Framework for Feature-Based Architectural Information Modelling

Jos P. van Leeuwen

Design Systems Report 1999 / 2

Design Systems Reports

This is a series of reports of the
group Design Systems (Ontwerp Systemen),
Faculty of Architecture, Building, and Planning,
Eindhoven University of Technology.
ISSN: 1388-9591

editors:
dr.ir. Henri H. Achten
ing. Jan Dijkstra
dr.ir. Jos P. van Leeuwen

Copies can be ordered from:
Secretary of the group Design Systems,
Eindhoven University of Technology,
Faculty of Architecture, Building, and Planning,
Main Building 4.05
P.O. Box 513
5600 MB Eindhoven
The Netherlands

Other issues in the series of Design Systems Reports are:

1999/1 **Conjoint Analysis and Virtual Reality**
Exploring the possibilities
Jan Dijkstra, Harry J.P. Timmermans, Walther A.H. Roelen

1998/2 **Workshop Case Based Reasoning and Design**
Featuring: Prof. Mary Lou Maher
Reported by: H.H. Achten, S.M. Mallory-Hill, B. de Vries

1998/1 **Virtual Reality Design Information System / Distributed Interactive Simulation**
Henri Achten, Joran Jessurun, Jos van Leeuwen, Bauke de Vries

CIP. GEGEVENS KONKLIJKE BIBLIOTHEEK DEN HAAG

Framework for Feature-Based
Architectural Information Modelling

Jos P. van Leeuwen
Eindhoven: Technische Universiteit Eindhoven,
Faculteit Bouwkunde, groep Ontwerp Systemen.
ISSN: 1388-9591

ISBN: 90-6814-102-3

Keywords: architectural design and decision support systems,
product modelling, feature modelling, design information systems,
computer aided architectural design.

Contents

1. A framework for flexibility and extensibility

Introduction to the theory of Feature-based Modelling that was developed by [van Leeuwen 1999] to facilitate the requirements of flexibility and extensibility imposed on design information modelling by the dynamic nature of architectural design.

1

The theory presented in the PhD thesis 'Modelling Architectural Design Information by Features' [van Leeuwen 1999], addresses the requirements that must be met by the way design information models are defined and structured, and what type of information has to be dealt with in these models. This DS Report extracts chapter 7 from this thesis, which develops a framework for information modelling that fulfils these requirements. It forms the basis for the design and development of design support systems that incorporate the Feature-based modelling approach. Hence, the framework is called the Feature-Based Modelling framework or FBM framework. The main issue in the development of such a framework is that the information models that result from working with this framework should meet the requirements of flexibility and extensibility that have been formulated in [van Leeuwen 1999]. The development of the framework is based on object orientation, which is regarded the most appropriate technology to define and structure information within the context of this research project. A *Feature* therefore is an object in the framework, its definition and structure are provided in the object's type definition which is called the *Feature Type*[1]. Accordingly, a design information model is built up from Features of which the definitions are formulated in conceptual models of Feature Types, also called Feature Type Libraries. The actual design information models, to distinguish them from conceptual models with Feature Types, are called instance models or Feature models.

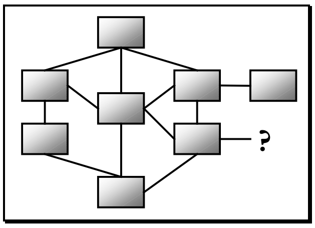


Figure 1 Extensibility: adding new object types to a conceptual model.

Departing from the above, extensibility means that in the course of design, as more information becomes known and gets defined by a designer, Feature Types can be defined to meet the particular needs of the design context (see figure 1). These new Feature Types can then be added to the conceptual model of Feature Types. From these new Feature Types, instances can be created in the particular design information model, being the actual Features.

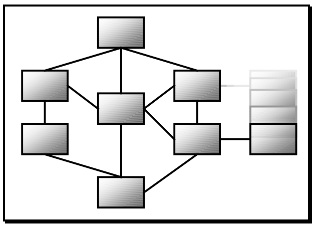


Figure 2 Flexibility: restructuring object types or objects in a model.

Flexibility (see figure 2) is required both at the level of the conceptual model and at the level of the instance model. For the extensibility to work, the structure of information defined in Feature Types in the conceptual model needs to be adjusted for new Feature Types to fit in. This is the flexibility at the conceptual level. It involves the addition and modification of relationships between Feature Types.

At the level of the instance model, which contains the actual Features, flexibility means that the definition of Features, and in particular their inter-relationships, allows restructuring the network of Features in accordance with the changing state of the design. This implies that relationships between Features are not rigidly defined and can be removed, added or modified at will by a designer. Which, in turn, implies that this kind of relationship is not defined at the conceptual level, but at the level of instantiated Features instead. This notion of relationships defined at the level of instantiated information models will play an important role in the development of the framework.

The following sections describe the information-modelling framework in terms of the processes that take place at the various levels within the framework, and the way information is structured within the framework.

[1] In fact, this is only one of two points of view from which instantiation takes place in the framework, see section 5 on this twofold instantiation.

2. Processes in Feature modelling

Four processes facilitate the tasks of Feature modelling: Feature Type definition, Feature Type classification, Feature Instantiation, and feature modification.

2

Essentially, four processes take place in the framework for Feature-based modelling, these are displayed in the IDEF-0 schema of figure 4. Although they are presented and discussed here in a successive order, they do not necessarily form a linear sequence, as is shown by the back looping data-flows. In Feature modelling activities, the described processes will take place in arbitrary order, as is required for the design-case at hand. The processes much resemble modelling activities in product modelling and other object oriented approaches. Yet, apart from the resulting information structure which in the Feature-based approach differs largely from these approaches, the major difference in terms of processes is that the definition of the object type in the Feature-based approach also lies in the hands of the designer, and is not privileged to the developer of information models. This section introduces the four processes as shown in figure 4.

The first process to be discussed is called ‘Feature Type definition’ in which domain knowledge is transformed into a formal description: a Feature Type. A particular set of knowledge from the domain of design is identified as a relevant concept that will serve as a basic entity in modelling and reasoning about designs and that will be represented formally by a Feature Type. The Feature Type describes the kind of knowledge that is represented by this formalisation and how it is structured. It is later used to create instances of this concept in actual design models, where the formalised knowledge is applied in a particular design case. How domain knowledge can be formalised and what the resulting structures of information will look like, is discussed in more detail in sections 3 to 6. Various approaches that can be followed in defining Feature Types are discussed in [van Leeuwen 1999].

In the second process, Feature Types are classified into libraries of Feature Types. These libraries in fact form the conceptual models that represent domains of design-knowledge and are the basis for modelling information in actual design cases. Classification mainly serves two purposes. The first is to help a designer organise the formalised body of domain knowledge into categories of Feature Types. The second is to allow standardised domain knowledge to be unambiguously accessible by those who conform to the standard and aim to use the standardised domain knowledge for modelling purposes and exchange of design information.

The third of the processes to be discussed is the one where Features are instantiated from Feature Types. Feature Instantiation involves the selection from a Feature Type Library of an appropriate Feature Type in the context of the current design case, and creating a Feature Instance based on the content-type and structure of information as defined in the Feature Type. The Feature Instance is then provided with the actual information as applicable for the particular design case, and is related to the structure of Feature Instances that are already present in the model representing the design case. Clearly, if no appropriate Feature Type is available for the specific circumstances in the current design, a new Feature Type must be defined prior to the instantiation procedure. How information is structured and how Features are inter-related is discussed in detail in sections 3 to 6, while section 7 discusses more elaborately the issues in instantiation of Feature Types.

The fourth process taking place in Feature-based modelling and the final one to be discussed here, is modification of Feature models. Modification of the information represented in Feature Instances may take place in various forms, depending on the way the information is structured in the Feature Instance and in the Feature Type, and depending on the effect of the modification as desired by the designer. The information present in the Feature Instance can simply be modified by changing, for instance, its numerical value. Another way of modifying the model is to change the inter-relationships between Feature Instances, which may result in different values for information in the model, or even in differently structured information. Before these various ways of modifying a Feature model are discussed further, the Feature-based structure of information is examined in the next three sections.

3. The Feature-based information structure

This section describes three layered infrastructure for defining information concerning Feature Types and Instances and three levels of abstraction.

3

The framework for Feature-based information modelling defines a three-layered information infrastructure (see figure 3). The three layers of abstraction in this model accommodate the required functionality of Feature models and conceptual Feature models (libraries of Feature Types) to enable extensibility and flexibility of these models. The dashed box in figure 3 indicates the middle layer containing Feature Type definitions. These are either *Generic Feature Types* or *Specific Feature Types*. The difference between the two and their relationship is explained in section 3.1; for now it suffices to describe Generic Feature Types as standardised and Specific Feature Types as customised, or designer-defined, Feature Types. The middle layer contains Feature Types collected in Feature Type Libraries. The Feature Types define the type of information that a Feature may contain and the structure of this information. For example, a Feature Type called 'Material' would have such properties as 'colour', 'texture', 'durability'. Each of these properties are defined as types of Features themselves and referred to by the 'Material' Feature Type. This ensures that property-definitions can be shared by various Feature Types and that duplicate definitions of such properties are not necessary.

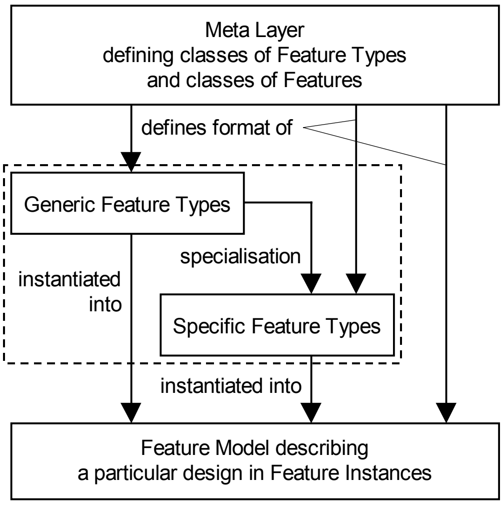


***Figure 3** The three-layered infrastructure of Feature-based modelling with three levels of abstraction.*

The bottom layer of the three-layered model contains the actual information describing a particular design. This information is represented in collections of Feature Instances, also called Features in short. A collection of Feature Instances forms a Feature model. Feature Instances are instantiations of Feature Types, or

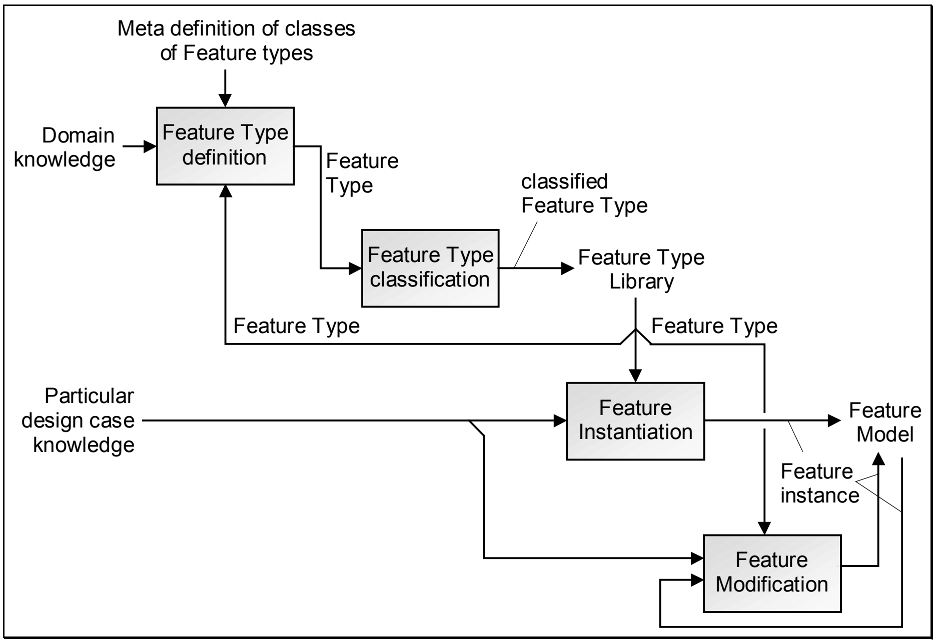


***Figure 4** Activities of Feature modelling. Formalisation of domain knowledge into Feature Types, which are classified into libraries of Feature Types. Instantiation of Feature Types into Feature Instances comprising Feature models (IDEF-0 schema) [van Leeuwen and Wagter 1998].*

conversely, Feature Types define the structure and type of information contained in Feature Instances. An instance of the Feature Type ‘Material’ would be the Feature named ‘Concrete’, that has the value ‘grey’ for the property ‘colour’, the value ‘rough’ for the property ‘texture’, and the value ‘high’ for the property ‘durability’. Each of these values of the properties are instances of the respective Feature Types and are related by reference to the Feature ‘Concrete’. This structure is similar to the structure of property definitions in Feature Types, and is to ensure that duplicate instances of such properties are not necessary where usage of the same property is actually intended. The colour of another material that should match as closely as possible the colour of this concrete would simply refer to the same colour property. Using references for properties that are modelled as Features guarantees that such sharing of properties is always possible.

The top layer in the model in figure 3 is the so-called Meta Layer. This layer defines the format in which Feature Types and Feature Instances are to be defined. The format specifies the different kinds of Feature Types and Feature Instances that can be defined in the framework: these are classes of Feature Types and classes of Feature Instances. The remainder of this thesis will mainly focus on the definition and structure of these classes and their relationships to each other and design support systems. Classes of Feature Types are defined in section 6, classes of Feature Instances in section 7.

3.1 Generic versus Specific Feature Types

As indicated before, the difference between Generic Feature Types and Specific Feature Types is that Generic Feature Types are standardised, whereas Specific Feature Types are defined, for instance, by a designer for specific purposes. For Feature-based modelling systems, the difference between the Generic and Specific Feature Types has only limited relevance. Both categories can be extended with newly defined Feature Types, albeit that new Specific Feature Types will probably appear much more frequently than new Generic Feature Types. Feature Types can be defined ‘from scratch’, i.e. without an underlying Feature Type from which information is inherited, or as a specialisation of another Feature Type. Specialisation can occur within a category of Feature Types, but a Specific Feature Type may also be a specialisation of a Generic Feature Type.

4. A typology of relationships

The various relationships that can exist between entities in an architectural design information model are categorised into a limited set: specialisations, decomposition, associations, and specialisations.

4

A survey of relationships between information entities in architectural product models has resulted in the distinction of the following kinds of relationships.

specialisation relationship	category 1
decomposition relationship	category 2
structural dependencies	category 3 or 4
adjacency	
tolerance	
dimensional relationship	
positional relationship	
existential dependency	
algorithmic relationship	

These relationships are categorised into four types, three of which are known from object oriented approaches.

1. Specialisation
   Specialisation indicates that a given type of Feature is a sub-type of another type. The sub-type inherits all characteristics of the super-type and distinguishes itself from the super-type by adding characteristics: the result is a specialised type. Often this sort of relationship is denoted as an `is_a` relationship.
2. Decomposition
   Decomposition indicates another kind of hierarchy, in which information entities[2] are divided into parts, or components. There is no inheritance, the components just contain a part of the total of characteristics. Decompositions are often called has_a relationships.

[2] The term ‘entity’ is used to refer to both Feature Types and Feature Instances. The reason for this is that the discussed relationships are applicable to both the conceptual and instantiated level of information modelling.

The inverse mechanism of specialisation is called generalisation, while the inverse of decomposition is called aggregation.

3. Association
   Associations are not hierarchical relationships, there is no inheritance, nor division of characteristics. This relationship indicates any association of two entities of information that does not fall in either categories of specialisation or decomposition. However, a particular type of association is distinguished separately and called specification.
4. Specification
   The above distinguished types of relationships appear in OO approaches to information analysis. A fourth type is distinguished in the FBM framework: a specification relationship is a kind of association indicating that one entity specifies information about another. A specification is used to model information that directly determines the characteristics of a concept. It is not to be confused with decomposition, since the specified information does not need to be a part of the concept. An example of a specification is a type of Feature called Door for which the manufacturer-details are specified by a type called Manufacturer. Note that the specification relationship is directed from the Door to the Manufacturer as 'specified by' and conversely from the Manufacturer to the Door as 'specifies'.

Martin and Odell [1995] distinguish compositional relationships from non-compositional relationships. Compositional relationships are further distinguished based on the combination of three properties, concerning:

- Configuration: whether or not a functional or structural relationship exists between parts or between part and object.
- Homeomerity: when parts are the same kind of thing as the whole, they are homeomerous.
- Invariance: parts are invariant when they cannot be separated from the whole without destroying the whole.

Martin and Odell advance this into a classification of compositional relationships, as shown in table 1. Looking at these different kinds of compositions helps to understand the different types of relationships that are permitted in the FBM framework in this research project. The table lists how the kinds of compositions recognised by Martin and Odell are represented as Feature relationships in the framework of the project in this thesis.

Table 1 Compositional relationships [Martin and Odell 1995] and how they relate to the relationships in the FBM framework.

compositional relationships (Martin and Odell [1995])	Feature relationships
Component – Integral Object composition Defines a configuration of parts within a whole (Scenes – Film; Wheels – Cart).	***Decomposition***
Material – Object composition Defines an invariant configuration of parts within a whole (Milk – Cappuccino; Iron – Car). Compared to component – integral object, the parts in this case cannot be removed from the object.	***Specification***
Portion – Object composition Defines a homeomeric configuration of parts within a whole (Metre – Kilometre; Slice - Loaf). Compared to the first two, the parts are now of the same kind as the object.	***Decomposition***
Place – Area composition Defines a homeomeric and invariant configuration of parts within a whole (San Francisco – California; peak – mountain). Compared to the portion – object composition, here each homeomeric piece cannot be removed.	***Decomposition***
Member – Bunch composition Defines a collection of parts as a whole (Tree – Forest; Ship – Fleet). In the composition relationships above, the parts bear a particular functional or structural relationship to one another or to the object they comprise. In the member – bunch composition, membership of a collection is the only requirement for a part to be in the composition.	***Decomposition***

compositional relationships (Martin and Odell [1995])	Feature relationships
Member – Partnership composition Defines an invariant collection of parts as a whole (Stan Laurel – Laurel and Hardy). As compared to member – bunch compositions, these members cannot be removed from the whole.	***Decomposition***

All of the above kinds of compositions are modelled in the FBM framework as Decomposition relationships, except for the Material – Object relationship. In the perspective of this project, material characteristics are specifications of objects that are modelled with the Specification relationship. The other kinds of compositions are not further distinguished by the type of the relationship. However, the specific meaning of the relationship is to be expressed using the role name of the relationship.

Some examples of non-compositional kinds of relationships are given by Martin and Odell in contrast with the compositions. These are listed in table 2 and again compared to the Feature relationships.

Table 2 Non-compositional relationships [Martin and Odell 1995] and how they relate to the relationships in the FBM framework.

non-compositional relationships (Martin and Odell [1995])	Feature relationships
Topological inclusion Examples: Customer in the Store; Meeting in the Afternoon.	***Association***
Classification inclusion Examples: Gone with the Wind is part of the set of objects of the class Book.	***Instantiation***
Attribution Examples: Lighthouse has Height and Weight, yet Height is not part of a Lighthouse.	***Specification***
Attachment Examples: Toes are attached to Feet, and are also part of Feet (= composition), yet Earrings are attached to Ears, but are not part of Ears.	***Association***
Ownership Examples: A Bicycle has Wheels and Wheels are part of a Bicycle (= composition), yet a Girl has a Bicycle, which is not a part of the Girl.	***Association***

The Classification inclusion is clearly a Instantiation in the FBM framework. From the other relationships, the attribution is modelled as a Specification relationship between Features, where, in the given example, the Height specifies a property of the Lighthouse. All other relationships are interpreted in this project as Associations between Features: the Meeting is associated with the Afternoon, the Bicycle is associated with the Girl.

An equivalent of the Specialisation or inheritance relationship is not mentioned in the classification by Martin and Odell.

Application and interpretation of relationships

Relationships in the framework can be distinguished at two levels. At the typological level, relationships are defined within Feature Types. A relationship may be optional, but in principle a relationship at typological level implies that all instances of the particular type will instantiate the relationship. In object oriented approaches, this level of relationship is defined using the first three categories shown above.

In addition, the framework also allows relationships between instances to be defined at the level of instances only, i.e. without being defined at the typological level. This new aspect in information modelling does not normally appear in OO approaches. It is further discussed in section 8, after the introduction and discussion of the classes of Feature Types and Feature Instances that are defined in the framework's Meta Layer.

Because Feature models, especially when using these instance level relationships, allow very flexible structures of Feature relationships to develop during a design process, it is necessary that decompositions, associations, and specifications can be recognised by the modelling system. This is true in particular if the system is to use this interpretation of the relationships in order to find certain semantics in the model. Without having to attempt to interpret the semantics given by a designer to these relationships by means of labels, role names, the system can already distinguish, for instance, which relationships are merely associations, and which are specifications with more detail about the object being modelled.

For a deeper understanding of the various relationships in a Feature model it is necessary to have knowledge about the exact meaning of the individual relationships between Features, which can only be accessed by looking at the role names that are given to these relationships by the designer. Automated interpretation, for instance for the purpose of propagating operations on one Feature to other Features that are related with a specific role to that Feature, requires that the role names of relationships are also known in advance to the design system. This implies that classification and standardisation of role names is necessary.

5. Twofold instantiation

The three-layered information infrastructure for Feature modelling introduces two kinds of instantiation relationships: both Feature Types and Feature Instances are instantiations of the classes in the framework.

5

The three-layered information infrastructure (see figure 3) introduced a double instantiation relationship. Therefore the usage of a consistent terminology is required, to distinguish between the different levels of information definition and the two sorts of instantiation. The first sort of instantiation relationship, following the object oriented paradigm, is found in the model between the Feature Types and the Feature Instances. This is an instantiation relationship seen from the point of view of a designer working with the types and instances. When modelling a design, a designer selects a Feature Type and creates an instance based on the formal definition included in the type. In this perspective, *the Feature Instance is an instance of the Feature Type.*

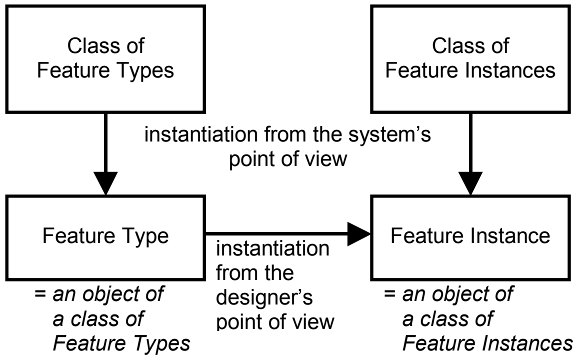

Class of Feature Types
Class of Feature Instances
instantiation from the system's point of view
Feature Type
Feature Instance
instantiation from the designer's point of view
= an object of a class of Feature Types
= an object of a class of Feature Instances

Figure 5 Two kinds of instantiation: from the system's point of view and from the designer's point of view.

However, because Feature Types can be defined 'on the fly' by designers, the structure of the Feature Instances cannot be known in advance. Yet in order to allow a computer-system to work with the new types and instances, the structure of both types and instances must follow certain rules. This is where the second sort of instantiation appears, because the *type of content* and the *type of structure* of the Feature Types and Feature Instances must be defined formally. Following the object oriented paradigm again, this is done in the Meta Layer by means of two groups of classes: classes of Feature Types and classes of Feature Instances. As a result, from the viewpoint of the computer-system, *the Feature Types are instances of the classes of Feature Types* and *the Feature Instances are instances of the classes of Feature Instances.* Hence, the relationship between the Feature Types and the Feature Instances is no longer an instantiation relationship. Yet, the designer wants to retain this point of view on the instantiation relationship. Therefore the system should maintain the content and structure of the Feature Instances in such a way that they can be presented to the designer as instances of the Feature Types. Internal to the system, however, both Feature Instances and Feature Types are instances of the classes defined in the Meta Layer.

To avoid confusion, the instances from the point of view of the system will be called 'objects', while the instances from the point of view of the designer will retain their name of 'Feature Instances'. However, for both points of view the term instantiation will remain to be used.

Following the schema in figure 5 and the terminology of objects and instances as mentioned, it can be concluded that a Feature Instance is an instance of a Feature Type, but at the same time it is an object of the class of Feature Instances. A Feature Type is an object of the class of Feature Types. Instantiation, thus, takes place in two occasions in the system's point of view: instantiation of a class of Feature Types into an object of this class; and instantiation of a class of Feature Instances into an object of this class. The latter kind of instantiation is the same event of instantiation that takes place in the designer's point of view: instantiation of a Feature Type into a Feature Instance.

6. Classes of Feature Types

This section describes in detail the definition of the classes of Feature Types, which determine the information that can be defined in terms of Feature Types.

6

The sections 6 to 8 discuss the classes of Feature Types and classes of Feature Instances that are defined in the Meta Layer of the framework. These classes are presented, using the EXPRESS-G[3] notation technique, in diagrams of four schemata: a schema *FeatureTypes*, a schema *FeatureLibraries*, a schema *FeatureInstances*, and a schema *FeatureModels*.

Feature Types are the formal representations of domain knowledge. They contain the definition of information concerning a particular concept, a typological aspect of architectural design. Chapter 6 in [van Leeuwen 1999] concludes with a proposition for a categorisation of the domain of architectural design on the basis of which Feature Types can be defined. This categorisation concerns the semantic content of the Feature Types from the architectural point of view; it does not concern the content of Feature Types from an information technology point of view. The latter point of view requires different considerations, although the end-result must correspond with the former point of view.

From the information technological point of view, a Feature Type defines what type of content a Feature may have and in what structure it is organised. Also the relationships with other Feature Types and therefore the types of relationships that can be found between Features are defined in the Feature Type[4]. From the designer's point of view, a Feature Type is said to be instantiated into a Feature Instance. Therefore, the classes of Feature Types and the classes of Feature Instances work tightly together.

Six pairs of classes of Feature Types and Instances are defined in the framework. These six classes are designed to cover the information modelling needs that are raised by the categorisation of Feature types in chapter 6 in [van Leeuwen 1999]. The classes consecutively allow the modelling of:

- simple data types
  for modelling simple values in terms of, e.g., string and numbers;
- enumerated data types
  for modelling enumerated symbolic values,
  e.g. transparent – translucent – opaque;
- complex, or aggregated, types
  for modelling concepts that have a more complex structure, these consist of relationships to other concepts;
- geometric data types
  for modelling concepts that represent a geometric shape;
- constraints
  for modelling constraints on concepts, e.g., the structural dependency between a column and a beam;
- procedural knowledge
  for modelling concepts that represent behaviour; this uses event handling as a basis.

Each of the above kinds of information is represented by a pair of classes for the Feature Type and the corresponding Feature Instance. These are formally defined and can be graphically and textually represented.

Graphical and textual notation of the classes

Working with a design support system that is based on the Feature-based framework, it will be the designer who is defining Feature Types. The design support system will then have to deal with the newly defined types of information. For the communication of the Feature Type definitions between designer and system, it is necessary to use representations of them that can be understood by both. As is usual in the area of Information

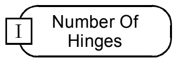
[3] EXPRESS-G is the graphical counterpart of EXPRESS, which is the data-definition language defined in ISO-10303 [ISO TC184 1994c], better known as STEP. The diagrams in this thesis use EXPRESS-G with some minor additions, for instance for the representation of the elements in an 'enumeration' entity.

[4] In section 8 it is argued that additional relationships can be defined at the level of Feature Instances.

and Communication Technology, a graphical notation technique as well as a textual notation technique[5] is used in this research project to represent both Feature Types and Feature Instances. Existing notation techniques have been found inapt for the representation of the specific characteristics of these entities in the Feature-based framework. Along with the definitions of the classes in this and the next section, the graphical and textual notation technique that has been defined for these classes are presented and examples are given.

The textual notation reminds of the declaration of classes in OO languages such as C++. It's syntax is defined using the Wirth Syntax Notation (WSN). Some choices in the syntax definition of the textual notation have been made in an arbitrary, though well considered, manner, such as the choice of valid characters for identifiers in the framework and the representation of dates. This kind of decision is not discussed here at length, since the eventual outcome is not relevant at this stage of the framework development.

The graphical notation is in many ways similar to other graphical notation-techniques, but aims to be sufficiently different in order to avoid confusion with other techniques like the EXPRESS-G notation that is used for the classes in the Meta Layer. The main entities in the graphical notation are rounded boxes representing Features and Feature Types, containing their identifier.



Basic elements of the notation

The following basic elements in the notation are not further expanded in the syntax specifications: string, number, integer, real, boolean, date, and identifier. In the syntax specifications, they are printed in italics. Their meaning is briefly described here. A string is a character-string containing any sequence of characters, enclosed in double quotes ("). Double quotes can be included in a string by preceding them with a backward slash (\). A number is any numeric value, including integers and reals. Boolean values are indicated by the keyword boolean. They can be either of the literals true and false. A date is a numeric representation of a date value, in this format: yyyymmdd.

An identifier is a character-string starting with an alphabetic character and containing any alpha-numeric characters and/or any of the following characters: ~ # $ _. It may contain spaces and line-feeds in the graphical notation, but in the textual notation these are to be omitted or replaced with the underscore character (_).

6.1 The base class: FeatureType

The classes of Feature Types are based on an abstract[6] class called FeatureType. This base class defines characteristics applicable to all Feature Types, including their identification, the name of their author, their date of definition, and a description. The base class is graphically presented in figure 6.

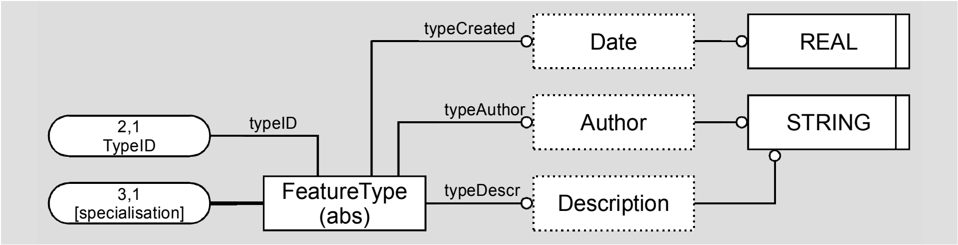


Figure 6 Diagram 1 of Schema FeatureTypes: Definition of the abstract base class FeatureType.
(This is a preliminary diagram, for the eventual diagram, see figure 19 on page 26.)

[5] Examples of textual representations are programming languages such as Pascal, C, Lisp, Prolog, et cetera, and data definition languages such as the EXPRESS language developed in ISO 10303 or STEP. These textual representations are formal and explicit enough to be unambiguously interpreted by computer-systems, and still readable enough for humans. They form the most basic medium of communication between human and computer.

Examples of graphical representations are the notation techniques developed for data modelling methods, such as IDEF, NIAM, EXPRESS-G, and the OO approaches developed by, e.g., Booch, Meyer, Rumbaugh, et cetera. Originally, the purpose of these notation techniques was to support software developers in the early stages of software design. However, they are now evolving to become the main medium in advanced interfaces for software development.

[6] An abstract class, in object oriented contexts, is a class that can not be instantiated, i.e. no objects of such a class can be created. The purpose of this kind of class is merely to serve as a super-class for sub-classes that are derived from the super-class, allowing these sub-classes to share common characteristics. A base class is a super-class that itself has no super-classes.

Feature Types are identified by their `typeID`, which is an instance of the class `TypeID`, see figure 7. A `TypeID` is built up using a rather flat structure, which is related to the way Feature Types are to be classified in Feature Type Libraries. The `TypeID` has a `typeName` and a `sectionID`, which in turn has a `sectionName` and a `libraryName`. This structure conforms to the categorisation of Feature Types in sections within Feature Type Libraries (see also figure 22 on page 29).

`TypeIDs` of Feature Types are used for reference within the definition of other Feature Types and within the context of Feature Instances in Feature models. Within its scope the `TypeID` should be unique. This is likely to be a precarious matter, since designers are free to define their own libraries of Feature Types and, at the same time, want to use standardised or commercial Feature Type Libraries. Having computers generate unique IDs is in itself not a complicated issue, yet generating meaningful unique IDs obviously is.

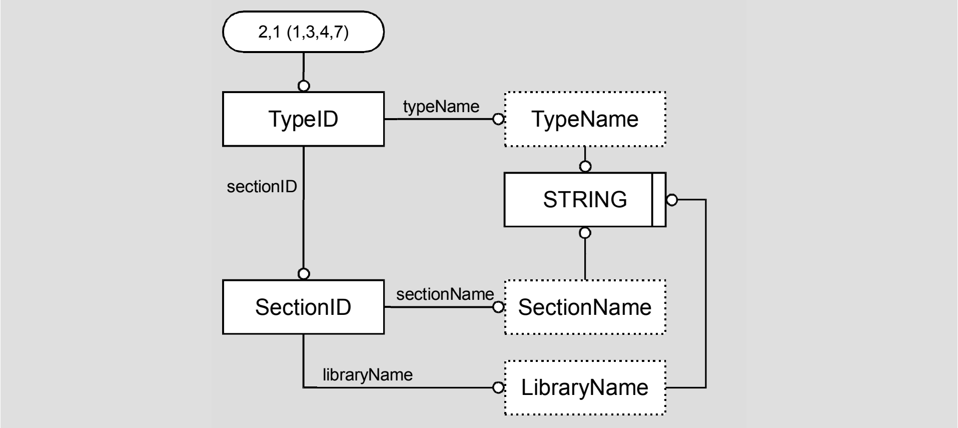


Figure 7 Diagram 2 of Schema FeatureTypes: Definition of ***`TypeID`****.*

Notation of the abstract base class ***`FeatureType`***

Since the base class `FeatureType` is an abstract class, objects of this class cannot be created. Therefore this class does not need a graphical notation. However, the notation of the attributes of the `FeatureType` class is important, since they form the basis for the notation of all other types. Referring to figure 6, the attributes of the `FeatureType` class are: `typeID`; `typeCreated`; `typeAuthor`; and `typeDescr`. The `typeID` is built up from a `libraryName`, a `sectionName`, and a `typeName`.

Syntax of the attribute `typeID` of the class `FeatureType`

`typeID = sectionID '.' typeName .`
`sectionID = libraryName '.' sectionName .`
`libraryName = identifier .`
`sectionName = identifier .`
`typeName = identifier .`

The `libraryName` and `sectionName` are included in the `typeID` to allow a Feature Type to be uniquely identified. In the Graphical notation of the Feature Types, the `typeName` is sufficient to identify a type, given that all other types within the context of a diagram are defined in the same section and library and the section and library are named in the caption of the diagram. If more than one section is represented in the diagram, the `sectionNames` need to be included in the `typeIDs` of those types that are defined in a section different from the section mentioned in the caption of the diagram. The same is true for the level of libraries: types from libraries other than the one mentioned in the diagram's caption need to include the `libraryName` in their `typeID`.

In examples, the `sectionName` and `libraryName` are not required in neither graphical nor textual notation, as is demonstrated in most of the examples throughout this chapter. The other attributes of `FeatureType` do not have a graphical notation, but are textually represented as is shown below.

```
        typeCreated = 'TypeCreated' '{' date '}' .
         typeAuthor = 'TypeAuthor' '{' string '}' .
          typeDescr = 'TypeDescr' '{' string '}' .
```

The general structure for the syntax of a Feature type definition is given below. The attribute `typeBehaviour` is included here in the textual notation, but defined only after the introduction of the subclass Handler Feature Type in section 6.2.6 on page 27 (see also figure 19 on page 26).

General syntax for the definition of Feature Types

```
           type-def = simple-type-def | enum-type-def |
                      complex-type-def | geometric-type-def |
                      constraint-type-def | handler-type-def .
      standard-body = typeCreated typeAuthor typeDescr
                      [ typeBehaviour ] .
```

6.2 Subclasses of the class `FeatureType`

The diagram in figure 8 shows the set of Feature Type classes that can be instantiated. All of these subclasses inherit the attributes of the base class `FeatureType`. There are six subclasses of Feature Types. Simple Feature Types can be used for the definition of Feature Types that represent simple data types. Enumeration Feature Types are used for the definition of a data type that is defined by a list of names, enumerating the possible values that instances of that data type may assume. The class of Complex Feature Types allows the definition of structures of Feature Types, using the various kinds of relationships as discussed in section 4. Geometric Feature Types are defined for the specific representation of geometric data. Constraint Feature Types provide the ability to define constraints on the values of Feature Instances and on the relationship

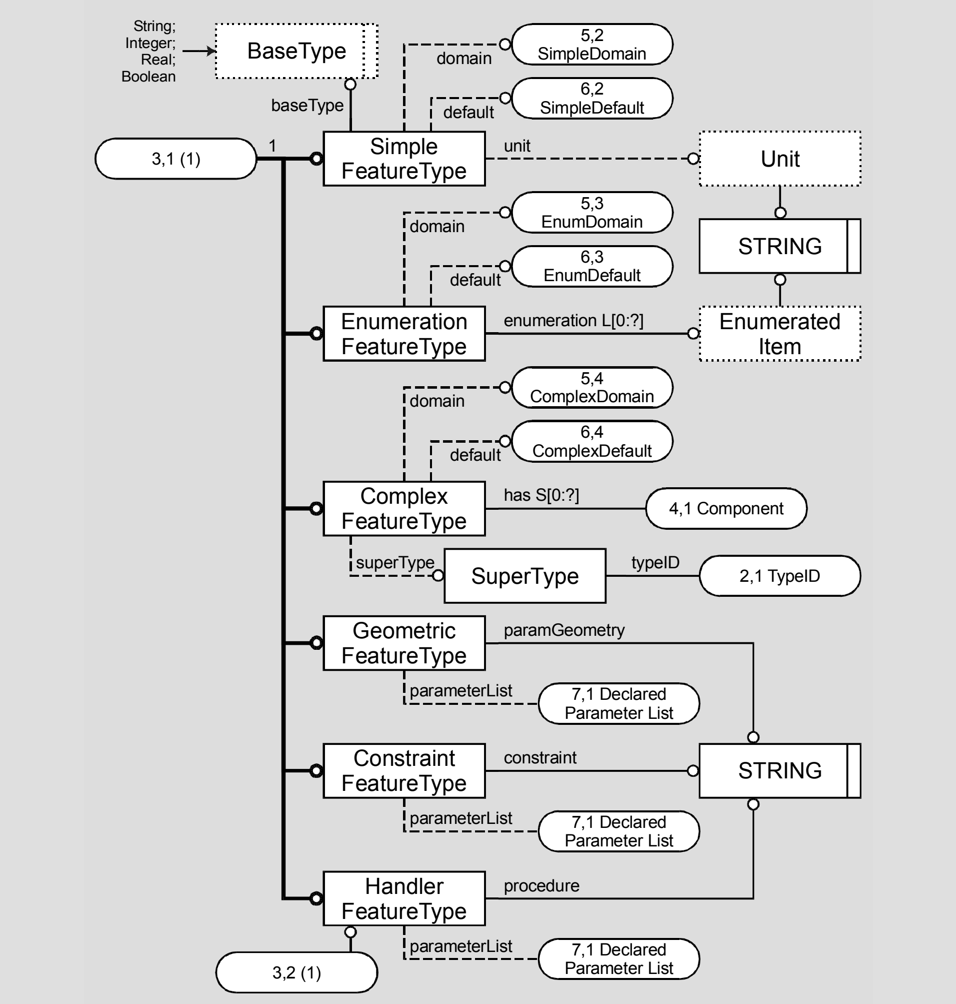


Figure 8 Diagram 3 of Schema FeatureTypes: Definition of the subclasses of `FeatureType`.

between Feature Instances. Handler Feature Types, finally, are used for the definition of behaviour, they will be used to attach behaviour to the level of Feature Types and the level of Feature Instances as well.

In the remainder of this section, these various subclasses of `FeatureType` are discussed in more detail and their notation and examples are presented.

6.2.1 The class `SimpleFeatureType`

This class facilitates the definition of Feature Types that represent simple data such as integers, reals, character strings, and boolean data. The type of data that the Feature Type defines is specified in the value of the `baseType` enumerated attribute. Every Simple Feature Type must specify its base type. A `unit` for the data may be specified if relevant, e.g. `m2` or `W/m2`, and a default value for the simple data can be included. Simple Feature Types may also specify a domain for the values that instances may assume. Domains and defaults are discussed in detail in section 6.3.

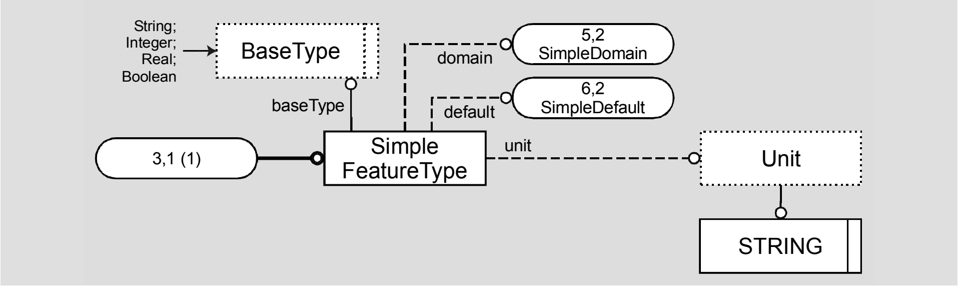


Figure 9 Excerpt from Diagram 3 of Schema FeatureTypes (see figure 8 on page 15): Definition of the class `SimpleFeatureType`.

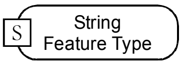
Notation of the class `SimpleFeatureType`

The four different simple data types that can be defined each have, for reasons of legibility, a distinct notation, both graphical and textual. The general syntax for a Simple Feature Type definition is as follows:

General syntax for the definition of Simple Feature Types

```
     simple-type-def = string-type-def | integer-type-def |
                       real-type-def | boolean-type-def .
```

The graphical notation for Simple Feature Types with base-type string consists of a rounded box with the `typeID` of the Feature Type and a letter `S` in a square on the left.



Syntax for the definition of string-based Simple Feature Types

```
     string-type-def = 'string' typeID '{' string-body '}' .
         string-body = standard-body [ string-domain-decl ]
                       [ string-default-decl ] .
  string-domain-decl = 'TypeDomain' '{' string-domain '}' .
 string-default-decl = 'TypeDefault' '{' string '}' .
```

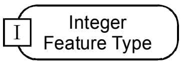
Domain and default for Simple Feature Types are not graphically represented. The syntax for domains can be found in section 6.3.

Example of a string-based Simple Feature Type

S
Spatial
Function

```
string Briefing.SpatialPlan.SpatialFunction {
   TypeCreated {19980704}
   TypeAuthor {"Jos van Leeuwen"}
   TypeDescr {"Function of a space"}
   TypeDefault {"Living"}
}
```

For reasons of brevity and clarity, in the remaining examples the standard-body part of the definition, including the date, author, and description of a type, is left out. Also, the `libraryName` and `sectionName`, in the above example `Briefing` and `SpatialPlan` respectively, are omitted. For the other Simple Feature Types, the syntax is defined similarly, as is shown in the following.



Syntax for the definition of integer-based Simple Feature Types

```
    integer-type-def = 'integer' typeID '{' integer-body '}' .
        integer-body = standard-body [ numeric-domain-decl ]
                       [ integer-default-decl ] [ unit-decl ] .
 numeric-domain-decl = 'TypeDomain' '{' numeric-domain '}' .
integer-default-decl = 'TypeDefault' '{' integer '}' .
           unit-decl = 'TypeUnit' '{' string '}' .
```

Example of a integer-based Simple Feature Type

```
integer NumberOfHinges {
   TypeDomain {2,3,..5}
   TypeDefault {3}
}
```

Syntax for the definition of real-based Simple Feature Types

```
       real-type-def = 'real' typeID '{' real-body '}' .
           real-body = standard-body [ numeric-domain-decl ]
                       [ real-default-decl ] [ unit-decl ] .
   real-default-decl = 'TypeDefault' '{' real '}' .
```

Example of a real-based Simple Feature Type

```
real Area {
   TypeDomain {[0,->>}
   TypeDefault {25.9}
   TypeUnit {"m2"}
}
```

Syntax for the definition of boolean-based Simple Feature Types

```
    boolean-type-def = 'boolean' typeID '{' boolean-body '}' .
        boolean-body = standard-body [ boolean-domain-decl ]
                       [ boolean-default-decl ] .
 boolean-domain-decl = 'TypeDomain' '{' boolean-domain '}' .
boolean-default-decl = 'TypeDefault' '{' boolean '}' .
```

Example of a boolean-based Simple Feature Type

```
boolean IsExterior {
   TypeDefault {true}
}
```

6.2.2 The class `EnumerationFeatureType`

A simple data type that cannot be defined using the class `SimpleFeatureType` is the enumeration. Enumerations are data types that serve to identify a single selection from an enumerated list of names. The Enumerated Feature Type includes an ordered set of character strings denoting the identifiers that instances of the enumeration may assume as their value. A default value for the enumeration type can be included, as well as a domain for the value of instances. A domain for this Feature Type, that already specifies a range of possible values, seems redundant or even irrelevant, but is included for completeness. It may bear relevance in cases where a (temporary) limitation on the enumeration is required.

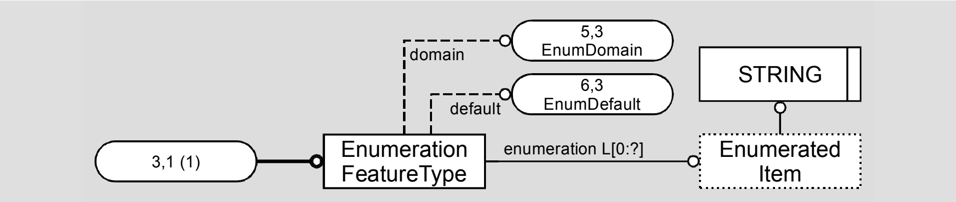
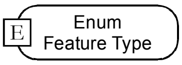


Figure 10 Excerpt from Diagram 3 of Schema FeatureTypes (see figure 8 on page 15): Definition of the class `EnumerationFeatureType`.

Notation of the class `EnumerationFeatureType`

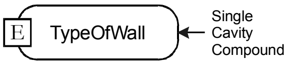
The class of Enumeration Feature Types is graphically noted by a letter E in the square of the symbol. Its textual notation is similar to those of the Simple Feature Types, but includes the ordered set of enumerated identifiers preceded by the label `TypeItems`. In the graphical notation, the enumerated identifiers are not necessarily made visible, but can be indicated by listing all identifiers and connecting this list with the enumeration symbol using an arrow line. The syntax for domains can be found in section 6.3.



Syntax for the definition of Enumeration Feature Types

```
       enum-type-def = 'enum' typeID '{' enum-body '}' .
           enum-body = standard-body enum-decl [ enum-domain-decl ]
                       [ enum-default-decl ] .
           enum-decl = 'TypeItems' '{' enum-items '}' .
          enum-items = identifier { ',' identifier } .
    enum-domain-decl = 'TypeDomain' '{' enum-domain '}' .
   enum-default-decl = 'TypeDefault' '{' identifier '}' .
```

Example of an Enumeration Feature Type



```
enum TypeOfWall {
   TypeItems {Single, Cavity, Compound}
   TypeDefault {Compound}
}
```

6.2.3 *The class* `ComplexFeatureType`

Information that is in any sense structured in a more complex manner needs to be defined using the class `ComplexFeatureType`. This class defines a simple composition structure using a set of `Components`. A `Component` is a *reference* to another Feature Type; Complex Feature Types do not *contain* other Feature Types (compare object attributes in OO approaches), but merely refer to other Feature Types. This restriction is very valuable for the resulting flexibility of information models, mainly because it allows Feature Types to share components. However, this approach also introduces some potential complications, in particular for the integrity management of the Feature Types and Instances, for example when Feature Types are modified. A similar structure of references is found at the instance level of Complex Features, which allows Feature Instances to share components. An example of such a structure is given in figure 31 on page 36.

The usage of `ComplexFeatureType` can lead to both tree-like (hierarchical) and lattice-like information structures. The contents and structure of `Components` are discussed in detail in the next subsection and presented in figure 12. A default for the Complex Feature Type can be specified, which refers to an instance within the same library-section; the existence of instances in Feature Type Libraries is addressed in a separate discussion in section 6.4. A Complex Feature Type also has an optional domain for the instances of the type, referring to a set of instances within the same library-section (see also the discussion and diagrams for defaults and domains in section 6.3).

A `ComplexFeatureType` optionally relates to a `SuperType`, which refers to the `TypeID` of a Feature Type forming the `ComplexFeatureType`'s supertype. This establishes single inheritance of Complex Feature Types. One restriction is in effect here: only `ComplexFeatureTypes` can be the supertype of a `ComplexFeatureType`. This restriction follows the fact that Complex Feature Types can themselves not contain data in the way for instance a `SimpleFeatureType` contains data. `ComplexFeatureTypes` only contain references and can therefore not inherit from, e.g., `SimpleFeatureTypes`. Other Feature Types than `ComplexFeatureTypes` are included in the 'inheritance-tree' by reference through the components. A Complex Feature Type that has a supertype inherits all the components of its supertype and those of its supertype's supertypes.

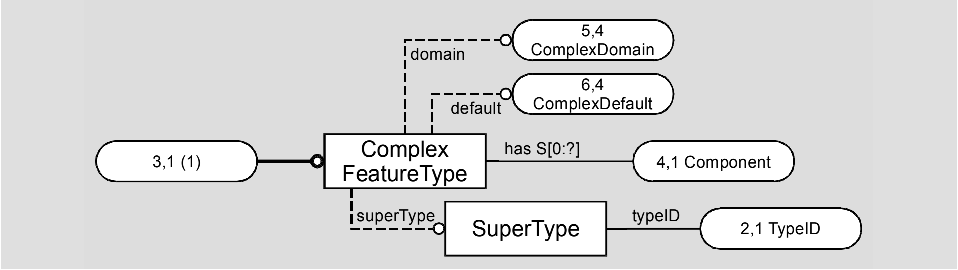


Figure 11 Excerpt from Diagram 3 of Schema FeatureTypes (see figure 8 on page 15): Definition of the class `ComplexFeatureType`.

The support-class `Component`

Complex Feature Types (objects of the class `ComplexFeatureType`) are decomposed into objects of the supporting class `Component`. Each component of a Complex Feature Type has a role indicating the kind of relationship the component has with the Complex Feature Type. The role is specified by the `roleType` (an enumerated value indicating a decomposition, association, or specification type of role; note that specialisation is not a valid role type, this relationship is modelled using the `superType` attribute of a `ComplexFeatureType`) and identified by its `rolename`, which is a string of characters.

`Components` are either `TypeComponents` or `InstanceComponents`. `TypeComponents` are components that are declared at the type level and given values at instantiation time. The actual data that define these components are specified during instantiation and stored in a Feature model, using the `Component` class of the FeatureInstances schema (see figure 29). At the type-level, the `TypeComponents` merely declare the *structure* of the Complex Feature Type. An example of the usage of a `TypeComponent` is the declaration of a `ComplexFeatureType` called `material` which has a `TypeComponent` called `colour`. This declaration specifies that all materials have a colour, without providing any value for the colour.

In a Complex Feature Type, `TypeComponents` can have multiple occurrences, which, at the type level, may be limited in number by means of the `Cardinality`. The cardinality of a component specifies a minimum and maximum number of occurrences that a component should have. During design, the cardinality should, however, not be enforced upon the designer. If the cardinality of a component is not specified, this means that the cardinality is `[0..1]`, which is to say that the component is a single, but optional, component. Components that are not supposed to be optional must specify the cardinality; for single components this means a cardinality of `[1..1]`.

The values that a component can assume are specified by its domain, and it can also be given a default value. Note that both domain and default override, in the context of the Complex Feature Type, the domain and default defined for the Feature Type referred to by the component. However, the contents of both domain and default must, of course, be in accordance with the Feature Type the component refers to (the different kinds of domains and defaults are presented in section 6.3).

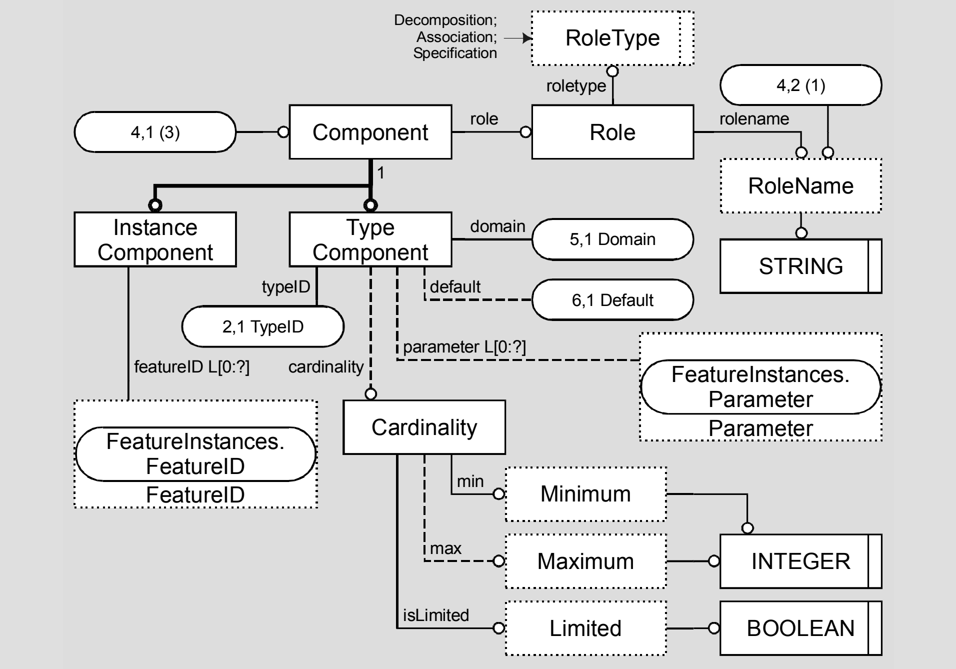


***Figure 12** Diagram 4 of Schema FeatureTypes: Definition of the support class* **`Component`** *for Complex Feature Types.*

An example to illustrate the function of the cardinality in a component is the relationship of a wall with several elements in the wall. The actual elements in a specific wall are instances of a component in the Complex Feature Type defining the wall. This component refers to the `TypeID` of the Feature Type defining the elements; it has a role with, for instance, the `roletype Decomposition` and a `rolename 'element'`; and it has a cardinality that specifies a minimum of 0 elements with no upper limit: `[0..?]`. This example is worked out in more detail on page 21 where the graphical and textual notation of the definitions of Complex Feature Types is presented.

`InstanceComponents`, as opposed to `TypeComponents`, do not merely declare the structure of a Complex Feature Type, they also define the values that are contained in the components of this structure. This means that the value of such a component is defined for all instances that are to be created from the particular Complex Feature Type. For example, a Complex Feature Type called `wooden beam` would have a component of a type called `material`, but this component should always represent the characteristics for the material wood. Therefore this component does not refer to the `TypeID` of a Feature Type called `material`, but rather to the `FeatureID` of the Feature Instance called `wood` of the Feature Type `material`. This instance must be stored in a Feature Type Library, since it is referred to directly in the definition of Feature Types. Thus, Feature Type Libraries do not only contain definitions of Feature Types, but also definitions of Feature Instances that have significance at the type-level[7]. These Feature Instances can also be referred to at the instance level. More on this aspect is found in section 6.4. `InstanceComponents`, like `TypeComponents`, can refer to multiple Feature Instances.

Because this part of the definition of the Complex Feature Type represents an invariant information structure, the cardinality for this relationship need not be defined. Although the type of the Feature Instances that participate in an `InstanceComponent` is not specified for the component, they should all be of the same type.

Components in a Complex Feature Type can refer to any other class of Feature Type, including the Geometry, Constraint, and Handler Feature Types. These three types, which are defined further on in this section, may accept parameters for their functionality and behaviour. It is by means of these parameters how information is communicated from the Complex Feature Type and its properties to, for instance, a geometry component that is to represent that Feature Type.

For example: a Complex Feature Type called `Door` may be defined with a specification by two components `h` and `w` of the types `Height` and `Width` respectively, and an association to a type `DoorGeometry` that represents the door geometrically. The `DoorGeometry` must somehow be related to the door’s height and width, which is done by passing the components `h` and `w` as parameters to the `DoorGeometry` component. This example is elaborated after the definition of the Geometric Feature Type on page 23.

Notation of the class `ComplexFeatureType`

Complex
Feature Type

Syntax for the definition of Complex Feature Types

```
    complex-type-def = ‘complex’ typeID [ ‘(’ super-typeID ‘)’ ]
                       ‘{’ complex-body ‘}’ .
        super-typeID = typeID .
        complex-body = standard-body [ complex-domain-decl ]
                       [ complex-default-decl ]
                       { decomp-decl | assoc-decl | spec-decl } .
 complex-domain-decl = ‘TypeDomain’ ‘{’ complex-domain ‘}’ .
complex-default-decl = ‘TypeDefault’ ‘{’ featureID ‘}’ .
         decomp-decl = ‘Has’ component-decl ‘;’ .
          assoc-decl = ‘Assoc’ component-decl ‘;’ .
           spec-decl = ‘Spec’ component-decl ‘;’ .
      component-decl = ( type-component-decl | inst-component-decl ) .
 type-component-decl = typeID roleName [ cardinality ]
                       [ ‘{’ domain ‘}’ ] [ param-list ]
                       [ ‘=’ default ] .
 inst-component-decl = roleName [ ‘[’ integer ‘]’ ] ‘=’ featureID .
            roleName = identifier .
         cardinality = ‘[’ number ‘..’ ( number | ‘?’ ) ‘]’ .
              domain = string-domain | numeric-domain | boolean-domain |
                       enum-domain | complex-domain .
             default = string | number | boolean | featureID .
```

Parameters can be passed at the instance level or at the type level. Passing parameters at the type level means that the role names of components of a Complex Feature Type are passed as parameters and that all instances of that Complex Feature Type use this way of passing parameters. The above example shows this approach. Another form of passing parameters at the type level is to pass identifiers of Feature Instances created at the type level as parameters. This form is necessary, for instance, when constant values must be passed at the type

[7] This aspect of the framework is to some extent comparable to the usage of, for instance, static data in class declarations in C++.

level; Feature Instances are required to provide these constant values. The issue of Feature Instances at the type level is further discussed in section 6.4.

Passing parameters at the instance level is discussed in section 7.2 on page 36. Parameters passed at the instance level override parameters that are passed at the type level. The attribute `parameter` is defined on page 37.

The syntax for the class `ComplexFeatureType` provides the possibility to include, in parentheses after the identifier of the type, the `typeID` of the super-type for the defined Feature Type, denoting a Specialisation relationship. The relationships to the components of the Complex Feature Type are textually denoted in the body of the notation by three keywords; `Has`, `Assoc`, and `Spec`, for respectively Decomposition, Association, and Specification relationships. These relationships are further declared by specifying the `typeID` of the component, its role name, and optionally its cardinality, domain, and default.

The term `featureID`, used for the defaults and instance components in a Complex Feature Type, is defined in section 7.1 on page 32. The term `param_list` is defined in section 7.2 on page 37.

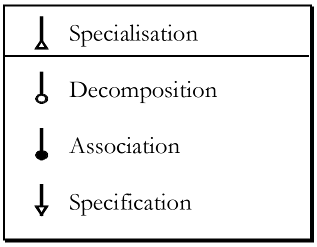


Figure 13 Relationship symbols

Before an example of a Complex Feature Type is presented, the graphical notation of the different relationships need to be defined. Relationships are shown by a line connecting the two related Feature Types. The relationships are to be interpreted bi-directional, yet one direction is emphasised and given a `roleName`. The emphasised direction of a relationship is graphically marked by a symbol at the end of the line. This symbol also indicates the type of the relationship as is shown in figure 13.

Specialisation relationships can be abbreviated by joining the two symbols of supertype and subtype, as shown in figure 14. The Feature Type at the beginning of the line is also the type that, in the textual notation, contains the relationship as a component. Yet, in the case of a specialisation, it is the subtype at the end of the line that declares the relationship in the textual notation. Note that specialisation relationships can only be modelled through the super-subtype construction of Complex Feature Types.

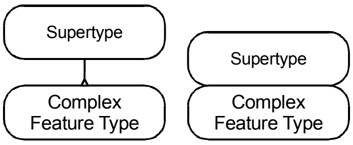
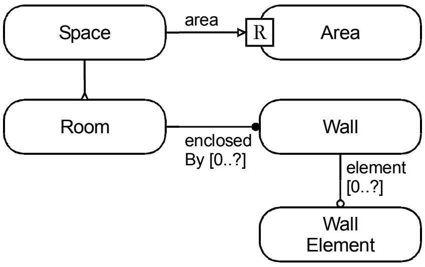


Figure 14 Abbreviated inheritance relationship.

Example of a Complex Feature Type



```
real Area {
   TypeDefault {0.0}
}

complex Space {
   Spec Area area {[12,->>} = 27.5;

complex Room(Space) {
   Assoc Wall enclosedBy[0..?];
}

complex Wall {
   Has WallElement element[0..?];
}

complex WallElement {
}
```

In the above example, the Complex Feature Type `Space` has a specification relationship to the type `Area` with the role named `area`. The domain for this specification is such that the minimum area for the space is 12, and its default is 27.5. Both domain and default are given for the *component* `area` to override any domain and default defined in the *type* `Area`.

Note that the Feature Type `WallElement` is a Complex Feature Type that has no components. This is to demonstrate and remind that the definition of Feature Types is not a determined process, but is an ongoing activity during the design process. The details regarding the composition of the wall elements will be specified in later stages during design. Perhaps the type `WallElement` will be replaced by another, readily detailed Feature Type.

The support class `DeclaredParameterList`

The three classes of Feature Types that remain to be defined, Geometric, Constraint, and Handler Feature Types, all use parameters to provide either external or internal processes with access to the information in a Feature model. In the case of Geometric Feature Types, they provide an external, parametric, geometric

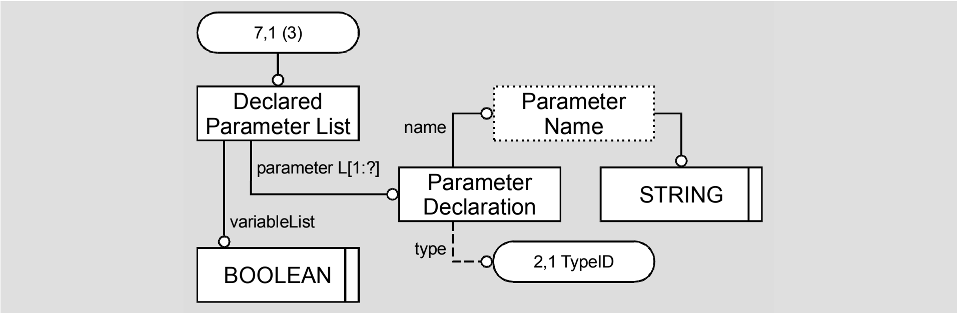


Figure 15 Diagram 7 of Schema FeatureTypes: Definition of the support class `Declared ParameterList`.

modeller with the parameters required to generate the desired geometries. Handler Feature Types contain procedures that require access to the Features available in a model. This communication of Feature data to those processes is done using lists of parameters. The support class `DeclaredParameterList` is defined to allow those lists of parameters to be declared; it is used to specify the type of parameters that will be used by the various processes.

A `DeclaredParameterList` is an ordered list of parameter declarations. Ordered, because the order of the parameters in this list is important for passing them to the accessing process. Parameter declarations consist of a name for the parameter and the identifier of the Feature Type that the parameter must be an instance of. If this type is not specified, then any type of Feature Instance may be passed for this parameter; the parameter is then untyped. Figure 15 shows the definition of the support class `DeclaredParameterList`. The list of parameters should at least contain one parameter, but may be of variable length, meaning that the parameter of the last declared type may be passed repeatedly. If this is the case, the flag `variableList` is set to `True`.

Notation of declared parameter lists

Declared parameter lists are used by Geometric Feature Types, Constraint Feature Types, and Handler Feature Types. The textual notation of a declared parameter list is in parentheses after the `typeID` of the Feature Type it belongs to. Each parameter declaration is shown by the Feature Type of the parameter and the name of the parameter. If the type is not significant, it is replaced by the keyword `ANY`. A parameter list can be of variable length, which is indicated by an ellipsis after the last parameter (at least one parameter must be declared in a parameter list). This is not graphically represented.

In the graphical notation, parameters are shown as relationships to the Feature Types of the parameters, with an association symbol. The name of the parameter is written next to the relationship line. Untyped parameters are indicated using the symbol shown on the right.

Syntax for the declaration of parameter lists

```
    decl-param-list = param-decl { ',' param-decl } [ ',...' ] .
         param-decl = ( typeID | 'ANY' ) param-name .
         param-name = identifier .
```

6.2.4 *The class `GeometricFeatureType`*

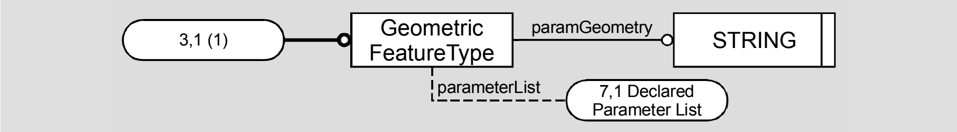


Figure 16 Excerpt from Diagram 3 of Schema FeatureTypes (see figure 8 on page 15): Definition of the class `GeometricFeatureType`.

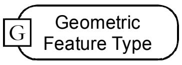
The class for Geometric Feature Types is included here but not yet detailed. A relationship with some form of parametric geometry is predictably necessary. However, how this parametric geometry should be defined has not been part of this research project; it is merely represented in the framework by a string indicating the type of geometry, e.g. sphere, box. At this point in the project, it is simply assumed that a parametric geometry modeller module will become a part of the design support system, and that this module can access the Geometric Features and their parameters in the model. The basis for such a module could be found in commercially available geometric modelling engines such as Parasolid[8] and ACIS[9].

[8] Parasolid is a trademark of Parametric Technology Corporation (www.ptc.com).

The types of the parameters that the parametric geometry requires, are defined by the attribute `parameterList` of the Geometric Feature Type, which is a list of parameter declarations. The actual passing of parameters is done either at the instance level by means of a Geometric Feature Instance, or at the type level where a Geometric Feature Type is referenced as a component in a Complex Feature Type, by means of the inclusion of a parameter list for that component (see also page 20).

Notation of the class `GeometricFeatureType`

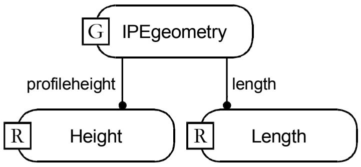
The graphical notation of Geometric Feature Types includes a letter G in the square box on the left of the symbol. In the textual notation, the type of geometry is indicated in the body of the definition, using the keyword `TypeGeometry`. The notation of the declared parameter list is already discussed separately.



Syntax for the definition of Geometric Feature Types

```
geometric-type-def = 'geometry' typeID '(' [ decl-param-list ] ')'
                     '{' geometric-body '}' .
    geometric-body = standard-body geometry-type .
     geometry-type = 'TypeGeometry' '{' identifier '}' .
```

Example of a Geometric Feature Type

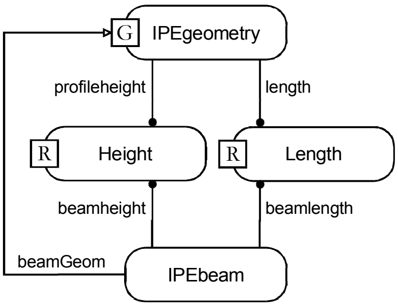


```
geometry IPEgeometry (Height profileheight,
   Length length) {
   TypeGeometry {IPE}
}
```

The above example shows the notation of a Geometric Feature Type called `IPEgeometry`. It represents a parametric geometry that is indicated by the name IPE and declares two parameters, one of the Feature Type `Height`, with the name `profileheight`, another of the Feature Type `Length`, with the name `length`. This assumes that there exists a parametric geometry with the name `IPE` that takes two parameters, specifying the height and length of the IPE profile, and that has embedded knowledge about the other dimensions of the profile, based on the given `height` parameter.

A second example shows how Geometric Feature Types can be related to from within a Complex Feature Type.

Example of using a Geometric Feature Type as a component in a Complex Feature Type



```
geometry IPEgeometry
   (Height profileheight,
    Length length) {
   TypeGeometry {IPE}
}

complex IPEBeam {
   Spec Height beamheight;
   Spec Length beamlength;
   Spec IPEgeometry beamGeom
   (SELF.beamheight, SELF.beamlength);
}
```

This example demonstrates how the components `beamheight` and `beamlength` of the Complex Feature Type `IPEbeam` serve as parameters to the component `beamGeom`. Graphically this relation cannot be expressed, the diagram only shows the declaration of the parameters of `IPEgeometry` and its role as a component in the `IPEbeam` type.

The definition and syntax of passing parameters is described in section 7.2 on page 37.

6.2.5 The class `ConstraintFeatureType`

Constraints form the basis for the relationship of information models with constraint solving and constraint checking modules of design support systems. Constraints have been applied in design to model and maintain restrictions on the geometry of designed objects.

[9] ACIS is a trademark of Spatial Technology (www.spatial.com).

One approach to applying constraints in geometric modelling situations, is to use temporal interval algebra. Allen [1983] defines five relationships between a point and an interval: ahead, front-touch, in, back-touch, and behind. This approach has been used in [Kelleners et al. 1997; Kelleners 1999] to develop an object-oriented constraint solver that handles interval-interval constraints that can be applied on box geometries. The solver recognises the following five types of constraints: connection constraints (e.g. touch, align sides), distance constraints, a contains constraint, a non-intersection constraint, and unary constraints (e.g. fixed position, fixed side, fixed dimension, fixed orientation, minimum dimension, maximum dimension). The latter are not interval constraints but necessary for the solving process.

Dohmen [1998] describes a system that implements constraint-based Feature validation. The system handles constraint types such as attach constraints, which allow the attachment of geometric Features to one another, and semantic constraints for topologic properties, comparable to the connection and non-intersection constraints by Kelleners. Furthermore, Dohmen deals with geometric constraints, specifying e.g. the position, or distance, of Features relative to each other. Algebraic constraints relate more general parameters of Features, for instance specifying dimensional proportions. Dimension constraints restrict the domain of Feature parameters.

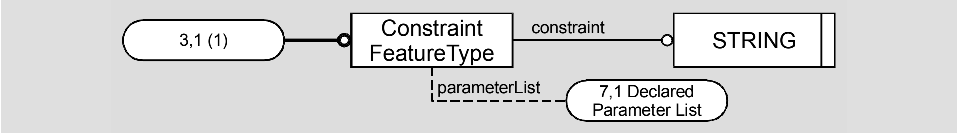


***Figure 17** Excerpt from Diagram 3 of Schema FeatureTypes (see figure 8 on page 15): Definition of the class `ConstraintFeatureType`.*

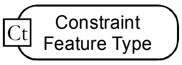
Although the technology of constraint solving as such is not a part of this research project, their integration in design support systems is anticipated by the definition of the Constraint Feature Types. In this project, a constraint is identified in a Constraint Feature Type by the attribute `constraint` which is represented as a string. The parameters used by the constraint are declared by the list of parameter declarations that is represented by the attribute `parameterList`. In principle, relating the Feature model to constraints in this manner is not restricted to constraints on geometric data only, but is open to specifying constraints on any other type of data as well. It allows constraints to be applied on any other design aspect, such as structural design or construction planning. The sole restriction is that the constraint can be reduced to some form of mathematical equation or set of equations for which a solver is available. Two such applications of Constraint Feature Types are clear:

Implementation of 'derived attributes' of a Feature, e.g. the calculated outcome of an algebraic equation with reference to various locations in the Feature model. An alternative approach for implementing a 'derived attribute' is by determining its value using an procedure in a Handler Feature Type (see below).

Automated management of integrity and consistency of the information in a Feature model. Rules for integrity management that can be expressed using algebraic equations can be modelled through Constraint Feature Types. More complicated procedures require the Handler Feature Types.

Notation of the class `ConstraintFeatureType`

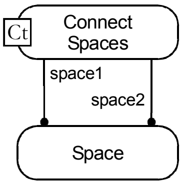
The graphical notation of Constraint Feature Types includes the letters Ct in the square box on the left of the symbol. The string representing the constraint is shown in the textual notation using the keyword `TypeConstraint`. The declared parameter list is noted textually and graphically as has been discussed before.



Syntax for the definition of Constraint Feature Types

```
constraint-type-def = 'constraint' typeID '(' [ decl-param-list ] ')'
                      '{' constraint-body '}' .
    constraint-body = standard-body constraint-type .
    constraint-type = 'TypeConstraint' '{' identifier '}' .
```

Example of a Constraint Feature Type



```
constraint ConnectSpaces (Space space1,
   Space space2) {
   TypeConstraint {adjacentTo}
}
```

6.2.6 *The class* `HandlerFeatureType`

Event handling will be an important mechanism in design support systems. Its purpose is to incorporate procedural knowledge in both the conceptual model (the Feature Types) and the actual models (the Feature Instances). Using event handlers, procedural knowledge can be captured in models concerning, e.g., instantiation of types, user-interaction, validation, problem-solving, and model-evolution.

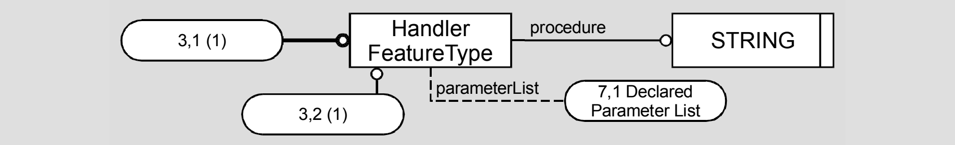


Figure 18 Excerpt from Diagram 3 of Schema FeatureTypes (see figure 8 on page 15): Definition of the class **`HandlerFeatureType`*.***

Handler Feature Types, as presented in figure 18, include a procedure that is executed in response to the occurrence of an event. It is beyond the scope of this thesis to specify in which form the procedures are implemented and how they are executed by design support systems. However, some of the required functional capabilities of these procedures can be deduced from the following issues. Information must be passed to the procedure concerning the context in which it is called. For this purpose, a list of parameters is declared for the Handler Feature Type (see the attribute `parameterList` in figure 18 and its definition in figure 15). Besides the information passed through this list to the procedure of an event handler, event handlers need to have 'knowledge' about their direct environment, such as the Feature Instance it is attached to (this is the Feature Instance that notifies the event handler that the event has occurred). This also includes the relationships of this Feature Instance, both typological and instance level relationships (see section 8). To have knowledge about its environment, in this case, means that from within the evaluated procedure, data can be accessed that is located in the notifying Feature Instance, and data can be retrieved from the relationships of this notifying Feature Instance. The procedure should have access also to data that is not directly related to the notifying Feature Instance, but that is located elsewhere in the Feature model. This capability provides a way for event handlers to manifest themselves as 'active' relationships between Feature Instances in a model. It allows, for instance, modifications to a Feature Instance to have an effect on other parts of the Feature model.

Most likely, also knowledge external to the Feature model is relevant for the evaluation of the procedure of an event handler. The integration of specific applications with a Feature modelling system, for instance a cost calculation module, would require access to external data, such as catalogues, price-lists, and other cost related information.

Although its importance is well acknowledged, the exploration of the possibilities of event handling mechanisms in a design support system is not elaborated within the scope of this research project. Some of the envisioned applications of event handling include the following.

Modelling dependencies between parts of a Feature model by means of event handlers. An example is the instantiation, modification, or deletion of Features as a result of the occurrence of a particular event.

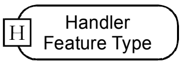
Implementation of 'derived attributes' of a Feature, e.g. the calculated outcome of an algebraic function with input from various locations in the Feature model. An alternative approach for implementing a 'derived attribute' is by determining its value using an equation in a Constraint Feature Type. Handler Feature Types are expected to introduce additional functionality for those cases where a procedural approach for the determination of the derived attribute's value is required.

Automated management of integrity and consistency of the information in a Feature model. Here, a similar consideration can be made: rules for integrity management that can be expressed using algebraic equations can be modelled through Constraint Feature Types. More complicated procedures require the Handler Feature Types.

Integration of external information sources in a Feature-based design support system, leading to enhancement of the issues mentioned above. For this application of Handler Feature Types, their procedure must be allowed to access the external sources.

Notation of the class `HandlerFeatureType`

The graphical notation of Handler Feature Types includes a letter H in the square box on the left of the symbol.

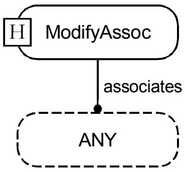


Syntax for the definition of Handler Feature Types

```
    handler-type-def = ‘handler’ typeID ‘(’ [ decl-param-list ] ‘)’
                       ‘{’ handler-body ‘}’ .
        handler-body = standard-body handler-procedure .
   handler-procedure = ‘Procedure’ ‘{’ code ‘}’ .
```

The declared parameter list of the Handler Feature Type is optional and included in parentheses after the `typeID` attribute. The term `code` is not further specified, because the syntax and semantics of the language of procedures have not been defined in this project. In the example below, the code is replaced by comments, explaining how the procedure will perform.

Example of a Handler Feature Type



```
handler ModifyAssoc (ANY associates) {
   TypeAuthor {"Jos van Leeuwen"}
   Procedure {
      // modify all instances that are somehow
      // associated to the notifying instance
      // and that are specified by the
      // ‘associates’ parameter.
   }
}
```

Scenarios for using Handler Feature Types

Three different scenarios are foreseen in the framework, in which event handlers are applied in a model. Two types of events are distinguished in these scenarios. In the first two scenarios, the event is triggered by a Feature Instance, signifying for instance a modification of the instance that should be reacted upon by the event handler. This kind of event is called a Feature-event. The third scenario deals with events that are triggered by the modelling system itself, for instance events signifying that an evaluation task has been started. These events are called System-event. Events are enumerated identifiers that are not further specified in the scope of this project.

Scenarios 1 and 2 for modelling event handlers: handling Feature-events

In scenarios 1 and 2, an event handler is attached to a Feature Type, which is called the notifying Feature Type. This can be any Feature Type, therefore the definition of the abstract base class `FeatureType` requires redefining. In the new definition of the class `FeatureType`, which is shown in figure 19, a Feature Type can have a set of `EventHandlers`. An `EventHandler` identifies the event it handles and is identified in the context of the notifying Feature Type by the attribute `handlername`. The difference between the scenarios 1 and 2 lies in the way parameters are passed to the event handler.

In scenario 1, the parameters are defined at the level of the notifying Feature Type, meaning that all instances use the same parameter definitions. These parameter definitions are provided by means of a single Handler Feature *Instance* that is attached to the notifying Feature *Type*. In this scenario, the `handler`

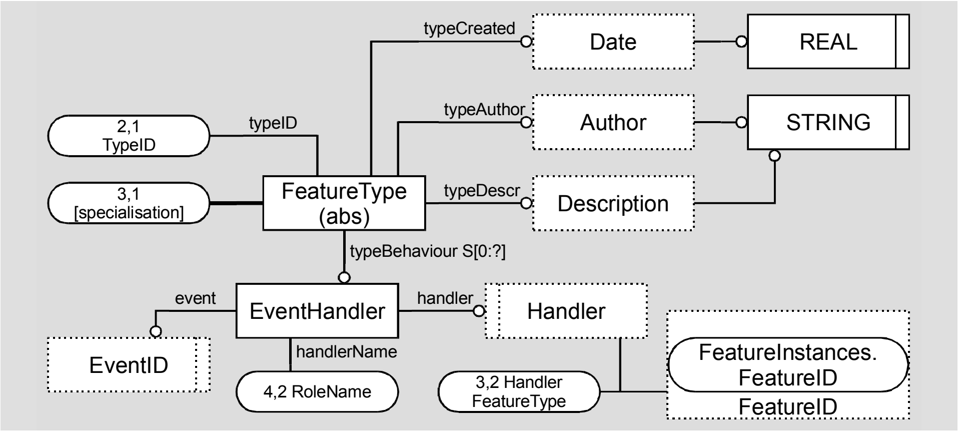


Figure 19 Diagram 1 of Schema FeatureTypes: Definition of the abstract base class `FeatureType`. (This diagram supersedes the diagram in figure 6.)

attribute of the `EventHandler` entity in figure 19 contains the `FeatureID` of a Handler Feature Instance. This scenario is useful only when all handling of the event for this Feature Type should, at any time, be passed the same parameters. However, it should be noted that a parameter may refer to the role of a component of a complex Feature Instance, which provides certain flexibility. In case the parameters differ per instance of the notifying Feature Type, scenario 2 must be followed.

In scenario 2, each instance of the notifying Feature Type has its own definition of parameters and therefore its own Handler Feature Instance. At the type level, the `TypeID` of the Handler Feature Type is specified by the `handler` attribute in the definition of the notifying Feature Type (see figure 19). At the instance level, each notifying Feature Instance has a relationship to a Handler Feature Instance that specifies the parameters. This relationship is an attribute of the class `FeatureInstance` which is defined in section 7.1.

Scenario 3 for modelling event handlers: handling System-events

The third scenario deals with System-events that are triggered by the modelling system rather than by Features. In this scenario, event handlers are not attached to Feature Instances, but are modelled by themselves: the Handler Feature Instances exist independently in the Feature model. The parameters that are specified in such a Handler Feature Instance do not depend on a notifying Feature. Examples for the three scenarios are given in section 7.2 on page 39.

Notation of the attribute `typeBehaviour` of the class `FeatureType`

The relationship of an event handler with a notifying Feature Type is defined using the attribute `typeBehaviour`, as shown in figure 19. This attribute includes either the `typeID` of the Handler Feature Type, or the `featureID` of the Handler Feature Instance, depending on how the parameters are to be passed. Furthermore, it includes the role name for the event handler in the context of the notifying Feature Type and the identifier for the event that is handled. The inclusion of the `typeBehaviour` attribute in the definition of Feature Types is already anticipated in the textual notation for the attributes of the class `FeatureType` in section 6.1. Its syntax is further defined as follows.

Syntax for the attribute `typeBehaviour` of the class `FeatureType`

```
        typeBehaviour = ‘Behaviour’ ‘{’ { event-decl } ‘}’ .
           event-decl = ( typeID | featureID ) roleName
                        ‘[’ eventID ‘]’ ‘;’ .
              eventID = identifier .
```

Examples of how to model the relationships between notifying Feature Types and Instances and Handler Feature Types and Instances are given in section 7.2 on page 39, after the definition of the class `HandlerFeature`.

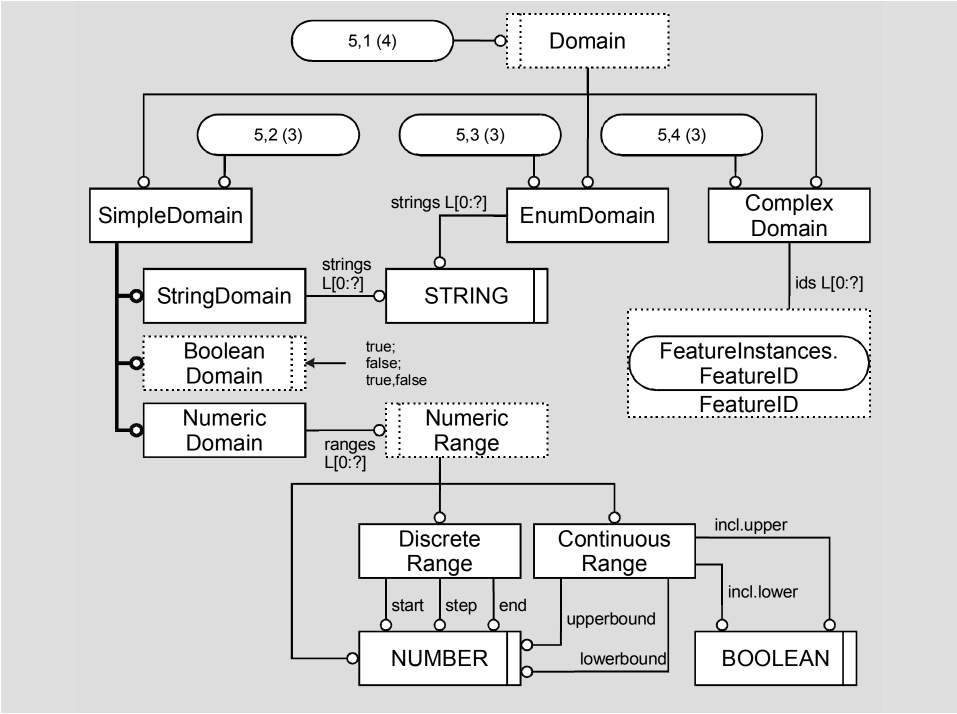


Figure 20 Diagram 5 of Schema FeatureTypes: Definition of Domains.

6.3 Domains and defaults

Feature Types, as has been described in the previous sections, may specify *domains* and *defaults* for their instantiation. The definitions of the various kinds of domains are presented in figure 20 and the different kinds of defaults are found in figure 21.

For Simple Feature Types and Enumeration Feature Types the domain specifies what values are valid for the instantiated Features. The default specifies the initial value a Feature will get during instantiation. The contents of the `SimpleDomain` and `SimpleDefault` for Simple Feature Types depend on the `baseType` (and possibly the `unit`) of the Feature Type. For example, a Feature Type `Area` may specify a minimum area of 12.5 square meters, with a default value of 17. A `SimpleDomain` is either a set of character strings, a selection of boolean values, or a numeric domain, which is a set of numeric ranges. Some possibilities of numeric ranges are included in the diagram: a single numeric value; a discrete range, i.e. a succession of numeric values; and a continuous range with an upper and lower bound. Note that in the EXPRESS language, of which the graphical counterpart EXPRESS-G is used in the diagrams, a Number data type includes both the Integer and Real data types. This means that occurrences of the Number data type can be either integer or real values.

Enumeration Feature Types have an `EnumDomain` and `EnumDefault` that select from the enumerated identifiers that are defined within the Enumeration Feature Type.

For Complex Feature Types, the issue of domains and defaults is rather more complicated. A `ComplexDefault` for the Complex Feature Type as a whole specifies an instantiation of the Feature Type complete with instantiated components. This Complex Feature Instance must be included in the same Feature Type Library section as the Complex Feature Type. When, during a modelling session, such a Complex Feature Type is instantiated, the default action will be to create a reference to the Feature Instance included in the Feature Type Library. Similarly, a `ComplexDomain` specifies a set of Feature Instances that are included within the same library-section. Instantiating a Complex Feature Type with such a domain is restricted to selection of a reference from this set of Feature Instances in the Feature Type Library.

Notation of Domains

The syntax for domains varies per class of Feature Type. Five types of domains are defined: string-domains for Simple Feature Types with base type string, numeric-domains for Simple Feature Types with base type integer or real, a boolean-domain for base type boolean, enum-domains for Enumeration Feature Types, and complex-domains for Complex Feature Types.

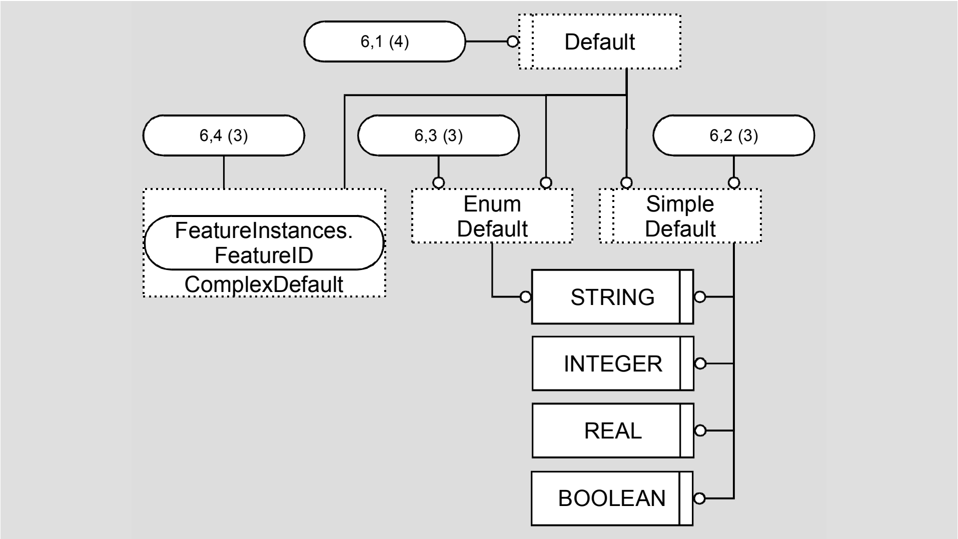


Figure 21 Diagram 6 of Schema FeatureTypes: Definition of Defaults.

Syntax for domains

```
      string-domain = string { ‘;’ string } .
     numeric-domain = numeric-domain-term { ‘;’ numeric-domain-term } .
numeric-domain-term = discrete-domain | continuous-domain | number .
    discrete-domain = number ‘,’ number ‘,..’ [ number ] .
  continuous-domain = ( ‘<’ | ‘[’ ) ( ‘<-’ | number ) ‘,’ ( ‘->’ |
                      number ) ( ‘>’ | ‘]’ ) .
     boolean-domain = boolean { ‘;’ boolean } .
        enum-domain = identifier { ‘;’ identifier } .
     complex-domain = featureID { ‘;’ featureID } .
```

The term `featureID` is defined in section 7. Some examples of these domains are given hereafter to clarify their meaning further.

Examples of the notation of Domains

Examples	Meaning
string: "living"; "cooking"	The string value may only be picked from the ones listed.
numeric: 2; 3; 5,10,..100 <-4,0]; [2,->>	Values may be 2 or 3 or in the range 5 to 100 in steps of 5, including 5 and 100. Values may be any real number between -4 and zero, but not including -4, or greater than or equal to 2.
boolean: true	The value will always be true.
enum: single; double	The enumerated value may be one of these identifiers.
complex: ht03; ht05; dstA3	Listed are the IDs of the Features that may be selected in the context of this domain.

6.4 Feature Type Libraries

The structure of Feature Type Libraries, as has been briefly indicated already in the beginning of this section, is a rather flat, hierarchical structure. Feature Types are categorised into sections in Feature Type Libraries. The assumption that such a flat hierarchy should suffice is based on the observation that most classification systems in use at present do not provide more levels. However, using this means of categorisation in practice will need to confirm this. If necessary, the structure of Feature Type Libraries must be amended to allow for nested sections of Feature Types. The initial structure of Feature Type Libraries is presented in figure 22, showing a decomposition of a Feature Type Library into sections which in turn contain the Feature Types.

At various occasions in the above sections, it has become clear that the inclusion of Feature Instances in Feature Type Libraries is a necessity. The reasons for this are summarised below.

defaults and domains for Complex Feature Types
The defaults and domains for Complex Feature Types are instances that need to be accessible from within the Feature Type Library and therefore are included in the library;
defaults and domains for components of Complex Feature Types
The same is true for the defaults and domains of the type-components of Complex Feature Types. If type-components refer to Complex Feature Types, these specify the initial and valid instances of those Complex Feature Types within the context of the Complex Feature Type that contains the references. Again the instances need to be accessible from within the Feature Type Library;
instance-components of Complex Feature Types
Besides type-components, Complex Feature Types can also contain instance-components, which are not

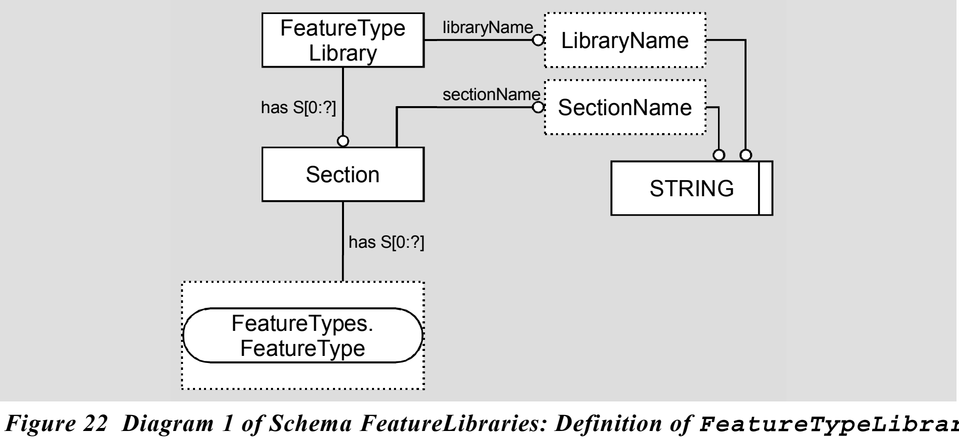


Figure 22 Diagram 1 of Schema FeatureLibraries: Definition of **`FeatureTypeLibrary`**
(This is a preliminary diagram, for the eventual diagram, see figure 23.)

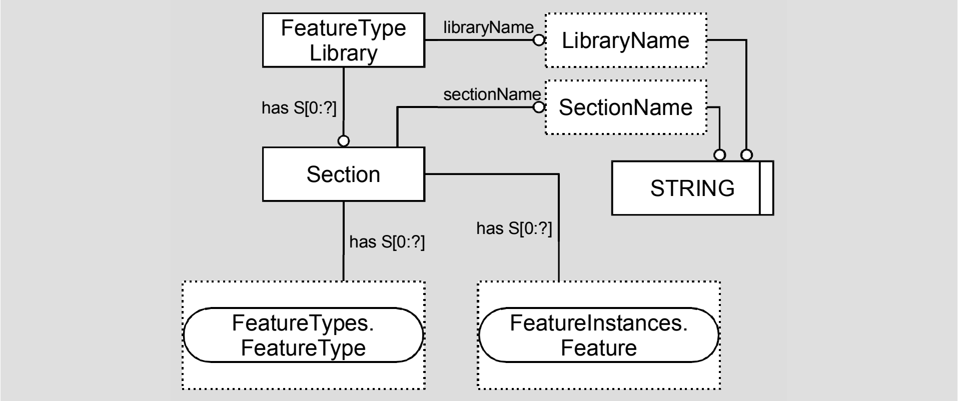


Figure 23 Diagram 1 of Schema FeatureLibraries: Definition of `FeatureTypeLibrary`
(This diagram supersedes the diagram in figure 22.)

references to Feature Types, but references to Feature Instances. These instances must also be available in the Feature Type Libraries.

Handler Feature Instances specifying the parameters for event handlers attached to any Feature Type
As discussed during the definition of Handler Feature Types, and referring to scenario 1 of how to use these types, a Handler Feature Instance can be used in the definition of any Feature Type to specify the parameters for handling events that occur in relation to the Feature Type. The Handler Feature Instance, for this purpose, must be available with the definition of the Feature Type, i.e. in the Feature Type Library.

Feature Instances passed as parameters at the type level
For the passing of constant values to components of Complex Feature Types at the type level, it is necessary that Feature Instances are available at the type level to provide these values.

This leads to a revised structure of Feature Type Libraries as displayed in figure 23. Besides Feature Types, a section within a Feature Type Library now also contains Feature Instances. The class `Feature` is defined in section 7.1.

7. Classes of Feature Instances

This section describes in detail the definition of the classes of Feature Instances, which determine the kind of information that can be modelled in terms of Feature Instances.

7

While Feature Types are the formal representation of domain knowledge, defining typological aspects of architectural design, Feature Instances are the formal representation of information concerning a particular design. What kind of information a Feature Instance contains and in what structure, is defined by a Feature Type of which the instance is instantiated. Analogous to the classes of Feature Types presented in section 6, this section presents the classes of Feature Instances that define the format for the Feature Instances that can be created within the framework.

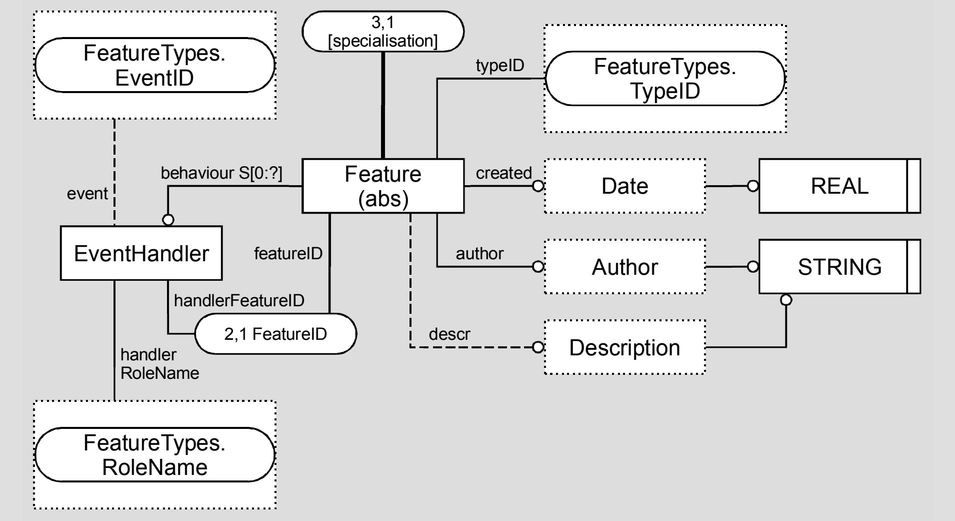


Figure 24 Diagram 1 of Schema FeatureInstances: Definition of the abstract base class `Feature`. (This is a preliminary diagram, for the eventual diagram, see figure 37 on page 42.)

7.1 The base class: `Feature`

The classes of Feature Instances are based on the abstract class `Feature`. The class defines that all Feature Instances contain the date and author of their instantiation, as well as an optional description. A graphical representation of the definition of the base class `Feature` is found in figure 24. As the name of this class indicates, the term Feature Instance is abbreviated to Feature, both terms are used interchangeably in the rest of this chapter.

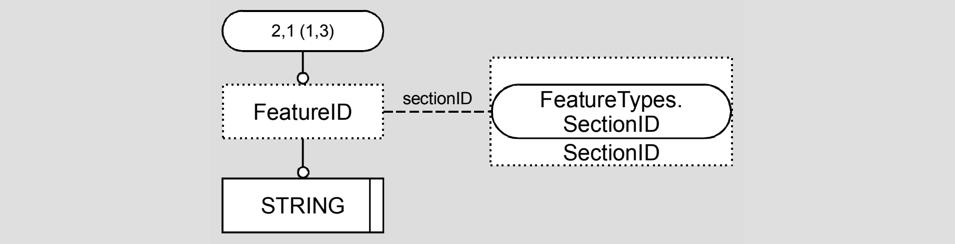


Figure 25 Diagram 2 of Schema FeatureInstances: Definition of `FeatureID`.

The Feature Type that a Feature is instantiated from is referred to in the Feature by its `typeID`. In the schema FeatureInstances this is a reference to the schema FeatureTypes. Features are identified by their `featureID`, represented in figure 25. `FeatureID`s are to be unique within their context, which is either a particular Feature model or a section in a Feature Type Library. A Feature Instance can be included in a Feature Type Library, as discussed in section 6.4, for reference in the definition of Feature Types or for reference from within Feature models. The `FeatureID` of a Feature Instance that is included in a Feature Type Library consists, besides a string, of the `sectionID` indicating the section in the library that contains it.

Any Feature Instance may define certain behaviour, which is modelled by means of the `behaviour` attribute (see figure 24). This attribute refers to a list of event handlers, which either have been defined at the type level of the Feature, or are added for a particular instance only. The event handler specifies the identifier of the Handler Feature that is to handle the event. In the case of an event handler defined at the type level, the event to be handled is specified at the type level. For handlers that are added at the instance level, the event must be indicated by the `event` attribute of the handler. Finally, each handler has a role name in the context of the Feature Instance.

Notation of the abstract base class `Feature`

The abstract base class `Feature` does not need a graphical or textual notation, because objects of this class cannot be created. Its attributes, however, are inherited by its subclasses and therefore need a textual notation. These attributes are, as shown in figure 24, `featureID`, `typeID`, `created`, `author`, `descr`, and `behaviour`. The attribute `typeID` specifies the type of which this Feature is an instance. This attribute is defined earlier on page 14.

Syntax for the attributes `featureID`, `created`, `author`, `descr`, and `behaviour` of the class `Feature`

```
        featureID = [ sectionID ‘::’ ] instanceName .
     instanceName = identifier .
          created = ‘Date’ ‘{’ date ‘}’ .
           author = ‘Author’ ‘{’ string ‘}’ .
            descr = ‘Descr’ ‘{’ string ‘}’ .
   inst-behaviour = ‘Behaviour’ ‘{’ { event-handler } ‘}’ .
    event-handler = featureID roleName ‘[’ eventID ‘]’ ‘;’ .
```

The attribute `featureID` consists of an identifier that is to be unique within its context, normally a Feature model. However, Feature Instances may also be included in Feature Type Libraries, in which case the ID also includes the `sectionID` (i.e. library name and section name) to indicate where they are located. The `sectionID` is followed by two colons and the name of the Feature Instance.

General syntax for the notation of Feature Instances

```
     feature-inst = simple-inst | enum-inst | complex-inst |
                    geometric-inst | constraint-inst | handler-inst .
standard-inst-body = created author descr
                    inst-behaviour inst-relations .
```

The attribute `behaviour` refers to the set of event handlers that are associated with the particular Feature Instance. An event handler is specified by the event identifier and by the identifier of the Handler Feature Instance that is associated with the event. See also the description of Handler Feature Instances in section 7.2 on page 39.

The `inst-relations` element of the standard body of an instance notation refers to an attribute of the class `Feature` that is not yet presented. It is used to describe relationships that are not defined at the type level, but that are added only for the particular Feature Instance. This aspect of instantiation is further discussed in section 8 on instance level relationships.

7.2 Subclasses of the class `Feature`

The classes of Feature Instances that can be instantiated are all subclasses of the abstract class `Feature`, inheriting all its characteristics. They are represented in the diagram in figure 26 and discussed separately in the remainder of this subsection.

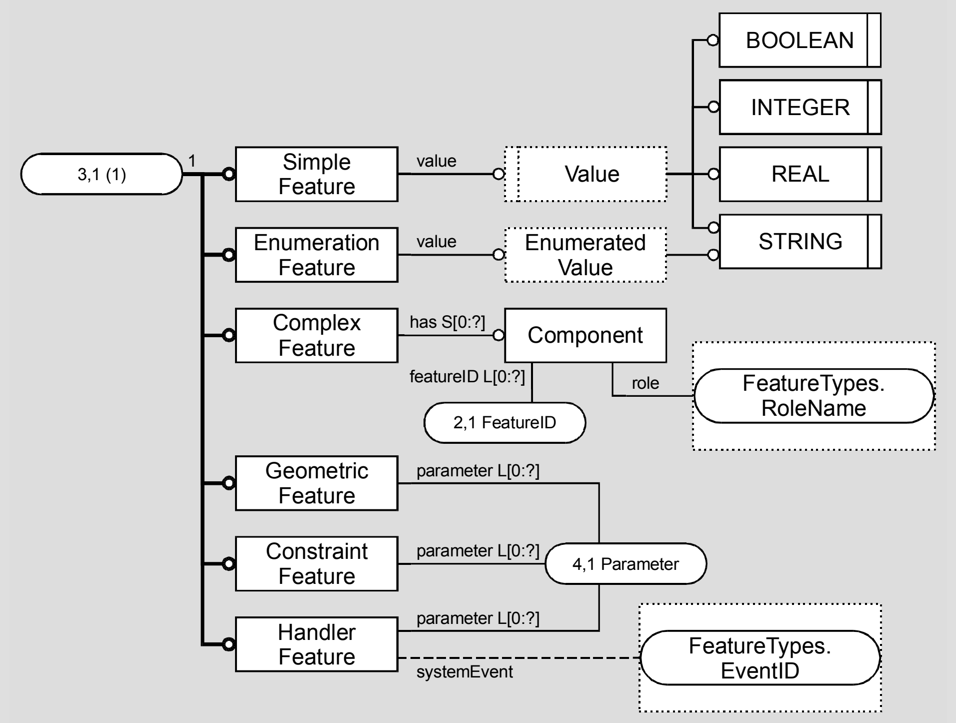


Figure 26 Diagram 3 of Schema FeatureInstances: Definition of the subclasses of ***`Feature`****.*

7.2.1 The class ***`SimpleFeature`***

The class `SimpleFeature` is the class of Feature Instances that are instantiated from Feature Types that have been defined using the class `SimpleFeatureType`. The instances of this class contain a single, simple value, such as integers, reals, and strings. What actual type of data it is that a particular Simple Feature Instance contains, is defined by the Simple Feature Type, which is referenced in the `typeID`[10] property defined in the super class `Feature` from which this class inherits (see figure 24). Also the unit of the data, e.g. `m2` or `W/m2`, is defined in the corresponding Simple Feature Type, as are the domain for the data and the default value (see the definition of the class `SimpleFeatureType` in figure 8 on page 15).

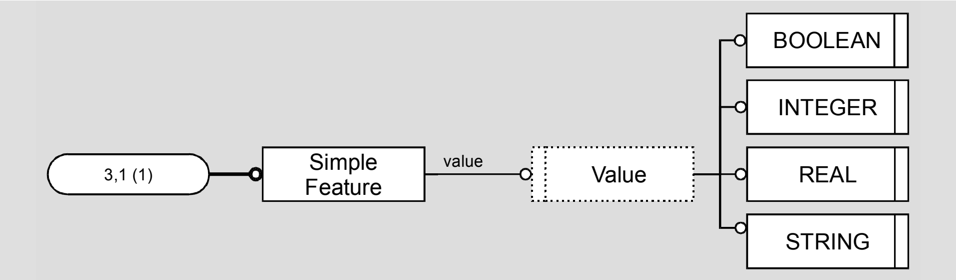
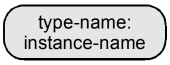


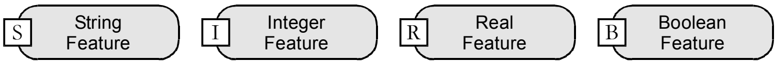
Figure 27 Excerpt from Diagram 3 of Schema FeatureInstances (see figure 26 on page 33): Definition of the class ***`SimpleFeature`****.*

[10] The fact that for Simple Feature Instances the `typeID` attribute should refer to the `typeID` of a Simple Feature Type, and not of any other Feature Type, is implicitly assumed here. Such restrictions would clearly have to be built in to an implementation of the framework, but including them in the schemata presented here would not particularly contribute to the clarity of the work. Similar assumptions are made throughout the rest of the discussion of the framework with respect to the kind of identifiers that are relevant and valid in particular contexts.

Notation of the class `SimpleFeature`



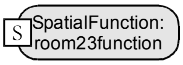
The graphical notation of Feature Instances is similar to the notation of Feature Types. Feature Instances are represented by rounded, grey boxes with the name of the Feature Type, followed by a colon and the name of the Feature Instance; `sectionIDs` are not graphically represented. For Simple Feature Instances, alternatively, the name of the Simple Feature Instance can be replaced by the value of the Simple Feature Instance. In the example below, the name of the instance `room23function` is replaced by its value `"Pantry"`. The graphical notation of Simple Feature Instances shows a letter S, I, R, or B in the square on the left of the rounded, grey box to indicate the base type of the instance.

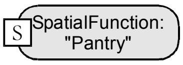


Syntax for the representation of Simple Feature Instances

```
     simple-inst = typeID featureID '=' '{' simple-inst-body '}' .
simple-inst-body = standard-inst-body 'Value' '{' value '}' .
           value = string | number | boolean .
```

Example of a Simple Feature Instance





```
Briefing.Plan.SpatialFunction room23function = {
   Date {19980704}
   Author {"Jos van Leeuwen"}
   Descr {"Function of room 23"}
   Value {"Pantry"}
}
```

What data type constitutes the value of a Simple Feature Instance depends on the `baseType` attribute of the defining Simple Feature Type. This cannot be read from the notation of the Simple Feature Instance.

For reasons of brevity and clarity, the `standard-inst-body` part of Feature Instances is omitted in the remaining examples in this and the following chapters. Also the `sectionID` (library name and section name) of `typeIDs` is omitted for the same reason.

7.2.2 The class `EnumerationFeature`

The class `EnumerationFeature` defines the objects that are instances of Enumeration Feature Types. The value of an Enumeration Feature is an identifier, a string, selected from the range defined by the enumeration of the particular Enumeration Feature Type (see figure 8 on page 15). Again, the domain and default for this value is defined in the corresponding Feature Type.

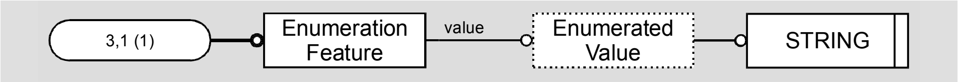
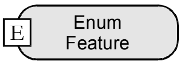


Figure 28 Excerpt from Diagram 3 of Schema FeatureInstances (see figure 26 on page 33): Definition of the class `EnumerationFeature`.

Notation of the class `EnumerationFeature`

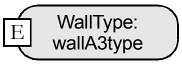
The graphical notation of Enumeration Feature Instances shows a letter E in the square on the left of the rounded, grey box.

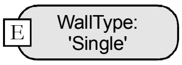


Syntax for the representation of Enumeration Feature Instances

```
     enum-inst = typeID featureID '=' '{' enum-inst-body '}' .
enum-inst-body = standard-inst-body
                 'EnumValue' '{' identifier '}' .
```

Example of an Enumeration Feature Instance





```
WallType WallA3type = {
   EnumValue {Single}
}
```

In the textual notation, the value of the Enumeration Feature Instance is noted by the identifier selected from the enumeration defined by the Enumeration Feature Type. As an alternative to showing the name of the enumeration instance, the identifier selected for the value of the instance can be shown in the graphical notation, yet enclosed in single quotes as shown in the example.

7.2.3 The class `ComplexFeature`

Objects of the class `ComplexFeature` are instances of Complex Feature Types. They form structures of components that refer to other Feature Instances. The components have a role within the context of the Complex Feature. A component may refer to more than one Feature Instance, in which case those instances share the same role in their relationship to the Complex Feature Instance. The relationship of a component to a Complex Feature Instance is defined at the conceptual level, i.e. in the definition of the corresponding Complex Feature Type. Therefore, the role name of the component corresponds to the role name that was specified in the Complex Feature Type and the number of instances that can share the same role is specified by the cardinality of the component in the type's definition (see figure 12 on page 19).

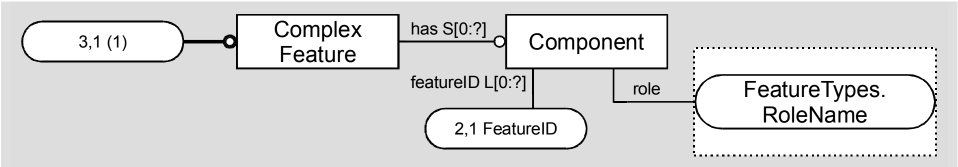


Figure 29 Excerpt from Diagram 3 of Schema FeatureInstances (see figure 26 on page 33): Definition of the class **`ComplexFeature`*.***

The component structures of Complex Feature Types are used to model three kinds of relationships: decompositions, associations, and specifications. Inheritance relationships, or specialisations, are modelled using the super-subtype structure of Complex Feature Types. The supertype of any Complex Feature Type is always again a Complex Feature Type. As a result, a Complex Feature Instance inherits the relationships defined by its type's supertype. Therefore, the component structure of a Complex Feature Instance also includes the components defined in the Complex Feature Types that are found above its type in the hierarchy of specialised Complex Feature Types, see figure 30.

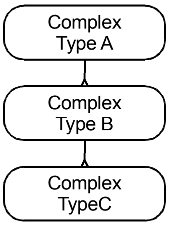
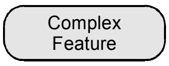


Figure 30 Specialisation hierarchy of Complex Feature Types. Analogous to other OO approaches, Type B inherits all the components defined by Type A, while Type C inherits all components defined by B and A.
Instances of Type C will contain components of all three types.

Notation of the class `ComplexFeature`

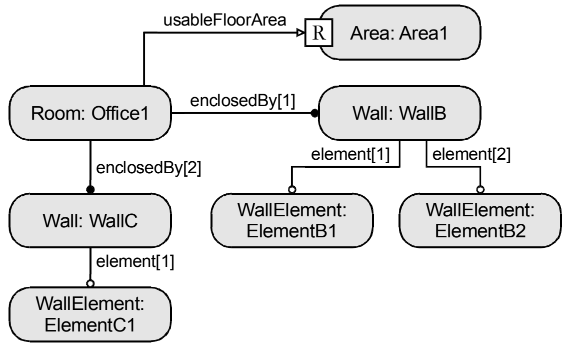
The graphical notation of Complex Feature Instances shows a rounded, grey box without additional symbols, containing the name of the Feature Type and the name of the instance. The relationships to its components, i.e. to other Feature Instances, are shown in both the textual and graphical notation using the role name and, if the cardinality is other than 1, the index of the related Feature Instance within the particular component. Components that are not instantiated are not shown in either notation.



Syntax for the representation of Complex Feature Instances

```
        complex-inst = typeID featureID '=' '{' complex-inst-body '}' .
   complex-inst-body = standard-inst-body { component-item } .
      component-item = roleName [ '[' integer ']' ] '=' featureID ';' .
```

Example of a Complex Feature Instance



```
Area Area1 = {
   Value {29.7}
}

Room Office1 = {
   enclosedBy[1] = WallB;
   enclosedBy[2] = WallC;
   usableFloorArea = Area1;
}

Wall WallB = {
   element[1] = ElementB1;
   element[2] = ElementB2;
}
```

NB. The textual notation of the above example is not shown completely.

The graphical representation of another example is shown in figure 31. This example demonstrates the effect of defining components of Features as references. Components of a Complex Feature Instance are not contained in the instance, but are relations, references, to other Feature Instances. This allows Feature Instances to share components. In the example, the 'function' component of both rooms 21 and 22 refer to the same Feature of the type `Function` that has the value 'Office'.

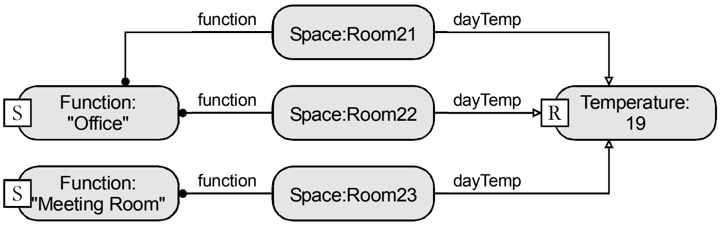


Figure 31 Shared Features made possible through the reference structure of the components of Complex Features.

The support class `Parameter`

The last three subclasses of Feature Instances use parameters to pass information from the Feature model to either parametric geometry, constraints, or event handlers. These subclasses are the classes of Geometric, Constraint, and Handler Feature Instances. They are defined next in this section. Parameters can be passed in four forms:

1. identifiers of Feature Instances;
2. components of Complex Feature Instances;
3. instance level related Feature Instances;
4. references to Feature Instances.

These four forms determine the definition of the support class `Parameter` which is shown in figure 32. Because parameters are passed in ordered lists, their position in the list must correspond to that of their declaration in the declared parameter list of the Feature Type (see figure 15 on page 22).

In the first form, where identifiers of Feature Instances are passed as parameters, the attribute `featureID` of a parameter simply contains the `FeatureID` of that Feature Instance. This provides access to the information available in the Feature Instance, such as the value of a Simple Feature Instance.

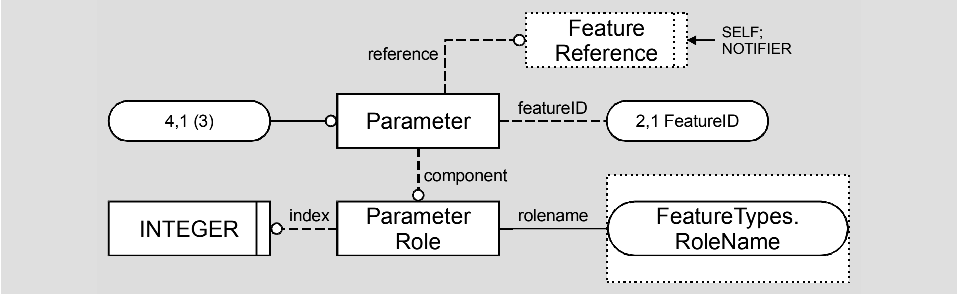


Figure 32 Diagram 4 of Schema FeatureInstances: Definition of the support class `Parameter`.

If the Feature Instance is a Complex Feature Instance, the `component` attribute of the parameter may refer to one of the components of that Complex Feature Instance by specifying the role name of the component; this is the second form of parameter passing. If the component refers to multiple Feature Instances, which depends on the cardinality of the component as defined in the Complex Feature Type, an index is required to indicate the exact Feature Instance that is to be passed as parameter. If that index is *not* specified, this means that, by way of the role name, *all* Feature Instances referred by the component are passed as the parameter.

Because relationships between Features can also be defined at the instance level, as discussed in section 8, in the third form of parameter passing, the role name refers to such an instance level relationship. Again this is done by specifying the role name of the relationship. Any Feature Instance, not just Complex Feature Instances, can have such instance level relationships.

The last form of parameter passing involves references to Feature Instances in the context of the parameters. For all three kinds of Feature Instances that use parameters, a parameter may refer to that Feature Instance itself by specifying the value `SELF` for the attribute `reference`. This attribute replaces the `featureID` attribute; they are mutually exclusive. For Handler Features, a parameter may refer to the notifier of the event handler by using the value `NOTIFIER` for the attribute `reference`.

The usage of the `reference` attribute is demonstrated after the definition of Handler Features. In the schema in figure 32, the attributes `featureID` and `reference` are shown as optional attributes, while in fact they are mutually exclusive.

Notation of parameter lists

Lists of parameters are textually noted as comma-delimited lists. Each parameter consists of the `featureID` of the Feature Instance that is to be passed as the parameter, or one of the keywords `SELF` and `NOTIFIER` as appropriate.

In case a parameter should refer to a component of a Complex Feature Instance or to an instance level relationship, the role name, preceded by a period, and the optional index in square brackets follow the identifier of the Feature Instance. This is done in both the textual and graphical notation.

Syntax for the notation of parameter lists

```
          param-list = parameter { ',' parameter } .
           parameter = ( featureID | 'SELF' | 'NOTIFIER' )
                       [ component-param ] .
     component-param = { '.' roleName [ '[' integer ']' ] } .
```

In graphical notations, a parameter is shown as an association relationship from the Feature Instance that uses it to the Feature Instance that is passed as parameter. The name of the parameter is shown next to the relationship line. If the parameter list is declared as a variable length list, then the parameters that have the same name are shown using an index in square brackets behind the parameter name, for example `oneOrMoreDoors[3]` indicates the third parameter to be passed with that name.

The references `SELF` and `NOTIFIER` are shown using Complex Feature symbols, regardless of the actual type of Feature. They are not resolved, i.e. not replaced by the referred Feature Instances. The symbol of the Feature Instance passed as parameter, if this is a Simple Feature Instance, may show the value of the simple data rather than its identifier.

Examples are given after the definition of the classes of Feature Instances that use parameters, next in this section.

7.2.4 The class `GeometricFeature`

Instances of a Geometric Feature Type merely consist of relationships to the Feature Instances that act as parameters for the parametric geometry defined in the Geometric Feature Type. The parameters form an ordered list that corresponds to the ordered list of parameter declarations in the Geometric Feature Type. In the framework, it is assumed that any explicit representation of the geometry is generated from the data available in the Feature model and has a repository that is, both transient and persistent if desired, external to the Feature model. The relationships between the external geometric model and the Geometric Feature Instances in the Feature model are to be maintained by the geometric model.

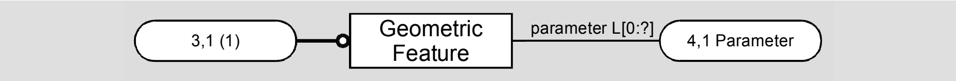
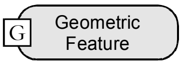


Figure 33 Excerpt from Diagram 3 of Schema FeatureInstances (see figure 26 on page 33): Definition of the class `GeometricFeature`.

Notation of the class `GeometricFeature`

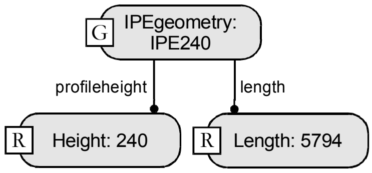
In the graphical notation, the relationships of the Geometric Feature Instance with its parameters are shown using the association relationship symbol accompanied by the name of the parameter. In the textual notation, the parameter list is enclosed in parentheses and follows the identifier of the Geometric Feature Instance. The keyword `SELF` may be used to make a parameter refer to the Geometric Feature itself.



Syntax for the representation of Geometric Feature Instances

```
     geometric-inst = typeID featureID ‘(’ [ param-list ] ‘)’ ‘=’
                      ‘{’ geometric-inst-body ‘}’ .
geometric-inst-body = standard-inst-body .
```

Example of a Geometric Feature Instance



used type definition:

```
geometry IPEgeometry (Height
   profileheight, Length length) {
   TypeGeometry {IPE}
}

IPEgeometry IPE240
   (Height240, BeamLengthBC) = {
}
```

7.2.5 The class `ConstraintFeature`

While Constraint Feature Types define constraints and their required types of parameters, Constraint Feature Instances only need to specify which Feature Instances actually provide the values for these parameters. The identifiers of these Feature Instances are inserted in the ordered list of parameters, which corresponds to the ordered list of parameter declarations defined by the Constraint Feature Type. As with the parameters of Geometric Feature Instances, the parameters may actually be referring to components of Complex Feature Instances, in which case the role name and possibly an index are required.

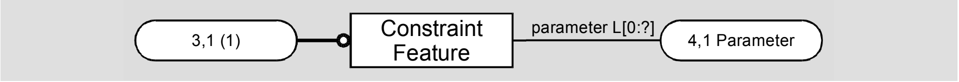
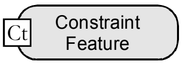


Figure 34 Excerpt from Diagram 3 of Schema FeatureInstances (see figure 26 on page 33): Definition of the class `ConstraintFeature`*.*

Notation of the class `ConstraintFeature`

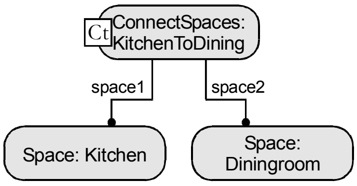
Similar to the Geometric Feature Instances, the relationships to the parameters of Constraint Feature Instances are shown with association relationship symbols in the graphical notation. The textual notation shows the parameter list in parentheses after the `featureID` of the Constraint Feature Instance. The keyword `SELF` may be used to make a parameter refer to the Constraint Feature itself.



Syntax for the representation of Constraint Feature Instances

```
     constraint-inst = typeID featureID ‘(’ [ param-list ] ‘)’ ‘=’
                       ‘{’ constraint-inst-body ‘}’ .
constraint-inst-body = standard-inst-body .
```

Example of a Constraint Feature Instance



used type definition:

```
constraint ConnectSpaces (Space space1,
   Space space2) {
   TypeConstraint {adjacentTo}
}

ConnectSpaces KitchenToDining
   (Kitchen, Diningroom) = {
}
```

7.2.6 The class `HandlerFeature`

A Handler Feature Instance provides the parameters that are to be passed to the procedure defined by the Handler Feature Type that it is an instance of. The parameters are provided in a list, similar to those of Constraint and Geometric Feature Instances.

Handler Features can be attached to a Feature Type or a Feature Instance, to handle so-called Feature-events. When such a Feature-event is triggered, the system is notified that the procedure defined by the Handler Feature Type should be called, passing it the parameters specified in the Handler Feature Instance. Handler Features can also be modelled to handle so-called System-events, in which case they are not attached to a notifying Feature Type or Feature Instance. The identifier of the System-event must be specified for this kind of Handler Feature Instance (see the definition in figure 19 on page 26).

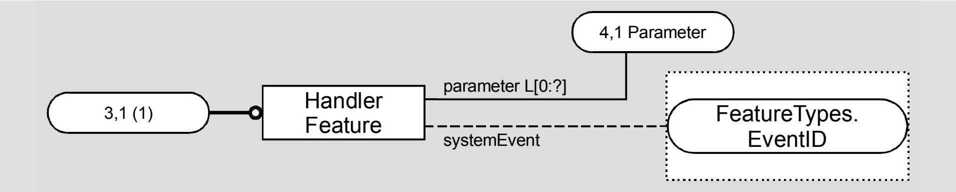
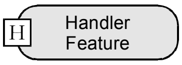


Figure 35 Excerpt from Diagram 3 of Schema FeatureInstances (see figure 26 on page 33): Definition of the class `HandlerFeature`.

Notation of the class `HandlerFeature`

In order to define the notation of Handler Feature Instances, the three scenarios described after the definition of the class `HandlerFeatureType` are followed (see page 26).

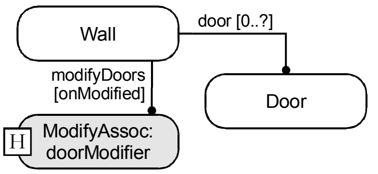


Syntax for the representation of Handler Feature Instances

```
       handler-inst = typeID featureID [ '[' eventID ']' ]
                      '(' [ param-list ] ')' '='
                      '{' handler-inst-body '}' .
  handler-inst-body = standard-inst-body .
```

In the first scenario, a Handler Feature is attached to a Feature Type, specifying the parameters for all event handling that should occur for the instances of that Feature Type. In this scenario, the handler should have access to the notifying Feature Instance and to its directly related Feature Instances. Because the Handler Feature is attached at the type level, the identifier of the notifying Feature is not yet known. Therefore the keyword `NOTIFIER` is introduced in the definition of parameter lists for handlers, which may substitute the `featureID` in both the graphical and textual notation of parameters. Note that the keyword `SELF` may be used to refer to the Handler Feature itself in order to access its instance level relationships. The `eventID` in this scenario is not relevant, because the event to be handled is specified by the notifying Feature Type.

Example of a Handler Feature Instance, attached to a Feature Type (scenario 1)

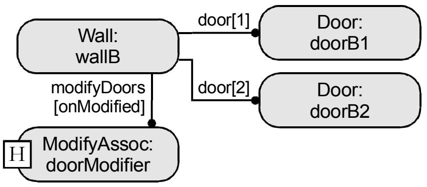


```
ModifyAssoc doorModifier
   (NOTIFIER.door) {
}

complex Wall {
   Behaviour { doorModifier
      modifyDoors [onModified];
   }
   Assoc Door door[0..?];
}
```

In the second scenario, a Handler Feature is attached to a Feature Instance, specifying the parameters for the event handling of that instance only. Although the `featureID` of the notifying Feature is now known, the same keyword `NOTIFIER` may be used to refer to that Feature.

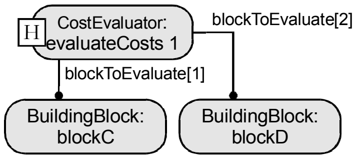
Example of a Handler Feature Instance, attached to a Feature Instance (scenario 2)



```
Wall wallB {
   door[1] = doorB1;
   door[2] = doorB2;
   Behaviour { doorModifier
      modifyDoors[onModified];
   }
}
```

The third scenario describes the handling of so-called System-events. This is done by Handler Features that are not attached to a Feature Instance or Feature Type.

Example of a Handler Feature Instance for handling of a System-event (scenario 3)



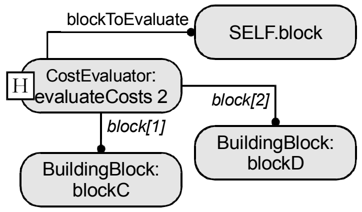
used type definition:

```
handler CostEvaluator
   (BuildingBlock blockToEvaluate, ...) {
   Procedure {
      // evaluate the costs related
      // to the buildingblock passed
      // as the 'blockToEvaluate'
      // parameter(s).
   }
}

CostEvaluator evaluateCosts_1
   [onStartEvaluate] (blockC, blockD) = {
}
```

In the above example, the parameter `blockToEvaluate` of the handler is specified by directly assigning the identifiers of the `blockC` and `blockD` instances. An alternative approach, which effectively does the same thing, is to use instance level relationships to relate both `BuildingBlock` instances to the Handler Feature. For the parameter, the role name of this relationship (`block`) can now be used, together with the keyword `SELF` to refer to the Handler Feature itself. This example illustrates the usage of instance level relationships, which are discussed in the section 8.

Example of a Handler Feature Instance for handling of a System-event (scenario 3)



```
CostEvaluator evaluateCosts_2
   [onStartEvaluate] (SELF.block) = {
   Assoc block[1] = blockC;
   Assoc block[2] = blockD;
}
```

7.3 Feature Models

Features are organised into Feature models. The organisation of a Feature model, as shown in figure 36, is defined as a simple collection of Features. Feature models are identified by the `ModelName` attribute. A consequence of this simple organisation of Features is that any subdivision of the model into, for instance, hierarchical levels of info, must be devised in another manner. The advantage of this approach, which is very much related to the objectives of Feature modelling that are being aimed at in this project, is of course that any subdivision can be made accordingly to the designer's intent.

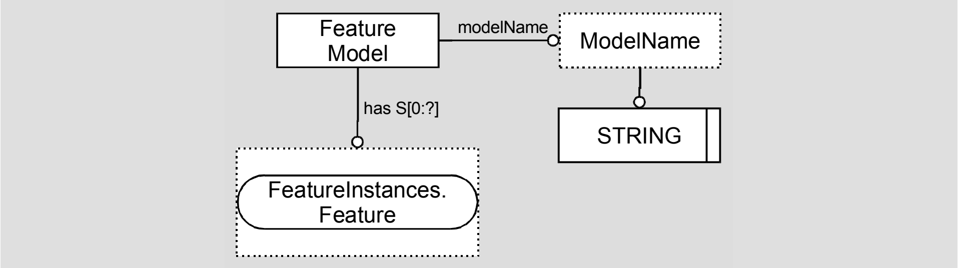


Figure 36 Diagram 1 of Schema FeatureModels: Definition of ***`FeatureModel`****.*

For such a subdivision, special Feature Types must be instantiated to model the backbone of the structure to which all Features in the model are to be attached. For example, if a subdivision of the model is requested into the levels of 'building', 'storeys', 'spaces', 'building elements', 'components', etc., Feature Instances with interrelationships will be modelled to represent these levels. Subsequently, all Features describing information at the level of storeys will be modelled with a relationship to the 'storeys' Feature. These relationships are preferably modelled at the instance level only in order to keep the Feature Types more generically applicable (see the next subsection). If the different storeys in a building are to be distinguished, multiple 'storey' instances must be modelled.

The Feature Types that are to be used to build up a hierarchy of levels in a Feature model can, as mentioned before, be defined to the needs of the designer. However, it is likely that the definition of Generic Feature Types, i.e. a standard set of hierarchical systems, would be beneficial, especially for the purpose of communication of models. The basis for the definition of these hierarchies could be found in those used in current modelling techniques, such as product modelling, and in more traditional approaches for documentation, such as specification writing and budgeting.

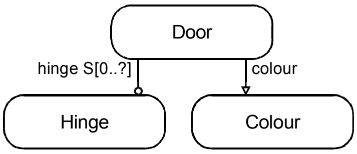
8. Instance level relationships

For the purpose of flexibility, instance level relationships allow a designer to model relationships between Features that are not defined at the typological level.	**8**

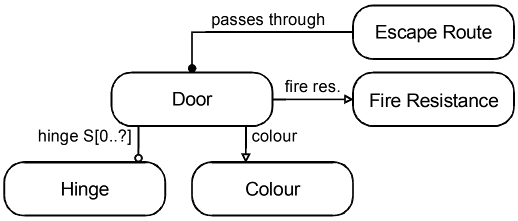
If a relationship is desired between Feature instances that is not defined at the typological level, a problem arises because the relationship cannot be instantiated. The following example will be used to discuss the options for solving this problem: A designer wants to model the relationship between doors and an escape route through the building. This implies that certain doors require adequate fire resistance. In the conceptual model used by this particular designer, it happens that fire resistance is not defined as part of the Feature Type `Door`.

Three different actions that can be taken to solve this problem are discussed below:

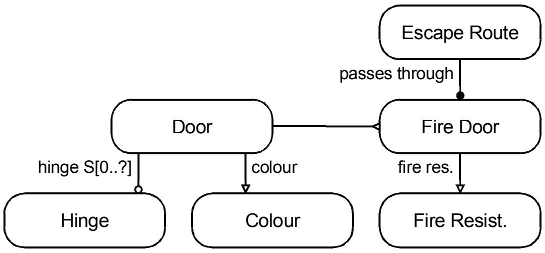
1. modifying the type definition;
2. creating a new type or sub-type;
3. modelling the relationship for the particular instance only.



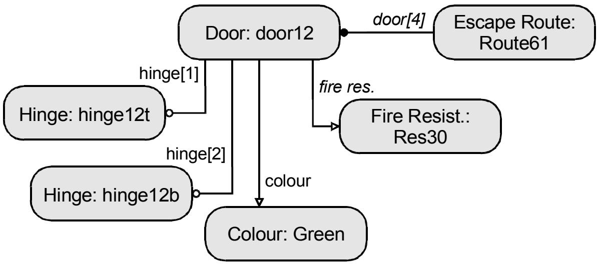
All three of these options need to be provided by design information systems based on the Feature modelling framework.



The first option is to change the definition at the typological level, adding the desired relationship to the Feature Type. This solution is acceptable if all instances of the particular type require this relationship. Making the relationship optional increases the chances that this is a satisfactory situation, however, in many cases the desired relationship bears no relevance to other instances of the same type: for many doors fire resistance may not be significant. Adding to the definition of a Feature Type, its size and complexity continue to increase and eventually it will loose its relevance, which is: being the common denominator of its instances.



A second option is to define a new Feature Type, either as an amended copy of the original Feature Type, or as a Feature sub-type that inherits the super-type's characteristics while adding the desired relationship: a `Fire_Door`-type is defined. This approach avoids the problem of growing Feature Types. However, the definition of a new Feature Type for each divergent instance may result in an unmanageable number of Feature Types (security doors, automatic doors, etc). Furthermore, an instance can only be derived from one type: these door-types are exclusive so that fire doors cannot be attached to the security system. Apart from this, it may not be the designer's notion of the situation that 'adding a relationship to a particular entity' is regarded as the creation of a new concept.



The actual situation in the third option may well be described as a designer adding a non-typical relationship to a Feature instance (the fire resistance concerns only *this* door), without regarding this as a change in concept of the original Feature Type. This situation should be modelled exactly in this manner, for the information model to represent most accurately the rationale of this design decision.

What is required thus, is a sort of incidental relationship that can be modelled between Feature instances without the necessity of including the relationship in the definition of the Feature Type. For this purpose we distinguish a *type-relationship* from an *instance-relationship*. Type-relationships are defined at typological level and have possible relevance to *all* instances of a type, whereas instance-relationships are added at instance level without being defined at typological level, thus without relevance to other instances of the same type. In the diagram, the roles of instance-relationships are shown in italics.

Three of the four categories of relationships described in section 4, are relevant also for the kind of relationship that can be expected at instance level. Table 3 lists the seven types of relationships that are part of the FBM infrastructure. The three types of instance-relationships are semantically equivalent to their counterparts at the typological level.

Table 3 Relationships at two levels.

	type-relationship	**instance-relationship**
specialisation	is_a	-
decomposition	has_a	instance_has_a
association	association	instance_association
specification	specification	instance_specification

One of the most important implications of instance-relationships for information modelling systems concerns the way information is searched for and addressed in the information model. Instead of using knowledge from the structure of the conceptual model (the typological level), the system must now search for relationships at instance level as well in order to find the requested information in a model. For instance, information concerning costs may be available in the model by means of relationships defined for the different types of elements in a building, but in addition, certain costs may be added, as a specification, to particular instances of building elements that bring additional costs to the construction process: ‘all steel columns of the type HE230A cost $x.xx per meter, but the one labelled D23 costs an additional $z.zz because it is harder to position.’

8.1 Implications for the framework

The purpose of instance level relationships is to allow unforeseen relationships or non-typical relationships to

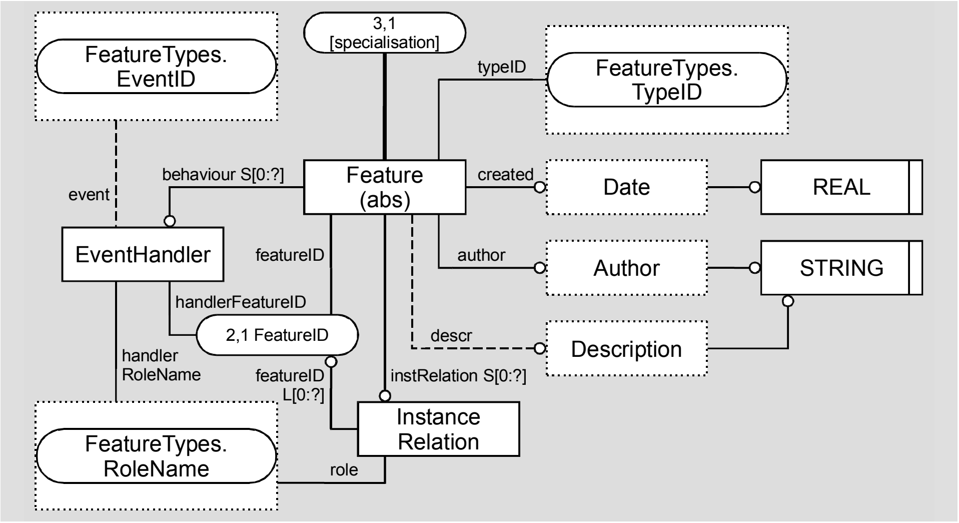


Figure 37 Diagram 1 of Schema FeatureInstances: Definition of the abstract base class `Feature`*. (This diagram supersedes the diagram in figure 24.)*

be added to Feature Instances. These relationships cannot be restricted by the definition of the Feature Type of the instance in question and must be allowed for all kinds of Feature Instances, not just Complex Feature Instances. This implies that instance level relationships need to be included in the definition of the base class `Feature`. The revised definition of the class `Feature` is represented in the diagram in figure 37. The revised `Feature` class defines the possibility of adding relationships between Feature Instances and giving these relationships a role. This structure is similar to the component structure defined in Complex Feature Types. The `InstanceRelation` entity in the diagram specifies the role of the instance level relationship, and has a list of Feature identifiers that refer to the related Feature Instances. This allows an instance level relationship to exist with multiple Feature Instances, using the same role name, similar to the way this is possible with components in a Complex Feature Type. Although the type of the instance level relationship is not pre-defined, the modelling system should ensure that the Feature Instances that participate in one such relationship should be of the same type.

Notation of instance level relationships

In the textual notation, instance level relationships are noted in a similar way as the definition of relationships at the type level. The same three keywords are used for the definition of the three kinds of relationships, `Spec`, `Has`, and `Assoc`. The relationship is further defined by the `featureID` of the related Feature Instance, the role name of the relationship, and an optional integer in square brackets, which indicates the index of the related Feature for the specified role name. Because there is no definition of this relationship at the conceptual level, there also is no definition of its cardinality, domain, or default. In fact, this means that the cardinality of instance level relationships is always `[0..?]`. If an instance level relationship needs to be added using the same role name as another instance level relationship, both will need to be given the additional integer to distinguish between them. In both the textual and graphical notation, instance level relationships are shown with the role name in italics.

Syntax for the attribute `instRelation` of the class `Feature`

```
     inst-relations = { inst-spec-decl | inst-decomp-decl |
                        inst-assoc-decl } .
     inst-spec-decl = ‘Spec’ inst-relation-decl .
   inst-decomp-decl = ‘Has’ inst-relation-decl .
    inst-assoc-decl = ‘Assoc’ inst-relation-decl .
 inst-relation-decl = roleName [ ‘[’ integer ‘]’ ] ‘=’ featureID ‘;’ .
```

References

Allen, J.F. 1983. Maintaining knowledge about temporal intervals. *Communications of the ACM* vol. 26.11: pp.832-843.

Dohmen, M. 1998. *Constraint-Based Feature Validation*. PhD. Thesis. Delft University of Technology.

Kelleners, R.H.M.C., R.C. Veltkamp, and E.H. Blake. 1997. Constraints on Objects: a Conceptual Model and an Implementation. In: *Proceedings of the 6th Eurographics Workshop on Programming Paradigms in Graphics*, Sep. 1997 (ed. Arbab and Slusallek), pp.67-78.

Kelleners, R.H.M.C. 1999. *Constraints in Object-Oriented Graphics*. Ph.D. Thesis. Eindhoven: Eindhoven University of Technology.

van Leeuwen, J.P., and H. Wagter. 1998. A Features Framework for Architectural Information, a case study. In: *Artificial Intelligence in Design '98* (ed. Gero and Sudweeks), pp.461-480. Dordrecht, NL: Kluwer.

van Leeuwen, J.P. 1999. *Modelling Architectural Design Information by Features*. PhD. Thesis 29 June 1999. Eindhoven, The Netherlands: Eindhoven University of Technology.

Martin, J., and J.J. Odell. 1995. *Object-oriented methods: a foundation*. Englewood Cliffs: Prentice Hall.

Index