Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

Yao Guo

MASTER THESIS

Handling Complexity in Organizational Modeling

Submitted on 8 June, 2015

Supervisor: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander University Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

_____

[CITY], [DATE]

License

_____

[CITY], [DATE]

Abstract

In information system development, the inherent complexity nature is transferred from the business domain to the software systems via the modeling phase. This thesis aims to handle the complexity issues in organizational modeling and the validation of this research is based on the resource visualization project at Audi AG.

In order to examine the complexity issues in organizational modeling, a distinctive division of complexity patterns (structure-consistent and structure-flexible) in organizations was created. Meanwhile, in order to find a better solution for this research topic, two modeling approaches, Object-Oriented Modeling (OOM) and Dynamic Object Model (DOM), were employed to model and implement the identical business case. This research also formulated an evaluation framework where cognitive complexity metrics were used to assess UML class diagrams and software complexity metrics were used to assess the source code. Quantitative analyses between complexity patterns and modeling approaches were conducted.

The results of the research indicate that compared to OOM, DOM demonstrated a complexity pattern independent characteristic. With DOM an application can adapt to both consistent and flexible organizational structures by only modifying configuration files without programming. With respect to the complexity and volume reduction effects in modeling and coding, DOM demonstrated far more effective performance beyond OOM, in all evaluation dimensions. In addition, DOM created more flexibility than OOM to adapt to the dynamic business in organizational modeling.

Zusammenfassung

In der Informationssystementwicklung wird die inheränte Komplexität aus dem Business-Bereich über die Modelierungsphase auf die Software-Systeme überführt. Diese Arbeit zielt darauf ab, die Komplexitätsprobleme der Modellierung zu beheben. Die Evaluation dieser Forschung basiert auf dem Ressourcen-Visualisierungsprojekt von der Audi AG.

Zur Analyse der Komplexitätsprobleme gibt es zwei Komplexitätsstrukturen, struktur-konsistent und struktur-flexibel. Um eine bessere Lösung für dieses Forschungsproblem zu finden wurden mit Objekt orientiertem Modeling und dem dynamischen Objekt Model zwei weitere Lösungsansätze zur Modellierung und Implementation verwendet. Diese Forschung formuliert zudem einen Evaluationsrahmen, in der kognitive Komplexitätsmetriken verwendet wurden, um die UML-Klassendiagramme und Softwarekomplexitätsmessdaten verwendet wurden um den Quellcode zu bewerten.Es wurde eine quantitative Analyse zwischen Komplexitätsstruktur und Modellierungsansätzen durchgeführt.

Bezüglich der Komplexität und der Menge von Reduktionseffekten bei der Modellierung und Codierung, konnte DOM in allen Bewertungsdimensionen eine weitaus effektivere Leistung nachweisen. Darüber hinaus bietet DOM mehr Flexibilität als OOM um auf eine dynamische Organisation angepasst werden zu können.

Acknowledgements

In full gratitude I would like to acknowledge the following individuals who encouraged, inspired, supported, assisted, and sacrificed themselves to help me during the study.

First of all I would like to express my gratitude to my supervisor Professor Dirk Riehle, for his remarkable support and guidance during this research. Before this thesis, his AMOS project provided me with the opportunity to work on the thesis related project and access to the industry. During the thesis, his care encouraged me to cope with the difficulties in my research and personal life.

I also sincerely appreciate Mr. Uwe Laemmle for his support and advice when I was working on the project at Audi. His attitude of quality, philosophy of design and leadership set an example and standard in my life and future career.

For all technical assistance and moral support, I would like to thank my friends, Kathrin König, Christian Mühlroth, Martin Gumbrecht and Jan-Philipp Stauffert in the AMOS project first, where we initiated the related project with excellent and memorable cooperation. I also would like to thank Philipp Eichhorn for the discussion and for his inspiration. Also I would like to thank Michael Mayer and Achille Salace for their help when I was working on the thesis at Audi.

I would like to thank my family and especially my parents for all their love and support in my life, even if I decided to leave a stable life in Japan for the adventure in Germany. Their love and beliefs on me give me the best strength to cope with difficulties. Lastly, I would like to thank my best friend, Kevin Burley, for his help at any time in so many years and his mental and financial support during my studies in Germany.

Nürnberg, May, 2015

Yao Guo

Table of contents

List of figures

List of tables

List of abbreviations

IS	Information systems
ISD	Information system development
OOM	Object oriented modeling
DOM	Dynamic object model
UML	Unified Modeling Language
CC	Cognitive complexity
SC	Structure-consistent
SF	Structure-flexible
SWC	Software complexity
DAO	Data access object
JSON	JavaScript Object Notation

1 INTRODUTION

1.1 Motivation

In information system development (ISD), requirements and analysis models are built to represent the business and problem domains before implementation. The transformation from the user's business domain to an abstract design of software magnifies the complexity inherent in system development (Pressman, 2004). The complexity of models can ultimately bring a significant influence to the quality of the software implementation, in terms of the understandability of models for the business client and technical developers, efforts investment during implementation and flexibility for the modification of application in maintenance and so on. However, the relationships between the complexity of models (especially in organizational modeling) and the inherent complexity in the resulting software are still ambiguous and not well examined or handled. It also lacks a systematic evaluation framework for modeling artifacts.

In addition, there are a number of modeling approaches in practice. With traditional Object-Oriented Modeling (OOM), systems have little flexibility to adapt to the changes of businesses, since the classes are designed for the different types of business entities and the associate attributes and methods with them (Joseph & Johnson, 2002). The business and system are conjoined in a fixed relationship in the form of a static object model (Johnson, 1998). Due to this drawback, the business complexity (structural and functional) is migrated and directly reflected through to the software architecture with tight-coupling during modeling, which planted the quality concerns in complexity, extensibility and maintainability. Especially in hierarchical organizations, the fixed affiliation relationships in business are hard-coded in the programs, where the violation of separation of concerns emerged.

In order to enable the flexibility of software systems to adapt to the dynamics of a business, after OOM, Adaptive/Dynamic Object Model (DOM) was developed in recent decades, which provides more flexibility along with complexity.

This thesis aims to handle the complexity issues in organizational modeling from both business and technical perspectives, by investigating the complexity issues in organizational modeling, presenting a new vision of complexity issues with Complexity Patterns (CP), as well as a complexity measure framework with Cognitive Complexity (CC) metrics for models and Software Complexity (SWC) metrics for the resulting software artifacts. In order to provide a quantitatively comparable foundation for the evaluation framework, OOM and DOM are applied to the implementation of the identical business case from Audi AG. The quantitative analysis demonstrates how modeling approaches influence the complexity and flexibility in organizational modeling.

1.2 Research question

This thesis answers the research question of how to handle the complexity in organizational modeling.

Since the complexity issues are transferred into IS during the early phase, the modeling phase, this requires that the complexity issues are examined from both business and technical perspectives, to clarify the complexity patterns in organizations, relationships between modeling complexity and software complexity, and also to show the benefits (flexibility) to the end users, while employing different modeling approaches: Object-Oriented Modeling (OOM) and Adaptive/Dynamic Object Model (DOM).

1.3 Scope of work

Chapter 2 is designed to build the theoretical foundation, clear the confusion of addressing complexity issues in organizational modeling, create a division of complexity patterns to represent the problem domains in organizational modeling and formulate an evaluation framework for cognitive complexity in models and software complexity in the resulting software artifacts.

Chapter 3 introduces a case abstraction of the Audi project, Resource Visualization. The functionality for a single module is abstracted. The modeling and implementation scopes are restricted to those functionality definitions.

Chapter 4 introduces two modeling approaches employed in this research: Object-Oriented Modeling (OOM) and Dynamic Object Model with Type Object (DOM).

Chapter 5 presents the modeling and implementation results employing OOM and DOM approaches to the identical business case adapted to the complexity patterns.

Chapter 6 analyzes the modeling and implantation results employing OOM and DOM and presents quantitative analysis for those two approaches adapted to complexity patterns. Furthermore, it discusses the relationships between model complexity and software complexity while applying different modeling approaches toward complexity patterns.

Chapter 7 summarizes the thesis by presenting the findings, limitations and directions for future work.

2 COMPLEXITY IN INFORMATION SYSTEMS

Information systems development (ISD) is constantly emphasized for its inherent attribute of complexity, since it involves not only various technical issues but also the organizational factors from the business domain (Xia & Lee, 2003; Scott & Vessey, 2002). The studies from organization science, management science and computer science have defined individual principles and evaluation methods of complexity issues, which illustrated how they can enrich IS thinking (Mitleton-Kelly & Land, 2004).

In this research, the analysis foundation of complexity issues in organizational modeling consists of three steps: (1) clarify the confusions of complexity issues in relevant areas and narrow the research focus; (2) Summarize a division of complexity issues in organizational modeling; (3) Formulate a measurement of complexity in organizational modeling

This chapter starts with an extended spectrum of complexity issues in disciplines (section 2.1.1), outlining the relevant contributions to complexity issues in each domain, made by organization science and IS management, highlighting some their own key principles and methods for addressing complexity to build a theoretical reference, and defining the research scope of complexity issues in the study. Along with that, the taxonomy of complexity (section 2.1.2) in IS was created with complexity issues in the business domain and technical implementation. Furthermore, after investigating those, a division of complexity patterns (section 2.1.3) in organizational modeling is created to represent the problem domains and employed as a methodological basis to develop the topic of handling complexity in organizational modeling. Lastly, the points from the investigation of previous sections are involved in formulating an evaluation framework (section 2.1.4) for cognitive complexity in models and software complexity in the resulting software artifacts.

2.1 Complexity in Disciplines

This section is unfolded, with a gradual path of theoretical foundation exploration, from organization science, IS management, and the taxonomy of Complexity in IS.

2.1.1 Complexity in organization science

Complexity and systems

In organization science, the studies developed complex/nonlinear models, applying Complexity theory and Chaos theory to address the adaption of dynamics in social and scientific environments (Anderson, 1999; Daft, 1992). Mitleton-Kelly (2003), introduced the ten generic principles of complex evolving systems for developing a theory of complex social systems, which incorporated the work on Complex Adaptive Systems (CAS). Organizations can be viewed as complex adaptive systems (CAS), since they fulfill the fundamental CAS

principles: self-organization, complexity, nonlinearity, emergence, interdependency, co-evolution, chaos, and self-similarity (Macmillan, 2004). CAS tends to form a decompounded hierarchy, where the elements are loosely coupled with one another (Simon 1996; Anderson, 1999).

Complex behavior emerges from the intricate inter-twining, or inter-connectivity, of elements within a system and between a system and its environment (Mitleton-Kelly, 2003). CAS distinguishes itself from the general concept of complexity (complication) with nonlinearity, emergence and dynamics, etc. features applying Complexity and Chaos theory.

Complexity and organization dimensions

Some studies have provided a closer perspective of complexity issues in organizations. Complexity is also viewed as a dimension of organizational design (Claver-Cortés et al, 2012). The degree of horizontal, vertical and spatial differentiation indicates the level of complexity of an organization (Burton & Obel, 2004; Fredrickson, 1986; Robbins, 1990; Claver-Cortés et al, 2012). The horizontal differentiation indicates the numeric width of the division of functional units in an organization; the vertical differentiation indicates the depth of functional units in an organization in terms of lower diversity and a higher degree of centralization of functional units; spatial differentiation gives the autonomy and flexibility to different organization units (Claver-Cortés et al, 2012).

The above in-depth understandings of organization create a guide to view organizational complexity issues in the case of IS. The following facts can be acknowledged and mapped for the utilization in the IS domain, as summarized in Table 2-1.

Table 2-1 Mapping Complexity in Organization Science and IS Development

Organization Science	IS Development
Constant dynamics in environments	Modeling the dynamic changes in environments
Organizations as complex adaptive systems	Models to enable the flexibility for adaptations
Hierarchy decomposition and loose coupling	Decomposed hierarchies, loose coupling and reflective architecture
horizontal, vertical and spatial differentiation of the level of complexity of an organization	Complexity in IS can be viewed with three dimensions: width, depth and extensibility

2.1.2 Complexity in IS management

In IS management, there are a great number of studies focusing on the interdisciplinary research across management science and IS. Their focus spread over organizational management in IS projects, ISD quality management and IS design.

Land (1999), together with Mitleton-Kelly (2004), viewed an IT project and its business ecosystem as a complex system and applied Complexity theory to observe how IS can be understood and improved with the principles (co-evolution, emergence, self-organization, space of legacy) of Complexity theory, from an IS management perspective.

Lee and Xia (2002) defined the taxonomy of ISD project complexity with a broader spectrum involving the nature of complexity and locus of complexity. They formulated the measurement framework and its dimensions for ISD projects complexity, from the macro perspective of IS project management (as presented in the Figure 2-1).

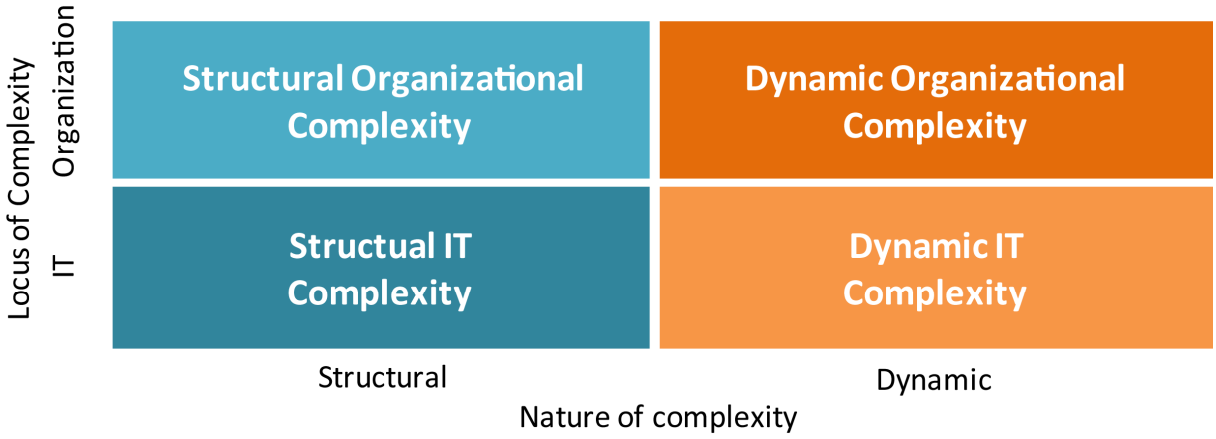


Figure 2-1 A Conceptual Framework for ISD Project Complexity

Mitleton-Kelly and Land (2004) pointed out that complex behavior arises due to (A) high connectivity and interdependency and (B) multidimensionality; interconnected elements contributed to the complexity of the system: (1) emergent properties, (2) organizational restructuring, (3) interdependency, (4) less support for maintenance and further development, (5) space of legacy (gaps between the business' needs and technical capabilities); (6) inherent problems in the relationship between IT development and business strategy can be reduced during their co-evolution.

Benbya and McKelvey (2006) also attempted to bring the Complexity theory and Chaos theory into IS design and defined a conceptual model as well as principles to explaining "how co-evolutionary adaptations of IS design with changing user requirements will result in more effective system design and operation." Mitleton-Kelly and Benbya's observations also occurred in this research.

Botchkarev and Finnigan (2014) summarized the complexity attributes in aspects of structural, technological, organizational, project management, uncertainty, ambiguity, end-users, dynamics, constraints, etc. Botchkarev also contributed a project as a system of systems for complexity mapping and complexity frameworks. Kluth et al. (2014), introduced an advanced complexity management model, Stuttgart Complexity Model, including four complexity dimensions in IS projects. Lu et al. (2014), illustrated a quantitative evaluation framework for an IS project complexity measurement.

After reviewing those studies, the methodological principles from those studies were selected and introduced in this research, for instance, complexity patterns for ISD projects and complexity metrics, as well as the evaluation framework for further development in Section 2.2 and Section 2.3, respectively. The following selected aspects from IS management can be learned and mapped for utilization in the ISD domain, as summarized in Table 2-2.

Table 2-2 Mapping Complexity in IS Management and IS Development

IS Management	IS Development
Connectivity and interdependency	Dependency and relationship (de)construction
Structuring	Flexibility in system architecture design
Emergent properties	Dynamics in the system for emergence
Maintenance support	Extensibility in system design
Taxonomy and abstraction of issues	Taxonomy of complexity patterns in IS
Complexity dimensions	Modeling along the division of dimensions
Measurement of solutions	Measurement metrics of models and software

2.1.3 Taxonomy of Complexity in IS

As described by Pressman (2004) software design is a process where requirements are translated into a representation in the form of software. The translation process from user requirements to software also magnifies the complexity inherent in systems development (Benbya & McKelvey, 2006). The interaction chaos between those two parties manifests that organizational factors are more influential than technical issues. Also as defined by Baccarini (1996), the complexity issues in ISD projects can be classified into two categories, organizational complexity and technological complexity. Similarly, Bruegge and Dutoit (2010), named this division as the application domain (representing all aspects of the user's problem and entities of the environment) and solution domain (representing the system design and object design activities of the development process).

This taxonomy provided by Lee and Xia (2002) (see Figure 2-1) was validated via a statistical questionnaire on ISD project experience from a macro (industry, scale, budget, duration, etc.) project management perspective, rather than conducting examinations on specific internal factors (entities, relationships, collaborations, means of modeling, software artifacts, etc.) that influence the internal complexity of a software product, from a micro perspective.

Therefore, in the context of the complexity of organizational modeling, this section contributes to presenting a taxonomy of complexity in the business domain and technical implementation from a micro perspective of an IS project by examining their individual

inherent features with regard to complexity issues in IS. In fact, it is more precise and suitable to summarize the nature of complexity described by Lee and Xia (2002) with complexity patterns for organizational modeling, which are introduced in Section 2.2. Meanwhile, the factors of complexity in the business domain and technical implementation will be used as fundamental elements for constructing the complexity measure framework in Section 2.3.

2.1.3.1 Complexity in the business domain

As one side of the complexity in IS, the business complexity is considered, consisting of product, organizational and process complexity. These categories can be reflected in the IS in several ways: the complexity of products and services affects the product master data; the organizational structure complexity impacts the enterprise architectural design, and the level of process standardization influences the data processing flow, as illustrated in Figure 2-2(Dijke & Scheele, 2013).

Specifically, product complexity provides variants of inputs to the IS, which are maintained as data foundations associated with database design, transactional processes design, etc. The business environment changes over time and the business adapts itself to the actual environments by raising new business requirements. This (sometimes) requires massive modification of the application under critical conditions, for instance, time urgency, budgets, down-time restrictions, etc.

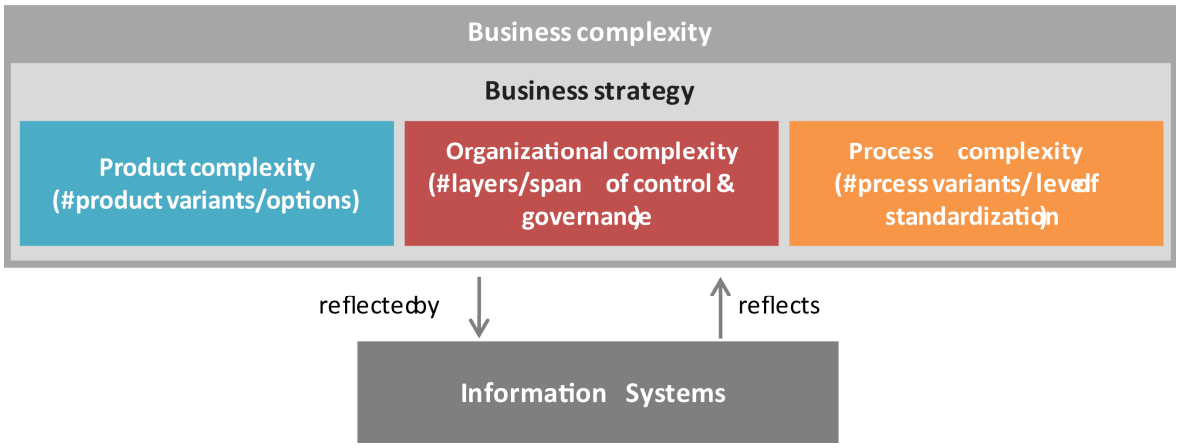


Figure 2-2 Mapping of business complexity and IS (Dijke & Scheele, 2013)

Organizational complexity contributes the framework of a business with its layer structure, functional span of control as well as dataflow, and the governance principles and business logic. In most of the cases, the organizations are formed with hierarchical layers, bureaucratic or flat. This indicates that organizational affiliation relationships are formed in settled interconnection, which implies little flexibility in adapting to the business change.

Process complexity creates process variants and resulting functional modules. Processes are designed independently to address specific process control requirements. However,

unrewarded or non-lean processes are also created with attached and redundant resources. As opposed to standardization, the harmonization of processes, providing standardized solutions across business units, makes the business units reuse common resources for each process and product variant.

Complexity drivers in the business domain

With regard to complexity in the business domain, this research adopts the major drivers observed in previous studies, closely relevant to the core topic, a rather full coverage of all factors, although more is reported by Schwandt (2009). For a clear understanding, the above suggests the following main complexity drivers, in organization-driven complexity:

(1) Diversification of target business domains

(2) Organizational structure

(3) Scale of given standardized processes

The business domains can be mapped to the domains or classes in models or programs, which require a lot of diversity on business requirements. The organizational structure (see Appendix A) influences the business logic in system architecture design and the correlations of domains. The standardization of processes indicates the resource allocation and collaboration in the system. Further mapping technical complexity and complexity measures is conducted in the later sections. In addition, hierarchical organizational structure is adopted as the context in this research for any further discussion.

2.1.3.2 Complexity in technical development

Costs, flexibility and quality are usually recognized as the KPIs of IT performance management, however, a key hidden variable is neglected – IT complexity. The unintentional neglect occurred is so often due to the difficulty of mastering the IT complexity, since a valid all-in-one measure for IT Complexity does not exist (Leukert et al, 2011).

Therefore, customized evaluation criteria for the complexity in technical development should be adopted with empirical reports from the IT domains working with organizational modeling. Here, a practical division of those criteria from the financial industry (the validity stands due to its large scope of organizational characteristics) is employed to present the complexity issues in technical development (Leukert et al, 2011):

(1) Data, associated with logical and physical data objects

(2) Functions, associated with functionality or the business and process logic

(3) Interfaces, associated with standardized interoperability between the IT assets

In addition, in state-of-the-art development, systems are designed on top of system architecture and supported with extensible frameworks, which influence the complexity during implementation. For instance, in a web application, MVC architecture frameworks are in the center of process control, object creation, page forwarding, and data persistency and so on.

(4) Architecture, associated with the collaborations between data/objects, functions and interfaces

The above suggests corresponding mappings with business complexity, as illustrated in Figure 2-3. This clearly demonstrated that the inherent complexity nature in IS derives from the business and problem domains, in which organizational complexity is at the position of gathering products and processes for serving itself. The further reflection of business complexity and technical complexity in this research will be introduced in the Section 3.2 case abstraction for the research topic. Apart from those, obviously employed programming languages, persistence techniques and so on, also influence the technical complexity. However, those factors are out of this discussion scope.

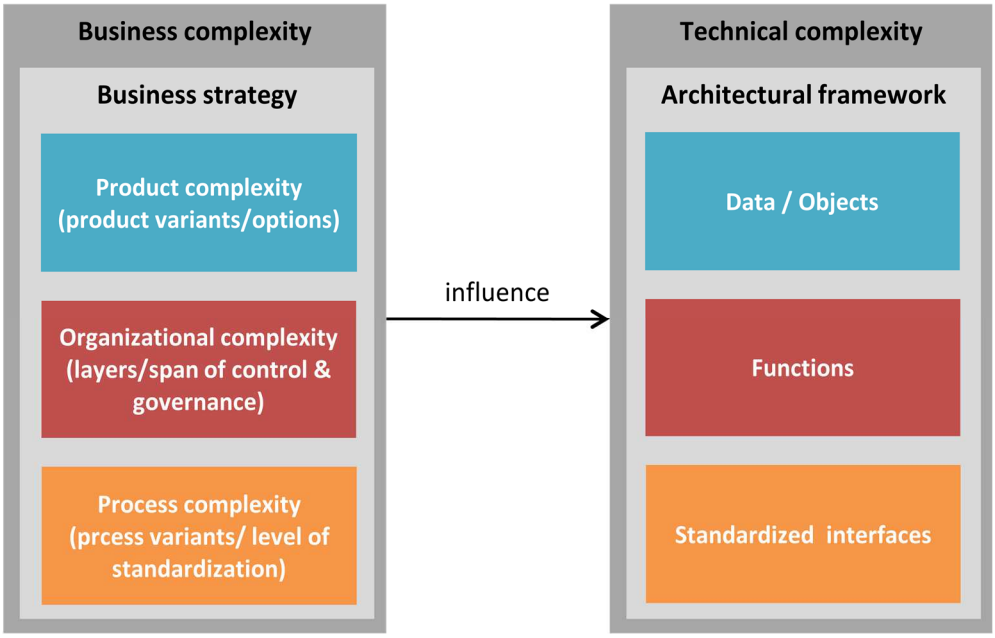


Figure 2-3 Mapping between business complexity and technical complexity

Furthermore, the complexity in technical implementation rests on two aspects: complexity in the presentation of design models and the resulting software product. In the standard software development life cycle, requirement analysis and modeling with Domain Specific Language (DSL), especially with Unified Modeling Language (UML), is conducted in its early phase. After modeling, the developers promote the development process by concretizing the conceptual understanding of business entities, relationships, and collaborations into source code with specific programming language. The business complexity along with its design quality, represented by UML, is usually embedded into the source code during the transformation phase.

2.1.4 Discussion context of the complexity topic in this research

There are some debates about using the terminology of "complexity" in IS. Regardless of correctness, it is used in some specific domains in particular contexts. According to the primitive definition in organizational science, the "complexity" along with Chaos theory indicates that systems interact dynamically with the external environments and evolve and adapt correspondingly. In IS management, some phenomena manifest on the macro system compositional and architectural level. Those studies provided some awareness of the circumstances where the systems are created as a guide for addressing complexity issues in IS on a higher level, to recognize the patterns of complexity in real life.

However, in particular, the complexity in the modeling phase is mostly recognized as the difficulty to understand the class diagrams, with regard to association, aggregation, generalization and dependencies (Genero et al, 2000). Therefore, this research narrows the focus down to the cognitive structural complexity in organizational modeling and the complexity in the resulting experimental software artifacts, as well as their correlations, with quantitative and qualitative measures.

Cognitive structural complexity for a model refers to the difficulty for a human being to understand this model. The conceptual models are presented with Unified Modeling Language (UML) and their complexity is measured with defined UML complexity metrics. The technical implementation is conducted with Java language and consistent programming styles as well as design patterns, which only differ according to the target modeling approaches and complexity patterns in combination. The software complexity is evaluated with software complexity metrics and analysis tools. The complexity topic development flow is illustrated in the Figure 2-4.

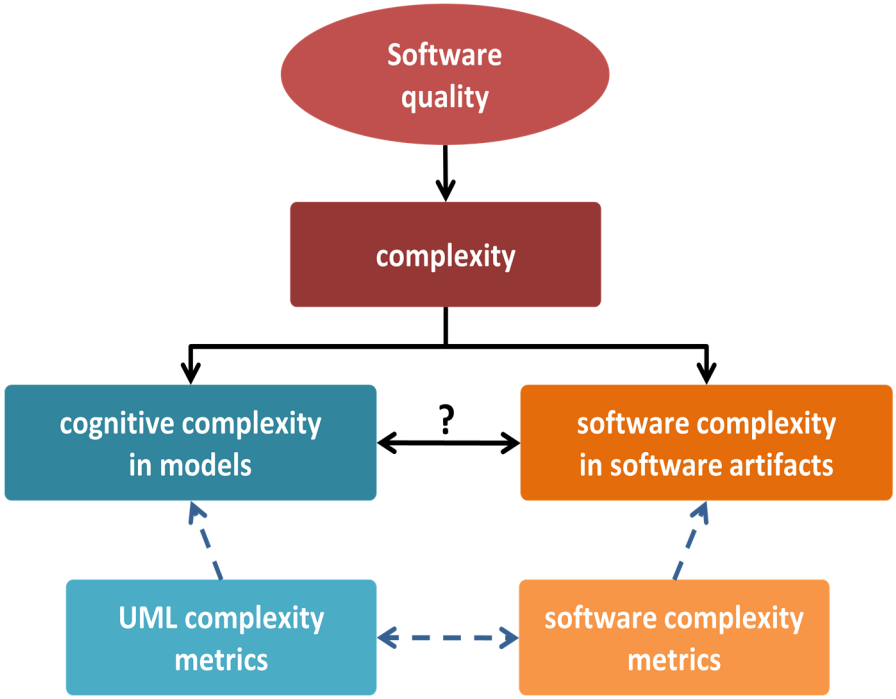


Figure 2-4 Complexity topic development flow

2.2 Complexity Patterns in Organizational Modeling

Organizational complexity contributes the framework of business with its layer structure, functional span of control, dataflow, and the governance principles and business logic. Organizational structure defines how tasks are divided, grouped, and coordinated in organizations (Newstrom, 2014). It is the foundation for various functional settings, the component for relationship mapping, and the handler of attached responsibilities.

The uncertainty and dynamics of the organizational structure in the practical project at Audi AG was one of the reasons to promote a division of organizational structures along with corresponding functional designs. A division of complexity patterns in organizational modeling was designed to represent the problem domains with a structure-consistent pattern and a structure-flexible pattern. Each pattern can find similar instances for specific business functions in real life.

2.2.1 Structure-consistent pattern

Structure-consistent pattern denotes that the organization structure is extended with an identical duplication of the vertical hierarchical structure on any horizontal branches.

Consistent organizational structure (traditional organizational structure) used to be recognized as well-organized structure, vertically structured and characterized by distinct job classifications and top-down authority structures. However, it is also denounced by its bureaucratism and low efficiency. It is critical in industries, whose business requires stable and massive infrastructure investment and target markets on the geographical scope. The organizational affiliation relationships are formed in settled interconnections, which imply less flexibility requirements to adapt to the business change over time. Only a full duplication of the hierarchical structure in the organization or the organization functions the entire process. It consists of heavyweight settings and predetermined fixed business logic.

Similar instances

In the production department in the traditional automotive industry, production is organized and divided into different production halls with assigned responsibilities. Physically within the production hall, assembly lines are established and their management responsibilities are subject to the production hall. Within the assembly line, the parts of vehicles are assembled in particular locations. From the management perspective, only the full path of this vertical structure makes the entire production process functional.

In the healthcare industry, hospitals are always organized in a functional organizational structure with a pyramid-shaped hierarchy to carry out a strict chain of command (Thompson et al, 2015). From the president, chair of department, section chief to doctor and nurse, this vertical structure is strictly established in all branches of the organization.

2.2.2 Structure-flexible pattern

Structure-flexible pattern denotes that an organization is extended with mutable hierarchical structure in any horizontal and vertical branches.

The organizational affiliation relationships are formed with loosely coupled interconnections, which imply more flexibility adapting to the business change over time. It consists of lightweight settings and customized business logic.

Similar instances

An automotive company used to establish the complete layers in the organizational structure, however, in its joint venture companies, due to lack of control, management power cannot reach all units in the organization. In addition, for a newly-built subsidiary or factory overseas, considering the investment risk, not all of the facilities and management suits will be established at one time. In this case, a flexible structure of the organization and supporting IS should be able to adapt to these needs.

Another example can be well observed in the government structure in many countries. For instance, in Germany, Bayern state contains Munich, Nuremberg and many other cities. A state has a higher administrative authority than a city. However, as an exception, Berlin as a city has the same higher administrative authority as Bayern and under Berlin's hierarchical layer, no more cities can be found. The same governmental management examples can also be observed from those special administrative regions in countries.

Therefore, for those newly-emerged or exceptional cases, the IS should capture the reasonable exceptions and special needs.

2.2.3 Modeling approaches and complexity patterns in implementation

Applying two modeling approaches (OOM and DOM) to tackling complexity patterns, structure-consistent (SC) and structure-flexible (SF), the experimental implementations were conducted with the combination illustrated in Figure 2-5.

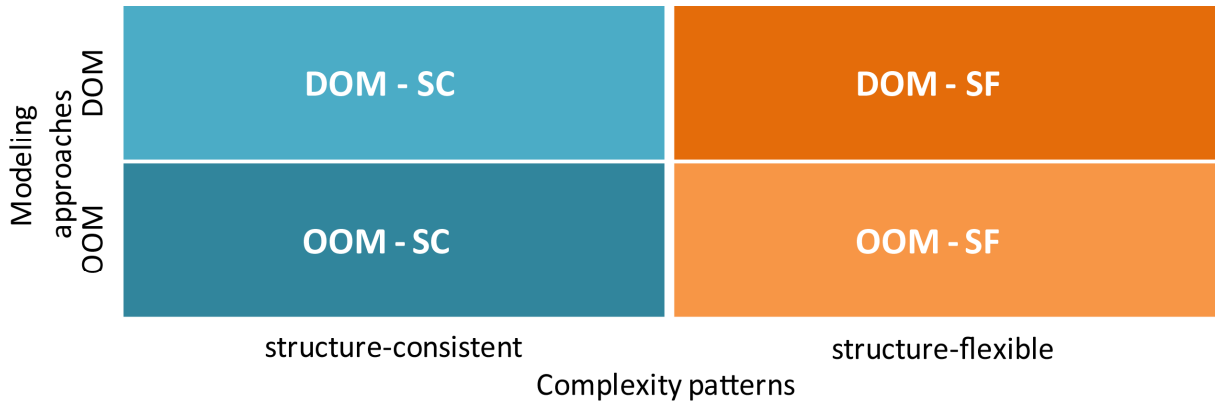


Figure 2-5 Mapping modeling approaches and complexity patterns

2.3 Complexity Measure

Since there is either little influence to complexity in the software, or it requires a great deal of efforts to modify or redo after implementation, without a quantitative measure of model and software complexity, projects have to rely on the personal experience of individuals.

As discussed in Section 2.1, there are two aspects of complexity in ISD: one is inherently generated from the business domain, which is reflected in requirements analysis with UML models; the other one is the technical implementation, which realizes the UML models and transforms its complexity in the source code. Furthermore, in this section, the complexity measurement issues are tackled via those two aspects with complexity in UML models and software complexity in the software products, by introducing existing complexity measurement metrics for UML models and software products, which form a complexity evaluation framework for organizational modeling.

2.3.1 Complexity measure for UML models

In UML, there are nine modeling diagrams classified in two categories: class diagrams, object diagrams, component diagrams, and deployment diagrams, as structural diagrams, modeling the static view of the system; statechart diagrams, activity diagrams, sequence diagrams, and collaboration diagrams as behavioral diagrams, depicting the dynamic view of the system. In this research, class diagrams are employed to evaluation methods and frameworks.

2.3.1.1 Selection of model evaluation method

Recker (2005), introduced non-empirical (feature comparison, metamodeling, metrics approach, paradigmatic analysis, contingency identification, ontological evaluation) and empirical evaluation methods (survey, laboratory experiment, field experiment, case study, action research) in ISD. In those methods, the metrics approach is adopted, since quantitative metric values are comparable to address complexity and appropriateness of a method (Rossi & Brinkkemper, 1996).

Since class diagrams in UML present a static view of the system, the evaluation of models with class diagrams can only be performed with structural measurement and static analysis, analyzing structural properties of classes, such as size, coupling, inheritance, and complexity. The structural properties of a class are considered indicative of the cognitive complexity of the class.

Cognitive complexity denotes that the mental burden falls on the people who deal with the classes (developers, inspectors, testers, maintainers, etc.), namely the understandability of models designed in the early phase as a media between stakeholders (Briand & Wüst, 2001). In general, high cognitive complexity of classes causes undesirable external qualities, such as

decreased maintainability and testability, or increased fault-proneness (Briand & Wüst, 2001). The relationship between structural class properties, cognitive complexity, and external quality can be illustrated as in Figure 2-6.

Figure 2-6 Relationships between class, cognitive complexity and SW quality

Therefore, in this research the evaluation method for UML class models is based on cognitive complexity metrics.

2.3.1.2 Cognitive complexity in UML class diagrams

A number of metrics were developed to measure the complexity in UML class diagrams in recent decades. Briand and Wüst (2001) introduced a structural measurement on cognitive complexity to identify internal quality properties of the source code that have an impact on the maintainability and reusability of the system classes. Similar research was also conducted by Genero (2001), regarding the relationship between the understandability of models and software maintainability. However, a “canonical set of non-redundant metrics" does not exist to “capture” all significant design properties that can be “valid for all systems” (SDmetrics, 2015).

Therefore, based on those cases, this research defines its own cognitive complexity measurement framework for UML class diagrams, selecting metrics from numerous existing studies focusing on major issues in cognitive complexity in UML class diagrams, in four structural dimensions: size, coupling, inheritance, and complexity.

Size metric

The Size metric is used to count the number of classes in the UML class model.

$$CCsize = \sum NC$$

CC_{size} : cognitive complexity on the size dimension
NC: number of classes in UML class diagrams

Size metrics contribute to estimating costs or effort for implementation, review, testing, and maintenance. In an iterative development process, more and more functionality is added in over time, possibly containing unrelated responsibilities. This, in turn, negatively impacts the understandability, reusability, and maintainability of the design model (SDmetrics, 2015).

Coupling metric

Coupling is the degree to which the classes in a model are connected with a "client-supplier" relationship in terms of import coupling or export coupling (Briand+b, 1997).

MIC (Method Invocation Coupling): MIC indicates the relative number of classes to a given class which sends messages, also called import coupling (Badri et al, 2009).

CBO (Coupling Between Objects) counts for a class, the number of other classes to which it is coupled (Chidamber & Kemerer, 1994). Two classes are considered coupled when methods declared in one class use methods or instance variables defined by the other class (Badri et al, 2009). CBO typically implies export coupling.

In UML, coupling consists of a broad scope of class relationship types, such as dependency (weak coupling), realization (weak coupling), association, aggregation, composition and inheritance. The inspection of this dimension focuses on the number of couplings associated with domain classes. Therefore, the coupling metric is summarized as:

$$CCcoupling = \sum (NICD + NXCD)$$

$CC_{coupling}$: cognitive complexity on the dependency dimension
NICD: number of import couplings associated with domain classes
NXCD: number of export couplings associated with domain classes

If high export coupling classes (CBO) change in the future, the system can be largely impacted if the interface is affected by the change (SDmetrics, 2015).

Inheritance scale metric

Inheritance allows a better reusability of the code. The inheritance dimension here is different from the inheritance in the coupling dimension. This inheritance dimension is defined as the indication of the scale of structural complexity in the core domain models with a product of both horizontal and vertical dimensions.

NOC (Number of Children): NOC is the number of immediating subclasses subordinated to a class in the class hierarchy.

DIT (Depth of Inheritance Tree): DIT of a class is given by the length of the inheritance path from the root of the inheritance hierarchy to the class on which it is measured (number of ancestor classes) (Chidamber & Kemerer, 1994).

Since NOC and DIT indicate the two dimensions of inheritance, width and depth, a more comprehensive metric for inheritance measurement should multiply them together as a benchmark:

$$CCinheritance\ scale = NOC * DIT$$

$CC_{inheritance\ scale}$: cognitive complexity on the inheritance dimension
NOC: the width of the inheritance tree in UML class diagrams
DIT: depth of the inheritance tree in UML class diagrams

Complexity Metric

The complexity here refers to the potential cyclomatic complexity in the resulting programs in design models. In this research the number of cyclomatic methods in domain classes are calculated:

$$CCcomplexity = \sum NCM$$

CCcomplexity: cognitive complexity on the structure complexity dimension
NCM: number of cyclomatic methods in domain classes

It is considered to be a good indicator of fault proneness. A high complexity value indicates a higher risk of faults in the class, also lower understandability (Basili et al, 1996; Emam et al, 2001).

2.3.2 Complexity measure for software

McCabe (1976) discussed a graph-theoretic measure for complexity and its uses in the management and control of program complexity. Halstead (1977) designed the Halstead software difficulty metric, also recognized as lines of code (LOC) metric, to identify measurable properties of software, and the relations between them. Rossi (1994) proposed a set of seventeen complexity metrics, divided into three categories: independent measures, aggregate metrics, and method-level metrics. Rossi and Brinkkemper (1996) also presented a systematic approach for measuring properties of methods, to measure the relative complexity of single diagram techniques and of complete systems development methods. A number of researchers conducted fundamental studies on complexity measurement. Conte et al. (1989), Fenton (1991) and El Emam & Koru (2008) also proposed different kinds of complexity metrics, as code-based measures.

2.3.2.1 Software complexity metrics

Software complexity metrics employed in this research are divided into four categories: size, dependency, inheritance, and complexity, aligned with the dimensions in cognitive complexity in UML class diagram (see Section 2.3.1.2), and also supported by the analysis tools CodePro (CodePro, 2015) and STAN (STAN4J, 2015).

Size metrics

Size metrics indicate the numerical scale of software. A number of closely relevant metrics for measuring the size dimension for software complexity in this research are selected:

- Number of Classes (NC)
- Number of Methods
- Number of Lines (NOL)
- Lines of Code (LOC)
- Average Number of Fields Per Unit
- Average Number of Methods Per Unit

Dependency /coupling metrics

An excessive coupling between a system's classes has a significant impact on modularity (Badri et al, 2009). A reduction of coupling between classes must take place to "improve modularity and promote encapsulation" (Larman, 2003). Due to these reasons, the dimension is inspected with the following metrics:

- Afferent Coupling (Import Coupling)
- Efferent Coupling (Export Coupling)
- Coupling Between Objects (CBO)

Afferent Coupling (import coupling): "The number of types outside the target elements that depend on types inside the target elements" (CodePro, 2015).

Efferent Coupling (export coupling): "The number of types inside the target elements that depend on types outside the target elements "(CodePro, 2015).

Coupling Between Objects (CBO): counts for a class the number of other classes to which it is coupled (Chidamber & Kemerer, 1994).

Inheritance metrics

Since inheritance allows a better reusability of the source code, the degree of inheritance is inspected with the following metrics:

- (Average) Number of Sub Units / Number of Children
- (Average) Depth of Inheritance Hierarchy

(Average) Number of Sub Units / Number of Children: "The (average) number of subtypes for the types defined in the target elements" (CodePro, 2015).

(Average) Depth of Inheritance Hierarchy: "The (average) depth of the units defined in the target elements. The depth of an interface is defined to be one. The depth of any other class is defined to be one more than the depth of its superclass" (CodePro, 2015).

Complexity Metrics

A high complexity indicates a higher risk of faults in the classes, also higher difficulty for comprehension:

- Cyclomatic Complexity (CC)
- Weighted Methods (WM)

Cyclomatic Complexity: The McCabe's cyclomatic complexity is equal to number of different paths in a method (function) plus one (McCabe, 1976).

Weighted methods (per class): The sum of the complexities of its methods in an evaluation unit.

2.3.2.2 Software complexity measurement tools

The software complexity measurement tools employed in this research are two Eclipse plug-ins: STAN from Stan4j and CodePro from Google. They support different statistical analysis metrics for size, dependency, inheritance, complexity, as shown in Table 2-3.

Table 2-3 Software complexity metrics (selected) in analysis tools

Metric name	CodePro	STAN
Size		
Number of Classes (NOC)	√	√
Number of Methods	√	-
Lines of Code (LOC)	√	√
Average Number of Fields Per Unit	√	√
Average Number of Methods Per Unit	√	√
Coupling		
Efferent Coupling (Export Coupling) in domain classes	√	√
Afferent Coupling (Import Coupling) in domain classes	√	√
Coupling Between Objects (CBO)	-	√
Inheritance		
(Average) Number of Sub Units / Number of Children	-	√
(Average) Depth of Inheritance Hierarchy	√	-
Complexity		
Cyclomatic Complexity (CC)	√	√
Weighted Methods (WM) (per Class)	√	√

2.3.3 Complexity measurement framework in this research

The complexity measurement framework in this research consists of two parts: (1) evaluation of cognitive complexity in UML class diagrams and (2) evaluation of software complexity in the resulting software artifacts based on the identical business case.

2.3.3.1 Evaluation framework for cognitive complexity in UML class diagrams

As discussed in Section 2.4.1.2, the evaluation framework for cognitive complexity is examined from four dimensions: size, coupling, inheritance, and complexity, as illustrated in Table 2-4.

Table 2-4 Cognitive complexity dimensions and inspection

Dimensions	Inspections
Size	implementation efforts or workload
Coupling	change efforts for maintenance and comprehension
Inheritance scale	reusability of the code
Complexity	indicator of fault proneness and comprehension

After implementation, the analytical data for cognitive complexity will be filled in the radar to create a visualized evaluation, as shown in Figure 2-7.

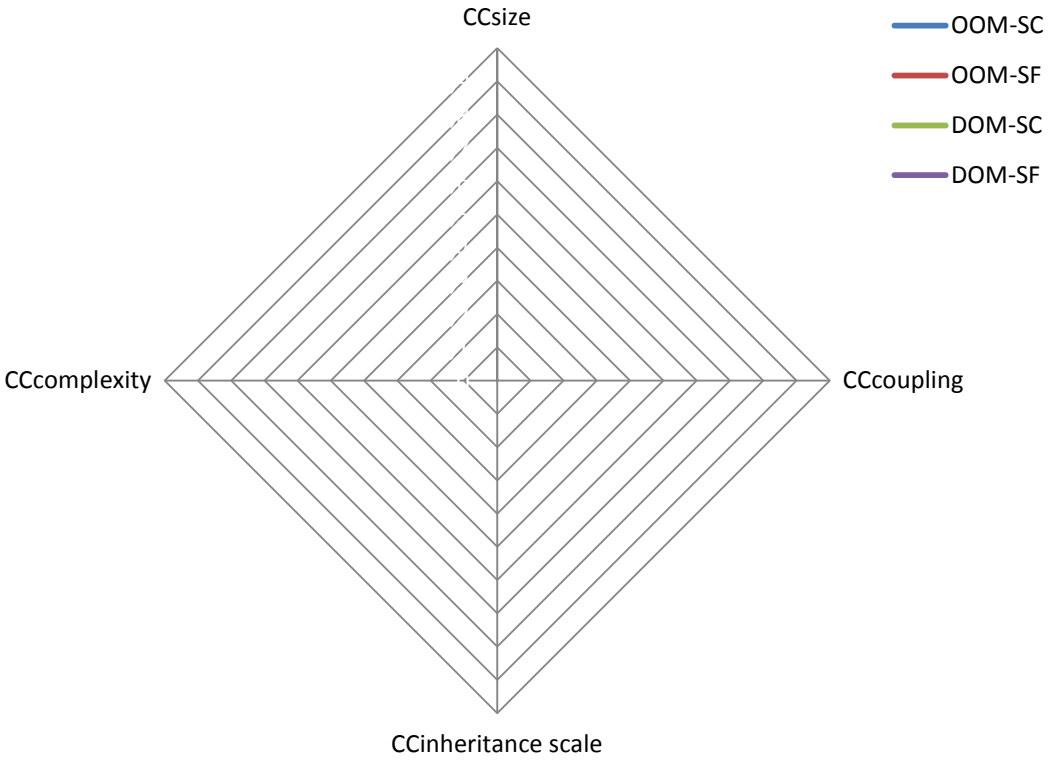


Figure 2-7 Visualized evaluation for cognitive complexity

2.3.3.2 Evaluation framework for software complexity in the resulting software

There are a number metrics illustrated in Table 2-1 for observing the software quality from various perspectives. However, for an integrated evaluation framework and focused

evaluation criteria, only the starred sections are selected to represent the dimension, as shown in Table 2-5.

After implementation, the analytical data for software complexity will be filled in the radar to create a visualized evaluation, as shown in Figure 2-8.

Table 2-5 Statistical evaluation framework for software complexity

Metric names	OOM-SC	OOM-SF	DOM-SC	DOM-SF
Size				
Number of Classes*				
Number of Methods				
Lines of Code (LOC)				
Average Number of Fields Per Unit				
Average Number of Methods Per Unit				
Coupling				
Afferent Coupling (Import Coupling) in domain classes*				
Efferent Coupling (Export Coupling) in domain classes*				
Coupling Between Objects (CBO)				
Inheritance				
(Average) Depth of Inheritance Hierarchy*				
(Average) Number of Sub Units / Number of Children*				
Complexity				
Cyclomatic Complexity (CC)				
Weighted Methods (WM) *				

*selected factors represent the respective dimension in graphical evaluation framework

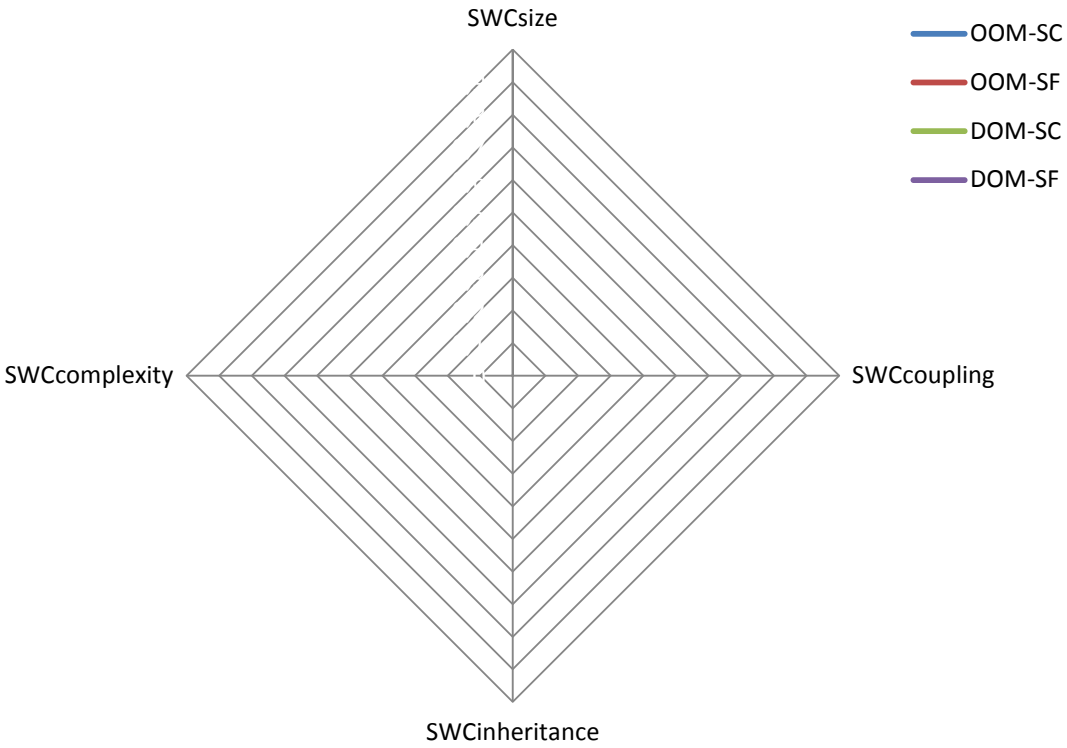


Figure 2-8 Visualized evaluation for software complexity

3 CASE ABSTRACTION OF AUDI PROJECT

The Resource Process Visualization project at Audi was an extended project after the AMOS project in the university. In order to adapt to the requirements in real life, the entire application was modified and implemented with multiple functional modules, such as the major module of resource visualization, a configuration tool of organizational units, data integration portal and data synchronization, etc.

3.1 Case Abstraction for Research Topic

Aligned with the research topic, an abstracted business case from the resource process visualization project at Audi was created with the minimum but the major functional module in the application, the hierarchical resource visualization.

Basic concepts of this business case are summarized below:

(1) The application aims to supply an application of AUDI's global production plants.

(2) A navigateable world map allows hierarchical browsing from the top to bottom layers of the organization, in order of Global, Hall (production hall), Line (assembly line), Location (locations for placing testing devices in a line), Device (testing devices containing the testing data of electrical components) and Component (electrical components).

(3) At each hierarchy of browsing, the aggregated testing status is displayed for that level. Meanwhile, the testing status of each element from the lower hierarchy is also presented.

(4) The aggregation strategy can be modified.

(5) Joint venture companies and the newly-built subsidiaries of Audi AG might have different organizational structures than the existing ones.

4 MODELING APPROACHES IN INFORMATION SYSTEM DEVELOPMENT

According to Rumbaugh (1991), modeling is used for four purposes: testing physical entities before building them (simulation), communication with customers, visualization (alternative presentation of information), and reduction of complexity. The purpose and effects of complexity reduction are not clear while employing different modeling approaches.

This chapter introduces two modeling approaches, Object Oriented Modeling (OOM) and Dynamic Object Model (DOM) and particularly highlights their structural characteristics, instead of a full coverage. The structural characteristics are the major components in the evaluation framework in this thesis.

4.1 Object-Oriented Modeling Approach

The object oriented (OO) paradigm was first applied in programming to improve the productivity of the source code. However, afterwards the OO idea was used in the entire development process including the analysis and design phases (Booch, 1991; Cobryn, 1999). The world consists of objects as well as their behaviors, so the best way of modeling reality is with objects and that is where object-oriented approach came in (Podgorelec Hericko & Juric, 2004).

4.1.1 Object Oriented Modeling

4.1.1.1 Characteristics of OOM

According to Rumbaugh (1991), the characteristics of an Object-Oriented approach include four aspects: identity, classification, polymorphism, and inheritance.

Identity denotes that data can be quantized into discrete, distinguishable entities called objects. It is recognized as the unique reference for each object.

Classification requires that the objects with the same data structure (attributes) and behavior (operations) should be grouped into a class. Each object is an instance of its class.

Polymorphism enables the same operation to behave differently on different classes. Different objects can respond to the same message in their own way. An operation is an action or transformation that an object performs or is subject to.

Inheritance means the sharing of attributes and operations among classes based on a hierarchical relationship. Each sub-class takes on all of the properties and inherits characteristics of its super-class adds its own unique properties.

4.1.1.2 Analysis models in OOM

In object-oriented modeling, three complementary analysis models are used to describe a system, illustrated in Table 4-1.

Table 4-1 Analysis models in object-oriented modeling (Podgorelec, 2004)

Models	Represents:	Consists of:
Object model	System structure	Classes, objects associations, links, methods, properties
Dynamic model	System behavior	States, events, actions, transitions
Functional model	System functionality	In UML: actors, use cases In OMT: actors, data stores, data flows

The object model describes the static structure of the objects in a system and their relationships. It contains object diagrams, with the components of classes, objects associations, links, methods and properties (Bruegge, 2010).

The dynamic model describes how a system changes over time and the interactions among the objects and is represented by state machine and sequence diagrams. It contains state diagrams, whose nodes are states and whose edges are transitions between states caused by events (Bruegge, 2010).

The functional model describes the data value transformations within a system and is represented by use cases and scenarios. It contains data flow diagrams whose nodes are processes and whose edges are data flows (Bruegge, 2010).

The object model is the most fundamental model providing the structural characteristics of the system. Object models are used for model evaluation in this research.

4.1.2 Object Models with UML

Object-oriented modeling is typically used via use cases and abstract definitions of the important objects. The most common language used for OOM is the Object Management Group's Unified Modeling Language (UML). Object models depicted with UML class diagrams present the static structure of systems.

Object models are presented with UML class diagrams, which contain entities in the problem represented by classes and the relationships between them.

4.1.2.1 Types of relationships in UML

UML relationships are used to define the structure between model elements. Typical relationships include associations, dependencies, generalizations, and realizations, etc.

An association relationship is a structural relationship between two model elements. In this relationship, the objects of one classifier connect and can navigate to objects of another classifier. In a bidirectional relationship, an association connects two classifiers, the primary as supplier and secondary as client (IBM, 2015). There are two important association relationships, aggregation and composition. An aggregation relationship defines a classifier as a part of another one. The lifecycle of the part classifier is independent from the lifecycle of the whole classifier. A composition relationship defines a whole-part relationship, where the lifecycle of the part classifier is dependent on the lifecycle of the whole classifier (IBM, 2015).

A dependency relationship shows that changes to one model element (the supplier or independent model element) can cause changes in another model element (the client or dependent model element). Since a change in the client does not affect the supplier, the supplier model element is independent. The client model element is dependent on the supplier's messages (IBM, 2015).

A generalization relationship suggests that a specialized (child) model element is extended from a general (parent) model element. The parent model element may contain one or more children, and any child model element can have one (single inheritance) or more parents (multiple inheritance) (IBM, 2015).

A realization relationship is an implementation relationship between a classifier and a provided interface. The realization relationship defines that the realizing classifier must conform to the contract that the provided by the interface (IBM, 2015).

Those typical relationships are employed for modeling in this research.

4.2 Dynamic Object Model Approach

In order to enable the flexibility of creating and modifying the software, another approach, Adaptive/Dynamic Object Model (DOM), was developed in recent decades. In DOM, previous researchers contributed a number of studies to present the separation of the entity (on the operational level) from the entity type (on the knowledge level) with DOM, employing the Type Object pattern (Fowler, 1996; Yoder et al, 1998; Riehle et al, 2005), which created the flexibility to modify the software, even at runtime.

This section introduces the related research regarding DOM, Type Object and a more complicated practice called Nested Type Object. According to the project experiences, the scenarios of object creation in the Type Object and Nested Type Object are summarized for the future researcher, in the respective subsections.

4.2.1 Dynamic Object Model

The Adaptive/Dynamic Object Model (DOM) is an instance-based software architectural pattern that represents domain-specific entities, attributes, relationships, and behavior using metadata (Foote & Yoder, 1998; Yoder et al, 2001; Yoder & Johnson, 2002). DOM architecture consists of several patterns to represent a domain model and its behavior (HEN-TOV, 2010). DOM is a compound pattern (Riehle, 1997; Riehle et al, 2005). It is composed of several smaller patterns: Type Object and Property pattern or Type Object, Property List and Value Holder patterns as a more specific division (Johnson & Wolf, 1996; Foote & Yoder, 1998; Riehle et al, 2000).

In DOM, the Type Object is used to divide the system into entities and entity types. The metadata descriptions of domain-specific definitions are stored externally in terms of a property container and interpreted at runtime. Users can change the metadata to reflect changes in the domain. More details regarding Type Objects and Property patterns are introduced in the next subsection.

DOM enables runtime entity and behavior referencing and mapping with its components (Type Object, Property List and Value Holder), instead of hard coding and tight coupling in programs. DOM is also known as the Active Object Model, Adaptive Object Model, Reflective Object Mode and Runtime Domain Model and so on.

4.2.2 Type Object

4.2.2.1 Motivation

In traditional OOD, domain classes are designed with fixed attributes and behaviors. Any change to the domain classes requires recompilation and redeployment. In addition, from

the domain modeling perspective, any subclasses should be explicitly defined in advance, which means if a class has 100 sub level domains, 100 subclasses should be implemented and would cause class explosion in the system. The Type Object was designed to solve those issues: runtime modification of objects, avoiding bloated inheritance and higher reusability.

4.2.2.2 Structure

As the major component in DOM, the Type Object is used to create the separation of an object in traditional OOM with an entity and the type of that entity. Entities have attributes that can be defined, employing Attributes. Each Attribute has its own type, namely AttributeType. Each EntityType can specify the types of the Attributes for its Entities (the realization approaches differ). It distinguishes a type level from an instance level (Riehle et., 2005), where the type level is called "knowledge level" and instance level is call "operational level" according to Fowler (1996). The simplified relationships are presented in Figure 4-1.

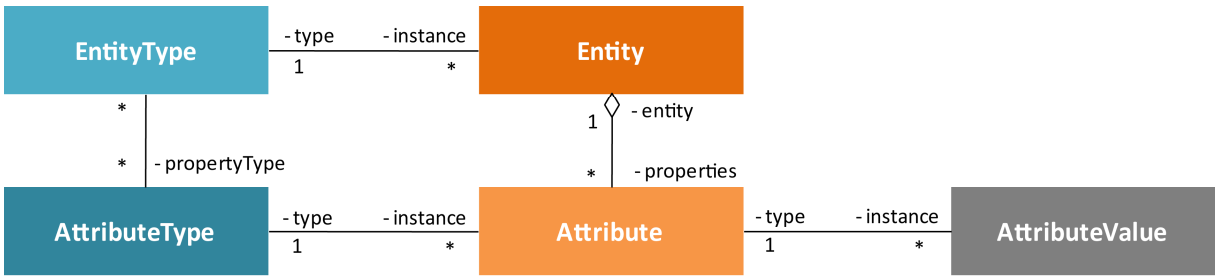


Figure 4-1 Type Object (Riehle et al, 2000)

With this separation, in order to create an object (the traditional one in OOD), four classes should be implemented: the Entity class, EntityType class, Attribute class, AttributeType class. An object of any of those classes holds the values of that class, as mapped with traditional OOD in Table 4-2.

Table 4-2 Understanding the structure of Type Object

In Type Object an object of	Mapping in traditional OOD
EntityType class	the type of the target object to create in OOD
Entity class	as a value container for the target object to create in OOD, containing the values of the attributes in OOD. These attributes are instantiated with **Attribute** class as a prerequisite.
AttributeType class	the type of an attribute of the target object in OOD
Attribute class	as a value container for the target object to create in OOD, containing the values of the attributes in the target object in OOD. The values are held in the **AttributeValue** class.

The Type-referencing relationship in DOM is considered to be used for twice, between the Entity and EntityType, as well as between the Attribute and AttributeType. The Type-

referencing relationship between the Entity and EntityType is the basis of creating an object and this relationship cannot be implicitly constructed, but is directly dependent via the classes. However, the relationship between the Attribute and AttributeType can be implemented in an implicit relationship by removing the dependency from classes to external configuration files.

As described by Riehle et al. (2000), the relationships between Entities and Attributes, as well as between EntityTypes and AttributeTypes, are conjoined with Property Lists, which are externally defined property/configuration files. In addition, since the Property class combing the AttributeValue class plays a role of holding the values for attributes, this class collaboration was named the Value Holder. Therefore, a full structure of DOM can be illustrated as show in Figure 4-2.

Since no previous researchers have explicitly presented the process of object creation with the Type Object, the next subsections will interpret this process step by step.

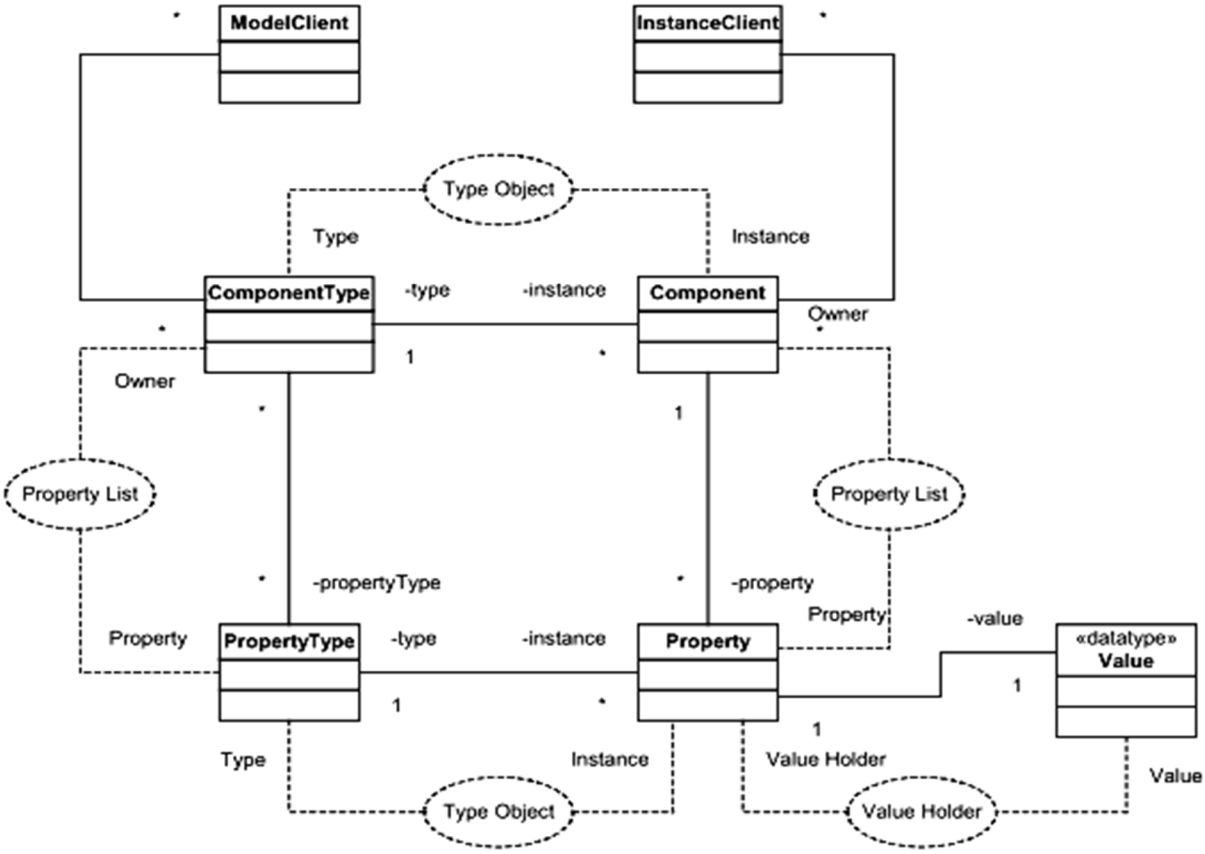


Figure 4-2 Full illustration of DOM (Riehle et al, 2000)

4.2.2.3 Scenario of object creation in Type Object

The creation and initialization of objects, representations of entities and attributes, and the definition of relationships and behavior are controlled by metadata. In DOM, object creation and initialization rely on inter-dependent relationships, which are significantly different from

the straight forward convention in traditional OOD. This complexity causes great difficulty for model designers and developers to apply this pattern in practice.

The *TypeObject* creation process in DOM can be described with the following scenario:

(1) Input to EntityType
As a type-driven architectural pattern, the *TypeObject* creation in DOM usually starts with an input of "type", provided by the client class or a factory method for Type Object instance creation. In order to create a specific instance, the ID of the target instance is also specified.

(2) Initialization of Entity
An Entity is initialized within the lifecycle of the EntityType. During the initialization of the Entity, a chain of initializations of its Attributes are invoked.

(3) Initialization of Attribute
Attributes are initialized within the lifecycle of the Entity. The major tasks in this step are to access the data source, separate data values and types referring to the *Property List* for the EntityType-AttributeType relationship, and store the data values in the *Value Holder*.

(4) Data (de)construction
During the above Initialization processes, the AttributeType is used as a reference to separate data from its types and also to reassemble them together. However, for this purpose the AttributeType does not have to build direct dependency to the Attribute object as a second Type Object, but bind their relationships in *the Property List*. An AttributeType does not have to be responsible for creating an Attribute instance, if just as a reference in the *Property List*, which could be different from the relationship between the EntityType and Entity.

4.2.3 Nested Type Object

Johnson and Woolf (1996) reported a more practical and complicated pattern in DOM, the Nested Type Object pattern. The Nested Type Object denotes the case where a *TypeObject* not only has its own information as its attributes, but also contains child elements (their structure is (similar to) a duplication of the parent element) as its attributes and this hierarchical relationship tree grow deeper recursively. In other words, a *TypeObject* is nested with another *TypeObject* recursively in a tree structure.

4.2.3.1 Scenario of object creation in Nested Type Object

Due to the recursive characteristic of the Nested Type Object, the object creation processes have to be modified slightly. The *NestedTypeObject* creation process can be described with the following scenario:

(1) Input to EntityType
This step is the same as step (1) in *TypeObject* creation process.

(2) Initialization of Entity
An Entity is initialized within the lifecycle of the EntityType. During the initialization of the Entity, a chain of initializations of its Attributes is invoked. Meanwhile, a recursive process of initializing its child elements / sub-type Entities is taking place (in parallel). Those two branches execute different process paths:

- Initializing an Entity's own Attributes:
  Requires finishing step (3) and (4) below.

- Initializing child Entities:
  Requires finishing step (1), (2), (3) and (4), with a recursive process.

Since creating an Entity requires a Type and an ID as input, a parent-child relationship can only be recursively used once on each layer. Therefore, the recursion will not be an infinite loop and will stop after creating its own child elements.

(3) Initialization of Attribute
This step is the same as step (3) in *TypeObject* creation process.

(4) Data (de)construction
This step is the same as step (4) in *TypeObject* creation process.

5 RESULTS OF MODELING APPROACHES TOWARD COMPLEXITY PATTERNS

Based on the theoretical foundation of complexity patterns (structure-consist and structure-flexible) in organizations (see Section 2.2), the identical business case (see Chapter 3) was implemented with OOM and DOM for each complexity pattern. As a result, three applications were implemented during the research process (OOM-SC, OOM-SF and DOM-SC&SF) to create the data basis for quantitative analysis. During the implementation, DOM successfully adapted to both complexity patterns simultaneously, due to its significant characteristics of dynamics.

5.1 Common Grounds in Implementations

In order to create consistency in software development styles and the comparable source code, an additive and minimum reduction development principle was implemented. Therefore, some application components were reused since the first solution (OOM for SC) and the architecture of the applications also stayed consistent.

5.1.1 Application architecture design

The application architecture was designed with a typical 4-tier architectural pattern, containing a presentation layer, business logic layer, data access layer and data source layer, as shown in Figure 5-1.

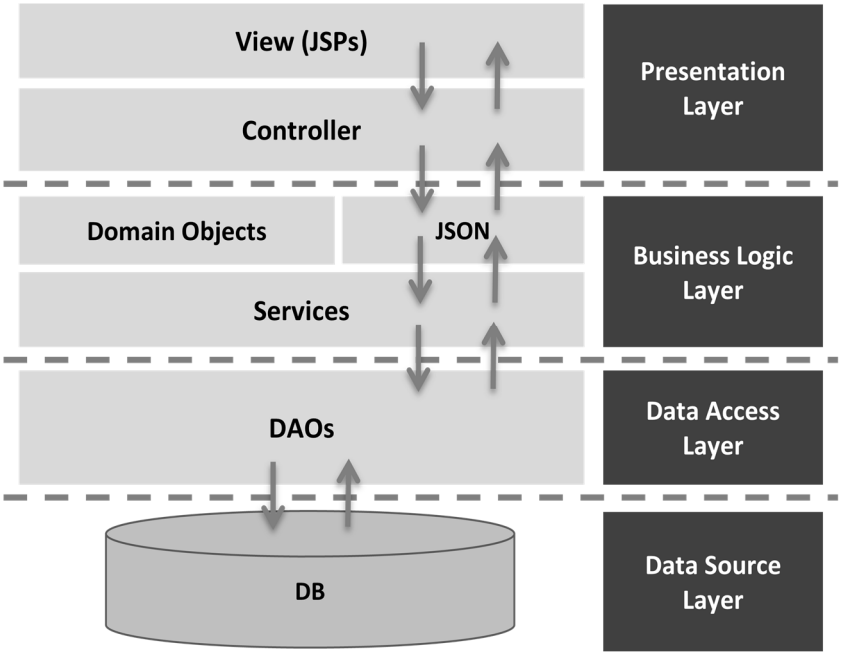


Figure 5-1 Application architecture design

A significant addition to this application was the intent of separation of concerns: every layer stays independent from others and in loosely coupled relationships. In addition, the aggregation of the testing status was also separated into batch programs after every update instead of low-efficiency and heavy computation while retrieving any hierarchy object at runtime

5.1.2 Application components

Three major common application components were reused for all implementation targets: user interface, database models, and application framework, although they are slightly different in the specific implementation. The applications were refined and modified with a clear MVC (Model-View-Controller) architectural pattern.

5.1.2.1 User Interface

The basic user interface settings were derived from the university project AMOS, an open source development project with Agile Methods. However, during the implementation in this research, a great number of efforts were invested to modify or create new contents in the user interface, as well as at the back-end, for a clean MVC architecture.

The layouts of the user interface in the resulting applications contain login, global, factory, production hall, assembly line, location, device and component views.

5.1.2.2 Database models

The database design reflects the business domain in the persistency in database. The implementations with OOM for both SC and SF complexity patterns employed the same mechanism, tight-coupling, for accessing the database; the implementation with DOM for both SC and SF complexity patterns employed loose-coupling for accessing the database.

Tight-coupling means that after an SQL query, the data extraction logic was written in the JAVA source code in a fixed and predefined attribute-column relationship between the JAVA object and database. Any change of the business domain (JAVA object) requires recompiling and redeployment of the entire application.

Loose-coupling means that after an SQL query, the data extraction logic was written in the external property files instead of the JAVA source code. The coupling is in a flexible and customized attribute-column relationship between the JAVA object and database. Any change of the business domain does NOT require recompilation and redeployment of the entire application.

Although, as the major part of the application, the business logic with JAVA was implemented according to different modeling approaches, none of them have any influence to the persistence layer of the application and the database models stay identical to any of those modeling and implementation diversities.

5.1.2.3 Framework

In order to create a clear separation of layers and MVC architecture, Struts2 was employed in the implementation, for request control, page forwarding, data access, as well as the support for the uniformed data transfer interface in terms of JSON format.

One of the significant advantages in the Struts2 framework (the same in Spring framework) is that the requested pages or actions are configured in the configuration file, struct.xml. The configuration settings can be used as handlers of application logic control. This reduced the programming workload and created a cleaner source code with separate layers.

A configuration mapping can refer to a webpage and also can invoke the services in a class and return the results with a uniformed data format in terms of JSON format, as shown in Figure 5-2.

```
<action name="globalRedirect" class="de.fau.reprovis.action.GlobalRedirectAction" method="execute">
    <interceptor-ref name="secureStack"/>
    <result name="success">/WEB-INF/content/GlobalView.jsp</result>
</action>

<action name="global" class="de.fau.reprovis.action.GlobalAction" method="execute">
    <interceptor-ref name="secureStack"/>
    <result name="success" type="json"/>
</action>

<action name="factoryRedirect" class="de.fau.reprovis.action.FactoryRedirectAction" method="execute">
    <interceptor-ref name="secureStack"/>
    <result name="success">/WEB-INF/content/FactoryView.jsp</result>
    <result name="input">/WEB-INF/content/login.jsp</result>
</action>

<action name="factory" class="de.fau.reprovis.action.FactoryAction" method="execute">
    <interceptor-ref name="secureStack"/>
    <result name="success" type="json"/>
</action>
```

Figure 5-2 Application configuration file in Struts2 framework (partial)

5.2 Object-Oriented Modeling Solutions for Complexity Patterns

As discussed in Section 2.3, complexity patterns in organizational modeling does not only indicate the different organizational structures, but also the inherent business characteristics in the models of IT systems. This phenomenon was observed in this research.

5.2.1 OOM for structure-consistent pattern

OOM for structure-consistent pattern is the most common thinking in IT system development with canonical data models and application models, where the hierarchical layers in the organization are determined in a fixed and logically immutable order and relationships. The models and implementation have a clear separation of responsibilities but low reusability.

5.2.1.1 Domain models in SC pattern

There are seven domain models, every one of which is aligned with its own business domain: global, factory, production hall, assembly line, location, device or component, as shown in Figure 5-3.

HierarchieElementBean was designed as the super class for the specific hierarchical elements (GlobalBean, FactoryBean, HallBean, LineBean, LocationBean, DeviceBean and ComponentBean), containing the common attributes: id, parent and childs/children. Within the HierarchieElementBean, child elements are initialized with this super class instead of initiating them in the sub classes.

Due to the structure-consistent characteristic, the subordinate relationships between hierarchy elements can be observed from their dependencies: a Global hierarchy element explicitly requires Factories as its child elements; a Factory explicitly requires Halls as its child elements; a Hall explicitly requires Lines as its child elements; a Line explicitly requires Locations as its child elements; a Location explicitly requires Devices as its child elements; a Device explicitly requires Components as its child elements.

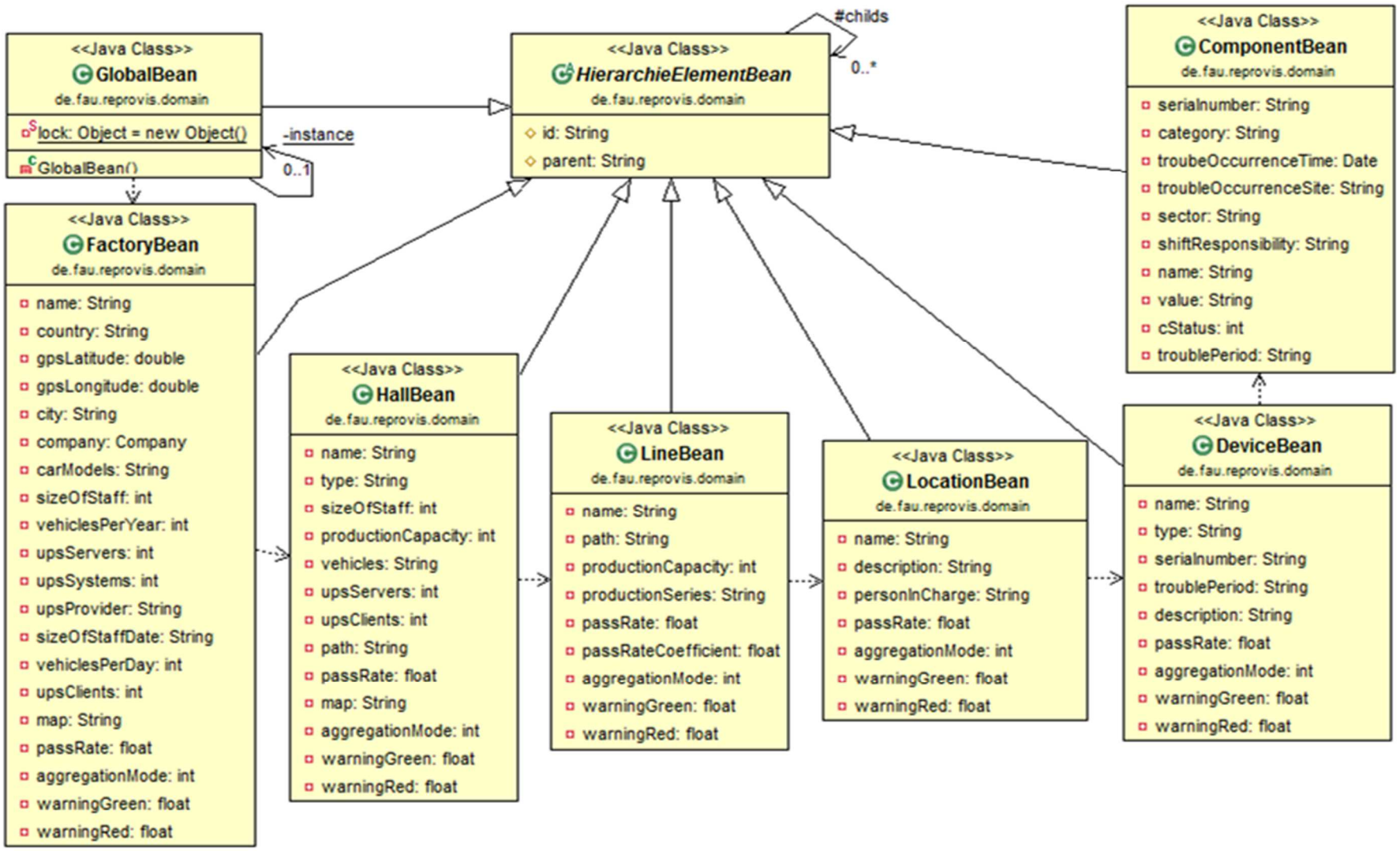


Figure 5-3 Domain models in OOM-SC implementation

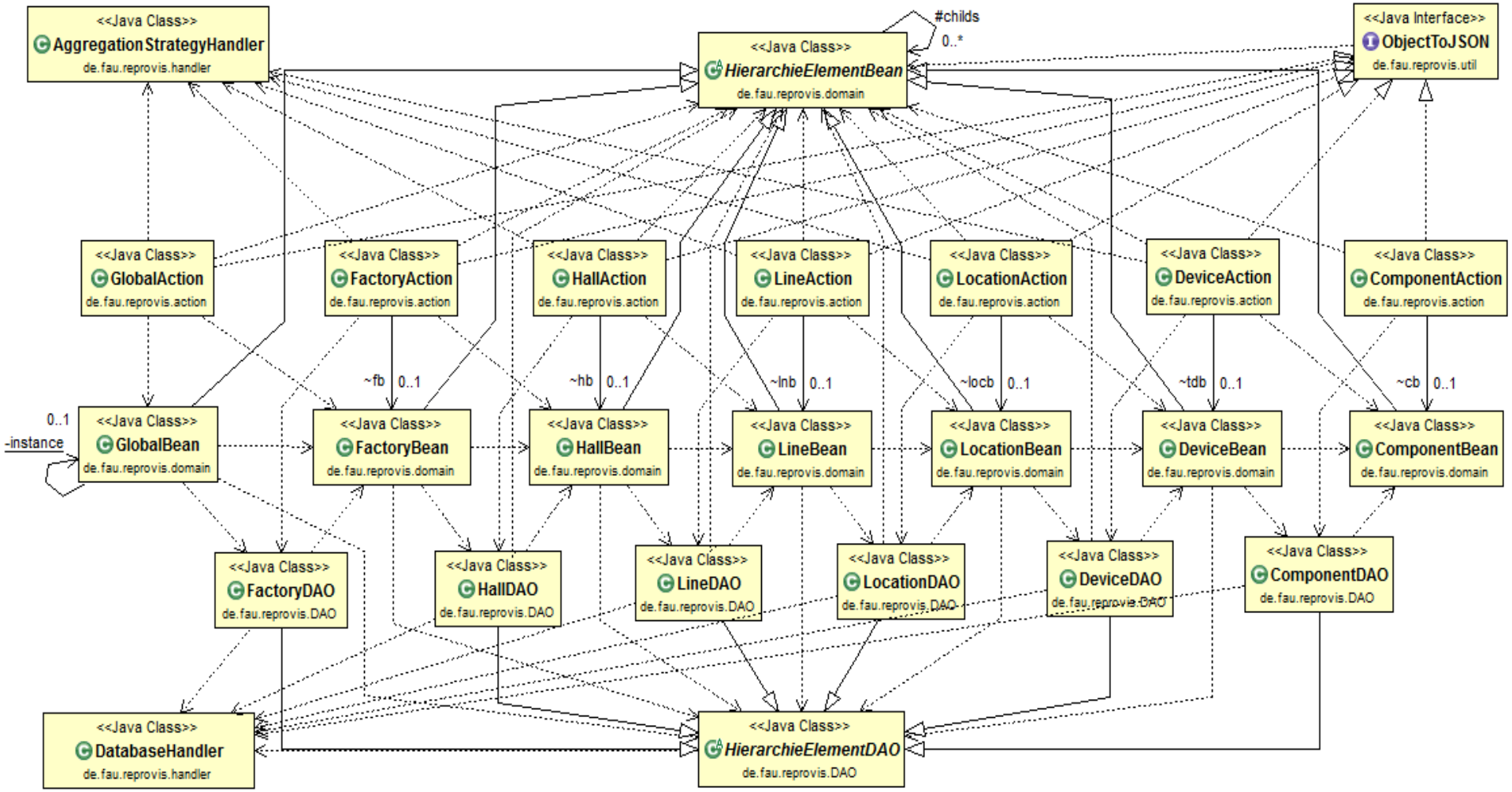


Figure 5-4 Application model in OOM-SC implementation

5.2.1.2 Application model in SC pattern

As introduced in Section 2.2, business complexity influences the software significantly, especially with regards to the specific architectural design. This is observed in the application model for the structure-consistent pattern, as shown in Figure 5-4.

The scenario for executing this application model is described below:

(1) When a call is triggered on the frontend, the framework forwards pages (use individual RediretAction classes, not shown in the diagram) and invokes an Action to response and fetch data.

(2) In the Action, the corresponding Hierarchy Bean is used to create an instance of its own hierarchy element, and the child instances are created by the super class HierarchieElementBean.

(3) Afterwards, all the child instances are returned in the form of a list, List<HierarchieElementBean>.

(4) For database accessing, each Hierarchy Bean uses its own DAO to fetch its own information, while HierarchieElementDAO was only responsible for fetching information for its sub-level/child instances with a fixed affiliation relationship.

In this application model, any change of logical relationships between hierarchy elements will cause a great deal of effort to modify the application; also, it might not be possible to do so without reengineering the entire application.

5.2.1.3 Functional features and class collaborations

As introduced in Chapter 3, a minimum set of features for resource visualization are implemented for the core research objective instead of other irrelevant features. As for the features, the associated classes are sub-structured into five major collaborations in the application model, as illustrated in Figure 5-5.

Aggregation collaboration

- Purpose: convert the pass rate for each hierarchy element, stored in the database, into a meaningful aggregated testing status
- Role types: Action classes as Client, AggregationStrategyHandler as a server Class

Data uniformization collaboration

- Purpose: after the data is retrieved from database before transferring to the frontend, all data is converted with a uniformed data format, in terms of JSON
- Role types: Action classes as Client, ObjectToJSON as a service interface

Own information access collaboration

- Purpose: on every page (hierarchical layer), access the data for that hierarchy element (also the instance)
- Role types: Action classes as Client, Hierarchy Bean as a domain class, DAOs as server classes

Child's information access collaboration

- Purpose: on every page (hierarchical layer), access the data for the child level instances of that hierarchy element (also the instance)
- Role types: Action classes as Client, Hierarchy Bean as a domain class, DAOs as server classes

Database access collaboration

- Purpose: responsible for acquiring data source and data fetching from database
- Role types: DAOs as Client, DatabaseHandler as a server class

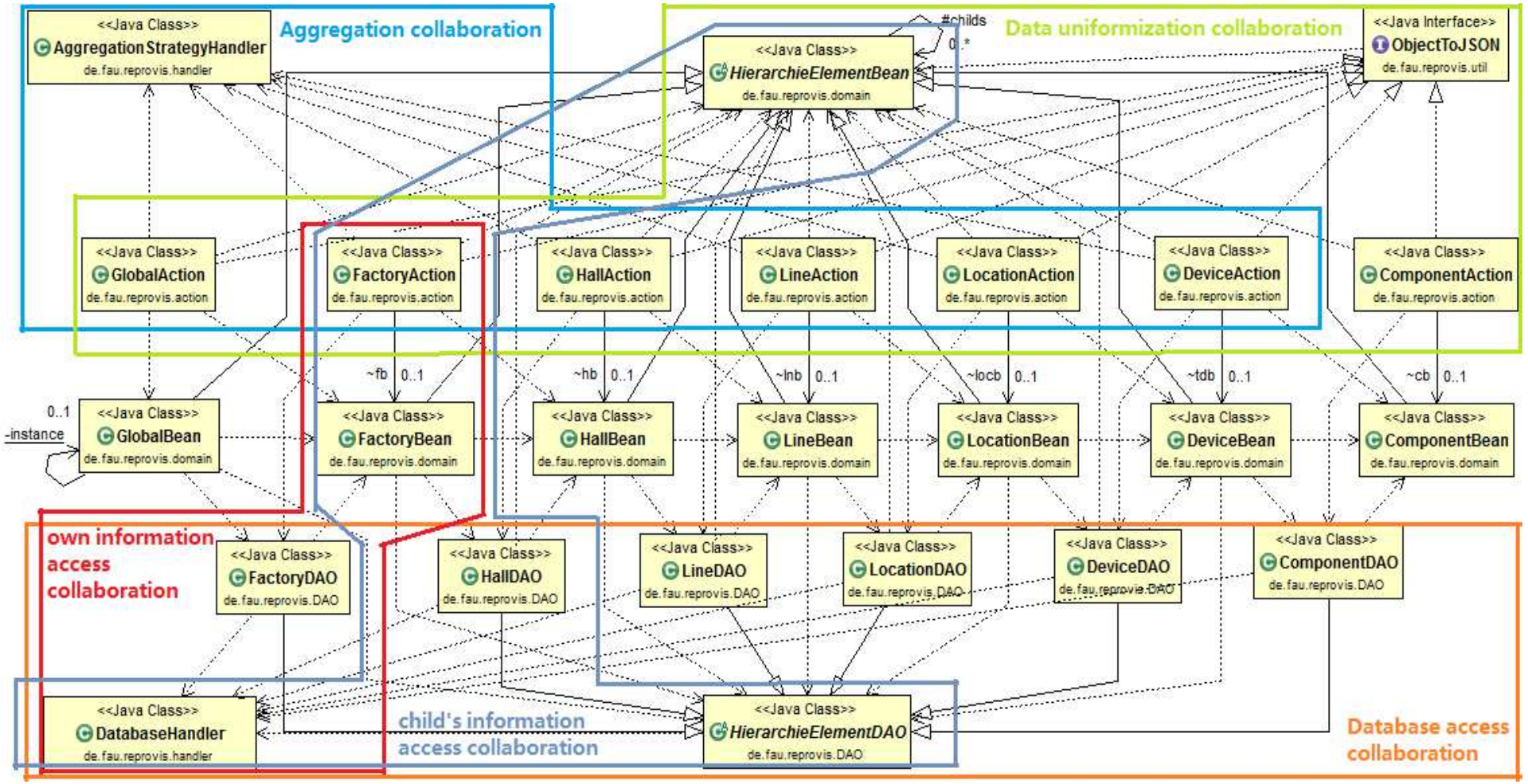


Figure 5-5 Class collaborations in OOM-SC implementation

5.2.2 OOM for structure-flexible pattern

A structure-flexible pattern indicates that the hierarchical layers in the organization are determined in logically mutable orders and relationships, which is recognized as changeable in the organization.

OOM for a structure-flexible pattern practices the separation of concerns between business decisions and IS and loosens its structural coupling. It facilitates the organization to adapt its business to the changing situation with a lighter-weight organizational roll-out. The models and implementation demonstrated a clear separation of relationships but still low reusability.

5.2.2.1 Domain models for SF pattern

The Structure-flexible pattern does not change the inherent attributes domain models (global, factory, production hall, assembly line, location, device and component). However, the inherent logical links between hierarchy elements are removed, as shown in Figure 5-6.

HierarchieElementBean was designed as the super class for the specific hierarchical elements (GlobalBean, FactoryBean, HallBean, LineBean, LocationBean, DeviceBean and ComponentBean), containing the common attributes: id, parent, status and childs/children. Within the HierarchieElementBean, child elements are initialized with this super class instead of initiating them in the sub classes.

Due to the structure-flexible characteristic, the subordinate relationships between the hierarchy elements cannot explicitly be observed from the models, instead the relationships are defined in an external configuration file: superHierarchyControl.xml (as shown in Figure 5-7). At runtime, a hierarchy element's child is not predetermined and is only decided when an object of that hierarchy element is initiated. The child instances are created while their parent object is initiated.

Therefore, in Figure 5-6, no explicit dependencies between hierarchy elements can be observed from the models.

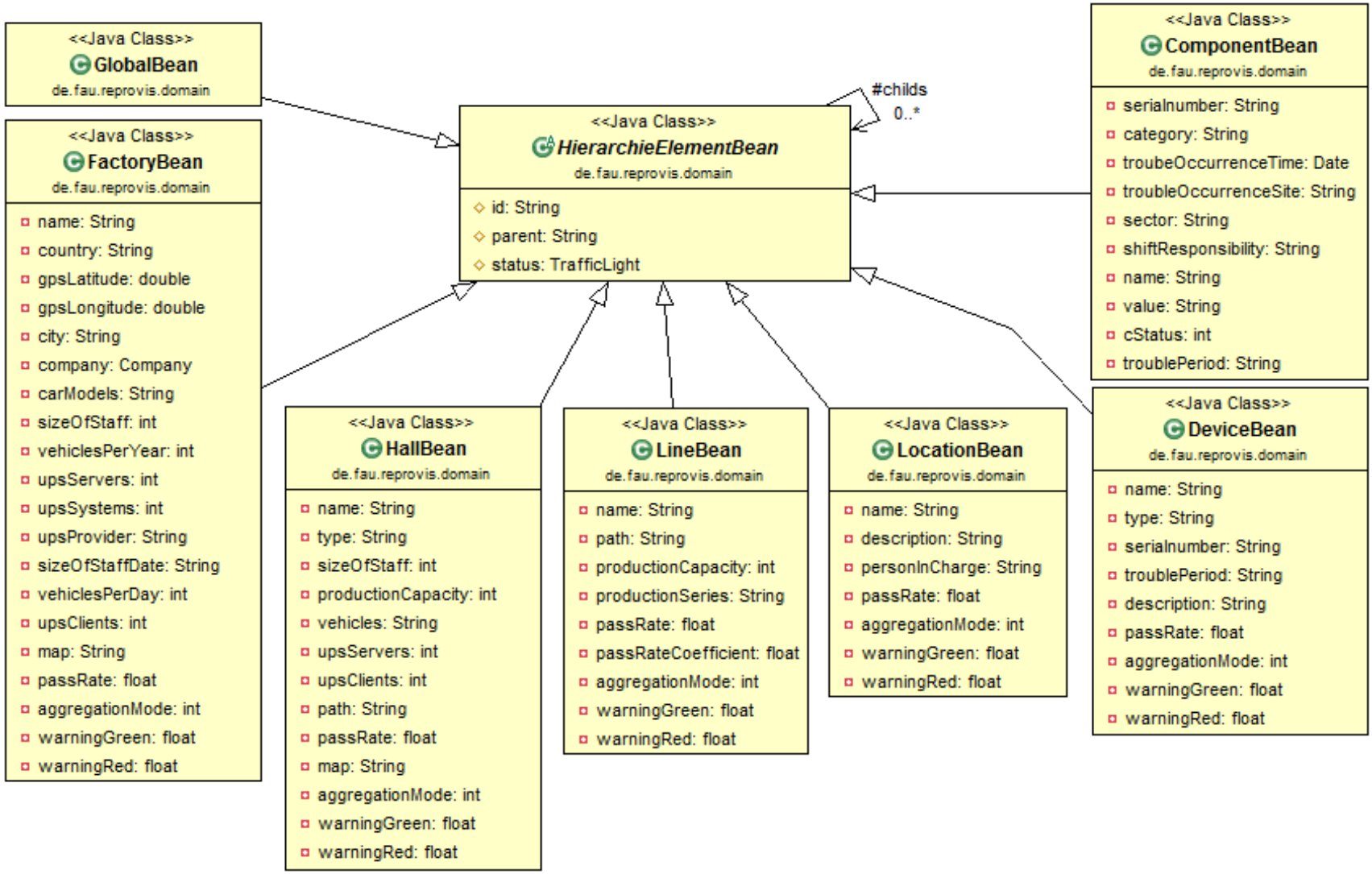


Figure 5-6 Domain models in OOM-SF implementation

The superHierarchyControl.xml was designed for several functional purposes:

(1) Block <hierarchy-element-mapping>
In this block, the overall hierarchical structure and the mapping between Class name and hierarchy name are defined.

(2) Block <hierarchy-scope-definition>
In this block, the factory is used as the handler hierarchy unit and hierarchical flexibility is restricted within a factory. As shown in Figure 5-7, the organizational structures differ due to the setting. Factory (id="02") has the full hierarchical structure, while factory (id="15") only has a simplified structure with two layers.

superHierarchyControl

```
<?xml version="1.0" encoding="UTF-8"?>
<super-hierarchy-control>
    <hierarchy-element-mapping>
        <hierarchy-element class="Global">global</hierarchy-element>
        <hierarchy-element class="Factory">factory</hierarchy-element>
        <hierarchy-element class="Hall">hall</hierarchy-element>
        <hierarchy-element class="Line">line</hierarchy-element>
        <hierarchy-element class="Location">location</hierarchy-element>
        <hierarchy-element class="Device">device</hierarchy-element>
    </hierarchy-element-mapping>
    <hierarchy-scope-definition>
        <hierarchy-unit-scope id="02">
            <property level="1">factory</property>
            <property level="2">hall</property>
            <property level="3">line</property>
            <property level="4">location</property>
            <property level="5">device</property>
        </hierarchy-unit-scope>
        <hierarchy-unit-scope id="15">
            <property level="1">factory</property>
            <property level="2">device</property>
        </hierarchy-unit-scope>
    </hierarchy-scope-definition>
</super-hierarchy-control>
```

Figure 5-7 Hierarchy control for OOM-SF

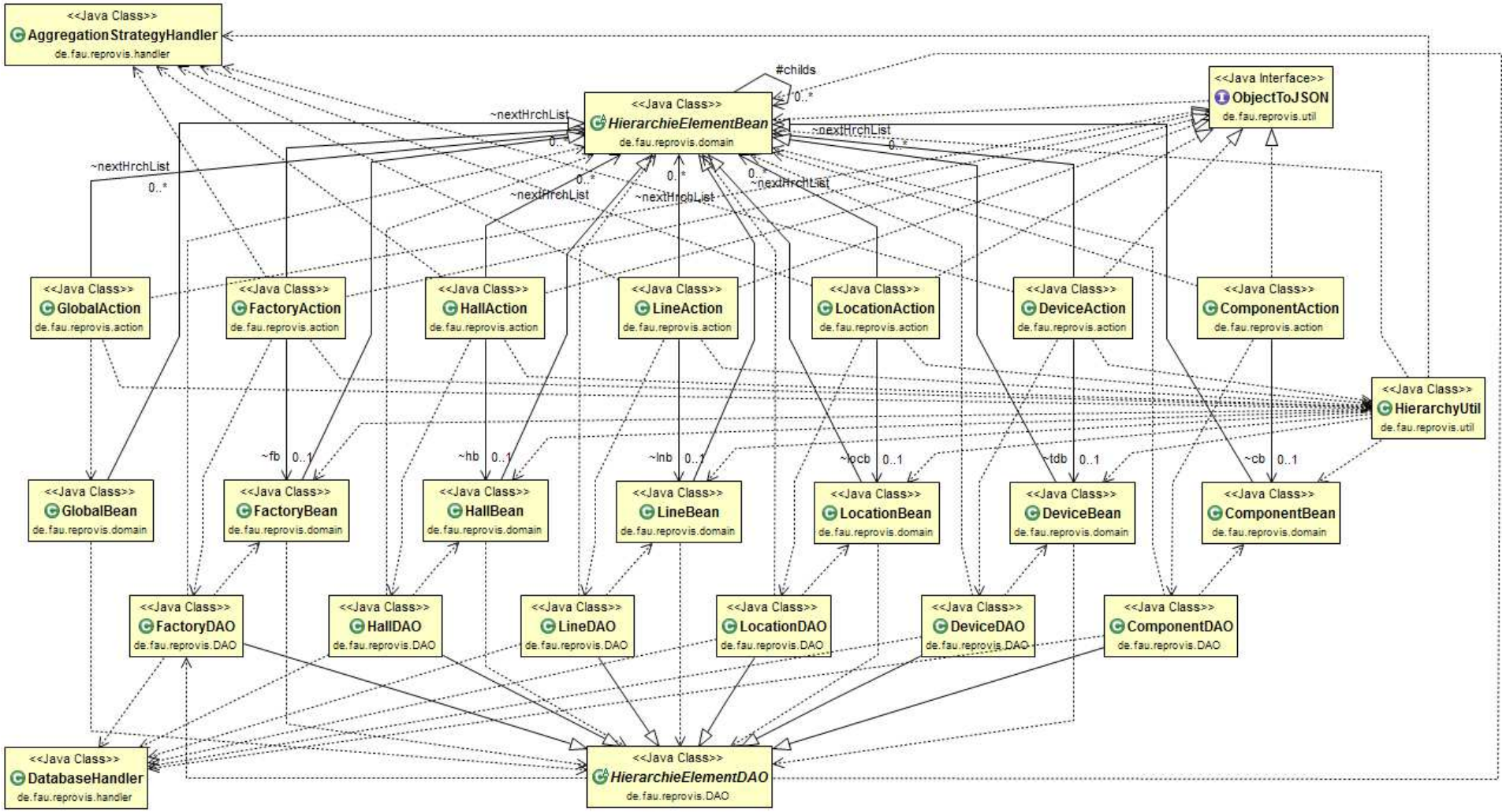


Figure 5-8 Application model in OOM-SF implementation

5.2.2.2 Application model

Also as introduced in Section 2.1.2, the business complexity indicates unpredictable changes over time. This possibility is also observed in the modeling and implementation for structure-flexible pattern, as shown in Figure 5-8.

The scenario for executing this application model can be described as follows:

(1) When a call is triggered on the frontend, the framework forwards pages (use individual RediretAction classes, not shown in the diagram) and invokes an Action to response and fetch data.

(2) In the Action, the corresponding Hierarchy Bean is used to create an instance of its own hierarchy element.

(3) Before creating its child instances, HierarchyUtil is referred to get the hierarchy definition, and afterwards, the child instances are created by the super class HierarchieElementBean.

(4) At last, all the child instances are returned in the form of list, List<HierarchieElementBean>.

(5) For database accessing, each Hierarchy Bean uses its own DAO class to fetch its own information, while HierarchieElementDAO was only responsible for fetching information for its sub-level/child instances with fixed affiliation relationship.

In this application model, any change of logical relationships between hierarchies causes relatively less effort to modify the application than the models with a structure-consistent pattern.

5.2.2.3 Functional features and class collaborations

As introduced in Chapter 3, a minimum set of features for resource visualization is implemented for the core research objective instead of other irrelevant features. As for the features, the associated classes are sub-structured into six major collaborations in the application model, as illustrated in Figure 5-9

Aggregation collaboration

- Purpose: convert the pass rate for each hierarchy element, stored in database, into meaningful aggregated testing status
- Role types: Action classes as Client, AggregationStrategyHandler as a server Class

Data uniformization collaboration

- Purpose: after the data is retrieved from database before transferring to the frontend, all data is converted with uniformed data format, in terms of JSON

- Role types: Action classes as Client, ObjectToJSON as a service interface

Own information access collaboration

- Purpose: on every page (hierarchical layer), access the data for that hierarchy element (also the instance)
- Role types: Action classes as Client, Hierarchy Bean as a domain class, DAOs as server classes

Child's information access collaboration

- Purpose: on every page (hierarchical layer), access the data for the child level instances of that hierarchy element (also the instance) by referencing the hierarchy settings
- Role types: Action classes as Client, Hierarchy Bean as a domain class, HierarchyUtil as well as DAOs as server classes

Database access collaboration

- Purpose: responsible for acquiring data source and data fetching from database
- Role types: DAOs as Client, DatabaseHandler as a server class

Hierarchy reference collaboration

- Purpose: within a specific factory, every single hierarchy element's child layer is defined in an external configuration file and the relationships can be changed
- Role types: Action classes as Client, Hierarchy Bean as a domain class, HierarchyUtil as server class

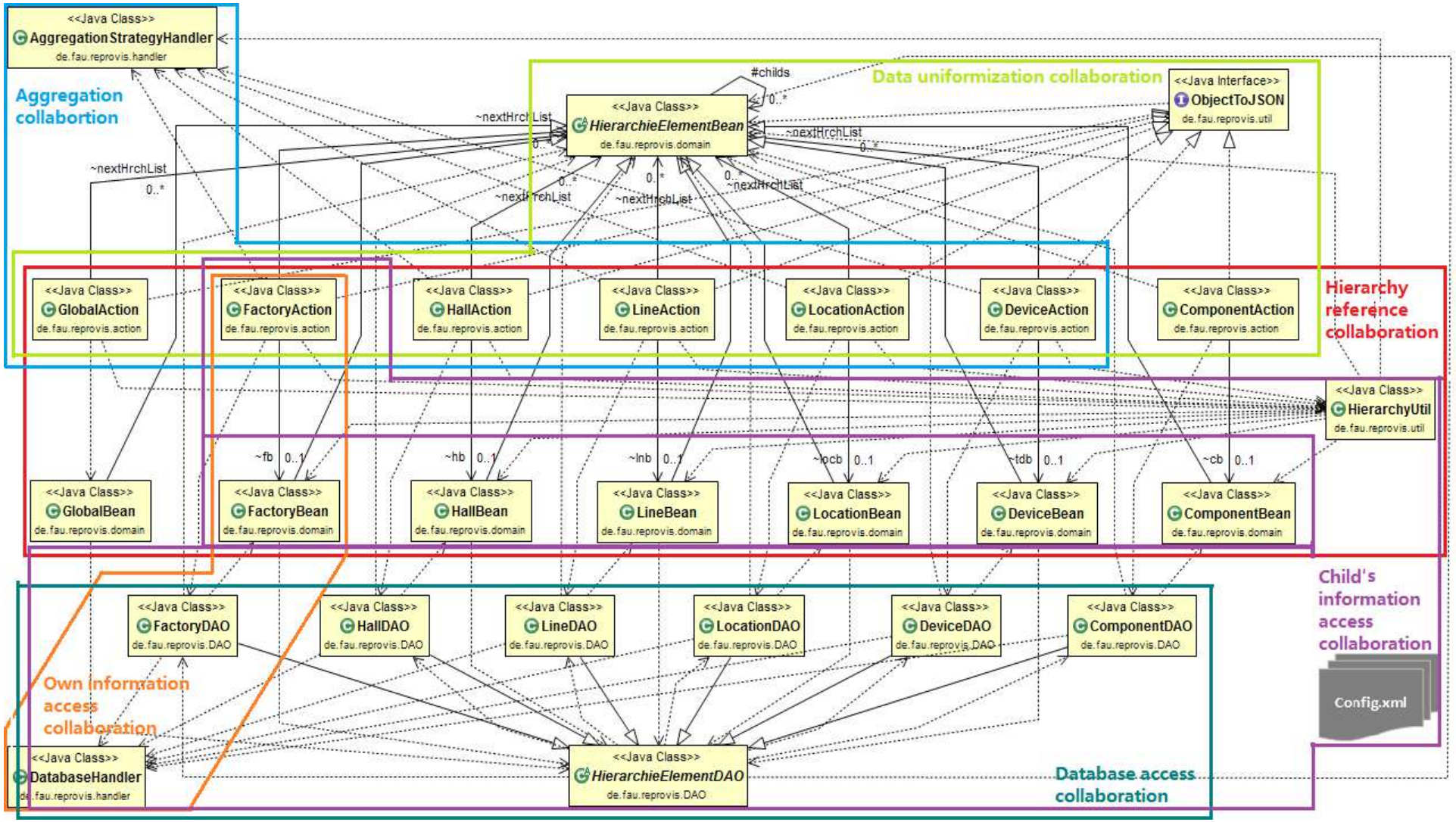


Figure 5-9 Class collaborations in OOM-SC implementation

5.3 Dynamic Object Model for Complexity Patterns

DOM is characterized with creating the flexibility to modify the software even at runtime without programming. This feature was significantly observed in the implementation for complexity patterns. The complexity patterns were well addressed with DOM only by modifying configuration files. The following subsections demonstrate the modeling and implementation results for DOM tackling complexity patterns from the perspective of data models, application model and functional features as well as class collaborations.

5.3.1 Type Object for complexity patterns

Due to the configurable and loose-coupling characteristic in DOM, the structure-consistent pattern in a DOM solution indicates that the hierarchical relationships between hierarchy elements should be defined and configured with a consistent hierarchical logic and structure in the external configuration file for all branches in the organization. The structure-flexible pattern in the DOM solution indicates that the hierarchical relationships between hierarchy elements can be defined and configured with a flexible logic and structure in an external configuration file.

In addition, since each *TypeObject* is an independent object representing a hierarchy element, the hierarchical relationship definition plays the role of chaining the hierarchy elements together.

5.3.1.1 Adaption to both SC and SF patterns

The solution for adapting to both SC and SF patterns was designed with two separate tag blocks in the configuration file as shown in Figure 5-10 and Figure 5-11.

The application checks any input ID in the <flexible-typeObject-hierarchies> block first to see if it is an exception (according to the ID) from the full hierarchy definition in <default-typeObject-hierarchies>. If so, its child hierarchy is returned with the defined value in the childType attribute in the hierarchy-element tag; if not, the application uses the hierarchy settings from the default structure defined in <default-typeObject-hierarchies> and according the current Type to retrieve the next level in an ascending order.

Adapting to a structure-flexible pattern with DOM does not require modifying the source code in the backend. However, the alignment for different attributes for presentation is required. Nevertheless, the application can still be modified at runtime without recompilation and redeployment. DOM for complexity patterns demonstrated both the capacity of loose-coupling between hierarchy elements and high reusability at the same time.

TOHierarchyControl ×

```
<?xml version="1.0" encoding="UTF-8"?>
<typeObject-hierarchies>
    <default-typeObject-hierarchies>
        <hierarchy-element level="0" type="global"/>
        <hierarchy-element level="1" type="factory"/>
        <hierarchy-element level="2" type="hall"/>
        <hierarchy-element level="3" type="line"/>
        <hierarchy-element level="4" type="location"/>
        <hierarchy-element level="5" type="device"/>
        <hierarchy-element level="6" type="component"/>
    </default-typeObject-hierarchies>
    <flexible-typeObject-hierarchies>
        <hierarchy-element id="1010" type="hall" childType="line"/>
        <hierarchy-element id="1011" type="hall" childType="location"/>
    </flexible-typeObject-hierarchies>
</typeObject-hierarchies>
```

Figure 5-10 Hierarchy control for DOM-SC

TOHierarchyControl ×

```
<?xml version="1.0" encoding="UTF-8"?>
<typeObject-hierarchies>
    <default-typeObject-hierarchies>
        <hierarchy-element level="0" type="global"/>
        <hierarchy-element level="1" type="factory"/>
        <hierarchy-element level="2" type="hall"/>
        <hierarchy-element level="3" type="line"/>
        <hierarchy-element level="4" type="location"/>
        <hierarchy-element level="5" type="device"/>
        <hierarchy-element level="6" type="component"/>
    </default-typeObject-hierarchies>
    <flexible-typeObject-hierarchies>
        <hierarchy-element id="1010" type="hall" childType="line"/>
        <hierarchy-element id="1011" type="hall" childType="location"/>
    </flexible-typeObject-hierarchies>
</typeObject-hierarchies>
```

Figure 5-11 Hierarchy control for DOM-SF

5.3.1.2 Domain models for SC and SF patterns

A *TypeObject* (introduced in Section 4.2) is composed with two *Type* classes, two *Instance* classes and a V*alue Holder* class as the minimum setup. The concrete object representing a specific domain is created and decided at runtime, according to the type-instance definition for each entity and its attributes in the Property List (Figure 5-12) in the form of description. Therefore, no concrete classes for any specific domain can be explicitly seen from the data

models, but conceptually understood as domain specific. In addition, since any relational links between the concrete domains on the modeler or developers' mind are defined in the external hierarchy definition configuration file, no explicit relationships or dependencies can be observed from the data models either.

PropertyList

```
<?xml version="1.0" encoding="UTF-8"?>
<Property-definition>
    <Property-unit id="global">
        <attribute name="id" type="string"/>
    </Property-unit>
    <Property-unit id="factory">
        <attribute name="id" type="string"/>
        <attribute name="parent" type="string"/>
        <attribute name="name" type="string"/>
        <attribute name="country" type="string"/>
        <attribute name="gpsLatitude" type="double"/>
        <attribute name="gpsLongitude" type="double"/>
        <attribute name="city" type="string"/>
        <attribute name="company" type="string"/>
        <attribute name="carModels" type="string"/>
        <attribute name="sizeOfStaff" type="int"/>
        <attribute name="vehiclesPerYear" type="int"/>
        <attribute name="upsServers" type="int"/>
        <attribute name="upsSystems" type="int"/>
        <attribute name="upsProvider" type="string"/>
        <attribute name="sizeOfStaffDate" type="string"/>
        <attribute name="vehiclesPerDay" type="int"/>
        <attribute name="upsClients" type="int"/>
        <attribute name="map" type="string"/>
        <attribute name="passRate" type="float"/>
        <attribute name="aggregationMode" type="int"/>
        <attribute name="warningGreen" type="float"/>
        <attribute name="warningRed" type="float"/>
    </Property-unit>
    <Property-unit id="hall">
```

Figure 5-12 Property List in DOM

However, there is an issue when fetching data with the Type. All the data for this Type will be returned to the application, while the frontend might only need a sub-set of the properties. This should be adapted and defined in the uniformed data transfer interface in the form of JSON format. In order to achieve the adaption between frontend fields and fetched data from the backend, a Properties2Frontend.xml configuration file was required as a coordinator in between. Properties2Frontend.xml works as a filter to select necessary properties for presentation.

Properties2Frontend

```
<?xml version="1.0" encoding="UTF-8"?>
<Property-definition>
    <Property-unit id="global">
        <attribute name="id" type="string"/>
    </Property-unit>
    <Property-unit id="factory">
        <attribute name="id" type="string"/>
        <attribute name="name" type="string"/>
        <attribute name="country" type="string"/>
        <attribute name="gpsLatitude" type="double"/>
        <attribute name="gpsLongitude" type="double"/>
        <attribute name="city" type="string"/>
        <attribute name="company" type="string"/>
        <attribute name="carModels" type="string"/>
        <attribute name="sizeOfStaff" type="int"/>
        <attribute name="vehiclesPerYear" type="int"/>
        <attribute name="upsServers" type="int"/>
        <attribute name="upsSystems" type="int"/>
        <attribute name="upsProvider" type="string"/>
        <attribute name="sizeOfStaffDate" type="string"/>
        <attribute name="map" type="string"/>
        <attribute name="vehiclesPerDay" type="int"/>
        <attribute name="upsClients" type="int"/>
        <attribute name="passRate" type="float"/>
        <attribute name="status" type="string"/>
        <attribute name="statusName" type="string"/>
    </Property-unit>
    <Property-unit id="hall">
        <attribute name="id" type="string"/>
        <attribute name="name" type="string"/>
```

Figure 5-13 Properties mapping to frontend

Considering a nested object, the nesting can be created on an operational level (instance level), although conceptually the nesting logic is based on the knowledge level (type level).

Given the issues above, the data model for a *TypeObject* in DOM can be illustrated as in Figure 5-13.

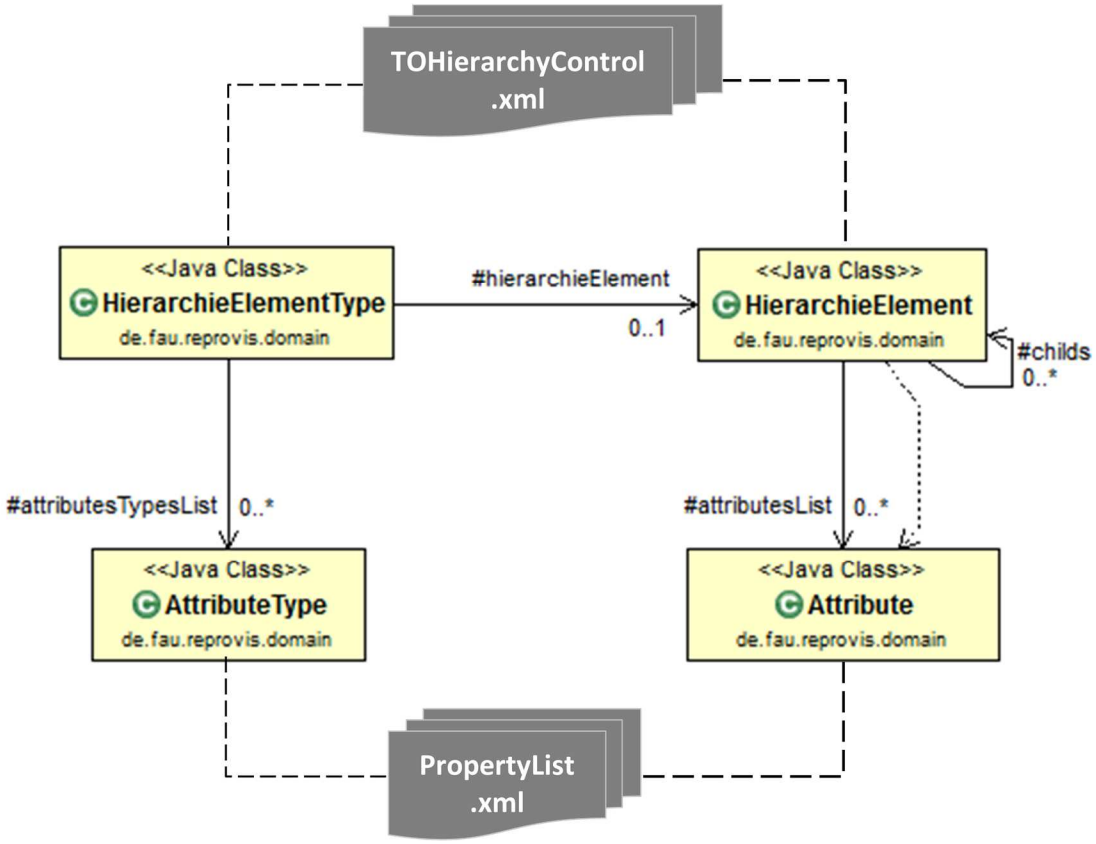


Figure 5-14 Domain models in DOM-SC and DOM-SF implementation

5.3.1.3 Application model for SC and SF patterns

With DOM, the unpredictable business complexity in organizations can be handled on several levels: (1) the hierarchical relationships in organizations can be configured in the hierarchy control configuration file at runtime; (2) different attributes within an organization can be modified in the property list at runtime; (3) different behaviors of each hierarchy type can be easily adapted with a Strategy pattern.

Significantly different from OOM, as illustrated in Figure 5-14, this DOM application presents an extremely abstract model layout, without any concrete classes for specific domains: a HierarchieElementAction class replacing all Action classes for each hierarchy as in OOM, a Type Object class (containing HierarchieElementType, HierarchieElement, AttributeType, Attribute and AttributeValueBean classes) replacing all domain classes, and a HierarchieElementDAO class replacing all DAO classes for each hierarchy's database accessibility.

Executing a DOM application model requires understanding the type-driven procedure with the Type Object, which is difficult for the developers who are new to this concept. The scenario for executing this application model can be described as follows:

(1) When a call is triggered on the frontend, the framework forwards pages (use individual RediretAction classes, not shown in the diagram) and invokes HierarchieElementAction to response with an input Type and ID to fetch data.

(2) In HierarchieElementAction, according to the input Type and ID, with the assistance of HierarchieElementFactory, a *TypeObject* is created (see Section 4.2.2.3 and Section 4.2.3.1) to represent the specified (by Type) domain and hold the values for the specified (by ID) instance.

(3) Within the lifecycle of HierarchieElementAction while creating a *TypeObject*, HierarchieElementDAO is employed to access the values according to the Type as well as ID and also the values for the sub-level/child instances by referring the Type and ID.

(4) During the creation of the *TypeObject*, predefined behavioral strategies are applied.

(5) In the end, all instances are transformed to a uniformed data format (i.e. JSON) and returned to the frontend.

In this application model, any change of logical relationships between hierarchies causes the no effort to modify the application but only the settings in the configuration files.

5.3.1.4 Gained experiences from Type Object and Nested Type Object

After practicing with DOM, some important and valuable experiences were gained for modeling dynamic circumstances in the business domain and implementing software systems employing DOM with Type Object and Nested Type Objet patterns, which were summarized together with the introduction of DOM for future researchers, in the following subsections:

- Scenario of object creation in Type Object (see Section 4.2.2.3)
- Scenario of object creation in Nested Type Object (see Section 4.2.3.1)

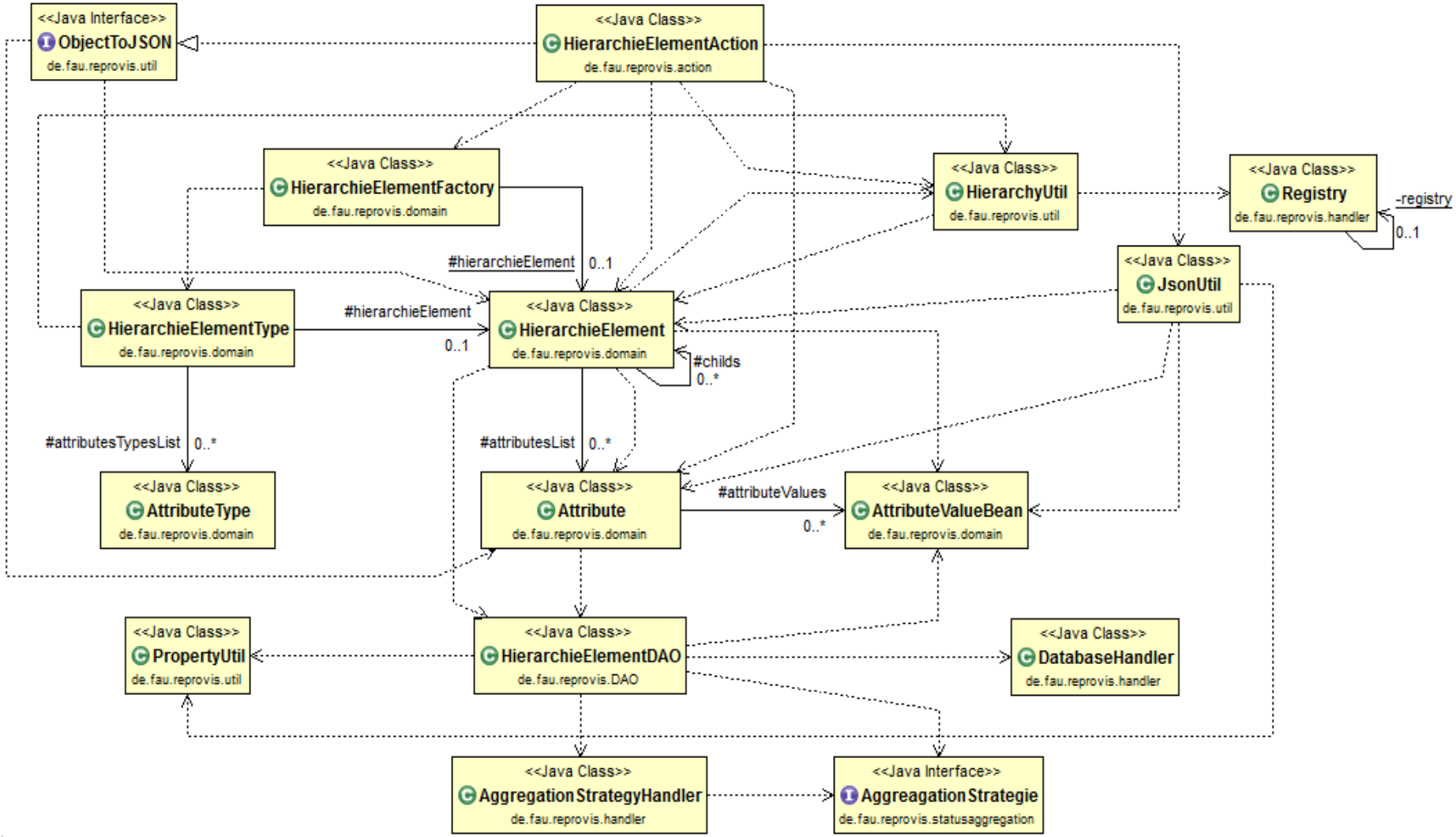


Figure 5-15 Application model in DOM-SC and DOM-SF implementation

5.3.1.5 Functional features and class collaborations

As introduced in Chapter 3, a minimum set of features for resource visualization are implemented for the core research objective, instead of other irrelevant features. As for these features, the associated classes are sub-structured into seven major collaborations in the application model, as illustrated in Figure 5-15.

Factory method collaboration

- Purpose: with factory method, the client can create an instantiated *TypeObject* with input Type and ID
- Role types: HierarchieElementAction as client, HierarchieElementFactory as server class

Hierarchy reference collaboration

- Purpose: a hierarchy element finds the hierarchical relationship underneath its layer
- Role types: HierarchieElementAction and HierarchieElement as Client, HierarchyUtil as server class

Type Object collaboration

- Purpose: create a descriptive *TypeObject* for a specific Type
- Role types: HierarchieElementType and AttributeType as type/knowledge level classes; HierarchieElement, Attribute and AttributeValueBean as instance/operational level classes

Type Object instantiation collaboration

- Purpose: instantiates the instance/operational level classes with concrete values
- Role types: HierarchieElement, Attribute and AttributeValueBean as client, HierarchieElementDAO and PropertyUtil as server classes

Database access collaboration

- Purpose: responsible for acquiring data source and data fetching from database
- Role types: DAOs as Client, DatabaseHandler as a server class

Aggregation collaboration

- Purpose: convert the pass rate for each hierarchy element into a meaningful aggregated testing status
- Role types: HierarchieElementDAO as client, AggregationStrategyHandler as server class

Data uniformization collaboration

- Purpose: after the data is retrieved from the database before transferring to the frontend, all data is converted with a uniformed data format, in terms of JSON
- Role types: HierarchieElementAction classe as client, ObjectToJSON as a service interface

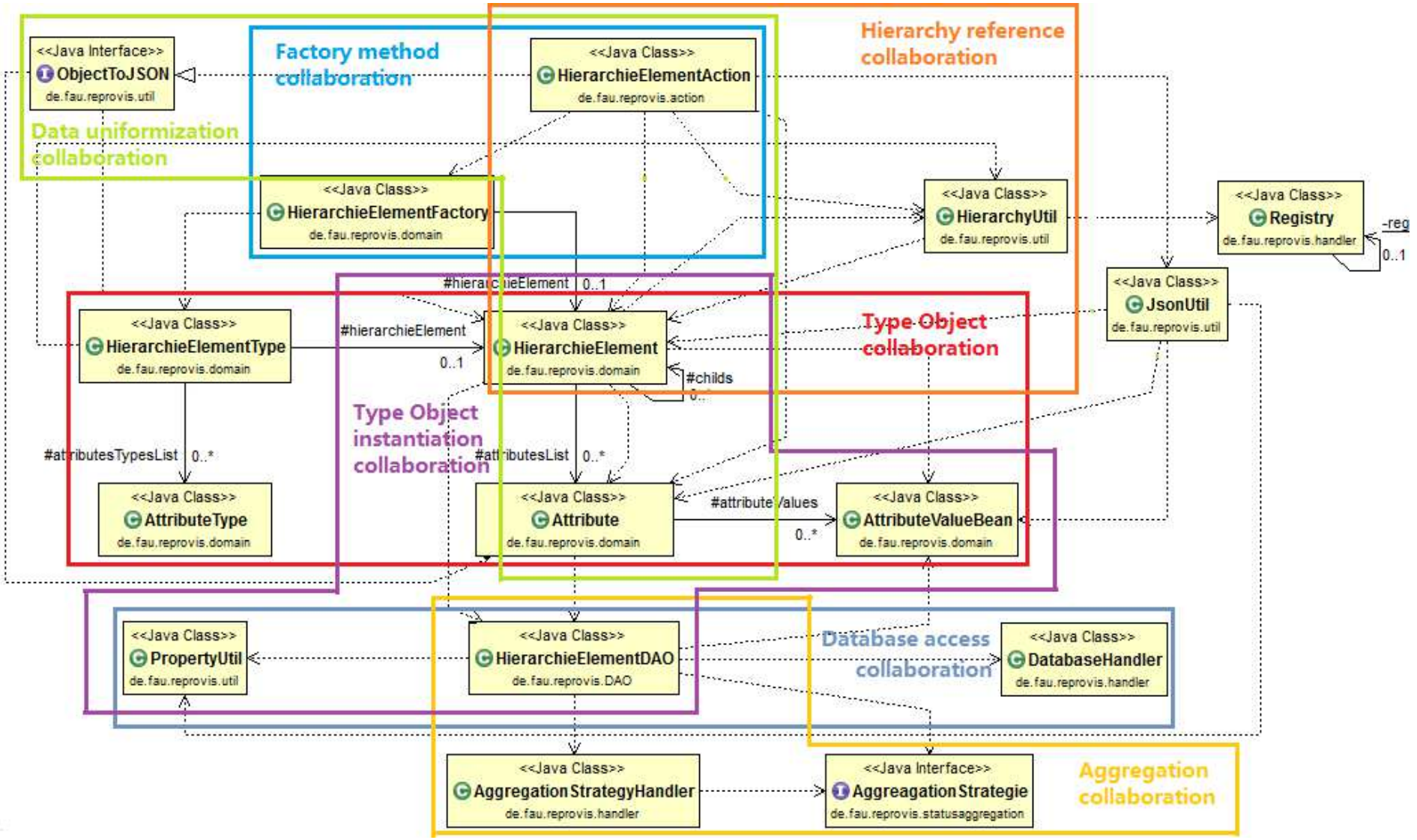


Figure 5-16 Class collaborations in DOM-SC and DOM-SF implementation

6 ANALYSIS AND DISCUSSION

Based on the modeling and implementation results demonstrated in Chapter 5, in this section the cognitive complexity in UML models (class diagram) and software complexity in the resulting software artifacts are examined and further discussed to discover to what extent different modeling approaches (OOM and DOM) influence the resulting software artifacts in the context of adapting to complexity patterns (SC and SF) in organizational modeling.

The analytical data sampling is based on the following foundations:

For cognitive complexity, the data sampling focuses on the major classes in the domain models and application models as shown in Figure 5-3, Figure 5-4, Figure 5-6, Figure 5-8, Figure 5-13, and Figure 5-14. The reason is that other supporting classes receive little influence from the core models after the application was modeled and implemented with different modeling approaches (OOM or DOM) for different complexity patterns (SC or SF) .

For software complexity, the data sampling is conducted with the identical software complexity analysis tools. Non-domain related classes could be included in the calculation. However, even so, this happens to all implementations with different modeling approaches (OOM or DOM) and complexity patterns (SC or SF). Therefore, the bias of sampling is limited.

The sampling perspectives for both evaluations are designed to be different, since one is for model evaluation and the other one is for source code assessment. Nevertheless, the inter-relationship between cognitive complexity, software complexity and complexity patterns in organizational modeling is the key and can be indicated.

6.1 Analysis of the Solutions with the OOM Approach

The identical business case was modeled and implemented with the OOM approach on two complexity patterns (SC and SF). The analyses of OOM-SC and OOM-SF are conducted in the following sub-sections

6.1.1 Analysis of OOM for the structure-consistent pattern

6.1.1.1 Cognitive complexity in UML models

Size dimension

According to Figure 5-3, in order to create a domain object, two classes were employed: a super class HierarchieElementBean and a specific domain class XXXBean. However, in the entire application, according to Figure 5-4, for all seven domains (Global, Factory, Hall, Line,

Location, Device and Component), eight classes are used. In addition, for database accessibility, one XXXDAO class for each domain and a super DAO class are required. Taking AggregationStrategyHandler, ObjectToJSON, DatabaseHandler classes into account, 25 classes need to be designed for the main framework of the application model. Therefore:

$$CCsize = \sum NC = 25$$

Coupling dimension

In UML class diagrams, coupling includes dependency, realization, association, aggregation, composition, inheritance, and so on. According to Figure 5-4, all Action and DAO classes are afferent/import-coupling to domain (bean) classes; all domain (bean) classes are efferent/export-coupling themselves.

$$CCcoupling = \sum (NICD + NXCD) = 14 + 8 = 22$$

Inheritance scale dimension

The inheritance dimension indicates the scale and structural complexity in the core domain models with a product of both horizontal and vertical dimensions. HierarchieElementBean has seven child classes and the depth (starts from 1) of this inheritance is one; therefore:

$$CCinheritance\ scale = NOC * DIT = 7 * 2 = 14$$

Complexity dimension

The complexity refers to the potential cyclomatic complexity in a class, and is calculated as the sum of weighted methods in the domain classes.

$$CCcomplexity = \sum NCM = 1 + 3 + 3 + 3 + 3 + 2 + 2 + 1 = 18$$

The results of cognitive complexity in UML models with OOM for the SC pattern are collected as shown in Table 6-1.

Table 6-1 Statistics of cognitive complexity dimensions in OOM-SC solution

Dimension	Value
Size	25
Dependency	22
Inheritance scale	14
Complexity	18

6.1.1.2 Software complexity in the implementation

The software complexity is calculated by automatic tools of CodePro and STAN as introduced in Section 2.4.2.2. The calculation scope is different from the cognitive complexity (focusing on the most influencing domain classes), but applied to the entire application source code. The results are collected in Table 6-2.

Table 6-2 Statistics of software complexity dimensions in OOM-SC solution

Metric name	OOM-SC
Size	= **55**
Number of Classes*	55
Number of Methods	462
Lines of Code (LOC)	3656
Average Number of Fields Per Unit	3.52
Average Number of Methods Per Unit	8.40
Coupling	= **33**
Afferent Coupling (Import Coupling) in domain classes*	23
Efferent Coupling (Export Coupling) in domain classes*	10
Coupling Between Objects (CBO)	3.36
Inheritance	2.7x7 = **18.9**
(Average) Depth of Inheritance Hierarchy*	2.7
(Average) Number of Sub Units / Number of Children*	7
Complexity	** 698/10 = **69.8**
Cyclomatic Complexity (CC)	1.43
Weighted Methods (WM) *	698

*selected factors representing the respective dimension in a graphical evaluation framework
**divided by 10 for statistical convenience

6.1.2 Analysis of OOM for the structure-flexible pattern

6.1.2.1 Cognitive complexity in UML models

Size dimension

According to Figure 5-6, in order to create a domain object, two classes were employed: a super class HierarchieElementBean and a specific domain class XXXBean. However, in the entire application, according to Figure 5-8, for all seven domains (Global, Factory, Hall, Line, Location, Device and Component), eight classes are used. In addition, for database accessibility, one XXXDAO class for each domain and a super DAO class are required. Taking AggregationStrategyHandler, ObjectToJSON, DatabaseHandler, and HierarchyUtil classes into account, 26 classes need to be designed for the main framework of the application model. Therefore:

$$CCsize = \sum NC = 26$$

Coupling dimension

According to Figure 5-8, all Action, DAO and the hierarch reference utility classes are accessibility to domain (bean) classes; all domain (bean) classes are efferent/export-coupling themselves.

$$CCcoupling = \sum (NICD + NXCD) = 15 + 8 = 23$$

Inheritance scale dimension

The inheritance dimension indicates the scale and structural complexity in the core domain models with a product of both horizontal and vertical dimensions. HierarchieElementBean has seven child classes and the depth (starts from 1) of this inheritance is one; therefore:

$$CCinheritance\ scale = NOC * DIT = 7 * 2 = 14$$

Although, in some cases, the width (number of children) of the inheritance tree should be a sub set of the full width, due to the since the reduction of child hierarchy elements, which means $1 \leq \mathrm{NOC} \leq 7$ and 14 is the maximum value for the inheritance dimension in the OOM-SF solution.

Complexity dimension

The complexity refers to the potential cyclomatic complexity in a class, and is calculated as the sum of weighted methods in domain classes.

$$CCcomplexity = \sum NWMD = 1 + 2 + 2 + 2 + 2 + 2 + 2 + 1 = 14$$

The results of cognitive complexity in UML models with OOM for the SF pattern are collected as shown in Table 6-3.

Table 6-3 Statistics of cognitive complexity dimensions in OOM-SF solution

Dimension	Value
Size	26
Dependency	23
Inheritance scale	14
Complexity	14

6.1.2.2 Software complexity in the implementation

The software complexity calculation method is identical for all solutions, as introduced in Section 6.1.1.2. The results are collected as below:

Table 6-4 Statistics of software complexity dimensions in OOM-SF solution

Metric name	OOM-SF
Size	= **59**
Number of Classes*	59
Number of Methods	493
Lines of Code (LOC)	4001
Average Number of Fields Per Unit	3.38
Average Number of Methods Per Unit	8.35
Coupling	= **36**
Afferent Coupling (Import Coupling) in domain classes*	26
Efferent Coupling (Export Coupling) in domain classes*	10
Coupling Between Objects (CBO)	3.1
Inheritance	2.59x7 = **18.13**
(Average) Depth of Inheritance Hierarchy*	2.59
(Average) Number of Sub Units / Number of Children*	7
Complexity	** 768/10 = **76.8**
Cyclomatic Complexity (CC)	1.48
Weighted Methods (WM) *	768

*selected factors representing the respective dimension in a graphical evaluation framework

**divided by 10 for statistical convenience

6.2 Analysis of Solutions with the DOM Approach

As presented in Section 5.3, the modeling and implementation results are identical for DOM-SC and DOM-SF solutions, since the both consistent and flexible structure patterns are defined and can be modified in the external configuration file, rather than the rather source code. Therefore, the analysis of them is conducted together.

6.2.1 Analysis of DOM for complexity patterns

6.2.1.1 Cognitive complexity in UML models

Size dimension

According to Figure 5-13, in order to create a domain object, four classes were employed: two type classes (HierarchieElementType and AttributeType) and two instance classes (HierarchieElement and Attribute). Meanwhile, according to Figure 5-14, in the entire application the four classes are also used for all potential domains. In addition, for database accessibility, only one HierarchieElementDAO class is required. Taking AggregationStrategyHandler, AggreagationStrategie, ObjectToJSON, DatabaseHandler, and PropertyUtil classes and so on into account, 15 classes need to be designed for the main framework of the application model. Therefore,

$$CCsize = \sum NC = 15$$

Coupling dimension

According to Figure 5-14, the Action and DAO classes are afferent or import-coupling to domain classes; all Type Object associated domain classes (Attribute, AttributeType, AttributeValueBean, HierarchieElement, HierarchieElementType) are efferent or export-coupling themselves. Therefore,

$$CCcoupling = \sum (NICD + NXCD) = 2 + 5 = 7$$

Inheritance scale dimension

Only one class is responsible for being a super or sub class in the Type Object, the HierarchieElement class. It has unpredicted nested child elements with the assistance of configuration files, which cannot count in class diagrams. The depth (starts from 1) of this inheritance at least can be recognized at a parent-child relationship with depth 2. However, the width (number of children) of the inheritance tree should be a sub-set of the full width, due to the reduction of child hierarchy elements, which means $0 \leq \mathrm{NOC} \leq 7$. Therefore,

$$CCinheritance = NOC * DIT = 0 * 2 = 0$$

Complexity dimension

The complexity refers to the potential cyclomatic complexity in a class, and is calculated as the sum of weighted methods in domain classes (Attribute, AttributeType, AttributeValueBean, HierarchieElement, HierarchieElementType).

$$CCcomplexity = \sum NWMD = 2 + 1 + 1 + 5 + 1 = 10$$

The results of cognitive complexity in UML models with DOM for the SC pattern are collected as shown in Table 6-5.

Table 6-5 Statistics of cognitive complexity dimensions in DOM-SC and DOM-SF solutions

Dimension	Value
Size	15
Dependency	7
Inheritance scale	0
Complexity	10

6.2.1.2 Software complexity in the implementation

The software complexity calculation method is identical for all solutions, as introduced in Section 6.1.1.2. The results are collected in Table 6-6.

Table 6-6 Statistics of software complexity dimensions in DOM-SC and DOM-SF solutions

Metric name	DOM-SC
Size	= **43**
Number of Classes*	43
Number of Methods	179
Lines of Code (LOC)	1860
Average Number of Fields Per Unit	1.55
Average Number of Methods Per Unit	4.16
Coupling	13+7 = **20**
Afferent Coupling (Import Coupling) in domain classes*	13
Efferent Coupling (Export Coupling) in domain classes*	7
Coupling Between Objects (CBO)	1.76
Inheritance	2.39x0 = **0**
(Average) Depth of Inheritance Hierarchy*	2.39
(Average) Number of Sub Units / Number of Children*	0
Complexity	** 385/10 = **38.5**
Cyclomatic Complexity (CC)	1.11
Weighted Methods (WM) *	385

*selected factors representing the respective dimension in a graphical evaluation framework

**divided by 10 for statistical convenience

6.3 Discussion

This section discusses about the impact of different modeling approaches (OOM and DOM) to the cognitive complexity in models and the software complexity in the resulting software, where complexity patterns (SC and SF) were applied to each modeling approach. Therefore, the discussion scope consists of OOM-SC, OOM-SF, DOM-SC, and DOM-SF, aligned with the modeling and implementation results in Chapter 5.

6.3.1 Impact of modeling approaches on cognitive complexity

Complexity patterns in organizational modeling were used as the second influencing factor in solution design and implementation, after modeling approaches. Consequently, the results in the four dimensions (size, coupling, inheritance scale and complexity) of cognitive complexity in this research demonstrated interesting and significant changes, as shown in Table 6-7.

Table 6-7 Comparison of cognitive complexity in solutions

Dimension	OOM-SC	OOM-SF	DOM-SC	DOM-SF
CCsize	25	26	15	15
CCcoupling	22	23	7	7
CCinheritance scale	14	14	0	0
CCcomplexity	18	14	10	10

6.3.1.1 Impact of OOM approach in cognitive complexity dimensions

With OOM, in order to adapt to the structure-flexible pattern, the size dimension (number of classes) increased, because a centralized hierarchy control class was required to achieve the flexibility and coordination after removing the tight coupling between domain classes.

Due to this centralized hierarchy control, more coupling relationships were created between action classes, hierarchy control and domain classes.

Since in the OOM-SF solution the target was to remove the fixed hierarchical relationships between domain classes and action classes, no changes happened to the inheritance scale.

After the fixed relationships between actions and domain classes were removed, the internal cyclomatic complexity in the major classes in the application model was reduced.

It suggests that the overall cognitive complexity when using OOM for complexity patterns in organizational modeling demonstrated limited advantages and some disadvantages.

6.3.1.2 Impact of DOM approach in cognitive complexity dimensions

As introduced in Section 5.3.2, adapting from a structure-consistent pattern to a structure-flexible pattern with the DOM approach was solved by configuring the dynamic hierarchical relationships in the configuration file, rather than in the application models. There was no change on the class level. It demonstrated that the DOM approach is a complexity pattern independent approach.

Therefore, in each dimension of cognitive complexity, DOM certainly demonstrated the identical performance for both complexity patterns. In other words, with DOM, the application can adapt to a flexible organizational structure from a consistent one easily, without extra modification to the source code.

6.3.1.3 Comparison between OOM and DOM's impact to cognitive complexity

Effects in complexity and volume reduction

For either of the complexity patterns, the DOM approach demonstrated higher cognitive complexity reduction in all dimensions compared with the OOM approach:

- Reduced low reused domain classes (by 40%)
- Removed low reused couplings between action and domain classes (by 70%)
- Eliminated explicit inheritance in the domain models (by 100% in inheritance scale)
- Avoided unnecessary cyclomatic complexity in the classes (by approximately 60% in average)

A visualized comparison between OOM and DOM's impact on cognitive complexity applying complexity patterns in organizational modeling is presented in Figure 6-1.

Improvement in flexibility

The results suggest that the dynamic characteristics in DOM created the flexibility to construct organizational relationships in the models far beyond OOM.

DOM does not only support the flexibility of modifying relationships between organization hierarchies, but also the flexibility of changing the attributes in domain classes, while OOM can only achieve the former one with extra and heavy effort.

In addition, the model designers and developers can use the existing knowledge and understanding about the application model in further development without considering new complexity in the models. All of modification can be done in the configuration files, which also enables a runtime modification without programming or recompiling in the resulting application.

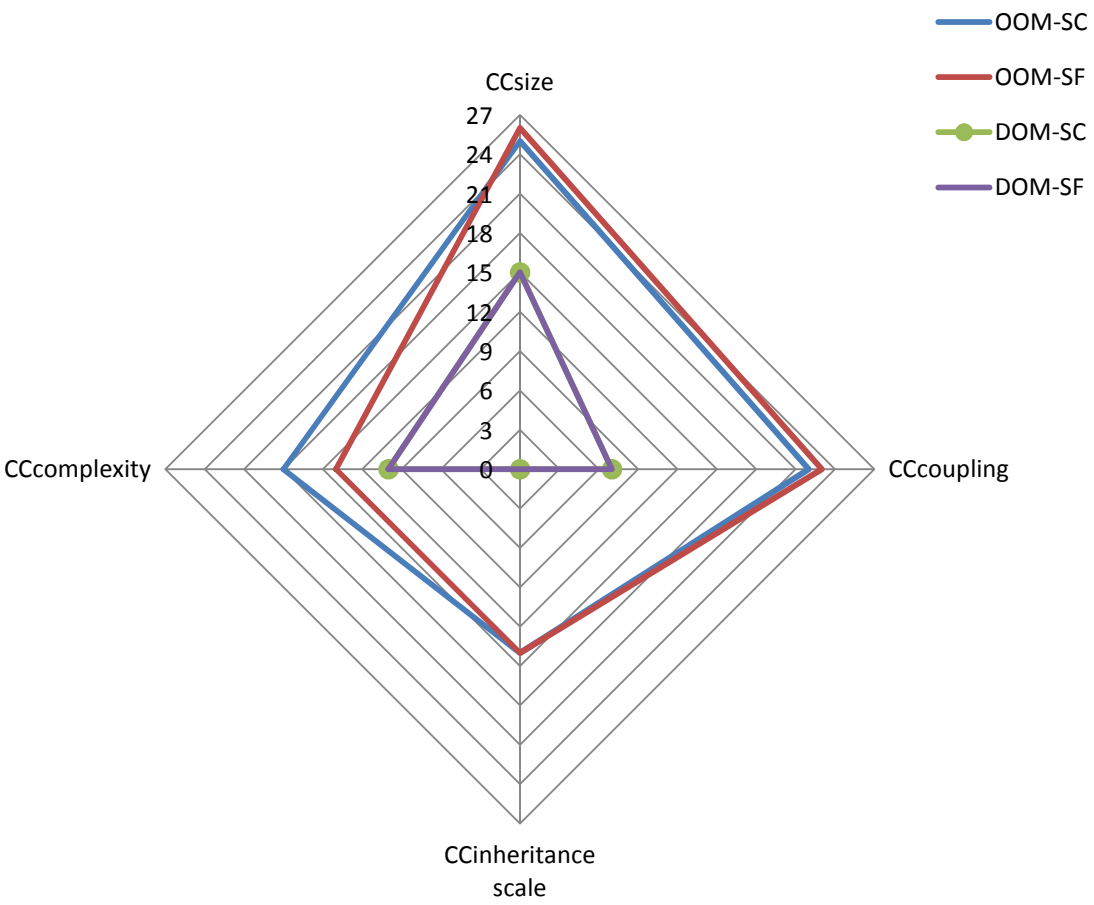


Figure 6-1 Comparison of cognitive complexity in solutions

6.3.2 Impact of modeling approaches on software complexity

The results of software complexity demonstrated a more precise evaluation in the resulting source code after modeling. The software complexity is also examined in four dimensions (size, coupling, inheritance and complexity), although their definitions (see Section 2.4.2) are different from the ones for cognitive complexity, but they represent the major drivers in each dimension for software complexity. The detailed results of software complexity in this research demonstrated interesting and significant changes, as shown in Table 6-8.

6.3.2.1 Impact of the OOM approach in software complexity dimensions

With OOM, aligned with cognitive complexity in models, in order to adapt to a structure-flexible pattern from a structure-consistent pattern, more classes were required to be implemented to support the centralized hierarchy control.

Unsurprisingly as a consequence in the size dimension, more classes were implemented containing more necessary methods (XML configuration retrieval, hierarchy coordination and so on). The number of classes increased by 7%, the number of methods increased by 6.7% and lines of code increased by 9.4%. On the contrary, the average number of fields /

methods per class decreased. This indicates that the newly added classes bring lower cohesion and lower reusability in the application. Therefore, in the size dimension, the OOM approach increased the complexity when adapting to a structure-flexible pattern.

In the coupling dimension, while adapting to the structure-flexible pattern, more afferent (import) couplings were required to receive hierarchical structure data after referencing the hierarchical control classes and increased by 13%. This increase came after removing old afferent couplings and adding new ones. At the same time, the efferent (export) couplings did not increase, since the structure-flexible pattern does not require a domain class to be used in another one, but emphasizes the reduction of coupling between organizational hierarchy elements. On the other hand, the coupling between objects decreased since the structure-flexible pattern requires the fixed coupling between the hierarchy objects to be moved to the configuration file. Therefore, the OOM approach showed limited advantages for controlling the coupling in the application while achieving flexibility in the organizational structure.

In the inheritance dimension, no changes happened to the inheritance relationships in the application, while adapting to the structure-flexible pattern.

Table 6-8 Detailed comparison of software complexity in solutions

Metric names	OOM-SC	OOM-SF	DOM-SC	DOM-SF
Size				
Number of Classes*	55	59	43	43
Number of Methods	462	493	179	179
Lines of Code (LOC)	3656	4001	1860	1860
Average Number of Fields Per Unit	3.52	3.38	1.55	1.55
Average Number of Methods Per Unit	8.40	8.35	4.16	4.16
Coupling				
Afferent Coupling (Import Coupling) in domain classes*	23	26	13	13
Efferent Coupling (Export Coupling) in domain classes*	10	10	7	7
Coupling Between Objects (CBO)	3.36	3.1	1.76	1.76
Inheritance				
(Average) Depth of Inheritance Hierarchy*	2.7	2.7	2.39	2.39
(Average) Number of Sub Units / Number of Children*	7	7	0	0
Complexity				
Cyclomatic Complexity (CC)	1.43	1.48	1.11	1.11
Weighted Methods (WM) *	698	768	385	385

*selected factors representing the respective dimension in a graphical evaluation framework

In the complexity dimension, both cyclomatic complexity and weighted methods in the application increased. The extra implementation for adapting to the structure-flexible pattern required more complicated and even cyclomatic methods. The number of weighted

methods increased by 10%. This indicates significant complexity on the method level. Therefore, OOM is inclined to increase the complexity in the methods when applied to adapting the flexible organizational structure, which increases the difficulty for understanding, testing and maintenance.

6.3.2.2 Impact of the DOM approach in software complexity dimensions

As introduced in Section 5.3.2, adapting from a structure-consistent pattern to a structure-flexible pattern, the solution with the DOM approach was solved by configuring the dynamic hierarchical relationships in the configuration file, rather than in the source code.

Therefore, in each dimension of the software complexity, the DOM approach demonstrated the identical software complexity in the solutions for both complexity patterns. In other words, with DOM the application can adapt to either a consistent organizational structure or a flexible organizational structure without modifying the source code. In addition, a runtime modification in the application also becomes possible.

Although, it is important to emphasize that the data consistency in the database before and after changing the organizational structure will be damaged. Additional tools are necessary to support examining data consistency, clear and reconstruct data aligned with the changes. These extra costs were not included in the software complexity, since the solution could vary and/or with using existing data management tools.

6.3.2.3 Comparison between OOM and DOM's impact on software complexity

Effects in complexity and volume reduction

According to Table 6-8, to either of the complexity patterns, the DOM approach demonstrated higher cognitive complexity reduction in all dimensions compared with the OOM approach:

- Reduced classes (by approximately 24% in average)
- Reduced number of methods (by approximately 62% in average)
- Reduced lines of code (by approximately 51% in average)
- Reduced import coupling (depending on other classes to receive inputs) (by approximately 46% in average)
- Reduced export coupling (a class is used in another class as an object) (by approximately 33% in average)
- Reduced overall objects coupling (by approximately 45% in average)
- Reduced depth of inheritance hierarchy (by approximately 33%)
- Reduced number of children (by 100%)
- Reduced cyclomatic complexity in the classes (by approximately 24% in average)
- Reduced weighted methods in the classes (by approximately 47% in average)

The above figures suggest that DOM has significant advantages in not only software complexity reduction but also implementation volume reduction.

According to the evaluation framework in this research, the representatives of each dimension in the software complexity evaluation are selected from the detailed results of software complexity in solutions (presented in Table 6-2, Table 6-4 and Table 6-6), as shown in Table 6-9.

Table 6-9 Comparison of software complexity in solutions

Dimension	OOM-SC	OOM-SF	DOM-SC	DOM-SF
SWCsize	55	59	43	43
SWCcoupling	33	36	20	20
SWCinheritance	18.9	18.13	0	0
SWCcomplexity*	69.8	76.8	38.5	38.5

*divided from the original values for statistical convenient

A visualized comparison between OOM and DOM's impact on cognitive complexity applying complexity patterns in organizational modeling is presented in Figure 6-2.

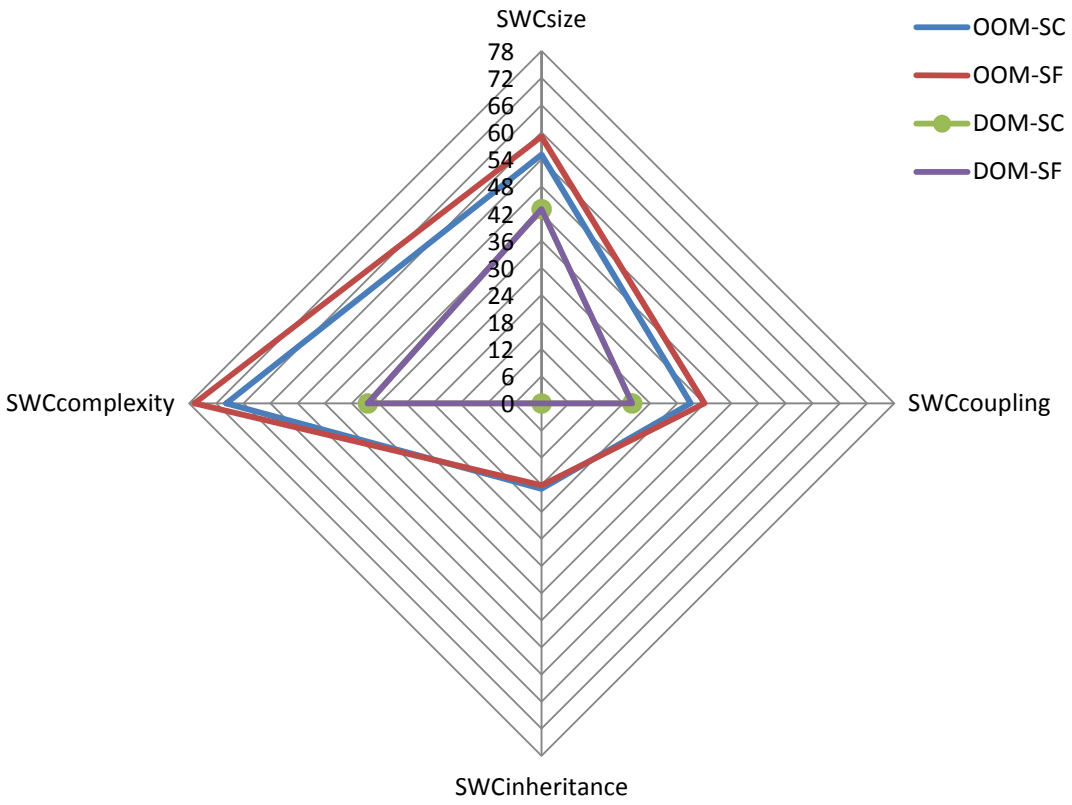


Figure 6-2 Comparison of software complexity in solutions

Improvement in flexibility

The results suggest that the dynamic characteristics in DOM created the flexibility to construct organizational relationships in the models far beyond OOM.

DOM does not only support the flexibility of modifying relationships between organization hierarchies, but also the flexibility of changing the attributes in domain classes, while OOM can only achieve the former one with extra and heavy efforts.

In addition, the model designers and developers can use the existing knowledge and understandings about the application model in further development without considering new complexity in the models. All of the modification can be done in the configuration files, which also enables a runtime modification without programming or recompiling in the resulting application.

6.3.3 Consistency comparison between cognitive and software complexity

Figure 6-3 demonstrated the consistency of changes between cognitive complexity and software complexity in all dimensions while employing different modeling approaches (OOM and DOM) to the identical business case.

This consistency also suggests that the design of the evaluation framework in this research has a robust validity for assessing the complexity in organizational modeling.

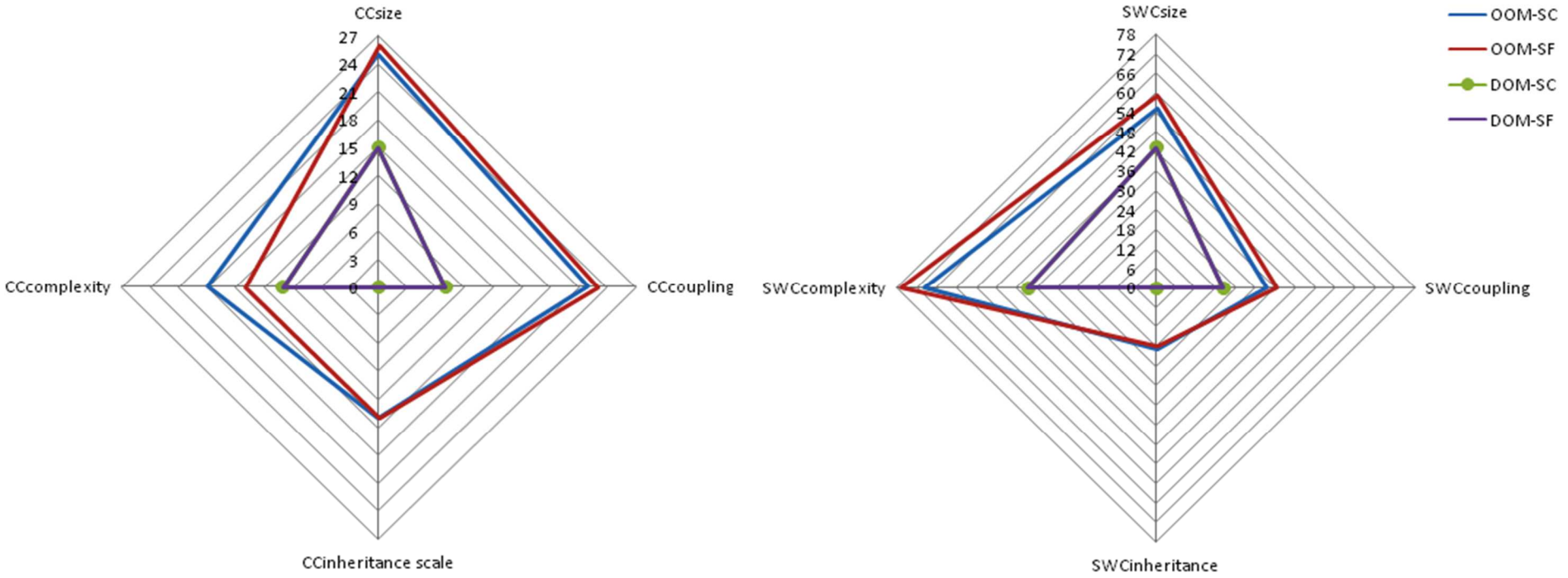


Figure 6-3 Consistency comparison between cognitive complexity and software complexity

7 CONCLUSIONS AND FUTURE WORK

7.1 Summary and Conclusions

This master thesis examined the inherent complexity nature in both the business and technical domains in organizational modeling. A distinctive division of complexity patterns, structure-consistent (SC) and structure-flexible (SF) was created to represent the problem domains in organizational modeling. Two modeling approaches, Object-Oriented Modeling (OOM) and Dynamic Object Model (DOM), were employed to model and implement the identical business case, resource visualization project at Audi AG, and find a better solution for the research topic. The framework of research development was established with the combinations of modeling approaches and complexity patterns, as OOM-SC, OOM-SF, DOM-SC and DOM-SF.

Furthermore, in order to give a solid answer to the research question, quantitative evaluation and analyses were conducted, after modeling and implementing all target applications. The application models provided solid foundations for cognitive complexity analysis for UML class diagrams regarding the modeling complexity and quality. The application source code provided precise foundations for software complexity analysis for the resulting software artifacts after modeling.

The evaluation for the complexity in UML models was designed with a four-dimension (size, coupling, inheritance scale and complexity) framework. Aligned with that, an evaluation framework for software complexity was designed with four dimensions on size, coupling, inheritance and complexity. Based on the statistical results, the comparisons were conducted from both the dimension of complexity patterns within a modeling approach as well as the approach-wide dimension.

The results demonstrated a number of advantages in employing DOM to handle complexity in organizational modeling, which was verified with the resulting software artifacts. Meanwhile, there are challenges that also need to be addressed in future work.

Below is the summary of conclusions drawn upon the comparison of modeling approaches for handling complexity in organizational modeling:

- Compared to the OOM approach, the DOM approach demonstrated a complexity pattern independent characteristic. With the DOM approach, an application can adapt to both consistent and flexible organizational structures by only modifying configuration files without programming.

- The DOM approach presented significant volume reduction effects in all evaluation dimensions, over the OOM approach, regardless of UML models or software implementations.
- With respect to the complexity reduction effects, the DOM approach demonstrated far more effective performance in modeling and coding beyond OOM, in all evaluation dimensions.

- The DOM approach created more flexibility for the end users to adapt to the dynamic business than OOM approach.

- The DOM approach transferred the modeling complexity and implementation complexity from classes into the configuration files, which became a hidden and transparent risk for end users and technical experts. New techniques and associated knowledge management tools are necessary to address the hidden complexity in configurations.

- Dynamic architecture design creates the power to make the abstraction of domains independent and standardize the processes to build relationships.

7.2 Future Work

Multiple modules and larger scales

This research was based on a single functional module in a system. In a multiple and large scale application, the complexity with DOM in organizational modeling and its software artifacts might appear in an exponential growth.

DOM data consistency tools

There is an issue with the data consistency after changing the hierarchy relationships and attributes for each Type in the Type Object. This consistency check, data movement and data (re)constriction, should be assisted with an extra module or tools.

DOM experts

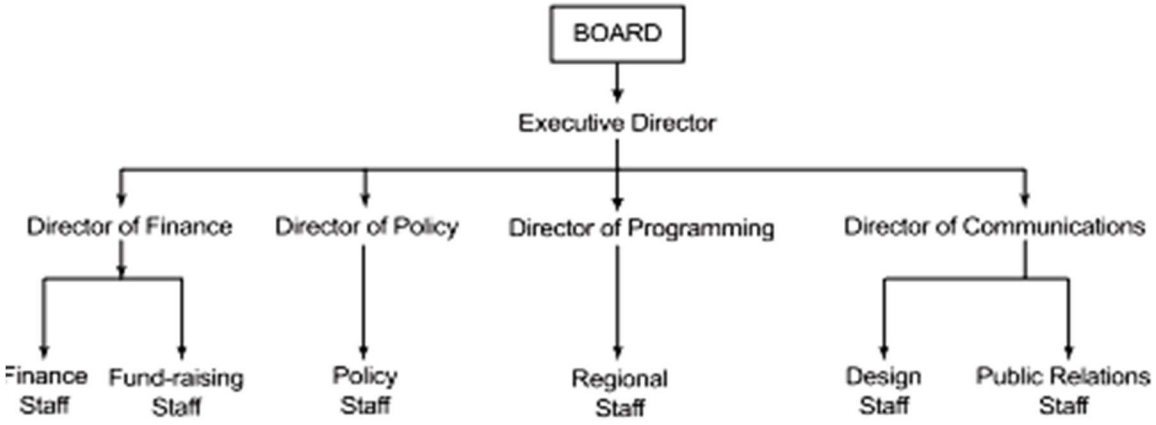
The major difficulty for applying DOM in real life might exist in the unfamiliarity of this approach, which is often reported in development (distinct object lifecycle), testing (implicit real-time relationship between class type and data), and maintenance (knowledge maintenance in the configuration and modeling).

Knowledge management tools

Since the modeling and implementation complexity were transferred from classes into the configuration files employing the DOM approach, knowledge management tools are essential for the users, modelers and developers to understand the application.

Appendix A – Types of organizational structure

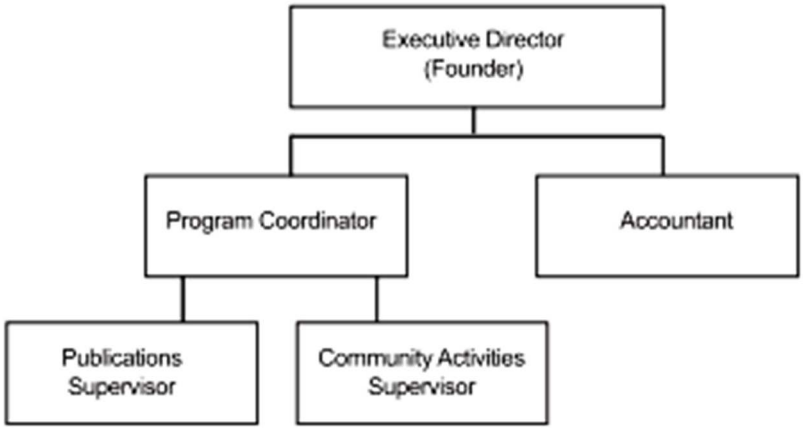
Traditional Hierachy



Team Structure

Network Organization

Emerging or Launch/Growth Structure



(Source: pathfinder.org)

References

Fowler, M. (1997). *Analysis Patterns, Reusable Object Models* (1st ed.). Addison-Wesley Professional.

Yoder, J., Foote, B., Riehle, D., Tilman, M. (1998). Metadata and active object models. *Addendum to the 1998 proceedings of the conference on Object-oriented programming, systems, languages, and applications*. Vancouver, BC, 0.22-A22,

Johnson, R. (1998). *The Dynamic Object Model*. Retrieved February 23, 2015, from http://st-www.cs.illinois.edu/users/johnson/papers/dom/

Yoder, J., & Johnson, R. (2002). The Adaptive Object-Model Architectural Style. *Software Architecture*. 3-27.

Riehle, D., Tilman, M., & Johnson, R. (2005). Dynamic Object Model. In *Pattern Languages of Program Design 5*. MA: Addison-Wesley.

Pressman, R. (2010). *Software engineering: A practitioner's approach* (7th ed.). McGraw-Hill.

Lee, G., & Xia, W. (2002). Development of a Measure to Assess the Complexity of Information Systems Development Projects. *Twenty-Third International Conference on Information Systems.*

Xia, W., & Lee, G. (2003). Complexity of Information Systems Development Projects: Conceptualization and Measurement Development. *Journal of Management nformation Systems, 22(1)*, 45-83.

Scott, J., & Vessey, I. (2002). Managing risks in enterprise systems implementations. *Communications of the ACM - Supporting Community and Building Social Capital, 45(4)*, 74-81.

Anderson, P. (1999). Complexity Theory and Organization Science. *Organization Science, 10(3)*, 216-232.

Daft, R. (1992). *Organization theory and design* (4th ed.). South-Western.

Benbya, H., & Mckelvey, B. (2006). Toward a complexity theory of information systems development. *Information Technology & People Info Technology & People, 19(1)*, 12-34.

Simon, H. (1996). *The sciences of the artificial* (3rd ed.). Cambridge: MIT Press.

McMillan, E. (2004). *Complexity, Organizations and Change*. Routledge.

Gruhn, V., & Laue, R. (2006). Complexity metrics for business process models. In: *9th International Conference on Business Information Systems,* Austria

La Rosa, M., Ter Hofstede, A., Wohed, P., Reijers, H., Mendling, J., & Van Der Aalst, W. (2011). Managing Process Model Complexity via Concrete Syntax Modifications. *IEEE Transactions on Industrial Informatics,* 255-265.

Cardoso, J. (2008). Business Process control-Flow complexity: Metric, Evaluation, and Validation. *International Journal of Web Services Research, 5(2*), 49-76.

Schneider, A., Zec, M., & Matthes, F. (2014). Adopting Notions of Complexity for Enterprise Architecture Management. *Enterprise Architecture and Organizational Success, Twentieth Americas Conference on Information Systems*, Savannah.

Kinnunen, M. (2006). Complexity Measures for System Architecture Models. *System Design and Management Program: Massachusetts Institute of Technology*.

Land, F., & Mitleton-Kelly, E. (1999.) Complexity and information systems. In *Davis, Gordon, (ed.). The Blackwell Encyclopedic Dictionary of Management Information Systems*. Blackwell.

Mitleton-Kelly, E., & Land, F. (2004). Complexity & information systems. *Blackwell Encyclopedia of Management*.

Mitleton-Kelly, E. (2003). Ten Principles of Complexity & Enabling Infrastructures. *Complex Systems & Evolutionary Perspectives of Organisations: The Application of Complexity Theory to Organisations*.

Kluth, A., Jäger, J., Schatz, A., & Bauernhansl, T. (2014). Method for a Systematic Evaluation of

Advanced Complexity Management Maturity. *Procedia CIRP*, 69-74.

Lu, Y., Luo, L., Wang, H., Le, Y., & Shi, Q. (2014). Measurement model of project complexity for large-scale projects from task and organization perspective. *International Journal of Project Management,* 610-622.

Tegarden, D., Sheetz, S., & Monarchi, D. (1992). A Software Complexity Model of Object-oriented Systems. *Decision Support System*s, 241-262.

Booch, G. (1991). *Object oriented design with application*s. Redwood City, CA: Benjamin/Cummings Publishing Company.

Booch, G., Rumbaugh, J., & Jacobson, I. (1999). T*he Unified Modeling Language: User Guide*. Addison-Wesley.

McCabe, T. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering SE-2(4)*, 308-320.

Halstead, M. (1977). *Elements of software science*. NY: Elsevier.

De Bakker, K., Boonstra, A. & Wortmann, H. (2010) Does risk management contribute to IT project success? A meta-analysis of empirical evidence. *International Journal of Project Management, 28(5)*, 493–503.

Cao, Q., Gu, V., & Thompson, M. (2012). Using Complexity Measures to Evaluate Software Development Projects: A Nonparametric Approach. *The Engineering Economist, 57*, 274-283.

Pizzi, N. (2011). Mapping Software Metrics to Module Complexity: A Pattern Classification Approach. *Journal of Software Engineering and Applications, 4*, 426-432.

Baccarini, D. (1996). The concept of project complexity—a review. *International Journal of Project Management, 14(4)*, 201-204.

Cellier, F.E. (1996), Object-Oriented Modeling: Means for Dealing With System Complexity,

Proc. *15th Benelux Meeting on Systems and Control, Mierlo*, The Netherlands, pp.53-64.

Rumbaugh, J. (1991). *Object-oriented modeling and design*. NY: Prentice Hall.

Jacobson, I. (1992). *Object-oriented software engineering: A use case driven approach*. NY: ACM Press.

Chen, Yih-Chang (2001) *Empirical modelling for participative business process reengineering*. PhD thesis, University of Warwick.

Bruegge, B., & Dutoit, A. (2010). *Object-oriented Software Engineering: Using UML, patterns and Java* (3rd ed.). NJ: Prentice Hall; Pearson.

Kobryn, C. (1999). UML 2001: A standardization odyssey. *Communications of the ACM, 42*, 29-37.

Podgorelec, V., Herieko, M., & Juric, M. (2004). Assessing Software Complexity from UML Using Fractal Complexity Measure. *Second IEEE International Conference on Computational Cybernetics*, 237-242.

De Champeaux, D., Lea, D., & Faure, P. (1992). The Process of Object-Oriented Design. *OOPSLA '92 Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, 45-62.

Siau, K., & Cao, Q. (2001). Unified Modeling Language (UML) — A Complexity Analysis. *Journal of Database Management, 12(1)*, 26-34.

Siau, K. and Rossi, M. (1998). Evaluating Information Modeling Methods — A Review. *Thirty-first Hawaii International Conference on System Sciences (HICSS-31), 5*, 314-322.

Meyer, B. (1997). *Object-oriented software construction* (2nd ed.). Indianapolis, IN: Pearson Professional Education.

Lientz, B., & Swanson, E. (1980). *Software Maintenance Management: A Study of the*

Maintenance of Computer Application Software in 487 Data Processing Organizations. MA: Addison-Wesley.

Genero, M., Piattini, M., & Calero, C. (2000). Early measures for UML class diagrams. *L' Objet, 6(4)*.

Claver-Cortés, E., Pertusa-Ortega, E., & Molina-Azorín, J. (2012). Characteristics of organizational structure relating to hybrid competitive strategy: Implications for performance. *Journal of Business Research*, 993-1002.

Burton, R., & Obel, B. (2004). *Strategic Organizational Diagnosis and Design: The Dynamics of Fit* (3rd ed.). New York: Springer US.

Fredrickson, J. (1986). The Strategic Decision Process and Organizational Structure. *Academy of Management Review, 11(2)*, 280-297.

Robbins, S. (1990). *Organization theory: Structure, design, and applications* (3rd ed.). N.J.: Prentice Hall.

Van Dijke, A., & Scheele, R. (2013). A Structured Approach to Reducing SAP Complexity. *Consolideren of Excelleren: Haal Meer Waarde Uit Uw ERP-systeem*, 36-40.

Schwandt, A. (2009). *Measuring organizational complexity and its impact on organizational performance – A comprehensive conceptual model and empirical study*. Technische Universität Berlin.

Brooks, F. No Silver Bullet — Essence and Accident in Software Engineering. *Proceedings of the IFIP Tenth World Computing Conference*, 1069–1076.

Leukert, P., Vollmer, A., Alliet, B., & Reeves, M. (2011). IT Complexity metrics – How do you, measure up?. The Capital Markets Company. NV.

Newstrom, J. (2014). *Organizational Behavior: Human Behavior at Work* (14th ed.). New York, NY: McGraw-Hill.

SDMetrics - *UML design size, coupling and complexity*. (n.d.). Retrieved March 5, 2015, from

http://www.sdmetrics.com/DProp.html#Complex

Gruhn, V., & Laue, R. (2006). Adopting the Cognitive Complexity Measure for Business Process Models. *5th IEEE International Conference on Cognitive Informatics*, 236-241.

Silva, D., Weerawarna, N., Kuruppu, K., Ellepola, N., & Kodagoda, N. (2013). Applicability of three cognitive complexity metrics. *8th International Conference on Computer Science & Education*, 573-578.

Rossi, M., & Brinkkemper, S. (1996). Complexity metrics for systems development methods and techniques. *Information Systems, 21(2)*, 209-227.

Badri, M., Badri, L., & Touré, F. (2009). Empirical Analysis of Object-Oriented Design Metrics:Towards a New Metric Using Control Flow Paths and Probabilities. *JOT The Journal of Object Technology, 8(6)*, 123-142.

Li, W., & Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of Systems and Software, 23*, 111-122.

Basili, V., Briand, L., & Melo, W. (1996). A validation of object-oriented design metrics as quality indicators. *IIEEE Trans. Software Eng. IEEE Transactions on Software Engineering, 22(10)*, 751-761.

Briand, L., Daly, J., & Wurs, J. (1997). The dimensions of coupling in object-oriented systems, *Proceedings of OOPSLA*.

Briand, L., & Wüst, J. (2001). Integrating Scenario-based and Measurement-based Software Product Assessment. *Journal of Systems and Software, 59(1)*, 3-22.

Briand, L., Wüst, J., Daly, J., & Porter, D. (2000). Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software, 51(3)*, 245-273.

Emam, K., Benlarbi, S., Goel, N., & Rai, S. (2001). The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, *27(7)*, 630-650.

STAN4J. (2008). *Structure Analysis for Java*. Retrieved April 14, 2015, from http://stan4j.com/

CodePro Analytix. (n.d.). Retrieved May 28, 2015, from https://developers.google.com/java-dev-tools/codepro/doc/features/features

IBM. Retrieved May 28, 2015, from http://www-01.ibm.com/support/knowledgecenter/SS8PJ7_9.1.1/com.ibm.xtools.modeler.doc/topics/rreltyp.html

Larman, C. (2003). *Applying UML and Design Patterns, An introduction to object-oriented analysis and design and the unified process* (2nd ed.). NJ: Prentice Hall.

A. Botchkarev, P. Finnigan (2014) *Complexity in the Context of Systems Approach to Project Management*. Retrieved 2015, from: http://arxiv.org/ftp/arxiv/papers/1412/1412.1027.pdf

Recker, J. (2005). Conceptual Model Evaluation. Towards more Paradigmatic Rigor. In *J. Castro and E. Teniente (eds.): Proceedings of the CAiSE'05 Workshops – Volume I*. FEUP, Porto, Portugal, 569-580.

Chidamber, S., & Kemerer, C. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering, 20(6*), 476-493.

Kascheck, R., & Mayr, H. (1998). Characteristics of Object Oriented Modeling Methods. *EMISA Forum, 8(2),* 10-39.

Steele, J., Son, Y., & Wysk, R. (2001). Resource Modeling for the Integration of the Manufacturing Enterprise. *Journal of Manufacturing Systems, 19(6)*, 407-427.

Foote, B., & Yoder, J. (1998). Metadata and Active Object-Models, Proceedings of Plop98. *Technical Report #wucs-98-25*, Department of Computer Science., Washington University.

Yoder, J., Balaguer, F., & Johnson, R. (2001). Architecture and Design of Adaptive Object-Models. *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2001)*. Tampa, FL.

Yoder, J., & Johnson, R. (2002). The Adaptive Object-Model Architectural Style., I*FIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance (WICSA 2002)*, Montréal, QC

Riehle, D. (1997). A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose. *Ubilab Technical Report 97-1-1*. Zürich.

Johnson, R., & Wolf, R. (1998). *Type Object. Pattern Languages of Program Design 3*. Addison-Wesley.

Johnson, R. & Oakes, J. (n.d.). *The User-Defined Product Framework.* Retrieved March 1, 2015, from: http://st-www.cs.illinois.edu/users/johnson/papers/udp/

Genero, M., Piattini, M., & Calero, C. (2005). A Survey of Metrics for UML Class Diagrams. *Journal of Object Technology, 4(9)*. 59-92.

Thompson, J., Buchbinder, S., & Shanks, N. (n.d.). *An Overview of Healthcare Management.* Jones and Bartlett Learning, LLC. Retrieved March 7, 2015 from: http://samples.jbpub.com/9780763790868/90868_CH01_FINAL_WithoutCropWith.pdf

Löffler, C., Westkämper, E. & Unger , K. (2012) Changeability in Structure Planning of Automotive Manufacturing. *Procedia CIRP, 3,* 167-172.