10 Object Modelling

Object Modelling

Introduction

Object modeling is used for making both Domain Object Models and for System Design Object Models.

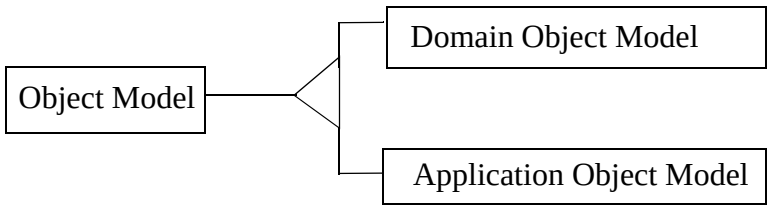

Figure 10-1: How object modelling is used in TIMe

In this theme the main elements of object modelling are covered, independent of language/notation. Object models are described in UML or SDL.

- If you want an overview of object modeling, read Elements of an object model (p.10-3).
- If you want a step-by-step introduction and guidelines to object modelling, you follow the route starting with Establishing an object model (p.10-4).
- If you want a more in-depth treatment of the approach to object modelling behind this method, read What is Object Modelling (p.10-10).

Throughout the text there will be examples (in UML and SDL) on the various parts of object models, and there will be guidelines for the use of UML and SDL for object modelling. In Guidelines for the use of UML and SDL for object modelling (p.10-18) there are more general guidelines covering whole models.

In order to learn about UML and SDL, consult the UML Tutorial and the SDL Tutorial.

Elements of an object model

An Object Model consists of a set of different *elements*:

- Object classes with attributes, relations and connections
- Attributes
- Relations
- Connections
- Generalisation/specialisation
- Aggregation
- Classes with constraints on their environments
- Behaviour associated with an object model
- Localisation (nesting)

Establishing an object model

Most object oriented methods and books on those contain useful guidelines and examples on how to identify the right (and best) objects and classes. Few of them are, however, made for the purpose of being used together with SDL as a design language and with typical SDL applications in mind. The following is therefore a set of guidelines that emphasises this. It is recommended to learn from other object oriented methods, and then use this method as a set of guidelines on what is important to take into consideration when the design language is SDL.

The activity of establishing an object model consists of going through all the elements of an object model.

The activity of establishing an Object Model in general is covered here, while the special activities involved in making Domain and System Design Object Models are covered in Making domain object model and Development application activities, respectively.

Establishing an Object Model in general consists of considering the following elements:

- Object classes with attributes, relations and connections (p.10-4)
- Relations (p.10-5)
- Attributes (p.10-5)
- Connections (p.10-5)
- Generalisation / specialisation (p.10-6)
- Aggregation (p.10-7)
- Classes with constraints on their environments (p.10-8)
- Behaviour associated with the object model (p.10-8)
- Localisation (nesting) (p.10-9)

Object classes with attributes, relations and connections

For a detailed description, look at Object classes with attributes, relations and connections (p.10-11).

The identification of object class with attributes and relations is the most fundamental element of object modelling, and it is supported by most Object Oriented Analysis tools.

The identification of objects is based on the idea that they shall model phenomena in the application area, and a classification of these phenomena.

So look for phenomena by considering the following aspects:

- *Substance*
- *Properties* of substance.
- *Transformations* on substance.

Note that normally a phenomenon will have several of the aspects, but some may be the dominant.

These aspects give certain guide-lines for selection of phenomena. We have to consider tangible things where the major aspect is substance. We have to consider properties and finally we have to consider the transformations on the substance.

Do not restrict yourself to the identification of typical “data” (property) phenomena, but also look for “action” (transformation) phenomena. SDL provides the means for taking such action phenomena and define classes and subclasses of these active objects. In the first place it is not important to distinguish between active and passive objects - be prepared that any object may become active, when actions to be done in order to fulfil required properties are associated with objects.

UML notation for classes and relations

If you know already that an class of object is a class of active objects performing concurrent with other objects, you may specify this (in UML by a comment or a stereotype, in SDL by making it a process). If you do not know this, just make it an UML object.

Connections

For a detailed description, look at Connections in detail (p.10-13).

Connections are put between classes for which the objects will communicate. Connections between classes will imply that there is a number of mscs describing the interaction pattern between objects of the classes.

Relations

For a detailed description, look at Relations in detail (p.10-13).

UML notation for relations

Relations come in two categories: constructive and illustrative relations.

We recommend to use UML relations for both kinds of relations. This implies that if relations are illustrative, then the UML notation for relations shall be used. The semantics of these may then not be defined in UML, but they have anyway to be implemented specifically.

Attributes

For a detailed description, look at Attributes in detail (p.10-13).

UML notation for attributes

When looking for attributes, consider typical “value” properties of objects.

The notation for attributes of identified classes is UML. They may either be collected in one diagram or combined with a class/relation diagram.

Guidelines

- Some attributes are obvious from a domain or design analysis, while other attributes may have to do with the required properties. If some of these properties have been specified, then go through them and make sure that the objects being involved have the necessary attributes.
- Required behaviour does not have to be associated with objects only, but may also be covered by attributes. This is done by defining the attribute type as a class and the associate the behaviour with operations on the values of the attribute. The requirement is that the operation behaviour can be defined as pure functions with no side-effects on the object.
- Attributes of objects do not have to be simple type attributes. Attribute types may be some of the classes identified in the domain analysis. After having identified attributes and their types, it should be investigated if these types also should be included in the object model.
- When to use attributes and when to use relations? If a class is only related to one other class, then consider if this property may as well be modelled by an attribute (of the same class that is related).
- What about simple object references? Shall they be handled as relations? If one of the endpoints of the reference is an active object that will not be an object in a database part of the system, then define the property as an attribute of type Reference.

Generalisation / specialisation

For a detailed description, look at Generalisation / specialisation in detail (p.10-14).

Specialisation/generalisation is a special relation between classes. The notation for this relation emphasizes that it is not just one among other relations between classes, but that it has to be supported by the design and implementation notations.

UML supports this is as a special construct. The graphics for this relation is different from the graphics for relations. The subclass relation is the obvious example on a constructive relation, that is it shall also be a relation in the SDL functional design, so UML should be used for this purpose.

When describing generalisation/specialisation there are two different ways of doing it:

- Specify the relation between the classes as a whole, without considering the details of any of the classes. This is normally referred to as describing the generalisation/specialisation hierarchy.
- Specify one class as a specialisation of another, including the details of what is inherited and how the additional properties relate to these.

Guidelines

- Use only single inheritance. If properties of multiple classes are required use either aggregation, delegation (if supported or easy to implement in the design/implementation language) or communication.

- Be aware that for active objects the UML object model with inheritance shall be carried directly over to the corresponding SDL design object model, with implied inheritance of behaviour. This means that it is not sufficient to be sure that attributes and operations of the superclass are common for all possible subclasses - behaviour must also be common.
- S-rule: subclasses as models of specialised concepts.
  Subclasses shall represent special concepts and *not* versions or *variants* of the superclass. See to that general classes only get properties that are guaranteed to belong to objects of all possible subclasses. A subclass inherits all of the attributes and operations of the superclass.For operations identify the properties they shall have independent of possible subclasses and which are important for the use of the operation.
- S-rule: virtual or non-virtual.
  Classes are made reusable in two different ways: general either in the sense that operations can be *redefined* (virtual operations) or in the sense that the class *depends* on some other entities in its context. The first is considered when defining general classes intended for specialisation: mark the operations that shall be redefinable as virtual operations, while those that are supposed to have the definition in all subclasses are not defined as virtuals. The second way of making classes general is covered in Localisation (nesting) (p.10-9).
- S-rule: generality separation
  Organise your classes in specialization hierarchies such that the general concepts need no specific information about the different specialisations.
- A-rule: library search
  See if there is a Y in a library which is similar to an X in your dictionary. When a similarity is found, either make X a direct specialisation of Y or restructure your library by making a Z which can be specialised to both X and Y.
- S-rule: controllability and flexibility
  If you are designing both the general and the special concepts, do not sacrifice the controllability of the general ones for flexibility of the specialised ones.
- S-rule: adaptable components (for SDL object modelling)
  Achieve adaptable components by introducing virtual types. Ensure that such types get proper general names. Balance the adaptability of virtuals by using ATLEAST; to limit the redefinability.
- N-rule: ATLEAST and FINALIZED
  In specialising, take care to balance flexibility and analysability properly using ATLEAST; and FINALIZED to constrain the virtual types.

Aggregation

For a detailed description, look at Aggregation in detail (p.10-15).

UML notation for aggregation

Objects may be composed of other objects by means of aggregation.

UML has a separate construct for this (real aggregation).

The recommendation is to use UML for real aggregation. Use real aggregation when the aggregate is an object itself and when the part objects shall have a life-time that is the same as the aggregate.

Guidelines

- Distinguish between “relation aggregation” and “real aggregation”. Be aware not to confuse this with where the classes of the contained objects are defined. In SDL terms use only “real aggregation” in cases where you would use blocks as part of blocks, process sets as part of blocks or services as part of processes. In case of process sets it shall be important that there are process sets connected with signal routes and preferably that the process sets are (1,1) sets. If the number of processes in the set vary and if processes are solely identified by PIds, then the process sets and the signal routes do not really corresponds to objects being part of a “real aggregate” object.
- Note that while UML supports an “open” aggregation (the outside entities can be connected directly to the entities in the aggregate), SDL supports a “closed” aggregation. SDL has in addition the combination of aggregation and nesting. Be therefore aware that you may have to transform a UML open aggregate into a closed SDL aggregate.

Classes with constraints on their environments

For the more important classes make a sketch that includes the most important classes in the expected environments of the classes and express possible constraints on these.

The specification of classes in the class environment serves three purposes:

- Understanding the environment helps to understand the class.
- The roles played by the environment classes help to validate the application of instances.
- The co-ordinates of relations that an instance may participate in can be precisely specified.

If the class in question is considered without any specification of how it is composed, then it is done in UML, the only difference from normal UML use is that only one of the classes are fully specified with attributes and relations.

If the class and its constraints on its environment involves any of the components of the class, then the notation for aggregation is used.

Behaviour associated with the object model

The objects of the systems that are the target of this method are not just data objects with some associated operation, but the objects themselves exhibit behaviour and often in interaction with other objects. Behaviour associated with the object model (p.10-16) <!def!> is therefore important to describe.

When we have identified SDL types in the object model for the functional design, then we may specify the behaviour in terms of SDL process graphs.

Behaviour associated with functional roles belong to the property models.

Identified classes of objects may, however, also have behaviour that we want to express before we go to the step of functional design in SDL. For this purpose we have the choice of pure MSC (with naming conventions in order to integrate with the object model) and sequence diagram in UML. The only alternative that gives standard MSC is the first, and this is also the alternative that gives the best integration with the SDL design.

In our descriptions we will use MSC.

Localisation (nesting)

For a detailed description, look at Localisation (p.10-17).

Guidelines

- S-guideline: Localise classes.
  If it is known that a class is only meaningful within the context (scope) of another class, express this informally or by a special relation (“origin” or “enclosing”).
- S-guideline: Localise first, then globalise.
  If you do not know where to define a class, define it locally to the object where you need the class. If it turns out that the class is not dependent upon other classes in the enclosing class, then it may be defined globally.
- Try to identify collections of related classes.
- S-rule: context parameters (SDL for object modelling).
  Achieve independence of signals and data types by introducing context parameters. Balance this independence by constraining the context parameters.

What is Object Modelling

There are many approaches to object orientation. The approach followed in this method is that an object model is regarded as a *physical model*, simulating the behaviour of either a real or imaginary part of the world. In the same way as other perspectives on modeling is based upon notions that are well-defined outside computer science (e.g functions, logic), the physical modeling approach is based upon notions that are not specific for computer science. Physical models have been used in many other areas.

The physical *modeling* approach is followed by most existing object oriented methods, while object oriented design and implementation languages are often based upon a more *programming* approach to object orientation, with emphasis on objects being collections of data and operations, on encapsulation and on code reuse. While the modeling approaches advocate that subclasses model special concepts and therefore shall be extensions of the superclass, the programming approach is more flexible in the sense that it allows subclasses to override properties from the superclass.

The modeling approach requires that we have languages for analysis and design/implementation that matches, otherwise we would be forced to transform an analysis object model according to the modelling approach to a design object model in a languages that follows the programming approach. The method relies on UML for analysis and SDL for design: they both follow the modelling approach, and it is therefore straight forward to transform from UML to SDL. The following will contain a description of the object modeling approach together with an indication of the corresponding concepts in UML and SDL and of the transition from UML to SDL.

The main property of physical modelling is that it is based upon a conception and understanding of the application domain in terms of *phenomena* and *concepts*, and that physical models will have elements which directly reflect these phenomena and concepts. The physical model will consist of objects, that represent the phenomena, and of classes that represent concepts. Note that even though there are no formal definition of phenomena and concepts, then the representations of these by means of objects and classes are based on a formal language.

Associated with phenomena and concepts are a number of structuring and abstraction mechanisms:

- Identification of phenomena and the *classification* of these into concepts.
- Part/whole *aggregation*, that is phenomena as part of other phenomena.
- *Relation* composition, that is a phenomena has relations to other phenomena instead of having them as parts.
- *Specialisation* of concepts. Classification relates all phenomena with the same set of properties into a concept. Specialisation is a mechanism for the structuring of sets of concepts with similar properties into general and specialized concepts.
- *Localisation* of definitions: Some phenomena and concepts are only meaningful within the context of a specific phenomenon or concept.

Object oriented modeling according to this approach consists of applying these structuring and abstraction mechanisms.

An object model is based upon the classical notion of phenomena and concepts.

The classical notion of a concept is characterised by the following:

- extension, the collection of phenomena that the concept covers;
- intention, a collection of properties that in some way characterise the phenomena in the extension of the concept;
- designation, the collection of names by which the concept is known.

Representing concepts by classes and phenomena by instances of these classes follows this pattern: the instances belong to the extension, the class definition gives the intention and the class name represents the designation.

Consequences of this approach are:

- Classes are not instances, not even sets of instances, but “only” the definition of common properties of instances generated according to this class.
- Generalisation/specialisation (inheritance) should be used to represent concept hierarchies and it is represented differently from relations between sets of instances.
- Instances as models of phenomena shall be able to model the fact that phenomena often consist of part-phenomena. Therefore we recommend a special notation for aggregation, also different from normal relations.
- Not all classes are candidates for reuse. Some classes may only be meaningful within the context of other classes and should therefore not be made more general than that.

Object classes with attributes, relations and connections

The identification of objects is based on the idea that they shall model phenomena in the application area. This is of course not enough to hint on what kind of phenomena to look for and how they shall be modeled by objects. The following aspects of phenomena should be taken into consideration:

- *Substance* is the physical material that makes phenomena exist over time. Substance is characterized by unique identity, a certain volume and a unique location in time and space.

UML

In UML objects are the elements that cover this aspect. Objects have a special notation that includes the name of the object, and the class of the object defines what kind of attributes form the substance of the object.

SDL

The elements in SDL that represent this aspect are instances: that is blocks, processes, services, procedure invocations, variables, signal instances. Part/whole and relation composition stem from the modeling of this aspect.

- *Properties* of substance. All properties associated with substance have to be obtained by measurements. A given property of some substance may be observed by performing a measurement.

UML

Attributes of UML are the means for describing properties of objects. Operations

with a result type may be used to model the fact that properties have to obtained by a measurement.

SDL *Variables are the SDL means to represent properties. Values are used to denote the result of measuring the properties. Value returning procedures are new mechanisms in SDL92 to describe measurements.*

- *Transformations* on substance. An information system is characterized by transformations which change its substance and thereby its measurable properties. Transformations are partially ordered sequences of events. In object models, transformation is the result of *behaviour of objects*, either involving properties of itself or other objects.

UML *UML covers this by operations on objects. UML has a separate dynamic (behaviour) model for describing the action sequences of objects, while we associate this directly with the objects. As SDL has support for this, we use SDL behaviour descriptions for this purpose.*

SDL *In SDL the transitions of processes/services and procedures are the means for describing transformations. Actions of transitions may involve other instances, by signal exchanges and by remote procedure calls.*

Note that normally a phenomenon will have several of the aspects, but some may be the dominant. Phenomena where transformation of properties are the most interesting, (action phenomena), have a rich set of possible representations in SDL92: processes for concurrent sequences of actions, services for alternating sequences and procedures for sequences that are parts of larger sequences.

All phenomena in the real world are different. Abstraction arises from a recognition of similarities between phenomena (and concepts) and the decision to concentrate on these similarities, and to ignore the differences. An abstraction covers a group of phenomena characterized by certain properties.

Classification is a process that either applies to phenomena (objects) and thereby generates concepts (classes) or is applied to concepts and thereby generates general or special concepts.

UML *Identified concepts are represented in UML by classes. The definition of the class represents the intension of the concept, while the objects according to this class represent the extension of the concept.*

SDL *Identified concepts are represented by types in SDL. The definition of the type represents the intension of the concept, while the instances according to this type represent the extension of the concept.*

Classification may also be done on identified objects of an object model. It may be so that the similarities become apparent when the objects are specified to some detail.

Objects may of different *kinds*, depending on whether they have their own action sequence or not, and whether they perform concurrent with other objects or not.

Passive object do not have their own action sequence. They may have operations, which may be executed by other objects.

Active objects may either perform *concurrently* with other concurrent objects, or they may perform *alternatingly* with other alternating objects as parts of concurrent object. Alternation means that one object is executing at a time and that common attributes may be accessed with synchronisation. Concurrent objects may either communicate by sending messages or by synchronised execution of operations.

UML *UML supports the distinction between active and passive objects, but not the distinction between concurrent and alternating objects. UML only considers the behaviour of single objects and not communication. UML is therefore not used to model this aspect.*

SDL *SDL have processes that execute concurrent with other objects, and service as part of processes execute alternating, depending on the incoming signal.*

See Behaviour associated with the object model (p.10-16) for the behaviour of single objects.

Relations in detail

A relation represents *application specific* relationships between objects of the involved classes. Instances of a relation are called links and consist of tuples of object references. Structural “relations” as subclass-of and part-of are *not* regarded as relations, but as separate constructs. As an example, the fact that a class is a subclass of another class does not imply that there are any links between objects of the two classes.

UML *UML supports relations directly by means of associations.*

SDL *SDL does not support relations, only object references by means of PId variables.*

Connections in detail

Objects are connected if they are involved in communication with each other. This is different from objects being related, as this will only imply that the objects may be reached by navigating along the relations.

For the use together with SDL as the design language, connected objects will mainly be objects that in SDL will be represented by blocks or processes.

Attributes in detail

Attributes of objects are supposed to model the “value” properties of the corresponding phenomena. These properties change value (state) over time and they are constituent parts of the phenomena. Attributes will therefore be objects of some attribute type, that defines the possible values of the attribute and these are obtained and changed. In contrast, relations are used to model properties that are not constituent parts of the phenomena but rely on the existence of some other phenomena.

Some attributes model the fact that an object can refer to other objects. This is not the same as relations, as the object reference is only valid as an attribute of some object.

UML *Attributes are directly supported by UML attributes. Attributes can be of predefined attribute types, and not of classes in the object model. UML introduces the possibility that attributes can be of classes. Object references are supported by a special use of associations.*

SDL *SDL variables can be of user-defined types. This comes directly from the UML object model. Object references are supported for processes (variables of type PId).*

Generalisation / specialisation in detail

For classification of concepts we have the notions of generalisation and specialisation. Generalisation is a means to focus on similarities between a number of concepts and to ignore their differences. To generalize is to form a concept that covers a number of more special concepts based on similarities of the special concepts. The intension of the general concept is a collection of properties that are all part of the extension of the more special concepts. The extension of the general concept contains the union of the extensions of the more special concepts. The inverse mechanism is to specialise: to form a more special concept from a general one.

In a corresponding object model, a class representing a specialised concept *inherits* the properties of the class representing the general concept. It is also said to be a *subclass* of the general (*super*)class.

From this follows that the generalisation/specialisation relation between classes is not the same as ordinary relations. While two classes associated with a relation implies that instances of these two classes are related, a subclass of another class implies that an instance of the subclasses has the properties of the superclass and the properties specified in the subclass definition.

The approach followed here advocates that subclasses only *extend* the properties of the superclass. It also advocates the distinction between virtual and non-virtual properties of classes: virtual properties can be redefined in subclasses, while non-virtual cannot. The benefit of this is that the specifier of a general class can assure that some of the properties will be the same for all subclasses.

Inheritance in this approach implies inheritance of both attributes, operations *and* behaviour in terms of FSMs, that is:

- Inheritance of all attributes and operations, possibly redefinition of virtual operations.
- Inheritance of states and transitions
- Inheritance and possibly redefinition of virtual operations as part of transitions
- Inheritance and possibly redefinition of virtual transitions.

UML *In UML the specialisation is represented by inheritance between classes. If operations are introduced in the UML object model, be aware that there is no mechanism for distinguishing between virtual and non-virtual operations.*

SDL *In SDL, specialisation are described by inheritance between types. The specialised types will be described only by additional properties to those of the supertype and by redefinitions of virtual types and transitions. Generalisation is supported by defining types that have virtual properties.*

Aggregation in detail

The part/whole composition is used to model that some phenomena are integral parts of other phenomena. This is not the same as functional decomposition, but functional decomposition is a special case, where the emphasis is on phenomena providing functionality.

In most object-oriented languages there is little direct support for composition. It is usually supported indirectly through instance variables as in e.g.Smalltalk. Instance variables (and thereby composition) are often considered implementation details and are not part of the public interface of an object. Alternatively, composition is often simulated using multiple inheritance. A consequence of this is that most object- oriented languages have good support for classification, but poor support for composition.

In addition to the obvious purpose of modeling that wholes consist of parts, part objects may also be used to model that the containing object are characterized by various *aspects*, where these aspects are defined by other classes.

Composing/decomposing by means of aggregation is the counterpart to relation composition. The relationship to another object is a dynamic property - the related object may at different points in time be different objects. A part object will be the same object throughout the life-time of the containing object.

Representing parts by relations/references to separate objects is of course possible in languages that do not support part objects, but it will not ensure that the relations/references are not later changed to denote other objects.

Part objects may be compared with variables of types. The variable is the same object throughout the life-time of the containing object, but its value may change.

Normally variables are of general or predefined types that are globally defined. The approach followed here supports, however, that part objects may be objects of any class, also of locally defined classes. As locally defined classes can be subclasses of globally defined classes, it is possible to make part objects that both have general properties and properties that are specific for the containing class.

In most cases part object are introduced because they model some important property of the containing object - they are not just introduced as an “implementation” of the containing object. In these cases it is important to be able to *refer to part objects*. The approach therefore allows references to part objects, and it also supports the notion of *open aggregates*.

An open aggregate is ...

Why open aggregates? It seems to be in contradiction to what is regarded as one of the main properties of object orientation (encapsulation). These are the reasons:

- Aggregation is not only used for “implementation purposes”, that is parts of an aggregate are not only introduced in order to implement the functionality/interface of the container object. Often parts of an aggregate have a modeling purpose.
- Open aggregation is used to analyse situations that will end up in SDL designs where the container objects are blocks and where the contained objects are processes, and where the primary identification of the receivers of messages are identification of the processes. The processes are here the main objects, and the blocks are just used for a (static) structuring of many processes.
  The corresponding situation would arise in C++ if C++ had a similar grouping concept for objects and where the objects are still the main entities.
  Aggregates may also turn up during design and implementation, introduced for implementation purposes. In these cases open aggregation should not be used. In SDL this situation will develop either block types where the contained processes will send signals via gates and not directly to processes denoted by PIds, or process types where the contained objects are services: they cannot not be addressed directly.
- Open aggregation may be used just for the purpose of distributing functionality between parts (during analysis), while the design and implementation may choose to close the aggregate. Communication links directly to parts are then just used as an illustration. *In SDL this implies that communication links to processes as part of a block will either be turned into a corresponding number of gates, or all communication between blocks will be merged to one gate. This requires that the incoming signals on a gate may be uniquely distributed to the contained processes and that these either are contained in (1,1) process sets or it is not so important which process of a process set that gets the signal.*

SDL

Instead of multiple inheritance in order to give a class of object the properties of a set of classes, it is possible to define the class to have part objects of the classes. The properties are then indirectly available.

UML *In UML this is directly supported by having part objects according to any class.*

SDL *In SDL this is only supported in restricted forms: properties that may be defined as properties of a set data types or as properties of a set service types can be made available for process types (by defining variables of these data types or by defining the process type by means of services), while properties of a set of process types cannot be made available for a process type.*

Behaviour associated with the object model

Behaviour is associated directly with objects in the Object Model. Each object can have one sequence of action. Partial sequences are represented by operations that are performed by objects. Some objects may perform concurrently with other objects, while other objects may be performed as part of the sequence of a concurrent object. This means that if an object is supposed to model a phenomenon that have several sequences, then this object is composed of objects, one for each sequence.

Behaviour is not handled by a separate model, but is handled as part of the object model. The specification of inheritance in the object Model is carried over to the Behaviour Model and has direct implications for the inheritance of behaviour.

Localisation

Some phenomena and concepts are only meaningful within the *context* of a specific phenomenon or concept. Localisation of definitions supports this and gives rise to nesting of definitions. Scope rules and binding rules determines how nested definitions may use entities defined in enclosing definitions.

Most approaches to object orientation and most object oriented languages support localisation of methods: a method is located in the class defining the objects that may perform the method, and from within the method definition the object attributes are visible.

UML

UML does not support nesting of classes, so if this is required, either specify this in a comment or by some naming conventions. Alternatively, make a class symbols within the enclosing class symbol.

SDL

SDL definitions may be nested and thereby support localisation. Type definitions may, however, be located where it is most convenient, as long as they are visible from they are supposed to be used. If identified types in an early stage should be specified as part of the system specification (and not yet as part of a package), they may simply be defined at the system level, without considering where they in fact belong. General types of e.g processes and procedures can be defined at system level, in order to be used in several blocks of the system. More special types should be defined where they are used.

A package of types is the ultimate example on non-localised type definitions, while exported procedures will most often be defined locally to the process (type) that exports it. Signals are often defined in the nearest enclosing block in which they are used between processes. As demonstrated in the example with composition of services, context parameters provide the mechanism to make a type definition independent on the enclosing scope.

Guidelines for the use of UML and SDL for object modelling

Here we present:

Constructive versus illustrative parts of object models

The use of UML relies on the following distinction between *illustrative* and *constructive* parts of descriptions:

- An *illustrative* part of a domain object (or property) model description is a part that is not automatically transformed into a corresponding design.
  Examples are relations between what becomes SDL processes in the design that have to be “implemented” because SDL does not provide relations.
- A *constructive* part of a domain object (or property) model description is a part that may be automatically transformed into a corresponding design.
  Examples are parts of object models with relations that may be transformed to data-base schemes; a subtype relation between two types in the domain object model that is transformed to the corresponding relation between the corresponding SDL process types.

For illustrative parts of a description of e.g. an object model we recommend to use the extensions to UML.

For constructive parts of a description of e.g. an object model we recommend to use some existing object model notation, here exemplified by UML.

- UML for *constructive* parts of object models, including the specification of classes, attributes and relations that are supposed to survive as relations in the functional design. We recommend that the relation aggregation should not be used.
- UML for the *illustrative* parts of the conceptual model specification and for the aggregation relation.
- MSC for the specification of behaviour properties associated with both classes and roles.
- The UML notation for object modelling is described in the Tutorial on UML, while an MSC Tutorial provides an introduction to MSC used for Property modelling.

When to use UML and when to use SDL?

As a general guideline, UML is used for domain object modelling and SDL for design object modelling. The reasons for using UML for domain object modelling are:

- It is richer than SDL on relations that may be important for the understanding and communication about the domain.
- It is easier to make incomplete descriptions, e.g just specifying the names of attributes and not the types.
- It is not necessary to take decisions on whether an object is a block, process, service or type value.

If these reasons are not important, then SDL may be used as well. SDL may also be used for the domain object modelling in cases where

- state oriented properties of domain objects are part of the domain model,
- communication links should be described with great detail (signals and signal lists.

SDL will also be used when inheriting a domain model in terms of packages of SDL types.

SDL is normally used for design object models, but there is one case where it may be more appropriate with UML:

- the “system” really consists of a database component (in UML), a user interface component (in some other language) and a “controlling” component (in SDL).

How to use UML and SDL in combination

This is the topic of the coming Z.109 standard SDL with UML, and a discussion on this will be included in the next edition of TIMe.

How to map UML models into SDL models

UML may have been used for illustrative purposes, and for those parts the mapping may require some work.

- Attributes of user defined classes
  The classes defining types of attributes are mapped onto data types. Operations are mapped to operators.
- Operations on objects
  These are mapped to remote procedures.
- Real aggregations of active objects.
  Two different cases apply:

- If the container object is just a container object, that is it has no attributes, operations or behaviour, then the mapping is to an SDL block with processes.
- If the container object has attributes, operations or behaviour, then the mapping is to a block with an extra process that gets the properties of the container object.

- Real aggregations of passive objects.

- If the container object is just a container object, that is it has no attributes or operations, then the mapping is to a struct data type.

- If the container object has attributes or operations, then the mapping is struct object with all the attributes and operations.

- How to map relations
  SDL does not support as relations, so either this part of the UML model should be covered by a database part of the application, or the relations shall be mapped onto PIds, but this only works for processes.

- Specialisation
  If only single inheritance is used, then the mapping is straight forward. Behaviour is probably specified directly in SDL, so there is no need for a mapping. If UML is used for describing the behaviour of objects, then the following mappings apply:

- tbd

- Localisation
  Mapping is straight forward.

List of figures

List of definitions

Aggregation

All non-trivial systems are composed from components. The process of putting components together to form a whole is called aggregation. Aggregation enables us to associate a single concept and a name with a composite object. This helps to simplify matters considerably when we are dealing with the object as a whole. But to build the object and use it correctly we need to understand what it consists of.

An aggregate is an object in itself and the part objects are parts of this object only. This is in contrast to aggregation just by using ordinary relations.

The opposite process of decomposing a whole into parts is called partitioning (or decomposition).

We distinguish between relation aggregation and real aggregation.

Attributes

Attributes of objects are “value” properties that are not covered by part objects (aggregation). Attributes are defined by a name and a type. In Domain Object Models this is informally specified, but it is still worthwhile to use a type that will be defined as an attribute type or class in the Design Object Model.

For the specification of attributes in UML, see attribute specification in UML.

For the specification of attributes in SDL, see variable definition in SDL.

Behaviour associated with an object model

If a class of objects has been identified as part of the object modelling, then it is possible to associate behaviour with objects of this class. If some behaviour has been identified without being associated with any object or class (but a role), then it is possible to associate it with classes later or combine it with other behaviour specifications to new roles or classes.

Depending upon the nature of the behaviour that is desirable to express, it is either expressed in terms of MSC or in fragments of SDL process graphs. The latter may be applicable if the analysis is based upon existing specifications in SDL or in case it is desirable to specify behaviour properties like “instance of type AccessPoint” shall always (that is in any state) accept a Log signal and respond to the Logger with the current status of the point”.

Class with constraints on its environment

Classes are often defined with a specific purpose in mind, and especially for the behaviour of a class (typically becoming a process type in SDL) it is necessary to know what other processes will be in the environment. This is typical for the scenario with several equally “important” objects that have to co-operate in order to do a task. It will, however, reduce the reusability of the class in other contexts where these other objects will not be. A quite different scenario is the specification of a typical “server” object class that should work in any context and where the behaviour is independent on the behaviour of the client objects.

A specification of a class with constraints on it environment contains the following elements:

- The class definition in focus may contain a definition of the attributes of the class (the intention).
- The environment of a class is important for the understanding of its purpose and constraints. Therefore, the environment of importance has been depicted outside the class. Entities in the environment represent roles.
- When the class is instantiated there will be entities in the actual instance environment that will play the roles. Therefore, all instances must comply with the roles given to them by the other instances.

A class definition may include a prescription of what we consider a valid instance environment. The entities and relations in the environment of a class represent roles that shall be played by actors in the environment of an instance of the class.

Connections

Objects are connected if they are involved in communication with each other. This is different from objects being related, as this will only imply that the objects may be reached by navigating along the relations.

When using SDL as the design language, connected objects will mainly be objects that will be represented by blocks or processes in SDL.

Generalisation/specialisation

For classification of concepts we have the notions of generalisation and specialisation. Generalisation is a means to focus on similarities between a number of concepts and to ignore their differences. To generalize is to form a concept that covers a number of more special concepts based on similarities of the special concepts.

The intension of the general concept is a collection of properties that are all part of the extension of the more special concepts. The extension of the general concept contains the union of the extensions of the more special concepts. The inverse mechanism is to specialise: to form a more special concept from a general one.

Note that the exact meaning of specialisation will only be given when it is applied in a formal language. When using specialisation in the domain object modelling it is recommended to use it in a way that will not be very different from the meaning in the design.

Localisation (nesting)

Some phenomena and concepts are only meaningful within the *context* of a specific phenomenon or concept. Localisation of definitions supports this and gives rise to nesting of definitions. Scope rules and binding rules determine how nested definitions may use entities defined in enclosing definitions.

Object classes with attributes, relations and connections

This aspect of object modelling has to do with identification of classes without considering how many instances there will be in a given system and also without considering how they are used in the design of specific systems or other instances.

Real aggregation

Real aggregation is supported by UML.

Real aggregation implies:

- that the part object is only part of one object, and
- that possible relations specified with the part object (class) as endpoint only hold for the part object and not for all objects of this class.

UML adorns the association with a filled diamond and calls it *composition*.

Relation aggregation

This is the form of aggregation where the part objects are just related to the composite object with a special relation, but still just a relation. This was the only form of aggregation supported by OMT.

UML adorns the association with a hollow diamond and calls it *aggregation*.

Relations

A relation represents *application specific* relationships between objects of the involved classes. Instances of a relation are called links and consist of tuples of object references. Structural “relations” such as subclass-of and part-of are *not* regarded as relations, but as separate constructs.

Relations can be used either as the basis for automatic generation of the corresponding part of functional design (e.g. a database part of the design) - that is as *constructive* parts of the conceptual model, or as *illustrations* of properties that will be “implemented” in some way in the design.