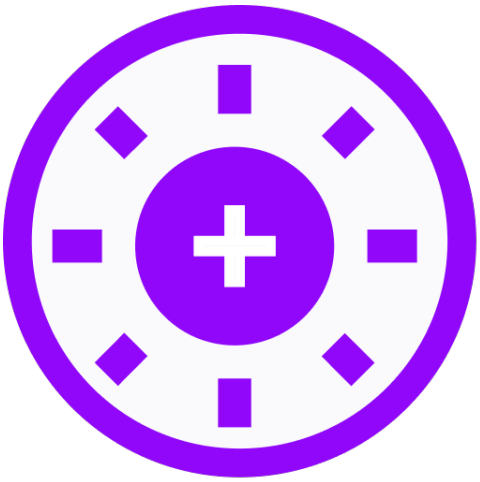

Chalmers tekniska högskola

OBJEKTORIENTERAT PROGRAMMERINGSPROJEKT

Final Report for EnergyCalculator

Alexander Larnemo Ask
Jonatan Bunis
Vegard Landrö
Mohamad Melhem
Alexander Larsson Vahlberg

VAMAJ

2019/10/25

Contents

1 Introduction

The harnessing of solar power for both commercial and private use is a growing trend. This project aims to adopt that trend by providing value to the user through information about their potential gains with solar power. The user can through the program see whether solar power would be a viable investment economically, as well as if there would be a future possibility of profit.

The comparisons are made based on the users current electricity provider. This is meant to further encourage the user towards investing in solar power. The user should also be assisted in his/her choice of solar panel so that a good balance between installation cost and produced energy can be suited to the user. The program aims to help private residents as well as businesses with the decision process concerning the acquirement and use of solar power. These two groups are therefore the projects stakeholders.

1.1 Design goals

The goal is to have a loosely coupled and modular application that is easy to extend. It should be simple to add new calculations and also to change what the calculations are based on.

1.2 Definitions, acronyms, and abbreviations

The following is a list of words used throughout this document and their explanations:

GUI ”Graphical User Interface” also ”User Interface”, the part of the program that the user sees and interacts with.

DoD ”Definition of Done”, a list of requirements that have to met for a User Story to be concidered done.

UserStory short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.

Java an independent programming language.

IDE Integrated Development Environment.

MVC Model,View;Controller. A well known design pattern which is hugely used in applications that have a GUI representation. It aims to seperate the

Hi-Fi High fidelity, a sketch with much detail.

JavaFx JavaFX is a software platform for creating and delivering desktop applications, as well as rich Internet applications (RIAs) that can run across a wide variety of devices.

SceneBuilder a visual layout tool that lets users quickly design JavaFX application user interfaces, without coding.

FXML An XML-based user interface markup language created by Oracle Corporation for defining the user interface of a JavaFX application

Figma A GUI design Mockup Tool.

UML ”Unified Modeling Language”, A visual language for mapping and representing systems. Can be used for modelling och class diagrams, sequence diagrams and more. In this project used for Domain and design model.

NH "Number of Hours", estimated time.

URL an acronym for Uniform Resource Locator and is a reference (an address) to a resource on the Internet.

JSON JavaScript Object Notation

2 Requirements

2.1 User Stories

This subsection lists the user stories that define the application's requirements, in terms of the end-user's wanted functionality. User stories surrounded by asterisks (E.g. *UserStoryX*) are not implemented in the application due to time constraints. User stories that are developer focused can be found in the appendix.

2.1.1 UserStory1.2 : Solar panel choice

Description:
As a user I want to be able to select different solar panels and compare them with my current electricity provider plan, in terms of cost and produced energy.

Confirmation:
- The user can select different solar panels in the application.
- The user can input electricity provider specifics.
- Calculations are based on the given information.

2.1.2 UserStory3 : Available installation space

Description:
As a homeowner I want to know the cost of a potential solar-panel installation based on the available space my roof has.

Estimated time: 1 hour

Confirmation:
- The user is able to enter a house's available space for a solar panel installation.
- The user gets the installation expenses based on input.

2.1.3 UserStory4 : Current electricity consumption and cost

Description:
As a homeowner I want to be able enter my current electricity cost and consumption so that comparisons can be based on it.

Confirmation:
- The user can enter electricity cost
- The user can enter electricity consumption.

2.1.4 UserStory4.1 : Energy production surplus

Description:
As a money-loving person, I want to check if I would get electricity surplus so I can sell some of it.

Estimated time: 1 hour

Confirmation:
- The user gets expected solar energy production.
- The user can enter energy consumption.
- The program calculates surplus based on a user's energy consumption and solar energy production.

2.1.5 UserStory4.2 : Break-even

Description:
As a money-loving person, I want an estimation of when I would start turning a profit after a purchase of solar power.

Estimated time: 1 hour

Confirmation:
- The user can see an estimate of when a property's solar power setup would start turning a profit.

2.1.6 UserStory6 : Solar hour statistics

Description:

As a solar power prospector I want the yearly amount of solar insolation my property has to be a variable in calculations, so that I can determine if solar power would be viable.

Estimated time: 2-12h hours

Confirmation:
- The amount solar insolation a property has is based on actual real world data and statistics.

2.1.7 UserStory7 : Government subvention

Description:
As an economical person I want to be able to see how much the government would be willing to subvent in my potential purchase of solar power to save money.

Estimated time: 1 hour

Confirmation:
- The user gets an estimate of how much the government would be subventing the installation cost.

- The user gets an estimate of the subvented installation cost.

2.1.8 UserStory7.1 : User friendly design

Description:
As a user I want the program to have a straight forward GUI that conveys what input is needed for a prospect of a solar setup based on my prerequisites, since that's what I'm interested in.

Estimated time: 4 hours

Confirmation:
- The program clearly conveys to the user what input is required and where said input is needed.

- The user finds the GUI pleasant to look at and easy to navigate.

2.1.9 *UserStory10*: Estimated energy consumption

Description:
As a user I want to be able to calculate electricity consumption based on my address, in order to calculate electricity costs.

Estimated time: 5 hours

Confirmation:
- The user's address can be entered.

- The address is taken into account in all relevant calculations.

2.1.10 *UserStory10.2* : Environmental effect

Description:
As a homeowner I want to be able to see how much my current energy alternative effect global warming so that I can compare it with solar power

Estimated time: 6 hours

Confirmation:
- A user's current energy source's environmental effect is compared with a potential solar powered energy source.

2.1.11 UserStory12 : Find current location

Description:
As a user I want an easy way to use my current location in the calculations, since I want calculations based on where I live.

Estimated time: 2-12 hours

Confirmation:
- The user can press a button to find the current location.

- Current location is used in calculations.

2.2 Definition of Done

The following is the DoD used in the project:

1. All of the acceptance criteria of the user story have been fulfilled.
2. All code is commented.
3. All code has tests that run without failure.
4. All code has passed code review and potential changes have been made.
5. All changes have been committed and pushed to the master branch and the master branch is running without error.
6. The RAD and SDD Documents have a few paragraphs about the classes pertaining to the user story.
7. The Design UML has been updated to show relevant classes.

2.3 User interface

In the beginning of the project a preliminary sketch of the user interface was made to help with the visualization of the program (See figure 1). []

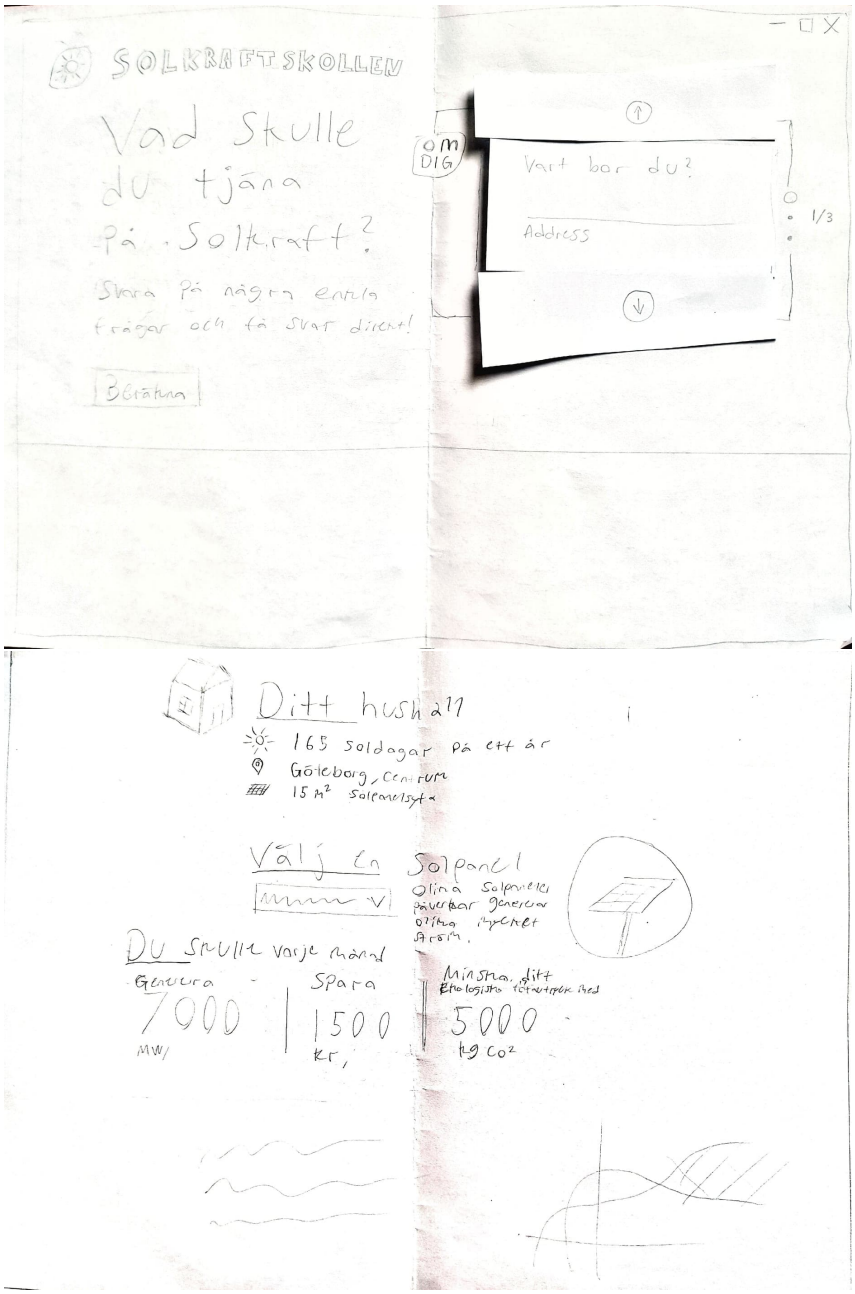

Figure 1: A Preliminary sketch of the interface

Later the interface was made more defined through the creation of a High fidelity (detailed) mockup sketch, created in ”Figma” (See figure 2). [1]

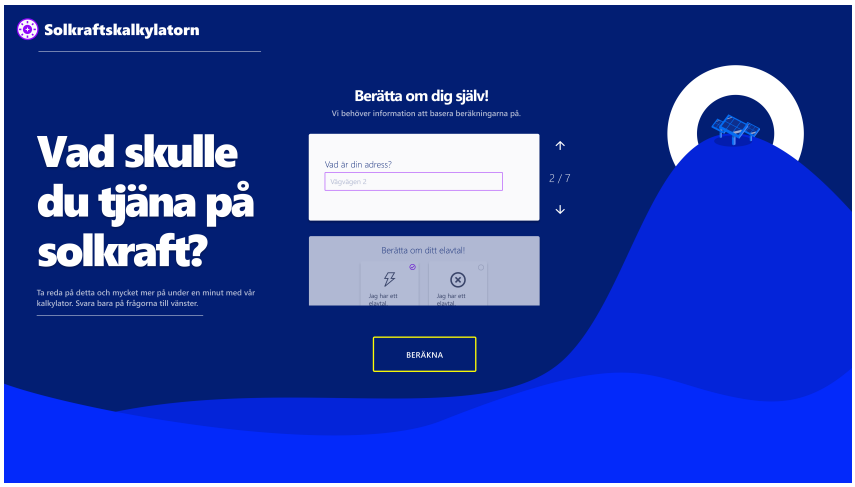

Figure 2: A Hi-Fi mockup of the interface

The Graphical user interface consists of a main scene which has a dynamic part where questions needed for the calculations are viewed as question cards. The user is be able to navigate between different question without breaking the flow or changing scenes. When answering the whole form, the results can be shown in another page. This flow was made to create a clear distinction between information gathering and information provision.

The GUI was created with the intent of making the process feel light and interesting. The final result of the implemented GUI can be seen in figure 3. The GUI differs from the mockup mostly due to technical limitations and practical reasons.

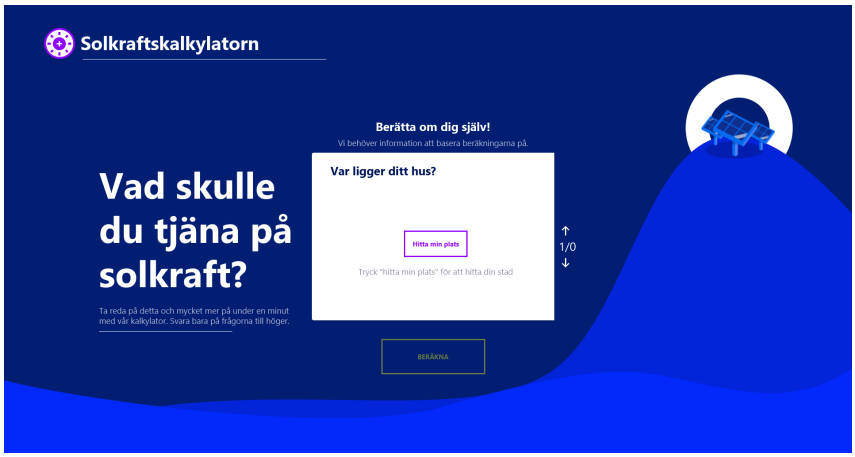

Figure 3: The GUI of the actual program

3 Domain model

The program is a type of calculator that uses user input and input from various **APIs** to produce informative results which is then fed back to the user through a **GUI**. The user is the main information object of the program and the user is the owner of a number of properties. The properties in turn hold most the information necessary in order to make the calculations. The **model** aims to mimic the information relations that would be found in the real world so that information can be accessed and encapsulated in intuitive ways. The way the application works is illustrated in the domain model (figure 4) and further explained in chapter 3.1.

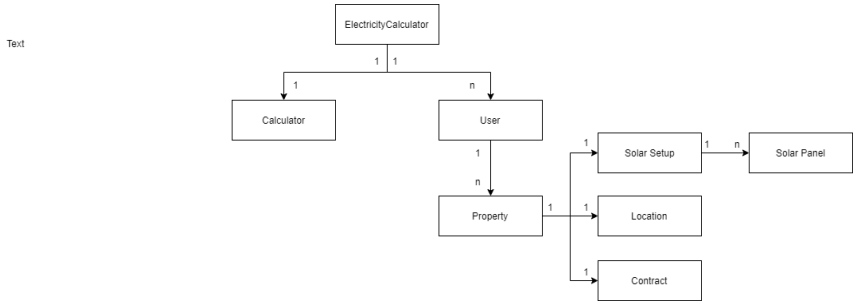

Figure 4: The domain model of the program

3.1 Class responsibilities

The **calculator** is responsible for the main logic of the program i.e. the actual calculations in the program. The **user** is an object that holds some data specific to the current user, it also holds the important paets of the model that are going to be used by the **calculator**, that is to say, the user is the central object in the model hierarchy. The **calculator** will retrieve information from the **user** (i.e. the model) perform its calculations after which the view will be updated to display results.

The structure of the program has been created specifically so that the information structure mimics reality (for readability and structure) and so that the model is independent of the rest of the program so that the program is modular.

Every user can have multiple Properties. A **property** is the object with the actual necessary information to perform the calculations. The information is stored as objects; **Solar Setups**, **Location**, Electricity Provider **(contract)**. The actual information in theses objects is gathered through different measures: through data from APIs and through direct input from the user.

The **location** class is a representation of a location. It holds data and attributes that are used in the calculations.

The **Solar Setup** holds the needed logic to instantiate a **solar setup** object. A **solar setup** has multiple **Solar panel**s and information about the collection of **solar panel**s as a whole.

Contract is an abstract representation of an electricity provider contract. The **contract** is used by different parts of the code in order to store needed information about the users electricity provider to carry out further calculations.

4 System architecture

At a high level the program architecture was designed with design principles in mind. Most prominently were the **"SOLID"** principles considered. Coherency and independence were constantly taken into consideration when implementing components, this meant following the **"High-Cohesion, Low-Coupling"** principle which resulted in a code with less complexity and better readability.

The program was also designed with extensibility and maintainability in mind, this meant following the well known **"MVC"** pattern to make sure that the **model** was without external dependencies so that it could potentially be reused.

To further increase re-usability of the code, classes were constantly grouped into logically coherent components and the structure was built trying to resemble reality. Furthermore to separate calculations from information objects, a calculation component was created separately from the rest of the model. This meant that one could always expect to make calculations in one place, since the program was so heavily focused around calculations.

4.1 Model

The **Model** Package holds classes representing data as well as the core logic of the program.They represent the current state of the applications and hold the main logic. The model consists of different packages which in the turn represents different aspects of the model.

1. **Calculator**

Name	Package	Responsibility
Calculator	calculator	This interface is package-private and contains three methods for all calculators that implement it. The first method is 'calculate' with a hashmap as argument and return type. The second is getRequiredInput, it returns a set with Datakeys representing the required data for a specific calculation. The third method is getOutput, this also returns a set of Datakeys but this time representing the output of a calculation.
Datakey	calculator	This Enum class contains the names of all values handled by the calculators. It's used as keys in a Hashmap that stores values connected to the Datakeys
CalculatorFacade	calculator	This class contains the only two public methods in the package, calculateAll and calculateSpecific. Both take a Hashmap with Datakey and Double and returns a new Hashmap that has the values of the calculations added. It also contains a final static list with an instance of every class that implements the Calculator interface. It uses the getRequiredInput and getOutput methods to find what Calculator to use with a given set of data.

Name	Package	Responsibility
ElectricitySurplusCalculator	calculator	this class calculates the amount of electricity produced compared to how much electricity is consumed. The calculate method returns how many more kWh's are produced compared to consumed, it returns 0 if consumption is higher than production.
InstallationCostCalculator	calculator	This calculator class calculates the cost of a potential installation of solar panels. The calculations are based of a solar panels cost, a solar panels required installation space and the available installation space of a property.
SolarPanelProduction	Calculator	This calculator class is responsible for calculating how much power (kW/h) a given solar setup is able to produce. It depends on several variables retrieved from calculatorFacade, and uses these values to calculate the power output of the specified solar panel.
LevelizedCostOfElectricity	Calculator	This calculator class is responsible for calculating the levelized cost of electricity from a potential solar installation. Levelized cost of electricity is the cost per generated kWh over the expected lifetime of the setup.
YearsToBreakEvenCalculator	Calculator	This calculator class is responsible for calculating how many years it will take to earn back the cost of the potential solar setup. It uses data on the capital and operational cost of the setup and electricity prices of the grid.

2. **Contract**

Name	Package	Responsibility
Contract	model.contract	Abstract representation of an electricity provider contract.Holds data common to an electricity contract

3. **Property**

Name	Package	Responsibility
Property	model.property	Used to create property objects
Location	model.property	Holds location attributes.
Coordinate	model.property	has x,y coordinates which are used in location

4. **SolarSetup**

Name	Package	Responsibility
SolarPanel	model.solarsetup	A data representation of a solar panel holding attributes pertaining to such an object.
SolarSetup	model.solarsetup	Holds SolarPanel objects and other attributes that belong to the whole setup and are needed for calculations.

5. **User**

Name	Package	Responsibility
User	model.user	The user of the program, holds most of the model objects. Exists for future possibility of extension to multiple users.

4.2 ViewController

The **ViewController** Package contains both controllers with strong connections to their respective FXML files in the **View** package as well as a standalone primary controller **PrimaryController** and a utility class **SceneSwitcher**. As mentioned, the **views** package consists of controller classes for FXML files. These controllers are divided into dynamic and non-dynamic based on the way in which the application uses the FXML files, whole-page FXML controllers are considered non-dynamic whereas list items are considered dynamic and similar.

Name	Package	Responsibility
PrimaryController	ViewController	Connects the Viewcontroller package with other code areas. It acts like facade for the package as well as providing the other controllers with shared functionality.
SceneSwitcher	ViewController	Is used to change the ”non-dynamic” scene in the application.

4.3 Services

The services package is a package that provides data gathering from external sources. This could be in the (actually implemented) form of APIs or potentially files. The service package exposes its functionality through the **ServiceFacade**.

Name	Package	Responsibility
ApiParser	services	An abstract class that retrieves data from a url and returns a wanted object. The concrete implementations of this parser treats the data retrieved (which is a string) as different types of objects. For example, one implementation of ApiParser is ApiJsonParser which parses the string and returns a JsonObject.
LocationCreatorAPI	services	Creates location objects by fetching data from a specific API. Holds much code specific to a single API, could almost be seen as a sort of adapter to the API.
ServiceFacade	services	Exposes wanted functionality of the service package outward toward the rest of the program.
Geolocation	geolocation	This is the class that ties the whole Geolocation package together, and acts as the ’facade’ towards the rest of the service package, and any potential classes that want to use the Geolocation service. It fetches the IP address of the user, and then gets the current latitude/longitude and city of the user.

Name	Package	Responsibility
GeolocationService	geolocation	This is the class that actually interfaces with the API itself (Maxmind GeoIP2) to gather the relevant data. It loads the database file, and then uses a given IP-address to lookup the position of the user. This position data is then returned and later used in the Geolocation class to be forwarded to the model to be used for calculations and other operations.
IPAddress	geolocation	The purpose of this class is to automatically provide the IP-address of the user, to be used in the GeolocationService class. The IP is automatically gathered using Amazon Web Services (AWS), and then returned as a String that can be used when fetching the position data of the user.

4.4 resources

Resources package holds resources or data that can be accessed by the code. Images, icons, fonts, databases and FXML files can all be found in resources package.

5 System design

Since the program has a graphical user interface representation, a **MVC** pattern was used to separate the different components of the program (see figure 5). The program is based on **JavaFx** graphical component. Ultimately, the conclusion was made that trying to implement a strict **MVC** separation with a separate Controller class may not be the best way to design a **JavaFX** desktop application. This meant the usage of a different version of the **Model-View-Controller** design pattern - which was more suitable for the application- called Model-ViewController where the view and controller modules are merged into 1. This was a better choice due to the nature of JavaFX entailing that these components are nearly inseparable.

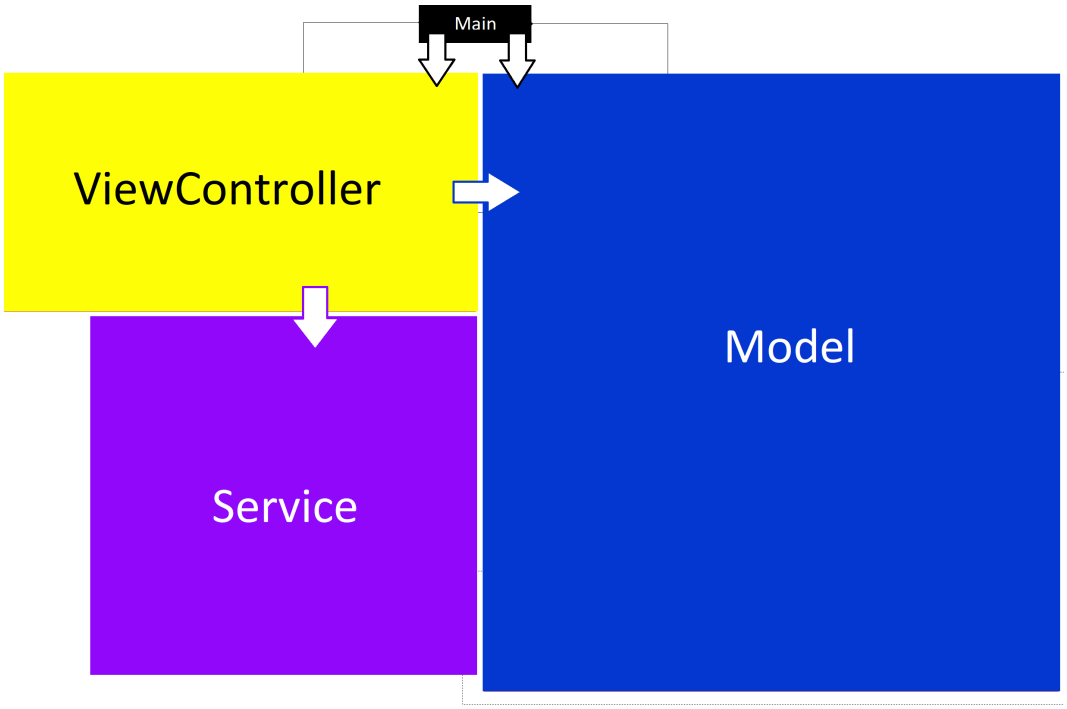

Figure 5: Blocks representing the design model of the program, a clear MVC pattern with a service package

The **”Modelfacade”** class is the class used as the model's face outwards towards the rest of the program. The whole **model** including the **Modelfacade** is free from dependencies on the rest of the program. This was a conscious decision as it follows best practices. More specifically does the structure follow **MVC** which first and foremost requires that the **model** package is not dependent on any other part of the program. This makes the code modular and creates the possibility of using different views and controllers for the same model (see 6).

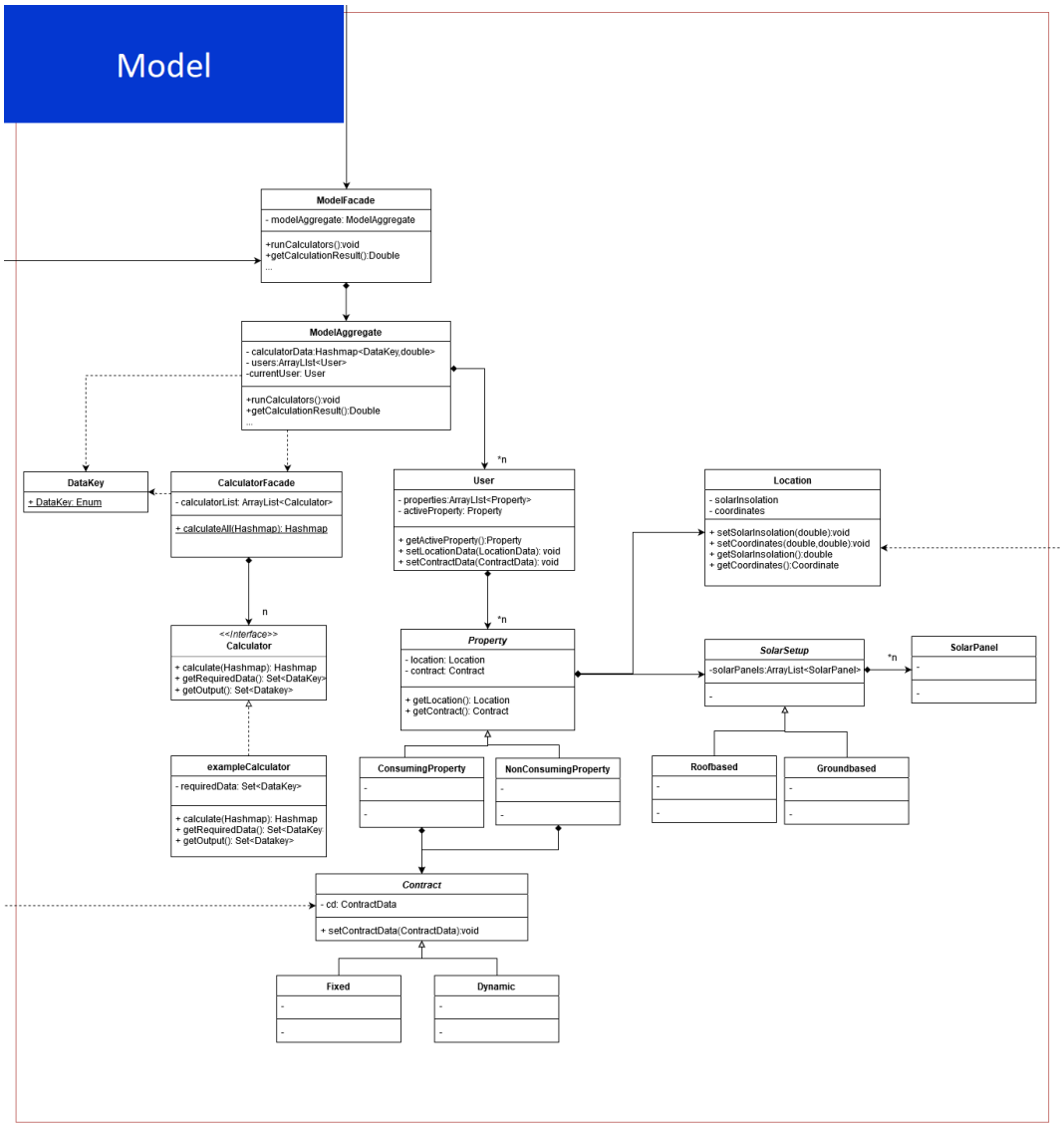

Figure 6: the model block

In order to gather data from the model from outside sources (such as API:s or files) we have created the **service** package(see figure 7). The purpose of this package is to provide completed model objects, without the **model** being dependent on the API:s being used to gather the data. This allows us to seamlessly switch the source of data, without affecting the **model** at all. This is achieved by using **interfaces** that guarantee that the data sources used in the **service** package will be adapted to objects that fit our **model**. The different implementations of the ”creator” classes have code specific to the way that data is gathered but with the product always being the expected object. For example the **”LocationCreatorAPI”** needs to connect to a **NASA** API [2] the way that this API has to be parsed specifically is contained here, yet the result is the expected: a **”Location”** object.

To make it possible to connect our program to any **API** an **API parser** was created that collects data in the form of a string from the **API** and then parses the String in different ways depending on the implementation of the parser. The class follows the Template method pattern [3] so that new types of parsers can be easily added without needing to duplicate code.

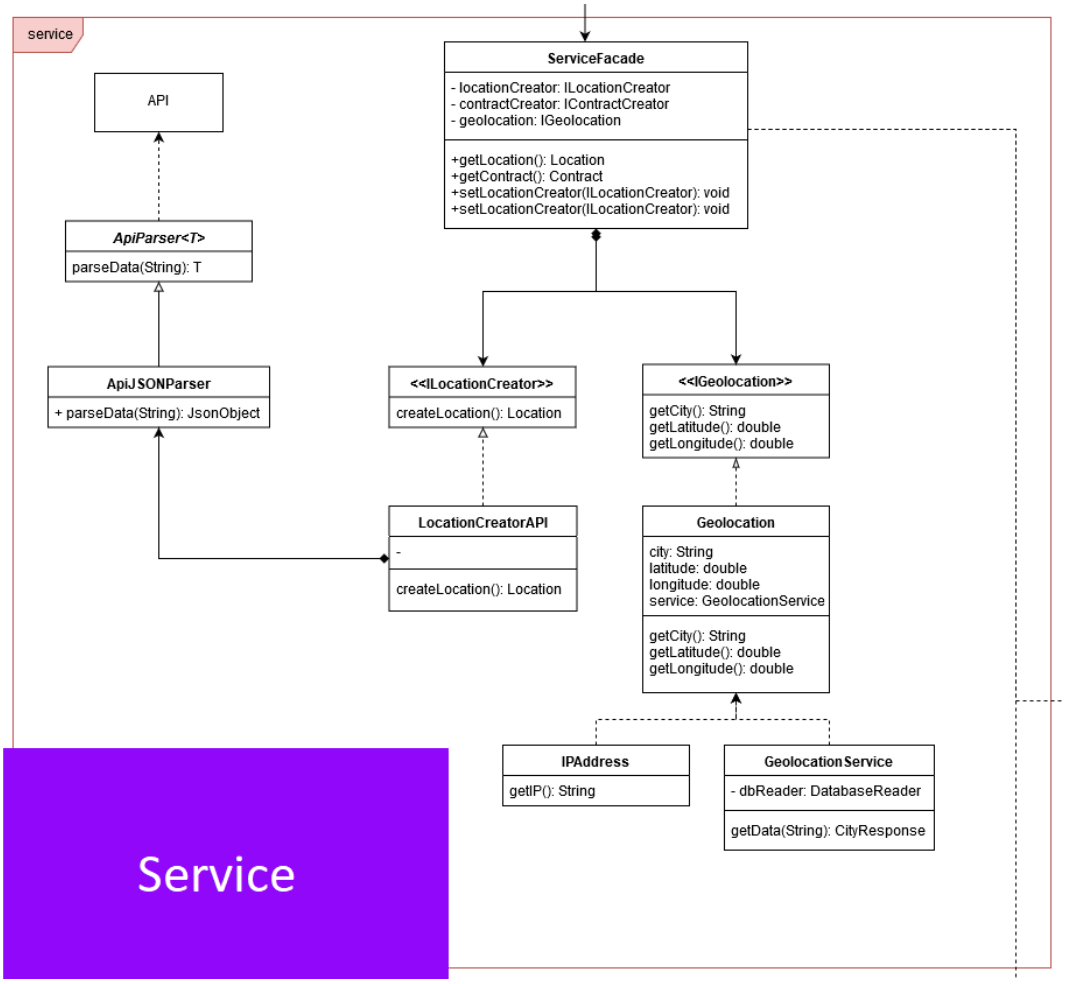

Figure 7: The service block

As mentioned previously, the code implements a different approach of the classical **MVC**. **SceneBuilder** is used to transfer all UI(user interface) code into **FXML** files, which means that there is no ”code” for the UI components.The logic is therefore handled inside a Controller class that is strongly coupled with the SceneBuilder-generated fxml. Due to that, it became harder to clearly separate the **views** from the **Controller** since there is no code for the View. The **Controller** class is then defined in the respective **FXML** file. This also meant that view and controller logic had to be boundled together which meant the creation of the **ViewController** package (see figure 8).

Utility classes -**scene switcher**. The**scene switcher** class is a utility class that allows for easy scene switching in the primary stage. The controller classes can then use **scene switcher** to change the current scene, without having to retrieve the reference to the stage in an unnecessarily complex way.

```
Stage stage = (Stage) node.getScene().getWindow();
stage.setScene(...);
```

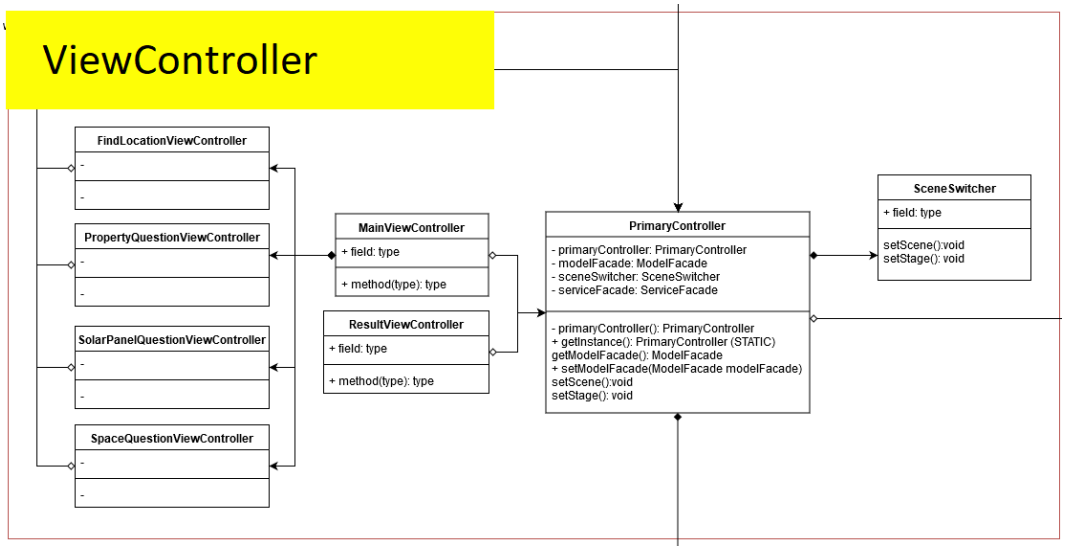

Figure 8: The view-controller block

6 Quality

Tests are performed using **JUNIT**[4] and are found in the **"src/test"** directory of the project. In addition to testing the quality of the code has been controlled using an analytical tool called **"Bettercodehub"**[5]. This tool checks the **GitHub** codebase against 10 software (as can be seen in figure 9) engineering guidelines(see figure 6),including code quality, code architecture and tests. The results of the analysis was taken into account when refactoring code but was not considered the highest authority of the quality of the codebase. The end-result of **7/10** was according to the program (see figure 10) due to a lack of **separation of concern**, **high coupling** and lack of **test automation**. The lack of test automation was no news as the code is rather difficult to test, at least concerning code coverage, due to most of it being structural objects holding information. The **"high coheshion, high coupling"** however did not feel applicable for the most part as the code was designed with the design principle **"High cohesion, low coupling"** in mind.

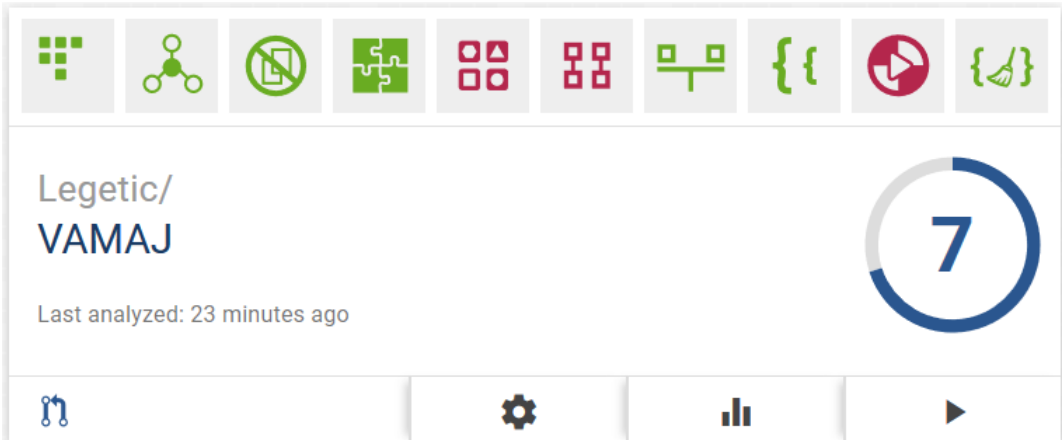

Figure 9: The overall result given by **bettercodehub**

.png

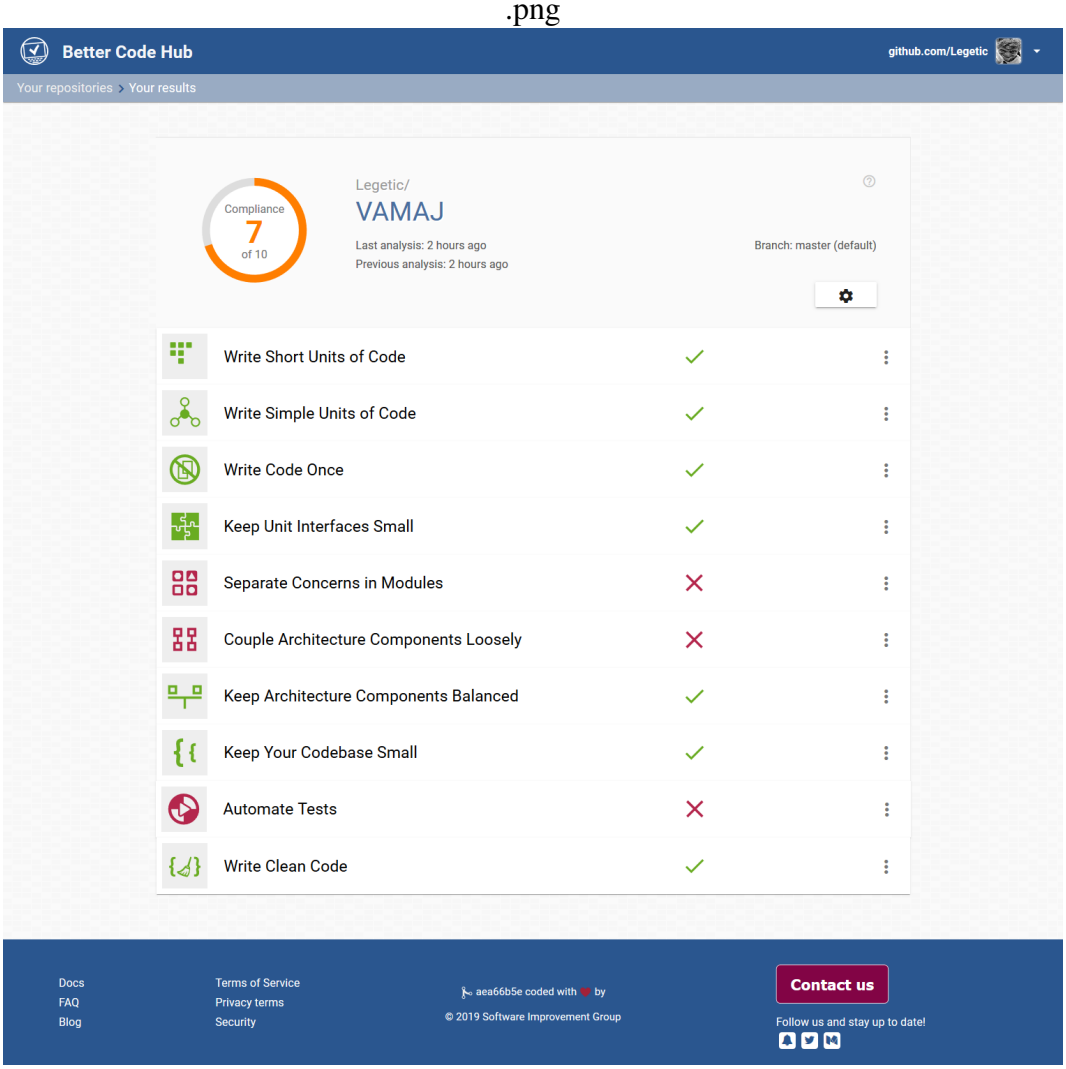

Figure 10: Code guidelines checked by **bettercodehub**

7 Peer review

The following section is the peer-review of group-15's code (Pimp). The review follow a specific structure where code design, structure, oop aspects are reviewed.

7.1 Does the project use a consistent coding style?

Yes. code chunks are well-organised. The code style is consistent. The comments are found above the code. No redundant methods or spaces that pollute the styling.

7.2 Is the code reusable?

One example of where the code is not reusable is the interface "IModel" which is an interface with 25+ methods. This strongly goes against Interface segregation principle and makes it difficult to reuse the interface (which should be one of its main purposes). There are other examples of monster interfaces. Single responsibility principle also seems to be violated in multiple places such as "ModelImpl" which as a result of implementing the "IModel" interface has over 25 methods. This makes it difficult to reuse methods and functionality in general.

7.3 Is it easy to maintain?

The difficulty of maintaining the code varies throughout the program This is based on the fact that the code is overcomplicated in some areas, especially where there are large or unnecessary interfaces. However there are some areas, where the code structure is easier to understand (example: command package), where it is easy to maintain/add new code.
Nevertheless, the code has high testing coverage, is well documented and follows oop principles. However, violations of the SOLID principles can be found, especially ISP and OCP.

7.4 Can we easily add/remove functionality?

- **OCP?**
  The extended usage and implementation of factory pattern makes it easier to add new functionality, as new factory methods can be added to create new objects, without having to change the code in too many places. In some places,

there are large interfaces that can be difficult to reuse, as you have to also change all the classes that are implementing that interface.

- **SOC?**
  The packages are well separated, that is, each package is well encapsulated and has a well defined purpose. Module-pattern is well implemented in all areas of the program, which contributes to encapsulation and SOC.

- **Is the code loosely coupled?**
  From analyzing the UML-diagram, it seems that the code is on the contrary highly coupled. The code should bear in mind the KISS principle. Keep It Simple Stupid. Don't make the code more complicated that is should be. We feel that there are too many interfaces, sometimes interfaces that are too large (25+ methods).

7.5 Are design patterns used??

As it is listed in the provided SDD, the application implements a handful of design patterns.

7.6 Is the code documented?

The code is well documented using javadoc.

7.7 Are proper names used?

The naming is mostly fine but there are some examples of poor naming such as: "AbstractComposite" which does not practically mean anything. Their interfaces use the "I" prefix, ex. ICopiable. Specifying the type in the name is unnecessary.

7.8 Is the design modular? Are there any unnecessary dependencies?

"The benefit of adhering to the ISP is that it keeps the code decoupled. This ultimately makes the code base easier to change, troubleshoot, and maintain."[6] The code is highly separated into independent modules. Almost every package has a public interface to hide the internal details of the package. Modularity also means that the code is easy to maintain, refactor and test. This may be challenging due to

the code having big and “bloated” interfaces that does a lot, also ISP is followed poorly.

7.9 Does the code use proper abstractions?

- The usage of interfaces sometimes seem somewhat poorly motivated or thought through. For instance: the “IModel” interface is extremely large which goes against the “Interface segregation Principle.”Another example is IColor, what is the usage of an interface here? Are they planning on implementing different types of Colors that work in other ways?
- The command package is well written, as it is easy to remove/add new commands and incorporate them into the design without having to rewrite a lot of code.

7.10 Is the code well tested?

Yes. There are plenty of tests (especially for model package) that covers the most essential parts of the code.

7.11 Are there any security problems, are there any performance issues?

- The program is rather sluggish, this is most likely due to the creation and management of many objects (for example the pencil “dragged” method). Perhaps try to solve this differently with for example “Object-Pooling”[7].
- It seems that the code the code is mostly well encapsulated. It also seems that thought has been put into pursuing immutability (for example “AbstractCommand”) which is positive as it mitigates the security risks concerning alias problems and unexpected changes and access.

7.12 Is the code easy to understand? Does it have an MVC structure, and is the model isolated from the other parts?

- Despite the fact that the code in general uses design patterns and good practices extensively the code is still extremely difficult to understand. This could

be caused by over complication in some areas and poor structure. Furthermore the provided UML diagram does not help to understand the code as it is very complex. These two facts most likely correlate and it is very difficult to see that the program was designed before it was implemented (It is unlikely for someone to design a structure that complex and unreadable).

- It seems that the code makes use of a MVC pattern where the Model notifies the view through an observer-pattern. This makes it so that the model communicates with the view without depending on it.

7.13 Can the design or code be improved? Are there better solutions?

- The design largely follows the SOLID-principles. The biggest violation of SOLID is the Interface segregation principle, since there are huge interfaces that don't have a clear single purpose.
- The code is well commented and follows a common style. Some of the naming of variables and methods can be improved for clarification.
- Naming of packages and classes is at times unclear, the name should clearly reflect what the package or class is.
- The design model should be reviewed thoroughly and refactoring of mayor parts should be considered.

References

[1] Figma, “Mockup tool,” 2019. [Online]. Tillgänglig: https://www.figma.com Hämtad: 2019-10-07.

[2] NASA, “Power project data sets.” 2019-10-07. [Online]. Tillgänglig: https://power.larc.nasa.gov/ Hämtad: 2019-10-07.

[3] SourceMaking, “Template method design pattern.” 2019-10-07. [Online]. Tillgänglig: https://sourcemaking.com/design_patterns/template_method Hämtad: 2019-10-07.

[4] JUNIT, “The new major version of the programmer-friendly testing framework for java.” 2019-10-20. [Online]. Tillgänglig: https://junit.org/junit5/ Hämtad: 2019-10-20.

[5] S. I. Group, “Write better code with a definition of done..” 2019-10-20. [Online]. Tillgänglig: https://bettercodehub.com Hämtad: 2019-10-20.

[6] A. Vathanakamsang, “Interface segregation principle,” 2019. [Online]. Tillgänglig: https://medium.com/@a.vathanaka/interface-segregation-principle-5f8a2014de2c Hämtad: 2019-10-22.

[7] saketkumr, “Object pool design pattern,” 2019. [Online]. Tillgänglig: https://www.geeksforgeeks.org/object-pool-design-pattern/ Hämtad: 2019-10-22.

Appendices

Developer-related user stories

.0.1 Story identifier : Development environment

Description:
As a group of developers, we want to have a well-functioning developing environment with all needed tools installed and ready, so we can do our job.

Confirmation:
[1] All members of the group have functioning development tools (Git, Maven, Junit, SceneBuilder och Intellij)

.0.2 Story identifier : Basic functionality

Description:
As a Developer I want to be able to see a visual representation of a rudimentary version of the program aswell as have a good model structure of the program with basic functionality so that I can get a clearer view of the next development steps and so that I know where things should go.

Confirmation:
- There is a simple GUI that represents the information input in the program.
- The classes specified in the design model have been created. and associated with eachother appropriately.
- Det finns statiska defaultvärden för kalkylationen.
- Det finns en simpel ekonomisk algoritm som beräknar mellanskillnad.

.0.3 UserStory3.1 :

Description:
As a developer, I want to create a service module to communicate with different API:s, in order to gather data for my domain model and calculations.

Time estimation: 2h-12h.

Confirmation:

- We have (service) classes that can create model objects based on data from the API:s.

- We have created interfaces that allow us to add new API:s, without disturbing other parts of the program.

.0.4 UserStory3.2 :

Description:
As a developer I want the way that information travels into the model through user input is consistent with how it travels from API:s so that the code structure is logical.

3h

Confirmation:
- The controller forwards input to the model that creates the objects.

- Immutability is used as much as possible.

.0.5 UserStory13 :

Story name:
Description:
As a developer I would like to see how the functionality of the GUI should work so that I know how to implement my features.

3h.

Confirmation:
- Placeholder features are implemented in the GUI .