The Software Architecture of CraneFoot

- An independent analysis by the Fraunhofer Center for Experimental Software Engineering, Maryland, USA

Dharmalingam Ganesan (dganesan@fc-md.umd.edu)

Executive Summary

This report presents an architecture analysis of the pedigree software Cranefoot. The analysis was conducted by the Fraunhofer Center for Experimental Software Engineering, Maryland (CESE), USA[1] as part of an evaluation of open source pedigree software. The initial goal of that analysis was to evaluate the CraneFoot tool from the perspective of code quality and ease of integration and change. Since the analysis might be useful for other users of Cranefoot we decided to make it publicly available.

A quote from the CraneFoot web-site[2]: "The principle behind the design of CraneFoot was simple: Quality before quantity. Scientists, like I, often make the mistake of adding features as soon as they get the program to run correctly for some input. This leads to incomprehensible and unmanageable sprawling code that is of little use to anyone. I have actively avoided such a tendency - and released the source code - with the hope that other people can also enjoy the results."

The CraneFoot tool works well, but like many other open source software, it lacks architecture and requirements documentation. This report tries to address that issue by providing some software architectural views and metrics extracted from the CraneFoot implementation. We used the SAVE tool[3] [5] to semi-automatically extract architectural views from the source code (written in the C/C++ language) of CraneFoot. We present analyses and experiences of analyzing the CraneFoot tool.

Keywords: software architecture, view-based software architecture, metrics, pedigrees.

[1] http://fc-md.umd.edu
[2] http://www.artemis.kll.helsinki.fi/cranefoot/
[3] http://www.theSAVEtool.com

Table of Contents

1 Introduction

1.1 Problem Statement

Our partner Biofortis[4], USA, currently uses a commercial pedigree visualization product in their Labmatrix software product. However, the users are not pleased with the visualization quality of that commercial product. More specifically, that commercial product does not adequately visualize complex family trees. Hence, in order to improve the user's satisfaction, our partner decided to search for alternatives. To this end, they wanted to investigate whether the CraneFoot open source pedigree visualization tool is a good alternative for visualizing complex family pedigrees.

However, our partner was not sure about the ease-of-change with respect to adding new requirements or modifying existing ones in CraneFoot. Therefore, as part of the BRISP they requested Fraunhofer Center for Experimental Software Engineering Maryland to perform a systematic analysis of the code quality with respect to ease-of-change.

In order to analyze the Cranefoot software from these perspectives we produced the following architecture views: conceptual view, structural view, behavioral view, and implementation view of Cranefoot. It is our hope that developers and maintainers will find the architectural views useful for enhancing the Cranefoot tool for their purposes.

1.2 Challenges

In this project, we faced a number of challenges. First, we lacked specific domain knowledge about Pedigrees. Second, apart from a user manual, the CraneFoot tool had no design documents. Third, it was not possible to have face to face meetings with the developer, because CraneFoot is an open source tool and the developer was far away (in Finland).

To overcome these challenges, an architecture-centric approach was used. In this report, the details of the approach and the results of architecture and code quality assessments are presented.

[4] http://www.Biofortis.com

2 View-based Architecture Documentation

Software architectures encompass structural and behavioral properties of software systems and their relation to their environments [1][2][3]. One architecture view is typically not enough to completely describe the architecture of a software system so several views or perspectives are needed. This results in documentations that consist of multiple architectural views. Each architectural view is an abstraction from the described software system(s), but different architectural views abstract from different details or aspects.

An architectural view prescribes the types of components and the types of relationships used to describe the software system (i.e., the connectors), as well as properties of these component and connector types. Consequently, each architectural view presents different information, is used by different stake-holders, and addresses different concerns.

Architectural views evolve over time. The types of components and the types of relationships used to describe the software system, as well as properties of these component and connector types change over time when realizing new products, new or changed requirement. Consequently, each architectural view is updated in architecture development to reflect these different or modified concerns.

The next subsections give an overview of the architecture of the CraneFoot tool using a set of architectural views, namely the conceptual view, the structural view, the behavioral view, and the implementation view.

2.1 Conceptual View

The conceptual view is the most abstract architectural view used for describing the architecture of a system. This view is closest to the application domain and independent of solution aspects, like software implementation technologies or hardware techniques. As such, it provides a very brief overview of a system and is interesting for all stakeholders. It is, however, especially suited for marketing, users, customers, and everyone interested in a high-level overview of the implementation view and not having detailed technical knowledge. Furthermore, it can be a key facilitator to interact with domain experts, who are not interested in the details of the software system, but what the system does in terms of domain concepts.

The conceptual view captures the application domain by mapping the functionality of the system to conceptual components and showing data stores, external interfaces, and hardware devices. It also depicts the interplay and relationships among the various elements. The view largely differs from the other architectural views in its scope, its structure, and the concerns it addresses. Basically, the conceptual view addresses the following concerns:

- **Functional requirements**. How does the system meet its requirements and what functionality is actually provided by the system? How is the impact of changes in requirements or the domain minimized?
- **Interfaces provided by the system**. Which external interfaces are provided by the system or its hardware components and which actors interact with the system? How is the system connected to its environment?
- **Domain-specific hardware/software integration**. How is domain-specific hardware and software incorporated into the system and how does it interact with the rest of the system?

The conceptual view is represented by using an UML-based notation with appropriate stereotypes for the various conceptual elements. Table 1 summarizes the elements and relations used for representing the conceptual view.

UML Element	Description
Component Name	Conceptual component
—○	External interface
→	Usage dependency
⬭	Conceptual interface group

Table 1 – Elements and relations used in the conceptual view

2.1.1 Process for Constructing a Conceptual View

As noted earlier, Fraunhofer Center Maryland had no earlier experience with the domain of pedigrees. Therefore, it was a non-trivial task to sketch a conceptual view of CraneFoot. In order to construct a meaningful conceptual view, appropriate domain abstractions should be known up-front. It is our position that just from the source code conceptual views (or other architectural views) cannot be constructed, because the source code is nothing but a collection of algorithms and data processing with a lot of details. To make sense of the source code, certain level of domain expertise is critical and in this case first had to be acquired. Thus, in contrast to most reverse engineering literature, we emphasize the role of domain knowledge in extracting meaningful architectural views from the source code.

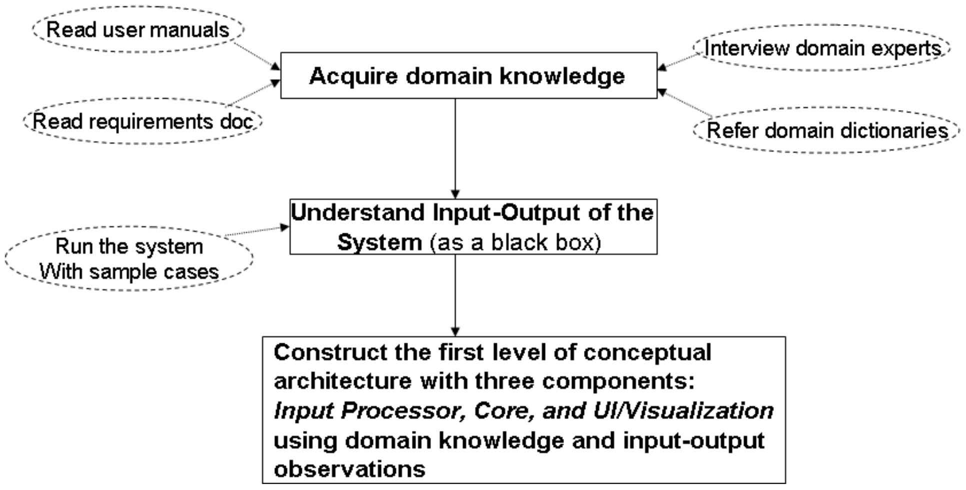

Read user manuals
Read requirements doc
Acquire domain knowledge
Interview domain experts
Refer domain dictionaries
Run the system With sample cases
Understand Input-Output of the System (as a black box)
Construct the first level of conceptual architecture with three components: Input Processor, Core, and UI/Visualization using domain knowledge and input-output observations

Figure 1 - Process for Constructing Conceptual View

Our process for extracting the conceptual view (see Figure 1) consists of three steps.

The goal of the first step is to acquire domain knowledge through reading the available user manuals, requirements documentation, referring domain dictionaries, and interviewing domain experts. This step is not necessarily coupled with a single system under study; instead it covers nearly all systems in the same domain, resulting in a domain level understanding instead of implementation specific understanding of a single system. Hence, we stress the point that this is the most crucial step in the whole process for constructing conceptual views.

The goal of the second step is to understand a particular system in the same domain by observing its input space and corresponding outputs. Although such a mapping between inputs and outputs could be, in theory, established statically through source code reading, it is almost next to impossible for non-trivial systems. Thus, in order to save time and effort, one could run the system with sample inputs and observe the output. This activity will result in a better understanding of a) requirements covered by the system under study, b) details related to input and output formats, and c) details related error messages produced as output (if any), and hence the types of input data validation done by the system, and the rules the input data should obey.

The goal of the last step is to construct a high-level conceptual architecture. At this level, the conceptual architecture is made of three components, namely the input data processor, core, and output or visualization. The data flows from the input data processor to the core and finally to the user.

2.1.2 Application to CraneFoot System

We now explain how these steps were applied to CraneFoot.

Step 1 – Acquire Domain Knowledge. We used wikipedia to get a brief introduction to Pedigrees. The focus was to understand the basics of pedigrees and why they are important. We also had a brief discussion with our partner who already uses another pedigree visualization tool into their product. In addition, we also read the user manual supplied with the distribution of the CraneFoot tool.

Step 2 – Understand Input-Output of the System. Using the user manual of the CraneFoot tool and some pedigree samples, we ran the CraneFoot tool and observed input-output pairs. The CraneFoot tool accepts as an input a configuration file with support for a lot of configuration parameters. We analyzed a sample configuration file and improved our understanding of CraneFoot's configuration capabilities.

Step 3 – **Construct a conceptual architecture**. Based on our understanding of the input-output of CraneFoot, we decomposed the conceptual architecture into three major components: a) Input parser, which parses the input files and checks syntax errors in the input data, b) Core components, which analyze the input data for semantic errors, manages configuration parameters, and implements force-directed layout algorithms and simulated annealing optimization techniques, and c) the visualization or printing component, which outputs pedigrees in a postscript format.

Using the above three steps, we sketched our own conceptual view of CraneFoot as shown in Figure 2.

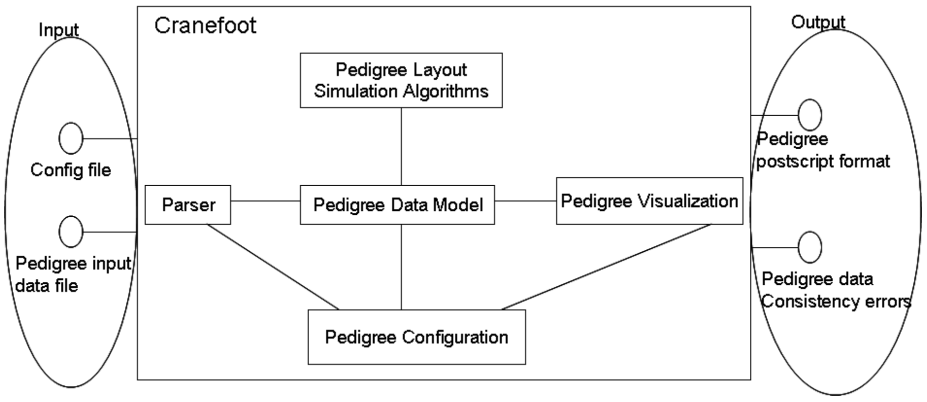


Figure 2 – Conceptual View on CraneFoot

Table 2 and Table 3 describe the external interfaces and the conceptual components of the CraneFoot tool, respectively.

Interfaces	Responsibilities
Input	The CraneFoot tool requires as an input a configuration file that contains the name of the input data file, parameters, such as font size, pattern types, icon shapes, etc., related to visualization of pedigrees.
Output	The CraneFoot tool outputs a postscript file that contains the pedigree graph. If there are data consistency errors (e.g., brother marrying sister) in the input data then the tool produces appropriate error messages.

Table 2 – External Interfaces

Conceptual Components	Responsibilities
Parser	The parser component is responsible for parsing the input pedigree data file and invoke the pedigree data model component
Pedigree Data Model	The pedigree data model is basically a collection of data structures that contain model family tree. The elements of the model are, family, member, branch, etc.,
Pedigree Layout and Simulation Algorithms	The CraneFoot tool uses force-directed simulation based layout algorithms for drawing pedigrees. The layout and simulation algorithms component implements mathematical logic for aesthetically pleasing visualization
Pedigree Visualization	The CraneFoot tool produces a postscript file as an output. The pedigree visualization component contains a collection of postscript modules corresponding to various elements (e.g., nodes, edges, colors, patterns) for drawing pedigrees.
Pedigree Configuration	The CraneFoot tool supports various input and output configurations, which are managed by the pedigree configuration component and used by other components.

Table 3 – Conceptual Components

2.1.3 The Structural View

The structural view describes the functional decomposition of the system and captures the static structure in terms of subsystems and components, the interfaces provided by them, and the relationships between the various elements. It addresses how the conceptual solution provided in the conceptual view is addressed with available software platforms and technologies. Note that the structural view only describes the static structure of a system and therefore does not provide any information about dynamic aspects and behavior. Contrary to the conceptual view, the structural view is quite detailed and relatively close to the implementation. It is therefore especially interesting for technical stakeholders such as developers and project managers, as it can be directly used, for example, for work assignments. Together with the behavioral view the structural view provides a good basis for assessing an architecture regarding the fulfillment of quality requirements like reusability, maintainability, and extensibility. To a certain degree, however, the structural view can also be interesting for other stakeholders.

The structural view addresses the following concerns:

- Mapping of the solution to a software platform
- Realization of the functionality using the available technologies
- Provided and required services
- Testing support
- Dependencies among components
- Reuse of components and/or subsystems

The structural view is represented using a structural model for the decomposition of the system into subsystems, components, and interfaces (e.g., UML class diagram with appropriate stereotypes and packages for components and subsystems). To depict the elements of the structural view that are not directly supported by the Unified Modeling Language, such as sub-systems or components, stereotypes are used. A subsystem, for example, is represented as a stereotyped UML package, which is drawn as a box with a tab attached to the upper left edge of the box. To indicate that it is actually a subsystem rather than a normal package, the stereotype «subsystem» pre-cedes the name of the subsystem. Likewise, components are modeled by the UML class symbol stereotyped with «component». Finally, the UML us-age notation has been used to describe use dependencies between elements.

Table 4 summarizes the elements of the Unified Modeling Language (UML) used for representing the structural view.

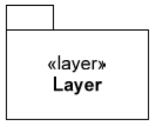
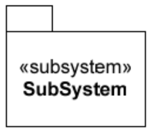
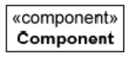
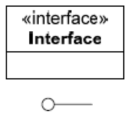
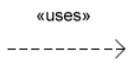
UML Element	Description
	Layer
	Subsystem
	Component
	Interface
	usage dependency

Table 4 – Elements and relations of the structural view

2.1.4 CraneFoot and External Component Dependencies

The quality of software not only depends on its own implementation but also on its external dependencies. Thus, it was important to evaluate not only the quality of CraneFoot but also its external dependencies. Moreover, legal issues related to licensing of all its dependencies should be clarified before adopting the tool.

As the first step, we applied the SAVE tool [4][5] to extract CraneFoot's external dependencies. As shown in Figure 3, the CraneFoot implementation depends only on standard C header files and libraries. Hence, we focused our architecture and code analysis effort on CraneFoot alone.

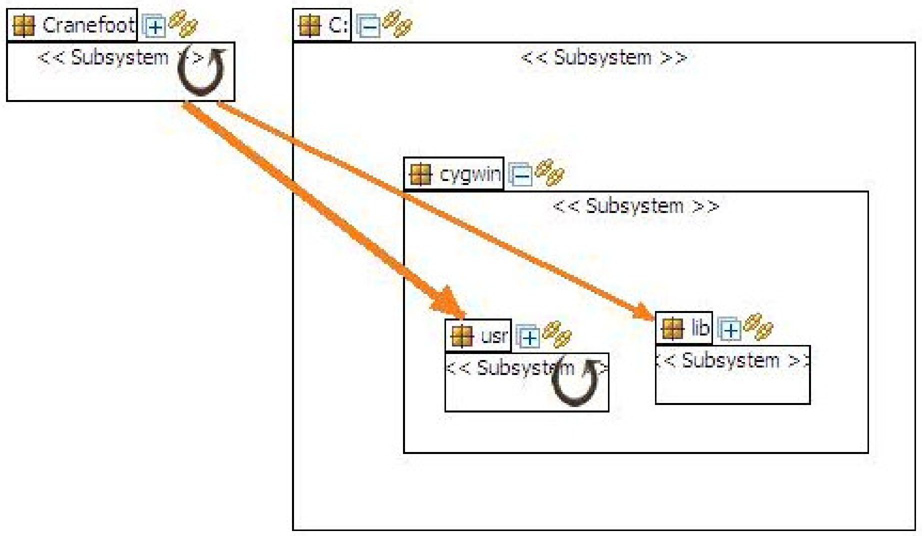


Figure 3 - External Dependencies of CraneFoot

2.1.5 Extracting the Implemented Architecture of CraneFoot

Goal: To make the implemented software architecture explicit in order to a) evaluate the structural complexity b) support change-impact analysis for new requirements as part of the future work.

Challenges: We faced the following challenges in extracting the implemented architecture (i.e., module view) of CraneFoot.

1) There are no design documents.
2) All files are placed in one directory. Thus, the hierarchical decomposition of the architectural design is not explicitly visible in the directory structure.
3) CraneFoot is an open source system, and there is a lack of face-to-face meetings with developers/architects.
4) We lacked domain knowledge at the beginning of the project.

2.1.5.1 Extraction Structural View using Conceptual View and Source Code

We overcame the above challenges using a combination of top-down (using the conceptual view) and bottom-up (debugging, source code reading, naming conventions) analysis. The extraction of the implemented architecture proceeded in an incremental fashion. In each increment, we refined the extracted architecture by adding more details to it. Of course, there is no law/rule concerning the number of increments and the level of details added to the extracted view. In our opinion, this is a debatable topic in the field of software architectures. For our goal, we continued increments until we were able to feel the right level of decomposition and sufficient details to do a systematic evaluation of the structural complexity of CraneFoot. In this case, the list of new requirements influenced the level of details necessary for the architecture description. For instance, one of the new requirements deals with layout of family branches based on the generation that individuals belong to. In order to reason about this requirement, we show modules related to layout algorithms and highlighted the most important layout principles: intra-branch and inter-branch layout. In general, we believe the “needs” or scenarios derive the level of abstraction in the architecture description.

As mentioned earlier, all CraneFoot’s files are placed under one directory, probably to simplify the compilation process. However, there are certain naming conventions for naming files that we could use. These conventions together with code reading and debugging allowed us to assign most of the files to components in the conceptual architecture. It is worth noting that some files, although they matched the naming conventions of files in other components, were placed in a different component due to the nature of the implementation logic within the file.

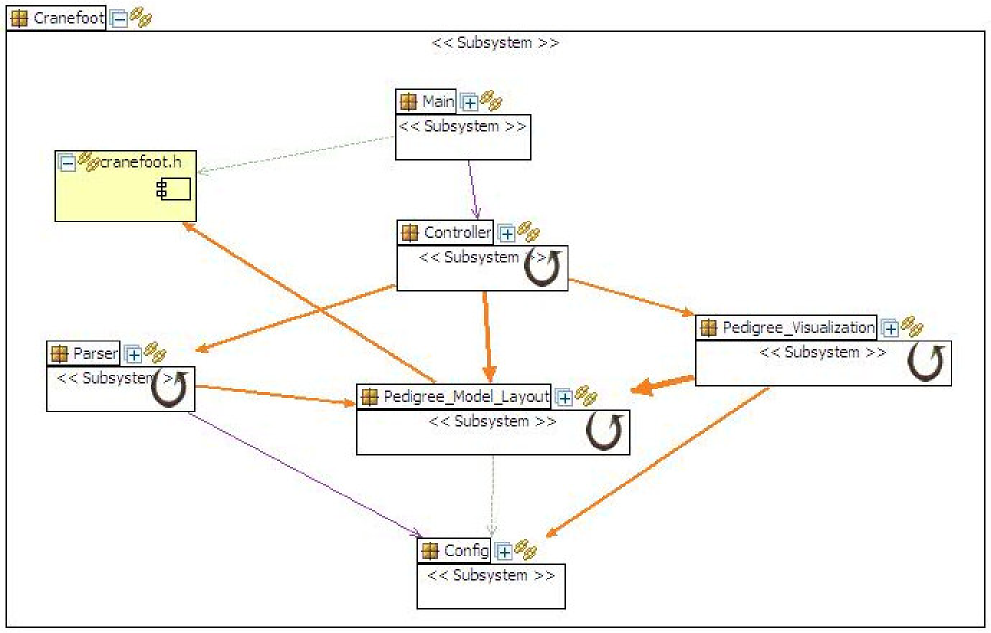


Figure 4 – Structural View – Subsystems and Interfaces of CraneFoot Implementation

After a few iterations where we moved and split operations and components resulted in the structural view shown in Figure 4. We didn’t “blindly” cluster files based on naming conventions nor did we aim for a view with minimum dependencies. Instead, we attempted to come up with a logical and simple decomposition. Furthermore, we can explain the actual story behind components and the meaning of their dependencies to other developers/analysts in an intuitive way.

The following table describes the responsibilities of each subsystem as depicted in Figure 4.

Subsystem/Interfaces	Responsibilities
CraneFoot.h	The main interface of the CraneFoot tool.
Main	The starting/entry point to the CraneFoot tool.
Controller	The subsystem that controls the workflow of the CraneFoot tool.
Parser	Subsystem used to parse the input pedigree data
Model and Layout Algorithm	**Model:** Subsystem that contains the data model of the input pedigree data. **Layout Algo:** This subsystem is made of two components, namely the **Inter-branch** and **Intra-branch** layout. The core layout algorithms based on force-directed simulated annealing and walker algorithms [8] are implemented in this subsystem.
Pedigree Visualization	This subsystem is made of two components: Print and Postscript. The Print component is responsible for printing nodes, edges and links in the pedigree. The Postscript component contains the postscript format for various shapes necessary to draw the pedigree.
Config	This subsystem manages configuration details specified by the user in one of the input files to the CraneFoot tool. Config subsystem contains a model of the configuration parameters set by the user.

Table 5 – Subsystems and Interfaces of CraneFoot

2.1.6 Pedigree Model and Layout Subsystem

The pedigree model and layout subsystem is decomposed into two components (see Figure 5). The Family component is responsible for managing the pedigree data model. The Layout component implements layout algorithms. The bi-directional dependency between these two components is not an architectural issue, because the Family component requests the Layout component to compute the position of nodes/icons, and the layout algorithm uses data from the data model within the Family component vice-versa.

Although the pedigreeobject.h file is not included by files in Family and Layout_Algo, it is being included indirectly through CraneFoot.h.

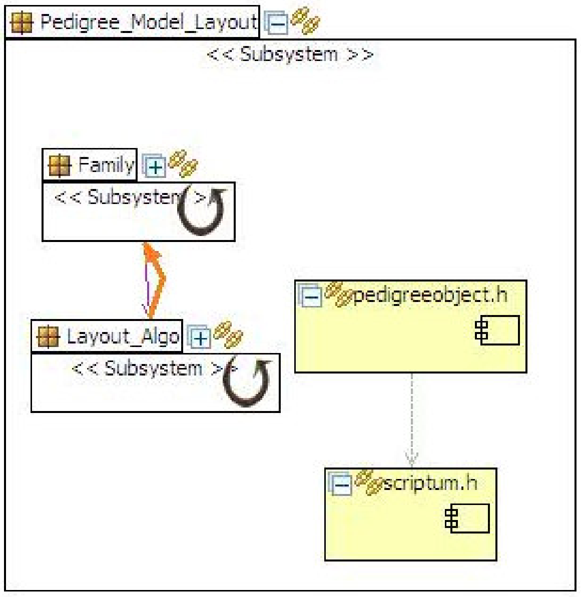


Figure 5 - Structural view of Pedigree Model Management and Layout

2.1.6.1 Family Model Subsystem

One of the core subsystems is the one related to managing the family model of the input pedigree data. This subsystem is responsible for checking the semantic correctness of the input data, and split each family into branches. This model provides API's for querying family data, such as get all children of a parent. The architecture of this subsystem is shown in Figure 6.

As we can observe from the figure, most of the file names share a common prefix. However, files such as member.h, member.cpp, and branch.cpp also deal with a data model of families; this example shows that pure file named based clustering would not be able to under the semantic of these files, without a domain expert role.

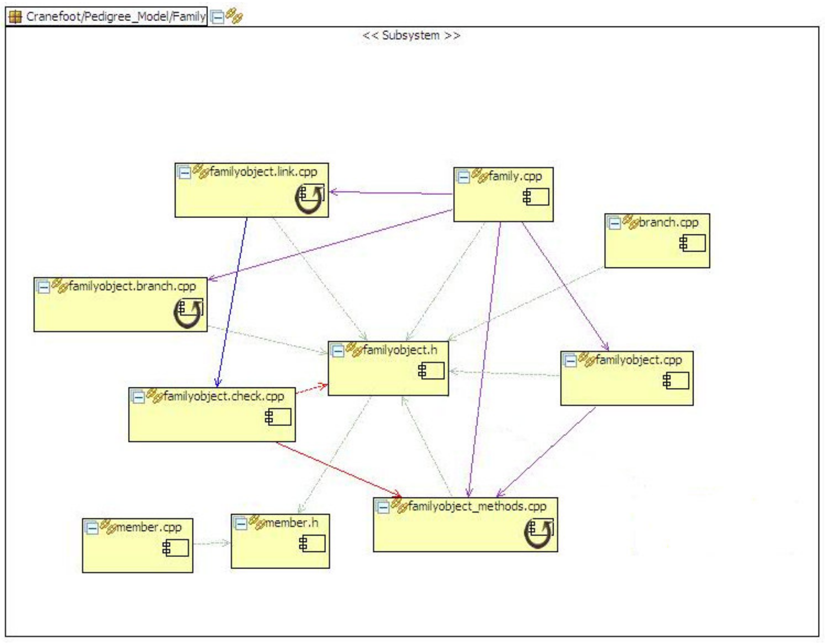


Figure 6 - Structural view of Family model subsystem

2.1.6.2 Layout Subsystem

One of the core portions of the CraneFoot tool is layout of pedigrees. By reading the user manual, general introduction materials about CraneFoot, we learned that it applies force-directed layout algorithms to produce aesthetically pleasing layouts.

We read literatures to learn the basics of force-directed layout algorithms. In these algorithms, ideas of basics physics concepts, such as repel and attract forces among particles, are employed to produce layouts that

- Minimize edge crossings
- Neither keep connected nodes too far nor too close
- Nearly uniform edge lengths

The CraneFoot tool borrows those ideas from the literature on force-directed algorithms. We searched using terminologies mentioned in the literature to localize layout algorithms within the CraneFoot implementation. Debugging also helped us to better understand the control and data flow to these algorithms. Searching and debugging let us to the conclusion that CraneFoot's layout algorithm works in two phases, namely intra-branch and inter-branch (see Figure 7). In the intra-branch phase, it does layout (i.e., Walker tree layout algorithm) on each branch of a family. In the next phase, it applies force-directed simulation algorithms to produce layout in such a way that branches are neither too close to each other nor too far.

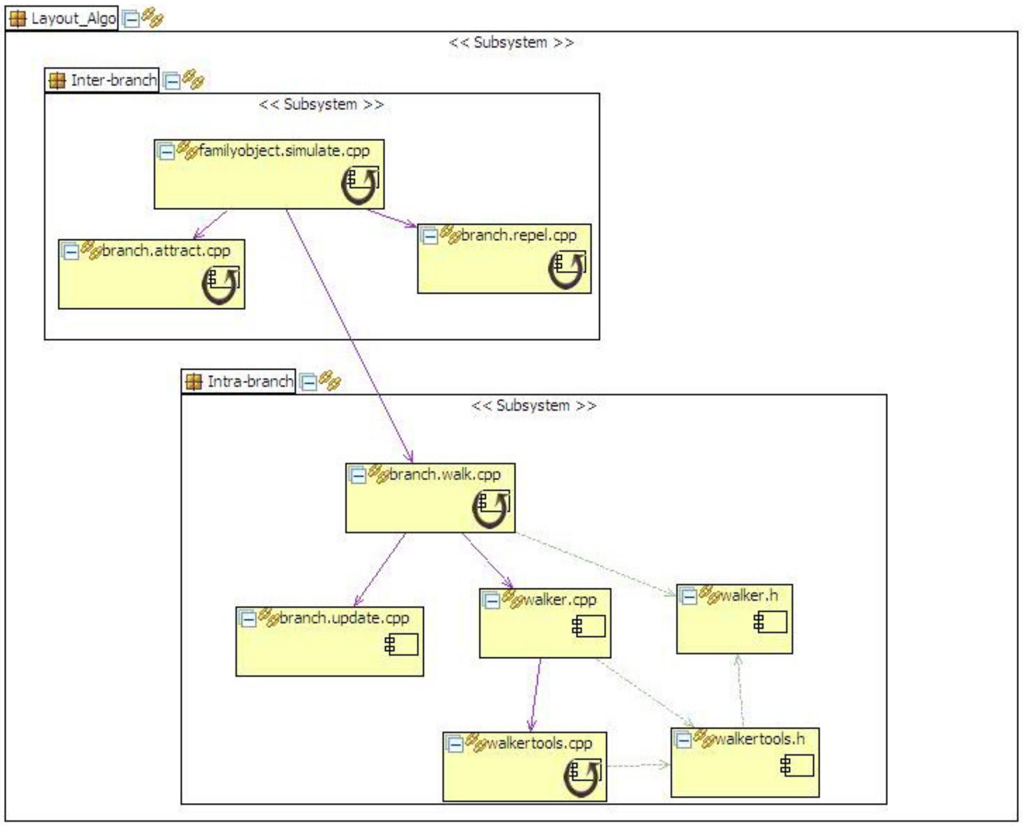


Figure 7 - Structural view of Layout Algorithm Subsystem

2.1.7 Pedigree Visualization

From the conceptual view, we understood that pedigrees are drawn in a postscript format. We used this knowledge, and searched of postscript syntax in the implementation. This activity together with debugging of control and data flows led us to organize the visualization subsystem as shown in Figure 8 into two subsystems: Print and Postscript. We explain these two subsystems in the next subsections.

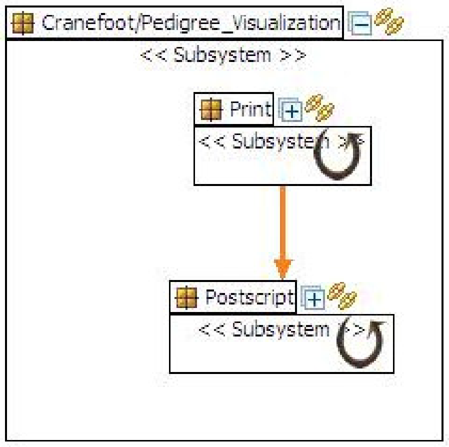


Figure 8 - Structure of Pedigree Visualization.

2.1.7.1 Printing Subsystem

Pedigrees are basically graphs with nodes and edges. In addition, colors, patterns, and various node shapes are used to visualize various phenotypes/genotypes information. CraneFoot also produces a legend in order to explain the meaning of node shapes/patterns/colors. We primarily used this knowledge about the output of CraneFoot to localize files that implement these functionalities. The extracted printing subsystem is shown in Figure 9. We can observe that there is a common prefix for all files involved in printing pedigrees. Moreover, the file names themselves are explaining their roles, enabling program understanding and architecture extraction.

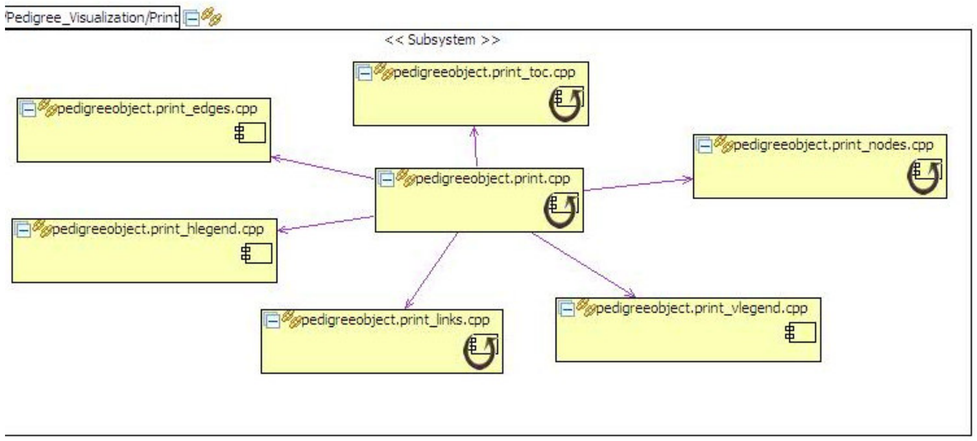

Pedigree_Visualization/Print
<< Subsystem >>
pedigreeobject.print_toc.cpp
pedigreeobject.print_edges.cpp
pedigreeobject.print_nodes.cpp
pedigreeobject.print.cpp
pedigreeobject.print_hlegend.cpp
pedigreeobject.print_links.cpp
pedigreeobject.print_vlegend.cpp

Figure 9 – Structural of Printing Subsystem

2.1.7.2 Structure of Postscript subsystem

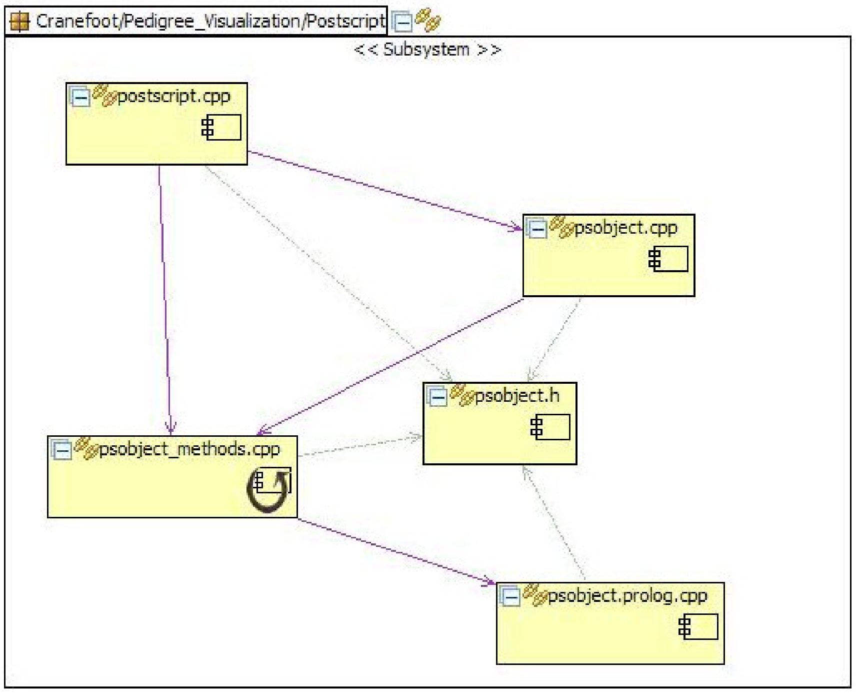


Figure 10 - Structure of Postscript subsystem.

2.2 Behavioral View

The behavioral view illustrates how the architectural elements defined in the structural view interact with each other and realize the required behavior. For that purpose scenarios are used. Scenarios are short textual descriptions of anticipated usages of a system. For a number of typical usage scenarios, the behavioral view shows how the architecture actually “works”.

The behavioral view is concerned with the dynamic behavior of a system. Whereas in the structural view only the usage dependencies between sub-systems and components are shown, the behavioral view shows how the various architectural elements actually interact and behave in order to fulfill a certain usage and the required behavior. In particular, the behavioral view shows for a number of typical usage scenarios which elements of the architecture interact, which operations are invoked by an element, and which messages and events are passed between elements. For each scenario, a separate model is used. Note that in a behavioral view only the architectural elements required for fulfilling a certain scenario are depicted. The behavioral view is especially of interest for developers and everyone concerned with the behavior of the architecture.

For the representation of the behavioral view, UML activity diagrams, collaboration diagrams, and sequence diagrams can be used. We selected sequence diagrams to show how the behavior of the CraneFoot tool.

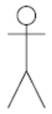
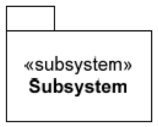
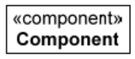
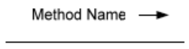
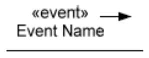
UML Element	Description
	Actor
	Subsystem
	Component
	Method invocation
	Event

Table 6 – Elements and relations of the behavioral view

2.2.1 High level Behavior of CraneFoot

Basically the CraneFoot tool runs as follows:

Step 1: The user passes to the main function a configuration file that contains all necessary configuration parameters related to pedigree data files and visualization settings.

Step 2: The main function requests the controller to draw a pedigree.

Step 3: The controller invokes the parse in order to parse the configuration file and all pedigree input files. The parse then initializes the data model.

Step 4: For every input family, an instance of the family class is created. The family model checks the semantic correctness of the input data (e.g., brother marrying a sister).

Step 5: For each family, the family model requests the layout algorithm to produce layouts for visualization.

Step 6: For each family data, the layout algorithm first computes layout for the branches of the family using the Walker algorithm. In the next phase, the layout algorithm applies force-directed layout concepts at the branch level to position each branches.

Step 7: The controller then requests the visualization subsystem to print out the pedigrees in a postscript format.

These steps are captured in Figure 11. We can notice that the CraneFoot tool works in an intuitive manner. An interesting observation from this figure is that CraneFoot is a configurable tool. The user can configure the way the input data is processed, the contents of the pedigree model is constructed, and the contents of output postscript files.

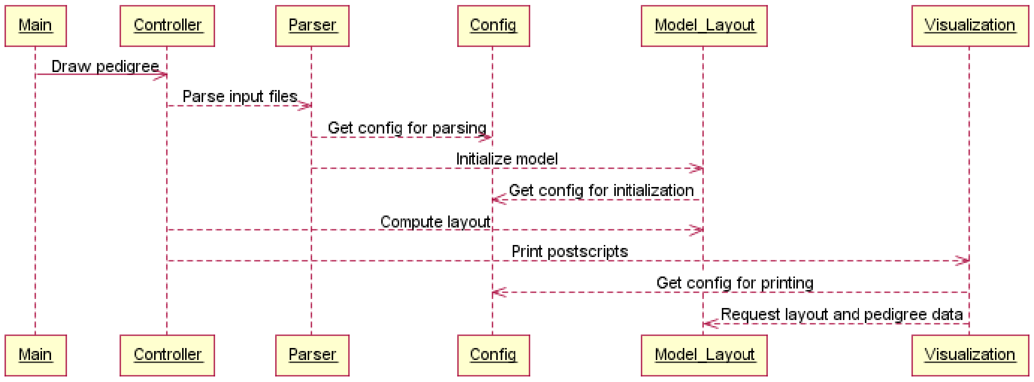


Figure 11 – Behavioral View of the CraneFoot tool

2.3 Implementation View

The implementation shows how the source code of a system is organized in the development environment. It captures how architectural elements defined in the structural view are organized in the development, integration, or configuration management environments. This view describes the mapping of the logical elements from the structural view to physical elements of the development environment. In particular, the implementation view depicts how the subsystems, components, and the interfaces defined in the structural view are mapped to entities of the development environment and applied implementation technology such as source code files, folders, COTS components, libraries, etc. Also, the dependencies and interrelation-ships between source code files are illustrated.

Since the implementation view is very technical, the main stakeholders are developers concerned with the implementation and realization of a system with implementation technologies. The view is critical for the management of development activities as build-processes and addresses mainly the following concerns:

- Build time reduction
- Tools for development environments
- Integration
- Version and release management
- Testing

The implementation view is represented using UML diagrams as well as tables. While the diagrams are basically used for providing the structuring of the folders, tables are better suited for depicting larger amounts of data such as the numerous source and header files in folders. In Table 7 a template for this purpose is given. Also, they can be used for showing the map-ping between elements from the structural view such as subsystems and components and the respective elements of the implementation view. Table 8 gives an example of how the mapping can be documented.

Implementation Folder	Source and Header Files

Table 7 – Template for Documenting the Contents of Folders

Structural Element	Implementation Element

Table 8 – Template for Documenting the Mapping of Elements

For the graphical representation of the implementation view, UML diagrams with customized elements are used. In particular, the package notation in conjunction with the stereotype «folder» is used for depicting folders. Source code files such as .c files are depicted by using the UML component symbol, which represents binary or executable components, together with the stereo-type «source». Between source code files and header files, an import relationships exists which is depicted using the stereotype «import». Table 9 summarizes the elements and relations used for representing the implementation view.

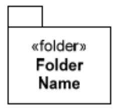
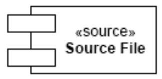
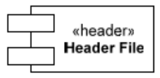
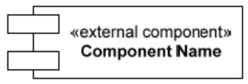
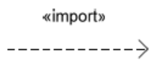
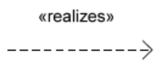
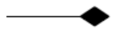
UML Element	Description
	Folder
	Source file (e.g.. .h or .c file)
	Header file
	External component
	Imports dependency
	Realizes dependency
	Composition

Table 9 – Elements and Relations of the Implementation View

The implementation view is given in Table 10 and Table 11.

Structural Element	Implementation Element
Main	main.cc

Structural Element	Implementation Element
Controller	pedigree.cpp, Pedigreeobject.cpp, Pedigreeobject.run.cpp
Parser	pedigreeobject.import.cpp, pedigreeobject.configure.cpp
Config	configTable.cpp, Tablet.h
Model	family.h, family.cpp, branch.connect.cpp, familyobject.cpp, familyobject.branch.cpp, familyobject.check.cpp, familyobject.link.cpp, branch.cpp, familyobject_methods.cpp, member.h, member.cpp, pedigreeobject.h, scriptum.h
Layout Algo\Inter-Branch	**Inter-Branch:** familyobject.simulate.cpp, branch.attract.cpp, branch.repel.cpp
Layout Algo\Intra-Branch	**Intra-Branch:** walker.cpp, branch.walk.cpp, branch.update.cpp, walker.h, walkertools.h, walkertools.cpp
Visualization\Print	pedigreeobject.print_toc.cpp, pedigreeobject.print.cpp, pedigreeobject.print_edges.cpp, pedigreeobject.print_nodes.cpp, pedigreeobject.print_links.cpp, pedigreeobject.print_hlegend.cpp, pedigreeobject.print_vlegend.cpp
Visualization\Postscript	postscript.cpp, psobject.cpp, psobject_methods.cpp, psobject.h, psobject.prolog.cpp

Table 10 – Mapping Structural Elements to Implementation View

Implementation Folder	Source and Header Files
Source	In CraneFoot, all files are placed in one folder/directory.

Table 11 – Implementation View Folder Structure

3 Code Quality Assessment using Metrics

To get an understanding of the current implementation and to detect potential problems and improvement areas, the source code of the CraneFoot tool has been analyzed quantitatively by means of software metrics.

3.1 Metric Selection

Software metrics are an unbiased means to objectively qualify the source code of a component or application. Typical software metrics are the size of the code (measured in lines of code, number of statements, and so on) and the code complexity (measured through complexity figures such as the Cyclomatic complexity). The measurement of source code provides useful information for the assessment of its quality, predicting to some extent the external system quality characteristics, such as maintainability, reliability, extensibility, and portability. Measurements may be used to obtain a picture of the quality both of a single component and of an entire program.

The application of software metrics to a code base can be an effective overall gauge of software quality. Note, however, that applicability of a particular metric to a domain is usually subjective. Highly coupled code, for example, may have been intentionally designed that way for performance reasons; consequently, a coupling metric that suggests problems with this code must be evaluated in the context of the overall application.

The key to effective utilization of metrics is to set an acceptable range measured values for each metric should fall in and to look for the outliers (i.e. values that are outside the defined range or extremely deviate from the mean determined for the analyzed code). The comparison of measured values for metrics to the acceptable ranges can lead to a representative view of the quality of the analyzed source code. The outliers, on the other hand, indicate areas where further analyses such as reviews should be performed.

In case of the analyzed CraneFoot system, the metrics summarized in the following table have been used.

Metric	Description
Total Number Lines of Code	The number of all lines in the source code. This includes code, comments, and blank lines.
Active Lines of Code	The number of lines that contain source code. Note that a line can contain source and a comment and thus count towards multiple metrics.
Inactive Lines of Code	This is the number of lines that are inactive from the view of the preprocessor. In other words, they are on the FALSE side of a #if or #ifdef preprocessor directive.
Number of Comment Lines	The number of lines containing a comment. This can overlap with other code counting metrics. For instance “int j

Metric	Description
	= 1; // comment" has a comment, is a source line, is an executable source line, and a declarative source line.
File Size (LOC)	The number of lines in a source code file.
Function Size (LOC)	The number of lines making up a single function.
Cyclomatic Complexity	The Cyclomatic Complexity as defined by McCabe in [6].
Nesting	The maximum nesting of control structures (if, while, for, switch, etc.) in a function.

Table 12 - Definition of Applied Metrics

3.2 Overview Metrics

Table 12 depicts the project overview metrics for the CraneFoot tool. Note that the values for maximum file size, average file size, maximum function size, and average function size show two values for the number of lines of code: the first value including blanks and comments whereas the second value only includes lines that actually contain source code statements.

Metric	Value
Number of Code Files	52
Total Number Lines of Code	9553
Active Lines of Code	7073
Number of Comment Lines	3236
Maximum File Size (Total)	434
Maximum File Size (Code)	337
Average File Size (Total)	183
Average File Size (Code)	100
Maximum Function Size (Total)	316
Maximum Function Size (Code)	126
Average Function Size (Total)	29
Average Function Size (Code)	25
Max Cyclomatic Complexity	52
Maximum Nesting	5

Table 13 - Project Overview Metrics

The simplest and hence in practice most often used metric is the number of lines of code. The more lines of code in a file, the more difficult it is to understand the file. As shown in Figure 12, the maximum file size is 460, which is acceptable and not yet an anomaly.

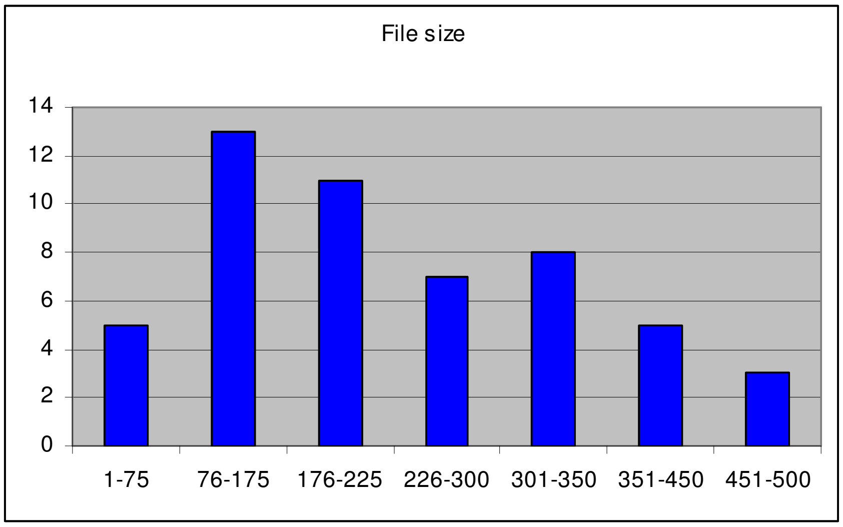


Figure 12 - File Size Histogram

An indication for areas in the code which are hard to maintain and in particular hard to understand are functions with a large number of maintainable code lines[5]. The maximum function size is 316, which is quite above the average value, if compared to other systems.

[5] Blank lines and comment lines are included; lines containing only curly brackets are excluded.

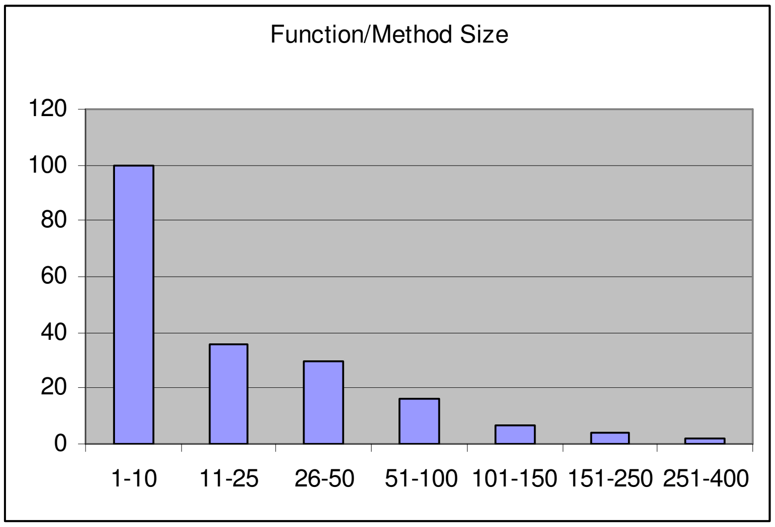


Figure 13 - Function Size Histogram

Some programming standards recommend that functions should not have more than 50 maintainable code lines, whereas others suggest that a function should not exceed 200 maintainable lines. Figure 13 shows the frequency of the values for the function size. As can be seen, the vast majority of the functions have less than 50 lines of code. The function sizes are only rarely above 250 lines of code and the maximum function sizes in most cases stem from one single function.

As code complexity has an effect on how difficult a component or system is to test and maintain, we calculated Cyclomatic Complexity for all functions. This metric, which has been proposed by McCabe [6], counts the number of independent paths in the control flow graph of a program component. Its value depends on the number of branches caused by conditional statements (if-then-else). A common application of Cyclomatic complexity is to compare it against a set of threshold values. Table 14 gives an example of typical threshold values.

Cyclomatic Complexity	Risk Evaluation
1-10	A simple program, without much risk
11-20	More complex, moderate risk
21-50	Complex, high risk program
greater than 50	Non-testable program (very high risk)

Table 14 - Threshold Values for Cyclomatic Complexity

High Cyclomatic complexity indicates inadequate modularization or too much logic in one function. Some strict programming standards dictate that a function may have a maximum Cyclomatic complexity of 10. Basically, functions with a Cyclomatic complexity greater than 50 should be analyzed in more detail and then the implementation improved.

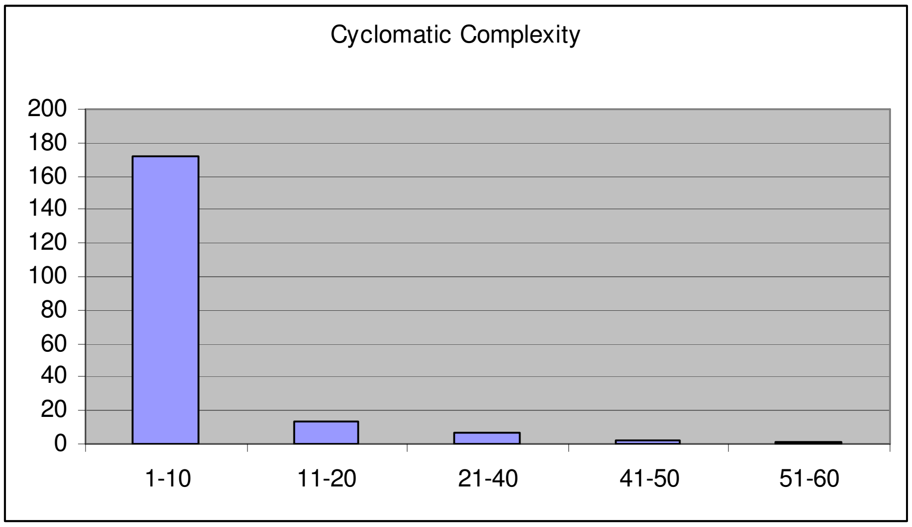


Figure 14 - Histogram for Cyclomatic Complexity

Figure 14 shows the distribution of cyclomatic complexity for the CraneFoot tool. Except a few functions, most of them have the cyclomatic complexity within 10. This implies is that code complexity is under control so far.

4 Evaluation of the Maintainability Aspect Ease-of-Change

4.1 Interpretation: Analysis of Structural Complexity of CraneFoot – Making use of Extracted Views

We used the extracted views for evaluating the structural complexity of CraneFoot. We believe semantic cohesion, separation of concerns, and (unwanted) dependencies among subsystems are the major factors contributing to the structural complexity of any implementation.

From the extracted views, we were able to justify that the implementation can be decomposed into logical subsystems. Furthermore, we did not come across any unwanted dependencies among subsystems.

CraneFoot's implementation clearly separates concerns, such as configuration, data parsing, model building, layout algorithms, and Postscript generators.

Using the SAVE tool complemented with our domain knowledge, we were able to visualize all major subsystems of CraneFoot, which would not have been possible otherwise.

The extracted views showed us that the structural complexity is well managed in CraneFoot.

4.2 Interpretation: Analysis of Code Quality

In this process of architecture extraction, as mentioned earlier, we ended up reading and analyzing many files. Our qualitative feeling about the source code is that it is well-organized and follows uniform coding conventions. There are some locations in the code which are complex due to the complexity because of the algorithms in use. However, in general, the code quality is well controlled.

Software metrics complemented our qualitative belief that the implementation is indeed good with help of numbers. Moreover, using bar-charts we could explain in general how good the quality is. Simple metrics like the function size and cyclomatic complexity provide good insight about the overall implementation quality. Furthermore, these metrics set the stage for quantitatively monitoring the evolution of the system as new features are introduced. In addition, these metrics gives indication about potentially risky code elements. When changes are realized, these metrics can be used to check whether there are risks associated with changes. Thus, enabling to control the implementation quality as the system evolves.

4.3 Feedback from the CraneFoot Developer

We gave this report to the inventor the CraneFoot tool in order to get his feedback. Our goal was to check whether the architectural views extracted using our approach are in indeed meaningful and a good representation of CraneFoot.

The CraneFoot developer replied: "I was absolutely fascinated when I read the document, because this type of analysis is unheard of in the kind of academic environment I'm working in. I think your approach to managing code complexity is of course critical for large software projects, but it seems that even smaller packages such as CraneFoot will benefit from a careful study of the code structure.

I think that the document will be highly valuable for people who wish to tweak CraneFoot for their own purposes. I thought that the report was well structured and easy to read, even for someone who is not an expert on software production. The metrics were very interesting and I liked that you had explained also the target values for the metrics."

5 Conclusion

In this report, we described the implemented software architecture of Cranefoot. We described our approach for evaluating the complexity (structural and code-level) of an unfamiliar domain implemented in an open source project. We followed an architecture-centric approach for that goal. We described the system's architecture using views, such as conceptual view, structural view, behavior view, and implementation view.

We claimed that a combination of top-down and bottom-up approach is a practical way of extracting these architectural views from an implementation. We emphasize that domain knowledge and knowledge about inputs-outputs of the system under study are extremely important to extract architectural views from an implementation. In our opinion, this point is often missing in reverse engineering literature. Using domain knowledge and input-outputs, together with general software development experiences, we believe a conceptual view of a system can be sketched. The level of details in the conceptual view is primarily driven by the list of requirements/scenarios to be introduced in the system. After that, naming conventions, keywords searching, and debugging could be employed to map source code files into components in the conceptual view. Using tools like SAVE, the extraction of dependencies among the components becomes an easy task.

Although we do argue against clustering algorithms that are used in reverse engineering for extracting architectural views, there are some notable drawbacks we observed in this case study. Clustering algorithms are domain-independent. Thus, they don't "understand" the domain meaning of the elements they cluster. Clustering algorithms don't explain the role of each component from a domain perspective. Also, they don't explain which dependencies among components are Ok, and which ones are not. Furthermore, it is nearly impossible to produce a hierarchical decomposition of an implementation just based on general purpose clustering algorithms. Every domain probably has its own elements and its own hierarchies. Nevertheless, as a part of the future work, we plan to compare architectural views produced by clustering tools against the semi-automated approach explained in this report.

Our conclusion is that a combination of top-down and bottom-up approaches of reverse engineering is practical and produces appealing results.

For the CraneFoot tool, the extracted views showed that the implementation has a good structural decomposition, and the complexity is well managed with a careful decomposition. Similarly, the code-level evaluation using metrics revealed that code quality is well managed.

A quote from the CraneFoot web-site: ”The principle behind the design of CraneFoot was simple: Quality before quantity. Scientists, like I, often make the mistake of adding features as soon as they get the program to run correctly for some input. This leads to incomprehensible and unmanageable sprawling code that is of little use to anyone. I have actively avoided such a tendency - and released the source code - with the hope that other people can also enjoy the results."

Our analysis confirmed this quote using architectural views and metrics from the implementation.

5.1 Acknowledgements

This work is funded by the STTR CA CA105217-O1A1 "Bioscience Research Integration Software Platform (BRISP) project supported by National Institute of Health, USA, and the ArQuE project supported by the German BmBf.

We thank Ville-Petteri Maekinen, the inventor of Cranefoot, for his comments on this report, and also making the tool available for the public.

We used some of the templates created by Jens Knodel, Fraunhofer IESE, Germany for describing the architectural views of Cranefoot. Dr. Mikael Lindvall helped edit the report.

We thank Steve Chen, Mark Brocato, and Jian Wang, the members of BioFortis, Columbia, for technical discussions on pedigree visualization and open source topics. In addition, we thank all members of STTR project for their encouragement, in particular, Dr. Steve Bova for fruitful discussions.

References

[1] P. Clements, Documenting Software Architecture: Views and Beyond. Addison-Wesley, Reading, MA, 2002.

[2] P. Clements, R. Kazman, M. Klein. Evaluating Software Archi-tectures: Methods and Case Studies. Addison-Wesley, 2002.

[3] C. Hofmeister, R. Nord, D. Soni. Applied Software Architecture. Addison-Wesley, Reading, MA, 2000.

[4] J. Knodel, M. Lindvall, D. Muthig, and M. Naab. Static Evaluation of Software Architectures. 10th European Conference on Software Maintenance and Reengineering, Bari, Italy, 2006.

[5] www.thesavetool.com

[6] V-P. Maekinen, et al. High-throughput pedigree drawing. European Journal of Human Genetics (2005) 13, 987–989, 2005.

[7] T. McCabe. A Complexity Measure. IEEE Transactions on Software Engineering, 2(4):308–320, 1976.

[8] Walker JQ II: A node-positioning algorithm for general trees. Software Practice Experience; 20: 685-705, 1990.

List of Figures

List of Tables