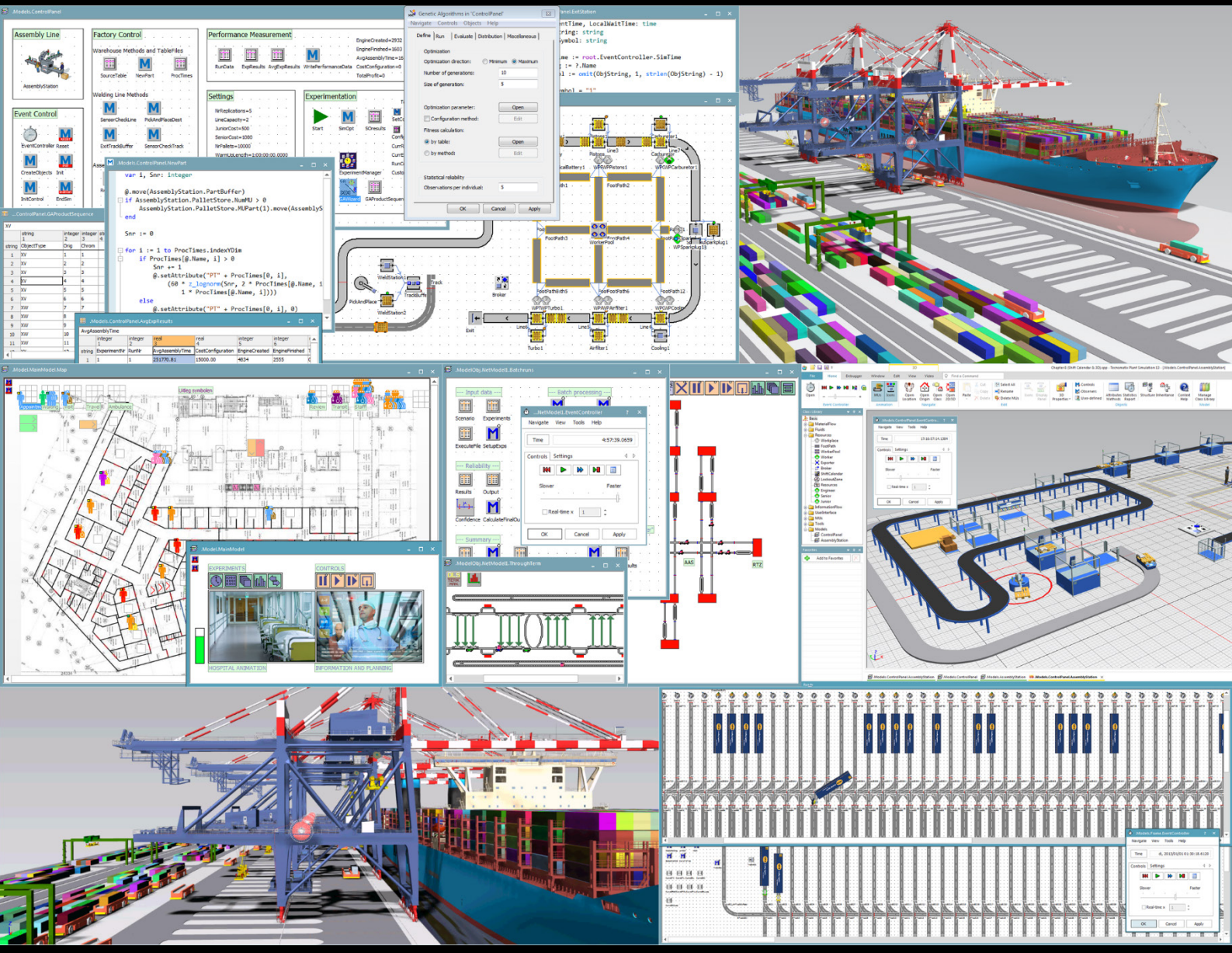
Simulation Modelling using Practical Examples: A Plant Simulation Tutorial

SOFTWARE VERSION 13.0 – LAST UPDATE 17/10/2017



Martijn R.K. Mes

UNIVERSITY OF TWENTE.

Simulation Modelling using Practical Examples: A Plant Simulation Tutorial

Dr.ir. M.R.K. (Martijn) Mes
Associate Professor
University of Twente
Faculty of Behavioural, Management and Social sciences
Department of Industrial Engineering and Business Information Systems
P.O. Box 217
7500 AE Enschede
The Netherlands

Phone: (+31 53 489)4062
E-mail: m.r.k.mes@utwente.nl
Web: http://www.utwente.nl/bms/iebis/staff/mes/

UNIVERSITY OF TWENTE.

CONTENTS

1 Introduction

Simulation modelling is an excellent tool for analysing and optimizing dynamic processes. Specifically, when mathematical optimisation of complex systems becomes infeasible, and when conducting experiments within real systems is too expensive, time consuming, or dangerous, simulation becomes a powerful tool. The aim of simulation is to support objective decision making by means of dynamic analysis, to enable managers to safely plan their operations, and to save costs.

1.1 What is Simulation?

In the assignments throughout this tutorial, we rely on the theory and terminology as presented in the frequently used simulation books from Robinson (2014) [1] and Law (2015) [2]. Robinson (2014) defines simulation as:

> *Experimentation with a simplified imitation (on a computer) of an operations system as it progresses through time, for the purpose of better understanding and/or improving that system.*

And Law (2015) states:

> *In a simulation we use a computer to evaluate a model numerically, and data are gathered in order to estimate the desired true characteristics of the model.*

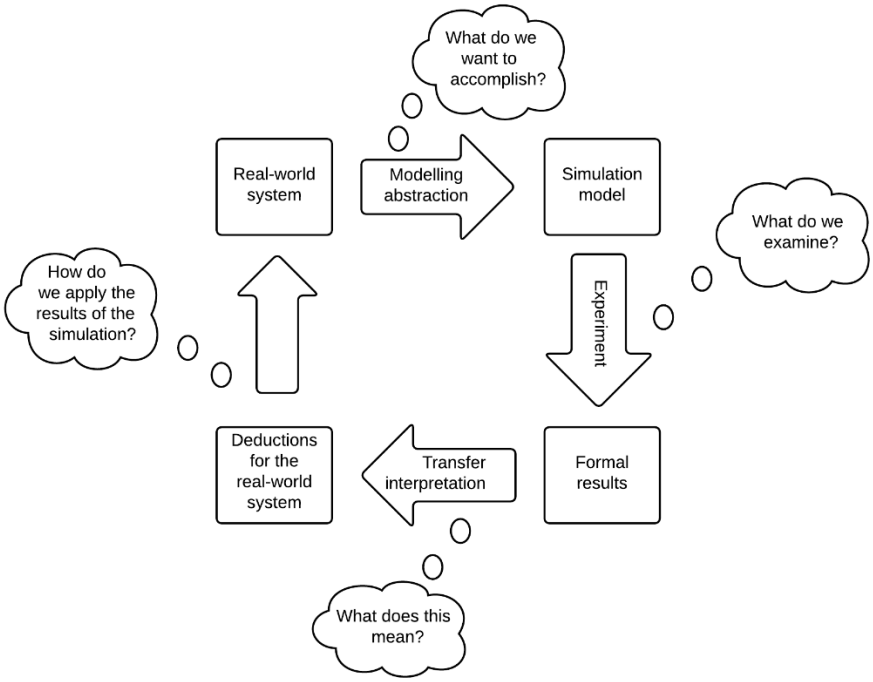
Simulation aims to achieve results that can be transferred to a real world installation. In addition, simulation defines the preparation, execution, and evaluation of carefully directed experiments within a simulation model. As a rule, you will execute a simulation study using the following steps:

- You first check out the real-world installation you want to model and gather the data you need for creating your simulation model.
- You then abstract this real-world installation and create your simulation model according to the aims of the simulation studies.
- After this, you run experiments, i.e., execute simulation runs with your simulation model. This will produce a number of results, such as how often machines fail, how often they are blocked, which set-up times accrue for the individual stations, which utilisation the machines have, etc.
- The next step will be to interpret the data the simulation runs produce.
- Finally, management will use the results as a base for its decisions about optimizing the real installation.

Developing your simulation model is a cyclical and evolutionary process. You will start out with a first draft of your model and then refine and modify it to make use of the intermediary results the simulation runs provide. An illustration of this process can be found on the next page. Eventually, after several cycles, you will arrive at your final model. As a simulation expert, you must never lose sight of these questions:

[1] Robinson, S. (2014) Simulation: The Practice of Model Development and Use (2nd edn). Palgrave Macmillan.
[2] Law, A.M. (2015) Simulation Modeling and Analysis (5th edn). McGraw-Hill.

- What do you want to accomplish with the simulation study?
- What are you examining?
- Which conclusions do you draw from the results of the simulation study?
- How do you transfer the results of the simulation study to the real-world installation?



1.2 Time-Oriented Simulation versus Discrete Event Simulation

In the real world, time passes continuously. For instance, when watching a part move along a conveyor system, you will detect no leaps in time. The time the part takes to cover the system is continuous, such that the curve for the distance covered is a straight line.

A discrete event simulation (DES) program on the other hand only takes into consideration those points in time (events) that are of importance to the further course of the simulation. Such events may, for example, be a part entering a station, leaving it, or moving on to another machine. Any movements in between those events are of little interest to the simulation.

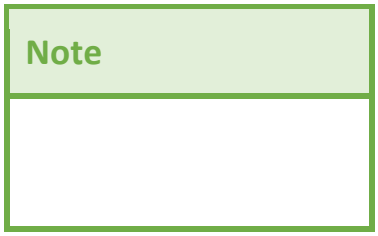
Plant Simulation uses DES. One major advantage of DES over time-oriented simulation (continuous or time-step simulation) is performance. Since the program can simply skip all the moments in time that are not of interest, it is possible to simulate years of factory operation in just minutes. That is particularly useful when you want to simulate different configurations of the same system, and make several replications for each configuration. Plant Simulation has built-in functionalities for exactly that purpose, which we will cover in Chapter 5.

1.3 Hints for Using the Tutorial

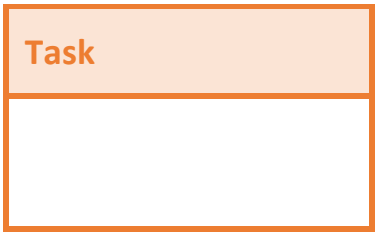
The examples in this tutorial are intended to get you started with Plant Simulation. The most important features of Plant Simulation are introduced and used in examples. However, do not expect an in-depth discussion of all topics, as these are covered in the Step-by-Step Manual from Siemens and the Plant Simulation help function.

Previous knowledge of the program is not required, as all examples are described in detail. To work your way through the examples and exercises you need to have Plant Simulation installed on your computer, or you need to have access to a computer running Plant Simulation.

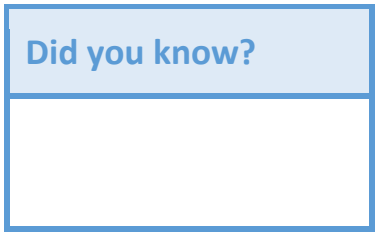
You will find three types of boxes throughout the tutorial:



Notes provide additional information that is worth keeping in mind while working with Plant Simulation. Pay particular attention to them in case you get stuck.



Tasks give step-by-step instructions on how to build models in Plant Simulation. You should aim to understand what is happening at every step.



These boxes reveal details and advanced functionality of Plant Simulation that are interesting to mention, but not needed for full comprehension. You can safely skip these if you are in a hurry.

1.4 Overview of the Tutorial

Chapter 2 provides an introduction to Plant Simulation and the basic building blocks that are provided. The remaining chapters 3 to 9 are divided into two parts. In Part A, chapters 3 to 5, you will build a number of basic models in Plant Simulation. Each of these chapters end with a working simulation model and an assignment. The simulation models presented in these chapters revolve around one running example: the modelling and optimisation of a General Practitioner's office.

- Chapter 3 involves building a simple model, using the concepts of branching, push-pull, and prioritisation along the way.
- Chapter 4 introduces advanced usage of *TableFiles*, which can store a great deal of information that is generated in a model, with an emphasis on verification and validation. This chapter also elaborates on the use of random numbers.
- Chapter 5 lets you build a more complex model that implements appointments and uses the *ExperimentManager* to carry out a number of experiments with several replications.

Plant Simulation can be used to model many types of real world systems, such as hospitals, factories, computer networks, transportation networks, airports, etc. Moreover, the program supports numerous advanced concepts, such as workers and assembly lines. In Part A you only used the basic functionalities of Plant Simulation. Chapter 6 provides a preview of the more advanced Plant Simulation concepts, which are presented in more detail in Part B of this tutorial.

In Part B, chapters 7 to 9, you will build more advanced and more graphically oriented simulation models. Again, each of these chapters contain an assignment. However, we use another running example throughout these chapters, namely of a car manufacturer.

- Chapter 7 involves building a simplified model of a car manufacturer, thereby introducing the concept of *Frames* to create your own building blocks and the use of debugging.
- Chapter 8 introduces the use of *Lines* and *Workers* to model a more realistic manufacturing environment. The chapter ends with a 3D simulation model of the car manufacturer.
- Chapter 9 introduces the topic of *Simulation Optimisation*, where you systematically perform experiments to find the best settings of the factory.

2 Overview of Plant Simulation

Plant Simulation is software for integrated, graphic and *object-oriented* modelling, simulation, and animation. Many complex systems may be modelled and displayed in great detail closely resembling reality.

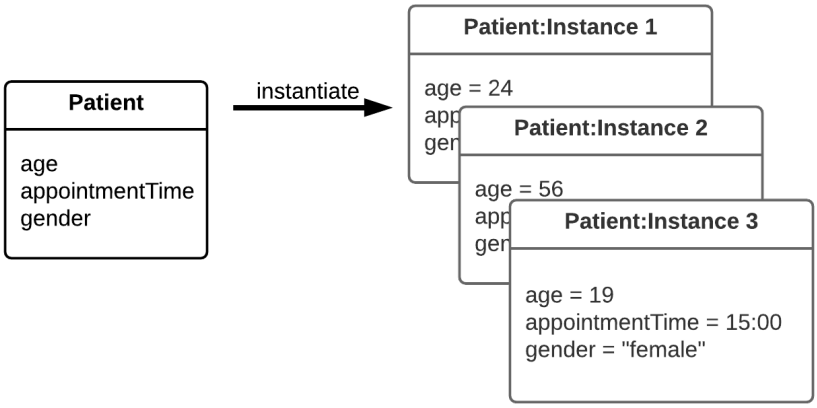
2.1 Object Orientation

Plant Simulation is completely *object-oriented*. Understanding the basic principles of object orientated programming enables you to model large, complex systems in an organised and maintainable way.

Classes, Attributes, and Instances

As an example, suppose we want to model the patients in a system. The relevant properties of a patient are the patient's age, appointment time, and gender. We do not care about the exact age, appointment time and gender yet, but we only recognise that the model of a patient should have these properties. In *object-oriented* design terms, we have the *class Patient*, which has the *attributes age*, *appointmentTime*, and *gender*.

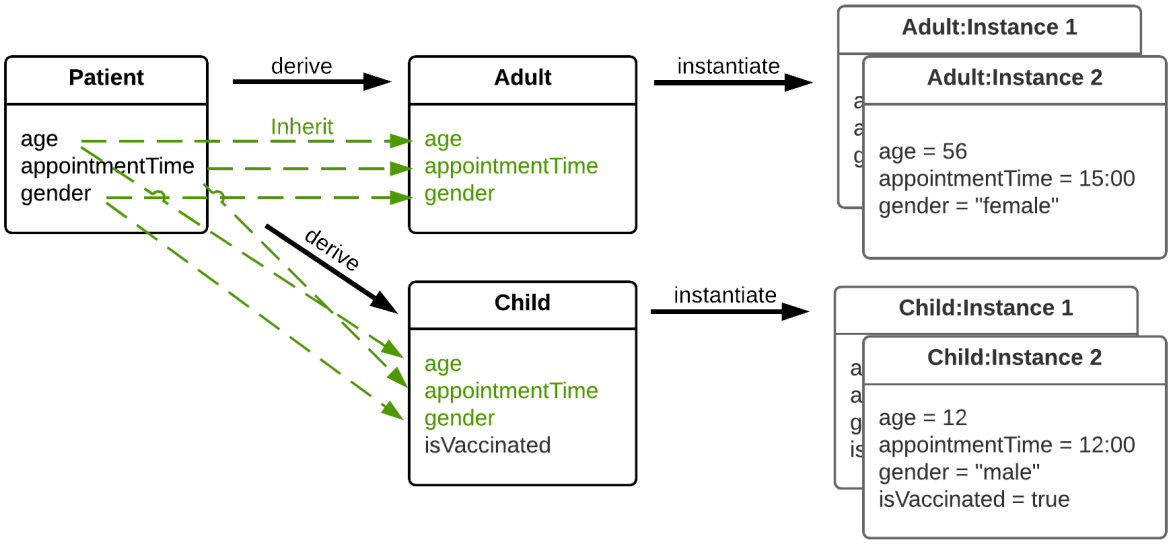
The class *Patient* does not represent individual patients yet, but it rather describes the properties of all patients. To get individual patients, we *instantiate* from the class *Patient*. Individual patients are now called *instances* of the class *Patient*. Each instance will have the same attributes (*age*, *appointmentTime,* and *gender*), but the values of those attributes can differ from instance to instance.



Derivation and Inheritance

The model of our system gets more complex. We also want to make a distinction between adults and children. They both have all the properties of a patient, but for children we also need to know whether they are vaccinated or not. We could implement this by adding the attributes *ageCategory* and *isVaccinated* to the class *Patient*, affecting both adults and children. However, object orientation gives us a more suitable tool for this goal: derivation. We *derive* two new classes from the original class Patient and call them *Adult* and *Child*. For the latter class, we add the attribute *isVaccinated*. When we talk about the relation between the classes Patient, Adult and Child, the class Patient is the *parent* or *origin* class*,* whereas the classes Adult and Child are *children* or *subclasses* of the class Patient.

The classes Adult and Child *inherit* the three existing attributes from Patient, which means that if we update (e.g., rename) those attributes in the class Patient, then that update will be reflected in both Adult and Child as well. However, inheritance only works from the origin to the subclass. If you change an inherited attribute in Adult or Child, then inheritance will be turned off for that attribute and the changes will not be reflected in the class Patient. Finally, we can instantiate and derive from Adult and Child like any other class.



Note

1. Both classes and instances are referred to as objects in Plant Simulation. However, it is always important to keep the distinction between object classes and object instances in mind.
2. Instead of deriving, it is also possible to duplicate a class. When duplicating, you essentially derive from a class without inheritance, such that you simply get an identical, separate copy of the duplicated class.
3. Plant Simulation consists of various basic classes to form the building blocks of your model. You can use duplication and derivation to create your own, special-purpose classes. However, sometimes you need to create a new class from a collection of existing classes. The built-in object *Frame* enables you to do that (multiple basic classes can be placed on one *Frame* to form a new class); this topic will be discussed in Part B of this tutorial. The only *Frame* that will be considered in Part A of this tutorial is the one containing your complete simulation model; we denote this *Frame* by *RootFrame*.

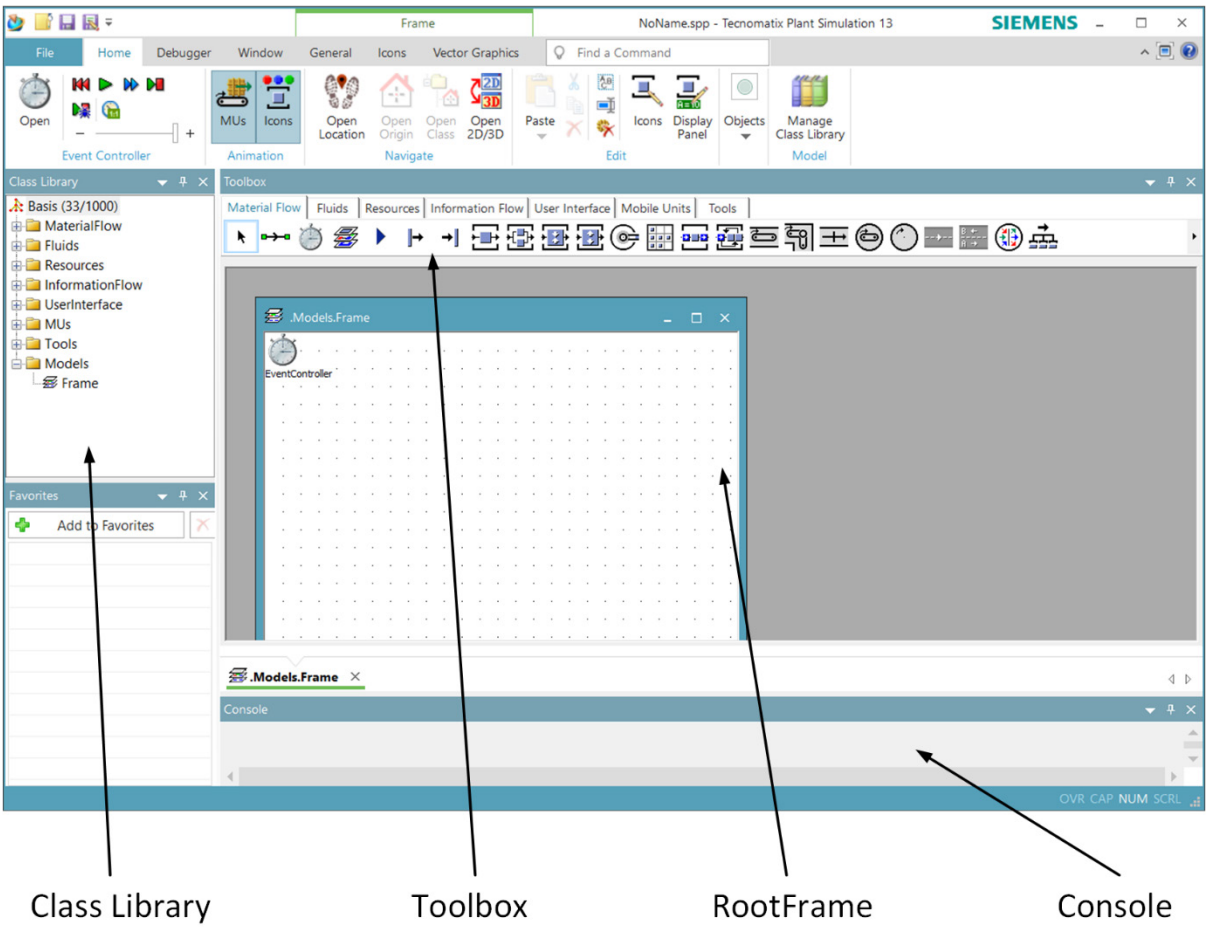
2.2 The Desktop

Task: Opening a new model

1. Start Plant Simulation.
2. Choose *Create New Model.*
3. The program asks you if you want to create a model with 2D, 3D or both. Just choose *2D only* to proceed.

You will now see the desktop in the figure below. It consists of a number of toolbars and docking windows:

- *Class Library*. Structured view of all object classes available in the current model. The object classes are stored in the familiar Windows tree format. You can add folders yourself and move, copy, create and delete classes. It is wise not to delete the basic classes of Plant Simulation, because you will need many of them to construct your models!
- *Console*. This is a window that shows information about the actions Plant Simulation executes.
- *Toolbox*. A structured view of object classes in the model. It is convenient to use default toolboxes (and to construct customised toolboxes) for object classes that you use frequently. Using toolboxes, you can insert object classes more easily in your model.
- *RootFrame*. Holds all the objects that make up your model.

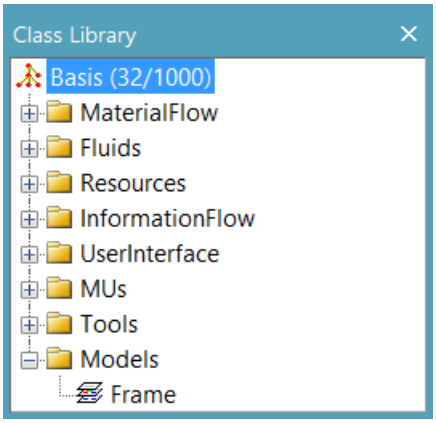


Note

1. If a toolbar or window is not activated yet, then you can activate it in the *Ribbon* by selecting *Window* → *Dockable Windows.*
2. Opening, closing and saving your model can be done using the *File* menu. You can open only one model at a time, so you have to close a model before you can open a new one.
3. You can enable or disable animation in the *Home* menu. Disabling animation will speed up your simulations, for instance when you need to make a large number of replications.
4. In the *File* → *Preferences* menu you have access to several settings. For instance, you can change the time format under *General*, you can select what elements to show and hide under *Modeling*, and you can modify the license settings under *License*. Instructions for setting up the license are covered in a separate document.

2.3 Working with the Class Library and Toolbox

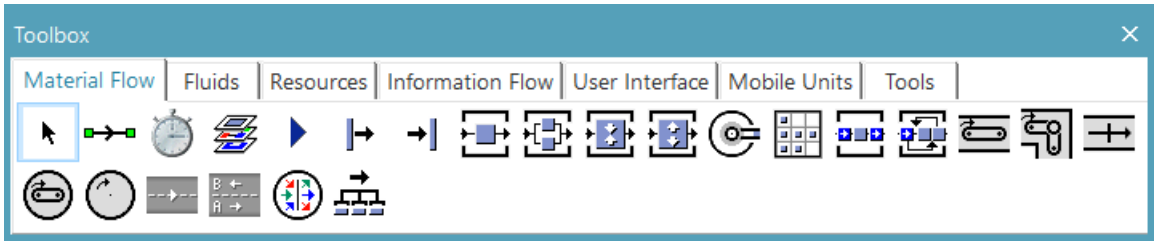
Plant Simulation provides a set of basic objects. These have features that - in many cases - allow you to directly use them in a simulation model. However, installations in the real world display such a wide variety of constellations that it is impossible to predict each and every situation and provide the appropriate objects. This is why Plant Simulation offers basic objects that you may modify in any perceivable way to meet your specific needs. These objects are called *application objects*.



The basic objects Plant Simulation provides can be classified using simple criteria. Knowing this system allows you later on to find a Plant Simulation object that represents a real world part. If you do not find an object that fits, you may model one by modifying or combining existing basic objects. You can find the objects in the *Class Library*. Plant Simulation shows the built-in objects in the *Class Library* in a hierarchical view in folders and subfolders. You can add, rename, and delete folders when needed. By default the *Class Library* contains eight folders:

- *MaterialFlow* objects serve for (i) transporting or processing mobile/moving unit (*MUs*) objects within models (active) and (ii) storing parts and displaying tracks on which parts are moved (passive).
- *Fluids* objects, such as *Pipes*, *Tanks* or *Mixers*, facilitate the modelling of so-called free-flowing materials. These materials can be in liquid, gaseous or pourable form.
- *Resource* objects serve for adding human workers to a processing station and let workers move on paths between workstations related to production stations.
- *InformationFlow* objects serve for exchange of information between objects (for example, a *Variable*, a *TableFile* or a *Method*).
- *UserInterface* objects facilitate the interaction between the user and a model, for example, *Dialogs* for model input and *Charts* and *Reports* for model output.
- *MUs* (moving units) represent the flow of materials. The distinction with other object classes is that *MUs* move through the model; this holds for the *MU* types *Entity* (products), *Container*, and *Transporter*.
- *Tools*, a folder to store special add-ins for performing specific simulation tasks.
- *Models* is a folder to store the models that you make as a user. You can also create additional folders and subfolders to structure your *Class Library*.

As a shortcut to access object classes, you may use the *Toolbox*, which is a container for the different Plant Simulation toolbars that hold the objects of the *Class Library*. Each tab contains a *Toolbox* with objects. It is a matter of personal preference whether you want to use the *Class Library* or the *Toolbox* for model construction. Generally, the *Toolbox* will be faster.

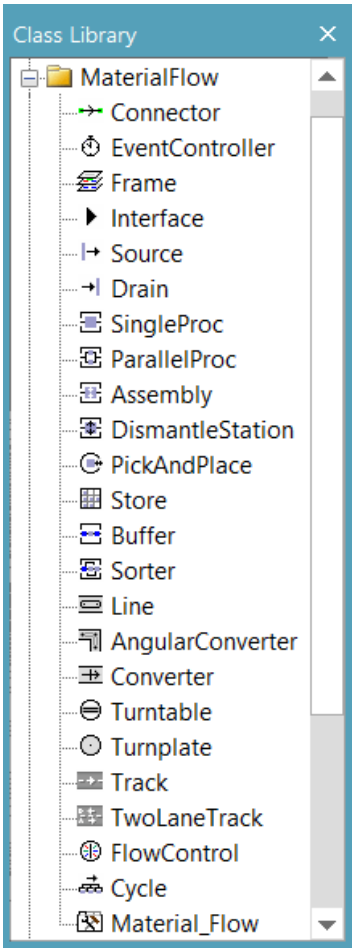


2.4 Overview of Basic Objects

As shown in the previous section, Plant Simulation provides a set of basic objects, grouped in different folders in the *Class Library*. We now present the most commonly used basic objects from the standard library (additional libraries can be added by going to *Home* → *Manage Class Library*) grouped according to these folders used by Plant Simulation: Material Flow, Resources, Information Flow, User Interface, Mobile Units, and Tools.

2.4.1 Material Flow

The *MaterialFlow* folder contains the basic object classes as shown before. We will discuss the most important objects from this folder briefly. Their use will be clarified when using them later on in this tutorial.



Connector

The *Connector* establishes connections between *MaterialFlow* objects, such that *MUs* (see Section 2.4.5) can move through the model. An arrow in the middle of the connector indicates the direction. A single connection can only point in one direction.

EventController

Plant Simulation is a discrete event simulator, i.e., the program only inspects those points in time, where events take place within the simulation model. The *EventController* manages and synchronises these events.

Frame

The *Frame* serves for grouping objects and to build hierarchically structured models. Each new model starts with a *Frame* where the EventController is placed on; this *Frame* is denoted by *RootFrame*.

 **Interface**

The *Interface* represents entry and exit interfaces on a *Frame*. It is used to connect multiple *Frames* with each other, such that *MUs* can flow through them.

 **Source**

The source creates *MUs* and attempts to pass them on. It is used at places where a *MU* is created/generated (usually at the start of a process). The time between the consecutive creations of *MUs* can be specified by a random variable.

 **Drain**

The object destroys *MUs* after processing them. It is used at places where *MUs* should leave the system (usually at the end of a process).

 **SingleProc**

The object receives a *MU*, retains it during the processing time and then attempts to pass it on. For example, a machine with capacity 1.

 **ParallelProc**

The object receives a *MU*, retains it during the processing time, and then attempts to pass it on. Several *MUs* may be processed at the same time. Processing times may differ and *MUs* may pass each other. For example, a machine with capacity >1.

 **Store**

The object receives passive *MUs*. A *MU* remains in the *Store* until it is removed by a user control. It can be used, for example, for a store shelving system.

 **Buffer**

The object receives a *MU*, retains it during a given dwell time, and then attempts to pass it on. When the preceding stations are unavailable (e.g., occupied or in failure), the MU stays in the *Buffer*. *MUs* can exit the *Buffer* in the same order in which they entered it (FIFO) or in the opposite direction (LIFO). These options are denoted by buffer type *Queue* and *Stack* respectively.

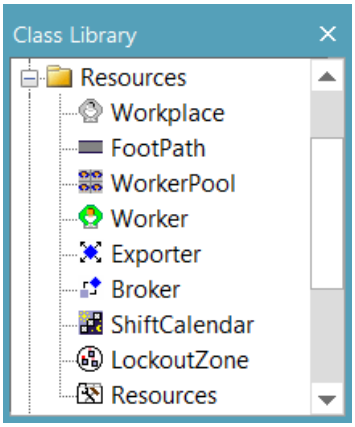
 **PlaceBuffer**

The object is similar to the *Buffer*, but with more advanced functionality (it consists of a sequence of stations that need to be visited sequentially by every *MU*). The *PlaceBuffer* is not part of the built-in objects from the *Toolbox*, but you can add it by clicking *Manage Class Library* on the *Home* ribbon tab.

 **Sorter**

Similar to a *PlaceBuffer*, but with sorting functionality. Sorting can be done based on, e.g., an object attribute value or the output of a sorting *Method*.

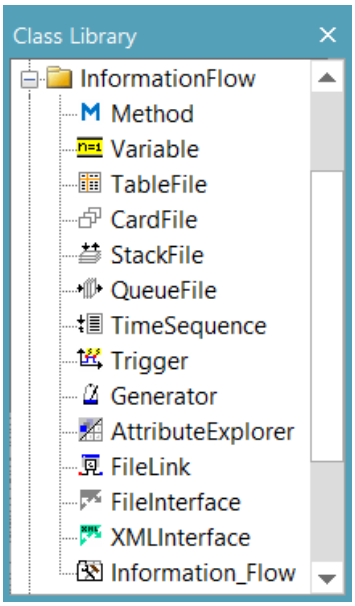
2.4.2 Resources



The resources folder contains some more specialised objects that facilitate the modelling of operators on the shop floor. We only describe the *ShiftCalendar* here.

ShiftCalendar
Allows you to configure the operating hours of objects.

2.4.3 Information Flow



The *InformationFlow* folder contains the basic object classes as shown here. We will discuss the most important objects from this class briefly. Their use will be clarified when using them in the examples.

Lists are provided to record large amounts of data, to store them and to make them available during simulation. They provide the functionality of a database in a real world installation. Plant Simulation provides *StackFile*, *QueueFile*, *CardFile,* and *TableFile*. These lists differ in their dimensions and the *Method*s provided for accessing them. In this tutorial, we will only use the *TableFile*.

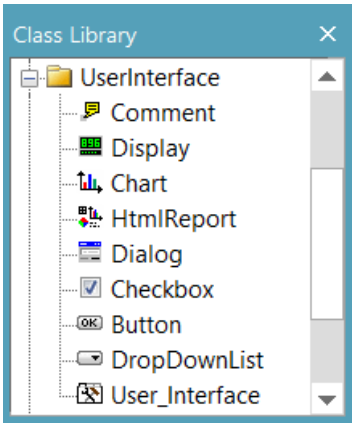
Method
The *Method* enables the modeller to program custom logic into the model, using the programming language *SimTalk 2.0* (See note in section 3.9).

Variable
The class *Variable* is a global variable that may be accessed by all objects.

TableFile
The *TableFile* is one of the most important information flow objects in Plant Simulation. It serves as a two-dimensional data container. Its elements may be randomly accessed. In addition, a number of search and read functions is available.

Generator
The *Generator* allows you to call *Methods* at predefined times during the simulation.

2.4.4 User Interface



UserInterface objects facilitate the interaction between the user and a model, for example with *Dialogs* for model input and *Charts* and *Reports* for model output.

Comment

The *Comment* enables you to add additional descriptions and notes. To the model.

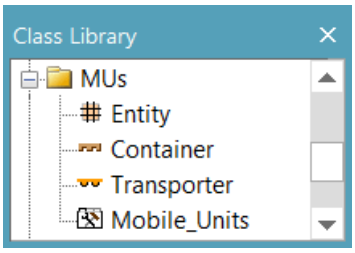
Display

The *Display* displays values during a simulation run. Values can be displayed in string form or as bars.

Chart

A *Chart* can be used to visualise the data generated by a model.

2.4.5 Mobile Units



These classes can represent every kind of product, pallet, container or vehicle that moves through a (logistics) system.

Entity (MU)

This is the object or *Moving Unit* that gets moved around in a simulation model. It can represent anything that must pass different stations for processing, e.g., patients, products, parts, and orders.

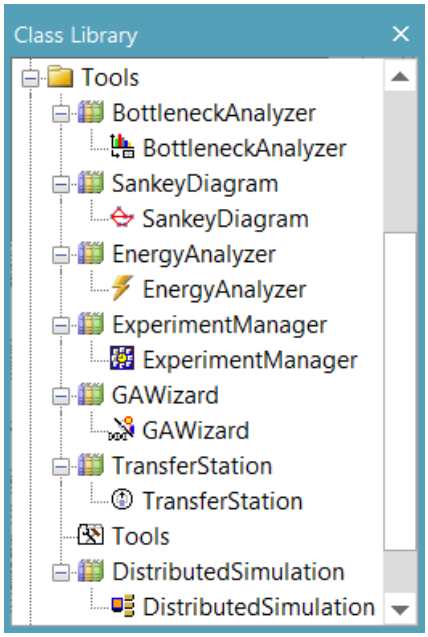
Container

Similar to the *Entity*, this is a mobile object during the simulation. It has a loading space that may contain *MUs*. It represents any kind of container, e.g., palettes and boxes.

Transporter

Similar to the Container, but the Transporter is self-propelled and its speed is user-defined. It represents any kind of transporter, e.g., AGVs and forklifts.

2.4.6 Tools



This folder contains add-ins for performing specific simulation tasks. In this tutorial, we only use the tools *ExperimentManager* and *GAWizard*.

ExperimentManager

Use this object to configure a list of experiments and the number of replications per experiment. The *ExperimentManager* then carries out all the predefined experiments.

GAWizard

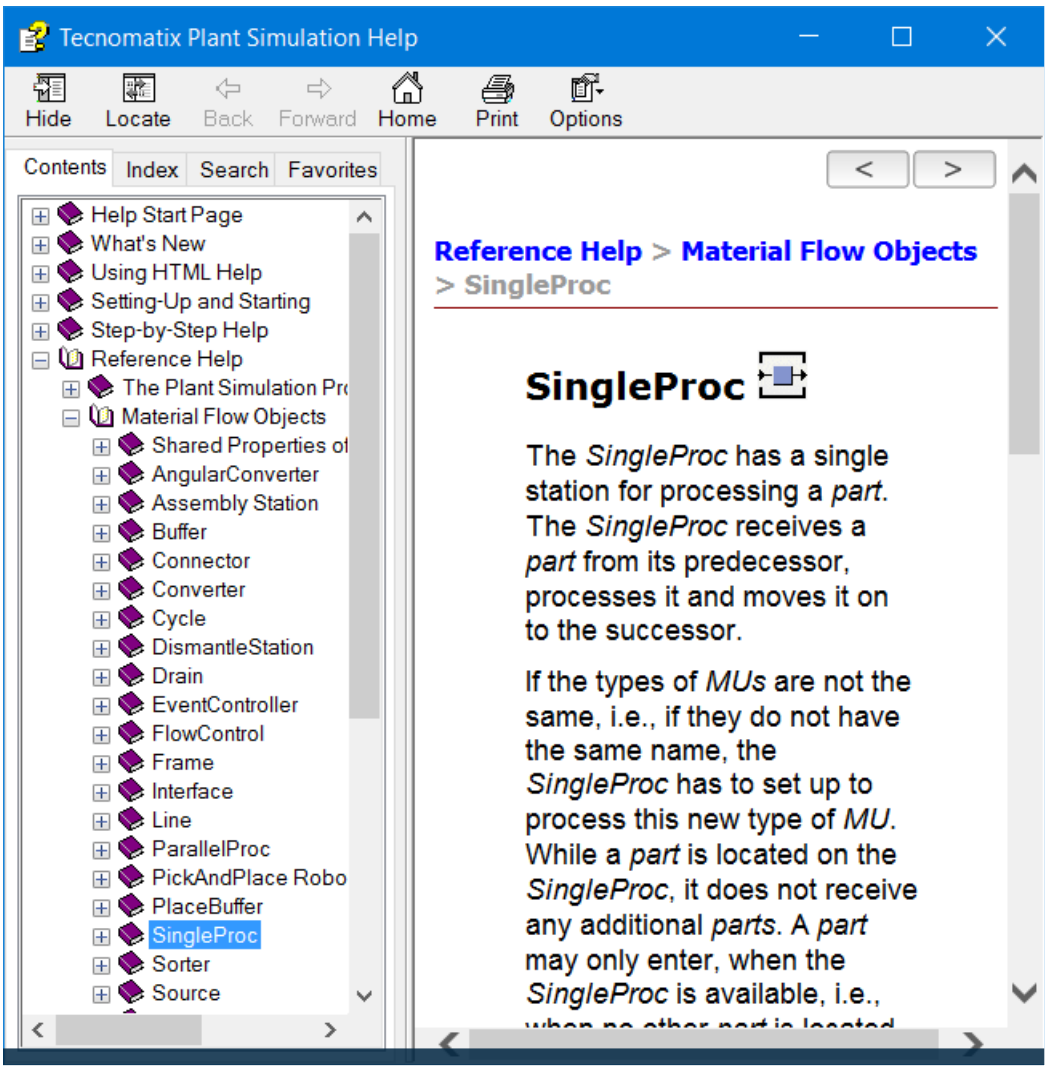
This tool can be used in case the number of predefined experiments grows very large and if it takes too much time to carry out all of them. The *GAWizard* makes use of a so-called *Genetic Algorithm* to select the next experiment based on the results of previous experiments. This way, the number of experiments carried out can be reduced considerably, while still providing good solutions.

In the following task you learn how to get help on a specific object.

Task: Search in the help function for the features of the *SingleProc*

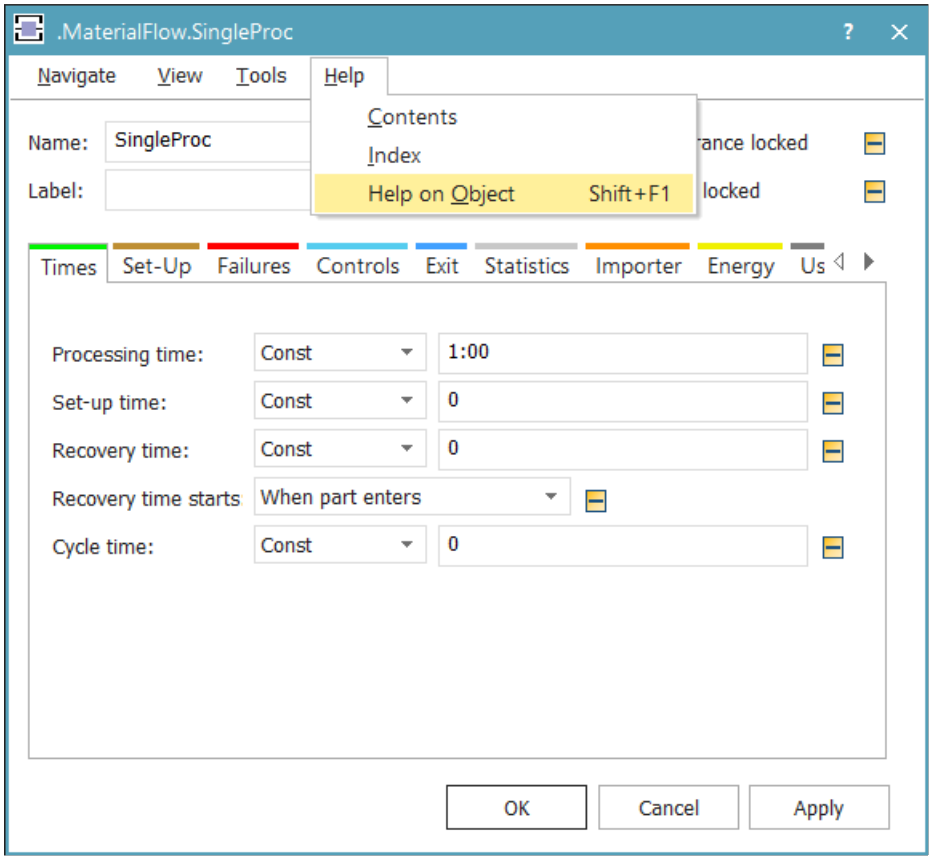
Suppose that you wish to know more about the features of the *SingleProc*. Then proceed as follows:

1. Select *Help* → *Contents* from the *File* menu. You arrive at the object help.
2. Select *Reference Help* → *Material Flow Objects* → *SingleProc*. You should see the following screen:



3. This page gives you an overview of the *SingleProc*, as well as links to related topics.

4. Close the *Help* window.
5. A shortcut is to use the *Help on Object* function. Inside the *Class Library*, open the *MaterialFlow* folder and double-click *SingleProc.* Now select *Help*, and then *Help on Object* from the local menu. Another shortcut is to select an object, or even a specific attribute of an object, and press *F1.*



2.5 Objects used in this Tutorial

The aim of this tutorial is to get you familiar with the Plant Simulation software as well as with simulation modelling in general. The aim is not to discuss all the functionalities of the Plant Simulation software, but only introduce the basic functionalities that are often present also in other graphical simulation software packages. An overview of the standard Plant Simulation objects used in this tutorial can be found below.


Pointer


Connector


EventController


Frame


Interface


Source


Drain


SingleProc


ParallelProc

 AssemblyStation

 PickAndPlace

 Store

 Buffer

 PlaceBuffer

 Sorter

 Line

 Track

 FlowControl

 Workplace

 FootPath

 WorkerPool

 Worker

 Broker

 ShiftCalendar

 Method

Variable

 TableFile

Generator

 Comment

Button

 Entity

 Container

 Transporter

 ExperimentManager

GAWizard

2.6 Functionality, animation, and visualisation

Plant Simulation comprises all the features needed to model the functional aspects of most real-world systems. However, it also contains features for animation purposes and visualisation of results, e.g., in graphs. Animation and visualisation are used extensively in commercial applications to communicate with, convince, or simply impress the client. It is also a useful tool for debugging a model, because a modeller can see with his own eyes whether the model behaves as expected.

In an academic environment, it is usually sufficient to focus on functionality, since the model itself is often not the primary output. Animation is then only used for debugging, and turned off when possible to increase model performance. Visualisation is deferred to a further analysis of the data generated by a model, e.g., in Excel. In this tutorial, we will focus on the basic objects that allow you to model the functional aspects of a system. This is the quickest way to get acquainted with the tools needed to model a large variety of systems, and it makes the transition to other simulation software packages easier. In other words, the aim of this tutorial is to provide insight into simulation model implementation in general rather than presenting an exhaustive list of Plant Simulation features.

Part A: Basic Simulation Modelling

MODELLING A GENERAL PRACTITIONER'S OFFICE

3 Building a Model: General Practitioner

Simulation models allow you to capture the properties of a real-world system and experiment with different configurations that could improve the real-world system. Usually, the model starts out simple, and complexity is only added when needed to better describe the essential properties of the real-world system. The system that we will model in this chapter is that of a General Practitioner, or a GP for short. This GP starts out simple: he has one office and the consult with each patient takes a known amount of time. We will then expand the system step-by-step to include a waiting room and two GPs. Then, one GP decides to specialise in consulting children, while the other GP treats adults. Finally, you will implement basic prioritisation in the waiting room to reduce perceived waiting times.

In this chapter, you will model the general practitioner using Plant Simulation. You will learn to use the standard classes available in Plant Simulation and to derive subclasses from these classes. Furthermore you learn to use the *Icon Editor* and learn the principles of the push-block theorem.

Subjects dealt with in this chapter:

- Simple model building
- *MaterialFlow* objects
- *Moving Units* (*MUs*)
- Customised attributes
- Icon editor
- Entrance and exit controls
- Simple *Methods*
- Prioritisation
- Simple performance measurement

3.1 Creating a New Model

The General Practitioner will be modelled in a single *Frame*, called the *RootFrame* (see Chapter 2). It is important that you save your work on a regular basis, because it might occur that Plant Simulation crashes (due to coding errors like an infinite loop).

Task: Create a new model

1. Start Plant Simulation.
2. Choose *Create New Model.*
3. Click *2D only* in the dialog window that pops up.

3.2 A Basic Model

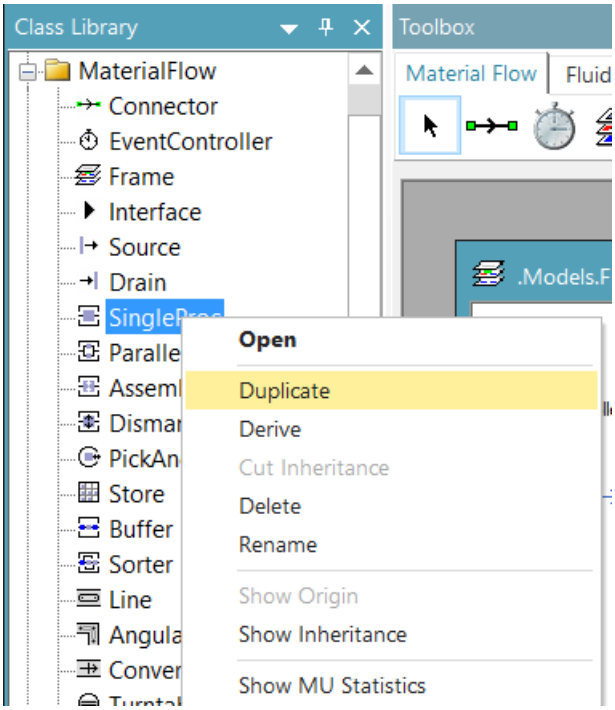
Before you are able to build complex simulation models, it is necessary to learn the modelling basics. We start with a basic model of a general practitioner's office, who gives consultation to patients. To build this model we will need three basic objects, namely the *Source*, *SingleProc*, and *Drain.* These objects need to be connected using the *Connector*.

- The *Source* object generates the patients (*MU*s) in the General Practitioner model.
- The *SingleProc* object processes the patients for a certain processing time. This object will represent the general practitioner.
- The *Drain* object allows the patients to leave the model after they have visited the general practitioner.
- The *Connector* connects the basic objects, which makes it possible for the patients to flow through the model.

Since we intend to use the object general practitioner more frequently, it is useful to duplicate and instantiate the object first from the *SingleProc*. By creating a separate class in the *Class Library*, we will save time when expanding our model later on. Due to inheritance, we only have to create and configure the object for the general practitioner once and not for every instance we use of the general practitioner. Another advantage is that if we would like to adjust our general practitioner (for example the processing time) we only need to do this once. The *Source* and *Drain*, however, are instantiated directly, because we will only use them once in our model and for that reason there is no added value in duplicating them first.

Task: Create the GeneralPractitioner

1. Right-click on the object *SingleProc* in the folder *MaterialFlow*.
2. Select duplicate. You now have an object named *SingleProc1* in the folder *MaterialFlow*.

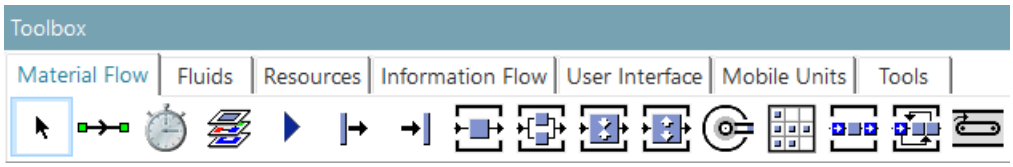


3. Rename *SingleProc1* to *GeneralPractitioner*. You can do this by right-clicking the object and selecting *Rename*, or by selecting the object and pressing *F2*.

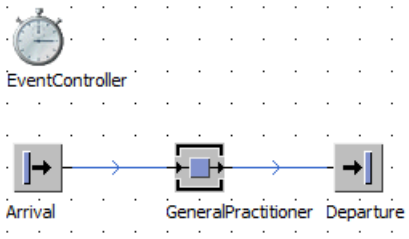
All the objects we would like to use are now available in the folder *MaterialFlow*. The next step is to build your first basic model.

Task: Create the basic model

1. Drag the *Source, Drain,* and *GeneralPractitioner* object, from the *MaterialFlow Class Library* or toolbox, to your *RootFrame*.



2. Connect the objects with each other. To connect the objects manually, you need to select the *Connector* from the *Toolbox*.
3. Connect the objects by clicking on the object where you want to start the connection and then on the object that you want on the other end of the connection.
4. Rename the *Source* and *Drain* to *Arrival* and *Departure* respectively.



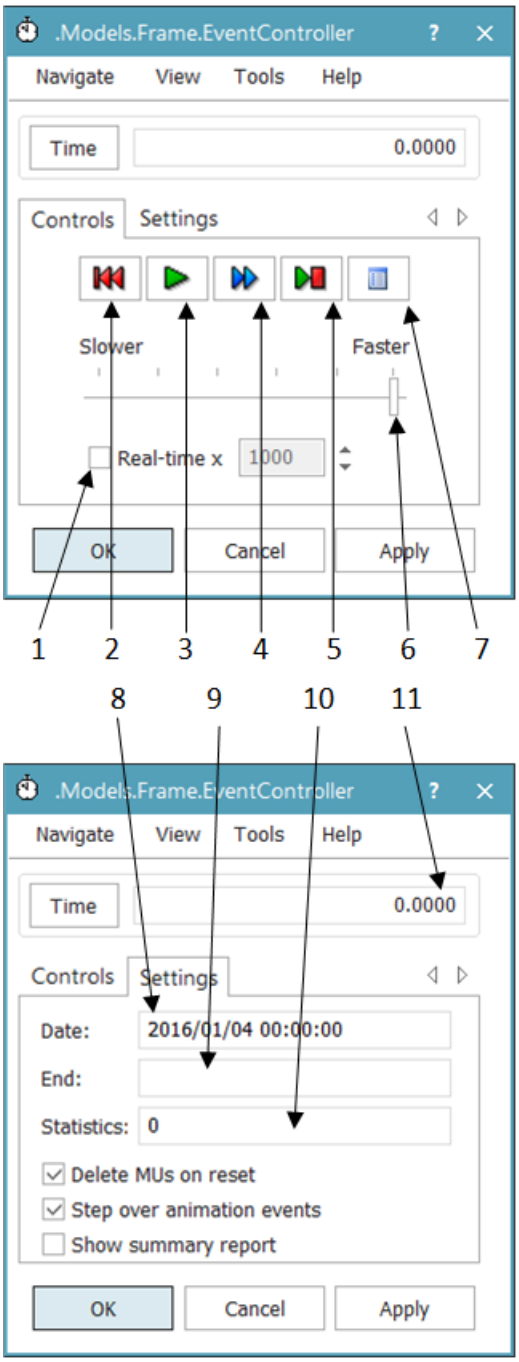
Did you know?

If you have to connect multiple objects, activate connection mode by holding the Ctrl-Key while making connections. This way, you do not have to select the *Connector* from the toolbox every single time. To terminate connection mode, click the right mouse button in the model, select the *Pointer* or hit the *Esc-key*. An alternative way of connecting objects is to place them right next to each other and then move them apart; a *Connector* will then be placed in between automatically.

3.3 The EventController

In order to see whether your model works properly, you should test it using the *EventController.* One of its tasks is to coordinate events that take place during the simulation. In order to do that, the EventController governs the event list, which is a list of all the events that are scheduled to happen in the future. For now, it is important to remember that the *EventController* window is where you start, stop, reset, and alter the speed of your simulation. By definition, the *EventController* is placed on the *RootFrame* containing the entire model, not on a *Frame* containing a sub-model or on an object.

1. **Real time.** Usually, the simulation jumps from event to event in a fluctuating pace (e.g., when there is no event, the simulation clock simply jumps forwards in time). Select the *Real time* option to let the simulation time run in a constant pace instead, corresponding with the real time (e.g., "Real time x 10" means the simulation model tries to run 10 times faster than reality).
2. **Reset**. Remove all *MUs* and reset the simulation clock to zero.
3. **Start/Stop**. Let the simulation run; jumping from event to event. Click it again to pause.
4. **Fast forward.** Disable animation to increase performance, and run the simulation at a high speed.
5. **Step.** Execute the next event in the event list. Useful for debugging.
6. **Speed**. Move the slider to choose a simulation speed.
7. **Event debugger**. Opens the event list. The event list will be elaborated on in the "Did you know?" at the end of Section 5.3.
8. **Date**. The starting date of the simulation.
9. **End.** The ending time of the simulation. For example, set it to 70:00:00:00 if the simulation must run for 70 days (see Section 3.4 for an explanation of the time format).
10. **Statistics**. The time at which the model starts collecting statistics. Use this when your model has a warm-up period.
11. **Simulation clock**. By default, this clock shows the relative time, i.e., the elapsed simulation time since the start date. Click *Time* to switch to the absolute simulation date.



Task: Test your model

1. Double-click on the *EventController* and a dialog window will open.
2. Click on the *Reset* button to return the model into a predefined starting position.
3. Start the simulation by clicking on the *Start/Stop* button.
4. Stop the simulation by clicking on the *Start/Stop* button.
5. You should see that the *MU*s are flowing in your model.
6. Reset the simulation by clicking the *Reset* button.
7. The simulation clock should reset to zero and all *MUs* are removed from the model.

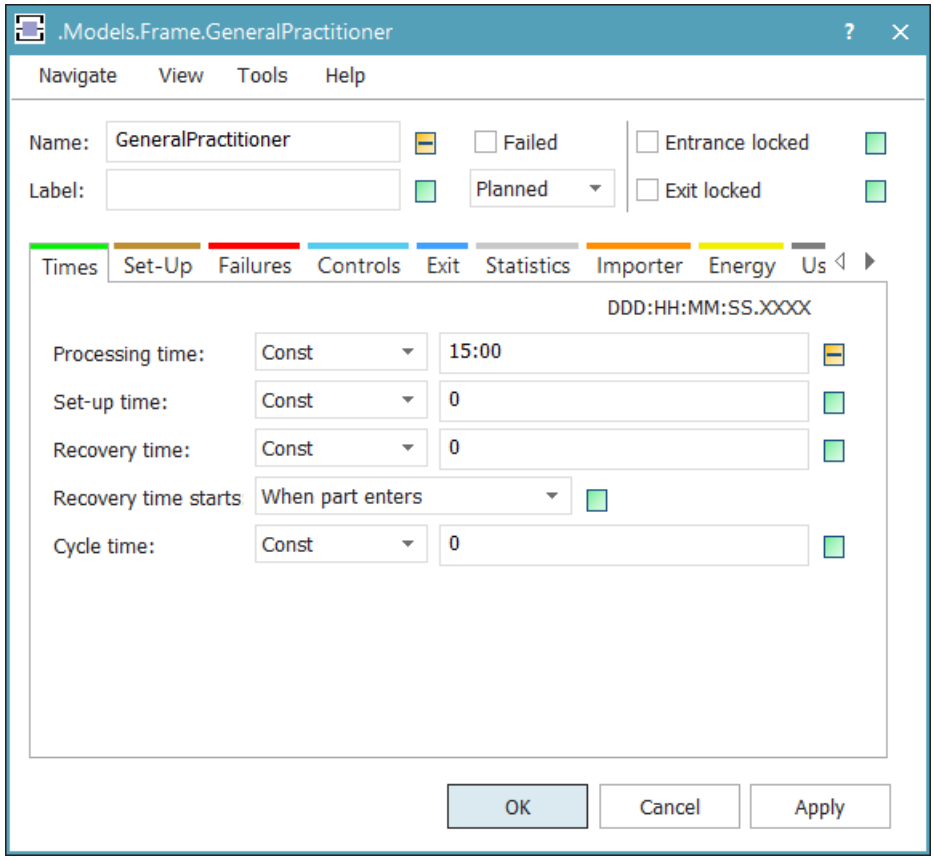
3.4 Interarrival Times and Processing Times

Now that you have built the fundamentals of the basic model, you need to adjust the settings of the objects in order to simulate a real-world system. Plant Simulation has the following time format: DD:HH:MM:SS.XXXX, which stands days, hours, minutes, seconds, and milliseconds, respectively. If you want to adjust, e.g., the processing time of a certain object, then you need to follow this time format. You can, however, also enter the duration of a certain process in seconds. Plant Simulation will translate the amount of seconds to the time format previously described.

You can change the settings of an object by double-clicking the object. If you double-click on the object in the *Class Library* you will adjust the settings for every object in your model that has been derived or instantiated from that object. This is useful if you need to adjust the processing time for multiple objects, e.g., for the *GeneralPractitioner*.

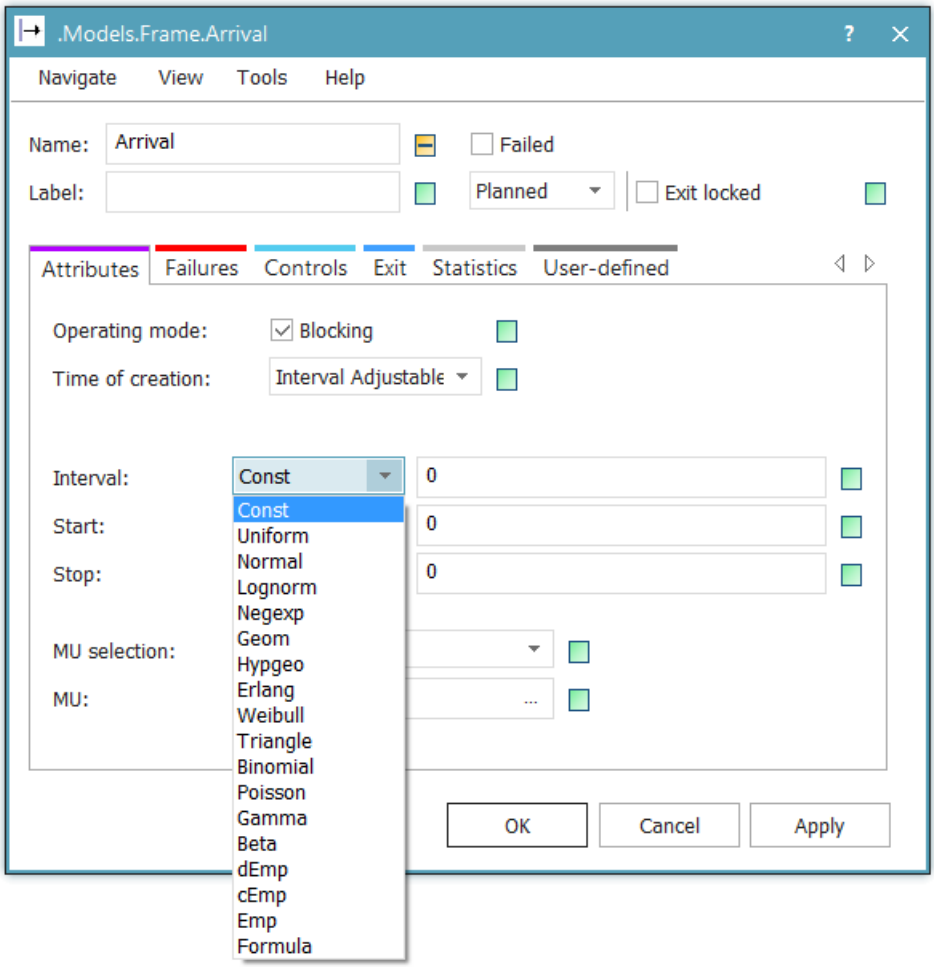
Task: Adjust the GeneralPractitioner object

1. Double-click on the *GeneralPractitioner* in the folder *MaterialFlow* from your *Class Library*.
2. Set the *Processing time* of the *GeneralPractitioner* to 15:00. Leave the *Processing time* distribution on *Const* (short for Constant).



3. Double-Click on the *GeneralPractitioner* in your model (on the *RootFrame*).
4. Note that the *Processing time* of this object is adjusted due to inheritance.

As illustrated in Section 3.2, the *Source* object generates the *MU*s. You can adjust the creation of these *MU*s by changing the settings of the object. If you double-click on the *Source* object in your model then the following dialog window appears:



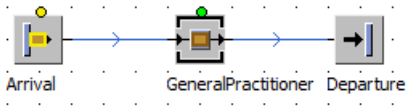
Interval describes the time duration between generating two *MU*s. You may also set a Start time or a Stop time for generating *MU*s. As you can see from the dialog window, the interarrival times are set to be constant (*Const*). You can change this by adjusting the interarrival distribution. Click on the drop-down button and select the corresponding distribution that fits your data. Note that when you want to model a Poisson arrival process, you need to select the *Negexp* (negative exponential) distribution.

Task: Adjust the *Arrival* object

1. Double-click on the *Arrival* object in your model.
2. Set the Interval time to a constant 720 seconds.
3. Close the dialog window.
4. Double-click on the *Arrival* object in your model.
5. Note that the time format is adjusted to 12:00.
6. Double-click on the *Source* object in the folder *MaterialFlow* from your *Class Library*.
7. Note that the interval time has not been changed and still is 0. The reason is that inheritance of the interval time was turned off the moment you changed it in the object instance.

3.5 Add a Waiting Room

You might have noticed that the processing times set in Section 3.4 will create an unbalanced model (utilisation $\rho = \lambda/(c\mu) \geq 1$). The combined processing rate of the General Practitioner is namely lower than the arrival rate of patients. If you run your model with the adjusted processing times, you might notice that the *MU*s at *Arrival* sometimes will turn yellow.



The yellow icon indicates that the *MU* is currently *blocked*, because it cannot move further in the model. If the next processing station is available again, it will display its regular icon again and continues in the model. Note that no new *MU*s are created when a blocked *MU* occupies the *Source*. This might be desired behaviour, e.g., for modelling a queueing system where *MU*s decide not to enter the system when the queue is too long. In most cases, however, this behaviour leads to an invalid model!

Did you know?

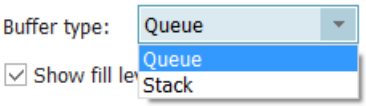
Plant Simulation provides the following pre-defined *states* for material flow basic objects.

- Static objects: *blocked, failed, recovering, operational, paused, setting-up, working, waiting*
- Moving objects: *operational, waiting, failed, paused*

During animation, the program displays the appropriate icon depending on the state of the basic object, provided an icon with the name exists. By default the basic objects have the icons listed above. The switching between icons during simulation, depending on the state, becomes visible in the following exercises.

Plant Simulation offers various objects to temporarily store *MU*s as briefly described in Chapter 2, namely:

- The *Store* object. This object is one of the passive objects in Plant Simulation, which means that it has no setup time or processing time and no exit control. The storage places are organised in a matrix, with an X-dimension and a Y-dimension. As long as the *Store* has available places, it can receive *MU*s. The *MU*s can remain in the store until they will be removed by using a control *Method*.
- The *PlaceBuffer* object. In this type of buffer, the *MU*s cannot pass each other. The first *MU* to be passed on to the next processing step will be the *MU* with the longest waiting time in the buffer (First In First Out, FIFO). When that *MU* is passed on, all the other *MU*s move forward one place. Note that the *Class Library* might not contain the *PlaceBuffer* by default. In that case, you can add this object to the *Class Library* by going to *Home* → *Manage Class Library* and check the checkbox before *PlaceBuffer*.
- The *Buffer* object. As opposed to the *PlaceBuffer*, the buffer does not have a place-oriented structure. You can determine a buffer type for unloading the *MU*s.

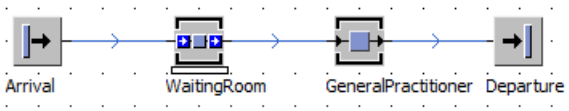


  If you select *Queue*, the *Buffer* will unload the *MU*s using the First In First Out (FIFO) principle. If you select *Stack*, the *Buffer* will unload the *MU*s using the Last In First Out (LIFO) order.
- The *Sorter* object. In this object, the *MU*s can be rearranged in a different order. The *Sorter* will prioritise the *MU*s present in the *Sorter*. It can use the following selection criteria to determine the prioritisation of the *MU*s, which will be demonstrated later on:
  - Duration of Stay (FIFO)
  - *MU* attribute
  - Method

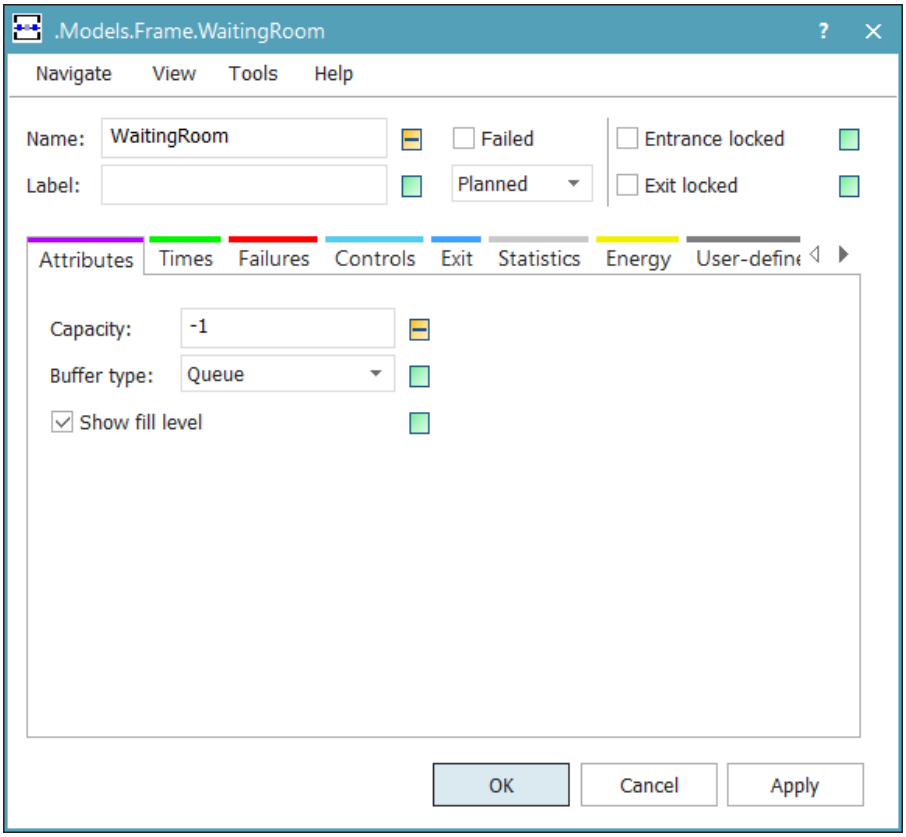
The capacity of these objects can be adjusted in their settings dialog window. It is possible to set the capacity to infinity: if you enter *-1* as capacity, your storage object has infinite capacity.

Task: Add a Waiting Room

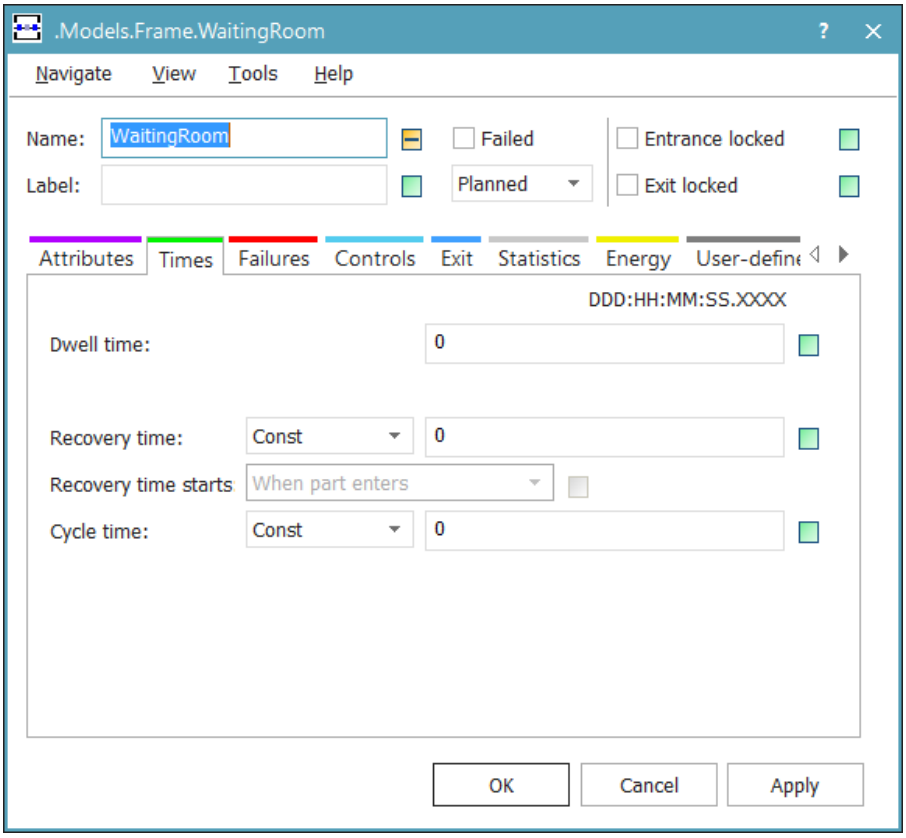
1. Delete the *Connector* between your *Arrival* object and your *GeneralPractitioner* object.
2. Insert a *Buffer* object between your *Arrival* object and your *GeneralPractitioner* object.
3. Rename the *Buffer* object to *WaitingRoom*.
4. Connect the objects.



5. Set the *WaitingRoom* capacity to unlimited (-1).



6. Click on the tab *Times*.
7. Make sure the *Dwell time* is set to 0.
8. Test your model.

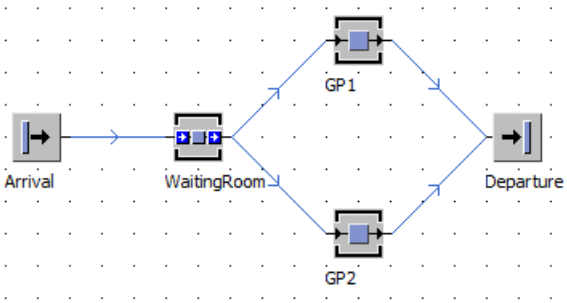


3.6 Multiple General Practitioners

For the next step you will add a second general practitioner to your model. Since we have already derived a *GeneralPractitioner* subclass from the *SingleProc* class, we can simply instantiate a new *GeneralPractitioner* object without having to configure everything (e.g., the processing time) again.

Task: Add an additional General Practitioner

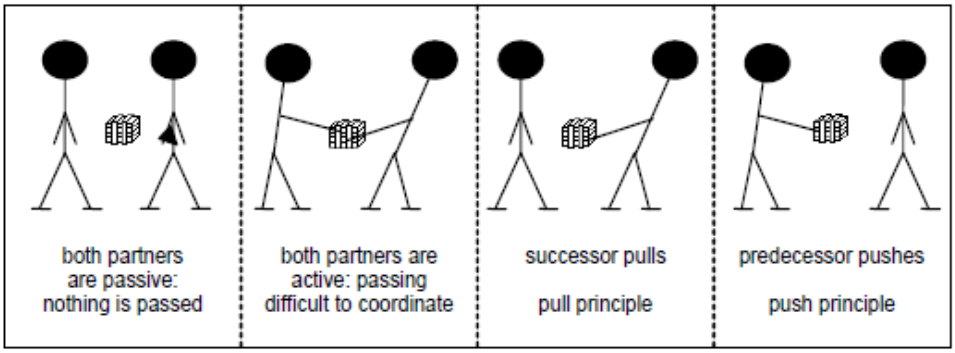
1. Drag a second *GeneralPractitioner* object to your model.
2. Connect the *WaitingRoom* object and the *Departure* object with the second *GeneralPractitioner* object.
3. Rename your *GeneralPractitioner* objects to GP1 and GP2, respectively.



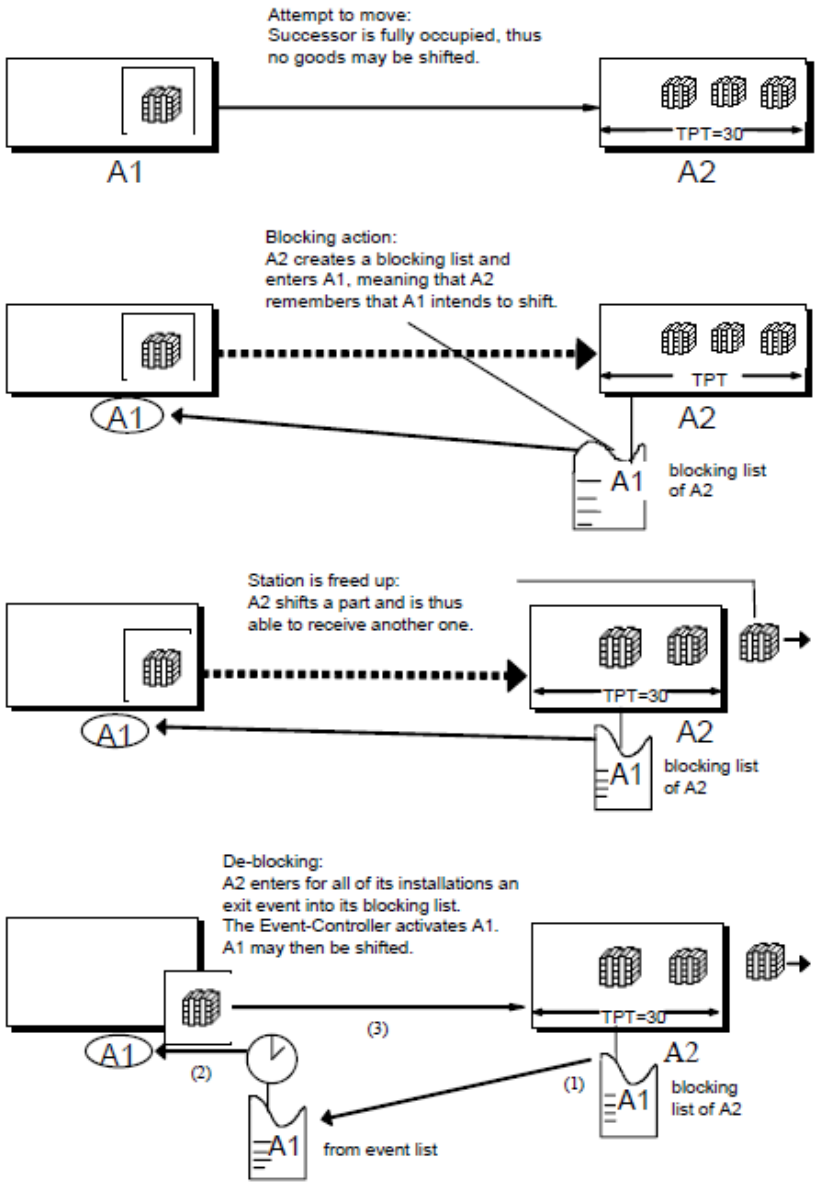
4. Test your model.
5. Note that the *MU*s are passed to the *GP* object that has been idle for the longest time.

Did you know?

You might wonder what happens internally when parts that are ready to move are blocked. The illustration below displays four different possibilities for passing parts. Plant Simulation has implemented the **push-block theorem** as basic functionality when passing parts between basic objects. An object with a part that is ready to be moved actively attempts to move that part on to its successor (push principle). If the successor cannot receive the part at the moment, the blocking principle is activated, guaranteeing that the object ready to move a *MU* is re-activated as soon the successor is ready to receive the part.



The push-block theorem ensures that parts are passed on using the basic functionality of objects and that the stream of events is not interrupted by blockages, failures or pauses. The following illustrations show how the **push-block theorem** works in Plant Simulation.



3.7 Adults and Children

Currently the *MU*s in your model are of the type *Entity* from the folder *MUs* in your *Class Library*. For our model we would like to make a distinction between the type of patients, namely between adults and children (see Section 2.1). For this purpose we will (again) make use of inheritance. First we will create the *MU Patient*.

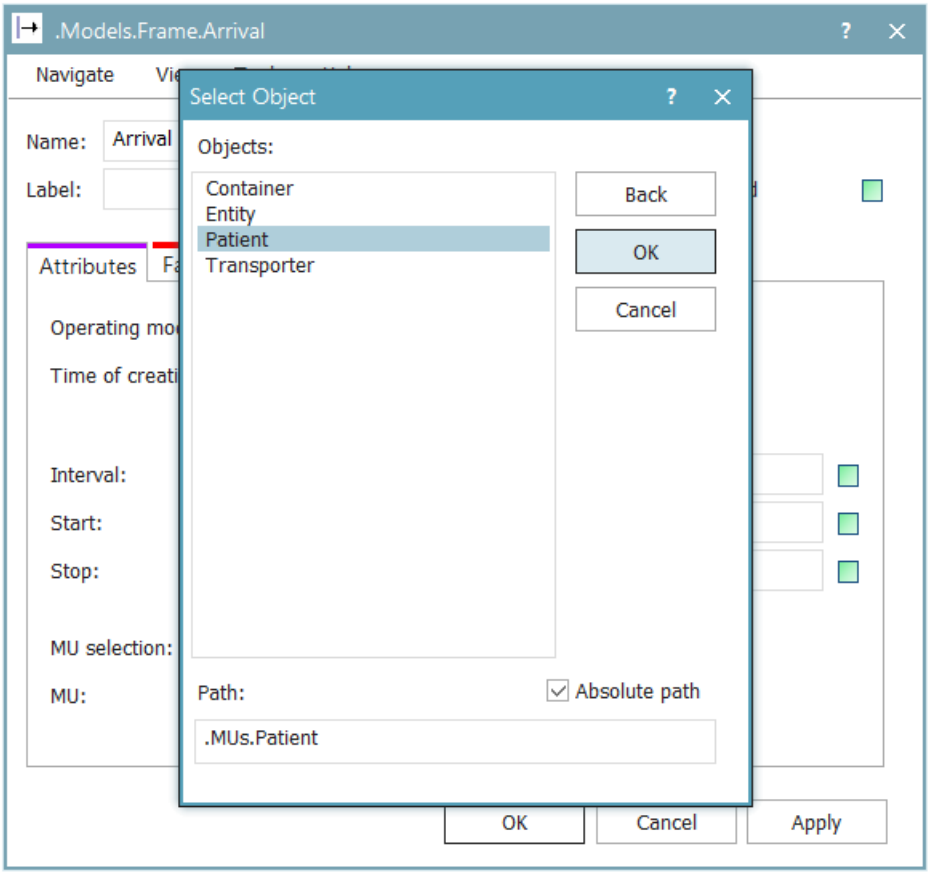
Task: Create a MU Patient

1. Right-click on the object *Entity* in the folder *MUs*.
2. Select *Duplicate.* You now have an object named *Entity1* in the folder *MUs*.
3. Rename the *MU* to *Patient.*

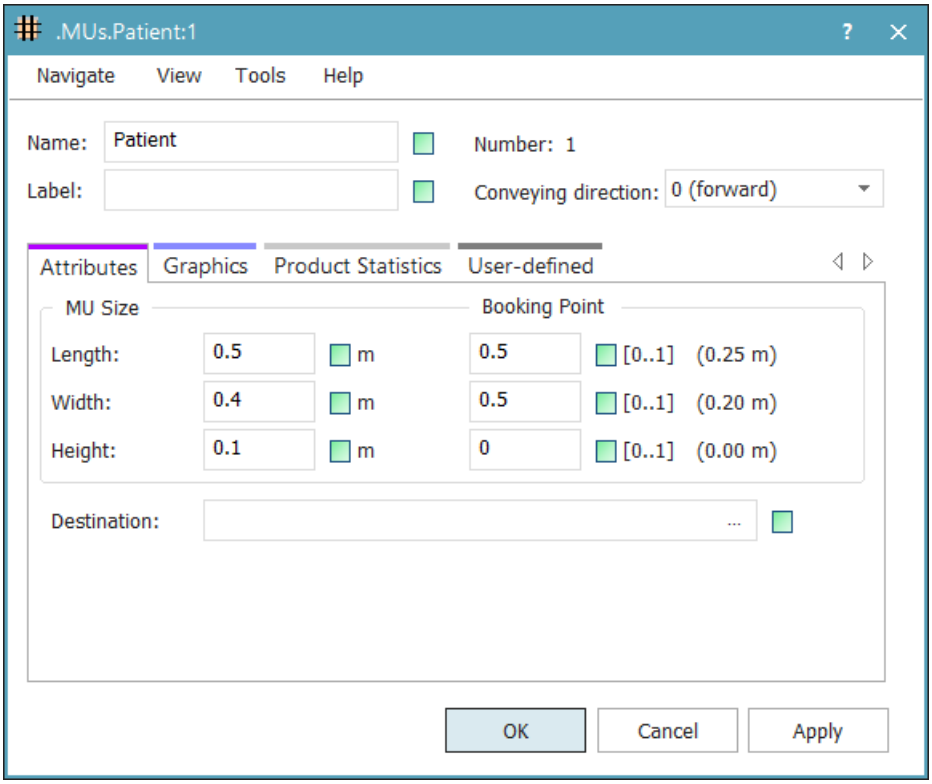
The next step is to set the newly created *MU Patient* as input for your model. Therefore you need to adjust the *Source* object in your model.

Task: Change the input for a Source object

1. Double-click on the *Arrival* object in your model.
2. Click on the ... button in the input field of *MU*, which currently states **.MUs.Entity*.
3. Select *Patient* and press OK.



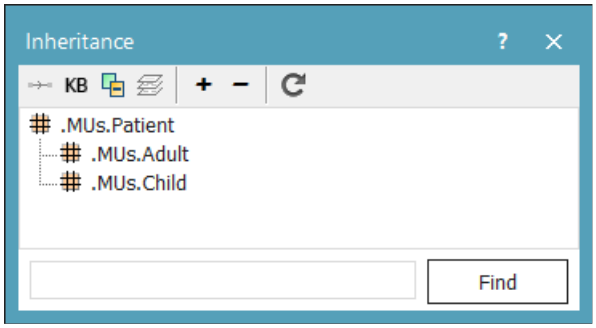
4. Close the dialog window.
5. Run your model and pause it when some *MU*s have been generated.
6. Click on a *MU* and note that the *MU*s are now *Patients*.



Because the *Patient* could either be a child or an adult, we would like to distinguish between the two. Therefore you need to derive two additional subclasses from the *MU Patient*.

Task: Create subclasses from the MU Patient

1. Derive from the *MU* Patient two new *MU*s.
2. Rename them *Adult* and *Child*, respectively.
3. Right-click on *Patient* and select *Show Inheritance*.
4. Note that *Adult* and *Child* are subclasses of *Patient*.



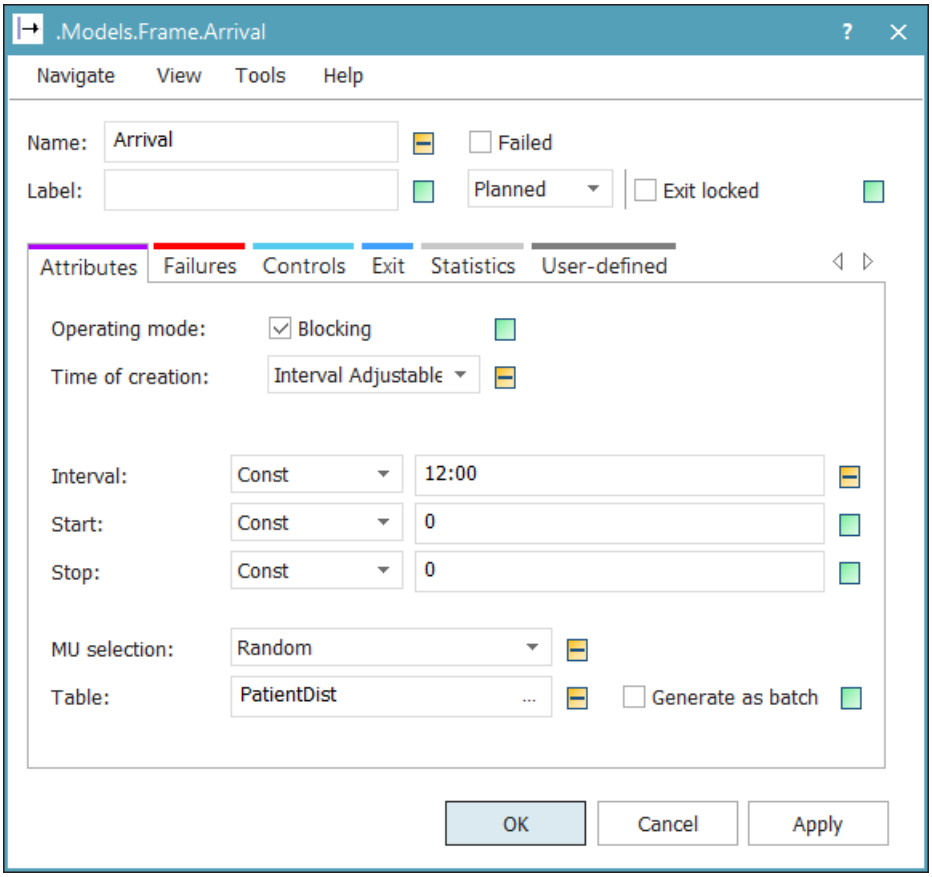
If you run your model you will still only receive the *Patient* as input. In order to get either a *Child* or an *Adult* as input *MU*s, we will make use of a *TableFile*, which can be found under *InformationFlow*. The *TableFile* is a two-dimensional list, which can store information for various purposes in simulation projects, e.g.:

- Storage of production plans
- Collection of information
- Parameters for objects

For our purposes, we use the *TableFile* to set the relative proportion of *Childs* and *Adults* that enter the model.

Task: Insert a TableFile

1. Insert a *TableFile* from *InformationFlow* in your model.
2. Rename it to *PatientDist*.
3. Double-click on the *PatientDist* to see the current layout.
4. Close the *TableFile*.
5. Double-click on the *Arrival* object.
6. Change the *MU selection* to random.
7. Click on the ... button in the input field of *Table* and select *PatientDist*.



8. Click *Apply* and close the dialog window.
9. Double-click on the *PatientDist* object and note that its layout has changed.

As the previous task has illustrated, the *TableFile* has been automatically adjusted to the type of information that it will hold. It is possible to do this manually, but we will not elaborate on this feature. Now that the *TableFile* is selected as the input for the *Arrival* object, you will need to specify what type of *MU* will be generated from the *TableFile*. It is also necessary to define the relative frequency in which they arrive.

Task: Fill the TableFile

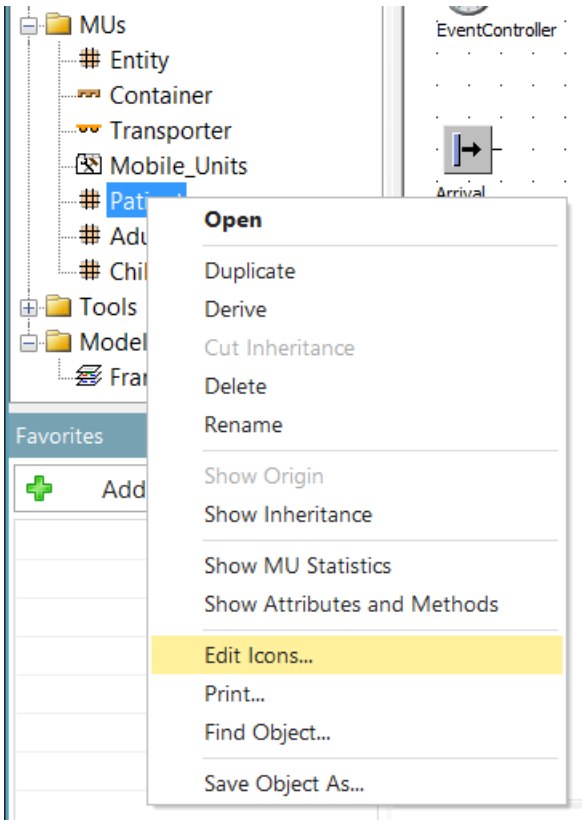
1. Open *PatientDist* by double-clicking.
2. Drag the *MU* Adult to the first cell under the column *MU* (or just type in the location of the *MU* in the *Class Library*).
3. Drag the *MU* Child to the second cell under the column *MU*.
4. Enter the frequencies 0.60 and 0.40 in the column Frequency.

	object 1	real 2	integer 3	string 4	table 5
string	MU	Frequency	Number	Name	Attributes
1	.MUs.Adult	0.60			
2	.MUs.Child	0.40			

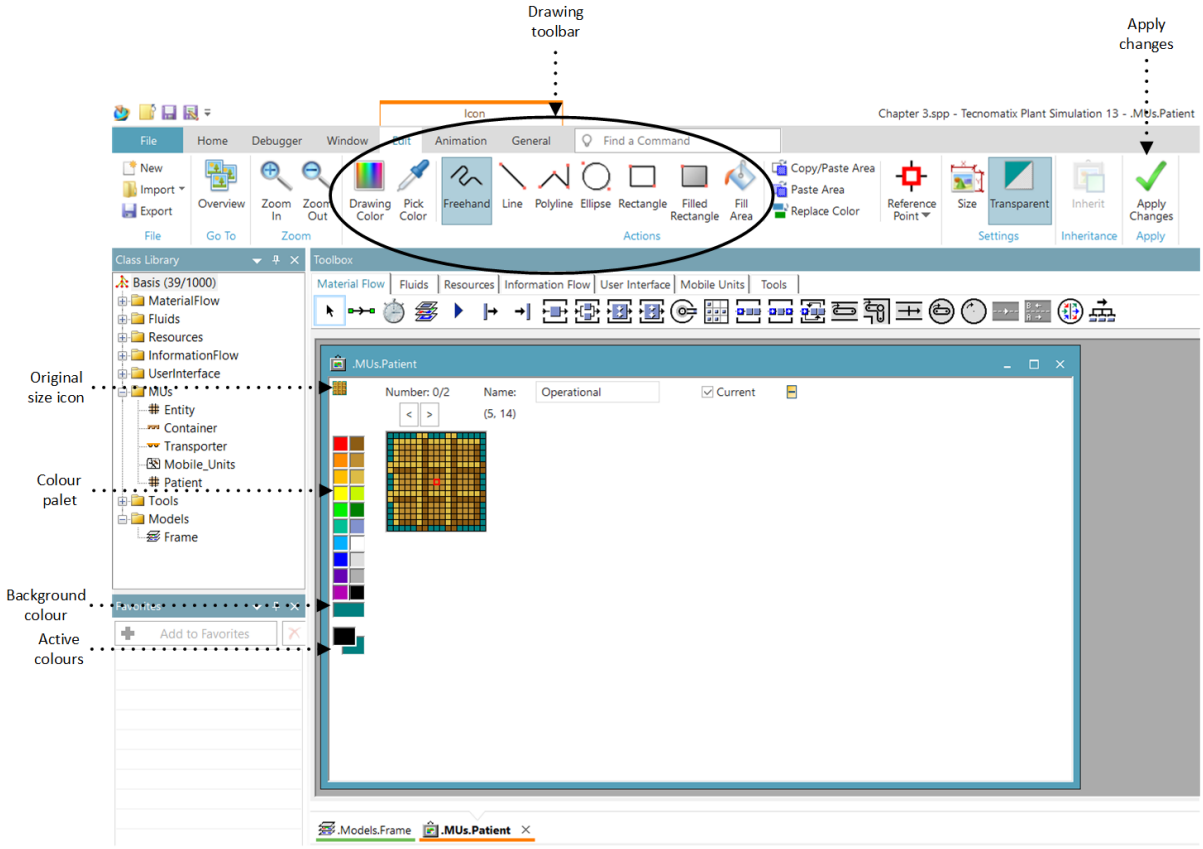
5. Close the *TableFile*.
6. Test your model.
7. Pause the model and note that some *MU*s are named Child and others Adult.

3.8 Icons

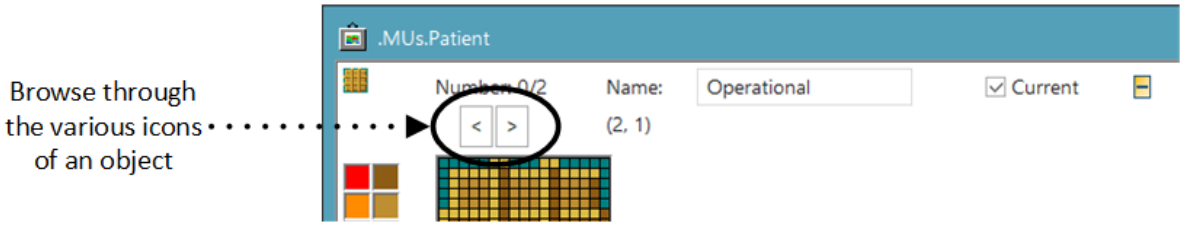
As you might have noticed, it is quite hard to distinguish what type of *MU* is flowing through the model. It could either be an *Adult* or a *Child*. For visualisation purposes it could be useful to create new icons for the objects you have created. Adjusting the icons of your objects might help you to spot errors you have made in your model or help the customer, for whom you perform a simulation study, to understand the model. To create new or adjust current icons, use the *Icon Editor*. You can open the *Icon Editor* by clicking the right mouse-button on the desired object in the *Class Library*, and then choose the menu item *Edit Icons...*



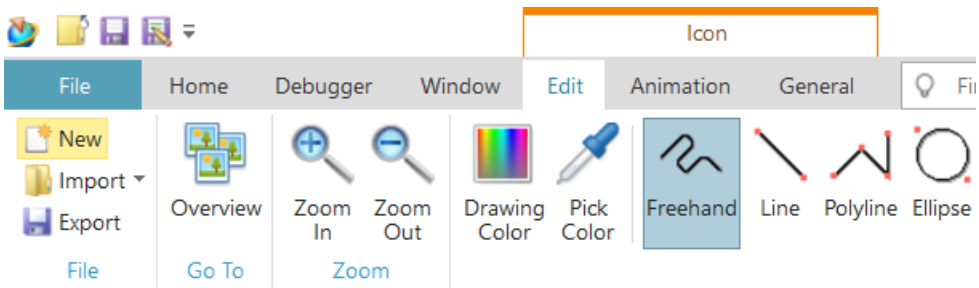
This will open the *Icon Editor* in a new dialog window.



You can edit the icon in the drawing window by clicking on a drawing tool while in *Draw* mode and by selecting a colour from the colour bar. Use the *Copy/Paste Area* tool to select, copy, cut and move parts of the icon. Each object in the *Class Library* is assigned a set of icons. Each icon has a number assigned by the system and a name assigned by the user. The different icons may be used to display different states of the object. The most common state is Operational.



The icon displayed while inserting the object has *Current* checked. You may add additional icons any time. To accomplish this, choose *New* in the *Edit* menu. You can, however, also choose to adjust the standard icons of the object.



Under the *Edit* menu you can also adjust the size (measured in pixels) of an icon. To remove an icon, choose *Cut* (*Ctrl + X*) in the *Home* menu.

Did you know?

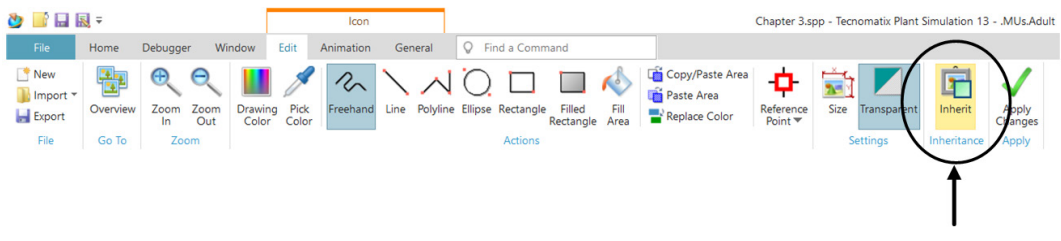
Here are some helpful hints for dealing with icons:

- You may set one or several icons for an object. In case there are multiple icons present at one object, these icons can change state dependent (e.g., operational and waiting) or this change can be triggered using a *Method* (using the command *CurrIcon*).
- Each icon is automatically assigned a number by Plant Simulation and optionally a name by the user. Names have to be unique, i.e., an object may not have multiple icons with the same name.
- The number of icons assigned is unlimited.
- Each basic object in Plant Simulation has a number of pre-defined states (operational, failed, pause, blocked). Icons having this name will be automatically displayed when being in that state. This will be demonstrated later on.
- Each object has an icon with the number (No.) 0, named *Default* and a size of 41 * 41 pixels. This is the icon used for display in the *Class Library*. The name *Default* of this icon may not be changed. In a model though, this object may have a different icon.
- The maximum size is 4000 * 4000 pixels.
- As icons take up a considerable amount of memory, the icon set of an object is only saved once. All objects of this type use the identical set of icons. If an icon is changed, this change is passed on to all instances of this object.
- It is also possible to import .BMP and .GIF-images as icon.

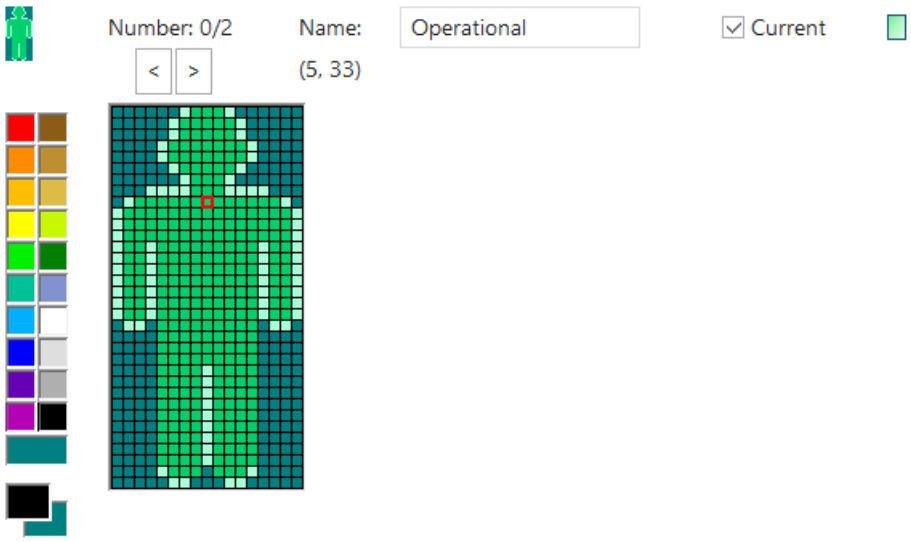
In order to distinguish between a Child and an Adult in the model, you need to modify the icons of the *MUs Adult* and *Child*.

Task: Modify the Icons of the Adult and Child

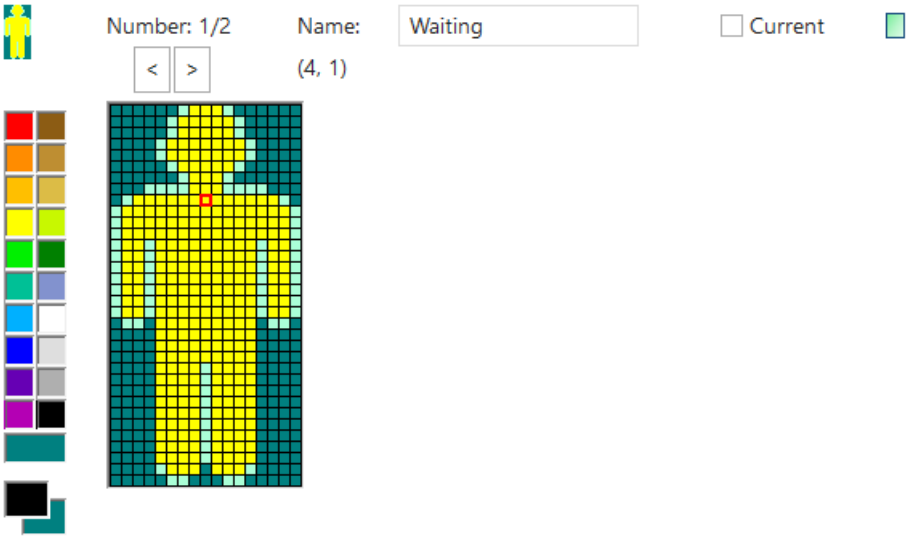
1. Open the *Icon Editor* for the *MU Adult*.
2. Make sure that the button *Inherit* is turned off (because you do not want to use the custom icon for both the Patient and Adult class).



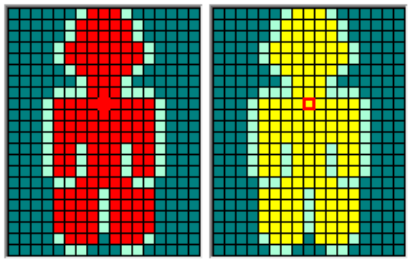
3. Draw an image for the *MU Adult*. For instance:



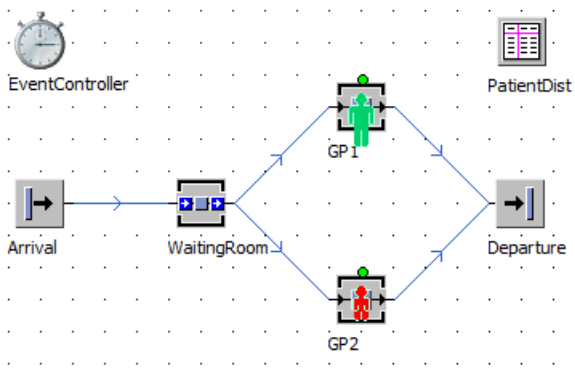
4. Make sure you apply the changes (clicking the green checkmark or pressing F7).
5. Browse in the *Icon Editor* to the image with the name *Waiting*.
6. Again make sure that the button *Inherit Image* is off.
7. Draw an image (or copy/paste from the previous image) that illustrates that the *MU* state is *blocked*, e.g.:



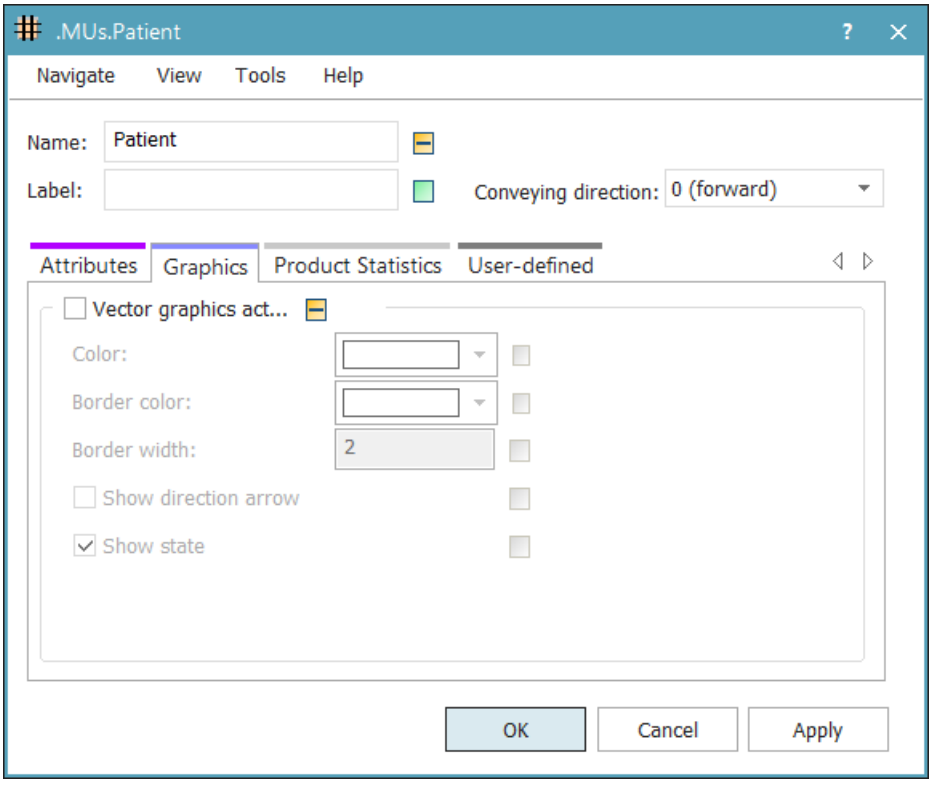
8. Repeat Steps 1 to 7 for the *MU Child*. For instance:



9. Run your model. The icons you drew will be present in the model.

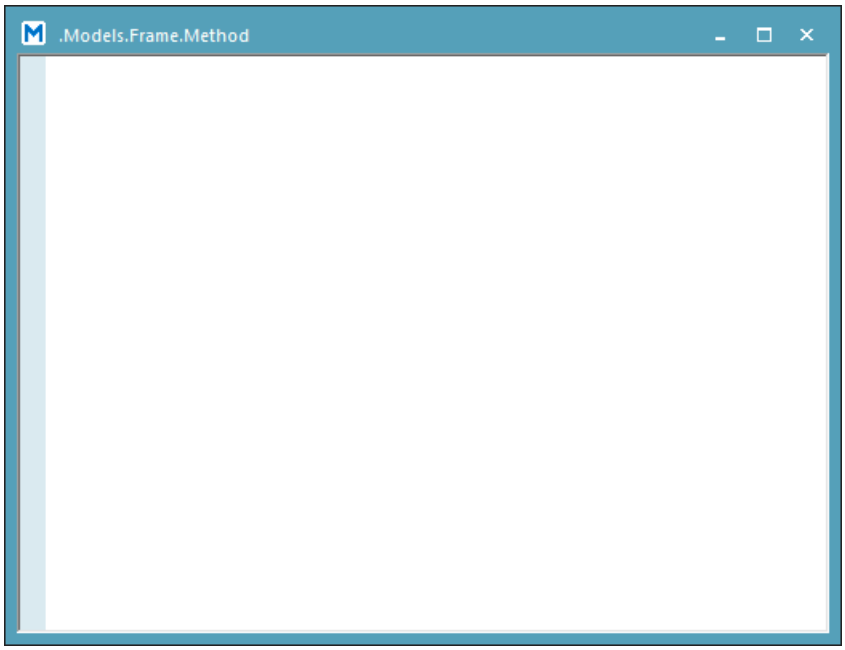


If the dark-green areas in the illustrated icons are not transparant, then open the *Icon Editor*, and turn on *Transparent* in the *Edit* menu. If you do not see the icons at all, you might need to turn off the *VectorgraphicsActive* property. You can do this for both the MUs *Adult* and *Child* by double-clicking on the MU *Patient* in the *Class Library*. Then go to the tab *Graphics* and uncheck the checkbox before *Vector graphics active*.



3.9 Specialised General Practitioners

In the model so far, we have treated the general practitioners as being identical. We now assume that the general practitioners have a specialisation in either treating children or in treating adults. The behaviour of the basic Plant Simulation objects are not sufficient for this purpose. Therefore we need to extend the standard features of these objects using custom code. Plant Simulation provides *SimTalk 2.0* for this purpose, which is the standard programming language in Plant Simulation (see the note at the end of this section on different versions of *SimTalk*). With this programming language you can program all kinds of custom logic. You need to use the *InformationFlow* object *Method* to program *SimTalk 2.0* in your model. If you implement a *Method* in your model you will get the following dialog window by double-clicking on the object.



In a *Method* you declare the local variables you intend to use at the first lines of the *Method*, by making use of the keyword var. The actual logic of your function/procedure will be inserted after the declaration of the variables. After you are done programming your *Method*, you apply the changes by clicking the green checkmark in the *Ribbon* or by pressing F7.

Task: Send the Patient to the Waiting Room

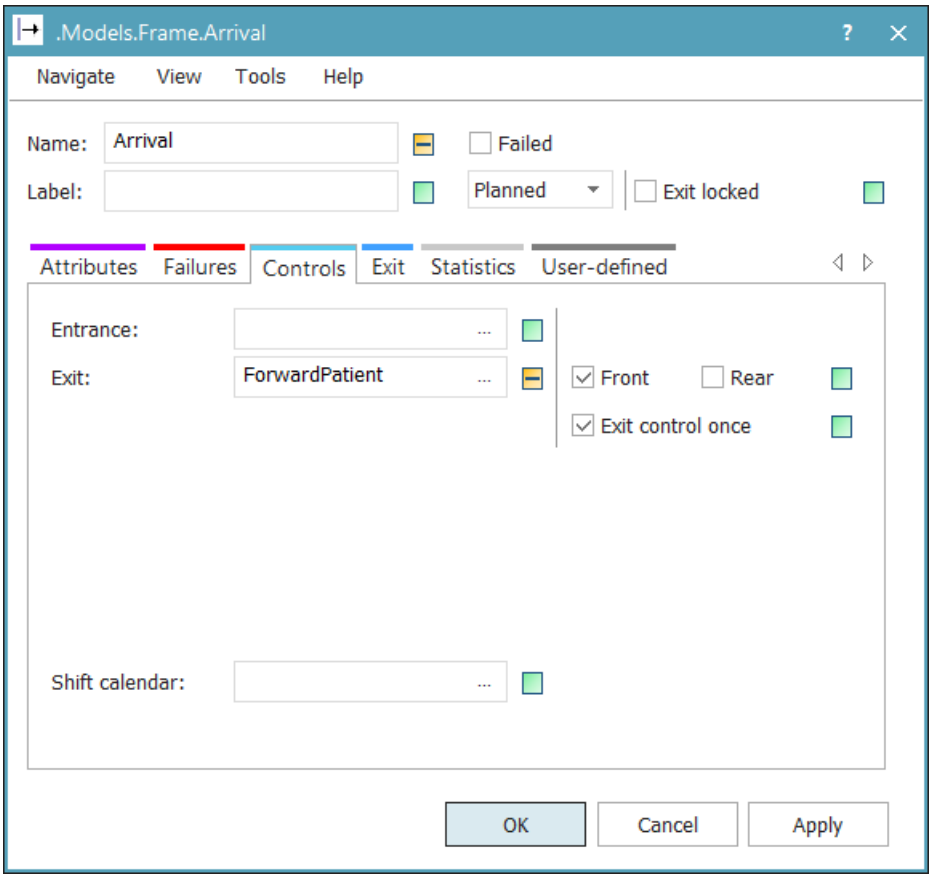
1. Remove the *WaitingRoom* object from your model.
2. Rename *GP1* to *AdultGP* and *GP2* to *ChildGP*.
3. Insert two *Sorters* in your model. Name them *AdultWR* and *ChildWR* respectively (WR stands for WaitingRoom).
4. Set the capacity of both sorters to infinity (-1) and keep all other values to their default settings.
5. Connect the *AdultWR* to the *AdultGP*.
6. Connect the *ChildWR* to the *ChildGP.*
7. Insert a *Method* object and name it *ForwardPatient*.
8. Double-click on *ForwardPatient*.

9. Insert the following code:

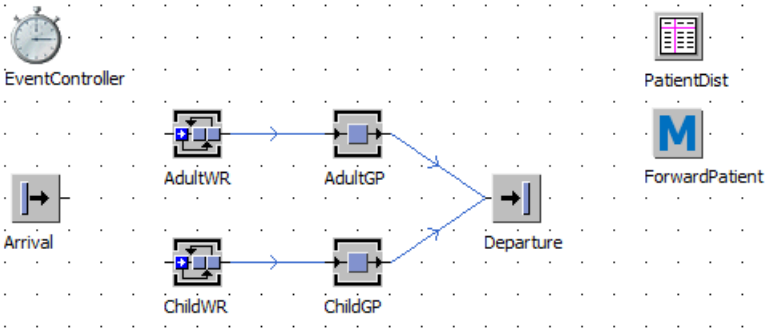
```
if @.origin = .MUs.Adult
    @.move(AdultWR)
elseif @.origin = .MUs.Child
    @.move(ChildWR)
else
    debug
end
```

(@.origin refers to the parent class in the *Class Library*; alternatively, you may use @.name = "Adult")

10. Apply the changes and close the dialog window.
11. Double-click on the *Arrival* object.
12. Click on the tab *Controls* and choose as *Exit* control the *Method ForwardPatient*.



13. Close the dialog window.
14. Test your model.
15. In case you made an error in one of your *Methods* (in this case this might happen in *ForwardPatient*), the program stops and shows a red bar on a programming line resulting in an error and provides error information at the bottom of the *Method* window. In such cases, you need to terminate the simulation by pressing Ctrl+T or using the button  .



As you can see, there are no connectors between *Arrival* and the waiting rooms anymore. The reason is that the *Method* now takes care of moving patients to the waiting room (using the @.move() command). In fact, it is strongly recommended to remove a *Connector* when a *Method* takes care of moving *MUs*, to prevent unexpected behaviour. The more sophisticated your model gets, the more often you will find yourself using *Methods* to move *MUs* from one *MaterialFlow* object to another. Whereas the *Connector* is convenient for simple models, you will often need more fine-grained control over the movement of your *MUs*, for instance to properly route them, or let them dwell in a *MaterialFlow* object for a longer time.

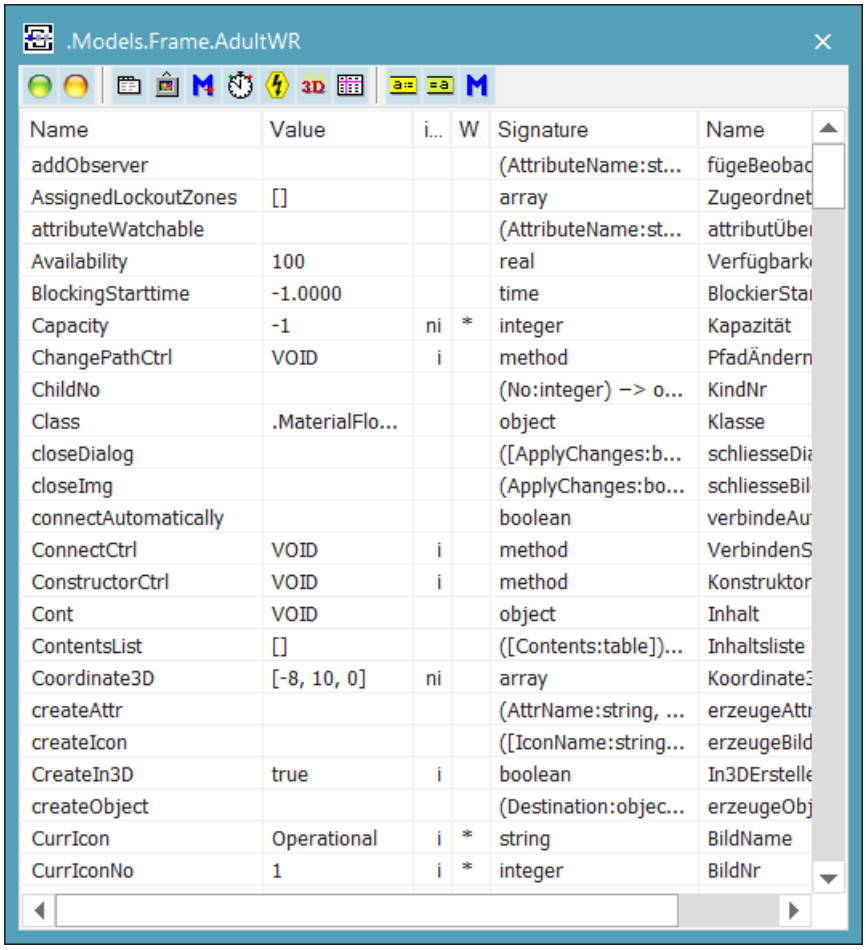
In the current model, the *push* principle is illustrated, since the patients are being pushed to their destination. The *Method* object in the model can trigger different actions (either to go to the *AdultWR* or to the *ChildWR*). This way of linking material flow and information flow is called the **sensor actor theorem**.

Note: The Anonymous Identifier

We learned that material flow and information flow may be connected by the sensor actor theorem. A *MU* that reaches the exit of a material flow object activates the exit control of this object by its exit sensor (similarly for the entrance control). In the control you may employ the character "@" to access the *MU* without knowing its name. The "@" is a cursor (reference) to the *MU* that activated the sensor and caused the *Method* to be executed. There is also the anonymous identifier "?", which is a cursor to the *MaterialFlow* object that holds the triggering *MU* (see Section 4.2).

Did you know?

You can get an overview of all attributes and methods of an object by right-clicking on the object and selecting *Show Attributes and Methods…* or by pressing F8 when you have selected the object. This might be useful when you program certain interactions with the object, and you are curious as to what attributes and methods exist to manipulate the object. Click on the green bullet to only show the standard attributes, which suffice for most simulation models.



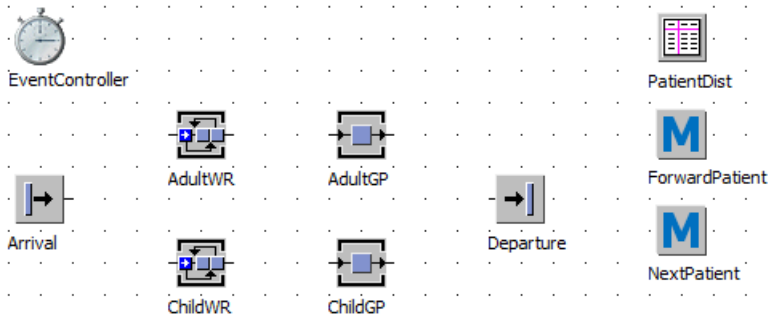
In the model you created there is not only a *push* feature, but also a *pull* feature (see Section 3.6). When a patient is ready at the general practitioner, the general practitioner needs to call the next patient from the waiting room, i.e., need to *pull*. For illustrative purposes, we implement this pull feature in a *Method* also.

Task: Select the Next Patient

1. Delete all remaining connectors from the model.
2. Insert a *Method* object, name it *NextPatient,* and open it.
3. Insert the following code (see also code in the Appendix) and try to understand it:

```
-- If the patient enters a waiting room, check if there is room at the
-- respective GP. If so, push the patient to the GP.
if ? = AdultWR and not AdultGP.Occupied
    @.move(AdultGP)
elseif ? = ChildWR and not ChildGP.Occupied
    @.move(ChildGP)
end

-- If this is an exit event from a GP, then move the patients to the exit.
-- Also check whether a new patient can be pulled to the GP.
if ? = AdultGP
    @.move(Departure)
    if AdultWR.Occupied
        AdultWR.MU(1).move(AdultGP)
    end
elseif ? = ChildGP
    @.move(Departure)
    if ChildWR.Occupied
        ChildWR.MU(1).move(ChildGP)
    end
end
```

4. Apply the changes and close the dialog window.
5. Open the *AdultGP* object.
6. Click on the tab *Controls* and choose as *Exit* control the *Method NextPatient*.
7. Close the dialog window.
8. Open the *AdultWR* object.
9. Click on the tab *Controls* and choose as *Entrance* control the *Method* NextPatient.
10. Repeat Steps 5 till 9 for the *ChildGP*.
11. Test your model.



Note that the attributes used in the *Method NextPatient* can all be found by looking at the attributes of the individual objects *AdultWR*, *AdultGP*, *ChildWR*, and *ChildGP* (right-clicking these objects or pressing F8, see earlier remark). Instead of using the attribute *Occupied*, we also could have used the attributes *full*, *empty*, or *numMu*:

"Not AdultGP.Occupied" = "AdultGP.empty" = "AdultGP.numMu=0"

Note that these equalities do not necessarily hold for other objects than the *SingleProc*. Finally note that we now have removed all *Connectors* from the model, such that all *MU* movements are handled by *Methods.*

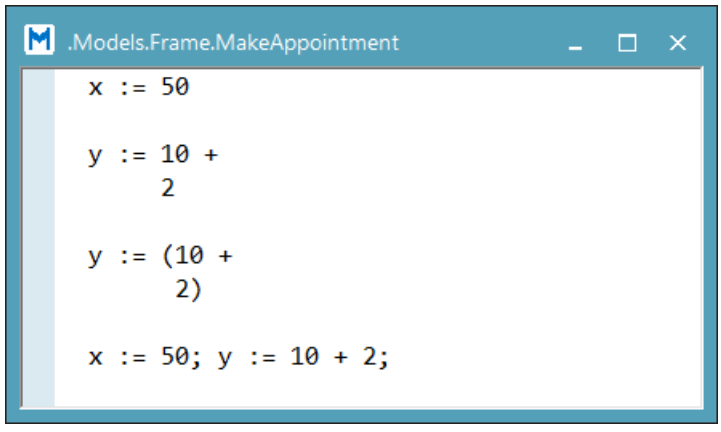
Note: *SimTalk 1.0* and *SimTalk 2.0*

The built-in programming language *SimTalk* provides great modelling flexibility, since you are no longer bound by the functionalities of existing Plant Simulation objects. The *SimTalk* source code needs to be entered into *Methods*. There are two versions of *SimTalk*, the original *SimTalk 1.0* and the new *SimTalk 2.0*. You can select a version with the ribbon button *Method > Tools > New Syntax*. Both types of *Methods* can be used in the same model in parallel. However, *SimTalk 2.0* makes programming *Methods* in Plant Simulation faster, easier, and less error-prone. Throughout this tutorial, we use *SimTalk 2.0*. The main differences of this version compared to the original *SimTalk 1.0* are the following:

- You no longer need to type a semicolon at the end of each statement.
- You no longer need to use the keywords is, do, end for your code, where all variables need to be declared between is and do, and the rest of the code needs to be placed between do and end.
- Simplified control flow statements (e.g., if-else-end instead of if-then-else-end).
- New operators for adding, subtracting, or multiplying a value (e.g., x += y is short for x := x + y)
- Changed about equal operators (e.g., ~= instead of ==)
- New div/mod operators.
- Different way of referencing of methods and global variables.
- An improved syntax for *Lists* and *TableFiles*.

Note: Structuring your code in *Simtalk 2.0*

The line end defines the end of a statement. If the statement is not yet complete (e.g., y:=10+), then it will automatically continue on the next line. You can use parentheses to force the statement to continue on the next line. The semicolon separates multiple statements in one line. Be careful with line enters and use proper indentation to improve readability and to avoid mistakes.

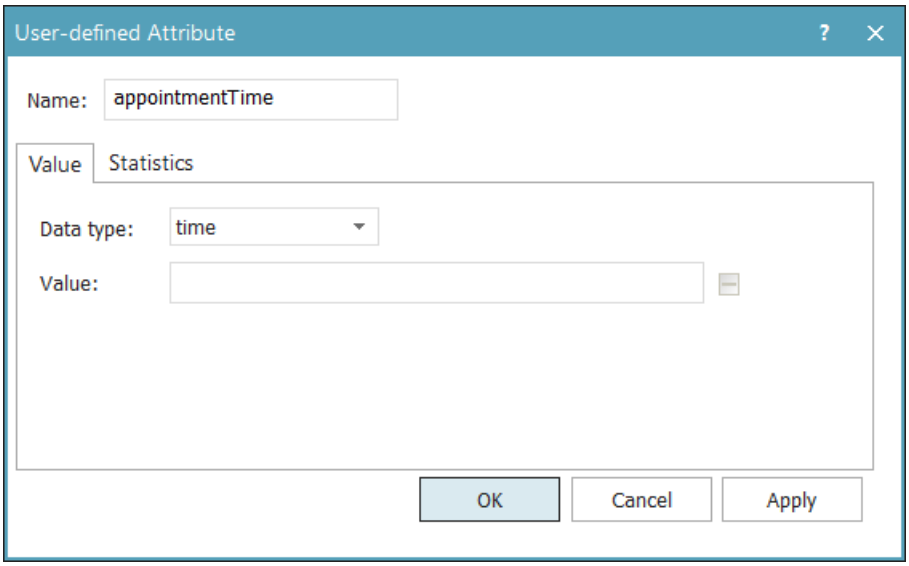


3.10 User-defined Attributes

In a realistic simulation study it is likely that more information is known about the patients. It is easy to imagine that every arriving patient will have a certain appointment time with their GP and that some patients will be late, while others arrive early. In order to realise such a situation in Plant Simulation, it is necessary to create *User-defined Attributes* for our patients.

Task: Create a User-defined Attribute

1. Open the *MU* Patient.
2. Click on the tab *User-defined*.
3. Click on the button *New*.
4. Name the attribute *appointmentTime*.
5. Set the data type to *time*.



6. Click OK and close the dialog window.
7. Note that due to inheritance the *MU Child* and *Adult* will also have this attribute.

Did you know?

In addition to standard attributes, each basic material flow object may be appended with additional information formed by customised attributes. You might use it to add information concerning part type, order number, state of processing, etc. Each customised attribute consists of a *name*, a *data type*, and a *value*. It can only save a single value. The number of customised attributes is *not* limited. A customised attribute corresponds to a global variable in a programming language.

Plant Simulation offers several data types. The most important ones are listed here:

- boolean — logical value, may be TRUE or FALSE.
- integer — integer number, such as 1, 127, -3566.
- real — floating point number, such as 3.141, -382.657.
- string — text, such as "This is text", "123 – ABC".
- object — cursor to an object.

The data type determines the *value range*, i.e., which entries are permitted and which are not. Depending on the data type, different *operators* are available. Operators allow you to perform arithmetical and logical operations on variables. Important operations are:

- boolean: AND, OR, NOT, =, /=
- integer: +, –, *, +=, –=, *=, div, mod, ...
- real: +, –, *, /, +=, –=, *=, ...

The following examples illustrate how these operators work:

- 10 mod 3 gives 1, 11 mod 3 gives 2, and 12 mod 3 gives 0.
- 10 div 3 gives 3, 11 div 3 gives 3, and 12 div 3 gives 4.
- If x = 10, then x += 3 gives 13 (equal to x := x + 3, see the note later in this section).
- If x = 10, then x –= 3 gives 7 (equal to x := x – 3).
- If x = 10, then x *= 3 gives 30 (equal to x := x * 3).

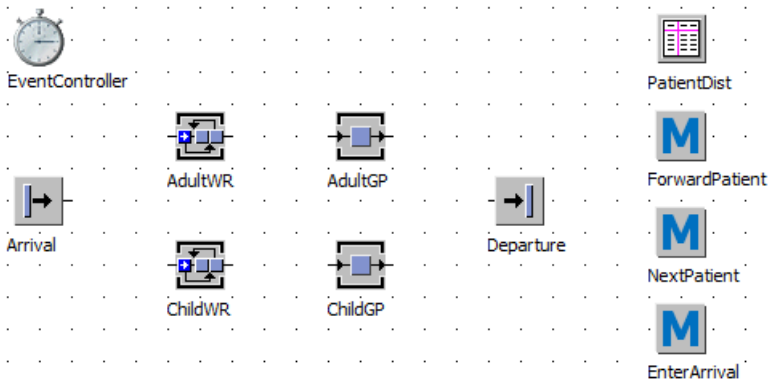
In order to generate an appointment time for our patients, we again make use of the *Method* object.

Task: Define the Appointment Time

1. Insert a *Method* object, name it *EnterArrival* and open it.
2. Insert the following code:

```
@.appointmentTime := EventController.SimTime + z_uniform(1, 0, 1800)
```

   Resembling the situation where patients arrive uniformly between 0 and 30 minutes too early; the 1 refers to a random number stream that we explain later on in this section.
3. Apply the changes and close the dialog window.
4. Open the *Arrival* object.
5. Click on the tab *Controls* and choose as *Entrance* control the *Method EnterArrival*.
6. Run your model and pause it after some time.
7. Double-click on a *MU*.
8. Click on the tab *User-defined*.
9. Note that the attribute *appointmentTime* now has a value.



Note: Comparison versus assignment

You might have noticed that sometimes = or := is used in *Methods*. The = operator is used to compare two variables, for instance in an if statement (e.g., variable1 = variable2). The := operator is used to assign a new value to a variable (e.g., variable := 4.6).

Besides the appointment time of a patient, the urgency or severity of its reason to visit the general practitioner might also be important. We add an additional *User-defined Attribute* for this patient characteristic.

Task: Define the Urgency of a Patient

1. Open the *MU Patient.*
2. Click on the tab *User-defined*.
3. Click on the button *New*.
4. Name the attribute *urgency*.
5. Set the data type to *integer*.
6. Click OK and close the dialog window.
7. Double-click on *EnterArrival*.
8. Add the following line to your code:

```
@.urgency := floor(z_uniform(2, 1, 4))
```

   (uniform random number between 1 and 4 rounded down to an integer 1, 2 or 3)
9. Test your model.

Did you know? Random number streams

A typical simulation model uses *random* numbers in numerous places. In fact, computers do not generate true random numbers; they use a function that takes a *seed value* and then generates a stream of numbers that are called *pseudorandom*. The function that generates this stream is called the *pseudorandom number generator*, or *PRNG* for short.

A proper *PRNG* always generates the same stream of numbers for the same seed value. This property is useful for simulation purposes, since experiments can be compared more easily if they use exactly the same stream of random numbers. In Plant Simulation, random number streams are generated in all places where you use a random distribution. Now notice the difference in inputs for the following two specifications of a probability distribution:

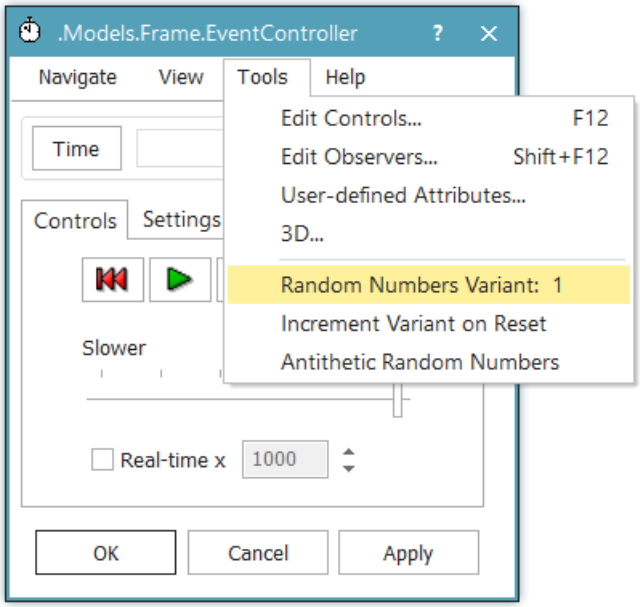
In a *MaterialFlow* object:

In a *Method*:

```
z_negexp(1, 25)
```

In both cases, an exponential random distribution with $\beta = 25$ seconds is specified, but in the *Method*, an additional integer is provided. This is the seed value, or in other words the number of the random number stream from which the random numbers are drawn. It is common practice to use different random number streams for different stochastic processes.

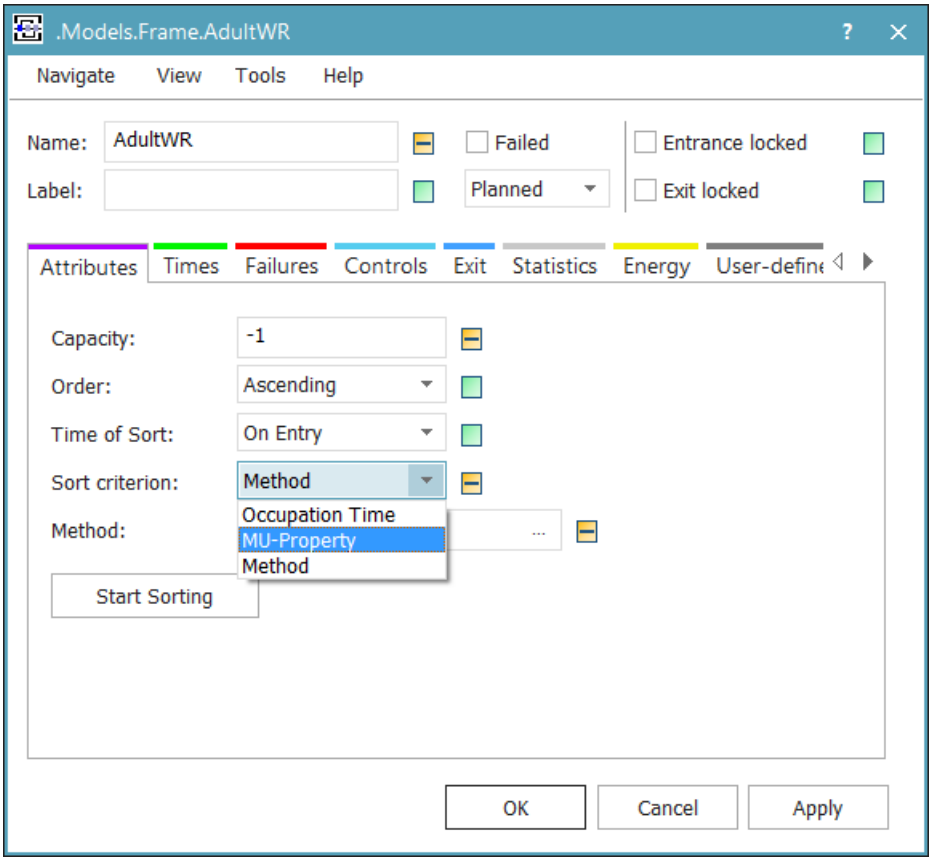
For the *MaterialFlow* objects, it is not possible to specify a specific random number stream since Plant Simulation automatically provides each *MaterialFlow* object with a unique number. However, it is possible to change the random number streams of all *MaterialFlow* objects by setting the *Random Numbers Variant* (see below). For more information on how to use the *RandomNumbersVariant*, see Section 9.4.



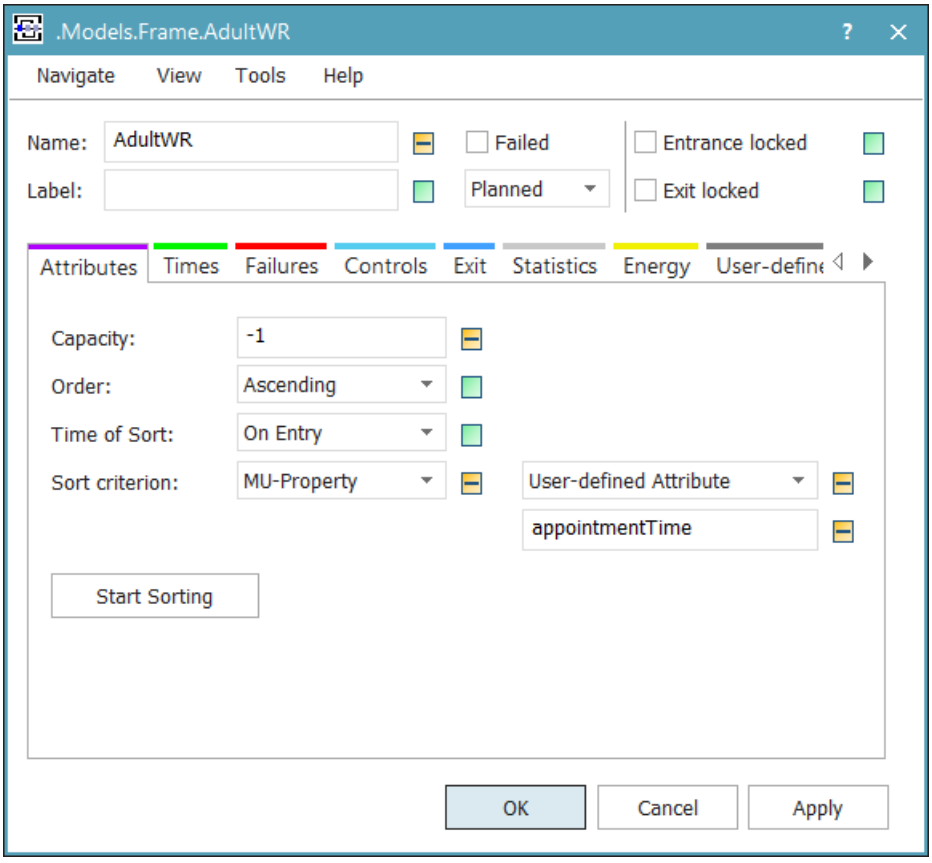
In a fair system, the priority rule for patients, i.e., who is being served first, would not be First Come First Served (FCFS) as currently done in the model. A more desirable way to sort the patients is by sorting them on the appointment time they have. The *Sorter* object that you inserted earlier suits this purpose.

Task: Prioritise the Patients

1. Double-click on the *AdultWR* object.
2. Change the sort criterion to *MU-property*.



3. Select *User-defined Attribute* from the new options that appear.
4. Type *appointmentTime* in the cell below.



5. Click OK.
6. Repeat the steps 1-4 for the *ChildWR*.
7. Test your model.

3.11 Performance Measurement

In order to validate and test your model, you need to install some sort of performance measurement. For this purpose you could use the *Variable* object from the folder *InformationFlow*. The difference with the local variable you define in a *Method* (at the first lines, by making use of the keyword var) is that the *Variable* object is a global variable. This means that the data stored by this variable can be used in all the *Methods*. *Methods* can set and read the value of these global variables.

As Key Performance Indicator (KPI) in this model, we will use the *perceived waiting time*: $wt_{perceived}$. We define it as the difference between a patient's appointment time $t_{appointment}$ and the moment he/she starts the consultation with the GP t_{GP}. However, it is zero if the patient starts the consultation before the appointment time, e.g., because he/she arrived at the GP office early and there are no other patients in the waiting room. Furthermore, the waiting time is perceived to be higher if a patient's urgency is higher (a high urgency corresponds to a low urgency number). In summary, the perceived waiting time is modelled as follows:

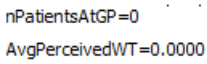
$$wt_{perceived} = \max[\,(t_{GP} - t_{appointment}) \cdot x\,,0\,]$$

where

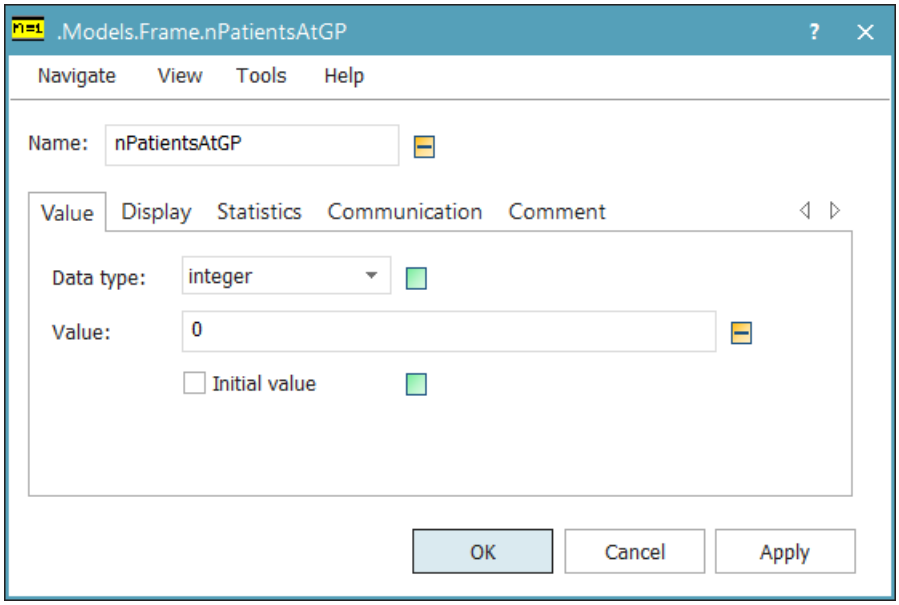
$$x = \begin{cases} 2.0 & \text{if urgency} = 1 \\ 1.0 & \text{if urgency} = 2 \\ 0.5 & \text{if urgency} = 3 \end{cases}$$

Task: Global Variables

1. Insert two *Variable* objects in your model.
2. Name them *nPatientsAtGP* and *AvgPerceivedWT*, respectively.



3. Open *nPatientsAtGP*.
4. Set the data type to *integer*.



5. Click OK.
6. Open *AvgPerceivedWT*.
7. Set the data type to *time*.
8. Click OK.

If you would run your model now, the values of the global variables will not change, because they are not updated by a *Method*. We therefore need to insert a *Method* that sets the value of these global variables. We also need a *Method* that will reset the global variables when we reset a simulation run.

Task: Method for the Global Variables

1. Insert a *Method* object, name it *CalcPerceivedWT* and open it.
2. Insert the following code:

```
var PerceivedWT: time

if @.urgency = 1
    PerceivedWT := 2.0 * max(0, EventController.SimTime - @.appointmentTime)
elseif @.urgency = 2
    PerceivedWT := 1.0 * max(0, EventController.SimTime - @.appointmentTime)
elseif @.urgency = 3
    PerceivedWT := 0.5 * max(0, EventController.SimTime - @.appointmentTime)
end

AvgPerceivedWT := (AvgPerceivedWT * nPatientsAtGP + PerceivedWT) / (nPatientsAtGP + 1)
nPatientsAtGP  := nPatientsAtGP + 1
```

3. Apply the changes and close the dialog window.
4. Insert a *Method* object and name it *Reset*. This *Method* is called every time you press the reset button in the *EventController* (there are also *Methods* that get called when the model initialises and when the simulation ends, see the “Did you know?” in Section 4.2).

5. Double-click on *Reset*.
6. Insert the following code:

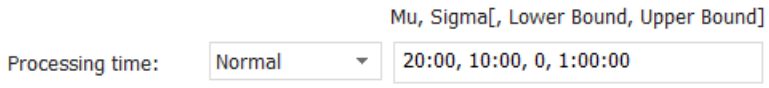
```
DeleteMovables
AvgPerceivedWT := 0
nPatientsAtGP := 0
```

   (the command DeleteMovables removes all *MUs* from the model)
7. Apply the changes and close the dialog window.

We would like to calculate the average perceived waiting time, which means that the *Method* needs to be called when the patient enters the General Practitioner's office. Moreover, to make the model more realistic, we assume that the processing time of the general practitioner is not constant anymore but follows a Normal distribution.

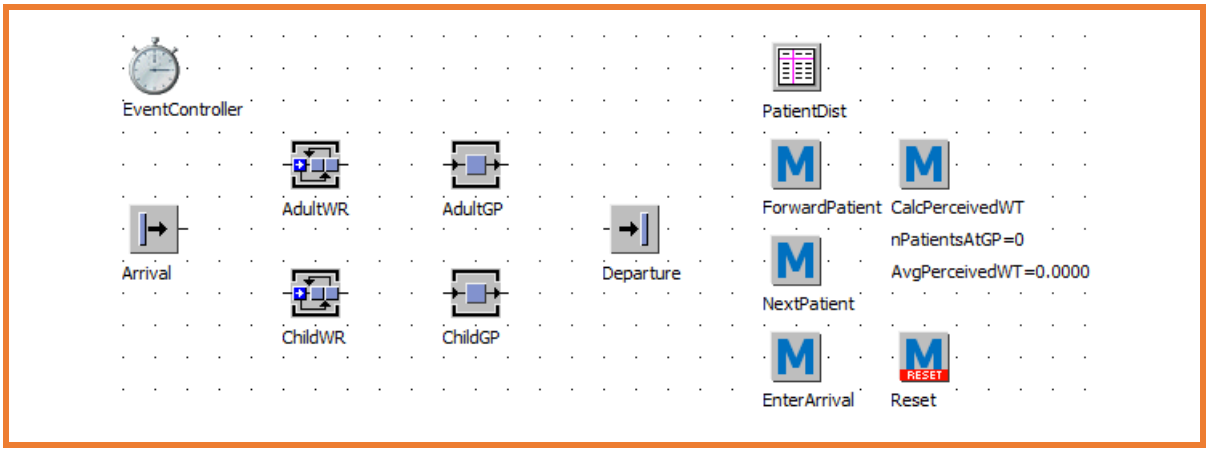
Task: Calculation of the Global Variables

1. Open the *AdultGP* object.
2. Set the *Processing time* to *Normal* and fill in the following parameters:



   (as done here, you can specify an upper and lower bound for the normal distribution; the reason is that an observation from the Normal distribution can theoretically be < 0, which is an invalid *Processing time*, or $+\infty$, which would stall the simulation)
3. Click on the tab *Controls*.
4. Set the *Method CalcPerceivedWT* as entrance control.
5. Click OK.
6. Repeat the steps 1-5 for the *ChildGP*. For the *Processing time* of the *ChildGP* use the following parameters.

7. Open the *Arrival* object.
8. Set the *Interval* to 15 minutes.
9. Click OK.
10. Set an end time of 1000 days in the EventController: open the EventController, open the Settings tab, and type 1000:00:00:00 in the End field (1000 days).
11. Test your model.

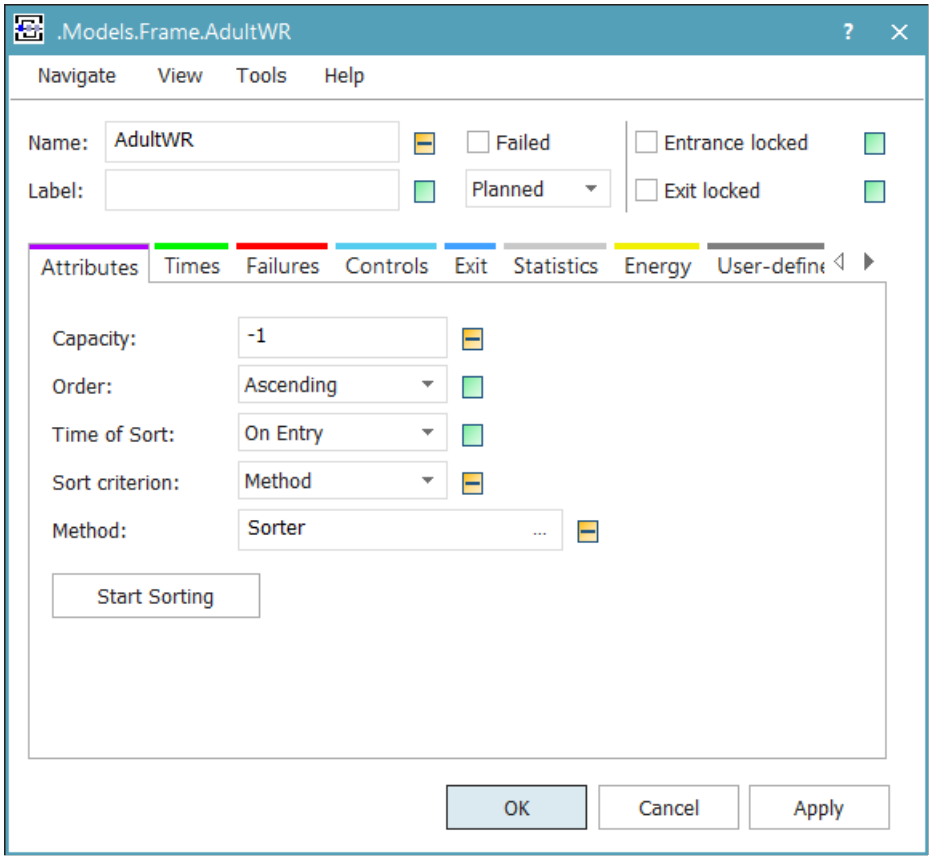


3.12 Sort Using a Method

In some cases, sorting on only one attribute might not be appropriate. There might be a situation in which you would like to sort on various attributes of a *MU* or would like to install some sophisticated rules when sorting. For this purpose we can use the *Method* as a sort criterion for the *Sorter*.

Task: Sorting Method

1. Insert a *Method* object and name it *Sorter.*
2. Double-click on *AdultWR*.
3. Select *Method* as *Sort criterion* and select the *Method Sorter* as input.

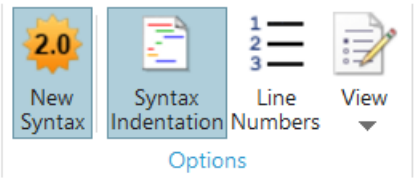


4. Click OK and Yes in the popup window.
5. Repeat steps 2-4 for the *ChildWR*.
6. Double-click on the *Method Sorter* and you will see that the standard layout has changed.

```
-> real
-- @
result := 1
```

You might not see exactly this code, but the code below (left). This is the code from the old syntax *SimTalk 1.0* (see note in section 3.9). You can convert it to the new syntax by going to the *Tools* menu in the *Ribbon* and selecting *New Syntax* (see figure below). If this does not work (happens in some cases in version 13.0), then deselect *New Syntax*, apply changes, close the method, open it again, and select *New Syntax* again.

```
: real
is
do
    -- @
    result := 1;
end;
```



The *Methods* we programmed so far were so-called *Procedures*, which are *Methods* that do not return a value. However, the *Sorter Method* returns a value of the type *real* (this is defined in the first line), which makes this *Method* a so-called *Function.*

Did you know? Structure of Methods

Methods can be used as procedures or as functions, depending on whether the *Method* provides a return (result). As an example, consider the function *AddSubtract* given below.

```
param a, b: real, add: boolean -> real

var c: real

if add
    c := a + b
else
    c := a - b
end
return c
-- instead of "return c" you can also use "result := c"
```

This *Method* requires two reals and a Boolean as input parameters. The Boolean indicates whether the two reals should be added or subtracted. If we call the *Method* with a value of *a* = 1, *b* = 3 and *add* = true, the function will return us as output a value of 4: AddSubtract(1,3,true) = 4. If we create a different case, for instance with *a* = 1, *b* = 3 and *add* = false, this will give us as output a value of minus 2: AddSubtract(1,3,false) = -2. Within the *Method AddSubtract*, the first line states the input parameters param a,b: real, add: boolean and the data type that will be returned by this *Method* "->real". The local variables, i.e., variables only used within this code, are placed after the first line, indicated by the keyword var, in this case only the real "c". At the end of the code, the output will be returned (through the use of return or result).

A sorting *Method* is expected to return a floating-point number (*real*), which can be used to rank the *MUs* in the *Sorter.* When the sorting order is set to *Ascending*, then the lower the value returned by the sorting *Method*, the higher the priority of the corresponding *MU* (i.e., values of the *MUs* increase looking from the front to the back of the queue). When two *MUs* happen to be ranked at the same number, then the *MU* that has dwelled in the sorter for the longest time will be prioritised. This is what currently happens, because every *MU* now gets a rank of 1. We can rank patients on their priority again using a simple modification of the *Sorter Method*.

Task: Sort on Appointment Time with a Method

1. Double-click on the *Method Sorter*.
2. Insert the following code:

```
-> real
-- Return appointment time
result := @.appointmentTime
```

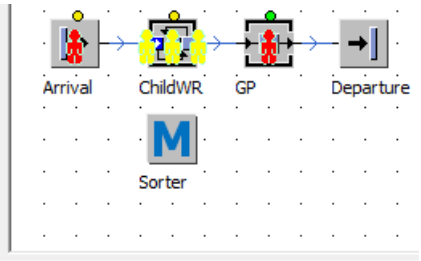
3. Apply the changes and close the dialog window.

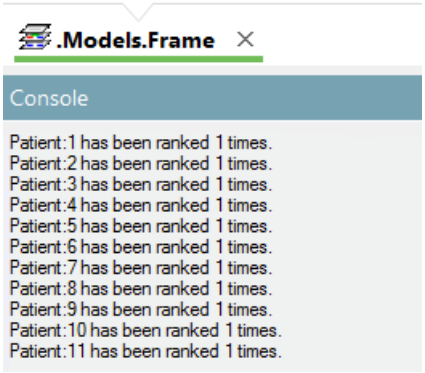
By returning the appointment time as the value from our *Method*, we sort the same way as we have done in the final task of Section 3.9. In the assignment at the end of this chapter, you need to improve the performance of the model by using a *Method* that sorts the patients.

Did you know? When to rank

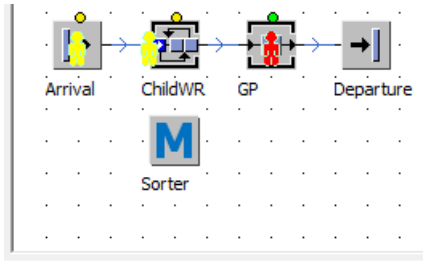
By default, a *MU* gets ranked using the sorting *Method* only when it enters a *Sorter*. This is fine in our case, because the *MU's* appointment time will not change while it is in the *Sorter*. However, there are cases where the ranking for every *MU* must be updated every time a *MU* enters or exits the *Sorter.* You can enable the latter behaviour by setting the *Time of Sort* to *On Access*.

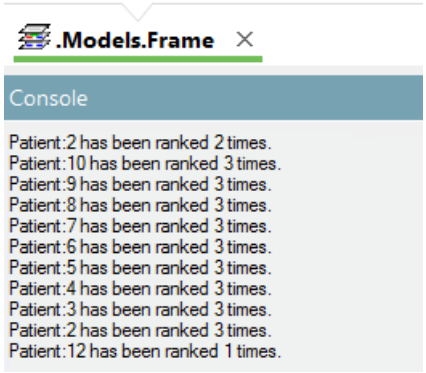
Sorting on entry





Sorting on access





3.13 Assignment A1: Improved Prioritisation

Currently, patients are only prioritised on their appointment time, even though the patient's urgency has a strong effect on the perceived waiting time. Furthermore, patients that arrive at the GP before their appointment time always get a perceived waiting time of zero, such that prioritising urgent but early patients might not always be optimal.

Assignment A1: Suggest an implementation of the sorting rule by modifying the code in the *Sorter Method*. It makes sense to use the *appointmentTime* and/or *urgency* in your sorting *Method*. Note that changing the *Sorter Method* might also require adjusting the settings of the *Sorter* (*Order* and *Time of Sort*). To test your solution, run the model for 2000 days (see Section 3.3) and try to get an *AvgPerceivedWT* that is as low as possible. Motivate your choice for the improved appointment rule. It should be possible to reach an *AvgPerceivedWT* below 15 minutes. You are not allowed to change anything else about the model (such as the GP processing times) since that would invalidate the model.

4 Building a Model: Tracking Patients and Performance

If you carry out a simulation study for a client, your simulation report is most likely an advice on how to set-up a certain system, which investments to make, or what process interventions to apply. You form a scientific foundation on how to reach an optimal process design for a certain system and build a simulation model to experiment with organisational interventions under different scenarios. In order to provide credibility for your report, you should never lose sight of two things, namely the verification and validation of your model.

Verification is the process of verifying that your programmed simulation model corresponds to your conceptual model (i.e., your model on paper and your flowcharts). It is important that your model is free of bugs when you start your simulation experiments, and that the client is involved in the modelling process by discussing the modelling assumptions and flowcharts of decisions and processes. This way, the credibility of your model will increase.

Validation is the process of checking whether the simulation model is an accurate representation of the actual system for the particular objectives of the study. One way of validating a model is to compare a dataset of output values from the model with the values that are observed in the real-life system.

In this chapter, we introduce the concepts that are needed for verification and validation. You will build a model that is able to keep track of the patients in your system for which we will use *TableFiles*. For debugging and model validation, tracking your patients might be an important concept, because you would like to see whether your model is realistic and free of errors. By tracking the patients in your model, you might see that a patient is awfully long waiting in the waiting room or that he skips certain processes that he is supposed to follow. In addition to debugging, when your model grows larger and more complex, mobile units (*MUs*) can end up just about anywhere in the model. In order to keep an oversight, it is useful to have a table that contains the current location of all moving units in the model.

In order to check the correctness of your model, it is important that performance measurement is done correctly. You need to gather statistics about your model and patients in order to see whether your model provides realistic results and to determine which adjustments improve the system.

Subjects dealt with in this chapter:

- Validation and verification
- *TableFiles*
- The M/M/2 queue
- Tracking *MUs*
- *Methods*
- Performance measurement
- Exporting data to Excel

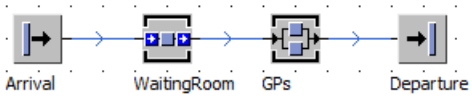
4.1 A Basic Model

For this chapter, we will build a basic model with two servers. The two-server system will have both exponential interarrival times and processing times, such that we have an M/M/2 queue. We will build this model in Plant Simulation using the standard object *ParallelProc*.

The *ParallelProc* has the same basic behaviour as the *SingleProc*, but then with multiple places. The object has an X-dimension and a Y-dimension and the multiplication of these dimensions is the total amount of places in the *ParallelProc*. In a *ParallelProc* without a control *Method*, an arriving *MU* will always be placed on the processing station that has been idle for the longest time in the *ParallelProc*.

Task: Build the Basic Model

1. Start Plant Simulation.
2. Choose *Create New Model.*
3. Click *2D only* in the dialog window that pops up.
4. Insert the following basic objects in your model: a *Source* (name it *Arrival*), a *Buffer* (name it *WaitingRoom*), a *ParallelProc* (name it *GPs*) and a *Drain* (name it *Departure*).
5. Connect the objects.



You need to specify the interarrival time of *Arrival* and the service times of *GPs*, such that the system has exponential interarrival times and processing times. Note that the exponential distribution is often referred to as the negative exponential distribution, which is also the name Plant Simulation uses.

Task: Specify the Interarrival Times and the Service Times

1. Open the *Arrival* object.
2. Set the *Interval* to a *Negexp*-distribution with parameter β = 15:00.

3. Click OK and open the *WaitingRoom* object.
4. Set the *Capacity* to unlimited, the *Dwell time* to zero, and close the dialog window.
5. Double-click on the *GPs* object.
6. Change the dimensions of the *ParallelProc* such that the total amount of places equals two.

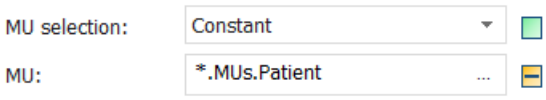
7. Set the *Processing time* to a *Negexp*-distribution with parameter β = 28:00.

8. Click OK and test your model.

Just like in the previous chapter, we would like to have Patients flowing through our model, so we need to derive a new object from the *Entity* in the folder *MUs*.

Task: Create a Patient

1. Derive a new object from the *Entity* in the *MUs* folder.
2. Rename the object *Patient*.
3. Set the object as input for the *Arrival* object.



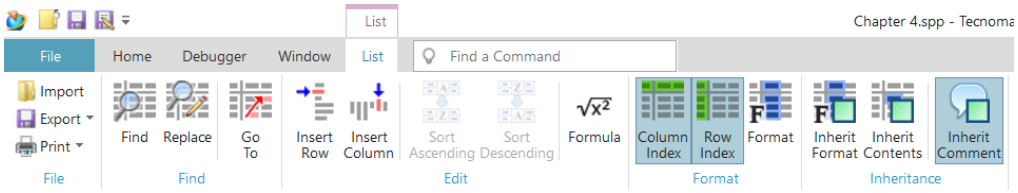
4. Create new icons for the object (for the *Operational Icon* and the *Waiting Icon*).

4.2 Tracking patients in a TableFile

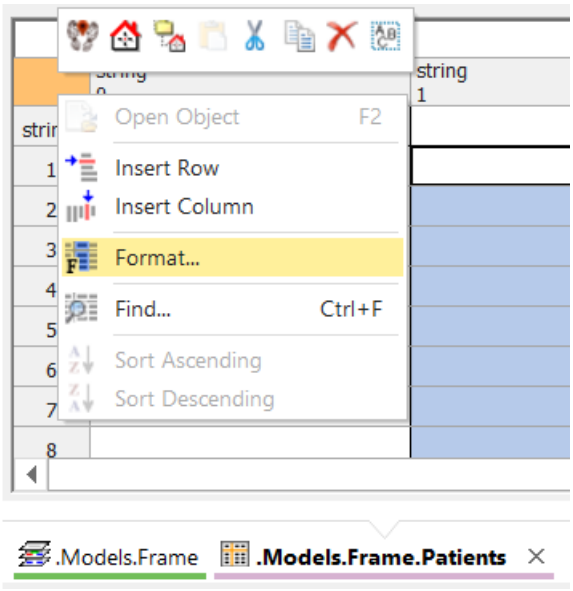
As stated in the introduction of this chapter, when models grow larger, it might be harder to keep track of all *MUs* without the proper tools. In order to keep the oversight, it is useful to have a table that contains the current location of all *MUs* in the model. Furthermore, such a table can keep track of all kind of other unit-level attributes and statistics. We will first create a *TableFile*, which contains all the information we want to collect for each patient. We also set up a counter for the number of patients in the system with the object *Variable* and a *Method* that will clear the *TableFile* automatically when the system resets.

Task: Keep track of the Patients

1. Insert the object *TableFile*, name it *Patients*, and open it.
2. Click on *List* → *Column Index Active* as well as on *List* → *Row Index Active* in the *Ribbon*. Make sure that *Inherit Format* is off, since you want to use a different format of the table in your model then used for the *TableFile* object in the *Class Library*.



3. Right-click on the corner cell of the header row (see the screenshot below) and select *Format…*.



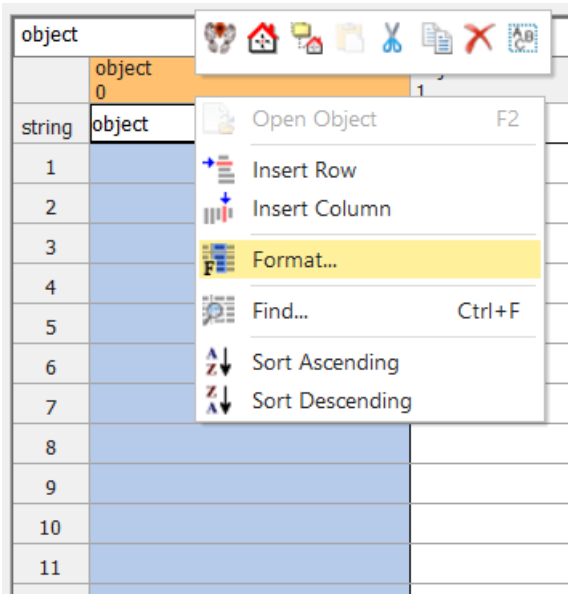
4. In the *Dimension* tab, set the number of columns to 8. Close the dialog.
5. Give the columns the following names and formats:

	object 0	object 1	integer 2
string	object	location	urgency

time 3	time 4	time 5
arrivalTime	appointmentTime	waitingTime

time 6	time 7	time 8
perceivedWaitingTime	arrivalAtGP	processingTime

(you can change a format by right-clicking on a column header and selecting *Format…*)



6. Close the *TableFile* dialog window.
7. Insert an object *Variable* and name it *nPatients*.
8. Insert a *Method* and name it *Reset*.

9. Insert the following code (see the Note at the end of Section 4.3 for an explanation of the argument that is passed to .delete()):

```
-- Clear the table files and reset the patient counter to 0
Patients.delete({0,1}..{*,*})
nPatients := 0

-- Clear all remaining movables in the model
deleteMovables
```

10. Apply changes and close the dialog window.

Did you know?

The *Method Reset* has been briefly introduced in Chapter 3. This *Method* will be called when you press the *Reset* button in the *EventController*. There are also other *Methods* that have a special purpose, namely the *Methods Init* and *EndSim*. The *Method Init* is the first *Method* to be executed when you start the simulation. This can be useful when you want to create a number of *MUs* immediately at the start of each run. The *Method EndSim* is similar to the *Method Init*, but is called at the end of a simulation run.

Now that the basic fundamentals for tracking our patients have been set, we need *Methods* that will update that information. Every time a patient enters a new object (e.g., the *WaitingRoom* or the *GPs*), the information needs to be updated.

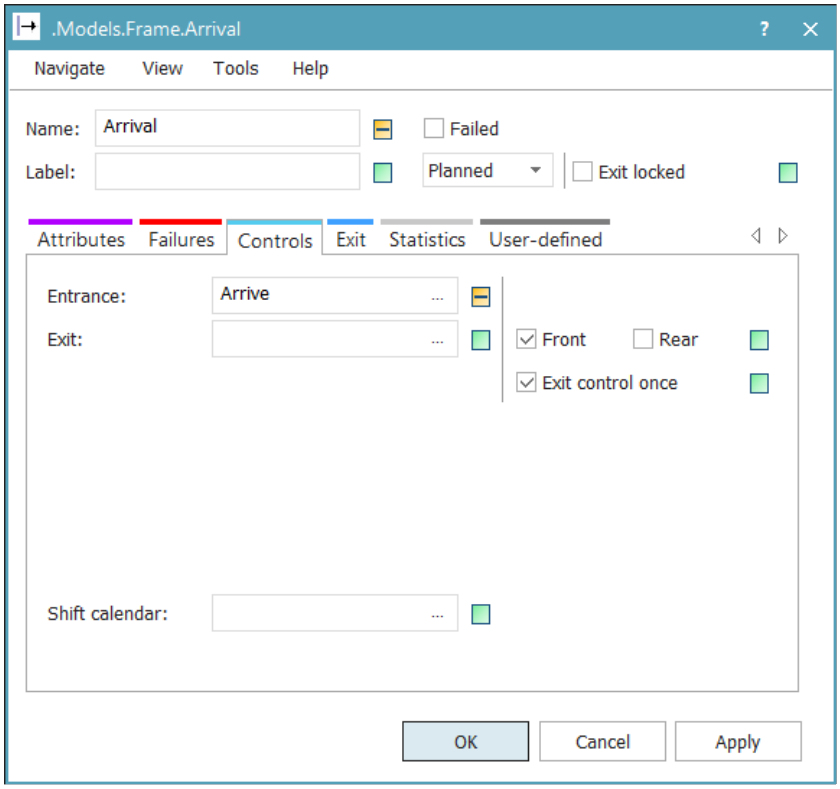
When patients enter the model, a *Method* must create a new row with the information about the *Patient* in the *TableFile Patients*. The initial information available about the *Patient* is its arrival time, appointment time, and urgency. In contrast with the previous chapter, we do not set the *User-defined Attributes appointmentTime* and *urgency* of an individual *Patient*, but instead we write these values directly to a *TableFile* keeping track of the performance data of all patients.

Task: Create an Arrival Method

1. Insert a *Method* and name it *Arrive*.
2. Insert the following code:

```
var n: integer

-- Enter the patient into the table file.
n := nPatients + 1
nPatients := n

Patients["object", n]           := @
Patients["location", n]         := ?
Patients["arrivalTime", n]      := EventController.SimTime
Patients["appointmentTime", n]  := EventController.SimTime +
                                   z_normal(1, -300, 1800)
Patients["urgency", n]          := floor(z_uniform(2, 1, 4))
```

3. Apply the changes and close the dialog window.
4. Open the object *Arrival*.
5. Set the *Method Arrive* as *Entrance Control*.



6. Click OK and test your model. Pause it after some time.
7. Double-click on the *TableFile Patients* and see that it is now filled with data.

In *Arrive*, we have defined the *local variable* "n". The only reason for this is code readability, since we would otherwise have to write "nPatients" instead of the much shorter "n" in several places in the code. Moreover, the location of the patient as logged in the *TableFile Patients* is still the *Arrival* object, so we will need to write a *Method* that updates the location of the patient every time it moves. This happens when the patient enters the waiting room, the general practitioner, or leaves the system.

You might notice that the cells in the first column in *Patients* are mostly filled red with the statement (?). This means that the objects these cells point to do not exist anymore. The reason is that patients leave the model once they have been processed, but their corresponding rows are not removed from the table. Moreover, you will see a number 0 in the header of the first column. The column number of the index column, i.e., the column that contains the *Row Indices*, is always zero. See the note at the end of Section 4.3 for more information on *TableFile* indices.

	object 0	object 1
string	object	location
1	(?)	*.Models.Frame.Arrival
2	(?)	*.Models.Frame.Arrival
3	(?)	*.Models.Frame.Arrival

Task: Create the Entrance Controls

1. Insert a new *Method* in your model and name it *UpdateLocation*.
2. Insert the following code:

```
param patient: object, location: object

Patients["location", patient] := location
```

   (this *Method* must be called by other *Methods* such that the location of a moving object is changed when needed)
3. Apply changes and close the dialog window.
4. Insert a new *Method* in your model and name it *EnterWR*.
5. Insert the following code:

```
UpdateLocation(@, ?)
```

   (note that this *Method* calls another *Method* with @ and ? as the two *arguments*)
6. Apply changes and close the dialog window.
7. Open the *WaitingRoom* object and set the *Method EnterWR* as *Entrance Control.* Click OK.
8. Insert a new *Method* in your model and name it *EnterGPs*.
9. Insert the following code (see also code in the Appendix) and try to understand it:

```
var now:          time
var urgency:      integer
var timeSinceApp: time

UpdateLocation(@, ?)

-- Define variables.
now          := EventController.SimTime
urgency      := Patients["urgency", @]
timeSinceApp := max(0, now -
    max(Patients["arrivalTime",@], Patients["appointmentTime", @]))

-- Calculate and save statistics.
Patients["waitingTime", @] := now - Patients["arrivalTime", @]
Patients["arrivalAtGP", @] := now

if timeSinceApp <= 0
    Patients["perceivedWaitingTime", @] := 0
elseif urgency = 1
    Patients["perceivedWaitingTime", @] := timeSinceApp * 2.0
elseif urgency = 2
    Patients["perceivedWaitingTime", @] := timeSinceApp * 1.0
elseif urgency = 3
    Patients["perceivedWaitingTime", @] := timeSinceApp * 0.5
end
```

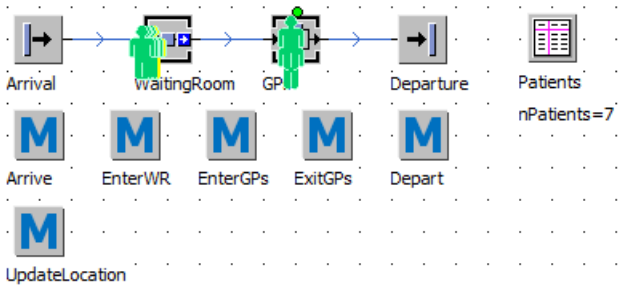
10. Apply changes and close the dialog window.
11. Double-click on the *GPs* object and set the *Method EnterGPs* as *Entrance Control.* Click OK.
12. Insert a new *Method* and name it *ExitGPs.*
13. Insert the following code:

```
Patients["processingTime", @] := EventController.SimTime -
                                 Patients["arrivalAtGP", @]

@.move(Departure)
```

14. Apply changes and close the dialog window.
15. Double-click on the *GPs* object and set the *Method ExitGPs* as *Exit Control.* Click OK.
16. Insert a new *Method* and name it *Depart*.
17. Insert the following code:

```
UpdateLocation(@, ?)
```

18. Apply changes and close the dialog window.
19. Double-click on the *Departure* object and set the *Method Depart* as *Entrance Control.* Click OK.
20. Test your model. Pause after some time and check whether every column in your *TableFile Patient* will be filled for every *Patient*.



We could have programmed this model more efficiently, because in the model so far we have some *Methods* that do exactly the same. We will, however, extend these *Methods* later on. With the current model it is possible to collect all the information about the *Patient*.

Note: @ and ?

In Chapter 3, we have mentioned the anonymous identifier @. This @ designates the *MU* that triggered the control, so in the case of our statement UpdateLocation(@, ?), the @ refers to the specific patient that triggers the *Method* in which the statement is written. The ? refers to the material flow object that holds the @ and does to actual calling, e.g., the object *GPs.*

We pass the patient @ and the calling object ? to the *Method UpdateLocation*. The first line of this *Method* reads param patient: object, location: object, which means that we must pass two objects as arguments when calling *UpdateLocation*. These objects are then available within the *UpdateLocation Method* under the names patient and location. Note that we could have used the shorter notation param patient, location: object.

4.3 Calculating Statistics and Performance Measures

The information that is now being collected in the *TableFile* allows for all kinds of performance analyses. In this case, we want to calculate the average waiting time experienced by a patient, the average perceived waiting time, and the total time spend in the system. It is possible to simply use the *TableFile Patients* for this, but that approach will become inefficient when the number of rows grows larger. Therefore we will implement a second *TableFile*, which stores all the relevant information. We will extend the *Depart Method*, such that it will copy the relevant information to the new *TableFile*, calculates the desired statistics, and then removes the superfluous information from the *TableFile Patients*. Note that due to the additions of a new *TableFile* and global *Variables*, the *Method Reset* also needs to be extended.

Task: Calculate the desired information

1. Insert the object *TableFile* and name it *PatientStats*.
2. Click on *Column Index* in the *List* menu on the *Ribbon.* Make sure that *Inherit Format* is off.
3. Right-click on the corner cell of the header row and select *Format….*
4. In the *Dimensions* tab, set the number of columns to 5. Close the dialog.
5. Give the columns the following names and formats:

	integer 1	time 2	time 3	time 4	time 5
string	arrivalDay	waitingTime	perceivedWaitingTime	processingTime	lengthOfStay

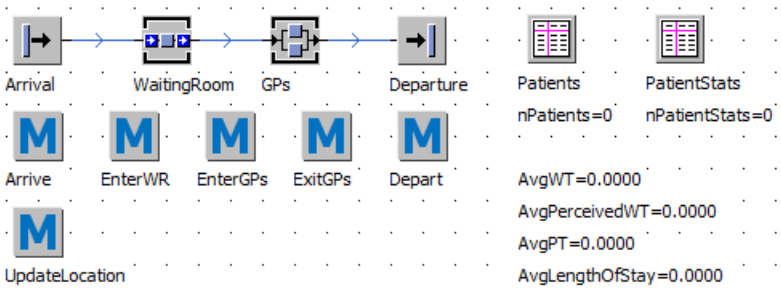
6. Insert a *Variable* object in the *RootFrame* and name it *nPatientStats*.
7. Insert four *Variables* in the *RootFrame* and name them *AvgWT*, *AvgPerceivedWT*, *AvgPT*, and *AvgLengthOfStay*. Set the data type for these four *Variables* to *Time*.
8. Double-click on the *Method Depart*.
9. Add the following code to the existing code:

```
-- Copy patient statistics to PatientStats
nPatientStats += 1
PatientStats["arrivalDay", nPatientStats]           := floor(Patients["arrivalTime", @] / 86400)
PatientStats["waitingTime", nPatientStats]          := Patients["waitingTime", @]
PatientStats["perceivedWaitingTime", nPatientStats] := Patients["perceivedWaitingTime", @]
PatientStats["processingTime", nPatientStats]       := Patients["processingTime", @]
PatientStats["lengthOfStay", nPatientStats]         := @.statAvgLifeSpan

-- Recalculate statistics
AvgWT           := PatientStats.meanValue({2,*})
AvgPerceivedWT  := PatientStats.meanValue({3,*})
AvgPT           := PatientStats.meanValue({4,*})
AvgLengthOfStay := PatientStats.meanValue({5,*})

-- Remove the patients from the Patients table file
Patients.cutrow(@)
nPatients -= 1
```

10. Apply changes and close the dialog window.
11. Double-click on the *Method Reset* and add the following code:

```
PatientStats.delete
nPatientStats    := 0
AvgWT            := 0
AvgPerceivedWT   := 0
AvgPT            := 0
AvgLengthOfStay  := 0
```

12. Test your model.



We consistently use counters like *nPatients* and *nPatientStats* to count the number of rows in a table. This way we can quickly see how much data is contained in a *TableFile*, and *Methods* can use the counter to find the next empty row. The latter can be done without a counter as well. As you can see by right-clicking on a *TableFile* and selecting *Show Attributes and Methods*, the *TableFile* object has an attribute called YDim, which you can use to count the number of occupied rows in the *TableFile*, and consequently the first empty row will be TableFile.YDim + 1. Other related attributes are *XDim*, *indexXDim*, and *indexYDim*.

Notes: TableFile indices

The *TableFile* uses the following notations for indexing a specific cell or range:

- Specific Cell: TableFile[column, row]
- Range: {column n, row n}..{column m, row m}

The range index consists of two attributes, which are separated by two dots. The first attribute of the range specifies the upper left corner of the range and the second attribute specifies the lower right corner of the range. You could also specify a single column or row by only using {column, *} or {*, row}, respectively. The * indicates that the whole X- or Y-dimension of the *TableFile* will be selected.

Some of the *TableFiles* that you have created so far make use of column indices and row indices. These types of indices allow you to use any type of variable as your index. For instance, in the *TableFile Patients*, the column indices are the *strings* object, location, … . The row indices are pointers to object instances, such as *.MUs.Patient:1, *.MUs.Patient:2, …. You already use both types of indices in the *Method Depart*, to find the location of the table cells that need to be copied from *Patients* to *PatientStats.*

4.4 Prioritising Patients using a TableFile

In our current model, we are able to collect all kind of performance statistics. The information stored in our *TableFile Patients* can not only be used to provide the information for our performance measurements, but for instance also to prioritise the patients currently present in the waiting room. Suppose that we need to sort all the patients in the waiting room based on their urgency (whereby 1 is the most urgent and 3 the least urgent), and then on their appointment time. In order to do so, we need to add an extra *Method*. This *Method* will walk through the *TableFile Patients* in search of the patient that has the most urgent injury and the earliest appointment time. There are various *Methods* to search through a *TableFile*. In the following task you use the for-loop, but you could for example also sort the *TableFile* and select the *Patient* in the first row to be sent to the general practitioner if a place becomes available. The latter approach is displayed in the "Did you know?" at the end of this section.

Task: Prioritise Patients with a TableFile

1. Remove the connectors to and from the *GPs*.
2. Insert a *Method* and name it *GetPatient.*
3. Insert the following code (see also code in the Appendix) and try to understand it:

```
->object
var i,bestUrgency: integer
var winner: object
var bestAppointmentTime: time

-- Initialize worst case
winner              := void
bestUrgency         := 4
bestAppointmentTime := EventController.End+86400

/*
Loop through patients to determine the patients with the highest
urgency and appointmentTime
*/

for i := 1 to Patients.YDim
    if Patients["location", i] = WaitingRoom and
        Patients["urgency", i] <= bestUrgency and
        Patients["appointmentTime", i] <= bestAppointmentTime

        winner              := Patients["object", i]
        bestUrgency         := Patients["urgency", i]
        bestAppointmentTime := Patients["appointmentTime", i]
    end
next

-- Return the patient.
result := winner
```

(remember that ->object in the first line means that this *Method* will return a value with data type *object*; also note that i and bestUrgency are defined on the same line, which is only possible for variables with the same data type, unless you separate the declarations with a semicolon: var i, bestUrgency: integer; var winner: object)

4. Apply changes and close the dialog window.
5. Double-click on the *Method EnterWR* and add the following codes (two separate pieces of code, decide for yourself were to put them):

```
var p: object
```

and

```
if not GPs.Full
    p := GetPatient
    p.move(GPs)
end
```

(note that the *Method GetPatient* is called and the result of this *Method* is assigned to the local variable p)

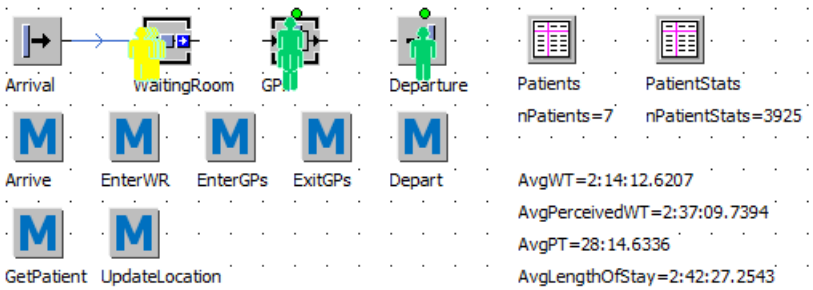
6. Apply changes and close the dialog window.
7. Double-click on the *Method ExitGPs* and add the following codes (two separate pieces of code, decide for yourself were to put them):

```
var p: object
```

and

```
if WaitingRoom.Occupied
    p := GetPatient
    p.move(GPs)
end
```

8. Apply changes and close the dialog window.
9. Set an end time of 100 days in the EventController: open the EventController, open the Settings tab, and type 100:00:00:00 in the End field (100 days).
10. Test your model.



Did you know?

There are various options to search a *TableFile,* each with different advantages and disadvantages. Below you can find an alternative to the option we have used in the previous task (the for-loop). However, even though this approach takes less lines of code, it is certainly not more efficient in our case (it is not efficient to sort all patients if we need to select a single patient).

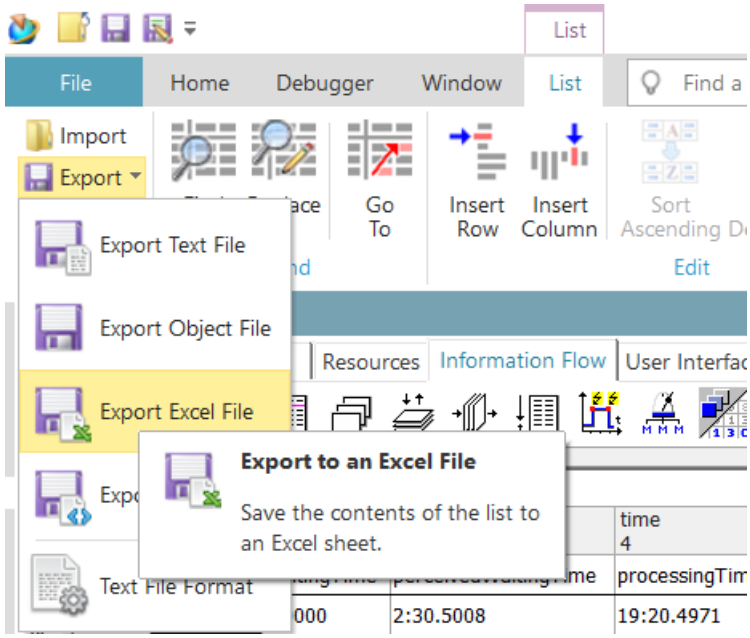
```
->object
var i: integer -- Row number
var p: object -- Patient

/*
Sort patients first by urgency and arrivalTime
Then set the cursor to the beginning of the table
and start searching for the first row that has
location WaitingRoom. Get the row number.
*/
Patients.sort("urgency", "arrivalTime", "up")
Patients.setCursor(1, 1)
Patients.find({1,*}, WaitingRoom)
i := Patients.CursorY

-- Get the patient that corresponds to the row and return it
result := Patients["object", i]
```

4.5 Exporting the Performance Statistics to Excel

As has been stated before, the purpose of a simulation study is to learn about system behaviour and performance. Therefore we have collected data from our model in the *TableFiles* of our model. If we would like to analyse our data further, such as performing statistical tests or computing the confidence intervals, there might be a need to export the data from our model. We will show a way in which you can export your data to Excel.

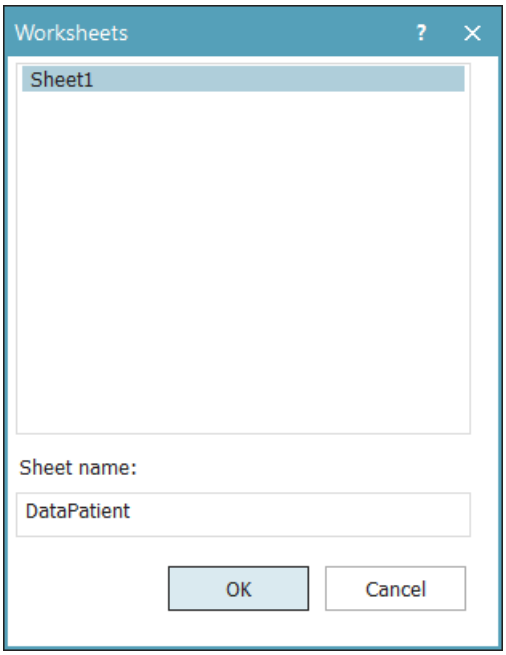
Task: Export Data to Excel

1. Run your model for some time and then pause it.
2. Double-click on the *TableFile PatientStats*.
3. Click on *Export* in the *List* menu on the *Ribbon* and then select *Export Excel File*.



You will get the option to save the data on your computer. It will then ask in which worksheet you would like to store your data.

4. Select a worksheet to save your data and click OK.



Your data is now stored in an Excel file.

Alternatively you can manually select an area of information you would like to save and use the shortcuts Ctrl-C and Ctrl-V to paste it in an Excel file or another spreadsheet program.

Note: Time format in Excel

When copying the data into Excel, you may notice that the program has difficulty reading the columns that are formatted as *time* variables. The easiest way to fix this is to change the format of these columns to *real*. The columns will now contain times in seconds, which are easy to work with in Excel.

In case you prefer to work with time formatting, keep in mind that the *time* format in Plant Simulation is counted in seconds whereas the time format in Excel is counted in days. As such, an hour in Plant Simulation is stored as a floating-point number with value $60 \cdot 60 = 3600.0$, and in Excel an hour is stored as a floating-point number with value $\frac{1}{24} \approx 0.041667$. Thus, to convert a column from Plant Simulation with times denoted in seconds, you must first divide each value with $24 \cdot 60 \cdot 60 = 86400$, and then apply the Time format in Excel.

Also note that Plant Simulation uses a point (.) as the decimal mark by default, whereas Excel might use a comma (,) depending on your regional settings. If you do not want to change the regional settings, you may also copy your data directly into Excel and replace all points by commas.

4.6 Assignment A2: Fitting a random distribution and validation

Currently, the random distributions used for the interarrival times and processing times are Exponential with $\lambda = \frac{60}{15}$ and $\lambda = \frac{60}{28}$, respectively (rate in hours). However, you have some doubts as to whether these random distributions best describe the real world. Therefore, you have asked the General Practitioner's office to collect observations on the actual interarrival times of patients and the times needed for consultation. You receive the two requested datasets measured over the last 2000 patients visiting this office (see accompanying files).

Assignment A2.1. Fit a probability distribution to these two datasets using Excel or a similar spreadsheet program. Determine the probability distribution that best fits the data, estimate the parameters, and perform a goodness-of-fit test. Use the outcomes to update your simulation model.

As a validation measure, the General Practitioner's office performed measurements of the time span between a patient's moment of arrival and its moment of departure, which is referred to as the length of stay. They aggregated the data per day, i.e., the data set contains daily averages of the patients' length of stay (see accompanying file).

Assignment A2.2. Run your model with the new settings for arrival times and consultation times to collect a sample of patients' length of stay observations (available in the *TableFile PatientStats*). First, create a 95% confidence interval of the average length of stay per day. Second, use an appropriate statistical procedure to determine whether the length of stays observed in the model are consistent with the real-world observations.

Hint. It is statistically not correct to validate your model using the individual length of stay observations, since the individual observations are correlated. However, you can use the column *arrivalDay* in *PatientStats* to calculate the average length of stay per arrival day. Compare these *batch means* to the real world data.

5 Building a Model: Experimenting

The timing of events in the real world can often not be predicted with absolute certainty. For instance, when a patient has made an appointment, it is usually not possible to determine in advance whether that patient will be on time or not. However, it is possible to make some estimation of the probability of being on time, or even treat the difference between the arrival time and the appointment time as a random variable as we did in the previous chapter.

Random variables in models allow you to describe the real world more precisely, but they do pose a new problem: how can different configurations of a model be compared if their performance depends on chance? The answer is to treat the performance of a configuration as a random variable as well, which can be estimated with more confidence if a larger sample of observations is available. We then use standard statistical tests to determine which configuration is better, in a similar manner as we validated our model from the previous chapter. This can be done by creating several *replications* of each model configuration, using the *ExperimentManager* in Plant Simulation.

Subjects dealt with in this chapter:

- Event list and scheduling events
- Input parameters
- Creating appointments
- Performance measurement
- Performing experiments

5.1 Specifications

In this chapter, we use the *ExperimentManager* to optimise a General Practitioner's office that has the following specifications:

1. The system has a front desk, a waiting room with infinite capacity, and two general practitioners that provide consultations to patients.
2. Patients must have an appointment before they can be consulted by a GP. Each day has a fixed number of equally spaced appointment slots between 8:00 and 20:00 (the GPs work in shifts such that we have two GPs during these 12 hours, there is no need to explicitly model the change of shift).
3. Each morning at 8:00, a random number of new patients call to the front desk for an appointment. The number of patients that calls follows a Poisson distribution with $\lambda = 48$.
4. The front desk appoints patients a time slot on a first come first serve basis, starting with the first slot in the morning. When all slots for a day are taken, the remaining patients are appointed to the first available slots on a subsequent day. When that day is fully booked as well, new patients are appointed to the next subsequent day, etc.
5. Immediately after setting an appointment, a patient will start waiting at home until the appointment time.
6. A patient will arrive at the GP office at exactly the appointment time. From that moment, the patient starts waiting in the waiting room until consultation at the GP starts.
7. The time required for consultation follows a Lognormal distribution with $\mu = 30$ minutes and $\sigma = 20$ minutes.
8. GPs continue to start new consultations with patients until the waiting room is empty and there are no more planned patients for that day, or until it is 21:00.
9. Patients that have received their consultation leave the system.

10. Patients that are still in the waiting room after 21:00 are immediately sent to the front desk to reschedule. Obviously, rescheduling is something we want to avoid as much as possible. Consultations that started before 21:00 are always completed.
11. A patient that needs to be rescheduled, will be appointed a new time slot by the front desk. They will be appointed to the first available time slot available as usual. There is no preferential treatment.
12. After rescheduling, the patient starts to wait at home again, and he or she reappears at the GP office at exactly the new appointment time.
13. The waiting time at home is perceived to be 20 times less bothersome than the waiting time in the waiting room. Both types of waiting time are counted from the moment the patient has called the front desk for the first time. In other words: the waiting time counters do not reset when a patient reschedules.

The front desk wonders how many appointment slots should be made available each day, such that:

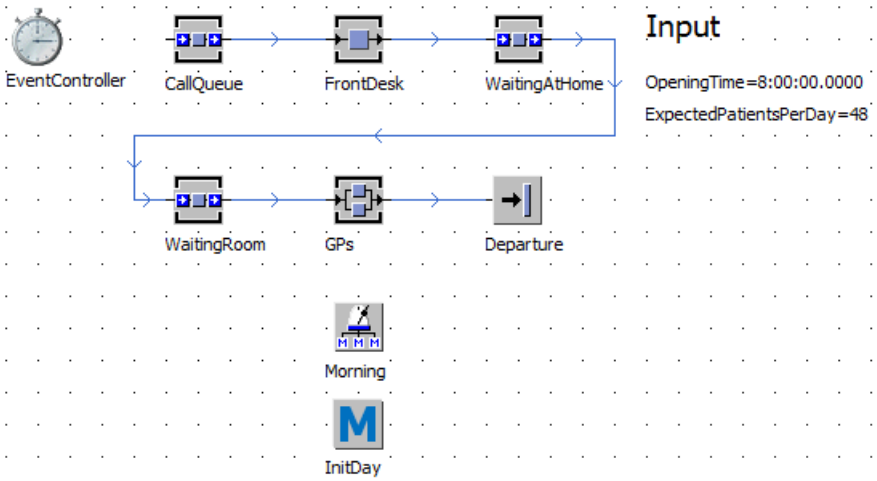
1. The perceived waiting time is minimal. The formula used to calculate the perceived waiting time is $wt_{perceived} = 1/20 \times wt_{at\_home} + wt_{waiting\_room}$.
2. The reschedule rate (average number of reschedules per patient) is less than 0.05.

5.2 General Practitioner's Office with a Front Desk

Before worrying about experiments, replications, or appointments, it is useful to create a simplified version of the final model and expand from there. There is one new object in the next task: the *Generator*. The *Generator* can be configured to call a predefined *Method* at predefined times.

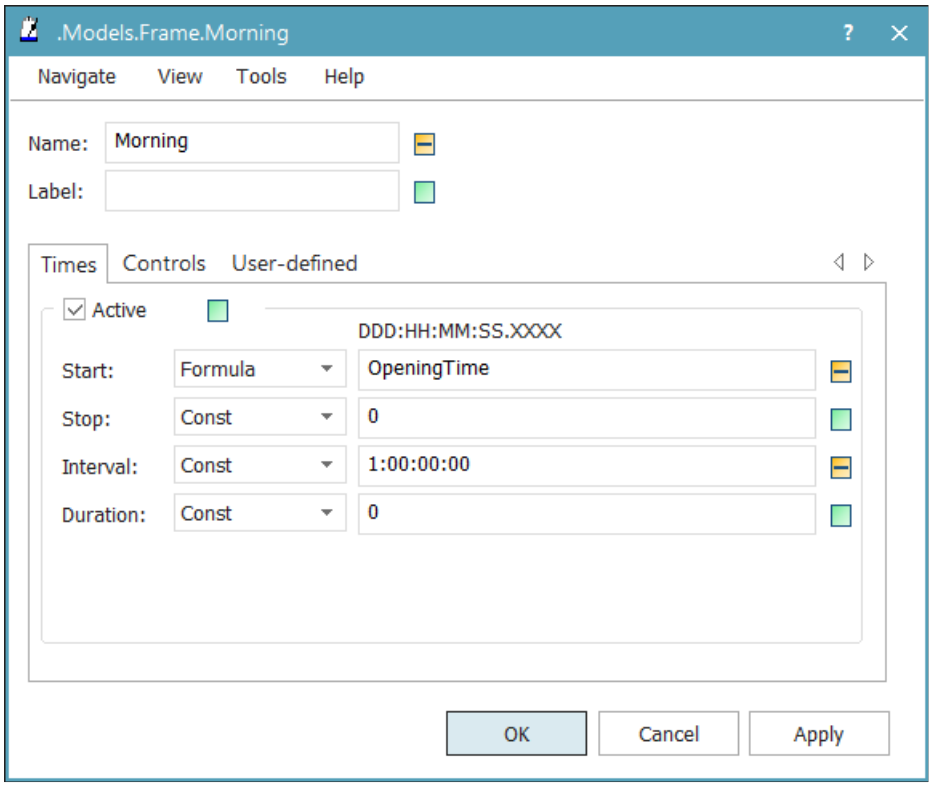
Task: The basics

1. Use the *EventController*, three *Buffers*, one *SingleProc*, one *ParallelProc*, one *Drain*, five *Connectors*, one *Comment*, two *Variables*, one *Generator*, and one *Method* to create the following model (see Note at the end of this section for an explanation on *Comments*).



2. Set the *Capacity* to -1 and the *Dwell time* to 0 in *CallQueue*, *WaitingAtHome*, and *WaitingRoom*.
3. Set the *Processing time* of the *FrontDesk* to 0.
4. Set the number of *GPs* to 2 by setting the Y-dimension to 2.

5. Configure the *Processing time* in *GPs* such that it matches the system specification.
6. Let the *Generator* start at the *OpeningTime* and set its *Interval* to one day. Open the *Controls* tab and enter *InitDay* in the *Interval* field. This way, *InitDay* is called each day at 8:00 o'clock. Note that the *Start* field must be set to the type *Formula*.



7. Derive a *Patient* from the *Entity MU*. Create icons for the *MU*.
8. Let the *Method InitDay* create a random number of patients and move them to the *CallQueue* using the following code:

```
-- InitDay
-- Purpose: Create new patients
-- Called by: Morning
var nNewPatients, i: integer

nNewPatients := 0

for i := 1 to nNewPatients
    .MUs.Patient.create(CallQueue)
next
```

9. Obviously, the code above is not complete yet, since the *Method* creates zero patients. Try to determine the correct code to use on the right hand side part of line 6, such that the correct number of patients is created each day (see the note below for a hint).
10. Set an end time in the EventController: open the EventController, open the Settings tab, and type, e.g., 100:00:00:00 in the End field (100 days).
11. Test the model.

At the top of *InitDay*, you see three lines of comments, which describe the purpose of the *Method* and indicate which objects call the *Method*. *Method* descriptions like these is a must in any good model, since they allow you and other modellers to keep track of what each *Method* does without having to walk through the *Method*'s code itself.

Note: Random distributions in *SimTalk 2.0*

SimTalk 2.0 features built-in functions for drawing random numbers from all common random distributions. These functions all start with z_ and take an integer seed value (corresponding to a unique sequence of random numbers) as first argument. Some commonly used functions are z_poisson, z_normal, z_exp, z_gamma, z_uniform, and z_binomial.

Note: Creating comments

Sometimes, a simple comment can make a model clearer and more appealing. Creating a comment is straightforward:

1. Drag a comment  into the *RootFrame*.
2. Open it and go to the tab *Display*.
3. Here you can type the comments text and apply formatting.

5.3 Adding Events to the Event List

There are several ways to simulate the effect of appointments. One of them is to decide on a patient's appointment time, determine the difference with the current simulation time, and let the patient dwell in a buffer for that time before proceeding to the waiting room. Another way is to use the event list: Plant Simulation holds a list of every event that is scheduled to happen in the future (i.e., events that we currently know are going to happen), and *SimTalk 2.0* can be used to add events to that list using &object.methCall(…). See the "Did you know?" on the next page for a further elaboration of the event list. For now, we let patients proceed to the waiting room 30 minutes after arriving at the front desk. The actual appointment making algorithm is added to the model in a later task.

Task: &object.methCall(…)

1. Remove the *Connector* between *WaitingAtHome* and *WaitingRoom*.
2. Create a *Method MakeAppointment*, set it as the Entrance control in *FrontDesk* and program it as follows:

```
-- MakeAppointment
-- Purpose: Let a patient arrive after 30 minutes
-- Called by: FrontDesk (Entrance)
&OnAppointment.methCall(1800, @)
```

3. Create a *Method OnAppointment* and program it as follows:

```
-- OnAppointment
-- Purpose: Move a patient to the waiting room
-- Called by: MakeAppointment (MethCall)
param patient: object

patient.move(WaitingRoom)
```

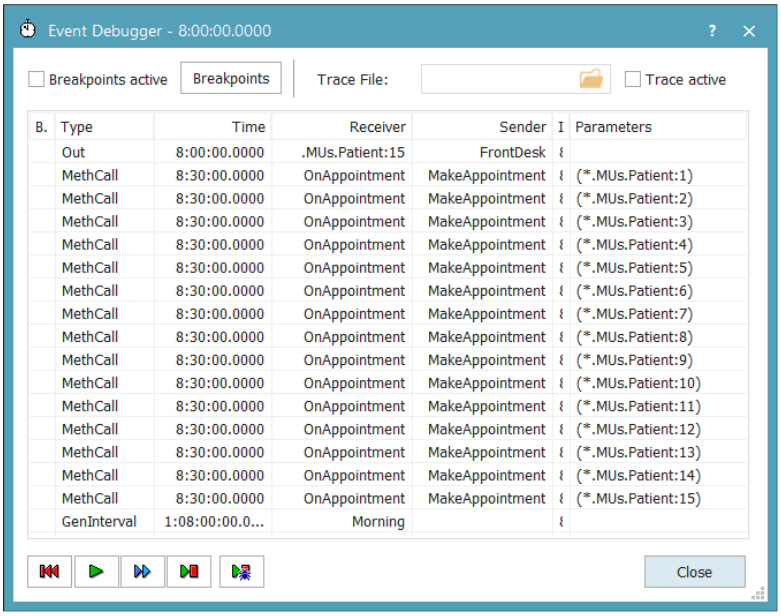
4. Run your model and verify that patients are forwarded to the *WaitingRoom* at 8:30:00.

We have used &object.methCall(callAt, [arguments, ...]) to schedule the movement of patients to the waiting room. In fact, this code adds a future event to the internal Plant Simulation event list. In this code, object is the *Method* that will be called in the future. The moment of calling is the current simulation time plus the time defined in callAt. Optionally, you can define an arbitrary number of arguments that will be passed to the *Method* when it is called. In this case, we pass an instance of the *Patient* object.

Did you know? Browsing through the event list

The event list is a central concept in discrete event simulation. As mentioned before, this is a list of events that are scheduled to occur in the future. New events are added as soon as the exact time of occurrence becomes known. For instance, when a *MU* enters a *SingleProc*, the *SingleProc* will schedule an event in the event list stating that the part must exit the *SingleProc* at the current time plus the processing time. Our model schedules a *MethCall* (short for *Method Call*) for each appointment, stating that *OnAppointment* must be called at $current\ time + 1800\ seconds$, with the current *MU* as an argument. Once scheduled, you can see these events in the event list:

1. Open the *EventController* and set the simulation speed to about 50%.
2. Run the model for a few seconds.
3. Open the *Event Debugger*  (the event list).
4. You now see a list of the events that are scheduled:

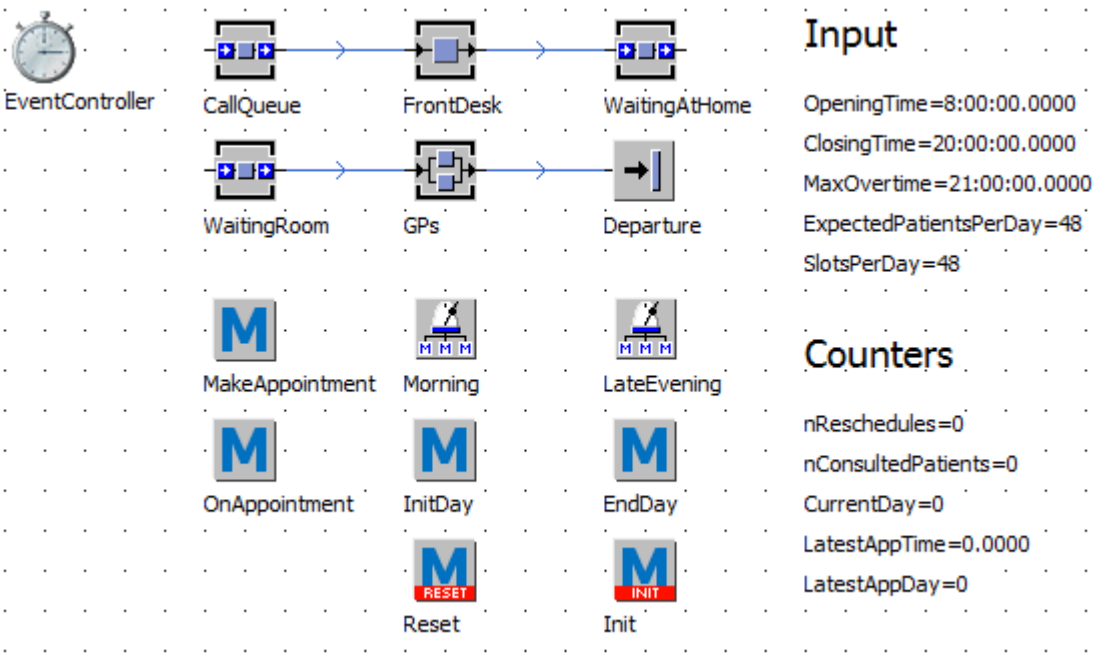


5.4 Adding Counters and Input Parameters

Counters are an easy way to keep track of what has happened in the model. Setting them up is easy: just determine which counters are needed, what their initial value is, when they must be incremented, and when they must be reset. Input variables are variables that you commonly change when experimenting with the model. They will also prove to be useful when we add the *ExperimentManager* later on.

Task: Counting down the days

1. Expand the model to reflect the screenshot below. Note that the *Connector* between *WaitingAtHome* and *WaitingRoom* has been removed.



2. Configure *LateEvening* to call *EndDay* every day at the time defined in the *Variable MaxOvertime*.
3. Program *EndDay* such that it increments *CurrentDay*.

```
-- EndDay
-- Purpose: Increment the day counter
-- Called by: LateEvening

CurrentDay += 1
```

4. Program *Reset* and *Init* such that they reset and initialize the counters, and delete all *MUs* from the model.

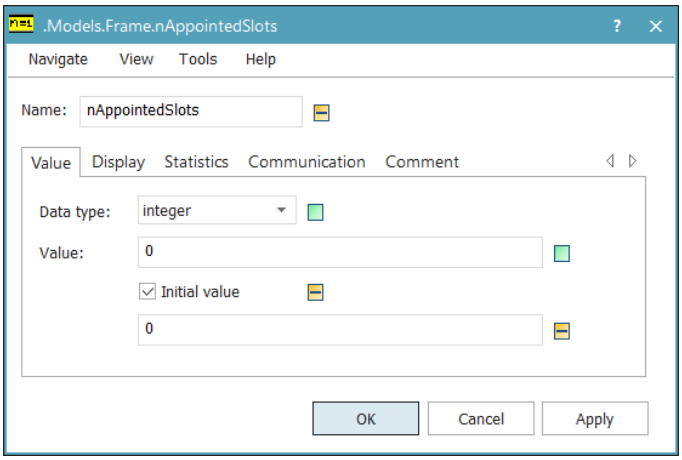
```
-- Reset
-- Purpose:   Reset counters MUs
-- Called by: EventController (Reset)

nConsultedPatients          := 0
nReschedules                := 0
CurrentDay                  := 0
LatestAppTime               := 0
LatestAppDay                := 0

deleteMovables
```

```
-- Init
-- Purpose:   Initialise counters
-- Called by: EventController (Init)

LatestAppTime       := OpeningTime
LatestAppDay        := -1
```

Not all counters are being updated yet, but they will be later on when we implement a more sophisticated appointment algorithm and patient tracking procedure.

Did you know?

As you might have noticed, Plant Simulation has a built-in feature for setting the initial value of a *Variable*, which would make the code in *Reset* redundant. However, the advantages of using *Reset* to reset counters are that all initial values are collected in one place, and all counters reset as soon as the user clicks the reset button in the *EventController* instead of when the model initialises. The former may be more intuitive.



5.5 Appointments

It is now time to implement the most important part of this model: the appointment algorithm. The logic of the algorithm we implement is as follows:

1. Calculate the time difference between appointment times as the time available for appointments on a day divided by the number of appointment slots per day.
2. If the previously scheduled patient had an appointment time that has passed (e.g., yesterday), then schedule the new appointment at the opening time of the current day.
3. Otherwise, calculate the new appointment time as the appointment time of the previously scheduled patient plus the time difference between appointments.
4. If the new appointment time is at or after the closing time, then schedule it at the opening time of the next day.

Task: Setting appointments

1. Open *MakeAppointment* and implement the appointment algorithm (see also code in the Appendix) and try to understand it. Note that you also need to create custom attributes for the patients, and that in the remainder of this chapter you might need to add more attributes without this tutorial explicitly pointing this out.

```
-- MakeAppointment
-- Purpose:   Determine the next available appointment slot and assign it to a patient.
-- Called by: FrontDesk (Entrance)

-- Calculate the appointment time for the patient.
if LatestAppDay < CurrentDay
    LatestAppTime := OpeningTime
    LatestAppDay  := CurrentDay
else
    LatestAppTime += (ClosingTime - OpeningTime) / SlotsPerDay
end

-- If the appointment time is after closing time, then appoint the patient
-- to the subsequent day.
if LatestAppTime >= ClosingTime
    LatestAppTime := OpeningTime
    LatestAppDay  += 1
end

-- Set the patient information.
@.appointmentTime := latestAppDay * 86400 + latestAppTime
@.callTime        := EventController.SimTime
@.waitingAtHomeTime.increment(@.appointmentTime - EventController.SimTime)

-- Let the patient arrive at the appointed time.
&OnAppointment.methCall(@.appointmentTime - EventController.SimTime, @)
```

2. Run the model at a slow speed, to see that the patients arrive in equally spaced intervals between 8:00 and 20:00.

MakeAppointment increments the *waitingAtHomeTime* attribute instead of just setting it. The same line could be written as "@.waitingAtHomeTime := @.waitingAtHomeTime + @.appointmentTime - EventController.SimTime" or shorter as "@.waitingAtHomeTime += @.appointmentTime - EventController.SimTime". The waiting time is incremented because it is possible that a patient has to wait at home more than once due to rescheduling.

5.6 Tracking Patients and Calculating Performance indicators

The model specification calls for two types of waiting times to be calculated: the waiting time spent at home, and the waiting time spent in the waiting room. *MakeAppointment* already calculates the former, since it has all the information needed to do so. We will now implement a *Method* that calculates the other type of waiting time. We also implement several performance indicators and update them every time a patient starts a consultation or a leaves the system.

Task: Tracking patients

1. Create a *Method TrackAppointment* and set it as an *Entrance Control* in *GPs* and *Departure*.
2. Use an if … elseif statement to execute different blocks of code depending on whether *GPs* or *Departure* calls the *Method*.

```
-- TrackAppointment
-- Purpose: Update patient statistics, recalculate performance indicators
-- Called by: GPs (Entrance), Departure (Entrance)
var n: integer

if ? = GPs
    -- Update patient statistics

elseif ? = Departure
    -- Recalculate performance indicators

end
```

3. If *GPs* calls *TrackAppointment*, then determine the arrival time at the GP and the waiting time spent in the waiting room:

```
@.arrivalAtGP := EventController.SimTime
@.waitingRoomTime.increment(EventController.SimTime - @.appointmentTime)
```

4. If *Departure* calls *TrackAppointment*, then recalculate a number of performance indicators.

```
-- Increment the consulted patients counter.
nConsultedPatients += 1

-- Recalculate performance indicators.
n                       := nConsultedPatients
AvgWaitingAtHomeTime    := (n-1)/n * AvgWaitingAtHomeTime + 1/n * @.waitingAtHomeTime
AvgWaitingRoomTime      := (n-1)/n * AvgWaitingRoomTime + 1/n * @.waitingRoomTime
AvgPerceivedWaitingTime := 0.05 * AvgWaitingAtHomeTime + AvgWaitingRoomTime
RescheduleRate          := (n-1)/n * RescheduleRate + 1/n * @.reschedules
```

5. Add the *Variables AvgWaitingAtHomeTime, AvgWaitingRoomTime, AvgPerceivedWaitingTime*, and *RescheduleRate* to the *RootFrame*. Also update *Reset* such that these *Variables* are reset to zero once the model resets.

Notice the formula used to update *AvgWaitingAtHomeTime*, *AvgWaitingRoomTime*, and *RescheduleRate*. Let X_i be the observation of the waiting at home time, waiting room time, or the number of reschedules for the i^{th} patient, and let n be the number of patients used to calculate the performance indicator. Then the formula used in *MakeAppointment* is derived as follows:

$$\overline{X} = \frac{\sum_{i=1}^{n} X_i}{n} = \frac{\sum_{i=1}^{n-1} X_i}{n} + \frac{X_n}{n} = \frac{n-1}{n} \cdot \overline{X}_{1,\ldots,n-1} + \frac{1}{n} \cdot X_n \qquad \text{where } \overline{X}_0 = 0$$

5.7 Rescheduling Patients

The model almost complies with the specifications. The only missing feature is that of rescheduling patients that are still in the waiting room when the clock hits 21:00. Fortunately, we already have the *Method EndDay* that is called at exactly 21:00. We expand this *Method* such that it takes all patients in the waiting room, calculates their waiting time, increments their reschedules counter, and moves them to *CallQueue*, such that they get a new appointment just like every *new* patient.

Task: Back to square one

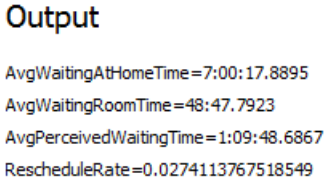
1. Expand the *Method EndDay* (see also code in the Appendix) such that it takes care of patients that are in the waiting room after 21:00, and try to understand it:

```
-- EndDay
-- Purpose: Increment the day counter and move patient to FrontDesk
-- Called by: LateEvening
var i: integer
var patient: object

-- Increment the day counter
CurrentDay += 1

-- Move any patient that is left in WaitingRoom to FrontDesk,
-- such that it will reschedule.
for i := 1 to WaitingRoom.NumMU
    patient := WaitingRoom.MU(1)
    patient.waitingRoomTime.increment(EventController.SimTime - patient.appointmentTime)
    patient.reschedules.increment
    patient.move(CallQueue)

    nReschedules += 1
next
```

2. Test your model and verify that all output variables are now being updated as the model runs.

Note: Integer or real?

If the global *Variable RescheduleRate* does not get updated when you run the model, make sure that its data type is set to *real* and not to *integer*. The variables under *Output* should have the same number format as the variables shown in the figure below.

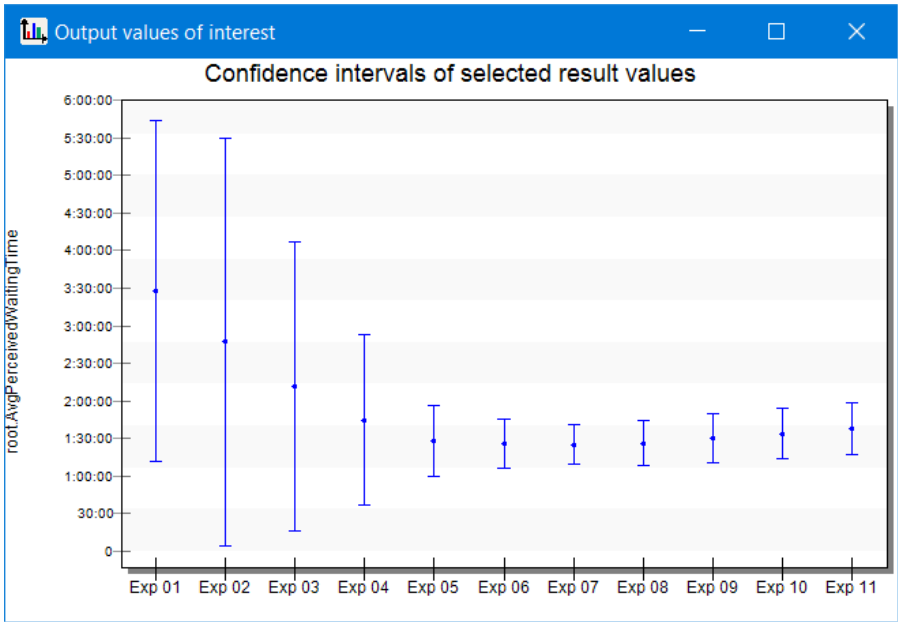


5.8 Experimenting

Now that the model works according to the specifications, we can perform some experiments to determine the optimal number of appointment slots on a day. For this purpose, Plant Simulation comes with the *ExperimentManager*. Using this object involves setting one or more input variables, defining experiments (input variables with their settings), choosing a number of replications, and defining one or more output variables.

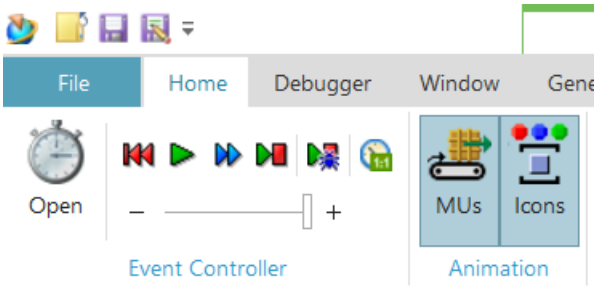
Task: Setting up the ExperimentManager

1. Include the *ExperimentManager* object  in the *RootFrame* and open it.
2. Check the *Use input value*s box and click *Define Input Variables*. Drag the *SlotsPerDay Variable* into the first cell of the table that opens (you can also type the location of this variable into the cell). Close the table.
3. Click *Define Experiments*. Expand the table to the number of experiments/configurations that you want to study by right-clicking a cell and selecting *Append Row* (or just press enter). Type *true* in all cells of the first column. Type different values for the number of appointment slots per day in the second column (the experiments). Close the table.
4. Click *Define Output Values*. Expand the table to 4 rows and drag the variables *AvgPerceivedWaitingTime, RescheduleRate, AvgWaitingAtHomeTime*, and *AvgWaitingRoomTime* into the cells of the first column. Close the table.
5. Set *Observations per experiment* to 3 (the number of replications).
6. Open the *Evaluation* tab and set the *Intervals* to *Confidence*.
7. Click *Reset* and then *Start* to let the simulation run.
8. In the report that appears afterward, navigate to *Statistical Evaluations – Statistics of output values* to get a graph for each output variable. Or, in the *ExperimentManager*, under tab Evaluation, select show chart (more detailed results can also be found under this evaluation tab of the *ExperimentManager*). You should see something similar to the figure below.



Note: Disable/enable animation

- The *ExperimentManager* disables *MU* animation and Icon animation while running by default, in order to increase performance.
- To disable or enable the animation, use *Home* → *MUs* and *Home* → *Icons*.

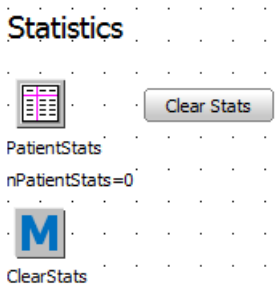


5.9 Creation of a Dataset

The *ExperimentManager* collects a large number of statistics by default, including the averages of output variables, their standard deviations, and the p-values to determine whether experiment outputs are significantly different from each other. Detailed results (including the results per replication) can be found under the Evaluation tab of the *ExperimentManager*. However, it is often desirable to continue the analysis in a separate program. Therefore, we are going to create a *Method* that gathers all performance data in a comprehensive dataset that can be exported to an external program, such as Excel. Since we do not want the data in the table to be deleted each time the model resets (which happens between replications), we use a button to clear the table at our own discretion instead of the *Method Reset*.

Task: Gathering data

1. Use a *Comment*, *TableFile, Variable, Button*, and a *Method* to create the following set of objects.



2. Right-click the button and select *open* to access the button's configuration. Set the *Name* to "ClearStatsButton", the *Label* to "Clear Stats", and the *Control* to "ClearStats".
3. Program the *Method ClearStats* as follows.

```
-- ClearStats
-- Purpose: Clear the PatientStats table
-- Called by: ClearStatsButton

PatientStats.delete
nPatientStats := 0
```

4. Open the *TableFile PatientStats*, activate the column index, and format the columns as follows.

	integer 1	integer 2	integer 3	time 4	time 5	integer 6
string	experiment	run	patientNumber	waitingAtHomeTime	waitingRoomTime	reschedules

5. Open *TrackAppointment* and insert the following code such that statistics are logged when a patients leaves the model.

```
-- Log statistics.
nPatientStats += 1
PatientStats["experiment", nPatientStats]        := ExperimentManager.currExpNo
PatientStats["run", nPatientStats]               := ExperimentManager.currRunNo
PatientStats["patientNumber", nPatientStats]     := nConsultedPatients
PatientStats["waitingAtHomeTime", nPatientStats] := @.waitingAtHomeTime
PatientStats["waitingRoomTime", nPatientStats]   := @.waitingRoomTime
PatientStats["reschedules", nPatientStats]       := @.reschedules
```

6. Reset and start the *ExperimentManager*.
7. Open the *PatientStats* table to verify that information on each patient has been stored, preceded by the experiment and run (=replication) number.
8. Click the *ClearStats* button to remove the data from the table.

The simulation model now works according to the specifications, experiments can be defined and executed, and performance data is properly stored. However, before running the experiments, we have to carefully think about a possible warm-up period, run length, and number of replications. We determine these aspects using some Key Performance Indicator (KPI), in this case the perceived waiting time. For this purpose, it is convenient to introduce one additional performance table where we store the perceived waiting time per patient and per replication. After determining the warm-up period, run length, and number of replications, we can remove this additional performance table and remove or comment the related code (e.g., putting the code between /* and */).

Task: Gathering KPI data per replication

1. Add the following code to *TrackAppointment*.

```
-- Log perceived waiting time per replication
nReplicationPatientStats += 1
m := ExperimentManager.NumRuns * (ExperimentManager.currExpNo - 1) +
        ExperimentManager.currRunNo
ReplicationPatientStats[m, nReplicationPatientStats] :=
        0.05 * @.waitingAtHomeTime + @.waitingRoomTime
```

2. Create the corresponding table *ReplicationPatientStats*, global variable *nReplicationPatientStats*, and local variable *m* (with proper formatting / data types).
3. Reset *nReplicationPatientStats* to zero after each run by adding a line of code to *Reset* (in contrast with the global variable *nPatientStats*, do we need to reset *nReplicationPatientStats* after each run, study the code to find the difference between these two counters).
4. Modify the *Method ClearStats* in such a way that the table *ReplicationPatientStats* is cleared in a similar manner as *PatientStats*.

5.10 Assignment A3: Warm-up Period, Run Length, and Number of Replications

When running the model after implementing the performance statistics in Section 5.6, you may have noticed that it takes some time for the output variables to balance out. The reason is that it takes the model some time to "warm-up". Consequently, the first few days of simulation results might not be representative for the real system performance. Furthermore, each run should have a reasonable run length.

Assignment A3.1: Select a proper warm-up period and run length denoted in simulation days. Provide argumentation for your choice of warm-up period and run length, and the configuration(s), i.e., value(s) of *SlotsPerDay*, you use to determine these settings.

After determining the warm-up period, you need to make sure that statistics are only collected after the warm-up period. One way of doing this is to set the warm-up period in the EventController, under the tab Settings, at Statistics (below the setting for end time) and to add the following condition to TrackAppointment:

```
if EventController.SimTime > EventController.StartStat
    ...
end
```

Note that, to achieve the desired run length, you need to increase the end time of the simulation with the length of the warm-up period.

The first few runs (replications) of Section 5.8 showed interesting results, but the relatively large confidence intervals suggest that the results might not be statically significant (confidence intervals of various experiments overlap). Increasing the number of replications can reduce the width of these confidence intervals, but more replications do imply longer calculation times.

Assignment A3.2: Select a proper number of replications per experiment using an appropriate statistical analysis in Excel or a similar spreadsheet program. Choose the number of replications corresponding to a relative error of at most 5% ($\gamma = 0.05$) and a 95% confidence level ($\alpha = 0.05$). Again, it is up to you what value(s) for *SlotsPerDay* to use.

Assignment A3.3: Provide advice on the number of appointment slots to use per day. To support your choice, provide insight into the performance of this setting (giving confidence intervals on the average perceived waiting time and reschedule rate). In addition, provide insight into the spread of the perceived waiting times.

6 Demonstration of More Advanced Concepts

In the previous chapters of this tutorial, you have learned about the basics of Plant Simulation; however, the possibilities of Plant Simulation go way further than previously shown. This chapter will elaborate on some of these more advanced concepts and serves as a preview of Part B of this tutorial. First, it is shown how you can gain access to the demos and examples available in Plant Simulation. The other sections will elaborate on advanced concepts that are frequently used for simulation purposes.

In Section 6.2, the concept of *Frames* will be treated. In the preceding chapters, only the *RootFrame* has been used, but you may also create *Frames* in the library on which you can add a number of other objects. These *Frames* can be put into the *RootFrame* or in another *Frame*, which creates a dynamical hierarchy of models. This enables you to construct simulation models that are well structured and reflect the natural hierarchy of systems to be simulated. In addition, this enables you to break down complex tasks into manageable chunks. The concept of *Frames* will be treated in depth in Chapter 7.

Next, in Section 6.3, the concept of line objects will be treated. The transportation of patients, products or vehicles may be displayed by using *SingleProcs* as the previous chapters have shown. However, for these situations Plant Simulation offers an alternative, namely objects such as a *Track*. The objects *Lines* and *Tracks* will be treated in depth in Chapter 8.

The final advanced concept that will be discussed is the so-called *Worker* (Section 6.4). You might imagine that there are situations where we would like to simulate a situation in which an employee works at multiple workstations and has to move on paths between these workstations. This concept involves the objects *WorkerPool* and *Worker*. Also these objects will be treated in more depth in Chapter 8.

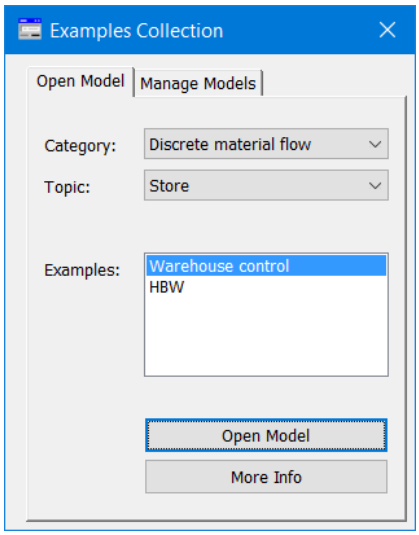
6.1 Examples and Demos in Plant Simulation

As stated before, much more is possible with Plant Simulation than only the topics treated in this tutorial so far. In order to get an idea of some of the other possibilities in Plant Simulation, you might want to take a look at the demos and examples present in Plant Simulation. If you start Plant Simulation, a dialog window, named *Start Page,* will open. Select the option *Example Models* in this dialog window. This will open the Plant Simulation *Step-by-Step Help* where you can find various example models.

Example Models

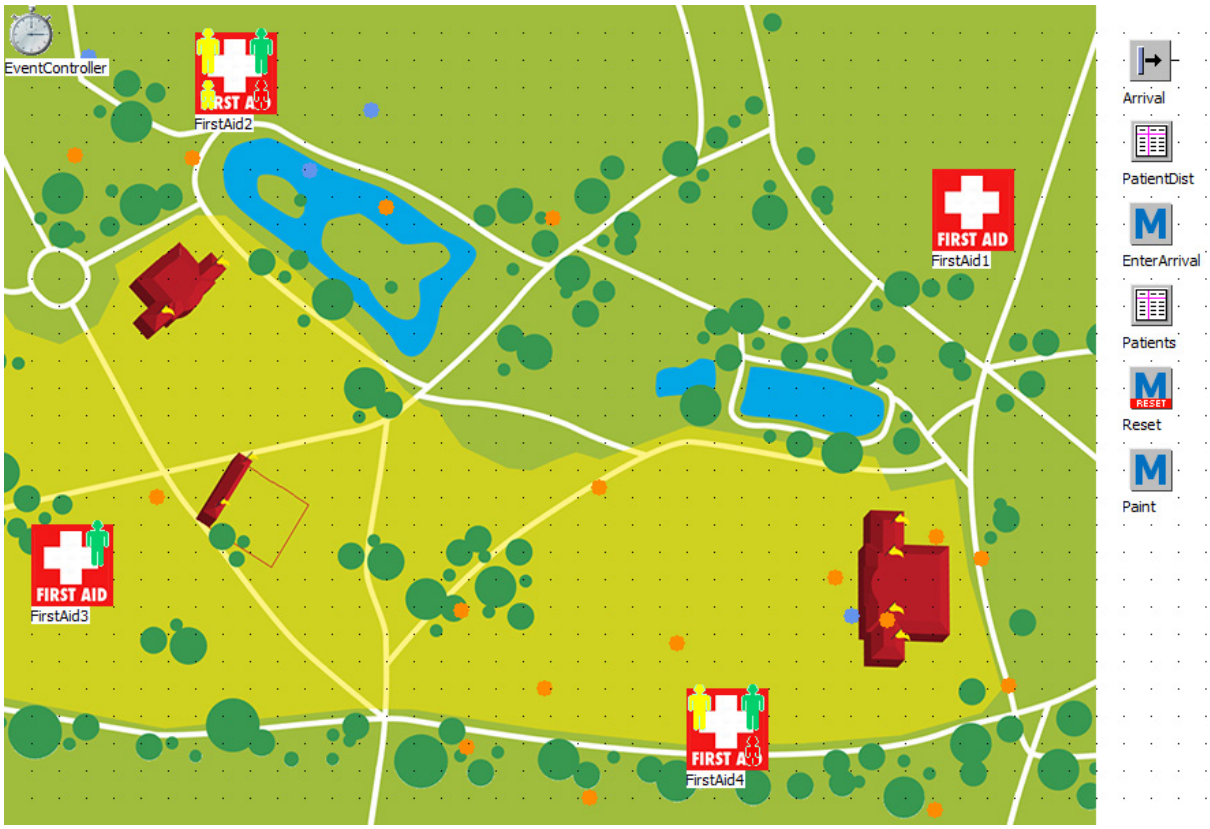
	**Small Parts Production**	Shows a simple example of a small parts production and how to use the Experiment Manager.
	**Nanobox Assembly**	The model is based on the Nanobox assembly line of Siemens Karlsruhe. A Nanobox is a small computer especially designed for the needs of production facilities.
	**Turbo-Charger Assembly**	Shows how to increase productivity to meet increasing customer requirements.
	**Body in White**	Shows how to optimize buffer capacities and how to compare same model with different parameters.
	**Rear Axle Assembly**	Shows how to optimize the layout and how to define strategies and capacities.
	**Due Date Optimization**	Shows how to optimize a simulation model with genetic algorithms. (Requires a Plant Simulation Professional license.)
	**3D Carbody**	Provides a concise demo model of the 3D visualization.
	**Small Examples**	Provides sample models demonstrating how to solve various modeling tasks.

If you click on the *Small Examples*, you can select a category and a related topic for which you then can choose a model you would like to view. These models might give you an idea of the possibilities of Plant Simulation and might even help you modelling certain processes.



6.2 Frames

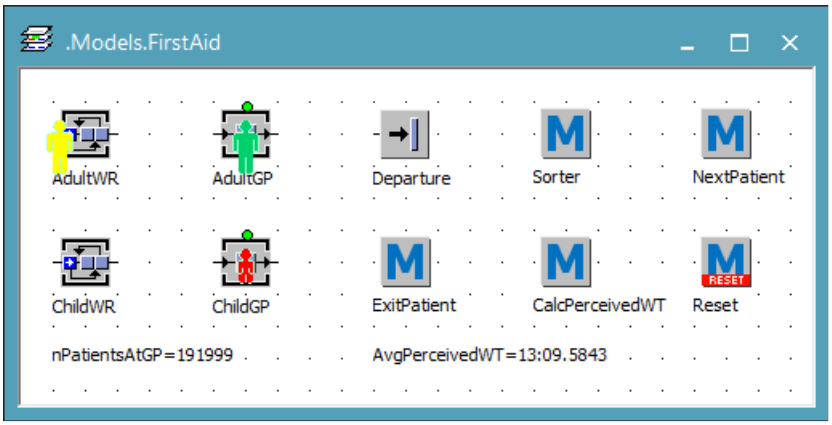
The models, you have made in Chapter 3, 4 and 5, were all made on a *Frame* called the *RootFrame* (see Chapter 2). However, if your model grows in complexity and you have a lot of different processes to model, or repetition of similar processes, you might want to consider using multiple *Frames*. For example, if you have a number of similar processes, you might create a *Frame* of this process and add multiple instances of this *Frame* on your *RootFrame*. This allows you to structure your models, which makes them easier to build and maintain through *modular design* and *object-oriented programming*.



If you build a *Frame* in the *Class Library*, you can use it multiple times. In order to create a new *Frame*, you need to right-click on the folder *Models* in your *Class Library* and select *New* and then *Frame*.

We will illustrate the advantage of using *Frames* with an example. Suppose we have a festival terrain (see previous page) for which we need to setup first aid posts. As you can imagine, we need to spread to first aid posts efficiently in order to get a good coverage over the whole festival area. Due to the fact that we will need multiple first aid posts, it would then be wise to model one single first aid post in a *Frame* and add it multiple times to our festival terrain.

We have used a slightly adjusted version of the model that has been built in Chapter 3. The GP from Chapter 3 has been remodelled to a first aid post. The *Arrival* object has been removed from the model and added to the *RootFrame*. Furthermore, we have added a *Method ExitPatient*, which will remove the patient from the *TableFile Patients*. This adjusted version of the model from Chapter 3 is put into a *Frame* and will be reused four times for our new model. The *Frame* of the first aid post consist of the following model:



On the *RootFrame*, there is an image of the festival terrain where the four first aid posts are located. All these four posts are derived from the same *Frame* and are therefore subjective to *Inheritance* (see Chapter 2). Also a *Source* object (called *Arrival*) is added to the *RootFrame*. The distribution of *Adults* and *Childs* is defined in the *TableFile PatientDist*, and each new patient is characterised (using *User-defined Attributes*) by a location on the festival terrain (x- and y-coordinate) and an urgency via the *EnterArrival Method*. In the same *Method* it is determined, which first aid post is the closest to the patient's location. The code of *EnterArrival* is shown below.

```
-- Determine patient characteristics
@.Xcoord:=z_uniform(1,0,800)
@.Ycoord:=z_uniform(2,0,600)
@.urgency:=floor(z_uniform(2,1,4))

-- Determine nearest first aid station
TheObject:=str_to_obj("FirstAid"+sprint(1))
TheDistance:=sqrt(pow(@.Xcoord-TheObject.Xpos,2)+pow(@.Ycoord-TheObject.Ypos,2))
Winner:=TheObject
MinDist:=TheDistance
for i:=2 to 4
    TheObject:=str_to_obj("FirstAid"+sprint(i))
    TheDistance:=sqrt(pow(@.Xcoord-TheObject.Xpos,2)+pow(@.Ycoord-TheObject.Ypos,2))
    if TheDistance<MinDist
        MinDist:=TheDistance
        Winner:=TheObject
    end
next
```

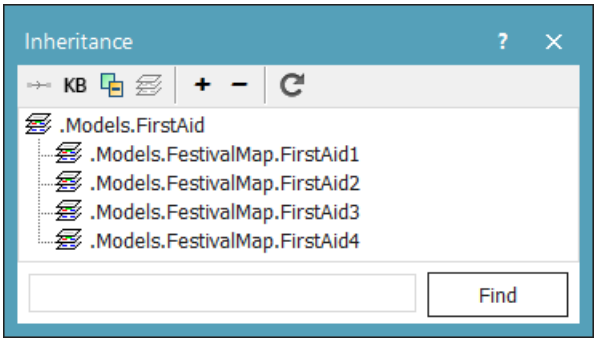
Each new patient is send to the nearest first aid post. Via a *Method Paint*, we draw, for each patient currently present at a first aid post, a dot on the festival map to indicate the original location of the patient. The dot will be removed if the patient is treated. To be more precise, each time something changes (arrival or departure of a patient), we remove all dots and then paint the dots for all patients currently listed in the *TableFile Patients*. The code of *Paint* is shown below.

```
var i,NodeSize,TheX,TheY,NodeColor: integer

root.eraseLayer(1)
NodeSize:=11
for i:=1 to Patients.yDim
    if Patients["Child",i]
        NodeColor:=15570276
    else
        NodeColor:=36095
    end
    TheX:=Patients["Xcoord",i]
    TheY:=Patients["Ycoord",i]
    root.drawEllipse(1,TheX-((NodeSize-1)/2),TheY-((NodeSize-1)/2),NodeSize,NodeSize,NodeColor,-1)
next
```

Note that to refer, within a *Method*, to an object that is not located on the *Frame* that contains the *Method*, you need to provide a path working from the *RootFrame*. In our example, the *Method ExitPatient* in the *Frame* of our first aid post refers to objects on the *RootFrame*, namely the *TableFile Patients* and the *Method Paint*. If the patient leaves the model, it will be removed from the table *Patients*, because it is not present in the model anymore and therefore does not need to be painted on the festival map. We refer to the *TableFile Patients* located on the *RootFrame* by "root.Patients", as shown in the following code.

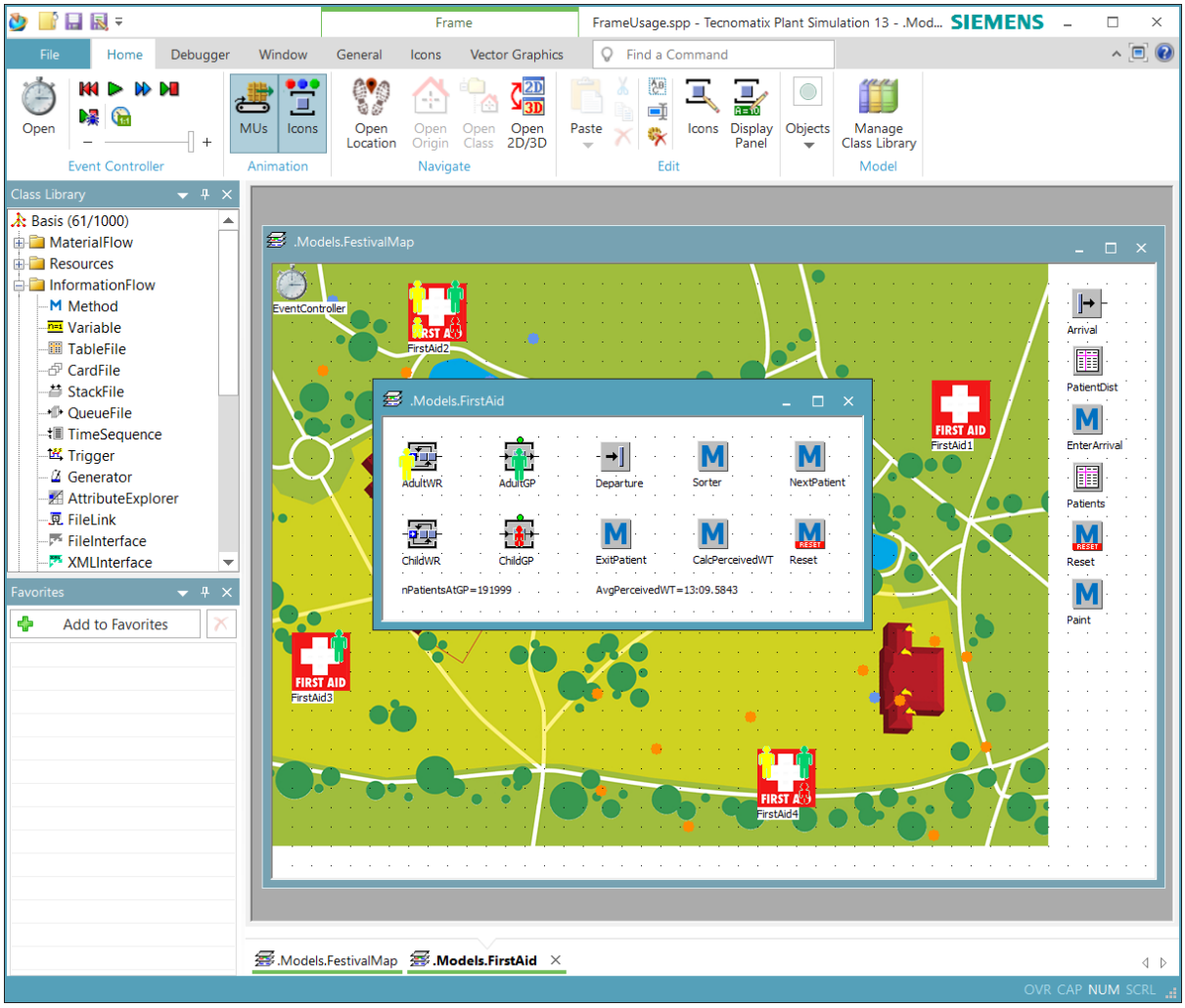
```
root.Patients.cutRow(@)
root.Paint
```

You need to take into account that when you would like to have different processing times for your first aid post, it is necessary to adjust them separately in the specific *Frame* (so, in your model not within the *Class Library*). Inheritance of the *FirstAid* class can be made visible by right-clicking on this class and selecting *Show Inheritance*.



Note that, when using *Frames*, we have to be even more careful with the concept of inheritance (see Chapter 2). Suppose we open one first aid post located on the *RootFrame* of our model; *FirstAid2* in the figure below. The complete path of this object is ".Models.FestivalMap.FirstAid2", which is obviously different from the path ".Models.FirstAid" of the class *FirstAid* within our library (note that within our model, we use the shorthand notation "root.FirstAid2" for the path of *FirstAid2*).

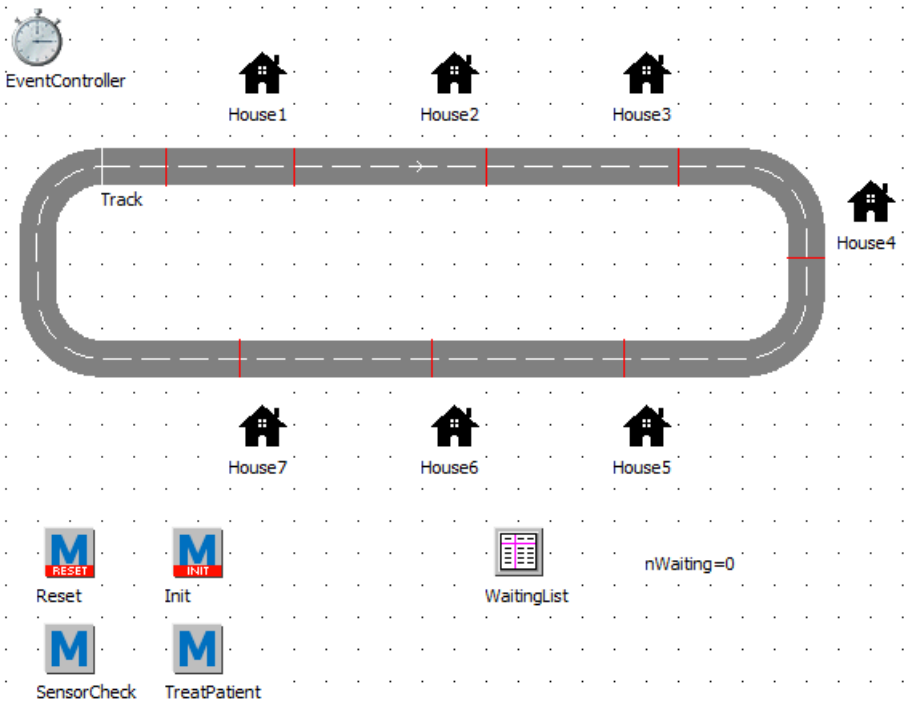
Now, if we want to change, e.g., the code within the *Method Sorter*, such that the changes are applied to all four first aid posts, we need to change the *Sorter* within the library, i.e., at ".Models.FirstAid". To avoid mistakes, Plant Simulation also locks editing of the objects (derived from the library classes), e.g., code within *Methods* is marked light grey, and, unless we turn off inheritance, we cannot edit the code. A more in depth treatment of *Frames* can be found in Chapter 7.



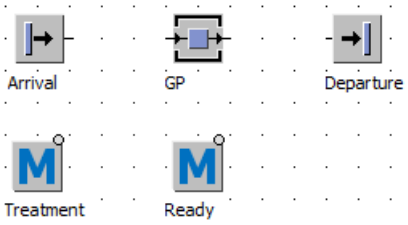
6.3 Line Object

Sometimes the process you intend to model involves the transportation of entities (e.g., patients) and the transportation might involve the necessity of a vehicle. This is a typical process for which the use of the *Track* object and the *Transporter* object might be beneficial.

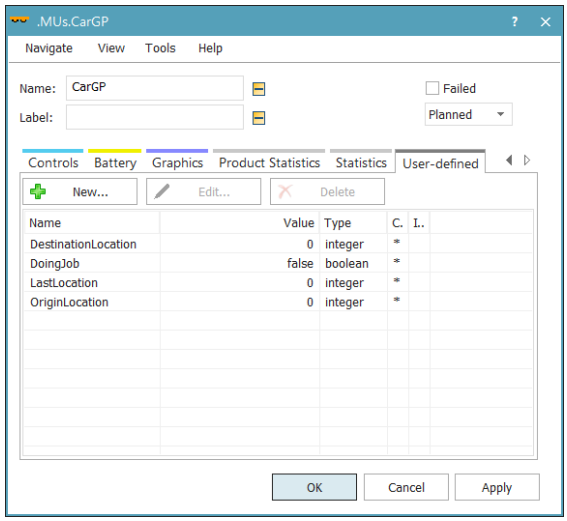
The *Transporter* is the only moving object that is able to use the *Track*. These objects can respectively be found under the folder *MUs* and *MaterialFlow* in your *Class Library*. You can vary various attributes of these objects, such as the length of the track and the speed of the *Transporter*. You can add sensors to your track, which can be used to trigger a certain action when the transporter passes such a sensor. These sensors can be found by opening the track (double-click on the track), going to the tab *Controls*, and selecting *Sensors*. Sensors are defined by a distance from the start of the track. Once a transporter arrives at the sensor (either by front or rear depending on how you defined it at the sensor), a *Method* will be triggered. We illustrate the usage of the *Transporter* and the *Track* with an example of a GP that needs to visit patients at their homes.



For illustrative purposes, there are seven houses, which all have sensors to indicate the location of a house near the track. We have one sensor check at the start of the track, which is the position to which the GP will drive when no patient is waiting for help of the GP. The ambulance (object *Transporter)* represents the GP, who visits the houses. Note that we have used *Frames* (see previous section) for modelling the individual houses, see figure below for the contents of such a *Frame*.



The *Transporter* has been given various *User-defined Attributes* to ensure that it visits the patients and houses in the correct order. The attributes we defined can be found in the following figure:



If a patient needs to be visited, the *Method Treatment* in the *Frame* of a *House* will add the patient's location to the *TableFile WaitingList* on the *RootFrame*. This *Method* is called when a patient enters the *Arrival* object. It will then check whether the GP is busy or not. If the GP is available, it will change the destination location of the GP and indicate that the GP is doing a job now and the GP will start driving. A part of that code can be found below:

```
if ? = .Models.Frame.House1.Arrival and root.Track.MU.DoingJob = False
    root.Track.MU.DestinationLocation := 2
    root.Track.MU.DoingJob := True
    root.Track.MU.Stopped := False
elseif ? = .Models.Frame.House2.Arrival and root.Track.MU.DoingJob = False
    root.Track.MU.DestinationLocation := 3
    root.Track.MU.DoingJob := True
    root.Track.MU.Stopped := False
```

Note that the question mark is needed to indicate from which *Frame* the *Method Treatment* is called. If the GP is busy, the patients that need his assistance will be added to the *TableFile WaitingList*. Now that the GP is driving, the sensors will be used to see whether the GP reached his destination location or not. For this purpose every time a sensor is passed, the *Method SensorCheck* is called.

```
param SensorID : integer, Front : boolean

Track.MU.LastLocation := sensorID -- Save the new location of the transporter

if root.Track.MU.LastLocation /= root.Track.MU.DestinationLocation and root.Track.MU.DoingJob = True
    root.Track.MU.Stopped := False
elseif root.Track.MU.LastLocation = root.Track.MU.DestinationLocation and root.Track.MU.DoingJob = True
    root.Track.MU.Stopped := True
    root.TreatPatient
elseif root.Track.MU.DestinationLocation = 1 and root.Track.MU.LastLocation = root.Track.MU.DestinationLocation
    root.Track.MU.Stopped := True
end
```

If the GP has reached its destination, the *Method TreatPatient* will be called, which moves the patient in the *House* (on the *Source* object *Arrival*) to the *GP* (a *SingleProc*). The *Transporter* will stay at the sensor located at the corresponding *House* until the treatment is finished. If the treatment is finished, the *Method Ready* will be called, which moves the patient to the *Departure* and searches whether there are any other patients waiting for help. If not, the *Transporter* will return to its starting position.

Other Examples

As mentioned in the introduction of this chapter, many example models can be found in Plant Simulation itself. In the table below, a short selection of examples involving the usage of the *Transporter* is given. A more in depth treatment of *Lines*, *Tracks*, and *Transporters* can be found in Chapter 8.

Category	***Topic***	***Examples***
Continuous Material Flow	Tugger Trains	Tugger Trains
Continuous Material Flow	Tugger Trains	Towers of Hanoi
Continuous Material Flow	Tugger Trains	Type-specific unload
Continuous Material Flow	Sensors	Sensors
Continuous Material Flow	Automatic Routing	Automatic Vehicle Routing

6.4 Workers

It is also possible in Plant Simulation to use *Workers*, which can represent employees, who conduct tasks, can be assigned to different work places, and switch between these work places possibly requiring some transportation time. Plant Simulation has standard objects for this purpose, which will be briefly explained here.

Worker: a working person who performs jobs on a *Workplace*.

Workplace: the actual place at the station where the *Worker* performs his job.

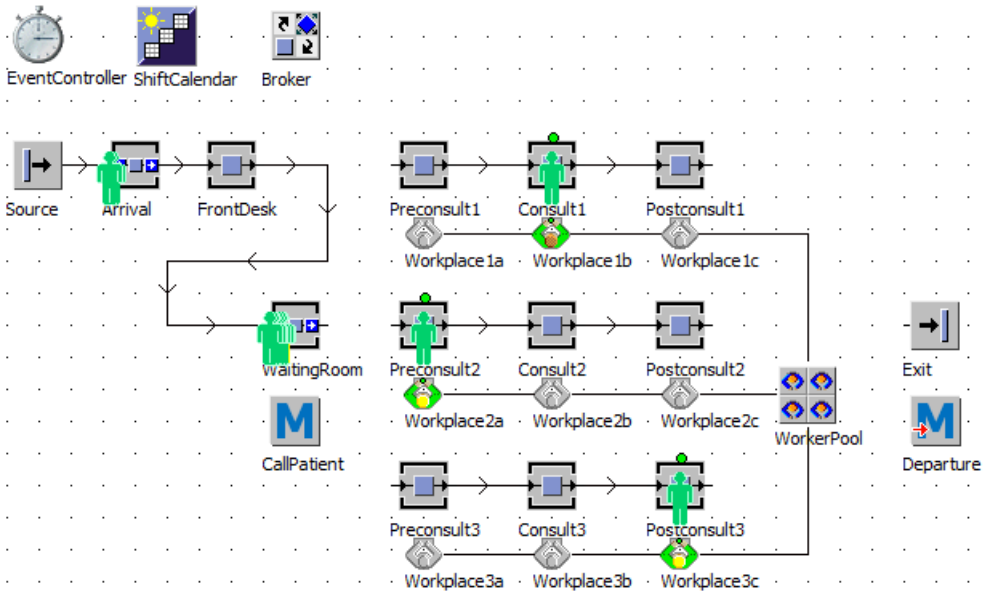
WorkerPool: the *Workers* are created in the object *WorkerPool* and they stay in this object when they are no jobs to perform.

Footpath: on the object *Footpath*, the workers move between the *Workplace* and the *WorkerPool*.

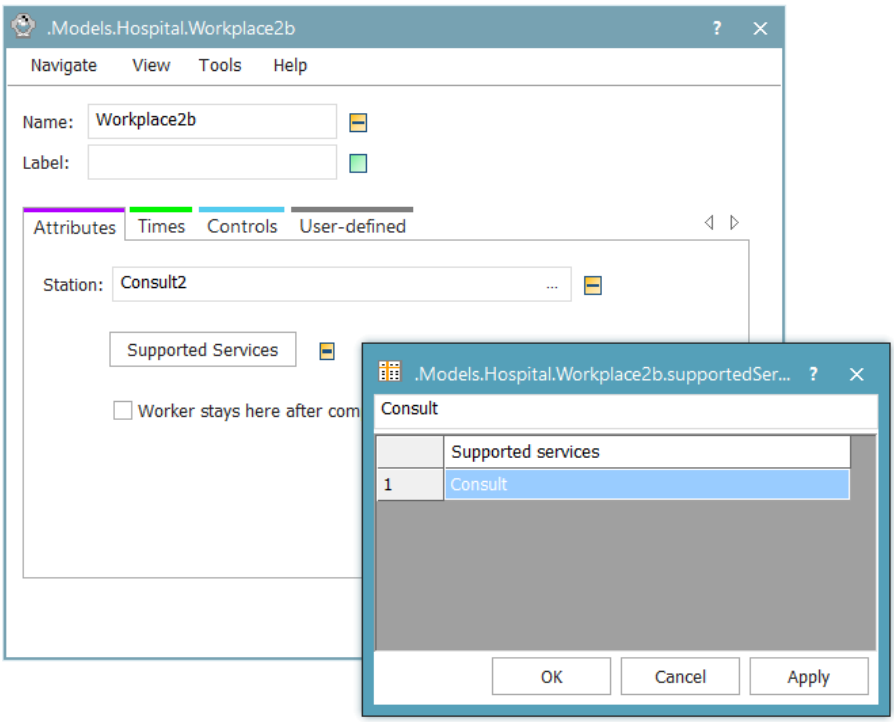
Broker: the object *Broker* is the intermediator between the objects that demand a service and the objects that supply that service.

As an example, consider an outpatient clinic where patients arrive according to a Poisson process, with mean interarrival time of one minute. Each patient requires a sequence of three stages: pre-consult, consult, and post-consult. A patient receives all stages in the same room; however, the consult is given by a physician, and the other two stages by a physician assistant. The clinic employs one physician and two physician assistant. There are three rooms available and opening hours are from 8:00 till 15:30.

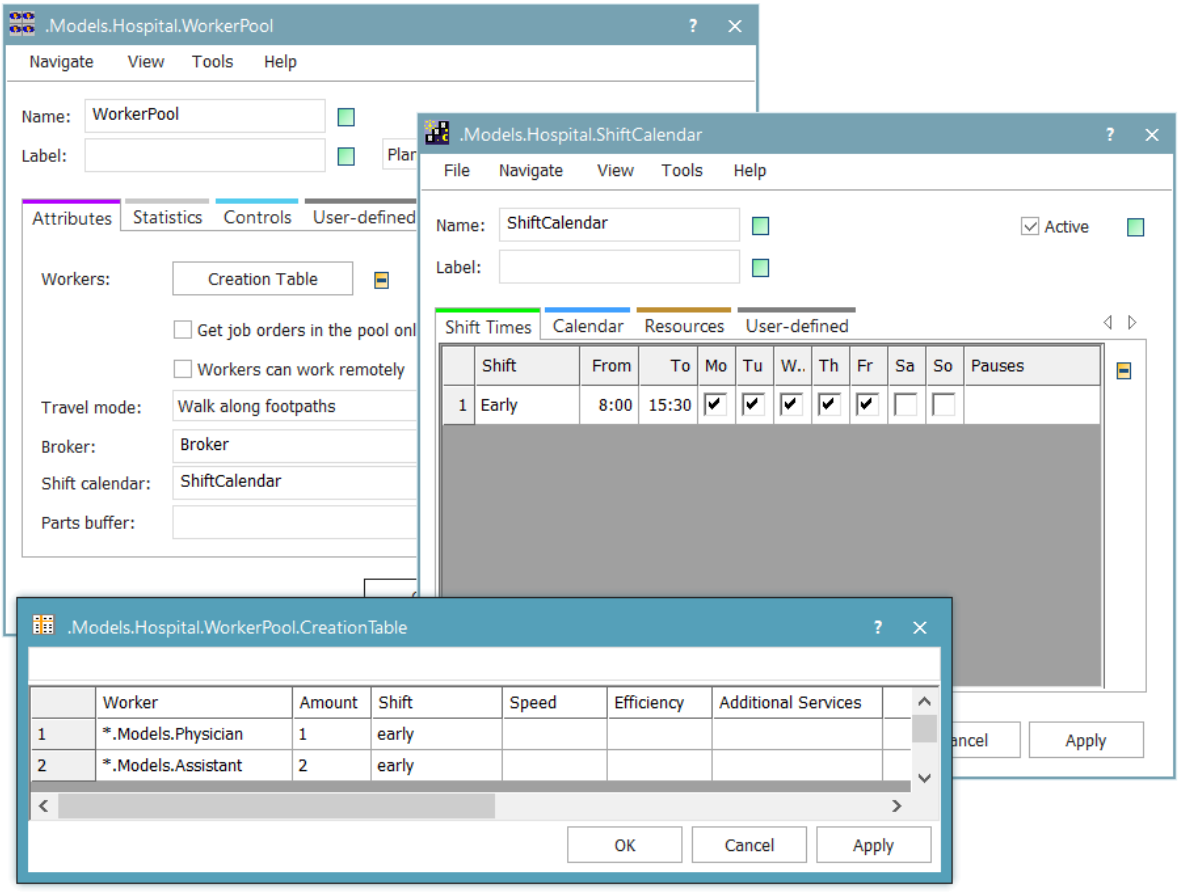
The challenge of modelling this system is that the employees (physician and physician assistants) need to switch between rooms, so basically treating patients in parallel. This can be modelled by using the workers as shown in the figure below.



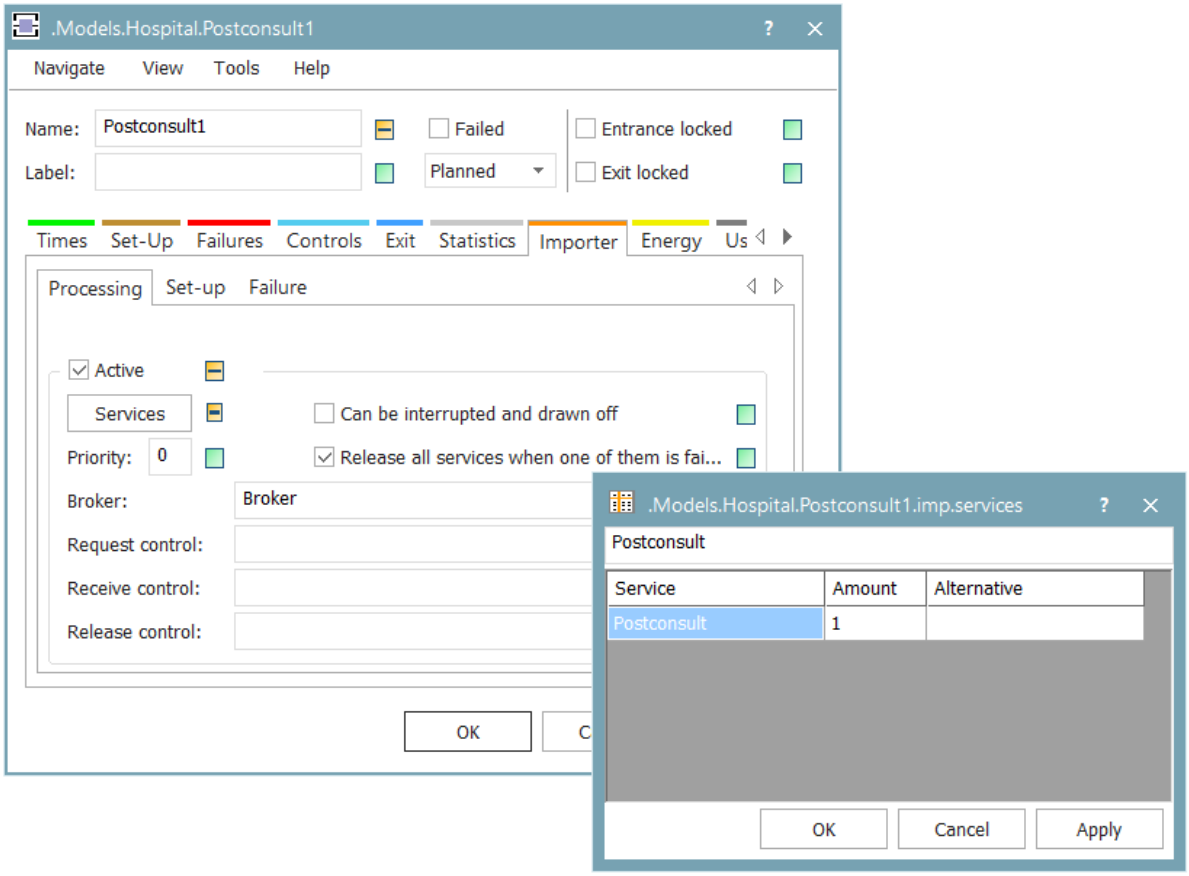
Each workplace is attached to the corresponding stage-room (room and stage of treatment combination) by filling in the *Station* attribute; in the example below *Workplace2b* is connected to *Consult2*. In addition, each *WorkPlace* requires a certain service; in the example below, *Workplace2b* requires the service *Consult*, which can only be provided by the physician.



A *WorkerPool* is added to generate the required staff. The *WorkerPool* and the *Source* are connected to the *ShiftCalendar*. For the *Source*, the *ShiftCalendar* needs to be specified under tab *Controls*. Under the tab *Attributes*, you also need to specify the *Operating mode*, stating whether patients should also arrive outside the shifts (with Blocking checked, patients also arrive outside the shifts, but will be kept on the Source until the start of the shift).



In the figure above, it is also visible that the *WorkerPool* is connected to the *Broker*. In a similar manner, you also need to connect the *SingleProcs* to the Broker, as shown in the figure below. No adjustments have been made to the *Broker* itself.



Other Examples

You can find various other examples of the use of workers in Plant Simulation itself. A selection of examples is given in the table below. A more in depth treatment of *Workers* and *ShiftCalendars* can be found in Chapter 8.

Category	***Topic***	***Examples***
Discrete Material Flow	Dismantle Station	Scrapyard
Resources	Worker	Carry One Part
Resources	Worker	Carry Multiple Parts
Resources	Worker	Worker Strategies
Resources	Worker	Manpower

The topics that were briefly introduced in this chapter will be explained in more detail in Part B of this tutorial. In addition, a few other more advanced features will be presented, such as 3D modelling and Simulation Optimisation.

Part B: Advanced Simulation Modelling

MODELLING A CAR MANUFACTURER

7 Building a Model: Car Manufacturer

In Part A, Chapters 3 till 5, you have learned the basics of the Plant Simulation software and how to conduct a simulation study. In Chapter 6, you received a preview of the more advanced Plant Simulation features. Throughout these chapters, we used a General Practitioner's office as running example. In Part B, Chapters 7 till 9, we will go more into depth on the possibilities of Plant Simulation, now using a Car Manufacturer as running example. As you can imagine, the factory of the Car Manufacturer consists of various elements, such as conveyor belts and transporters.

This chapter will once again treat some of the basics in building a model, but in contrast to the previous chapters, you will now create your own building blocks. Until now you have been building your model in the *RootFrame* with objects provided by Plant Simulation. In this chapter you will learn about the concept of *Frames* and you will make a first basic factory. You will create your own elements of the factory and will use *Animation Points* to improve the visualisation when using *Frames*. This chapter will also go more into depth on debugging for which the *Method Debugger* could be used and other useful ways to detect errors in your model.

Subjects dealt with in this chapter:

- *Frames*
- *Animation Points*
- *Interface*
- *FlowControl*
- *Debugging*
- *State Dependent Icons*
- *Failures*

7.1 Setup of the Car Manufacturer

Before we start building the *Frames* and all the other components of the factory, we will setup the products this factory produces. This particular Car Manufacturer produces two types of cars, namely the XX and the XY type. Both types of cars share common production steps, but XY requires an additional production step, namely a special paint coating. We will set up the different *MUs* in *Plant Simulation* similar to what we have done in Section 3.7.

Task: Create a MU Car Type

1. Start Plant Simulation and choose *Create New Model.*
2. Click *2D only* in the dialog window.
3. Rename the current *Frame* within the folder *Models* to *CarFactory.*
4. Right-click on the object *Entity* in the folder *MUs*.
5. Select *Duplicate.* You now have an object named *Entity1* in the folder *MUs*.
6. Rename *Entity1* to *CarType.*
7. Derive from the *MU CarType* two new *MU*s.
8. Rename them *XX* and *XY*, respectively.

7.2 Frames as Building Blocks

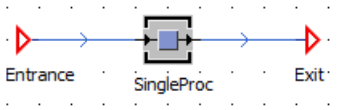
Previously, we have built models within the *RootFrame* (see Chapter 2). If your model grows larger, you may want to add hierarchy to your model for a better overview and to avoid making mistakes. Furthermore, you might want to create your own objects in the *Class Library* consisting of multiple standard building blocks. For these purposes, the object *Frame* is useful. Within a *Frame* we can group existing objects and build hierarchically structured models. You can create new *Frames* within the *Class Library* by right-clicking on the folder *Models* and selecting *New* > *Frame*.

The *Frame* you renamed to *CarFactory* will serve as the *RootFrame* of the model we are going to build. Within that model we need to set up a conveyor belt in order to transport the cars. The conveyor belt will consist of four basic components, namely a straight, bend, confluence and branching object. For each of these basic components, we will use a separate *Frame*.

Before we start building a new *Frame*, we need to introduce a new object, namely the object *Interface*. The *Interface* gives the possibility to connect *Frames* with other *Frames*, including the *RootFrame*. When multiple *Frames* are connected with each other, *MUs* can flow through them. You can find the object *Interface* in the *Class Library* in the folder *MaterialFlow*. We will now build our first *Frame*.

Task: Create a Straight

1. Right-click on the folder *Models* and create a new *Frame.*
2. Rename it *Straight*.
3. Insert two *Interfaces* and one *SingleProc* in the *Frame*.
4. Connect the *Interfaces* with the *SingleProc.*



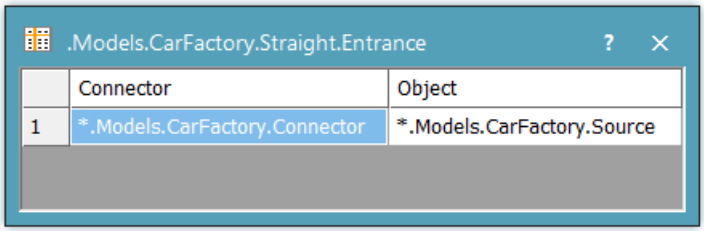
5. Double-click on the *SingleProc* and check whether the *Processing time* is 1 minute.
6. Right-Click on the *Frame Straight* in the folder *Models.*
7. Open the *Icon Editor* by clicking *Edit Icons.*
8. Draw a suitable icon for the object *Straight*, indicating a straight transport belt.

If you would like to refresh your memory regarding *Connectors* and connecting objects, see Section 3.2. In order to distinguish between the *Frames* we are going to build, it is useful to give them customised icons. You can refresh your knowledge on *Icons* and the *Icon Editor* in Section 3.8.

Task: Test your Frame

1. To add a *Frame Straight* to your *RootFrame CarFactory*, simply drag a *Straight* from your *Class Library* into your *RootFrame CarFactory*.
2. Add a *Source* and *Drain* to your *RootFrame CarFactory.*
3. Leave the settings at default.

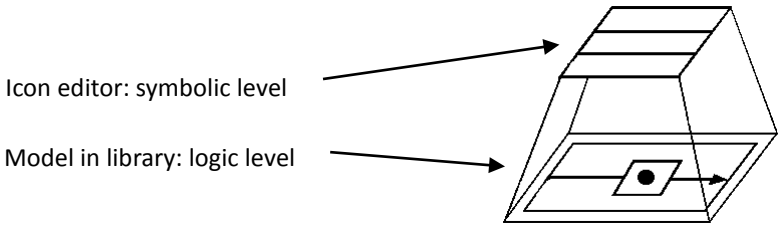
4. Connect the *Source* with the *Frame Straight*.
5. Connect the *Frame Straight* with the *Drain*.
6. Double-click on the *Straight* in your *RootFrame CarFactory*.
7. Right-click on *Entrance* and select *Open External Connections List*.



Note that the object *Source* is connected correctly.

8. Check that the object *Drain* is connected correctly as well (to the *Exit* interface).
9. Go to the *RootFrame* and run you model at a low speed (see *EventController*).

You might have noticed, while looking at the *RootFrame* only, that the *MUs* in your model are not displayed on the object *Straight*. The reason for this is that we have not defined *Animation Points* in the icon of the *Frame Straight*. We need to make a distinction between the logic level and the symbolic level of each *Frame* or each application. The logic level describes the functionality of the object; this level consists of the standard building blocks you use to create your model in the *Frame*. The icons and animation belong to the symbolic level. The distinction between the logic and symbolic level is illustrated below.

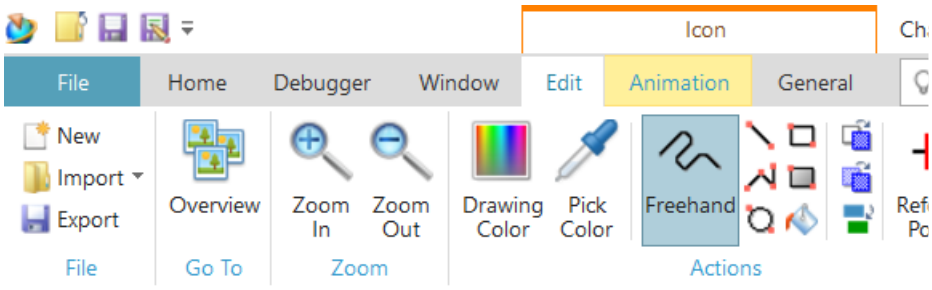


The animation points have to be inserted into the icon and linked to the objects on the logic level to allow *MUs* that move through the objects on the logic level to be visible on the symbolic level. During simulation, the icons of the *MUs* are displayed on the animation points of the objects they are located on.

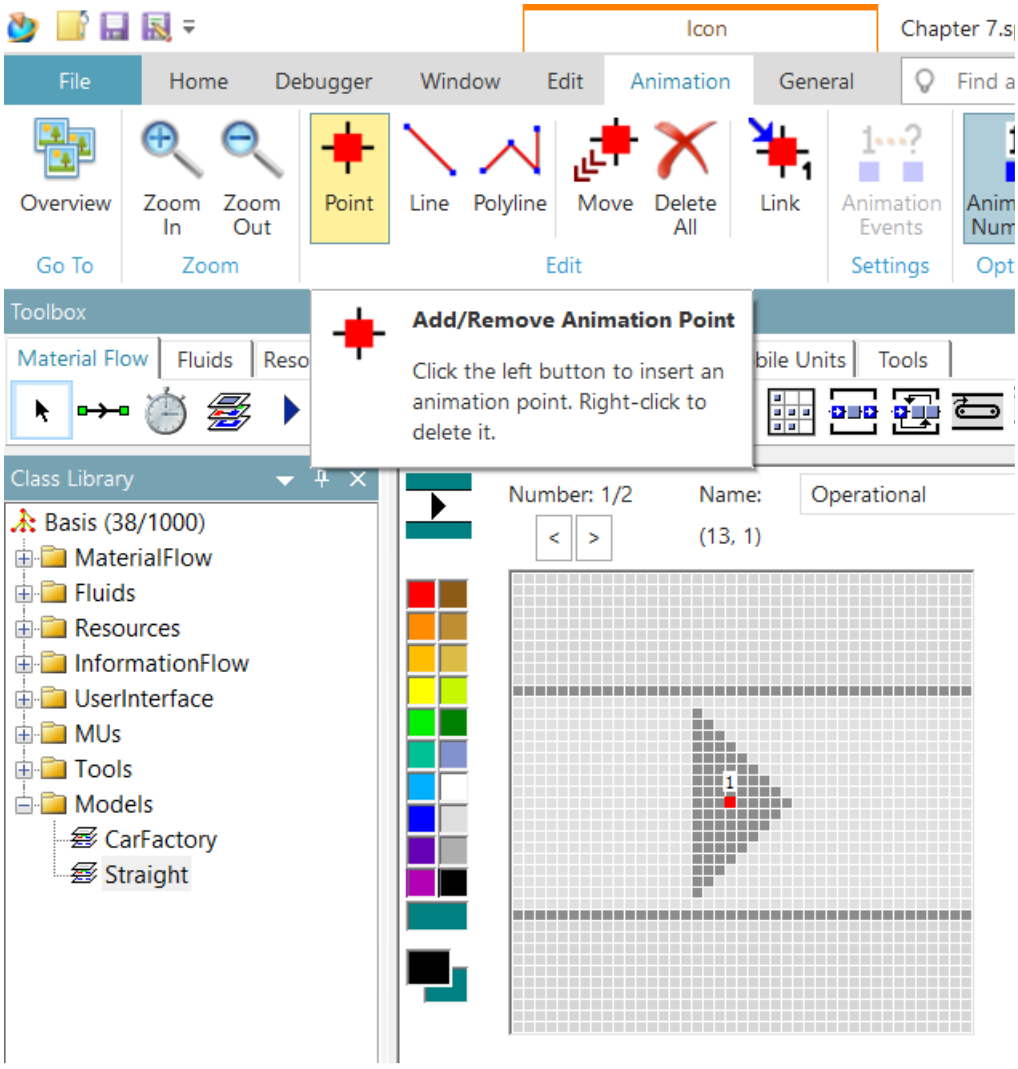
To determine how the icon of the *MU* is assigned to the animation point of the object icon, the *MU* icon has a *Reference Point*. When assigning the icon of the *MU,* the program positions this reference point onto the animation point. By default, the reference point of basic objects is located in the centre. In the *Icon Editor* it is denoted by a red pixel.

Task: Setting Animation Points

1. Right-Click on the *Frame Straight* in the folder *Models.*
2. Open the *Icon Editor.*
3. Switch to *Animation Mode* by clicking on the menu *Animation* in the *Ribbon*.



4. By default, the Icon *(Add/Remove Animation) Point* is selected. Add an *Animation Point* to the *Icon.*



5. Link the *Animation Point* to the *SingleProc* located at the *Frame Straight*, by clicking the icon *Link (Animation Point)*, which is the  icon.
6. Click on the *Animation Point* you have set and it will open the *Frame* with its objects.
7. Click on the object you want to assign the animation point to, which in our case is the *SingleProc.*

8. Open the *RootFrame* and run your model. If you have done this task correctly, the object *Straight* now also displays *MUs* on the icon level.

In Chapter 2 we have stressed the importance of the *Inheritance* concept. This also applies to *Frames*. If we adjust the *Frame Straight* within the *RootFrame* directly, this would mean that only the specific *Straight* we have placed within the *RootFrame* will be adjusted. If we adjust the *Straight* by clicking on the object in the *Class Library*, this change would affect all the instances of *Frames Straight* due to inheritance.

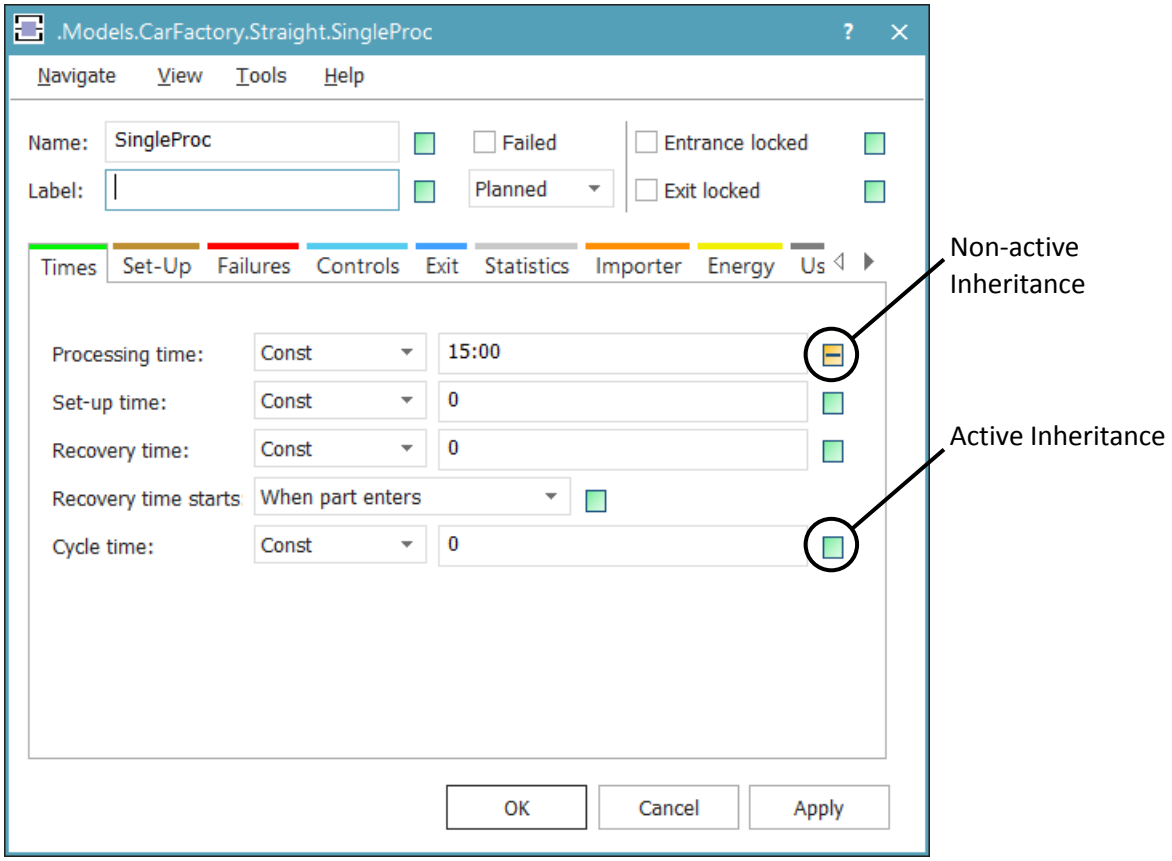
The differences between the classes and objects derived from the classes can also be seen from their location. For example, when you double-click the object *Straight* in the *RootFrame*, the location ".Models.CarFactory.Straight" appears and when you double-click the *Frame Straight* from the *Class Library*, the location ".Models.Straight" appears. We will illustrate this with an example.

Task: Inheritance with Frames

1. Remove the current *Connectors* between the objects in the *RootFrame.*
2. Insert two additional *Straights.*
3. Connect the objects.

   Note that each *Straight* has a *Processing time* of 1 minute by default.
4. Double-click on the *Frame Straight* in the *Class Library*.
5. Change the *Processing time* of the *SingleProc* to 5 minutes.
6. Return to the *RootFrame* and note that each *Straight* has a *Processing time* of 5 minutes.
7. Change the *Processing time* of the first *Straight* to 15 minutes by double-clicking on it in the *RootFrame*.
8. Note that the *Processing time* of the other *Straights* remain 5 minutes.
9. Double-click on the *Frame Straight* in the *Class Library*.
10. Change the *Processing time* of the *SingleProc* back to 1 minute.
11. Open the first *Straight* in the *RootFrame* and note that it is still 15 minutes, because you made a direct adjustment to this object disabling inheritance of this property.

The previous task illustrated that inheritance in the manually adjusted *Frame* was partly disabled. By changing an attribute of a single instance of a *Straight*, we have set the inheritance for that attribute of that *Straight* to non-active.

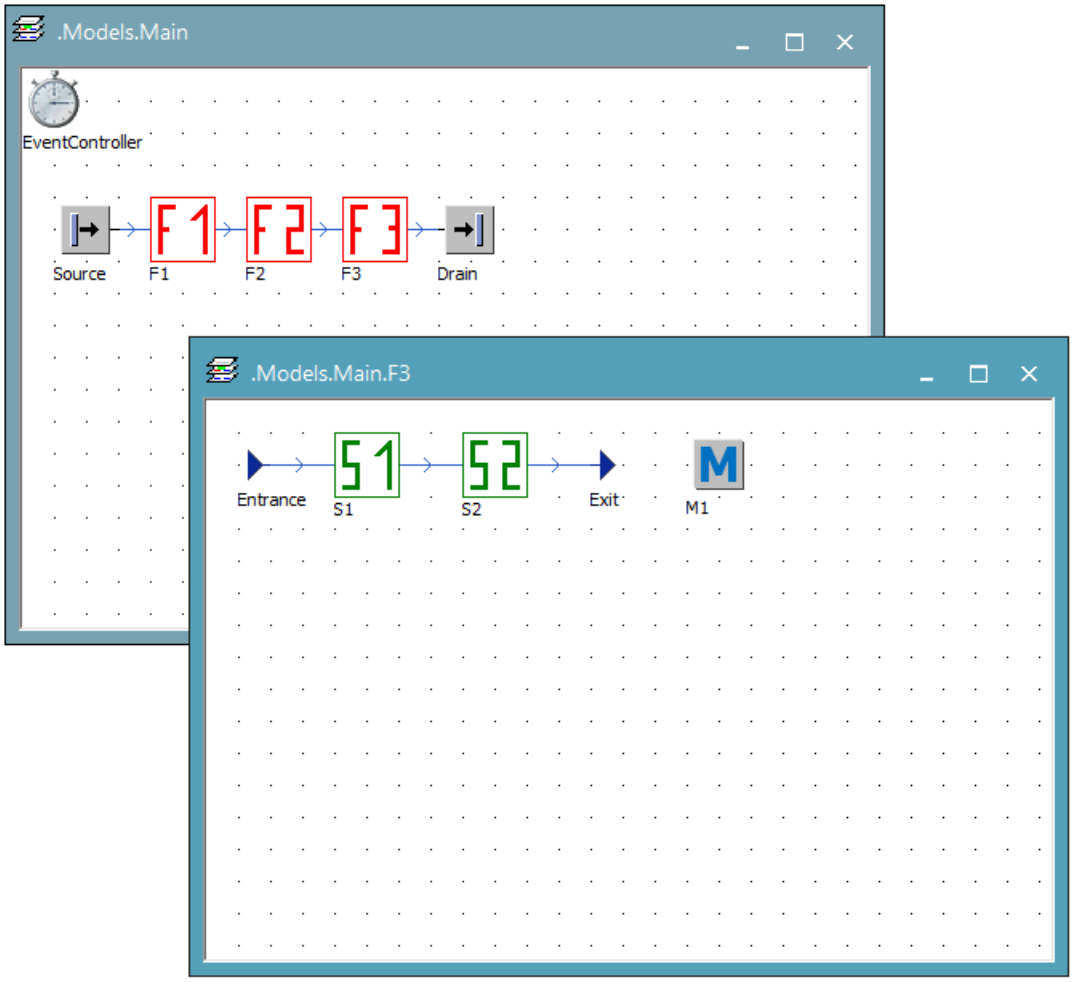


By manually changing the processing time to 15 minutes, we changed the active inheritance setting to non-active. If we would like to change it back to active inheritance, we only have to click on the inheritance button of the *Straight* instance as shown in the figure above. If you do so, you will notice that the processing time is set the same as its parent, namely 1 minute. The inheritance selection box, as illustrated in the figure above, can be particularly useful when you want to inherit only some attributes from the parent. For example, a machine has the same processing time as its parent, but different failure behaviour. Then you inherit the processing time, but not the failure behaviour.

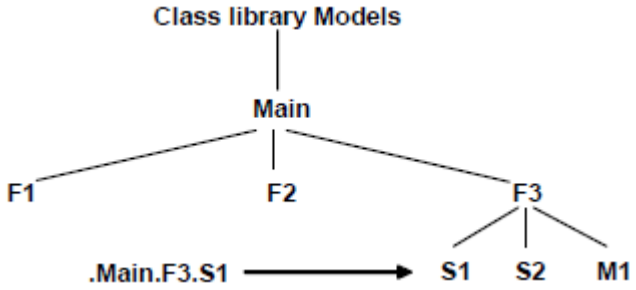
Did you know?

If you accidentally made adjustments to the object (*Child Frame*) instead of to the class (*Parent Frame*), thereby loosing active inheritance, there is an easy way to restore it. Go to the *Parent Frame* itself and *select* everything (Ctrl-A) within the *Frame*, *cut* all the objects (Ctrl-X) and then *paste* the objects back (Ctrl-V). After doing so, all settings in the *Child* (the object you inserted in the *RootFrame*) are exactly the same is in the *Parent*, with active inheritance. If we look at the previous task, we could restore the settings of the first *Straight* by clicking on the *Frame Straight* in the *Class Library* and *selecting* all objects, *cut* them, and then *paste* them back.

To make efficient use of the inheritance concept, you will need to understand the way *Plant Simulation* addresses and accesses objects. The way in which Plant Simulation accomplish this is by using paths. We illustrate the idea of paths by using an example. The example consists of one *RootFrame* called *Main* with three *Frames* (*F1, F2,* and *F3*) on it, and the *Frame F3* consists of the *Frames S1* and *S2* and a *Method M1*. The Plant Simulation implementation can be found in the following figure.



The hierarchical structure of this model is shown in the following scheme.

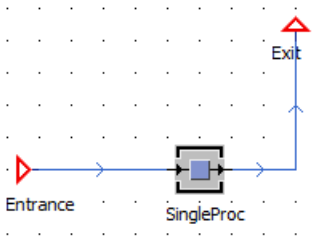


One way to access the *Frame S1* is to use the absolute path, which is ".Main.F3.S1". As you have noticed while working with *Plant Simulation*, the period is used as a separator when we change to the next hierarchy level. The period at the beginning denotes that we are using an absolute path. If a path begins with a period, *Plant Simulation* begins its search for the first object mentioned in the path in the *Class Library*. An alternative way of using an absolute path is to refer to the *RootFrame*; to access *Frame S1* we can use "root.F3.S1".

As opposed to the absolute path, a relative path does not start in the *Class Library* but in a name space. Thus, the path does not start with a period, but with a name. If, for example, you mention the name *S1* in the *Method M1* (in *Frame F3* on the model *Main*), *Plant Simulation* searches for the object *S1* within the active name space (*Frame F3*). Alternatively, we could use the tilde ~ to refer to a name space one level up in the hierarchy; to access F2 from the *Method* M1 on F3, you can use "~.F2" (although the absolute path "root.F2" would be easier here).

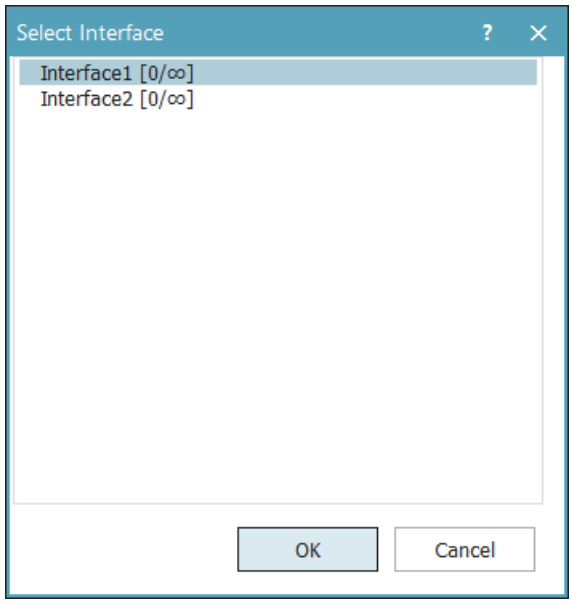
Task: Create a Bend

1. Right-Click on the folder *Models* and create a new *Frame.*
2. Rename it *Bend*.
3. Insert two *Interfaces* and one *SingleProc* in the *Frame* (use Ctrl-T to rotate objects).
4. Connect the *Interfaces* with the *SingleProc* as indicated below (to create corners in the connector, use the Ctrl-Key).



5. Draw an icon for the object that represents a 90° turn and set an *Animation Point*.
6. Double-click on the *SingleProc* and set the *Processing time* to 2 minutes.
7. Close the application object.

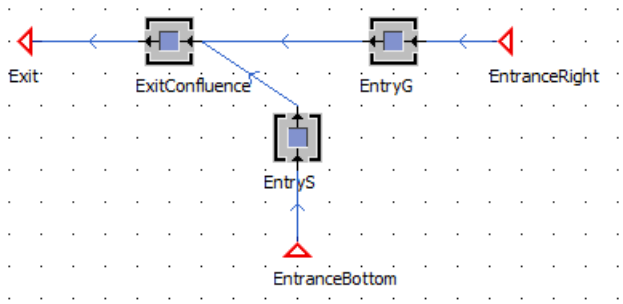
When you build an object with multiple entry or exit points and you would like to connect them, you will get a list with possible entries or exit interfaces (if there is more than one unconnected interface). You will then need to select the desired entry or exit interface from a list:



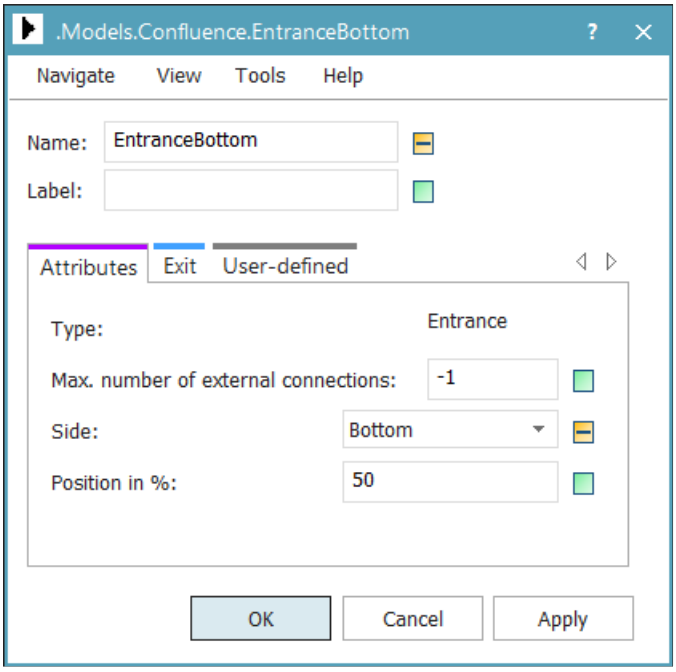
As you can see in this particular example, *Interface1* and *Interface2* have 0 external connections from the maximum possible amount of infinite external connections. You can change the maximum amount of external connections by double-clicking on the *Interface* and adjust *Max. number of external connections*. We now create an object *Confluence*, where two belts come together into one belt.

Task: Create a Confluence

1. Right-Click on the folder *Models* and create a new *Frame.*
2. Rename it *Confluence*.
3. Insert three *Interfaces* and three *SingleProcs* in the *Frame*.
4. Connect the *Interfaces* with the *SingleProcs* as indicated below.



5. Rename the objects accordingly.
6. Set the *Processing time* of the *SingleProc ExitConfluence* to 0.
7. Make sure that the connection side of every *Interface* object is correct by double-clicking an *Interface* object and selecting the correct option from the drop-down menu *Side*.

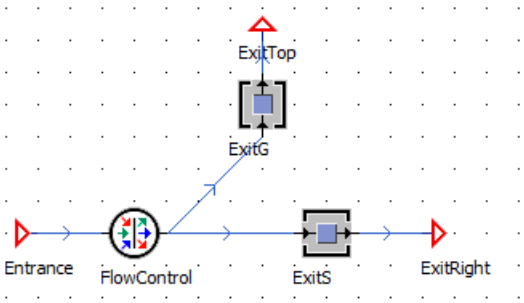


8. Insert three *Animation Points* on the *Frame* icon.
9. Assign the *Animation Points* of the icon to the three *SingleProcs.*
10. Close the application object.

The final *Frame* we will construct for our conveyor belt is for the object *Branching*. Sometimes, given a condition, the *MU* can follow different paths. In order to make that possible, we will need an object that allows *Branching*. A useful object for the *Frame Branching* is the standard object *FlowControl*, which can be found in the folder *MaterialFlow* from the *Class Library.* The object can be used when you need to model common strategies for the flow of materials. The *FlowControl* itself, however, does not hold/process a *MU*, but only ensures the distribution of the *MUs* to the subsequent stations.

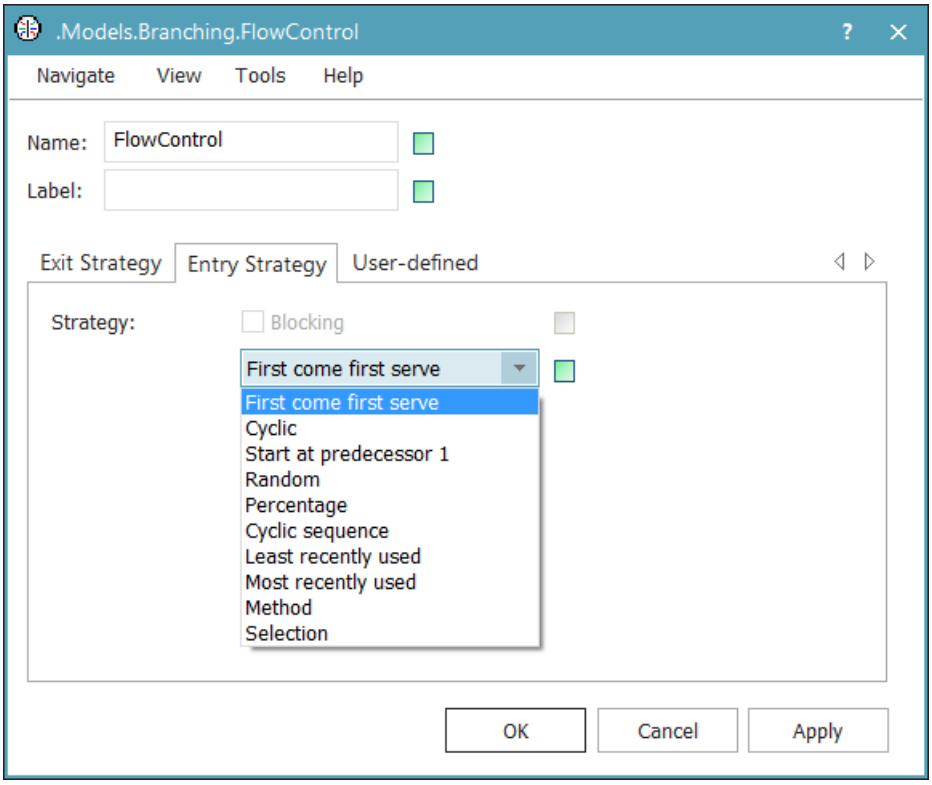
Task: Create an object Branching

1. Right-Click on the folder *Models* and create a new *Frame.*
2. Rename it *Branching*.
3. Insert a *FlowControl* in the *Frame.*
4. Insert three *Interfaces* and two *SingleProcs* in the *Frame*.
5. Connect the objects as it is indicated below.



6. Rename the objects accordingly.
7. Make sure that the connection side of every *Interface* object is correct by double-clicking an *Interface* object and choosing from the drop-down menu *Side*.
8. Draw an icon for the *Frame.*
9. Insert two *Animation Points*.
10. Assign the *Animation Points* of the icon to the two *SingleProcs.*
11. Close the *Frame*.

For now, we set the *Entry Strategy* of the *FlowControl* to *First Come First Serve*. Double-Clicking on the object and selecting the tab *Entry Strategy* shows a range of common strategies. We will later (in Section 7.5) add a *Method* to distribute the *MUs* over the assembly line.



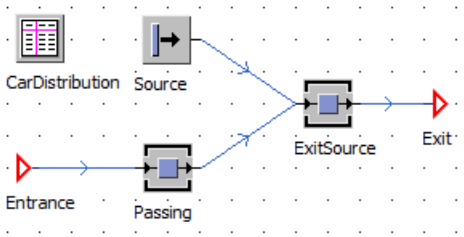
You might recall from Section 3.9 that we used *Methods* for distributing the *MUs* over subsequent stations. The object *FlowControl* is just another possibility for distributing *MUs*.

7.3 Source and Drain

Besides a conveyor belt to distribute the products, we will also need a *Source* and *Drain* object. However, because a conveyor belt system is often a closed circuit, products are often able to surpass the *Source* or *Drain*. The original *Source* and *Drain* objects cannot be surpassed when they are placed in the middle of two *Straights*. Therefore, we decided to create our own *Source* and *Drain* object.

Task: Create a Source

1. Right-Click on the folder *Models* and create a new *Frame.*
2. Rename it *Source*.
3. Insert two *Interfaces* and two *SingleProcs* in the *Frame*.
4. Insert a *TableFile* and a *Source* object.
5. Connect and rename the objects as indicated below.



6. Set the *Processing time* of the *SingleProc ExitSource* to 0 minutes.

7. Set the interval time of the *Source* to *Negexp* 8 minutes (Poisson arrivals).
8. Draw an icon for the object and set three *Animation Points*.
9. Assign the *Animation Points* to the *SingleProcs* and to the *Source.*
10. Configure your *TableFile* as indicated below (see Section 3.7, note that you can also drag the *TableFile CarDistribution* onto the Source and set the *MU Selection* in the *Source* to *Random*).

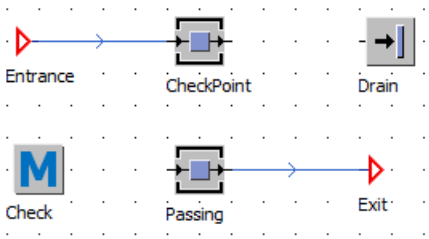
	object 1	real 2	integer 3	string 4	table 5
string	MU	Frequency	Number	Name	Attributes
1	.MUs.XX	0.75			
2	.MUs.XY	0.25			

11. Set the *TableFile* as input for your *Source* object (see Section 3.7)
12. Add a *Variable* to your *Frame Source* and name it *StationName*.
13. Set the data type of the *Variable* to *String* with the value *Source*.
14. Close the application object.

We also need to create a *Drain* from which finished products can leave the system. However, products who still need to undergo certain production steps, need to be able to surpass the *Drain.*

Task: Create a Drain

1. Right-click on the folder *Models* and create a new *Frame.*
2. Rename it *Drain*.
3. Insert two *Interfaces* and two *SingleProcs* in the *Frame*.
4. Insert a *Method* and *Drain* object.
5. Connect and rename the objects as indicated below.



6. Set the *Processing time* of the *SingleProc CheckPoint* to 0 minutes.
7. Set the *Method Check* as *Entrance Control* for the *SingleProc CheckPoint.*
8. Insert the following code in the *Method Check*.

```
-- Method to determine whether a Product could leave the Assembly Line
if @.dest = "Drain"
    @.move(Drain)
else
    @.move(Passing)
end
```

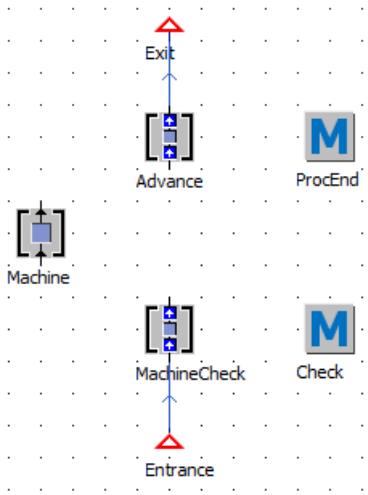
9. Add a *Variable* to the *Frame Drain* and name it *StationName*.
10. Set the data type of the *Variable* to *String* with the value *Drain*.
11. Draw an icon for the object and set three *Animation Points*.
12. Assign the *Animation Points* to the *SingleProcs* and to the *Drain.*
13. Close the application object.

7.4 Processing Stations

In order to process the different Car Types, we need to install Processing Stations, namely a station that will assemble the cars and a station that will paint the cars (only of the type XY). We assume that only one car can be handled at a station at the same time. If the station is occupied, the *MU* will need to surpass the station and continue on the assembly line. It will have to return at the station at a later point in time.

Task: Create Processing Stations

1. Right-click on the folder *Models* and create a new *Frame.*
2. Rename it *ProcessingStation*.
3. Insert two *Interfaces*, one *SingleProc*, and two *Buffers* in the *Frame*.
4. Add two *Methods* to the *Frame*.
5. Connect and rename the objects as indicated below.



6. Set the *Dwell time* of the *Buffers* to 0, the buffer sizes to infinite (capacity of -1), and the *Processing time* of *Machine* to 2 minutes.
7. Set the *Method Check* as *Exit Control* for the *Buffer MachineCheck*.
8. Insert the following code in the *Method Check*.

```
if Machine.empty and @.dest=StationName
    /*The station has to be empty for the MU to be processed,
        if not it has to pass the processing station */
    @.move(Machine)
else
    @.move(Advance)
end
```

9. Set the *Method ProcEnd* as *Exit Control* for the *SingleProc Machine*.
10. Insert the following code in the *Method ProcEnd*.

```
@.StateProc += 1
@.Dest := root.ProductionPlan.bestNdest(@.name, @.StateProc)
@.move(Advance)
```

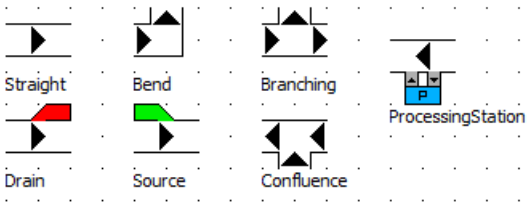
    Note that we will create the *Frame ProductionPlan* in Section 7.5 for which this code is needed.

11. Add a *Variable* to the *Frame ProcessingStation* and name it *StationName*.
12. Set the data type of the *Variable* to *String* with the value *EnterStationNameHere*.
13. Draw an icon for the object and set three *Animation Points*.
14. Assign the *Animation Points* to the three *SingleProcs.*
15. Close the application object.

Note: Anonymous Identifiers

We have addressed two anonymous identifiers, namely ? and @, in Section 4.2, but there are more of them. In step 10 of the previous task we have used for instance *root* as an identifier. We also could have replaced the code *root* by *location* (states the location of the *MU* or object) or by ~ (refers to a *Frame* one level up in the hierarchy). Other anonymous identifiers are *current*, *rootFolder*, and *self*.

In total we have now created eight building blocks of our car manufacturer: seven custom objects and one main model to connect them. With this, we are able to construct a first version of our factory. We have used the following icons for our custom objects. You are encouraged to use your own icons.



7.5 Production Plan

Before connecting the building blocks of the car manufacturer, we need to take care of proper routing on the assembly line. To support this, we create a new *Frame* called *ProductionPlan*. To refresh your memory regarding working with the object *TableFile*, check Chapter 4.

Task: Create a Production Plan

1. Right-click on the folder *Models* and create a new *Frame.*
2. Rename it *ProductionPlan*.
3. Insert a *TableFile* and a *Method* in the *Frame*.
4. Name the *TableFile aPlan* and the *Method bestNdest*.
5. Configure the *TableFile aPlan* as follows.

	string 0	string 1	string 2	string 3
string	Product	Station1	Station2	Station3
1	XY	Assembly	Paint	Drain
2	XX	Assembly	Drain	

6. Insert the following code in the *Method bestNdest*.

```
-- The function bestNdest returns the destination of the product.
-- Input is:
-- muType: type of MU
-- stateProd: step in the production process
param muType: string, stateProd: integer -> string
-- 2 input variables, which results in 1 output of the type string.
result := aPlan[stateProd, muType]
```

   Note that this *Method bestNdest* acts as a function of two variables (of type string and integer) that returns a string. To refresh your memory on using inputs and outputs of *Methods*, check Section 3.12.

7. Give the *Frame* a suitable icon and close the *Frame ProductionPlan*.
8. Open the *Frame Branching*.
9. Add a *Variable* to the *Frame* and name it *SuccessorLeft,* with data type *String*.
10. Add three *Methods* on the *Frame* and name them *Branch, Reset* and *Init*.
11. Insert the following code in the *Method Reset*.

```
SuccessorLeft := ""
```

12. Insert the following code in the *Method Init* and try to understand its purpose.

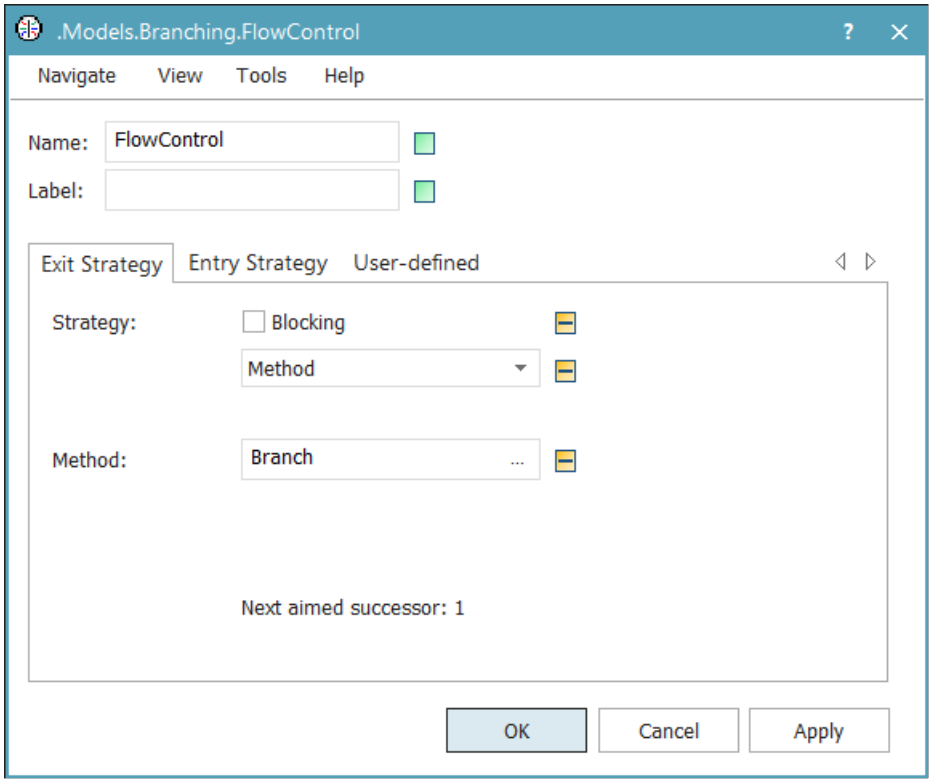
```
-- Determine the station on the left
SuccessorLeft := ExitG.succ.~.StationName
```

13. Insert the following code in the *Method Branch*.

```
param r: integer -> integer
-- @ is the incoming product
if @.dest = SuccessorLeft
    return 1
else
    return 2
end
```

   Take care of using the correct return numbers for the object *FlowControl*. The numbers of the outgoing connectors can be found by clicking on the *General* menu in the *Ribbon* while the *Frame Branching* is active, and going to *View Options* > *More* > *Successors*.

14. Adjust the settings of the object *FlowControl* to the following.



15. Close the *Frame Branching.*
16. Insert a new *User-defined Attribute* for the *MU CarType* and name it *dest*. Set the data type to *String* and give it the value *Assembly*.
17. Insert a new *User-defined Attribute* for the *MU CarType* and name it *StateProc*. Set the data type to *Integer* and give it the value 1.

We have inserted a *Method Reset* and *Init* in the previous task. As you might recall, these type of *Methods* have a special purpose in *Plant Simulation*. You can refresh your memory on those *Methods* in Chapter 3 and Chapter 4. In this particular case, the *Reset* sets the value for the *Variable SuccessorLeft* to blank again. The *Init* is the first *Method* to be executed when you start the simulation and in this case it will determine what processing station is connected to the object *ExitG*, which is stated in the *Variable StationName* in the *Frame ProcessingStation*.

After a *MU* leaves the object *Machine* in the *Frame ProcessingStation*, the *Method ProcEnd* is triggered, which gives the *MU* new values for its *User-defined Attributes StateProc* and Dest. Via the *TableFile aPlan* the destination of the *MU* is determined, which is triggered by the function stated in the *Method bestNdest*. This *Method* gives the *MU* its next destination after it has been processed in the *ProcessingStation*. Recall that the initial destination of both *MU* types is set to be *Assembly*.

Note: The Tilde ~

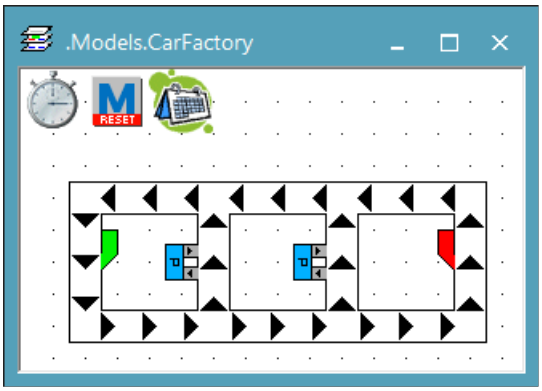
You might have noticed that we have used the sign ~ in the code of the previous task. The tilde allows us to navigate to the next higher level in the *Frame* hierarchy without having to state the exact name (see Section 7.2). This makes it easier to generalise certain steps.

Task: First Version of the Factory

1. Clean the *Frame CarFactory* and make sure only the *EventController* is present in the *Frame*.
2. Add the following objects to the *Frame CarFactory*: 6 *Straights*, 2 *Branchings*, 2 *Confluences*, 4 *Bends*, 1 *Drain*, 1 *Source*, 2 *ProcessingStations,* and the *ProductionPlan*. Make sure that the *Processing time* of the *SingleProc* in *Straight* is 1 minute.
3. Add a *Method Reset* to your *RootFrame* and add the following:

```
DeleteMovables
.Mus.CarType.initStat
```

   The *initStat* command is used to reset the statistics of car types *XX* and *XY*. After a run, you can click on these car types in the *Class Library* to find relevant statistics such as the average throughput times (lifespan) of the generated instances.
4. Create the following layout for the assembly line of this Factory and make sure that all customised objects that have interfaces are correctly connected.



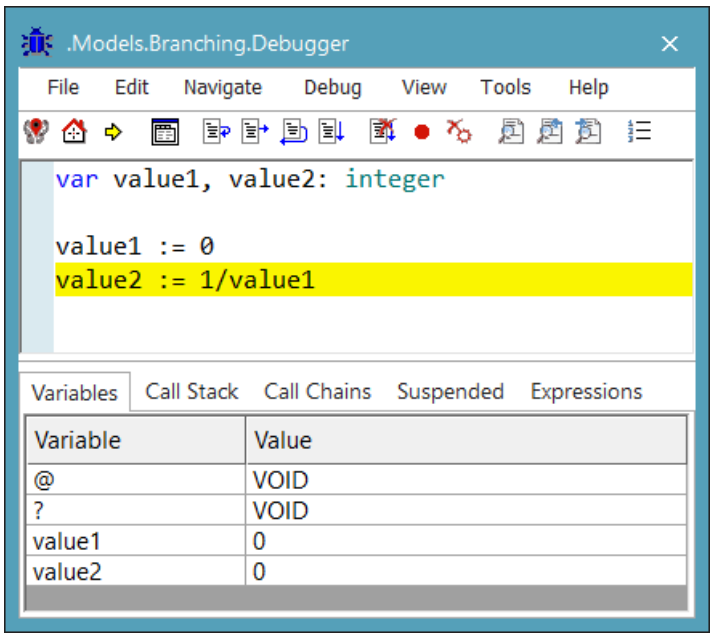
5. The two processing stations are two similar instances of the class *ProcessingStation*. Apply the necessary changes such that the machine on the left becomes the painting station and the machine on the right the assembly station. Hint: consider the logic placed on the *Frames Branching* and *ProductionPlan*. Further, increase the *Processing time* of the painting *Machine* from 2 minutes to 3 minutes.
6. Set the end time of your simulation to 100 days in the *EventController* and run your model. The mean life time of the cars XX and XY should be approximately 26 and 41 minutes respectively.

7.6 Debugging

No matter how many time you spend constructing a model or writing *Methods*, mistakes are easily made. Especially now you are able to construct larger models with the help of *Frames*. Mistakes in Plant Simulation, or in any other programming environment, lead to so-called *bugs*. A short and general definition of the word bug is: any error or fault that leads to unwanted behaviour of your program. Debugging is the task of repairing those errors and faults. You have already seen some tools Plant Simulation has to aid you debugging:

- Animation (Section 2.6)
- The step button in the Event Controller window (Section 3.3)
- The event list (Section 5.3)

A type of bug that we have not discussed specifically are bugs in *Methods*. While working through the tutorial, you have written a decent amount *Methods* (code) already. Some of your *Methods* might grow quite large and figuring out why a *Method* is not functioning as intended can be difficult. Although you might discover these bugs by using the tools mentioned above, Plant Simulation offers you an even better option to find and repair them, namely the *Method Debugger*. The *Method Debugger* pops up when a *Method* contains a line of code that cannot be executed or results in an error, but can also be opened manually.



The *Method Debugger*

The *Watch Window*

 *Step Over* (F10)

 *Continue* (F5)

 *Watch Window* (F12)

 *Step Into* (F11)

 *Terminate Simulation*

 *Insert or Remove Class Breakpoints* (F9)

 *Step Out (*Shift +F11)

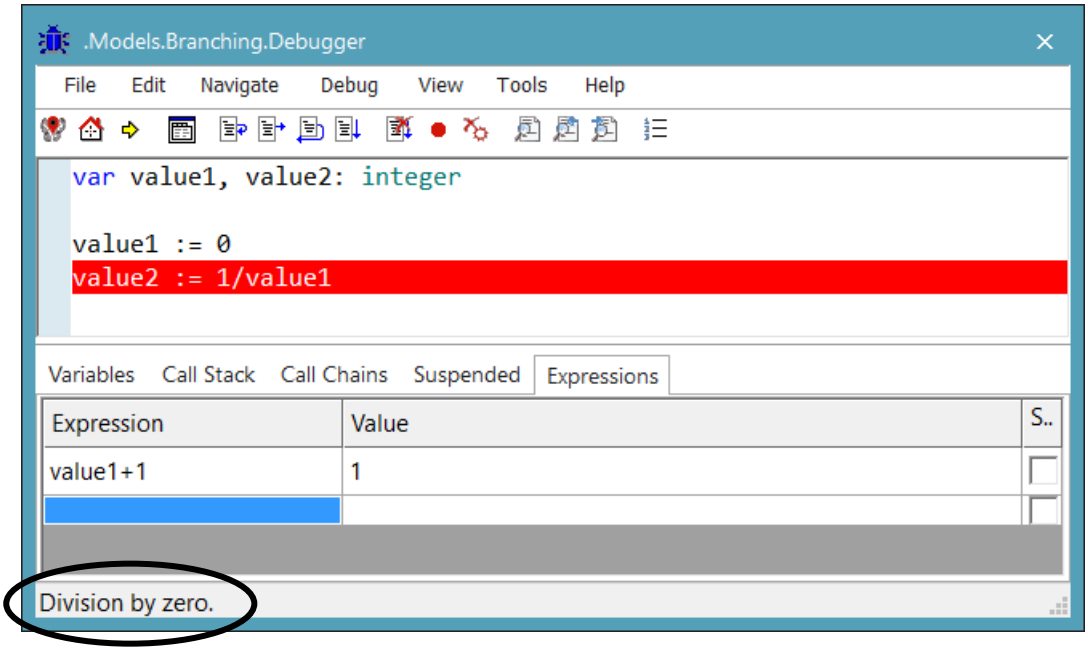
 *Remove all Breakpoints*

You can also find these functions in the *Method Debugger* by clicking on the tab *Debug*.

Note: The Method Debugger and Method Editor

It is important to understand the difference between the *Method Debugger* and the *Method Editor.* The *Method Debugger* is not made to edit *Methods*; its purpose is purely to evaluate. Changing *Methods* in the *Method Debugger* may cause unexpected events. However, it is possible to evaluate certain expressions while debugging by making use of the tab *Expressions* in the *Watch Window*. When using the tab *Expressions,* the source code is not modified, but the *Method Debugger* gives the resulting value from an expression given by the user. This functionality can be useful if you want to test different solutions in order to fix a bug in the source code. Do not forget to delete your expressions afterwards, to avoid unintended execution at later points in time.

The *Method Debugger* checks for syntax errors, wrong paths, wrong names of objects, and more. The *Method Debugger* pops up and highlights the line that contains an error in red when Plant Simulation encounters an error during simulation. The *Method Debugger* also shows a message telling you what is wrong in the status bar of the *Method Debugger*. For example, if you divide by zero, you will see the following screen appearing when you run your model:



You will learn how to use several options incorporated in the *Method Debugger* by evaluating two relatively simple *Methods*. Everything you will learn in the remainder of this paragraph is even more useful when *Methods* grow larger or become more complicated.

Task: Use the Method Debugger

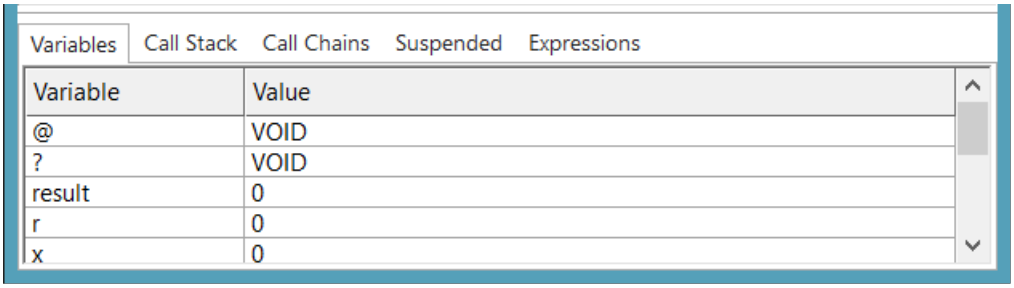
1. Open the *Branching* object in your *Class Library*.
2. Create two new *Methods* called *Randomdest* and *Randomise* in the object *Branching.*
3. Set the *Method Randomdest* as *Method* for the *Exit Strategy* of your *FlowControl.*
4. Insert the following code in *Randomdest*:

```
param r : integer -> integer

var x, value1, value2: integer

value2 := 0
value1 := Randomise
for x := 0 to value1
    value2 := value2 + 2
next

if value2 > 19
    return 1
else
    return 2
end
```

5. Insert the following code in *Randomise*:

```
-> integer

var value3: real

value3 := z_uniform(1, 5, 12)
result := floor(value3)
```

6. Return to the *Method Randomdest* and press *Tools* > *Run* > *Debug* in the *Ribbon* or F11. Note that the line is now yellow, since you opened the *Method Debugger* yourself and no error occurred yet.
7. If you do not see a *Watch Window,* open one by pressing *Debug* > *Watch Window*, F12 or the button  . Activate the *Variables* tab of the *Watch Window* if it is not already active. Note that the *Variables* tab of the *Watch Window* shows the values of the variables in your *Method*.

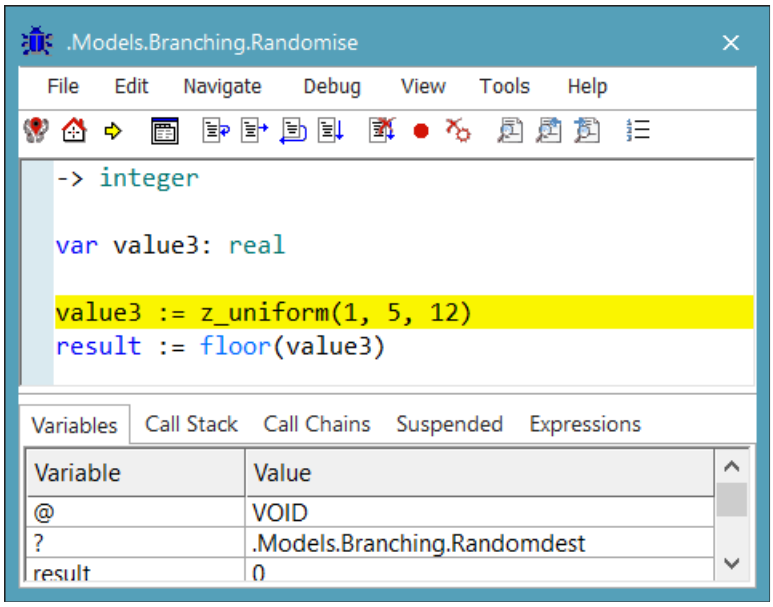


8. In the *Method Debugger,* press *Debug* > *Step Over*, F10, or the button  several times. Note how the *Watch Window* updates the values of your variables every time you *Step Over* a line. Continue using *Step Over* until you passed all lines of code and the *Method Debugger* closes.

In the previous task, the *Variables* tab of the *Watch Window* not only shows the variables defined within the *Method*, but also the anonymous identifiers "@" and "?". These two variables show you what triggered the *Method*. Since the *Method* in this task was activated by hand and not triggered by a *MU*, *Object*, or *Method*, the values of "@" and "?" are VOID. By using *Step Over*, the *Method Debugger* evaluated every line of code in the *Method Randomdest.* In line 6, the *Method Randomise* is called and the *Method Debugger* steps over it, whereas we might be interested in what is happening within the *Method Randomise*. To avoid this, we can use the *Step Into* functionality.

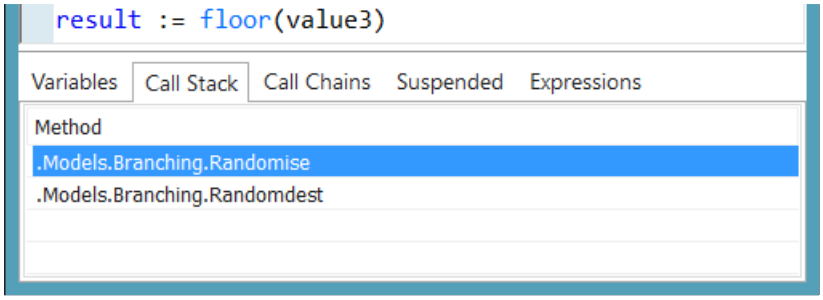
Task: Step into

1. Return to the *Method Randomdest* and open the *Method Debugger* (Step 6 of the previous task).
2. In the *Method Debugger*, use *Step Over* (F10 or the button ) on the first line and use *Step Into* (F11 or the button ) on the second line (alternatively use *Step Into* on both lines). After using *Step Into* on the second line, you will see the following screen:



As you can see you are not in the *Method Randomdest* anymore, but in the *Method Randomise*. Also note that the value of the "?" variable is now that of the *Method* activating *Randomise*: the *Method Randomdest.*

3. Activate the *Call Stack* tab of the *Watch Window*. This tab shows the *Methods* that are called. The bottom one is the first *Method* that was activated, the top one is the *Method* being executed at this moment and obviously the last one activated.



4. Open another *Watch Window* (F12 or the button ). The two *Watch Windows* operate independently so that you can keep track of two tabs at a time.
5. Step out of the *Method Randomise* (Shift+F11 or the button ).
6. Now *Continue* (F5 or the button ) to exit the *Method Debugger.*

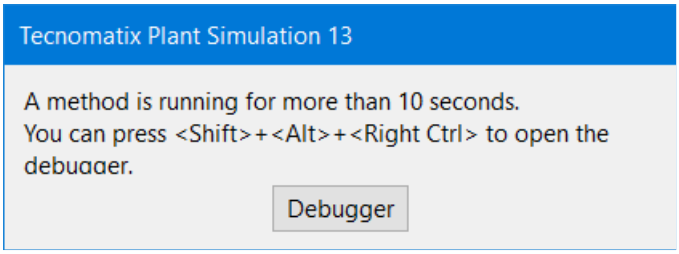
Step Over kept you within the *Method Randomdest* while *Step Into* also showed you exactly what happened in the *Method Randomise* and any later called *Methods* if there were any. If you step into a *Method* and later on realise this was not necessary, you can press Shift + F11 or the button  to return to the one that activated the current *Method*. You can mix the use of *Step Into* and *Step Over* any way you want to fix your bugs efficiently.

Note: Interrupting a never ending Method

Bugs in *Methods* may result in long running times or even in infinite loops, for example, when we reset the counter of our for-loop every time we do an iteration:

```
var x: integer

for x := 0 to 100
    x := 0
next
```

This loop will never end if we would run it (F5). Luckily the *Method Debugger* makes it possible to interrupt the *Method* by pressing the keys Ctrl+Alt+Shift at the same time. If the *Method* takes more than ten seconds, the following screen also appears:



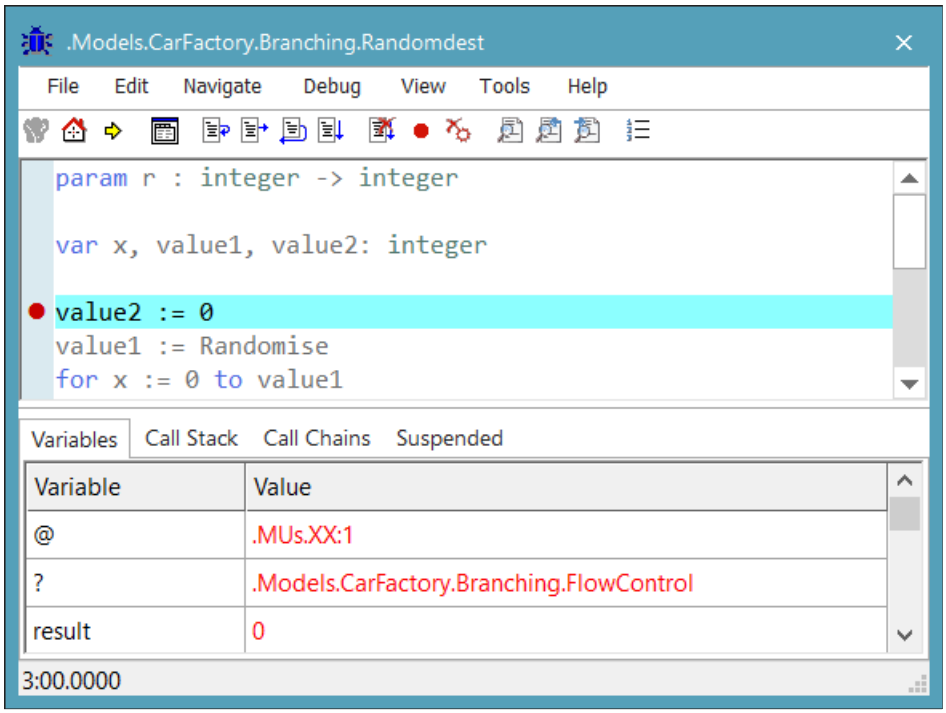
From this screen, you can open the *Debugger*, where you can end the run. This is an extremely simplified example of a for-loop that never ends, but the risk of infinite loops easily occurs when using while-loops.

Sometimes you need to know how a *Method* behaves during the actual simulation. This is possible by placing a *Breakpoint* in a *Method* at a specific line. The *Breakpoint* forces Plant Simulation to pause the run once it arrives at that line, after which the *Method Debugger* opens. There are two types of *Breakpoints* in Plant Simulation:

- *Class Breakpoints,* these *Breakpoints* account for every *Method* of that class.
- *Instance Breakpoints,* these *Breakpoints* only account for the specific instance.

Task: Making use of Class Breakpoints

1. Return to the *Method Randomdest.*
2. Enter a *Class Breakpoint* in line 5 (F9 or the button ).
3. Close the *Method* and run your model at medium speed. The following screen will appear:



From this point, you can evaluate your *Method* as you have done previously using *Step Over, Step Into*, and the *Watch Window.*

4. *Continue* (F5 or the button) to exit the *Method Debugger.* After a while the *Method Debugger* will reappear, triggered by a *Mu* passing a *Flow Control* in an instance of the object *Branching*.
5. For now you are done *debugging* and simulating so remove the break point (the button) and terminate the simulation (Ctrl+T or the button).

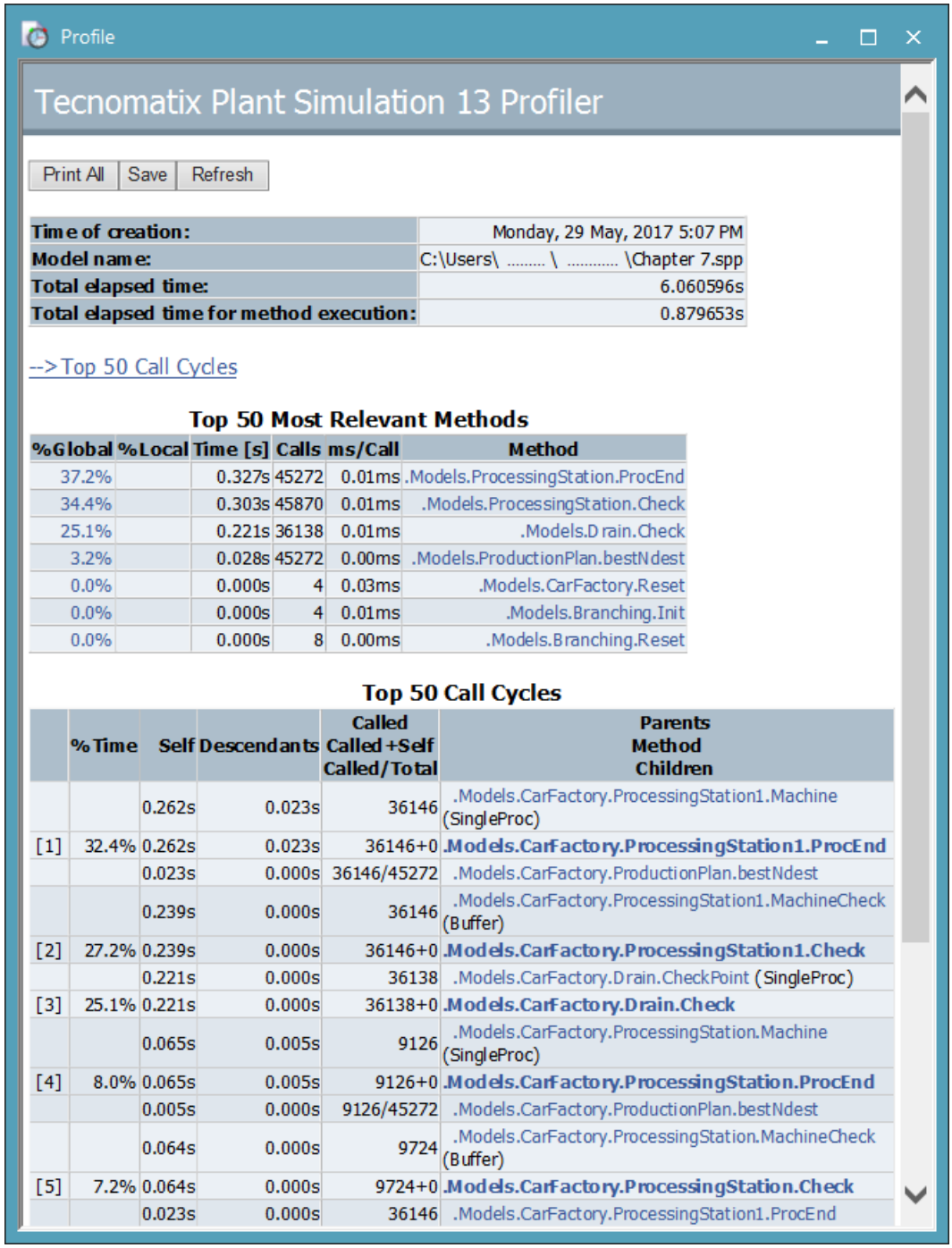
Note: Instance Breakpoints and Class Breakpoints

Instance Breakpoints have the same properties as *Class Breakpoints*, except for the fact that they are only triggered by the specific instance they are put in. Remember the task *Inheritance with Frames* you completed a few pages back and the difference between changing something in a class or an instance. The same principle applies to *Class Breakpoints* and *Instance Breakpoints*. With the small remark that you can enter a *Class Breakpoint* (by pressing *Debug > Class Breakpoint* in the *Method Debugger* or F9) in any instance and it will still appear in every other instance of that class. You can enter an *Instance Breakpoint* by pressing *Debug > Instance breakpoint* in the *Method Debugger* or Shift + F9.

Sometimes your model might function as intended, but it takes an extreme amount of time to complete a run. It might be hard to find what the cause of the slow simulation is. Insight in what *Methods* or events take the most time is given by the *Profiler*. The *Profiler* collects data about run time consumption and generates a report with its findings.

Task:

1. Change the object *Branching* back to its original setting by connecting the *Method Branch* to the *Flow Control* (do this within the *Class Library*).
2. Activate the profiler by pressing *Debugger > Profiler > Activate Profiler* in the *Ribbon.*
3. Reset and run the simulation.
4. Open the report of the *Profiler* by pressing *Debugger > Profiler > Show Profile* in the *Ribbon.* Something like the following screen should appear:



Profile

Tecnomatix Plant Simulation 13 Profiler

Print All | Save | Refresh

Time of creation:	Monday, 29 May, 2017 5:07 PM
Model name:	C:\Users\ ..... \ ..... \Chapter 7.spp
Total elapsed time:	6.060596s
Total elapsed time for method execution:	0.879653s

-->Top 50 Call Cycles

Top 50 Most Relevant Methods

%Global	%Local	Time [s]	Calls	ms/Call	Method
37.2%		0.327s	45272	0.01ms	.Models.ProcessingStation.ProcEnd
34.4%		0.303s	45870	0.01ms	.Models.ProcessingStation.Check
25.1%		0.221s	36138	0.01ms	.Models.Drain.Check
3.2%		0.028s	45272	0.00ms	.Models.ProductionPlan.bestNdest
0.0%		0.000s	4	0.03ms	.Models.CarFactory.Reset
0.0%		0.000s	4	0.01ms	.Models.Branching.Init
0.0%		0.000s	8	0.00ms	.Models.Branching.Reset

Top 50 Call Cycles

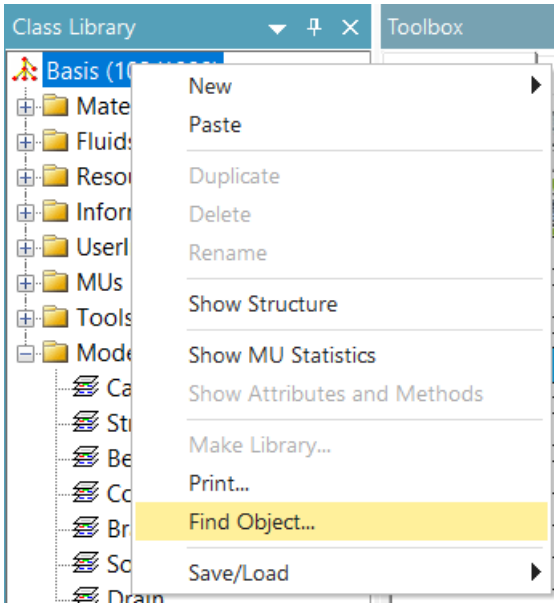
	%Time	Self	Descendants	Called Called+Self Called/Total	Parents Method Children
		0.262s	0.023s	36146	.Models.CarFactory.ProcessingStation1.Machine (SingleProc)
[1]	32.4%	0.262s	0.023s	36146+0	**.Models.CarFactory.ProcessingStation1.ProcEnd**
		0.023s	0.000s	36146/45272	.Models.CarFactory.ProductionPlan.bestNdest
		0.239s	0.000s	36146	.Models.CarFactory.ProcessingStation1.MachineCheck (Buffer)
[2]	27.2%	0.239s	0.000s	36146+0	**.Models.CarFactory.ProcessingStation1.Check**
		0.221s	0.000s	36138	.Models.CarFactory.Drain.CheckPoint (SingleProc)
[3]	25.1%	0.221s	0.000s	36138+0	**.Models.CarFactory.Drain.Check**
		0.065s	0.005s	9126	.Models.CarFactory.ProcessingStation.Machine (SingleProc)
[4]	8.0%	0.065s	0.005s	9126+0	**.Models.CarFactory.ProcessingStation.ProcEnd**
		0.005s	0.000s	9126/45272	.Models.CarFactory.ProductionPlan.bestNdest
		0.064s	0.000s	9724	.Models.CarFactory.ProcessingStation.MachineCheck (Buffer)
[5]	7.2%	0.064s	0.000s	9724+0	**.Models.CarFactory.ProcessingStation.Check**
		0.023s	0.000s	36146	.Models.CarFactory.ProcessingStation1.ProcEnd

In this report you can see the total elapsed time versus the time required for *Method* execution. In our case, *Method* execution only took around 0.88 seconds. Another relevant piece of information is the %global column. This column shows the percentage the specific *Method* took of the total time for *Method* execution.

Did you know?

Animation increases the running time, therefore it is often turned off when running multiple replications or simulations that take a lot of time. Animation can be turned off by pressing in the menu *Home* of the *Ribbon* (left button to turn off *MU* animation and right button to turn off icon animation) or by pressing in the EventController when starting the simulation. Working with large tables also slows down Plant Simulation significantly. So before you decide to store all your data of all runs and replications in one table and update it after every process step, think carefully about:

- What to store in your table.
- When to update your table.
- How many tables you use.



To conclude this section, let us have a look at the *Find Object* function. Although its name might let you believe otherwise, *Find Object* enables you to search for a variety of things: objects, conditions, text within a *User-defined Attribute* of an object, code in a *Method*, and a value within a *List* or *Table*.

With larger models, you might lose overview. You see for example that a certain object does not behave as it is supposed to, but you are not exactly sure which *Methods* refer to the object. You can search for the name of the object in Methods using *Find Object*. The *Find Object* window is opened by right-clicking on the *Basis* in the *Class Library* (root of the tree) and left-clicking *Find Object.* From the drop down, select "Source Code" to look in Methods only, resulting in a list of all Methods and their location mentioning the name of that object. To limit your search to a specific folder only, right-click in the *Class Library* on that specific folder and open the *Find Object* window from there.

Most of the time, you do not want to see every instance of a *Method* but just the class, so check the box next to "Ignore inherited name or text".

Did you know? Bugs can have serious consequences

As mentioned in the introduction of this section, bugs can hide in any kind of software. As harmless as they might seem, bugs can have serious consequences. One of the most famous examples is the millennium bug. In short, the bug came down to the fact that software programmers kept track of the current year with only two digits. So the first day of January in the year 2000 became 01-01-00. Most programs interpreted this date as 01-01-1900, resulting in crashes and errors all over the world. One of the most extreme cases of damage caused by a bug might be the crash of the Ariane 5 spacecraft. Parts of codes that were written for Ariane 4 were reused. Ultimately this made the primary and back-up system crash, resulting in an explosion of the spacecraft. Luckily the Ariane 5 was an unmanned spacecraft.

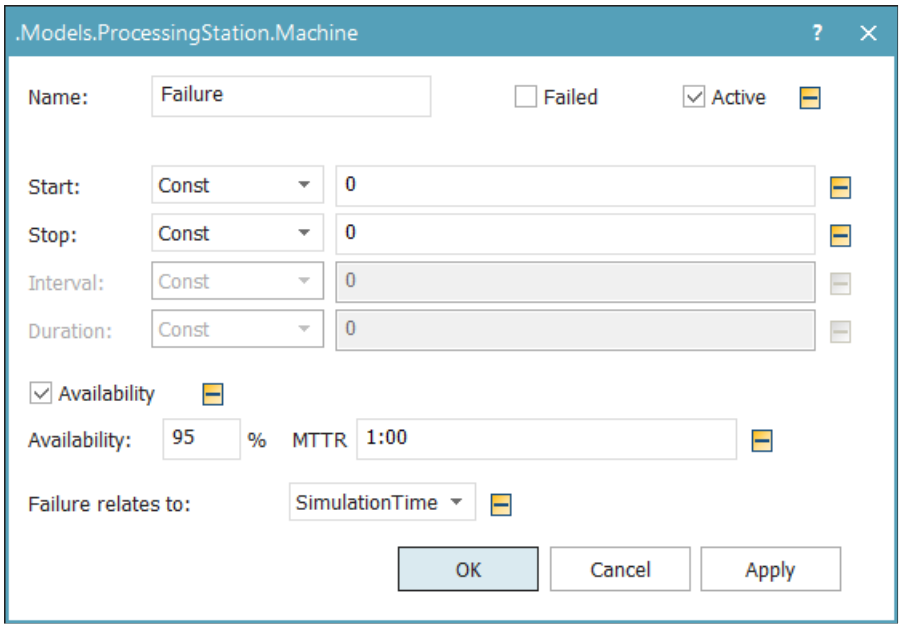
7.7 Machine Failures

Till now we used machines that had an availability of 100%, failing was simply not an option. In a real life situation machines can fail due to a lot of different reasons. Different failures may have different repair times or a different rate of occurrence. You can model this in Plant Simulation.

Task: Model a simple failure

1. Open the *ProcessingStation* model from your *Class Library.*
2. Open the *Machine* object.
3. Click on the tab *Failures.*
4. Click on the button *New…* 

, the following screen should appear:



5. Set *Availibility* to 75% and MTTR to 2 minutes (Mean Time To Repair).
6. Make sure that *Failure relates to* is set to *SimulationTime.*

7. Click OK, make sure that the box in front of your failure in the *Active* column is checked and press OK again.
8. Open the *Method Check*. Besides checking that the *Machine* is available and the *MU* has the right destination, also check whether the *Machine* is not in a failed condition:

```
if Machine.empty and @.dest=StationName and Machine.failed=false
```

9. Reset and run your model at slow to medium speed and open one of the processing stations. Notice how a red dot appears when the *Machine* fails.

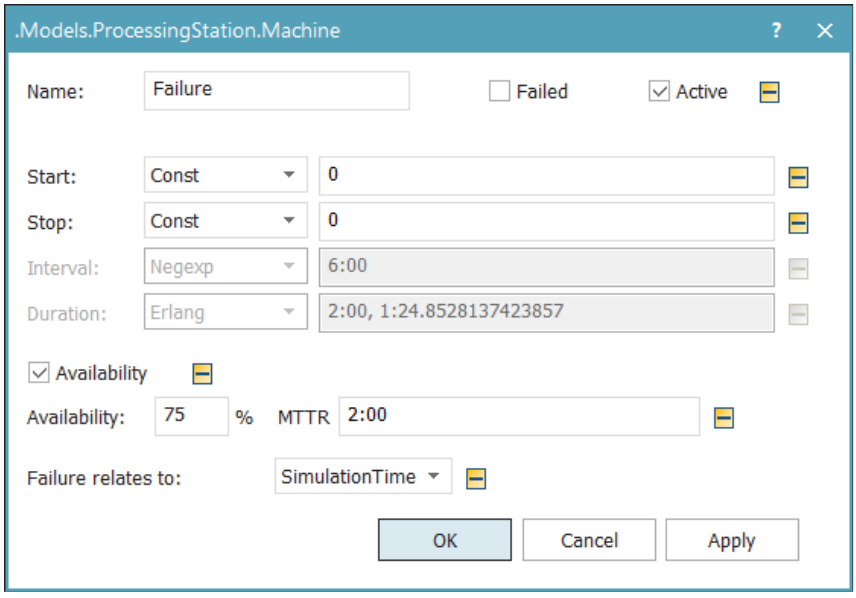
We modelled our failure by setting its availability, which is not a very specific way of defining how many failures occur. We did not specify the distribution of the time between failures and the same goes for the distribution of the MTTR. Plant Simulation actually uses its own settings when you model failures by only giving values for availability and MTTR.

Note: LEDs

When our *Machine* failed during the simulation, a red dot appeared on its icon. Such a dot is called a LED in Plant Simulation and is used to show the state of an object. You might have seen them before on Methods (grey or green dots in our model) or on *MUs* (green or yellow dots). You will learn more about LEDs in the next section.

Task: Check the distribution of a failure

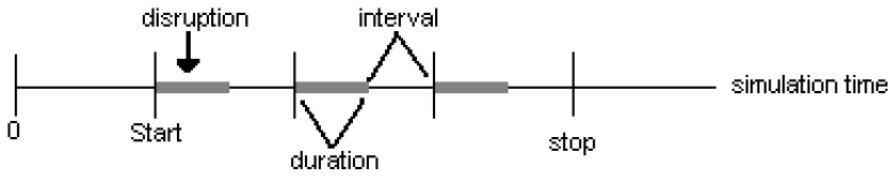
1. Open the *ProcessingStation* from your *Class Library.*
2. Open the *Machine* object.
3. Click on the tab *Failures.*
4. Choose to edit the failure you created in the previous task, the following screen should appear:



Even though we only set the availability to 75% and the MTTR to 2 minutes, Plant Simulation uses an exponential distribution for the interval times, with a beta of 6 minutes, and an Erlang distribution for the duration, with a mu of 2 minutes and a sigma of 1 minute and 24.85 seconds. When you uncheck the box in front of *Availability* in the window of a failure, you can define times and distributions yourself. Remember the *Generator* that you used in Section 5.2, where you activated the *Method InitDay* every day. The entry fields *Start, Stop, Interval* and *Duration* work the same for the *Generator* as for failures. In case of failures, the following holds:

- *Start* is the time that you want the failures to start occurring.
- *Stop* is the time that you want the failures to stop occurring.
- *Interval* is the time between the end of the last failure and the occurrence of a new failure.
- *Duration* is the time a failure takes to be repaired.

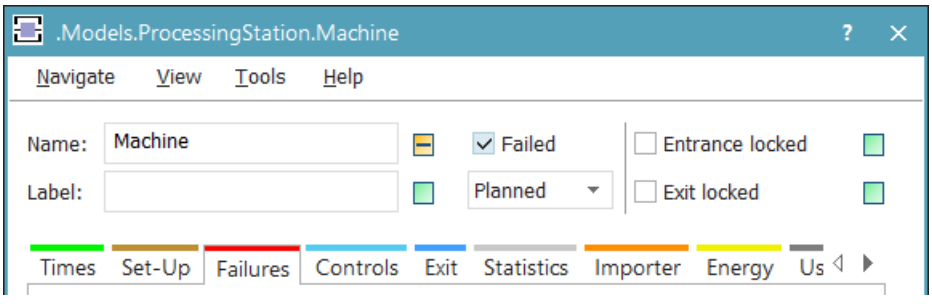
The four entry fields are illustrated in the following figure:



Another important aspect of failures is the time they relate to. We defined the failure to be related to the simulation time, but you can also choose the options operating time and processing time. Operating time is the time when the object (in our case *Machine*) is operational. Operating time is interrupted by pauses and failures. Processing time is the time that the object is actually processing a *MU*. For lines and conveyors, which can also fail, processing time is the time the speed is not zero. It does not matter if the line or conveyor is transporting a *MU* or moving empty.

Note: Manual failures

If you want to see how your model behaves when a certain machine brakes down, you can manually cause it to fail. Open the object you want to see failed and check the box in front of *Failed.* You can also activate or deactivate a failure with *Methods* using "Machine.failed := true".



7.8 State dependent Icons

In Section 3.2 you have learned how to create your own icons and how the icons could depend on the state of the corresponding object. The green patient you drew in Chapter 3 represented an operational *MU* and the yellow patient represented a waiting *MU*. In this paragraph you will learn more about state dependent icons. Recall that objects can be in several different states (Section 3.5). Static objects can be blocked, failed, recovering, operational, paused, setting-up, working, or waiting. Moveable objects can be operational, waiting, failed, or paused. We cannot always force objects to take on every sort of state, as we did in the previous paragraph, forcing *Machine* into the state failed. This is because some states depend on other states or attributes of the object. A *SingleProc* can be failed and paused at the same time, although it cannot be waiting and paused at the same time. Conceptually this last example makes perfect sense, since a machine that is paused is simply not waiting. The fact that an object can take on two states causes difficulties when using state dependent icons, therefore Plant Simulation mostly uses LEDs by default. An object that is in multiple states simply has multiple active LEDs, as shown in the next two examples.

- The *SingleProc Machine* from the previous paragraph that is failed and paused (red and blue LED).
- A *Method* which code is inherited and is being executed (grey and green LED).

You can also create new icons that do not have a related attribute or state.

Task: Create your own state icons

1. Right-click the *MU CarType* in your *Class Library.*
2. Click on *Edit Icons* in the drop down menu.
3. Go to the *Edit* menu in the *Ribbon,* and choose *New* in the *File* menu.
4. Rename this icon *Assembled.*
5. Create another *Icon* and name it *Painted.*
6. Draw two icons that represent an assembled *MU* and a painted *MU.*
7. Apply Changes and close the *Icon Editor.*
8. Open the *ProcessingStation* from your *Class Library*.
9. Open the *Method ProcEnd*.
10. Add the following if-statement below @.move(Advance)

```
if StationName = "Assembly"
    @.CurrIcon := "Assembled"
else
    @.CurrIcon := "Painted"
end
```

11. Apply changes and close the *Method ProcEnd.*
12. Run your model at a slow speed and observe the two instances of *ProcessingStation*.

If everything works as it is supposed to, you will see the icons of the *MUs* change when they move from *Machine* to *Advance*. Note that instead of using @.CurrIcon := ”Painted”, we could also use @.CurrIconNo := 3, where 3 refers to the third available icon for *CarType*.

Note: Icon animation with state dependent icons

It might be the case that as soon as the *MUs* with customised icons move to the *Confluence* objects, their icons change back to the icon *Operational*. This is caused by the use of *State Icons*, that automatically changes the icon upon a state change, e.g., to *Operational* or *Waiting*. In our case this might happen at the *Confluence*. For most *Material Flow* objects, you can disable the standard *State Icons* in the *Icon Editor*. For *MUs* this is not possible. However, a possible work around is to change the *Name* of the state dependent icons:

1. Right-click the *MU CarType* in your *Class Library* and click on *Edit Icons*.
2. Rename the *Operational* icon to *Operational1.*
3. Apply changes and close the *Icon Editor.*
4. Run your model and watch the animation.

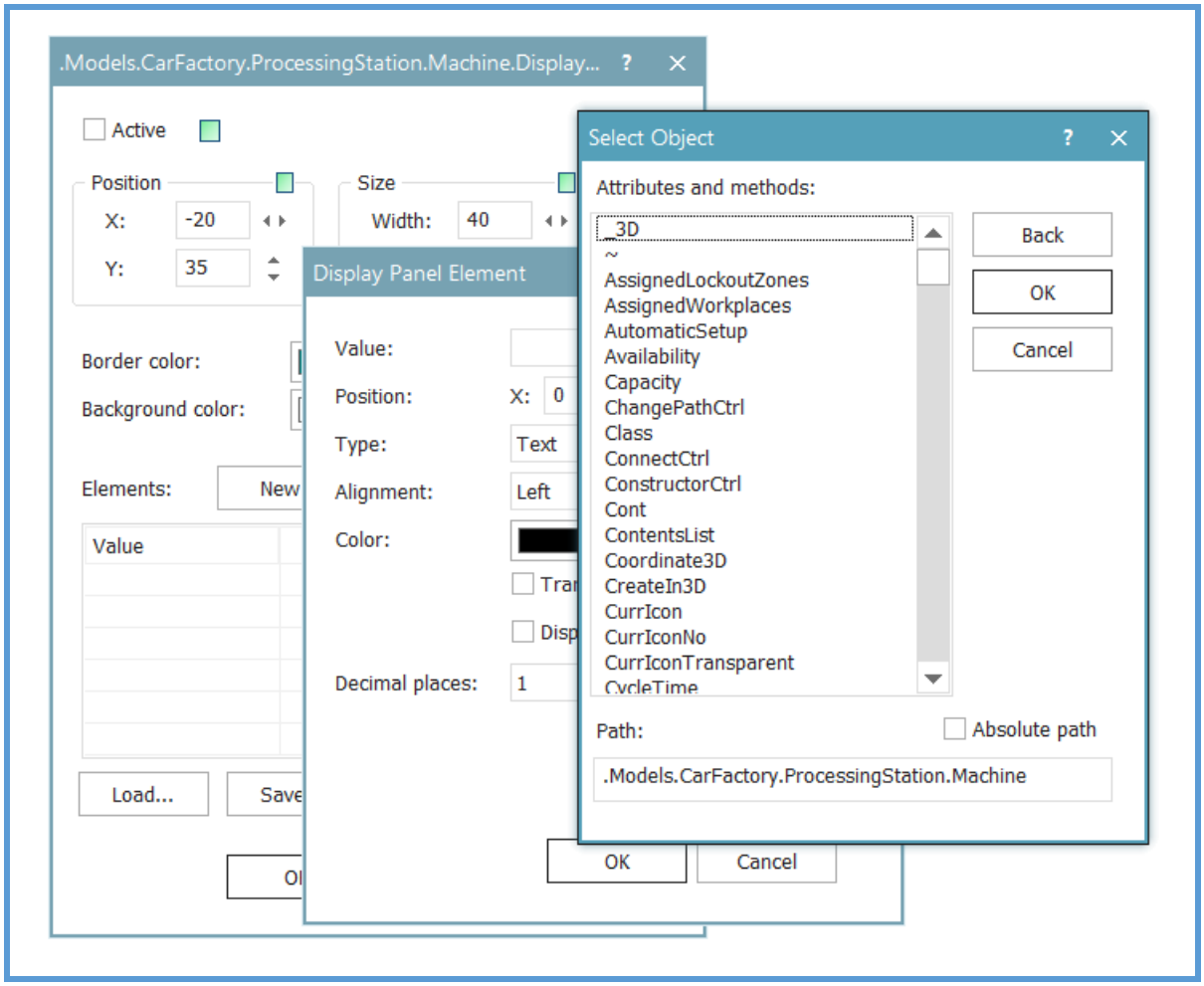
Now when a *MUs* icon changes, it remains this way until the next change, which can either be changing into the icon *Waiting* (since we have not renamed this icon and the use of State Icons is still enabled) or *Painted*. When it takes on the icon *Waiting*, it will not automatically change back to *Assembled* or *Painted* when the waiting is over, we need to define the desired behaviour ourselves when working with customised icons.

Note: Parent or child icons

In the previous task we gave the parent *CarType* two new icons. We could also give the two child objects (*XX* and *XY*) new icons, if we want the icon to depend on the *MU* type for example (remember the GP in Section 3.8). If the Icons of the *XY* and *XX* car have the same name and number as the icons we created for the parent *CarType*, our code in the *Method ProcEnd* would still work while both *MUs* could have different icons.

Did you know? Creating your own LEDs

You can create your own LEDs that depend on attributes you choose. To do this, right-click the object, then choose *Edit Display Panel*. For more information, we refer to the topic of "*Show States with LEDs"* help function of Plant Simulation.



7.9 Assignment B1: Object-Oriented Modelling

In the final model of the car manufacturer, the routing of cars is modelled using the object *ProductionPlan*. However, there are more ways to model the routing control of cars. You are challenged to model the routing control differently. In addition, you extend the model with another machine and incorporate the use of batch processing.

Assignment B1.1: Recreate the final model of the car manufacturer without the use of *ProductionPlan* or any other table to support the routing decisions. Program the routing control instead as simple as possible. When replacing the *ProductionPlan* with code, focus on, e.g., the classes *Branching* or *ProcessingStation* or look for clues in the *User-defined Attributes* of the *SingleProcs* or *MUs* used in the current model. It should be possible to avoid the use of *ProductionPlan* by changing only one *Method*. Verify your new model for its correctness: use the debugging steps presented in this chapter, use visual checks, and compare the throughput times of the changed model with those of the original model. Briefly reflect (say 3 sentences) on the benefit of using the *ProductionPlan* instead of using your routing logic.

The manager of the car manufacturing company considers a redesign of his factory. First, a high tech machine needs to be added to the end of the production process (an additional loop between *Assembly* and *Drain*) that applies a special coating to all cars. The processing time of this machine is 2 minutes; you may assume it has similar failure characteristics as the other two processing stations. Second, the machine for the paint shop needs to be replaced with a new one. The new machine provides a higher quality paintjob, but it requires batches of 4 cars to be processed simultaneously (processing time per car remains the same). As the production hall already needed to be extended to fit in the coating machine, the manager also reserved an extra area to buffer cars that are waiting to be painted. You may assume this area never forms a bottleneck. Next to the new machines, the research and development department has come up with a new type of car called XZ. This new type of car also needs to be assembled and needs to receive the special coating, but it does not need to be painted. Furthermore, the time required to assemble XZ is 3 minutes (hint: set the processing time of this machine to *Formula* instead of *Const*, and refer to a *User-defined Attribute* for the processing time). The arrival frequency of all cars combined in this new situation is 20% lower than originally, and the arrival frequency of XZ is equal to the arrival frequency of XY in the new situation. Hint: Plant Simulation automatically adjusts frequencies to ratios in the *Random Frequency Table*; hence, frequencies do not need to add up to 1 in this *Table*.

Assignment B1.2: Change the model resulting from Assignment B1.1 to provide insight to the manager on how these changes would influence the current production process. Specifically, your task is to:

a. Add the new machine to the production process of the existing product types.
b. Change the paint shop so that it only works with batches of 4 cars (there is no standard option for this, you need to code it yourself). Pay explicit attention to object-oriented modelling, as this is the main topic of this assignment (think about how to change existing classes or add new classes).
c. Add the arrival process of the new car XZ.
d. Adjust the routing control.

Again verify your model using debugging and visual checks. Run the model to determine the throughput times of the three different cars.

Assignment B1.3: Suggest an improvement of the layout of the system in terms of throughput times. Use the model resulting from Assignment B1.2 and implement changes without reducing the total length of the conveyer belt, since this is required as buffer space. Provide a comparison between the throughput times for the different cars within the existing and improved system. Make sure both systems (existing and improved) are placed in the same file using separate *RootFrames*.

Deliverables:

- Your models for the three assignments.
- The throughput times of all car types within the three systems.
- A flowchart for the routing logic implemented at Assignment B1.2. Hint: (1) time does not progress while executing a method and (2) consider the time at which cars receive a new destination.
- A flowchart for the new paint shop (the batching process).
- A report introducing your models and showing the results and flowcharts.

8 Building a Model: lines and workers

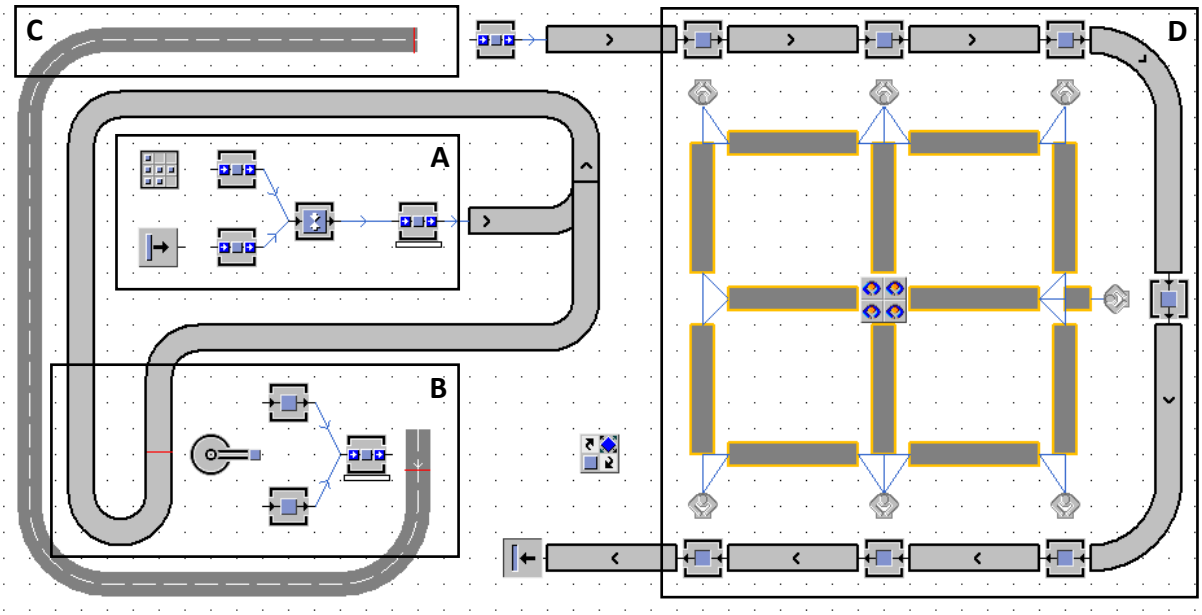
In this chapter we continue modelling the factory, but instead of using the custom routing objects from the previous chapter (straight, bend, confluence, and branching), we use some standard objects of Plant Simulation to represent transportation paths and belts.

In this chapter you will first learn that *MUs* can also contain other *MUs* and how to model this. The second section deals with the objects *Lines* and *Transporters.* To be more specific, we will show you how you can easily model a transport system through *Lines* and *Tracks*, but also the differences and similarities between *Lines* and *Tracks.* Next up are *Workers* and *Worker pools*, which add a new dimension to your model when it comes to situations where humans are involved. Last but not least we will address the topic of experimenting without the use of the *ExperimentManager* (see Chapter 5). This chapter ends with an assignment that will not only test your Plant Simulation skills, but also requires statistical expertise to determine a proper warmup period and number of replications.

The final model of this chapter contains a few typical parts, which you could encounter in any production facility. Normally it would be wise to create separate *Frames* for every part (see Chapter 7), but in order to create a nice 3D model, we will only make use of one *Frame* for the actual station in this chapter. Next to this *Frame*, we will use another *Frame* that functions as a control panel. Each part of our production facility will contain one or more objects of Plant Simulation that we have not explained yet. You will build the following parts:

A. A warehouse, where parts of a product are placed on a pallet.
B. A line that transports pallets from the warehouse to welding stations.
C. A track on which the products are moved by an automated vehicle.
D. An assembly line where workers operate machines.

Your final model in this chapter should look like follows, with the letters referring to the list above:



Subjects dealt within this chapter:

- *MU* types: the *Container* and the *Transporter*
- *Lines*, *Tracks and Sensors*
- *Workers* and *Worker pools*
- *Shift Calendars*
- 3D modelling with standard objects
- Experimenting without the *ExperimentManager*

8.1 Setup of the Warehouse

We will introduce several new objects in the upcoming paragraphs. Some of these objects offer new functionalities, but you might encounter objects that carry out operations you could have modelled already, using a different approach, e.g., with *SingleProcs* and *Methods*. Next to the functionalities of basic objects, they enable you to create 3D simulations without any knowledge of 3D modelling.

Task: Create the RootFrame

1. Create a new model.
2. Rename your *RootFrame*, which is called *Frame* by default, to *ControlPanel*.
3. Create a new *Frame* and rename it *AssemblyStation*.
4. Place an instance of *AssemblyStation* on your *ControlPanel.* Remember: when possible, apply your changes only to the object in your *Class Library*.
5. Add three *Methods* to the *ControlPanel,* it might be convenient to press the Ctrl-Key so you do not have to select the *Method* object three times.
6. Rename the *Methods* to *Reset, Init*, and *CreateObjects*.

Note: Importing an object of the tutorial model

To speed up model building in this chapter (Section 8.1-8.3), you may decide to import the accompanying file "AssemblyStation.obj". Although this saves some time in creating the *Frame AssemblyStation*, you still need to go through all tasks in this chapter (i) to create the *Frame ControlPanel* and other objects, and (ii) to create the necessary connections between the two frames. If you decide to use this prefabricated object, first remove your *Frame AssemblyStation* form the previous task. Right-click *Basis* in the *Class Library*, select *Save/Load*, and *Load Object*. Select the *AssemblyStation* object. In the window that opens, select *Replace All*, to replace all objects in your *Class Library* with the ones from the loaded class. Again, place an instance of *AssemblyStation* on your *ControlPanel*. The error you receive is caused by invalid connections between both frames. These connections need to be made throughout the tasks in this chapter.

Task: Create the MUs

1. Duplicate the *MU Entity* from your *Class Library* and rename the duplicate to *Engine*.
2. Derive five *MUs* from *Engine* and rename them to *XV, XW, XX, XY, and XZ.*

3. Create 16 *User-defined Attributes* to all *Engines* (so to the *MU Engine*) with data type *Time* and the following names:

PTElectricalBattery	WTElectricalBattery
PTPistons	WTPistons
PTCarburetor	WTCarburetor
PTSparkplug	WTSparkplug
PTCooling	WTCooling
PTAirfilter	WTAirfilter
PTTurbo	WTTurbo
TimeWareHouseWelding	WTExit

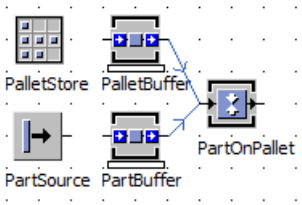
4. Duplicate the Container *MU* from your *Class Library* and rename the duplicate to *Pallet.*

Do not worry if some parts do not yet make sense to you, we will use them later. First let us continue building the *Warehouse*. In the *Warehouse* the *Pallets* will be stored, the parts will be generated from a *Source*, and the parts will be placed on pallets. We will show how you can model the situation where just a limited number of pallets is available, as might be the case in real life manufacturing facilities.

Since some steps are quite basic, we will explain them in less detail. If you find yourself having trouble with objects that have been used before, like the *Source*, look them up in previous chapters to refresh your knowledge. When it comes to specific settings of an object, leave them as they are by default, unless stated otherwise.

Task: Create the Warehouse

1. Add the following objects to the *Frame AssemblyStation*: two *Buffers,* a *Source,* a *Store*, and an *Assembly*.
2. Add the following objects to the *ControlPanel*: a *Method* and a *TableFile*. Rename the *Method* to *NewPart* and the *TableFile* to *SourceTable.*
3. Rename and connect the objects in your *AssemblyStation* as indicated below:



4. Set the Dwell time of the *Buffers* to 0, and the capacity to infinite.
5. Open the *PartSource* and select the tab *Attributes.* At the *Operating mode,* uncheck *Blocking* (we will come back to this in Section 8.4). Next, set *MU selection* to *Percentage* and set *SourceTable* as *Table*. Use ".Models.ControlPanel" as first part of the absolute path (Plant Simulation adds a * before the path). You can also select the browse option (…), check *Absolute path*, and use the *Back* button to navigate to the *SourceTable*. Keep this procedure in mind throughout this chapter when referring to the *ControlPanel* from the *Frame AssemblyStation*.
6. Set the interval of the source to an exponential distribution with a mean of 3 minutes.

7. Check if the *SourceTable* in your *ControlPanel* now has the correct format, add the *MUs* to the table (for example .MUs.XV), and set the portion of XV to 0.30, XW to 0.25, XX to 0.05, XY to 0.15, and XZ to 0.25.
8. Set the *X*- and *Y-Dimension* of *PalletStore* to 100, so that it has a total capacity of 10,000.
9. Enter the following code in the *Method NewPart*:

```
@.move(AssemblyStation.PartBuffer)
if AssemblyStation.PalletStore.NumMU > 0
    AssemblyStation.PalletStore.MUPart(1).move(AssemblyStation.PalletBuffer)
end
```

10. Set the *NewPart Method* as exit control for the *PartSource.*
11. Open *PartOnPallet*, go to the tab *Attributes*, and set Assembly mode to *Attach MUs*. Go to the tab *Times* and set the *Processing time* to a constant of 1 minute (default setting).

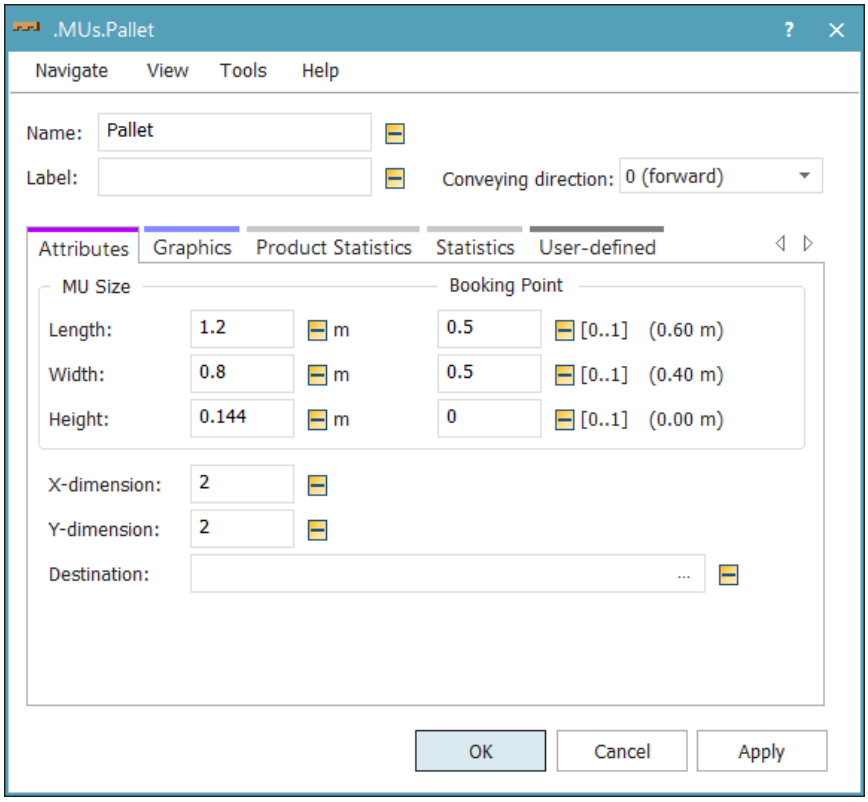
Let us take a look at what we have so far. The *PartSource* creates the right *MUs,* and every time a *MU* tries to leave the *PartSource*, the *Method Newpart* is called. This *Method* moves the specific *MU* to the *PartBuffer* and if there are any *Pallets* left, it also moves a *Pallet,* denoted by muPart(1)*,* to the *PalletBuffer*. When both a *Pallet* and one of the *Engines* arrive at the *PartOnPallet* object, the specific parts are placed on the *Pallet*. There is no successor for the *PartOnPallet* yet, so the *Pallet* with an *Engine* on it will stay in the *PartOnPallet* and block incoming *MUs.*

Note: The Store object

The *Store* object is a passive storage object. Different from the *Buffer* object, a *Store* does not try to push *MUs* to its successor. If you want to move *MUs* from the *Store* to another object, you need to actively pull. Storing *MUs* can simply be done by connecting it to an exit of another object. The *Store* object is just an addition to the model, which we could replace by a *PalletBuffer*.

Before we continue with the model, let us go in more depth on what our *Pallet* exactly is and what it does. We derived our *Pallet* from the *MU* type *Container* from the *Class Library*. The *MU* type *Container* can be used to model any type of "container like" object such as boxes, tables, pallets, types of packaging, etc. One could choose to give *User-defined Attributes* to a *Container,* let it be processed or buffered just like the *MUs* we derived from the *MU* type *Entity.* The most important difference, however, is that the *Containers* have the ability to transport *Entities.*

To transport *Entities,* the *Container* has a certain capacity that depends on the *XDim* and *YDim* attributes, just like the processing capacity of a *ParallelProc.* You can change the *XDim* and *YDim* of the *Container* by double-clicking on the *Container* in your *Class Library.* If you do so, you see the following screen:



Here you can set the *XDim* and *YDim* by changing *X-dimension* and *Y-dimension* respectively, and change the *MU length, MU width, MU height*, and more. Let us continue with our *Warehouse* and put some *Pallets* in our *Store.*

Task: Create objects

1. Add a variable of type integer to your *ControlPanel*.
2. Rename the variable to *NrPallets* and set its value to 10,000.
3. Give the *MU Pallet* in your *Class Library* the *User-defined Attribute PalletNr.* Set the data type of the attribute to integer and its standard value to 0.
4. Enter the following code in the *Method CreateObjects* located on the *ControlPanel*:

```
var l: integer
var t: object

for l := 1 to NrPallets
    -- The following line creates 1 pallet t
    t := .MUs.Pallet.create(root.AssemblyStation.PalletStore)

    -- We can change the attribute of the pallet by reffering to t
    t.PalletNr := l
next
```

5. Enter code in the *Method Reset* so that it deletes all *MUs* every time it is triggered.
6. Make sure the *Method Init* calls the *CreateObjects Method*.
7. For testing purposes, add a drain to your *Warehouse*, connect it to the *PartOnPallet*, and test your model.

After the creation of 10,000 engines, the *Store* does not have any *Pallets* left and therefore the engines get stuck in the *PartBuffer* and the *PartSource.* In the end of our process, we will have to return the *Pallets* to the store, otherwise we will run out of *Pallets* eventually. We will make sure that *Pallets* return to the store when we work on the final processing steps of our production facility.

Note: How to handle MUs on MUs

We used the *PartOnPallet* to place a *MU* of type *Engine* on the *Pallet.* We could also do this by using a *Method*. The command *Move* also works for moving from or to a *MU* type *Container. Also* attributes like *Cont* and *NumMu* function the same as before. Further, if you now need to refer to an attribute of a *MU* that is placed in a *Container*, you need to be careful with the path you use. For example, if you want to access a *User-defined Attribute ProcTime* of a *MU* that is placed on a *Pallet*, then you have to use the path: *Pallet.cont.ProcTime*. If a *Pallet* contains more than one *MU*, you can use *MuPart(x)*, where *x* stands for the position of the specific *MU* on the *Pallet.*

As you can see in the picture of the overall layout, shown in the introduction of this chapter, the *MUs* will go through a number of different processing stations. The different engines have different processing times for these stations. There are different ways to model this, but in this case we will use the *User-defined Attributes* we created.

Task: Assign processing times to the MUs

1. Create a new *TableFile* on your *ControlPanel* and name it *ProcTimes*.
2. Set column and row indexes to active.
3. Set the number of rows to 7 and the number of columns to 5.
4. Set the data type of the 5 columns to integer.
5. Enter the following:

	string 0	integer 1	integer 2	integer 3	integer 4	integer 5
string		XX	XY	XZ	XV	XW
1	ElectricalBattery	1	1	0	0	1
2	Pistons	0	1	1	1	0
3	Carburetor	0	1	1	1	0
4	SparkPlug	0	1	1	1	0
5	Cooling	1	1	1	1	0
6	AirFilter	1	1	1	1	0
7	Turbo	1	1	0	1	1

6. Extend your *NewPart Method* by adding the missing parts from the picture below (see also code in the Appendix) and try to understand it:

```
var i, Snr: integer

@.move(AssemblyStation.PartBuffer)
if AssemblyStation.PalletStore.NumMU > 0
    AssemblyStation.PalletStore.MUPart(1).move(AssemblyStation.PalletBuffer)
end

Snr := 0

for i := 1 to ProcTimes.indexYDim
    if ProcTimes[@.Name, i] > 0
        Snr += 1
        @.setAttribute("PT" + ProcTimes[0, i],
            (60 * z_lognorm(Snr, 1 * ProcTimes[@.Name, i],
                0.5 * ProcTimes[@.Name, i])))
    else
        @.setAttribute("PT" + ProcTimes[0, i], 0)
    end
next
```

For every *Engine*, the attribute names that start with PT contain values of the processing times. The attribute names starting with WT will be used to measure the waiting time at certain stations, but this will be done after we created the specific stations. In our *Method NewPart*, the loop makes sure a time is set for every of the seven stations, and we use @.Name to make sure we are looking in the right column for the specific *Engine.* The number from the *TableFile ProcTimes* represents the mean for the lognormal distribution, with a standard deviation of half the mean value as shown in the table. The variable Snr stands for stream number and makes sure that every station has its own random number stream.

Did you know?

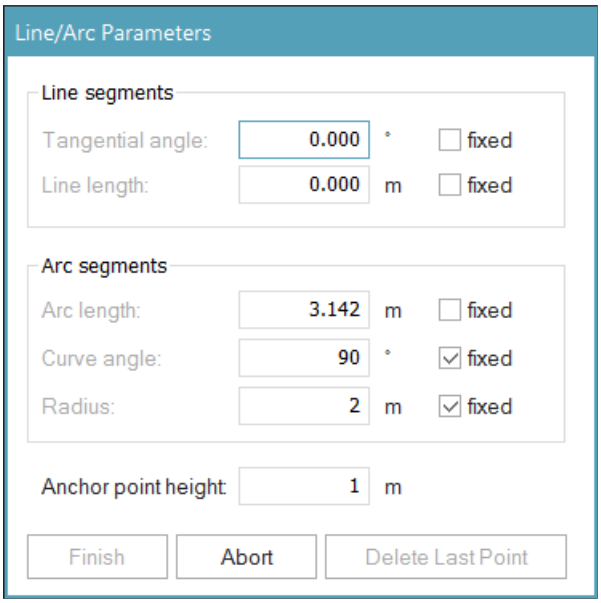
We discussed the different data types in Section 3.10. There are all sorts of transformation options that you can use in *Methods*, e.g., obj_to_str. However, there is no such thing as obj_to_str for attributes, but we can use getAttribute and setAttribute as we did in the previous task. The getAttribute() statement returns the value of the attribute you ask for, by naming it between the brackets. For example @.getAttribute("WTSparkPlug") returns the value of the *WTSparkPlug* attribute of the *MU* that activated the *Method*. Logically, setAttribute sets the defined attribute to a given value. The setAttribute statement requires two input parameters, the first one is the name of the attribute, the second is the value that you want to assign to that attribute.

8.2 Line Objects

In the previous chapter we created our own line by making use of *Frames* and *SingleProcs*. Plant Simulation also has a standard object to model a *Line*, namely the *Line* and *Track* objects. The *Line* and *Track* have a lot of similarities, the biggest difference is that on a *Line, MUs* can move on their own while on a track they are usually placed on a *Transporter*. This also brings us to the biggest difference between *MUs* of the type *Container* and *MUs* of the type *Transporter:* a *Transporter* moves on its own whereas a *Container* and *Entity* move forward through the forward motion of a *Line*. We will start by creating a simple *Line* for which, as said, no transporter is needed.

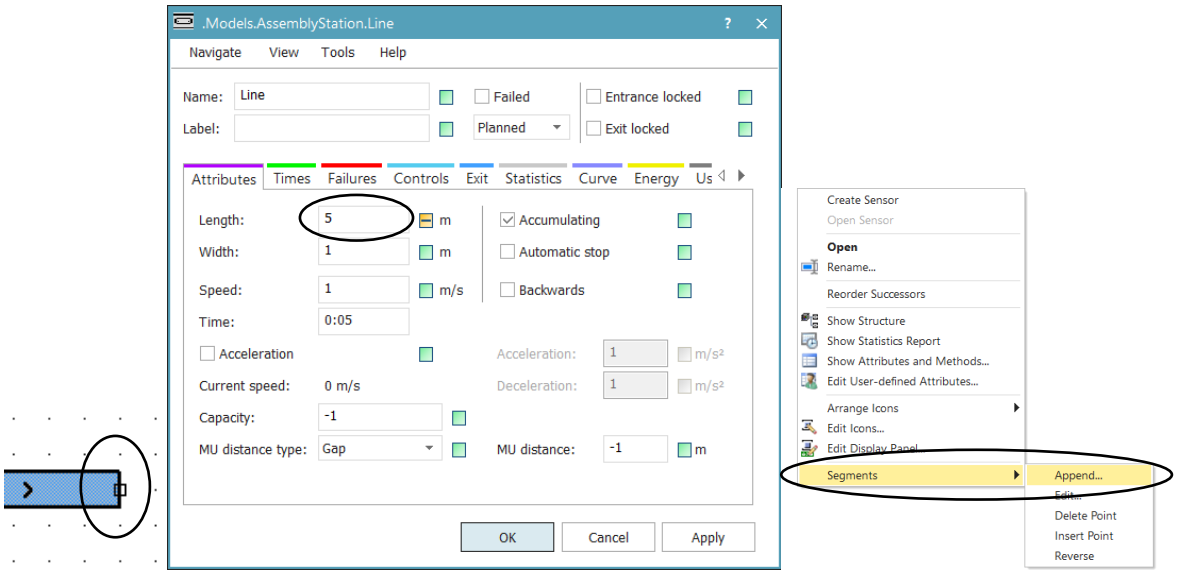
Task: Create a Line

1. Add a *Buffer* to the *Frame AssemblyStation*.
2. Rename the *Buffer* to *LineBuffer*, and connect the entrance of *LineBuffer* to the exit of *PartOnPallet*.
3. Set the *Dwell time* of the *LineBuffer* to 0, and set the capacity to 4.
4. Now add a *Line* by clicking on the *Line* object in your *Toolbox*, the following screen should appear:



5. Create a straight horizontal line of around 5 meters, left-click to start placing the *Line*, left-click to confirm the end point and then follow up by a right-click to stop making *Line* segments and automatically close this screen.

Congratulations you have created your first *Line*, just like that. If you wanted to continue with the *Line*, you could have simply continued by another left-click or by appending other segments.

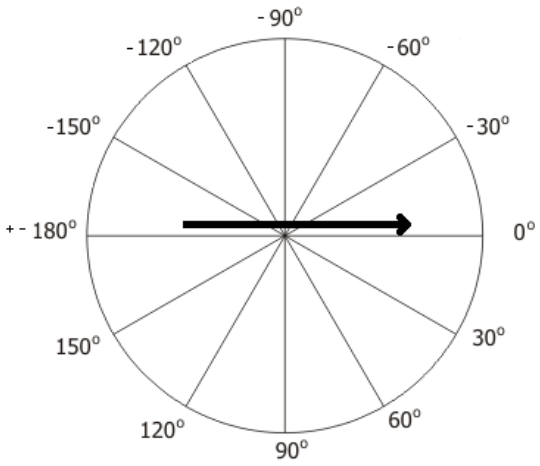


The *Line* you just created can be edited by:

- Simply dragging the black squares to another location.
- Double-clicking the *Line* object and setting a new value for length.
- Right-clicking the *Line* object and choosing the option *Segments > Append*.

The last option is the only one that offers the same range of options as creating a new *Line*. Let us explore the "*Append"* option further. In the screen that is titled *"Line/Arc Parameters",* see previous task, you can either check or uncheck boxes that say *fixed*. If you do not check these boxes, the values in the corresponding fields depend on your mouse movement. If you do check these boxes and have entered a specific value, whatever you do with your mouse, the part of the line you draw will have the specific value. The options you have for straight *Line* segments are:

- *Tangential angle*: the angle relative to the previous *Line* segment. Ranges from -180 to 180. For the first *Line* segment it represents the angle with a horizontal line from left to right. In the figure, the arrow represents the direction of the previous *Line* segment.
- *Line length*: simply the length of your straight *Line* segment.



To create an arc, do the same as with a straight *Line* (left-click the *Line* object in your *Toolbox,* left-click for a starting point, and left-click for an end point), but now hold down the Ctrl-Key after you have chosen a starting point. Instead of straight lines you now draw arcs, until you release the Ctrl-Key.

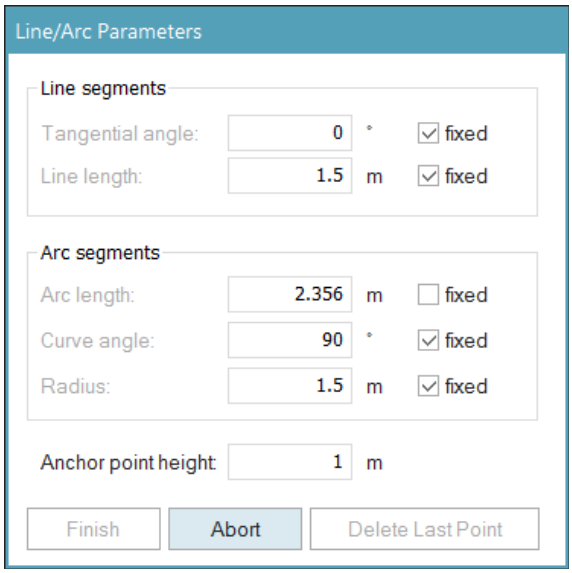
The options you have for arcs are:

- *Arc length*: the absolute length of the *Line* segment.
- *Curve angle*: the angle in degrees as seen from the centre of the circle the arc is put on.
- *Radius*: the radius of the arc, as if it was a whole circle.

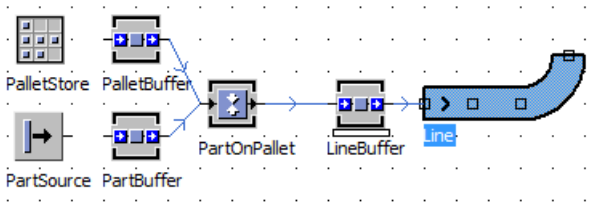
When you enter a value for the arc length, Plant Simulation automatically adjusts the *Radius* or the *Center angle* and vice versa, since they depend on each other. You can always set two out of three. By creative use of the entry fields and the control button, you can easily create a complex *Lines* consisting of various segments of straight segments and arcs.

Task: Create a Line

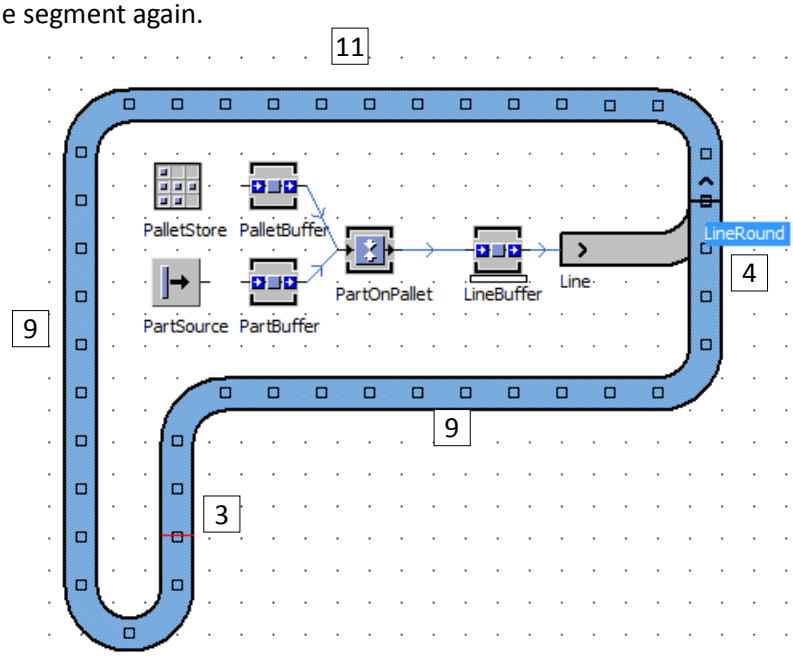
1. Delete the *Line* segment that you have created in the previous task.
2. Delete the *Drain* that you have created to test the *PartOnPallet* Machine.
3. Click on the *Line* object in the *Toolbox*.
4. Set the following fixed values (the non-fixed value will be adjusted in automatically):



5. Create the *Line* shown in the picture below, using the Ctrl-Key. The line contains two straight *Line* segments and one arc segment, as the black squares indicate.



6. Create another *Line* as shown in the picture below, using the same fixed values as the previous *Line*, only now start with a *Tangential angle* of 90 degrees and set it back to 0 after the first segment. The number of straight line segments are shown in the figure (numbers within the squares) and is also indicated by the yellow squares. Note that it might be the case that after each arc segment, you need to define the length of the straight line segment again.



7. Rename your second line to *LineRound*.
8. Connect *Line* with *LineRound*.
9. Connect the start and end of *LineRound* with each other.
10. Move *LineRound* so that its end overlaps with the end of *Line*.
11. Reset and run your model at medium speed and stop when no more *MUs* can get on the *LineRound*.

The number of *MUs* that fit on *LineRound* depends on the length of the *Line* and the length of the *MU Pallet*. The length of the *Pallet* can be found by double-clicking on it or pressing F8 (attribute length). If you do this, you will see that its length is 1.2 meter. To check the length of *LineRound,* you can also double-click it, or press F8 (attribute *Length*). Since *LineRound* is *68.14* meters long, it can hold a maximum of *56 Pallets*.

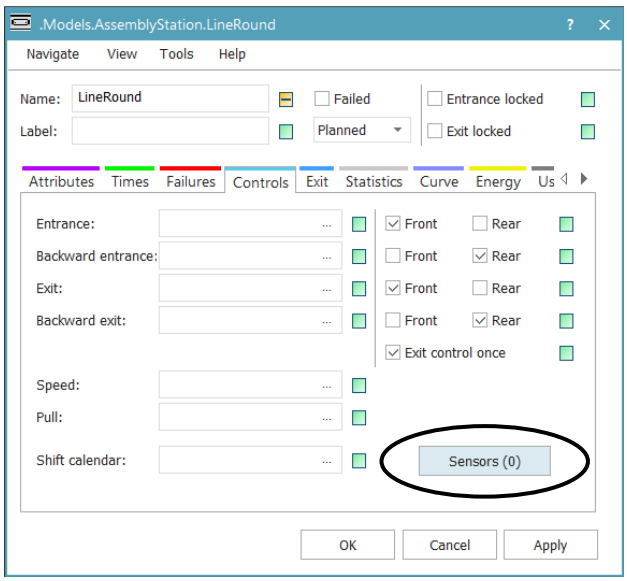
Note: Moving objects

When moving objects by mouse, the objects normally stick to the grid. If you want to move an object per pixel, which is the smallest amount possible in Plant Simulation, use the arrow keys of your keyboard.

The *LineRound* now functions almost identical to the conveyors at the airport, the difference is that at our *Line* there are no people to pick-up their luggage. Let us assume we want to model a machine that picks up the *MUs* from the *LineRound*. In the following task we will introduce the standard object that moves the *MUs* when triggered. Plant Simulation has the option to put in a sensor somewhere along a *Line* or *Track.* A sensor is activated when a *MU* passes it and then it triggers a *Method*.

Task: Create a sensor on your Line

1. Add a new *Method* on your *ControlPanel* and name it *SensorCheckLine*.
2. Open *LineRound*, select the tab *Controls*, and click on *Sensors(0)*.



3. Create a new sensor. For the position choose *Length* if it is not already selected, and set the length to 42.5 meters.
4. Select the new *Method SensorCheckLine* as control. The *LineRound* now contains a red bar as already shown in the figure from the previous task.
5. Click OK to apply changes and close the window.
6. Add a *PickAndPlace* from the *Material Flow* tab of your *Toolbox* (or the *MaterialFlow* folder in your *Class Library*) next to the red line of the *Sensor* on the *LineRound*.
7. Add the following command to the *Method SensorCheckLine*:

```
param SensorID: integer, Front: boolean

if (@.MUPart.Name = "XX" or @.MUPart.Name = "XZ") and
    AssemblyStation.WeldStation1.Full = false and
    AssemblyStation.PickAndPlace.Empty
    @.Move(AssemblyStation.PickAndPlace)
elseif (@.MUPart.Name = "XV" or @.MUPart.Name = "XW" or @.MUPart.Name = "XY") and
    AssemblyStation.WeldStation2.Full = false and
    AssemblyStation.PickAndPlace.Empty
    @.Move(AssemblyStation.PickAndPlace)
end
```

Note: Sensors on Length-oriented objects

Sensors can be placed on all *Length-oriented* objects, such as *Lines, Tracks*, and *TwoLaneTracks*. In principle, *Sensors* work the same on every of those objects. You can choose to activate a sensor at the front or the rear of a passing *MU*, and whether you want it to be always activated or only when the sensor is the destination of the *MU*.

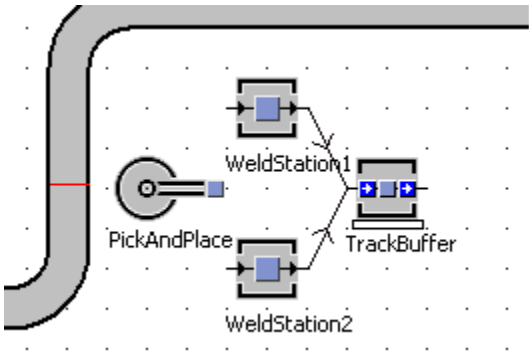
Did you know?

A single *Length-oriented* object can have multiple sensors. You can create a *Method* for every sensor, or use one *Method* for all sensors. When you set a *Method* as control for a sensor, it will be formatted such that it will work as a function with two input parameters: *SensorID* and *Front*. The *SensorID* is an integer type variable, and is set to the *SensorID* number of the sensor that calls the *Method.* Using this variable, you can write code that provides specific actions when a certain sensor is triggered by a *MU.* The *Front* variable is of type Boolean, and is *true* when the sensor should be triggered when the front of the *MU* passes the sensor, and vice versa.

The *PickAndPlace* object that is added does not yet function as we intend. After we move the *MU* to the *PickAndPlace*, it needs to know what to do with the *MU*. The *PickAndPlace* needs a *Method* to determine the *Target* of a *MU* it has picked up. We start the next task with creating two more *SingleProcs* that are the possible destinations of our *MUs*, after they are picked up by the *PickAndPlace*.

Task: Completing the PickAndPlace

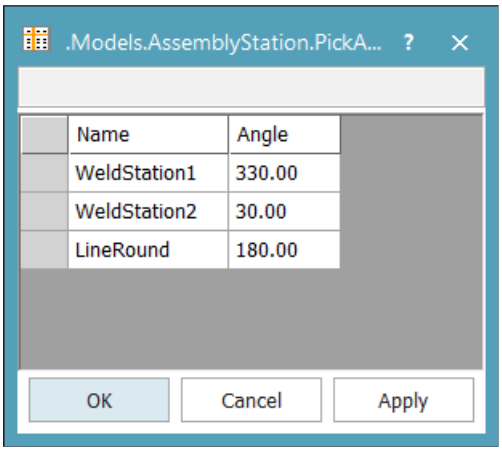
1. Create two new *SingleProcs* and one new *Buffer* in your *Frame AssemblyStation*.
2. Set the Dwell time of the *Buffer* to 0, and set its capacity to 4.
3. Place and name them accordingly:



4. Set the *Processing time* of *WeldStation1* to *Gamma(100, 1.5)* and of *WeldStation*2 to *Gamma(80, 1.25)*.
5. Create a new *Method* on the *ControlPanel* and name it *PickAndPlaceDest.*
6. Set the new *Method* as *Target* for your *PickAndPlace*, *Target* can be found under the tab *Controls*.
7. Enter the following code to your *Method*:

```
if @.MuPart.Name="XX" or @.MuPart.Name="XZ"
    AssemblyStation.PickAndPlace.SetDestination(AssemblyStation.WeldStation1)
else
    AssemblyStation.PickAndPlace.SetDestination(AssemblyStation.WeldStation2)
end
```

8. Double-click the *PickAndPlace* and click the box next to the *Angles Table* button to turn off inheritance.
9. Click the *Angles Table* button and enter the following:



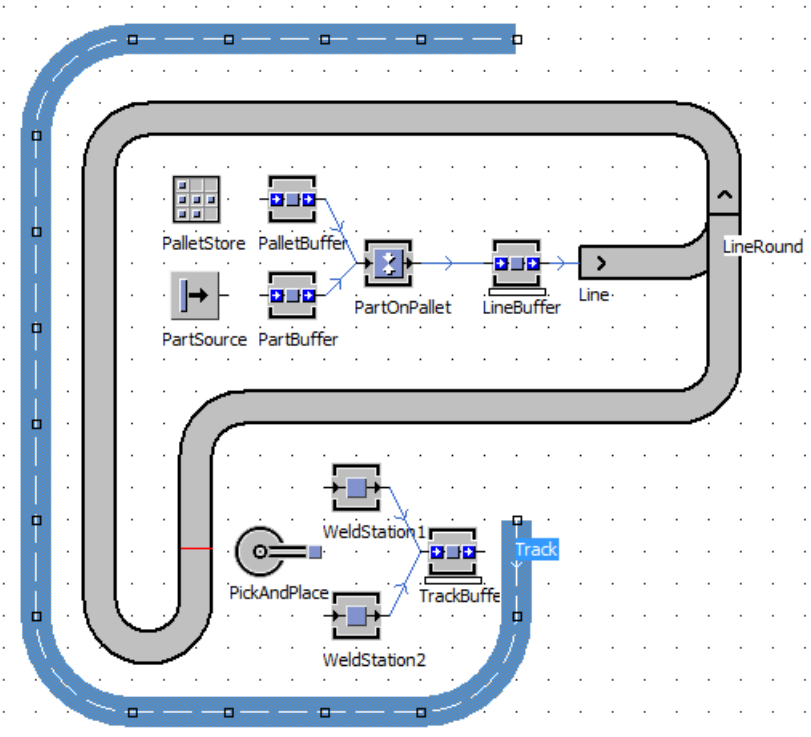
10. Set the *Time factor* to 5.
11. Press OK, reset and finally run your model until both *Weldstation1* and *Weldstation2* are full.

12. For a more realistic animation, check the box *Real time* in the EventController. This function enables the simulation to run equally fast at all times instead of jumping from event to event. The time behind *Real time* states how much faster the simulation should preferably run (if allowed by the required computation times) then the real time. For animation purposes, fill in a value of, e.g., 25. Do not forget to turn off this functionality when performing experiments to speed up the simulation time.

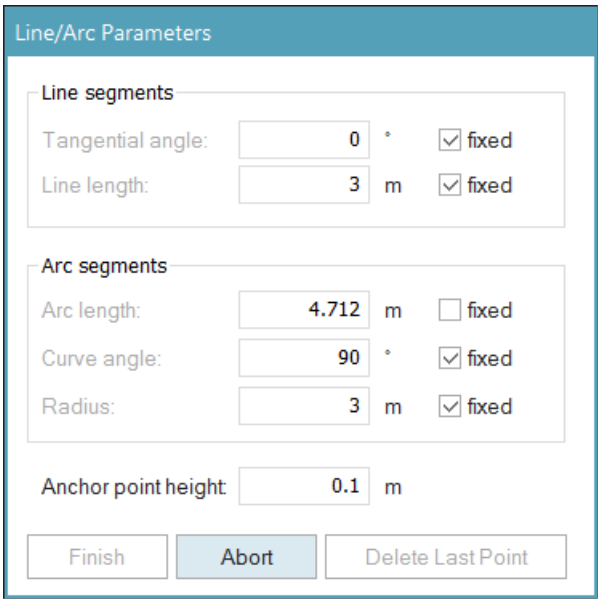
The *PickAndPlace* we used could also be modelled with a *SingleProc* and a *Method*, but the *PickAndPlace* has a stronger visual appeal in both 2D and 3D. After the engines passed the welding stations, they travel across our production facility on an automated vehicle that can carry one *Pallet* at a time. The automated vehicle needs a track to move on.

Task: Create a Track

1. Create a *Track* around the objects you previously constructed using the following layout and using the settings as mentioned in the next step.



2. The *Track* starts with a straight segment with a *Tangential angle* of 90 degrees, all the other segments have a *Tangential angle* of 0 degrees, see the following figure:



3. Create a *Method* on the *ControlPanel* and call it *SensorCheckTrack*.
4. Create a *Buffer* next to the end of the *Track*, rename it to *BufferAssemblyLine,* set the dwell time to 0, and set its capacity to infinity.
5. Create two sensors, one at 1.6 meters from the beginning and one at 53 meters and set your *Method SensorCheckTrack* as control for both of them.
6. Enter the following code to *SensorCheckTrack* (see also code in the Appendix) and try to understand it:

```
param SensorID: integer, Front: boolean

if SensorID = 1
    if Assemblystation.TrackBuffer.Empty
        @.Backwards := false
        @.Stopped := true
    else
        Assemblystation.Trackbuffer.MUPart(1).Move(@)
        @.Backwards := false
    end
elseif SensorID = 2
    if @.Empty
        @.Backwards := true
    else
        @.cont.cont.TimeWarehouseWelding := EventController.SimTime -
            @.cont.cont.CreationTime
        @.cont.cont.WTElectricalBattery := EventController.SimTime
        @.cont.Move(Assemblystation.BufferAssemblyLine)
        @.Backwards := true
    end
end
```

The *Track* itself is now ready, but there are two things missing to let it function properly. The first thing missing is a vehicle to move on the *Track*. On the *Track,* a *MU* of the type *Transporter* can move by itself. In the *SensorCheckTrack*, the @ that triggers the *Sensor* is a *MU* of the type *Transporter*. By changing the attributes *stopped* and *backwards* you can control the movement of the *MU*. As with the other *MUs* used so far, a *Transporter* is created during a run and not modelled before the simulation starts. Finally, we need a trigger to re-activate the movement of the *Transporter*.

Task: Create a Cart

1. Duplicate the *Transporter MU*.
2. Rename the duplicate to *Cart*.
3. Add the following line to the *Method CreateObjects*:
   .MUs.Cart.create(root.AssemblyStation.Track, 1.6)
4. Create a *Method* on your *ControlPanel* and rename it to *ExitTrackBuffer*.
5. Enter the following code:

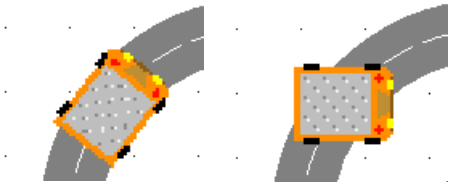
```
if AssemblyStation.Track.MU.Stopped
    ?.MUPart(1).Move(AssemblyStation.Track.MU)
    AssemblyStation.Track.MU.Stopped := false
end
```

6. Set the *Method ExitTrackBuffer* as exit control for your *TrackBuffer.*
7. Reset and run your model to make sure the *Track* and *Cart* function properly.

Now every time we call the *Method Init*, the *Method CreateObjects* is called and a *Cart* is created on the *Track* at the 1.6 meter point. Keep in mind that if you choose to *create* a *MU* this way, you have to choose a point that can hold the total length of the chosen *MU*. If you use a *MU* of 2 meter long at the 1.6 meter point, it will not be created.

Did you know?

If you do not want your *MUs* to rotate according to the direction of your *Length-oriented* object, you can turn this off in the menu of the corresponding *Track* or *Line*.



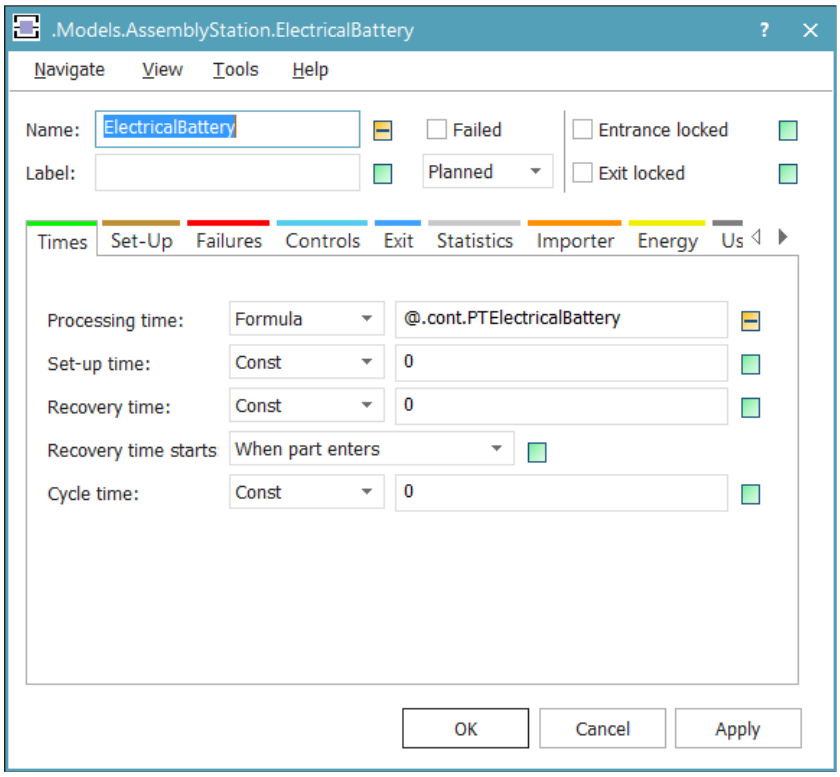
8.3 The Engine Assembly Line

In this section we are going to build the final part of our factory, namely the assembly line for the various engines the factory produces. The assembly line consists of a transportation belt (the line), various workplaces along the line where various parts for the engines are assembled, and employees (*Workers*) that work at these workplaces. First, we create the line and the workplaces.

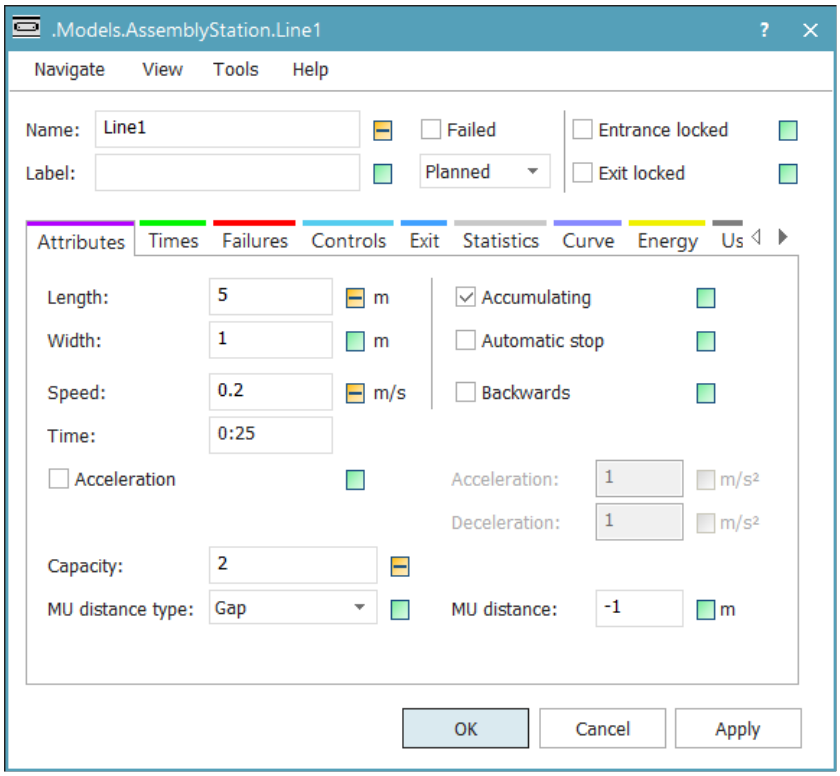
Task: Start Building the Assembly Line

1. Insert 7 *SingleProcs* on the *AssemblyStation* and give them the following names:
   - ElectricalBattery
   - Pistons
   - Carburetor
   - Sparkplug
   - Cooling
   - Airfilter
   - Turbo

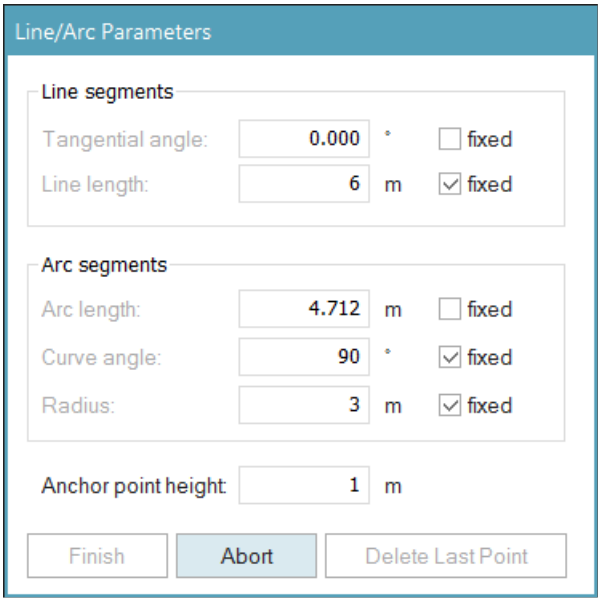
   Note that these stations will have lot of the same properties and attributes in common. If you would like to work efficiently, you could make use of inheritance.
2. Set the *Processing time* of each *SingleProc* as *Formula* and use the relevant attribute of the *MU* as input. For the *ElectricalBattery*, it would be as follows:



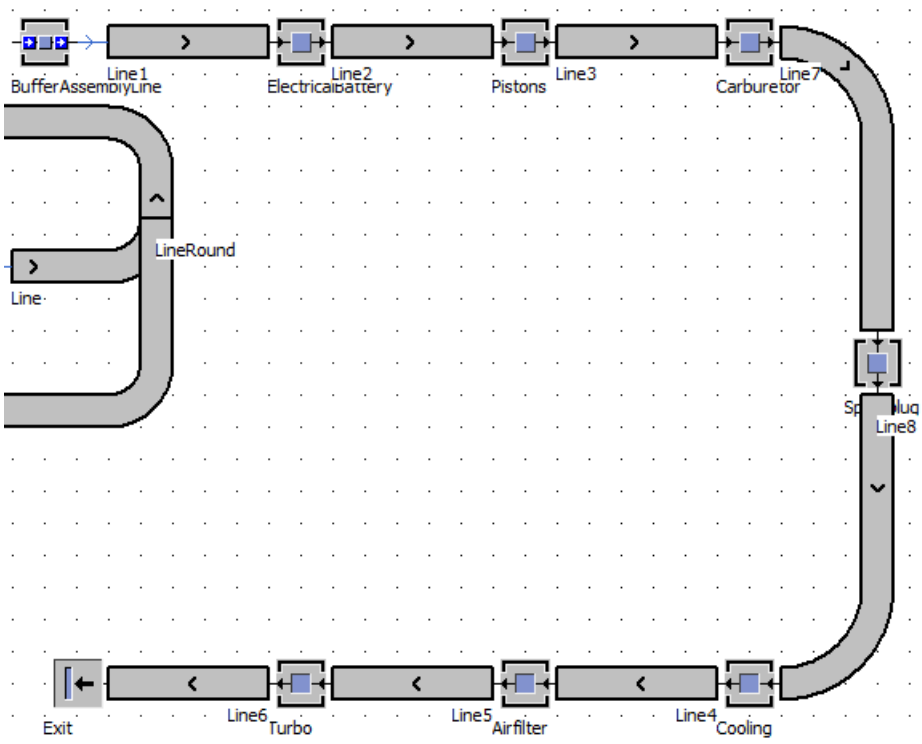
3. Insert a *Drain* object and name it *Exit*.
4. We will now connect the newly placed *SingleProcs* by using *Lines*.
5. Create 6 separate straight *Lines*, which have the following specifications: *Length = 5m, Speed = 0.2 m/s*, and *Capacity = 2*.



6. Create two *Lines* that are a combination of one curved and one straight segment. Click the *Line* symbol from the *Toolbar* and give it the following specifications:



7. While working with the Ctrl-Key, create the two combined lines as shown in the following figure of the assembly line:



Note that the lines should be connected to the *SingleProcs*, the first line should also be connected to *BufferAssemblyLine*, and the last line should be connected to *Exit*. Remind that you can rotate physical objects using Ctrl-T.

8. Set the speed of the two combined lines to 0.2 m/s and the capacity to 2.

We have now linked the basic foundations of our assembly line. The next step is creating the workplaces and the work environment for the workers who are going to assemble the engines. Therefore, we introduce the following worker related objects.

Worker. The object *Worker* represents a working person who performs jobs on a *Workplace*. You can set various *Worker*-related settings, e.g., his specialties (type of jobs he can perform) and capacity (how many jobs he can perform).

Workplace. The object *Workplace* is the actual place at a station where the *Worker* performs his job. It is possible to assign the *Workplace* to various material flow objects, such as *SingleProcs*. The *Worker* stays at the *Workplace* while it performs a job, and at most one *Worker* can stay at the *Workplace* at any time.

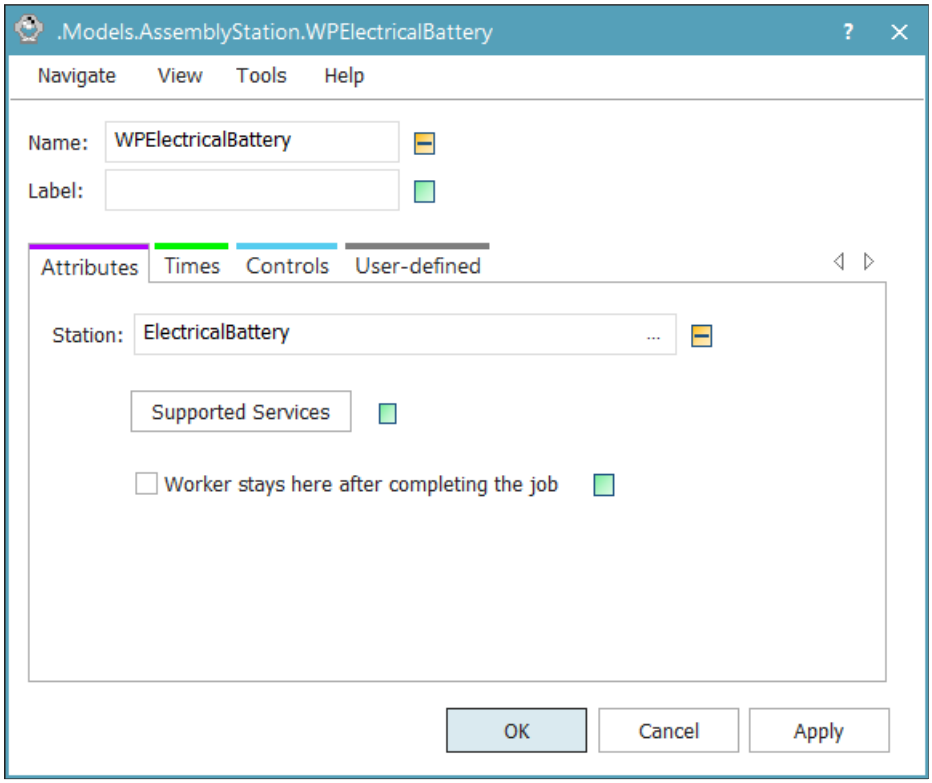
WorkerPool. The *Workers* are created in the object *WorkerPool* and they stay in this object when there no jobs to perform.

Footpath. On the object *Footpath*, the workers move between the *Workplace* and the *WorkerPool*. The time required for movement (e.g., walking between machines) is defined by the length of the *Footpath* as well as the *Speed* of the *Worker*.

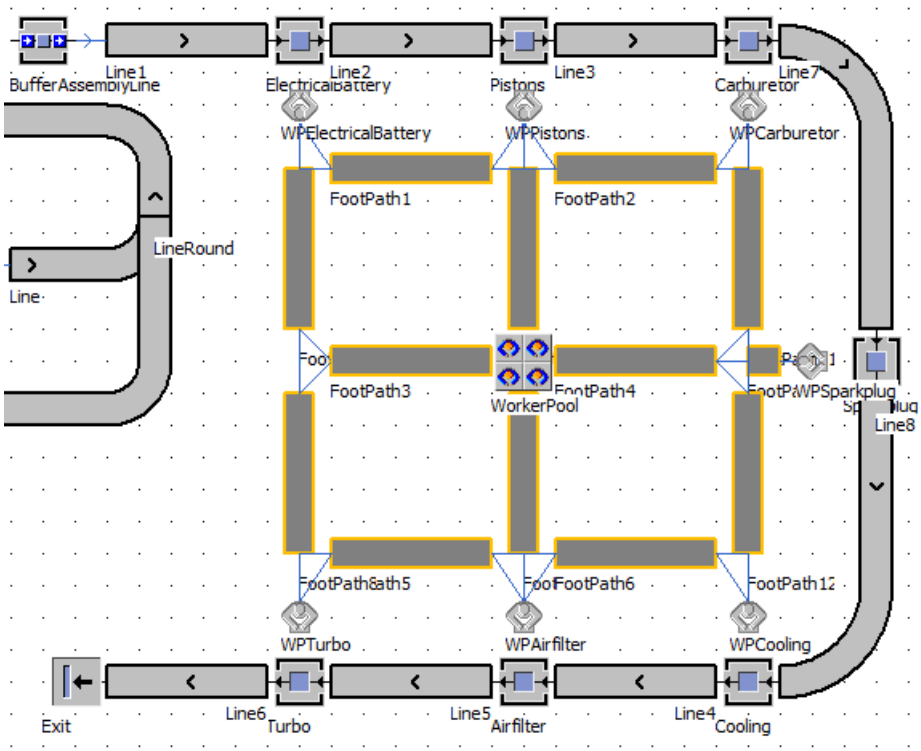
Broker. The object *Broker* is the intermediator between the objects that demand a service and the objects that supply that service. This object allocates the *Workers* to the various *Workplaces*, depending on their specialties, this allocation is done based on user defined rules and does not consists of any form of optimisation.

Task: Create the Work Environment

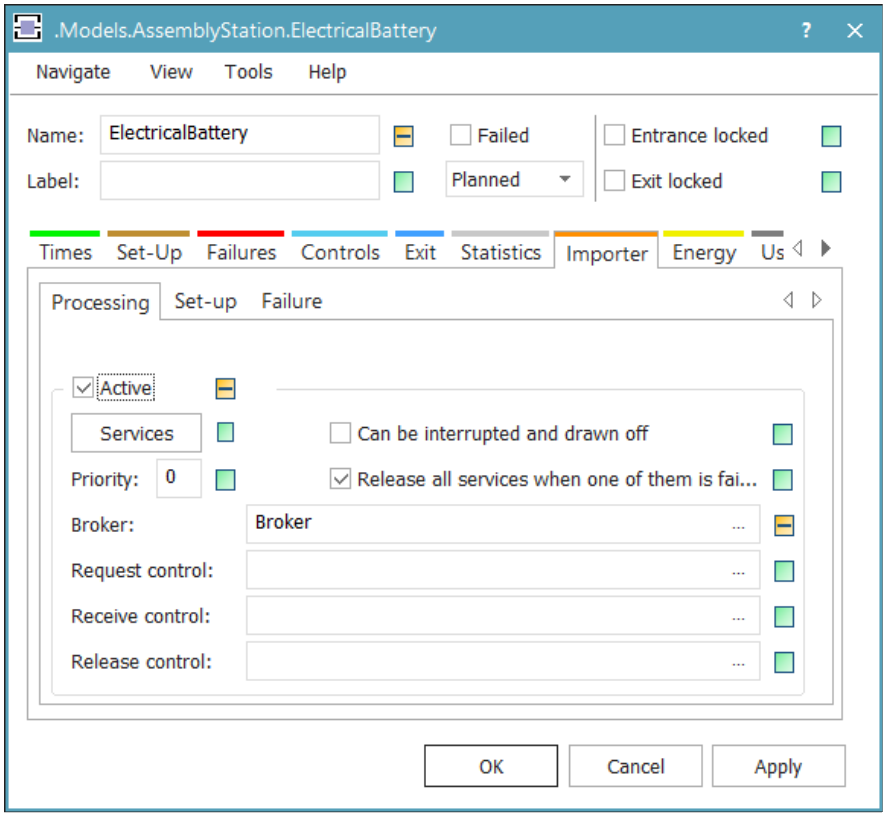
1. Insert a *Workplace* object at each *SingleProc* (7 in total).
2. Name them respectively (in accordance to the nearest located *SingleProc*): *WPElectricalBattery*, *WPPistons*, *WPCarburetor*, *WPSparkplug*, *WPCooling*, *WPAirfilter*, and *WPTurbo*.
3. Connect them to their corresponding station by double-clicking on the object *Workplace* and under the tab *Attributes* selecting the right station (or drag the *Workplace* on top of the corresponding *SingleProc*).



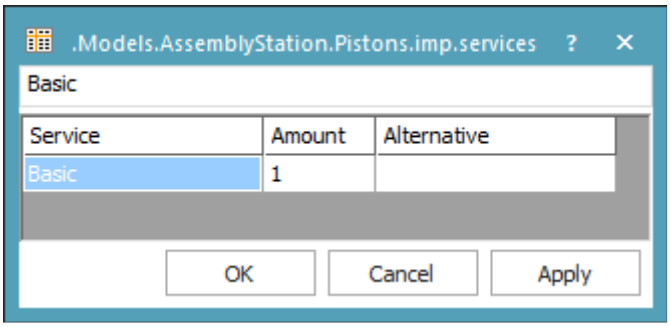
4. Insert a *WorkerPool*.
5. In order to allow the *Workers* to move between *Workplaces* and the *WorkerPool*, we need to create *FootPaths*.
6. Create 6 horizontal *FootPaths* with a length of 5m.
7. Create 6 vertical *FootPaths* with a length of 5m.
8. Create 1 horizontal *FootPath* with a length of 1m.
9. Connect the *FootPaths* with the *Workplaces* and the *WorkerPool* as indicated in the figure below. Note that the *Workers* should be able to walk past occupied *Workplaces*.



10. Insert a *Broker*.
11. Double-click on the *WorkerPool* and set the *Broker* as the *Broker* for the *WorkerPool*.
12. We need to state that the work stations require a *Worker* for the processing of the *MU*. At each work station (7 *SingleProcs*), go the tab *Importer*, set the *Importer* to *Active*, and state that the *Broker* is the *Broker* you inserted in your *Frame*.



13. At each work station (7 *SingleProcs*), go the tab *Importer*, and set the *Services for Processing* for the *SingleProcs Pistons*, *SparkPlug*, *Cooling* and *AirFilter* to *Basic* with amount 1 (you might need to turn off inheritance for this attribute).

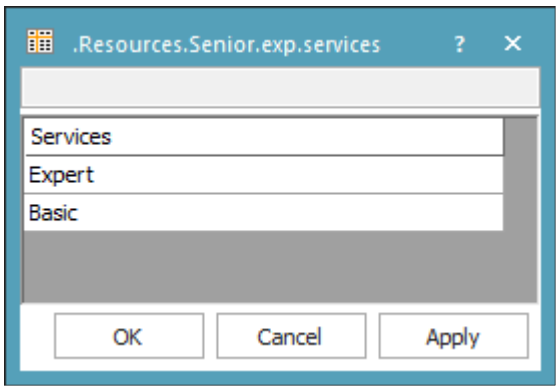


14. Do the same for the other stations, but then using *Expert* for *Service*, with amount 1.

We have now established the working environment for our *Workers* and the only thing missing are the *Workers* themselves. In the next task we are going to establish the working force for our assembly station.

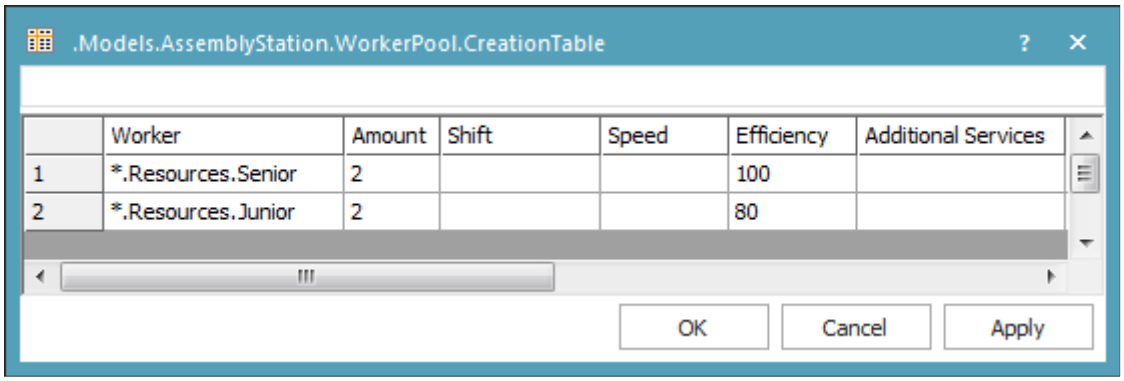
Task: Create the Workforce

1. Duplicate a *Worker* in the *Class Library* and rename it *Engineer*.
2. Set the walking speed of the *Engineer* to *1.38 m/s* (double-click, tab *Attributes*).
3. Derive two *Workers* from the *Engineer* in the *Class Library* and rename them to *Senior* and *Junior* respectively.
4. Create *Icons* for the *Senior Engineer* and the *Junior Engineer* that allows you to visually distinguish them. You might need to turn off inheritance of the image. Take care of the fact that the worker icon has multiple versions depending on the state of the worker (*Operational*, *Failed*, *Pause*, and *Working*).
5. We need to state what service the worker can supply, which needs to be stated in the tab *Attributes* of the *Worker*.
6. Double-click on the *Senior Worker* and click on the button *Service* in the tab *Attributes*.
7. State that the *Senior Worker* provides the *Service Expert* and *Basic.*



8. Repeat the steps for the *Junior Worker*, but state that he only provides the *Service Basic.*
9. Double-click on the *WorkerPool* and configure the *CreationTable* to generate two *Senior Workers* and two *Junior Workers*.

10. Set the *Efficiency* of the *Senior Worker* to 100% and the *Efficiency* of the *Junior Worker* to 80%.



11. In order to allow a *Worker* to work and move to the *Workplace*, we need to state which services are supported by the *Workplace*.
12. Double-click on the *WPElectricalBattery* object, click the button *Supported Services*, and add the supported service *Expert*. Do the same for *WPCarburetor* and *WPTurbo*.
13. Give the other stations the supported service *Basic*.
14. Test your model and check whether it runs without bugs.

You might have noticed that something odd will occur in the current setting of your model if you followed the tutorial correctly. At a certain point in time, around day 20 (≈10,000/(24x60)), you will run out of *Tables* on which you store your *Engines*. We will need to write a *Method* that will make sure that the *Tables* are returned to the *Store* where they can be reused.

Task: Returning the Tables

1. Insert a *Method* in your *Frame ControlPanel* and rename it to *ReturnTables*.
2. Insert the following code:

```
-- Delete MU's and send Table back
@.cont.Move(AssemblyStation.Exit)
@.Move(AssemblyStation.PalletStore)
```

3. Set the *Method* as the *Exit Control* for the final line (line after the *Turbo* station).

Next, we need to keep track of the waiting times of the *Engines* at the different stations.

Task: Determining the Waiting Time

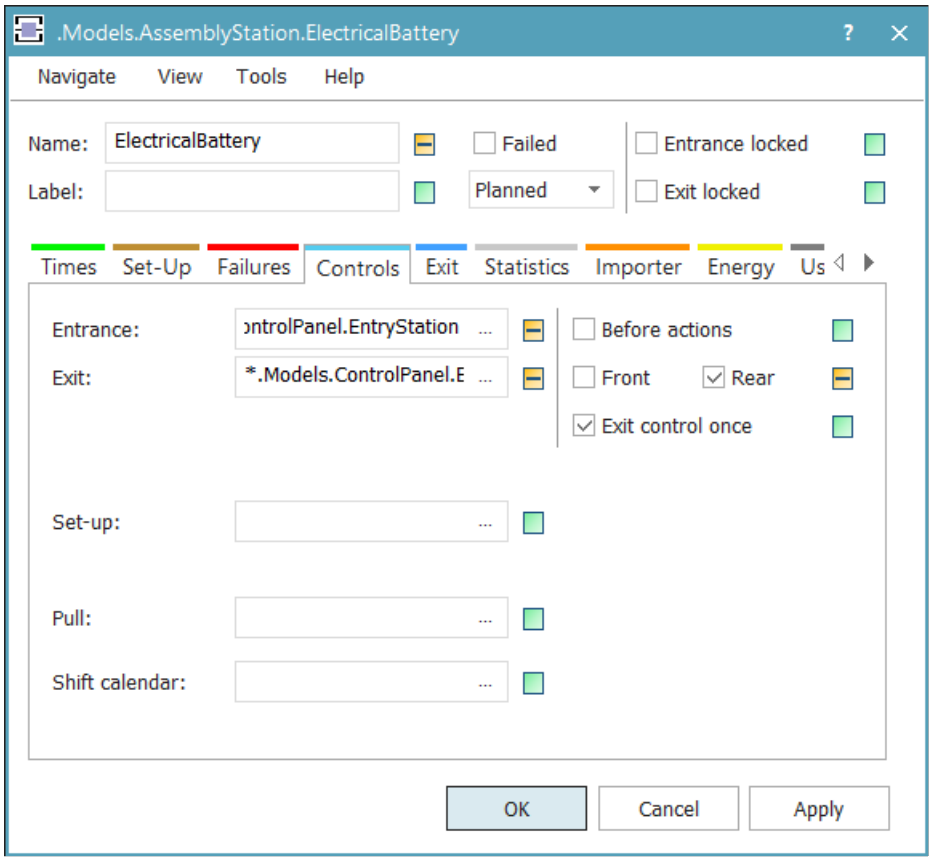
1. Insert two *Methods* in your *Frame ControlPanel* and rename them to *EntryStation* and *ExitStation*.
2. Insert the following code in *EntryStation*:

```
var ObjString: string

ObjString := ?.Name

if @.cont.getAttribute("PT" + ObjString) = 0 then
    @.move
end
```

3. Insert the following code in *ExitStation*:

```
var CurrentTime, LocalWaitTime: time
var ObjString: string

CurrentTime := root.EventController.SimTime
ObjString := ?.Name
LocalWaitTime := CurrentTime - @.cont.getAttribute("WT" + ObjString) -
                 @.cont.getAttribute("PT" + ObjString)
@.cont.setAttribute("WT" + ObjString, LocalWaitTime)
@.cont.setAttribute("WT" + ?.succ.succ.name, CurrentTime)
```

4. Try to understand the code of *EntryStation* and *ExitStation*. If you do not understand the code, you can refresh your knowledge on the anonymous identifier ? in Section 4.2.
5. Set the *Methods EntryStation* and *ExitStation* as *Entry* and *Exit Control* respectively for each of the seven *Processing Stations* at the *Worker Assembly Line*. Make sure that the *Exit Control* is triggered by the rear of the *MU* (otherwise we have to specify, within the *Method ExitStation*, that the *MU* needs to continue to the next object).



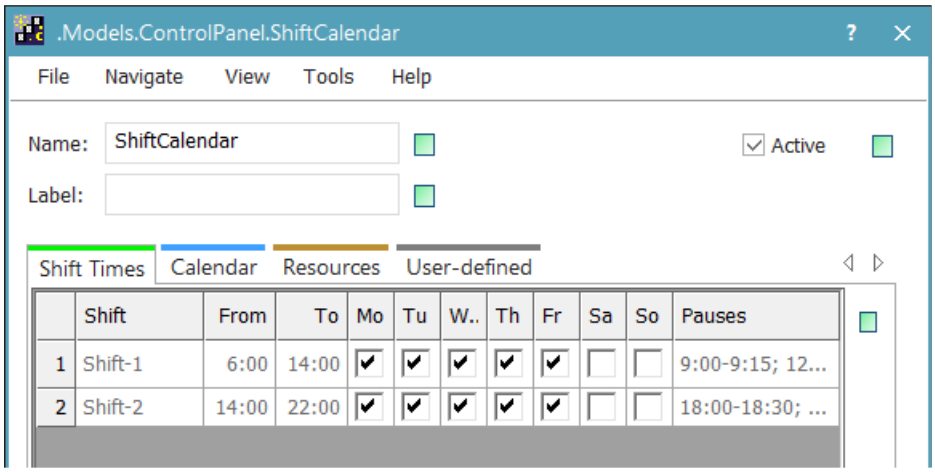
The reason we need to set the exit control to rear and not front, is that in the latter case Plant Simulation also calls the *Method* when the *MU* tries to exit the station whereas its successor (*Line*) is full. In this case the *MU* waits till there is room at the next object, upon which the *Method* is called again, resulting in wrong calculation of waiting times. Setting the exit control to rear ensures that the *Method* is only called when the rear of the *MU* exits the station, this only happens when it is possible to move the *MU* to the succeeding *Line*.

8.4 Shift Calendars

ShiftCalendars are an easy way to incorporate shifts for *Material Flow* objects and *Workers*. Several shifts can be created per *ShiftCalendar* and for every *Worker*-type you can define which shifts they work. Objects can also work completely different shifts but need a specific *ShiftCalendar* per set of shifts to do so. If a certain set of machines A works shifts 1, 2 and 3 and set of machines B works shift 4 and 5, you need two *ShiftCalendars*: one for shifts 1, 2 and 3 and one for shifts 4 and 5. In the following tasks you will get familiar with the basics of the *ShiftCalendar*. In the next section, we will shortly introduce 3D simulation. For the sake of simplicity, we will not continue to the next chapter with *ShiftCalendars* or 3D in the model, but continue with the model as it is at this point in time.

Task: Create a shift calendar

1. Save your current model and create a separate saved file to incorporate the changes from this and the next section.
2. Add a *ShiftCalendar* to your *ControlPanel.*
3. Double-click the *ShiftCalendar*, the following screen should appear:

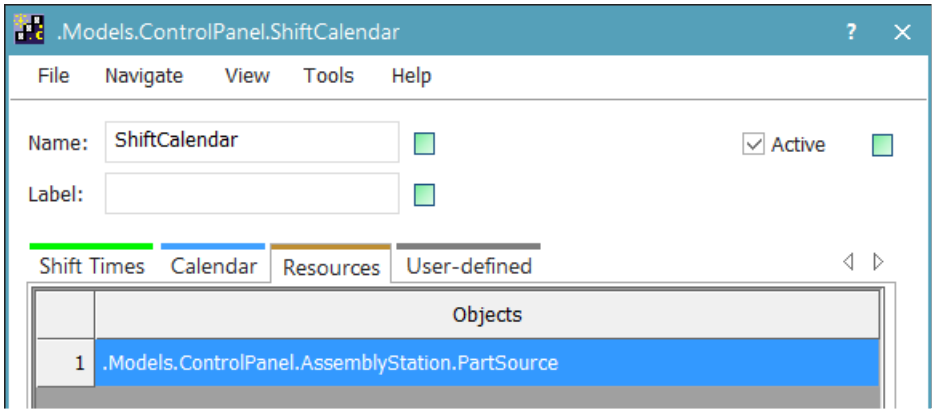


4. Edit the *Shifts*, so that they both also cover Saturday and Sunday, by simply checking the corresponding boxes (you might need to turn off inheritance for this attribute).
5. Change the starting time of Shift-1 to 5:00 and its ending time to 13:00.
6. Click *OK* to confirm the changes made, and exit this screen.

The *ShiftCalendar* now contains the right shifts, but the model is not yet following these shifts because we need to define what objects work only within these shifts. Also note how you can define *Pauses*, enter a starting and an ending time for your pause and separate them with a dash. If you want to enter another pause, simply add a semicolon and repeat the same steps. If you want the *ShiftCalendar* to control a "non-*Worker*" object, you can simply add this object by "drag and drop" to the resources tab of the *ShiftCalendar*.

Task: Create shifts for the Source

1. Double-click your *ShiftCalendar.*
2. Click on the *Resources* tab.
3. Drag the *Source* (from your model, i.e., from the *Frame AssemblyStation* placed on the *RootFrame*) into the white field under objects. It should look like this:



   Alternatively, you could have defined the *ShiftCalender* under the *Control* tab in the *PartSource* object itself.
4. Click *OK* to confirm the changes made, and exit this screen.
5. Run the model for 10 days. Note that now the source only generates *Engines* during the defined shifts. When enabling *Blocking* under the *Attributes* tab of the *PartSource*, the source will generate *Engines* at all times and buffer them till the start of a shift, upon which a batch of *Engines* will be provided to the *PartBuffer*.

To control the shifts of *Workers,* enter the *WorkerPool* as object under the tab Resources in the *ShiftCalendar*. Open the specific *Worker* type in the *Class Library* and specify the shifts the *Worker* is supposed to work. If a *Worker* type works more than one shift, separate the names of the shift with a semicolon. If you do not specify shifts, a worker works every shift available in the *ShiftCalendar* that controls its *WorkerPool.*

Note:

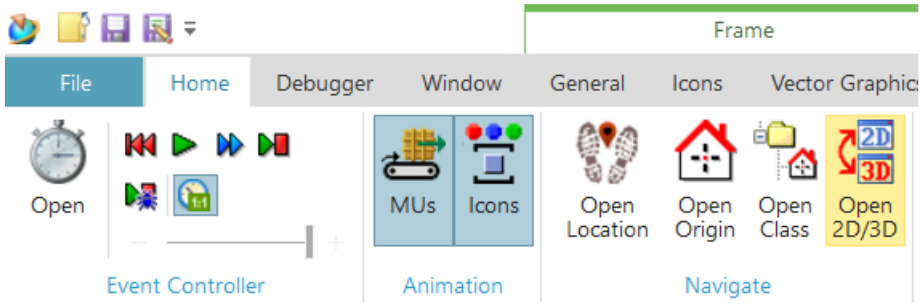
When you have specified a shift, and it ends while a process is ongoing, the process will be paused and continued when the required resource (in this case a *Worker*) becomes available again. Until then, the *MU* is waiting and the machine will be in the state *Waiting for an Exporter*, showing an orange LED.

8.5 3D modelling

It is possible to build completely new 3D models using your own 3D shapes of *MUs* and other objects. If you are curious about what is possible, take a look at the example: *3D Carbody* (at the *Start Page* of Plant Simulation select *Example Models > 3D Carbody*). In this tutorial, we will simply explain how to turn your existing 2D model into a 3D model, and how to navigate around your 3D model.

Task: Create a 3D model

1. Open your instance of *AssemblyStation* (located on the *RootFrame*).
2. Press the *Open 2D/3D* button and choose *Yes* in the diolag window that appears.

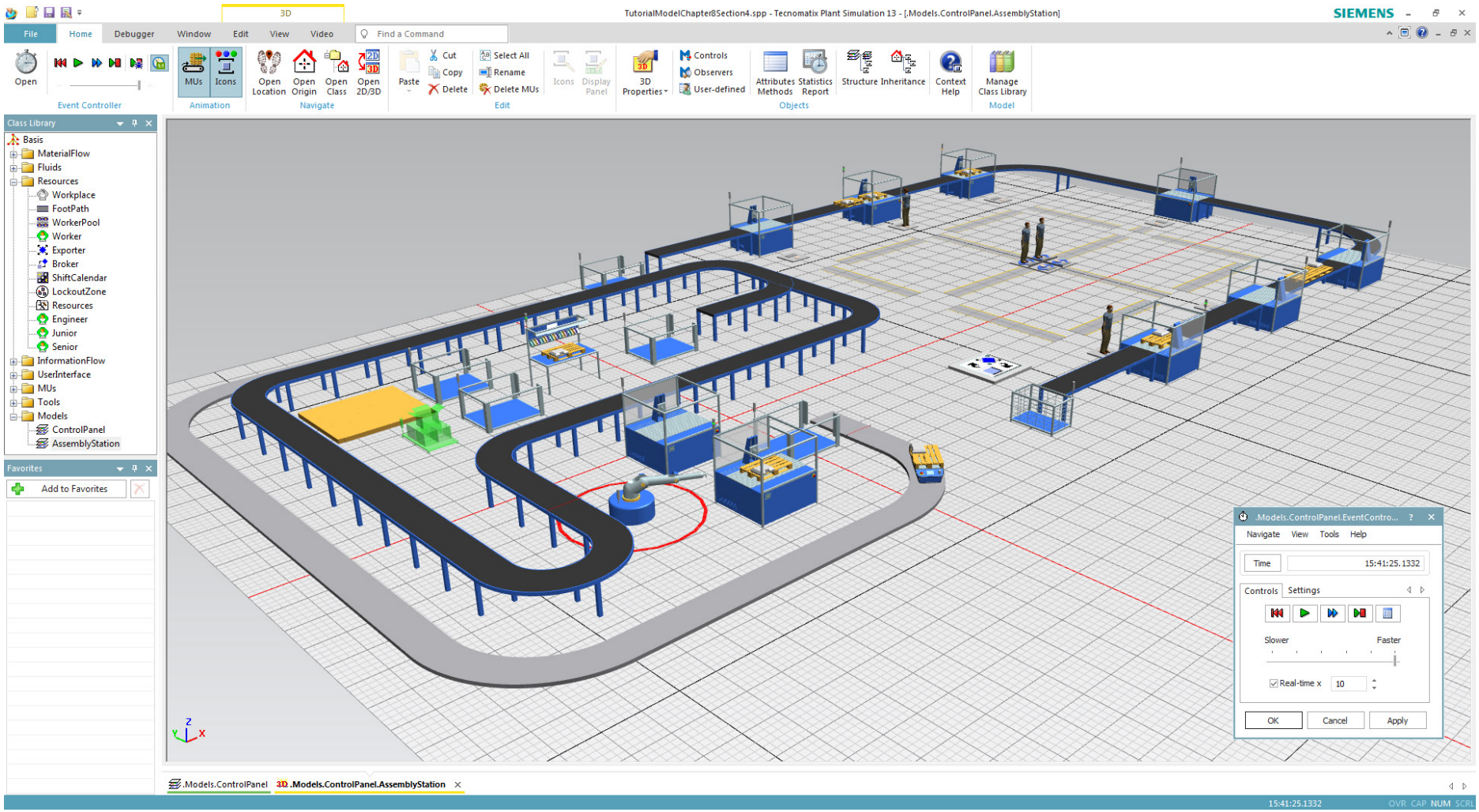


3. Run your model at a reasonable speed, so that you see every object in action. For animation purposes, turning on the real-time option in the *EventController* might be convenient (see Section 8.2).

There are several ways to navigate around a 3D model, we will explain how to do this by using your mouse.

- To zoom in or out, simply use your scroll wheel.
- To navigate around your model, hold down the right mouse button and move your mouse.
- To change your point of view, hold down both mouse buttons and move your mouse.

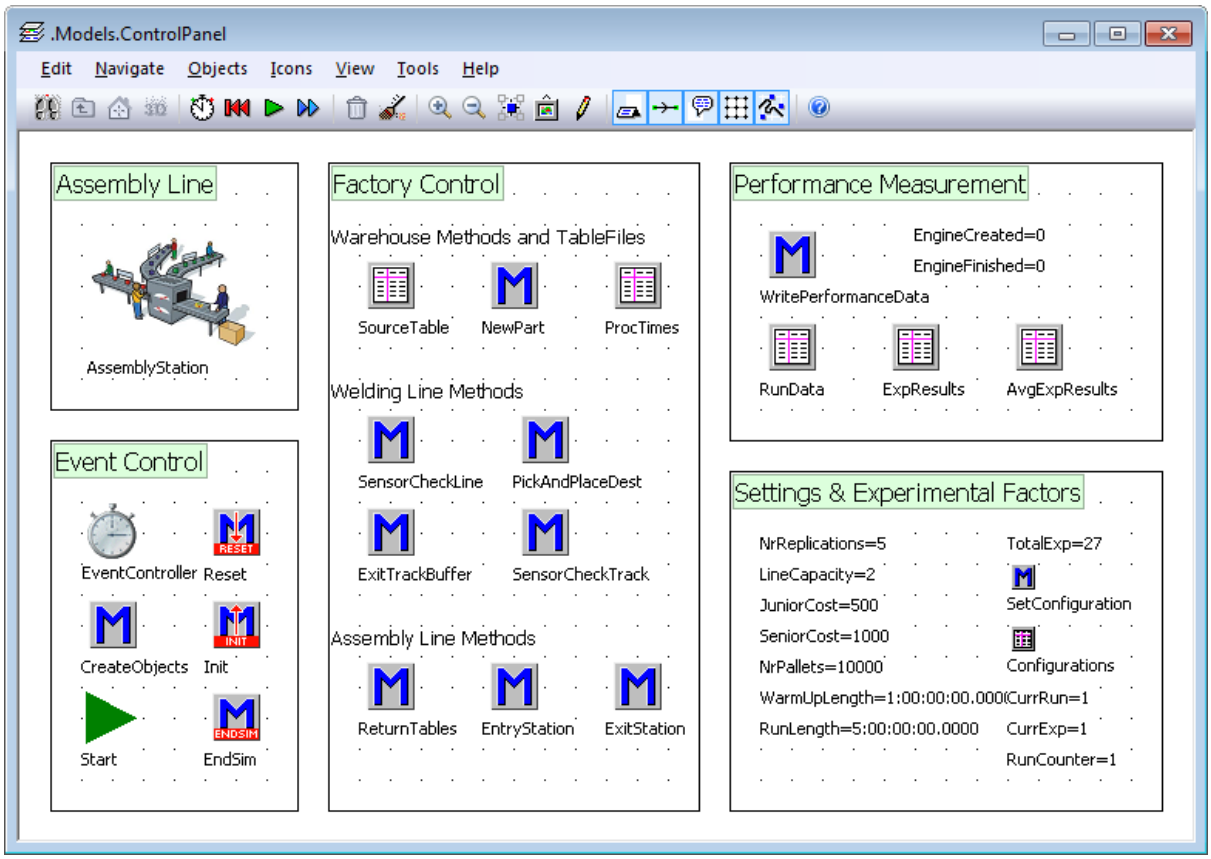
It is not necessary to save this model; 3D models are a lot larger than 2D models, since a whole 3D Library is created.


TutorialModelChapter8Section4.spp - Tecnomatix Plant Simulation 13 - [.Models.ControlPanel.AssemblyStation]
SIEMENS
File
Home
Debugger
Window
Edit
View
Video
Find a Command
Open
MUs
Icons
Open Location
Open Origin
Open Class
Open 2D/3D
Paste
Cut
Copy
Delete
Select All
Rename
Delete MUs
Icons
Display Panel
3D Properties
Controls
Observers
User-defined
Attributes Methods
Statistics Report
Structure
Inheritance
Context Help
Manage Class Library
Event Controller
Animation
Navigate
Edit
Objects
Model
Class Library
Basis
MaterialFlow
Fluids
Resources
Workplace
FootPath
WorkerPool
Worker
Exporter
Broker
ShiftCalendar
LockoutZone
Resources
Engineer
Junior
Senior
InformationFlow
UserInterface
MUs
Tools
Models
ControlPanel
AssemblyStation
Favorites
Add to Favorites
.Models.ControlPanel.EventContro...
Navigate
View
Tools
Help
Time
15:41:25.1332
Controls
Settings
Slower
Faster
Real-time x
10
OK
Cancel
Apply
.Models.ControlPanel
.Models.ControlPanel.AssemblyStation
15:41:25.1332
OVR CAP NUM SCRL

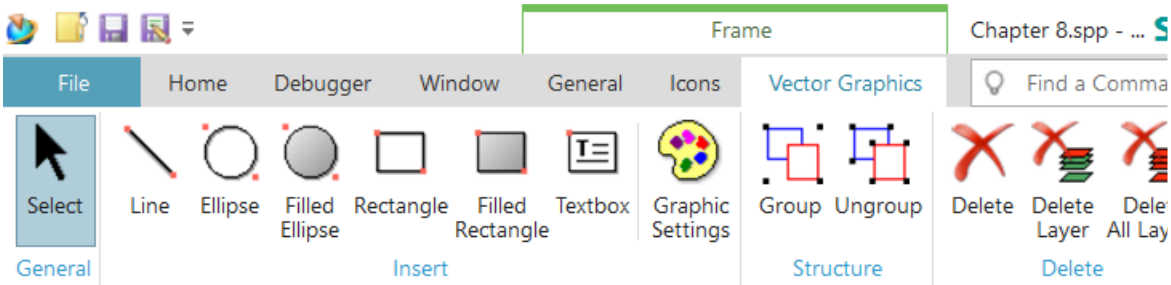
8.6 Creating a Control Panel for Experimenting

We have created a model of a car engine manufacturer, but in order to see what the ideal configurations for the assembly line are, we have to perform experiments. To properly manage the experiments, we need to complete the control panel. In Chapter 5 we have used the *ExperimentManager* for conducting experiments, but it is also possible to control the experiments yourself. Here we will control the experiments by making use of *TableFiles, Variables,* and *Methods*. If you would like to refresh your memory on using *TableFiles* for tracking objects and writing data to them, see Chapter 4. In Chapter 9 we will elaborate more on different techniques for analysing your model and conducting experiments.

For 3D purposes we have created the assembly line into one *Frame*, which is a good way to present a model to a client. In this chapter, we will further improve the control panel, by creating oversight and add variables that control different settings. At the end of this chapter your control panel will look similar to the following figure:



You can use the *Vector Graphics Mode* to draw rectangles for organizing your model. You can access this mode by clicking on *Vector Graphics* in the *Ribbon* while you have opened your *Frame*.



Note:

While *Vector Graphics Mode* is active, you cannot move or change objects. To exit *Vector Graphics Mode*, go to a different menu in the *Ribbon*. The *Vector Graphics Mode* allows you to draw:

- lines
- ellipses
- filled ellipses
- rectangles
- filled rectangles
- text boxes

You can work on several different layers or group objects (*Vector Graphic Mode* objects in this case) together. The *Vector Graphics Mode* is a great way to organise models.

Task: Organise the Control Panel

1. Open your original model resulting from the end of Section 8.3, as saved in the first step of the first task of Section 8.4 (so without the *Shift Calendar* and 3D animation).
2. Draw the required rectangles via the *Vector Graphics Mode*.
3. Add the *Comments* (available under *Toolbox > User Interface*).
4. Rearrange the *Methods* and *Objects* you inserted so far.

First we will make sure that we can track the performance of our model. For that reason we will add a *Method* and *TableFiles* that will be able to store the *User-defined Attributes* of our *Engines*.

Task: Create the Performance Measurement Panel

1. Add a *Method* and three *TableFiles* to your panel (area *Performance Measurement*).
2. Rename the *Method* to *WritePerformanceData*, and rename the *TableFiles* to *RunData, ExpResults*, and *AvgExpResults* respectively.
3. Add two *Variables* and rename them *EngineCreated* and *EngineFinished* respectively. Set their data type to *Integer*.
4. Add the following code to *SensorCheckTrack*, somewhere around the code where you also set the attributes of the *Engine* (where the *Engine* is represented by @.cont.cont because it is placed on a pallet on a transporter):

```
if EventController.SimTime > WarmUpLength
    EngineCreated += 1
end
```

5. Create 17 columns in the *TableFile ExpResults* of data type *Real*. Next, create a column index and copy-paste the following column names in the column index (row 0):

PTElectricalBattery PTPistons PTCarburetor PTSparkplug PTCooling PTAirfilter PTTurbo
WTElectricalBattery WTPistons WTCarburetor WTSparkplug WTCooling WTAirfilter WTTurbo WTExit
TimeWarehouseWelding TotalLeadTime

6. Give each column in the *TableFile RunData* the data type *Real*.

7. Give the *TableFile AvgExpResults* the following layout (take care of the data types):

	integer 1	integer 2	real 3	real 4	integer 5	integer 6
string	ExperimentNr	RunNr	AvgAssemblyTime	CostConfiguration	EngineCreated	EngineFinished
1						

8. Insert the following code in the *Method WritePerformanceData* (see also code in the Appendix) and try to understand it:

```
var i: integer
var CheckTotalLeadTime: real

if EventController.SimTime > WarmUpLength

    -- Update performance
    @.WTExit := root.EventController.SimTime - @.WTExit
    EngineFinished += 1

    for i := 1 to 15
        ExpResults[i, EngineFinished] := @.getAttribute(ExpResults[i, 0])
    next

    ExpResults["TimeWarehouseWelding",EngineFinished] := @.TimeWarehouseWelding
    ExpResults["TotalLeadTime",EngineFinished] :=
        ExpResults.Sum({1,EngineFinished}..{16,EngineFinished})
    RunData[CurrRun, EngineFinished] := ExpResults["TotalLeadTime",EngineFinished]

    -- Code verification
    CheckTotalLeadTime := EventController.SimTime - @.CreationTime
    if abs(ExpResults["TotalLeadTime", EngineFinished] - CheckTotalLeadTime) > 1
        debug
    end

end
```

9. Set the *Method* as *Entrance Control* for the *Exit* of the *Assembly Line*.

We are going to simulate multiple replications of various experiments. As we are going do this manually without the use of the *ExperimentManager* (see Chapter 5 and 9), we need to make sure that we still keep track of the data we want to use for analysing the different configurations. This data will be stored in the *TableFile AvgExpResults*. The next panel we are going to build is the *Settings & Experiment Factors Panel*.

Task: Create the Settings & Experiment Factors Panel

1. Add a *TableFile* with the name *Configurations* and a *Method* with the name *SetConfiguration* to the *Settings & Experiment Factors Panel*.
2. Give the *TableFile Configurations* the following layout:

	integer 1	integer 2	integer 3
string	ExperimentNr	JuniorNr	SeniorNr

3. Insert the following code in the *Method SetConfiguration*.

```
var y: integer -- Row counter
var i, j: integer -- Counters for the loop

y := 1 -- Initialisation

for i := 0 to 6 -- Maximum Amount of Juniors Present
    for j := 1 to 7 - i -- Maximum Amount of Seniors Present
        if i + j >= 2 -- Need enough employees
            -- Write Table Information (Configurations)
            Configurations["JuniorNr", y] := i
            Configurations["SeniorNr", y] := j
            Configurations["ExperimentNr", y] := y
            y += 1
        end
    next -- j
next -- i

TotalExp := y - 1
```

4. Insert 6 variables for the Settings, with the following names and data types:
   - *NrReplications (Integer)* with setting 5.
   - *LineCapacity (Integer)* with setting 2.
   - *JuniorCost (Integer)* with setting 500.
   - *SeniorCost (Integer)* with setting 1000.
   - *WarmUpLength (Time)* with setting 1:00:00:00 (1 day).
   - *RunLength (Time)* with setting 5:00:00:00 (5 days).
5. Insert 4 variables to be used as counters:
   - *TotalExp (Integer)*
   - *CurrRun (Integer)*
   - *CurrExp (Integer)*
   - *RunCounter (Integer)*

The variables we added have the following purposes:

- *NrReplications*: Setting for the total amount of replications you want to use.
- *LineCapacity*: Capacity of a single piece of the assembly line.
- *JuniorCost*: Cost for 24 hour of junior personnel.
- *SeniorCost*: Cost for 24 hour of senior personnel.
- *WarmUpLength*: Variable in which you can set the warm up length of your replication.
- *RunLength*: Variable in which you can set the length of a complete replication.
- *TotalExp*: The number of experiments you are going to conduct in your simulation study. This is set automatically within the *Method SetConfiguration.*
- *CurrRun*: Counter for the current replication in an experiment.
- *CurrExp*: Counter for the current experiment of your simulation study.
- *RunCounter*: Counter for keeping track of the number of replications carried out so far.

The code *SetConfiguration* is used to define all configurations with a maximum of 7 employees of one type, a minimum of 2 employees in total, and with at least one senior. Experimenting with all the configurations cost a lot of time and you can imagine that many configurations are simply not feasible.

The final panel we need to set up is the *Event Control Panel*.

Task: Create the Event Control Panel

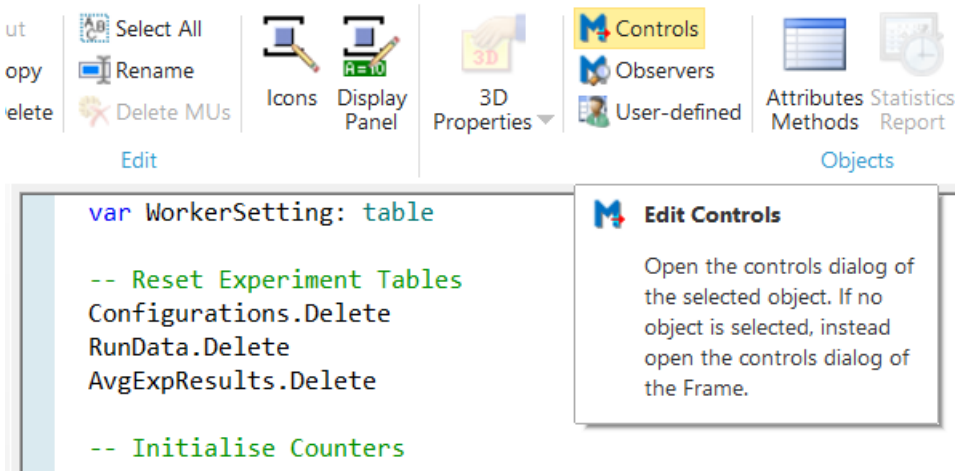
1. Add two *Methods* and rename them *Start* and *EndSim* respectively.
2. Insert the following code in the *Method EndSim* (see also code in the Appendix) and try to understand it:

```
var WorkerSetting: table

-- Stop EventController
EventController.Stop

-- Store Data
AvgExpResults["ExperimentNr", AvgExpResults.YDim + 1] := CurrExp
AvgExpResults["RunNr", AvgExpResults.YDim] := CurrRun
AvgExpResults["EngineCreated", AvgExpResults.YDim] := EngineCreated
AvgExpResults["EngineFinished", AvgExpResults.YDim] := EngineFinished
AvgExpResults["AvgAssemblyTime", AvgExpResults.YDim] :=
    ExpResults.meanValue({"TotalLeadTime", 1}..{"TotalLeadTime", *}) -
    ExpResults.meanValue({"TimeWarehouseWelding", 1}..{"TimeWarehouseWelding",*})
AvgExpResults["CostConfiguration", AvgExpResults.YDim] :=
    (Configurations["JuniorNr", CurrExp] * JuniorCost +
    Configurations["SeniorNr", CurrExp] * SeniorCost) * (RunLength / 86400)
    -- Divide by 86400 to multiply the cost with the number of days

if CurrRun < NrReplications  -- Run an additional replication
    CurrRun += 1
    RunCounter += 1
    EventController.Reset
    EventController.Init
    EventController.Start
elseif CurrRun = NrReplications and CurrExp < TotalExp
    -- Run a new configuration and set the new worker settings
    CurrExp += 1
    WorkerSetting.Create
    AssemblyStation.WorkerPool.getCreationTable(WorkerSetting)
    WorkerSetting[2, 1] := Configurations["SeniorNr", CurrExp]
    WorkerSetting[2, 2] := Configurations["JuniorNr", CurrExp]
    AssemblyStation.WorkerPool.setCreationTable(WorkerSetting)
    -- Other settings
    CurrRun := 1
    RunCounter += 1
    RunData.Delete
    EventController.Reset
    EventController.Init
    EventController.Start
else
    -- Finish simulation
    promptMessage("Simulation finished")
end
```

3. Insert the following code in the *Method Start* (see also code in the Appendix) and try to understand it:

```
var WorkerSetting: table

-- Reset Experiment Tables
Configurations.Delete
RunData.Delete
AvgExpResults.Delete

-- Initialise Counters
CurrRun := 1
CurrExp := 1
RunCounter := 1
EventController.End := RunLength + WarmUpLength

-- Set the configurations
SetConfiguration

-- Reset Workers
WorkerSetting.Create
AssemblyStation.WorkerPool.getCreationTable(WorkerSetting)
WorkerSetting[2, 1] := Configurations["SeniorNr", 1]
WorkerSetting[2, 2] := Configurations["JuniorNr", 1]
AssemblyStation.WorkerPool.setCreationTable(WorkerSetting)

-- Start simulation
EventController.Reset
EventController.Init
EventController.Start
```

4. Select *Home > Objects > Controls* in the *Ribbon* while in the *Dialog Window* of the *Method Start*.



5. In the new dialog, fill in *self* at the *Select* field.
6. Give the *Method Start* a new *Icon* in which you are able to distinguish it from the rest.
7. Adjust the code of the *Method Init* to the following:

```
-- Set line capacity
AssemblyStation.Line1.Capacity := LineCapacity
AssemblyStation.Line2.Capacity := LineCapacity
AssemblyStation.Line3.Capacity := LineCapacity
AssemblyStation.Line4.Capacity := LineCapacity
AssemblyStation.Line5.Capacity := LineCapacity
AssemblyStation.Line6.Capacity := LineCapacity
AssemblyStation.Line7.Capacity := LineCapacity
AssemblyStation.Line8.Capacity := LineCapacity

-- Create Movables
CreateObjects
```

8. Adjust the code of the *Method Reset* to the following:

```
-- Empty system
AssemblyStation.deleteMovables

-- Delete Data
ExpResults.Delete
EngineCreated := 0
EngineFinished := 0

-- Set the random number stream
EventController.RandomNumbersVariant := CurrRun
```

9. In the *EventController*, tab *Settings*, uncheck *Show summary report*, to avoid receiving a report after every replication.
10. Run your model using the *Method Start*. To validate your model, you can use the following numbers:
    a. Experiment 1: *AvgAssemblyTime*≈1645 and *CostConfiguration*=10,000;
    b. Experiment 14: *AvgAssemblyTime*≈695 and *CostConfiguration*=15,000;
    c. Experiment 27: *AvgAssemblyTime*≈960 and *CostConfiguration*=20,000.

The *Method EndSim* is, just like the *Methods Init* and *Reset*, a *Method* that is being called under certain conditions of the *EventController*. The *EndSim* is called when the end time of your simulation run is reached. In our particular case, *EndSim* will store the data of a simulation run in the table *AverageExperimentResults* and increment the counters until all the experiments are completed.

We have set for the *Method Start* the control *Self*. This means that when we single click on this *Method*, it will run. In our particular case, it triggers the creation of configurations, initialises several counters, apply the settings of the first configuration, and start running the first replication of the first configuration. All other replications and configurations are triggered by the *Method EndSim*.

8.7 Assignment B2: Warm-up Period, Run Length and Number of Replications

The car manufacturing company CeeCar Inc. produces a wide range of car components for other car manufacturers. Until now, the company has focussed on relatively small car components, such as start engines, radiators, and carburetors. But recently they decided to expand their business towards producing fully assembled car engines. They already designed the assembly line, but hired you to provide insight into the performance that can be expected from this system. The company provides a dataset of the interarrival times of unassembled car engines that arrive at the assembly line.

Assignment B2.1. Fit a probability distribution to the dataset (see accompanying file) using Excel or a similar spreadsheet program. Follow the procedure as shown in the book of Law (2015): hypothesizing distributions, parameter estimation, checking fit with plots, QQ plot, and goodness-of-fit test. Next, use this distribution for the interarrival times in your simulation model (*PartSource*).

Assignment B2.2. Carry out experiments to support decision making for the management of CeeCar Inc. For setting up the experiments, you should distinguish between the following steps (see lecture sheets):

a. Determine the type of simulation using the book of Law (2015).
b. Estimate the required warm up period using Welch's graphical procedure (you can use the table *RunData* for this, remember to first set the variable *WarmUpLength* to zero). Motivate your choice of system configurations for applying this approach.
c. Choose the number of runs and run length, given a "replication/deletion approach". Note that as far as the confidence interval is concerned, use $\gamma = 0.05$ (relative error allowed) and $\alpha = 0.05$ ($1-\alpha$ = confidence level).
d. Analyse the outcomes considering the different performance indicators from the *TableFile AvgExpResults* and rank the outcomes according to a combined indicator of your choice.
e. For the best and second-best system configurations resulting from the previous step, compare the difference in outcomes using a paired t-test. Is the difference significant?

Assignment B2.3. For the configuration of your choice, provide insight into the utilisation of machines and workers, and produce a histogram for the distribution of the lead times of engines.

Assignment B2.4. For Assignment B2.2, the use of common random numbers (CRN) has been assumed. Study the model to confirm the use of CRN and change the code to disable CRN. Study the relevance of using CRN for comparing both configurations from Assignment B2.2e. Does it make sense to use CRN here? Motivate your answer.

Assignment B2.5. Write a management summary for this project (max. 1 A4). This includes a short description of the system, research questions posed, research set up, and main findings.

Deliverables:

- Models and answers to assignments B2.1-B2.5 (including computations and motivation).
- A report introducing your models, providing the answers and motivation, and containing the management summary. As far as computations are concerned, you may refer to spreadsheet models. Please add these spreadsheets as well.

9 Building a Model: Simulation Optimisation

This chapter will elaborate on some of the extended features of *Plant Simulation*. First, we will slightly adjust the model you have built in Chapter 8. We will add some additional constraints to the model, namely that the engines can only continue on the assembly line in a predefined batch. Furthermore, we will extend the assembly line such that two *MUs* can be processed at the same time per station. This also means that we will add an additional workplace to each station in order to process two *MUs* at the same time.

In the second section of this chapter, we will add the *ExperimentManager* to the model. In Chapter 5 we have already introduced the *ExperimentManager*, and in this chapter we will continue to build on that knowledge.

In the third section, we will introduce the *Genetic Algorithm*, for which *Plant Simulation* has a specific tool, namely the *GAWizard*. We will elaborate on the basics of this tool and algorithm. We will use this tool to determine what will be the best sequence for releasing the products (within the batch) to the *Assembly Line* as you might recall that different product types require different processing steps.

The final section will be about *Simulation Optimisation*, which is finding the best input variables without considering each possibility as we have done previously. In the *ExperimentManager* you will predefine the experiments you wish to carry out, which is not considered to be a *Simulation Optimisation* approach. The *Genetic Algorithm* is typically an example of *Simulation Optimisation*. There are a wide range of algorithms you could use for *Simulation Optimisation*. In this final section we will introduce the basics of building your own algorithm, which you can apply to find the best input parameters for your model. In the assignment of this chapter you will need to build such an algorithm.

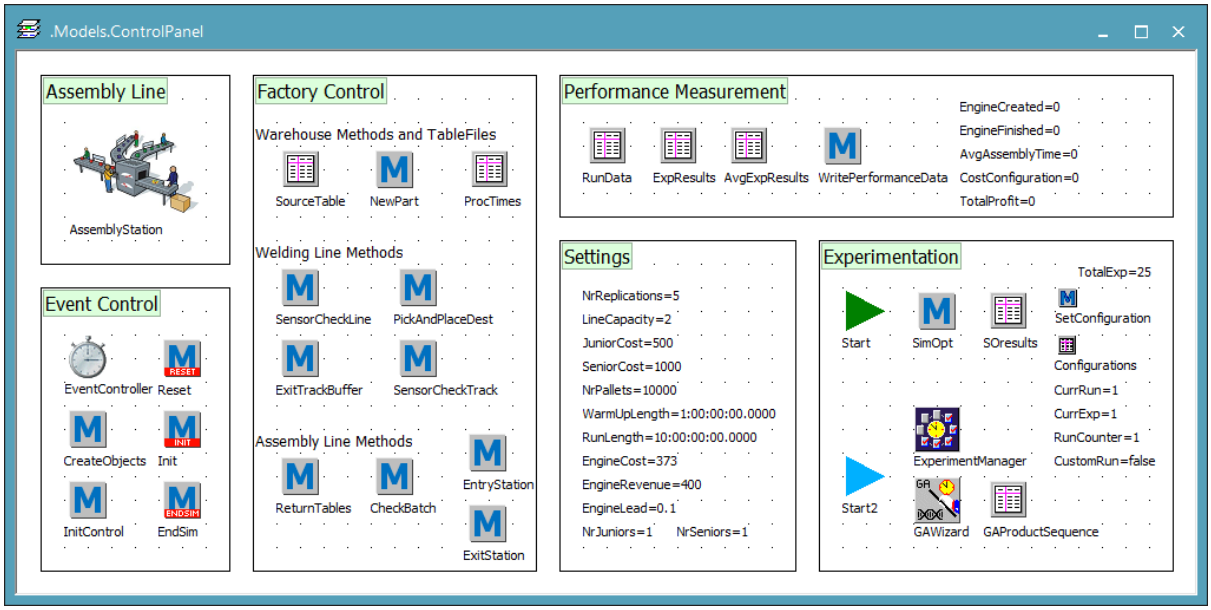
Subjects dealt with in this chapter:

- *ExperimentManager*
- *GAWizard*
- *Simulation Optimisation*

9.1 Adjusting the model of Chapter 8

In this section, we will change our model, such that we can use the *ExperimentManager* and *GAWizard* to perform runs, but also the custom runs from the previous chapter. Furthermore, we will make some adjustments to the assembly line and introduce batch processing. Some code written for the model in Chapter 8 does not apply for working with the *ExperimentManager*. So we make some of this code conditional on whether we perform experiments like we did in the previous chapter, or by using the *ExperimentManager* or *GAWizard*.

We will now start to make adjustments to the *ControlPanel* to enable the model to run experiments in different ways. At the end of this chapter, your *ControlPanel* might look like this:



Task: Enable the model to run experiments in different ways

1. Add nine variables to your *ControlPanel* and name them as follows: *AvgAssemblyTime* (data type *Real*), *CostConfiguration* (data type *Real*), *TotalProfit* (data type *Real*), *CustomRun* (data type *Boolean*), *NrJuniors* (data type *Integer*), *NrSeniors* (data type *Integer*), *EngineCost* (data type *Integer*), *EngineRevenue* (data type *Integer*), and *EngineLead* (data type *Real*).
2. Set the value of *EngineCost*, *EngineRevenue*, and *EngineLead* to 373, 400, and 0.1 respectively. EngineLead represents the loss of interest of engines during the leadtime, and is approximately given by the average value of an engine during the leadtime times the daily interest rate (1.1^(1/365)-1).
3. Set the value of *NrJuniors* and *NrSeniors* both to 2.
4. Open the *Method Reset*.
5. Add the following code:

```
AvgAssemblyTime := 0
CostConfiguration := 0
TotalProfit := 0
```

6. Make the code that sets the random number stream conditional on the use of *CustomRun* (experiments without the *ExperimentManager* or *GAWizard*):

```
-- Set the random number stream
if CustomRun
    EventController.RandomNumbersVariant := CurrRun
end
```

7. Open the *Method EndSim* and add the following code just below the line that stops the EventController:

```
-- Perform general computations
if CustomRun
    AvgAssemblyTime := ExpResults.meanValue({"TotalLeadTime", 1}..{"TotalLeadTime", *}) -
        ExpResults.meanValue({"TimeWarehouseWelding", 1}..{"TimeWarehouseWelding", *})
end

CostConfiguration := (NrJuniors * JuniorCost + NrSeniors * SeniorCost) * (RunLength / 86400)
TotalProfit := EngineRevenue * EngineFinished - EngineCost * EngineCreated -
    CostConfiguration - (EngineLead * EngineFinished * AvgAssemblyTime / 86400)
    -- Divide by 86400 to multiply the cost with the number of days
```

8. Make all lines of code in *EndSim* below the newly inserted code conditional on *CustomRun* in the following way:

```
-- Store Data in case of CustomRun
if CustomRun
    AvgExpResults["ExperimentNr", AvgExpResults.YDim + 1] := CurrExp
    AvgExpResults["RunNr", AvgExpResults.YDim] := CurrRun
    AvgExpResults["EngineCreated", AvgExpResults.YDim] := EngineCreated
    AvgExpResults["EngineFinished", AvgExpResults.YDim] := EngineFinished
    AvgExpResults["AvgAssemblyTime", AvgExpResults.YDim] := AvgAssemblyTime
    AvgExpResults["CostConfiguration", AvgExpResults.YDim] := CostConfiguration
    AvgExpResults["TotalProfit", AvgExpResults.YDim] := TotalProfit
    if CurrRun < NrReplications  -- Run an additional replication
```

   Take care of closing this if-statement.

9. The figure from the previous step also shows some modifications to the performance indicators *AvgAssemblyTime* and *CostConfiguration* in the *AvgExpResults* table, and the addition of a new indicator *TotalProfit*. Add these changes to your code as well. Also add a new column to *AvgExpResults* and name it *TotalProfit* with data type *real*.
10. The last thing you need to change in *EndSim* is to replace the 5 lines of code related to the worker settings (from the line WorkerSetting.create till the line setCreationTable) with the following:

```
NrSeniors := Configurations["SeniorNr", CurrExp]
NrJuniors := Configurations["JuniorNr", CurrExp]
```

11. Try to understand the changes you have made to the *Method EndSim*. To check these changes, the complete code of the revised *Method EndSim* can be found in the Appendix.
12. Open the *Method Start*.
13. Add the following code above the part that resets the experiment tables:

```
-- Set experiment type
CustomRun := true
```

14. Replace the lines of code related to resetting the workers (5 lines) by:

```
-- Reset Workers
NrSeniors := Configurations["SeniorNr", 1]
NrJuniors := Configurations["JuniorNr", 1]
```

    (later on in Section 9.2, we create a new *Method* to initialise the *WorkerPool*)

15. Create a copy of the *Method Start* (right-click on *Start* and then right-click somewhere else to remove the pop-up, then press Ctrl-C and Ctrl-V).

16. Rename this *Method* to *Start2* and change its icon. This new *Method* will be used to initialise the experiments before using the *ExperimentManager* or *GAWizard*.
17. Open the *Method Start2*.
18. Set the value of *CustomRun* to false and remove all code below the line that sets the end time of the *EventController* (code for setting the configuration, resetting the *Workers*, and starting the simulation). We will not make using of these elements as they will be carried out by the *ExperimentManager* or the *GAWizard*.
19. Add the following code to the *Method WritePerformanceData* after the lines required for code verification (this coding is required for computing the average time spent on the assembly line per engine):

```
-- Update average leadtime
AvgAssemblyTime := ((EngineFinished -1) * AvgAssemblyTime +
    (ExpResults["TotalLeadTime", EngineFinished] -
        ExpResults["TimeWarehouseWelding", EngineFinished])) / EngineFinished
```

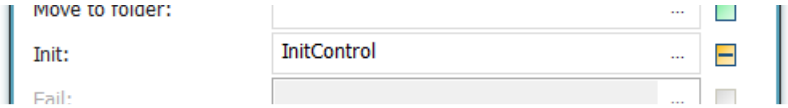
In the previous task, we have made some adjustments to the *Methods Start* and *EndSim* that impact the initialisation of the *WorkerPool*. The *WorkerPool* needs to be set before calling *EventController.Init* (so before calling the initialisation *Methods*). In the previous chapter, we solved this by setting the *WorkerPool* for the first experiment within the *Method Start* and for all subsequent experiments in the *Method EndSim* (so just before running the next experiment). This option is less suitable to be used in combination with the *ExperimentManager* and the *GAWizard* (Sections 9.2 and 9.3). Therefore, we create a so-called *InitControl*, which is a *Method* that is called at the beginning of a simulation run during the initialisation phase, but before the objects are initialised and before initialisation *Methods* are executed. As stated in the Plant Simulation help function, this option is particularly suitable to change the *Worker Creation Table* within the *WorkerPool*.

Task: Setting up an InitControl

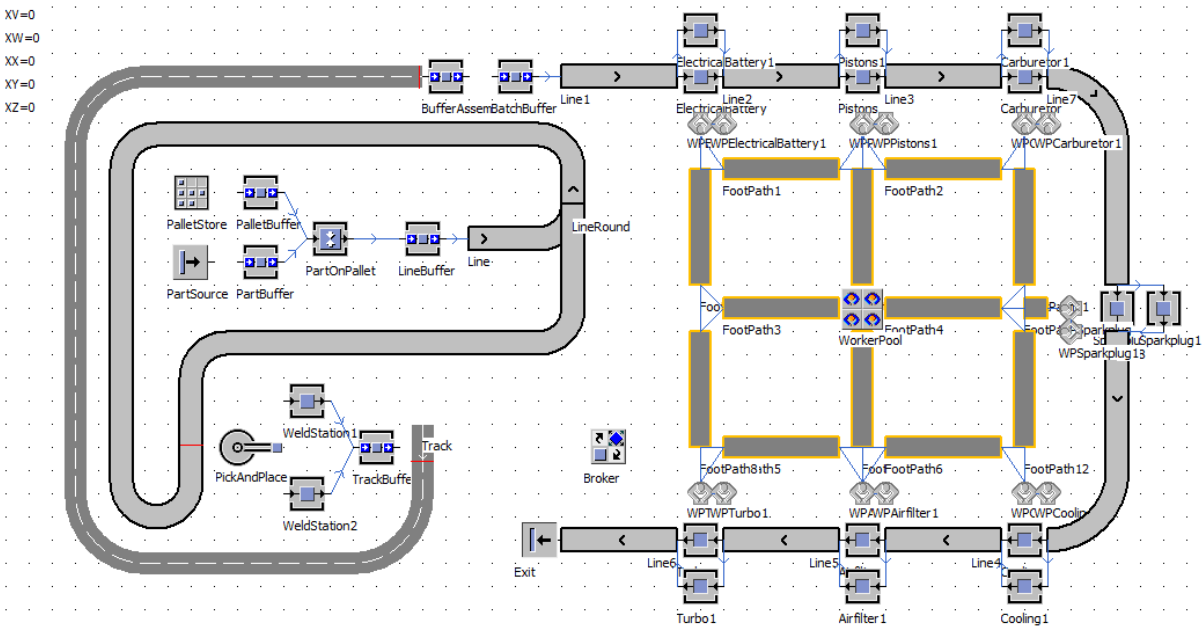
1. Create a new *Method* and name it *InitControl*.
2. Add the following code:

```
var WorkerSetting: table

-- Reset Workers
WorkerSetting.create
AssemblyStation.WorkerPool.getCreationTable(WorkerSetting)
WorkerSetting[2, 1] := NrSeniors
WorkerSetting[2, 2] := NrJuniors
AssemblyStation.WorkerPool.setCreationTable(WorkerSetting)
```

3. Open the EventController.
4. Click on *Tools > Edit Controls*.
5. Assign the *Method InitControl* to *Init*:



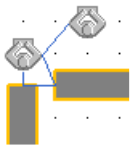
We will now start to make adjustments to the *AssemblyStation*, which requires some rearrangement of objects. The final model will look like this:



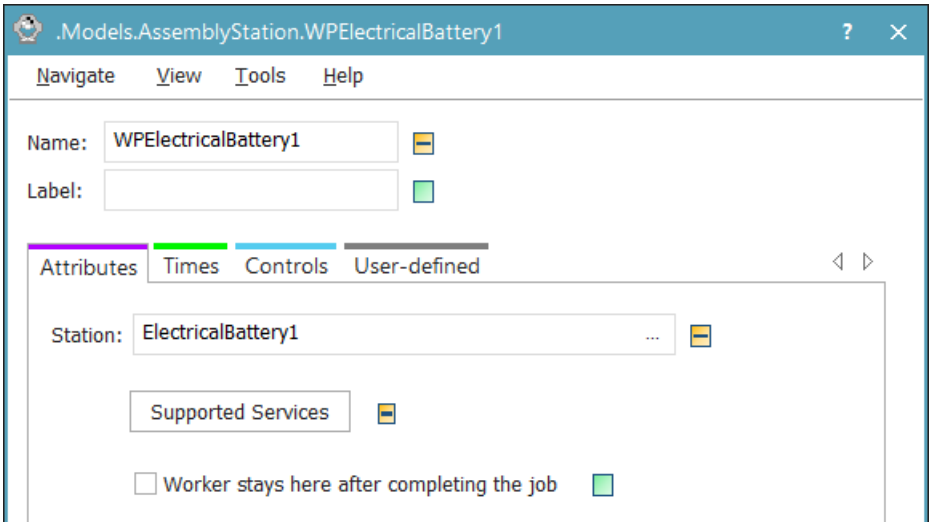
We will first add the additional *SingleProcs* and *WorkPlaces*.

Task: Additional Objects

1. Open the *AssemblyLine* from the *Class Library*.
2. Copy each of the 7 *SingleProcs* corresponding to the 7 workstations on the *AssemblyLine* (Ctrl-C) and paste them on the same *Frame* (Ctrl-V) next to the original stations (see figure above). Connect the copied stations in the same way as the original stations.
3. Copy each *Workplace* and place it next to the original workplace and make sure that they are connected.



4. Change the name of each copied *WorkStation* to *WorkStation1*, and of each copied *WorkPlace* to *WorkPlace1*. Connect the copied WorkPlaces to the copied Workstations, e.g., for WPElectricalBattery1:



5. Open the *Method ExitStation* on your *ControlPanel*.
6. Add the following code after ObjString:=?.name; and try to understand it:

```
LastSymbol := omit(ObjString, 1, strlen(ObjString) - 1)
if LastSymbol = "1"
    ObjString := copy(ObjString, 1, strlen(ObjString) - 1)
end
```

   Do not forget to add the required *Local Variable*.
7. Apply the same modification as in *ExitStation* to *EntryStation*.
8. Open the *Method NewPart* on your *ControlPanel*.
9. Multiply both the *Mu* and *Sigma* of the *LogNormal* distribution by two (resulting in parameter values 2 and 1 at most). We should be able to handle the increase in processing times given that we doubled the capacity at each work station.

We now only need to add the constraint of releasing products in batches to the assembly line.

Task: Releasing the Products in Batches

1. Make room between the *Welding Line* and *Assembly Line* (see the figure of the final model).
2. Remove the connection between the *BufferAssemblyLine* and *Line1*.
3. Add a *Buffer* to the *Frame AssemblyStation* and name it *BatchBuffer*, use a *Capacity* of -1 and a *Dwell Time* of 0.
4. Connect the *BatchBuffer* with the first line of the *Assembly Line*.
5. Add five variables to the *Frame AssemblyStation* and name them respectively *XV, XW, XX, XY* and *XZ* with data type *Integer*.
6. Add the following code to the *Method Reset* in the *ControlPanel*:

```
                                AssemblyStation.XV := 0
                                AssemblyStation.XW := 0
                                AssemblyStation.XX := 0
                                AssemblyStation.XY := 0
                                AssemblyStation.XZ := 0
```

7. Insert a new *Method* to the *ControlPanel* and name it *CheckBatch*.
8. Add the following code to *CheckBatch* (see also code in the Appendix):

```
/* This Method will check whether all the Batch Items are
   present within the Buffer and if the products can be
   released on the assembly line. */
-----------------------------------------
var y, i, j: integer

str_to_obj(sprint("AssemblyStation.",@.Cont.Name)).value :=
str_to_obj(sprint("AssemblyStation.",@.Cont.Name)).value + 1

if AssemblyStation.XV >= 6 and
    AssemblyStation.XW >= 5 and
    AssemblyStation.XX >= 1 and
    AssemblyStation.XY >= 3 and
    AssemblyStation.XZ >= 5

    for i := 1 to GAProductSequence.YDim
        j := 1 -- initialise loop over Buffer content
        while GAProductSequence["ObjectType",i] /=
                AssemblyStation.BufferAssemblyLine.MU(j).Cont.Name
            j := j + 1
        end
        AssemblyStation.BufferAssemblyLine.MU(j).Move(AssemblyStation.BatchBuffer)
    next

    AssemblyStation.XV := AssemblyStation.XV - 6
    AssemblyStation.XW := AssemblyStation.XW - 5
    AssemblyStation.XX := AssemblyStation.XX - 1
    AssemblyStation.XY := AssemblyStation.XY - 3
    AssemblyStation.XZ := AssemblyStation.XZ - 5
end
```

   Note that part of this code is required for the *GAWizard*, which will be become clear in Section 9.3.

9. Set this *Method* as *Entrance Control* for the *BufferAssemblyLine*.
10. Add a *TableFile* to the *ControlPanel* and name it *GAProductSequence*.
11. Define for the *TableFile GAProductSequence* its first column's data type as a *String* and give it the name *ObjectType*.
12. Fill the column *ObjectType* subsequently with the following data:
    - 6 rows containing XV
    - 5 rows containing XW
    - 1 row containing XX
    - 3 rows containing XY
    - 5 rows containing XZ

13. In Assignment B2 at the end of the previous chapter, you have changed the arrival rate of the *PartSource*. Set the mean time between part arrivals back to the original value of 3 minutes.

We have now adjusted our model completely to make use of the *ExperimentManager* and the *GAWizard* on which we will elaborate in the next sections. But first, let's validate your model.

Task: Validating your model

1. To validate your model, set the *WarmUpLength* to 1 day, the *RunLength* to 10 days, and the number of replications to 5.
2. Validate your results using a configuration with 4 seniors and no juniors. To avoid running all the configurations, you can temporarily change the *Method SetConfiguration*, by changing the lower and upper bounds for the variables i and j.
3. Run your model by pressing the *Method Start*, and check if there are any unexpected bugs. The results for this configuration should be approximately:
   - *EngineCreated* = 4800 ((86400/180)*10)
   - *EngineFinished* = 4800 (should be able to process all incoming engines)
   - *AvgAssemblyTime* = 4444
   - *CostConfiguration* = 40,000 ((0*500 + 4*1000)*10)

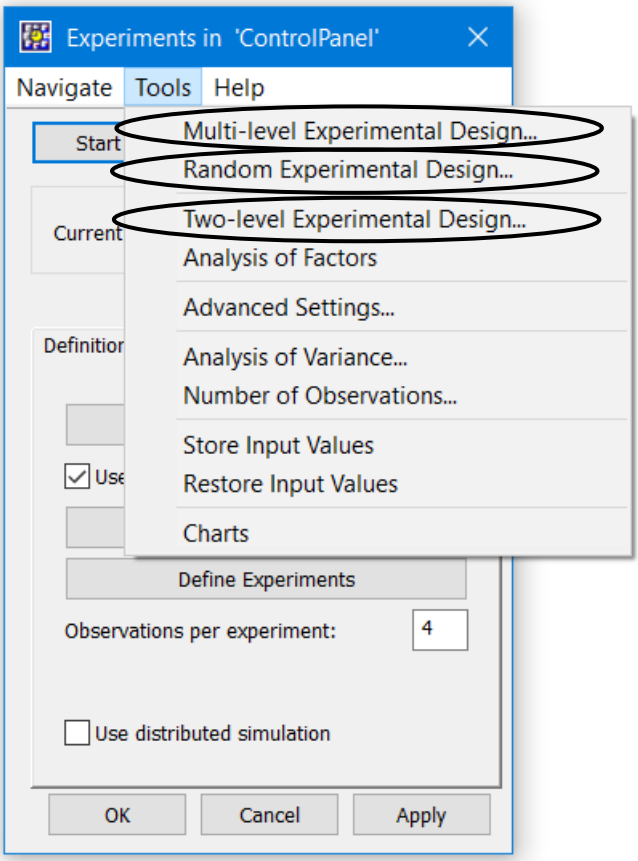
9.2 The ExperimentManager

The *ExperimentManager* has been introduced already in Chapter 5. Just like the model in this chapter, the model of Chapter 5 had counters, input variables, and certain output values. You defined your own experiments by setting different values for the input variables in Section 5.8. In the case of just one factor, with a limited number of possible values, it might be easy to define all experiments yourself. However in the case of multiple factors, the required number of experiments might grow very large very fast, especially when there is an unclear interaction between input variables, not to mention the use of continuous variables instead of discrete variables. The *ExperimentManager* can help you defining all the required experiments. There are three possible options we will review in this section. But first we need to add the *ExperimentManager*.

Task: Setting up the ExperimentManager

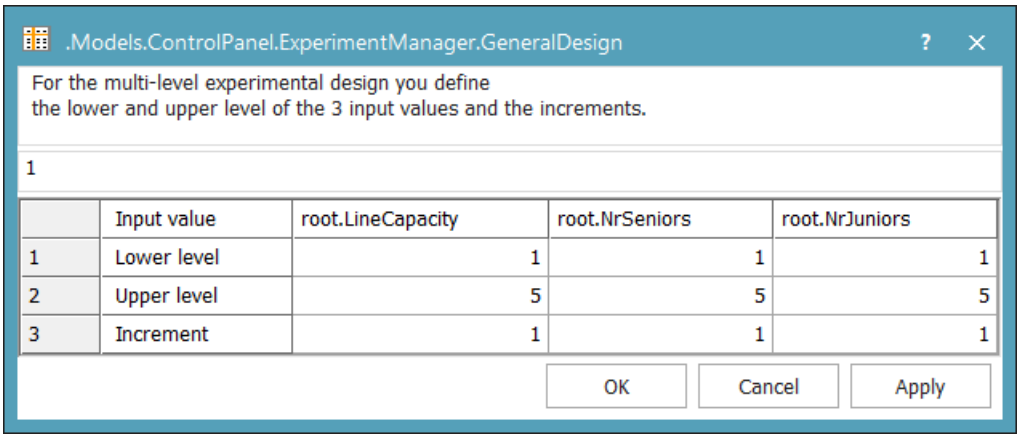
1. Add the *ExperimentManager* to your *ControlPanel.*
2. Set *LineCapacity*, *NrSeniors*, and *NrJuniors* as *Input Variables.*
3. Set *AvgAssemblyTime* and *CostConfiguration* as *Output Values*.

The ExperimentManager now defines the setting for *LineCapacity, NrSeniors*, and *NrJuniors*, and is ready to be used. There are several options for this, which we will not demonstrate in full detail since they speak for themselves. The three most interesting options are shown in the figure below.



The *Multi-level Experimental Design* option lets you specify:

- A lower bound for the input value
- An upper bound for the input value
- An increment size for every input variable



	Input value	root.LineCapacity	root.NrSeniors	root.NrJuniors
1	Lower level	1	1	1
2	Upper level	5	5	5
3	Increment	1	1	1

After you clicked OK, the *ExperimentManager* generates every possible experiment given your input. A pop-up appears showing a warning that new experiments are defined and that previous experiments will be overwritten, if you have defined experiments already. The *Random Experimental Design* also lets you enter a lower and upper bound, but not the level of increment, since all values will be determined at random within the defined ranges. The Two-level Experimental Design only creates experiments for a minimum and maximum value of every factor. This design is also denoted by a 2^k-factorial design.

9.3 The Genetic Algorithm

As shown in the previous section, multi-level experiments can quickly lead to a huge number of simulation runs. Especially when considering all possible combinations of a large number of input factors. The number of alternative configurations to simulate and compare could then literally be in the hundreds of thousands. By the application of the so-called *Genetic Algorithms* (GA's), the absolute number of experiments carried out is reduced considerably while there is still a chance of finding good solutions. GA is a biology-based general optimisation technique and a particular class of the evolutionary computation algorithms. The origin of this algorithm is the process of natural selection: individuals who adapted best to their environment, procreate the best. GA helps us to decide what alternative system configurations to simulate as well as how to evaluate and compare their results.

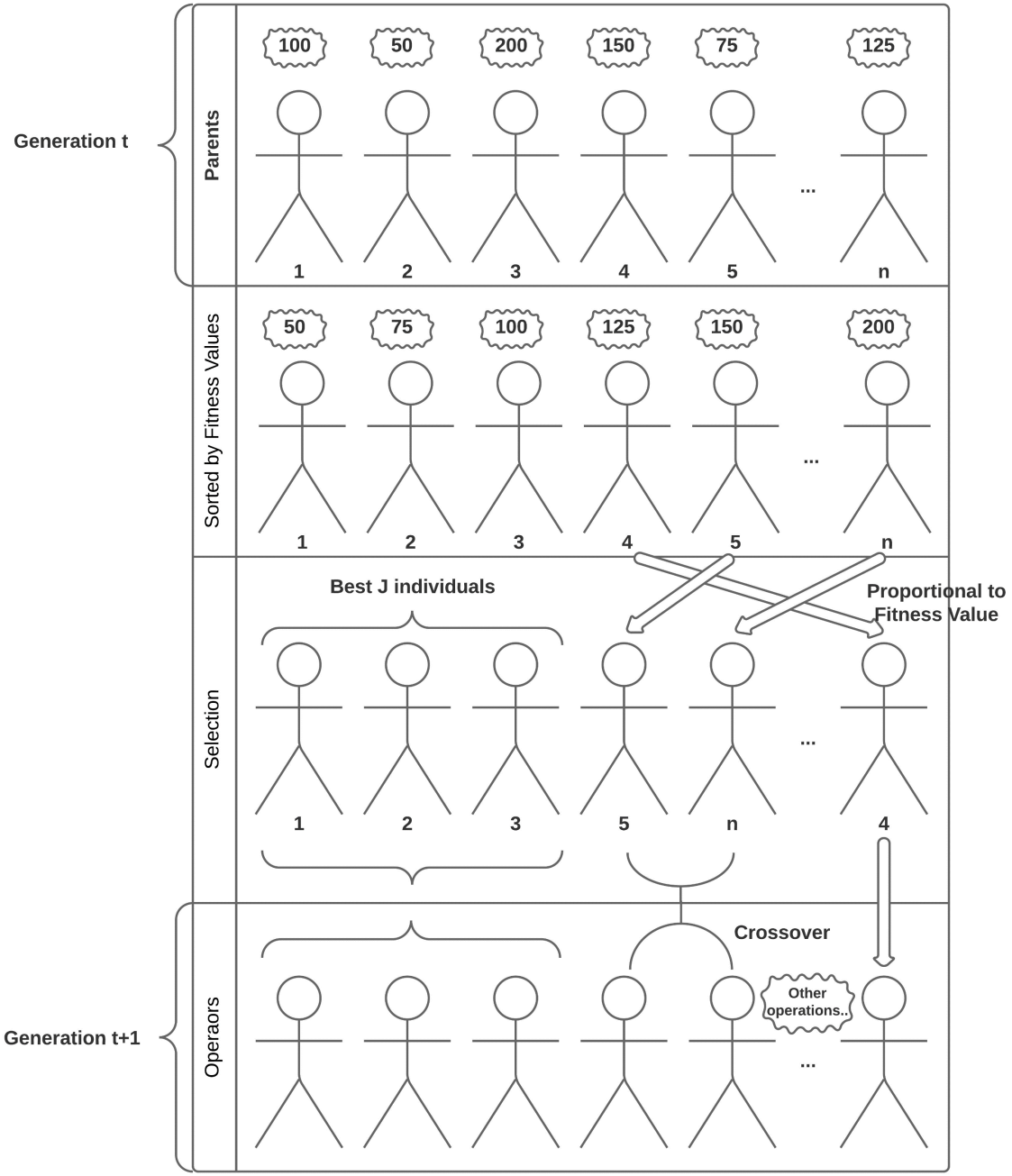
In the upcoming paragraphs, we will start with describing the principles of GAs in more detail and pinpoint characteristics of well-suitable problems to deal with by GAs. Subsequently, we extend the model we created so far by implementing the *GAWizard* of *Plant Simulation* to a new arising optimisation problem. In the model we restrict ourselves to implementing only one optimisation problem, although the *GAWizard* can be used for more types of optimisation problems as we discuss later on.

The GA algorithm starts with an initial population of solutions (i.e., configurations of input parameters). A solution together with the corresponding solution value, is called an *individual*. Solely the solution, so without solution value, is called a *chromosome*. The solution value of an individual is denoted by *fitness value*. The population of defined individuals form a so-called *generation*. By considering the fitness values of all individuals within a generation, the algorithm performs all kind of (on nature based) operations to create a new generation. These new solutions are denoted by *offspring*, while the individuals of the previous generation are denoted by *parents*. Obviously the higher the fitness value of an individual, the more chance is has to reproduce.

Offspring is formed by using several *operators*. From (most of the time) high to low probability of occurrence possible, these operators are:

- Selection, the fitter an individual solution the more times it is likely to be selected to reproduce
- Crossover, a random exchange between two chromosomes to create offspring
- Inversion, inverts a chromosomes' sequence within a defined range
- Mutation, a random flip within a chromosome

The size of individuals in a generation is limited. As a result, the parent generation dies and the offspring generation is becoming the new parent generation. The figure below illustrates the basic evolution idea graphically.



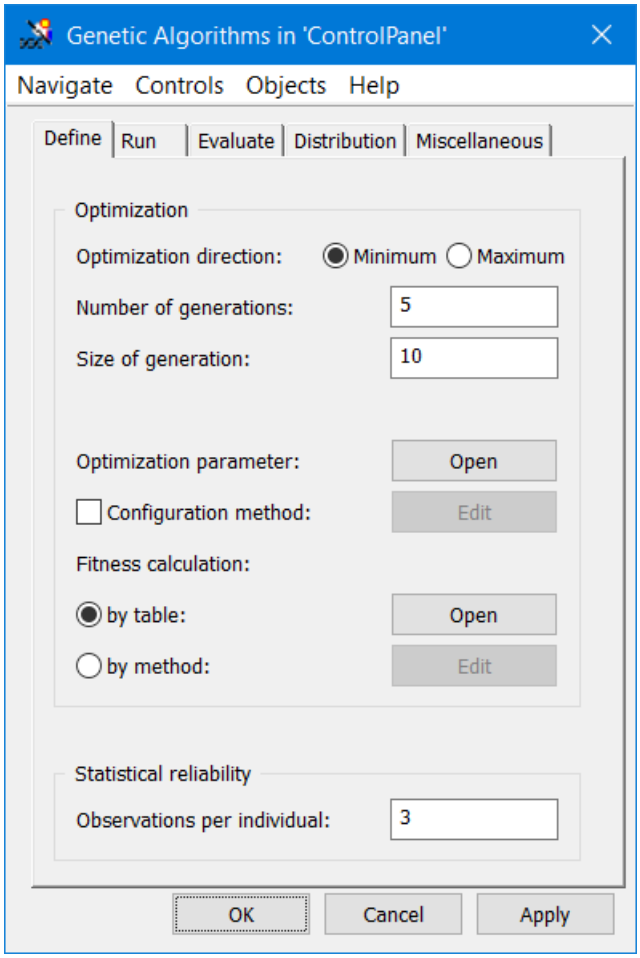
Usually, the process of creating new generations is done until one of the following conditions is met:

- Tolerance threshold is reached
- Maximum number of generations has passed
- Maximum amount of computational time has passed
- Fitness value has plateaued

GAs provide good solutions for complex problems but do not guarantee that a global optimum will be found. GAs are less suitable for optimisation problems that have a few narrow peaks (good solutions) and a lot of solutions with almost similar fitness values, since the fitness does not direct the algorithm towards the peaks in most of the cases. On the other hand, GA will be effective in situations with multiple peaks with similar fitness values. As mentioned in the Plant Simulation help files (Add-Ins Reference Help > Genetic Algorithms > Genetic Algorithms and Simulation > Optimization Tasks), typical problems for which GAs are considered to be a suitable solution approach are characterised by:

- Complex and large solution space, such as combinatorial optimisation problems
- Unknown characteristics of the solution space
- Discontinuities within the solution space that do not allow for mathematical-numerical optimisation
- Many soft restrictions, which are restrictions that do not necessarily have to be kept, but lead to worse results

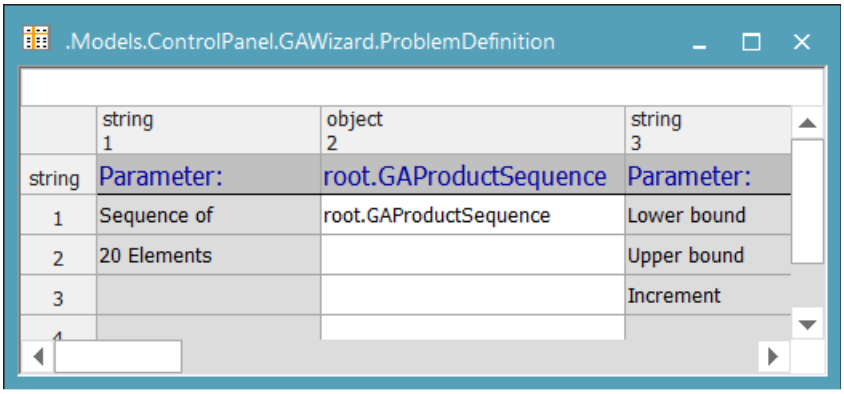
Now that we know the basics about GA and the *GAWizard*, we will implement the *GAWizard* in our model. The dialog window of the *GAWizard* has the following interface:



In the next task we will set the correct settings for determining what the best sequence is for releasing our products (within the batch) to the *Assembly Line*.

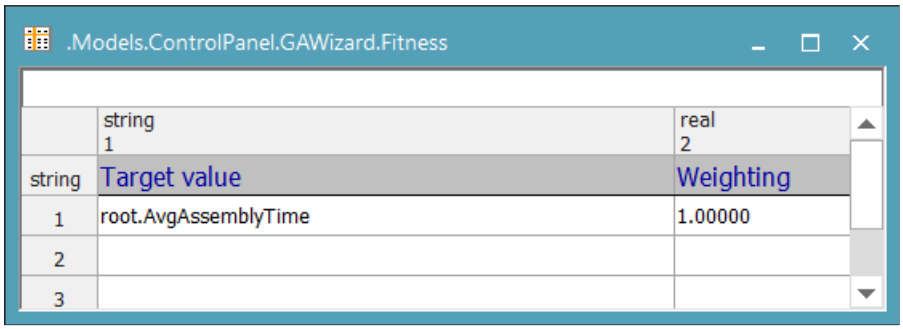
Task: Implementing the GAWizard

1. Add the *GAWizard* (from the *Class Library* under *Tools*) to the *ControlPanel*.
2. Input and output factors can be imported for the *GAWizard* in a similar manner as for the *ExperimentManager*. Hold down the Shift-Key and drag-and-drop the *GAProductSequence* table to the *GAWizard*. If we now open the *Sequence* table, we can see that there are two columns added: *Origin* and *Chrom*. The column *Origin* consists of the original row sequence and the column *Chrom* contains the sequence which belongs to one particular individual. We show later on that the *GAWizard* changes this sequence.
3. Open the *GAWizard* and click on *Open* next to *Optimization parameter*. A table is shown that presents the problem definition for the *GAWizard*. This table should look as follows:



	string 1	object 2	string 3
string	Parameter:	root.GAProductSequence	Parameter:
1	Sequence of	root.GAProductSequence	Lower bound
2	20 Elements		Upper bound
3			Increment

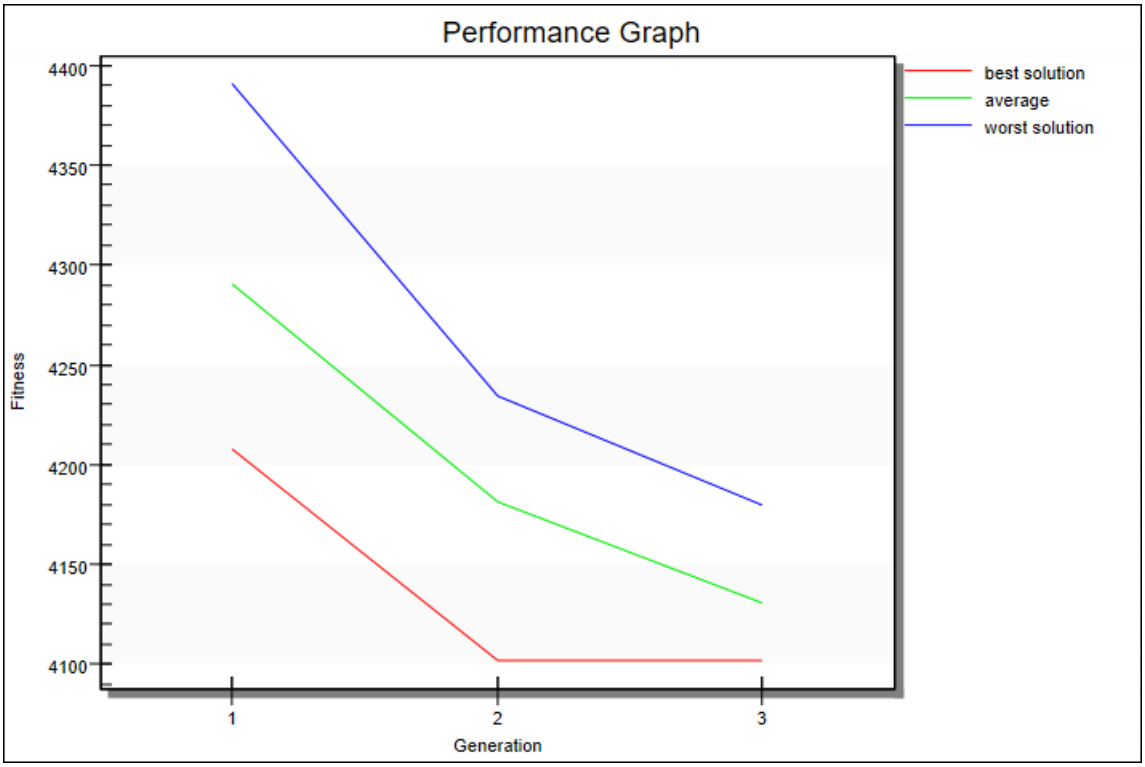
4. Drag-and-drop the *AvgAssemblyTime* to the *GAWizard* (without holding down the Shift-Key). Check if this variable is included within the fitness calculation by opening the fitness calculation table within the *GAWizard*:



	string 1	real 2
string	Target value	Weighting
1	root.AvgAssemblyTime	1.00000
2		
3		

5. Open the *GAWizard* and set the number of generations to 3, size of generation to 4, and the number of observations per individual to 3. Leave the optimisation direction to minimum and press the *Apply*-button. Note that we set the GA-parameters relatively low on purpose to illustrate the idea. Note that if you get an error stating "Wrong HTML directory" after pressing the *Apply*-button, go to the tab *Miscellaneous* and uncheck the box before "Save the report after the experiment run".
6. The GAWizard only optimises the product sequence. You need to set the other experimental factors by hand. Set the value of the variables *NrJuniors*, *NrSeniors,* and *LineCapacity* to 0, 4, and 2 respectively.
7. First click the *Method Start2* to reset the experiments. Next, go to tab *Run* of the *GAWizard*, click *Reset*, and then *Start* to start the optimisation. It could take a few minutes before the *GAWizard* is finished.

8. After the *GAWizard* is finished, a dialog shows the time and you can analyse the behaviour of the parameters in the HTML-report (press the button *Show* under the tab *Evaluate*). You should see a figure similar to this:



The best input parameters are set into the model, the corresponding solution can be found in the table *GAProductSequence*. If you want to get back to the initial settings, just press the *Reset*-button in the *GAWizard* on the tab *Run* (the sequence from the column *Origin* will be restored).

We can stop the optimisation run after the individuals of the active generation have been completely evaluated and then modify the settings and the parameters of the object *GAWizard*. In that way we can decide after a while, for instance, to add more generations or to increase the probability of a cross-over. To stop the simulation run during the GA-optimisation, we can click on *Stop* on the tab *Run*. As long as there are still individuals evaluated of the current investigated generation, the button shows *Wait*. When these evaluations are done, we can modify the parameters and press the *Start* button again.

Did you know?

In the first generation, Plant Simulation evaluates the number of generations that you have entered in the wizard as the size of the generation. In each of the subsequent generations, the *GAWizard* evaluates twice as many individuals. In addition, you may also need multiple observations for evaluating the fitness values appropriately. When, for example, you use 5 observations for evaluating an individual and create 100 generations with a size of generation of 30, Plant Simulation has to execute 5970 simulation runs (5* (30 + 2 * 30 * 99)).

As you might have noticed, you can change a lot of settings in the *GAWizard*. The changeable aspects are not limited to setting the number of generations, size of generations, and observations per individual only. Concerning the output (defined in the fitness calculation), you can either have a *TableFile* defined with your factors or create a *Method* of your own to define an output factor. In the *TableFile* you can give output factors weights. For the input factors, you can also use a *TableFile* or *Method*.

To modify the GA-settings, such as the selection rules, you can open the *GAWizard*, open the tab *Objects* and go to *GA Control*. If you want to adjust more advanced settings of the GA, you can either open the *Controls* tab in the *GAWizard* or press the ALT-key while double-clicking on the *GAWizard* in your *Frame*. The latter option reveals the underlying *Frame* of the *GAWizard*. It requires some more understanding of how the GA works, but to make modifications you should look at the *GAAllocation*, *GAOptimization* and *GASequence* (hidden in *SeqLoc*) objects.

Did you know?

Plant Simulation contains a myriad number of interesting examples about working with the *GAWizard*. To examine these examples, open *Plant Simulation* to show the *Start Page*. In this page you select *Examples/Infos* under the *Getting Started* headline. Scroll down to the *Concise Modeling Examples* and click the *Examples collection*. Choose as category *Tools and optimization*, with as topic *Genetic algorithms*. Several examples are exposed there. In addition, in the *Examples/Infos* screen the *Due Date Optimization* model presented under *Example Models* also makes use of the *GAWizard*. For your simulation models, you might get inspired by these examples.

The optimisation task you did can be described as a *sequencing task*. The *GAWizard* can be used for *allocation tasks* and *selection tasks* as well. When you drag-and-drop objects into the *GAWizard*, it automatically knows which kind of task it should perform. In a sequence task, a sequence is varied (stored in a list) for each experiment to determine an optimum. In addition, you can define position restrictions for each item in the sequence by entering the permitted positions for each item in the object *GASequence* within the *GAWizard*. For instance, you can define that product X must be the first in the sequence in all the cases. An allocation task assigns numerical values between minimum and maximum values. An example of these kind of tasks is the buffer size allocation. You can easily insert these input factors by drag-and-drop, manually inserting via the optimisation parameter table, or through configuring your own *Method*. The selection optimisation task works almost similar to the allocation tasks; however, now there is a predefined number of alternative input factors that do not necessarily have numerical values but could also be categorical. For instance, you can drag-and-drop (with shift-key) a *Sorter* object into the *GAWizard* and choose the *Order*-property. Then you have to define the possible values it can take in the input table. Also more advanced (combinatorial-optimisation) tasks can be implemented as selection tasks. Ultimately, the main advantage of the *GAWizard* is the wide applicability and ease of implementation.

Did you know?

In principle you can define both sequence and allocation tasks as selection tasks. This has advantages, but also some drawbacks. We leave it up to you to investigate this difference by exploring the Plant Simulation help documentation. We recommend that if you have (simple) sequence or allocation tasks, you should implement these as such tasks and not as selection tasks.

9.4 Simulation Optimisation

In Simulation Optimisation, we search for the best settings of experimental variables given a limited budget regarding computation time. Regarding the latter, we want to spend our computation time wisely by exploring the whole search space without spending too much time on non-promising settings.

Up to this point, we designed three ways of performing experiments:

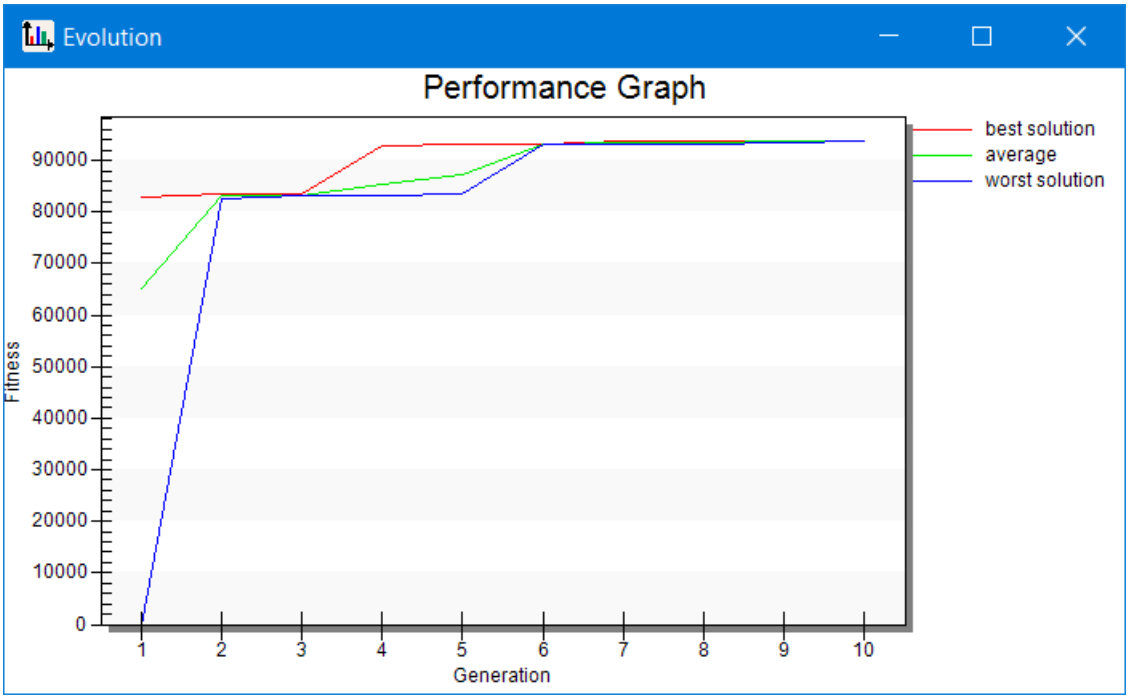
1. Experimentation using the *Method EndSim* in combination with a table *Configurations*.
2. Experimentation with the *ExperimentManager*.
3. Experimentation with the *GAWizard*.

In the first two options, we defined all experiments beforehand, irrespective of their solution quality. In the last option, the experiments are not predetermined, but the search is directed towards promising solutions throughout the generations. However, the third option still assumes an equal run length and equal number of replications per experimental setting. Obviously, it would make sense to let this depend on the quality of the solutions found so far. For example, once we could say with a certain confidence that a given setting is worse than an earlier evaluated setting, we could stop simulating this setting. Another thing we did not consider so far was optimizing over all experimental settings simultaneously. In the first two options, we only varied the number of workers. In the third option, we only optimised over the assembly sequence.

In the remainder of this section, we show (i) how the third option can be extended by adding the remaining experimental factors, and (ii) how the first option can be used to determine the experimental settings on the fly.

Task: Extending the GAWizard

1. Hold down the Shift-Key and drag-and-drop the variables *NrJuniors* and *NrSeniors* to the *GAWizard*.
2. Change the objective value in the *GAWizard* by opening the table under *Fitness calculation* on the tab *Define* and replacing *AvgAssemblyTime* with *TotalProfit*.
3. On the same tab, set the *Optimization direction* to *Maximum* (confirm using *Apply*).
4. Open the *Method EndSim*.
5. Change the line of code to compute the *TotalProfit* by making sure negative values are set to zero, by using max(0,…), because the *GAWizard* is not able to handle negative values.
6. Run the *GAWizard*. When running GA for 10 generations, with generation size 5 and observations per individual 5 (confirm using *Apply*), the results could be something like this (running time approximately 10 minutes):



Experimentation with the *GAWizard* can also be performed in stages. For example, using the setup from the previous task, you could first perform a run for, say, 10 generations. Then look at the resulting best sequence from the table *GAProductSequence*, copy this sequence (first column) to memory, reset the *GAWizard* (the original sequence will then be placed in the table) and copy back the sequence from memory. In a second stage, you then start with a better initial solution. In addition, you could fix some of the values to a smaller search domain (e.g., number of workers between 2 and 4).

Task: Create a custom Simulation Optimisation Method

1. Open the *Method Start*.
2. Comment all lines of code that use the *Method SetConfiguration* or *TableFile Configurations*, since we no longer use predefined configurations.
3. Set initial values for *NrJuniors* and *NrSeniors* in the *Method Start*:

```
-- Reset Workers
NrSeniors := 1
NrJuniors := 1
```

4. Create a new *Method* and name it *SimOpt*.
5. Open the *Method EndSim*.
6. Remove the whole code (Ctrl-X) below the computation of *TotalProfit* that is made conditional on *CustomRun* and paste (Ctrl-V) the code into the *Method SimOpt*. Now call the *Method SimOpt* from the place where you removed the code, such that the remaining code for *EndSim* looks as follows:

```
var WorkerSetting: table

-- Stop EventController
EventController.Stop

-- Perform general computations
if CustomRun
    AvgAssemblyTime := ExpResults.meanValue({"TotalLeadTime", 1}..{"TotalLeadTime", *}) -
        ExpResults.meanValue({"TimeWarehouseWelding", 1}..{"TimeWarehouseWelding", *})
end

CostConfiguration := (NrJuniors * JuniorCost + NrSeniors * SeniorCost) * (RunLength / 86400)
TotalProfit := max(0, EngineRevenue * EngineFinished - EngineCost * EngineCreated -
    CostConfiguration - (EngineLead * EngineFinished * AvgAssemblyTime / 86400))
    -- Divide by 86400 to multiply the cost with the number of days

-- Store Data in case of CustomRun
if CustomRun
    SimOpt
end
```

In principle, your model should still run as before, except that it starts with given initial settings and that after the running the required replications, it uses the empty table *Configurations* to set the next configuration.

7. Add a *TableFile* and name it *SOresults*.
8. Give the table the following format:

	integer 1	integer 2	real 3	real 4	table 5
string	NrJuniors	NrSeniors	ValueMean	ValueStdev	AllValues
1					

Where the data type *Table* of column 5 consists of 1 column of data type *Real*.

9. Open the *Method Start*.
10. Add code to empty the table with Simulation Optimisation results (SOresults.delete) just below the code for emptying the other tables.
11. Add the following code to *SimOpt* just after updating the table *AvgExpResults* and try to understand its purpose:

```
NotFound := true
for i := 1 to SOResults.YDim
    if SOResults["NrJuniors", i] = NrJuniors and
        SOResults["NrSeniors", i] = NrSeniors
        NotFound := false
        SOResults["AllValues", i][1, SOResults["AllValues", i].YDim + 1] := TotalProfit
        SOResults["ValueMean", i] := SOResults["AllValues", i].meanValue({1,1}..{1,*})
        SOResults["ValueStdev", i] :=
            SOresults["AllValues", i].standardDeviation({1, 1}..{1, *})
    end
next
if NotFound
    SOresults.create("AllValues", SOresults.YDim + 1)
    SOresults["NrJuniors", SOresults.YDim] := NrJuniors
    SOresults["NrSeniors", SOresults.YDim] := NrSeniors
    SOresults["ValueMean", SOresults.YDim] := TotalProfit
    SOresults["ValueStdev", SOresults.YDim] := 1000000
    SOresults["AllValues", SOresults.YDim][1, 1] := TotalProfit
end
```

12. Change the code in *SimOpt* that sets the number of workers in the following way:

```
NrSeniors := floor(z_uniform(1, 1, 6))
NrJuniors := floor(z_uniform(1, 1, 6))
```

13. Open the *Method Reset*.
14. Let the *RandomNumbersVariant* depend on the variable *RunCounter* instead of *CurrRun*:

```
if CustomRun then
    EventController.RandomNumbersVariant := RunCounter;
end;
```

    This change is necessary to avoid using the same configuration over and over again due to the previous use of Common Random Numbers.

15. Set the variable *TotalExp* on the *ControlPanel* to 25.
16. Run your model by clicking the *Method Start*.

The simulation optimisation procedure developed in the previous task is quite simplistic in the sense that it randomly generates the next configuration, without considering the value of configurations already executed. However, the code in *SimOpt* can be extended quite easily to let the values of *NrJuniors* and *NrSeniors* depend on earlier results. For example, we can evaluate the different neighbours of the current solution by adding/removing a single worker, then set the best configuration as current solution and repeat the procedure.

Another extension would be to make the number of replications variable. For example, in first instance we could perform a limited number of replications per configuration. After a while we could spend more replications on the more promising configurations to increase our confidence in their value.

A final extension is the use of a flexible stopping criterion. Currently, we use a fixed setting for the number of configurations to consider (25). An alternative would be to let it depend on the convergence of results. For example, when the improvements in found solution values are limited, or no improvement have been found during the past simulation runs, we might decide to stop.

9.5 Assignment B3: Simulation Optimisation

In this chapter, several ways of performing experiments and finding good configurations were introduced. However, we only briefly explored their use without considering all options available. Furthermore, we only considered a limited set of experimental factors. In this assignment, you are asked to apply the three ways of experimenting, i.e., using the *ExperimentManager*, the *GAWizard*, and your own Simulation Optimisation approach, more extensively to provide recommendations to the car manufacturer CeeCar Inc. regarding the optimal design of the assembly line.

Assignment B3.1: Use the *ExperimentManager* to perform experiments of the following types: multi-level experimental design (full factorial), random experimental design, and two-level experimental design (2^k-factorial design). For the 2k-factorial design, use *TotalProfit* as reset value: the value for which the main effects and two-way interaction effects are calculated. The latter effects can be found in the *ExperimentManager* under *Tools > Analysis of Factors*. Compare the usefulness of the three different types of experiments and specifically argue why a 2^k-factorial analysis is less appropriate here. Repeat the 2^k-factorial analysis using *AvgAssemblyTime* as reset value and reflect on the outcomes.

Assignment B3.2: Use the *ExperimentManager* in combination with the *GAWizard* to evaluate a whole range of possible experimental settings considering the factors *NrJuniors*, *NrSenior*, *LineCapacity*, and arrival rate of the *PartSource*. The logic of varying the arrival rate (interarrival times or engines) is that it is not directly clear what setting would result in the best performance: a higher arrival rate might result in higher revenues from selling the engines, but also in higher employee costs, and vice versa. To do this, first use the *ExperimentManager* to get an idea of a promising search area. Use the resulting search area as input for the *GAWizard*.

Assignment B3.3: Create your own Simulation Optimisation approach by extending the *Method SimOpt* (possibly in combination with extending the *TableFile SOresults* and adding additional *Methods* and *TableFiles*). In this extension, the decision regarding the next configuration to consider (to simulate) should depend on the results of previously evaluated configurations. Optionally, you may even go further by using a variable number of replications depending on the values of the configurations and use a dynamic stopping criterion, as mentioned at the end of Section 9.4.

Assignment B3.4: Provide advice to CeeCar Inc. regarding the optimal design of their assembly line, taking into account the different criteria that might be relevant for their final investment decision. Write down your analysis and provide compelling figures to support CeeCar Inc. in their multi-criteria decision making.

Deliverables:

- The model from Assignment B3.3.
- A report consisting of a description of your approach used in Assignments B3.1-B3.3, explanation of your Simulation Optimisation approach from Assignment B3.3, the results from Assignments B3.1-B3.3, and an in-depth analysis of the experimental results, supported by tables and figures (Assignment B3.4).

Appendix

Code: NextPatient from Section 3.9

```
-- If the patient enters a waiting room, check if there is room at the
-- respective GP. If so, push the patient to the GP.
if ? = AdultWR and not AdultGP.Occupied
        @.move(AdultGP)
elseif ? = ChildWR and not ChildGP.Occupied
        @.move(ChildGP)
end

-- If this is an exit event from a GP, then move the patients to the exit.
-- Also check whether a new patient can be pulled to the GP.
if ? = AdultGP
        @.move(Departure)
        if AdultWR.Occupied
            AdultWR.MU(1).move(AdultGP)
        end
elseif ? = ChildGP
        @.move(Departure)
        if ChildWR.Occupied
            ChildWR.MU(1).move(ChildGP)
        end
end
```

Code: EnterGPs from Section 4.2

```
var now:              time
var urgency:          integer
var timeSinceApp:     time

UpdateLocation(@, ?)

-- Define variables.
now                   := EventController.SimTime
urgency               := Patients["urgency", @]
timeSinceApp          := max(0, now - max(Patients["arrivalTime",@], Patients["appointmentTime", @]))

-- Calculate and save statistics.
Patients["waitingTime", @] := now - Patients["arrivalTime", @]
Patients["arrivalAtGP", @] := now

if timeSinceApp <= 0
        Patients["perceivedWaitingTime", @] := 0
elseif urgency = 1
        Patients["perceivedWaitingTime", @] := timeSinceApp * 2.0
elseif urgency = 2
        Patients["perceivedWaitingTime", @] := timeSinceApp * 1.0
elseif urgency = 3
        Patients["perceivedWaitingTime", @] := timeSinceApp * 0.5
end
```

Code: GetPatient from Section 4.4

```
->object
var i,bestUrgency: integer
var winner: object
var bestAppointmentTime: time
```

```
-- Initialize worst case
winner                  := void
bestUrgency             := 4
bestAppointmentTime     := EventController.End+86400

/*
Loop through patients to determine the patients with the highest
urgency and appointmentTime
*/

for i := 1 to Patients.YDim
    if Patients["location", i] = WaitingRoom and
        Patients["urgency", i] <= bestUrgency and
        Patients["appointmentTime", i] <= bestAppointmentTime

        winner                  := Patients["object", i]
        bestUrgency             := Patients["urgency", i]
        bestAppointmentTime     := Patients["appointmentTime", i]
    end
next

-- Return the patient.
result := winner
```

Code: MakeAppointment from Section 5.5

```
-- MakeAppointment
-- Purpose:   Determine the next available appointment slot and assign it to a patient.
-- Called by: FrontDesk (Entrance)

-- Calculate the appointment time for the patient.
if LatestAppDay < CurrentDay
    LatestAppTime := OpeningTime
    LatestAppDay  := CurrentDay
else
    LatestAppTime += (ClosingTime - OpeningTime) / SlotsPerDay
end

-- If the appointment time is after closing time, then appoint the patient
-- to the subsequent day.
if LatestAppTime >= ClosingTime
    LatestAppTime := OpeningTime
    LatestAppDay  += 1
end

-- Set the patient information.
@.appointmentTime := latestAppDay * 86400 + latestAppTime
@.callTime       := EventController.SimTime
@.waitingAtHomeTime.increment(@.appointmentTime - EventController.SimTime)

-- Let the patient arrive at the appointed time.
&OnAppointment.methCall(@.appointmentTime - EventController.SimTime, @)
```

Code: EndDay from Section 5.7

```
-- EndDay
-- Purpose: Increment the day counter and move patient to FrontDesk
-- Called by: LateEvening
var i: integer
var patient: object
```

```
-- Increment the day counter
CurrentDay += 1

-- Move any patient that is left in WaitingRoom to FrontDesk,
-- such that it will reschedule.
for i := 1 to WaitingRoom.NumMU
	patient := WaitingRoom.MU(1)
	patient.waitingRoomTime.increment(EventController.SimTime - patient.appointmentTime)
	patient.reschedules.increment
	patient.move(CallQueue)

	nReschedules += 1
next
```

Code: NewPart from Section 8.1

```
var i, Snr: integer

@.move(AssemblyStation.PartBuffer)
if AssemblyStation.PalletStore.NumMU > 0
		AssemblyStation.PalletStore.MUPart(1).move(AssemblyStation.PalletBuffer)
end

Snr := 0

for i := 1 to ProcTimes.indexYDim
		if ProcTimes[@.Name, i] > 0
			Snr += 1
			@.setAttribute("PT" + ProcTimes[0, i],
				(60 * z_lognorm(Snr, 1 * ProcTimes[@.Name, i],
					0.5 * ProcTimes[@.Name, i])))
		else
			@.setAttribute("PT" + ProcTimes[0, i], 0)
		end
next
```

Code: SensorCheckTrack from Section 8.2

```
param SensorID: integer, Front: boolean

if SensorID = 1
		if Assemblystation.TrackBuffer.Empty
			@.Backwards := false
			@.Stopped := true
		else
			Assemblystation.Trackbuffer.MUPart(1).Move(@)
			@.Backwards := false
		end
elseif SensorID = 2
		if @.Empty
			@.Backwards := true
		else
			@.cont.cont.TimeWarehouseWelding := EventController.SimTime -
				@.cont.cont.CreationTime
			@.cont.cont.WTElectricalBattery := EventController.SimTime
			@.cont.Move(Assemblystation.BufferAssemblyLine)
			@.Backwards := true
		end
end
```

Code: WritePerformanceData from Section 8.6

```
var i: integer
var CheckTotalLeadTime: real

if EventController.SimTime > WarmUpLength

	-- Update performance
	@.WTExit := root.EventController.SimTime - @.WTExit
	EngineFinished += 1

	for i := 1 to 15
		ExpResults[i, EngineFinished] := @.getAttribute(ExpResults[i, 0])
	next

	ExpResults["TimeWarehouseWelding", EngineFinished] := @.TimeWarehouseWelding
	ExpResults["TotalLeadTime", EngineFinished] :=
		ExpResults.Sum({1, EngineFinished}..{16, EngineFinished})
	RunData[CurrRun, EngineFinished] := ExpResults["TotalLeadTime", EngineFinished]

	-- Code verification
	CheckTotalLeadTime := EventController.SimTime - @.CreationTime
	if abs(ExpResults["TotalLeadTime", EngineFinished] - CheckTotalLeadTime) > 1
		debug
	end

end
```

Code: EndSim from Section 8.6

```
var WorkerSetting: table

-- Stop EventController
EventController.Stop

-- Store Data
AvgExpResults["ExperimentNr", AvgExpResults.YDim + 1] := CurrExp
AvgExpResults["RunNr", AvgExpResults.YDim] := CurrRun
AvgExpResults["EngineCreated", AvgExpResults.YDim] := EngineCreated
AvgExpResults["EngineFinished", AvgExpResults.YDim] := EngineFinished
AvgExpResults["AvgAssemblyTime", AvgExpResults.YDim] :=
	ExpResults.meanValue({"TotalLeadTime", 1}..{"TotalLeadTime", *}) -
	ExpResults.meanValue({"TimeWarehouseWelding", 1}..{"TimeWarehouseWelding",*})
AvgExpResults["CostConfiguration", AvgExpResults.YDim] :=
	(Configurations["JuniorNr", CurrExp] * JuniorCost +
	Configurations["SeniorNr", CurrExp] * SeniorCost) * (RunLength / 86400)

-- Divide by 86400 to multiply the cost with the number of days
if CurrRun < NrReplications  -- Run an additional replication
	CurrRun += 1
	RunCounter += 1
	EventController.Reset
	EventController.Init
	EventController.Start
elseif CurrRun = NrReplications and CurrExp < TotalExp
	-- Run a new configuration and set the new worker settings
	CurrExp += 1
	WorkerSetting.Create
	AssemblyStation.WorkerPool.getCreationTable(WorkerSetting)
	WorkerSetting[2, 1] := Configurations["SeniorNr", CurrExp]
	WorkerSetting[2, 2] := Configurations["JuniorNr", CurrExp]
	AssemblyStation.WorkerPool.setCreationTable(WorkerSetting)
	-- Other settings
	CurrRun := 1
```

```
            RunCounter += 1
            RunData.Delete
            EventController.Reset
            EventController.Init
            EventController.Start
else
            -- Finish simulation
            promptMessage("Simulation finished")
end
```

Code: Start from Section 8.6

```
var WorkerSetting: table

-- Reset Experiment Tables
Configurations.Delete
RunData.Delete
AvgExpResults.Delete

-- Initialise Counters
CurrRun := 1
CurrExp := 1
RunCounter := 1
EventController.End := RunLength + WarmUpLength

-- Set the configurations
SetConfiguration

-- Reset Workers
WorkerSetting.Create
AssemblyStation.WorkerPool.getCreationTable(WorkerSetting)
WorkerSetting[2, 1] := Configurations["SeniorNr", 1]
WorkerSetting[2, 2] := Configurations["JuniorNr", 1]
AssemblyStation.WorkerPool.setCreationTable(WorkerSetting)

-- Start simulation
EventController.Reset
EventController.Init
EventController.Start
```

Code: Revised EndSim from Section 9.1

```
var WorkerSetting: table

-- Stop EventController
EventController.Stop

-- Perform general computations
if CustomRun
            AvgAssemblyTime := ExpResults.meanValue({"TotalLeadTime", 1}..{"TotalLeadTime", *}) -
                        ExpResults.meanValue({"TimeWarehouseWelding", 1}..{"TimeWarehouseWelding", *})
end

CostConfiguration := (NrJuniors * JuniorCost + NrSeniors * SeniorCost) * (RunLength / 86400)
TotalProfit := EngineRevenue * EngineFinished - EngineCost * EngineCreated -
            CostConfiguration - (EngineLead * EngineFinished * AvgAssemblyTime / 86400)
            -- Divide by 86400 to multiply the cost with the number of days

-- Store Data in case of CustomRun
if CustomRun
            AvgExpResults["ExperimentNr", AvgExpResults.YDim + 1] := CurrExp
            AvgExpResults["RunNr", AvgExpResults.YDim] := CurrRun
```

```
        AvgExpResults["EngineCreated", AvgExpResults.YDim] := EngineCreated
        AvgExpResults["EngineFinished", AvgExpResults.YDim] := EngineFinished
        AvgExpResults["AvgAssemblyTime", AvgExpResults.YDim] := AvgAssemblyTime
        AvgExpResults["CostConfiguration", AvgExpResults.YDim] := CostConfiguration
        AvgExpResults["TotalProfit", AvgExpResults.YDim] := TotalProfit
        if CurrRun < NrReplications  -- Run an additional replication
                CurrRun += 1
                RunCounter += 1
                EventController.Reset
                EventController.Init
                EventController.Start
        elseif CurrRun = NrReplications and CurrExp < TotalExp
                -- Run a new configuration and set the new worker settings
                CurrExp += 1
                NrSeniors := Configurations["SeniorNr", CurrExp]
                NrJuniors := Configurations["JuniorNr", CurrExp]
                -- Other settings
                CurrRun := 1
                RunCounter += 1
                RunData.Delete
                EventController.Reset
                EventController.Init
                EventController.Start
        else
                -- Finish simulation
                promptMessage("Simulation finished")
        end
end
```

Code: CheckBatch from Section 9.1

```
/* This Method will check whether all the Batch Items are
   present within the Buffer and if the products can be
   released on the assembly line. */
-----------------------------------------
var y, i, j: integer

str_to_obj(sprint("AssemblyStation.", @.Cont.Name)).Value := str_to_obj(sprint("AssemblyStation.", @.Cont.Name)).Value + 1

if AssemblyStation.XV >= 6 and AssemblyStation.XW >= 5 and AssemblyStation.XX >= 1 and AssemblyStation.XY >= 3 and
AssemblyStation.XZ >= 5

        for i := 1 to GAProductSequence.YDim
                j := 1 -- initialise loop over Buffer content
                while GAProductSequence["ObjectType",i] /= AssemblyStation.BufferAssemblyLine.MU(j).Cont.Name
                        j := j + 1
                end
                AssemblyStation.BufferAssemblyLine.MU(j).Move(AssemblyStation.BatchBuffer)
        next

        AssemblyStation.XV := AssemblyStation.XV - 6
        AssemblyStation.XW := AssemblyStation.XW - 5
        AssemblyStation.XX := AssemblyStation.XX - 1
        AssemblyStation.XY := AssemblyStation.XY - 3
        AssemblyStation.XZ := AssemblyStation.XZ - 5
end
```