Object Design

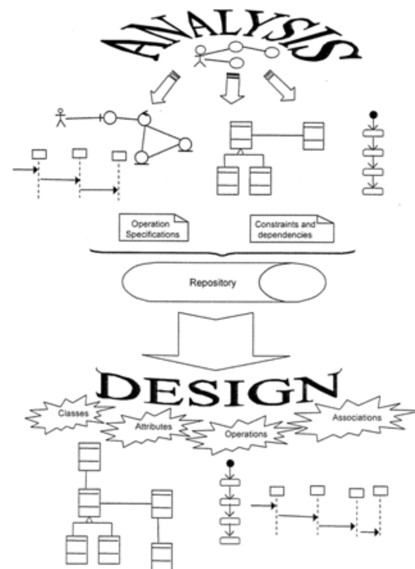
Chapter 14

In this Lecture you will Learn:

- How to apply criteria for good design
- How to design associations
- The impact of integrity constraints on design
- How to design operations
- The role of normalization in object design

Information Sources for Object Design

- Object Design is concerned with the detailed design of the objects and their interactions.
- Object Design is particularly concerned with the specification of the attribute type, how operations function and how objects are linked to other objects.



Class Specification: Attributes

- An attribute's data type is declared in UML using the following syntax
  - name ':' type-expression '=' initial-value '{'property-string'}'
- Where
  - ***name*** is the attribute name
  - ***type-expression*** is its data type
  - ***initial-value*** is the value the attribute is set to when the object is first created
  - ***property-string*** describes a property of the attribute, such as constant or fixed

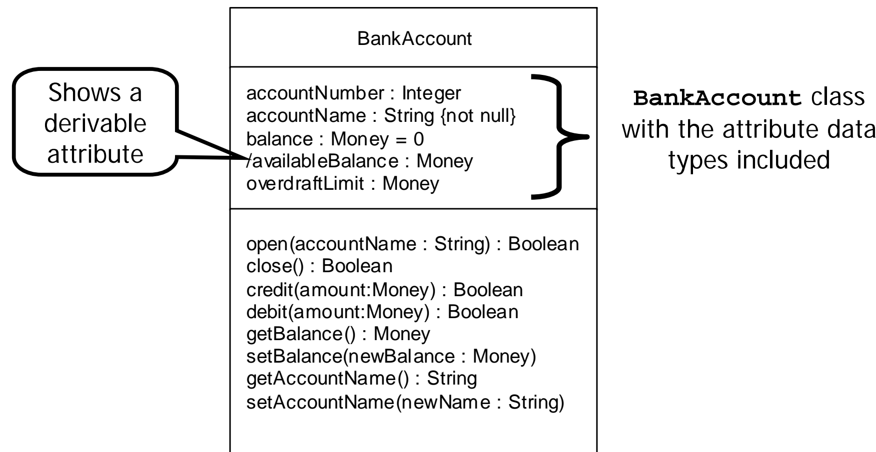
Class Specification: Attributes

- Common primitive data types include Boolean (true or false), Character (any alphanumeric or special character), Integer (whole numbers) and Floating-Point (decimal numbers).
- In most object-oriented languages more complex data types, such as Money, String, Date, or Name can be constructed from the primitive data types or may be available in standard libraries.
- The attribute name is the only feature of its declaration that is compulsory.

Attributes Example

- The attribute balance in a BankAccount class might be declared with an initial value of zero using the syntax
  - balance: Money = 0.00
- Attributes that may not be null are specified
  - accountName: String {not null}
- Arrays are specified
  - qualification[0..10]: String

Class Diagram Notation for Attribute



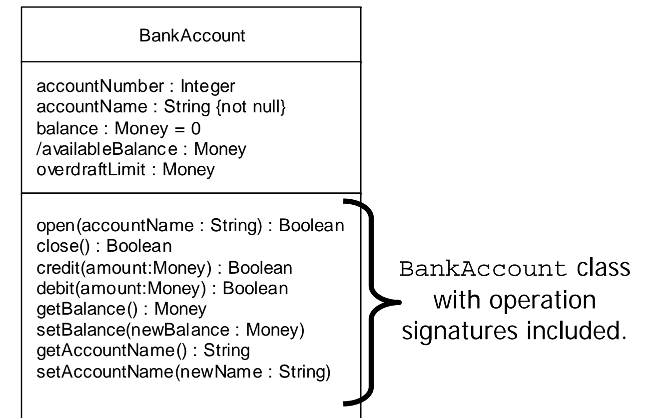
Class Specification: Operations

- The syntax used for an operation is
  - operation name ‘(’parameter-list ‘)’‘:’ return-type-expression
- An operation’s **Signature** is determined by the operation’s name, the number and type of its parameters and the type of the return value if any

Operation Example

- The BankAccount class might have a credit() operation that passes the amount being credited to the receiving object and has a Boolean return value
  - credit(amount: money): Boolean
- A credit() message sent to a BankAccount object could have the format
  - creditOK = accObject.credit(500.00)

Class Diagram Notation for Operation



Which Operations will Show?

- **Primary Operations** are the create, destroy, get and set operation.
- Generally don't show primary operations
- Only show constructors where they have special significance
- Varying levels of detail at different stages in the development cycle

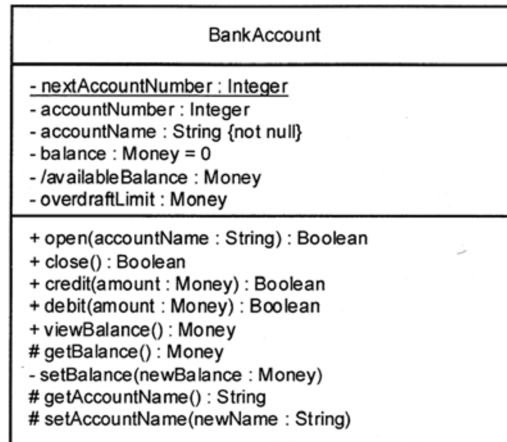
Visibility

- An attribute or operation may be assigned one of three levels of visibility i.e. public (+), private (-), protected (#) or package (~).

Visibility symbol	Visibility	Meaning
+	Public	The feature (an operation or an attribute) is directly accessible by an instance of any class.
-	Private	The feature may only be used by an instance the class that includes it.
#	Protected	The feature may be used either by the class that includes it or by a subclass or descendant of that class.
~	Package	The feature is directly accessible only by instances of a class in the same package.

Class Diagram Notation for Visibility

- BankAccount class with visibility specified



Class Exercise

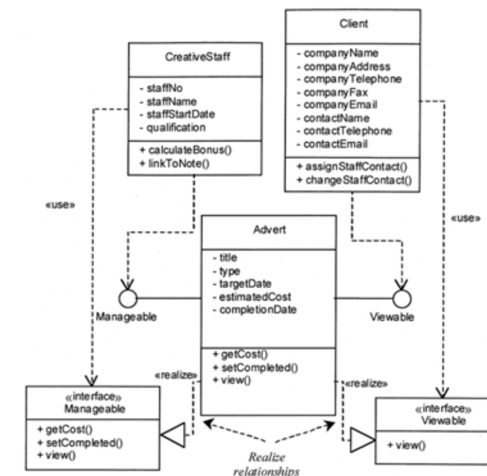
- Why should attributes be private?

Interfaces

- UML supports two notations to show interfaces
  - The small circle icon showing no detail
  - A stereotyped class icon with a list of the operations supported
- Normally only one of these notations is used on any one diagram
- The realize relationship, represented by the dashed line with a triangular arrowhead, indicates that the client class (e.g. Advert) supports at least the operations listed in the interface

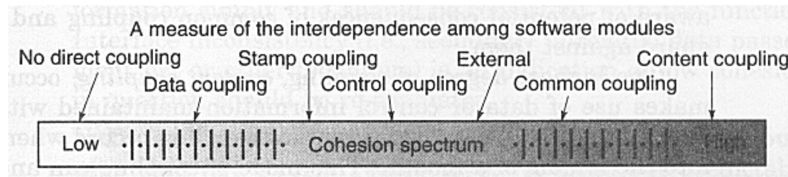
Class Diagram Notation for Interfaces

- Interfaces for the Advert class



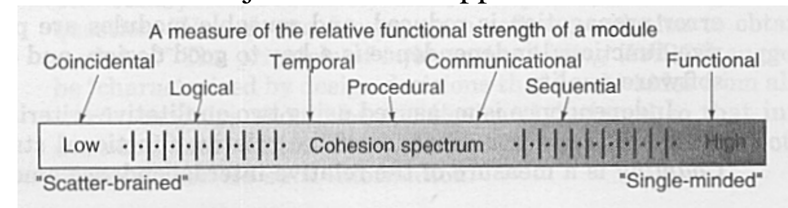
Criteria for Good Design: Coupling

- **Coupling** describes the degree of interconnectedness between design components
- It is reflected by the number of links an object has and by the degree of interaction the object has with other objects



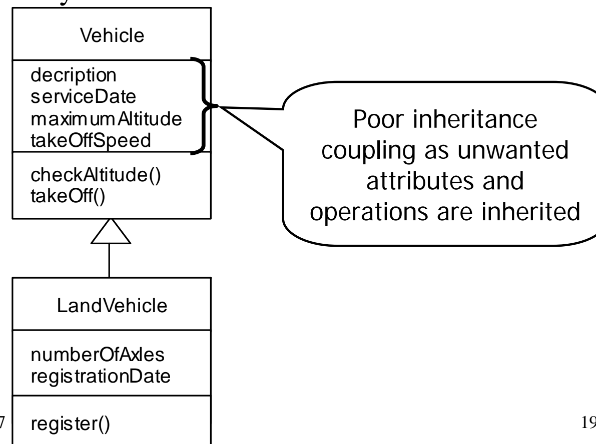
Criteria for Good Design: Cohesion

- **Cohesion** is a measure of the degree to which an element contributes to a single purpose
- The concepts of coupling and cohesion are not mutually exclusive but actually support each other
- Coad and Yourdon (1991) suggested several ways in which coupling and cohesion can be applied within an Object-Oriented approach



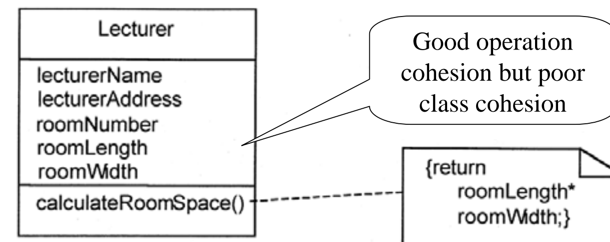
Inheritance Coupling

- **Inheritance Coupling** describes the degree to which a subclass actually needs the features it inherits from its base class



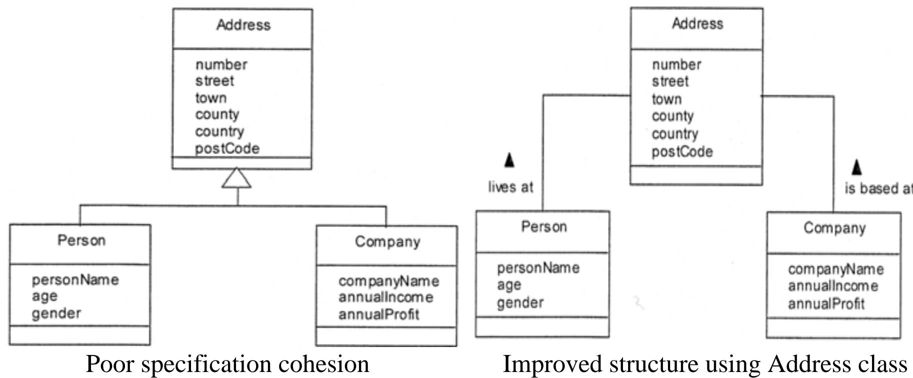
Operation Cohesion

- **Operation Cohesion** measures the degree to which an operation focuses on a single functional requirement.
- Good design produces high cohesive operations, each of which deal with a single functional requirement.



Specialization Cohesion

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



Poor specification cohesion   Improved structure using Address class

Class Exercise

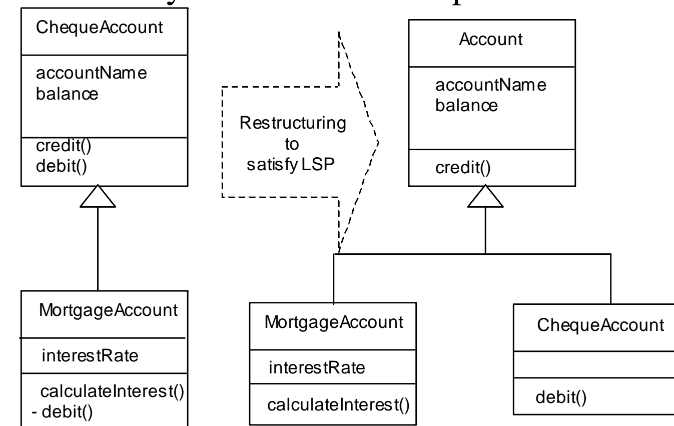
- How does the application of the concepts of coupling and cohesion help to produce object-oriented designs?

Liskov Substitution Principle (LSP)

- Liskov Substitution Principle (LSP) is another design criterion that is applicable to inheritance hierarchies.
- Essentially the principle states that, in object interactions, it should be possible to treat a derived object as if it were a base object without integrity problems
- If the principle is not applied then it may be possible to violate the integrity of the derived object

Liskov Substitution Principle Example

- Disinheritance of debit() means that the left-hand hierarchy is not Liskov compliant



Advantages of Applying LSP

- The production of inheritance structures that exhibit high levels of inheritance coupling.
- The resulting classes in the inheritance structure are cohesive.
- There is no anomalous behavior when base class operations are applied to derived class objects and as a result the program code is much safer.

Disadvantages of Applying LSP

- The resulting inheritance hierarchy may not map precisely onto problem domain objects. (It is a moot point whether this is a disadvantage or not though it does complicate traceability from the problem domain to the implementation to some extent.)
- If base classes are not chosen carefully extending an inheritance hierarchy may require its complete restructuring to ensure Liskov compliance and this may be result in significant changes to programme code. For this reason it may not always be cost effective to apply LSP (again a debatable point) but when LSP is not enforced the resulting structures are likely to be less safe. Many developers take a pragmatic view to applying LSP because of this contention between the perception that development and maintenance costs are higher and the development of systems that are safe because maintaining the integrity of derived classes is easier with LSP.

Further Design Guidelines

- Coad and Yourdon (1991) and Yourdon (1994) suggest further design guidelines that are included in the list below:
  - Design Clarity
  - Don't Over-Design
  - Control Inheritance Hierarchies
  - Keep Messages and Operations Simple
  - Design Volatility
  - Evaluate by Scenario
  - Design by Delegation
  - Keep Classes Separate

Design Clarity

- A design should be made as easy to understand as possible.
- This reinforces the need to use design standards or protocols that have been specified.

Don’t Over-Design

- Developers are on occasions tempted to produce designs that may not only satisfy current requirements but may also be capable of supporting a wide range of future requirements.
- Designing flexibility (or any other non-functional requirement) into a system has a cost, the system may take longer to design and construct but this may be offset in the future by easier and less expensive modification.
- However, it is not feasible to design for every eventuality.
- Systems that are over-designed in first instance are more difficult to extend if the modifications are not sympathetic to the existing structure.

Control Inheritance Hierarchies

- Inheritance hierarchies should be neither too deep nor too shallow.
- If a hierarchy is too deep it is difficult for the developer to understand easily what features are inherited.
- There is a tendency for developers new to O-O to produce over-specialized hierarchies, thus adding complexity rather than reducing it.
- Yourdon (1994) suggests that hierarchies of up to nine levels are manageable, whereas Rumbaugh et al. (1991) suggest that more than about four or five levels is too many.

Keep Messages and Operations Simple

- In general it is better to limit the number of parameters passed in a message to no more than three (of course, a single parameter might be a complete object).
- Ideally an operation should be capable of specification in no more than one page.

Design Volatility

- A good design will be stable in response to changes in requirements.
- It is reasonable to expect some change in the design if the requirements are changed.
- However, any change in the design should be commensurate with the change in requirements.
- Enforcing encapsulation is a key factor in producing stable systems.

Evaluate by Scenario

- An effective way of testing the suitability of a design is to role play it against the use cases using CRC cards if appropriate.

Design by Delegation

- A complex object should be decomposed (if possible) into component objects forming a composition or aggregation.
- Behavior can then be delegated to the component objects producing a group of objects that are easier to construct and maintain.
- This approach also improves reusability.

Keep Classes Separate

- In general, it is better not to place one class inside another.
- The internal class is encapsulated by the other class and cannot be accessed independently.
- This reduces the flexibility of the system.

Designing Associations

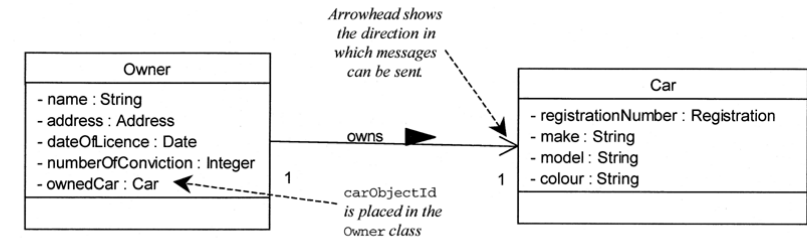
- Determine direction of message passing
  - i.e. the navigability of the association
- In general an association between two classes A and B should be considered with the questions
  - Do objects of class A have to send messages to objects of class B?
  - Does an A object have to provide some other object with B object identifiers?
- If yes then A needs Bs object identifier

Designing Associations

- An association that has to support message passing in both directions is a two-way association
- A two-way association is indicated with arrowheads at both ends
- Minimizing the number of two-way associations keeps the coupling between objects as low as possible

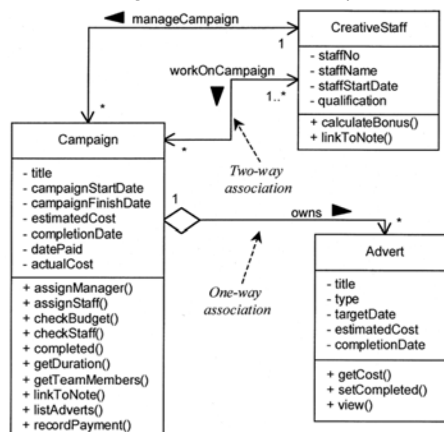
One-to-One Association

- Example for the One-way One-to-One association



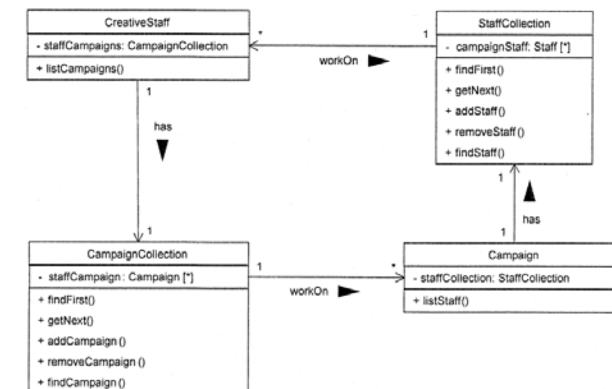
One-to-Many Association

- Example for the Fragment of Class Diagram for the Agate Case Study



Many-to-Many Associations

- This is the design for the works On Campaign association

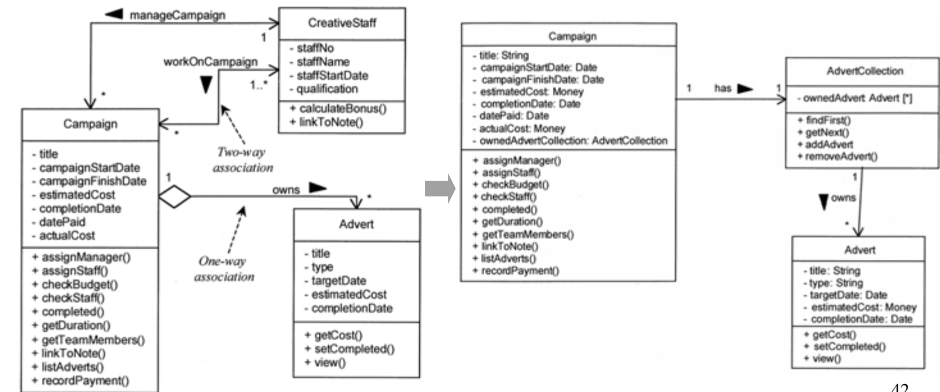


Collection Classes

- **Collection classes** are used to hold the object identifiers when message passing is required from one to many along an association
- OO languages provide support for these requirements. Frequently the collection class may be implemented as part of the sending class (e.g. Campaign) as some form of list

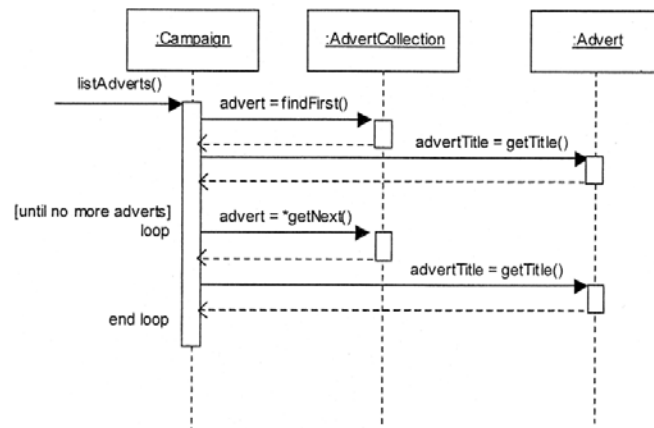
Collection Classes – Class Diagram

- Example of the One-to-Many Association using a Collection Class



Collection Classes – Sequence Diagram

- This sequence diagram shows the interaction when using a collection class

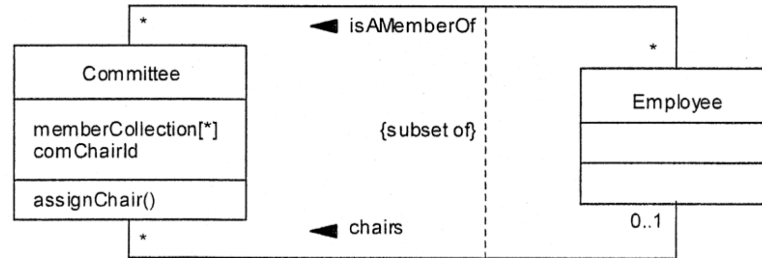


Integrity Constraints

- **Referential Integrity** that ensures that an object identifier in an object is actually referring to an object that exists. It is maintained by including appropriate checks in object constructors and destructors. These checks would ensure that object references were either null (if the multiplicity constraints permitted this) or that the object reference refers to an object that exists.
- **Dependency Constraints** that ensures that attribute dependencies, where one attribute may be calculated from other attributes, are maintained consistently
- **Domain Integrity** that ensures that attributes only hold permissible values

Constraints between Associations

- Dependency constraints can also exist between or among association.



Designing Operations

- Various factors constrain algorithm design:
  - The cost of implementation
  - Performance constraints
  - Requirements for accuracy
  - The capabilities of the implementation platform

Designing Operations

- Factors that should be considered when choosing among alternative algorithm designs
  - Computational complexity
  - Ease of implementation and understandability
  - Flexibility
  - Fine-tuning the object model

Normalisation

- Normalization may be useful in OO approaches
  - When implementing the system with a relational database management
  - As a guide to decomposing a large, complex (and probably not very cohesive) objects
- Objects need not be normalised but it is important to remove redundancy