ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

THESIS PRESENTED TO
ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
Ph. D.

BY
Sébastien ADAM

SYSTEMATIC INFERENCE OF THE CONTEXT OF UTILIZATION OF THE DESIGN KNOWLEDGE BY USING A REFERENCE MODEL

MONTREAL, JUNE 07, 2016

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Mr. Alain Abran, Thesis Supervisor
Department of LOG/TI at École de technologie supérieure

Mrs. Ghizlane El Boussaidi, Thesis Co-supervisor
Department of LOG/TI at École de technologie supérieure

Mrs. Catherine Laporte, President of the jury
École de technologie supérieure

Mr. Christian Desrosiers, Chair, Board of Examiners
Department of LOG/TI at École de technologie supérieure

Mr. Hamid Mcheick, External Evaluator
Université du Québec à Chicoutimi - UQAC

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON MAY 12, 2016

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGMENTS

I would like to thank my family and friends for the support they provided me through my doctoral project. In particular, I acknowledge my wife for her love, encouragement, efforts, assistance, and sacrifices.

I would like to express my gratitude to my supervisors, the professors Alain Abran and Ghizlane El Boussaidi, for their expertise and understanding that added to my doctoral experience. A very special thanks to Alain for his assistance in writing reports, articles, and this thesis, and for inspiring me kindness, promptness, and motivation over many ways. A very special thanks to Ghizlane for his assistance in defining the focus of my project. I thank both of you for our exchanges that enriched my doctoral experience.

I express also my gratitude to the other members of my board of examiners.

SYSTEMATIC INFERENCE OF THE CONTEXT OF UTILIZATION OF THE DESIGN KNOWLEDGE BY USING A REFERENCE MODEL

Sébastien ADAM

ABSTRACT

Software engineering is concerned with systematic procedures for obtaining software that meets the customer's expectations. Taking into account the impacts of the software design artifacts when designing the architecture of a software system is critical, but it remains a major challenge. The contribution of the architecture to achieve or not targeted objectives results from the utilization in the architecture and in the detailed design of an appropriate set of software design artifacts (SDAs) such as styles, tactics, and design patterns. The styles and design patterns organize the design decisions, and the tactics are building blocks of these styles and patterns. The software designer is responsible for applying tactics, patterns, and styles that best achieve the targeted objectives. This requires understanding what objectives are affected by the styles, patterns, and tactics applied, identifying which styles and patterns best support a set of tactics, and discerning which set of design decisions produces the best balance across the targeted objectives. The software designers encounter at least three problems when discerning the design context and measuring the effects of a style, a design pattern, or a tactic on a set of objectives:

1. the representation schemes usually used to describe the SDAs force the software designers to extract from textual descriptions the finer-grained decisions and the related explications about how they impact the objectives;
2. the explanations of these impacts are described in terms of characteristics of quality, and they are not precisely detailed and supported with contextual design rationale;
3. the effects of a design decision are not quantified but merely discussed textually making it hard to evaluate which decision is better than others in a particular context.

This research project provides a reference model of software design artifacts for describing the styles, patterns, and tactics using a set of software design artifacts and arguments. This reference model and the related techniques will support designers to systematically analyze styles, tactics, and design patterns for inferring the order of treatment of the related issues from given sets of software design artifacts and contextualized arguments.

Keywords: Design knowledge management, Design artifact, Reference model, and Design decision support system.

INFÉRENCE SYSTÉMATIQUE DU CONTEXTE D'UTILISATION DES CONNAISSANCES DE CONCEPTION À L'AIDE D'UN MODÈLE DE RÉFÉRENCE

Sébastien ADAM

RÉSUMÉ

La définition d'activités systématiques pour développer des logiciels satisfaisants les attentes des parties prenantes est une préoccupation du génie logiciel. Concevoir l'architecture et la conception détaillée d'un logiciel est une activité critique. Atteindre les objectifs ciblés résulte de l'utilisation d'un ensemble approprié d'artéfacts de conception tels que les styles, les tactiques et les patrons. Les styles et les patrons organisent les décisions de conception. Les tactiques sont des blocs de construction des styles et des patrons. Le concepteur du logiciel est responsable d'appliquer les styles, les patrons et les tactiques pour permettre d'atteindre au mieux les objectifs ciblés. Pour satisfaire les parties prenantes, le concepteur doit:

- comprendre les objectifs affectés par les styles, les patrons et les tactiques appliqués;
- identifier les styles et les patrons pour supporter au mieux l'ensemble des tactiques;
- prendre des décisions pour produire le meilleur équilibre entre les objectifs ciblés; et
- comprendre les effets des artéfacts de conceptions utilisés.

Le concepteur du logiciel a quelques préoccupations lors de la mesure des effets des styles, des patrons et des tactiques sur un ensemble d'objectifs ciblés, incluant:

- les schémas de représentations textuelles ou graphiques généralement utilisés pour décrire les styles, les patrons et les tactiques obligent le concepteur à extraire les décisions de conception et les connaissances de leurs effets sur le logiciel;
- les effets sont décrits en termes de caractéristiques de qualité et ne sont pas précisément détaillés et soutenus par des explications contextuelles; et
- les effets ne sont pas quantifiés, ce qui rend difficile d'évaluer quelle décision est mieux qu'une autre dans un contexte particulier.

Ce projet de recherche propose un modèle de référence pour décrire les styles, les patrons et les tactiques à l'aide d'un ensemble d'artéfacts de conception et d'arguments. Le modèle de référence et les techniques connexes sont proposés pour soutenir le concepteur lors de l'analyse des styles, des patrons et des tactiques utilisés dans un contexte particulier. La méthodologie proposée permet d'inférer à partir d'un ensemble d'arguments contextualisés l'ordre de traitement des problèmes liés à l'utilisation des artéfacts de conception.

Mots-clés: gestion des connaissances de conception, artéfact de conception, modèle de référence, système d'aide à la décision de conception.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF ABREVIATIONS

ABAS	Attribute-Based Architectural Styles
ADD	Attribute-Driven Design
ADL	Architecture Description Language
CS	Classification Scheme
DD	Design Decision
DK	Design Knowledge
EA	Enterprise Architecture
GOF	Gang-Of-Four
IEC	International Electrotechnical Commission (www.iec.org)
IEEE	Institute of Electrical and Electronics designers (www.ieee.org)
ISO	International Organization for Standardization (www.iso.org)
OCL	Object Constraint Language
OWL	Web Ontology Language
SAD	Software Architecture Description
SAM	Software Architecture Mapping
SDA	Software Design Artifact
SDD	Software Design Description
SEI	Software Engineering Institute (www.sei.cmu.edu)
SIS	Software-Intensive System
SSM	Software Structures Map
SWRL	Semantic Web Rule Language
UML	Unified Modeling Language (www.uml.org)

INTRODUCTION

Software Engineering and Software Architecture

Software engineering is defined as the systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software to improve its quality; i.e., its ability to support the stakeholders' needs [IEEE610]. In particular, software designers improve the level of software quality by educating themselves and using bodies of knowledge, standards, best practices, and certification mechanisms [Kruc06].

The software designer's community considers the design process, the software architecture it provides, and the reuse of design knowledge as fundamental levers for quality [Iso42010, Ovas10, Kim09, Scot09, Shaw06, Bert05, Bass03, Bach03, Clem03]. In the "Guide to the Software Engineering Body of Knowledge – SWEBOK", software architecture design is a key sub-area of software engineering [Abra01, Garl00a].

Software architecture describes design elements from which software products are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns [Medv00]. Several studies (e.g., [SEI11, Bass03, Shaw96, Garl95]) list benefits of an appropriate architecture and conclude that the quality of a software system depends largely on the quality of its architecture. During the development process, the systematic development of complex software architecture that supports the specified quality requirements remains a major challenge.

The Design Knowledge (DK) Base

Much knowledge and support for software design is provided by the software architecture literature including catalogs of styles, design patterns, and tactics, design decisions and quality models (e.g. [Iso42010, Iso25000, Zimm12, Ovas10, Zimm09, Kim09, Scot09, Bass03, Clem03, Gran02, Gamm95]).

Design decisions, styles, design patterns, and tactics are used to explicitly describe reusable design knowledge (DK). An architectural style and a design pattern organize the design elements in a way that has been recognized as a proven solution to a design problem. Each design decision, style, design pattern, and tactic may promote or disadvantage one or more quality requirements. Software designers describe and use the DK in various ways for building software that support the quality requirements. The inadequate usage of DK may cause significant impact on software quality when the most prioritized quality requirements are disadvantaged. The software designers need to add many details to produce an implementable design, which may reduce the claimed benefits of the styles, design patterns, or tactics.

For example, Figure 1.1 presents the *Client-Server* architectural style and Figure 1.2 presents the *increase available resources* architectural tactic. To maintain the system level of performance when the number of clients increases, the *Client-Server* style facilitates the addition of a second server. The addition of a server and its interactions with the clients may reduce the level of security of the system, which has more possible points of attack for intruders. In such a context, the architects need to take decisions among a large space of solutions [Scot09].

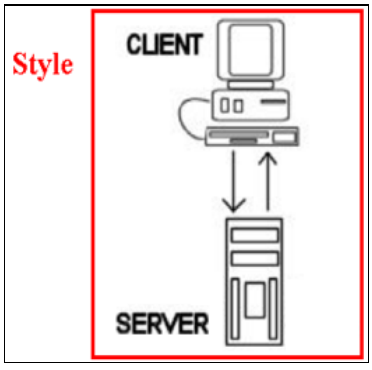


Figure 1.1 The *client-server* architectural style

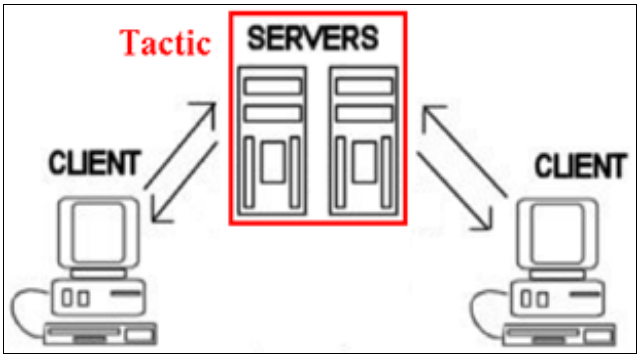


Figure 1.2 The *increase available resources* architectural tactic

The Software Designer Role

The software designers execute many activities described in [Bass03] for designing a software product with proven characteristics of quality. To take into account multiple objectives during the design process, the software designers select and prioritize the quality requirements that are architecturally relevant. The most prioritized quality requirements are called the architectural drivers. The software designers use the architectural drivers to make design decisions for designing parts of the architecture. Each decision may affect one (sensitivity point) or more (tradeoff point) quality requirements. The software designers have to evaluate alternatives and make tradeoffs among conflicting decisions in order to reduce risks and determine sets of design decisions that best support the project objectives.

Research Problem

There are multiple sources (users, marketing, etc.) and categories (constraints, business drivers, technical limits, etc.) of project objectives relating to the software design process. The evaluation of many design decisions is made before any software is built. A lack of detail about either the problem or solution space may lead software designers to inappropriate or suboptimal design decisions [Kozi11]. The software designers employ the DK base to understand, tailor, and describe alternate designs that have proven to be useful for previous projects with comparable contexts and project objectives.

Styles, design patterns, and tactics are mostly described in textual format. They may be complex and their interactions are not always evident. The software designers need to add significant amounts of details to produce an implementable design, which may reduce the claimed benefits of the styles, patterns, and tactics. To take full advantage of accumulated knowledge, the software designers need frameworks and tools to manage the DK and relate it to the decisions taken and software design artifacts (SDAs) used for designing the software.

The software designers need to discern:

1. the SDAs, issues, and arguments that best describe the design context;
2. the design decisions including styles, tactics, and design patterns that produce the best balance across orthogonal, complementary, and antagonistic objectives ; and
3. the objectives that are affected by the styles, design patterns, and tactics used.

The objectives include the quality requirements that should be precisely defined since the architecture is built to support them. Systems often have different sets of requirements for different modes of execution. Many of the particular quality requirements are in conflict (e.g., adding efficiency is often realized at the price of portability and maintainability) and the qualitative nature of these requirements makes the appropriate levels of satisfaction difficult to clarify. Evaluating the impacts of a set of design decisions on a set of objectives is a complex activity. In particular, the design context may be complex considering the nature of the objectives, the number of SDAs, including styles, design patterns, and tactics available, and their relationships with the quality requirements.

Many organizations maintain design decisions, SDAs, and tailored information items in a DK base to help document control, development, and maintenance activities. Most of the models, methods, and tools provide limited analysis capability and views in the DK base [Ovas10, Bach07, Bass05, Tyre05, Bass03, Clem03]. Reusing the DK contributes to the design capability of the organizations and accrues returns of investments in designing the software and building the systems [Bass03]. By managing the artifacts produced for designing software-intensive systems, the software designers may reuse the resulting DK during the development and decision processes of current and future projects.

The software designers encounter at least three problems when analyzing the design context and the effects of the design decisions on a set of objectives:

1. The representation schemes and design document templates usually used to share the DK (i.e., design decisions, styles, patterns, and tactics) force the software designers to extract and infer the finer-grained decisions, context knowledge, and explications about how the SDAs impact the objectives from the textual descriptions.
2. The issues and the impacts are merely discussed in terms of quality characteristics making it hard to evaluate which decision is better than others in a particular context.
3. The techniques, models, and tools that support the DK management usually aim at sharing the design decisions using templates and do not support sharing finer-grained SDAs and other activities such as acquiring, reusing, evaluating, and maintaining DK.

Research Question, Hypothesis, and Approach

This doctoral project is characterized in terms of the research question it investigates, the research approach it adopts, and the criteria by which it evaluate the results. From our point of view, the software design artifacts (SDAs) constitute the explicit DK. For this doctoral project, the research question was: "what is a good DK management approach (i.e., DK model and techniques) for supporting the software designers when inferring and describing SDAs and DK related to particular decisions points of the design process?"

The design decisions are useful for aggregating cohesive sets of SDAs at particular decisions points during the design process. The research's main hypothesis was that a valuable DK management approach should provide techniques for:

- managing the finer-grained SDAs that relate to each decision point;
- describing the design decisions, issues, and impacts using the finer-grained SDAs; and
- supporting the inference of the SDAs and issues related to a particular decision point.

Finally, a valuable approach should be assessed using the requirements for DK management defined in the literature.

The research hypothesis has been verified using a conceptual, theoretical, and qualitative empirical research approach. The conceptual part was required to identify and clarify the meaning, relationships, descriptions, and use of the finer-grained SDAs in to order to make specific proposals about how to manage them. The theoretical part was required to develop a reference model and techniques for managing the DK and the related SDAs, issues, and impacts. The qualitative empirical part was required to demonstrate the reliability and usability of the proposed reference model and techniques for managing the DK.

Research Goal, Research Sub-Goals and Research Objectives

The research goal was to develop supports for guiding the software designers when inferring the context of the design decisions during the design process. To tackle this research goal the following research strategy has been chosen:

1. identify and understand the descriptions of SDAs from catalogs of styles, design patterns, and tactics, and software architecture and software design documents.
2. structure the SDAs in a reference model for DK management; and
3. develop techniques for acquiring DK and supporting the software designers when using the reference model and DK base of SDAs during the design process.

This research project has permitted to develop an approach using the existing works, of [Iso42010, Iso12207, Zimm12, Ovas10, Zimm09, Kim09, Scot09, Gran08, Bach07, Kruc06, Tang06, Bass05, Tyre05, Bass03, Clem02, Abra01, Gamm95].

The research goal includes the following sub-goals (A-C) and research objectives (1-7).

Sub-goal A. Develop a description format for describing the SDAs and relationships that are used by the software designers during the design process.

Research objective 1: Establish descriptive criteria for describing the finer-grained SDAs that compose the styles, design patterns, and tactics.

Research objective 2: Establish exclusive criteria for classifying these finer-grained SDAs.

Research procedure: This research objective requires studying and representing with descriptive and exclusive criteria the finer-grained SDAs. The resulting criteria will be used to develop a description format for describing the SDAs, including styles, design patterns, and tactics.

Sub-goal B. Define a reference model for representing the SDAs and their relationships.

Research objective 3: Develop a structure for creating semantic networks of SDAs that may be used for describing styles, design patterns, and tactics.

Research procedure: A semantic network is used as a form of knowledge representation; it represents semantic relations between concepts. Relating finer-grained SDAs that relate to the styles, design patterns, and tactics in a semantic network will allow to represent the SDAs as instantiations of semantic networks. The styles, design patterns, and tactics will instantiate common nodes and relations for constituting the semantic networks using aggregations of SDAs. The reference model should be suited for representing the node types (e.g., Tactic) and relations (e.g., Mandatory or Exclusive-or) of the semantic networks.

The related works on tactics, styles, and design patterns should be the starting points for populating the design knowledge base using the descriptions given in [Bass03, Clem03, Gran02, Gamm95]. Such aggregations of artifacts should make discernible every part of the styles, design patterns, and tactics descriptions, instead of using the textual format that obscures significant information.

Research objective 4: Define an argument format for describing the impacts of particular utilizations of the SDAs.

Research procedure: Describing the arguments that relate to the utilization of the SDAs will allow to populate a design knowledge base of common issues and arguments. The styles, design patterns, and tactics will relate to common issues and arguments.

Sub-goal C. Systematize the utilization of the reference model and descriptions formats.

Research objective 5: Establish a technique and work instructions that help the software designers populate a design knowledge base using the reference model, and the descriptions of styles, design patterns, and tactics.

Research objective 6: Establish a technique and work instructions that help the software designers populate a design knowledge base of arguments that relate to particular utilizations of SDAs.

Research objective 7: Establish a technique and work instructions that help the software designers analyze the impacts of particular utilizations of the SDAs.

Originality and Expected Benefits

Many studies have proposed different models, techniques, and tools to describe and reuse the design knowledge (DK) [Ovas10, Kim09, Scot09, Harr08, Bach05, Bass03, Clem03]. However, previous studies have underlined the importance of having a reference model (RM) and the related techniques that can be used for managing the DK (see Section 2.4.3) using the finer-grained SDAs related to the design decisions, design patterns, tactics, and styles. This research is a step toward understanding, describing, and reusing the design decisions, tactics, design patterns, and styles, and their relationships to other SDAs of the DK base.

Five research deliverables were produced for this research project:

- a Reference Model (RM) for describing the SDAs, issues, and arguments;
- a Design Knowledge (DK) base of tactics, patterns, and styles knowledge;
- a technique for populating a DK base of SDAs using the RM;
- a technique for populating a DK base of arguments using the RM;
- a technique for reusing the DK base while designing.

Research Methodology

Figure 1.3 presents the phases, activities, and outcomes of the research methodology that have been executed. The phases are subdivided into lists of tasks and detailed in the next subsections.

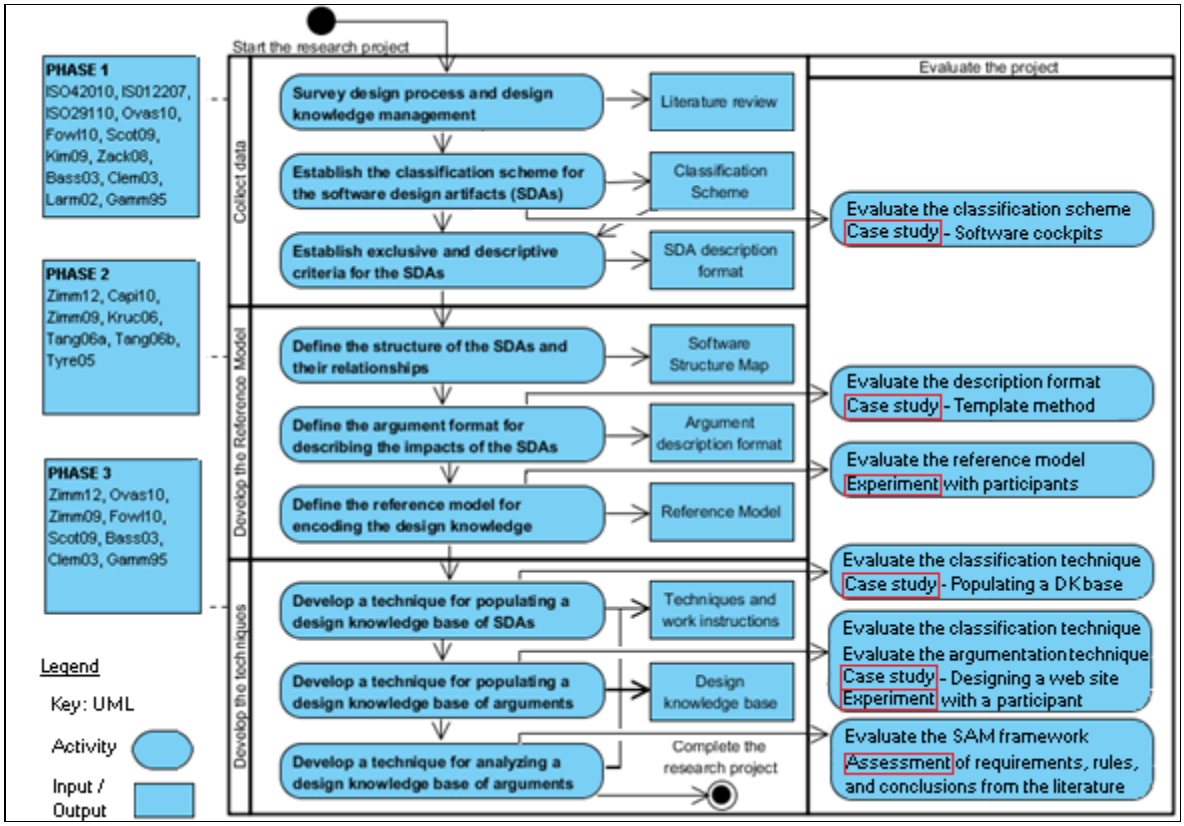


Figure 1.3 Activity diagram of the research methodology

Phase 1 – Collect Data

Phase 1 aimed at collecting data about the software design artifacts (SDAs) and their relationships. The following research tasks have been executed:

- Study the literature review for the state-of-the-art on design approaches, software design artifacts (SDAs), and design knowledge (DK) management.
    - Analyze models, techniques, and tools relating to SDAs and DK management.
    - Analyze techniques and tools that use DK bases to support software designers.
- Summarize the state-of-the-art for design approaches, SDAs, and DK management.
- Identify the finer-grained SDAs that provide DK, including the SDAs that relate to the styles, design patterns, and tactics descriptions.
    - Identify the issues relating to DK management and SDAs.
- Establish the software architecture mapping (SAM) framework for classifying SDAs.
- Apply the SAM framework in a case study.

Outcomes: literature review (see Chapter 1), classification scheme and SDA description format (see Chapter 4), case study (see Chapter 3 and Appendix I).

Phase 2 – Develop the Reference Model

Phase 2 aimed at developing the reference model and convey the design knowledge extracted from the literature review. The following research tasks have been executed:

- Describe the node types and relations of the semantic network of SDAs.
- Describe the argument format and relations of the semantic network of arguments.
- Develop the concepts of the reference model that constitute the semantic networks.
- Case study: Apply the reference model to the description of the tactics, design patterns, and styles given in [Scot09, Bass03, Clem03, Gamm95].

Outcomes: SSM, argument description format, case study – see Chapter 3 and Appendix II.

Phase 3 – Develop techniques

Phase 3 aimed at developing the techniques for using the reference model. The following research tasks have been executed:

- Develop heuristics for identifying the SDAs.
- Develop a classification scheme and a decision tree for classifying the SDAs.
- Develop steps and work instructions that support using the reference model for populating a DK base of semantic networks of SDAs.
- Develop steps and work instructions that support using the reference model for populating a DK base of semantic networks of arguments.
- Develop steps and work instructions for using the DK base of related SDAs and arguments for analysis the impacts of utilization of the SDAs in project contexts.

Outcomes: classification technique (Chapter 4), argumentation technique (Chapter 5), analysis technique (Chapter 6), case study (Chapter 6).

Validation of Research Results

This section summarises the validation objectives and validation scope, and activities that have been conducted for validating the proposed Software Architecture Mapping (SAM) framework. As stated in [Zimm09], "research contributions in software engineering must be validated. A validation of the monetary value and business benefits such as opportunities to increase revenue or reduce cost would be required when creating a business case for the development of a commercial version of the proposed solution". Such an analysis is difficult to perform in practice [Zimm09] and was not a primary goal of the thesis validation. "The important validation objectives were to evaluate technical feasibility, to confirm the practical value for the target audience, and to evaluate the usability. Practical value and usability have been considered but remain to be evaluated in more details, i.e., whether software designer are willing and able to apply the SAM framework and whether such application is beneficial" [Zimm09].

Validation Activities

The case studies and experiments described in CHAPTER 3 and a self-assessment of requirements served as the primary validation activity types. The evaluation activities focused on confirming the key hypothesis that SDAs, SSMs, arguments, and views recur and can be modeled according to the reference model.

The case studies have been developed to evaluate the technical feasibility of the SAM framework concepts by creating:

- the SSMs of a framework in the context of software cockpits;
- the SSMs of a web site in the context of a web engineering project; and
- the SMMs of design patterns, styles, and tactics used in software engineering courses.

The experiments have been conducted with human participants for evaluating the reliability, efficiency, accuracy, and usability of the SAM framework.

1. For the first experiment, selected research results were proposed to participants for eliciting issues and describing arguments related to the design of a framework in the context of a detailed design course at ÉTS.
2. For the second experiment, selected requirements were proposed to a participant for creating the SSMs, eliciting the issues, and describing the arguments related to the design of a web site. In addition, the participant was required to create the SSM of the Template Method design pattern.

Another validation activity was to conduct self-experiments. For instance, previous versions of the SAM framework have been used for teaching the software architecture and detailed design courses at ÉTS. These activities helped to ensure that the developed reference model is applicable for software designers independent of their expertise and experience.

Finally, a self-assessment of requirements has been conducted using the requirements catalog described in section 2.6 as a source of validation criteria for the case studies. In addition, a prototype tool was developed to support the proposed approach.

The preliminary validation results were used to improve subsequent versions of the SAM framework and reference model. The validation produced sufficient evidence that the core concepts such as the reference model work in practice. The justification for conducting case studies is that the selected cases yielded a reasonable coverage of the concepts proposed in the SAM framework without causing unmanageable validation efforts for the involved researchers and the case study participants.

Thesis Organization

The thesis is organized as follows.

CHAPTER 1 presents the literature review. This chapter introduces the basic concepts (i.e., software architecture, styles, tactics, design patterns, and characteristics of quality) related to the software architecture design and detailed design. This literature review provides the synthesis on the works, challenges, and issues related to:

1) the representation of styles, tactics, and design pattern;
2) the activities and artifacts of the design process; and
3) design knowledge (DK) management.

It also provides a summary of the requirements for design methods and DK management. Finally, it describes the proposed structure of software design artifact (SDA) that is the basic concept for developing our approach. Appendix I describes the SDAs of the design process.

CHAPTER 2 presents the proposed Software Architecture Mapping (SAM) framework, SAM process, and the reference model for managing the SDAs. This chapter presents the arguments justifying the reference model and describes its limits. This chapter also compares the SAM framework with approaches in the literature review and presents its limits.

CHAPTER 3 presents seven examples of utilizations of the SAM framework, including five cases studies and two controlled experiments. These examples are the outputs of the validation process for evaluating the SAM framework. This chapter describes the software structures maps (SSMs) that were created, the SDAs that were classified, the arguments that were described for various academic and industrial contexts.

CHAPTER 4 presents the classification technique for creating a Software Structures Map (SSM) that classifies and relates the SDAs.

CHAPTER 5 presents the argumentation technique for relating the software design artifacts (SDAs) to the activities and dimensions they impact. This technique supports the elicitation of the issues and the description of the arguments that relate to the utilization of the SDAs.

CHAPTER 6 presents a technique for supporting the analysis of the arguments using multi-dimensional views. This technique supports the systematic inference of

- the order of treatment of arguments related to a context of application of SDAs; and
- the order of utilization of the related design knowledge.

CHAPTER 7 presents the conclusions, contributions, and future work of this research project.

CHAPTER 1

LITERATURE REVIEW

1.1 Basic Concepts

1.1.1 Software Development Approach

The international standard ISO 29110 on Lifecycle Profiles for Very Small Entities (VSEs) [Iso29110] propose a set of activities that constitute any software development approach for very small entities (i.e., enterprise, organizations, departments, or projects – up to 25 people). Such entities often implement software used in larger systems that require suppliers of high quality software. This standard decomposes the ***software development*** into two processes: project management (PM) and software implementation (SI). An output of the PM process is the project plan. The purpose of the SI process is the systematic execution of the analysis, design, construction, integration and tests activities for implementing software products according to the project plan and the specified requirements. The standard integrates practices based on the selection of standards elements from ISO 12207 on Software life cycle processes [Iso12207], and ISO 15289 on Content of life-cycle information items (documentation) [Iso15289] for the PM and SI processes. The focus of this thesis is on the SI process.

The execution of the SI process is driven by the project plan, which guides the execution of the software requirements analysis, software architectural and detailed design, software construction, software integration and test, and product delivery activities. The customer usually provides a statement of work as an input to the development approach. The PM process establishes a project plan based on the statement of work. The software designers use the project plan to perform the SI process in order to produce a software configuration that satisfies the customer and other stakeholders. More precisely, the focus of this thesis is on the software architecture design and detailed design.

1.1.2 Software Architecture Design and Detailed Design

Software requirements are defined, analyzed for correctness and testability, approved by the stakeholders, baselined, and communicated to software designers who will design the software. ***Software architecture design and detailed design*** are usually performed separately in order to describe the software components and connectors and their related software units. Subsequently, the software construction activity produces and tests the software units in order to verify their consistency with requirements and the design. The software is produced, including performing integration of software units, components and connectors. The verification and validation of the work products are performed in order to achieve consistency among the work products in each activity. Finally, the software configuration is integrated and delivered to the acquirer in accordance with the agreed requirements.

First in the design, the software architecture design (SAD) activity aims at developing the software architecture. The SAD produces the software components and connectors, their internal and external interfaces, and their topologies and semantics [Iso12207, Bass03]. The SAD aims at establishing consistency and traceability between software design and software requirements.

Second, the software detailed design (SDD) activity aims at developing a detailed design of each component and connector. The SDD describes the software units that compose each component and connector, including the external interfaces, structures, and sequences of interactions of the units [Iso12207]. The SDD aims at establishing consistency and traceability between detailed design, software requirements, and architectural design. For some authors [Bass03, Clem03], the software architecture is the most important deliverable; and the establishment of the consistency and traceability between detailed design, software requirements, and architectural design is a challenge.

1.1.3 Software Architecture

The literature defines the ***software architecture*** from many perspectives [SEI11, Bass03, Clem03, Medv00]:

1. a centerpiece artifact;
2. the set of principal design decisions about the software system;
3. the software design artifacts that pervade all major facets of software systems; the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns;
4. the software structures (e.g., modules; components and connectors; or allocation) of the system;
5. the set of elements such as modules (e.g., class), components and connectors (e.g., process), and their visible properties, behavior, and relationships (e.g., synchronization) in different structures and contexts (e.g., execution);
6. the set of principal design decisions about the software system that pervade all major facets of a software system, including structural, deployment, non-functional, evolution, and runtime concerns [Medv00]; and
7. the structures of the software system composed of several types of elements (e.g., class and process) and relationships (e.g., subdivision and synchronization).

There is no universally accepted definition of software architecture [Clem03], although it is a centerpiece artifact for software designers and its roots run deep in software engineering [SEI11, Abra01, Garl00a]. The software architecture concerns the externally visible portion of the elements; it represents an abstraction of the software system. It defines elements such as software components and connectors along with their relationships (topology), behavior, and visible properties. Externally visible properties refer to those assumptions other elements can make of a component or a connector [Clem03]. The software architecture describes how a component uses, is used, is connected to, and interacts with other elements in different contexts (e.g., compilation and execution).

As presented in [Clem03], the software architecture is what makes the sets of elements work together as a successful whole. [Perr92] proposes the following model: Software Architecture = {Elements, Form, Rationale}. This model refers to processing elements, data elements, and connecting elements. In [Bass03], the authors refine the model where Software Architecture = {Components and Connectors, Topology and Semantic, Rationale}.

1.1.4 Architectural Style and Tactic

Architectural styles are important software design artifacts that allow a software designer to reuse the collected wisdom of the architecture design community to solve recurring problems [Klei99]. Table 1.1 provides an example of a representation scheme used to describe the architectural styles. The representation scheme is used to describe the properties, components and connectors, relations, constraints, and strengths of the Layered System architectural style. An architectural style organizes the components and connectors of the software architecture [Bass03]. It describes known properties and patterns of data and control interaction among the components [Shaw96, Busc96]) to enable reuse and evolution of the design [Klei99]. It is a package of decisions that allows reasoning about the system design in terms of desired properties [Bass03]. It defines how to carry out the design and imposes constraints, semantics, vocabularies, and types for the components and connectors, along with qualitative reasoning about the strengths and weaknesses of the design [Klei99, Garl94]. An architectural style helps to interpret and analyze the software architecture.

Table 1.1 Example of a representation scheme used to describe the architectural styles

Properties
• A layer is an intermediary between software, hardware, or layers. • Each layer has a public interface that provides a cohesive set of services. • The public interface is more than just the API (assumptions, etc.). • A change in one layer does not affect the lower layers.
Components / Connectors
• One or more layers are defined. • The name, content, and cohesion scheme of each layer is specified. • The interface should not expose functions dependent on a particular platform.

Relations
• The inter/intra-relationships of each layer are specified. • The exceptions of each layer are specified.
Constraints
• The order of interaction is important (closest layer, layer bridging, etc.). • The use of the upper layers is prohibited (except: exceptions, data flow, etc.). • A layer cannot be above and below another layer at the same time. • Each software component is allocated to a single layer.
Strengths
• Permit system evolution. • Facilitate work assignment. • Favor reuse. • Manage complexity.

A style packages many tactics: ***architectural tactics*** are the building blocks of the software architecture and styles [Bass03]. They are among the first design decisions made during the development process. As presented in [Scot09], "architectural design is a complex search through a large space of possibilities, the use of tactics guides and constrains this search and makes it more tractable. A tactic suggests an analytic model for design and analysis, which may range from guidelines and heuristics to precise mathematical models".

A tactic may help to increase the level of quality of a system. Each tactic specifies how a quality attribute can be controlled through design decisions to achieve a response [Bass05]. For example, layering a system may increase maintainability by isolating a system from changes in the underlying platform. The layer style achieves this isolation by using many tactics described in [Scot09, Bach07, Bass03]: Semantic Coherence, Abstract Common Services, Use Encapsulation, Use an Intermediary, and Restrict Communication Paths. A tactic is a decision for tailoring software designs, styles, and design patterns.

1.1.5 Design Pattern

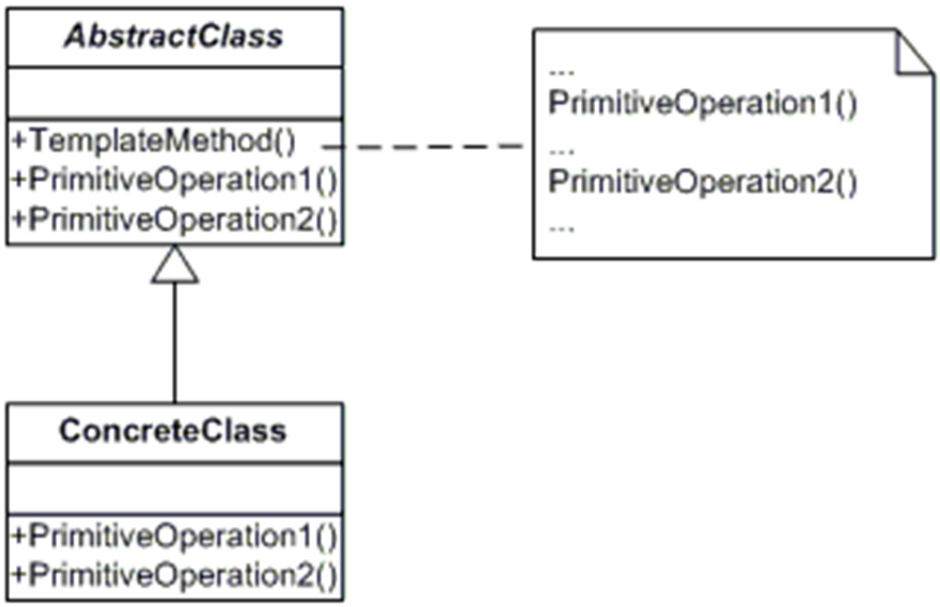
As mentioned in [Clem03], styles and ***design patterns*** "are alike in that both are catalogued partial design solutions, captured in practice, and must be instantiated and completed before application to an actual system". However, "a style tends to refer to a coarser grain of design solution than a pattern". Design patterns are usually finer-grained engineering artifacts that pact together a set of design decisions such as tactics and describe how they affect software product quality. Both design patterns and styles package a set of tactics.

A design pattern describes a general solution to a recurring problem that occurs under specific circumstances [Gran02, Gamm95]: it organizes the software modules in a way that has been recognized as a proven solution to a design problem. It describes the design concerns to be solved and the software design it proposes to address these concerns. The general solution has been recognized to be useful for designing previous software products with comparable characteristics of quality. However, a design pattern is only advice, and the software designers have to figure out how to apply it to his circumstances. Software designers may use the same design pattern many times, but it is never exactly the same project-specific solution. Table 1.2 provides an example of a representation scheme used to describe the design patterns. In general, a design pattern description has four essential elements [Gamm95]: the pattern name, the problem it addresses, the solution it provides, and the consequences it implies. The representation scheme is used to describe the Template Method design pattern.

Table 1.2 Example of a representation scheme used to describe the design patterns

Pattern Name • Template Method
Problem • Implement the invariant parts of an algorithm only once • Factor and localize common behavior to avoid code duplication • Control extension

Solution



Key: UML

Consequences

- Facilitate factoring out common behavior.
- Permit inverted control structure.
- Reduce duplication of code.
- Favor reuse.

A design pattern may be used to evaluate, before any software is built, how well the software design may support software product quality. To properly choose and apply the patterns and their related tactics, the software designers need to evaluate what characteristics of quality are affected by each of the design patterns and tactics used, and discern what set of patterns and tactics produces the best balance across the quality requirements. However, a pattern is complex and its interactions with other patterns and tactics are not always evident. In particular, the boundaries between the design patterns are fuzzy. As stated in [Fowl08], "designers can never just apply the solution blindly, which is why pattern tools have been such miserable failures". Design decisions have to be made in order to use a design pattern.

1.1.6 Characteristics of quality

There are many definitions of software quality, each reflecting a particular quality philosophy and approach. A quality model, as proposed in [Iso9126, Gali04, Bass03], may be used to make the meaning of the quality requirements and the level of quality more precise and easy to evaluate for a system. The ***characteristics of quality*** of a software system such as maintainability or performance are usually defined in a quality model by specifying the set of quality attributes required for the stakeholders' software acceptance [Bass03].

Software designers are required to analyze how their most relevant design decisions affect the characteristics of quality. The literature provides many definitions of software quality [Iso24765, Iso25000, Gali03]. In this thesis, software quality refers to the level to which a system, component, or process meets specified requirements as well as the needs and expectations of clients and users [Iso24765]. This vague definition is not useful in practice. Software designers may use quality models [Iso9126, Gali03] to make the meaning of quality requirements and the level of quality more precise and easy to evaluate.

The series of standards ISO 25000, also known as SQuaRE (System and Software Quality Requirements and Evaluation) [Iso25000], is a framework for the evaluation of software product quality. The international standard ISO 25010 on System and software quality models [Iso25010], and related works [Iso9126, Gali03, Bass03], recommend to hierarchically decompose software product quality into multiple characteristics.

The ISO 25010 standard provides a quality model that decomposes the software product quality into eight characteristics of quality: functionality, reliability, operability, performance efficiency, security, compatibility, maintainability, and transferability. Characteristics are further subdivided into subcharacteristics. Each subcharacteristic is further subdivided into a set of measures. For example, Table 1.3 presents the definitions of two subcharacteristics and three measures related to maintainability.

The standard proposes to use characteristics, subcharacteristics, and measures as a checklist of issues related to quality. Software designers use such measures, and other artifacts such as specific scenarios [Bass03], to specify the software quality requirements, evaluate the design decisions, and assess whether the resulting software meets the requirements or design objectives.

Table 1.3 presents three measures related to maintainability. Each measure (e.g., Activity recording) is defined by a formula (e.g., a / b) that contains the operands (e.g., a and b). The software designers assign values to the operands in order to reflect the design decisions that shape the software product. The software designers need to make design decisions that optimize a set of measures. Therefore, the design problem may be formulated by a set of threshold values assigned to the measures (e.g., $X_i = 0.7$). The threshold values are the targets software designers need to control when evaluating a software design. The evaluation results and the threshold values are compared to determine the software quality.

Table 1.3 Measures, formula, and operands for maintainability [Iso9126]

Sub-Characteristic	**Maintainability Subcharacteristic Definition**	
	Definition	***Measure, Formula, and Operands***
Analysability	Capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified	Activity recording X = A / B A = Number of implemented data login items B = Number of data items to be logged
		Readiness of diagnostic function (DF) X = A / B A = Number of implemented DFs B = Number of DFs required
Changeability	Capability of the software product to enable a specified modification to be implemented	Change recordability X = A / B A = Number of implemented data login items B = Number of data items to be logged

1.2 Effects of Styles, Design Patterns, and Tactics on the Software Quality

As presented in [Scot09, Bass03, Clem03], both design patterns and styles package a set of tactics. These authors define a pattern or a style as a description of a solution to a multi-variable problem, and a tactic as a description of a solution to a single-variable problem. A tactic is primarily used to support a single characteristic of quality in [Scot09, Bass03] but it may also be used to support multiple characteristics as well.

For example, the Layer pattern achieves portability by encapsulating platform-specific details behind stable interfaces. In this case, the tactic "Use Encapsulation" is necessary for both portability and maintainability. Each tactic may relate to a set of measures [Kozi11] for which the software designer measures the impact of the tactic on the software product's quality. Each measure details a software quality objective. Each characteristic of quality may relate to multiple objectives.

The problem of attaining software product quality involves optimizing a set of orthogonal, complementary, and antagonistic targets. The software designers may express the design problem as a multi-objective optimization problem (MOOP) or a multiple objective non-linear programming problem [Kozi11, Ralp85, Stad84, Stad79].

This kind of problems has at least four definitions that software designers need to specify:

1. the set of objective functions that defines the problem-space;
2. the set of threshold values that defines constraints on the solution-space;
3. the set of solutions that defines the solution-space;
4. the definition of the aggregate objective function for optimizing the solutions.

The set of objective functions may be specified using a vector **M** = [f_1 … f_m] of m measures formulas with M Ɛ Q, where Q is a quality model that defines objectives functions to maximize or minimize.

Software designers use threshold values to control that evaluation results are acceptable for all measures. The threshold values may be specified using a vector **T** = [t_1 … t_m] of m threshold values assigned to the measures formulas. The vector **T** specifies the acceptable evaluation results that candidate design solutions need to achieve. The threshold values are the degrees of freedom that constrain the solution-space S Ɛ C, where C is the search space.

In the set of solutions, each design solution provides a set s = (d_1 … d_n) of n design decisions (e.g., using a style, a design pattern, or a tactic) that defines the solution-space s Ɛ S. A design solution s_c = (d_1 … d_n)$_c$ is a candidate solution if each of its related evaluation result r_i for any measure formula f_i (with only one formula for each measure) is at least equals to the related threshold value.

For a candidate solution s_c, F(s_c) represents the aggregate objective function that combines the evaluation results of a vector M(s_c) = [f_i(s_c) | (1<=i<=m)] of m objectives functions. If the optimality is a maximum, max(F(s_c)) represents a design problem that is a multi-objective optimization problem. The problem is the search of a candidate design solution s_c of n design decisions that maximizes the aggregate objective function F in a solution-space S.

A weighted-sum approach may be used to let the software designers influence how a design decision d impacts a vector **M** = [f_1 … f_m] of m objective functions. Each objective function is multiplied by a weighted value £$_i$, where R_m is a set of valid weighted values. The weights are usually based on the issues and the measures that need to be evaluated by the software engineers. The objective functions and the set of weights need to be normalized in order to sum values of consistent magnitudes.

1.3 Approaches for Representing Tactics, Design Patterns and Styles

To evaluate the effects of the design decisions (e.g., using styles or tactics) on the characteristics of quality, software designers need to represent and use these concepts in a manner that supports systematic evaluations. Catalogs of design artifacts such as tactics and styles allow a software designer to reuse the collected wisdom of the software engineers' community to solve recurring problems [Bach07]. Representing appropriate DK in a format useful for supporting the software implementation process involves numerous vocabularies and constraints [Kim10a]. Software designs are based on styles, design patterns, and tactics that provide a domain-specific design vocabulary and constraints on the design solution.

This section presents a list of representation schemes used to describe styles, design patterns, and tactics. This analysis of the literature gives an idea of the general trends in the research community [Kim10a, Scot09, Bass03, Clem03, Ande01, Klei99, Shaw97, Bush96]. Software designers have worked on formalizing architecture documentation practice into the IEEE standard 1471-2000 [Ieee1471], a recommended practice for architectural description of software-intensive systems. This standard establishes a framework of concepts and a vocabulary for discussing architectural issues of software systems. It specifies the required content of architectural descriptions, which may be useful for understanding the required content of styles and design patterns descriptions. Clements *et al.* [Clem03] detail what should be contained in an architectural description.

1.3.1 Representation Schemes for Tactics

The tactics are finer-grained design decisions that constitute architectural styles and design patterns. This section presents three representation schemes for tactics: a catalog of architectural tactics [Scot09, Bass03], a UML-based graphical modeling notation for tactics [Kim10a, Kim09, Gies07, Booc99], and a formal representation of tactics [Wyet09]. An overview of advantages and disadvantages of these representation schemes is provided next.

1.3.1.1 Catalog of Architectural Tactics

A catalog of architectural tactics has been proposed in [Bass03]. In this catalog, each tactic (e.g., Increase available resources) is related to a quality attribute (e.g., Performance) and a specific interest (e.g., Resource management) that describes the issue to be solved. A tactic is documented in a textual format and describes the situation when it can be applied. It may be accompanied by box-and-line drawings that present the organization of the tactic's components and connectors. Subsequent works have refined parts of the catalog and categorization introduced in [Bass03].

In [Scot09], the authors review the availability tactics. They refine some tactics into lower-level tactics (e.g., System Monitor is refined into Heartbeat and Watchdog). In addition to refining the categorization, the authors do the same as in [Bass03] and provide some examples of specific implementation techniques and the expected results for each tactic. To illustrate how a tactic is described in [Bass03] and [Scot09], the following description is taken from [Scot09]:

> "Exception Detection refers to the detection of a system condition that alters the normal flow of execution. For distributed real-time embedded systems, the Exception Detection tactic can be further refined to include System Exceptions, Parameter Fence, and Parameter Typing tactics. System Exceptions will vary according to the processor hardware architecture employed and include faults such as divide by zero, bus and address faults, illegal program instructions, and so forth. The Parameter Fence tactic incorporates an a priori data pattern (such as 0xDEADBEEF) placed after any variable-length parameters of an object."

1.3.1.2 Feature and UML-Based Modeling

Architectural tactics may be described in more formal languages as well. Some researchers use Unified Modeling Language (UML) graphical diagrams to describe the structural and behavioral aspects of tactics [Kim09]. UML is a popular graphical modeling notation for object-oriented software development [Booc99]. It provides capabilities such as multiple design views, semi-formal semantics, and a formal Object Constraint Language (OCL) for expressing constraints on design elements [Gies07]. UML 2.0 allows software designers to represent the architectural aspect of software.

In [Kim09], the authors propose to use a feature modeling approach [Czar00] to link together the desired quality attributes, the interests of stakeholders, and the architectural tactics. The root of the feature model gives the name of the quality attribute (e.g., safety), each node of the first level specifies a stakeholder interest (e.g., resist the attacks), and each lower-level node is connected to one or more lower-level nodes and gives the name of an architectural tactic (e.g., authenticate users).

A link between two tactics is a relationship that the software designer should consider to elaborate a design that supports the related quality attribute and interests. To support the implementation of the high-level design, Kim *et al.* [Kim09] specify each tactic using a role-based meta-modeling language (RBML). They use UML to represent the structural and behavioral constraints of each tactic using UML-based roles in a class and sequence diagram. The semantics of the tactics are specified by a set of roles. A role extends an element of the UML meta-model by adding a set of constraints that the instances of the role must respect. Kim *et al.* [Kim09] also define rules for the systematic liaison and composition of the tactics.

1.3.1.3 Formal Specifications

A formal specification of tactics may help to clarify the required quality and make explicit the decisions software designers may have to make regarding the tactics. A formal specification can be analyzed. Wyet has used the Z formal notation to specify the tactics for the security quality attribute [Wyet09]. These specifications serve to prove that the system specification is consistent and correctly implements the tactics. They provide a framework to analyze specific security mechanisms.

1.3.2 Representation Schemes for Design Patterns and Styles

Some authors of the pattern community claim that styles may be subsumed by the idea of patterns [Gies06]. Styles and design patterns are not distinguished consistently. Usually the terms design patterns refer to lower-level artifacts, and architectural styles refer to higher-level artifacts [Gies06]. A pattern describes a solution to a recurring problem that occurs in a specific context [Alex77]. The pattern states the design issue to be solved, the trade-offs between the issues involved, and the situation when it can be applied.

1.3.2.1 GOF-Based Template for Design Patterns

A pattern can be documented in various forms, which include the Gang-of-Four (GOF) [Gamm95] and Coplien [Copl96] forms. These forms contain sections for intent, motivation, applicability, structure, participants, collaborations, consequences, and related patterns. The textual format of these forms may be accompanied by box-and-line drawings that present the organization of the pattern's components and connectors.

1.3.2.2 Catalogue of Styles

Specific formats exist to represent styles as well. Many formats of styles are textual [Garl94, Shaw96, Bush96, Ande01] and human-oriented. They describe styles in natural language as a collection of components, connectors, and constraints (i.e., topological or semantic) on how they can be combined. In [Shaw97], Shaw and Clements consider a style as a set of design rules that identify the kinds of components and connectors that can be used to compose a system, together with local and global constraints on their topology. The authors use the control and data issues as well as other criteria to classify the styles. They aim to "establish a uniform descriptive standard for styles, provide a systematic organization to support uses of information about styles, and help choosing styles for a given problem". A primary objective of their classification is to capture the common meanings of the informal styles descriptions into a systematic form.

In [Klei99], the authors introduce the notion of an Attribute-Based Architectural Style (ABAS). They explicitly associate a style with a qualitative or quantitative reasoning framework. "Every ABAS comprises a problem description, a stimulus to which the ABAS is to respond and the expected response measures, an architectural style that provides designers with the wisdom of preceding designers faced with similar problems, a description of how the quality attribute models are related to the style, and the conclusions about the predicted architectural behavior. Linking analytic models to architectural styles allows an architect to reuse the cumulated experience of the various attribute communities".

An ABAS is specific to only one quality attribute and it makes reusing styles with predictable properties the foundation for more precise reasoning about architectural design [Klei99]. However, all of the representations including ABASs are more textual than formal.

1.3.2.3 Architecture Definition Languages (ADL)

Architectural styles are usually represented in forms that are more human-oriented. Styles may be described using formal techniques as well. Examples of formal techniques are the style specifications in Architecture Definition Languages (ADLs) [Garl00b]. An ADL may be used to model components, connectors, and topologies at a high level of abstraction and focuses on abstract architecture and explicit treatment of connectors.

As mentioned in [Medv00], an ADL provides an abstract description of the style and a foundation for architecture construction. It exposes the high level style constraints and the rational for specific choices. An ADL supports domain specific styles descriptions and system constraints checking of conformance to style constraints, quality attributes, and component and connectors dependencies. However, ADLs subsume different formal semantic theories and thus focus on different application domains, architectural styles, or aspects of the architectures they model [Medv00].

[Medv00] present a classification methodology and a framework for comparing ADLs. They identify and compare many ADLs and conclude that every ADL must provide the means for the explicit specification of the architectural components, connectors, and topologies. This study is helpful to understand what a style description may comprise. It identifies the strengths and weaknesses of each ADL, which allows designers to choose an appropriate ADL for particular needs. Complementary languages, such as the formal specification language Z [Spiv92], have been used to model styles rules and constraints [Loul06, Medv99b, Abow95, Abow93] and realize dynamic style refinement and composition via strict consistency checking [Nadh08, Loul06, Loul04].

1.3.2.4 Formal Representations of Styles

Formal methods, such as the Z notation, use mathematical models for model checking and theorem proving [Clar96, Wing98]. In particular, the Z notation provides a framework within which styles may be specified (connectors are usually not explicit [Meht00, Alle97]), designed, analyzed, and verified in a systematic rather than ad hoc manner. Extensions such as the Object-Z [Smit00a, Duke95], an object-oriented version of the Z notation, have been proposed to simplify the use of formal methods and augment capabilities.

1.3.2.5 UML-Based Representations of Styles

Many researchers facilitate the use of formal methods through Unified Modeling Language (UML [Booc96]) graphical diagrams [Mila08]. They use automated graph transformation [Loul04, Guo05] and generation of the Z schemas skeletons [Dupu00]. UML 2.0 allows designers to represent the architectural aspect of software in accordance with architectural styles more effectively [Kace05].

Still, many modeling constructs from ADLs cannot be mapped directly to the UML [Gies07]. UML extensions have been proposed to represent architectural concepts (e.g., topological constraint and connectors) [Medv99b, Robb98]. Nonetheless, UML cannot be extended to model and efficiently express every feature of every ADL. Bringing together UML and formal methods may help to make style representations more rigorous.

The OCL [Warm99] is the specification language that aims to formalize the UML. UML is based on a meta-model that states rigid rules and constraints on the elements of a UML diagram. However, the OCL is more appropriate for that purpose [Omg06]. It is a formal technique that can be used to specify invariants on classes, types and interfaces as well as pre-conditions and post-conditions, state changes, guards, and constraints on operations and methods.

The creators of OCL claim that OCL is easier to read and write than other formal languages and that there is no need for a strong mathematical background to use OCL [Omg06]. As mentioned in [Rich02], a formal foundation should make the meaning of constraints more precise and should help to eliminate ambiguities and inconsistencies. Another important aspect of a formal specification language is its ability to support refinement.

1.3.2.6 Ontology-Based Representations of Styles

Some researchers apply the Web Ontology Language (OWL) and the Semantic Web Rule Language (SWRL) to improve the semantics of architectural styles, components, and connectors [Zhan09]. The OWL is a XML-based language for describing ontologies [Smit00b]. An ontology defines a conceptualization of a particular domain. It is created using concepts from the domain, properties of those concepts, and relationships between concepts.

The OWL provides the building blocks for specifying the semantics of styles in a well-defined manner [Bern01]. The Semantic Web Rule Language (SWRL) is an OWL-based rule language [Horr04]. SWRL allows users to write rules in terms of OWL concepts such as classes, properties, individuals, and data values. It provides deductive reasoning capabilities when performing inference. The mapping between the OCL and the SWRL has been addressed in the literature [Rewe06]. The main benefit of such an approach is that UML/OCL rules can be mapped into all other rule languages (e.g., Jess and Prolog).

1.3.3 Synthesis of the Representations of Tactics, Design Patterns, and Styles

1.3.3.1 The Representation of Tactics

The representations of tactics in [Scot09, Bass03] use a textual format that makes it difficult to systematically select appropriate tactics, to compare them, and to evaluate the consequences of applying a tactic or a combination of tactics for a particular problem.

The term tactic usually refers to fine-grained artifacts, but the textual format provides no exclusive criteria and makes it difficult to determine whether something is a tactic or not. In [Kim09], the authors use UML 2.0 to represent the architectural tactics. However, UML has neither the semantics nor the capability to characterize all the tactics' properties [Kim09].

Extension to UML results in dependence to a tool, which reduces the portability of the representation. In addition, this approach provides no support for evaluating the impact of a tactic on the software quality characteristics. In [Wyet09], the authors use the Z formal notation to specify the tactics and prove that the system specification is consistent and correctly implements the tactics. However, such a formal method to represent tactics has not been largely used yet. The Z notation uses complex formal semantics and elements of logic and mathematics that require advanced skills to represent the tactics.

1.3.3.2 The Representation of Design Patterns

The textual format of GOF-based templates [Gamm95] makes it difficult to select appropriate patterns, to compare them, and to evaluate the consequences of applying a pattern or a combination of patterns for a particular problem [Gies06, Alle97]. The term design pattern usually refers to design artifacts, but the textual format provides no exclusive criteria and makes it difficult to determine whether something is a design pattern or not. In addition, this approach provides no support for evaluating the impact of a design pattern on the software characteristics of quality. Still, such a textual format to represent design patterns is largely used.

1.3.3.3 The Representation of Styles

The list of style representation schemes mentioned so far [Clem02, Ande01, Medv00, Klei99, Busc96, Garl95] illustrates some trends in the research community. Most software architectures are based on one or more architectural styles that provide a domain-specific design vocabulary and a set of constraints on how styles are used.

None of the works and representation schemes cited so far takes architectural tactics into account: they are attentive to styles only. These approaches provide little or no support for selecting styles based on the required level of quality and to evaluate the impact of a style on the level of quality. Their primary concern is the achievement of the functional requirements.

Applying the GOF-based templates to the style concept requires a broad interpretation of the pattern concept. The templates used to represent lower-level concepts such as patterns are not intended to represent higher-level concepts such as styles [Monr97]. The level of abstraction of styles is not consistent from one author to the next [Clem02, Busc96, Garl95] making it difficult to decide whether a description is a style or not.

1.4 Approaches for Supporting Architectural Design

1.4.1 Attribute-Driven Design Method

One of the approaches that support the architectural design process is the Attribute-Driven Design method (ADD) [Bass03]. ADD proposes an iterative process to designing software architecture. At each iteration of the ADD, the architect chooses the architectural tactics and styles that satisfy the most important quality attributes for that iteration, called architectural drivers. The ADD describes a cycle for planning the design fragments, implementing the software structures, and verifying the resulting design. It repeats this cycle until all architectural drivers are met. To choose and apply the tactics and styles that best achieve the drivers, software designers can use methods such as the ones proposed in [Bass05, Bach03].

ADD as described in [Wojc06, Bass03] starts after the requirements analysis in the software implementation process. It provides the first level of decomposition of the modules (systems, subsystems, layers, packages, classes, etc.). The system is the first module decomposed. Then, each resulting module is considered for decomposition. The choice of the module to decompose is based on the architectural drivers. ADD iteratively decomposes a system or system element by applying architectural tactics and styles that meet the quality requirements of the system.

The ADD method iterates on three activities and seven steps. Planning the design aims to select the types of elements that achieve the requirements (steps 1 to 4). Implementing the design aims to instantiate the elements to satisfy the requirements (steps 5 and 6). Verifying the design aims to determine if the resulting software design meets the requirements (step 7).

1.4.2 Quality Ontology and Architectural Knowledge Base

Another approach used to support the architectural design process is given in [Ovas10]. They propose an approach to fully integrate quality requirements in the software design process. Their approach allows the software designer to manage and track the quality attributes from the requirements specification to the architecture design. The approach proposes a process with three phases: 1) the quality requirements modeling phase, 2) the architectural modeling phase where quality requirements models are transformed into architectural models, and 3) the evaluation phase. The first two phases are divided into two processes.

Each of the two first phases includes a knowledge engineering process and a software engineering process. The knowledge engineering process aims at creating quality ontology and an architectural knowledge base, while the software engineering process uses this ontology and knowledge to model the quality requirements and the software architecture of a particular system. The ontology represents the architect’s understanding of a quality attribute, while the quality requirements model represents the client's needs.

The architectural knowledge base is a directory of reusable artifacts, including generic and domain specific tactics and styles, as well as profiles of quality attributes. The tactics and styles are organized to allow a search using the name of the quality attributes they support. The quality requirements model can therefore be used to define the styles that serve as a starting point of the architecture and the tactics that are used to refine the styles. The entities of the resulting architectural models are annotated with the quality attributes they support.

The objective of the third phase is to evaluate the architecture to determine the level to which the requirements were attained and suggest improvements. The architect makes the assessment in three stages. First, it prioritizes the quality attributes. For the architectural drivers identified, a tradeoff analysis is performed. Then the software designer evaluates the quality attributes of high and medium importance. Finally, the software designer compares the results of the evaluations with the acceptance criteria derived from the requirements and then identify possible improvements.

1.4.3 Limits of the Approaches

Many approaches have been proposed to support the architectural design and decision processes [Zimm12, Ovas10, Kim09, Bach07, Tyre05, Bass05, Bass03, Bach03, Clem02], but few approaches (e.g. [Zimm12, Ovas10]) support the software designers in managing and keeping track of the accumulated knowledge during the architectural design process. The focus of ADD is the process of architecting systems in order to satisfy a set of quality attributes and to manage tradeoffs between these attributes. ADD provides no support to manage the artifacts that it produces and uses. The approach proposed by Ovaska *et al.* [Ovas10] focuses on finding tactics and styles using quality attributes. While this is very useful, an architect still needs to keep track of the rationale, objectives and constraints that led the choice of the quality attributes.

Although many architectural tactics and styles have been described and cataloged [Scot09, Bass03, Clem02, Shaw96, Busc96, Garl94] in the literature, few approaches support the software designers in selecting and using the appropriate styles. In all cases, both tactics and styles are documented in textual formats. The textual formats of tactics and styles make it difficult to select appropriate tactics and styles, to compare them, and to evaluate the consequences of applying each of them for a particular problem [Scot09, Kim09]. In particular, the binding and composition rules given in [Kim09] are in a textual format.

Various models and tools for the management of architectural knowledge have been compared in [Pari08] according to some properties, including stakeholder-specific content, easy manipulation of content, descriptive in nature, and support for codification, personalization, and collaboration [Shah09]. Shahin *et al.* conclude that many models capture and document the rationale, constraints, and alternatives of architectural decisions [Shah09].

Existing models express similar concepts in different terms. Also, there is a lack of tool-support, particularly for personalization. In addition, the management of the relationships between the architectural decisions and the elements they influence (e.g., files and views) is still a challenge [Tyre05, Khal10]. Many models use a textual format for describing the architectural decisions and do not keep track of the resulting artifacts.

1.5 Summary of the Activities and Artifacts of the Design Process

This section summarizes the common activities and SDAs related to the design process. From the literature, Table 1.4 identifies six activities (i.e., select, identify, define, specify, describe, and evaluate) for the design process, altogether with the SDAs generated or used by each activity. The proposed list of activities and SDAs is based on the selection of activities and SDAs from the related works [Iso42010, Apri11, Bass03, Clem03] and the vocabulary and activities from ISO 12207 on Software life cycle [Iso12207, Iso29110].

Related works decompose the design process into finer-grained or coarser-grained activities than the decomposition proposed in Table 1.4. From this perspective, the design process is decomposed in a manner that emphasizes the refinements of the artifacts while designing. The activities in the upper rows provide artifacts that are used by the activities in the lower rows for developing, evaluating, and describing designs. The inputs and outputs of these activities are SDAs that should be managed – see Appendix I for more descriptions of these SDAs.

Table 1.4 The proposed activities of the design process

Activities	Attribute-Driven Design (ADD)	ISO12207	ISO29110	Relevant SDAs
Select the objectives	Confirm there is sufficient requirements information	Establish and document software requirements	Understand requirements specification	Need, goal, expectations, risks, politics, business model, situational factors, requirements, constraints, business rules, domain objects, processes, activities, tasks, procedures
	Refine requirements and make them constraints	Refine requirements and make them constraints		
	Choose an element of the system to decompose	Schedule for software integration		
	Identify candidate architectural drivers			
Identify knowledge artifacts	Choose a design concept that satisfies the architectural drivers			Architectural concerns, application domain, standards, regulations, conventions, properties, patterns, styles, tactics
Define architectural artifacts	Instantiate architectural elements and allocate responsibilities	Allocate the requirements to its software components	Identify software components and associated interfaces	Architectural design rationale, architectural risks, assumptions, scenarios, design fragments
Specify system artifacts	Define interfaces for instantiated elements	Refine the software components to facilitate detailed design	Provide the detail of software components and their interfaces to allow the construction in an evident way	Detailed design rationale, system's risks, assumptions, operation contracts, modules, components and connectors
		Develop a top-level design for the database		
		Develop a top-level design for the interfaces		
Describe architectural artifacts		Document a top-level design for the database	Document the software component identification	Glossary, views, viewpoints
		Document a top-level design for the interfaces		
		Develop and document preliminary versions of user documentation		
Evaluate software structures	Verify requirements	Define and document preliminary test requirements		Acceptance and assurance criteria, internal measures, external and in-use measures, evaluation records
		Evaluate the architecture, interface, and database designs		
		Conduct review(s)		
		Document the results of the evaluations		

1.6 Approaches for Supporting Design Traceability

Existing approaches attempt to support design traceability with specific processes, models, and tools. They assist software designers in their decision-making activities by characterizing and managing the design decisions, the design rationale, and the relationships between them [Zimm12, Ovas10, Zimm09, Wanf09, Jans07, Kruc06, Tang06, Jans05, Tang05, Tyre05, Jans04, Bass03, Clem03]. Tang et al. [2009] classified architectural knowledge into four general categories: context knowledge (problem space), general knowledge (styles, tactics, and patterns), reasoning knowledge (design decision and design rationale), and design knowledge (design fragments and software structures). This section presents the reasoning knowledge.

1.6.1 Design Decisions

Software designers make design decisions (DDs), such as choosing patterns, styles, and tactics. Zimmermann and al. [Zimm09, Kruc06] proposed eight decision types, including decisions for pattern selection, pattern adoption, technology selection, technology profiling, vendor asset selection, and vendor asset configuration.

As stated in [Zimm12, Tyre05], "developers want guidance on how to proceed with a design. Customers want a clear understanding of the environmental changes that must occur and assurance that the design meets their business needs. Other designers want a clear, salient understanding of the design's key aspects, including the rationale and options the original designer considered". The purpose of the design decisions proposed in [Zimm10] is to:

- "Provide a single place to find design decisions
- Make explicit the rationale and justification of design decisions
- Preserve design integrity
- Ensure that the design is extensible and can support an evolving system
- Provide a reference of documented decisions
- Avoid unnecessary reconsideration of the same issues"

Table 1.5 presents the template proposed in [Tyre05] for capturing the information of a DDs. Many recent design approaches provide support for documenting and using DDs as core artifacts of software design [Zimm12, Ovas10, Zimm09, Wanf09, Jans07, Kruc06, Jans05, Tyre05, Jans04], including design rationale [Tang07, Tang06], architectural decision models [Zimm09], decision relationships [Zimm09, Kruc06]. These approaches provide tool support for architectural knowledge management and decision-making by maintaining a knowledge repository. They treat software design as a design decision process and manage architectural and design knowledge for documenting design decisions explicitly. These decision-centric approaches capture design rationale and use requirements as a basis to support reasoning.

Table 1.5 Architecture decision description template (adapted from Tyre05)

Issue	Describe the architectural design issue
Decision	State the decision rationale
Status	State the decision's status
Group	Group to help organize the set of decisions
Assumptions	Describe the underlying assumptions (limits) in the environment
Constraints	Capture constraints to the environment that the decision poses
Positions	List the viable design alternatives
Argument	Outline why the designers selected a position
Implications	State the decision's implications
Related decisions	List the related decisions
Related requirements	Map the decision to the objectives or requirements
Related artifacts	List the related architecture or design artifacts
Related principles	List the agreed-upon set of principles
Notes	Capture notes and issues discussed

In spite of that, most software designers omit to document the DDs and design rationale, which may lead to costly support efforts for system evolution, lack of communication between the stakeholders, and limited reusability of software artifacts [Capi10, Wanf09, Tyre05, Bosc04].

Table 1.6 presents a summary of the related works and state of the research on DDs. In most development processes, DDs are not documented explicitly but are implicit in the designs [Wanf09, Tang06]. Software designers may not have the time or the ability to document their designs [Tyre05]. In addition, existing tools provide limited support for managing DDs and the rationale that lead to them. Most approaches do not relate DDs to individual design artifacts. They support defining and sharing design decisions. In addition, existing approaches provide limited support for managing the knowledge of the problem space that influenced the design, the styles, tactics, and patterns used in the design, and the related design artifacts, design rationale, and design decisions [Tang09].

Table 1.6 Current state of the research on design decisions

Related works	**"+" means the related works DO realize the claim** **"-" means the related works DO NOT realize the claim**
Zimmermann et al. – IBM research laboratory (2007 to 2012)	+ describe and formalize an architectural decision model + describe a metamodel and modeling principles for design decisions + describe architectural patterns as conceptual architecture alternatives + capture decisions required, decisions made, and possible solutions + describe dependency relations, integrity constraints, and production rule + describe steps: identification, making, and enforcement of decision
Tang et al. (2009)	+ provide a comparative study of architecture knowledge management tools + define 10 criteria of an evaluation framework for tools + define usage scenarios for architectural knowledge management tools
Capilla and Babar (2008)	+ describe the concept of variability model + describe binding time, variation points, variants, and their relationships + associate design decisions to variation points and variants + review existing tools for capturing and managing design decisions - check the inconsistencies in the variability model
Boer et al. (2007)	+ compare tools capabilities for decisions modeling
Kruchten et al. (2006)	+ define a semantic ontology for decisions + describe attributes and types of decisions + describe when and how decisions are made + define types of decision dependencies + focus on the visualization of the decisions + identify many use cases for decision knowledge - describe formally what is an architectural decision - treat design problem and solution as distinct entity - separate decisions required and decisions made - propose concepts for structuring decision models

Abrams et al. (2006)	+ provide modeling tool support for design artifacts + introduce a topic hierarchy + define an outcome attribute in the decision entity + define alternatives as a separate entity
Akerman and Tyree (2006)	+ define an ontology for decisions to support the design of software architectures
Tyree et al. (2005)	+ define a template for documenting architectural design decisions
Jansen and Bosch (2005)	+ view software architecture as a composition of a set of design decisions + treat decisions as a first class architecture design concept + focus on change over time as a dominating force for decision making + distinguish design problems and solutions to them + outline the attributes that are required to capture related knowledge + integrate decision models with models for other viewpoints + compare tools capabilities for decisions modeling - introduce their metamodel in text and figures - explicit dependencies between different problems or different solutions - propose concepts for structuring decisions and fragments - propose solutions for the reuse of architectural decision knowledge
Bass et al. (2003)	+ mention the term architectural decision + describe tactics as architectural decisions + propose quality attributes and design concerns for classifying tactics - define what is an architectural decision
IBM Unified Method Framework	+ define a template for capturing architectural decisions + in use on professional services engagements for IBM clients since 1998 + provide reference architectures with decisions made during design - formally specify the metamodel
Tool Support SEURAT, PAKME, ADDSS, AREL, Archium, Knowledge Architect, SPLE	+ define metamodels for managing decisions + provide tools with decision modeling capabilities + capture design decisions, design rationale, and design models + support basic decisions dependencies + support traceability between requirements, design decisions, and design + support software architecture design, documentation, and evaluation + provide a knowledge repository of generic and specific knowledge + document the chain of dependencies between decisions - provide support for managing the decisions and the design rationale - support variability management - relate design decisions to individual architectural parts - ensure the integrity of the decision model - maintain explicit relationships between design artifacts

1.6.2 Design Rationale

"Design rationale capture the reasons behind the design decisions" [Tang06a]. Many works confirm the need to manage the design rationale in an effective design reasoning model for system maintenance [Tang06a, Tang05, Tyre05, Bosc04, Clem03, Bass03, Ulri02, Perr92]. Tang et al. present a survey of nine types of generic design rationales from the literature: design constraints, design assumptions, weakness, benefit, cost, complexity (risk), certainty of design (non-risk), certainty of implementation (non-risk), and tradeoffs. They classify additional types of factors that influence design into three categories: business goals oriented, requirement oriented (functional and non-functional), constraints and concerns [Tang06a].

Software designers capture design rationale either to deliberate about a design or to track the results of the reasoning [Tang06a, Tang06b]. The approaches to representing design rationale include argumentation-based [Lee91, Kunz70, Toul58] and template-based [Iso42010, Ieee1016, Bass03, Clem03] representations. Argumentation-based approaches use networks of arguments and issues, and a resolution process for deliberation about a design [Lee97, Lee91, Kunz70]. Deliberation refers to the act of considering different points of view for coming to a reasoned design. Template-based approaches use formatted documentation for capturing the result of the reasoning [Iso42010, Iso1016, Tyre05, Clem03]. For argumentation-based approaches, Tang et al. [Tang06a] identify three challenges that concern Template-based approaches as well:

1. the identification of the knowledge for reasoning;
2. the creation of the design reasoning model to retain the knowledge; and
3. the utilization of the design rationale to help understand a design.

Tang et al. [Tang06a] also identify issues for these approaches:

1. the cognitive burden to capture the design rationale;
2. the lack of traceability of both
    a. the design artifacts being discussed; and
    b. the relationships between the design rationale.

1.7 Summary of Design Knowledge (DK) Management

DK management requires insight into the organization and its processes in order to tailor activities, techniques, and tools to the context, and it requires insight into the SDAs produced or used by each activity of the design process. Organizations that develop or maintain software should manage the DK. In this thesis, a software design artifact (SDA) is any conceptual artifact that is part of the DK related to the problem and solution spaces from which software designers develop and maintain software designs.

The related works' questions, objectives, open issues, and future studies provide insights, templates, and techniques for DK management. This summary captures the common vocabulary, issues, challenges, and activities related to DK management. The purpose of this summary on DK management is to develop a better understanding of the related challenges.

1.7.1 Reasons, Challenges, and Issues for Managing DK

A standard definition of DK, and a standard definition of DK management, that would make consensus is still not found in the literature. Ad-hoc DK management hinders standardization and causes confusion and ambiguity [Pari08]. It is recommended in [Pari08] for software designers "to be specific in defining the semantic of their DK to get over this lack, which helps community to work on a common realization of the term".

Software designers manage the DK for many reasons:

- they need to understand and tailor alternate design solutions that have proven to be useful for designing previous projects with comparable contexts and objectives;
- they aim at improving the design capability of the organizations, and accruing returns on investments in designing the software and building the systems;
- they reuse the DK for improving the design process of actual and future projects.

The software designers need to manage the DK for evaluating how each SDA impacts the software design and the capability of the system to satisfy stakeholders' needs. Thinking about each SDA from multiple perspectives may be difficult. Insufficient details about the SDAs and their relationships and interactions may lead software designers to inappropriate or suboptimal decisions. In addition, the business context (e.g., software product lines and technologies) and changing objectives may force the designers to re-evaluate the initial design decisions. Transforming legacy designs according to new contexts [Ulri02] requires DK management. To paraphrase [Luze13], the purpose of the DK management process is to provide relevant, timely, and complete DK to designated parties during and, as appropriate, after a software product life cycle for supporting the decision-making activities and improving the resulting designs.

For achieving successful DK management, software designers must realize pre-requisites:

1. understand the DK management process;
2. understand the design process and related decision-making activities;
3. manage the SDAs that constitute the DK relating to these two processes.

Many reasons make the DK difficult to manage [Pari08, Tyre05]:

1. software designers often do not document the DK they use [Tyre05];
2. approaches for architecting software focus on the components and connectors, and structures of allocation [Bass03];
3. DK is often not shared with the appropriate stakeholders;
4. DK is not used by the users when they have the possibility to use it; or
5. the design process does not support DK management.

The lack of traceability results in maintenance cost, design erosion, and lack of DK. From this survey of the literature [Iso42010, Zach11, Ovas10, Kim09, Scot09, Pari08, Tang06, Tyre05, Deme03, Bass03, Clem03, Argo00, Medv00, Dave98, Bush96, Gamm95, Garl95, Szul95], Table 1.7 presents the issues described in related works for DK management.

Table 1.7 Issues for design knowledge management

1. Lack of traceability of software design artifacts	21. Lack of scientific rigor
2. Limited analysis capability	22. Lack guarantees of validity
3. Locating the expertise	23. Spatial, temporal, technical, and social concerns
4. Lack of recipient motivation	24. Challenges between diverse design communities
5. Lack of recipient absorptive capacity	25. Different cultures, vocabularies, and referential
6. Lack of recipient retentive capacity	26. Lack of up-to-date knowledge
7. Lack of source motivation	27. Lack of explicit collaboration between teams
8. Lack of perceived reliability of source	28. Complex relationships of knowledge item
9. Causal ambiguity (why sharing)	29. Rotation of personnel
10. Lack of trust relationships	30. Design knowledge management overhead
11. Misunderstanding of the design knowledge	31. Lack of measurable indicators
12. Need for tailored design knowledge	32. Inadequate management support
13. Tacit, implicit, explicit design knowledge	33. Inadequate skill of participants
14. Need for tailored forms of design knowledge	34. Improper organizational structure
15. Lack of consistency of the design knowledge	35. Lack of widespread contribution
16. Intolerance of mistakes	36. Lack of relevance, quality, and usability
17. Intolerance of redundancy	37. Need tailored approaches, models, and tools
18. Lack of upfront discussion	38. Improper budgeting
19. Media change the context for communicating	39. Lack of responsibility and ownership
20. Lack of meeting places	40. Flexible learning objectives

1.7.2 DK Management in Practice

Many works in the literature depict approaches, models, and tools devoted to DK management [Pari08]. Organizations maintain the SDAs and tailored information item in DK databases or using other supports in order to make the DK explicit. They may share the SDAs by using many documents (e.g., System Design Document, Interface Design Document, Database Design Description, Software Design Description, Interface Design Document, Software Requirements Specification).

Seven conclusions are retained from the literature on approaches, models, and tools for managing DK [Khal10, Shah09, Tyre05]:

1. models document rationale, constraints, and alternatives of design decisions;
2. models express similar concepts in different terms;
3. software designers lack support for personalization;
4. software designers lack support for managing relationships between design decisions and design artifacts (e.g., files and views);
5. models use a textual format for describing the design decisions;
6. models do not keep track of many relationships between design artifacts; and
7. current approaches focus on a subset of the activities of the DK management process.

This literature review has not identified any approach taking into account all the activities and artifacts identified in this thesis related to DK management and design process. Most of the approaches, models, and tools provide limited views into the DK base [Zimm09, Ovas10, Kim09, Pari08, Tyre05, Clem03, Bass03]. The approaches support the design process but few approaches support the software designers using multiple perspectives (e.g., quality, people, functions, activities) for managing the DK during the design process.

1.7.3 The Proposed Activities of the DK Management Process

Table 1.8 identifies the activities of the DK management process: acquiring, defining, reusing, sharing, communicating, evaluating, and managing. This table links the activities to the techniques, issues, and works from the literature. The proposed list of activities is based on the selection of activities from the related works [Luze13, Zimm12, Rus02, Ulri02] and the vocabulary and activities from ISO 12207 on Software life cycle [Iso12207].

Table 1.8 Activities, techniques, and issues of the DK management process

Acquiring - Techniques for acquiring the design knowledge from people or artifacts 1. Analyzing code and test cases 2. Analyzing documentation 3. Analyzing software design artifacts 4. Analyzing version history 5. Interviewing / surveying people 6. Running software Issues from Table 1.6: 1 to 13, 16, 17, 32 to 35, 40 References: Deme03, Bass03, Ulri02, Dave98, Szul95
Defining a design knowledge base - Techniques for representing the design knowledge in forms that facilitate its management 1. Cataloguing of software design artifacts 2. Cataloguing best practices for the design process 3. Defining a design knowledge database referential (e.g., software architecture description document) 4. Defining a standard representation of the design knowledge (e.g., ontologies, notations, and templates) 5. Defining a standard vocabulary Issues from Table 1.6: 2, 12 to 14, 32 to 35 References: Ovas10, Kim09, Scot09, Tyre05, Bass03, Clem03, Medv00, Bush96, Gamm95, Garl95, Iso42010
Reusing - Techniques for reusing the design knowledge during the design process 1. Analyzing the design knowledge databases 2. Generating software design artifacts 3. Analyzing the software design artifacts 4. Using a design knowledge database referential 5. Using a standard vocabulary Issues from Table 1.6: 1, 2, 11 to 15, 32, 33, 36 References: Ovas10, Kim09, Bass03
Sharing - Techniques for sharing the design knowledge person-to-artifact in forms that improve its management 1. Standardizing ontologies, notations, and templates 2. Using a design knowledge database 3. Documenting the software architecture 4. Documenting software design decisions 5. Documenting lessons learned 6. Modeling views on the software architecture 7. Using standard graphical notations

Issues from Table 1.6: 7 to 17, 32, 33, 40 References: Zimm12, Ovas10, Zimm09, Kim09, Pari08, Bass03, Clem03, Tyre05, Argo00, Szul95
Communicating - Techniques for conveying the design knowledge person-to-person 1. Meeting people 2. Teaching people Issues from Table 1.6: 4 to 13, 18 to 20, 32 to 34, 40 References: Pari08, Ulri02
Evaluating - Techniques for assessing the validity of the design knowledge 1. Using assessment checklists 2. Using measurable indicators 3. Executing review Issues from Table 1.6: 21, 22, 31, 32, 33, 36 References: Apri11, Bass03
Managing - Techniques for managing the process of acquiring, evaluating, defining, reusing, sharing, visualizing, and communicating the knowledge 1. Planning design knowledge management process 2. Tailoring approaches, models, and tools 3. Tailoring design knowledge management process 4. Controlling design knowledge management process 5. Using reward schemes 6. Providing cultural support 7. Creating communities of practice Issues from Table 1.6: 13, 23 to 30, 32 to 39 References: Zach11, Argo00, Szul95

1.8 Summary of the Requirements for Design Methods and DK Management

Based on the literature review, this section summarizes the requirements for approaches supporting design methods and DK management [Zimm12, Zimm09, Hofm07, Kruch06]. Zimmermann et al. classify these requirements according to three categories [Zimm09]: software engineering method (entire software lifecycle), software architecture design method (design process), and DK management (DK management). These requirements are used in [Zimm09] to analyze existing design methods. They will be similarly used to assess whether the approach proposed in this thesis meets the following requirements (see related works).

Requirements for software engineering method (see [Zimm09]):

R1: Method anatomy = process + notation + supporting techniques and content;

R2: Provide standard description format and metamodel;

R3: Be broadly applicable and actionable, e.g., provide templates and examples;

R4: Provide link between requirements engineering (analysis) and design work;

R5: Ease method content authoring (extensibility) and tailoring (usability).

Requirements for software architecture design method (see [Zimm09]):

R6: Provide multiple architectural viewpoints;

R7: Be driven by quality attributes and stakeholder goals;

R8: Support decomposition of complex design issues (architectural analysis);

R9: Support composition of resolved design issues (architectural synthesis);

R10: Provide a managed to do list;

R11: Support architecture evaluation.

Requirements for DK management (see [Zimm09]):

R12: Obtain required knowledge

R13: Tailor identified knowledge

R14: Document decisions

R15: Align with other models

1.9 The Proposed Structure of Software Design Artifacts (SDAs)

Based on the literature review, this section proposes a structure of software design artifacts (SDAs) for DK management. This structure of SDAs is the basic concept supporting the approach proposed in this thesis, which defines SDAs as any conceptual artifact that

1) provides design knowledge (DK) about the problem or solution spaces of a software design, and
2) corresponds to the identification heuristics presented in Section 4.4.

A SDA is either elementary or composite. The proposed heuristic is that an elementary SDA does not require the utilization of another SDA in the design solution, while a composite SDA does require the utilization of another SDA from the solution space when being used. For example, a tactic is an elementary SDA as proposed in [Bass03], while a design pattern and a style are composite SDAs [Clem03, Gamm95]. Tactics from [Bass03] described in Appendix IV require no SDA from the solution space. The Template Method design pattern requires the utilization of the polymorphism tactic [Gran02, Gamm95]. A SDA may have one or more applications (e.g. resulting in multiple descriptions of the tactics [Scot09, Kim09, Bass03, Lars02], design patterns [Gran02, Gamm95], and styles [Clem03]).

From our point of view, the SDAs constitute the explicit DK that relates to both the design process and the DK management process. From the literature, many SDAs and relationships between them are identified and represented in the SDA structure proposed in Figure 1.1 where a SDA may be, but is not limited to [Iso25010, Iso42010, Zimm12, Harr11, Medv10, Jans06, Tyre05, Bass03, Clem03, Bach03, Gamm95]:

Elementary SDAs

- a tactic [Bass03],
- a quality attribute scenario [Bass03],
- a measure [Iso25010]

Composite SDAs

- a design pattern [Gran02, Gamm95],
- a style [Clem03],
- a design decision [Zimm12, Zimm09, Tyre05],
- a view [Clem03, Iso42010],
- an architectural description [ISO42010],
- or any input or outcome of the design process.

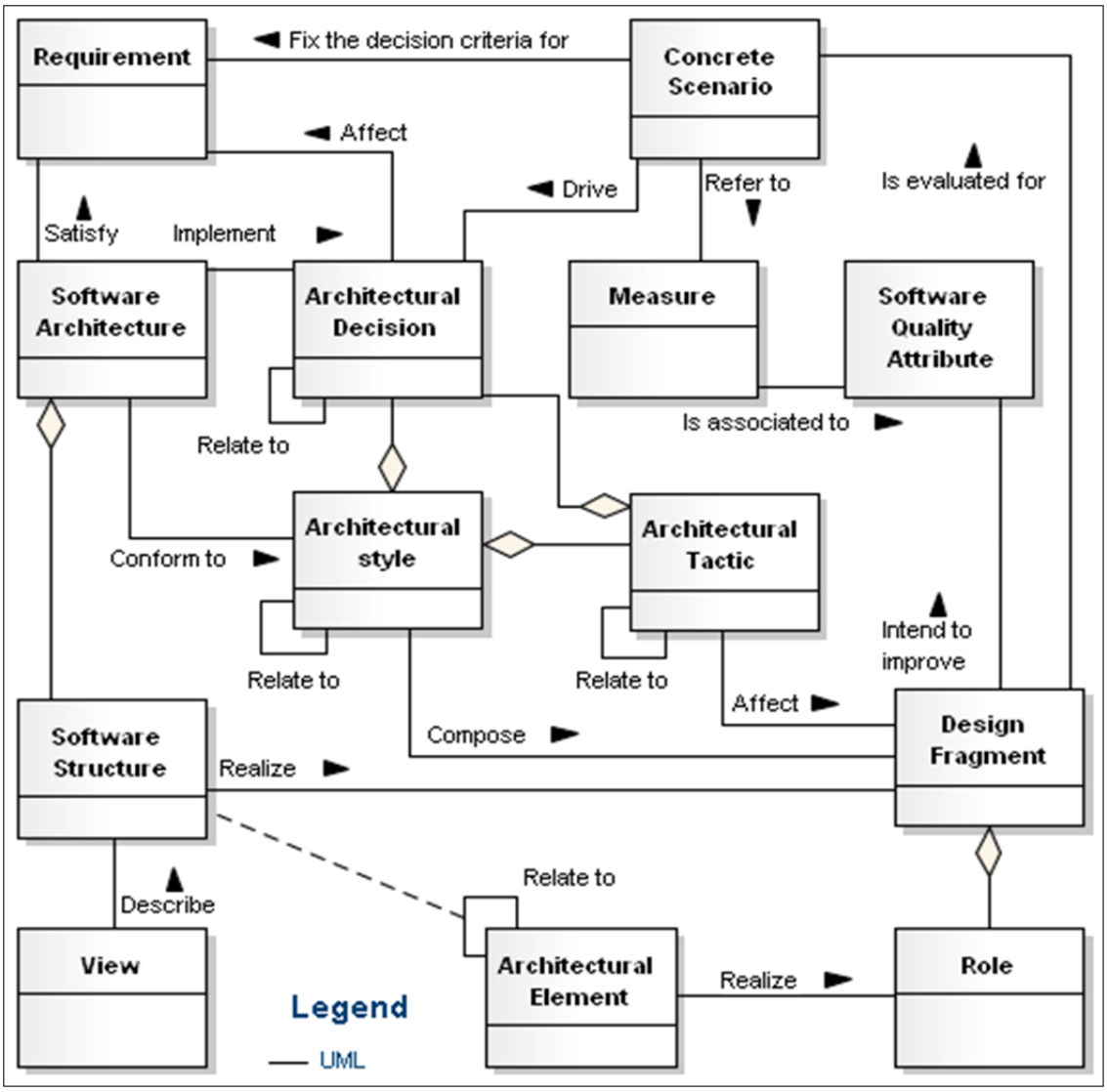


Figure 1.1 Proposed structure of software design artifacts in the SAM framework

CHAPTER 2

THE PROPOSED SOFTWARE ARCHITECTURE MAPPING (SAM) FRAMEWORK

This chapter presents the Software Architecture Mapping (SAM) framework developed in Phase 2 of our research methodology. This chapter is organized as follows and presents:

- an overview of the proposed SAM framework (Section 2.1);
- the proposed activities of the SAM framework (Section 2.2);
- the proposed reference model of the SAM framework (Section 2.3);
- the arguments for justifying the proposed reference model (Section 2.4);
- the limits of the proposed reference model (Section 2.5);
- the positioning of the SAM framework in the literature review (Section 2.6); and
- the limits of the SAM framework (Section 2.7).

2.1 The proposed Software Architecture Mapping (SAM) framework

Figure 2.1 presents an overview of the proposed SAM framework. The colored shapes are the concepts that support the Attribute-Driven Design (ADD) method [SEI11, Nort07, Bass03]. The SAM framework is based on these concepts from the literature (i.e., quality attributes [Iso25010, Bass03], architectural decisions [Zimm12, Zimm09, Jans06, Kruc06, Tyre05], software architecture [Iso42010, Bass03, Medv00], styles [Clem03], tactics [Kim09, Scot09, Bass03], design patterns [Gran02, Gamm95], and analysis methods [Bass03]). To manage the knowledge that relates to existing models and description templates, the SAM framework defines four basic concepts that constitute its reference model (i.e., the SDA, software structures map (SSM), argument, and view).

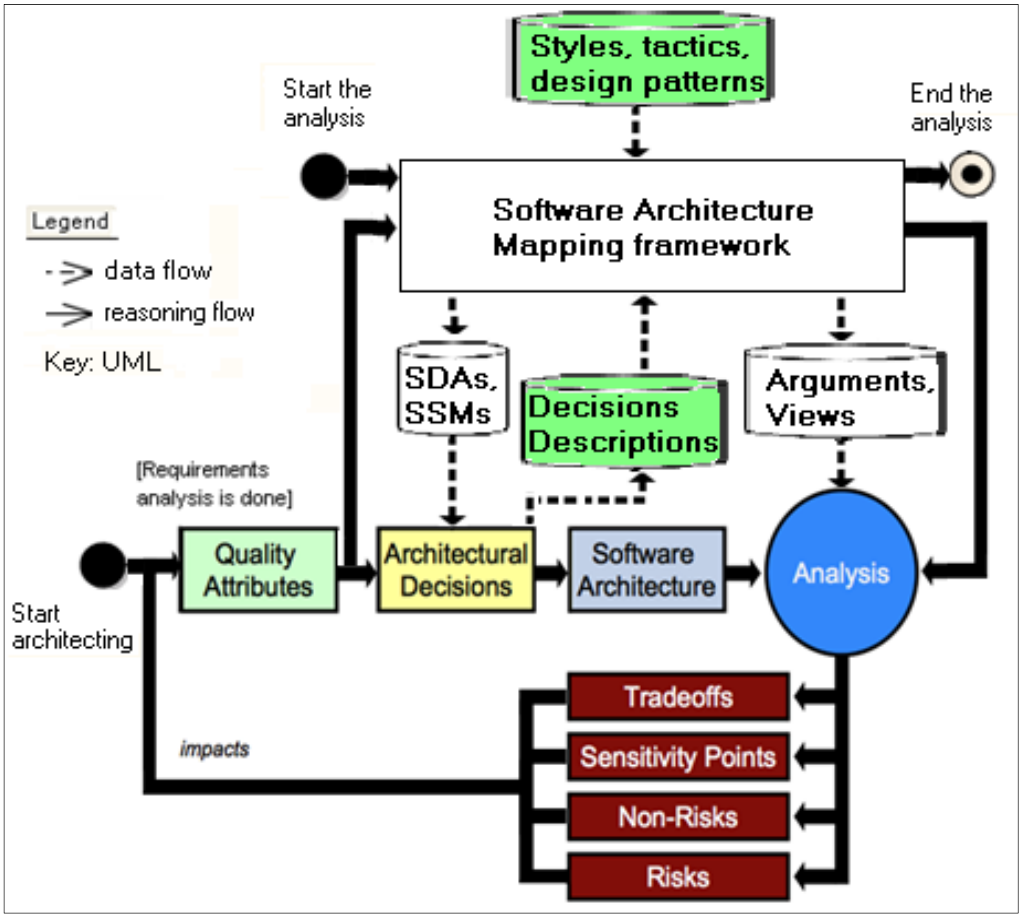


Figure 2.1 Overview of the Software Architecture Mapping (SAM) framework

The two starting points in Figure 2.1 illustrate the two use cases proposed in this thesis for the SAM framework. Firstly, the SAM process may be executed for acquiring and sharing the knowledge extracted from descriptions of styles, tactics, and design patterns. Then, the resulting design knowledge base (i.e., SDAs and SSMs) will be used to support the design process. At particular decision points in the design process (e.g., selection of a pattern [Zimm12, Zimm09]), the software designers will use the SSMs of styles, tactics, or patterns as checklists of SDAs for eliciting issues, describing arguments, and creating views. For a specific decision point, a SSM will record the general, contextual, and design knowledge [Tang06], and the arguments will record the reasoning knowledge.

2.2 The proposed Software Architecture Mapping process and roles

Figure 2.2 presents the overview of the proposed SAM process: it provides three activities (i.e., create a software structures map (SSM)), describe arguments, and analyze arguments. The SAM process aims at inferring the order of treatment of the arguments related to the utilization of particular SDAs during the design process.

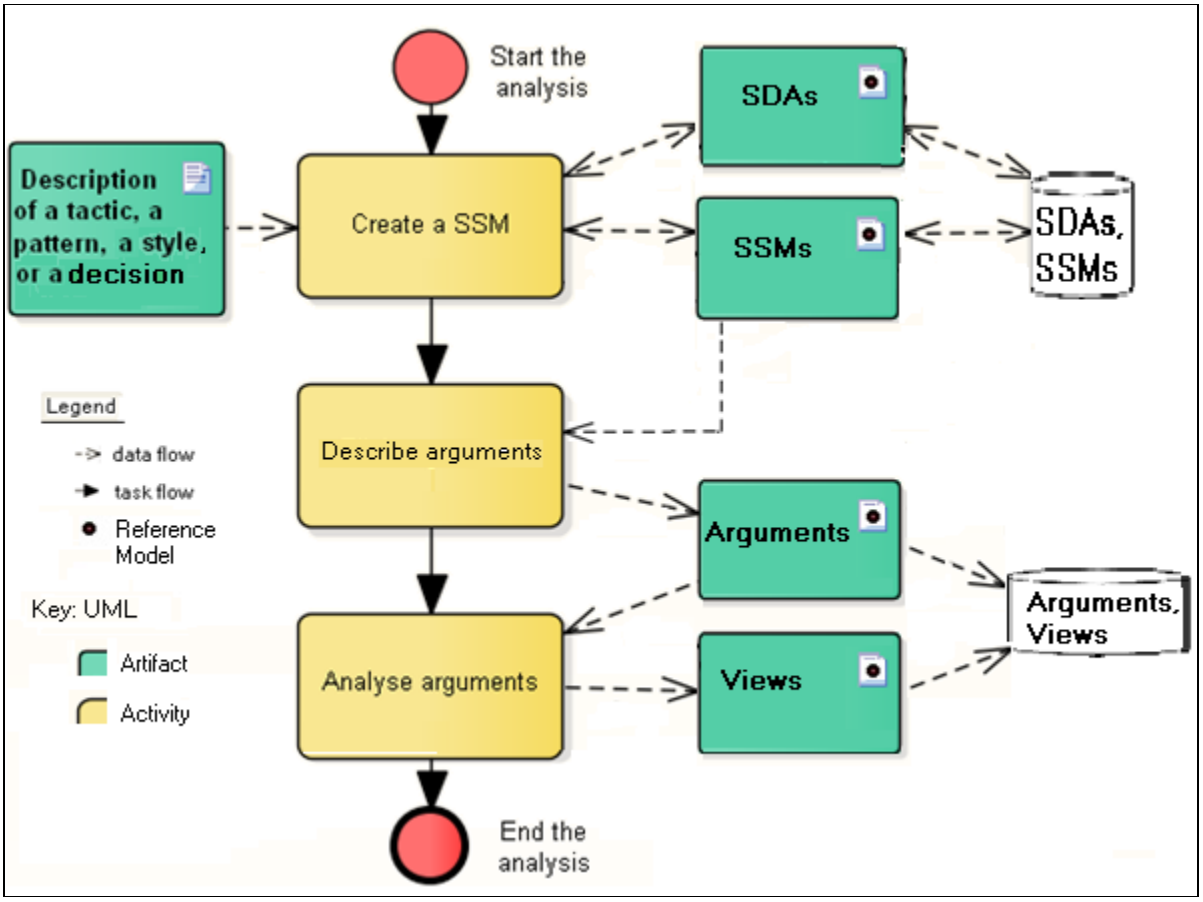


Figure 2.2 The proposed SAM process

SAM is the process of managing the design knowledge base that organizes the SDAs and the related DK used during the development process. Figure 2.3 presents the task flow and data flow that exist between the SAM process and the architecting activities from Table 1.4.

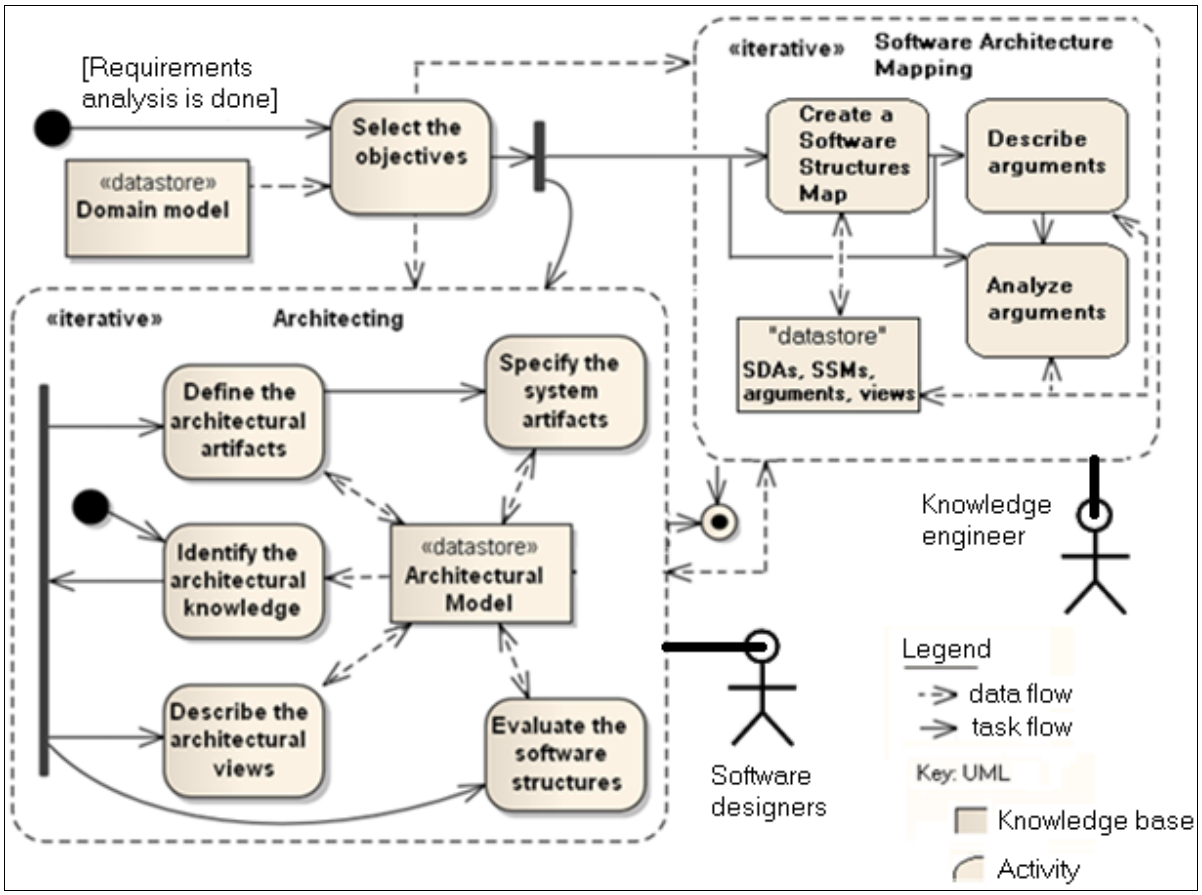


Figure 2.3 Overview of the Software Architecture Mapping process

The proposed SAM process includes three activities for supporting the analysis of a SDA, and its related SDAs and issues:

1. Create a SSM: a SSM is created for classifying and relating the SDAs in a semantic network (a SSM is a traceability matrix for the SDAs and their relationships).
2. Describe arguments: the arguments are described for eliciting the issues that relate to the SDAs.
3. Analyze arguments: the arguments are analyzed to create views that support inferring the order of treatment of the related arguments based on rankings provided during the analysis.

The SAM framework defines two phases of knowledge processing:

1. Asset creation is performed by a knowledge engineer, i.e., a software designer tasked with the creation of assets (i.e., SDAs, SSMs, arguments, and views);
2. Asset consumption is performed by software designers that use the DK in the reusable assets on their projects.

2.3 The proposed reference model

Figure 2.4 presents the proposed reference model of the SAM framework. The reference model includes four concepts:

- the software design artifact (SDA),
- the software structures map (SSM),
- the argument, and
- the view.

The argument aggregates the issue, reasoning description, dimensions, and activities.

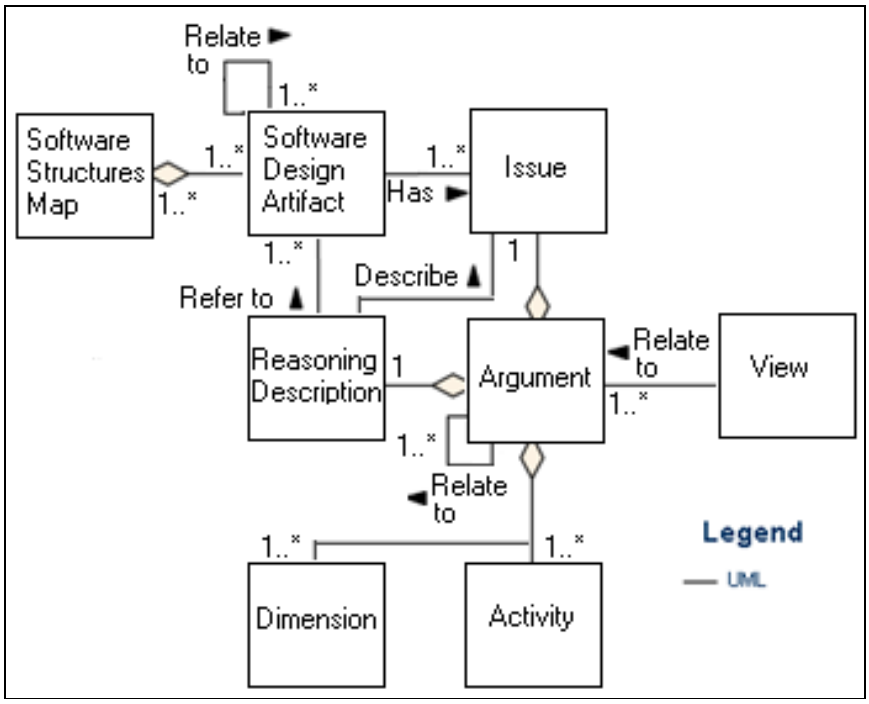


Figure 2.4 The proposed reference model of the SAM framework

Each SDA has some related SDAs and issues. The SAM framework proposes to use

1. the SSMs for structuring the SDAs,
2. the arguments for describing the issues that relate to the SDAs, and
3. the views for analyzing the impact of the arguments on dimensions and activities.

2.4 Justification of the proposed reference model

The proposed reference model addresses the conclusions retained from the literature review [Khal10, Shah09, Pari08, Tyre05]. In particular, the reference model:

- captures rationale, constraints, design decisions, and the related explanations and quantifications about how they impact objectives using SDAs, SSMs, and arguments;
- reduces the possibility to express similar concepts in different terms using finer-grained SDAs;
- takes into account all activities and SDAs identified in the literature review related to the design process and DK management;
- supports personalization for context-specific design process and DK management using personalized SSMs;
- captures the relationships between design decisions and SDAs using SSMs;
- captures the relationships between SDAs using SSMs;
- provides multiple perspectives for managing the DK using arguments and views;
- supports an integrated approach of the design process and DK management;
- captures the DK from textual catalogs using SDAs, SSMs, and arguments;
- supports the selection and comparison of the SDAs using SSMs; and
- supports the evaluation of the SDAs and the consequences of applying each of them using SSMs, arguments, and views.

2.5 Limitations of the proposed reference model

The proposed reference model has limitations with regards to the conclusions retained from the literature review [Khal10, Shah09, Pari08, Tyre05]. In particular, the reference model:

- captures the DK in a textual form;
- does not capture:
    - the relationships between design decisions,
    - the contextual knowledge (e.g., names, dates, version number),
    - attributes and types of decisions,
    - when and how decisions are made, and
    - types of decision dependencies; and
- does not make explicit the relationships between SSMs.

2.6 Positioning the SAM framework within the literature

This section aims at positioning the SAM framework as a software engineering method, a software design method, and a design knowledge management method as described in [Zimm09], and a design documentation method based on the rules described in [Clem02]. The next sections present the requirements, rules, and conclusions from the literature that have been used for assessing the SAM framework.

Methods Requirements Coverage

Table 2.1 aims at assessing the SAM framework with regards to the requirements established in CHAPTER 1 for software engineering methods, software architecture design methods, and architectural knowledge management. CHAPTER 3 introduces applications of the SAM framework that were developed during the validation process to support the assessments presented in Table 2.1.

Table 2.1 Methods requirements coverage

Requirement	SAM framework	Assessment

R1: Method anatomy = process + notation + supporting techniques and content	SAM process, descriptions formats, work instructions, techniques for classification, argumentation, and analysis	the three techniques and work instructions support the three activities of the SAM process
R2: Provide standard description format and metamodel	SDAs, issues, and arguments description formats, reference model	the description formats for issues and arguments support the reference model
R3: Be broadly applicable and actionable, e.g., provide templates and examples	Classification scheme (CS), reference model, examples of SSMs	the techniques and templates are applicable to design patterns, tactics, styles, and design decisions
R4: Provide link between requirements engineering (analysis) and design work	CS, argument format	the CS and the argument format provide this link
R5: Ease method content authoring (extensibility) and tailoring (usability)	CS, heuristics, work instructions	the CS, work instructions, and heuristics can be authored and tailored
R6: Provide multiple architectural viewpoints	CS, argument format	the CS and argument format provide multiple viewpoints
R7: Be driven by quality attributes and goals	CS, argument format	the CS and argument format provide multiple viewpoints
R8: Support decomposition of complex design issues (architectural analysis)	CS, argument format	the CS and argument format support the decomposition of complex SDAs and issues
R9: Support composition of resolved design issues (architectural synthesis)	CS, argument format	the CS and argument format support the composition of designs and rationale
R10: Provide a managed to do list	CS, argument format, analysis technique	the SSMs, arguments, and analysis technique provide

		managed to do lists
R11: Support architecture evaluation	CS, argument format, analysis technique	arguments, views, and analysis technique support architectural evaluation
R12: Obtain required knowledge	Classification technique, argument format	the classification technique and argument format provide research capabilities
R13: Tailor identified knowledge	Techniques for classification, argumentation, and analysis	the techniques support tailoring the context, general, reasoning, and design knowledge
R14: Document decisions	Techniques for classification, argumentation, and analysis	the techniques document context, general, reasoning, and design knowledge
R15: Align with other models	CS, argument format	the CS align with multiple models for design patterns, tactics, styles, and decisions

2.6.1 Assessment of the rules for architectural documentation

Table 2.2 aims at assessing the SAM framework with regards to the rules established in [Clem02] for architectural documentation. CHAPTER 3 introduces applications of the SAM framework that support the assessments presented in Table 2.2.

Table 2.2 Assessment of the rules for architectural documentation

Rule	SAM framework	Assessment
R1: Write documentation from the reader's point of view	Work instructions, classification scheme	the work instructions and classification scheme support writing documentation for a software designer's point of view
R2: Avoid unnecessary repetition	Reference model, description formats	the SDAs, SSMs, issues, arguments, and description formats reduce unnecessary repetition
R3: Avoid ambiguity	Classification scheme, description formats	the classification scheme and the description formats provide fine-grained SDAs, SSMs, and arguments
R4: Use a standard organization	Classification scheme, description formats	the classification scheme and the description formats provide the standard organization
R5: Record rationale	Classification scheme, SDAs, SSMs, arguments	the column 'Why' of the classification scheme and the arguments capture the rationale of the design decisions
R6: Keep documentation current	SAM process, work instructions, description formats	the SAM process, work instructions, and description formats aim at keeping the documentation current
R7: Review documentation for fitness of purpose	Work instructions, reference model, description formats	the work instructions aim at producing fine-grained SDAs, SSMs, arguments, and description formats

2.6.2 Assessment regarding the related works on design decisions

Table 2.3 aims at assessing the SAM framework with regards to the claims established in the literature review for design decisions. CHAPTER 3 introduces applications of the SAM framework that support the assessment presented in Table 2.3.

Table 2.3 Assessment of the SAM framework for the related works on design decisions

The SAM framework DOES realize the following claim
+ describe an architectural decision model
+ describe a metamodel and modeling principles for design decisions
+ capture decisions required, decisions made, and possible solutions
+ define a semantic ontology for decisions
+ separate decisions required and decisions made
+ propose concepts for structuring decision models
+ define alternatives as a separate entity
+ define a template for documenting architectural design decisions
+ view software architecture as a composition of a set of design decisions
+ treat decisions as a first class architecture design concept
+ outline the attributes that are required to capture design knowledge
+ propose solutions for the reuse of architectural decision knowledge
+ define a template for capturing architectural decisions
+ define metamodels for managing decisions
+ capture design decisions, design rationale, and design models
+ support basic decisions dependencies
+ support traceability between requirements, design decisions, and design
+ support software architecture design, documentation, and evaluation
+ provide a knowledge repository of generic and specific knowledge
+ document the chain of dependencies between decisions
+ provide support for managing the decisions and the design rationale
+ relate design decisions to individual architectural parts

2.7 Limitations of the SAM framework

Table 2.4 presents the limitations of the SAM framework with regards to the claims established in the literature review for the related works on design decisions.

Table 2.4 Limits of the SAM framework

The SAM framework DOES NOT realize the following claim
- describe dependency relations, integrity constraints, and production rules
- describe formally what is design decisions, attributes and types of decisions
- describe when and how decisions are made
- define types of decision dependencies
- treat a design problem and its solution as distinct entities
- support variability management
- ensure the integrity of the decision model

CHAPTER 3

EXAMPLES OF UTILIZATION OF THE SAM FRAMEWORK

This chapter presents seven examples of utilization of the SAM framework, including five cases studies and two experiments. The order of presentation corresponds to the order of realization of the examples. The objectives of the cases studies and experiments were oriented towards the evaluation of the relevance, value, and effectiveness of the SAM framework:

- Verify that the SAM framework meets the needs for which it was developed
- Demonstrate the value of the SAM framework for the user
- Identify the strengths and weaknesses of the SAM framework
- Determine how the SAM framework should be improved

Section 3.1 describes the SSMs, arguments, and views that were produced for the context of projects developing software-intensive systems (SISs). This case study provides an example of the utilization of the classification scheme (CS) and the SSM description format of the SAM framework. APPENDIX I describes the SDAs that were classified using the CS for this case study. APPENDIX II describes the context and SSMs that were created.

Section 3.2 describes the SSMs, arguments, and views that were produced for the context of an undergraduate course on object-oriented software design at ETS. This case study provides an example of the utilization of the CS, the classification and argumentation techniques, and the description formats of the SAM framework. This case study describes a SSM and arguments for a utilization of the Template Method (TM) design pattern published at the Software Engineering and Knowledge Engineering (SEKE 2013) conference.

Section 3.3 describes the experiment with human participants that was conducted in the context of a graduate course in software engineering at ETS. This experiment provides an example of application of the reference model and description formats of the SAM framework. APPENDIX III describes the inputs, outputs, and analysis of the experiment.

Section 3.4 describes the SSMs that were produced for encoding catalogs of styles [Clem02], design patterns [Gamm94], and tactics [Bass03], and an analysis of the outputs of the case study. This case study provides an example of the utilization of the classification scheme and the classification technique of the SAM framework. APPENDIX IV describes the SSMs of the modifiability tactics [Bass03] produced for this case study. This case study has been published as a paper at the Software Engineering and Knowledge Engineering (SEKE 2015) conference. The detailed version of this case study has been submitted to the Journal of Software Engineering and Knowledge Engineering (JSEKE 2016).

Section 3.5 describes the SSMs and arguments that were produced for developing the web site of a small organisation that planned to sell products online, and an analysis of the outputs of the case study. This case study was performed to evaluate the technical feasibility of applying the techniques of the SAM framework in a small web engineering problem. APPENDIX V describes the outputs of the case study.

Section 3.6 describes the controlled experiment and the SSMs and arguments that were produced by a human participant who applied the SAM framework to the web engineering problem described in **Section 3.5**. The experiment was conducted to evaluate the usability of the SAM framework. APPENDIX VI describes the work statement for the experiment.

In addition, a support tool was developed and used for managing the SDAs and SSMs of the SAM framework. This prototype provides the SDA and SSM managers. The SDA manager implements the classification scheme of the SAM framework and is based on the Java programming language and the Eclipse development platform. The SSM manager implements a Java-based compiler that provides a lexical and syntactical parser for the SSMs and arguments of the SAM framework. The compiler was based on SableCC [Gagn98]. This prototype was not planned for this research project and is not described in this thesis. The next sections present the validation activities that were conducted for the research project.

3.1 Case study: the SAM framework in the context of a SIS

For this case study, a SSM and the related contextual reasoning were created for analyzing the architectural concern “Scope of the framework”. This SDA drives many design decisions. This section describes two versions of the SSM and the arguments for illustrating how a SSM is iteratively created. The SSMs and the context were based on the technical documentation and industrial background of projects developing full flight simulators (FFS) [Bass03]. APPENDIX I describes the SDAs that were used to create the SSM. APPENDIX II describes the detailed SSM that was created by executing many iterations of the SAM process, and the reasoning for this case study. The following sub-sections present:

- the introduction to the context of software cockpit systems (Section 3.1.1);
- the SSM created during iteration 1 (Section 3.1.2);
- some arguments described during iteration 1 (Section 3.1.3);
- some arguments analyzed during iteration 1 (Section 3.1.4);
- the SSM updated during iteration 2 (Section 3.1.5);
- an analysis of the case study (Section 3.1.6); and
- a description of how the SAM framework addresses the conclusions (Section 3.1.7).

3.1.1 Introduction to the context of software cockpit systems

Projects involving development of FFS training devices deal with constraints on time and budgets for flight test data, vendor data, aircraft parts, engineering hours, verification, validation, and customization. These training devices must meet very aggressive cost targets and regulation controls. This competitive context led organizations to make a technological paradigms shift from procedural approaches to object-oriented and component-based approaches. The development team is required to design the software cockpit system (SCS) framework that will support the implementation of various SCSs. The framework is required to provide common classes that SCSs will reuse for simulating the cockpit of various airplanes. The major goals are to reduce maintenance costs and eliminate design defects in SCSs.

3.1.2 The activity "Create a SSM" – iteration 1

The first activity of the SAM process aims at creating a SSM.

Table 3.1 presents the SSM that was manually created during the first iteration of the SAM process.

Table 3.1 The SSM of the architectural concern "Scope of the framework" – version 1

Interrogative	
SDA type	**SDA description**
Why	
Goal	Reduce maintenance costs
Goal	Eliminate design defects
Architectural concern	Scope of the framework
When	
Situational factor	Legacy systems transformation strategy
Organizational risk	Development paradigm shift
Regulation	FFS Level D control
What	
Constraint	Shorten schedule
	Limited budget
Property	Extensibility
	Reusability
	Framework
	Object-oriented paradigm
	Component-based paradigm
Which	
Style	Layered system
Architectural tactic	Abstract Common Services
Design fragment	SCS framework layer
	SCS layer

3.1.3 The activity "Describe arguments"

The second activity of the SAM process aims at describing the arguments related to a SSM. For this case study, some issues and arguments were described for identifying additional SDAs that relate to the SDA "Scope of the framework". The examination of
Table 3.1 has allowed to describe the issues in Table 3.2 and the arguments in Table 3.3. The arguments provide the reasoning descriptions about the issues and refer to activities and dimensions that are strengthened (+) or weakened (-) by the issues.

For this example, the arguments refer to:

- three activities – see Table 3.2:
  - managing (M),
  - designing (D), and
  - implementing (I), and
- three dimensions – Table 3.3:
  - functions (F),
  - people (P), and
  - quality (Q).

Table 3.2 describes some issues and activities that were considered. The table provides the issue number and description, and the activities inferred from each issue description.

Table 3.2 Issues related to the architectural concern "Scope of the framework"

Issue #	Issue description (SDA + verb + complement)	Activities
1	The object-oriented paradigm is not well mastered	D I M
2	The component-based paradigm is not well mastered	D I M
3	The reusability objectives are not defined	D I M
4	The extensibility objectives are not well defined	D I M
5	The layered system style is not well mastered	D I M

Table 3.3 presents some arguments that were described for explaining the issues, and the dimensions inferred from the reasoning description of each argument. For each argument, the table provides the argument number, its related issue number, its reasoning description, and the dimensions impacted (- or +) by the argument. For example, the argument #1 may negatively impact people and quality considering that the team member role will be executed by humans and their object oriented skills may impact the quality of the software product.

Table 3.3 Arguments related to the SSM of the concern "Scope of the framework"

Arg. #	Issue #	Reasoning description	Dim.
1	1	The candidate team members lack of skills, expertise, and knowledge for using the object-oriented paradigm	-P -Q
2	2	The software designers have difficulty to define the software components of the SCS	-Q
3	3	The legacy systems transformation strategy make it difficult to validate the reusability objectives of the SCS framework	-F -Q
4	4	The software designers have difficulty to establish a consensus for the extensibility objectives of the SCS framework	-P -Q
5	5	The design constraints of the layered system style have not been examined for the SCS framework	-Q

3.1.4 The activity "Analyze arguments"

The ranking (H: high, M: medium, L: low, and X: not relevant) was used to describe how much each activity, dimension, and argument from Table 3.3 is relevant to the context. Table 3.4 presents a fictive contextualization of the activities, dimensions, and arguments. The ranking has been quantified, for illustrative purposes, as H=100, M=10, L=1, and X=0.

Table 3.4 indicates that designing and quality are the most important factors for the context. As a result of the rankings, the weight of argument #1 (M=10) will be multiplied by ten thousand (10000 = 100 * 100) in the view's cell that intersects the design activity (H) and quality dimension (H) (i.e., argument 1 is part of this cell) while its weight will be multiplied by one thousand (1000 = 100 * 10) in the view's cell that intersects the design activity (H) and functions dimension (M). A total impact value is then computed for each cell of the view by summing the multiplied weights (i.e., argument's ranking * activity's ranking * dimension's ranking) of the arguments it contains. These values are then translated into priorities (1 is the highest priority). The priorities in Table 3.5 proposes the following order of treatment for the arguments #1 to #5: 4 (H*H*H=1000000), 2 (H*H*H=1000000), 1 (M*H*H =100000), 3 (L*H*H=10000), and 5 (X*H*H=0).

Table 3.4 Rankings for the activities, dimensions, and arguments of the SCS framework

Activities' rankings for the analysis	
Implementing	L
Designing	H
Managing	M

Dimensions' rankings for the analysis	
People	M
Functions	M
Quality	H

Arguments' rankings		**Related factors**	
Argument	Iteration 1	Activities	Dimensions
1	M	D I M	P Q
2	H	D I M	Q
3	L	D I M	F Q
4	H	D I M	P Q
5	X	D I M	Q

Table 3.5 View of the SCS framework arguments

Iteration 1	**Dimension**			
Activity	F	P	Q	
D	4	3	1	**8**
I	9	8	7	**24**
M	6	5	2	**13**
	19	**16**	**10**	

3.1.5 The activity "Create a SSM" – iteration 2

Table 3.6 presents the updated version of the SSM proposed for the architectural concern "Scope of the framework". The evolution of the SSM results from the design decisions made to address the arguments. The addition of some SDAs will support new arguments, and it will impact the ranking of some arguments. The cycle (SDAs>Arguments>Decisions>SDAs) may continue until the arguments rankings equal some thresholds (e.g., L). For example, the argument #4 in Table 3.3 is addressed by adding two SDAs to the SSM in Table 3.6.

Table 3.6 Added SDAs for the SSM of the concern "Scope of the framework"

Arg. #	Issue #	Reasoning description	Dim.
4	4	The software designers have difficulty to establish a consensus for the extensibility objectives of the SCS framework	-P -Q

Interrogative	
SDA Type	**SDA Description**
What	
Scenario	Every software system implements a common interface
Which	
Architectural tactic	Localize changes

3.1.6 Analysis of the case study

This case study reinforced evidence regarding the validity of the proposed classification scheme (CS) and the need for techniques and work instructions that support populating knowledge bases of SDAs and SSMs. The following conclusions result from the analysis of the case study:

- Reliability - A large number of SDAs were classified using all cells of the CS. No SDA was rejected. The capacity of the CS for classifying a large number SDA types in the context of complex SIS provided evidence that the CS is reliable.

- Usability - The SDA types were used for discerning the semantic of each SDA. Most of the SDAs were easy to classify. However, some SDAs were difficult to classify and reinforced evidence regarding the need for exclusive and description criteria for describing SDAs.

- Reliability - The focus of the case study was the creation of the SSM. The arguments were described in APPENDIX II to provide the reasoning descriptions of the SSM. These arguments lack precision and format. In addition, the views were described for illustrative purposes and were not used for the case study. The views and arguments reinforced evidence regarding the need for techniques and a support tool for managing the DK. In particular, the views were not useful without tool due to management overhead.

3.1.7 How the SAM framework addresses the conclusions of the case study

The SAM framework addresses some conclusions of the case study.

For the usability,

- the classification technique and work instructions proposed in CHAPTER 4 support populating knowledge bases of SDAs and SSMs.

For the reliability,

- the argumentation technique proposed in CHAPTER 5 supports describing the arguments; and
- the description formats proposed in Section 5.5.1 and Section 5.7.1 for the issues and arguments should reduce the lack of precision and format of the descriptions.

3.2 Case study: the SAM framework for analysing the TM design pattern

This case study has been developed for applying the classification scheme, the argumentation technique, and the argument description format of the SAM framework. This section presents the SSM and arguments created for analysing the Template Method (TM) design pattern. This SDA relates to many SDAs. This section describes the context of the SSM and some related arguments for illustrating how a SSM is created using a catalog of design patterns [Gamm95]. The following sub-sections present:

- the description of the TM design pattern (Section 3.2.1);
- the SSM created for the TM description in [Gamm95] (Section 3.2.2);
- some arguments related to the TM design pattern (Section 3.2.3); and
- an analysis of the case study (Section 3.2.4).

3.2.1 Description of the TM design pattern

The Template Method (TM) design pattern is used for providing reusability and extensibility of algorithms in object-oriented software [Gamm95]. It aims to implement the skeleton of an algorithm in a base class, and calls primitive methods that subclasses override to provide concrete behavior. The base class interface declares the algorithm as a template method, which calls abstract primitive methods that represent the algorithm’s variation points.

The subclasses implement the primitives to specialize the algorithm. As a result, the algorithm’s structure is written only once and is indirectly specialized in subclasses, which reduces duplication of code and enforces class interface stability. Also, the template method allows the addition of instrumentation in the base class, and lightens users' duty since he is no longer required to call a primitive.

3.2.2 SSM of the TM design pattern

Table 3.7 presents the SSM of the TM design pattern, as described in [Gamm95]. The SDA Id is used to establish the relationships in Table 3.8. The SDA Type refers to a cell in the classification scheme. The SDA description provides the meaning of the SDA.

Table 3.7 The SSM of the Template Method design pattern

Software Design Artifacts		
Id	**Type**	**Description**
Why		
Dc1	Design concern	Avoid code duplication
Dc2		Control subclasses extension
Ac3		Localize changes
Ac4		Prevention of ripple effect
Dr1	Design rationale	Fix the steps of the algorithm and their ordering
Dr2		Let subclasses define the steps of the algorithm
Dr3		Maintain the algorithm's structure
Dr4		Limit extension points
Dr6		Provide default behavior
Dr7		Control access to the operations
When		
Si1	Situational factor	Multiple kinds of primitive operations
Co1	Convention	Naming convention
Sy1	Symbol	UML notation
What		
Re1	Requirement	Specify for subclass writers which operations are hooks
Re2		Specify for subclass writers which operations are abstract
Pr1	Property	Object-oriented paradigm
Pr4		Object-oriented programming language
Pr2		Reusability
Pr3		Extensibility
Op1	Operational.	Define an abstract base class
Op4		Define a template method
Op5		Define a concrete child class

Op5		Define hook operations
Op7		Declare a final template method
Op8		Declare protected primitive operations
Op9		Declare abstract primitive operations
Vp1	Viewpoint	Class diagram
Vp2		Sequence diagram
Vp3		Package diagram
Which		
Ro1	Role	Subclass writers
Sp1	Structural pattern	Template Method
Sp2		Factory Method
Ta1	Tactic	Abstract Common Services
Ta2		Information hiding
Ta3		Semantic coherence
Ta4		Maintain existing interface
Ta5		Use Encapsulation
Ta6		Use an Intermediary
Ta7		Restrict Communication Paths
Sf1	Structural fragment	C++ language
Sf2		Class library
Ss1	Software structure	Abstract class definition
Ss2		Concrete class definition
Ss3		Template method definition
Ss4		Primitive operation declaration
Ss5		Primitive operation definition
Ss6		Hook operation definition
How		
Be1	Behavior	The template method controls the order of execution
Be4		The hook operations do nothing by default
Where		
Af1	Allocation fragment	Class file
As1	Allocation structure	AbstractTemplate.cpp
As2		ConcreteTemplate.cpp

Table 3.8 summarizes the relationships extracted from [Gamm95] for the proposed SDAs.

Table 3.8 Relationships between the SDAs of the Template Method design pattern

SDA	Relationship	SDA
Sp1	Mandatory	Ta1
Sp1	Optional	Ta2
Sp1	Optional	Ta3
Sp1	Optional	Ta4
Sp1	Optional	Ta5
Sp1	Optional	Ta6
Af2	Uses	Af1
Ss1	Generalizes	Ss2
Ss2	Specializes	Ss1
Ss3	Calls	Ss5
Ss3	Calls	Ss6
Ss1	Composes	Ss3
Ss1	Composes	Ss4
Ss1	Composes	Ss6
Ss2	Composes	Ss5

3.2.3 Arguments related to the TM design pattern

The proposed argumentation technique of the SAM framework aims at describing issues that occur by using the SDAs. This section presents the results of applying the technique to the utilization of the Template Method (TM) design pattern.

Step 1) Task 1 to Task 3 for eliciting issues

The first step aims at eliciting issues that occur by using the TM design pattern. Table 3.7 provides the SSM of the TM, as described in [Gamm95]. For this case study, three activities (i.e., managing, designing, and implementing) related to the classification scheme (CS) (see Section 4.5.2) and the design knowledge management (DKM) process (see Section 1.7.3) were considered:

- Managing (M) refers to the activity "Select the objectives" of the CS or any activity of the DKM process; it deals with roles (e.g., subclass writers), situational factors, and conventions (e.g., naming convention) that constitute the organizational system.
- Designing (D) refers to the activity "Identify knowledge" or "Specify system" of the CS. Designing deals with the detailed structures (e.g., abstract class) and the requirements (e.g., a threshold for the execution time) that refine the architectural properties.
- Implementing (I) usually deals with algorithms and specific characteristics (e.g., which keyword: while or for).

The selected SDAs, relationships, and activities provide the SSMs for eliciting the issues. Each selected SDA will be examined as a root for trees of related SDAs. Some issues will be elicited by focusing on one SDA after another. Table 3.9 presents the descriptions of some issues that may hinder the usage of the TM design pattern, and the related activities. Each issue refers to either a SDA or a relationship between two SDAs.

Table 3.9 Some issues related to the SDAs of the TM design pattern

Issue #	Issue description (SDA + verb + complement)	Activities
1	The naming convention is not described	M I D
2	The template method behavior is subject to change	M I D
3	The primitive operation is not well identified	I D
4	The hook operation is not well identified	I D
5	The object-oriented paradigm is not well mastered	M I D
6	The reusability objective is not well defined	M I D
7	The extensibility objective is not well defined	M I D
8	The subclass writer role is not described	M I D
9	The programming language is not well mastered	M I D
10	The primitive operation can be called by any caller	I D
11	The subclass writer does not use the naming conv.	M I
12	The abstract class lacks cohesion	D
13	The template method may be overridden	M I

In the next step, the arguments will tie each issue to some SDAs, activities, and dimensions.

Step 2) Task 4 and Task 5 for describing the arguments

The reasoning description relates an issue to SDAs, activities, and dimensions constituting an argument; it describes the reasoning that supports the issue description. For this case study, the dimensions people (P), quality (Q), and functions (F) will be examined. Table 3.10 describes some arguments related to the TM design pattern. For each argument, the table provides the argument number, its related issue number, its reasoning description, and the dimensions impacted (- or +) by the argument.

For example, the issue #5 may impact people and quality considering that the software designer's role will be executed by a human and his object oriented skills may impact the quality of the software product.

Table 3.10 Arguments related to the TM design pattern

Arg. #	Issue #	Reasoning description (the SDAs are underlined)	Dim.
1	1	The subclass writer has difficulty to identify the template method, the primitive operation, and the hook operation	-P Q
2	2	Modifying the template method's behavior will impact the software products that depend on this behavior	-F Q
3	3	The subclass writer has difficulty to identify the primitive operation	-Q
4	4	The subclass writer has difficulty to identify the hook operation	-F Q
5	5	Using the object-oriented paradigm requires levels of skills, expertise, and knowledge of candidate team members	-P Q
6	6	Modifying the reusability objective requires modifying the interface of the class that implements the template method	-F Q
7	7	Modifying the extensibility objective requires modifying the interface of the class that implements the template method	-F Q
8	8	The subclass writer has difficulty to identify the template method, the primitive operation, and the hook operation	-P
9	9	The subclass writer has difficulty to use the object-oriented language	-P Q
10	10	An uncontrolled call to the primitive operation will cause a functional problem	-F
11	12	There are too many primitives operations	-Q
12	12	The low cohesion makes reusing the abstract class more tedious	-Q
13	12	The class cohesion is proper for the team's expertise	+Q
14	13	The subclass writer may override the template method	-F
15	13	A final method cannot be overridden	+F
16	13	The final mechanism is hackable	-FQ

The arguments describe plausible impacts that may occur by using the design pattern. However, only appropriate descriptions and utilizations of the SDA lead to planned impacts on the dimensions.

3.2.4 Analysis of the case study

This case study reinforced evidence regarding the validity of the proposed classification scheme (CS), reference model, and description formats of the SAM framework. The following conclusions result from the analysis of the case study:

- Reliability - A large number of SDAs, issues, and arguments were described using the proposed description formats. This case study provided evidence that the reference model and description formats have the potential for expressing all categories of DK.

- Usability - The argumentation technique proposed in CHAPTER 5 was used to describe the issues and the arguments. The description formats and the verbs, activities, and dimensions have proved to be useful checklists for identifying issues and arguments.

- Usability - The focus of the case study was the creation of the SSM and the description of the related issues and arguments. The classification and argumentation techniques have proved useful for populating the DK base of reusable SDAs, issues, and arguments in a systematic manner.

- Usability - The description of the issues reinforced evidence regarding the need for a tool-support for managing the DK. Many issues should have been inferred using a tool-support and the activities of the DK management.

The descriptions examined for identifying the SDAs and relationships were formatted and divided into sections (e.g., Pattern Name, Intent, Structure, Collaborations, Consequences) according to a template, as described in [Gran08, Gamm95]. Each section of a design pattern description provided knowledge for identifying particular SDAs. Table 3.11 summarizes from which sections of the design pattern's template used in [Gamm95] were extracted the information for the proposed SDAs.

Table 3.11 Design pattern description: sections and SDAs

Sections of the design pattern description	SDA type
Intent, Motivation	Rationale
Consequences	Property
Implementation, Participants, Sample Code, Structure	Operationalization
Consequences, Participants, Sample Code	Behavior
Collaborations, Structure	Structure
Implementation	Convention
Consequences	Procedure
Consequences	Role
Applicability, Implementation, Known Uses	Situational factor

Table 3.12 summarizes in which cells of the classification scheme were classified the SDAs used for describing the TM design pattern in [Gamm95].

Table 3.12 Classification counts for the SDAs of the TM design pattern

	Why	**When**	**What**	**Which**	**How**	**Where**	
Objectives		1	2	1			**4**
Knowledge	4	1	4	9			**18**
Fragment				2		1	**3**
Structure	7		7	6	2	2	**24**
Description		1	3				**4**
Evaluation							**0**
	11	**3**	**16**	**18**	**2**	**3**	**53**

The counts in Table 3.12 indicate that the description of the TM design pattern provides a higher number of SDAs related to software structure than the number of SDAs related to software fragment. The counts are coherent with the claim that design patterns describe more detailed designs than tactics and styles that describe architectural designs. In addition, the column “Which” in Table 3.12 provides the highest count, which is coherent with the categorization of the TM as a structural design pattern proposed by the GOF in [Gamm95].

3.3 Experiment: human participants for applying the reference model

This section describes the experiment that was conducted to evaluate the reference model, the work instructions, and the argument description format of the SAM framework in the context of the graduate course SYS869 “Sujets spéciaux: Expérimentation en génie logiciel”. The following sub-sections present:

- the experiment planning (Section 3.3.1);
- the experiment process and schedule (Section 3.3.2);
- the experiment subjects, groups, and profiles (Section 3.3.3);
- the proposed design context and the collected data (Section 3.3.4);
- the statistics from the collected data (Section 3.3.5);
- the analysis of the experiment (Section 3.3.6);
- the limitations of the experiment (Section 3.3.7); and
- a description of how the SAM framework addresses the conclusions of the experiment (Section 3.3.8).

3.3.1 Experiment planning

The object of the experiment was the proposed reference model of the SAM framework. The focus of the experiment was the evaluation of the data collected from the participants who used the analysis approach proposed by the SAM framework. The following characteristics were examined:

- Reliability (repeatable outputs);
- Efficiency (time, effort, cost, results);
- Usability (required background), and
- Accuracy (validity of the evaluation)

3.3.2 Experiment process and schedule

Figure 3.1 presents the process of three activities (i.e., Preparation, Execution, and Evaluation) planned for the experiment. The preparation activity was planned for presenting the experiment process and related descriptions, analysis model, and forms to the participants. The execution activity was planned for analyzing a design problem and the related software design artifacts using the proposed analysis approach, model, and form. The evaluation activity was planned for analyzing the participants' forms and reporting on the experiment.

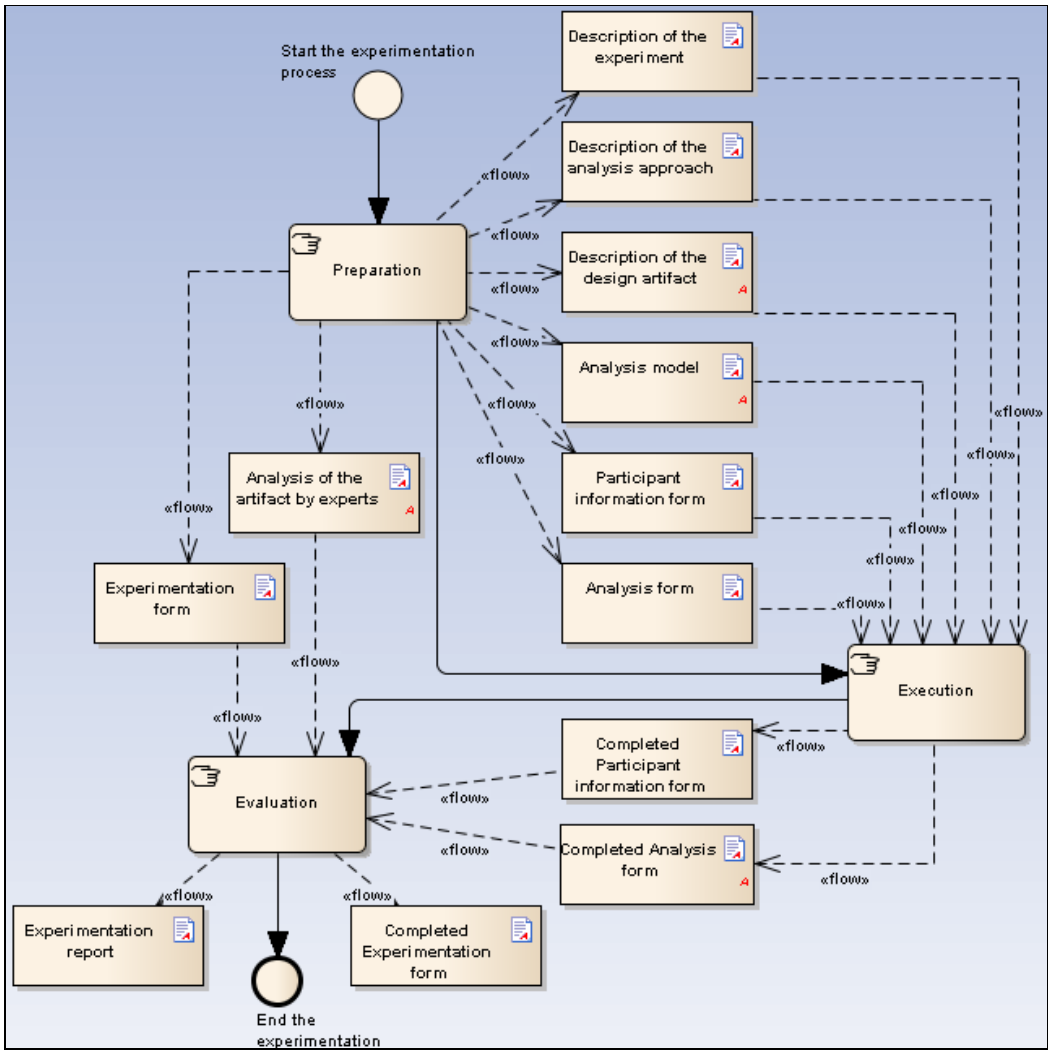


Figure 3.1 Overview of the process planned for the experiment

During the experiment, the participants were asked to analyze the description of the software framework introduced in Section 3.3.4. One hour was scheduled for executing the experiment in a workshop. The execution of the experiment includes:

- a presentation (30 min.) to the participants of
    - the experiment;
    - the approach described in CHAPTER 2;
    - the design problem to be examined; and
    - the software design artifacts to be examined.
- the individual execution of the analysis by the participants (30 min.)
    - Analyze the proposed design problem;
    - Analyze the eight proposed issues;
    - Elicit two additional issues;
    - Describe ten arguments;
    - Rank the arguments.

3.3.3 Experiment subjects, groups, and profiles

Twenty participants participated in the experiment. The participants of the experiment were the graduate students registered in SYS869 2013, as well as practitioners invited to the experiment workshop organized at ETS during the summer of 2013. There was no risk to the people involved in this experiment. No identifying information from participants was collected. Participants voluntarily participated in the experiment by attending the training and the experiment sessions.

The participants were grouped according to the following profiles:

- Subjects of the study: post-graduate and undergraduate students, from academia and industry

- Selection criteria: three (3) profiles, according to the level of experience in designing software, as follows:

  Profile#1: participants with less than two years in designing software.
  These were graduate students in engineering or science; or were students at the undergraduate level in software engineering
  Profile#2: practitioners with two (2) years of experience in designing software
  Profile#3: practitioners with five (5) years of experience in designing software

From the participants' profiles, the participants were clustered into three groups for preparing the experiment (i.e., for adjusting the training to be given to the participants before the execution of the experiment):

- Group 1: limited knowledge (30 min. training)
  - Background: science, engineering (e.g., project manager and coder)
  - Training: software development, design patterns, analysis process and model
- Group 2: sufficient knowledge (20 min. training)
  - Background: design activity, software design patterns
  - Training: software development, analysis process and model
- Group 3: advanced knowledge (10 min. training)
  - Background: software development, design activity, software design patterns
  - Training: analysis process and model

3.3.4 The design context and collected data

The design context of this experiment was based on the project proposed to undergraduate students for designing a software framework that provides the skeleton of a dice game. The design of the software framework was required to provide a set of classes that can be reused and extended to allow the software implementation of various dice games. Three patterns were required to be used in this project: Iterator, Template Method (TM), and Strategy. The resulting Dice Game Software Framework (DGSF) was required to be simple enough to be understood by junior programmers with backgrounds only in procedural programming. The experiment was planned for a manual data collection by the participants in a workshop. From the participant and analysis forms, the data in Appendix III were collected.

3.3.5 Statistics from the collected data

The following information was obtained from the analysis of the collected data. Table 3.13 presents the number of participants according to the groups and profiles, and the number of missing responses in their analysis forms out of a total of ten possible responses.

Table 3.13 Number of participants and missing responses, and ratio of missing responses

	Nb. Participants	**Nb. Missing Resp.**	**Nb. Missing Resp. / Nb. Part.**
Manager	8	49	6,1
Architect	3	17	5,7
Designer	4	32	8,0
Programmer	5	38	7,6
Profile 1	9	76	8,4
Profile 2	5	22	4,4
Profile 3	6	38	6,3

3.3.6 Analysis of the experiment

The results reinforced evidence regarding the need for support for the proposed analysis approach, including techniques and work instructions for supporting it. In particular, the following conclusions result from the analysis:

- Reliability - A large number of arguments should be rejected due to the lack of consistency between the reasoning description and the impacted activities and dimensions of each argument.
- Reliability - Combining the responses of the participants for each particular issue leads to similar impacted activities and dimensions for every argument (i.e., every argument has impact on all the proposed activities and dimensions). However, the related reasoning descriptions do not describe all the proposed impacts on the activity and dimension identified. In addition, the architects provide a more consistent set of impacted activities and dimensions for each argument.
- Reliability - Many arguments require interpretation due to the lack of precision and format in the reasoning description, issue, and argument. The participants provide sentences that lack semantics and syntax.
- Usability - The architects provides the highest number of responses and number of valid arguments. This is consistent with the proposed design problem context related to the architecture of a framework.
- Usability - The participants provide impacted activities and dimensions that seem to be consistent with their backgrounds and profiles (i.e., managers refers to people, architect refers to quality, etc.).

- Accuracy - In many cases, for a particular issue, the related reasoning description provides new issues instead of explaining how the examined issue impacts the activities and dimensions. This indicates a lack of understanding of the purpose of the reasoning description.
- Efficiency - A large number of participants do not provide the two additional issues that were required in the analysis form. This indicates that issue identification is complex or that the allowed time and training for the experiment was not sufficient.

3.3.7 Limitations of the experiment

The generalization of results is limited due to the fact that:

- only twenty participants participated in the experiment,
- only one hour was used to conduct the experiment,
- only an academic design problem context was used, and
- multiple interpretations of the arguments were possible.

In addition, the criteria for the rejection of an argument need to be clarified.

3.3.8 How the SAM framework addresses the conclusions of the experiment

The SAM framework addresses some conclusions of the experiment.

For reliability,

- the argument validation heuristics proposed in Section 5.7.2 should reduce the number of arguments rejected due to a lack of coherence;
- the description formats proposed in Section 5.5.1 for the issues and in Section 5.7.1 for the arguments should reduce the lack of precision and format of the descriptions.

For accuracy,

- the argumentation technique proposed in CHAPTER 5 supports describing how the issues impact the activities and dimensions.

For efficiency,

- the classification technique proposed in CHAPTER 4 supports populating knowledge bases of SDAs and SSMs that should make the elicitation of issues more effective.

3.4 Case study: the classification technique for analyzing catalogs of DK

For this case study, many SSMs were created for populating a base of reusable DK using the proposed classification technique. The descriptions of the tactics in [Bass03], the design patterns in [Gamm95], and the styles in [Clem03] were used to create the corresponding SSMs. Any tactic, design pattern, or style being examined drove the inclusion of the other SDAs from its description into the SSM (i.e., any SDA in the SSM is cohesive with the tactic, design pattern, or style for which the SSM is being created). The following sub-sections presents:

- the SSM of the Layered System style (Section 3.4.1);
- the SSM of the modifiability tactics (Section 3.4.2); and
- an analysis of the results of the case study (Section 3.4.3).

3.4.1 SSM of the Layered style

Table 3.14 presents the SSM of the Layered style described in [Clem03].

Table 3.14 The SSM of the Layered style

SDA	Type	Description
Why		
Ac1	Concern	Manage complexity
Ac2	Concern	Communicate the structure
Ac3	Concern	Localize changes
Ac4	Concern	Prevention of ripple effect
Ara1	Rationale	Partition software into layers with public interfaces
Ar2	Rationale	Isolate each layer from changes in other layers
Vd1	Description	Inter and intra-layer usage rules
Vd2	Description	Exceptions to the usage rules
When		
Si1	Situational	Unused services in a layer

Si2	Situational	Multiple layers
Ari1	Risk	Assumptions about layer's properties
Ari2	Risk	Restricted well-defined upward usages
Ari3	Risk	Layer bridging
Sy1	Symbol	UML notation
What		
Co1	Constraint	If layer A is above layer B, then layer B cannot be above layer A
Co2	Constraint	Every unit of software is allocated to exactly one layer
Re1	Requirement	Every unit of software has a platform independent interface
Re2	Requirement	Layers interact according to a strict downward ordering relation
Pr1	Property	Modifiability
Pr2	Property	Portability
Vp1	Viewpoint	Layer diagram
Im1	Measure	Number of upward usages
Im2	Measure	Number of layer bridging
Im3	Measure	Cohesion
Which		
Sp1	Pattern	Layered style
Ta1	Tactic	Information hiding
Ta2	Tactic	Semantic coherence
Ta3	Tactic	Maintain existing interface
Ta4	Tactic	Abstract Common Services
Ta5	Tactic	Use Encapsulation
Ta6	Tactic	Use an Intermediary
Ta7	Tactic	Restrict Communication Paths
Sf1	Fragment	Interface for the layer
Sf2	Fragment	Upper layer
Sf3	Fragment	Lower layer
St1	Structure	Upper virtual machine
St2	Structure	Lower virtual machine
Sv1	View	Layered view
Ro1	Role	User of a layer
Where		
Ap1	Pattern	Work assignment

3.4.2 SSM of the modifiability tactics

Table 3.15 presents the SSM of the modifiability tactics described in [Bass03].

Table 3.15 The SSM of the modifiability tactics

Software Design Artifacts		
Id	**Type**	**Description**
Why		
Go1	Goal	Control the time to implement, test, and deploy changes
Go2		Control the cost to implement, test, and deploy changes
Dc1	Concern	Localize changes
Dc2		Prevention of ripple effect
Dc3		Defer binding time
Ra1	Rationale	Ensure that anticipated changes in a module are semantically coherent
Ra2		Assign responsibilities in a module that have semantic coherence
Ra3		Ensure that responsibilities work together without excessive reliance on other modules
Ra4		Reduce the number of modules directly affected by a change
Ra5		Restrict changes to a small set of modules
Ra6		Limit anticipated changes in scope
Ra7		Provide common services through specialized modules
Ra8		Restrict changes to a small set of modules
Ra9		Assign responsibilities in order to minimize the effects of the changes
Ra10		Allow a module to compute a broader range of functions based on input
Ra11		Define an input language for a module
Ra12		Ensure that changes can be made by adjusting the input language
Ra13		Restrict possible options in order to minimize the effects of the changes
Ra14		Reduce the necessity of making changes to modules not directly affected by a modification
Ra15		Assign responsibilities for an entity into smaller pieces
Ra16		Make some information private, and other information public
Ra17		Make public responsibilities available through specified interface
Ra18		Separate the interface from the implementation
Ra19		Create public abstract interface that mask variations
Ra20		Embody variations within the existing responsibilities
Ra21		Embody variations by replacing one implementation of a module with another
Ra22		Restrict the modules with which a given module shares data

Ra23		Insert an intermediary that manages activities associated with a dependency
Ra24		Convert the data syntax produced by a module into that assumed by another
Ra25		Convert the syntax of a service from one form into another
Ra26		Mask changes in the identity of an interface
Ra27		Enable the location of a module to change without affecting another module
Ra28		Guarantee the satisfaction of all requests within certain constraints
Ra29		Create instances as needed by actions of an intermediary
Ra30		Support plug-and-play operation
Ra31		Do registration at runtime
Ra32		Do registration at loadtime
Ra33		Set parameters at startup
Ra34		Allow late binding of method calls
Ra35		Allow loadtime binding
Ra36		Allow runtime binding of independent processes
When		
Sr1	Risk	Difficult to mask changes to the meaning of data and services
Sr2		Difficult to mask dependencies on quality of data or quality of services
Sr3		Difficult to mask dependencies on resource usage and resource ownership
Sr4		An intermediary cannot compensate for semantic changes
Sr5		Additional overhead to manage the registration
Sr6		Additional overhead to manage the initialization
Sr7		Additional overhead to manage the late binding
Sr8		Additional overhead to manage the loadtime binding
What		
Pr1	Property	Modifiability
Pr2		Reusability
Oc1	Opera.	Declare abstract signature
Im1	Measure	Coupling
Im2		Cohesion
Im3		Number of modules that require changing to implement a change
Im4		Number of modules directly affected by a change
Im5		Number of modules that consume data produced by the given module
Im6		Number of modules that produce data consumed by the given module
Which		
Ta1	Tactic	Maintain semantic coherence
Ta2		Abstract common services
Ta3		Anticipate expected changes

Ta4		Generalize the module
Ta5		Limit possible options
Ta6		Hide information
Ta7		Maintain existing interface
Ta8		Restrict communication paths
Ta9		Use an intermediary
Ta10		Runtime registration
Ta11		Configuration files
Ta12		Polymorphism
Ta13		Component replacement
Ta14		Adherence to defined protocols
Sf1	Fragment	Application framework
Sf2		Middleware software
Sf3		Interpreter
Sf4		Blackboard repository
Sf5		Passive repository
Sf6		Broker
Sf7		Name server
Sf8		Façade
Sf9		Bridge
Sf10		Mediator
Sf11		Strategy
Sf12		Proxy
Sf13		Factory
Sf14		Resource manager
Sf15		XML configuration file
St1	Structure	Module of constants input parameters
St2		Public interface
St3		Module that consumes data
St4		Module that produces data

3.4.3 An analysis of the results of the case study

This case study reinforced evidence regarding the usefulness of the proposed classification scheme (CS), techniques, and work instructions for populating DK bases of SDAs and SSMs. A large number of SDAs were classified using all cells of the CS. No SDA was rejected. The capability of the CS to classify all the SDAs extracted from various catalogs of DK provided evidence that the CS is reliable. The SDA types were used for discerning the semantic of each SDA. Most of the SDAs were easy to classify.

Table 3.16 presents the distribution of the SDAs into the sections of the style's template used in [Clem03].

Table 3.16 Style description: sections and SDAs

Sections of the style description	**SDA**
Overview	Rationale
Properties	Property
Elements	Operationalization
	Behavior
Relations, Topology	Structure
Implementation	Convention
Consequences	Procedure
Consequences	Role
Applicability	Situational factor

Table 3.17 summarizes the classification counts for SDAs related to tactics and styles, in which cells of the classification scheme are classified the SDAs used for describing the Layered System style in [Clem03].

Table 3.17 Classification counts for the SDAs of the Layered System style

	Why	When	What	Which	How	Where	
Objectives		2		1			**3**
Knowledge	4		2	8		1	**15**
Fragment	2	3	2	3			**10**
Structure			2	2			**4**
Description	2	1	1	1			**5**
Evaluation			3				**3**
	8	**6**	**10**	**15**	**0**	**1**	**40**

Table 3.18 summarizes in which cells of the classification scheme are classified the SDAs used in [Bass03] for describing the modifiability tactics.

Table 3.18 Classification counts for the SDAs of the modifiability tactics

	Why	**When**	**What**	**Which**	
Objectives	2				**2**
Knowledge	3	8	2	14	**27**
Fragment	36			15	**51**
Structure			1	4	**5**
Description					**0**
Evaluation			6		**6**
	41	**8**	**9**	**33**	**91**

The counts in Table 3.17 and Table 3.18 indicate that the descriptions of the tactics and style provide a higher number of SDAs related to software fragments than the number of SDAs related to software structure. The counts are consistent with the claim that design patterns describe more detailed designs than tactics and styles that describe architectural designs. The column “Which” in Table 3.17 also provides the highest count, which is consistent with the categorization of the Layered style as a structural style proposed by Clement and *al.* in [Clem03].

3.5 Case study: the SAM framework for designing a web site

This section presents the case study selected for applying the SAM framework to the development of a web site. The following sections present:

- the context of the case study (Section 3.5.1),
- the decision points considered for the case study (Section 3.5.2),
- the SSMs produced for developing the web site (Section 3.5.3) ,
- the analysis of the case study (Section 3.5.4), and
- the limitations of the case study (Section 3.5.5).

3.5.1 Context of the case study

This case study was conducted in the context of the development of a web site for a small organization that reproduces framed diplomas on metal. The project deals with constraints on the budget and the following requirements for the web site:

- The web site is always available;
- The web site is available on all platforms;
- The web site is available in French and English;
- Five web pages compose the site: Entry, Home, Enterprise, Products, and Contacts;
- The web pages shall be valid according to the strict syntax of HTML;
- The same presentation (e.g., font and background) is used for all web pages;
- It is possible to modify the presentation (e.g., font and background) of a web page;
- It is possible to send an email to the company using a form on the web page;
- It is possible to navigate forward and backward between the descriptions of the products.

3.5.2 Decision points considered for the case study

Table 3.19 presents the decision points that were identified during the literature review and used for triggering the activities proposed by the SAM process during this case study.

Table 3.19 The decision points used for triggering the activities of the SAM process

Decision point	**Triggered activity**	**Output**
Apply a style	Create a SSM	SSM of the style
	Create a SSM	SSM of the context of application of the style
	Describe arguments	Arguments related to the context of application
Apply a pattern	Create a SSM	SSM of the pattern
	Create a SSM	SSM of the context of application of the pattern
	Describe arguments	Arguments related to the context of application
Apply a tactic	Create a SSM	SSM of the tactic
	Create a SSM	SSM of the context of application of the tactic
	Describe arguments	Arguments related to the context of application
Define a fragment	Create a SSM	SSM of the context of definition
	Describe arguments	Arguments related to the context of definition
Specify a structure	Create a SSM	SSM of the context of specification
	Describe arguments	Arguments related to the context of specification
Select a technology	Create a SSM	SSM of the context of selection
	Describe arguments	Arguments related to the context of selection

3.5.3 SSMs created for developing the web site

The following SSMs and arguments were created during the execution of the first iteration of the web site development project. For this case study, some SDAs were described using two versions of the same SSM to illustrate that creating a SSM is an iterative activity. The name of the SSM (e.g., SSM – Client/Server) is the name of the SDA that provides cohesiveness for the SSM. In particular the decision of using the Client-Server style is a prerequisite. A web site is hosted on a web server. The web navigator sends requests for web pages to the web server. The following SSMs are two versions of the same SSM. During the creation of the first SSM, the focus was on identifying this fundamental style.

SSM – Client-Server

Decision point: Apply a style
Property: Scalability
Style: Client-Server

During the review of the SSM, the focus was on identifying explicitly the SDAs that relate to the utilization of the style.

SSM – Client-Server

Decision point: Apply a style
Property: Scalability
Style: Client-Server
Fragment: Web server
Fragment: Web navigator
Fragment: Web site

The following SSMs are two versions of the same SSM. During the creation of the first SSM, the focus was on identifying a pattern that will support portability.

SSM – Use a light client

Decision point: Apply a pattern
Property: Portability
Pattern: Use a light client

During the review of the SSM, the focus was on identifying the requirement and SDAs that relate to the utilization of the pattern. In particular, the web navigator will be a light client.

SSM – Use a light client

Decision point: Apply a pattern
Requirement: The web site is available on all platforms
Property: Portability
Style: Client-Server
Fragment: Web navigator
Pattern: Use a light client

The following SSMs are two versions of the same SSM. During the creation of the first version, the focus was on identifying the business model related to the web server.

SSM – Web server

Decision point: Select a technology
Requirement: The web site is always available
Property: Availability
Constraint: Limited budget
Business model: External contract
Structure of objects: External organisation
Fragment: Web server

During the review of the SSM, the focus was on identifying the SDAs that relate to the web server and describing an argument that supports the choice of an external organization for hosting the web server.

SSM – Web server

Decision point: Select a technology Requirement: The web site is always available Property: Availability Constraint: Limited budget Business model: External contract Structure of objects: External organisation Structure of objects: Web site owner Fragment: Web site Fragment: Web server Risk: Availability issues Issue: the availability property is difficult to ensure Reasoning: an external organisation will manage the web server to ensure the availability of the web site

The following SSMs are two versions of the same SSM. During the creation of the first version, the focus was on identifying the risk and the assumption that makes it acceptable.

SSM – Security

Decision point: Apply a tactic Tactic: Adherence to defined protocols Protocol: HTTP Requirement: It is possible to send emails to the company using a form on the web page Properties: Security Risk: Personal information usurpation Assumption: The user is responsible for not sharing personal information

During the description of the arguments, the risk and the assumption led to the identification of the following structure of objects, domain object, and argument.

SSM – Security

Decision point: Apply a tactic
Tactic: Adherence to defined protocols
Protocol: HTTP
Requirement: It is possible to send emails to the company using a form on the web page
Properties: Security
Domain object: User
Structure of objects: Web site owner
Risk: Personal information usurpation
Assumption: The user is responsible for not sharing personal information
Assumption: The web site owner is responsible for not requesting personal information

Issue: HTTP is a protocol with insecure exchanges
Reasoning: The HTTP protocol makes the user's information accessible
Dimension: Quality, People

The following SSMs are also two versions of the same SSM. During the creation of the first version, the focus was on identifying the risk of presentation problems.

SSM – Limited budget

Decision point: Select a technology
Fragment: Web navigator
Fragment: Operating platform
Requirement: The same presentation (e.g., font) is used for all web pages
Constraint: Limited budget
Process: Validating presentation
Risk: Presentation problem

Issue: The presentation is difficult to validate

During the review of the SSM, the focus was on identifying the SDAs that relate to the process of validating presentations and describing an argument that supports the choice of an external organization for hosting the web server.

SSM – Limited budget

Decision point: Select a technology
Fragment: Explorer web navigator
Fragment: Safari web navigator
Fragment: Windows OS
Fragment: Apple IOS
Requirement: The same presentation (e.g., font) is used for all web pages
Constraint: Limited budget
Process: Validating presentation
Risk: Presentation problem
Issue: The requirement will be difficult to meet
Reasoning: The presentation is not managed identically on all platforms
Reasoning: The limited budget makes it impossible to validate a presentation on all platforms

The following SSMs were also created during the first iteration of the SAM process.

SSM – Portability

Decision point: Select a technology
Properties: Portability
Fragment: Web navigator
Language: HTML
Language: Java Script
Language: CSS
Assumption: The language is supported by all web navigators

SSM – Responsiveness

Decision point: Apply a tactic
Requirement: It is possible to send emails to the company using a form on the web page
Properties: Responsiveness
Behavior: The web client process validates the entries of the user form
Language: Java Script

SSM – HTTP

Decision point: Apply a tactic
Properties: Interoperability
Tactic: Adherence to defined protocols
Protocol: HTTP

SSM – Anticipate expected changes

Decision point: Apply a tactic
Requirement: It is possible to modify the presentation (e.g., font) of a web page
Design concern: Localize changes
Properties: Modifiability
Tactic: Anticipate expected changes
Issue: The requirement is not specific enough

SSM – Configuration files

Decision point: Apply a tactic
Requirement: It is possible to modify the presentation (e.g., font) of a web page
Design concern: Localize changes
Properties: Modifiability
Tactic: Configuration files
Language: CSS
Assumption: The configuration files will support modifying the presentation of a web page

SSM – Reusability

Decision point: Apply a tactic
Requirement: The same presentation (e.g., font) is used for all web pages
Design concern: Localize changes
Properties: Reusability
Tactic: Configuration files
Language: CSS
Assumption: The configuration files will provide the same presentation for all web pages

Many SSMs including the following SSM were created during the second iteration of the SAM process.

SSM – Syntax validator

Decision point: Select a technology
Requirement: The web pages shall be valid according to the strict syntax of HTML 4.01
Constraint: Limited budget
Property: Validity
Property: Portability
Task: Validate a web page
Fragment: Web page
Structure of objects: W3C
Fragment: Syntax validator
Issue: Validating a web page is not difficult
Reasoning: The syntax validator will ensure the validity of the web pages

3.5.4 Analysis of the case study

This case study reinforced evidence regarding the usefulness of the proposed classification scheme (CS) and descriptions formats for populating DK bases of SDAs and SSMs. Many SDAs were classified using the SDAs types. No SDA was rejected. The capability of the CS for classifying the SDAs used in a web engineering context provided evidence that the CS is reliable.

This case study also reinforced evidence that the SAM framework has some of the issues presented in Table 1.7 for the DK management. Table 3.20 describes the issues for the SAM framework in the context of this case study.

Table 3.20 Issues for the SAM framework

Issue for the DK management	**Brief description**
• Need for tailored forms of design knowledge	• Need for tailored SSMs and decisions types
• Complex relationships of knowledge item	• Relationships for SDAs, SSMs, arguments, and decisions
• Design knowledge management overhead	• Overhead for using the issue description format
• Lack of measurable indicators	• Lack of comparable approaches and results
• Lack of relevance and usability	• Arguments and views were not useful for this case study
• Inadequate tool support	• For managing SDAs, SSMs, arguments, and views
• Lack of scientific rigor	• Steps of the SAM process may be intertwined or skipped

In particular, a small number of issues and arguments were described. The requirements were selected in order to limit the number of issues related to the design of the web site. In addition, some arguments do not describe the impacts of issues on the activities and dimensions. The requirements were also selected to limit these impacts. Therefore, the issues and reasoning descriptions were sufficient to describe the arguments. There is a management overhead for describing the issues using the proposed description format. The SDA, verb, and complement require the user to describe the design problem issues in a formatted manner. The format is simple but not intuitive for some issues. For this case study, views were not useful. The overhead for producing views without tool-support is significant.

3.5.5 Limitations of the case study

The generalization of results is limited due to the fact that:

- only two participant participated in the case study,
- a limited schedule and budget were used to conduct the case study,
- only a small design problem context was used, and
- multiple interpretations of the requirements were possible.

3.6 Experiment for evaluating the SAM framework with a human participant

This section describes the experiment that was conducted with a human participant for evaluating the classification technique and the argumentation technique of the SAM framework. The following sub-sections present:

- the context of the experiment (section 3.6.1),
- the experiment planning (Section 3.6.2);
- the experiment process and schedule (Section 3.6.3);
- the experiment subjects (Section 3.6.4);
- the participant profile (Section 3.6.5);
- the design context and data collection (Section 3.6.6);
- the SSMs created by the participant (Section 3.6.7 and Section 3.6.8); and
- the limitations of the experiment (Section 3.6.9).

3.6.1 Context of the experiment

The experiment was conducted in a research laboratory with no budget or constraints on the schedule. The experiment was conducted by a human participant required to apply the SAM framework in order to produce:

Part 1 – the SSM for the Template Method design pattern, and
Part 2 – the SSMs, issues, and arguments for designing a web site.

3.6.2 Experiment planning

The objects of the experiment were the tasks of the SAM process, the classification scheme, and the descriptions formats for the SSMs, issues, and arguments of the SAM framework.

The participant was required to provide SSMs, issues, arguments, and feedback about his utilization of the SAM framework. The goal of the experiment was the evaluation of the data collected from the participant. The following characteristics were examined:

- Reliability (repeatable outputs);
- Efficiency (time, effort, cost, results);
- Usability (required background, relevance of work instructions), and
- Accuracy (validity of the evaluation)

3.6.3 Experiment process and schedule

The process of three activities (i.e., Preparation, Execution, and Evaluation) planned for the experiment was similar to the process presented in Figure 3.1.

The preparation activity was planned for presenting the experiment process and the related descriptions, analysis model, and forms to the participant in a meeting. A two hour period was scheduled for the presentation.

The execution activity was conducted by the participant. Two hours were scheduled for the analysis of the TM design pattern in a workshop. For the analysis of the web engineering problem, three workshops of two hours were scheduled.

The evaluation activity was planned to analyze the participant's forms and to report on the experiment.

In particular, the following activities were executed during the experiment:

- a presentation to the participant of :
    - the reference model (SDA, SSM, and argument);
    - the classification and argumentation techniques;
    - the classification scheme and description formats;
    - examples of SDAs, SSMs, issues, and arguments;
    - the experiment and work instructions;
    - the SDA types to be examined;
    - the activities of the DK management to be examined;
    - the TM design pattern to be examined;
    - the web engineering problem to be examined.
- the execution of the analysis by the participant to:
    - create the SSM of the TM design pattern
    - create SSMs, elicit issues, and describe arguments for the web site.

3.6.4 Experiment subject

One person participated in the experiment. The participant was a graduate student with a masters degree in software engineering at ETS. There was no risk to the participant involved in this experiment. No identifying information from the participant was collected. The participant voluntarily participated in the experiment by attending the training and the experiment workshops.

3.6.5 Participant profile

The participant was selected according to the following profile:

- graduate student in software engineering;
- more than five years of experience in designing software; and
- background: web development, styles, tactics, design patterns.

3.6.6 Design context and data collection

The experiment was planned for data collection by the participant in a workshop. For the first part of the experiment, the SSM of the TM design pattern and qualitative feedback about the approach were collected. The second part of the experiment was conducted in the context of the web engineering problem proposed in section 3.5. The web site was required to be simple enough to be designed by the participant within the limited budget and time. For designing the web site, the participant was required to use the requirements presented in 3.5.1 and the decisions points presented in 3.5.2 for creating SSMs and describing arguments. The SSMs, issues, arguments, and qualitative feedback were collected from the participant.

3.6.7 Part 1 – SSM created by the participant for the TM design pattern

For the first part of the experiment, the participant was required to:

1) identify the SDAs from the description of the TM design pattern in [Gamm95],
2) classify the SDAs, and
3) infer the SSM of the TM pattern.

For the first iteration, the participant was required to identify SDAs from the pattern description but without being briefed about the SAM framework. The participant was asked to consider any conceptual artifact that provides design knowledge about the problem or solution spaces of a software design. The SDAs in Table 3.21were identified by the participant.

Table 3.21 SDAs identified by the participant without using the SAM framework

Avoid code duplication
Control subclasses extension
Fix the steps of the algorithm and their ordering
Let subclasses define the steps of the algorithm
Reusability
Abstract class definition
Concrete class definition
Template method definition
Primitive operation definition
The template method calls the primitive operation

The participant was briefed about the SAM framework for the second iteration. Then, the participant was required to create the SSM of the TM design pattern. The SSM in Table 3.22 was created by the participant.

Table 3.22 SSM created by the participant for the TM design pattern

Design pattern	Template method
Design concern	Avoid code duplication
	Control subclasses extension
Design rationale	Localize common behavior
	Implement the invariant parts of an algorithm once
	Factorize the steps of the algorithm and fix their ordering
	Let subclasses define the steps of the algorithm
	Provide default behavior
	Limit extension points
	Minimize primitive operations
Requirement	Specify hook operation
	Specify abstract operation
Convention	Naming convention
Property	Reusability
Operationalization	Define an abstract base class
	Define a template method
	Define a concrete child class
	Define hook operations
	Declare a final template method
	Declare protected primitive operations
	Declare abstract primitive operations
Structural pattern	Factory Method
Behavior	The template method calls the primitive operations

3.6.7.1 Analysis of Part 1

The catalog of patterns described in [Gamm95] was used by the participant for creating the SSM of the TM design pattern. The SDAs described in other catalogs including [Bass03] and [Clem02] were not considered in Part 1.

The SSM in Table 3.22 was compared to the SSM in Table 3.7 for the TM design pattern. The required SDAs were identified by the participant but not the SDAs in Table 3.23.

Table 3.23 SDAs that were not identified by the participant

SDA type	**SDA**
Design rationale	Control access to the operations
Situational factor	Multiple kinds of primitive operations
Role	Subclass writers
Structural fragment	C++ language
	Class library
Software structure	Abstract class definition
	Concrete class definition
	Template method definition
	Primitive operation declaration
	Primitive operation definition
	Hook operation definition
Software behavior	The hook operation does nothing by default
Allocation fragment	Class file

The SDAs classified as “operationalization” in the problem space usually have a one-to-one correspondence with the SDAs classified as “software structure” in the solution space. An operation contract describes an operationalization and an operation declaration is the software structure that implements the contract. This correspondence may cause ambiguity. Using the interrogatives (i.e., what and which) of the classification scheme and the inference heuristics reduces this ambiguity. Nonetheless, corresponding SDAs seem redundant.

The participant was required to report which parts of the SAM framework were used for classifying each SDA. Table 3.24 summarizes the feedback provided by the participant about his utilization of the SDA type descriptions, classification scheme, and inference heuristics for Part 1.

Table 3.24 Summary of utilization of the SAM framework for Part 1

SDA Type	Classification Scheme (Int.)	Classification Scheme (Act.)	Inference heuristics	SDA Type Description
Design pattern	X			X
Design concern	X	X	X	X
Design rationale	X	X	X	X
Requirement	X	X		X
Convention	X			X
Property	X			X
Operationalization				
Software behavior	X	X		X

The SDAs of types “Design pattern”, “Convention”, and “Property” were classified only by using the interrogative and the SDA type descriptions. To classify the SDAs of types “Design concern”, “Design rationale” and “Requirement”, the participant used every parts of the classification technique and reported the following remarks:

- The SDA types, the CS, and the inference heuristics refer to many concepts. The interrogatives and activities of the CS were used to reduce the significant burden of understanding the concepts and their relationships.

- The task “Identify verbs and nouns” was used to identify the SDAs. However, the pattern descriptions provide examples of applications for each pattern. It was hard to discern between the SDAs that relate to examples and SDAs that constitute the design pattern.

- The interrogatives and activities of the decision tree were used for identifying candidate SDAs types for each relevant verb and noun.

- The inference heuristics were used for classifying the SDA “Specify the hook operation” as a “Requirement” and for discerning that the requirement is addressed to the TM class writer, which has been classified as a “Role”.

- The inference heuristics and the concepts of software fragment and software structure were used for discerning the difference between a design concern and a design rationale.

3.6.7.2 Conclusions of Part 1

The SMM and feedback provided by the participant reinforced evidence regarding the reliability and usability of the proposed tasks of the classification technique, the CS, the SDAs types, the decision tree, and the inference heuristics. The following conclusions were reported for Part 1:

- Reliability - Many SDAs were classified using many SDA types ;
- Reliability - No knowledge item was rejected (i.e., all SDAs were classified) ;
- Usability - All expected SDAs were identified ;
- Usability - All SDAs were classified in the expected cases of the CS ;

- Usability - Some SDAs were classified and then rejected due to the lack of instructions for discerning the SDAs that relate to the pattern description and the SDAs that relate only to examples of applications of the pattern.
- Usability - Some SDAs were renamed due to the lack of naming instructions.

3.6.8 Part 2 – SSMs created for developing the web site

A design knowledge base of SDAs including patterns [Gran02, Gamm95], styles [Clem02], tactics [Bass03], design concerns, and properties were provided to the participant who described the following SSMs and issues during the workshops. APPENDIX VI describes some inputs and outputs for the workshops.

SSM – Client-Server

Decision point: Apply a style
Style: Client-Server
Fragment: Web navigator
Fragment: Web server

SSM – Three-tier architecture

Decision point: Apply a style
Property: Availability
Style: Three-tier architecture
Fragment: Web server
Fragment: Application server
Fragment: Database server

SSM – Interoperability

Decision point: Apply a tactic
Properties: Interoperability
Tactic: Adherence to defined protocols
Protocol: HTTP

SSM – Portability

Decision point: Select a technology
Properties: Portability
Requirement: The web site is available on all platforms
Language: HTML
Language: Java Script
Language: CSS
Issue: The requirement is not well defined
Reasoning: The platforms may be workstations, mobile devices, or software among others

SSM – Availability

Decision point: Apply a tactic
Properties: Availability
Requirement: The web site is always available
Tactic: Use a spare device
Tactic: Use a spare process
Allocation fragment: Web server host device
Behavioral fragment: Web server process
Behavioral fragment: Load balancer process
Behavior: The load balancer detects when the web server process does not respond
Behavior: The load balancer handles the initialisation of the web server process
Behavior: The load balancer handles the distribution of work
Issue: The requirement is not well defined
Reasoning: Modifying the web site may impact its availability for controlled periods of time

3.6.8.1 Analysis of Part 2

During the first workshop, the participant designed the web site by sketching UML diagrams to help visualize decisions and alternatives about the design. Then, the participant created the SSMs and described the related SDAs of types “Property”, “Design pattern”, “Style”, “Tactic”, and “Fragment”. Many SDAs were not identified and the decisions were not detailed at the end of the first workshop. In particular, the issues, the arguments, and the SDAs of types “Design concern”, “Design rationale”, and “Requirement” were not described.

At the beginning of the second and third workshops, the participant was required to review the SSMs for describing additional or invalid SDAs, issues, and arguments. For addressing the issues elicited during a workshop, some SDAs were added to or retrieved from the SSMs during subsequent workshops. At the end of the second workshop, the SSMs and decisions were sufficiently detailed to support an implementation of the web site. However, the decomposition of the SSMs and the cohesiveness between the related SDAs were somehow deficient. For example, the SSM “Availability” previously introduced may have been decomposed into the two following SSMs by separating the two tactics “Use a space device” and “Use a space process”.

SSM – Availability

Decision point: Apply a tactic
Properties: Availability
Requirement: The web site is always available
Tactic: Use a spare device
Allocation fragment: Web server host device

SSM – Availability

Decision point: Apply a tactic
Properties: Availability
Requirement: The web site is always available
Tactic: Use a spare process
Behavioral fragment: Web server process
Behavioral fragment: Load balancer process
Behavior: The load balancer detects when the web server process does not respond
Behavior: The load balancer handles the initialisation of the web server process
Behavior: The load balancer handles the distribution of work

During the third workshop, the participant was required to decompose coarse-grained SSMs and identify additional or invalid SDAs, issues, and arguments. The participant used the propositions of both the classification and argumentation techniques for Part 2 and reported the following remarks:

- The issue description format was useful for identifying and describing the issues. The proposed activities of the DK management were useful for identifying some issues.

- The argument description format (i.e., reasoning) was more intuitive and useful than the issue description format for thinking about the design problem. The scope of the arguments did not seem useful for documenting the decisions.

- The instructions for evaluating the SSMs (e.g., granularity and required SDAs types) were not explicitly detailed.

3.6.8.2 Conclusions of Part 2

The SMMs and feedback provided by the participant reinforced evidence regarding the reliability and usability of the proposed CS, the SDAs types, the SSM description format, and the issue description format.

The following conclusions were retained for Part 2:

- Reliability - Many SDAs were classified using many SDA types ;
- Reliability - No knowledge item was rejected (i.e., all SDAs were classified) ;
- Usability - ALL SDAs were classified in the expected cases of the CS ;
- Usability - Some SDAs were renamed due to the lack of naming instructions ;
- Usability - Some SDAs of particular types were not identified ;
- Usability - The SSMs were less intuitive than UML diagrams for designing ;
- Usability - The SSMs were more effective than UML diagrams for documenting ;
- Usability - Many SSMs were incomplete at the end of the first workshop ;
- Usability - Some SSMs were coarse-grained at the end of the second workshop ;
- Usability - SDAs, issues, and arguments were missing at the end of the workshops.

3.6.9 Limitations of the experiment

The generalization of results is limited due to the fact that:

- only two participants participated in the experiment,
- only a few hours were used to conduct the experiment,
- only a small design problem context was used, and
- multiple interpretations of the requirements were possible.

CHAPTER 4

A TECHNIQUE FOR CREATING A SOFTWARE STRUCTURES MAP

This chapter presents the classification technique developed in the first activity of Phase 3 of our research methodology in Figure 1.3. This chapter proposes a technique for creating a software structures map (i.e., the first activity of the SAM process "Create a SSM" – see Figure 2.2). It presents a classification scheme (CS) that organizes SDAs into a matrix, in a manner derived from the Zachman Framework [Zach11] for enterprise architecture. An instantiation of this CS is a traceability matrix called a software structures map (SSM) that records the SDAs and their relationships. The approach is illustrated through the analysis of the Template Method (TM) design pattern as an example of a SDA.

This chapter is organized as follows. Section 4.1 presents an overview of the proposed classification technique. Section 4.2 to Section 4.8 describe the six tasks of the technique and the propositions for supporting the SAM framework, including the identification heuristics, classification scheme, decision tree, SDA description, relationship description format, inference heuristics, and software structures map. Section 4.9 presents the conclusions, contributions, and future work of this chapter.

4.1 The proposed classification technique

Figure 4.1 presents the proposed classification technique which aims at creating a SSM by extracting the verbs and nouns for structuring the SDAs and relationships that constitute the description of a style, a design pattern, or a tactic. This figure presents the task flow for the six tasks of the proposed classification technique and the data flow for the inputs and outputs of each task. In particular, the SAM framework proposes four inputs to support the classification technique: identification heuristics, decision tree, classification scheme, and inference heuristics.

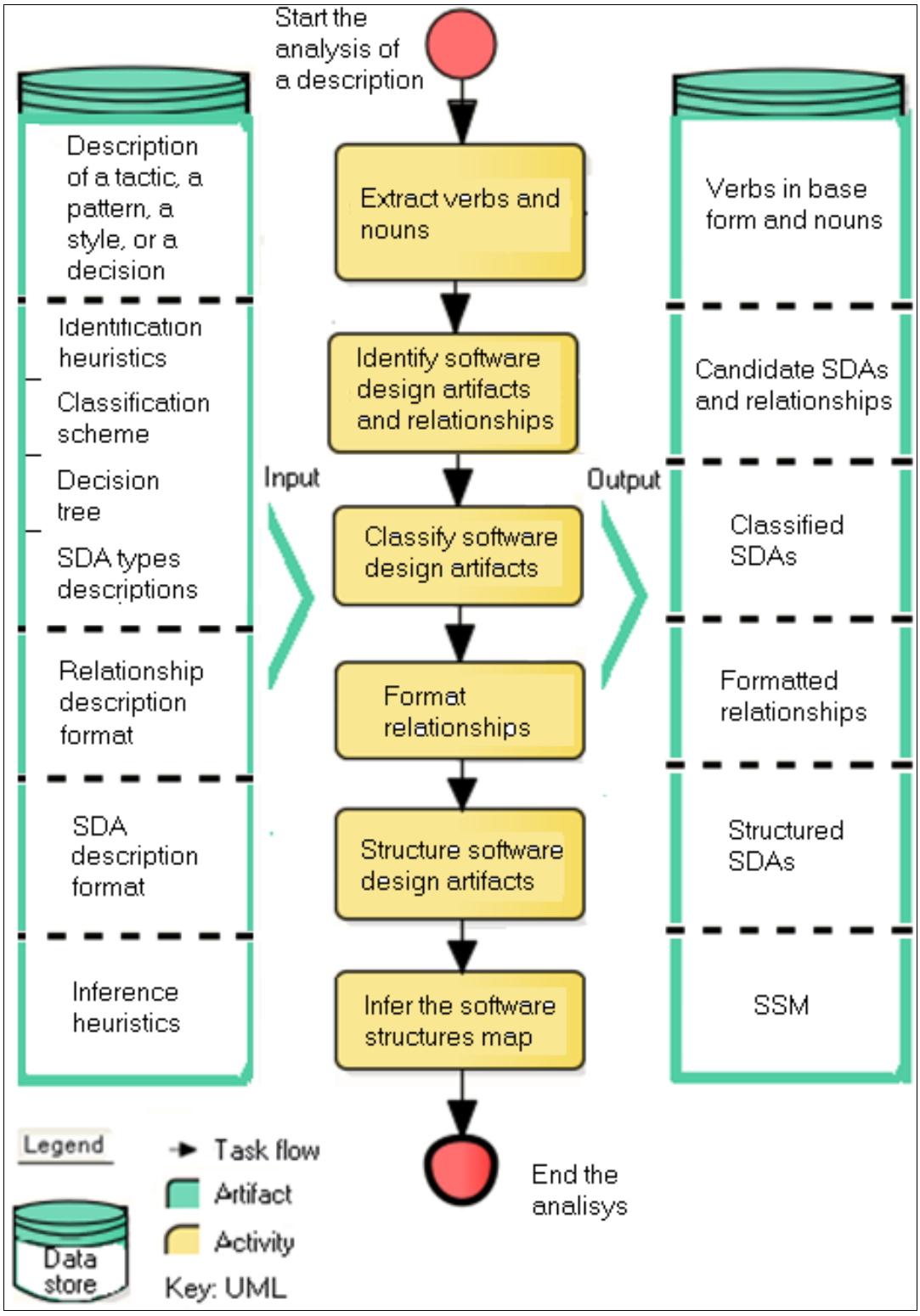


Figure 4.1 The proposed classification technique of the SAM framework

4.2 The tasks of the classification technique

Six tasks constitute the proposed classification technique:

1. extract verbs and nouns,
2. identify SDAs and relationships,
3. classify the SDAs,
4. format the relationships,
5. relate the SDAs, and
6. infer the SSM.

Tasks 1 and 2 aim at identifying candidate SDAs and relationships from the description of a decision, a style, a pattern, or a tactic using the identification heuristics. Tasks 3 and 4 aim at classifying the SDAs using the CS, the decision tree, and the SDAs descriptions, and formatting the relationships using the relationship format. Tasks 5 and 6 aim at structuring the SDAs and inferring the SSM by using the relationships and inference heuristics.

4.3 Task 1 – Extract verbs and nouns

Task 1 of the classification technique aims at extracting the verbs and nouns from the descriptions of design patterns, tactics, or styles. Table 4.1 presents the verbs and nouns extracted from the following description of the "Exception Detection" tactic. The verbs and nouns are selected to be classified. Expressions that certainly do not describe knowledge are removed. Verbs should be extracted in their basic form, which means that the verbs are not conjugated (i.e., infinitive verbs without the "to").

> "Exception Detection refers to the detection of a system condition that alters the normal flow of execution. For distributed real-time embedded systems, the Exception Detection tactic can be further refined to include System Exceptions, Parameter Fence, and Parameter Typing tactics. System Exceptions will vary according to the processor hardware architecture employed and include faults such as divide by zero, bus and address faults, illegal program instructions, and so forth. The Parameter Fence tactic incorporates an a priori data pattern (such as 0xDEADBEEF) placed after any variable-length parameters of an object."

Table 4.1 Verbs and nouns that describe the "Exception Detection" tactic in [Scot09]

Verb expression	**Noun expression**
Refer	Exception detection tactic
	Detection of system condition
Alter	Normal flow of execution
Refine	Distributed real-time embedded systems
	System Exceptions tactic
	Parameter Fence tactic
	Parameter Typing tactic
Vary	Processor hardware architecture
Include	Divide by zero fault
	Bus fault
	Address fault
	Illegal program instructions

4.4 Task 2 – Identify SDAs and relationships

Task 2 of the classification technique aims at verifying the verbs and nouns for identifying candidate SDAs and relationships. The nouns are usually the objects (i.e., SDAs) in the sentence, and the verbs are some actions (i.e., SDAs) or relations between the objects (i.e., relationships). For guiding the identification of the SDAs, the SAM framework proposes that a SDA provide knowledge related to a design, using the following identification heuristics. The proposed heuristics are adapted from the heuristics used in [Bour02] for identifying the fundamental principles of software engineering.

1. less specific than software implementation, i.e. implementation may be selected, within a particular technological context, to accomplish the intent of an SDA;
2. more enduring than software implementation, i.e. an SDA should be described in a way that allows multiple implementations;
3. typically discovered or abstracted from practice and should have some correspondence with best practices such as styles, design patterns, and tactics;
4. coherent with more general or specific artifacts;
5. precise enough to be capable of analysis;
6. related to one or more SDAs.

4.5 Task 3 – Classify the SDA

Task 3 uses the classification scheme (CS) for structuring the SDA, and the decision tree and SDA descriptions for guiding its classification into a cell of the SSM. The CS of the SAM framework is adapted from the CS of the Zachman Framework (ZF) for enterprise architecture [Zach11].

4.5.1 The Zachman Framework for Enterprise Architecture

The traceability of the artifacts that result from the design decisions is problematic for the software architecture as it is for the enterprise architecture. The Zachman Framework (ZF) for enterprise architecture [Zach11] proposes a classification scheme for that problem. The ZF classifies the artifacts related to the enterprise architecture into a two dimensional matrix. Six interrogatives (What, Where, When, Why, Who and How) label the columns of the matrix, and six levels of perspective label the rows for transforming more abstract ideas (upper row) into more concrete ideas (lower row). The ZF is indeed a taxonomy that organizes the artifacts of the enterprise architecture (EA) into multiple perspectives.

The ZF “is simply a logical structure for classifying and organizing the descriptive representations of an Enterprise that are significant to the management of the Enterprise, as well as to the development of the Enterprise's systems”. Zachman's vision is that a holistic approach to EA that explicitly addresses every relevant issue from every relevant perspective should best accomplish business value and agility. The enterprise is viewed as an organizational system. The EA provides the blueprint for realizing this organizational system; it organizes the business processes, technologies, and information systems of the enterprise. To manage the complexity of the EA, the ZF organizes its structures and behaviors, principles, policies, and standards as a collection of perspectives represented in a two-dimensional matrix.

The ZF does not define a methodology or any specific technique for managing the artifacts. The matrix is a template that structures the artifacts of the EA, such as goals, rules, processes, material, roles, locations, and events. The ZF classifies and organizes the descriptive representations of an EA. The level of detail in the ZF is a function of each cell that describes one perspective of the EA. Each cell refers to a model (e.g., a list, a table, or a diagram) that addresses specific concerns and stakeholders. Zachman affirms that the ZF “yields the total set of descriptive representations relevant for describing an enterprise” [Zach11]. This classification scheme has not yet been adapted for the software architecture.

4.5.2 The proposed classification scheme (CS)

Table 4.2 presents the proposed classification scheme (CS) and Figure 4.2 presents the four perspectives of the CS:

- organizational space;
- design space;
- problem space; and
- solution space.

The CS organizes the SDAs extracted from the analysis of the descriptions of styles, design patterns, and tactics, and quality models and standards. The CS captures the SDAs about the design problem and solution spaces, and about explicit or implicit relationships between the SDAs. The CS captures the SDAs that influence the life cycle of a system.

The CS organizes the SDAs into a matrix based on the Zachman Framework for enterprise architecture [Zack11]. The matrix classifies the SDAs according to their descriptions and relationships, as described in [Bass03, Clem03, Iso42010, Iso9126, Apri11, Leff08]. More specifically:

- the rows represent the activities of the software design process, and
- the columns represent the interrogatives (why, when, what, which, how, and where).

The outcomes of the following activities occupy the row labels: select the objectives, identify the knowledge that has been successful in achieving similar objectives, and define, specify, describe, and evaluate the software architecture.

The problem space is split into the interrogatives why, when, and what.

- The rationale (WHY issues) provides reasoning about the problem.
- The context (WHEN issues) describes the contextual influences on the solution.
- The drivers (WHAT issues) define the problem.

The solution space is split into the interrogatives: which, how, and where.

- The structures of domain objects and design elements have roles (WHICH issues) in realizing the solution. Usually, they have:
    - to execute designed behaviors (HOW issues), and
    - assigned locations (WHERE issues).

The SDAs in the top row of Table 4.2 define the problems and solutions from an organizational perspective. The ones in the five lower rows do the same from a design perspective. Each lower-row contains artifacts for refining the interrogatives of the row that is above it, from the general objectives to the specific system artifacts.

Table 4.2 The proposed classification scheme of the SAM framework

	Rationale (Why)	Context (When)	Driver (What)	Structure (Which)	Behavior (How)	Allocation (Where)
Select objectives	needs, expectations, goals	organizational risks, politics, business model, situational factors	requirements, constraints, business rules	structures of domain objects	processes, activities, tasks, procedures	allocation of domain objects
Identify knowledge artifacts	architectural concerns	application domain standards, regulations, conventions	architectural properties	patterns and tactics	patterns of interactions	patterns of allocation
Define architectural artifacts	architectural design rationale	architectural risks, assumptions	scenarios	structural fragments	behavioral fragments	allocation fragments
Specify system artifacts	detailed design rationale	system's risks, assumptions	operation contracts	structures of modules	behaviors of components and connectors	allocations of elements
Describe architectural views	views descriptions	external entities, scopes, vocabularies, symbols	viewpoints	structural views	behavioral views	allocation views
Evaluate software structures	acceptance / assurance criteria	external and in-use metrics	internal metrics	structural evaluation records	behavioral evaluation records	physical evaluation records

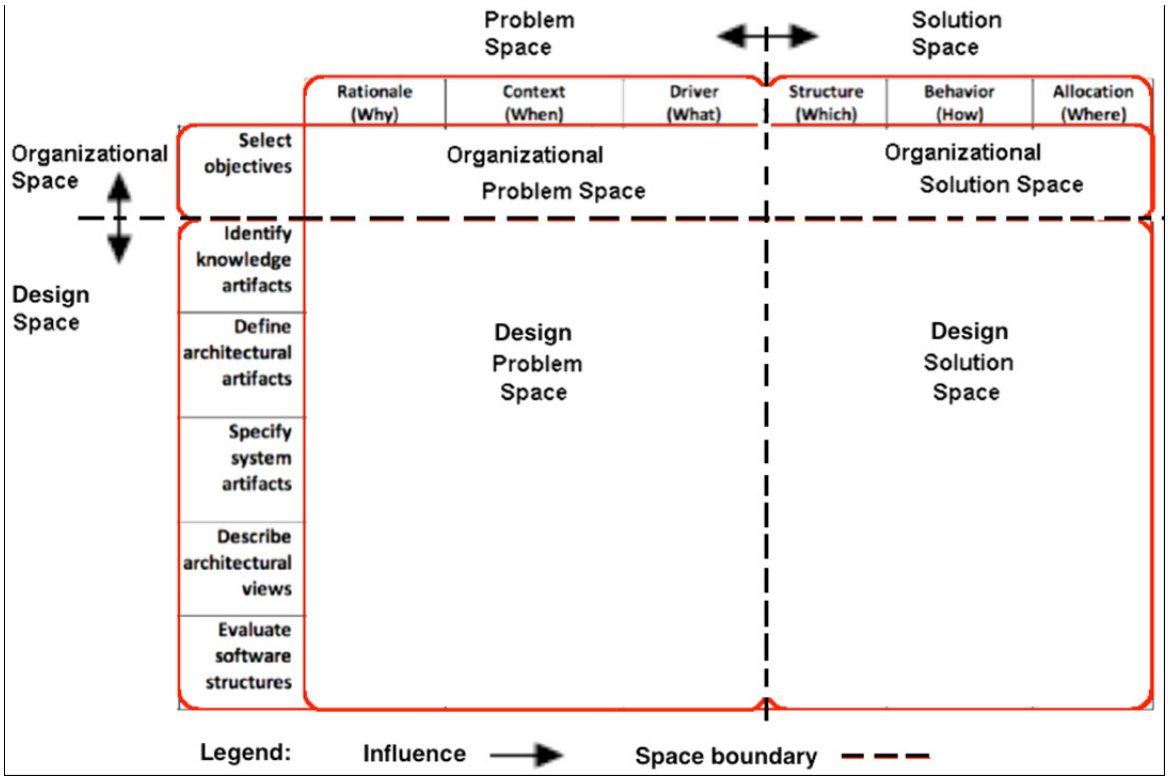


Figure 4.2 Perspectives of the CS: organizational, design, problem, and solution

Each activity that labels a row in the CS is regarded both from the perspective of the problem space and the solution space. In addition, the SDAs of both procedural and technical solutions are organized into the CS. The procedural solutions define the processes, activities, and tasks that the user of a software product should realize to produce the outcomes needed by the stakeholders. The technical solutions provide the artifacts that the user employs to achieve his objectives. The procedural and technical solutions are intertwined. The procedural solutions describe how the user should employ the technical solutions to attain his objectives. The technical solutions support, limit, and constrain how the user can use a software-intensive system.

4.5.3 The proposed decision tree

The decision tree in Figure 4.3 is used for classifying the SDAs. The question form (as proposed in [Zimm12]) is used for supporting the classification task. Software designers will use the following questions in sequence for classifying the design knowledge item being examined in a column (interrogative), a space (organizational or design), and a row (activity) of the CS. Then, they will select an artifact from the targeted cell. The questions begin with the prefix "Does the SDA describe". Each question relates to one of the four main questions presented in the decision tree: which interrogative, space, activity, and artifact best render the meaning of the SDA in the context of a SSM?

1. Which interrogative?
   - why: "… a reasoning for the SSM?"
   - when: "…a contextual information for the SSM?"
   - what: "… a target for a solution?"
   - which: "… the element of a solution?"
   - how: "… the behavior of an element?"
   - where: "… the allocation of an element?"

2. Which space?

   - organizational: "… the organizational space? "

   - design: "… the design space? "

3. Which activity? (only SDAs classified into the design space)

   - reusing knowledge: "… an information that is part of the design knowledge base?"

   - architecting software: "… an information about a design fragment?"

   - designing software: "… an information about a design structure?"

4. Which artifact?

   - use the SDA descriptions

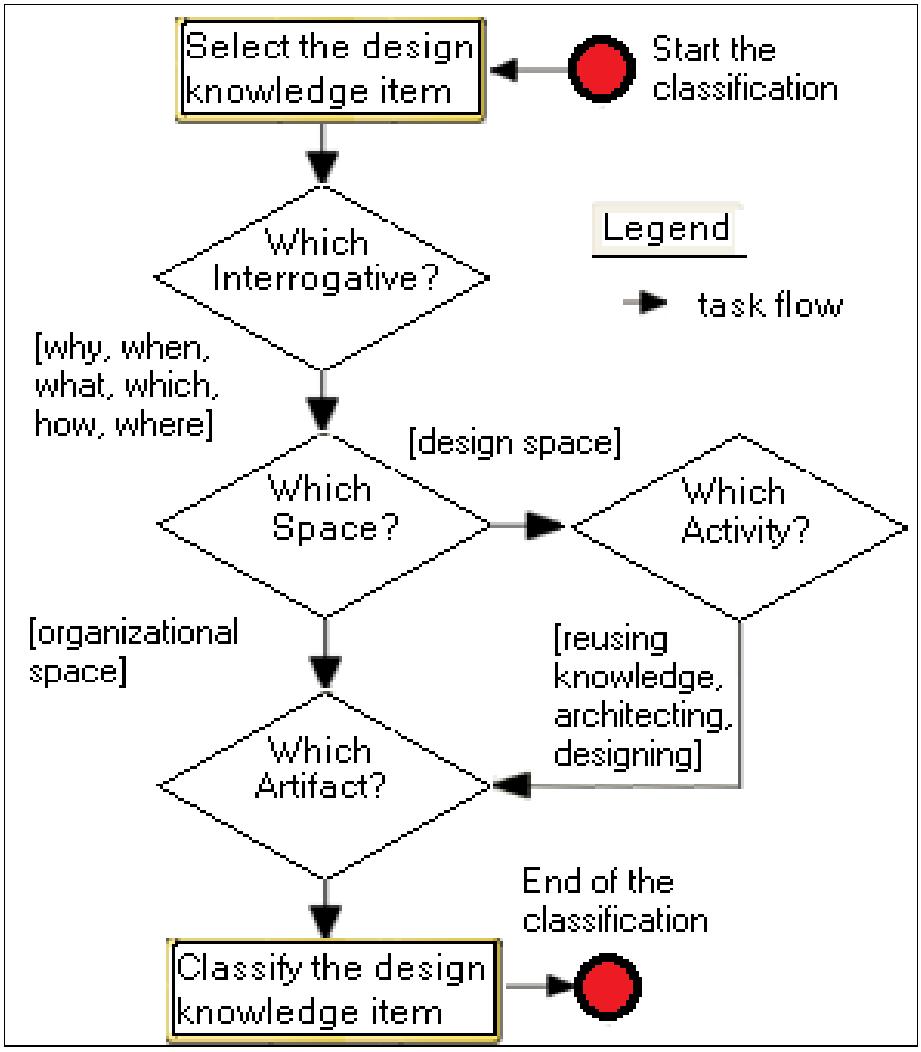


Figure 4.3 The decision tree for classifying a design knowledge item

4.5.4 The proposed SDAs descriptions

To classify an artifact, Table 4.3 to Table 4.8 describe the SDAs proposed in the literature. This section describes the SDAs that relate to the top four rows of the CS.

Table 4.3 The descriptions of some SDAs related to the Why interrogative

Why: These SDAs provide reasoning for the SSM
Need: a statement of what is necessary for a system to be suitable (e.g., lift up developers' productivity) **Goal:** a desired outcome of user interaction with a software product (e.g., control the time to implement a software component)
Architectural concern: an area of interest specified with respect to a goal in terms relevant for architecting (e.g., define the scope of the software development kit) **Design concern:** an area of interest specified with respect to a goal in terms relevant for designing (e.g., avoid code duplication)
Architectural rationale: a statement of reasons for a design fragment (e.g., isolate each layer from changes in other layers)
Design rationale: a statement of reasons for a software structure (e.g., define an algorithm, defer steps to subclasses)

Table 4.4 The descriptions of some SDAs related to the When interrogative

When: These SDAs describe the context of the SSM
Situational factor: a factor of the organizational context that is problematic (e.g., legacy systems transformation strategy) **Business model:** a model of how an organization develop software (e.g., prototyping on contract) **Policy:** a position of governance that organize control over humans (e.g., politic for the security of information) **Organizational risk:** a risk at strategic level for an organization (e.g., development paradigm shift)
Standard: a set of requirements, specifications, guidelines, or characteristics (e.g., the international standard ISO 42010) **Convention:** a de facto standard (e.g., naming convention)
Architectural risk: a risk at architectural level for architecting a design fragment (e.g., layer bridging) **Architectural assumption:** taking for granted some SDAs in the SSM for architecting a design fragment (e.g., the properties of a layer)
Design risk: a risk at the detailed design level for designing a software product (e.g., deprecating an operation) **Design assumption:** taking for granted some SDAs in the SSM for designing a software structure (e.g., the signature of an operation)

Table 4.5 The descriptions of some SDAs related to the What interrogative

What: These SDAs provide the targets for the solution space
Requirement: a condition that is realizable by a software product (e.g., the software product shall provide up-to-date status in debug mode) **Organizational constraint:** a limit that constrains some SDAs in the SSM (e.g., object-oriented technologies)
Architectural property: a condition about a property of the elements or relations of a design fragment (e.g., performance, object-oriented paradigm) **Architectural constraint:** a limit that constrains the elements or relations of a design fragment (e.g., a unit of software is allocated to exactly one layer)
Scenario: a description of how a software product should respond to a stimulus
Operation contract: an operation that is part of a module interface

Table 4.6 The descriptions of some SDAs related to the Which interrogative

Which: These SDAs provide the elements and relations of the solution space
Domain object: a human, device, or software interacting with the system to execute some tasks (e.g., subclass writer) **Structure of domain objects:** a set of domain objects interacting with the system to execute some tasks (e.g., development team)
Design pattern: a description of how the elements of a design fragment relate to each other in order to address a design concern (e.g., client-server style) **Design tactic:** a description of how a quality attribute can be controlled by using a design tactic to achieve a response measure (e.g., use an intermediary)
Structural fragment: a set of elements and relationships of a design fragment (e.g., instantiation of the template method)
Structure of modules: a set of elements and relationships of a software structure (e.g., implementation of the template method)

Table 4.7 The descriptions of some SDAs related to the How interrogative

How: These SDAs provide the behaviors of the solution space
Process: a description of a sequence of activities, inputs, and outputs **Activity:** a description of a sequence of tasks of a process **Task:** a description of a step of an activity **Procedure:** a description of the tasks, inputs, and outputs of an activity
Pattern of interactions: a description of how the elements of a design fragment should interact (e.g., client-server style)
Behavioral fragment: a description of the interactions among a set of software elements of a design fragment (e.g., instantiation of the client-server style)
Behavior of components and connectors: a description of the interactions among a set of software elements of a software structure (e.g., implementation of a client-server protocol)

Table 4.8 The descriptions of some SDAs related to the Where interrogative

Where: These SDAs describe where the elements of the solution space are allocated
Allocation of domain objects: a description of where the elements of the organizational solution space are allocated (e.g., an activity of a process allocated to a work station)
Pattern of allocation: a description of where the elements of a design fragment should be allocated (e.g., work assignment style)
Allocation of architectural elements: a description of where the elements of a design fragment are allocated (e.g., a software process allocated to a processor)
Allocation of components and connectors: a description of where the elements of a software structure are allocated (e.g., an instantiated module allocated to a software process)

4.6 Task 4 – Format the relationship

Task 4 aims at formatting the relationship between the SDAs using the proposed description format. The relationships from the DK base will be used for identifying any match in meaning between a candidate relationship and a formatted relationship.

4.6.1 The proposed relationship description format

The SAM framework identifies some relationships between the SDAs from the literature [Iso42010, Ovas10, Zimm09, Shah09, Pari08, Kim09, Bass03, Clem03, Gamm95] – see Table 4.9. Each relationship is described using a unique identifier, a description of the relationship, and the SDAs between which the relationship applies.

Table 4.9 The relationships of the SAM framework

Relationship	**Description of the relation SDA-to-SDA**
Mandatory [Kim09, Zimm09]	A SDA mandatories another SDA : Property-to-Concern, Concern-to-Tactic, Tactic-to-Tactic, Pattern-to-Tactic
Optional [Kim09]	A SDA optionally implies another SDA : Property-to-Concern, Concern-to-Tactic, Tactic-to-Tactic, Pattern-to-Tactic
Exclusive-or [Kim09, Zimm09]	A SDA excludes another SDA : Property-to-Concern, Concern-to-Tactic, Tactic-to-Tactic, Pattern-to-Pattern
Inclusive-or [Kim09, Zimm09]	A SDA may be used with another SDA : Tactic-to-Tactic, Pattern-to-Pattern
Constrain [Kim09]	A SDA constrains another SDA : Tactic-to-Tactic, Pattern-to-Tactic
Encapsulate [Iso42010, Gamm95]	A SDA encapsulates another SDA : Structure-to-Operation
Generalize [Iso42010, Gamm95]	A SDA generalizes another SDA : Structure-to-Structure
Specialize [Iso42010, Gamm95]	A SDA specializes another SDA : Structure-to-Structure, Scenario-to-Property
Compose [Iso42010, Gamm95]	A SDA composes another SDA : Structure-to-Structure
Aggregate [Iso42010, Gamm95]	A SDA aggregates another SDA : Structure-to-Structure

Realize [Iso42010]	A SDA realizes another SDA : Structure-to-Operation
Instantiate [Iso42010, Jans08]	A SDA instantiates another SDA : Fragment-to-Pattern, Structure-to-Pattern, Structure-to-Tactic
Influences [Zimm09]	A SDA influence another SDA : Concern-to-Concern
Refinedby [Zimm09]	A SDA is refined by another SDA : Concern-to-Concern
DecomposesInto [Zimm09]	A SDA decomposes into another SDA : Concern-to-Concern
Triggers [Zimm09]	A SDA triggers another SDA : Pattern-to-Concern

4.7 Task 5 – Structure the SDAs

Task 5 aims at structuring the SDAs. To establish the relationships between the SDAs, the relationships extracted from the description are combined with the relationships from the SAM framework. The network of SDAs can be derived through the selection of relationships in the resulting set of candidate relationships.

4.8 Task 6 – Infer the SSM

The tasks of the classification technique aim at inferring the SSM using the SDAs and relationships extracted from the descriptions and the inference heuristics. The extracted SDAs can be combined with the SDAs from the existing SSMs. The SSM can be inferred through the analysis of the resulting set of SDAs.

4.8.1 The proposed inference heuristics

Table 4.10 to Table 4.15 present the inference heuristics proposed for inferring a SSM using the classified SDAs and the formatted relationships. The inference heuristics aim at controlling the level of cohesiveness between the SDAs of a SSM. Only one SDA drives the cohesiveness of a SSM (i.e., any SDA within this SSM must be cohesive with this driver SDA).

Table 4.10 Inference heuristics for the SDAs related to the Why interrogative

SDAs	Inference heuristics
Need, Goal	- Describe reasoning for the organizational problem space - Not directly measurable - Influence all SDAs of a software structures map (SSM)
Architectural concern, Design concern	- Part of the design knowledge base - Describe concerns for the SSM's design space - Influence all SDAs of a SSM's design space - Relate to a goal in the SSM
Architectural rationale	- Set rationale for elements and relations of a design fragment - Relate to an architectural concern in the SSM
Design rationale	- Set rationale for elements and relations of a software structure - Relate to a design concern in the SSM

Table 4.11 Inference heuristics for the SDAs related to the When interrogative

SDAs	Inference heuristics
Situational factor, Business model, Politic, Organizational risk	- Describe the organizational context - Influence some SDAs of the SSM - Relate to a concern in the SSM
Standard, Convention	- Part of the design knowledge base - Describe the context of the SSM's design space - Influence some SDAs of the SSM's solution space - Relate to a SDA in the SSM's organizational space
Architectural risk or assumption	- Describe the architectural context of the SSM's - Influence some SDAs of the SSM's solution space - Relate to a SDA in the SSM's organizational space
Design risk or assumption	- Describe the design context of the SSM - Influence the elements and relations of a software structure

Table 4.12 Inference heuristics for the SDAs related to the What interrogative

SDAs	Inference heuristics
Requirement, Organizational constraint	- Describe an organizational condition or limit - Influence some SDAs of the SSM's solution space - Relate to a goal in the SSM
Architectural property, Architectural constraint	- Part of the design knowledge base - Describe an architectural condition or limit - Not directly measurable - Influence some SDAs of the SSM's design space - Relate to a goal in the SSM
Scenario	- Describe a stimulus on the system and a measure of its response - Directly measurable - Influence some SDAs of the SSM's design space - Relate to an architectural property or constraint in the SSM - Relate to a design fragment in the SSM
Operation contract	- Describe an operation of a module interface - Relate to a software structure in the SSM

Table 4.13 Inference heuristics for the SDAs related to the Which interrogative

SDAs	Inference heuristics
Structure of domain objects	- Describe the elements, relationships, and responsibilities of the organizational solution space
Pattern, Tactic	- Part of the design knowledge base - Describe the structural elements, relationships, and responsibilities of a design fragment
Structural fragment	- Describe the architectural elements, relationships, and responsibilities of a design fragment
Structure of modules	- Describe the design elements, relationships, and responsibilities of a software structure

Table 4.14 Inference heuristics for the SDAs related to the How interrogative

SDAs	Inference heuristics
Process, Activity, Task, Procedure	- Describe the behavior of some elements in the SSM's organizational solution space - Relate to some SDAs of the design solution space
Pattern of interactions	- Part of the design knowledge base - Describe the pattern of interactions of the software elements - Relate to a behavioral fragment in the SSM
Behavioral fragment	- Abstract and project-specific - Describe the interactions among the elements of a design fragment in the SSM
Behavior of components and connectors	- Concrete and project-specific - Describe the interactions among the elements of a software structure in the SSM

Table 4.15 Inference heuristics for the SDAs related to the Where interrogative

SDAs	Inference heuristics
Allocation of domain objects	- Describe the allocation of some domain objects in the SSM's organizational solution space - Relate to some SDAs of the SSM's design solution space
Pattern of allocation	- Part of the design knowledge base - Describe a pattern of allocation of software elements - Relate to a design fragment in the SSM
Allocation of architectural elements	- Project-specific - Describe the allocation of the elements of a design fragment in the SSM
Allocation of components and connectors	- Project-specific - Describe the allocation of the elements of a software structure in the SSM

4.8.2 The proposed Software Structures Map (SSM)

The *software structures map* (SSM) is:

- an instantiation of the CS, and
- a matrix of traceability.

A SSM records design knowledge (DK) about a software design. SSMs should be managed as part of the DK. A SSM captures DK about direct or indirect relationships between SDAs. The SAM framework relies on this knowledge base of SSMs which trace the SDAs used during the design process. Table 4.16 presents the table format used for representing a SSM. Each interrogative regroups only the SDAs classified into the corresponding column of the CS. The SDA type gives the corresponding line of the CS.

Table 4.16 The table format used for representing a SSM

SDA Type	SDA Description
Why	
When	
What	
Which	
How	
Where	

4.9 Summary of contributions

The contributions of this chapter are:

1. a technique for:
    a. extracting and structuring the SDAs using the SSMs; and
    b. transforming textual descriptions to networks of SDAs.
2. a classification scheme and a decision tree for classifying the SDAs;
3. work instructions for supporting the creation of a SSM; and
4. descriptions of SDAs and relationships based on a uniform SSM format;

CHAPTER 5

A TECHNIQUE FOR DESCRIBING ARGUMENTS

5.1 Introduction

This chapter presents the argumentation technique developed during the second activity of Phase 3 of our research methodology in Figure 1.3. Argumentation is defined as reasoning using imperfect knowledge by eliciting arguments for exploring issues. The SAM framework uses arguments for describing how the selected SDAs may impact the activities and dimensions under examination. An activity relates to other activities for constituting a process of an organizational system. Work teams have to take into account the SDAs they used in order to adapt how to perform some activities, and to address the issues of the systems they are developing.

The proposed argumentation technique of the SAM framework includes two steps:

1) selecting the SDAs from one or more SSMs, and the activities being examined for eliciting the issues that occur by using each SDA, and
2) describing the arguments that explain how the issues may impact the activities and dimensions.

This chapter describes the tasks, inputs, and outputs of the argumentation technique, as presented in Figure 5.1. The hypothesis is that describing arguments using the technique proposed in this chapter should support the identification of important issues that occur by using a SDA such as a pattern, tactic, or style during the development of a system.

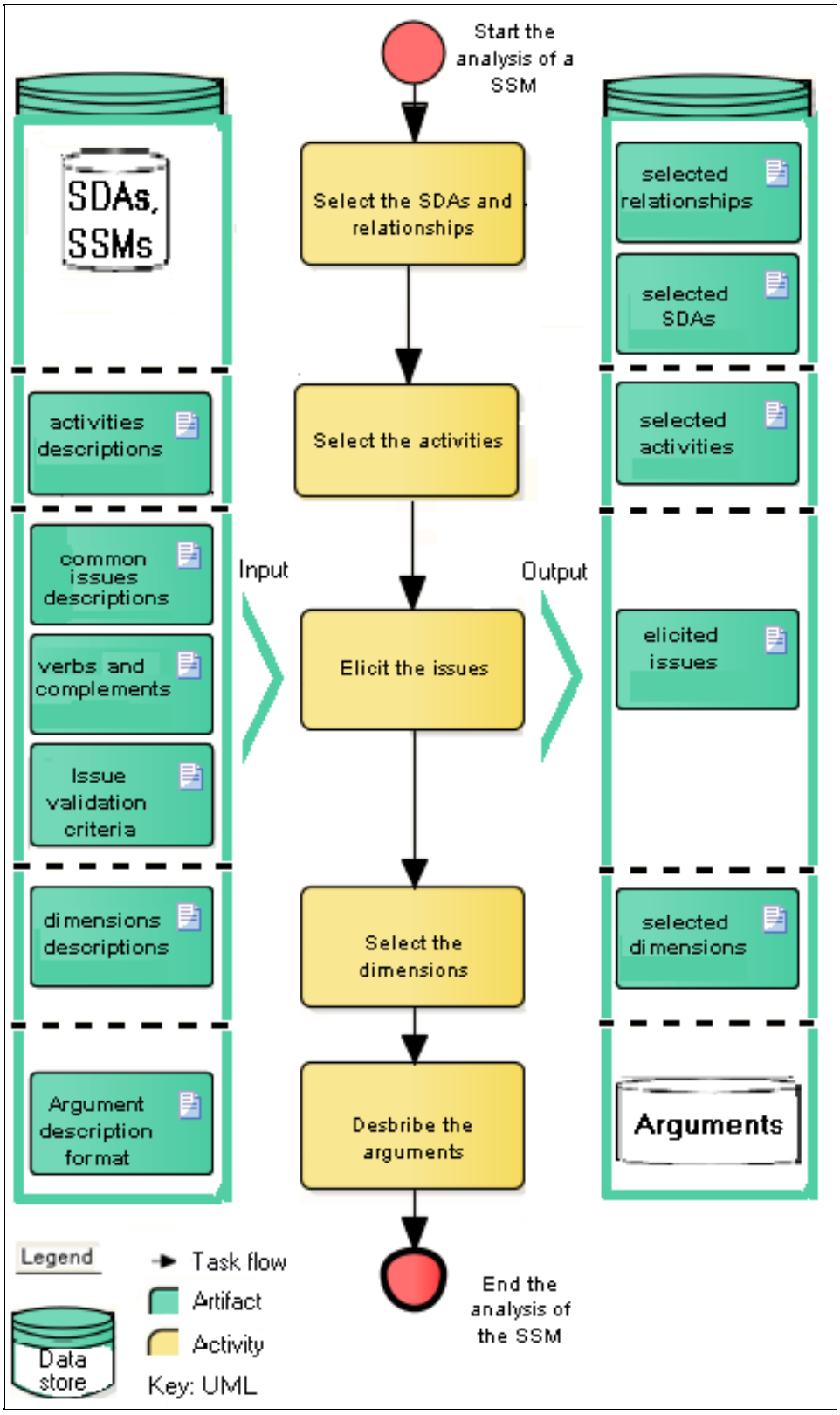


Figure 5.1 The argumentation technique of the SAM framework

5.2 The tasks of the argumentation technique

Five tasks constitute the proposed argumentation technique – see Figure 5.1:

1) select the SDAs and relationships,
2) select the activities,
3) elicit the issues,
4) select the dimensions, and
5) describe the arguments.

The classification technique of the SAM framework (see CHAPTER 4) will have produced the SSMs. The argumentation technique proposes to use the SSMs for selecting the SDAs and relationships (task 1). The selected SDAs and relationships, and the activities descriptions of both the design process (see Section 1.5) and design knowledge management process (see Section 1.7.3) will be used for selecting the activities to be examined. Then, the selected SDAs, relationships, and activities, a list of common issues (see Section 5.5.2), and an issue description format (see Section 5.5.1) will be used for eliciting the issues (task 3). Then, the elicited issues and the proposed dimensions descriptions will be used for selecting the dimensions (task 4) to be examined (see Section 5.6). Finally, the argumentation technique aims at describing the arguments (task 5) that provide reasoning about the elicited issues and their plausible impacts on the selected activities and dimensions.

The impacts of an argument may differ depending on the context of use of a SDA. The arguments may be analysed iteratively for addressing the issues that relate to the utilization of the SDA. The analysis of the arguments is part of the analysis technique of the SAM framework. The proposed analysis technique will be described in CHAPTER 6.

5.3 Task 1 – Select the SDAs and relationships

During the execution of a design process, the arguments related to a SDA will usually vary from being relevant to being irrelevant as a result of the evolution of the software being designed. Therefore, the arguments will usually be described iteratively. The selection of the SDAs and relationships will depend on the participants that execute the argumentation technique, the design process they are executing, and the focus of the current iteration. The argumentation technique allows different understandings of the SDAs as the design evolves.

5.4 Task 2 – Select the activities

In the SAM framework, the activities of the design process are explicitly related to the SDAs through the classification scheme (see Section 4.5.2). In addition, the activities of the design knowledge management process are related to all SDAs; this means that any SDA may be acquired, defined, reused, communicated, shared, and managed. Specifically, a SDA is related to an activity if the addition of the SDA to a system may cause change in the activity description being examined. The activities descriptions are usually formatted and divided into sections (e.g., Activity identifier, Tasks, Inputs, and Outputs) according to a template, as described in [ETVX, NASA, Iso12207]. An activity description refers to a set of cohesive tasks [Iso24765]. The tasks are cohesive as they contribute to the achievement of a common goal. A task usually relates to one or more SDAs.

5.5 Task 3 – Elicit the issues

5.5.1 The proposed issue description format

An issue occurs by introducing a SDA into a system being developed. One or more issues may be elicited for every identified change to a system. The argumentation technique uses a specific format to describe the issues. An issue description is composed of a SDA (subject), a verb, and a complement, as presented in Table 5.1. Each issue description summarizes a problem that occurs by using a SDA.

Table 5.1 Examples of issue descriptions using a SDA, a verb, and a complement

SDA	Verb and Auxiliary	Complement
Object-oriented paradigm	Is not	Mastered
Template method	Is	Subject to change

The SAM framework proposes a list of verbs such as in Table 5.2 that will be used for describing the issues. Each verb is described using an identifier and a description, and is related to a list of usual complements. The verbs are described in terms of shared meaning components and similar syntactic behavior of words used for describing issues. Verbs do not provide means for full semantic inference; however, they capture abstractions (e.g. syntactic or semantic) that provide additional data about the issues, and they express something that alters the meaning of the issues descriptions. Verbs also support change from ad-hoc issues descriptions to predicate-issue structures. The verbs are used as a mean to ease the elicitation of issues and to provide an issue description format.

Table 5.2 The proposed list of verbs

Verb and Auxiliary	Description
Is / Is not	Express an intrinsic state of being
Has / Has not	Express an extrinsic state of being
Do / Do not	Express an action or an absence of action
Can / Can not	Express a possibility or a limit
Exclude / Require	Express a binding between multiple parties
Augment / Lack / Make / Reduce	Express a consequence of a state of being
Shall / Must	Express a requirement or an unavoidable action
Should / Should not	Express recommendation or possibilities
Will / Will not	Express a self-declaration of intent
May / May not	Express a permission or a restriction

5.5.2 The proposed common issues

A list of common issues will be used as a support for eliciting issues. Some of the issues presented in Section 1.7.1 are described in Table 5.3 using the issue description format proposed by the SAM framework.

Table 5.3 Examples of common issue descriptions for the SAM framework

Issue description from the literature	SDA type	Verb	Complement
Lack of traceability	Requirement	Lack	traceability
Limited analysis capability	Behavioral fragment	Have	a limited analysis capability
Locating the expertise	Domain object	Cannot	locate the expertise
Lack of recipient motivation	Domain object	Lack	motivation
Lack of source motivation	Domain object	Lack	motivation
Lack of recipient absorptive capacity	Domain object	Lack	absorptive capacity
Lack of recipient retentive capacity	Domain object	Lack	retentive capacity
Lack of trust relationships	Domain object	Lack	trust relationships
Misunderstanding of the design knowledge (DK)	Domain object	Be	misunderstanding the DK
Need for tailored design knowledge	Domain object	Require	tailored DK
Tacit, implicit, explicit design knowledge	Domain object	Require	explicit DK

5.5.3 The proposed issue validation heuristics

The issue validation heuristics aims at verifying that the issue descriptions adhere to the criteria described in Table 5.4. These criteria are adapted from the description of the characteristics of a good software requirements specification detailed in [Ieee860].

Table 5.4 The proposed issue validation heuristics

Criterion	Issues validation heuristic
Correct	An issue is correctly described if it has all required criteria
Atomic	An issue is atomic if it relates to only one SDA
Unambiguous	An issue is unambiguous if it is described in terms that only allow a single interpretation
Complete	An issue is complete if relevant descriptive information is provided
Consistent	An issue is consistent if there are no conflicts within its description
Unique	An issue is unique if there is no other issue that allow the same interpretation
Analyzable	An issue is analyzable if analysis can be made completely, consistently, and correctly
Verifiable	An issue is verifiable if a person or tool can check it for correctness

5.6 Task 4 – Select the dimensions

A dimension (e.g., quality) is a perspective on a set of evaluation results used to determine the successful utilizations of a SDA (e.g., object-oriented paradigm). A SDA relates through its intrinsic issues (e.g., the object-oriented paradigm is not mastered) to the dimensions that it may impact. Five dimensions (adapted from [Wieg97]) are described in Table 5.5: functions, people, budget, schedule, and quality.

The SAM framework mandates that a dimension be described using an identifier and a generic question (as proposed in [Zimm12]). For each dimension, we propose to use a generic question that summarizes the impacts (+ or -) of any issue on the evaluation results. For example, the impacts of any issue on the quality may be summarized by the following generic question: what is the estimated impact of the issue in terms of the capability of the system to deliver (+) or not (-) quality? This generic question may be made specific for any issue and evaluation result being examined during the analysis: what is the estimated impact of the issue ‘The object-oriented paradigm is not mastered’ in terms of the capability of the system to deliver (+) or not (-) ‘reusable modules’?

Table 5.5 The proposed dimensions of the SAM framework

Dimension	Generic question	Example of a specific question
Functions	What is the estimated impact of the issue X in terms of the capacity of the system to execute (+) or not (-) the software function Y?	What is the estimated impact of the issue 'The object-oriented language is not appropriate' in terms of the capacity of the system to execute (+) or not (-) the software function 'Load balancing'?
People	What is the estimated impact of the issue X in terms of the capacity of a human to execute (+) or not (-) the task Y?	What is the estimated impact of the issue 'The object-oriented paradigm is not mastered' in terms of the capacity of a 'Programmer' to execute (+) or not (-) the task 'Implementing a subclass'?
Budget	What is the impact of the issue X in terms of the number of budgeted resources saved (+) or invested (-) to execute a task Y?	What is the estimated impact of the issue 'The object-oriented paradigm is not mastered' in terms of the number of budgeted resources saved (+) or invested (-) to execute the task 'Implementing a subclass'?
Schedule	What is the estimated impact of the issue X in terms of the number of work hours saved (+) or invested (-) to execute a task Y?	What is the estimated impact of the issue 'The object-oriented paradigm is not mastered' in terms of the number of work hours saved (+) or invested (-) to execute the task 'Implementing a subclass'?
Quality	What is the estimated impact of the issue X in terms of the capability of the system to deliver (+) or not (-) quality?	What is the estimated impact of the argument 'The object-oriented paradigm is not mastered' in terms of the capability of the system to deliver (+) or not (-) 'reusable modules'?

5.7 Task 5 – Describe the arguments

5.7.1 The proposed argument description format

The SAM framework describes an argument using an aggregation of factors. A factor is defined as an essential element for planning the utilization of SDAs such as design patterns, tactics, or styles. In the SAM framework, an argument aggregates at least five factors. Table 5.6 presents the names and the descriptions of the five proposed factors constituting any argument description. A factor may be a SDA, an issue, a reasoning description, an activity, or a dimension. The argument description ties the factors altogether.

Table 5.6 Factors constituting the argument description of the SAM framework

Name	Description
SDA	Software design artifact being examined
Issue	Problem that occurs by using or not using a SDA
Reasoning	Reasoning description about an issue or a solution
Activity	Set of cohesive development tasks
Dimension	Perspective on a set of evaluation results

An issue is related to a dimension if there is any suspicion that the occurrence of the issue in a system may produce the variation (+ or -) of a dimension evaluation result. The generic question associated to each dimension is used as a means to facilitate thinking about relevant variations. Each dimension will be examined in turn.

A reasoning description about an issue describes the chain of reasoning that ties together the argument's parts: it exposes the relationships between a set of factors. The following shortened reasoning description refers to three SDAs, an activity, an issue, and four dimensions: "Using an object-oriented paradigm requires levels of skills, expertise, and knowledge. The software designer does not master the object-oriented paradigm. This issue impacts the software product's quality."

The two last parts of the argument description format specify the scope of the argument. It refers to activities and dimensions that are strengthened (+) or weakened (-) by the argument. The activities are inferred from the activities related to the SDA exposed in the argument's reasoning description while the dimensions are inferred from the dimensions impacted by the issue that prompted the argument.

5.7.2 The proposed argument validation heuristics

The argument validation heuristics aims at verifying that the argument descriptions adhere to the criteria described in Table 5.7. These criteria are adapted from the description of the characteristics of good software requirements specification detailed in [Ieee860].

Table 5.7 The proposed argument validation heuristics

Criterion	Arguments validation heuristic
Correct	An argument is correctly described if it has all required criteria
Unambiguous	An argument is unambiguous if it allows a single interpretation
Complete	An argument is complete if relevant descriptive information is provided
Consistent	An argument is consistent if there are no conflicts within its description
Unique	An argument is unique if there is no other argument that allow the same interpretation
Analyzable	An argument is analyzable if analysis can be made completely, consistently, and correctly
Verifiable	An argument is verifiable if a person or tool can check it for correctness

5.8 Summary of contributions

The contributions of this chapter are:

1. a technique for eliciting issues and describing arguments using the SSMs and SDAs;
2. an argument format for relating the SDAs to their factors of influence (i.e., SDAs, issues, reasoning, activities, dimensions);
3. work instructions for supporting the description of issues and arguments; and
4. descriptions of issues and arguments based on uniform description formats.

CHAPTER 6

A TECHNIQUE FOR ANALYZING ARGUMENTS

6.1 Introduction

This chapter presents the analysis technique developed during the third activity of Phase 3 of our research methodology in Figure 1.3. This chapter describes a technique to support a systematic analysis of the SSMs and arguments. The proposed analysis technique will use:

- the ranked dimensions and activities for inferring the SSMs to be analyzed,
- the inferred SSMs for providing the SDAs and relationships to be analyzed,
- the selected SDAs and relationships for inferring the arguments to be analyzed,
- the ranked arguments for producing quantitative information in views, and
- the views for identifying relevant arguments related to the utilization of the SDAs.

The impacts of an argument will differ depending on the context of use of a SDA. Section 6.2 presents the tasks of the analysis technique, and Section 6.3 to Section 6.6 describes these tasks. Section 6.5.1 presents an example of structured arguments. Section 6.7 presents the summary of the contributions.

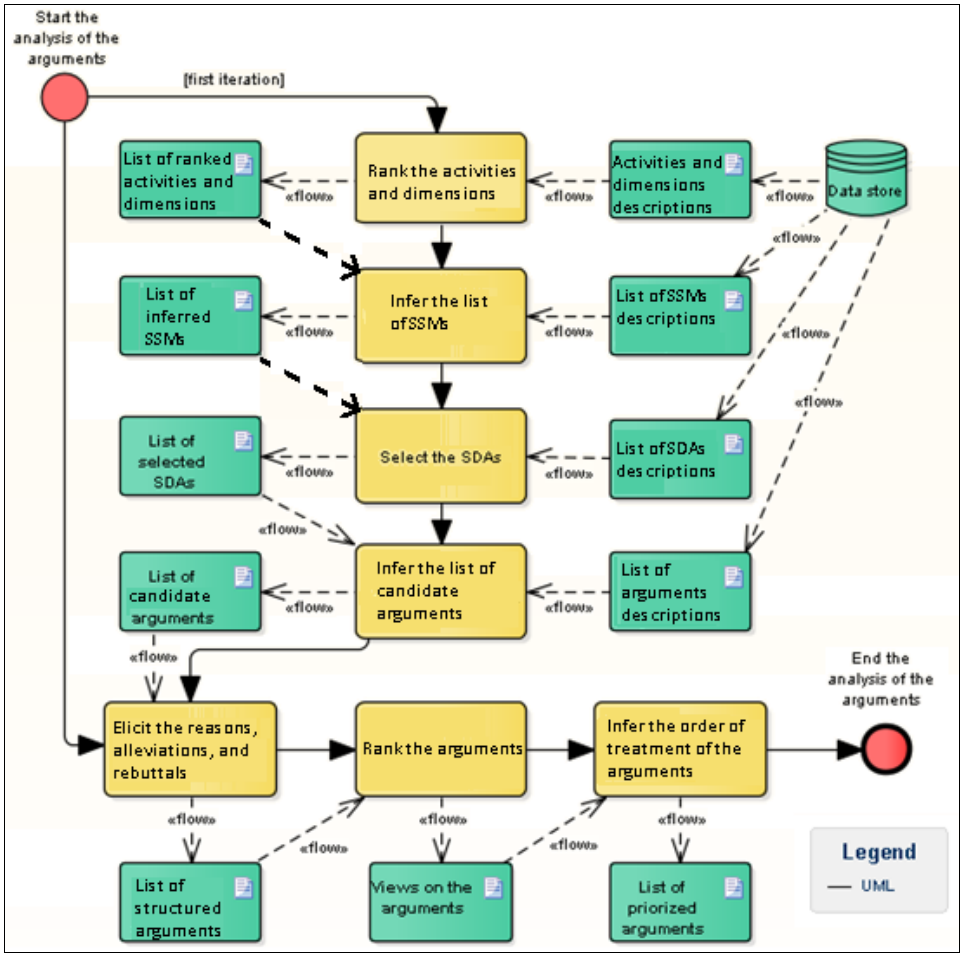


Figure 6.1 The analysis technique of the SAM framework

6.2 The tasks of the analysis technique

The following tasks constitute the proposed analysis technique of the SAM framework:

1) rank the activities and dimensions
    - for inferring the list of SSMs,
2) select the SDAs and relationships
    - for inferring the list of candidate arguments,
3) describe the reasons, alleviations, and rebuttals for the structured arguments,
4) rank the arguments and generate views
    - for inferring the order of treatment of the arguments.

The analysis technique will produce weighted arguments and quantitative views.

6.3 Task 1 – Rank the activities and dimensions

The technique will use rankings for evaluating how much each activity and dimension is relevant for a project's context. The rankings will differ depending on the project's context. The technique will use rankings for filtering the arguments that should be further analyzed.

6.4 Task 2 – Select the SDAs and relationships

The analysis technique will use the classification scheme and the ranked activities to infer the list of candidate SSMs. The rows of the classification scheme that correspond to the ranked activities will provide the SDAs from which the SSMs will be inferred. Then, the technique will use the selected SDAs, their related arguments, and the ranked dimensions to infer the list of candidate arguments. The arguments that relate to the ranked dimensions will be the candidate arguments.

6.5 Task 3 – Describe the structured arguments

The list of candidate arguments will be used for eliciting reasons, alleviations, and rebuttals. For each candidate argument, Task 3 iterates on four steps performed as follows:

1) select the candidate argument being examined;
2) describe its reasons, alleviations, and rebuttals;
3) structure the resulting arguments;
4) verify that the structured argument is correctly described.

The output of this task is a list of structured arguments. The first step of Task 3 aims at selecting the candidate argument being examined by reasoning about how its related issue may impact the ranked activities and dimensions considering the project's context. The second step aims at describing reasons, rebuttals, and alleviations that affect the intensity of the candidate argument. The third step aims at structuring the candidate argument, which implies relating it to its reasons, rebuttals, and alleviation. Finally, the fourth step aims at verifying that the structured argument is correctly described, as characterized by the proposed arguments validation heuristics in Table 5.7.

6.5.1 The proposed structured argument format

An *argumentation* structures a set of arguments. The primary argument provides the claim, reasoning, activities, and dimensions of the argumentation. Reasons, rebuttals, and alleviations are connection points. The reasons are arguments that support the claim. The rebuttals are counter-arguments for the claim. The alleviations are arguments that affect the intensity of the argument. These related arguments describe how the SDAs may contribute to the creation or resolution of issues. For example, the SDA "Naming Convention" is used to describe the rebuttal or alleviation "The naming convention is well described" for the candidate argument "The hook operations are not well identified".

Table 6.1 The structured argument format

Argument: argumentation's claim, reasoning, activities, and dimensions
Reasons: arguments that support the claim
Rebuttals: arguments that establish the falsity of the claim
Alleviations: arguments that reduce the intensity of the claim

The primary argument supports the elicitation of reasons that augment the intensity of the argumentation's claim. For example, both issues "The extensibility objectives are not well defined" and "The deferred steps are not well known" are parts of reasons that support the claim "The template method is subject to change".

6.6 Task 4 – Rank the arguments and generate views

The technique will use rankings for evaluating how much each argument is relevant to a project's context. The rankings will differ depending on the project context. The technique will use the rankings for adjusting the weights of these arguments. The arguments will be contextualized and their weights will be calculated using the rankings. Each argument is potentially the root of a tree of arguments that contains reasons, rebuttals, and alleviations. The arguments relating to the most prioritized activities, dimensions, and arguments will produce higher values in the contextual (i.e., quantified) views. The analysis technique will infer a generic multi-dimensional view of the arguments that relate to the selected activities and dimensions under analysis.

6.6.1 The proposed multi-dimensional views

Table 6.2 presents an example of a view where the rows are labeled with the activities designing, implementing, and managing, and the columns are labeled with the dimensions functions, people, and quality.

Table 6.2 Example of a generic multi-dimensional view

	Function	**People**	**Quality**
Designing			
Implementing			
Managing			

The rankings of the activities, dimensions, and arguments will generate contextual views that are subjective and quantified. These views will be used to identify critical factors to the project, which correspond to view's cells that have higher values. The view's cells will be prioritized based on their values. The most prioritized cell (i.e., with a priority of 1) will be used for reasoning further about factors that relate to this cell in order to nullify or reduce its value. Then, after these critical factors are addressed, their ranking will be adjusted.

The adjusted rankings will provide new priorities. The analysis technique will iterate these steps (i.e., identifying flaws and taking actions accordingly) until the user is satisfied with the values in the views (i.e., specific threshold values are attained). The weighting may be different depending on the project's context and nature. These rankings are used for filtering the arguments that shall be further analyzed from the multi-dimensional view. One experiment for applying the SAM framework was in the context of an undergraduate course of object-oriented software design at ETS. The project analyzed in this experiment focused on the design and implementation of the skeleton of a dice game software framework (DGSF). Table 6.3 presents a contextualization of the factors and a view for the DGSF.

Table 6.3 Contextualization of the dice game framework

Activities' rankings for the analysis	
Architecting	M
Designing	H
Implementing	M
Managing	L
Dimensions' rankings for the analysis	
Budget	X
Functions	M
People	M
Quality	H
Schedule	M

Arguments' rankings for each iteration			
Arg.	**Iter1**	**Iter2**	**Iter3**
1	L	L	L
2	H	H	H
6	M	L	X
7	L	X	X
9	H	H	H
11	X	X	X
15	L	L	L
22	H	L	X
24	X	X	X
25	H	X	X
29	X	X	X

Figure 6.2 Multi-dimensional view of the arguments related to the DGSF

Activity	Dimension			
Iteration 1	**F**	**P**	**Q**	**S**
A	10	11	3	8
D	6	5	1	2
I	13	12	4	7
M	16	15	9	14
Iteration 2				
A	7	6	1	5
D	14	13	16	12
I	11	10	15	9
M	8	4	2	3

6.7 Summary of contributions

The contributions of this chapter are:

1. descriptions of views based on a uniform description format;
2. a technique for creating views using activities and dimensions; and
3. a technique for inferring the order of treatment of the arguments using ranked factors.

CONCLUSIONS AND FUTURE WORK

To use design knowledge (DK), software designers face challenges such as understanding and tailoring styles, design patterns, and tactics. Such software design artifacts (SDAs) may be complex and their interactions with other SDAs are not always obvious. The software designer needs to add significant amounts of details to produce an implementable design, which may reduce the claimed benefits of the styles, patterns, and tactics. To take full advantage of the DK, the designers need frameworks and tools to manage this knowledge but also to relate it to the decisions taken and the resulting artifacts of the software design. Many organizations maintain artifacts and tailored information items in databases to help the document control, development, and maintenance activities.

Research contributions

The software designers should benefit from systematic support. This research project allowed the development of the Software Architecture Mapping (SAM) framework introduced in CHAPTER 2. The SAM framework aims at supporting software designers in managing the DK during the design process. Specifically, this project allowed the development of the following solutions for supporting the software designers, illustrated with case studies:

1. A reference model for describing the SDAs, including styles, design patterns, and tactics along with their relationships.
2. A technique for populating a DK base using the reference model and the descriptions of design patterns, tactics, and styles.
3. An argument format for describing the issues and impacts related to the utilization of the SDAs in particular contexts.
4. A technique for populating a DK base of arguments using the argument format.
5. A technique for supporting the analysis of the impact of design patterns, tactics, styles, or other SDAs on a software design.

CHAPTER 2 also presented a requirements self-assessment that has been conducted using the requirements from the literature on DK management (e.g., architectural documentation rules [Clem02]). In addition, the SAM framework has been applied in industrial contexts (i.e., software cockpits design and web engineering) and academic contexts (i.e., catalogs of styles, patterns, and tactics, undergraduate design courses, and web engineering) for evaluating its technical feasibility and usability for novice designers. CHAPTER 3 presented five cases studies and two xperiments.

In CHAPTER 4, we presented a technique and the reference model that support the analysis of the DK. We identified information items and established descriptive and exclusive criteria for the finer-grained SDAs composing styles, design patterns, and tactics. Then, we use finer-grained SDAs for representing styles, design patterns, tactics, and design decisions as aggregations of SDAs. Such aggregations of SDAs make discernible every part of the DK, instead of using the usual textual format that may obscure significant information. In CHAPTER 5, we proposed an argument-based technique for relating the SDAs to the activities and dimensions they impact. Finally, in CHAPTER 6, we presented a technique for supporting the analysis of the arguments using multi-dimensional views in order to systematically infer the order of treatment of the arguments in a particular context. The three proposed techniques support the software designers for managing DK. For constituting these techniques, we proposed novel contributions; reference model, identification heuristics, decision tree, classification scheme, inference heuristics, issue format, and argument format.

How the SAM framework meets the research goal and objectives

The SAM framework meets the research goal and supports the software designers when managing DK and issues that arise in designing systems. Table 6.4 summarizes the evaluation activities performed for the SAM framework and presented in Chapter 3.

Table 6.4 Evaluation activities performed for the SAM framework

Activity	Context	Inputs	Outputs
Case study	Software cockpits	Software architecture description	SDAs, SSMs and arguments
Case study	Undergraduate course (LOG121)	Work statement	SDAs, SSMs, arguments, and views
Experiment	Graduate course (SYS869)	Work statement	SDAs and arguments
Case study	Styles, tactics, and patterns	Bass03, Clem02, Gran02, Gamm95	SDAs, SSMs, and arguments
Case study	Web site of an organisation	Work statement	SDAs, SSMs, and arguments
Experiment	Pattern and web engineering	Work statement	SDAs, SSMs, and arguments
Assessment	SAD and DKM	Requirements, conclusions, and rules from the literature	Assessments results

The SAM framework meets the research goal, sub-goals (A-C), and objectives (1-7), as follows:

- The classification technique of the SAM framework presented in CHAPTER 4 aims at systematizing the creation of a SSM. This technique allows the software designers to describe (Objective 1) and classify (Objective 2) the SDAs using the descriptive and exclusive criteria, the classification scheme, and the decision tree of the SAM framework. Then, these SDAs are used to describe the styles, design patterns, and tactics using SSMs (Objective 2). Such aggregations of SDAs make discernible every part of the styles, design patterns, and tactics descriptions, instead of using the current textual format that obscures significant information. The classification technique of the SAM framework (Objective 4) was applied in a case study presented in CHAPTER 3. This case study uses the works on tactics, styles, and design patterns in [Gran08, Bass03, Clem03, Gamm94] for populating a DK base (Objective 3).

- The argumentation technique of the SAM framework (Sub-goal C) presented in CHAPTER 5 aims at describing arguments (Objective 5). This technique allows the software designers to argument about the issues that relate to the SDAs and their relationships for creating network of arguments. Then, these arguments may be used to describe the issues that relate to the styles, design patterns, and tactics. The argumentation technique has been applied in a case study presented in CHAPTER 3. This case study uses issues extracted from our literature review on design knowledge management as a starting point for populating the reference model (Objective 6).

- The analysis technique of the SAM framework (Sub-goal D) presented in CHAPTER 6 aims at systematizing the analysis of the SDAs. This technique allows the software designers to analyze the issues related to the SDAs and generate views. These views may be used for inferring the order of treatment of the issues, and then the context of utilization of the SDAs. In particular, the resulting views should help the software designers to apply styles, design patterns, and tactics in a particular context by addressing the right issues at the right time (Objective 7).

Limitations of the research project and future work

This research project permitted to identify additional future work that should contribute to enhance experiences of using the SAM framework. Future work related to the classification technique includes:

1. validating the technique through experimentation with industrial participants;
2. developing support tools for:
    a. inferring the context of utilization of design patterns, tactics, or styles;
    b. populating a design knowledge base of SDAs and SSMs;
    c. generating a SSM from a UML diagram, and vice-versa;
    d. recognizing design patterns, tactics, or styles from a SSM; and
    e. merging the SSMs of design patterns, tactics, or styles.

Future work related to the argumentation technique includes:

1. validating the technique through experimentation with industrial participants;
2. developing tool supports for:
    a. populating a design knowledge base of issues and arguments;
    b. inferring the issues and arguments related to the utilization of a SDA; and
    c. inferring the list of SDAs that provide solutions for the issues and arguments.

Future work related to the analysis technique includes:

1. validating the technique through experimentation with industrial participants;
2. developing tool supports for:
    a. inferring the order of treatment of a set of arguments;
    b. generating views from arguments, and vice-versa.

The following research tasks should also be executed in future work:

- Elaborate heuristics to apply and analyze the impacts of the tactics, design patterns, and styles described using the reference model of the SAM framework.
- Develop algorithms for analysing DK bases of SDAs, SSMs, arguments, and views.
- Develop a tool for supporting the techniques of the SAM framework.

The software designer should execute pre-defined tasks for creating SSMs, eliciting arguments, and analysing arguments. Four case studies were presented in this thesis, including a case study with participants. However, the techniques were not evaluated in the context of large software development projects in the industry. Future work should evaluate the understandability and usability of the SAM framework in industrial contexts.

Future work also includes developing a tool that supports the reference model and systematic inference of the context of application of the styles, design patterns, and tactics from the objectives. The operationalization of the reference model will provide algorithms, decision trees, and heuristics for automating the three techniques proposed in this thesis.

The rules given in [Kim09] will provide a baseline for automating the composition and binding of the styles, design patterns, and tactics. The guidelines for systematically exploiting the reference model in a tool and the quality of the data produced by the three techniques and tool will be evaluated in future work.

There are at least four future benefits of the SAM framework that should also be evaluated:

- Enhance the skills of the actors involved in the design process.
- Allow the creation of tools to support the design process.
- Support the systematic construction of software architecture.
- Support the automation of the design process.

APPENDIX I

ACTIVITIES AND SOFTWARE DESIGN ARTIFACTS OF THE DESIGN PROCESS

This section describes the activities and software design artifacts of the design process identified from the literature.

Select the objectives

Activity Description. Selecting the objectives aims at choosing the most important objectives that should drive the design decisions (DDs) about the software architecture [Bass03]. An objective is something toward which work is to be directed. The objectives define the organizational and technical problems that software architectures must address. Somehow the objectives influence the procedures and teamwork organization that support the life cycle of software architecture [Bass03]. Software architecting requires such objectives to be clearly defined, communicated, shared, evaluated, and reuse during the design process. Approaches, models, and tools exist for supporting the selection of the objectives.

Related artifacts. The first row of the proposed classification scheme (see Figure 4.1) presents common artifacts types used for sharing the objectives that a system should sustain. Needs, goals, and expectations are the rationales for the objectives, the guide for architecting. A **need** gives a meaning to the DDs: it states what is necessary for a system to be suitable (e.g., the system shall reduce maintenance costs) and for the user to realize its tasks effectively and efficiently (e.g., the system shall lift up developers' productivity). Needs are translated into more precise objectives that the solution shall achieve. A **goal** gives a specific direction to the DDs but does not specify where the end is. A goal is a desired outcome of user interaction with a product that describes in terms not directly measurable the final product (e.g., the framework shall ease developments of system-specific components).

An **expectation** (or feature [Leff03]) is a goal that is realizable under a specified organizational context by a domain object, which may be a human, device, or software interacting with the system to execute some tasks at specific locations in the environment (e.g., the system shall provide up-to-date status in debug mode). An expectation may be defined as the result of a business use-case [Leff03], which describes the actors who participate in the business activities and how these activities take place.

Needs, goals, and expectations are made realizable and measurable by translating them into software **requirements** [Leff03], which state the conditions that govern the design of the architectural elements. Altogether, these artifacts record the organizational objectives that are used by the software designers to tailor specific architectural and system artifacts.

Identify the Knowledge Artifacts

There are commonalities among the systems an organization develops and maintains. Recurring approaches are used by the organizations that lead software projects, from having standard domain models, to the way in which developers write code [Ulri02]. An organization gains efficiency when patterns can be defined by skilled practitioners and propagated across the work teams. Propagating the knowledge engenders increasing returns because it can be reused once created [Bass03].

Activity Description. The software designer uses his background to identify the design knowledge. This knowledge results from the design concerns, domain-specific and contextual information, and architectural properties and designs that have been products of successful developments of similar software architectures [Bass03]. They constitute a directory of reusable information that influences the software architecture of SISs. This activity of the design process aims to identify the DK artifacts that have proven useful in defining and attaining objectives similar to the ones selected in the previous activity. The outcome of this activity is a set of system-independent artifacts that seem to be useful for attaining the objectives.

Design Artifacts – Architectural Knowledge. The second activity of the CS summarizes the artifacts we considered as providers of the DK. We use the architectural and design concerns to bind the architectural problem space, but we consider they constitute the rationales of the DDs, not the drivers that shape the software structures. The architectural concerns refine the business goals in terms relevant for defining the architectural problem space (e.g., define the scope of a product-line [Bass03]). The design concerns are more specific about the design problems (e.g., localize changes in a component). The software designers need to address design concerns soon for architecting the right level of quality and functional capacity in SISs. More precise context about the application domain may be required to identify pertinent concerns and make the DDs. Different industries may have distinct standards, regulations, and conventions that ease interactions and reduce duplication of effort.

The architects employ the architectural and design concerns and what they know about the context of the problem for making choices about the relevant architectural properties, which may be quality attributes [Bass03] or characteristics of quality that shall be inherent in the software architecture (e.g., maintainable, distributed). Clements et al. [Clem03] define a property as additional information about entities and relations, such as names and characteristics of quality. The software designers may use quality models [Iso25010] to define the characteristics of quality of SISs. Then, they use quality attributes to identify relevant measures, which shall permit to quantify the characteristics of quality and thus more objectively evaluate the level of quality and functional capacity of SISs [Bass03].

Many DDs are made for architecting properties of SIS. The architects have the responsibility to choose, compose, apply, and maintain the set of tactics and patterns that provide the desired properties [Kim09]. These artifacts encode reusable DK about solutions to well-known problems. They provide generic solutions to address common design and architectural concerns. Each pattern and tactic may promote or disadvantage one or more properties. The inadequate usage of the patterns and tactics may cause significant impacts when crucial properties are not guaranteed. We classified tactics and patterns into the solution space, but they also provide insights and descriptions for understanding the problem space.

The software designers first use the architectural properties to identify tactics [Bass03], design patterns, and styles [Clem03] of structural, behavioral, and allocation types. An architectural pattern is a package of decisions. It describes how the entities of a design fragment relate to each other to provide properties and software structures and behaviors, which address one or more design concerns. A pattern is a description of known properties, patterns of data, control interaction, constraints, semantics, vocabularies, and types for the entities of the solution, along with qualitative reasoning about the strengths and weaknesses of the proposed solution [Gran02, Gamm95]. A pattern is a composite of multiple architectural tactics. A tactic is a basic design decision, which tailors software architectures and patterns [Scot09, Kim09, Bass03]. It aims to address a design concern [Bass03]. A tactic specifies how a single quality attribute can be controlled through a design decision to achieve a response measure [Bass03].

Indeed a pattern is a composite of multiple architectural tactics. For example, Scott and Bachmann have described the Layer pattern in [Scot09]. The Layer pattern supports the maintainability of a system by isolating each layer from changes in other layers. This isolation is achieved using many tactics to control maintainability: Semantic Coherence, Abstract Common Services, Use Encapsulation, Use an Intermediary, and Restrict Communication Paths. In addition, the Layer pattern achieves portability by encapsulating platform-specific details behind stable interfaces. The tactic “Use Encapsulation” is necessary for both portability and maintainability.

Structural tactics and patterns focus on the elements required for solving a problem, along with the relationships and responsibilities of those elements. Behavioral tactics and patterns focus on the interactions that a set of components and connectors shall perform to solve a problem, although they may imply structural solutions. Tactics and patterns of allocation type focus on allocation of elements that constitute the solution to a problem [Bass03].
Likewise, they may imply structural solutions.

Define the Architectural Artifacts

The software architecture is composed of design fragments tailored by the software designer, which design alternate solutions by making DDs for achieving the stated and implied objectives. The specialization of the architectural knowledge should permit to produce a set of architectural artifacts for defining abstract software structures (i.e., design fragment) and architectural interactions and allocations to be detailed further by the system artifacts.

Activity Description. The third activity aims to define the artifacts that describe the problem and solution spaces in terms relevant for architecting. The architectural artifacts precise the objectives identified in the upper-rows. They provide explanations, contextual information, and conditions that contribute influences on the design fragments of the software under construction. This activity should serve to define both the architectural problem space and the design fragments, which describe generic solutions for the architectural problems.

Design Artifacts – Architectural Artifacts. The third row of the CS summarizes the architectural artifacts examined in this thesis. The concerns are refined into rationales, which explain the DDs that structure the entire software. Artifacts such as generation tables and general scenarios [Bass03] capture the risks, events, assumptions, and circumstances that may affect elements of the system. They help to identify key parameters that must be reasoned about and offer a way to refine the vague requirements and architectural properties into more detailed scenarios [Bass03].

We classified the quality attribute scenarios as artifacts for recording information about the problem space. General and concrete scenarios are distinguished in [Bass03]. General scenarios are independent of any system and characterize the quality attributes that potentially any system may exhibit [Bass03]. They contain quality attribute parameters used to identify appropriate reasoning frameworks [Bach05], which encapsulate quality attribute knowledge and decision guidelines useful to understand the parameters of the problem and define the structures and behaviors of the system.

General scenarios need to be made system specific [Bass03], which consists to affect value to each part of a general scenario, for fixing the decision criteria used to evaluate the structures and behaviors of a system. A concrete scenario is an instance of a general scenario. It is a quality attribute requirement used to specify and control a quality attribute that a system shall exhibit [Bass03]. The quality attribute requirements specify the characteristics of quality (e.g., performance and usability) required for a system [Bass03]. A concrete scenario is used to describe how the system shall response to a specific stimulus in a precise context for providing an acceptable level of quality to the stakeholders.

A concrete scenario has six parts labeled source, stimulus, artifact, environment, response, and measure [Bass03]. The measure of the response is what should be tested as a threshold for the acceptable level of quality specified by the stakeholders. The quality attribute scenarios are also used to identify roles (or generic responsibilities) to be assigned later to design fragments, which define cohesive sets of architectural elements (also called entities or tailored roles) from which system specific artifacts are instantiated.

General and concrete scenarios may be managed as reusable architectural artifacts for driving DDs. The most architecturally relevant concrete scenarios are called architectural drivers in [Bass03]. They stem from business goals and user needs and some of them may be important drivers for architecting. The architects use these drivers to make DDs and tailor roles to specific problems for designing parts of the architecture, which are design fragments. Each decision of the architect may affect one (sensitivity point) or more (tradeoff point) of the architectural drivers. The architects use analysis methods [Bass03] to evaluate the alternatives and make tradeoffs among the conflictual decisions in order to reduce the risks and make the DDs that shape the design fragments in a manner that best support the drivers.

The DDs and the resulting design fragments are architectural artifacts [Tyre05]. As defined in [Jans08], a DD is a description of additions, subtractions, and modifications to the software architecture, the reasons behind the decision, and the rules, constraints, and requirements enforced by the resulting design fragment. A DD may report additional information [Tyre05, Shah09], including references to external artifacts such as plans and risks. We classified the information recorded for a DD into both the problem and solution spaces on the row third of the CS.

The generic roles identified from the DK are instantiated into design fragments and result in elements of the software architecture such as modules, components, and connectors [Bass03]. The software architect must choose, understand, bind, compose, and tailor the DK for defining the design fragments that satisfy the parameters' values defined by the concrete scenarios [Scot09, Kim09]. The design fragments comprise the entities, which represent the types of elements that will be instantiated into system-specific elements to achieve the conditions [Bass03, Fair07, Khal10]. A design fragment gives a decomposition of entities with architectural responsibilities, relationships, and interactions, and the locations of those entities in the environment. As stated by Fairbanks [Fair07], the design fragments are used to define the scale of a solution for more specific design decisions to be made. Therefore, the DDs and design fragments define the architectural designs and thus bind the detailed designs, constraining both problem and solution spaces.

Specify the System Artifacts

The specification of the system artifacts aims to refine the generic entities of the design fragments into specific architectural elements, which will be parts of the SISs. The architects need detailed rationales, system-specific contextual information, and realizable conditions to make more specific design decisions about the architectural structures under construction.

Activity Description. The fourth activity should serve to refine the architectural problems and instantiate the architectural structures, which implement the technical solutions to the problems. The major outcome should be a concrete description of the rationales, contextual considerations, and conditions of the problem space and the resulting architectural structures. The specification of the system artifacts aims to instantiate the architectural entities provided by the candidate design fragments for satisfying the conditions quantified by the concrete scenarios [Wojc06]. It should permit to refine the problem space and realize the architectural structures, which implement the technical solutions to the organizational and architectural problems.

Design Artifacts – System Artifacts. The fourth row of the CS summarizes the system artifacts we identified in this thesis. The concerns are refined into rationales, which explain the detailed design decisions that govern the implementation of the architectural elements. A quality model defines measurement thresholds (parameters' values) that specify the acceptable level of quality for the software structures under construction [Bach03]. These thresholds become the measures of the concrete scenarios [Bass03], which precise the context of use and conditions that affect specific architectural structures. Each scenario may lead to many method contracts [Meye97], which specify pre-conditions, post-conditions, and exception conditions, inputs and side effects, and invariants for methods that will supply implementations for the generic responsibilities defined in the design fragments.

The method contracts specify evaluation criteria for the software modules. They provide software documentation for the behavior of the methods and thus facilitate code reuse. The software architects bind, compose, and tailor the design fragments for architecting the software structures that satisfy the concrete scenarios [Scot09, Kim09] and the associated method contracts. The architectural structures are elaborated to realize the architectural fragments by specifying the elements, methods, properties, interactions, and locations that will characterize the software [Bass03, Wojc06]. These structures provide the architectural modules, components and connectors, and allocation schemes.

Describe the Architectural Views

The architectural description of software-intensive systems aims to communicate the DDs and the resulting software structures, behaviors, and allocation schemes of the software architecture. The description of the architectural problem and solution spaces is considered as essential for sharing the architectural knowledge with various stakeholders and understanding the impacts of future changes to the systems [Iso42010, Clem02].

Activity Description. The fifth activity should permit to produce the architectural views that describe the problems and the resulting software architecture. The major outcome of this activity should be a set of views that provide descriptions of rationales, contextual considerations, and conditions that form the software structures. A view that reports too much information may be fragmented into many view packets, each showing a fragment of the entire view [Iso42010].

Design Artifacts – Architectural Views. The concept of view is defined in [Iso42010, Clem02]. The fifth row of the CS reports the artifacts we selected for mapping the views that make up the software architecture document [Iso42010]. A view is introduced by a concise description that recaps the purpose and contents of the view. It provides explanation, justification, and reasoning about the DDs that have been made. The context of each view is defined by the view scope and symbols, and the vocabulary of the view used to show interactions with external entities.

A view conforms to a viewpoint [Iso42010, Clem02]. A viewpoint defines the purposes and audience for, the set of concerns to be treated by, and the modeling, evaluation, and consistency-checking techniques used by any conforming view [Iso42010]. A view is a description of the structures, behaviors, or allocations schemes of the software from the perspective of a cohesive set of concerns defined by a viewpoint [Iso42010]. It visually represents and textually explains a specific type of architectural elements that compose the system, their properties, and the relations among them.

Evaluate the Software Structures

Various analysis methods aim at evaluating software architectures from both organizational and architectural perspectives [Bass03]. Every method obliges the architects to identify the evaluation criteria, perform the analysis, and report the results of the assessment. The architects compare the results with the evaluation criteria and identify possible improvements. The evaluation of the software structures aims to provide quality records, which track objective evidence that the software architecture sustains the selected objectives. The architects need to record the evaluation criteria and the resulting appraisal data as parts of the DK [Bass03].

Activity Description. The sixth activity should produce the artifacts used to determine the level of achievement of the requirements. The outcome should be a set of evaluation criteria and records that provide the appraisal data. Sufficient records should be made to furnish objective evidence of quality achievements. These records shall be identifiable and made available as inputs for the acceptance and assurance processes.

System Artifacts. The sixth row of the CS presents the evaluation criteria and records we considered in this thesis. The architects use assurance criteria to provide control over the architecting activities in order to ensure that the work team is doing the job right. Then, the assessors use the acceptance criteria to determine if the requirements are met, in order to ensure that the work team did the right job. In the SAM we refer to acceptance testing by the architect prior to end the architecting iteration.

As defined in [Iso25010], software architecture "quality can be evaluated by measuring internal attributes (typically static measures of intermediate products), or by measuring external attributes (typically by measuring the behavior of the code when executed), or by measuring quality in use attributes. The objective is for the product to have the required effect in a particular context of use". The software architecture is evaluated for internal attributes. It is an interim product that is mostly seen from the internal and developers view.

Design control measures should provide appraisal data for verifying or checking the adequacy of design. The evaluation process should provide records for assessing the quality of the architectural structures, behaviors, and allocation schemes.

APPENDIX II

EXAMPLE OF A SSM FOR A SOFTWARE COCKPIT SYSTEM FRAMEWORK

This section describes our example of a SSM for an object-oriented and component-based framework required to support the development of software cockpit systems [Bass03]. We partitioned the SSM in Table A II.1 and Table A II.2, respectively the problem and solution spaces. The SSM is based on our findings about architecting flight simulators. The SDAs are extracted from our experience and the literature about software architecture, design knowledge, and architecting software-intensive systems, including flight simulators [iso42010, iso25000, Bass03, clem03, Ulri02, Mars85, Foga67, Perr66].

Reasoning descriptions for the SSM

A software cockpit system (SCS) framework provides classes, which software designers extend for developing the software that simulates the cockpit of various airplanes. In this example, the SCS framework is part of a software development kit (SDK) that also provides tools and documentation for architecting, building, and maintaining SCSs. The systems that extend and compose the classes of the SCS framework acquire the capacity to support third parties environments, including hardware dispatchers that may control the life cycle of a simulation, from loading to exiting the software systems. The framework influences the procedural solution and shape the work team that executes development tasks.

The work team is formed of software and system specialists, and junior software engineers, which participate in developing the components that constitute the software cockpit systems. The system development process is based on the concepts defined in the SCS framework, which define built-in services to ease some of the design processes. The developers work in parallel to build specialized parts of the SCS. Junior software designers follow a standard procedure for building most of the SCS components.

Table A II.1 Example of a SSM (artifacts of the problem space)

Rationale (Why)	Context (When)	Driver (What)
• Goal: Reduce maintenance costs • Sub-Goal: Eliminate design defects	• Business model: Prototyping on contract • Situational factor: Legacy systems transformation strategy • Risk: Development paradigm shift	• Constraint: Shorten schedules, Limited budgets
• Architectural concern: Scope of the framework • Design concern: Abstract common services, Localize changes, Prevent ripple effects, Defer binding time	• Regulation: FFS Level D control	• Property: Modifiability, Reusability, Framework, Component-Based, Object-Oriented, C++
• Rational: Provide core software modules, program to stable interfaces, and use inversion of control to promote reusability of pre-tested modules, • Rational: Compose and configure software systems at runtime to ease changes of implementations, and provide tool-support for editing configuration files	• Risk: Legacy motion systems depend on hardware • Assumption: Editor generates valid composition and initialization files	• Scenario: A single element controls loading, initializing, executing, and exiting software systems • Scenario: Every software systems (or units) implement a common interface
• Rational: Minimize impacts of changing the component's behaviors, composition, initialization, and destruction • Rational: Standardize the component composition, initialization, execution, and destruction methods and configuration files	• Risk: Complex customization and parallel development of multiple components • Assumption: An external dispatcher controls the simulation process	• Contract: The load method instantiates the software system • Contract: The init method initializes the software system • Contract: The execute method simulates the software system • Contract: The exit method terminates the software system
• Description: These views introduce basic components' structures, behaviors, and external relationships	• Symbol: Unified Modeling Language	• Viewpoint: Module, Component-and-Connector, Allocation
• Acceptance criteria: Designs with no major defects	• In-Use: Design time / Implementation time	• Internal: Number of design defects created

Table A II.2 Example of a SSM (artifacts of the solution space)

Structure (Which)	Behavior (How)	Allocation (Where)
•Work team: Software development	• Procedure: Component development	• Object: Single work station, Software specialists, System specialists, Junior developers
• Pattern: Layered system (bridging), Decomposition • Tactic: Limit option, Maintain interface, Use configuration file, Use standard language	• Pattern: Template method, Factory method	• Pattern: Work assignment, Implementation • File Type: C++ header, C++ source, Dynamic linked library, Windows executable, XML, ZIP
• Layer: Windows OS, C++, Foundation, Core • Decomposition: SDK [Core, Editor], Core [Component, Factory, Parser]	• Use-Case: Editor writes compo.xml, Editor writes init.xml, Core reads compo.xml, Core composes units, Core reads init.xml, Core initializes units, Core executes units, Core exits units	• File: SDK.zip [Core.dll, Editor.exe], Core.dll [Component.h, Component.cpp, Parser.h, Parser.cpp, Factory.h, Factory.cpp]
• Class: Component [execute, doExecute, load, doLoad, init, doInit, exit, doExit], Parser [parse], Factory [create]	• Template method: [load, doLoad], [init, doInit], [exit, doExit], [execute, doExecute] • Factory method: create • load reads compo.xml, calls parse, * create, doLoad • init reads init.xml, calls parse, doInit	• Process: Simulation [execute, doExecute, load, parse, create, doLoad, init, doInit, exit, doExit] • File: System.zip [System.dll, compo.xml, init.xml]
• View: System foundation classes	• View: System execution package	• View: System packaging
• V&V: Formal inspection records	• V&V: Critical flow analysis records	• V&V: Prototyping records

The development process allows complex software functions such as memory management to be handled by a group of software specialists that are responsible for designing those functions. Software and system specialists use the framework to develop software foundations for various cockpits' systems. Junior software designers concentrate their efforts on the implementation of the functionalities to be simulated. Most of the functions and tests run on a single workstation. The following reasoning descriptions are used for creating the SSM in Table A II.1 and Table A II.2.

- Three goals for architecting software product-lines: reduce maintenance costs, reduce development costs, and increase quality.
- Reduction of maintenance costs is a primary goal. Many studies report that software maintenance makes up most of the total cost of software development projects.
- Legacy systems make up a large part of the problem and solution spaces.
- Each customer has its specific customization requirements, which require prototyping and changes that often cause significant maintenance costs.
- Every cockpit needs updates after delivery to correct faults or improve its level of quality.
- Maximizing reusability of pre-tested components is essential.
- Organizations use a framework to impose a set of reusable components along with standardized component development procedures executed by structured teams, with defined roles and task-specific tools.
- A framework is a technical solution that provides proven software designs and implementations for producing better software products and significant cost savings by defining common architectural structures and behaviors. The framework can be specialized to produce custom products. By extending the framework, the software products will have similar structures, which make them easier to develop and maintain. A framework is shaped by many patterns that provide known properties. It controls the main body of execution and lets developers write the code it calls.

- The architectural properties of a framework written in C++ will be parts of any SCS. An object-oriented language such as C++ relies heavily on inheritance and dynamic binding to achieve reusability. It requires standardization and shall provide the capacity to configure custom SCSs without having to touch the implementations. Object-oriented programming techniques such as data abstraction, encapsulation, interfaces, inheritance, and polymorphism are tactics in [Scot09, Bass03] used to encapsulate variable implementations behind stable interfaces. Existing methods are reused and extended by inheriting from the framework base classes and overriding pre-defined hook methods using patterns like Template Method.
- Encapsulating together a set of operations with the data they access allows designers to decompose problems into collections of interacting components and connectors [Bass03]. This modularity localizes the impact of changes and makes the software easier to understand and maintain. In addition, layered system increases software maintainability since it enables the delivery of pre-tested components in each of the layers. Configuration files may be edited in standard XML (Extensible Markup Language) and used by a factory to instantiate generic components into specific objects.
- Classes are packaged in a dynamic link library (DLL) file and tools are provided in executable files over the Windows operating system.
- Libraries contain code and data that provide services to independent programs. This encourages the sharing and changing of code and data in a modular fashion, and eases its distribution.
- Modularity allows changes to be made in a single self-contained DLL shared by several applications without any change to the executable applications themselves.
- Executable applications and libraries link to each other through the linking process, which defers binding at runtime.

- To achieve consistency for declarations in different translation units in C++, header files contain declarations of the constants, types, data, and functions publically provided by the class.
- A header file is included in source files containing executable code and/or data definition.
- The framework aims to provide fundamental services and classes to facilitate development and maintenance of the SCSs.
- The basic classes shall be reused under various contexts (e.g., debug and instructor modes) for distinct systems (e.g., brakes and engines).
- The basic classes shall be used to create frameworks that can best fulfill particular system's needs.
- Certain systems may not be replaced in short term.
- The first prototypes will integrate procedural and object-oriented components.
- The Editor is a task-specific tool for editing configuration files used to compose and initialize only object-oriented components when simulations start.
- The SDK shall provide proper framework and toolset to develop SCSs, given the component development procedure.
- The framework shall promote a component-based programming approach.
- The framework shall provide a layer of core modules from which custom components will be written and controlled at runtime.
- The Core layer will be composed of standard software units, which can be extended and composed to build custom SCSs.
- The Core layer will sit over the foundation layer that provides services for simulating, the C++ language, and the Windows operating system, causing layer bridging.
- The standard software units will perform key abstractions such as the Component, Factory, and Parser modules that can be reused for developing the software that sits over it. The Component module encapsulates SCSs behind a stable interface and controls their execution in a standardized manner. It is the base software unit

for building SCSs components. The aircraft will be decomposed into many systems, which in turn are built using many logical components.

- A composition file permits to specify in XML and compose at load-time a component using a factory pattern. In addition, the initialization file permits to specify properties whose value will be set per instance at load time from the XML file; these properties may affect the later behavior of the component instance.
- The Editor will allow the user to visually edit the composition, properties, and parameters of the components.
- The Component module uses these configuration files to instantiate system-specific components without containing any system-specific logic.
- Each module will have a header file used to describe its API and a source file that used to define the executable code.
- The modules will be packaged in the dynamic-link library Core.dll.
- The executable file Editor.exe will provide the Editor module.
- Standardization will save time and money in development and allow for easier maintenance.
- The predictability of the development process increases as standardization is spread over more elements.
- The Component class will define the foundation on which software designers can base their systems.
- The Component class will implement the Template pattern. The template methods can be indirectly specialized (using the “do” methods), which enforces class interface stability, allows the addition of instrumentation in the base class, and lighten user’s responsibility since it is no longer required to call the overloaded method.
- The standardized Component class will ease the customization of the components that compose the various SCSs.
- The Component class shall provide the services to manage the entire life cycle of the SCSs components. It shall allow users for customizing a component that extends the Component class.

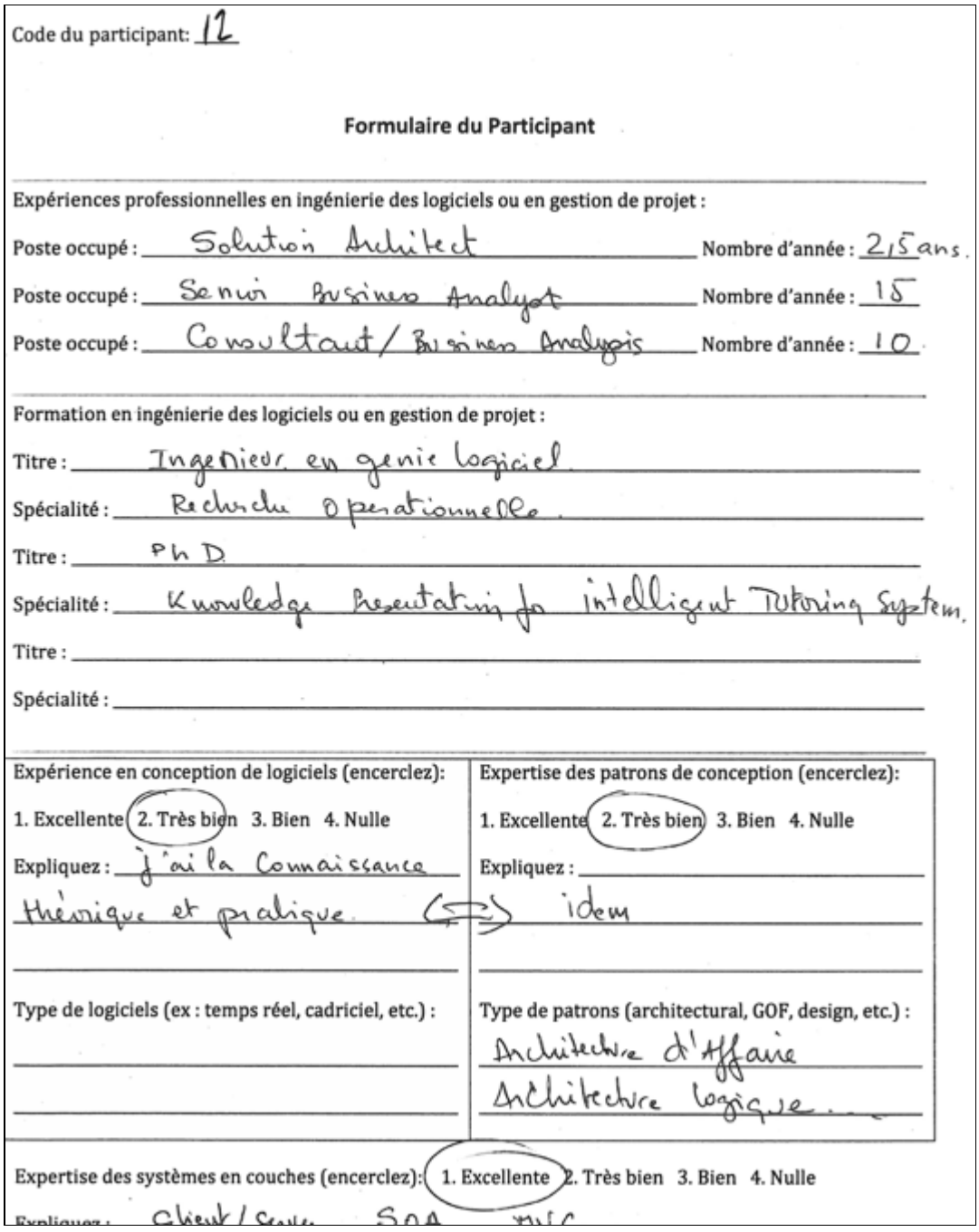
- The Component class will define the load, init, execute, and exit template methods [32] that constitute a standard execution interface. This class will also define the doLoad, doInit, doExecute, and doExit primitive methods that user classes can redefine to be called at particular moments by the execution package.
- Every cockpit's component will be controlled according to template methods where connections are standardized.
- Any dispatcher that knows the Component interface may be responsible for executing the simulation. The Component interface will ease future changes in simulated environments.
- The load method will be called on every component the first time the component is called. This method will load the composition file associated to the component and instantiate children of this component if any.
- The factory method will instantiate an object specified by its class name. The init method will be called on every component the first time the component is called; just after the load method is executed.
- This operation will be used to prepare the application for the beginning of the simulation. The load and init methods will call the parser to process the XML configuration files.
- When initialization is done, the execute method will be called on every component if the component is linked to the dispatcher.
- The exit method will be called on every component at the component destruction. It shall be used to flush buffers on disk or any other operation normally done at application's closure.
- The XML file will contain values used by the doInit method of the component to set its properties at initialization time. For a SCS, these files will be packaged together with the DLL of the system.
- Software designers will use the Unified Modeling Language (UML) to draw graphical diagrams, which is a de-facto modeling language with wide acceptance and tool support for object-oriented software development. It supports multiple

viewpoints, semi-formal semantics, and a formal language for fixing constraints on design elements.

- The view System Foundation Classes will present the framework as a set of classes offering services to ease the creation and maintenance of software components. It will conform to the module viewpoint, which requires breaking up the system into a set of decomposable modules.
- The view System Execution Package will show the relationship between the components of the framework, the simulation dispatcher, and specific SCSs. The dispatcher will use the Component interface to execute the simulation logic of a system. This view will conform to the component-and-connector viewpoint, which requires breaking up the system into a set of executable units.
- The view System Packaging will show the system's packaging that should be made so that generic system packages can be used for customizing various application of this system. This view conforms to the allocation viewpoint and documents the relationships between the framework and its environment.

APPENDIX III

INPUTS AND OUTPUTS OF THE EXPERIMENT WITH HUMAN PARTICIPANTS



Code du participant: 12

Formulaire du Participant

Expériences professionnelles en ingénierie des logiciels ou en gestion de projet :

Poste occupé : Solution Architect Nombre d'année : 2,5 ans.

Poste occupé : Senior Business Analyst Nombre d'année : 15

Poste occupé : Consultant / Business Analysis Nombre d'année : 10

Formation en ingénierie des logiciels ou en gestion de projet :

Titre : Ingenieur en genie logiciel

Spécialité : Recherche Operationnelle.

Titre : Ph D

Spécialité : Knowledge Presentation for intelligent Tutoring System.

Titre :

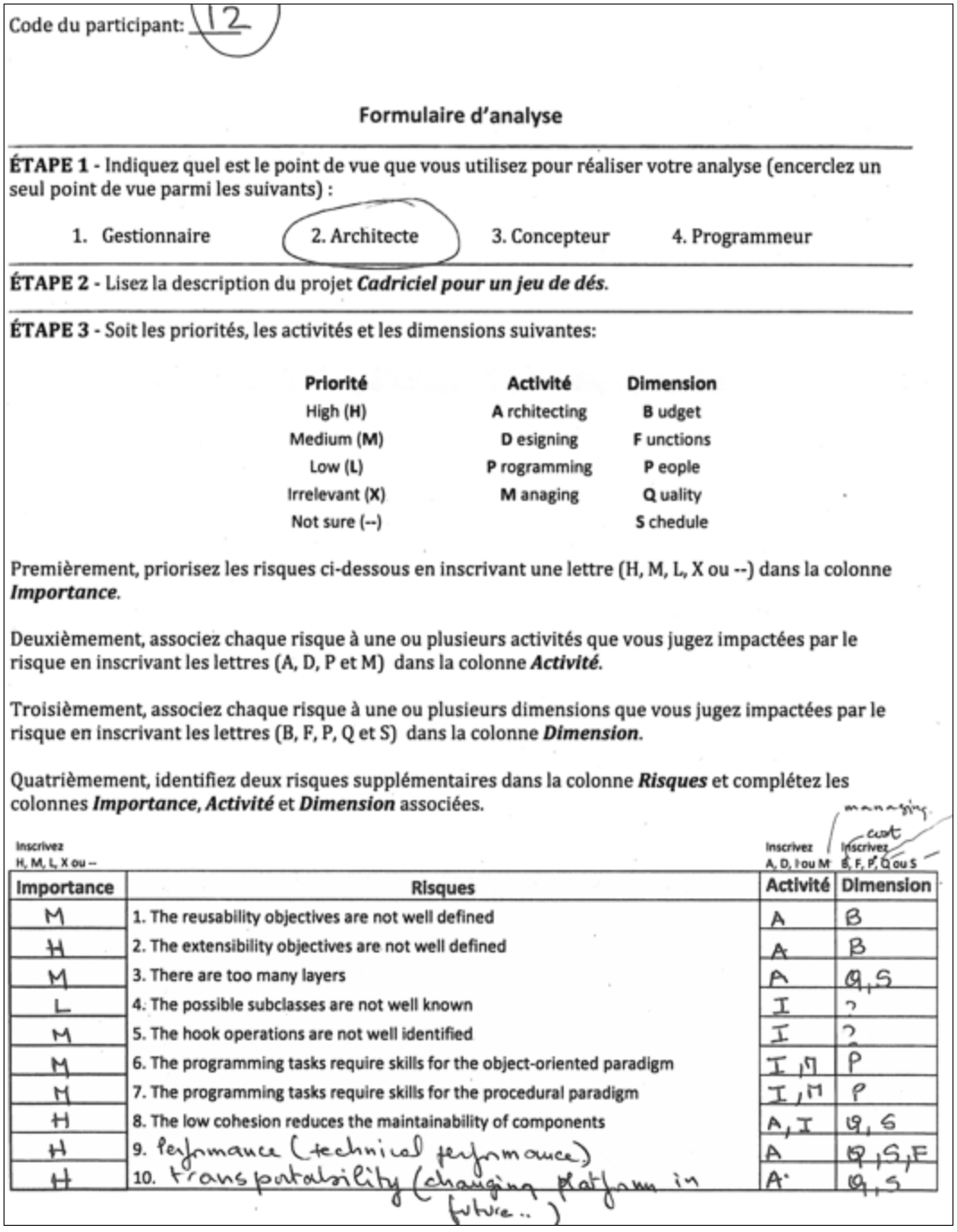
Spécialité :

Expérience en conception de logiciels (encerclez):	Expertise des patrons de conception (encerclez):
1. Excellente (2. Très bien) 3. Bien 4. Nulle	1. Excellente (2. Très bien) 3. Bien 4. Nulle
Expliquez : j'ai la Connaissance théorique et pratique ⇔ idem	Expliquez :
Type de logiciels (ex : temps réel, cadriciel, etc.) :	Type de patrons (architectural, GOF, design, etc.) : Architecture d'Affaire, Architecture logique

Expertise des systèmes en couches (encerclez): (1. Excellente) 2. Très bien 3. Bien 4. Nulle

Expliquez : Client / Serv. SOA MVC

Figure A III.1 Participant form of an architect



Code du participant: 12

Formulaire d'analyse

ÉTAPE 1 - Indiquez quel est le point de vue que vous utilisez pour réaliser votre analyse (encerclez un seul point de vue parmi les suivants) :

1. Gestionnaire 2. Architecte 3. Concepteur 4. Programmeur

ÉTAPE 2 - Lisez la description du projet ***Cadriciel pour un jeu de dés***.

ÉTAPE 3 - Soit les priorités, les activités et les dimensions suivantes:

Priorité	Activité	Dimension
High (**H**)	**A** rchitecting	**B** udget
Medium (**M**)	**D** esigning	**F** unctions
Low (**L**)	**P** rogramming	**P** eople
Irrelevant (**X**)	**M** anaging	**Q** uality
Not sure (--)		**S** chedule

Premièrement, priorisez les risques ci-dessous en inscrivant une lettre (H, M, L, X ou --) dans la colonne ***Importance***.

Deuxièmement, associez chaque risque à une ou plusieurs activités que vous jugez impactées par le risque en inscrivant les lettres (A, D, P et M) dans la colonne ***Activité***.

Troisièmement, associez chaque risque à une ou plusieurs dimensions que vous jugez impactées par le risque en inscrivant les lettres (B, F, P, Q et S) dans la colonne ***Dimension***.

Quatrièmement, identifiez deux risques supplémentaires dans la colonne ***Risques*** et complétez les colonnes ***Importance***, ***Activité*** et ***Dimension*** associées.

Inscrivez H, M, L, X ou --		Inscrivez A, D, P ou M	Inscrivez B, F, P, Q ou S
Importance	**Risques**	**Activité**	**Dimension**
M	1. The reusability objectives are not well defined	A	B
H	2. The extensibility objectives are not well defined	A	B
M	3. There are too many layers	A	Q, S
L	4. The possible subclasses are not well known	I	?
M	5. The hook operations are not well identified	I	?
M	6. The programming tasks require skills for the object-oriented paradigm	I, M	P
M	7. The programming tasks require skills for the procedural paradigm	I, M	P
H	8. The low cohesion reduces the maintainability of components	A, I	Q, S
H	9. Performance (technical performance)	A	Q, S, F
H	10. transportability (changing platform in future..)	A	Q, S

Figure A III.2 Analysis form of an architect (page 1)

Code du participant: 12

Cinquièmement, expliquez brièvement l'importance, les activités et les dimensions que vous avez associez à chaque risque.

# risque	Courte explication
1	Extensibility : si une Architecture n'est pas Extensible/ evolutive, ceci risque de la "costly" qd on veut ajouter de la fonctionalité
2	une Architecture non performante peut ⟨ être rejeté / costly / quality (impact)
3	Transportability is another risk if not considered combranging a platform in the future may be costly ...
4	low cohesion limits extensibility and can be costly / reduce Quality ... ...
5	All Medium risks are more related to
6	Management and People. Up to the Organisation to invest on training / edu cation / mentoring ...
7	
8	Low risks (subclass not known: this usually have a low impact on Architecture
9	
10	

Figure A III.3 Analysis form of an architect (page 2)

Code du participant: 14

Formulaire d'analyse

ÉTAPE 1 - Indiquez quel est le point de vue que vous utilisez pour réaliser votre analyse (encerclez un seul point de vue parmi les suivants) :

1. Gestionnaire 2. Architecte 3. Concepteur 4. Programmeur

ÉTAPE 2 - Lisez la description du projet ***Cadriciel pour un jeu de dés***.

ÉTAPE 3 - Soit les priorités, les activités et les dimensions suivantes:

Priorité	Activité	Dimension
High (H)	Architecting	Budget
Medium (M)	Designing	Functions
Low (L)	Programming	People
Irrelevant (X)	Managing	Quality
Not sure (--)		Schedule

Premièrement, priorisez les risques ci-dessous en inscrivant une lettre (H, M, L, X ou --) dans la colonne ***Importance***.

Deuxièmement, associez chaque risque à une ou plusieurs activités que vous jugez impactées par le risque en inscrivant les lettres (A, D, P et M) dans la colonne ***Activité***.

Troisièmement, associez chaque risque à une ou plusieurs dimensions que vous jugez impactées par le risque en inscrivant les lettres (B, F, P, Q et S) dans la colonne ***Dimension***.

Quatrièmement, identifiez deux risques supplémentaires dans la colonne ***Risques*** et complétez les colonnes ***Importance***, ***Activité*** et ***Dimension*** associées.

Inscrivez H, M, L, X ou -- Importance	Risques	Inscrivez A, D, I ou M Activité	Inscrivez B, F, P, Q ou S Dimension
M	1. The reusability objectives are not well defined	A M	F Q
H	2. The extensibility objectives are not well defined	A D M	F Q
L	3. There are too many layers	A	P F Q
M	4. The possible subclasses are not well known	A D	F Q
H	5. The hook operations are not well identified	D A	F Q
M	6. The programming tasks require skills for the object-oriented paradigm	M P A D	B P S Q
L	7. The programming tasks require skills for the procedural paradigm	M P A D	B P S Q
L	8. The low cohesion reduces the maintainability of components	A D	F B S Q
M	9. Trop de restriction sur l'utilisation de patrons	A D M	P F Q
H	10. Trop d'éléments doivent être réimplémenté	M A D	P Q

Figure A III.4 Analysis form of a designer (page 1)

Code du participant: 14

Cinquièmement, expliquez brièvement l'importance, les activités et les dimensions que vous avez associez à chaque risque.

# risque	Courte explication
1	Les objectifs doivent être bien identifiés
2	les objectifs " " " "
3	le framework doit resté simple mais le risque n'est pas élevé
4	la documentation doit être bien construite et claire, l'architecture et le design bien préparé.
5	Les Fonctions Hook sont ce sur quoi les implémenteurs vont s'accrocher et doivent donc être bien identifié
6	La programmation OO est nécessaire mais comme le design est simple on classe medium.
7	Peu d'algorithmes procédurales, à première vue, ont besoin d'être utilisé.
8	Comme le framework reste simple il est peu probable que la cohésion soit basse.
9	Plusieurs restrictions sont posés à la fin, elle peuvent affecté en mal le framework et la façon de travaill
10	Si un grande partie du framework doit être modifié pour un certain jeu ceci le rend inutile

Figure A III.5 Analysis form of a designer (page 2)

Code du participant: 15

Formulaire d'analyse

ÉTAPE 1 - Indiquez quel est le point de vue que vous utilisez pour réaliser votre analyse (encerclez un seul point de vue parmi les suivants) :

1. Gestionnaire 2. Architecte 3. Concepteur 4. Programmeur

ÉTAPE 2 - Lisez la description du projet ***Cadriciel pour un jeu de dés.***

ÉTAPE 3 - Soit les priorités, les activités et les dimensions suivantes:

Priorité	Activité	Dimension
High (H)	A rchitecting	B udget
Medium (M)	D esigning	F unctions
Low (L)	P rogramming	P eople
Irrelevant (X)	M anaging	Q uality
Not sure (--)		S chedule

Premièrement, priorisez les risques ci-dessous en inscrivant une lettre (H, M, L, X ou --) dans la colonne ***Importance***.

Deuxièmement, associez chaque risque à une ou plusieurs activités que vous jugez impactées par le risque en inscrivant les lettres (A, D, P et M) dans la colonne ***Activité***.

Troisièmement, associez chaque risque à une ou plusieurs dimensions que vous jugez impactées par le risque en inscrivant les lettres (B, F, P, Q et S) dans la colonne ***Dimension***.

Quatrièmement, identifiez deux risques supplémentaires dans la colonne ***Risques*** et complétez les colonnes ***Importance***, ***Activité*** et ***Dimension*** associées.

Inscrivez H, M, L, X ou --		Inscrivez A, D, I ou M	Inscrivez B, F, P, Q ou S
Importance	**Risques**	**Activité**	**Dimension**
H	1. The reusability objectives are not well defined	ADPM	BFPQS
H	2. The extensibility objectives are not well defined	ADPM	BFPQS
L	3. There are too many layers	A	F
L	4. The possible subclasses are not well known	D	F
H	5. The hook operations are not well identified	AD	BFPQS
L	6. The programming tasks require skills for the object-oriented paradigm	P	P
X	7. The programming tasks require skills for the procedural paradigm	P	P
M	8. The low cohesion reduces the maintainability of components	A	F Q6
H	9. The market (target users) is not well defined.	AD	F
H	10. Framework too complicated or not easily customizable	M	B

Figure A III.6 Analysis form of a programmer (page 1)

Code du participant: 15

Cinquièmement, expliquez brièvement l'importance, les activités et les dimensions que vous avez associez à chaque risque.

# risque	Courte explication
1	Aspect critique qui risque de faire échouer le projet, et donc affecte directement ou indirectement toutes les disciplines et toutes les dimensions
2	" "
3	Il y a juste 3... donc c'est pas beaucoup, pas assez pour avoir un impact important.
4	Pas un réel problème, en tenant que le framework scope soit mieux défini. Les subClass vont utiliser les méthode de la classe parent
5	Impossible de compléter l'architecture sans en définir les nodes.
6	"En 2013, si ton programmeur ne comprend pas le paradigme OOP, mieux vaux en trouver un autre."
7	Le code est fait en OOP, donc ça ne s'applique pas.
8	Ça peut affecter l'horaire dans le sens où les risques à pour va prendre plus de temps.
9	Il faut d'abord faire une étude de marché et identifier en détails les besoins des client potentiel.
10	C'est ce qui arrive à beaucoup de frameworks, il faut que Thor réussis à appliquer facilement le framework générique à un besoin spécifique

Figure A III.7 Analysis form of a programmer (page 2)

Code du participant: 13

Formulaire d'analyse

ÉTAPE 1 - Indiquez quel est le point de vue que vous utilisez pour réaliser votre analyse (encerclez un seul point de vue parmi les suivants) :

(1.) Gestionnaire 2. Architecte 3. Concepteur 4. Programmeur

ÉTAPE 2 - Lisez la description du projet ***Cadriciel pour un jeu de dés***.

ÉTAPE 3 - Soit les priorités, les activités et les dimensions suivantes:

Priorité	Activité	Dimension
High (**H**)	**A** rchitecting	**B** udget
Medium (**M**)	**D** esigning	**F** unctions
Low (**L**)	**P** rogramming	**P** eople
Irrelevant (**X**)	**M** anaging	**Q** uality
Not sure (--)		**S** chedule

Premièrement, priorisez les risques ci-dessous en inscrivant une lettre (H, M, L, X ou --) dans la colonne ***Importance***.

Deuxièmement, associez chaque risque à une ou plusieurs activités que vous jugez impactées par le risque en inscrivant les lettres (A, D, P et M) dans la colonne ***Activité***.

Troisièmement, associez chaque risque à une ou plusieurs dimensions que vous jugez impactées par le risque en inscrivant les lettres (B, F, P, Q et S) dans la colonne ***Dimension***.

Quatrièmement, identifiez deux risques supplémentaires dans la colonne ***Risques*** et complétez les colonnes ***Importance***, ***Activité*** et ***Dimension*** associées.

Inscrivez H, M, L, X ou --		Inscrivez A, D, I ou M	Inscrivez B, F, P, Q ou S
Importance	**Risques**	**Activité**	**Dimension**
H	1. The reusability objectives are not well defined	A, D	B, P
M	2. The extensibility objectives are not well defined	A, D	F, P
M	3. There are too many layers	I, M	Q
L	4. The possible subclasses are not well known	I,	Q, S
M	5. The hook operations are not well identified	D, I	F, Q
H	6. The programming tasks require skills for the object-oriented paradigm	A, D, I	P, S
H	7. The programming tasks require skills for the procedural paradigm	A, D, I	B, P, S
M	8. The low cohesion reduces the maintainability of components	I, M	B, S
	9.		
	10.		

Figure A III.8 Analysis form of a manager (page 1)

Code du participant: 13

Cinquièmement, expliquez brièvement l'importance, les activités et les dimensions que vous avez associez à chaque risque.

# risque	Courte explication
1	LA RÉ-UTILISATION (OU SON ABSENCE) A UN IMPACT MAJEUR SUR L'EFFORT NÉCESSAIRE ASSOCIÉ AUX DIFFÉRENTES IMPLANTATIONS
2	L'EXTENSIBILITÉ, BIEN QU'UTILE, EST DIFFICILEMENT RÉALISABLE À MOINS D'AVOIR DES LIGNES DIRECTRICES CLAIRES.
3	UN TROP GRAND NOMBRE DE COUCHES ALOURDIT LA MAINTENANCE.
4	
5	LA MÉCONNAISSANCE DES "HOOKS OPÉRATIONS" AFFECTE L'EXPLOITATION EFFICACE DU LOGICIEL
6	LES REGROUPEMENTS DES TÂCHES DE PROGRAMMATION ONT UN IMPACT DIRECT SUR LES PROGRAMMEURS ET LE TEMPS NÉCESSAIRE AU PROJET
7	VOIR 6. LE LANGAGE PROCÉDURAL, EN GÉNÉRAL, SE PRÊTE MAL À LA RÉ-UTILISATION DE LOGICIELS.
8	L'IMPACT SE FAIT SENTIR DAVANTAGE POUR LES IMPLÉMENTATIONS FUTURES.
9	
10	

Figure A III.9 Analysis form of a manager (page 2)

# Participant	Point of view	Background	Exp. Conc.	Exp. Patrons	Exp. Layered	Profile
1	Gestion.	Inf. app.	NU	B	B	1
2	Gestion.	Inf. app.	EX	TB	TB	3
5	Gestion.	Inf. app.	TB	B	X	2
8	Gestion.	Inf. app.	EX	B	B	3
9	Gestion.	Inf. app.	B	NU	B	1
11	Gestion.	TI	B	NU	B	1
13	Gestion.	Ing. Log.	B	B	B	2
18	Gestion.	Inf. app.	TB	B	TB	1
3	Architecte	TI	EX	TB	EX	1
16	Architecte	Inf. app.	EX	TB	TB	3
12	Architecte	Ing. Log.	TB	TB	EX	2
10	Concep.	Ing. Log.	EX	B	EX	3
14	Concep.	Ing. Log.	EX	TB	EX	1
17	Concep.	Inf. app.	B	NU	B	1
19	Concep.	Ing. Log.	B	B	B	1
6	Prog.	Ing. Log.	B	B	X	1
7	Prog.	Inf. app.	B	B	TB	2
15	Prog.	Ing. Log.	TB	TB	TB	2
4	Prog.	Inf. app.	B	B	TB	3
20	Prog.	Ing. Log.	EX	TB	NU	3

Legend : EX – Excellent, TB – Tres bien, B – Bien, NU - Nulle

Figure A III.10 Data collected using the participant form

	Issue 1			Issue 2			Issue 3			Issue 4			Issue 5		
# Part	Imp	Act	Dim	Imp	Act	Dim	Imp	Act	Dim	Imp	Act	Dim	Imp	Act	Dim
1	L	AD	BFPQS	H	AD	BFPQS	L	A	QS	M	AD	FQS	M	D	F
2	H	AP	BS	L	AD	F	H	ADP	BS	M	PM	BPQS	M	PA	QF
5	M	AD	BPQ	H	ADP	BFPQ	M	ADI	BFPQ	M	ADP	BFPQ	M	ADP	BFPQS
8	H	AP	BFQS	H	ADM	BFQS	L	AP	BF	H	AP	BFQS	M	D	F
9	H	A	F	H	D	Q	L	A	F	M	D	F	M	M	P
11	M	M	B	M	M	B	L	A	F	L	C	Q	M	C	F
13	H	AD	BP	M	AD	FP	M	IM	Q	L	I	QS	M	DI	FQ
18	H	AD	F	M	D	F	X	X	X	X	X	X	X	P	PBF
3	M	AD	BSQ	H	ADP	BSQ	X	X	X	H	AD	BSQ	M	ADP	BSQ
16	H	A	F	H	A	F	L	ADP	F	H	AP	FQ	M	D	F
12	M	A	B	H	A	B	M	A	QS	L	I	X	M	I	X
10	H	D	QS	M	A	F	H	D	Q	M	D	F	X	I	F
14	M	AM	FQ	H	ADM	FQ	L	A	PFQ	M	AD	FQ	H	DA	FQ
17	H	ADP	QBF	M	PM	BFQS	L	AP	FQ	X	DP	FQ	M	PD	QBF
19	M	A	B	H	A	B	M	A	PQ	X	X	X	X	X	X
6	H	A	F	H	DPA	BS	L	PM	FQ	H	DA	BQ	M	D	F
7	H	ADP	BPQ	H	DP	BPS	X	DP	BPQ	X	P	BFPQ	M	ADP	BSP
15	H	ADPM	BFPQS	H	ADPM	BFPQS	L	A	F	L	D	F	H	AD	BFPQS
4	M	AD	FPB	M	AD	FPS	L	P	FQ	M	DP	BS	M	P	PQ
20	H	D	F	H	D	F	X	A	Q	H	D	F	X	A	Q

Figure A III.11 Data collected using the analysis form (part 1)

	Issue 6			Issue 7			Issue 8			Issue 9			Issue 10		
# Part	Imp	Act	Dim	Imp	Act	Dim	Imp	Act	Dim	Imp	Act	Dim	Imp	Act	Dim
1	H	PM	P	H	PM	P	M	A	BQS	L	PM	PS	X	X	X
2	H	ADP	BPS	L	P	F	M	ADP	BQS	H	AM	BS	H	ADP	BPQS
5	H	MAD	BPS	L	AD	X	H	MADP	BPQS	H	MDP	BFPQS	X	X	X
8	M	P	P	X	X	X	X	X	X	X	DAP	BFQS	X	X	X
9	H	P	P	X	P	P	M	D	Q	X	X	X	X	X	X
11	H	P	P	H	P	P	H	C	Q	X	X	X	X	X	X
13	H	ADI	PS	H	ADI	BPS	M	IM	BS	X	X	X	X	X	X
18	H	P	PBQS	X	X	X	L	AD	FQ	H	PM	BSQ	X	X	X
3	M	I	P	X	X	X	X	X	X	H	A	BS	X	X	X
16	M	P	BQP	L	P	BQP	H	DA	BQ	M	AD	FQ	X	X	X
12	M	IM	P	M	IM	P	H	AI	QS	H	A	QSF	H	A	QS
10	M	M	P	M	M	P	H	D	Q	X	X	X	X	X	X
14	M	MPAD	BPSQ	L	MPAD	BPSQ	L	AD	FBSQ	M	ADM	PFQ	H	MAD	PQ
17	L	P	QPF	L	P	QPF	M	AD	QBS	H	PDM	BQS	X	X	X
19	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
6	M	M	QS	X	X	X	M	AD	B	X	X	X	X	X	X
7	L	PM	BPS	L	PM	BPS	X	ADPM	BFPS	X	X	X	X	X	X
15	L	P	P	X	P	P	M	A	FQS	H	AD	F	H	M	B
4	H	DP	BPQ	X	X	X	H	AD	BPQ	X	X	X	X	X	X
20	H	P	P	X	P	P	X	D	F	X	X	X	X	X	X

Figure A III.12 Data collected using the analysis form (part 2)

APPENDIX IV

THE SSMS OF THE MODIFIABILITY TACTICS

Table A III.1 to Table A III.14 present the SSMs of the modifiability tactics described in [Bass03].

Table A IV.1 Software design artifacts (SDA) of the tactic "Maintain semantic coherence"

SDA	Type	Description
Why		
Go1	Goal	Control the time to implement, test, and deploy changes
Go2	Goal	Control the cost to implement, test, and deploy changes
Dc1	Concern	Localize changes
Ra4	Rationale	Ensure that anticipated changes in a module are semantically coherent
Ra5	Rationale	Assign responsibilities in a module that have semantic coherence
Ra6	Rationale	Ensure that responsibilities work together without excessive reliance on other modules
What		
Pr1	Property	Modifiability
Im2	Measure	Coupling
Im3	Measure	Cohesion
Im1	Measure	Number of modules that require changing to implement a change
Which		
Ta2	Tactic	Maintain semantic coherence

Table A IV.2 Software design artifacts (SDA) of the tactic “Abstract common services”

SDA	Type	Description
Why		
Go1	Goal	Control the time to implement, test, and deploy changes
Go2	Goal	Control the cost to implement, test, and deploy changes
Dc1	Concern	Localize changes
Dc2	Concern	Prevention of ripple effect
Ra1	Rationale	Reduce the number of modules directly affected by a change
Ra2	Rationale	Restrict changes to a small set of modules
Ra3	Rationale	Limit anticipated changes in scope
Ra4	Rationale	Provide common services through specialized modules
What		
Pr1	Property	Modifiability
Pr2	Property	Reusability
Im1	Measure	Number of modules directly affected by a change
Im1	Measure	Number of modules that require changing to implement a change
Which		
Ta1	Tactic	Abstract common services
Sf1	Fragment	Application framework
Sf2	Fragment	Middleware software

Table A IV.3 Software design artifacts (SDA) of the tactic "Anticipate expected changes"

SDA	Type	Description
Why		
Go1	Goal	Control the time to implement, test, and deploy changes
Go2	Goal	Control the cost to implement, test, and deploy changes
Dc1	Concern	Localize changes
Ra1	Rationale	Limit the number of modules directly affected by a change
Ra2	Rationale	Restrict changes to a small set of modules
Ra3	Rationale	Assign responsibilities in order to minimize the effects of the changes
What		
Pr1	Property	Modifiability
Im1	Measure	Number of modules directly affected by a change
Im1	Measure	Number of modules that require changing to implement a change
Which		
Ta1	Tactic	Anticipate expected changes

Table A IV.4 Software design artifacts (SDA) of the tactic “Generalize the module”

SDA	Type	Description
Why		
Go1	Goal	Control the time to implement, test, and deploy changes
Go2	Goal	Control the cost to implement, test, and deploy changes
Dc1	Concern	Localize changes
Ra2	Rationale	Allow a module to compute a range of functions based on input
Ra3	Rationale	Define an input language for a module
Ra4	Rationale	Ensure that changes can be made by adjusting the input language
What		
Pr1	Property	Modifiability
Im1	Measure	Number of modules that require changing to implement a change
Which		
Ta2	Tactic	Generalize the module
Fr1	Fragment	Interpreter
St1	Structure	Module of constants input parameters

Table A IV.5 Software design artifacts (SDA) of the tactic "Limit possible options"

SDA	Type	Description
Why		
Go1	Goal	Control the time to implement, test, and deploy changes
Go2	Goal	Control the cost to implement, test, and deploy changes
Dc1	Concern	Localize changes
Ra2	Rationale	Restrict options in order to minimize the effects of the changes
What		
Pr1	Property	Modifiability
Im1	Measure	Number of modules that require changing to implement a change
Which		
Ta2	Tactic	Limit possible options

Table A IV.6 Software design artifacts (SDA) of the tactic “Hide information”

SDA	Type	Description
Why		
Go1	Goal	Control the time to implement, test, and deploy changes
Go2	Goal	Control the cost to implement, test, and deploy changes
Dc1	Concern	Localize changes
Dc1	Concern	Prevent ripple effects
Ra4	Rationale	Reduce the necessity of making changes to modules not directly affected by a modification
Ra5	Rationale	Assign responsibilities for an entity into smaller pieces
Ra6	Rationale	Make some information private, and other information public
Ra7	Rationale	Make public responsibilities available through specified interface
What		
Pr1	Property	Modifiability
Im1	Measure	Number of modules that require changing to implement a change
Which		
Ta2	Tactic	Hide information

Table A IV.7 Software design artifacts (SDA) of the tactic "Maintain existing interface"

SDA	Type	Description
Why		
Go1	Goal	Control the time to implement, test, and deploy changes
Go2	Goal	Control the cost to implement, test, and deploy changes
Dc1	Concern	Localize changes
Dc2	Concern	Prevention of ripple effect
Ra1	Rationale	Separate the interface from the implementation
Ra2	Rationale	Create public abstract interface that mask variations
Ra3	Rationale	Embody variations within the existing responsibilities
Ra4	Rationale	Embody variations by replacing one implementation of a module with another
When		
Sr1	Risk	Difficult to mask changes to the meaning of data and services
Sr2	Risk	Difficult to mask dependencies on quality of data or quality of services
Sr3	Risk	Difficult to mask dependencies on resource usage and resource ownership
What		
Pr1	Property	Modifiability
Im1	Measure	Number of modules that require changing to implement a change
Which		
Ta1	Tactic	Maintain existing interface
St1	Structure	Public interface
St2	Opera.	Declare abstract signature

Table A IV.8 Software design artifacts (SDA) of the tactic “Restrict communication paths”

SDA	Type	Description
Why		
Go1	Goal	Control the time to implement, test, and deploy changes
Go2	Goal	Control the cost to implement, test, and deploy changes
Dc1	Concern	Localize changes
Dc2	Concern	Prevention of ripple effect
Ra1	Rationale	Restrict the modules with which a given module shares data
What		
Pr1	Property	Modifiability
Im1	Measure	Number of modules that require changing to implement a change
Which		
Ta1	Tactic	Restrict communication paths
St1	Structure	Module that consumes data
St1	Structure	Module that produces data
Im1	Measure	Number of modules that consume data produced by the given module
Im1	Measure	Number of modules that produce data consumed by the given module

Table A IV.9 Software design artifacts (SDA) of the tactic “Use an intermediary”

SDA	Type	Description
Why		
Go1	Goal	Control the time to implement, test, and deploy changes
Go2	Goal	Control the cost to implement, test, and deploy changes
Dc1	Concern	Localize changes
Dc2	Concern	Prevention of ripple effect
Ra1	Rationale	Insert an intermediary that manages activities associated with a dependency
Ra2	Rationale	Convert the data syntax produced by a module into that assumed by another
Ra3	Rationale	Convert the syntax of a service from one form into another
Ra4	Rationale	Mask changes in the identity of an interface
Ra5	Rationale	Enable the location of a module to change without affecting another module
Ra6	Rationale	Guarantee the satisfaction of all requests within certain constraints
Ra7	Rationale	Create instances as needed by actions of an intermediary
When		
Sr1	Risk	An intermediary cannot compensate for semantic changes
What		
Pr1	Property	Modifiability
Im1	Measure	Number of modules that require changing to implement a change
Which		
Ta1	Tactic	Use an intermediary
Fr1	Fragment	Blackboard repository
Fr2	Fragment	Passive repository
Fr3	Fragment	Broker
Fr4	Fragment	Name server
Fr5	Fragment	Façade

Fr6	Fragment	Bridge
Fr7	Fragment	Mediator
Fr8	Fragment	Strategy
Fr9	Fragment	Proxy
Fr10	Fragment	Factory
Fr11	Fragment	Resource manager

Table A IV.10 Software design artifacts (SDA) of the tactic “Runtime registration”

SDA	Type	Description
Why		
Go1	Goal	Control the time to implement, test, and deploy changes
Go2	Goal	Control the cost to implement, test, and deploy changes
Dc1	Concern	Defer binding time
Ra1	Rationale	Support plug-and-play operation
Ra2	Rationale	Do registration at runtime
Ra3	Rationale	Do registration at loadtime
When		
Sr1	Risk	Additional overhead to manage the registration
What		
Pr1	Property	Modifiability
Which		
Ta1	Tactic	Runtime registration

Table A IV.11 Software design artifacts (SDA) of the tactic "Configuration files"

SDA	Type	Description
Why		
Go1	Goal	Control the time to implement, test, and deploy changes
Go2	Goal	Control the cost to implement, test, and deploy changes
Dc1	Concern	Defer binding time
Ra1	Rationale	Set parameters at startup
When		
Sr1	Risk	Additional overhead to manage the initialization
What		
Pr1	Property	Modifiability
Which		
Ta1	Tactic	Configuration files
Fr1	Fragment	XML configuration file

Table A IV.12 Software design artifacts (SDA) of the tactic "Polymorphism"

SDA	Type	Description
Why		
Go1	Goal	Control the time to implement, test, and deploy changes
Go2	Goal	Control the cost to implement, test, and deploy changes
Dc1	Concern	Defer binding time
Ra1	Rationale	Allow late binding of method calls
When		
Sr1	Risk	Additional overhead to manage the late binding
What		
Pr1	Property	Modifiability
Which		
Ta1	Tactic	Polymorphism

Table A IV.13 Software design artifacts (SDA) of the tactic “Component replacement”

SDA	Type	Description
Why		
Go1	Goal	Control the time to implement, test, and deploy changes
Go2	Goal	Control the cost to implement, test, and deploy changes
Dc1	Concern	Defer binding time
Ra1	Rationale	Allow loadtime binding
When		
Sr1	Risk	Additional overhead to manage the loadtime binding
What		
Pr1	Property	Modifiability
Which		
Ta1	Tactic	Component replacement

Table A IV.14 Software design artifacts (SDA) of the tactic “Adherence to defined protocols”

SDA	Type	Description
Why		
Go1	Goal	Control the time to implement, test, and deploy changes
Go2	Goal	Control the cost to implement, test, and deploy changes
Dc1	Concern	Defer binding time
Ra1	Rationale	Allow runtime binding of independent processes
When		
Sr1	Risk	Additional overhead to manage the runtime binding
What		
Pr1	Property	Modifiability
Which		

APPENDIX V

INPUTS AND OUTPUTS OF THE CASE STUDY IN WEB ENGINEERING

Work statement

The following web pages were required by the customer for the web site.

Figure A V.1 Web page 'Entry.html'

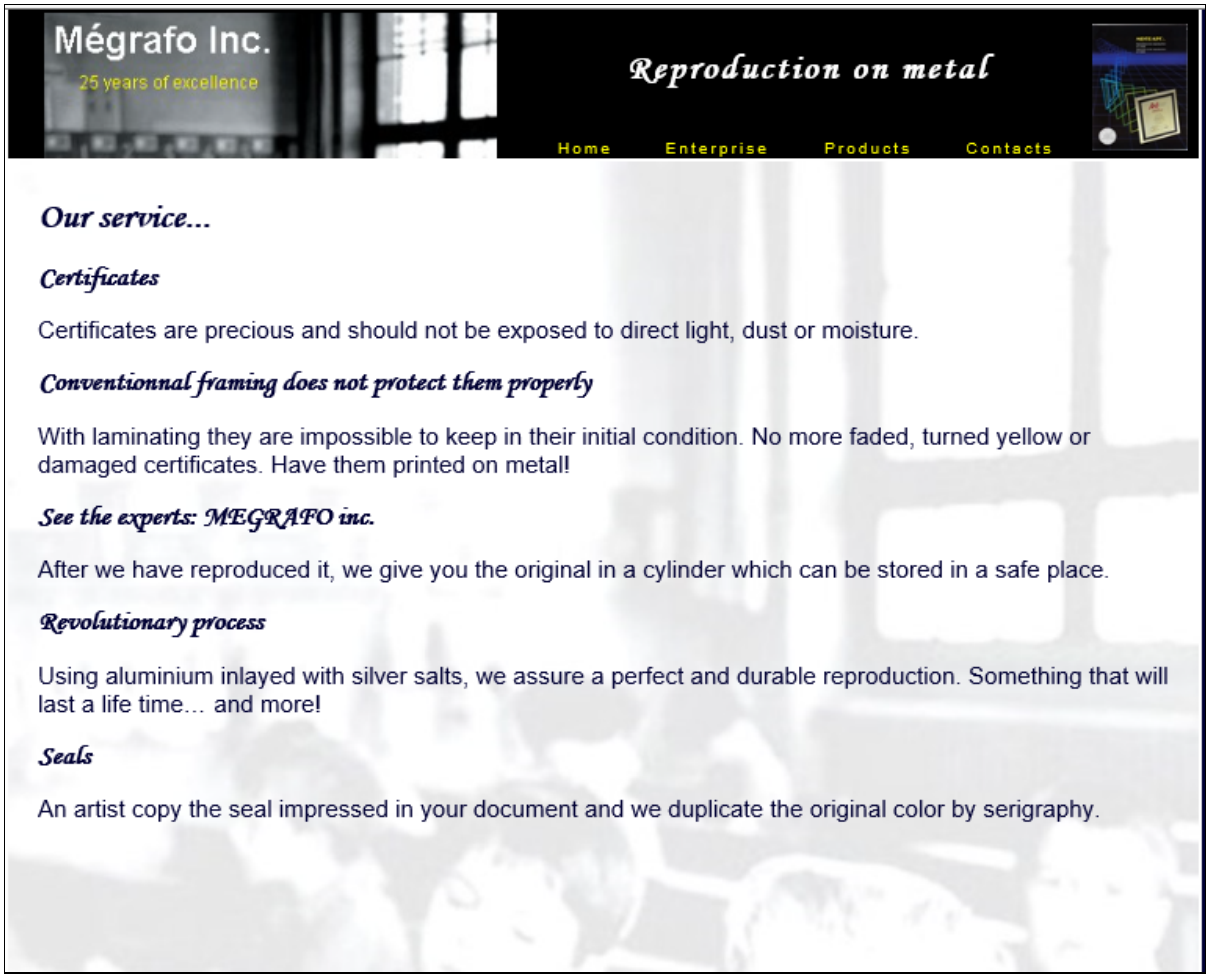


Figure A V.2 Web page ‘Home.html’

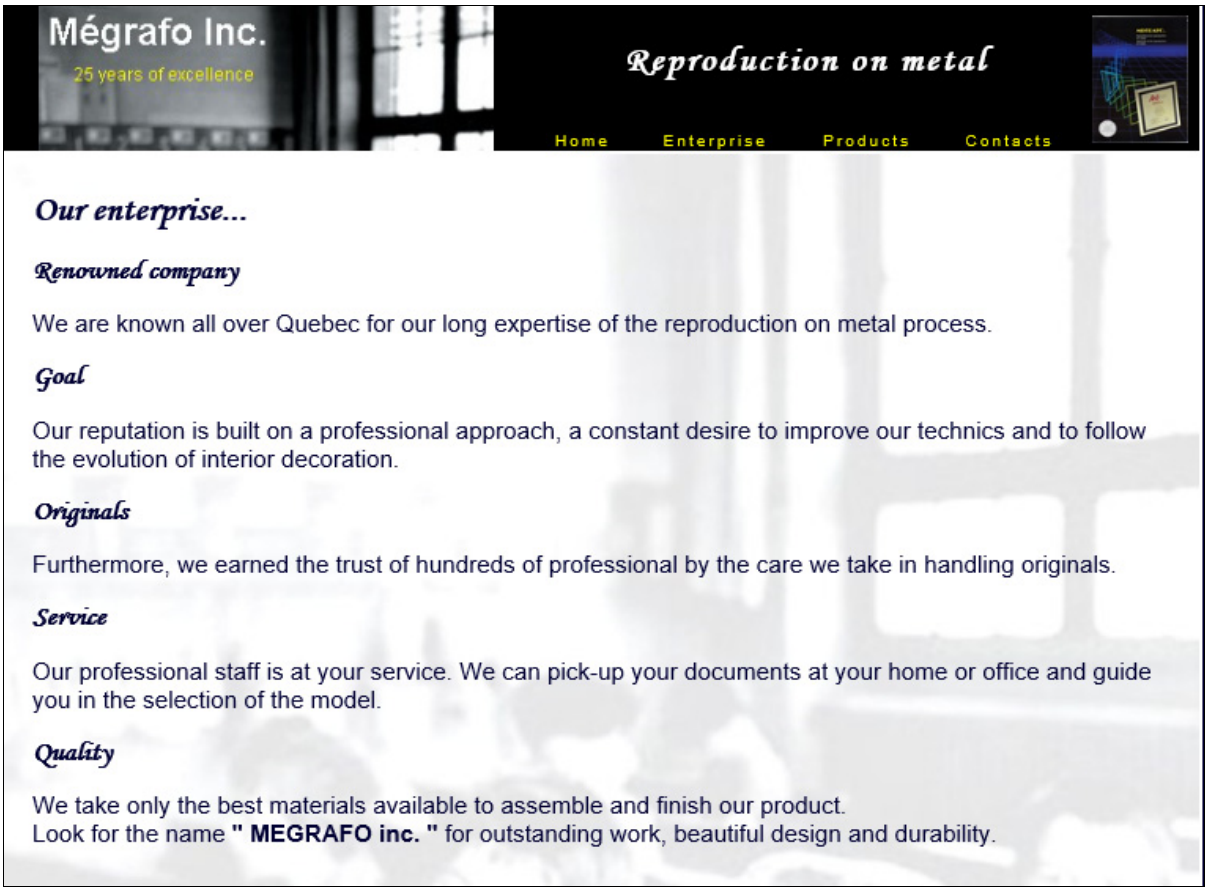


Figure A V.3 Web page 'Enterprise.html'

Figure A V.4 Web page 'Products.html'

Figure A V.5 Web page 'Contacts.html'

APPENDIX VI

INPUTS AND OUTPUTS OF THE EXPERIMENT WITH A HUMAN PARTICIPANT

Work statement for the development of the web site

“Votre tâche consiste à concevoir le site Web de la compagnie « Mégrafo Inc. ». Cette compagnie reproduit des diplômes sur du métal encadré. Utilisez les fichiers disponibles dans megrafo.zip.

Règle de conception

Vous devez créer votre site à l'aide du langage HTML. Vous devez remettre des pages valides (XHTML et CSS). Vous pouvez utiliser les validateurs du W3C pour vous assurer de la validité de vos pages. Vous devez déterminer le type de document le plus stricte possible pour chaque fichier.

Répertoire de travail

Créez le répertoire /megrafo dans lequel vous mettrez tous les fichiers html que vous utiliserez. Créez le répertoire /megrafo/images dans lequel vous mettrez toutes les images que vous utiliserez. Créez le répertoire /megrafo/css dans lequel vous mettrez toutes les feuilles de styles que vous créez.

Feuilles de style externes : styles_base.css et liens.css

Créez un fichier nommé styles_base.css. Indiquez le commentaire « Styles de base pour le site Web de Mégrafo. » dans le haut du fichier. Dans ce fichier, créez les styles suivants.

L'élément body a l'image de fond « rd1956_filigrane_l.jpg », cette dernière est fixée et ne se répète pas. Dans l'élément body, le texte est de couleur « #000033 », cette dernière est spécifiée dans la forme courte de CSS. La police par défaut du texte est « Monotype Corsiva », mais cette dernière n'est pas supportée par tous les navigateurs. Pour ces derniers, la police du texte est « sans-serif ». La taille du texte est de 12 points.

Le texte défini dans l'élément h1 a une taille de 18 points, dans l'élément h2 une taille de 14 points et dans l'élément h3 une taille de 12 points. Dans l'élément p, la police par défaut du texte est «Tashoma », mais cette dernière n'est pas supportée par tous les navigateurs. Pour ces derniers, la police du texte est « sans-serif ».

Créez un fichier nommé liens.css. Indiquez le commentaire « Styles des liens pour le site Web de Mégrafo. » dans le haut du fichier. Dans ce fichier, créez les styles suivants.

Le texte dans l'élément a n'est pas enrichi. Il est de couleur « #ffff00 », cette dernière est spécifiée dans la forme courte de CSS. La police par défaut du texte est «Helvetica », mais cette dernière n'est pas supportée par tous les navigateurs. Pour ces derniers, la police du texte est « sans-serif ». La taille du texte est de 8 points et les caractères sont espacés (ou le « crénage » est) de « 0.2em ». L'espace entre la zone de contenu et la bordure est de 10 pixels en haut et en bas, et de 0 pixel à gauche et à droite.

Le texte dans l'élément a:hover n'est pas enrichi. Il est de couleur « #ffffff », cette dernière est spécifiée dans la forme courte de CSS. La police par défaut du texte est «Tashoma », mais cette dernière n'est pas supportée par tous les navigateurs. Pour ces derniers, la police du texte est « sans-serif ». La taille du texte est de 8 points. L'espace entre la zone de contenu et la bordure est de 10 pixels en haut et en bas, et de 0 pixel à gauche et à droite. La bordure du bas a une largeur de 1 pixel, est solide et de couleur « #ffff00 ». Cette dernière règle souligne le texte d'un trait jaune lorsque le curseur passe au-dessus d'un lien.

Dans les fichiers nommés accueil.htm, entreprise.htm et contacts.htm, insérez un lien vers la feuille de style externe « styles_base.css ». Dans le fichier menu.htm, insérez un lien vers la feuille de style « liens.css ».

Styles internes

Dans le fichier nommé menu.htm, la couleur de fond de l'élément body est « #000000 », cette dernière est spécifiée dans la forme courte de CSS. Dans l'élément body, le texte est de couleur « #ffffff », cette dernière est spécifiée dans la forme courte de CSS. La police par défaut du texte est « Monotype Corsiva », mais cette dernière n'est pas supportée par tous les navigateurs. Pour ces derniers, la police du texte est « sans-serif ». La taille du texte est de 14 points et les caractères sont espacés (ou le « crénage » est) de 2 pixels. L'espace entre la zone de contenu et la bordure est de 0 pixel pour tous les côtés. Le texte défini dans l'élément h1 a une taille de 18 points et n'est pas indenté.

Dans le fichier nommé contacts.htm, les bordures en bas et à gauche de l'élément table ont une largeur de 2 pixels, sont solides et de couleur « #000033 », cette dernière est spécifiée dans la forme courte de CSS. Les autres bordures (en haut et à droite) ont une largeur de 0 pixel. L'espace entre la zone de contenu et la bordure est de 5 pixels pour tous les côtés. L'espace entre les bordures est également de 5 pixels pour tous les côtés.

Dans le fichier nommé marge.htm, la couleur de fond de l'élément body est « #000033 », cette dernière est spécifiée dans la forme courte de CSS. Dans l'élément body, le texte est de couleur « #ffffff », cette dernière est spécifiée dans la forme courte de CSS.

Dans le fichier nommé produits.htm, la couleur de fond de l'élément body est « #000033 », cette dernière est spécifiée dans la forme courte de CSS. La police par défaut du texte est « Monotype Corsiva », mais cette dernière n'est pas supportée par tous les navigateurs. Pour ces derniers, la police du texte est « sans-serif ». Le texte défini dans l'élément h1 a une taille de 18 points. Dans l'élément p, la police par défaut du texte est «Tashoma », mais cette dernière n'est pas supportée par tous les navigateurs. Pour ces derniers, la police du texte est « sans-serif ». Les images ont une largeur de 122 pixels, une hauteur de 107 pixels et ont une largeur de 0 pixel pour les bordures. Les images sont centrées dans la page à l'aide d'un élément de niveau bloc dont la classe se nomme « centre ».

Styles en ligne

Dans tous les fichiers, remplacez chaque attribut dédié à la présentation par une règle CSS valide. De plus, éliminez les attributs inutilisés. Par exemple, l'attribut border ="0" associé à l'élément table dans le fichier contacts.htm est inutile puisqu'une règle CSS créée précédemment produit le même effet.

Liste des propriétés utilisées

Les propriétés CSS utilisées pour ce travail pratique sont les suivantes :

background-attachment, background-color, background-image, background-repeat, border, border-bottom, color, font-family, font-size, height, letter-spacing, padding, text-align, text-decoration, text-indent, width

BIBLIOGRAPHY

[Abow93] G. Abowd, R. Allen, and D. Garlan. "Using Style to Understand Descriptions of Software Architecture." First ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 9-20, Los Angeles, CA, December 1993.

[Abow95] Abowd, G. D., Allen, R., Garlan, D., "Formalizing style to understand descriptions of software architecture", ACM Transactions on Software Engineering and Methodology, pp. 319–364, 1995.

[Abra01] Abran, A., Bourque, P., Dupuis, R., and W. Moore, J., "Guide to the Software Engineering Body of Knowledge", IEEE Press, Piscataway, NJ, USA, 2001.

[Abra06] Abran , A., Al-Qutaish, R. E., Cuadrado-Gallego, J., "Investigation of the Metrology Concepts in ISO 9126 on Software Product Quality Evaluation", In Proceedings of the 10th WSEAS International Conference on Computers (ICComp'2006), July 13-15, Athens, Greece, 2006, pp. 864-872. (ISBN: 960-8457-47-5)

[Alex77] Alexander, C., Ishikawa, S., Silverstein, M., "A pattern language: towns, buildings, construction", volume 2 of Center for Environmental Structure, Oxford Univ. Press, New York; ISBN 0-19-501919-9, 1977.

[Alle97] Robert Allen and David Garlan. A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 6, July 1997, 213-249.

[Ande01] Jonas Andersson and Pontus Johnson. 2001. Architectural Integration Styles for Large-Scale Enterprise Software Systems. In Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing (EDOC '01). IEEE Computer Society, Washington, DC, USA, 224-.

[Argo00] Argote, L., Ingram, P. 2000. "Knowledge transfer: A basis for competitive advantage in firms", Organizational Behavior and Human Decision Processes, 82, 150–169.

[Bach01] Bachmann, F., Clements, P., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., Special report, CMU/SEI-2001-SR-010, SEI Workshop on Software Architecture Representation, May 2001.

[Bach03a] Bachmann, F., Bass, L., Klein, M., “Deriving Architectural Tactics: A Step Methodical Architectural Design”, Technical Report, CMU/SEI-2003-TR-004, March 2003.

[Bach03b] Bachmann, F., Bass, L., Klein, M., “Preliminary Design of ArchE: A Software Architecture Design Assistant”, Technical Report, CMU/SEI-2003-TR-021, September 2003.

[Bach07] Bachmann, F., Bass, L., Bianco, P., “Software Architecture Design with ArchE”, Software Engineering Institute, March 2007.

[Barb95] Mario Barbacci, Thomas H., Longstaff Mark, H. Klein, Charles B. Weinstock, Technical report, CMU/SEI-95-TR-021 ESC-TR-95-021, December 1995.

[Bass03] Bass, L., Clements, P., Kazman, R., “Software architecture in practice”, 2nd, Addison Wesley, 2003.

[Bass05] Bass, L., Ivers, J., Klein, M., Merson, P., “Reasoning Frameworks”, Technical report, CMU/SEI-2005-TR-007, July 2005.

[Bern01] Tim Berners-Lee, James Hendler, and Ora Lassila, “The semantic web”, Scientific American, 35–43, May 2001.

[Bert05] A. Bertolino, A. Bucchiarone, S. Gnesi, H. Muccini, “An architecture-centric approach for producing quality systems”. In QoSA, pages 21–37, 2005.

[Bhat05] Sutirtha Bhattacharya and Dewayne E. Perry. 2005. Predicting Architectural Styles from Component Specifications. 5th Working IEEE/IFIP Conference on Software Architecture (WICSA '05). IEEE Computer Society, DC, USA, 231-232.

[Booc96] G. Booch, I. Jacobson, J. Rumbaugh, “Unified Modeling Language for Object Oriented Development”, Rational Software Corporation, 1996.

[Booc99] G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 1999.

[Bosc04] J. Bosch, "Software architecture: The next step" in Lecture Notes in Computer Science. vol. 3047, 2004, pp. 194-199.

[Busc96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons, New York, NY, USA.

[Capi08] Capilla, R., Babar, M. A.: On the Role of Architectural Design Decisions in Software Product Lines Engineering. Software Architecture: Second International Conference, ECSA 2008

[Clar96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. ACM Comput. Surv. 28, 4 (December 1996), 626-643.

[Clem03] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures – Views and Beyond. Addison Wesley, Boston (2003)

[Copl96] Coplien, J., Software Patterns, New York; ISBN 1-88484-250-X, 1996.

[Copl97] James O. Coplien. 1997. Idioms and Patterns as Architectural Literature. IEEE Software. 14, 1 (January 1997), 36-42.

[Cord06] Rogelio Limon Cordero and Isidro Ramos Salavert. 2006. Using Styles to Improve the Architectural Views Design. International Conference on Software Engineering Advances (ICSEA '06). IEEE Computer Society, DC, USA, 49-57.

[Czar00] Czarnecki, K., Eisenecker, U., "Generative Programming: Methods, Tools, and Applications", Addison Wesley, 2000.

[Dave98] Davenport, T. H., Prusak, L. 1998. Working knowledge: How organizations manage what they know. Boston, MA. Harvard Business School Press.

[Deme03] Demeyer, Ducasse et Nierstrasz, Object-Oriented Reengineering Patterns, 2003, Morgan Kaufmann Publishers

[Duke95] Roger Duke, Gordon Rose, and Graeme Smith. 1995. Object-Z: a specification language advocated for the description of standards. Computer Standards & Interfaces, 17, 5-6 (September 1995), 511-533.

[Dupu00] Sophie Dupuy, Yves Ledru, and Monique Chabre-Peccoud. 2000. An Overview of RoZ: A Tool for Integrating UML and Z Specifications. 12th International Conference on Advanced Information Systems Engineering (CAiSE '00), Benkt Wangler and Lars Bergman (Eds.). Springer-Verlag, London, UK, 417-430.

[East93] Eastwood, A., “Firm fires shots at legacy systems”, Computing Canada, p. 17, 1993.

[Erli00] Erlikh, L., “Leveraging legacy system dollars for E-business”, IEEE IT Pro, 2000, 17-23.

[Fire05] Donald G. Firesmith: “Quality Requirements Checklist”, in Journal of Object Technology, vol. 4, no. 9 November-December 2005, pp. 31-38, www.jot.fm/issues/issue_2005_11/column4

[Gamm95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[Garl94] D. Garlan, M. Shaw, "An introduction to software architecture", Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[Garl95] D. Garlan, D.E. Perry, "Introduction to the special issue on software architecture", IEEE Transactions on Software Engineering, 21(4): 269–274, 1995.

[Garl00a] D. Garlan, "Software architecture: a roadmap”, Conference on The Future of Software Engineering, pages 91–101, New York, NY, USA, 2000, ACM.

[Garl00b] David Garlan, Robert T. Monroe, and David Wile. Acme: architectural description of component-based systems. In Foundations of component-based systems, Gary T. Leavens and Murali Sitaraman (Eds.). Cambridge University Press, New York, NY, USA 47-67, 2000.

[Gies06] Simon Giesecke. 2006. Taxonomy of architectural style usage. In Proceedings of the 2006 conference on Pattern languages of programs (PLoP '06). ACM, New York, NY, USA, , Article 32 , 10 pages.

[Gies07] Simon Giesecke, Matthias Rohr, Florian Marwede, and Wilhelm Hasselbring. 2007. A style-based architecture modelling approach for UML 2 component diagrams. In Proceedings of the 11th IASTED International Conference on Software Engineering and Applications (SEA '07), Jeffrey E. Smith (Ed.). ACTA Press, Anaheim, CA, USA, 530-538.

[Gold05] Elspeth Golden, Bonnie E. John, and Len Bass. 2005. The value of a usability-supporting architectural pattern in software architecture design: a controlled experiment. In Proceedings of the 27th international conference on Software engineering (ICSE '05). ACM, New York, NY, USA, 460-469.

[Grad87] R. B. Grady, D. L. Caswell, "Software measures: establishing a company-wide program", Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[Grun04] Grünbacher, P., Egyed, A., Medvidovic, N., “Reconciling Software Requirements and Architectures with Intermediate Models,” Journal for Software and System Modeling (SoSyM), Vol.3, N.3, August 2004, pp. 235-253.

[Guo05] Ping Guo, Gregor Engels, and Reiko Heckel. 2005. Architectural Style - Based Modeling and Simulation of Complex Software Systems. In Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC '05). IEEE Computer Society, Washington, DC, USA, 367-374.

[Hami99] Ali Hamie. 1999. Enhancing the Object Constraint Language for More Expressive Specifications. In Proceedings of the Sixth Asia Pacific Software Engineering Conference (APSEC '99). IEEE Computer Society, Washington, DC, USA, 376-.

[Harr08] Harrison, N.B., Avgeriou, P., “Incorporating Fault Tolerance Tactics in Software Architecture Patterns”, Proceedings of the RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems, pp. 9-18, 2008.

[Hofm07] Christine Hofmeister, Philippe Kruchten, Robert L Nord, Henk Obbink, Alexander Ran, Pierre America: “A general model of software architecture design derived from five industrial approaches”, Journal of Systems and Software, Vol. 80, Issue 1, pp. 106--126 (2007)

[Horr04] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, Mike Dean, SWRL: A Semantic Web Rule Language, W3C Recommendation, http://www.w3.org/Submission/SWRL/, May 2004.

[Hors06] Horstmann, C., “Object-Oriented Design and Patterns”, Second Edition, Wiley, 2006.

[IEEE1471–00] IEEE Std. 1471–2000, “IEEE Recommended Practice for Architectural Description of Software-Intensive Systems”, IEEE, 2000.

[IEEE610.12-90] IEEE Std 610.12-1990 (R2002), “IEEE Standard Glossary of Software Engineering Terminology”, IEEE, 1990.

[Inkp96] Inkpen, A. C. (1996) “Creating knowledge through collaboration”, California Management Review, Vol 39, No 1, pp 123-140.

[ISO24765-08] ISO-IEC-24765, “Systems and software engineering vocabulary”, International Organization for Standardization, 2008.

[ISO9126-01] ISO/IEC-9126, “Software Engineering - Product Quality Model”, International Organization for Standardization, Geneva (Switzerland), 2004.

[ISO42010] Standard, I.: ISO/IEC 42010 Systems and Software Engineering - Recommended Practice for Architectural Description of Software-Intensive Systems, 2011.

[ISO25000] Standard, I.: ISO/IEC 25000 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE), 2014.

[Jack06] Jackson, S., Chuang, C., Harden, E., Jiang,Y. 2006. Toward developing human resource management systems for knowledge-intensive teamwork. Research in Personnel and Human Resources Management, 25, 27–70.

[Jans04] Jansen, A., Bosch, J.: Evaluation of Tool Support for Architectural Evolution. In: 19th IEEE International Conference on Automated Software Engineering, pp. 375--378. IEEE Computer Society, Linz (2004)

[Jans05] Jansen, A., Bosch, J., "Software Architecture as a Set of Architectural Design Decisions," in Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on, 2005, pp. 109-120.

[Jans07] Jansen, A., Van der Ven, J., Avgeriou, P., and Hammer, D. K., "Tool Support for Architectural Decisions," in Software Architecture, 2007. WICSA '07. The Working IEEE/IFIP Conference on, 2007, pp. 4-4.

[Kace05] Mohamed Hadj Kacem and Ahmed Hadj Kacem. 2005. Using UML2.0 and GG for Describing the Dynamic of Software Architectures. In Proceedings of the Third International Conference on Information Technology and Applications (ICITA'05) Volume 2 - Volume 02 (ICITA '05), Vol. 2. IEEE Computer Society, Washington, DC, USA, 46-51.

[Kell08] Kelly, S., Tolvanen, J., “Domain-specific Modelling: Enabling Full Code Generation”, ISBN: 978-0-470-03666-2, 2008.

[Khal10] Khaled, L.: Achieving Goals through Architectural Design Decisions. Journal of Computer Science. 6, 1424--1429 (2010)

[Kim09] Kim, S., Kim, D-K., Lu, L., Park, S., “Quality-driven architecture development using architectural tactics”, in the journal of Systems and Software Vol.82, Issue 8, August 2009, pp.1211-1231.

[Kim10] Kim, S., Kim, D., Park, S. “Tool support for quality-driven development of software architectures”, In Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE '10), ACM, New York, NY, USA, 127-130, 2010.

[Kim10a] Jung Soo Kim, David Garlan. 2010. Analyzing architectural styles. J. Syst. Softw. 83, 7 (July 2010), 1216-1235.

[Klei99] M. Klein, R. Kazman, "Attribute-Based Architectural Styles", Technical report, CMU/SEI-99-TR-022, October 1999.

[Kozi11] Koziolek, A., Koziolek, H., Reussner, R.: PerOpteryx: Automated Application of Tactics in Multi-Objective Software Architecture Optimization. In: Proceedings International Conference on the Quality of Software Architectures, pp. 33--42 (2011)

[Kruc06] Kruchten, P., Lago, P., Vliet, H.V.: Building up and Reasoning about Architectural Knowledge. In: 2nd International Conference on the Quality of Software Architectures (QoSA), pp. 39--47. (2006)

[Kruc95] Kruchten, P.: Architectural Blueprints—The “4+1” View Model of Software Architecture. Paper published in IEEE Software 12 (6), pp. 42--50. IEEE Computer Society, Linz (1995)

[Lams03] van Lamsweerde, A., “From System Goals to Software Architecture,” in Formal Methods for Software Architectures, LNCS, vol.2804, pp.25-43, 2003.

[Larm05] C. Larman, “Applying UML and patterns: an introduction to object-oriented analysis and design and the unified process”, Upper Saddle River (NJ), Prentice Hall, 2005.

[Losa03] F. Losavio, L. Chirinos, N. Lévy, A. Ramdane-Cherif, “Quality characteristics for software architecture”. Journal of Object Technology, 2(2): 133–150, 2003.

[Loul04] Imen Loulou, Ahmed Hadj Kacem, Mohamed Jmaiel, and Khalil Drira. 2004. Towards a Unified Graph-Based Framework for Dynamic Component-Based Architectures Description in Z. In Proceedings of the The IEEE/ACS International

Conference on Pervasive Services (ICPS '04). IEEE Computer Society, Washington, DC, USA, 227-234.

[Loul06] I. Loulou, A. H. Kacem, M. Jmaiel, and K. Drira. 2006. Compositional specification of event-based software architectural styles. In Proceedings of the IEEE International Conference on Computer Systems and Applications (AICCSA '06). IEEE Computer Society, Washington, DC, USA, 337-344.

[Medv10] Nenad Medvidovic and Richard N. Taylor. 2010. Software architecture: foundations, theory, and practice. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10), Vol. 2. ACM, New York, NY, USA, 471-472.

[Medv99a] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. “A Language and Environment for Architecture-Based Software Development and Evolution.” In Proceedings of the 21st International Conference on Software Engineering (ICSE’99), Los Angeles, CA, May 1999.

[Medv99b] N. Medvidovic and D. S. Rosenblum. “Assessing the Suitability of a Standard Design Method for Modeling Software Architectures.” In Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), pages 161-182, San Antonio, TX, February 1999.

[Medv00] Medvidovic N., Taylor R.N., “A classification and comparison framework for software architecture description languages”, IEEE Transactions on Software Engineering, Vol.26, No.1, p.70-93, January 2000.

[Medv02] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. 2002. Modeling software architectures in the Unified Modeling Language. ACM Trans. Softw. Eng. Methodol. 11, 1 (January 2002), 2-57.

[Meht00] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. 2000. Towards a taxonomy of software connectors. In Proceedings of the 22nd international conference on Software engineering (ICSE '00). ACM, New York, NY, USA, 178-187.

[Mike09] Categories of software requirements, http://www.mikethearchitect.com/2009/04/qualifying-architecture-with-quality-attributes.html

[Mila08] Mohamed Nadhmi Miladi, Mohamed Hadj Kacem, Achraf Boukhris, Mohamed Jmaiel, and Khalil Drira. 2008. A UML rule-based approach for describing and checking dynamic software architectures. In Proceedings of the 2008 IEEE/ACS International Conference on Computer Systems and Applications (AICCSA '08). IEEE Computer Society, Washington, DC, USA, 1107-1114.

[Mill03] J. Miller, J. Johansson, MDA Guide, Object Management Group, 2003. www.omg.org/docs/omg/03-06-01.pdf

[Monr96] Robert T. Monroe and David Garlan. 1996. Style-Based Reuse for Software Architectures. In Proceedings of the 4th International Conference on Software Reuse (ICSR '96). IEEE Computer Society, Washington, DC, USA, 84-.

[Monr97] Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. 1997. Architectural Styles, Design Patterns, and Objects. IEEE Softw. 14, 1 (January 1997), 43-52.

[Mylo92] Mylopoulos, J., Chung, L., Nixon, B., "Representing and using non-functional requirements: a process-oriented approach," IEEE Transactions on Software Engineering, 1992; 18(6).

[Nadh08] Mohamed Nadhmi Miladi, Mohamed Hadj Kacem, Achraf Boukhris, Mohamed Jmaiel, and Khalil Drira. 2008. A UML rule-based approach for describing and checking dynamic software architectures. In Proceedings of the 2008 IEEE/ACS International Conference on Computer Systems and Applications (AICCSA '08). IEEE Computer Society, Washington, DC, USA, 1107-1114.

[Nort07] Linda Northrop, Architecting High Quality Software, September, 2007 © 2007 Carnegie Mellon University

[Omg06] OMG. Object Constraint Language Version 2, Object Management Group, May 2006, http://www.omg.org/spec/OCL/2.0/.

[Ovas10] Ovaska, E., Evesti, A., Henttonen, K., Palviainen, M., Aho, P., "Knowledge based quality-driven architecture design and evaluation", Journal of Information and Software Technology, Vol.52 pp.577–601, June 2010.

[Pahl09] Claus Pahl, Simon Giesecke, and Wilhelm Hasselbring. 2009. Ontology-based modelling of architectural styles. Inf. Softw. Technol. 51, 12 (December 2009), 1739-1749.

[Pari08] Parizi, R.M., Ghani, A.: Architectural Knowledge Sharing (AKS) Approaches: a Survey Research. Journal of Theoretical and Applied Information Technology, 1224--1235 (2008)

[Ralp85] Ralph E. Steuer, "Multicriteria Optimization -Theory, Computation and Application", 1985

[Rewe06] REWERSE I1 Rule Markup Language (R2ML), 2006, http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/6.

[Reza05] Hassan Reza and Emanuel Grant. 2005. Quality-Oriented Software Architecture. In Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I - Volume 01 (ITCC '05), Vol. 1. IEEE Computer Society, Washington, DC, USA, 140-145.

[Rich02] Mark Richters and Martin Gogolla. 2002. OCL: Syntax, Semantics, and Tools. In Object Modeling with the OCL, The Rationale behind the Object Constraint Language, Tony Clark and Jos Warmer (Eds.). Springer-Verlag, London, UK, UK, 42-68.

[Robb98] J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum. "Integrating Architecture Description Languages with a Standard Design Method." In Proceedings of the 20th International Conference on Software Engineering (ICSE'98), pages 209-218, Kyoto, Japan, April 1998.

[Scot09] Scott, J., Kazman, R., "Realizing and Refining Architectural Tactics: Availability", Technical report, CMU/SEI-2009-TR-006, August 2009.

[Seac03] Seacord, R., Plakosh, D. & Lewis, G., "Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices", SEI Series in Software Engineering, Addison-Wesley, 2003.

[Shah09] Shahin, M., Liang, P., Khayyambashi, M.R.: Architectural Design Decision: Existing Models and Tools. In: WICSA/ECSA 2009, pp. 293--296. IEEE, Cambridge (2009)

[Shar10] Sharafi, S.M., Ghazvini, G.A., Emadi, S., "An analytical model for performance evaluation of software architectural styles", Software Technology and Engineering (ICSTE), pp.394-398, 2010.

[Shaw95] M. Shaw and D. Garlan. "Formulations and Formalisms in Software Architecture." Jan van Leeuwen, editor, Computer Science Today: Recent Trends and Developments, Springer-Verlag Lecture Notes in Computer Science, Volume 1000, 1995.

[Shaw96] M. Shaw, D. Garlan, "Software architecture: perspectives on an emerging discipline", Prentice-Hall, USA, 1996.

[Shaw97] Mary Shaw and Paul C. Clements. 1997. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In Proceedings of the 21st International Computer Software and Applications Conference (COMPSAC '97). IEEE Computer Society, Washington, DC, USA, 6-13.

[Smit00a] G. Smith. The Object-Z Specification Language. Kluwer Academic Publisher, 2000.

[Smit00b] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. Owl web ontology language guide. W3C Recommendation, http://www.w3c.org/TR/owl-guide/, February 2004.

[Spiv92] Spivey, J.M., "The Z Notation: A reference manual", 2nd edition, Prentice Hall International Series in Computer Science, 1992.

[Stad79] Stadler, W., "A Survey of Multicriteria Optimization, or the Vector Maximum Problem," Journal of Optimization Theory and Applications, Vol. 29, pp. 1-52, 1979.

[Stad84] Stadler, W. "Applications of Multicriteria Optimization in Engineering and the Sciences (A Survey)," *Multiple Criteria* Decision Making –Past Decade and Future Trends, ed. M. Zeleny, JAI Press, Greenwich, Connecticut, 1984.

[Stan09] Standish Group, "CHAOS Report", West Yarmouth, Massachusetts, Standish Group Report, 2009.

[Tang10] Antony Tang, Paris Avgeriou, Anton Jansen, Rafael Capilla, and Muhammad Ali Babar. 2010. A comparative study of architecture knowledge management tools. J. Syst. Softw. 83, 3 (March 2010), 352-370.

[Tria95] Triantaphyllou, E., Mann, H. S., "Using the Analytic Hierarchy Process for Decision Making in Engineering Applications: Some Challenges", International Journal of Industrial Engineering: Applications and Practice, 2(1):35--44, 1995.

[Tyre05] Tyree, J., Akerman, A.: "Architecture Decisions: Demystifying Architecture". IEEE Software 22, 19--27 (2005)

[Wanf09] Wanfeng Bu, Antony Tang, and Jun Han, "An analysis of decision-centric architectural design approaches", Technical Report: SUTICT-TR2009.01, 2009.

[Warm99] J. Warmer and A. Kleppe. The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, 1999.

[Wyet09] Wyeth, A., Zhang, C., "Formal specification of software architecture design tactics for the security quality attribute", California State University, Sacramento, Master thesis, 2009.

[Wing98] Jeannette M. Wing. 1998. Formal Methods: Past, Present, and Future. In Proceedings of the 4th Asian Computing Science Conference on Advances in Computing Science (ASIAN '98), Jieh Hsiang and Atsushi Ohori (Eds.). Springer-Verlag, London, UK, 224-.

[Wojc06] Wojcik, R. & a.l., "Attribute-driven design", SEI, 2006. http://www.sei.cmu.edu/publications/documents/06.reports/06tr023.html

[Wood07] W. G. Wood, "A practical example of applying ADD", SEI, 2007. http://www.sei.cmu.edu/publications/documents/07.reports/07tr005.html

[Wood09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. 2009. Formal methods: Practice and experience. ACM Comput. Surv. 41, 4, Article 19 (October 2009), 36 pages.

[Zach11] The Zachman Framework, http://zachman.com/about-the-zachman-framework

[Zimm12] O. Zimmermann, C. Miksovic, J. Küster, Reference Architecture, Metamodel and Modeling Principles for Architectural Knowledge Management in Information

Technology Services. Journal of Systems and Software, Elsevier. Volume 85, Issue 9, Pages 2014-2033, Sept. 2012.

[Zimm11] O. Zimmermann, Architectural Decisions as Reusable Design Assets. IEEE Software, Volume 28, Issue 1, Pages 64-69, Jan./Feb. 2011.

[Zimm09] O. Zimmermann, J. Koehler, F. Leymann, R. Polley, N. Schuster, Managing Architectural Decision Models with Dependency Relations, Integrity Constraints, and Production Rules. Journal of Systems and Software, Elsevier. Volume 82, Issue 8, August 2009, Pages 1249-1267.

[Zhan09] Zhang, W., Hansen, K. M., Fernandes, J., “Towards OpenWorld Software Architectures with Semantic Architectural Styles, Components and Connectors”, In Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '09), IEEE Computer Society, Washington, DC, USA, 40-49, 2009.

[Zou07] Zou, X. Huand, J.C., Settimi, R., and Solc, P. “Automated classification of non-functionnal requirements”. ACM, Requirements Engineering, p. 103-120, 2007.