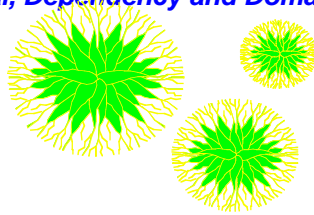


## ***XIX. Class Design***

***What is Class Design?  
Types of Design Classes  
Class Specifications and Interfaces  
Components, Sub-Systems and Packages  
Cohesion and Coupling  
Designing Associations  
Integrity Constraints  
Referential, Dependency and Domain Integrity***



## ***Class Design***

- Within the context of architectural design, class design:
  - ✓ Produces full definitions of classes, associations, algorithms and interfaces of operations;
  - ✓ Adds classes that will be useful during implementation;
  - ✓ Defines object interactions and object lifetimes in terms of interaction and state diagrams;
  - ✓ Optimises data structures and algorithms.

## ***Input/Output for Class Design***

- The input is assumed to consist of:
  - ✓ Use cases that describe the functional requirements for the new system; also sequence and state/activity diagrams that describe these use cases in more detail;
  - ✓ Class diagrams that describe the kinds of things the information system will be managing information about.
- The outputs of class design are:
  - ✓ Class packages which describe the overall software architecture of the new system;
  - ✓ Supporting sequence, state/activity diagrams that give additional details about the design.

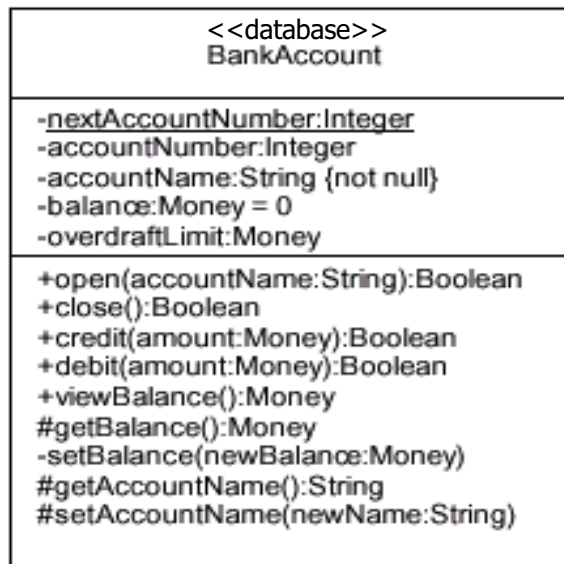
## ***Types of Design Classes***

- Most classes defined during requirements analysis represent objects about which information will be stored in the system database.
- During design, we distinguish four types of classes:
  - ✓ **Persistent database classes** (D), correspond to application classes and describe what will be stored persistently in the system database;
  - ✓ **Entity classes** (E) represent in-memory, run-time data structures for persistent database classes;
  - ✓ **Boundary classes** (B) specify interface functions;
  - ✓ **Control classes** (C) specify business logic functions.

## Class Specifications

- **Attribute signature**  
name: `:` type-expression `=` initial-value `{property-string}`
- **Operation signature**  
Operation name: `( parameter-list )`:` return-type-expression
- **Object Visibility**
  - ✓ **+ Public** -- The feature is directly accessible by any class;
  - ✓ **- Private** -- The feature may only be used by the class that includes it;
  - ✓ **# Protected** -- The feature maybe used by either the class that includes it or by a subclass of that class;

## An Example Database Class



## **An Example Entity Class**

<<entity>> Customer
name:String addr: String listOfAccts:List create()

Notes: `Customer` objects are created by accessing the `CustomerDB` and `BankAccounts` part of the database to build a single `Customer` object which collects all account information about a customer.

## **An Example Control Class**

<<control>> TransferAmount
transfer(acc1, acc2, amount)

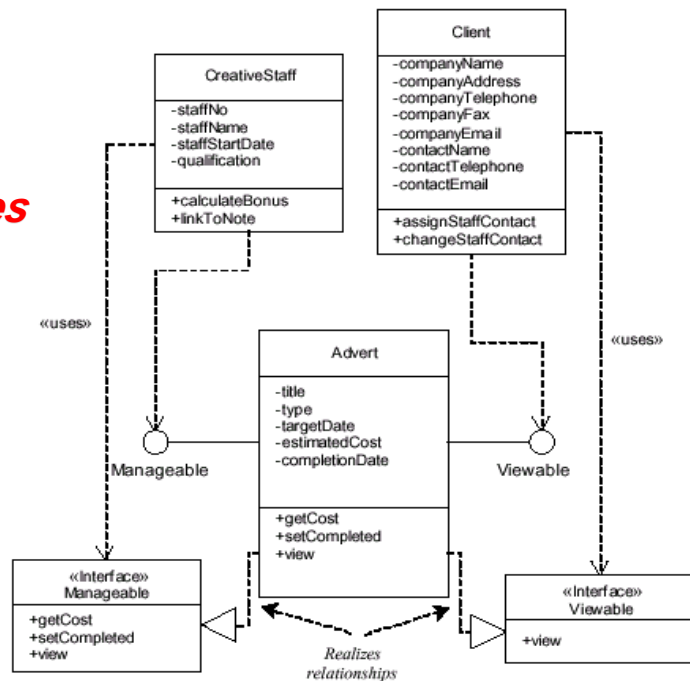
## **An Example Boundary Class**

Look at examples of view and control classes from the previous section.

## Class Interfaces

- An **interface** is a group of externally visible (public) operations.
- An interface is like a class, but contains no internal structure, has no attributes, no associations and no implementation of its operations.
- The **realizes** relationship indicates that the target class supports at least the operations listed in the interface

## Class Interfaces



## ***Class Design***

- So, a class design consists of a set of packages which contain classes and other packages, and which represent components or sub-systems.
- The grouping of classes into packages may be done from several different points of view:
  - ✓ By architectural tier -- Boundary, Control, Entity, Database;
  - ✓ By functional relationship -- MVC;
  - ✓ By authorship -- who designed what;
  - ✓ ...

## ***Cohesion and Coupling***

Criteria for good sub-system/package design:

- ***Coupling*** measures the degree of interconnectedness between design components/sub-systems/packages.
- The degree of coupling is reflected by the number of links a class has, and by the degree of interaction the class has with other classes.
- Low coupling is preferable in a design for many good reasons, e.g., easier to understand and modify the design.
- ***Cohesion***, on the other hand, measures the degree to which an element (subsystem, module, or class) contributes to a single purpose.
- Of course, we want a highly cohesive design.

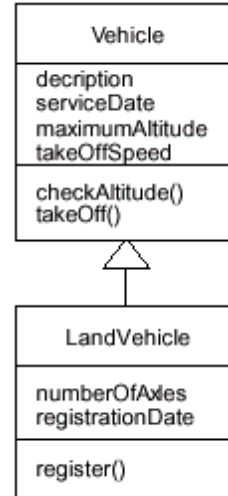
## Minimizing Coupling

### ■ Interaction coupling

- ✓ Measures the number of message types an object sends to other objects and the number of parameters passed with these message types;
- ✓ Should be kept to a minimum in order to reduce the possibility of changes rippling through object interfaces;

### ■ Inheritance coupling

- ✓ Degree to which a subclass actually needs the features (attributes or operations) it inherits;
- ✓ A subclass with unnecessary attributes or operations is more complex than it needs to be and instances of the subclass unnecessarily use up more memory.



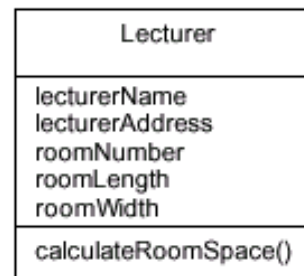
## Maximizing Cohesion

### ■ Operation cohesion

- ✓ Measure the degree to which an operation focuses on a single functional requirement.
- ✓ Good design produces highly cohesive operations, each of which deals with a single functional requirement.

### ■ Class cohesion

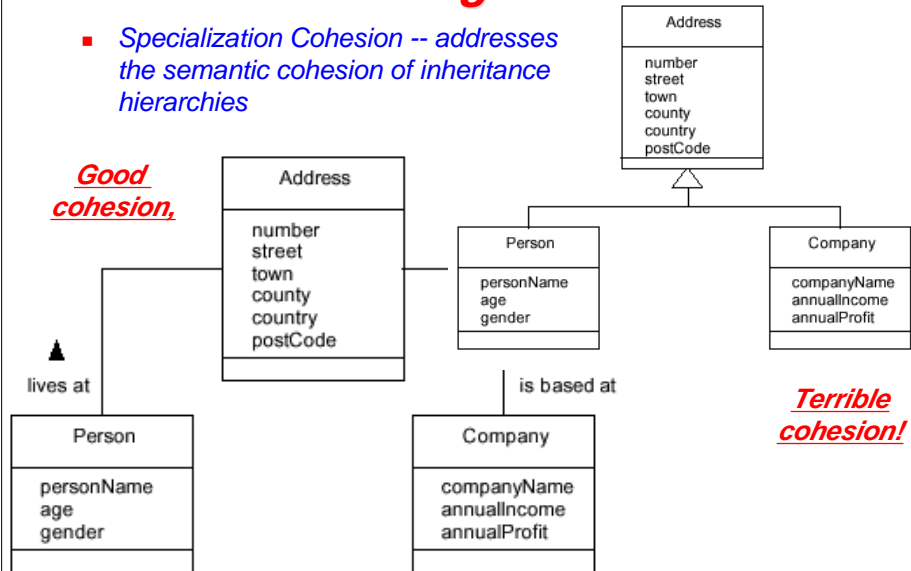
- ✓ Degree to which a class is focused on a single requirement.



**Good operation cohesion,  
...but lousy class cohesion**

## Maximizing Cohesion

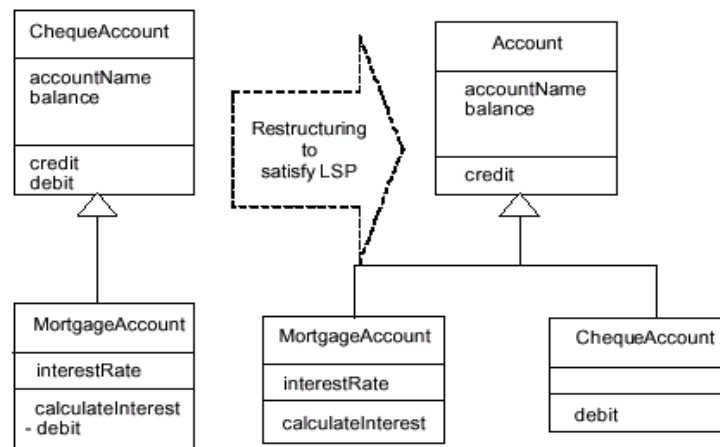
- *Specialization Cohesion -- addresses the semantic cohesion of inheritance hierarchies*



Class Design -- 15

## Liskov Substitution Principle

- *In class hierarchies, it should be possible to treat a specialized object as if it were a base object.*





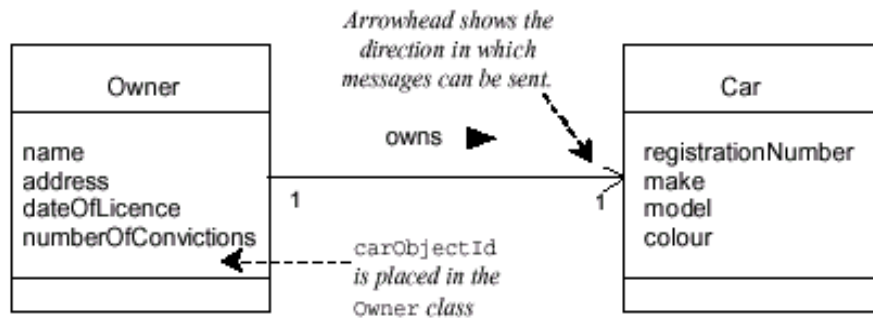
## More Design Principles

- **Clarity** -- A design should be easy to understand.
- **Do not over-design** -- Developers are tempted to produce designs that may not only satisfy current requirements but may also be capable of supporting a wide range of future requirements.
- **Inheritance hierarchies** -- Neither too deep nor too shallow!
- **Keep messages and operations simple**: Limit the numbers of parameters passed in a message; specify operations in no more than one page.
- **Design volatility** -- A good design should be stable in response to change in requirements; enforcing encapsulation is a key factor in producing stable systems.
- **Evaluation by scenario** -- Can be done with a role play based on use cases, using CRC cards.
- **Design by delegation**: A complex object should be decomposed into component objects forming a composition or aggregation

## Designing Associations

- Each association needs to be analysed to determine whether it should be a **one-way** or a **two-way** association.
- Depending on multiplicities, we may use collection classes (e.g., lists).
- Need to ask questions about object visibility:
  - ✓ does object A need to know object B's object-id?
  - ✓ does it need to communicate to third-party objects the object-id?

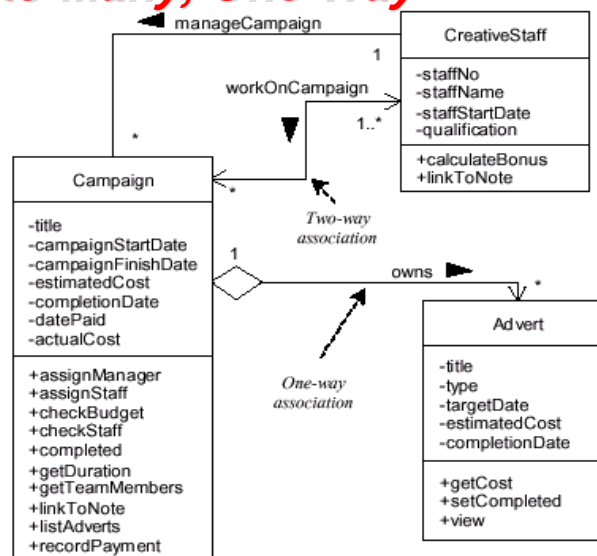
## Designing Associations One-to-One, One Way



- **Owner** needs to send messages to **Car** but not vice versa.
- Association may be implemented by placing an attribute to hold the object identifier for the **Car** class in **Owner** objects.

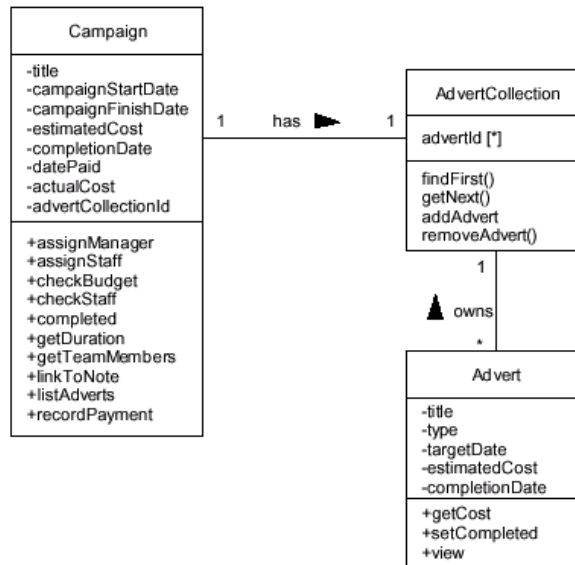
## Designing Associations: One-to-Many, One-Way

- The object identifiers could be held in a simple one-dimensional array in the Campaign object, but program code would have to be written to manipulate the array.



## Collection Classes

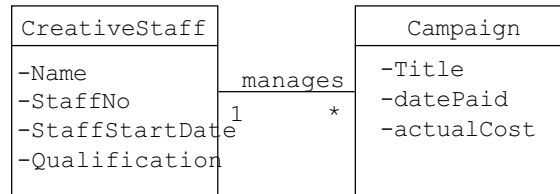
These are classes whose instances are lists, bags, or sets. Collection classes are useful for one-to-many associations



## Integrity Constraints

- We'll discuss three types of integrity constraints (...there are many others....)
- **Referential Integrity** ensures that an object identifier in an object actually refers to an object that exists.
- **Dependency Integrity** ensures that attribute dependencies are maintained, where one attribute may be calculated from other attributes.
- **Domain Integrity** ensures that attributes only hold permissible values

## Referential Integrity

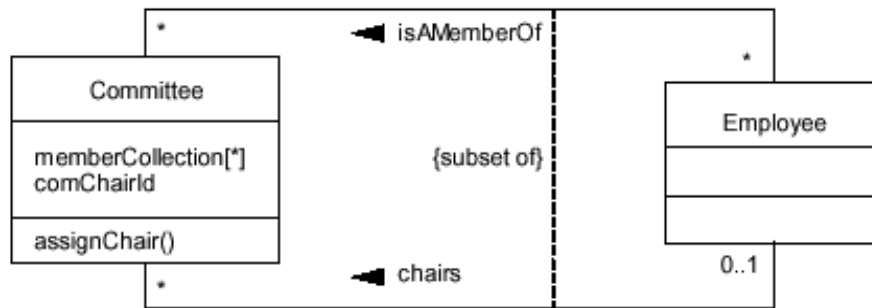


- A Campaign *must* have a CreativeStaff instance as its manager.
- What happens if the manager is deleted?
- Referential integrity is maintained by ensuring that the deletion of a CreativeStaff object that is a campaign manager **always** involves allocating a new campaign manager.

## Dependency Constraints: Derived Attributes

- The value of a derived attribute may be calculated from other attributes.
- For example, the total advertising cost can be calculated by summing the individual advert costs and storing the value in the attribute totalAdvertCost in the Campaign class or by calculating every time it is required.
- However, whenever the cost of an advert changes, or an advert is either added to or removed from a campaign the totalAdvertCost attribute has to be adjusted.
- This can be done by sending message adjustCost() to the Campaign object.

## Constraints Between Associations



- Enforced by placing a check in `assignChair()` to confirm that the `Employee` object identifier passed as a parameter is already in the collection class of committee members.

## Designing Operations

- Determine the best algorithm for the required function.
- Factors constraining algorithm design:
  - ✓ The cost of implementation;
  - ✓ Performance constraints;
  - ✓ Requirements for accuracy;
  - ✓ The capabilities of the chosen platform.
- Factors to be considered when choosing among alternative algorithm designs
  - ✓ The computational complexity of candidate algorithms;
  - ✓ Ease of implementation and understandability;
  - ✓ Flexibility;
  - ✓ Fine-tuning the object model.