

CS10102302

UML面向对象建模

赵君峤 长聘副教授

计算机科学与技术学院

同济大学

How to Think in Terms of Objects?

- The desired end result of OO design is a **robust** and **functional object model** — a system.
- Don't try to conform to any standards or conventions when solving a problem.
- The whole idea is to be creative.

ACTIVE AGENT WALKS TO WATER



BY MISHRAKA

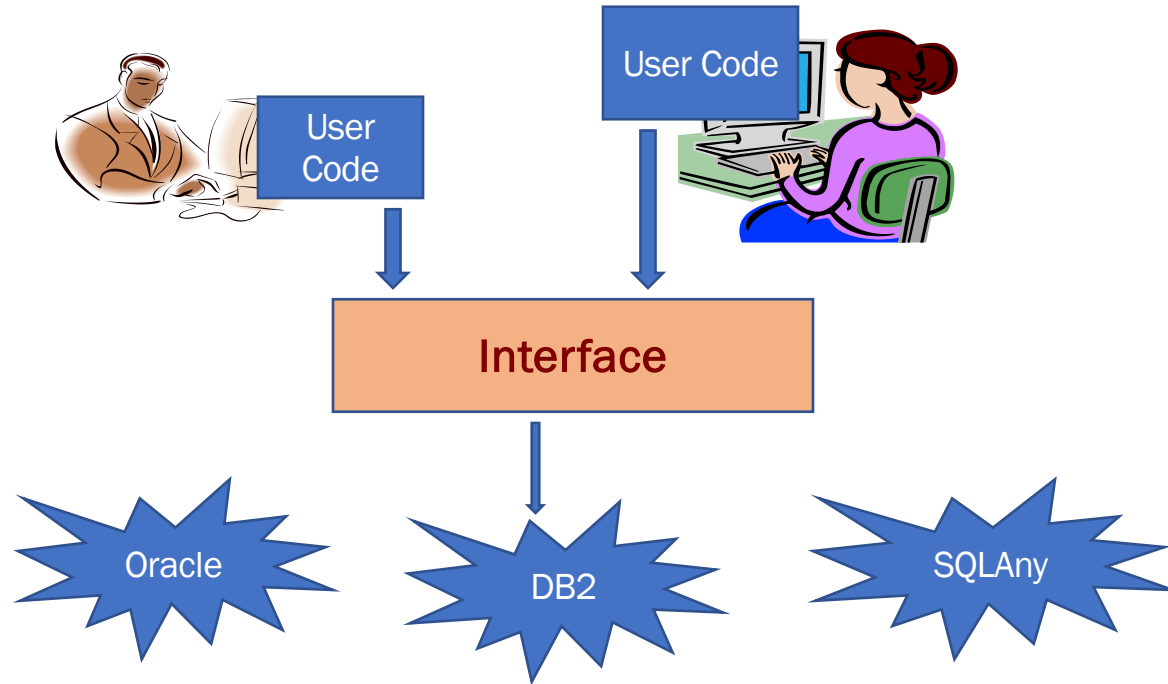


Three important things

- To develop a good sense of OO thought process:
 - ✧ Knowing the difference between the interface and implementation
 - ✧ Thinking more abstractly
 - ✧ Giving the user the minimal interface possible



The Interface



区分接口与实现

• 接口的标准化 vs. 实现的演进

class System

DataBaseReader

- + open() : void
- + close() : void
- + goToFirst() : void
- + goToLast() : void
- + howManyRecords() : int
- + areThereMoreRecords() : boolean
- + positionRecord() : void
- + getRecord() : char
- + getNextRecord() : char

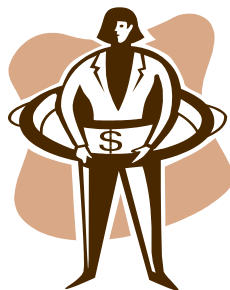
```
public void open(string name) {  
    /* some application-specific processing */  
    /* call the Oracle API to open the DB*/  
    /* more application specific processing */  
}
```

```
public void open(string name) {  
    /* some application-specific processing */  
    /* call the SQLAnywhere to open the DB*/  
    /* more application specific processing */  
}
```

抽象的接口



Abstract



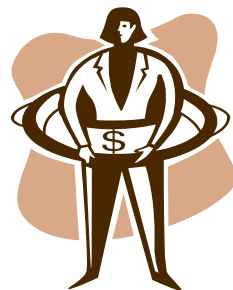
Take me to the Airport



不太抽象的接口



Not So Abstract



Turn Left

Turn Right

Turn Right

Turn Left

Turn Left



Giving the User the Minimal Interface Possible

- Provide the user with as little knowledge of the inner workings of the class as possible:
 - Give the users only what they absolutely need
只给必须的
 - Public interface are all the users will ever see
只看公开的
 - Design class from user's perspective, not IS/technical perspective
只为用户考虑

最小用户负担原则

Determine the Users

- Customers ? 50%
- **Services Principle**



Provide Service:
As long as I make a
profit



Engage Service:
As long as I don't pay
too much





Determine Object Behaviors

Use Cases!!!

Many earlier choice will not survive public
interface cuts!





Identify Environmental Constraints

Environments impose limitations on what an object
can do.

Pre/Post/Exceptions...



Identifying public Interfaces

- How to use taxi object:
 - Get into the taxi
 - Tell the cabbie where you want to go
 - Pay the cabbie
 - Give the cabbie a tip
 - Get out of the taxi.
- What to do to use Taxi:
 - Have a place to go
 - Hail a taxi
 - Pay the Cabbie Money



class System

Cabbie

```
+ hailTaxi() : void
+ enterTaxi() : void
+ greetCabbie() : void
+ specifyDestination() : void
+ payCabbie() : void
+ tipCabbie() : void
+ leaveTaxi() : void
```

Identifying the Implementation

- **Anything other than the public interface** can be considered the implementation
- Users will **never** see any of it **ever**
 - methods signature (name and parameter list)
 - Actual code inside
- Anything considered implementation can **change w/o affecting the interface**
- Implementation provide **answers** to the user **expects**
- Interface provides how user sees the object, while the implementation is the **nuts and bolts** of the objects
- Implementation contains the **code** that **represents the state** of an objects

Object-Oriented Analysis (OOA)

- OOA is a collection of like-minded requirements modeling and analysis techniques for software systems.
- Proposed in the late 80's, such techniques have been influenced primarily by **object-oriented programming**, but also **semantic data models** and **semantic networks**.
- **Basic idea**: streamline software development by making **objects**, **classes**, **methods** and the like the atomic units out of which one builds requirements, designs and implementations.



Basic Concepts

- Such techniques focus on the **things** that exist within the application domain, model them with **objects**.
- These techniques use **classification**, **generalization**, **aggregation** to structure object assemblies.
- **Actions** (services/activities) are associated with objects.
- **State changes** are effected by actions performed on objects.



Origins of Object-oriented Analysis

- **Object-Oriented Programming** – Tries to adopt as many of the O-O programming features to O-O design and analysis [Booch86]
- **Database design** – adopts semantic data modeling ideas, including E-R diagrams and generalization, aggregation, classification [Chen 76]
- **Structured Analysis** – including SADT and other Structured analysis techniques [Ross 77]
- **Knowledge Representation** – uses ideas from **frame** based and **semantic network** representations [Borgida85]



Coad's Object-oriented Analysis

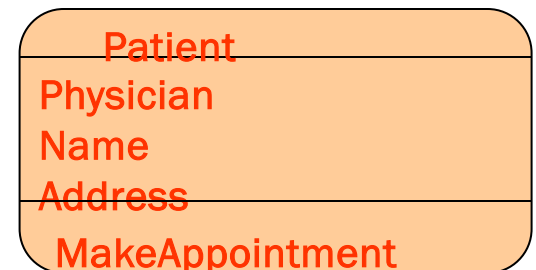
- Proposed by Peter Coad [*Coad91*]
- An **object** is defined as a real world entity related to the problem domain, with “crisply defined boundaries”.
- Objects are encapsulated with attributes and behavior.
- OOA offers five kinds of concepts: **objects**, **attributes**, **structures**, **services** and **subjects**.



Identifier

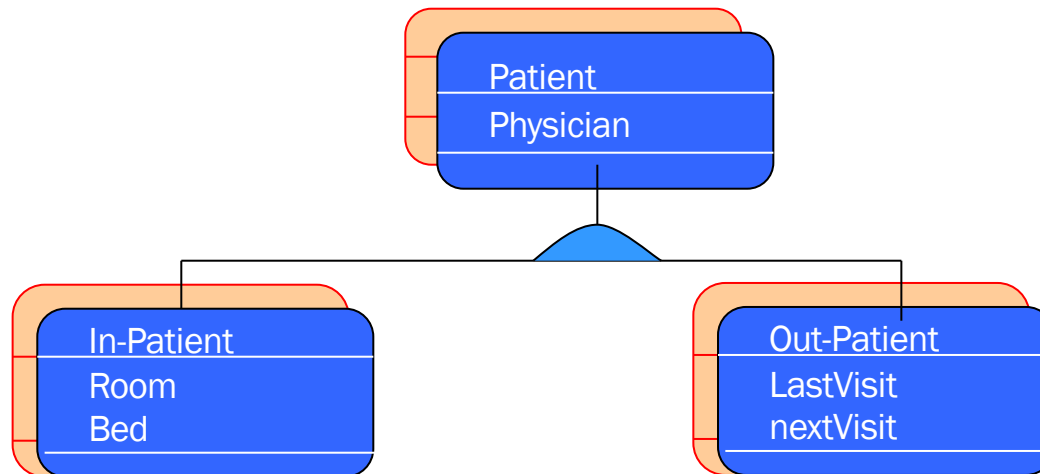
Attributes

Services



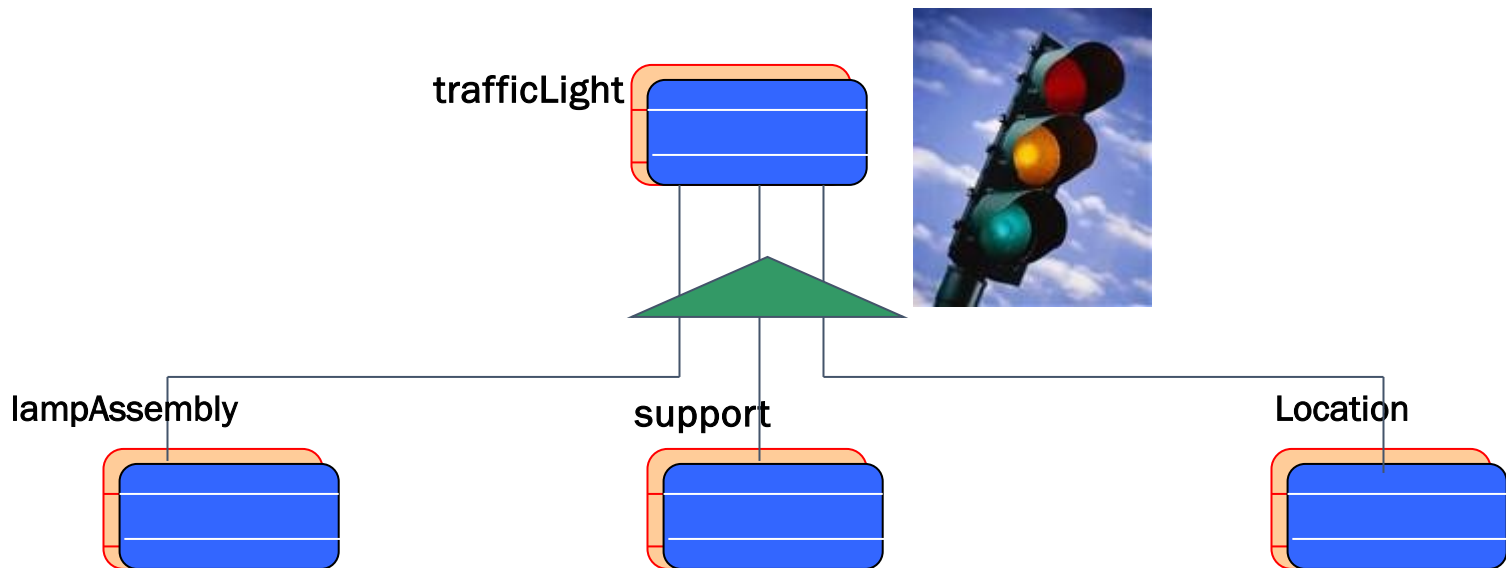
Gen-Spec Structures

- Gen-Spec (generalization/specialization) structures organize classes into taxonomies. **Patients** are either **in-patients** or **out-patients**. The **physician** attribute of **patients** is inherited by both **in-patients** and **out-patients**.



Whole-Part Structures

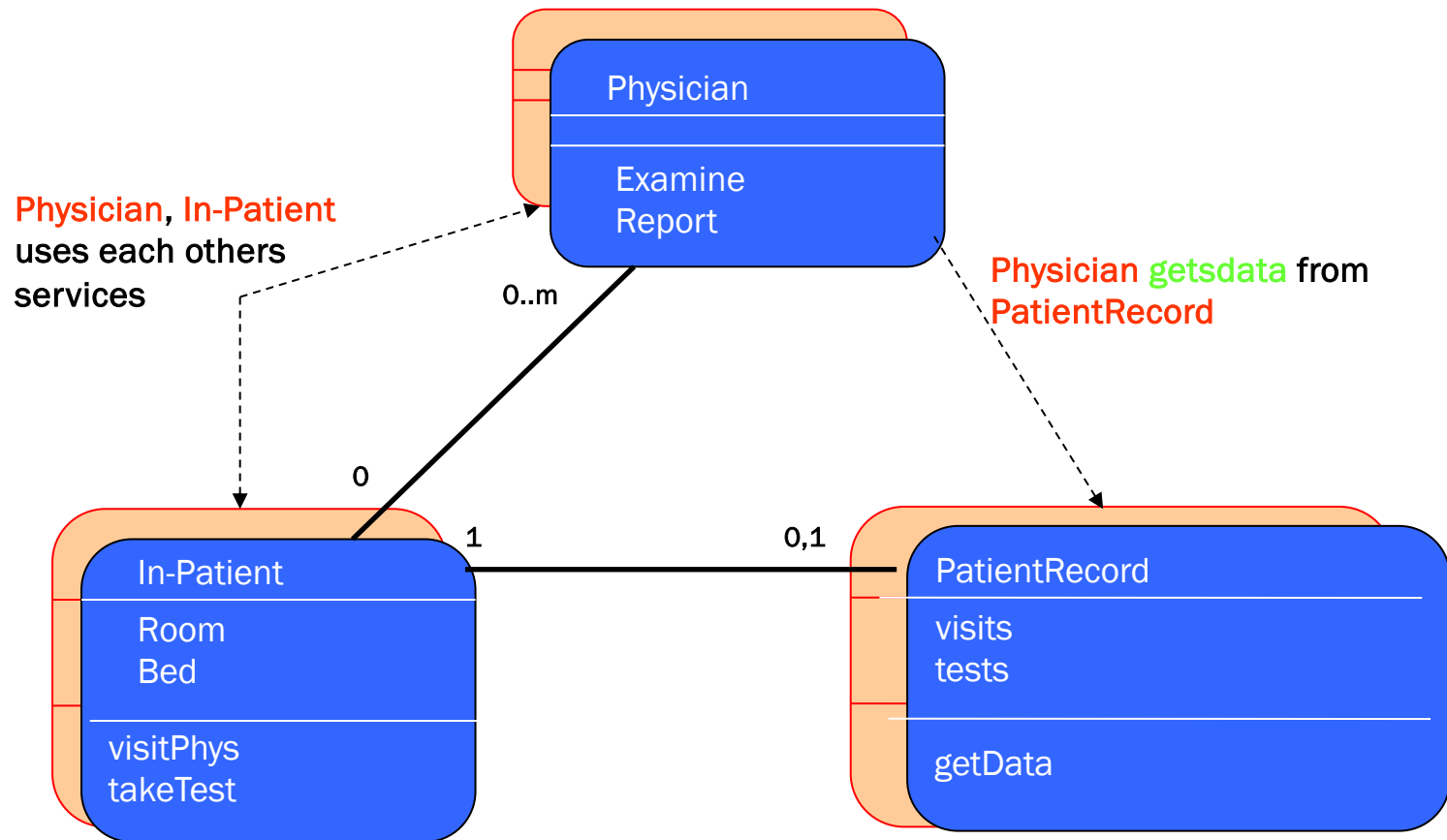
- Whole-Part structures describe an object as an assembly of other objects. A **traffic light** consists of 0 to 3 **lamp Assemblies**, a single **support** and a single **location**.



Services

- Objects provide services to other objects in their environments. For example, a physician object may provide services **examine**, **report**.
- Coad distinguishes three types of services:
 - **Occurrence services**, whereby objects are created, destroyed, changed, ...;
 - **Calculate services**, where an object performs a calculation for some other object;
 - **Monitor services**, where an object is monitoring some process to see if some condition applies;
- A special notation is used (dashed-line arrow) to indicate that an object is using services from another object.

Services and relationships





Methodology

- Identify objects and **classes** (i.e., generic objects)
- Identify **structures** and build generalization, aggregation hierarchies.
- Define **subjects**. These partition all the objects and classes of an object model into subject layers, which represent the application from a particular perspective. Often whole Gen-Spec or Part-Whole structures are grouped under one subject.
- Identify information that should be associated with each object. Place **attributes** at the right structural level.
- Define **services** for each class.

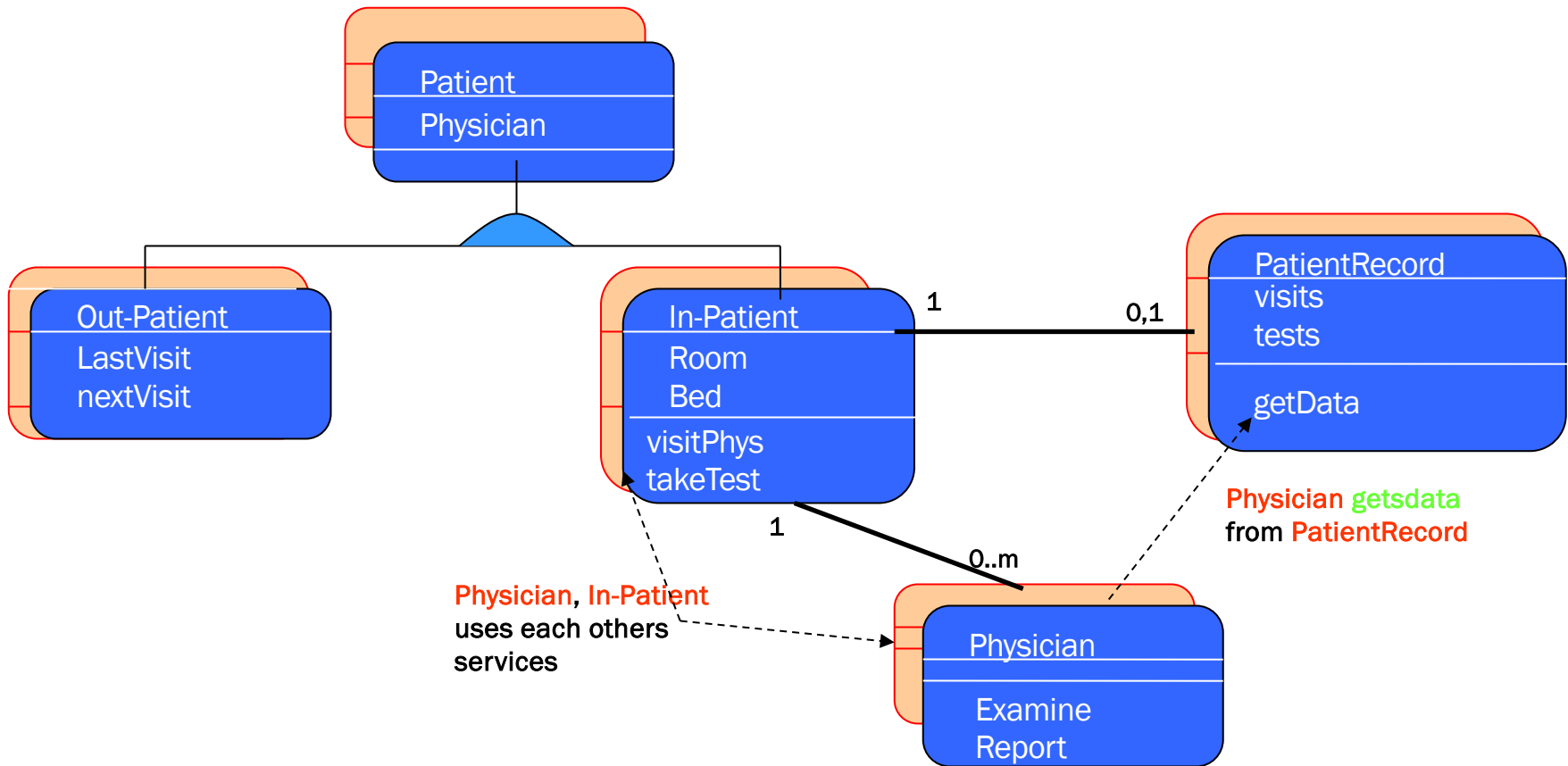
What is Good about OOA?

- Advances the state-of-practice!
- Earlier prevalent modeling techniques, such as structured analysis (SA), data flow diagrams (DFD), entity-relationship diagrams (ERD), ... were:
 - **Fragmented**, e.g., use of data flow and entity-relationship diagrams
 - **Weakly structured**: see DFD and ERD
 - **Informal**: see DFD
 - Based on an **outdated** programming paradigm (structured programming)

What is questionable about OOA?

- Is OOA intended for modeling **software**, or **applications**? If only the former, then it is meant for requirements modeling!
- Its ontological assumptions. Who says that **objects**, **relationships**, **services** and the like constitute a good set of primitive concepts for modeling the real world?...organizations?...people? ...industrial processes?
- The promise of easier design and implementation won't work for large systems where requirements, design and implementation have drastically different architectures and are based on very different concepts.

What does this model? —A system, or the world?

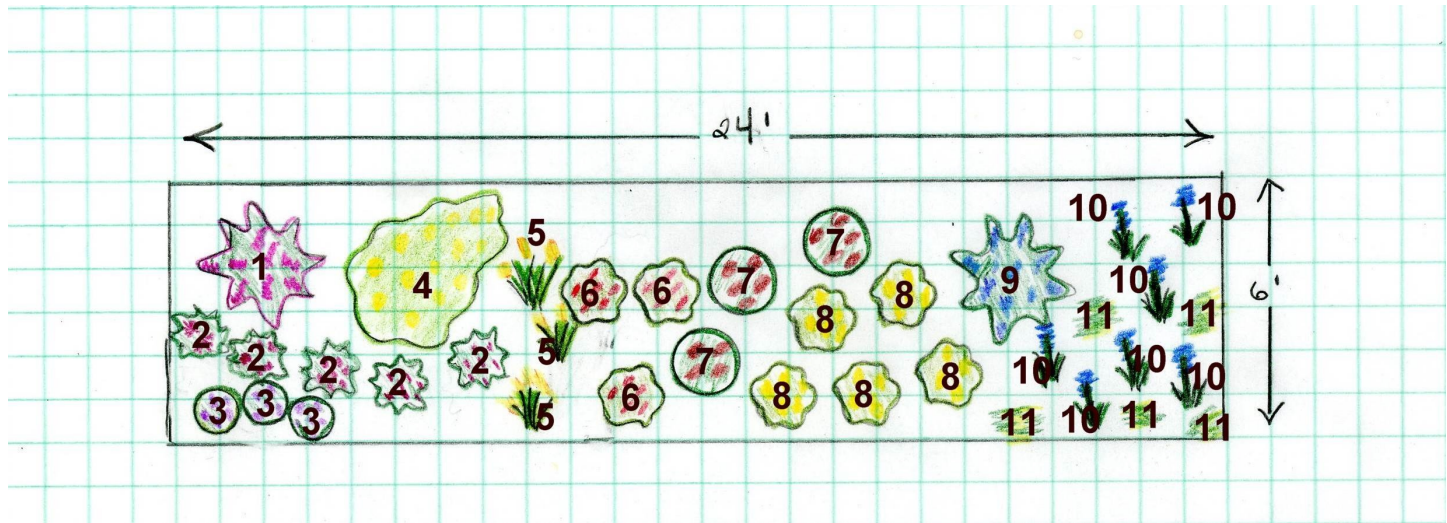


The ongoing design process

Despite the best intentions and planning, in all but the most trivial cases, the design is an **ongoing** process...

Even after a product is in testing, **design changes** pop up.

It is up to the **project manager** to draw the line that says **when** to stop changing a product and adding features.



Designing with Objects

The truth is:

You can create a very bad OO design just as easily as you can create a very bad non-OO design...



Object Oriented Analysis

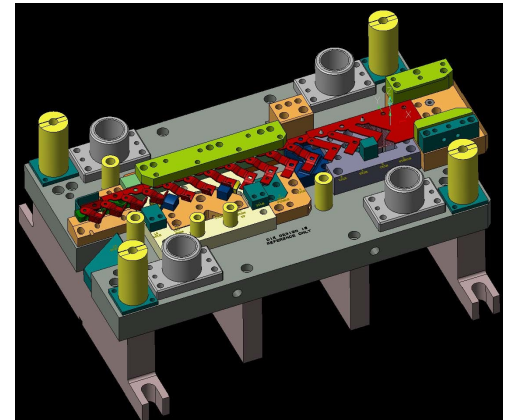
- Background
 - Model the requirements in terms of **objects** and the **services** they provide
 - Grew out of object oriented design
 - Applied to modelling **the application domain** rather than the program
- Motivation
 - OO is (claimed to be) more '**natural**'
 - As a system evolves, the functions it performs need to be changed more often than the objects on which they operate...
 - ...a model based on objects (rather than functions) will be more stable over time...
 - ...hence the claim that object-oriented designs are more maintainable
 - OO emphasizes importance of **well-defined interfaces** between objects
 - compared to ambiguities of dataflow relationships

NOTE: OO applies to requirements engineering because it is a modeling tool. But we are modeling domain objects, not the design of the new system

- Questions?

A solid OO design process

- Doing the proper analysis
- Developing a statement of work that describes the system
- Gathering the requirements from this statement of work
- Developing a prototype for the user interface
- Identifying the **classes**
- Determining the **responsibilities** of each class
- Determining how the various classes **interact** with each other
- Creating a high-level model that describes the system to be



CRC Card

Class:	Classname		
	Responsibilities	Collaborations	

Blackjack

- The objective is to get a card total that is closest or equal to 21.
- To win, you must beat the Dealer's hand without exceeding a card total of 21.
- You always play against the Dealer, no matter how many players are at the table.
- In Blackjack, face cards, cards from 2 to 10, carry their face value.
- An Ace can either be counted as 1 or 11, whichever is better for the owner.
- Picture cards, Jack, Queen, and King, carry a value of 10.





Blackjack

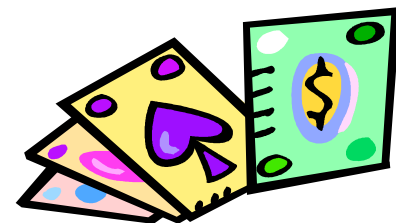
- To begin, you need to place a bet. Most casinos define a limit on the bet that you can place.
- If you are the only player at table, after you place your bet, you are dealt two cards face up. You and the Dealer both can see those cards.
- The Dealer is also dealt two cards, one card face up and the card face down. The Dealer's hidden card is called the Hole Card, and the visible card is called Up Card.
- If multiple players are playing at the table, each player is dealt one card face up and the Dealer is dealt the hole card. Again, each player is dealt another card face up. Consecutively, the Dealer is dealt the up card.

Blackjack

- After the cards are dealt, you can decide to improve your hand by drawing more cards. Once you are satisfied with your hand, you fold and stand.
- After you stand, the Dealer tries to improve their hand, without exceeding a card total of 21. If your hand beats the Dealer's you win, otherwise you lose.
- You win when:
 - You get a hand better than the Dealer's, without exceeding the total of 21.
 - The Dealer goes bust, you win automatically.
 - Remember, in case you and the Dealer get identical hand total, it ends up in a tie. And in that case, your bet is returned to you.
- You lose when:
 - Your hand total exceeds 21.

Let's play some game 😊

- We will create a program that simulates a game of Blackjack.
- Assume that a customer has come to you with a proposal that includes a very well-written **statement of work** and **a rule book** about how to play blackjack.
- We are going to **design** the system, not how to **implement** the game...
- This will culminate in the discovery of the classes, along with their responsibilities and collaborations.



Identifying the Blackjack Classes

Classes correspond to nouns... but...

Remember: Don't try to get things right the first time...



Nearly anything can be an object...

Source: Adapted from Pressman, 1994, p242

- External Entities

- ...that interact with the system being modeled
 - E.g. people, devices, other systems

- Things

- ...that are part of the domain being modeled
 - E.g. reports, displays, signals, etc.

- Occurrences or Events

- ...that occur in the context of the system
 - E.g. transfer of resources, a control action, etc.

- Roles

- played by people who interact with the system

- Organizational Units

- that are relevant to the application
 - E.g. division, group, team, etc.

- Places

- ...that establish the context of the problem being modeled
 - E.g. manufacturing floor, loading dock, etc.

- Structures

- that define a class or assembly of objects
 - E.g. sensors, four-wheeled vehicles, computers, etc.

Some things cannot be objects:

- *procedures (e.g. print, invert, etc)*
- *attributes (e.g. blue, 50Mb, etc)*

Blackjack Game

- The objective is to get a card total that is closest or equal to 21. To win, you must beat the **Dealer's hand** without exceeding a card total of 21.
- You always play against the Dealer, no matter how many **players** are at the **table**.
- Card Ranks
- In Blackjack, **face cards**, **cards** from 2 to 10, carry their **face value**. An **Ace** can either be counted as 1 or 11, whichever is better for **the owner**.
- **Picture cards**, **Jack**, **Queen**, and **King**, carry a value of 10.



Blackjack

- To begin, you need to place a **bet**. Most casinos define a limit on the bet that you can place.
- If you are the only player at table, after you place your bet, you are dealt two cards face up. You and the Dealer both can see those cards. The Dealer is also dealt two cards, one card face up and the card face down. The Dealer's hidden card is called the **Hole Card**, and the **visible card** is called Up Card.
- If multiple players are playing at the table, each player is dealt one card face up and the Dealer is dealt the hole card. Again, each player is dealt another card face up. Consecutively, the Dealer is dealt the up card.

Blackjack

- After the cards are dealt, you can decide to improve your hand by drawing more cards. Once you are satisfied with your hand, you fold and stand. After you stand, the Dealer tries to improve their hand, without exceeding a card total of 21. If your hand beats the Dealer's you win, otherwise you lose.
- You win when:
 - You get a hand better than the Dealer's, without exceeding the total of 21.
 - The Dealer goes bust, you win automatically.
 - Remember, in case you and the Dealer get identical hand total, it ends up in a tie. And in that case, your bet is returned to you.
- You lose when:
 - Your hand total exceeds 21.

A list of possible objects

- Game
- Blackjack
- Dealer
- Players
- Player
- Cards
- Card
- Deck
- Hand
- Face Value
- Suit
- Winner
- Ace
- Face card
- King
- Queen
- Jack
- Bet



Initial BlackJack Classes

Card

Deck

Hand

Dealer

Player

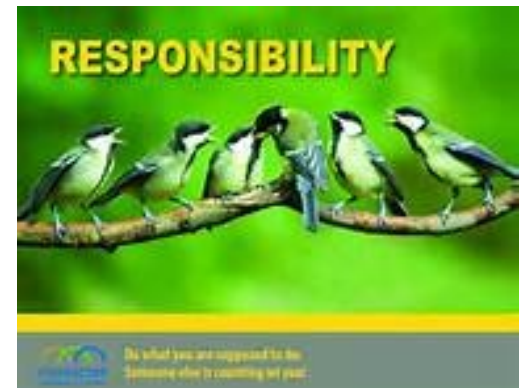
Bet

Identifying the Classes' Responsibilities

Responsibilities relate to actions, so select the verbs...

Remember:

1. Not all verbs will end up as responsibilities
2. Need to combine several verbs sometimes
3. Some may popup later
4. Keep revising and updating both
5. When two classes share a resp., each class have it.



Possible responsibilities for our classes

- Card
 - Know its face value
 - Know its suit
 - Know its value
 - Know whether it is a face card
 - Know whether it is an ace
 - Know whether it is a joker
- Bet
 - Know the type of bet
 - Know the value of the current bet
 - Know how much the player has left to bet
 - Know whether the bet can be covered
- Deck
 - Shuffle
 - Deal the next card
 - Know how many cards are left in the deck
 - Know whether there is a full deck to begin
- Hand
 - Know how many cards are in the hand
 - Know the value of the hand
 - Show the hand

Possible responsibilities for our classes

- Dealer

- Deal the cards
- Shuffle the deck
- Give a card to a player
- Show the dealer hand
- Calculate the value of the dealer's hand
- Know the number of cards in the dealer's hand
- Request a card
- Determine the winner
- Start a new hand

- Player

- Request a card
- Show the player's hand
- Calculate the value of the player's hand
- Know how many cards are in the hand
- Know whether the hand value is over 21
- Know whether the hand value is equal to 21 (and if it is a blackjack)
- Know whether the hand is below 21

Identifying the Collaborations: Using UML Use Cases

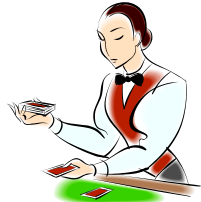
Identify the objects and the messages exchanges...

Remember:

The purpose is not to document all possible scenarios, but to refine the classes and responsibilities ...

An Example Scenario

- Dealer



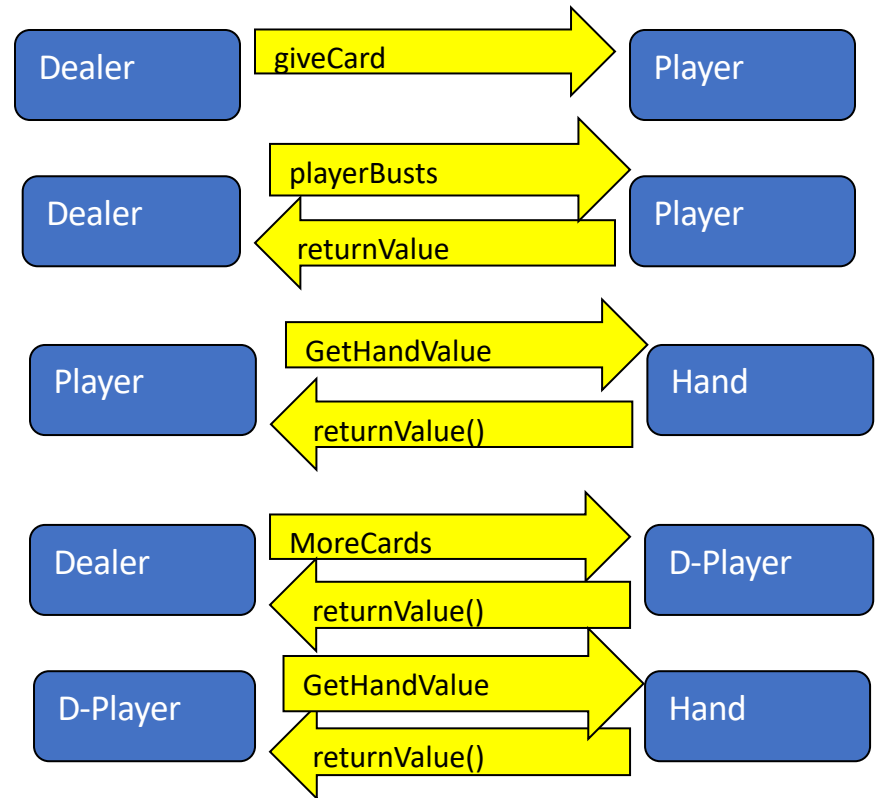
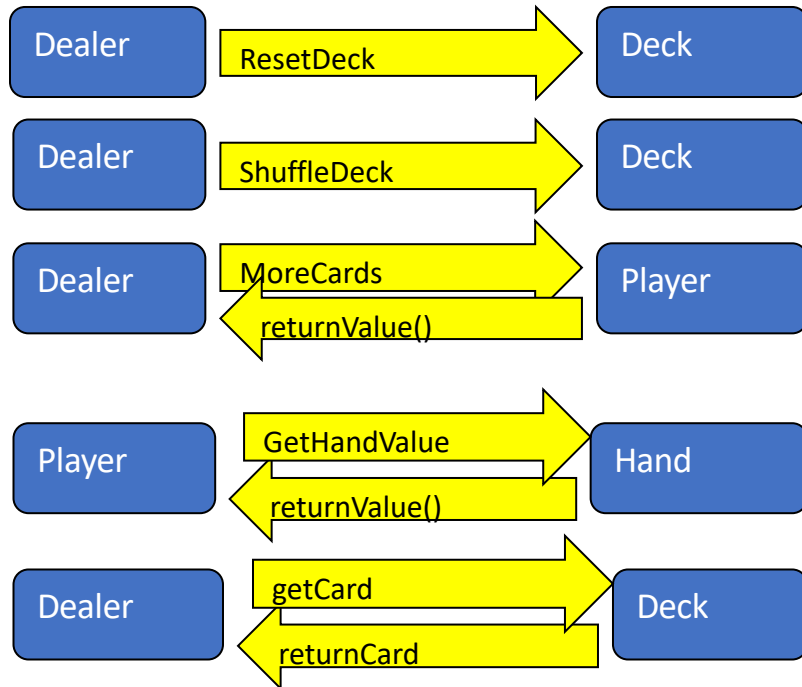
- Dealer shuffles deck
- Dealer deals initial card
- Dealer adds cards to dealer's hand
- Hand returns value of dealer's hand to dealer
- Dealer deals player another card
- Dealer asks player whether he wants another card
- Dealer gets the value of the player's hand
- Dealer sends or requests bet value from players

- Player



- Player makes bet
- Player adds card to player's hand
- Hand returns value of player's hand to player
- Dealer asks player whether he wants another card
- Player adds the card to player's hand
- Hand returns value of player's hand to player
- Player adds to/subtracts from player's bet attribute

Determine Collaborations



Card

Class:		Card	
	Responsibilities	Collaborations	
	Get name	Deck	
	Get Value		

Deck

Class: Deck		
	Responsibilities	Collaborations
	Reset deck	Dealer
	Get deck size	Card
	Get next card	
	Shuffle Deck	
	Show deck	

Dealer

Class: Dealer		
	Responsibilities	Collaborations
	Start a new game	Hand
	Get a card	Player
		Deck

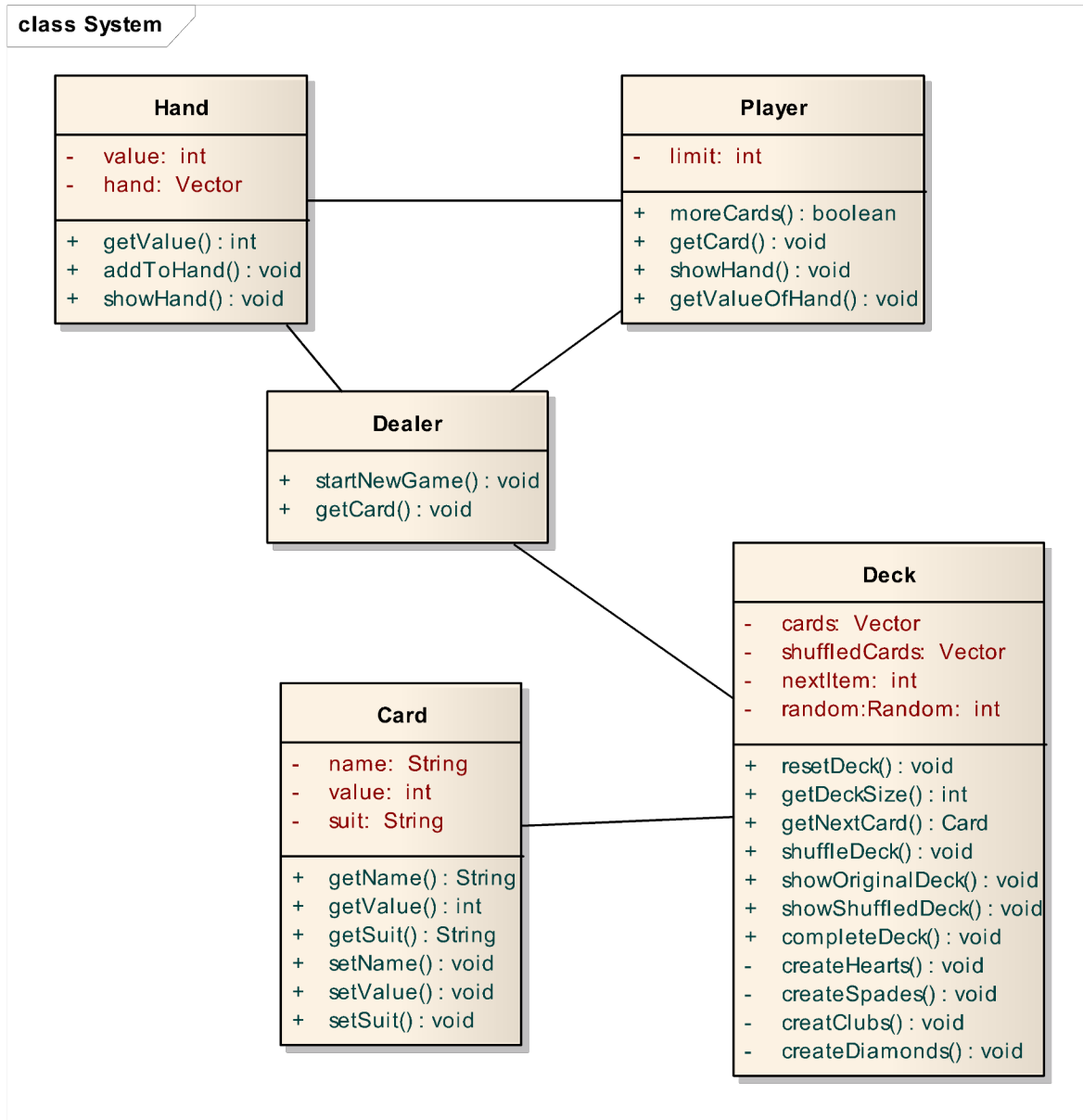
Player

Class: Player		
	Responsibilities	Collaborations
	Want more cards	Dealer
	Get a card	Hand
	Show hand	
	Get value of hand	

Hand

Class: Hand		
	Responsibilities	Collaborations
	Return Value	Dealer
	Add a card	Player
	Show hand	

UML Class diagram for BlackJack

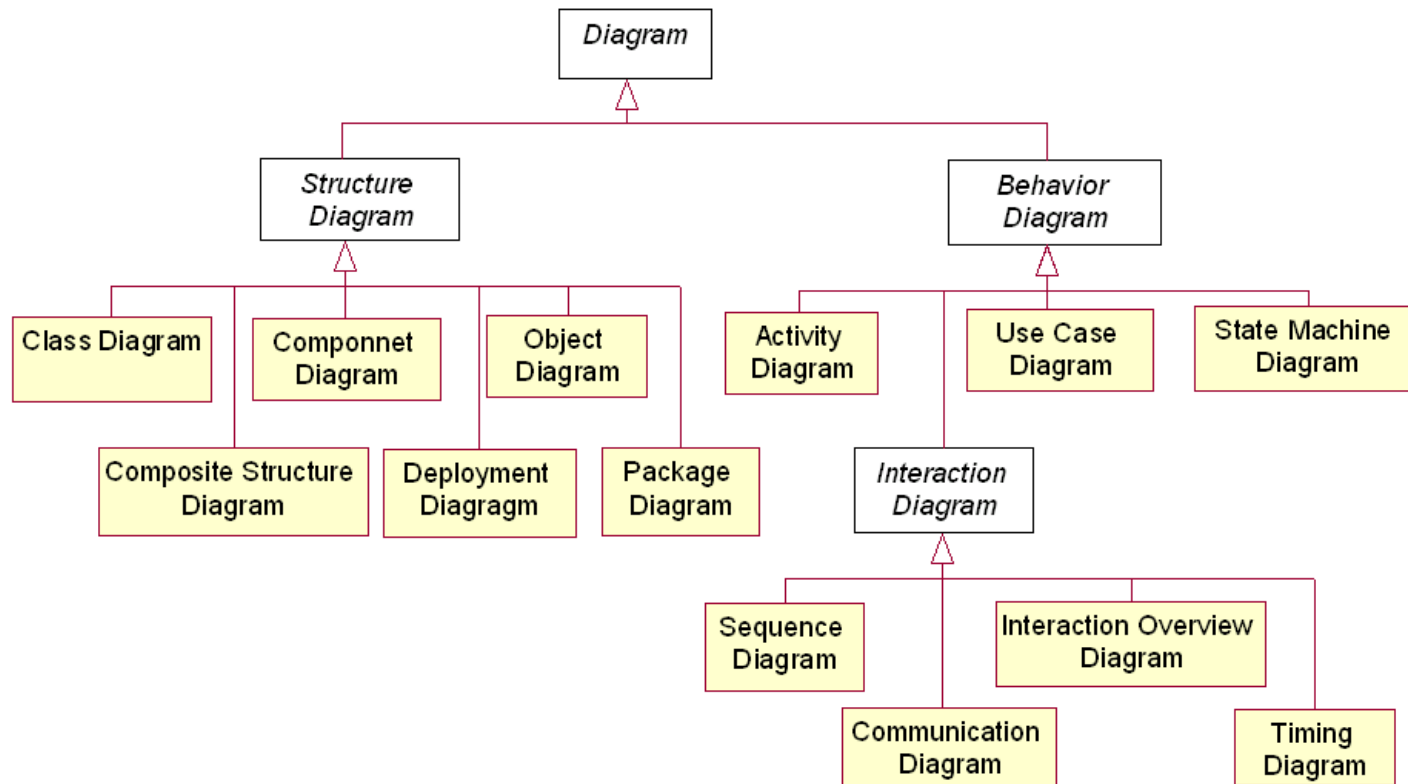


Object Oriented Modelling

- Object Oriented Analysis
 - Rationale
 - Identifying Classes
 - Attributes and Operations
- UML Class Diagrams
 - Associations
 - Multiplicity
 - Aggregation
 - Composition
 - Generalization



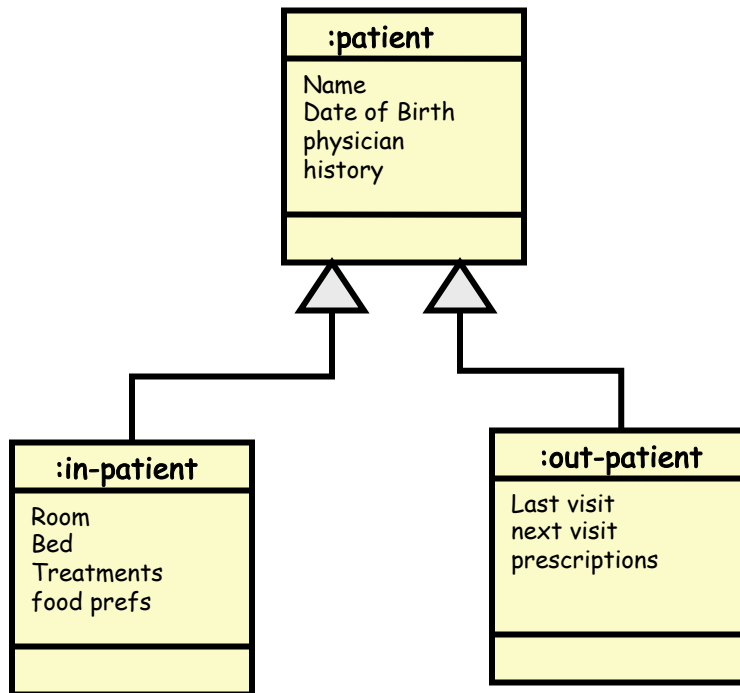
Classification of Diagrams in the UML 2.0



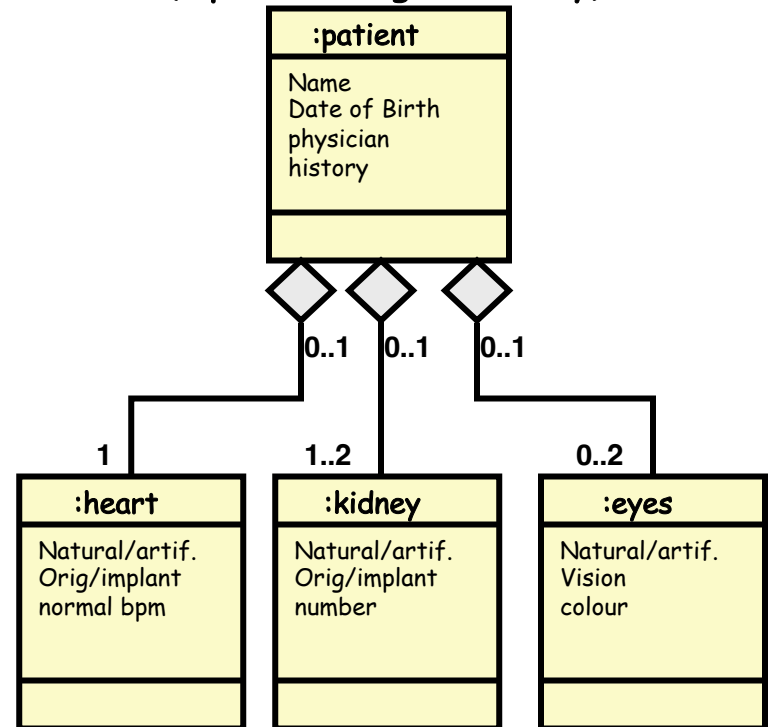
A brief UML example

Source: Adapted from Davis, 1990, p67-68

Generalization
(an abstraction hierarchy)

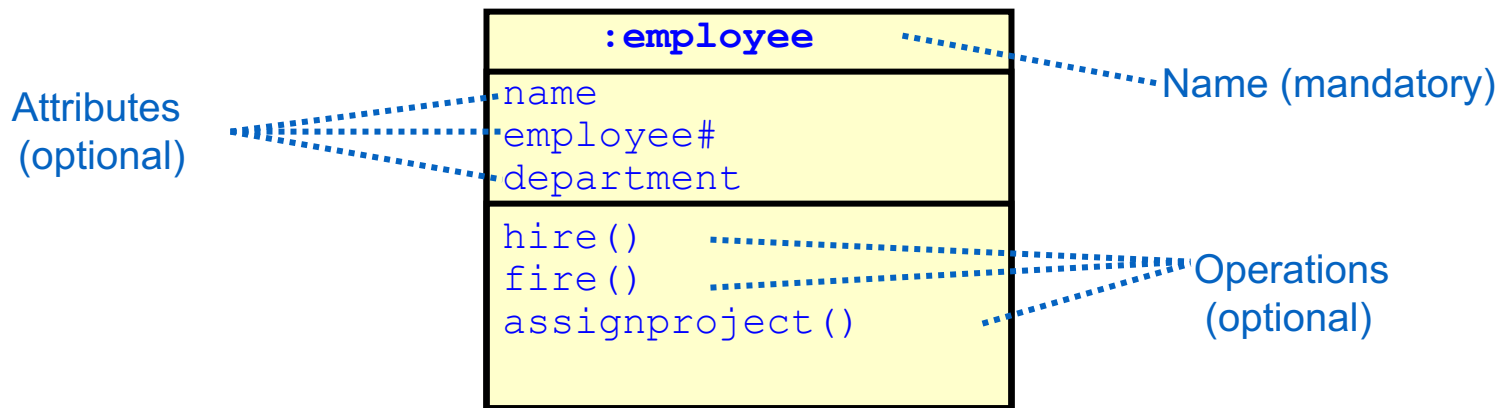


Aggregation
(a partitioning hierarchy)



What are classes?

- A class describes a group of objects with
 - similar properties (attributes),
 - common behaviour (operations),
 - common relationships to other objects,
 - and common meaning ("semantics").
- Examples
 - **employee**: has a name, employee# and department; an employee is hired, and fired; an employee works in one or more projects



Objects vs. Classes

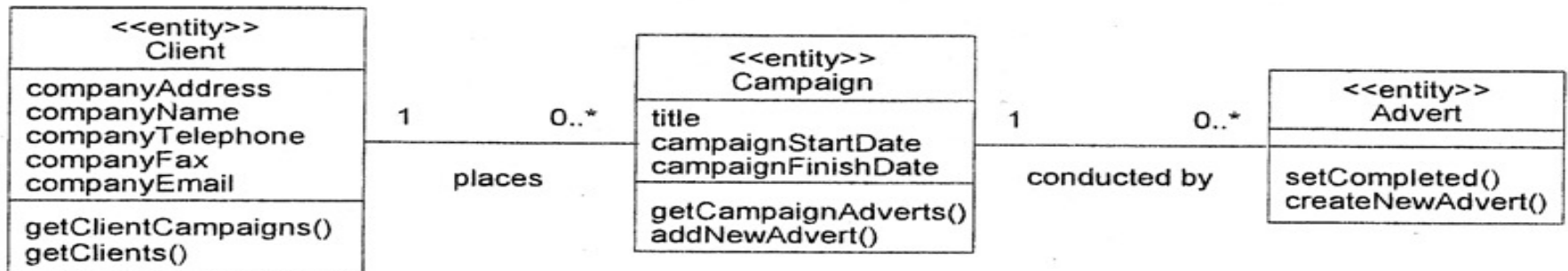
- The instances of a class are called objects.
 - Objects are represented as:

Fred_Bloggs:Employee
name: Fred Bloggs Employee #: 234609234 Department: Marketing

- Two different objects may have identical attribute values (like two people with identical name and address)
- Objects have associations with other objects
 - E.g. Fred_Bloggs:employee is associated with the KillerApp:project object
 - But we will capture these relationships at the class level (why?)
 - Note: Make sure attributes are associated with the right class
 - E.g. you don't want both managerName and manager# as attributes of Project! (...Why??)

Associations

- Objects do not exist in isolation from one another
 - A relationship represents a connection among things.
 - In UML, there are different types of relationships:
 - Dependency
 - Association
 - Aggregation and Composition
 - Generalization
 - Realization
 - Note: The last two are not useful during requirements analysis
- Class diagrams show classes and their relationships

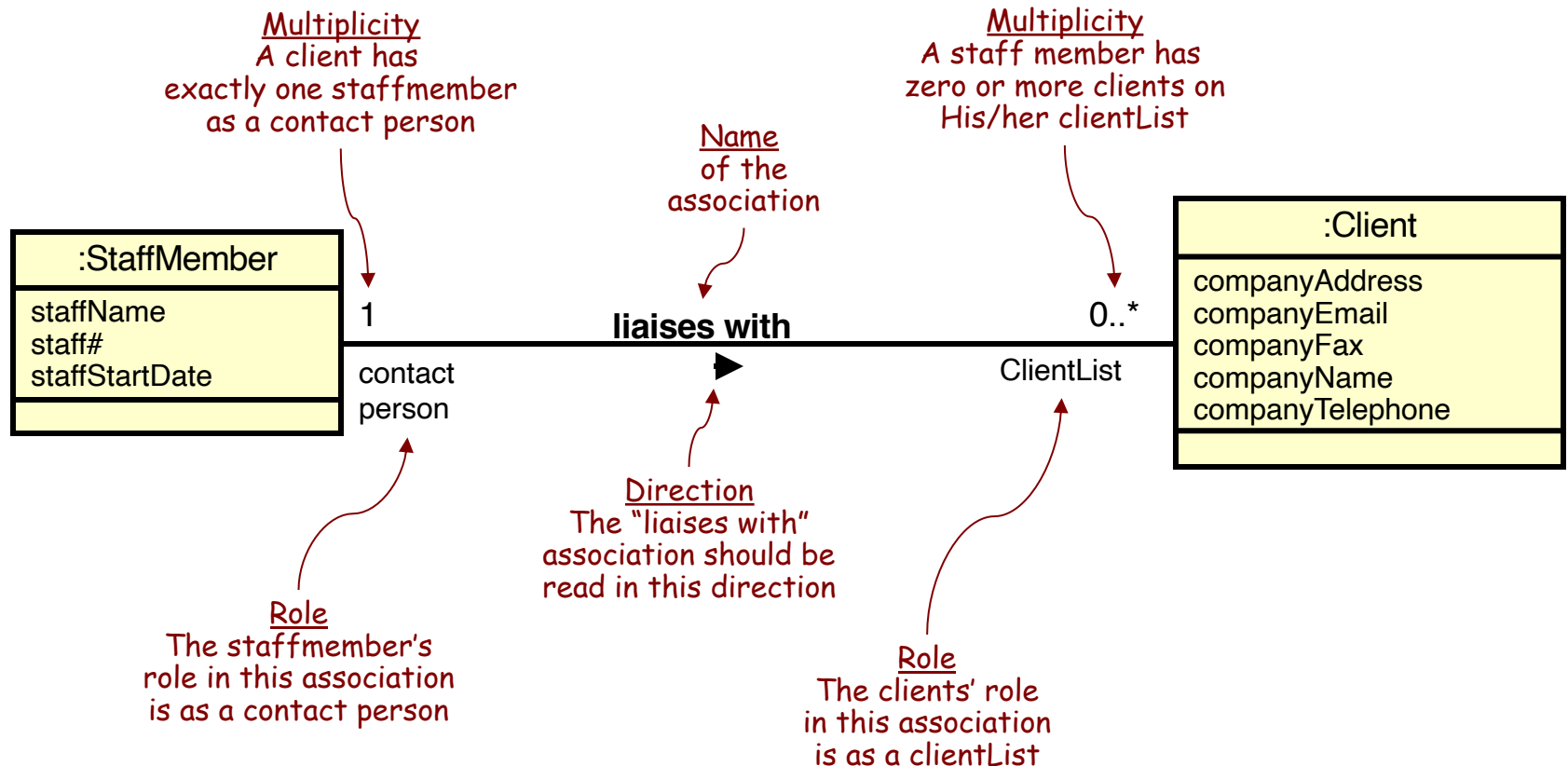


Association Multiplicity

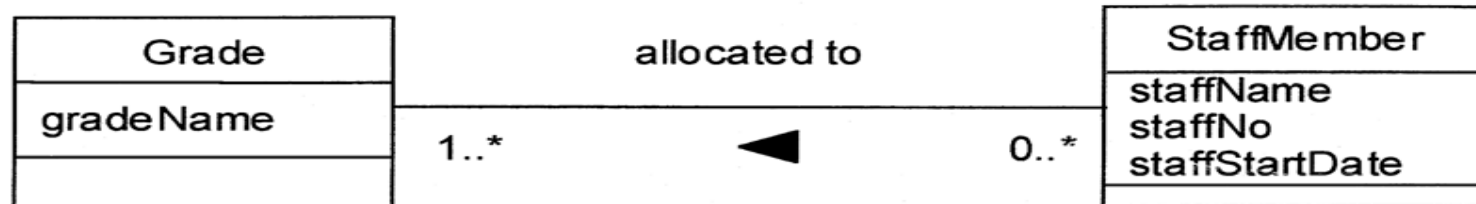
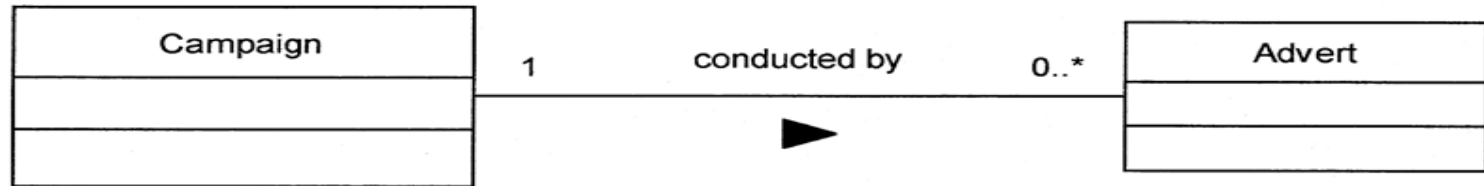
- Ask questions about the associations:
 - **Can a campaign exist without a member of staff to manage it?**
 - If yes, then the association is optional at the Staff end - zero or more (0..*)
 - If no, then it is not optional - one or more (1..*)
 - If it must be managed by one and only one member of staff - exactly one (1)
 - **What about the other end of the association?**
 - Does every member of staff have to manage exactly one campaign?
 - No. So the correct multiplicity is zero or more.
- Some examples of specifying multiplicity:

• Optional (0 or 1)	0..1	
• Exactly one	1	= 1..1
• Zero or more	0..*	= *
• One or more	1..*	
• A range of values	2..6	

Class associations

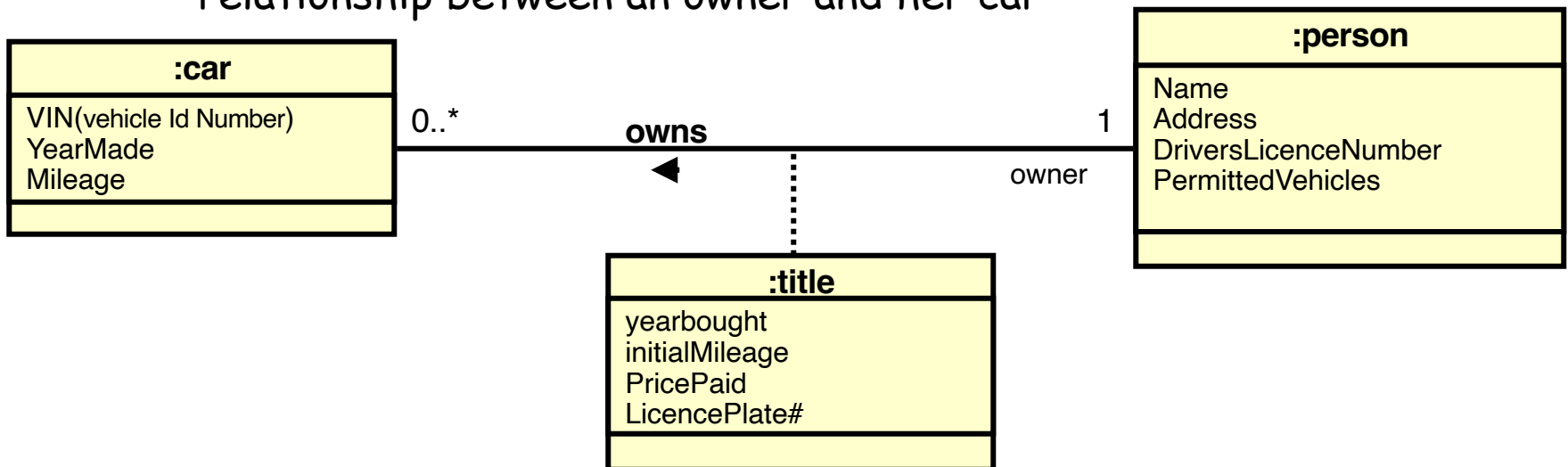


More Examples



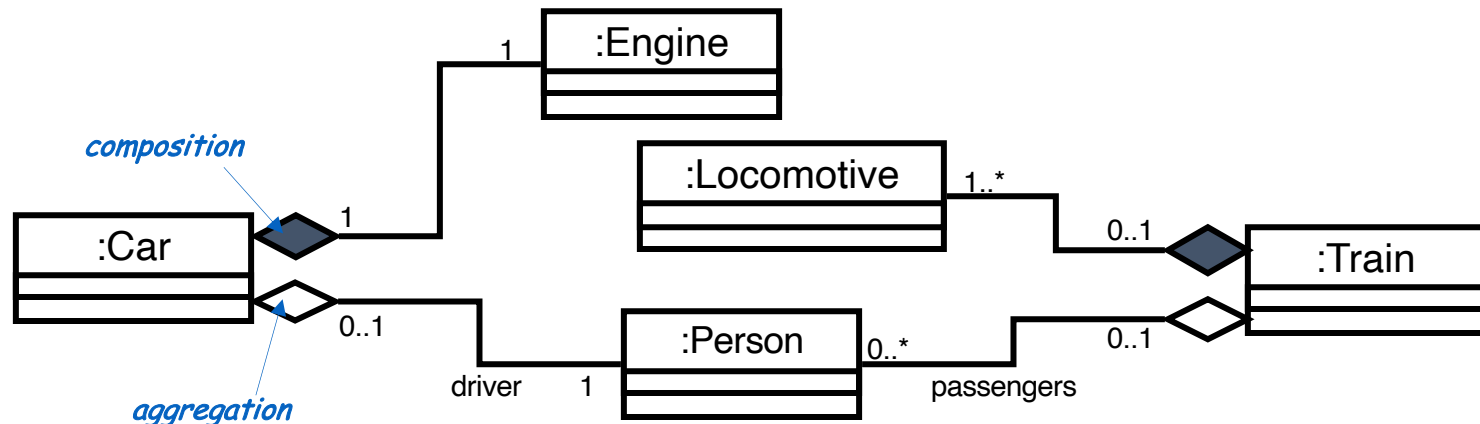
Association Classes

- Sometimes the association is itself a class
 - ...because we need to retain information about the association
 - ...and that information doesn't naturally live in the classes at the ends of the association
 - E.g. a "title" is an object that represents information about the relationship between an owner and her car



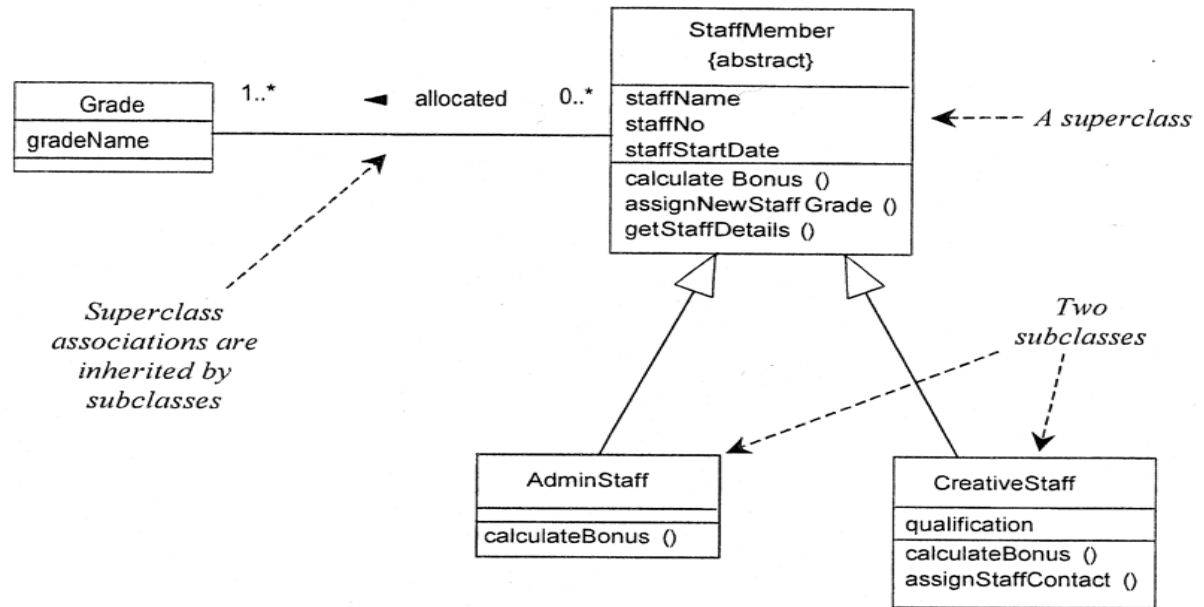
Aggregation and Composition

- Aggregation
 - This is the “**Has-a**” or “**Whole/part**” relationship
- Composition
 - Strong form of aggregation that implies ownership:
 - if the whole is removed from the model, so is the part.
 - the whole is responsible for the disposition of its parts



Generalization

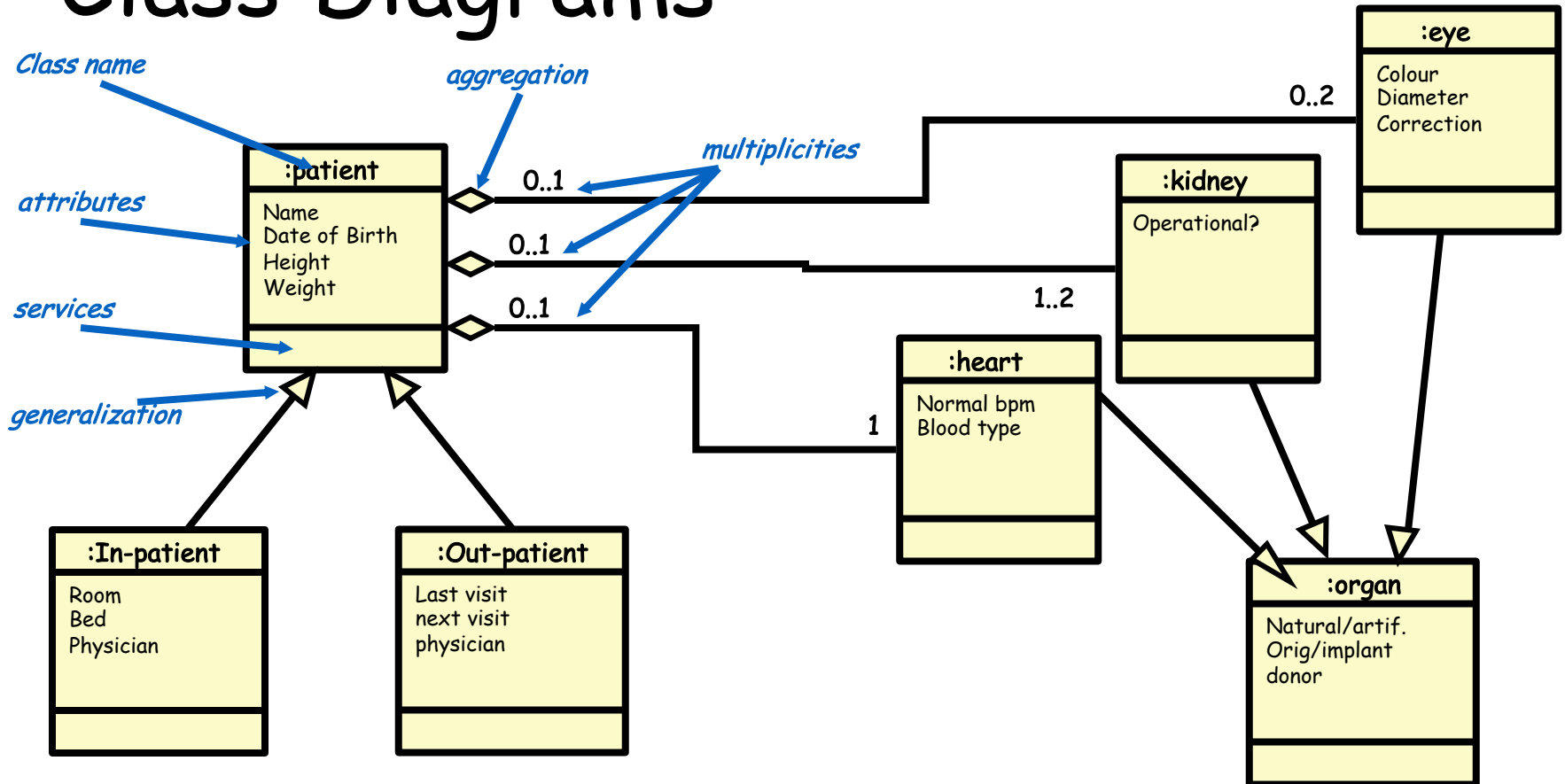
- Subclasses inherit attributes, associations, & operations from the superclass
- A subclass may override an inherited aspect
 - e.g. AdminStaff & CreativeStaff have different methods for calculating bonuses
- Superclasses may be declared **{abstract}**, meaning they have no instances
 - Implies that the subclasses cover all possibilities
 - e.g. there are no other staff than AdminStaff and CreativeStaff



More on Generalization

- Usefulness of generalization
 - Can easily add new subclasses if the organization changes
- Look for generalizations in two ways:
 - **Top Down**
 - You have a class, and discover it can be subdivided
 - Or you have an association that expresses a "kind of" relationship
 - *E.g. "Most of our work is on advertising for the press, that's newspapers and magazines, also for advertising hoardings, as well as for videos"*
 - **Bottom Up**
 - You notice similarities between classes you have identified
 - *E.g. "We have books and we have CDs in the collection, but they are all filed using the Dewey system, and they can all be lent out and reserved"*
- But don't generalize just for the sake of it
 - Be sure that everything about the superclass applies to the subclasses
 - Be sure that the superclass is useful as a class in its own right
 - I.e. not one that we would discard using our tests for useful classes
 - Don't add subclasses or superclasses that are not relevant to your analysis

Class Diagrams



Modelling principles

- Facilitate Modification and Reuse
 - **Experienced analysts reuse their past experience**
 - they reuse components (of the models they have built in the past)
 - they reuse structure (of the models they have built in the past)
 - **Smart analysts plan for the future**
 - they create components in their models that might be reusable
 - they structure their models to make them easy to modify
- Helpful ideas:
 - **Abstraction**
 - strip away detail to concentrate on the important things
 - **Decomposition (Partitioning)**
 - Partition a problem into independent pieces, to study separately
 - **Viewpoints (Projection)**
 - Separate different concerns (views) and describe them separately
 - **Modularization**
 - Choose structures that are stable over time, to localize change
 - **Patterns**
 - Structure of a model that is known to occur in many different applications

Modelling Principle 1: Partitioning

- Partitioning
 - captures aggregation/part-of relationship
- Example:
 - goal is to develop a spacecraft
 - partition the problem into parts:
 - guidance and navigation;
 - data handling;
 - command and control;
 - environmental control;
 - instrumentation;
 - etc
 - Note: this is not a design, it is a problem decomposition
 - actual design might have any number of components, with no relation to these sub-problems
 - However, the choice of problem decomposition will probably be reflected in the design



Modelling Principle 2: Abstraction

Source: Adapted from Davis, 1990, p48 and Loucopoulos & Karakostas, 1995, p78

- Abstraction
 - A way of finding similarities between concepts by ignoring some details
 - Focuses on the general/specific relationship between phenomena
 - Classification groups entities with a similar role as members of a single class
 - Generalization expresses similarities between different classes in an 'is_a' association
- Example:
 - requirement is to handle faults on the spacecraft
 - might group different faults into fault classes

based on location:

- ↳ instrumentation fault,
- ↳ communication fault,
- ↳ processor fault,
- ↳ etc

OR

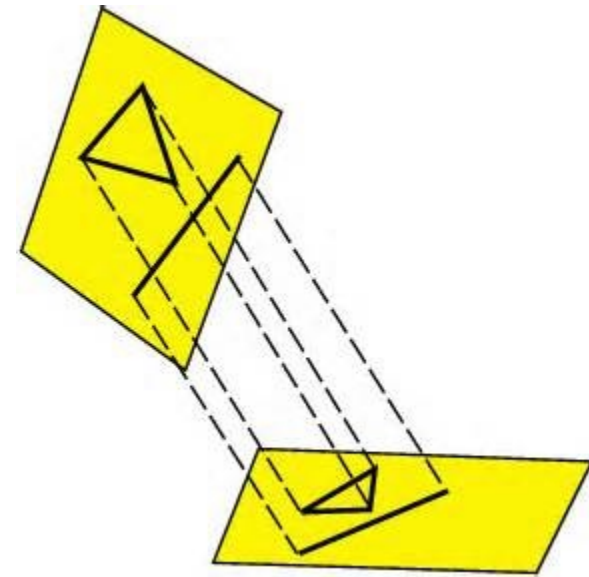
based on symptoms:

- ↳ no response from device;
- ↳ incorrect response;
- ↳ self-test failure;
- ↳ etc...

Modelling Principle 3: Projection

Source: Adapted from Davis, 1990, p48-51

- Projection:
 - **separates aspects of the model into multiple viewpoints**
 - similar to projections used by architects for buildings
- Example:
 - **Need to model the requirements for a spacecraft**
 - **Model separately:**
 - safety
 - commandability
 - fault tolerance
 - timing and sequencing
 - Etc...
- Note:
 - **Projection and Partitioning are similar:**
 - Partitioning defines a 'part of' relationship
 - Projection defines a 'view of' relationship
 - **Partitioning assumes the parts are relatively independent**



Summary

- Understand the objects in the application domain
 - **Identify all objects that stakeholders refer to**
 - **Decide which objects are important for your analysis**
 - **Class diagrams good for:**
 - Visualizing relationships between domain objects
 - Exploring business rules and assumptions via multiplicities
 - Specifying the structure of information to be (eventually) stored
- OO is a good way to explore **details** of the problem
 - Avoids the fragmentary nature of structured analysis
 - provides a coherent way of understanding the world
- But beware...
 - **temptation to do design rather than problem analysis**
 - In RE, class diagrams DO NOT represent programming (e.g. Java) classes
 - **For analysis, use UML diagrams as sketches, not as blueprints**
 - But may become blueprints when used in a specification

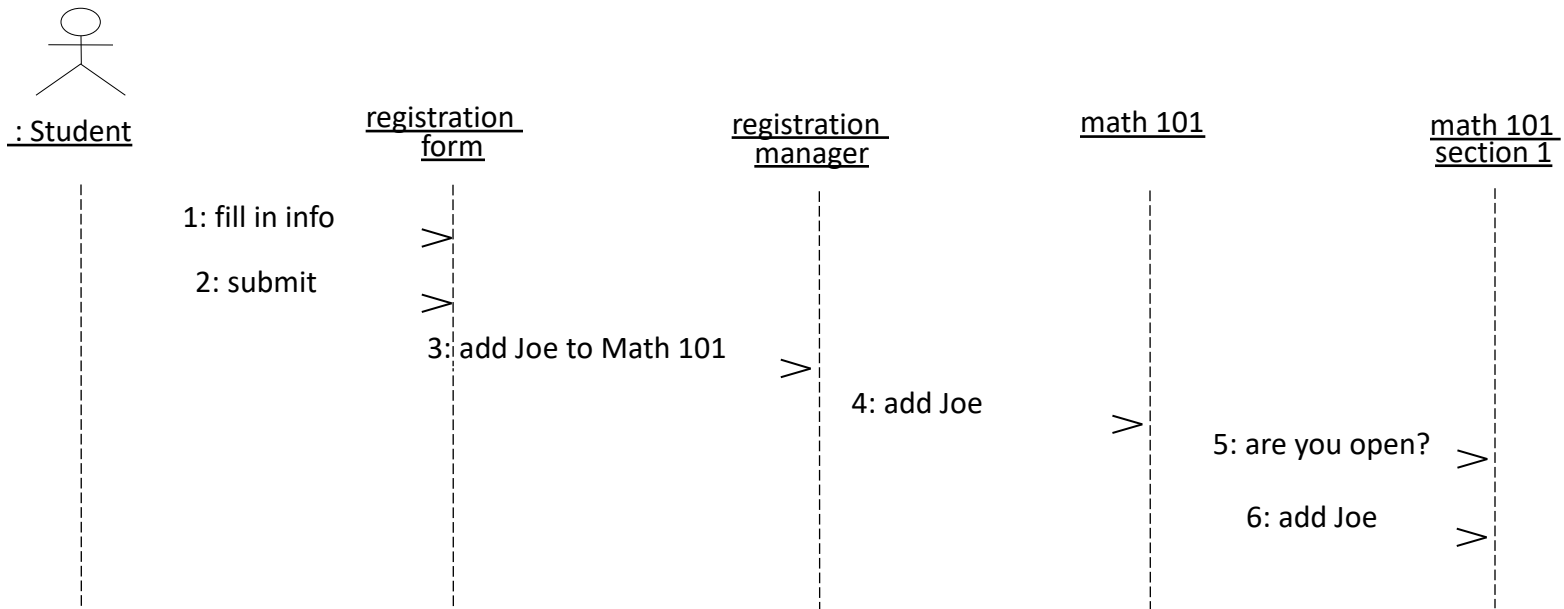
Why Build Models?

- To **understand** the problem better
- To **communicate** with other persons
- To **find errors** or omissions
- To **plan** out the design
- To **generate code**

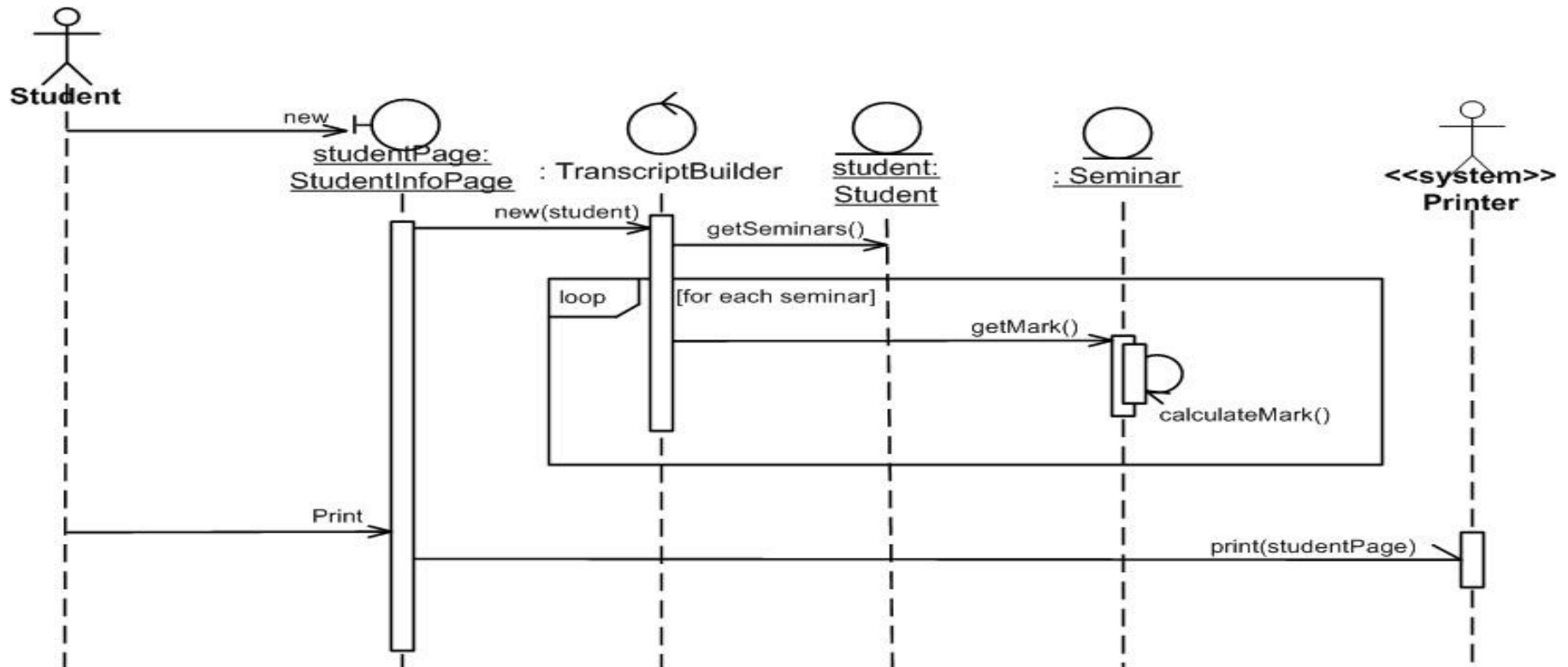


Sequence Diagram

- A sequence diagram shows step by step what must happen to accomplish a piece of functionality provided by the system.

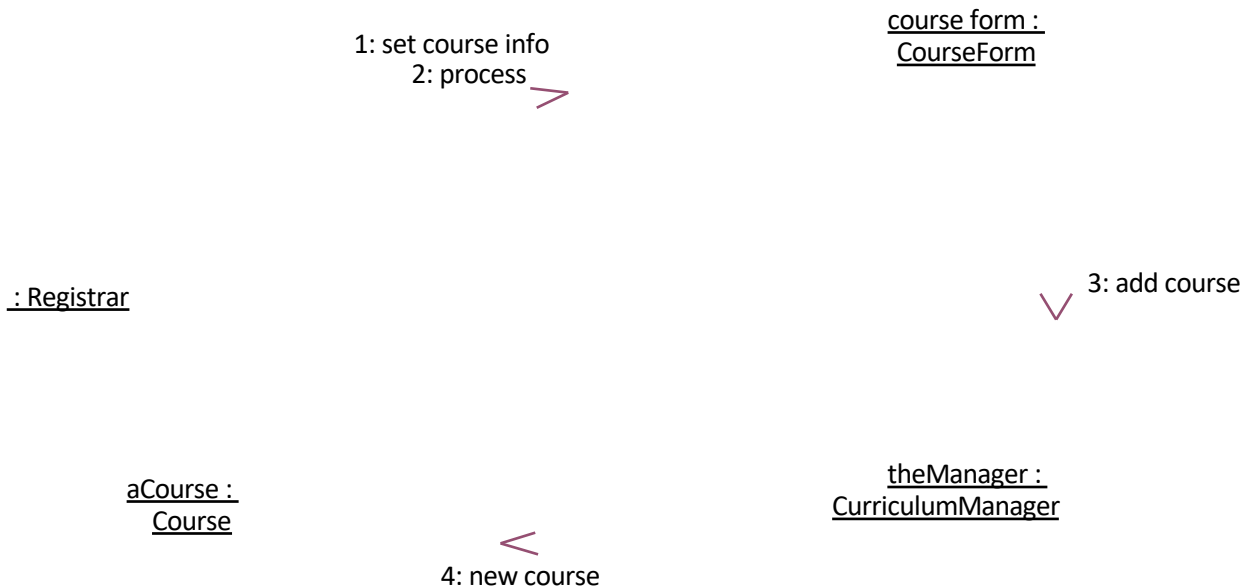


- **sequence diagram** : Interaction between objects;
emphasis on sequences

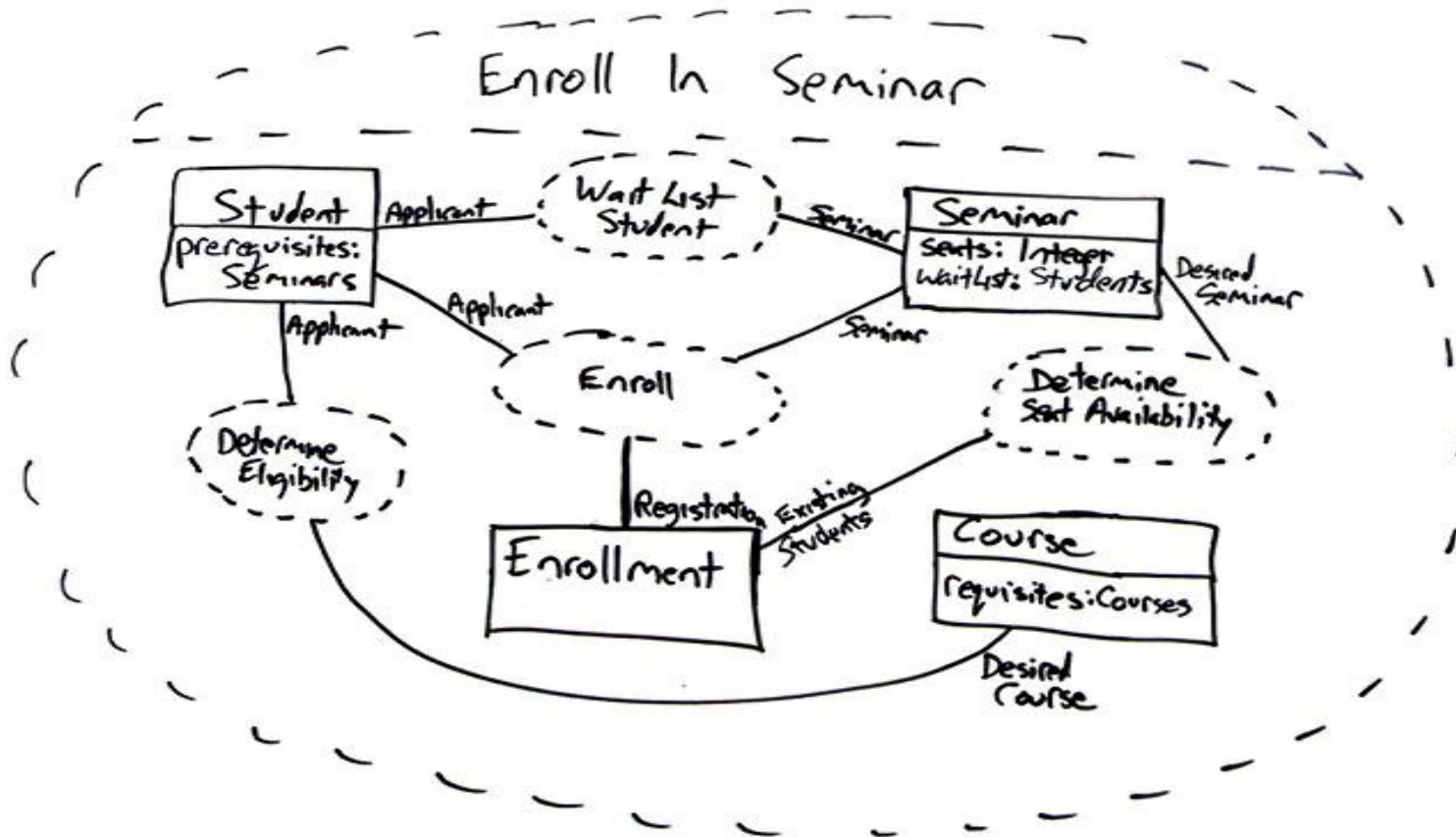


Collaboration Diagram

- A collaboration diagram displays object interactions organized around objects and their links to one another.

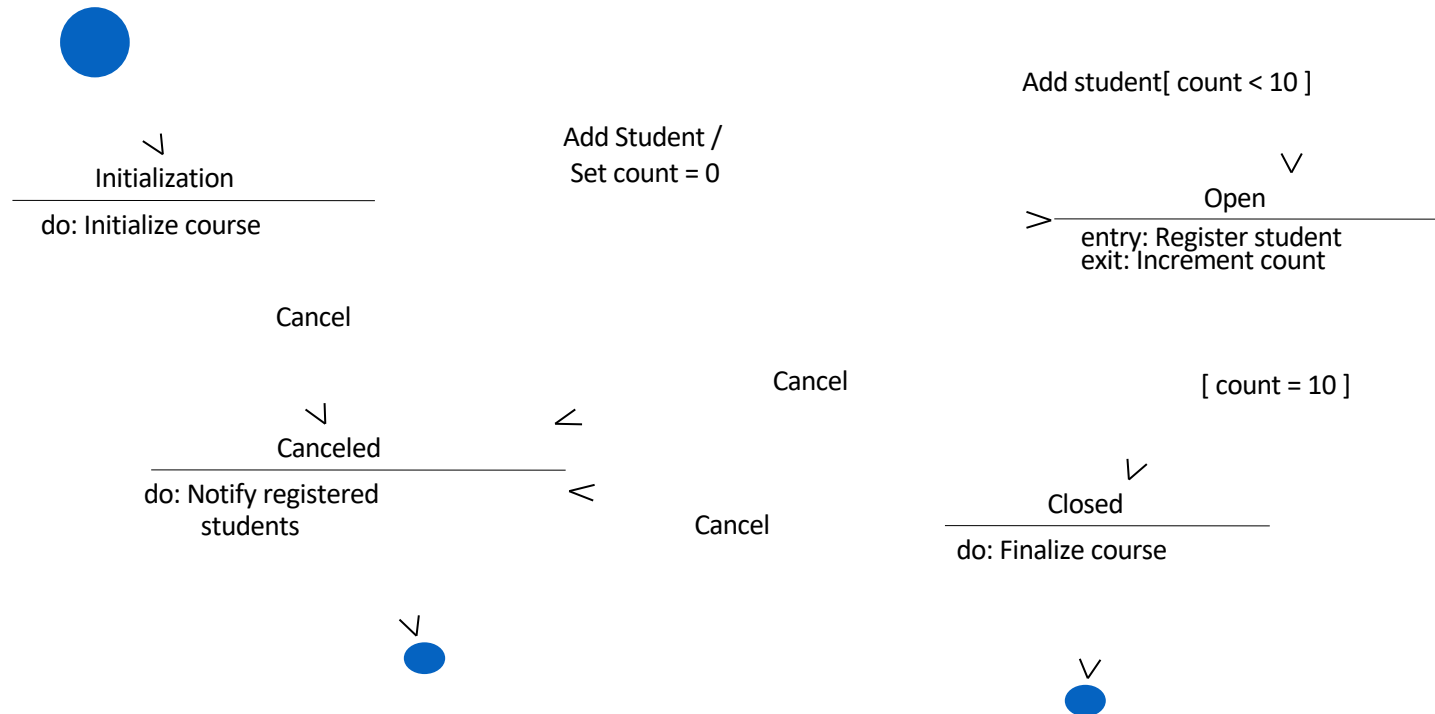


- collaboration diagram (协作图) : Interaction between objects;
emphasis on links

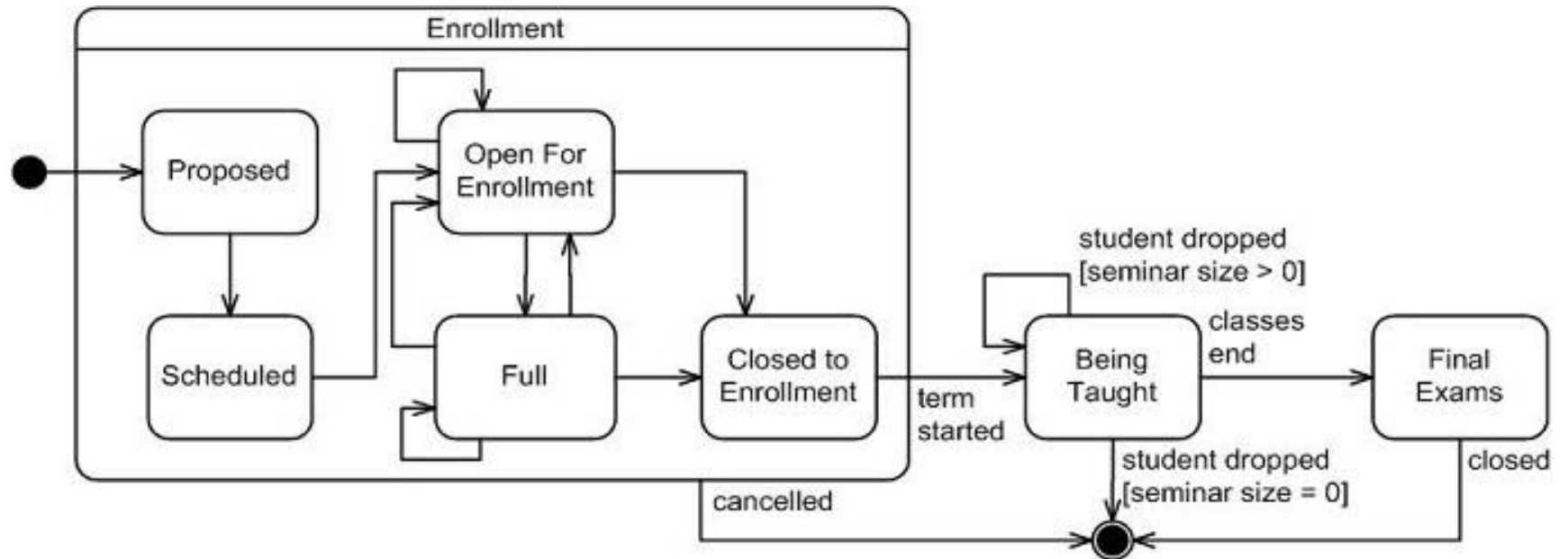


Statechart Diagram

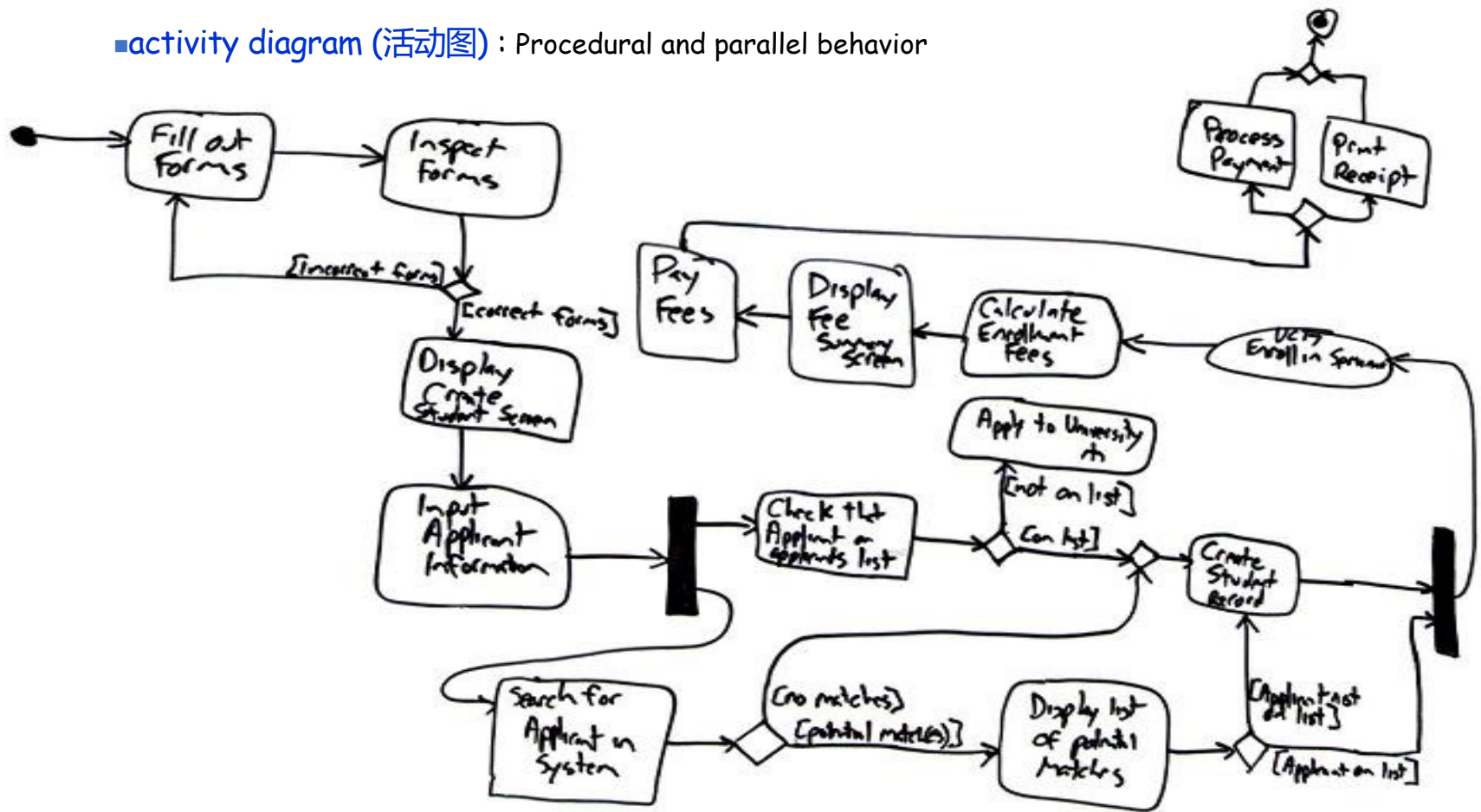
- A statechart diagram shows the lifecycle of a single class.



- statecharts diagram (状态图) : how events change an object over its life

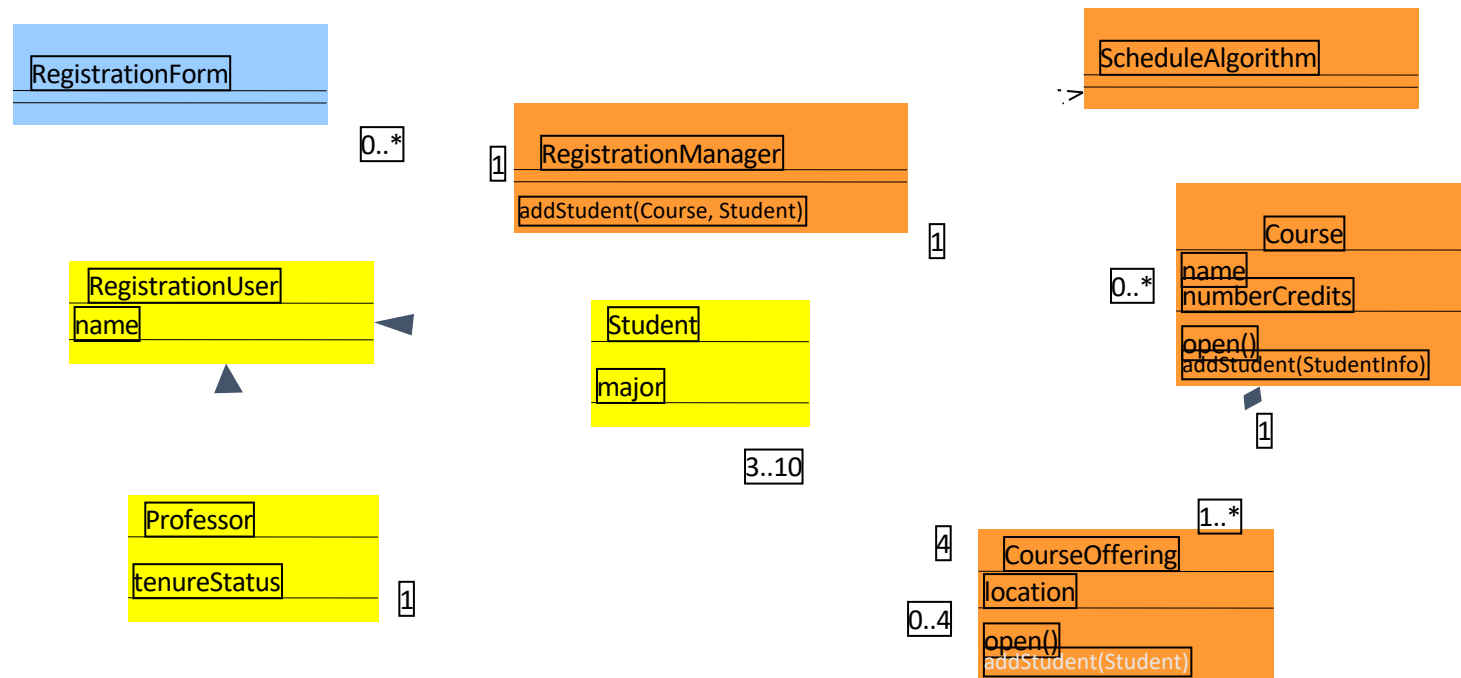


■ activity diagram (活动图) : Procedural and parallel behavior



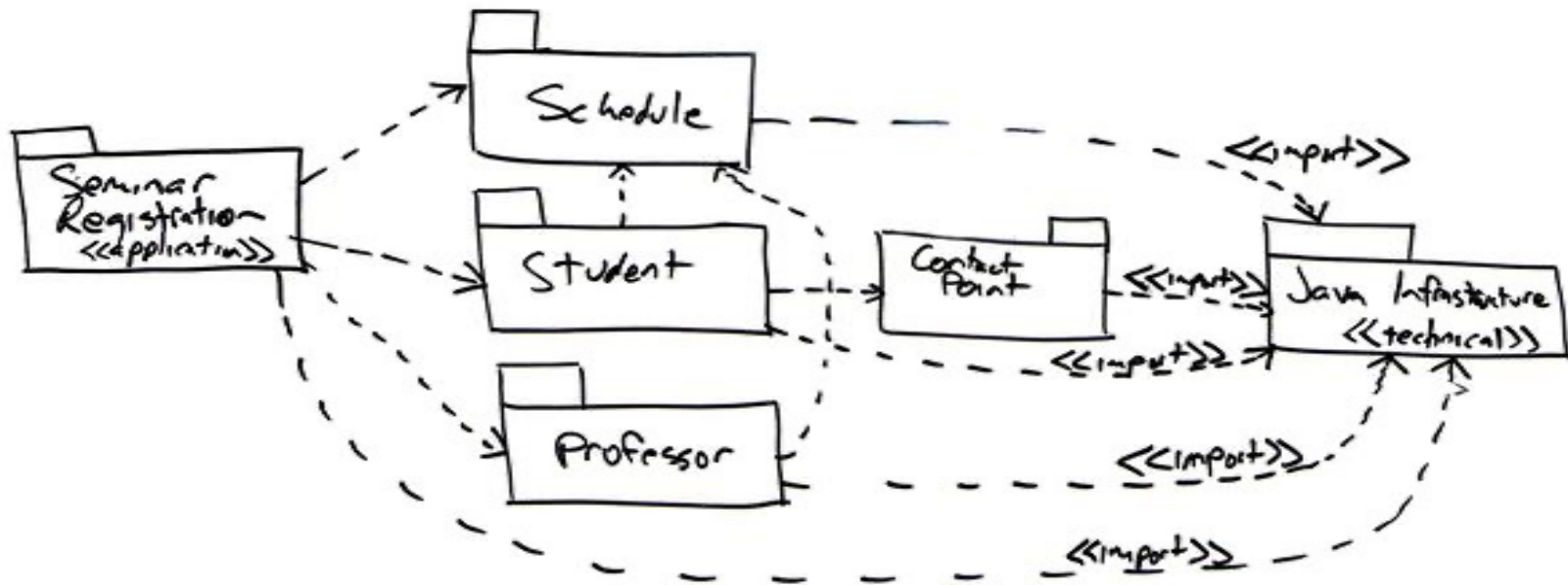
Class Diagram

- A class diagram shows the structure of your software.

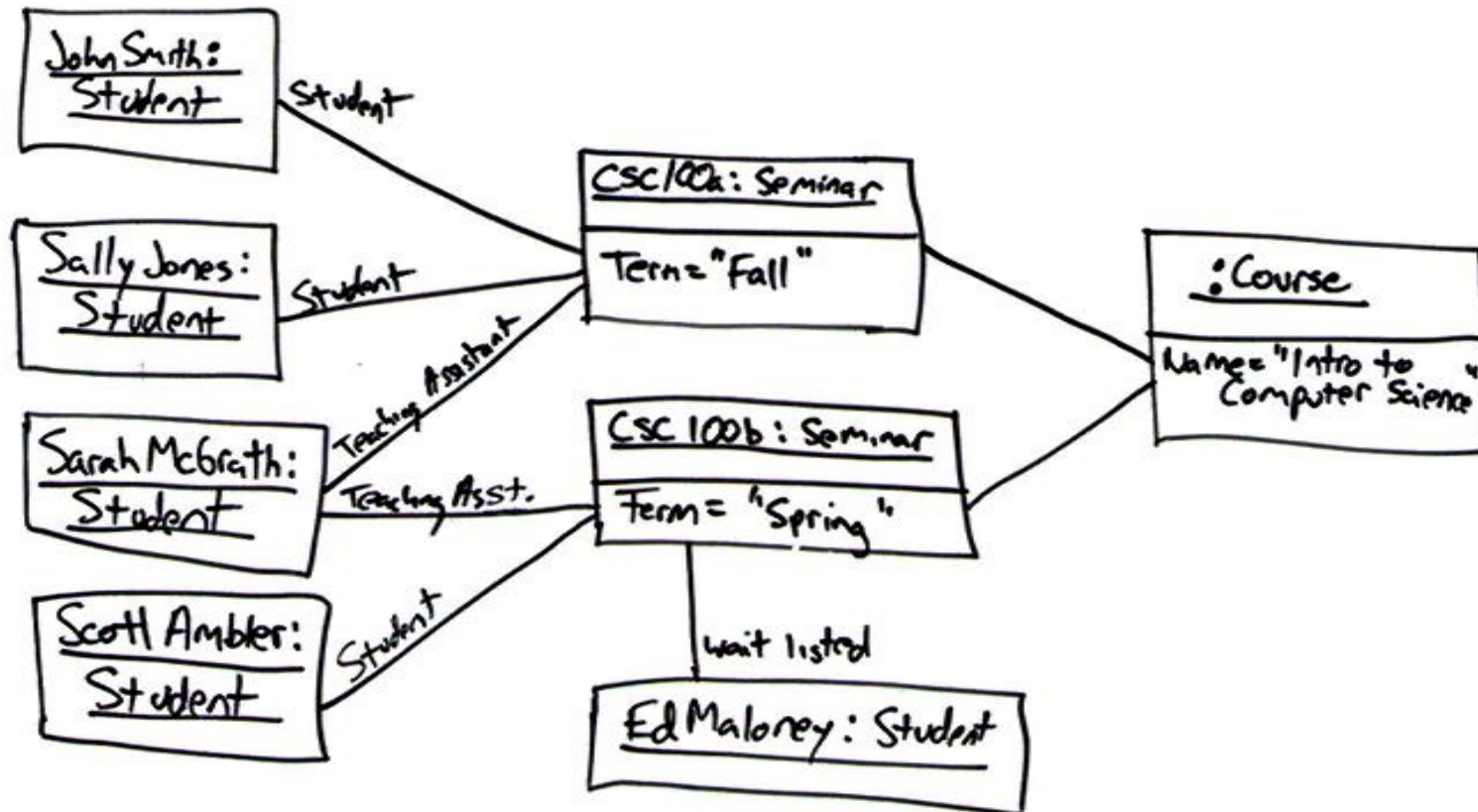


■ Package Diagram

Compile-time hierarchic structure

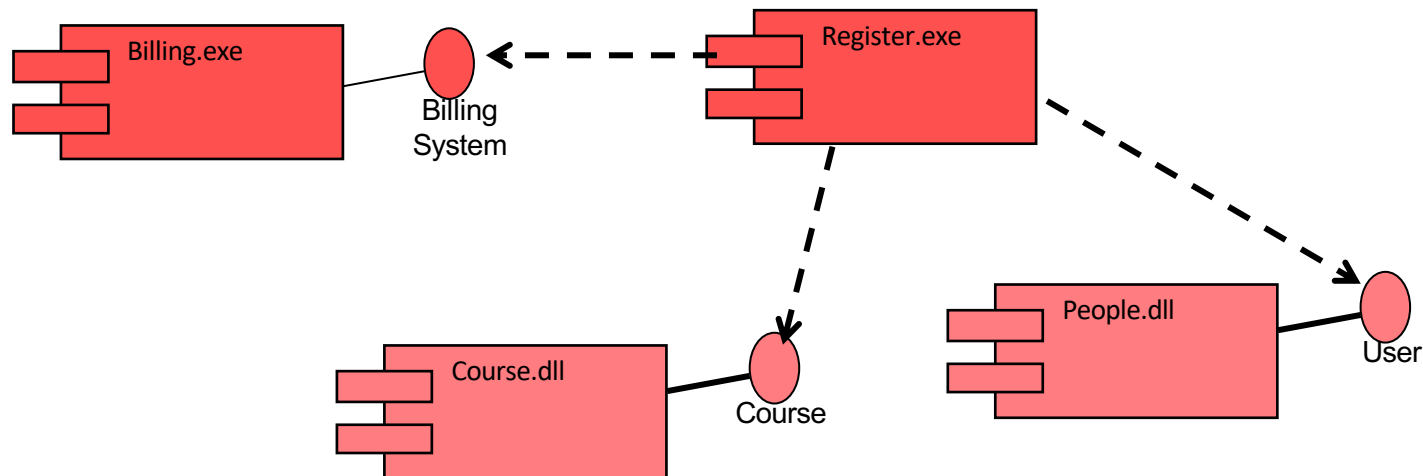


object diagram (对象图): Example configurations of instances

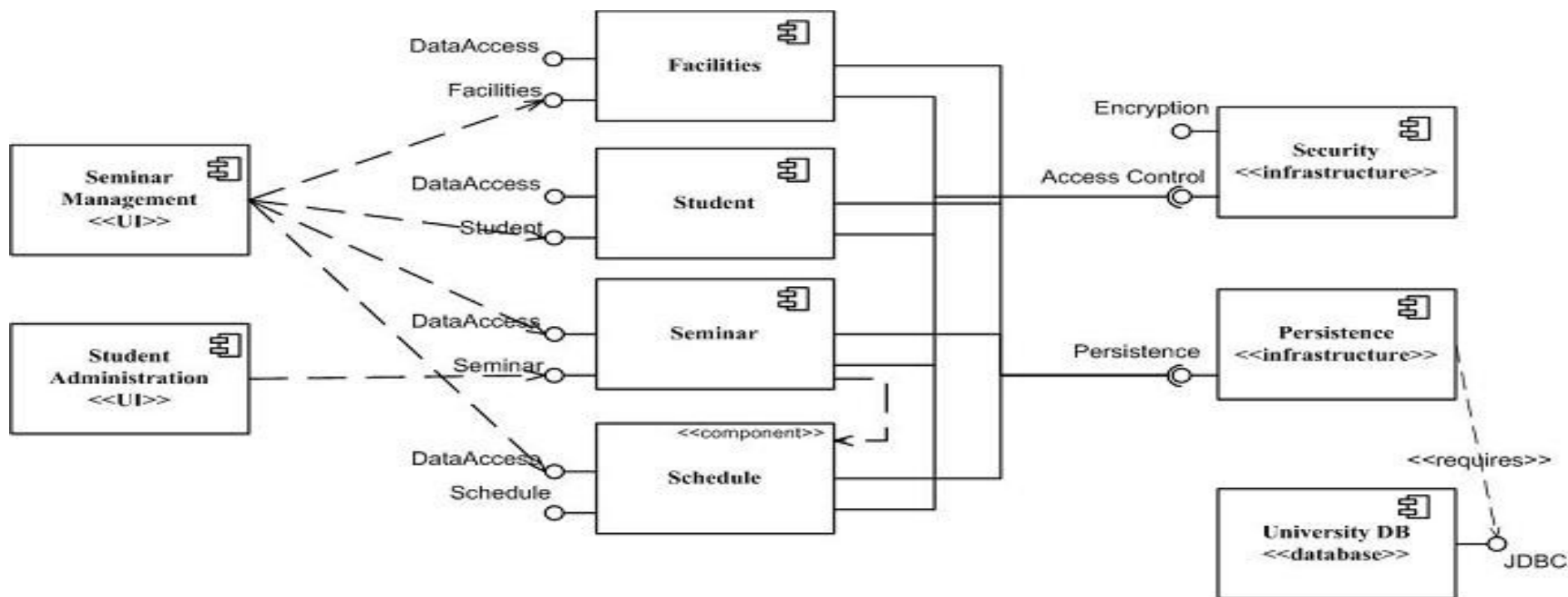


Component Diagram

- A component diagram illustrates the organization and dependencies among software components.

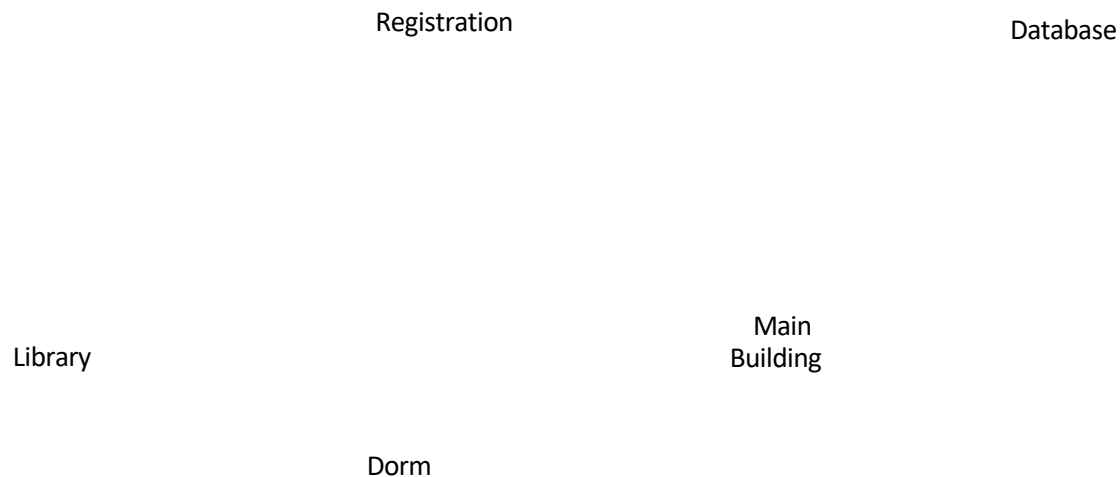


component diagram (构件图) : structure and connections of components

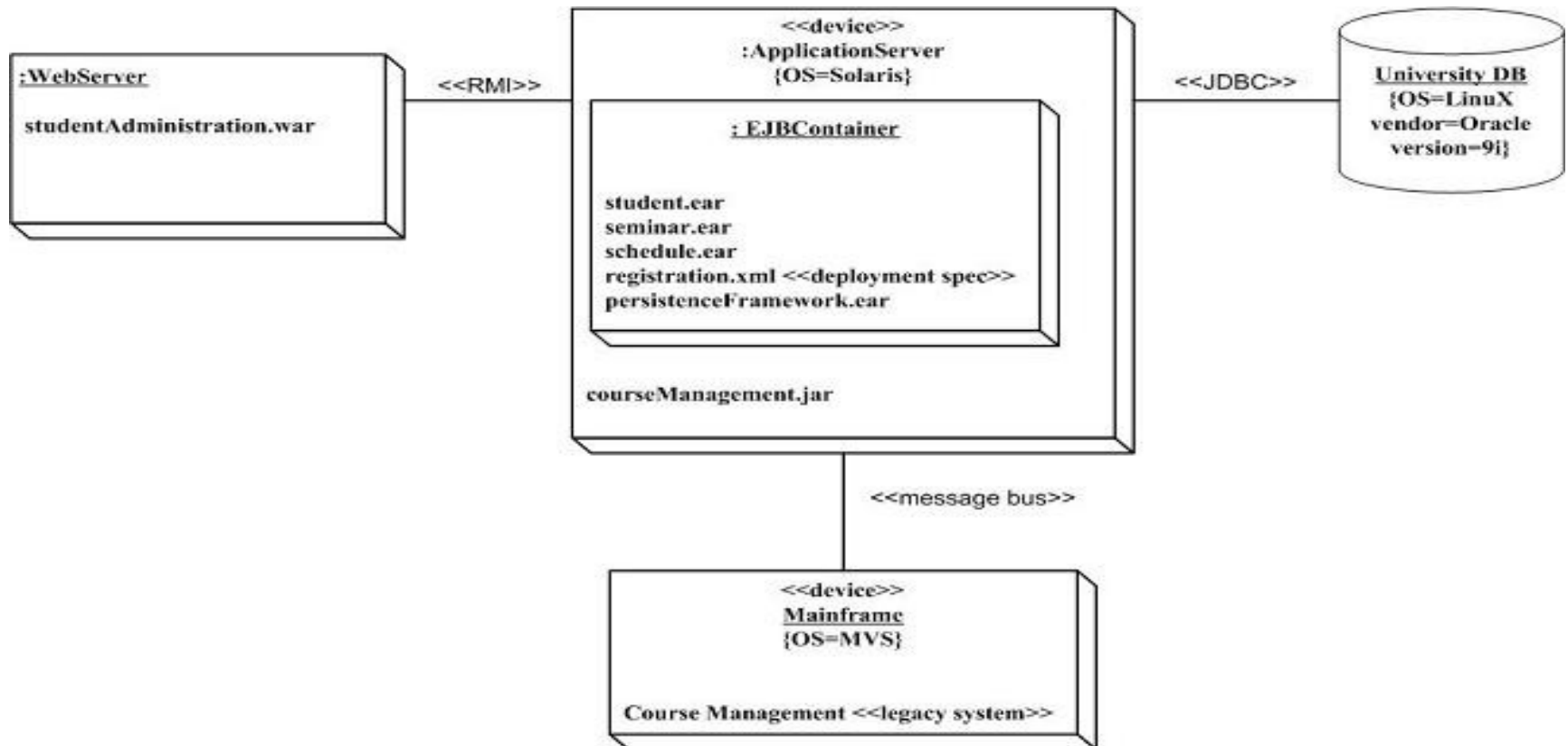


Deployment Diagram

- A deployment diagram visualizes the distribution of components across the enterprise.

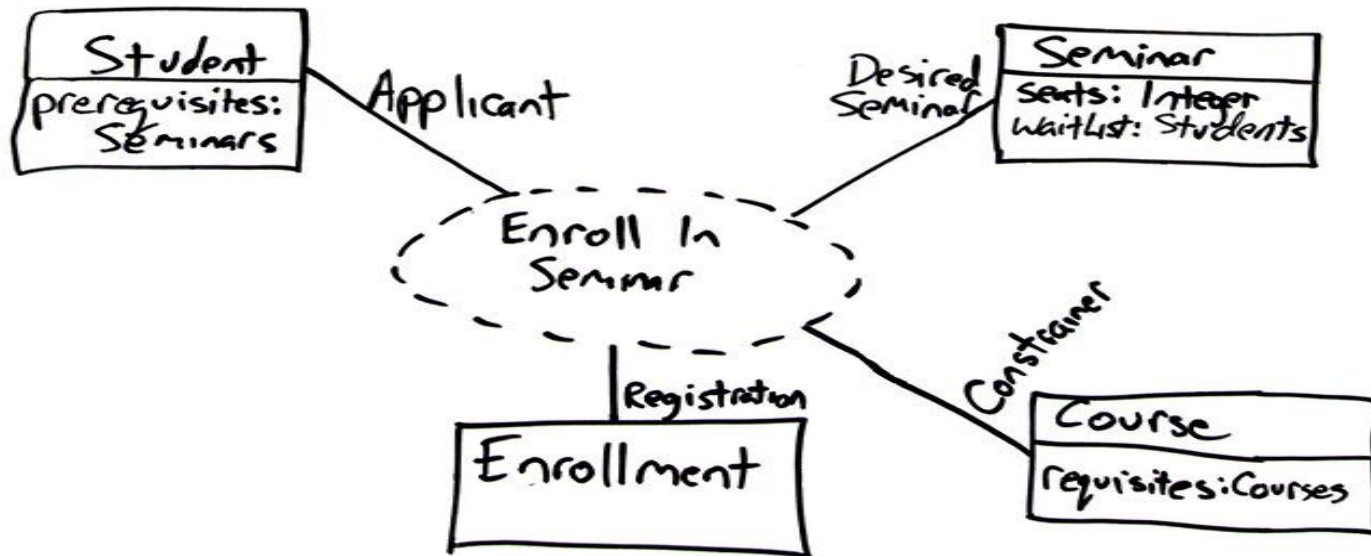


■ deployment diagram (部署图) : deployment of artifacts to nodes



■ Composite Structure Diagram (组成结构图)

Runtime decomposition of a class



Reference

- 清华大学国家级精品课程 《软件工程》 主讲人 刘强 副教授 刘璘 副教授
- https://www.icourses.cn/sCourse/course_3016.html
- https://www.xuetangx.com/course/THU08091000367/5883555?channel=learn_title



谢谢大家！

THANKS

