Deliverable 2.1
Software Design Document

Revision............................. 0.1
Submission date ...................... 2017-12-09 (m3)
Due date ............................... 2017-11-30 (m2)
Lead contractor ....................... ER

Authors:
- Pekka Savolainen............... VTT
- Juha Kiviluoma ................. VTT
- Erkka Rinne ........................ VTT
- Joseph Dillon ..................... ER
- Kris Poncelet..................... KUL
- Manuel Marin ..................... KTH

<table>
<thead>
<tr>
<th>Dissemination level</th>
</tr>
</thead>
<tbody>
<tr>
<td>PU Public</td>
</tr>
<tr>
<td>CO Confidential, only for members of the consortium (including the Commission Services)</td>
</tr>
</tbody>
</table>
**Description of the related task and the deliverable.**

Extract from DoA

T2.1 High level system design

Task leader: ER; Participants: VTT; Duration: M01-M30

This task will use key concepts and ideas presented in the proposal and in literature to create a high-level design of different tools and a shell framework that governs their interactions. The design will be informed by the needs of the case studies (WP6) by employing use cases. The design will address the modular composition of the tools. It will sketch a user interface for all parts of the modelling chain to guide Task 2.4 ‘User interfaces’. The task will produce a high-level software design document, which will define the requirements for the shell and the various components, the detailed design of which will be carried out in later tasks for specific tools. This task will commence when the project starts and will result in a first draft of the design document in the first month. During the design of the component tools, the design document will be kept updated and this task will resolve any arising interoperability issues.

... 

D2.1 : Software Design Document

- First high level version M02
- More specific design from tasks 2.2 - 2.7 will be used to update the document
- Final version M30

**Planned resources PM of T2.1**

<table>
<thead>
<tr>
<th>VTT</th>
<th>UCD</th>
<th>KUL</th>
<th>KTH</th>
<th>ER</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.0</td>
</tr>
</tbody>
</table>

**Comments**

<table>
<thead>
<tr>
<th>V</th>
<th>Date</th>
<th>Authors</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2017-12-09</td>
<td>VTT, ER, KUL, KTH</td>
<td>First high level version</td>
</tr>
</tbody>
</table>

**Disclaimer**

The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information as its sole risk and liability. The document reflects only the author’s views and the Community is not liable for any use that may be made of the information contained therein.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under Grant Agreement No. 774629. Topic: LCE-05-2017 Tools and technologies for coordination and integration of the European energy system
# Table of contents

Abstract......................................................................................................................4

1. Introduction............................................................................................................5

2. System Overview...................................................................................................7
   2.1 Description of Main Building Blocks.................................................................7
   2.2 Constructing Modelling Tasks from Building Blocks ........................................9
   2.3 Execution of Tools............................................................................................10
   2.4 Viewing and Editing Project Data.....................................................................11

3. Spine Toolbox Use Cases...................................................................................13
   3.1 Manage Project Use Case................................................................................13
   3.2 Generate Data Collections Use Case...............................................................14
   3.3 Execute Project Use Case................................................................................15
   3.4 Manage Output Data Use Case.........................................................................16
   3.5 Set Up New Tool Use Case..............................................................................16

4. Requirements, Assumptions, Dependencies, and Constraints.......................18
   4.1 Functional Requirements................................................................................18
      4.1.1 UI Requirements......................................................................................18
      4.1.2 Data Management Requirements............................................................19
      4.1.3 Project Execution Requirements.............................................................20
   4.2 Non-functional Requirements..........................................................................21
      4.2.1 Implementation Language and Dependencies........................................21
      4.2.2 Coding Style............................................................................................23
      4.2.3 Versioning, Version Control and Deployment............................................23
      4.2.4 Security and Testing Requirements..........................................................25

5. System Architecture............................................................................................27
   5.1 Model Classes.................................................................................................29
   5.2 View Classes.................................................................................................29
   5.3 Controller Classes.........................................................................................30
   5.4 Design for Tool Execution.............................................................................30

6. Spine Data Structure...........................................................................................32

7. References ...........................................................................................................33
Abstract

Spine Toolbox is an application that provides means to define, manage, and execute energy system models. It gives the user the ability to collect, create, organize, and validate model input data, execute a model with selected data and finally archive and visualize results/output data. Spine Toolbox is designed to support the creation and execution of scenarios using the Spine Model. This is an important part of the application but it can also support other models and tools as long as they follow the conventions of Spine Toolbox or there is an interpreter between the application and the external tool. One of the conventions is the Spine data structure, which is an entity-relationship data model for a structured yet flexible storage of data. The interface to the data structure is an integral part of both Spine Toolbox and Spine Model because it enables them to communicate using a common vocabulary. Spine Toolbox will be implemented in Python and the Spine Model in Julia.

This deliverable presents a high-level software design for Spine Toolbox and for the various tools it supports. It contains the system overview, application use cases, functional and non-functional requirements, chosen implementation language(s), dependencies, versioning, application validation requirements, testing and security requirements, and notes on the enforced coding style. The aforementioned have been collected in co-operation with Spine members and stakeholders, who will also be the first users of the application. The last two chapters; system architecture and the Spine data structure, will be updated as plans for the design become more evolved during the project and the implementation.

This first version of this deliverable is due in M02 of the project and the final version is due in M30. Spine is an open-source project. In the fall of 2018, Spine Toolbox source code and documentation in their then current form will be released to the public via a web-based version control repository. The chosen license for Spine Toolbox is likely to be GNU Lesser General Public License (LGPL). Spine Toolbox documentation, manual and all original graphics will be released with the Creative Commons BY-SA 4.0 license. We hope to attract a lively and active community around Spine that will continue the development even after the project has ended.
1. INTRODUCTION

Spine Toolbox is an application, which provides means to define, manage, and execute energy system models that are becoming increasingly complex. It gives the user the ability to organize, collect, create and validate model input data, execute a model with selected data and then archive and visualize the output data. Spine Toolbox (i.e. the application) provides the following features for the energy system model developer:

- Scenario construction
- Data management & validation
- Data conversion & verification
- Energy system model execution
- Result data visualization

Spine Toolbox will be implemented as a cross-platform desktop GUI application for Windows, Macintosh and Linux platforms. Application architecture will be designed to support also other user interfaces, such as a command line interface or a HTML5 based interface. However, desktop application is the first priority. Spine Toolbox will be released as an open-source project to the public in the fall of 2018. The envisioned license for the application and its code is GNU Lesser General Public License (LGPL). You can find details about the license on the Free Software Foundation (FSF) website\(^1\). Spine Toolbox documentation, manual and all original graphics and icons will be released with the Creative Commons BY-SA 4.0 license.

This deliverable presents a high-level software design for Spine Toolbox and for the various tools it supports. First, a general overview of the software system is given. This is followed by Spine Toolbox use cases, which have been defined in cooperation with the first users of the application. These define the requirements that the application needs to fulfill. The requirements are used to construct the architecture of the application. Logical view of the software system is presented as UML class diagrams, which show the static relationships between parts of the system. They contain the main attributes, operations and interfaces between different parts of the design. All classes have their distinct characteristics and interfaces that need to be defined carefully so that the design becomes modular and easier to maintain. Another important part of this deliverable is the Spine general data structure and its interface with Spine Toolbox. This must be defined with care so that discussions between the application and the Spine Model developers become more efficient and unambiguous.

The first version of this deliverable, which is due in M02, is intended as a high-level software design document that can be used to start the implementation of the application. The goal is to present a concise and easy to read blueprint on how the application will be implemented. The intention is to update this deliverable as plans become more detailed during the project. The final version of this deliverable is due in M30. The software code and documentation will be made public via a web-based version control repository service such as GitHub or BitBucket in M12. The initial development is done using a private version control repository that is open to all project members.

The deliverables and tasks depicted in Figure 1 are interconnected to this deliverable. The high-level system design is the task of T2.1 and it is documented in the first release version of D2.1. Later, the other tasks (T2.2-T2.8) of WP2 will complement D2.1 with the topics listed in the figure. This deliverable provides input for the Spine Model documentation (D3.1) as well as for the Data Acquisition and Conversion Tools deliverable (D4.1). D4.1 in turn will provide input for deliverables D5.1 and D5.2. Table 1 contains the most important terms and their definitions used in this deliverable.

\(^1\) https://www.gnu.org/licenses/
Figure 1. References to other documents and tasks.

Table 1. Definitions.

<table>
<thead>
<tr>
<th>Term</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case study</td>
<td>Spine project has 13 case studies that help to improve, validate and deploy different aspects of the Spine Model and Spine Toolbox.</td>
</tr>
<tr>
<td>Model</td>
<td>Refers to two things in this deliverable depending on the context. It refers to energy system models everywhere except in the software architecture context (chapter 5). In chapter 5, model refers to data in a general sense. The context should be evident. Spine Model is an energy system model and is not called a ‘model’ in the software architecture context.</td>
</tr>
<tr>
<td>Scenario</td>
<td>A scenario combines data collections to form a meaningful data set for the target tool</td>
</tr>
<tr>
<td>Spine data structure</td>
<td>Spine data structure defines the format for storing and moving data within Spine Toolbox. A generic data structure allows representation of many different modelling entities. Data structures have a class defining the type of entity they represent, can have properties and can be related to other data structures. Spine data structures can be manipulated and visualized within Spine Toolbox while Spine Model will be able to directly utilize as well as output them.</td>
</tr>
<tr>
<td>Spine Model</td>
<td>An interpreter, which formulates a solver-ready mixed-integer optimization problem based on the input data and the equations defined in the Spine Model. Outputs the solver results.</td>
</tr>
<tr>
<td>Spine Toolbox</td>
<td>A desktop application to define, manage, and execute various energy system simulation models. Provides tools to organize and collect input data and tools for post-processing and visualizing output data.</td>
</tr>
<tr>
<td>Task</td>
<td>A piece of work to be done or undertaken by Spine Toolbox.</td>
</tr>
<tr>
<td>Tool</td>
<td>An external tool/model that the application is able to execute. A tool object contains a reference to the model code, external program that executes the code, and input data (e.g. files) that the model code requires. Spine Toolbox executes the tool in a sub-process. Spine Model is also an external tool.</td>
</tr>
<tr>
<td>Use case</td>
<td>Potential way to use Spine Toolbox. Use cases together are used to test the functionality and stability of Spine Toolbox and Spine Model under different potential circumstances.</td>
</tr>
</tbody>
</table>
2. SYSTEM OVERVIEW

Objective of Spine is to allow the implementation of a wide range of energy system models that will vary significantly in geographical, sectoral and temporal scope. To support this, Spine Toolbox will utilize problem independent data and user interface structures. A problem independent structure will be able to support many kinds of modelling problems, which is important as the types of modelling problems are likely to change significantly over time. Even though the structure will be problem independent, Spine still aims to make it easy and intuitive for the user to define the problem to be solved.

The main idea for the application is to give the user a set of building blocks that can be used to construct a project that can solve a problem. Projects contain the necessary input data and the selected external model(s) and it can be executed to produce results. An external model is an external program that the application is able to execute. In Spine, external models are called tools. A tool object contains a reference to the tool code, external program that executes the code, and input data (e.g. files) that the tool requires. Spine Model will be one of the tools that Spine Toolbox supports.

2.1 Description of Main Building Blocks

Spine Toolbox is project based - the user starts a separate project for each purpose. Project is a Spine Toolbox concept that can be used to save user’s work so that the user does not have to restart his or her work every time the application is started. A project consists of a data processing chain that is built by the user by combining the available building blocks. Project either contains the data or has references to the data that the project’s tools require. In addition, project contains any user preferences that can be configured in the application. The main building blocks for the data processing chain are data store, data collection, data manipulator, tool, and view. In order to form a data processing chain, the main building blocks are connected into each other either with a one-to-one relationship or a one-to-many relationship. The connection between blocks is visualized to the user by drawing an arrow from a block to the blocks that are further down in the chain. The descriptions for these blocks and their (initial) icons are given in Table 2.

Table 2. Main building blocks of Spine Toolbox

<table>
<thead>
<tr>
<th>Building Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Store</td>
<td>Contains the (main) data used in a Spine Toolbox project. The contents can be defined and modified in Spine Toolbox. Most often contains the base scenario for the project that can then be manipulated further before processing the scenario with a tool. Data within the data store follows the Spine data structure format.</td>
</tr>
<tr>
<td>Data Collection</td>
<td>Data collection can be a connection to an original data source, e.g. spreadsheet file or a database. In this case, data in the source file needs to conform to the format that Spine Toolbox is capable of reading. Data collection can be a combination of other data collections. It can also be manipulated using data manipulators. Data collection can also be defined in Spine Toolbox (i.e. native data collection). Data within data collections follows the Spine data structure format.</td>
</tr>
<tr>
<td>Filter</td>
<td>Manipulators transform the data coming from a data collection, data store or a tool before passing it onward. They are used to verify, validate, change, combine, or create data. Manipulator types include a filter, a recipe, and a randomizer but others could be added in the future. Manipulators create alternative scenarios when they define an alternative value for a particular data item. Manipulations take place within Spine Toolbox.</td>
</tr>
</tbody>
</table>
Recipe

Tools process data that is passed to them in order to produce an output data store. Tools are executed as independent applications. Spine Toolbox needs to know how to call the tool and how to pass the data collection to the tool. Tool types include models (including the Spine Model), data verifiers, data conversion tools, and display tools. A tool has its own source, which is defined in the same way as a connection to an outside data collection. Typically, this would be a connection to a Git repository. Tool code can also be manipulated within Spine Toolbox. Tools developed within Spine can directly exploit the Spine data structure. Other tools require a translation from the Spine data structure to the tool format (and possibly back) and this has to be built for each tool separately.

View

View is like a filter but is used to display a selected part of data from a data store or from a data collection. Data in a view can be visualized in Spine Toolbox or it can copied to the clipboard so that it can be visualized or modified further in another program. The display process can be automated, so that the Spine Toolbox shows the results from a tool after the tool has executed.

Scenarios are formed from data stores and data collections whenever there are parallel values for particular data items. When a data store that contains parallel scenarios is passed to a tool, each scenario is executed separately (although this may happen in parallel). Scenario definition contains the full pathway how the scenario has been created, but the user can also name scenarios. Scenarios include new data stores and data collections as they progress through the execution chain. A scenario going into a tool is an input scenario and the scenario coming out from a tool is a result scenario. The result scenario is consequently a child of the input scenario and the tool in question. A project may contain multiple scenarios or just a single scenario.

Recipe is a Spine Toolbox manipulator that can be used to make new executable tasks by combining tasks that are already present in the project. The way that these combinations are made is defined as an equation in a recipe block. This is an effective and a quick way to make a large number of new tasks. One example of a recipe is a Cartesian product but the application should support other combinatorial mathematics as well. If the recipe block had two input data collections, then the output from the recipe would be the Cartesian product of these two data collections. Table 3 contains an example of how recipes can be used. Below the header row, there are nine tasks organized on five levels. A new task is produced from the table with a Cartesian product of tasks with N levels. If the user chooses to include all five levels then the number of potential task groups would be $2 \times 1 \times 3 \times 2 \times 1 = 12$. Three example tasks with five levels would be:

- Low VG prices.Balmorel.2010.JMMwHist.JMMpp
- Low VG prices.Balmorel.2011.JMMwHist.JMMpp
- Low VG prices.Balmorel.2012.JMMwHist.JMMpp

If the user chooses to ignore some levels in the table, the number of potential tasks raises even higher. For example, one task with four levels could be:
- High VG prices.Balmorel.2012.JMMwStoSSch

Table 3. Tasks in a table

<table>
<thead>
<tr>
<th>Level name</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario data</td>
<td>Low VG prices</td>
</tr>
<tr>
<td></td>
<td>High VG prices</td>
</tr>
<tr>
<td>Investment model</td>
<td>Balmorel</td>
</tr>
<tr>
<td>Time series</td>
<td>2010</td>
</tr>
<tr>
<td>Operational model</td>
<td>JMMwHist</td>
</tr>
<tr>
<td></td>
<td>JMMwStoSSch</td>
</tr>
<tr>
<td>Operational</td>
<td>JMMpp</td>
</tr>
</tbody>
</table>

2.2 Constructing Modelling Tasks from Building Blocks

Figure 2 shows an example of a simple modelling task that could be represented in Spine Toolbox as a project. An initial working name for the part of the application where the building blocks are inserted to is called a project canvas. Data store on the left hand side of the project canvas contains the base scenario data. This is combined with a data collection with model definitions and sensitivities by using a recipe block. The output of the recipe block is given to the external model block (tool) for execution. Tool output is saved into another data store block that contains the results. This is a simple project that basically just passes data through the tool to produce results.

Figure 2. Simple project in Spine Toolbox.
Figure 3 shows an example case where two models are connected in series and the latter is using the results of the first one. Model input data is gathered in data collections, which can be filtered to select only parts of the data. Intermediate input data is stored in data stores. Base input data is combined with a data sensitivity collection in order to create a number of scenarios. The user can examine the results using views into the data. Data collections can connect to different data sources, such as Excel files, or SQL databases. To make these data sources compatible with the Spine data structure format, data conversion tools must be developed. They are not depicted in the figure but it is likely that these will be added as their own building blocks to the project. Data collections can also be directly compatible with the Spine data structure format. The models can be executed separately or the whole project can be executed at once.

2.3 Execution of Tools

One of the main features of Spine Toolbox is the ability to execute various energy system models. From the application’s point of view, energy system models are external models, which means that they are separate and independent programs that may have been (and most are) implemented with a different programming language than Spine Toolbox. These external models are called tools in Spine Toolbox. There are essentially two options in how the application can execute them. The first option is to run the external model in a sub-process that is spawned by the main process. The second option is to embed some portion of the model into the application. Both approaches have advantages and disadvantages. There are two main advantages in the sub-process option. The first is compatibility with any model or program and the second is the option to execute the modeling task in a computing cluster. The disadvantage of the sub-process option is that the access and control of the model is very limited from Spine Toolbox while the execution is running. On the other hand, embedding the model into Spine Toolbox would give us more access and control of the model but adding compatibility for a new model would require a significant amount of work. In addition, there is a risk that every time a tool is updated, Spine Toolbox needs to be updated as well. As one of the requirements of Spine Toolbox is to be compatible with a vast number of different models, we have selected the sub-process option. Figure 4 depicts the components involved in executing a tool in a sub-process. Essentially, a
tool consists of three ingredients; the tool code, the program that executes the tool code (external program) and the input files that the tool code requires. The responsibility of Spine Toolbox is to make a tool instance (depicted as a blue tool box) that can be submitted to a sub-process for execution. The output from a sub-process depends on the actual software library, the external program and the operating system that is used to invoke the sub-process. The most typical output values from a sub-process we can get are a process ID (PID), a process state, an exit status, and a return code. When a tool has finished execution, the main process (Spine Toolbox) is notified and the results can be viewed or visualized. Implementation details are given in the system architecture chapter.

![Diagram of tool execution in a sub-process](image)

**Figure 4. Tool execution in a sub-process.**

### 2.4 Viewing and Editing Project Data

The data store block represents some or all of the data in a project in the Spine data structure format. The user is able to access and manipulate the data by opening the data store view. The user may do this for example by double-clicking on a data store block or there might be a shortcut to project data in the application. An example view of a data store is depicted in Figure 5. Users can configure the data to suit the needs of the project they are working on by adding, editing, and importing objects. In addition, copying or pasting entries from the clipboard will be supported. One possibility for storing the data is a SQLite database that could then be used in the Spine Model or other tools directly, as there are bindings for SQLite for both Python and Julia available. There are also other options that are being considered for storing the data, such as MySQL, so the final decision has not been made yet. Spine data structure is described in its own chapter later in this deliverable.
Figure 5. Data store view.
3. SPINE TOOLBOX USE CASES

This chapter describes the main requirements for Spine Toolbox as use cases. Use cases describe critical behavior of the application. The use cases have been defined by the whole project group together in order to find common ground between the developers and the users. This way, the software designers and the people carrying out the implementation can focus on the essential and avoid doing unnecessary work. Figure 6 depicts a high-level use case diagram of the main use cases in Spine Toolbox. It serves as a table of contents for the individual use cases.

![High-level Use Case Diagram]

**Figure 6. High-level Use Case Diagram.**

### 3.1 Manage Project Use Case

<table>
<thead>
<tr>
<th>Actors</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>User starts the application and either creates a new project or loads an existing one.</td>
</tr>
</tbody>
</table>

Main course:

1. User starts the application. The application starts as ‘blank’ with no project open.
2. User selects ‘Create project’ option from a dedicated button or a keyboard shortcut. The system displays the ‘Create project’ view.
3. User gives a name and a description for the project.
4. User sets other configurable options for the project.
5. User clicks ‘Ok’ button when he/she is happy with the project. The system closes the ‘Create project’ view, sets up the project (i.e. creates necessary folders and files) and presents the main application view with an empty project.
6. User is happy with the project, clicks ‘Save project’ button (or keyboard shortcut) and closes the application.

Alternative course:
1. User clicks on ‘Load project’ button (or presses keyboard shortcut). The system opens a view, where the user can browse his/her local or network folders.
2. User browses to the project file location that he/she wants to open and clicks ‘Open’ or double-clicks the file. The system closes the file browser view, loads the project, sets up the project settings and shows the main application view.
3. The main application view shows the project canvas, which contains the data collections, tools, manipulators and data stores that were saved into the project.

### 3.2 Generate Data Collections Use Case

<table>
<thead>
<tr>
<th>Actors</th>
<th>User, ODBC database, Input data file (e.g. text, spreadsheet, or binary file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>User wants to create, connect and manage data collections.</td>
</tr>
</tbody>
</table>

**Preconditions:**

- User has loaded a project or created a new one

**Post-conditions:**

- Data collections are ready to be passed on to data stores, tools or manipulators
- Data collections are represented in Spine data structure format

**Main course (Create new data collection):**

1. Project canvas is displayed in the application main window. It may be empty or already contain data collection icons, tool icons or other icons (See e.g. Figure 2).
2. User clicks new data collection and enters the name
3. The user clicks on the start-from-scratch button
4. A window opens which gives an overview of all types of data/data collections, which may be specified. For example, different sheets, each containing certain type of data. Examples of sheets can be system settings, geographical information, temporal information, technology specification, policy choices, demand, renewable time series, etc. An example of a window in which the user can manage or view data may look similar to the data store view depicted in Figure 5. Note: A window like this also opens when the user selects a data collection and clicks ‘view data’.
5. When the user is happy with the data collection, he/she clicks ‘Finish data collection’
6. Data collection icon with the given name is added on the project canvas

**Alternative course one (Connect data collection):**

1. Project canvas is displayed in the main window
2. The user clicks new data collection and enters the name
3. The user selects the source of data to be imported into the project. The source can be database, Excel file, text file (e.g. CSV) or a binary file.
4. The application checks that the data is in a supported format
5. The application makes preliminary validation that data is in a format that can be converted into Spine data structures
6. The application converts the data into Spine data structure format
7. User can view and browse the new objects in the Spine data structure
8. If the user is happy with the data collection, he/she clicks ‘Finish data collection’
9. Data collection icon with the given name is added on the project canvas

**Alternative course two (Make solver settings data collection):**

1. Project canvas is displayed in the main window
2. The user clicks new data collection and enters the name (e.g. solver settings)
3. User makes a new file (or modifies an existing file), which contains solver settings for some processing tool
4. When the user is happy with the file he/she clicks ‘Finish data collection’
5. Data collection icon with the given name is added on the project canvas

The solver settings data collection is now ready to be connected into a processing tool icon, which represents the simulation model (see Figure 7)

4. When the user is happy with the file he/she clicks ‘Finish data collection’
5. Data collection icon with the given name is added on the project canvas

The solver settings data collection is now ready to be connected into a processing tool icon, which represents the simulation model (see Figure 7)

![Figure 7. Two data stores, a data collection with solver settings and a processing tool.](image)

### 3.3 Execute Project Use Case

<table>
<thead>
<tr>
<th>Actors</th>
<th>User, Database</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Summary</strong></td>
<td>The user wants to use a computational tool to create a set of results from input data collections.</td>
</tr>
</tbody>
</table>

**Preconditions:**

- User must have a project open
- Project must contain an external model and some input data for it

**Post-conditions:**

- Results from tools are ready to be archived or visualized.

**Main course:**

1. Optional: user selects where to save the results and other files following from executing a tool (otherwise default place)
2. Optional: user selects whether or not he/she sees the progress of the tool in case the tool supports this.
3. Optional: User clicks ‘validate project’ to check if the processing tools in the project have valid input data or if there are errors in the combination of data collections.
4. Optional: User selects a processing tool icon and clicks ‘validate’ to check if that processing tool has valid input data available in the project.
5. User starts the execution by either
   a) Clicking on ‘execute project’ button. The whole project will be executed, or by
   b) Selecting a portion of the project from the project canvas and clicking ‘execute selected’ button

6. The application can either check that all processing tools have valid input data available or it can skip this step. This is a user configurable setting. Note: For the input data validation to work, the processing tool must be configured in a way that this information is available for the project.
7. The user receives error messages, warning messages or ‘Everything went fine’ message
8. Optional: user receives more detailed information (objective function, model and solver status, computation time, etc.)

### 3.4 Manage Output Data Use Case

<table>
<thead>
<tr>
<th>Actors</th>
<th>User, Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>The user wants to archive and visualize output data (results) from simulation models. He/she wants to know which data and code using which settings resulted in which output. He/she also wants to know how long it took to run the simulation, when the simulation was started and when it ended as well as if there were any errors. User wants a visual representation of the data for which a graphing tool is needed.</td>
</tr>
</tbody>
</table>

Preconditions:
- A processing tool must have finished (either with or without errors)

Post-conditions:
- Result files are archived so that they contain all the necessary information on how these results were calculated. This includes at least metadata about the input data collections or data stores, filters that were used, and relevant information about the processing tool settings that were used.

Main course:

1. User selects a data store block, which contains results and clicks on ‘view results’ button
2. A window opens with all relevant information about the run that produced these results. This window also contains an area with the result data in Spine data structure format. The view can be similar to the example in Figure 5. This view, however, provides restricted editing features compared to the create data store view.
3. User selects the data that he/she is interested in and clicks ‘Make graph’. Note: What kind of graphs are available will be decided later but at least some time series data may be visualized.
4. The application makes a sanity check for the data and if this passes then a graph is drawn into a new window.
5. User wants to export this graph into an image so he/she chooses ‘export as .png’, selects a file name and clicks ‘Ok’.

### 3.5 Set Up New Tool Use Case

<table>
<thead>
<tr>
<th>Actors</th>
<th>User, Version control system (Git)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>The user wants to use a new tool with Spine Toolbox. The new tool will be available in consequent sessions.</td>
</tr>
</tbody>
</table>

Main course:

1. User has a tool (e.g. simulation model) that could be used in Spine Toolbox
2. The user selects ‘Add New Tool’ feature in the application
3. User enters data needed for Spine Toolbox to understand the new tool. This includes at least, new model main program file name and the path to it and an external program that is needed to run the model. In addition, location of input data that the model requires must be entered.
4. Optional: user may give the main program file name as a Git commit name, in which case also the Git repository must be given.
5. User applies changes
6. When a user adds a new tool block on the project canvas, the new external model is now available.
4. REQUIREMENTS, ASSUMPTIONS, DEPENDENCIES, AND CONSTRAINTS

This chapter presents the main requirements for Spine Toolbox. They have been categorized into functional and non-functional requirements. Functional requirements, in general, include data manipulation features, UI views, or other specific functionality that define what the application is supposed to accomplish. Non-functional requirements specify criteria that can be used to judge the operation of a system. In the following sub-chapters, the functional requirements have been categorized into UI, data management, and project execution requirements. Non-functional requirements are categorized into dependency, end-user environment, verification, validation, security, and performance requirements. Functional requirements have been collected from discussions with Spine project members who will be the initial users and testers of the application.

4.1 Functional Requirements

The following requirements form a list of features that have been requested by Spine project members. They are not in any particular order and they are not by any means complete. These requirements have been used to make a more complete list of features categorized by application release version number. This is discussed later in this chapter (see Versioning sub-chapter). An up-to-date list will be kept in a selected web-based project management and issue tracking application.

4.1.1 UI Requirements

- Project based workflow:
  - User can save the project so that the project can be loaded the next time the application is started. If user already has a project, it may be loaded.
- Graphical and user-customizable representation of the data processing chain:
  - Projects contain a canvas, where the user can add different blocks that can be moved, copy-pasted and connected to each other. These blocks represent data collections, manipulators, tools, or other items that the user might need in his/her work. The idea is to enable the user to make a complete visual data processing chain from the input data files to the result data files.
- Show only part of the hierarchy:
  - When defining relations between data collections and tools, the screen can become cluttered if it is a big modelling exercise. Consequently, it should be possible to hide detail. E.g. show only a higher level data collection and hide those that are underneath. Then when taking the mouse cursor over the higher level data collection, show what’s underneath.
- Undo/redo action(s):
  - When using the graphical user interface it should be possible to undo and redo changes. These include moving building blocks, including new items, deleting existing items, changing filters, and recipes. Make possible history of changes with undo/redo for specific classes (probably not used for time series since that would be too much data)
- Visualization of self-made constraints:
  - Self-made constraints as text within Spine Toolbox?
- Symbols to allow replacement and calculation of values
  - Values in the data store could utilize symbols and the actual values to be sent to the tool would be solve by calculating the value field based on the defined values for the symbols.
4.1.2 Data Management Requirements

- Create data collection:
  - Users can create a new data collection in an interface that can at first be just a text field. Later it could be a table (that should conform to the same principles as a spreadsheet table that is readable by the application). This data collection is saved in Spine data structure format.

- Connect data collection:
  - Users can connect data collections to a project by selecting them from the file system. The application must check that it can read the data collection into Spine data structure format.

- Set alternative data collections:
  - One data collection can be replaced by another
  - Enable creating new data collections by combining existing ones. This can be achieved for example by:
    - Setting data collections into a hierarchy so that a higher level data collection includes lower level data collections (Tree structure i.e. parent-child relationship)
    - Defining a recipe. Users can connect data collections by using combinatorial mathematics.

- Generate random data collections:
  - The user wants to generate multiple data collections for a Monte Carlo simulation. He or she selects the ‘base-case’ data collection, specifies which variables are random and their distribution. Finally, the user selects the number of data collections to generate and launches the process. The generated data collections could then be saved as part of the project. This is the job of the Randomizer block.

- Compare data between data collections:
  - Highlight differences between data collections. Most obvious case is to compare the input data collections of two or more scenarios (our output). This should probably be a separate tool or manipulator.

- Create and apply a filter:
  - User can define a filter for the selected data collection that passes only part of the data onward (and in doing so creates a reduced data collection). Filter can be active or inactive. A filter should be able to recognize wildcards and negatives.

- Select when to store a data collection as Spine data structure:
  - Data collections are connected to the application by pointing out the source, which it can read. If this data collection is then connected to a tool, it gets converted to Spine data structure format on the fly when its execute scenario task is executed. However, a data collection could also have a property where it stores the data collection in a Spine data structure file. In this case, the Spine data structure would be generated from the sources only when the data collection is flagged for rereading (or re-execution might be a better word - reading the data from sources is a task after all).

- Create a base scenario from data collections:
  - The user wants to create a scenario, give the scenario a name and save it. The user wants to specify the specifics of this scenario (i.e., the user wants to select the set of ‘data collections’ which together form the scenario. This can be done by collecting the desired data from different data sources into a data store block, which is saved as a Spine data structure.
- Create derivative scenarios by combining the base scenario with data collections that define sensitivities (changes to the base scenario) using recipes
- Create a set of scenarios:
  - The user wants to create a number of similar scenarios
- View data within a scenario:
  - Users need a view into scenario data. (e.g. as in Figure 5)
- Tool code can be manipulated within Spine Toolbox:
  - E.g. by using alternate versions from Git is one way, but there could also be a regular expression enabled script editor embedded in the application.
- Choose Spine data structure storage format:
  - Find the best way to implement views on the data store and data collections and decide the database format (MySQL, SQLite?)
  - Consider the trade-off between efficiency and usability for key-value vs. entity tables (extra zeros in the entity table, but easier to implement)
- Enable tracing back from the values (processes in the case of Spine Model) to the linked equations

4.1.3 Project Execution Requirements

- Pass through data:
  - A simple first working version of the application, where it can connect to a data collection, send it to a tool and receive the resulting output data collection.
- Execute several tasks:
  - When the user wants to execute some or all scenarios in the project, the application forms an execution pipeline of tasks and starts to execute them.
- Parallel execution:
  - When parallel execution is enabled, the application can send multiple tasks at once to be executed on different threads/machines depending on the allocated computational resources. Once there is room for new tasks to be executed, the application allocates them as well.
  - Users can shut down their own computer and be informed when the execution has finished.
- Allocation of computational resources:
  - User should be able to tell the application the different places where it is ok to compute tasks. This should probably be a feature separate from a project. A specific computer has naturally its own resources available, but in addition, outside resources can be appointed by giving appropriate handles to file systems and computation units.
- Change what is to be executed:
  - When Spine Toolbox is executing tasks, it should be possible to change the recipes, data collections and relations between data collections and tools so that Spine Toolbox will understand what is still valid and what is not. Preferably, this should include some kind of activation button, so that the user can design changes without actually forcing the changes before the activation is engaged. In addition, user can mark finished tasks to be re-executed.
- See the progress:
When Spine Toolbox is executing tasks, see which tasks are finished, which tasks have been executed, and which tasks are still pending.

- Compare computation algorithms:
  - The user can compare the results of two or more similar computation tools. Some or all input data is common to all tools. Examples include:
    - Two different energy system models for the same purpose
    - Two development versions of the same computation process
    - One process with different parameter settings

### 4.2 Non-functional Requirements

#### 4.2.1 Implementation Language and Dependencies

Spine Toolbox will be implemented in Python\(^2\). It is an open source dynamically typed programming language that was created by Guido Van Rossum in 1990. Today, it is one of the top ten most referenced computer languages on the Internet\(^3\). Here are some of the highlights of the language:

- Language and its standard library are intuitive and easy to learn
- Beginners can become productive with Python very quickly
- Experts can exploit its vast advanced features, such as partial function application, metaprogramming, and threading
- The language has a simple yet elegant object-oriented design
- Python code is easy to read and write
- The language is highly scalable. It is used for projects varying in size from hundreds, to hundreds of thousands of lines of code
- It is well suited for rapid development and makes refactoring easy
- Programs are portable across platforms
- Python is easily extensible, by writing custom libraries in Python, or by writing extensions in other languages such as C and C++
- Programs are concise. A Python solution is about 50% the size of a comparable C++ solution.

Overtime, Python has evolved to an even cleaner and more accessible language. The Python Software Foundation is dedicated to improving and developing the language. A new version of the Python interpreter is released annually, and smaller updates and bug fixes are released constantly. A major revision of the language was done in 2008 when Python 3.0 was released. This was a somewhat controversial move for Python because version 3.0 was not backwards compatible with older 2.x versions. In general, for beginners the 3.0 version offered an even cleaner and more readable language but there is still a large community of developers that are using the 2.x version. Python 2.7 is the most popular of the 2.x versions and there are minor fixes still being released for it but the main development of the language has been in 3.x for years now. Python 2.7 maintenance will stop sometime in 2020, so it makes sense to use Python 3 for Spine Toolbox implementation. The most recent Python now available is 3.6 (as of November 2017) and we will be using it because of the active development, bug fixes, and the most advanced features. There will be occasional tests for earlier 3.0-3.5 versions but the main support is for versions 3.6+.

There are a number of Python frameworks available for developing cross-platform GUI applications. Most of the frameworks are based on the same cross-platform GUI technologies, of which the most popular are Gtk, Qt, Tcl/Tk and wxWidgets. Python developers can access these technologies by using GUI frameworks such as Tkinter, PyGObject, PyQt, PySide, or WxPython. Tkinter is actually part of the Python standard library and it provides an interface to Tcl/Tk. It is quite handy and versatile for small to medium size applications. However, it is not very well suited for handling large amounts of data that Spine Toolbox must be able to handle. PyGObject is a library of bindings to

---

\(^2\) [https://www.python.org/](https://www.python.org/)

\(^3\) [https://www.tiobe.com/tiobe-index/](https://www.tiobe.com/tiobe-index/)
GLib/GObject/GIO/GTK+ (implemented mostly in C/C++). It is mostly used for GNOME application development and as such is not a good candidate for Spine Toolbox, because the main development and main end-user environment is envisioned to be Windows. WxPython is implemented as a set of Python extension modules that wrap the GUI components of the wxWidgets library, which is written in C++. However, it still not quite ready for Python 3. Both, PyQt and PySide are libraries of bindings for the Qt toolkit (implemented in C++). The difference between the two are that a company called Riverbank Computing\(^4\) develops PyQt and PySide is an open source project that is now officially supported by the Qt Company\(^5\). Riverbank offers the PyQt library free of charge with GPL v3 license, or with a small fee, there is also a commercial license available. The problem with the GPL v3 license is that if we develop Spine Toolbox with a library that is released as a GPL license, then all derivatives of that work must be released with a GPL license as well. This is incompatible with the envisioned license for Spine Toolbox (LGPL). In the past, whenever a new version of Qt was released, Riverbank was quicker to release a new version of PyQt than the PySide project. PyQt is more mature and has some bindings to Qt that PySide does not, but in general, they are very much alike. PySide actually consists of two projects; PySide and PySide2. PySide is a project that made bindings for the older Qt4 version. The last release of PySide was made in 2015 and it provided the complete bindings for Qt4.8. PySide2 project continues the work by releasing bindings for the current Qt 5.x releases. The most recent (as of Nov. 2017) PySide2 version is 5.9 and the implementation of Spine Toolbox will start with it.

Qt is a cross-platform application development framework developed originally by Nokia but it is now owned by the Qt Company. Qt comes with a large and comprehensive set of widgets containing buttons, line edits, tables, tree views, calendars and even a web browser that supports among others HTML, CSS, and JavaScript. In addition, Qt supports creating custom widgets by subclassing existing widgets or by starting from scratch. Qt is actually more than just a GUI library. It contains all kinds of classes for application development, including XML parsing and writing, networking, database interaction, etc. These reasons, in addition with thousands of users online forming a support community, make a very compelling case to choose PySide2 for Spine Toolbox development.

To be able to fulfil all the requested features of Spine Toolbox in the allotted time, we must use other open-source Python packages in addition to PySide2. The final decisions on what packages are used will be made during implementation. It happens quite often in implementation that some packages may look promising at first, but there might be a reason to change it to another. Table 4 contains some potential packages or libraries for Spine Toolbox.

<table>
<thead>
<tr>
<th>Environment</th>
<th>Package Name</th>
<th>License</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>PyCall</td>
<td>MIT</td>
<td>Call Python functions from Julia</td>
</tr>
<tr>
<td>Python</td>
<td>PySide2</td>
<td>LGPL</td>
<td>Library of Qt bindings for Python</td>
</tr>
<tr>
<td>Python</td>
<td>PyJulia</td>
<td>MIT</td>
<td>Python interface to Julia</td>
</tr>
<tr>
<td>Python</td>
<td>GDX2py</td>
<td>MIT</td>
<td>Python package for reading and writing GAMS Data Exchange (GDX) files</td>
</tr>
<tr>
<td>Python</td>
<td>OpenPyXl</td>
<td>MIT/Expat</td>
<td>Library to read/write Excel files</td>
</tr>
<tr>
<td>Python</td>
<td>MatPlotLib</td>
<td>Uses only BSD compatible code. Based on PSF license</td>
<td>Result visualization / graph drawing</td>
</tr>
<tr>
<td>Python</td>
<td>SeaBorn</td>
<td>BSD</td>
<td>Graph plotting / Builds on top of MatPlotLib</td>
</tr>
<tr>
<td>Python</td>
<td>Ggplot</td>
<td>BSD</td>
<td>Graph plotting / Builds on</td>
</tr>
</tbody>
</table>

\(^4\) https://riverbankcomputing.com/

\(^5\) https://www.qt.io/
End-user environment is envisioned to be Windows, Linux, or Macintosh operating systems. Windows XP is not supported in Python 3.5+ anymore so Spine Toolbox will not support it. Main implementation and testing is done on Windows 7 or Windows 10 (64-bit) but the application should support Windows 8.x and Windows Vista as well. Developers and end-users with Linux, Macintosh or other operating systems are welcomed to ensure cross-platform support.

4.2.2 Coding Style

Spine Toolbox developers are expected to follow Google Python Style Guide\(^6\) with the following exceptions:

- Maximum line length is 120 characters. Longer lines are accepted for the exceptions given in the Google Python style guide.
- Google style docstrings with the title and input parameters are required for all classes, functions, and methods. For small functions or methods only the summary is necessary. No need for attributes and return values in all.

Note:
- Remember PEP8

4.2.3 Versioning, Version Control and Deployment

Spine Toolbox will be developed by dividing the progress into versions. The first working version capable of basic operations will be 0.1. Every time enough progress has been made to warrant a new version, the application will be deployed. That is, there will be an installation package released for Windows, which contains Spine Toolbox application and all of its dependencies (including Python). There are at least three Python packages that can be used for application deployment. These are PyInstaller, Cx_Freeze, and Py2exe. Installation packages for other platforms are made on demand if the developers get access to the desired platform. The chosen version control system for Spine Toolbox source codes is Git\(^7\). The initial development will happen in a Spine internal Git repository hosted by VTT. In the fall of 2018, the source codes will be released to the public by transferring them to a web-based Git version control repository hosting service. This is most likely going to be GitHub\(^8\).

Lists of main features that are required to reach a new version are given below. Developers will set up a web-based project management and issue tracking tool for tracking the progress of the implementation. One possible tool for such a task is Redmine\(^9\). These tools enable the users and developers of Spine Toolbox to report bugs and request new features and, in addition, it enables Spine members and project management to follow the progress of the implementation.

4.2.3.1 V0.1

Contains basic functionality that enables the user to pass through data from a data store, through a tool and into a result archive (maybe to a second data store).

Main features:

---

\(^6\) https://google.github.io/styleguide/pyguide.html
\(^7\) https://git-scm.com/
\(^8\) https://github.com/
\(^9\) http://www.redmine.org/
• First version of Spine Toolbox projects and a way to save & load them
• Adding data store, and a tool to project
• Present project contents to user
• First version of a data store view
• Adding a tool to project
• Execution of a (dummy) Julia tool
• Support for a rudimentary version of Spine data structure

![Diagram](image)

**Figure 8. Basic functionality in Spine Toolbox V0.1.**

### 4.2.3.2 V0.2
First version that enables the user to see the project as a data processing chain as in Figure 9.

![Screenshot](image)

**Figure 9. Data collection, tool, data store, and a view for visualizing result data.**

Main Features:

• Application should be able to draw a representation of the project as in Figure 9. (No need for the user to drag & drop blocks yet)
• Support for manipulator blocks. First versions of recipes & filters.
• Support for SQLite/MySQL (if these database technologies are chosen to store Spine data structures for projects)
• Better support for Spine data structures
• Support for GAMS models
• Connect/create data collections
• Simple Result Archive View
4.2.3.3 V0.3
First version to support dragging and dropping project building blocks into a project canvas.

Main features:
- Support for dragging, dropping and connecting building blocks on a project canvas
- First version of View blocks for output visualization
- Support for data conversion tools
- Support for other tools than Julia and GAMS (such as Windows batch scripts)

4.2.4 Security and Testing Requirements

4.2.4.1 Application Security
Spine Toolbox aims to follow the guidelines set up by the Certification Requirements for Windows Desktop Apps document\textsuperscript{10}. The document contains the technical requirements and eligibility qualifications that a desktop app must meet in order to participate in the Windows 10 Desktop App Certification Program. Successfully passing Windows App Certification allows the app to be showcased in the Windows Compatibility Center and gives a permission to display the certification logo in the app or on Spine website. While Spine Toolbox will be cross-platform, the document has sections that are general enough so that they can be applied to the development of Linux and Macintosh apps as well. Here are the technical requirements (see the document for details):

- Apps are compatible and resilient
- Apps must adhere to Windows security best practices
- Apps support Windows security features
- Apps must adhere to system restart manager messages
- Apps must support a clean, reversible installation
- Apps must digitally sign files and drivers
- Apps don’t block installation or app launch based on an operating system version check
- Apps don’t load services or drivers in safe mode
- Apps must follow User Account Control guidelines
- Apps must install to the correct folders by default
- Apps must support multi-user sessions
- Apps must support x64 versions of Windows

To validate compliance with these requirements, Microsoft provides The Windows App Certification Kit, which is one of the components included in the Windows Software Development Kit (SDK) for Windows 10.

4.2.4.2 Data Security
This section will be updated when there is more information about the type of data that Spine Toolbox will process and where the data is stored. Some data sources may be confidential. These must be secured in a way that they are not distributed to third parties.

4.2.4.3 Application Testing and Verification
Spine Toolbox developers will write unit tests for the application that will be included with the application source code. As the resources of the implementation team are limited, no 100% testing coverage is to be expected. Main components, however, will go through rigorous testing to ensure that Spine Toolbox core features remain dependable even when new features are constantly being added. Main components that must remain dependable are, for instance, database interface and storage, project saving and loading, and interface to external models. The dependability is ensured by making a

\textsuperscript{10} https://msdn.microsoft.com/en-us/library/windows/desktop/mt674655(v=vs.85
hook to code version control repository that runs all unit tests when a developer is trying to commit new code. All tests must pass before a commit to the repository is accepted.
5. SYSTEM ARCHITECTURE

This chapter presents the architecture of Spine Toolbox. First, the architectural pattern for the whole system is selected and then the individual parts of the system are designed in their own sub-chapters. Architectural patterns are templates for concrete software architectures. An architectural pattern is a general reusable solution to a commonly occurring problem. They enable developers to agree on main components and interfaces of the system and they help in dividing the implementation work into components that can be developed and tested independently. An architectural pattern is defined in [BM96] as “An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationship between them”. There are many architectural patterns to choose from and each have their advantages and disadvantages. Some of the most well-known patterns are; Layered (n-tiered), Client-server, Pipe-filter, Master-slave, Broker, Microkernel, Microservices, BlackBoard, and Model-View-Controller (MVC). Each of these patterns has their own specific application type, for which they are used over and over again. For example, layered pattern is built around a database. The layers are arranged so that data enters the top layer and works its way down each layer until it reaches the bottom, which is usually a database. Client-server pattern is a great match for web development projects and pipe-filter pattern is popular among image processing applications. If the software system that is being designed is going to be an enterprise level software, then it is common to divide the system into smaller subsystems that are all designed with their own architectural pattern.

The subsystems of a software architecture, as well as the relationships between them, can be compartmentalized into smaller architectural units. Each of these smaller units can be described by using design patterns. One definition of a design pattern is “A Design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly recurring structure of communicating components that solves a general design problem within a particular context.” [GH98]. Design patterns make it easier to reuse successful designs and architectures. They can even improve the documentation and maintenance of existing systems. Design patterns are smaller in scale than architectural patterns and they are usually independent of a particular programming language (i.e. you can design a set of subsystems before choosing the programming language). The application of a design pattern has no effect on the fundamental structure of the whole software system. There are 23 design patterns presented in [GH98]. However, that book was one of the first attempts to formalize design patterns in general and since then, hundreds of new ones have been proposed. Design patterns are solutions to problems that software developers encounter over and over again. Experienced developers have known these solutions before [GH98] was published so it really just presented the solutions also to people just starting in the profession. An example, of a problem that can be solved by exploiting a design pattern is the undo-redo capability found commonly in e.g. word processing applications. A design pattern for providing applications with this functionality is called the Command pattern.

In Spine Toolbox, users interact with the application to produce data that is visualized depending on their preferences. The Model-View-Controller (MVC) architectural pattern offers these capabilities, so we are choosing it for our high-level architecture. MVC was developed by Smalltalk-80 programmers and it was made popular in the book [BM96]. MVC inherently uses five design patterns; Factory Method, Observer, Decorator, Composite and Strategy.

Figure 10 presents the components in the MVC pattern.

- The model component encapsulates core data and functionality. The model is independent of specific output representation or input behavior. When talking about model in MVC context, model means data.
- View components display information to the user. A view obtains the data it displays from the model. There can be multiple views of the model.
- Each view has an associated controller component. Controllers receive input, usually as events that denote mouse movement, mouse clicks, or keyboard input. Events are translated to service requests, which are sent either to the model or to the view. The user interacts with the system solely via controllers [BM96].
The separation of the model (data) from the view and controller components allows multiple views of the same model. If the user changes the model via the controller of one view, all other views dependent on this data should reflect the change. To achieve this, the model notifies all views whenever its data changes. The views in turn retrieve new data from the model and update their displayed information. This functionality is provided by the Observer pattern and it is implemented explicitly in Qt (and PySide2) as the signal-slot mechanism. The basic idea of the Observer pattern can be summarized as in Figure 11, where the observer isolates the model from referencing the views directly. The primary advantage of the MVC is that it makes model classes reusable without modification. This is to keep the design simpler, more understandable, and easier to implement. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse. In some applications, the controller and the view are the same object but in this project, we aim to keep them separated. This enables us to change the view without rewriting the controller.

The next sub-chapters present the main classes of Spine Toolbox divided into model, view, and controller modules. The rest of the sub-chapters in this chapter present the designs of individual features in Spine Toolbox. The designs are early drafts that represent the initial ideas before implementation has started.
5.1 Model Classes

Figure 12 depicts the static class structure of the application’s model classes. The DataStoreModel class inherits the Qt SQL classes assuming that Spine data structures in a data store are stored in an SQL database. Results are read from the same kind of class structure than input data. The project tools are stored in a list structure in memory. The ToolModel is populated by the tools that are read from a project save file.

![Diagram of model classes](image)

Figure 12. Model classes.

5.2 View Classes

Figure 13 depicts classes that represent the applications views. MainForm class represents the view that is shown to the user when the application starts. In Qt, this class must inherit the QMainWindow class. Other views inherit the QWidget class and they are created and instantiated when the user chooses to do so from the main window.

![Diagram of view classes](image)

Figure 13. View classes.
5.3 Controller Classes

Figure 14 depicts the most important controller classes in the application. Note that the line between classes does not mean inheritance but only that which classes are aware of each other. At this point, the design is straightforward. The Main class is aware of all other controller classes and it controls everything. This is likely to evolve when new requirements and features are added to the design. ToolExecution module is designed further in the next sub-chapter.

```
Configuration

ProjectHandler  Main  ToolExecution

Scenario
```

*Figure 14. Controller classes.*

5.4 Design for Tool Execution

Figure 15 presents a class diagram for the classes involved in executing a tool in a sub-process. User initiates the execution by clicking on start execution button. The application creates an instance of a GAMSTool or a JuliaTool depending on the type of tool the user is trying to execute. The application creates an instance of a ToolInstance class, which knows the command that is to be executed and has a reference to the Sub-process instance. QProcess is the Qt class that actually runs the command in the sub-process. When the sub-process has finished, it sends a signal to the execution_finished() slot and notifies the user about this.
Figure 15. Class diagram for tool execution.
6. **SPINE DATA STRUCTURE**

Spine Toolbox needs to move and present many different kinds of data. Not all different forms of data can be known beforehand, and hence the data structure in Spine has to be able to support many kinds of data without changes to the application code. As a result, the data structure will be generic, following an entity-relationship data model [CH76]. In this data model, data is presented as data objects that can have relationships between each other. The structural features of the data can therefore be expressed using just two tables: an object table and a relationship table. From the perspective of Spine Toolbox, this is very useful as it can then handle any data with the same code.

Energy system modelling uses a lot of data as well as different kinds of data items. If all data is in two tables, this can result in cluttered data. This will be mitigated by views that filter the data into more user-friendly packages. Time series data (or any other matrix data) writ large would make the tables too large for fast access and consequently time series data will be packaged within special matrix data objects so that only their metadata is stored in the object table.

The Spine Model will be very adaptable. In order to access this adaptability when creating scenarios, the settings for the Spine Model (and any other tool that can be made compatible) will also be stored within the Spine data structure. If possible, this will be extended to the settings of Spine Toolbox project and the relationships between the entities in a Spine Toolbox project. The data structure of the Spine Model will be described in the Spine Model documentation, but it will naturally comply with this entity-relationship based Spine data structure.

The Spine data structure will need several supporting tables in addition to the object and relationship tables that hold the actual data (Figure 16). The object class defines the different types of data objects and the object parameters table defines what parameters each object class can have. Similar supporting tables are also needed for the relationships between the objects.

![Figure 16. Database tables needed to represent the Spine data structure.](image)
7. REFERENCES


