DYNAMIC BUILDING MODEL INTEGRATION

by
Dewald Viljoen

Thesis presented in partial fulfilment of the requirements for the degree Master of Science in Civil Engineering at the University of Stellenbosch

Supervisor: Dr. G.C. van Rooyen
Faculty of Engineering
Department of Civil Engineering

March 2012

DECLARATION

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: 2012-01-09

Abstract

The amount and complexity of software applications for the building industry is increasing constantly. It has been a long term goal of the software industry to support integration of the various models and applications. This is a difficult task due to the complexity of the models and the diversity of the fields that they model. As a result, only large software houses have the ability to provide integrated solutions on the basis of a common information model. Such a model can more easily be established since the different software is developed within the same group. Other software suppliers usually have to revert to importing and exporting of data to establish some form of integration. Even large software houses still sometimes make use of this technique between their different packages. In order to obtain a fully integrated solution, clients have to acquire complex and expensive software, even if only a small percentage of the functionality of this software is actually required. A different approach to integration is proposed here, based on providing an integration framework that links different existing software models. The framework must be customisable for each individual's unique requirements as well as for the software already used by the individual. In order for the framework to be customisable, it must either encompass the information requirements of all existing software models from the outset, or be flexible and adaptable for each user. Developing an encompassing software model is difficult and expensive and thus the latter approach is followed here. The result is a model that is less general than BIM-style models, but more focussed and less complex. The elements of this flexible model do not have predetermined properties, but properties can instead be added and removed at runtime. Furthermore, derived properties are not stored as values, but rather as methods by which their values are obtained. These can also be added, removed and modified at runtime. These two concepts allow the structure and the functionality of the model to be changed at runtime. An added advantage is that a knowledgeable user can do this himself. Changes to the models can easily be incorporated in the integration framework, so their future development is not limited. This has the advantage that the information content of the various applications does not have to be pre-determined. It is acknowledged that a specific solution is required for each integration model; however the user still has full control to expand his model to the complexity of BIM-type models. Furthermore, if new software models are developed to incorporate the proposed structures, even more seamless and flexible integration will be possible. The proposed framework is demonstrated by linking a CAD application to a cost-estimation application for buildings. A prototype implementation demonstrates full integration by synchronising selection between the different applications.

Opsomming

Die hoeveelheid en kompleksiteit van sagteware programme vir die bou industrie is konstant aan die vermeerder. Dit was nog altyd 'n lang termyn doelwit van die sagteware industrie om integrasie van die verskeie modelle en programme te ondersteun. Hierdie is 'n moeilike taak as gevolg van die kompleksiteit van die modelle, en die diversiteit van die velde wat hierdie programme modelleer. Die gevolg is dat net groot sagteware huise die vermoë het om geïntegreerde oplossings te bied op die basis van 'n gemeenskaplike inligting model. So 'n tipe model kan makliker bymekaargestel word siende dat al die verskillende sagteware binne dieselfde groep ontwikkel word. Ander sagteware verskaffers moet gewoonlik gebruik maak van sogenaamde uitvoer/invoer tegnieke om 'n mate van integrasie te verkry. Selfs groot sagteware huise maak ook gebruik van hierdie tegnieke tussen hulle verskeie pakkette, in plaas van om die programme direk met mekaar te koppel. Om 'n vol geïntegreerde oplossing te verkry, moet kliënte komplekse en duur sagteware aanskaf, selfs al word net 'n klein gedeelte van die funksionaliteit van hierdie sagteware gebruik. 'n Verskillende benadering word hier gevolg, gebaseer op 'n integrasie raamwerk wat verskillende bestaande sagteware modelle met mekaar koppel. Die raamwerk moet aanpasbaar wees vir elke individu se unieke opset. Vir die raamwerk om aanpasbaar te wees, moet dit óf alle bou industrie inligting inkorporeer van die staanspoor af, óf dit moet buigbaar en aanpasbaar wees vir elke gebruiker. Om 'n model te ontwikkel wat alle bestaande inligting inkorporeer van die staanspoor af is moeilik en duur, dus word die tweede benadering gevolg. Die eindresultaat is 'n model wat minder omvattend is as BIM-tipe modelle, maar eerder gefokus en minder kompleks. Die elemente van hierdie buigbare model het nie voorafbepaalde eienskappe nie, eienskappe kan bygevoeg en weggevat word terwyl die program hardloop. Verder word afgeleide eienskappe nie gestoor as waardes nie, maar eerder as metodes wat gebruik word om hulle waardes mee af te lei. Hierdie konsepte laat toe dat die struktuur en funksionaliteit van die model verander kan word terwyl die program hardloop. 'n Verdere voordeel is dat 'n kundige verbruiker die veranderinge self kan doen. Veranderinge in die modelle kan maklik ingesluit word in die integrasie model, so toekomstige ontwikkeling word nie beperk nie. Dit beteken dat die inhoud van die modelle nie vooraf bepaal hoef te word nie. Al het die raamwerk 'n gespesialiseerde oplossing vir elke gebruiker tot gevolg, het die gebruiker nogtans volle beheer om sy model uit te brei tot die omvattendheid van BIM-tipe modelle. Indien nuwe sagteware modelle ontwikkel word met die integrasie raamwerk in gedagte, kan nog gladder en buigbare integrasie moontlik wees. In hierdie tesis word 'n tekenprogram met 'n kosteberaming program gekoppel om die voorgestelde raamwerk te demonstreer. 'n Prototipe implementering demonstreer volle integrasie deur seleksie binne die programme te sinchroniseer.

Acknowledgements

The findings documented in this thesis would not have been possible without the admirable help and guidance of Dr. G.C. van Rooyen.

Furthermore, Prof. B. Firmenich provided invaluable guidance and examples regarding the design of efficient software systems.

Countless internet bloggers freely providing practical advice and source code.

Table of Contents

List of Figures

Introduction

New software applications are constantly being developed for the building environment. The functionality of existing software is also constantly being expanded. The result is an ever increasing amount of applications, with ever increasing functionality. Along with the increased functionality comes increased complexity. In most cases these software tools operate within a specific environment, and each has its own information structure. Often some information has to be shared between the various applications. Usually the user has to either import/export between the environments or re-enter the information completely from scratch.

A typical example is when quantity surveyors use 3D CAD models, created by architects, to create Bills of Quantities. Sharing information by exporting and importing of data results in multiple files sharing similar information. If not done correctly, which is often the case, this result in loss of data integrity (Section 1.3).

Originally, different software houses developed software packages that suit the needs of the different professions. For example, one software house developed software for Quantity Surveyors; another developed CAD software for architects; another developed finite element method software for structural engineers, etc. In some cases integration was possible between the packages by importing and exporting of files. However, essentially all these packages functioned in separate environments.

In response, large software developing companies started to grow their software packages to address all possible needs of professions in the building environment and then attempted to integrate the different packages. The advantage being that software integration can more easily occur within one company than between different software developing companies. Integration techniques involved creating a common information model shared by all the different packages. However, an information model that addresses all software needs in the building environment is extremely complex. As an intermediate result, so-called integration still involved some kind of import/export technique. This does not represent seamless integration.

In recent years attempts have been made to create building information models that store information required by all software packages used in the building environment. The aim was to integrate different applications seamlessly on the basis of this common information model. After examining these common information models (refer Section 1.2) it is found that data integrity is often compromised (Section 1.3). Furthermore, developing applications based on such a complex model is an expensive process.

Smaller, more specialised applications often provide efficient solutions for certain problems. These applications are developed by small-scale developers who cannot afford to integrate their applications with the larger common information models. While complex common information models have its place in facilitating larger, main-stream building environment applications; it is also desirable for smaller, more specialised applications to operate within an integrated environment.

If a user desires an integrated software environment, different existing applications can also be linked with each. Applications can in this way be integrated according to the custom needs of a specific user. It is hypothesized that the combined information requirements of these different applications will be less complex than the all-encompassing common information models of larger main-stream applications. Furthermore, if the applications that the user already uses can be integrated with each other, there is no need for the user to convert to new software applications.

Each user will then require a custom integration solution. A framework is required that allows different applications to be linked with each other in such a way that information is efficiently managed. To enable customisability, the framework will have to be flexible. This framework will allow specialized software to run in an integrated environment without the need to be integrated into a large all-encompassing software package.

This thesis will demonstrate a prototype of a flexible modelling framework that allows integration of several existing applications. Since the purpose of this framework is to integrate existing software, each with its own user interface, the main focus will be on the internal architecture of the framework. Only a basic flexible user interface is developed to demonstrate the different functions of the framework.

Definitions

While the terms defined in this chapter can have different arbitrary definitions; in the context of this thesis, these terms should be interpreted as defined here. To prevent incorrect interpretation of some of the terms, the reader is referred to in-depth discussions of these terms in specified sections, rather than providing full discussions here.

- **Data Integrity:** The term database integrity is used in various different software disciplines, each with its own definition. In its broadest meaning it refers to the "… trustworthiness of system resources over their entire life cycle" (Wikipedia, 2011). In the context of this thesis, it will be used to describe data in a computer model that accurately reflects data

- **Seamless integration:** Refers to integration between programs in such a way that changes in one program are immediately reflected in all the other programs.

- **Building environment:** The Building Environment refers to all parties directly and indirectly associated with the construction industry. This includes, but is not limited to; contractors, civil and structural engineers, architects, project managers and quantity surveyors.

- **BIM/IFC:** Building Information Modelling, or BIM, refers to the concept whereby every aspect of a building through its entire life cycle is modelled; while Industry Foundation Classes, or IFC, is an open standard for BIM controlled by the international organisation buildingSMART.

- **Integration model:** see Section 1.3

- **Abstraction:** see Section 1.5

- **CAD:** Computer Aided Design (CAD) describes software applications used to create computer models consisting of the geometrical information of a building.

- **BoQ:** The Bill of Quantities (BoQ) to a document containing all pricing information for a building projects. All aspects of the project that contribute to its cost, e.g. tasks, materials, equipment hiring, etc., are each listed as an Item in the Bill of Quantities.

- **Derived attributes:** An attribute whose value is derived by an algorithm using the values of other attributes.

- **Functionalities:** In this dissertation, the word functionalities refer to specific uses of an application, an object or an entity. For example, the functionalities of a CAD-application will include amongst other functionalities: drawing a line, offsetting a line, creating a rectangle, inserting a dimension, defining a plot area, saving a drawing into a file, exporting a drawing to a supported format. A functionality of line object in a CAD application would be to calculate its length.
- **Core-model and supplementary models:** See Section 4.3.
- **GUI:** Graphical User Interface. The part of an application displayed on the computer screen, allowing the user to interact with the application.
- **MVC:** Model-View-Controller. See Section 1.6.
- **UML:** Unified Modelling Language. For a basic description of the UML concepts used in dissertation please refer the online article written by Donald Bell (Bell, 2004).

1 Overture

This chapter introduces key concepts required to understand this thesis. Existing software solutions are also discussed and a case study that will be referred to throughout the thesis. The prototype implementation for this thesis is based on this case study. The chapter ends off with the objectives of this thesis.

1.1 Case study

In order to better illustrate the concepts of the software framework being developed, a prototype integration framework was developed for a specific case study. The most important aspects of the framework will be explained in the light of this case study. This is done to assist the reader in grasping the key concepts. It is very important to note that the proposed framework is not intended as a solution to only this case study, but rather for all cases similar to it.

The case study is as follows:

An integrated solution has to be developed for a building company that specialise in small housing projects. The company works with 2-dimensional drawings (building plans), and wants to be able to integrate it with a program that assists in compiling and reconciling the Bill of Quantities of each project (call it the BoQ program). Users want to be able to select a component in the electronic drawings, and then the bill of quantity items relevant to this object must be selected and displayed in the BoQ program. After a project is completed, the pricing information for the project is stored in a database. The prices for the different items for the different completed projects can then be viewed by the project managers in order to better estimate prices for future projects.

The aforementioned problem requires firstly a highly customised solution, and secondly, integration on an intricate level. This will emphasize the two most important requirements of the proposed framework, namely flexibility and seamless integration.

The case study requires the following deliverables:

- Applications are required to manage electronic drawings and Bills of Quantities, which must be integrated with each other and linked to a database.
- The Bill of Quantity items has to be linked to electronic drawing components, and selection synchronised between the items and components.

1.2 Existing software solutions

For the purpose of emphasizing the stated problems, a discussion follows of existing software packages that could possibly address the requirements of the case study.

One of the largest software houses specializing in building environment software is run by Autodesk, Inc. Software from this software house includes (but is not limited to) applications such as

- Autodesk Revit Architecture: software designed for use by architects to create 3-dimensional building models.
- Autodesk Revit Structure: software quite similar to the suite above but for structural engineering purposes.
- Autodesk Quantity Takeoff: designed for quantity surveyors to compile Bills of Quantities from various different formats of construction drawings.
- Autodesk Navisworks: designed to integrate different software to assist in controlling building projects.
- Autodesk Vault: used for managing data produced by Autodesk's BIM applications.

Only the applications most relevant to the case study are listed above.

To ensure the different applications are relevant to all individuals involved in the construction process, each application contains an extensive amount of functions. An individual might require only a small percentage of these functions.

In the case study example, only a limited amount of functions from these applications will be used as demonstrated now. Firstly, the company uses 2-dimensional drawings, whereas these suites work mostly with 3-dimensional models. The Quantity Takeoff program allows Quantity Surveyors to compile Bill of Quanitities from various sources – 3-dimensional BIM models, 2-dimensional drawings in different formats, photos, etc. The company in the case study only works with 2-dimensional drawings in a specific format. This application contains a host of functionalities useful only to Quantity Surveyors and not this company. While the company only requires 2-dimensional drawings (and probably only has the skills to use these), it has to choose between two very complex 3-dimensional modelling tools. The same is relevant to the Naviswork and Vault applications. Furthermore, information exchange between the different applications can mostly only be established by exporting data from the one application and then importing it to the other.

Bentley Systems Inc. is another large software house with quite similar applications. The same complexity is experienced here.

Of course the case study company can use smaller more specialized applications such as:

- Several light-weight and some even open source 2-dimensional drawing programs such as AutoCAD Lite (also by AutoDESK), software by Caddie Software, Microstation (by Bentley Systems, Inc.).
- WinQS: a highly specialized Quantity Surveying application designed specifically for Quantity Surveying within the Southern African environment.
- Cademia: an open-source freeware 2D CAD application that is very basic, but extremely light weight. Several other open-source CAD programs exist, however Cademia has been used as an example implementation in this dissertation since it is developed in Java.
- Several existing programs can be used to create and manage a database. Since Microsoft Access is normally readily installed as part of the Microsoft Office suite, it was used to create the database for the prototype.

The problem occurs when integration is desired. It would be a far reaching goal to find a combination of existing software for the case study example that can export and import between each other; not to mention seamless integration.

Thus the company would be forced to convert to the Autodesk products or something similar. These applications are all complex, requiring extensive training and expensive hardware resources, as well as software licensing expenses.

1.3 Ensuring data integrity

One of the most important rules of maintaining data integrity is: store data only once. If the same data is stored in more than one location, data from one location can become inconsistent with data from the other locations. Once an inconsistency can occur, data integrity is compromised.

This rule implies that if two different files share certain information, data is being stored in more than one location and therefore data integrity can be compromised.

In mathematical terms, let the set A be the set of applications for the building environment.

$$A \coloneqq \{a \mid a\ is\ an\ application\ for\ the\ building\ environment\}$$

In the case study example, A might consist of two applications: a CAD application and a Bill of Quantities application.

Let the object set $M(a_i)$ contain the objects (information) stored be the application a_i. Each of these sets is analogous to the underlying information model of the different applications.

$$M(a_i) \coloneqq \{m(a_i) \mid m(a_i)\ is\ an\ object\ stored\ by\ application\ a_i\}$$

The set C contains the conglomerate of all the information of the building environment, and will be the union of all the sets $M(a_i)$:

$$C = \bigcup_{A} M(a_i)$$

Let the set B contain all the applications utilised by a specific end-user. This set will be a subset of A.

$$B \subseteq A$$

Where:

$$B \coloneqq \{b \mid b\ is\ an\ application\ utilised\ by\ the\ end - user\}$$

Some information is shared between the various applications and is collected in set I as follows:

$$I = \bigcap_{B} M(a_i)$$

If information shared by one or more applications is stored separately for each program, the result is that certain information will be stored in more than one place. This means data integrity is being compromised.

One solution is to store all information for all applications in the same standard file format. This file format will have to cater for all information in C. The amount of information contained in C is massive and therefore extremely complex. This set is also continuously growing as new software is developed constantly. Therefore, this file format would have to be changed frequently to accommodate new or improved applications. Changing such a complex format is a lengthy and expensive process that will have to be managed by a centralised committee.

Another approach involves developing an integration model analogous to the set I. This model serves to collect all information that is shared by the applications being utilised by the user, ensuring that data integrity is always maintained. A model that readily integrates all existing software applications would be as complex as the IFC model. However, if this integration model is flexible, it can be adapted and customised for every individual's requirement, while keeping it as simple as possible. With the case study, this integration model has to collect the geometric data (refer to Figure 1).

Unfortunately it is not always possible for all data to be stored in one place only. Take the above example, if all common data were to be stored in the integration model, the internal structure of the CAD program would have to be changed to retrieve and store geometric data from the integration model instead of its own CAD model. It is desirable to use the existing CAD program without having to change its internal structure.

If the same data is stored in more than one location, a controller must ensure that the data in the different locations is always synchronised. In this case the Integration model serves a different purpose – to control the shared information ensuring that both the CAD-model and the BOQ-data are in-sync with each other.

Another way in which data integrity can be compromised involves the storage of derived attributes. Derived attributes are attributes that are calculated from, or influenced by, other attributes. In other words, a derived attribute depends on other attributes. If a derived attribute is stored together with the attributes it depends on, an inconsistency can occur whenever one of these attributes are changed without the derived attribute being updated.

Take the example of storing information for a Wall element. One could store attributes as follows:

- length
- width
- height
- surface area

In the simplest form, the surface area is calculated as the product of the length and the height, thus it is a derived attribute. When the length or height is changed without updating the surface area, an inconsistency occurs. Ultimately, the method by which the surface area is achieved should be stored, instead of the value of the derived attribute.

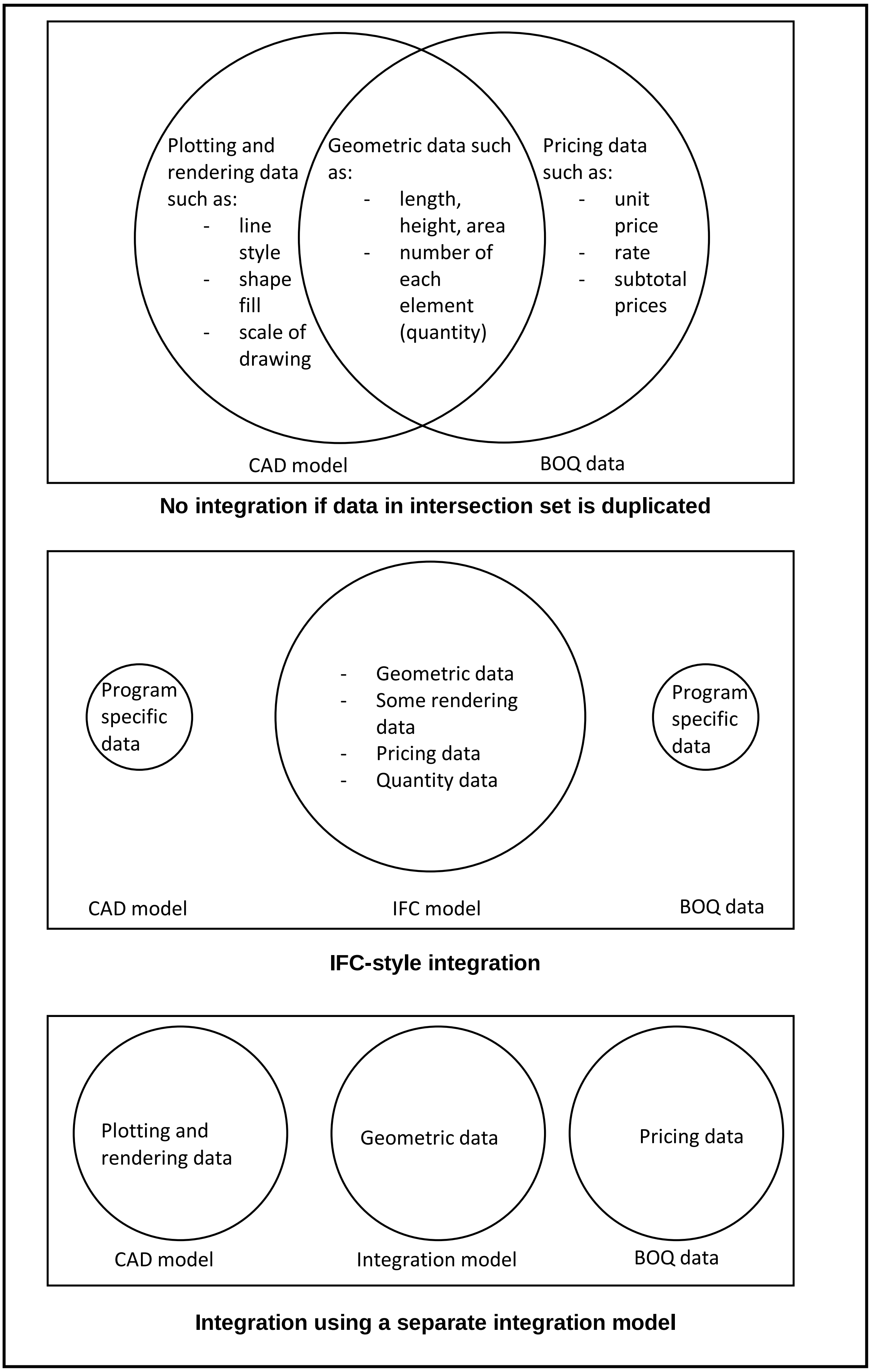


Figure 1 Storage of building information

A system that can guarantee 100% data integrity at all times is not always possible, but this situation should be strived towards when designing a new system. The two concepts described above can be used as benchmarks, and whenever it cannot be reached, a check should be built into the system to ensure data integrity.

1.4 Flexible and customisable as a solution to complexity

The building environment includes a great range of software applications to cater for every discipline involved. Most of these applications will share some information and functionality with each other (refer Figure 1). As before, let the set A be the set of applications for the building environment.

$$A \coloneqq \{a \mid a\ is\ an\ application\ for\ the\ building\ environment\}$$

The set $F(a_i)$ is the set of functionalities that the application a_i provides.

$$F(a_i) \coloneqq \{f(a_i) \mid f(a_i)\ is\ a\ functionality\ provided\ by\ application\ a_i\}$$

The set U is the set of functionalities required by a specific user. $U(a_i)$ is the set of functionalities that a specific user requires from application a_i, where $U(a_i)$ is a subset of $F(a_i)$. The set $U(a_i)$ can be an empty set.

$$U \coloneqq \{u \mid u\ is\ a\ functionality\ required\ by\ a\ specific\ user\}$$

$$U(a_i) \coloneqq \{u(a_i) \mid u(a_i)\ is\ a\ functionality\ required\ by\ a\ user\ provided\ by\ application\ a_i\}$$

Where:

$$U(a_i) \subseteq F(a_i)$$

With:

$$U = \bigcup_{A} U(a_i)$$

The set B contains all the applications which the user utilises. The applications of this set are such that the corresponding set $U(a_i)$ is not empty.

$$B \subseteq A$$

Where:

$$B \coloneqq \{b \mid b\ is\ an\ application\ utilised\ by\ the\ end-user\}$$

The set $T(b_i)$ consists of all the functionalities provided by the software utilised by the end-user. The set U (functionalities required by the user) is a subset of the set $T(b_i)$.

$$T(b_i) = \bigcup_{B} F(b_i)$$

$$U \subseteq T(b_i)$$

What this boils down to is that the total amount of functionalities that user will be faced with, set $T(b_i)$; will contain functionalities that is not necessarily required by him.

In most circumstances the complexity of applications increase with the amount of functionalities provided by them. Functionalities require information that is stored in an information model, in this case the underlying building model of each application. The amount of information of these underlying models will therefore increase with the amount of functionalities that require information from it. Complexity will increase with the amount of information stored in the model. An engineering model that stores information for all the functionalities in set $F(a_i)$ can be compared to the IFC-style building models. These models will be the largest (and therefore most complex) models since it stores information for all possible functionalities required by the building environment.

In theory, the complexity of a building model can therefore be lowered by decreasing the amount of functionalities it has to cater for. One way of lowering the amount of functionalities is by only integrating the functionalities of set U. However, it is not always possible to separate the functionalities of each application and only utilise the required ones. For this reason, integration will occur for the functionalities in set $T(b_i)$. The process of selecting only the functionalities required by a specific end-user is a customisation process.

The amount of functionalities in $T(b_i)$ can be further reduced by ensuring the complement of $U \cap T(b_i)$, in other words the functionalities not being utilised by the end-user, is as low as possible. More specialised applications will contain less functionalities that is not utilised by the user. Applications that are customised for the user's needs will contain even less unutilised functionalities. For this reason it is important to use smaller more specialised and customised applications.

Currently, mainstream IFC-style models are moving in a direction where the information for all applications is stored in one general model. The result is a large and complex model with each user only utilising a small portion of the model. Only large software houses have sufficient resources to create such a large model. Furthermore, to aid in usability, the model is split between applications for the most common portfolios, e.g. the architecture application is kept separate from the structural design application, which is separate from the quantity take-off application. In this way integration is possible within each discipline – one application is utilised for all tasks within each discipline. However, information exchange between the applications still occurs by exporting and importing files. The result is that the models for the different applications are saved separately, resulting in duplication of information and loss of data integrity. Furthermore, seamless integration is not possible across the different applications.

In some cases, especially in smaller companies, individuals have to perform tasks from a wider portfolio. They will require basic functionality from more than one of these IFC-style applications. In other words, they will require a bigger set of applications to work from (B), and fewer functionalities from each application ($U(a_i)$). If they were to use these large IFC-style applications, they will be faced with a large amount of functionalities that they do not utilise. Furthermore, the final set of applications will not be seamlessly integrated.

For these individuals, the mainstream IFC-style models do not provide an integrated solution. A better approach would be to select smaller, more specialised applications with only the functionalities required by the individual and creating an integration model customised for each individual. For an integration model to be customisable, it will have to be flexible.

1.5 Abstraction to achieve flexibility

According to (Google Inc., n.d.), abstraction is "[The] process of considering something independently of its associations, attributes, or concrete accompaniments". In more practical terms, the abstraction process commonly involves modifying objects to be applicable to more situations similar to the one it was originally intended for. This is achieved by reducing the information defined by an object to only the information of specific relevance to that object.

An example within the context of the case study would be as follows. It could be argued that a typical building model consists of walls, windows and foundations (to name only a few). All of these entities have a name and location, aside from other properties specifically relevant to each entity. The name and location are properties that these entities have in common.

It can also be argued that walls, windows and foundations are building elements, and all building elements by default have a name and location. Since walls, windows and foundations are building elements, they inherit the name and location properties. Now when defining the properties of a wall, window or foundation; it is not necessary to specify that it has a name and location. Only the information specifically relevant to each of these entities has to be defined. For example, a wall can be defined as consisting of a length, height and width; but since it is a building element, it will automatically also have a location and name. The result is that the amount of information in the definition of a wall object is reduced. A building element is an abstract object and cannot exist on its own – a non-abstract object might be a wall, which is a building element.

After this abstraction process, it can be argued that a building model consists of building elements, which can be walls, windows or foundations. If a programmer or user wants to define a new type of building element, say a roof; the programmer does not have to define the name and location properties. A roof is a building element, which implies that it has these properties. The programmer can therefore focus on the information that differentiates a roof from a wall, window or foundation. This makes it easier to define new building elements, and thus makes the integration structure more flexible.

The "is a" relationship between walls, windows and foundations, and building elements; is analogous to generalisation/specialisation in programming terms. The wall, window and foundation objects are instances of the classes *Wall*, *Window* and *Foundation*. The structure of the *BuildingElement* object can be defined in an abstract or non-abstract class, in which case (sub)classes *Wall*, *Window* and *Foundation* will have to extend this (super)class. A *BuildingElement* can also be defined in an interface, in which case the classes *Wall*, *Window* and *Foundation* will have to implement this interface. An interface can only define certain methods that these classes (or their subclasses) must implement; it cannot define attributes and implemented methods. The structure developed by this thesis follows Java-style single-inheritance, where a class can only extend one other class. Classes can, however, implement more than one interface.

The following UML-diagram demonstrates the specialisation process for the wall, window and foundation entities described above.

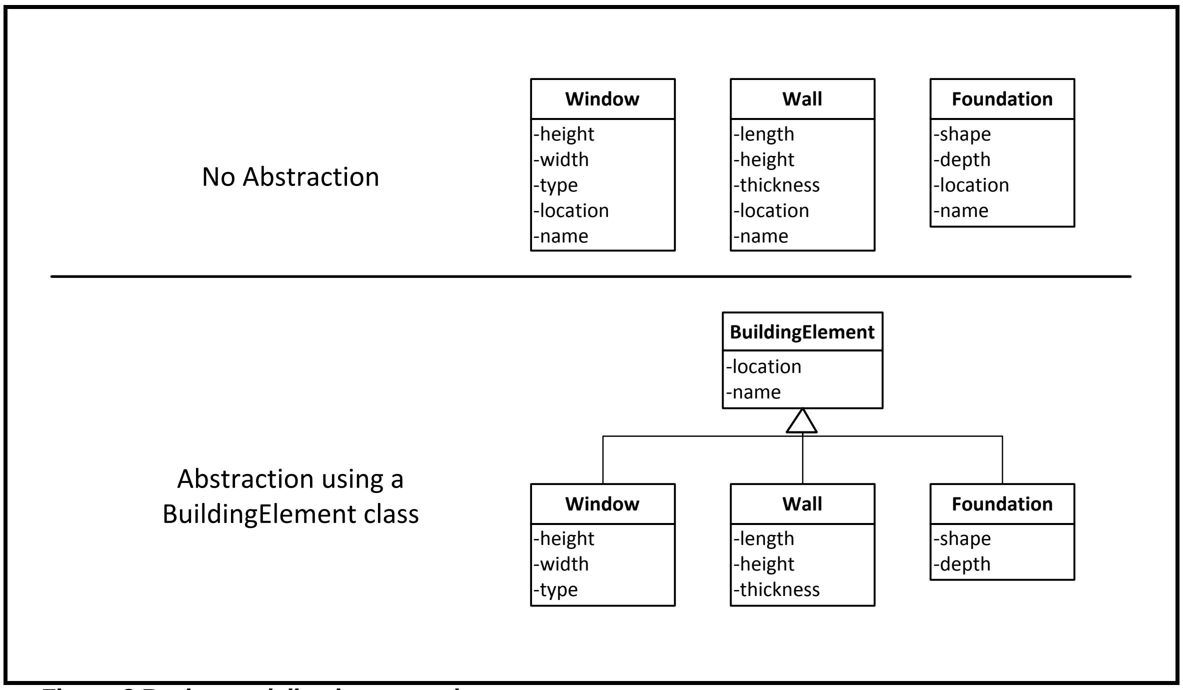


Figure 2 Basic specialisation example

This example, and further examples that demonstrate how abstraction can increase flexibility, are discussed in more detail in Sections 2.2 and 2.4.

1.6 Model-View-Controller architecture with the emphasis on Model

Model-View-Controller (MVC) is a term used for a software architecture utilised by most GUI based software applications. It is a proven architecture and holds several advantages for the framework developed in this thesis.

MVC architecture separates the software into three separate parts, the Model, View and Controller. According to Eckstein (Eckstein, 2007), the Model "represents data and the rules that govern access to and updates of this data. In enterprise software, a model often serves as a software approximation of a real-world process." In short the Model consists of the part of the application that changes for every new problem. This is the part of the application that will be saved in a file. In the building environment, the objects in the Model will represent entities of a real world building project. It is always preferable for files to be backwards compatible – files saved using older versions of an application must be compatible with newer versions. Therefore, while changes can easily be made to the other parts of an application, the structure of the Model must preferably change as little as possible. The view consists mostly of the GUI and provides a view on the information stored in the model. The user interacts with the Controller, which then modify the Model based on the input from the user.

The following diagram shows how the different parts interact with each other.

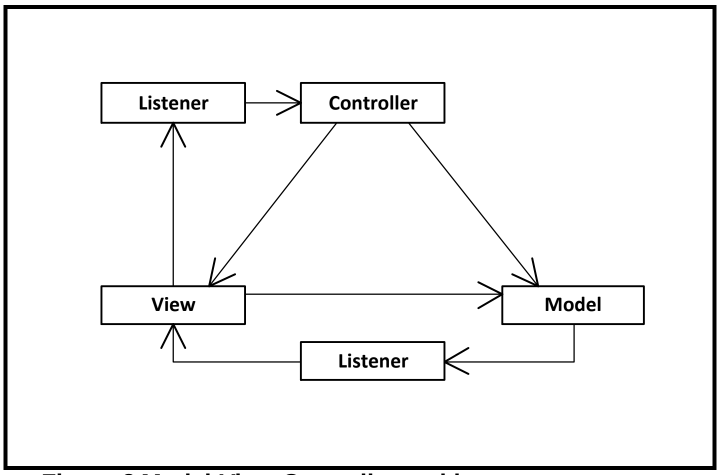


Figure 3 Model-View-Controller architecture

The Model and View are directly associated with the Controller, this means that the Controller can directly invoke methods on the View and Model. However, methods are invoked in the Controller using a Listener pattern; usually as a result of user input. The Controller then chooses how to react to this input. The Model is also directly associated with the View and it can therefore directly acquire information from the Model; however methods are invoked in the View also by means of a Listener pattern.

The Model and Controller are the most important parts of the application and form the core of the application. It is hard to make changes to these parts. The View is the first part to be adapted for customisation purposes since it is the part of the program that the user has direct interaction with.

As mentioned in Section 1.4, a customised application can help in reducing the complexity of an application. Creating a customised application involves consultation with the end-user and then creating a GUI with only the functionalities required by the user.

If customisation requires changes to the core of either of the applications, a new version of the application is created in the process. It is not hard to imagine how different versions of a program can cause confusion. Furthermore, interoperability between different versions of a program can lead to unexpected problems. Customisation should therefore be managed in such a way that it adds on, or plugs into, a stable core framework.

The process of customisation is mainly a commercial exercise and if the Model and Controller is flexible enough to support such customisation, this can be an inexpensive process. Of academic value is creating a Model, Controller and basic View that is flexible enough to easily support customisation.

1.7 Thesis objective

Wikipedia (Wikipedia, 2011) defines a software framework "*… is an abstraction in which software providing generic functionality can be selectively changed by user code, thus providing application specific software.*"

In the context of this definition, the focus of this dissertation can be described as follows: Designing a software framework that provides a basis for integrating applications in the Building Environment. This basis must consist of a flexible Model, Controller and basic View; and support efficient customisation for the integration of specific applications.

Another point worth mentioning is that users of software in the Building Environment are commonly engineers and technical personnel with some programming knowledge. For these individuals it is not cost effective to program applications from the ground up. However, their basic programming skills may be sufficient to further modify the software for individual needs. For this reason, certain parts of the framework can be exposed to the end-user to support this additional modification. This would significantly increase the flexibility of the framework. The exposed parts are simplified through abstraction techniques in order for it to be easily understandable.

The focus of the framework is not on creating a GUI and therefore only a basic GUI will be created to demonstrate the flexibility of the underlying core. Furthermore, a complex application can be perceived as simple by the user as a result of a good user interface. A complex underlying core can in this case hamper customisation and cause unexpected problems. An emphasis is therefore placed on ensuring the simplest possible underlying core that inherently ensures data integrity at all times.

The goals of the framework can be summarised as follows:

- Models based on this framework must be less complex than BIM-style models.
- It must be flexible enough to allow efficient customisation.
- Customisation should not necessitate changes to the core of the framework.
- It must allow end-users with basic programming skills to customise models based on this framework.

The Java programming language is designed to be used across all computer platforms. This would simplify cross-platform integration. After more than 15 years in use it may be considered a mature programming language. It is used by a wide variety of software developers and vendors, which means it is well supported. A wide variety of existing open source code and examples allows applications to be developed at a rapid pace. For these reasons the framework prototype is developed in the Java programming language, although the concepts could easily be transferred to other object orientated languages as well. Unified Modelling Language (UML) diagrams are used, allowing the concepts to be understandable to a wide audience.

As mentioned, a prototype implementation was developed to demonstrate the key concepts of the framework. In order to keep the prototype as basic and understandable as possible, only functionalities that demonstrate key concepts were implemented. The source code was created with the idea in mind that the reader should be able to easily understand it upon examination. Therefore, and in order not to divert attention from the key concepts of this thesis; only brief descriptions are given at the end of most chapters to introduce the reader to the source code. The reader is advised to examine the accompanying source code if further clarification is required.

2 Creating a flexible Model

The most important part in a Model-View-Controller architecture is the Model since the View and Controller follows from that. As mentioned in Section 1.5, the goal is to create a flexible and abstract Model for the integration framework. It is hard to achieve high levels of abstraction before first understanding the main problems that the Model must address. Therefore, a basic model is created first of all, which is then abstracted to achieve desired levels of flexibility. The chapter ends of with a brief introduction to certain parts of the source code.

2.1 Basic model

The more closely the objects of a software model resemble entities from the real world, the more understandable the model becomes.

Starting off then with the real world situation, the company from the case study will be working with small housing building projects. Essentially each project will consist of a house, which can be broken up into walls, windows, foundations, etc. For the sake of simplicity the structure of the model will be demonstrated using only these three entities and new types of entities can be defined at a later stage with no loss of generality. More importantly, the additional entities can be defined when the application is customised.

Based on the assumptions above, the most basic software Model can consist of a *Building* object, which comprises of *Wall*, *Window* and *Foundation* objects. The *Building* object can contain information such as the physical address, client, and budgeted price. Each of the *Wall*, *Window* and *Foundation* objects will consist of some unique and some similar information.

Figure 4 consists of two parts; the first part shows a UML class-diagram of these objects and their attributes. The second part displays instances of an example project consisting of a *Building* object, BoschendalStreet; two *Wall* objects, WallEast and WallNorth; one *Window* object, WallEastWindow1; and two *Foundation* objects, FoundationEast and FoundationNorth.

The *Wall*, *Window* and *Foundation* objects are similar to ones described in Section 1.5, Figure 2. Aside from attributes that are specific to each of these objects, all of them have a name and a location. If the programmer wishes to store these objects in one set in the *Building* object, these objects will have to be stored as objects of type *Object*. If this is the case, simply retrieving the information stored in the name and location attributes will require expensive reflexion techniques. The following code snippets demonstrate the difference for the simple process of retrieving a specific *Wall* object by specifying its name.

Using reflexion:

```
public Wall getWall(String name) {
   Set<Object> elements = getElements();
   for(Object element: elements)
      if(element instanceOf Wall) {
         Wall wall = (Wall) element;
         if(wall.name==name)
            return wall;
      }
   return null;
}
```

Without reflexion:

```
public Wall getWall(String name) {
   Set<Wall> walls = getWalls();
   for(Wall wall: walls)
      if(wall.name==name)
         return wall;
   return null;
}
```

With the reflexion process an additional check (`element instanceOf Wall`) has to be performed and the returned object first has to be casted as a *Wall* object.

To prevent this, the *Building* object consists of a set for each of these component objects (sets walls, windows and foundations). If a new object is required for customisation reasons, e.g. a *Roof* object, a new separate set will have to be created for storing the *Roof* objects, requiring a change within the core of the program. Customisation should preferably not change the core of an application. If different customised applications share the same core structure, interoperability between them is easier.

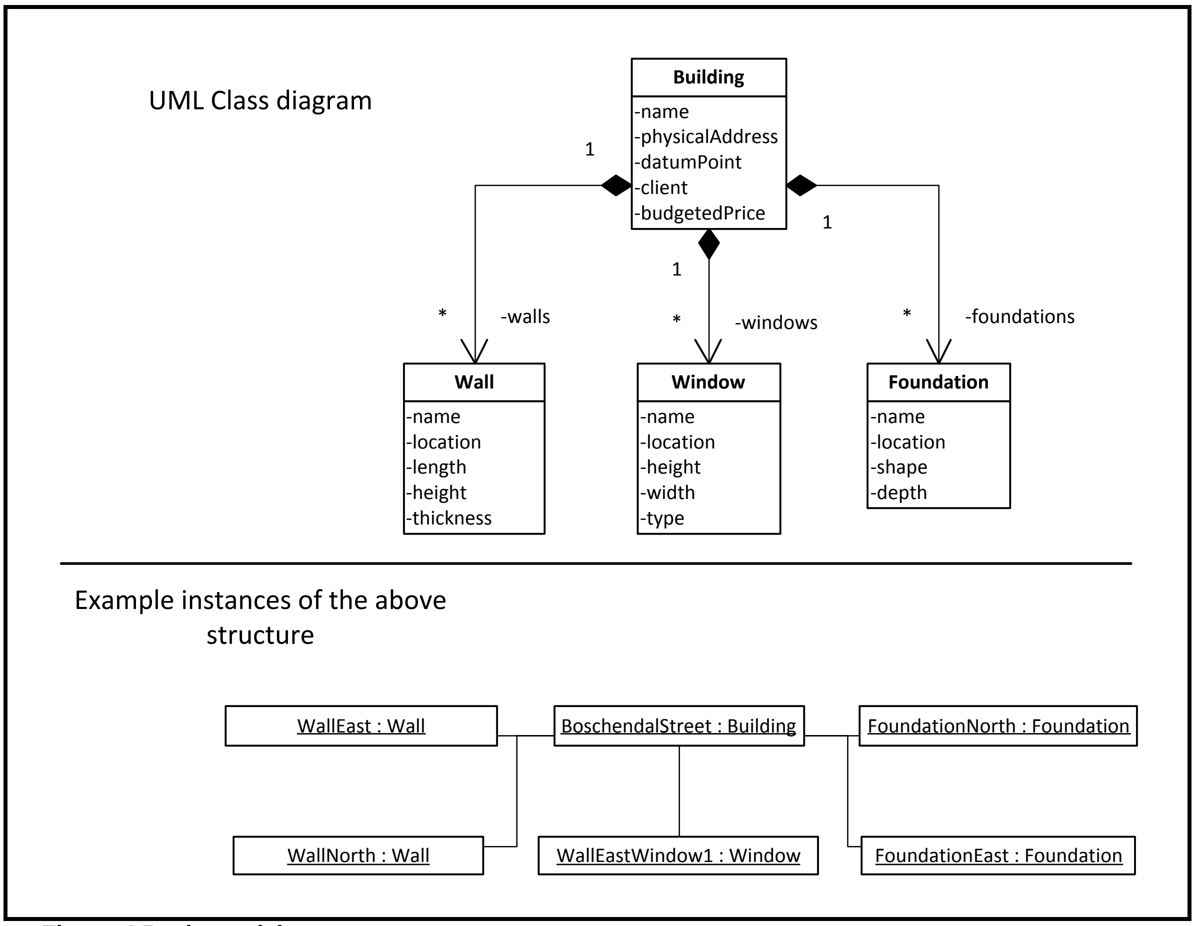


Figure 4 Basic model

2.2 First abstraction

As discussed in Section 1.5, it is desirable to define the attributes that are similar in classes *Wall*, *Window* and *Foundation* in a separate superclass, thus reducing the amount of information defined in each class. Figure 5 demonstrates this structure.

Classes *Wall*, *Window* and *Foundation* now only define the attributes that are unique to each. A further advantage is that these objects can be stored in one set within the *Building* object. Custom *BuildingElement* objects can also be stored within this set as long as it extends the class *BuildingElement*. This means that custom type objects can be created and stored in the Model of the application without any change to the core of the application.

The most important advantage of this abstraction is that when building elements are created, only their defining attributes and methods have to be specified. More complex methods and attributes might be required by the application, for example an effective hashCode-method using a unique persistent identifier. By creating these methods and attributes in the *BuildingElement* class, the user or programmer does not have to do it every time a custom

BuildingElement is created. This allows any programmer or user to define his own custom building elements without having to understand the intricacies of the application.

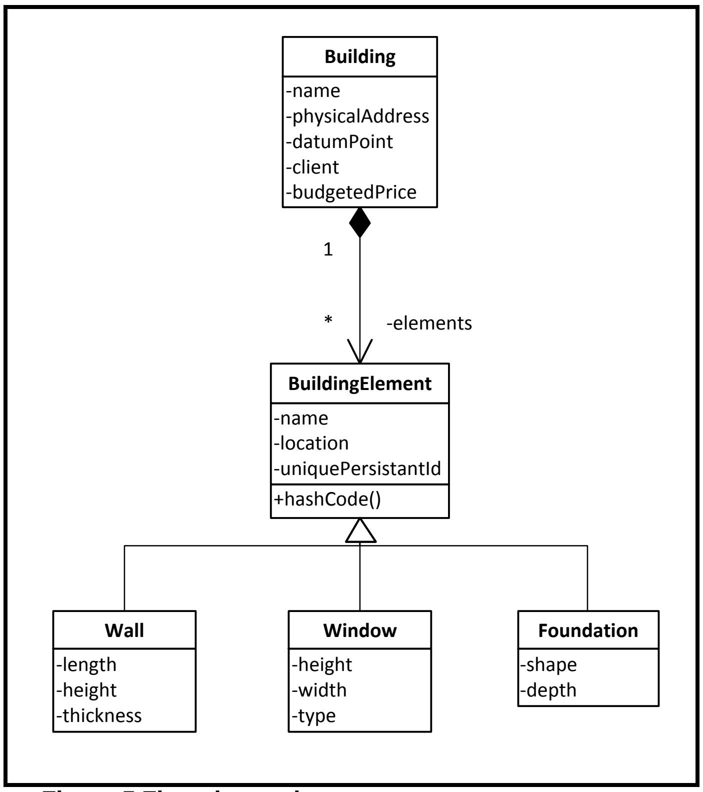


Figure 5 First abstraction

2.3 Examining object-oriented models

The structure at this point is essentially similar to most other BIM-style object models. Some of the components of the model are predefined and custom components can be defined by extending another component. In order to obtain a more flexible system, the method by which objects from the real world is mapped to software models has to be examined.

In a real world situation complex systems are broken down into simpler, more understandable **entities**. It is quite clear how a building can be broken down into its different parts such as walls, windows, foundations, roofs, columns, beams, slabs, etc. However, more abstract concepts such as a Bill of Quantities can also be broken up into entities. A Bill of Quantities does not necessarily consist of wall and foundation entities; rather it would consist of masonry, concrete and reinforcement items. Entities can be split up into two parts, namely **properties** and **functionalities**. Examples of properties are length, height, window type, unit rate, task duration, etc. In terms of functionality, Bill of Quantity items are used to calculate the price of the project. The properties can assume different values, for example two beam entities will both have a length property, but the value of the length for each can differ. It is said that the entities are **equivalent** and characteristic of **type** beam.

In mathematical terms, let the set F collect all entities related to the Building Environment.

$$E \coloneqq \{e \mid e\ is\ an\ entity\ in\ the\ building\ environment\}$$

The set $F(e_i)$ is the set of functionalities of the entity e_i and $P(e_i)$ is the set of properties of entity e_i.

$$F(e_i) \coloneqq \{f(e_i) \mid f(e_i)\ is\ a\ functionality\ of\ entity\ e_i\}$$

$$P(e_i) \coloneqq \{p(e_i) \mid p(e_i)\ is\ a\ property\ of\ entity\ e_i\}$$

Two entities e_i and e_j are of the same type (equivalent) if the set $F(e_i)$ consists of exactly the same functionalities as set $F\left(e_j\right)$; and the set $P(e_i)$ consist of exactly the same properties as the set $P\left(e_j\right)$, even though the values of the properties in $P(e_i)$ can differ from the values of the properties in $P\left(e_j\right)$. In other words for two entities to be of the same type, the two entities must consist of the same functionalities and properties, but the values of the properties can differ. If two entities are of the same type, and in addition the values of the properties are equal, the two entities equal. Entities are therefore defined by their sets of functionalities and properties.

In object-oriented programming, software models comprise of **objects**. The objects are normally defined to resemble real world entities (refer Section 2.1). Objects are defined in **classes**. Classes specify certain **attributes** and **methods** that an object must have. Attributes or data fields are used to represent the properties of real world entities. Methods are used to represent the functionalities of the object.

As an example, consider the real world entity "item". Each item has a specific unit rate and quantity, and it is used to calculate the price of a project. In a software model, this entity can be represented by an *Item* object. An *Item* object will have two attributes, namely unitRate and quantity; and a method calculateItemPrice, which calculates the price of the item as unitRate × quantity. By summation of item costs, the total price of the project can be computed.

In mathematical terms, let the set O collect all objects in a software model.

$$O \coloneqq \{o \mid o\ is\ an\ object\ in\ a\ software\ model\}$$

The set $M(o_i)$ is the set of methods of the object o_i and $A(o_i)$ is the set of attributes of object o_i.

$$M(o_i) \coloneqq \{m(o_i) \mid m(o_i)\ is\ a\ method\ of\ object\ o_i\}$$

$$A(o_i) \coloneqq \{a(o_i) \mid a(o_i)\ is\ an\ attribute\ of\ object\ o_i\}$$

Similar to entities, objects o_i and o_j are equivalent if set $M(o_i)$ consists of exactly the same methods as set $M(o_j)$; and the set $A(o_i)$ consist of exactly the same properties as the set $A(o_j)$. Objects are equal if, in addition to the above conditions, the values of the properties are equal as well.

2.4 Second Abstraction

A problem with models created using object-oriented programming techniques is that all attributes and methods have to be defined in the source code of the program. Once the application has been compiled, the attributes and methods defined by the different classes are fixed. To change them requires a new version of the application, which can result in problems between different versions of the application.

Another problem is that classes often contain attributes and methods that do not directly resemble any properties and functionalities of entities in the real world. In most cases this situation is unavoidable as these attributes and methods are required by more intricate methods in the core of the application. To programmers in charge of customisation and end-users without knowledge of the intricacies of the core, these attributes and methods can be confusing.

In order to address these problems a new class is created, namely the *Feature* class. Each element of the *BuildingModel* comprises a set of *Feature* objects. These *Feature* objects more directly represent the properties of the entities in real world. *Feature* objects do not replace the attributes of the objects, but should rather be used in conjunction with attributes. An *Element* object is therefore not anymore defined only by its methods and attributes, but rather by its methods, attributes and features. In keeping with the terms laid out in Section 2.4, let the set $U(o_i)$ collect all the features of object o_i.

$$U(o_i) \coloneqq \{u(o_i) \mid u(o_i)\ is\ a\ feature\ of\ object\ o_i\}$$

Objects o_i and o_j are now equivalent if set $M(o_i)$ consists of exactly the same methods as set $M(o_j)$; and the set $A(o_i)$ consist of exactly the same properties as the set (o_j); and the set $U(o_i)$ consist of exactly the same features as the set $U(o_j)$. Objects are equal if, in addition to the above conditions, the values of the properties as well as values of the features are equal.

Properties and attributes are characterised by a name and a value. For example a wall entity has a length of 12.5 m. It therefore has a property named "length" with a value of "14.6" assigned to it. Similarly, *Feature* objects consist of a name and value. Implementing this into the model demonstrated in Figure 5, the structure defined in Figure 6 is achieved.

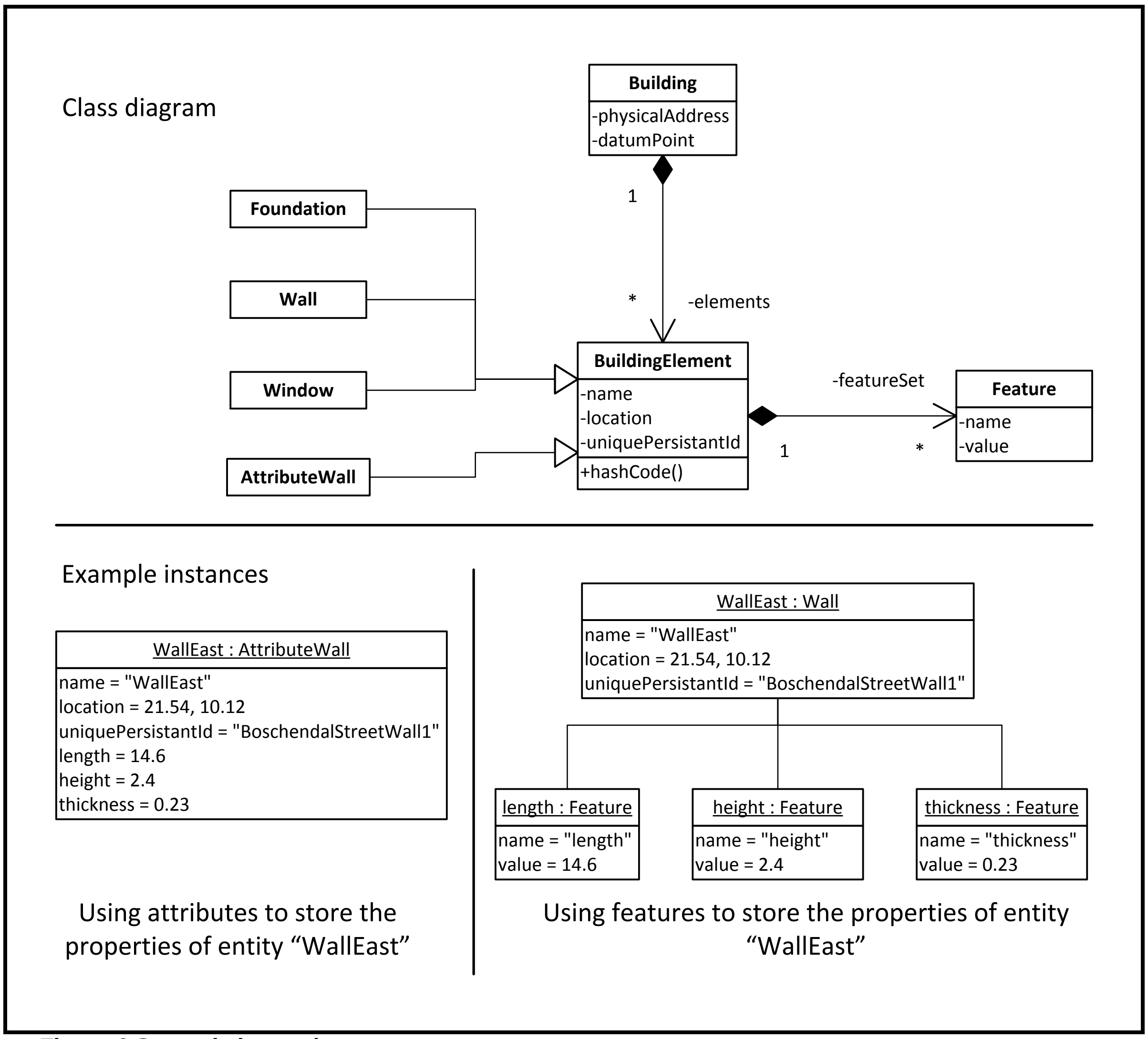


Figure 6 Second abstraction

In Figure 6 the eastern wall entity of the Boschendal Street building has a length of 14.6 m, a height of 2.4 m and a thickness of 0.23 m. It therefore has properties length, height and thickness. It can be represented by the object *AttributeWall*, which uses attributes to represent the properties of the entity. It can also be represented by a *Wall* object. The class *Wall* extends the class *BuildingElement*. *BuildingElement* defines an object attribute called featureSet, which is a set that stores *Feature* objects. Since the class *Wall* is a subclass of class *BuildingElement*, the WallEast object inherits this set and stores three different features in this set: length, height and thickness. The values of these features are 14.6, 2.4 and 0.23 respectively. The *Wall* object still has other attributes as well, such as the uniquePersistantId, but these are not of interest to the end-user.

The set of features can be expanded at runtime. The user can decide to add another feature, e.g. a feature to represent the thermal efficiency of the wall. This new feature will form part of the definition of the object along the other standard features. Customisation can now be achieved at runtime, instead of within the source code.

2.5 Feature objects

Objects that have feature sets have to be differentiated from objects that do not. A new class is created for this purpose – class *FeatureObject*. The term "feature objects" will be used for the remainder of this thesis to describe these objects. The exact definition of feature objects follows later in this section after another concept is introduced.

As mentioned in Section 2.3, standard software objects are defined using a class. The class, together with its superclasses, defines the object's attributes, set $A(o_j)$. Furthermore, the class along with its superclasses define the methods of the object, set $M(o_j)$. If a class implements an interface, the interface will prescribe certain methods that the class must implement. Interfaces therefore also influence the set of methods of an object.

A new concept has to be introduced to define default features of feature objects, namely the concept of a "family" and its "members". A family has a set of features, called traits, that its members must have. Differently put, if a feature object is a **member of** a certain family, it has all the traits of that family.

The **member of** relationship between a feature object and its family is analogous to the **instance of** relationship between an object and its class.

Feature objects are members of a family regardless of their methods and attributes. This means that objects that are instances of two different classes can be members of the same family.

A formal definition of feature objects now follows:

Feature objects are objects that have a set of features. A feature object is a member of one and only one family. A family has a set of traits that all of its members have in their sets of features.

The above definition only describes the concept of feature objects and the relationship between them and their families. A software structure has to be developed that support this. Starting off then, the classes *FeatureObject* and *Family* will be used to represent feature objects and families. A *FeatureObject* object has a set that stores *Feature* objects, it also has a *Family* attribute called family. It has methods for adding and removing features from its feature set. In this way features can be added and removed at runtime. A *Family* object has a set of prescribed features, called traits. Figure 6 is revisited below, now incorporating the feature object concept.

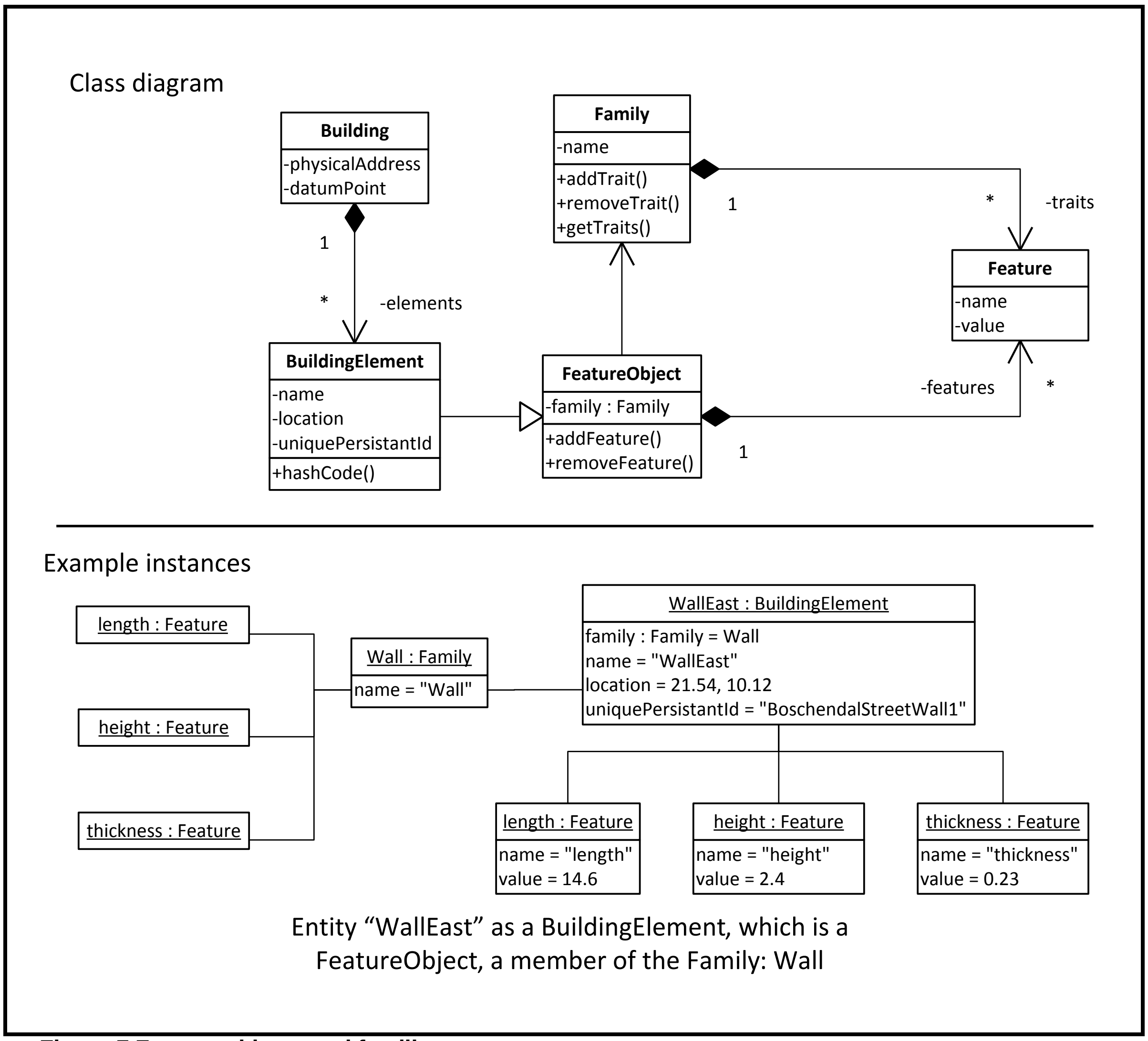


Figure 7 Feature objects and families

As demonstrated in Figure 7, a *Building* object still consists of *BuildingElement* objects; however the *BuildingElement* class now extend the *FeatureObject* class. The WallEast entity is represented by a *BuildingElement*. Since a *BuildingElement* is a *FeatureObject*, it has an attribute called family. In the case of object WallEast, its family attribute is the family Wall, whose traits are length, height and thickness.

2.6 Feature types

When defining attributes, it is useful to be able to define the datatypes of properties. For example, the height of a wall is a number, while the name of the object is represented with text. In order to distinguish between different kinds of features, different types of *Feature* objects are necessary. For example, a building element entity is either finished or still unfinished, a feature that can be represented with a Boolean (true or false).

A typed feature is a feature whose "value" attribute is of a specific data type. For example a *NumberFeature* can be used to represent numerical properties. A double attribute can be used to store the value of a *NumberFeature*.

As a basis, the three most common data types are: text, numbers and Booleans. Three different types of *Feature* objects are used to represent these in the framework: *TextFeature*, *NumberFeature* and *BooleanFeature*. All three of these are subclasses of the class *Feature*. The value-attribute of a *TextFeature* is of type String, the standard datatype for text in Java. The *NumberFeature* has a Java primitive double as its value-attribute. The *BooleanFeature* has a Java primitive boolean as its value-attribute. The value-attribute of standard *Feature* objects is of type *Object*. All objects in Java extend this class by default, and therefore any type of object can be stored in this *Feature* object. This is done to allow users to store custom object-types in features as well.

The different types of *Feature* objects are deliberately **not** called *StringFeature* and *DoubleFeature* or *IntegerFeature*, since an end-user might not recognise or understand these names. The ideology behind *Feature* objects is to allow the end-user to do some customising as well, in other words to bring the inner workings of the application closer to the nonprogrammer end-user. For this reason their names are kept as close as possible to real-life counterparts. Furthermore, a parseString-method is included for each that converts text to a value that can be stored in the value-attribute. Reason being that most input from an end-user will be in the form of text input.

A programmer can create his own *Feature* object types by extending the standard *Feature* object.

2.7 Prototype source code of relevance to this chapter

Of specific relevance to this chapter is the object types *FeatureObject* and *Feature*. Both these are defined not using classes, but rather using interfaces. This allows a programmer to create customised objects and implement the functionality in his own way. Furthermore, a programmer might want existing classes to implement this functionality without having to extend the *FeatureObject* or *Feature* classes. This is especially relevant to Java where polymorphism is not allowed – the existing class might already extend another class. If a programmer does not want to implement functionality himself, abstract classes was created that implement some of the most intricate functionality. This pattern is repeated at numerous other places in the source code.

In the case of the object type *FeatureObject*, the abstract class *AbstractFeatureObject* readily implements the interface *FeatureObject* and some of the functionality required by the interface. A user can easily extend this abstract class and complete the abstract methods.

In the case of the *Feature* object type, no abstract class was created. If the programmer wants to create custom *Feature* objects without implementing the *Feature* interface, he can extend the class *StandardFeature* and override methods as necessary.

Interface *FeatureObject* and class *AbstractFeatureObject* is in the package *model*, while interface *Feature* and class *StandardFeature* is in the package *model.features*. Of specific importance is the clone-method in *Feature* objects. The family traits are stored as a set of *Feature* objects. Whenever a new *FeatureObject* object is created, the traits of its family are cloned and added to its set of features.

In the prototype implementation, a *FeatureObject* object with a "null" family attribute is interpreted as a *FeatureObject* object that does not belong to a family. In the *AbstractFeatureObject* class, the getFamily-method returns null, and should be overridden if the programmer desires an object to belong to a family. The reason for this is that, because of the usefulness of the feature object concept, many objects within the prototype implement this functionality. This allows the user to add features not just to the components of the models but to all other objects that implement the feature object functionality.

As an example, *Family* objects are also *FeatureObjects*, allowing a user to add features to *Family* objects. A user can therefore add collective features, shared by all members of a family, to the family object instead of its individual members. The mass of a wall per unit volume, for instance, is the same for all wall entities; and should rather be stored as a feature of the "Wall" family. The difference between features of a family and features of its members is similar to the difference between static and object attributes. Class Family is located in the package *model*.

3 Enhancing the Model

This chapter enhances the model created in Chapter 2 by introducing a better way of modelling derived features that ensures data integrity. The structure of the model is further improved by adding child-parent structural concepts. The result is a more efficient and logical model structure.

3.1 Derived features

Section 1.3 explains how incorrect storage of derived attributes can lead to loss of data integrity. The same argument is valid for features.

As an example, take the wall entity represented in Figure 7 by the feature object WallEast. The mass of a wall is equal to the product of the length, height, thickness and density of the wall. The mass is therefore a derived property that depends on the length, height, thickness and density of the wall. If any of these properties change, the mass must also change.

In the software model, the mass property can be represented by an attribute of the *Wall* object; however, if this is the case, an inconsistency can occur between the mass and the attributes or features that it depends on. A better way of representing the mass of an entity is to create a method that calculates the mass instead of storing it as an attribute. Every time the mass of the object will be required, this method has to be called, which will calculate the mass correctly according to the current state of the object. The problem with using methods to represent derived properties is, as with attributes, that methods have to be defined within the source code. Methods cannot be added to objects after creation.

In the proposed integration framework, derived properties have to be stored as a different type of feature. These features must not store values, but rather the methods by which the values of the features are obtained. The question is how can the methods be stored.

In normal programming procedures, the user creates source code, which basically is a set of instructions for the computer to perform. The source code is created in a language such as C#, Java or Basic to name only a few. Depending on the language, the code is either interpreted or compiled, or a combination of both, in order for the computer to perform the set of instructions.

In order to create a way in which the methods behind derived features can be stored, the process described above has to be modelled somehow, albeit on a much smaller scale than is the case for normal programming purposes. The programming process consists of three steps as shown in the UML State chart of Figure 8.

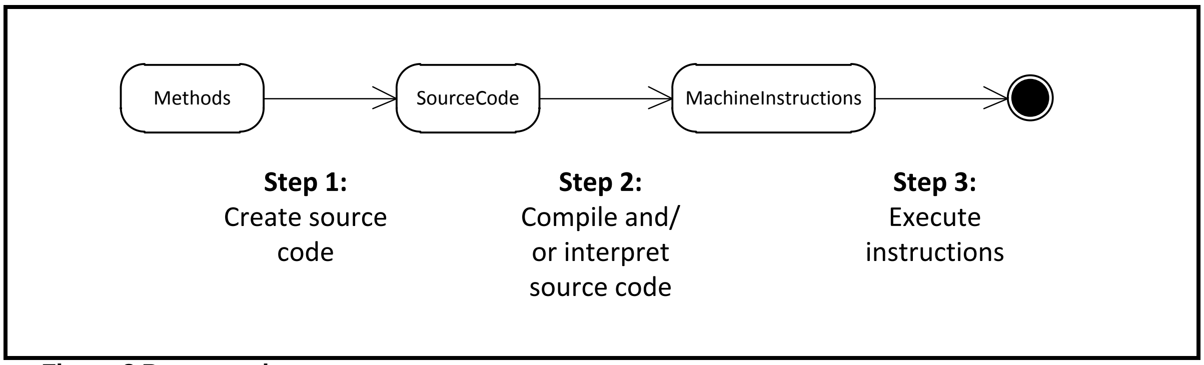

Methods
SourceCode
MachineInstructions
Step 1: Create source code
Step 2: Compile and/ or interpret source code
Step 3: Execute instructions

Figure 8 Programming process

In order for the process to be imitated, an interpreter or compiler object is needed to carry out Step 2 in Figure 8. The function of the interpreter or compiler is to convert the source code created by the user into a set of instructions that the computer can perform. Since the interpreter or compiler must "understand" the source code created by the user, the user will have to create source code in a certain programming language. An interpreter that can understand that language must then be used to create computer instructions.

Since the framework is modelled in a Java environment, it was decided to use an interpreter that can interpret source code written with in the Java programming language. The BeanShell interpreter created by Pat Niemeyer is a "small, free, embeddable Java source interpreter" (BeanShell, n.d.). The advantage of the BeanShell interpreter is that it runs within the same program environment as the application, which means the objects of the application can be directly referenced and used. Furthermore, the BeanShell interpreter also supports some scripting features such as an extensible set of shell-like commands and also optionally typed variables. This makes it easier for novice programmers to put methodology into source code, which makes it more accessible to end-users.

Even though normal Java programming procedures are simulated with the BeanShell interpreter, it differs in the way it converts source code into instructions executed by the computer. In normal Java programming procedures, step 2 involves compiling the source code to byte code, which is code that the Java virtual machine can interpret. The Java virtual machine is an application that simulates a universal computer platform regardless of the operating system. This allows the same byte code to be executed on any platform without any conversion necessary. For all intents and purposes the Java virtual machine can be seen as the computer in Step 3 of Figure 8. The BeanShell interpreter is a subsystem of the application used to create the model. It interprets source code inserted by the user and directly uses this code as a set of instructions (script) for the computer (Java virtual machine) to execute. Figure 9 is a UML sequence diagram that shows in more detail the sequence of events from creation of the application up to its end.

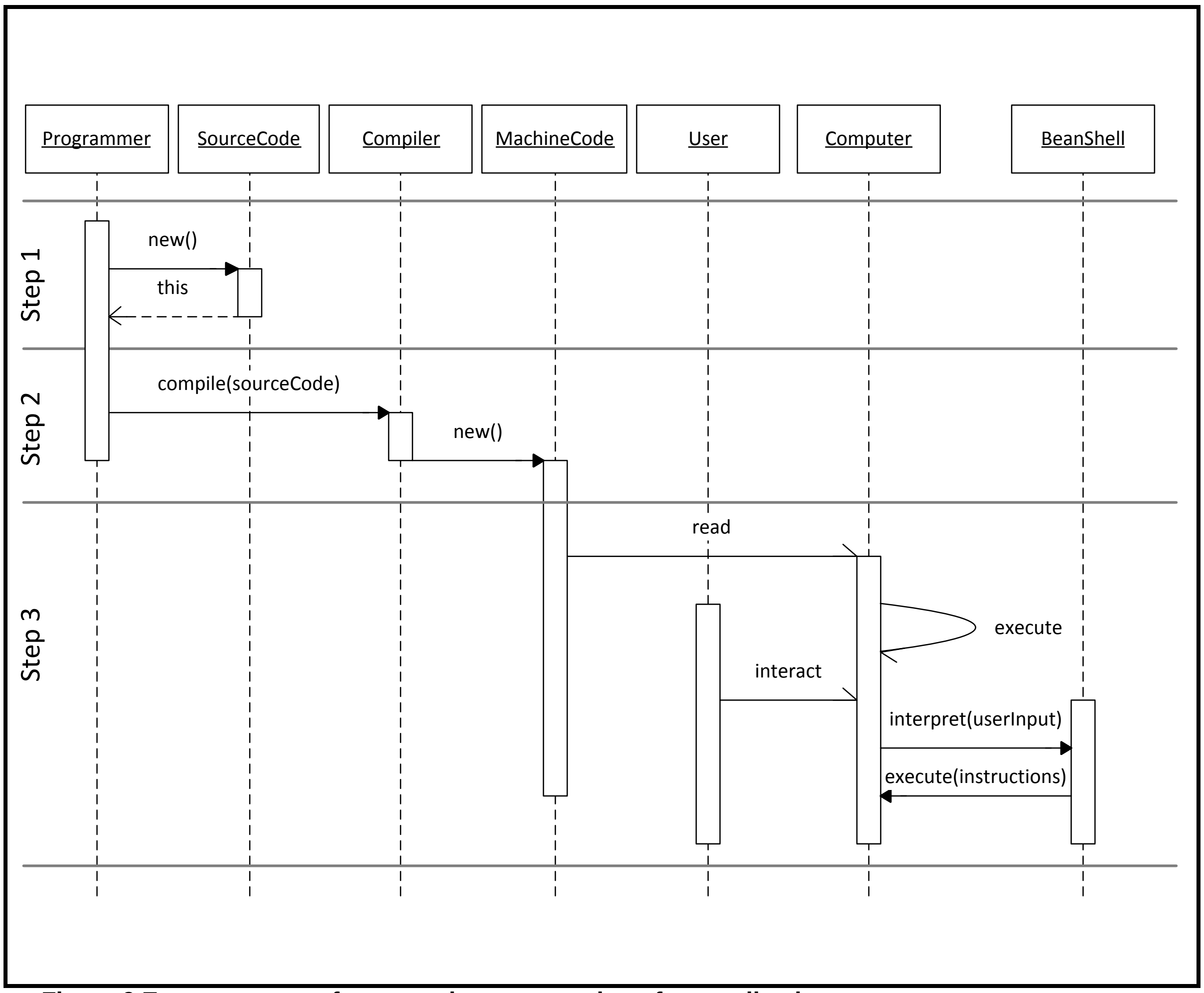


Figure 9 Event sequence from creation to execution of an application

In Figure 9, a programmer creates source code which is then compiled using a compiler. The compiler uses the source code and generates machine code, or in Java terms byte code. As soon as the application is started, the computer reads the instructions in the machine code and starts to execute them. As the computer executes the instructions, it interacts with the user. Some of the interactions might include the user creating source code for the BeanShell interpreter to interpret. As soon as the BeanShell interpreter has interpreted the newly created source code, it directly instructs the computer what to do; without first creating machine code.

The BeanShell interpreter can therefore be used to change a program without having to change the source code. A further advantage is that it operates within the application's environment, i.e. it can directly access the objects within the application and call methods on these objects and modify their attributes.

To incorporate the BeanShell interpreter in the framework, two new object types are required. The *DerivedFeature* object is a special type of *Feature* object used to represent derived properties. It is a subclass of class *Feature*. The class *BeanShellFeature* is a subclass of class *DerivedFeature*. An object of this class has a BeanShell interpreter that it uses to compute its value. Whereas normal Feature objects store values assigned to them directly as value-attributes, *BeanShellFeature* objects store its methods in the form of scripts. A script is text that formulates the instructions that the computer must carry out to derive the value of the feature. Each interpreter will have its own format in which this text must be. With the Beanshell interpreter this text will resemble normal Java source code.[1] When the getValue-method is called on a *BeanShellFeature* object, the interpreter evaluates the text in the script by interpreting it and instructing the computer to carry out the steps documented in the script.

As an example, say the user wants to create a derived feature that represents the surface area of a wall. The surface area is calculated as:

$$surfaceArea = length * height$$

The script that the user has to insert is simply:

```
value = length * height;
```

The getValue-method in class *BeanshellFeature* is as follows:

```
public Object getValue() {
    . . .                // Code left out for demonstration purposes
    interpreter.eval(script);
    . . .                // Code left out for demonstration purposes
    return value;
}
```

Some code has been left out to better illustrate the interpreter concept. Each interpreter will differ in the way it is stored, called, created, etc.; therefore only key concepts is shown in the code snippet above.

Although a Java-based interpreter was chosen for the prototype of the framework created by this thesis, a different interpreter could be used with as much success. There exist several interpreters similar to the BeanShell interpreter that can be used. A programmer can also

[1] Refer to the *BeanShellFeature* class description at the end of this chapter for a more in-depth explanation of how the BeanShell interpreter is utilised in this class.

create his own scripting language and interpreter and incorporate it into the application similarly to how the BeanShell interpreter was incorporated.

3.2 Child-parent structures

The child-parent structure is a useful structure that can be used to simplify large object models. The concept is that a parent has several child objects. Each of the child objects can, in turn, consist of several other child objects, etc. The parent and child objects can be the same object type. The resulting structure resembles a tree as shown in Figure 10.

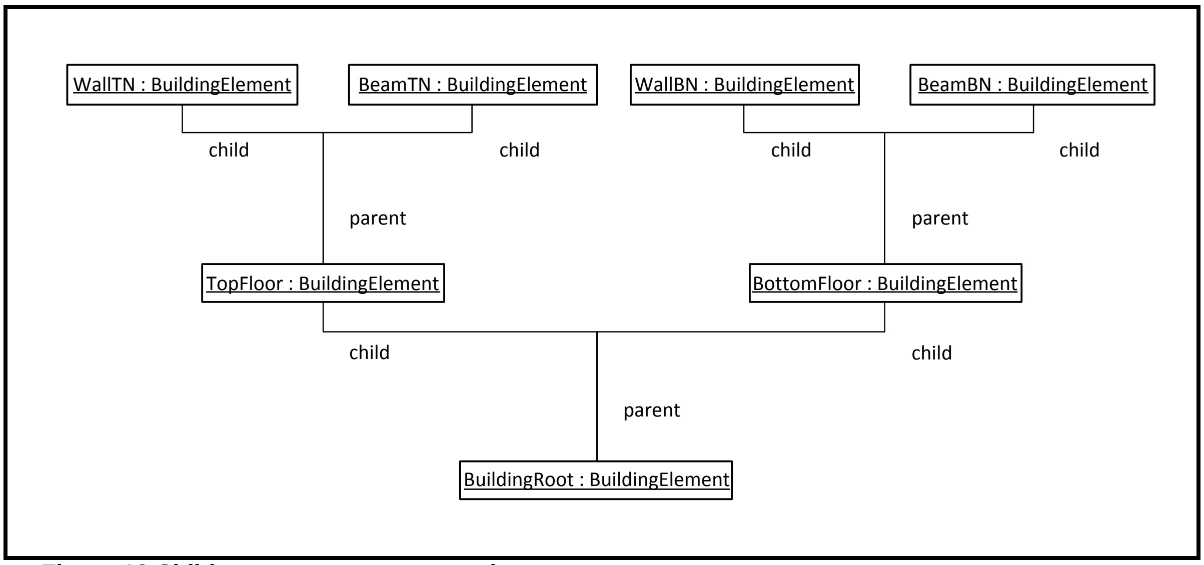


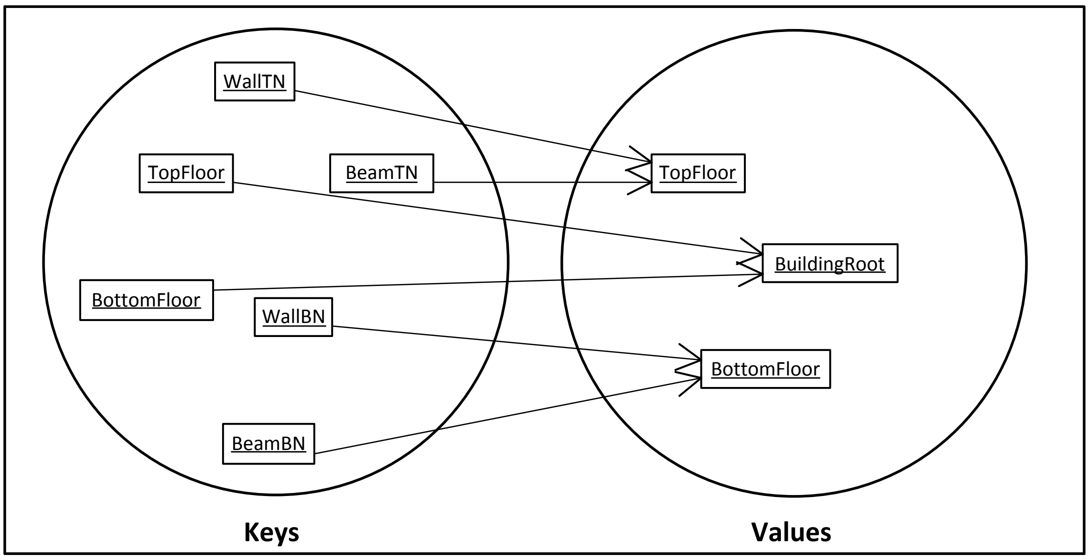
Figure 10 Child-parent structure example

It comes quite naturally to divide entities in the real world into groups of similar entities, then to further divide these groups into more specialised groups, and so forth. As an example, it is easy to understand how a building can be divided into different floors, and then divide the different floors into beams, columns, slabs, walls, windows and doors. If desired, each of these entities can also be divided into more specialised entities. In this way, a complex concept can be made more understandable. Figure 10 shows how *BuildingModel* objects can be subdivided to better resemble their real world counterparts.

In theory, if all the objects in the tree are of the same type, each of these objects has a parent object and a set of child objects. One way of implementing this structure is to add two attributes to the objects in the tree; a parent attribute and a set of children objects. The problem with this structure is that an inconsistency can occur between the parent attribute of the children of an object and its set of children. Referring to Figure 10 as an example, the "TopFloor" object might have object "WallTN" in its set of child objects, but the object "WallTN" has "BottomFloor" as its parent attribute. Then according to "TopFloor", "WallTN" is one of its children, while according to "WallTN", its parent is "BottomFloor".

To avoid this, the child-parent structure, that is built into the core of the framework, has an internal structure that passively prevents inconsistencies, as described below. The relation between children and parents is an example of a "many to one" relation. One parent can have many children, but a child can only have one parent.

The map concept also represents a "many to one" relation. A map consists of key-value pairs. A key can only occur once in a map, and can therefore only be associated with one value. Two different keys can however be associated with the same values. If a map is used to model the child-parent relation, where the child objects (keys) are mapped to parent objects (values), it would be impossible to create a structural inconsistency. As shown in Figure 11 it is impossible for "TopFloor" to have "WallTN" as a child if "WallTN" maps "BottomFloor" as its parent.



KeyValue pairs:
WallTN, TopFloor
BeamTN, TopFloor
TopFloor, BuildingRoot
BottomFloor, BuildingRoot
WallBN, BottomFloor
BeamBN, BottomFloor

Figure 11 Using a map to model many-to-one relations

Several classes that model the map concept exist within the standard Java libraries, among these the HashMap. A HashMap uses hashing techniques to efficiently retrieve the value associated with a given key. A problem with standard maps is that they only provide functionality for retrieving the values of a given key. It is not possible to retrieve or access the keys of a specific value. This functionality is required for the child-parent relation since it is necessary to be able to find the children (keys) of a specific parent (value). For this purpose a special type of Map is required for the integration framework, call it a *ReverseAccessMap*. A *ReverseAccessMap* has the functionality to return a set of keys that map to a specified value in addition to functionality of a normal map.

The map that controls the child-parent relations must be stored in one central object, such as the *Building* object.

To avoid confusion all *BuildingElement* objects must by definition have a parent *BuildingElement* object assigned to it. If it is desired that a *BuildingElement* object must have no parent assigned to it, it is assigned the "BuildingRoot" object. In order to ensure that all *BuildingElement* objects have a parent *BuildingElement* object assigned to it, all *BuildingElement* objects must be present in the keyset of the parent-child map. Instead of using a special set to collect all the *BuildingElement* objects in the building model, they are collected directly in the keyset of the child-parent map.

As can be seen in Figure 12, if an extra set is used to collect all objects in the *Building* object, this set will be redundant. All the keys of the childParentMap already represents a set and all *BuildingElement* objects must by definition be contained in this set.

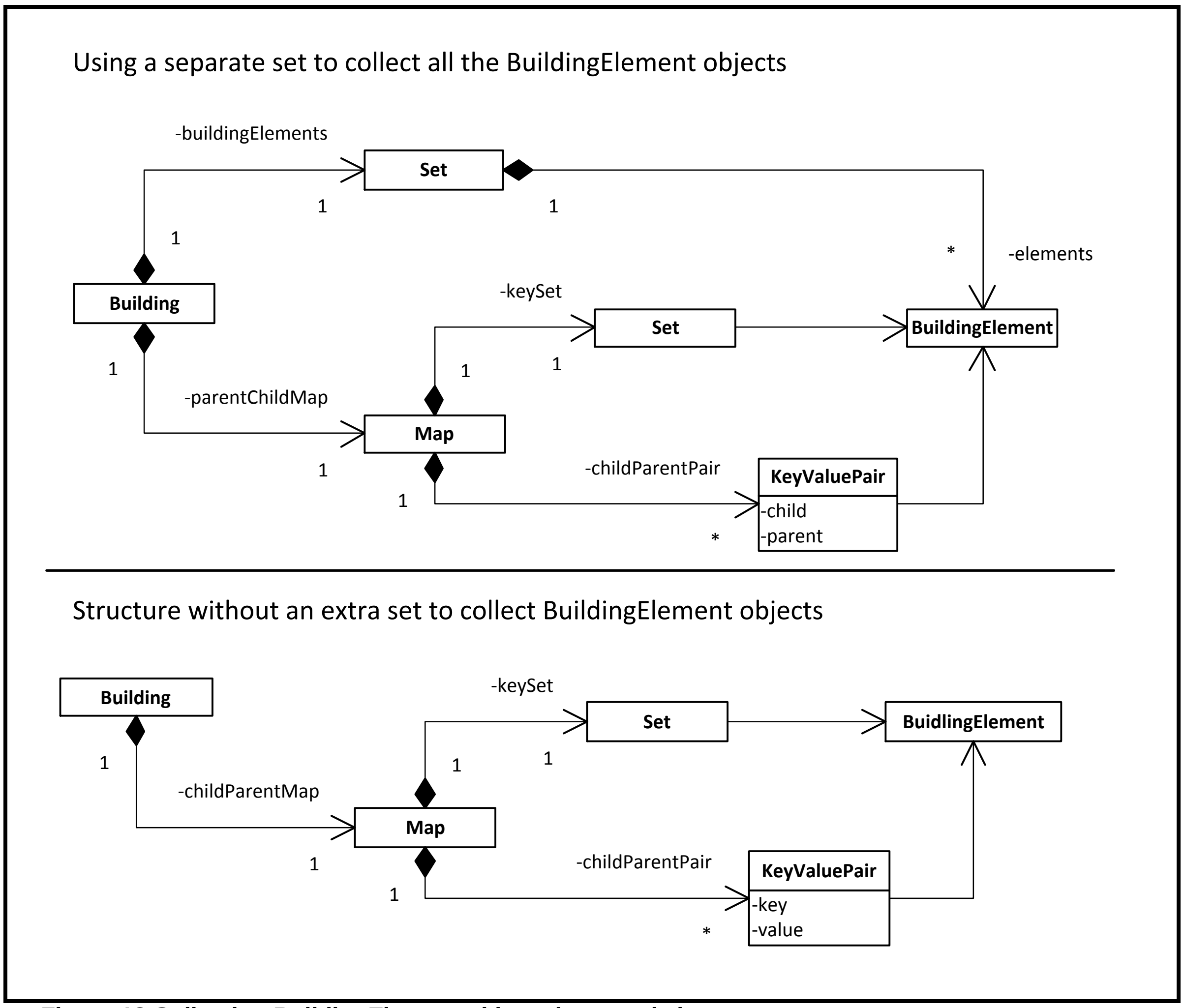


Figure 12 Collecting BuildingElement objects in a map's keyset

3.3 Child-parent structure for classifying Family objects

Just as child-parent relations can be useful to structure the *BuildingElement* objects, it can also be useful to structure the different *Family* objects. As an example, in a similar fashion to the structure used in Figure 10, the Beam and Wall families are "children" of the Floor family. Figure 13 shows the structure of the different *Family* objects as well the *BuildingElement* objects of Figure 10. This time the objects are arranged in a more logical top-down manner.

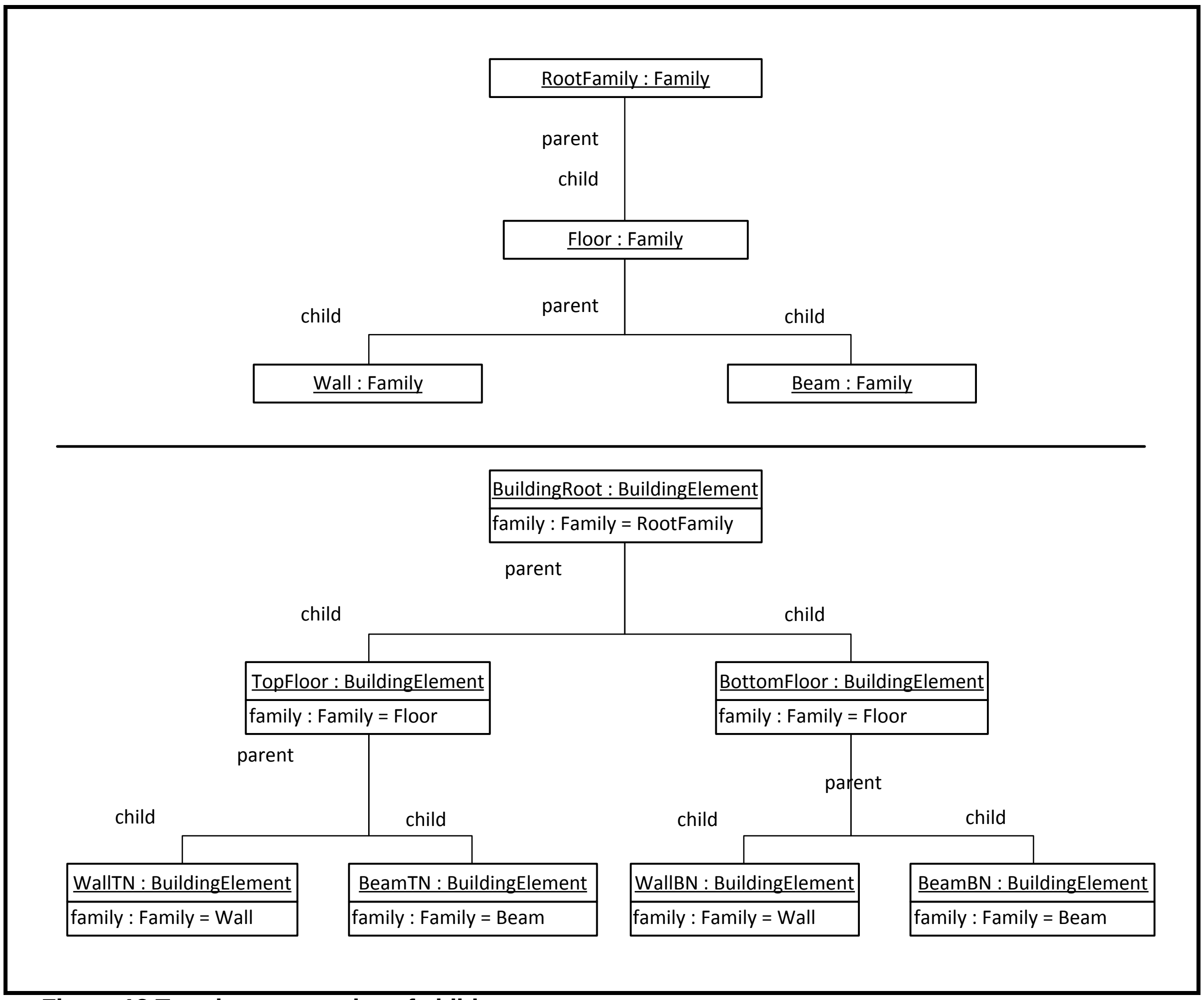


Figure 13 Top-down examples of child-parent structures

The relations on parent *Family* objects and their children objects should not be confused with the inheritance relationships between superclasses and their subclasses or interfaces and their implementing classes. Inheritance relationships imply that the subclass or implementing class inherits the attributes and methods of the superclass and the interface. This is not the case for child-parent relations between families. The children *Family* objects do not inherit the attributes, methods or traits of its parent *Family* object. Furthermore inheritance relationships represent "is a" relationships. If class *BuildingElement* extends class *FeatureObject*, it is implied that a *BuildingElement* object "is a" *FeatureObject*. A wall is not a floor, even though the *Family* object "Wall" is a child of *Family* object "Floor". In Java programming, coherent classes are grouped into packages, which resemble files in different folders. This is done to create a more understandable structure within the source code. The child-parent classification system resembles this type of structure.

All the families of a model have to be collected in a centralised object. Once again, as in the case of child-parent relationships between *BuildingElement* objects, a map can be used to serve the dual purpose of collecting all the *Family* objects and controlling the child-parent relations between them.

3.4 Prototype source code of relevance to this chapter

The *BeanshellFeature* class located in package *model.features* demonstrates the use of the Beanshell interpreter. Each *BeanshellFeature* object has its own *Interpreter* object. The interpreter allows certain attributes in the script to be predefined. For example, the value-attribute is set to the represent the value-attribute of the feature. The *FeatureObject* object that consists of the *BeanshellFeature* is the *BeanshellFeature*'s host. All of the features of the host are also predefined as attributes that can be used in the script. As a demonstration, refer the following script:

```
value = height * length;
```

The interpreter will interpret the script as follows: Set the value-attibute of the *BeanshellFeature* object to the value of the "height" feature of the host *FeatureObject* object, multiplied by the value of the "length" feature of the host *FeatureObject* object. Whenever the getValue-method of the feature is called, the interpreter will interpret the above script and then return whatever is stored in the value-attribute.

The initializeScript-method within class *BeanshellFeature* establishes these mappings.

The *ReverseAccessMap* used to collect the *Family* and *Element* objects are located in the package *util*. It uses two maps to maintain its data – a mainMap and a reverseMap. It is an observable map and all registered listeners are notified as soon as changes occur in it.

Objects with child-parent relations described above implements the *ChildParentObject* interface in the package *model*. The class that contains the map used to control the child-parent relations implements the *ChildParentController* interface, also in the package *model*. The *Family* objects and *BuildingElement* objects of Figure 13 are examples of *ChildParentObject* objects. They would share the same *ChildParentController*, namely object "Building".

4 Structuring supplementary models

Building on the concepts of the previous Chapters, this Chapter focusses on the internal structure of the applications that will be integrated with each other. The core model and supplementary models concept is discussed, however the applications required by the case study are examined first in order to gradually introduce abstract similarities between the applications being integrated. Ultimately, for seamless integration to take place, these structures have to be linked with each other. However, before the structures can be linked, the objects that must be associated with each other have to be defined. In line with the case study, two ways of defining these objects are demonstrated. Firstly, a basic BoQ program structure is defined from the ground up, demonstrating how the principles of the previous chapters can be incorporated from the beginning. Secondly, a structure is defined that operate alongside an existing structure of a CAD application. This demonstrates how existing applications can be brought into the flexible integration environment. The next chapter will demonstrate how these structures can be linked with each other.

4.1 Case study: Bill of Quantities

The company from the case study requires a basic Bill of Quantities application; namely the BoQ program. To understand how such a program can be integrated with other engineering applications, the real world entities that it represents must first be examined.

A Bill of Quantities essentially comprises a list of items. All aspects of the project that contribute to its cost, e.g. tasks, materials, equipment hiring, etc., are listed as items in the Bill of Quantities. In order to add structure to the document, it is divided into different categories, each of which is in turn divided into subcategories, and so forth up to the level where the actual items are listed.

The BoQ program follows the structure suggested by the JBCC (Joint Building Contracts Committee, n.d.). According to this structure, a Bill of Quantities document is divided into different sections. Each of these sections is divided into different bills. Two different sections can contain the same bills, for example two sections can be used to represent two phases of the project, with each section consisting of the same bills. A bill in turn is subdivided into different headings, which is subdivided again into subheadings. Finally, the actual items are listed under the subheadings. Figure 14 shows how this structure can be modelled using conventional object oriented methods. At the most basic level, it can be said that one *BillOfQuantites* object consists of many *Item* objects. *Section*, *Bill*, *Heading* and *Subheading* objects are added to assist in classifying the different *Item* objects.

An item has a unit rate, a quantity and a unit. Its rate is calculated as the product of its unit rate and quantity. A subheading has a subtotal, which is the sum of the items listed underneath it. A heading's subtotal is the sum of the subtotals of its subheadings, and so forth for the sections and bills.

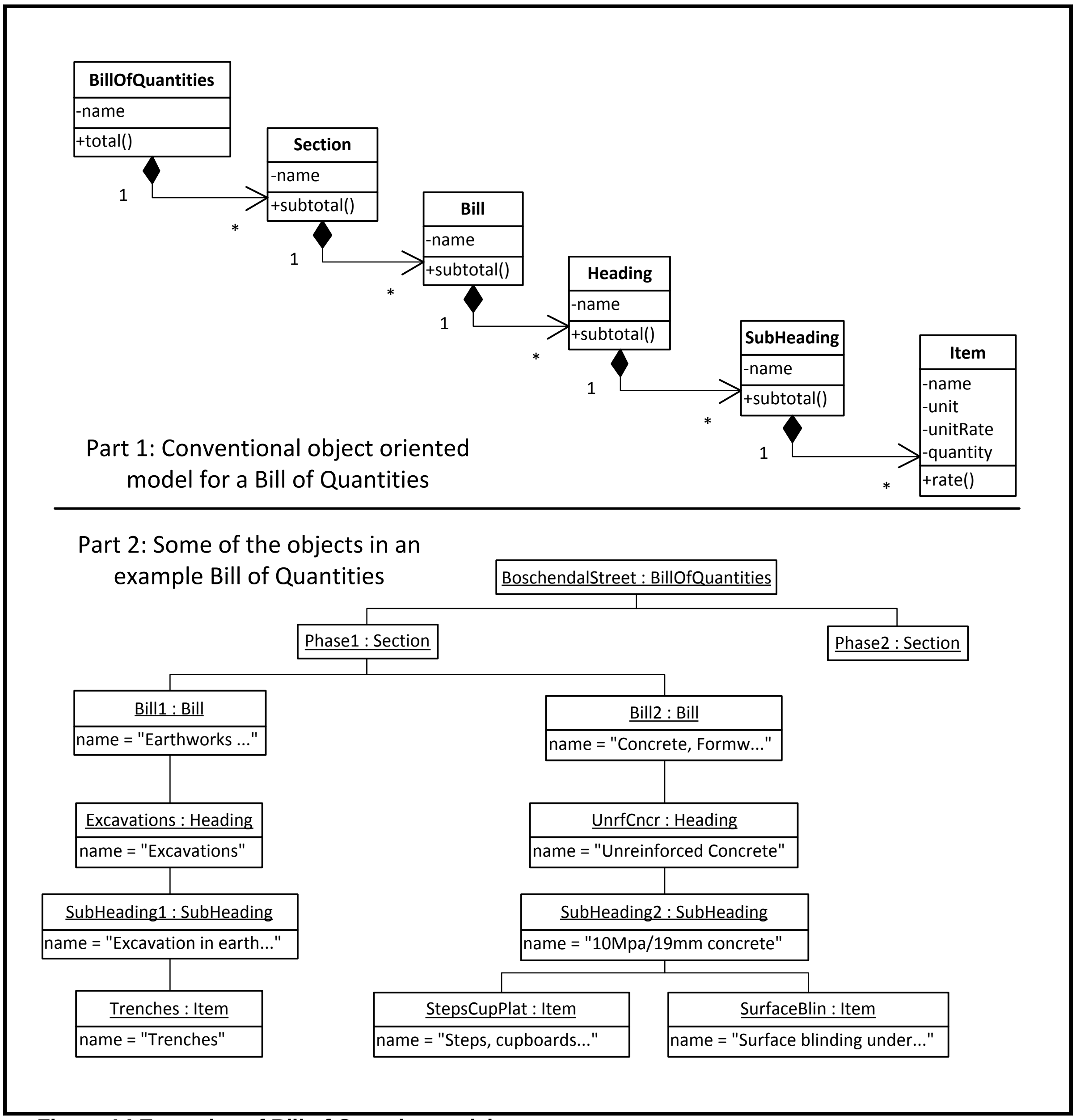


Figure 14 Examples of Bill of Quantity models

When defining software objects for the entities described above, these objects can benefit to a great extent from the structures built into the *BuildingElement* objects as described in Chapters 2 and 3. For example, class *Item* can be a specialisation of class *FeatureObject*, and its quantity property can be defined using a *Feature* object. If the user at a later stage wants to insert a custom method to be used for calculating the quantity of the item, he can do so by simply changing the type of feature used to represent the quantity of the item.

Upon closer examination of the structure of the Bill of Quantities, it resembles the same structure used to represent the model described in Chapters 2 and 3. Whereas a *Building* object consisted of different *BuildingElements*, similarly a *BillOfQuantities* object consists of several *BoQElement* objects. The child-parent relationships used between the different *BuildingElement* objects can now be used to categorise the BoQ items into different subheadings, headings, bills and sections. The family concept can be used to differentiate between *Item* objects, *Subheading* objects, *Heading* objects, *Bill* objects and *Section* objects; and to define the standard features of each.

Figure 15 shows how the concepts of features, families and child-parent relationships can be used to create the Model part of the BoQ program. The basic structure is the same as the one displayed in Figure 7, with the *Building* object replaced by a *BillOfQuantities* object and the *BuildingElement* objects replaced by *Element* objects. A *BillOfQuanities* object consists of *Family* objects – "Section", "Bill", "Heading", "Subheading" and "Item". It also consists of *Element* objects, which can be a member of either of the above families. Each of the *Family* objects consists of a set of *Feature* objects, and if any of the *Element* objects is a member of a *Family*, it must have these types *Feature* objects in its feature set.[2] In order to avoid overcomplicating the figure, the child-parent relationships are not shown.[3] The object "Phase1", which is a member of *Family* object "Section", has the object "Bill1" as one of its children objects. "Bill1" again has the object "Excavations" as one of its children objects. This pattern repeats itself up to object "Trenches", which has no children objects. The object "Trenches" is a member of the "Item" family, which means it has a feature "quantity". If the user so desires, this feature can be changed at runtime.

The Bill of Quantities can therefore be constructed in two different ways: using a traditional object oriented modelling approach, with attributes and methods; or using the flexible modelling approach based *Feature* and *Family* objects. The latter approach, as per its design requirements, will bring more flexibility into the system and will enable the user to accomplish further customisation at runtime.

[2] See Section 2.5 for a detailed definition of the "member of" relationship between a *FeatureObject* and its *Family* object

[3] Child-parent relationships is defined in Section 3.2

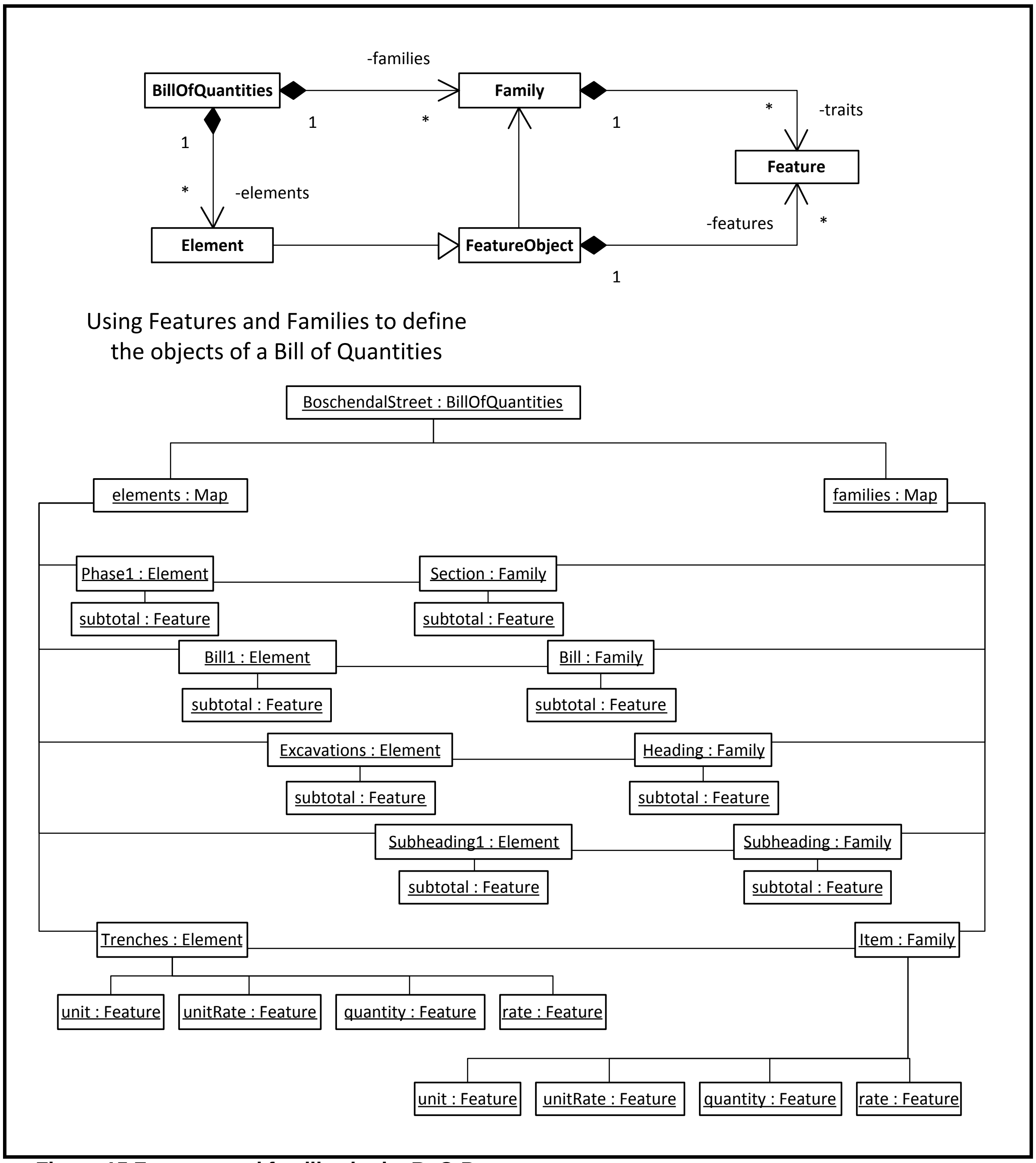


Figure 15 Features and families in the BoQ Program

4.2 Drawing application

In order to demonstrate the framework developed this thesis, a free CAD application called Cademia is used for the drawing part. Cademia is an open source 2D CAD application developed in Java (Cademia, n.d.).

Although different terms can be used to describe the lines and shapes that make up an electronic drawing, the term used by Cademia is components. Any line, rectangle, ellipse, polygon, dimension, etc. in a Cademia drawing is therefore a component entity. At the most basic level, a drawing consists of components.

Cademia, as with most other CAD applications, allows plugins to be created allowing for custom components to be defined. A *Component* interface is provided that all custom components must implement. It is quite useful to create custom *Component* objects that more directly represent real life counterparts. As an example, the normal *LineComponent* class can be extended to define objects that represent wall entities. As with the BoQ program it can be quite useful to introduce the concepts developed for the model described in Chapters 2 and 3 to the different drawing components.

In order to illustrate the concept of custom drawing components, a custom *Component* object was developed for the framework that can be used in a wide variety of situations. This *Component* object is used to group different existing *Component* objects, call it the *IntelligentComponent*. An *IntelligentComponent* object comprises a set of *Component* objects. When a user selects either of these objects, the *IntelligentComponent* object is selected, instead of the individual objects in the set. A command is used that adds existing *Component* objects to an *IntelligentComponent* object. A user can therefore load an existing electronic drawing in Cademia and then convert the drawing components to intelligent components. An *IntelligentComponent* is upon creation associated with a *CademiaElement* object, which is similar to *BuildingElement* and *BoQElement* objects. These objects are stored in a centralised *CademiaModel* object using child-parent concepts.[4] Using the family and feature concept combined with the child-parent concepts developed in Chapters 2 and 3, these *CademiaElement* objects can be classified according to the user's liking.

Figure 16 shows a structure quite similar to the one in Figure 7. The Building object is replaced by a Drawing object, which consists of CademiaElement objects.

[4] See Section 2.5 for a description of a *FeatureObject*, and Section 3.2 for a description of child-parent relationships

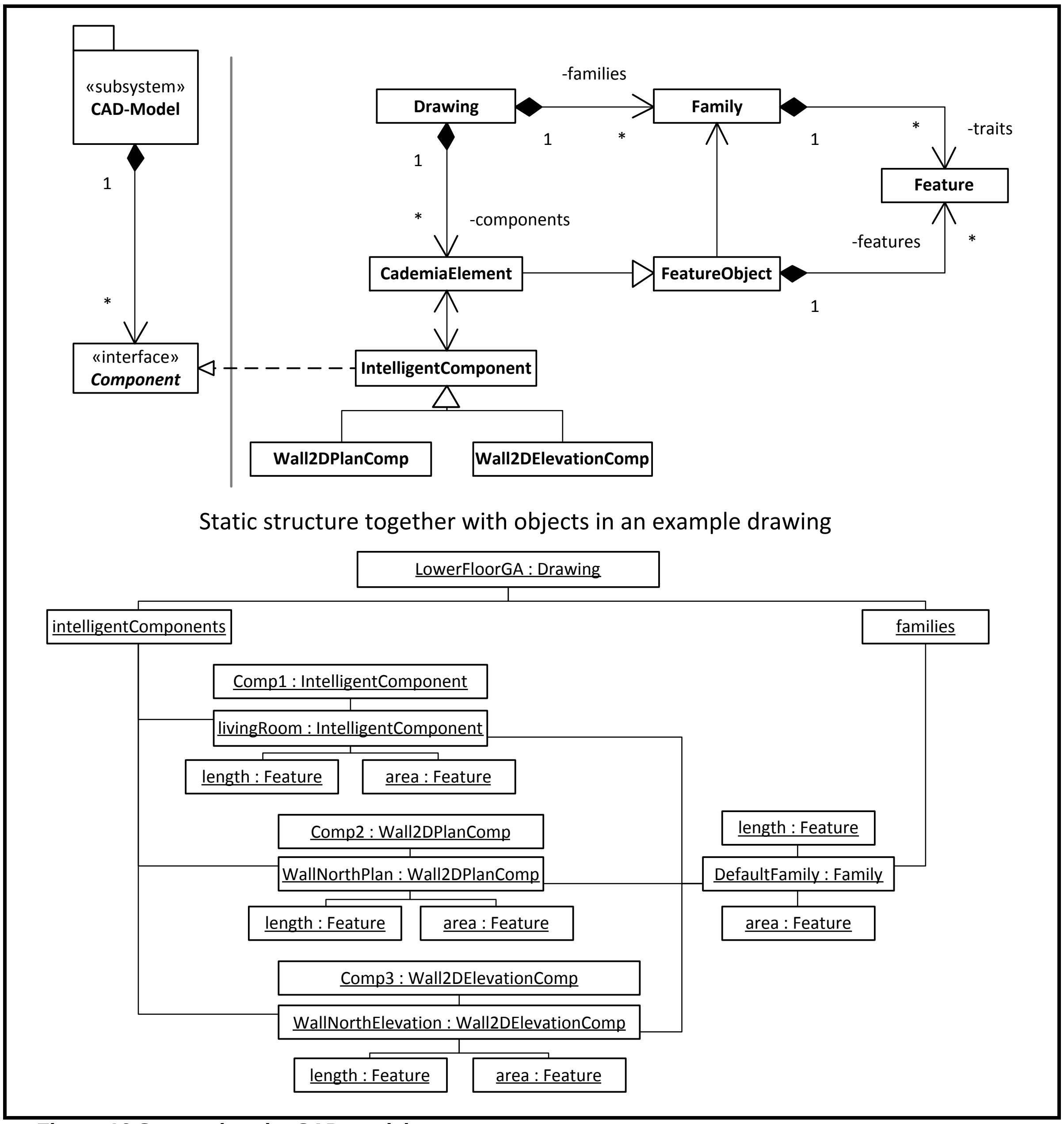


Figure 16 Structuring the CAD model

It should be pointed out that the CAD application still has its own internal model, indicated by the subsystem CAD-Model in Figure 16. *Component* objects contained in this structure has to be added to the *Drawing* object, in which case these *Component* objects will be contained in both the internal CAD Model as well as the *Drawing* object. The *Drawing* object only contains the *Component* objects that the user wants to link with other objects. The *CademiaElement* objects can be regarded as wrapper objects, wrapping objects from within the internal structure of an existing application. The wrapping process allows the different objects to be linked with objects from other applications.[5]

[5] Chapter 5 discusses how the objects are to be linked.

Since Cademia runs within the Java virtual machine, its objects can be directly accessed and used. This might not always be possible. In this case the wrapper objects will have to replicate the objects in the application. Systems will have to be put into place that ensures the wrapper objects are synchronised with the actual objects they replicate.

4.3 Third abstraction and the model concept

Before proceeding with the third abstraction process, the model concept has to be considered. In this chapter three different models are under discussion, while a fourth model is discussed in chapters 2 and 3.

The BoQ program has a model of its own. This model consists of a *BillOfQuantities* object, which consists of *Element* objects. This model is used to manage the pricing information of a project. The BillOfQuantities model is a supplementary model.

In addition to the BoQ program model two models are defined for the CAD application. The first is the internal model of the CAD application, the structure of which is not important. This model manages geometric information about the project. Another structure is defined to operate alongside the CAD model, call it the Drawing model. This model wraps the objects of the CAD model. It allows objects from other applications to be linked with the objects in the CAD model. In essence, the Drawing model rearranges the geometric information from the CAD model in such a way that it can be interpreted, used and modified by the other applications within the integration framework. Once this wrapper model is established, the internal CAD model is not directly used anymore and can be ignored. The Drawing model is also a supplementary model.

Finally, a core-model is discussed in chapters 2 and 3. Before discussing the purpose of this model, another type of model has to be described. The concept of a model can easily be confused with a 3-dimensional model created by an advanced 3D CAD application. These models typically do not manage only geometric information about a project. The idea behind BIM is to expand 3-dimensional models to include the information of all applications in the building environment. However, it is not necessary to have a 3-dimensional model as the basis for a building project. This is what the purpose of the core-model is. It serves as the basis model that connects the different supplementary models with each other. The elements of the Drawing model represent the shapes of real world entities. In contrast, the elements of the core-model are intended to directly represent real world entities. All supplementary information, including information in the Drawing model, can be reached through the core model.

Using the core-model as the basis model, different electronic drawings can be linked to it. This is useful since more than one construction drawing is normally used to depict a building element. This structure also allows different building elements to be represented by the same drawing element. This is useful since it is common practice to create one drawing element for a series of similar building elements.

The structure of the supplementary models is almost exactly the same as the structure of the core-model. Instead of redefining this structure for every supplementary, it should rather be defined as an abstract structure. Due to the similarities, the third abstraction process only involves renaming the objects from the original core model to more generic names.

The *Building* object from Figure 7 can better be described as a *Model* object. The *BuildingElement* objects are also specialised examples of *Element* objects. The objects of the supplementary models extend these objects and add attributes, methods and features required by their corresponding applications.

Figure 17 shows the structure of the framework after the third abstraction process. The top half displays the static structure, while the bottom half displays object instances of the three different *Model* classes, together with their standard sets of *Family* objects for each of these *Model* classes. Due to the high level of abstraction, the structure of the models does not have to be defined only in classes. Classes are useful to define methods and attributes used internally by the applications. The structures that directly represent real world structures should rather be mapped using the *Family* objects, together with their traits.

To better explain the difference between defining structure using classes and using Family objects, take the following two examples.

Example 1: Each of the elements has a unique id that is used internally by the application to manage data exchange between the different elements. This id is managed by the application and should never be changed by the user. The id is best defined using an attribute. Reason being that it is something that does not directly represent a real world property of an entity, and it is managed and used internally by the application.

Example 2: A user requires functionality that calculates the surface area of a wall. Initially, he requires this surface area to be calculated as the product of the length and height of the wall. The surface area is a property of a wall entity, therefore it is best represented in the model using a *Feature* object. By adding a surfaceArea *Feature* object to the Wall family's set of traits, the structure of the *Wall* objects is changed. *Wall* objects are now required to have a surfaceArea *Feature* object in its feature set. If the user wants to change how the surface area is calculated, the script of the surfaceArea *Feature* object can be changed at runtime. If the surface area was modelled using an object method, the source code would have had to be changed. Changing the script of a *Feature* object means changing the functionality of the model. Adding or removing *Feature* objects means changing the internal structure of the model. Therefore the functionality and the structure of a model can be changed at runtime, by the user himself.

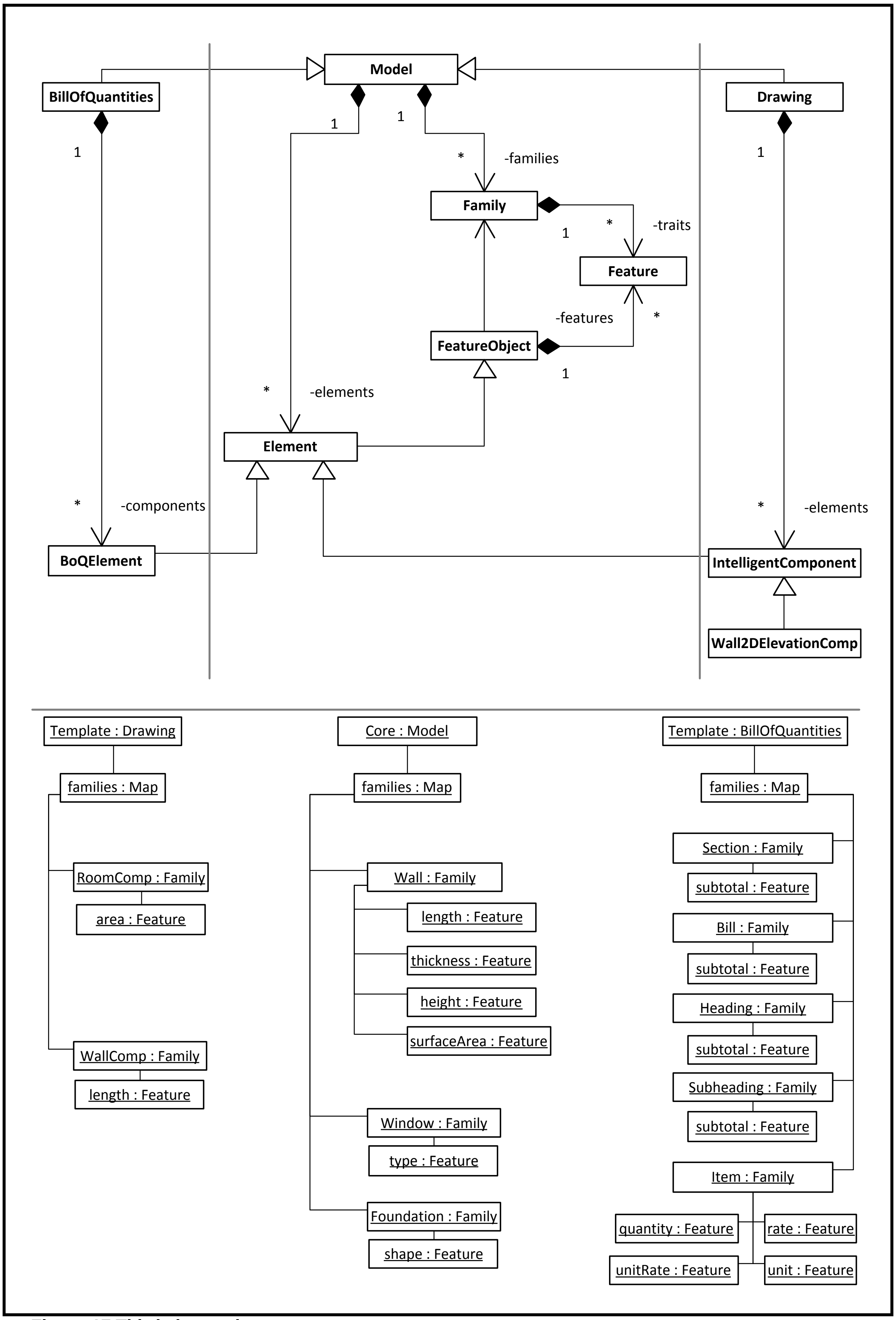


Figure 17 Third abstraction

4.4 Prototype source code of relevance to this chapter

The classes relevant to the BoQ program can be found in the package *BoQ*, while the classes relevant to linking the Cademia application can be found in the package *cademia*.

During the early stages of a project, the exact items and their unit rates and quantities are normally not known. The BoQ program was designed in such a way that the user can insert an estimated quantity and unit rate for each of the *BoQElement* objects. The user can select the subtotal for each *BoQElement* to be equal either to the product of the estimated quantity and unit rate, or to the sum its children's subtotals. A user can therefore start off by assigning estimated unit rates and quantities for the different sections. As the project progresses through its planning phases, estimated unit rates and quantities can be assigned to the bills. The user can then select the subtotal for the sections to be equal to the sum of the subtotals of its bills, which would at that stage respectively be equal to the sum of the estimated unit rates and quantities. At a later stage, this process will be repeated for the headings, subheadings and finally the items.

IntelligentComponent objects have methods getLength and getArea. The getLength-method returns an array of doubles containing the lengths of each of the Component objects it consists of. The getArea-method returns a double-array with the areas of the *Component* objects it consists of. A *CademiaElement* object is associated with one and only one *IntelligentComponent*. This component is stored as an attribute in the *CademiaElement* object. These objects have two *DerivedFeature* objects, length and area. By default the length is calculated as the average of the double returned by the getLength method of the associated *IntelligentComponent* object, though this script can be changed by the user. The area is calculated similarly.

The development of plugins for CAD programs usually involves defining new custom components according to the requirements of the user. For instance, a user might rather want to insert a "Wall" than physically drawing lines and shapes that represent walls. Whenever the "Wall" is inserted, the program readily inserts a *Component* object that resembles a wall. Although the *IntelligentComponent* can be used for a wide variety of situations, it is still useful to develop customised *Component* objects according to the user's requirements. The visual part of a component has to be defined in a way that the drawing application can interpret. This information therefore has to be defined using attributes and methods in a Component class. However, components often contain information other than the visual information. Examples of this information are methods to calculate its area or length. This information should preferably be defined using *Feature* objects in the component's associated *CademiaElement* object to allow users to customise it at a later stage.

5 Control, bringing everything together

Once supplementary models have been structured according to the guidelines of Chapter 4, elements of these models can be directly linked with each other. A central controller is required to manage these links, which forms part of the Controller portion of the MVC architecture. The Controller portion includes structures for a command system, managing unique identifiers, updating of derived features and database integration.

5.1 Controller: Class Workspace

Before being able to link various elements from the different models with each other, a class has to be established that controls the different models and their elements. From Figure 17 it becomes clear that the framework is designed to deal with many different models, each with their own elements. Furthermore, different applications will be accessing and modifying these models, each with its own user interface. A class is required that manages the different models and user interfaces. For this purpose, the *Workspace* class is created. In the sections that follow, functionalities that demonstrate the importance of this class will will be described.

Whereas different models can be dynamically created and destroyed during a session, a constant designated class is required where these models can be registered as they are created or destroyed. If a model has to be loaded or created, it has to be registered with this class. All classes must be able to reach this class, therefore it has to be centralised. For this reason, all the methods and attributes of the *Workspace* class are static. This allows the methods and attributes to be called directly on the class itself, instead of by reference to an instance of this class. An object therefore does not need a direct reference to a *Workspace* object to enable it to call the methods of this class.

5.2 Element associations

Once the structures from Chapter 4 have been established, the elements of the supplementary models can be linked directly with each other. This means that information can be exchanged directly between the different elements.

To demonstrate how elements can be linked directly with each other, take the example shown in Figure 18. Say the northern wall on the top floor of the building is represented by the core-model element "WallTN", which is a member of the "Wall" family. The Bill of Quantities has an item "Two-brick walls", and consequently the supplementary BoQModel has a "TwoBrickWall" element. Furthermore, the user added a *LineComponent* object in an electronic drawing to an *IntelligentComponent* object. This object is supposed to be a 2-dimensional plan-view representation of e.g. the northern wall of the top and lower floor. In the real world, a wall entity has properties length and height. The "WallTN" object therefore requires two features, length and height.

The quantity of the item "Two-brick walls" is based on the surface area of the wall, equal to the product of the length and height. The quantity of object "TwoBrickWalls" therefore depends on the length and height of object WallTN. The length of object WallTN must be retrieved from the "NorthWallsPlan" object. These dependencies are depicted with dashed lines in Figure 18. Clearly the objects NorthWallsPlan, WallTN and TwoBrickWall must somehow be associated with each other. Somehow therefore, *BoQElement* objects must be associated with core-model *Element* objects, which in turn must be associated with *IComponent* objects.

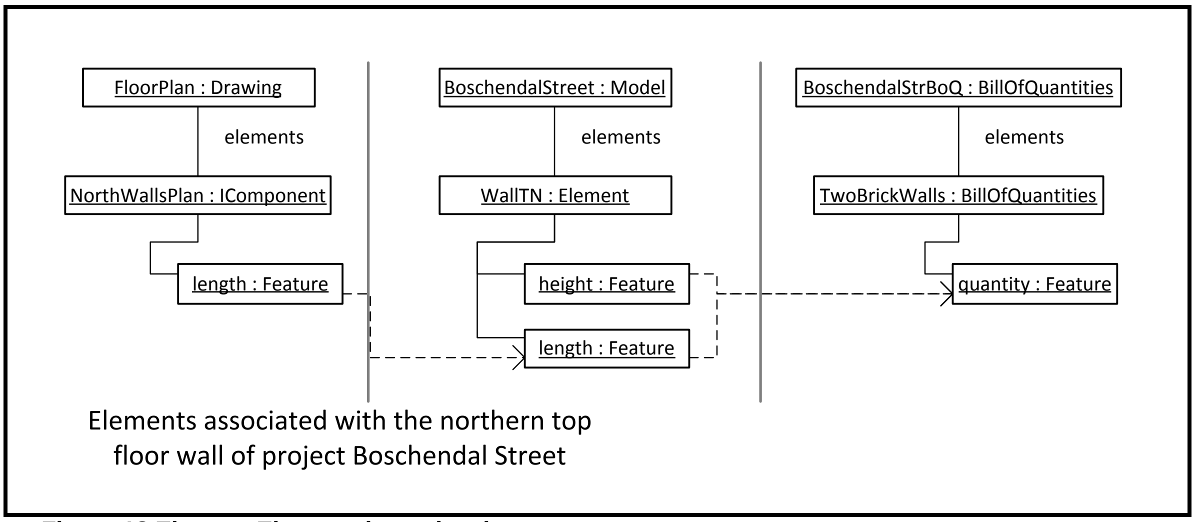

FloorPlan : Drawing
elements
NorthWallsPlan : IComponent
length : Feature
BoschendalStreet : Model
elements
WallTN : Element
height : Feature
length : Feature
BoschendalStrBoQ : BillOfQuantities
elements
TwoBrickWalls : BillOfQuantities
quantity : Feature
Elements associated with the northern top floor wall of project Boschendal Street

Figure 18 Element-Element dependencies

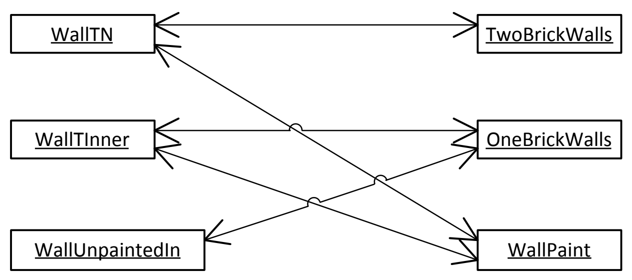
Due to the third abstraction process[6] these objects are all specialised types of *Element* objects. The *BoQElement-Element* association as well as the *Element-BoQElement* association can therefore all be described as *Element-Element* associations. A structure is therefore required that allows different elements to be associated with each other. This will not only cater for *Item-Element*, or *Element-BoQElement* associations, but for all *Element-Element* associations. This means that any *Element* object can be associated with any other *Element* object, which allows more flexibility than a structure that only cater for say an items and its associated core-model elements.

It is useful for the association between elements to be bidirectional. This, for example, allows a *BoQElement* object to reach an associated *Element* object, while at the same time

[6] See section 4.3

allowing the *Element* object to reach the *BoQElement* objects. The problem is that one *Element* object can be associated with more than one *BoQElement* object, while a *BoQElement* object can be associated with more than one *Element* object. Take for example the WallTN object. The TwoBrickWalls object is already associated with it; however a "Paint" item must also be associated with the same WallTN object. Furthermore, the WallTN object is not the only object that contributes to the quantity of the TwoBrickWall object.

As with child-parent relationships, due to the associations between different *Element* objects being bidirectional, inconsistencies can occur if not managed correctly. However, unlike one-to-many child-parent relationships, these associations are many-to-many relationships. They can therefore not be controlled using mappings. Consider the objects in Figure 19. The associations between them are indicated with arrows. Each object is also listed together with its associated objects below the diagram.



WallTN :	WallPaint, TwoBrickWalls
TwoBrickWalls :	WallTN
WallTInner :	OneBrickWalls, WallPaint
OneBrickWalls :	WallTInner, WallUnpaintedIn
WallUnpaintedIn :	OneBrickWalls
WallPaint :	WallTN, WallTInner

Figure 19 Many-to-many Element-Element relations

Adding an association or removing it is in every case a two step process. Say, WallTInner has to be associated with TwoBrickWalls instead of OneBrickWalls. OneBrickWalls can be removed from WallTInner's set of associated elements, and TwoBrickWalls can be added to it. However, OneBrickWalls will still have WallTInner in its set of associated elements, and TwoBrickWalls not.

Clearly then, if the association are not handled properly, inconsistencies can easily occur in the structure. As with child-parent relationships, these associations form part of the core structure and should be controlled in such a way that inconsistencies cannot occur. Similarly to child-parent relationships, an object has to be used that controls *Element-Element* associations. A map, however, can only control many-to-one relationships, not the many-to-many relationships required by these associations. A special class was created for this purpose – the *JunctionTable*.

A *JunctionTable* instance manages many-to-many associations between objects of the same type. Associations can be added or removed from a *JunctionTable* object. In both cases the *JunctionTable* object will ensure the different sets associated objects are synchronised.

Since one *JunctionTable* instance will have to control all *Element*-*Element* associations, this object has to be kept in a central location. The *Workspace* class therefore contains this *JunctionTable* object that controls all *Element*-*Element* associations.

5.3 Command system

An important programming practice, that supports Undo and Redo functionality, is that of the Command system. Instead of calling methods directly on the objects in an application's model, a *Command* object is created that performs all the required steps. A *Command* object has a doCommand method, which executes the methods that modify the model in an exact sequence.

As a demonstration, say the user wants to add a *Feature* object to the traits set of a *Family* object. Technically speaking all objects that are members of this family has to add this same feature to its feature set, otherwise they will not be members of the family anymore. Two steps have to be performed whenever a feature is added to a family, i.e. adding the feature to the family's trait set, and adding the feature to the feature sets of the members of the family. Somehow it must be ensured that these two steps are always performed when adding features to a family's traits. A *Command* object can be created that performs the two steps whenever its doCommand method is called. Instead of adding a feature directly to the objects in the model, a *Command* object is used that ensures all necessary steps are performed.

The steps described above can be built into the structure of the model, however in some cases it might be desired to add a feature to a family's traits without adding it to the members as well. If the structure ensures that the feature is added to its members as well, the feature will have to be removed from the members after adding it to the family's traits. A better practice therefore is to create a *Command* object for each case. It is not always clear whether to build a check into the structure or using *Command* objects to ensure all necessary steps are performed when the model is modified. In general, however, a more flexible system is achieved while checks are still maintained by using *Command* objects.

The command system furthermore allows undo and redo of actions. If a *Command* object implements an undo method in conjunction with a redo method, these methods can be called whenever a user wants to reverse a command, or re-execute a reversed command. The undo method must change the model back to the state it was before the command was executed. The redo method will only be called if the undo method has been called. It must ensure the system is changed back to the state it was in after the command was performed the first time.

In order for a sequence of commands to be undone, they must be stored in a command manager. This object has to be maintained for the duration of a session. For this reason the *Workspace* class controls this object and the commands that it executes. Class *Workspace* has a static method doCommand that receives a *Command* object. It then decides whether this *Command* can be executed and passes it on to its *CommandManager* attribute. It is therefore not necessary to directly have a reference to the *CommandManager* object since the *Workspace* class manages the execution of commands and this object internally.

5.4 Unique identifiers and a collaborative environment

To allow further flexibility into the framework, especially if the framework is to be used in a collaborative environment, the concept of unique identifiers has to be introduced. A unique identifier is assigned to an object to allow the object to be retrieved using only the value of the identifier. An identifier is usually in the form of a *String*, and in some cases represents the name of the object.

If an *Element* object can be reached using an identifier, it is not necessary for *Element-Element* associations to be made directly between two *Element* objects. The association can be made on the basis of their unique identifiers. In this way an element can be associated with another element, before either of them has been created. Upon creation, the first *Element* object does not necessarily have to call methods on the second. By the time it has to call methods on the second *Element* object, this object would have been created.

In a collaborative environment, in most cases it becomes unavoidable to have different versions of the same model within the team. Two or more members might be working on different parts of the same model. The result is that two or more versions of the same object can exist. In effect, each version of the object is an object instance of its own. If the different versions of the object can be reached by specifying the object's identifier, the different versions can easily be reconciled with each other.

In Java, *Strings* are immutable and unique in the sense that more than one instance of the same *String* cannot exist. For this reason, *String* identifiers are used in this framework. This would allow, for example, a user to add a feature to an *Element* object by specifying its identifier *String*. The user does not have to search through a list of *Elements* to be able to specify the *Element* object.

Ensuring that the assigned identifier *Strings* are unique can become hard. If a user has to assign names to each of the elements of a model, the user would soon run out of names. Furthermore, two different projects can both have a "Wall North", even though these two elements are not supposed to be the same objects. One way around this problem is to let the computer generate random *Strings* for identifiers. However, since these random *Strings* will be unrecognisable to the user. A user is unlikely to memorize random *Strings* to be able to reach an object quickly. To solve this problem, an algorithm is used to assign unique identifiers to objects that resemble a name. For example, the name generated for the first *Element* object, member of *Family* "Wall", added to the core-model for project "BoschendalStreet" is: "BochendalStreetWall1". The next *Element* object added would have the identifier *String*: "BoschendalStreetWall2". In this way the identifiers are unique and also recognisable by the user.

It is not only the *Element* objects that implement the concept of identifiers. *Model* and *Family* objects also implement this concept, allowing them to be identified by the user. This means that no two objects of type *Model*, *Family* or *Element* can share the same identifier *String*. By introducing the identifier concept to these objects, users can share and exchange information with each other without having to exchange complete models with each other. If one user wants to share a change in an *Element* object, he only has to send the updated *Element* object to the other user.

All objects that implement the identifier concept implements the *NamedObject* interface. These objects are all stored in the *Workspace* class, which means they can easily be reached by calling a static method on the *Workspace* class. The *Workspace* class associates the identifier with the object. In response certain commands, the *Workspace* will associate the identifier with another object. This new object might be an updated version of the previous object. When the original identifier is specified, the updated version of the objects will be returned. This means is that *NamedObject* objects must never be directly referenced in the source code; their identifiers should rather be used to reach them. This ensures that the correct version of the object is used at all times. For example, *Element* objects must not be directly inserted into the in the *Model* object's child-parent map. Instead, it contains the identifiers of the *Element* objects in the model.

5.5 Updatable derived features

In order to improve the efficiency of the framework, the concept of an *UpdatableDerivedFeature* object was introduced. This structure can compromise data integrity though – if used incorrectly.

UpdatableDerivedFeature objects are *DerivedFeature* objects that calculates its value and then stores it. Whenever the getValue method is called, the stored value is returned. Whenever one of the features or attributes it depends is changed, its stored value must be recalculated. To ensure that the stored value of these *Feature* objects remain valid, a special listener object listens for changes to the *Feature* objects that they are dependent on. As soon as a *Feature* object is changed, the listener is notified and the update method is called on the *Feature* objects that are influenced by this change. The sequence in which the *Feature* objects are updated is important. For this purpose another structure has to be built into the control system.

Consider the TwoBrickWalls object from Figure 18. Its quantity feature depends on the length and height features of the WallTN object. Graph theory can be used to model the dependencies between different features. A graph consists of a set of vertices and edges. A directed graph has directed edges, which are edges with a specific direction. The first part of Figure 20 repeats the *Element* objects and their *Feature* objects from Figure 19, the second part shows a directed graph. The directed graph consists of 4 vertices, a, b, c and d; and directed edges going from a to c, c to d, and b to d. It becomes clear that dependencies are similar to the directed edges of the graph.

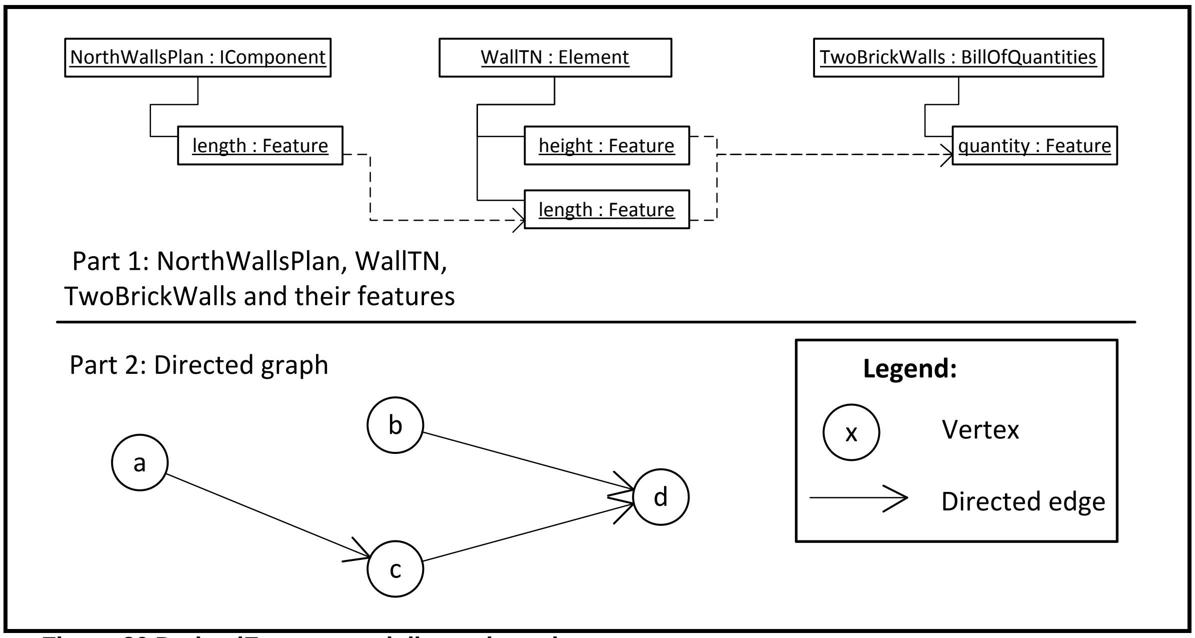


Figure 20 DerivedFeatures and directed graphs

When modelling the dependencies using graph theory, the vertices represent the *Feature* object and directed edges are used to represent the dependencies between *Feature* objects. In Figure 20, if the feature represented by vertex a is changed, the features represented by vertex c and d has to be updated in sequence. If the feature represented by d is updated before c, the value stored in that feature will be incorrect.

A path is formed by traversing from one vertex to another via directed edges. The length of a path equals the number of edges it traverses. A vertex is termed an ancestor of another vertex if there is a path going from it to the other vertex. A vertex is termed a direct ancestor of another if there is a path of length one from it to the other vertex. Vertex a is therefore an ancestor of vertex d and a direct ancestor of vertex c.

In order to determine the correct update sequence, the graph has to be topologically sorted. A topological sorting assign ranks to each of the vertices (Pahl & Damrath, 2001). In laymen's terms, the rank of a vertex is the length of the longest path to it, measured from a vertex without an ancestor. For more precise definitions, see Pahl & Damrath (2001). In effect the rank of a vertex corresponds to the step in the update sequence that its associated *Feature* object must be updated. Figure 21 shows a directed graph before and after it has been topologically sorted.

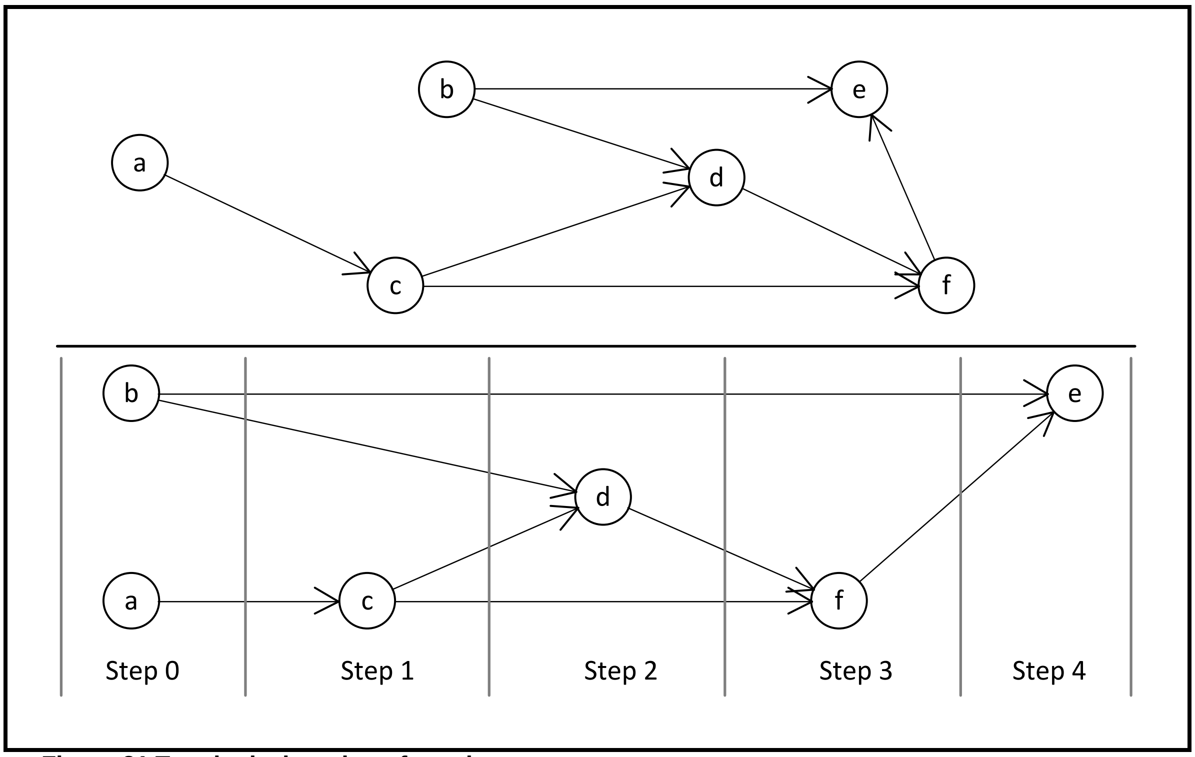


Figure 21 Topological sorting of graphs

Only acyclic graphs can be topologically sorted. Acyclic graphs are directed graphs that does not contain any cycles. In Figure 22 an example is shown where the user introduced a

Feature object to represent the cost per square meter of a wall. He then calculates the wall surface area by dividing the rate from the Bill of Quantities with this feature. The result is a cycle is formed, clearly seen between vertices e, b and d. Programmatically an infinite loop is formed.

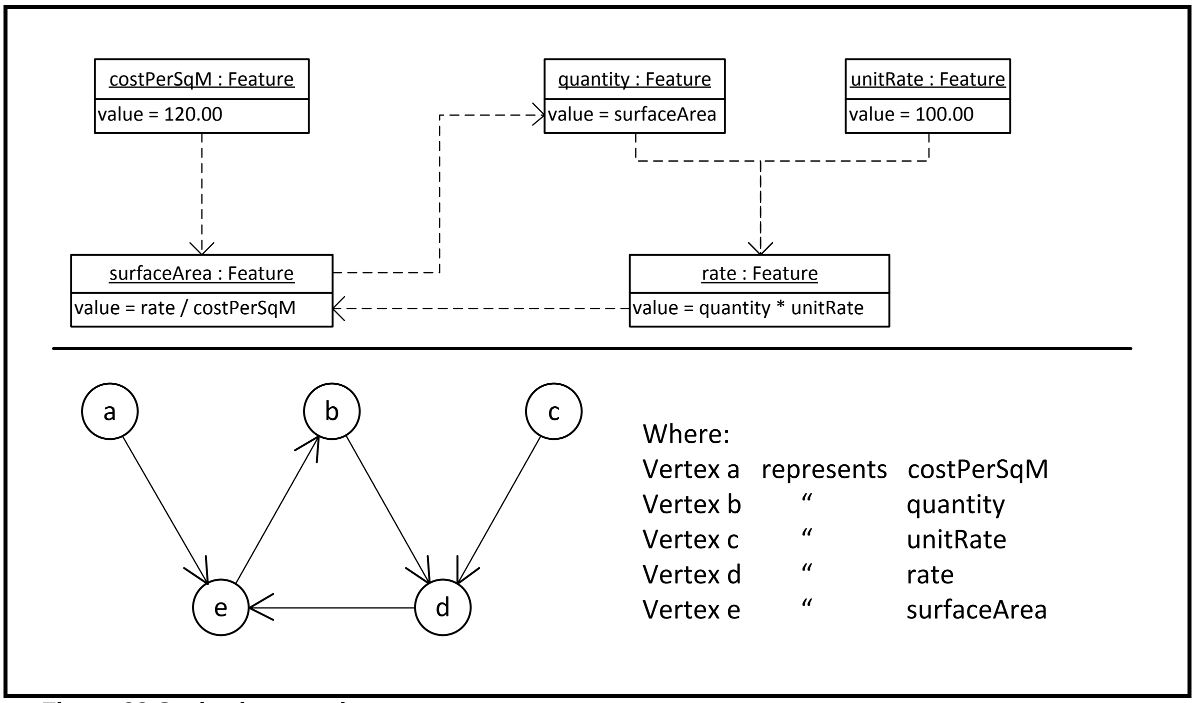


Figure 22 Cycles in an update sequence

The example in Figure 22 demonstrates bad programming procedures. The cost per square meter of a wall is similar to the unit rate. The same information is therefore stored twice, and the figure also demonstrates how an inconsistency is formed between the two features.

For the purpose of managing the updating of *Feature* objects, one central graph is required in the framework. This graph must contain vertices for every *Feature* object in every model, i.e. including the core model and all supplementary models connected with it. Whenever one of the *Feature* objects is updated, a sub-graph has to be created containing all the vertices that either directly or indirectly depends on the *Feature* object. This sub-graph has to be topologically sorted, and the dependent *Feature* objects updated in the correct sequence. The graph is managed by an *UpdateManager* object, located in the *Workspace* class. All *Feature* objects are observable. Whenever a *Feature* object's value is changed it will fire an event to all registered *Listener* objects. Whenever a *Feature* object is added the *UpdateManager* object has to be registered as a listener to this object. Furthermore, whenever a *DerivedFeature* object is added, a check first has to be performed to ensure that no cycles are formed due to the addition of the *DerivedFeature* object.

One of the advantages of using *UpdatableDerivedFeature* objects over the basic *DerivedFeature* objects of Section 3.1 is that cycles are detected before the object is added. This prevents the computer from going into an infinite loop when calculating the value of a *DerivedFeature*. If the value of the rate feature in Figure 22 were to be calculated, the computer would enter an infinite loop.

Another advantage is in the form of efficiency. Say feature a is dependent on feature b, which is dependent on feature c. With basic *DerivedFeature* objects, whenever the value of c is calculated, the getValue method of b will be called, which will calculate the value of b. When calculating the value of feature b, the getValue method of feature a will be called, which will recalculate the value of feature c. Thus the process of getting the value of feature c involves 3 calculation processes. With *UpdatableDerivedFeature* objects, the values are calculated and stored when a feature object is added or changed; from thereon end the getValue method merely returns the stored value.

The problem with *UpdatableDeriveFeature* objects is that listeners cannot be added to normal object attributes and *DerivedFeatures* can be dependent on these as well. This means an attribute can be changed, but since no event is fired for this change, the features that depend on it will not be updated. There are ways to solve this problem; however the programmer will have to consciously put a system in place to avoid inconsistencies. Such a system might involve declaring all attributes as private, with setter methods that fire update events.

Another problem with *UpdatableDerivedFeatures* is to detect its ancestors. Attempts were made to detect the ancestors directly from the BeanShell script; however the process became cumbersome and ineffective. The creator of the feature will therefore have to declare all ancestors. While the concept of ancestors can be easily grasped by programmers, the possibility exists that users will not declare ancestors correctly.

UpdatableDerivedFeatures can be used effectively by experienced users for features that have cumbersome derivations. However, the programmer in charge of customising the framework will have to ensure that the correct systems are in place to detect changes in attributes. Furthermore, if an algorithm can be created that can detect ancestors from the derived feature's script; this type of derived features can become more accessible to end-users.

5.6 Database integration

As projects of the case study are started and completed, the information in each must be stored in one database. Since the *Workspace* class stays constant, while different models are registered and unregistered; this class must be used to manage any database.

Databases are normally fully customised and influenced to a great extent by the individual requirements of a company. For the case study, a basic database was designed for demonstration purposes. The entities of this database can be used as a basis when creating a customised database. The data entities somewhat resemble the objects in the Bill of Quantities application, however there is a difference in the way the unit price and rate of items are stored.

Different projects and also different sections of the same project can have the same items; however the unit rate and quantity of an item will differ for each project and section. In the database, a Project entity consists of more than one Section entity. The relation between Item entities and Section entities is a many-to-many relation. An intersection table, table Item_Section, is used for this relation. Each entry in this table has a unitRate and quantity. An item can therefore be associated with more than one Section, and for each association a different unit rate and quantity is recorded.

An Item entity is associated with a Subheading entity, which is associated with a Heading entity. A Heading entity is associated with a Bill entity. Each of these relations is many-to-one relations. The association between Section entities and Bill entities should not be included in the database structure since it can be derived. The Item_Section table must rather be used to find the Item entities associated with a particular Section entity. By using the Subheading entity associated with an Item entity, and then using the Heading entity associated with the Subheading entity, and so forth; the Bill entity that contains the Item entity can be derived. Figure 23 illustrates the relations between the entities.

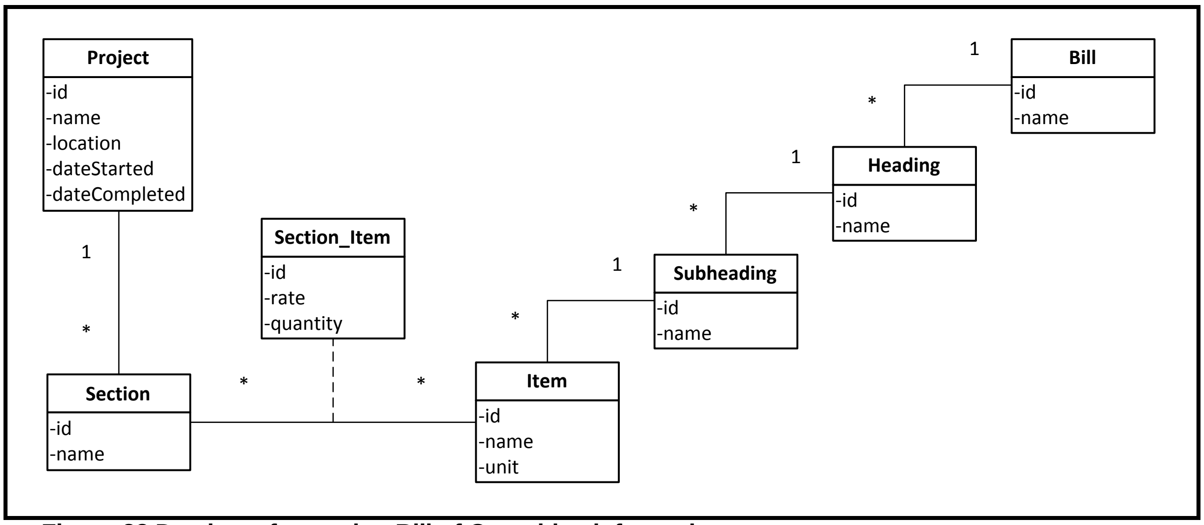


Figure 23 Database for storing Bill of Quantities information

5.7 Prototype source code of relevance to this chapter

The *Workspace* class is located in the package *control*. Class *JunctionTable* is located in the package *util* and Class *NamedObject* is located in the package *model*.

Since the software developed for this thesis is only a prototype implementation, a *CommandManager* class was not created. Provision was made for it in the *Workspace* class – all commands are executed using the executeCommand-method in the *Workspace* class. Currently this method directly executes the command by calling the doCommand method directly on the command. If a *CommandManager* is to be used, only this method should be changed to rather send the command to the *CommandManager* for execution than executing it directly. An existing *CommandManager* from one of the applications can readily be used. This would mean that one integrated command sequence will be managed for all the different applications by one *CommandManager*. This might not be desirable for the user, therefore Undo/Redo actions should be customised with thorough consideration of user requirements.

Different commands are located in the package *control.commands*. An important command, called *BuildingModelCreator*, is located in package customisation. When the user runs the prototype implementation, this command instantiates all the objects. Any command can execute several other commands, as illustrated by the *BuildingModelCreator* command. Most of the commands in the prototype implementation are demonstrated in the *BuildingModelCreator* class.

Another command of note is the command *CreateFamilyFromPath* located in package *control.commands*, which reads a text file and then creates a *Family* object with attributes and traits as specified in the file. This command allows users to define families in text files as classes are defined in .java files. For the correct format of the text contained in these files, refer to the package *BoQ.families*, which contains the files that define the families of the BoQ program.

A single *Feature* can also be created using a text file. The length and area features of the *CademiaElement* objects are declared in files "length" and "area", located in package *cademia.components*. The *BuildingModelCreator* command demonstrates how these files are used to create the features.

Class *UpdateManager*, located in package control, makes use of a graph implementation project called Plep2011 (Eygelaar, 2008). *UpdatableFeature* objects are added to this class, which will then check whether any cycles result from the *UpdatableFeature* object's addition to the update graph. The *UpdateManager* registers a listener to each of the *UpdatableFeature* object's predecessors. Predecessors can be of type *Feature*, or *FeatureObject*. The listener will react if a *Feature* object notified it of its value changing, and it will react if a *FeatureObject* object notified the listener that any of its attributes changed. The listener will react by updating all successors of the notifying *Feature* or *FeatureObject*.

For the link between a Java application and a Microsoft Access database file to function properly, certain drivers need to be setup first on the computer on which the software is executed. Since the prototype is meant to be executed on different computers, and also due to the high degree of customisation is involved in creating a database, this part of the framework was excluded from the prototype. However, on a Microsoft Windows computer, a Microsoft Access database file was successfully linked with a Java application using Microsoft ODBC. It could not be established whether this would be possible on a different platform.

For a database to be linked with a Java application, a Connection object is required. This object connects with a specified database, given the correct drivers are installed. In the case of linking a Java application with a Microsoft Access database, a JDBC driver is required. The Connection object is then used to execute normal SQL-queries on the database. Class *Workspace* provides a central location from where all objects of importance to the database can easily be accessed. It also provides a persistent location from where a database can be managed as different models of different projects are loaded.

6 Basic GUI

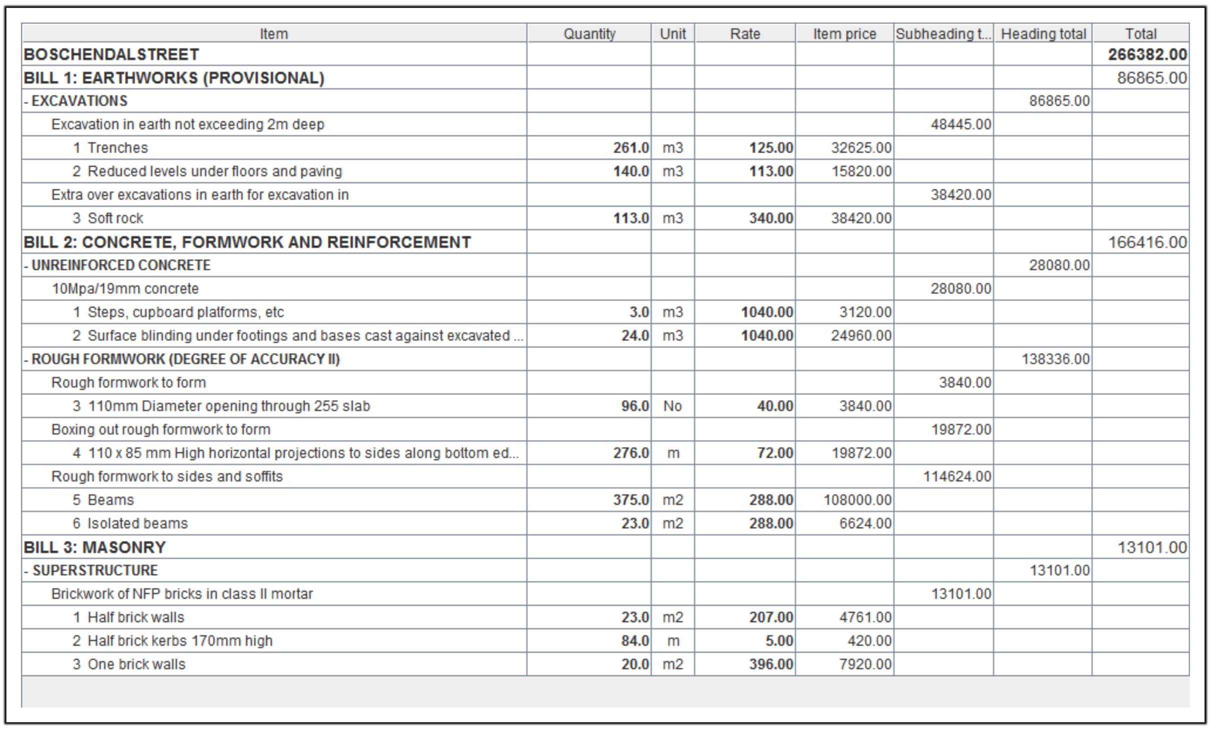
A basic view is discussed in this chapter that allows a user to view information of the different models. In addition, the view allows the user to directly access and modify the features in the feature objects contained in each model. In order to demonstrate the functionality of the view, a BoQ program based on the view is also discussed.

6.1 Basic view

Most of the functionality of the framework lies in integrating applications with each other. Integration occurs primarily on a "behind-the-scenes" basis. Once the framework has been customised by a programmer, the end-user will interact mainly with the applications. Special commands can be added to the applications in plug-in packages, which interact with the underlying core model. As an example, a command might be included in the CAD plug-in that creates a 2D representation of a wall consisting of two lines. This CAD application command can include core model commands that create and links a core model element, which is a member of the "Wall" family.

It is useful, however, to have a basic view of the underlying core model in order to directly control interactions between the different models and their elements. A CAD application is used to convert information into a visual form. Even though it displays lines, shapes, etc. on screen, the information used to create the display is still stored as numbers, text and Booleans in an information model. The view of the framework will have to display the vast amounts of information stored in the core-model and supplementary models.

The most common way to display large amounts of information is by using a table. Figure 24 shows a screenshot of a table displaying Bill of Quantity information. Users would easily grasp an interface based on a table, since it is the underlying structure of spreadsheets. Spreadsheets are widely used for technical information processing. Elements of a model can be displayed as rows in a table, with each column displaying a property of the element. A table can soon become overpopulated, resulting in a confusing number of rows. A tree can be used to reflect the child-parent relationships between the elements and can also assist to simplify the information displayed in the table. Figure 25 shows how a tree can be used to display the elements of a model and the child-parent relationships between them. A combination of a tree and a table would form an efficient basic view of the different models.



Item	Quantity	Unit	Rate	Item price	Subheading t...	Heading total	Total
BOSCHENDALSTREET							**266382.00**
BILL 1: EARTHWORKS (PROVISIONAL)							86865.00
- EXCAVATIONS						86865.00	
Excavation in earth not exceeding 2m deep					48445.00		
1 Trenches	**261.0**	m3	**125.00**	32625.00			
2 Reduced levels under floors and paving	**140.0**	m3	**113.00**	15820.00			
Extra over excavations in earth for excavation in					38420.00		
3 Soft rock	**113.0**	m3	**340.00**	38420.00			
BILL 2: CONCRETE, FORMWORK AND REINFORCEMENT							166416.00
- UNREINFORCED CONCRETE						28080.00	
10Mpa/19mm concrete					28080.00		
1 Steps, cupboard platforms, etc	**3.0**	m3	**1040.00**	3120.00			
2 Surface blinding under footings and bases cast against excavated ...	**24.0**	m3	**1040.00**	24960.00			
- ROUGH FORMWORK (DEGREE OF ACCURACY II)						138336.00	
Rough formwork to form					3840.00		
3 110mm Diameter opening through 255 slab	**96.0**	No	**40.00**	3840.00			
Boxing out rough formwork to form					19872.00		
4 110 x 85 mm High horizontal projections to sides along bottom ed...	**276.0**	m	**72.00**	19872.00			
Rough formwork to sides and soffits					114624.00		
5 Beams	**375.0**	m2	**288.00**	108000.00			
6 Isolated beams	**23.0**	m2	**288.00**	6624.00			
BILL 3: MASONRY							13101.00
- SUPERSTRUCTURE						13101.00	
Brickwork of NFP bricks in class II mortar					13101.00		
1 Half brick walls	**23.0**	m2	**207.00**	4761.00			
2 Half brick kerbs 170mm high	**84.0**	m	**5.00**	420.00			
3 One brick walls	**20.0**	m2	**396.00**	7920.00			

Figure 24 Bill of Quantities displayed in a JTable

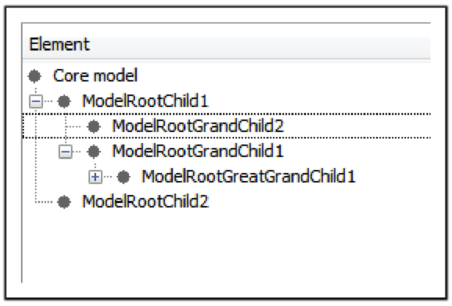
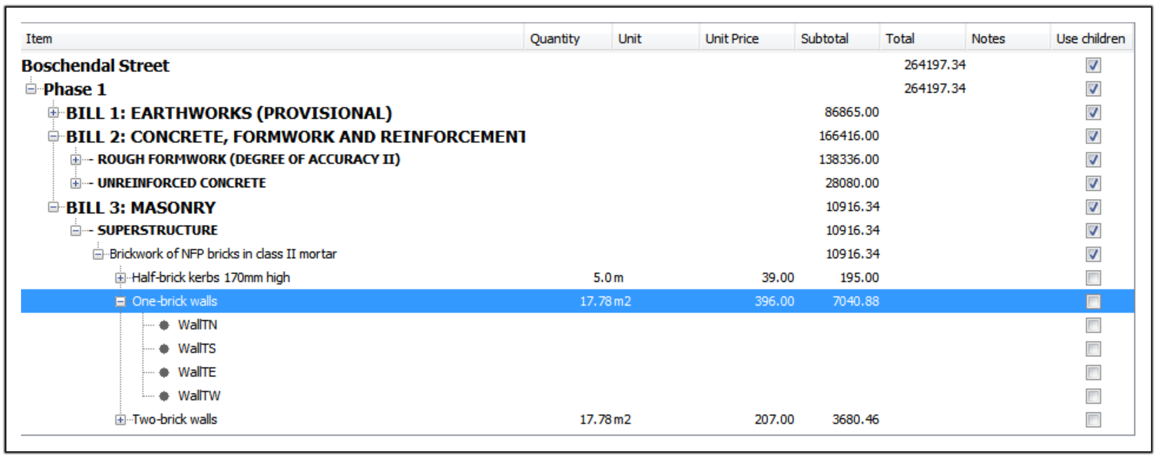


Figure 25 Screenshot of a basic JTree

6.2 Combining a tree with a table

In Java, *JTable* instances are used to display information in a table format, while *JTree* objects are used to display tree structures. A combination of the two, where the resulting view consists of a table as the basis, with the first column resembling a tree; would allow the user to expand and contract sets of rows. This concept is explored in a series of articles published by the Sun Developer Network entitled "Creating TreeTables in Swing" (Milne, 2003), "Creating TreeTables: Part 2" (Violet & Walrath, 2003) and "The Swing HTML Parser" (Violet, 2003). The combined viewer is termed a *JTreeTable*. Figure 26 shows how the Bill of Quantities table from Figure 24 can be simplified by contracting some of the rows.



Item	Quantity	Unit	Unit Price	Subtotal	Total	Notes	Use children
Boschendal Street					264197.34		☑
⊟ **Phase 1**					264197.34		☑
⊞ **BILL 1: EARTHWORKS (PROVISIONAL)**				86865.00			☑
⊟ **BILL 2: CONCRETE, FORMWORK AND REINFORCEMENT**				166416.00			☑
⊞ - **ROUGH FORMWORK (DEGREE OF ACCURACY II)**				138336.00			☑
⊞ - **UNREINFORCED CONCRETE**				28080.00			☑
⊟ **BILL 3: MASONRY**				10916.34			☑
⊟ - **SUPERSTRUCTURE**				10916.34			☑
⊟ Brickwork of NFP bricks in class II mortar				10916.34			☑
⊞ Half-brick kerbs 170mm high	5.0 m		39.00	195.00			☐
⊟ One-brick walls	17.78 m2		396.00	7040.88			☐
● WallTN							☐
● WallTS							☐
● WallTE							☐
● WallTW							☐
⊞ Two-brick walls	17.78 m2		207.00	3680.46			☐

Figure 26 Bill of Quantities displayed in a JTreeTable

As with all Java viewers, a *JTreeTable* viewer has its own model instance. A *JTreeTable* object has a *TreeTableModel* instance, which contains information such as the names of each column, the structure of the tree and the information for each cell of the table. A *JTreeTable* consists of *TreeNode* objects. The *TreeNode* objects are used to create the tree in the first column as well as populating the cells of the table. Each row uses a *TreeNode* object to populate the row's cells. By expanding a row, its children rows are displayed directly beneath it. Any object can implement the *TreeNode* interface. A *TreeNode* object must have a method that returns its children objects, a method returning its parent object and a method that returns whether it is a leaf or not. A leaf is a *TreeNode* object with no children.

As mentioned before, the framework consists of different models. For example, the screenshot in Figure 26 is a view of the elements of the *BillOfQuantities* model. The elements of this model contain all the information necessary to create the tree of a *JTreeTable*, as well as populating the cells of the table part. In some ways therefore it resembles a *TreeTableModel*; however it only contains some of the information of a *TreeTableModel* instance. It does not contain information such as the size of each column or row, the names of the columns and rows, etc. This information is specific to the *JTreeTable* viewer, and not the rest of the framework. It is therefore preferable to keep a viewer's model instance separate from the core-model and participating models; whether it be a *TreeTableModel*, *TreeModel*, *ListModel*, etc. Furthermore, it would be advantageous to enable more than one viewer to view the contents of a model. Each must be able to manage its own viewer specific information, such as column name, row size, etc. On the other hand, the different elements in the model readily implement the structure and methods required of *TreeNode* objects. If these elements are directly used as the *TreeNode* objects of a

TreeTableModel, information is not duplicated. The *Element* class therefore implements the *TreeNode* interface to make this possible. A *JTreeTable* viewer of the different families can also be useful, therefore the *Family* objects also implement the *TreeNode* interface.

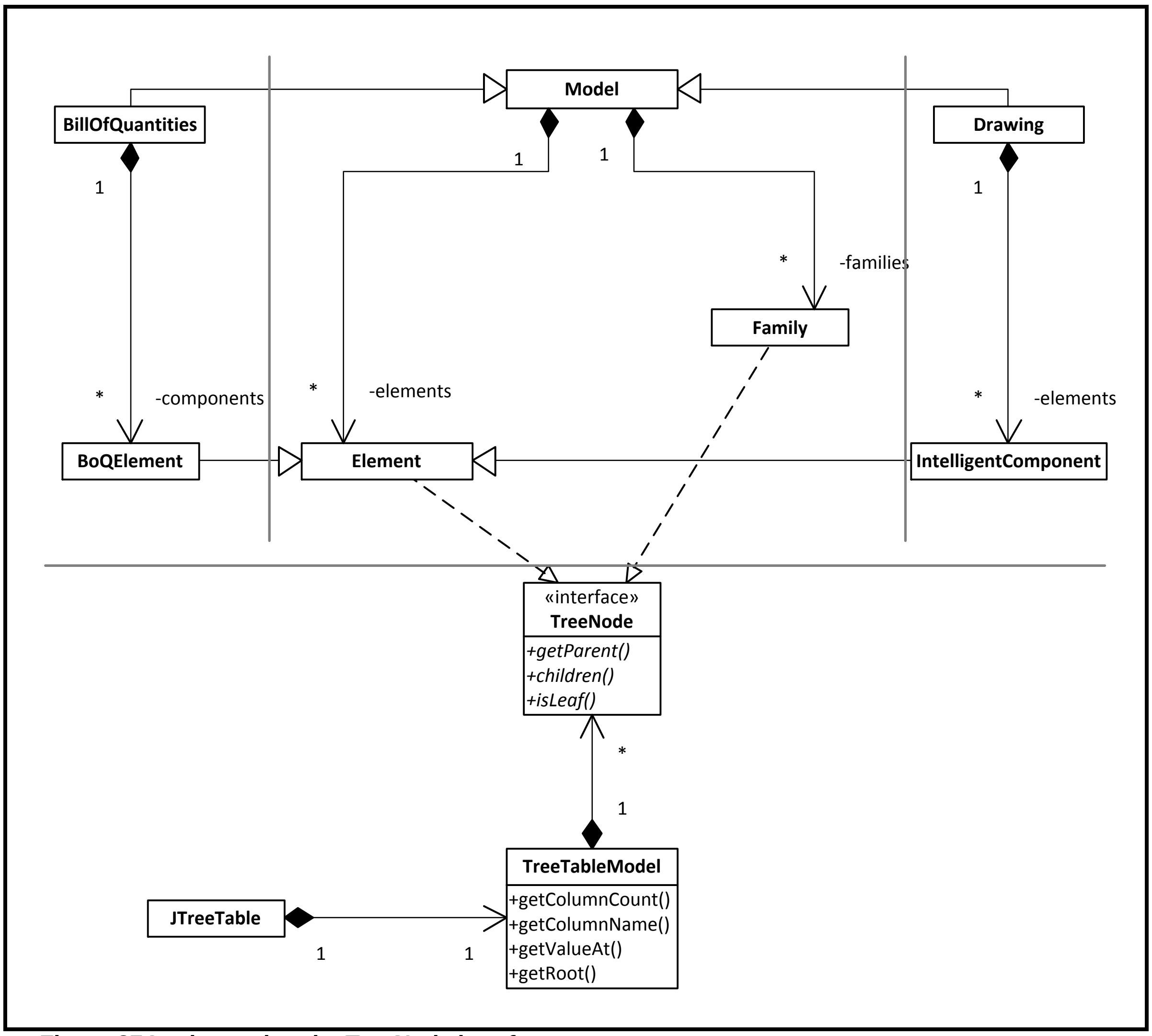


Figure 27 Implementing the TreeNode interface

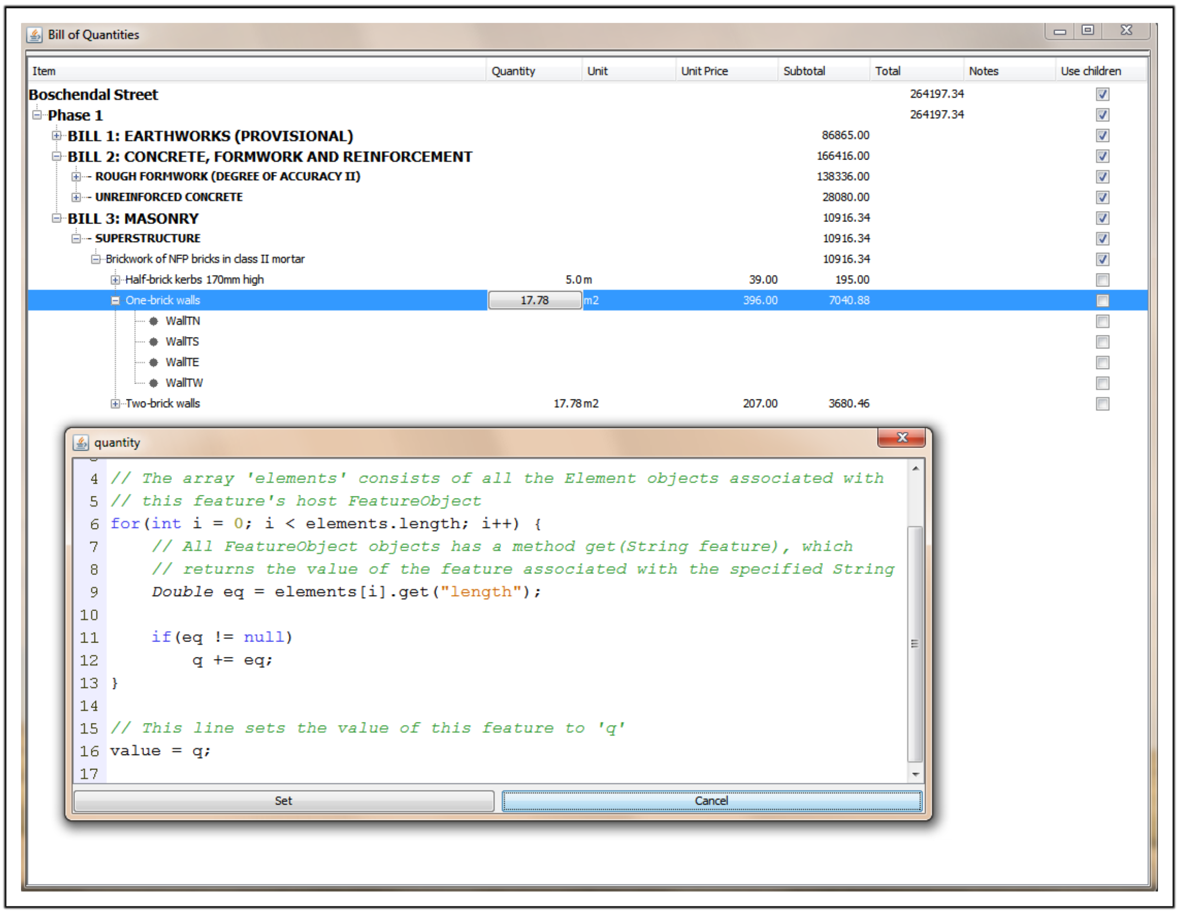
A special implementation of the *TreeTableModel* was created that allows columns to be added dynamically. When adding a column, the name of a feature must be specified. Each row will then display in this column, the value of its feature for the *Element* the row represents. Since *BoQElement* and *IntelligentComponent* objects extend the class *Element*, these objects can readily be used as *TreeNode* objects in this dynamic *TreeTableModel* object.

6.3 Bill of Quantities demonstration

In order to demonstrate the functionalities of the *TreeTable* concept, the viewer of the Bill of Quantities application is based on a *JTreeTable*. A Bill of Quantities is normally displayed in table format and therefore the *JTreeTable* concept can be used to great effect.

Figure 28 shows a screenshot of an example Bill of Quantities. When the a cell in the Quantity column is clicked, a script editor dialog opens, that allows the user to edit the script of the Quantity feature of the selected *BoQElement* object.

The screenshot shows item "One brick walls" selected and its script is displayed in the editor window. The script calculates the quantity as the sum of the Length features in an array called elements. This array is predefined and contains the *Element* objects associated with the item, namely WallTN, WallTS, WallTE and WallTW. These objects are displayed as the *BoQElement* object's leaves.



Item	Quantity	Unit	Unit Price	Subtotal	Total	Notes	Use children
Boschendal Street					264197.34		✓
Phase 1					264197.34		✓
BILL 1: EARTHWORKS (PROVISIONAL)				86865.00			✓
BILL 2: CONCRETE, FORMWORK AND REINFORCEMENT				166416.00			✓
- ROUGH FORMWORK (DEGREE OF ACCURACY II)				138336.00			✓
- UNREINFORCED CONCRETE				28080.00			✓
BILL 3: MASONRY				10916.34			✓
- SUPERSTRUCTURE				10916.34			✓
Brickwork of NFP bricks in class II mortar				10916.34			✓
Half-brick kerbs 170mm high	5.0	m	39.00	195.00			☐
One-brick walls	17.78	m2	396.00	7040.88			☐
WallTN							☐
WallTS							☐
WallTE							☐
WallTW							☐
Two-brick walls	17.78	m2	207.00	3680.46			☐

```
// The array 'elements' consists of all the Element objects associated with
// this feature's host FeatureObject
for(int i = 0; i < elements.length; i++) {
    // All FeatureObject objects has a method get(String feature), which
    // returns the value of the feature associated with the specified String
    Double eq = elements[i].get("length");

    if(eq != null)
        q += eq;
}

// This line sets the value of this feature to 'q'
value = q;

```

Figure 28 Script editor screenshot

The user can change the script of the quantity feature to be equal to the surface area of the item's associated walls. Only one of the lines of code has to be modified. Originally this line reads:

```
    q += elements[i].get("length");
```

This line must be modified to:

```
    q += elements[i].get("length") * elements[i].get("height");
```

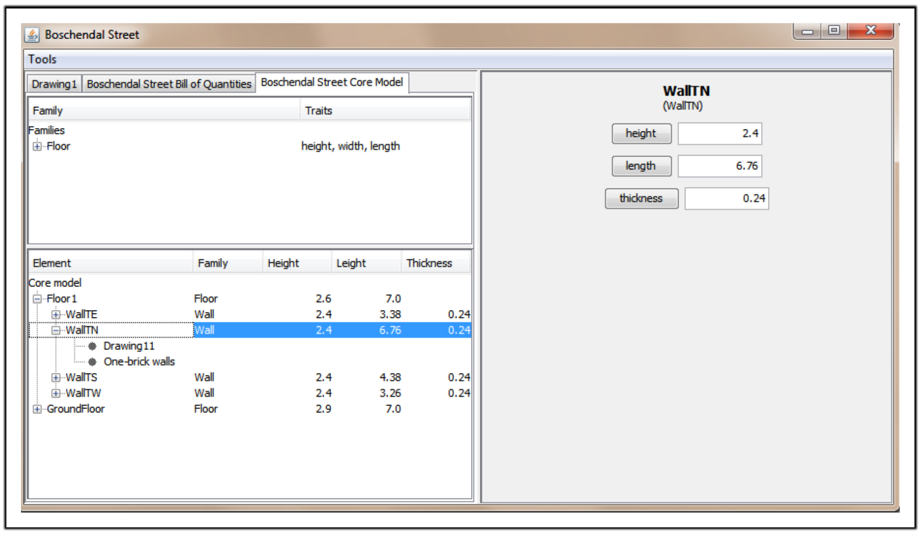
At a later stage, the user might require the surface area of all openings in the wall, such as window and door openings, to be subtracted from the quantity. Depending on how these openings were incorporated in the structure of the models, this change can be added as seamlessly as demonstrated above.

The viewer displayed in Figure 28 is an instance of object type *ModelPanel*, which is a subtype of Class *JPanel*. A *ModelPanel* is created by simply specifying the *Model* object that it must display and the required columns of the table. This object can then be added to any *Container* as desired.

6.4 Main model viewer

In addition to the windows of the applications being integrated with each other, an additional window was developed to provide functionality for directly viewing and manipulating the contents of the core-model as well as the supplementary models. This window is an instance of the class *IntegrationFrame*.

The content pane of the *IntegrationFrame* is divided into two areas – a main area with a tabbed pane for viewing the different models, and an auxiliary panel for viewing more specific information of selected objects and also where wizards are executed. The following two screenshots of the *IntegrationFrame* shows some its functionality.


Boschendal Street
Tools
Drawing1
Boschendal Street Bill of Quantities
Boschendal Street Core Model

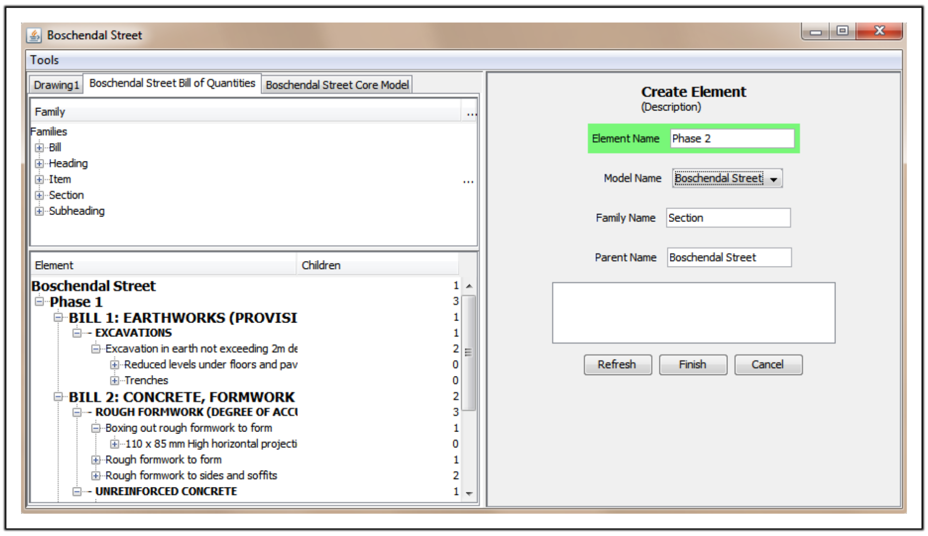
Family	Traits
Families	
Floor	height, width, length

Element	Family	Height	Leight	Thickness
Core model				
Floor1	Floor	2.6	7.0	
WallTE	Wall	2.4	3.38	0.24
WallTN	Wall	2.4	6.76	0.24
Drawing11				
One-brick walls				
WallTS	Wall	2.4	4.38	0.24
WallTW	Wall	2.4	3.26	0.24
GroundFloor	Floor	2.9	7.0	

WallTN
(WallTN)

height	2.4
length	6.76
thickness	0.24

Figure 29 Screenshot of the IntegrationFrame


Boschendal Street
Tools
Drawing1
Boschendal Street Bill of Quantities
Boschendal Street Core Model
Refresh
Finish
Cancel

Family

Families
- Bill
- Heading
- Item
- Section
- Subheading

Element	Children
Boschendal Street	1
Phase 1	3
BILL 1: EARTHWORKS (PROVISI	1
- EXCAVATIONS	1
Excavation in earth not exceeding 2m de	2
Reduced levels under floors and pav	0
Trenches	0
BILL 2: CONCRETE, FORMWORK	2
- ROUGH FORMWORK (DEGREE OF ACCU	3
Boxing out rough formwork to form	1
110 x 85 mm High horizontal projecti	0
Rough formwork to form	1
Rough formwork to sides and soffits	2
- UNREINFORCED CONCRETE	1

Create Element
(Description)

Element Name: Phase 2
Model Name: Boschendal Street
Family Name: Section
Parent Name: Boschendal Street

Figure 30 Screenshot showing an active wizard

Figure 29 shows the model "Boschendal Street Core Model" being displayed in the main area. The element "WallTN" is selected and its features are displayed in the auxillary panel. The element is associated with elements "Drawing11" and "One-brick walls", which are displayed as its leave objects.

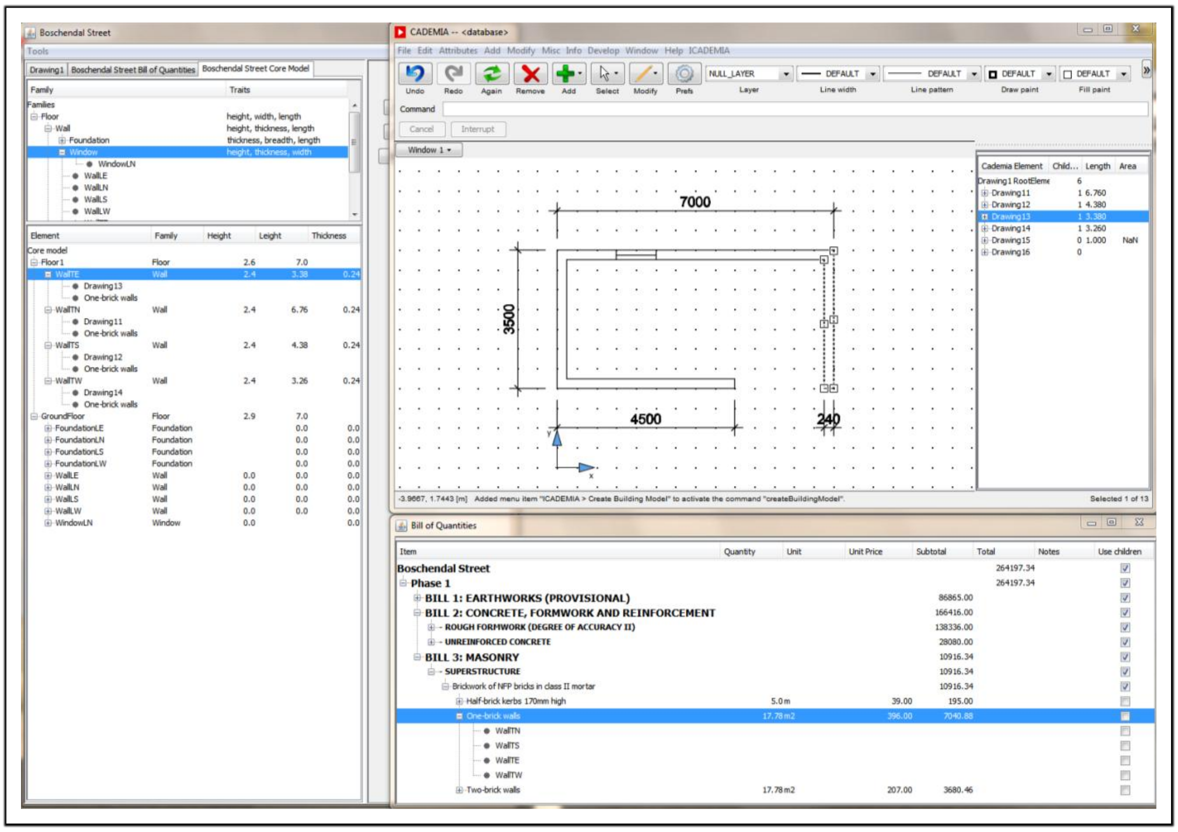
In Figure 30, the model "Boschendal Street Bill of Quantities" is displayed in the main area. As can be seen, the *ModelPanel* displayed in this time differs from the one displayed in Figure 28 – it has fewer columns and it also displays the different families in the model. Essentially, the *ModelPanel* instance displayed in the main area can be modified, at runtime, to include the the columns of the *ModelPanel* instance displayed in Figure 28.

The auxillary pane displayed in Figure 30 shows a wizard that will create an *Element* object according to the parameters specified by the user. A programmer can easily create custom wizards by extending the abstract class *ParameterAction*. Class *ParameterAction* is a subtype of the interface *javax.swing.AbstractAction*, which is used to create objects such as buttons, menu items, etc. Among other information, an *Action* object specifies the name of the button or menu item, and also what must happen when it is pressed. In other words, when a programmer creates a custom *ParameterAction* object, this object can be used to create a button. Whenever this button is pressed, the application will automatically create a wizard according to the parameters specified by the custom *ParameterAction* object, and when "Finish" button on the wizard is pressed, the *ParameterAction* object will be notified.

The wizard in Figure 30 displayed as a result of the "Create Element" menu item being pressed. This menu item was created using a *CreateElementAction* object, which is a subtype of class *ParameterAction*. The *CreateElementAction* object specifies that it requires four parameters. The first is of type *String*, the other is of type *Model*, the third of type *Family* and the fourth of type *Element*. The application creates *ParameterRenderer* objects for each of these parameters according to their type. For example the *ParameterRenderer* for parameter of type *Model* comprises a combobox with the models in the workspace. Once again a programmer can create custom *ParameterRenderer* objects. He can then specify the object type this custom renderer must be used for with the method setDefaultRederer in class *ParameterWizard*.

Figure 31 shows three different windows next to each other. To the left is the *IntegrationFrame*, with the Cademia application in the upper right portion and the BoQ program in the lower right portion. The Cademia application has a toolbar on the right-hand side that contains a *ModelPanel* instance showing the elements of the Cademia supplimentary model. Whenever an *IntelligentComponent* object is selected in the drawing pane, its associated *DrawingElement* object will be selected in the toolbar. Selection is synchronised by default between all *ModelPanel* objects, meaning if an *Element* object is selected in one, the same *Element* object will be selected in all other *ModelPanel* instances containing the object. Furthermore, if there are any *Element* objects associated with the selected one, these will be selected in the other *ModelPanel* instances. For example, as soon as the appriate component is selected in the Cademia drawing pane, "Drawing13" *CademiaElement* will be selected in all *ModelPanel* instances that displays Cademia model. Furthermore, its associated *Element* object, "WallTE" will also be selected in the *ModelPanel* of the *IntegrationFrame*. The same happens vice versa, selecting *Element* "WallTE" will result in *Element* "Drawing13", resulting in its associated *IntelligentComponent* object also being selected in the Cademia drawing pane. The same is also valid between the *IntegrationFrame* and the BoQ program.

To further illustrate integration between the applications, drag-and-drop is possible between the different *ModelPanel* instances. For example, if a user drags *Element* "Drawing15" from the Cademia toolbar to element "WindowLN" in the main area of the *IntegrationFrame*, the two will be associated with each other. In other words, after the dragging and dropping, "WindowLN" will be associated with "Drawing13" and vice versa - ergo selecting one will result in the other being selected.



Family	Traits
Families	
Floor	height, width, length
Wall	height, thickness, length
Foundation	thickness, breadth, length
Window	height, thickness, width
WindowLN	
WallLE	
WallLN	
WallLS	
WallLW	

Element	Family	Height	Leight	Thickness
Core model				
Floor1	Floor	2.6	7.0	
WallTE	Wall	2.4	3.38	0.24
Drawing13				
One-brick walls				
WallTN	Wall	2.4	6.76	0.24
Drawing11				
One-brick walls				
WallTS	Wall	2.4	4.38	0.24
Drawing12				
One-brick walls				
WallTW	Wall	2.4	3.26	0.24
Drawing14				
One-brick walls				
GroundFloor	Floor	2.9	7.0	
FoundationLE	Foundation		0.0	0.0
FoundationLN	Foundation		0.0	0.0
FoundationLS	Foundation		0.0	0.0
FoundationLW	Foundation		0.0	0.0
WallLE	Wall	0.0	0.0	0.0
WallLN	Wall	0.0	0.0	0.0
WallLS	Wall	0.0	0.0	0.0
WallLW	Wall	0.0	0.0	0.0
WindowLN	Window	0.0		0.0

Cademia Element	Child...	Length	Area
Drawing1 RootEleme	6		
Drawing11	1	6.760	
Drawing12	1	4.380	
Drawing13	1	3.380	
Drawing14	1	3.260	
Drawing15	0	1.000	NaN
Drawing16	0		

-3.9667, 1.7443 [m]   Added menu item "ICADEMIA > Create Building Model" to activate the command "createBuildingModel".   Selected 1 of 13

Item	Quantity	Unit	Unit Price	Subtotal	Total	Notes	Use children
Boschendal Street					264197.34		✓
Phase 1					264197.34		✓
BILL 1: EARTHWORKS (PROVISIONAL)				86865.00			✓
BILL 2: CONCRETE, FORMWORK AND REINFORCEMENT				166416.00			✓
- **ROUGH FORMWORK (DEGREE OF ACCURACY II)**				138336.00			✓
- **UNREINFORCED CONCRETE**				28080.00			✓
BILL 3: MASONRY				10916.34			✓
- **SUPERSTRUCTURE**				10916.34			✓
Brickwork of NFP bricks in class II mortar				10916.34			✓
Half-brick kerbs 170mm high	5.0 m		39.00	195.00			☐
One-brick walls	17.78 m2		396.00	7040.88			☐
WallTN							☐
WallTS							☐
WallTE							☐
WallTW							☐
Two-brick walls	17.78 m2		207.00	3680.46			☐

Figure 31 Screenshot of the integrated applications next to each other

Selection is coordinated in class *Workspace*, and any object implementing *SelectionListener* interface will be notified of selection events once registered in class *Workspace*. The *ParameterRenderer* objects, for example, are also *SelectionListeners* and as soon as a selection event occurs react accordingly.

The *ModelPanel* class readily provides great functionality to a programmer wanting to integrate applications with each other. As the Cademia toolbar demonstrates, by incorporating it into the applications somehow, functionality for viewing and modifying internal model information is immediately at the fingertips of the user.

6.5 Prototype source code of relevance to this chapter

The package *view.treeTable* contains the classes used to create the *JTreeTable* and *ModelPanel* objects. The *IntegrationFrame* is contained in package view. The *CreateElementAction* class is located in package *view.actions*, which also contains class *ParameterAction*. Parameter renderer source code is located in package *view.wizard*.

Furthermore, the BoQ program is displayed in an instance of class *BoQFrame*, located in package *BoQ*.

A basic *JEditorKit* (JSyntaxPane, 2011) is used to display and edit scripts.

7 Results

A framework was created that can be used to model entities of a building project. Entities are modelled as *Element* objects that use features, instead of attributes and methods, to represent properties of real life entities. This means that attributes and methods can be used exclusively for application specific functions, which end-users do not have to interact with. This allows end-users to create and customise *Element* objects to model real-life entities according to unique requirements, without requiring knowledge of application specific attributes and methods. The end-user therefore only has to deal with the information relevant to his unique requirements. As a result models based on this framework are only as complex as the user requires, and significantly less complex than all-encompassing BIM-style models. These models are also flexible enough to allow customisation, even by the end-user at runtime.

The features of *Element* objects can contain scripts, which are sets of instructions that the computer must perform to calculate the feature's value. By integrating appropriate interpreters into the integration environment, the scripts can resemble source code in standard programming languages. As with standard features, these types of feature objects can be added or removed at runtime. This means that functionality can be added to the model at runtime, with no modification to the source code. End-users with basic programming skills are thereby enabled to customise the functionality of the model.

The objectives of the thesis were therefore fully met.[7]

For demonstration purposes, an implementation of the framework was created in Java to address the problems for an example case study.[8]

A basic application was created with which a Bill of Quantities can be managed. A plug-in was created for a CAD application that allows its drawing components to be linked to items in the Bill of Quantities application. Functionality was added that synchronises selection between the applications. A database was designed that stores the pricing information of completed projects contained in the Bill of Quantities application. It was also demonstrated how objects can execute queries directly on this database.

The deliverables required by the case study were therefore also met.

[7] Refer Section 1.7
[8] Refer Section 1.1

The addition of script features ensures data integrity between the different applications is maintained. Furthermore, a system was created that allows derived properties to be updated as soon as one of its dependents is changed. This ensures data is concurrent at all times.

8 Recommendations

While the objective of the thesis was to create a flexible, integrated modelling framework; some recommendations follow on how a model based on this framework should be implemented.

As mentioned before, the software environment must be customised for individual requirements; and the customisation process would normally involve creating a customised view. It is recommended that the view allows buttons, menus and toolbars to be added dynamically at runtime. To support this, the creation of buttons, menus and toolbars must be accomplished using commands. These commands can then be executed at runtime.

Furthermore, a type of macro command can be implemented to make further use of the script concept. The idea of the macro command would be to allow the user to create a custom command that executes a user-defined script. The customised view can allow menu items and buttons to be associated with this command, thus allowing the user to add personalised functionality to the view. These macro commands might require user input for some of its parameters. It is therefore recommended that the view incorporates a standard command line or wizard interface that will allow parameter values to be specified when executing macro commands. Also, standard scripts can be defined as predefined methods, similar to functions used in spreadsheets. An example of such a standard method would be calculating the average value of a specified feature in associated *Element* objects.

It is further recommended that all implementations operate on a client-server basis, where all models are located on a central server and is modified by commands from the client. This would allow different individuals in a collaborative environment to operate on one central model. The result is that duplication of data is minimised and therefore data integrity improved.

A client server approach also supports applications to be created for mobile devices. These mobile devices can be used on the building site to update the building model. A user can for instance mark certain building entities as complete while on site. These changes are then uploaded to the model on the server, which generates payment certificates accordingly. *Feature* objects can be created that allow photos captured on site to be attached to the different *Element* objects. While the prototype for the framework was created with collaboration in mind, this concept is not demonstrated in the prototype. Persistent identification allows the information of persistently identified objects to be exchanged. However, care has to be taken when exchanging feature objects to ensure the relations between the updatable features of Section 5.5 are managed correctly. When replacing one *Element* object with another, its *Feature* objects must be removed from the *UpdateManager* object and replaced with the new *Element* object's features.

Whenever models are defined, it is recommended that all properties of real life entities be modelled as *Feature* objects. This enables them to be easily modified for future purposes. Furthermore, the updating system described in Section 5.5 can be used to manage information in a more effective manner.

When saving the different models, it should be kept in mind that each model consists of two parts, the *Element* objects and the *Family* objects. The *Family* objects contain information about the structure of the customised models. The *Element* objects contain information relevant to each building project. Files for different projects will therefore mostly contain the *Element* objects of the models. A customisation file will contain information about the families, amongst other. The control part can be modified to ensure that child-parent relations between *Element* objects reflect those between the corresponding *Family* objects. Thereby, the child-parent structure of the *Element* objects is contained as part of the information contained in the *Family* objects.

Customisation is often achieved at the expense of standardisation. Standardisation is important to enable interoperability between individuals in the same company, as well as interoperability between different companies. For standardisation between different companies, a common core object structure will have to be utilised by all companies. Customisation for different companies can be achieved by means of plug-ins or extensions of this structure, together with the usual custom family definitions. However, interoperability between individuals of the same company requires a higher level of standardisation. It is recommended that individuals of the same company operate on applications that are similar at least on source code level. Individuals can still define custom *Family* objects.

Conclusion

An integrated modelling environment can be achieved without a complex all-encompassing building model. The applications used to create these complex models often provide functionality not required by smaller companies, thus overcomplicating the building model faced by the end-user. While these models can provide integrated solutions to larger companies with extensive software requirements, it is possible for smaller companies to create integrated modelling environments using more specialised applications.

In order to achieve integration of small-scale applications, the model-concept must be perceived in a different manner. While models were originally perceived as 2D, 3D or even 4D CAD models containing mostly geometric information; building models have grown to include information from a much wider spectrum. The 3D building components of the most common CAD models have grown to include ever-growing lists of additional information, such as pricing, scheduling, structural design information, etc. It is not necessary for the geometric components to be the basis of a building model; rather focus can be shifted away from this portion to the additional information associated with building models. Once this is achieved the applications that create and manage the additional information can be effectively integrated, along with the CAD applications, into a single multi-facetted model.

Furthermore, the perception must also be changed that building models are predefined and fixed. Flexible, dynamic building models, allowing customisation for unique requirements, can reduce complexity. Flexibility in a building model has the further advantage of reducing the effort required to incorporate future additions.

While customisation has a role in reducing complexity, the role of standardisation to assist in interoperability across company boundaries must not be disregarded. In order to accomplish standardisation, custom building models will have to be created in accordance to a single flexible integration framework.

Reference List

BeanShell, n.d. *What is BeanShell?.* [Online]
Available at: http://www.beanshell.org/intro.html
[Accessed 19 November 2011].

Bell, D., 2004. *UML basics: The class diagram.* [Online]
Available at:
http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/
[Accessed 15 September 2011].

Cademia, n.d. *http://www.cademia.org/frontend/index.php?folder_id=251.* [Online]
Available at: http://www.cademia.org/frontend/index.php?folder_id=251
[Accessed 23 November 2011].

Eckstein, R., 2007. *Java SE Application Design With MVC.* [Online]
Available at: http://www.oracle.com/technetwork/articles/javase/index-142890.html
[Accessed 22 November 2011].

Eygelaar, A. B., 2008. *Plep2011.* Stellenbosch: s.n.

Google Inc., n.d. *Google Online Dictionary.* [Online]
Available at:
http://www.google.com/webhp?hl=en#hl=en&gbv=2&q=abstraction&tbs=dfn:1&tbo=u&sa=X&ei=DZuzTt_ZHM258gOltu2QBQ&ved=0CCIQkQ4&bav=on.2,or.r_gc.r_pw.,cf.osb&fp=3f59574a404d4048&biw=1280&bih=814
[Accessed 4 November 2011].

Joint Building Contracts Committee, n.d. *About the JBCC.* [Online]
Available at: http://www.jbcc.co.za/about-the-jbcc.htm
[Accessed 23 November 2011].

Milne, P., 2003. *Creating TreeTables in Swing.* [Online]
Available at: http://java.sun.com/products/jfc/tsc/articles/treetable1/
[Accessed 27 November 2011].

Pahl, P. J. & Damrath, R., 2001. Structure of Graphs. In: *Mathematical foundations of computational engineering: a handbook.* Berlin; Heidelberg: Springer-Velag, pp. 538-583.

Violet, S., 2003. *The Swing HTML Parser.* [Online]
Available at: http://java.sun.com/products/jfc/tsc/articles/bookmarks/
[Accessed 27 November 2011].

Violet, S. & Walrath, K., 2003. *Creating TreeTables: Part 2.* [Online]
Available at: http://java.sun.com/products/jfc/tsc/articles/treetable2/index.html
[Accessed 27 11 2011].

Wikipedia, 2011. *Data Integrity.* [Online]
Available at: http://en.wikipedia.org/wiki/Data_integrity
[Accessed 11 November 2011].

Wikipedia, 2011. *Software Framework.* [Online]
Available at: http://en.wikipedia.org/wiki/Software_framework
[Accessed 11 November 2011].

Note: Secondary sources, such as Wikipedia, were used to ensure that the definition of terms that could have been arbitrarily defined is in line with industry norms.