Free University of Bolzano-Bozen

Faculty of Computer Science

2009/2010

Applying the Scientific Method in the Definition and Analysis of a new Architectural Style

Supervisor:

Prof. Barbara Russo

Student:

Diego Marmsoler

Contents

1 Introduction

"Science is a method of investigating nature – a way of knowing about nature – that discovers reliable knowledge about it" [**1**]. This definition of science is the motivation behind this work where the author tries to apply the well known scientific method to define and further investigate the architectural style guiding the development of the Microsoft Dynamics AX product line.

This introduction follows the deficiencies model proposed by Creswell [**2**] as a template for writing a solid introduction to a research study. Hence it begins by describing the research problem and continues by pointing to studies that have addressed this problem. It concludes by recognizing deficiencies in those studies, thus emphasizing the importance of this study by pointing out how it addresses these deficiencies.

Taylor et al. [**3**] recognize "refined experience" in form of design patterns [**4**], architectural styles and patterns [**5**] and domain-specific software architectures [**6**] as the major tool for a software architect in the design of a new software system. However, in order to be useful at all, such "refined experience" must comprise reliable knowledge in form of a formal definition of the style, hypotheses about the elicited properties of the style and empirical evidence to support these hypotheses.

In Literature, the notion of architectural styles was coined in building architectures already 25 B.C. in Vitruvius M. Pollio's famous treatise "de Architectura" [**7**] before it was brought forward to the software engineering discipline by Dewayne Perry and Alexander Wolf in 1992 [**8**]. A first collection of architectural styles follows in Mary Shaw and David Garland famous paper on architectural styles and patterns [**5**]. Recently the concept was again refined in the two seminal works on software architecture by Bass et al. [**9**] and Taylor et al. [**3**]. Also the concept of software adaptability was inspired by the physical discipline [**10**] and was brought to the software engineering discipline by Taylor et al. [**3**]. Furthermore, Taylor et al. [**3**] in their work investigate the relationship of software architecture and adaptability and propose a set of styles promoting adaptability.

However the approaches used to investigate architectural styles in the above studies lacks rigor and reliability. The styles are defined informally and its properties derived intuitively in the absence of any empirical evidence. Furthermore none of the above works identifies an architectural style similar to the one presented later in this text. Hence, the following text begins with an extensive literature review on architectures, architectural styles and adaptability in section 3. Later the text presents the *Architectural Style Analysis Method* (ASAM) in section 4.1 derived by adapting the scientific method [**1**] to the definition and analysis of software architectural styles. ASAM is then adapted for the analysis of a new architectural style in sections 4.2.2 by first presenting a tool to collect raw data about the style in action, then presenting metrics derived by applying the goal-question-metrics framework [**11**], and finally presenting another tool to obtain the measures of the metrics from the collected raw data. Furthermore, ASAM is applied to define and investigate a new architectural style defined as *Partial Refinement* architectural style: it begins by formally defining the style in section 5 using the ALFA [**12**] framework and investigating in detail the identified *Selector* connector as a crucial component of the style. In section 6 an empirical observation on 5 Microsoft Dynamics AX [**13**] implementations is presented and a statistical analysis of the data obtained follows in section 6.4. Finally the text inductively derives a set of 4 related hypotheses from the observation to depict some of the elicited properties of the style.

Thus, the intent of this text is twofold: first the text aims to present ASAM, a reliable method which can be used to acquire reliable knowledge about an architectural style and hence significantly improve our knowledge about architectural styles and their elicited properties; second the text identifies the new Partial Refinement architectural style and applies ASAM to define it formally and reliably derive hypotheses on the elicited properties of the style. Therefore the text would be of interest above all for researchers searching for a reliable method to define and analyze new architectural styles, but likewise the text is also considered worthwhile for software architects which can use the "refined experience" in form of the new architectural style in conjunction with the inductively derived hypothesis on elicited properties, for the design of new software systems.

1.1 Research Questions

From the purpose statements in the above introduction, following research questions emerge:

RQ1: How is it possible to acquire reliable knowledge about an architectural style in order to reliably predict its elicited properties?

RQ2: How can the architectural style observed in Microsoft Dynamics AX systems be defined formally and what are the properties elicited by the style?

2 Definition of Terms

Definition: A software systems **architecture** is the set of principal design decisions made about the system [**3**].

Definition: An **architectural style** is a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system [**3**].

Definition: An **architectural pattern** is a named collection of architectural design decisions that are applicable to a recurring design problem, parameterized to account for different software development contexts in which that problem appears" [**3**].

Definition: A **design pattern** names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design [**4**].

Definition: A **domain-specific software architecture** comprises: a) a reference architecture, which describes a general computational framework for a significant domain of applications, b) a component library, which contains reusable chunks of domain expertise, and c) an application configuration method for selecting and configuring components within the architecture to meet particular application requirements [**6**].

Definition: A **non-functional requirement** is a requirement that is not directly concerned with the specific functions delivered by the system, but may relate to emergent system properties [**14**].

Definition: **Adaptability** is a Software system's ability to satisfy new requirements and adjust to new operating conditions during its lifetime" [**3**].

Definition: A **product line** (engineering) comprises a set of products which are tied together by similarities in how they are designed or constructed [**3**].

Definition: The **scientific method** is a method of discovering reliable knowledge about nature based on empirical evidence (empiricism), practicing logical reasoning (rationalism), and possessing a skeptical attitude (skepticism) about presumed knowledge that leads to self-questioning, holding tentative conclusions, and being undogmatic (willingness to change one's beliefs) [**1**].

Definition: A **scientific hypothesis** is an informed, testable, and predictive solution to a scientific problem that explains a natural phenomenon, process, or event [**1**].

Definition: A **corroborated hypothesis** is one that has passed its tests, i.e., one whose predictions have been verified [**1**].

Definition: A **scientific fact** is a highly corroborated hypothesis that has been so repeatedly tested and for which so much reliable evidence exists, that it would be perverse or irrational to deny it [**1**].

Definition: A **scientific theory** is a unifying and self-consistent explanation of fundamental natural processes or phenomena that is totally constructed of corroborated hypotheses [**1**].

3 Literature Review

According to Yair and Timothy [**15**], an effective literature review is characterized by "helping the researcher to understand the existing body of knowledge", thus "providing a solid theoretical foundation for the proposed study" and "substantiating the presence of the research problem". Further it should "justifying the proposed study as one that contributes something new to the Body of Knowledge" and "framing the valid research methodologies, approaches, goals, and research questions for the proposed study". In order to meet these goals, they proposes a framework based on a "systematic data processing approach comprised of three major stages: 1) inputs (literature gathering and screening), 2) processing (following Bloom's Taxonomy [**16**]), and 3) outputs (writing the literature review)." The following review was conducted using this framework and is motivated in fulfilling the ideas mentioned above. In doing so it follows the structure proposed in [**2**] where Creswell developed a model to structure a literature review based on five main components:

- An introduction that tells the organization of the literature review section.
- A topic to address the independent variable(s) proposed by the study.
- A topic to address the dependent variable(s) proposed by the study.
- A third topic to address studies on the relationship between the independent variable(s) and the dependent variable(s).
- A summary that highlight the key research studies relevant to the proposed study, their general findings that relate to the proposed study, and support for the need of additional research on the proposed topic.

3.1 Introduction

In order to meet the above structure, the following sections first touch upon the independent variable of this study, the general topic of software architecture and architectural styles respectively. Thereby it starts in section 3.2 presenting the general foundations needed to understand the discipline, before it goes deeper into some of the fundamental concepts of architectural models, components, connectors and their configuration. It recognizes refined experience in form of architectural styles as a fundamental tool for software architects and hence proceeds with a short review of such styles in section 3.2.3. The review continues by addressing quality attributes as the dependent variable of this study in section 3.3, thereby focusing on software adaptability as the key property elicited by the investigated style. The review continues by investigating current literature for studies on the relationship between architectural styles and software adaptability in section 3.4. Finally the section closes with a brief summary of the major findings of the review in section 3.5.

3.2 Software Architecture

Many of the concepts that today build the foundation for the study of software architecture can be found already a quarter-century ago in the seminal works done by Edgar Dijkstra and David Parnas. Any literature review of software architecture would by no means be complete without reviewing also the works of these software engineering pioneers.

Dijkstra's contribution to the scientific community by its seminal work on structured programming [**17**] [**18**] [**19**], and abstraction and refinement [**18**] [**19**] [**20**] directly influenced what today became software architecture. In his lifelong war against the unstructured organization of program code [**17**] and his promulgation towards a more structured organization [**18**] [**19**] of elements is what is core to software architecture today. Furthermore its advice to employ abstraction in designing Software [**18**] [**20**] and the concept of separation of concerns [**21**] is still one of the major tools for software architects in designing their systems. In the early 1970, Parnas deepen this observations with its seminal work on information hiding [**22**],

hierarchical structures [**23**], Program Families [**24**] [**25**], the relation between architecture and quality attributes such as reliability [**24**] and architecture as key concept in software adaptation [**25**]. Parnas dictum to "design for change" [**25**] is inherent in almost any successfully designed architecture today. In [**22**] Parnas recognizes the advantages of structuring a system in such a way to hide fragile design decision and therefore decouple the system in order to make it easier to maintain. In his work on Program Families [**24**] [**25**] Parnas recognizes the advantages inherent in viewing a system as a member of a family and designing it accordingly. Further Parnas recognizes the advantage of hierarchical structures and their ubiquity in Software [**23**]. Albeit some of the concepts mentioned above, today may seem trivial and inconspicous, they were key in laying the foundation in what today is called software architecture and it is important to note that we are "standing on the shoulders of [this] giants" [**26**] in current work on software architecture. Thus, one will find references to these early concepts everywhere in the following text.

However, the first works treating software architecture explicitly as a separate discipline and hence forming the foundation to this new field of study can be found in Perry and Wolf's [**8**] seminal 1992 work on software architecture in general and Shaw and Garlan's [**5**] following work on architectural styles in 1996. In their seminal work Perry and Wolf try to develop an intuition about software architecture by looking at several architectural disciplines traditionally been considered sources of ideas for software architecture so far. They began discussing hardware and network architectures and further expanded their inquiry on building architecture as the "classical" architectural discipline. They conclude with the belief that "we find in building architecture some fundamental insights about software architecture: multiple views are needed to emphasize and to understand different aspects of the architecture; styles are a cogent and important form of codification that can be used both descriptively and prescriptively; and, engineering principles and material properties are of fundamental importance in the development and support of a particular architecture and architectural style". However, the analogy to building architectures must be taken with a pinch of salt and so Bass et al. [**9**] correctly argue that a building architect is faced with quality attributes considerable different than a software architect has and therefore they suggest to not take the analogies too far as they break down fairly quickly. This claim is also supported by Tailor et al. in [**3**] where the authors recognize several shortcomings of the analogy to building architectures and depict several of its limitations and problems.

3.2.1 Definition

Even though software architecture is recognized as a fundamental concept in the successful development of today's complex and critical software systems there is still no common understanding of the precise meaning of the term [**27**] and therefore this section walks through the huge universe of definitions and its implications for an architecture.

Let the journey through this universe of definitions begin with a first stop at the prime fathers view of software architecture. Inspired by the art of building architecture, Perry and Wolf [**8**] propose a model of software architecture where elements capture the system's building blocks, their purpose and the services it provides:

$$Architecture = \{elements, form, rationale\}$$

In this model, architecture comprises three types of building blocks: *Processing elements* performing calculations and acting upon *Data elements* which contain the information used and transformed. Further they identify *Connecting elements* as a key concept in their model, acting as the glue that holds all the elements together. They play a fundamental rule in distinguishing between different architectures and affect the characteristics of a particular architecture or architectural style. The form constraints the choice of architectural elements by capturing minimum desired constraints on the properties for those elements. Furthermore the form constrains the relationships between those elements, how they interact and how they are organized. The form also captures the importance of such a property or relationship to enable distinction from aesthetics and engineering and allows further the selection among alternative properties or relationships. Furthermore, the third dimension of their definition captures the system designers rational for specific choices,

hidden information in the original architect's minds. This component allows one to answer the why question about an architecture. Why this architectural style was chosen? Why this elements and this form was chosen? The rational explicates the satisfaction of the system constraints determined by its non-functional aspects.

Let us consider now Bass et al. [**9**] definition of the term software architecture. They define software architecture as follows: "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them". To understand better this definition it's inevitable to look further into the implications inherent with this definition: First of all "architecture *defines* software elements" by focusing on the relationship between them and suppressing details that do not affect how they use, are used by, relate to or interact with other elements. Therefore architecture abstracts from internal implementation details which are not architectural. This is exactly what can also be found in Parnas [**22**] early work on hiding design decisions which are likely to change. Furthermore the definition sheds light upon the fact that "systems can and do comprise more than one structure" which implies that there is no single structure for a system, but rather there are different structures for a system that all convey different architectural information for the same architecture. This implication is indeed coherent with Perry and Wolf's [**8**] early perception to use multiple views to describe an architecture. Third the definition makes clear that explicitly or implicit "every computing system with software has a software architecture". This by no means presumes that every system was developed with a specific architecture in mind or even that anyone knows the systems architecture. Rather each system can be shown to comprise elements and the relationship among them. This totally agrees with Parnas [**23**] early insight that every system has a [hierarchical] structure and that in order for this statement to carry any information at all it is important to specify the way the system is divided into parts and the relationship between these parts. Another important insight from the above definition is that "the behavior of each element is part of the architecture" insofar as that behavior bear on externally visible properties and hence influences other architectural elements and guides the interaction and contributes to the overall system acceptability. The definition further does not care of whether "the architecture for a system is a good one or a bad one", allowing architectures to meet or not meet its functional or non-functional requirements.

Another fundamental definition of software architecture comes from Taylor et al. [**3**]: "A software systems architecture is the set of principal design decisions made about the system". Let us now take a closer look on this statement and consider the implications of this definition of software architecture. Note the central role of the notion of *design decision* in this definition of software architecture. The authors define design decisions as decisions „encompass[ing] every aspect of the system under development" and relate them to system structure, functional behavior, interaction, non-functional properties and implementation issues. Furthermore one may recognize the adjective *principal* in the definition. This is another central term in the definition. It implies that there are some design decisions which are not considered architectural. It distinguishes architectural vs. non architectural design decisions. The definition of principal and therefore the consideration of a design decision as architectural or not however are largely subjective and depend on the context in which the system is developed. This differentiation of architectural and non architectural decisions is implicit also in the definition provided by Bass et al. [**9**]. Another implication from the above definition is the existence of a set of such principal design decisions at any time during the development and the evolution of the architecture. But this is a rather dynamic process and this means that architecture must have a temporal component. This is an addition to Perry and Wolf's [**8**] definition which does not explicitly capture evolution in their definition. Furthermore the authors distinguish between a prescriptive and a descriptive architecture, which implies that the architecture consists also of the rational, which is perfectly consistent with the definition given by Perry and Wolf [**8**].

Another useful definition of system architecture is the one tightened in the ANSI/IEEE Standard 1471-2000: "Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and principles governing its design and evolution" [**28**]. However this definition does not explicitly refer to software architecture and rather depicts its similarity to other kinds of architectures.

3.2.2 Architectural Models

Although each architecture is different, Taylor et al. [**3**] elaborate on Perry and Wolf's [**8**] work and identify some basic concepts that are common to almost all architectural descriptions: *components* as the "architectural building blocks that encapsulate a subset of the system's functionality and/or data", c*onnectors* as another first class "architectural building blocks that effect and regulate interactions among components", *interfaces* as "the points at which components and connectors interact with the outside world", *configurations* comprising "a set of specific associations between the components and connectors of a software systems architecture" and *rationale,* containing "the information that explains why particular architectural decisions were made, and what purpose various elements serve". In the following section we will take a closer look on these principal elements of architecture.

3.2.2.1 Components

The notion of a modular element to encapsulate certain design decisions and hide them behind well defined public interfaces goes back to Parnas in 1972 [**22**]. He recognizes the benefits yielded if "every module [in a system] is characterized by its knowledge of a design decision which it hides from all others" and "[choose] its interface or definition […] to reveal as little as possible about its inner workings".

Another important insight on components is provided by Shaw et al. [**29**], where the authors define a component as a *locus of computation and state* in a system. In this definition one can recognize that while the first two kinds of elements identified by Perry and Wolf [**8**] became what nowadays would be called a software component, the last one flow into the different concept of software connector threaten explicitly in the next section.

Another widely cited definition on software components can be found in Szyperski's [**30**] seminal work on components where he defines them as a unit of composition with contractually specified interfaces and explicitly context dependencies only. According to the author, a software component can be deployed independently and is subject to composition by third parties.

On top of the above definitions, Taylor et al. [**3**] provide the following definition of a software component: A software component is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context. The key implication of the above definition is the observation that a component appears as a black box which is accessible from the outside only, and only via its public interface. Therefore it substantiates the fundamental software engineering principles of encapsulation, abstraction and modularity [**22**]. Another observation is the explicit treatment of the components execution context in form of the components required interface. This context is assumed to be provided by other components and comprises the availability of specific resources, such as data files or directories, required system software like operating systems, middleware and runtime environments, and finally the hardware configurations needed to execute the component. A last but crucial observation stated by Taylor et al. is the relationship of a component to a specific application or problem domain to which it belongs. This domain dependence is one of the major aspects a component differs from the below described connectors which are primarily domain independent. However, Taylor et al. identify also so called utility components which are mostly application and also domain independent and usually provide a large superset of any system's particular needs.

3.2.2.2 Connectors

The notion of a software connector as a first class architectural element stem from Perry and Wolf's [**8**] seminal work. According to the authors these connecting elements are "the glue that holds the different pieces of the architecture together". Furthermore they recognize that "these connecting elements play a fundamental part in distinguishing one architecture from another and may have an important effect on the characteristics of a particular architecture or architectural style". Despite the early advice of Perry and Wolf on the key role connectors plays in the discipline of software architectures, they were considered for long time as less

important than processing or data elements. While research focused mainly on component structure, interfaces and functionality, connecting elements only played a secondary role [**30**].

This Problem however was finally uncovered by Mehta et al. [**31**] who provides an extensive treatment of the concept ending up with a complete taxonomy of software connectors. According to the authors every connector is build around two elementary principles, those for managing the *flow of control* and those for managing the *flow of data.* Furthermore each connector follows the necessary condition to maintain one or more channels or ducts used to link different components to implement former elementary principles. Consider Figure 1 for a graphical representation of their connector classification framework.

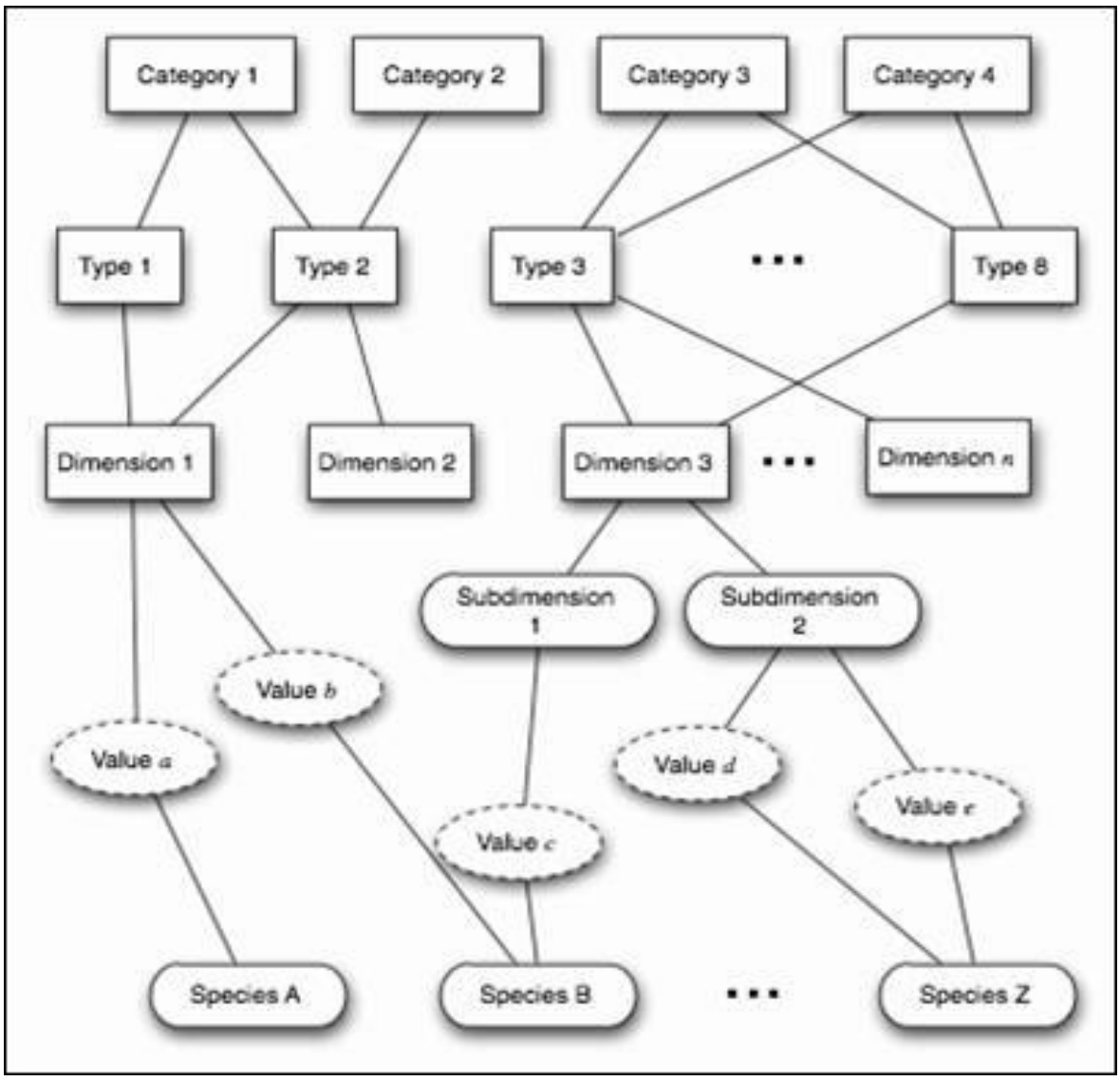

Figure 1: Structure of the connector classification framework [31].

In this framework they recognize four core types or roles of interaction services a connector can provide:

- *Communication* as the reification of the flow of data principle supporting the transfer of data between components.
- *Coordination* as the conception of the second elementary principle of flow of control.
- *Conversion* which enables adaptation of two normally different components to allow heterogeneous components to interact.
- *Facilitation* services which are used to mediate and streamline component interaction.

In order to realize their services, connectors may belong to one or more *connector types* with each type having some defined points of variation, so called *dimensions* and *values* to choose from in filling this dimensions. In choosing values for each dimension one can create a *connector species* which can either be considered as *simple* or *composite* (higher-order) species, depending on whether they use dimensions of one or multiple connector types. The authors also identify several types a connector can belong to with the different dimensions enabling the variability for a connector and possible values to instantiate a specific species:

- *Procedure call connectors* provide communication and coordination services varying in dimensions like parameters, entry point, invocation, synchronicity, cardinality and accessibility. A specific procedure call connector will now be created by choosing among several values to fill these dimensions. An example of a simple species of this connector type might be the typical programming level procedure call, while a higher order species would emerge by applying some facilitating services to create a RPC connector.
- *Event connectors* are similar to procedure call connector in that they provide communication and coordination services in a different way. Connectors of this type can vary in dimensions such as cardinality, delivery, priority synchronicity, notification causality and mode. Examples of such type of connectors are found in windowing systems or distributed applications that require asynchronous communication.
- Another identified type is the *data access connector* type which provides communication and conversion services. Variation is implemented in dimensions like locality, access, availability, accessibility, life cycle and cardinality. Such a connector would allow to access data from a data store component and translate the data in type of differences in the required and provided data format. An example of such a connector type would be a DBMS.
- A *linkage connector* type is the simplest among all the types but simultaneous it is the most important one enabling the establishment of ducts to form the foundation upon which other connectors can perform their actions. They offer facilitation services and vary in types of reference, granularity, cardinality and binding.
- *Stream connector* types are mere communication based and vary in delivery bounds, buffering, throughput, state, identity, locality, synchronicity, format and cardinality. They allow for transferring a large amount of data between components and may be combined with other connectors such as the data access connector to perform database and file storage access. Examples of species from such a type would be the UNIX pipes or the TCP/UDP communication sockets.
- *Arbitrator connectors* play a key role in managing different components which are not "intelligent" enough to coordinate their activities by themselves. Arbitrators therefore provide coordination and facilitation services and varies in dimensions such as fault handling, concurrency, transactions, security and scheduling.
- *Adaptor connectors* are used to connect components in heterogeneous environments where components have not been designed to interoperate. Therefore providing conversion services and varying in dimensions like Invocation conversion, packaging conversion, protocol conversion, and presentation conversion.
- Finally *distributor connector* types are used in distributed systems to identify and route interaction among distributed components. They never exist by themselves, but serve other types such as streams or procedure calls hence providing facilitation services. Such types vary in dimensions such as naming, delivery and routing. Famous examples of such connector types are several network services such as naming, routing, and switching services.

In their seminal work, Taylor et al. [**3**] pick up the above ideas and elaborate them further. They define a software connector as an architectural element tasked with effecting and regulating interactions among components. Furthermore they come up with a process of selecting an appropriate connector for a given system and provide a connector compatibility matrix to support conception of composite connector species.

3.2.2.3 Configuration

Another important concept in every architectural description is how specific components and connectors are composed to meet the systems objectives. Such a composition is also called a configuration and defined by Taylor et al. [**3**] as "a set of specific associations between the components and connectors of a software systems architecture". However the authors note that establishing an interaction path between two components is a necessary, but by no means a sufficient condition for communication. So called architectural mismatches may occur for example due to incompatible interfaces.

3.2.2.4 Modeling Techniques

In Taylor et al. [**3**] one can find a summary of a panoply of modeling techniques useful for the development of an architectural model, ranging from natural language and informal power-point modeling over more sophisticated languages like UML [**32**] or so called architecture description languages like Darwin [**33**] or Rapide [**34**]. Furthermore the Authors note some domain- and style specific ADL's like Koala [**35**], Weaves [**36**] and the architecture analysis and design language [**37**], Acme [**38**], The architecture description markup language [**39**] and xADL [**40**]. Finally Kruchten [**41**] suggests 5 different views for an architectural model to communicate it in a sufficient manner.

Metha and Medvidovic [**12**] however recognize some deficiencies in the traditional way of characterizing architectural styles in terms of constrains on processing and data components, connectors, and their configurations. Hence, they suggest a five-way characterization of styles using structure, interaction, behavior, topology and data. On top of this characterization they developed ALFA, the assembly language for architectures, as a framework "for understanding, and as a result composing" architectural styles from so called architectural primitives. These primitives are fine-grained, low-level abstractions, each of which focuses on a single aspect of an architectural style. Furthermore, the authors identify a set of such primitives consisting of eight nouns, capturing the form of architectural style elements, and nine verbs capturing the element's function:

- Data: Datum
- Structure: Particle, Output, Input, Twoway
- Interaction: Duct, Relay, Birelay, holds, loses
- Behaviour: create, send, receive, handle, reply
- Topology: connect, disconnect

3.2.3 Architectural Styles

Before going deeper into the concept of architectural styles, let's first take a look on the physical discipline to develop an intuition on how creating a good architecture. A fundamental insight to the design process as a whole can be found in Jones [**42**] work about industrial design. He suggests starting the process with the gathering of information before a set of alternative arrangements for the design as a whole can be derived on top of that information. After then one of these alternatives is selected for further elaboration to a point where the work can be split up for detailed design by many people working in parallel. This approach to design nowadays pervades the software engineering world and is to be found in many traditional software development processes like the waterfall or spiral model. However, a crucial step in this process is thy way such a set of alternative arrangements is derived. Taylor et al. [**3**] propose several strategies to address the problem of identifying this starting point and detect refined experience in form of architectural styles and patterns as the Grand tool to manage this problem. They further propose to "use the fundamental design tools of software engineering": abstraction [**18**] [**19**] [**20**] and modularity in form of simple machines and separation of concerns [**21**] [**22**] to "carve with those knives" to that design. Furthermore the authors advice to isolate creative inspiration to those parts where it is really needed.

The importance of style in architecture to constrain an architecture to yield specific properties out of the architecture was documented in physical discipline already in 25 BC, in Vitruvius [**7**] famous treatise on

architectures. Later on, Perry and Wolf [**8**] again bring the concept from the physical discipline to the software engineering world. They recognize that we have not yet arrived at the stage where we have a set of architectural styles with their accompanying standard design elements which can be used by the architect so that he can focus on those elements where customization is crucial. They define architectural style as follows: "If architecture is a formal arrangement of architectural elements, then architectural style is that which abstracts elements and formal aspects from various specific architectures". Thereby they spot that "an architectural style is less constrained and less complete than a specific architecture." They note that the boundaries between a style and a specific architecture are porous and "an architectural style [..] encapsulates important decisions about the architectural elements and emphasizes important constraints on the elements and their relationships" thereby focusing on those decisions suffering erosion and drift.

On top of the definition of Perry and Wolf, Taylor et al. [**3**] defines an architectural style as a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system". Furthermore the authors came up with a fundamental statement on the relationship between design patterns, architectural styles, architectural patterns and domain-specific architectures. Thereby they classify them according to domain knowledge and scope with porous boundaries between the concepts.

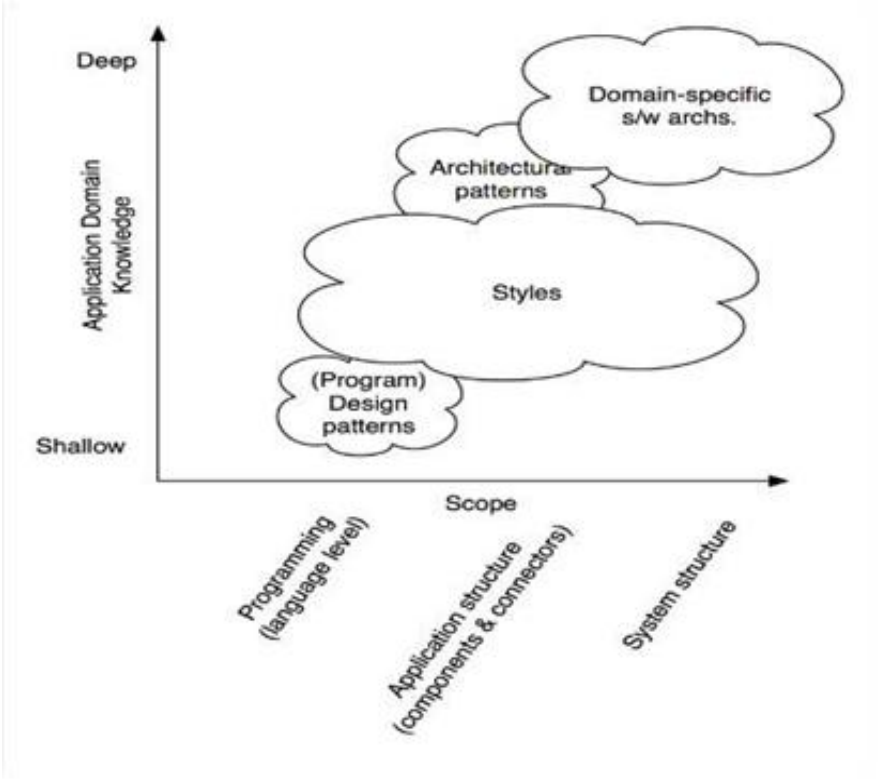

Figure 2: Domain knowledge versus scope in showing the relationship between styles and patterns [3].

In Figure 2 the horizontal axis represents the application scope of the lessons of experience where the vertical line represents the domain knowledge clutched within the experience. According to the diagram it appears that design patterns such as that in [**4**] are very specific applicable for a given programming language while they are applicable within a broad domain context. Further, the authors note, that architectural patterns are a named collection of architectural design decisions that are applicable to a recurring design problem, parameterized to account for different software development contexts in which that problem appears, such as the one in [**43**]. Domain specific architectures on the other hand are very high level in the sense of system

structure but very specific in the domain knowledge. The authors suggest to compose a DSSA of a reference architecture, a corresponding library of software components for that architecture and a method for choosing and configuring them.

On the other hand, Bass et al. [**9**] do not consider this explicit distinction from style to pattern and define them as a description of element and relation types together with a set of constraints on how they may be used. According to the authors an architectural pattern and an architectural style describe the same concept. A set of constraints on element types and their patterns of interaction. However, the authors agree in the assumption, that patterns exhibit known quality attributes.

Some of the first documented styles can be found in the work of Shaw and Garlan [**5**] in their 1996 book where they identify a asset of well known styles observed in industry: the pipes and filter style, data abstraction and object oriented organization, event based, implicit invocation, layered systems repositories, table driven interpreters, distributed processes, main program and subroutine, domain specific software architectures, state transition systems, process control systems. The authors further advice on how to combine several pure styles to gain a so called heterogeneous architecture.

Taylor et al. [**3**] elaborate on this work and first distinguish between s*imple* and *more complex styles*. Furthermore they propose an outline to classify and organize simple styles and provide seven classes of such styles:

- The t*raditional language-influenced* styles reflect simple low-level styles that naturally following from using some of the imperative languages. These kinds of styles would be positioned at the outermost position, with respect to the scope dimension, in the above style-cloud. Of course styles in this category can be combined with other, more general styles to obtain some intended quality properties. The authors mention further two examples for these style categories: main program and subroutines and object-oriented style.
- The l*ayered* styles on the other hand encompass all styles following the well known layer principle, wherein the architecture is separated into ordered layers and programs within one layer may obtain services from a layer below it. The authors mention two important styles belonging to this class: the virtual machine style and the client-server style.
- The d*ataflow* styles category comprises all such styles where data movement between independent processing elements is crucial. Such styles may be batch-sequential and pipe-and-filter.
- *Shared State* styles are those styles where the single components communicate through a global repository. The design attention within these styles is explicitly on these structured, shared repositories. One common example for this category would be the blackboard style or the rule-based/expert system with a knowledge base as the shared memory.
- In the *interpreter styles* category, architectures following these styles work on a current execution state and enable the dynamic, on-the-fly interpretation of commands to modifying such state. The identification of the next command may be affected by the result of executing the previous command. Members of this class would be the b*asic interpreter* style used in Microsoft Excel to evaluate Excel's formulas; or the m*obile code style,* in which the place to execute the commands may vary over time.
- The *implicit invocation* style groups styles with loosely coupled elements due to calls which are invoked indirectly and implicitly as a response to a notification or an event. Two familiar styles in this class are the publish-subscribe style and the event based style.
- Finally there is another important style which is grouped in the equally named class, the p*eer-to-peer* style, well known for its decentralized state and logic.

The authors further identified more complex styles, derived by leveraging a combination of some of the simple styles to yield powerful design tools capable of handling challenging design problems. They mention the *C2 (component and connector) style* as a combination of concepts from the layered and event-based architectures and the *distributed object style* as the result of a desire to extend the benefits of a core style to a distributed, heterogeneous world.

3.3 Software Quality Attributes

Software quality attributes are specified in form of so called non-functional software requirements which, according to Sommerville [**14**], are not directly concerned with the specific functions delivered by the system but may relate to emergent system properties such as reliability, response time, performance, and others. Furthermore Sommerville notes that they are often more critical than individual functional requirements and failing to meet such a non-functional requirement can mean that the whole system is unusable. Therefore he proposes to state them quantitatively, so that they can be objectively tested. An interesting approach to characterizing such quality requirements can be found in Bass et al. [**9**] where the authors identify several shortcomings with current taxonomies and definitions and propose so called quality attribute scenarios to characterize and define quality attributes in a meaningful way.

In current literature however, a lot of such quality attributes can be found and many of them have their own research and practitioner communities. A detailed treatment of all this attributes is outside the scope of this text. Hence the following sections concentrates on adaptability as a key property elicited by the style investigated in this study.

3.3.1 Software Adaptability

Again it is important to note that in the recent work on software architecture and adaptation we are plowing old ground. In his early work, David Parnas [**24**] recognizes pragmatically that "a set of Programs […] constitute a family whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members". Parnas [**25**] further recognizes four "obstacles commonly encountered in trying to extend or shrink systems":

- *Excessive information distribution* due to the lack of information hiding, where "too many programs were written assuming that a given feature is present or not".
- A *chain of data transforming components*, where the data format between different components change and makes it "hard to remove because the output of its predecessor is not compatible with the input requirements of its successor"
- *Components* that perform *more than one function* "because the functions seem too simple to separate" thereby manifesting itself in too large components comprising too much functionality.
- *Loops in the uses relation* as a side effect of excessive and unrestricted reuse through components leading to a system where "nothing works until everything works".

Parnas addresses these pitfalls in [**23**] [**25**] with the advice to *anticipate change* and *identify variability points* already in requirements elicitation, employ *information hiding* by isolating changeable parts behind fixed interfaces, apply the *virtual machine* concept which teaches us to "stop thinking of systems in terms of components that correspond to steps in processing" and instead employ abstraction at the level of operators and operand types and designing the *uses structure* by constraining the use of the uses relation for example in a hierarchical way. Parnas [**25**] also lays the foundation to what today became product line engineering with the groundbreaking assumption that development and maintenance costs will be significantly reduced if design for a system proceeds with the whole family in mind rather than just a sequence of individual systems [**24**], thus exploiting commonalities across systems.

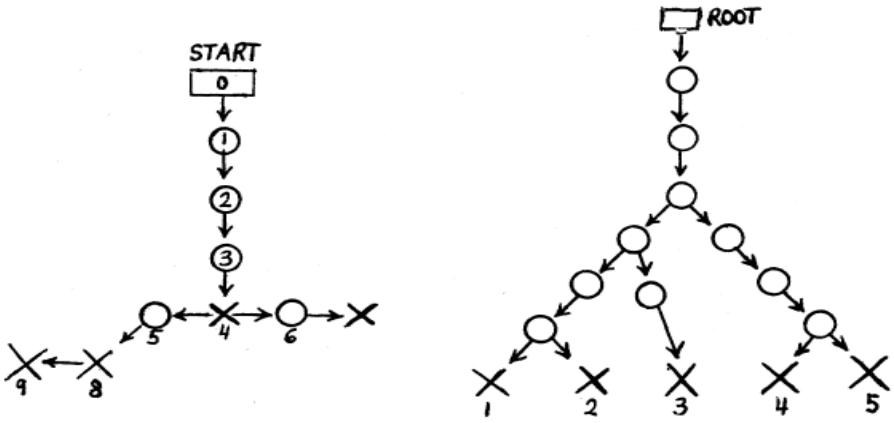

Figure 3: Representation of development by sequential completion and using "abstract decisions" [24].

In Figure 3 Parnas depicts the traditional development as sequential completion in contrast to the program family approach or "abstract decisions". In the above Figure, an X represents a working system, an O stands for an intermediate representation but not a working program, and an arc represents a design decision. Furthermore Parnas recognizes two development paradigms suitable for the design of program families, differing in the method in which they represent the partial designs: stepwise refinement [**18**], promulgating the use of abstract operators and operand types refined in subsequent steps; and module specification [**22**], leveraging interface specification to hide varying design decisions in different subfamilies.

Taylor et al [**3**] recently elaborated on the concept of software adaptation and defines software adaptability as "a software system's ability to satisfy new requirements and adjust to new operating conditions during its lifetime". Furthermore the authors draw from the physical building discipline in its effort to study this property. They recognize an interesting inside by considering change in building architectures. In his insightful work Steward Brand [**10**] describes *how* and *why* buildings change over time and categorizes the types of change that can be made to a building architecture in six so called shearing layers. *Site* represents the geographical settings providing the base for generations of ephemeral buildings. *Structure*, as the incarnation of the buildings, consists of the foundation and the load bearing elements. *Skin*, as the malleable surface of the building, regularly suffers from changes. *Services* as pervading net of wires and plumbing, s*pace plane* as the internal layout of the building in terms of walls, doors and floors and last but not least the transient order of *stuff* like pictures, phones or desks.

Although software in its very nature consists of far more malleable elements it can be equally categorized into different layers: the *architecture* as the software systems structure, or the *user interface* as its skin. However this classification will not become obvious by simply looking at a systems source code, but requires a deeper analysis. However if we could make a similar, explicit categorization of a software system according to the nature and cost of making a change within those dimensions, this would help us to understand the necessary techniques to effect a given change and allow us to dependably estimate the time and cost to perform changes.

Another lesson from Brands observations that layers change at different rates is the advice to limit coupling between elements belonging to different shearing layers. This lesson is extending Parnas early advice to design for change [**25**] with the admonition to constrain connections between elements according to the type of layer they belong to.

A last insight from the building discipline comes from the fact that "because of the different rates of change of its components, a building is always tearing itself apart". This is similar in software engineering, thus, one should avoid making changes to software systems cutting across layer boundaries. This problem results naturally if changes are based only upon local knowledge of source code without considering software systems architecture.

3.4 Architectural Styles promoting Adaptability

In recent works on software architectures, a software systems architecture is determined as the key factor in achieving quality attributes for the system [**9**] [**3**]. Hence the following chapter presents different architectural styles, identified by Taylor et al. [**3**] as appropriate for supporting adaptation. The authors therefore classify them into two categories: *Interface-focused* architectural solutions, which reflect David Parnas dictum to "design for change" [**25**], and *(strong) architecture based* approaches. According to the authors, the primary focus of former technique is only in adding functionality in the form of a new module and does not purport to support all types of change.

3.4.1 Application Programming Interfaces (APIs)

In this style, the core application exposes an interface in form of a set of functions, types, and variables. Developers may then add new modules which can use these interfaces to modify the existing application. The interrelationship of the new modules itself is thereby unconstrained by the API and the original application cannot make calls to the new modules.

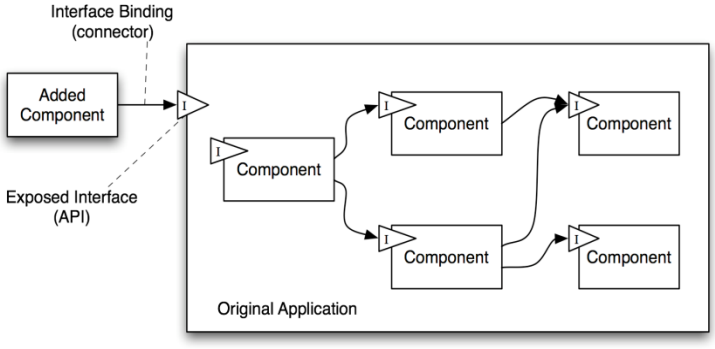

Figure 4: Using an API to extend an application with a new module [3].

Figure 4 depicts this situation graphically, where a new component has been added and can access the features of the original application only by those interface exposed by the API.

3.4.2 Plug-Ins

In this style, instead of the added components calling the original application, the application calls out to the added component through a predefined interface. In order for the original application to be aware of the existence of the plug-in, this must become registered with the application.

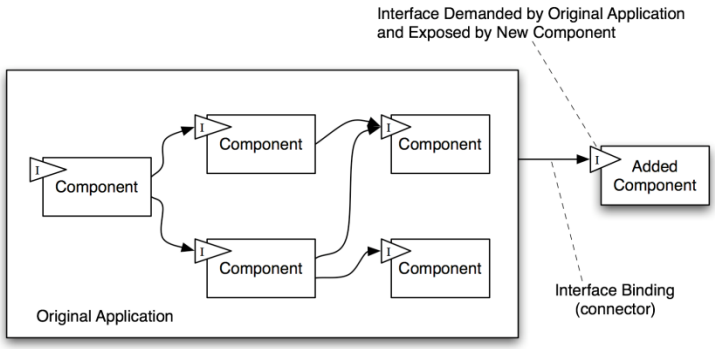

Figure 5: Using a plug-in interface to extend an application [3].

Figure 5 demonstrates the style in action, where the core application calls out to an added component and thereby adapting its behavior.

3.4.3 Component/Object Architectures

In this style, the original application is no longer viewed as a monolithic entity, but its internal structure is exposed, and hence amenable for change. New components can interact directly with any internal component and also replace it by exposing a compatible interface.

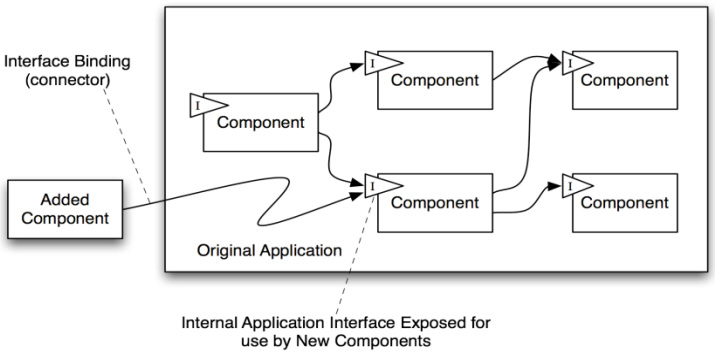

Figure 6: Application modification via the component/object architecture approach [3].

Figure 6 shows this situation where the added component directly calls on the interface exposed by an internal component.

3.4.4 Scripting Languages

This style in essence is a use of the interpreter style described earlier. The core application provides its own programming languages through which the applications behavior can be modified.

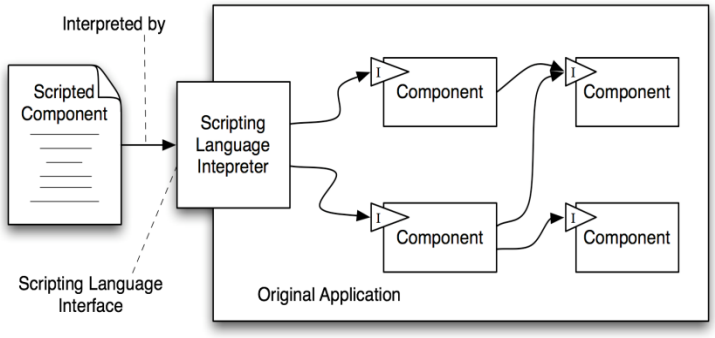

Figure 7: Application adaptation by means of a scripting language [3].

Figure 7 depicts the situation in which the commands of a component, implemented using the applications own programming language, will be interpreted by the core applications interpreter; thus, changing the applications behavior.

3.4.5 Event Interfaces

This technique is an application of the event-based style presented earlier. The core application provides two types of interfaces through which its behavior can be modified: *Incoming* event interface, where the new components can act on messages received; and *outgoing* event interface, where the new components can generate new messages.

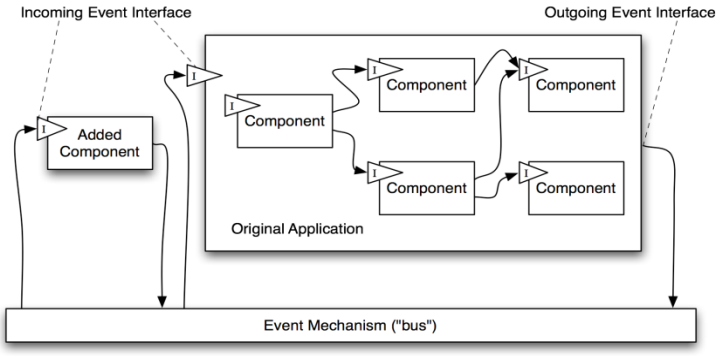

Figure 8: An event architecture used to support extension [3].

Figure 8 shows this situation graphically where a new component will be added to communicate with the application through an event mechanism connector.

3.4.6 (Strong) Architecture Based Approaches

For more complex kinds of adaptations than just adding new components, the authors suggest to use explicit architectural models, faithful to the implementation, and applying consistent use of architectural styles described earlier. They consider the connector as a crucial element in highly adaptive systems. They claim that to the extent that an architectural style enables easy attachment and detachment of components from one another, that style facilitates adaptation. Hence they point to the implicit invocation architectural style mentioned earlier using the event based connector as an example of an architectural style implementing adaptation through its connector.

3.5 Summary

The above literature review began by a general treatment of software architecture, investigating several definitions and their implications presented in current literature. Further the notions of component, connector and configuration were presented as key concepts in any architecture and a detailed review of them followed. A connector classification framework was presented to investigate software connectors and classify them according to their properties. Refined experience in form of architectural styles and patterns was found to be a key tool for software architects in the design of new software systems. Furthermore some architecture description languages were mentioned to define an architecture and the ALFA framework was found to be a useful tool for the formal definition of architectural styles by composing them from so called architectural primitives. The notion of quality attributes was presented and the attribute of adaptability was threatened in more detail. It was found that software architecture plays a key role in elicited properties of a software system; adaptability hence depends above all on a software systems architecture and especially on the connectors used in it. Thus, a review on architectural styles promoting adaptability finally concludes the literature review.

4 Research Approach

The following section describes the approach taken in the definition and analysis of the observed style. For this purpose the section begins first by presenting an appropriate method to acquire reliable knowledge about an architectural style. In section 4.1, the architectural style analysis method (ASAM) is derived by adopting the scientific method for architectural style purposes. Subsequently the application of the method in the definition and analysis of the new architectural style is explained in section 4.2.

4.1 ASAM: The Architectural Style Analysis Method

According to Schafersman [**1**], science is the only method that results in the acquisition of *reliable* knowledge, that is knowledge that has a high probability of being true because its veracity has been justified by a reliable method. Therefore ASAM is derived from the scientific method by applying former to the definition and analysis of software architectural styles, in order to acquire reliable knowledge about them. Schafersman places the scientific method within a context of scientific thinking, which in turn is based on three central components: empirical evidence (empiricism), logical reasoning (rationalism) and possession of a skeptical attitude (skepticism). Roughly speaking Schafersman describes the scientific method in form of a process consisting of following activities:

1. *Identification of a problem or phenomena*: A problem or phenomena identified in nature is defined formally to enable further investigation. Any attempt to gain knowledge must start here.
2. *Observation of the problem or phenomena*: The problem or phenomena under investigation will be observed in order to acquire relevant information about it. These observations, and all that follow, must be empirical in nature; they must be sensible, measurable, and repeatable, so that others can make the same observations.
3. *Proposing a solution to the problem or an explanation of the phenomena*: On top of the empirical observation a set of suggested solutions or explanations can be derived by inductive reasoning. In science these claims are called scientific hypotheses and must be predictive and testable. A definition of hypotheses can be found in section 2.
4. *Testing the hypothesis*: In this step, one has to deduce predictions from the hypotheses and test them. The hypotheses and its predictions must be tested either by performing experiments or by making further observations.
5. *Accept or reject the hypothesis*: If the hypotheses fail the tests they have to be abandoned or modified and tested again. If they pass the tests it becomes a corroborated hypothesis and can now be tested by other scientists. If it passes further testing, it becomes highly corroborated and finally a scientific fact. For definition of latter one can again consider section 2.
6. *Construct, support, or cast doubt on a scientific theory*: Such a theory explains nature by unifying many, once unrelated facts or corroborated hypotheses; they are the strongest and most truthful explanations of how the universe, nature, and life came to be, how they work, what they are made of, and what will become of them.

One can now apply this process for the definition and analysis of software architectural styles by adapting each single activity: The problem or phenomena under investigation would be a new *architectural style* observed in industry. The style must then be defined formally using a formal architecture description language. *Observations* of the style in action, empirical in nature, must follow. On top of them, *hypotheses* on the elicited properties of the style can be stated by inductive reasoning. These hypotheses and deduced predictions must then be *tested* by other observations or experiments before they become corroborated and eventually scientific facts. Finally the corroborated hypotheses and facts can then be used to develop a *theory* explaining how the style elicits the identified properties.

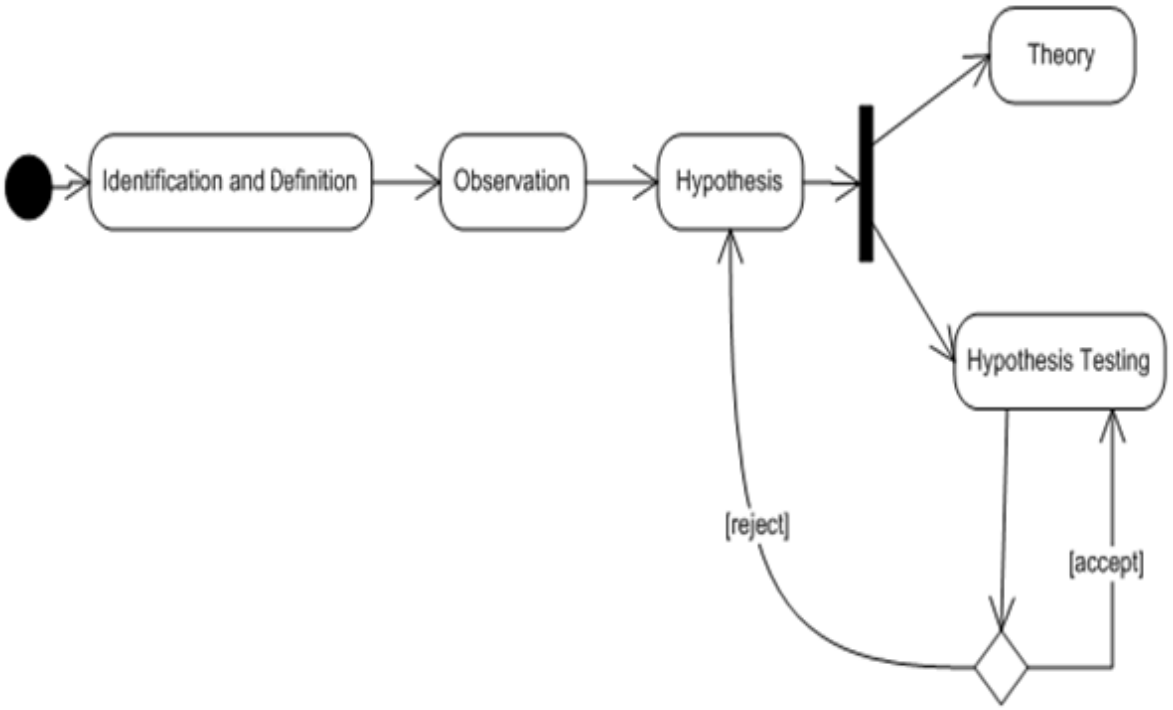

Figure 9: Activities of ASAM depicted in an UML activity diagram.

Figure 9 depicts this situation graphically in an UML [**32**] activity diagram. It follows now a closer look on the single activities of the process:

1. *Identification and Definition*: In this stage at the beginning of the process, a new architectural style must be identified. The new style must then be defined using a formal architectural description language. In literature a lot of such languages exists, but in this text the ALFA [**12**] framework is recommended as it allows composition of a new style from so called architectural primitives.
2. *Observation*: The next activity deals with observing the style in action. The observations must be empirical in nature, which means they must be sensible, measurable, and repeatable. This text recommends the use of software metrics [**44**] and more specific the goal-question-metrics framework [**11**] to define how to gather empirical evidence of the style.
3. *Hypothesis*: On top of the empirical observations one can now induce hypothesis about the properties elicited by the architectural style. Such hypothesis must be predictive and measurable and should therefore be stated formally using some kind of formal language.
4. *Hypothesis Testing*: Once defined, the stated hypotheses must be tested and either accepted or rejected. In order to do so, predictions should be deduced from the hypotheses and checked for consistency with new observations or experiments. If the hypotheses fail, they must be modified and tested again or even abandoned, if they pass, they become corroborated and have a high probability of being true. A corroborated hypothesis must be tested again and can be used in the construction of a scientific theory.
5. *Theory*: If a set of corroborated hypothesis exists, a theory emerges which explains how the style under investigation addresses its elicited properties.

This text deals only with the first three steps of ASAM: first a new style is identified and defined formally using ALFA, then an empirical observation of the style in action follows and finally some hypothesis are derived by induction and described formally. The hypotheses must be further tested in order to become corroborated and reliably describe the elicited properties. Consider section 8 for a suggestion on future work.

4.2 ASAM in Action

This section depicts the application of the early defined method for the definition and analysis of a new observed style. As stated above this text deals only with the first three steps of ASAM: the definition of the style, an empirical observation of the style in action and the derivation of some hypothesis by inductive reasoning based on the observation.

4.2.1 Identification and Definition

The investigated style is discovered in Microsoft's Dynamics AX product line. It follows an extensive research in current literature, as proposed by Yair and Timothy in [**15**] for a formal treatment of the style. Unfortunately only a rather shallow depiction of the style in form of some "box-and-line" diagram can be found in [**13**], lacking rigor and reliability. Hence this work begins in section 5 by a formal definition of the style under investigation as proposed by ASAM.

4.2.2 Observation

Following ASAM, the next major step deals with an empirical observation of the style in action. The observation is performed in-context as suggested in [**45**] on a local Microsoft Dynamics AX partner supplying a variant for the product line. Empirical data is collected for 5 different customer implementations. However, due to timing conflicts, the data collection session has to be conducted before useful software metrics [**44**] are defined. Therefore data elevation is split in three distinct phases: *data collection* to collect rather general raw data; *metrics definition* to define the metrics needed for the empirical observation; and *data extraction* to get the defined metrics out of the general data.

4.2.2.1 Data Collection

To collect general raw data about the Microsoft Dynamics AX systems, a Microsoft Dynamics AX Code Metrics Collection (CMC) framework is developed for Microsoft Dynamics AX 4 and Microsoft Dynamics AX 2009 respectively. The framework is implemented using Microsoft's proprietary X++ language on top of a standard Microsoft Dynamics AX implementation. The framework enables easy development of individual tasks by simply implementing a given interface. The tasks can then be configured to run on specified entities belonging to defined layers of the system. Figure 10 depicts the logical part of the CMC tool in form of an UML [**32**] class diagram.

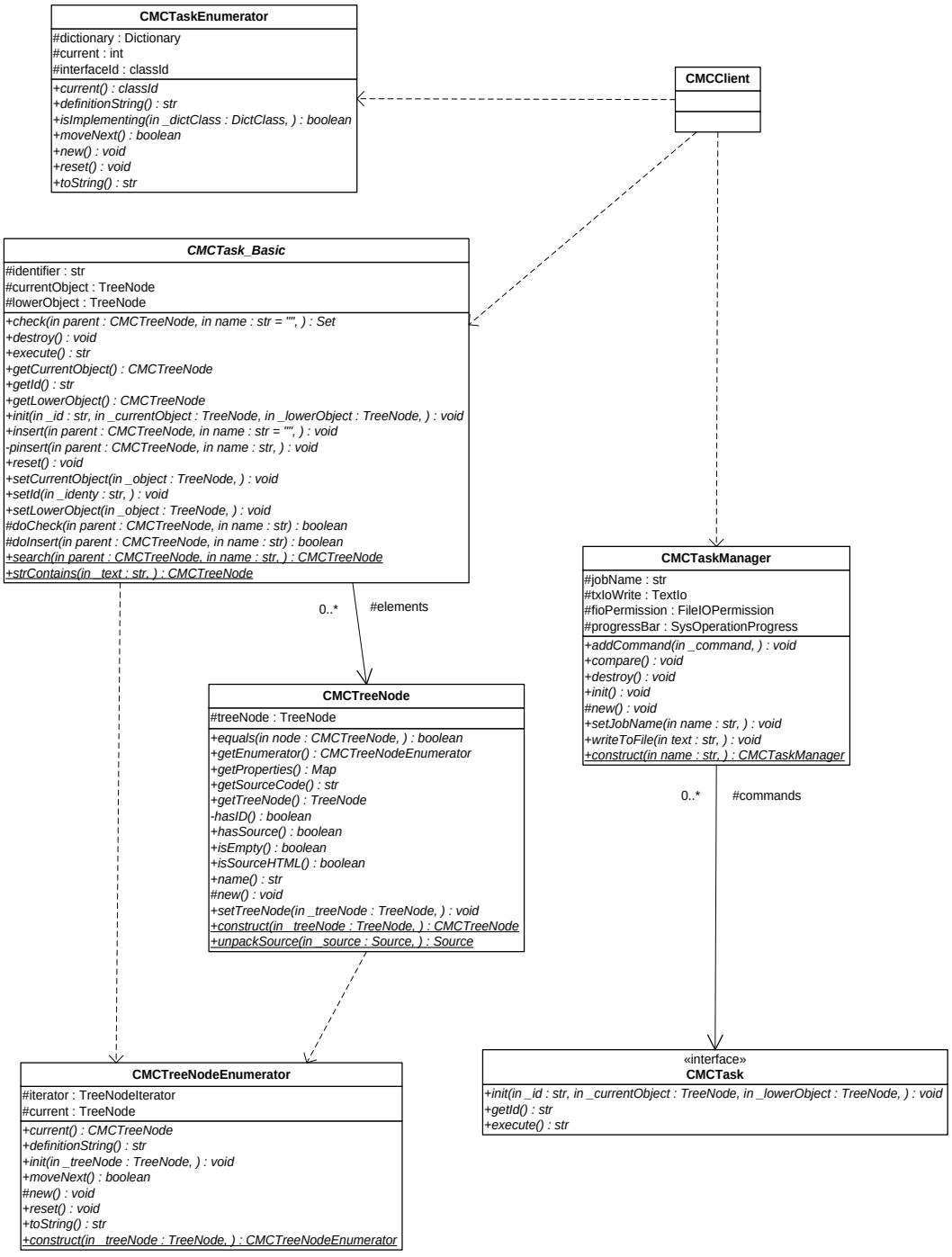

Figure 10: UML class diagram of the CMC tool.

As one can recognize, the framework is developed in the *command design pattern* [**4**]. A *CMCClient* (Client) instantiates various *CMCTask* implementations (Command), provides the required *TreeNode* Objects (Receiver) and passes them to the *CMCTaskManager* (Invoker). Latter finally invokes the commands at a later time. Using the command pattern enables easy integration of progress bars and the execution of the code as batch at a specified time. Furthermore an abstract class *CMCTask_Basic* is provided to implement the default behavior of a *CMCTask* implementation.

To get a better understanding of the CMC tool, Figure 11 and Figure 12 depicts the graphical user interface of the framework where configurations can be made and finally started. Former figure shows the tab where the domain can be selected on which the tasks should run. To configure the domain, the type of entity and the layer must be specified. Latter figure shows the tab where the name of all the tasks appears and can be enabled or disabled. As one can recognize there is also the possibility to specify an output file where the measures will be stored.

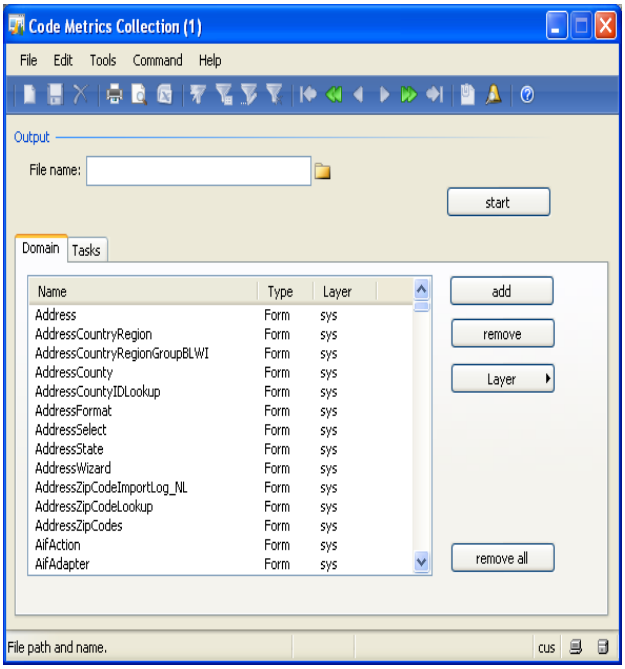

Figure 11: Graphical user interface of CMC showing the domain configuration

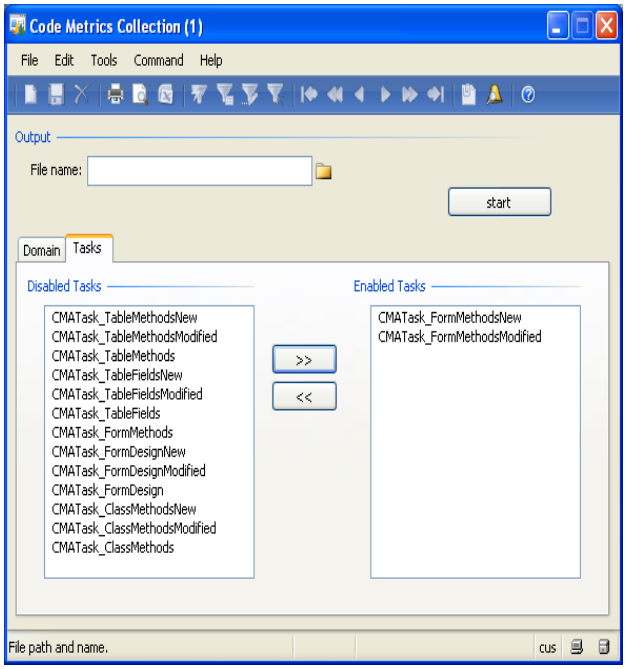

Figure 12: Graphical user interface of CMC showing the task configuration

In order to collect the general raw data of the investigated Microsoft Dynamics AX systems, 10 CMC task are implemented. Table 1 summarizes the tasks and the type of metric collected by each task.

Task Name	Task Description
CMCTask_ClassMethods	class name and the method names of the class in the current layer
CMCTask_FormDesign	form name and the control names of the form in the current layer
CMCTask_FormMethods	form name and the method names of the form in the current layer
CMCTask_TableFields	table name and the field names of the table in the current layer
CMCTask_TableMethods	table name and the methods names of the table in the current layer
CMCTask_ClassMethods_mod	class name and the name of its methods modified in the current layer
CMCTask_FormDesign_mod	form name and the name of its controls modified in the current layer
CMCTask_FormMethods_mod	form name and the name of its method modified in the current layer
CMCTask_TableFields_mod	table name and the name of its fields modified in the current layer
CMCTask_TableMethods_mod	table name and the name of its methods modified in the current layer

Table 1: Tasks implemented to collect raw data for further analysis.

Finally the tasks are run by the CMC framework on all fife Microsoft Dynamics AX systems and the collected raw data is stored for further analysis after metrics definition.

4.2.2.2 Deriving Metrics

In this section the goal-question-metrics framework [**11**] is used to derive useful software metrics [**44**] for the empirical analysis of the style. First of all the goal of the analysis is presented, leading to the questions threatened explicitly later in the section and finally the metrics are defined by the end of the section.

The goal of the empirical observation might be defined as follows:

- G: Observe the architectural style defined in section 4.2.1 in action to gather relevant information about the nature of the style.

In order to narrow the focus, a set of 11 questions were directly derived to address this goal:

- Q1: What is the size of the Base component? (M4)
- Q2: How is the Base component partitioned into Code, Data and Design entities? (M1, M2, M3)
- Q3: How many Code Elements were added to the Base component for each customer implementation?(M5)
- Q4: How many Code Elements of the Base component were modified for each customer implementation?(M6)
- Q5: How many Data Elements were added to the Base component for each customer implementation?(M7)
- Q6: How many Data Elements of the Base component were modified for each customer implementation?(M8)
- Q7: How many Design Elements were added to the Base component in each implementation?(M9)
- Q8: How many Design Elements of the Base component were modified for each customer implementation?(M10)
- Q9: How many Code Elements of the Base component were modified for each single Entity of each customer implementation?(M11)
- Q10: How many Design Elements of the Base component were modified for each single Entity of each customer implementation?(M12)
- Q11: How many Data Elements of the Base component were modified for each single Entity of each customer implementation?(M13)

Before metrics can be derived on top of the above questions, the notion of Code, Data and Design Element has to be defined. Each Element reflects an atomic unit of the corresponding type. For the Code Element, the unit must comprise a set of computational logic and is therefore defined as *method.* Data Elements represent the atomic unit of data which is defined as *field* and Design Elements encapsulate the atomic design unit and are defined as design *control*. Another primitive which needs to be defined is the notion of an Entity. An Entity is defined as a collection of Elements of some type. Thus, a class in the object oriented paradigm represents an Entity composed of Code Elements. A form in Microsoft Dynamics AX however is an Entity comprising Design and Code Elements. On top of these primitives a set of functions can be defined to support the metric development:

$$num(t, e) = Number\ of\ Elements\ of\ type\ t\ per\ Entity\ e\ in\ base$$

$$add(t, e) = Number\ of\ Elements\ of\ type\ t\ added\ per\ Entity\ e\ in\ base$$

$$mod(t, e) = Number\ of\ Elements\ of\ type\ t\ modified\ per\ Entity\ e\ in\ base$$

With these primitives one can now define following metrics to answer the questions identified earlier in this section:

- $M1: \sum_{each\ Entity\ e\ in\ base} num(code, e)$
- $M2: \sum_{each\ data\ Entity\ e\ in\ base} num(data, e)$
- $M3: \sum_{each\ design\ Entity\ e\ in\ base} num(design, e)$
- $M4: M1 + M2 + M3$
- $M5: \sum_{each\ Entity\ e\ in\ base} add(code, e) + \sum_{each\ added\ Entity\ e\ in\ refinement} num(code, e)$
- $M6: \sum_{each\ Entity\ e\ in\ base} mod(code, e)$
- $M7: \sum_{each\ data\ Entity\ e\ in\ base} add(data, e) + \sum_{each\ added\ data\ Entity\ e\ in\ refinement} num(data, e)$
- $M8: \sum_{each\ data\ Entity\ e\ in\ base} mod(data, e)$
- $M9: \sum_{each\ design\ Entity\ e\ in\ base} add(design, e) + \sum_{each\ added\ design\ Entity\ e\ in\ refinement} num(design, e)$
- $M10: \sum_{each\ design\ Entity\ e\ in\ base} mod(design, e)$
- $M11: \bigvee_{Entity\ e\ of\ base} \frac{mod(code,e)}{num(code,e)}$
- $M12: \bigvee_{data\ Entity\ e\ of\ base} \frac{mod(data,e)}{num(data,e)}$
- $M13: \bigvee_{design\ Entity\ e\ of\ base} \frac{mod(design,e)}{num(design,e)}$

To get a better understanding of how each single component of the GQM directly influenced the development of other components, Figure 13 depicts the relationship of goal, questions, and metrics.

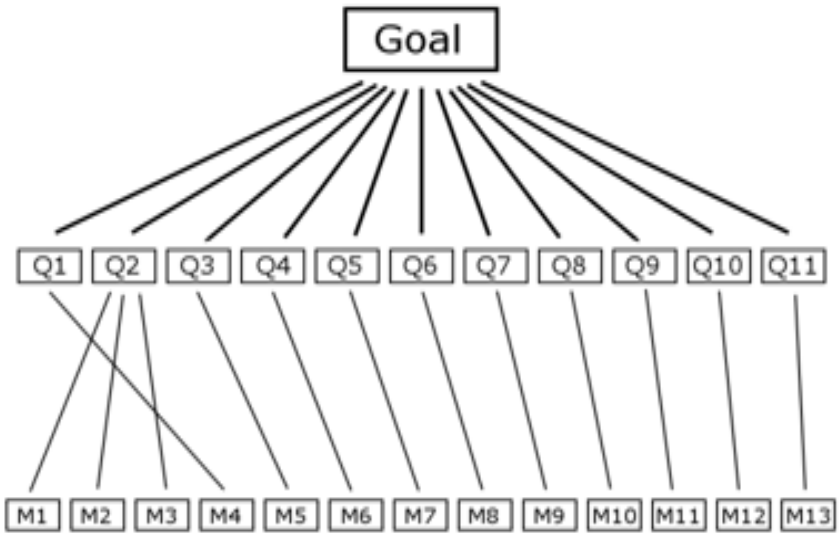

Figure 13: Graphical representation of the GQM showing the relationship between goal, questions, and metrics.

4.2.2.3 Data Extraction

In order to extract the derived metrics from the general data collected by the CMC tool, another tool is developed in java and groovy. Figure 14 shows the UML [**32**] class diagram of the Code Metrics Analysis (CMA) tool. A groovy script is used to parse the general raw data obtained by the CMC tool and construct an in memory model of this data using the corresponding Java classes. Finally the script uses the in memory model to calculate the derived metrics and outputs the results into a new file.

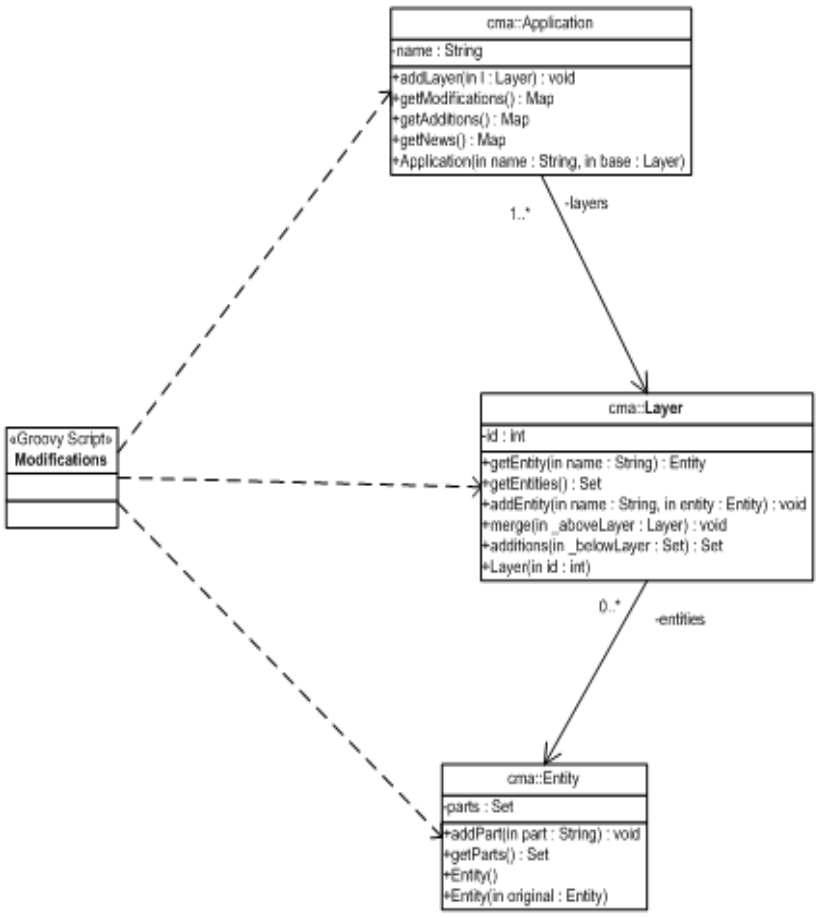

Figure 14: UML class diagram of the CMA tool.

The data extracted by the CMA tool is finally imported into Microsoft Excel for further analysis. The detailed report of the data can be found in section 6.

4.2.3 Hypothesis

On top of the insights uncovered by the empirical observation a set of related hypotheses is derived. Section 7 provides a formal model on top of the empirical evidence, consisting of 4 hypotheses on properties elicited by the style. A call to further test the derived hypotheses follows in section 8 where suggestions for future work can be found.

.

5 Identification and Definition

The following section defines the observed style formally, by first depicting a high level structural representation of it in xADL's [**40**] canonical visualization Archipelago [**46**] before leveraging the ALFA framework [**12**], threatened in detail in section 3, to characterize the style and decompose it into its architectural primitives. The section concludes with an analysis of the connector type, which is a crucial component for every style, thus applying the framework proposed by Mehta et al. in [**31**].

5.1 A new Style emerges: Partial Refinement

The style, named *Partial Refinement*, can be classified according to the outline used by Taylor et al. [**3**] into the class of *layered styles*. To provide a rough overview of the style, Figure 15 depicts its structure according to xADL's visual representation Archipelago:

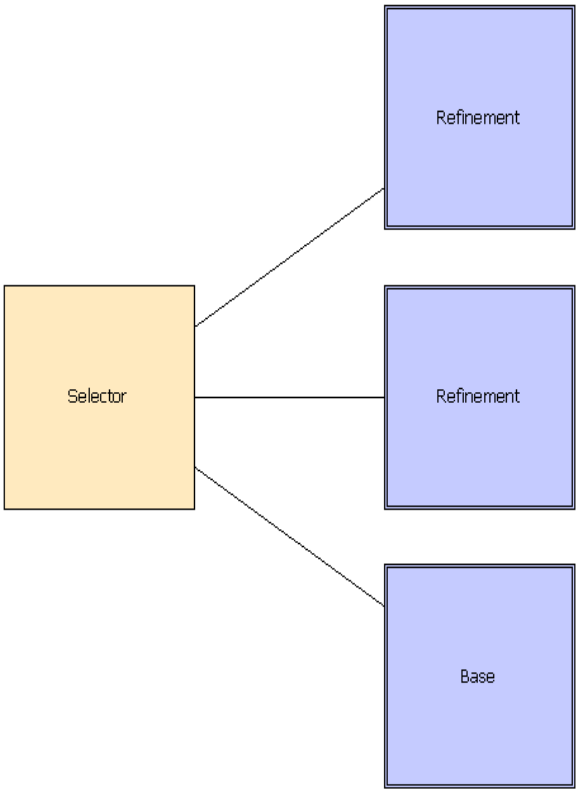

Figure 15: Partial Refinement architectural style depicted in xADL's visual representation Archipelago.

As one can recognize, the architectural style is composed of a *Selector* connector, a *Base* component, and one or more *Refinement* components. The Base component contains the standard logic of the application partitioned through a collection of *Entities* of different types. Each Entity is further decomposed into *Elements* of some type. Each Refinement component can then adapt the standard application logic by adding new Entities or changing the behavior of some of the Entities. Changing the behavior of an Entity can be achieved by adding new Elements to or modifying the properties of some existing Elements of the Entity. The Selector connector coordinates the execution of the application by selecting the correct component and transferring data and control to the Entity in this component. A detailed treatment of the Selector connector follows in section 5.3.

5.2 Definition

The following section defines the style presented above formally to provide a robust foundation for further investigation. Table 2 describes the style first using the five-way characterization proposed in [**12**]:

Data	Each single Element of each Entity has a unique *identifier* within each component. However the identifier is shared between components to enable adaptations of an Element through Refinement components. Each Element further executes within a well defined *context*. This context is defined either by the Elements Entity or it can be passed explicitly by the Elements caller. Latter could be parameters in case the Element is a method or configurations if the Element is a design control. A simple *boolean* value is used to acknowledge the availability of an Element in a component.
Structure	The style is composed of one *Base* component which contains the general application logic and a set of *Refinement* components which can adapt the application by enhancing or modifying the Base component. The logic is distributed across so called Entities of different types belonging to some component. Each Entity is further decomposed into Elements which represents the atomic units in the architectural style. By implementing an Element in a Refinement component, the behavior of the corresponding Entity in the Base component can be modified. Furthermore one *Selector* connector coordinates the execution of the application. If an Entity needs to consider another Entity, it calls on the Selector which then delegates the call to the Entity in the correct component.
Interaction	The Selector *blocks* if he checks the availability of the Entity in the refinement components. However the calling for the execution of an Entity by the components and the actuating of the execution of the Entity are *non blocking* interactions.
Behavior	If a component needs to consider an Entity, it calls on the Selector connector. The connector then checks each Refinement component on the availability of the Entity. After being notified by all components about the availability of the Entity, the Selector considers a *priority* list and actuates the execution of the Entity on the component with the highest priority.
Topology	A Selectors *call* port is connectable to a components *execution* port. Furthermore, a components *call* port is connectable to the Selectors *select* port. The Selectors *check* port is connectable to a Refinement components *confirm* port.

Table 2: Five-way characterization of the Partial Refinement architectural style.

On top of this characterization one can now decompose the style into its primitives. The identified *Datum* primitives in this style are the Elements identifier, the execution context and a boolean value to acknowledge availability. There are 3 *Particle* primitives identified, a Base component, some Refinement components and a Selector connector. There is further one *Output* portal for the Base component to call on the Selector for executing an Entity and one *Input* portal to be notified about the need to execute some Entity. Similarly, a Refinement component has one *Output* portal for demanding the execution of an Entity and one *Input* portal to get incoming execution calls. However the Refinement component has also a *Twoway* portal to get notified about the need of executing an Entity and eventually confirm the availability of the Entity in the component. The Selector connector is provided with an *Output* portal to demand the execution of an Entity and an *Input* portal to get notified about the demand of executing an Entity. Furthermore the Selector provides a *Twoway* portal to check on the existence of an Entity in all Refinement components. Figure 16 shows the graphical composition of the style, uncovering the internal behavior of the Selector connector.

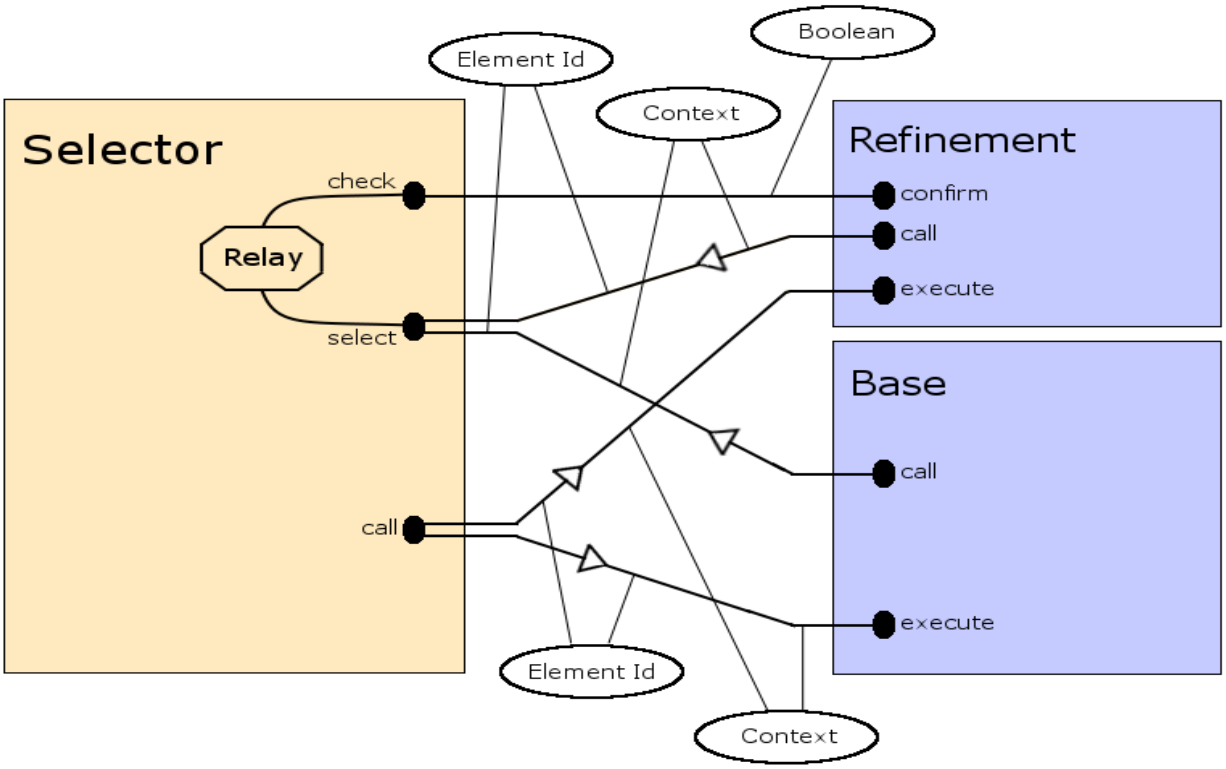

Figure 16: "Architectural primitives" forming the elements of the Partial Refinement architectural style.

From the above figure, the interaction characteristics of the style become clear. The Selectors call Output portal is connected to the execute Input portal of the Base and Refinement components through a *Duct* with functions one *holds* and none *loses*. Similarly, the components call Output portal is connected with the Selectors select Input portal using a *Duct* with one *holds* and none *loses*. Finally, the Selectors check Twoway portal is connected with the Refinement components confirm Twoway portal through a zero *holds*, and none *loses Duct*. Note the *Relay* used by the Selector connector to connect the select Input portal with the Twoway check portal to call on each Refinement component for the availability of the required Entity.

5.3 The Selector Connector

As for all styles the connector plays a fundamental role for this style and hence requires a deeper analysis. Thus, the connector classification framework proposed by Mehta et al. [**31**], and described in more detail in section 3, is leveraged in the following analysis of the Selector connector.

The Selector connector provides three types of services, which are *communication*, *coordination*, and *facilitation* services. The connector is a so called composite or higher order connector, composed of two base types: *event* connector, providing communication and coordination services; and *distributor* connector, providing facilitation services. The Selector connector sends and receives data through asynchronous notifications called events. An event arrives as an Entity in a component needs to call on another Entity. The Selector checks the availability of the Entity in Refinement components and produces a new event to notify a component about the need to execute one of its Entities. Furthermore, the connector acts as mediator and distributes communication and coordination information among different components according to their priority.

6 Observation

In the following section the results of the empirical observation of the Partial Refinement architectural style in action are presented and analyzed. The metrics derived in section 4.2.2.2 were applied to 5 systems build according to the Partial Refinement style defined earlier. Two of them are Microsoft Dynamics AX 4, the other three Microsoft Dynamics AX 2009 implementations. The following two sections provide a general overview of the Base component of Microsoft Dynamics AX 4 and Microsoft Dynamics AX 2009 respectively, a description of the domain for which to refine the Base component, and an overview of the Refinement components for each implementation. It follows a deeper insight into the nature of the Refinement components by a detailed analysis of the modified Elements for each of the Entities in the Base component. Finally the section concludes with a statistical analysis of the observed data in section 6.4.

6.1 Case 1

The first observations were done on two implementations of Microsoft Dynamics AX 4, whose Base component is composed of three main types of Entities: Code Entities, containing Code Elements; Data Entities, composed of Code Elements and Data Elements; and Design Entities, comprising Code Elements and Data Elements.

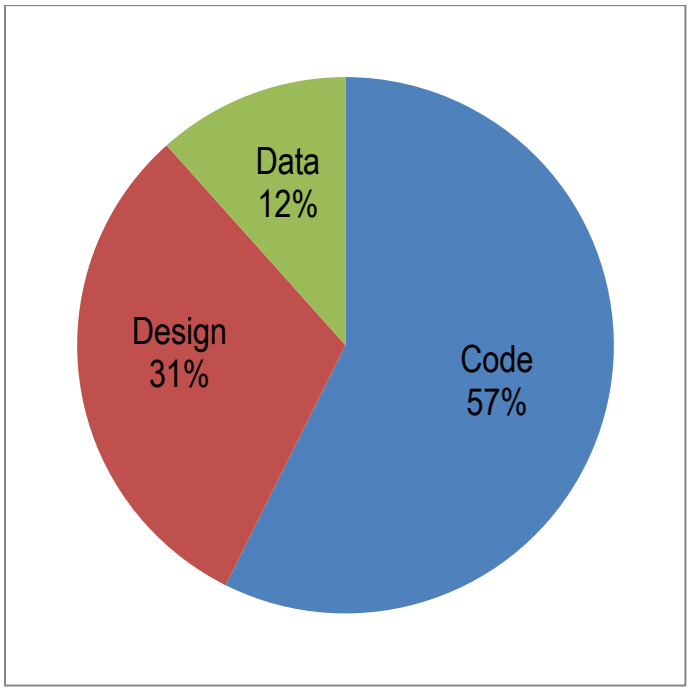

Figure 17: Entity types in Microsoft Dynamics AX 4 Base component.

Figure 17 depicts the distribution of Entity Elements across the different types. More than 50% of Elements in the Base component are Code Elements belonging to Code Entities, Design Entities or Data Entities respectively. More than a fourth of the Elements are Design Elements belonging to Design Entities and only a tenth of the Elements are of type Data belonging to Data Entities.

Before proceeding with the observation, it follows now a brief description of the two subjects, that is, the refined Microsoft Dynamics AX 4 implementations.

6.1.1 Subject 1-1

The first observation was done on a customization of Microsoft Dynamics AX 4 for a wholesales company with 74 users. The company is located in Germany and supplies instruments for the car industry with an assortment of about 10000 goods for almost 55000 customers.

6.1.2 Subject 1-2

The second observation was done on a Microsoft Dynamics AX 4 customization for a software house with around 50 users. The company with headquarter in Italy has branch offices in Germany and Hungary and supplies about 10 products for 100 customers.

6.1.3 Results

The following figures provide an overview of the refinement done on each type of Element for each of the two subjects. In these figures, the blue pie depicts the number of modified/added Elements in Refinement components relative to the total number of Elements in the Base component. The other pie presents the distribution of these modifications/additions across the different subjects and the number of Elements modified/added respectively. The red part represents the modifications/additions common to both of the refinements, while the green and violet parts show the number of modifications/additions performed exclusively for one refinement. Thus, the total number of refinements done on one of the subjects comprises the red part (common refinements) and the corresponding other part (exclusive refinements).

Figure 18 depicts the modification of existing Code Elements by Refinement components and Figure 19 shows the number of Code Elements added by the Refinement components.

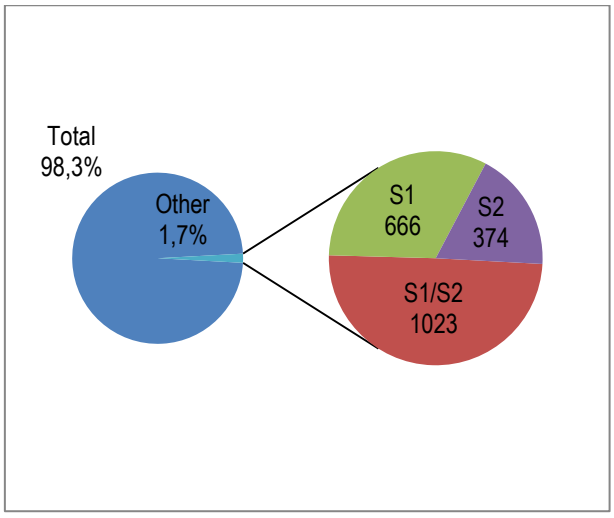

Figure 18: Modifications of Code Elements for each subject.

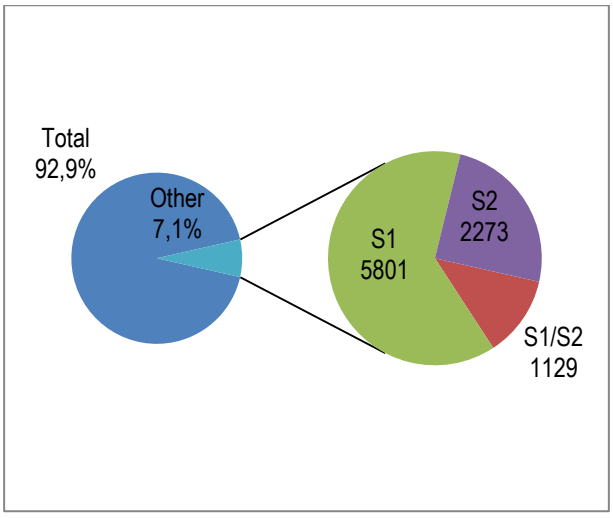

Figure 19: Addition of Code Elements for each subject.

Figure 20 shows the number of Design Elements modified for each of the subjects and Figure 21 shows the number of Design Elements added by the Refinement components.

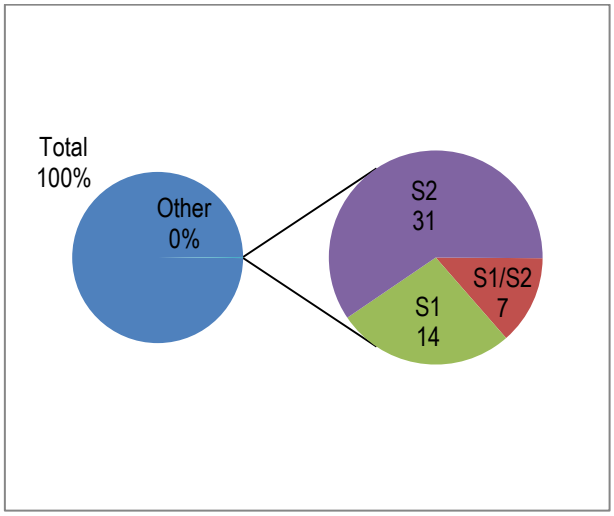

Figure 20: Modifications of Data Elements for each subject.

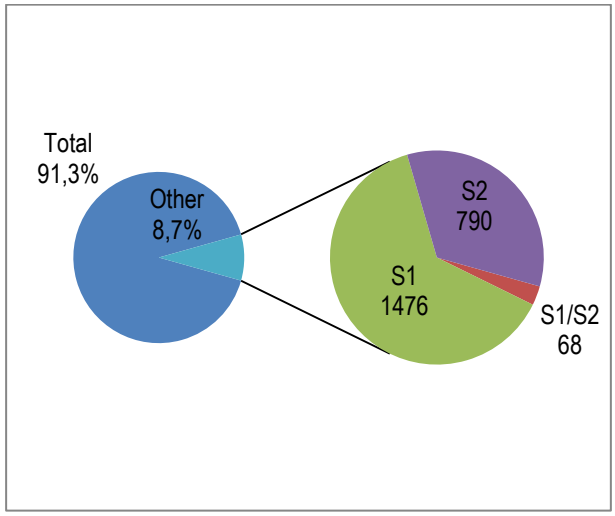

Figure 21: Additions of Data Elements for each subject.

Figure 22 presents the number of Data Elements which were modified by Refinement components and Figure 23 shows the number of Data Elements added for each subject.

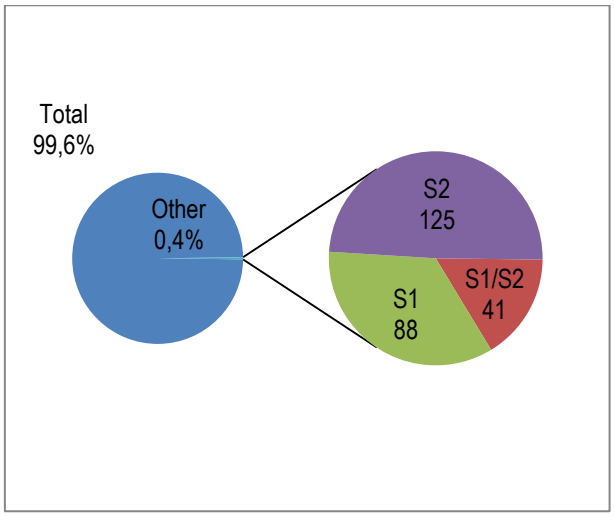

Figure 22: Modifications of Design Elements for each subject.

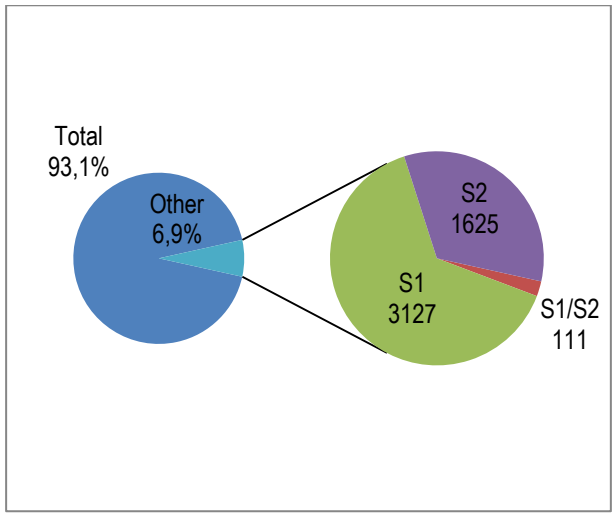

Figure 23: Additions of Design Elements for each subject.

Considering the above figures, it appears that for each type of Element, the number of additions of such Elements by Refinement components is far higher than the number of modifications on existing Elements. Another impression from the above figures is that the modifications and additions of Elements exclusive for one of the systems (violet/green part) is slightly higher than the number of modified or added Elements common to both of the systems. Furthermore it seems that there is a relationship between the number of added Elements and the number of Elements modified. Finally if one considers the numbers in the figures, it seems that there is a significant difference on the number of modifications/additions, depending of the type of the Elements. Section 6.4 applies some statistics to further investigate on these questions.

6.2 Case 2

Further observations were done on three implementations of Microsoft Dynamics AX 2009. As in its predecessor, the Base component is composed of three major types of Entities: Code Entities, containing Code Elements; Data Entities, composed of Code Elements and Data Elements; and Design Entities, comprising Code Elements and Design Elements.

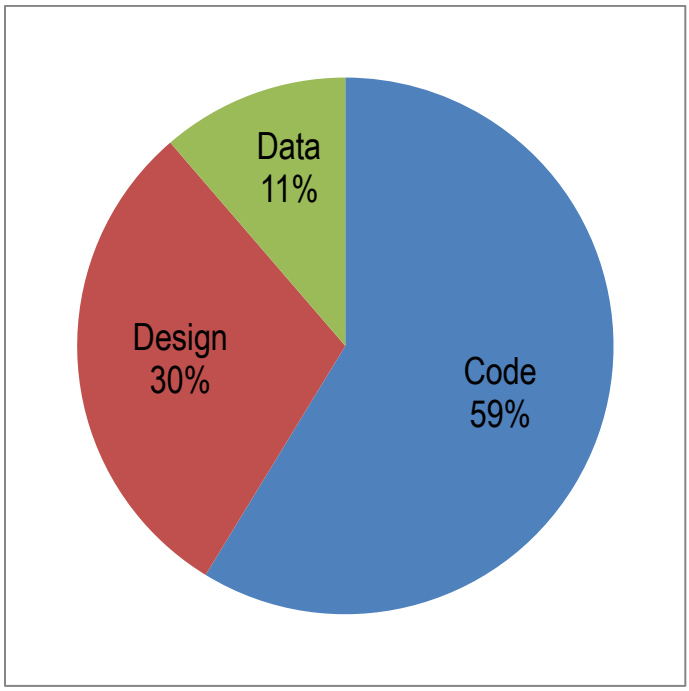

Figure 24: Entity types in Microsoft Dynamics AX 2009 Base component.

Figure 24 depicts the distribution of Entity Elements across the different types. Again, more than 50% of Elements are Code Elements, more than a fourth comprise Design Elements, and only a tenth represents Data Elements.

However, before proceeding with the observation, it follows now a brief description of the three subjects for which the Base component was refined.

6.2.1 Subject 2-1

The third observation was done on a Microsoft Dynamics AX 2009 customization for a wholesales company with around 40 users. The company employs around 50000 employees distributed across 300 units worldwide. The company supplies about 7000 products through 17000 clients.

6.2.2 Subject 2-2

The fifth observation was done on a customization of Microsoft Dynamics AX 2009 for a wholesales company with about 12 users. The company is a small company located in Swiss.

6.2.3 Subject 2-3

The last observation was done on an implementation for a wholesales company with 65 users. The company is located in Canada and supplies about 50000 products.

6.2.4 Results

Again, the following section presents an overview of the refinement done on each type of Element for each of the three subjects. In the following figures, the blue pie depicts the number of modified/added Elements by Refinement components, relative to the total number of Elements in the Base component. The other pie presents the distribution of these modifications/additions across the different subjects and the number of Elements modified/added by Refinement components. The red part represents the modifications/additions common to all three subjects, while the green, violet and cyan blue parts represent commonalities of only two of the subjects and the pink, light blue and orange parts represents the number of Elements modified/added exclusive for each single subject.

Figure 25 shows the number of Code Elements which were modified through Refinement components. On the other hand, Figure 26 shows the number of Code Elements added by these components.

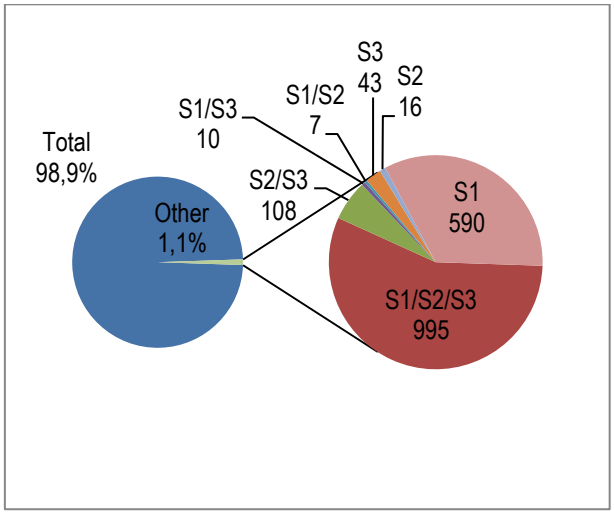

Figure 25: Modifications of Code Elements for each subject.

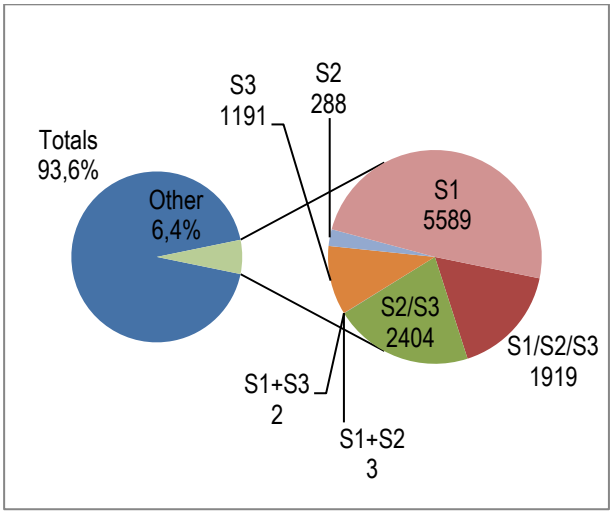

Figure 26: Additions of Code Elements for each subject.

Figure 27 shows the number of Design Elements modified for each of the subjects and Figure 28 depicts the number of Design Elements added in order to refine the Base component.

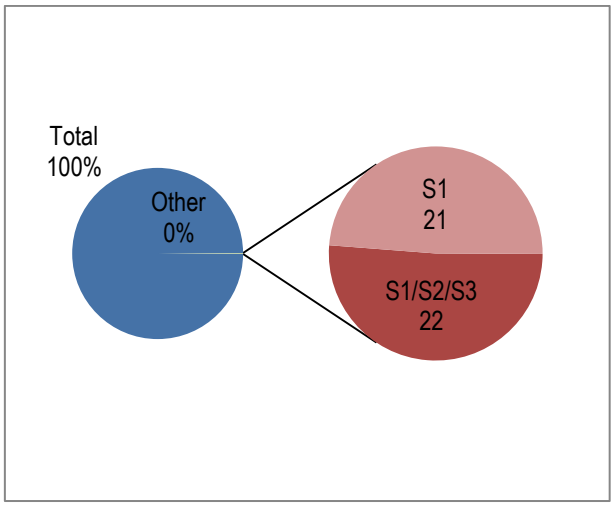

Figure 27: Modifications of Data Elements for each subject.

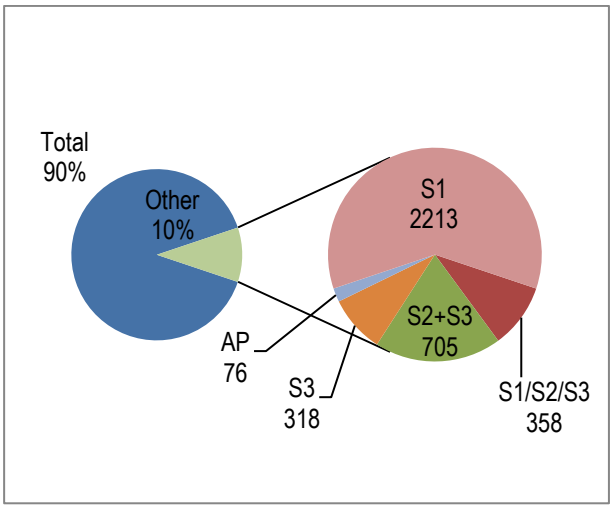

Figure 28: Additions of Data Elements for each subject.

Finally, Figure 29 presents the number of Data Elements modified for each refinement and Figure 30 presents the number of Data Elements added in Refinement components.

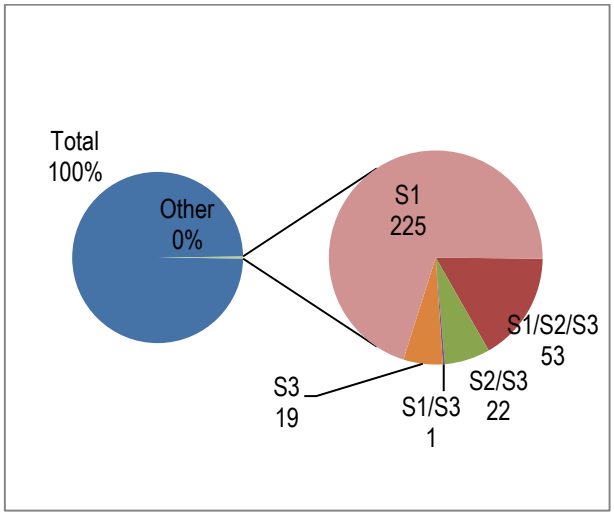

Figure 29: Modifications of Design Elements for each subject.

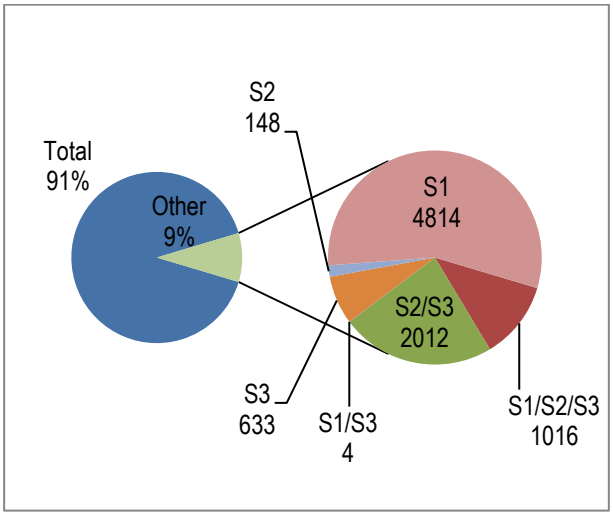

Figure 30: Additions of Design Elements for each subject.

As in the first case presented in section 6.1, it seems that the number of added Elements is several times higher than the number of modified Elements. Furthermore again it seems that the set of Elements reused across different systems decreases with the number of systems for which the Elements can be reused. The figures above strengthen the assumption that there might be a relationship between the number of added Elements and the number of Elements modified. Finally the suspicion that there is a significant difference on the number of modifications/additions, depending of the type of the Elements will be corroborated by the above figures. In section 6.4 a statistical analysis on the data is performed to investigate these assumptions.

6.3 Modifications

The following section now takes a closer look on the nature of the modified Entities. Hence, for each of the 5 subjects it depicts a graphical representation of the modifications done for each Entity. In the following figures, the horizontal axis represents the different Entities modified by a subject, while the vertical axis represents the percentage of Entity Elements. The blue area shows the percentage of Elements not modified by Refinement components, while the red area depicts the percentage of Elements modified and reused for more than one subject. The yellow part however represents the modifications performed exclusively by one of the subjects.

6.3.1 Subject 1-1

Figure 31 depicts a graphical representation of the modification of Code Entities for subject 1-1. Figure 32 on the other hand represents the modifications done on the same subject, but on Design Entities and finally Figure 33 presents the modifications done on Data Entities for subject 1-1 again.

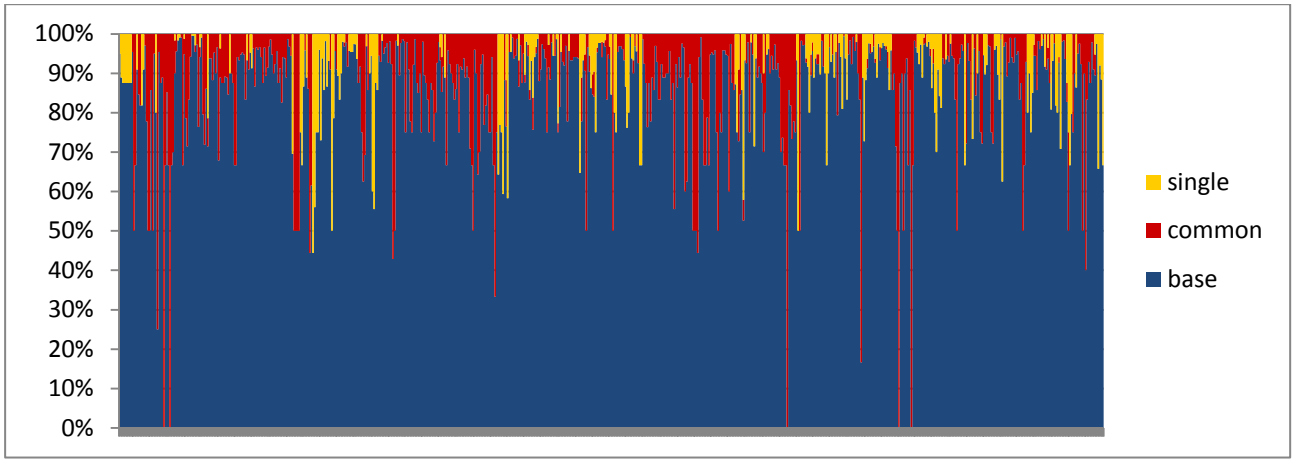

Figure 31: Modified Code Elements per Entity modified in subject 1-1.

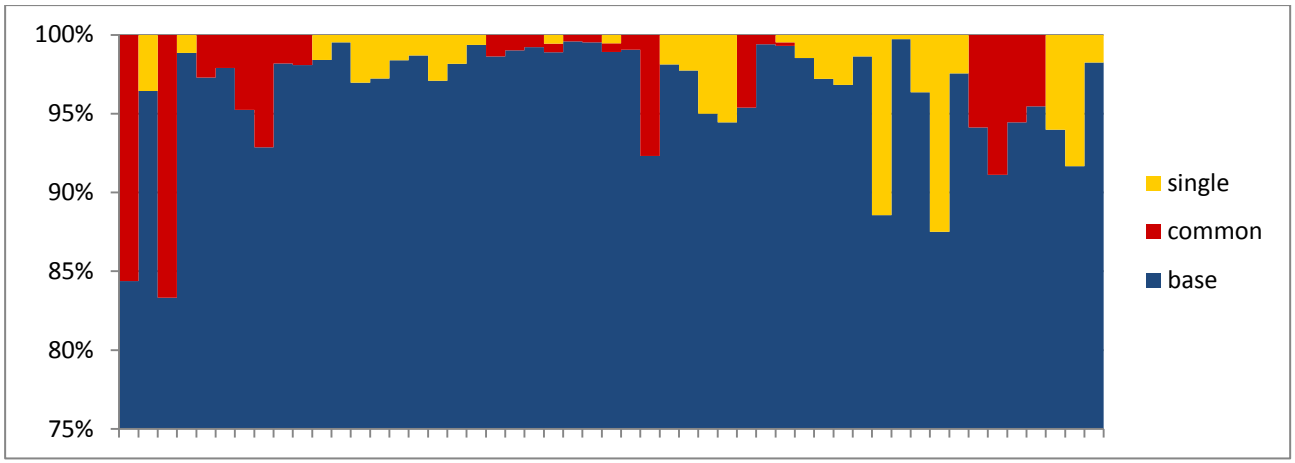

Figure 32: Modified Design Elements per Design Entity modified in subject 1-1.

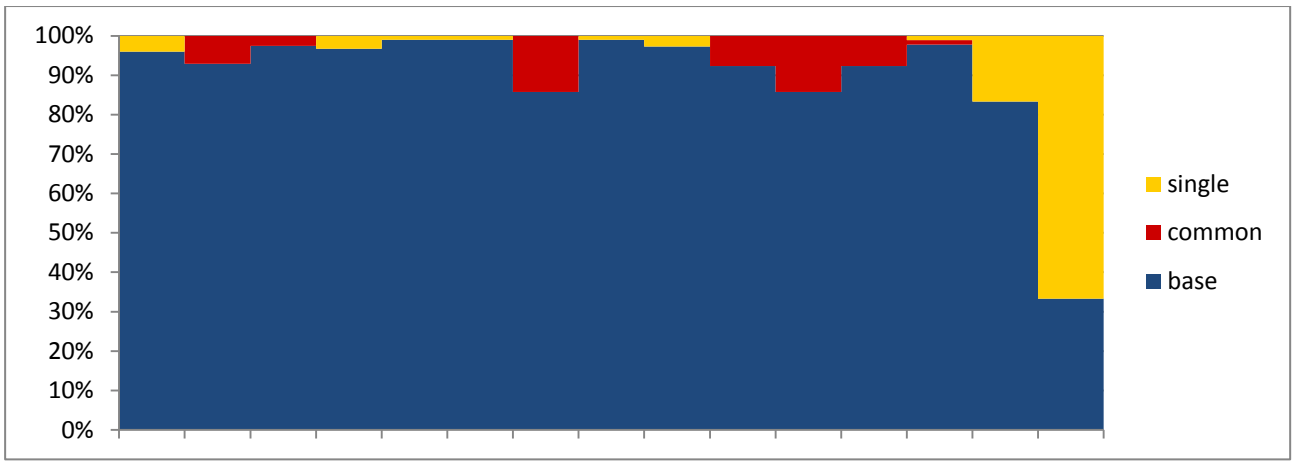

Figure 33: Modified Data Elements per Data Entity modified in subject 1-1.

6.3.2 Subject 1-2

Figure 34 shows the modifications on Code Entities for subject 1-2 while Figure 35 depicts modifications on Design Elements for the same subject. Figure 36 on the other hand represents modifications of Data Elements for subject 1-2.

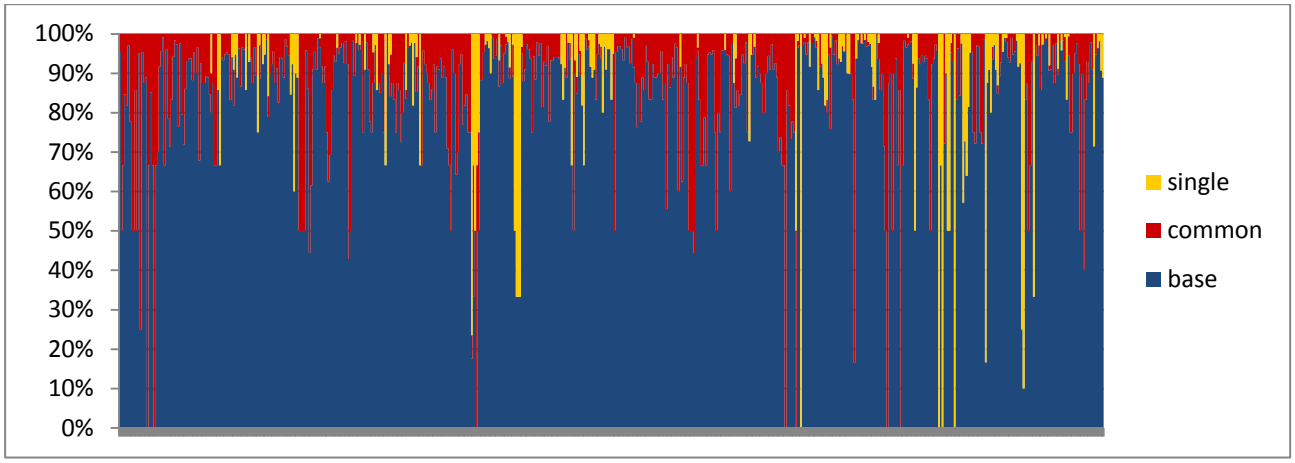

Figure 34: Modified Code Elements per Entity modified in subject 1-2.

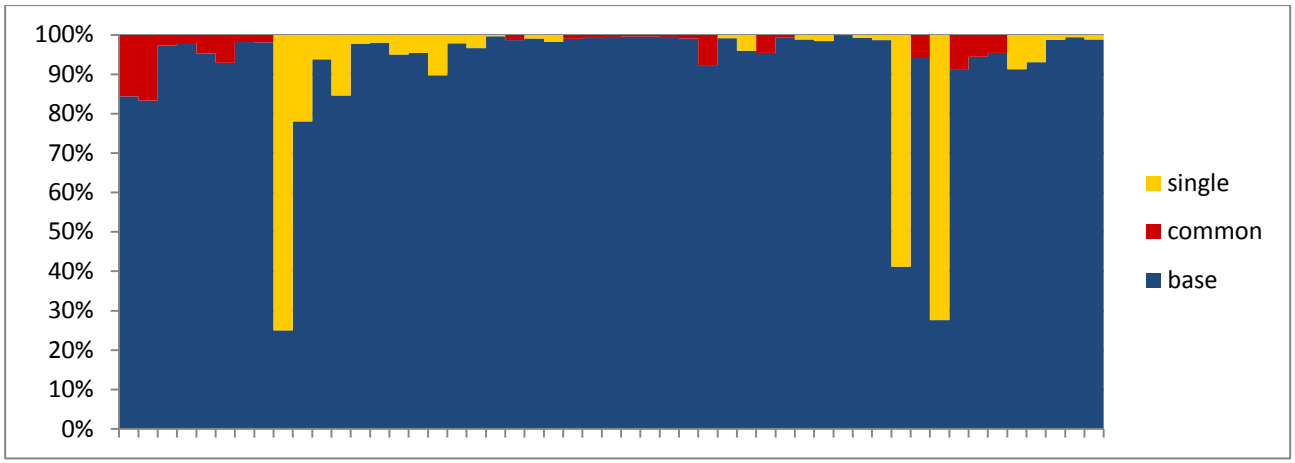

Figure 35: Modified Design Elements per Design Entity modified in subject 1-2.

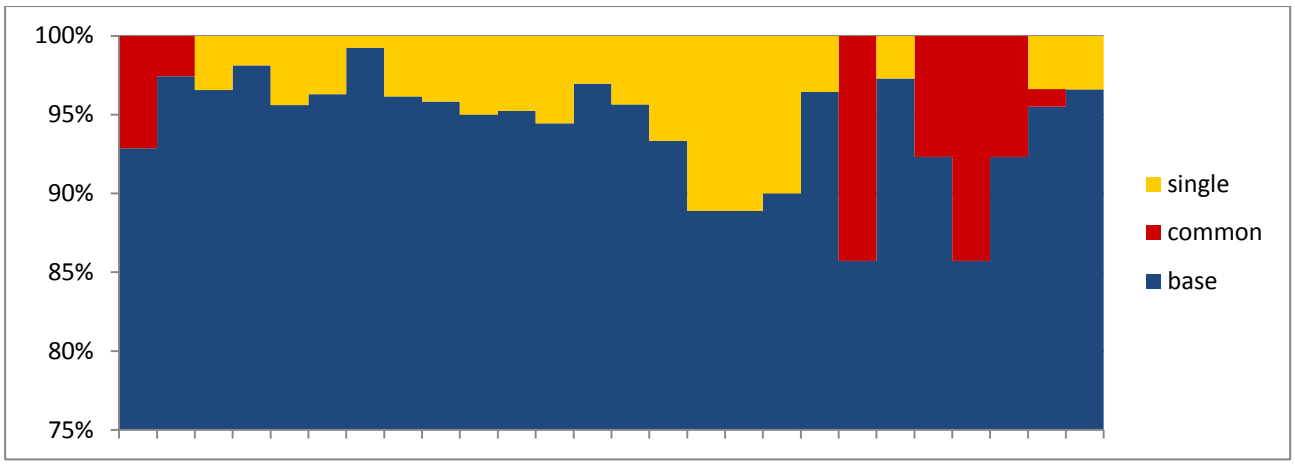

Figure 36: Modified Data Elements per Data Entity modified in subject 1-2.

6.3.3 Subject 2-1

Figure 37 depicts a graphical representation of the modification of Code Entities for subject 2-1. On the other hand, Figure 38 represents the modifications done on the same subject, but on Design Entities and finally Figure 39 presents modifications done on Data Entities for subject 3-1 again.

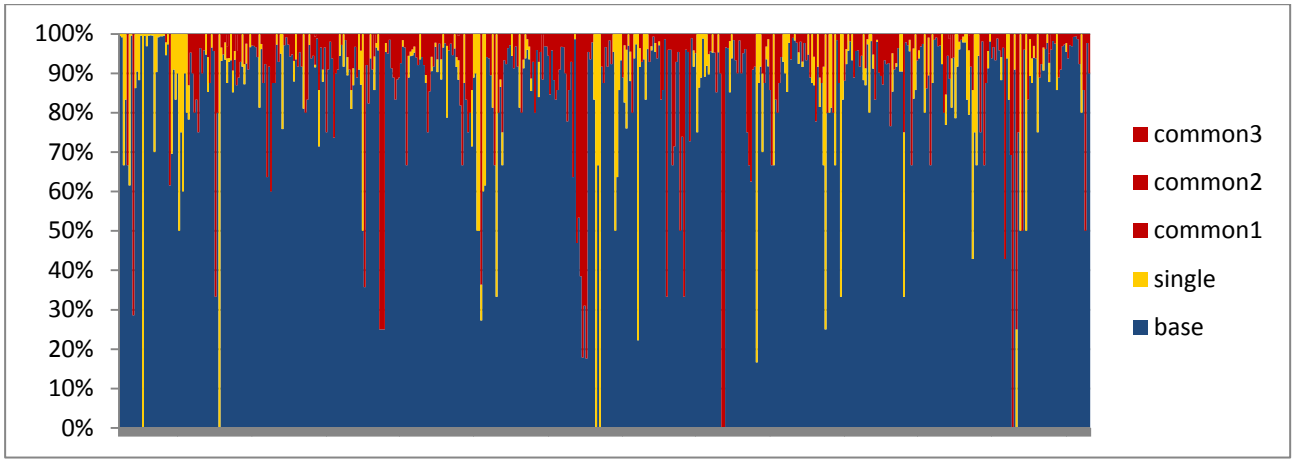

Figure 37: Modified Code Elements per Entity modified in subject 2-1.

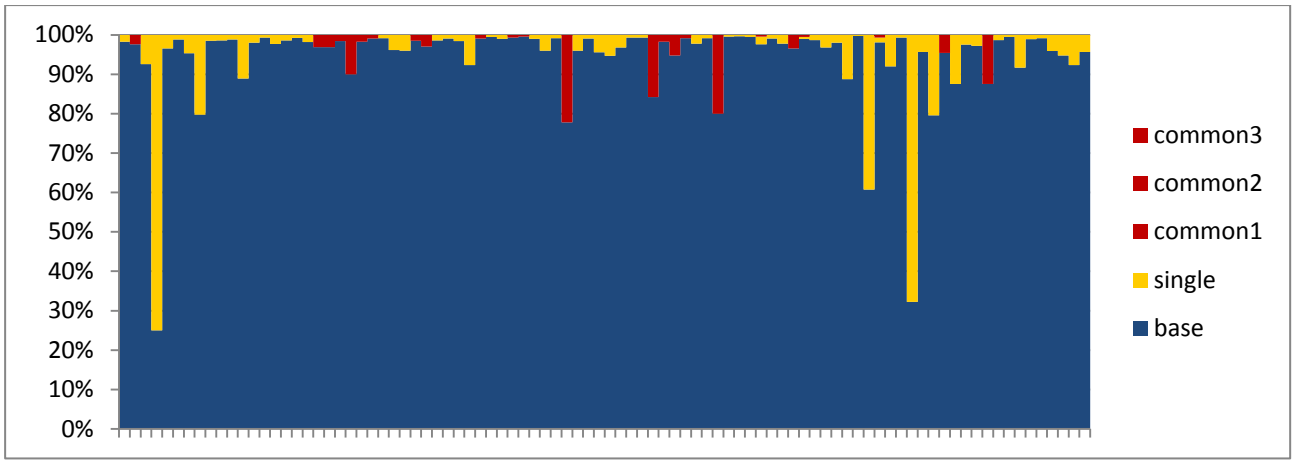

Figure 38: Modified Design Elements per Design Entity modified in subject 2-1.

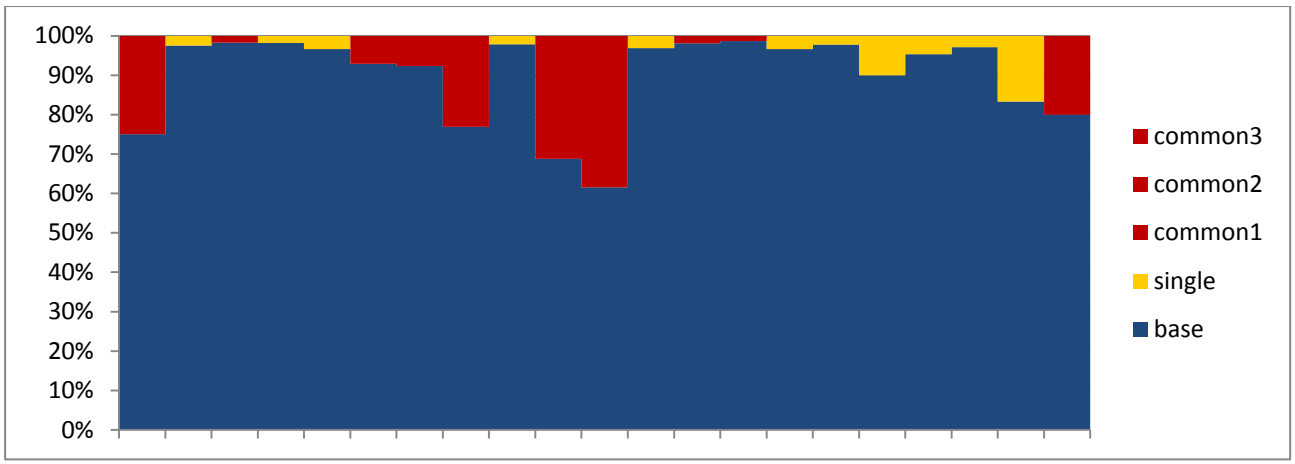

Figure 39: Modified Data Elements per Data Entity modified in subject 2-1.

6.3.4 Subject 2-2

Figure 40 depicts graphically the modification of Code Entities for subject 2-2. Figure 41 on the other hand represents the modifications done on the same subject, but on the Design Entities and finally Figure 42 presents the modifications done on Data Entities for the same subject.

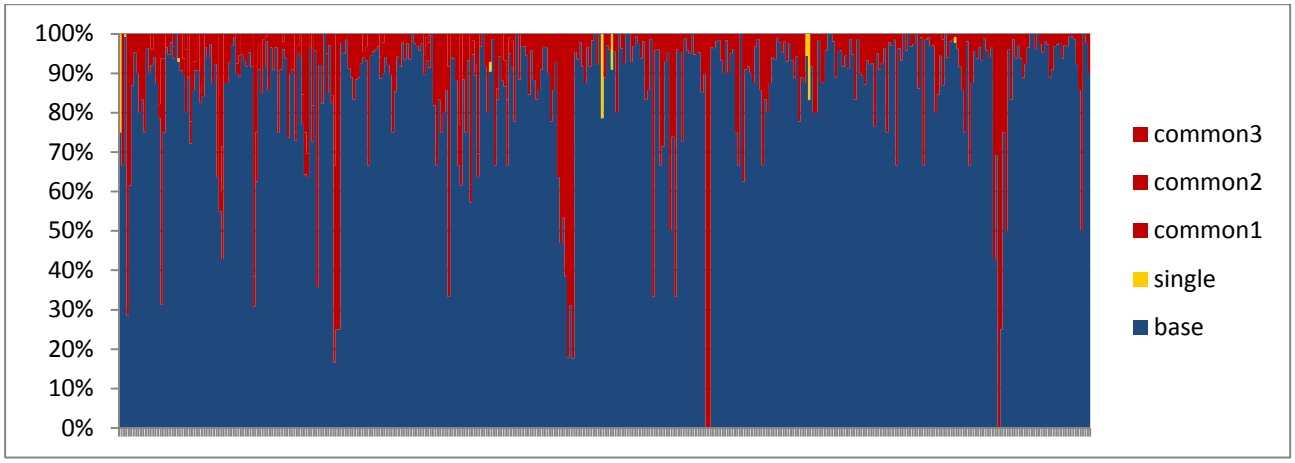

Figure 40: Modified Code Elements per Entity modified in subject 2-2.

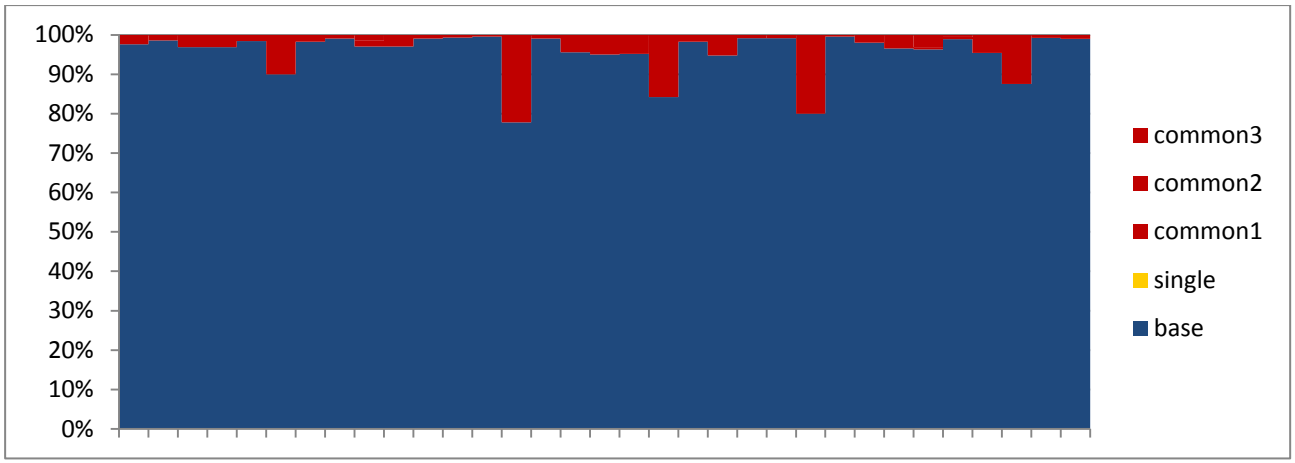

Figure 41: Modified Design Elements per Design Entity modified in subject 2-2.

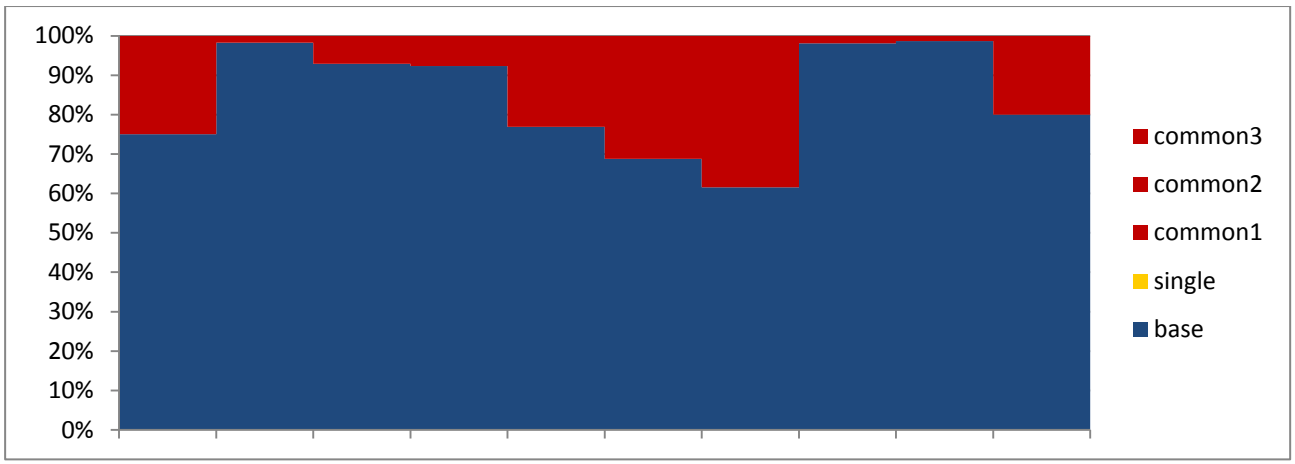

Figure 42: Modified Data Elements per Data Entity modified in subject 2-2.

6.3.5 Subject 2-3

Figure 43 depicts the modification of Code Entities for subject 4-3. On the other hand, Figure 44 represents the modifications done on the same subject, but on the Design Entities and finally Figure 45 presents the modifications done on Data Entities for subject 4-3 again.

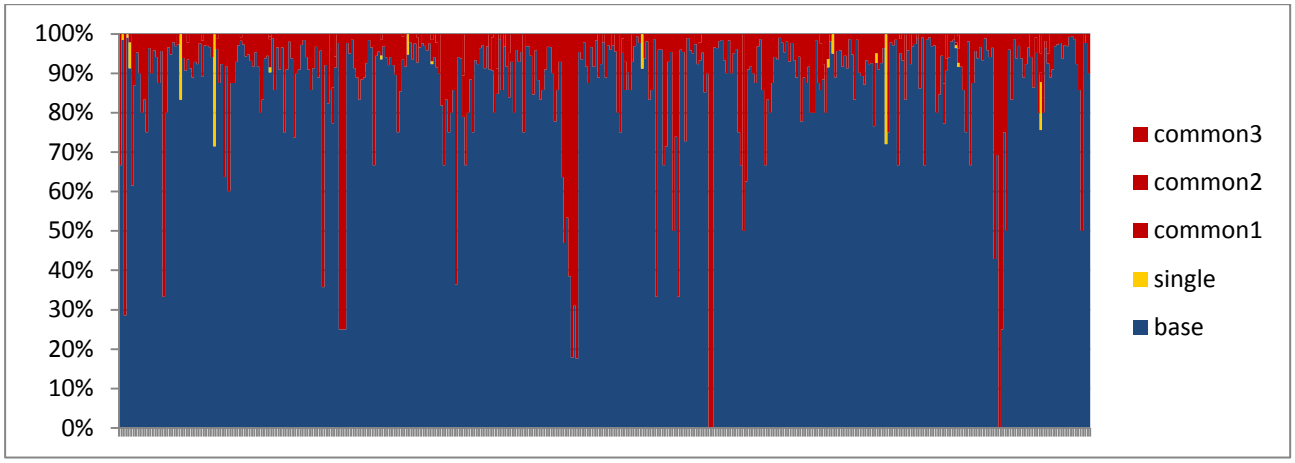

Figure 43: Modified Code Elements per Entity modified in subject 2-3.

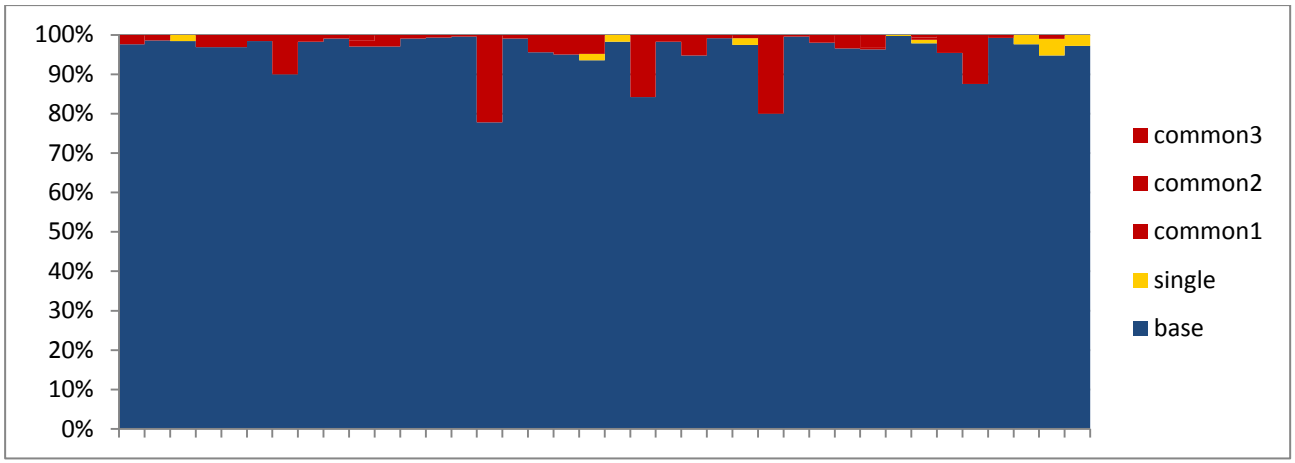

Figure 44: Modified Design Elements per Design Entity modified in subject 2-3.

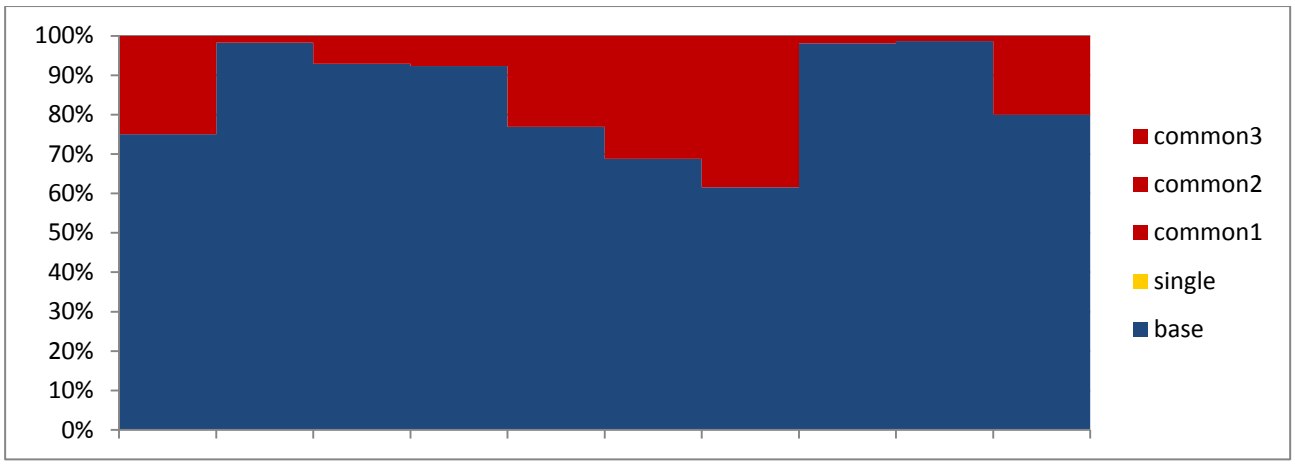

Figure 45: Modified Data Elements per Data Entity modified in subject 2-3.

From the above figures one might assume, that modifications on Entities of any type are fine grained and cross cutting. In other words, it seems that modifications of Entity Elements are equally distributed across the modified Entities and that these modifications comprise only a low percentage of all the existing Elements of an Entity. The following section now tries to investigate this question by a statistical analysis on the observed data.

6.4 Discussion

The following section presents a short statistical analysis on the above observations to investigate the questions raised in the previous sections; thus laying the foundation for the derivation of the hypotheses stated in section 7. The analysis is split into three major parts: first of all it takes a closer look on the relationship of additions and modification of Elements in Refinement Components. It follows an analysis of the degree of commonalities for the refined Elements. Finally the section concludes with a deeper treatment of the Element modified per Entity. For the avoidance of doubt, it should be noted that the author is aware of the fact that 5 samples are by no means statistically relevant, but as stated earlier, this text tries to reason inductive on base of a small sample to derive a set of hypothesis which are asked to be further tested. Hence the following analysis tries to only get an insight into the nature of the style and not to statistically proof some claims.

6.4.1 Addition vs. Modification

The following section presents an analysis of the different kinds of refinement. Additions of new Elements and modifications of existing Elements represent the two ways a Refinement component can adapt a Base component. The first impression from the earlier observation is that former kind of refinement, addition of new Elements, are multiple times higher than modifications of existing Elements. The second feeling is that there must be a kind of relationship, linear in nature, between the two kinds. It seems rational that an increasing number of new Elements require an increasing number of modifications to existing Elements in order to integrate the new Elements.

Figure 46 depicts graphically the average number of additions and modifications done to the Base component in order to refine it for the different domains. The error bars depicts the standard deviation to the mean, thus almost 70% of the values lie within these bars.

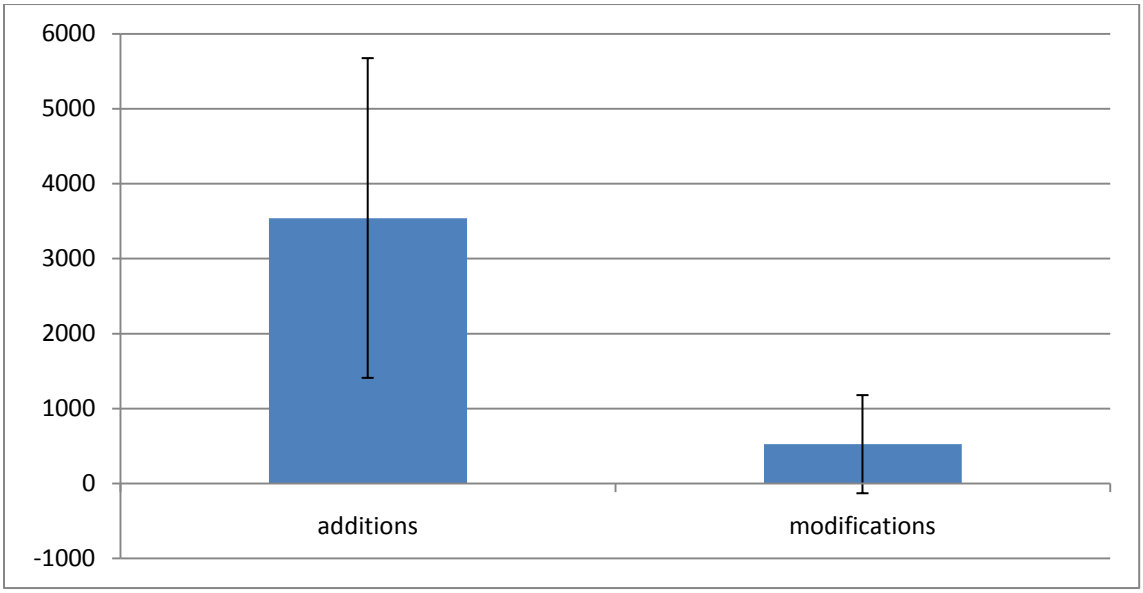

Figure 46: Average number of Elements added/modified by each subject with the corresponding standard deviation.

From the above figure the suspicion that the number of additions is multiple times higher than the number of modifications will be substantiated. Also if the error bars are quite wide, it seems that adding new Elements will be the dominant kind of refining a system while modifications will only help to integrate the new Elements.

Figure 47 engage into a possible relationship of the two kinds of refinements. The horizontal axis represents the number of new Elements, while the vertical axis represents the number of Elements modified. Each entry in the graph depicts the combination of modifications and additions done for the refinement of one type of one of the subjects. Furthermore the regression line is shown with the corresponding correlation coefficient.

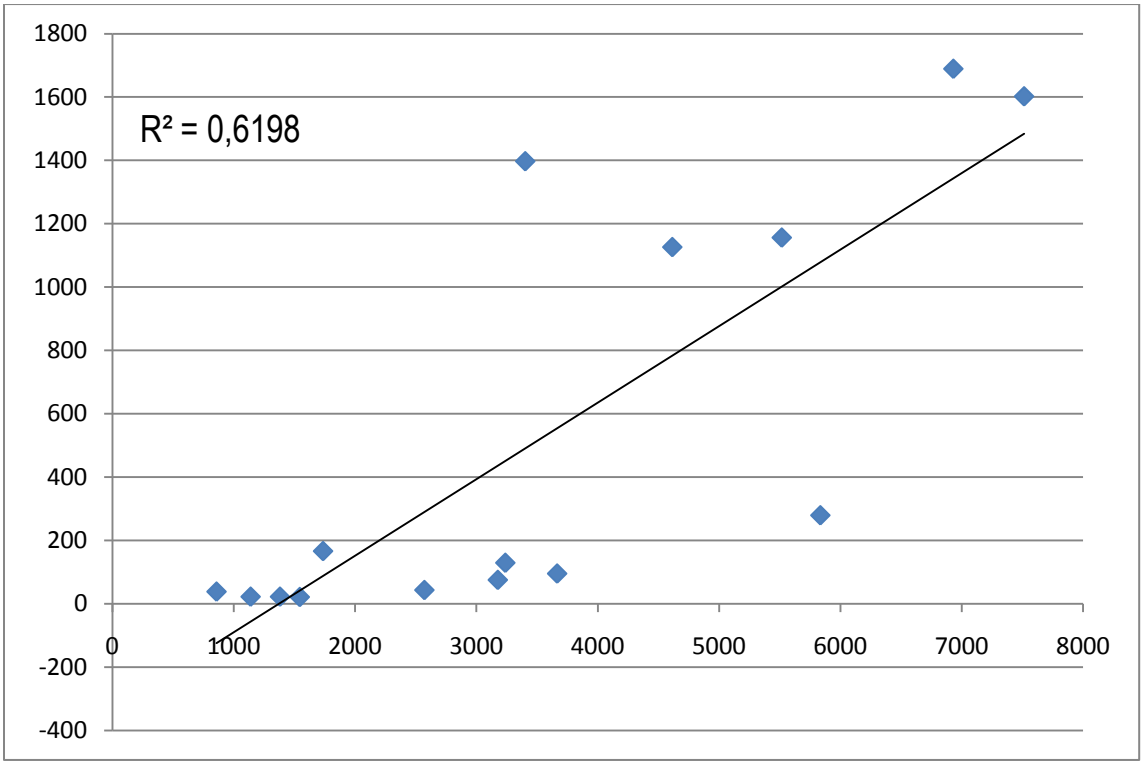

Figure 47: Relationship between the numbers of Elements modified and the number of Elements added for each type and each subject.

From the above figure, it seems that indeed there is a linear relationship between the number of additions and modifications performed for the refinement of one type of one subject. Also if a correlation coefficient of 0.62 is not sufficient to confirm statistically an eventual relationship, due to the low number of samples it could be an indicator for the existence of such a relationship.

6.4.2 Other Relationships

An interesting question which emerges from the above observation would be whether or not there exists a relationship between the number of additions and modifications respectively, and the reuse potential of such refinements. One might expect that the number of domains for which an addition or modification can be used decreases with the size of the set of refinements. Furthermore the question arises whether there is a significant difference in the number of modifications or additions respectively for different types of elements. From the observation it seems that the type of element has no significant impact on the number of additions or modifications respectively.

To investigate the above questions, Figure 48 depicts the number of additions and modifications of Elements grouped by the number of systems using these Elements. Furthermore an error bar depicts the standard deviation from the mean.

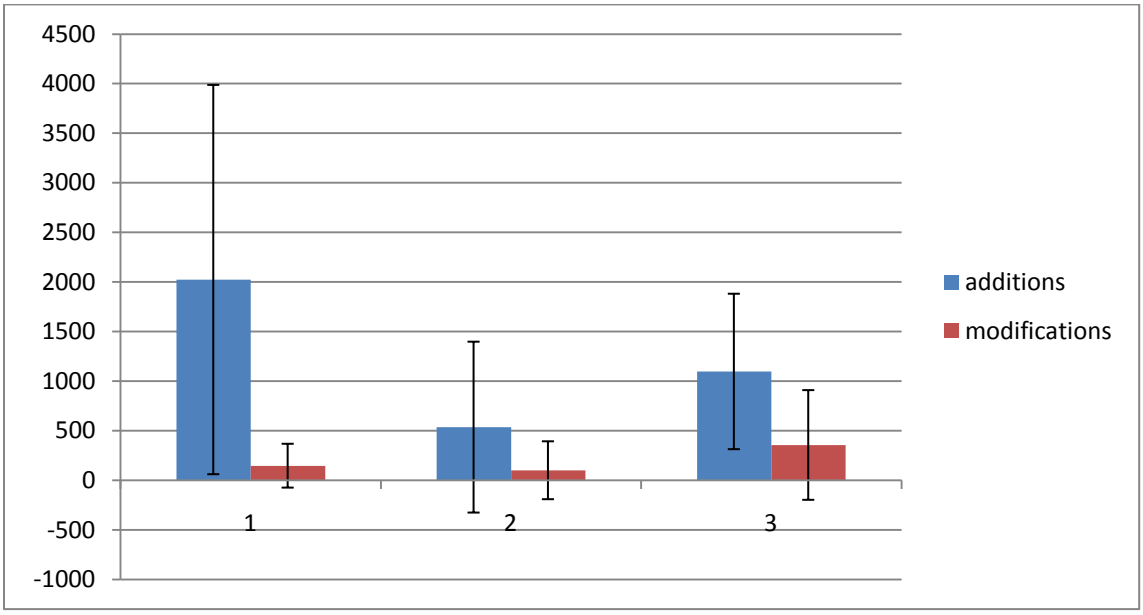

Figure 48: Number of added/modified Elements according to their reuse potential.

From the width of the error bars it seems that the distribution of the data from the observation is rather broad and therefore not considered significant for deriving any hypotheses.

Figure 49 depicts the number of new Elements added or Elements modified for each of the three types. Again an error bar shows the standard deviation to the average value.

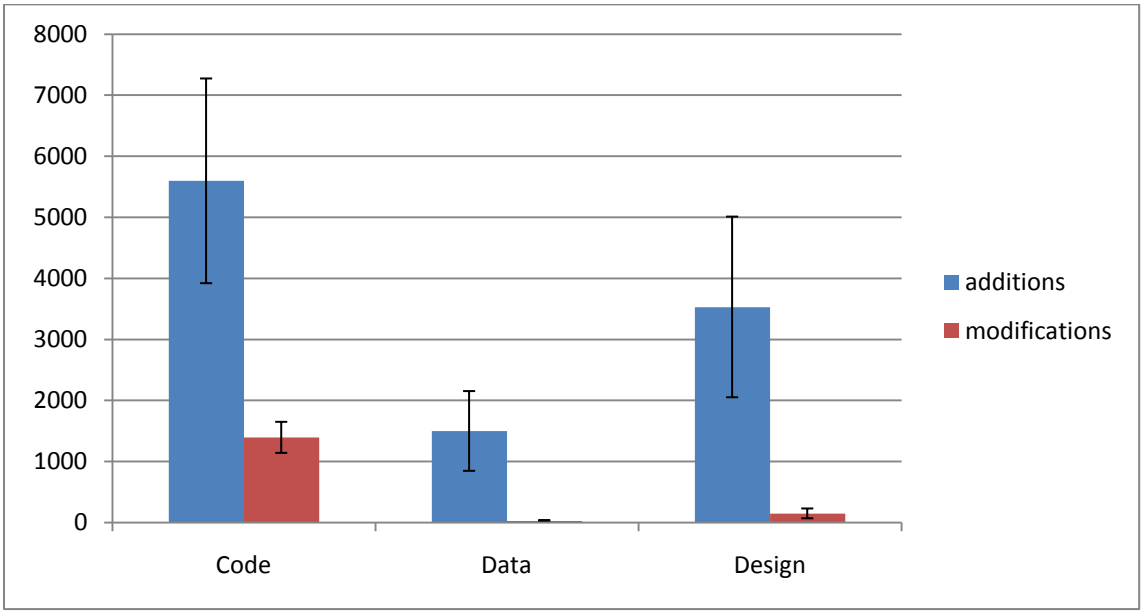

Figure 49: Number of added/modified Elements according to the Element type.

At a first impression the figure seems to suggest a significant difference in the number of additions and modifications respectively for different kind of Element types. But reflecting on section 6 one can see that in

both cases, Microsoft Dynamics AX 4 and Microsoft Dynamics AX 2009 over 50% of the Elements of the base component are Code Elements, on the other hand, 30% are Design Elements and only 10% comprise Data Elements. Hence, the differences uncovered by the above figure follows logically from the distribution of Element types in the Base component and do not reflect any causal relationship of Element type and number of additions or modification of Elements.

6.4.3 Modifications

Considering the observation presented in section 6.3 on the modifications of Elements for single Entities, it appears that the modifications done on an Entity are rather fine grained, that is, small in size and distributed equally among the modified Entities. Another interesting question arises from the observed data, namely whether there is a relationship between the number of added Elements and the percentage of modified Elements per Entity. It seems obvious that a high number of Elements added implies a high percentage of modifications to single Entities in order to integrate the new Elements.

Figure 50 shows the average percentage of modified Entities for each of the three types, grouped per subject. The error bar on each of the single bars depicts the standard deviation from the mean, i.e. the boundary capturing about 70% of the values.

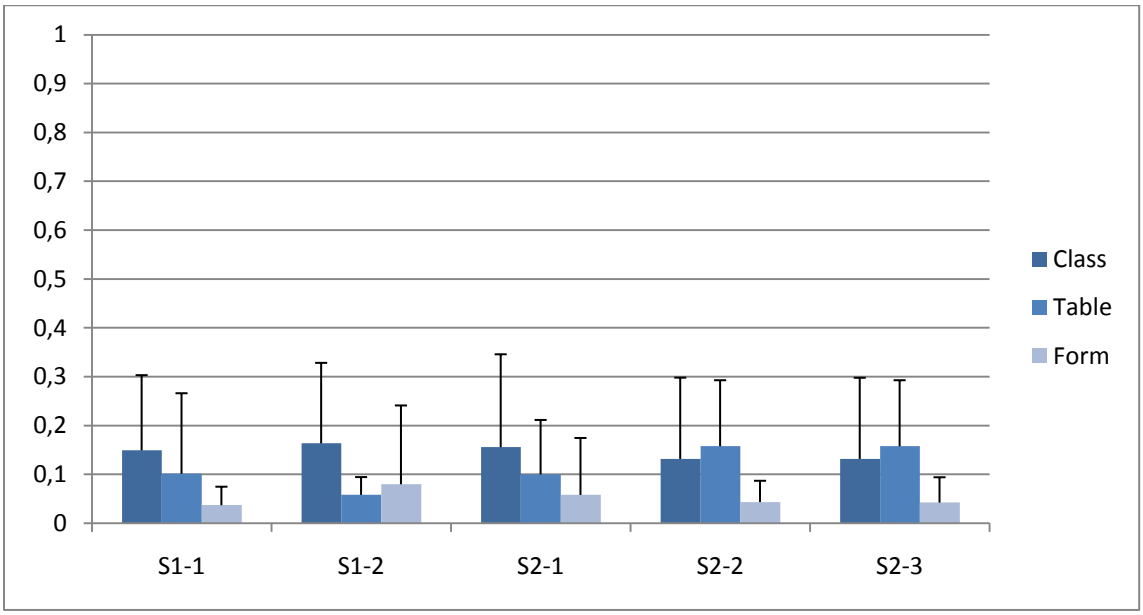

Figure 50: Modified Elements per Entity for each subject for each type.

Due to the high value of the standard deviation, the above figure seems to reject the assumption that the modifications are equally distributed among the modified Entities. Nevertheless the figure confirms the claim that the number of modified Elements per Entity is rather small, that is less than 50%.

In order to consider another view on the number of modified Elements per Entity, Figure 51 depicts the same data as Figure 50 in a different way. The percentage of modifications to existing Entities is represented as a boxplot where the boxes capture values from the lower to the upper quartiles and the whiskers shows the variations from the 0,025 Percentile up to the 0,975 Percentile.

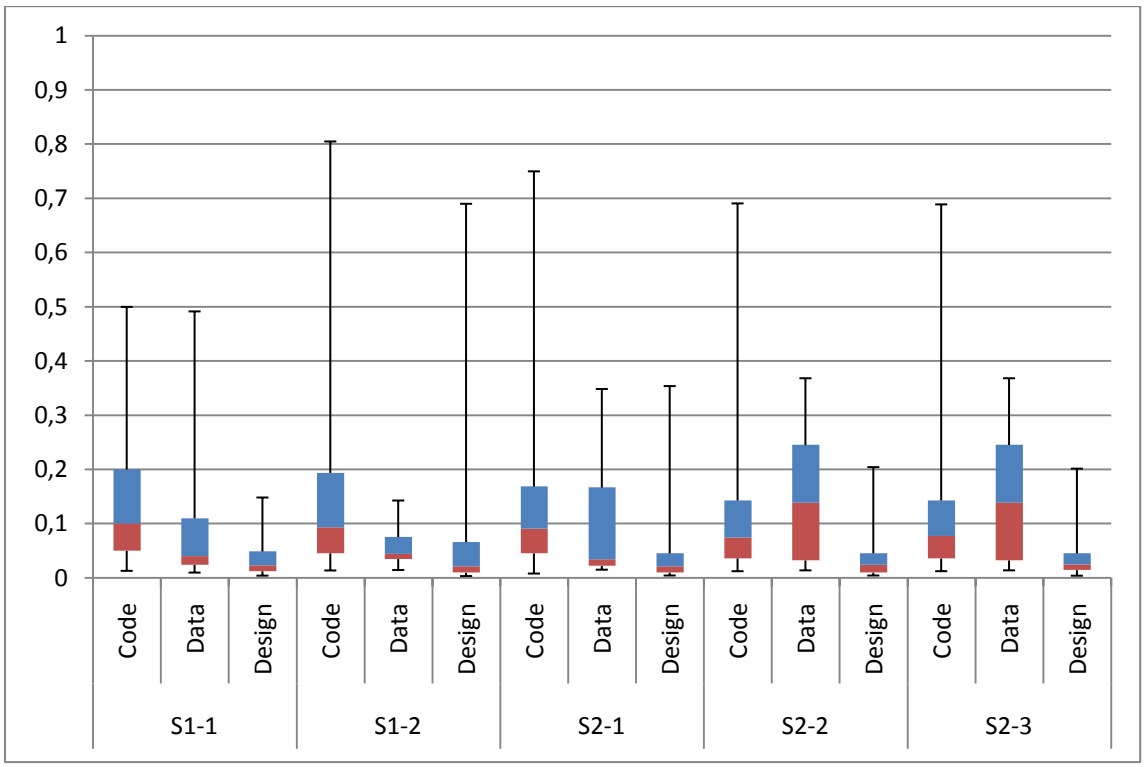

Figure 51: Modified Elements per Entity for each subject for each type as boxplot.

The above figure shows again a breadth distribution of the values, mainly for the upper 50%. Nevertheless all the boxes reside below the 0.25 boundary which confirm the assumption of a fine grained nature of the modifications to existing Entities.

In Figure 52 one can recognize the relationship of the number of added Elements and the number of Elements modified per Entity. The horizontal axis represents the percentage of modified Elements per Entity, while the vertical axis represents the number of added Elements. The entries in the graph depict the combinations of added Elements per subject and modified Elements per Entity. Furthermore the corresponding regression line is shown with the proper correlation coefficient.

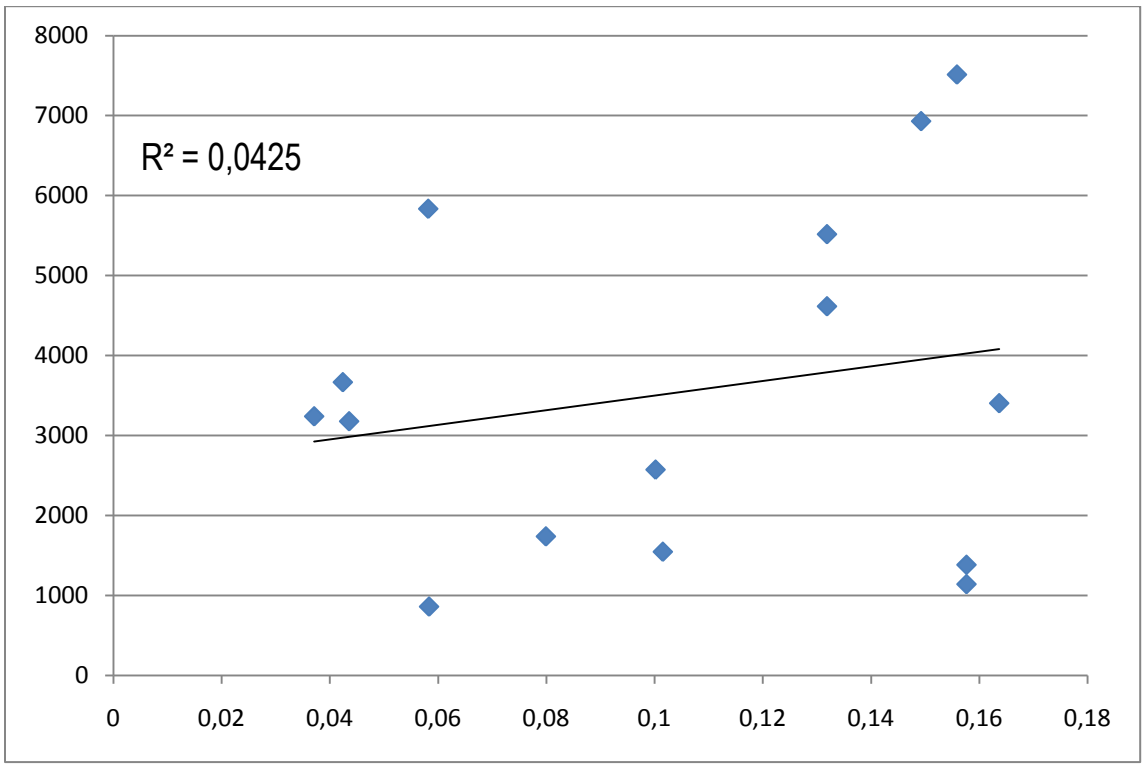

Figure 52: Relationship between the numbers of Elements added by each subject and the modifications to Entities of each type by the same subjects.

The above figure not only denies the existence of a relationship between the values, but with a correlation coefficient of 0.045 induces the assumption that there is no relationship at all. This might be an interesting observation since it contradicts the obvious claim that a high number of introduced Elements imply a likewise high number of modifications to existing Elements.

Again the author notes to be aware of the fact that such a small sample has no statistical significance, but as mentioned above this section should only lay the foundation to inductively derive a set of hypotheses. The following section therefore tries to state a set of hypotheses about the Partial Refinement architectural style founded in the analysis done in this section.

7 Hypothesis

The following section tries to leverage the insights obtained from section 6 to derive a set of related hypotheses on the properties elicited by the *Partial Refinement* architectural style defined in section 5.

In order to state the hypotheses, first a set L is defined as the set of all possible Entity Elements and E as the set of all possible Entities, where each $X \in E$ aggregates some of the elements of L. Furthermore, a bijective function $e: E \rightarrow P(L)$ is defined to capture this relationship between Entity and corresponding Elements. A Base component B can now be defined as an element of the powerset of all possible Entities E: $B \in P(E)$.

In order to refine a Base component, a Refinement component uses two kinds of mechanisms: *adding* new Elements by adding new Entities or enhancing existing Entities, and *modifying* existing Entities by modifying their Elements. Figure 53 depicts this situation graphically, where the wine red circle represents a Base component and the other circles Refinement components respectively.

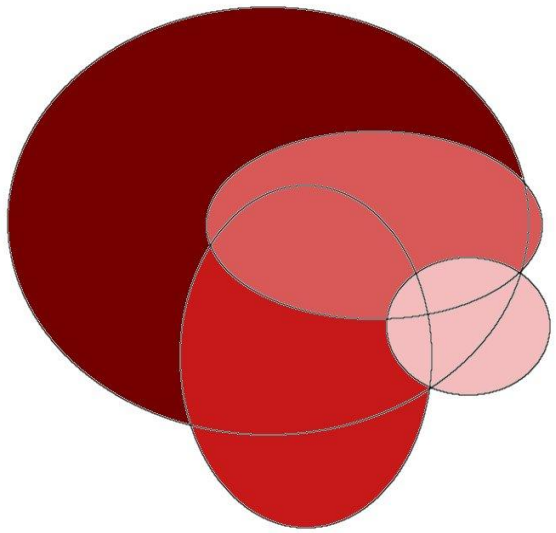

Figure 53: Refinement of a Base component by Refinement components.

As one can recognize from the above figure, a refined system is composed of three kinds of Entities: original, unaltered Entities of the Base component, Entities modified by Refinement components, and new Entities added by Refinement components. For this purpose, three functions should be defined as $P(E) \rightarrow P(E)$, to capture these relationships respectively: o as the function retaining all original Entities from the Base component; a as the function capturing new Entities added by Refinement components; and m as a function returning all Entities modified by Refinement components. Let us capture now the implementation of a system according to the Partial Refinement architectural style within a function i defined as $i: P(E) \rightarrow P(E)$ and composed of three distinct components:

$$i(B) = o(B) \cup a(B) \cup r\big(m(B)\big)$$

A conceptual overview of the function r used to refine the Entities modified by Refinement components is visualized in Figure 54. The wine red circle represents an original Element of the Entity, and the other circles modifications to this Element or additions of new Elements by Refinement components.

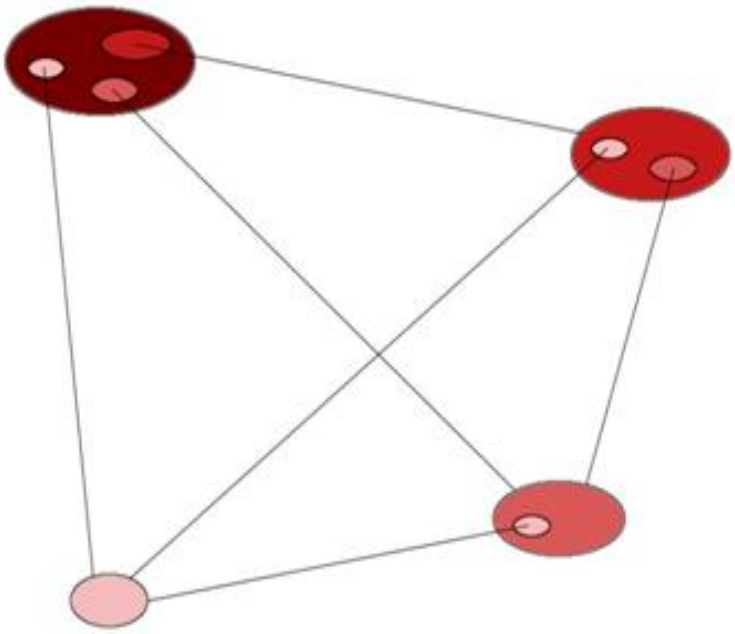

Figure 54: Modification of an Entity in the Base component by modifying existing and adding new Entity Elements.

As one can recognize from the figure above, function r refines base Entities by using two kinds of mechanisms: adding new Elements to an Entity or modifying existing Elements. Hence, function r can be defined as a mapping from $P(E)$ to another $P(E)$ as $r: P(E) \rightarrow P(E)$. The function can further be described as follows:

$$r(\mathrm{M}) = \{e^{-1}(oE(x) \cup mE(x) \cup aE(x)) | x \in M\}$$

In other words, function m applies functions oE, mE and aE respectively to each element x of M. Thereby, functions oE, mE and aE are defined as $E \rightarrow P(L)$ and capture original, modified and added Elements for each Entity respectively. Furthermore function r applies the inverse function for e to the union of the result sets of the functions oE, mE and aE to get the corresponding Entity comprising the new set of Elements. Finally, the obtained Entity is added to the result set of function m.

7.1 Hypothesis 1: Addition over Modification

From the analysis presented in section 6.4.1 it seems that refining the Base component requires far more additions of new Elements than modifications to existing Elements. This hypothesis can be captured in form of a relationship between some of the above functions:

$$\sum_{X \in m(B)} |aE(X)| + \sum_{X \in a(B)} |e(X)| \gg \sum_{X \in m(B)} |mE(X)|$$

7.2 Hypothesis 2: Relationship between Addition and Modification

Another assumption derived by the analysis in section 6.4.1 is that there exists a linear relationship between the number of added Elements and the number of Elements modified, which means that the number of Elements modified depends on the number of new Elements added. This hypothesis can be stated formally as:

$$\sum_{X \in m(B)} |aE(X)| + \sum_{X \in a(B)} |e(x)| = K * \sum_{X \in m(B)} |mE(x)| + Z$$

7.3 Hypothesis 3: Fine grained Modifications to existing Entities

This text further observed that only fine grained modifications to existing Entities are needed. Hence, the number of Elements modified for each Entity modified will reside below some Base-component-specific value k:

$$\bigvee_{X \in m(B)} (|mE(x)| < k(B))$$

Furthermore, k is defined as a function, mapping a Base component to a percentage value of modifications needed for an Entity; k is observed to be lower than 25% for 75% of the Entities.

7.4 Hypothesis 4: Constant Modifications to existing Entities

A last hypothesis is that there is no relationship between the number of added Elements and the percentage of modifications needed for an Entity. This means that independently on how many Elements will be added by a refinement, the modifications to existing Entities will remain as proposed by hypothesis 3. However, the increasing number of added Elements will be compensated by an increasing number of modified Entities as proposed in hypothesis 2. Consider therefore two refinements, with Base components B1 and B2 respectively, then this hypothesis can be stated formally as:

$$\sum_{X \in m(B1)} |aE(X)| + \sum_{X \in a(B1)} |e(X)| > \sum_{X \in m(B2)} |aE(X)| + \sum_{X \in m(B2)} |e(X)| \rightarrow |k(B1)| = |k(B2)|$$

The hypotheses presented in this section were derived by inductive reasoning on top of the empirical observations of the style in action. However, before the stated hypotheses become reliable knowledge about the elicited properties of the Partial Refinement architectural style, they require further testing. Hence the next section calls on future work testing the hypotheses and its predictions to finally corroborate them and eventually make them scientific facts.

8 Conclusions and Future Work

Driven by the research questions identified in section 1.1, this text began with an extensive literature review on software architecture, architectural styles, software adaptability, and software product families in section 3. In doing so, the text identified a lack of reliability in current approaches to investigate software architectural styles. Such styles are defined informally and elicited properties are derived intuitively, lacking any empirical evidence. Furthermore the review of current literature uncovered the absence of a formal definition and analysis of the architectural style driving the development of Microsoft Dynamics AX implementations.

Thus, the text presented the *Architectural Style Analysis Method* (ASAM) in section 4.1, a method derived by the standard scientific method [**1**] and adapted for investigating software architectural styles. ASAM is presented as a process of 5 distinct activities, leading to the acquisition of reliable knowledge about architectural styles. In section 4.2, ASAM was adapted for the purpose of investigating a new architectural style. Thus, in section 4.2.2.1 a tool was presented to collect raw data about the style in action. In section 4.2.2.2 a set of metrics were derived using the goal-question-metrics framework [**11**] to analyze the obtained raw data. Finally in section 4.2.2.3 another tool was presented to obtain measures of the defined metrics from the collected data.

Furthermore, ASAM is applied in the analysis of a new architectural style guiding the development of Microsoft Dynamics AX systems: the observed style was first defined formally as *Partial Refinement* architectural style in section 5 by applying the ALFA framework [**12**] and investigating closer the *Selector* connector as a crucial component of the style. It followed an empirical observation of the style in action on 5 Microsoft Dynamics AX [**13**] implementations in section 6 with a short statistical analysis of the obtained data in section 6.4. On top of that observations, a set of 4 related hypotheses are derived and presented formally in section 7.

However, the work could only address 3 of the 5 activities proposed by ASAM and it lacks the testing of the hypothesis and the development of a scientific theory relating the hypotheses in order to explain the nature of the Partial Refinement architectural style. Furthermore, ASAM should be applied to more architectural styles in order to obtain reliable knowledge of these styles, which can be used to reliably predict properties elicited by these styles.

9 Acknowledgements

The author wants to take the opportunity to express his sincere appreciation to all the people who have contributed to the development of this thesis.

First of all I want to thank Professor Barbara Russo for accompanying me throughout the work. She was a moving power in the development of the text and contributed a lot with its considerable academic expertise.

Other important contributors were my colleagues at Wuerth Phoenix, Thomas Lorenzi, Michael Walzl, and Hillbrand Klaus. Without their support in collecting the data for the empirical observation, the thesis wouldn't be as it is.

Further I want to thank all the people who have been appreciative of my absence during the development of this text. A big thank-you to mom, dad, and Julia for their help and support.

10 Works Cited

[1] Steven D. Schafersman. (1994) Free Inquiry - An Introduction to Science. [Online]. http://www.freeinquiry.com/intro-to-sci.html

[2] John W Creswell, *Research design. Qualitative, quantitative, and mixed methods approaches*. Thousand Oaks, California: Sage Publications, Inc, 2003.

[3] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy, *Software architecture. Foundations, theory, and practice*. Hoboken: John Wiley & Sons, Inc, 2010.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*.: Addison-Wesley Professional, 1994.

[5] Mary Shaw and David Garlan, *Software architecture. Perspectives on an emerging discipline*. Upper Saddle River, NJ: Prentice Hall, 1996.

[6] Barbara Hayes-Roth, Karl Pfleger, Philippe Lalanda, Philippe Morignot, and Marko Balabanovic, "A Domain- Specific Software Architecture for Adaptive Intelligent Systems," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 288-301, April 1995.

[7] Marcus Pollio Vitruvius, *De architectura. 10 Books.*, 25 BC.

[8] Dewayne Perry and Alexander Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT*, p. S. 40–52, 1992.

[9] Len Bass, Paul Clements, and Rick Kazman, *Software architecture in practice*, 2nd ed. Boston, Mass.: Addison-Wesley, 2008.

[10] Stewart Brand, *How Buildings Learn: What Happens After They're Built*.: Penguin, 1995.

[11] Victor R. Basili and M. David Weiss, "A Methodology for Collecting Valid Software," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, pp. 728-738, November 1984.

[12] Nikunj R Mehta and Nenad Medvidovic, "Composing architectural styles from architectural primitives," in *Foundations of Software Engineering*, Helsinki, Finland, 2003, pp. 347 - 350.

[13] Arthur Greef, Michael Fruergaard Pontoppidan, and Lars Dragheim Olsen, *Inside Microsoft Dynamics AX 4.0*. Redmond, Washington: Microsoft Press, 2006.

[14] Ian Sommerville, "Software requirements," in *Software Engineering*.: Addison-Wesley, 2007, ch. 6, pp. 117-141.

[15] Levy Yair and Ellis J Timothy, "A Systems Approach to Conduct an Effective Literature Review in Support of Information Systems Research," *Informing Science Journal*, vol. 9, p. 181–212, 2006.

[16] Benjamin S. Bloom, *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain*.: Addison Wesley Publishing Company, 1956.

[17] Edsger W Dijkstra, "go to statement considered harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147 - 148, March 1968.

[18] Edsger W. Dijkstra, "Structured programming," in *SOFTWARE ENGINEERING TECHNIQUES*, Rome, Italy, 1970, pp. 65-68.

[19] Edsger W. Dijkstra, "A Personal Perspective," in *Selected Writings on Computing*.: Springer, 1982, pp.

1-14.

[20] Edsger W. Dijkstra, "The structure of the “THE”-multiprogramming system," *Communications of the ACM*, vol. 11, no. 5, pp. 341-346, May 1968.

[21] Edsger W. Dijkstra, "On the role of scientific thought," in *Selected writings on computing: a personal perspective*. New York, NY: Springer, 1982, p. 60–66.

[22] David L Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, pp. 1053 - 1058, 1972.

[23] David L. Parnas, "On a ‘Buzzword’: Hierarchical Structure," in *IFIP Congress 1974*, Stockholm, 1974, pp. 336-339.

[24] David L Parnas, "On the design and development of program families," *IEEE Transaction on Software Engineering*, vol. SE-2, no. 1, pp. 1-9, 1976.

[25] David L. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Transaction on Software Engineering*, vol. SE-5, no. 2, pp. 128 - 138, March 1979.

[26] Isaac Newton, Letter to Robert Hooke, February 5, 1676, Available at http://en.wikiquote.org/w/index.php?title=Isaac_Newton&oldid=1047638.

[27] Software Engineering Institute. Getting Started - Find out how others define software architecture. [Online]. http://www.sei.cmu.edu/architecture/start/definitions.cfm

[28] IEEE Computer Society, Systems and software engineering - Recommended Practice for architectural description of software-intensive systems, July 15, 2007.

[29] Mary Shaw et al., "Abstractions for Software Architecture and Tools to Support Them," *IEEE Transaction on Software Engineering*, vol. 21, no. 4, pp. 314-335, 1995.

[30] Clemens Szyperski, *Component software: Beyond object oriented programming*.: Addison-Wesley Professional , 2002.

[31] Nikunj R Mehta, Nenad Medvidovic, and Sandeep Phadke, "Towards a Taxonomy of Software Connectors," in *International Conference on Software Engineering* , Limerick, Ireland, 2000, pp. 178 - 187.

[32] Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide*. MA: Addison-Wesley Professional, 1998, Addison-Wesley Object Technology Series.

[33] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer, "Specifying Distributed Software Architectures," in *Specifying Distributed Software Architectures*, Berlin, 1995, pp. 137-153.

[34] David C. Luckham and James Vera, "An event-based architecture definition language," *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717-734, 1995.

[35] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee, "The Koala component model for consumer electronics software," *IEEE Computer*, vol. 33, no. 3, pp. 78-85, 2000.

[36] Michael M. Gorlick and Rami R. Razouk, "Using Weaves for Software Construction and Analysis," in *Thirteenth International Conference on Software Engineering*, 1991, pp. 23-34.

[37] Peter H. Feiler, Bruce Lewis, and Stephen Vestal, "The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems

Engineering," in *RTAS 2003 Workshop on Model-Driven Embedded Systems*, Washington, DC, 2003.

[38] David Garlan, Robert T. Monroe, and David Wile, "ACME: An Architecture Description Interchange Language," in *CASCON*, Toronto, Ontario, Canada, 1997, pp. 169-183.

[39] John Spencer, "Architecture Description Markup Language(ADML): Creating an Open Market for IT Architecture Tools," The Open Group, White Paper 2000.

[40] Eric Dashofy, André van der Hoek, and Richard N. Taylor, "A Comprehensive Approach for the Development of Modular Software Architecture Description Languages," *ACM Transactions on Software Engineering and Methodology(TOSEM)*, vol. 14, no. 2, pp. 199-245, 2005.

[41] Philippe Kruchten, "Architectural Blueprints—The "4+1" View," *IEEE Software*, vol. 12, no. 6, pp. 42-50, November 1995.

[42] Christopher J. Jones, *Design Methods: seeds of human futures*. New York: John Wiley & Sons Ltd., 1970.

[43] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture*. Chichester, UK: John Wiley and Sons Ltd, 1996, vol. 1.

[44] Tom Gilb, *Software Metrics*.: Chartwell-Bratt, 1976.

[45] Kristin Braa and Richard Vidgen, "Interpretation, intervention, and reduction in the organizational laboratory: a framework for in-context information system research," *Accounting, Management and Information Technologies*, vol. 9, no. 1, p. 25–47, 1999.

[46] Eric M. Dashofy, "Supporting Stakeholder-Driven, Multi-View Software Architecture Modeling," University of California-Irvine, Irvine, CA, Ph.D. Thesis 2007.