PART 2: Software specification with UML

- Use cases
- Class diagrams
- OCL
- UML tools.

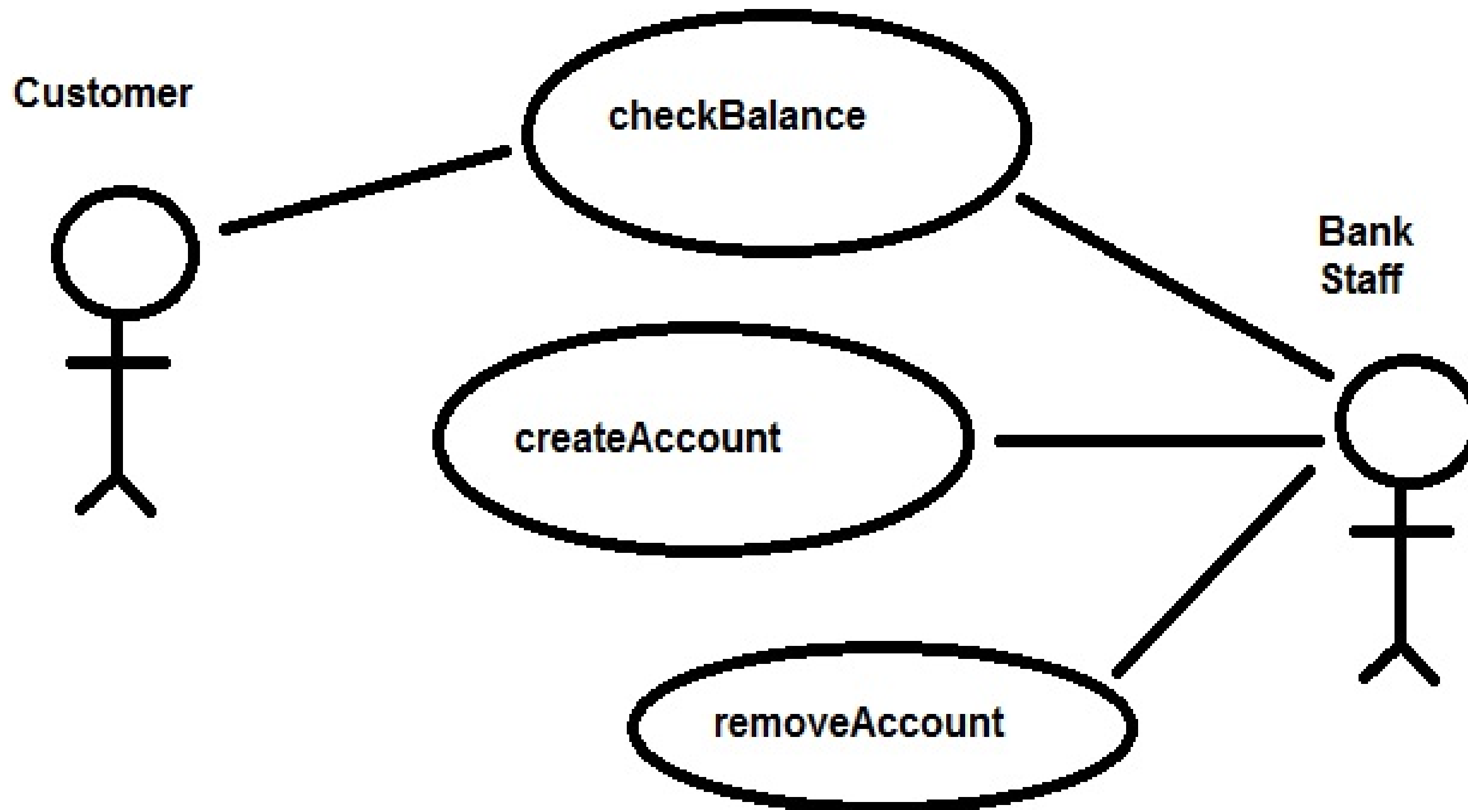
Chapters 3, 4, 5 of textbook relevant to this part.

Use Cases

- Define functionalities of system, *services* it provides to users
- Each use case has name, written in oval, linked to *actors* who interact with the case (stick figures)
- Eg., *checkBalance* use case can be used by customers or bank staff, whilst *createAccount* is only for staff.
- Actors are roles played by people, or external systems that can interact with our system.

Use cases defined during requirements analysis, to identify system purpose/capabilities.

Originated by Jacobson in 1992, now one of most widely-used software notations.



Example of use case specification

Use Cases

Diagram derived from outline text requirements:

> "The banking system should enable customers and staff to check the balance of an account, and enable staff to create and remove accounts of customers."

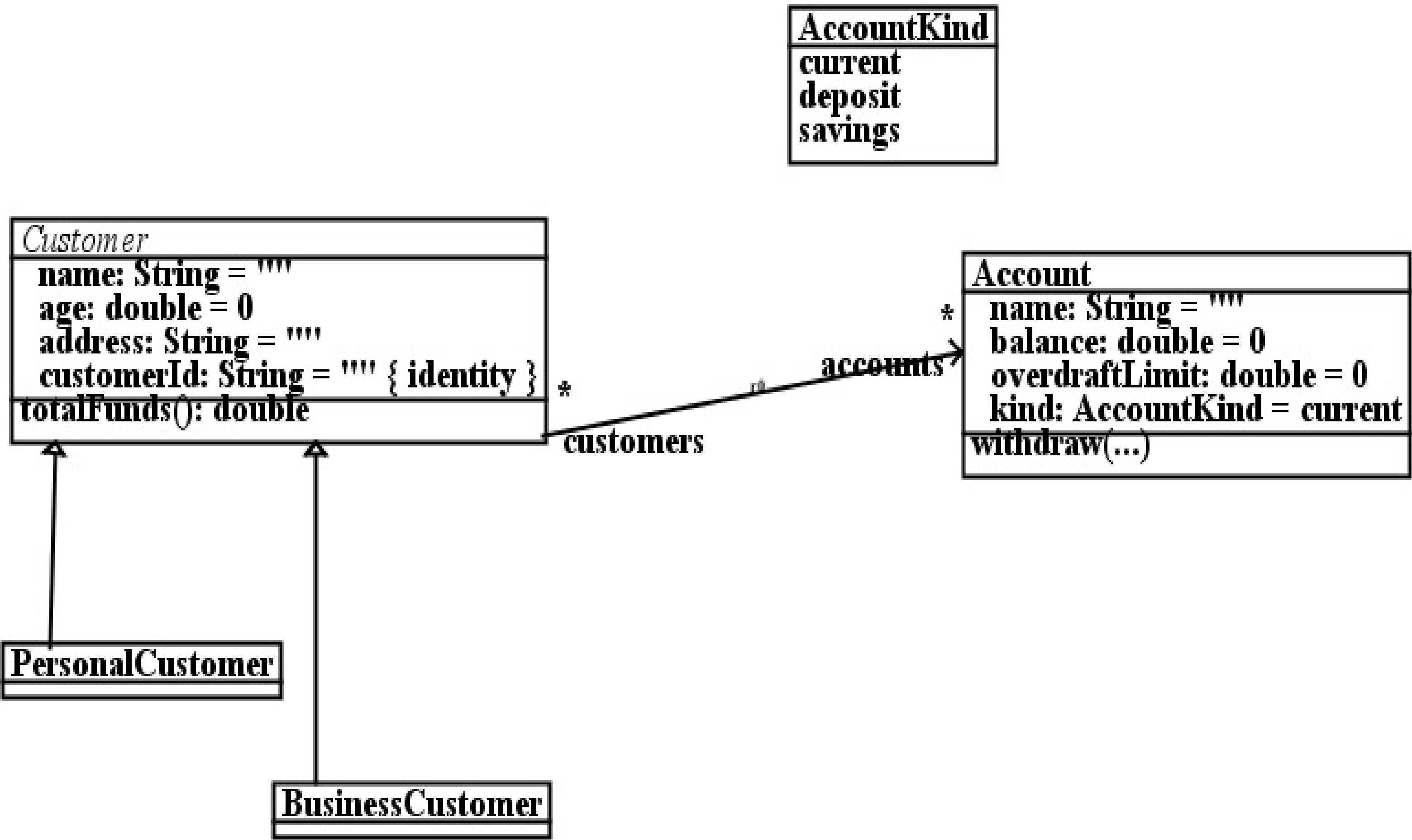
Typical of natural language requirements, this is incomplete & imprecise – use case modelling can help to improve.

Use Cases

For each use case, define its intended actions and responsibilities:

- *checkBalance*: given an account identifier (*accountId*), display balance of account
- *createAccount*: given customer identifier (*customerId*), account *accountId*, create new account with *accountId*, add this to customer's accounts
- *removeAccount*: given an *accountId*, delete the account, remove from accounts of its customer(s).

During requirements analysis, identify system data together with use cases.



Banking system classes

Use Cases

- Use cases involve sequence of steps, which perform operations on objects of system
- Use cases coordinate object behaviours to produce overall required functionality
- Eg., *checkBalance(aId : String)*:

```
Lookup account with accountId = aId;
Display balance of this account;
```

- *createAccount(cId : String, aId : String)*:

```
Lookup customer with customerId = cId;
Create new account with accountId = aId;
Add account to the customer accounts;
```

Use Cases and scenarios

- May be several scenarios for each use case – alternative situations + behaviours
- Eg., for *checkBalance*(*aId*), 2 scenarios: (i) there is an account with accountId = aId, (ii) there isn't
- For *createAccount(cId,aId)* 4 scenarios: (i) no customer exists with cId; (ii) customer with cId, but no account with aId, (iii) both customer and account exist already, but account not in customer's accounts; (iv) both customer + account exist, account already in customer's accounts.
- For each scenario, identify what use case should do.

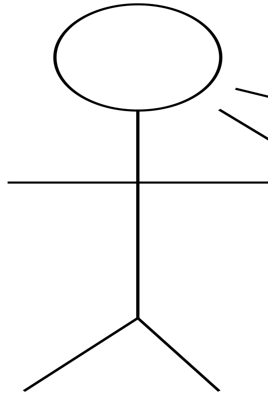
Use Cases and scenarios

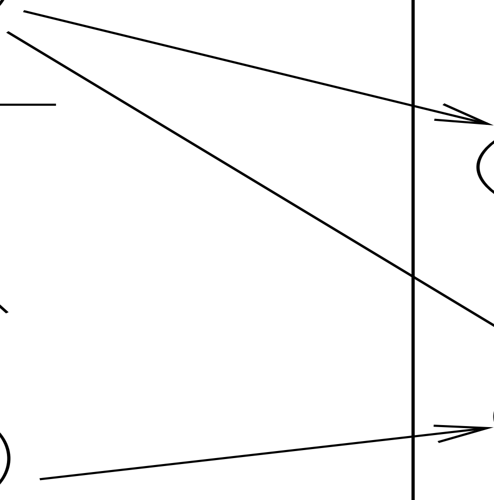
$createAccount(cId, aId)$ scenarios:

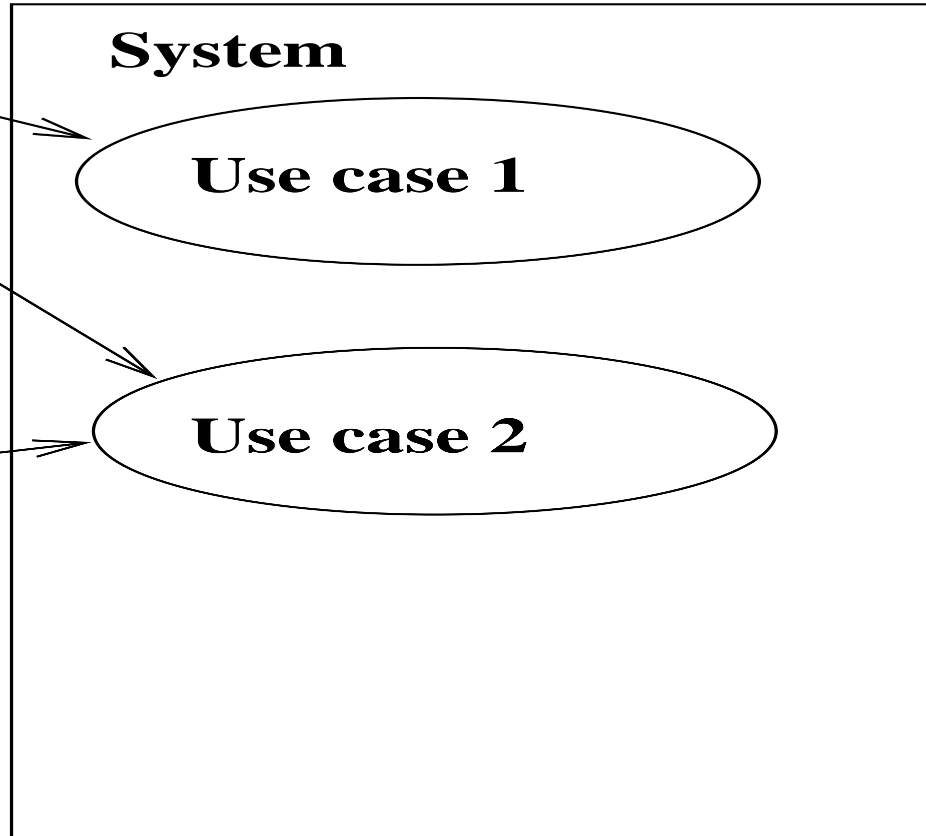
Customer with cId?	*Account with aId?*	*account in customer accounts?*	*Actions*
no	–	–	Error message – no customer
yes	no	–	Create account, add to customer's accounts
yes	yes	no	Add account to customer's accounts
yes	yes	yes	Warning message that account already in accounts

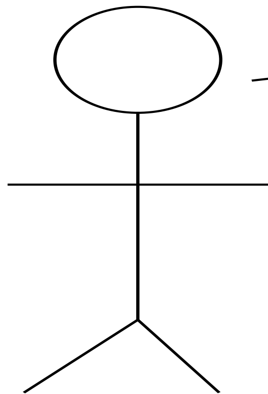
Use case diagrams

- Use case model describes (1) system to be constructed, (2) *actors* – representing a role played by person/entity that interacts with system, and (3) *use cases* – families of usage scenarios of application, grouped into similar cases of functionality.
- A use case generalises similar scenarios of use of system: a scenario is an *instance* of a use case.
- Actors also referred to as *agents*.
- In following example, two functions of system, first can be used by actor A only, second by both actors.





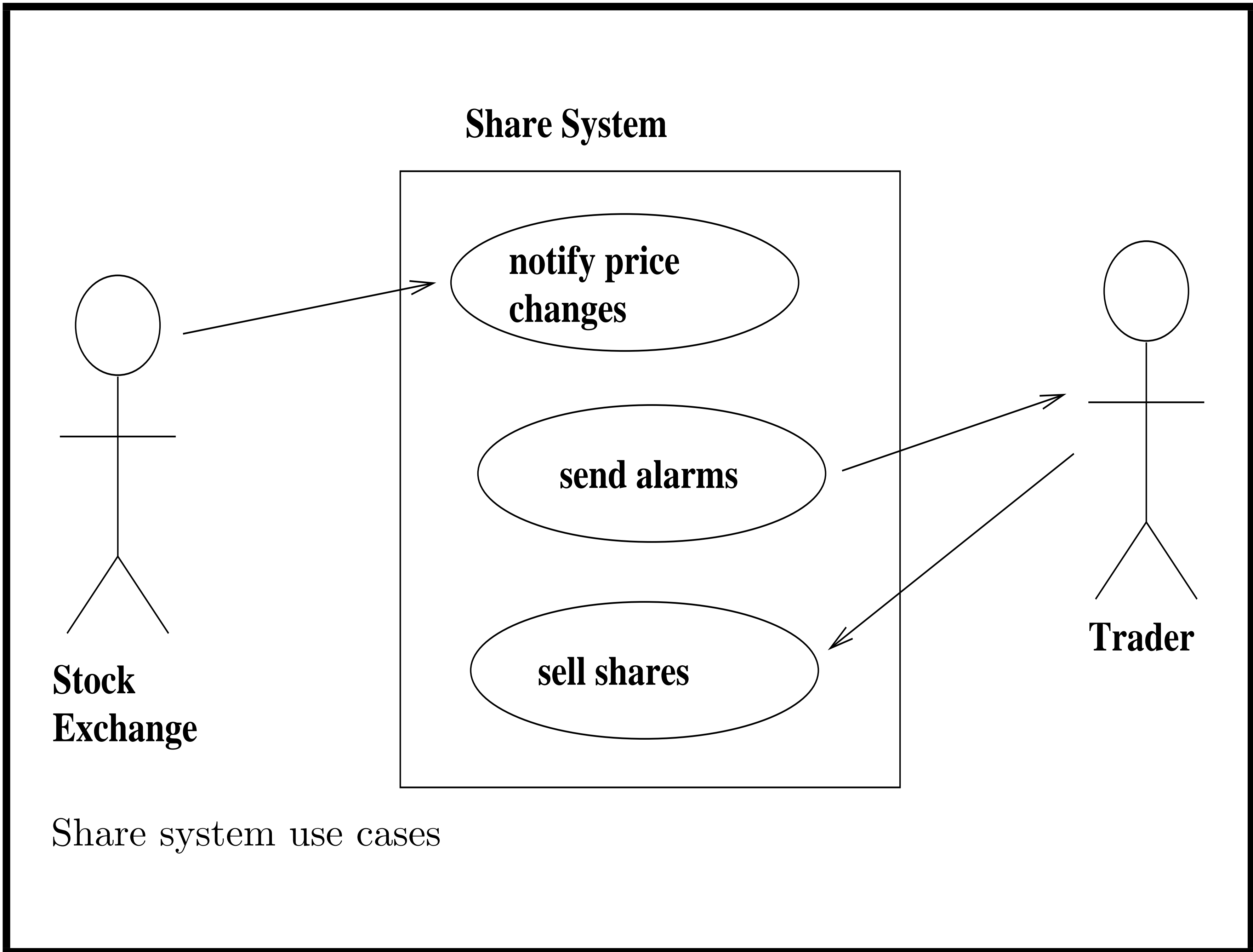




Simple use case model

Use case example – Share system

- Following figure gives use cases for share management system.
- Price changes are informed to system from stock exchange; trader can sell shares and receives alerts when share price changes pass preset limits.
- Scenarios can be shown graphically on interaction diagrams.



Share system use cases

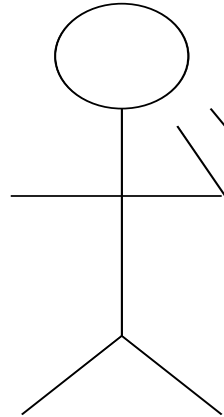
Inclusion and Extension of Use Cases

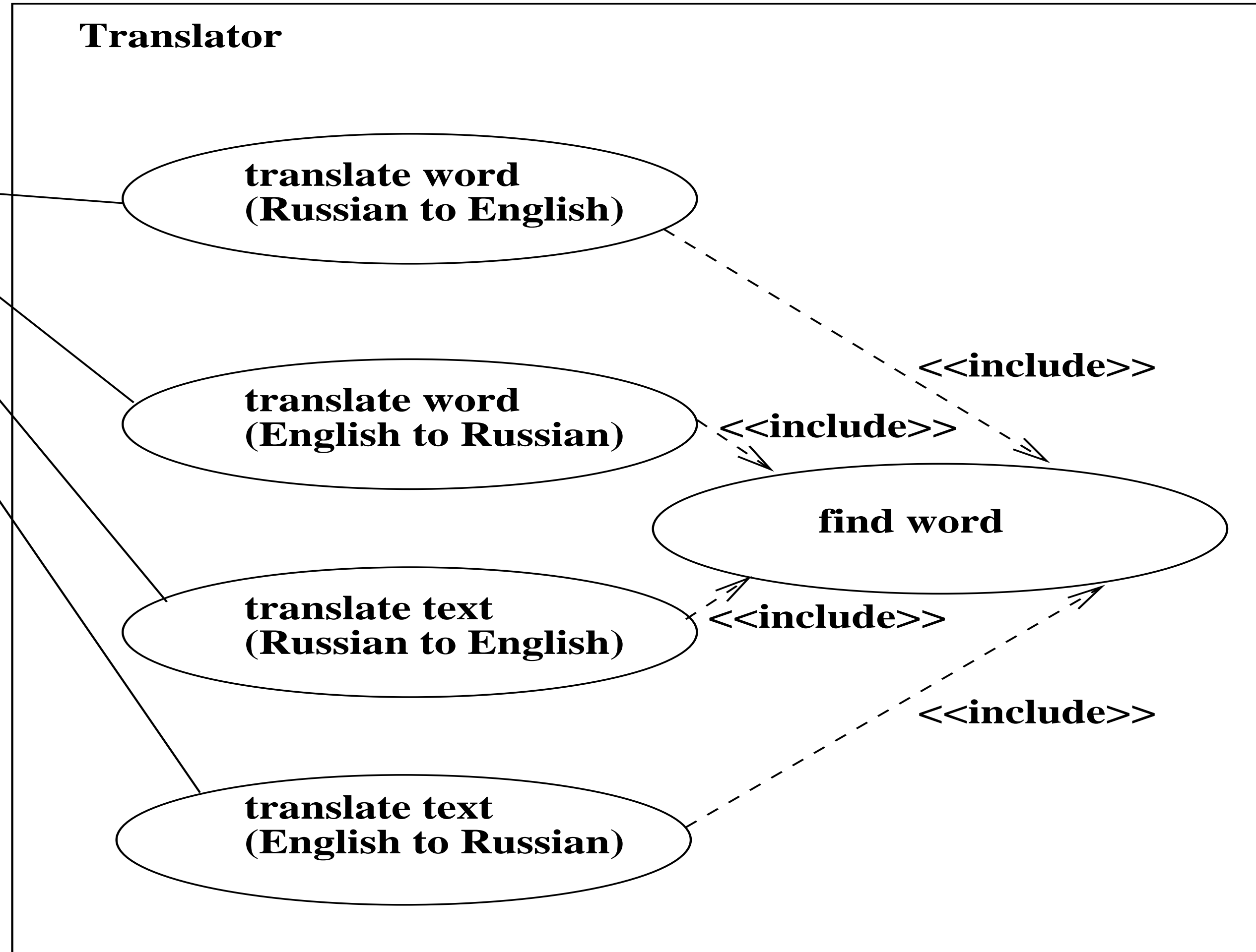
Includes: Use case $uc1$ *includes* use case $uc2$ if doing $uc1$ always involves doing $uc2$. Particularly useful if $uc2$ is common subtask of two or more use cases.

Extends: Use case $uc1$ *extends* use case $uc2$ if $uc1$ provides additional functionality used for $uc2$ in certain cases. Eg, calculating bank charges for customers has extension of imposing penalty charges for unauthorised overdrafts.

Example: language translation system

- Figure shows use cases of Russian–English translation system.
- Could be common subtask *find word* of the use cases, included in each.
- Invalid to have cycles in $\ll$ *include* $\gg$ or $\ll$ *extend* $\gg$ relations, but chains are ok.





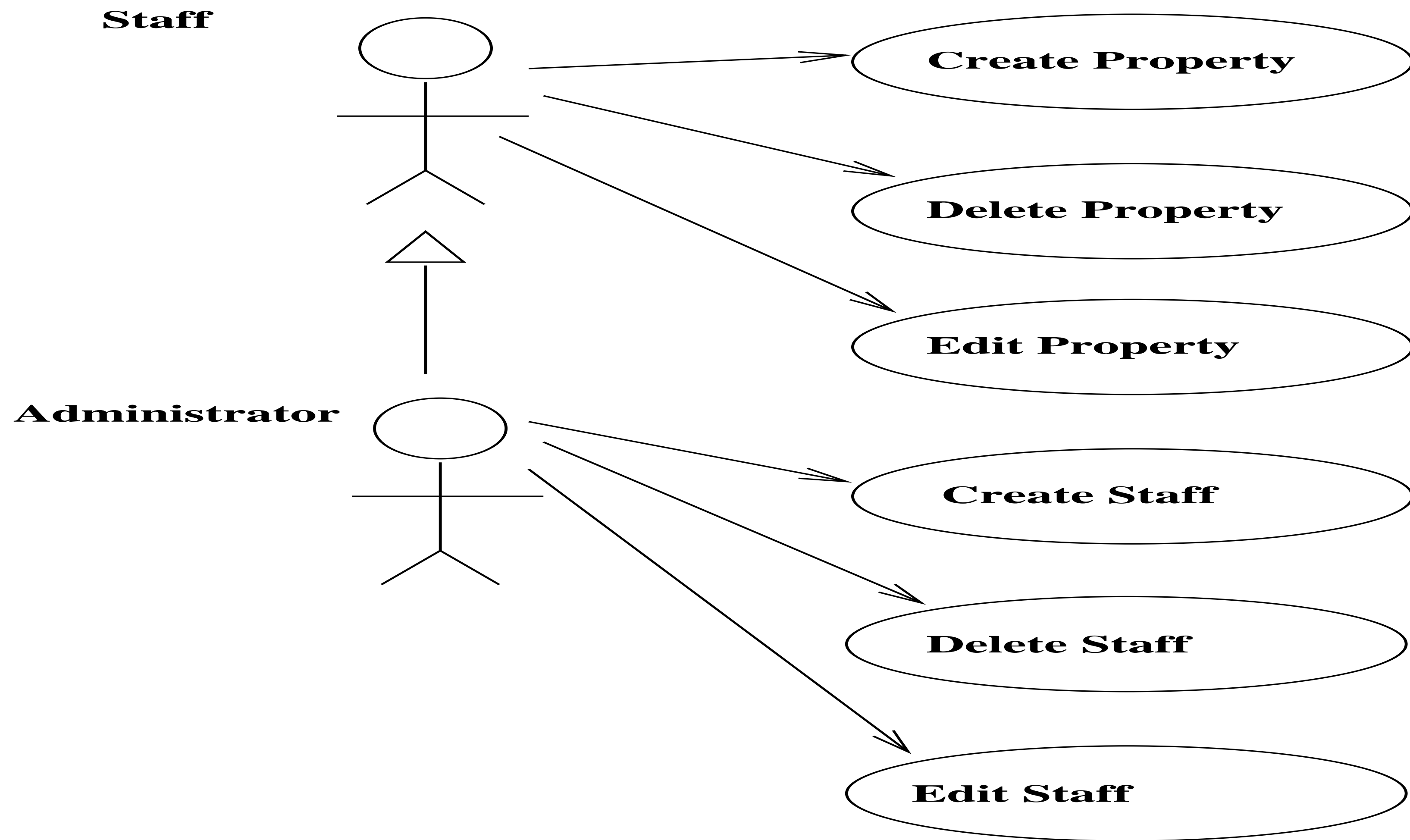
Inclusion of use cases

Use Case Relationships

Direction of arrows is common across all UML notations: arrow from element $e1$ to $e2$ indicates that $e1$ depends on $e2$: eg, by $e1$ invoking operations of $e2$ or by referring to its data.

For $\ll$ include $\gg$, use case at source of arrow depends on functionality of use case at target. For $\ll$ extend $\gg$, use case at source only has meaning with respect to use case it extends.

A use case may inherit another, if it is specialised form of the other. An actor may also inherit from another actor – specialised actor can perform all use cases of its ancestor, and possibly additional use cases.



Inheritance of use case actors

Use Cases

- Use cases may have *preconditions*, defining when they are valid to execute
- Eg., *createAccount(cId : String, aId : String)* could have precondition that a customer already exists with $customerId = cId$
- Use cases can have *postconditions*, defining effect and result, by series of expressions.

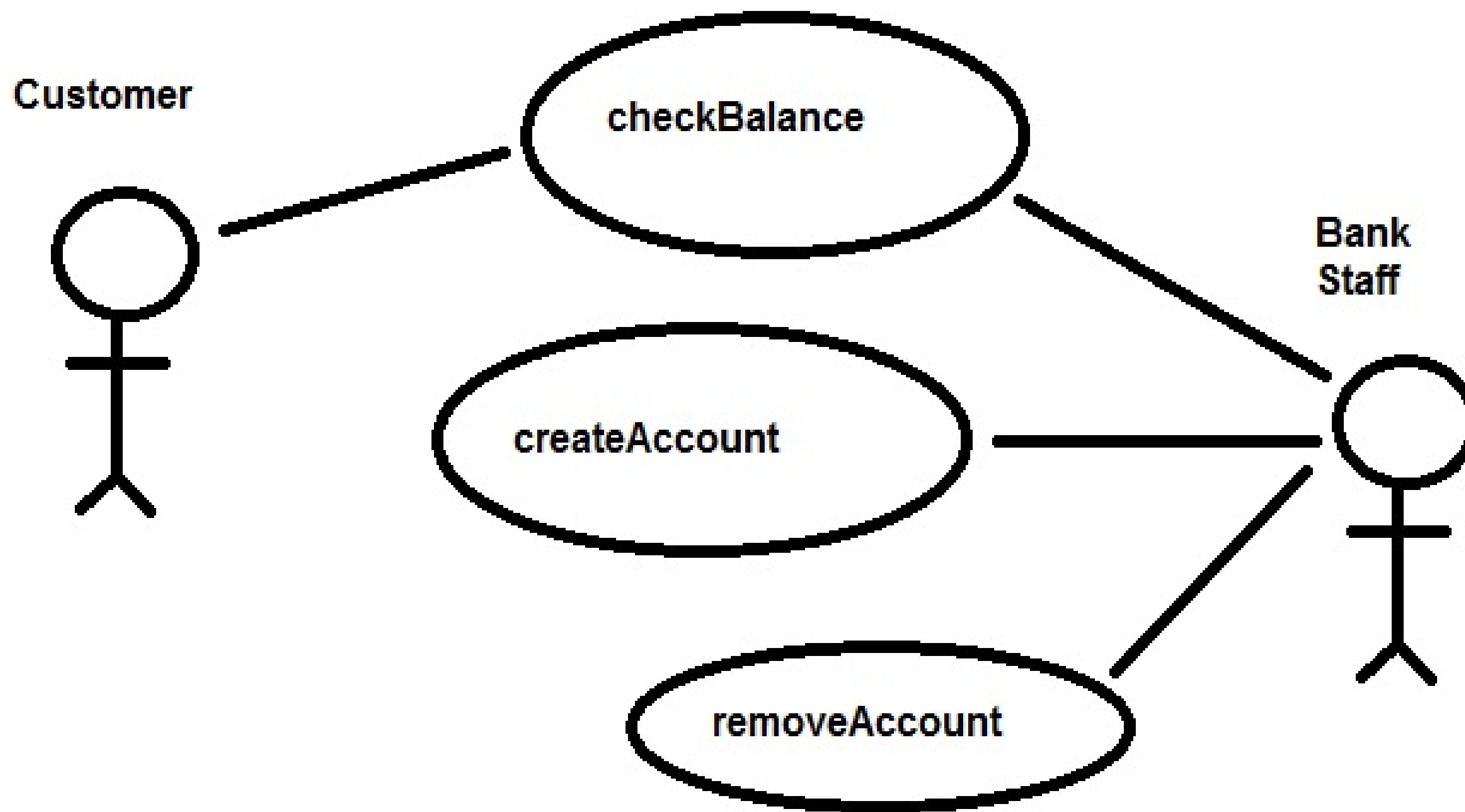
Use case validation

Having constructed use case model, need to review it for:

- Completeness – are all required use cases represented?
- Correctness – are use case name, description, actors, relationships correct?
- Consistency – are cases used in consistent manner?

Can refactor complex use cases into simpler ones, if one use case is doing too much/too complex.

Alternatively, merge 2 use cases that are similar/variants of each other.



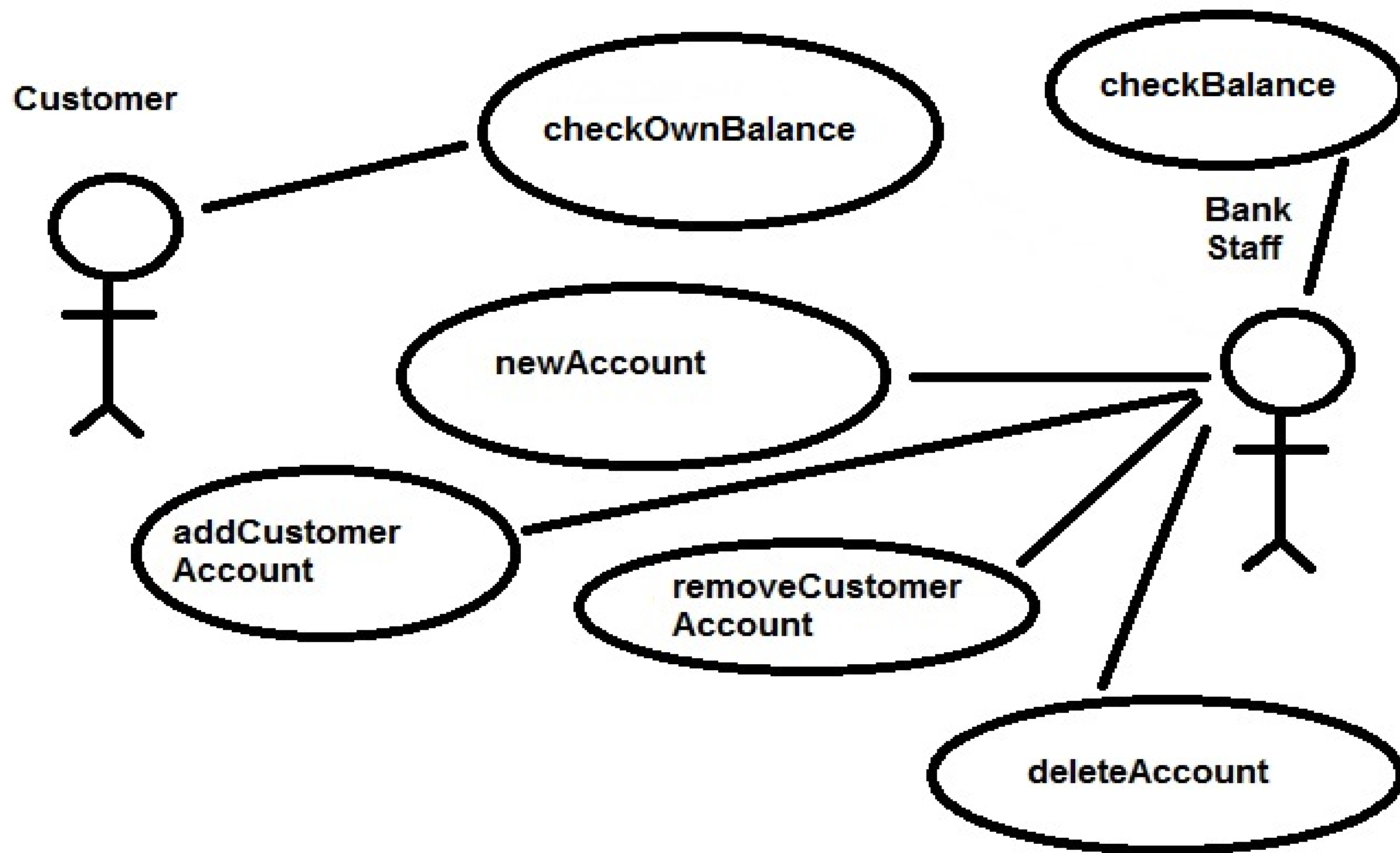
Banking use cases, version 1

Use case validation

For banking system:

- Completeness – no: staff will actually need to create accounts for a *group* of customers, not only 1 customer. Also to link and unlink accounts from customers.
- Correctness – *createAccount* is misleading name, as both creates + links, or only links (if account already exists)
- Consistency – no, *checkBalance* for customers should only permit a customer to see their *own* balances. Need 2 versions.

For banking system, review suggests should be use cases *checkBalance*, *checkOwnBalance*, *newAccount*, *addCustomerAccount*, *removeCustomerAccount* and *deleteAccount*.



Improved banking use cases

Improved use cases

- $checkBalance(aId)$ for staff – displays balance of the account
- $checkOwnBalance(cId, aId)$ for customers – only displays balance of one of customer's accounts
- $newAccount(aId)$ creates account if it doesn't already exist
- $addCustomerAccount(cId, aId)$ links customer + account
- $removeCustomerAccount(cId, aId)$ unlinks them
- $deleteAccount(aId)$ deletes the account, if it has no customers.

Coursework

Divided into 3 stages:

1. Requirements analysis + initial specification (class diagram, use case diagram)
2. Design (detailed steps for use cases, detailed definitions of operations)
3. Implementation and testing.

Investment analysis system

- Coursework concerns management + analysis of investments.
- An investor may purchase *bonds*. A bond has a *term* (number of years to expiry), *coupon* (percentage of investment, paid to investor at regular intervals), *frequency* of payments. Bonds have names, eg. "UK government bond" and purchase date.
- Assume *frequency* = 1 per year, investment = 100.
- Investors receive back invested sum at term, + final coupon payment.

Eg., invest £100 in bond b paying 10% coupon annually for 5 years. 4 payments of £10, one of £110.

System enables new bonds to be defined + added to investor's portfolio.

Investment analysis system

- Bonds have a *price*: investor pays this with sum invested
- System should compute and display the *payout* of all bonds: sum of payments. £150 in example.
- System should compute + display bond *value*s: sum of *discounted* payments, using inflation rate r:
  X after N years of inflation r is $\frac{X}{(1+r)^N}$.
  Eg., £10 1 year in future, with 5% inflation, is worth 10/1.05 today = £9.52.

System computes *discount* for each payment, and hence *value* of bond. In example, value is £121.65 for r = 0.05.

Investment analysis system

- For each bond, system should calculate + show its *Macaulay duration* for rate r:

  $$duration = (\Sigma_{p:payments} \frac{p.time * coupon}{(1+r)^{p.time}} + \frac{term * 100}{(1+r)^{term}})/value$$

  Where $p.time$ is time of p, in years from start of bond.

  In example, 4.25 years for r = 0.05

- For each bond, system should compute + show its *internal rate of return*: the r such that

  $$price = \Sigma_{p:payments} \frac{coupon}{(1+r)^{p.time}} + \frac{100}{(1+r)^{term}}$$

  Ie., $price = value(r)$.

This final use case is quite challenging to implement.

Coursework: First stage

- Carry out requirements elicitation and document system functionalities as use cases. Point out any ambiguous or incomplete requirements.
- Draw initial class diagram of system data including classes, attributes and associations, but not details of operations.

Teams will be selected randomly.

Teams should have leader/coordinator + allocate tasks to members.

You can use a UML/drawing tool of your choice (list on slide 125).

Class diagrams

- Show *entity types* of system: data types (classes) which have instances (objects) with internal structure, such as attribute values
- *Value types* include integers, reals, strings – instances of these are values with no attributes/internal structure
- Classes have name (usually singular, with initial capital)
- Classes have series of *attributes* of value type.

Class diagrams formalise data requirements such as “For each customer, their name, age and address are recorded”.

Customer
name: String = ""   age: double = 0   address: String = ""

Example of class specification

Class diagrams

- Can be used for initial conceptual modelling of a system
- For system specifications, informal and formal
- As executable specification in model-based development (MBD)
- Usually a UML class can be translated directly to a Java, C#, C++ class, or to a C `struct`.

Attributes

- Intrinsic properties of an object (class instance)
- Permanently attached to the object – although attribute value can change unless it is *readOnly* (*frozen*, *final*, *const*)
- Usually written with lowercase initial letter.

If $att : T$ declared in class C, and obj is instance of C, then $obj.att$ is a value of type T.

Identity attributes

- Attributes $eId : String$ which uniquely identify objects of their class
- If $e1 : E$ and $e2 : E$ with $e1.eId = e2.eId$, then $e1 = e2$
- Same as concept of *primary key* for relational databases.

We use notation $E[eval]$ for instance of E with eId value $eval$.

Eg., $Account[$“0112136”$]$ is account object with this accountId value.

Account

accountId: String = "" { identity }
name: String = ""
balance: double = 0

Identity attribute example

Enumeration types

- Types with finite number of named values
- Distinct named values listed in an ≪ *enumeration* ≫ rectangle on class diagram
- The type can be used as type of attributes.

Formalise requirements such as "There are three different kinds of account: current, deposit, and savings."

AccountKind
current deposit savings

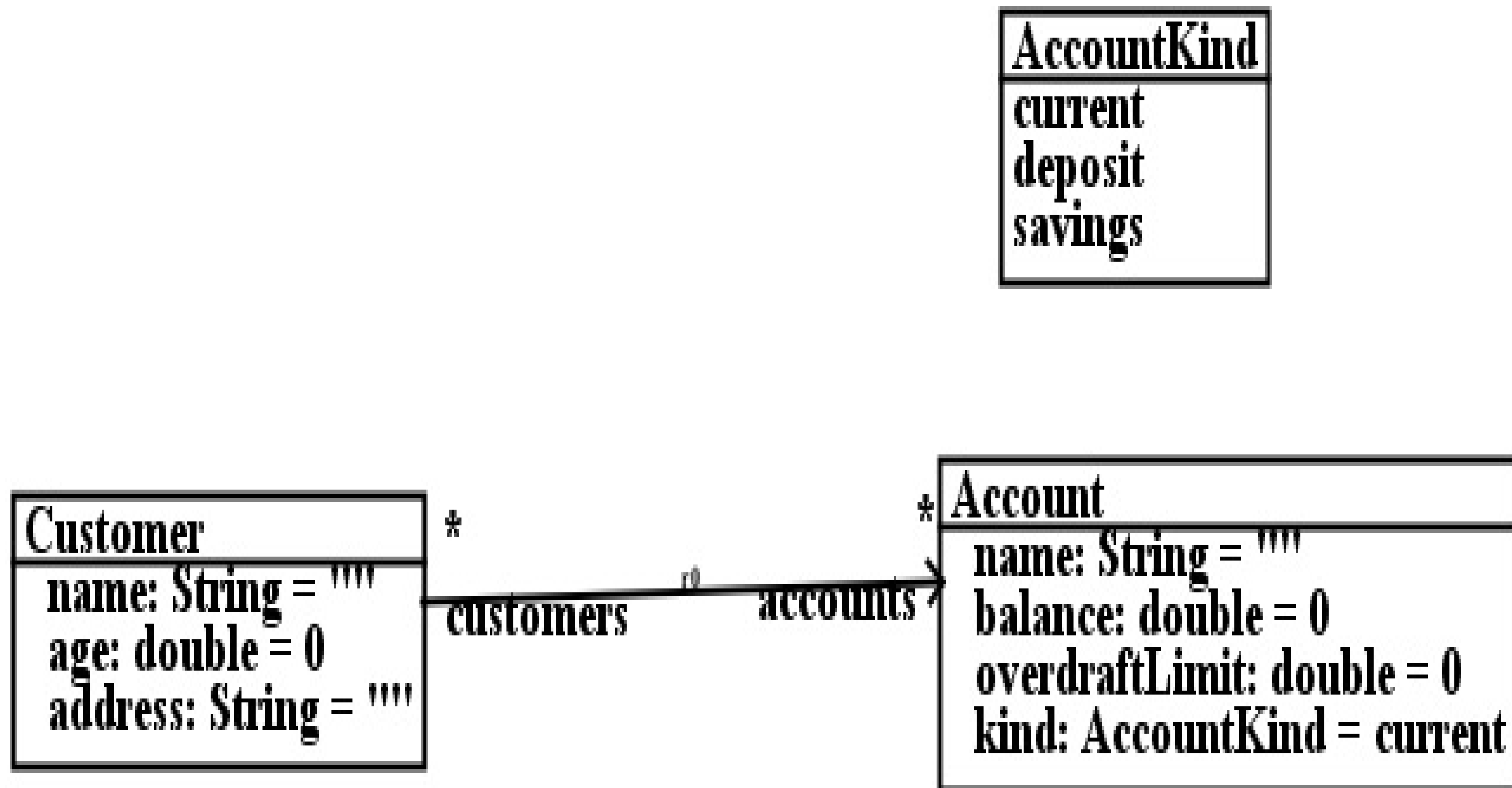
Account
name: String = "" balance: double = 0 overdraftLimit: double = 0 kind: AccountKind = current

Example of enumeration

Associations

- Define relationships between entities
- Elements of an association are pairs $obj1 \mapsto obj2$ of instances of the source and target entity types
- Rolenames and multiplicities at both ends (role1 name – at start of arrow – is optional)

Formalise requirements such as "Each customer has a set of accounts, and each account may belong to several customers".



Example of association

Consistency rules

- Cannot have two attributes with same name in same class
- Cannot have both an attribute and a role with same name
- Cannot have two associations from one class with same rolenames at their other ends – even if they end at different classes.

Association multiplicities

*	Any finite number of objects at this end can be linked to one object at other end.
0..1	At most one object at this end can be linked to one object at other end.
1	Exactly one object at this end can be linked to one object at other end.

* means role at that end is a set or sequence of objects of end class, unbounded size. 0..1 means role is set/sequence of size ≤ 1. 1 means it is a specific (non-null) object of end class.

Associations

- If $A \; {}^{m1}\!\!\mathop{—}\limits_{r}^{*} \; B$ then for each instance *obj* of A, $obj.r$ is set (possibly empty) of B objects.
- Eg., $c.accounts$ for customer c
- r is a *feature* of A, as are attributes of A. Data features usually *private* in Java, C++, etc.

```
class Customer
{ private String name = "";
  private double age = 0;
  private String address = "";
  private Set<Account> accounts = new HashSet<Account>();
  ...
}
```

Associations

- If $A \ {}^{m1}\!\!-\!\!{}^{1}_{r} \ B$ then for each instance *obj* of A, $obj.r$ is a single B object. *many-one* association.
- If $A \ {}^{m1}\!\!-\!\!{}^{*}_{r} \ {}^{\{ordered\}} \ B$ then for each instance *obj* of A, $obj.r$ is *sequence* (possibly empty) of B objects.
- If $A \ {}^{m1}\!\!-\!\!{}^{0..1}_{r} \ B$ then for each instance *obj* of A, $obj.r$ is set of 1 or 0 (empty set) of B objects. Also considered as optional B object.

$A \ {}^{m1}_{r1}\!\!-\!\!{}^{m2}_{r2} \ B$ association with neither $m1$, $m2$ being 1, is called a *many-many* association.

Bi-directional associations

- If $A \ {}_{r1}^{m1}\!\!-\!\!{}_{r2}^{m2} \ B$ then $r1$ is a feature of B, with multiplicity $m1$, $r2$ is a feature of A, with multiplicity $m2$
- $r1$ and $r2$ depend on each other: if $b : a.r2$ then $a : b.r1$, and vice-versa, for $A \ {}_{r1}^{*}\!\!-\!\!{}_{r2}^{*} \ B$
- Maintaining this consistency is difficult using hand-written code
- Bi-directional associations create strong semantic links between classes; should only be used if essential to problem.

Eg., *Customer—Account* relationship.

```
class Account
{ private String name = "";
  private double balance = 0;
  private double overdraftLimit = 0;
  private int kind = 0;

  private Set<Customer> customers = new HashSet<Customer>();
  ...
}
```

If $a.customers.contains(c)$ then $c.accounts.contains(a)$ should be true, and vice-versa.

Associations

- Concepts of attribute and association end are quite similar; both are called *properties* of classes in UML 2.
- **Rule for associations**

  If association between classes $C1$ and $C2$ has role r at $C2$ end, then for each object obj of $C1$, $obj.r$ is a collection of $C2$ objects.
- If multiplicity at $C2$ end is 1, $obj.r$ is a single $C2$ object.

Common mistake: putting multiplicities on wrong end of association.

Must go on end *nearest* **class whose number of instances in relation are being constrained.**

Role name is on association end opposite to class of which it is a property.

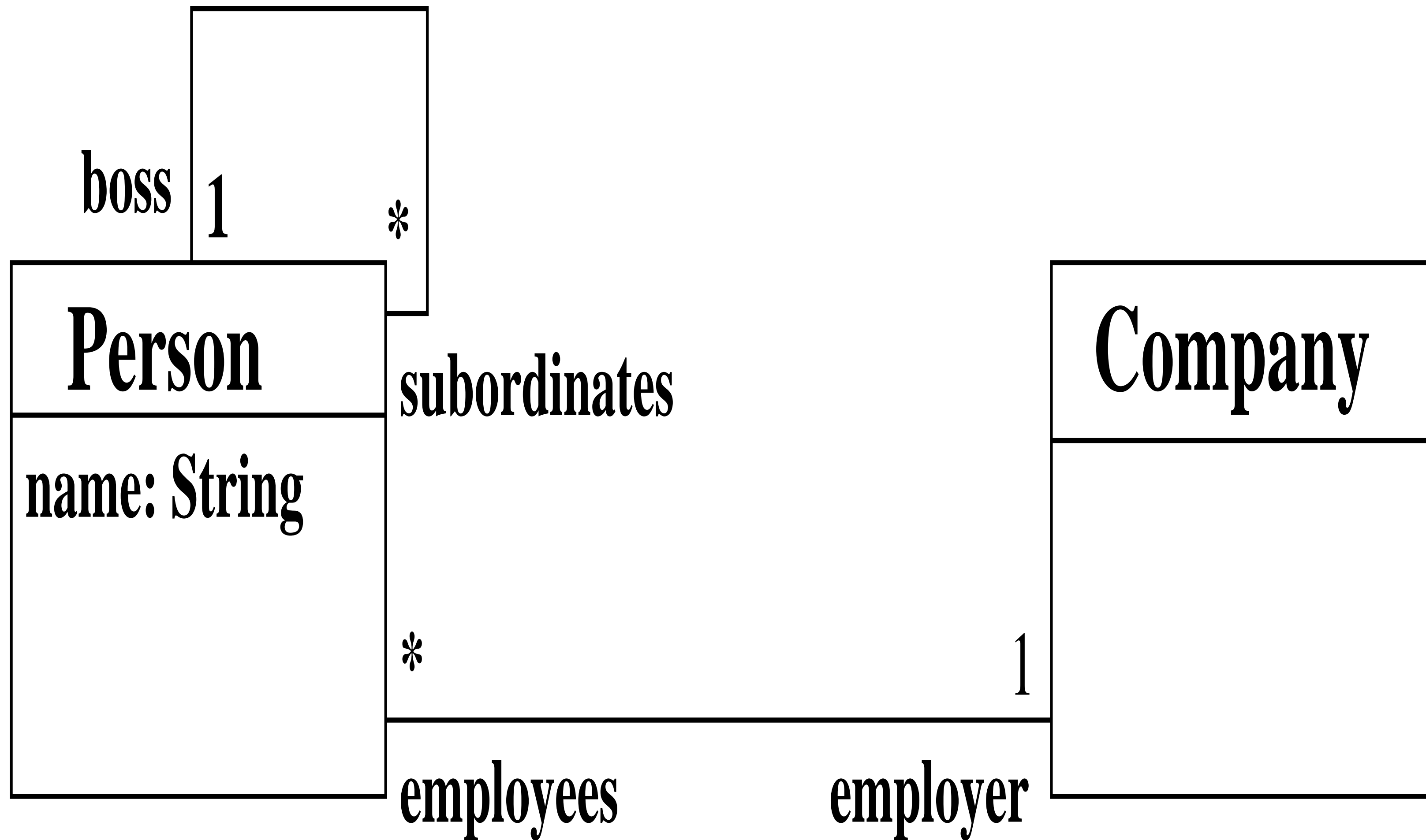
Multiplicity constraints can define any interval of natural numbers (possibly with no upper bound).

UML Notation	*Meaning*
*	any number of objects allowed (including zero)
1..*	at least one object
1	exactly one object
n	exactly n objects
a..b	at least a and at most b

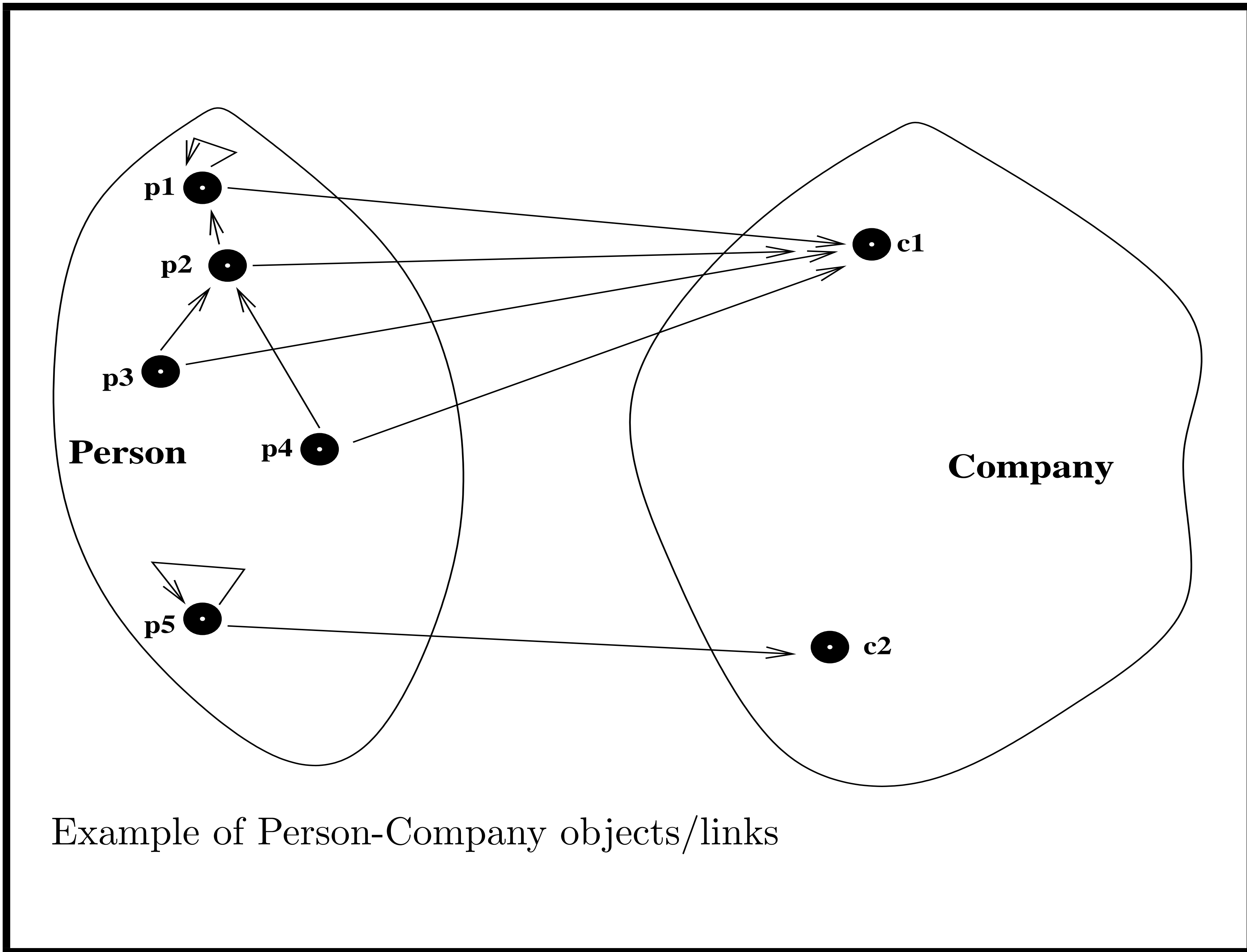
Association examples

- Most common form of association is many-one association, * multiplicity at one end, 1 multiplicity at other.
- Acts like function from * end to 1 end.
- Example below has self-association on *Person*, each person has 1 boss (+ in other direction, a set of subordinates).
- Other association gives company for each person, and for each company, its set of employees.

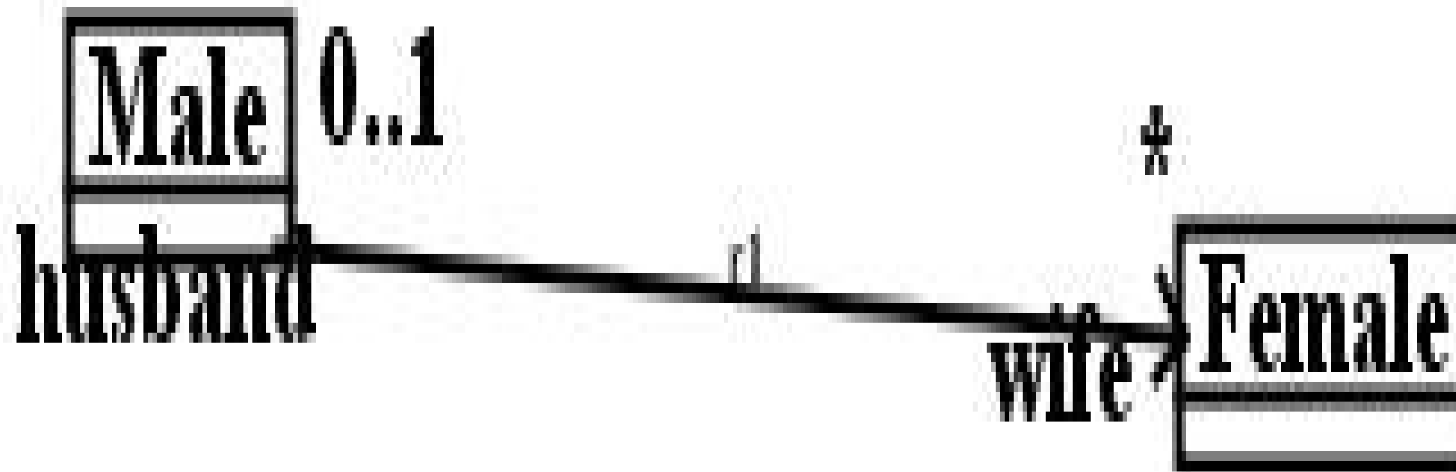
  "Each person has an employer, which is a company. Each company has a set of employees, which are persons".



Employment class diagram



Example of Person-Company objects/links



Another bi-directional association example

Special association notations

- Association roles can be *derived* (computed from other information in model): role name written with prefix "/".
- Associations can have a *navigation* direction, indicating how implementation code will navigate it. Arrow on the end that will be navigated to.
- Arrows can be placed on both ends if bidirectional reference is required.
- Association without arrows on either end means no information about navigation.
- Forbidden navigation indicated by diagonal cross on end to which navigation is not possible.

Aggregation

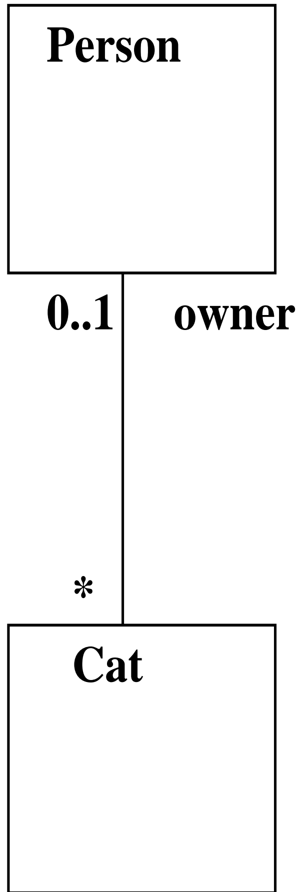
- Models situation where one class has a whole-part relation to another (eg., car-wheels)
- Represented by black diamond at 'whole' end
- Semantic effect is if a 'whole' object is deleted, so are all its linked parts (cascaded delete)
- Multiplicity must be 1 or 0..1 at 'whole' end.

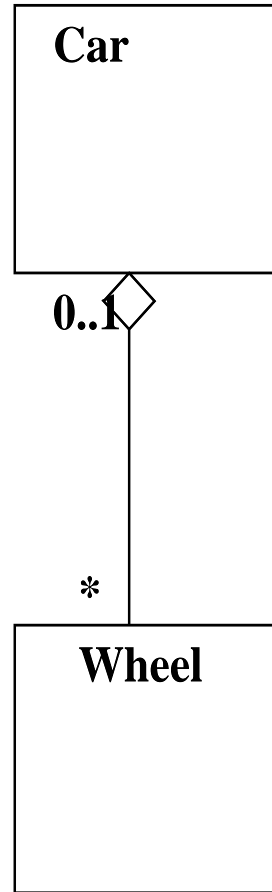
Is this diagram correct? Can a wheel exist without being attached to a car?

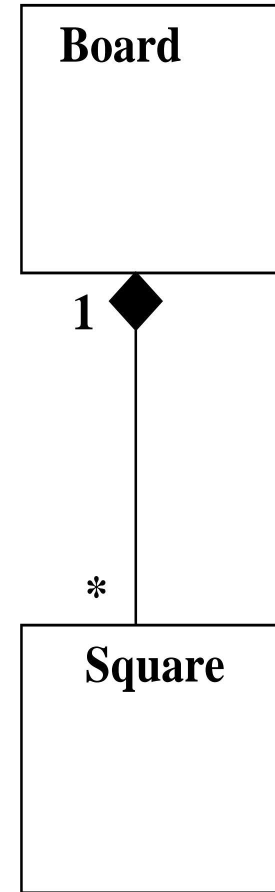
Aggregation example

Aggregation and composition

- Association represents concept of 'has' between entities: "a person has some cats", inheritance represents 'is a': "a business customer is a customer", etc.
- Aggregation represents concept of one object being 'part of' another. Eg., "a wheel is part of a car", or "a square is part of a board".
- Strong form of aggregation is *composition* – represented by filled diamond at 'whole' (owner) end. Parts cannot exist without whole, and vice-versa. Eg., a chess board + its squares.
- UML also has simple aggregation, represented by open diamond at 'whole' end. Parts/wholes can exist independently.





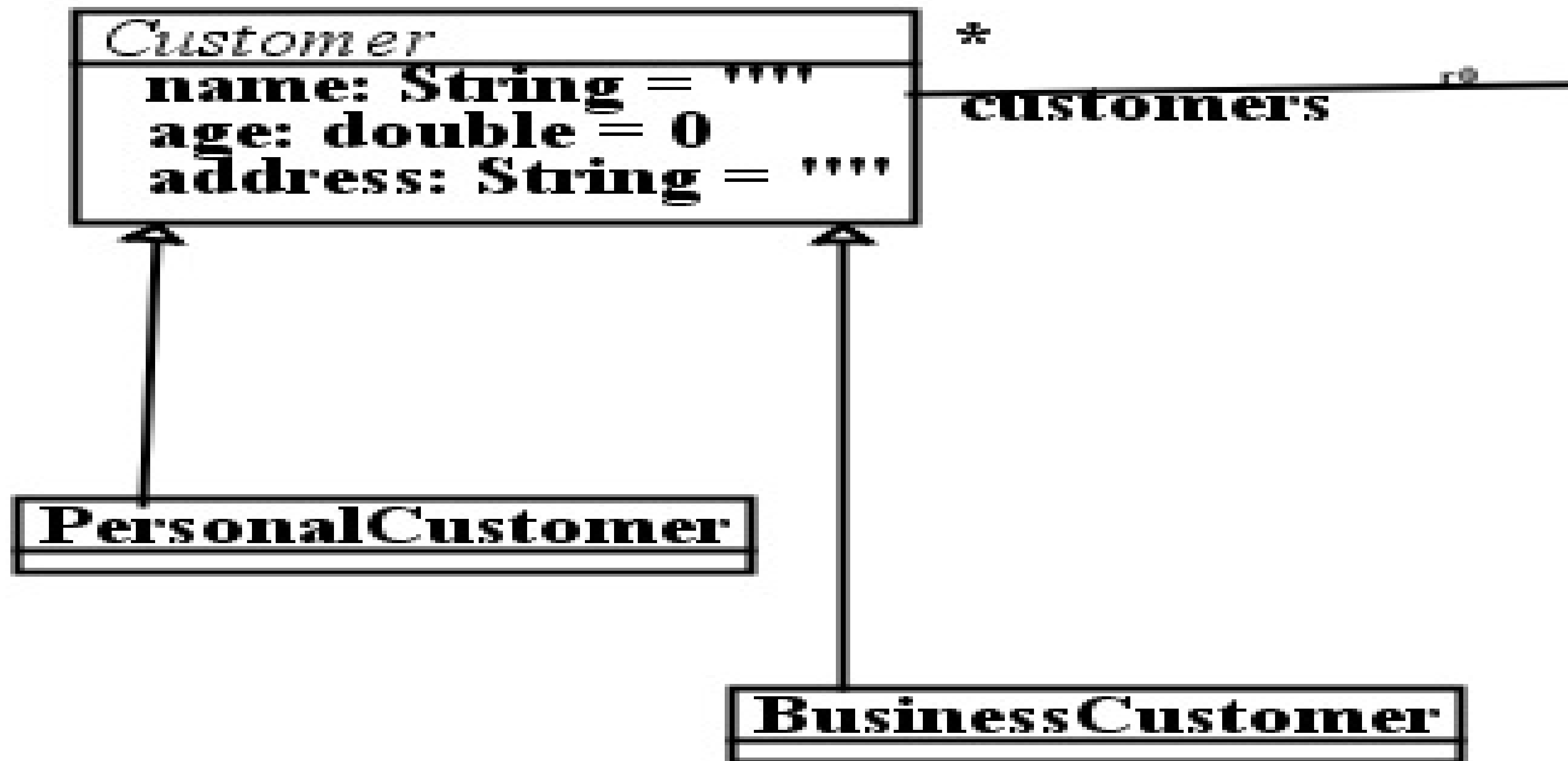


Aggregation and composition

Inheritance

- Defines specialisation/generalisation relationships between entity types
- Inheritance arrow points from subclass (specialised entity type) to superclass (generalised entity)
- **No rolenames/multiplicities**
- Superclass is usually *abstract*: instances cannot be created for it, only for subclasses. Names of abstract classes written in italic font.
- Instantiable (non-abstract) classes are called *concrete* classes.

Models requirements such as “Customers may be either business customers or personal customers.”



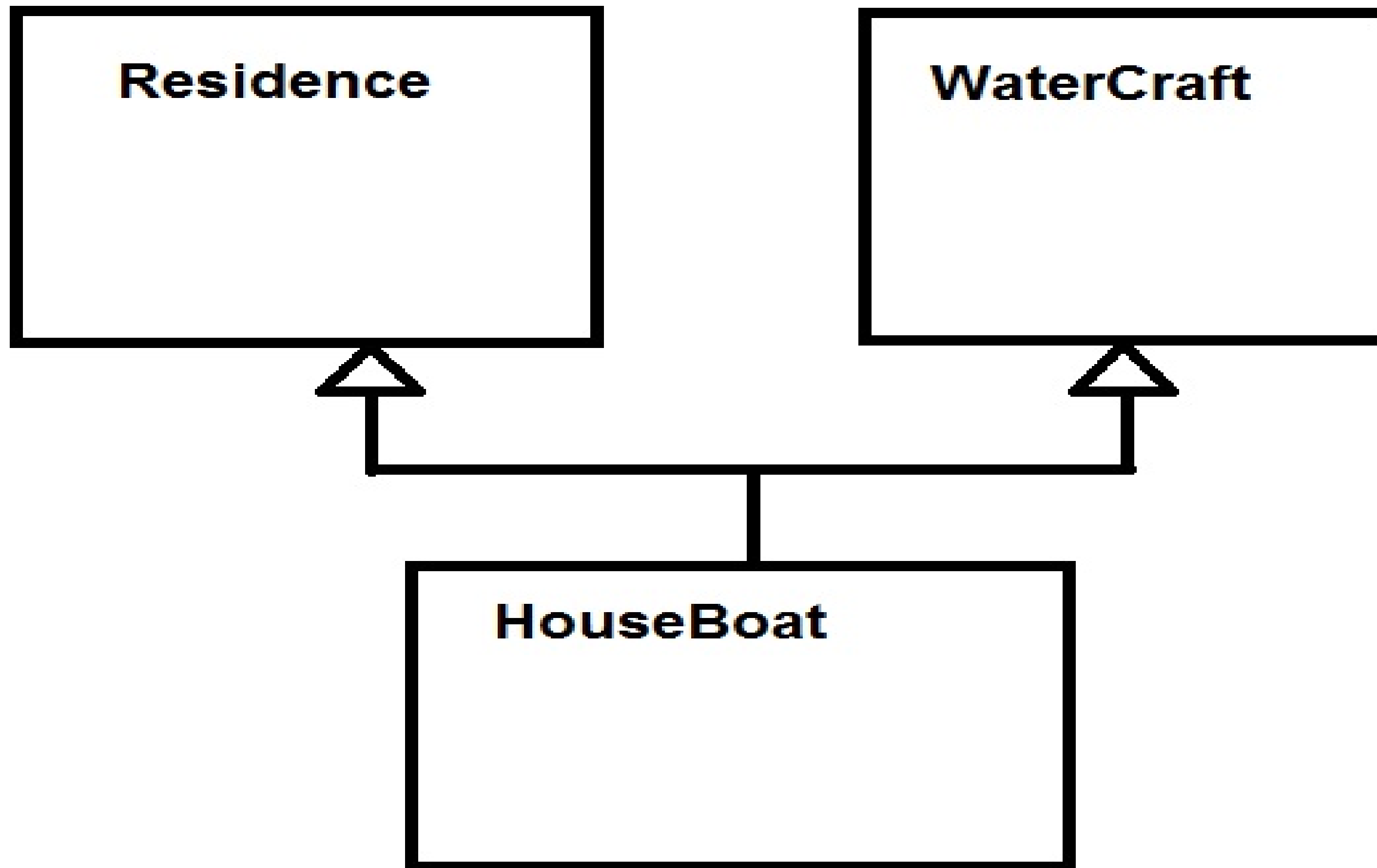
Example of inheritance

Inheritance

- Can have *multiple subclassing*: several specialisations of same superclass (eg., *PersonalCustomer* and *BusinessCustomer* as subclasses of *Customer*)
- **All features of all superclasses are inherited by a subclass**
- Same concept as Java `extends` and C++ `public` inheritance.
- Subclasses usually *exclusive* – no object can belong to more than one subclass simultaneously.

Multiple inheritance

- means that a class may directly inherit from more than one other class, eg, *HouseBoat* inherits from *Residence* and *WaterCraft*.
- UML permits this, but some languages, eg., Java, restrict inheritance to *single inheritance*: each class has at most one immediate superclass.
- Cycles of inheritance are always disallowed: if A inherits from B, then B cannot inherit directly or indirectly from A.
- Java and C# disallow multiple inheritance. C++ permits it.



Multiple inheritance example

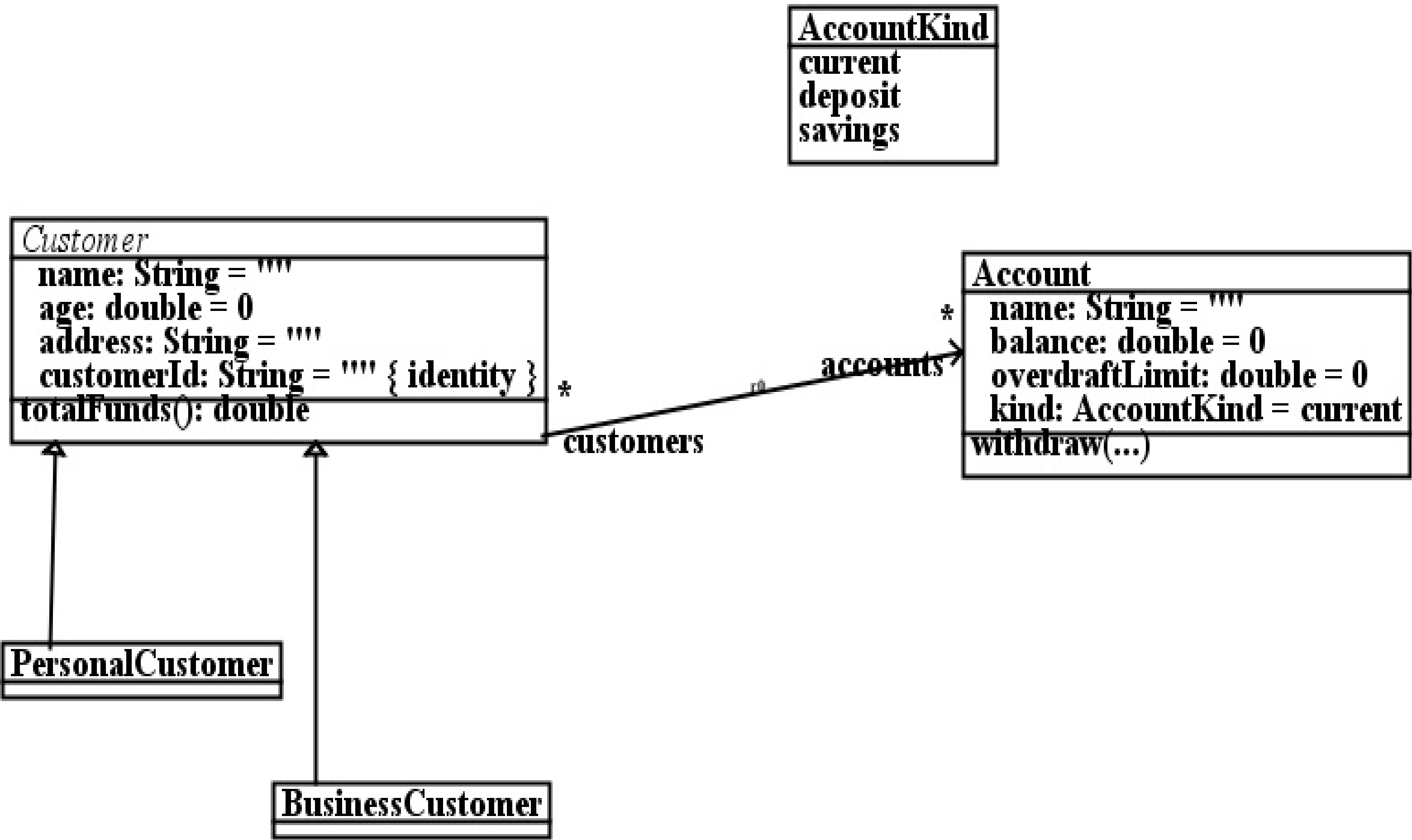
Operations

- Classes may have *operations* or *methods*. Attributes, roles and operations/methods are called the *features* of a class.
- Operations specify behaviour of objects of class – how they respond to events or messages directed at the objects.
- Operations listed in class rectangle beneath attributes.
- Operation names usually written with initial lowercase letter.

Operations

- Operations of a class are either *query operations*: return a value, do not update object state or *updaters*: modifying object state (may return a value)
- Operations can be specified by *preconditions* and *postconditions*: expressions that define assumptions at start of execution, + define result state at end
- Alternatively, behaviour can be defined by state machine or activity/pseudocode (see Part 3).

$totalFunds() : double$ is query operation. $withdraw(amt : double)$ and $deposit(amt : double)$ are updaters.



Classes with operations

Operations

- Operations are *query*, if do not modify internal state of object, or *update*, if they do.
- Query operations are written with $\{query\}$ constraint. They **must** have a result type.
- If c is *Customer* object, then $c.totalFunds()$ is a double value.
- In contrast, $a.withdraw(500)$ is call of update operation *withdraw* on account a. Update operations usually do not have result types.
- Any operation can have input parameters, which supply information to it.

Rules for operations

- If $op(x : T) : S$ is query operation of class C, obj an object of C, and e value of type T, then $obj.op(e)$ evaluates to value of type S.
- If $op(x : T)$ is update operation of C, obj an object of C, and e value of T, then $obj.op(e)$ represents invocation of op with parameter e on obj.

Operations

- Effect of operation on an object can be specified by *postcondition* constraint in OCL.
- An operation is *declared* in a class if it is written, together with its parameters and their types, and its result type, if any, in class box.
- It is *defined* in a class if it is declared in the class or in a superclass.
- Operation may be *abstract* – not intended to be executed on an object of this class (but only on objects of subclasses).
- Abstract operations written in italic font in class box.

Operation specifications

```
query totalFunds() : double
pre: true
post:
  result = accounts->collect(balance)->sum()
```

Precondition *true* means operation always available.

Value of expression assigned to *result* in postcondition is value returned by operation call.

In this case, sum of balances of accounts of customer is returned.

Operation specifications

To define updaters, use notation att@pre to denote value of data feature *att* at start of operation:

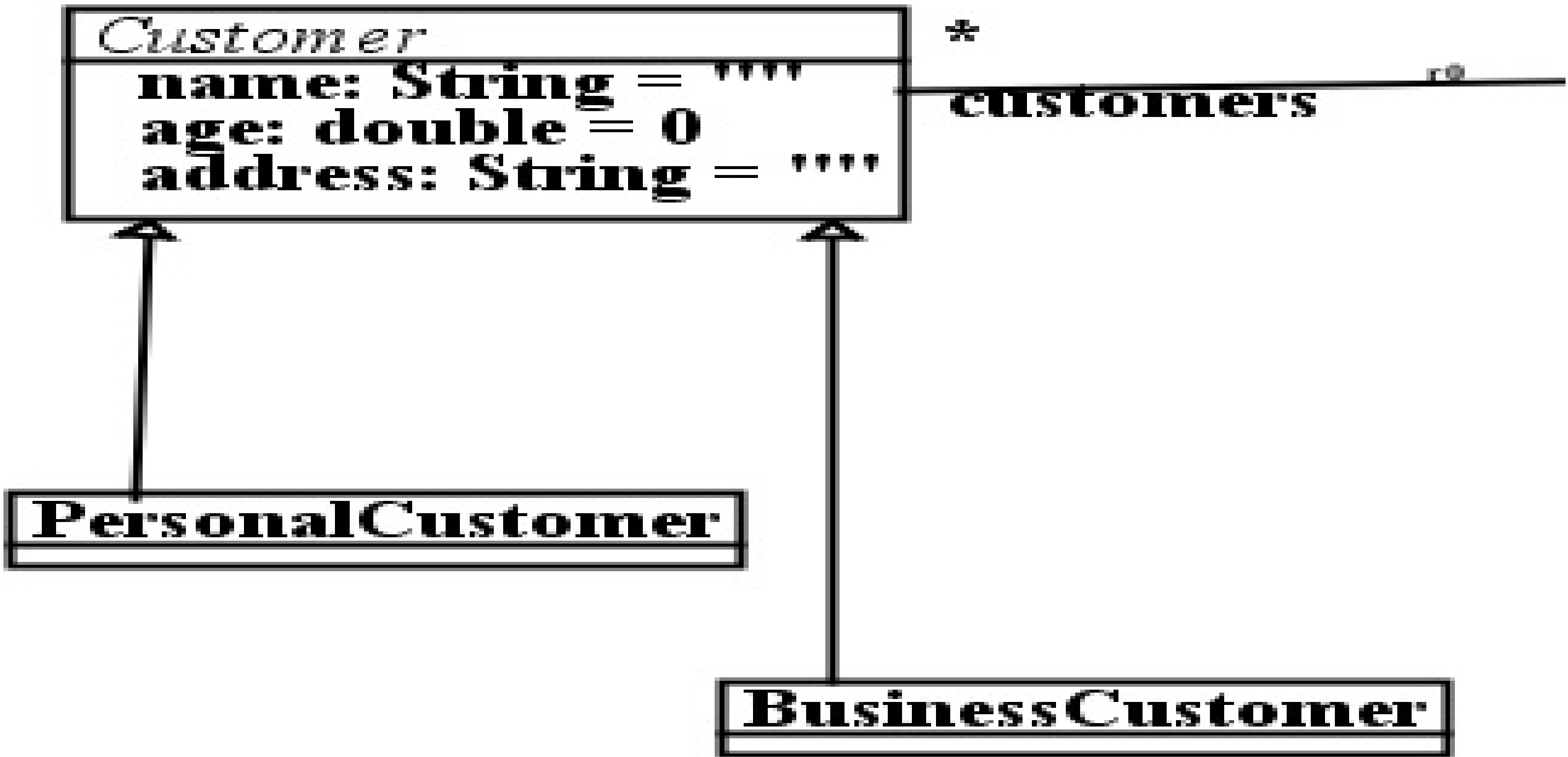
```
withdraw(amt : double)
pre: balance - amt  >=  -overdraftLimit
post:
  balance = balance@pre - amt
```

New value of balance expressed in terms of old value balance@pre.

Operation should only be invoked if $balance - amt \geq -overdraftLimit$. The *amt* is then subtracted from *balance*.

Operations and inheritance

- Classes may have several subclasses, eg., *Customer* has two subclasses, for two different types of bank customer, personal or business.
- One reason for introducing subclasses is if they have additional data or different behaviour from superclass. Business customers are taxed differently from personal, for example.
- Generally, should not have excessive numbers of subclasses, eg., a model with 100 subclasses of one class is probably wrong.
- Enumerations provide another way to distinguish special cases of objects, eg., kinds of *Account*.



Subclasses of Customer

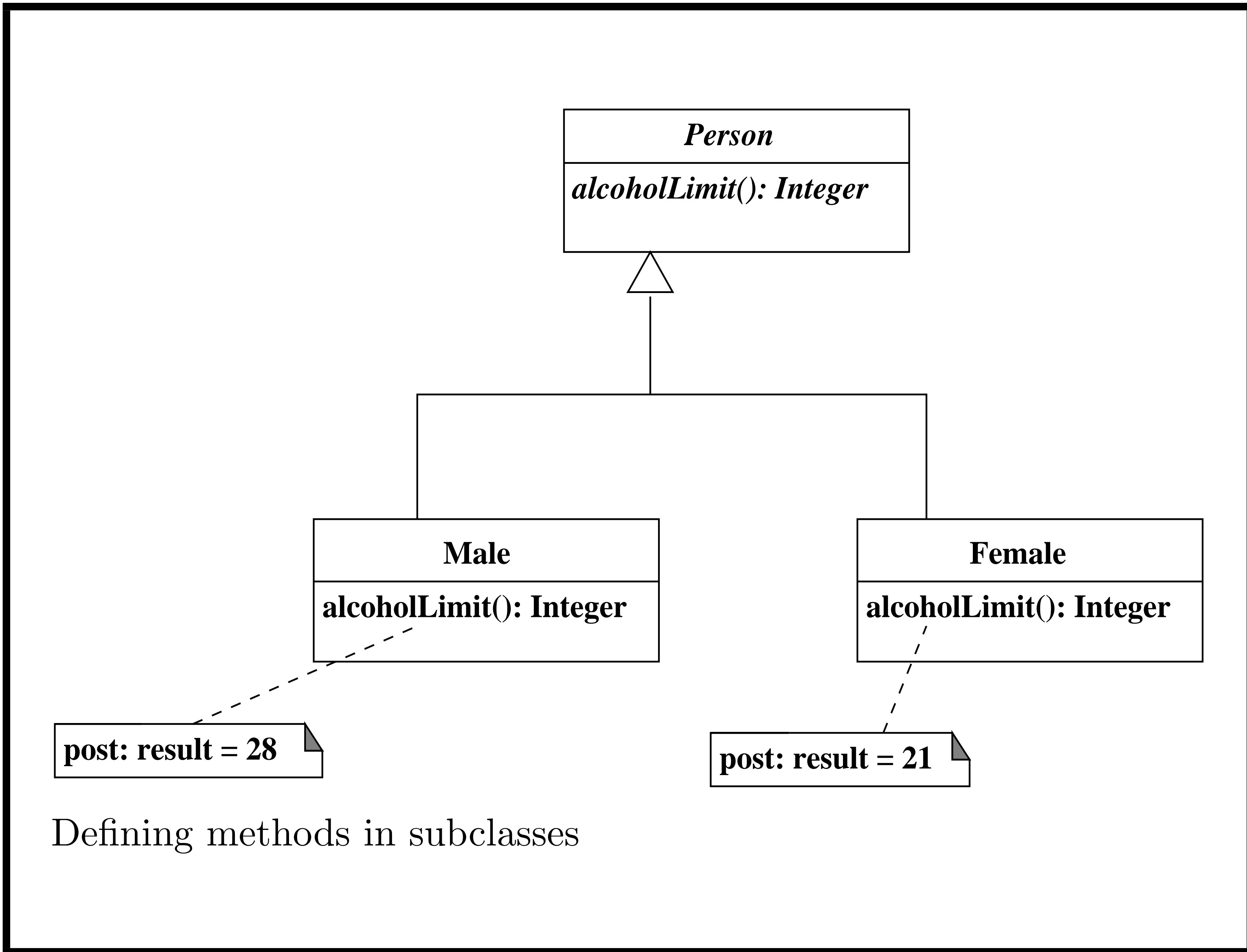
Abstract classes

- Because *every* customer must be one of these two kinds, superclass *Customer* itself has no direct instances. Called an *abstract* class, its name written in italic/slanting font to indicate this.
- Abstract classes cannot have direct instances created (cf., in Java, *new* $C(...)$ cannot be executed if C is abstract). Non-abstract classes are termed *concrete*.

Abstract operations

- Abstract superclasses may have *abstract operations* – non-executable operations – subclasses specify executable versions.
- Eg, superclass *Person* could have abstract operation $alcoholLimit() : Integer$ which returns weekly alcohol limit in units. In subclasses *Male* and *Female* operation can be given specific definitions: return 28 in first case and 21 in second.
- Abstract operations are written with italic font.

A class which has an abstract operation must itself be abstract.



Defining methods in subclasses

Inheritance of operations

- Operations can be specified in both a class and a subclass – subclass version *redefines* superclass version.
- Subclass version must be compatible with superclass version: same parameter and result types.
- For *Person*, operation $alcoholLimit() : Integer$ returns *some* (unspecified) integer. In subclasses *Male* and *Female*, operation versions specify particular integer results.

Supports *polymorphism/dynamic dispatch*: effect of $p.alcoholLimit()$ depends on class of p at *runtime*.

Static operations

- Operations usually apply to individual objects, refer to data of object in pre/postconditions/code: *instance-scope* operations.
- Possible to define operations independent of objects, called *static* or *class-scope* operations.

  Denoted by underlining declaration in class box. Invoked using class name: $C.op(p)$
- Same concept as in Java, likewise for *static* attributes.

Static operations cannot be redefined in subclasses.

Clients and Suppliers

If class A can call operations of class B via navigable association from A to B, then A is termed a *client* of B, and B is a *supplier* of A.

Exercise:

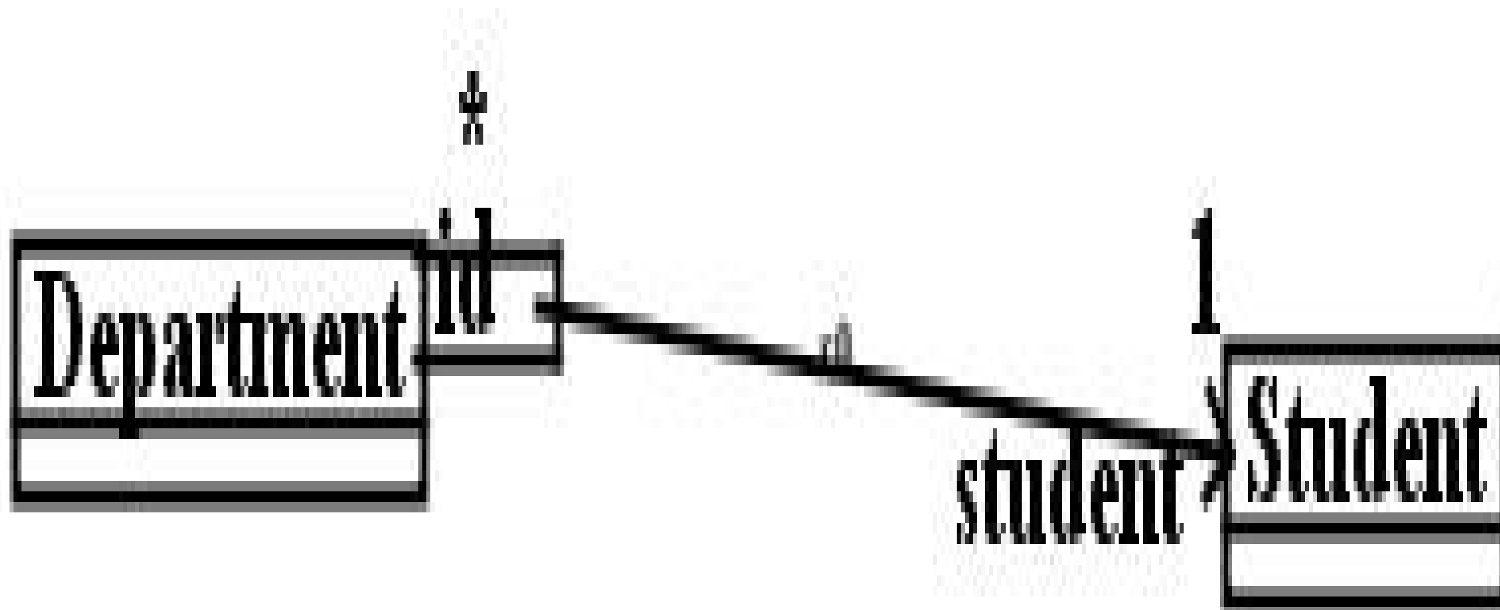
Experiment with creating simple classes with attributes, using the UML-RSDS tools (nms.kcl.ac.uk/kevin.lano/uml2web). Eg., create a *Person* class with $name : String$ and $age : int$.

Ordered associations

- End of an association may have annotation $\{ordered\}$ beside it, if multiplicity at end is not one.
- Annotation means that set of objects at role end is ordered, forming a *sequence*.
- First element of a sequence r is $r \rightarrow at(1)$, $r \rightarrow at(2)$ for second, etc.
- Also written as $r[1]$, $r[2]$, etc. in UML-RSDS.
- Numbering of sequences starts from 1 in UML, instead of 0 (in Java, C++, etc).

Qualified associations

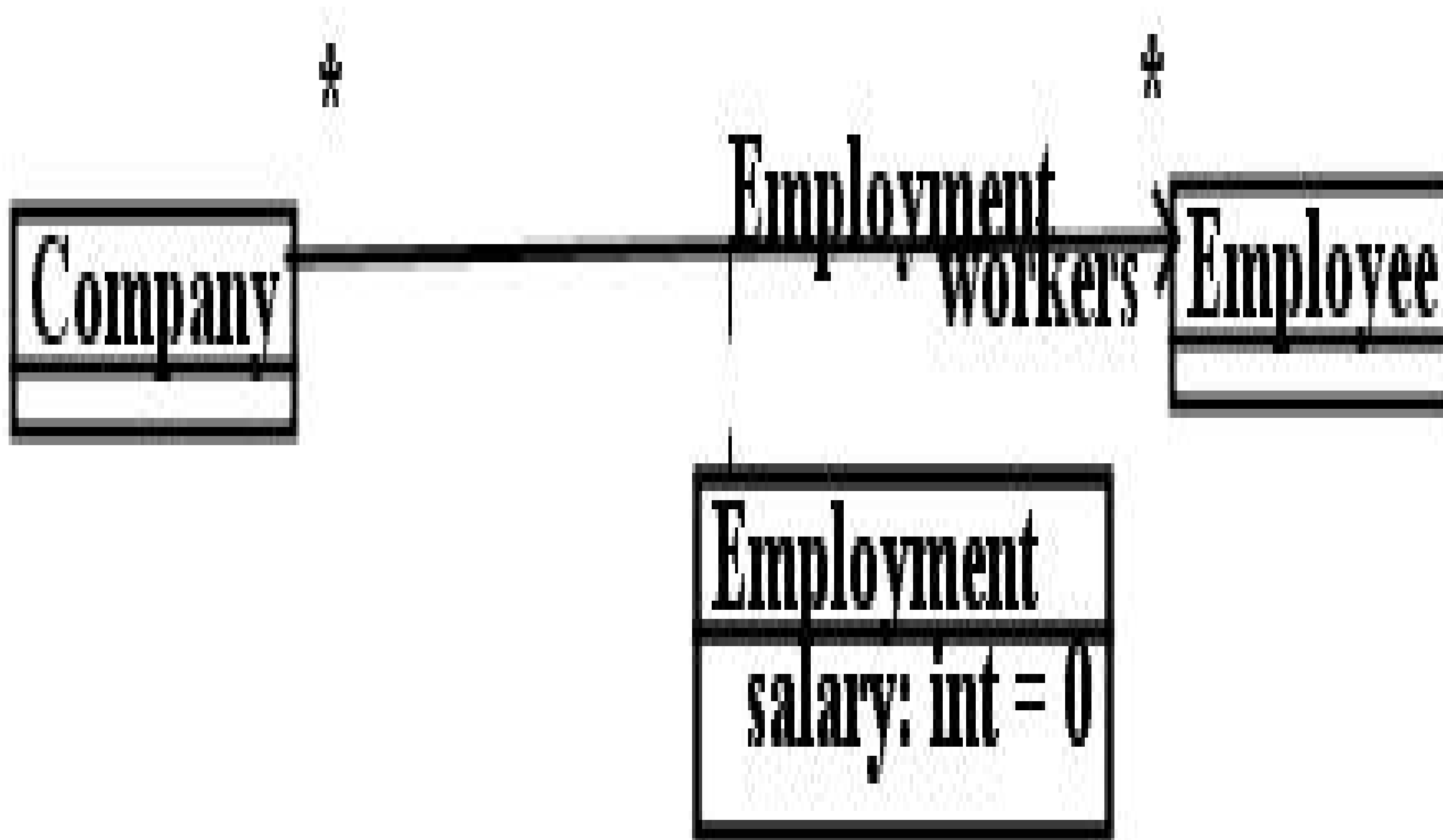
- Associations may also be *qualified* – these map from a value, and an object at one association end, to identify object or set of objects at other end. Eg., students within a department have unique student id.
- Given Department d, and id value $x : String$, the student with this id value is referred to as $d.student[x]$.
- Multiplicity of *student* role is 1 or 0..1 because at most one student in department can have a given id value.
- Correspond to Java Map, C++ map, etc.



Qualified association example

Association classes

- Association classes are associations *and* classes: have own set of attributes, operations, roles, in addition to participating in association.
- Association classes are denoted by separate association and class symbols, connected by dashed line.
- Eg.: each Employment $c \mapsto e$ links one Company c to one Employee e, has specific salary for this pair.
- Useful in situations where relationship has high significance + should be implemented by its own class. Eg., Marriage.



Association class example

Interfaces and implementation inheritance

- For modular development, use notion of *interface*.
- An abstract class which contains list of operations. Clients of interface invoke these operations, implementations of interface must implement them.
- Interfaces written as class rectangles with stereotype $\ll$ *interface* $\gg$.
- An interface cannot inherit from a class, but can be endpoint of an association from a class. All features of an interface must be public.
- **Interface is abstract definition of a module – list of signatures of its operations**.

Interfaces

- Concrete classes which implement interfaces, eg., *ArrayList* implementing *List* in Java, must provide particular implementations for each operation of all interfaces they implement.
- Such classes are said to *realize* their interfaces.
- An interface allows client class to make assumptions about what operations can be performed on object it accesses, without needing to declare precise concrete class for this object.
- $l.add(x)$ can be called on *any* kind of list l.

Example exam question

Draw a class diagram to specify the data of a railway network control system: this system records routes, which consist of a sequence of locations, each with a specific location number (unique in the network).

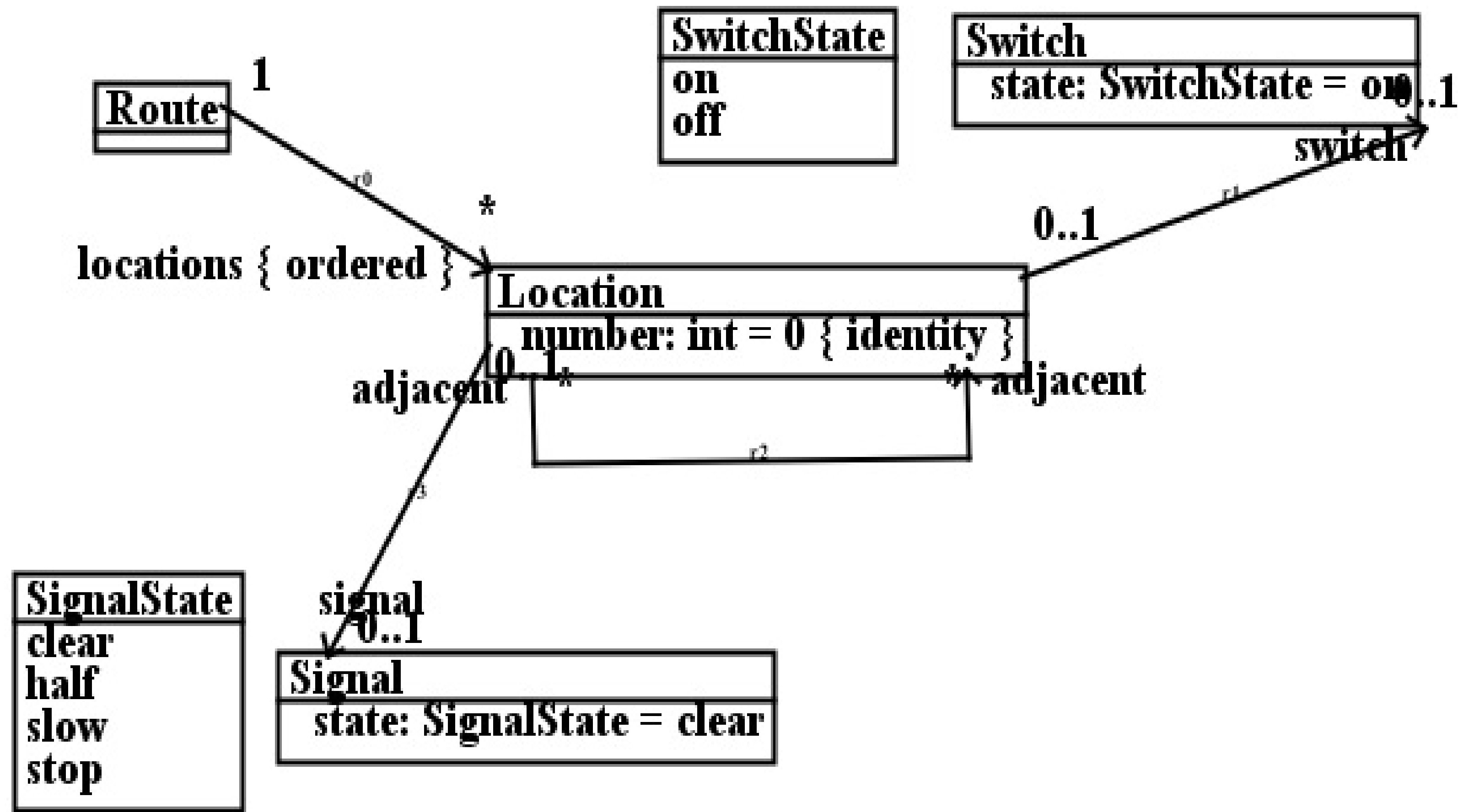
Locations may have associated equipment: a signal or a switch (possibly both). A signal has a state which is either clear, half-speed, slow, stop. A switch has a state which is either on or off. Locations may be adjacent to between 1 and 3 other locations.

Solution

Usually different possible solutions for any modelling problem. Solution should precisely express all stated information about the system.

Multiplicity of both ends of *adjacent* relation should be 1..3. Would also be valid to define abstract superclass *Equipment* of *Switch* and *Signal*.

Can a location be in several routes? If so, should be * at Route end of *Route—Location*.

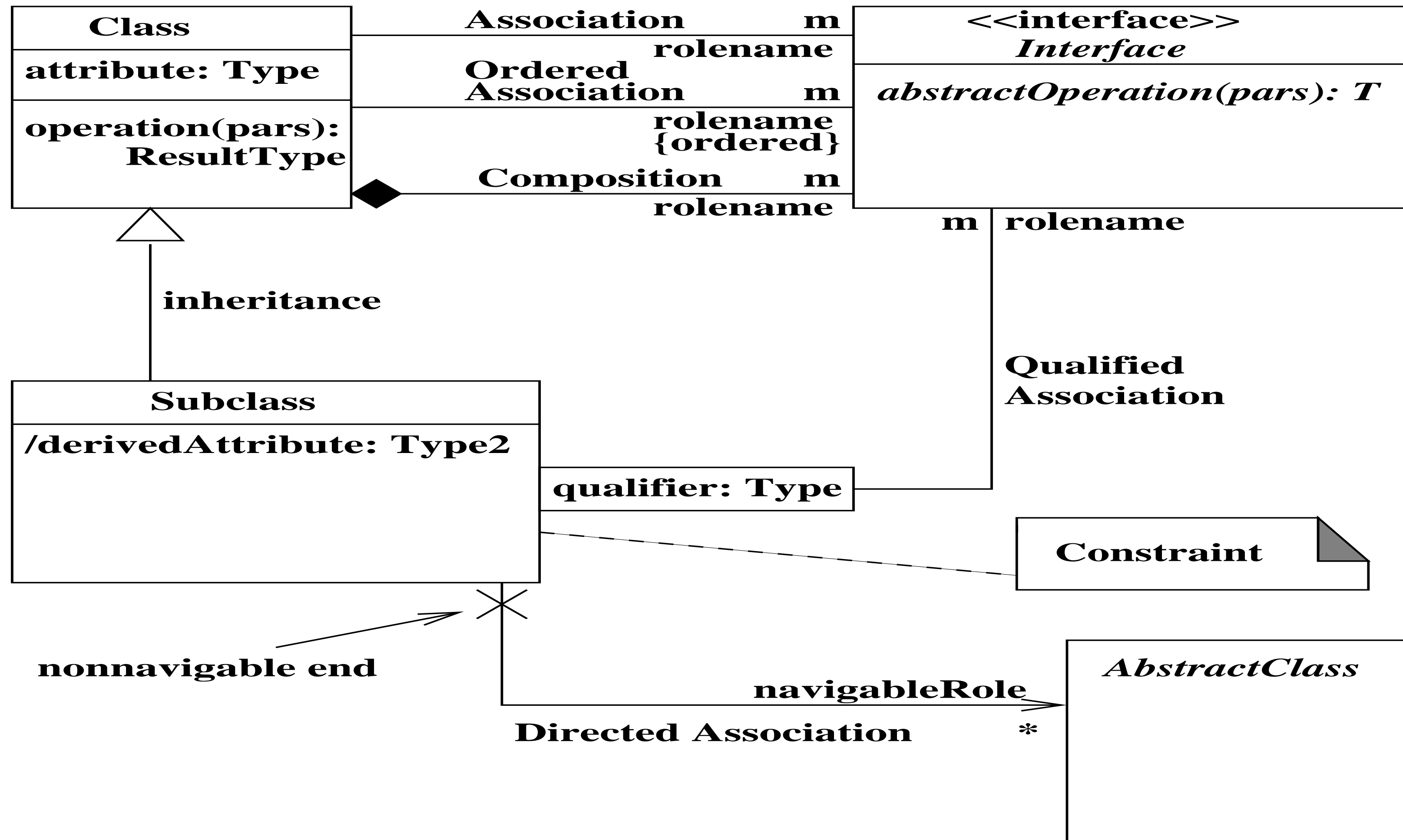


Train network class diagram

Other class diagram elements (not in this course)

There are many other class diagram notations in UML 2.*:

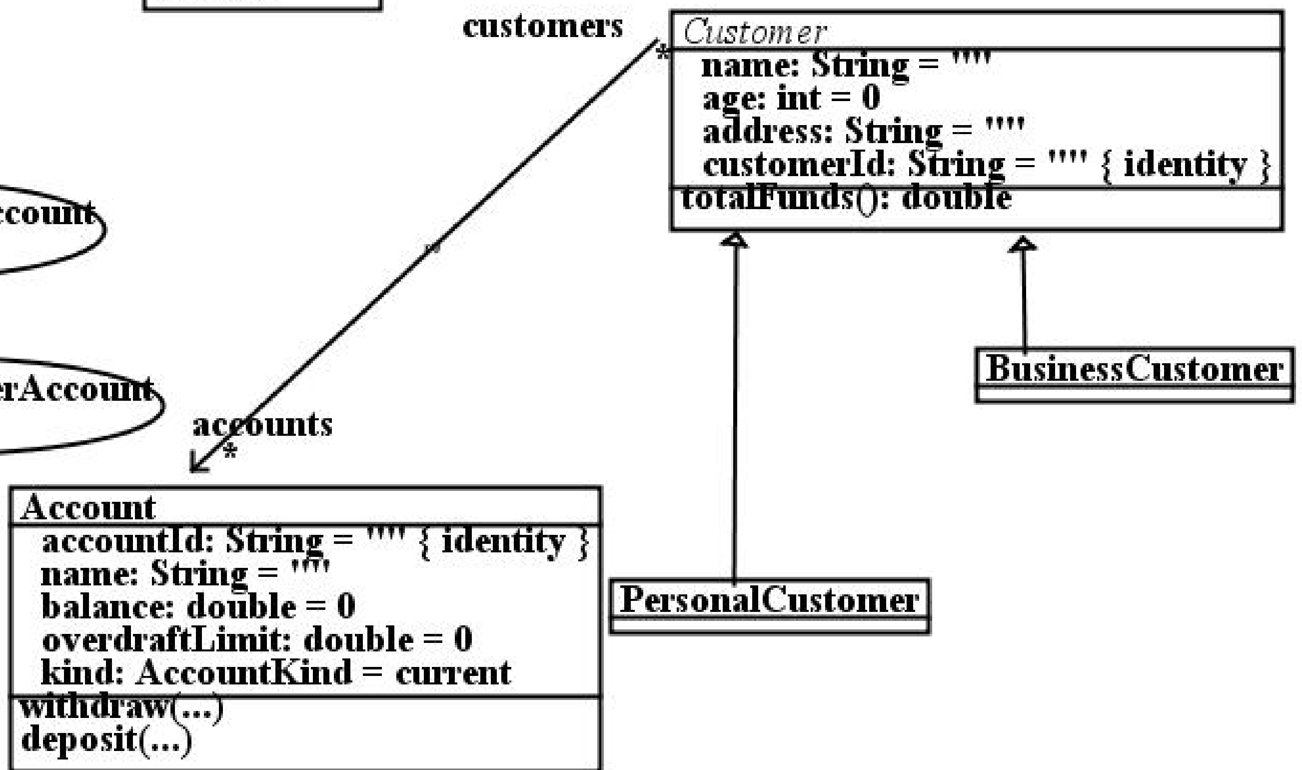
- Ternary and higher associations, between three or more classes, represented by association lines from diamond symbol to each of associated classes.
- Inheritance between associations, allowing a specialisation of an association to be defined, typically where one (or more) end of association is subclass of an end of original association and other ends are unchanged.
- Nested classes: classes defined within other classes.
- Templated classes, which carry a type parameter that can be used within class as if it were specific type, but which will be instantiated elsewhere.



Summary of UML class diagram notations (this course)

Combining use cases and class diagrams

- Use cases read + update data from class diagram, may call operations of classes
- Can show use cases on class diagram, without actors, to give complete system specification (data + functionality).



Completed banking system specification

Test case construction

Test cases can be written for completed system:

- Define objects, attribute and role values, and parameter values for use case tests
- At least one test for each scenario of each use case
- Identify expected results and effect of the test.

Test case construction

Test models can be written as, eg.:

```
a1 : Account
a1.accountId = "a1"
a1.balance = 100
a2 : Account
a2.accountId = "a2"
a2.balance = -66
c1 : PersonalCustomer
c1.customerId = "c1"
c2 : BusinessCustomer
c2.customerId = "c2"
a1 : c1.accounts
```

Test cases

For *checkBalance*(*aId*):

Scenario	*Test*	*Expected result*
No account with aId	*checkBalance*(“”)	Error message “No account”
	checkBalance(“*a*3”)	Error message “No account”
Valid account	*checkBalance*(“*a*1”)	100 displayed
	checkBalance(“*a*2”)	-66 displayed

Test cases

For $addCustomerAccount(cId, aId)$:

Scenario	*Test*	*Expected result*
No customer with cId	$addCustomerAccount$ (“”, “$a2$”)	Error message “No customer ”
No account with aId	$addCustomerAccount$ (“$c1$”, “$a3$”)	Error message “No account a3”
Valid customer, account, already linked	$addCustomerAccount$ (“$c1$”, “$a1$”)	Message “customer already has account” displayed
Valid customer, account, not linked	$addCustomerAccount$ (“$c1$”, “$a2$”)	Model updated with a2 : c1.accounts

OCL (Object Constraint Language)

- Expression language used with UML
- Defines logical conditions, for preconditions, postconditions, invariants, etc
- Defines values of numeric, string, entity or collection types
- Precisely defines operation and use case functionalities.

Constraints visually expressed on diagrams as ‘dog-eared’ boxes (notes).

Class invariants: constraint $balance \geq -overdraftLimit$ on Account.

Preconditions: constraint $pre : balance - amt \geq -overdraftLimit$ on $withdraw(amt : double)$.

Postconditions: constraint

$$post : \quad balance \quad = \quad balance\texttt{@pre} \quad - \quad amt$$

on withdraw defines effect of operation.

Can add constraints to examples shown previously. Eg, to assert that each person has same employer as their boss:

Employment class diagram with class invariant constraint

OCL (Object Constraint Language)

- Numeric value types *Integer*, *Real*. String value type *String*
- Usual numeric operators +, -, *, /, $<$, $>$, $\leq$, $\geq$, etc
- Numeric functions $r.sqrt()$, $r.cos()$, $r.pow(p)$, etc
- String functions $s.size()$, + (concatenation), $s.toLowerCase()$, etc.

OCL (Object Constraint Language)

Entity types:

- If E is a class diagram entity type, instances $e : E$ can be used in OCL expressions, and features $e.att$, $e.role$. Objects can be compared with $=$, $/=$ (also written as $<>$)
- A constraint with *context* E can refer directly to features of E, eg.:

  ```
  Account::
      balance >= -overdraftLimit
  ```

  as invariant of *Account*
- *self* object.

OCL numeric types and operators

- Instead of OCL Integer and Real, we use `int` (32 bit signed integers) and `double` which correspond to types in Java, C, C++ and C#. `long` (64-bit integers) also available.
- All operators take double values as arguments except as noted.
- Three operators: ceil, round, floor, take a double and return an int.

Expression	*Meaning as query*
-x	unary subtraction
x + y	numeric addition, or string concatenation (if one of x, y is a string)
x - y	numeric subtraction
x * y	numeric multiplication
x / y	integer division (div) if both x, y are integers, otherwise arithmetic division
x mod y	integer x modulo integer y
x.sqr	Square of x
x.sqrt	Positive square root of x
x.floor	Floor integer of x

Expression	*Meaning as query*
x.round	Rounded integer of x
x.ceil	Ceiling integer of x
x.abs	Absolute value of x
x.exp	e to power x
x.log	e-logarithm of x
x.pow(y)	y-th power of x
x.sin	sine of x (given in radians)
x.cos	cosine of x (given in radians)
x.tan	tangent of x (given in radians)
Integer.subrange(st,en)	Sequence of integers starting at st and ending at en, in order

OCL numeric types and operators

- Other math operators in common between Java, C# and C++ are: log10, cbrt, tanh, cosh, sinh, asin, acos, atan. These are double-valued functions of double-valued arguments.
- Some functions may not be available in old versions of Java (log10, cbrt, tanh, cosh, sinh).
- Math functions can also be written as $argument{\rightarrow}sqrt()$, $x{\rightarrow}pow(y)$, etc.

Other operators

Expression	*Meaning as query*	*Meaning as update*
$x : E$ E entity type	x is an existing instance of E	Create x as a new instance of E (for concrete E)
$x : s$ s collection	x is an element of s	Add x to s
`s->includes(x)` s collection	Same as $x : s$	Same as $x : s$
$x \; / : s$ s collection	x is not an element of s	Remove x from s
`s->excludes(x)` s collection	Same as $x \; / : s$	Same as $x \; / : s$

Expression	*Meaning as query*	*Meaning as update*
$x = y$	x's value equals y's value	Assign y's value to x
$x < y$ x, y both numbers, or both strings	x's value is less than y's. Likewise for $>$, $<=$, $>=$, $/=$ (not equal)	
$s <: t$ s, t collections	All elements of s are also elements of t	Add every element of s to t
`t->includesAll(s)`	Same as $s <: t$	Same as $s <: t$
`t->excludesAll(s)` collections s, t	No element of s is in t	Remove all elements of s from t

Collections

Two main kinds of collection in OCL: sets and sequences.

- Sets are *unordered* collections with unique membership (duplicate elements not permitted), sequences are *ordered* collections with possibly duplicated elements. Indexes start at 1 in OCL.
- If br is unordered * role of class A, $ax.br$ is a *set* for each $ax : A$.
- If br is an ordered * role of class A, $ax.br$ is a *sequence* for each $ax : A$.
- If $objs$ is set of A objects, $att : T$ an A attribute, then $objs.att$ is a set of T values (no duplicate values).

(NB., in exam necessary OCL operators will be given).

Expression	*Meaning as query*
$Set\{\}$	empty set
$Sequence\{\}$	empty sequence
$Set\{x1, x2, ..., xn\}$	set with elements $x1$ to xn
$Sequence\{x1, x2, ..., xn\}$	sequence with elements $x1$ to xn
`s->including(x)`	s with element x added
`s->excluding(x)`	s with all occurrences of x removed
`s - t`	s with all occurrences of elements of t removed
`s->prepend(x)`	sequence s with x prepended as first element
`s->append(x)`	sequence s with x appended as last element
`s->count(x)`	number of occurrences of x in sequence s
`s->indexOf(x)`	position of first occurrence of x in s
$x\bigvee y$	set union of x and y

Expression	*Meaning as query*
$x /\backslash y$	set intersection of x and y
$x \frown y$	sequence concatenation of x and y
`x->union(y)`	Same as $x \backslash/ y$
`x->intersection(y)`	same as $x /\backslash y$
`x->any()`	Arbitrary element of non-empty collection x
`x->front()`	front subsequence of sequence x
`x->tail()`	tail subsequence of sequence x
`x->first()`	first element of sequence x
`x->last()`	last element of sequence x
`x->sort()`	sorted (ascending) form of sequence x
`x->sortedBy(e)`	x sorted in ascending order of y.e for y : x

Expression	*Meaning as query*
`x->sum()`	sum of elements of collection x
`x->prd()`	product of elements of collection x
`x->max()`	maximum element of collection x
`x->min()`	minimum element of collection x
`x->asSet()`	set of distinct elements in collection x
`x->asSequence()`	collection x as a sequence
`x->isDeleted()`	destroys all elements of collection x.
	Can also be used on individual objects x

Language-independent versions of operators such as *addAll* in Java, *AddRange* in C# and *std* :: *set_union* in C++.

OCL (Object Constraint Language)

Collection types:

- $Set(T)$ is type of sets of T: a set $Set\{v_1, ..., v_n\}$ is an *unordered* collection of elements, with no duplicates (each element occurs only once)
- $Sequence(T)$ is type of sequences of T: a sequence $Sequence\{v_1, ..., v_n\}$ has elements in the listed order. Elements can occur multiple times.
- Eg., $Set\{1, 9, 9, 1\}$ only has 2 elements: $Set\{1, 9\}$, whilst $Sequence\{1, 9, 9, 1\}$ has 4 elements.

Set of all instances of a class E is $E.allInstances()$. Changes over time as instances are created/deleted.

OCL (Object Constraint Language)

Collection operators use $\rightarrow$ symbol:

- $s \rightarrow size()$ is size of collection s
- $s \rightarrow sum()$ is sum of elements of collection s (of numbers/strings)
- $s \rightarrow prd()$ is product of elements of collection s (of numbers)
- $x : s$ is true if s contains element x, false otherwise. Also written as $s \rightarrow includes(x)$
- Eg., $9 : Set\{1, 9, 9, 1\}$
- Eg., $Sequence\{1, 9, 9, 1\} \rightarrow sum() = 20$

OCL (Object Constraint Language)

Collection operators:

- $s \rightarrow collect(e)$ is collection of values $x.e$ for x in collection s
- $s \rightarrow select(x \mid P)$ is subcollection of collection s, containing those $x : s$ that satisfy P
- $s \rightarrow reject(x \mid P)$ is subcollection of collection s, containing $x : s$ that do not satisfy P
- Eg., $accounts \rightarrow collect(balance)$ is sequence of *balance* values for a customer's accounts
- Eg., $accounts \rightarrow select(balance \geq 0)$ are the accounts that are not overdrawn.

Operators can be chained, eg: $accounts \rightarrow collect(balance) \rightarrow sum()$ is sum of balances of accounts of a customer.

OCL (Object Constraint Language)

$objs \rightarrow collect(e)$ is collection of all values of $x.e$ for x in $objs$.

$s \rightarrow sum()$ corresponds to std::accumulate(s.begin(), s.end(), 0) in C++ STL. $s \rightarrow collect(e)$ corresponds to std::transform(s.begin(), s.end(), result.begin(), e).

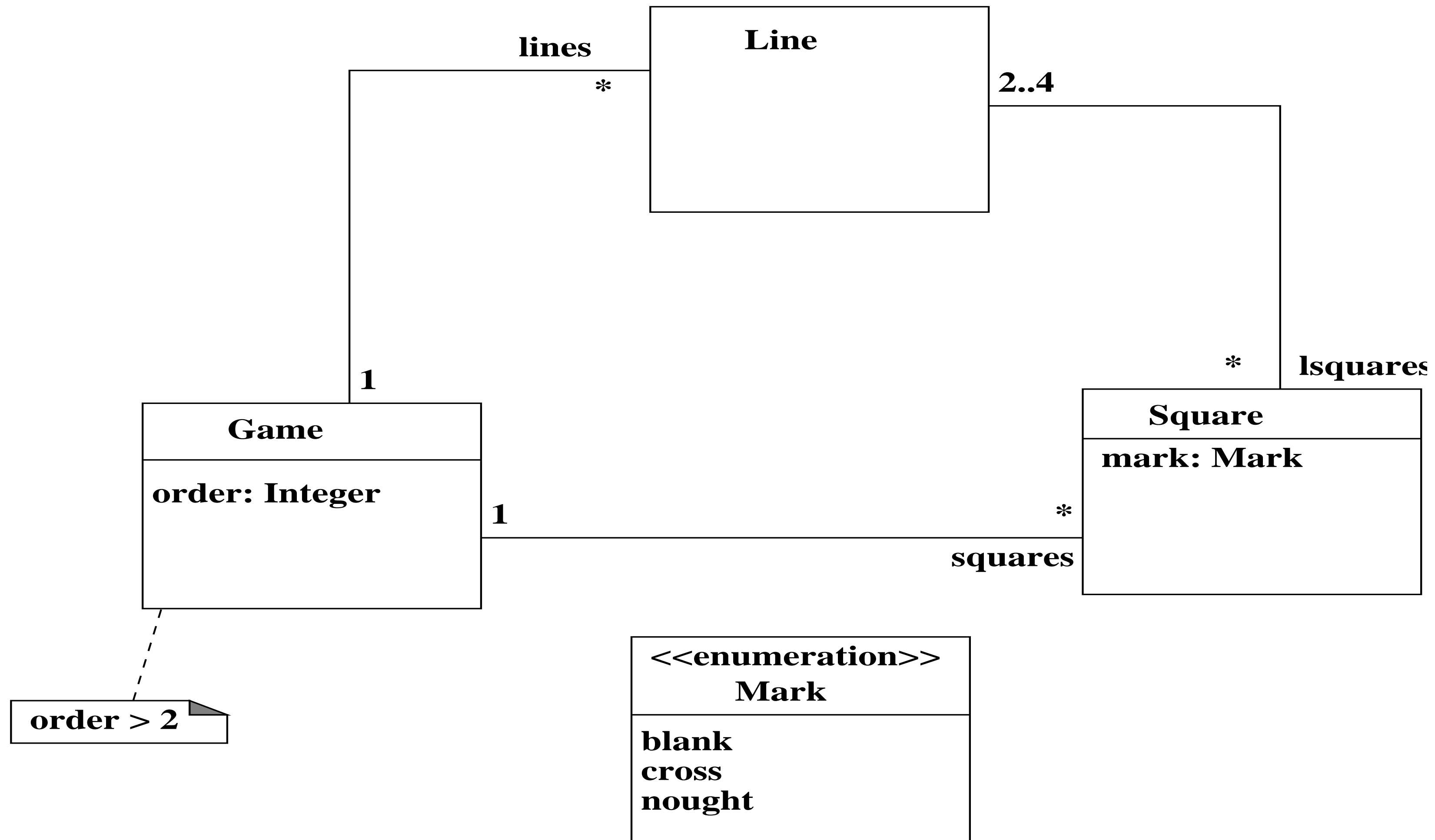
Navigation expressions

Can use "." symbol to navigate around a class diagram. Eg., $c.accounts.balance$ is set of *balance* values of account objects a in $c.accounts$.

Following rules determine whether result of navigation expression $e.f$ is a set or sequence:

e	feature f	$e.f$
single object	attribute	single value
	1-multiplicity role	single object
	set-valued role	set of objects
	sequence-valued role	sequence of objects
set of objects	attribute	set of values
	1-multiplicity role	set of objects
	set-valued role	set of objects
	sequence-valued role	set of objects
sequence of objects	attribute	sequence of values
	1-multiplicity role	sequence of objects
	set-valued role	set of objects
	sequence-valued role	sequence of objects

Best to avoid long chained navigations, as difficult to understand.



Specification of (general) noughts/crosses game

Navigation examples from Noughts-and-crosses specification

$$g.lines.lsquares$$

is navigation through two unordered roles from $g : Game$ object – returns set, should be collection of all existing squares:

$$g.lines.lsquares = Square.allInstances()$$

(also $= g.squares$).

At start of game, all squares are blank:

$$g.squares.mark = Set\{blank\}$$

Identity attributes

- If attribute *att* has {*identity*} beside declaration, means no two different objects of class can have same value for this attribute.
- Attribute values can be used as unique identities for objects of class.
- Eg, should not be two different *Account* objects with same *accountId* value.
- If class C has identity attribute *key* then

$$a : C \ \& \ b : C \ \& \ a.key = b.key \ \Rightarrow \ a = b$$

C

key : T {identity}
att1: T1
...
attn: Tn

Class with identity attribute

Identity attributes

- Identity attributes provide useful way to refer to objects.
- For identity attribute $key : T$ of class C, set

  $$C \rightarrow select(key = val)$$

  always has 0 or 1 element, for each val in T.
- Can write this set/object as $C[val]$.
- Identity attributes correspond to primary keys in databases. Can have several identity attributes, for alternative lookup.

OCL (Object Constraint Language)

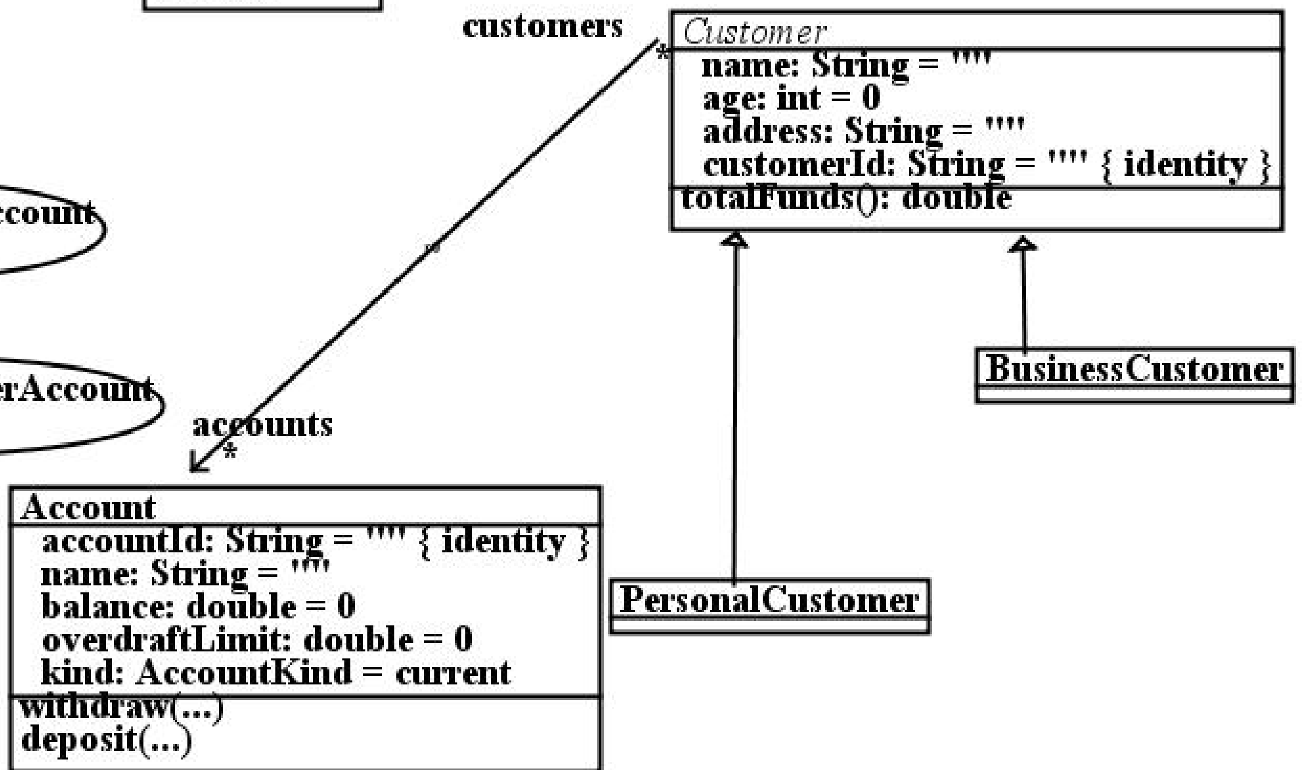
Quantifiers:

- $s \rightarrow forAll(x \mid P)$ is true if *every* element x of collection s satisfies P, false otherwise
- $s \rightarrow exists(x \mid P)$ is true if *some* element x satisfies P, false otherwise
- Eg., $Set\{1, 9, 9, 1\} \rightarrow forAll(x \mid x \leq 10)$ is true.

OCL (Object Constraint Language)

Expressions can define behaviours:

- $x = v$ can be interpreted as "Set value of x to v"
- $x : s$ can be interpreted as "Add x to s"
- $s \rightarrow forAll(x \mid P)$ as "Make P true for every element x of s"
- $E \rightarrow exists(x \mid P)$ for concrete entity type E as "Create an instance x of E and initialise it to satisfy P"



Banking system with use cases

OCL (Object Constraint Language)

Use case postconditions/effects:

- Eg., *checkBalance(aId : String)*:

```
aId : Account->collect(accountId)  =>
        Account[aId].balance->display()

aId /: Account->collect(accountId)  =>
        ("No account with id = " + aId)->display()
```

- *newAccount(aId : String)*:

```
true  =>
    Account->exists( a | a.accountId = aId )
```

If an account already exists with *aId*, the $\rightarrow$*exists* simply looks it up, does not modify/recreate it.

UML/Drawing Tools

- UML-RSDS: nms.kcl.ac.uk/kevin.lano/uml2web
- MagicDraw: www.nomagic.com/products/magicdraw.html
- Papyrus (in Eclipse): https://www.eclipse.org/papyrus
- IBM Rational Rhapsody: ibm.com/software/products
- Visio (Microsoft Office).

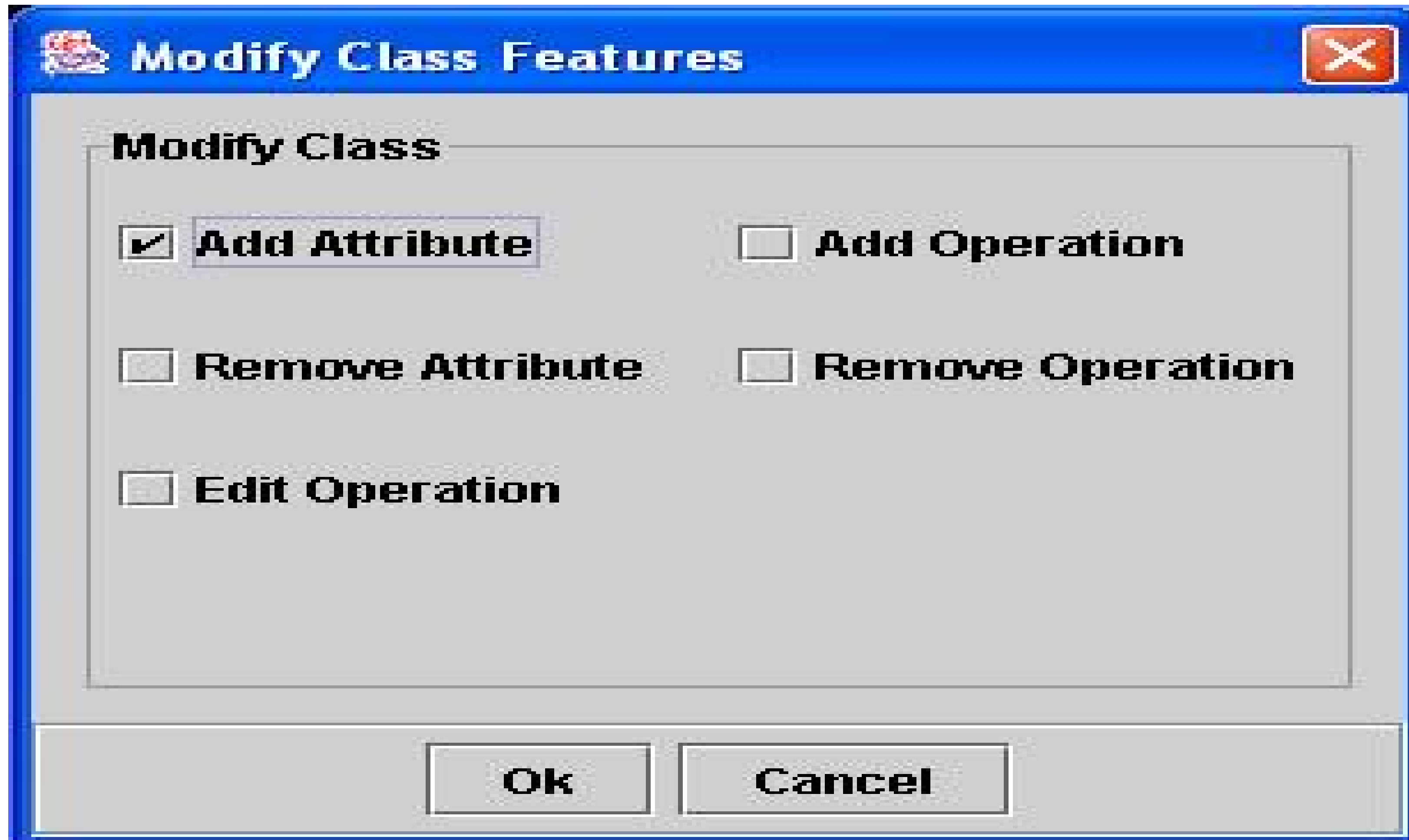
For coursework, recommended to use some UML tool to draw diagrams + produce code.

Using UML-RSDS

- Download umlrsds17.zip, extract files
- Type `java -jar umlrsds.jar` on command line in extracted umlrsds17 directory
- Create classes, associations, etc.
- To edit a class, click on it, or use *Edit* menu *Modify* option
- Associations are created by selecting *Create Association* and dragging mouse from source to target class. *Role*2 field must be filled in – end to which association is navigated. *Role*1 may be left blank.

Using UML-RSDS

- *Save* saves current model in output/mm.txt; *Generate Design* creates design; *Build* options generate code.
- File menu option *Recent* loads system model from output/mm.txt
- Code is placed in *output* subdirectory. Compile Controller.java and GUI.java here.
- Errors in compilation usually indicate errors in the specification – change the specification, not the code!



Edit class dialog

Edit Attribute Properties

Name: x

Type: int

Initial value:

Kind

Sensor Internal Actuator Derived

Updateability

Frozen Modifiable

Scope

Class Instance

Uniqueness

Unique Not unique

Ok Cancel

Attribute definition dialog

Edit Association Properties

Name: r0

Role1: husband

Role2: wife

Card1: 0..1

Card2: *

Ordering of role2

☑ Unordered ☐ Ordered ☐ Sorted < ☐ Sorted >

Updateability of role2

☐ Frozen ☑ Modifiable

role2 add only

☑ Not add only ☐ Add only

Stereotypes: none

Ok Cancel

Association definition dialog

Summary of Part 2

- Introduced essential UML class diagram notations
- Introduced use case concepts
- Introduced core OCL features and uses