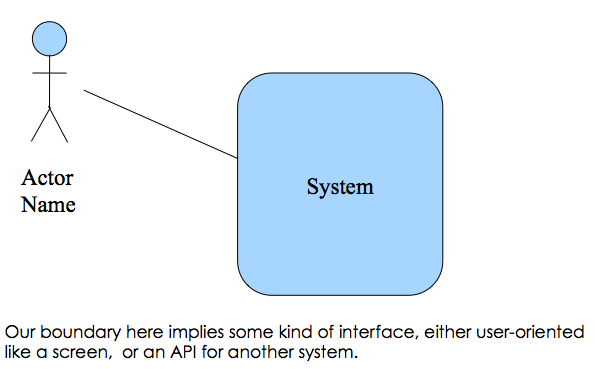
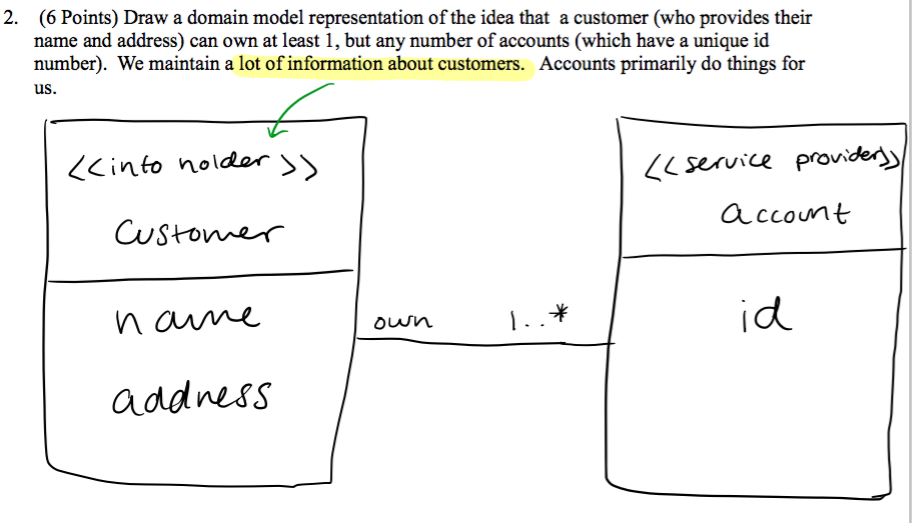
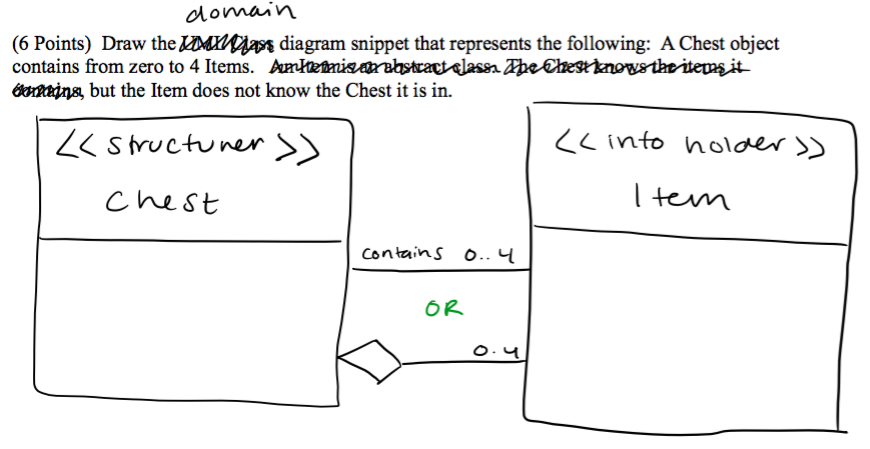
* Exam Questions
  + Time boxed = always delivering something on time even if cut functionality
  + Iterative development = dividing the project into smaller pieces completed as mini-projects
    - Only fully detail the use cases for sprint backlog before starting an iteration
  + Use cases
    - Specify interactions b/t external actors and system
    - Describe what has to be true for the use case to execute
    - Describe key stakeholders and their goals
  + SCRUM, product backlog contains
    - Prioritized list of all the features the app will have
    - The hours left to work on each feature
    - Total hours left on entire project
  + Spring backlog
    - List of all tasks scheduled for this sprint
    - Name of person responsible for each task
    - Hours left to work on each task
  + Gradle
    - Allows team to perform all tasks related to build and deployment w/ only one command
    - Minimizes configuration requirements for new team members
    - Allow automated runs of tests and deployment
  + Git
    - Allow multiple team members to modify files
    - Provide access to the current code base for everyone on the team
    - Allow the creation of different versions of the application through branching
  + Characteristics of Objects
    - Knows thins
    - Does things
    - Collaborates w/ other objects
    - Makes decisions
  + Use Case Titles
    - Verb + direct object
    - “authenticate user”
  + Main success scenario
    - First step
      * Trigger event, kicks off whole chain of events
      * Something external actor does
    - Should contain the path which has no exception conditions
    - Detail the interactions b/t the User and the System
    - Does NOT contain alternative descriptions when user has an option, that’s alternative paths
  + Valid architectural styles
  + Domain models
  + Robustness diagrams
  + UML
    - Actors is something w/ behavior and something outside the system we are building
    - Primary, Supporting, Offstage
  + Domain model
    - Contains real world conceptual classes
    - Contains relationships b/t the conceptual classes
    - Contains classes that the customer would understand
  + Strategies for finding objects in the domain model include
    - Analyzing the nouns the problem description
  + Domain Models
    - Information holder (keep facts)
    - Service provider (perform operation)
    - Interfacer (support internal/external communication)
    - Structurer (maintain relationships)
    - Controller (actively direct activities)
    - Coordinator (delegates work, reacts with rules)
  + OOA
    - Most important thing about analysis phase is to **understand the customer problem**

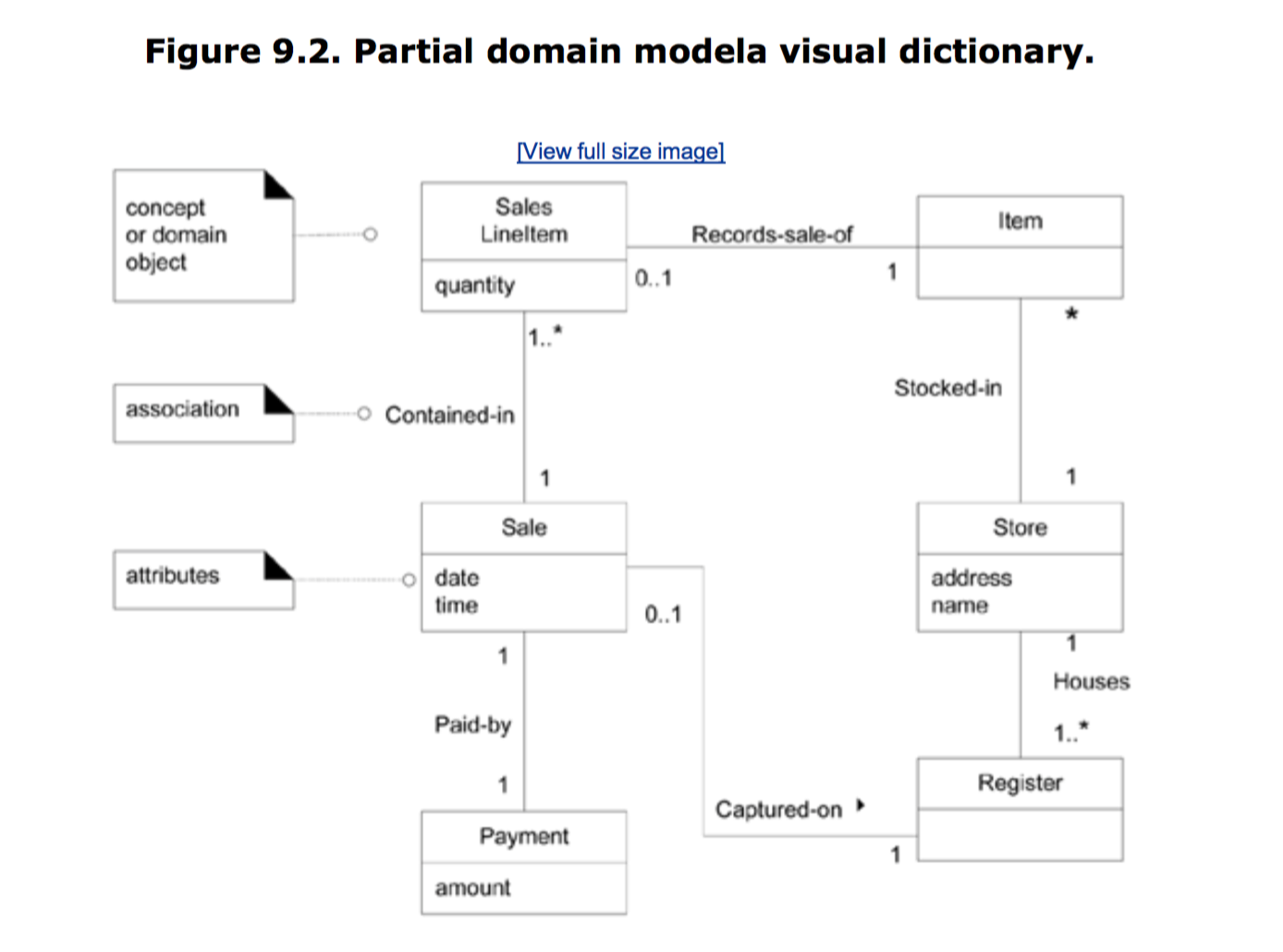
Software Development

* + Waterfall
  + Agile
    - Individuals and interactions over process and tools
    - Working software over comprehensive documentation
    - Customer collaboration over contract negotiation
    - Responding to change over following a plan
  + Iterative delivery
    - At end of iteration something is delivered
  + SCRUM
    - Timeboxed iterations called sprints
    - Each sprint ends w/ ready deliverable
    - Use product and sprint backlogs
* Introduction to UML and Use Cases
  + Context Diagram

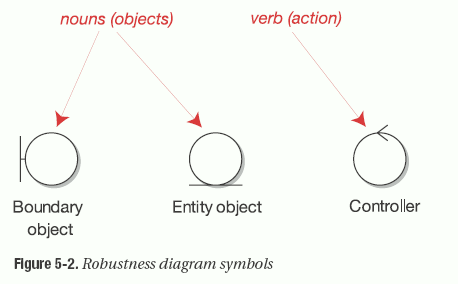


* + Actors
    - Primary
    - Supporting
    - Offstage
  + Use Cases
    - Verb + direct object
    - Primary actor
  + Main Success
    - Happy path
    - Begins w/ trigger event
    - User does something, system does something
    - Avoid implementation in description
  + Alternative paths
    - Alternative actions
    - Errors system might have to handle
    - Numbered w/ pattern **(look at book)**
  + Pre and post conditions
* OO Analysis – Domain Models

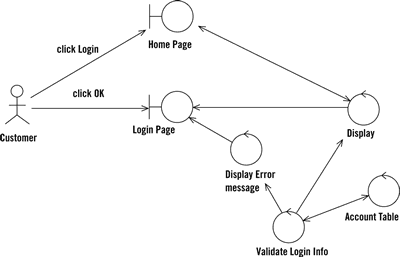




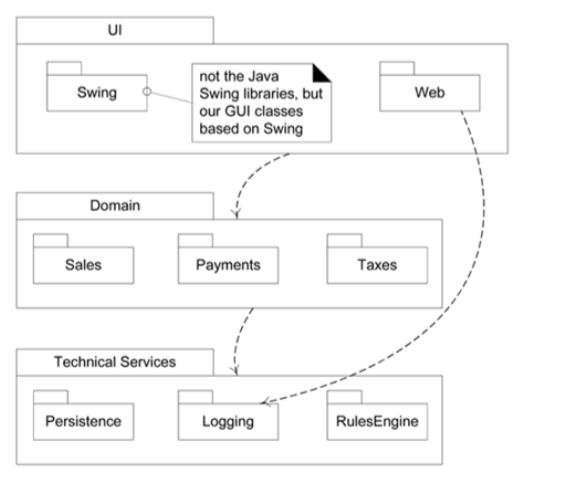
* + Role Stereotypes
    - Information holder (keep facts)
    - Service provider (perform operation)
    - Interfacer (support internal/external communication)
    - Structurer (maintain relationships)
    - Controller (actively direct activities)
    - Coordinator (delegates work, reacts with rules)
* Use Case Robustness Diagrams
  + Tie together the use case and domain model
  + Components
    - Actor – stick figure from use cases
    - Boundary elements – represent UI screens, APIs and other interface elements
    - Control elements – glue between boundary and domain elements. Contains business logic. May or may not become classes in the design
    - Domain elements or entities – things from the domain model that represent data in the system
  + Icons
    - Actor – stick person



* + - Rules
      * Actors can only talk to boundary objects
      * Boundary objects can only talk to controllers and actors
      * Entity objects can only talk to controllers
      * Controllers can talk to boundary objects, entity objects and other controllers but not actors
      * Show alternative diagrams
      * Don’t put design into diagram



* Software Architecture
  + Architectural views
    - A view is a representation of a set of system elements and the relations associated with them.
  + Common architectural styles
    - Client – server (thin/fat)
    - Peer-to-peer
    - Blackboard/shared memory
    - Pipe and filter
    - Model – view – controller
    - N-tier
    - Layered
    - Implicit invocation/event-driven
  + Control Styles
    - Centralized – a few objects make all the decisions and direct other objects when to do specific tasks
    - Delegated – there are several coordinator objects which locally direct a few objects
    - Dispersed – Objects work without central direction to accomplish tasks
  + UML Package Diagram



* Android
  + Operating System
  + Activity – visual representation of an android application. Uses views and fragments to create a user interface
  + Fragment – optional constructs inside activities to support display on different devices or provide reuse
  + View – interface widget like a button or text field
  + View group – basically a layout manager to establish where the view will go on the actual screen
  + Intent – an async message that allows the application to request functionality from other components
  + Service – a background task that does not require a UI component
  + Content provider – an interface to application data. Usually the on-board SQLite database
* **Model:** Information for the application.
* **View:** Visual Representation of the Information.
* **Controller:** Mechanism to allow user to change information or view.
* **OOA vs. OOD**
* In analysis, we are mostly concerned with the DOMAIN MODEL. What are the objects in the domain and how do they collaborate. We do not think about how things are done, only what has to be done.
* In design, we need to integrate an APPLICATION MODEL. What objects do I need to add to get this thing to run on a computer and to be realized in some programming language
* Cardinality (Multiplicity)
* 1 - one
* 0..1 – zero to one
* 0..\* - zero to infinite
* 1..\* - one to infinite
* \* - infinite
* **Design Patterns**
* **~Information Expert** - Assign a responsibility to the class that has the information necessary to fulfill the responsibility.
* **~Singleton** - want to limit the application to only one instance of a particular class, but need global access to that class. Used to control access to key resources. ToDo: override new, make static accessor method. private constructor. Caveats: anyone has access reduces reuse
* **~Factory** - Define an interface for creating an object, but let subclasses decide which class to instantiate. Allows a class to defer instantiation to subclasses.
* Think of a multi-document application framework. An application object may know when an object needs to be created, but not which object. How do we create the correct object when needed?
* Can also be used when a complex initialization of objects is necessary, for instance when aggregation is heavily used.
* Can also be used to take advantage of memory-optimization like object pools, cached objects, etc.
* **~Command** - Encapsulate commands in objects, so we can queue them, undo them or make macros.
* **~Prototype** - I need to create customized objects without knowing the exact class. “I want something just like this, except with new instance values”...Solution – clone/copy the object and initialize the data.
* **~Observer** - Allows objects to dynamically register their interest in being notified of any changes that occur in the state of the observed object.
* **~Null Event** - An alternative to using null to indicate the absence of an object to delegate an operation to. Can eliminate a test for null by using an object that doesn’t do anything. This allows objects to always behave the same way, i.e. use an object without any additional tests required.
* **~Delegation** - A way to extend and use the functionality of a class by writing an additional class with added functionality that uses instances of the original class to provide original functionality. An alternative to inheritance. Used to handle multiple roles. In Java “event sources” delegate responsibility for processing an event to an “event listener."
* **~Strategy** - Allow selection of algorithm to vary by object and time.
* **~State Pattern** - Like Strategy, but what if data had to vary instead of the behavior? Or what if needed to simulate state machine?...Solution: Encapsulate changing data in a class or each state of state machine in a class
* **~Template Method** - I have a method, where some steps are common to all subclasses, but others are specialized....Solution. Create a method with concrete steps, but that calls abstract methods for the specialized parts.
* **~Flyweight** - I have a bunch of classes, but I need to minimize the number of objects I am using. Class has only those attributes shared amongst all instances.
* **~Visitor** - I need to apply different operations to a collection of objects. I want to centralize these operations. I want to reduce coupling. For example in a word processor, grammar check, spell check, table of contents builder, outliner all need to traverse the document. Solution: Implement a visitor object that knows how to traverse the object in question (a document in our example). Subclass the visitor for each specialized operation.
* **Basic Principles (SOLID)**
* **S = Single Responsibility** - Each class should have a single overriding responsibility (High Cohesion). Each class has one reason why it should change
* **O = Open/Closed** - Objects are open for extension but closed for modification. Extension via inheritance, polymorphism.
* **L = Liskov Substitution** - Subclasses should be substitutable for their base classes. Consider behavior, not just “is a”. Square/rectangle problem. Preconditions cannot be strengthened. Postconditions cannot be weakened.
* **I = Interface Segregation** - Don’t make large multipurpose interfaces – instead use several small focused ones. Don’t make clients depend on interfaces they don’t use. Class should depend on each other through the smallest possible interface.
* **D = Dependency Inversion** - Depend on abstractions, not concretions. Program to interfaces not implementations. Program to most abstract class possible. Hollywood Principle