Quality Impact of Configuration and Customisation on Configurable Software

Geoffrey Lydall

A dissertation submitted to the Faculty of Engineering and the Built Environment, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg, September 2018

Declaration

I declare that this dissertation is my own, unaided work, except where otherwise acknowledged. It is being submitted for the degree of Master of Science in Engineering to the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination to any other university.

Signed this ____ day of _____ 20__

_____

Geoffrey Lydall

Abstract

A case study is performed on a weighbridge application which allows for configurations and customer-specific modifications. A literature review includes topics of software quality, software customisation, and ontology. The effects of the customisations and modifications are evaluated for the structural and functional quality of the system, and the configuration architecture assessed for its success in accommodating configuration and customisation from a quality perspective. A statistical model is defined to estimate how the number of defects may change with modifications to a system. The structural quality is measured using the Maintainability Index and Overview Pyramids. The functional quality is assessed using defect data recorded in the task tracking software Jira and the revision history stored using the version control software Git. The amount of modification is measured using the number of rules defined per customer. The results indicate that structural quality is unaffected by the modifications, and that the functional quality is reduced as more customisation rules are defined indicating a partial success of the architecture.

To Joslin...

Acknowledgements

I would like to thank the many people for their support and assistance in this study without whom it would not have been possible.

To my wife, Joslin, who has always been there for me and been generous with her unfailing love, support, understanding and encouragement.

To my supervisor, Dr Levitt, for his guidance, feedback, patience and encouragement.

To Marian for her selfless editing and advice.

To my family for their support encouragement.

To Basil, Garrick, and Bernice for their time, assistance, and advice.

Last but not least, I would like to thank the software product vendor that generously supplied the material for the study.

Contents

List of Figures

List of Tables

Nomenclature

CC Cyclomatic Complexity

CIL Common Intermediate Language

CRUD Create Read Update Delete

CSV Comma Separated Value

ERP Enterprise Resource Planning

GQM Goal Question Metric

HV Halstead Volume

IT Information Technology

LOC Lines of Code

MI Maintainability Index

NOC Number of Classes

NOM Number of Methods

NOP Number of Packages

MVVM Model - View - View-model

OO Object Oriented

SaaS Software as a Service

SIG Software Improvement Group

SME Small and Medium Enterprises

SMM SIG Maintainability Model

UI User Interface

UML Unified Modelling Language

VS Visual Studio

XML eXtensible Mark-up Language

Chapter 1

Introduction

Software projects carry a notorious reputation of being late and over budget [1]. Methodologies and processes such as Agile Software Development, PSP, SCRUM, RUP, CMMI and SEMAT aim to improve the quality of the software development process, while formal software design principles such as SOLID, Design Patterns [2], and continuous improvements in software technologies and language developments aim to improve the structural quality of software. However, despite these developments, projects are still late and exhibit quality problems.

Software quality is regarded as consisting of three principle dimensions: Process, Structural, and Functional quality [3]. Even if a project is well built using sound process and methodology, this is no guarantee that the functional quality (as perceived by the user) of the project will be of a high standard. Component based software technologies could be leveraged to overcome these quality problems and help control cost in software projects [4]. Software vendors will be able to specialise in specific software components and economic viability of these specialisations realised through reuse volumes.

In order to explore reuse of software components from a quality perspective, a case study is proposed to investigate the effects that customisations have on the quality of specialised off the shelf software. Should quality be negatively impacted by the customisation of the product, it may further inform build vs buy decisions for organisations which use specialised software. For example, if customising software significantly reduces the functional quality, then organisations may be better off building the specific software.

An understanding of quality from a manufacturing and services perspective is explored - considering that software has aspects of both a good and a service. Metrics for

software quality and size of customisation are explored for a comparison of quality against the extent of customisation. A descriptive statistical model is presented that suggests a mechanism how functional quality is affected by customisation.

The application in the case study is a weighbridge application which is described in Chapter 4.2.

1.1 Overview of the Dissertation

This section provides an overview of the dissertation and describes what work is covered in which chapter. The literature for the study is largely covered in the chapters for which the literature is relevant.

Chapter 1 introduces the study and covers previous and related work including software quality, software as a service, and ontology.

Chapter 2 provides a more in depth discussion on software quality. The chapter defines software quality and presents metrics for the measurement of software quality.

Chapter 3 presents the research questions for the study, the scope of the study, the aim and motivation of the study, and a statistical model for the purpose of reasoning about expected outcomes of the research questions.

Chapter 4 presents the research methodology. This chapter introduces the application and customers under study, describes how quality will be measured, and what kind of data will be gathered to answer the research questions.

Chapter 5 provides details about the manner in which the application is customised and how the size of a modification or customisation is determined.

Chapter 6 provides details about how code quality metrics are acquired using a combination of existing tools and software written to augment the existing tools. The Maintainability Index is covered, including how its calculation in Visual Studio and how generated code can be excluded from the calculations. Details are provided on the Overview Pyramid and how the additional metrics not provided by Visual Studio are calculated in order to produce it.

Chapter 7 provides details about the functional quality is measured using defect data stored in the task tracking and revision control for the software under study. The

software Git and Jira are also described.

Chapter 8 presents the results of the study and an analysis of those results. The research questions are answered and threats to the validity of the study are discussed.

Chapter 9 presents the conclusion of the dissertation. A brief summary of the findings are presented and future work is proposed.

1.2 Previous and Related Work

The work outlined in chapter 2 is considered, as well as some additional studies indicated below.

1.2.1 Customising Software

Enterprise Resource Planning (ERP) systems and Software as a Service (SaaS) provide areas where academic studies contain related work.

Some literature on implementing ERP systems in organisations is considered, noting some challenges that ERP implementations may present.

ERP systems have been adopted in business and are at times customised to meet the specific needs of a customer in the workplace [5]. Light's work discusses the maintenance implications of various changes to an ERP system for both the customer and the vendor. Some of the challenges include a customisation for one customer which actually competes with the customisations for another customer, and also points out fragility in the upgrade path of customisations. This work, however, does not quantify the effects of these customisations for either the customer or the vendor.

Light's work offers insight into the types of changes that had been applied to the ERP system for the customers as well provide a qualitative overview of what can be expected. Light noted the following types of changes in customising the software (in order of potential for required maintenance, high to low):

- Change functionality
- Add functionality
- Process automation

- Amend reports/displays
- New report

Light's work provides an example of the kinds of work to be expected from the Weighbridge Software for this study. The software is described in Chapter 4.2.

Also considered, is the methodology presented by Guido [6], which assesses the feasibility of ERP implementation strategies. This study provides a formal understanding of the business context in which a system is going to operate. Specific mentions include the alignment of the ERP system to business. Guido's work has also been applied to numerous studies, one of which is Kumar's work [7] that concludes customisation beyond 30% adds considerable risk to the project. The work outlines customisations as "any modifications or extensions that change how the out-of-box ERP system works". Kumar's conclusion stems from an analysis of error counts in modules of the system and recommendations from the system vendor.

1.2.2 Software as a Service

Software as a Service (SaaS) is a relatively new concept in which a software vendor provides a system to a customer through an on-line channel, often using some form of tenancy architecture. This channel of delivery typically offers more limited customisations than a traditional vendor software deployment, although this is then touted as an area of cost saving. What is significant about this, is the notion that businesses may be willing to adapt their business model to be better aligned to what the vendor offers in order to realise a cost saving; competitive advantage; or process efficiency [8]. The vendor can also deliver further value to their customers through analyses of the usage patterns of their software [9][10].

Xin's work [8] proposes nine hypotheses with regards to adoption factors for SaaS:

H1 Clients with a higher degree of desired customization for a given software application are less likely to adopt the SaaS model than the on-premises model.

H2 Clients with higher demand volume uncertainty for a given software application are more likely to adopt the SaaS model than the on-premises model.

H3 Clients with higher demand uncertainty for client-specific functionality for a given software application are less likely to adopt the SaaS model than the on-premises model.

H4 Demand uncertainty for client-specific functionality moderates the relationship between the degree of desired customization and client's propensity to adopt the SaaS model for a given application.

H5 Clients with a large number users for a given software application are less likely to adopt the SaaS model than the on-premises model.

H6 Clients with more extensive internal IT capabilities are less likely to adopt the SaaS model for a given application than the on-premises model.

H7 Clients with high cost of capital are more likely to adopt the SaaS model than the on-premises model.

H8 Clients that are more receptive to peer organizations' influence in their IT decision making are more likely to adopt the SaaS model.

H9 Clients with more mature[1] enterprise IT architecture are more likely to adopt the SaaS model.

These hypotheses highlight that fewer customisation options for customers are typically detractors from using SaaS, but also state that organisations "more receptive to peer organisation's influence in their IT decision making" are more likely to adopt a SaaS model in order to mimic successful behaviours or to compensate for a poor IT capability. This implies that organisations may adopt a business model provided by a third party which is considered already successful. Xin also cites the *cost of capital* as another motive in adopting SaaS. Xin's work has been later considered in Seethamraju's work [11] which highlights the advantages of Small and Medium Enterprises (SMEs) sacrificing control and adopting SaaS systems. These advantages are realised by the relatively ad hoc processes and small scales of SMEs compared with larger organisations, where the change process is simpler; the product is cheaper; and the customer's requirements do not include integration concerns.

Seethamraju's work also contrasts SaaS and on premise ERP solutions for organisations. The work states that whilst ERP offers more customisation, it also typically undermines the practices of the ERP system, and highlights the advantages in terms of cost and process efficiency that SMEs may gain by using SaaS over on premise ERP.

Sun's work [9] describes four different cost models for SaaS vendors that include varying degrees of customisation by the vendor with corresponding cost implications.

[1] In this context, *mature* refers to the organisation's ability to organise logic, data, and infrastructure.

The four models plotted in Figure 1.1 are summarised below as:

A Native Design

B Smooth Evolvement

C Pulse Evolvement

D Failure Management

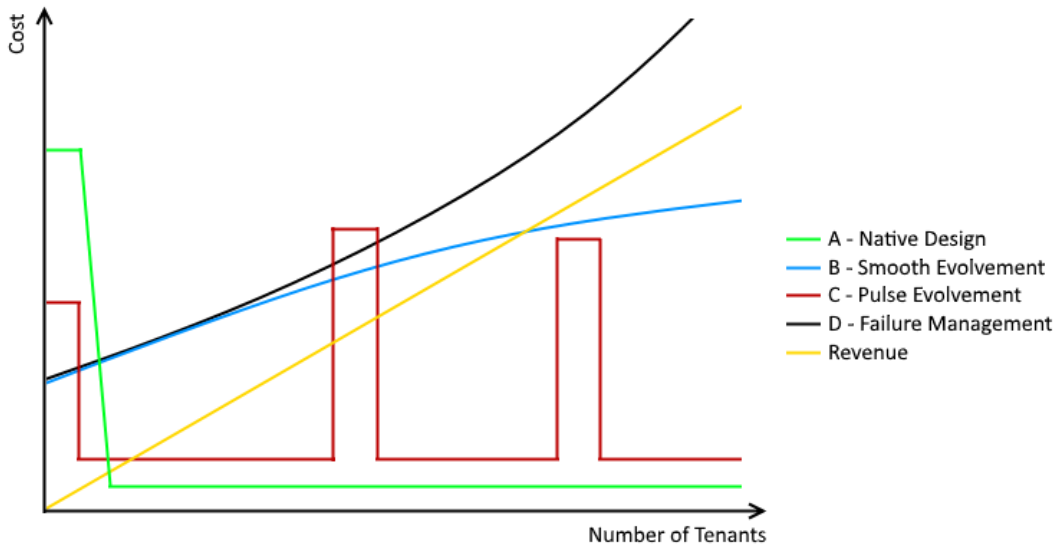


Figure 1.1: Illustrative cost estimates for the four SaaS models outlined by Sun.

The *Native Design* model is an upfront specification and development effort and minimal expenses through the rest of the product. *Smooth Evolvement* is an on-going design and development model with a levelling out of costs with increasing tenants. *Pulse Evolvement* is a model where feature development is applied in pulses - similar to successive Native Design models with increasing tenants. The *Failure Management* model is like the Smooth Evolvement model except that the cost increase accelerates with an increased number of tenants.

Sun's work suggests that the most successful vendor models are the Native Design and Pulse Evolvement models - both of which have infrequent changes that are dependent on extensive product design and analysis across a range of tenants, and therefore constitutes a shared process model for customers.

1.2.3 Ontology

The concept of ontology is a separate, but related concept to this study. The principle of ontology is to provide a conceptual model for systems in an integrated way that can be communicated and understood in an unambiguous way [12].

There is a large body of knowledge that relates to ontology and Software and Knowledge Engineering. [13, 14, 15, 16].

Such studies with respect to ontology are related because it is implied by the hypothesis that the software product itself forms a standard model, or ontology, for the domain in which the software is specialised in. This then explains why specialised software products could realise benefits in terms of Component Based Software Engineering and as well as quality improvements in software. These benefits originate from the deep expertise of the engineers that produce this software since the understanding is ideally deep and complete.

This study is framed within a context of ontology as it applies to engineering practices and quality in software.

1.2.4 Conclusion

A context for study has been introduced as quality in software engineering and an outline of the dissertation has been presented.

A review of previous and related works has been presented, including software as a service, customisation of ERP systems, and ontology.

Chapter 2

Quality

This chapter discusses the definition of software quality. The definition is drawn from a number of different sources in literature. Much work has been done to describe a range of metrics that describe the software's maintainability.

The quality definition is discussed from a range of perspectives, including the three aspects of internal, external, and functional quality; and the quality framework defined in ISO9126. The final definition also draws from more traditional definitions used in manufacturing and services.

Also discussed is the notion of defect as a measure of quality.

2.1 Defining Quality

The definition of quality in the scope of software engineering needs to be understood in order to understand what is being proposed to be measured.

This definition begins with an understanding of software quality as defined by David Chappel [3]. His paper outlines three aspects of software quality: Functional; Structural; and Process.

- **Functional:** The quality as would be perceived by the user - e.g. performance, defects, ease of use.
- **Structural:** The quality as would be perceived by developers - e.g. maintainability, testability, security.

- **Process:** The quality of the project as would be perceived by a project manager or sponsor - milestones, budgets, repeatability.

Functional and structural quality can also be thought of as Steve McConnell's external and internal quality, respectively [17].

Within this framework, it can be understood that popular processes or techniques such as PSP/TSP, Scrum, Kanban, RUP, CMMI and SEMAT target process quality [18]; testing (user and internal) can target functional quality through the detection and removal of bugs and functional verification; and unit tests and other code quality metrics can target the structural quality.

ISO 9126 [19] is a standard that defines a set of quality characteristics and sub characteristics for software systems. These characteristics are:

- Functionality
  - Suitability
  - Accuracy
  - Interoperability
  - Security
  - Functionality compliance
- Reliability
  - Maturity
  - Fault tolerance
  - Recoverability
  - Reliability compliance
- Usability
  - Understandability
  - Learnability
  - Operability
  - Attractiveness
  - Usability compliance
- Efficiency

  - Time behaviour
  - Resource utilization
  - Efficiency compliance
- Maintainability
  - Analysability
  - Changeability
  - Stability
  - Testability
  - Maintainability compliance
- Portability
  - Adaptability
  - Installability
  - Co-existence
  - Replaceability
  - Portability compliance

As can be seen, the standard is broad and covers more than the technical aspects of software, including more human factors such as usability and functionality.

Hegedűs' [20] work provides a summary of quality metrics from a source code perspective, and refers back to ISO 9126, mentioning that it provides a framework within which to conduct further research into software quality. The work covers:

- Existing maintainability models.
- A probabilistic maintainability model and its validation.
- A maintainability model for C#.
- Implementation and evaluation of developed tools and models.

The work highlights that most software quality research has a principle focus on the maintainability sub characteristics of ISO 9126 as these are of immediate interest due to the effect of maintainability on the cost of change. Although the source code itself is key to maintainability, ISO 9126 does not prescribe any specific source code metrics.

Of the functional quality sub characteristics, in Hegedűs' model only the *Performance Rules* and *Security Rules* (each written and tested as FxCop rules) are cited as quality "measures". Although these are considered as functional quality measures in Hegedűs' model, they provide little insight into the user's overall quality experience - the considered definition of functional quality. More information on FxCop can be found in Appendix A.2.

There are no direct means of measuring functional quality. A deeper definition of quality must first be considered from a manufacturing and services perspective [21]. In this regard:

- Goods can be considered in the dimensions of: Performance; Features; Reliability; Durability; Conformance; Serviceability; Aesthetics; and Perceived Quality.
- Services can be considered in the dimensions of: Reliability; Tangibles; Responsiveness; Assurance; and Empathy.
- Software can be regarded as both a good and a service, thus taking on aspects of both of the above. Many of these dimensions are difficult to objectively measure, however, their absence can be easily measured.

Functional quality can be inferred from the absence of defect, and defect can be measured. Freimut's work [22] considers approaches for measuring defect for the purposes of quality improvement strategies. His work provides a process to track defect introduction and detection for the purposes of establishing a quality assurance baseline.

Kuan's [23] work focuses on a quality comparison of the open and closed source systems from an economics perspective. Kuan's work directly measures defect rates in order to develop a Cox Hazard Model [24]. The results then provide an indication of the kind of hazard each system presents to its prospective users and therefore infers a level of quality.

2.2 Quality Metrics

This section explores some of the quality metrics discussed in literature.

There is much work in software engineering that relates the areas of component based engineering and quality. Ravichandran's [4] work discusses a decision making strategy for reuse. Ravichandran specifically refers to the concepts of domain and contextual distance. These distances refer to differences in the application domain and environment respectively. These are quantified in terms of the effort required to close the distances. Ravichandran refers to Shepperd's estimation by analogy, however, any effort estimation tool could theoretically be used. Estimation by analogy is an estimation technique based on experience where the work to be done is compared with the effort required to complete previous work [25] - similar to *story points* in Scrum [26]. Furthermore, the domain and contextual distances can be retrospectively known if the actual effort expended is measured.

Pantazopolous [27] suggests bug count as a "real-time" measure of quality. Kuan's [23] work also refers to bug counts as a measure. Bug count histories can be useful in identifying problematic areas of the code base. These bug counts can be determined either from bug tracking systems or from check-in histories. Khomh's work [28] also uses bug count on Mozilla Firefox releases as a measure of quality, along with uptime of the application.

It is then concluded that whilst the functional quality may be difficult to measure, the absence of quality is relatively simple to measure and can suffice as a measure of quality.

Another measure of quality is the effort required in maintaining software in operation - in particular the measure of effort for client support. Even if the effort is coarsely-grained against the customer for whom the effort is spent, this can still be useful as customisations can also be associated with a single customer.

2.2.1 Structural Quality Metrics

Some candidates for a unified structural quality model include the Maintainability Index [29, 30], or the Software Improvement Group (SIG) Maintainability Model [31, 32]. Sjoeberg's [33] work compares a number of Maintainability Models and concludes that such metrics are a poor predictor of maintainability and suggests that sophisticated metrics are overrated and instead advocates for the use of simpler metrics such as file and class size.

Bijlsma's work [34] investigates whether increased internal quality leads to faster

resolution time of issues for open source projects. In his work, Bijlsma tested hypotheses using a *Spearman's Rank Correlation*, stating that the maintainability ratings and issue resolution time do not follow a normal distribution. The Spearman's Rank Correlation correlates rising and falling trends instead of the actual measurements. Bijlsma used the SIG Maintainability Model in his work.

Nugroho's work [35] looks at the cost of repair and interest on technical debt using the SIG Maintainability Model, proposing the use of Software Maintainability as a measure of technical debt. Nugroho's work quantifies technical debt as the cost of correcting a shortfall in measured maintainability.

Lanza [36] introduces the *Overview Pyramid* which proposes the presentation of software metrics in a way that allows a collection of metrics to be presented in a "report". These reports present a collection of measures in three areas, namely: Size & Complexity; Coupling; and Inheritance. These areas all present ratios of metrics and Lanza's work also includes a set of benchmarked normal ranges for these measures. This work is useful for comparing and contextualising structural qualities of assemblies. More information on the Overview Pyramid can be found in Section 6.3. Given that Lanza's work allows for a comparison and focuses on simple size metrics (as per Sjoeberg's conclusion [33]), this makes his work a preferred candidate for measuring structural quality.

2.2.2 Functional Quality Metrics

Although the proposed measure of functional quality, or absence thereof, is number of bugs, it is also considered that there are other measures presented in literature. Note that none of these approaches are a direct measurement of quality, but are rather a means of attempting to determine a measure of *fitness for purpose*.

Potential metrics for measuring quality include Goal Question Metrics (GQM)[37, 38], which is a scheme for providing a measure of functional usefulness or fit for purpose metrics. Hall's work [39] furthers this approach and there are even some recent refinements to this approach such as those presented by Kelemen [40]. Kelemen's work proposes the term *Measurement Based Software Quality Assurance Framework*. At the core of these approaches is quantifying how *fit for purpose* a system is. These approaches may be useful in assessing the value of software customisations, however, would also require a survey involving persons who use the systems being measured.

Function point analysis is also considered as a method of measuring functional change.

A Function Point [41, 42] is a single unit of functionality in a business system, such as an input, output, search, etc. Klusener's work [43] investigates the measurement of function points from source code, particularly in the case of extension projects.

2.3 Conclusion

Quality in software is introduced and defined as a multi-faceted concept with aspects of both a product and a service and is constituted in different forms from internal and external (or structural and functional) perspectives.

Recognised quality standards specifications are presented as well as other work in literature that explore the measurement of software quality. Multiple measures exist for both structural and functional quality, including the Maintainability Index for structural quality and bug count for functional quality.

This chapter also considers the question of quantifying the size of changes to software.

Chapter 3

Research Question

The study is hypothesised on the idea that software customisation or modification negatively impacts the quality of software as measurable through an increased number of bugs.

Software vendors may choose to use a configurable architecture designed to accommodate such customisations or modifications as requested by customers. One such mechanism is using a dependency injection framework [44] that can be configured to override the original class types used at runtime with types defined in custom assemblies. In this way, all customers share a common basic functionality and the software is polymorphically extended with new behaviours that fulfil the new business requirements. A specific description of such an approach is discussed in Chapter 4.2.

Within this context, three research questions are identified for study:

R1 **Given a system that is designed for configurability through dependency injection to accommodate customisation, how does the structural quality of these customisations compare with the rest of the system?**

R2 **Given a system that is designed for configurability through dependency injection to accommodate customisation, what effect does implementing these customisations have on the functional quality as perceived by the user?**

R3 **Within the context of R1 and R2, how successful is the architecture in supporting the changes required by different customers?**

While it follows that the more software is customised, the more effort is required

to implement the customisations; the focus of these questions is rather on the quality effects. These customisations are applied from two principle mechanisms: Configurations, and Plug-ins.

Answers to the research questions may provide insight into how successful software customisation architectures are at accommodating the specific business needs of customers from a quality perspective.

3.1 Aim and motivation

This study is motivated by a desire for better quality in software as inspired by quality innovations in the manufacturing space.

The aim is to contribute towards a methodology that provides better data to inform business of a decision to build or buy when implementing a software solution. The term build or buy specifically refers to the decision of whether to develop a bespoke software solution or to use an off the shelf product.

Software is customised for a variety of reasons, some of which include mismatches between business requirements and the functionality provided by the software. It is hypothesised that these customisations can lead to problems of reduced functional quality problems and increased cost of maintenance.

If a model exists that can estimate the quality impact and therefore ultimately a cost component due to the quality impacts of customisation, more optimised software plans can be developed.

3.2 Scope

This study is limited to the scope of using existing software metrics for quantifying the customisations made to a software product and the effect that these changes have on the quality of the software as measured by code metrics.

This study will not be investigating concerns related to the customisations such as:

1. Why the software was chosen.
2. Why the business wants the customisations done.

3. Why a specification for customisations is poor.

4. Why the business processes have the requirements that they do.

Such concerns are important in quality, however, of interest is only the presence or absence of a quality problem, as opposed to how the quality problem occurred.

This is because the aim is to determine whether customising software reduces quality or not, rather than how the customisations came to be. This is then further rationalised by the notion of being able to inform decisions of whether to build or buy when it comes to implementing software solutions.

3.3 Theoretical Change Defect Model

In order to reason about the outcomes of the study, a model must be developed. The model begins with the consideration that a user will perceive poor quality when there are defects in the system, and that those defects are introduced through the customisation process. This process can be modelled statistically.

It must be noted that this model is not intended as a predictive model, but rather as a descriptive model in order to understand the problem space of the perceived quality of the system by the user. The model uses four variables to calculate a probability for a defect being precipitated by making changes to a system. The four variables are tabulated in Table 3.1. A statistical model diagram is illustrated in Figure 3.2 and some sample plots for the model are illustrated in Figure 3.3.

3.3.1 Models in Literature

Included in literature are two specific areas of work that are related to defect prediction. These models use empirical data and sophisticated statistical models to predict how many defects exist in a software system.

Fenton and Neil provide a critique of various defect prediction approaches [45], including regression models developed using size and complexity measures; testing data; process data; and combinations of these. Fenton and Neil favour the use of Bayesian networks [46] that use data collected from across the entire software development life cycle in order to predict defects.

Nagappan's work [47] presents a density defect regression model to predict the number of defects produced from a given amount of code churn for a system under development. The work is the product of a case study of a single large system for an industrial organisation, and concludes that code churn can be used to predict defect, even in individual binaries.

3.3.2 Statistical Model

Table 3.1: Components for the probability tree in figure 3.2.

Symbol	Description
n	Count of modified units (features)
N	Count of all units (features)
k	Probability of introducing an error in a unit
D	Probability of a dependency relationship between one unit and another

The model referred to in this section is for a hypothetical system which has features that a user may interact with. A feature in this system constitutes the base unit of "size" in the model.

A trial in the case of the statistical model is defined as a single interaction of a user in the system. In this trial, the user will interact with a single feature in a system made of a number of features. For simplicity, it will be assumed that the feature selection follows a uniform distribution. The total number of features in the system shall thus be defined as N. It is known then that in the customisation process, some of those features will have been modified which will therefore be defined as n modified features. This is somewhat analogous to the code churn of binaries in Nagappan's work [47].

Next, it is considered that there is a chance that a modified unit in the system has a defect. For simplicity, a uniform distribution of defects will also be assumed. The probability of defect will be defined by k. Literature [17, 48] indicates that a reasonable range for this value is approximately 0.05 to 0.1 for an experienced developer.

This model, however, is naive and does not consider the interactions between units in the system. The simplest extension to this model is to model a dependency of one unit on another unit in the system. This dependency then implies that a defect in the dependency will also be experienced in the dependant. This dependency can

also be modelled statistically, again assuming a uniform distribution for the purposes of simplicity. This probability of dependency is defined as D. The effects of this addition can be understood by exploring varying values of D.

The above model therefore has 6 classes of trial outcomes indicated in Figure 3.1.

Given the above mentioned definitions, a model can be represented using a probability tree in Figure 3.2. The probability of a "defect" outcome for the trial is the sum of all defect outcome leaf nodes of the tree which have been labelled as 1 through 6.

The symbol definitions for the model are tabulated in Table 3.1.

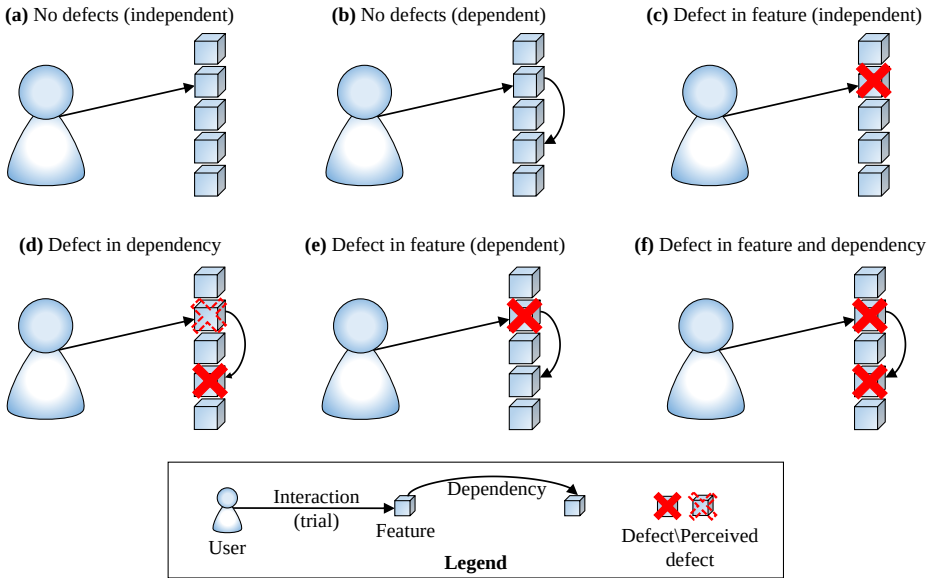


Figure 3.1: Classes of trial outcomes.

The plots for the model in figure 3.3 are produced using the functions in listing B.1.

3.3.3 Analysis

An analysis of the plots in Figure 3.3 indicates a hypothetical degradation of quality as might be perceived by a user given an increased probability of a defective unit in a trial.

First, and most intuitively, modifying more units results in an increased probability of the trial resulting in a defect in proportion with k.

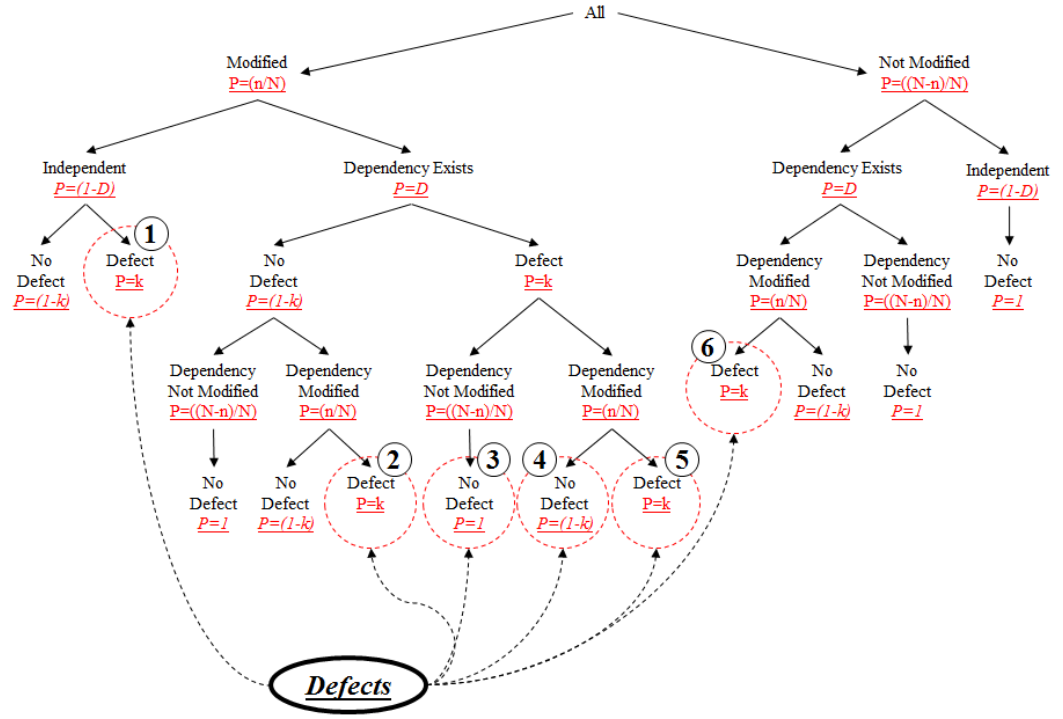


Figure 3.2: A probability tree that describes the proposed model.

Variations on D show that increased dependencies add an exponential factor to the degradation of quality. This is for a model where each unit has up to only one dependency, and so can be considered as a lower bound.

3.3.4 Pareto Principle

The Pareto Principle [49], also known as the 80/20 principle is a term coined in the early twentieth century from an observation in the late nineteenth century that 80% of land in Italy was owned by 20% of the people. Statistically, the principle follows a power law that can be modelled using a Pareto distribution.

This pattern has been noted in many different fields, including software engineering where, for example, 80% of software bugs typically come from 20% of the code.

This may make for a more realistic statistical modelling for the model in section 3.3.2. In the context of this model, the Pareto Principle would imply that 80% of dependencies are fulfilled by 20% of all units that are dependencies, and that 80% of the defects are present in only 20% of the units.

This would have a biasing effect that would add additional branches in the probability

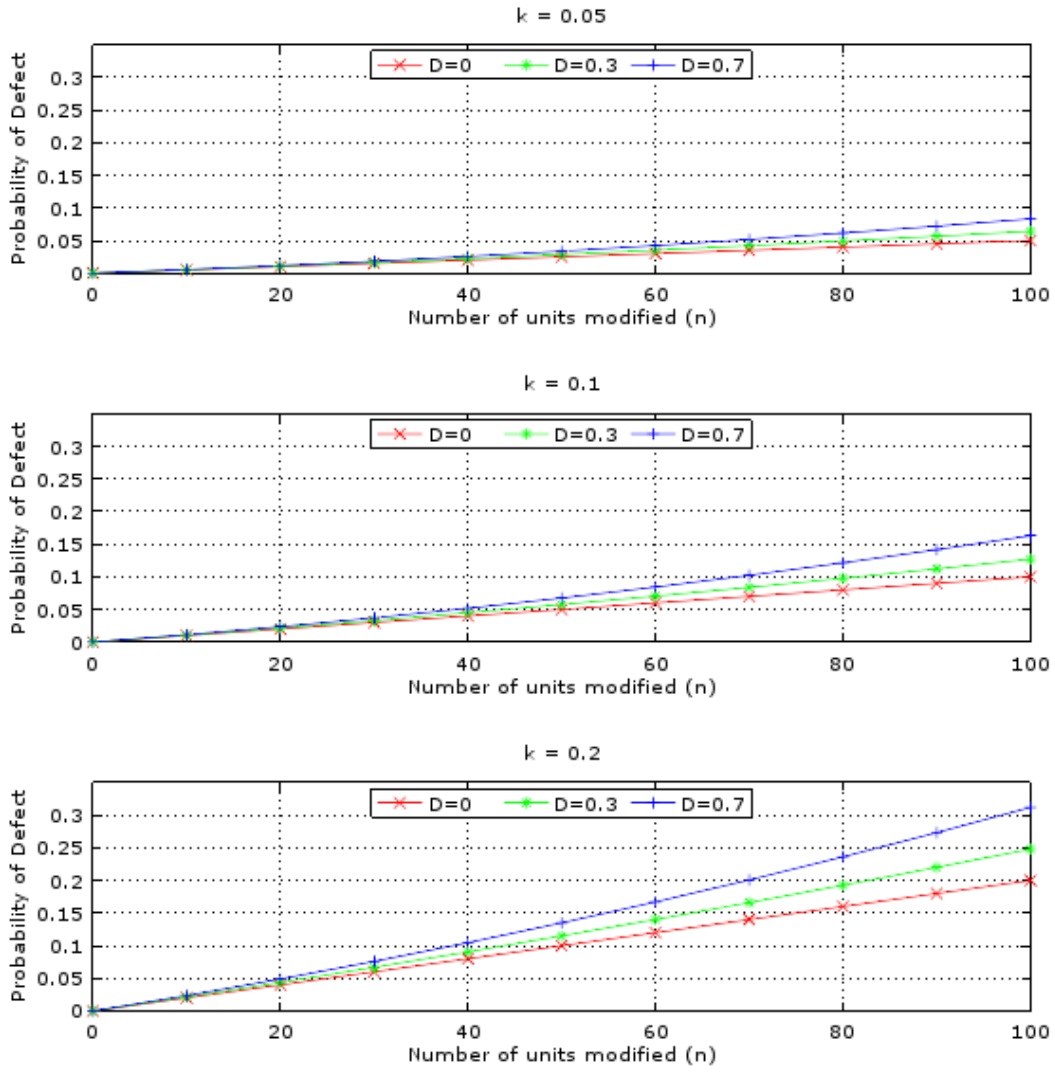


Figure 3.3: Plots of the model of *probability of defect* vs n for varying values of D and k for $N = 100$.

tree with probabilities of 0.8 and 0.2 (for example). These branches would occur specifically before *dependency* and *defect* nodes in figure 3.2. The tree of nodes below the new "Pareto" nodes would be duplicated across each side of the new branches.

Although adding a bias using a Pareto distribution may be more realistic, it would not change the key characteristic of the model since the bias is independent of n (the number of units changed). The exponential factor resulting from variations in D (probability of dependency) would still be present due to its dependency on n.

Therefore it can be concluded that applying the Pareto Principle to the model is not necessary.

3.3.5 Internal versus External Quality Effects

This study proposes that the internal quality remains relatively consistent whilst the external quality degrades with customisation.

Referring back to an understanding of the internal and external quality definitions in Section 2, it is reiterated that the internal, or structural quality is the quality as may be experienced by the developer of the software. This means the technical qualities of the code such as size, class coupling, cohesion, complexity, and the SOLID principles. The external or functional quality refers to the quality as may be experienced by the user. This means qualities such as correctness of function, ease of use and so on.

The model presented in Section 3.3 indicates a model for defect introduction following an assumption that a defect is almost unavoidable using a statistical model formulated using defect data in literature [50]. Thus it is necessary to account for why the internal quality may remain relatively constant, but the external quality reduces.

It is considered that functional quality is reduced conceptually by a few main causes:

1. Poor specification leading to incorrect function
2. Poor conceptual understanding of the domain leading to incorrect function
3. Unforeseen side-effects leading to logical errors

Whilst the structural quality is reduced by way of:

1. Violation of object oriented and software design principles
2. Inadequate code review for correcting poor design.

It can be argued that a strong knowledge of design skills (enhanced by developer training) and a strong code review practice means that the two causes mentioned above are addressed by the software process of the team. This process, however, will not be as effective at picking up logical errors from side effects (although these may be reduced by way of better structural quality), and is unlikely to be very effective in mitigating analysis related errors as none of the code review personnel are customers with insight into the intended function.

The implication is that a skilled team of software developers is not as effective at ensuring functional quality as it would be for structural quality.

Given that a customisation task results in new analysis work, opportunities for poor specification, or poor understanding are introduced. Furthermore, in the case of accommodating a customisation by way of a configurable component, teams are creating opportunities for logical errors from unforeseen side effects to be introduced.

Thus it is concluded that a hypothesis proposing that modifications to a system resulting in reduced functional quality whilst leaving structural quality relatively unaffected is not an unreasonable one. This does not consider the effects of other concerns such as technical debt.

3.4 Conclusion

The research questions have been stated, the aim and motivation, and the scope of the study have been presented.

A statistical model has been presented to explore hypothetical outcomes of the research. Using the model, internal and external quality effects of change on the software are also explored. It is hypothesized that structural quality will remain consistent, but functional quality will reduce at an increasing rate with an increased number of changes to the system.

Chapter 4

Methodology

This chapter discusses the methodology used in the study.

A description of the case study approach is provided, leading into specific goals for data collection to inform the research question. The weighbridge application under study is described as well as its configuration architecture; how the configuration will be examined and quantified; how quality will be inferred from the measured quantities; and how the research questions will ultimately be answered.

4.1 Approach

The study takes the form of a case study of a weighbridge application which is deployed to numerous customers in varying industries. The case study will be guided by literature from Fenton and Fleeger [37] as well as Easterbrook [51].

From Fleeger, it is appreciated that case studies are difficult to control and reproduce, however, the choice of a case study is validated by the notion that the software has been produced outside of the control of this study.

Given the work from Easterbrook, the research question is of a causal nature, and that the case study is to be used in an exploratory manner to investigate the phenomena of what effect modification of software has on its internal and external quality attributes.

Both Fleeger and Easterbrook validate the notion that specific, purposeful samples are taken as opposed to random sampling.

In this case, the customers selected in the case study have been selected for the

nature of their modifications and the developers' perception of the customers. The customers are described in more detail in Chapter 4.3.

To answer the research questions, the case study aims to determine:

1. How many bugs have been logged per customer?
2. What is different about the software deployed for each customer?
3. By how much does the software configuration and customisation differ between each customer?
4. How is the amount to which these customers differ relate to the number of bugs introduced for these customers?
5. How do the number of issues raised against that customer relate to other code metrics?

The number of bugs logged per customer will be used as the measure of quality for the software as experienced by the customer. As discussed in Chapter 2, measures for functional quality are somewhat difficult to find, although defect counts or densities have been used as a measure of quality.

One potential source of error in this methodology is that it assumes that the same process is applied by all customers when a defect is encountered. For example, the willingness to log a defect, or ability to identify or describe a defect for each customer may not be the same.

Given the aim of the study, in order to determine how the amount of customisation relates to the number of bugs introduced, a mechanism for measuring an amount of change is required. In order to determine a mechanism to measure the amount of change, an understanding of how the software is changed is first required.

Answering these specific questions may provide insight into the answers to the research questions for the study - specifically, gauging the success of a configuration architecture.

4.2 Description of Weighbridge Application

The weighbridge application is designed to manage the weigh-in and weigh-out of freight in order to verify that the load of a shipment that was dispatched matches

the load received at the destination. A major component of the product offering is integration into existing Enterprise Resource Planning (ERP) systems. These integrations will not be considered in the scope of this study due to poor access to data for the integrations.

A *shipment* is the central concern of the application domain which is contained within a *dispatch transaction*. The dispatch transaction contains link information to periphery concerns such as the vehicles participating in the shipment, vehicle drivers, description and mass of the payload. Other examples of information that can be tracked includes legislative compliance information, unique identifiers, and custom fields that can be configured to store arbitrary data.

The application has been in operation for several years in the mining and agricultural industries, and recently road ordinance (checking for overweight vehicles). The application is in its third major revision since its initial release. The application is written in C# and uses Microsoft SQL Server for data persistence. Revision control was previously managed by SVN [52], however, this has been migrated to Git. A full revision history is available.

As of its third revision (the revision under study), the application is designed to accommodate a series of customisations by way of features that can be configured. The configurable features include enabling and disabling visual controls; specifying validation rules; and custom commands on actions (eg, do “x” on save). In order to allow further customisation, such as custom commands on save, the application features an architecture that can dynamically load plug-in libraries, as specified by configuration files. These files are defined per customer. Some kinds of configurations simply specify options for built-in components, whilst other configurations use code injection using a compiled plug-in assembly. Configuration options that are “built-in” are referred to as *core* configurations, whilst the configurations that are sourced from plug-ins are referred to as *customised* configurations.

The plug-in architecture, however, does provide direct persistence for custom data where required, but this is instead accommodated through the use of data bags and generic data fields that can be used by a custom plug-in if required. Should generic data fields not be sufficient to accommodate the requirement, then the model is formally extended to accommodate the new requirement. One such requirement is from a road ordinance customer which introduced the need for an account which is linked to the shipment. This requirement was accommodated by creating formal concern in the domain model for accounts, to which *shipments* can be linked. The

notion of a shipment in the original application also changed from meaning a *weigh-out* and *weigh-in* to a single *weigh-in*.

4.3 Customer Selection

Of the available customers for the product, three specific customers have been selected for the case, prior to analysis. These customers have been selected based on the following criteria:

- Using a recent version of the software
- Degree of customisation

The recent version of the software is a criterion selected in order to ensure that an appropriate comparison is being made between the customers.

The degree of customisation is chosen in order to provide three different samples all of which have different levels of customisation. The target levels include:

1. Out the box (Little or no customisation)
2. Heavy Customisations
3. Domain Modified

4.3.1 C1 - Out the box

The out the box customer is one with little or no customisation and wishes to use the product "out the box" with at most a configuration of the built in feature controls. This then implies that any "custom" code for this customer should then lie mostly within the realm of configuration.

In this case it is Customer C1.

It is expected that this customer has the highest functional quality (i.e. Fewest bugs).

This customer operates in the agricultural sector and deals with shipments of agricultural produce.

4.3.2 C2 - Heavy Customisations

A customer with heavy customisations or integrations is one where the changes include a specific addition or change of features. These changes are presented with a higher number of rules defined for the customer than for C1, and includes plug-in code specified in the configuration.

In this case it is Customer C2.

It is expected that this customer presents a moderate degradation in functional quality (i.e. moderate number of bugs).

This customer operates in the mining sector and deals with shipments of ores.

4.3.3 C3 - Domain Modified

A customer which constitutes having a domain modified means that the customer's requirements introduce a change to the software which the original domain or design could not accommodate. These changes are therefore accommodated by introducing new domain concepts into the software that can be configured - in other words, this falls outside the realm of the configuration architecture.

In this case, it is customer C3. The customer in question requires a different process compared with the process originally intended by the software.

The originally intended software process is for a truck, loaded with goods to *weigh-out* from a location with a specific mass, and for the same truck to *weigh-in* at a destination and for the masses to be compared and checked that they match. Customer C3 is a roads development agency that changes this model because it only requires that all trucks to only ever *weigh-in*. Furthermore, in the original model, a single vehicle's data life cycle is limited to a single *shipment*, whereas customer C3 requires that the truck be added to a running *account* to which charges can be levied and tracked.

It is expected that this customer will have the most severe degradation of functional quality (most number of bugs).

This customer operates a national road network and uses the software to record and assist with the enforcement of compliance with vehicle weight limits.

4.4 Measuring Customisation and Quality

The amount of customisation can be quantified on a number of dimensions, including the number of defined configurations for the customer (both core and customised), as well as "how much" code (and other corresponding code metrics) have been written for any customisations.

Given that each customer's installation is customised by way of configuration files, the amount of customisation can be measured by the number of configured items for the customer. These can also further be classified by way of a built-in option, and custom plug-in.

4.4.1 Measurements

Measurements can be sourced from a variety of artefacts. The potentially relevant sources of information for this work include:

- Design documents - Business Requirements Specifications, UML diagrams
- Source code
- Configuration Files
- Revision history
- Database schemas

The database schemas offer little interest since if a change to the database schema in the system is required, then the concern will be formally adopted into the application, effectively making such code new core code.

Whilst the design documents may seem useful, a focus on the metric extraction from the code and configuration files is preferred. This is because the configuration files are XML formatted which is easier to parse and analyse than the free text specification documents. The configuration files also use very specific and consistent terms (since they configure the application) which make natural grouping categories for analysis.

Thus the data sources of interest used in the study are the source code; configuration files; and revision history, and the following measurements are extracted from these sources:

- **Source Code**
  - Maintainability Index - per module
  - Overview Pyramid - per module
    - Lines of code
    - Cyclomatic Complexity
    - Count of Packages
    - Count of Classes
    - Count of Methods
    - Count of Calls to other Classes
    - Count of Fanout
    - Average Depth of Inheritance
    - Average Hierarchy Height
- **Configuration Files**
  - Count of total rules
  - Count of rules configured to use custom code
  - Count of core configurations unique to customer
  - Count of customised configurations
- **Jira**
  - Count of issues per customer

More detail on these metrics can be found in Chapter 6.

4.5 Determining Quality from the Metrics

The quality will be determined in two dimensions: A measured set of code metrics; and the measured number of defects. This allows the determination of structural quality (from code metrics) and functional quality (from defects).

The extraction of established code metrics such as the Maintainability Index (MI) or the SIG Maintainability Model will provide an objective measure of how the internal, structural quality of the application has changed. An MI measurement can be performed using Microsoft Visual Studio [53], which is significant as the application is written using .NET technologies and thus making MI the preferred

metric. The MI measurement will be captured for both modifications and additions allowing for a further comparison of the effect of each on quality. In other words, it can then be determined whether the nature of a change (addition or modification) has any impact on structural quality.

Quality impacts may also be inferred by changes to the code model which violate software engineering principles such as SOLID [54, 55]. For example, introducing new concerns to a class would violate both Single Responsibility and the Open/Closed Principle. Violations of the Liskov Substitutability principle are an indication of possible unexpected behaviours at runtime. This, however, can only be inspected manually.

Such changes are in principle precipitated by the need to make a feature built into the application configurable. This is because a customer will request a specific change that may not yet be accommodated by some form of configuration, and therefore changes must be made to provide a configuration that will accommodate the customer's needs.

4.6 Determining the Success of the Architecture

A configurable architecture aims to reduce the effort in accommodating the customer's requirements, whilst still delivering a high level of functional quality to users.

Given the quality metrics for both functional and structural quality, the data can be used to make an assessment regarding the success of the architecture from a quality perspective.

A success or failure is then determined by comparing the resulting qualities for varying degrees of customisation. For both structural and functional quality, the architecture can be considered a success if they are unaffected by the customisation.

In the case of structural quality, a successful architecture then means that the architecture is able to provide space with-in the software to accommodate changes without compromising the maintainability of the software.

In the case of functional quality, a successful architecture then means that developers have enough freedom to effect functional change in the software to meet the business requirements. This means that the architecture does not impede the development team's ability to correctly express custom business requirements or business rules,

and that the team is able to do so without introducing unintended side effects.

4.7 Conclusion

The methodology of the study is presented. The study takes the form of a case study of a weighbridge application. A description of the application and its architecture is provided and the three classes of customer for study are provided.

The three classes of customer include an out the box customer, heavily modified customer, and domain modified customer.

Customisation and quality are measured using metrics calculated from the source code and configuration files. Source code is used to calculate the Maintainability Index and Overview Pyramid measurements and the configurations are used to count the number of customisations and configurations.

The success or failure of the architecture is determined by comparing the resulting quality as measured for each of the customers.

Chapter 5

Sizing Customisations

This chapter describes the way in which customisations are implemented in the application as well as how these are then sized so that they can be compared.

Sizing the customisations for each customer will allow the customisations to be reasoned with in a quantified way.

5.1 Nature of Customisations

The application supports a diversity of customisations ranging from branding and UI label changes to custom process definitions, validations, and behaviours.

Such configurable behaviours are facilitated by the dependency injection [56] and the Model - View - View-model (MVVM) [57] pattern used, illustrated in Figure 5.1.

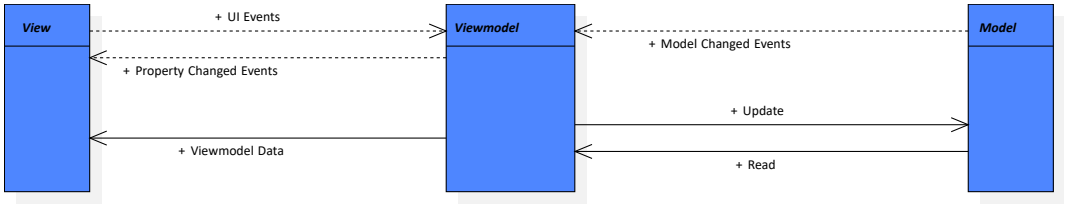


Figure 5.1: The Model - View - View-model pattern

The technical details of how customisations are implemented and provided is described in Chapter 5.2.

From Figure 5.1 the view, view-model and model are all fully fledged objects. The view publishes UI events down to the view-model, and reads data and property

updates from the view-model - which may include data such as the label text for a control; event bindings for buttons; or validation rules; etc.

The view-model serves as an intermediary between the model and the view and is principally responsible for the orchestration of, or translating concerns between the view and the model. The view-model may also contain UI logic be managed by the view alone such as calculated properties or event handlers.

The model is typically the application domain but can be any data source.

An application which uses dependency injection can perform a runtime binding between the view, view-model and model and also use a factory method pattern to source these objects for binding. Using a factory method means that the construction of the objects can be controlled dynamically and thus provides the architecture for configuration.

5.2 How the Application is Customised

The application is customised for each customer by specifying *configurations* for the application. Configurations are specified in XML files. The configuration files contain a collection of *rule contexts* that are identified by a *key* in order that the application *modules* can identify the appropriate rule contexts to configure the module. The rule contexts can contain a *view-model*; a *view*; *model*; and other rule contexts. The rule context itself is a container for a module's configuration and a single rule context counts as a rule.

The view-model configuration can contain *settings* whose value will override a default value defined within the view-model in the module. The type and assembly name is used to resolve which property the configuration will be assigned to within the module. View-Models can also contain injectable *strategies* (or commands which will also be referred to as strategies) that define the behaviour of particular UI *events* (e.g. Save). Strategies are the means by which customer-specific plug-in code is injected into the customer's specific runtime, although there are also *core* behaviours which are common to multiple customers (which can be thought of as the default behaviour). All custom strategies for a customer are distributed in a single assembly.

The view configuration can contain *control settings*, whose value will override a default defined setting for a visual control in the UI. Similar to the view-model, the

type and assembly name is used to resolve the type to be configured.

Model objects do not contain configurations.

The structure of the configurations is outlined in Figure 5.2 and an example shown in Listing 5.1. Figure 5.3 illustrates how the configuration files are more concretely realised into a configured application, specifically as a view.

Listing 5.1: Sample Configuration XML

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="ruleContextSection" ... />
  </configSections>
  <ruleContextSection>
    <ruleContexts>
      <ruleContext key="Backdated" type="..." category="Shipment">
        <ruleContexts>
            ...
          <viewmodel name="..." assemblyName="..." />
          <model name="..." assemblyName="..." />
        </ruleContext>
        </ruleContexts>
        <viewmodel name="..." assemblyName="...">
          <container>
            <register type="..." mapTo="...">
              <property name="CanSaveStrategies">
                <array>
                  <dependency type="..." />
                   ...
                </array>
              </property>
              <property name="BeforeSaveCommands">
                ...
              </property>
              <property name="AfterSaveCommands">
                 ...
              </property>
            </register>
          </container>
        </viewmodel>
        <model name="..." assemblyName="..." contextKey="..." />
      </ruleContext>
    </ruleContexts>
  </ruleContextSection>
</configuration>
```

In summary, the application is customised for a customer by defining rules which can:

- Set configurable options within the application

- Inject custom code that will be executed on specific actions

5.3 Measuring the Customisations

Given how the application is customised, the amount of customisation can be measured by:

- Counting the number of rules
- Counting the number of settings and control settings
- Counting the number of core and plug-in strategies
- Measuring the size (LOC) of the client's plug-in assembly

The above measurements are performed by parsing the configuration XML into a C# program and performing some aggregates on the data and loading the data in Excel as a Comma Separated Value (CSV) file. This is performed by producing matching C# classes which the XML configurations are deserialised into using the standard .Net Framework deserialisers. This is different from how configurations are typically read into .Net applications using the application configuration components of the framework.

The number of rules is determined by counting the number of rule context objects.

The number of settings and control settings are determined by summing the total number of settings and control settings, grouped by the type name of the view-model.

The strategies are extracted by counting the number of *dependency* elements per customer, including the source assembly per strategy. This data was then augmented using Microsoft Excel to check whether the assembly refers to the customer or not, indicating whether it is a core or a plug-in strategy.

The LOC for each customisation is taken using the LOC code metric from the plug-in assembly. The plug-in assembly is determined from type references in the XML configuration - for example, in Listing 5.1, the `type` attribute for the `dependency` node.

These measurements therefore provide a measurement of the size of the customisations for the customer.

5.4 Conclusion

Technical details of the customisation using configuration and dependency injection are provided.

The size of the customisation is determined through a combination of counts including the number of rule, settings, plug-ins, and lines of code in custom plug-ins.

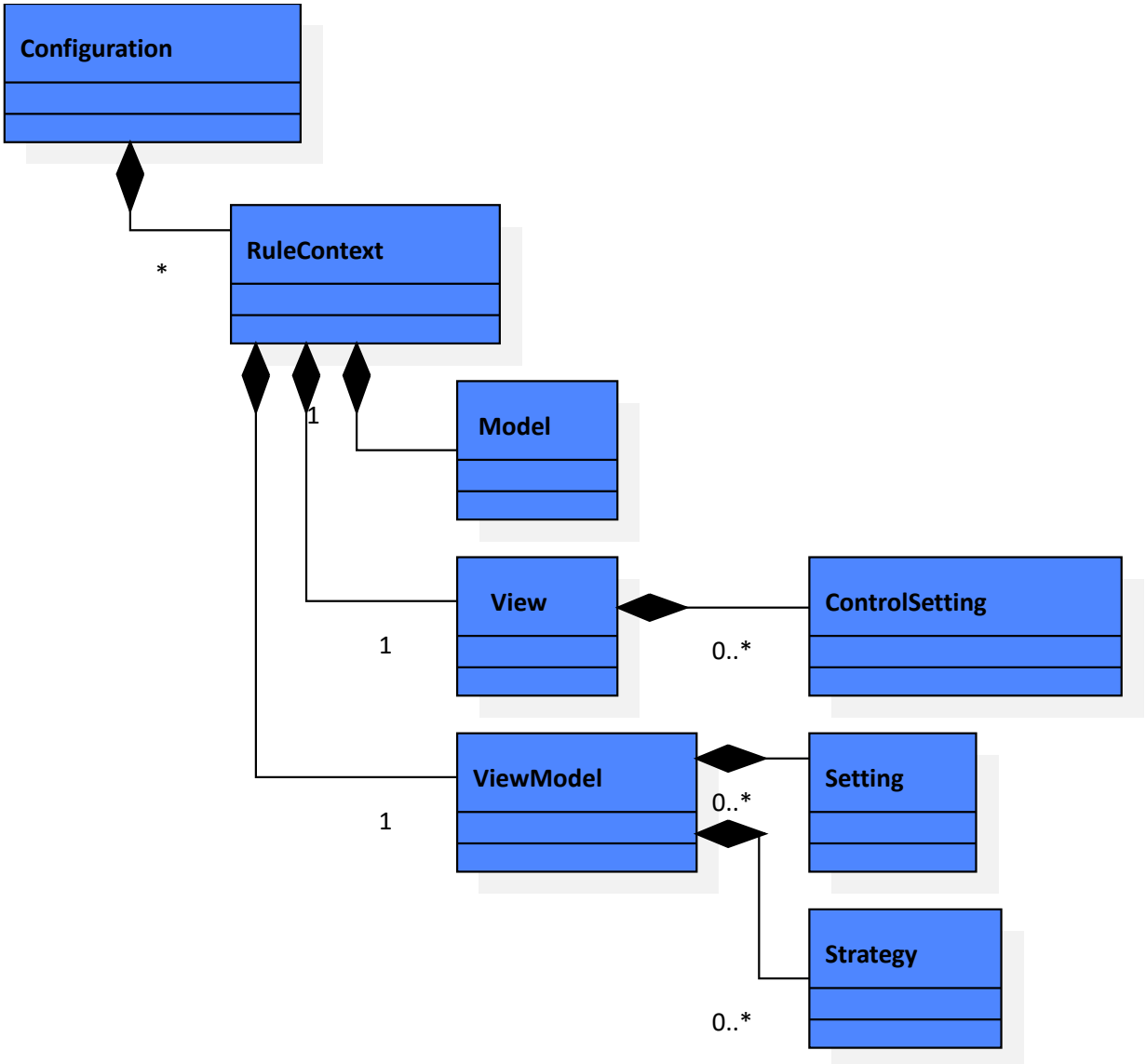


Figure 5.2: Pseudo class diagram illustrating the structure of configurations

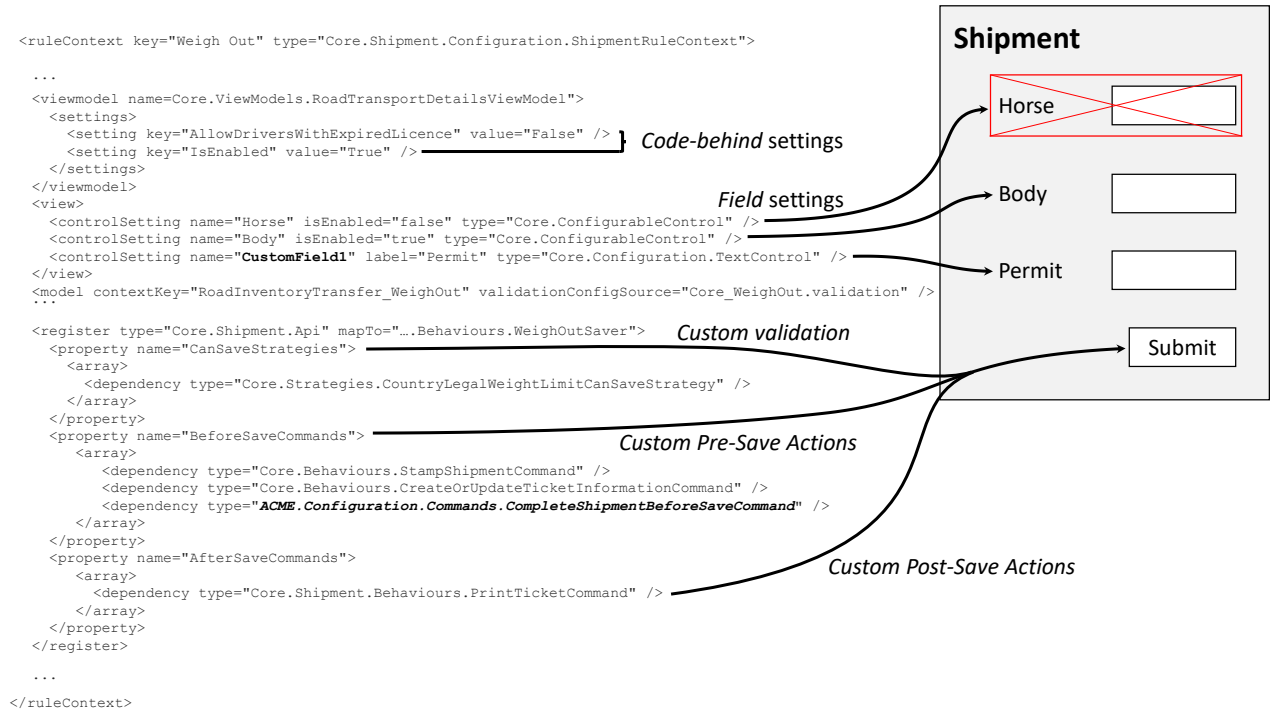


Figure 5.3: Illustration of how configurations relate to the configuration of the application. Individual pieces of XML represent actual configurations, however this view shows an abridged configuration for brevity.

Chapter 6

Code Metrics

This chapter discusses the code metrics used in the study. Code metrics are a selected mechanism for assessing the internal quality of a system.

The collection and calculation of metrics is discussed, including the use of Visual Studio tooling, and implementation of other metric tools, such as the recalculation of the maintainability index and overview pyramids.

6.1 Acquiring Metrics

Some of the desired metrics can be acquired using Visual Studio tooling. In particular, the `FxCop` [58, 59] metrics calculator was used on the relevant assemblies in order to produce XML files which contain metrics for the assembly. `FxCop` will produce the following metrics at varying levels of detail (Assembly; Namespace; Type/Class; Member/Method):

1. Maintainability Index
2. Cyclomatic Complexity
3. Class Coupling
4. Lines of Code
5. Depth of Inheritance

See A.2 for more information on `FxCop`.

The lines of code and Depth of Inheritance measures are self-describing.

The cyclomatic complexity is best described as the number of branches in a program, including method calls.

The class coupling metric refers to the number of classes that the unit under measure collaborates with. In other words, the number of distinctly referenced classes in the code. Both class and interface types constitute a coupling.

The Maintainability Index metric (discussed in detail in 6.2) provides an indication of how maintainable an application is.

Additional metrics can be calculated using code analysis tools as discussed for producing the *Overview Pyramids*.

6.2 Maintainability Index

Of note is that the Maintainability Index as produced by Visual Studio is not identical to the SEI promoted index introduced by Oman and Hagemeister in 1992 but has 2 minor differences [60]. The first is that Oman and Hagemeisters' original index had a range of 0 to 171 [61]. The Visual Studio Team has normalised this metric to 0 to 100. Secondly, the original index also included a factor for the number of comments in the code [29], but this is not included in the Visual Studio Metric.

The index itself is the result of a study performed by Oman and Hagemeister which involved a regression analysis on several software systems written in C and Pascal. A range of metrics were gathered for each system, and a maintainability survey conducted on these systems. Their regression analyses were verified against six other software systems not included in the original eight systems which produced the model. The intention of the work was to determine which software metrics are good predictors of maintainability.

The study presented a one, four and five metric polynomial model (a metric per term) for predicting maintainability. These regression models were assessed for their accuracy in predicting maintainability under a wide range of conditions, aiming to ensure that an excess of one of the factors results in an over or under prediction of maintainability. Oman and Hagemeister modified their 4 metric polynomial in the Coleman paper [29] to instead use the average Halstead volume over the average Halstead Effort, citing "that the volume is a non-decreasing function with

concatenation". Thus the paper defines maintainability as:

$$\begin{aligned} MI = 171 & \\ & - 5.2 \times ln(HV) \\ & - 0.23 \times CC \\ & - 16.2 \times ln(LOC) \\ & + (50 \times sin(\sqrt{2.46 \times COM})) \end{aligned} \tag{6.1}$$

The symbols as in equation 6.1 are as follows:

Table 6.1: Symbols for maintainability index in equation 6.1

Symbol	Explanation
HV	Halstead's Volume
CC	Cyclomatic Complexity
LOC	Average LOC per Module
COM	Average comments per LOC

6.2.1 Halstead Volume

The Halstead Volume is not used directly in this study, but is instead used as part of the Maintainability Index.

The Halstead Volume is one of a set of metrics introduced by Howard Halstead in 1977 [62] and is the product of the *Program Length* and the logarithm of the *Program Vocabulary*, which are functions of the number of operands and operators.

Loosely speaking operands are variables and constants whilst operators are everything else.

If a program such as in Listing 6.1 is considered, the operators include elements such as `()` / `begin..end`, `>`, `<`, `and`, `if..then..end`, `else`, `=`, and the new lines terminating each statement. The operands are then `a`, `b`, `3`, `5` and `1`.

Halstead's definition of program vocabulary (η) is the sum of the distinct operators and operands which is:

Listing 6.1: Sample Ruby program for Halstead metrics

```
1 if (a > 3) and (a < 5)
2 then
3         begin
4                 b = 5
5         end
6 else
7         b = 1
8 end
```

$$\eta = 9 + 4 = 13 \tag{6.2}$$

The definition for the program length(N) is the sum of the total occurrences of each operator and operand, which is:

$$N = 11 + 7 = 18 \tag{6.3}$$

And thus the program volume is defined as:

$$V = Nlog_2\eta = 67 \tag{6.4}$$

A base two logarithm of the vocabulary is taken since this is the minimum number of bits needed to uniquely identify each identifiable concern in the program. This number, multiplied by the program length then provides a language independent view of the size of a program.

Given Equation 6.4, the Halstead volume will increase linearly with the length of the program and logarithmically with each new concept introduced.

In summary, the Halstead Volume provides a measure of the size of the program independent of language and character set.

6.2.2 Visual Studio Implementation

Oman and Hagermeister's 1994 report [61] indicated that the percentage of lines which were comments had a bearing on the maintainability of the system, thus raising questions regarding the validity of the Visual Studio maintainability index metric. Within the scope of this study, however, it is not the absolute maintainability, but the relative maintainability which is of interest. Questions have also been raised with respect to the ranges which Visual Studio defines as "Red", "Yello" and "Green" and what these categories mean. A Microsoft blog post [63] indicates that the red and yellow categories are set low in order to filter out noise for automated build tools so that only units with a clear maintainability problem are detected - as such the category information for the maintainability index need not be captured for the purposes of this study.

One motivation for excluding comments from the metric could be the in-line *XML Code Doc* in C# (that lives in the code files) which would create an unusually large number of comments per LOC creating an apparent increase in code maintainability as created by documentation rather than comments.

The Visual Studio definition of the maintainability index is:

$$MI_{\mathtt{vs}} = MAX\left[0,\big(171 - 5.2 \times ln(HV) - 0.23 \times CC - 16.2 \times ln(LOC)\big) \times \frac{100}{171}\right] \tag{6.5}$$

The *MAX* function will limit maintainability to a minimum of zero. Whilst this is possible given the definition of the maintainability index, it is relatively meaningless as the program is already so large and/or complex that it will not be maintainable.

6.2.3 Generated Code

The weighbridge application makes use of a code generator to provide domain classes and basic architectural concerns such as: object relational mapping; and service

contracts. This means that a significant portion of the Create, Read, Update, and Delete functions (CRUD) are managed by the generated code for both server and client side of the application.

In order to prevent the generated code from skewing the results of the customised code (which have no generated components other than are common to the application), these should be filtered out. In order to filter these components out, the aggregated roll up metrics must be recalculated from the type member level metric data.

Cyclomatic Complexity, Class Coupling and Lines of Code are all simply additive, and the aggregated depth of inheritance is the maximum value of all its child measures.

The *Maintainability Index*, however, is not only dependent on logarithms of the lines of code and Halstead volume, but is also computed using *average values* at aggregated levels, and so has a non-linear, and therefore non-additive, result based on those measures. Whilst the Lines of Code and Cyclomatic Complexity are already available, the Halstead Volume is not. However, the Halstead volume can be recovered from the Maintainability Index using the Cyclomatic Complexity and Lines of Code:

$$HV = e^{MI'} \tag{6.6}$$

$$where$$

$$MI' = \frac{171 - 0.23 \times CC - (16.2 \times ln(LOC)) - 1.71 \times MI}{5.2}$$

Although Visual Studio uses a non-linear MAX function, in practice this has no effect unless the Maintainability Index is less than zero - which is very unusual and does not occur in the dataset for the current case study.

The formula in equation 6.6 can be verified by using the calculated Halstead Volume to recover the identical Maintainability Index using equation 6.5 - at both base and aggregated levels.

Thus the aggregated Maintainability index can be recalculated excluding specific type members from the assembly.

In order to determine which type member records to exclude from the metric data, each record indicates the source file from which it is compiled. Since all generated code for the application lives under the `generated` directory in each project, generated code can be excluded by means of filtering all members who's source file contains

`generated`.

6.3 Overview Pyramid

One of the problems with most of the metrics gathered using Visual Studio is that they require some method of normalisation in order to make a judgement with regards to the quality of the customisations compared with the core product.

The Maintainability Index is a standalone metric and can be used comparatively. However, more information can be gleaned using the other metrics of the code. By using the Overview Pyramid [36] from Lanza's work on Object Oriented metrics, not only can a normalised view of the code quality be created which can then be compared, but also judgements can be made regarding the quality of that code according to the meaningful thresholds defined in his work.

The Overview Pyramid uses normalised ratios of raw metrics to provide a comparative view of the object-oriented metrics. The Pyramid has three sections:

- Size and Structural complexity
- Coupling
- Inheritance

The coupling metrics require the number of operation calls and the number of called classes (`FANOUT`). The `FANOUT` is an available metric given the `coupling` metric from Visual Studio, however, the number of `CALLS` is not available, and thus the coupling metrics cannot be computed unless additional steps are taken to acquire this metric. This can be done by means of using tools that provide code analysis, such as `Roslyn` or `Mono.Cecil` [64] to interrogate either the source or compiled CIL. This can be used to produce the required metric and augment the given metric set.

The inheritance metrics require the inheritance depth details which can be acquired using a similar technique to the `CALLS` metric, and once again, augment the measurement set.

This then provides all the data necessary to produce an *Overview Pyramid* for each assembly and compare quality.

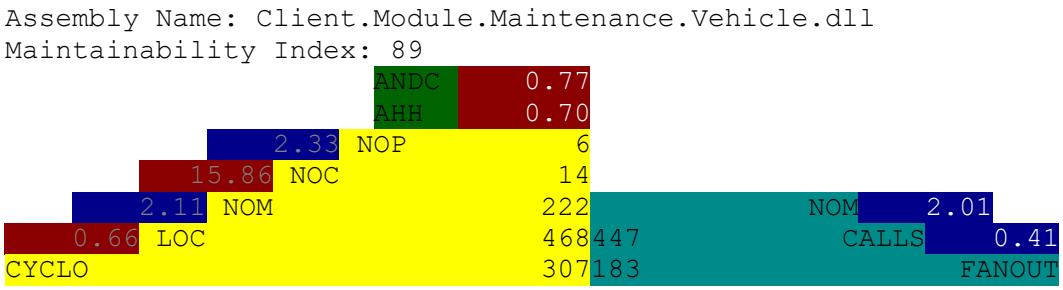


Figure 6.1: An example overview pyramid annotated with the maintainability index and the source assembly name.

6.3.1 Example

An example overview pyramid is illustrated in Figure 6.1.

The pyramid has been produced using a C# program that outputs the calculated metrics to console for inspection. The pyramids are annotated with the maintainability index and assembly name for convenience.

6.4 Tool Verification

In order to verify that the custom written tool's metrics are correct, a test is performed against a case that is small enough to be verified manually and the results compared against those collected by the tool.

The sample program is contained in appendix A.

A table showing a comparison between the manually and tool collected metrics from a sample application are contained in table 6.2.

6.4.1 Notes on Metrics

Visual studio will count an interface as a class when determining the number of classes. This is because the metrics tool uses the compiled CIL to perform the measurements on, and an interface is internally represented as a class. These interfaces then have a cyclomatic complexity of 1 per method since each method represents a branch in

code. Other language features such as getters, setters, and default constructors will also introduce a cyclomatic complexity of 1 per feature instance.

The class coupling metric (`FANOUT`) counts all references, inheritances, and type checks on fields. Type checks on properties, however, will not result in an increased coupling because the calling method is decoupled from the actual type by the getter method. Inheritances are counted because the constructor must also make a call to the constructor of the base class. This does not happen for interfaces because interfaces do not have constructors. Each getter or setter will also indicate a coupling of one if it is of another type. Visual Studio will also indicate a point of coupling for library types (e.g., `System.Console`). However, the coupling is rolled up in a namespace or module according to the number of unique classes in that scope, which means that the tool must use the total sum of coupling for each class. An interesting issue was also found where the FxCop tool was giving the correct numbers, but Visual Studio itself sometimes came short.

Lines of code is (contrary to what it may seem) a non trivial measurement. This is not a measure of the absolute lines of code, but rather a measure of the size of code that lives inside methods, i.e. ignoring class definitions, declarations and other language "features". Specifically, Visual Studio will perform an estimated count of the number of lines of code based on the compiled CIL. Although this will then not include things like class definitions or declarations, it will, however, include "invisible" code such as default constructors and initialisers. Thus it is difficult to accurately size what the Visual Studio lines of code should be, however, consistent sizing is what is most important. The supplied measure was counted by examining the CIL

Table 6.2: Comparison of tool and manual counts of metrics for the test case.

Metric	Manual	Tool	VS or Custom
Cyclomatic Complexity	21	21	VS
Number of classes	8	8	VS
Number of packages	2	2	VS
Number of methods	18	18	VS
Lines of code[a]	26	29	VS
Avg hierarchy height	0.50	0.50	Custom
Avg dependant classes	0.33	0.33	Custom
Fanout	10	10	VS(coupling)
Calls	9	9	Custom

[a]Discrepancy is explained in Chapter 6.4.1

in IlSpy [65] and counting the number of `call` or `newobj` instructions, as well as an extra count per looping construct.

6.5 Application Specific Metrics

The application under study also has its own set of metrics that can be measured. These include the following:

- How many rules are defined
- How many *strategies* are defined
- How many times a piece of custom code is used to define a behaviour

Strategies refer to the *strategy pattern* [2] which essentially describes a behaviour that is injectable through an interface. These strategies are used to react to particular events in the application, such as saving a *shipment*.

6.6 Roslyn and Mono.Cecil

Both `Roslyn` and `Mono.Cecil` provide the ability to perform an analysis on code and gather metrics.

The two technologies are different, however, both can provide an abstract syntax tree from the code for analysis. `Roslyn` produces a tree from C# source code, and `Mono.Cecil` produces the tree from compiled CIL.

`Mono.Cecil` is used in this study as it is a technology of familiarity. Although this only allows for analysis of the CIL instead of the actual source code, the key structures under measure are equivalent.

6.7 Conclusion

The Overview Pyramid and extraction of code metrics from the application is presented. Metrics are extracted using a combination of Visual Studio and custom tooling.

The custom tooling aims to augment the Visual Studio provided metrics in order to provide Overview Pyramids. The custom tooling is developed using `Mono.Cecil` to inspect compiled IL.

Details of the computation of the Maintainability Index including specifics of the Visual Studio implementation is presented and a technique for removing specific modules from an aggregate Maintainability Index is provided.

Chapter 7

Bug and Issue Analysis

Using the data stored in the Git [66] version control history and a Jira [67] database, it is possible to collect bug and task data for the application. This data includes useful information such as:

- The type of issue (Task, Bug)
- Issue Number
- Summary and description
- Date that the task was captured
- The specific customer for whom this issue is relevant to
- Severity of the issue

An analysis of the data provides insights into the functional quality of the system - that is, the quality of the functioning of the product as may be perceived by the users.

The data collected spans from November 2015 to April 2017. This provides approximately 16 months of operational data for the product.

7.1 Git

Git is a version control system which allows developers to store a code base and track changes between the revisions to the code base. These change sets are accompanied by log entries that are used to describe what the purpose of the changes are. Git

effectively allows one to travel back in time against the source code, or to search through the changes that have been applied to it.

Change sets that fix bugs can be found by searching for the bug issue numbers in the git log. Git can provide a summary report indicating which files (and therefore classes) were changed, as well as indicating the number of lines added or removed in the change to that file. This is useful for sizing the change sets. This is similar to an approach used by Zhang [68] to extract defect counts from open source project which do not have a bug tracking database. Note, these lines are not directly comparable to the IL lines, although are close enough.

7.2 Jira

Jira is a software engineering tool from Atlassian which provides functions such as bug tracking and task management.

The project team uses Jira to assist with planning and tracking the work performed for development work and bug fixes. When a bug is logged in Jira, a note is made against a corresponding customer for whom the issue is relevant for. Code commits that fix bugs logged in Jira reference the issue number for the bug, thus making commits for bugs resolvable against the bug by searching the Git log history.

Jira allows a Comma Separated Value (CSV) formatted download of the data for analysis. The analysis of the data can be performed using Microsoft Excel Pivot tables to aggregate data. This facilitates the measurement of count of bugs by customer, for example.

7.3 Conclusion

A bug and issue analysis is performed using the data contained in the bug tracking software Jira and the version control software Git. The Jira data is exported to CSV format and bug issue types filtered out. Bugs are sourced from Git using the commit history log and where possible correlated to Jira using the issue tracking number in the commit message.

Chapter 8

Results

The metrics show a somewhat unexpected result that in retrospect is also an intuitive one.

Results show how the increasing number of bugs in the system are correlated with the increasing number of rules defined for the customer. Results are also interpreted in the context of the research questions and some findings made that inform the conclusion.

A label and description for each customer is indicated in Table 8.1.

Table 8.1: Label and description of customers for the case study

Customer Label	Description
C1	Out the Box
C2	Heavy Configuration
C3	Domain Modified

8.1 Bugs and Commits

An analysis of the commit and bug data shows a strong correlation, however, given that commits were selected for analysis because they were commits for bug fixes, this is not useful.

Typically a single bug, and occasionally two bugs were fixed in a single commit.

8.2 Bugs Logged Per Customer

The measure of bugs logged per customer was sourced from the Jira data. The number of issues per customer is shown in Table 8.2 and Figure 8.2.

Table 8.2: Bug counts per customer

Customer	Bug Count
C1	38
C2	123
C3	25

8.3 Measure of Rules and Customisations

The customisation size measures are shown in Figure 8.2. The numbers measured are listed in Table 8.3

Table 8.3: Counts collected for rules per customer. The LOC refers to the LOC for plug-in assemblies.

Customer	Plug-ins	Rules	LOC	Controls	Settings	Strategies
C1	2	385	1572	164	253	16
C2	28	642	205	492	516	115
C3	31	362	3587	198	522	44

The LOC in Table 8.3 for C2 is seemingly low since the rules defined refer mostly to code that is part of the core application as opposed to code in a custom assembly. Specifically, although the rules defined are specific to the customer, the code implementing those rules is not.

8.4 Rules Versus Bugs

In an analysis of the initial three cases, a strong correlation between the number of rules and bugs is presented whilst the Maintainability Index seems to be relatively unaffected by the number of rules added, as illustrated in Figure 8.1.

Given the data, adding more rules appears to cause more bugs in the system. However, correlating the number of bugs to the quantity of code in the custom configurations presents with an inverse correlation. This initially seems surprising since the hypothesis is that more custom code leads to more bugs due to a chance of introducing an error.

In order to understand the observed results in the context of the hypothesis, it is observed that more custom configurations result in fewer bugs. Inspecting the files which are modified in order to resolve the bugs shows that the files updated are in the core configurations and not the custom configurations. A drawn conclusion is that writing code to specifically accommodate a rule for only that customer results in fewer bugs than would otherwise be introduced had a core configuration been introduced.

Another interesting result is that C3 has the lowest number of bugs, although it was expected that C3 would have the highest number of bugs. C1 has fewer bugs than C2 as expected. The expectations were driven by the degree of changes made for each customer, and C3's changes include a change to the system domain.

The data also presents a very high correlation between the number of bugs and number of commits, however, this is expected because these commits were allocated against the customer using bug issue number. Thus by virtue of this methodology, bugs cause commits.

Referring back to the theoretical model plotted in Figure 3.3, it was expected that the negative effects on quality would have an exponential component, although in the results plotted in Figure 8.1 they indicate an apparent linear relationship with number of rules changed - which would be analogous to the number of changed units in the theoretical model. This can be explained by having a low D value in practice, suggesting that there is a low dependency between rules in the system. This is validated by noting the system's decoupled module design. Individual strategies are specific to modules and are not shared between modules.

8.5 Analysis

This section covers an analysis that attempts to glean insight from the recorded metrics.

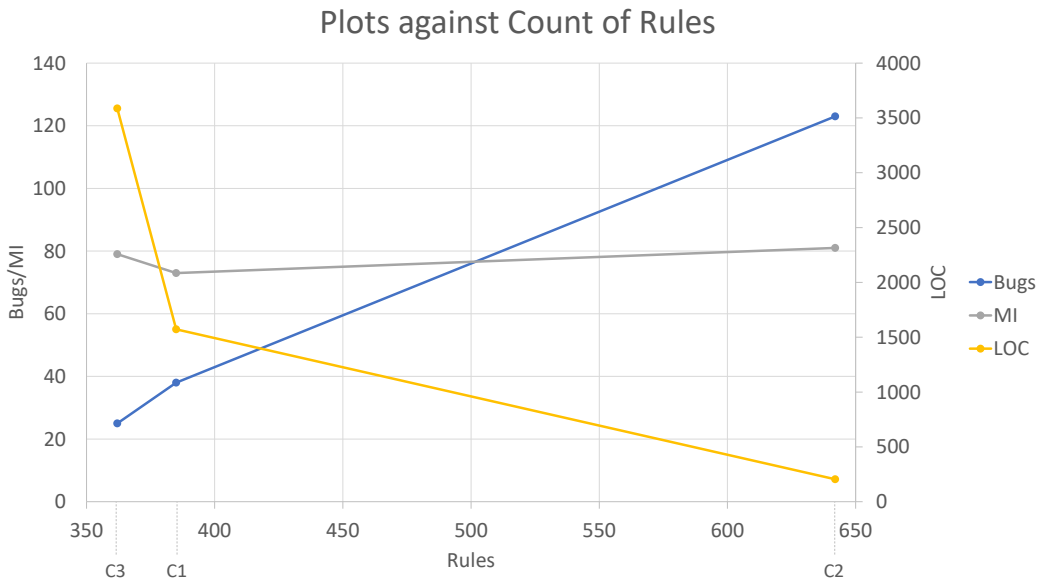


Figure 8.1: Plot illustrating a correlation between rules and bugs with inverse correlation against plug-in lines of plug-in code. Note that the multipliers are provided in order that they can be plotted in a range that is comparable with the rest of the metrics.

The quality of code is compared in three areas, namely:

- Core: The configurable module code
- Plug-in: The plug-in strategy code
- Other: Other assemblies in the application

The gathered metrics are summarised by customer in Figure 8.2.

8.5.1 Size Comparison

A comparison of sizes for the different areas is indicated in Table 8.4. The amount of module (core) and plug-in code is surprisingly high, however, this also suggests that a lot of effort has been put into making the application configurable.

8.5.2 Quality Comparisons

Considering the code metrics outlined, it is noted that the maintainability index is relatively constant across all three areas, although slightly lower for plug-ins in Table

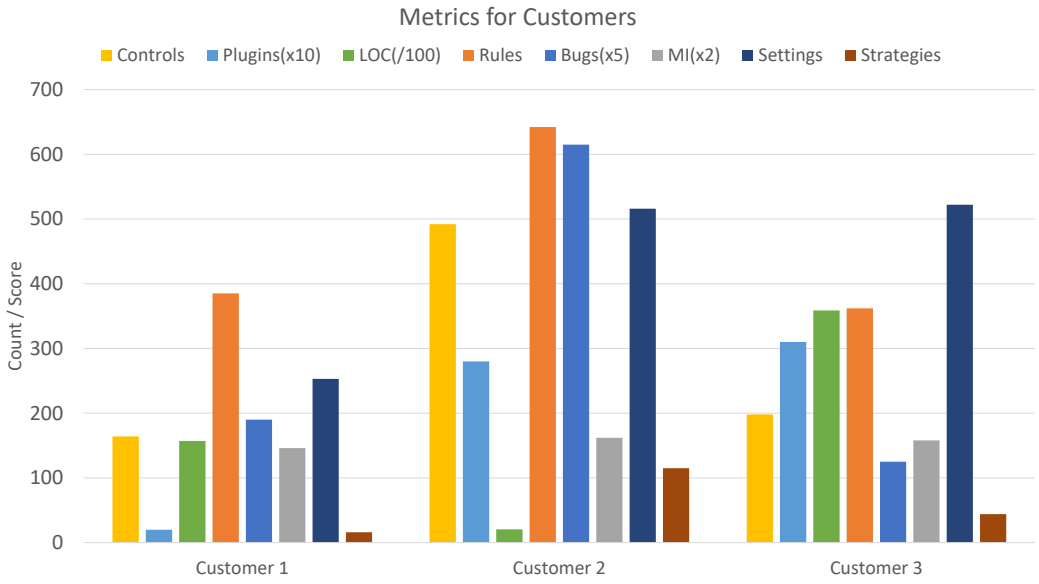


Figure 8.2: Summary of metrics for each customer

Table 8.4: Table of comparison of size by area as a total across all customers in the case study.

Area	LOC	Assembly Count
Core	11,790	32
Plug-in	20,092	28
Other	60,586	43

8.5.

Table 8.5: Table of comparison of maintainability index averaged over the customers under study.

Area	MI
Core	90
Plug-in	81
Other	91

The next quality metric of interest is the cyclomatic complexity of the code. Using Lanza's overview pyramid, the number of modules in each rating is counted for each area, shown in Table 8.6. Notably, most code falls in the *High* category, however, only about 64% of plug-in assemblies are rated at that complexity.

The plug-in code includes modules that have longer methods that have higher levels of fanout per call and more calls per method when compared with other areas.

A listing of overview pyramids can be found in Appendix C.

Table 8.6: Table of comparison complexity rating counts per assembly across all customers under study (from the overview pyramids) by area for all metrics.

Area	High	Medium	Low
Cyclomatic Complexity Per LOC			
Core	32	0	0
Plug-in	18	1	9
Other	43	0	0
LOC Per Method			
Core	-	-	32
Plug-in	7	3	18
Other	-	-	43
Calls Per Method			
Core	8	9	15
Plug-in	16	-	12
Other	2	4	37
Fanout Per Call			
Core	2	8	22
Plug-in	9	1	18
Other	16	1	26

8.5.3 Correlations

The metrics are correlated by customer. As discussed in Chapter 2.2.1, a Spearman rank correlation is used to perform the correlations as defects are not normally distributed, as per Bijlsma's work [34]. It is noted that the Spearman rank correlation correlates increasing and decreasing rank trends and not the metrics themselves. Given that there are only three samples, (and the ranks are therefore 1, 2 and 3) it is expected then that correlations will be rounded to factors of 50%.

The most significant correlation results are:

- An increase in rules strongly correlates with an increase in bugs.
- An increase in the amount of customer-specific code has a strong correlation with a decrease in the number of bugs - suggesting that customer-specific code is more successful at correctly accommodating customer business needs.

Table 8.7: Spearman's Rank Correlations for gathered metrics

	Bugs	Rules	Plug-ins	MI	LOC	Controls	Settings
Rules	100%	-	-	-	-	-	-
Plug-ins	-50%	-50%	-	-	-	-	-
MI	50%	50%	50%	-	-	-	-
LOC	-100%	-100%	50%	-50%	-	-	-
Controls	50%	50%	50%	100%	-50%	-	-
Settings	-50%	-50%	100%	50%	50%	50%	-
Strategies	50%	50%	50%	100%	-50%	100%	50%

- An increase in the amount of customer-specific code also correlates strongly with a decrease in the number of rules - suggesting that changes are accommodated using either code or rules.

A more detailed analysis is discussed in the results for the research questions.

8.6 Research Questions

Given the collected data the research questions can now be considered within the context of the results. Final assessments are presented in the conclusion.

8.6.1 R1 - Structural Quality

The structural quality effects are measured using the Maintainability Index and remains relatively unaffected by the amount of customisation across the three customers.

It is noted however that the structural quality of plug-in code is of a marginally lower level than for the rest of the system as can be seen in Table 8.5. Given that the complexity of the plug-in code is lower as seen in Table 8.6, it can be inferred that the decreased maintainability index is caused by a combination of longer methods and higher fanout and calls which would result in an increased Halstead Volume, and reduce maintainability (see Chapter 6.2).

8.6.2 R2 - Functional Quality

The functional quality effects are in particular considered with respect to the manner in which the customisations have been implemented.

Results indicate that defect counts are lower in cases where fewer rules are defined and more customer-specific code is implemented, than for cases where more rules are defined and less customer-specific code.

Of note is that the plug-in code is typically of lower complexity than core code (see Table 8.6). This may explain why a larger amount of customer-specific code correlates strongly with fewer bugs raised. An argument can be made hypothesising that the increased complexity of the core code is a product of an attempt to accommodate multiple cases through configuration.

The increased fanout and calls per method may be the result of orchestration work being done in the plug-in code which needs to interact with multiple areas of the system.

8.6.3 R3 - Architectural Success

The quality results are mixed.

Structural quality indicates that the structural quality is relatively unaffected by the customisation.

Functional quality data indicates that the functional quality is decreased with an increased number of defined rules. The quality reduction also inversely correlates with the amount of customer-specific code.

Results suggest that better functional quality is achieved by writing customer-specific code rather than providing a system with configurable modules. The architecture is successful in that it allows for this customer-specific code to be injected, but is not successful in that the quality is reduced as more configurable modules are consumed for a customer.

It is concluded that customers may have experienced better quality had all customer-specific needs been met with customer-specific code, rather than providing configurable modules that are configured for that customer.

As indicated in the review of the functional quality effects analysis, this may arise out of an increased complexity in providing configurable modules.

8.7 Threats to Validity

A number of threats to validity [51] are identified.

The largest threat to validity is an external threat, presented by a case study consisting of a single product across only three customers. The data presented for these cases may not represent a general population of customers and software systems.

An internal threat to validity is that the full nature of variability between customers cannot be completely known. For example, quality is measured by defect and thus assumes that defects are logged to each customer using an equivalent process. This can only be known through greater engagement with the software life-cycle, including the vendor and customer staff. This may also be offset by including more cases in the study.

Another potential internal threat to validity is that the collected data treats all counts uniformly. For example, all rules and bugs are treated equally. Specifically, the given correlations are for general cases rather than delving further into more detailed categories which may assist in compensating for variability in the rules and bugs.

8.8 Conclusion

Correlations in the case study and the answers to the research questions are presented.

Structural quality as measured by the maintainability index is consistent at 90 and 91 for the shared code (core and other) and is slightly reduced for custom plug-ins at 81. The Maintainability Index is unaffected by the number of rules defined. It is therefore concluded that structural quality is not affected by the customisation.

The functional quality as measured by defect count is negatively affected by increased configuration. Increased bug count is correlated with increased rules and with decreased custom plug-in code.

The architecture is successful in providing a high maintainability index for customised code, however, it is unsuccessful in maintaining a high functional quality through configuration.

Chapter 9

Conclusion

A case study has been performed on software designed with a configurable architecture. The cases under study include three customers of the software, each with a varying amount of customisation and configuration.

Quality was measured along two dimensions, namely structural (internal) quality, and functional (external) quality. The internal quality was measured using the maintainability index, as well as size and complexity measures from Lanza's Overview Pyramid. The external quality was measured as a quality reduction through the number of bugs.

The number of bugs was measured by counting the number of bugs per customer logged in the bug tracking system and referenced in the source control logs.

The amount of customisation was measured by measuring the number of configured elements and the size of custom code.

Key findings for the research questions are:

R1 - Structural quality effects: The structural quality has been found to be relatively unaffected by the number of defined rules. Customer-specific source code also retains a comparable level of structural quality when compared with other customers and the original source code., albeit slightly reduced.

R2 - Functional quality effects: The functional quality appears to decrease with an increased number of defined rules, and increase with an increase in the amount of customer-specific code.

R3 - Success of the architecture: The architecture is regarded as only being partially successful as structural quality remains resilient against the use of configurable modules and plug-ins. In terms of functional quality, the architecture appears to be less successful for cases of using configurable modules to be shared among customers, with the number of bugs increasing with the number of defined configurations (rules).

9.1 Future Work

Whilst the above set of measurements would appear to support the hypothesis. It is noted however that the customers selected were selected in order to satisfy the criteria outlined for Customers C1, C2 and C3.

Thus it suggests that a further investigation into the nature of the results for the additional customers, or to extend the study to other projects. Suggestions include open-source projects which have a high degree of configurability and where developers are diligent in the tracking of bugs in some form of bug or task tracking software.

Bibliography

[1] K. E. Emam and A. G. Koru, "A replicated survey of IT software project failures," *IEEE Software*, vol. 25, no. 5, pp. 84–90, 2008.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Abstraction and reuse of object-oriented design*. Springer, 1993.

[3] D. Chappell, "The three aspects of software quality." `http://davidchappell.com/writing/white_papers/The_Three_Aspects_of_Software_Quality_v1.0-Chappell.pdf`, 2011. Accessed: 17 September 2017.

[4] T. Ravichandran and M. A. Rothenberger, "Software reuse strategies and component markets," *Communications of the ACM*, vol. 46, no. 8, pp. 109–114, 2003.

[5] B. Light, "The maintenance implications of the customization of ERP software," *Journal of software maintenance and evolution: research and practice*, vol. 13, no. 6, pp. 415–429, 2001.

[6] C. Guido and R. Pierluigi, "A methodological proposal to assess the feasibility of ERP systems implementation strategies," in *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*, pp. 401–401, IEEE, 2008.

[7] M. V. Kumar, A. Suresh, and P. Prashanth, "Analyzing the quality issues in ERP implementation: a case study," in *Emerging Trends in Engineering and Technology (ICETET), 2009 2nd International Conference on*, pp. 759–764, IEEE, 2009.

[8] M. Xin and N. Levina, "Software-as-a-service model: Elaborating client-side adoption factors," in *Proceedings of the 29th International Conference on Information Systems, Paris, France*, Dec. 2008.

[9] W. Sun, X. Zhang, C. J. Guo, P. Sun, and H. Su, "Software as a service: Configuration and customization perspectives," in *Congress on Services Part II, 2008. SERVICES-2. IEEE*, pp. 18–25, IEEE, 2008.

[10] A. Dubey and D. Wagle, "Delivering software as a service," *The McKinsey Quarterly*, vol. 6, no. 2007, p. 2007, 2007.

[11] R. Seethamraju, "Adoption of software as a service (saas) enterprise resource planning (ERP) systems in small and medium sized enterprises (SMEs)," *Information systems frontiers*, vol. 17, no. 3, pp. 475–492, 2015.

[12] A. B. Benevides, "An ontologically well-founded framework for modelling business organizations, processes and services.," in *ICBO*, Citeseer, 2012.

[13] H.-J. Happel and S. Seedorf, "Applications of ontologies in software engineering," in *Proc. of Workshop on Sematic Web Enabled Software Engineering"(SWESE) on the ISWC*, pp. 5–9, Citeseer, 2006.

[14] M. M. Al-Debei and D. Avison, "Developing a unified framework of the business model concept," *European Journal of Information Systems*, vol. 19, no. 3, pp. 359–376, 2010.

[15] T. R. Gruber *et al.*, "A translation approach to portable ontology specifications," *Knowledge acquisition*, vol. 5, no. 2, pp. 199–220, 1993.

[16] T. R. Gruber, "The role of common ontology in achieving sharable, reusable knowledge bases," *KR*, vol. 91, pp. 601–602, 1991.

[17] S. McConnell, *Code complete*. Microsoft press, 2004.

[18] H. V. Vliet, *Software Engineering - Principles and Practice*. John Wiley & Sons Ltd, third ed., 2008.

[19] ISO, ISQS, "ISO/IEC 25010. 2011," *Systems and software engineering—Systems and software Quality Requirements and Evaluation (SQuaRE)—System and software quality models*, 2011.

[20] R. Ferenc, P. Hegedűs, and T. Gyimóthy, "Software product quality models," in *Evolving Software Systems*, pp. 65–100, Springer, 2014.

[21] M. Davis and J. Heineke, *Operations Management: Integrating manufacturing and services*. McGraw-Hill/Irwin, fifth ed., 2005.

[22] B. Freimut, C. Denger, and M. Ketterer, "An industrial case study of implementing and validating defect classification for process improvement and quality management," in *Software Metrics, 2005. 11th IEEE International Symposium*, pp. 10–pp, IEEE, 2005.

[23] J. Kuan, "Open source software as lead user's make or buy decision: a study of open and closed source quality," *Stanford Institute for Economic Policy Research, Stanford University*, 2002.

[24] D. R. Cox and D. Oakes, *Analysis of survival data*, vol. 21. CRC Press, 1984.

[25] M. Shepperd, C. Schofield, and B. Kitchenham, "Effort estimation using analogy," in *Proceedings of the 18th international conference on Software engineering*, pp. 170–178, IEEE Computer Society, 1996.

[26] E. Coelho and A. Basu, "Effort estimation in agile software development using story points," *International Journal of Applied Information Systems (IJAIS)*, vol. 3, no. 7, 2012.

[27] R. Singham and D. H. Steinberg, *The ThoughtWorks® Anthology: Essays on Software Technology and Innovation*. Pragmatic Bookshelf, 2008.

[28] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality?: an empirical case study of Mozilla Firefox," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pp. 179–188, IEEE Press, 2012.

[29] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer*, vol. 27, no. 8, pp. 44–49, 1994.

[30] K. D. Welker, "The software maintainability index revisited," *CrossTalk*, vol. 14, pp. 18–21, 2001.

[31] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pp. 30–39, IEEE, 2007.

[32] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, no. 2, pp. 287–307, 2012.

[33] D. I. Sjøberg, B. Anda, and A. Mockus, "Questioning software maintenance metrics: a comparative case study," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pp. 107–110, ACM, 2012.

[34] D. Bijlsma, M. A. Ferreira, B. Luijten, and J. Visser, "Faster issue resolution with higher technical quality of software," *Software quality journal*, vol. 20, no. 2, pp. 265–285, 2012.

[35] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, pp. 1–8, ACM, 2011.

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

[37] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach: Brooks*. Cole, 1998.

[38] V. R. Basili and H. D. Rombach, "The tame project: Towards improvement-oriented software environments," *Software Engineering, IEEE Transactions on*, vol. 14, no. 6, pp. 758–773, 1988.

[39] T. Hall and N. Fenton, "Implementing effective software metrics programs," *IEEE software*, no. 2, pp. 55–65, 1997.

[40] Z. D. Kelemen, G. Bényász, and Z. Badinka, "A measurement based software quality framework," *arXiv preprint arXiv:1408.3253*, 2014.

[41] A. J. Albrecht, "Measuring application development productivity," in *Proc. of the Joint SHARE/GUIDE/IBM Application Development Symposium*, pp. 83–92, 1979.

[42] A. J. Albrecht and J. E. Gaffney, "Software function, source lines of code, and development effort prediction: a software science validation," *IEEE transactions on software engineering*, no. 6, pp. 639–648, 1983.

[43] S. Klusener, "Source code based function point analysis for enhancement projects," in *2013 IEEE International Conference on Software Maintenance*, pp. 373–373, IEEE Computer Society, 2003.

[44] E. Razina and D. S. Janzen, "Effects of dependency injection on maintainability," in *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA*, p. 7, 2007.

[45] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on software engineering*, vol. 25, no. 5, pp. 675–689, 1999.

[46] N. Fenton, M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause, and R. Mishra, "Predicting software defects in varying development lifecycles using Bayesian nets," *Information and Software Technology*, vol. 49, no. 1, pp. 32–43, 2007.

[47] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pp. 284–292, IEEE, 2005.

[48] W. Hayes and J. W. Over, "The personal software process (PSPSM): An empirical study of the impact of PSP on individual engineers.," tech. rep., DTIC Document, 1997.

[49] M. Iqbal and M. Rizwan, "Application of 80/20 rule in software engineering waterfall model," in *Information and Communication Technologies, 2009. ICICT'09. International Conference on*, pp. 223–228, IEEE, 2009.

[50] W. S. Humphries, *PSP: A self-improvement Process for Software Engineers*. Addison-Wesley, first ed., 2005.

[51] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," *Guide to advanced empirical software engineering*, pp. 285–311, 2008.

[52] Apache Software Foundation, "Apache™ subversion®." `https://subversion.apache.org/`, Jan. 2018.

[53] Microsoft Corporation, "Visual Studio 2013." `https://visualstudio.microsoft.com/`, Jan. 2018. Version 12.

[54] R. C. Martin, "Principles of OOD." `http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod`. Accessed: 29 October 2015.

[55] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[56] M. Fowler, "Inversion of control containers and the dependency injection pattern." `https://martinfowler.com/articles/injection.html#ServiceLocatorVsDependencyInjection`, Jan. 2004.

[57] J. Smith, "The Model-View-ViewModel (MVVM) design pattern for WPF," *MSDN Magazine.[Online] February*, 2009.

[58] J. Kresowaty, "FxCop and code analysis: Writing your own custom rules," 2008.

[59] Microsoft Contributors, "FxCop - What's new in Visual Studio 2013." `https://blogs.msdn.microsoft.com/devops/2013/07/03/what-is-new-in-code-analysis-for-visual-studio-2013/`, Jan. 2018. Version 12.021005.1.

[60] A. van Deursen, "Think twice before using the "maintainability index"." https://avandeursen.com/2014/08/29/think-twice-before-using-the-maintainability-index/, Aug. 2014. Accessed: 17 September 2017.

[61] P. Oman and J. Hagemeister, "Construction and testing of polynomials predicting software maintainability," *Journal of Systems and Software*, vol. 24, no. 3, pp. 251–266, 1994.

[62] M. H. Halstead, *Elements of software science*, vol. 7. Elsevier New York, 1977.

[63] Microsft Contributors, "Maintainability index range and meaning." https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/, Nov. 2007. Accessed: 17 September 2017.

[64] Mono Project, "Mono.Cecil." https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/, Jan. 2018. Version 0.9.6.0.

[65] Open Source Contributors, "IlSpy - .Net decompiler." http://www.ilspy.net/, Jan. 2018.

[66] Open Source Contributors, "Git." https://git-scm.com/, Jan. 2018.

[67] Atlassian Pty Ltd (ABN 53 102 443 916), "Jira — issue & project tracking software." https://www.atlassian.com/software/jira, Jan. 2018.

[68] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 182–191, ACM, 2014.

Appendix A

Technical Detail of Calculation of Code Metrics

A.1 Introduction

There are a variety of options with regards to the collection of metrics from the source code. This appendix covers the technical details of how the metrics for the study are gathered.

Some metrics are provided by `FxCop` which ships with Microsoft Visual Studio and supplementary metrics have been calculated with a custom tool using the C# code inspection library `Mono.Cecil`. The supplementary metrics are used to complete the Overview Pyramid.

A sample program is also provided in order to verify that the metric calculations are correct.

A.2 `FxCop`

`FxCop` is a tool provided by Microsoft which can be used on build servers to perform arbitrary code inspections on builds. Microsoft provides a set of packaged FxCop rules which produce code metrics (packaged as `metrics.exe`). In particular, it can calculate the following metrics for your code:

1. Maintainability Index

2. Cyclomatic Complexity
3. Class Coupling
4. Lines of Code
5. Depth of Inheritance

These metrics are also aggregated and available at the following levels:

1. Target (Assembly)
2. Module
3. Namespace
4. Type
5. Member

The metrics are output as XML and can be easily parsed using an XML library for further augmentation, filtering, or aggregation.

A.3 Mono.Cecil

`Mono.Cecil` is a library for inspecting and modifying .NET assemblies. Of particular interest are the functions that allow the inspection of the byte code. These inspections can tell you for example if a type is a `class`, `interface` or `enum`; or allow you to examine each instruction in a method body, including whether the instruction is a method call, the name of the method and the class it belongs to.

A.3.1 Calculating the `CALLS` Metric

The CALLS metric is the *distinct number of calls to other classes per method.* Using `Mono.Cecil`, each body of each type's methods can be interrogated and calls to other classes add to a set. The total length of this set can then be stored in a map keyed by the interrogated method's full name. This then allows for the required metric of the number of operation calls to be measured.

A.3.2 Calculating the ANDC Metric

The ANDC metric is the *average number of derived classes* measure. This can be measured again using `Mono.Cecil` to interrogate each type that is not an `interface` or `enum`. For each type, the base class is recursively iterated until the base class `System.Object` is found, and a map of counts keyed by full class name is incremented for the matching key (if missing, the key is added). The final metric is then calculated by summing the values of the map and dividing by the number of class types.

A.3.3 Calculating the AHH

The AHH metric is the *average hierarchy height* measure. This can be measured alongside the ANDC metric, by tracking the depth of recursion, and storing that number in a map, keyed by the last non "System.Object" type name. If the map already contains a value for the type name, then the larger of the two values is stored. The final metric is then calculated by taking the sum of the values in the map and dividing that number by the number of values in the map.

A.4 Sample Program

The sample program does nothing useful and instead provides a collection of relationships between classes and methods that is small enough to be able to perform manual counts against. A class diagram for the application can be seen in figure A.1.

```
1  //AnInterface.cs:
2          using System;
3          using System.Collections.Generic;
4          using System.Linq;
5          using System.Text;
6          using System.Threading.Tasks;
7
8          namespace Application
9          {
10             public interface AnInterface
11             {
12                 void DoSomethingElse();
13             }
14         }
15
16 //FooBase.cs:
17         using System;
18         using System.Collections.Generic;
19         using System.Linq;
```

```
20          using System.Text;
21          using System.Threading.Tasks;
22
23          namespace Application
24          {
25              public class FooBase
26              {
27              }
28          }
29
30  //Program.cs:
31          using System;
32          using System.Collections.Generic;
33          using System.Linq;
34          using System.Text;
35          using System.Threading.Tasks;
36          using Application.AnotherNamespace;
37
38          namespace Application
39          {
40              class Program
41              {
42                  static void Main(string[] args)
43                  {
44                      var baz = new Baz();
45                      var bar = new Bar();
46                      var foo = new Foo(bar, baz);
47
48                      foo.FoosTheBars();
49                  }
50              }
51          }
52
53  //UnusedInterface.cs:
54          using System;
55          using System.Collections.Generic;
56          using System.Linq;
57          using System.Text;
58          using System.Threading.Tasks;
59
60          namespace Application
61          {
62              public interface UnusedInterface
63              {
64                  void DoSomething();
65              }
66          }
67
68  //AnotherNamespace/AnotherBar.cs:
69          using System;
70          using System.Collections.Generic;
71          using System.Linq;
72          using System.Text;
73          using System.Threading.Tasks;
74
```

```
75          namespace Application.AnotherNamespace
76          {
77              class AnotherBar : Bar
78              {
79              }
80          }
81
82  //AnotherNamespace/Bar.cs:
83          using System;
84          using System.Collections.Generic;
85          using System.Text;
86
87          namespace Application.AnotherNamespace
88          {
89              public class Bar
90              {
91                  private int num = 0;
92
93                  public int FooTheBar()
94                  {
95                      return num++;
96                  }
97              }
98          }
99
100 //AnotherNamespace/Baz.cs:
101         using System;
102         using System.Collections.Generic;
103         using System.Text;
104
105         namespace Application.AnotherNamespace
106         {
107             public class Baz
108             {
109                 public void UseTheFoo(int sum)
110                 {
111                     int mod = 0;
112
113                     for (int i = 0; i < 10; i++)
114                     {
115                         mod += sum % i;
116                     }
117
118                     Console.WriteLine(mod);
119                 }
120             }
121         }
122
123 //AnotherNamespace/Foo.cs:
124         using System;
125         using System.Collections.Generic;
126         using System.Text;
127
128         namespace Application.AnotherNamespace
129         {
```

```
130             public class Foo : FooBase, AnInterface
131             {
132                 public Bar Bar { get; set; }
133                 public Baz Baz { get; set; }
134                 bool isBarFlag = false;
135 
136                 public Foo(Bar bar, Baz baz)
137                 {
138                     Bar = bar;
139                     Baz = baz;
140                 }
141 
142                 public void FoosTheBars()
143                 {
144                     int sum = 0;
145                     while (sum <= 10)
146                     {
147                         sum = Bar.FooTheBar();
148                     }
149 
150                     Baz.UseTheFoo(sum);
151                 }
152 
153                 public void DoSomethingElse()
154                 {
155                     if (Bar is Bar)
156                     {
157                         isBarFlag = true;
158                     }
159                 }
160 
161                 public void AnotherMethod()
162                 {
163 
164                 }
165             }
166         }
```

A.5 Conclusion

Technical details of the code metric calculations are presented. Metrics provided by `FxCop` are supplemented with the output of a custom program built using `Mono.Cecil`.

Calculations have been validated with a sample program which is small enough to be able to perform the measurements by hand. The sample program has been designed to provide a variety of cases fit measurements such as Average Number of Derived Classes, Average Hierarchy Height, and Calls.

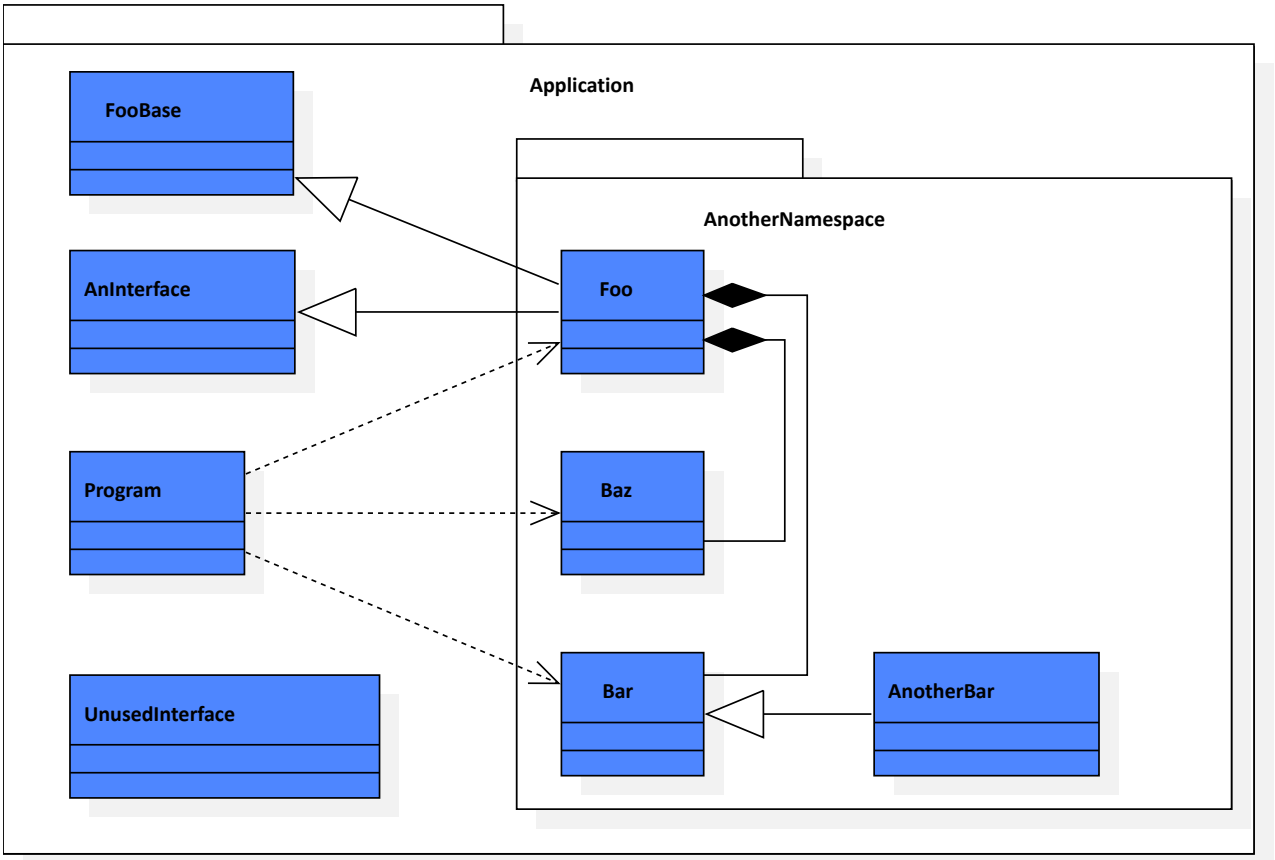


Figure A.1: Class diagram for the sample program

Appendix B

Matlab Code for Hypothetical Model

B.1 Introduction

A Matlab program has been produced to explore and produce plots of the hypothetical defect model presented in chapter 3.

The program provides a combined function that can be evaluated for all parameters of the model.

B.2 Program Listing

The program listing that provides a function implementing the model is presented below:

Listing B.1: Matlab functions for the probabilistic model

```
>>function y=p1(n,N,D,k);
y=(n./N).*(1-D).*k;end;
>>function y=p2(n,N,D,k);
y=(n./N).*D.*(1-k).*(n./N).*k;end;
>>function y=p3(n,N,D,k);
y=(n./N).*D.*(k).*((N-n)./N).*k;end;
>>function y=p4(n,N,D,k);
y=(n/N).*D.*k.*(n./N).*(1-k);end;
>>function y=p5(n,N,D,k);
y=(n./N).*D.*k.*(n./N).*k;end;
```

```
>>function y=p6(n,N,D,k);
y=((N-n)./N).*D.*(n./N).*k;end;

>>function y = p(n,N,D,k);
y=p1(n,N,D,k)+
p2(n,N,D,k)+
p3(n,N,D,k)+
p4(n,N,D,k)+
p5(n,N,D,k)+
p6(n,N,D,k);
end;
```

B.3 Conclusion

A program listing is presented that provides a Matlab function for the model presented in Chapter 3.

Appendix C

Overview Pyramid Outputs

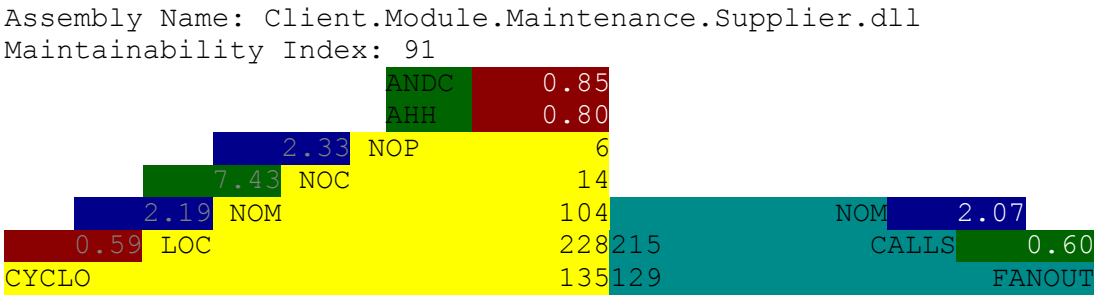
C.1 Introduction

The overview pyramids have been produced using a C# program as previously discussed in chapter 6.

This appendix provides a listing of output Overview Pyramids

C.2 Listing of overview Pyramids

The listing of Overview Pyramids follows:



```
Assembly Name: Client.Module.Maintenance.Consignment.dll
Maintainability Index: 90
                          ANDC      0.77
                          AHH       0.70
                 2.33 NOP             6
            8.43 NOC                 14
       2.42 NOM                     118                NOM      2.70
 0.58 LOC                        285319                CALLS    0.50
CYCLO                            165160                      FANOUT
-----------------------------------------------------

Assembly Name: Client.Module.Maintenance.VehicleConfiguration.dll
Maintainability Index: 91
                          ANDC      0.79
                          AHH       0.64
                 4.00 NOP             7
            8.25 NOC                 28
       2.53 NOM                     231                NOM      2.46
 0.57 LOC                        584569                CALLS    0.34
CYCLO                            334194                      FANOUT
-----------------------------------------------------

Assembly Name: Client.Module.Maintenance.OrderGroup.dll
Maintainability Index: 88
                          ANDC      0.75
                          AHH       0.67
                 2.83 NOP             6
            7.53 NOC                 17
       2.61 NOM                     128                NOM      2.91
 0.57 LOC                        334373                CALLS    0.45
CYCLO                            191167                      FANOUT
-----------------------------------------------------

Assembly Name: Client.Module.Reports.dll
Maintainability Index: 88
                          ANDC      0.89
                          AHH       0.27
                 3.81 NOP            32
            3.35 NOC                122
       2.71 NOM                     409                NOM      4.03
 0.62 LOC                       11091648                CALLS    0.20
CYCLO                            692328                      FANOUT
-----------------------------------------------------

Assembly Name: Client.Module.VideoFeedSnapshot.dll
Maintainability Index: 83
                          ANDC      0.22
                          AHH       0.22
                 1.20 NOP             5
            3.33 NOC                  6
       3.65 NOM                      20                NOM      5.40
 0.45 LOC                         73108                CALLS    0.51
CYCLO                             3355                      FANOUT
-----------------------------------------------------
```

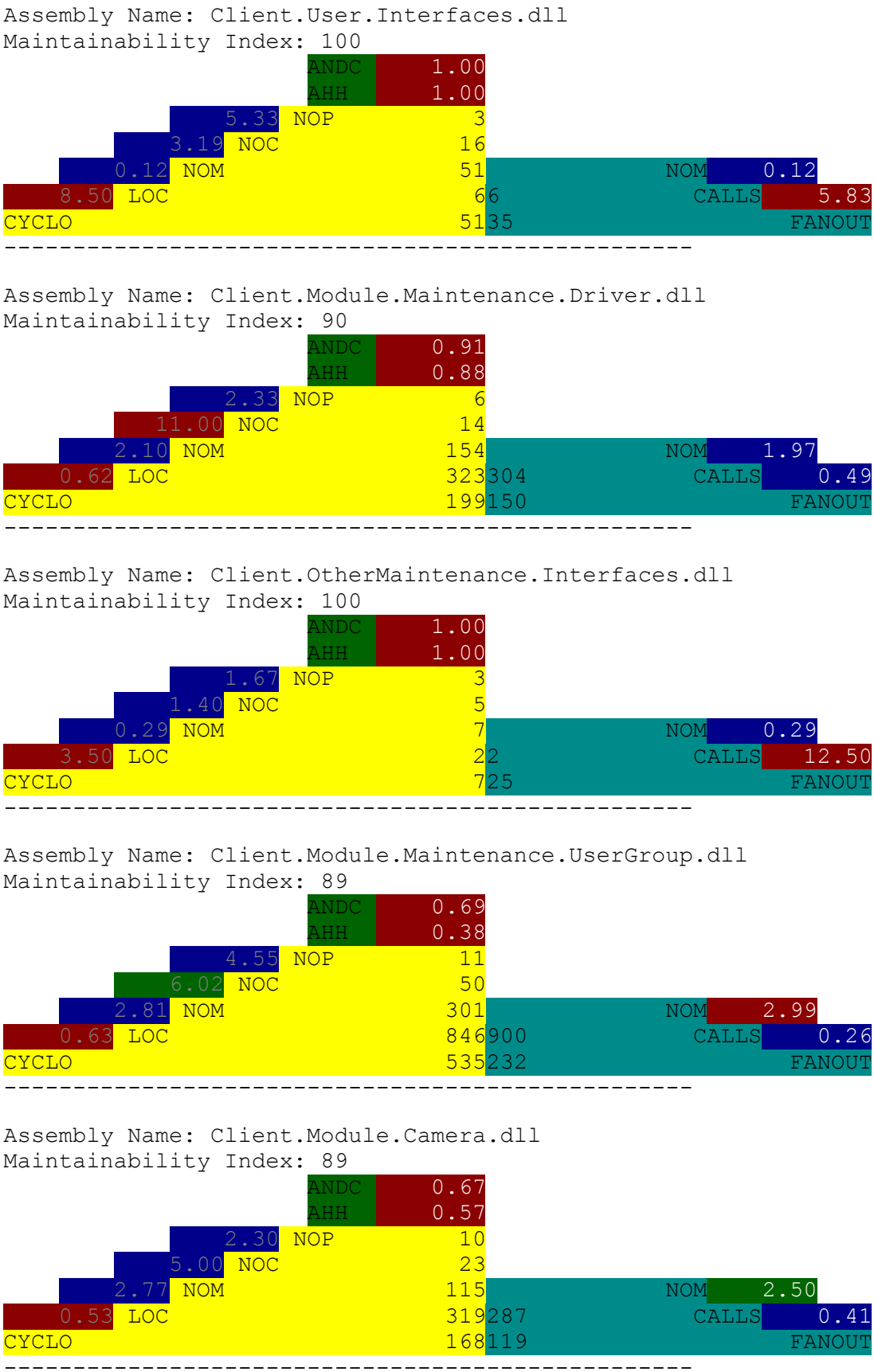
```
Assembly Name: Client.Site.Interfaces.dll
Maintainability Index: 100
                         ANDC      1.00
                         AHH       1.00
                 7.00    NOP          3
             1.81        NOC         21
         0.21            NOM         38                NOM      0.21
     4.75                LOC         88                  CALLS      4.63
CYCLO                    CYCL        37                        FANOUT
-------------------------------------------------
```

```
Assembly Name: Client.Module.Maintenance.Sku.dll
Maintainability Index: 89
                         ANDC      0.85
                         AHH       0.78
                 2.57    NOP          7
             7.61        NOC         18
         2.72            NOM        137                NOM      2.43
     0.53                LOC        372333               CALLS      0.46
CYCLO                    CYCL       197154                    FANOUT
-------------------------------------------------
```

```
Assembly Name: Client.Module.Menu.dll
Maintainability Index: 87
                         ANDC      0.69
                         AHH       0.67
                 3.00    NOP          5
             3.47        NOC         15
         3.62            NOM         52                NOM      4.29
     0.51                LOC        188223               CALLS      0.48
CYCLO                    CYCL        95108                     FANOUT
-------------------------------------------------
```

```
Assembly Name: Client.Module.Maintenance.Vehicle.dll
Maintainability Index: 89
                         ANDC      0.77
                         AHH       0.70
                 2.33    NOP          6
            15.86        NOC         14
         2.11            NOM        222                NOM      2.01
     0.66                LOC        468447               CALLS      0.41
CYCLO                    CYCL       307183                    FANOUT
-------------------------------------------------
```

```
Assembly Name: Client.Module.Maintenance.DeviceCentral.dll
Maintainability Index: 83
                         ANDC      0.78
                         AHH       0.72
                 5.00    NOP          8
             6.30        NOC         40
         3.17            NOM        252                NOM      4.02
     0.50                LOC        8001012              CALLS      0.24
CYCLO                    CYCL       399247                    FANOUT
-------------------------------------------------
```


Assembly Name: Client.User.Interfaces.dll
Maintainability Index: 100
ANDC 1.00
AHH 1.00
5.33 NOP 3
3.19 NOC 16
0.12 NOM 51
NOM 0.12
8.50 LOC 66
CALLS 5.83
CYCLO 5135
FANOUT
Assembly Name: Client.Module.Maintenance.Driver.dll
Maintainability Index: 90
ANDC 0.91
AHH 0.88
2.33 NOP 6
11.00 NOC 14
2.10 NOM 154
NOM 1.97
0.62 LOC 323304
CALLS 0.49
CYCLO 199150
FANOUT
Assembly Name: Client.OtherMaintenance.Interfaces.dll
Maintainability Index: 100
ANDC 1.00
AHH 1.00
1.67 NOP 3
1.40 NOC 5
0.29 NOM 7
NOM 0.29
3.50 LOC 22
CALLS 12.50
CYCLO 725
FANOUT
Assembly Name: Client.Module.Maintenance.UserGroup.dll
Maintainability Index: 89
ANDC 0.69
AHH 0.38
4.55 NOP 11
6.02 NOC 50
2.81 NOM 301
NOM 2.99
0.63 LOC 846900
CALLS 0.26
CYCLO 535232
FANOUT
Assembly Name: Client.Module.Camera.dll
Maintainability Index: 89
ANDC 0.67
AHH 0.57
2.30 NOP 10
5.00 NOC 23
2.77 NOM 115
NOM 2.50
0.53 LOC 319287
CALLS 0.41
CYCLO 168119
FANOUT

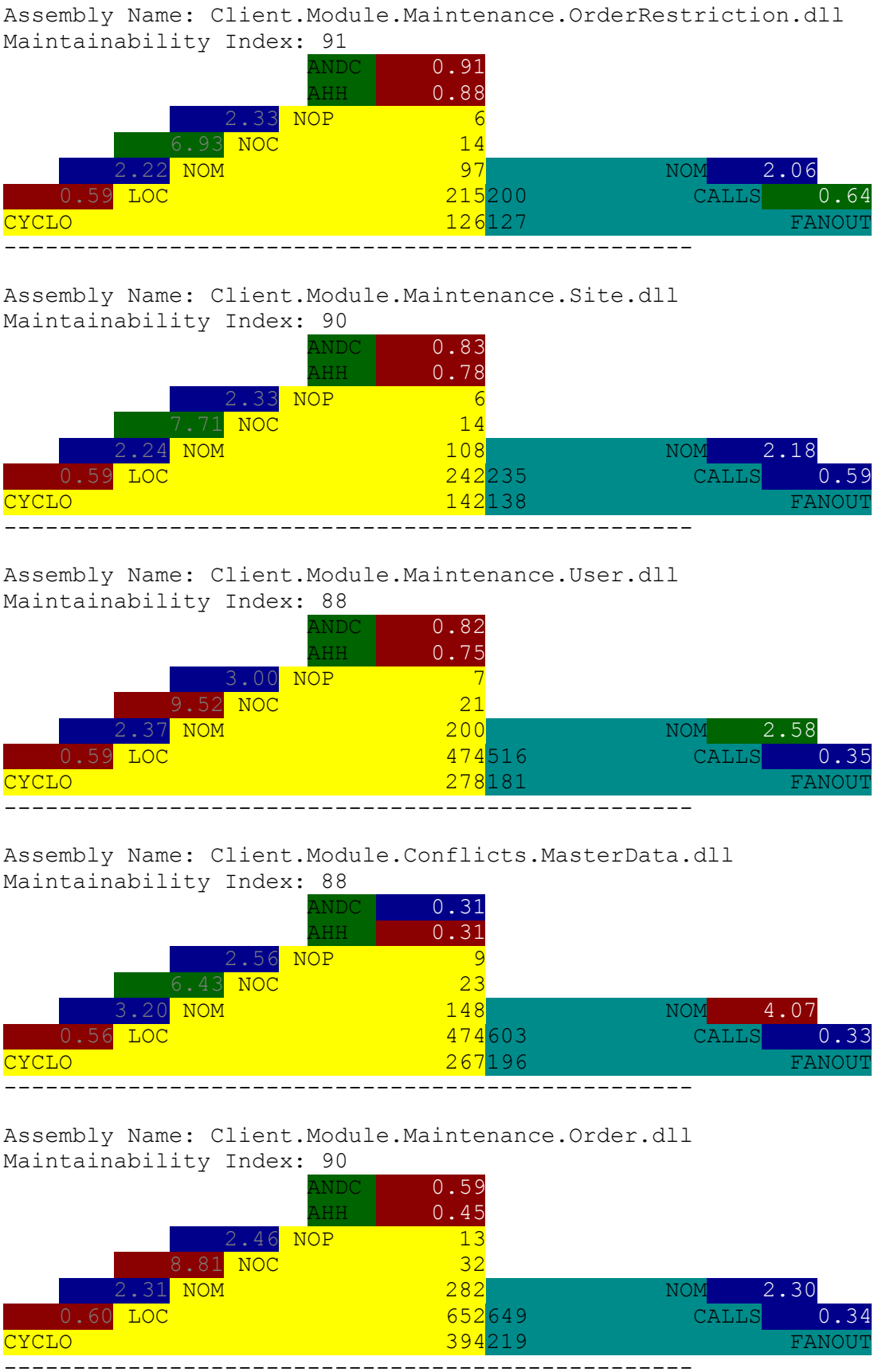
```
Assembly Name: Client.Module.Maintenance.LegalWeightLimit.dll
Maintainability Index: 91
                         ANDC      0.91
                         AHH       0.88
                 2.33    NOP          6
             8.07        NOC         14
         2.14            NOM        113                NOM      2.05
     0.60                LOC     242232                  CALLS       0.59
CYCLO                        144138                             FANOUT
--------------------------------------------------
```

```
Assembly Name: Client.Erp.Interfaces.dll
Maintainability Index: 100
                         ANDC      1.00
                         AHH       1.00
                11.25    NOP          4
             2.16        NOC         45
         0.18            NOM         97                NOM      0.18
     5.71                LOC       1717                  CALLS       3.18
CYCLO                          9754                             FANOUT
--------------------------------------------------
```

```
Assembly Name: Client.Module.Maintenance.Category.dll
Maintainability Index: 90
                         ANDC      0.83
                         AHH       0.78
                 2.33    NOP          6
             8.14        NOC         14
         2.18            NOM        114                NOM      2.05
     0.60                LOC     249234                  CALLS       0.58
CYCLO                        150136                             FANOUT
--------------------------------------------------
```

```
Assembly Name: Client.Module.Maintenance.Haulier.dll
Maintainability Index: 91
                         ANDC      0.91
                         AHH       0.88
                 2.33    NOP          6
             7.64        NOC         14
         2.14            NOM        107                NOM      2.02
     0.61                LOC     229216                  CALLS       0.63
CYCLO                        140135                             FANOUT
--------------------------------------------------
```

```
Assembly Name: Client.Module.Maintenance.OrderRestrictionGroup.dll
Maintainability Index: 91
                         ANDC      0.83
                         AHH       0.78
                 2.33    NOP          6
             6.86        NOC         14
         2.54            NOM         96                NOM      2.59
     0.57                LOC     244249                  CALLS       0.55
CYCLO                        138138                             FANOUT
--------------------------------------------------
```


Assembly Name: Client.Module.Maintenance.OrderRestriction.dll
Maintainability Index: 91
ANDC 0.91
AHH 0.88
2.33 NOP 6
6.93 NOC 14
2.22 NOM 97
NOM 2.06
0.59 LOC 215200
CALLS 0.64
CYCLO 126127
FANOUT
Assembly Name: Client.Module.Maintenance.Site.dll
Maintainability Index: 90
ANDC 0.83
AHH 0.78
2.33 NOP 6
7.71 NOC 14
2.24 NOM 108
NOM 2.18
0.59 LOC 242235
CALLS 0.59
CYCLO 142138
FANOUT
Assembly Name: Client.Module.Maintenance.User.dll
Maintainability Index: 88
ANDC 0.82
AHH 0.75
3.00 NOP 7
9.52 NOC 21
2.37 NOM 200
NOM 2.58
0.59 LOC 474516
CALLS 0.35
CYCLO 278181
FANOUT
Assembly Name: Client.Module.Conflicts.MasterData.dll
Maintainability Index: 88
ANDC 0.31
AHH 0.31
2.56 NOP 9
6.43 NOC 23
3.20 NOM 148
NOM 4.07
0.56 LOC 474603
CALLS 0.33
CYCLO 267196
FANOUT
Assembly Name: Client.Module.Maintenance.Order.dll
Maintainability Index: 90
ANDC 0.59
AHH 0.45
2.46 NOP 13
8.81 NOC 32
2.31 NOM 282
NOM 2.30
0.60 LOC 652649
CALLS 0.34
CYCLO 394219
FANOUT

```
Assembly Name: Client.Module.AboutScreen.dll
Maintainability Index: 93
                         ANDC      1.00
                         AHH      1.00
                 1.67 NOP            3
              3.80 NOC               5
         1.84 NOM                   19                NOM      1.42
    0.69 LOC                      3527                  CALLS    1.19
CYCLO                             2432                        FANOUT
-------------------------------------------------
```

```
Assembly Name: Client.Module.Maintenance.Wagon.dll
Maintainability Index: 92
                         ANDC      0.91
                         AHH      0.88
                 2.33 NOP            6
              7.79 NOC               14
         2.11 NOM                   109               NOM      1.86
    0.61 LOC                      230203                CALLS    0.64
CYCLO                             141129                      FANOUT
-------------------------------------------------
```

```
Assembly Name: Client.Module.Maintenance.Customer.dll
Maintainability Index: 91
                         ANDC      0.85
                         AHH      0.80
                 2.33 NOP            6
              7.43 NOC               14
         2.19 NOM                   104               NOM      2.07
    0.59 LOC                      228215                CALLS    0.60
CYCLO                             135129                      FANOUT
-------------------------------------------------
```

```
Assembly Name: Client.Module.Maintenance.Location.dll
Maintainability Index: 91
                         ANDC      0.91
                         AHH      0.88
                 2.33 NOP            6
              6.50 NOC               14
         2.31 NOM                   91                NOM      2.19
    0.57 LOC                      210199                CALLS    0.64
CYCLO                             120128                      FANOUT
-------------------------------------------------
```

```
Assembly Name: Client.Transport.Interfaces.dll
Maintainability Index: 100
                         ANDC      1.00
                         AHH      1.00
                13.33 NOP            3
              2.38 NOC               40
         0.17 NOM                   95                NOM      0.17
    5.94 LOC                      1616                  CALLS    3.19
CYCLO                             9551                        FANOUT
-------------------------------------------------
```

```
Assembly Name: Client.Module.SplashScreen.dll
Maintainability Index: 86
                        ANDC      0.40
                        AHH       0.40
                1.33 NOP             3
            6.00 NOC                 4
        2.29 NOM                    24                NOM      2.83
    0.65 LOC                      5568                  CALLS    0.50
CYCLO                             3634                        FANOUT
-------------------------------------------------
```

```
Assembly Name: Client.Module.Security.dll
Maintainability Index: 87
                        ANDC      0.61
                        AHH       0.50
                4.18 NOP            11
            5.43 NOC                46
        2.82 NOM                   250                NOM      3.29
    0.58 LOC                     70682                  CALLS    0.27
CYCLO                            40721                        FANOUT
-------------------------------------------------
```

```
Assembly Name: Client.Module.Maintenance.Container.dll
Maintainability Index: 91
                        ANDC      0.91
                        AHH       0.88
                2.33 NOP             6
            6.93 NOC                14
        2.27 NOM                    97                NOM      2.15
    0.57 LOC                     22020                  CALLS    0.63
CYCLO                            12613                        FANOUT
-------------------------------------------------
```

```
Assembly Name: Client.Module.AdminFunctions.dll
Maintainability Index: 91
                        ANDC      0.72
                        AHH       0.40
                4.90 NOP            10
            5.73 NOC                49
        2.45 NOM                   281                NOM      2.46
    0.61 LOC                     68969                  CALLS    0.29
CYCLO                            42220                        FANOUT
-------------------------------------------------
```

```
Assembly Name: Client.Module.DeviceTesting.dll
Maintainability Index: 90
                        ANDC      0.81
                        AHH       0.67
                4.83 NOP             6
            6.14 NOC                29
        2.41 NOM                   178                NOM      2.33
    0.56 LOC                     42941                  CALLS    0.38
CYCLO                            24115                        FANOUT
-------------------------------------------------
```

C.3 Conclusion

A listing of Overview Pyramids has been presented.