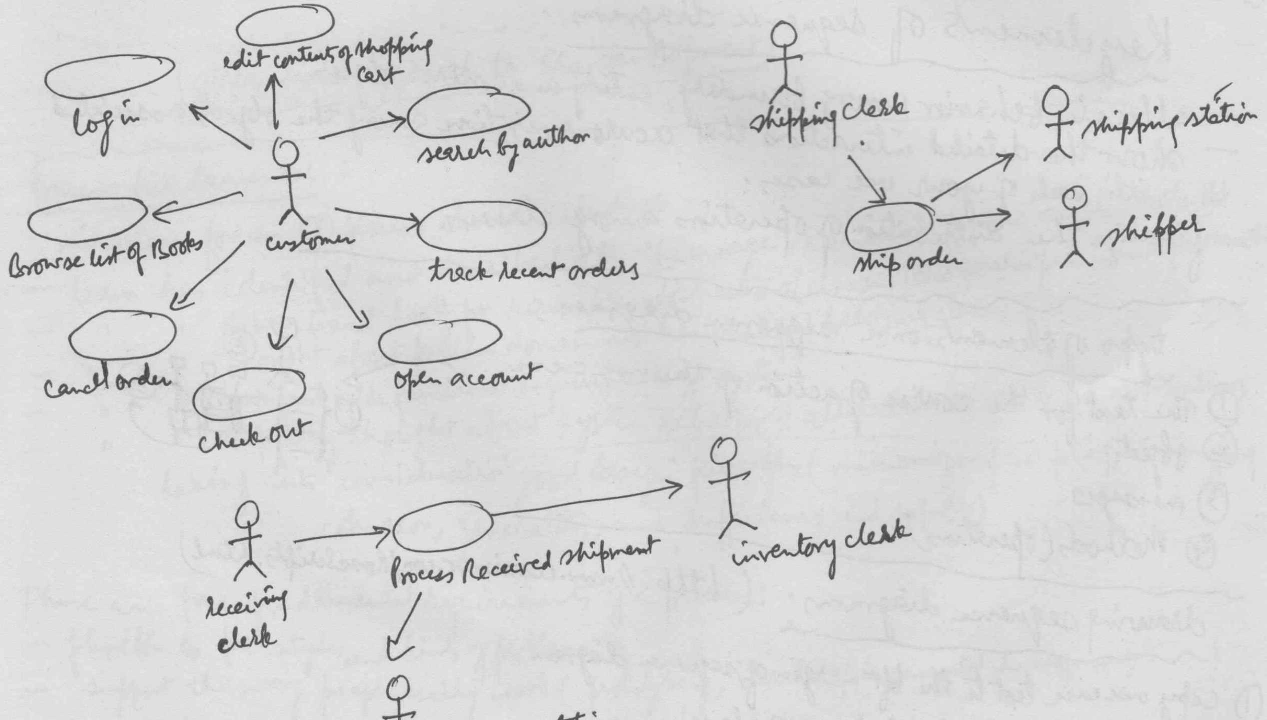


- The goal is to account for everything the user might do.

* each package should correspond with a chapter, or at least a major section, in your user manual.

* you should =

- ~~[illegible]~~ write usage scenario text (actions the ~~the~~ users are taking and the responses that the system generates.)
- describe usage (what the system will do)
- write the use cases a little bit in detail.
- take care of user interface
- keep explicit names for your boundary objects (with which actors will be interacting)
- write using user's perspective
- describe user interactions and system responses
- give text for alternative courses of action
- focus on what is "inside" a use case (how you get there or what happens afterwards)
- not spend ~~[illegible]~~ a month deciding whether to use includes or extends.



Robustness diagram ~~symbols~~ symbols:

⊢○ boundary object (actors use in communicating with the system)

○ entity object (objects from domain model)

↻ control object (controllers that ~~glu~~ "glue" between boundary objects and entity objects)

Rules: ① Actors can only talk to boundary object. (웃 ⟶ ⊢○)

② boundary objects can only talk to controllers and actors. (⊢○ ⟷ ↻)

③ entity objects can only talk to controllers. (↻ ⟷ ○)

④ Controllers can talk to boundary objects, entity object, and other controllers, but not to actors. (↻ ⟷ ↻)

boundary object, entity object — nouns

controller — verb

} nouns cannot talk to other nouns. verb can talk to ~~other~~ nouns / verbs.

Key elements of sequence diagrams:

- allocate behavior among boundary, entity and control objects
- show the detailed interactions that occurs over time among the objects associated with each of your use cases.
- finalize the distribution of operations among classes.

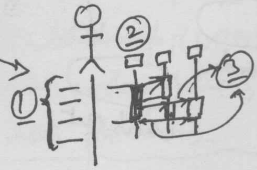
types of elements on a sequence diagram:

① The text for the course of action of the use case

② objects

③ messages

④ Methods (operations)



drawing sequence diagrams: (http://www.iconixsw.com/RoseScripts.html)

① copy use case text to the left margin of sequence diagram

② add the entity objects (instance of class)

③ add the boundary objects ~~&~~ and actors.

④ which method go on which classes (decide).
~~work~~ work through the controllers, one at a time, and figure ~~out~~ out how to allocate the behavior among the collaborating objects.