Software Quality Assessment of a Web Application for Biomedical Data Analysis

Kristina Lietz
Bioinformatics Group
Fraunhofer ITEM
Hannover, Germany
kristina.lietz@item.fraunhofer.de

Ingmar Wiese
Bioinformatics Group
Fraunhofer ITEM
Hannover, Germany
ingmar.wiese@item.fraunhofer.de

Lena Wiese[*]
Bioinformatics Group
Fraunhofer ITEM
Hannover, Germany
lena.wiese@item.fraunhofer.de

ABSTRACT

Data Science as a multidisciplinary discipline has seen a massive transformation in the direction of operationalisation of analysis workflows. Yet it can be observed that such a workflow consists of potentially many diverse components: like modules in different programming languages, database backends, or web frontends. In order to achieve high efficiency and reproducibility of the analysis, a sufficiently high level of software engineering for the different components as well as an overall software architecture that integrates and automates the different components is needed. For the use case of gene expression analysis, from a software quality point of view we analyze a newly developed web application that allows user-friendly access to the underlying workflow.

CCS CONCEPTS

• Software and its engineering → Requirements analysis; **Maintaining software**; **• Applied computing → Computational transcriptomics**; **Bioinformatics**;

KEYWORDS

Data Science workflow, Gene expression analysis, Software quality, Web service

ACM Reference Format:
Kristina Lietz, Ingmar Wiese, and Lena Wiese. 2021. Software Quality Assessment of a Web Application for Biomedical Data Analysis. In *25th International Database Engineering & Applications Symposium (IDEAS 2021), July 14–16, 2021, Montreal, QC, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472163.3472172

1 INTRODUCTION

Data has evolved to become one of the most important assets for any business or institution in recent years and constitutes one of the key factors driving current innovations. For example, the steep spike in technologies for personalized medicine enabled by wide applications of genome or transcriptome analysis are facilitated by a constant collection of (big) data. Workflows in a Data Science context refer to the processing of data from its raw form to the finished interpretation and result visualization. Such a workflow includes steps such as cleaning, feature engineering, model generation, and validation. In order to fully operationalize the workflow, this process may also include the deployment of the workflow in an interactive webservice.

As a typical use case for an analysis workflow we address gene expression analysis. Gene expression analyses are important in areas such as drug development and tumor research. The challenge of analyzing genome and transcriptome data volumes has become increasingly important in recent years and cover aspects of exploratory data analysis, statistics and machine learning. Since scientists performing gene expression experiments usually lack statistical and computer science knowledge to handle those amounts of data, usable interfaces are needed. To enable an easy analysis of the data, we designed and implemented a web service that abstracts from the mathematical details and any programming tasks. The web interface was implemented using the R Shiny framework which was chosen because R libraries are widely used for gene expression analysis. Yet so far the R Shiny framework itself has not been thoroughly evaluated regarding its capability for implementing complex data science applications.

As the main goal of this article we provide an evaluation of our web application in terms of software quality. More precisely, our focus lies on comparing the maintainability of modularized versus non-modularized Shiny applications as exemplified by our gene expression web application.

A minor secondary goal of the article is to cover the combination of aspects with respect to workflows and DevOps by briefly describing an implementation of an IT system infrastructure that serves as our basis for a flexible deployment of the data analysis workflows and web applications. We believe that a consideration of DevOps paradigms can simplify development and operations of typical Data Science workflows and enables a high quality in terms of reproducibility, reusability and automation be achieved with low maintenance effort.

Outline. The paper is structured as follows. Section 2 describes the gene expression analysis workflow as a use case. Section 3 introduces the topic of software quality. Section 4 summarizes our requirements analysis and presents related work. Section 5 discusses our approach towards modularity of our application. Section 6 presents general design principles and analyzes their analogies in Shiny. Section 7 gives an in-depth analysis of our application based on quality metrics. Last but not least, Section 8 touches upon

[*]Also with Institute of Computer Science, Goethe University Frankfurt.

aspects of a real-world deployment before concluding the article in Section 9.

2 USE CASE: GENE EXPRESSION ANALYSIS

Gene expression is defined as the process of using the sequence of nucleotides of a gene to synthesize a ribonucleic acid (RNA) molecule, which in turn triggers the synthesis of a protein or performs some other biological function in a cell. As a result of this process, the nucleotides' sequence determines the biological information of a gene [6].

2.1 Data Format

We focus on development of a web service to analyze gene expression data generated with Affymetrix microarray chips. Microarrays are a collection of genes or cDNAs arranged on a glass or silicon chip [4]. DNA molecules are located on spots or features on the microarray chips' silicon surface. On each feature are a few million copies of the same section of a DNA molecule. This section can be associated with a specific gene and is in Affymetrix chips composed of 25 nucleotides. cDNA molecules get labeled with a fluorescent dye and form the target molecule of the experiment. Finally, the intensity of the fluorescence of the individual spots is measured using special lasers. The more the oligonucleotides in the spots are hybridized, the stronger the emission of light will be. An image is generated from the intensity of the individual spot's fluorescence and stored for further analysis. The generated image is kept in the data (DAT) file format, which contains the measured pixel intensities as unsigned integers. A so-called CEL file summarizes the DAT file and is used in the further analysis.

Beyond the measurement data, additional metadata annotations have to be stored. The MicroArray Gene Expression Tabular (MAGE-TAB) specification is a standard format for annotating microarray data, which meets MIAME requirements. MIAME is a recommendation of the Microarray Gene Expression Database society. It is a list of information that should be provided at a minimum when microarray is published to allow description and reproduction [5]. MAGE-TAB defines four different file types: Investigation Description Format; Array Design Format; Sample and Data Relationship Format (SDRF); raw and processed data files. For the subsequent analysis, the SDRF file is of particular importance because information about the relationships between the samples, arrays (as CEL files), and data of the experiment is contained.

2.2 Workflow

The data analysis for a gene expression experiment is usually conducted by following a specific workflow. After performing a gene expression experiment, first of all the quality of the resulting data must be verified using different metrics. If the quality is insufficient, some data may need to be removed from the analysis or parts of the experiment may need to be repeated [13]. If the quality of the data is sufficient, the data needs to be preprocessed. This process consists of three steps: Background correction to remove influences on the data that have no biological cause; normalization to make individual samples comparable; and summarization to calculate one value from multiple measured values of the activity of the same gene [13]. Finally, the actual statistical data analysis is performed to answer the research question [15].

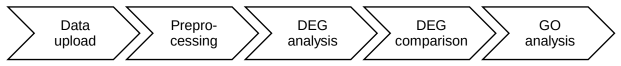

Figure 1: Workflow for analyzing differentially expressed genes (DEGs)

We present now the workflow from the user's point of view by detailing all steps followed by a user in our web application. These steps are initially derived from the MaEndToEnd workflow [17], yet several customizations had to be added as requested by the endusers of our web service. In Figure 1, the basic steps are visualized: Data upload, preprocessing, DEG analysis, DEG comparison and gene ontology (GO) analysis.

(1) The workflow starts with the upload of the raw input files. For this, the user has to define the source of the files, currently either the local file system, a SQL database connection or the GEO repository [10] are available as options. It is possible to filter the samples from the input data since not all samples may be of interest.
(2) As the first step of the preprocessing, the uploaded files can be filtered for their so-called gene chip types. An all-in-one pre-processing algorithm can be selected by the user, which is then applied to the data. It performs the steps of background correction, normalization, and summarization. Afterwards, the user can plot the data in various ways for quality control.
(3) Next follows the actual gene expression analyses. A histogram of the distribution of the p-values is plotted for each analysis, where it is expected that the frequency ranges between very high near 0 and low towards 1. The user verifies this. If the histogram does not meet the expectations, the workflow ends at this point. Possible faults could be incorrect data, incorrect pre-processing or improperly defined analyses. If the histogram meets the expectations, the next step is to set the significance thresholds. These define ranges of statistical values calculated during the analysis in order to identify differentially expressed genes (DEGs).
(4) In the next step of the workflow, it is possible to compare the performed gene analyses with each other. For the comparison, different plots and Venn diagrams should be created automatically.
(5) For classifying the identified differentially expressed genes (DEGs) in the biological context, gene ontology (GO) [7] based enrichment analyses are performed in our web service. This offers a uniform vocabulary, which applies to all eukaryotes.

Our web application offers a separate pane for each of the above workflow steps.

3 SOFTWARE QUALITY

As already described, the importance of data analysis applications is increasing. Whether a framework is suitable for the development of complex applications depends on many factors. We aim to analyse our web service according to standardized software quality metrics. The term software quality is defined by the International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) as the "degree to which a software product satisfies stated and implied needs when used under specified conditions" [16]; see also [12] for a recent survey. The ISO/IEC 25000 norm represents an international standard entitled "Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE)" including a section on System and software quality models. A distinction is made between the following quality models: *Quality in Use* and *Product Quality*. We will focus here on Product Quality that is defined as "characteristics [...] that relate to static properties of software and dynamic properties of the computer system" [16].

We now introduce a selection of these characteristics that we later consider in the assessment of our software.

(1) Functional suitability consists of the sub characteristics functional completeness, appropriateness, and correctness. It addresses the effectiveness of the software product and is assessed for our software by a comprehensive requirements analysis and subsequent evaluation in user tests.
(2) Usability includes appropriateness recognizability, learnability, operability, user error protection, UI aesthetics, and accessibility. Thereof, appropriateness recognizability is defined as "the degree to which users can recognize whether a product or system is appropriate for their needs". Usability of our software is also evaluated in user tests and feedback interviews.
(3) Maintainability is defined as "the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers". It includes the sub characteristics modularity, reusability, analyzability, modifiability, and testability. The sub characterizations are explained further because they are considered in more detail to evaluate our Shiny framework later on with quantitative metrics.
   (a) Modularity describes the separation of the software into components, where changes in a component should affect dependent components as little as possible.
   (b) Reusability means the use of assets in several software systems or with building new assets. Assets are defined as "work products such as requirements documents, source code modules, measurement definitions, etc." [16].
   (c) The sub characteristic analyzability describes the understanding of the interrelations in the software product, for example, the impact a change would have.
   (d) Modifiability describes the extent to which software can be changed without degrading quality or introducing errors. According to ISO/IEC, the modifiability can be impacted by modularity and analyzability.
   (e) Finally, testability describes the possibilities of setting up test criteria and verifying their compliance.
(4) Compatibility (within the characteristic co-existence) describes the impact on other software running on the same platform as well as (within the characteristic interoperability) the exchange of information with other software. These aspects are considered in our software since input and output file formats are supposed to be interchangeable with other software products by using standardized data formats, allowing access to public repositories (like Gene Expression Omnibus) as well as offering a versatile database system access.

We mention in passing that the Product Quality model [16] also covers other characteristics which are however out of the scope of this paper. The characteristics of performance efficiency describes the properties time behavior, resource utilization and capacity and depends on the used platform or environment. Reliability describes the maturity, availability, fault tolerance and recoverability of the software. Security includes the confidentiality, integrity, non-repudiation, accountability and authenticity of the software, which will not be considered in this article since the application is only accessible to researchers inside the institute. Finally, portability includes the adaptability, installability and replaceability of the product; while not quantitatively analyzing portability, we address this issue in Section 8.

4 REQUIREMENTS ANALYSIS AND COMPARISON TO RELATED WORK

For the implementation of the web service, the software requirements have to be specified first. During the complete development process, new requirements were identified, resulting from literature search and interviews with future users of the web service. As already mentioned, various requirements were derived from the MaEndToEnd workflow. Requirements were identified and documented during the entire development process.

A distinction between functional and non-functional requirements is made. Examples of non-functional requirements are the system's availability or safety aspects. The requirements are graded into high, medium and low concerning their relevance for implementation [24]. We identified 30 functional requirements and 4 non-functional requirements (which cannot be fully reproduced here in detail due to space restrictions); out of the functional requirements 13 and out of the non-functional requirements 2 were graded as high.

Because of the relevance of gene expression analyses, many tools for those are already available. For justifying the development of a new tool, the following paragraphs will give a short presentation of a selection of existing tools and show the differences between the defined requirements and the functionalities of these tools. The Transcriptome Analysis Console (TAC) software [1] is presented because Fraunhofer ITEM researchers currently use it for gene expression analyses. The other presented tools were selected based on a literature search; two of them were chosen because their functionalities and the defined requirements intersected as closely as possible: GEO2R [20] and Network-Analyst [25]. In particular, only tools that are open source and offer a graphical UI were selected.

The TAC software [1] is provided by Affymetrix Inc., the manufacturing company of the microarray chips. The gene expression analyses are mostly performed with the R packages provided by Bioconductor [14]. According to our requirements analysis we identified the following shortcomings of the software: Uploading or creating SDRF files is not possible, but definable attributes provide the functionality. Furthermore, it is possible to compare several analyses, but not to the required extent. An installation of the TAC software on the client is mandatory. The significance values cannot be set individually for the single DEG analyses. Also, performed analysis steps can only be traced by manually repeating the steps.

GEO2R offers an option for gene expression analysis available on the website of the GEO database [10]. According to our requirements analysis we identified the following shortcomings of the software: Local files and SDRF files cannot be uploaded. Furthermore, only one biological group per sample can be specified, which limits the possible analyses. GEO2R does not offer the ability to generate PCA plots and heat maps or filter the samples after quality control. Multiple DEG analyses can only be defined indirectly because all defined biological groups are automatically compared. For these, only similar significance thresholds can be defined. A comparison of several analyses is only offered for analyses made between samples of the same series and only via one Venn diagram.

NetworkAnalyst [25] is a free web-based tool for gene expression analysis that offers DEG analyses and network analyses. According to our requirements analysis we identified the following shortcomings of the software: Files in CEL format, as well as SDRF files, cannot be uploaded. However, the functionality of defining biological groups can be achieved by specifying the factors elsewhere. For preprocessing, the tool only applies the filtering of genes and the normalization step. Background correction, as well as summarization, does not seem to be performed. Moreover, the filtering of samples after quality control is not possible. Multiple DEG analyses can be defined, but not as precisely as required and individual significance thresholds can only be defined for the adjusted p-value. Several analyses can be compared, but not to the requested level.

Overall, no tool completely meets the requirements graded "high". Especially the comparison of analyses is only possible to a limited extent. The development of an own application for gene expression analysis within our institute offers further advantages. Data that has not yet been published can be analyzed without any concerns about data protection. In contrast, the security of third-party tools, especially if they are publicly available on the internet, has to be verified first. Moreover, if new requirements arise, the software can be flexibly extended.

5 SHINY MODULARIZATION

The programming language R provides an environment for the analysis and graphical visualization of data [22]. R was chosen for our implementation because the open-source software project Bioconductor provides many R packages for the analysis of gene expression data. Although these could also be used in other programming languages, the application's complexity can be reduced by limiting it to one programming language. This has a positive impact on the maintainability of the software. With R Shiny, a framework for the development of web-based applications directly in R is provided. An application is built from two main components: A UI object and a corresponding server function, which accesses the UI elements via their defined IDs. Also, the server function defines the logic for the functionality of the web app, for example, processing the data or plot variables. The elements defined in the UI object are either input or output elements. Values of the input elements are processed in the server function code, while the output elements visualize the results of those calculations. Moreover, Shiny uses a reactive programming model. Reactivity in Shiny means that when the input values are changed, the code sections that access the changed input elements are automatically re-executed. Thus, the corresponding output elements are also automatically updated.

As mentioned in Section 4, for the implementation of our web service, we first specified software requirements by close communication with the end users. During the complete development process, new requirements were identified, described, and evaluated. The process was thus iterative. A first prototype of our framework hence resulted in a *monolithic* source code.

Defining modules should improve the handling of complex Shiny applications. For this purpose, functions are used, which are the fundamental unit for abstraction in R. Modules consist of functions that generate UI elements and functions used in the module's server function. There is a global namespace within a Shiny application, so each ID of input or output elements must be unique within the app. By using functions for generating UI elements, the uniqueness of the IDs must be ensured. Shiny modules solve this problem by creating an abstraction level beyond functions.

Shiny modules have three basic characteristics:

(1) They cannot be executed alone but are always part of a larger application or module. The nesting of modules is therefore possible.
(2) They can represent input, output or both.
(3) They are reusable: both in several applications and several times in the same application.

For creating a module, one function is needed that defines UI elements and one function that uses these elements and contains the server logic. The UI function expects an ID as a parameter, which defines the namespace of the module. The caller of the function defines this. The namespace is applied to all IDs of the input and output elements for ensuring the correct mapping of the elements to the namespace. This ID as namespace must also be passed to the server function of the module. In this, the moduleServer function is invoked, where the actual server logic is defined. The objects input and output are aware of the namespace, so the elements with ID wrapped in the defined namespace can be referenced using the standard way input$elementID. UI elements outside the namespace cannot be addressed at all. In the regular components of the Shiny application, the UI function of the module is called in the UI object and the server function of the module in the regular server function. It is essential to pass the same ID to the corresponding functions.

In order to improve maintainability, we refactored the monolithic version of our app into a modularized one. Figure 2 illustrates an overview of the defined main modules and their sub modules of the web application. One module was defined for each main step of the workflow, resulting in the following main modules: dataUpload, preprocessing, degAnalysis, degComparison, and goAnalysis. Each of these main modules are realized within a separate tab/panel in the web application. Each of the main modules accesses at least one sub-module with more specific responsibilities.

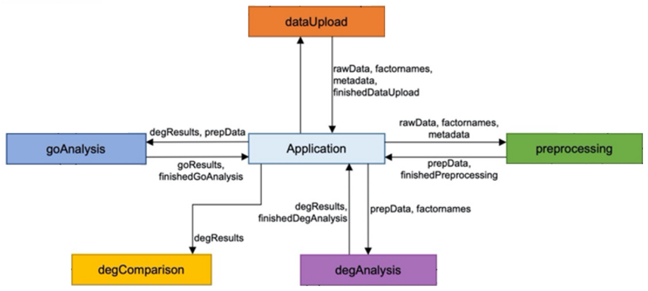
dataUpload
rawData, factornames, metadata, finishedDataUpload
degResults, prepData
goAnalysis
Application
goResults, finishedGoAnalysis
rawData, factornames, metadata
preprocessing
prepData, finishedPreprocessing
degResults, finishedDegAnalysis
prepData, factornames
degResults
degComparison
degAnalysis

Figure 2: Modularized application

	maintai-nability	sub characteristics			
		modu-larity	reusa-bility	analys-ability	modifi-ability
encapsulation	↑	↑	↑		↑
coupling	↓	↓	↓	↓	↓
cohesion	↑	↑	↑	↑	↑
size	↓			↓	↓
complexity	↓		↓	↓	↓

Table 1: Influence of OO design principles on the sub characteristics of the software quality property maintainability; an arrow pointing up (↑) indicates a positive influence of the design property on the quality (sub) characteristic and an arrow pointing down (↓) indicates a negative influence.

6 SOFTWARE DESIGN PRINCIPLES

This article focuses on the evaluation of the maintainability of Shiny applications. The focus is on the maintainability of the Shiny components that build and control the user interface (UI). As already specified, maintainability consists of the sub characteristics modularity, reusability, analyzability, modifiability, and testability. The latter is not considered in this evaluation of Shiny, as this is a very extensive topic and would exceed the scope of this paper. In the following, a relationship between these sub-characteristics and design principles from the object-oriented (OO) programming paradigm is first established. In this section, the term application refers to a Shiny web app and the term module refers to a Shiny module.

6.1 Influence of OO design principles on maintainability

When designing OO software products, several design principles are defined, supporting the enhancement of the software quality. In the following, a selection of the design principles defined by Schatten et al. [21] is presented and assumptions about their influence on the sub characteristics of maintainability are made. Only the principles relevant to the metrics calculation in Section 7 are considered. The assumptions about the impact on maintainability are based on the results of a literature review and the comparison of the definitions of the design principles to those of the maintainability characteristics.

Table 1 indicates with upward and downward arrows an overview of our assumptions made for the influence of the design principles on the software quality; that is, ideally higher encapsulation, looser coupling, higher cohesion, and less extensive and less complex software. We discuss our assumptions on software design principles in detail:

Encapsulation. With encapsulation, details of the implementation should be hidden from the outside, realized in the OO paradigm by private fields and methods. Communication between components should occur only via defined interfaces, which are public fields and methods. For example, this enables to change the encapsulated functionality of a component, without affecting the of calling components as long as the interface remains unchanged. The definition's similarity of the sub characteristic modularity of maintainability is noticeable [16]. Therefore, the assumption is made here that higher encapsulation also increases modularity, reusability, modifiability, and maintainability overall.

Coupling. Coupling describes the dependency of two components, whereby in general, the lowest possible coupling should be achieved. Thus, the effects of changes of a component are reduced in the dependent components [21]. Also, tight coupling complicates the analyzability of the software. Therefore, with increasing coupling, the maintainability of the software decreases.

Cohesion. Cohesion describes the degree to which the elements of a component are interrelated. As cohesion increases, the coupling between the individual components usually decreases [21]. Therefore, rising cohesion generally supports the maintainability of the product.

Size and Complexity. In the following, also metrics are considered, which are assigned to the design properties size and complexity. The assumption is made that with increasing size, especially the software's analysability and modifiability deteriorates while increasing complexity also impairs reusability.

6.2 Transfer of OO concepts to Shiny

Our aim is to analyze our Shiny app according to widely used OO software quality metrics [18]. Because in general the application modules in Shiny do not fully follow the the OO paradigm, some assumptions have to be made to calculate the metrics. For this purpose, a definition of the fields and methods of different encapsulation levels is made for the Shiny framework. Only the elements of a class that are relevant in the metric selection of Terragni et al. [23] are considered. Hence, for transferring the class elements of the OO paradigm to the Shiny framework in a meaningful way, the *class elements*, their functionality and possible encapsulation levels are presented next.

An object or instance has a defined state (defined by attributes or fields) and a defined behavior (defined by methods). In general, the behavior depends on the state, which means that the methods access the attributes. A class defines for a collection of objects the structure, the behavior and the relationship to other classes. Static elements of a class are the same for all instances of that class. Static methods can only be applied to the whole class and not to a single instance. Usually, they are used to assign a new value to class attributes without the influence of an instance or whenever an operation applies to all or several instances. The following

OO paradigm	Shiny framework
Fields	
Static fields	Fields with equal value for all instances; lockBinding in the environment
Private fields	Reactive Values (including Input object) defined inside application or module
Protected fields	-
Public fields	Fields declared in the R script outside of functions; not locked in the environment
Methods	
Static methods	-
Private methods	Functions defined inside server function
	Reactive expressions
	Observers (including render functions)
Protected methods	-
Public methods	Functions defined in the R script outside of the server function
	If application viewed: only server function
	If module viewed: UI and server function

Table 2: Mapping of OO class components to Shiny applications and modules

three levels of encapsulation are differentiated: private elements are accessible only within the class itself; protected elements are accessible in the class itself, in its subclasses and, depending on the programming language, in classes within the same package; public elements are accessible by all classes. The following statements can be made by transferring these definitions to the Shiny framework. Instances of a Shiny application or module are distinguished from each other by using different namespaces. An instance of an application or module also consists of a state and behavior. The state is primarily defined by input elements used by the defined behavior to calculate the output elements. The definition of the application or module determines the state and the behavior for a collection of instances. *Static fields* should have the same value for all instances of an application or module. *Static functions* should be applicable to all instances. The latter is not implementable for Shiny applications and modules because the principle of object management is not realized by default. However, the object management could be achieved using static attributes, but this was not implemented in the exemplary software, so that static methods in the Shiny framework will not be considered in further discussion. The distinction of instances based on the namespace allows two stages of encapsulation to be defined: Private elements are only accessible within their own namespace, and public elements are also accessible outside the namespace. The encapsulation level *protected* of the OO paradigm has no correspondence in Shiny since inheritance is not implementable in this framework.

We now discuss how the individual elements from the OO paradigm are mapped to elements of applications and modules of the Shiny framework base on these assumptions; an overview of this mapping is provided in Table 2.

It is assumed that *private fields* correspond to reactive values defined within the application or module in the Shiny framework. Reactive values are used in the defined behavior of the application or module to generate the output. Moreover, they are not accessible outside the namespace of the application or module. Fields that are defined outside of a function in the script of an application or module and are not locked in the global environment are seen as *public fields*. These elements can also be addressed outside the namespace itself. They might also be assumed as static, but any instance can change the value of the field. This may not affect all other instances of this application or module that already exist, especially in other sessions. Therefore, only fields with the same value for all instances and are locked in the environment via the lockBinding function are assumed to be *static fields*. A field has the same value for all instances if the value does not depend on any reactive values or method parameters of the functions. Locking the field prevents the assignment of a new value. Thus, the definition of static variables in the Shiny framework is stricter than in the OO paradigm, but otherwise, the equality of the field's value for all instances cannot be ensured. *Private functions* in the Shiny framework correspond to functions defined within the server function of an application or module, reactive expressions and observers, including render functions. These functions define the behavior of the application or the module by calculating the output in dependency of the state of an instance. Functions defined in the server function are not accessible outside of the namespace of the application or module. Observer and render functions can only be defined inside other functions, so they are automatically not callable from the outside. Also, they are not explicitly invoked in the program code anyways but are executed based on reactivity. Finally, *public functions* are functions defined in the R script outside of any other functions. For Shiny applications, this does not include the server function since this cannot be called outside the application in any meaningful way, except to create a new instance. For Shiny modules, the UI and server functions are also included as they provide the module's interface and are thus called by other modules or applications.

7 SOFTWARE QUALITY ANALYSIS

This section examines how the software quality characteristic maintainability, which was defined in Section 3, is affected by the modularization of Shiny applications by comparing modularized and non-modularized version of our application. With the help of the calculation of software quality metrics, the effect of the modularization of Shiny Apps on maintainability is considered quantitatively. These metrics are mostly derived from the OO paradigm, so they must be partially adapted to the Shiny paradigms. As such, a formal evaluation of the use of the Shiny framework for the development of larger, more complexly structured applications is made. We rely here only on very specific metrics for a static code analysis – which is an established method [9, 11, 23] – due to the fact that we compare two versions of our application with a fixed feature scope. An advanced assessment could also use a cost model based on change rate of the code; in [2] the authors argue that this cost model enables the prediction of future development costs.

7.1 Metrics calculation

Based on the correspondences between Shiny and the OO paradigm just discussed, the effect of modularizing Shiny applications (see Section 5) is examined. For this purpose, the presented application for gene expression analysis was implemented from an earlier development state both with modularization and without (monolithic). Any comments or other code documentation were removed in the source codes of both applications to increase their comparability; this is based on the assumption that differences in comments do not influence maintainability. Since both applications represent the same functionality, the actual calculation logic was abstracted into functions called by both applications. In the following, these methods are referred to as functionality methods or functions. Thus, the source code to be compared contains as far as possible only instructions concerning the UI as well as its control. For the quantitative comparison, metrics were calculated, which were mostly originally developed to evaluate classes of OO programming languages [23]. First, it will be discussed which of the metrics will not be calculated. The number of bytecode instructions (NBI) was used in [23] because applications developed in Java were considered. Compiling R program code into bytecode is possible, but an interpreter is used by default. Therefore, this metric is not calculated. We compare both our applications (monolithic versus modularized) without considering the code documentation, thus it would not be useful to calculate the lines of comment (LOCCOM) metric. The number of static methods (NSTAM) cannot be calculated since it is not possible or reasonable to declare a method of an application or a module as static in Shiny. Moreover, the concept of inheritance is not implementable in Shiny applications or modules, so any metrics that are based on inheritance cannot be calculated. This includes the depth of inheritance tree (DIT), the number of children (NOC), the measure of functional abstraction (MFA), the inheritance coupling (IC) and the coupling between methods (CBM). There is no equivalent in the Shiny framework to the encapsulation level protected used in the OO paradigm, as explained earlier. Therefore, the number of protected methods (NPROM) cannot be calculated.

Next, the metrics that were calculated for the sample applications are presented. The details of how they were calculated will also be discussed. Table 3 provides an overview of the calculated metrics and a brief description.

The lines of codes (LOC) metric counts the number of non-blank lines of the application or the module. The IntelliJ Plugin MetricsReloaded was used for the calculation of the LOC. This metric is used to evaluate the design property *size* and, therefore, has a low value. This applies to all metrics of this design property. The number of public methods (NPM), the number of fields (NOF) and the number of static fields (NSTAF) were calculated by counting the number of corresponding elements in the application or the module. Terragni et al. do not define which encapsulation should be considered for the calculation of the NOF. It is assumed that only public fields are counted since a separate metric exists for private fields. The number of method calls (NMC) is the sum of the number of method calls internal (NMCI) and the number of method calls external (NMCE). Thereby, the NMCI was calculated by counting all invocations of a function defined in the application or module itself. The NMCE is the number of all function invocations defined in another module or an external R package. Also, the calling of functionality methods is included in this metric.

Design Property Name		Description
Size	Lines of Code (LOC)	Number of non-blank lines
	Number of Public Methods (NPM)	Number of public functions in an application or module
	Number of Fields (NOF)	Number of public fields in an application or module
	Number of Static Fields (NSTAF)	Number of static fields in an application or module
	Number of Method Calls (NMC)	Number of function invocations
	Number of Method Calls Internal (NMCI)	Number of function invocations of function defined in the application or module
	Number of Method Calls External (NMCE)	Number of function invocations of function defined in other modules or packages
Complexity	Weighted Methods per Class (WMC)	Sum of the Cyclomatic Complexity of all function in the application or module
	Average Method Complexity (AMC)	Average of the Cyclomatic Complexity of all function in the application or module
	Response For a Class (RFC)	Number of functions that response to a message from the application or module itself
Coupling	Coupling Between Object classes (CBO)	Number of other modules or packages that an application or module is coupled to
	Afferent Coupling (Ca)	Measure of how many other applications or modules use the specific application or module
	Efferent Coupling (Ce)	Measure of how many other modules or packages are used by the specific application or module
Cohesion	Lack of Cohesion in Methods (LCOM)	Difference between the number of function pairs without and with common non-static fields
	Lack of Cohesion Of Methods (LCOM3)	Revised version of LCOM
	Cohesion Among Methods in class (CAM)	Represents the relatedness among functions of an application or module
Encapsulation	Data Access Metrics (DAM)	Ratio of the number of private fields to the total number of fields
	Number of Private Fields (NPRIF)	Number of private fields of an application or module
	Number of Private Methods (NPRIM)	Number of private functions of an application or module

Table 3: Descriptions of the calculated (class) metrics [23]

Next, metrics for quantitative assessment of the design property *complexity* are presented. For all metrics, a low value corresponds to low complexity. The weighted methods per class (WMC) is calculated by the summation of the Cyclomatic Complexity [19] of all functions defined in the application or module, which counts the linear-independent paths within a program. The R package cyclocomp was used to calculate the Cyclomatic Complexity of a single function. The metric average method complexity (AMC) is the average of the Cyclomatic Complexity of all functions in the application or module. Therefore, the AMCm of the application or module m is calculated by the formula $AMC_m = WMC_m / NPM_m$. The response for a class (RFC) is the number of functions that respond to a message from the application or module itself. It is calculated by counting the number of distinct functions that are invoked by the application or module, no matter if the function was defined inside the application or module itself or not. Functions called multiple times are counted only once [8].

The metrics for measuring the property *coupling* are presented next. Low values are indicative of loose coupling, while high ones indicate tight coupling. The coupling between object classes (CBO) is calculated by counting the distinct afferent and efferent modules or packages of an application or module [8]. Since the example applications do not have circular dependencies, this is the same as the sum of the afferent coupling (Ca) and the efferent coupling (Ce). Ca is calculated by counting the number of modules or applications that call the target module's functions or application. On the other hand, Ce is calculated by counting the modules or packages from which the target application or module calls functions. For example, if the application A invokes a function of module B, the Ca of B, as well as the Ce if A, increases by one. In the considered example, no packages are included for the calculation of Ca since packages usually do not call any functions of a Shiny application or module. In the calculation of Ce, applications were not regarded because no application or module calls another application.

Now, the metrics for evaluating the design property *cohesion* are discussed. The possible and desired values differ between the metrics. The lack of cohesion in methods (LCOM) corresponds to the difference between the number of function pairs without and with common non-static fields. The LCOM of an application or module m is calculated by $LCOM_m = b - c$, where b equals the number of function pairs that do not reference to similar non-static fields and c equals the number of function pairs that do reference to at least one similar non-static field [8]. Fields that are referenced indirectly by the function of interest via invoked functions are also counted. If the result is negative, LCOM is set to 0. This metric's value should be as low as possible, whereby a value of zero indicates a cohesive application or module. However, the evaluation of a value depends on the total number of defined functions in a component. Therefore, the metric LCOM3 was developed to address this problem of the lack of possibility of comparison. The LCOM3 of an application or module m is calculated by $LCOM3_m = ((x - f \cdot a))/((a - f \cdot a))$, where f equals NPM_m, a equals the sum of NOF_m and $NPRIF_m$ and x equals the sum of the number of referenced non-static fields of all functions. If only one function is defined in an application or module or has no non-static fields, LCOM3 cannot be calculated and is set to 0. The value of LCOM3 is always between 0 and 2, where 0 indicates a high cohesion. Values above 1 are critical since this shows the existence of values that are not accessed by any function within this application or module (Henderson-Sellers, 1996, p. 147). The metric cohesion among methods in class (CAM) of an application or module m is calculated by $CAM_m = p/((y+f))$, where p equals the sum of the number of different types of method parameters of each function defined in the application or module, f equals the sum of NPM_m and $NPRIM_m$ and y equals the number of distinct types of method parameter of all functions in the application or module. A type of a parameter is thereby the class a parameter should inherit, for example, a list, an integer or a data frame. Also, local variables defined in the surrounded function of a private function are considered to be parameters. The range of CAM is between 0, which indicates low cohesion and 1, indicating high cohesion [3].

Finally, the calculated metrics for the assessment of the design property *encapsulation* are presented. The number of private fields (NPRIF) and the number of private methods (NPRIM) is calculated by counting the corresponding elements in the application or module. The data access metrics (DAM) correspond the ratio of NPRIF to the total number of fields defined in an application or module.

7.2 Results of metrics and evaluation

The explained metrics were calculated for the modularized and non-modularized software. For the non-modularized software, the metrics were calculated collectively for the scripts run.R, ui.R, and server.R. These three scripts of the modularized software were also calculated as a group referred to as application. For modules of the modularized software, the metrics were calculated individually in each case. The sum and average of the values of the individual modules were calculated for each metric to optimize the comparability. For the design properties of coupling and cohesion, the calculation of the sum of the modules is not meaningful because these are a measure of the dependency on a component or within the component itself.

The differences in results between the modularized and non-modularized software are now considered for each design property. Thereby the relationship to the sub-characteristics of the maintainability is again addressed. An overview of the results is shown in Table 4. For the modularized software, in addition to the sum and the average, the results of only the application (app) itself are also shown.

The sum of the metrics results associated with the design property size is mostly larger for the modularized software than the non-modularized software results. Exceptions are the NOF and the NMCI and the NSTAF, for which there is no difference between the considered software. Looking at the average of the individual modules, they are all smaller than the monolithic application. The larger sum of the modularized application can be explained because at least two public methods are defined as almost every module interface. This increases the NPM, the NMCE and therefore also the NMC and the LOC. With the increasing size of software, the analyzability and thus modifiability decreases. The metrics of the design property size indicate worse maintainability of the modularized application as a whole in comparison with the monolithic software. However, the individual components are substantially more compact and individually better maintainable. The same statement can be made for the calculated metrics of complexity. The

Design Property	Metric	Modularized			Not Modularized
		Only App	All Modules (Sum)	All Modules (Avg)	
Size	LOC	68	1067	50.81	865
	NPM	0	54	2.57	0
	NOF	0	0	0	2
	NSTAF	0	1	0.05	1
	NMC	48	934	44.48	858
	NMCI	10	17	0.81	40
	NMCE	48	917	43.67	818
Complexity	WMC	8	210	10	148
	AMC	1.6	1.76	1.76	1.78
	RFC	29	207	23.05	168
Coupling	CBO	9	-	4.33	13
	Ca	0	-	0.95	0
	Ce	9	-	3.38	13
Cohesion	LCOM	0	-	7.9	2593
	LCOM3	0.59	-	0.34	0.96
	CAM	0.6	-	0.44	0.07
Encapsulation	DAM	1	0.99	0.99	0.95
	NPRIF	8	81	3.86	63
	NPRIM	5	65	3.1	83

Table 4: Results of the calculated metrics for the application in the non-modularized and modularized design

sum of the individual modules indicates a higher complexity of the entire modularized software than the non-modularized software. On average, however, the individual modules are again less complex than the non-modularized software. The increased WMC could also be caused by the increased number of functions, since each function has a Cyclomatic Complexity of at least 1. Therefore, the AMC is more meaningful for assessing the complexity, which indicates that the modularized software functions are marginally less complex. When calculating the sum of the RFC, the called functions were only counted once across all modules. Again, this higher value can be explained by the higher number of functions in the modularized software. Also, concerning complexity, the statement can be made that the modularized application has overall worse maintainability than the non-modularized software. Once again, however, the single modules are less complex and better maintainable. As already explained initially, only the average value of the modules is compared with the monolithic software for the properties coupling and cohesion. Thereby all computed values show a looser coupling as well as higher cohesion with the modularized software. The only exception is Ca, but this is since the monolithic software as an application cannot be called from outside. As discussed before, looser coupling increases all sub characteristics of maintainability. The in-creasing cohesion improves the loose coupling additionally and therefore has a positive effect on the maintainability too. Finally, the metrics used to evaluate the software design property encapsulation are evaluated. Here, it has to be said that encapsulation is unnecessary for monolithic architectures since the application is not called from the outside. Nevertheless, the calculated metrics indicate good encapsulation of the modules because especially DAM has a very high value. This increases the sub-characteristic modularity, reusability and modifiability. Altogether based on the metrics, the modularized software as an entire system is larger and more complex than the monolithic software. This indicates a worse analysability, modifiability, reusability and thus maintainability in total. However, the individual modules could also be regarded separately from each other, whereby the computed metrics indicate substantially higher maintainability. In addition to the quantitative evaluation, the results are now also discussed via a qualitative approach. Although modularization increases the overall size and complexity of software, it also improves its maintainability. However, this depends on the modularization implementation – a correct amount must be defined to have cohesive modules on the one hand and loosely coupled modules on the other hand. This allows a significant improvement of the reusability, analysability and therefore also modifiability of the software. Also, a well-chosen modularization offers the advantage of allowing the modules to be considered separately from one another. The regarded section of the software during the maintenance becomes much smaller and less complex than with a monolithic application. Also, the effect of changes within a module in other modules can be estimated and intercepted much better. Modules can be called multiple times in one or more applications, minimizing or even preventing redundant program code. Altogether with a qualitative view, the modularization simplifies the maintainability of the software substantially.

8 DEPLOYMENT

In a modern DevOps context pipelines usually concern automation from source code to the deployment of the finished product. Pipelines are omnipresent in today's IT landscape. From data pipelines to integration and testing pipelines up deployment pipelines, the term finds broad acceptance and a wide definition. In our system architecture, the steps considered are building, testing, packaging, and deploying a piece of software. We followed this DevOps paradigm in our software development process as follows.

The web service is executed in a Docker container on a server provided by the Fraunhofer ITEM. This server has 48 cores and 256 GB random access memory (RAM) and thus provides sufficient computing power to analyze the gene expression data. The program code is stored in a GitLab repository. A Dockerfile is used to define the build and deployment of the service and define and install prerequisites and dependencies. In turn, the building and pushing of the image into the Docker registry is controlled via a GitLab CI/CD file. This file enables continuous integration (CI) and continuous deployment (CD), meaning running a defined pipeline as well as deploying the application to production whenever a modification is made. The pipeline consists of a test stage for checking the Dockerfile and a deploy stage for deploying the service to the server. Using Portainer, a dashboard for the management of Docker containers, the container for deployment is created and maintained.

9 CONCLUSION

The Shiny applications found in the literature are often quite small and used as dashboards for data visualization. For handling the challenge of developing large applications, we investigated whether this framework is suitable for the development of complex applications.

Our analysis focuses on the formal evaluation of the maintainability of Shiny applications. We provided a comparison of modularized and non-modularized application version and followed a quantitative approach based on chosen software quality metrics. Quantitatively, modularized applications are larger and more complex than monolithic applications. However, the modules can be considered independently so that the resulting maintainability of modularized applications is rated better overall. Most software quality metrics found in the literature are developed to evaluate object-oriented (OO) principles. Therefore, a subgoal of our article was the investigation of the influence of OO principles on the maintainability of software. To be able to transfer these principles to Shiny, the equivalence of Shiny application elements to elements from the OO paradigm needed for the computation of the metrics was examined. For this, the purpose of the OO elements and components of Shiny with the same or similar purpose were investigated. Based on these assumptions, the software quality metrics were calculated. Our application is located in the biomedical realm; hence the choice for Shiny is rooted in the availability of libraries for the backend pipeline in the R programming language. We believe that our discussion generalizes to other web applications that follow a workflow character and that relies on tabs in the web pages for which Shiny components could be reused. Nevertheless, other domains that rely less on specific R backend libraries and that might have a focus on the microservice paradigm might profit more from alternative web frameworks (for example, based on Javascript).

In future work, we aim to add capabilities to execute workflows on other input file formats. A major research question in this regard is whether the modularization of our software proves to support the modifiability aspect.

ACKNOWLEDGEMENTS

This work was supported by the Fraunhofer Internal Programs under Grant No. Attract 042-601000.

REFERENCES

[1] Affymetrix, Inc. 2019. Transcriptome Analysis Console (TAC) 4.0.2 User Guide.

[2] Tibor Bakota, Péter Hegedűs, Gergely Ladányi, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. 2012. A cost model based on software maintainability. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 316–325.

[3] Jagdish Bansiya and Carl G. Davis. 2002. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on software engineering* 28, 1 (2002), 4–17.

[4] Verónica Bolón-Canedo and Amparo Alonso-Betanzos. 2019. *Microarray Bioinformatics.* Springer.

[5] Alvis Brazma, Pascal Hingamp, John Quackenbush, Gavin Sherlock, Paul Spellman, Chris Stoeckert, John Aach, Wilhelm Ansorge, Catherine A Ball, Helen C Causton, et al. 2001. Minimum information about a microarray experiment (MIAME)—toward standards for microarray data. *Nature genetics* 29, 4 (2001), 365–371.

[6] Terence A Brown. 2018. *Genomes 4th edition.* Garland science.

[7] Seth Carbon, Eric Douglass, Benjamin M Good, Deepak R Unni, Nomi L Harris, Christopher J Mungall, Siddartha Basu, Rex L Chisholm, Robert J Dodson, Eric Hartline, et al. 2021. The Gene Ontology resource: enriching a GOld mine. *Nucleic Acids Research* 49, D1 (2021), D325–D334.

[8] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.

[9] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. 1994. Using metrics to evaluate software system maintainability. *Computer* 27, 8 (1994), 44–49.

[10] Ron Edgar, Michael Domrachev, and Alex E Lash. 2002. Gene Expression Omnibus: NCBI gene expression and hybridization array data repository. *Nucleic acids research* 30, 1 (2002), 207–210.

[11] John E Gaffney Jr. 1981. Metrics in software quality assurance. In *Proceedings of the ACM'81 conference.* 126–130.

[12] Tamas Galli, Francisco Chiclana, and Francois Siewe. 2020. Software Product Quality Models, Developments, Trends, and Evaluation. *SN Computer Science* 1, 3 (2020), 1–24.

[13] R Gentleman and Wolfgang Huber. 2008. Processing Affymetrix Expression Data. In *Bioconductor Case Studies.* Springer, 25–45.

[14] Robert C Gentleman, Vincent J Carey, Douglas M Bates, Ben Bolstad, Marcel Dettling, Sandrine Dudoit, Byron Ellis, Laurent Gautier, Yongchao Ge, Jeff Gentry, et al. 2004. Bioconductor: open software development for computational biology and bioinformatics. *Genome biology* 5, 10 (2004), 1–16.

[15] Hinrich Gohlmann and Willem Talloen. 2009. *Gene expression studies using Affymetrix microarrays.* CRC Press.

[16] International Organization for Standardization. 2016. *Systems and Software Engineering: Systems and Software Quality Requirements and Evaluation (SQuaRE): Measurement of System and Software Product Quality.* ISO.

[17] Bernd Klaus and Stefanie Reisenauer. 2016. An end to end workflow for differential gene expression using Affymetrix microarrays. *F1000Research* 5 (2016).

[18] Michele Lanza and Radu Marinescu. 2007. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer Science & Business Media.

[19] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[20] National Center for Biotechnology Information. [n.d.]. https://www.ncbi.nlm.nih.gov/geo/info/geo2r.html.

[21] Alexander Schatten, Stefan Biffl, Markus Demolsky, Erik Gostischa-Franta, Thomas Östreicher, and Dietmar Winkler. 2010. *Best practice software-engineering: Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen.* Springer-Verlag.

[22] Paul Teetor. 2011. *R cookbook: Proven recipes for data analysis, statistics, and graphics.* O'Reilly Media, Inc.

[23] Valerio Terragni, Pasquale Salza, and Mauro Pezzè. 2020. Measuring Software Testability Modulo Test Quality. In *Proceedings of the 28th International Conference on Program Comprehension.* 241–251.

[24] Persis Voola and A Vinaya Babu. 2013. Comparison of requirements prioritization techniques employing different scales of measurement. *ACM SIGSOFT Software Engineering Notes* 38, 4 (2013), 1–10.

[25] Guangyan Zhou, Othman Soufan, Jessica Ewald, Robert EW Hancock, Niladri Basu, and Jianguo Xia. 2019. NetworkAnalyst 3.0: a visual analytics platform for comprehensive gene expression profiling and meta-analysis. *Nucleic acids research* 47, W1 (2019), W234–W241.