Class Specification

Association Specification

Class Specification (1)

- name’:’type-expression ‘=‘initial-value ‘{‘property-string‘}’
- operation name‘(‘parameter-list’)’ ’:’return-type-expression
- Primary operations: create, destroy, get and set

BankAccount
accountNumber : Integer accountName : String {not null} balance : Money = 0 /availableBalance : Money overdraftLimit : Money
open(accountName : String) : Boolean close() : Boolean credit(amount:Money) : Boolean debit(amount:Money) : Boolean getBalance() : Money setBalance(newBalance : Money) getAccountName() : String setAccountName(newName : String)

Class Specification (2)

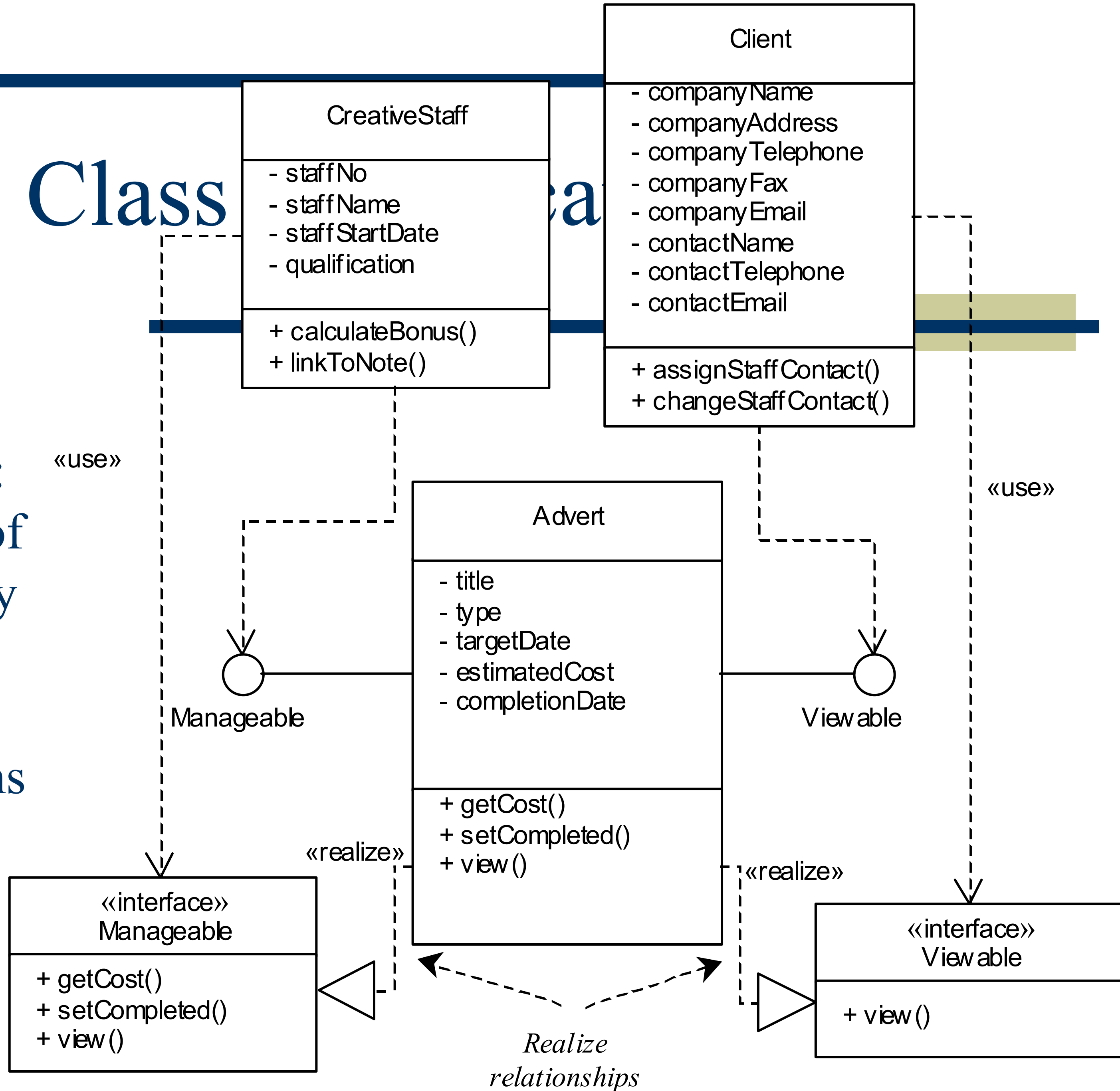
- Visibility: what is publicly accessible
  - May be specified as properties

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 decendant of that class.
~	Package	The feature is directly accessible only by instances of a class in the same package.

BankAccount
- nextAccountNumber: Integer - accountNumber: Integer - accountName: String {not null} - balance: Money = 0 - /availableBalance: Money - overdraftLimit: Money
+ open(accountName: String): Boolean + close(): Boolean + credit(amount: Money): Boolean + debit(amount: Money): Boolean + viewBalance(): Money # getBalance(): Money - setBalance(newBalance: Money) # getAccountName(): String # setAccountName(newName: String)

Class

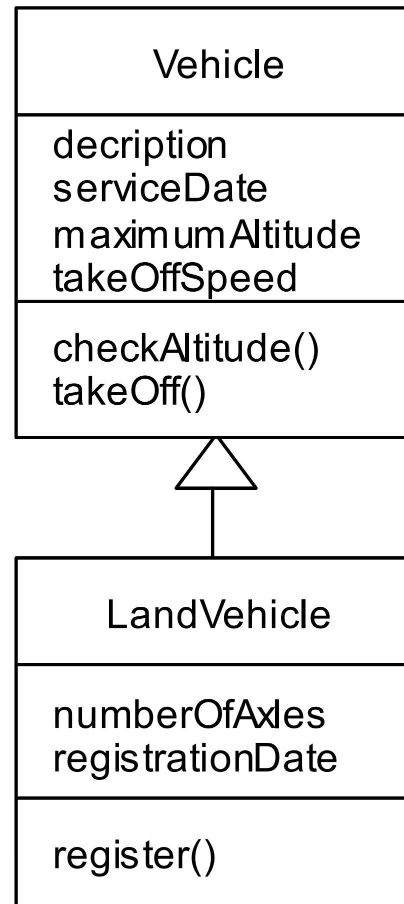
- UML interface: a group of externally visible (public) operations



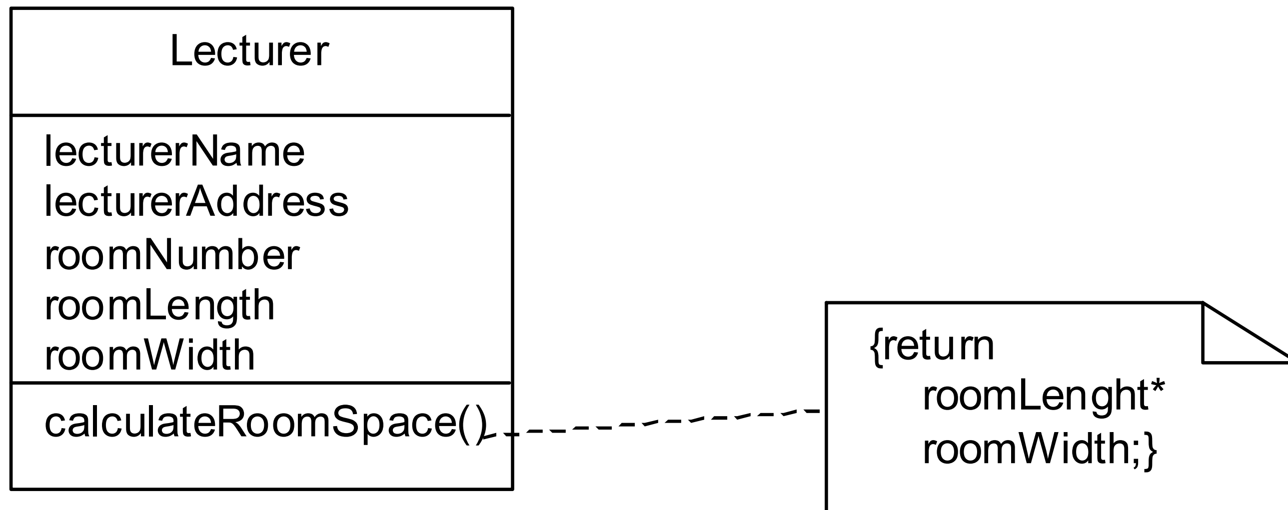
Criteria for Good Design (1)

- Coupling and cohesion
  - Degree of interconnectedness between design components – number of links and degree of interaction
  - Degree to which an element contributes to a single purpose
- Interaction coupling – low
- Inheritance coupling – high
- Operation cohesion – high
- Class cohesion – high
- Specialisation cohesion – high

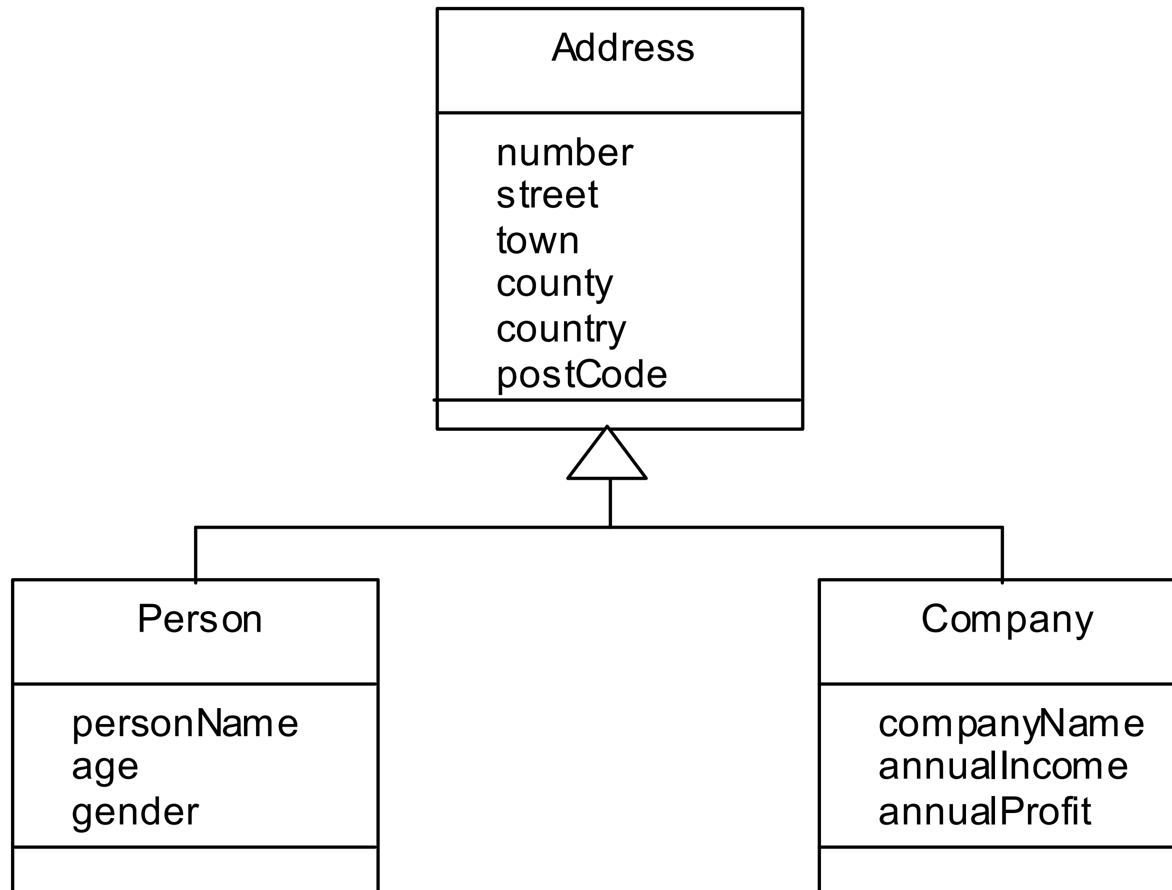
Criteria for Good Design (2)



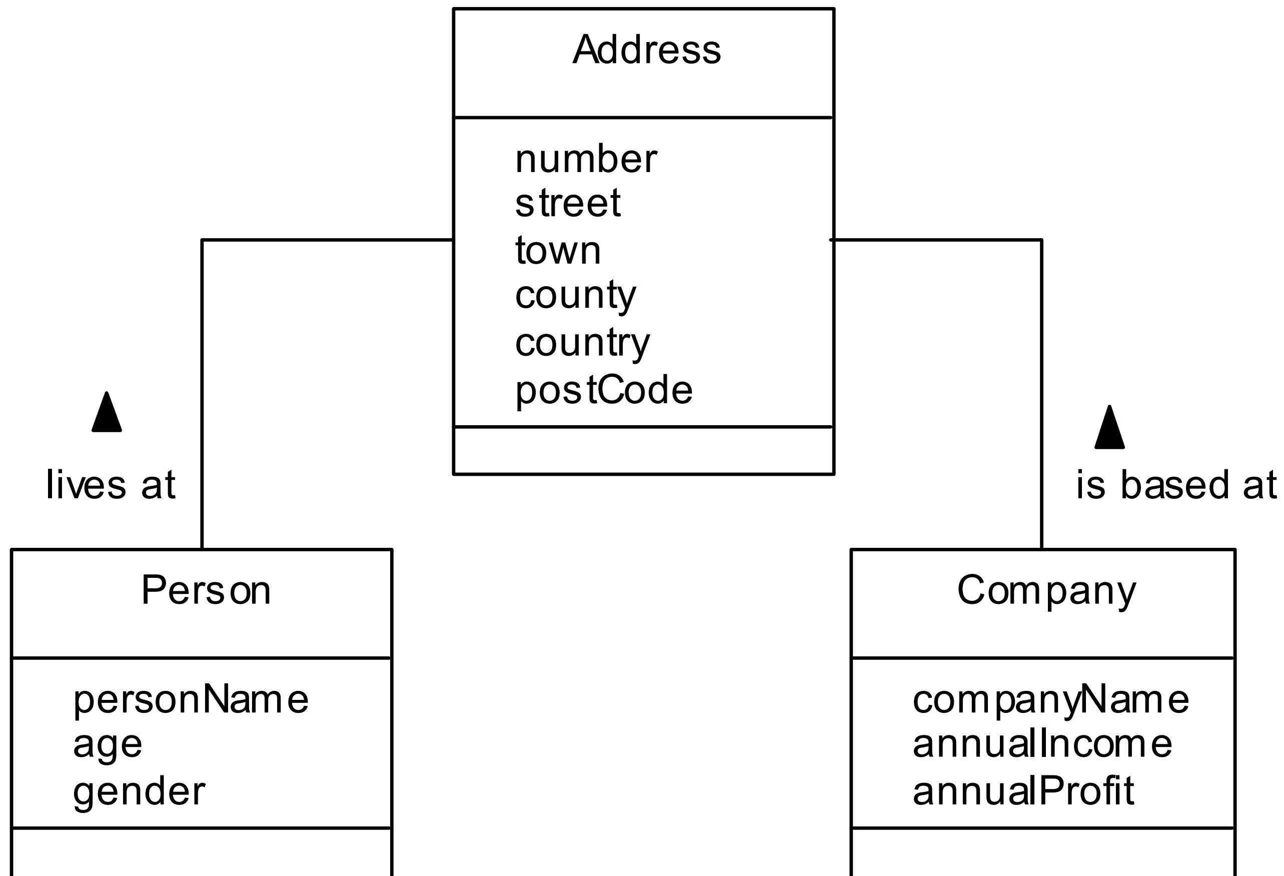
Criteria for Good Design (2)



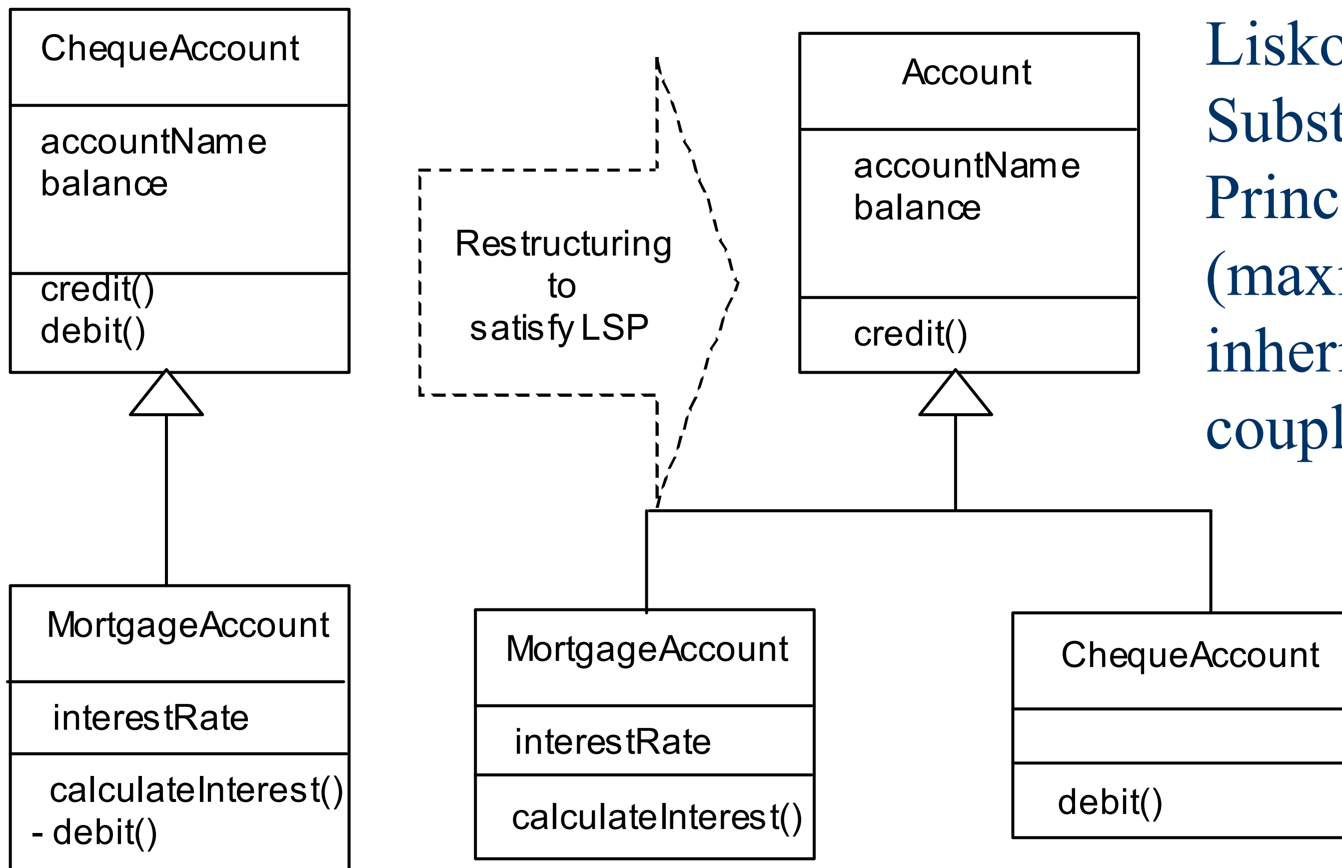
Criteria for Good Design (2)



Criteria for Good Design (2)



Criteria for Good Design (3)



Liskov Substitutability Principle (maximising inheritance coupling)

Criteria for Good Design (4)

- Design guidelines
  - Design clarity
  - Don’t over design (designing flexibility has a cost)
  - Control inheritance hierarchies (4 or 5 levels)
  - Keep messages and operations simple
  - Design volatility
  - Evaluate by scenario
  - Design by delegation
  - Keep classes separate

Generalisation and Inheritance (1)

- Generalisation and inheritance are not the same!
  - Generalisation is a semantic relationship between classes – superclass and subclass have the same interface
    - Substitution principle is central
    - Substitution leads to reduction of associations in the class diagram
  - Inheritance is the mechanism by which subclasses incorporate the structure and behaviour of their superclass
    - Inheritance may defeat substitutability!
    - Inheritance compromises encapsulation – protected features
    - Inheritance is a class concept – except in Smalltalk

Generalisation and Inheritance (2)

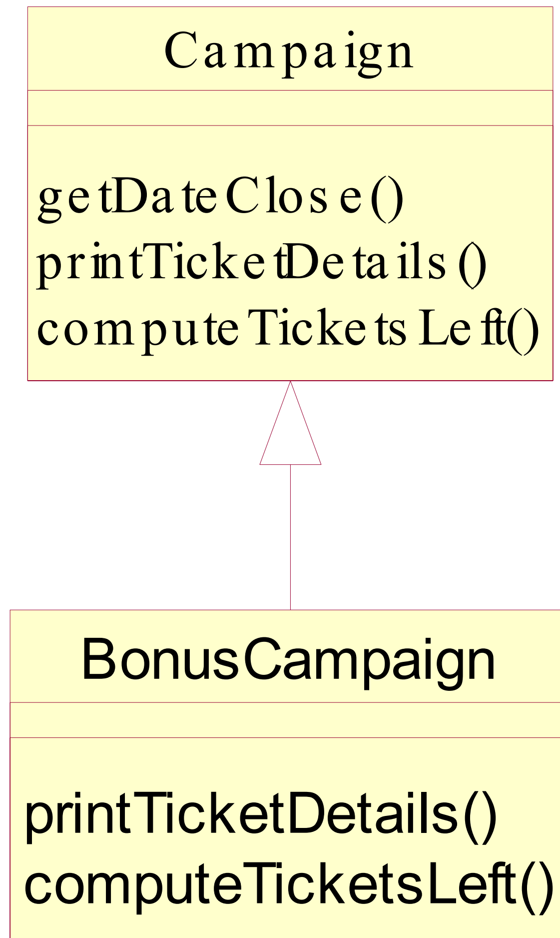
- Interface inheritance (subtyping, type inheritance)
  - Harmless
  - Abstract classes for interface declaration
- Implementation inheritance (subclassing, code inheritance, class inheritance)
  - Can be dangerous!
  - Code reuse and polymorphism
  - Overriding – up calls
  - Extension inheritance (proper) – incremental class definition
    - Overriding – more specific but with the same meaning

Generalisation and Inheritance (3)

- Restriction inheritance (problematic) – reuse of class properties
  - Maintenance problems
- Convenience inheritance (improper)
  - Extensive overriding

- Problems with implementation inheritance
  - Fragile base class – allow evolution of parent classes
    - Immutable public interfaces?
  - Overriding and callbacks (up calls)
    - Inherit interface and implementation without changes in the implementation
    - Inherit code and call it within own method with unchanged signature
    - Inherit code and and completely override it maintaining the signature
    - Inherit empty code declaration and provide an implementation
    - Inherit the method interface and provide an implementation

Generalisation and Inheritance (4)



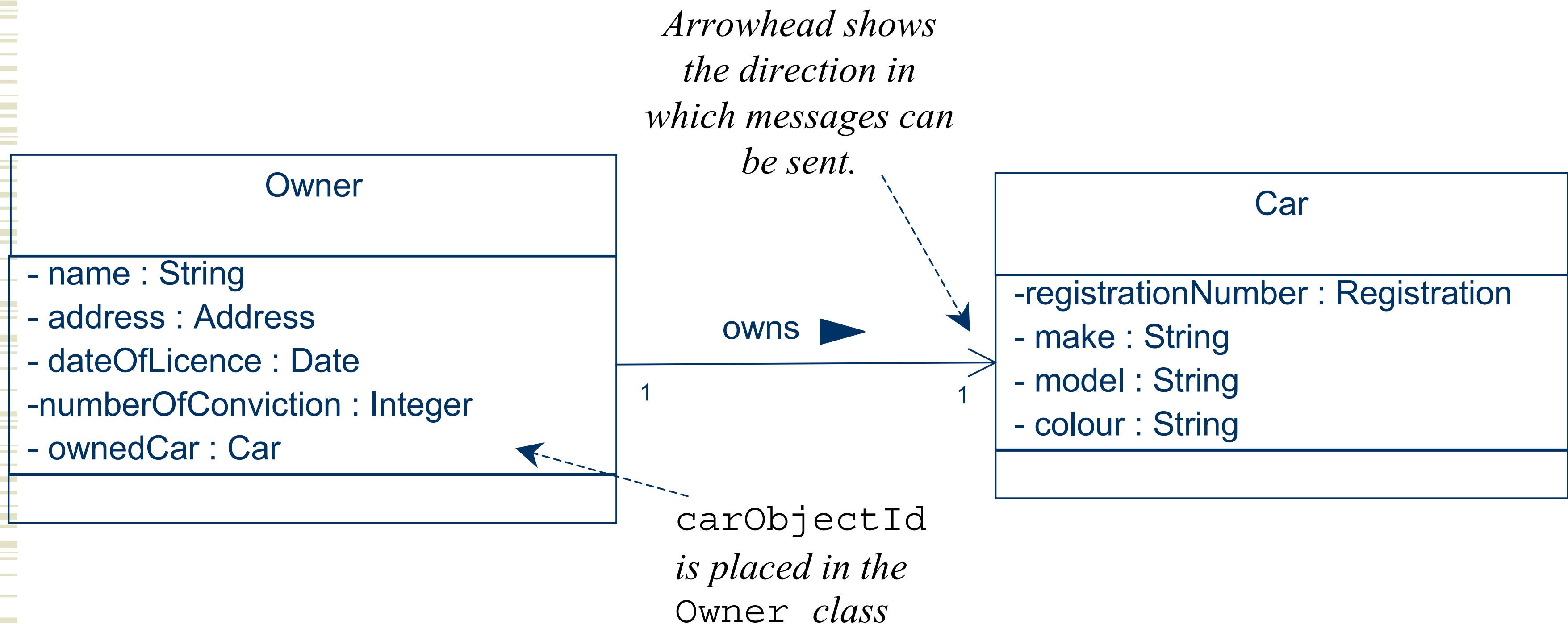
- Multiple inheritance
  - Interface – merging of interface contracts
  - Implementation
    - Operation renaming

Designing Associations (1)

- Associations indicate possibility of links
- Message-passing requires link
- Multiplicities restrict the number of links
- Association navigability
  - Do you have to send message?
  - Do you have to provide references?
  - But, references may be passed in messages!
  - Minimising the number of two way associations keeps coupling low

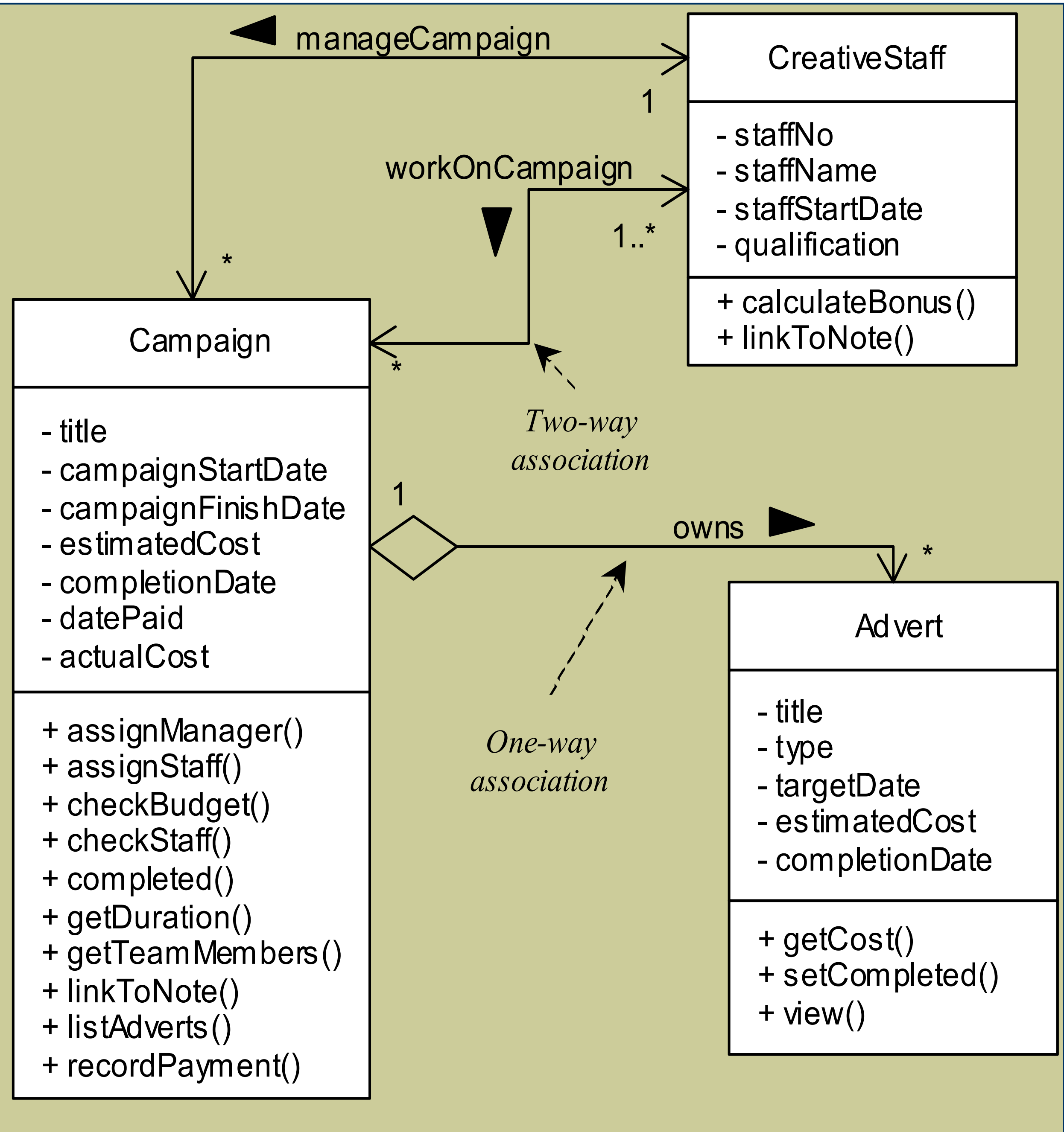
Designing Associations (2)

- One-to-one associations



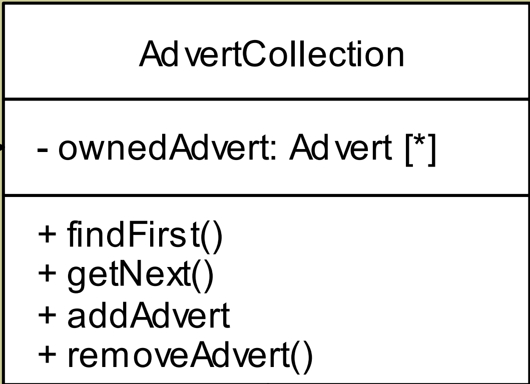
Des

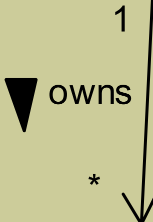
- One-to-many associations

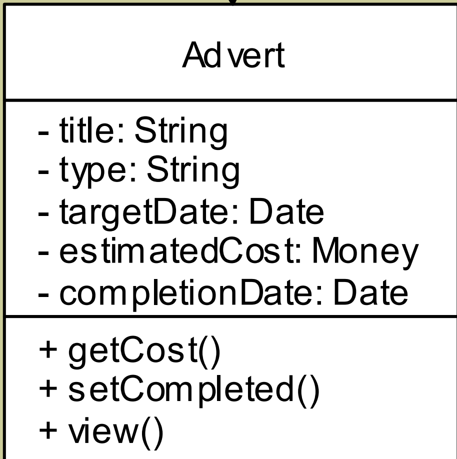


Campaign
- title: String
- campaignStartDate: Date
- campaignFinishDate: Date
- estimatedCost: Money
- completionDate: Date
- datePaid: Date
- actualCost: Money
- ownedAdvertCollection: AdvertCollection
+ assignManager()
+ assignStaff()
+ checkBudget()
+ checkStaff()
+ completed()
+ getDuration()
+ getTeamMembers()
+ linkToNote()
+ listAdverts()
+ recordPayment()

1
has
1


AdvertCollection
- ownedAdvert: Advert [*]
+ findFirst()
+ getNext()
+ addAdvert
+ removeAdvert()


1
owns
*


Advert
- title: String
- type: String
- targetDate: Date
- estimatedCost: Money
- completionDate: Date
+ getCost()
+ setCompleted()
+ view()

Designing Associations (5)

:Campaign

:AdvertCollection

:Advert

listAdverts()

advert = findFirst()

advertTitle = getTitle()

[until no more adverts]
loop

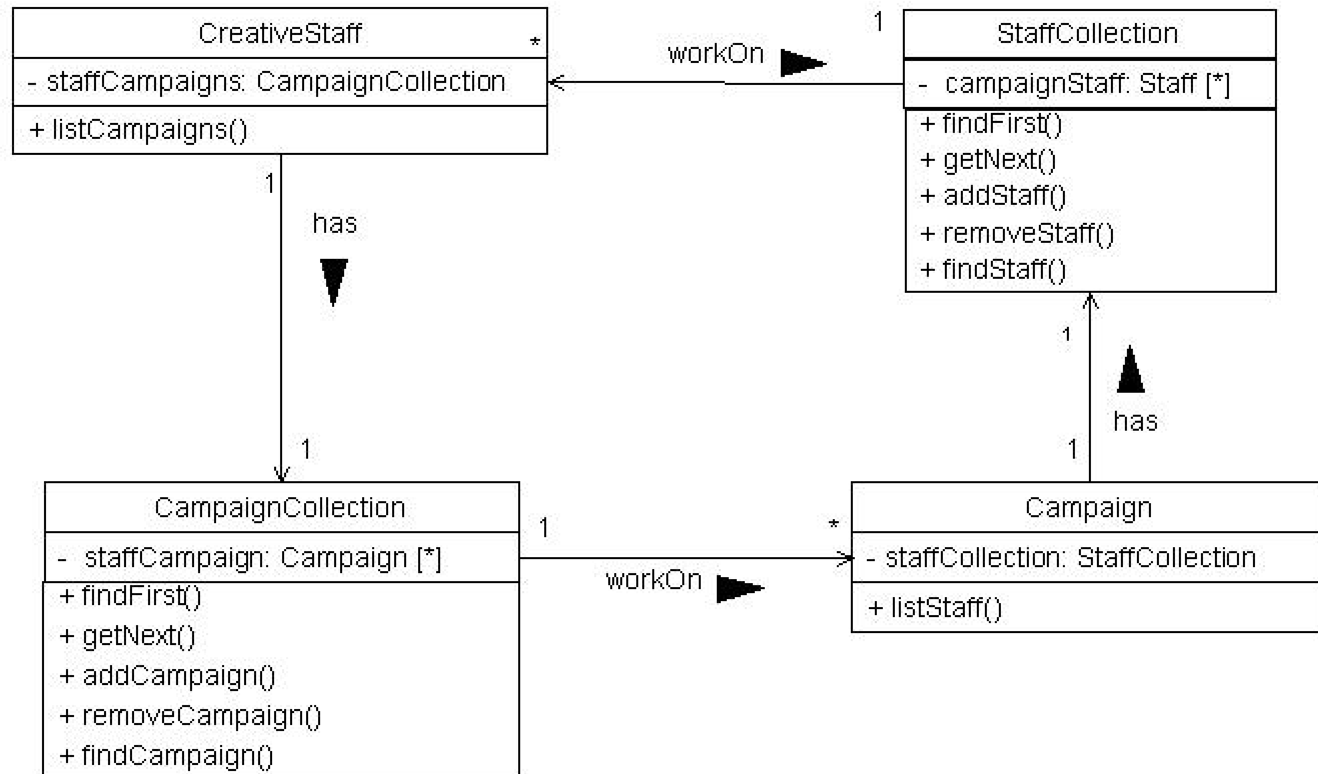
advert = *getNext()

advertTitle = getTitle()

end loop

Designing Associations (6)

- Many-to-many associations
  - Inner collection classes
  - Library collection classes

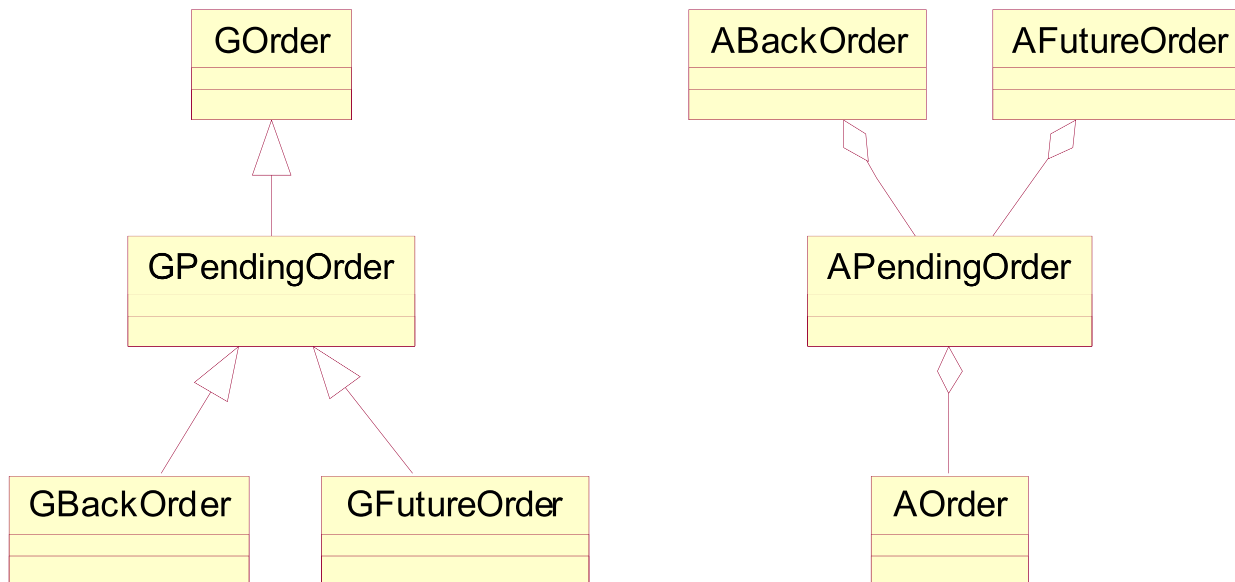


Aggregation and delegation (1)

- Aggregation and composition are kinds of associations
- Aggregation and composition are containment relationships
- Composition is a a kind of aggregation with existence dependency
- Different kinds of aggregation
  - Exclusive owns (composition, frozen)
  - Owns (composition)
  - Has (aggregation with transitivity and asymetricity)
  - Member (aggregation with many-to-many multiplicity)

Aggregation and delegation (2)

- Generalisation versus aggregation
  - Classes versus objects
  - Inheritance versus delegation



Aggregation and delegation (3)

- Delegation and prototypical systems
  - Delegation: composite object (outer) – component objects (inner)
  - Object (prototype) cloning
  - Aggregation – exposing the inner classes
  - Composition – encapsulating the inner classes
- Treaty of Orlando: same system functionality can be delivered with inheritance or delegation
  - Self-recursion has to be explicitly planned and designed into delegation
    - Fragile base class problem is a result of unplanned reuse
  - Delegation enables dynamic sharing and reuse
    - Anticipatory and un-anticipatory sharing

Integrity Constraints

- Referential integrity
  - Two-way associations
  - What happens when objects are removed?
  - Cascading deletes
- Dependency constraints
  - Derived attributes and associations
  - Synchronising operations
  - Prevention and exceptions
- Domain integrity
  - Ensure maintenance of invariants in update operations

Designing Operations

- Algorithm design
  - Cost of implementation
  - Performance constraints
  - Requirements for accuracy
  - Capabilities of the implementation platform
- Computational complexity
- Ease of implementation and understandability
- Flexibility
- Fine-tuning the object model
- Operation design documents
  - Activity diagrams
  - Formal specifications
- Some guidelines
  - Operation should reside in the same class as the attributes that manipulate
  - Minimise object interaction
  - Simplicity
  - Place operation that not owned by entity classes in control classes

Normalisation

- Functional dependencies
  - Attribute A is functionally dependent on attribute B if for every value of B there is precisely one value of A associated with it at any given time
  - Rules of normalisation group attributed along functional dependencies – redundancy reduction