Technological Feasibility Analysis

November 5, 2018

Team Amadeus

Mentor: Austin Sanders

Sponsors: Dr. Hélène Coullon & Frédéric Loulergue

Members: Wyatt Evans, Kyle Krueger, Melody Pressley, Evan Russell

Table of Contents

Introduction **2**

Technological Challenges **3**

Technological Analysis **3**

Reactive Interface Design and Prototyping 3

Generation of Code 9

Facilitation of Plugins 11

Simulation of MAD Assembly Deployment 15

Technological Integration **19**

Conclusion **20**

1. Introduction

Deploying large, complex pieces of software can be a difficult matter. There have been numerous solutions developed to make this process easier, but none are perfect. Many are hard to use, or lack the performance to make their use worth it in the first place.

Our project sponsor, Dr. Hélène Coullon, is a researcher with the STACK team, a part of Inria, the French national research institute on computer science. Their work has produced Madeus: a theoretical model for software deployment. Madeus defines the deployment process in parts via a well-defined mathematical syntax, and a corresponding Petri net-inspired diagram. The model also expresses every dependency between different software components. This enables software deployment to be done concurrently, with different components executing deployment independently until a dependency is required.

MAD (the Madeus Application Deployer), also an Inria project, is a Python application of the Madeus model; its goal is to allow users to deploy software according to the model. It gives an explicit syntax to Madeus by defining all aspects of it within Python modules. However, this representation of Madeus can be difficult to use. Although MAD is good at describing software deployment in the words of Madeus, users still have to deal with the difficulties of defining their software via a programming language. This can be tedious and prone to errors, at no fault of Madeus. The team at Inria wants the Madeus model to be more accessible to anyone interested in it. That's where we, Team Amadeus, come in: with the help of Dr. Coullon (and co-sponsor/NAU professor Frédéric Loulergue) we are to give Madeus a frontend that makes it easier to use.

Our solution is a GUI that enables users to utilize the Madeus model via the "Petri net-inspired diagram[s]" described above. This GUI will help increase usability by allowing what would usually be done manually in Python code to be done graphically instead. By providing clear visualizations, allowing drag-and-drop assembly building, and MAD code generation, we hope to give users easier access to the Madeus model.

This document serves as a feasibility analysis for our development of the GUI. Its goal is to make explicit the major technological challenges we foresee in this project, as well as our proposed solutions. We will begin by discussing the sources of the problems in Section 2. Then, in Section 3, there will be breakdowns for each problem featuring solutions, and a proof-of-feasibility discussing how we can test our solution to confirm we made the correct decision. Finally, in Section 4 we address how we plan to integrate all of our solutions together into one cohesive system.

2. Technological Challenges

For this project, technological challenges may originate from various sources. From a broad scope, we need to build our GUI using tools that will promote ease-of-use, while keeping the software's code elegant and modular so that it can be easily extended later in its life. The software must also be accessible, not only for end-users to operate, but also in terms of what it can run on - it should easily be run on Windows, Mac, and Linux.

At this stage in the requirements engineering of the project, we have isolated four major technological challenges related to our system:

- Reactive Interface Design and Prototyping
- Generation of Executable Python Code
- Facilitation of Plugins
- Simulation of MAD Assembly Deployments

All of these hurdles are related to the primary requirements of our GUI, and are necessary for us to overcome in order to complete this project.

3. Technological Analysis

Given these four technical challenges, we must research ways to address them. This section will address all four challenges individually: it will provide a deeper overview on each, give metrics to compare solutions against, discuss potential solutions, and finally provide what our research has determined to be the best solution for a problem. We will also provide a proof-of-feasibility, which will describe what we can do to validate whether or not our chosen solution addresses the metrics described.

3a. Reactive Interface Design and Prototyping

Intro

Although a "well-designed" user interface is desirable for any project, we should take extra consideration of this for our project beyond what is normally expected. It is ingrained in our project details on a fundamental level: the basis of our capstone work is improving accessibility to Madeus. As a team, we must find a way to easily and efficiently design prototypes to address the possibly-shifting needs of our project. That is, we need to survey potential UI Design/Prototyping tools to develop the best possible UI that we can.

The metrics used in this section will be as follows: “Visual Design Strength”, “Collaborative Abilities”, and “Adaptability”.

The first two metrics are straight-forward. “Visual Design Strength” reflects the variety and strength of options the tool provides to design visuals for a UI. For example, a tool that only provided the ability to place shapes on a canvas would receive a low score, and a tool that provided extensive abilities to create custom shapes, add images, modify colors, add animations, and control other various UI elements would receive a higher score. “Collaborative Abilities” reflects how easy it is to share and collaborate on projects created within the tool. For example, if a tool’s only way to share UIs is to take a screenshot of the window it would receive a lower score; if a tool allows real-time collaboration and native exporting of diagrams to PNG, JPEG, and other formats, it would receive a higher score.

The “Adaptability” metric requires more elaboration. Throughout the course of the semester we will receive information from our sponsors regarding what the ideal GUI would look like given the scope and goals of our project - both in terms of actual design as well as features. Given this feedback, we need to be able to quickly and easily change aspects of the UI to reflect their vision. It would be significantly beneficial to our workflow to reduce the overhead found in changing UIs to a minimum - in other words, our tool should promote adaptability. Another way to think about this metric is a combination of overall ease-of-use, as well as the level of modularity the tool provides in its representation of UI elements - not necessarily atomic elements such as shapes, but “elements” in a more abstract way, like entire subsections of the UI, potentially.

Having UI design tool that matches our metrics would enable us to minimize any ambiguity while seeking information from our sponsors regarding our GUI. Having explicit, tangible mockups and examples would prevent any miscommunications or misunderstandings, and would facilitate more efficient discussions about design patterns by allowing the sponsors to better verbalize their thoughts based on what they see, rather than what they *think* we’re describing.

Considered Approaches

There are three different tools for design mockup being considered:

- Atomic.io, a web-based tool for Chrome.
- Adobe XD, a desktop app by Adobe Systems.
- InVision, a web-based tool heavily oriented around team collaboration.

Approach #1 - Atomic.io:

Atomic.io is a web-based prototyping tool designed for Chrome. Atomic.io provides a good basis for visual design strength - it features basic shapes, images, and grouping of elements into replicable “components”. This combination of basic features can provide UI designers with a great deal of versatility to work with. It also provides a strong system for designing animations; one animation features a “timeline” giving each relevant UI element room to do whatever it needs to do - from moving an element, hiding an element, and scaling an element, it can be done within the timeline. One downside worth mentioning, however, is although UI elements can be replicated by turning them into a “component” and using the “add component” menu, there’s no simple way to copy/paste. This means every duplicated element has to either be manually created and tweaked to have the same properties (tedious), or made to be a component (not every item that sees repetition may be significant enough of an element to be considered a “component”). In other words, some items may fall into an awkward middle-ground in which designers will have to simply pick a method and work around the cons.

In terms of collaborative features, Atomic.io has a number of account tiers that provide differing levels of collaborative ability. The most basic free account allows one individual to work on a UI, and provides a shareable link to the prototype that showcases the UI and all of the associated animations and transitions. Although not included in the free account, there are tiers that allow real-time collaboration, as well as a “comment” feature for those who are not designers but have a shared link. This could be somewhat useful, but for the scope of this project we believe the benefits wouldn’t be utilized enough to warrant getting a paid account.

Finally, adaptability is executed well because of the tool’s ability to define major UI elements as distinct units, as well as its ability to manipulate those units. This gives designers the opportunity to modify small details of elements as well as visual features on a bigger scale. The tool itself is also easy to use, featuring an intuitive design. There are a number of main sections of the workspace - there is a list of pages, layers for a given page, and a canvas featuring the selected page; there is also a toolbox area, and a portion of the screen for modifying details of a selecting element on a page. It is also worth mentioning Atomic.io features a light scripting language. This promotes adaptability in numerous ways, one example being text fields with identical text may reference global variables instead of containing independent instances of the same text. This allows designers to change data in one place and see it reflected in many UI elements, instead of being required to change each and every element's data.

Visual Design: 5/5
Collaborative Ability: 3/5
Adaptability: 4/5

Total: 12/15

Approach #2 - Adobe XD:

Adobe XD is a relatively new product by Adobe Systems, with a focus on comprehensive UI/UX design. It came out of its beta version about a year ago - Oct. 18, 2017, and although it's newer than both of the other options being researched, it has enough unique features to deserve consideration.

In terms of strength of visual design capabilities, Adobe XD seems to provide many similar features that Atomic.io provides; general shapes, images, and "symbols", which are composite elements made of other elements. Animations are implemented in a very minimal way, however. All animated transitions are strictly page-to-page; that is, we can't design a transition that animates specific elements on a page, only how the page transitions to the next page (e.g. slide, fade, etc).

Adobe XD does well with regards to the second metric, "Collaborative Ability". Real-time collaboration isn't supported, but files can be exported and exchanged between designers for collaboration, image files may be exported, and shareable links may be generated (with free support for comments).

This tool does moderately well in terms of adaptability. It provides the means with which elements can be replicated and modified - the functionality is there, but it ultimately can be tedious to use. Adobe XD’s interface itself tends to feel clunky and awkward. This is especially with Adobe’s “artboards” (what is generally defined as a “page” in any other UI designer), as they are all permanently present on the screen. This hinders adaptability by making it difficult to focus on any one element found in an “artboard” because of the constant clutter.

Visual Design: 4/5
Collaborative Ability: 3/5
Adaptability: 3/5

Total: 10/15

Approach #3 - InVision:

InVision is a browser-neural web app designed for UI/UX prototyping.

In terms of visual design strength, InVision provides a unique way of designing interfaces. At its core, InVision’s method of UI prototyping consists of uploading pre-made images from a hard drive and connecting them via clicks on “hotspots”.

“Collaborative Ability” is the biggest appeal of InVision, and one of the main reasons why it initially caught our eye; InVision lends itself very well to software projects. To clarify: it has many mechanisms in place that help facilitate teamwork, more than either of the other two. It provides real-time collaboration and viewing on projects, and there is a task tracker built into every project on InVision, allowing all members of a project to track and add different “task cards” to the default “On Hold”, “In Progress”, “Needs Review”, and “Approved” sections (as well as any user-defined sections). Furthermore, it provides a very extensive ability to “comment” on almost every aspect of a given design mockup/prototype. InVision is also extensively documented, with most of the major functionalities that it provides being detailed in videos on their sleek website.

Because of the workflow mentioned before (InVision’s core is connecting uploaded images via clicks on “hotspots”) InVision doesn’t lend itself very well to adaptability. Because there are no UI “elements”, only pre-made images, the easiest way to adapt a UI design under this tool is to remake the uploaded image, which would be very tedious. Therefore, this tool is on the lower end of the adaptability scale.

Visual Design: 2/5
Collaborative Ability: 5/5
Adaptability: 2/5

Total: 9/15

<u>Chosen Approach</u>

Since analyzing the details of this challenge and researching potential solutions, we've decided the best long-term solution for UI design and UI prototyping is Atomic.io. After thoroughly comparing each of the solutions, we found this tool targets more of our needs than the other two approaches.

Feasibility Table: Reactive Interface Design and Prototyping

	Visual Design Strength	**Collaborative Abilities**	**Adaptability**	**Total Score**
Atomic.io	5/5	3/5	4/5	12/15
Adobe XD	4/5	3/3	3/5	10/15
InVision	2/5	5/3	2/5	9/15

<u>Proof of Feasibility</u>

A potential way to prove the feasibility of our choice later into development is to provide a demo in which a sample UI is refactored into a new UI with the same content and a different layout. The speed and ease with which this is done will determine whether or not we made the correct choice in this analysis. For example, given a desktop app UI with a hamburger/slide menu, team designer[s] should be able to refactor the UI into a "navigation header menu" style UI within 5-10 minutes.

3b. Generation of Code

Intro

Our sponsor's request was to create a GUI that will allow non-programmers the ability to quickly and easily design and implement Madeus assemblies. The back-end should be able to generate the corresponding MAD code that will be ready for execution.

Considered Approaches

With the overall solution in mind, the three solutions being considered are:

- Build an abstraction layer (of sorts) that will provide intermediary functions for interacting with the Madeus model and the GUI
- Branch the current MAD implementation and build the entire GUI around that
- Create a unique MAD solution with code generation at the heart.

Approach #1 - Develop a library of intermediary functions:

This approach has the benefit of being the most modular. Taking advantage of the currently existing codebase is as easy as importing the newly created library and using any needed function. It also holds the benefit of not needing to rewrite the existing code base. Because the existing code base is open source and easily available, the newly created library must be adaptable. This means that the GUI would continue to function properly if/when the current MAD implementation receives updates.

Modularity: 5/5
Future Plugin Support: 5/5
Ease of Application Updates: 5/5

Total: 15/15

Approach #2 - Extend the existing MAD implementation:

Building off of the current MAD implementation holds the benefit of being an all-in-one solution. The overall workflow for this solution would be to take the currently existing MAD implementation, add unique modifications with the GUI and Code Generation in mind, and then push to a new branch.

The downsides to this approach are faced when the current MAD implementation receives updates. Updates to the currently existing codebase may make the modifications obsolete, unusable, or unstable. For these reasons, building off of the existing codebase will probably be shelved for another approach.

Modularity: 2/5
Future Plugin Support: 2/5
Ease of Application Updates: 3/5

Total: 7/15

Approach #3 - Create a unique MAD implementation:

During the initial project selection process, the project sponsors provided a brief overview of what would be required of the chosen team for their specific project. After further research into the Madeus model, technical papers describing exactly how the Madeus model works were uncovered and examined. The freely available technical papers provide a way of creating a unique MAD implementation.

This solution would allow the most flexibility for the programmers. An entirely new deployment method could also be beneficial for the team that originally created the model. It could expose areas for change that were never before imagined, however, potential downsides of this approach include but are not limited to conceptual model updates. If the Madeus model was ever updated and those changes were not accounted for in the new MAD implementation, the GUI's entire code base would become obsolete.

Modularity: 2/5
Future Plugin Support: 4/5
Ease of Application Updates: 3/5

Total: 9/15

Chosen Approach

Based off of the needs of the many, it has been decided that building an intermediary library would be the best fit for this application. This approach allows the most flexibility in the areas of modularity, future plugin support, and ease of application updates.

Building a unique library provides the flexibility to build what's necessary. The table below shows how each approach was evaluated. It can be easily deduced that the library of intermediary functions is the most logical approach for the generation of code in our GUI.

Feasibility Table: Generation of Code

	Modularity	**Future Plugin Support**	**Ease of Application Updates**	**Total Score**
Intermediary Library	5/5	5/5	5/5	15/15
Existing MAD Implementation Extension	2/5	2/5	3/5	7/15
Unique MAD Implementation	2/5	4/5	3/5	9/15

Proof of Feasibility

Providing proof for this concept's viability comes from research pertaining to all the options explored. Building a unique library containing only pertinent functions is the easiest and most direct way of meshing multiple programs/paradigms together. The GUI will be written in the same language as the current MAD implementation. The majority of work outside of creating the GUI itself will be concentrated on the library. The library itself will also be written in the same language as the MAD implementation to try to avoid as many conflicts as possible. As the current MAD implementation is written in Python, it will be widely used throughout the entirety of the solution.

3c. *Facilitation of Plugins*

Intro

Our sponsors want to be able to add features to this software beyond what we deliver to them in May, since it is not feasible for any of us to predict every possible functionality that will be needed. In order to ensure that our software is extensible and customizable, we have been tasked with making our software capable of facilitating plugin functionality. Unlike some roadblocks, this is not something that can be solved by simply implementing a library of functions or utilizing a pre-built tool. Instead we must determine what framework and mindset to develop our software in such that the code we build will naturally facilitate plugins.

Considered Approaches

The language or tools used are not the deciding factors, since every programming language we are have considered (Python, Javascript, etc.) are all capable of this. The most important aspect for facilitating plugins is the way that the code is designed. We have determined three possible coding methodologies that can handle the creation and maintenance of plugins after the initial delivery of our software. The key points of interest are:

- Degree of possible extensibility
- Flexibility of security
- Level of challenge for end-users

A framework which sufficiently meets these three criteria will be a good candidate for use in our software.

Approach #1 - Shared Libraries:

This is one of the most common ways of implementing plugins across many languages. Shared Libraries are also often referred to as Dynamic Shared Objects or Dynamic-link libraries (DLLs). In essence, a shared library plugin structure involves a library of files that is dynamically loaded at runtime, such that the files can access each other and make use of functions across them. The dynamic loading ensures that the core system can run with or without any plugins being accessible, and that plugins can be found as they appear in the library, and (assuming they are written well) can be integrated into the program.

Shared libraries allow for a great level of control over what the plugins can do, so long as we develop a sufficiently customizable API for our software. This methodology will function like a Glass-Box; the end-user will be able to extend the software and see how it works, but not *change* core functionality. This will ideally suppress the risk of the end-user changing something about the software that they might not want to change. However, this is a more complex way to implement plugins for the end-user, as anyone writing a plugin will require sufficient knowledge of programming, as well as how our internal system works.

Degree of Extensibility: 5/5
Flexibility of Security: 4/5
Ease of Use for End-Users: 3/5

Total: 12/15

<u>Approach #2 - Script Directory:</u>

Script directories are another common way of implementing plugins. In this methodology, there are scripts written in some kind of scripting language (Python, Lua, etc.) that are picked up by the main application and ran in order to modify parts of the code. Since Python is our ideal language, that makes this method more viable than it would be otherwise.

This method is less robust, since scripting primarily uses existing functions instead of creating entirely new ones for extension. On one hand, that means that a Script Directory is more secure, and *potentially* easier to create plugins for, but on the other hand it means that the software would be less extensible in the long run. The ease of plugin creation for this method would also not be dramatically less that in a Shared Library, since the plugin creator will still need a solid understanding of programming and of our software.

Degree of Extensibility: 3/5
Flexibility of Security: 4/5
Ease of Use for End-Users: 3.5/5

Total: 10.5/15

<u>Approach #3 - Access Table:</u>

Using an Access Table to handle plugins is highly uncommon, but may work. This methodology would involve the creation of a table of values for each accessible part of the software. A plugin would edit these permissions, allowing the end-user to change certain features of the GUI, such as colors and sizes.

However, this is the least robust form of extensibility, functioning closer to a Black-Box since it would only allow the editing of minor features without a view of the internal workings. Therefore, this methodology will serve our goal of long term extensibility the least, but would likely be the most secure and easiest for the end-users to use.

Degree of Extensibility: 1/5
Flexibility of Security: 5/5
Ease of Use for End-Users: 4/5

Total: 10/15

Chosen Approach

In the table below, we have aggregated all of the statistics for each of the three approaches, based on the three key points of interest detailed earlier. Each approach has been assigned a score out of 5 for each of the points of interest based on the information we found in our research of the approaches.

Feasibility Table: Facilitation of Plugins

	Degree of Extensibility	**Flexibility of Security**	**Ease of Use for End-Users**	**Total Score**
Shared Libraries	5/5	4/5	3/5	12/15
Script Directory	3/5	4/5	3.5/5	10.5/15
Access Table	1/5	5/5	4/5	10/15

Our chosen approach is the Shared Library methodology. We have chosen this approach for a variety of reasons. It allows for the most complex extensibility, which is the primary goal of this tech challenge. The more complex creation of plugins with this method is less important, since our end-users are expected have at least some knowledge of programming, and not every end-user will be creating plugins anyway. Since our software is for simulation, visualization, and generation of MAD code, and will not actually be accessing networks or anything in that vein, security is not a huge issue. Nevertheless, this method will prevent end-users from changing the core functionality of the software. For these reasons, we believe this to be the best approach to the facilitation of plugins for our project.

Proof of Feasibility

We already know that this method is at least viable, since it is one of the most common ways of accomplishing plugin support. However, in order to confirm that it is the best fit for our project, we will need to do some additional testing. The plan going forward is to create a small program (likely a GUI), and set up a Shared Library system to try and add functionality reminiscent of our envisioned end project via plugins. We will know that this is a good choice if we are able to successfully extend the test GUI, and are also able to prevent certain features from being extended. This will show that our chosen method is not only viable, but meets our standards for extensibility and allows us to section off code that could be dangerous if the user has open access to it.

3d. Simulation of MAD Assembly Deployment

Intro

One of the main requirements besides the Graphical User Interface is to have a simulating run-time of the MAD deployment after the user created it in the Graphical User Interface. Dynamic runtime of the simulation will allow the user to see processes running through their directed acyclic graphs once created and will also allow the user to test the deployments. Frameworks that facilitate simple, fast, and intuitive animations and graphics will researched and considered to meet this technological challenge.

Considered Approaches

The metrics used to find a framework suitable for dynamic runtime of the simulation will be judged based on:

- Good visuals of two dimensional animations
- Implementation of graphics with a large library for support and future development
- Cross platform compatibility

A framework that meets these marks on each of these metrics will provide the user of the GUI effective visualization of their MAD assembly running through its deployment.

Approach #1 - Pyglet:

Pyglet is a library for the Python programming language that provides an object-oriented application programming interface for the creation of games and other multimedia applications. Pyglet runs on Microsoft Windows, Mac OS X, and Linux; it is released under BSD Licence. Pyglet looks promising for developing 2 dimensional animation because it was designed to develop games and other visually rich applications. It supports windowing, user interface event handling, OpenGL graphics, loading images and videos, and playing sounds and music. The major features of Pyglet are:

- **No external dependencies or installation requirements.** For most application and game requirements, *pyglet* needs nothing else besides Python, simplifying distribution and installation.
- **Take advantage of multiple windows and multi-monitor desktops.** *pyglet* allows you to use as many windows as you need, and is fully aware of multi-monitor setups for use with fullscreen games.
- **Load images, sound, music and video in almost any format.** *pyglet* can optionally use FFmpeg to play back audio formats such as MP3, OGG/Vorbis and WMA, and video formats such as DivX, MPEG-2, H.264, WMV and Xvid.

2D Animation: 5/5
Library for Graphics: 4/5
Cross-Platform: 5/5

Total: 14/15

Approach #2 - Kivy:

Kivy is a cross-platform python library for rapid development of applications. Kivy allows innovate user interfaces and multi-touch apps. Kivy uses “Widgets” which are graphical representations of objects rendered using a canvas. A Widget can be seen as both an unlimited drawing board or a set of drawing instructions. Kivy also provides many different types of Widget animations. These animations allow sequential, parallel, and repeating animations. Kivy implements a module called Atlas. Atlas manages textures by packing multiple textures into one. Atlas reduces the number of images loaded and speeds up the application loading. Atlas has been used to create a wide-ranging assembly of 2 dimension applications and games.

2D Animation: 3/5

Library for Graphics: 5/5
Cross-Platform: 5/5

Total: 13/15

Approach #3 - PyGame:

PyGame is free and Open Source python programming language library for making multimedia applications built on top of the excellent SDL library. PyGame is highly portable and runs on nearly every operating system. The PyGame module is great when relating to advanced graphics and sound. PyGame does not work well with an interactive shell. Pygame has no text input and output, instead the program displays output in a window by drawing graphics and text to the window. PyGame is currently the most popular and portable game library for python, with over 1000 free and open source projects that use pygame.

2D Animation: 4/5
Library for Graphics: 3/5
Cross-Platform: 5/5

Total: 12/15

Chosen Approach

The three frameworks researched were chosen because of their popularity and large community support. While researching these frameworks each one of them had features for and against them. The framework that provides the best overall practicality for simulating the MAD assembly deployment taking into account all the metrics is Pyglet. Pyglet provides a good library of 2D graphics, sprites and static images which can be used in conjunction with each other. Pyglet has an extensive amount graphics, documentation and tutorials. Pyglet is fully cross platform compatible. Finally Pyglet uses a OpenGL backend where all rendering is done efficiently by the graphics card, rather than the operating system.

Feasibility Table: Simulation of MAD Assembly Deployment

	2D Animation	**Library for Graphics**	**Cross-Platform**	**Total Score**
Pyglet	5/5	4/5	5/5	14/15
Kivy	3/5	5/5	5/5	13/15
PyGame	4/5	3/5	5/5	12/15

Proof of Feasibility

Providing proof of feasibility for this tech challenge is obtained through watching and creating tech demos with each framework. Creating a simple demonstration with each framework that demonstrates a metric used for judging of the simulated deployment will provide proof of feasibility for that particular framework.

4. Technological Integration

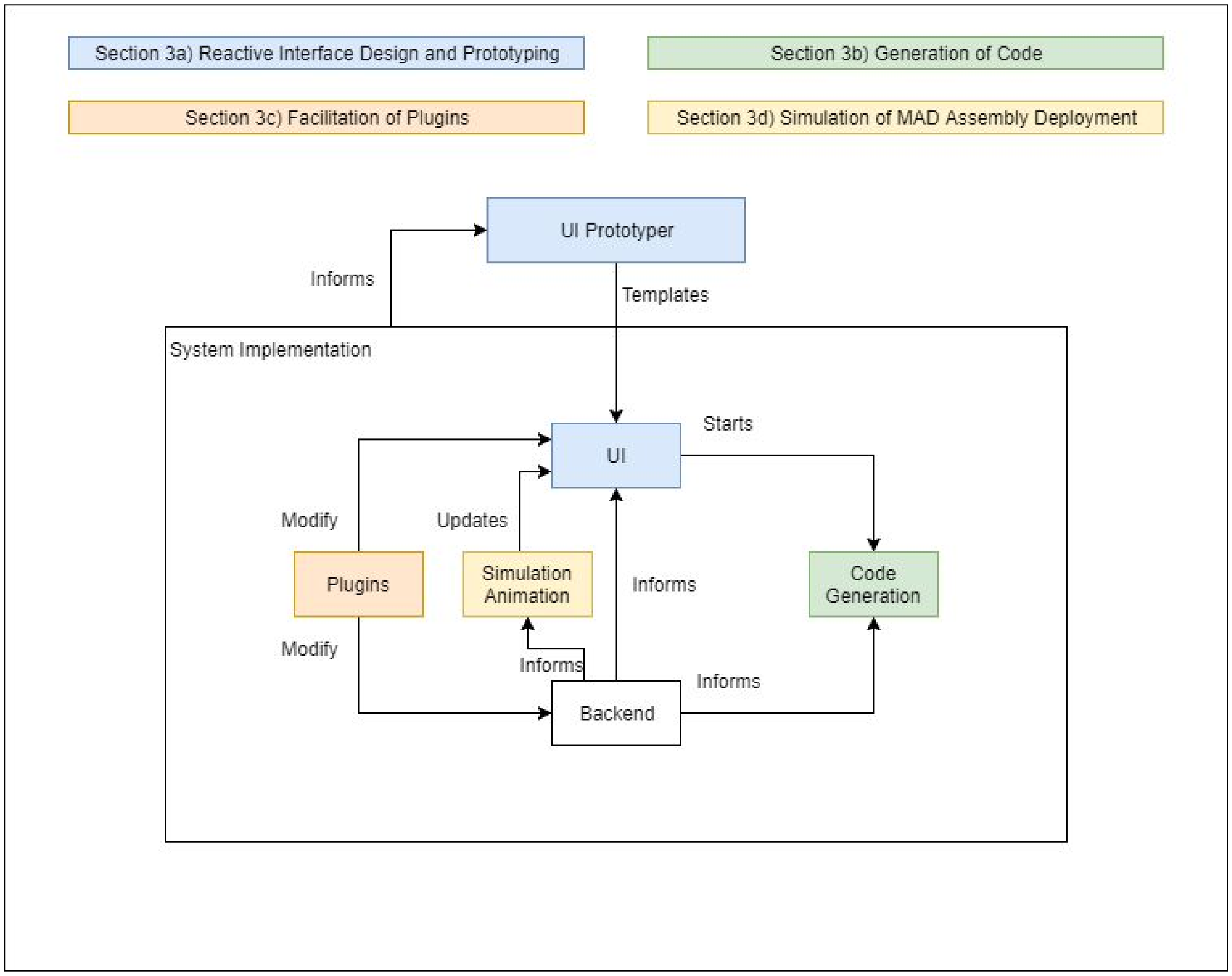
Section 3a) Reactive Interface Design and Prototyping
Section 3b) Generation of Code
Section 3c) Facilitation of Plugins
Section 3d) Simulation of MAD Assembly Deployment
UI Prototyper
Informs
Templates
System Implementation
Starts
UI
Modify
Updates
Plugins
Simulation Animation
Informs
Code Generation
Modify
Informs
Informs
Backend

Fig. 1: Predicted Software Architecture

One of the benefits of having multiple Python libraries and frameworks will be having the ability to use them in conjunction with one another, especially when related to our Graphical User Interface front-end and the MAD backend. The Python import system allows us to build modules and stack them together to be used harmoniously with associating and building certain modules together.

The simulation of the MAD deployment will depend upon the user generated Madeus model in the GUI. As a result, the simulated runtime will need to be integrated on top of or simultaneously with the generation of code.

The plugin facilitation, in its shared library system, should not need any support from other parts of the software aside from helper functions. The integration itself should be mostly standalone, and only modify the other parts instead of being dependent on those parts.

As per Fig. 1, the instance of UI in the code should be inspired by the UI prototyper and populated/informed by elements present in the backend. The simulation animation will update the UI, and the plugins may modify the UI. The UI will serve as the connection between the user and MAD code generation; a way to generate code from an assembly, such as a button, will be given. This code generation also features elements from the backend, such as the current elements present in the assembly.

5. Conclusion

As Team Amadeus, we are working with Dr. Hélène Coullon and Frédéric Loulergue on the “Framework for Distributed Software Automatic Deployment Execution and Analysis” project. This project involves developing a GUI frontend for the Madeus model, a complex deployment system that utilizes parallelism.

Our GUI must provide a streamlined way for users to create Madeus assemblies without worrying too much about the behind-the-scene details. To accomplish this, we must first understand the technology required and determine how we will overcome the technological hurdles.

At this time, we have determined four major technological hurdles. We must be able to *Reactively Create Interface Designs* for our GUI, with a system that should *Generate Executable Code*. Additionally, our software must be flexible and extensible. We must develop with these principles in mind, such that our software *Facilitates Plugins* and can be extended / modified over time. Finally, our GUI must allow for the *Simulation of MAD Assembly Deployment* through animations.

Our proposed solutions for each of the aforementioned problems, in addition to our confidence level in each of those proposed solutions are detailed in the table below.

Table of Solutions

Challenge	**Proposed Solution**	**Confidence Level**
Reactive Interface Design and Prototyping	Atomic.io	4/5
Generation of Code	Shared Library	5/5
Facilitation of Plugins	Shared Libraries	5/5
Simulation of MAD Assembly Deployment	Pyglet	3/5

We are confident that our analysis of our foreseen technological problems - and the solutions that we have selected for each of them - will serve to enhance the development process, and will elevate our end product to a higher level of quality.