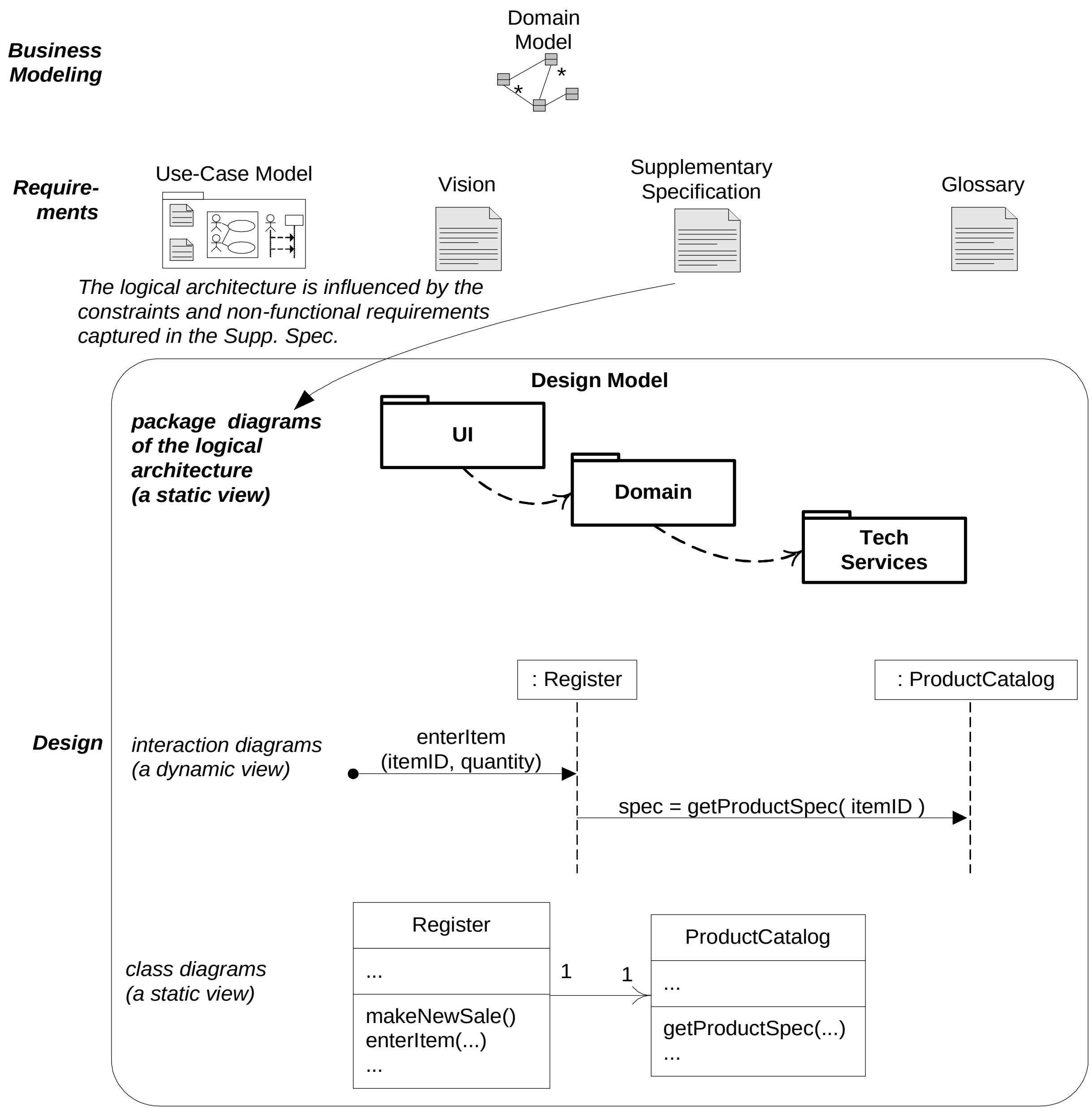
Software Design

Sample UP Artifact Relationships



Different Design Aspects

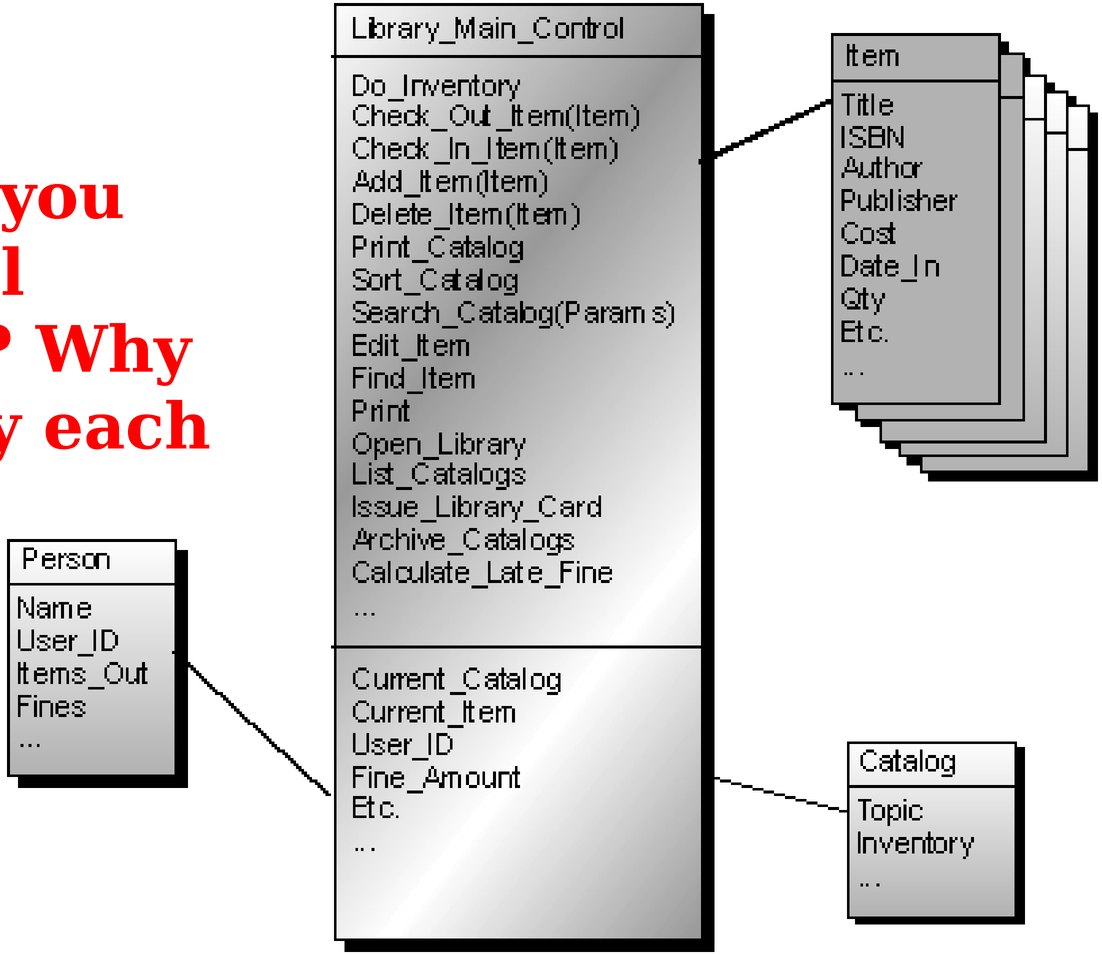
- *Architecture design*:
  - The division into subsystems and components,
    - How these will be connected.
    - How they will interact.
    - Their interfaces.
- *Class design*:
  - The various features of classes.
- *User interface design*
- *Algorithm design*:
  - The design of computational mechanisms.
- *Protocol design*:
  - The design of communications protocol.

What is Software Design?

- A software design expresses a solution to a problem in programming language independent terms.
- This permits a design to be implemented in any programming language.

Simple Library system - Existing design

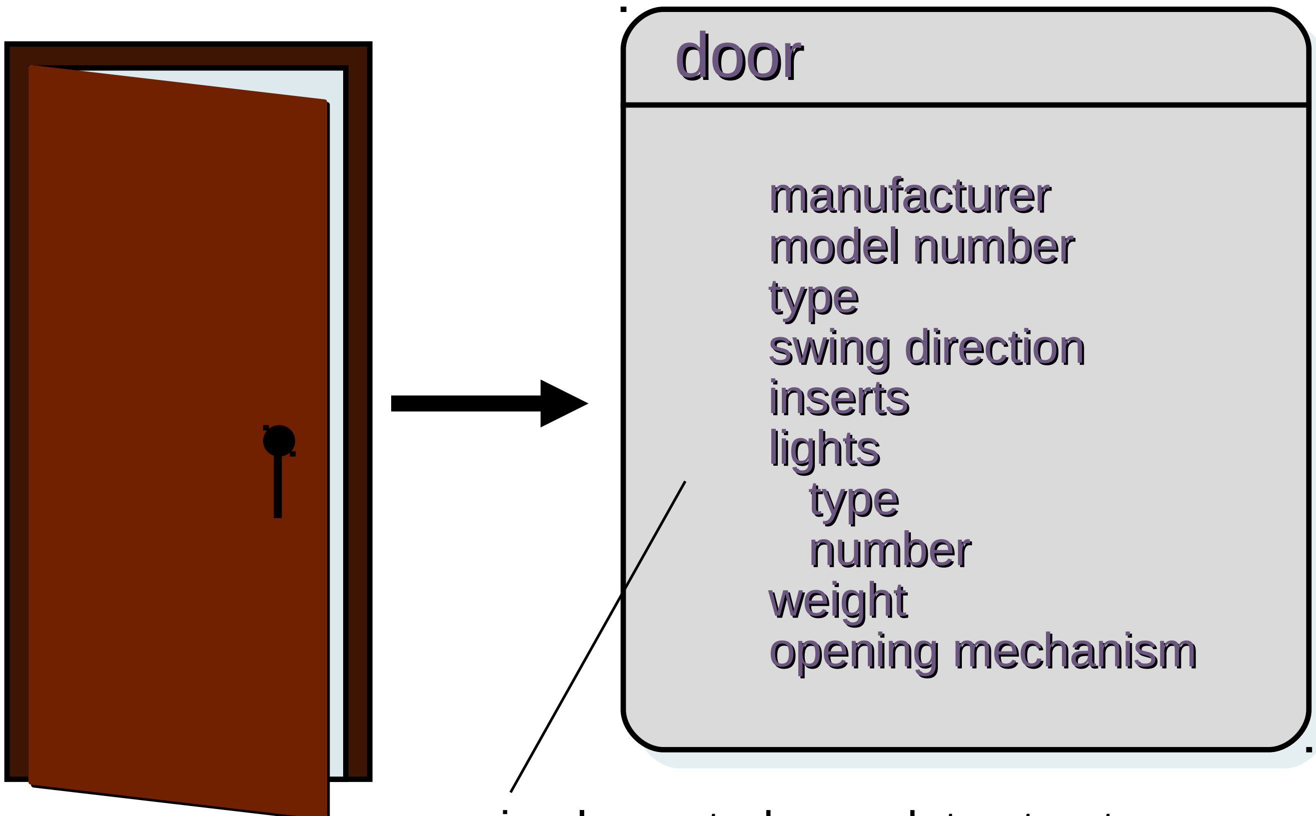
What areas do you see as potential problem areas? Why did you identify each of those areas?



Fundamental Concepts

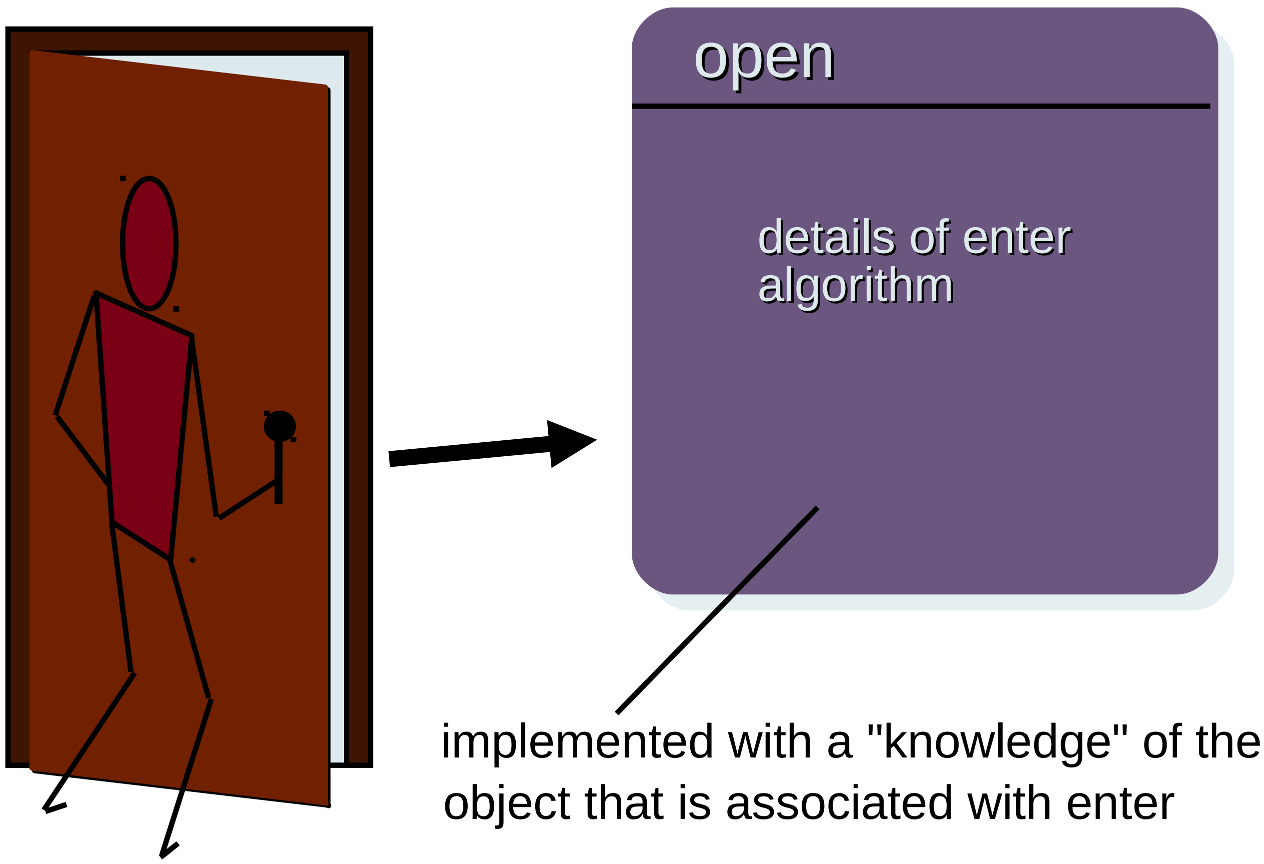
- Abstraction—data, procedure, control
- Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces
- Modularity—compartmentalization of data and function
- Hiding—controlled interfaces
- Refinement—elaboration of detail for all abstractions
- Design Classes—provide design detail that will enable analysis classes to be implemented
- Functional independence—High Cohesion and Low coupling
- **Patterns**—”conveys the essence” of a proven design solution

Data Abstraction



implemented as a data structure

Procedural Abstraction



Separation of Concerns

- Any complex problem can be more easily handled if it is **subdivided into pieces** that can each be solved and/or optimized independently
- A *concern* is a feature or behavior that is specified as part of the requirements model for the software
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

Modularity

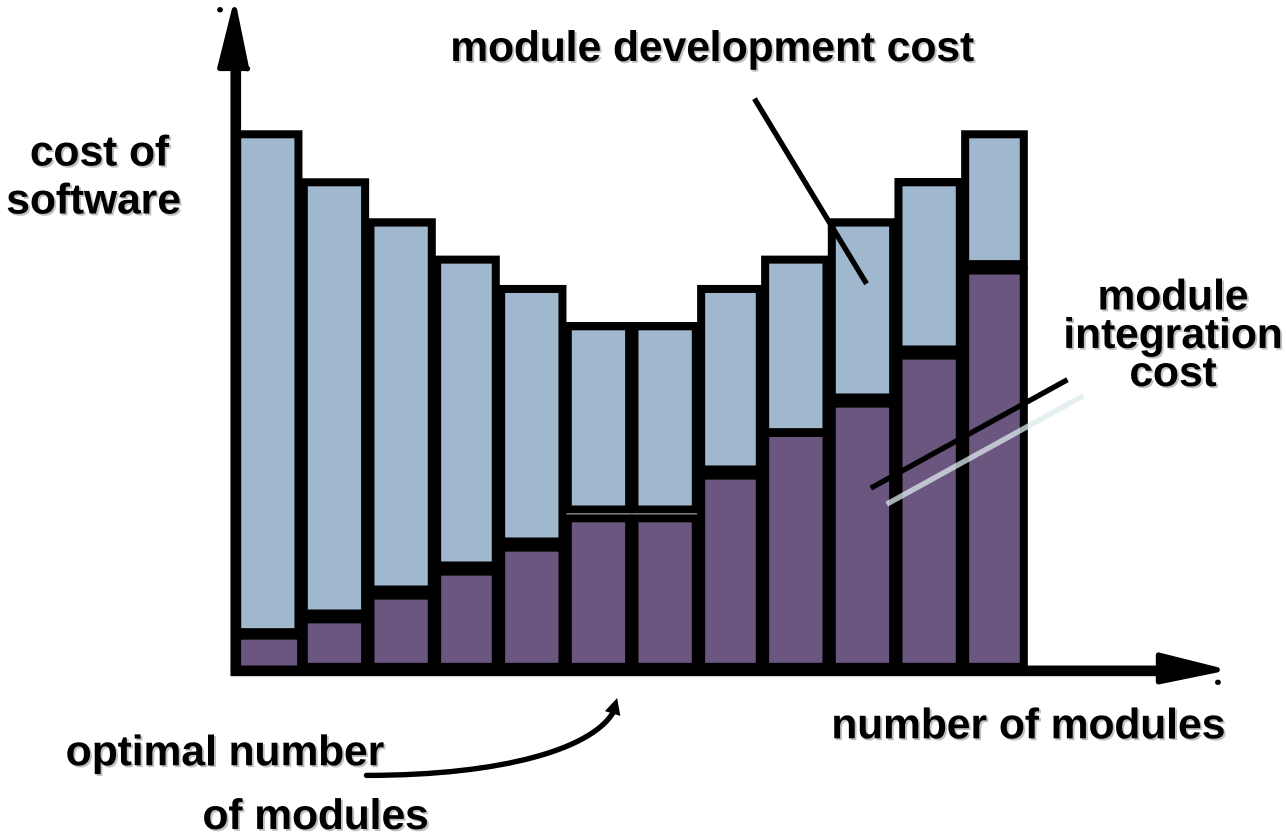
- "modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78].

- **Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.**

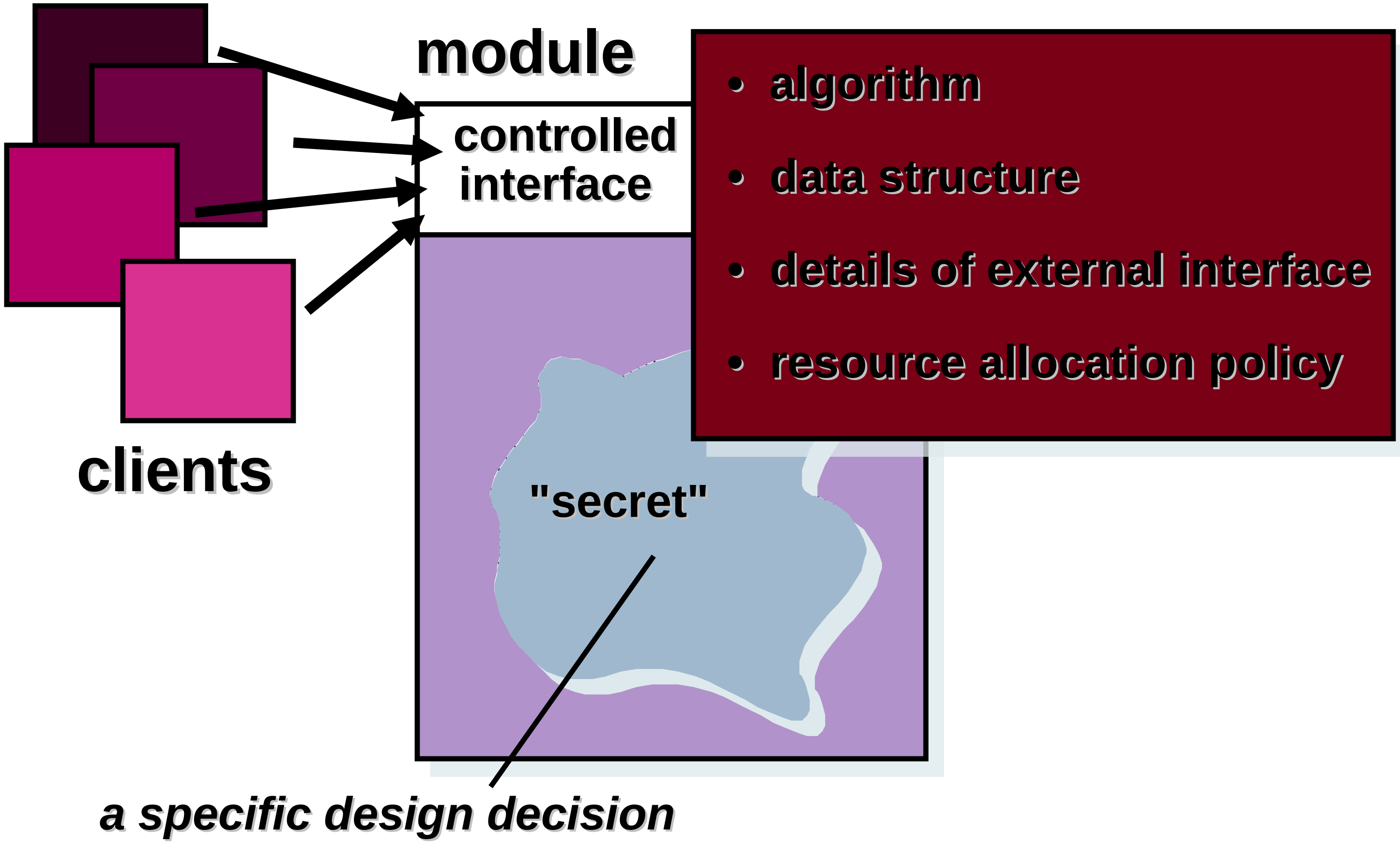
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the cost required to build the software.

Modularity: Trade-offs

What is the "right" number of modules for a specific software design?



Information Hiding



Why Information Hiding?

- reduces the likelihood of “side effects”
- limits the global impact of **local design** decisions
- emphasizes communication through **controlled interfaces**
- discourages the use of global data
- leads to **encapsulation**—an attribute of high quality design
- results in higher quality software

Stepwise Refinement

open

walk to door;
reach for knob;

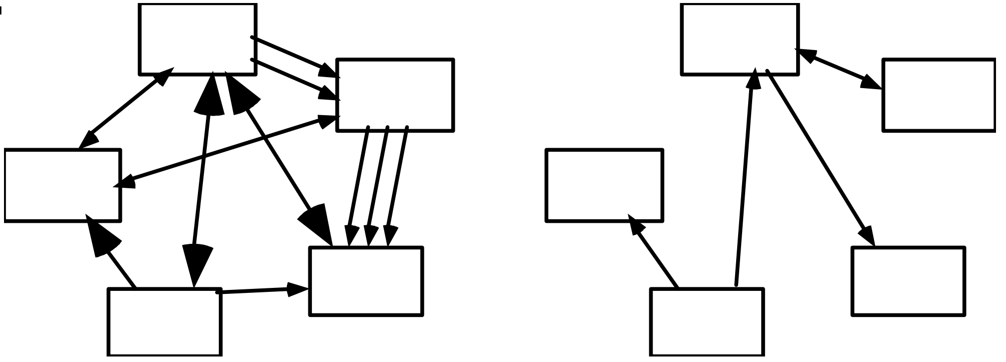
open door;

walk through;
close door.

```
repeat until door opens
turn knob clockwise;
if knob doesn't turn, then
    take key out;
    find correct key;
    insert in lock;
endif
pull/push door
move out of way;
end repeat
```

Coupling

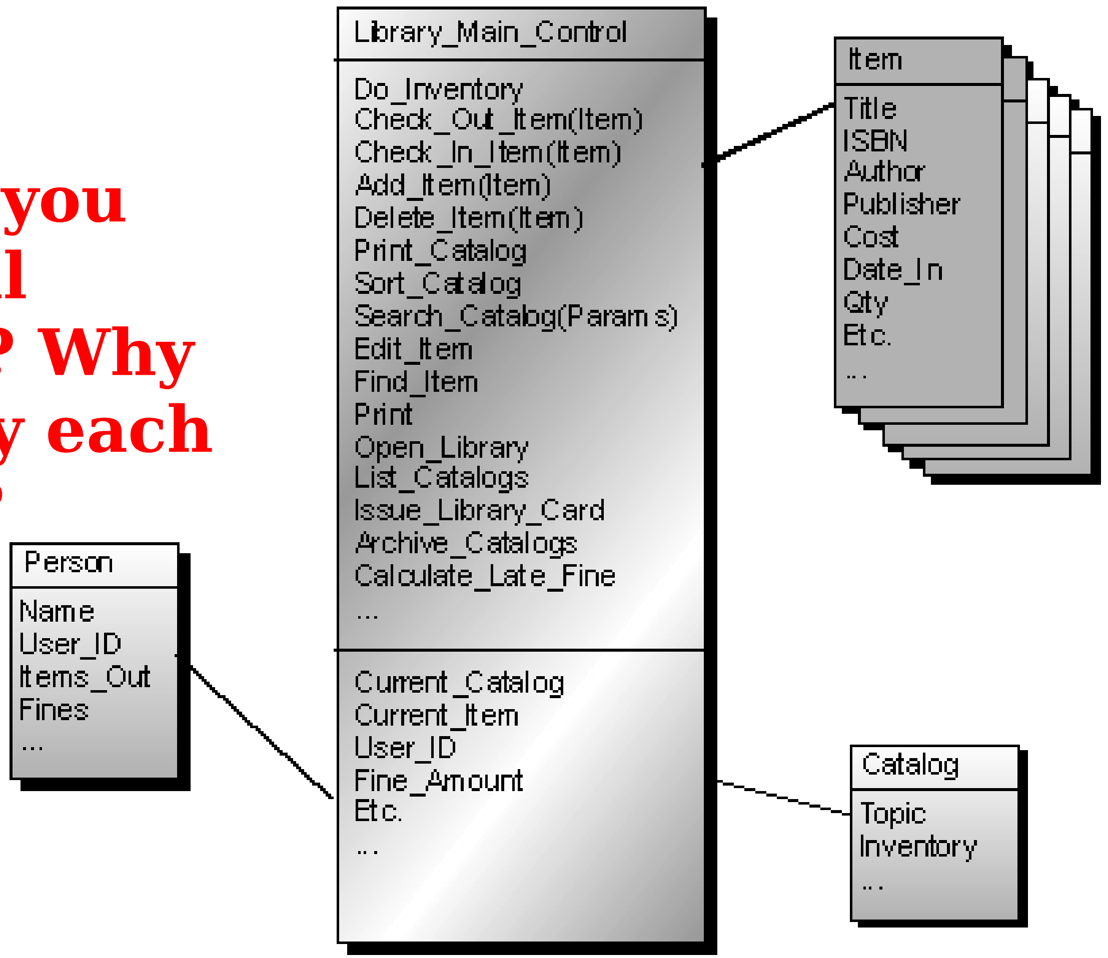
- *Coupling* occurs when there are *interdependencies* between one module and another



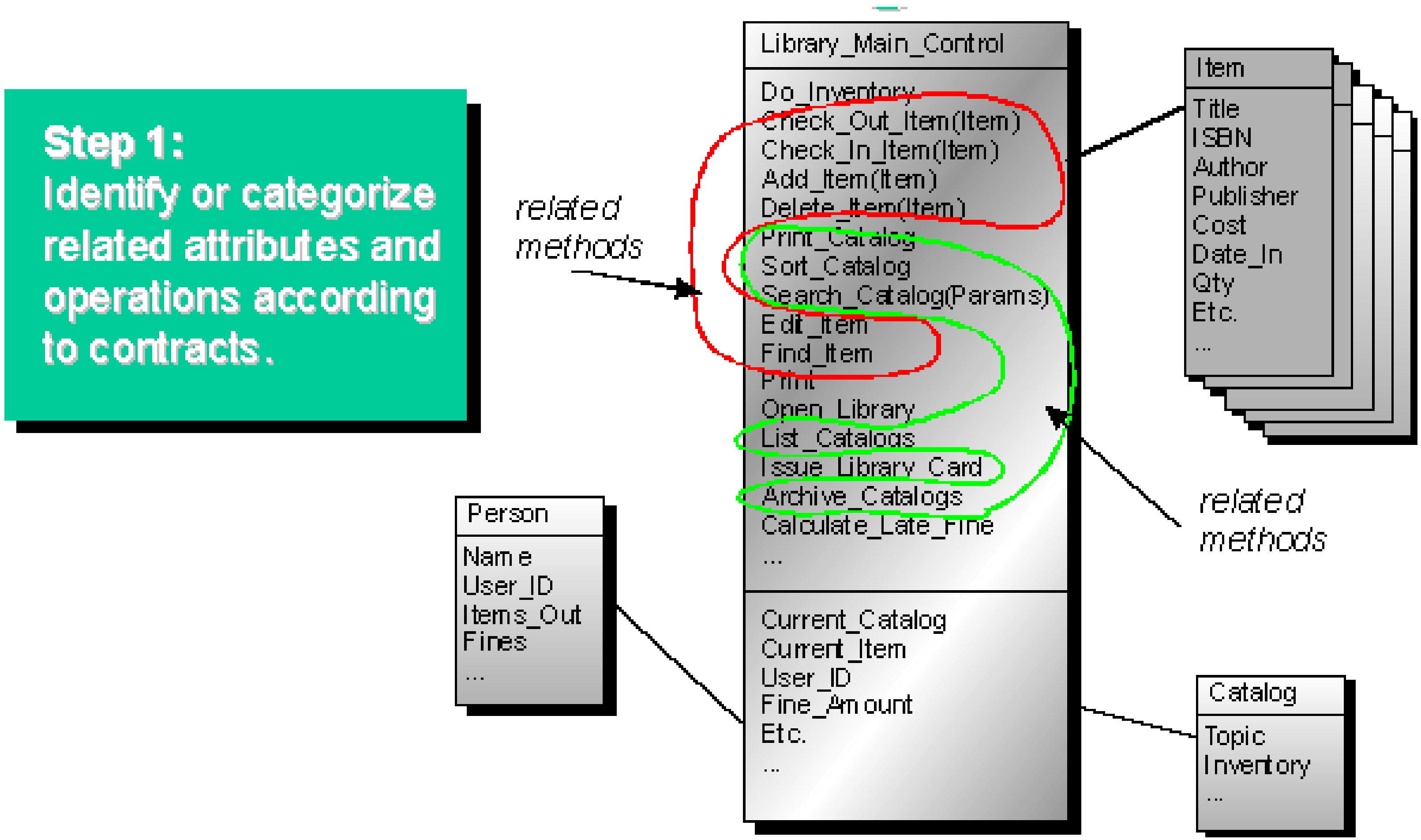
  - When interdependencies exist, changes in one place will require changes somewhere else.
  - A network of interdependencies makes it hard to see at a glance how some component works.

Revisting Library system - Existing design

What areas do you see as potential problem areas? Why did you identify each of those areas?

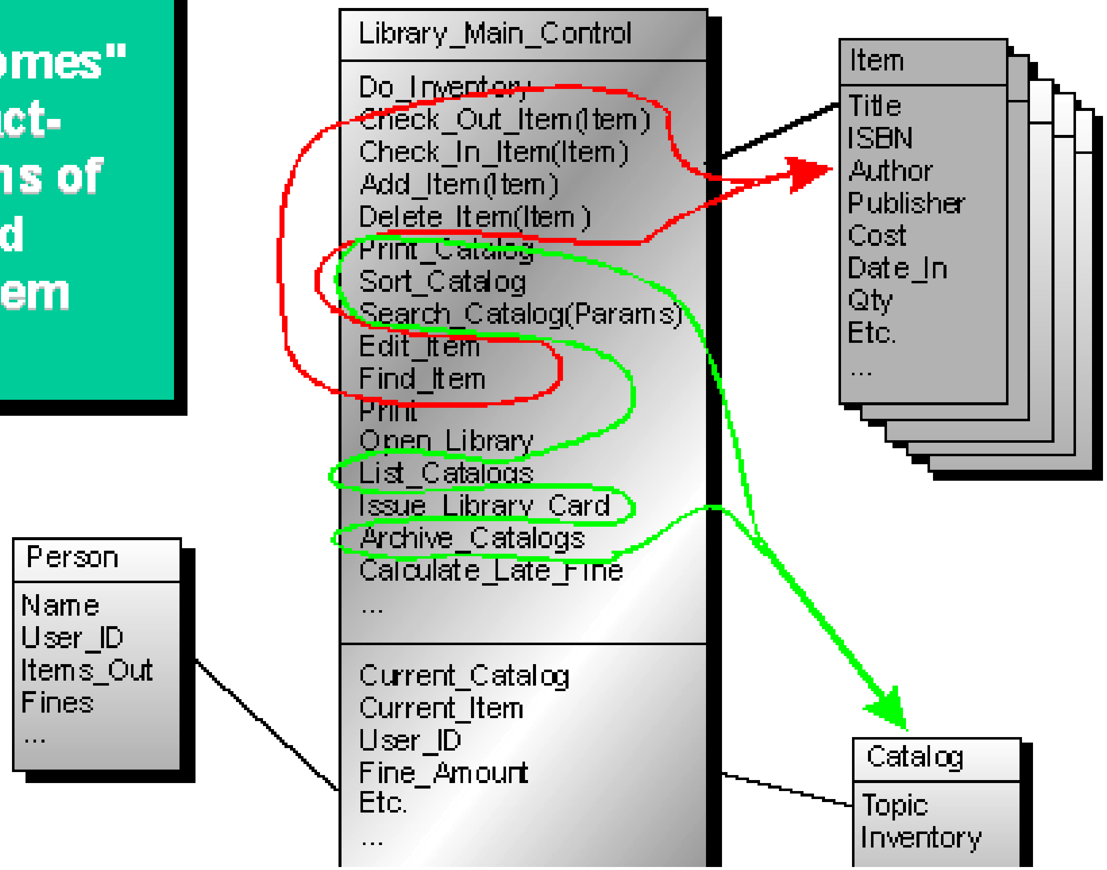


Library system – Changing the design

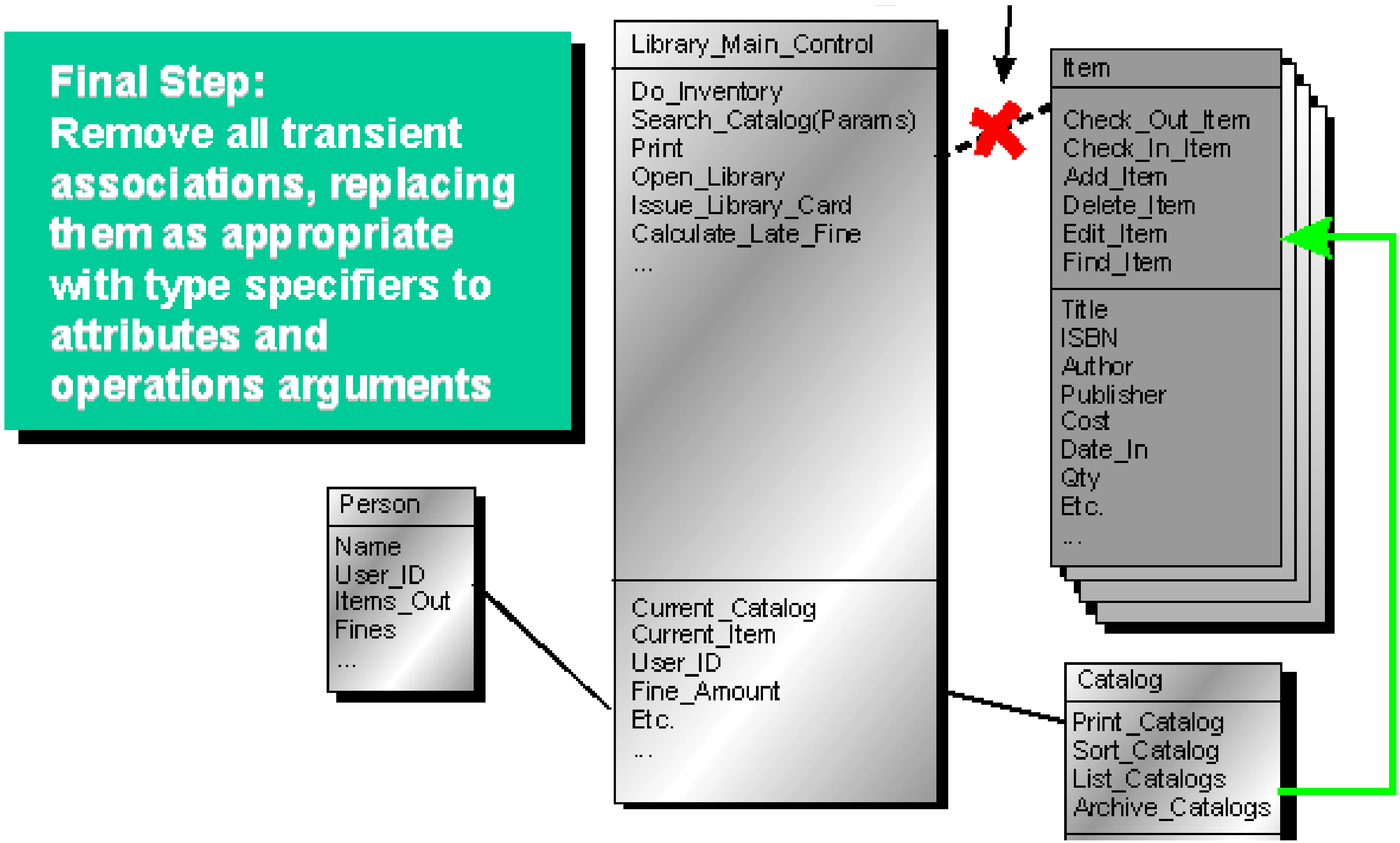


Library system – Changing the design

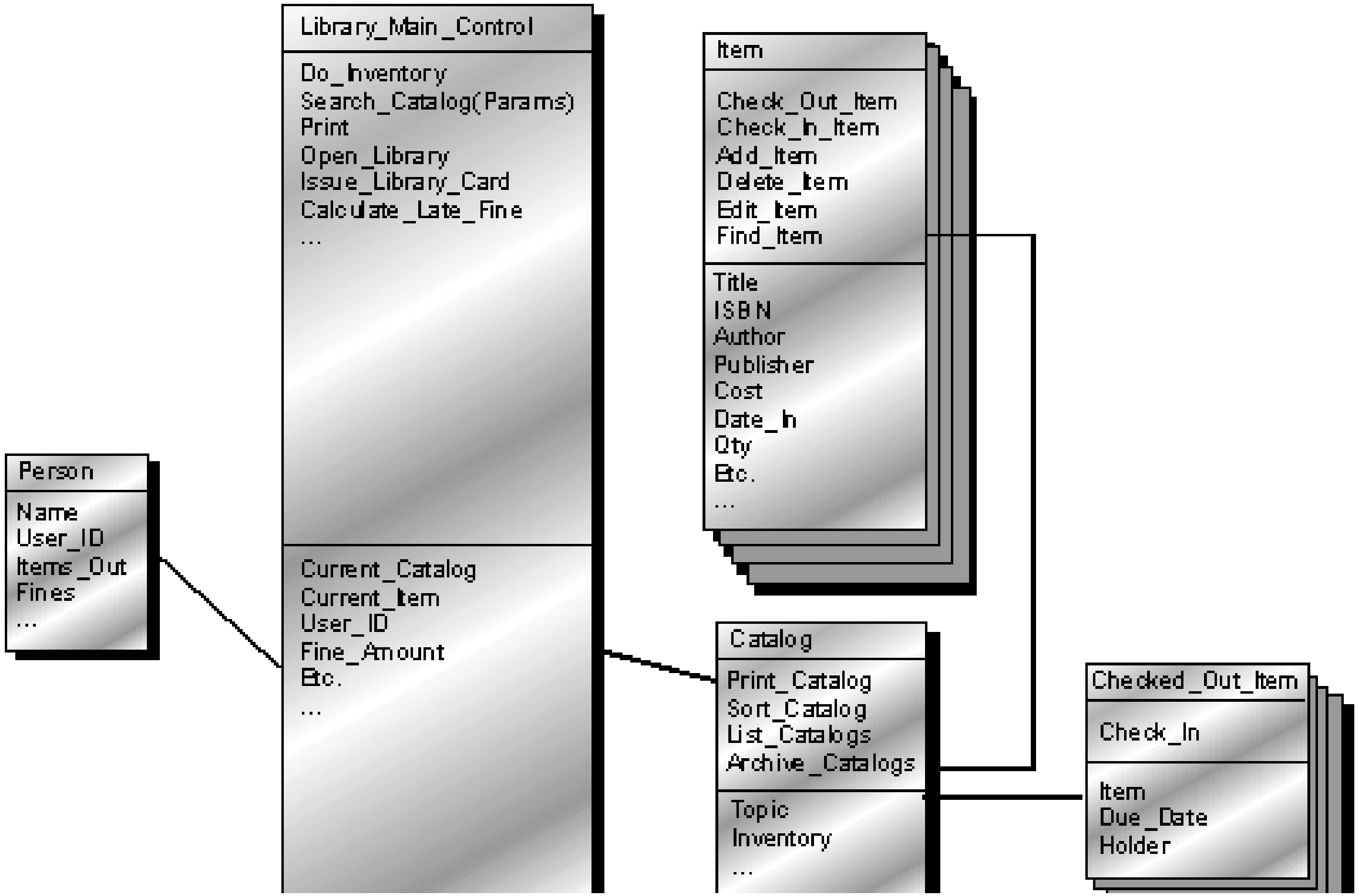
Step 2:
Find "natural homes" for these contract-based collections of functionality and them migrate them there



Library system – Changing the design



Library system – Changing the design



Design Patterns

Design Pattern Template

Pattern name**—describes the essence of the pattern in a short but expressive name**

Intent**—describes the pattern and what it does**

Also-known-as**—lists any synonyms for the pattern**

Motivation**—provides an example of the problem**

Applicability**—notes specific design situations in which the pattern is applicable**

Structure**—describes the classes that are required to implement the pattern**

Participants**—describes the responsibilities of the classes that are required to implement the pattern**

Collaborations**—describes how the participants collaborate to carry out their responsibilities**

Consequences**—describes the "design forces" that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented**

Related patterns**—cross-references related design patterns**

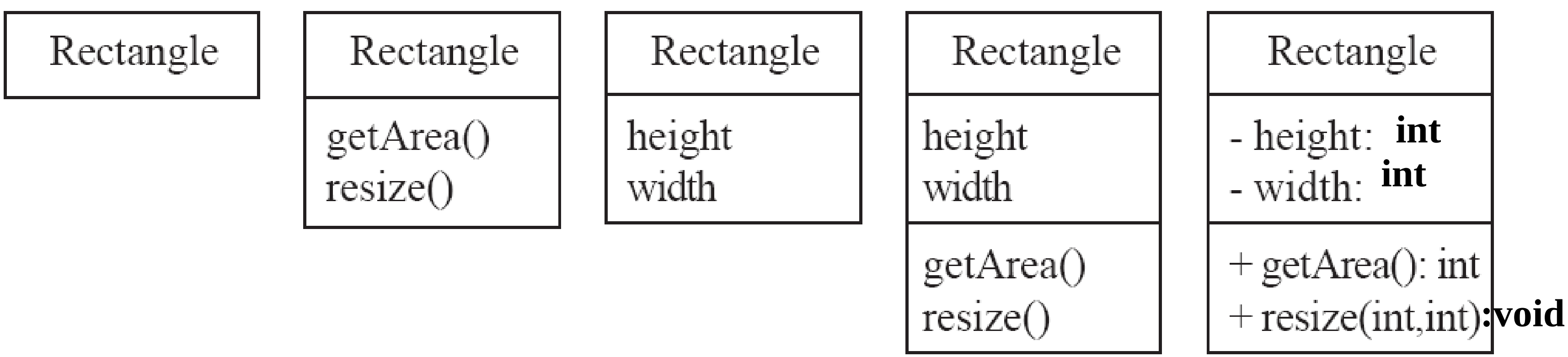
Essentials of UML Class Diagrams

- *The main symbols shown on class diagrams are:*
  - *Classes*
    - represent the types of data themselves
  - *Associations*
    - represent linkages between instances of classes
  - *Attributes*
    - are simple data found in classes and their instances
  - *Operations*
    - represent the functions performed by the classes and their instances
  - *Generalizations*
    - group classes into inheritance hierarchies

Classes

- A class is simply represented as a box with the name of the class inside
  - The diagram may also show the attributes and operations
  - The complete signature of an operation is:

    operationName(parameterName: parameterType …): returnType

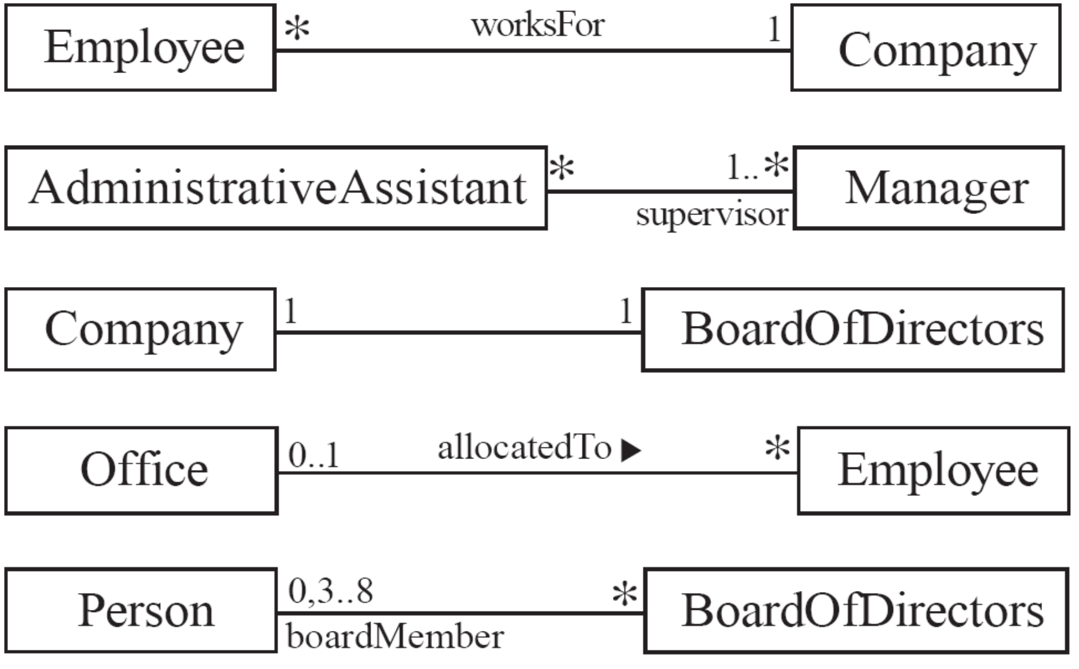


Associations and Multiplicity

An *association* is used to show how two classes are related to each other

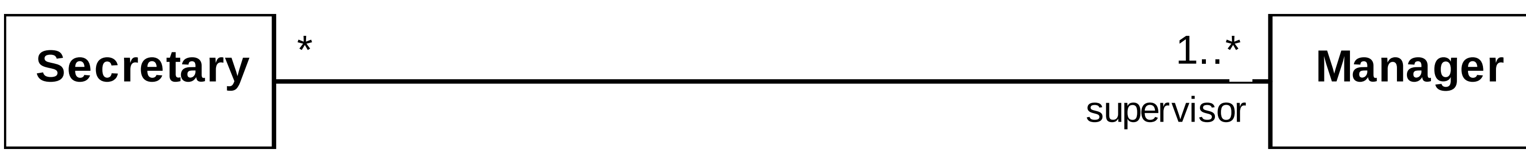
Symbols indicating *multiplicity* are shown at each end of the association

Each association can be labelled, to make explicit the nature of the association



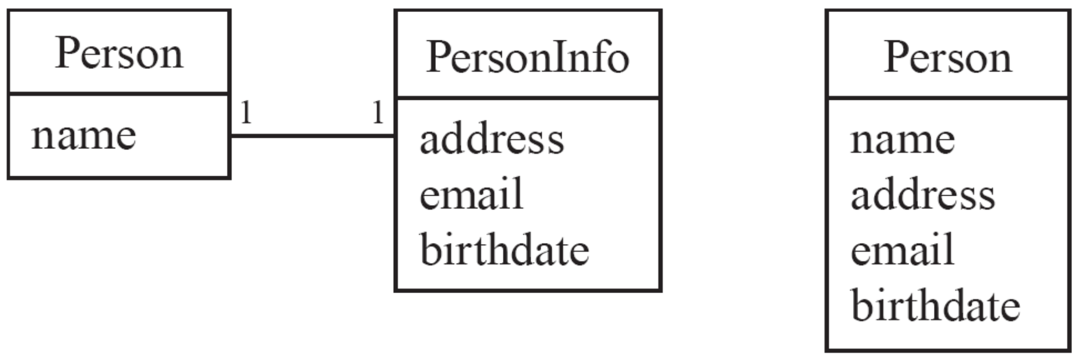
Analyzing and validating associations

- **Many-to-many**
  - A secretary can work for many managers
  - A manager can have many secretaries
  - Secretaries can work in pools
  - Managers can have a group of secretaries
  - Some managers might have zero secretaries.
  - Is it possible for a secretary to have, perhaps temporarily, zero managers?



Analyzing and validating associations

- Avoid unnecessary one-to-one associations
- Avoid this do this

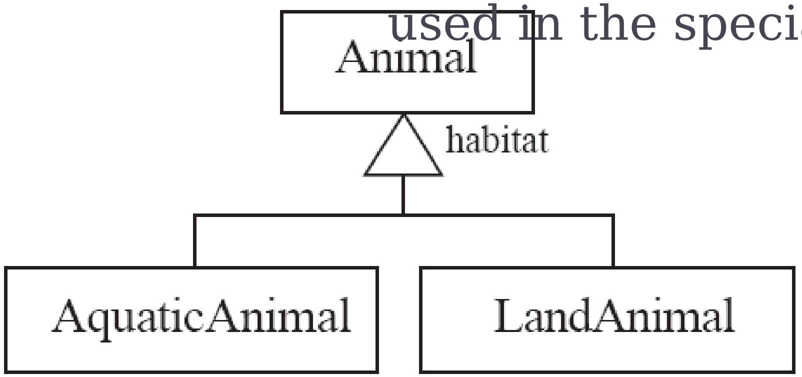


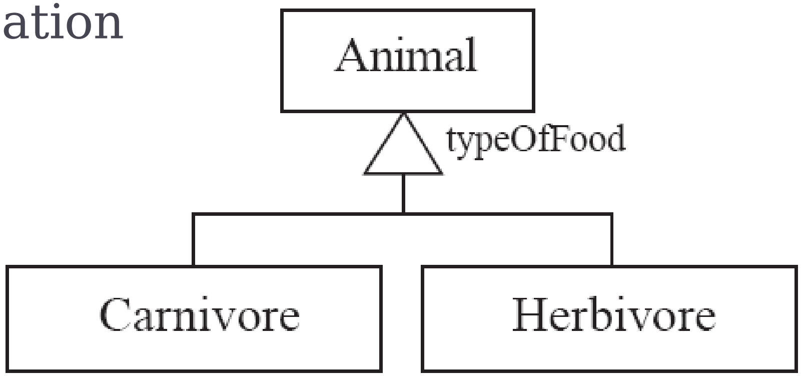
Directionality in associations

- Associations are by default are undefined, though many tools treat these as *bi-directional*.
- It is possible to limit the direction of an association by adding an arrow at one end

Generalization

- Specializing a superclass into two or more subclasses
  - The *discriminator* is a label that describes the criteria used in the specialization





Associations versus generalizations in object diagrams

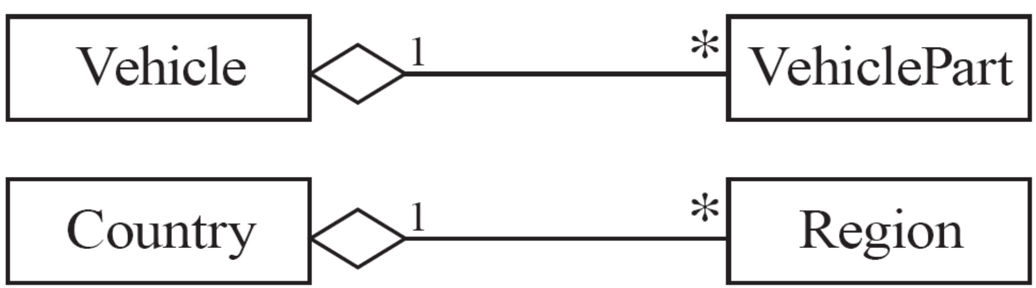
- Associations describe the relationships that will exist between *instances* at **run time**.
  - When you show an instance diagram generated from a class diagram, there will be an instance of *both* classes joined by an association

- Generalizations describe relationships between *classes* in class diagrams.
  - They do not appear in instance diagrams at all.
  - An instance of any class should also be considered to be an **instance** of each of that class's **superclasses**

More Advanced Features: Aggregation

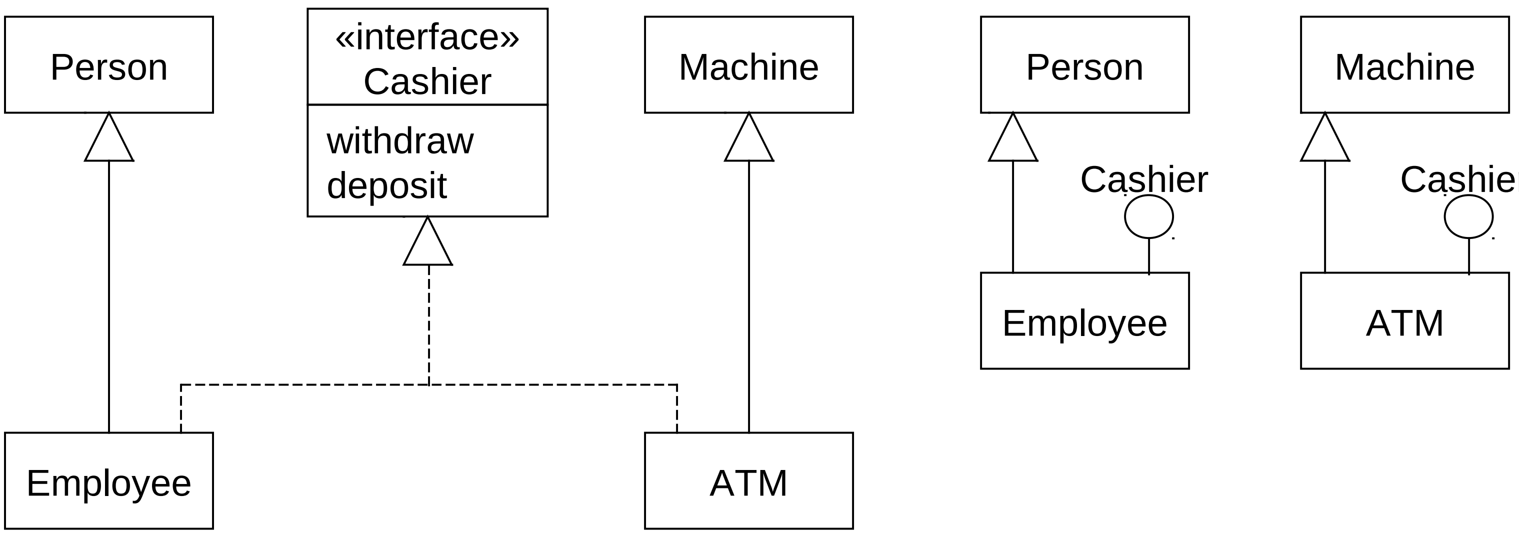
- Aggregations are special associations that represent ‘part-whole’ relationships.
  - The ‘whole’ side is often called the *assembly* or the *aggregate*
  - This symbol is a shorthand notation association named isPartOf

- As a general rule, you can mark an association as an aggregation if the following are true:
  - You can state that
    - the parts ‘are part of’ the aggregate
    - or the aggregate ‘is composed of’ the parts
  - When something **owns** or controls the aggregate, then they also own or control the parts



Interfaces

- An interface describes a *portion of the visible behaviour* of a set of objects.
  - An *interface* is similar to a class, except it lacks instance variables and implemented methods

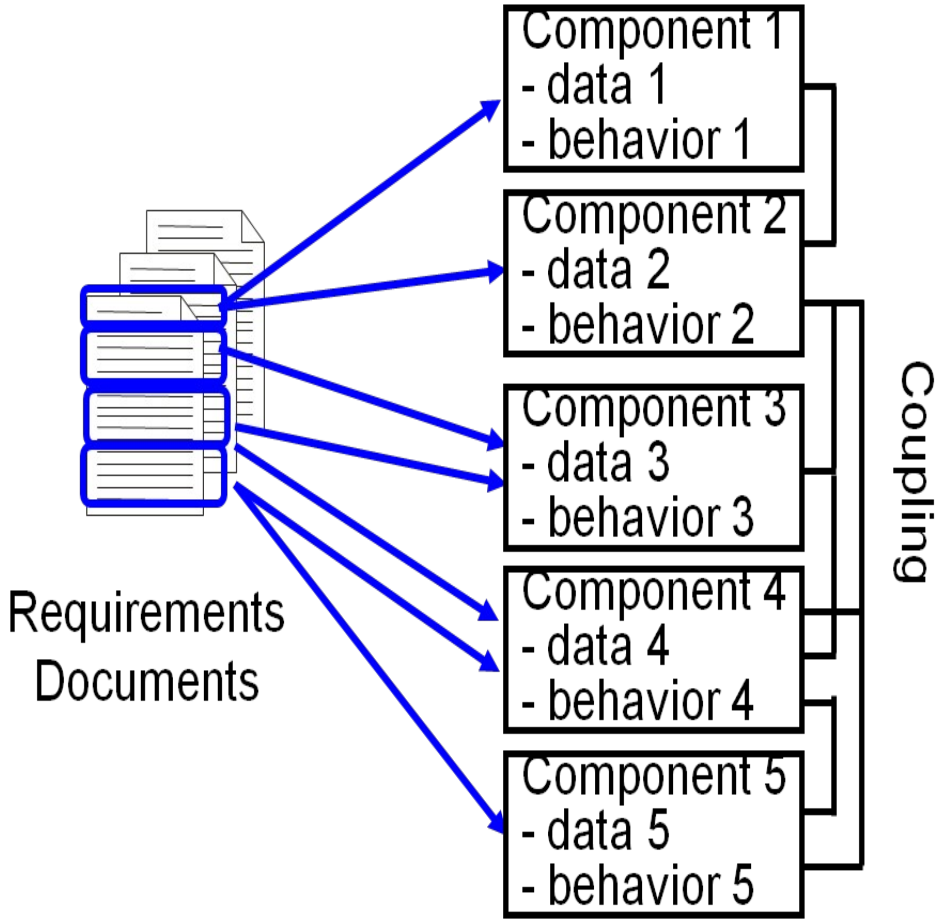


Suggested sequence of activities

- Identify a first set of candidate **classes**
- Add **associations** and **attributes**
- Find **generalizations**
- List the main **responsibilities** of each class
- Decide on specific **operations**
- **Iterate** over the entire process until the model is satisfactory
  - Add or delete classes, associations, attributes, generalizations, responsibilities or operations
  - Identify interfaces
  - Apply design patterns

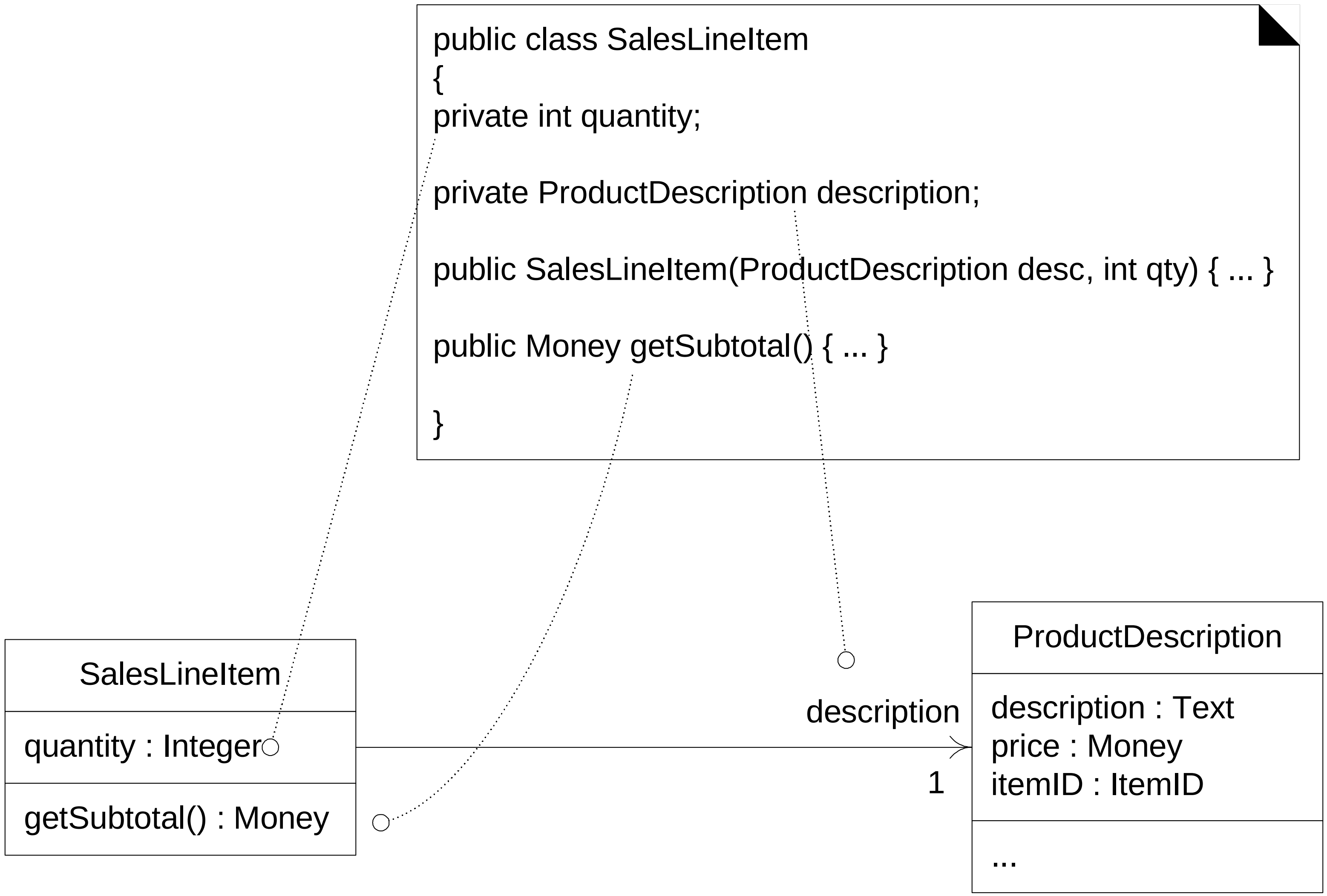
Don't be too disorganized. Don't be too rigid either!

Mapping Requirements to Design Components



- Design must satisfy requirements
  - Everything (data and behavior) in the requirements must be mapped to the design components
  - Decide what functionality goes into which component
- As you do the mapping, assess functional cohesion and coupling
  - Strive for **low** coupling and **high** cohesion

Mapping UML class diagram to code



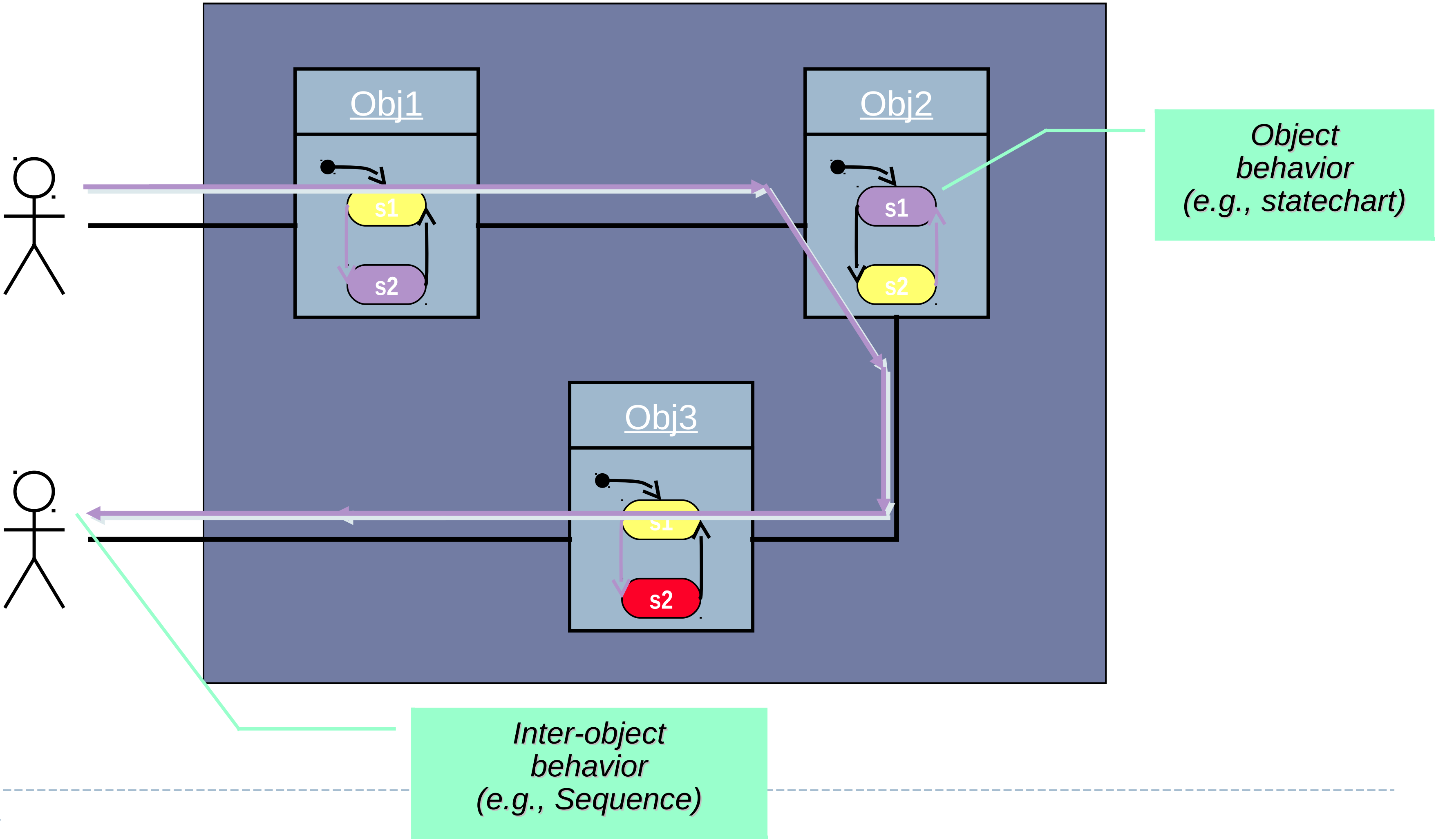
UML Behavioral Models

Specifying behavior using the UML

- Class models describe objects and their relationships
  - Behavior can be specified in terms of operation pre and postconditions, but behavior is not the primary focus of a class model
- Behavioral models in the UML
  - State diagrams: describe control aspects of a system – provides descriptions sequences of operations without regard for what the operations do.
  - Interaction models (Sequence diagrams): describe interactions among objects

How Things Happen in UML

- In UML, all behavior results from the actions of (active) objects



Interaction Diagrams

- Interaction diagrams are used to model the dynamic aspects of a software system
  - They help you to visualize how the system runs.
  - An interaction diagram is often built from a use case and a class diagram.
    - The objective is to show how a set of objects accomplish the required interactions with an actor.

Interactions and messages

- Interaction diagrams show how a set of actors and objects communicate with each other to perform:
  - The steps of a use case, or
  - The steps of some other piece of functionality.
- The set of steps, taken together, is called an ***interaction***.
- Interaction diagrams can show several different types of communication.
  - E.g. method calls, messages send over the network
  - These are all referred to as ***messages***.

Elements found in interaction diagrams

- Instances of classes
  - Shown as boxes with the class and object identifier underlined
- Actors
  - Use the stick-person symbol as in use case diagrams
- Messages
  - Shown as arrows from actor to object, or from object to object

Sequence Diagrams – Modeling Interaction

1. Student selects Register for Courses option

2. System retrieves a list of the available courses

3. Student specifies the desired course

4. System shows a list of the available sections

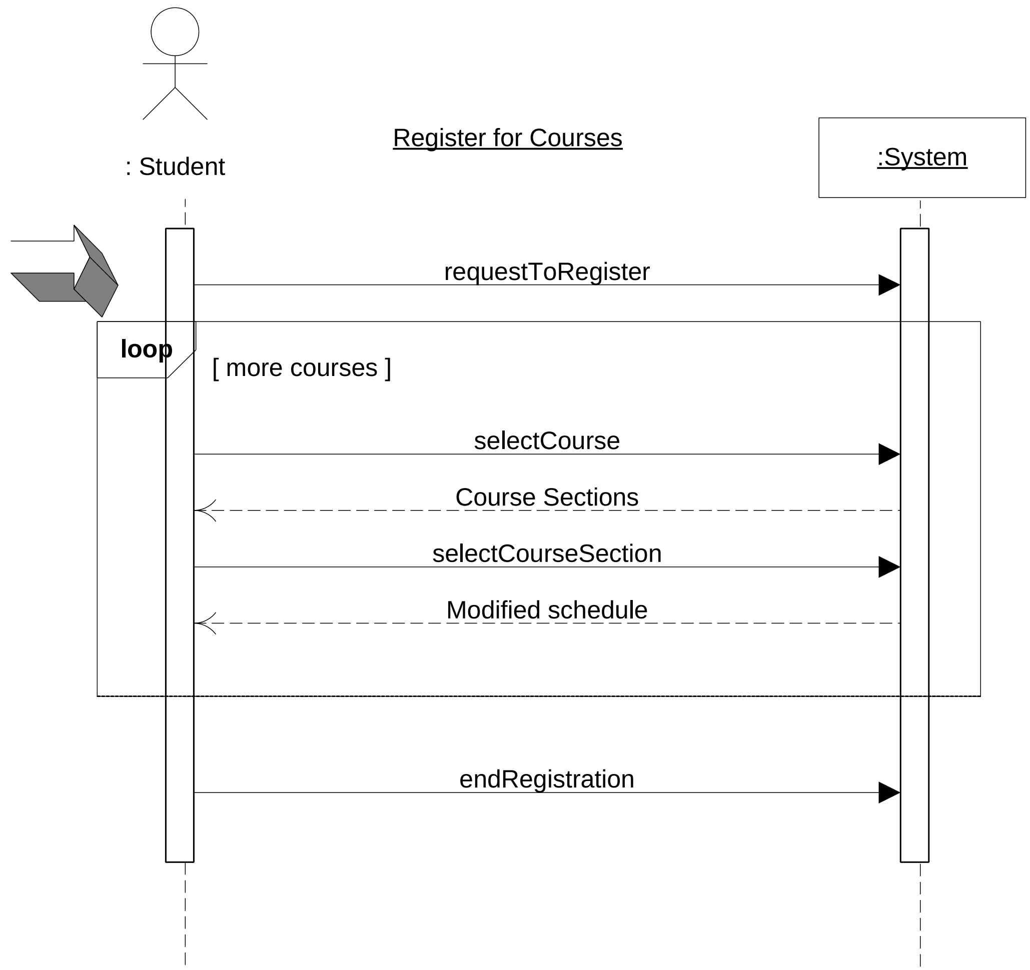
5. Student selects the course section

6. System verifies if the student has passed prerequisites

7. System add course section to student's Schedule

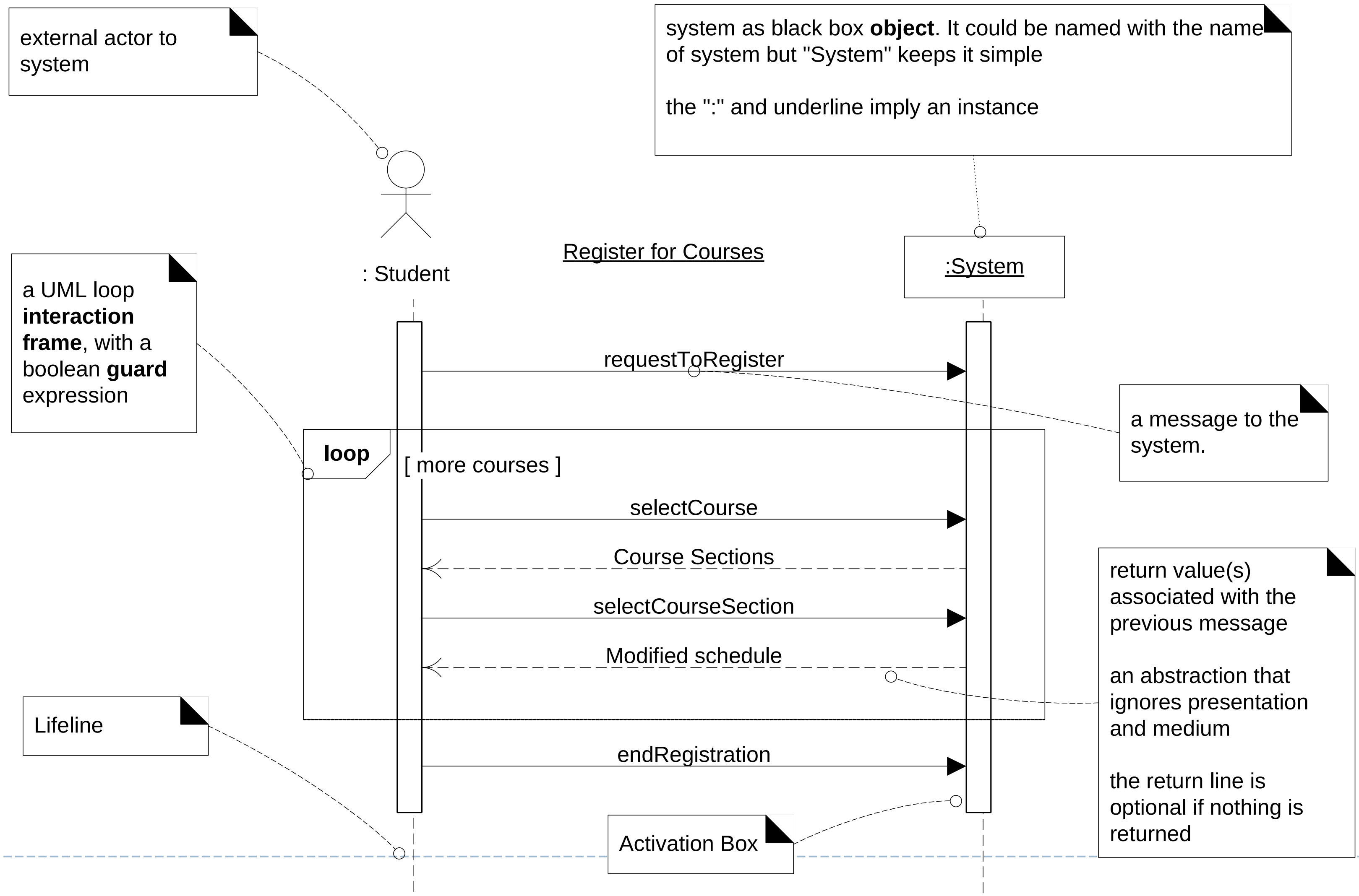
8. System displays modified student's Schedule

9. Steps 3-8 repeated until student finished



Based on System Sequence Diagrams
Larman, C. Applying UML and Patterns, 3rd Edition (Pearson Education, 2005)

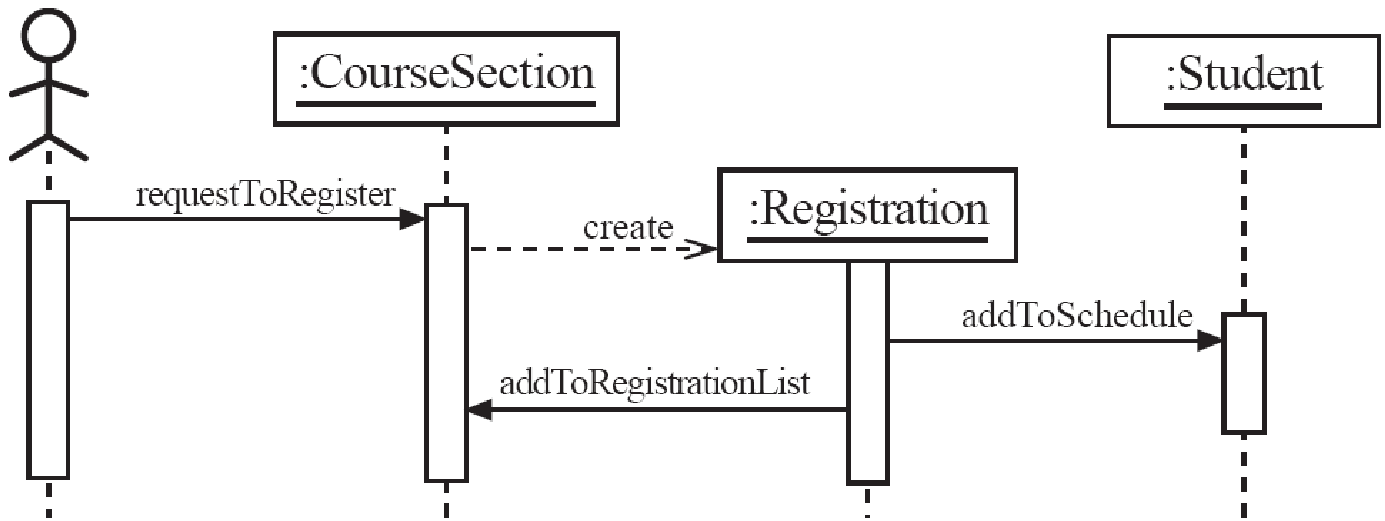
Sequence Diagrams – Elements

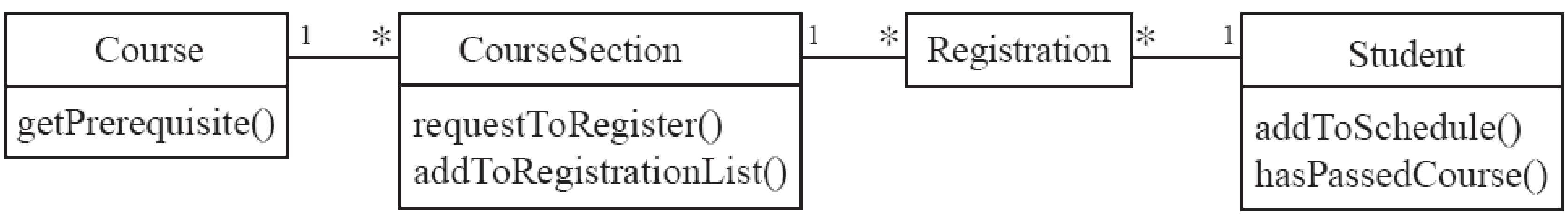


Sequence diagrams

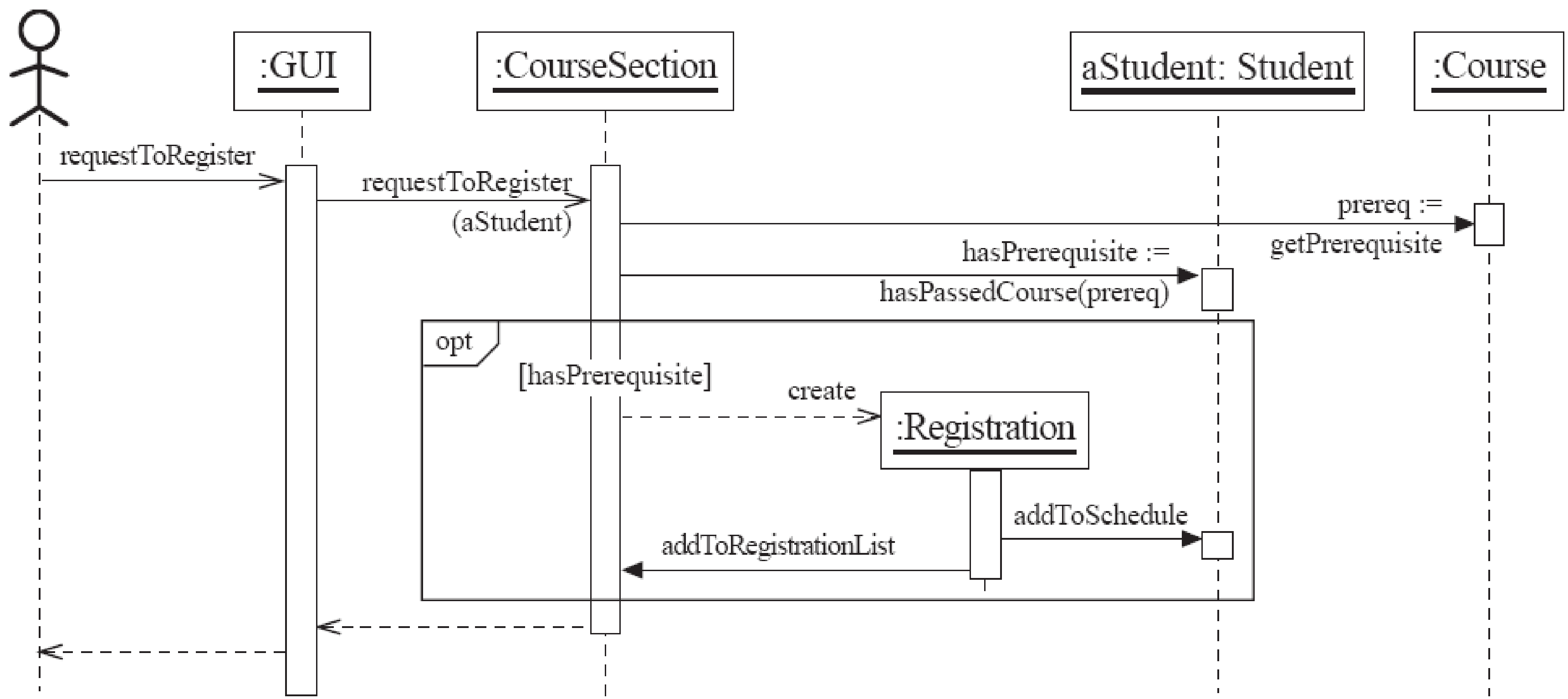
- A sequence diagram shows the sequence of messages exchanged by the set of objects performing a certain task
  - The objects are arranged horizontally across the diagram.
  - An actor that initiates the interaction is often shown on the left.
  - The vertical dimension represents time.
  - A vertical line, called a *lifeline*, is attached to each object or actor.
  - The lifeline becomes a broad box, called an *activation box* during the *live activation* period.
  - A message is represented as an arrow between activation boxes of the sender and receiver.
    - A message is labelled and can have an argument list and a return value.

Sequence diagrams – an example





Sequence diagrams – same example, more details



State diagrams

State diagrams

- A *state diagram* specifies the life histories of objects in terms of the sequences of operations that can occur in response to external stimuli.
  - For example, a state diagram can describe how an object responds to a request to invoke one of its methods.

- A state diagram describes behavior in terms of sequences of *states* that an object can go through in response to *events*.

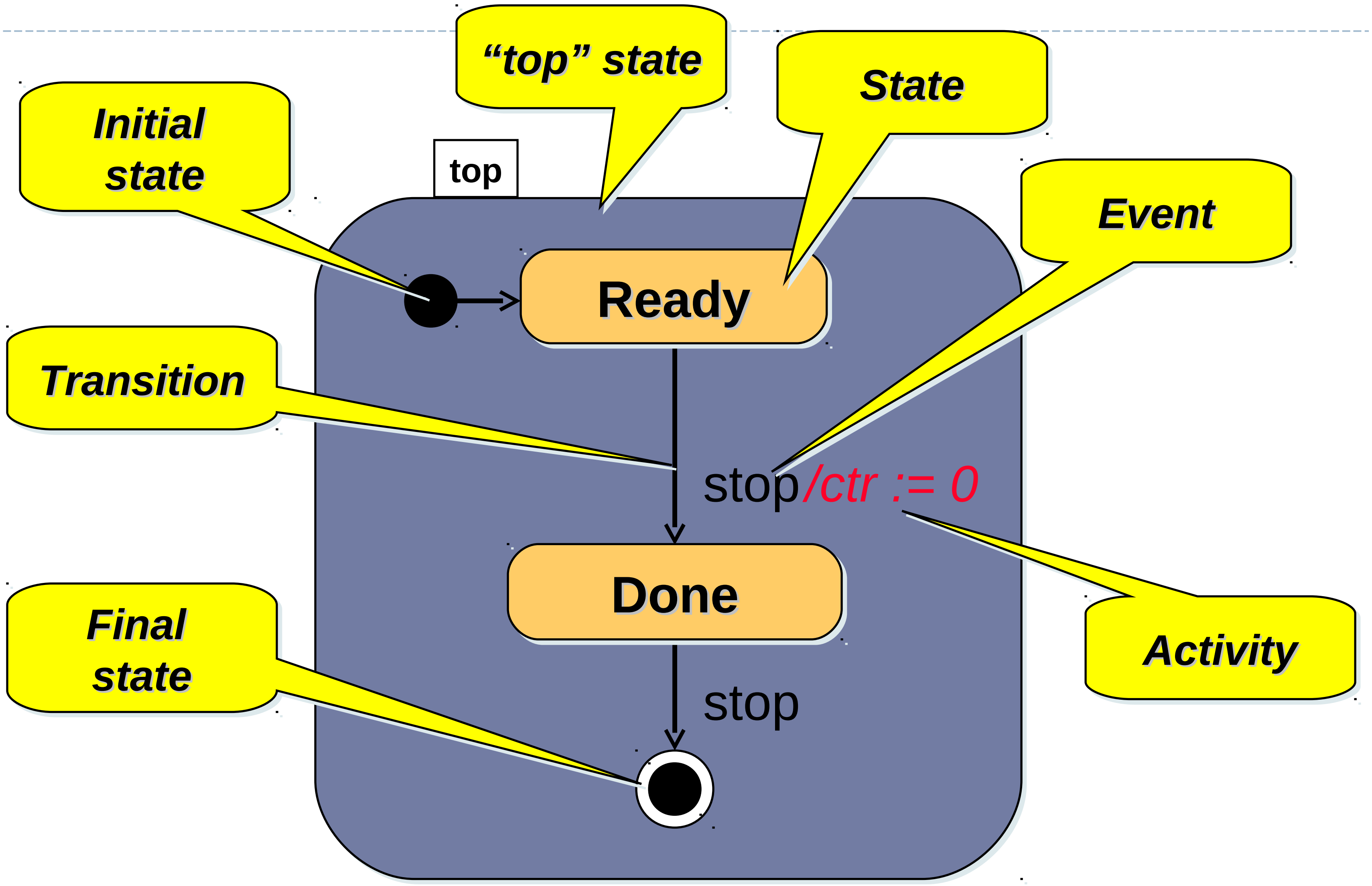
Key Concepts

- An *event* is a significant or noteworthy occurrence at a point in time.
  - Examples of events: sending a request to invoke a method, termination of an activity.
  - An event occurs instantaneously in the time scale of an application.
- A *state* is a condition of an object during its lifetime.
  - For example, a student is in the registered state after completing course registration.
  - A state is an abstraction of an object’s attribute values and links
    - For example, a bank account is in the Overdraft state when the value of its balance attribute is less than 0.

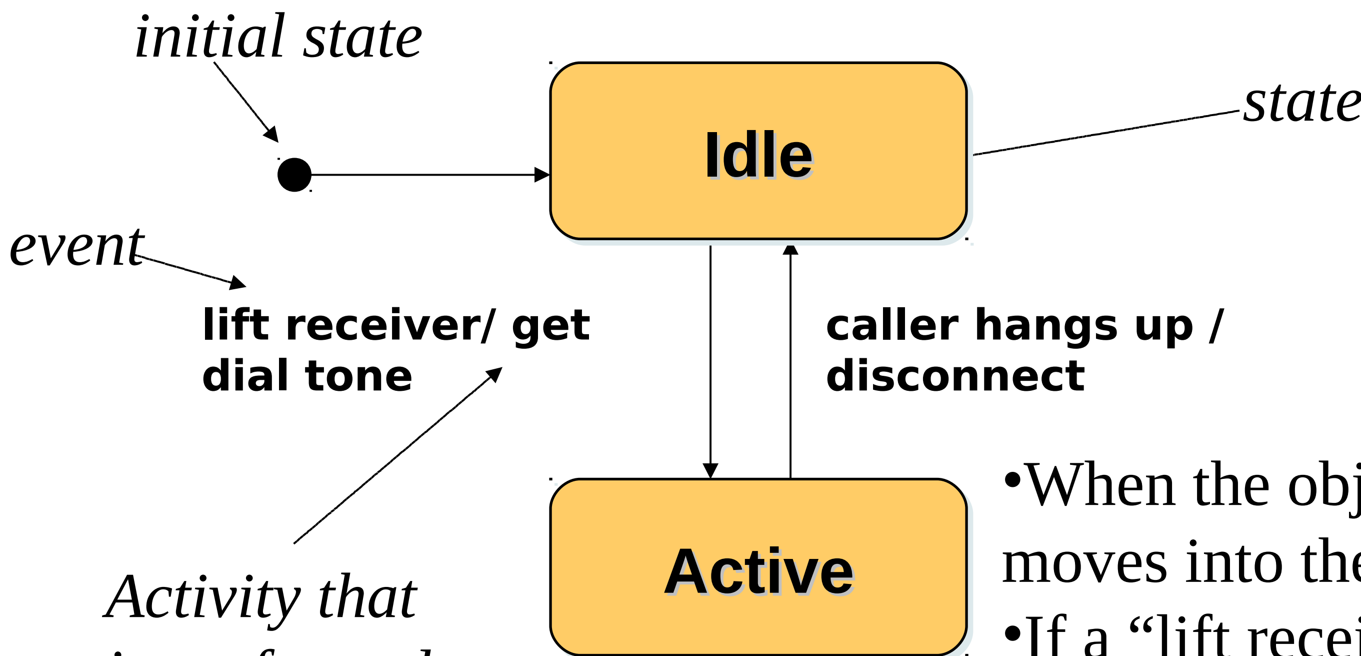
Key Concepts - 2

- A *transition* occurs when an event causes an object to change from its current (source) state to a target state.
  - For example, if a student is in the registered state and then drops out of the program then the student is in the "not registered" state.
  - The source and target states can be the same.
  - A transition is said to fire when the change from source to target state occurs.
- A *guard condition* on a transition is a boolean expression that must be true for a transition to fire
- An *activity* is a behavior that is executed in response to an event.

Basic UML State Diagram



Simple Example: Telephone Object

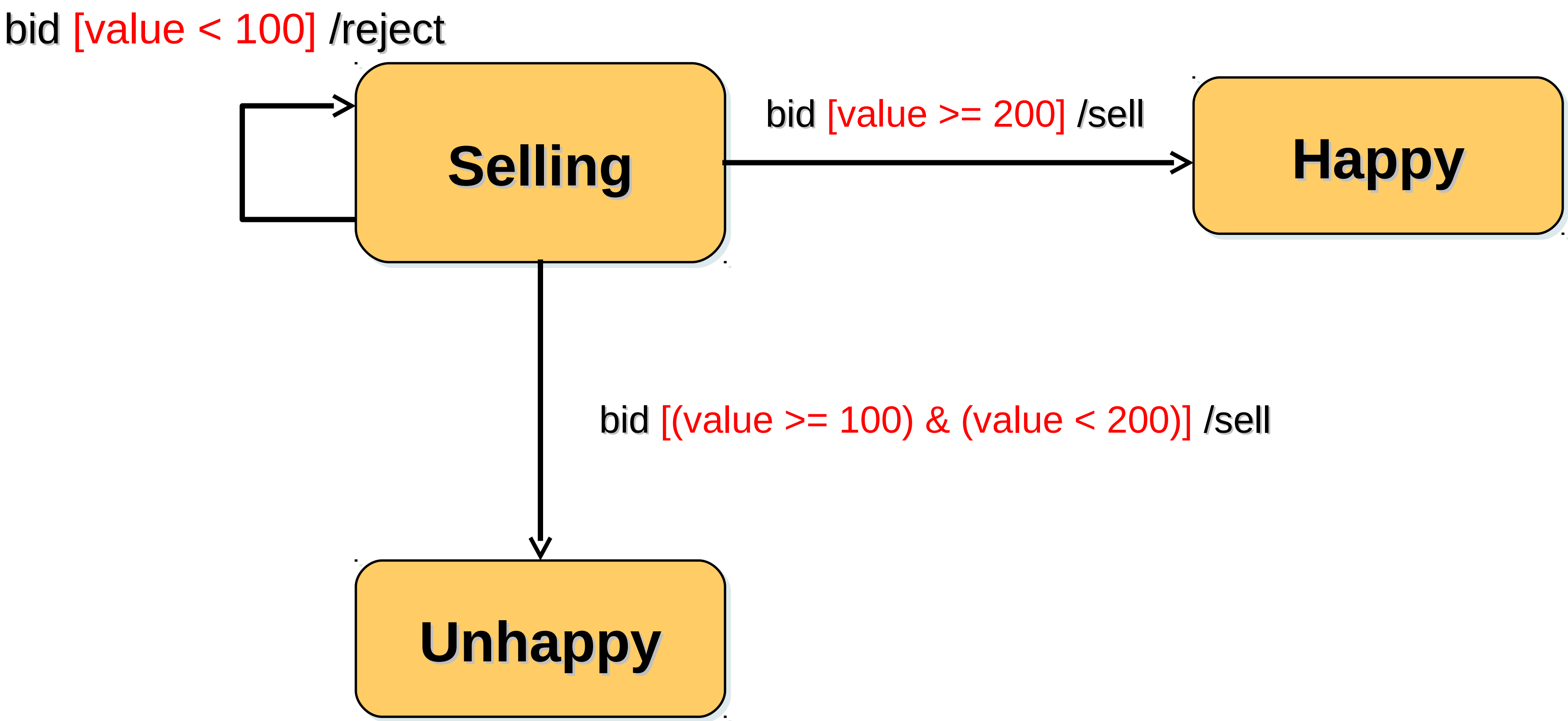


Activity that is performed during transition: it executes instantaneously

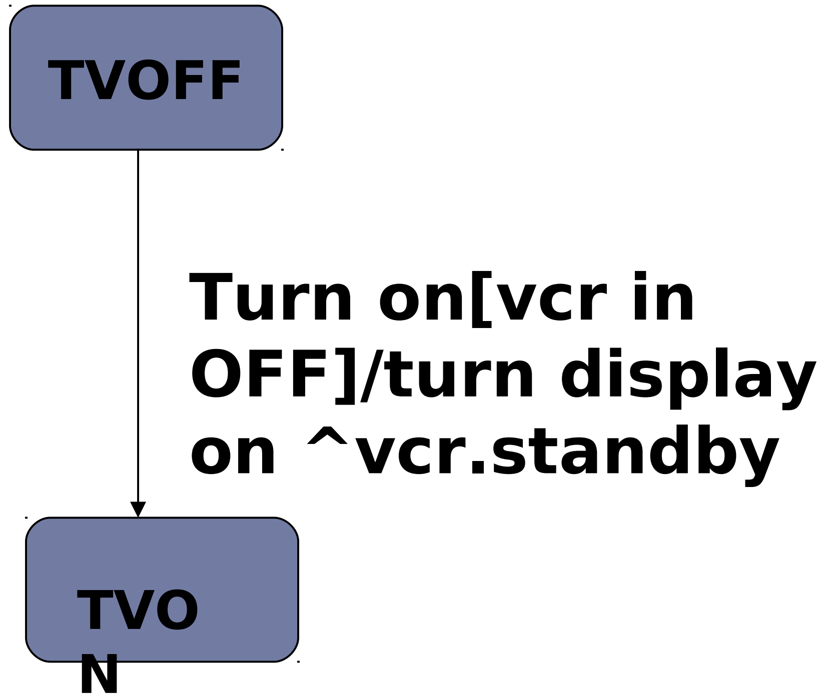
- When the object is created, it moves into the Idle state.
- If a "lift receiver" event is received the object moves from the Idle to the Active state. During the transition the "get dial tone" activity is executed.
- If a "caller hangs up" event occurs when in Active state, the object moves to the Idle state

Guards

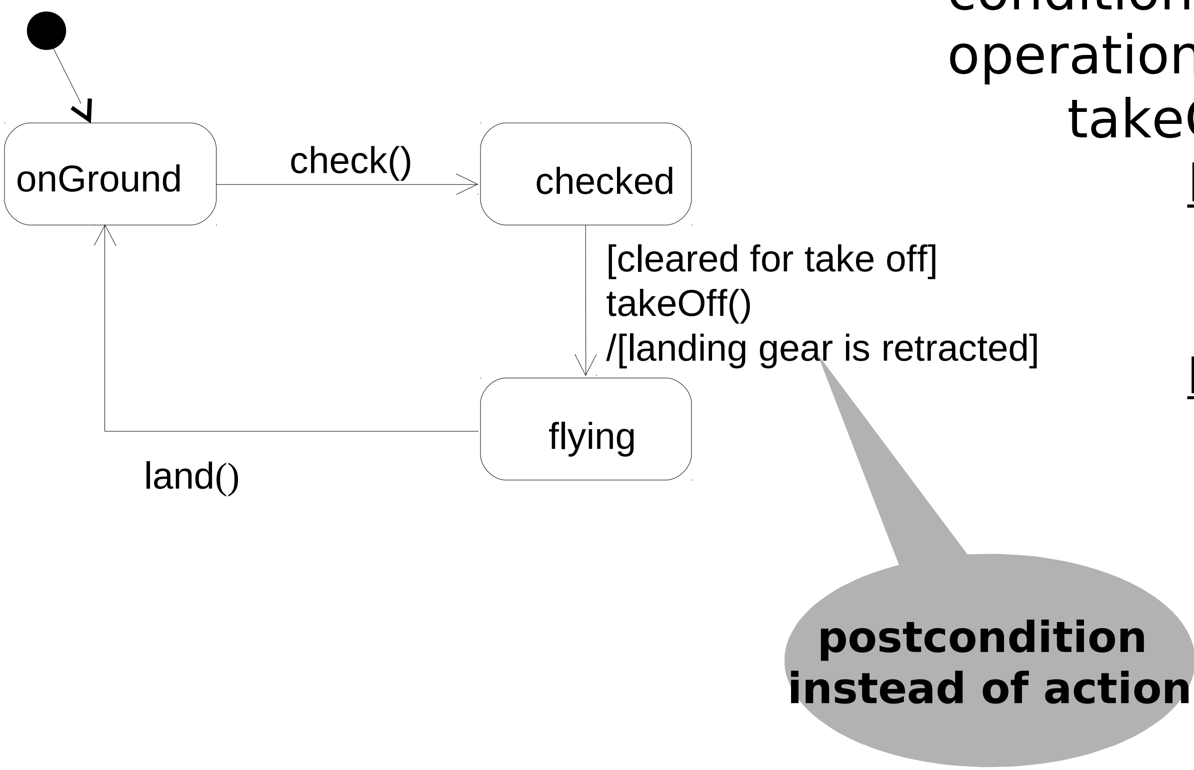
- Conditional execution of transitions
  - guards (Boolean predicates) must be side-effect free



TV - Example



Protocol State Machines



Equivalent to pre and post conditions added to the related operations:

takeOff()

<u>Pre</u>

-in state "checked"

-cleared for take off

<u>Post</u>

-landing gear is retracted

-in state "flying"

Another Example of a Protocol State Machine