

---

# Parameter Catalogs for Simulation

Kai-Holger Brassel, Hamburg, <[mail@khbrassel.de](mailto:mail@khbrassel.de)>

This work by Kai-Holger Brassel, Hamburg, is licensed under [CC BY-NC-ND 4.0](https://creativecommons.org/licenses/by-nc-nd/4.0/)<sup>1</sup>

© ⓘ ⓘ ⓘ

Non-final version: April 20<sup>th</sup>, 2021.

Go to [PDF-Version](#)<sup>2</sup>

## 1. Introduction



This introduction talks about the work of the author and others, but without bibliographic references. Currently, it is just meant as background to better understand the technical documentation in the sections to follow. Maybe it could be developed into a more serious paper later.

Simulation of energy supply and consumption of buildings at the level of districts or even cities not only requires elaborated algorithms but also careful design of model structure and parameters. Structural aspects include building geometry as well as arrangement of buildings, e.g. to take shadowing and heat transfer into account. Assigning usage patterns or energy components like heat pumps, PV, boilers, etc. to specific buildings also count as structural aspects of a simulation model. Moreover, this multitude of model entities has to be defined in more detail by lots of numeric, ordinal or nominal parameters. Our experience with developing simulation systems like INSEL and SimStadt showed that manual parametrization based on informal data collections, typologies, spreadsheet tables, etc. from different sources is tedious and often hard to reproduce. Instead, parametrization of complex models should be supported by software providing formally defined

---

<sup>1</sup> [https://creativecommons.org/licenses/by-nc-nd/4.0](https://creativecommons.org/licenses/by-nc-nd/4.0/)

<sup>2</sup> [ParameterCatalogs.pdf](#)

**parameter catalogs**, that are systematically created and updated by domain experts.

Parameter catalogs and the software to create, maintain and deploy them should be independent of any specific simulation software to enhance software modularity (separation of concerns). Ideally, modelers can enhance their simulation environment by adding suited parameter catalogs as software plug-ins and use them to parametrize model entities easily, e.g. via drag and drop.

Automatic parametrization of components in simulation models requires a formal data model which fits the simulation models in terms of content and structure and can pass information to them. Self-contained parameter catalogs fulfill this requirement by providing an application programmers interface (API) that can be queried for automatic, rule-based parametrization of simulation models.

To get good results fast, close collaboration with domain experts and short development cycles are desirable. We achieved this by exploiting techniques of so called **low code development**. Basically this means that domain experts encode their knowledge into a graphical diagram defining types of components, their relations, and attributes. From this diagram, program code for storing and manipulating data sets in main memory as well as code to write and read that data to and from XML files (or data bases) are automatically generated. A modern graphical user interface to create, read, update, and delete data (CRUD operations) can also be provided with no or very few lines of manually written code.

The overall motivation for the work on parameter catalogs for simulation is to make easier to develop and perform computer simulations in complex and *data rich* domains like building physics, transportation, and all kinds of urban infrastructure.

## **1.1. The Bigger Picture**

A good part of computer science was and is driven by the motivation to make it easier to develop computer programs of all sorts. "Higher" programming languages were invented to make programs human readable and soon special constructs for *functional programming* (computation without side effects) and *structured programming* (computation without go to statements) were introduced to help programmers writing and understanding ever growing programs. Then, between 1962 and 1967, program language Simula was developed especially

to deal with the challenges of simulating systems comprising of many different types of objects. This opened the door to more direct computer representations of real world objects, their attributes, relationships and behavior, ultimately leading to *object-oriented* software development that today is embodied in programming languages like Java, C++, Python, and graphical notations like the Unified Modeling Language (UML).

While these achievements had boosted the productivity of software developers, still the creation of correct, efficient and maintainable programs—including simulations—required a big deal of expert knowledge and experience. To overcome this bottleneck, starting in the 70s, so called 4th generation languages entered the stage. These languages were tailored to specific tasks like statistics ("S" 1976, "R" being its successor), database programming (SQL 1979), or simulation (MATLAB around 1979, Mathematica 1988, Modelica 1999) to name a few. By sacrificing generality, these special languages become more accessible to domain experts, not just trained software developers. To flatten the learning curve even more, formal *graphical* languages for special purposes were invented, e.g. Simulink for block diagram simulation models in 1984, Entity-Relationship-Diagrams for data modeling in 1976, UML for object-oriented systems design in the 1990s, or graphical languages to specify business and also scientific workflows around 2000.

This very short history of technologies for development of software in general, and simulations in particular, shall illuminate the tools at our disposal:

- general purpose programming languages that combine structured, functional and object-oriented approaches to enable the creation of big, modular software systems, often called "programming in the large"
- formal textual domain specific languages (DSLs) dedicated to solve specific tasks with ease
- formal graphical DSLs.

Note that DSLs more tend to describe *what* shall be achieved by a computation instead of describing in detail, *how* to achieve it. Therefore, DSLs usually look more like a model than like an algorithm.

Now back to the task at hand.

Some domains deal with a few types of simple objects to be simulated. Take the building blocks of an electric circuit as an example. The algorithms to simulate these correctly and efficiently may be quite complex—the model elements usually can be described by very few parameters like resistance or capacity. More complex domains like (regenerative) energy systems or building physics deal with more complex objects to be simulated, e.g. PV modules or layered walls of buildings, often coming in different types and configurations, and dozens of possibly interdependent parameters.

## 1.2. Lessons Learned

First a note on terminology: Instead of *parameter catalogs* in SimStadt we used term *library* like in *building physics library*. Obviously this was not a good choice, since *library* is used a lot in IT and programming with all sorts of meaning. Instead we started to talk about *data catalogs*, but in data science this term has specific meaning, namely: catalogs of data and data sources. Since our catalogs, first of all, shall grant structured access to parameters for simulated entities *parameter catalog* sounds more appropriate to me.

The problem of navigating huge parameter spaces and assembling complex simulation models popped up as the author worked on a diagram editor for **INSEL**, a simulation language and runtime environment developed for renewable energy systems simulation. To make existing catalogs on weather data, solar panels and inverter modules accessible to the modeler, special dialogs were added to the INSEL user interface that allowed browsing through the catalogs. Using this browsers, the modeler would choose a weather station, panel or inverter to parameterize a corresponding INSEL function-block. However, there are some severe disadvantages with this approach:

1. Parameter catalogs were stored in a proprietary data format on disk within the INSEL application distribution, meaning they could not used independently from INSEL by other interested parties (systems or users).
2. The catalogs have to be maintained by editing text files manually.
3. While INSEL modeler could browse the catalogs, searching and sorting were not supported.

4. Development of Java Swing UIs for the different kind of catalogs is time consuming as is their maintenance, e.g. if a catalog data format were to change.
5. Putting UIs to handle big amounts of data into a diagram editor is not very user friendly.

From 2013 to 2016, the simulation platform **SimStadt** was developed to make specific modeling and simulation workflows accessible to experts in urban planning and energy systems. Using INSEL and other simulators under the hood, the usage of 3D city data, provided as CityGML files, was a core requirement of this project.

To enable simulation of, say, the heating demand of a district, geometric building data had to be enriched with data on building physics and usage. To do so, existing informations about building physics and usage — often only available as informal typologies or tables — had to be provided to the SimStadt user on an abstract level, e.g. to choose between refurbishment scenarios. At the same time, specific building configurations and parameter sets had to be injected into the simulation models to obtain the desired results.

Again, we implemented parameter catalogs to fulfill these requirements, but compared to the quite simple catalogs used in INSEL, the data for building materials, window, wall and roof types as well as the typologies of buildings, households, usage patterns, and so on were more intricate. They had to be created iteratively in collaboration with domain experts. In this situation, manual coding data formats and access with a general programming language would have led to relatively long iteration cycles and high communication effort between programmer and domain expert. Instead, we decided to use a DSL for data modeling and use code generation whenever possible. Since SimStadt was developed within the Java eco-system we followed this standard approach.<sup>3</sup>

1. Developer and domain expert create a first version of the data model as XML Schema Definition (our DSL).
2. For plausibility checks one would use any standard XML editor to create example data conforming to the XSD.

---

<sup>3</sup> A similar approach is in use to standardize extensions to CityGML via so called application domain extensions (ADE) like the energy ADE for exchanging energy related data.

3. With JAXB (Java Architecture for XML Binding) Java code is generated to read our XML catalogs into Java objects that, in turn, can be accessed by SimStadt workflows to generate and parameterize simulations.
4. If required, developer and domain expert go back to step one to refine data model and catalog data.

After the data model for building physics catalogs had matured, we developed a desktop application for convenient creation and maintenance of building physics data catalogs separate from SimStadt. It was developed in Java with a user interface written in JavaFX and was well received by domain experts.

However, as a different catalog—this time for building usages—had to be created, it was quite difficult to reuse the XML schema and application code from the building physics catalog: The usage catalog data model was "pressed" into a form similar to the building physics catalog data model, and the UI code was "over-engineered" to accommodate both catalog's requirements.

### ***1.3. Low-Code-Development of Parameter Catalogs***

From INSEL and SimStadt we learned, that manual and automatic construction and parameterization of complex simulation models with many types of interrelated objects should be supported by the means of domain specific parameter catalogs.

Close collaboration with domain experts in designing and implementing these catalogs in short development cycles is desirable.

Parameter catalogs and the software for their creation, maintenance and deployment should be independent of any specific simulation software, (a) to be reusable and (b) not to overload simulation applications.

In SimStadt, catalog development was partly facilitated by a textual DSL for data modeling (XML schema language) and automatic generation of Java code from it. On the other hand, user interfaces and generation and parameterization of simulations from templates within SimStadt workflows had still to be coded manually hindering the routinely creation of new catalogs.

Now, in 2020, several developments in different projects provide an opportunity to re-think the topic of parameter catalogs for simulations, namely:

1. Plans for a new Urban Simulation Platform at Concordia University, Montreal.
2. New implementation of INSEL user interface based on the Eclipse application framework and Eclipse-Sirius diagram editors.
3. Enhancement of existing building physics and usage catalogs from SimStadt and their adaptation to new regions.
4. Development of a new comprehensive catalog of electric systems components to be used in SimStadt as well as in Concordia's Urban Simulation Platform.

In what follows, the new technology stack used to implement (4) is documented in detail. It uses four technologies to replace manual coding by code generation from models:

- *Ecore* to model the catalog's data and generate Java classes and persistence layer from it.
- *EMF Forms* for modeling and generating tables, forms and buttons to create, read, update, and delete data (CRUD).
- *E4*, the Eclipse way of modeling the application user interface itself, e.g. the placement and behavior of views, editors, toolbars, menus, and more.
- A template engine called *Handlebars* to generate fully parameterized simulation models from textual templates without programming.

The new technology stack is rooted in the Eclipse application framework and ecosystem.<sup>4</sup> Its main advantage is the possibility to implement CRUD applications like parameter catalogs and their underlying data models with no or very few lines of handwritten code (*low-code-development*).

Plans are to use the same approach also for implementation of (3). Since task (2) and maybe (1) will use Eclipse, too, close integration of parameter catalogs and simulation environments seems feasible. E.g., a user could drag an electric system component from a catalog onto an INSEL block for parametrization.

The Eclipse application framework offers:

---

<sup>4</sup>A comparable, but completely different approach would be to combine several web applications and services via portal software in web browsers.

- OSGI plug-in mechanism and UI framework for integrating applications and services
- *E4* application model to declaratively describe user interface's structure
- General notion of *project* with specific file types, help system, preferences etc.
- IDE support for important general purpose languages like Java, [Python<sup>5</sup>](#), Ruby, C, Fortran, C++
- Support for creating textual and graphical DSLs ([XText<sup>6</sup>](#), [Sirius<sup>7</sup>](#))
- Industry proven DSLs and code generators for data models and form based UIs via the [Eclipse Modeling Framework<sup>8</sup>](#) (EMF) providing:
  - [Ecore<sup>9</sup>](#) for model driven generation of Java classes and persistence layers for XML or data bases
  - [EMF Forms<sup>10</sup>](#) for describing and generating form based UIs
  - Mechanisms to adapt or extend data models and forms to special needs (e.g., we added quantities — that is numbers *with units* — to Ecore and EMF Forms, a feature very important for parameter catalogs)

---

<sup>5</sup> <https://marketplace.eclipse.org/content/pydev-python-ide-eclipse>

<sup>6</sup> <https://www.eclipse.org/Xtext>

<sup>7</sup> <https://www.eclipse.org/sirius>

<sup>8</sup> <https://www.eclipse.org/modeling/emf>

<sup>9</sup> <https://www.eclipse.org/ecoretools>

<sup>10</sup> <https://eclipsesource.com/blogs/tutorials/emf-forms-view-model-elements>



- Rich open source eco-system with lots of plugins and projects important for an urban simulation platform:
  - model server for distributed access and work on Ecore models, including model comparison and migration ([CDO](#)<sup>11</sup>, [EMFCompare](#)<sup>12</sup>)
  - a [Python implementation of Ecore](#)<sup>13</sup>
  - GIS: storage, processing, and visualization of geographical data (list of projects under the umbrella [LocationTech](#)<sup>14</sup>, e.g. user-friendly desktop internet GIS [uDig](#)<sup>15</sup>)
  - workbench for traffic simulation ([SUMO](#)<sup>16</sup>)
  - spatial multi-agent-simulation ([GAMA-Platform](#)<sup>17</sup>)
  - scientific workflows ([Triquetrum](#)<sup>18</sup>)
  - visualizations ([Nebula](#)<sup>19</sup>)
  - machine learning ([deeplearning4j](#)<sup>20</sup>)
  - 45+ projects in the area of [IoT](#)<sup>21</sup>
  - ...

As always, all that glitters is not gold. When we go through the details below, some bugs and inconsistencies, typical for open source projects of this age and size, have to be addressed.

---

<sup>11</sup> <https://projects.eclipse.org/projects/modeling.emf.cdo>

<sup>12</sup> <https://www.eclipse.org/emf/compare>

<sup>13</sup> <https://pyecore.readthedocs.io/en/latest>

<sup>14</sup> <https://projects.eclipse.org/projects/locationtech>

<sup>15</sup> <http://udig.refractor.net>

<sup>16</sup> <https://www.eclipse.org/sumo>

<sup>17</sup> <https://gama-platform.github.io/wiki/Home>

<sup>18</sup> <https://projects.eclipse.org/projects/science.triquetrum>

<sup>19</sup> <https://www.eclipse.org/nebula/widgets/visualization/visualization.php>

<sup>20</sup> <https://deeplearning4j.org>

<sup>21</sup> <https://iot.eclipse.org>

## 2. How to Implement Parameter Catalogs with Eclipse

At the end of this chapter, you should be able to build a running software prototype for creating and maintaining parameter catalogs based on a graphical data model of the domain you are an expert in.

To build data models and parameter catalogs from scratch, we first have to understand some basics about Eclipse, and then install the correct Eclipse package. Thereafter, we can model our data with *Ecore* considering some best practices, followed by the generation of Java classes and user interface (UI). Finally, we will install some plug-ins to "pimp" our Eclipse installation in order to add units and quantities to the mix.

### 2.1. Eclipse Basics

Eclipse<sup>22</sup> was originally developed by IBM and became Open Source in 2001. It is best known for its Integrated Development Environments (*Eclipse IDEs*), not only for Java, but also for C++, Python and many other programming languages. These IDEs are created on top of the Eclipse Rich Client Platform (Eclipse RCP), an application framework and plug-in system based on Java and OSGi. Eclipse RCP is foundation of a plethora of general-purpose applications, too.

First time users of Eclipse better understand the following concepts.

#### Eclipse Packages.

An Eclipse package is an Eclipse distribution dedicated to a specific type of task.<sup>23</sup> A list of packages is available at [eclipse.org](http://eclipse.org)<sup>24</sup>. Beside others it contains *Eclipse IDE for Java Developers*, *Eclipse IDE for Scientific Computing*, and the package we will use: *Eclipse Modeling Tools*. Note that third parties offer many other packages, e.g. *GAMA* for multi-agent-simulation or *Obeo Designer Community* for creating Sirius diagram editors.



Several Eclipse packages can be installed side by side, even different releases of the same package. Multiple Eclipse installations can run at the same time, each on its own *workspace* (see below).

---

<sup>22</sup> [https://en.wikipedia.org/wiki/Eclipse\\_\(software\)](https://en.wikipedia.org/wiki/Eclipse_(software))

<sup>23</sup> The notion of an Eclipse package has nothing to do with Java packages.

<sup>24</sup> <https://www.eclipse.org/downloads/packages/>

## Plug-ins / Features.

An installed Eclipse package consists of a runtime core and a bunch of additional plug-ins. Technically, a plug-in is just a special kind of Java archive (JAR file) that uses and can be used by other plug-ins with regard to OSGi specifications. Groups of plug-ins that belong together are called a *feature*.

Sometimes, a user will add plug-ins or features to an Eclipse installation to add new capabilities. E.g. writing this documentation within my Eclipse IDE is facilitated by the plug-in [Asciidoctor Editor](#)<sup>25</sup>. Plug-ins can easily be installed via main menu command `Help → Eclipse Marketplace...` or `Help → Install New Software...`. Some plug-ins may be self-made like our *City Units* plug-in that enables Ecore to deal with physical quantities.

## Git.

[Git](#)<sup>26</sup> is the industry standard for collaborative work on, and versioning of, source code and other textual data. Collaborative development of parameter catalogs benefits massively from using Git. Git support is built into *Eclipse Modeling Tools*, the Eclipse package we will use. However, if Eclipse needs to connect to a Git server that uses SSH protocol (not HTTPS with credentials), access configuration is more involved and may be dependent on your operating system.

Some users, anyway, prefer to use Git from the command line or with one of the client application listed [here](#)<sup>27</sup>, e.g. [TortoiseGit](#)<sup>28</sup> for Windows.

While it is required to get Git working at some point, we won't refer to it in this document and, for now, do not cover the installation of Git on your machine or configuration of Git in Eclipse.

## Workspaces.

When you start a new Eclipse installation for the first time, you are asked to designate a new directory in your file system to store an *Eclipse workspace*. Eclipse is always running with exact one workspace open. As the name implies, a workspace stores everything needed in a given context of work, namely a set of related projects the user is working on as well as meta-data like preference

---

<sup>25</sup> <https://marketplace.eclipse.org/content/asciidoctor-editor>

<sup>26</sup> <https://git-scm.com>

<sup>27</sup> <https://git-scm.com/downloads/guis>

<sup>28</sup> <https://tortoisegit.org>

settings, the current status of projects, to do lists, and more. In case a user wants to work in different contexts, e.g. on different tasks, command `File` → `Switch workspace` allows to create additional workspaces and to switch between them.



Any plug-in from the original Eclipse package or installed by the user later will be copied into the Eclipse installation directory, **not** in any workspace. Configuration and current state of plug-ins, on the other hand, are stored in workspaces.

## Projects.

An Eclipse project is a technical term for a directory that often contains:

- files of specific types for source code, scripts, XML files or other data
- build settings, configurations
- dependency definitions (remember the dependencies between plug-ins above?)
- other Eclipse projects.

`File` → `New` → `Project...` offers many different types of projects that the user can choose from, e.g. Java projects to create Java programs, Ecore modeling projects, or general projects, that simple hold some arbitrary files.<sup>29</sup>



Files that do not belong to a project are invisible for Eclipse!

The projects belonging to a workspace can either be directly stored within the workspace as sub-directories (the default offered to the user when creating a new project), or linked from it, that is the workspace just holds a link to the project directory that lives somewhere in the file system outside of the workspace. Linking allows to work with the same projects in different workspaces.

While it sometimes makes sense to share or exchange workspaces between users,<sup>30</sup> I do not recommend this for now. Projects, in contrast, are shared

<sup>29</sup> Projects possess one or more *natures* used to define a project's principal type.

<sup>30</sup> Or even work on the same workspace provided in the cloud, see [Eclipse Che](https://www.eclipse.org/che/technology/) [https://www.eclipse.org/che/technology/].

between users most of the time, usually via Git. In general, I would suggest to store Eclipse projects outside workspaces at dedicated locations in the user's file system. That way, we can follow the convention that local Git repositories should all be located under `<userhome>/git`.

## 2.2. Setup Eclipse Modeling Tools

### Install Java.

Eclipse runs on 64-bit versions of Windows, Linux, and macOS and requires an according Java Development Kit (JDK), version 11 or higher, to be installed on your machine. Even if such JDK already exists, please download OpenJDK, version **16** or newer for your operating system from [AdoptOpenJDK](#)<sup>31</sup>.<sup>32</sup> Choose hotSpot as Java Virtual Machine. Installation process is straight forward, but you can also find links to exhaustive instructions for your operating system.

New Java versions appear every six months, so one could tend to stick with older version 11 that comes with long time support (LTE) until next LTE version 17 arrives in autumn 2021. However, actual version 16 conforms to the latest security measures built into macOS Catalina, so it is a must if software we build here shall be deployed to these systems, too.

Note that different versions of Java coexist peacefully.

### Install Eclipse Modeling Tools.

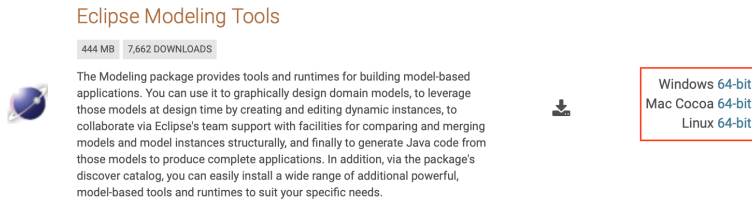
Now its time to download and install the correct Eclipse package *Eclipse Modeling Tools*, version 2021-03 or newer. Please go to [Eclipse download page for packages](#)<sup>33</sup>. On this page you may see "Try the Eclipse Installer" or similar. Do **not** follow this advice, since we want more control over what versions of Java and Eclipse shall be installed. Instead, look for package *Eclipse Modeling Tools* and follow the link for your operating system on the right:

---

<sup>31</sup> <https://adoptopenjdk.net>

<sup>32</sup> AdoptOpenJDK recently joined the Eclipse foundation and soon will change its name to *Adoptium* for legal reasons.

<sup>33</sup> <https://www.eclipse.org/downloads/packages>



**Figure 1. Download links for Eclipse Modeling Tools package**

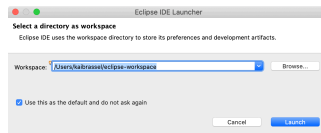
Finally, you can click on `download` and wait for the 400 something MB package to arrive.



Depending on the operating system, several security dialogs have to be acknowledged during installation and first launch of Eclipse.

The downloaded installation file contains the application simply named `Eclipse` ready to be copied into `Applications` on macOS or be installed in `Programs` on Windows. Since later you may add other Eclipse packages — or different versions of the same package — I suggest to rename the application more significantly to `EclipseModeling2103` or similar.

After installation has finished launch Eclipse for the first time and you will see a dialog for choosing a new empty directory as its workspace.



**Figure 2. Initial Dialog to Choose a Workspace Directory**

Again, more workspaces might come into existence later, so replace the proposed generic directory path and name with a more specific one, e.g. `EclipseModelingws`. The Eclipse main window appears with a `Welcome Screen` open. It contains links to exhaustive documentation on concept, features and usage of Eclipse that might be of interest later, especially:

- Overview
  - Workbench basics
    - Concepts: features, resources, perspectives, views, editors

- Opening perspectives and views
- Installing new software manually
- Team support with Git
- Learn how to use the Ecore diagram editor
- Launch the Eclipse Marketplace

For now, you can dismiss the welcome screen. It can be opened anytime by executing `help → welcome`. Now you should see the initial layout of Eclipse with *Model Explorer* and *Outline* on the left and a big empty editing area to the right with a *Properties* view below.

### 2.3. Modeling Parameter Catalogs for Simulation with Ecore

There are two hard problems in computer science: cache invalidation, naming things, and off-by-1 errors.

— Phil Karlton / N.N.

It takes time and effort to come along with good names for model entities, projects, files, and so on. Also, specific naming conventions are in place to enhance readability of models and program code. Since it is not always clear where names provided during modeling are used later, I compiled a list of names important in Ecore projects and added examples and comments to elucidate their meaning and naming conventions.

**Table 1. Naming**

Name	Demo Catalog Example	Real World Example
Namespace URI	<a href="http://example.org/democatalog">http://example.org/democatalog</a>	<a href="http://hft-stuttgart.de/buildingphysics">http://hft-stuttgart.de/buildingphysics</a>
Namespace Prefix	democat	buildphys
Base Package (reverse domain) <sup>a</sup>	org.example	de.hftstuttgart
Main Package	democatalog	buildingphysics
Eclipse Project <sup>b</sup>	org.example.democatalog	de.hftstuttgart.buildingphysics
Class Prefix	Democatalog	Buildingphysics

Name	Demo Catalog Example	Real World Example
XML File Suffix	democatalog	buildingphysics
Classes	e.g. SolarPanel	e.g. WindowType
Attributes	e.g. nominalPower	e.g. id
Associations	e.g. solarPanels	e.g. windowTypes

<sup>a</sup> [https://en.wikipedia.org/wiki/Reverse\\_domain\\_name\\_notation](https://en.wikipedia.org/wiki/Reverse_domain_name_notation)

<sup>b</sup> [https://wiki.eclipse.org/Naming\\_Conventions#Eclipse\\_Workspace\\_Projects](https://wiki.eclipse.org/Naming_Conventions#Eclipse_Workspace_Projects)

Classes are written in [Camel case notation](#)<sup>34</sup> starting with an upper case letter. Associations and attributes are written the same way, but starting with a lower case letter.

All other names should be derived from the globally unique *name space* of the project, in our example: `example.org/democatalog`. It consists of a global unique domain name and a path to the project, unique within that domain.

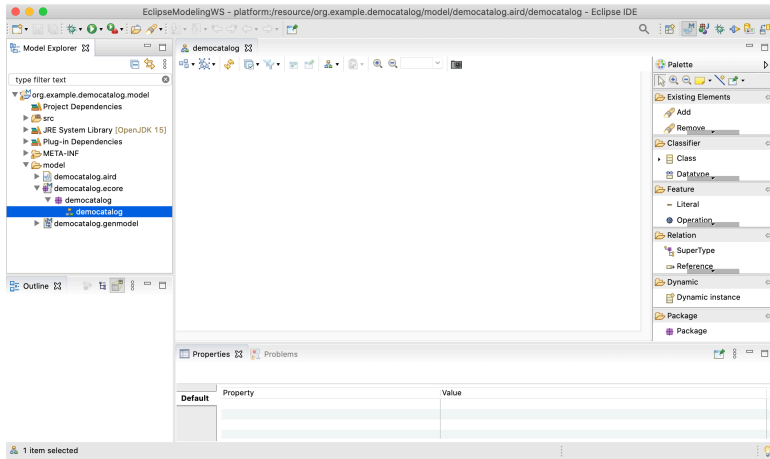
Use the names of example *Demo Catalog* to create your first Ecore modeling project:

1. Execute `File` → `New` → `Ecore Modeling Project` from main menu — not `Modeling Project`!
2. Name the project `org.example.democatalog.model` and uncheck `use default location` so that the new project is **not** stored in workspace but a different directory you create/choose, then click `Next` >
3. Provide `democatalog` as main Java package name, uncheck `use default namespace parameter` and provide <http://example.org/democatalog> as *Ns URI* and `democat` as *Ns prefix*
4. Click `Finish`.

Eclipse should look like below with an new empty graphical Ecore diagram editor opened. The diagram is automatically named `democatalog` after the package name for the Java classes that will be generated from it (provided above). The *Model Explorer* shows the contents of the new Ecore modeling project.

<sup>34</sup> [https://en.wikipedia.org/wiki/Camel\\_case](https://en.wikipedia.org/wiki/Camel_case)





**Figure 3. New Ecore Modeling Project**

To get your feet wet, do this:

1. Drag a *Class* from the palette on the right onto the editor's canvas: it will materialize as a rectangle labeled `NewEClass1`.
2. The class symbol should be selected initially, so you can see its attributes in the *Properties* view.
3. In there replace `NewEClass1` by `EnergyComponentsCatalog` to rename the class.
4. Click anywhere on the canvas and notice that the class symbol is deselected and the toolbar at the top adapts accordingly.
5. In the toolbar change 100% to 75% to scale diagram.
6. Execute `File` → `save` to save model and diagram on disk.
7. Close diagram editor `democatolog` by closing its tab.
8. Reopen saved diagram by double click on entry `democatolog` in *Model Explorer*.

Technically, everything is in place now to begin modeling the data that the projected catalog shall contain. Except ... understanding the basics of object-oriented modeling would be helpful. This is why developers should support domain experts at this stage.

### **Model Data with Class Diagrams.**

Ecore diagrams are simplified UML class diagrams. Here some resources on what this is all about:

- [Toronto Lecture on Object Oriented Modeling](#)<sup>35</sup>
- [UML 2 Class Diagrams: An Agile Introduction](#)<sup>36</sup>
- [UML @ Classroom: Eine Einführung in die objektorientierte Modellierung \(German Book\)](#)<sup>37</sup>



Beginners are strongly encouraged to read the first two resources. The first one contains a gentle introduction, especially suited for domain experts. The second one can also serve as reference.

We will touch central object-oriented concepts *Class*, *Object*, *Attribute*, *Association*, *Composition*, and *Multiplicity* in an example below, but work through above sources to get a deeper understanding and to enhance your modeling skills.

Note that above sources differentiate between *conceptual* and *detailed* models. We go for detailed models, since only these contain enough information to generate code. Having said this, it is usually a good idea to have two or three conceptual iterations at a white board to agree on the broad approach before going too much into detail. But even if one starts with Ecore models right away, these also can be adapted any time to follow a new train of thought.

See here the essential and typical structure of a parameter catalog in a class diagram. Instead of artificial example classes like *Foo* and *Bar* it shows classes from an existing catalog, albeit in very condensed form.

---

<sup>35</sup> <http://www.cs.toronto.edu/~sme/CSC340F/slides/11-objects.pdf>

<sup>36</sup> <http://agilemodeling.com/artifacts/classDiagram.htm>

<sup>37</sup> [https://www.amazon.de/UML-Classroom-Einführung-objektorientierte-Modellierung-ebook/dp/B00AIBE1QA/ref=sr\\_1\\_2?\\_\\_mk\\_de\\_DE=ÅMAŽÖÑ&dchild=1&keywords=UML&qid=1585854599&sr=8-2](https://www.amazon.de/UML-Classroom-Einführung-objektorientierte-Modellierung-ebook/dp/B00AIBE1QA/ref=sr_1_2?__mk_de_DE=ÅMAŽÖÑ&dchild=1&keywords=UML&qid=1585854599&sr=8-2)

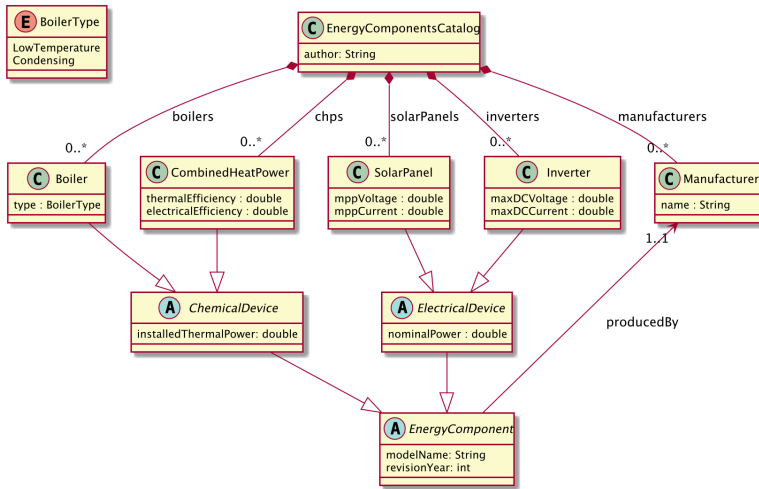


Figure 4. Principle Structure of a Parameter Catalog

The diagram models four types of technical components whose data shall be stored in the catalog, e.g. for parameterization of simulation models later: *Boiler*, *CombinedHeatPower*, *SolarPanel*, and *Inverter*.

The catalog itself is represented by class *EnergyComponentsCatalog*. Unlike dozens, hundreds, or even thousands of objects to be cataloged—Boilers, Inverters etc.—there will be just exactly **one** catalog object in the data representing the catalog itself. Its "singularity" is not visible in the class diagram, but an *Ecore* convention requires that all objects must form a composition hierarchy with only one root object.

**Composition.**

If, in the domain, one object is composed of others, this is expressed by a special kind of association called *composition*. Compositions are depicted as a link with a diamond shape attached to the containing object. In the *Boiler* case said link translates to: The *EnergyComponentsCatalog* contains—or is composed of—zero or more (0..\*) boiler objects stored in a list named `boilers`.



Note that class names—despite the fact that they model a set of similar objects—are always written in *singular*! Names for list-like associations and attributes usually are written in plural form.

**Inheritance.**

Besides composition of **objects**, the model above shows another, completely different, kind of hierarchy: the inheritance hierarchy between **classes**. Whenever classes of objects share the same attributes or associations, we don't like to repeat ourselves by adding that attribute or relation to all classes again and again. Instead, we add a *super class* to define common attributes and associations and connect it to *sub classes* that will automatically *inherit* all the features of their super class.

In our example above, common to all four energy components are attributes `modelName` and `revisionYear`, thus these are modeled by class `EnergyComponent` that is directly or indirectly a super class of `Boiler`, `CombinedHeatPower`, `SolarPanel`, and `Inverter`. Similar, `Boiler` and `CombinedHeatPower` share attribute `installedThermalPower` factored out by class `ChemicalDevice`. `SolarPanel` and `Inverter` share attribute `nominalPower` modeled in abstract class `ElectricalDevice`.

### **Associations.**

You probably noticed a fifth type of objects contained in the catalog, namely `Manufacturer` objects stored in list `manufactureres`. How come? Ok, here is the story:

## Domain Expert Meets Developer

*Exp*: "I'd like to store a component's manufacturer. Shall I add a String attribute `manufacturerName` to all classes like *Boiler*, *Inverter* and so on to store the manufacturer's name?"

*Dev* shudders: "Well, what do you mean by "... and so on"?"

*Exp*: "Basically, I mean all energy components."

*Dev*: "Fine. We already have a class representing all those energy components, brilliantly named *EnergyComponent*. Thus, we can define `manufacturerName` there, following one of Developer's holy principles: "DRY—Don't repeat yourself!" By the way: Is the name all you want to know about manufacturers?"

*Exp*: "Mhm, maybe we need to know if they are still in business ..."

*Dev*: "... or even since when they were out of business, if at all ..."

*Exp*: "... and the country or region they are active."

*Dev*: "Ok, so it's not just the name — we need a class `Manufacturer` to model all these information."

*Exp* sighs.

*Dev*: "Come on, its not that hard to add a class to our data model, isn't it?"

*Exp*: "Ok, but how can we express what components a manufacturer produces?"

*Dev*: "Wasn't it the other way around? I thought, you just wanted to know the manufacturer of a component?"

*Exp*: "What is the difference?"

*Dev*: "In data modeling, it is the difference between a uni-directional and a bi-directional association."

*Exp*: "...?"

*Dev*: "Let's put it that way: The difference between a link with an arrow on one side or on both sides."

*Exp*: "Ok. We don't need a list of components per manufacturer, but simply a reference from the component to its manufacturer."

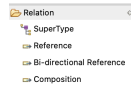
*Dev*: "Fine, then in *Ecore* please create a simple reference from class `EnergyComponent` to class `Manufacturer`, maybe named `producedBy`."

*Exp*: "I will try this and get back to you."

*Dev*: "Fine ... good meeting."

Observe in our data model, reference producedBy points *from* EnergyComponent to Manufacturer making it uni-directional reference. One can simply query the manufacturer of a product, but not the other way around. With a bi-directional reference both queries would be available.

Observe also the annotations 0..\* and 1..1 near class manufacturer. These are *multiplicities* of associations: An EnergyComponentsCatalog contains zero, one, or many objects of class Manufacturer and an EnergyComponent must reference exactly one manufacturer — not less, not more.



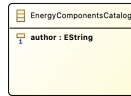
**Figure 5. Ecore Relations**

To recapitulate: Our example parameter catalog already exhibits all four types of relations provided by Ecore. You find these in the Ecore editor's palette shown here. To create a relation between a sub class and a super class use tool superType. Use the other tools to create an association between classes, may it be a simple (uni-directional) reference, a bi-directional reference, or a composition.

### **Attributes and Enumerations.**

Obviously, attributes are central in data modeling. Create one by dragging it from the palette onto our one and only class so far: EnergyComponentsCatalog. The class symbol will turn red to indicate an error. Hover with the mouse pointer over the new attribute and a tooltip with a more or less helpful error message will appear. Current error is caused by that no data type was set for the new attribute. Data types for attributes can be integer or floating point numbers, strings, dates, booleans, and more. To get rid of the error:

1. If not already selected, select new attribute by clicking at it in the editor.
2. In view *Properties* find EType and click button ... to see a quite long list of available data types.
3. Choose EString [java.lang:String] from the list and the error is gone.



**Figure 6. Class with Attribute**

Change the attribute's name to `author` and the class should look like shown here.

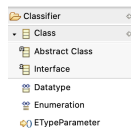
Most data types to choose from begin with letter **E** like in **E**core. These are just Ecore enabled variants of the respective Java types, thus, choose `EInt` for an int, `EFloat` for a 32 bit floating point number, `EDouble` for a 64 bit one, and so on.

Ecore allows to introduce new data types. We employ this feature later to enable data models with physical units and quantities.

There exists one other means to define the values an attribute can take, namely enumerations of distinct literals. Take *Monday*, *Tuesday*, *Wednesday*, ... as a typical example for representing weekdays. In our example data model you'll find one *Enumeration* named `boilerType` with values `LowTemperature` and `Condensing`.

### Homework.

The next section deals with generation of Java code from data models. To have more to play with, please implement our example model in Ecore now.



**Figure 7. Abstract Class**

To do this, there is one more thing to know about classes: the difference between ordinary classes and abstract classes. 'Ordinary class' doesn't sound nice, therefore, classes that are not abstract are called *concrete* classes. Our example diagram depicts abstract classes with letter **A** while concrete classes are labeled with **C**. You add abstract classes to a model with a special palette tool shown here.

The thing is: Objects can be created for concrete classes only!

In our example, it makes no sense to create an object from class *EnergyComponent*, because there is not such a thing like an energy component

*per se*. Therefore, this class is *abstract*. It is true that an inverter *is* an energy component, thus inheriting all its features, but it was *created as Inverter*, not as *EnergyComponent*.

Super classes will be abstract most of the time. So my advice is: Model a super class as abstract class unless you convince yourself that there exist real objects in the domain that belong to the super class but, at the same time, do not belong to any of its sub classes. In the Ecore editor properties view, you can specify if a class is abstract or not, simply by toggling check box `Abstract`.

Two more tips and you are ready to rock and roll! — At least with your homework.



An exhaustive user manual for Ecore diagram editor is available. Execute `Help → welcome` and follow link `Learn how to use the diagram editor`.



If Ecore models get bigger, you may find it more convenient to work with a form based UI instead of, or in addition to, the diagram editor. Open this kind of editor via command `open with → Ecore Editor` from the context menu over entry `democatalog.ecore` in the *Model Explorer* view. Note that Eclipse synchronizes different editors of the same content automatically.

## **2.4. Generation of Java Code from Data Model**

By now, your Ecore model should look like this:



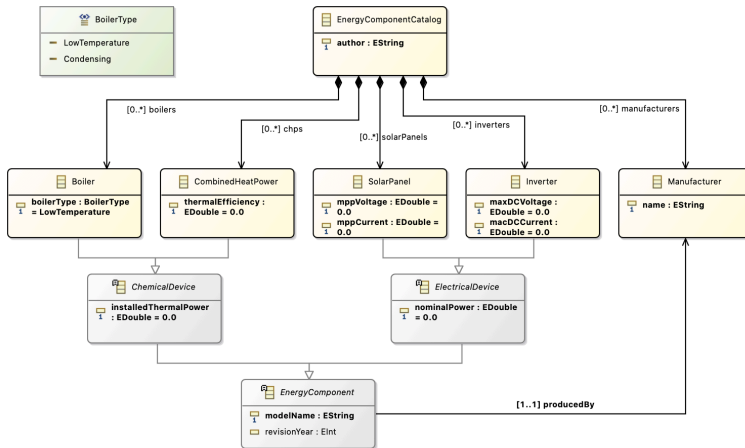


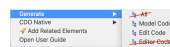
Figure 8. Example Model (Homework)

Let us bring the model to life, that is, generate code from it that creates, reads, updates, and deletes concrete data objects of modeled classes in computers. I would like to tell you that this is done with just one click but, actually, you need two or three:

1. Make sure all files are saved (File → Save All)
2. Execute generate → model code from the context menu of Ecore editor democatalog
3. Execute generate → edit code from the same context menu



Please do **not** execute Generate → All or Generate → Editor Code.



This would create code for a simple user interface, but we use more advanced EMF Forms for that later. If, by mistake, project `org.example.democatalog.editor` was created, just delete it from *Model Explorer* and do not forget to check delete project contents on disk in confirmation dialog.

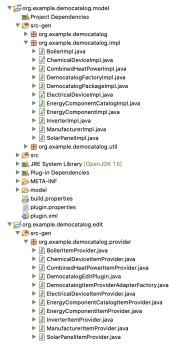


Figure 9. Generated Classes

Generate → Model code creates classes that represent the modeled data in code. These classes are located in three packages under directory src-gen in org.example.democatlog.model.

Generate → Edit code creates a whole new Eclipse project named org.example.democatlog.edit, again with generated classes under directory src-gen.

You may have a look at some Java classes for curiosity by double clicking at them in *Model Explorer*. There is no point in trying to understand the code in detail, but observe token @generated present in the comments of all classes, fields and methods. Classes, fields and methods marked with this token are (re)generated whenever above commands are executed.

Sometimes it maybe required to manually adapt generated code — after all our concern is "low code", not "no code" development. In that case, we will replace @generated by @generated NOT to prevent code regeneration.

After code generation, you may have noticed some warnings showed up in view *Problems*.



Figure 10. Warnings

In general, it is highly recommended to resolve warnings, and errors of course, but we will make an exception from the rule, since the warnings are uncritical and would reappear each time code is regenerated.

## 2.5. Generation and Tweaking of User Interface

In this section you will learn how to generate and tweak a CRUD user interface based on Ecore data model and Java classes created for our demo parameters catalog above. Topics described here are discussed in more detail in tutorial [Getting started with EMF Forms](https://eclipsesource.com/blogs/tutorials/getting-started-with-EMF-Forms/)<sup>38</sup>. To find out what user interface controls and layouts are provided by this framework have a look at [EMF Forms – View Model Elements](https://eclipsesource.com/blogs/tutorials/emf-forms-view-model-elements/)<sup>39</sup>. *EMF Forms* is already part of package *Eclipse Modeling Tools*, so we can create a third Eclipse project/plugin that implements a user interface for editing catalog data without further ado:

1. In the *Model Explorer* execute *EMF Forms* → *Create View Model Project* from context menu over *democatalog.ecore*
2. Leave project name *org.example.democatalog.viewmodel* as is but uncheck *use default location*—as we always do—and browse to the directory containing *org.example.democatalog.model*
3. Click *next >* and select *EnergyComponentsCatalog* as data element we want to create a user interface for
4. Leave *Fill view model* with *default layout* checked and click *Finish*.

According to these inputs a new project is created with file *EnergyComponentsCatalog.view* under directory *viewmodels*. This file opens automatically in a special *View Editor*.

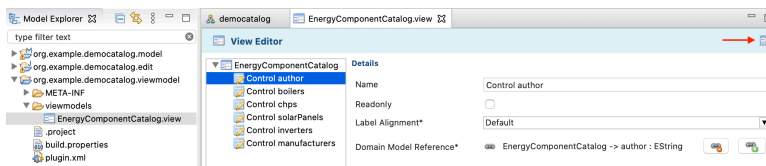


Figure 11. New View Model

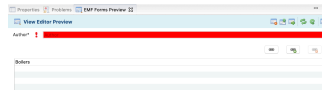
Like the **data** of our catalog are modeled as Ecore file using a dedicated graphical editor, so will our catalog's **user interface** be modeled in *.view* files, again using a special editor.

<sup>38</sup> <https://eclipsesource.com/blogs/tutorials/getting-started-with-EMF-Forms/>

<sup>39</sup> <https://eclipsesource.com/blogs/tutorials/emf-forms-view-model-elements/>

Since we opted for `Fill` view model with default layout the catalog's UI is filled initially with default controls for all data items assigned to Ecore type `EnergyComponentsCatalog` like a string control for author or list controls for boilers, chps, and so on.


See red arrow in the above screen-shot? It points to a button that opens a functional preview of the modeled user interface.



**Figure 12. User Interface Preview**



Double click on tab *EMF Forms Preview* to enlarge view for better handling — double click again to get back.

Enable auto refresh mode  to let each change in the view model instantly be reflected in the preview.

Given your screen is big enough, you may want to dock-out the preview by dragging tab *EMF Forms Preview* out of Eclipse's main window. Seeing editor and preview side by side is a great way to explore the possibilities of view models.

Red input field and exclamation mark in the preview signal missing or inconsistent data. Ecore data model specifies attribute `author` with a lower bound of one, meaning it is a mandatory attribute. As soon as an author's name is provided, the error indication disappears. This is what *functional* preview means. You can even create new boilers or other objects in lists provided, with all forms created "automagically" with respect to our underlying Ecore data model.

Of course, such automatic approach has its limits. In our case, a long list of lists is not very user-friendly, because one has to scroll up and down to find a specific list. Also, no specific object data are shown in the list and data can only be edited in a pop-up form (no inline editing).

How should a better UI look and feel like?

If there are many lists (types) of entities—the normal case for parameter catalogs — users should select what list they want to work with by selecting it from

a list or tree view that is always visible, the *master view*. Once a type is selected in the master view, a table with all objects of this list/type shall appear sidelong in a *detail view*, ready for editing.

In *EMF Forms* master-detail-views can be modeled either with *Categorization* or *Tree Master Detail* UI components. The latter not only allows to edit information displayed in the detail view, but also the tree of elements in the master view. Opposed to that, a *Categorization* presents a fixed hierarchy of elements to choose from. This is exactly what we need as there are only a fixed number of types of objects to be edited in a parameters catalog.

## Adding Tables to the UI

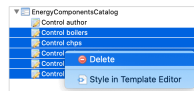


Figure 13. Delete default list controls

But first, we replace the default controls for lists of boilers, chps, and so on by tables. As shown here, select all list controls in the view model and execute `delete` from context menu. Refresh *View Editor Preview* to verify that only field `Author*` is left.

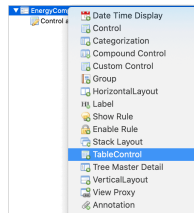




Figure 14. Create Table Control

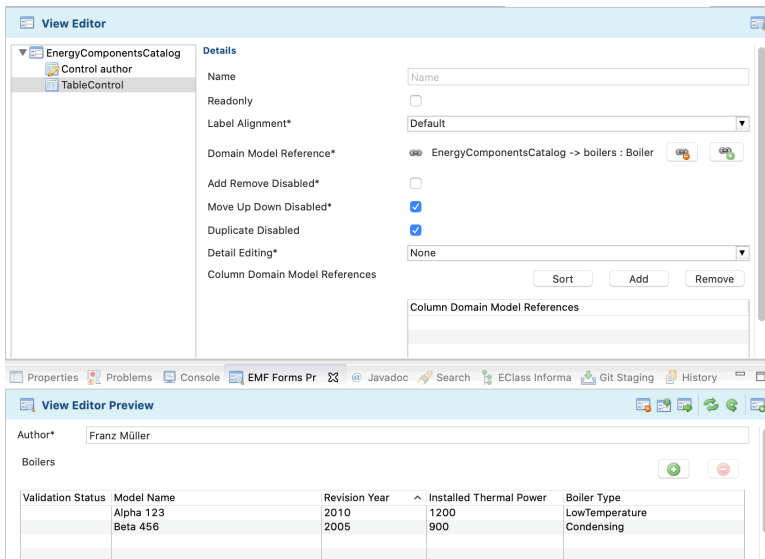
Next, create a table that shall display all boilers in a catalog: Select `EnergyComponentsCatalog`, activate context menu and choose `TableControl` from the list of available UI elements. (`EnergyComponentsCatalog` represents the root view of the UI and, as such, accepts quite a lot of different UI components as child components.)

Entry `TableControl` was inserted into the list of interface elements below `control author`. Checking updated preview reveals no table but a message basically saying that a reference to the domain model is missing, in other words:

*EMF Forms* does not know yet what data to present in table. Click on entry `TableControl` to see its details. A red exclamation mark indicates the missing domain model reference\*. Click on  and be ready to chase a sequence of dialogs:

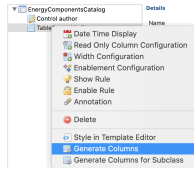
1. Click on another  in dialog `configure TableDomainModelReference`
2. In wizard `New Reference Element` select `model` → `FeaturePathDomainModelReference` and click `next >`
3. Click `Link domain model EFeature` and in appearing pop-up list choose reference to list of objects you want to edit in the table, e.g. `boilers`; confirm with `ok` safely ignoring warning about missing `PropertyDescriptor`.
4. Finish wizard `New Reference Element`
5. Finish dialog `configure TableDomainModelReference`.

This was some work, but as reward we get a fully specified table control in *View Editor* that "translates" into a preview where we can create, read, update, and delete boilers.



**Figure 15. Table for Boilers**

Moreover, clicking at a table header sorts all objects in it (rows) according to the values in this column. Column widths can adapted, too.



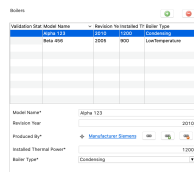
**Figure 16. Modify Table Control**

Table UIs can be tweaked in many ways, e.g. selection and sequence of columns can be declared via `list column domain model` references. To fill this list with defaults, execute `generate columns` from table control's context menu. Reorder them as you like or delete columns that are not important to the user.

Notice here an important overall feature of *EMF Forms*: If something is left unspecified, be it the view model for an Ecore object type or the specification of table columns, *EMF Forms* will always find a default solution! Applied to columns specification this means we get default columns automatically back in the moment the last column is removed from `list column domain model` references.

If explicit column specifications are present further configurations can be added to a table control from its context menu, e.g. initial column widths or read-only status of columns. See [here](#)<sup>40</sup> for details.

By default only attributes are displayed and directly editable in tables while references to other objects — in our case the reference to a manufacturer — are not.



**Figure 17. Default Panel for Boilers**

To get the default (*sic!*) editing panel for an selected table row, in *View Editor* just set `detail editing*` from `None` to `withPanel`, **press Tab**, and save. For boilers, *EMF Forms* will create the editing panel shown here. Regardless whether users edit data in the panel or directly in the table — both will stay in sync any time.

<sup>40</sup> <https://eclipsesource.com/de/blogs/2018/01/31/emf-forms-1-15-0-feature-enhanced-table-renderer/>



*View editor* exhibits an irritating behavior: With preview auto-refresh turned on, any change in the details view is reflected instantly in the preview, even without saving the form or leaving the edited field.

On the other hand, **saving** an updated view editor only takes into account edited fields after they have lost focus, e.g. by pressing *Tab* key or clicking with the mouse into another field. So, saving a form before the focus has shifted from the last edited field won't honor this edit, that is you won't necessarily get what you see.

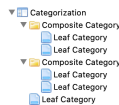
One last thing: Enter `boilers` as name for the table control so we can distinguish it from the other four table controls to come.

Yes! ... Please repeat above procedure to create table controls for `chps`, `solar panels`, `inverters` and `manufacturers`, too. I did this in about 3 minutes. ;-)

## ***Master-Detail View with Categories***

In last section we improved our catalog's UI by replacing simple object lists by tables that can be sorted, customized and edited inline as well as in an associated panel. Alas, instead a list of lists we have got an even bigger list of tables.

High time to introduce a master-detail view that presents categories of object types in a master view and, after one is selected, the according object table in the detail view.



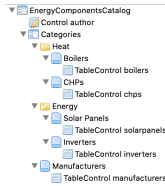
**Figure 18. Category Tree**

Add a *Categorization* view to the list of UI elements in *View Editor* by selecting `EnergyComponentsCatalog` and choose `categorization` from its context menu.

Now add two `Composite Category` elements and one `Leaf Category` to `categorization` from according context menu. This gives us three top level entries in the hierarchy.



In the same way add two Leaf Category elements to each composite Category resulting in the hierarchy depicted here.




**Figure 19. Completed View Model**

This screen shot shows the view model of our UI when finished. To get there:

1. Select UI element categorization and rename it to categories
2. Rename composite and leaf categories as depicted here
3. Drag all table controls one by one into the suited leaf category
4. Confirm master-detail view works as expected in the preview.



The UI hierarchy to access tables for entity types is independent, and usually will differ, from aggregation and inheritance hierarchies present in Ecore data model (compare fig. [Example Model](#)).

Note that *EMF Forms Preview* provides these buttons  to clear, load and store edited data. In fact, this feature gives us a fully functional prototype. At least during refinement of model and UI, data sets can be created, edited, and tested for usability without the need to built a full blown, deployable application — see parts *Accessing and Using Parameter Catalogs* and *Build (Parameter Catalog) Applications with Eclipse Tycho* below.

Be aware that in some cases the view model must adapt to changes in data model, e.g. a new leaf category and table component must be created for a new catalog object type. Other changes are automatically reflected in the generated UI, at least for default forms and default table columns. To our convenience, view model specifications incompatible with data model are indicated by error badges in the *View Editor*.

Changes in data model also can make existing XML data incompatible. There are tools for data migration, but for now, recreation of test data or manual editing of XML file is the way to go.

## 2.6. Add Units to the Mix

As mentioned earlier, parameter catalogs should be able to represent quantities, not just bare numbers. See [Unit of measurement libraries, their popularity and suitability](#)<sup>41</sup> for a systematic account of open source solutions in the this area.

Java provides an extensive framework to deal with quantities and their units defined in [Java Specification Request \(JSR\) 385](#)<sup>42</sup>. The reference implementation for this framework is [Indriya](#)<sup>43</sup>. Demos of its usage can be found at <https://unitsofmeasurement.github.io/uom-demos/>.

To make Indriya available Ecore data models and EMF Forms, the author has created two plug-ins that can easily be added to Eclipse. To do so, open dialog Help → Install New Software... and enter site [https://transfer.hft-stuttgart.de/pages/neqmodplus/indriya-p2/release\\_target\\_211/](https://transfer.hft-stuttgart.de/pages/neqmodplus/indriya-p2/release_target_211/) like depicted below.



Figure 20. Install Plug-in from Specific Update Site

Select Indriya plug-in, press next > and acknowledge all following dialogs, including security warnings.

Do the same for the City Units plug-in available at site [https://transfer.hft-stuttgart.de/pages/neqmodplus/de.hft-stuttgart.cityunits/release\\_target\\_101/](https://transfer.hft-stuttgart.de/pages/neqmodplus/de.hft-stuttgart.cityunits/release_target_101/) Finally, restart Eclipse to complete plug-in installation.

While the first plug-in installs Indriya, the second plug-in adds some specific units for urban simulation, EMF Forms editor fields for quantities, and Ecore types used for modeling quantities as attributes of classes.

Make the new Ecore types `quantityLong` and `quantitybouble` available like so:

<sup>41</sup> <https://onlinelibrary.wiley.com/doi/full/10.1002/spe.2926>

<sup>42</sup> [https://docs.google.com/document/d/12KhosAFriGCczBs6gwtJJDfg\\_QIANT92\\_lhxUWO2gCY/edit#heading=h.6698n7erex5o](https://docs.google.com/document/d/12KhosAFriGCczBs6gwtJJDfg_QIANT92_lhxUWO2gCY/edit#heading=h.6698n7erex5o)

<sup>43</sup> <https://unitsofmeasurement.github.io/indriya/>

1. Open sample Ecore Model Editor from the context menu over your Ecore model
2. From the context menu in this editor, execute Load Resource... and then Browse Target Platform Packages...
3. Select package <https://www.hftstuttgart.de/quantities> and confirm the addition of platform:/resource/de.hftstuttgart.cityunits.model/model/Quantities.ecore.

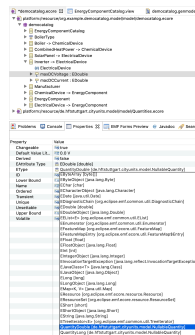


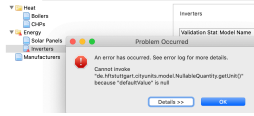
Figure 21. Set QuantityDouble Type

From now on, the new attribute types are available to model quantities with integer or floating point values as can be seen on the screenshot to the right.

Note here, that I also changed the default value from 0.0 to 0.0 v to indicate that maxDCVoltage of inverters is given in **Volt**.

The symbols for defining units follow SI and other standards, including decimal prefixes like m for Milli or G for Giga as well as derived units, that is: mv, gv or  $\text{kW}\cdot\text{h}/\text{m}^3$  are all valid unit definitions. This is all documented well in the resources mentioned at the top of this section, but for convenience, a table with valid units, including some specific units for urban simulation, is compiled in [UnitsExamples.md](#).

If a unit symbol cannot be interpreted, this error is not already detected while generating and compiling code from model, but not before run time when the application tries to create the default value. In that case, you will see an error message like this:



**Figure 22. Error Message for Wrong Unit Definition**

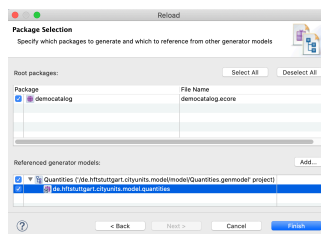


Be told that each attribute of type `quantityLong` or `quantityDouble` has to have a unit defined in its `default value Literal`, even optional attributes that do not require a numerical value to be set. For these, too, the Ecore model must specify the unit to use.

In other words: If you do not want to preset an attribute with a numerical default, you can omit the numerical part, but still must provide the unit symbol as `default value Literal`, e.g. `v` will work as well as `1.0 v` but leaving the numeric value initially undefined.

One last technicality. **Before** code from an Ecore model with attributes of type `quantityLong` or `quantityDouble` can be generated correctly, we must tell Eclipse to reuse the corresponding generator model from the City Units plug-in:

1. In package or project explorer find your generator model, e.g. `democatatalog.genmodel` and execute `reload...` from its context menu
2. Choose `Ecore model`, press `next >` and `Load` the model (again)
3. `next >` will open the page below. In section *Referenced Generator Models* select the `Quantities` generator model as depicted and click on `Finish`.



**Figure 23. Add Reference to Imported Generator Model**

If no generator model is available for selection, press button `Add...` to add it first (this only works if `quantityLong` or `quantityDouble` were used at least once in the Ecore model).

## **2.7. Summary**

Congratulations on making it this far. What have we achieved?

We get to know the *Eclipse Modeling Tools* IDE and created a graphical Ecore data model with one catalog class and five classes/types of domain objects therein. Classes have been defined by name, attributes, and relationships between them, often with cardinalities. Whenever classes shared some attributes or relationships we factored these out into super classes. An enumeration introduced a new attribute type as a set of named values.

From this data model, we issued commands to create Java code for representing the data in memory as well as to store and retrieve them on and from disk. Methods to create, read, update and delete data objects (CRUD) were generated, too.

We reflected on a good user interface for this data and used *EMF Forms* to model such an interface resulting in a full functional prototype.

Lastly, we enhanced Eclipse, Ecore and *EMF Forms* with two plug-ins for modeling, editing and persisting physical quantities as numerical values with defined units.

### **3. TBD: Accessing and Using Parameter Catalogs**

#### ***3.1. Accessing XML-Catalogs from Java***

#### ***3.2. Create Inset Models with Handlebars Templates***

#### ***3.3. Accessing XML-Catalogs from Python***

## 4. TBD: Distribution of Parameter Catalogs

Three plugins so for for the content and UI.

Missing: Deployable application and inclusion to third party libraries.

Building an application "around" the three plugins for Ecore data model and UI specification model.

### 4.1. Create an Eclipse Application

### 4.2. Use Maven and Tycho as Build System

#### Install Maven Support.

We are going to create a complete Eclipse desktop application from generated code. We also want to deploy this application for Linux, macOS and Windows operating systems. Eclipse offers several approaches for compiling and deploying such an application, traditionally with *Ant* scripts.

Creation and maintenance of these scripts turned out to be tedious and error prone. For quite some years now, the proposed—and mostly supported—method for building Eclipse applications is to use *Maven* build system, more specifically, a couple of Maven plug-ins, subsumed under the name *Tycho*.

Many Eclipse platforms have Maven support [M2Eclipse](https://www.eclipse.org/m2e/)<sup>44</sup> already built in, not so our *Eclipse Modeling Tools*. But don't worry: Installation of required Eclipse feature is easy and straight forward. And, by the way, you will acquire the indispensable skill of how to install new plug-ins/features to Eclipse.

First, tell your Eclipse installation where to look for the new software. Execute `Help → Install new Software...` to invoke dialog *Available Software* and press `Add...`. Sub-dialog *Add Repository* pops up.

---

<sup>44</sup> <https://www.eclipse.org/m2e/>

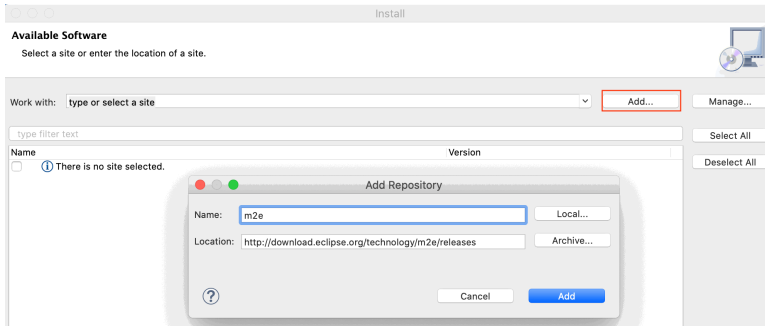


Figure 24. Add update site m2e

In there provide m2e as name and

`http://download.eclipse.org/technology/m2e/releases`

as location. After confirmation with Add, choose the new site to work with: Eclipse now looks up the site for available software.

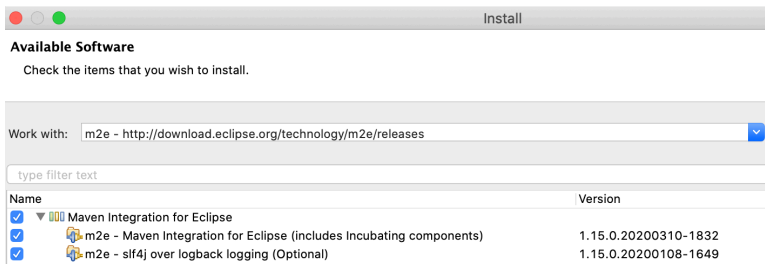


Figure 25. Choose features to install

Provided Group items by category is checked, above features are displayed. Check all features and confirm all following questions about licenses and security concerns. After download is complete—it can take a few minutes—restart Eclipse. Verify that Maven version 3.6.3 or above is now installed in window → Preferences... (or Eclipse → Preferences... on macOS) under Maven → Installations.

Figure 26. Check Maven installation

### Add third party Java libraries.

"Correct" way to add third party Java libraries as plugins



Example Indriya

### ***4.3. Deploy to P2 Repository***

### ***4.4. Versioning and Collaboration***

---