COMP 3700: Software Modeling and Design

(Interaction Design)

Agenda

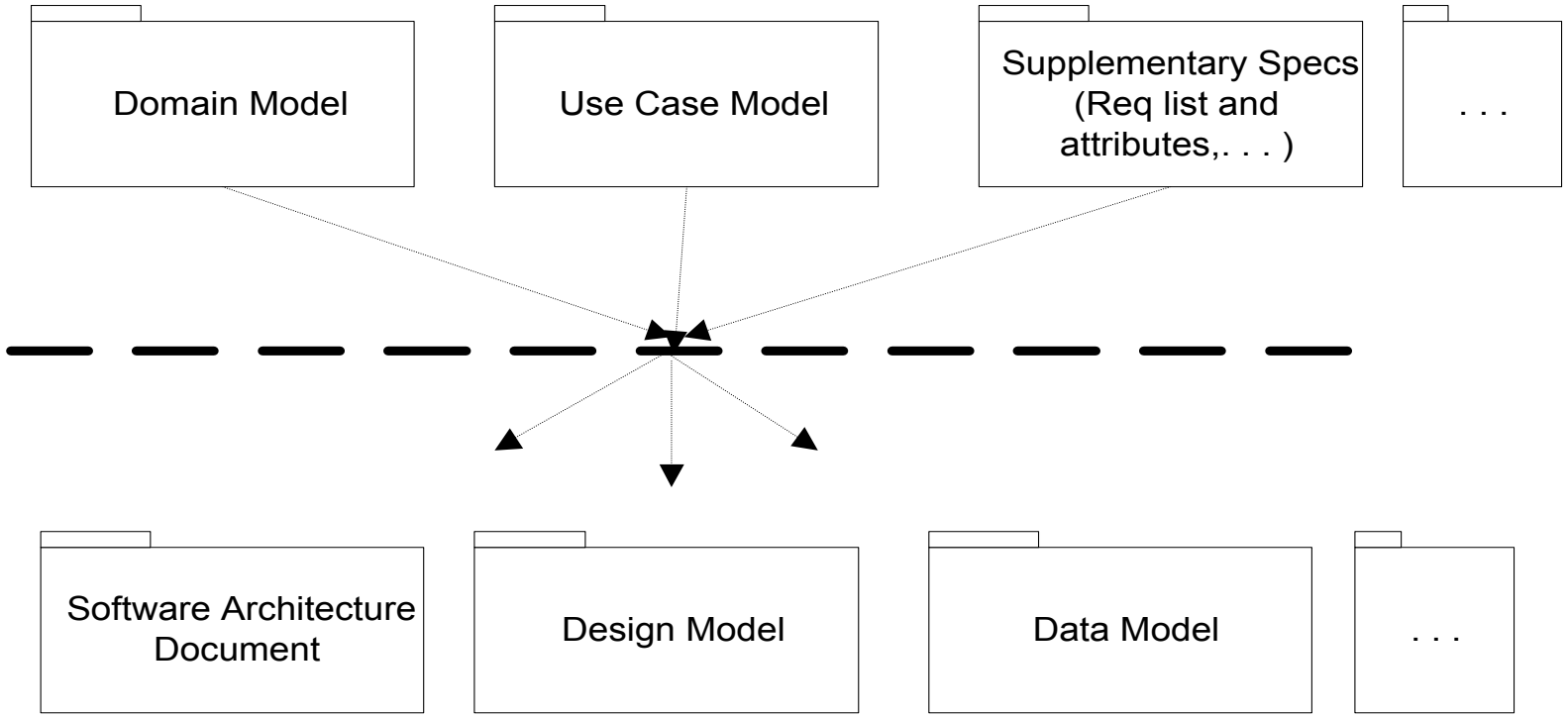
- Transition to Design
- Interaction Diagrams
- Designing Objects with Responsibilities
- GRASP Patterns

Transition to Design

- During requirements and analysis work, we
  - "Do the right thing"
  - Understand the domain
  - Clarify and record the constraints and requirements
  - Essentially ignore thinking about the design, and focus on understanding the problem.
  - . . .
- During design, we
  - "Do the thing right"
  - Create a software (and hardware) solution that meets the wishes of the stakeholders.

From Requirements to Design

- **A set of requirements-oriented artifacts (and thought) inspire design-oriented artifacts.**



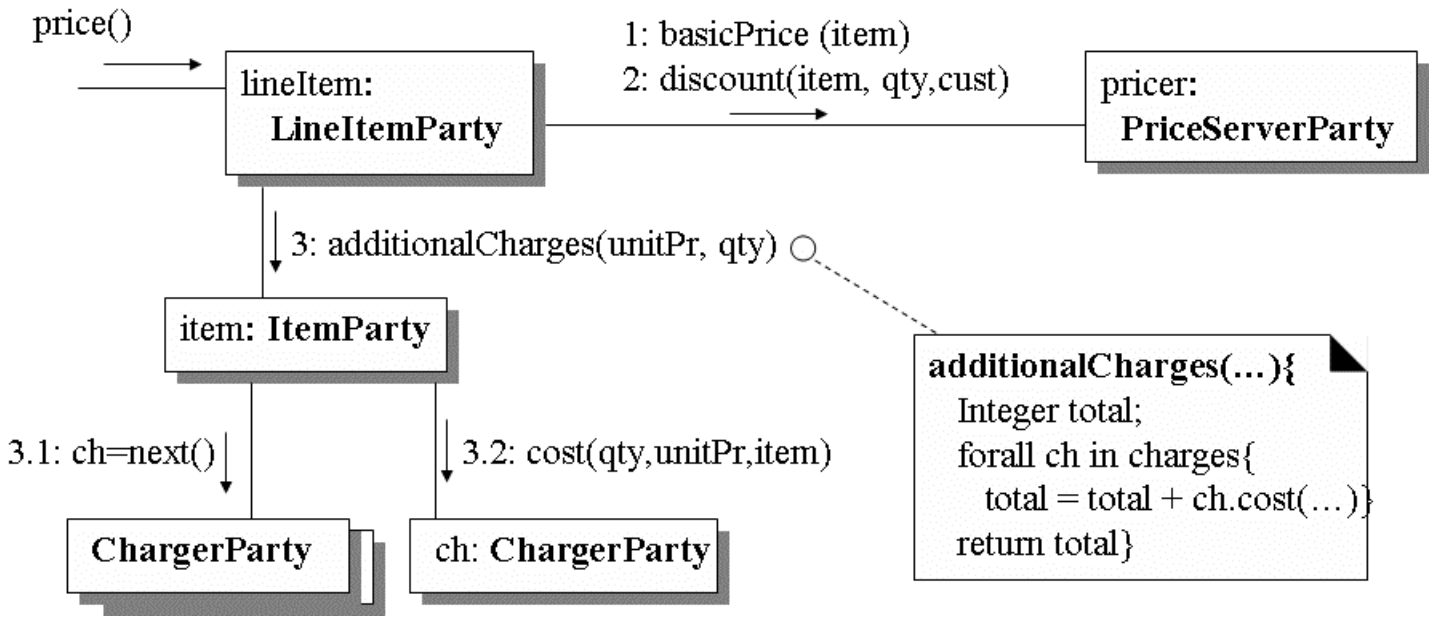
Changing Hats

- Until this time, we have been wearing an "investigator" hat, essentially ignoring what the solution should be.
  - "Do the right thing"
- Now, we take off the investigator hat, and put on our "designer" hats.
  - "Do the thing right"
- Of course, in the context of iterative development, we do this repeatedly.

Sequence and Collaboration Diagrams

- Collaboration diagrams show object interactions in a graph or network format.
- Sequence diagrams illustrate interactions such that object lifetimes are emphasized (in a kind of fence format).

```
price() {
  basicPr = pricer.basicPrice(item);
  discount = pricer.discount(item, qty, cust);
  unitPr = basicPr - (discount * basicPr);
  quotePr = uniPr + item.additionalCharges(unitPr, qty);
  return quotePr;}
```



Basic Collaboration Diagram Notation

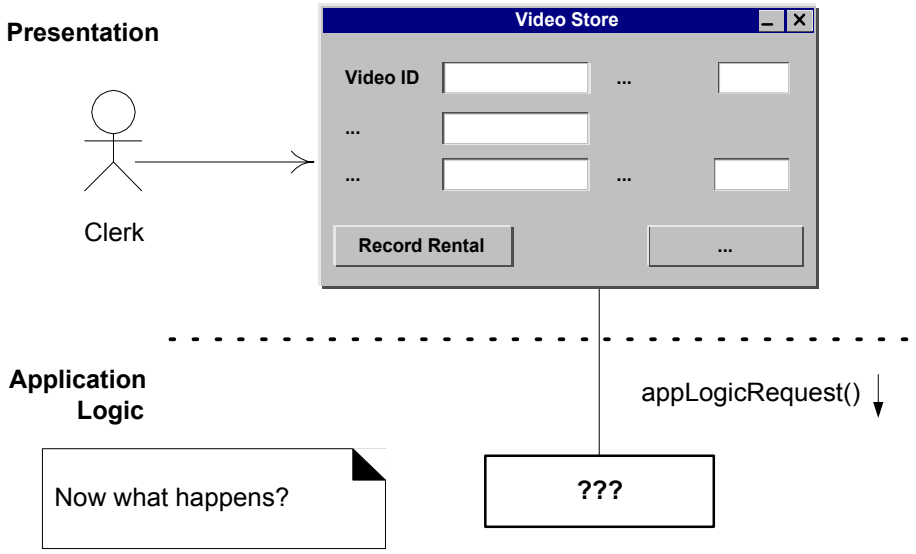
- Links
- Messages
- Message Number Sequencing
- Conditional Messages
- Iteration or Looping

Basic Sequence Diagram Notation

- Links
- Messages
  - Illustrating Returns
  - Creation of instances
- Object Lifelines and Object Destruction
- Conditional Messages
- Iteration

Assigning Responsibilities to Objects

- Now what happens?
- What object should receive this message?
- What objects should interact to fulfill the request, and how?



Responsibility-Driven Design (RDD)

- Detailed object design is usually done from the point of view of the *metaphor* of:
  - Objects have responsibilities
  - Objects collaborate
  - Similar to how we conceive of people
- In RDD we do object design such that we will ask questions such as:
  - What are the responsibilities of this object?
  - Who does it collaborate with?

Responsibilities

- Responsibilities are an abstraction.
  - The responsibility for persistence.
    - Large-grained responsibility.
  - The responsibility for the sales tax calculation.
    - More fine-grained responsibility.
- They are implemented with methods in objects.

Fundamental Principles of Object Design—GRASP Patterns

- What guiding principles help us assign responsibilities?
- These principles are captured in the GRASP patterns.
  - General Responsibility Assignment Software Patterns.
  - Very very fundamental, simple, basic principles of object design.

The 9 GRASP Patterns

1. Expert
2. Creator
3. Controller
4. Low Coupling
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Don't Talk to Strangers

Understanding and applying these is the most important, useful objective of this lecture.

(Information) Expert

- What is most basic, general principle of responsibility assignment?
- Assign a responsibility to the object that has the information necessary to fulfill it.
  - "That which has the information, does the work."
  - E.g., What software object calculates sales tax?

1. What information is needed to do this?
2. What object or objects has the majority of this information.

Creator

- What object creates an X?
  - Ignores special-case patterns such as *Factory*.
- Choose an object C, such that:
  - C contains or aggregates X
  - C closely uses X
  - C has the initializing data for X
- The more, the better.

Low Coupling

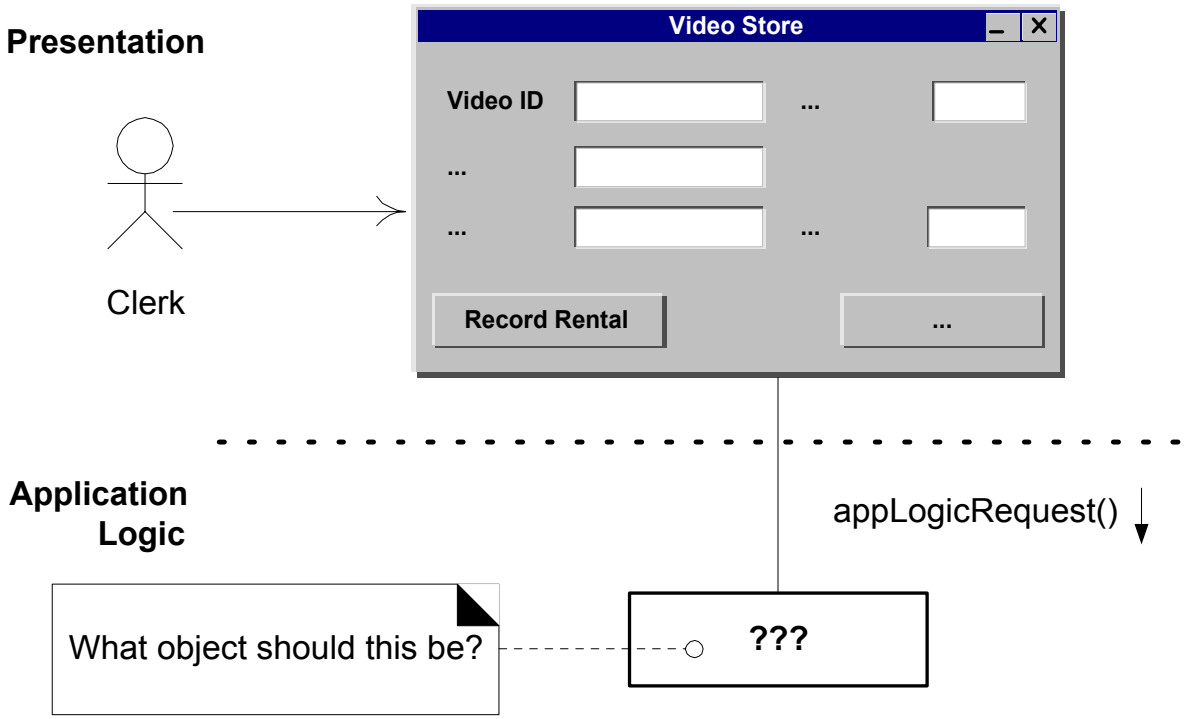
- Assign responsibilities so that coupling remains low.
- What does low coupling mean?
  - Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.
- A class with high coupling suffer from the following problems:
  - Changes in related classes force local changes.
  - Harder to understand in isolation.
  - Harder to reuse.

High Cohesion

- Assign responsibilities so that cohesion remains high?
- What does high cohesion mean?
  - Cohesion is a measure of how strongly related and focused the responsibilities of an element are. An element with highly related responsibilities, and which does not do tremendous amount of work, has high cohesion.
- A class with low cohesion is
  - hard to comprehend, reuse, and maintain
  - Delicate; constantly effected by change

Controller

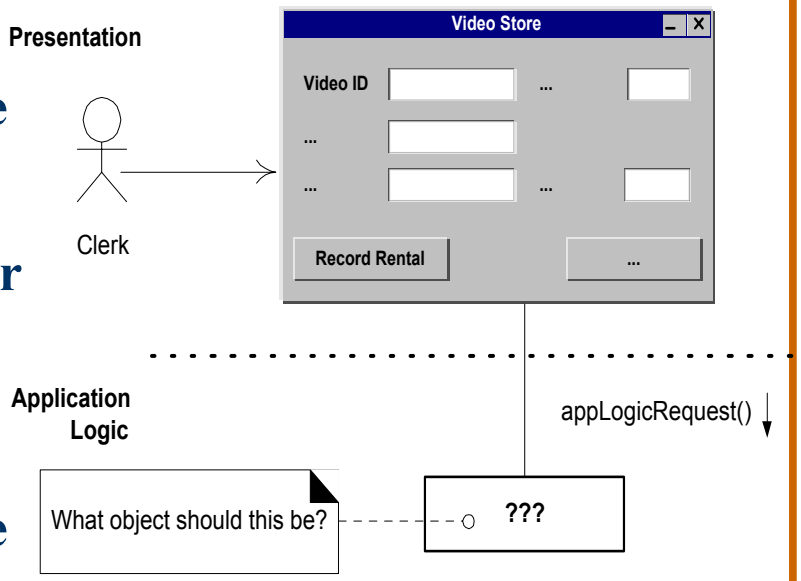
- What object in the domain (or application coordination layer) receives requests for work from the UI layer?



Controller

- **Candidates:**
  - **An object whose name reflects the use case.**
    - **e.g., RentingVideosUCHandler**

    part of the Application Coordination layer.
  - **An object whose name reflects the overall server, business, or large-scale entity.**
    - **A kind of "façade" object.**
    - **e.g., Store, RentingServer**



Polymorphism

- How to design for varying, similar cases?
- Assign a polymorphic operation to the family of classes for which the cases vary.
  - Don't use case logic.
- E.g., draw()
  - Square, Circle, Triangle

Pure Fabrication

- Where to assign a responsibility, when the usual options based on Expert lead to problems with coupling and cohesion, or are otherwise undersirable.
- Make up an "artificial" class, whose name is not necessarily inspired by the domain vocabulary.
- E.g., database persistence in
  - Video?
  - DatabaseFacade? (a Pure Fabrication)

Indirection

- A common mechanism to reduce coupling.

  Assign a responsibility to an intermediate object to decouple collaboration from 2 other objects.

Don't Talk to Strangers

- How to design to reduce coupling to knowledge of the structural connections of objects?
- Don't traverse a network of object connections in order to invoke an operation.
- Rather, promote that operation to a "familiar" of the client.

Class Visibility

- Attribute visibility from class A to B exists when B is an attribute of A.
- Parameter visibility from A to B exists when B is passed as a parameter to a method of B.
- Local visibility from A to B exists when B is declared as a local object within a method of A.
- Global visibility from A to B exists when B is global to A.

Design Class Diagram

- When to create Design Class Diagrams (DCDs):
  - In our presentation, DCDs follow the creation of interaction diagrams
  - Yet, in practice they can be created in parallel, as long as they are in synch.
  - It is possible and desirable to do a little interaction diagramming, then create DCDs, then extend the interaction diagrams some more, and so on.

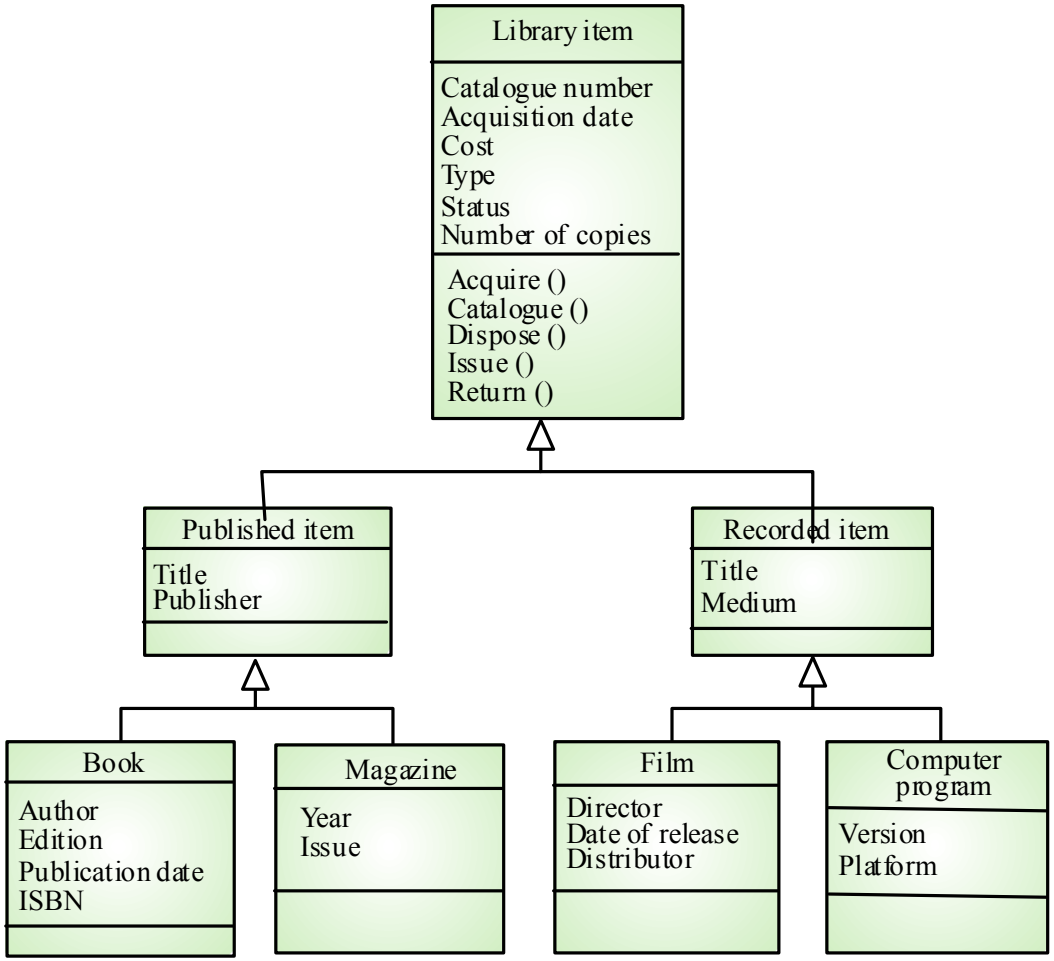
Design Class Diagram

- A DCD illustrates the specifications for software classes and interfaces in an application.
  - Classes, associations, attributes
  - Interfaces with their operations and constraints (OCL)
  - Methods
  - Attribute type information
  - Navigability
  - Dependencies

Design Class Diagram

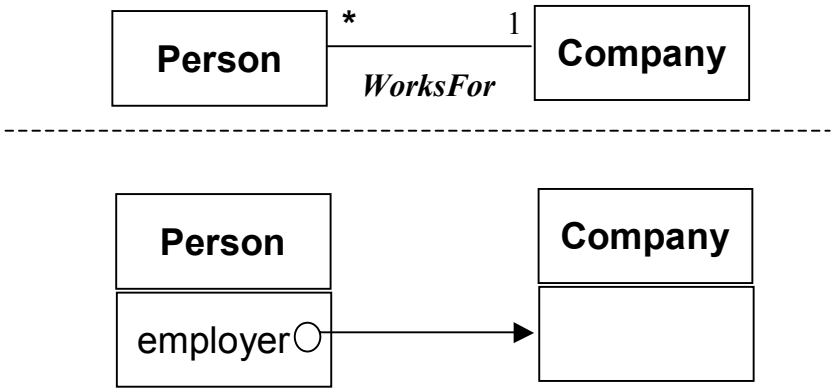
- Domain model vs. design model classes
- Steps in DCD generation:
  - Step1 :Identify software classes and illustrate them
  - Step 2: Add method names
  - Step 3: Add method and class constraints using OCL
  - Step 4: Add associations and navigation information
- An example from VRIS case study…….

Design Class Diagram



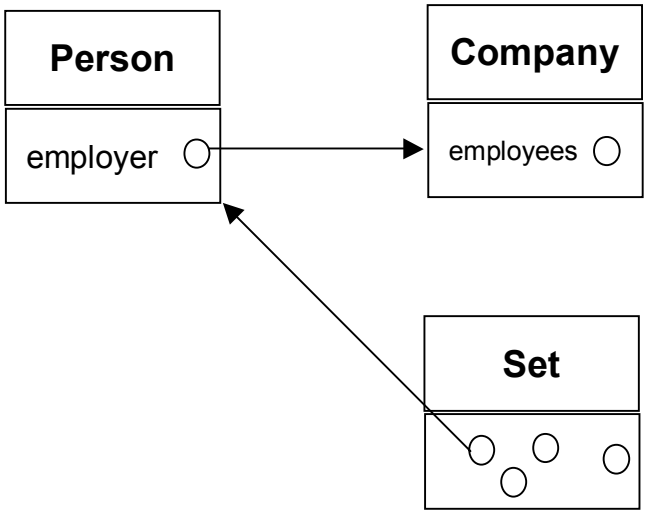
Implementation Model

- Realizing associations using visibility information constitutes the basis of design class diagram development.
- Operations of classes are identified using messages represented in the interaction diagram.
- Implementing associations:



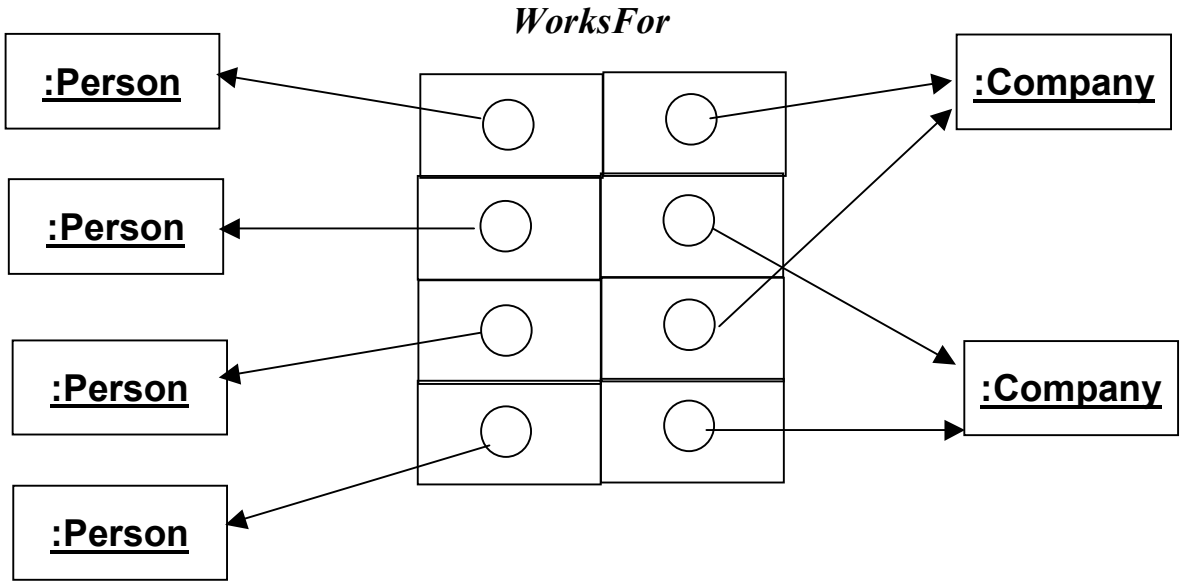
Implementation Model

- Implementing two way associations with pointers



Implementation Model

- **Implementing with an association object:**



Class Design

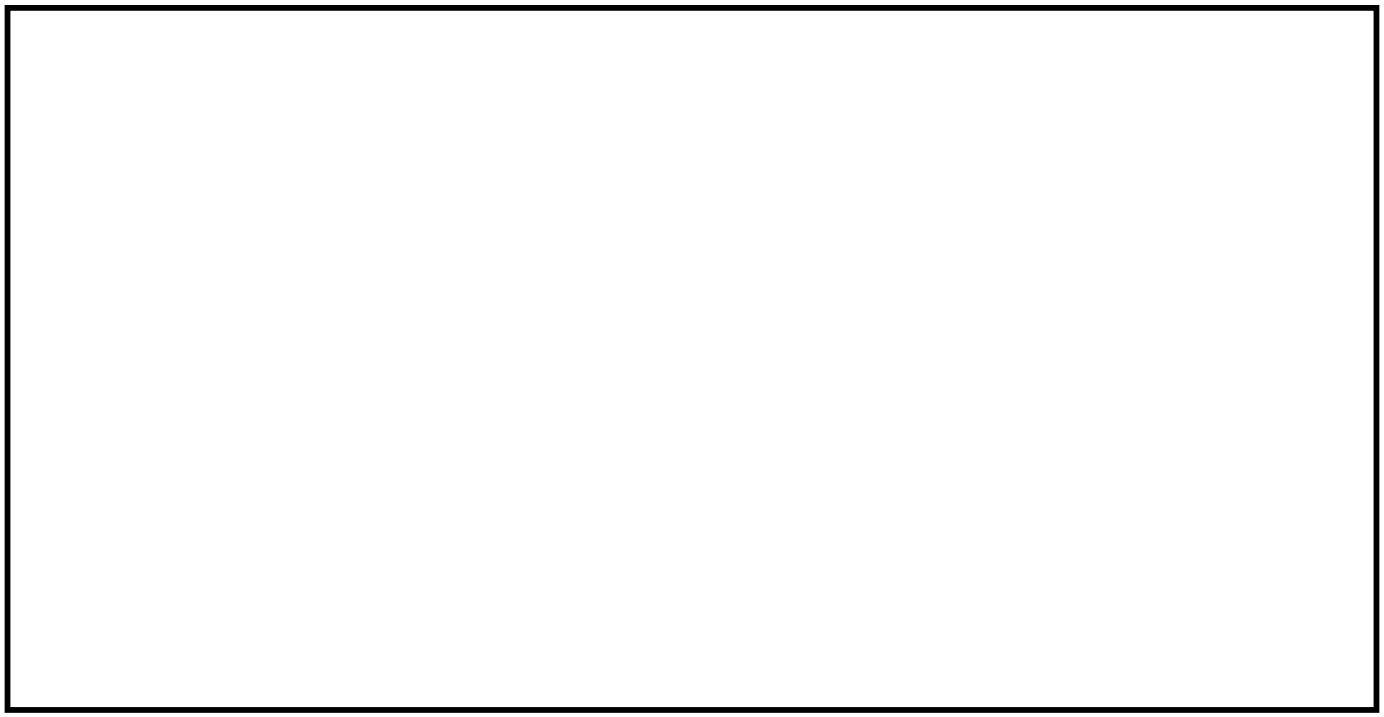
- Class design involves three major steps:
  - Interface design
  - State model design
  - Method logic design (pseudo code or activity diagram)

Interface Design

- Contracts are constraints on a class that enable class users , implementers, and extenders to share the same assumptions about the class. Contracts include three types of constraints:
  - Invariant is a predicate that is always true for all instances of a class. They are used to specify constraints among attributes of a class.
  - A precondition is a predicate that must be true before an operation is invoked. (useful for class user)
  - A postcondition is a predicate that must be true after an operation is invoked. (useful for class developer, extender)

Interface Design

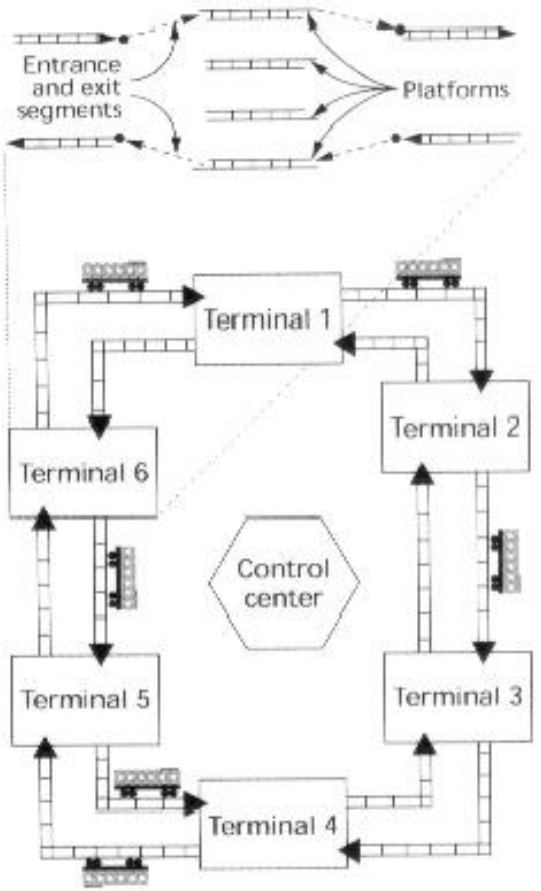
- **Example:**



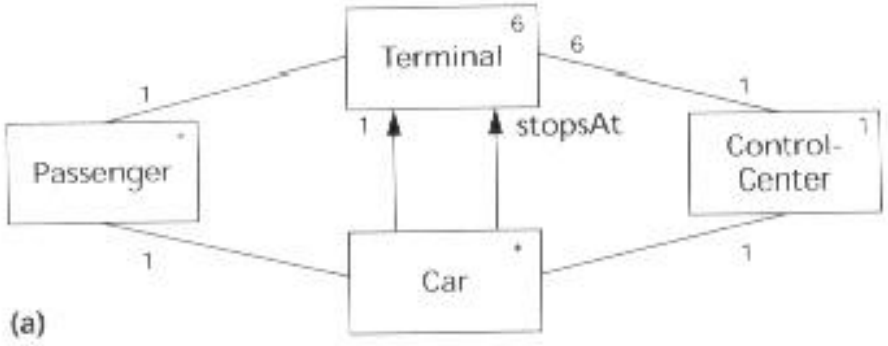
Class State Model Design

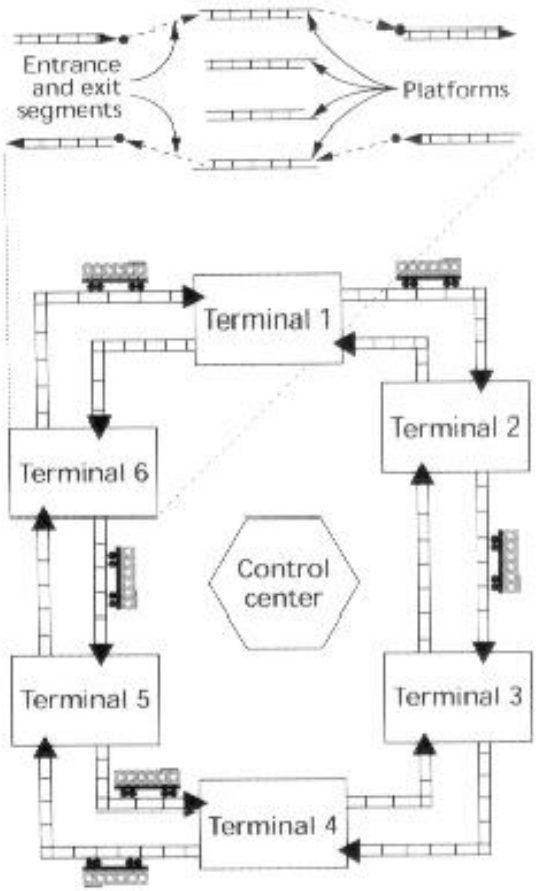
To explain the properties of our language set, we use the automated rail-car system in Figure 1, inspired by Vered Gafni. Six terminals are located on a cyclic path. Each pair of adjacent terminals is connected by two rail tracks, one for clockwise and one for counterclockwise travel. Several railcars are available to transport passengers between terminals. A control center receives, processes, and sends system data to various components.

As the enlargement of Terminal 6 shows, each terminal has a parking area containing four parallel platforms. Each platform can hold a single car. The four rail tracks (two incoming and two outgoing) are connected to a rail segment that can link to any one of the four platforms.

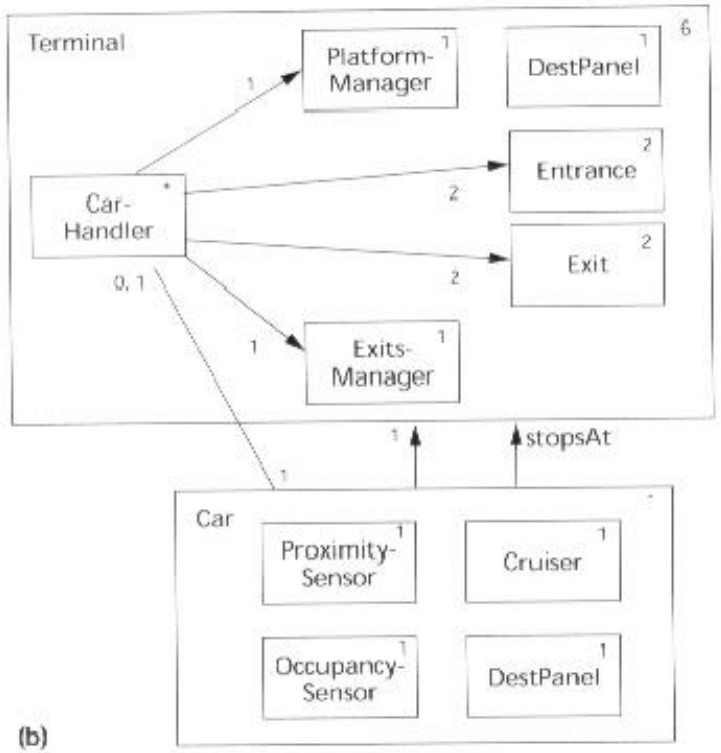


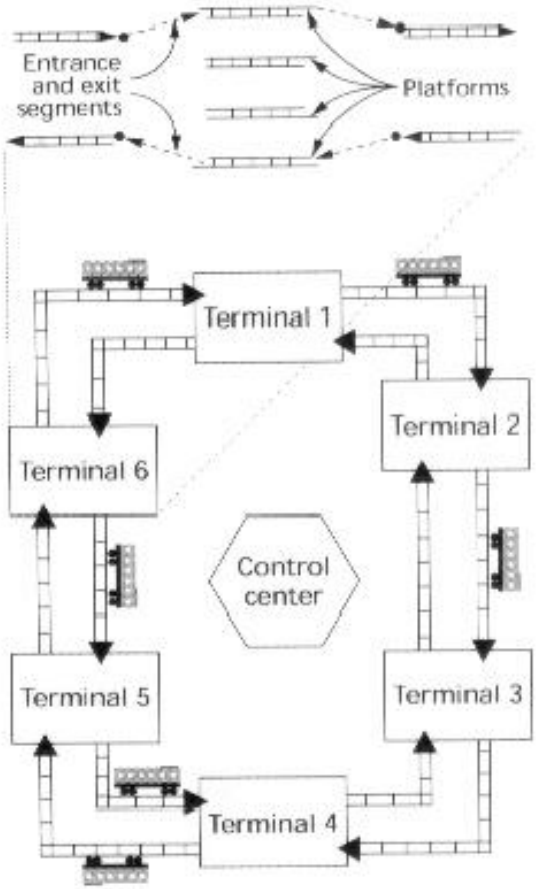
Class State Model Design



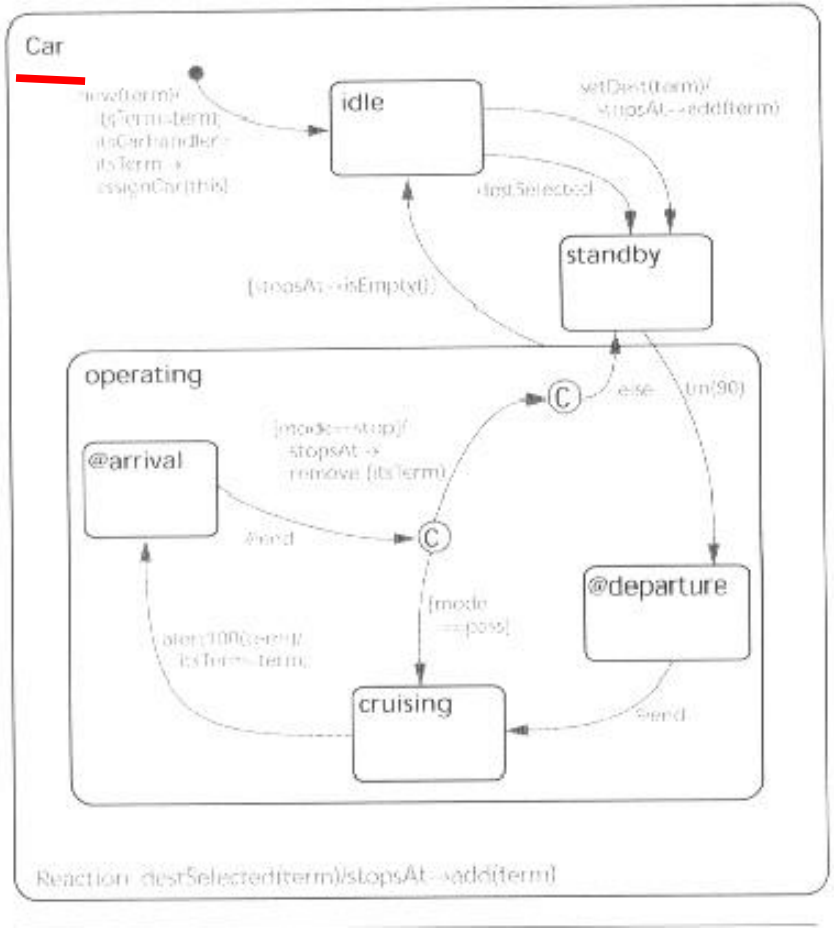


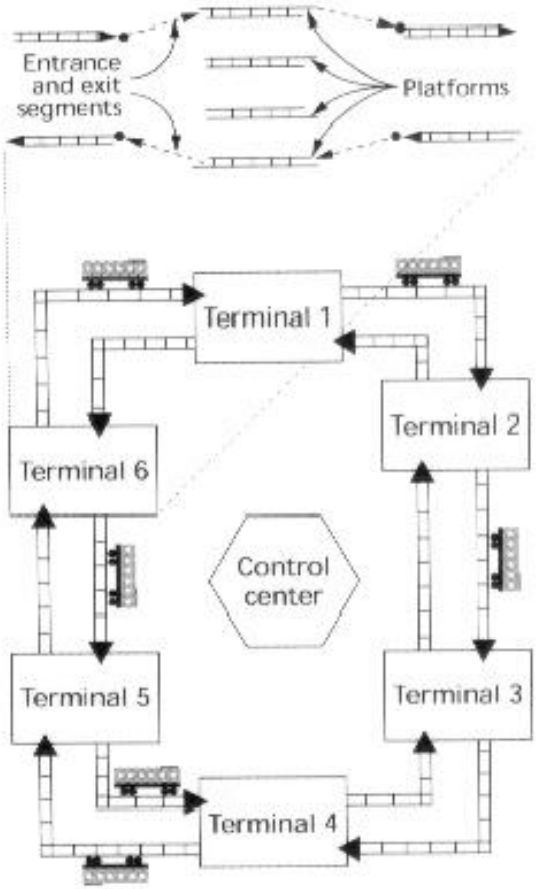
Class State Model Design



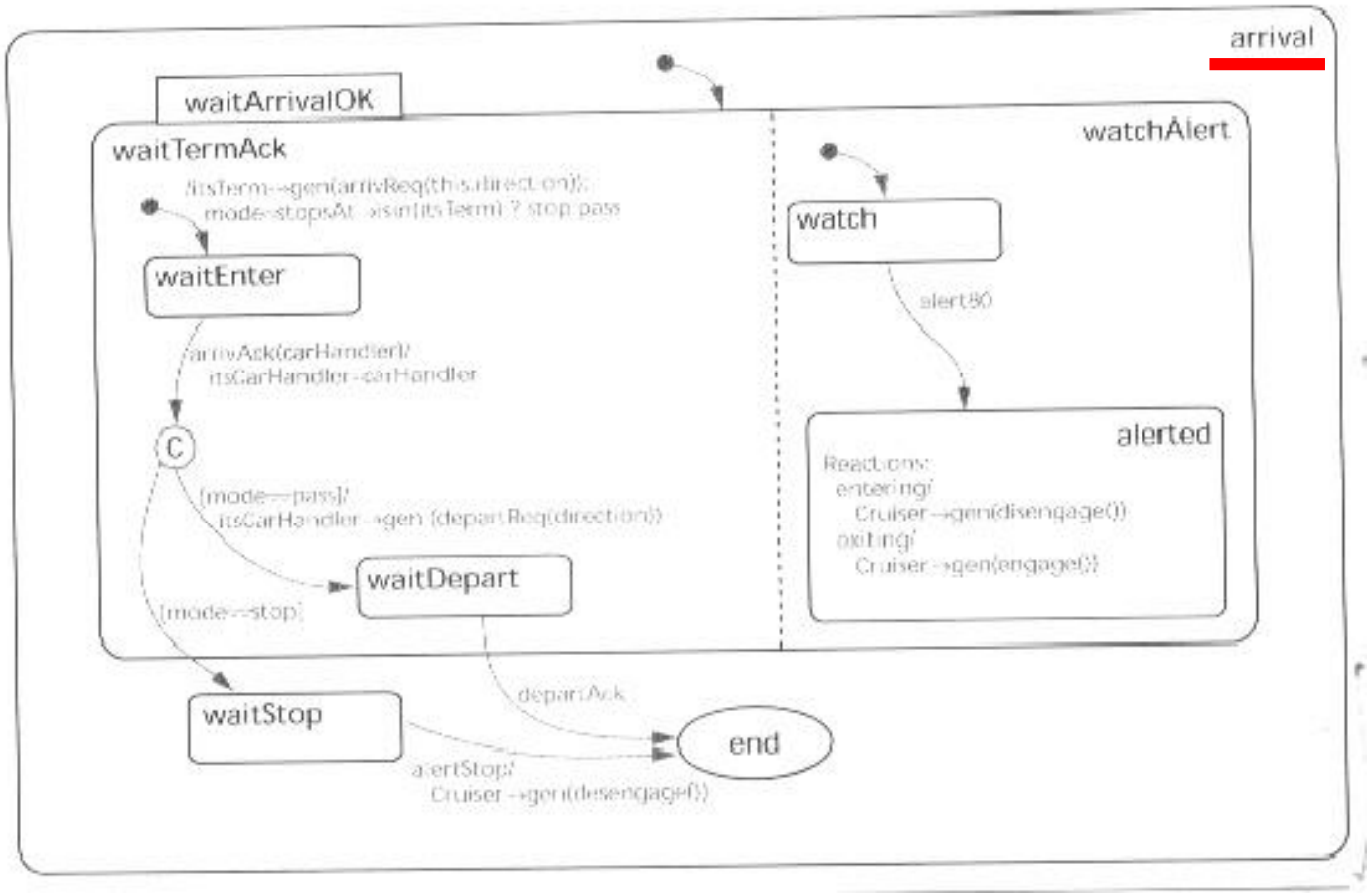


Class State Model Design

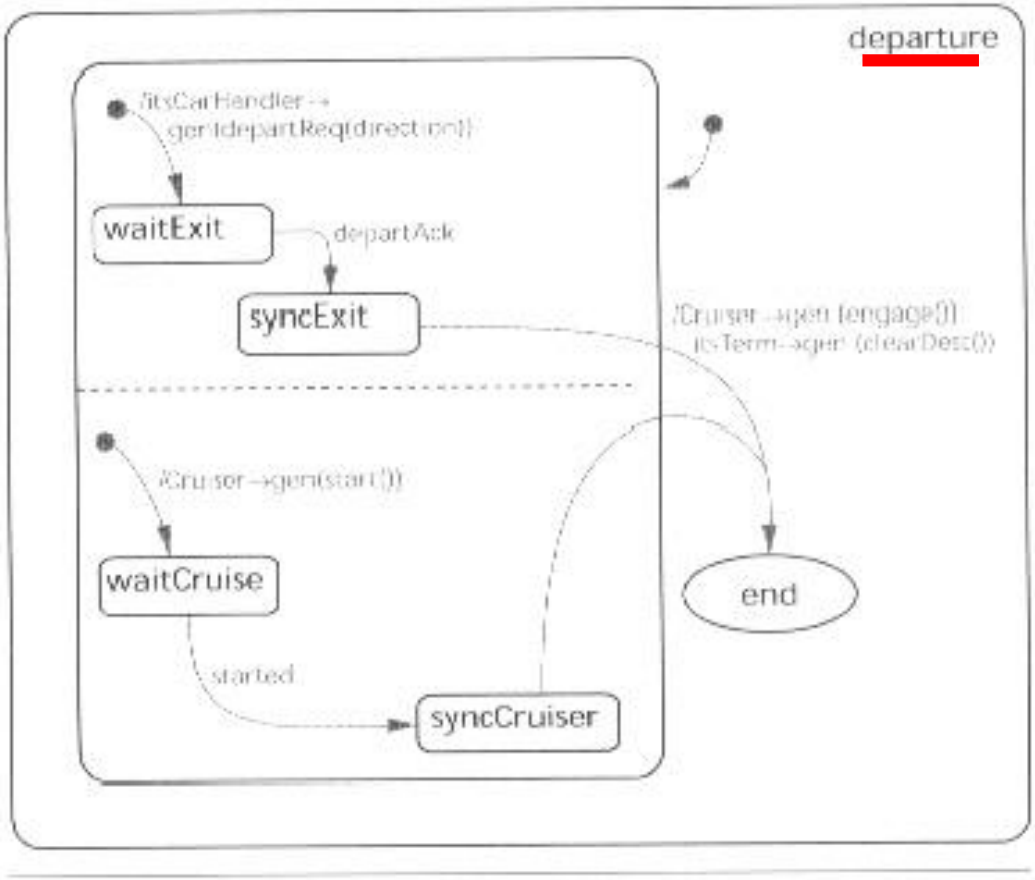




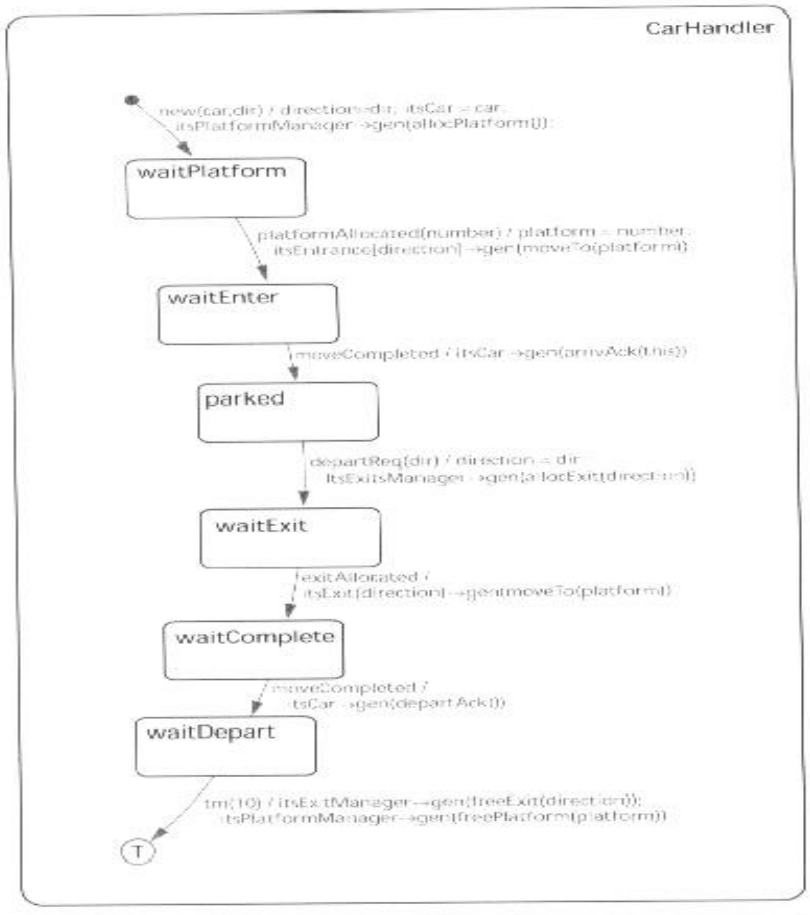
Class State Model Design



Class State Model Design



Class State Model Design



Method Design: Module 1

Module name	*read file name*
Module type	function, returns string
Arguments	none
Error messages	none
Files accessed	none
Files changed	none
Modules called	none
Narrative	The product is invoked by the user by means of the command string *wordcount* <file name> Using an operating system call, this module accesses the contents of the command string that was input by the user, extracts <file name>, and returns it as the value of the function.

Detailed Module Design: Module 2

Module name	*validate file name*
Module type	function, returns boolean
Arguments	file_name : string
Error messages	none
Files accessed	none
Files changed	none
Modules called	none
Narrative	This module makes an operating system call to determine whether file file_name exists. The function returns TRUE if the file exists, and otherwise FALSE.

Detailed Design: Module 3

Module name	*produce output*	
Module type	function	
Input arguments	word count : integer	
Output arguments	none	
Error messages	none	
Files accessed	none	
Files changed	none	
Modules called	*format word count*	
	arguments:	word count : integer formatted word count : string
	display word count	
	arguments:	formatted word count : string
Narrative	This module takes the integer word count passed to it by the calling module and calls *format word count* to have that integer formatted according to the specifications. Then it calls *display word count* to have the line printed.	

Method Logic Design

- Control-flow (UML activity) diagrams can be used, but
- Pseudocode is more common

```
void perform_word_count (void)
{
    string          validated_file_name;
    int             word_count;

    if (get_input (validated_file_name) is FALSE)
        print "error 1: file does not exist";
    else
    {
      set word_count equal to count_number_of_words (validated_file_name);
      if (word_count is equal to -1)
          print "error 2: file is not a text file";
      else
          produce_output (word_count);
    }
```