Picturing Programs

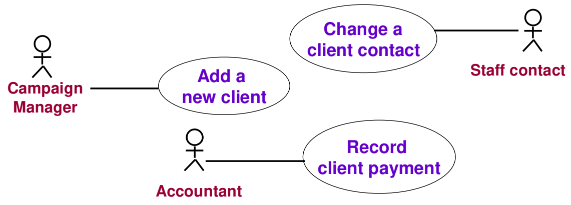
Jennifer Campbell
(slightly edited by Greg Wilson)
CSC301 – Fall 2007

Moving Towards Specifications

- **What functions will the new system provide?**
  - How will people interact with it?
  - Describe functions from a user's perspective
- **UML Use Cases**
  - Used to show:
    - the **functions** to be provided by the system
    - which **actors** will use which functions

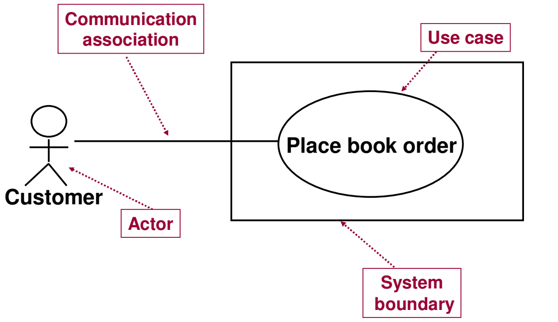
UML Use Case Diagrams

Capture the relationships between **actors** and **use cases.**



[BMF99]

Notation for Use Case Diagrams

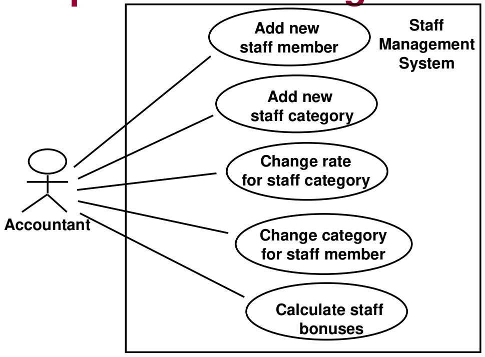


[BMF99]

Use cases and Actors

- **Use case**:
  - a pattern of behavior that the new system is required to exhibit
  - a sequence of related actions performed by an actor and the system via a dialogue.
- **Actor**:
  - anything that needs to interact with the system:
    - a person
    - a role that different people may play
    - another (external) system.

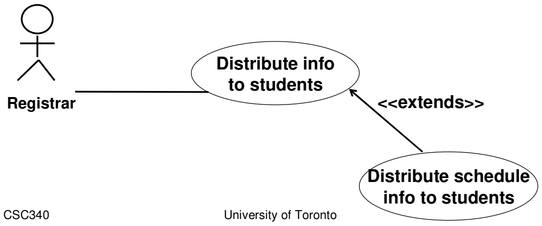
Example: Staff Management



<<extends>>

When one use case adds behaviour to a base case

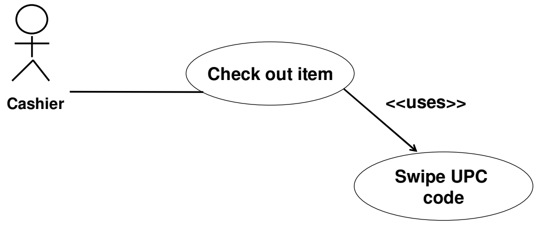
- used to model a part of a use case that the user may see as optional system behavior;
- also models a separate sub-case which is executed conditionally.



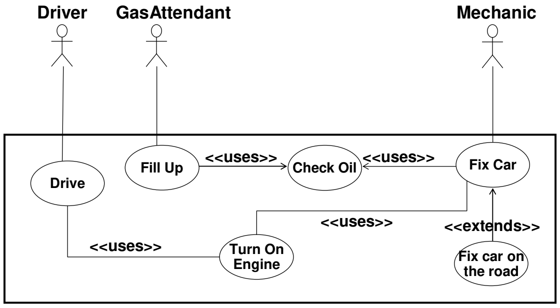
<<uses>>

One use case invokes another (like a procedure call);

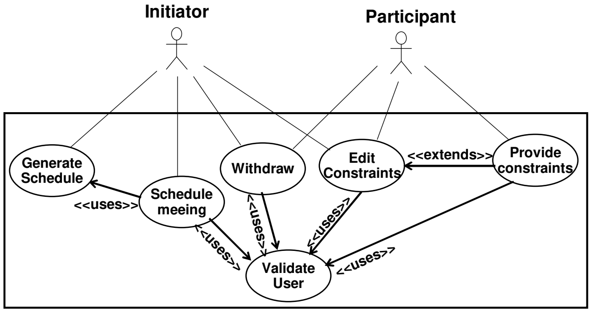
- used to avoid describing the same flow of events several times
- puts the common behavior in a use case of its own.



Example: Car



Example: Meeting Scheduler



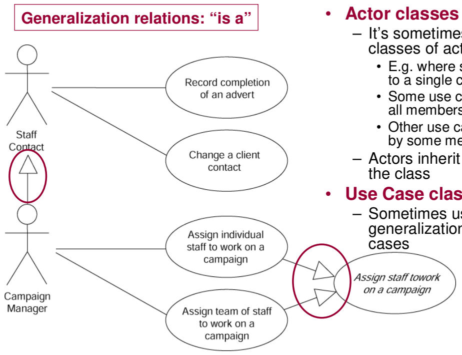
Identifying Actors

- **Look for:**
  - the users who directly use the system
  - also others who need services from the system
- **To find actors that are people/roles ask:**
  - Who will be a primary user of the system? (primary actor)
  - Who will need support from the system to do her daily tasks?
  - Who will maintain, administrate, keep the system working? (secondary actor)
  - Who or what has an interest in the results that the system produces ?
- **To find actors that are external systems ask:**
  - Which hardware devices does the system need?
  - With which other systems does the system need to interact with?

Finding Use Cases

- For each actor, ask the following questions:
  - Which functions does the actor require from the system?
  - What does the actor need to do ?
  - Does the actor need to read, create, destroy, modify, or store some kinds of information in the system ?
  - Does the actor have to be notified about events in the system?
  - Does the actor need to notify the system about something?
  - What do those events require in terms of system functionality?
  - Could the actor's daily work be simplified or made more efficient through new functions provided by the system?

Generalizations

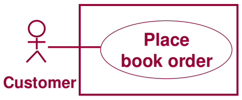

Generalization relations: “is a”
Staff Contact
Record completion of an advert
Change a client contact
Campaign Manager
Assign individual staff to work on a campaign
Assign team of staff to work on a campaign
Assign staff towork on a campaign

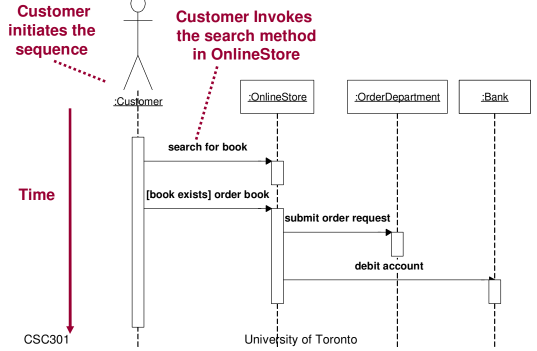
- **Actor classes**
  - It’s sometimes useful to identify classes of actor
    - E.g. where several actors belong to a single class
    - Some use cases are needed by all members in the class
    - Other use cases are only needed by some members of the class
  - Actors inherit use cases from the class
- **Use Case classes**
  - Sometimes useful to identify a generalization of several use cases

UML Sequence Diagrams

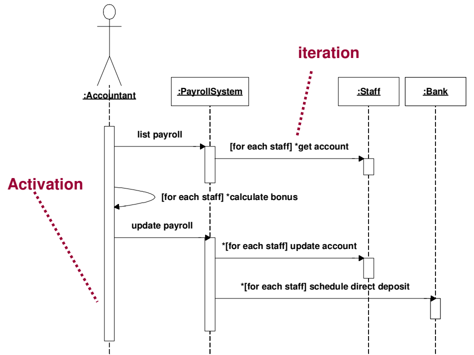
- Describe a Use Case using Sequence Diagrams
  - Sequence diagrams show step-by-step what’s involved in a use case
    - Which objects are relevant to the use case
    - How those objects participate in the function
  - You may need several sequence diagrams to describe a single use case.
    - Each sequence diagram describes one possible scenario for the use case
  - Sequence diagrams…
    - …should remain easy to read and understand.
    - …do not include complex control logic

Example: Place book order

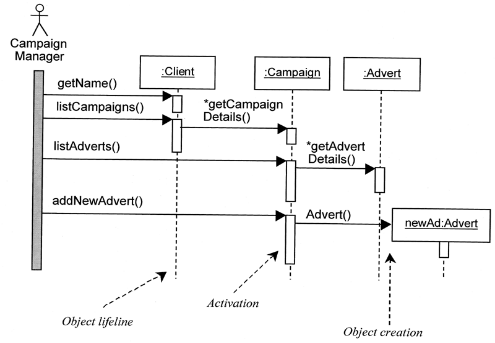




Example: Calculate staff bonuses


iteration
Activation
:Accountant
:PayrollSystem
:Staff
:Bank
list payroll
[for each staff] *get account
[for each staff] *calculate bonus
update payroll
*[for each staff] update account
*[for each staff] schedule direct deposit

Example: Add an advertisement



Modelling Sequences of Events

- Objects “own” information and behaviour
  - Objects don’t “know” about other objects’ information, but can ask for it.
  - To carry out business processes, objects have to collaborate.
    - …by sending messages to one another to invoke each others’ operations
  - Objects can only send messages to one another if they “know” each other
    - I.e. if there is an association between them.

UML Activity Diagrams: Legend

▭ **Activity**

● **Initial State**

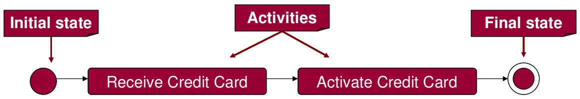
◉ **Final State**

▬ **Fork/Join**

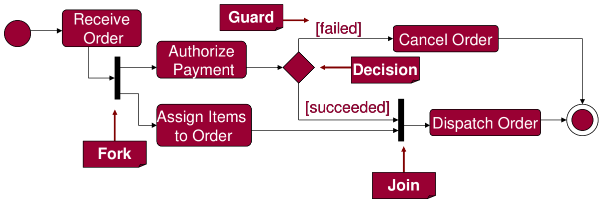
◇ **Decision**

Example 1: Credit Card Activation

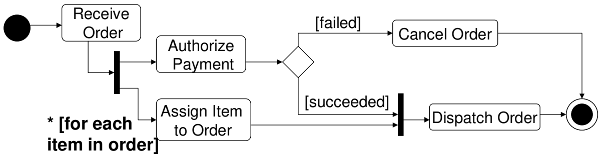
The customer receives the card, and then activates the card.



Example 2: Order System



Example 3: Order System (with loop)



A few style guidelines

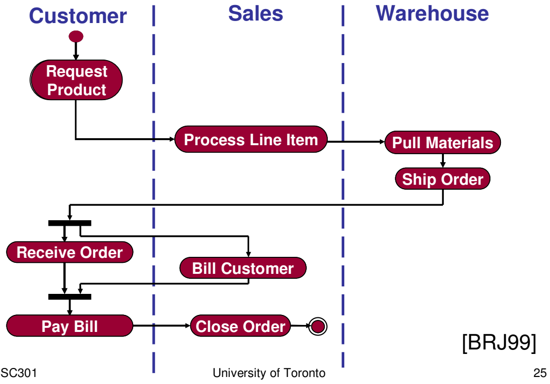
- The diagram should have start and end state(s).
- Diagrams are read from top-left to bottom-right
  - put the initial and final states in those locations
- Each activity should have at least one transition into it and at least one transition out of it.
- The diagram should be decidable
  - transitions out of a decision points should have mutually exclusive guards
  - the set of guards should be complete
- Each fork should have a corresponding join.

[Amb03]

Swimlanes

- Swimlanes can be used to group activities based on the actor (person, business unit, etc) who performs them.
- If an activity diagram is partitioned into swimlanes, than each activity must appear in exactly one swimlane.
- Transitions may cross swimlanes.

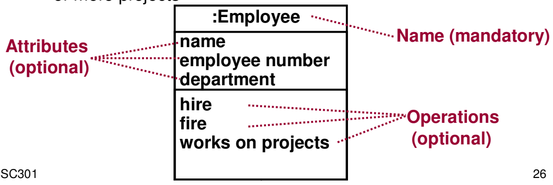
Example 3: Diagram with Swimlanes



[BRJ99]

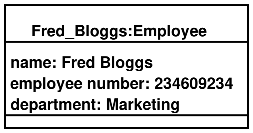
Classes

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



Objects vs. Classes

- The instances of a class are called objects.



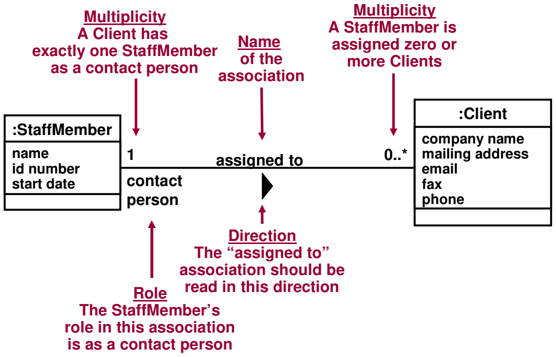
  - Two different objects may have identical attribute values (like two people with identical name and address)
- Objects have **associations** with other objects
  - E.g. **Fred_Bloggs:employee** is associated with the **KillerApp:project** object
  - But we will capture these relationships at the class 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??)

UML Class Diagrams

- UML Class Diagrams show classes and their relationships
- Relationships: connections between classes
  - Objects do not exist in isolation from one another
  - Types of UML relationships:
    - **Association**
    - **Aggregation and Composition**
    - **Generalization**
    - Dependency
    - Realization

} not useful for analysis

Class Associations



Multiplicity
A Client has exactly one StaffMember as a contact person

Name
of the association

Multiplicity
A StaffMember is assigned zero or more Clients

Direction
The "assigned to" association should be read in this direction

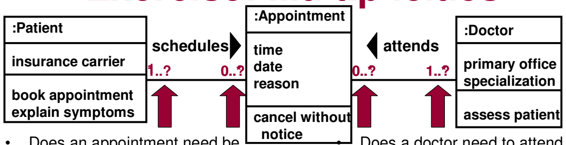
Role
The StaffMember's role in this association is as a contact person

Association Multiplicity

- **Multiplicity:** the minimum and maximum times an object can be associated with the related object
- **Examples:**

– Optional (0 or 1)	0..1
– Exactly one	1 = 1..1
– Zero or more	0..*
– One or more	1..*
– A range of values	1..6
– A set of ranges	1..3,7..10,15,19..*

Exercise: Multiplicities



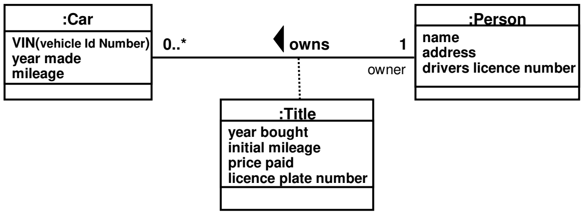
- Does an appointment need be scheduled by a patient?
  - Yes! An appointment is scheduled by at least one patient.
- Can the same appointment be schedule by multiple patients?
  - If not, then exactly one Patient schedules each appointment.
- Does a patient have to schedule an appointment?
  - No.
- Can a patient schedule more than one appointment?
  - If yes, then the multiplicity is 0..*
- Does a doctor need to attend appointments?
  - No.
- Can a doctor attend multiple appointments?
  - Yes.
- Does an appointment need to be attended by at least one doctor?
  - Yes.
- Can an appointment be attended by multiple doctors?
  - If yes, then the multiplicity is 1..*.

[DWT05]

Association Classes

Sometimes the association is itself a class

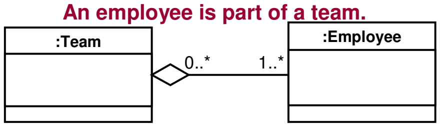
- …because we need to retain information about the association
- …and that information doesn't naturally live in the classes at the ends of the association
  - E.g. a "title" is an object that represents information about the relationship between an owner and her car



Aggregation

The “is part of” or “whole-part” relationship

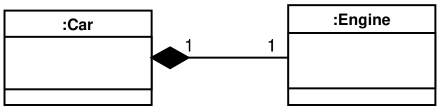
An employee is part of a team.



[Amb03]

Composition

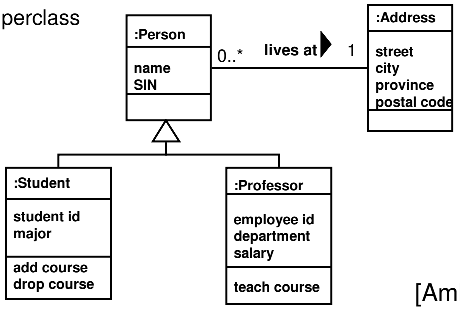
- Strong form of aggregation that implies ownership:
  - if the whole is removed from the model, so is the part
  - the whole is responsible for the disposition of its parts



Note: The multiplicity should be 1 when the association is compostion (i.e., a car should always have (at least) one engine).

Generalization

Subclasses inherit attributes, associations, & operations from the superclass



[Amb03]

More on Generalization

- **Usefulness**
  - Can easily add new subclasses if the organization changes
- **Look for generalizations in two ways:**
  - **Top Down**
    - Subdivide an existing class
    - 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.”*
  - **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”*

More on Generalization [2]

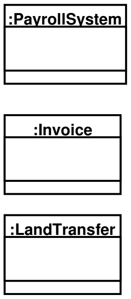
- **Don’t generalize just for the sake of it**
  - Be sure that everything about the superclass applies to the subclasses
  - Be sure that the superclass is useful as a class in its own right
  - Don’t add subclasses or superclasses that are not relevant to your analysis

Finding Classes

- Look for nouns and noun phrases in stakeholders’ descriptions of the problem
  - include in the model if they explain the nature or structure of information in the application.
- It’s better to include many candidate classes at first
  - You can always eliminate them later if they turn out not to be useful
  - Explicitly deciding to discard classes is better than just not thinking about them

Possible Classes

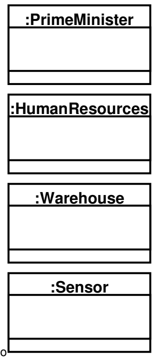
- **External Entities**
  - …that interact with the system being modeled
    - E.g. people, devices, other systems
- **Things**
  - …that are part of the domain being modeled
    - E.g. reports, displays, signals, etc.
- **Occurrences or Events**
  - …that occur in the context of the system
    - E.g. transfer of resources, a control action, etc.



[Pre97]

Possible Classes [2]

- **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.



[Pre97]

Selecting Classes

- Discard classes for concepts which:
  - Are beyond the scope of the analysis
  - Refer to the system as a whole
  - Duplicate other classes
  - Are too vague or too specific
    - e.g. have too many or too few instances
- Include external entities as classes if they:
  - Produce or consume information essential to the system

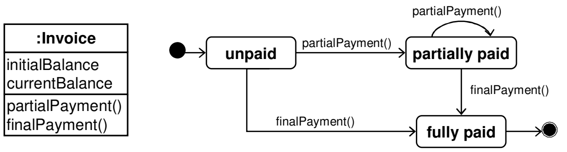
Coad & Yourdon's Criteria for Selecting Classes

- **Retained information**: Will the system need to remember information about this class of objects?
- **Needed Services**: Do objects in this class have identifiable operations that change the values of their attributes?
- **Multiple Attributes**: Does the class have multiple attributes?
- **Common Attributes**: Does the class have attributes that are shared with all instances of its objects?
- **Common Operations**: Does the class have operations that are shared with all instances of its objects?

[Pre97]

Object Behaviour

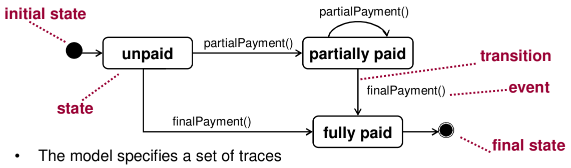
- All objects have "state"
  - An object has a value for each of its attributes
  - Each possible assignment of values to attributes is a **state**
    - (and non-existence is a state, although we normally ignore it)
- Example: Invoice object:



[Mac01]

Statecharts

- There are a finite number of states (all attributes have finite ranges)



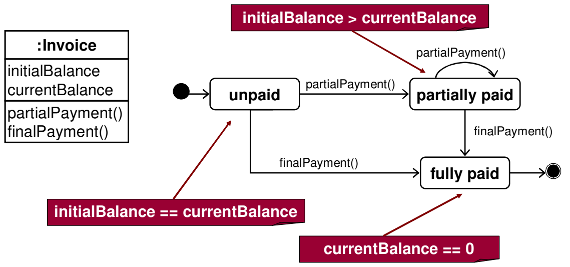
- The model specifies a set of traces
  - E.g. partialPayment();partialPayment;finalPayment()
  - E.g. partialPayment();finalPayment()
  - There may be an infinite number of traces (and traces may be of infinite length)
- The model excludes some behaviours
  - E.g. no trace may have a finalPayment() followed by a partialPayment()

Abstraction

- The **state space** of most objects is enormous
  - State space size is the product of the range of each attribute
    - E.g. object with five boolean attributes: 2^5+1 states
    - E.g. object with five integer attributes: $(\text{maxint})^5+1$ states
    - E.g. object with five real-valued attributes: …?
  - If we ignore computer representation limits, the state space is infinite
- Only part of that state space is "interesting"
  - Some states are not reachable
  - Integer and real values usually only vary within some relevant range
  - Often not interested in the actual values, just certain ranges:
    - **Example for Age**: age<18, 18≤age≤65, age>65
    - **Example for Cost:** cost ≤ budget, cost==0, cost > budget, cost > (budget+10%)

Exercise: Abstraction

For each state, what are the properties or value ranges of initialBalance and currentBalance that interest us?

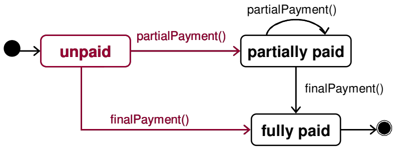


[Mac01]

Transitions

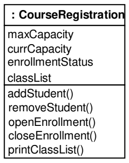
A transition consists of three parts:
event [guard] / action

- A **transition** is the movement from one state to another.
- States are either "on" or "off" at a given point in time.
- If a state is on, then transitions from it are enabled.
- Transitions are triggered by **events**.



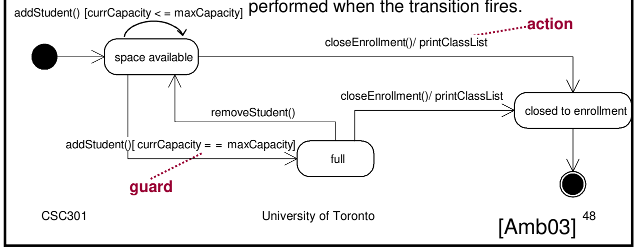
[SJB05]

Transitions [2]



event [guard] / action

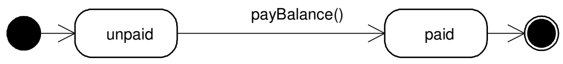
- A **guard** is a Boolean condition that must be true in order for a transition to "fire".
  - When an event occurs the guard conditions are checked, and if they are met, then the transition fires.
- The **action** is a procedural expression to be performed when the transition fires.


addStudent() [currCapacity < = maxCapacity]
action
closeEnrollment()/ printClassList
space available
removeStudent()
closeEnrollment()/ printClassList
closed to enrollment
addStudent()[currCapacity = = maxCapacity]
full
guard

[Amb03]

Events

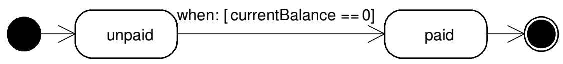
- *Call events* occur when an object receives a call for one of its operations to be performed
  - Example: Bill class



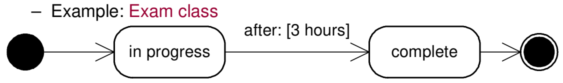
- *Signal events* occur when an object receives an explicit (real-time) signal
  - Example signal: Mouse click
  - More useful in design
  - Syntax is the same as call events

Events [2]

- *Change events* occur when a condition becomes true
  - denoted by the keyword 'when'
  - Example: Invoice class



- *Time events* mark the passage of a designated period of time
  - Example: Exam class



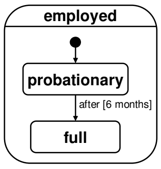
Superstates

States can be nested, to make diagrams simpler

- A superstate consists of one or more states.
- Superstates make it possible to view a state diagram at different levels of abstraction.

OR superstates

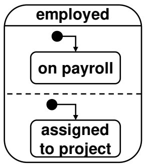
- when the superstate is "on", only one of its substates is "on"



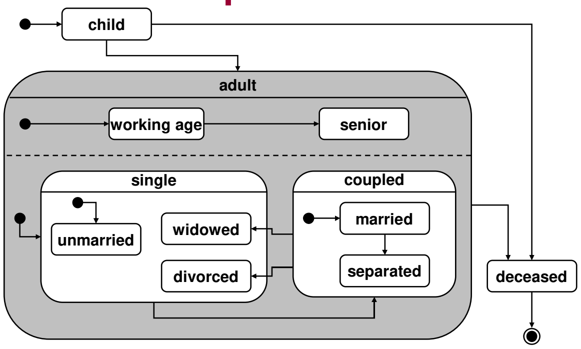
AND superstates

(concurrent substates)

- When the superstate is "on", all of its states are also "on"
- Usually, the AND substates will be nested further as OR superstates



Example: Person



Class Diagrams and Statecharts

- **Consistency Checks between diagrams**
  - Each statechart should correspond to one class on the Class diagram.
  - All **events** in a statechart should appear as:
    - operations (methods) of an appropriate class in the class diagram
  - All **actions** in a statechart should appear as:
    - operations (methods) of an appropriate class in the class diagram

Style Tips

- The diagram should have start and end state(s).
- Diagrams are usually read from top-left to bottom-right, so put the start and end states in those locations.
- Each state should have at least one transition into it and at least one transition out of it.
- The diagram should be deterministic.
- Use a superstate when multiple states have a common entry or exit condition.
- It is fine for guards on transitions from a state to not form a complete set.

[Amb03]

Entity Relationship (ER) Schema

- Comparable to **UML class diagrams**
  - Not equivalent
- Good for describing data requirements for a new information system.
- Direct, easy-to-understand graphical notation
- Translates readily to relational schema for database design
  - more abstract than relational schema
  - e.g. can represent an entity without knowing its properties

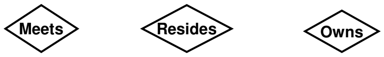
Entities

- Classes of objects with properties in common and an autonomous existence
  - E.g. City, Department, Employee, and Sale
- An instance of an entity is an object in the class represented by the entity
  - E.g. Stockholm, Helsinki, are examples of instances of the entity City
- Usually described using nouns.

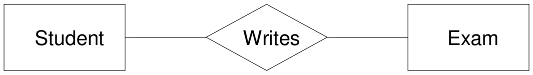
City **Department** **Employee** **Sale**

Relationships

- Logical links between two or more entities.
  - E.g. Resides is a relationship that can exist between a City entity and a Person entity
- An instance of a relationship is an n-tuple of instances of entities
  - E.g. the pair (Johanssen,Stockholm), is an instance in the relationship Resides.
- Usually described using verbs (sometimes nouns)



Examples

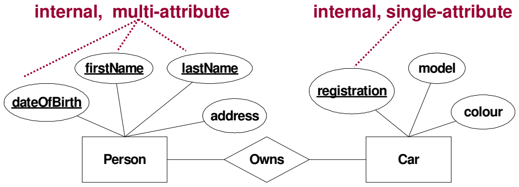


Attributes

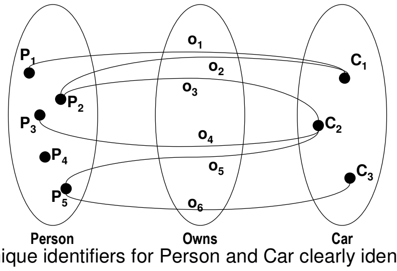
- Associate a value belong to a set (domain) with each instance of an entity or relationship.

Internal Identifiers

- Identifiers are also known as **keys**.
- An identifier may be formed by one or more attributes of the entity itself
- A relationship is identified using identifiers for all the entities it relates
  - E.g. the identifier for the relationship Person-Owns-Car is a tuple of the Person and Car identifiers

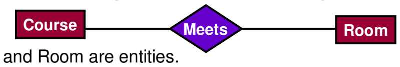


Example: Instances for Owns

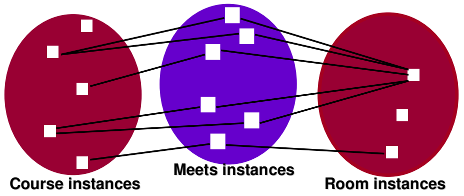


- The unique identifiers for Person and Car clearly identify each entity.
- Instances of the Owns relationship are tuples of (Person, Car).

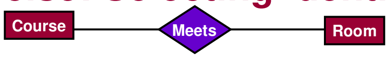
Meaning of ER Diagrams



- Course and Room are entities.
  - Their instances are particular courses (eg CSC340S) and rooms (eg BA1130)
- Meets is a relationship.
  - Its instances describe particular meetings.
  - Each meeting has exactly one associated course and room



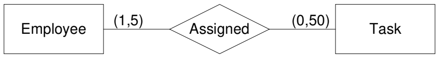
Exercise: Selecting Identifiers



- What attributes should we use to describe Meets?
  - <coursename,day,hour>
    - Only one section of a course can meet at a time (day and hour)
  - <coursename>
    - Only one meeting per given course name
  - <courseinstructor,room>
    - Only one meeting for a given instructor and room
      - An instructor must have all her meetings in different rooms!
  - <courseinstructor>
    - An instructor participates in at most one meeting

Cardinalities

- **Cardinalities constrain participation in relationships**
  - minimum and maximum number of relationship instances in which an entity instance can participate.
  - E.g.

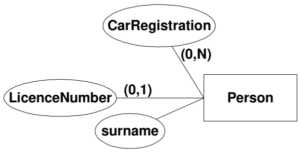


Employee is assigned 1 to 5 tasks. Tasks are assigned to 0 to 50 employees.

- Cardinality is **any pair of non-negative integers** (min,max), where min <= max
  - (1,1) - One-to-One
  - (1,N) - One-to-Many
  - (N,1) - Many-to-One
  - (M,N) - Many-to-Many

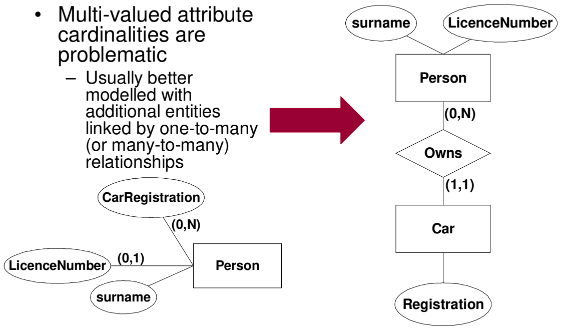
Cardinalities of Attributes

- Attributes can also have cardinalities
  - To describe the minimum and maximum number of values of the attribute associated with each instance of an entity or a relationship.
  - The default is (1,1)
  - Optional attributes have cardinality (0,1)

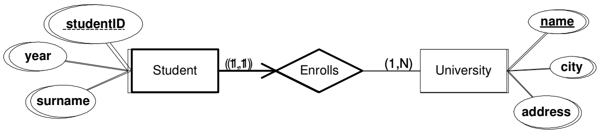


Cardinalities of Attributes [2]

- Multi-valued attribute cardinalities are problematic
  - Usually better modelled with additional entities linked by one-to-many (or many-to-many) relationships



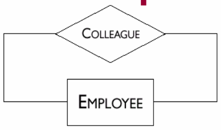
Weak Entities



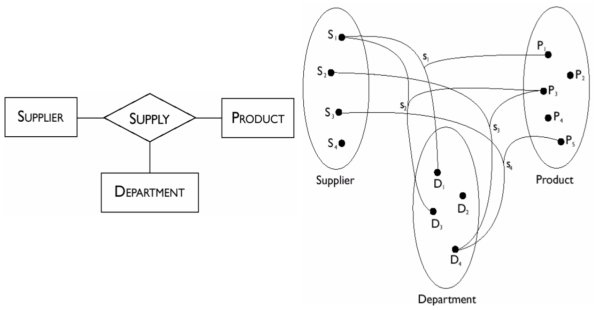
- Student is a weak entity.
  - Cannot be uniquely identified using only the studentID.
  - Need to know which university she is enrolled in.
- A weak entity can only be identified using the attributes of another entity.
  - It must be in a (1,1) relation with the relationship used to uniquely identify it.
- Also called an external identifier

Recursive Relationships

- An entity can have relationships with itself…

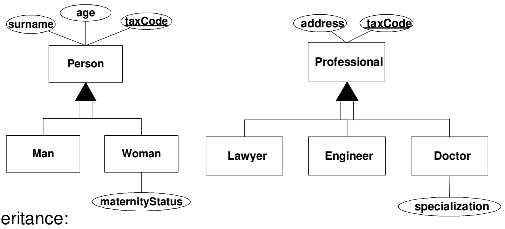


Ternary Relationships



Generalizations

- Show “**is-a**” relationships between entities



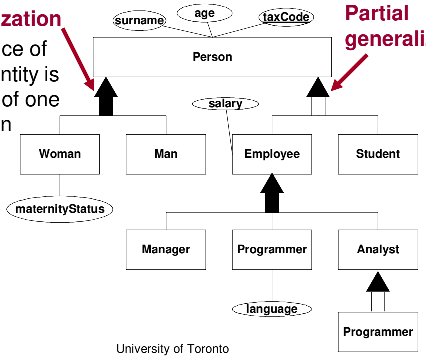
- Inheritance:
  - Every instance of a child entity is also an instance of the parent entity
  - Every property of the parent entity (attribute, identifier, relationship or other generalization) is also a property of a child entity

Types of Generalizations

Total generalization

- every instance of the parent entity is an instance of one of its children

Partial generalization



In My Opinion

- Models are only as useful as the tools used to work with them
- The act of modeling is more useful than the models themselves
- *But:* the more concurrency there is in a system, the more important modeling and proofs become