An Introduction to Object Oriented Analysis and Design using UML

Course Goals & Outline

- Provide an introduction to Object Oriented Analysis and Design (OOAD)
    - Concepts
    - Terminology
    - Techniques
- Provide an overview of the Unified Modeling Language
- Show how to apply basic OOAD techniques to a software engineering process

This Course Will Not:

- Make you an expert in OOAD
- Provide instruction on all aspects of the Unified Modeling Language
- Provide a software engineering process
- Turn you into a system architect
- Address OO programming

OOAD Benefits

- Improves team communications by providing a common design language & notation
- Provides a tool set for supporting a software engineering process
- Allows greater participation in the design process

How does OOAD relate to a software engineering process?

- A process tells us *who* does *what* and *when*, OOAD shows us *how*
- Provides a structure for design artifacts
  - Scope/Vision – Use Case Diagram
  - Conceptual Design - Use Cases
  - Physical Design – Sequence & Class Diagrams
  - Implementation – Deployment/Component Diagrams

OOAD is not new

- Over 200 years old
  - Used in early manufacturing at the turn of the 19th century
  - Enhanced by people like Henry Ford
  - Now perfected in the manufacturing and engineering worlds

OOAD is (relatively) new in software development

- A brief history of software development:
  - Large, monolithic systems combining data and application
  - Large database with separate logic
  - Modular data and logic

Terminology & Concepts

- Defining the term "object oriented"

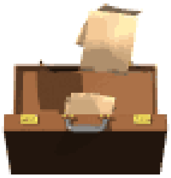
What is an Object?

A thing with which we interact

- It does something

  and/or
- It knows something

Objects in Our Business World



Files

Competitors

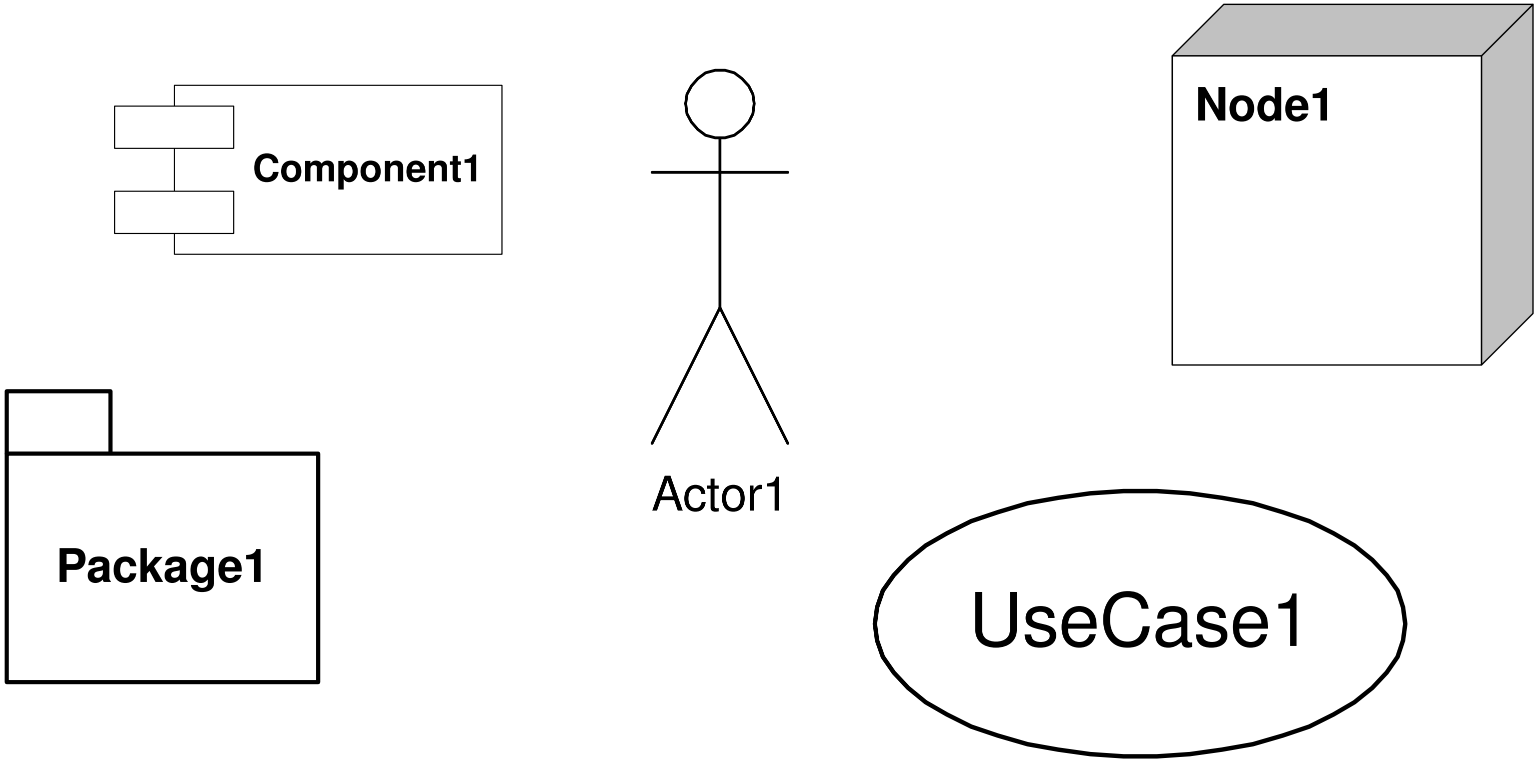
Employees

Assets

Customers

Systems

Objects in Our System World



Component1

Actor1

Node1

Package1

UseCase1

My object is not your object

- What you recognize as an object may not be what others recognize as an object……

The CEO’s objects:

Financial System

Marketing Department

Board of Directors

Takeover Target

Stock Holders

The CEO’s objects:

Financial System

The CFO's objects:

Accounts Receivable

General Ledger

Payroll System

Cash Account

The CFO’s objects:

Payroll System

The Payroll Clerk’s objects:

Timesheets

Employees

Pay Grades

Paychecks

Union Rules

The World View

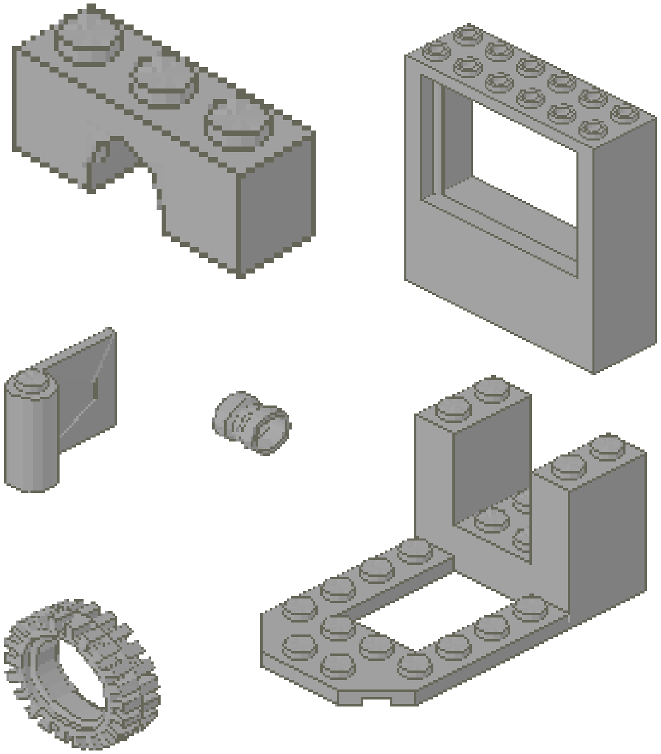
- Is different depending upon who you are
- Goes from high-level abstractions to low-level realizations:
  - A universe, solar system, Earth, North America, USA, California, Irvine, 123 Main Street, Suite 292, my cubicle, my coffee cup
  - Video rental stores, Blockbuster, Inventory, Action Movies, “Terminator”

Why do we care?

- We can use objects to describe, or model, the system we are trying to create
  - and in terms that are relevant to the domain
- Objects allow us to decompose a complex system into understandable components
  - and that allow us to build a piece at a time

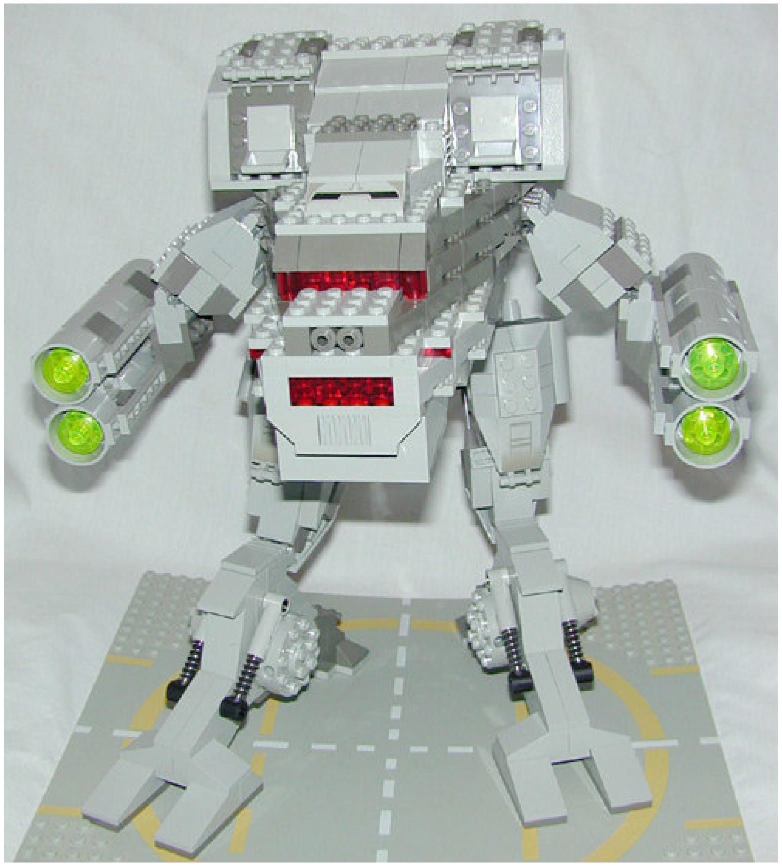
What is "Object Oriented"?

- Simplicity thru self-contained objects
- Complexity thru integration
- Interchangeability thru frameworks



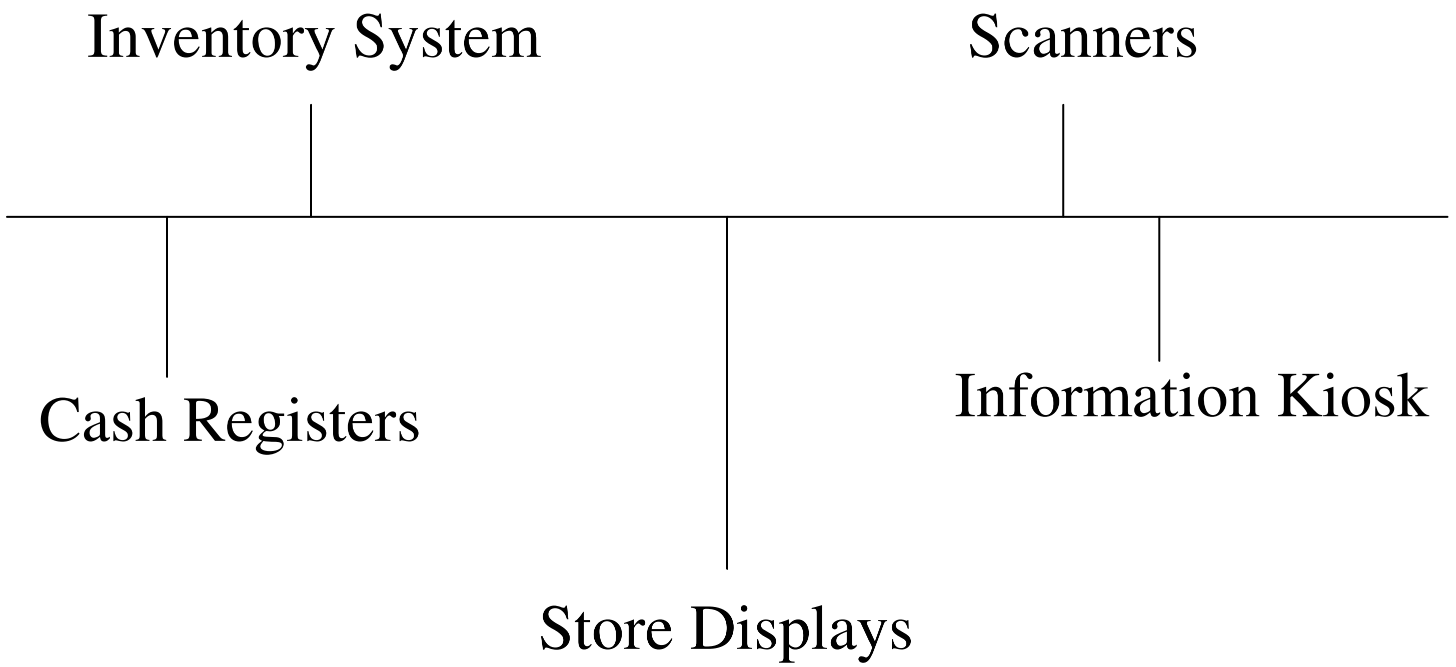
What is "Object Oriented"?

- Simplicity thru self-contained objects
- Complexity thru integration
- Interchangeability thru frameworks



Simple parts; complex whole

Video Rental Company
Framework from clerk's perspective


Inventory System

Scanners

Cash Registers

Information Kiosk

Store Displays

The OOAD Objective

- To identify the relevant objects in the subject domain
- To drill-down to relevant sub-objects
- To discover patterns and relationships
  - so that efficient object groupings can be made providing effective system architectures

Benefits of Object Technology

- Re-use
  - Shared components
- Stability
  - Interchangeable parts
- Reliability
  - Reduced complexity of individual components
- Integrity
  - Protected data & code
- Iterative Modeling
  - vs. interpretation & recreation

The Old Way

- Complex, single mainline code with multiple branches
- Brittle database schemas
- Maintenance by patch rather than refinement

The Old Way

- Complex, single mainline code with multiple branches
  - Single flowcharts written with scores or hundreds of elements, branches, etc.
- Brittle database schemas
- Maintenance by patch rather than refinement

The Old Way

- Complex, single mainline code with multiple branches
  - Single flowcharts written with scores or hundreds of elements, branches, etc.
- Brittle database schemas
  - Massive table structures supporting entire systems
- Maintenance by patch rather than refinement

The Old Way

- Complex, single mainline code with multiple branches
  - Single flowcharts written with scores or hundreds of elements, branches, etc.
- Brittle database schemas
  - Massive table structures supporting entire systems
- Maintenance by patch rather than refinement
  - Logic too complex to re-evaluate during a maintenance effort

Have we found the Silver Bullet to Analysis and Design?

Not quite………

The Dark Side of Object Technology

- New vocabulary and thought process
- Full benefits yet to be realized
- Ease of programming offset by complex design
- Code can still be too complex and poorly designed
- Requirements *still* constantly change

OOAD Concepts & Definitions

- Objects
- Behaviors & Responsibilities
- Classes
- Instantiation
- Properties

Objects can be many things…..

- Concrete real world things
  - Customers, inventory, invoices
- Conceptual things
  - Sales transaction, order processing

Objects have behaviors and responsibilities

Behaviors & Responsibilities

- Perform actions that have an outcome
  - Tell us about itself
  - Change itself
  - Initiate activities with other objects
- Have defined services
  - Have a “contractual obligation” with published services

Behaviors & Responsibilities Video Tape Object

- Perform actions that have an outcome
  - Will provide description of the movie
  - Will track shelf location
  - Change its status from “rented out” to “over due” to “sold”

Classes – Object Groupings

- Related groupings of objects with common responsibilities and behaviors

- Bob, Ted, and Sally are employees
- USA, England, and Spain are countries
- 112367, 432856, and 883210 are accounts
- Terminator, Star Wars, 2001 are movies

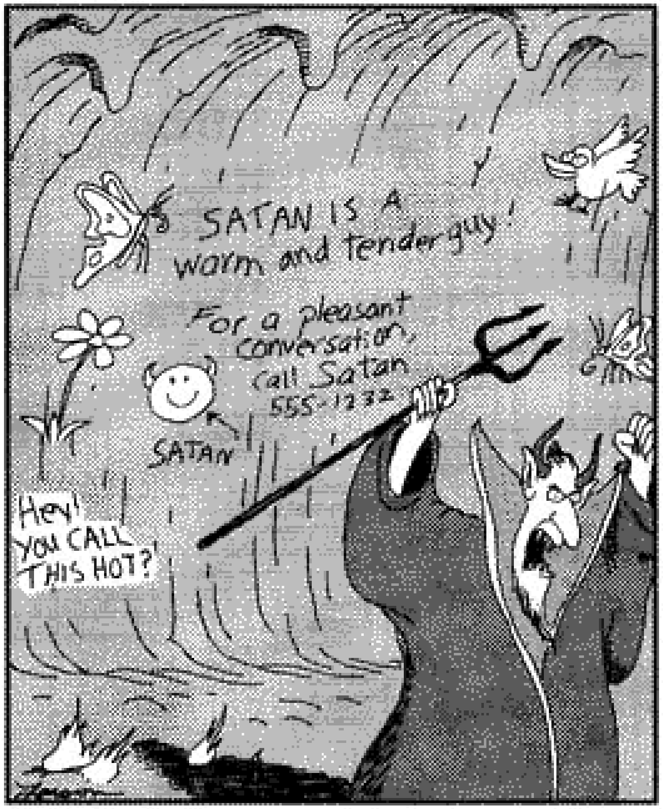
Instantiation

- An object is an instantiation of a class
  - When I hire a new employee “Joan”, she is an instantiation of the class “employee”
  - When you instantiate an object, you create an object which is patterned after a specific class
    - Casablanca is an instantiation of the class “movie”
- Class is the mold
  - An object is what comes out of the mold

Class Qualities

- High Cohesion
  - The internal relationship of behaviors and knowledge is focused and controlled
    - Reduces code "bloat"
- Low Coupling
  - The dependency between classes is limited and controlled
    - Improves re-usability and maintainability

That's the basics…..but the devil is in the details! Let's talk modeling theory…..


SATAN IS A warm and tender guy!
For a pleasant conversation, call Satan 555-1232
SATAN
Hey! You CALL THIS HOT?!

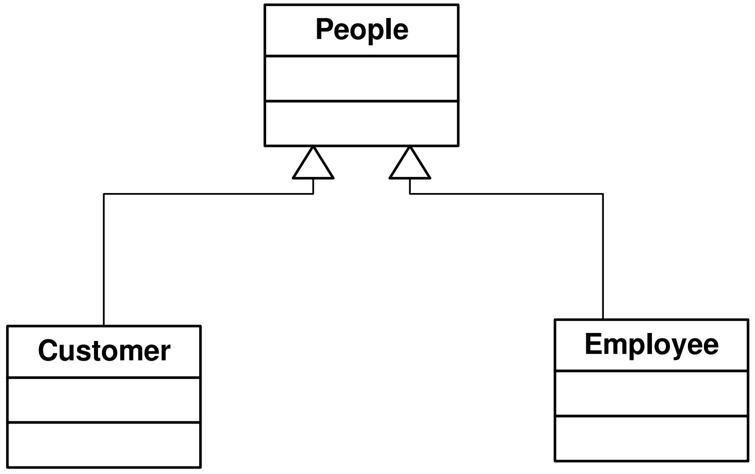
Graffiti in hell

Object Properties - Why

- Allow us to model an object’s roles and responsibilities
- Provide us with ways to communicate how objects are related

Communication "shorthand"….

- Employees and Customers are both kinds of people. They do "people" things but also have unique behaviors and responsibilities of their own.

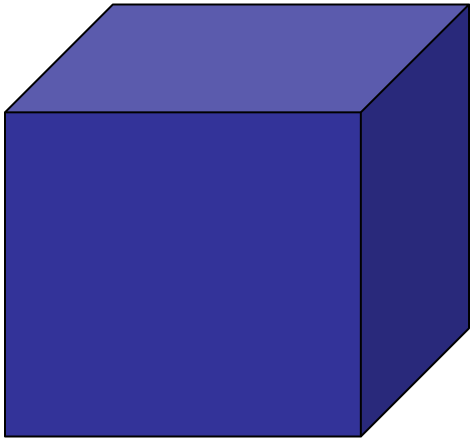


Object Properties - What

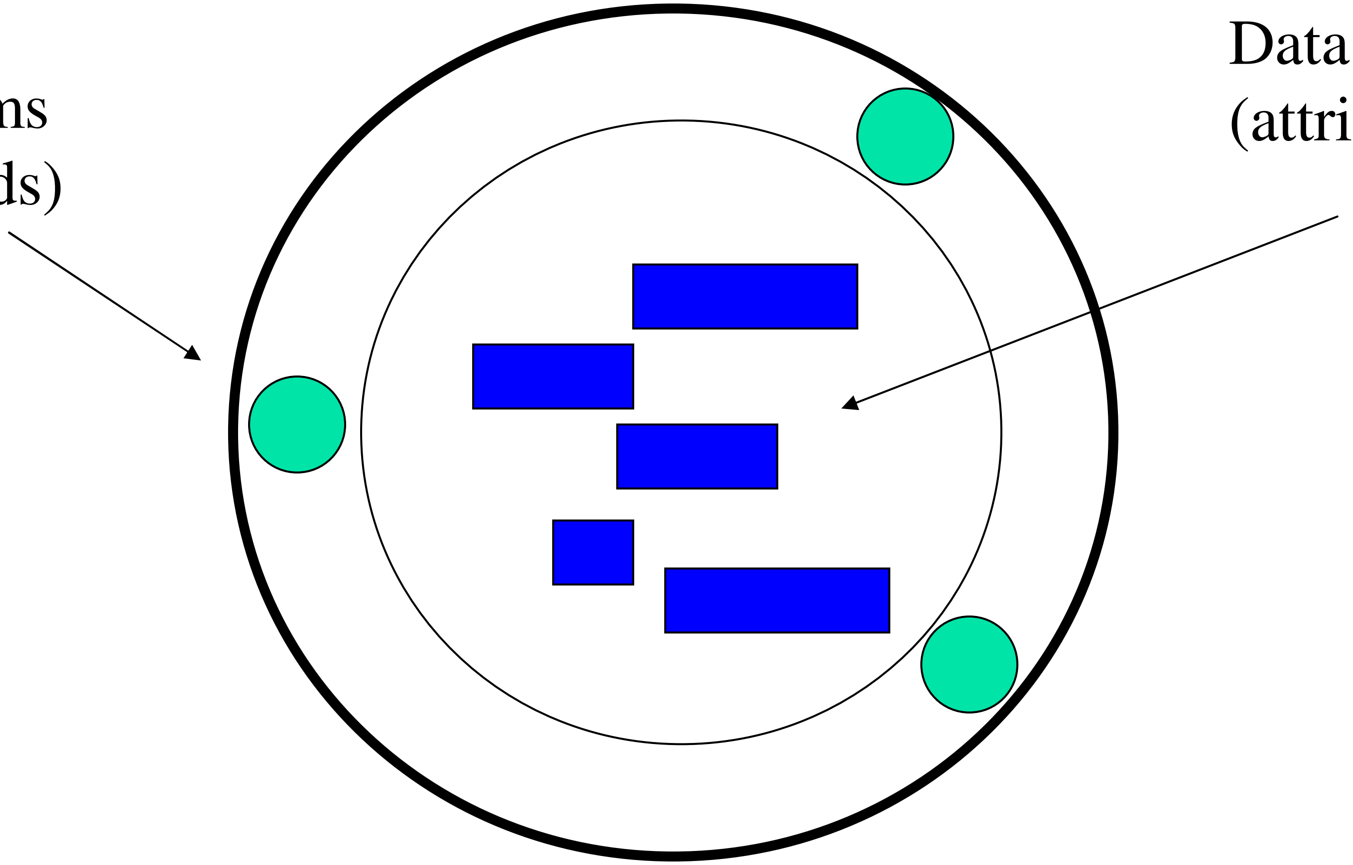
- Encapsulation (internal)
- Relations (external)
  - Association
  - Inheritance
  - Abstraction
  - Polymorphism

Object Properties - Encapsulation

- Objects are "black boxes" to each other
- They tell us:
  - What they know
  - What they will do
- How they do that is up to them!



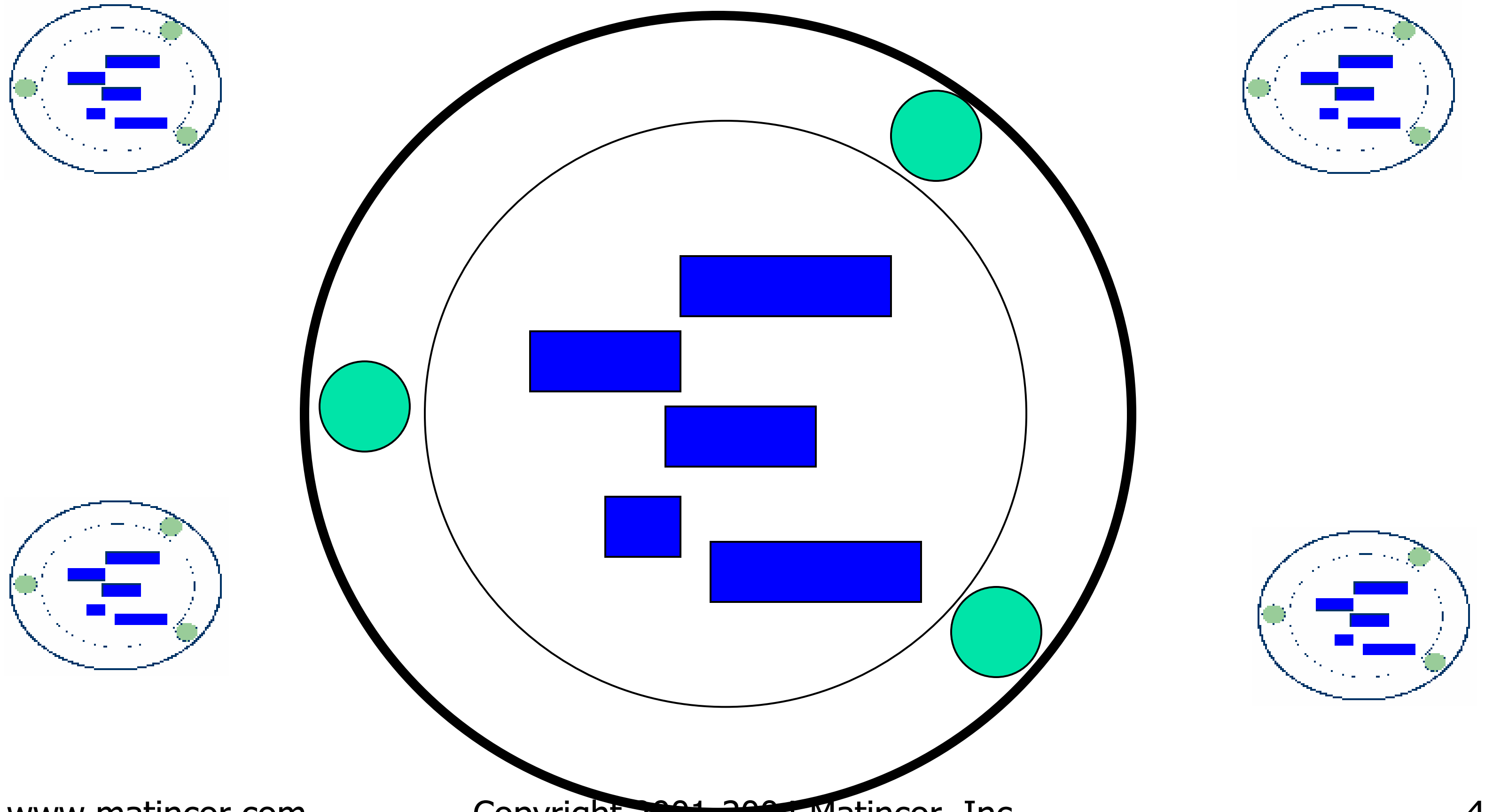
Object Properties - Encapsulation



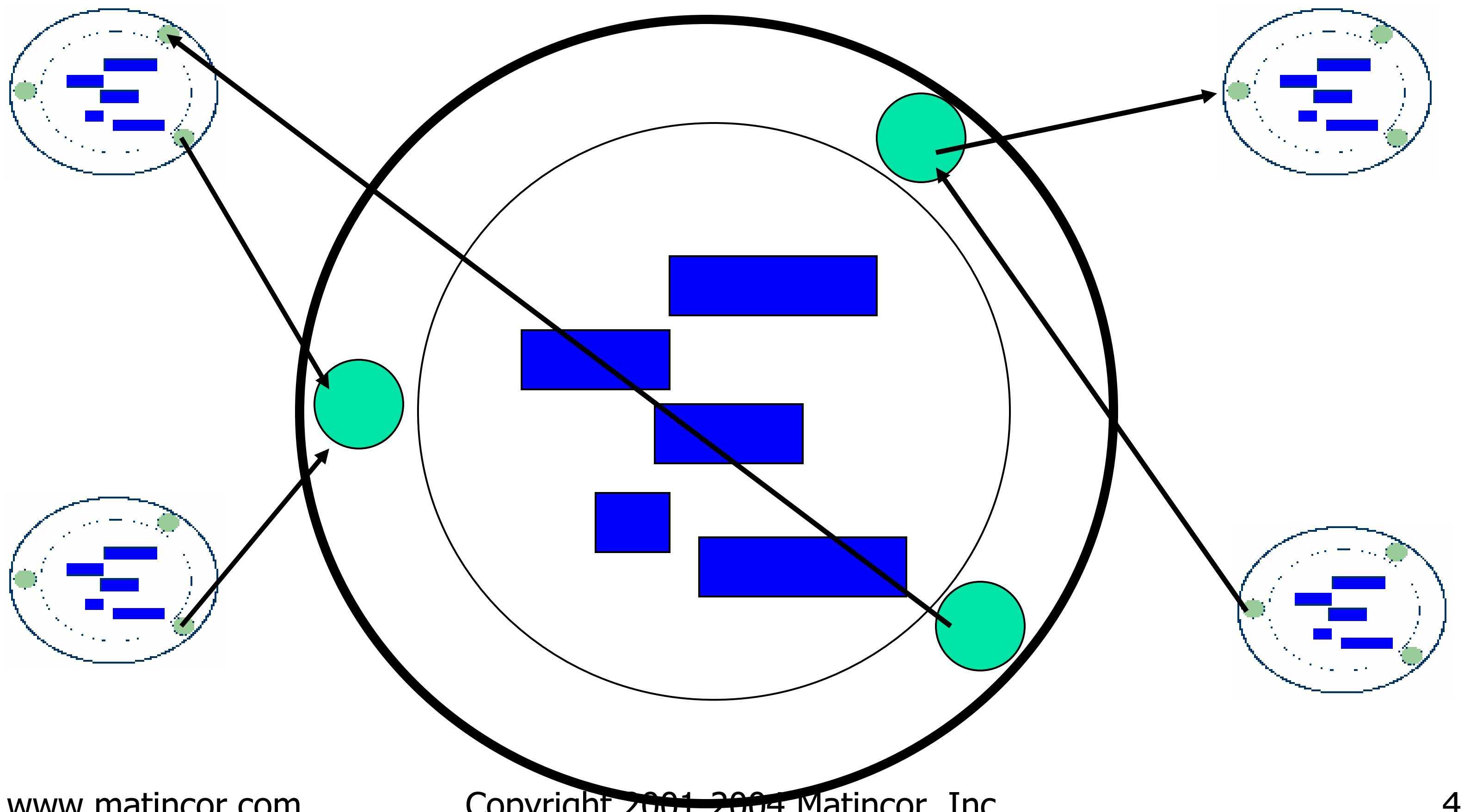
Programs (methods)

Data (attributes)

Object Properties - Encapsulation



Object Properties - Encapsulation



Encapsulation - Example

- A “person” object includes:
  - Attributes (data)
    - Name, address, birth date, phone number, marital status
  - Methods (programs)
    - Change address, calculate age, modify state (married vs. single), etc.

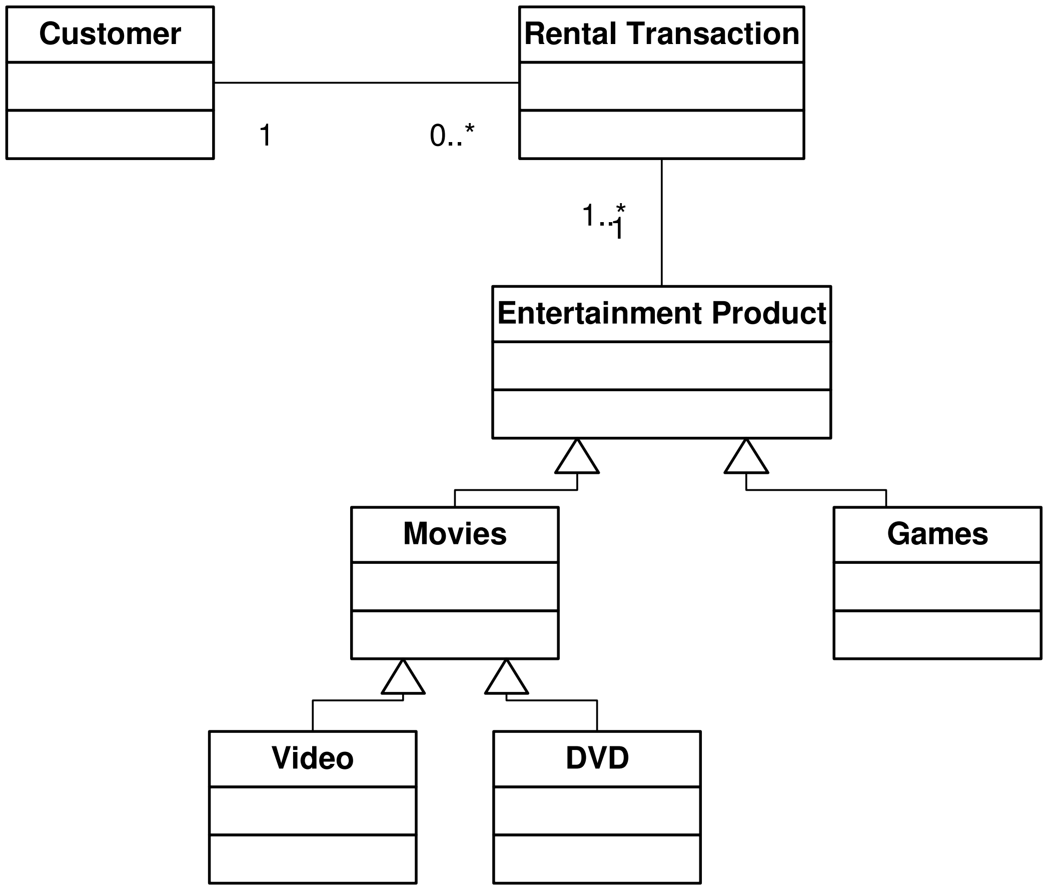
Encapsulation - Example

- A “person” object includes:
  - Attributes (data)
    - Name, address, birth date, phone number, marital status
  - Methods (code)
    - Change address, calculate age, modify state (married vs. single), etc.
  - Operations (doorway to methods)
    - A way to access methods (visible functions)
  - Interface
    - Collection of operations which access methods

Relations

- When objects interact with each other, they have a relationship
- Systems are defined by objects and their relationships

Relations – Video Store



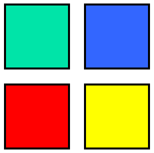
Object Associations

- Objects can collaborate with other objects
  - person can rent video tapes
- Objects can be closely tied to other objects
  - customer can have multiple accounts
- Objects can combine to form super-object
  - wheels + engine + body = automobile

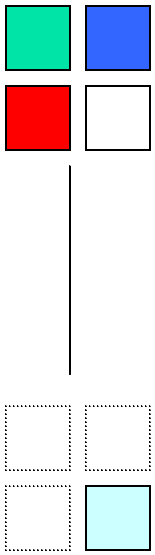
Object Properties - Inheritance

- Allows common operations and attributes to be shared among objects
  - Customer, employee, vendor can all be part of the person class
- Reflects parent / child relationships
  - Rental movie has several types: video, DVD, 8mm, etc.
- Usually denotes an “is a” relationship

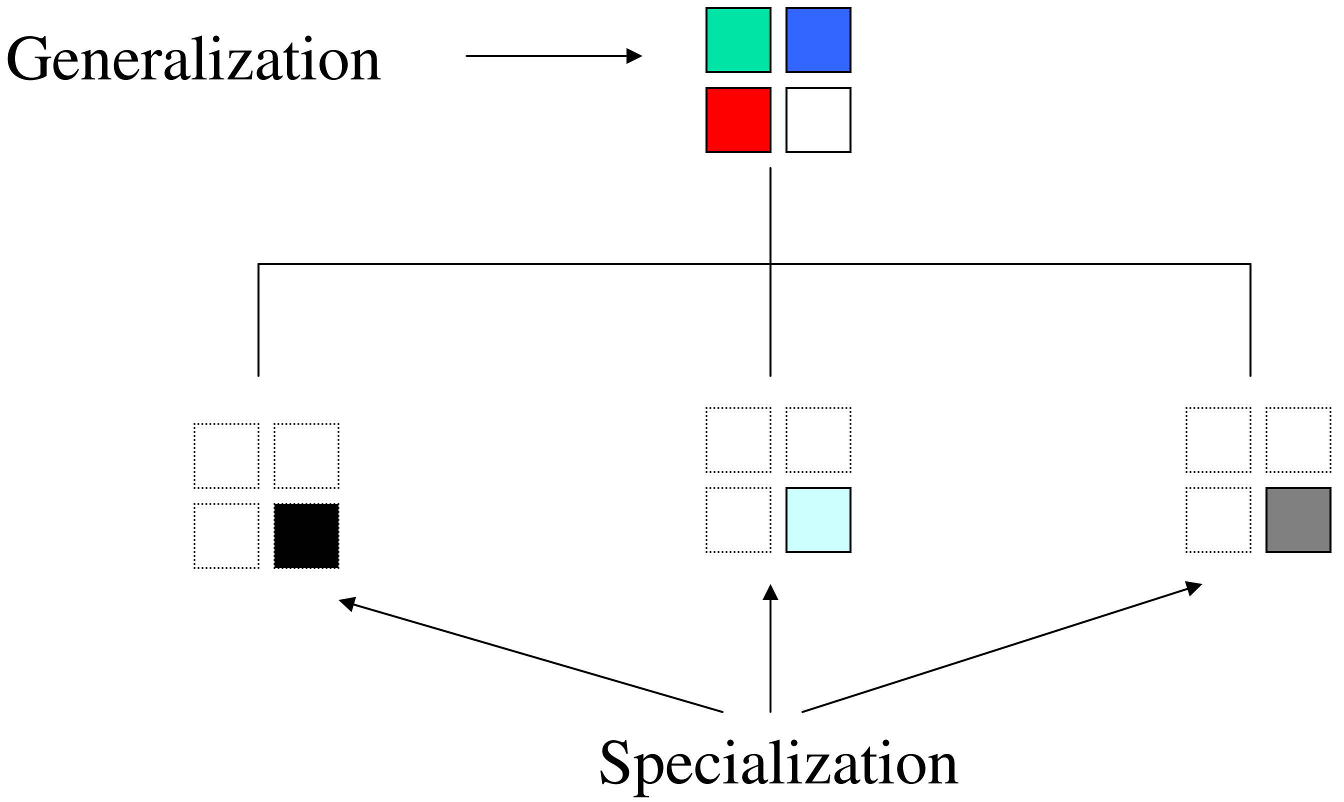
Object Properties - Inheritance



Object Properties - Inheritance



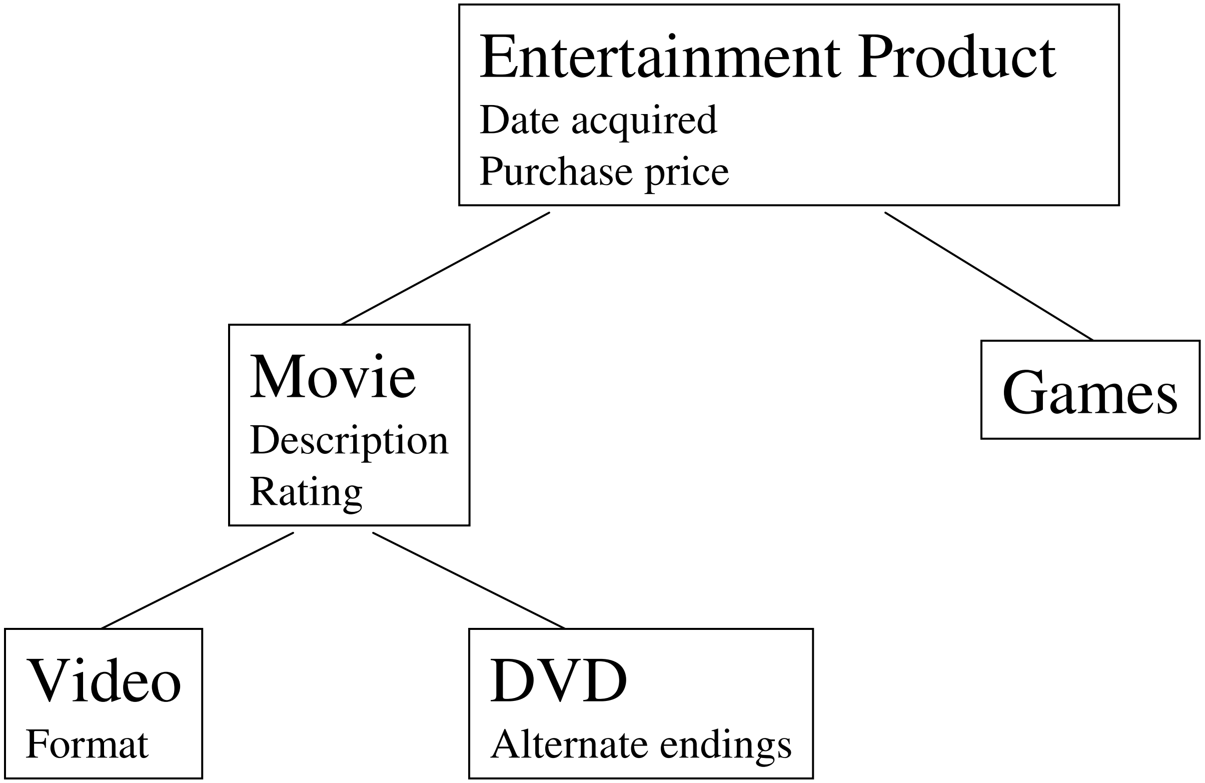
Object Properties – Inheritance



Generalization

Specialization

Inheritance – Video Store



Entertainment Product
Date acquired
Purchase price

Movie
Description
Rating

Games

Video
Format

DVD
Alternate endings

Inheritance Terms

- Specialization → Generalization
- Child → Parent
- Leaf → Root
- Class → Super-class

Inheritance - Example

- Generalization for “person” as previously shown.
- Specialization for “employee” type person.
  - Uses same Attributes and Operations but adds:
    - Hire date, salary, security clearance
- Allows us to add new specialized “person” type without re-inventing the entire wheel!

Object Properties - Abstraction

- A “super” generalization
  - Object > class > super-class > abstract class
  - Ted > employee > person > entity
- A class template
  - A class with no instantiated objects of its own
  - A class with no operations or attributes of its own
  - A class that declares what operations or attributes must be supported by sub-classes
    - Yet does not define how those operations are carried out or what the attributes are

Abstraction - Example

- Specialization of “employee” and “customer” as before
- Generalization of “person” as before
- Abstract class of “entity” which specifies that sub-classes will define “location”
  - Location is only a specification. There is no actual attribute or operation.
  - For “employee”, location is an internal office number only
  - For “customer”, location is a street address with city, state, zip

Abstraction - Example

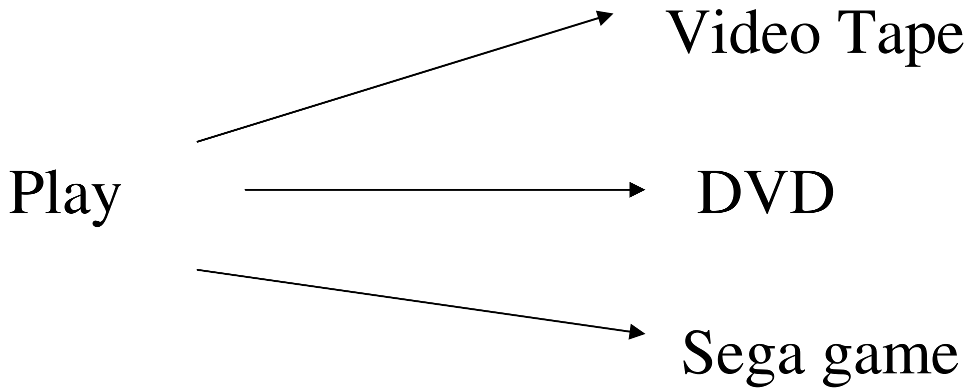
- Specialization of “employee” and “customer” as before
- Generalization of “person” as before
- Abstract class of “entity” which specifies that sub-classes will define “location”
  - Location is only a specification. There is no actual attribute or operation.
  - For “employee”, location is an internal office number only
  - For “customer”, location is a street address with city, state, zip
  - For “alien”, location is planet and galaxy name

Benefit of Abstraction

- Allows us to define an interface
  - for interacting with objects which are outside our system
- Allows us to define a flexible system
  - for extending our system in ways which we do not yet know about

Object Properties - Polymorphism

- The other side of the "abstract" property
  - Describes how an object experiences being a sub-set of an abstract class
- Receive same message - implement differently



Play

Video Tape

DVD

Sega game

Object Properties Review

- Encapsulation
- Association
- Inheritance
- Abstraction
- Polymorphism

Remember the OOAD Objective

- Identify the relevant objects in the problem domain that we are addressing
- Drill-down to the appropriate level of detail to discover relevant sub-objects
- Discover *patterns and relationships* so that efficient object groupings can be made providing effective system architectures
- *Dissect the domain, build the system*

That's all fine and dandy but……

- How do we use that information to translate our requirements into a system model?
- How do we physically represent that model?

The Unified Modeling Language

"A general purpose visual modeling language that is used to specify, construct, and document the artifacts of a software system."

-from The Unified Modeling Language Reference Manual by Rumbaugh, Jacobson, and Booch

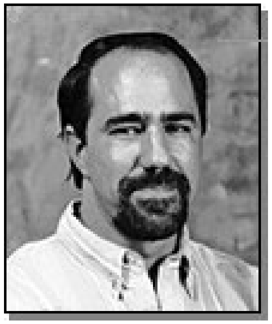
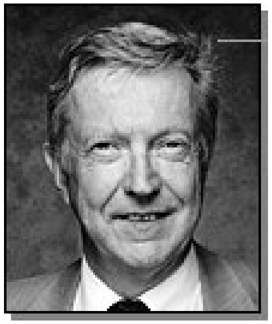
Visual Modeling

- Provides a method and standard notation for modeling
- Graphically oriented rather than text oriented
- Focus on conceptualization and abstraction
- Model evolves during project lifecycle

Visualize behavior rather than low-level constructs

Background

- Mostly Booch, Jacobson, and Rumbaugh
- UML evolved from their earlier works
- Now controlled by the Object Management Group (OMG)
- Variations and extensions exist

UML Version 2.0

- Approved by OMG in 2003
- Released May 2004
- Provides additional notation and models
- Enhances UML for use in code generation
  - supports Model Driven Architecture
- Most changes are "behind the scenes" to casual users

UML as a tool

- Whiteboard artifact
- Blueprints for architects
- Detailed design for code generation

- Use UML as it makes sense for the purpose at hand!

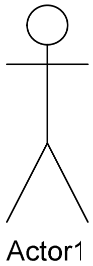
Views of the World

- Use Case Model
- Static Models
- Interaction Models

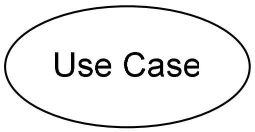
Use Case Diagram

- Initial system model
- Provides a graphical representation of services the system will provide
- Helps to establish project boundaries
- Used during the inception phase of the project

Use Case Diagram - components



Actor: Person, system, clock

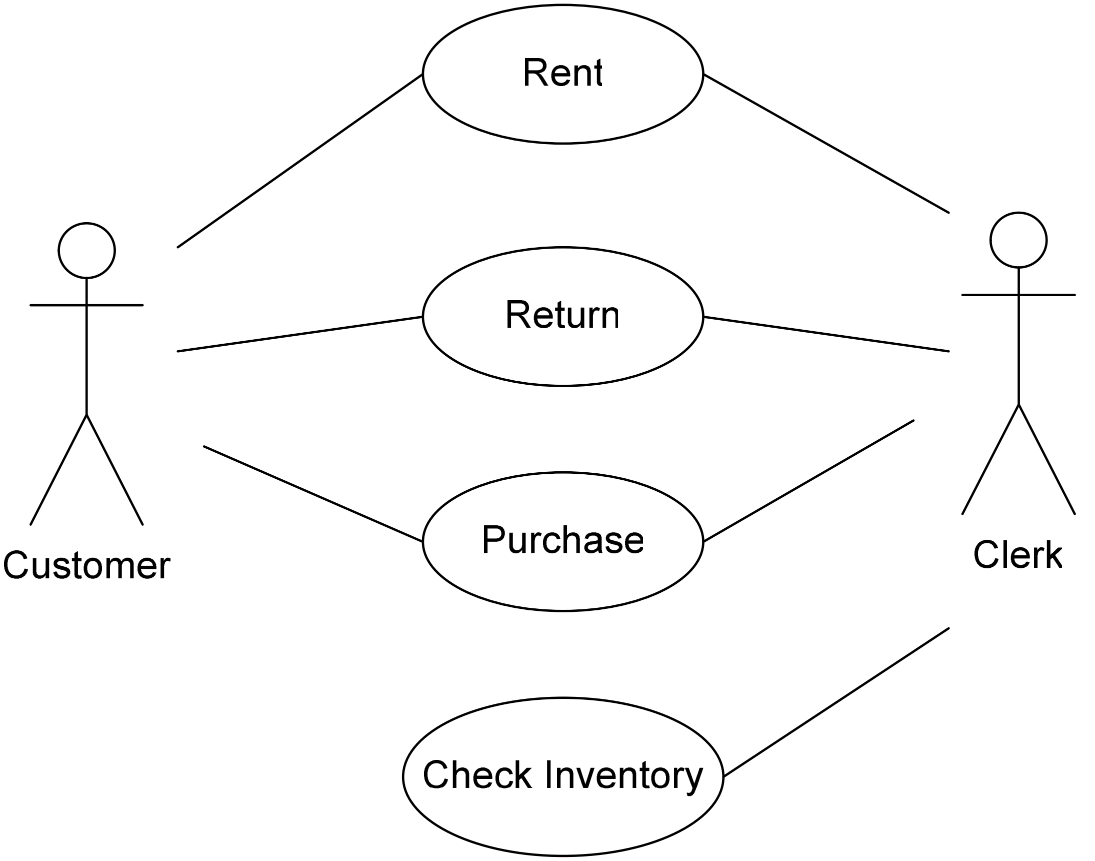


Use Case: A function of value for the Actor

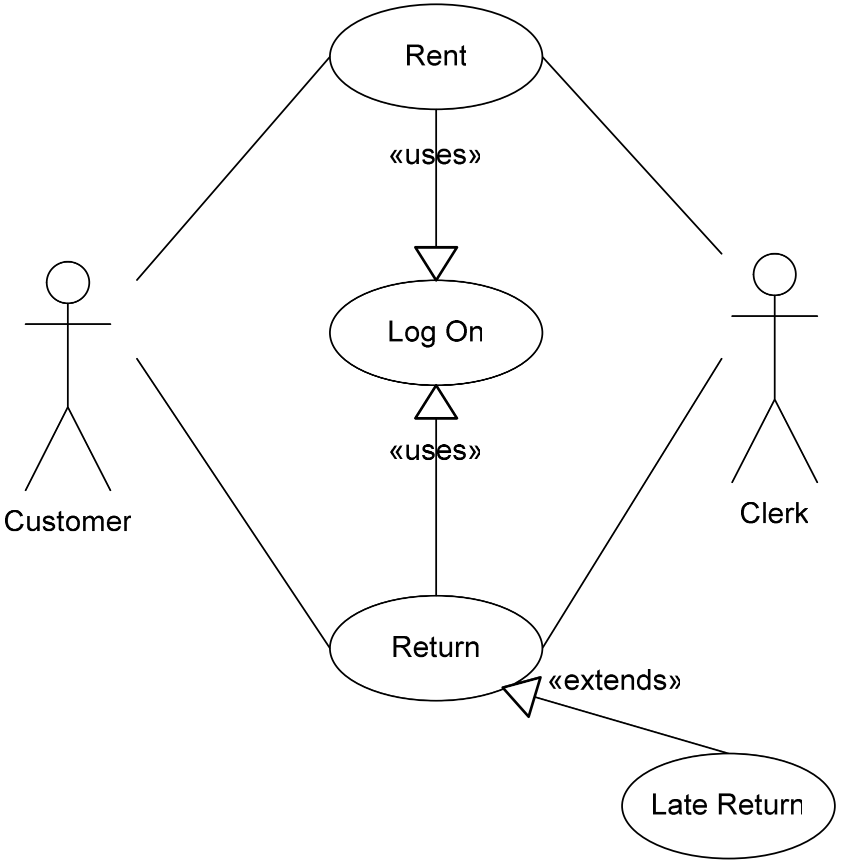
Communication: Link between Actor and Use Case

Use Case Diagram Example

ACME VIDEO RENTALS



Use Case Diagram Example



Use Cases

- “Flesh Out” the Use Cases identified in the Use Case Diagram
- Introduced in the elaboration/discovery phase of the project
- Represent the function as experienced by the “actor”
- Use Cases are text based
  - Have defined content
  - May have a defined context (templates)

Static Models

- Represent view of the system as a snapshot-in-time
- Show the structure of the system

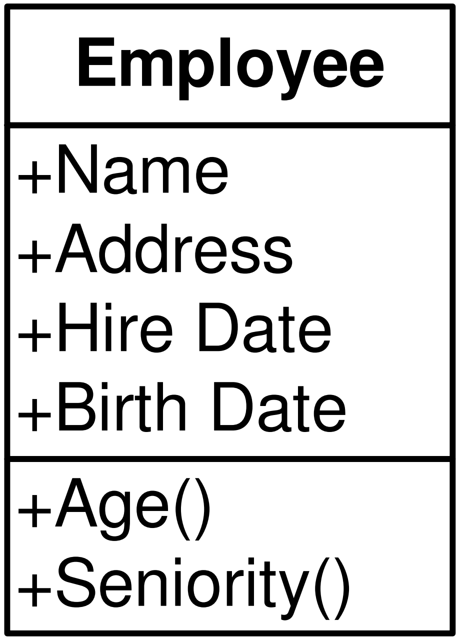
Static Models

- Represent view of the system as a snapshot-in-time
- Show the structure of the system

- Class
- Object
- Package
- Component
- Deployment

Class

- An individual class has:
  - Name
  - Attributes
  - Methods
- There are also advanced features:
  - Tags (meta-data)
  - Visibility notations
    - + public, # protected, - private



Employee

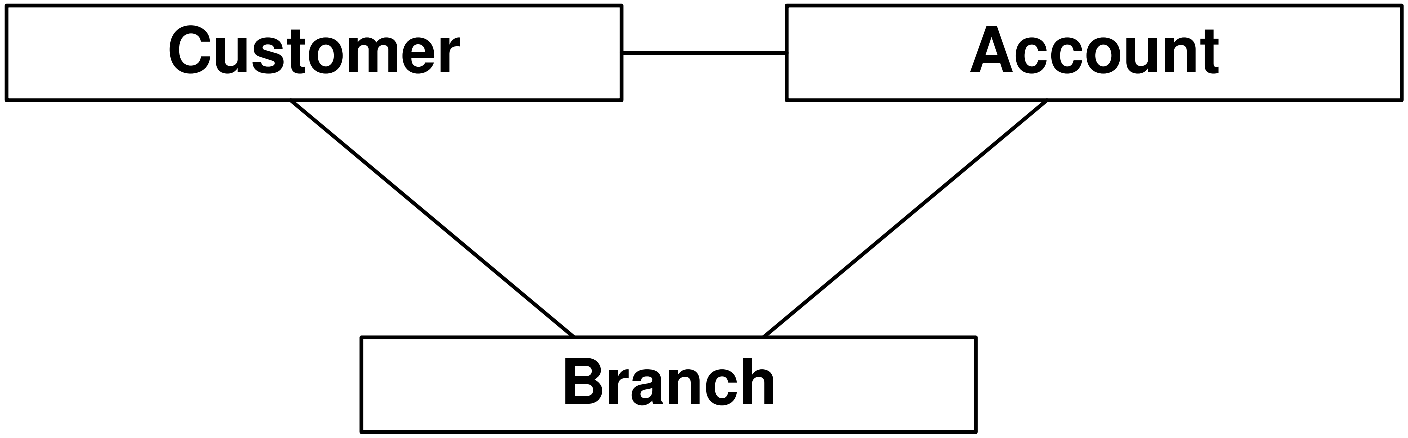
+Name
+Address
+Hire Date
+Birth Date

+Age()
+Seniority()

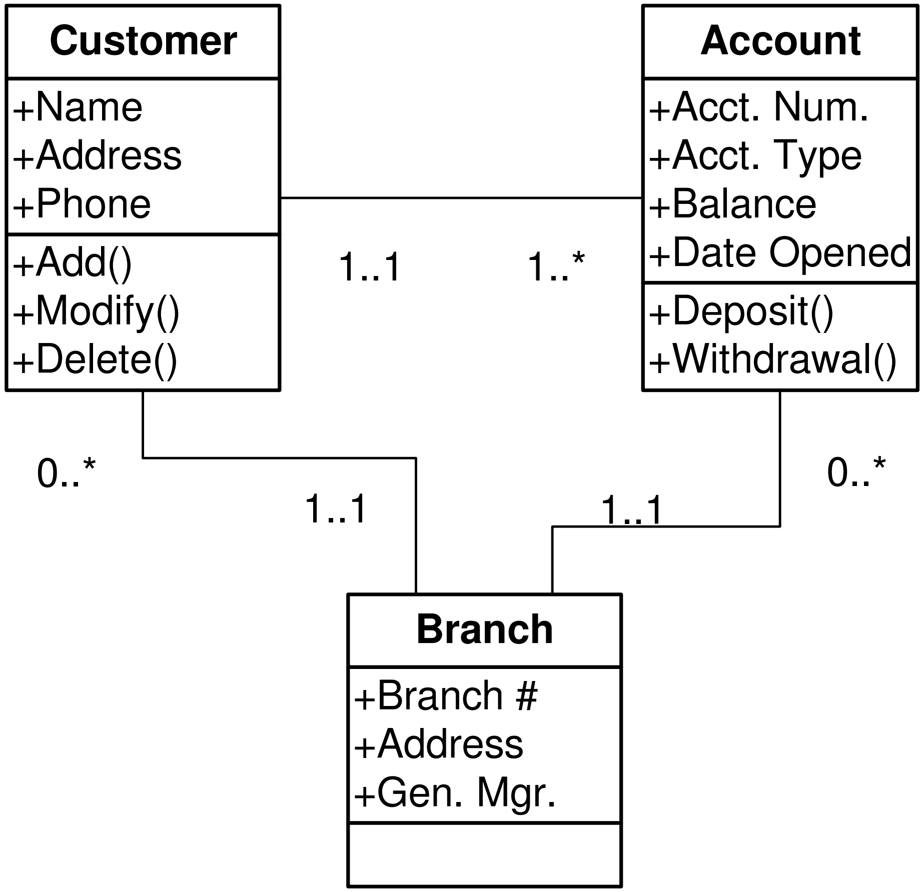
Class Diagram

- Shows relationship between classes
- The most common object model
- Can be shown at various levels of abstraction
- Introduced in the elaboration/discovery phase
  - After Use Cases
  - Continued use through design and construction phases

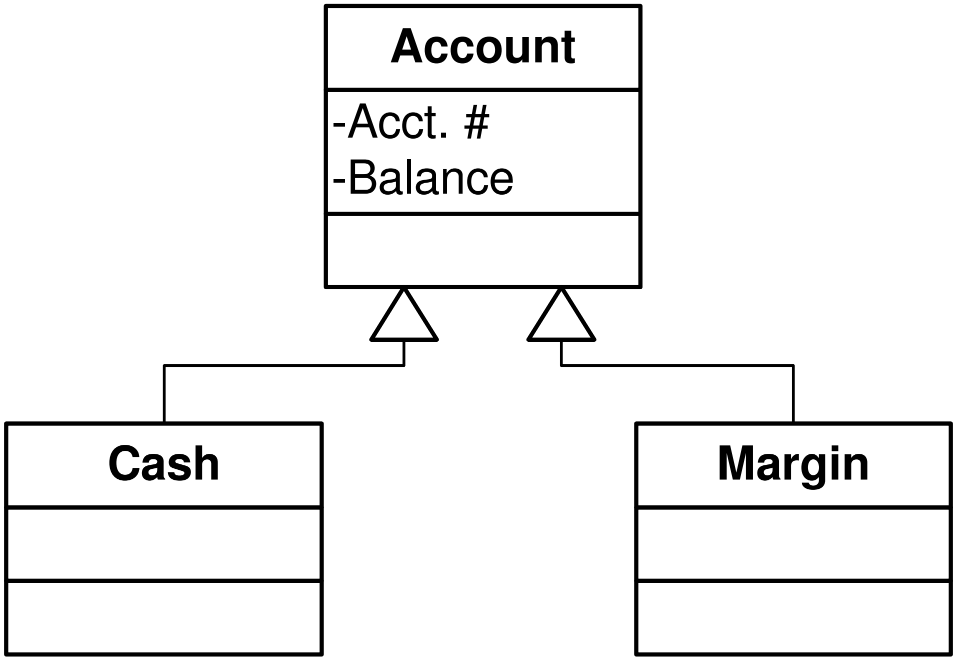
Class Diagram – Basic Level



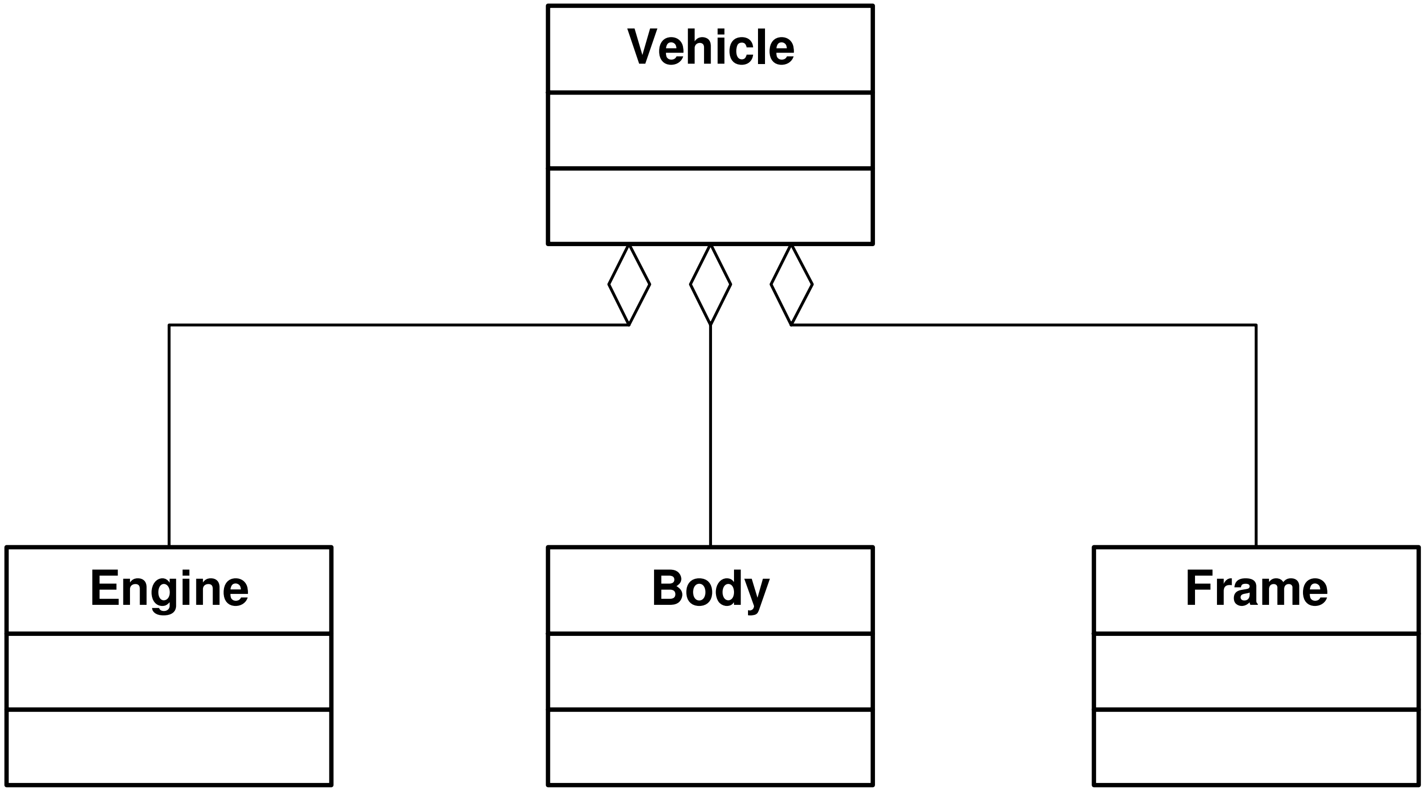
Class Diagram with Detail



Class Diagram w/ Generalization



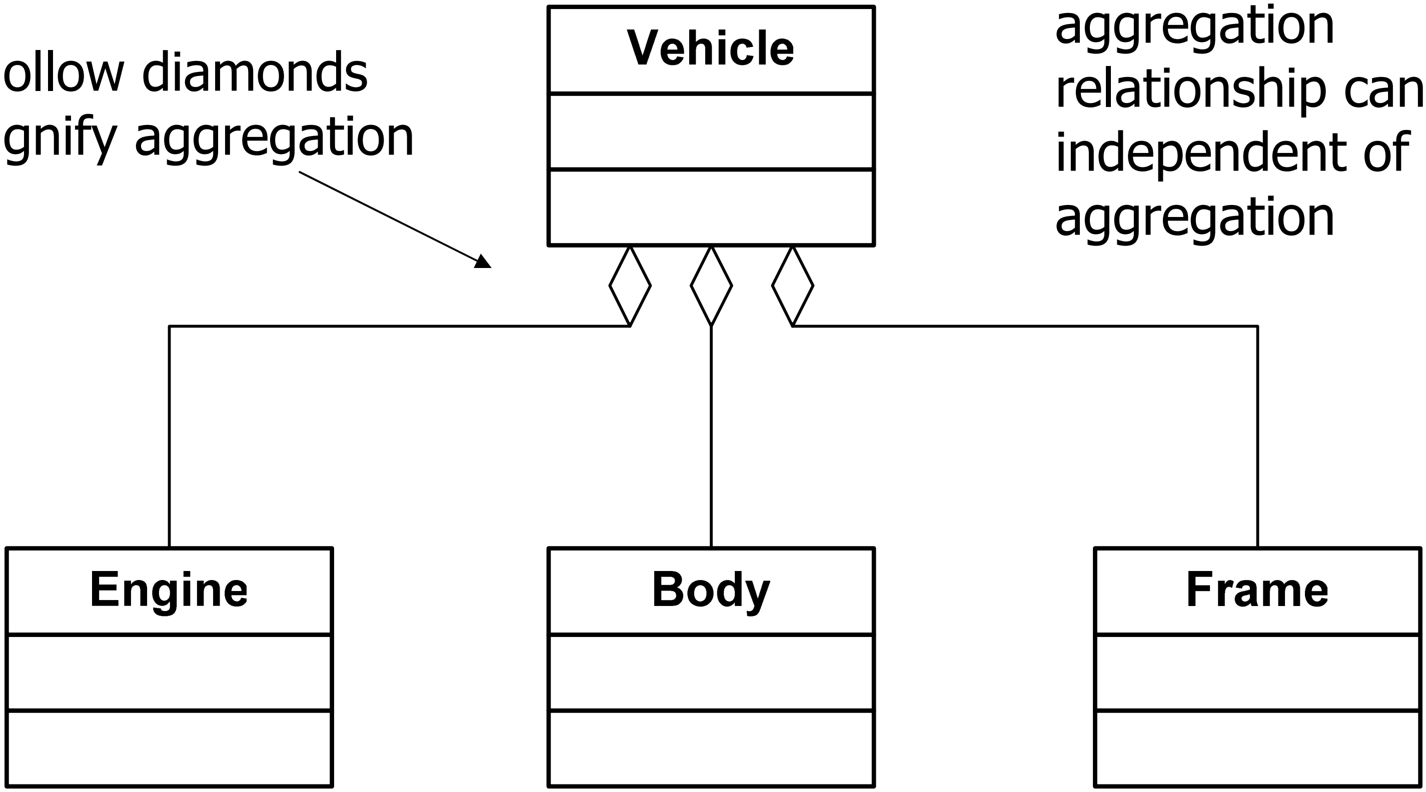
Class Diagram w/ Aggregation



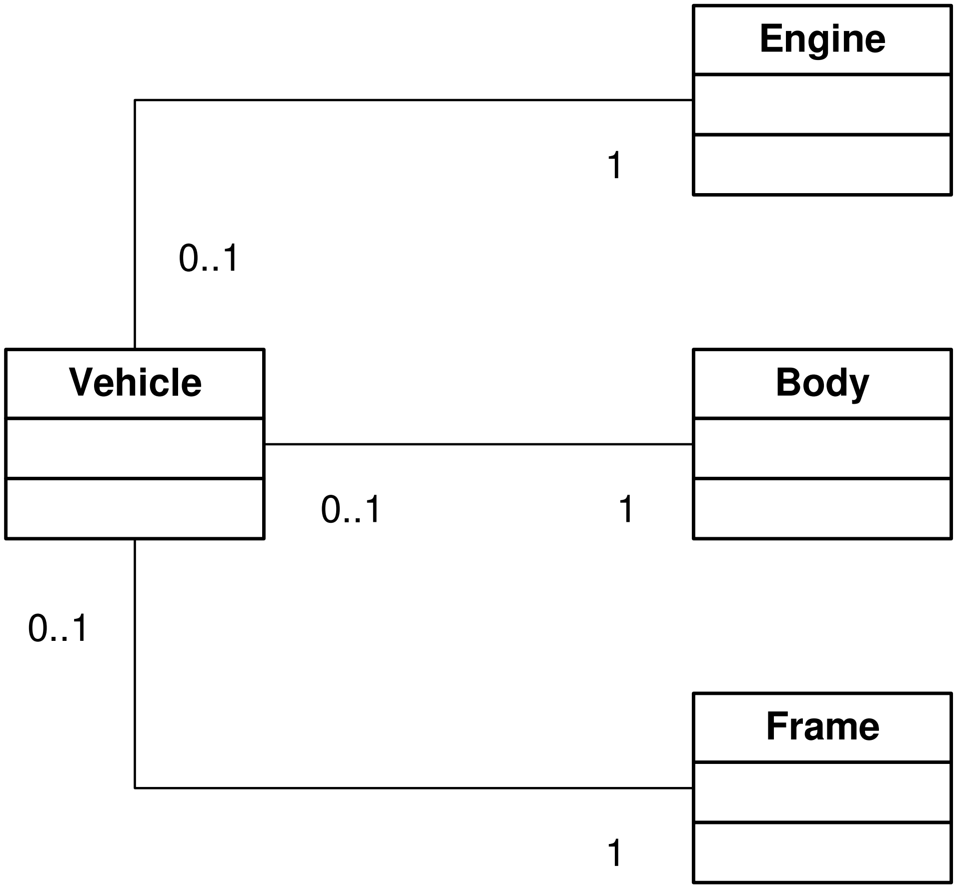
Class Diagram w/ Aggregation

Components of the aggregation relationship can exist independent of the aggregation

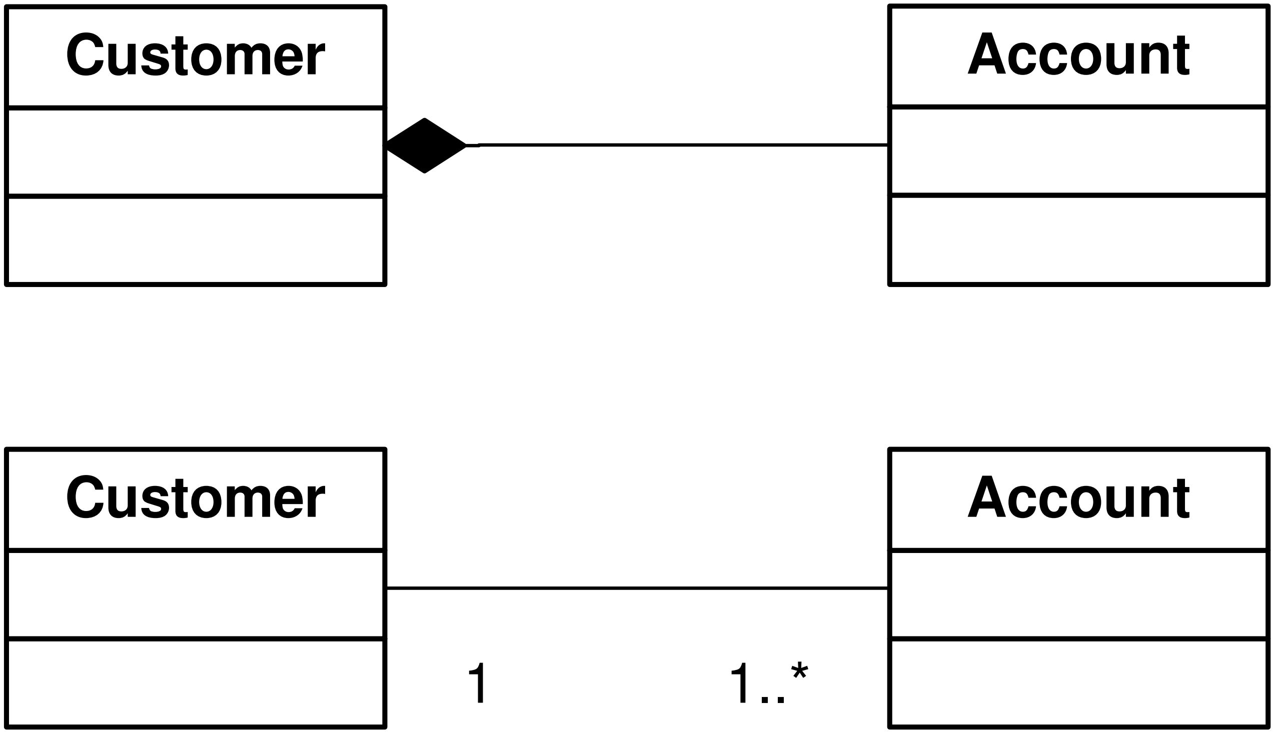
Hollow diamonds signify aggregation



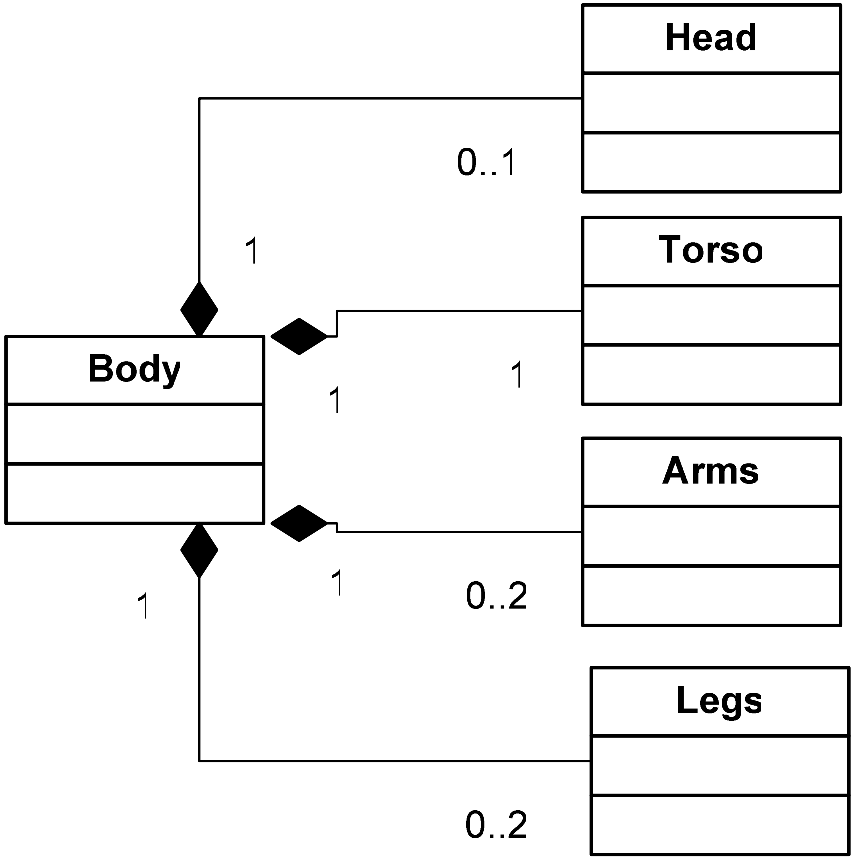
Aggregation – another way



Class Diagram w/ Composition

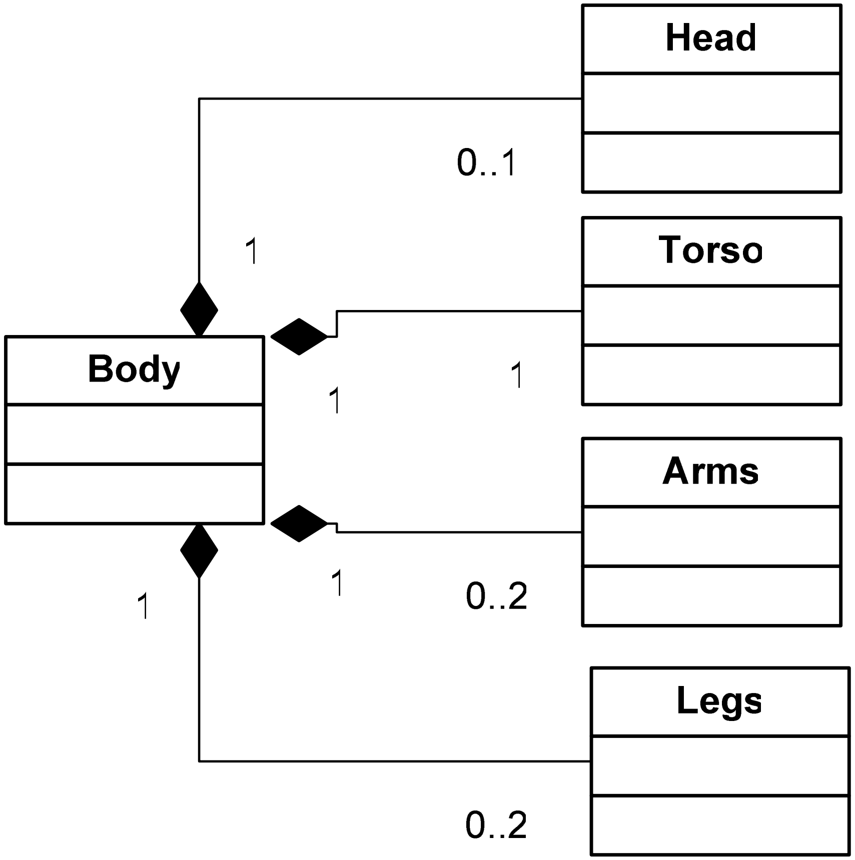


Composition - Example



Composition - Example

Solid diamonds indicates composition



Components of the composition relationship can not exist independent of the composition

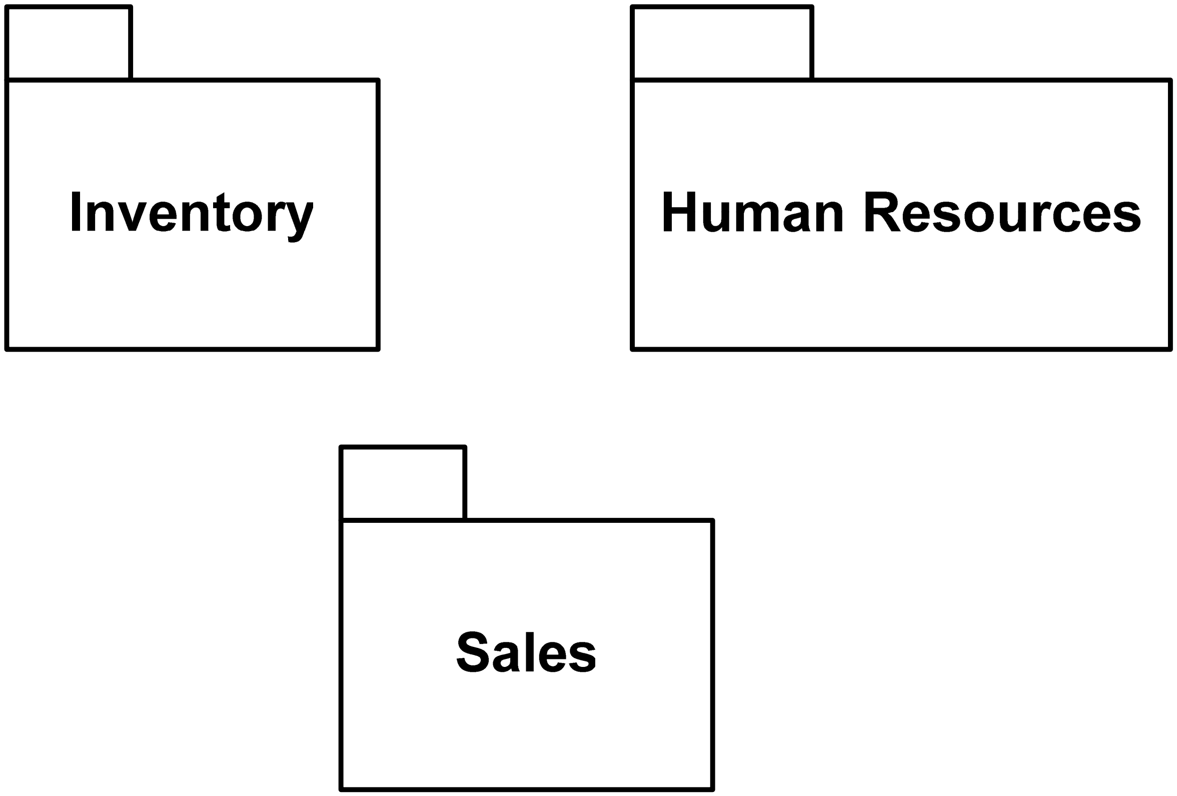
Object Diagram

- Looks like a class diagram *except*:
  - Demonstrates instantiated classes
  - Shows relationship between specific objects instead of classes
  - Used to give example of how a system will look under specific circumstances
  - Noted by object:class notation
    - Fred:student
    - 536390247:SSN

Package

- A flexible model used to combine elements to:
  - Represent a modular view of the system
  - Allow for general abstraction
- Can be used to combine:
  - Classes
  - Components
  - Nodes
  - Or any other UML construct

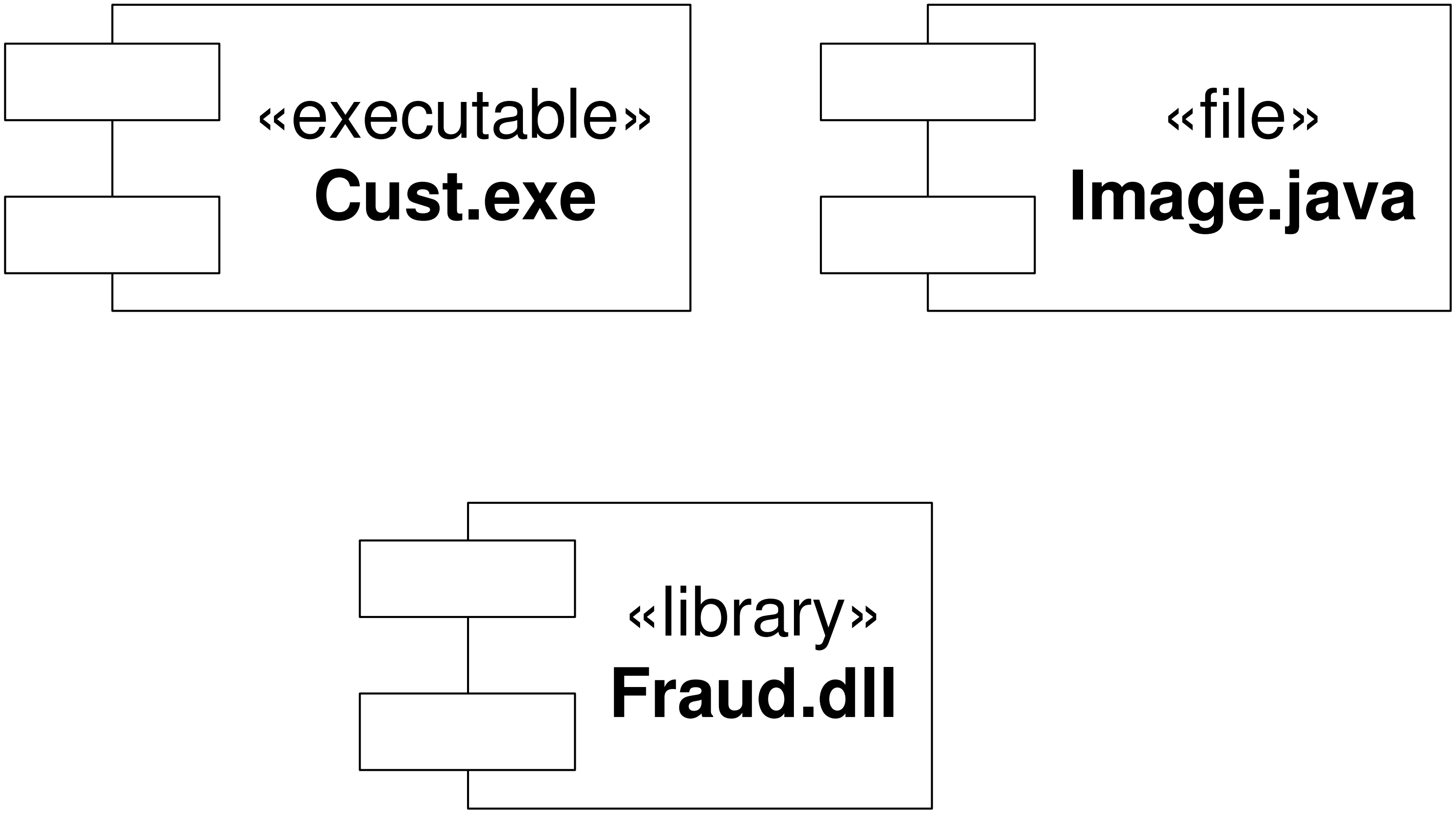
Package Diagram



Components

- Physical manifestation of software
- Contain code, database files, etc.
- Usually contain multiple classes
- Low coupling between components
- Often “pluggable” – replaceable by other components

Component Diagram – Prior UML versions

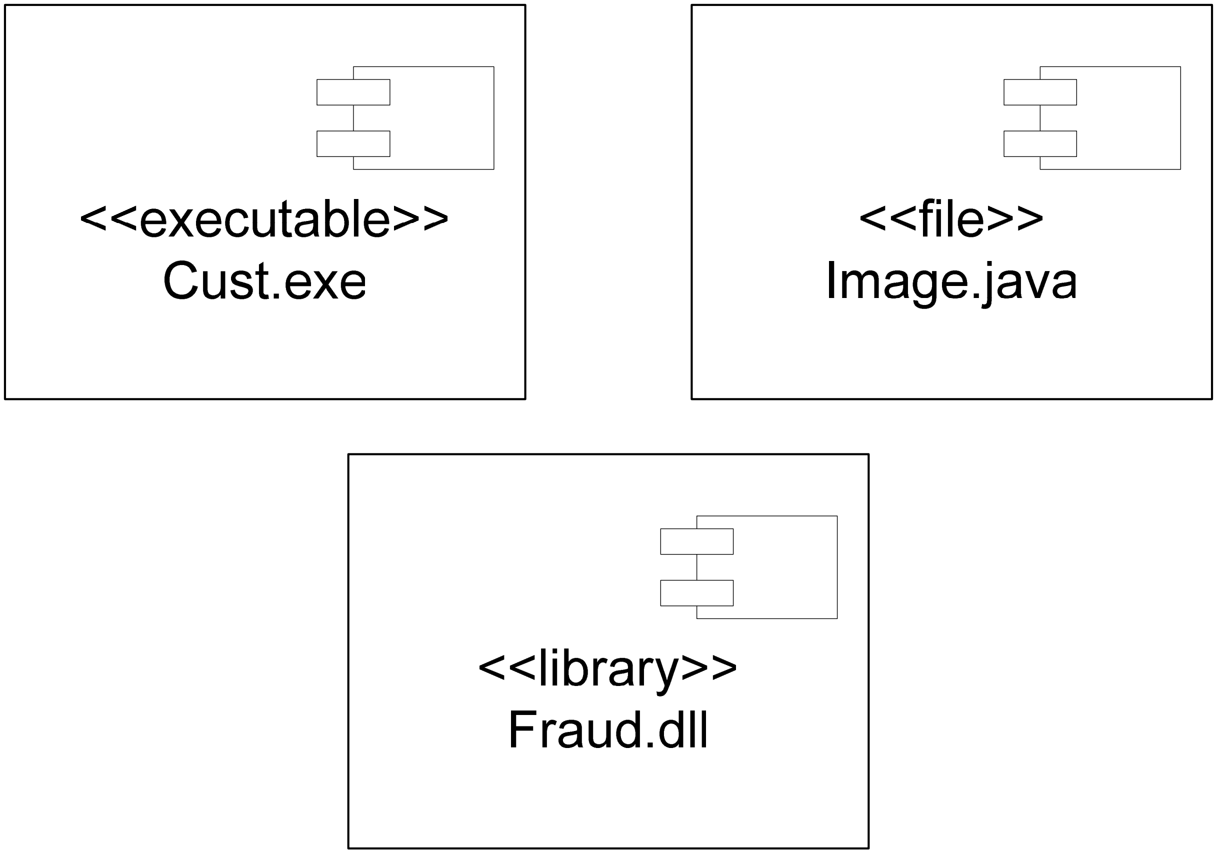


«executable»
Cust.exe

«file»
Image.java

«library»
Fraud.dll

Component Diagram – UML version 2

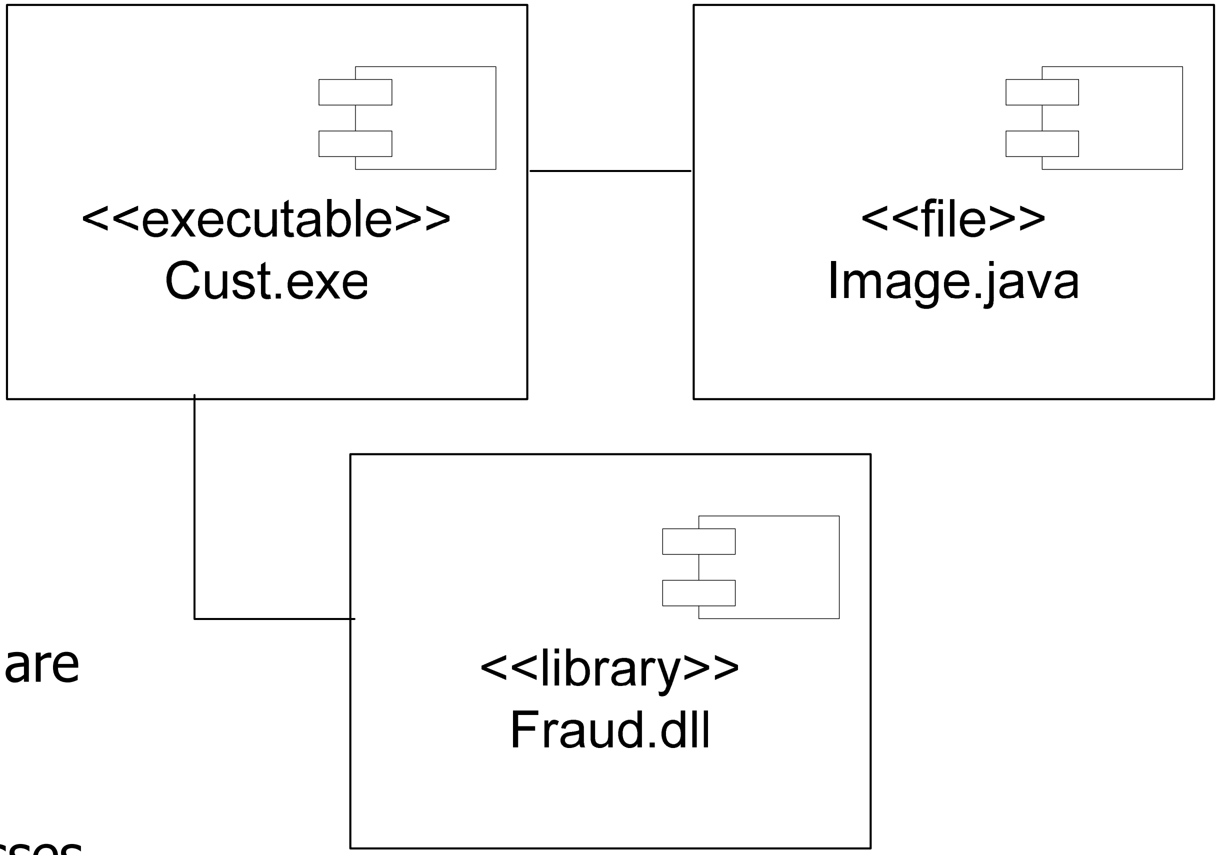


<<executable>>
Cust.exe

<<file>>
Image.java

<<library>>
Fraud.dll

Component Diagram – UML version 2

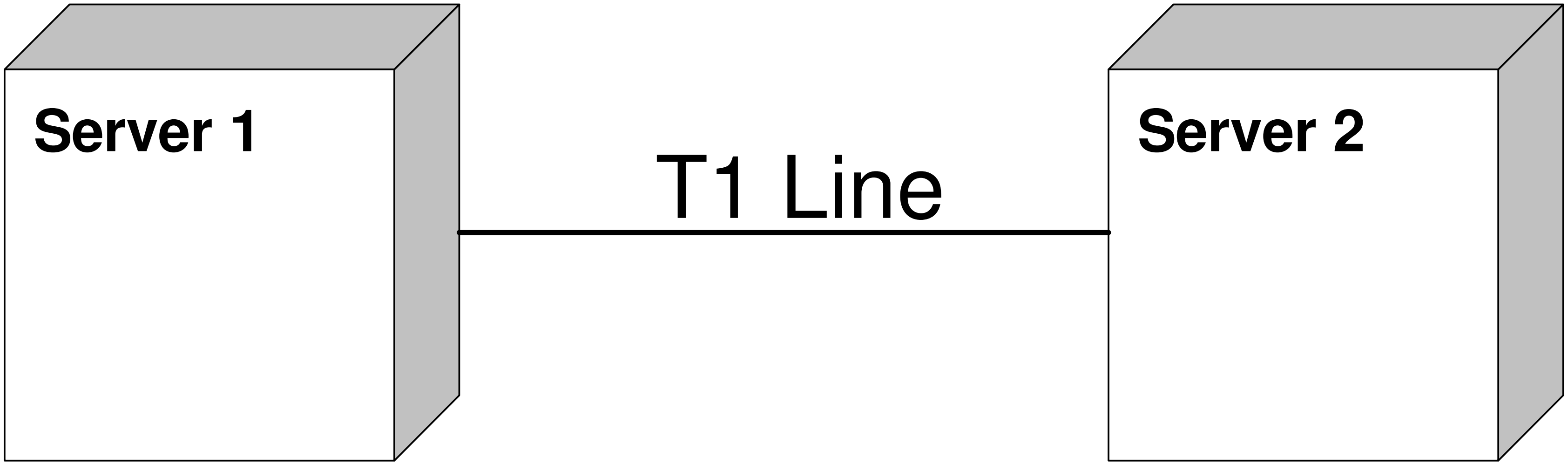


Associations between components are drawn like associations between classes

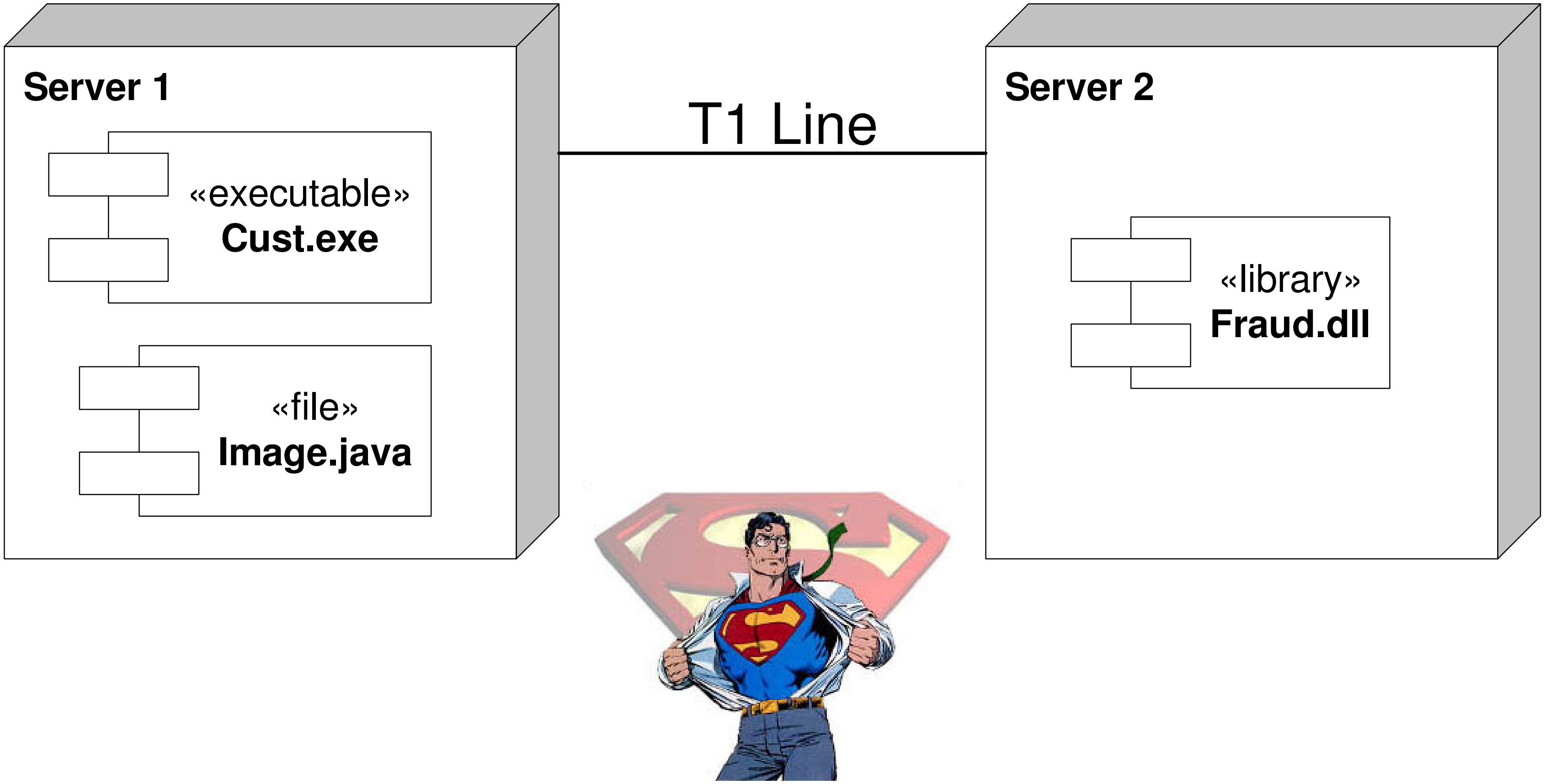
Deployment

- A model representing physical system components including:
  - Workstations
  - Servers
  - Embedded devices
  - Etc.
- A node on the deployment diagram usually has <u>processing capability</u> and <u>memory</u>

Deployment Diagram



Deployment + Component



Interaction Models

- Represent view of the system as it is executing
- Show the interaction of the system
- Show changes over time

Interaction Models

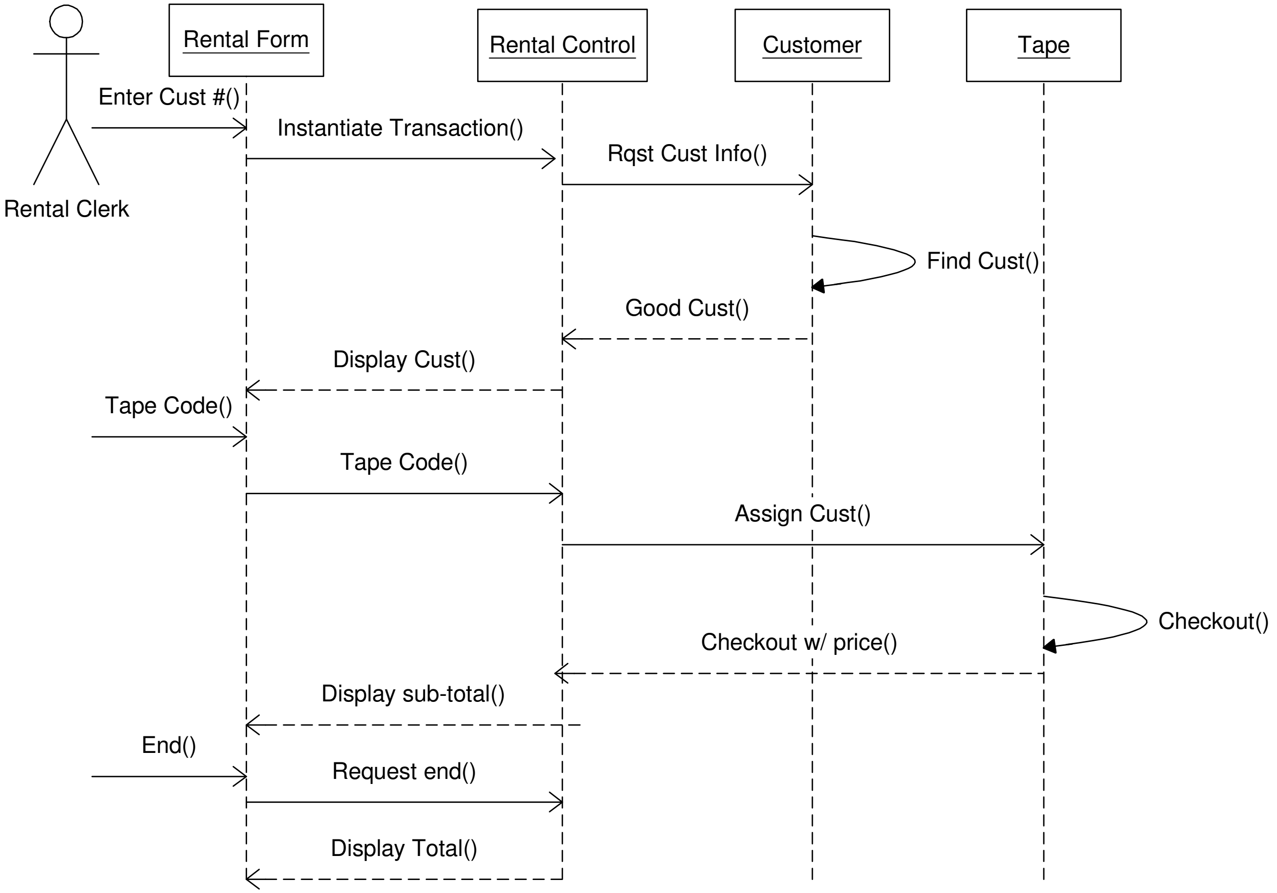
- Represent view of the system as it is executing
- Show the interaction of the system
- Show changes over time

- Sequence
- Communication
  - Collaboration UML 1.x
- Activity
- State Machine

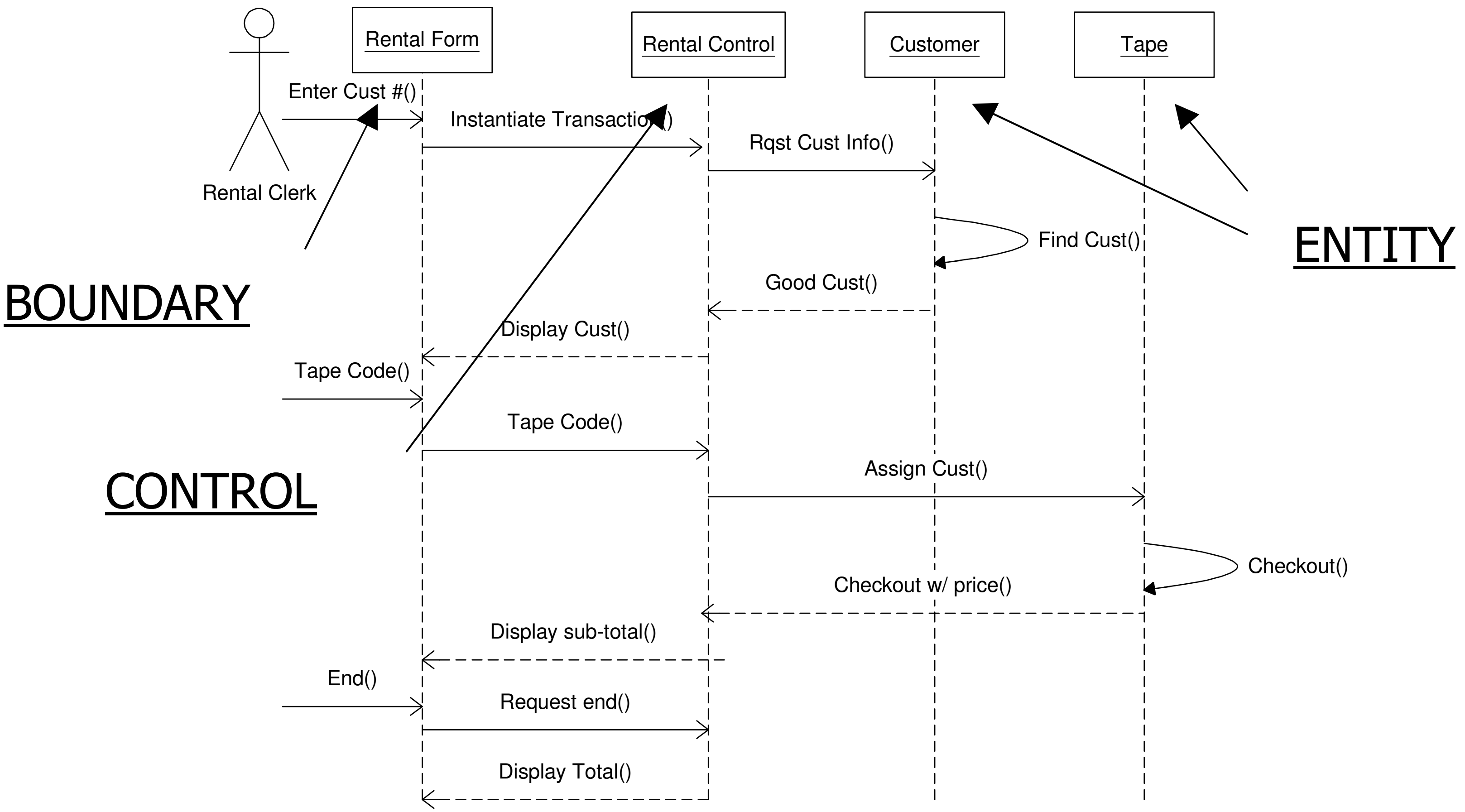
Sequence Diagram

- Represents a sequence of events
  - Usually tied to a single path thru a Use Case
  - Each possible execution path thru a Use Case should have its own Sequence Diagram
- Shows messages passing between objects over time.
  - Message and time oriented (vs. class or object relationship)
  - Shows the “lifecycle” of a single use case scenario

Sequence Diagram



Sequence Diagram



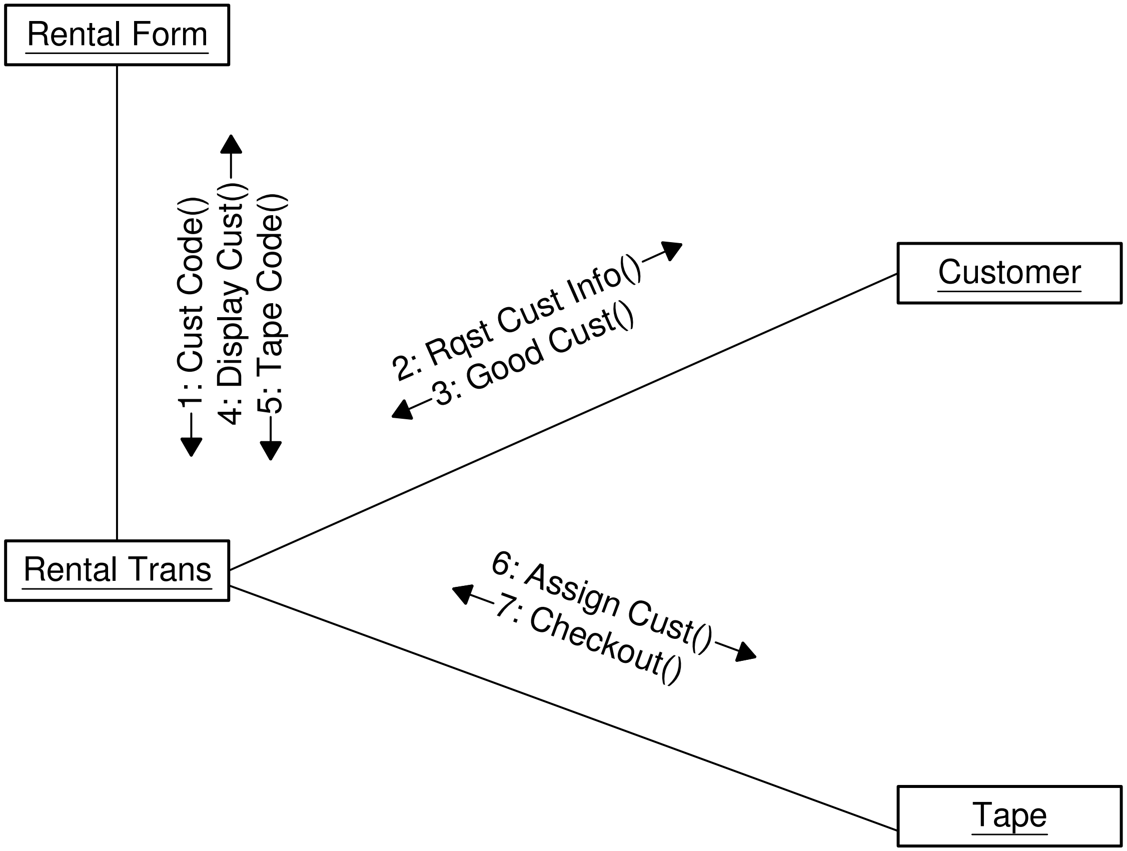
Sequence Diagram

- UML Ver. 2 introduced notation to show branching such as loops and if-then-else logic
- Uses a “frame” – a box around the steps which are repeated with a notation of the type of branch
- Can make the diagram difficult to read and understand

Communication (formerly Collaboration)

- Represents relationships between classes
  - Focused on classes or objects and their relationships in executing various scenarios
  - Points out potential bottlenecks and over-dependencies.
- Can be derived from Sequence Diagram
  - Many modeling packages will allow generation of Communication Diagrams from Sequence Diagram and vice versa.

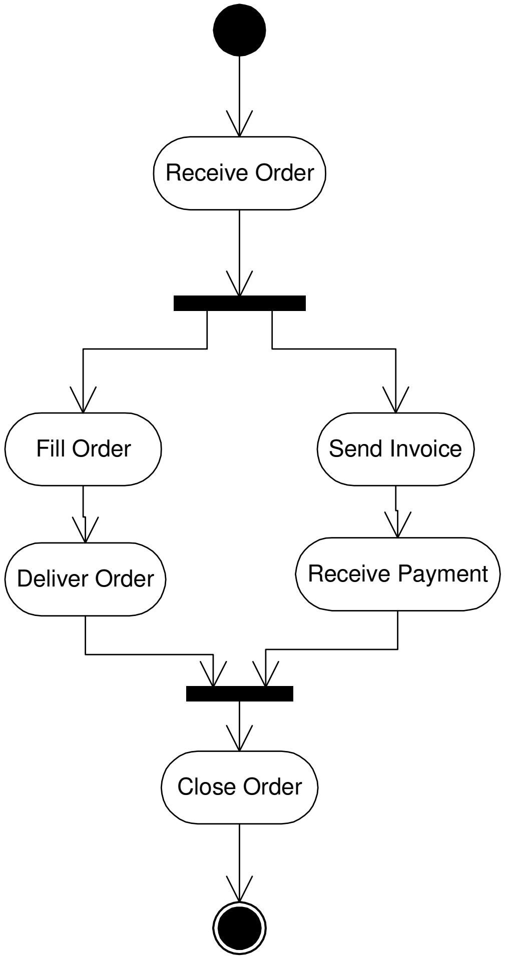
Communication Diagram



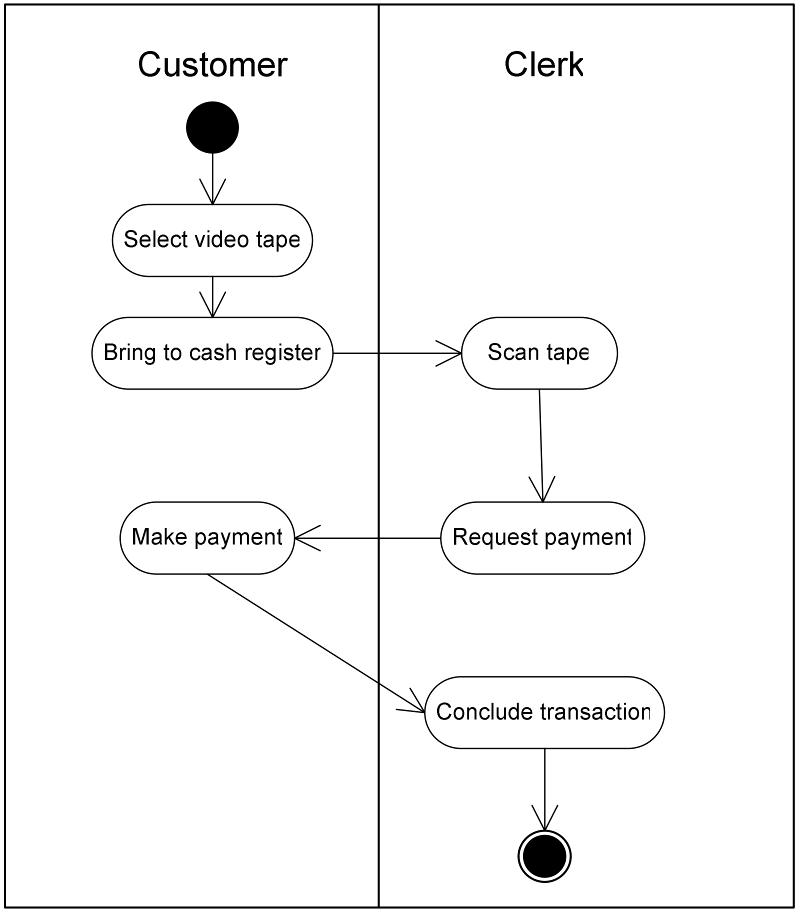
Activity Diagram

- Represents task activity
  - Includes parallel activity
  - Focus on action or changes to system state
    - (vs. class or object state changes)
- A flowchart with object notation
- UML 2 – nodes are referred to as "actions" instead of "activities"
- Used in multiple phases of a project
- Most frequently used for business process modeling

Activity Diagram

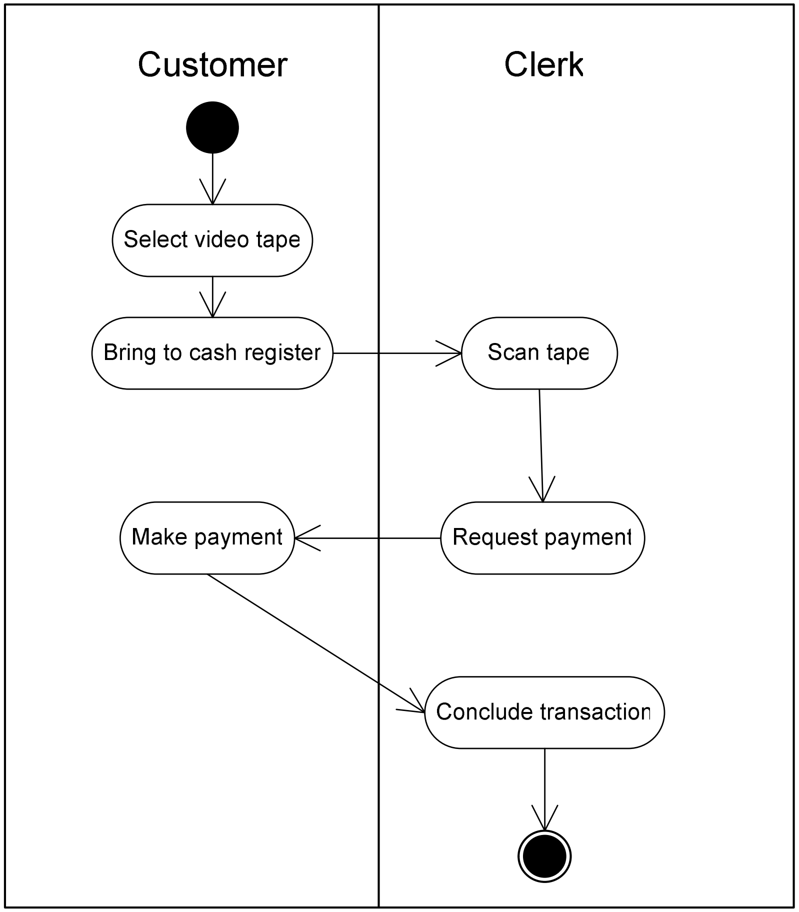


Activity Diagram w/ swim lanes



Activity Diagram w/ swim lanes

Activity diagram notation is becoming increasingly complex. Many new elements added in UML ver. 2



Notations for:

Time signals

Alternate terminations

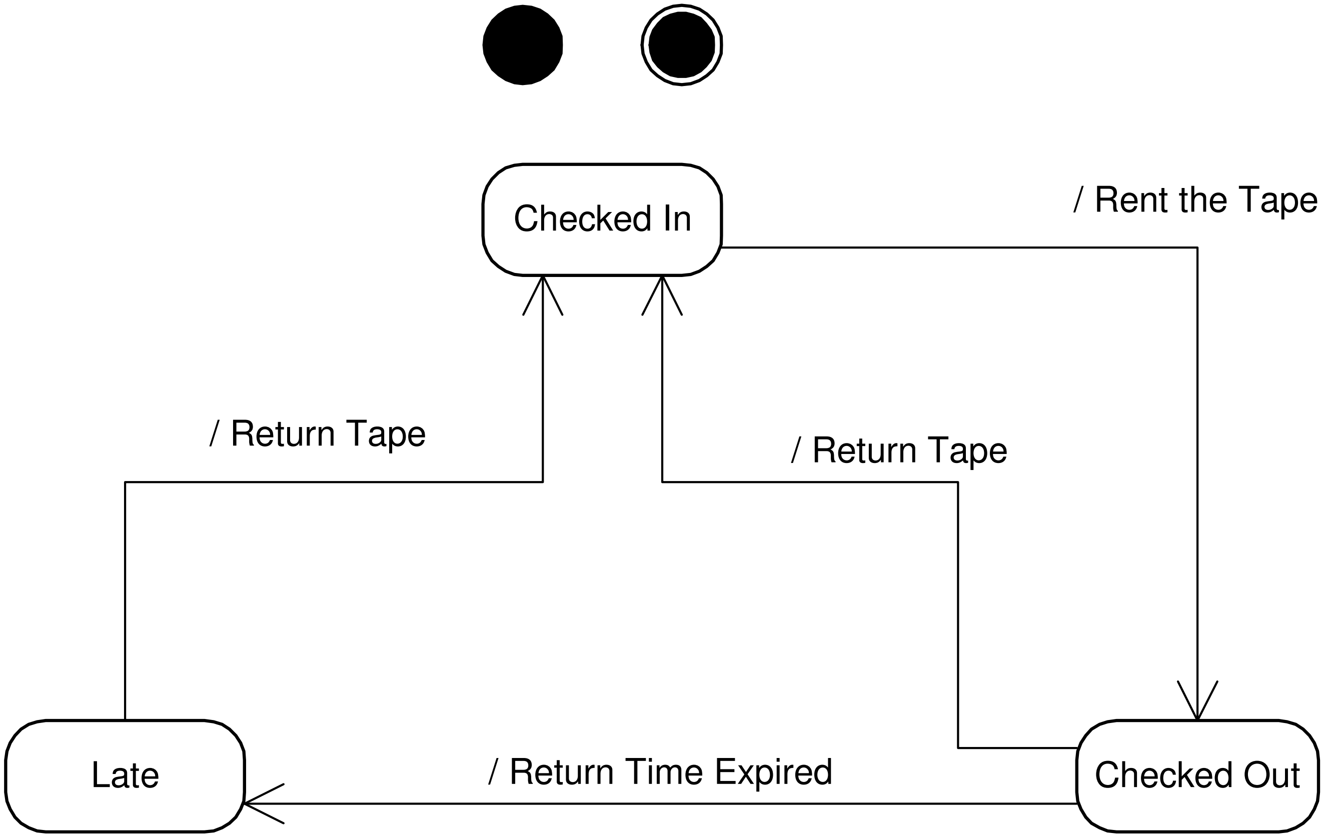
Pre- and Post-condition notes

Exception flows

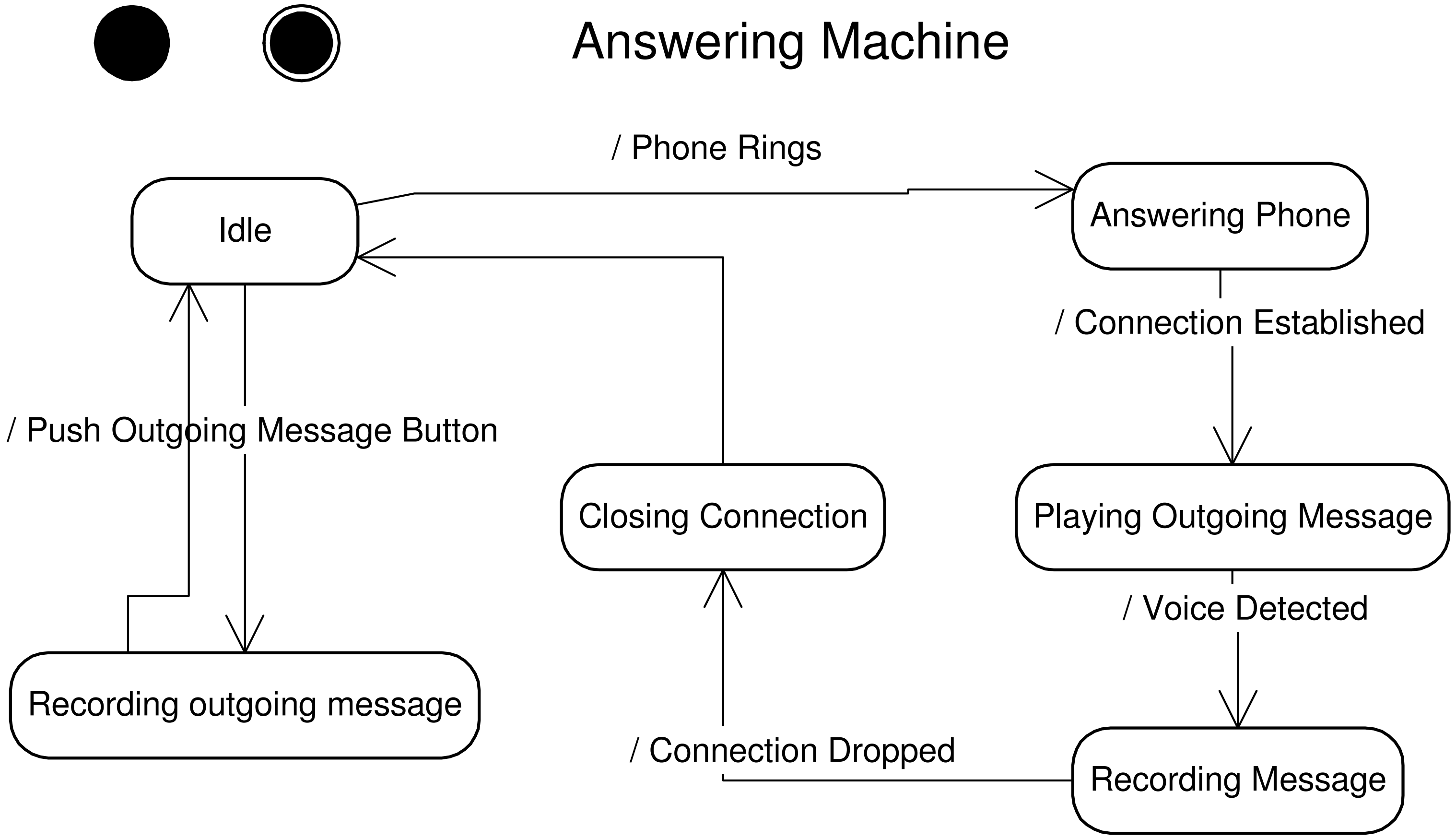
State Machine

- *State*: A condition in the life of an object during which it satisfies some condition, performs some activity, or waits for some event
- The state machine shows how activities change the state of an object
- Also referred to as State Transition
- Tends to be used during design phase

State Machine - Video Tape



State Machine – Answering Machine



Additional Models

- UML ver. 2 has introduced several new models
  - Interaction Overview Diagram
  - Composite Structure Diagram
  - Timing Diagram
- Still too soon to know if they will be generally adopted in the "real world"

Where are we?

- We understand the relationship between the software engineering process and OOAD
  - What and When vs. How
- We know what an object is
  - Behavior & responsibility
  - Classes define related objects
- We know how to describe object relationships
  - Encapsulation, associations, inheritance
- We know how to represent those relationships
  - Unified Modeling Language

We can “talk the talk”…

Now let’s try to “walk the walk”…….

Applying the Technology

- Integrate Object Oriented Analysis and Design techniques with a Software Engineering Process
  - Define the Project
  - Analyze the requirements
  - Model the architecture
  - Prepare the work packages

Define the Project

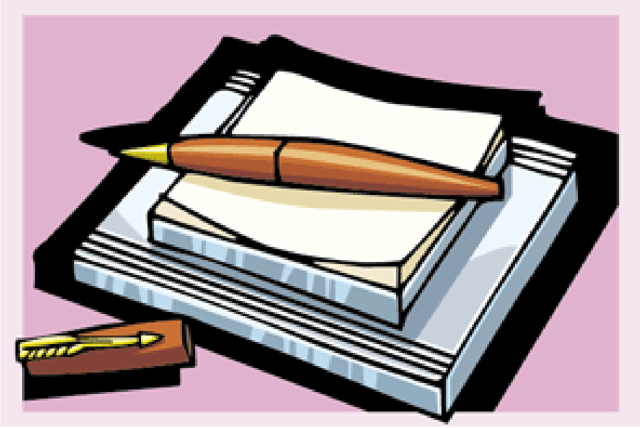
- Scope the System Domain
  - What are the key services or functions
    - Use Cases
  - What are the roles of the system users?
    - Actors

Exercise #1

- Prepare a Use Case diagram for an Automated Teller Machine
    - Identify Actors
    - Define Use Cases
    - Note Relationships

Gather Requirements

- Capture functional requirements
  - By Use Case
  - General for overall system
- Define non-functional requirements
  - Performance
  - Scalability
  - Usability
  - Etc.



Develop Use Cases

- A project team activity
- Based upon requirements
- Reflect actor’s experience
- Capture event sequence

Exercise #2

- Prepare one Use Case for ATM project

Analyze the Requirements

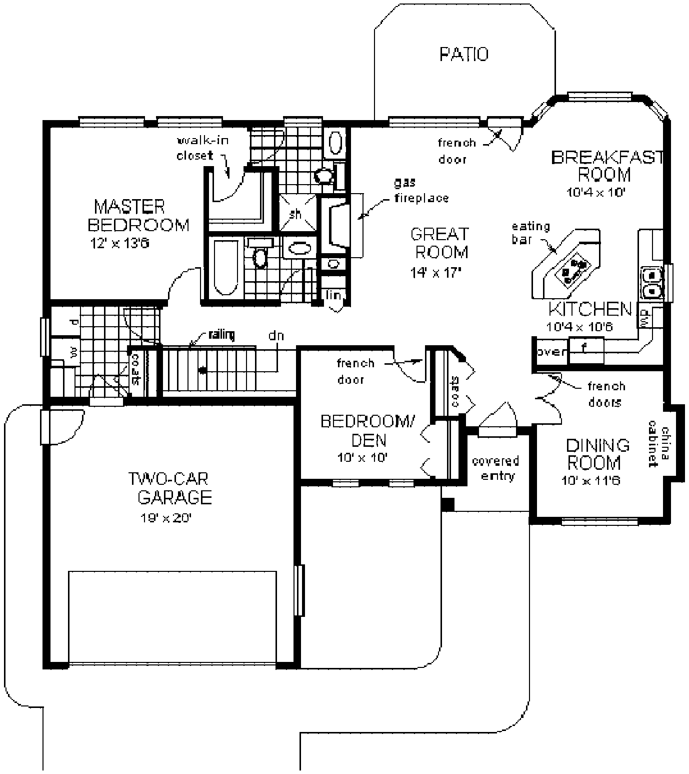
- Discover objects
    - Class Stereotypes
        - Boundary
        - Control
        - Entity
- Model object collaborations
    - Sequence Diagram

Exercise #3

- Conduct a Use Case drilldown for the ATM project
  - Prepare a sequence diagram

Define the architecture

- Model classes
- Architecture considerations
- Prepare work packages



Model Classes

- Behaviors & responsibilities
- Relationships

- A tool to help:
  Class-Responsibility-Collaboration cards

Class-Responsibility-Collaboration

- Known as CRC cards
- Introduced by Kent Beck and Ward Cunningham
    - creators of eXtreme Programming
- Used for class definition
- Not part of formal UML notation
- May be conducted during or after Sequence Diagram exercise

Exercise #4

- Prepare a class diagram for the ATM Use Case

Additional Modeling

- Depending on system complexity, determine whether there is a need for other models
- Use Package Diagrams to “summarize” complex systems
- Use Component + Deployment Diagrams to direct installation

Other Architectural Considerations

- Identify the constraints
  - Legacy systems, supported platforms, standards, distribution requirements, staff skills, budget, time, etc.
- System needs
  - Most frequently used Use Cases, scenarios, and objects
  - High risk design issues
  - Non-functional requirements

Prepare Work Packages

- Assign feature sets (use cases or groupings of use cases) to development teams
- Assign individual ownership responsibility for classes
  - Limit the number of developers who work on a specific class
  - Clearly document class interfaces

Managing OO Projects

- Iterative development
- Limited class complexity
- Frequent “Build & Test”
- Clearly define class interfaces

Iterative Development

- Begin with architecturally significant or high risk Use Cases
- Identify design patterns & re-use candidates
- Re-iterate by adding Use Cases
- Partition the application domain
  - Manage complexity with packages

Limit Class Complexity

- Limit behavior
  - High cohesion
- Reduce Cyclomatic Complexity (<10-15)
  - Improves maintainability and testability

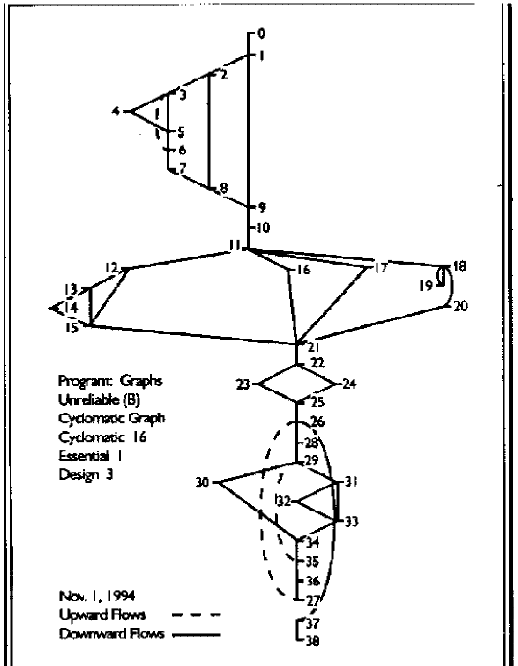
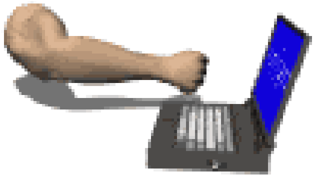


FIGURE 2: COMPLEX, MODERATE-RISK SOFTWARE MODULE.

Frequent Build & Test

- Limit development changes between Build & Test cycles
- Execute functional tests against use cases
- Execute performance and stress tests against packages



Clearly Define Class Interfaces

- Take the time to clearly design and define interfaces
  - Especially if work is spread across multiple development teams
  - Critical for web-services and Service-oriented architecture (SOA)
- Provide wrappers for legacy applications

Summary

- Object Oriented Analysis and Development provides a way to define and model a system
  - A development methodology combines software engineering processes and OOAD modeling
- But….
  - there is a steep learning curve. You must be prepared to exercise this method several times before you begin to become proficient!