Parameter Catalogs for Simulation

Kai-Holger Brassel, Hamburg, <mail@khbrassel.de>

Non-final version: September 7st, 2021.

Go to PDF-Version as of September 7th, 2021[2]

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

[1] https://creativecommons.org/licenses/by-nc-nd/4.0
[2] 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, Modellica 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:[3]

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.

[3] 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 be 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.
- *Eclipse Sirius* for modeling and generating tables, forms and buttons to **c**reate, **r**ead, **u**pdate, and **d**elete 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.[4] Its main advantage is the possibility to implement CRUD applications like parameter catalogs and their underlying data models with no or very view 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:

[4] 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[5], Ruby, C, Fortran, C++
- Industry proven DSLs and code generators for data models and UIs based on the *Eclipse Modeling Framework*[6] (EMF):
  - *Ecore*[7] for model driven generation of Java classes and persistence layers for XML or data bases
  - *Eclipse Sirius*[8] for describing and generating graphical and form based UIs
  - XText[9]: Support for creating textual DSLs.
  - Mechanisms to adapt or extend data models and forms to specific needs (e.g., we added quantities — that is numbers *with units* — to Ecore, a feature very important for parameter catalogs)

[5] https://marketplace.eclipse.org/content/pydev-python-ide-eclipse
[6] https://www.eclipse.org/modeling/emf
[7] https://www.eclipse.org/ecoretools
[8] https://www.eclipse.org/sirius/
[9] https://www.eclipse.org/Xtext

- 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[10], EMFCompare[11])
  - a Python implementation of Ecore[12]
  - GIS: storage, processing, and visualization of geographical data (list of projects under the umbrella LocationTech[13], e.g. user-friendly desktop internet GIS uDig[14])
  - traffic simulation (SUMO[15])
  - spatial multi-agent-simulation (GAMA-Platform[16])
  - scientific workflows (Triquetrum[17])
  - visualizations (Nebula[18])
  - machine learning (deeplearning4j[19])
  - 45+ projects in the area of IoT[20]
  - …

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.

10 https://projects.eclipse.org/projects/modeling.emf.cdo
11 https://www.eclipse.org/emf/compare
12 https://pyecore.readthedocs.io/en/latest
13 https://projects.eclipse.org/projects/locationtech
14 http://udig.refractions.net
15 https://www.eclipse.org/sumo
16 https://gama-platform.github.io/wiki/Home
17 https://projects.eclipse.org/projects/science.triquetrum
18 https://www.eclipse.org/nebula/widgets/visualization/visualization.php
19 https://deeplearning4j.org
20 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[21] 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.[22] A list of packages is available at eclipse.org[23]. Beside others it contains *Eclipse IDE for Java Developers*, *Eclipse IDE for Scientific Computing*, and *Eclipse Modeling Tools*. Note that third parties offer many other packages, e.g. *GAMA* for multi-agent-simulation or *Obeo Designer Community* for creating diagram and form editors. This is the package we will use later.

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).

[21] https://en.wikipedia.org/wiki/Eclipse_(software)
[22] The notion of an Eclipse package has nothing to do with Java packages.
[23] 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[24]. 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[25] 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[26], e.g. TortoiseGit[27] 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

[24] https://marketplace.eclipse.org/content/asciidoctor-editor
[25] https://git-scm.com
[26] https://git-scm.com/downloads/guis
[27] 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.[28]

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,[29], I do not recommend this for now. Projects, in contrast, are shared

[28] Projects possess one or more *natures* used to define a project's principal type.
[29] Or even work on the same workspace provided in the cloud, see Eclipse Che [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 Obeo Designer

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 is already installed on your machine, please download the OpenJDK version **16** or newer for your operating system from Adoptium[30]. 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 Obeo Designer, Community Edition.
Our graphical and form based modeling tools, e.g. Insel 9.0 and Parameter Catalogs, run on top of Eclipse Sirius[31]. Technically, the Eclipse Sirius project provides a set of open source features and plugins that can be added to any Eclipse package to transform it into a very flexible modeling workbench. Instead of adding these software components manually, we start with a pre-configured Eclipse package named *Obeo Designer*. Please download and install the latest version (11.5 at the time of writing) available at Download Obeo Designer Community[32].

[30] https://adoptium.net
[31] https://www.obeodesigner.com/en/product/sirius
[32] https://www.obeodesigner.com/en/download

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

After the 400 something MB package has arrived, unzip the downloaded file and move the resulting application named `ObeoDesigner-Community` into `Applications` on macOS, `Programs` on Windows, or similar on Linux.

Special installation note for macOS: As Obeo Designer currently is not code-signed, macOS consideres it as damaged. To work around this security feature, remove the quarantine status of the program like so:

1. Open a terminal in the folder containing the .app file
2. Execute: xattr -d com.apple.quarantine ObeoDesigner-Community.app
3. Double click the app to start.

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

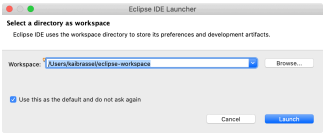

Figure 1. Initial Dialog to Choose a Workspace Directory

More workspaces might come into existence later, so replace the proposed generic directory path and name with a more specific one, e.g.`ObeoDesignerWS`. The main window appears with a Welcome Screen open. Especially under `Documentation` you will find exhaustive documentation on Eclipse that might be of interest later, e.g.:

- Workbench User Guide
  - Concepts: perspectives, projects, views, editors, features, resources, …
  - Tasks: Working with perspectives, views and editors, installing new software. …
- EGit Documentation

    - Git for Eclipse Users
    - EGit User Guide
- Ecore Tools User Manual: Learn how to use the Ecore diagram editor.

For now, you can dismiss the welcome screen. It can be opened anytime by executing `Help → Welcome`.

Now you should see the initial window layout with *Model Explorer* and *Outline* on the left and a big empty editing area to the right with a *Properties* view below.

Add Plug-ins to deal with Quantities and Units.
Parameter catalogs should be able to represent quantities, not just bare numbers. See Unit of measurement libraries, their popularity and suitability[33] 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[34]. The reference implementation for this framework is Indriya[35]. Demos of its usage can be found at https://unitsofmeasurement.github.io/uom-demos/.

To make Indriya available for use in Ecore data models, 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/` like depicted below.

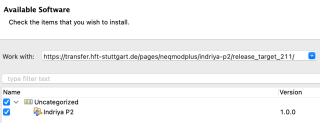

Figure 2. Install Plug-in from Specific Update Site

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

[33] https://onlinelibrary.wiley.com/doi/full/10.1002/spe.2926
[34] https://docs.google.com/document/d/12KhosAFriGCczBs6gwtJJDfg_QIANT92_IhxUWO2gCY/edit#heading=h.6698n7erex5o
[35] https://unitsofmeasurement.github.io/indriya/

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_110/ 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 and Ecore types used for modeling quantities as attributes of classes.

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. Exercise: Modeling a Parameter Catalog with Ecore

Before we start working on real catalog projects hosted in a Git repository in the next section, let us first create a demo project for playing around and learning basic modeling skills.

> 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 Expample
Namespace URI	http://example.org/democatalog	http://hft-stuttgart.de/buildingphysics
Namespace Prefix	democat	buildphys
Base Package (reverse domain)[a]	org.example	de.hftstuttgart
Main Package	democatalog	buildingphysics

Name	Demo Catalog Example	Real World Expample
Eclipse Project[b]	org.example.democatalog.model	de.uni-stuttgart.buildingphysics
Class Prefix	Democatalog	Buildingphysics
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

[a] https://en.wikipedia.org/wiki/Reverse_domain_name_notation

[b] https://wiki.eclipse.org/Naming_Conventions#Eclipse_Workspace_Projects

Classes are written in Camel case notation[36] 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

[36] https://en.wikipedia.org/wiki/Camel_case

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.

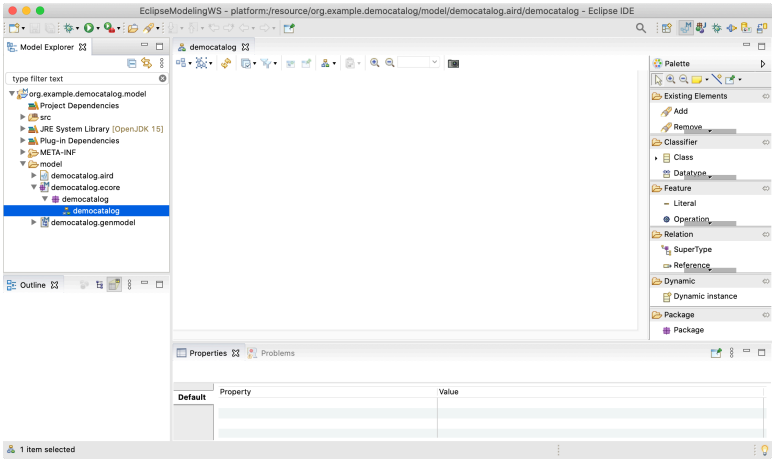

Figure 3. New Ecore Modeling Project

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 `democatalog` by closing its tab.
8. Reopen saved diagram by double click on entry `democatalog` 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[37]
- UML 2 Class Diagrams: An Agile Introduction[38]
- UML @ Classroom: Eine Einführung in die objektorientierte Modellierung (German Book)[39]

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.

[37] http://www.cs.toronto.edu/~sme/CSC340F/slides/11-objects.pdf
[38] http://agilemodeling.com/artifacts/classDiagram.htm
[39] https://www.amazon.de/UML-Classroom-Einführung-objektorientierte-Modellierung-ebook/dp/B00AIBE1QA/ref=sr_1_2?__mk_de_DE=ÅMÅŽÕÑ&dchild=1&keywords=UML&qid=1585854599&sr=8-2

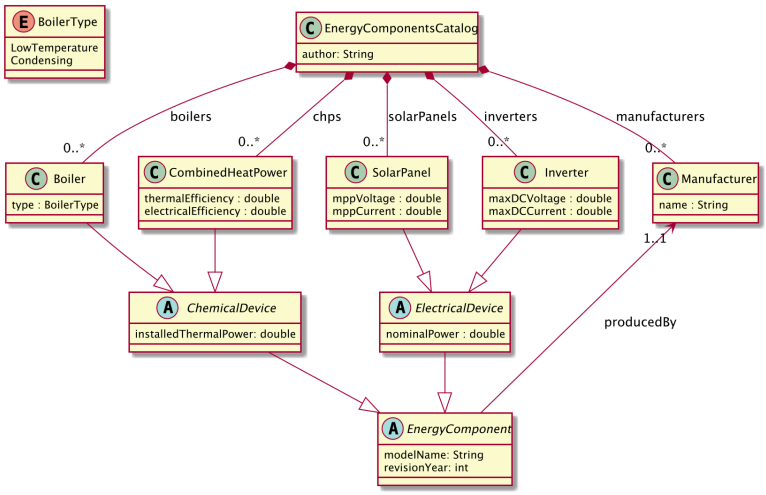

Figure 4. Principle Structure of a Parameter Catalog

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 `manfacturerName` 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.

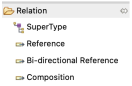

Figure 5. Ecore Relations

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.

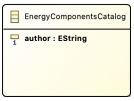

Figure 6. Class with Attribute

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.

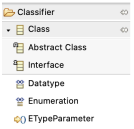

Figure 7. Abstract Class

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 at `Help → Welcome → Documentation → EcoreTools User Manual`.

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.

That's it for the data modeling part. By now, your Ecore model should look like this:

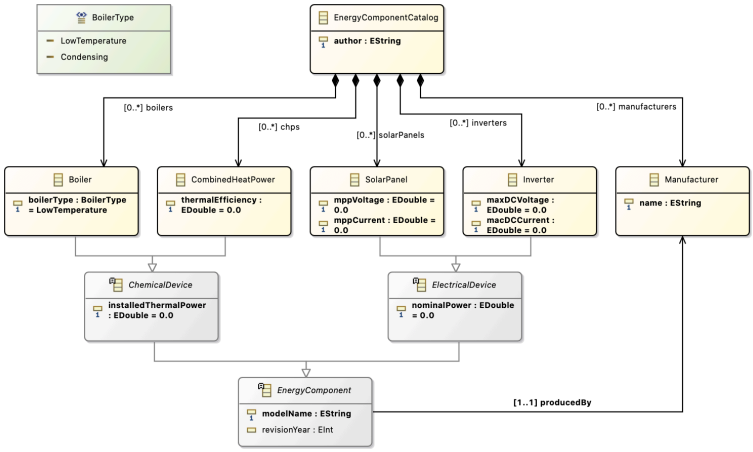

Figure 8. Example Model (Homework)

2.4. Making an Application to Create and Edit Data

In this section you will get a glimpse on how to create an application to create and edit data conforming to the Ecore data model of our demo parameters catalog.

Topics described here (and much more) are discussed in this Sirius Starter Tutorial[40].

If you are less interested in the details of UI creation, but more in working on already existing parameter catalog software and data, you may skip this section for now and proceed with ???.

Generation of Java Code from Data Model.
Let us bring the Ecore data model to life, that is, generate code from it that allows to create, read, update, and delete (CRUD) concrete data objects of modeled classes in computers:

1. Make sure all files are saved (`File → Save All`)
2. Execute `Generate → All` from the context menu of Ecore editor `democatalog`

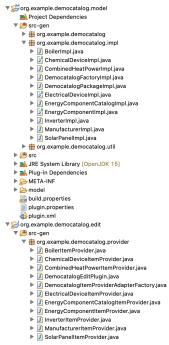

Figure 9. Generated Classes

`Generate → All` creates classes that represent the modeled data in code at first. These classes are located in three packages under directory `src-gen` in `org.example.democatalog.model`. Then, the command generates `Edit Code` and `Editor Code` within two new Eclipse projects named `org.example.democatalog.edit` and `org.example.democatalog.editor`, again with generated classes in `src-gen`.

[40] https://wiki.eclipse.org/Sirius/Tutorials/StarterTutorial/

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 is 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 of the respective item.

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.

Create a Prototype Application for Data Editing and UI Design.
Firstly, launch a new instance of the running *Obeo Designer* application:

1. Execute `Run → Run Configurations…` from the main menu and double click on *Eclipse Application* to get a *New_configuration*. You may want to rename *New_configuration* to *DemoCatalog* or the like.
2. Press `Run` to start the new Eclipse application that is basically a copy of your running *Obeo Designer* application but with a different workspace.
3. In the new application window close the welcome screen and open the Sirius perspective using the suited button in the top right corner of the main window. This perspective provides specific Sirius menus and new project types.

Secondly, create a project that will contain catalog data (remember Eclipse can only handle files that are part of a project):

1. From main menu execute `File → New → Modeling Project` — not `Ecore Modeling Project`!

2. Name the project `org.example.democatalog.data` and uncheck *Use default location* so that the new project — again — is **not** stored in workspace but a different directory you create/choose, usually a directory named like the project and sitting side by side to the model, edit and editor project directories created above.
3. Click `Finish`.

Thirdly, create a first XML file for catalog data:

1. From main menu execute `File → New → Other…` and type `demo` into search field *Wizards:*
2. Select `Example EMF Model Creation Wizards → DemoCatalog Model` and click `Next >`
3. Select `org.example.democatalog.data` as parent directory and name the data file `First.democatalog`
4. Click `Next >` and choose `Energy Component Catalog` as the root data object that will be created initially
5. Click `Finish`.

A new entry named *First.democatalog* should appear in the *Model Explorer*. Double-click it and a **generic** editor will open. In principle, one could use this editor to add new data to the catalog via `New Child >` in the context menu over entry `Energy Component Catalog`. Data of a selected entry can be edited in view *Properties* that is generic, too.

Please add two or three boilers this way to have some data to play with below.

When done, you may save *First.democatalog* so that after closing the application data will reappear if it is opened again as described above.

The generic editors don't get as far. Usually, one would like to have tables, custom property sheets, input validation, and more. Well, Sirius is all about creating nice graphical and form-based editors for data models specified in Ecore. To do this we need one more Eclipse project.

UI Design Project.
Like the **data** of our catalog is modeled as an Ecore file using a dedicated graphical editor, so will the **user interface** (tables, trees, diagrams, property

views) be modeled in a Sirius `.odesign` file that lives in a special Eclipse project that we create while still in the (second) Eclipse application that hosts the data:

1. Execute `File → New → ViewPoint Specification Project`
2. Name the project `org.example.democatalog.design`, uncheck *Use default location* as always and create/choose a directory with the same name as the project besides to the model, edit, editor, and data project directories
3. Click `Finish`.

A special editor for file `democatalog.odesign` appears automatically. In it select `MyViewpoint` and rename it in *Properties* to `Catalog`. A viewpoint provides a set of representations (tables, trees, diagrams, property views) that end-users can instantiate.

Adding a Table to the UI.
In what follows, we work with the *democatalog.odesign* editor. Say, we want to add a table for boilers to the UI:

1. From context menu over viewpoint *Catalogs* execute `New Representation → Edition Table Description` to create a new description that is automatically selected and shown in view *Properties*
2. To connect the table with its data model, choose tab *Metamodels* in *Properties*, click on *Add from registry* and select `http://example.org/democatalog`.
3. Go back to tab *General* and enter `Boiler_table` as *Id*.
4. In the green input field *Domain Class* press key ctrl-space and choose `democatalog::EnergyComponentCatalog` from the list.

From the above *Boiler_tables* know that they present data conforming to our `http://example.org/democatalog` data model and that boiler data are found as part — or "below" — the Energy Component Catalog.

Next, specify the lines to be displayed in the table:

1. From context menu over *Boiler_table* create `New Table Element → Line`
2. In tab *General* in *Properties*, enter `Boiler_line` as *Id*:
3. In the green input field *Domain Class* press key ctrl-space and choose `democatalog::Boiler` from the list.

And now for the columns:

1. From context menu over *Boiler_table* create `New Table Element → Feature Column`
2. In tab *General* in *Properties*, enter `Name_col` as *Id*:
3. In the green input field *Feature Name* press key ctrl-space and choose `modelName` from the list
4. Repeat the above steps for `boilerType` and `installedThermalPower` accordingly.

After addition of some foreground and background styles, the design of the UI looks like this.

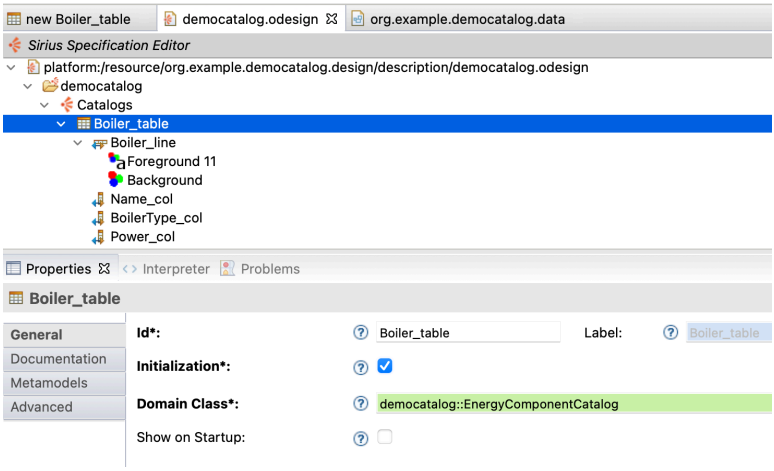

Figure 11. Boiler Tables Design

Save it!

To create an instance of the table just designed double-click on `representations.aird` in the data project:

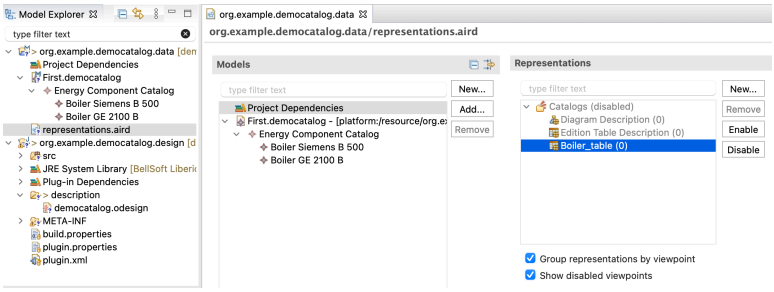

Figure 12. Administration of Model and Representations

In case viewpoint `Catalogs` under header *Representations* is still disabled as shown above, select it and press `Enable`. Then:

1. Press `New…` to open a *Create Representation Wizard*
2. Choose `Boiler_table` and click `Next >`
3. Select `Energy Components Catalog` as data source and click `Finish`
4. You are prompted for the new tables name: simply confirm the proposed name with `OK`.

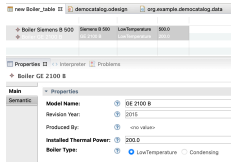

Figure 13. Boiler Table with Properties View

The screenshot on the right shows *new Boiler_table* with just two entries. Details of the selected entry are editable in *Properties*.

Is your table empty? In this case you probably did not add example data using the default editor as described above. But you can add new Boilers any time via command `New child → Boiler` in the context menu of `Energy Component Catalog` in section *Models* of the representations editor depicted above.

Note, that you can delete boilers from the table's context menu, but currently there is no button or menu entry to create new boilers. Such a command would have to be described in `democatalog.odesign` first.

Be aware that applications with UI design and example data launched from *Obeo Designer* are meant to be prototypes for the final software only. In fact, any saved

changes in the design file are instantly reflected in the UI. During refinement of model and UI, data sets can be created, edited, and tested for usability without the need to built deployable software component. (On deployment, see parts *Accessing and Using Parameter Catalogs* and *Build (Parameter Catalog) Applications with Eclipse Tycho* below.)

Iteratively the UI design must be adapted to changes in data model, although some changes are automatically reflected in the generated UI, at least for default forms. Data model changes can also can render 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.

As you may imagine, this is just the tip of the iceberg of what can be done with the Sirius framework for designing graphical UIs. While domain experts should be capable to create and to refine Ecore data models, the UI design of a parameter catalogs will mainly be done by software developers. However, since the UI is not implemented by program code, but a description in an `.odesign` file, domain experts can easily enhance and tweak it, e.g. by adding or reordering columns of a table.

2.5. Working with Git Hosted Parameter Catalogs

Ecore data models and Sirius based UI design are used to create parameter catalog software hosted in Git repositories. To work with these, all you need is Jave 16 and the *Obeo Designer* with plug-ins for handling of Units installed (see Section 2.2, “Setup Obeo Designer” for details.)

Import Modeling Projects from Git.
To connect to a Git repository open the *Import Projects from Git* wizard via `File → Import… → Git → Projects from Git → Clone URI`. Then:

1. Copy the URI of the git repository into the according input field, e.g.: https://rs-loy-gitlab.concordia.ca/parameter-catalogs-ecore/greenery-catalog.git and provide your credentials in fields *User* and *Password*. **Tick check box Store in Secure Store and provide a master password if required!** If you don’t, be prepared to be prompted for your credentials over and over again
2. Click `Next >` and select a repository branch to check out, usually *master*

3. Click `Next >` and choose the directory on your file system where to store the repository, e.g. `<user home>/git/greenery-catalog`. Here, we adhere to the convention is to have all git repositories stored in `<user home>/git/`
4. After data transfer has completed, the wizard offers to *Import existing Eclipse projects*. Click `Next >` and select the project with suffix `.model`, `.edit` and `.editor` for import, e.g. `ca.concordia.usp.greenerycatalog.model` etc.
5. Click `Finish`.

Now you can work on the data model like you did with the demo catalog. Find it under `model` in `ca.concordia.usp.greenerycatalog.model` (compare fig. Figure 3, “New Ecore Modeling Project”).

Catalog Data and UI Design.
For data inspection and editing — and possibly modifying the UI — launch a new instance of the running Obeo Designer application by executing `Run → Run Configurations…`, double-click on *Eclipse Application* to get a *New_configuration* and give it a meaningful name (e.g. *GreeneryCatalog*). Then, press `Run` to start the application, close the welcome screen and open the Sirius perspective using the suited button in the top right corner of the main window.

Simply reuse the *Run Configuration* specified above, when starting the application next time!

Now, import the projects that contain data and UI design, respectively:

1. Execute `File → Import…` for the import wizard
2. Browse to the directory containing the projects (e.g. `<user home>/git/greenery-catalog`) and check just the projects with suffixes `.data` and `.design` for import, e.g. `ca.concordia.usp.greenerycatalog.data`, `ca.concordia.usp.greenerycatalog.design`
3. Click `Finish`.

When closing the application, it asks to store or dismiss any changes in data or UI design. You can also save these any time with `File → Save All`.

Declare Quantities.

For simplicity, the demo catalog only used built-in attribute types like `EDouble`, `EInt`, or `EString`. On the other hand, real-world parameter catalogs use a custom type named *Quantity* that combines a numerical (double) value with a unit.

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 `kW·h/m³` are all valid unit definitions. This is all documented well in the resources mentioned in section Add Plug-ins to deal with Quantities and Units above, but for convenience, a table with valid units, including some specific units for urban simulation, is compiled in UnitsExamples.md.

To set an attribute's type to *Quantity* just select it in the model, choose tab *Semantic* in view *Properties*, click on *EType* and select *Quantity* from the list of available types. In the figure below, this was already done.

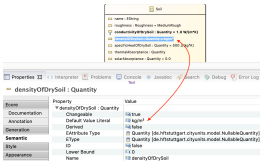

Figure 14. Quantity Default Values

The red arrow shows how a unit is defined in field *Default Value Literal*. E.g., attribute `densityOfDrySoil` has unit `kg/m³` assigned to it.

Note that, for this attribute, no numerical default value is given. In contrast, `conductivityOfDrySoil` is given a unit and a default numerical value: `1.0 W/(m*K)`.

The unit of a *Quantity* is definied by the sub-string that follows the first space character in the string given in *Default Value Literal*. The sub-string before that space is interpreted as default numerical value of the *Quantity*.

The rules for how a *Quantity* default value is converted to its unit and default (initial) numerical value are very "forgiving":

- If no unit is given or it cannot be parsed to a valid unit, it will be regarded as *dimensionless*. E.g., index values, fractions and percentages are dimensionless quantities by purpose. While units may be displayed in the UI

like `[kg]`, a dimensionless quantity will show up as `[]`, that is as the empty string.

- If no numerical default value is present, then the numerical value is regarded as undefined.
- You may choose to specify a quantity as dimensionless and without numeric default by leaving field *Default Value Literal* empty (or provide some non-sensical string).

By this rules, any string — including the empty string — will be interpreted as a *Quantity* somehow.

Declare Ranges.
What is the point in declaring a dimensionless quantity for an attribute, anyway, instead of just declare it `EDouble` or `EInt`? The answer is that quantities can — and most of the time will — have a range of valid values defined.

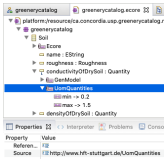

Figure 15. Quantity Range Definition

As you can see in the screenshot, the allowed range of attribute values is defined by a so called Ecore annotation named `UomQuantities`. It provides the minimal and/or maximal value for the attribute, inclusively. If a minimal or maximal value is omitted or invalid, the range is not limited on that side.

Adding annotations to an attribute does not work in the graphical Ecore editor, but only with the standard editor that is opened by `Open With → Sample Ecore Model Editor` from the context menu over the Ecore model file in *Model Explorer*.

In this editor, define a range like this:

1. From the context menu of the attribute of interest execute `New Child → EAnnotation` and type `http://www.hft-stuttgart.de/UomQuantities` into field `Source` as depicted above
2. From the context menu of the new `UomQuantities` annotation execute `New Child → Details Entry` and provide values for keys `min` and `max`, respectively.

The above typing is a one time effort only, since for defining further ranges, one simply copies an existing `UomQuantity` annotation from one attribute to another one and just edits the values for `min` and `max`.

If a catalog's end user tries to enter a number outside the given range in the UI, it will be adapted automatically to a valid value.

Declare Tooltips for Help.
Again, domain experts use a specific annotation to provide short help texts that inform end-users about an attribute's purpose, range and so on. (These texts are displayed as tooltips when the users mouse stays on top of a question mark.) And again, this is possible only in another kind of editor — this time the editor that is opened on a `.genmodel` in *Model Explorer*. Each Ecore model is accompanied by a `.genmodel` that lives besides the respective `.ecore` file. Open the required editor from its context menu with `Open With → EMF Generator`.

The picture below shows the details. Just open the `.genmodel` tree until you can select the attribute that shall be documented.

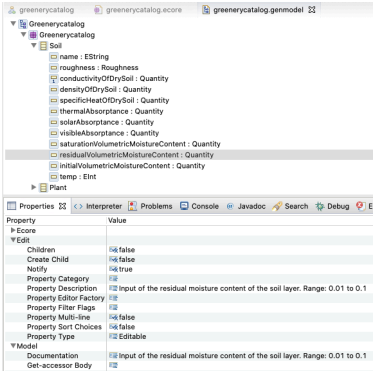

Figure 16. Quantity Documentation

In its `Properties` provide the tooltip in field `Edit → Property Description`. In this example, the same text was also copied to `Model → Documentation`. These texts are automatically inserted into comments in the generated program code, so that they can inform a programmer that wants to use the generated API.

2.6. Summary

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

We get to know the *Obeo Designer* 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 implemented a prototypical user interface for this data with *Eclipse Sirius* by providing a `.odesign` model for that very UI.

Lastly, we started working on real world parameter catalogs hosted in git repositories and introduced *Quantity* as a custom attribute type to model quantities as numerical values with defined units.

3. TBD: Accessing and Using Parameter Catalogs

3.1. Accessing XML-Catalogs from Java

3.2. Create Insel 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*[41] 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.

[41] https://www.eclipse.org/m2e/

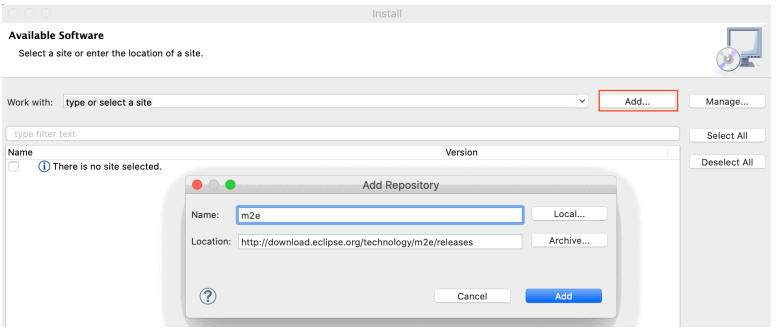

Figure 17. Add update site m2e

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.

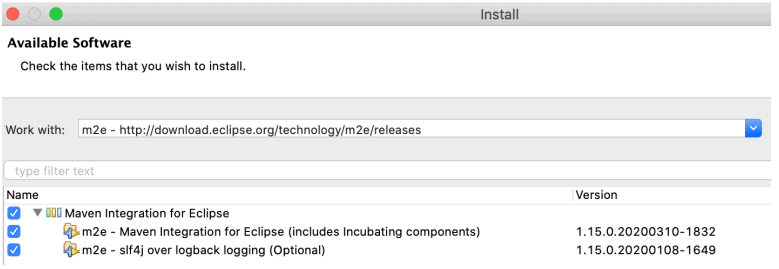

Figure 18. Choose features to install

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 19. 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