**Time Series Analysis Support for Data Scientists  
Software Design Specification**

E. Petersen (ezekielp) - 1-23-2021 – v1.1

**Table of Contents**

1. SDS Revision History 1

2. System Overview 2

3. Software Architecture 2

4. Software Modules 3

4.1. <Module Name> (Include one subsection for each module.) 3

5. Dynamic Models of Operational Scenarios (Use Cases) 5

6. References 5

7. Acknowledgements 6

# 1. SDS Revision History

This lists every modification to the document. Entries are ordered chronologically.

**Date Author Description**

**————— ——— ————————————————————————————**

1-22-2021 ezekielp Added initial document to repo, will modify template for project

# 1-23-2021 ezekielp Added first draft of preprocessing section to module section

1-30-2021 ezekielp Added first draft of System Overview, updated 4.1, added 4.2

# 2. System Overview

The Transformation Tree discussed here is a tool for Data Scientists for evaluating and executing Time Series pipelines. Consisting of a library of functions for manipulating such n-ary trees, the software is built around nodes representing common steps used by Data Scientists when processing Time Series data. These steps are primarily broken down into preprocessing/IO, modeling, and visualization/evaluation modules, with each representing one or more secondary classes of nodes to further assist organization and type checking within the tree.

The software allows a Data Scientist to create a tree, add processing steps in the form of nodes, replicate entire trees, subtrees, or pipelines, and save and load pipelines or trees. It also allows for the replacement of one step with another of the same class and ultimately “execute” a tree or pipeline.

# 3. Software Architecture

(The term *module* is used here to describe a separable piece of software that is a part of the entire computer system, such as a "student records" module. The term *component* is used somewhat interchangeably.)

***Software Architecture***

The description of the software architecture should capture and communicate the important design decisions regarding how the system is decomposed into parts, and the relationships among those parts (Faulk, 2017). The architecture should describe:

1. The set of components. This should be both in easy-to-read list form, and also in a diagram.

2. The functionality provided by (or assigned to) each component. This should be included in the list of components.

3. Which modules interact with which other modules. Describe how the components work together to achieve the overall system functionality. This should be indicated in the architectural diagram, and also described in brief paragraph form after the list of components.

This should be at a level of abstraction that you would use to explain to a colleague how the system works. It should be abstract and static, such as "The client database holds all of the client information and interacts with the data-cleaning module to ensure that no sensitive data gets released through the online system...." It should not be a detailed textual description of a dynamic flow of control such as "module A passes the record to module B which removes the user ID and then passes the record to module C...."

4. The rationale for the architectural design. This should be in paragraph or list form. Explain how and why this architecture was decided to be the best to solve the problem. See “Design Rationale” in the next section.

Do not keep your architecture simple just to reduce the number of modules you need to describe later in the SDS. If your architecture has only one module, your project may be too small for the purposes of an SDS, or too small for a class project, or there may be a problem with your design.

***Use descriptive names for components.***

Every module should have a name that is specific to the project. Do not use generic names such as "User Interface", "Model", "View", "Controller", "Database", "Back end", or "Front end". Instead, use names specific to the functionality of this system such as "Instructor Interface", "Student Interface", "Roster", "Student Records", "Roster View", "Grade View", and so on. Every module name should in some way convey the module's role *in this project*, not the role in a generic software design.

Do not name modules "client" or "server". The roles of client and server are relative to a particular service. A component can be a server with respect to one service and client with respect to another. (Faulk, Young)

# 4. Software Modules

## 4.1. Preprocessing

***4.1.1. Role and primary function.***

The preprocessing module is a library of functions that will be used as operators in the first layers of a Transformation Tree. Most functions will take raw TS data in the form of a DataFrame object (with perhaps other inputs) and return a modified version in the same format. The module allows for several preprocessing steps to be taken in a row once inserted into a tree.

This module also contains the data splitting functionality (into training, validation, and test sets) used prior to passing data to the modeling nodes.

The preprocessing module also contains the simple functions to read and write to a .csv file either to or from a DataFrame object. These functions will be instrumental as the very first and last steps in a pipeline.

***4.1.2. Interface Specification***

The following functions will be made available as public operators. Individual function descriptions will be included as a comment in the appropriate file:

- read\_from\_file(input\_file\_name) 🡨 May not be directly accessible, but handled behind the scenes

- write\_to\_file(output\_file\_name) 🡨 May not be directly accessible, but handled behind the scenes

- denoise(ts)

- impute\_missing\_data(ts)

- impute\_outliers(ts)

- longest\_continuous\_run(ts)

- clip(ts, starting\_date, final\_date)

- assign\_time(ts, start, increment)

- difference(ts)

- scaling(ts)

- standardize(ts)

- logarithm(ts)

- cubic\_root(ts)

- split\_data(ts, perc\_training, perc\_valid, perc\_test). <-

- design\_matrix(ts, input\_index, output\_index) <-

- design\_matrix(ts, mi, ti, mo, to) <-

- ts2db(…)

NOTE: Some of these functions may end up missing this list, as some are required in all cases and will be included when appropriate at a particular stage of the tree (for example the read/write from file and data splitting functions). <- means not directly called/used as a node operator and thus may not truly be part of the interface.

***4.1.3. A Static and Dynamic Model***

TODO: Ask about the diagrams for *each* module (seems like a few overall diagrams would be sufficient)

***4.1.4. Design rationale***

The preprocessing module was designed in order to factor out the logic behind the execution of the pipeline. We want all of the standard functionality already used by Data Scientists available, but we wish to treat them as operators of the nodes of the Tree. The library design allows us to swap in and out the various functions and leave the Tree organization to its own module. This was a fairly easy design decision to make with no other obvious alternatives as the functions included were required to be present in the final product and already separated into its own “module” in the project description. We could have placed all required functions, including those from the Modeling and Visualization/Evaluation into one large module, but it made more sense to split up according to function for later separation when designing the node structure within the tree.

***4.1.5. Alternative designs***

There were discussions of breaking up the module further into three parts with the intention of separating the IO and data splitting aspects of the preprocessing function set, however, since both of these pieces were fairly small and their functionality seemed to fall into the “preprocessing” archetype, it was deemed unnecessary.

## 4.2. Modeling

***The module’s role and primary function.***

The Modeling module makes up the middle layers of the Transformation Tree and is the primary logic for the system. It takes in pre-split data from nodes of the preprocessing module and creates forecasts based on that data. This is later passed to the last nodes in the tree for comparison if the tree is being tested, or will output the forecasts as a final product if the “tree” is in production.

***Interface Specification***

The following functions will be made available as public operators. Individual function descriptions will be included as a comment in the appropriate file:

- mlp\_model(input\_dimension, output\_dimension [, layers])

- mlp.fit(x\_train, y\_train) <-

- mlp.forecast(x) <-

- rf\_model()

- rf.fit(x\_train, y\_train) <-

- rf.forecast(x) <-

NOTE: <- means not directly called/used as a node operator and thus may not be truly part of the interface, rather it would be a part of the execution.

***A Static and Dynamic Model***

TODO: Ask about the diagrams for *each* module (seems like a few overall diagrams would be sufficient)

***Design rationale***

Like the preprocessing module, the modeling module was designed in order to factor out more of the logic behind the execution of the pipeline. We want all of the standard functionality already used by Data Scientists available, especially the many models with which to do analysis, but we wish to treat them as operators of the nodes of the Tree. The library design allows us to swap in and out the various models and leave the Tree organization to its own module.

***Alternative designs***

There were discussions about potentially adding more ML models as operators to the modeling nodes, however, due to time constraints, we kept the limited number of required models. It is our hope that due to the similarity of input data for ML modeling techniques, future maintenance and improvements on the program would be a relatively straightforward process of adding to the “backend” library of functions.

## 4.2. <Module Name> (Include one subsection for each module.)

Each module should be described with:

a. The module’s role and primary function.

b. The interface to other modules.

c. A static model.

d. A dynamic model.

e. A design rationale.

***The module’s role and primary function.***

Every module needs to be described first abstractly in terms of its function or role in the system, and then in more detail such as in terms of its data structures and functionality. When describing such details, lists and diagrams will probably provide be more readable and searchable (for the eyes) than paragraphs of text.

Throughout the document, lists and sublists of design specifications should be indented and numbered to make it easy to read and reference the specification details. Such as:

1. *Module Name*

1.1. *Design Specification Detail*

1.1.1 *Design Specification Sub-Detail*

Note how this permits reference to "SDS Item 1.1.1".

***Interface Specification***

A software interface specification describes precisely how one part of a program interacts with another. (Faulk, Young) This is not the user interface, but instead a description of services such as public methods in a class, or getters and setters. Describe the software interface that each module will make available to other software components, both internal and external. This will help to explain how components will interact with each other, what services each module will make available, how to access a module, and what needs to be implemented within a module.

Find the right abstraction for each interface specification. For example, describe the services that will be provided but not how they will be implemented within the module. An interface that reveals too much information is *over-specified* and limits freedom of implementation. (Faulk and Young, 2011)

***A Static and Dynamic Model***

Every software module should be described with a static or dynamic model, and probably both.

Each diagram should use a specific design *language*. There are enough languages that have been developed such that you will not likely need to devise your own. Most diagram languages emphasize a static or dynamic representation, and do not typically mix the two. For example, class diagrams are primarily static, though they hint at time-based dynamic activities in their method names. Sequence diagrams emphasize activities over time, but also list the static entities (such as the classes) that are interacting. But you do not typically see a dynamic activity indicated on an association between two classes in a class diagram.

Every diagram should be introduced with a caption that appears immediately below the diagram and should what is show in in that diagram. Each caption should starts with “Figure <x>.” and be referenced in the body of the text as “Figure <x>.” The caption should briefly describe what the diagram shows, such as “Figure 3. A sequence diagram showing the ‘Feed a child’ use-case.”

Diagrams can be hand-drawn and scanned in. There are some advantages to hand-drawn diagrams such as they are in some ways easier to modify. But they should ideally be scanned-in rather than photographed by hand, in part to keep the file sizes down. Do not fill an SDS with large (>1MB each) high-resolution photos of hand-drawn diagrams.

***Design rationale***

There will be a reason that each module (and the architecture) is designed as it is. The “design rationale may take the form of commentary, made throughout the decision process and associated with collections of design elements. Design rationale may include, but is not limited to: design issues raised and addressed in response to design concerns; design options considered; trade-offs evaluated; decisions made; criteria used to guide design decisions; and arguments and justifications made to reach decisions.” (IEEE Std 1016-2009) Your design should find a good separation of responsibility for each module. One design rationale required by the IEEE standard (1016-2009) is “a description of why the element exists, ... to provide the rationale for the creation of the element.”

***Alternative designs***

Your SDS for this class should include alternate designs that were considered for the architecture, and for each system or subsystem. This should result from either (a) considering multiple alternative designs considered during the project or (b) your design evolving over the course of the project.

These alternative design ideas and diagrams can be placed in a separate section entitled “Alternative Designs” or “Earlier Designs”, or could be placed immediately after each current design, and clearly marked as an alternative or earlier design.

This would not be part of a typical SDS but is included here to reinforce the idea that design is a process.

If you do not consider alternatives, you are not doing design. If you are not recording the alternatives that you consider, you are not engaged in a good design practice.

Describe alternative architectural decisions that were considered. (If there were no alternatives, then no “design” work was done.)

# 5. Dynamic Models of Operational Scenarios (Use Cases)

Every major Use Case should be described with a dynamic model such as a UML sequence diagram.

# 6. References

This section lists the sources cited in the creation of this template document. An SRS should reference all of the sources that it draws from. If sufficient citations are provided “in line” (at the point of reference) in the document, this section may not be necessary.

Faulk, Stuart. (2011-2017). CIS 422 Document Template. Downloaded from https://uocis.assembla.com/spaces/cis-f17-template/wiki in 2018. It appears as if some of the material in this document was written by Michal Young.

IEEE Std 1016-2009. (2009). IEEE Standard for Information Technology—Systems Design— Software Design Descriptions. <https://ieeexplore.ieee.org/document/5167255>

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM, 15*(12), 1053-1058.

# 7. Acknowledgements

Acknowledge any sources used in your document and your project.

This template builds slightly on a similar document produced by Stuart Faulk in 2017, and heavily on the publications cited within the document, such as IEEE Std 1016-2009.