Chalmers University of Technology

Objektorienterat programmeringsprojekt

System design document for EnergyCalculator

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

VAMAJ

Contents

1 Introduction

The application is primarily a calculator of different aspects of solar energy for consumers. The user can through the application get detailed results on efficiency and economical profit variables depending on their prerequisite data.

This document is a specification of the program and its technicalities. The following chapters will detail different aspects of the program in order to provide an overview of the actual structure och functionality of the program.

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

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 considered 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.

API ”Application Programming Interface”, an interface or communication protocol between a client and a server intended to simplify the building of client-side software.

Java an independent object-oriented 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 separate the different areas of code(logic, viewing,...)

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.

SOLID A set of design principles, defining broad aspects of good object oriented design.

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

JSON JavaScript Object Notation

2 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.

2.1 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.

2.2 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.

2.3 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.

2.4 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.

3 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 1). 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.

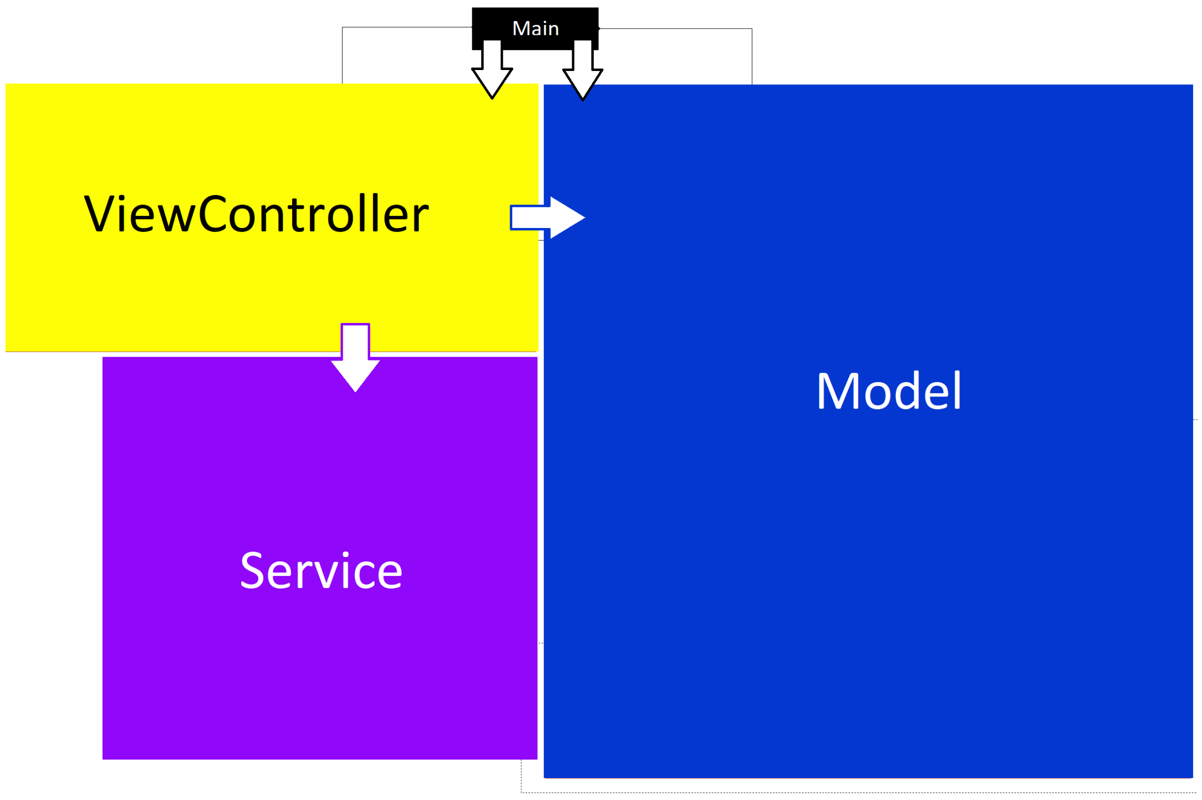


Figure 1: Blocks representing the design model of the program, a clear MVC pattern with a service package

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 2).

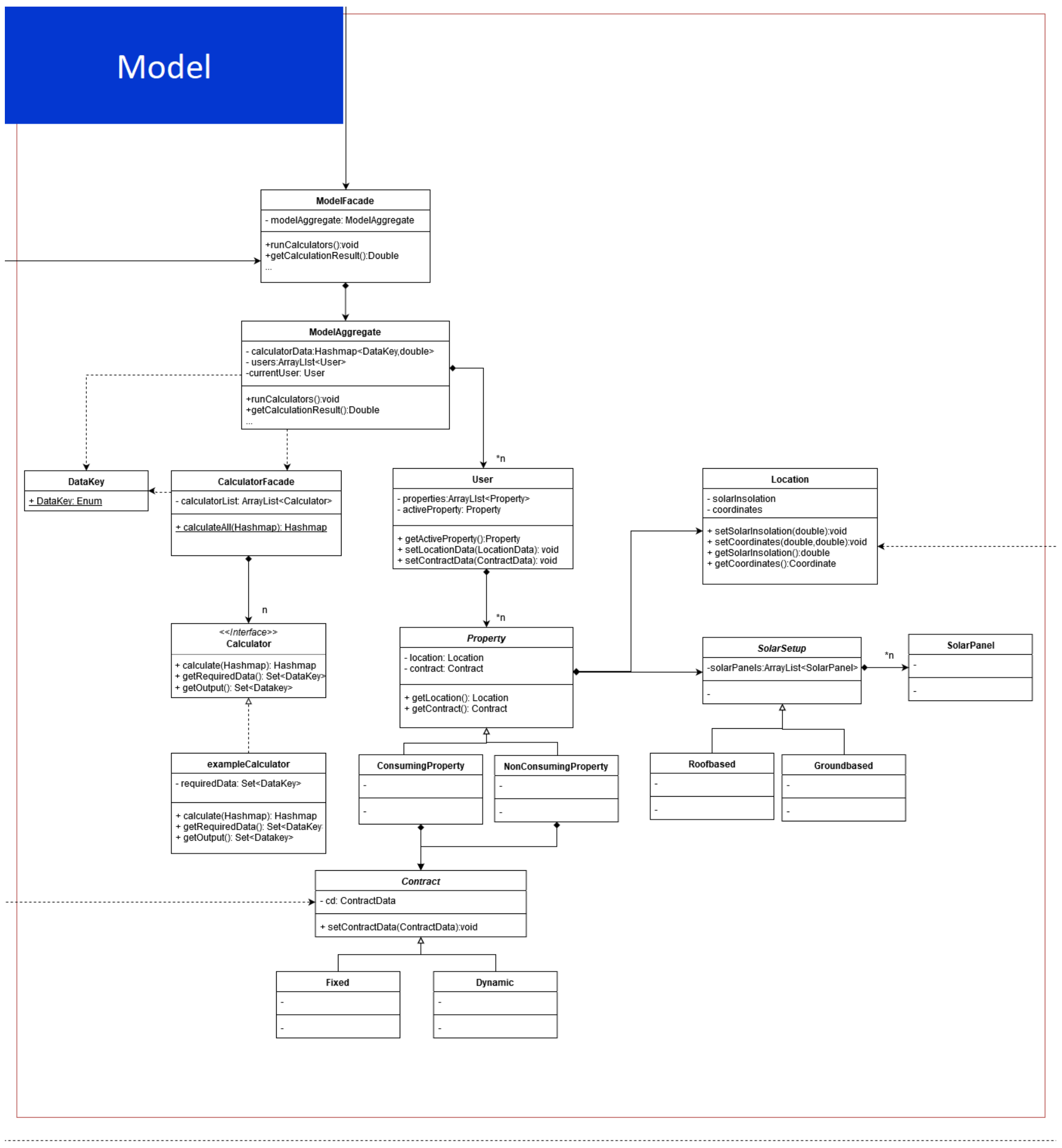

Model
ModelFacade
- modelAggregate: ModelAggregate
+runCalculators():void
+getCalculationResult():Double
...
ModelAggregate
- calculatorData:Hashmap<DataKey,double>
- users:ArrayList<User>
-currentUser: User
+runCalculators():void
+getCalculationResult():Double
...
DataKey
+ DataKey: Enum
CalculatorFacade
- calculatorList: ArrayList<Calculator>
+ calculateAll(Hashmap): Hashmap
n
<<Interface>>
Calculator
+ calculate(Hashmap): Hashmap
+ getRequiredData(): Set<DataKey>
+ getOutput(): Set<Datakey>
exampleCalculator
- requiredData: Set<DataKey>
+ calculate(Hashmap): Hashmap
+ getRequiredData(): Set<DataKey>
+ getOutput(): Set<Datakey>
*n
User
- properties:ArrayList<Property>
- activeProperty: Property
+ getActiveProperty():Property
+ setLocationData(LocationData): void
+ setContractData(ContractData): void
*n
Property
- location: Location
- contract: Contract
+ getLocation(): Location
+ getContract(): Contract
ConsumingProperty
NonConsumingProperty
Contract
- cd: ContractData
+ setContractData(ContractData):void
Fixed
Dynamic
Location
- solarInsolation
- coordinates
+ setSolarInsolation(double):void
+ setCoordinates(double,double):void
+ getSolarInsolation():double
+ getCoordinates():Coordinate
SolarSetup
-solarPanels:ArrayList<SolarPanel>
*n
SolarPanel
Roofbased
Groundbased

Figure 2: 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 3). 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 [1] 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 **”ApiParser”** 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** [2] so that new types of parsers can be easily added without needing to duplicate code.

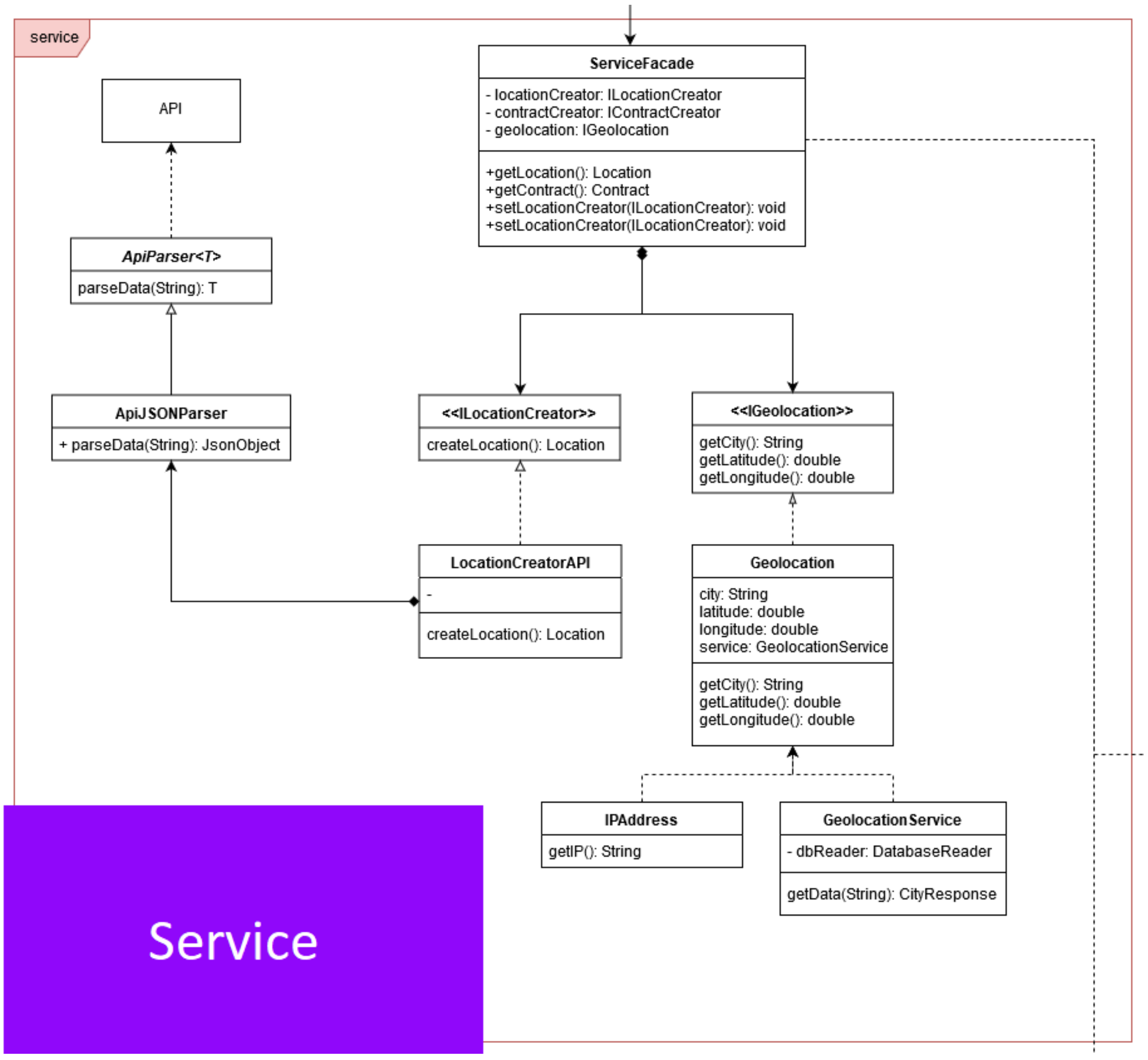


Figure 3: The service block

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 4).

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(...);
```

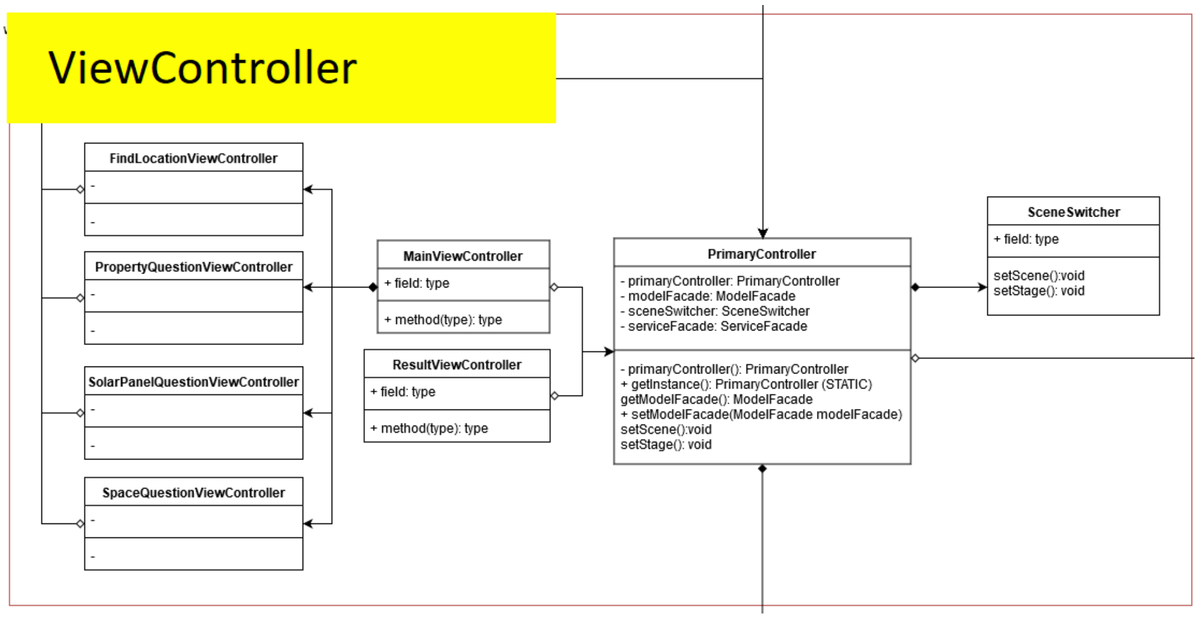

ViewController
FindLocationViewController
PropertyQuestionViewController
SolarPanelQuestionViewController
SpaceQuestionViewController
MainViewController
+ field: type
+ method(type): type
ResultViewController
+ field: type
+ method(type): type
PrimaryController
- primaryController: PrimaryController
- modelFacade: ModelFacade
- sceneSwitcher: SceneSwitcher
- serviceFacade: ServiceFacade
- primaryController(): PrimaryController
+ getInstance(): PrimaryController (STATIC)
getModelFacade(): ModelFacade
+ setModelFacade(ModelFacade modelFacade)
setScene():void
setStage(): void
SceneSwitcher
+ field: type
setScene():void
setStage(): void

Figure 4: The view-controller block

4 Quality

Tests are performed using **JUNIT**[3] 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"**[4]. This tool checks the **GitHub** codebase against

10 software (as can be seen in figure 5) 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 6) 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.

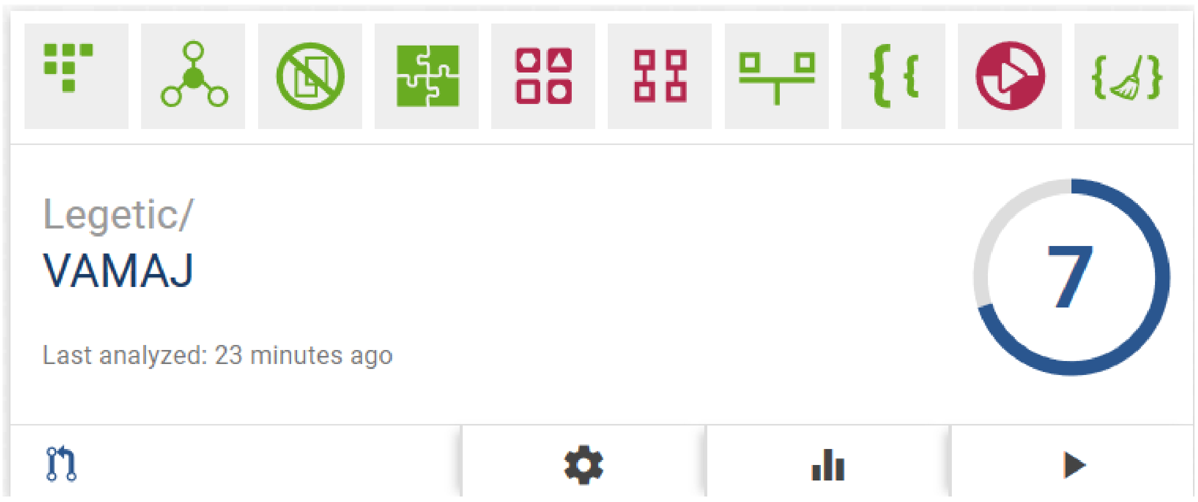
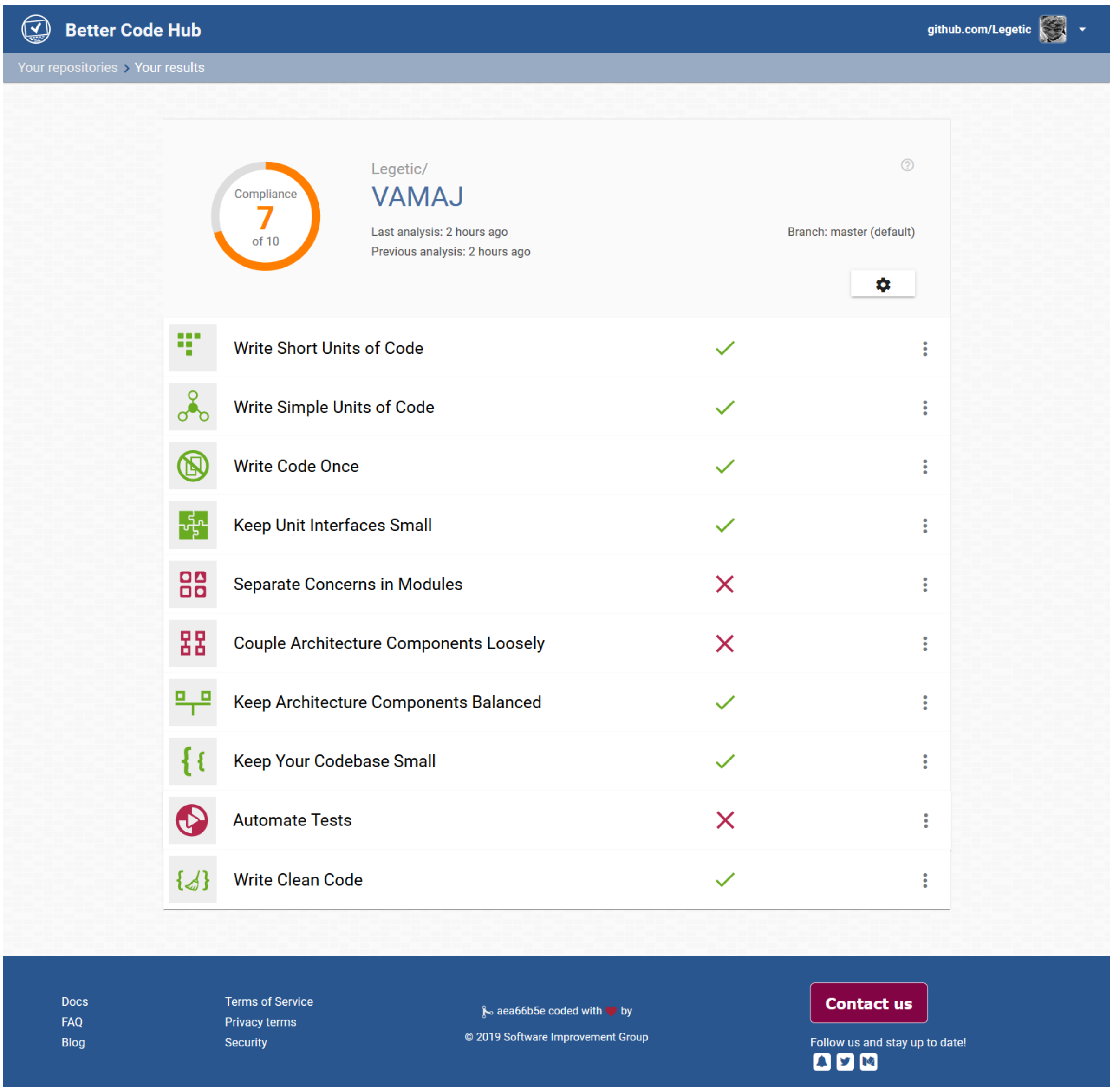


Figure 5: The overall result given by **bettercodehub**



Better Code Hub

github.com/Legetic

Your repositories > Your results

Compliance 7 of 10

Legetic/
VAMAJ

Last analysis: 2 hours ago
Previous analysis: 2 hours ago

Branch: master (default)

Write Short Units of Code	✓
Write Simple Units of Code	✓
Write Code Once	✓
Keep Unit Interfaces Small	✓
Separate Concerns in Modules	✗
Couple Architecture Components Loosely	✗
Keep Architecture Components Balanced	✓
Keep Your Codebase Small	✓
Automate Tests	✗
Write Clean Code	✓

Docs
FAQ
Blog

Terms of Service
Privacy terms
Security

aea66b5e coded with by

© 2019 Software Improvement Group

Contact us

Follow us and stay up to date!

Figure 6: Code guidelines checked by **bettercodehub**

References

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

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

[3] 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.

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