UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Taavi Luik

Designing a Pharmacogenetic Test as a Medical Software Device

Master's Thesis (30 ECTS)

Supervisor(s): Sulev Reisberg, PhD
Kersti Jääger, PhD

Tartu 2021

Designing a Pharmacogenetic Test as a Medical Software Device

Abstract:

Most people have genetic mutations which influence the efficacy of drugs and when taken into account, could improve health response. A pharmacogenetic test performed on the patient's genetic data is a possible solution. Although the test merely provides a software-based analysis of already existing genetic data, the test still classifies as a medical software device. This thesis gives an overview of pharmacogenetics, software-based medical devices, formulates requirements for such a device, provides a high level design, verifies the design against requirements, and discusses the results and perspectives.

Keywords:

Pharmacogenetics, software design, medical device

CERCS: P170 Computer science, numerical analysis, systems, control

Farmakogeneetilise testi kui tarkvaralise meditsiiniseadme disainimine

Lühikokkuvõte:

Enamikel inimestel esineb geenimutatsioone, mis mõjutavad ravimite toimet ning mille arvesse võtmisel paraneks individuaalne ravivastus. Patsiendi geeniandmete põhjal teostatud farmakogeneetiline test oleks üks võimalik lahendus. Kuigi vastav tarkvaraline test vaid analüüsib juba olemasolevaid geeniandmeid, liigitub see tarkvaraliseks meditsiiniseadmeks. Käesolevas töös antakse ülevaade farmakogeneetikast, tarkvaralisest meditsiiniseadmest, tuvastatakse sellise farmakogeneetilise testi seadme nõuded, pakutakse välja seadme kõrgtaseme disain, verifitseeritakse disaini nõuete vastu, ning arutletakse tulemuste üle.

Võtmesõnad:

Farmakogeneetika, tarkvara disain, meditsiiniseade

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1 Introduction

A significant part of interpatient variability in drug response is caused by genetic differences. It is estimated that most people have at least one genetic mutation that warrants a change in drug choice or dosage in order to ameliorate adverse drug reactions or to increase drug efficacy. The Estonian Biobank has genetic data available for approximately 200,000 gene donors for whom gene variant-directed pharmacogenetic suggestions could be given. The large number of donors and known genetic mutations, and the incompleteness of available genetic data calls for a software solution that would handle variability in data, be scalable and safe.

The purpose of this thesis is to help fill this gap by identifying and analyzing software requirements in pharmacogenetic test development, and by providing and verifying a high-level software design.

The thesis has the following structure: first, an overview of important terms is given (Section 2). Next, necessary background is explained (Section 3). This is followed by a description of methodology (Section 4) and a list of software requirements (Section 5). The software design is presented (Section 6), verified (Section 7), and discussed (Section 8), followed by concluding remarks (Section 9).

2 Terms and Notations

Allele – an alternative version of a gene inherited from one of the parents

Diplotype – a pair of haplotypes

Genetic variant – see Allele

Genotype – an organism's complete set of genes

Haplotype – a sequential block of DNA inherited from one of the parents

Nucleotide – a building block of DNA, made from a base (adenine, thymine, cytosine or guanine) and a phosphate molecule and a sugar molecule

Pharmacogenetics (PGx) – the study of genetic variability in drug response

Phased genetic data – genetic data separated into two haplotypes using reference genomes

Phenotype – observable traits (e.g. protein function) partly determined by genotype

Reference genome – a complete genome sequence that genetic variability is compared against

SNV (single nucleotide variant) or SNP (single nucleotide polymorphism) – genetic variant defined by a change in a single nucleotide

Star allele – a name used in the pharmacogenetics nomenclature system of an allele that combines one or more genetic changes that inherit together

3 Background

This chapter gives a short introduction to basic genetics, pharmacogenetics and pharmacogenetic testing. It is also explained how a pharmacogenetic test can be entirely software based, and how to implement the test in a clinical setting.

3.1 Basics of Genetics

Our traits are encoded by genes, which are formed by an ordered sequence of four DNA bases: adenine (usually abbreviated as A), thymine (T), cytosine (C), and guanine (G). Genes are segments of DNA that act as instructions to make proteins. We have two copies of genes, inherited from each parent. These different forms of a gene are called alleles and they contribute to each person's unique physical features.

DNA is subject to polymorphisms. The most common polymorphism affects a single DNA base and is therefore called a SNP (single nucleotide polymorphism) or SNV (single nucleotide variant), but small insertions and deletions (up to full gene deletions), multiple gene copy variants also exist. All types of variation can lead to changes in the function or activity of the protein that the gene encodes. Variants can occur in the protein coding regions (exons) or non-coding regions (introns and regulatory elements) of a gene. Exonic variants may change protein structure through amino acid substitution, introduction of an early stop codon, creation of an alternative splice variant (i.e. including or excluding some exons) or shifting the amino acid reading frame. Even synonymous variants, i.e. where the same triplet of nucleotides code for the same amino acid (e.g. AAC and AAT both result in asparagine), may lead to changes in protein activity. Gene copy number variations do not change the amino acid sequence, but generally increase protein activity, while gene deletions mean that no gene is transcribed and there will be no protein activity (Deenen et al., 2011).

A reference genome is used to show changes in a genome compared to an average genome. Each nucleotide in a reference genome has a specific position. A special identification number is given for common variants, e.g. position 94775367 on chromosome 10 can also be expressed as rs12769205. Structural variants could be shown, for example, as GTTTT>GTT or I>D (from *insertion* and *deletion*), depending on the technology used (Reisberg, 2019).

3.2 Personalized Medicine and Pharmacogenetics

The term *personalized medicine* (or *stratified medicine, precision medicine, P4 medicine)* refers to the application of genomics and molecular biosciences in medical care (Pokorska-Bocci et al., 2014). It takes advantage of the uniqueness of people encoded by their genes and tries to convert genetic differences into more efficient therapies.

Pharmacogenetics (PGx) – a form of personalised approach in medicine – can be defined as the study of heritable variability in drug response (Nebert, 1999). Specific genetic mutations can alter, for example, how certain drugs are absorbed, distributed, metabolized, or eliminated (Relling et al., 2015), and this knowledge can be used to facilitate a more targeted

prescribing of treatment in order to increase efficacy and safety (Lonergan et al., 2017). For many gene-drug pairs, the evidence of exact interaction mechanisms is still weak. However, the evidence for several gene-drug pairs is sufficient for practical clinical implementation, even if these drugs constitute a minority of all prescribed drugs (Dunnenberger et al., 2015).

The Clinical Pharmacogenetics Implementation Consortium (CPIC) maintains a list[1] of gene-drug pairs with available evidence levels for actionability. Of the total 440 known associations, 68 have been assigned A level evidence, which means that prescribing action is recommended and is highly likely to be effective and safe[2]. Currently, a total of 84 drugs have published guidelines[3]. Most of these guidelines use standardized phenotype nomenclature in order to make it more easily interpretable by clinicians (Caudle et al., 2017) and contain tables that map a diplotype to a phenotype, and a phenotype to a therapy recommendation. In order to use established guidelines, genomic variants must be extracted and possible diplotypes assigned based on genetic data (Klein et al., 2018).

CPIC collaborates with PharmGKB, a pharmacogenomics knowledge resource, which collects and curates lists of gene variants and their effects, using the star (*) allele nomenclature (combined into an *allele definition table* for a gene)[4]. A star allele has a name, consists of one or more SNVs that are inherited together in a population, and is associated with a certain change in protein function.

Often, star alleles are characterised by more than one SNV, even if only a single variant is known to result in change of function. This can become a problem if dealing with missing stretches of DNA sequence or position information - there might be enough data to infer a change in function, but not enough data to assign the corresponding allele, if perfect allele matches are searched for (Reisberg et al., 2019).

Moreover, in order to determine a pharmacogenetic phenotype, both alleles (maternal and paternal) have to be detected in the same individual. Combining variant information from two alleles results in a diplotype designation that can be associated with overall protein function in that individual. For example, one pharmacogene CYP2C19 forms an actionable gene-drug pair with Citalopram. Citalopram is one of the primary treatment options for major depressive and anxiety disorders. It is advised to consider an alternative drug for patients with at least one CYP2C19 increased function allele (due to decrease in efficacy) or two loss-of-function alleles (due to increased risk of side effects), one on each haplotype (Hicks et al., 2015). Reisberg et al. (2019) found that out of 9977 Estonians with a prescription of at least one drug linked to CYP2C19, 40.7% (n=4059) need dosing adjustment due to being either CYP2C19 poor, rapid, or ultrarapid metabolizers.

[1] https://cpicpgx.org/genes-drugs/ (23.03.2021)
[2] https://cpicpgx.org/prioritization/ (23.03.2021)
[3] https://cpicpgx.org/prioritization-of-cpic-guidelines/ (23.03.2021)
[4] https://www.pharmgkb.org/about (31.03.2021)

Multiple studies have estimated that the vast majority (>98.5%) of people need some dosage adjustment for at least one drug (Reisberg et al., 2019; Dunnenberger et al., 2015). One study examining CYP2C19 alleles among 2.3 million primarily European participants found that 30.4% have one or two increased function alleles (*ultrarapid metabolizers*) and 2.6% have two loss-of-function alleles (*poor metabolizers*) (Ionova et al., 2020). This study did not examine 6 other rare yet known CYP2C19 loss-of-function alleles (*4, *5, *6, *7, *8, *35) (Pratt et al., 2015), meaning that the true number of poor metabolizers can even be higher. The problem of not interrogating rare variants is common and it has been estimated that rare genetic variants are likely to account for a substantial part of unexplained differences in drug metabolism phenotypes for at least some drugs (Ingelman-Sundberg et al., 2018).

3.3 Software-based PGx Testing

Most pharmacogenetic tests currently available focus on the analysis of a single gene or a limited set of SNVs, and are performed only on demand. This type of testing has the drawbacks of being expensive, slow and requiring the doctor to have prior knowledge about the test specifics (Dunnenberger et al., 2015). DNA sequencing is becoming cheaper and various biobanks already contain genetic data for many people (Reisberg, 2019). With minor exceptions, DNA does not change over the course of one's lifetime and hence full-genome sequencing has to be done only once in life. In fact, full-genome sequencing is not necessarily required – genotyping arrays, which detect only specific nucleotides in the genome, make it possible to extract pharmacogenetic information by even faster and cheaper means (Reisberg, 2019; Dunnenberger et al., 2015). The bottleneck in pharmacogenetic phenotyping has moved from data generation to data analysis and interpretation (Alyass et al., 2015).

Raw genetic data shows merely the values of nucleotide bases at certain genomic positions, and is not useful on its own. To benefit from the genetic data collected in biobanks, research and clinical evidence for variant-drug effects must be available, be structured and kept up-to-date in the form of a knowledge base. There is also a need for an algorithm to combine the two sources of information, since genetic data is often incomplete and novel combinations of known mutations exist – guidelines do not state how to adjust dosage if multiple alleles exist on the same haplotype. The algorithm used in this software is derived from Reisberg et al (2019), a similar approach is also used by McInnes et al. (2020). These three components – genetic data, knowledge base and an algorithm – constitute the backbone of a PGx test. Since new data and knowledge are on the rise, software-based solutions are much needed to complement the more targeted laboratory tests.

Existing software-based pharmacogenetic solutions are either proprietary (Sabater et al., 2019), intended only for research (i.e. non-clinical) purposes (Klein et al., 2018; McInnes et al., 2020), operate on different types of data (Twesigomwe et al., 2021; Numanagić et al., 2018), or are specialized on specific genes (Twist et al., 2016).

3.4 GenMed Project

The GenMed project is a national initiative in personalized medicine that aims to build an IT infrastructure in Estonia that, through the national e-health system, would enable applying pharmacogenetics and polygenic risk score based tests in clinical practice[5].

As part of the GenMed project and pharmacogenetic test development, a knowledge base has been created that includes a curated list of allele-drug associations taken from CPIC and PharmGKB with a higher clinical evidence threshold. Additionally, variants that do not define a scientifically proven function are excluded from the knowledge base, which allows a more clinically sound and predictable pharmacogenetic testing.

One of the goals of this project is to create a software tool that uses a person's genetic data and the knowledge base to perform multiple pharmacogenetic tests (each targeting a specific gene) and outputs a pharmacogenetic phenotype upon each test. The software will not directly output drug recommendations – instead, these will be available through a separate registry. An IT infrastructure sets additional limits for the software development – it is intended to work as a command-line tool without a graphical user interface, and will be run in a Docker container. A simplified diagram showing the placement of the software within the surrounding infrastructure is shown in Figure 1.

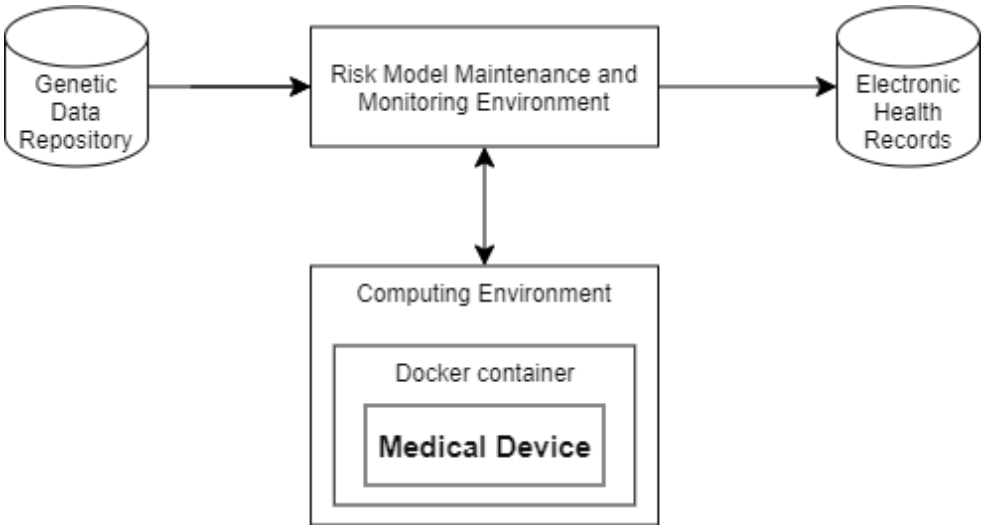

Figure 1. Position of the software, denoted as a 'Medical Device', in the surrounding IT infrastructure. The software is located in a Docker container in a computing environment (The High Performance Computing Center of University of Tartu), is run by a Risk Model Maintenance and Monitoring Environment, that receives genetic data from a data repository and passes the results into Electronic Health Records.

[5] https://sisu.ut.ee/genmed/en?lang=en (31.03.2021)

3.5 Medical Device

Bringing PGx testing to clinical usage requires complying with regulatory requirements and international standards, which have been developed due to accidents stemming from medical devices with bad software design and development practices (Coronato, 2018: 7).

According to European Union regulation on *in vitro* diagnostic medical devices (Regulation 2017/746), this software falls under *in vitro* diagnostic medical devices, since it predicts treatment response or reactions (Article 2, section 2e). The regulation mandates following the software development life cycle (Annex I, section 16).

The standard EN 62304:2006+A1:2015 'Medical device software - Software life-cycle processes' provides a framework of the life-cycle of a medical device with the end goal of safe design and maintenance. The life-cycle consists of various processes, each with inputs, activities (each consisting of interrelated tasks), and outputs. The main processes of the life-cycle are software development planning, requirements analysis, architectural design, detailed design, unit implementation, integration and integration testing, system testing, and software release. This thesis focuses on the requirements analysis and architectural design stages.

A software system consists of software items and software units. The system must be assigned a software safety class, which can be class A, B, or C, depending on the risk of harm that can stem from a hazardous situation, where the software can contribute. The higher the class, the more risk control measures may have to be implemented. The device in question belongs to class C. It is important to note that 'safety' refers to freedom from unacceptable risk, not any risk.

The standard requires a risk management process, achieved via complying with ISO 14971. The standard ISO 14971:2019 'Medical devices - Application of risk management to medical devices' is intended to establish a risk-management process that identifies hazards, assesses related risks and implements risk control measures (Coronato, 2018: 69). A hazard is defined as a potential source of injury or damage to the health of people, or damage to property or the environment.

A complete application and documentation of these standards is outside the scope of this thesis. However, the software design must have appropriate units and mechanisms that identify and react to hazards. This includes input data validation, knowledge base validation, program flow branches, etc.

A medical device incorporating software must be verified and validated (Regulation 2017/746). Verification refers to tasks that demonstrate that the system 'does the things right', whereas validation refers to tasks that demonstrate that the system 'does the right things' (Coronato, 2018: 13). Validation deals with evaluating if the software meets the intended use (Coronato, 2018: 109) and this becomes meaningful when looking at the system as a whole (e.g. with a functional knowledge base), and cannot be adequately done merely based on a high level design, and hence falls outside the scope of this thesis.

One of the best tools to do both verification is testing (Coronato, 2018: 109), yet the testing methods available for a high level design are limited. For example, dynamic testing techniques require a compiled software which is executed with specific input values and internal conditions, which are then compared with expected results (Coronato, 2018: 131).

To satisfy functional requirements, a sequence of responsibilities must be assigned throughout the design (Bass et al., 2012: 64). Therefore, one static testing method that can be employed is assessing whether each requirement has modules or components which are responsible for that functionality.

3.6 Aim of the Thesis

The goal of this thesis is to provide a high level software design for the PGx testing medical device that would meet the identified requirements. The design is presented in the form of structural and behavioral diagrams, the former showing modules and inner-module components, and the latter showing sequences of tasks inside each component.

4 Methodology

Ideally, a final or near-final list of requirements is used as an input for the architectural design process. However, often the final list of requirements is not available. Although some requirements for the software described in this thesis were predefined in various documents, a complete list of requirements was lacking. Therefore, the author had to first identify and formulate the requirements.

The requirements for this software came from three main sources – user needs, surrounding IT infrastructure, and international standards for medical devices. As part of the GenMed project, doctors were interviewed in order to understand their expectations for software-based pharmacogenetic testing. Some requirements state how the device integrates into existing and in-development IT infrastructure. Finally, many requirements came from medical device regulations.

The author analyzed these sources and compiled a list of requirements that affect the architectural design of the PGx test. A review is a common requirement validation technique (Bourque et al., 2014), therefore the list was reviewed by three other team members in the GenMed project working on pharmacogenetic test development. Adjustments were made based on the feedback. These requirements reflect a preliminary understanding and are subject to change through validation and verification tasks.

The test design process was iterative – after a set of requirements were specified, requirements were mapped to processes, which were grouped and assigned to components, which in turn were grouped to modules. Once modules were created, inter-module requirements were specified and previous modules improved to guarantee or identify the failing of meeting these requirements.

The author created high level architecture, identified main processes, and described each process in more detail. Since the processes exchange data, the data model is also described. Lower level details are left undefined, since some requirements are unspecified or subject to change. Deferring design decisions is not necessarily bad, since the longer one can delay decisions, the more information one will have to make these decisions properly (Martin, 2018: 141).

In order to avoid ‘ivory tower’ designs that are impractical or difficult to implement, the author created prototypes in the Python programming language, which implemented a design skeleton – modules and components, main program flow and exception handling – without providing a detailed implementation. The diagrams were created with diagrams.net desktop client software[6].

[6] https://github.com/jgraph/drawio-desktop/releases/tag/v14.5.1

5 Software Requirements

The following requirements are classified according to the schema provided in section 5.2.2 (Software requirements content) of IEC/EN 62304 Medical device software - Software life-cycle processes:

1. **Functional and capability requirements**
   - 1.1. The device must validate input data based on the requirements set by the device.
   - 1.2. The device must work for a short period, processes the genetic data for a single person, and terminate.
   - 1.3. The device must perform multiple independent PGx tests within a single run, each of which results in a phenotype:
     - 1.3.1. Performing a single PGx test shall consist of two main steps: assigning star alleles to both haplotypes, and assigning a phenotype.
     - 1.3.2. A star allele shall be assigned for a haplotype if all SNVs defining the star allele are provided in the data and the alleles match.
     - 1.3.3. If a loss-of-function star allele match is found for a haplotype, only that star allele must be assigned; otherwise all matching star alleles must be assigned.
2. **Performance requirements**
   - 2.1. The device must finish its work or stop after a timeout period, in which case it must not output any test results. The device must notify the user that this has happened.
3. **Software system inputs and outputs**
   - 3.1. Both input and output must be in JSON format, further specified within project documentation.
   - 3.2. The input genetic data:
     - 3.2.1. Must contain a specific set of variants
     - 3.2.2. Must be phased
     - 3.2.3. Must be aligned to the GRCh37.p13 reference genome
     - 3.2.4. Need not include all genetic variants used in the PGx tests
   - 3.3. The output consists of three parts: device metadata, results of tests, errors
     - 3.3.1. Metadata must contain software version number
     - 3.3.2. For each successful test, the output must contain:
       - 3.3.2.1. Name of the test
       - 3.3.2.2. Result (phenotype)
       - 3.3.2.3. Unit (always “NA”)
       - 3.3.2.4. Reference value (always “NA”)
       - 3.3.2.5. Additional textual info for the result
     - 3.3.3. Error messages must show clearly the reason of failure
4. **Interfaces between the software system and other systems**
   - 4.1. The device must use the interface specified by the Computing Environment, further specified within project documentation.
     - 4.1.1. Input must be received through the *standard input* (*stdin*) stream
     - 4.1.2. Output must be sent to the *standard output* (*stdout*) stream

4.1.3. Logs must be output to the *standard error* (*stderr*) stream

5. Software-driven alarms, warnings, and operator messages

5.1. The device output must include an error message in case an error has occurred

5.2. The device shall not output any test results in the following situations:

5.2.1. Timeout period is reached

5.2.2. Any error is encountered while parsing and validating input data

5.2.3. Any error is encountered while loading and validating the knowledge base

5.3. Tests that fail due to lack of available data must have a clear indication in the results of the test

5.4. All identified problems must be logged as a warning level message.

6. Security requirements

6.1. The device shall not provide any other interface to the user except as specified in requirement 4 (*Interfaces between the software system and other systems*)

6.2. The device shall not output any raw genetic data it received as input

7. User interface requirements implemented by software

7.1. The device shall not have a graphical user interface

8. Data definition and database requirements

8.1. The Knowledge Base must be distinct from other functionality of the device in order to be updatable by the manufacturer

8.2. The Knowledge Base must be in the format of text files

8.3. Each PGx test must be represented as a separate object that defines star alleles, their function, variants, and a diplotype-phenotype mapping

9. Installation and acceptance requirements of the delivered Medical Device Software at the operation and maintenance site or sites

9.1. The device must be able to perform a self-test:

9.1.1. The self-test must consist of end-to-end tests that simulate real use scenarios

9.1.2. Failure to pass the self-test must be clearly indicated in the output

10. Requirements related to IT-network aspects

10.1. The device shall not need nor use a network connection

10.2. The device shall not make changes to the file system (except for logging) nor depend on file system changes made in previous runs

11. User maintenance requirements

11.1. The device must provide the user with the ability to perform a self-test to verify that the device is working as intended

12. Regulatory requirements

12.1. The device must comply with EU Regulation 2017/746 on *in vitro* diagnostic medical devices[7]

[7] Full compliance with EU Regulation 2017/746 outperforms the scope of this thesis, but the design follows the regulation to the best of the author's expertise.

6 Design

Producing software design is open ended – there are many possible acceptable designs, each with its own trade-offs (Bass et al., 2012: 19). A good design meets all specified requirements, if they are not in conflict with one another, but the trade-offs can happen with 'quality attributes' such as modifiability (how easy it is to modify the software) (Bass et al., 2012: 117), testability (how easy it is to find defects through testing) (Bass et al., 2012: 159), usability (how easy it is for the user to use the software) (Bass et al., 2012: 175), or safety (avoiding states that lead to damage or injury) (Bass et al., 2012: 188).

Quality attributes can be met by making the right design decisions. For example, in order to increase system testability, individual elements must be made testable, meaning that their state must be made observable and controllable (Bass et al., 2012: 41). To increase interoperability and maintainability, it is important to specify which software units are responsible for communicating with external systems (Bass et al., 2012: 41).

The requirements specified in the previous section constrain, and in doing so also guide the software design process. In particular, the device must work for a short period of time, process the data of a single person, not use a network connection or receive any additional input from the user during runtime, and run through the command line without any need for a graphical user interface. Hence, the control flow can be relatively linear – modules can be run sequentially, except if a module fails and normal flow of control must be interrupted.

In the following sections, the proposed architecture design and data model are described.

6.1 Architecture

The architecture consists of five main units (modules), each containing one or more components (Figure 2). The five modules – Control, KnowledgeBase, Input, PGxTesting, and Output – handle program flow and self-testing, loading of knowledge base, loading of genetic data, performing PGx tests, and outputting results, respectively. Each module, its components and interaction between modules is further explained below.

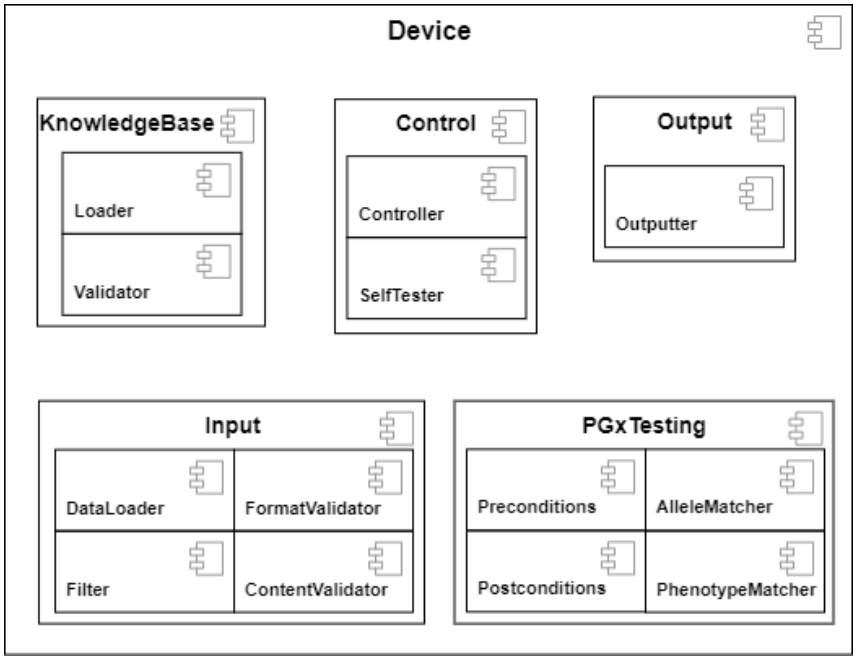

Figure 2. Five main modules and inner components

The two main control flow branches are:

- The intended use of the device – loading of knowledge base, genetic data, performing tests and outputting the results;
- Performing self-testing in the form of end-to-end tests and outputting the results.

Both branches must react to errors via handling exceptions. This is especially relevant to the intended use, where problems in loading the knowledge base or genetic data mean that PGx testing must not occur. The data flow in a typical run (i.e. intended use case without errors) is shown in Figure 3 below.

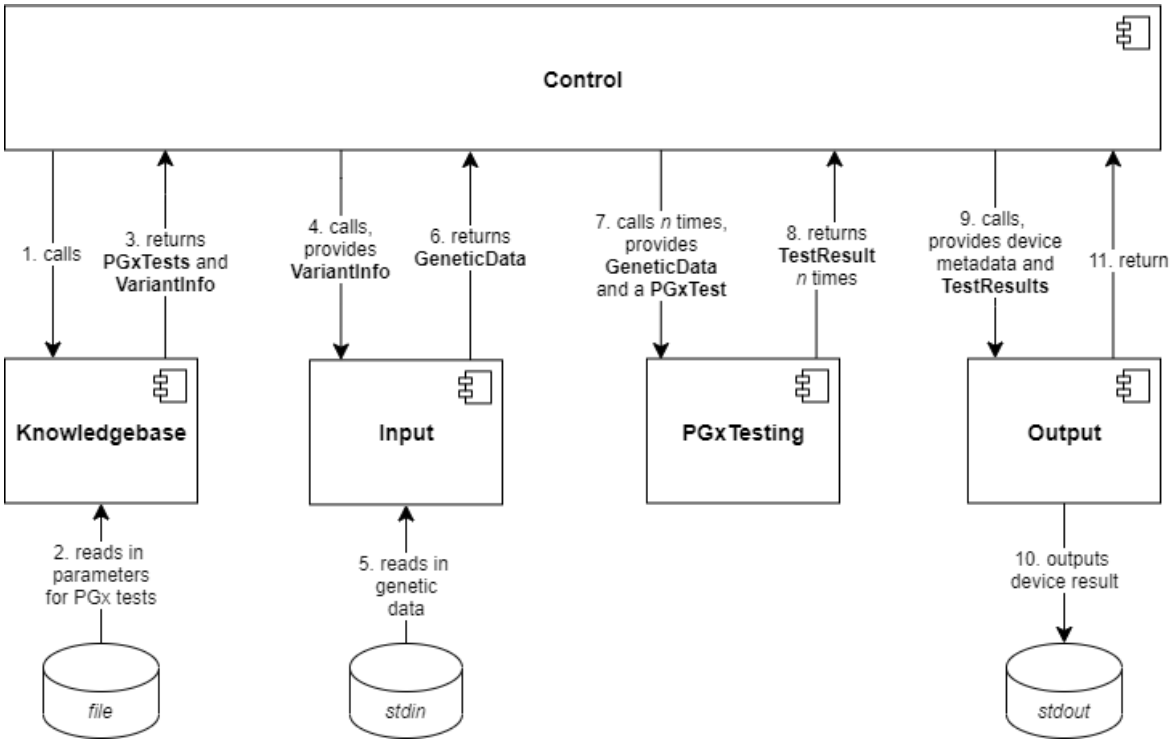

Figure 3. Data flow in a typical run. (1) The KnowledgeBase module is called, which (2) reads in data from files and (3) combines and returns results as domain objects. (4) The Input module is called and it (5) reads in genetic data from the *stdin* stream, (6) validates and translates it to domain objects, and returns a GeneticData object. (7) The PGxTesting module is called for each loaded PGxTest, (8) getting the same amount of TestResult objects back. (9) The Output module is called with test results and metadata, which then (10) combines the results and outputs it to the *stdout* stream. (11) The control flow returns to the Control module and the program terminates.

6.1.1 Control Module

The Control module initializes a logging framework (setting it output to the *stderr* stream), wires other modules together and handles the program control flow, which includes deciding between either self-testing or intended use (Figure 4). Control flow also captures and responds to exceptions. Since end-to-end testing must test the same process that occurs in the intended use case, this process is defined and shown separately (Figure 5). Self-testing can then wrap around it, specify a dataset which is used for a test, run the process, and then compare results against expected results (Figure 6).

The PGx testing process features timeout functionality. The implementation of this feature is to some extent dependent on the programming-language, but can consist of two threads – one runs the process, the other counts down from the timeout period. If the computation finishes before timer reaches zero, the timer thread must be stopped and process flow can continue; if the timer reaches zero before the process finishes, the timer thread must interrupt and stop the process thread.

If any exception is thrown, the program must eventually exit with a non-zero exit status code to signal abnormal termination.

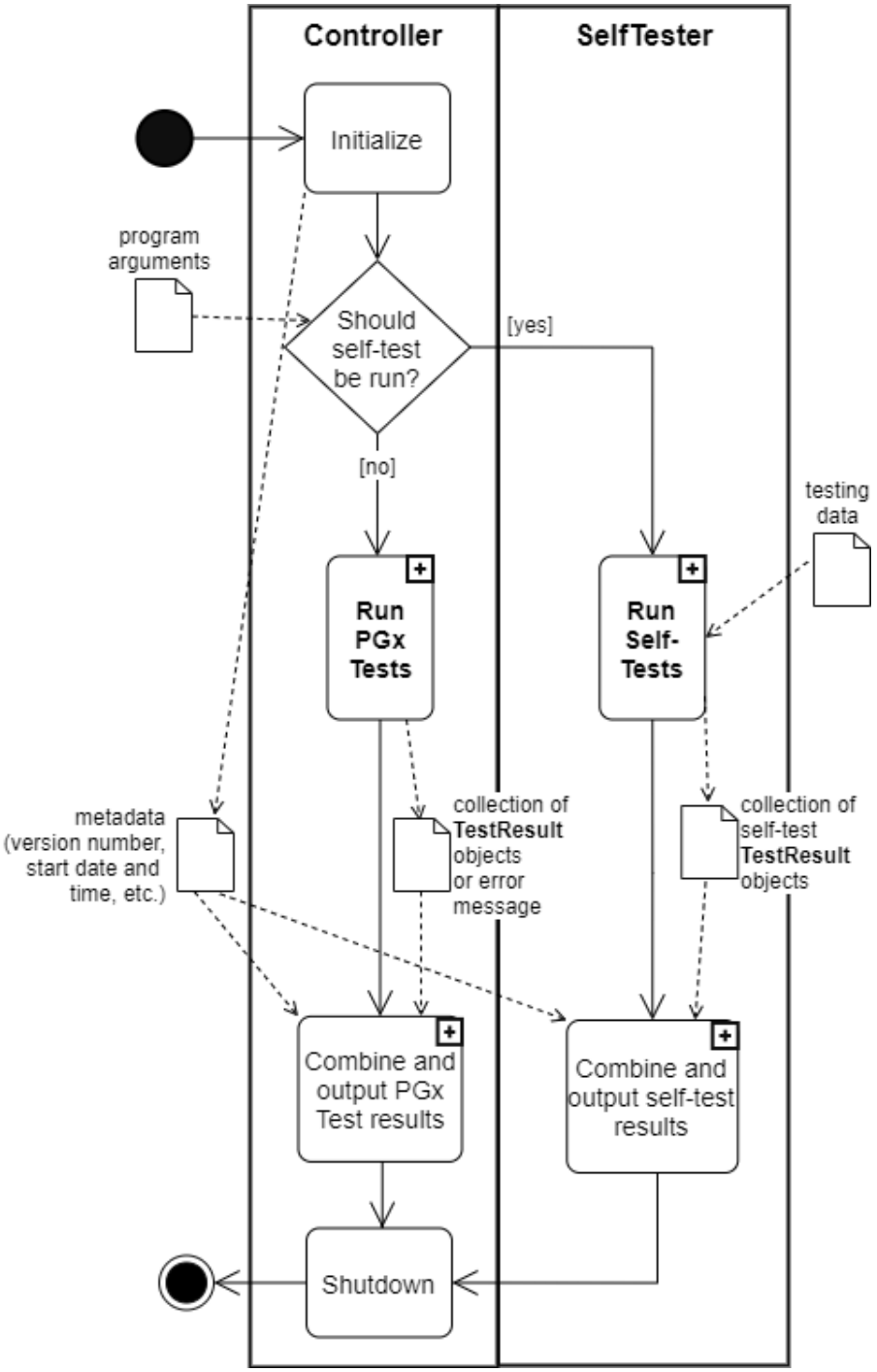

Figure 4. Control module flow. The two main branches, intended use case and self-testing use case, are shown.

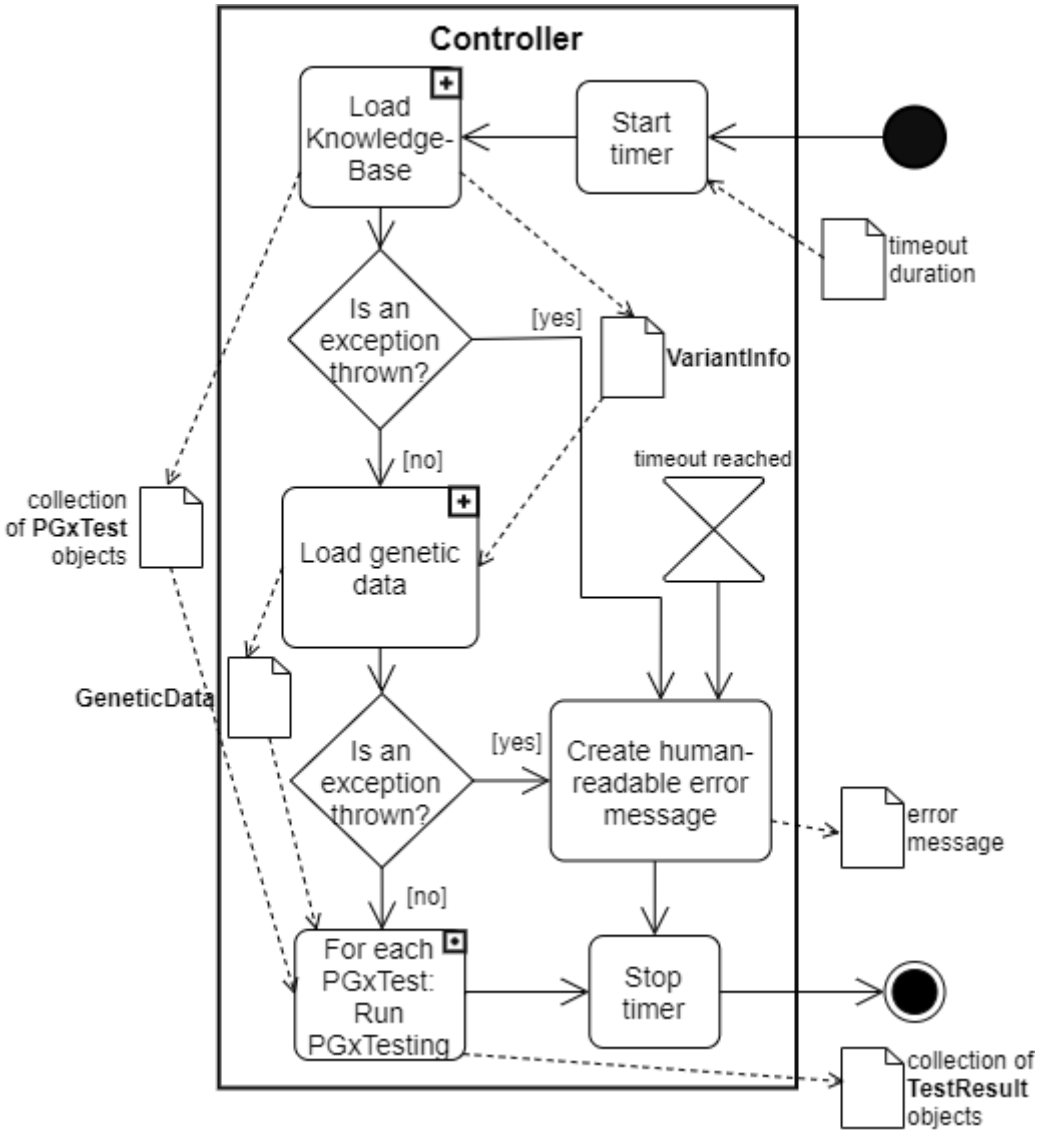

Figure 5. Process flow of running PGx testing – the three main steps are (1) loading of the knowledge base, (2) loading of genetic data, (3) performing each PGx test. In addition, a timer is used to deal with potential hanging. An exception can be thrown when the timer timeout reaches zero or in steps 1 or 2, signalling that PGx testing should not occur.

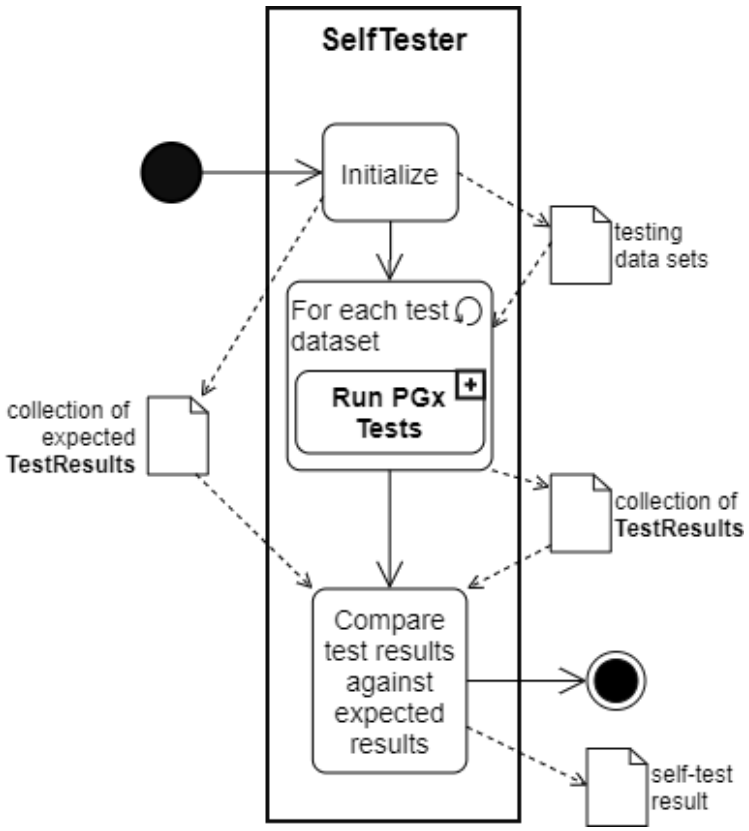

Figure 6. Control flow of running self-testing. First, testing datasets are specified and expected results loaded. Then, for each dataset, the same PGx testing control flow is run as in Figure 5. The outputs are compared against expected output and the self-test result is returned.

6.1.2 KnowledgeBase Module

The KnowledgeBase module (Figure 7) loads and validates the knowledge base. It consists of two components:

- The Loader loads files, does basic checks related to the file format (e.g. non-empty files exist and syntax is correct) and converts the loaded data to domain model objects. Since the knowledge base is given in textual format, this component must also verify that the files have not been tampered with. Integrity can be verified by calculating a checksum and comparing that against a previously calculated checksum.
- The Validator component validates the content of the data (e.g. each PGx test should have at least one star allele, each star allele in turn should be defined by at least one variant; functions assigned to star alleles must be used in diplotype-phenotype map etc.).

Both components perform some form of validation – the Loader runs file format specific validation, while the Validator operates on domain objects and checks their content. This decoupling means that if the incoming data format changes, only the Loader needs to be updated. Each component can also be developed independently.

This module must output two different objects: a collection of PGxTest objects, which are individual pharmacogenetic tests that will be run independently later on, and VariantInfo, which is a collection of data necessary for validating genetic data. The contents of these objects and exception handling are discussed in more detail in chapter 6.2 Domain Model.

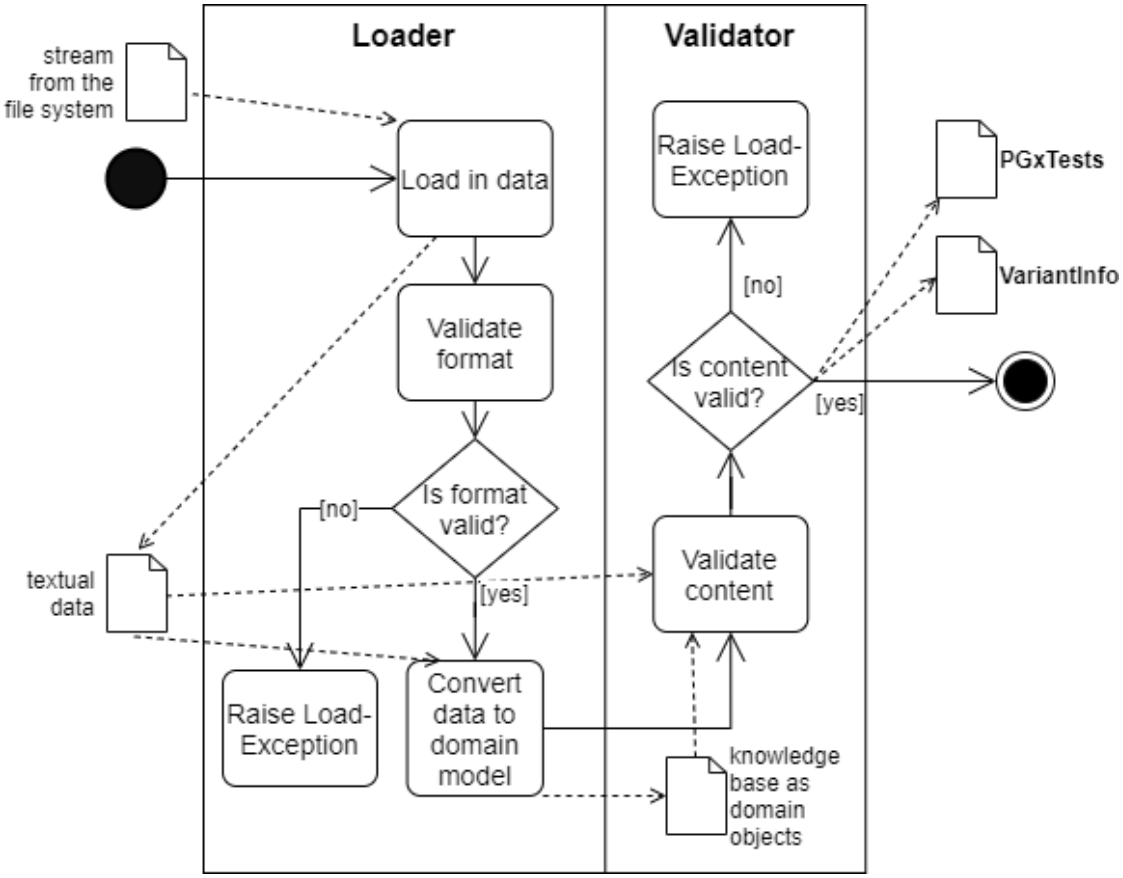

Figure 7. Process flow of running the KnowledgeBase module – loading files and validating the format, converting the data into domain objects, performing further validation and outputting the results. An exception is thrown if format or content validation fails.

6.1.3 Input Module

The Input module (Figure 8) is responsible for loading genetic data. Three different steps can be distinguished, which is mirrored in the decomposition into three components:

- The DataLoader component loads in data from the *stdin* stream, does parsing and format validation (e.g. JSON syntax is valid and all required keys exist etc.) and converts the loaded data to domain model objects.
- The Filter component uses VariantInfo to filter out unnecessary variants (if any).
- The ContentValidator uses VariantInfo to validate the data based on its content (e.g. that the values provided in each variant fall within expected ranges), but also checks for static requirements (e.g. data is phased).

Similar to the KnowledgeBase module, both DataLoader and ContentValidator do validation, but the checks in the former are format-specific and can be decoupled from

content-based validation. The Filter is necessary to handle cases where input data contains variants that are not used by the device and might impact performance. Both DataLoader and ContentValidator must throw a LoadException in case of uncorrectable mistakes to signal that PGx testing must not happen. Since not all genetic variants reach the ContentValidator, it is important to consider if and how this might impact identifying mistakes in the input, and might necessitate adding more tests to the DataLoader or inside the Filter itself.

The output of this module is a GeneticData object, the details of which is explained in section 6.2, Domain Model.

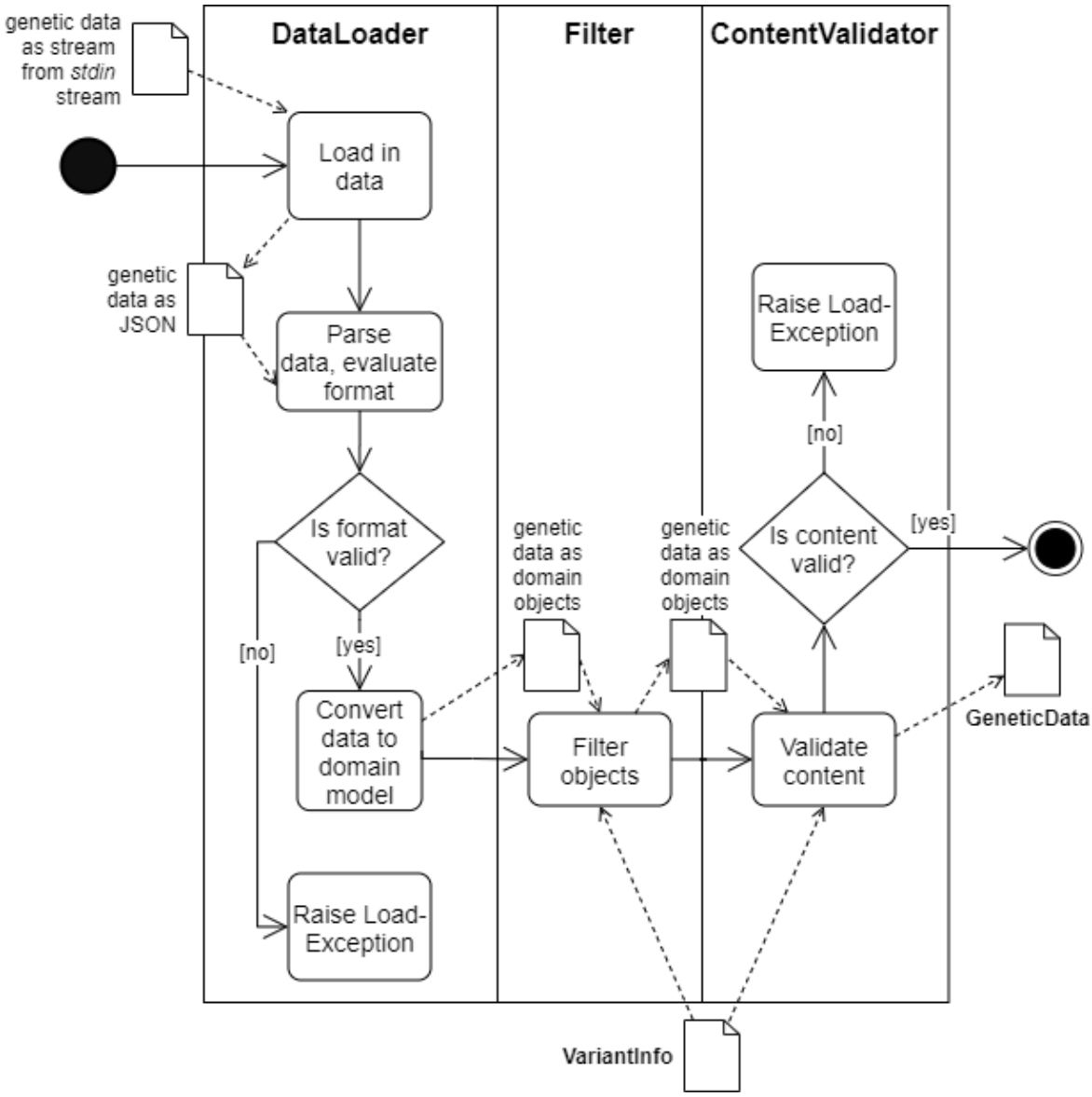

Figure 8. Process flow of loading genetic data, parsing and validating the format, converting the data into domain objects, filtering and validating based on content, outputting a GeneticData object. An exception is thrown if either validation fails.

6.1.4 PGxTesting Module

The PGxTesting module (Figure 9), the core of the medical device, takes in genetic data and parameters of a PGx test and outputs a phenotype. By this stage, it is assumed that genetic data and test parameters are validated and usable. The decomposition of this process resulted in four components:

- Preconditions component does preliminary checks to see if testing can proceed (e.g. that at least one variant defined in the PGxTest is present in genetic data). If any of these checks fail, then the module must return with a TestResult with 'unknown phenotype' and a reason for failure.
- AlleleMatcher assigns star alleles (and therefore also functions) to both haplotypes. The exact process is explained in the Appendix (Figures 11 and 12).
- PhenotypeMatcher iterates over all entries in the diplotype-phenotype mapping and assigns all possible phenotypes based on the assigned haplotype functions. Assigned haplotype functions and functions in map entries are position-invariant, so all combinations must be checked (Appendix, Figure 13).
- The Postconditions component contains checks that assess whether the resulting allele or phenotype assignment was valid. For example, an assignment of alleles with other than the loss of function is valid if all loss-of-function alleles are excluded. Additionally, if multiple alleles with different functionality are assigned on the same haplotype, the novel combination leads to an 'unknown phenotype' assignment.

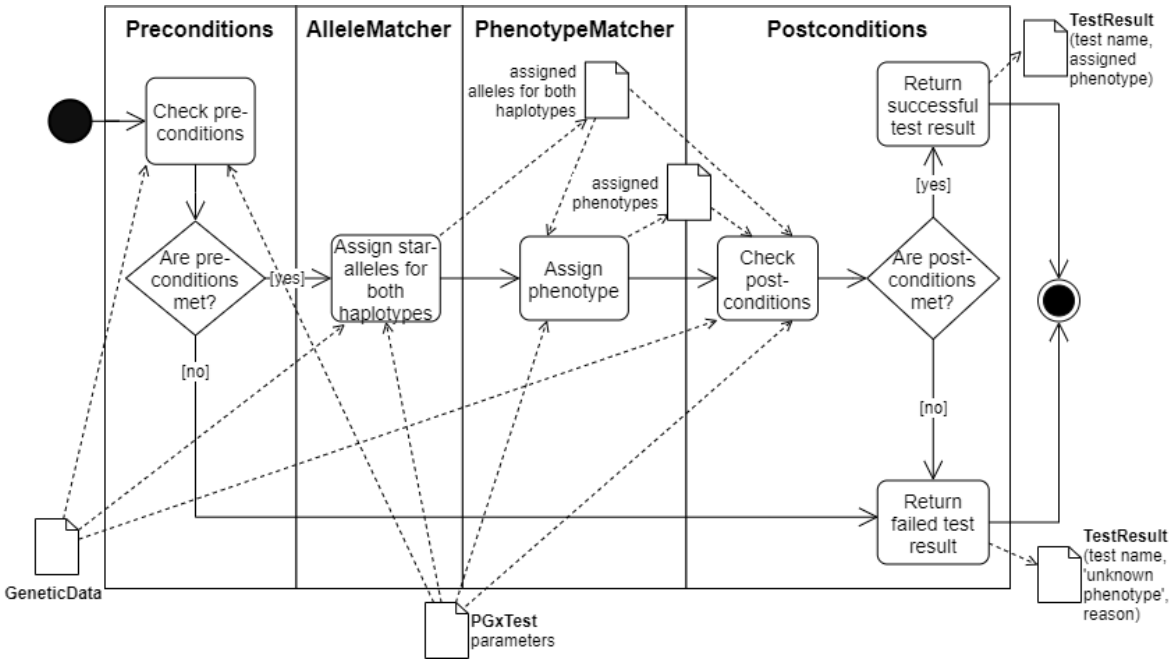

Figure 9. Process flow of running the PGxTesting module – performing a single PGx test. A set of preconditions are checked, followed by assigning star-alleles for both haplotypes and assigning phenotypes. Finally, postconditions are checked to see if the resulting phenotype is valid. The output is a TestResult object, which can be an assigned phenotype

or an ’unknown phenotype’ with a reason (e.g. not enough loss-of-function variants were provided in the genetic data).

6.1.5 Output Module

The Output module (Figure 10) consists of a single component, Outputter, which combines multiple types of textual data into a single structured message and outputs this to the *stdout* stream. In order to better segregate self-testing and intended use cases and hence increase device safety, a separate component or module must be created for either use case.

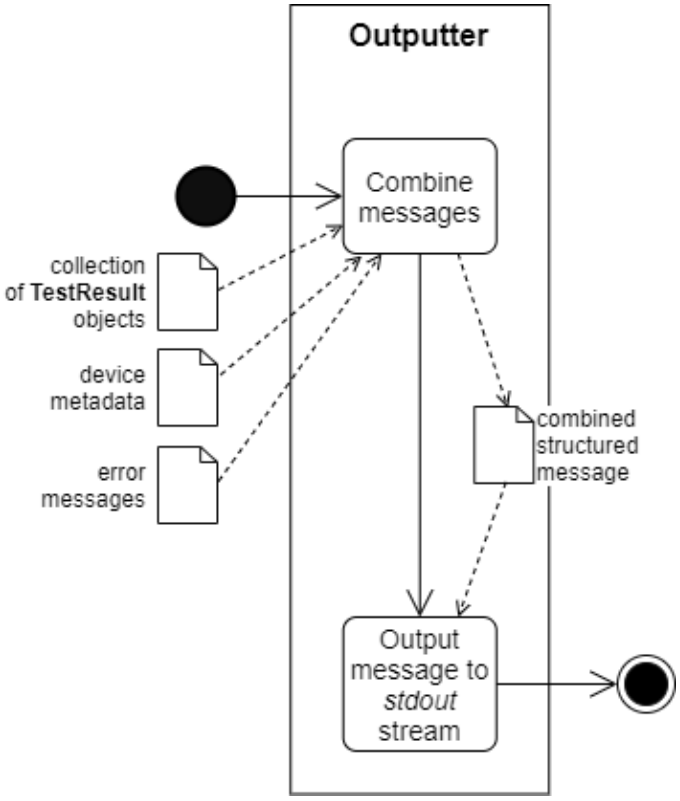

Figure 10. Process flow of combining and outputting results – device metadata and a collection of TestResult objects or error messages are combined into a single structured message and output to the stdout stream.

6.2 Data Model

A data model is used to describe static information structure using entities and their relationships (Bass et al., 2012: 13). As multiple modules in this design use another module output as an input, they need specifying in more detail. Internal data model that is independent of input and output data formats, makes the system more durable and resistant to changes in external data formats.

The data model consists of the following classes (shown with defining fields):

- GeneticVariant
    - rsID (string)
    - chromosome number (string)

    - position on chromosome (string)
    - allele (string)
- GeneticData
    - variants (map of chromosome number and position to allele values, e.g. ‘2:123’: [‘C’,‘T’])
- StarAllele
    - name (string)
    - variants (collection of GeneticVariant objects)
    - function (string)
    - is loss-of-function allele (boolean)
    - is wild-type allele (boolean)
- PGxTest
    - name (string)
    - star alleles (collection of StarAllele objects)
    - diplotype-phenotype mapping, which consists of entries of two functions mapped to a phenotype (e.g. [‘No function’, ‘No function’] : ‘Poor metaboliser’)
- VariantInfo
    - variants (collection of GeneticVariant objects)
    - rsID to chromosome-position mapping
- TestResult
    - name (string)
    - phenotype (string)
    - reason (string)
    - unit (always “NA”)
    - reference value (always “NA”)

Since the validity of PGx testing depends strongly on the knowledge base and genetic data, the corresponding modules must identify problems and have a signalling mechanism that breaks the normal control flow. This can be done via exceptions. Detecting exceptions refers to detecting conditions that alter the normal flow of program execution (Bass et al., 2012: 90).

A single exception type, e.g. LoadException, suffices. Modules can define custom exceptions that extend this class if necessary (e.g. to help map an exception to a human-readable error message), but it is important that this exception is used in cases where PGx testing must not occur. The Control module also wraps the running of these modules inside a block that catches this type of exception and redirects the program flow to exit gracefully.

Caught exceptions and correctable mistakes are accompanied with a warning level message recorded by the logging framework.

7 Verification

The aim of this chapter is to verify whether the architectural design meets the specified requirements. Each requirement is given in Table 1, and for each requirement, it is explained how the requirement is satisfied by the proposed design. The list was then given to an internal reviewer from the GenMed project team, which lead to minor modifications in the proposed design.

Table 1 – verification of design results via checking how requirements are met. The 'Requirement' column contains a requirement number and a short description for brevity. The full description of each requirement can be found in Section 5.

Requirement	**How the requirement is met**
1.1 - input validation	Dedicated module for format and content validation (Figure 8). KnowledgeBase module provides input for content based validation (Figure 5).
1.2 - device works for short period, analyses data for a single person	Control module flow loads in data for a single person, performs PGx tests, outputs results and terminates (Figure 4).
1.3 - runs multiple PGx tests	Control module flow runs PGxTesting module independently for each loaded PGxTest object (Figure 5). Each run returns a phenotype (Figure 9).
1.3.1 - a PGx test consists of allele matching and phenotype matching	PGxTesting module contains AlleleMatcher and PhenotypeMatcher components (Figure 9), assigning alleles and phenotype, respectively. The processes are further described in Figures 11 and 13 (Appendix).
1.3.2 - star allele assignment algorithm	AlleleMatcher component assigns alleles based on the algorithm shown in Figure 12 (Appendix).
1.3.3 - two stages of star allele assignment	AlleleMatcher assigns loss-of-function alleles first, then other alleles (Figure 12, Appendix).
2.1 - timeout	PGxTesting control flow contains a timeout mechanism (Figure 5). If timeout is reached, control flow is interrupted and the Output module receives only the error message.
3.1 - input and output in JSON format	Input module expects JSON and parses it (Figure 8). Output module combines data into JSON (Figure 10).

3.2.1 - input data contains specific variants	Input module contains the Filter component (Figure 8), which filters out unnecessary variants.
3.2.2 - input data is phased	Input module checks for phasity (ContentValidator component, Figure 8).
3.2.3 - input data is aligned to specific reference genome	Input module (ContentValidator component, Figure 8) checks that data is aligned to the expected reference genome and that rsID and chromosome-position pairs match with expected pairs in KnowledgeBase.
3.2.4 - input data need not contain all variants	Input module does not check that all variants from the KnowledgeBase are provided. PGxTesting module (Preconditions component, Figure 9) checks if a specific PGxTest has enough variants.
3.3 - output fields	Output module accepts test results, metadata and errors as input, and combines them (Figure 10).
3.3.1 - metadata contains version number	The Control module flow contains an event (Initialize, Figure 4), which creates the metadata object, which includes the software version number.
3.3.2 - PGx test results fields	PGxTesting module outputs TestResult objects, which contains all the specified fields (Figure 9).
3.3.3 - errors show reason of failure	The Control module flow contains an event which creates a human-readable error message if LoadException is thrown (Figure 5). Control flow is wrapped inside a try-catch block that catches runtime exceptions and exits gracefully (section 6.1).
4.1.1 - *stdin* stream	Genetic data is read through the *stdin* stream in the Input module (Figure 8).
4.1.2 - *stdout* stream	Device outputs the results to the *stdout* stream in the Output module (Figure 10).
4.1.3 - *stderr* stream	Control module flow event *Initialize* sets Logger to output to the *stderr* stream (Figure 4).
5.1 - output contains error messages	PGx testing control flow catches exceptions, creates human-readable error messages (Figure 5), and passes them to the

	Output module, which outputs them (Figure 9). Other, runtime exceptions are caught by the main control flow (section 6.1).
5.2.1 - no test results output in case of timeout	PGx testing control flow contains a timeout timer, which throws an exception if timeout is reached, Output module receives no test results (Figure 5).
5.2.2 - no test results output in case of faulty input	Input module validates data and throws a LoadException if mistakes are identified (Figure 8). PGx testing control flow catches LoadExceptions, Output module receives no test results (Figure 5).
5.2.3 - no test results output in case of faulty KnowledgeBase	KnowledgeBase module loads and validates the knowledge base and throws an exception if mistakes are identified (Figure 7). PGx testing control flow catches exceptions, Output module receives no test results (Figure 5).
5.3 - tests that fail due to lack of data must be indicated	The PGxTesting module has a component (Preconditions, Figure 9), which checks that enough data is provided for a specific test. Lack of data results in a TestResult which states that the test failed and provides a reason.
5.4 - identified mistakes are logged	Exceptions are caught (section 6.1) and followed with logging (section 6.2).
6.1 - interfaces	The device only accepts data from the *stdin* stream (Figure 8), outputs results to *stdout* (Figure 10), and logs to the *stderr* stream (section 6.2).
6.2 - output does not contain raw genetic data	The PGxTesting module outputs a TestResult object, which does not contain raw genetic data (Figure 8; section 6.2).
7.1 - no graphical user interface	The architecture does not contain a GUI module (Figure 2).
8.1 - knowledge base is updatable	The knowledge base is loaded from files and the corresponding process is delegated to a separate module (Figures 2, 3, 7).
8.2 - knowledge base contains text files	The KnowledgeBase module expects multiple text files, which are read, validated, and converted to domain model objects (Figure 7).
8.3 - each PGx test will have its own object	The KnowledgeBase module outputs multiple PGxTest objects (Figure 7).

9.1 - self-test possibility	The Control flow (Figure 4) contains a branch for self-testing.
9.1.1 - self-test performs end-to-end tests	Self-testing does the same procedure as in the intended use case - loading and validating KnowledgeBase, genetic data, and performing individual PGx tests (Figures 4, 5, 6).
9.1.2 - failed self-test result must be indicated in output	Self-testing control flow compares expected and actual test results, and indicates in the TestResults object if the self-test failed or passed (Figure 6). TestResults are passed to the Output module (Figure 10).
10.1 - no network connection required	The device contains all required data (KnowledgeBase, testing data) or receives required data through the *stdin* stream (Figure 3). No module requires a network connection.
10.2 - no dependence on file system changes	The device does not alter files in the file system - all output is sent to the *stdout* or *stderr* stream (Figure 3).
11.1 - user-initiated self-testing possibility	Self-testing is initiated via a program-level argument, which can be provided by the user (Figure 4).
12.1 - regulatory compliance	Risks are reduced via using a timeout mechanism, exceptions, segregation, input validation, providing human-readable error messages, self-testing.

8 Discussion

Designing a good architecture is difficult, since good architecture must satisfy multiple needs – make the system easy to understand, develop, maintain, and deploy, in addition to supporting the proper behavior of the system (Martin 2018: 137). In addition, the design must be communicated clearly in order to be correctly used. The main difficulty consisted of deciding which level of granularity to choose for both the design and the diagrams. An increase in granularity can lead to a cascading effect with many new design decisions required to be made. Fortunately, software-based PGx testing allows a divide and conquer approach: main processes and subtasks can be identified, smaller software units designed and combined to form a whole.

It was also difficult to extract and formulate requirements – end users can have expectations that are impossible to meet, external IT requirements are subject to change, regulation documents are lengthy and complicated.

This work has been interdisciplinary, bordering computer science, software engineering, genetics, and healthcare, and has required some familiarity and basic knowledge in each field. Inevitably, this knowledge is limited compared to that of experts in each field. However, new value arises from combining different fields of expertise and this thesis took on this non-trivial task and provided a good example for future developments.

Moreover, all the stated requirements were met through the choice of architecture and processes.

8.1 Design Strengths and Weaknesses

The main strength of the proposed design is modularity. Main steps in the process flow of a pharmacogenetic test are identified and have a corresponding module in the design. These processes are made as independent as possible, and interdependence is explicitly shown through module inputs and outputs. If possible, modules have been decomposed into components, each dealing with particular tasks. While the allele matching and phenotype matching are hard-coded, PGx testing definitions themselves are entirely data-driven, increasing scalability. However, this requires more thorough identification and reaction to related hazards (e.g. external knowledge base exporting process or loading and combining info from files is faulty).

Components that perform checks can be extended or additional components can be added to modules. Modules can have more outputs or inputs specified. A signaling mechanism (exception) is used to divert the control flow in case PGx testing must not commence, however the throwing of said exception is delegated to modules, which must contain appropriate checks that identify mistakes.

One drawback of the design is that there will be PGx tests in the future which would require additional data, e.g. gene copy numbers or deletions. Since it has not been decided how they will be represented in the incoming data, nor how the algorithm itself should use this data,

they have been left out from the current design. However, there are candidate solutions: a separate control flow branch and another AlleleMatcher component inside the PGxTesting module, or a separate module altogether. In either case, additional checks must be added to components in KnowledgeBase and Input modules, to identify mistakes related to these types of data.

Another potential problem concerns tri-allelic variants. For example, a CYP2C9 variant rs72558189 has both G>T and G>A polymorphisms, each leading to a different protein function. A marker in a genotyping array interrogates a single change (e.g. G>T), and a lack of signal means the absence of T, not necessarily the presence of G (which could be reasonably assumed if the variant is known to be bi-allelic). To assign a wild-type value of G, both G>T and G>A must be interrogated. However, tri-allelic variants constitute a minority in all variants and there is always interpopulation variability, meaning that in a particular population (e.g. Estonians) the variant might not occur even once.

Limited prototyping was employed to improve the architecture and data model. However, this was based on 10 important pharmacogenes. Data model or architecture might need modification if more known pharmacogenes would be explored or higher fidelity prototyping would be done.

8.2 Design Assumptions

The software depends strongly on the validity of the knowledge base or genetic data, which can be validated only partially inside the device. The correctness and usefulness of the device ultimately rests on external processes creating, maintaining and forwarding genetic data and the knowledge base to the device.

The design was constrained by the requirement that the device outputs a single phenotype per PGx test. This has two consequences. Firstly, genetic variants that lead to different phenotypes for different drugs are not easy to accommodate. A related problem is that sometimes guidelines suggest to interrogate different variants for different populations (Johnson et al., 2017). However, these variants tend to have weak or moderate level evidence, and the problem might disappear as evidence accumulates.

Secondly, lack of data can lead to situations where multiple phenotypes could be assigned. This results in a trade-off between usability and functionality – on the one hand, doctors have made it clear that they prefer as simple and unambiguous output as possible, but on the other hand, narrowing down a patient phenotype from five classes to two or three might already be useful. A relevant concept to consider is alert fatigue, referring to a desensitization to alerts as their number increases (Cash, 2009), if more than one possible phenotype is assigned and more alerts are shown. Then again, when to show an alert or not, can be decided by a clinical decision support system (CDSS). However, details related to integrating the software to CDSS have not yet been determined.

8.3 Future Work

The design presented above can be used as a basis to create a more detailed design – starting with the implementation of the domain model, followed by components and modules.

It is difficult to say what aspects of the design need improvement without implementing the software and testing it with real data and each dataset can have its own set of problems. An 'ivory tower' design might needlessly increase software complexity and lead to longer development time through the potential introduction of bugs and verification-validation tasks. However, the design is modular and supports modifications.

9 Conclusions

This thesis proposed a high level software design for a medical software device performing pharmacogenetic tests. To achieve this, the main processes included in the software based PGx testing were identified, software requirements formulated, and a software design created by mapping processes to software units and identifying the order of execution and connections between units. Finally, the proposed design was verified againts the predefined requirements. Formulating software requirements was difficult since end users have expectations that are hard to meet, the external IT requirements are subject to change, and relevant regulations and standards are lengthy.

The proposed design uses modular architecture where each module is responsible for specific tasks. The five main modules of the system are: Control, KnowledgeBase, Input, PGxTesting, and Output. Modularity and clearly defined data objects, including the knowledge base as a seprate module, are the strengths of the proposed design.

Even if the requirements are subject to slight changes in the future, the general architecture is expected to remain unchanged. Creating a higher fidelity prototype and validating it with a working knowledgebase and real data would be the logical next steps for the design.

10 References

Alyass, A., Turcotte, M., Meyre, D. (2015). From big data analysis to personalized medicine for all: challenges and opportunities. BMC medical genomics, 8, 33. https://doi.org/10.1186/s12920-015-0108-y

Bass, L., Clements, P., & Kazman, R. (2012). Software Architecture in Practice (SEI Series in Software Engineering) (3rd ed.). Addison-Wesley Professional.

Bourque, P., Fairley, R. E. (eds.) (2014). SWEBOK: Guide to the Software Engineering Body of Knowledge. Los Alamitos, CA: IEEE Computer Society. ISBN: 978-0-7695-5166-1

Cash J. J. (2009). Alert fatigue. American journal of health-system pharmacy : AJHP : official journal of the American Society of Health-System Pharmacists, 66(23), 2098–2101. https://doi.org/10.2146/ajhp090181

Caudle, K. E., Dunnenberger, H. M., Freimuth, R. R., Peterson, J. F., Burlison, J. D., Whirl-Carrillo, M., Scott, S. A., Rehm, H. L., Williams, M. S., Klein, T. E., Relling, M. V., & Hoffman, J. M. (2017). Standardizing terms for clinical pharmacogenetic test results: consensus terms from the Clinical Pharmacogenetics Implementation Consortium (CPIC). Genetics in medicine: official journal of the American College of Medical Genetics, 19(2), 215–223. https://doi.org/10.1038/gim.2016.87

Coronato, A. (2018). Engineering High Quality Medical Software: Regulations, standards, methodologies and tools for certification (Healthcare Technologies) (1st ed.). The Institution of Engineering and Technology. https://doi.org/10.1049/PBHE012E

Deenen, M. J., Cats, A., Beijnen, J. H., & Schellens, J. H. (2011). Part 1: background, methodology, and clinical adoption of pharmacogenetics. The oncologist, 16(6), 811–819. https://doi.org/10.1634/theoncologist.2010-0258

Dunnenberger, H. M., Crews, K. R., Hoffman, J. M., Caudle, K. E., Broeckel, U., Howard, S. C., Hunkler, R. J., Klein, T. E., Evans, W. E., & Relling, M. V. (2015). Preemptive clinical pharmacogenetics implementation: current programs in five US medical centers. Annual review of pharmacology and toxicology, 55, 89–106. https://doi.org/10.1146/annurev-pharmtox-010814-124835

Hicks, J. K., Bishop, J. R., Sangkuhl, K., Müller, D. J., Ji, Y., Leckband, S. G., Leeder, J. S., Graham, R. L., Chiulli, D. L., LLerena, A., Skaar, T. C., Scott, S. A., Stingl, J. C., Klein, T. E., Caudle, K. E., Gaedigk, A., & Clinical Pharmacogenetics Implementation Consortium (2015). Clinical Pharmacogenetics Implementation Consortium (CPIC) Guideline for CYP2D6 and CYP2C19 Genotypes and Dosing of Selective Serotonin Reuptake Inhibitors. Clinical pharmacology and therapeutics, 98(2), 127–134. https://doi.org/10.1002/cpt.147

Ingelman-Sundberg, M., Mkrtchian, S., Zhou, Y., & Lauschke, V. M. (2018). Integrating rare genetic variants into pharmacogenetic drug response predictions. Human genomics, 12(1), 26. https://doi.org/10.1186/s40246-018-0157-3

Ionova, Y., Ashenhurst, J., Zhan, J., Nhan, H., Kosinski, C., Tamraz, B., & Chubb, A. (2020). CYP2C19 Allele Frequencies in Over 2.2 Million Direct-to-Consumer Genetics Research Participants and the Potential Implication for Prescriptions in a Large Health System. Clinical and translational science, 13(6), 1298–1306. https://doi.org/10.1111/cts.12830

Johnson, J. A., Caudle, K. E., Gong, L., Whirl-Carrillo, M., Stein, C. M., Scott, S. A., Lee, M. T., Gage, B. F., Kimmel, S. E., Perera, M. A., Anderson, J. L., Pirmohamed, M., Klein, T. E., Limdi, N. A., Cavallari, L. H., & Wadelius, M. (2017). Clinical Pharmacogenetics Implementation Consortium (CPIC) Guideline for Pharmacogenetics-Guided Warfarin Dosing: 2017 Update. Clinical pharmacology and therapeutics, 102(3), 397–404. https://doi.org/10.1002/cpt.668

Klein, T. E., & Ritchie, M. D. (2018). PharmCAT: A Pharmacogenomics Clinical Annotation Tool. Clinical pharmacology and therapeutics, 104(1), 19–22. https://doi.org/10.1002/cpt.928

Lonergan, M., Senn, S. J., McNamee, C., Daly, A. K., Sutton, R., Hattersley, A., Pearson, E., & Pirmohamed, M. (2017). Defining drug response for stratified medicine. Drug discovery today, 22(1), 173–179. https://doi.org/10.1016/j.drudis.2016.10.016

McInnes, G., Lavertu, A., Sangkuhl, K., Klein, T. E., Whirl-Carrillo, M., & Altman, R. B. (2020). Pharmacogenetics at Scale: An Analysis of the UK Biobank. Clinical pharmacology and therapeutics, 10.1002/cpt.2122. Advance online publication. https://doi.org/10.1002/cpt.2122

Martin, R. (2017). Clean Architecture: A Craftsman’s Guide to Software Structure and Design (Robert C. Martin Series) (1st ed.). Pearson.

Nebert D. W. (1999). Pharmacogenetics and pharmacogenomics: why is this relevant to the clinical geneticist?. Clinical genetics, 56(4), 247–258. https://doi.org/10.1034/j.1399-0004.1999.560401.x

Numanagić, I., Malikić, S., Ford, M., Qin, X., Toji, L., Radovich, M., Skaar, T. C., Pratt, V. M., Berger, B., Scherer, S., & Sahinalp, S. C. (2018). Allelic decomposition and exact genotyping of highly polymorphic and structurally variant genes. Nature communications, 9(1), 828. https://doi.org/10.1038/s41467-018-03273-1

Pokorska-Bocci, A., Stewart, A., Sagoo, G. S., Hall, A., Kroese, M., & Burton, H. (2014). 'Personalized medicine': what's in a name?. Personalized medicine, 11(2), 197–210. https://doi.org/10.2217/pme.13.107

Pratt, V. M., Del Tredici, A. L., Hachad, H., Ji, Y., Kalman, L. V., Scott, S. A., & Weck, K. E. (2018). Recommendations for Clinical CYP2C19 Genotyping Allele Selection: A Report of the Association for Molecular Pathology. The Journal of molecular diagnostics: JMD, 20(3), 269–276. https://doi.org/10.1016/j.jmoldx.2018.01.011

Reisberg, S. (2019). Developing Computational Solutions for Personalized Medicine (Doctoral dissertation, University of Tartu, Tartu, Estonia). Retrieved from http://hdl.handle.net/10062/64628

Reisberg, S., Krebs, K., Lepamets, M., Kals, M., Mägi, R., Metsalu, K., Lauschke, V. M., Vilo, J., & Milani, L. (2019). Translating genotype data of 44,000 biobank participants into clinical pharmacogenetic recommendations: challenges and solutions. Genetics in medicine: official journal of the American College of Medical Genetics, 21(6), 1345–1354. https://doi.org/10.1038/s41436-018-0337-5

Regulation (EU) 2017/746 of the European Parliament and of the Council of 5 April 2017 on in vitro diagnostic medical devices and repealing Directive 98/79/EC and Commission Decision 2010/227/EU. OJ L 117, 5.5.2017, p. 176–332

Relling, M. V., & Evans, W. E. (2015). Pharmacogenomics in the clinic. Nature, 526(7573), 343–350. https://doi.org/10.1038/nature15817

Sabater, A., Ciudad, C. J., Cendros, M., Dobrokhotov, D., & Sabater-Tobella, J. (2019). g-Nomic: a new pharmacogenetics interpretation software. Pharmacogenomics and personalized medicine, 12, 75–85. https://doi.org/10.2147/PGPM.S203585

Twesigomwe, D., Drögemöller, B. I., Wright, G., Siddiqui, A., da Rocha, J., Lombard, Z., & Hazelhurst, S. (2021). StellarPGx: A Nextflow Pipeline for Calling Star Alleles in Cytochrome P450 Genes. Clinical pharmacology and therapeutics, 10.1002/cpt.2173. Advance online publication. https://doi.org/10.1002/cpt.2173

Twist, G. P., Gaedigk, A., Miller, N. A., Farrow, E. G., Willig, L. K., Dinwiddie, D. L., Petrikin, J. E., Soden, S. E., Herd, S., Gibson, M., Cakici, J. A., Riffel, A. K., Leeder, J. S., Dinakarpandian, D., & Kingsmore, S. F. (2016). Constellation: a tool for rapid, automated phenotype assignment of a highly polymorphic pharmacogene, CYP2D6, from whole-genome sequences. NPJ genomic medicine, 1, 15007. https://doi.org/10.1038/npjgenmed.2015.7

Appendix

I. Allele Matching and Phenotype Matching Algorithms

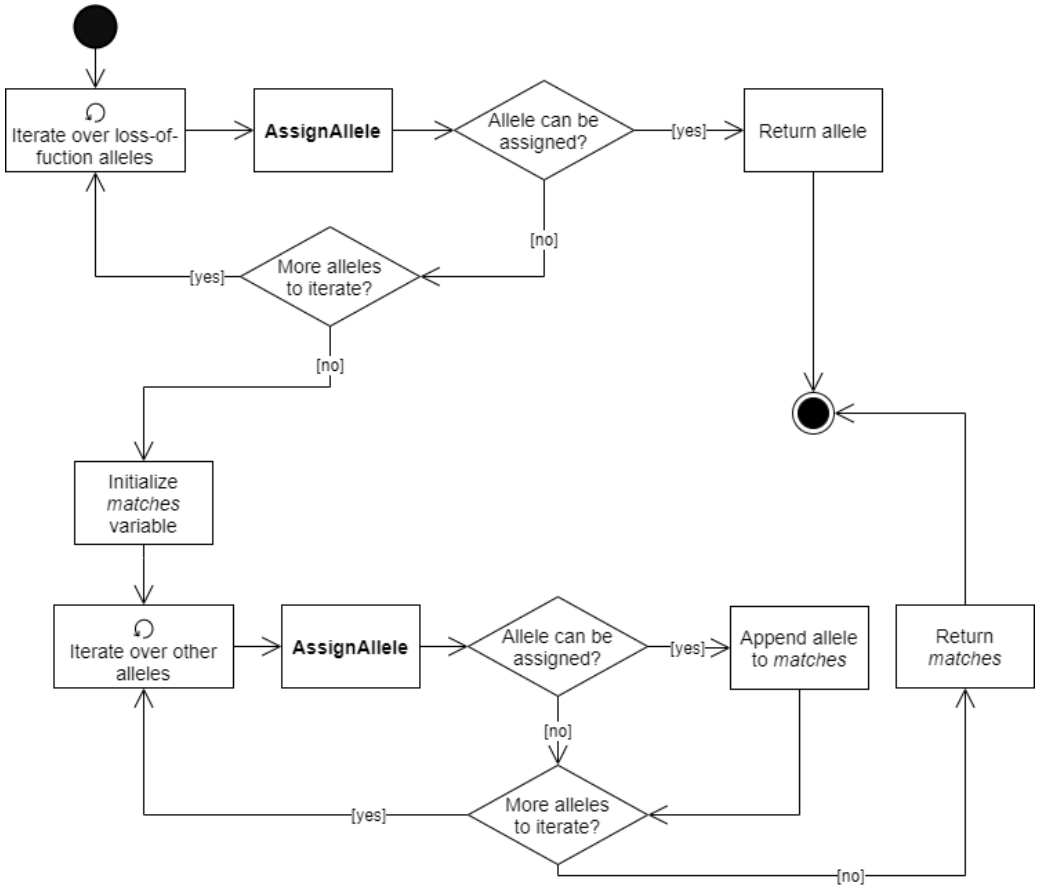

Figure 11. Algorithm for assigning star alleles for a haplotype. The algorithm requires genetic data for a haplotype and a list of alleles that are interrogated, and consists of two stages. First, a single loss-of-function star allele is searched for; if found, it is returned. Otherwise, all other star alleles are matched and returned. The algorithm for assigning a single allele (denoted as 'AssignAllele' above) is further explained in Figure 12.

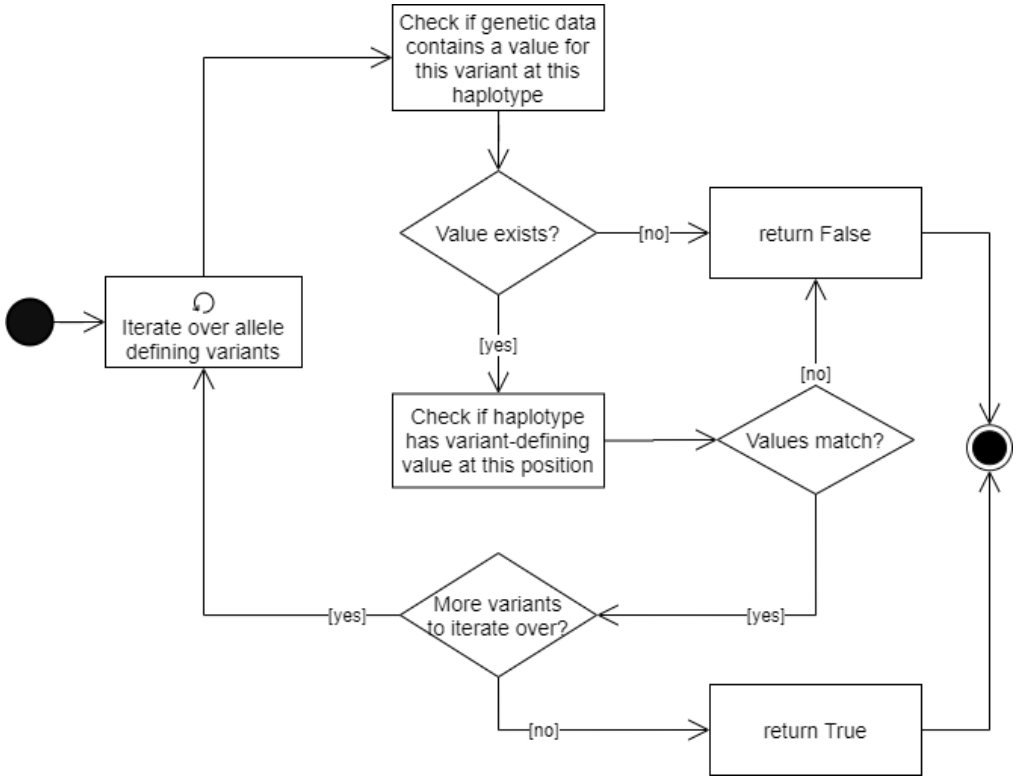

Figure 12. Algorithm for assigning a single star allele. The algorithm requires genetic data for a haplotype and a list of SNVs defining a star allele. The algorithm returns True if for all SNVs, respective data on the haplotype is provided, and the SNV allele matches the haplotype allele.

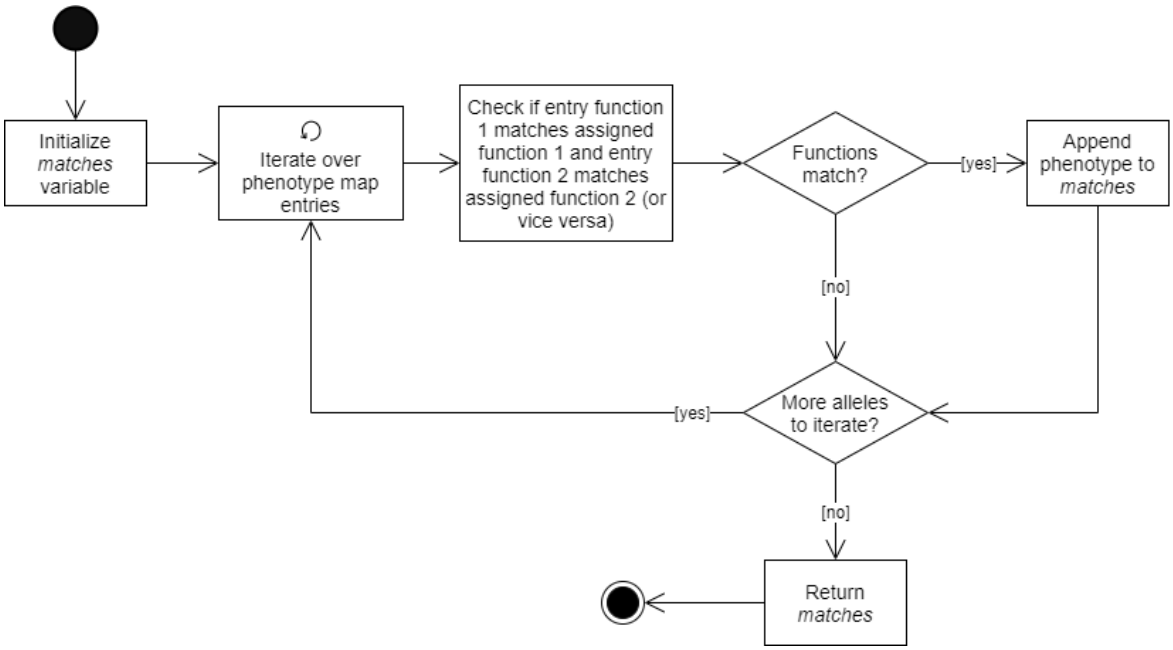

Figure 13. Algorithm for assigning phenotypes. The algorithm requires a mapping of functions to phenotype, and a tuple of assigned star alleles for each haplotype. All possible phenotypes are assigned and returned.

II. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Taavi Luik**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Designing a Pharmacogenetic Test as a Medical Software Device,
   supervised by Sulev Reisberg, PhD and Kersti Jääger, PhD.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Taavi Luik

14/05/2021