

METHODS FOR MODELLING UNIT OPERATIONS IN CHEMICAL PROCESS SIMULATORS

Kyra Tijhuis

EWI
SOFTWARE ENGINEERING

EXAMINATION COMMITTEE
Luís Ferreira Pires
Louis van der Ham
Gertjan Koster

Abstract

In chemical process engineering an engineer cannot evaluate chemical process designs with physical models. Instead the designs are evaluated with the help of computer tools. These tools are called process simulators.

In these process simulators the process is modelled by using predefined unit operations (e.g. mixing, distillation, heating) from the library of the process simulator. However, not all possible unit operations are in this library. Therefore, there is an option for the engineer to define their own unit operations. Despite this option, most engineers find it difficult to use a self-defined unit, since they need programming knowledge to define their own unit operations. A solution to this is to develop a method and an accompanying tool that makes it easier for process engineers to define their own unit operations.

The purpose of this thesis is to develop the aforementioned method and tool. This is done by following a software development methodology called model-driven engineering. This approach aims to define models that make sense from the point of view of a user familiar with the domain. These models are then used to develop the software. In this case, this means the process engineer is able to understand the models that are used to develop the tool.

Contents

1. Introduction.....	5
1.1. Motivation	5
1.2. Objectives	6
1.3. Approach	6
1.4. Structure.....	7
2. Model-driven Engineering.....	8
2.1. Historical perspective	8
2.2. Modelling.....	9
2.3. Software Engineering	9
2.3.1. Metamodelling	9
2.3.2. Transformations	11
3. Process Engineering.....	12
3.1. Process Simulation	12
3.2. UniSim Design.....	13
3.3. Unit operation	14
4. Solution Design.....	15
4.1. Overview.....	15
4.2. Metamodel	15
4.3. Transformation.....	15
4.4. Importing in UniSim.....	15
4.4.1. User Unit Operation	16
4.4.2. Extension Unit Operation	17
4.4.3. Conclusion	17
5. Metamodel.....	19
5.1. Requirements	19
5.2. Final version.....	19
5.3. Possible improvements	21
6. Transformation.....	22
6.1. Overview.....	22
6.2. Closer look	22
6.3. Possible improvements	25
7. Case Study	26
7.1. Model	26

7.2.	Transformation.....	26
7.3.	Result.....	27
8.	Final remarks	30
8.1.	General conclusions	30
8.2.	Future work	30
	References.....	31
	Appendix A: Glossary.....	32
	Appendix B: Code from model	33
	Appendix C: Transformation template.....	35
	Appendix D: Generated dehumidifier	35
	Appendix E: User Manual	45
E1.	Installing EMT and XPand	45
E2.	Import tool in Eclipse.....	51
E3.	Configuring view in Eclipse	53
E4.	Using the tool	55
E5.	Coding Tips	60

1. Introduction

In this chapter the motivation, objectives, approach and structure of the rest of the report are described.

1.1. Motivation

Chemical process engineering focuses on the design, operation, control and optimisation of chemical processes. The design process of a chemical plant starts with an open ended problem, such as a set of experimental results or a customer need. A chemical engineer needs to find the best solution for this problem. He does this by developing and evaluating a number of possible designs.

In many fields of engineering, a scale model is used to evaluate the different solution designs. In the field of chemical process engineering this is not possible, because of the large number of constraints and factors that needs to be considered and the costs involved in building such a scale model. The constraints and factors can arise in many ways. For example, some constraints are fixed and invariable, such as those that arise from physical laws, government regulations and industry standards. Others are less rigid and can be weakened by the engineer in order to find the best design. Another important factor is the cost of the design. The design can be very efficient, but if it is too expensive, it will not be profitable.

Hence, when a design in chemical process engineering has to be tested, a manner of testing has to be used that can take a lot of different factors into account and is relatively inexpensive [1]. Therefore, when the design of for example a chemical plant has to be evaluated, this is done with the help of systematic computer-based methods. This is called process simulation. There are a large number of process simulator software suites available nowadays [2]. They all work on a similar basis, i.e. by using the laws of conservation of mass and energy.

A process is modelled by using predefined unit operations (e.g. distillation, condensation and mixing). Unit operations are the basic steps of a process and describe a physical or chemical change. In addition to the predefined unit operations, the process simulation software has information about the chemical and physical properties of pure components, of mixtures of components and of reactions between components. When all this information is combined, the computer can calculate what will happen when a process takes place. The engineer can then compare the outcomes of the different proposed solutions, which helps him choose the best design.

However, it is not always possible to model the design by using the predefined unit operations the process simulator provides. Sometimes, the design the engineer has made includes unit operations that differ from the ones offered by the simulator. The structure and dynamics of the unit operation then have to be specified by the engineer. Most software suites provide a method to implement these self-defined unit operations.

Nonetheless, for chemical engineers it is not easy to implement the self-defined unit operations in these software tools, because the methods provided by the software suites often require programming knowledge in a specific programming language. For example, the UniSim Design Suite by Honeywell requires knowledge of Visual Basic to define a unit operation [3]. This requirement causes an almost insurmountable obstacle for a large number of process engineers.

This hurdle could be overcome by devising a method and accompanying tools that makes it easier for engineers to define and import their own operation model, in any process simulator. This method should

- allow the engineer to use the terms they understand;
- allow the engineer to define the structure and dynamics in a way they understand;
- output a self-defined unit operation the process simulator understands;

Model Driven Engineering (MDE) is a software development methodology that aims at raising the level of abstraction in the specification of software systems. It also aims to increase automation in software development. Lastly, it aims to reach the two previous objectives by allowing the definition of modelling languages intended for the problem domain experts.

The idea behind MDE is to use models when developing software systems. The models have different levels of abstraction, which helps manage the complexity of the software system. The models are used to generate executable models. This is done by automated transformation of a model with a higher level of abstraction into a model with a lower level. These transformations continue until the point is reached where the model can be made executable by using some code generator [4].

For that reason, MDE seems to be a suitable methodology to obtain a method to support chemical process engineers in the way mentioned before. It can be used to generate executable code from models, in this case meaning that the process engineer only has to define a model, using terms and equations they understand, to obtain a self-defined unit operation.

1.2. Objectives

The objective of this BSc assignment is to develop a tool, using MDE, which makes it easier and quicker for a chemical process engineer to define and import a self-defined model in a process simulator. The tool should not require the engineer to have any programming knowledge. It should use terms and equations that the engineer understands. The output of the tool should be an operation model that can be used in the process simulator software.

1.3. Approach

To achieve the objective of this assignment, the following steps have been taken:

1. Performed a literature study of metamodeling and transformations in order to understand the theory behind MDE.
2. Produced the solution design in which all steps to get a working tool have been investigated.
3. Set up the development environment, which is needed in order to define the metamodel and transformation.
4. Defined the metamodel, which is used to specify a language to represent the model in a way that is understandable for process designers.
5. Defined the transformation that can be used to generate executable code from the model.
6. Performed a case study to show how the developed tool works to solve a real problem.

1.4. Structure

The remainder of this report is structured as follows:

Chapter 2 presents the information about MDE necessary to understand this work.

Chapter 3 gives the necessary information on chemical process engineering and process simulators.

Chapter 4 presents the solution design. The problem could be solved in a number of different manners. All these different manners are discussed and our choice is justified.

Chapter 5 discusses the defined metamodel.

Chapter 6 describes the transformation that is used to generate executable code.

Chapter 7 reports on the case study that has been performed to illustrate how a unit operation can be generated from a model.

Chapter 8 concludes this report and gives recommendations for further research.

2. Model Driven Engineering

This chapter reports on the conducted literature study, presenting the basic concepts on which MDE is based. Section 2.1 gives some historical perspective on MDE. Section 2.2 discusses the advantages of modelling. Section 2.3 addresses some MDE technologies that are relevant for this work.

2.1. Historical perspective

Every organisation consisting of more than one person has at some point realised that their information technology infrastructure is basically a distributed computing system, since it consists of multiple computers that communicate and interact with each other to achieve a common goal. To achieve this goal, it is important that information stored in the system can be used anywhere, anytime. Practically this means that the information has to be accessible from across departments, across the company, across the world, but also, more importantly, it has to be possible to access information across service- or supply-chain, from supplier to customer. This implies that every computing device must be a global information device, connecting to other information systems as easily as putting a plug into a power socket.

However, this was not always the vision of the software developers. For the larger part of the 20th century, developers thought of devices as standalone appliances, never needing repair or integration. Applications were built only to fulfil their functional specification, not taking into account any future integration or update. Most programmers assumed that the application they built would only be used in the next few years [5]. An example of this way of thinking is the Year 2000 problem (or Millennium bug). Before the turn of the century, it was common in programming to represent the year using only the last two digits. This meant that after the year (19)99 the year would be represented by 00. This problem was only widely recognized in the mid-Nineties, but even after its recognition it remained mostly unresolved until the last few years of the decade [6].

Yet even after this problem, most software still was written ignoring the constantly shifting infrastructure, changing requirements and new technologies. But not only software was affected by this problem. Most data warehouses contained the same data in different formats. If application and data integration was to be obtained, something had to change. As the writer John Donne wrote in 1624: “No man is an island” [7]. Now something had to be done so one could also say “no computing device or data warehouse is an island”.

The change came when the interest in modelling data and applications picked up significantly. The IT industry came to the conclusion that it was important to be able to integrate and update applications. They realised that there was no excuse to build software without first making a well-thought out design; it would lead to easier system development, integration and maintenance. Another advantage was that some of the construction of the application could be automated based on its design. The situation can be made clear by using the construction of a building as an example.

Before, software was built similarly to a house standing on its own. It would not be possible to connect it with others, change or even redecorate it, because it was built for one use only. It would be falling down constantly, generating work for construction workers, but an unacceptable environment for those who lived and worked in the houses. When modelling is used to develop an application, the house has a well considered blueprint, which even could be used to do some basics (foundation) of the construction. The resulting house could be connected with other houses, it could

be altered and redecorated, and it would still be standing at the end. Therefore, it seems only logical to use modelling to develop applications and data structures [5].

2.2. Modelling

The observations above led to the Model-Driven Architecture (MDA) initiative. MDA promises to allow the definition of machine-readable application and data models which allow long-term flexibility of:

- implementation: new technologies can be integrated or addressed by existing designs
- integration: since implementation and design both exist at the time of integration, the production of data bridges and the connection of infrastructures can be automated
- maintenance: the design that is available is in a machine-readable form, which allows direct access to the specification of the system, simplifying maintenance
- testing and simulation: since the models can be used to generate code, they can also be used to confirm requirements, test infrastructures and simulate the behaviour of the system being designed

As we mentioned before, models are used to reach these goals [5]. A model can be described as a purposely abstracted, clear, precise and unambiguous conception. The model denotation is a precise and unambiguous representation, in some appropriate formal or semi-formal language [8]. With this definition in mind a model can be characterised as follows:

- a concrete representation of something in the real world (some system);
- a simplification, containing only the relevant details ;
- has a specific purpose;
- defined in an appropriate textual or graphical language [9]

Concluding, MDA aims to improve some qualities in software systems, namely adaptability, portability and reusability by using models [5]. However, MDA is not a methodology; it prescribes no specific methods or principles. Therefore, any practical use of MDA requires the selection of a development process [4].

2.3. Software Engineering

MDE is the combination of MDA technologies such as metamodeling and transformations with software engineering processes [4].

MDE can be used to develop software, and focuses on creating domain models. These domain models should make sense from the point of view of a user familiar with the domain. These models then serve as a basis for the new system.

Some MDE technologies relevant for this study are described below.

2.3.1. Metamodeling

A metamodel is a model, as the name suggests. A metamodel is often called a model of a model. In MDE a metamodel is seen as a model of a modelling language [9]. More detailed: ‘a metamodel is a model of the conceptual foundation of a language, consisting of a set of basic concepts, and a set of rules determining the set of possible models denotable in that language’ [8]. A metamodel models all

possible members of a set of models. The constraints that restrict the possible models expressible are determined by the metamodel of the modelling language.

Fig. 1 shows that above the level of the metamodel (M_2) there is another level called metametamodel (M_3). A metametamodel models a metamodel, just as a metamodel models a model.

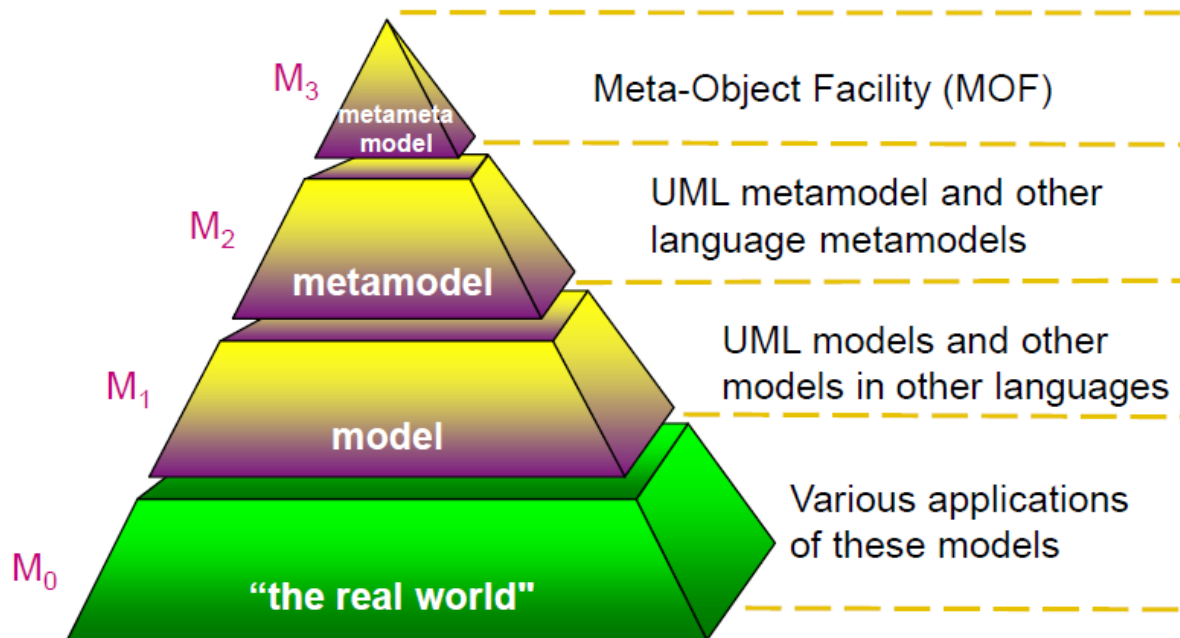


Fig. 1 The four levels of the Meta-Object Facility [9].

The same can be said about the modelling languages: the metametamodel models the language in which the metamodel is defined, the metamodel models the language in which the model is defined. Fig. 2 shows the relations between the different kinds of models.

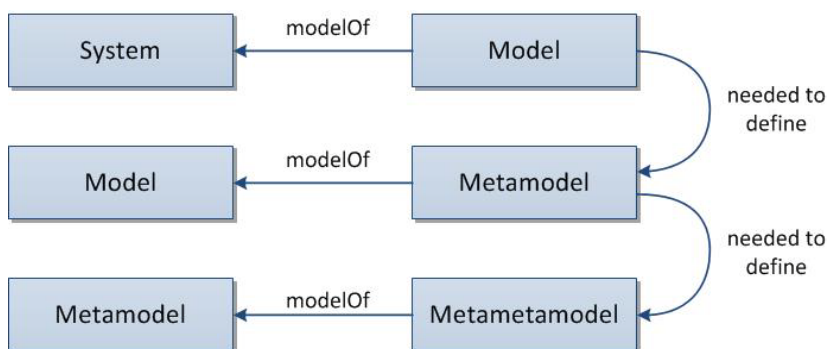


Fig. 2 The relations between the system, model, metamodel and metametamodel.

Therefore, to be able to define a metamodel, a metamodelling architecture is needed. An example of a metamodelling architecture is the Meta-Object Facility (MOF). MOF was developed to provide a metametamodel (the M_3 -layer) for the Unified Modelling Language (UML). UML is a general-purpose modelling language that is used in the field of object-oriented software engineering. It includes graphic notation techniques that help create visual models of object-oriented software system [12].

Here the second layer (M_2) is formed by UML itself. The M_2 -models are used to describe the M_1 -models, here the models written in UML. The last layer is the M_0 -layer, which contains the real-world objects [9].

Concluding, a metamodel is used to define a model, a metametamodel is needed to define a metamodel. Therefore, the first step in metamodelling is defining or choosing a metametamodel. After this the metamodel can be defined.

2.3.2. Transformations

After the model has been defined, it must be transformed into something executable [13]. To transform the model a transformation definition is needed. The general transformation pattern is shown in Fig. 3.

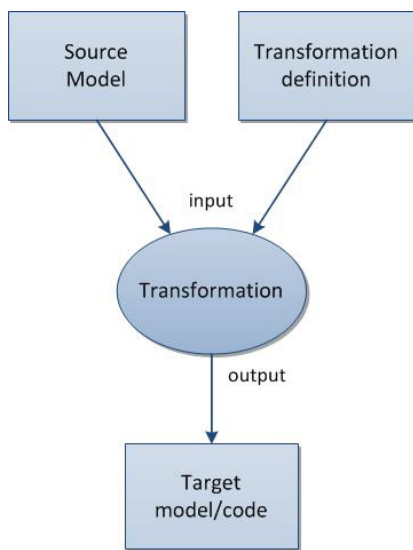


Fig. 3 The general transformation pattern [12].

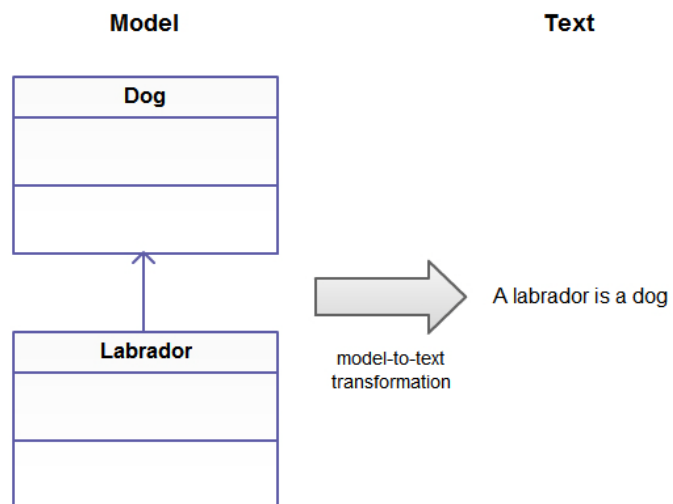


Fig. 4 An example of a very simple model-to-text transformation.

Figure 3 shows that a model can be transformed into a model (a model-to-model transformation), or into code (a model-to-text or M2T transformation).

In our case a M2T transformation is the best option, since code must be generated for the process simulator [11]. Therefore the model-to-model transformations are ignored and the focus is on the model-to-text transformations. An example of a very simple model-to-text transformation is shown in Fig. 4. In the model it is defined that a Labrador is a dog. The transformation takes the structure of the model and the information it contains and transforms it into a sentence in natural language.

A model transformation can be defined at model or at metamodel level. When the transformation is defined at metamodel level, it becomes reusable [12]. This is one of the major advantages of using models in software engineering. In our case it means that one metamodel can be transformed into a large number of different unit operations, different models combined with the same metamodel result in different unit operations.

3. Process Engineering

This chapter contains more specific information on process engineering. Section 3.1 discusses details about process simulation in general. Section 3.2 reports on how the process simulator used in this study models a chemical process. Section 3.3 analyses how a unit operation can be modelled.

3.1. Process Simulation

As mentioned in the introduction, there are several commercial process simulators available nowadays. They all have their own advantages and disadvantages, but they do share common features.

A number of features all these programs have are:

- A main executive program that controls and tracks the calculations.
- A library of modules (unit operations) that can simulate the equipment used in practice.
- A data bank of the physical and thermodynamic properties of pure components and mixtures thereof.

The structure of a typical simulation program is shown in Fig. 5.

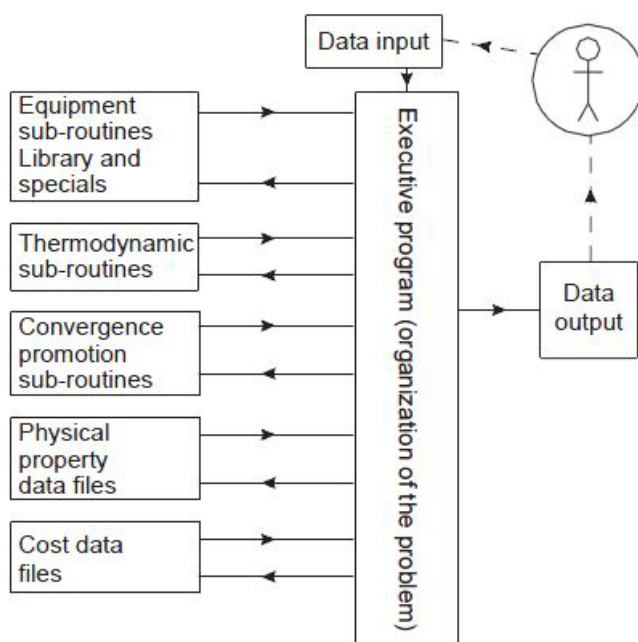


Fig. 5 A diagram of the structure of a typical simulation program [1].

However, process simulators can be divided into two groups when it comes to how the simulation is solved:

- *Sequential-modular programs*: the equations describing each unit operation (module) are solved in steps, module-by-module.
- *Simultaneous (or equation-oriented) programs*: the entire process is described by a set of equations, which are simultaneously solved. This means that simultaneous programs can also be used to simulate the unsteady-state operation of processes and equipment.

In the past, most available process simulators were of the sequential-modular type. Their advantage was that they were easier to develop than the equation-oriented programs. They also require less computing power and memory.

On the other hand, sequential-modular simulators cannot be used to solve dynamic, time-dependent models. Since dynamic models are described by thousands of differential equations, simultaneous program solvers are needed. They require more computing power and memory than the steady-state simulators, but with the development of fast powerful computers this is no longer a restriction.

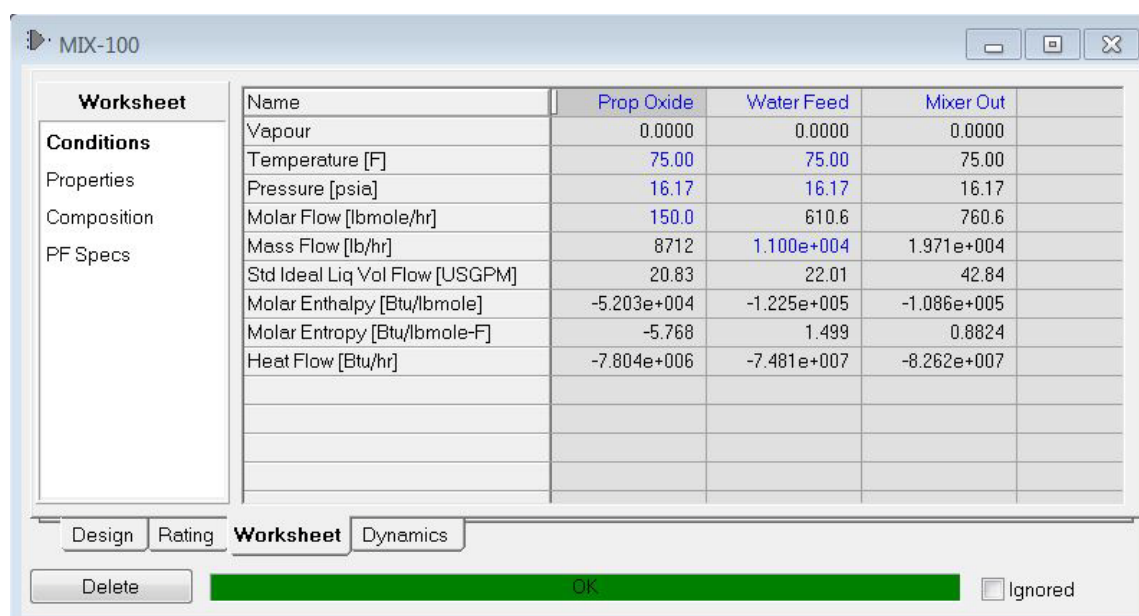
This has led to the development of hybrid programs. In these programs the sequential simulator and the simultaneous simulator both do what they are best at. The sequential simulator provides the initial conditions and the simultaneous simulator solves the equations [1].

In this study a typical process simulator is needed to test the developed tool. The simulator used is a hybrid simulator called UniSim Design R390TM which is licensed by Honeywell.

3.2. UniSim Design

The aim of this paragraph is to highlight certain design steps; it does not discuss in detail the whole simulation process from start to finish.

The first step in building a process simulation is selecting the chemical components involved. After this the reaction is defined. It is also possible to define a set of reactions. Next one or more feed streams are defined, which means that the properties (e.g. temperature, pressure and composition) of the feed stream(s) are specified. After this the unit operations can be installed and the corresponding parameters specified. When all the unit operations are installed, connected and specified, the simulation can be solved. The user can read the results from the property view of the unit operation. As example a property view of a mixer is given below. The property view displays the conditions of the connected streams. The properties or composition of these streams can be displayed by using the menu on the left.



Name	Prop Oxide	Water Feed	Mixer Out
Vapour	0.0000	0.0000	0.0000
Temperature [F]	75.00	75.00	75.00
Pressure [psia]	16.17	16.17	16.17
Molar Flow [lbmole/hr]	150.0	610.6	760.6
Mass Flow [lb/hr]	8712	1.100e+004	1.971e+004
Std Ideal Liq Vol Flow [USGPM]	20.83	22.01	42.84
Molar Enthalpy [Btu/lbmole]	-5.203e+004	-1.225e+005	-1.086e+005
Molar Entropy [Btu/lbmole-F]	-5.768	1.499	0.8824
Heat Flow [Btu/hr]	-7.804e+006	-7.481e+007	-8.262e+007

Fig. 6 The property view of a unit operation that mixes propylene oxide stream with a water stream.

Some properties of the feed stream must always be known. These properties are the components, molar flow (or total flow) and two of the following three properties: temperature, pressure and vapour fraction [1].

3.3. Unit operation

Unit operations are the parts of a chemical process in which physical and chemical operations take place. Fig.7 shows some very simple models of unit operations.

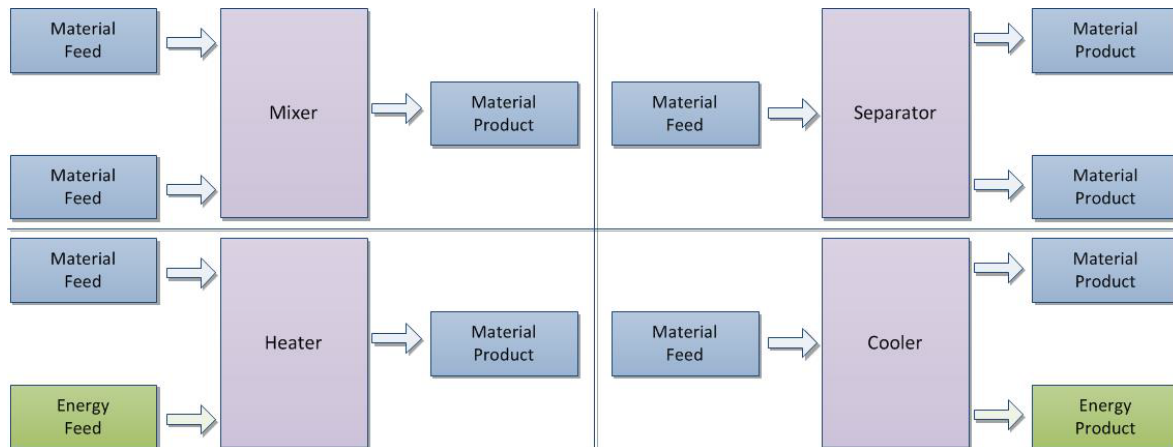


Fig. 7 Some models of unit operations.

When these models are analysed it becomes clear that a unit operation in general has a maximum of two material feeds or products and one energy feed or product. Hence, a unit operation can be generally modelled by the model in Fig. 8.

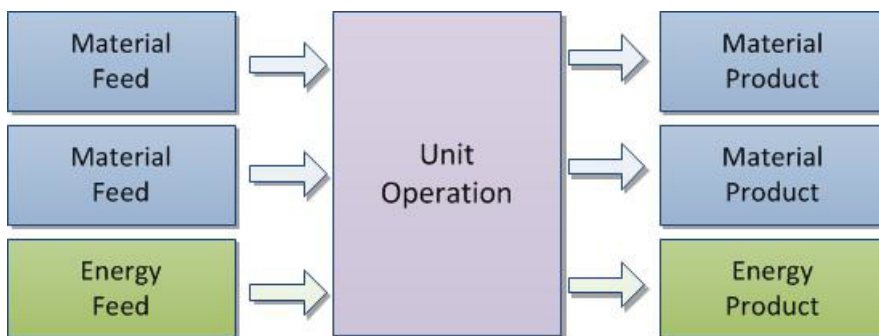


Fig. 8 A generalisation of a model of a unit operation.

Some unit operations have more than two feed or product streams. Examples are:

- Distillation, which may have three feed streams;
- Mixing, in which a large number of feed streams can be combined.

However, to keep the model of a unit operation simple, these instances are ignored in this study.

4. Solution Design

To design a solution for the problem, several steps have been taken. Section 4.1 gives an overview of those steps. Sections 4.2 to 4.4 describe these steps in detail.

4.1. Overview

The solution design consists of three steps. The first step is to define a metamodel, since a metamodel defines the language that is used to model the problem. The second step is a transformation from the model to text. This result of this transformation is a file in a language that the process simulator understands. This file then has to be imported in the process simulator, which is the last step.



Fig. 9 The steps that make up the solution design.

4.2. Metamodel

A metamodel is used to define the language in which the model is described (see Section 2.3.1). Ideally, the metamodel is defined in a way that enables the process engineer to put all the necessary information in the resulting model. In our case, it would mean that the engineer should be able to define the unit operation and its behaviour in the model.

The metamodel is defined with the program Eclipse Modelling Framework (EMF), which is a collection of plug-ins for the Eclipse Integrated Development Environment (IDE). The EMF project has developed a modelling framework and code generation facilities. It can be used to build tools and other applications based on structured data models [13].

4.3. Transformation

Our transformation should take all the information from the model and transform it into Visual Basic code, which is the language used by our process modelling tool. Since EMF is used for the metamodeling, it is logical to use one of the modelling components of Eclipse to implement the model-to-text transformation. There are several tools that can do this, but Xpand has been used in this work. Xpand uses a template to transform a model to text. Therefore, to be able to perform a transformation, a proper template has to be written [11].

4.4. Importing in UniSim

After the code is generated, the unit operation has to be imported in UniSim Design. There are several possibilities to do this, which are all discussed below.

The first possibility is that UniSim uses a library to store all defined unit operations. This would mean that a unit operation can be imported into UniSim by putting a self-defined unit operation in the library folder. However, this possibility does not work properly. After examining the program files of UniSim we concluded that UniSim does not use a library to store unit operations.

We also noticed that the source code of UniSim is not included in the program files. This means that a self-defined unit operation has to be imported in a way the UniSim designers devised and allowed. This left only two possibilities, namely to define:

- a User Unit Operation
- an Extension Unit Operation

Both possibilities are discussed below. Advantages and disadvantages are given and a choice between them is made and substantiated.

4.4.1. User Unit Operation

A user unit operation (UUO) makes use of automation, which means that programmers expose objects in the program, giving other applications or the operating system the opportunity to interact with these objects. Because only a number of objects of the program are exposed, the user does not need to understand the complete program to be able to interact with it.

To make use of this in UniSim Design, during development code was added to expose certain objects of the program. The end user can write Visual Basic code to access these objects and interact with them and no understanding of the UniSim source code is required.

A UUO is not very different from the Unit Operations already defined in UniSim, except that its complete behaviour can be defined by the user (as the name implies). To define this behaviour Visual Basic compatible code must be written.

There are three procedures that must be defined in order to define a UUO: Initialize(), Execute() and StatusQuery().

In Initialize() the names of the feed(s) and product(s) are defined and if necessary, the second feed and/or product are activated. It is also possible to name and activate the energy feed and product, if the component requires or generates energy during the process.

The Execute() procedure is called whenever a variable or the composition of a stream attached to the component is changed. Because of the change, the unit operation's calculations have to be redone. Therefore, all the unit operation's calculations are in this procedure.

The StatusQuery() procedure is called when UniSim wants to update the status information of an object. The purpose of this procedure is to provide warning and error messages. Warnings or errors could occur because, for example, connections or variable values are missing. The messages are shown in the UniSim Design status bar and at the bottom of the User Unit Operation property view [3].

When these three procedures are defined and the unit operation's structure and dynamics are known, it can be used in the process simulator.

The code that defines the UUO has to be written or pasted in a tab on the property view. Since this cannot be automated, the users always have to do this themselves (by hand).

The User Unit Operation can also be exported, which means that a self-defined unit operation, after it is defined, can be used in different UniSim projects.

Pros

- Easy to implement.
- Defined User Unit Operations can be exported and imported, therefore the same component can be used in different designs.

Cons

- Only two tabs in property view, which limits the amount of information and results that can be displayed.

4.4.2. Extension Unit Operation

The second possibility exploits the extensibility of UniSim Design. Extensibility allows one to enhance the existing functionality of a program with something that works well and efficiently. In case of a unit operation this would mean that the unit operation extension looks and feels the same as the unit operations given by the simulator itself. Once the extension is registered it can be used as if it were a part of the program. The ideal way of registering the extension would be by using a script that would automatically import the extension after generation and all the user would have to do is to start UniSim and use the new unit operation.

When an Extension Unit Operation (EUO) is used to define a model, nothing predefined is given, i.e., the user must build the Unit Operation from scratch. This means that there is no framework to be reused, which is a problem if the user is inexperienced either with UniSim or the used coding language.

The code should include explicit declaration of all the variables used, as well as the three procedures used in the User Unit Operation.

The extension can be defined in any OLE controller language. Examples of these languages are Visual Basic, C++ and Delphi.

The EUO differs from the UVO in the Property View, because with an EUO, this view must be constructed from scratch. This means that after implementing the code, the user also has to design the Property view before the extension can be used [3].

Pros

- Everything can be defined by the user, so the Unit Operation will behave completely according to the users' wishes.

Cons

- For a user not acquainted with UniSim or programming, it can take a lot of time to define a User Operation from scratch.
- The property view also needs to be designed from square one.

4.4.3. Conclusion

After researching both options, we decided to use the User Unit Operation option. With this option it is not possible to sculpt a unit operation completely to the users' wishes, but in our case time is a limiting factor, and the user unit operation option is most likely much faster to implement than the extension unit operation. This is because of several reasons, listed below:

- There is less code needed to define a user unit operation.
- The code is easier to understand, and hardly any knowledge of Visual Basic is required.
- UniSim automatically generates the property view, so it is not necessary to spend time constructing it.

There are, however, some drawbacks. The largest is that the generated code has to be pasted in the property view window, which means that our solution cannot be fully automated.

But since the advantages are far bigger than the disadvantages, the user unit operation is in our case the best option.

5. Metamodel

This chapter describes how the metamodel was defined. Section 5.1 details the requirements of the metamodel. Section 5.2 takes a close look at the resulting metamodel and its details. Section 5.3 gives possible improvements for the metamodel.

5.1. Requirements

To define a unit operation completely, the structure and the dynamics of the unit operation have to be described. In this case it means that the engineer should be able to put all information about structure and dynamics in a model. Therefore, the metamodel should be defined to work with all possible structures and dynamics.

The general structure of a unit operation was given in Section 3.3. A unit operation has three feed and three product nozzles. Two of those nozzles are for material streams, while one is for an energy stream. The unit operation uses at least one material feed and one material product nozzle. The other nozzles are optional. This characteristic should be considered in the design of the metamodel.

The dynamics of the unit operation should also be captured in a model. This means that the behaviour of the unit operation, expressed in mathematical equations, is considered in the metamodel, i.e., the metamodel should support the modelling of these mathematical equations.

5.2. Final version

The final version of the metamodel is shown in Fig. 10.

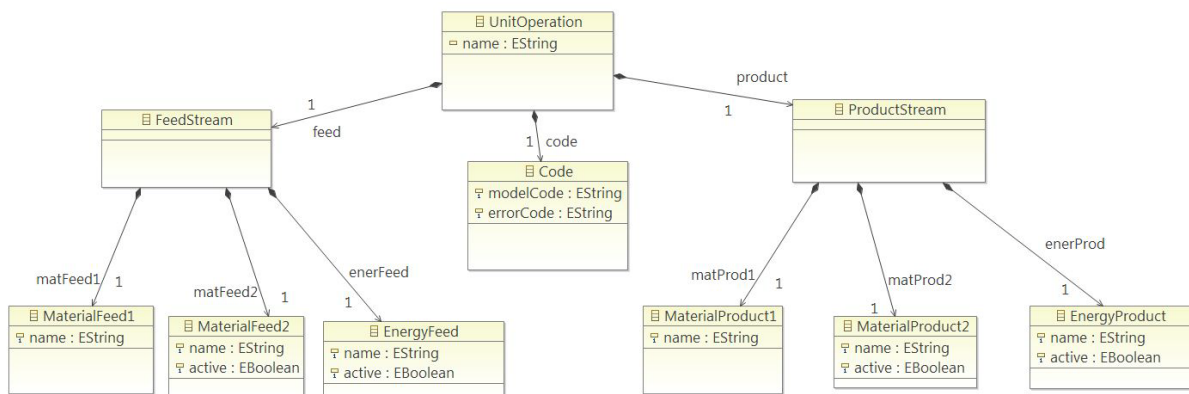


Fig. 10 The complete metamodel.

To explain certain design choices, the different parts of the metamodel are discussed below.

First we discuss the uppermost class of the metamodel. This is the class `UnitOperation` and it has one attribute, which is called `name` and defines the name of the unit operation. Each `UnitOperation` class has one `Code` child. It has also one `FeedStream` child and one `ProductStream` child. This part of the metamodel is in accordance the unit operation model defined in Fig. 8.

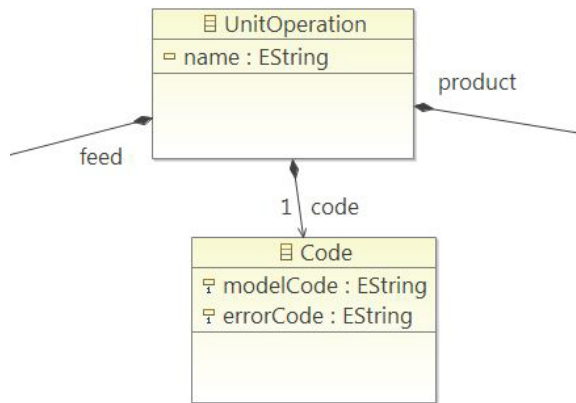


Fig. 11 The classes `UnitOperation` and `Code`.

This class `Code` has two attributes, namely `modelCode` and `errorCode`. As the name of the attributes indicates, these attributes hold code. The attribute `modelCode` contains the code that defines the behaviour of the unit operation. The attribute `errorCode` holds the code associated with the procedure `StatusQuery()` (see Section 4.4.1).

However, in Chapter 4 it was mentioned that the dynamics of the unit operation should be captured in a model. Due to time limitations, this has not been done.

The next part models the feed streams of the unit operation. All the different feed streams are represented by a class called `FeedStream`. This class does not have any attributes itself. `FeedStream` always has exactly three children, namely `MaterialFeed1`, `MaterialFeed2` and `EnergyFeed`.

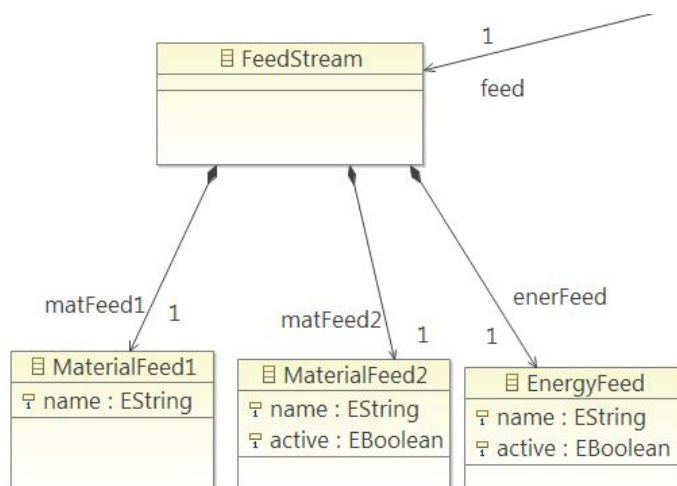


Fig. 12 Detail of the feed stream part of the metamodel.

The class `MaterialFeed1` has one attribute called `name`. The class `MaterialFeed2` has two attributes, namely `name` and `active`. The two material feeds differ from each other because the first material feed is always active, therefore the class `MaterialFeed1` does not have an attribute called `active`. The second material feed can be activated, but it can also be deactivated, and consequently the class `MaterialFeed2` has an attribute that enables the engineer to activate or deactivate the feed.

The third class is called EnergyFeed. This feed, like the second material feed, is not always necessary. Therefore, it has an attribute called active.

The last part of the metamodel defines the product streams.

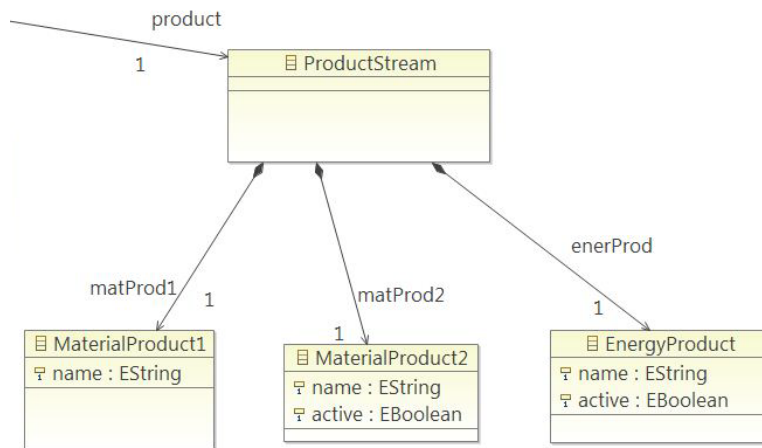


Fig. 13 Detailed view of the ProductStream class it's the lower classes.

Since the metamodel of the product streams has the same structure as the metamodel of the feed streams, it is not discussed in detail here. The design choices made for the product streams are essentially the same as those for the feed streams.

5.3. Possible improvements

The final model as discussed above still needs the user to input Visual Basic code to generate a working user unit operation. There are a couple of possibilities to improve this:

1. To find or design a tool that can translate equations from Matlab or Excel to Visual Basic. The user still has to define the code, but it would be done in a language that they understand, rather than Visual Basic.
2. To define a metamodel in this metamodel, instead of the class Code. This metamodel has its own transformation, to transform the model into Visual Basic code. Of the two options, this is most likely the most difficult to achieve.

6. Transformation

This chapter discusses the transformation of instances of the metamodel to Visual Basic code. Section 6.1 gives an overview of the different parts that make up the transformation template. Section 6.2 takes a closer look at some of those parts. Section 6.3 gives some possible improvements for the template.

6.1. Overview

In Section 4.4.1 we mentioned that to specify a unit operation in UniSim Design three procedures have to be defined, namely Initialize(), Execute() and StatusQuery(). These procedures are also recognisable in the transformation template. Tab. 1 shows the beginning of the template. Every time «EXPAND ... FOR ...» is used, another template is called. Some of these templates are discussed in the next paragraph.

Sub Initialize()	Begin of the procedure Initialize()
«EXPAND initFeed FOR feed» «EXPAND initProd FOR product»	Initializes the feed and product streams
End Sub	End of procedure
Sub Execute()	Begin of the procedure Execute()
«EXPAND exeFeed FOR feed» «EXPAND exeProd FOR product» «EXPAND exeCode FOR code»	Declares feed and product streams Gets the code from attribute modelCode*
End Sub	End of procedure
Sub StatusQuery()	Begin of the procedure StatusQuery()
«EXPAND sqMissingF FOR feed» «EXPAND sqMissingP FOR product» «EXPAND sqMissingC FOR code» «EXPAND sqInfoF FOR feed»	Deals with unconnected active nozzles Gets the code from attribute errorCode* Determines if enough information is known about the feed stream (e.g. temperature)
«EXPAND sqNozzleF FOR feed» «EXPAND sqNozzleP FOR product»	Checks if there are nozzles with more than one connection (since these connections are ignored)
«EXPAND sqBogusF FOR feed» «EXPAND sqBogusP FOR product»	Determines if there are connections to inactive nozzles
End Sub	End of procedure

Tab. 1 Beginning of the transformation template

* explained in Section 5.2

6.2. Closer look

In Section 5.2 we mentioned that the metamodel for the feed streams and the product streams is exactly the same. This is also true for the transformation template. Therefore, only the parts of the transformation that have to do with feed streams or code are discussed below. To improve readability, relatively uninteresting or duplicated lines are not shown (here duplicated means repeated code with a tiny change due to it being for a different stream). The complete transformation is given in Appendix C.

In the initFeed template, all the attributes of the feed streams are read. The code generated defines the names of the feed streams and determines if they are active. The same is done for the product streams in the template called initProd.

```

«DEFINE initFeed FOR uuo1::FeedStream»
    ActiveObject.Feeds1Name = "«this.matFeed1.name»"
    ActiveObject.Feeds2Name = "«this.matFeed2.name»"
    ActiveObject.Feeds2Active = «this.matFeed2.active»
    ActiveObject.EnergyFeedsName = "«this.enerFeed.name»"
    ActiveObject.EnergyFeedsActive = «this.enerFeed.active»
«ENDDDEFINE»

```

In the exeFeed template the feed streams are declared. Since this is only needed when the streams are active, there is a check to determine whether a stream is active. Again, the template exeProd for the product streams does exactly the same.

```

«DEFINE exeFeed FOR uuo1::FeedStream»
    Dim feed As Object
    Set feed = ActiveObject.Feeds1.Item(0)
    If feed Is Nothing Then GoTo EarlyExit

    «IF this.matFeed2.active» (almost the same code for the energy feed)
    Dim feed2 As Object
    Set feed2 = ActiveObject.Feeds2.Item(0)
    If feed2 Is Nothing Then GoTo EarlyExit
    «ENDIF»
«ENDDDEFINE»

```

The exeCode template splits the string from the attribute modelCode in the class Code into lines. The same is done in the template sqMissingC for the attribute errorCode.

```

«DEFINE exeCode FOR uuo1::Code»
    «FOREACH this.modelCode.split(";") AS a»«a.toString()»
    «ENDFOREACH»
«ENDDDEFINE»

```

The sqMissingF template checks if all the active nozzles have at least one feed stream attached to them. The same is done for the product stream nozzles in sqMissingP.

```

«DEFINE sqMissingF FOR uuo1::FeedStream»
    If ActiveObject.Feeds1.Count = 0 Then
        ActiveObject.AddStatusCondition(slMissingRequiredInformation, 1,
            "Feed Stream Required")
    End If

    «IF this.matFeed2.active» (almost the same code for the energy feed)
    If ActiveObject.Feeds2.Count = 0 Then
        ActiveObject.AddStatusCondition(slMissingRequiredInformation, 2,
            "Second Feed Stream Required")
    End If
    «ENDIF»
«ENDDDEFINE»

```

The template sqInfoF is the only template (beside the Code templates) that does not have a product stream counterpart.

In Section 3.2 we mentioned that there are two properties of a stream that must always be known (i.e. the components and the molar flow). In addition, two of the following three properties must be known as well: temperature, pressure and vapour fraction. This template checks to see if there is enough information known about the feed streams to calculate the unit operations.

Since the checks for the second feed stream are exactly the same, they are not shown here.

```
«DEFINE sqInfoF FOR uu01::FeedStream»
  AfterCompCheck:
    On Error GoTo That'sAll

    Dim GotTwo As Boolean
    GotTwo = False

    Dim feed As Object
    Set feed = ActiveObject.Feeds1.Item(0)

    If Not feed.Temperature.IsKnown Then
      ActiveObject.AddStatusCondition(slMissingOptionalInformation,
        12, "Feed Temperature Unknown")
      GotOne = True
    End If

    If Not feed.Pressure.IsKnown Then
      ActiveObject.AddStatusCondition(slMissingOptionalInformation,
        13, "Feed Pressure Unknown")
      If GotOne = True Then
        GotTwo = True
      End If
      GotOne = True
    End If

    If Not feed.VapourFraction.IsKnown Then
      ActiveObject.AddStatusCondition(slMissingOptionalInformation,
        10, "Feed Vapour Fraction Unknown")
      If GotOne = True Then
        GotTwo = True
      End If
      GotOne = True
    End If

    If GotTwo = True Then GoTo That'sAll
    GotOne = False

    If Not feed.MolarFlow.IsKnown Then
      ActiveObject.AddStatusCondition(slMissingOptionalInformation,
        14, "Feed Molar Flow Unknown")
      GotOne = True
    End If

    CMFsKnown = feed.ComponentMolarFraction.IsKnown

    If Not CMFsKnown(0) Then
      ActiveObject.AddStatusCondition(slMissingOptionalInformation,
        15, "Feed Composition Unknown")
      GotOne = True
    End If

    If GotOne = True Then GoTo That'sAll
«ENDDFINE»
```

This template checks if there are nozzles with more than one stream connected to them. The same is done for the product streams in sqNozzleP.

```

«DEFINE sqNozzleF FOR uuo1::FeedStream»
  If ActiveObject.Feeds1.Count > 1 Then
    ActiveObject.AddStatusCondition(slWarning, 20, "Additional Stream(s)
    of First Feed Ignored")
  End If

  «IF this.matFeed2.active» (almost the same code for the energy feed)
  If ActiveObject.Feeds2.Count > 1 Then
    ActiveObject.AddStatusCondition(slWarning, 21, "Additional Stream(s)
    of Second Feed Ignored")
  End If
  «ENDIF»
«ENDDEFINE»

```

The sqBogusF template checks if there are streams connected to inactive nozzles. Its counterpart for product streams is called sqBogusP.

```

«DEFINE sqBogusF FOR uuo1::FeedStream»
  Dim BogusCnxn As Boolean
  BogusCnxn = False

  «IF !this.matFeed2.active» (almost the same code for the energy feed)
  If BogusCnxn And ActiveObject.Feeds2.Count > 0 Then
    BogusCnxn = True
  End If
  «ENDIF»
«ENDDEFINE»

```

6.3 Possible improvements

The unit operation that is generated with the transformation only regards the first stream connected to a nozzle. This means that there is a maximum of two material feeds and two material product streams. Most of the time a unit operation has two or less feed or product streams, but if more streams need to be connected the template has to be changed.

7. Case Study

This chapter reports on the case study that has been done. The example used in the case study is a dehumidifier. A dehumidifier removes water from the feed stream. Of the two resulting products streams, one contains only water and the other contains the components of the feed stream minus the water.

Section 7.1 shows the model of the dehumidifier. Section 7.2 discusses the resulting transformation and Section 7.3 shows the dehumidifier after it has been imported in UniSim Design.

7.1. Model

Fig. 14 shows the model of a dehumidifier. The values of the attributes are given.

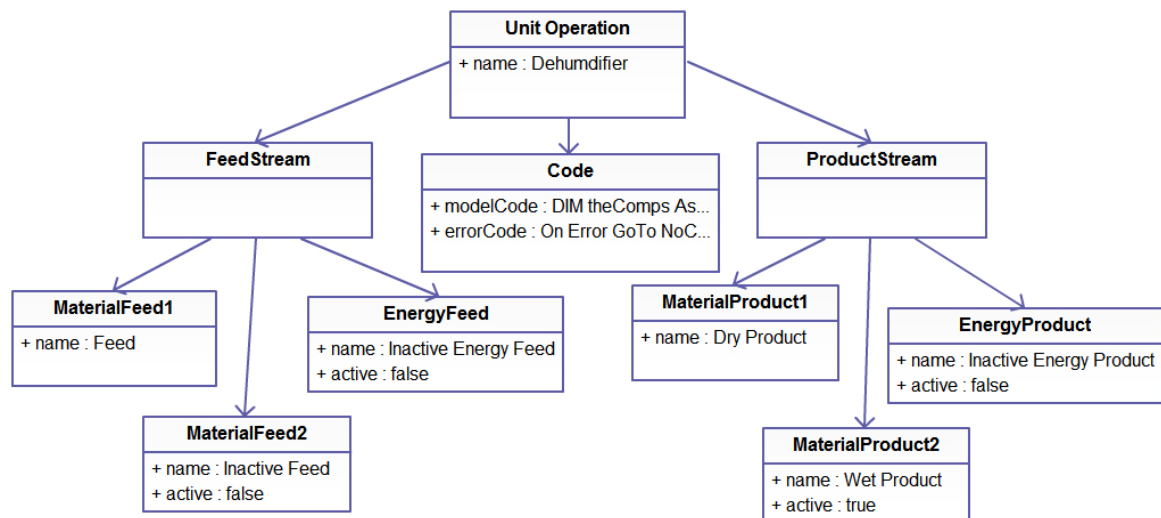


Fig. 14 The model of a dehumidifier.

The complete values of the attributes modelCode and errorCode from the class Code are not shown in this model, but can be found in Appendix B.

7.2. Transformation

The complete result of the transformation is given in Appendix D.

Below an example to show how the properties that are defined in the model of Fig. 14 are transformed into executable Visual Basic code by using our template. The code in this example is for Initialize() procedure.

Template	Result
ActiveObject.Products1Name = "«this.matProd1.name»"	ActiveObject.Products1Name = "Dry Product"
ActiveObject.Products2Name = "«this.matProd2.name»"	ActiveObject.Products2Name = "Wet Product"
ActiveObject.Products2Active = «this.matProd2.active»	ActiveObject.Products2Active = true
ActiveObject.EnergyProductsName = "«this.enerProd.name»"	ActiveObject.EnergyProductsName = "Inactive Energy Product"
ActiveObject.EnergyProductsActive = «this.enerProd.active»	ActiveObject.EnergyProductsActive = false

Tab. 2 The initProd template and the result this template gives after the transformation.

7.3. Result

The generated dehumidifier is imported into UniSim and tested. A feed comprising of several components is attached to the dehumidifier. The composition of this feed is shown in Tab. 3.

Component	Mole Fractions
Methane	0.40
Ethane	0.20
Propane	0.15
i-Butane	0.10
n-Butane	0.05
i-Pentane	0.05
H2O	0.05
<i>Total</i>	<i>1.00</i>

Tab. 3 The composition of the feed in mole fractions.

When fed into the dehumidifier the stream should be split into two product streams, one with only water and one with the rest of the components. The composition of those product streams should be as in Tab. 4.

Component	Mole Fraction Water stream	Mole Fraction Rest stream (rounded off)
Methane	0.00	0.4211
Ethane	0.00	0.2105
Propane	0.00	0.1579
i-Butane	0.00	0.1053
n-Butane	0.00	0.0526
i-Pentane	0.00	0.0526
H2O	1.00	0.0000
<i>Total</i>	<i>1.00</i>	<i>1.0000</i>

Tab. 4 The mole fraction of the rest stream can be calculated by dividing the feed stream mole fraction by 0.95 (1.00 – water). For methane the calculation is $0.4/0.95 \approx 0.42105$.

The composition UniSim gives can be seen in Fig. 15.

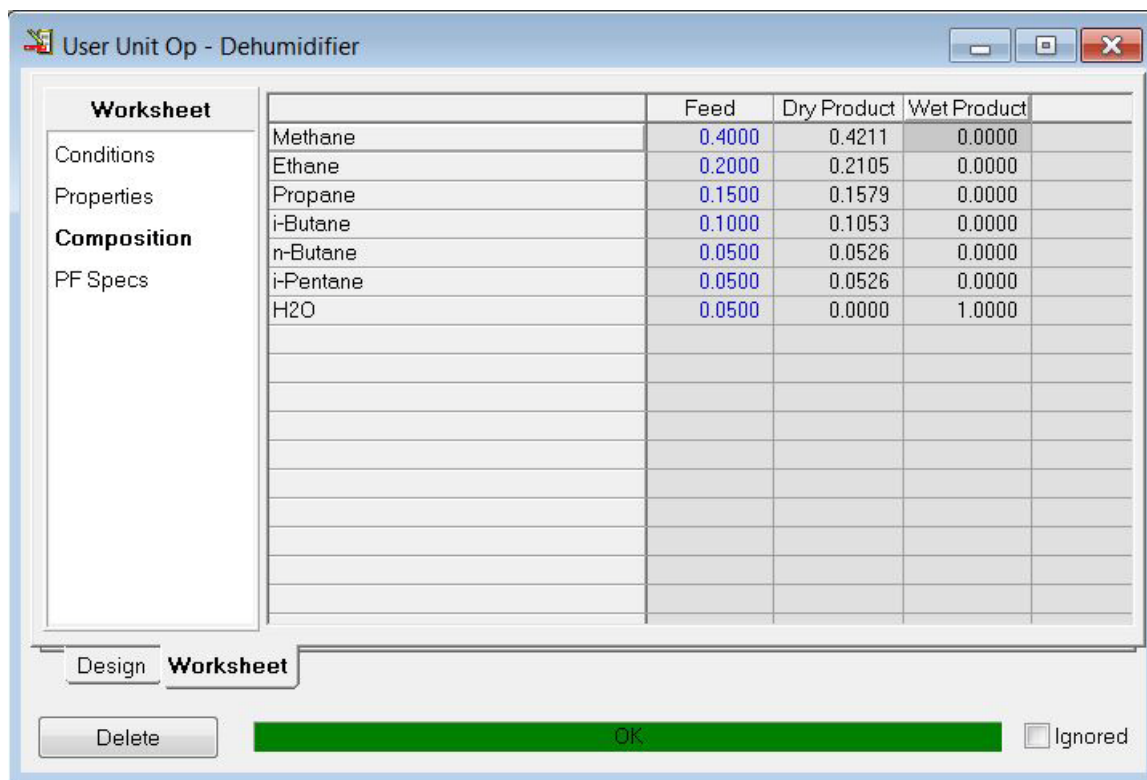


Fig. 15 The property view of a Dehumidifier user unit operation, showing the composition of the connected feed and product streams.

The figures UniSim has calculated are exactly the same as in Tab. 4. This means the composition of the product streams is equal to what was expected. A final check to determine the correctness of the unit operation is to check the molar flow. The molar flow of the feed stream is 220.5 lbmole/hr. From the composition of the feed stream (Tab. 3) we know that the mole fraction of the water stream is 5% of that of the feed stream. The rest of the feed stream goes into the dry product stream. This means that the molar flows of the different streams should be as in Tab. 5.

	Feed	Dry Product	Wet Product
Molar Flow (lbmole/hr)	220.5	209.475	11.025

Tab. 5 Molar flow of the streams connected to the dehumidifier.

UniSim gives the following conditions in the dehumidifier property view:

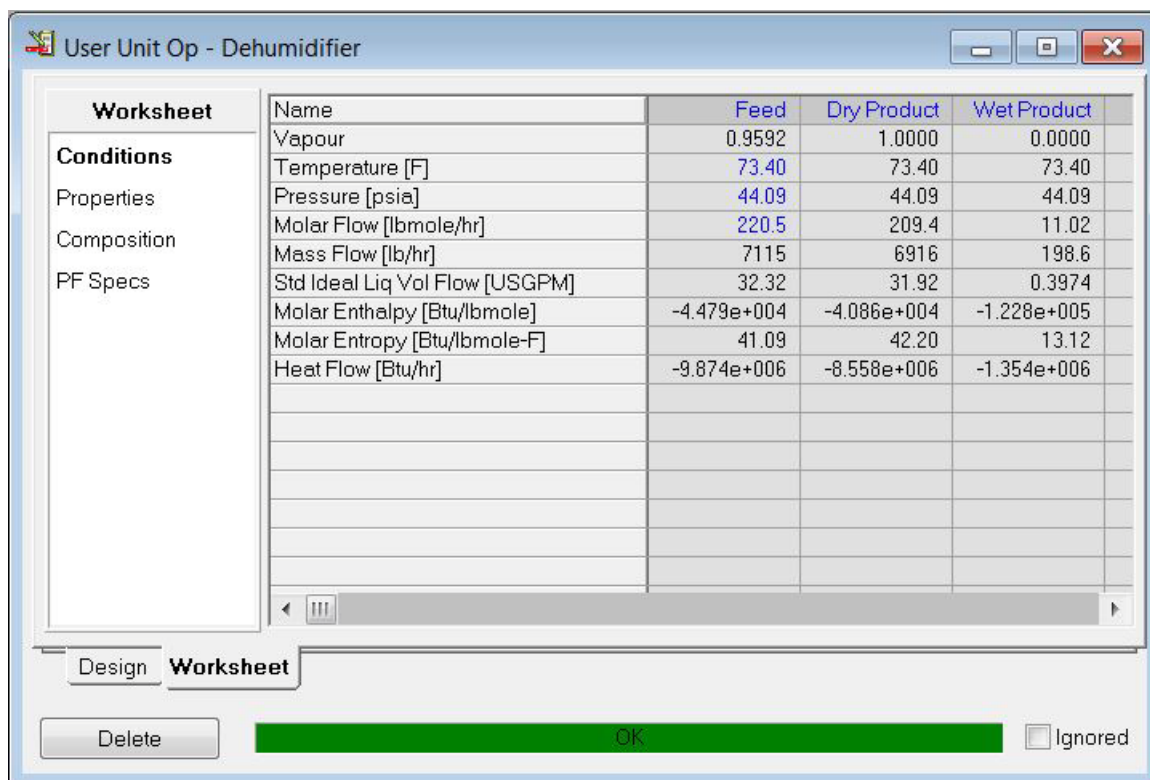


Fig. 16 Property view of a Dehumidifier user unit operation, open on the tab Conditions.

The fourth row of the Property view gives the figures that UniSim has calculated. Since they are rounded off it is impossible to say if the figures correspond exactly to those in Tab. 4, but it is most likely that they do.

Since the composition and the molar flow calculated by UniSim are equal to the expected figures, we conclude that the calculations made in the user unit operation are correct.

Some of the status messages are also tested. The tests and their outcome are given in the Tab. 6.

Test	Outcome (Message in status bar)	Verdict
Component to be split (in this case H2O) is not present in Fluid Package	Error: No water in Current Fluid Package	OK
Unknown feed stream temperature	Warning: Feed Temperature Unknown	OK
No connection to feed stream nozzle	Error: Feed Stream Required	OK
Connection to unused second feed stream nozzle	Warning: Connection(s) to Inactive Nozzles	OK
Two connections at dry stream nozzles	Warning: Additional First Product Stream(s) Ignored	OK

Tab. 6 Tests and their outcomes.

Since all tests have the expected outcome, we assume that the status messages work as expected.

8. Final remarks

Section 8.1 presents our general conclusions. Section 8.2 identifies some opportunities for future work.

8.1. General conclusions

In this study a method and tool have been developed to simplify the use of self-defined unit operation in process simulators. This has been done by using MDE technologies.

The road to a solution consisted of several steps. First a metamodel had to be defined. This metamodel looks quite like a model of a unit operation, thus making it easier to understand for the process engineer. The final version of the metamodel is not perfect, leaving possibilities for improvement.

Second, a transformation template had to be defined. This template transforms the model into executable code. This generated code then has to be copied and pasted into the UniSim User Unit Operation window.

If the process engineer wants to define his own unit operation, he has to define the unit operation model first. After the model is defined the code can be generated and pasted into UniSim, which creates a self-defined unit operation for the process engineer.

8.2. Future work

There are some improvements that can increase the benefits of the tool we developed.

The first and biggest improvement would be to find a way to remove the Visual Basic code from the model. Two parts of the model still have to be defined in Visual Basic code, thus still requiring some programming knowledge from the process engineer. In Section 5.3, two possible solutions for this problem are mentioned.

The second improvement is the development of a visual tool instead of a textual one. A visual tool makes it even easier for process engineers to model their own unit operations.

Another improvement can be made in the transformation template (mentioned in Section 6.3). Our tool only allows one stream to be connected per nozzle. Since there are some unit operations that require more than two feed or product streams, this is a drawback. However, this problem is most likely relatively easy to solve.

Finally, in Section 4.3 a choice was made between two ways for importing a user unit operation in UniSim Design. In this work the User Unit Operation was chosen, which is the easiest option to use, but not the most extensive. The Extension User Operation is more advanced, but also harder to implement. If time and knowledge of an OLE controller language are not a limiting factor, than the EUO is most likely the better choice. It can be defined completely to the users' wishes and the importing of the unit operations into UniSim can be fully automated. However, since the code required for the EUO has a completely different pattern in comparison with the code for the Uuo, the transformation template has to be rewritten (almost) completely.

References

- [1] G. Towler, R. Sinnott, Chemical Engineering Design, Principles, Practice and Economics of Plant and Process Design, Elsevier, 2007, Oxford
- [2] Wikipedia, List of chemical process simulators,
http://en.wikipedia.org/wiki/List_of_chemical_process_simulators, last modified: 2 March 2013, retrieved: 12 March 2013
- [3] Honeywell, UniSim® Design Customization Guide, R390 Release, 2009, Canada
- [4] I. Kurtev, L. Ferreira Pires, ADSA: Model-Driven Engineering, Presentation 1, 4 September 2012
- [5] J. Miller, J. Mukerji, MDA guide Version 1.0.1, Object Management Group, 2003
- [6] Wikipedia, Year 2000 problem,
http://en.wikipedia.org/wiki/Year_2000_problem#Programming_problem, last modified: 14 February 2013, retrieved: 20 February 2013
- [7] John Donne, Devotions upon Emergent Occasions, McGill-Queen's University Press, 1975, Quebec
- [8] E.D. Falkenberg, et al., A framework of information systems concepts, IFIP Working Group Vol. 8, 1998
- [9] I. Kurtev, L. Ferreira Pires, ADSA: Model-Driven Engineering, Presentation 2, 10 September 2012
- [10] Wikipedia, Unified Modeling Language,
http://en.wikipedia.org/wiki/Unified_Modeling_Language, last modified: 12 March 2013, retrieved: 13 March 2013
- [11] I. Kurtev, L. Ferreira Pires, ADSA: Model-Driven Engineering, Presentation 7, 15 Oktober 2012
- [12] I. Kurtev, L. Ferreira Pires, ADSA: Model-Driven Engineering, Presentation 4, 25 September 2012
- [13] The Eclipse Foundation, Eclipse Modeling Framework Project (EMF),
<http://www.eclipse.org/modeling/emf/>, retrieved: 4 March 2013

Appendix A: Glossary

Component

Part of a feed or product stream, for example water or methane

EUO

Extension Unit Operation, see chapter 4.4.2.

MDA

Stands for model-driven architecture

MDE

Stands for model-driven engineering

Metamodel

Defines the language that is used to model the problem

Model

A simplified representation of a system, in the language defined by the metamodel

Transformation

The process of converting a model into executable code

User

The user of UniSim Design

Unit operation

Part of a chemical process in which a physical or thermodynamic change takes place, for example mixing or distillation

UUO

User Unit Operation, see chapter 4.4.1.

Appendix B: Code from model

Below the code for the two attributes modelCode and errorCode from the class Code is given. On the right are comments to explain the lines. The comments made are for this particular unit operation, but general knowledge can be gained from them as well. If certain comments are not clear, reading Appendix E5 might help.

modelCode

compPosn = theComps.index("H2O")	finds the position of water in the current Fluid Package's component list (theComps is the variable that denotes the component list)
Dim CMFs As Variant	declares CMFs as a variant (CMF is an abbreviation for ComponentMolarFlow)
CMFs = feed.ComponentMolarFlowValue	assigns the molar flow of the feed per component to CMFs
CompFlow = CMFs(compPosn)	assigns the molar flow of the component (here H2O) to CompFlow
OtherFlow = feed.MolarFlowValue – CompFlow	OtherFlow is the molar flow of the feed stream minus that of the water (in this case the numbers are: $0.95 = 1 - 0.05$)
prod2.Pressure.Calculate(feed.PressureValue) prod2.Temperature.Calculate(feed.TemperatureValue) prod2.MolarFlow.Calculate(CompFlow)	the pressure and temperature of the water stream are calculated from the properties of the feed stream, the molar flow is calculated from the CompFlow
CMFs(compPosn) = 0.0	the molarflow of the water is set to zero, this stream now has the composition of the dry stream
prod.Pressure.Calculate(feed.PressureValue) prod.Temperature.Calculate(feed.TemperatureValue) prod.MolarFlow.Calculate(OtherFlow)	the pressure and temperature of the dry stream are calculated from the feed stream, the molar flow is calculated from the OtherFlow
For i=0 To theComps.Count-1 CMFs(i) = CMFs(i) / OtherFlow Next i	for all components, the new component molar flows in the dry stream are calculated the new molar flow equals the old molar flow divided by (in this case) 0.95
prod.ComponentMolarFraction.Calculate(CMFs)	the component molar fractions of the dry stream are calculated from the component molar flows of the dry stream
For i=0 To theComps.Count-1 CMFs(i) = 0.0 Next i	for all components, the component molar flows in the water stream are set to zero

CMFs(compPosn) = 1.0	the molar flow of water in the water stream is set to 1
prod2.ComponentMolarFraction.Calculate(CMFs)	the component molar fractions of the water stream are calculated from the molar flows of the water stream

errorCode

On Error GoTo NoComponent	if the next line gives an error, the program will continue at the line NoComponent:
compPosn = ActiveObject.Flowsheet.FluidPackage.Components.index("H2O")	if H2O is not in the current fluid package, this will give an error
GoTo AfterCompCheck	if there is no error, the program will continue at the line AfterCompCheck: (where other checks will be performed)
NoComponent:	If the program comes at this line there is no H2O in the fluid package
GotOne = True ActiveObject.AddStatusCondition(slError, 11, "No H2O in Current Fluid Package")	gives the user the error message that there is no H2O in the fluid package

Appendix C: Transformation template

Below the complete transformation template is given.

```
«IMPORT uuo1»

«DEFINE main FOR UnitOperation»
  «FILE this.name.toString()+".vb"»
Sub Initialize()
  «EXPAND initFeed FOR feed»
  «EXPAND initProd FOR product»
End Sub

Sub Execute()
  «EXPAND exeFeed FOR feed»
  «EXPAND exeProd FOR product»
  «EXPAND exeCode FOR code»
End Sub

Sub StatusQuery()
  «EXPAND sqMissingF FOR feed»
  «EXPAND sqMissingP FOR product»
  «EXPAND sqMissingC FOR code»
  «EXPAND sqInfoF FOR feed»
  «EXPAND sqNozzleF FOR feed»
  «EXPAND sqNozzleP FOR product»
  «EXPAND sqBogusF FOR feed»
  «EXPAND sqBogusP FOR product»
End Sub
  «ENDFILE»
«ENDDEFINE»

«DEFINE initFeed FOR uuo1::FeedStream»
  ActiveObject.Feeds1Name = "«this.matFeed1.name»"
  ActiveObject.Feeds2Name = "«this.matFeed2.name»"
  ActiveObject.Feeds2Active = «this.matFeed2.active»
  ActiveObject.EnergyFeedsName = "«this.enerFeed.name»"
  ActiveObject.EnergyFeedsActive = «this.enerFeed.active»
«ENDDEFINE»

«DEFINE initProd FOR uuo1::ProductStream»
  ActiveObject.Products1Name = "«this.matProd1.name»"
  ActiveObject.Products2Name = "«this.matProd2.name»"
  ActiveObject.Products2Active = «this.matProd2.active»
  ActiveObject.EnergyProductsName = "«this.enerProd.name»"
  ActiveObject.EnergyProductsActive = «this.enerProd.active»
«ENDDEFINE»

«DEFINE exeFeed FOR uuo1::FeedStream»
  On Error GoTo EarlyExit

  Dim feed As Object
  Set feed = ActiveObject.Feeds1.Item(0)
  If feed Is Nothing Then GoTo EarlyExit

  «IF this.matFeed2.active»
  Dim feed2 As Object
  Set feed2 = ActiveObject.Feeds2.Item(0)
  If feed2 Is Nothing Then GoTo EarlyExit
  «ENDIF»
```

```

«IF this.enerFeed.active»
Dim enfeed As Object
Set enfeed = ActiveObject.EnergyFeeds.Item(0)
If enfeed Is Nothing Then GoTo EarlyExit
«ENDIF»
«ENDDEFINE»

«DEFINE exeProd FOR uu01::ProductStream»
Dim prod As Object
Set prod = ActiveObject.Products1.Item(0)
If prod Is Nothing Then GoTo EarlyExit

«IF this.matProd2.active»
Dim prod2 As Object
Set prod2 = ActiveObject.Products2.Item(0)
If prod2 Is Nothing Then GoTo EarlyExit
«ENDIF»

«IF this.enerProd.active»
Dim enprod As Object
Set enprod = ActiveObject.EnergyProducts.Item(0)
If enprod Is Nothing Then GoTo EarlyExit
«ENDIF»
«ENDDEFINE»

«DEFINE exeCode FOR uu01::Code»
Dim theComps As Object
Set theComps = ActiveObject.Flowsheet.FluidPackage.Components

«FOREACH this.modelCode.split(";") AS a»«a.toString()»
«ENDFOREACH»
ActiveObject.SolveComplete
Exit Sub
EarlyExit:
' not enough info to calculate
«ENDDEFINE»

«DEFINE sqMissingF FOR uu01::FeedStream»
On Error GoTo ThatsAll

Dim GotOne As Boolean
GotOne = False

If ActiveObject.Feeds1.Count = 0 Then
    GotOne = True
    ActiveObject.AddStatusCondition(slMissingRequiredInformation, 1,
    "Feed Stream Required")
End If

«IF this.matFeed2.active»
If ActiveObject.Feeds2.Count = 0 Then
    GotOne = True
    ActiveObject.AddStatusCondition(slMissingRequiredInformation, 2,
    "Second Feed Stream Required")
End If
«ENDIF»
«IF this.enerFeed.active»
If ActiveObject.EnergyFeeds.Count = 0 Then

```



```

        GotOne = True
        ActiveObject.AddStatusCondition(slMissingRequiredInformation, 5,
            "Energy Feed Stream Required")
    End If
«ENDIF»
«ENDDEFINE»

«DEFINE sqMissingP FOR uuo1::ProductStream»
    If ActiveObject.Products1.Count = 0 Then
        GotOne = True
        ActiveObject.AddStatusCondition(slMissingRequiredInformation, 3,
            "Product Stream Required")
    End If

    «IF this.matProd2.active»
    If ActiveObject.Products2.Count = 0 Then
        GotOne = True
        ActiveObject.AddStatusCondition(slMissingRequiredInformation, 4,
            "Second Product Stream Required")
    End If
    «ENDIF»
    «IF this.enerProd.active»
    If ActiveObject.EnergyProducts.Count = 0 Then
        GotOne = True
        ActiveObject.AddStatusCondition(slMissingRequiredInformation, 6,
            "Energy Product Stream Required")
    End If
    «ENDIF»

    If GotOne = True Then GoTo ThatsAll
«ENDDEFINE»

«DEFINE sqMissingC FOR uuo1::Code»
    «FOREACH this.errorCode.split(";") AS a»«a.toString()»
    «ENDFOREACH»
«ENDDEFINE»

«DEFINE sqInfoF FOR uuo1::FeedStream»
    AfterCompCheck:
        On Error GoTo ThatsAll

        Dim GotTwo As Boolean
        GotTwo = False

        Dim feed As Object
        Set feed = ActiveObject.Feeds1.Item(0)

        If Not feed.Temperature.IsKnown Then
            ActiveObject.AddStatusCondition(slMissingOptionalInformation,
                12, "Feed Temperature Unknown")
            GotOne = True
        End If

        If Not feed.Pressure.IsKnown Then
            ActiveObject.AddStatusCondition(slMissingOptionalInformation,
                13, "Feed Pressure Unknown")
            If GotOne = True Then
                GotTwo = True
            End If
        End If
    End If
End Define

```

```

        GotOne = True
    End If

    If Not feed.VapourFraction.IsKnown Then
        ActiveObject.AddStatusCondition(slMissingOptionalInformation,
            10, "Feed Vapour Fraction Unknown")
        If GotOne = True Then
            GotTwo = True
        End If
        GotOne = True
    End If

    If GotTwo = True Then GoTo ThatsAll
    GotOne = False

    If Not feed.MolarFlow.IsKnown Then
        ActiveObject.AddStatusCondition(slMissingOptionalInformation,
            14, "Feed Molar Flow Unknown")
        GotOne = True
    End If

    CMFsKnown = feed.ComponentMolarFraction.IsKnown

    If Not CMFsKnown(0) Then
        ActiveObject.AddStatusCondition(slMissingOptionalInformation,
            15, "Feed Composition Unknown")
        GotOne = True
    End If

    If GotOne = True Then GoTo ThatsAll
    «IF this.matFeed2.active»
    Dim feed2 As Object
    Set feed2 = ActiveObject.Feeds2.Item(0)

    If Not feed2.Temperature.IsKnown Then
        ActiveObject.AddStatusCondition(slMissingOptionalInformation,
            16, "Second Feed Temperature Unknown")
        GotOne = True
    End If

    If Not feed2.Pressure.IsKnown Then
        ActiveObject.AddStatusCondition(slMissingOptionalInformation,
            17, "Second Feed Pressure Unknown")
        If GotOne = True Then
            GotTwo = True
        End If
        GotOne = True
    End If

    If Not feed2.VapourFraction.IsKnown Then
        ActiveObject.AddStatusCondition(slMissingOptionalInformation,
            11, "Second Feed Vapour Fraction Unknown")
        If GotOne = True Then
            GotTwo = True
        End If
        GotOne = True
    End If

    If GotTwo = True Then GoTo ThatsAll

```

```

GotOne = False

If Not feed2.MolarFlow.IsKnown Then
    ActiveObject.AddStatusCondition(slMissingOptionalInformation,
    18, "Second Feed Molar Flow Unknown")
    GotOne = True
End If

CMFsKnown = feed2.ComponentMolarFraction.IsKnown

If Not CMFsKnown(0) Then
    ActiveObject.AddStatusCondition(slMissingOptionalInformation,
    19, "Second Feed Composition Unknown")
    GotOne = True
End If

If GotOne = True Then GoTo ThatsAll
«ENDIF»
«ENDDEFINE»

«DEFINE sqNozzleF FOR uuo1::FeedStream»
    If ActiveObject.Feeds1.Count > 1 Then
        GotOne = True
        ActiveObject.AddStatusCondition(slWarning, 20, "Additional
        First Feed Stream(s) Ignored")
    End If

    «IF this.matFeed2.active»
    If ActiveObject.Feeds2.Count > 1 Then
        GotOne = True
        ActiveObject.AddStatusCondition(slWarning, 23, "Additional
        Second Feed Stream(s) Ignored")
    End If
    «ENDIF»
    «IF this.enerFeed.active»
    If ActiveObject.EnergyFeeds.Count > 1 Then
        GotOne = True
        ActiveObject.AddStatusCondition(slWarning, 24, "Additional
        Energy Feed Stream(s) Ignored")
    End If
    «ENDIF»
«ENDDEFINE»

«DEFINE sqNozzleP FOR uuo1::ProductStream»
    If ActiveObject.Products1.Count > 1 Then
        GotOne = True
        ActiveObject.AddStatusCondition(slWarning, 21, "Additional
        First Product Stream(s) Ignored")
    End If

    «IF this.matProd2.active»
    If ActiveObject.Products2.Count > 1 Then
        GotOne = True
        ActiveObject.AddStatusCondition(slWarning, 22, "Additional
        Second Product Stream(s) Ignored")
    End If
    «ENDIF»
    «IF this.enerProd.active»
    If ActiveObject.EnergyProducts.Count > 1 Then

```

```

        GotOne = True
        ActiveObject.AddStatusCondition(slWarning, 25, "Additional
        Energy Product Stream(s) Ignored")
    End If
«ENDIF»
«ENDDEFINE»

«DEFINE sqBogusF FOR uu01::FeedStream»
    Dim BogusCnxn As Boolean
    BogusCnxn = False

    «IF !this.matFeed2.active»
    If BogusCnxn And ActiveObject.Feeds2.Count > 0 Then
        BogusCnxn = True
    End If
    «ENDIF»
    «IF !this.enerFeed.active»
    If BogusCnxn And ActiveObject.EnergyFeeds.Count > 0 Then
        BogusCnxn = True
    End If
    «ENDIF»
«ENDDEFINE»

«DEFINE sqBogusP FOR uu01::ProductStream»
    «IF !this.matProd2.active»
    If BogusCnxn And ActiveObject.Products2.Count > 0 Then
        BogusCnxn = True
    End If
    «ENDIF»
    «IF !this.enerProd.active»
    If BogusCnxn And ActiveObject.EnergyProducts.Count > 0 Then
        BogusCnxn = True
    End If
    «ENDIF»
    If BogusCnxn = True Then
        GotOne = True
        ActiveObject.AddStatusCondition(slWarning, 7, "Connection(s) to
        Inactive nozzles")
    End If

    That'sAll:

«ENDDEFINE»

```

Appendix D: Generated dehumidifier

Below the generated dehumidifier is given. This is the complete transformation minus a few empty lines.

Sub Initialize()

```
    ActiveObject.Feeds1Name = "Feed"
    ActiveObject.Feeds2Name = "Inactive Feed"
    ActiveObject.Feeds2Active = false
    ActiveObject.EnergyFeedsName = "Inactive Energy Feed"
    ActiveObject.EnergyFeedsActive = false

    ActiveObject.Products1Name = "Dry Product"
    ActiveObject.Products2Name = "Wet Product"
    ActiveObject.Products2Active = true
    ActiveObject.EnergyProductsName = "Inactive Energy Product"
    ActiveObject.EnergyProductsActive = false
```

End Sub

Sub Execute()

```
    On Error GoTo EarlyExit

    Dim feed As Object
    Set feed = ActiveObject.Feeds1.Item(0)
    If feed Is Nothing Then GoTo EarlyExit

    Dim prod As Object
    Set prod = ActiveObject.Products1.Item(0)
    If prod Is Nothing Then GoTo EarlyExit

    Dim prod2 As Object
    Set prod2 = ActiveObject.Products2.Item(0)
    If prod2 Is Nothing Then GoTo EarlyExit

    DIM theComps As Object
    Set theComps = ActiveObject.Flowsheet.FluidPackage.Components
    compPosn = theComps.index("H2O")

    Dim CMFs As Variant
    CMFs = feed.ComponentMolarFlowValue
    CompFlow = CMFs(compPosn)
    OtherFlow = feed.MolarFlowValue - CompFlow

    prod2.Pressure.Calculate(feed.PressureValue)
    prod2.Temperature.Calculate(feed.TemperatureValue)
    prod2.MolarFlow.Calculate(CompFlow)

    CMFs(compPosn) = 0.0

    prod.Pressure.Calculate(feed.PressureValue)
    prod.Temperature.Calculate(feed.TemperatureValue)
    prod.MolarFlow.Calculate(OtherFlow)
```

```

For i=0 To theComps.Count-1
    CMFs(i) = CMFS(i) / OtherFlow
Next i

prod.ComponentMolarFraction.Calculate(CMFs)

For i=0 To theComps.Count-1
    CMFs(i) = 0.0
Next i
CMFs(compPosn) = 1.0
prod2.ComponentMolarFraction.Calculate(CMFs)

ActiveObject.SolveComplete
Exit Sub
EarlyExit:
' not enough info to calculate
End Sub

Sub StatusQuery()
    On Error GoTo That'sAll

    Dim GotOne As Boolean
    GotOne = False

    If ActiveObject.Feeds1.Count = 0 Then
        GotOne = True
        ActiveObject.AddStatusCondition(slMissingRequiredInformation, 1, "Feed Stream
        Required")
    End If

    If ActiveObject.Products1.Count = 0 Then
        GotOne = True
        ActiveObject.AddStatusCondition(slMissingRequiredInformation, 3, "Product Stream
        Required")
    End If

    If ActiveObject.Products2.Count = 0 Then
        GotOne = True
        ActiveObject.AddStatusCondition(slMissingRequiredInformation, 4, "Second Product
        Stream Required")
    End If

    If GotOne = True Then GoTo That'sAll

    On Error GoTo NoComponent
    compPosn = ActiveObject.Flowsheet.FluidPackage.Components.index("H2O")
    GoTo AfterCompCheck

    NoComponent:
    GotOne = True
    ActiveObject.AddStatusCondition(slError, 11, "No H2O in Current Fluid Package")

```

AfterCompCheck:

On Error GoTo ThatsAll

Dim GotTwo As Boolean

GotTwo = False

Dim feed As Object

Set feed = ActiveObject.Feeds1.Item(0)

If Not feed.Temperature.IsKnown Then

ActiveObject.AddStatusCondition(slMissingOptionalInformation, 12, "Feed
Temperature Unknown")

GotOne = True

End If

If Not feed.Pressure.IsKnown Then

ActiveObject.AddStatusCondition(slMissingOptionalInformation, 13, "Feed
Pressure Unknown")

If GotOne = True Then

GotTwo = True

End If

GotOne = True

End If

If Not feed.VapourFraction.IsKnown Then

ActiveObject.AddStatusCondition(slMissingOptionalInformation, 10, "Feed
Vapour Fraction Unknown")

If GotOne = True Then

GotTwo = True

End If

GotOne = True

End If

If GotTwo = True Then GoTo ThatsAll

GotOne = False

If Not feed.MolarFlow.IsKnown Then

ActiveObject.AddStatusCondition(slMissingOptionalInformation, 14, "Feed
Molar Flow Unknown")

GotOne = True

End If

CMFsKnown = feed.ComponentMolarFraction.IsKnown

If Not CMFsKnown(0) Then

ActiveObject.AddStatusCondition(slMissingOptionalInformation, 15, "Feed
Composition Unknown")

GotOne = True

End If

If GotOne = True Then GoTo ThatsAll

```

    If ActiveObject.Feeds1.Count > 1 Then
        GotOne = True
        ActiveObject.AddStatusCondition(slWarning, 20, "Additional First Feed
        Stream(s) Ignored")
    End If

    If ActiveObject.Products1.Count > 1 Then
        GotOne = True
        ActiveObject.AddStatusCondition(slWarning, 21, "Additional First Product
        Stream(s) Ignored")
    End If

    If ActiveObject.Products2.Count > 1 Then
        GotOne = True
        ActiveObject.AddStatusCondition(slWarning, 22, "Additional Second Product
        Stream(s) Ignored")
    End If

    Dim BogusCnxn As Boolean
    BogusCnxn = False

    If BogusCnxn And ActiveObject.Feeds2.Count > 0 Then
        BogusCnxn = True
    End If

    If BogusCnxn And ActiveObject.EnergyFeeds.Count > 0 Then
        BogusCnxn = True
    End If

    If BogusCnxn And ActiveObject.EnergyProducts.Count > 0 Then
        BogusCnxn = True
    End If

    If BogusCnxn = True Then
        GotOne = True
        ActiveObject.AddStatusCondition(slWarning, 7, "Connection(s) to Inactive nozzles")
    End If

    ThatsAll:
End Sub

```


Appendix E: User Manual

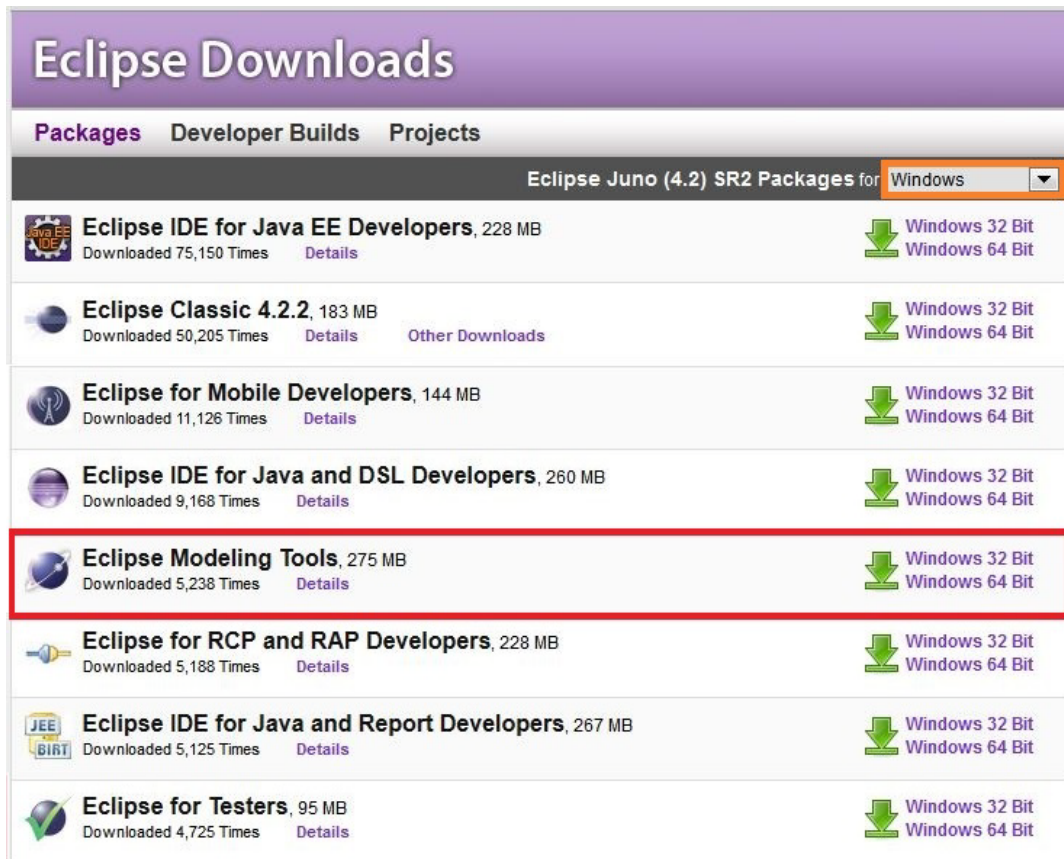
This user manual is aimed at users with UniSim experience and no knowledge of the Eclipse Modelling Framework. It is split into five parts. The first three parts deal with the first time operation of the tool. The fourth part describes how the tool can be used. The fifth part gives information and tips that help with the coding in Visual Basic.

E1. Installing EMT and XPand

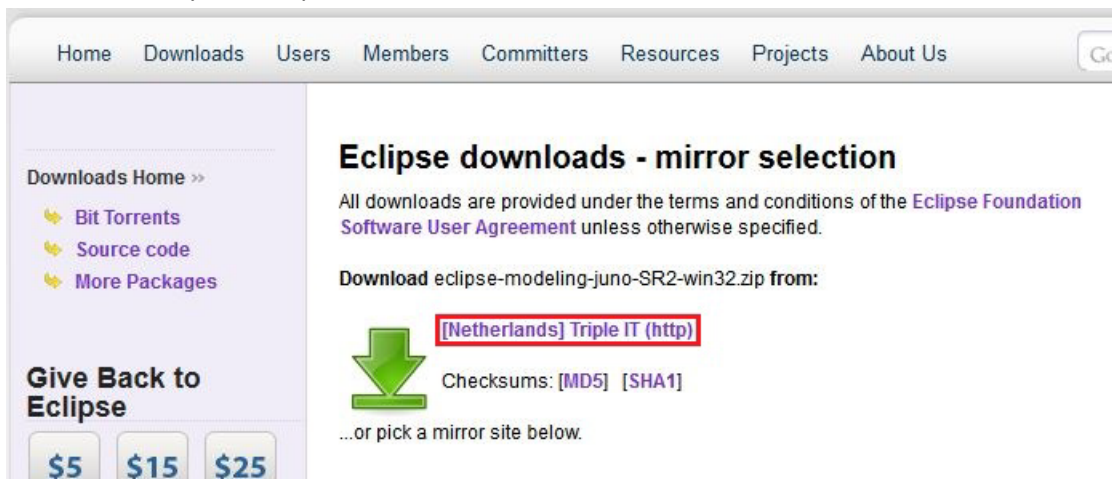
1. Eclipse needs **Java** to function. To check whether you have Java, go to java.com/download. To **check** which **version** of Java is installed on your computer, 'click Do I have Java?' (green box). If you do not have Java installed or if you have version 5 or lower, download the latest update.



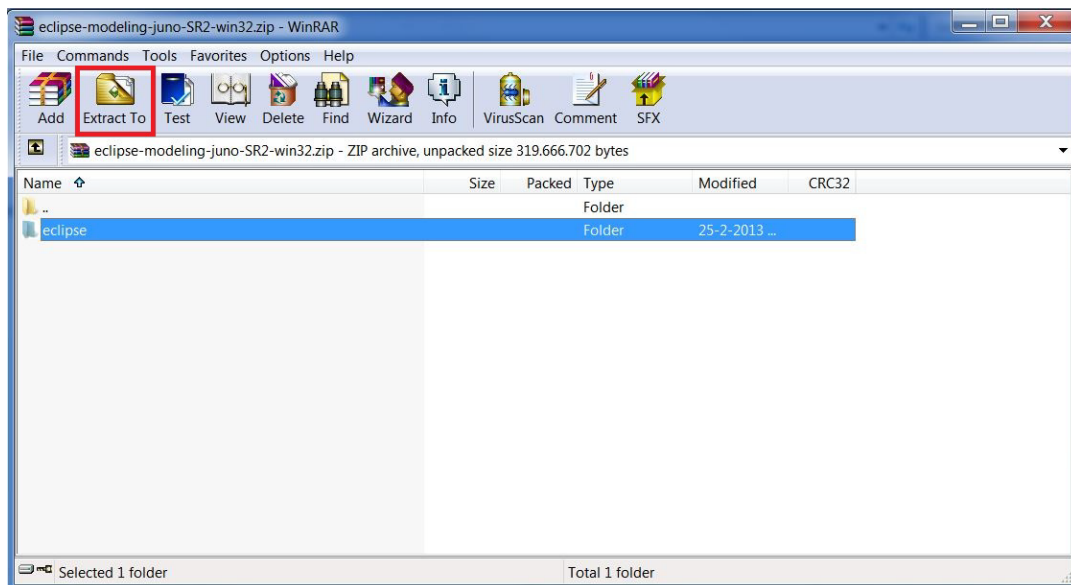
2. To **download Eclipse**, go to <http://www.eclipse.org/downloads/>
3. Download **Eclipse Modeling Tools** (red box) for your machine. If your operating system is not Windows, use the dropdown box highlighted in orange to select your operating system.



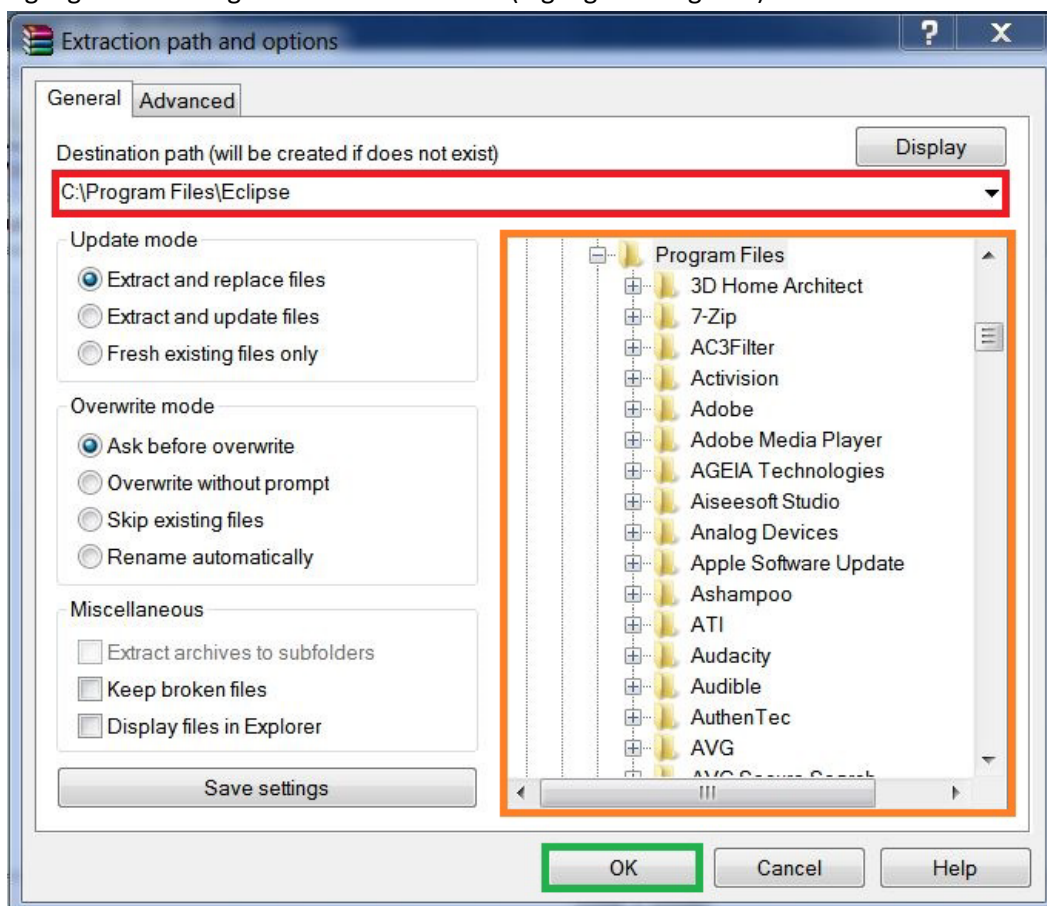
- You will go to a new site that looks like the picture below. Click on the link in the red box and **save the file** to your computer.



- Open** the downloaded .rar-file. You can find a program called WINRAR that does this on <http://www.rarlab.com/download.htm> (download and install it).
- Select the **eclipse** folder (one click) and click **Extract To** (red box).



7. A window will pop up. Choose the path where you want to **install Eclipse**. This can be done by typing in the box highlighted in red, or by choosing the destination folder in the box highlighted in orange. Afterwards click **OK** (highlighted in green).

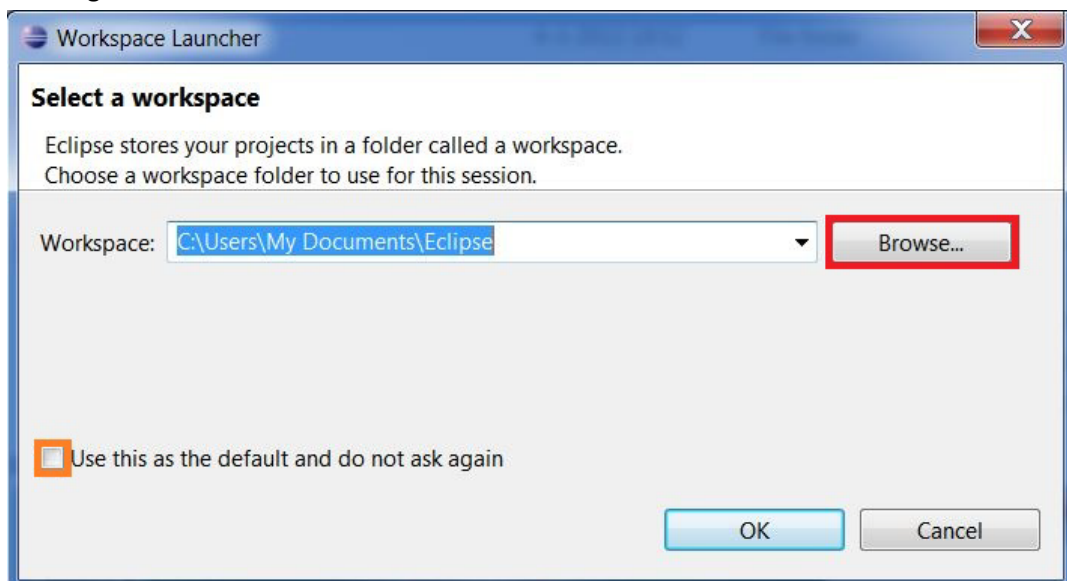


8. Eclipse will be extracted into the chosen destination folder. This will take a few minutes. **Close WINRAR** after it has finished.
9. **Open** the folder in which Eclipse is extracted. In it is one folder, named **eclipse**.

10. Open this folder and double click on **eclipse.exe** (highlighted in red). For easy future use: make a shortcut to eclipse.exe that you can put on your desktop.

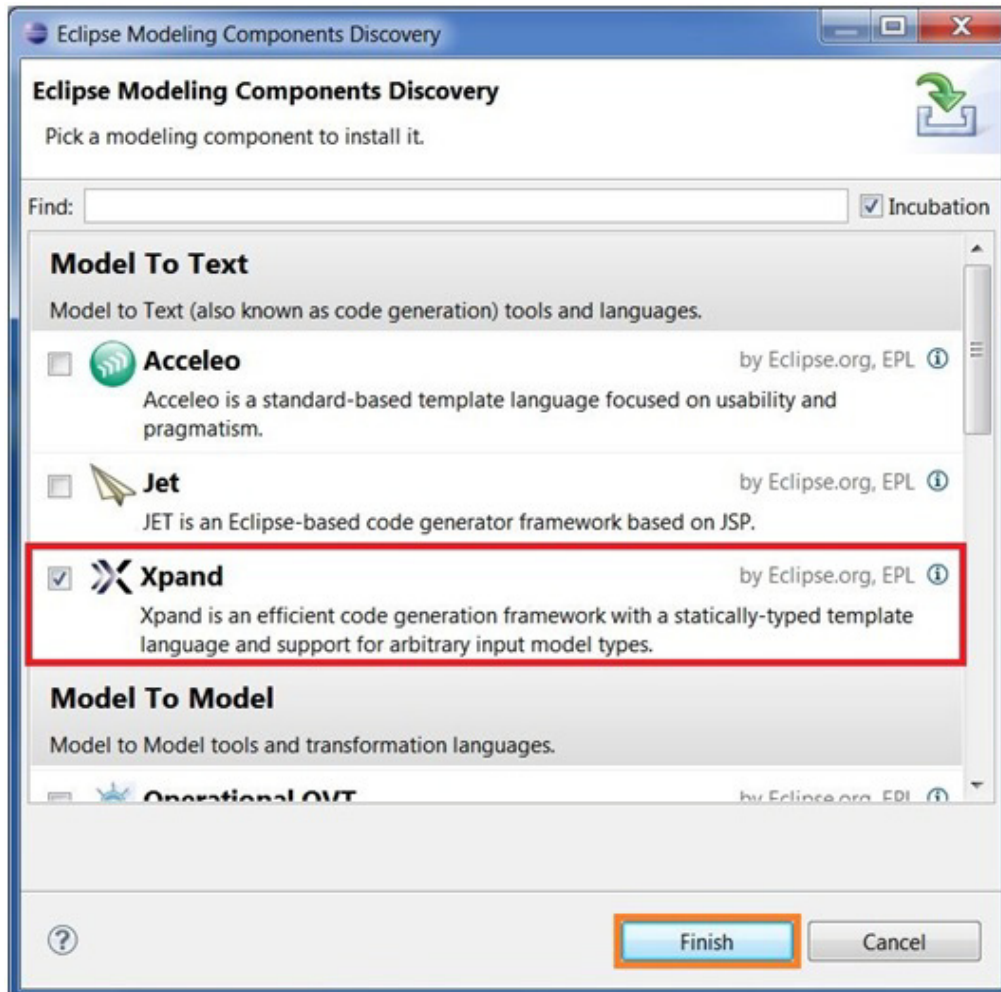
Name	Date modified	Type	Size
configuration	4-3-2013 19:54	File folder	
dropins	25-2-2013 0:19	File folder	
features	4-3-2013 19:53	File folder	
p2	4-3-2013 19:52	File folder	
plugins	4-3-2013 19:53	File folder	
readme	25-2-2013 0:19	File folder	
.eclipseproduct	4-2-2013 12:25	ECLIPSEPRODUCT ...	1 KB
artifacts.xml	4-3-2013 19:54	XML Document	281 KB
eclipse.exe	4-2-2013 13:05	Application	312 KB
eclipse.ini	4-3-2013 19:54	Configuration setti...	1 KB
eclipsesec.exe	4-2-2013 13:05	Application	24 KB
epl-v10.html	2-2-2012 8:31	Firefox HTML Doc...	16 KB
notice.html	2-2-2012 8:31	Firefox HTML Doc...	10 KB

11. Eclipse will open and ask where its workplace should be. Use browse (red) to **select a folder**, for example a folder called Eclipse in your Documents folder. Also **check** the box highlighted in orange. Click **OK**.

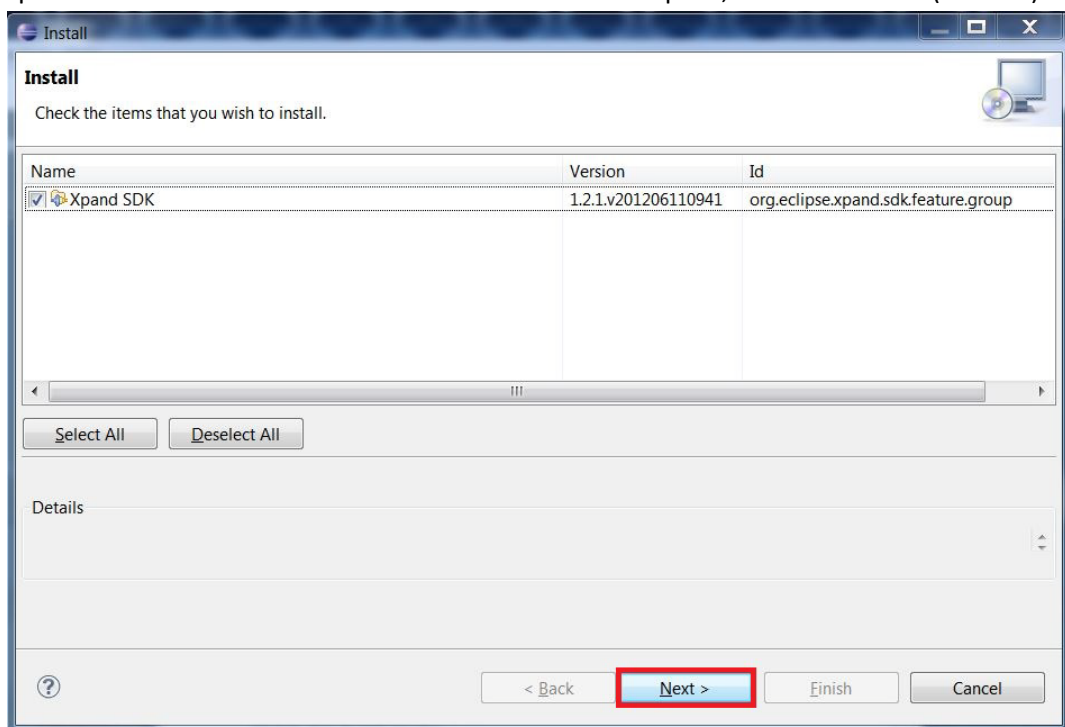


12. Eclipse is now setup and working. Now another modelling component has to be installed. From the **Help** menu, select **Install Modeling Components**. Eclipse will now connect to the internet to discover modelling components (a firewall might ask for permission). A window

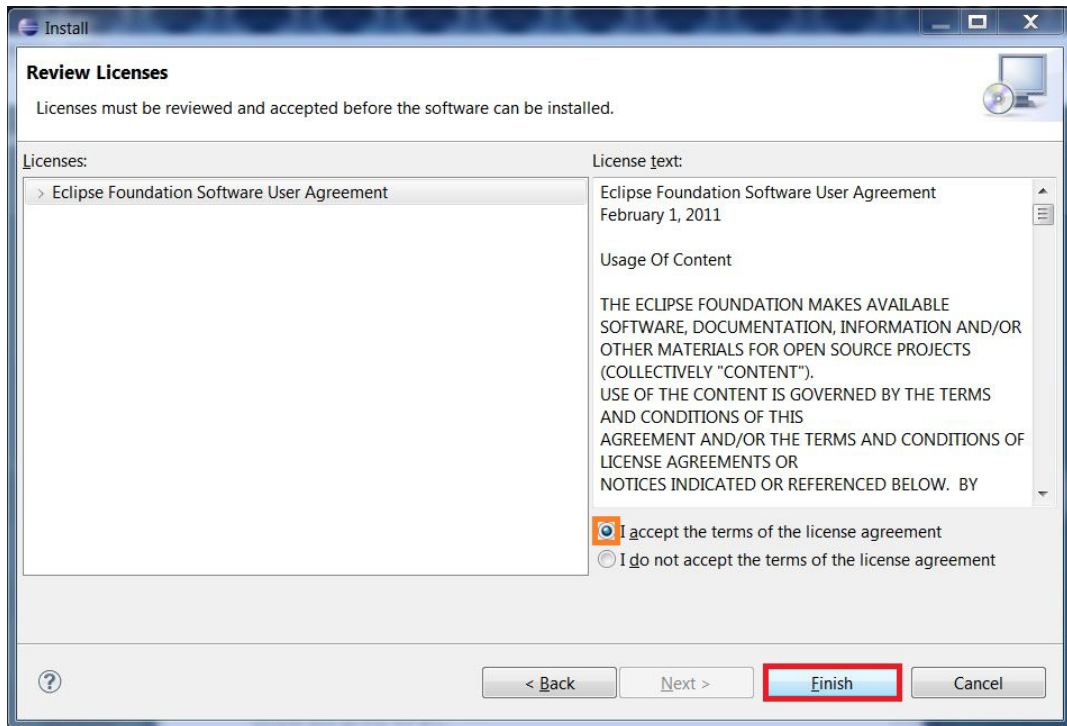
will open. **Check** the box in front of **Xpand** (red) and click **Finish** (orange).



13. Xpand will now be installed. When the window below opens, click **Next twice** (red box).



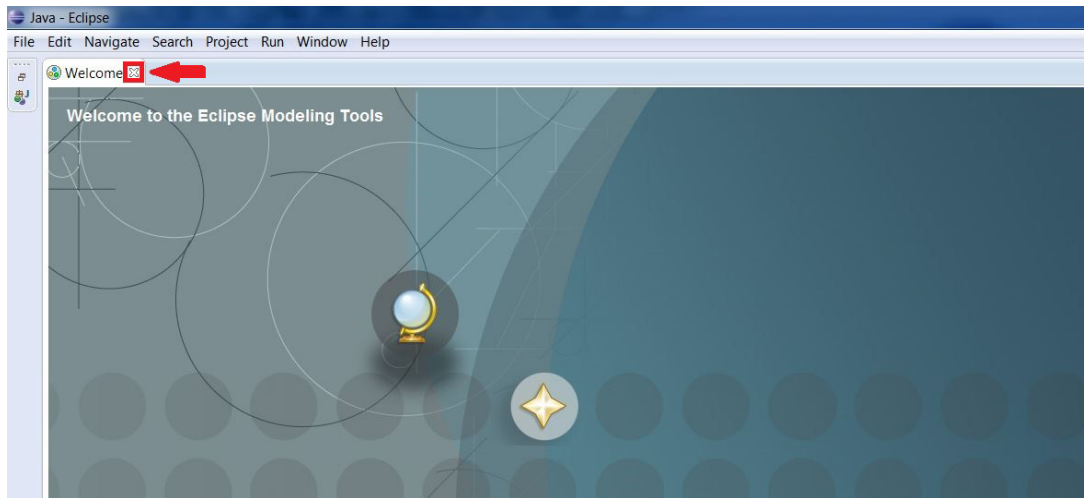
14. **Accept** the license agreement (orange box) and click **Finish** (red box).



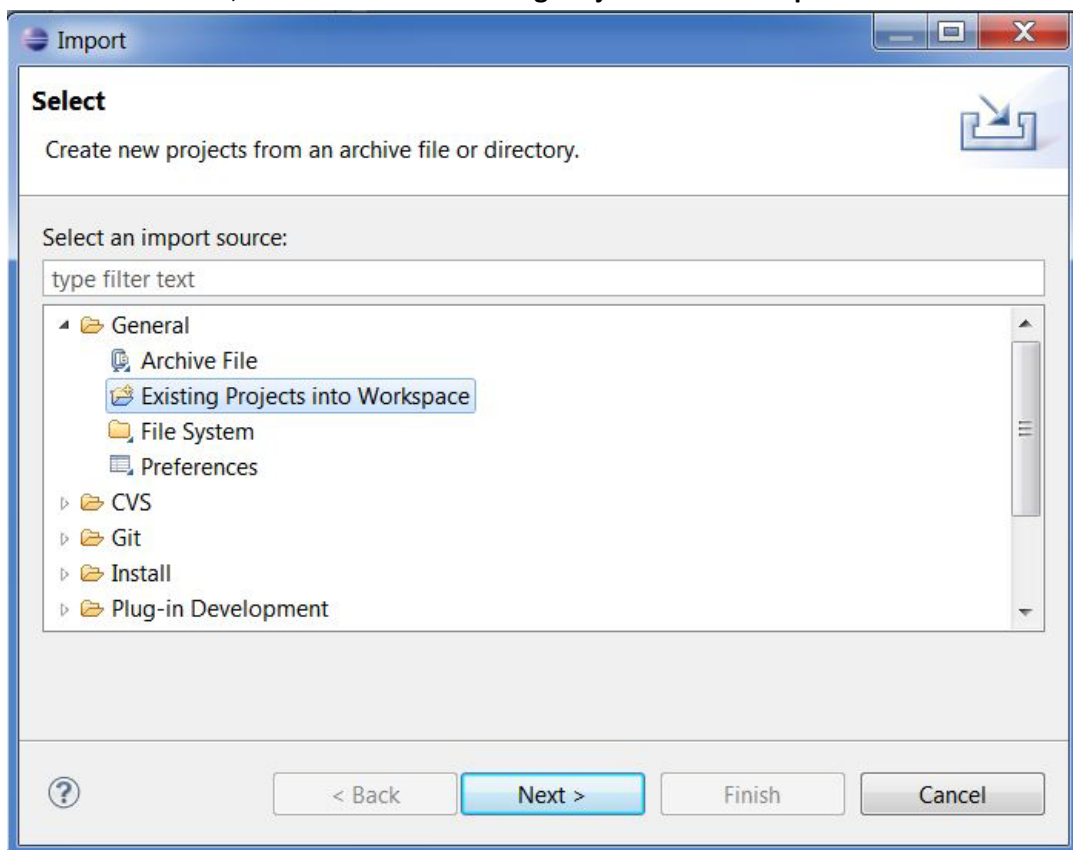
15. Eclipse asks to restart for the changes to take effect. Click **Yes**. After the restart, both Eclipse and Xpand are installed.

E2. Import tool in Eclipse

1. **Extract** the .rar-file called **UniSim User Unit Operation.rar** to a folder of your choice (you can use the program WINRAR again).
2. **Open Eclipse**. If you see a welcome screen, close it (red box).

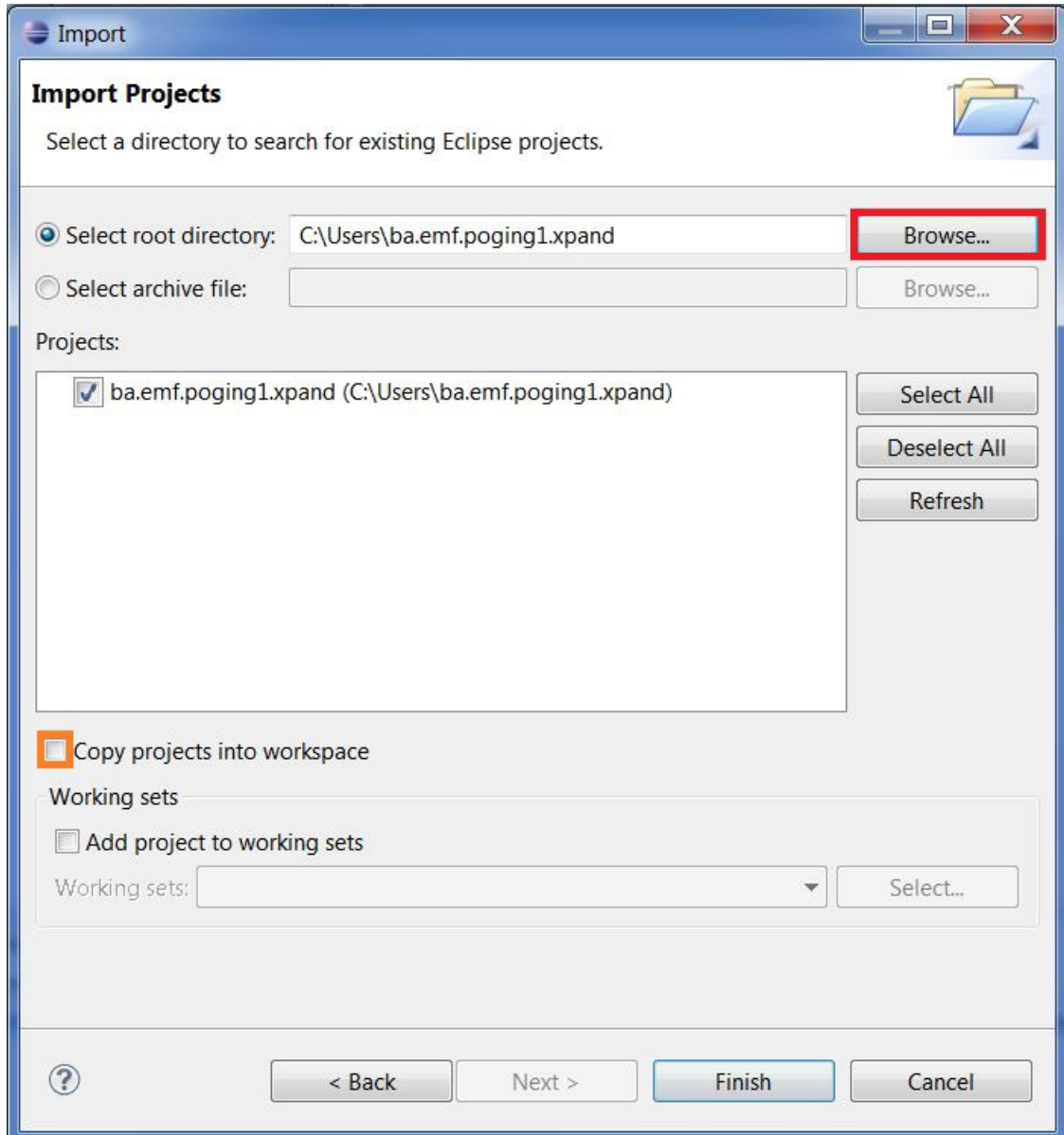


3. In the **File** menu, select **Import**.
4. In the new window, select **General -> Existing Projects into Workspace**. Click **Next**.



5. Use the **Browse** (red) behind Select root directory to select the folder in which you **extracted UniSim User Unit Operation.rar** (step 1). In that folder is a folder called **ba.emf.poging1.xpand**. Select it. If you want to **copy** the folder ba.emf.poging1.xpand into

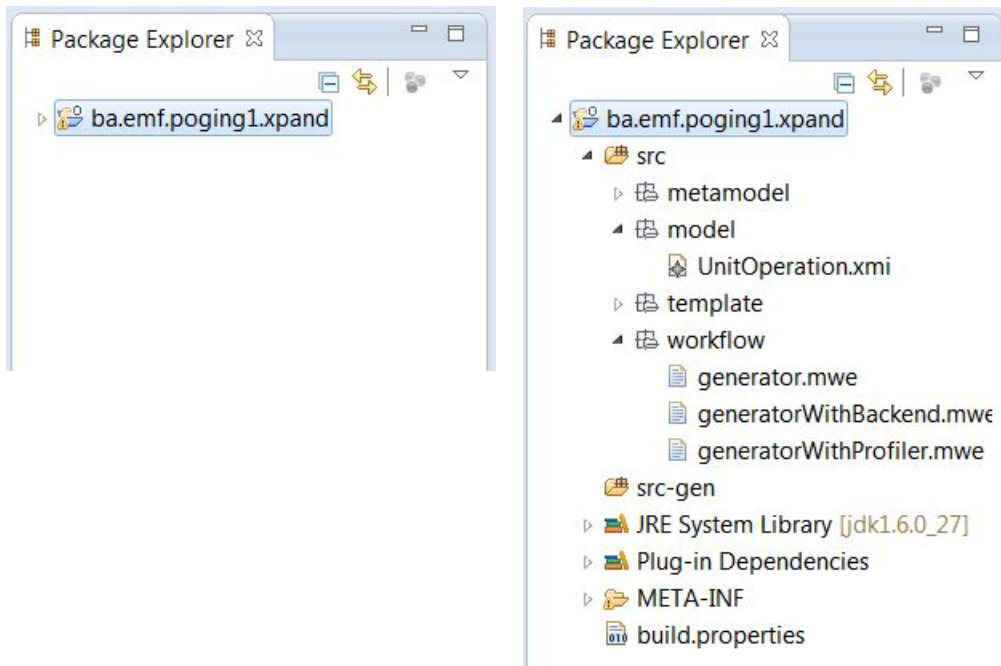
the workspace, **check** the orange box (not necessary for correct operation).



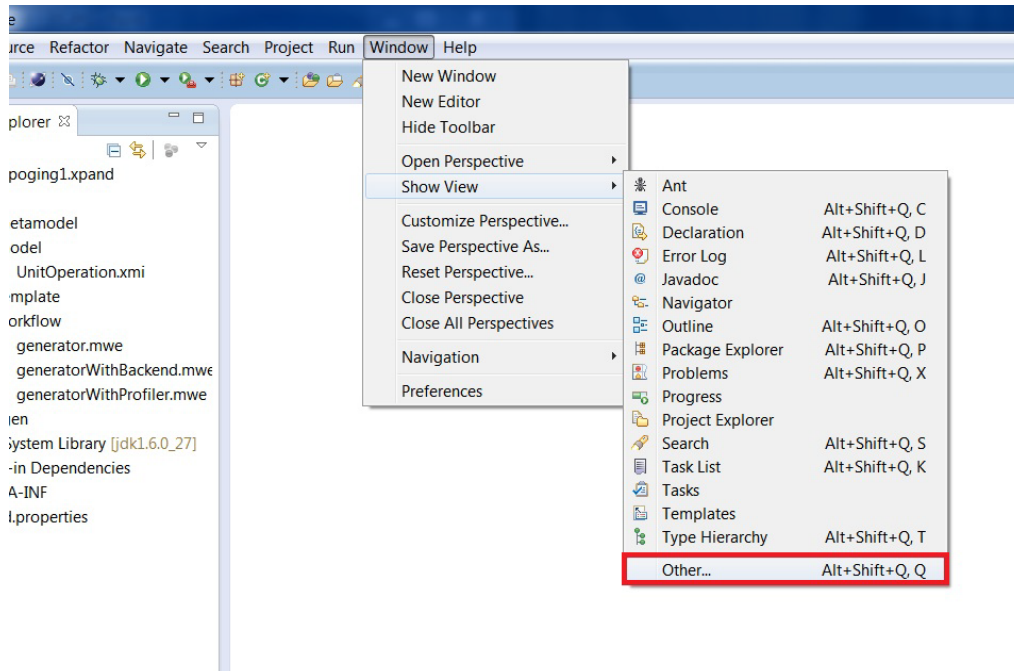
6. Click **Finish**. The tool is now imported in Eclipse and ready to use.

E3. Configuring view in Eclipse

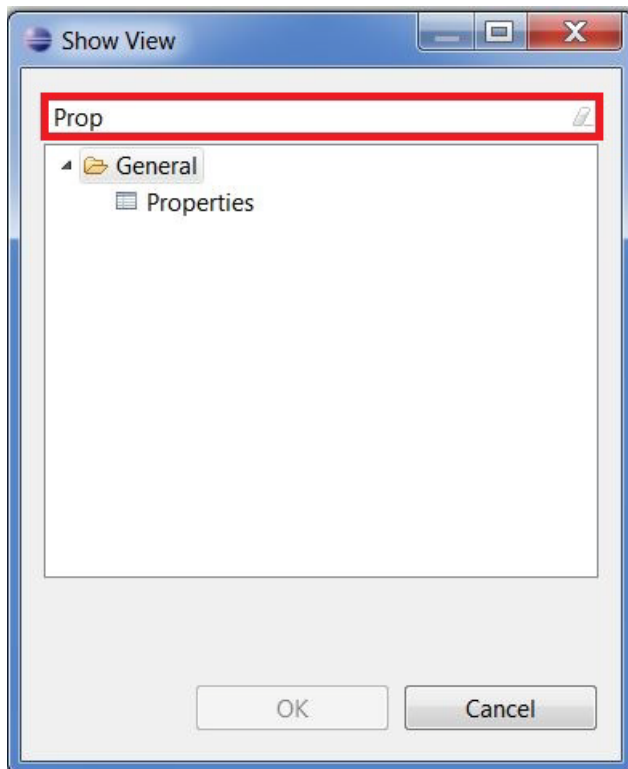
1. The **Package Explorer** (left side of the screen) will now look like the figure on the left. Click the little white triangles to make it look like the right picture.



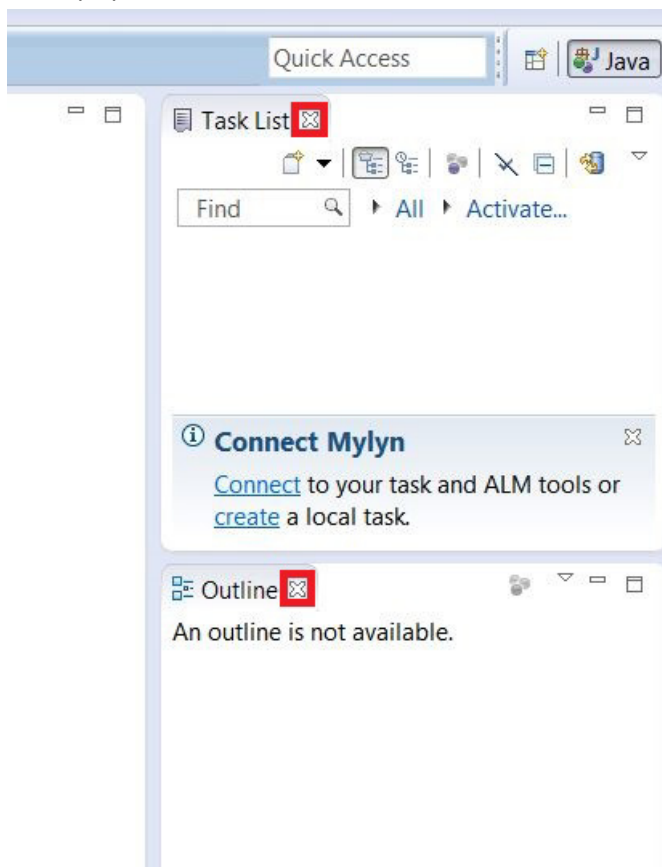
2. In the **Window** menu, go to **Show View** → **Other** (red box)



3. In the new window, begin to type **Properties** in the red box, or click General → Properties. Select Properties and click on **OK**.



4. Lastly, close Task List and Outline on the right side of the screen (red boxes), since they only take up space.



5. Now we have all the windows we need and we can start to use the tool.

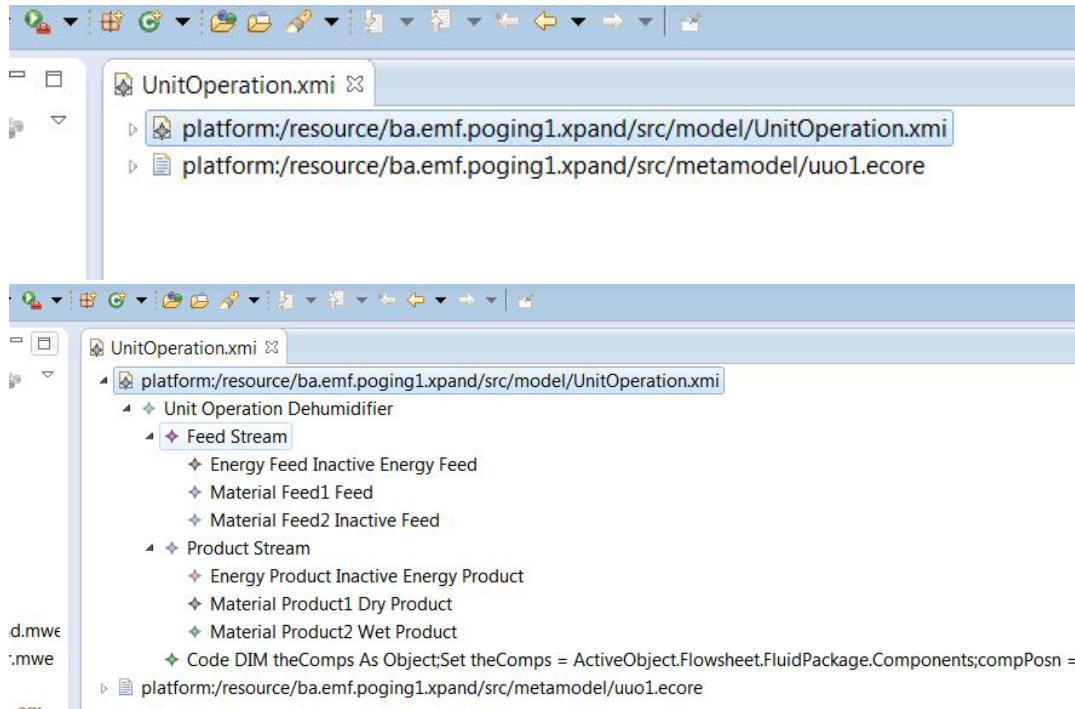
E4. Using the tool

There are two files that are important. These are:

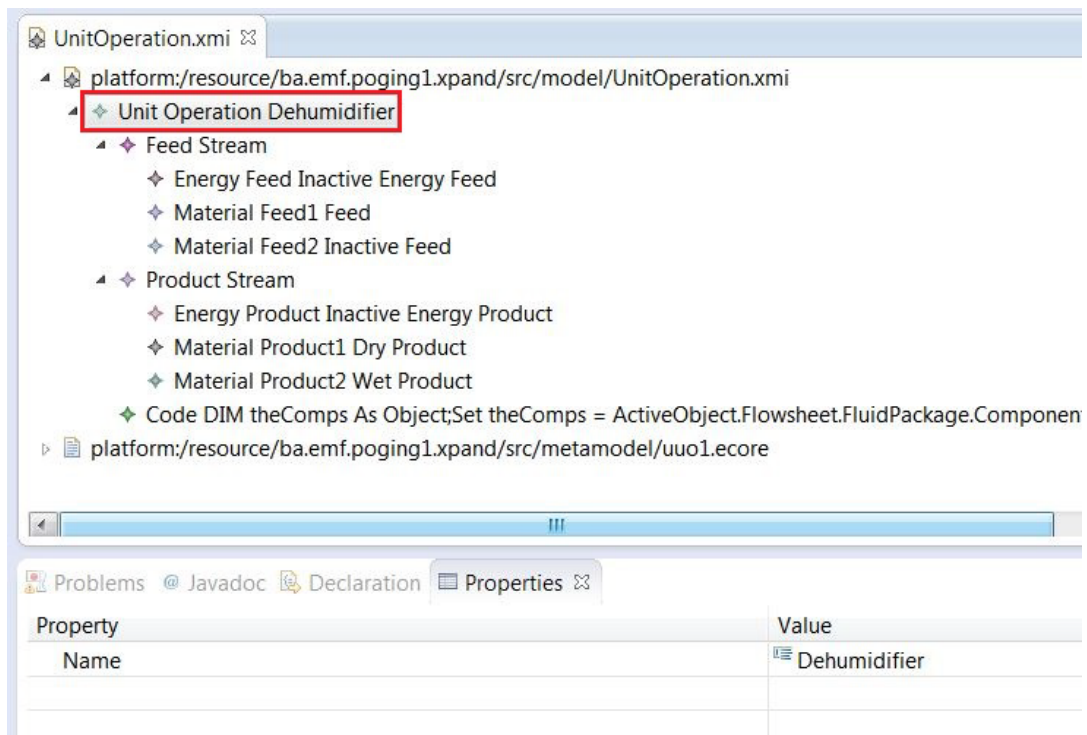
- i. src → model → UnitOperation.xmi (defines the model)
- ii. src → workflow → generator.mwe (generates the code)

I. Define the model

1. Double click **UnitOperation.xmi** in the Package Explorer. Eclipse will now look like the first picture below. Click the little white triangles until it looks like the second picture below.



2. The unit operation that is modelled here is a dehumidifier. All its characteristics can be changed in the Properties view (at the bottom). For example, when you click Unit Operation Dehumidifier (red box), you will see that it has one property, called Name. The value of this property is Dehumidifier. The **values** of all properties can be **changed** after they are clicked once.



3. Some characteristics, like Feed Stream and Product Stream do not have any properties. Therefore, when they are clicked, the Properties view will be empty.
4. Other characteristics have two properties. For example one called Name and one called Active. The Active property can only have two values, namely true or false. If the value is true, the feed or product stream in question is active, if the value is false, the feed or product is inactive.
5. There is also a characteristic called Code. Code has two properties, Error Code and Model Code. The values of these properties must be in **Visual Basic** code. The different lines of code must be **parted by a semicolon**. For example the code

```
Dim CMFs As Variant
CMFs = feed.ComponentMolarFlowValue
CompFlow = CMFs(compPosn)
```

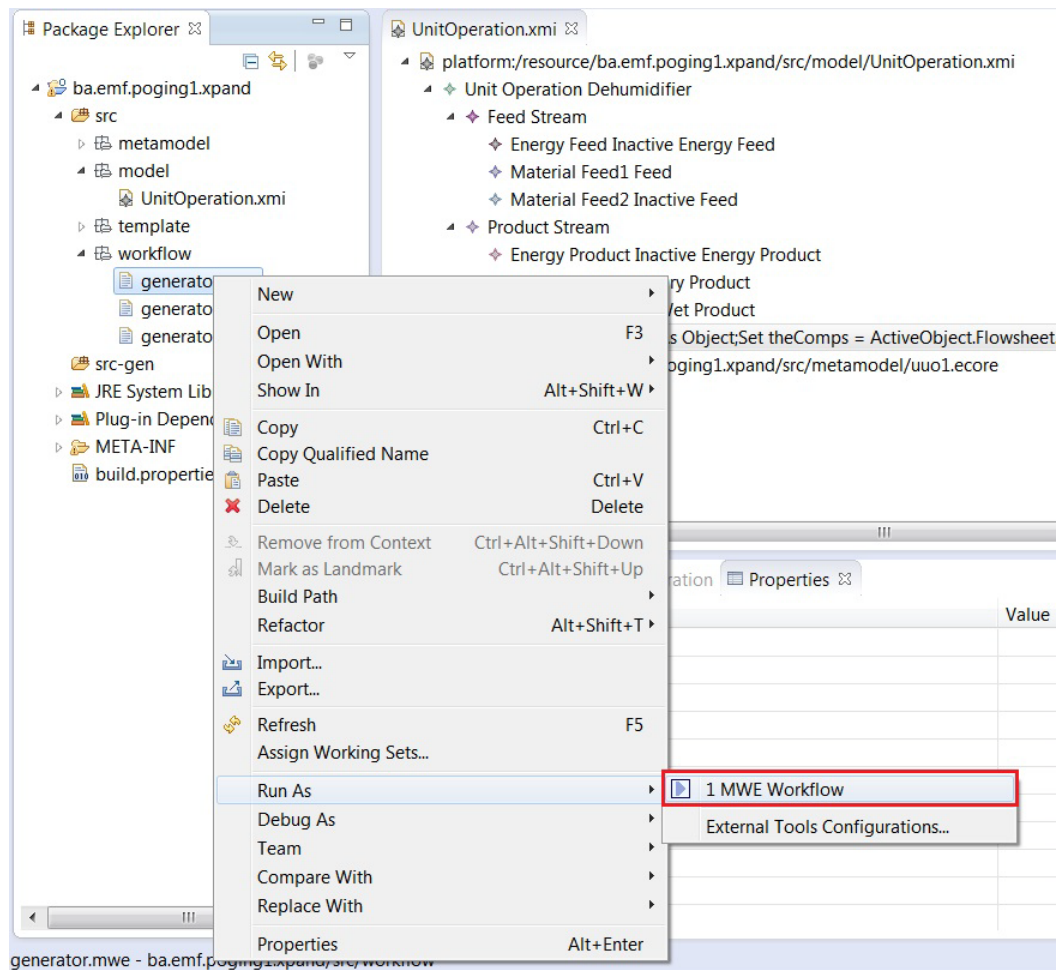
becomes

```
Dim CMFs As Variant;CMFs = feed.ComponentMolarFlowValue;CompFlow = CMFs(compPosn);
```

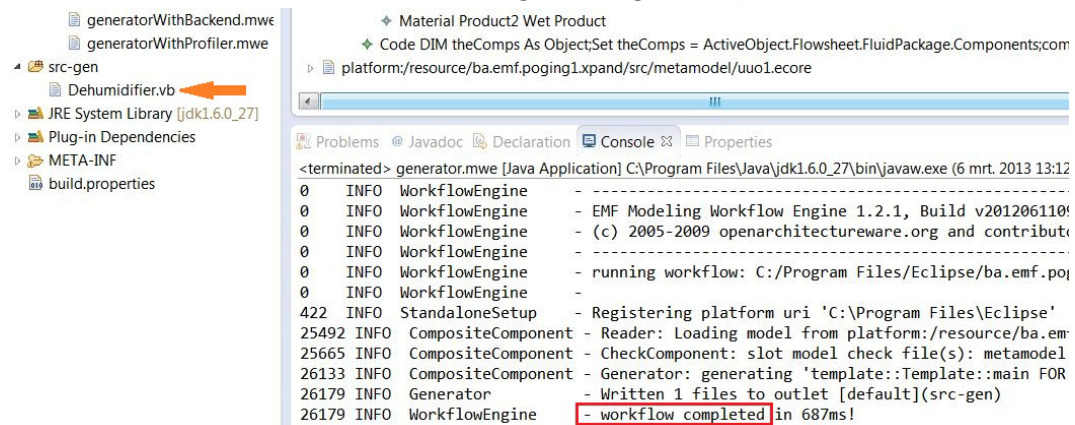
The easiest way to do this is by programming in **Notepad or Notepad++**, and exchanging the Enters for semicolons after you finished coding. The single line of code can then be pasted into the value of the property. The last paragraph of this appendix (E5 Coding Tips) gives some information and tips about the coding.
6. If all the property values are defined, the model is ready. Now the code can be generated.

II. Generate the code

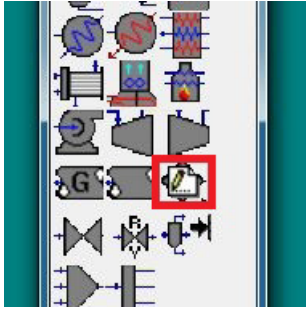
1. The code is generated by the file **generator.mwe** in src → workflow.
2. Click the file generator.mwe with your **right mouse-button**. Go to **Run As** and pick **MWE Workflow** (red box).



3. Your unit operation will now be generated. Depending on your machine, it could take a while. When the last line of the console reads 'workflow completed' (red box), you will see a file with the extension .vb in the folder src-gen (orange arrow).

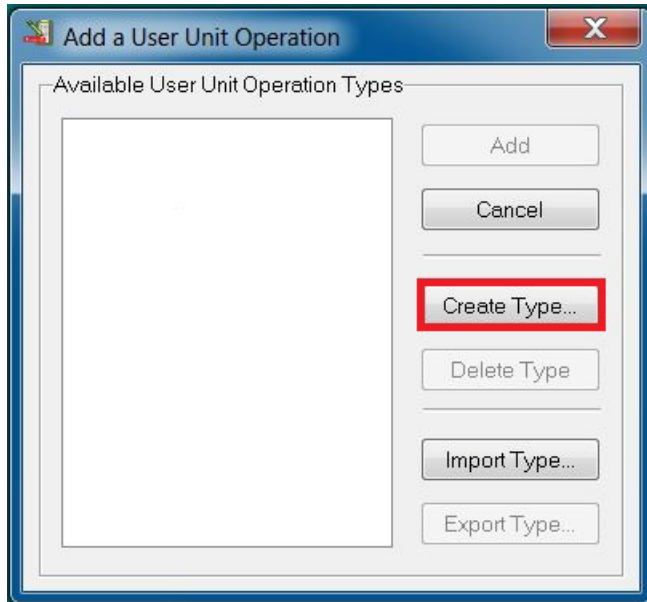


4. Open the file (in this case Dehumidifier.vb) and **copy** the complete content.
5. **Open UniSim Design** and open or create a case.
6. To add a new User Unit Operation, open the Object Pallet (press **F4**). Click the User Ops icon (red box) from the Object Palette. A small window with the same icon will appear.

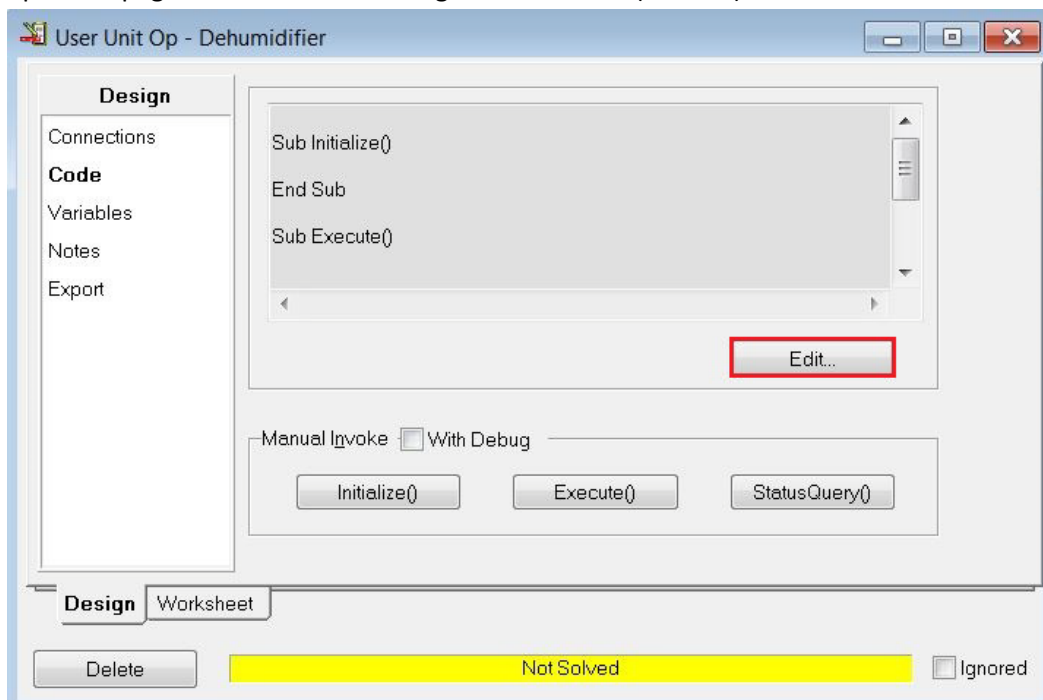


7. **Double click** on the icon in the new window.

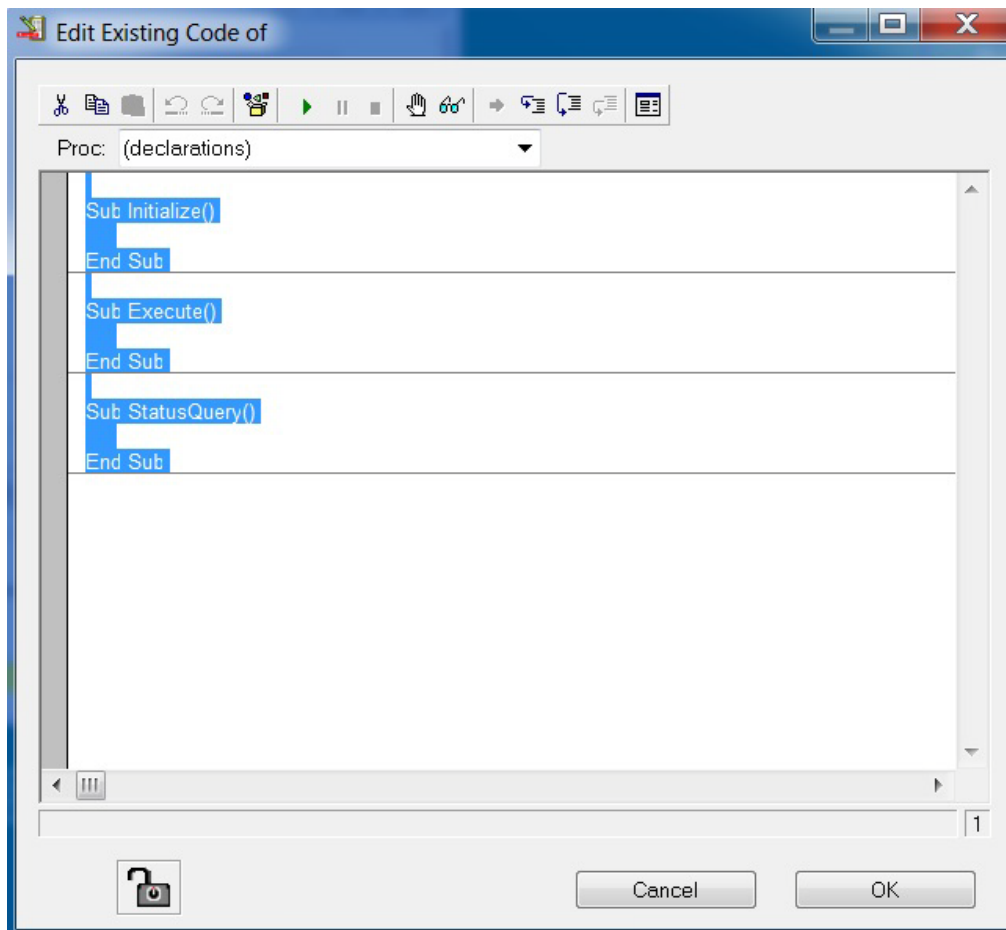
A new window will appear, looking like the figure below. Click on **Create Type** (red box).



8. Enter the **name** of the user operation in the window and click on **OK**.
9. The property view of the new user operation will appear.
10. Open the page **Code** on the tab Design and click **Edit** (red box).



11. Select and **delete** the **existing code**. **Paste** the code from the generated **.vb-file**.



12. Click **OK**. The unit operation can now be used like any other UniSim unit operation.

E5. Coding Tips

This paragraph gives several tips to make coding as easy as possible.

Names

The names of the feed and product streams are established in the transformation template. Therefore, when feed or product streams are mentioned in the modelCode or errorCode, their names are as follows:

Stream	name
Mandatory Feed Stream	feed
Second Feed Stream	feed2
Energy Feed Stream	enFeed
Mandatory Product Stream	prod
Second Product Stream	prod2
Energy Product Stream	enProd

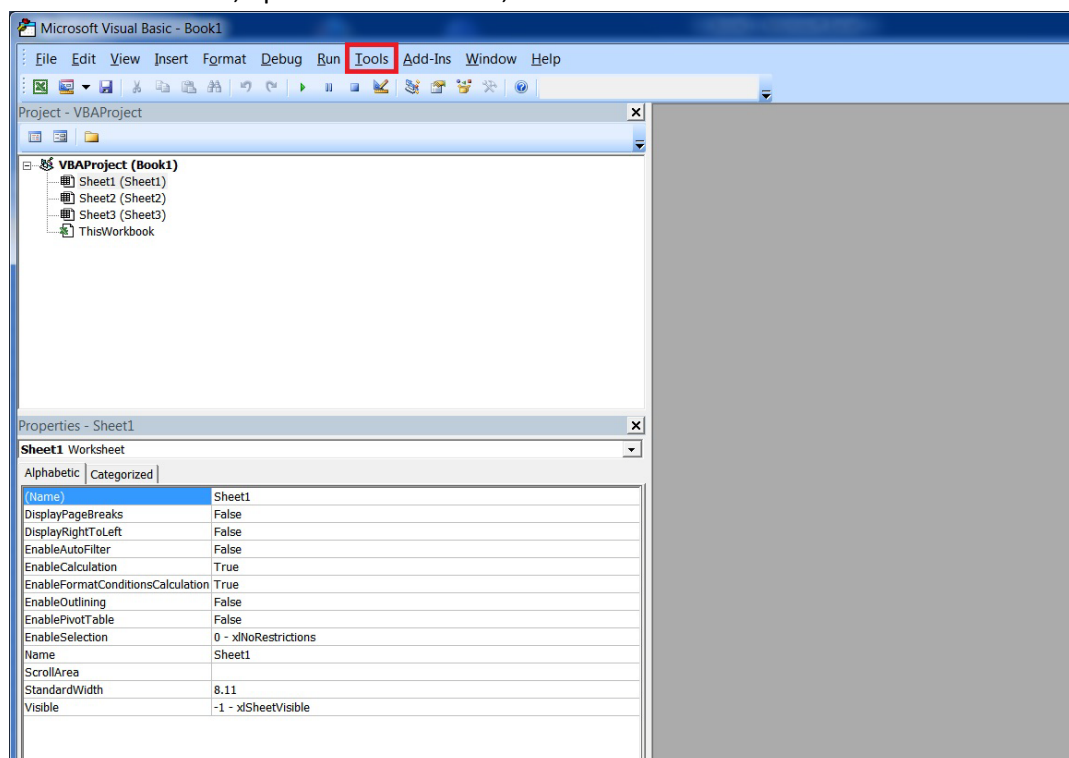
Other names that are already established:

Description	name
The components of the current fluid package	theComps

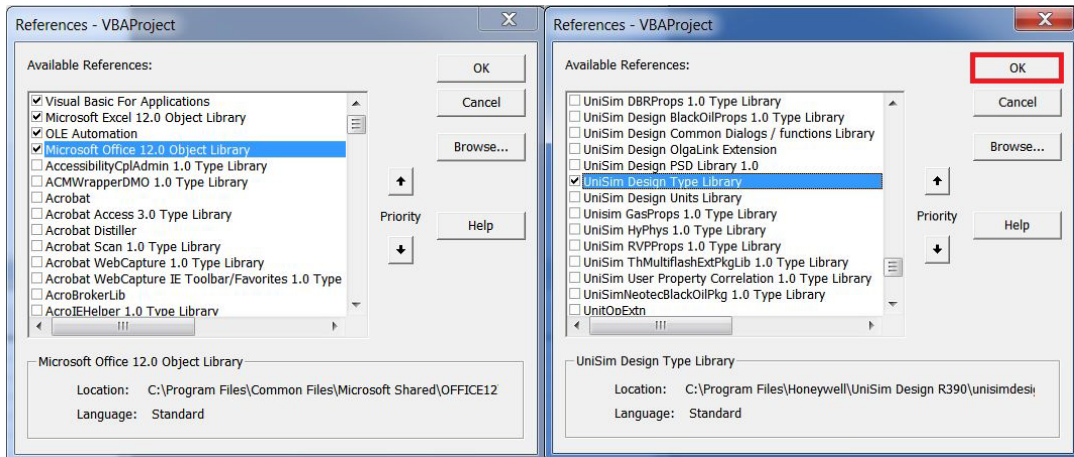
Type Library

All the methods that can be used in the code can be found in a library. You reach this library as follows:

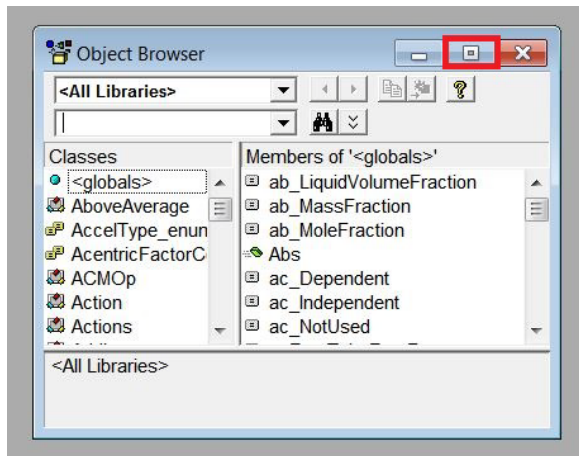
1. Open Microsoft Office **Excel**.
2. Press **Alt+F11**.
3. In the new window, open the **Tools** menu, click **References**.



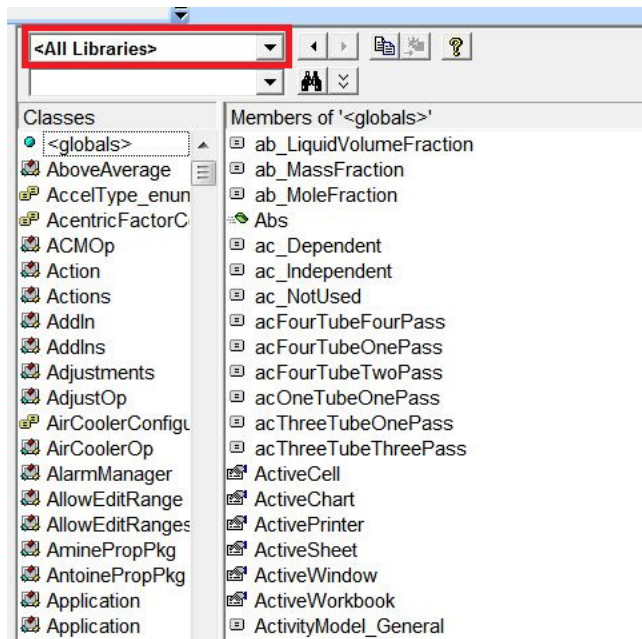
- A new window will appear (left). Scroll down to **UniSim Design Type Library** and check the box in front of it (right). Click **OK** (red box).



- Press **F2**. The object browser will open. **Maximise** it (red box).



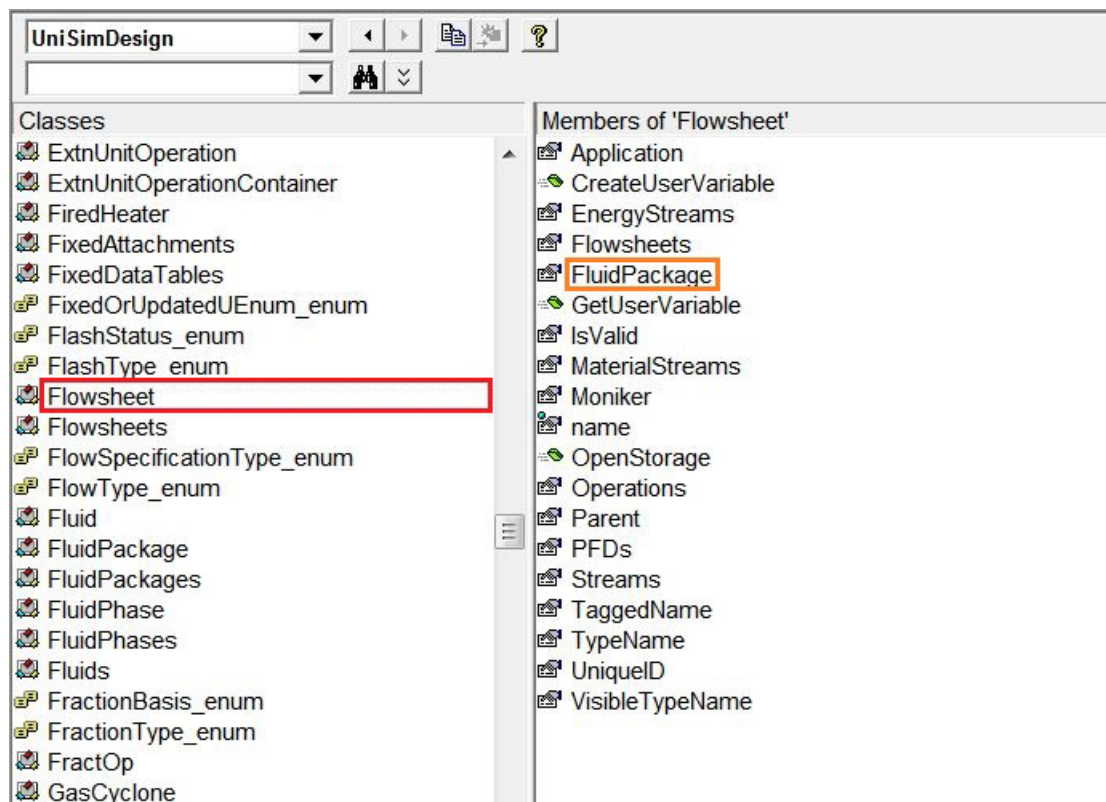
- In the red box, select **UniSimDesign**.



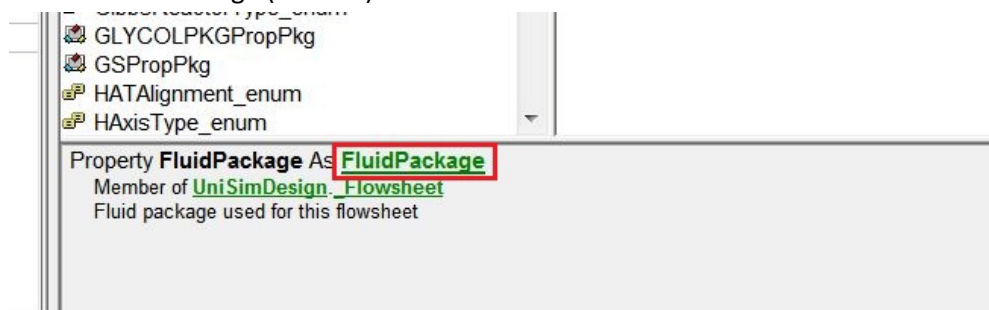
- You can now browse the type Library.

Navigating through type Library

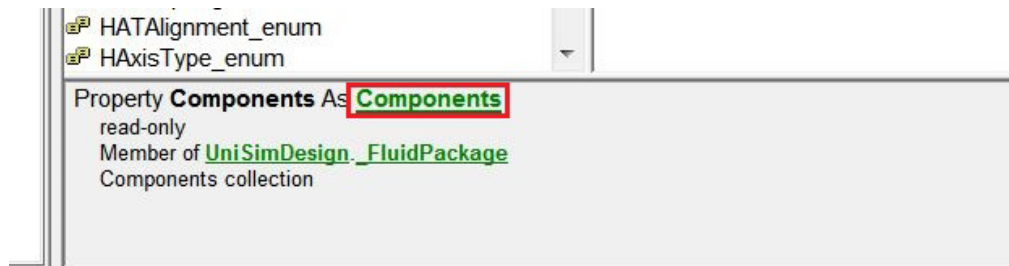
1. Complete the steps shown above.
2. This example uses a line of code from the dehumidifier template as an example. This line is: `Set theComps = ActiveObject.Flowsheet.FluidPackage.Components`. With the type library you can find out which methods you can use with a particular object. The object in this case is theComps. As the name suggests, this object contains the components of the current fluid package.
You can navigate through the library to this function like this:
3. Select **Flowsheet** in the classes section (red box). Make sure that you select Flowsheet and not Flowsheets!



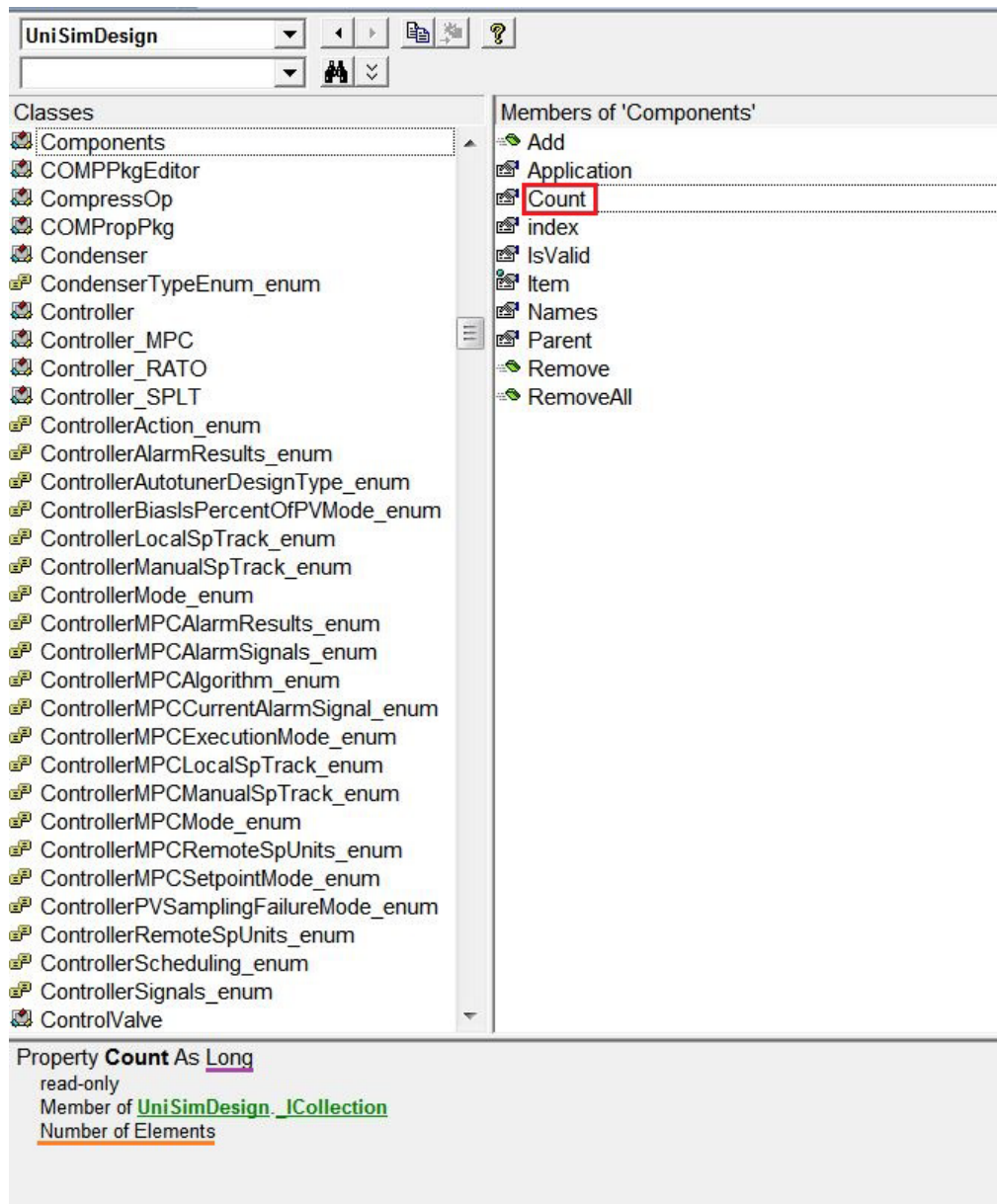
4. Flowsheet has a number of Members, shown on the right. In our example the next class is FluidPackage. Therefore, click on **FluidPackage** (figure above, orange box).
5. At the bottom of the window you will now see an explanation for the class FluidPackage. Click on FluidPackage (red box).



6. Now you are in the class FluidPackage and you can see its members. The next class of the example is Components, so click on Components in the Members section. Repeat step 5, but for the class Components (see below).

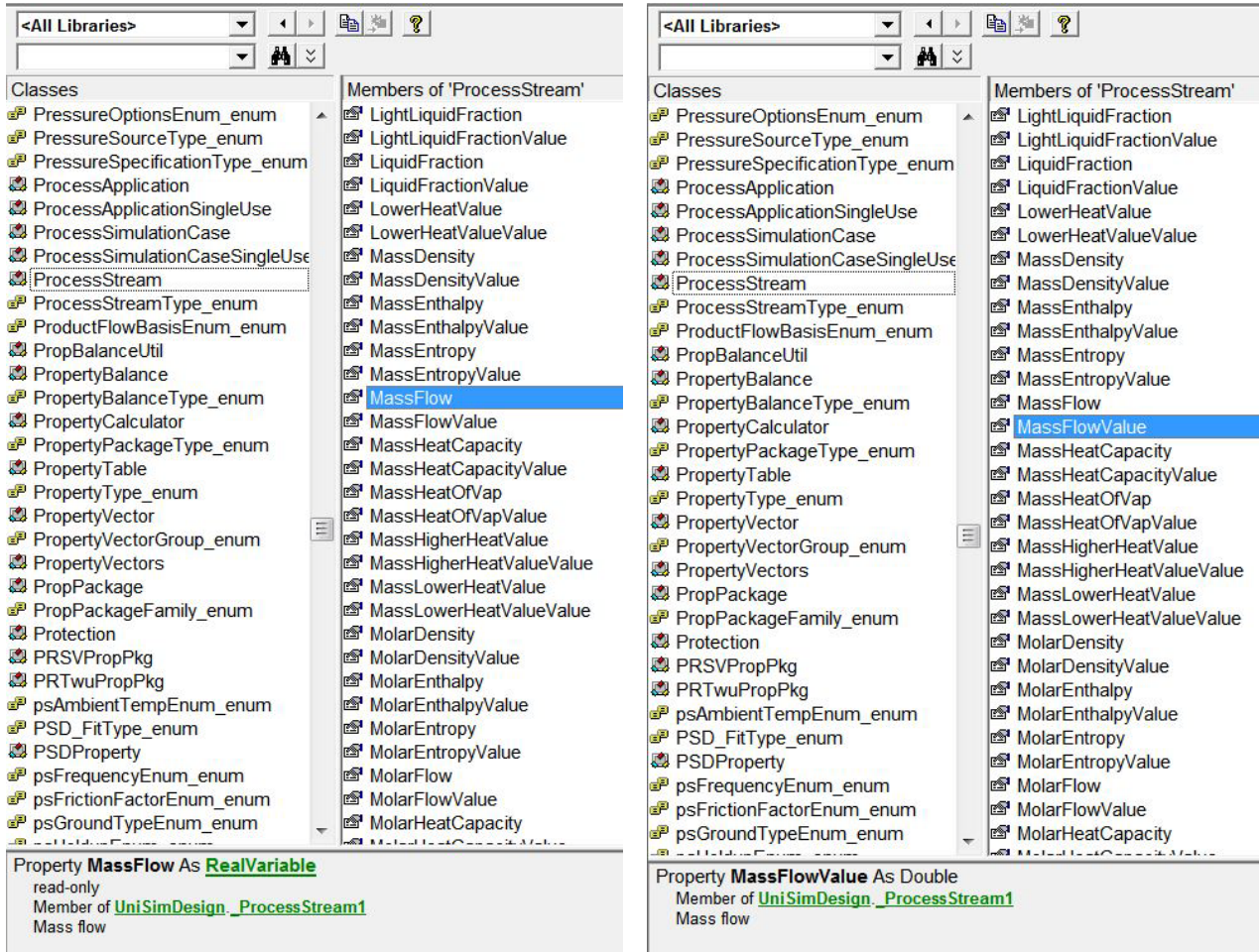


7. Now you can see that Components has 10 members. One of those members is index. In the code, there is a line `compPosn = theComps.index("H20")`. When index in the Members section is clicked, we get the information that index returns a Long (a number). Another line of code is `For i=0 To theComps.Count-1`. Here the method Count is used. When Count is clicked in the Members section (red box), we get the information that Count returns the number of Elements (orange) as a Long (purple).



Properties in the library

The properties of the streams can be accessed through the class `ProcessStream`. Most properties in `ProcessStream` have two members. For example `MassFlow`. There is a member called `MassFlow` and one called `MassFlowValue`. `MassFlow` gives the actual mass flow, the property itself. `MassFlowValue` gives the value of the property, thus the value of `MassFlow`.



You can see here that the `MassFlow` is a `RealVariable`, and the `MassFlowValue` is a `Double` (a number).

An example where this is used is the line `prod.Pressure.Calculate(feed.PressureValue)`.

The `Pressure` (real variable) of the product stream is calculated with the value of the feed stream pressure.

Arrays/Lists

In computer science, the first index number of an array or list is 0. This makes the last index of an array or list the length of the array/list -1. You can see an example in the following lines of code:

```
For i=0 To theComps.Count-1
    ' do something
Next i
```

If the array `theComps` has 7 indices, `theComps.Count` equals 7.

The first index of `theComps` is 0 and the last is 6.