Course: Software Engineering

Unit 5: Aggregation and Inheritance

Aims and Objectives

This unit aims to:

- Define the specification in Object-Z of more elaborated object-oriented systems built upon
  - Objects aggregation
  - Inheritance
- Illustrate these aspects by considering the example of a banking system that further extends the example credit card accounts system seen in Unit 4.

So far we have seen how to specify individual objects using class schema, and looked in detail at the main components of a class schema in the Object-Z specification language.

We have then increased the complexity level of our object-oriented specification. We have showed how to define in Object-Z concepts like object instantiations (using object identifiers), object interactions (using conjunction and choice operators in operation expression), and inter-object communication (using the parallel operator in operation expressions). These operation expressions allow the definition of composite operations in a given class in terms of operations of objects of other different classes. We have concluded the previous lecture with an example of specification of replacement of object identifier. One of the main key points of a class-based specification is the distinction between an object and its identifier. Identifiers of single objects do not normally change unless some replacement operation is required. Object states instead change each time one of its operations is applied to it. Object identifiers therefore rarely appear in the Δlists of the operations of a class schema.

In this lecture we'll further increase the level of complexity of our class-based specifications by introducing the formalization of more elaborate object-oriented features such as object aggregations, and inheritance.

We'll extend the example of credit card accounts system seen in Unit 4, by specifying a banking system that includes an arbitrary collection credit card accounts, and various operations on this collection.

Course: Software Engineering

Example: A Banking System

Making many simplifying assumptions:

- The system is an aggregate of (set of) credit card account objects.
- All credit card objects have limit equal to fixed *commonlimit*.
- At the beginning, the aggregate is empty.
- New initialised account object can be added to the aggregate.
- Any account object in the aggregate can be deleted.
- Each account object can independently perform *withdraw*, *deposit*, *withdrawAvail* operations.
- The system shall allow for any given two account objects in the aggregate to transfer the available fund of the first account into the second account.

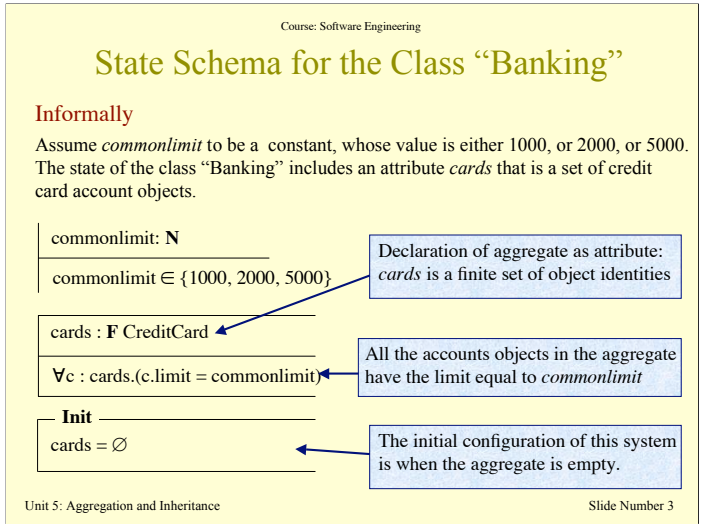
This slide gives the informal requirements of the banking system that we are going to specify.

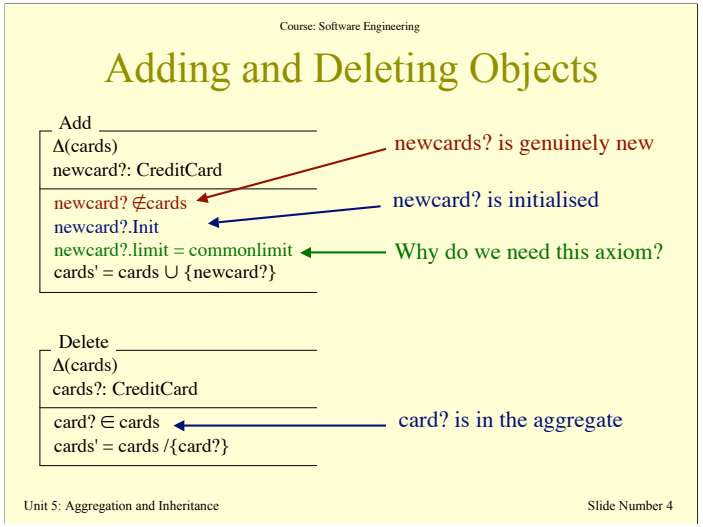
In particular our banking system, called ***banking***, includes a collection of objects of type credit card specified in the Unit 4. We require, as constraint of this system, that all the credit card accounts have the same limit of permitted overdraft. This is an unrealistic constraint, but we are considering it here in order to illustrate particular types of expressions.

The number of credit card accounts in the aggregate can vary. Specifically, we can add a new credit card account object or delete an account object that already exists in the aggregate.

The system shall allow each account object in the aggregate to perform its operations of withdraw, deposit and withdrawAvail, without any additional constraint. In addition the system shall provide an operation, called "*transferAvail*", that takes any two given credit card account objects existing in the aggregate and specifies the transfer of the available fund from the first account into the second account.

We are going to specify this system by defining a class schema called "Banking, and we'll discuss the new notations and constructs that we are going to introduce in this schema.





The definition of aggregate of credit card account objects is given by the declaration of the variable *cards* in the state schema shown here. The attribute *cards* is then an attribute that denotes an aggregate of objects. Specifically, the sort "**F** CreditCards" defines the set of all finite subsets constructed from the type CreditCard. Each element of "**F** CreditCards" is a finite set of credit card account objects identities. (Remember that here CreditCard is used as type not as class, so it refers to the sort of all possible identity values for credit card account objects.) Hence the attribute *cards* denotes a finite set of credit card account object identities.

The axiom given in the state schema specifies the system requirement that all the account objects in the aggregate must have the same limit. To refer to individual objects in the aggregate cards we can either use quantifiers (if we want to say "forall " or "there exist"), or just variables over the sort given by the same aggregate (e.g. *cards* in this case). So the axiom given in this state schema reads as "forall the account object identities that are in the aggregate (i.e. for all the account objects in the aggregate) the limit of these objects is equal to *commonlimit*. Note that from a logical perspective we could also have written this axiom in the following way:

$$\forall c: \text{CreditCard}.\ (c \in cards \rightarrow c.limit = commonlimit)$$

But Object-Z relaxes this mathematical definition by using *cards* as sort directly.

Finally, as required in the informal description of the system given in the previous slide, the initial schema of the class "Banking" includes the axiom cards = ∅ to specify that initially the aggregate cards is empty.

In this slide the two operations of adding and deleting an account object from the aggregate are specified.

In the case of adding a new account object, we need to make sure that (i) the object given in input to the operation is genuinely new and (ii) that it's in its initial configuration, since the requirement is to add a new initialised account object to the aggregate. The last axiom defines how the new aggregate should look like after the operation. What is important to notice is the third axiom. This says that the new account object should have its limit equal to the fixed *commonlimit* for the operation Add to be applicable. It's therefore like a pre-condition of the operation Add. This pre-condition could also have been inferred from the class invariant, i.e the axiom of the state schema applied to cards'. So the third axiom is in effect redundant, but it's good practice to include this redundancy for later stage in the software development. So, we have included this pre-condition in the operation schema: it adds just as additional observation for those who will use such specification to implement the system.

The schema Delete is quite straightforward.

Selecting Objects from Aggregates

Informally

The system has to perform the operations of withdraw, deposit and withdrawAvail on the individual account objects that are in the aggregate.

We need to promote these object's operations to operations of the Banking system. This requires a formal notation for selecting the object from the aggregate to which the operation is applied.

withdraw ≙ [card? : cards]. card?.withdraw
deposit ≙ [card? : cards]. card?.deposit

Selection (from the environment) of a particular object in the aggregate to which the operation is applied.

Only the state of the object identified by card? is changed in the aggregate

Selecting more than one Object

Informally

The system has to allow a "transferAvail" operation to take place between two given credit card account objects. The fund available has to be transferred from the first object to the second object.

transferAvail ≙ [from?, to? : cards | from? ≠ to?].
from?.withdrawAvail || to?.deposit

Selection from the set cards of two distinct credit card account objects. These are selected by input from the environment.

A final example of selection of objects from an aggregate is the selection of more than one object at the same time. In this case we want to specify that the Banking system can transfer available funds from a given account object of the aggregate into another different account object of the aggregate. These two account objects are assumed to be given to the system as input variables.

In this case the input variables from? and to? are identities of the two account objects: all the available amount has to be withdrawn from the object identified by from? and deposited into the account object identified by to?. Note that the definition of the selection is now given by

[from?, to? : cards | from? ≠ to?]

In Object-Z we can have very elaborated selection definition before even writing which operation should be applied to the selected object(s). We only limit ourselves to the examples given in this lecture.

The "Banking" Class Schema

Banking

| (commonlimit, Init, Add, Delete, withdraw, deposit, withdrawAvail, transferAvail)

commonlimit: **N**

commonlimit ∈ {1000, 2000, 5000}

cards : **F** CreditCard

∀ c : cards. c.limit = commonlimit

Init

cards = ∅

Delete

Δ(cards)
card? : CreditCard

card? ∈ cards
cards' = cards / {card?}

Add

Δ(cards)
card? : CreditCard

card? ∉ cards
card?.limit = commonlimit
card?.Init
cards' = cards ∪ {card?}

withdrawAvail = [card?:cards].card?.withdrawAvail

transferAvail = [from?:cards, to?:cards | from? ≠ to?].
from?.withdrawAvail || to?.deposit.

withdraw = [card?:cards]. card?.withdraw

deposit = [card?:cards]. card?.deposit

Inheritance

In object-oriented programming language:
code of existing classes can be reused in coding new classes.

In Object-Z formal specification:

» specifications can be developed incrementally
existing Object-Z classes can be reused to define new Object-Z classes

» architectural design aspects of the future system are captured
inheritance hierarchy of class schemas expresses the class inheritance hierarchy of the future implementation

» inheritance can be

single: a "subclass" inherits just one other class

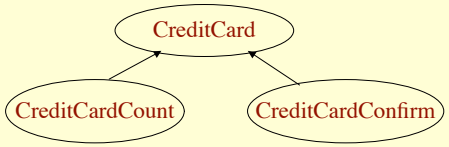
multiple: a "subclass" inherits more than one other class

In object-oriented programming, inheritance allows the code of existing classes to be reused in the coding of a new class. In class-based specifications, inheritance plays also a similar role. It provides a mechanism for incremental development of object-oriented specification, whereby features (attributes, operations, and Init schema) of existing Object-Z classes can be reused when creating new Object-Z classes. In these next few slides we explore this inheritance mechanism.

The inheritance feature of a class-based formal specification language captures, to some extend, the definition of architectural design decisions that are considered in the implementation of the system. The inheritance hierarchy of Object-Z class schemas can be seen, in fact, as a direct definition of the inheritance hierarchy of the classes to be defined in the implementation of the system. In the hierarchical structure of class schemas, the class schema that is defined reusing an existing class schema, is called "subclass" and the existing (used) class schema is called "superclass". This is also the terminology adopted in defining inheritance in object-oriented programming languages.

Inheritance in Object-Z specifications can be of two types: single inheritance when the subclass inherits just one other class, and multiple inheritance when the subclass inherits more than one other class. We will consider only the case of single inheritance, which corresponds to standard inheritance of classes in object-oriented programming languages. Multiple inheritance, known also as polymorphism, is not covered in this course.

Example: Specialising the Class CreditCard



- CreditCard class schema is as defined in Unit4
- CreditCardCount class schema refines CreditCard class schema
  - » the operation withdraw has also to count the number of withdrawals made
- CreditCardConfirm class schema extends CreditCard class schema
  - » it also includes an operation that first performs the withdraw operation and then confirms it by producing as output the value of the remaining funds.

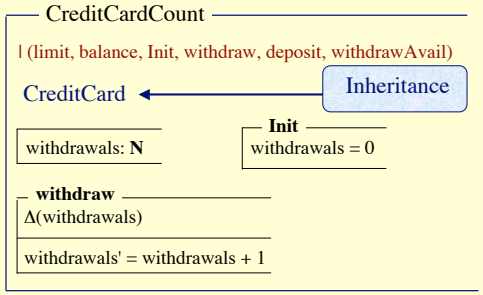
We consider two examples of single inheritance: one in which the subclass refines the superclass, the other in which the subclass extends the existing superclass. In particular we see how to define, using inheritance, two new classes, called CreditCardCount and CreditCardConfirm that are both subclasses of the already existing single superclass CreditCard.

The CreditCardCount specifies a particular type of CreditCard account object, which has the same features of the class CreditCard, with the exception that the operation "withdraw" of CreditCard has also to keep a count of the number of withdrawals made from the account. So if you like the class CreditCardCount is a refinement of the class CreditCard, in that the operation withdraw of the class CreditCard is "redefined" so to provide additional information or functionality.

The CreditCardConfirm also specifies a particular type of CreditCard account object, which has the same features of the class CreditCard, but it also provides a new feature: a new operation that performs the withdraw operation and then confirms this operation by producing in output the value of the remaining available funds in the account. So if you like the class CreditCardConfirm is an extension of the already existing CreditCard, in that its objects will also have a new operation in addition to the one already existing and inherited from the class CreditCard.

In the next few slides we will see how these two examples of inheritance can be defined in Object-Z.

Class Schema "CreditCardCount"



This is how the class schema CreditCardCount would be defined in Object-Z. The use of the class schema name CreditCard indicates that it inherits the existing class schema CreditCard.

In addition the new class includes the new attribute "withdrawals". So the initial schema has to specify the value this attribute should have when an object of this class is initialised. This explains the axiom given in the Init schema.

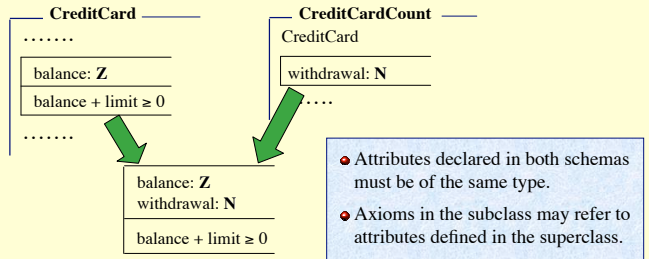
The operation withdraw, already defined in CreditCard, is further refined to include the change on this new attribute "withdrawals". This is given by just a simple increment by 1 (for each withdraw).

The main question is how does this new definition of withdraw interact with the already existing definition of withdraw given in the class schema CreditCard? And similarly, how the state schema, the initial schema of this sub-class interact with the existing state schema and initial schema of CreditCard?

Let's see in more detail this single inheritance definition.

Joining Features (state schemas)

The "full" state schema of CreditCardCount:



- Attributes declared in both schemas must be of the same type.
- Axioms in the subclass may refer to attributes defined in the superclass.
- Unintentional name clashes must be resolved by renaming attributes.

When a subclass is defined in order to refine its super-class, the newly defined features (attributes, Init and operations) of the subclass are assumed to be joined together with the existing features of the super-class, but taking into account unintentional name clashes. Let's see how this is defined.

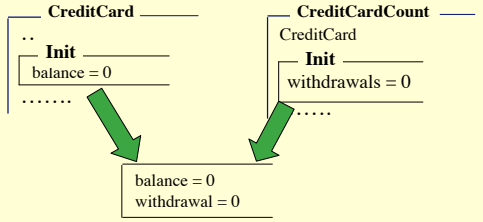
Firstly, the visibility list of the subclass is the only component of the sub-class that is not joined together with the visibility list of the super-class. These two visibility lists are kept independent, to allow different interfaces with the environment within the hierarchical structure of class schemas.

The constant definitions are joined: the full constant definition of the subclass is given by the collection of the constant definition in the super-class with the constant definition in the subclass. Note that constants with the same name must have the same identical definitions in the super-class and in the subclass.

The state schemas of the super-class and of the subclass are also assumed to be joined together. The full state schema includes as attributes the collection of state attributes in the super-class with the state attributes in the subclass, and the axiom part of the full state schema includes the collection of the respective axioms in the state schema of the super-class and that in the state schema of the subclass. Note that these axioms will have to be consistent with each other for the full state schema of the subclass to be defined.

Joining Features (Init schemas)

The "full" Initial schema of CreditCardCount:

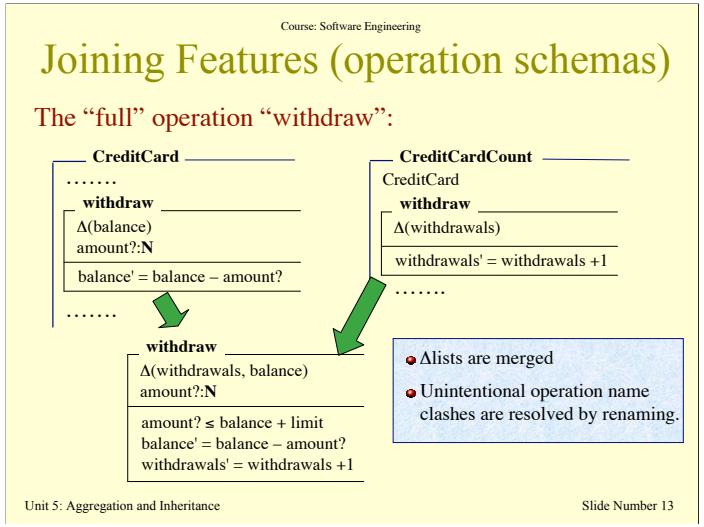


Axioms are joined together

The definition of the full initial schema in the subclass is simpler. The axioms that appear in the super-class and those that appear in the subclass are joined together. Any name clashes has at this point supposed to be already resolved by the definition of the full state schema.

Note:

In Unit 4 we said that the initial configuration of a class schema is given by the axioms of the initial schema together with the class invariants (axioms of the state schema). In this case also, the full initial configuration of the subclass is given by the axioms of the "full" state of the subclass, which we can call "full subclass invariant", together with axioms of the full initial schema.



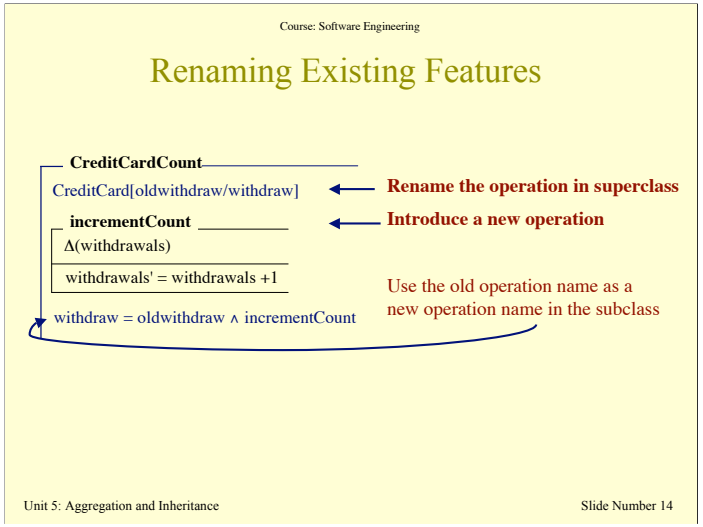
Operations defined in the subclass with the same name as operations defined in the super-class are assumed to be refinements. If this is not the case than the operation of the subclass has to be renamed to avoid to be considered as implicit joined operation with the corresponding operation in the super-class.

If the operation is genuinely a refinement, then its full definition is as follows:

The total Δ-list is defined by joining the Δ-lists of the corresponding operations in super-class and in the subclass.The declarations (or signatures) are also joined together. The axioms are joined together, but taking into account what changes and what does not change in the full state of the subclass.

This joined operation is similar to the conjunction operation (∧) that we have seen in Unit4. In this case the Δlists includes additional attributes that are not defined in the operation of the super-class.

In the next slide we give an example of renaming a feature of an existing super-class, to eliminate unintentional clashes. This is given to you just for completing the discussion, as we have mentioned renaming in these slides, but we will never use renaming.

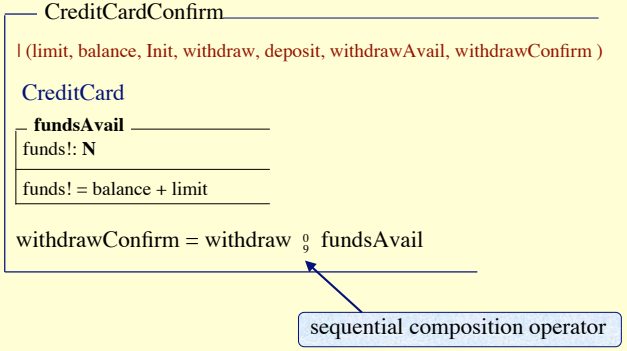


In this slide we have given an alternative way of defining the refined operation withdraw in the subclass, without requiring it to be joined together with the corresponding operation of the super-class.

The implicit joining process between operations of the super-class and operations of the subclass happens when both super-class and subclass have operations with the same name.

To stop this from happening (if needed when we are specifying a real large system), we need to use a special notation for renaming existing features in the super-class. The first line in the class schema given in this slide, says that we are inheriting the super-class CreditCard but with its operation withdraw renamed with the term “oldwithdraw”, within the context of this new subclass CreditCardCount. Specifying this renaming, within the class schema CreditCardCount, the operation name “withdraw” is now free and it can be used to define a new operation in the subclass.

Class Schema "CreditCardConfirm"



Summary

- Aggregates
  - » Collection of objects of the same class
  - » Use of selection expression in class operations
- Inheritance
  - » Refining or extending existing super-classes with subclasses
  - » Joining features between subclasses and their super-classes
  - » Sequential Composition operator.

The first thing to notice is the fact that the new operation "*fundsAvail*" in not included in the visibility list. This is because it's just an auxiliary operation used by the operation "*withdrawConfirm*".

The definition of "withdrawConfirm" uses an new operator. This is called the "sequential operator" and it specifies, in this case, first the application of the operation withdraw, as defined in the super-class, and then the operation fundsAvail on the resulting state the object. The consequence of this operation withdrawConfirm is therefore to withdraw a given amount from the account of a credit card and output the value of the available funds that remain in the account.

The sequential composition operator assumes therefore the existence of an intermediate state of the particular object: the transition prescribed by the first operation is sequentially composed with the transition prescribed by the second operation. The definition of each of such sequential transitions has to take into account the Δlist of the individual operations in the sequence. These Δlists can be different: each operation can change different sets of attributes of the same class. So, the transition of the first operation assumes that all the attributes supposed to be changed by the second operation but not by the first, are left unchanged in the resulting intermediate state (i.e. after the execution of the first operation), and similarly all the attributes that are supposed to be changed by the first operation but not by the second, are then left unchanged in the object state resulting after the execution of the second operation. Attributes common to the different Δlist are changed sequentially as the sequential operations are performed.

Note that the semantics of the sequential operator is somewhat similar to that of the parallel operation. Also for the sequential operator, outputs of the first operation and inputs of the second operation that have the same base name are considered to be equal. However, whereas the parallel operator assumes also inputs of the first and second operations with the same name to be equal and similarly for outputs, the sequential operation does not impose this constraint.