# Spine Deliverable 2.1 Software Design Document

Savolainen, Pekka T.; Kiviluoma, Juha; Rinne, Erkka; Soininen, Antti; Dillon, Joseph; Marin, Manuel

Published: 30/09/2021

*Document Version*
Publisher's final version

*License*
CC BY-SA

Link to publication

Co-funded by the European Commission within the H2020 Programme
Grant Agreement no: 774629
2017-10-01 until 2021-09-30 (48 months)

# Deliverable 2.1
# Software Design Document

| | |
|---|---|
| Revision | 1.0 |
| Submission date | 2021-30-09 (m48) |
| Due date | 2020-31-03 (m30) |
| Lead contractor | ER |

## Authors:

| | |
|---|---|
| Pekka T Savolainen | VTT |
| Juha Kiviluoma | VTT |
| Erkka Rinne | VTT |
| Antti Soininen | VTT |
| Joseph Dillon | ER |
| Manuel Marin | KTH |

| Dissemination level | | |
|---|---|---|
| PU | Public | X |
| CO | Confidential, only for members of the consortium (including the Commission Services) | |

| Deliverable administration | | | | | |
|---|---|---|---|---|---|
| **No & name** | **D2.1 Software Design Document** | | | | |
| **Status** | Final | **Due** | M30 | **Date** | 2021-09-30 |
| **Author(s)** | Pekka Savolainen VTT, Juha Kiviluoma VTT, Erkka Rinne VTT, Antti Soininen VTT, Joseph Dillon ER, Manuel Marin KTH | | | | |
| **Description of the related task and the deliverable. Extract from DoA** | **T2.1 High level system design**<br>**Task leader: ER; Participants: VTT; Duration: M01-M30**<br><br>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.<br>…<br>**D2.1: Software Design Document**<br>• First high-level version M02<br>• More specific design from tasks 2.2 - 2.7 will be used to update the document<br>• Final version M30 | | | | |

| **Planned resources PM of T2.1** | VTT | UCD | KUL | KTH | ER | | | | | Total |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | 0.0 |

| **Comments** | | | |
|---|---|---|---|
| **V** | **Date** | **Authors** | **Description** |
| 0.1 | 2017-09-12 | VTT, ER, KUL, KTH | First high-level version |
| 1.0 | 2021-09-30 | VTT, ER, KTH | Final version |

## Disclaimer

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

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 the results or output data. Spine Toolbox is designed to support the creation and execution of scenarios in optimization and simulation. In the Spine project, the main use case has been SpineOpt, which is a highly adaptable model generator for multi-energy systems. In addition, Spine Toolbox supports a wide variety of other models and tools if 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 SpineOpt because it enables them to communicate using a common vocabulary. Spine Toolbox is implemented in Python and SpineOpt in Julia.

This deliverable presents a high-level software design for Spine Toolbox and for the various tools it supports. It contains the application use cases, functional and non-functional requirements, system overview, chosen implementation language(s), dependencies, versioning, application validation requirements, testing and security requirements, and general development guidelines. The aforementioned have been collected in co-operation with Spine members and stakeholders, who have been using Spine Toolbox since its inception. The last chapter presents an overview of the Spine data structure.

The first version of this deliverable was published in project M02 and this final version presents the updated design of the software. Spine is an open-source project. In the fall of 2018, Spine Toolbox source code and documentation was released to the public. Spine Toolbox as well as the whole Spine software suite is available in a web-based version control repository system called GitHub (https://github.com/Spine-project). In addition, the user guide and other documentation is available on https://readthedocs.org/. Spine Toolbox is licensed under the GNU Lesser General Public License (LGPL). Spine Toolbox documentation, user guide and all original graphics have been released with the Creative Commons BY-SA 4.0 license. We hope to attract a lively and active community around the Spine software suite that will continue development also after the project has ended.

# 1. INTRODUCTION

Energy system models are becoming increasingly complex. Spine Toolbox is an application to help in dealing with this complexity by providing an easy-to-use graphical user interface for defining, managing, visualizing and executing energy system models. It gives users 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 provides the following features for energy system model developers:

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

Spine Toolbox has been developed as a cross-platform desktop GUI application for Windows, Macintosh, and Linux platforms. There is also a command line interface (CLI) for executing workflows. The user interface part of the application has been separated from the application data and control parts so it would be possible to also support HTML5 based interfaces but in this project, the focus has been on developing the GUI and the CLI interfaces. Spine Toolbox as well as the whole Spine software suite has been available to the public since the fall of 2018 on GitHub[1]. Installation instructions are available on Spine Toolbox landing page[2] (README.md). Spine Toolbox is licensed under the GNU Lesser General Public License (LGPL). You can find details about the license on the Free Software Foundation (FSF) website[3]. Spine Toolbox documentation, manual and all original graphics and icons are released under the Creative Commons BY-SA 4.0 license.

This deliverable presents the key features of Spine Toolbox and a high-level software design on the various components and packages it consists of. In addition, we present an overview on what third-party packages are used by the application (dependencies). In chapter 2, we present the basic use cases of the software. These were done in co-operation with Spine project members and stakeholders at the start of the project. In chapter 3, we present the original high-level application requirements, categorized to functional and non-functional requirements. During the project, the functional requirements have become more detailed, and we have used GitHub's issue tracking system[4] to document feature requests and bug reports. The high-level requirements were derived from the use cases in chapter 2 and the project plan, and they were used to generate the initial application architecture described in chapter 4.1. From chapter 4.2 onwards we describe the design of the latest version of the application. In Chapter 5, we present the dependencies (third-party packages) in the current version of the application, application testing practices and the installation options (deployment). Chapter 6 presents the milestone roadmap, which describes how the application has evolved over the course of the project and what features and improvements are envisioned for future versions. Finally, in chapter 7 we present the Spine data structure, which has been a vital part of the development process and the project. The structure has been refined to make a clear interface for Spine Toolbox developers and SpineOpt developers.

The goal in the design has always been to make it as modular, reusable, and readable as possible. The app has been distributed into four main packages where each package has its own clear-cut responsibility. This distribution helps in maintaining, optimizing, reusing, and testing the code. In addition, we believe that this solution also helps in attracting more people into the Spine community.

Table 1 contains the most important terms and their definitions used in Spine Toolbox and this deliverable.

***Table 1.*** *Definitions.*

[1] https://github.com/Spine-project/

[2] https://github.com/Spine-project/Spine-Toolbox

[3] https://www.gnu.org/licenses/

[4] https://github.com/Spine-project/Spine-Toolbox/issues

| **Term** | **Explanation** |
|---|---|
| Arc | Graph theory term. See Connection. |
| Case study | Spine project has 13 case studies that help to improve, validate and deploy various aspects of SpineOpt and Spine Toolbox. |
| Connection | An arrow on Spine Toolbox Design View that is used to connect project items to each other to form a DAG. |
| Data Package | A data container format consisting of a metadata descriptor file (datapackage.json) and resources such as data files. |
| Data sources | All original, unaltered, sources of data that are used to generate necessary input data for Spine Toolbox tools |
| Design View | A sub-window on Spine Toolbox main window, where project items and connections are visualized. |
| Direct predecessor | Immediate predecessor. E.g., in DAG x->y->z, direct predecessor of node z is node y. See also predecessor. |
| Direct successor | Immediate successor. E.g., in DAG x->y->z, direct successor of node x is node y. See also successor. |
| Directed Acyclic Graph (DAG) | Finite directed graph with no directed cycles. It consists of vertices and edges. In Spine Toolbox, we use project items as vertices and connections as edges to build a DAG that represents a data processing chain (workflow). |
| Edge | Graph theory term. See Connection. |
| GAMS | General Algebraic Modelling System. A high-level modelling system for mathematical optimization. |
| Julia | A programming language with an emphasis on high-performance. Nowadays, a popular choice for scientific computing. |
| Model | Refers to two things in this deliverable depending on the context. It refers to energy system models everywhere except in the software architecture chapter 4.1, where model refers to data in a general sense. The context should be evident. SpineOpt is an energy system model generator and is not called a 'model' in the software architecture context. |
| Node | Graph theory term. See Project item. |
| Predecessor | Graph theory term that is also used in Spine Toolbox. Refers to the preceding project items of a certain project item in a DAG. E.g., in DAG x->y->z, nodes x and y are the predecessors of node z. |
| Project | Spine Toolbox projects consist of project items and connections, which are used to build a data processing chain for solving a particular problem. Project workflows are executed using the rules of DAGs. There can be any number of project items in a project. Projects can be shared among users. |
| Project Item | Each project item in a Spine Toolbox project workflow defines an execution step depending on the type of the project item. Project items together with connections are used to build Directed Acyclic Graphs (DAG). Project items act as vertices and connections act as edges in the DAG. |
| Scenario | A scenario is a meaningful input data set for a tool or a model. |
| Solver | A software package intended for solving e.g. linear programming, mixed integer programming and other related problems. |
| Specification | A specialized instance of a Project Item. Defined by a JSON structure that contains metadata (settings) required by Spine Toolbox to execute the project item. Not all project items support specifications but for example, Tools and Importers do. |
| SpineOpt / SpineOpt.jl | An adaptable model generator for multi-energy systems. It is an interpreter, which formulates a solver-ready mixed-integer optimization problem based on the input data and the equations defined in SpineOpt. Outputs the solver results. The name SpineOpt.jl is used when there is a need to emphasize that it's written in Julia. |
| Spine data structure | 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 (see Spine database editor) while SpineOpt will be able to directly utilize as well as output them. |

| | |
|---|---|
| Spine database editor | A GUI editor in Spine Toolbox for manipulating and visualizing Spine data structures. |
| Spine Model | Spine Model was renamed to SpineOpt during Spine project |
| Spine Toolbox | A desktop application to define, manage, and execute various energy system simulation models. |
| Successor | Graph theory term that is also used in Spine Toolbox. Refers to the following project items of a certain project item in a DAG. For example, in DAG x->y->z, nodes y and z are the successors of node x. |
| Task | A piece of work to be done or undertaken by Spine Toolbox. |
| Tool | Project item that is used to execute a computational process or a simulation model. It can also be a data conversion process or a process for calculating a new variable. In general, tools refer to external tools/models that the application can execute. |
| Use case | Potential way to use Spine Toolbox. Use cases are used to test the functionality and stability of Spine Toolbox and SpineOpt under different potential circumstances. |
| Vertex | Graph theory term. See project item. |

# 2. Use Cases

This chapter describes the main requirements for Spine Toolbox as use cases. These use cases were collected at the start of the project before implementation had begun. Use cases describe critical behavior of the application. The use cases have been defined by Spine project members 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 1 depicts a high-level use case diagram of the main use cases in Spine Toolbox. It serves as a table of contents for individual use cases.

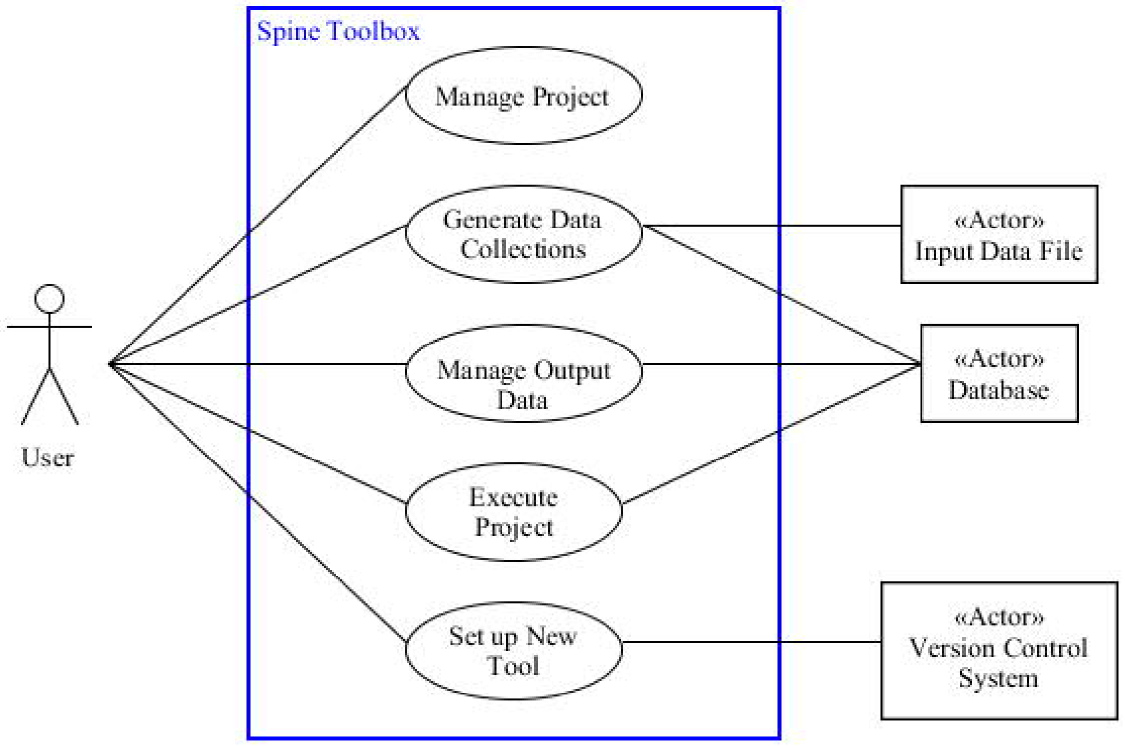


***Figure 1.*** *High-level Use Case Diagram.*

## 2.1 Manage Project Use Case

| | |
|---|---|
| Actors | User |
| Summary | User starts the application and either creates a new project or loads an existing one. |

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.

## 2.2 Generate Data Collections Use Case

| | |
|---|---|
| Actors | User, ODBC database, Input data file (e.g. text, spreadsheet, or binary file) |
| Summary | User wants to create, connect and manage data collections. |

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 (see chapter 7)

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 an example in 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 a certain type of data. Examples of sheets can be system settings, geographical information, temporal information, technology specification, policy choices, demand, renewable time series, etc. 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 2.)

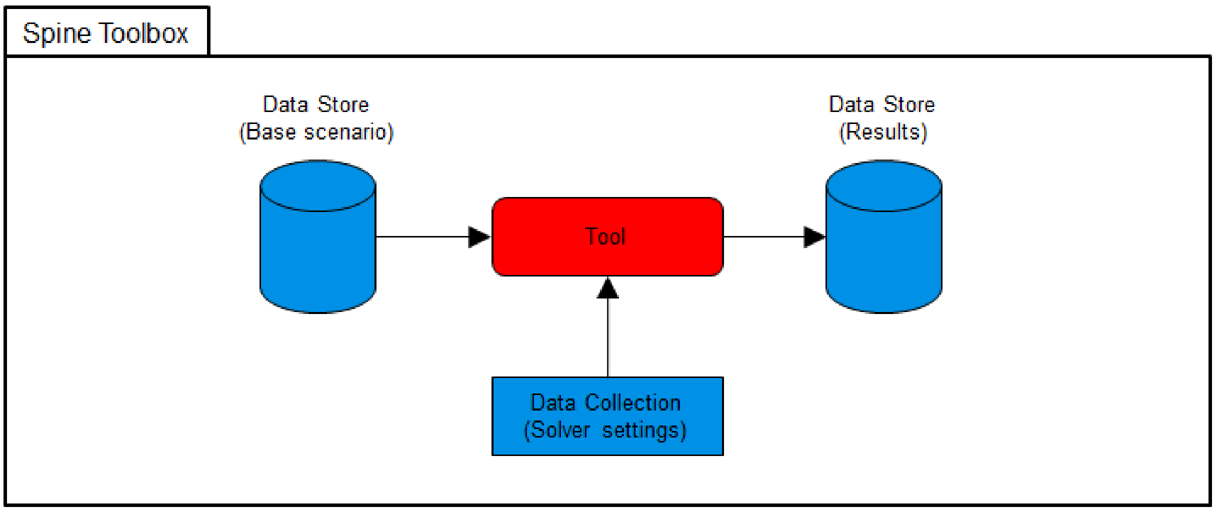


***Figure 2.*** *Two data stores, a data collection with solver settings and a processing tool.*

## 2.3 Execute Project Use Case

| | |
|---|---|
| Actors | User, Database |
| Summary | The user wants to use a computational tool to create a set of results from input data collections. |

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.)

## 2.4 Manage Output Data Use Case

| | |
|---|---|
| Actors | User, Database |
| Summary | 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. |

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. 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.’

## 2.5 Set Up New Tool Use Case

| | |
|---|---|
| Actors | User, Version control system (Git) |
| Summary | The user wants to use a new tool with Spine Toolbox. The new tool will be available in consequent sessions. |

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.

# 3. Requirements, Assumptions, Dependencies, and Constraints

This chapter presents the original high-level application requirements, categorized to functional and non-functional requirements. During the project, the functional requirements have become more detailed, and we have used GitHub's issue tracking system[5] to document feature requests and bug reports. The high-level requirements were derived from the use cases in chapter 2 and the project plan, and they were used to generate the initial application architecture. Functional requirements, in general, include data manipulation features, UI views, or other specific functionality that define what the application is supposed to accomplish.

The non-functional requirements in chapter 3.2 specify criteria that can be used to judge the operation of a system. These were collected at the start of the project, and they describe the envisioned dependencies that were considered then. Chapter 5 presents the dependencies that are used in the current version of the application. The non-functional requirements describe why we chose Python as the implementation language and why we use PySide2 as the main GUI library in Spine Toolbox.

## 3.1 Functional Requirements

Functional requirements have been categorized into UI, data management, and project execution requirements. This section forms a list of features that were requested by Spine project members. This list is not in any particular order, and it is not, by any means, complete. These are high-level requirements collected in the start of the project that were later used to make more precise feature requirements for the application. An up-to-date list of issues (feature requests and bug reports) has been kept, first in GitLab's issue tracker and later in GitHub's issue tracker when the development was moved there. The current issue tracker for Spine Toolbox can be found in https://github.com/Spine-project/Spine-Toolbox/issues. There's also a separate issue tracker for spinedb-api and spine-engine, found under their respective GitHub pages.

### 3.1.1 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,

[5] https://github.com/Spine-project/Spine-Toolbox/issues

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 solved by calculating the value field based on the defined values for the symbols.

### 3.1.2 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:

- o 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:

- o The user wants to create several similar scenarios

View data within a scenario:

- o Users need a view into scenario data

Tool code can be manipulated within Spine Toolbox:

- o 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:

- o Find the best way to implement views on the data store and data collections and decide the database format (MySQL, SQLite?)
- o 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

### 3.1.3 Project Execution Requirements

- Pass through data:
  - o 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:
  - o 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:
  - o 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.
  - o Users can shut down their own computer and be informed when the execution has finished.
- Allocation of computational resources:
  - o 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

## 3.2 Non-functional Requirements

### 3.2.1 Implementation Language and Dependencies

Spine Toolbox will be implemented in Python[6]. 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[7]. 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.

There are several 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 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 substantial amounts of data that Spine Toolbox must be able to handle. PyGObject is a library of bindings to GLib/GObject/GIO/GTK+ (implemented mostly in C/C++). It is mostly used for GNOME application development and as such is not a suitable 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 is still not ready for Python 3. Both, PyQt and PySide are libraries of bindings for the

[6] https://www.python.org/

[7] https://www.tiobe.com/tiobe-index/

Qt toolkit (implemented in C++). The difference between the two is that a company called Riverbank Computing[8] develops PyQt and PySide is an open-source project that is now officially supported by the Qt Company[9]. 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 an updated version of Qt was released, Riverbank was quicker to release an updated 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 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. For version v0.6.5 we require PySide2 5.14.

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 2 contains some potential packages or libraries for Spine Toolbox.

***Table 2.*** *Potential dependencies for Spine Toolbox.*

| Environment | Package Name | License | Notes |
|---|---|---|---|
| Julia | PyCall | MIT | Call Python functions from Julia |
| Python | PySide2 | LGPL | Library of Qt bindings for Python |
| Python | PyJulia | MIT | Python interface to Julia |
| Python | GDX2py | MIT | Python package for reading and writing GAMS Data Exchange (GDX) files |
| Python | OpenPyXl | MIT/Expat | Library to read/write Excel files |
| Python | MatPlotlib | Uses only BSD compatible code. Based on PSF license | Result visualization / graph drawing |
| Python | SeaBorn | BSD | Graph plotting / Builds on top of MatPlotLib |
| Python | Ggplot | BSD | Graph plotting / Builds on top of MatPlotLib |
| Python | Bokeh | BSD | Graph plotting |
| Python | Pygal | LGPL | Used for creating SVG charts |
| Python | QtDataVisualization | LGPL | Qt Data Visualization library. Part of PySide2 project. |

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 have been done on Windows 10 (64-bit) and Linux. Developers and end-users with Linux, Macintosh or other operating systems are welcomed to ensure cross-platform support.

### 3.2.2 Development guidelines

[8] https://riverbankcomputing.com/

[9] https://www.qt.io/

Development of Spine Toolbox has followed the guidelines set up by the Certification Requirements for Windows Desktop Apps document[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. Even though Spine Toolbox is a cross-platform application, the document has sections that are general enough so that the guidelines can be applied to development of Spine Toolbox on Linux and Macintosh as well. Here are the basic guidelines the developers of Spine Toolbox have followed throughout the development process:

- 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 do not block installation or app launch based on an operating system version check
- Apps do not 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.

### 3.2.3 Coding Style

Spine Toolbox developers are expected to follow Google Python Style Guide[11] 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 at all.

See D7.7 and the Contribution Guide for Spine Toolbox in User Guide for a complete guideline.

### 3.2.4 Application Testing and Verification

Spine Toolbox developers are expected to write unit tests for the application that will be included to the same repository as 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 robust 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 hook to code version control repository that runs all unit tests when a developer commits new code. This way, it is easy to keep track of which commit introduced problems.

[10] https://msdn.microsoft.com/en-us/library/windows/desktop/mt674655(v=vs.85

[11] https://google.github.io/styleguide/pyguide.html

# 4. 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 utilizes problem independent data and user interface structures. A problem independent structure is able to support many kinds of modelling problems, which is important, as the types of modelling problems are likely to change significantly over time. This means that Spine Toolbox could also be used for other kind of modelling than energy systems. Even though the structure is problem independent, Spine still aims to make it easy and intuitive for the user to define the problem to be solved.

The main concept behind Spine Toolbox is the idea of automating and composing different models for the integration of energy vectors through the well-known paradigm of computational workflow. There is an isomorphic equivalence between a composite model and a DAG (Directed Acyclic Graph) computational workflow where each node has four main elements: input from the previous node, an output to the successor, some internal operations (workflow step) and eventual access to external data sources. A computational node needs to receive the input from the predecessor node or an empty value from the root and during the execution interact with the external data source, both in reading and writing mode. At the end of the execution the node will push the output data to the successor node. The composition of various nodes will result in a computational workflow equivalent to a DAG. The software architecture of Spine Toolbox follows the workflow control architectural pattern, using a tight integration [6] between Spine Toolbox and Spine Engine.

In graph theory, a directed acyclic graph is a directed graph with no directed cycles. That is, it consists of vertices and edges, with each edge directed from one vertex to another. In Spine Toolbox, we let the user create a workflow that follows the rules of DAGs by giving them a set of project items (vertices) and directed arrows (edges), which the user can use to create a workflow. Workflows can be saved as a Spine Toolbox project that contains all necessary input data to run selected external model(s) to produce results. An external model is an external program that the application can 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. SpineOpt is one of the tools that is supported in Spine Toolbox.

Key Features

- Project based
- GUI and CLI interface
- Spine database editor
- Embedded Python Console
- Embedded Julia Console
- Python support
- Julia support
- GAMS support
- Executable and shell command support
- Data plotting functionality
- Data Import / Export in selected formats (e.g., CSV, JSON, Excel)
- Plugin support
- Parallel execution
- etc.

## 4.1 Approach to architecture design

This section describes the architectural pattern for the whole Spine Toolbox system. Architectural patterns are templates for concrete software architectures. An architectural pattern is a reusable solution to a commonly occurring problem. They enable developers to agree on the 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 [2] 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; each with 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 repeatedly. For example, the 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 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.” [4]. 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 [4]. 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 repeatedly. An example of a problem that can be solved by exploiting a design pattern is the undo-redo capability available in most modern applications today. 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 this concept, so this was a natural fit for our high-level architecture. MVC was developed by Smalltalk-80 programmers, and it was made popular in the book [2]. MVC inherently uses five design patterns: Factory Method, Observer, Decorator, Composite and Strategy.

Figure 3 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. **Note: In the context of MVCs, model means generic application specific 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 [2].

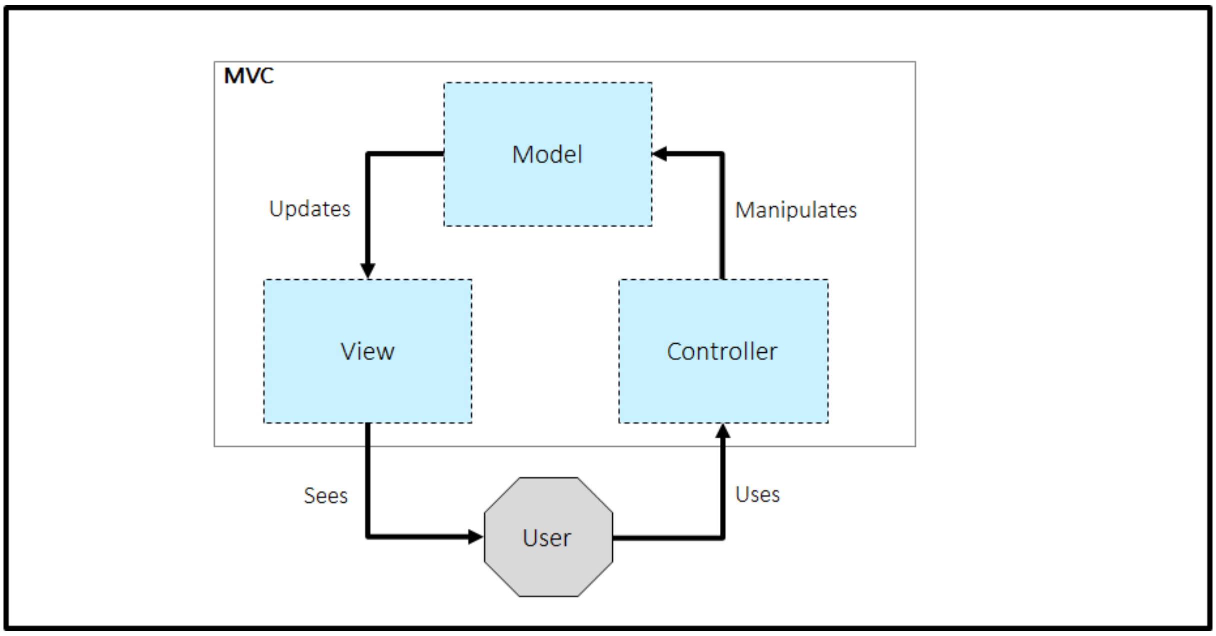


***Figure 3.*** *MVC architectural pattern.*

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 4, 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 developing Spine Toolbox, the objective has been to keep them separated. This enables us to change the view without rewriting the controller.

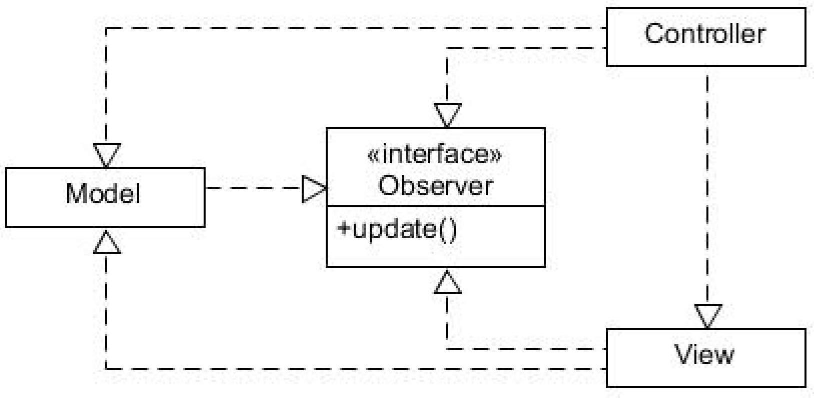


***Figure 4.*** *MVC with an Observer interface.*

## 4.2 Package organization and design

Spine Toolbox has been distributed into four packages, each with their own responsibilities and testing infrastructure. Spine Toolbox main package (spinetoolbox) contains the main GUI functionality and provides most of the things users see and interact with when using the application. It also contains the code for managing Spine Toolbox projects (creating, saving, loading) and workflows within projects.

One important convention in Spine is the Spine data structure, which is an entity-relationship-value 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 SpineOpt because it enables them to communicate using a common vocabulary. Package spinedb-api was designed for this purpose, it provides an interface and the functions to create, edit, and manage data in Spine data structure format. Spinedb-api is independent of other packages, it can be installed and used from the command line without Spine Toolbox if users so choose. The job of spinetoolbox is to provide a graphical user interface for managing databases using the API provided by spinedb-api.

Package spine-engine provides the execution functionalities of project items. The interaction between spinetoolbox and spine-engine has been designed in a way that the user commences workflow execution using spinetoolbox but the actual execution of scripts and programs in the workflow is done using methods and classes provided by spine-engine. One of the reasons for the separation of execution and workflow management was done to provide support for remote execution on a computational cluster. In addition to the GUI provided by spinetoolbox, there is a CLI (command line interface) available for executing Spine Toolbox projects. As the execution happens via spine-engine, it makes the CLI faster, easier to maintain, optimize and test. Spine-engine is independent of spinetoolbox but it does require spinedb-api. It can be used independently for executing DAGs provided that the input parameters adhere to spine-engine interface requirements.

Users create workflows (DAGs) in Spine Toolbox projects using project items. The project items are provided by a package called spine-items. This package requires spinetoolbox and is not meant to be used independently. Spinetoolbox (the package) does not require spine-items but it is not particularly useful without project items. Each project item in spine-items is separated into its own directory (package), furthermore, each project item has been split in two parts: static items and executable items. Static items provide the GUI elements such as icons and widgets for user interaction in Spine Toolbox. Executable items do not contain any GUI elements. They are created dynamically at run-time based on user selections when the user wants to execute a DAG. These executable items are then given to spine-engine for execution. Separating the project items from Spine Toolbox was done for the following reasons; it makes adding, maintaining, and testing project items easier, and it provides a way for third parties to swap the entire project item kit (provided by Spine project) to another kit.

In general, the four-package design of Spine Toolbox is there to increase modularity, maintenance, documentation, and testing of the application. In addition, this is an open-source project so this design should help in attracting a wider audience and user base. One concern in software development has, for some time now, been future proofing the application, With the current design, it is possible to switch the GUI parts of Spine Toolbox and modify it from a desktop application into a web application.

### 4.2.1 spinetoolbox package

The spinetoolbox package is a Python application that provides a graphical user interface to manage projects and data as well as to execute DAGs on a Spine Engine instance. Additionally, it provides base classes and utilities for project items in the spine-items package. The application can also be invoked from the command line to execute the DAGs of a given project without opening the GUI. The package depends on spinedb-api and spine-engine and a plethora of other third-party packages, including PySide2 which provides the Qt bindings needed for the GUI.

The main window of spinetoolbox is where users create, view and modify the DAGs of a project. It also contains Julia and Python consoles as well as different logs for application and engine messages. The main window consists of dockable subwindows which facilitates flexible placement of GUI elements.

Spinetoolbox has a large number of methods and classes for integrating project items including the ones in spine-items into the GUI. Most notable ones are the base classes and methods in the project_item subpackage and project_item_icon module which define the programming interface between toolbox and an item. Additionally, one of the docks in the main window is reserved for project items to draw their properties on. Items that are specification enabled can use the SpecificationEditorWindowBase class for their specification editors for consistent look and feel.

A considerable part of spinetoolbox code base is dedicated to Spine Db editor, a GUI that can be used to view and modify Spine databases. It is usually accessed via the Data Store project item. The editor provides various views of the database including pivot tables and a graph representation of entities and their relationships. A tabbed interface allows multiple databases to be open in the same window while a single tab can contain data from more than one database for comparison. The latter feature is specifically accessed from the View project item. DB editor uses spinedb-api as its backend but implements its own multithreaded data fetching and caching subsystem called DB manager on top of that. The goal of DB manager is to keep the DB editor as responsive as possible even with large datasets.

When a user decides to execute a project, spinetoolbox serializes the needed parts of the DAGs of currently open project and sends the data to Spine Engine. Then spinetoolbox listens to the status messages from the engine and updates the logs and views accordingly. All communication with the engine takes place in a separate thread to keep the GUI responsive. Julia and Python tools can be set up to use Jupyter consoles which are accessible in spinetoolbox as well enabling post-execution inspection and debugging.

Spinetoolbox allows installation and management of plugins. Plugins can be installed from a curated registry that is provided by the Spine project.

### 4.2.2 spinedb-api package

The spinedb-api package takes care of all communication with Spine databases. The main purpose of the package is to provide a single entry-point for database communication that all clients can use regardless of the underlying SQL engine. This approach avoids code duplication, ensures database integrity, and simplifies the creation of scripts that automate database operations. The package relies on sqlalchemy as main dependency to interact with SQL at a low level. The package is itself a dependency for the spinetoolbox, spine-engine, and spine-items packages.

Specifically, spinedb-api provides a class called DatabaseMapping that clients can instantiate using the URL of a Spine DB. The class implements methods to select, insert, update, and delete DB elements, as well as to check integrity both for insert and update operations. For example, to insert object classes into the database, the client would call DatabaseMapping.add_object_classes() with an iterable of Python dictionaries, where each dictionary specifies the fields of an object_class item. To make the changes permanent, the client would call DatabaseMapping.commit_session() with a meaningful commit message.

The package also provides a class called DiffDatabaseMapping that has the same interface as DatabaseMapping, but immediately commits any changes into temporary tables. Whenever the client calls DiffDatabaseMapping.commit_session(), the changes are ‘moved’ into the original tables. This technique thus enables concurrent editing over an arbitrary period of time. The DiffDatabaseMapping class is used by spinetoolbox to implement the Spine Db Editor.

Both DatabaseMapping and DiffDatabaseMapping can create a 'fresh’ Spine DB at the given URL or upgrade the Db schema to the latest revision. This behaviour is controlled by two Boolean keyword arguments in the class constructor. Database migration is implemented using the alembic package.

The spinedb-api package also implements Spine’s JSON specification for parameter values. For each parameter value type, namely ‘date_time’, ‘duration’, ‘array’, ‘time_pattern’, ‘time_series’, and ‘map’, there is a corresponding Python class with a rich API to access and edit the value (including internal indexes and values, whenever applies). Two free functions, from_database() and to_database() are provided to convert parameter values between their database representation in bytes, and the corresponding rich Python class.

The package also provides an API to specify database filters and manipulators that modify select-queries ‘on-the-fly'. The implementation relies on a set of custom free functions that receive a URL together with a filter or manipulator specification and return a modified URL with the filter or manipulator information embedded in the ‘query’ segment. The caller can then use the modified URL to instantiate DatabaseMapping or DiffDatabaseMapping, and all select-queries to the DB are subsequently altered in place according to the specified filter or manipulator. At the moment, clients can use this API to specify scenario or tool filters, class and parameter definition renaming schemes, and basic mathematical operations on parameter values.

The spinedb-api packages also provides functionality to define so-called ‘mappings’ to import and export database contents from or to a tabular data format. For each database element (object_class, alternative, parameter_definition, etc.) there is two corresponding Python classes to specify either an import-mapping or an export-mapping. These classes are instantiated with an integer that represents a row or column within a table. The resulting mapping instance can then be passed together with the specification of a data source/destination, to a function that carries out the corresponding import/export operation. The data source/destination can be anything ‘tabular’, and currently CSV, Excel, datapackage, SQL, and Gdx interfaces are supported.

Finally, spinedb-api also provides functions to create a socket server with a minimal API to access and write database contents. The purpose is to support non-Python applications that can have nonetheless access to a low-level socket API, such as the SpineInterface.jl package in Julia. The server URL is also understood by DatabaseMapping and DiffDatabaseMapping constructors, so that Python applications can also use it to instantiate those classes directly instead of using the server.

### 4.2.3 spine-engine package

The spine-engine package controls the execution of Directed Acyclic Graphs (DAGs), a.k.a, workflows, from Spine Toolbox projects. Such DAGs are composed of one or more executable project items each of which may have a particular specification, as well as connections from one item to another indicating execution priority and direction of data flow. In an upcoming version, there can also be conditional backward jumps which create loops in the DAG. Each DAG execution consists of two sweeps, one backwards, where items collect resources from their direct successors, and one forwards, where items collect resources from their direct predecessors and execute. In case there are loops, the jump conditions are evaluated during the forward sweep after the source item of the jump has executed. If the condition evaluates to true, the sweep is continued from the jump destination item. The package relies on dagster in order to use their powerful workflow orchestration machinery. The package is also a dependency for the spinetoolbox and spine-items packages.

Specifically, the spine-engine package defines the interface for executable project items and item specifications. Clients can implement these interfaces in order to define their own project items and specifications, such as Data Connection, Data Store, Importer, Exporter, and Tool, currently provided by spine-items. All these interfaces include a method to serialize the corresponding instance into a dictionary that can then be dumped into JSON.

Coupled with the above, the package also provides the SpineEngine class that can be instantiated using a workflow definition, that is basically a list of serialized executable project items and specifications, as well as a dictionary indicating connections. Internally, SpineEngine creates dagster solids from the different project items and two dagster pipelines for the backwards and forwards executions. Clients can then call SpineEngine.get_event() in a loop to retrieve events resulting from the execution of the associated workflow.

Each project item is executed in its own thread. Typically, a project item would spawn an asynchronous subprocess to do any heavy computations and block until the subprocess returns, thus allowing other threads to progress in the meantime (which ultimately results in parallel workflow execution). However, this is not a hard requirement and must be implemented by each executable project item, as is the case, e.g., for the Tool, Importer, and Exporter project items in spine-items. The spine-engine package only provides helper classes and functions to assist in this task.

Connections between project items support specifying scenario and/or tool filters that are applied on the resources that go through that connection during forward execution. Each connection may have an arbitrary number of filters which essentially is treated as multiple connections, one for each filter. This results in the workflow being split into multiple branches, until a DB writing operation makes it converge again into a single branch. Project items in the split zone are virtually replicated and each of the replicas is executed in its own thread, which as per the above enables parallel execution of the branches.

In an upcoming release, spine-engine may also be executed as a web server. Particularly, spine-engine listens to requests sent by spine_toolbox for DAG execution. The service API includes the following information:

- JSON-message including data (configuration) to be provided to the spine-engine for DAG execution
- ZIP-file containing all files of the project folder

The service API has been implemented with ZeroMQ messaging protocol. The JSON message and ZIP-file are transferred between the toolbox and spine_server with a multi-part message.

Currently, spine-engine extracts the received ZIP-file contents to the local storage, executes the DAG based on the attached JSON-message, and returns data received from spine_server back to the toolbox.

### 4.2.4 spine-items package

Project items are plugins that can be loaded into the spinetoolbox application and spine-engine. The standard set of project items is provided by the spine-items package. It includes a set of general purpose project items which are useful in most projects. The package depends on spinetoolbox, spine-engine, spinedb-api and other third party packages.

Each project item comes as a self-contained subpackage. Shared code is placed into shared modules in the spine-items package. The plugin loaders in spinetoolbox and spine-engine expect certain structure from these subpackages and therefore they must include certain modules and classes under specific names.

In a broad view, each project item consists of two parts: one is responsible for providing GUIs in spinetoolbox for setting up and managing the item while the other, the executable item, is instantiated within Spine Engine during execution. The GUI parts use methods and classes provided by spinetoolbox and are generally tightly coupled with it. The executable parts, on the other hand, do not depend on spinetoolbox or some of its heavier dependencies such as PySide2 making it possible to install spine-engine alongside spine-items on a remote machine without pulling spinetoolbox or its dependencies into the mixture.

## 4.3 Projects

Spine Toolbox is project based - the user creates a separate project for each purpose. A Spine Toolbox project contains one or more workflows (DAGs), that are created by connecting project items with provided links. Projects are used to save users work so they don’t need to restart their work every time the application is started. Spine Toolbox provides a Design View, a sort of canvas, where the user can drag-and-drop project items needed for solving a particular problem. Project items are connected with links that appear on Design View as arrows. The arrows are directed, which enables users to choose the order in which project items are executed. Project either contains the data or has references to the data that the projects items require. Users can choose the data files for solving a particular problem by choosing them anywhere on the file system or from a network drive. Databases in a project can either be SQLite files or server-based databases like MySQL. After solving a problem, the project allows users to choose where to save the result files. One advantage of the project-based approach is, that users can work on different problems in separate projects. After executing a workflow, Spine Toolbox archives the produced results. This enables users to bring back existing results and determine which input data and which model version produced which results. They can also revisit old projects to reproduce results for verification or proving purposes.

Each Spine Toolbox project employs a directory on users file system. Directory name is the (default) project name. This directory is used in saving the workflow, results, and specific project settings that can be configured in the app. Every time a new project item is added to a project, a new directory dedicated for the new project item is also added to the there. These are called data directories and even though all of them are not in use at the moment, they may come in handy in the future. This one-directory-one-project approach yields several benefits; it enables users to browse project data and results easily and it enables users to work on a same project in co-operation with a colleague by using a version control system (VCS) such as Git.

## 4.4 Project Items

The building blocks for creating a workflow are project items. Project items are connected to each other either with a one-to-one relationship or a one-to-many relationship. The description of project items and their icon on Design View in Spine Toolbox are shown in Table 3. Each item has its own properties, which can be viewed and edited in the Properties widget (available when selecting an item in Design View). Project item properties, its position on Design View and connections to other project items are saved when the project is saved.  See also, the User Guide for up-to-date information on project items.

***Table 3.*** *Project items in Spine Toolbox v0.6.5*

| | |
|---|---|
| Data Store | Data Store item represents a connection to a database, where the data follows the Spine data structure format. Currently, the item supports SQLite and MySQL dialects. The database can be accessed and modified in the Spine database editor, which is available e.g., by double-clicking the icon on Design View. 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. |
| Data Connection | Data connection provides access to (raw) data files. Typically, these are spreadsheets, csv files or databases but there is no restriction on the type of file that can be used here. The item supports two types of files: *references* are files anywhere on the file system while *data files* reside in the item's own data directory. Data Connections are useful as a place for raw data files. |
| Importer | Importer enables the user to define a mapping from tabulated data to Spine data structure format. We support CSV, Excel, GDX (GAMS Data eXchange), JSON, Data Package (Frictionless Data), and SQL (any supported by SQLAlchemy) as input data. When executed, it transforms the data coming from a Data Connection, Data Store, Exporter, or a Tool before passing it onwards. The heart of Importer is the Import Editor window in which the mappings from source data to Spine database entities are set up. The editor window can be accessed from the Importer's Properties widget. Importer mappings are saved as a specification (see section 4.4.1) that can be reused by other Importers or even other projects. |
| Exporter | Exporter outputs database data into tabulated file formats. Supported output file types now include CSV, Excel, GDX, and SQL. The output files can be passed onward in the workflow, or they can be analysed in an external software. At its heart, Exporter maps database items such as entity classes or entity names into an output table. Exporter saves its export mappings as a specification (see section 4.4.1) that can be reused by other exporters or even other projects. The export mappings can be viewed and edited in the Exporter Specification Editor. |
| Tool | Tool is the heart of a DAG. It is usually the actual model to be executed in Spine Toolbox, but it can be an arbitrary script or an executable as well. A Tool is specified by its specification (see section 4.4.1). In a typical use case, Tools process data that is passed to them in order to produce an output file, which is either archived into the results directory or passed onwards in the DAG for further processing. We now support four types of Tools: GAMS, Python, Julia, and executable. SpineOpt can be used in a Spine Toolbox DAG as a Julia Tool. An example on the versatility of an executable Tool type is that it can be used to incorporate Java programs or models into a Spine |

| | |
|---|---|
| | Toolbox DAG. Tool specifications are edited using the Tool Specification Editor, which can be accessed e.g., from the main Toolbar. |
| Data Transformer | Data transformers set up database manipulators for successor items in a DAG. They do not transform data themselves; rather, Spine Database API does the transformations configured by Data transformers when the database is accessed. Currently supported transformations include entity class and parameter renaming. |
| Gimlet | While being able to run most scripts and copyable executables, Tools cannot handle system commands or executables meant to run from system's PATH environment variable. This is a job for Gimlet. A Gimlet can execute an arbitrary system command with given command line arguments, input files and work directory. User can select the shell (also no-shell execution available), where the given command is executed. Supported shells on Windows are CMD and Powershell, while Bash is supported on Linux based systems. |
| View | A View item is meant for inspecting data from multiple Data Stores. It enables opening multiple databases in a single Spine database editor tab. Note that the data is opened in read-only mode, so modifications are not possible from the View item. |

### 4.4.1 Passing data between project items

Project items share data by files or via databases. One item writes a file which is then read by another item. Project item *resources* (instances of ProjectItemResource class) are used to communicate the URLs of these files and databases.

Both static items (parts of project items that are used by the Toolbox GUI) and their executable counterparts pass resources. The major difference is that static items may pass resource promises such as files that are generated during the execution. The full path to the promised files or even their final names may not be known until the items are executed.

During execution resources are propagated only to item's direct predecessors and successors. Static items offer their resources to direct successors only. Resources that are communicated to successor items are output files that the successor items can use as input. Currently, the only resource that is propagated to predecessor items is database URLs by Data Store project items. As Data Stores leave the responsibility of writing to the database to other items, it must tell these items where to write their output data.

The table below lists the resources each project item type produces during execution.

***Table 4.*** *Resources provided by project items*

| **Item** | **Provides to predecessor** | **Provides to successor** |
|---|---|---|
| Data Store[12] | Database URL | Database URL |
| Data Connection[13] | n/a | File URLs |
| Importer | n/a | n/a |
| Exporter | n/a | File URLs |
| Tool[14] | n/a | File URLs |

[12] Data Store provides a database URL to direct successors and predecessors. Note that this is the only project item that provides resources to its predecessors

[13] Data Connection provides paths to local files

[14] Tool's output files are specified in the Tool specification

| | | |
|---|---|---|
| Data Transformer[15] | n/a | Database URL |
| Gimlet | n/a | Resources from predecessor |
| View | n/a | n/a |

The table below lists the resources that might be consumed by each project item type during execution.

***Table 5.*** *Resources consumed by project items*

| **Item** | **Accepts from predecessor** | **Accepts from successor** |
|---|---|---|
| Data Store | n/a | n/a |
| Data Connection | n/a | n/a |
| Importer[16] | File URLs | Database URL |
| Exporter | Database URL | n/a |
| Tool[17] | File URLs, database URLs | Database URLs |
| Data Transformer | Database URL | n/a |
| Gimlet[18] | File URLs, database URL | Database URLs |
| View | Database URLs | n/a |

### 4.4.2 Specifications

Specification is a Spine Toolbox concept that has proven to be useful for achieving the requirements that we want to achieve with the app. It is a specialized instance of a project item, defined by a JSON structure that contains metadata (settings) required by Spine Toolbox to execute the project item. Presently Tools, Importers, Exporters, and Data Transformers support specifications. Specifications can be reused in other project items or even other projects. They also enable developing and sharing new project items as plugins. Each project item that supports specifications can have just one specification attached at a time, but it can be switched easily to another specification from a drop-down menu in project item Properties. Specifications are saved by default to the project directory.

Tool specifications

To execute a Julia, Python, GAMS, or an executable program in Spine Toolbox, users must first create a Tool specification for a project. Tool specification JSON contain the name, type, main program file and additional program files, required and optional input files and output files needed to run the program. Only name, type, and main program file fields are required. The other fields are optional. Users can create new Tool specifications or edit existing ones with the Tool Specification Editor, available in Spine Toolbox.

Importer specifications

Importer specifications contain a mapping from (source) input data to data in Spine structure format. In a typical use case, the source data is provided by a file in a Data Connection preceding the Importer. In addition, Importer requires a database URL from its successor for writing the mapped data. Creating and editing Importer specifications is done with the Import specification editor in which the mappings from source data to Spine database entities are set up. Supported source data file types are CSV, Excel, GDX (GAMS Data eXchange), JSON, Data Package (Frictionless Data), and SQL (any supported by SQLAlchemy).

Exporter specifications

As the name implies, Exporters do the opposite of what Importers do. They can be used to export Spine database data into tabulated file formats. Exporter specifications contain the mapping from database

[15] Data Transformer provides its predecessors' database URLs modified by transformation configuration embedded in the URL

[16] Importer requires a database URL from its successor for writing the mapped data. This can be provided by a Data Store

[17] Tool specification specifies tool's optional and required input files. Database URLs can be passed to the tool program via command line arguments but are otherwise ignored.

[18] Gimlet's resources can be passed to the command as command line arguments but are otherwise ignored

items such as entity classes or entity names into an output table. Each item has a user given output position on the table, for example a column number. By default, data is mapped to columns, but it is also possible to create pivot tables. Supported output file types include CSV, Excel, GDX, and SQL. The output files can be passed to successor project items in the DAG or they can be analysed in an external software.

Data Transformer specifications

Data transformer specifications set up database manipulators for successor items in a DAG. They do not transform data themselves; rather, Spine Database API does the transformations configured by Data transformers when the database is accessed. The specification is a filter command that is prepended to the database URL. Currently supported transformations include entity class and parameter renaming.

### 4.4.3 Plugins

Spine is an open-source project so users always have a chance of either contributing to a package developed in Spine, or they can fork a package and modify it however they wish. In addition to these options, Spine Toolbox supports customization via plugins by third-party developers. In general, plugins are software components that add a specific feature to an existing application. We have implemented a framework that supports two types of plugins. We hope that this framework helps in attracting a wider audience for Spine Toolbox by helping users modify the application to be useful in a wide variety of use cases.

Project item plugins

It is possible to add a completely new project item by either adding one to spine-items package or by developing a whole new set of project items. Third-party developers can find the instructions on how to get started in the Developers documentation section in the User Guide. This option offers the most possibilities for extending the application but is also a lot of work. We encourage users to make sure that what they want to achieve cannot be done with the existing features or by specifications before implementing a completely new project item.

Specification plugins

There are two repositories on Spine GitHub page related to plugins: PluginRegistry[19] and SpineOptPlugin[20]. The former is for exposing plugin locations to Spine Toolbox and the latter is an example plugin that contains two Tool specifications. In Spine Toolbox main menu, an item called Plugins opens a widget that lets users select a Plugin registry. By default, the official PluginRegistry is always available for users, but users may create their own Plugin registries if needed. PluginRegistry is a very simple repo. It only contains a single JSON file, which exposes URLs of other Git repos to Spine Toolbox. SpineOptPlugin contains two Tool specification and some example Julia code on how to get started with SpineOpt using Spine Toolbox. When a user loads the SpineOptPlugin to Spine Toolbox by using the Plugins main menu item, he will see two new icons in a special tool bar. These icons (load_template and run_spineopt) can be dragged to Design View to create new project items into a project that are ready to be executed. SpineOptPlugin repo also acts as an example on how specifications can be shared using Git. All users need to do is to create a specification they want to share, upload it to Git, update PluginRegistry to expose this new plugin repo and then the new plugin is available to anyone via the Plugins widget in Spine Toolbox.

## 4.5 Requirements for integrating external tools

Spine Toolbox can be used with a wide variety of external tools. Such tools can be e.g. GAMS models, Julia programs, Python programs, or other software. Julia or Python programs may also require a specific environment that they need for functioning properly. For Julia programs, Spine Toolbox supports Julia projects and for Python programs it supports, pure Python environments, virtual environments and

[19] https://github.com/Spine-project/PluginRegistry

[20] https://github.com/Spine-project/SpineOptPlugin

Conda environments. To run an external tool in a Spine Toolbox DAG, one must either make a Tool specification with a suitable type or use the Gimlet project item to run a shell command that calls the external tool.

### 4.5.1 GAMS support

Running GAMS programs requires an installation of GAMS on the users' system. Note that the bitness (32 or 64bit) must match the bitness of the Python interpreter. In Spine Toolbox, users must set GAMS discoverable in Spine Toolbox Settings (File ->Settings). GAMS models can be added to a workflow by creating a GAMS type Tool specification.

### 4.5.2 Julia support

Running Julia programs requires an installation of Julia on the users' system. Spine Toolbox supports Julia versions from 0.6 to the latest. In Spine Toolbox, users must set Julia discoverable in Spine Toolbox Settings (File ->Settings). If the Julia program requires a certain Julia project (environment), you can select this in Spine Toolbox Settings (File->Settings). Julia programs can be added to a workflow by creating a Julia type Tool specification. After creating one and adding it to a project, users can create a Julia Sysimage for this Tool from the Tool properties if needed.

### 4.5.3 Python support

As Spine Toolbox is written in Python, users are ready to run Python programs in Spine Toolbox DAGs as soon as the app is launched. Spine Toolbox source code runs in Python 3.7 or Python 3.8, but the app supports running Python programs written in Python 2.7 to Python 3.10. Python programs can be added to a workflow by creating a Python type Tool specification. The environment for a Python script can be defined simply by selecting a Python interpreter, which can be an interpreter in a pure Python environment or an interpreter in a virtual environment. Spine Toolbox supports Python Jupyter kernel specs, which are used to execute a Python program in the embedded Jupyter Console and to modify an environment as the user chooses. In addition, if a Python program requires a Conda environment (e.g. Calliope[21]), this environment can be selected for the Python program. In this case, the selected Python program is run in Jupyter Console where the selected Conda environment has been activated. Users can select an interpreter or an environment for each Python program (Tool specification) separately.

### 4.5.4 Executable support

The fourth tool specification type is called an executable and it basically allows the user to execute any executable file on their system. The most common ones on Windows are files with extensions .exe or .bat. There are no set up steps for running an executable program, one just needs to create an executable type Tool specification.

#### 4.5.4.1 Required and optional settings

All four (GAMS, Julia, Python, and Executable) Tool specification types must set the following settings in the Tool Specification Editor:

- Main program file. The file where the program entry point resides.
- Additional program files. Other files, the program needs to function (optional if executing in source directory, see chapter 4.6.1)

Optional settings for Tool specifications:

- Command line arguments
- Input files. Data files the program needs to function.
- Output files. Setting the output files to the specification makes them discoverable by successor project items in the DAG. In addition, these files will be archived after each execution.

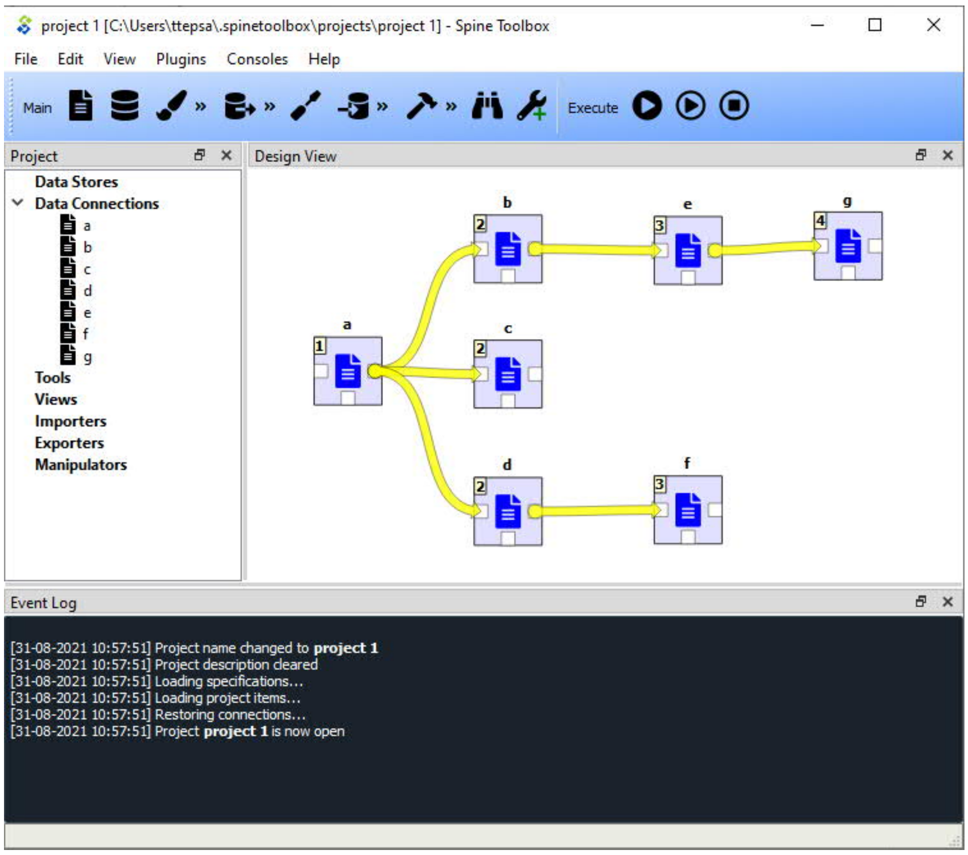
[21] https://www.callio.pe/

### 4.5.5 Shell command support

Users can add a shell command into a Spine Toolbox DAG by using the Gimlet project item. In Gimlet properties, one can enter the command to run, give command line arguments and select the shell to use (cmd.exe or powershell.exe for Windows, bash for others). In a future version, this functionality will be available as part of an executable Tool specification. At that point, Gimlet project items will become obsolete.

## 4.6 Executing DAGs

Spine Toolbox uses Breadth-First Search (BFS) algorithm for traversing DAGs in a project. BFS explores a directed graph layer by layer. The starting node *a* forms layer 0. The direct successors of *a* form layer 1. In general, all nodes that are successors of a node in layer *i* but not successors of nodes in layers *0* to *i−1* form layer *i+1*. Instead of saying that node *v* belongs to layer *i*, we also say that *v* has depth *i* or distance *i* from *a*. Picture below depicts an example DAG (workflow) created in Spine Toolbox. The DAG has been created using only Data Connections, but the main idea is the same no matter what project items are in a DAG. Node *a* is the starting node and also forms layer 0. Layer 1 consists of nodes *b*, c, and *d*. Layer 2 consists of nodes *e* and *f*. And finally layer 3 consists of node *g*. The traversal algorithm is implemented in *dagster*, which is a data orchestrator package equipped with a pipeline execution framework.



```
[31-08-2021 10:57:51] Project name changed to project 1
[31-08-2021 10:57:51] Project description cleared
[31-08-2021 10:57:51] Loading specifications...
[31-08-2021 10:57:51] Loading project items...
[31-08-2021 10:57:51] Restoring connections...
[31-08-2021 10:57:51] Project project 1 is now open
```

***Figure 5.*** *Project in Spine Toolbox v0.6.5 containing one DAG*

Classes and functions in spine-engine do the actual execution in Spine Toolbox. The execution order is determined by an algorithm implemented in *dagster*, which is a requirement of spine-engine. In the figure above, the number on the upper-left corner of project item icons shows the execution order of items. Because the BFS algorithm proceeds layer-by-layer, project items having the same number can be executed in parallel. When execution starts (user has clicked the ‘Play’ button in the Tool bar), we create an instance of spine engine worker (SpineEngineWorker) class on Spine Toolbox side on another thread. The purpose of the SpineEngineWorker class is to separate the execution thread from the main thread, which must be done to prevent the GUI from freezing until execution has finished. SpineEngineWorker is also responsible for catching logging messages from spine-engine and delivering them to Spine Toolbox. Logging messages from spine-engine are shown in Spine Toolbox widgets: Event Log and Item Execution Log. In addition, SpineEngineWorker enables us to update the GUI depending on the execution status while execution is still running. For example, we update the project item icon status on Design View to finished (a green tick if it was successful, a red cross if the execution failed) as soon as the executable project item has finished execution on spine-engine side. If there are more than one DAGs in the project, we instantiate a new SpineEngineWorker for each DAG. This means that each DAG runs in its own thread. SpineEngineWorker objects instantiate the main class in spine-engine, which is called SpineEngine. The most important input arguments for SpineEngine are

- items: Executable project items serialized into Python dictionaries
- specifications: Project item specifications (dictionary) used by project items
- connections: Dictionary containing information on which project items are connected to each other
- settings: Spine Toolbox execution settings
- project_dir: Path to project directory
- execution_permits: Enables selected project item execution. Users also have the option of executing only a selected part of a DAG.

As mentioned previously, SpineEngine class creates two pipelines from the DAG. During the forward pipeline execution, each project item is executed in its own thread. Typically, project items spawn an asynchronous subprocess, which blocks until the process has finished. For Tools, there are three options on how they are executed. These options are implemented on spine-engine side in classes ProcessExecutionManager, PersistentExecutionManager and KernelExecutionManager. ProcessExecutionManager provides the capability to spawn a subprocess that will block until execution has finished. GAMS and executable Tool specifications are executed using ProcessExecutionManager. For Julia and Python Tool specifications, there are two separate consoles embedded into the application to choose from. These are called Basic Console and Jupyter Console.

Execution part of the Basic Console is implemented on spine-engine side in PersistentExecutionManager class. The GUI part of the console is on spinetoolbox side in PersistentConsoleWidget class. When user selects to run a Julia Tool specification in the Basic Console, we start a Julia REPL in the Basic Console before execution and run the main script file there. If user has selected to run a Python Tool specification in the Basic Console, we start a Python REPL in the Basic Console and execute the main script file there.

Execution part of the Jupyter Console is implemented in KernelExecutionManager on spine-engine side. The GUI part of the console is on spinetoolbox side in JupyterConsoleWidget class. The Jupyter Console in Spine Toolbox is, in effect, the Jupyter QtConsole[22] application developed by the Jupyter[23] project, modified for our purposes and embedded into Spine Toolbox. Just like the original QtConsole, Jupyter Console in Spine Toolbox requires Jupyter kernels to work. Jupyter kernels interact with Jupyter Console with kernel specs (defined by a kernel.json file), which contain the programming language, the interpreter/compiler, and any additional arguments or settings for the kernel. Users can create kernel specs manually or use the Kernel Spec Editor widget found in Spine Toolbox. The programming languages that Jupyter Console now supports are Python and Julia but other programming languages (e.g. R) supported by Jupyter should also work in Spine Toolbox with only minor modifications.

A new (Basic or Jupyter) console is launched for each Python and Julia Tool specification in the workflow. This choice has the benefit of being able to run Tool specs in parallel but the downside is that memory usage may become a problem when a DAG is big.

In an upcoming version, there will be support for looping, which enables executing a certain part of a workflow repeatedly, until a user defined condition is met. Even though DAGs, as per definition, do not admit looping (they are acyclic), this feature is needed so frequently that we decided to build support for it anyway.

[22] https://qtconsole.readthedocs.io/en/stable/

[23] https://jupyter.org/

In executing Data Stores, Importers, and Exporters we use a class called ReturningProcess. It implements a class that allows Spine Toolbox to leverage multiple processors when executing these items. The process spawned by this class offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads.

Data Connection, Data Transformer, and View project items are not executed in the same way as the previous ones. As there is nothing to process or compute related to these items, they simply pass resources to successor items when execution proceeds to these items in the DAG.

### 4.6.1 Work directory vs. source directory execution

You can either execute a Tool in a work directory or in the source directory. Executing in a work directory means that for each Tool execution, a new directory is created. Before execution, the input data and the script or model files are copied to a work directory and the execution happens in that directory. After execution, the work directory contains the input data, the model, and the result files. When a user makes changes to a model and executes the Tool again, a new work directory is created where the input data and the modified model are copied to. This helps the user in keeping track of what results were created by which model, and with which input data. An optional execution mode is the source directory execution. In this case a work directory is not created but the model execution happens in the directory where the main program file resides. This may be a better option when the model or the input files are big because copying these files to work directory may take a significant amount of time. Users can select whether to execute a Tool in work or in source directory from the Tool specification editor or from Tool properties.

## 4.7 Scenarios

*Scenarios* are formed in Spine data stores by choosing 1 or more alternatives in particular order. Alternative is a data structure in Spine data store that allows to hold multiple values for the same parameter of the same entity. When Scenario data is read from the DB, later alternatives replace values from the earlier alternatives. Scenarios can be passed to the downstream tools using a scenario filter in the arrow connecting the data store and the tool. Execution of scenarios can take place in parallel.

*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 scenarios. 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 6 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 x 1 x 3 x 2 x 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 6.*** *Tasks in a table*

| **Level name** | **Tasks** | | |
|---|---|---|---|
| Scenario data | Low VG prices | High VG prices | |
| Investment model | Balmorel | | |
| Time series | 2010 | 2011 | 2012 |
| Operational model | JMMwHist | JMMwStoSSch | |
| Operational | JMMpp | | |

# 5. Dependencies, Testing and Deployment

## 5.1 Dependencies

Spine Toolbox requires Python 3.7 or Python 3.8. The following tables present the dependencies of each package and their license. Version number requirements and limitations are available in *requirements.txt* or *setup.py* files shipped with spinetoolbox, spine-engine, spine-items, and spinedb-api packages.

PySide2 is a package that provides Python bindings to Qt library, which is implemented in C++. Qt is a cross-platform application development framework and 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 more than just a GUI library. It contains all kinds of classes for application development, including XML parsing and writing, networking, database interaction, etc. The main dependency of spinetoolbox and spine-items is PySide2, which has been used to build all graphical components seen on screen. In addition, it has been extensively used in implementing the internal data (i.e. the Model in MVC) used in the application.

***Table 7.*** *The most important dependencies in spinetoolbox v0.6.5*

| Package name | License |
|---|---|
| spinedb-api | LGPL |
| spine-engine | LGPL |
| spine-items* | LGPL |
| pyside2 | LGPL |
| jupyter-client | BSD |
| qtconsole | BSD |
| sqlalchemy | MIT |
| numpy | BSD |
| matplotlib | BSD |
| scipy | BSD |
| networkx | BSD |
| pandas | BSD |
| pymysql | MIT |

* spine-items is not a ‘hard’ requirement of Spine Toolbox. The app does start without spine-items but the features in that case are quite limited.

***Table 8.*** *The most important dependencies in spinedb-api v0.14.0*

| Package name | License |
|---|---|
| sqlalchemy | MIT |
| alembic | MIT |
| faker | MIT |
| datapackage | MIT |
| numpy | BSD |
| openpyxl | MIT/Expat |
| gdx2py | MIT |

| | |
|---|---|
| ijson | BSD |

***Table 9.*** *The most important dependencies in spine-engine v0.12.0*

| Package name | License |
|---|---|
| spinedb-api | LGPL |
| dagster | Apache-2.0 |
| networkx | BSD |
| datapackage | MIT |
| jupyter-client | BSD |

In addition, we have embedded and modified a couple of classes into spine-engine from nb_conda_kernels[24] package.

***Table 10.*** *The most important dependencies in spine-items v0.9.0*

| Package name | License |
|---|---|
| spinetoolbox | LGPL |
| spinedb-api | LGPL |
| spine-engine | LGPL |
| pyside2 | LGPL |
| pyodbc | MIT |
| sqlalchemy | MIT |
| pygments | BSD |
| numpy | BSD |

## 5.2 Testing

We are using Python's unittest module as the backbone of our testing infrastructure. Each package has their own unit tests. In addition, spinetoolbox contains an additional test suite called execution tests. Executing a DAG uses code in spinetoolbox, spine-engine, spine-items, and possible from spinedb-api as well. A standard unit test only tests code in a single package so execution is not tested thoroughly by just unit tests. The execution test suite is meant for this exact purpose, testing the complete execution stack. We have automated running the tests, the tests are run automatically after every commit to the repository. We also use a code coverage tool for automatically calculating the test coverage after each commit.

## 5.3 Deployment

There are three options for installing Spine Toolbox:

1. Standard Python installation (PyPi)

Spine Toolbox is available[25] for installation on PyPi (Python Package Index), which is the de-facto package repository for Python. Users can install Spine Toolbox simply by using pip (*pip install spinetoolbox*)

2. Windows 64-bit Installer Package

[24] https://github.com/Anaconda-Platform/nb_conda_kernels

[25] https://pypi.org/project/spinetoolbox/

Windows installer packages are published periodically but not as frequently as the standard Python installation above. This option is suitable for users who cannot install Python or don't need to get the most recent updates. Download the latest installer package[26], run it, and follow the instructions to install Spine Toolbox. The (Python) code for the installer packages has been built using a Python package called Cx_Freeze. The first installer packages, containing the installation wizard, were done using Cx_Freeze as well. Nowadays, we use Cx_Freeze to build the code and then another Tool called Inno Setup[27] to make a professional looking installer package.

3. Installation from sources using Git

The development version is available on the 'master' branch of Spine Toolbox GitHub repository. This is always the latest version but it is not as robust as release versions. Note that the release versions are also available as separate branches in the repository.

See also deliverable 7.8 for the development roadmap after Spine project.

[26] https://github.com/Spine-project/Spine-Toolbox/releases

[27] https://jrsoftware.org/isinfo.php

# 6. MILESTONE ROADMAP

We have used semantic versioning[28] for Spine packages. In a nutshell, this means that the version number consists of MAJOR.MINOR.PATCH numbers. We have incremented, the:

- MAJOR version when we make incompatible API changes,
- MINOR version when we add functionality in a backwards compatible manner, and
- PATCH version when we make backwards compatible bug fixes.

Additional labels such as 'beta' and 'final' have been appended to the version number to distinguish pre-release versions from final release versions. The first working version capable of basic operations was 0.1. Every time a new minor version has been reached, there has been a single-file installation package for Windows, which contains Spine Toolbox application and all of its dependencies (including Python). In addition, the releases have been and will be archived into a Git branch. You can find the release branches by searching for branches named 'release-x', where x is the release version number. The following chapter presents the most important changes and advancements in each version. See CHANGELOG.md for more details about the changes.

## Version 0.1

Release date: 20.8.2018

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:

- Basic functionality
- Data Store, Tool, Data Connection project items
- Adding project items to project
- First version of projects and a way to save & load them
- Execution of GAMS and Julia Tools
- First version of Data Store Tree View Widget
- Support for a rudimentary version of Spine data structure
- Spinedb-api package
- User Guide
- Unit test framework

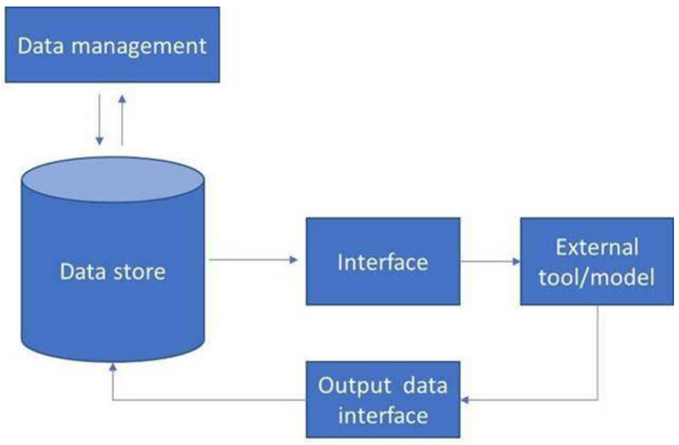


***Figure 6.*** *Basic functionality in Spine Toolbox V0.1.*

[28] https://semver.org/

## Version 0.2

Release date: 17.1.2019

This version supported connecting project items into a Directed Acyclic Graph

Main features:

- Tree View and Graph View for Data Stores
- View project item
- Support for executing executables
- Exporting datapackages to Spine data structure
- Embedded Julia Console
- Support for SQLite/MySQL
- Better support for Spine data structures

## Version 0.3

Release date: 6.9.2019

Execution improvements, support for Python Tools, Introduction of spine-engine package

Main features:

- Support for executing Python tool specifications
- Embedded Python Console
- Execution of selected project items
- Stop execution
- Importer project item
- New Spine databases can be created in any backend supported by spinedb-api.
- Executing Directed Acyclic Graphs instead of just Tools.
- Tabular view in Data Store

## Version 0.4

Release date: 3.4.2020

Main requirements:

Support for executing DAGs on spine-engine using dagster. Visual improvements and lots of refactoring.

Main features:

- Spine Toolbox as a Python package
- Exporter project item
- Tool configuration assistant for SpineOpt
- Merged tree, graph, and tabular views into one consolidated view.
- Support for undo/redo user actions
- Saving projects to any directory

## Version 0.5

Release date: 14.12.2020

Main requirements:

General improvements

Main features:

- Gimlet project item
- Better support for Jupyter kernels
- Kernel spec editor
- Automatic project upgrade framework
- Plugin support

## Version 0.6

Release date: 7.5.2021

Main requirements:

Parallel execution of projects with multiple DAGs on Spine Engine. Separation of project items from spinetoolbox by introducing the spine-items package.

Main features:

- Support for parallel/multicore processing.
- Dropped support of Python 3.6 and add support Python 3.8
- Data Transformer project item
- Closing projects
- Install Julia automatically from Spine Toolbox

## Version 0.7

Release date: ?

Main requirements:

Execute projects remotely by running spine-engine on a server.

Main features:

- Recipes (make combinations of alternatives easily)
- Support for looping in DAGs,
- Run spine-engine on a server
- Metadata support

## Version 1.0

Release date: ?

Main requirements:

Settle the API of spinetoolbox package

# 7. Spine Data Structure

Spine Toolbox needs to move and present many kinds of data. Not all different forms of data can be known beforehand, and hence the data structure in Spine must be able to support many kinds of data without changes to the application code. As a result, the data structure is generic, following an entity-relationship data model [3]. In this data model, data is presented as data objects that can have relationships with 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 especially useful as it can then handle any data with the same code.

Energy system modelling uses a lot of data as well as various 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) 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.

For example, SpineOpt is a very adaptable energy system model. To access this adaptability when creating scenarios, the settings for the SpineOpt (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 SpineOpt is described in the SpineOpt documentation, but it will naturally comply with this entity-relationship based Spine data structure.

The Spine data structure requires a web of tables to represent many diverse kinds of data structures (Figure 7). The object and relationship classes define categories for objects and relationships while the object parameters table defines what parameters each class can have. Both objects and relationships are wrapped inside a common ‘entity’ table as they share many characteristics like parameters and their values. The metadata in the figure is currently implemented with less detail. Single metadata can be linked with multiple entities and parameter values.

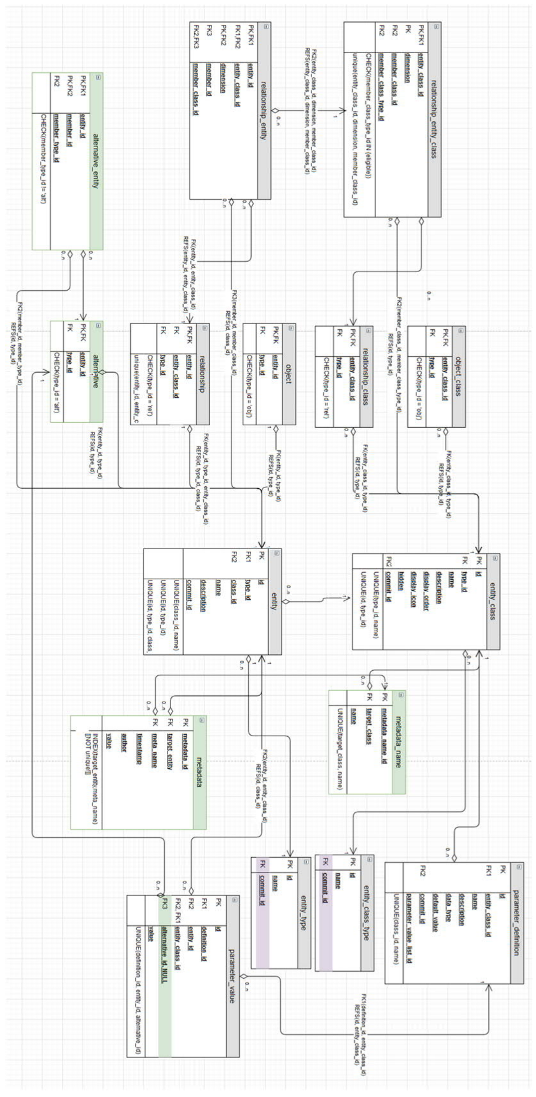


***Figure 7.*** *Database tables needed to represent the Spine data structure.*

# 8. REFERENCES

[1] Sinan Si Alhir. “UML in a Nutshell”. ISBN: 1-56592-448-7. O’Reilly & Associates, 1998.

[2] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. “Pattern-Oriented Software Architecture - Volume 1: A System of Patterns”. ISBN 13: 978-0-471-95869-7. John Wiley & Sons, 1996.

[3] Peter Chen. "The Entity-Relationship Model - Toward a Unified View of Data". *ACM Transactions on Database Systems.* **1** (1): 9–36, March 1976.

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. “Design Patterns - Elements of Reusable Object-Oriented Software”. ISBN: 0-201-63361-2. Addison-Wesley, 1998.

[5] Robert Hanmer. “Pattern-Oriented Software Architecture for Dummies”. ISBN: 978-1-119-96399-8. John Wiley & Sons, 1998.

[6] Software architectures to integrate workflow engines in science gate- ways, Future Generation Computer Systems 75 (2017) 239 – 255. doi:https://doi.org/10.1016/j.future.2017.01.005. URL http://www.sciencedirect.com/science/article/pii/ S0167739X17300249