COMPONENT–BASED SOFTWARE ENGINEERING: QUALIFICATION OF COMPONENTS DURING DESIGN

By

ANDREW STEVEN O'FALLON

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

August 2004

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of ANDREW STEVEN O'FALLON find it satisfactory and recommend that it be accepted.

_____

Chair

_____

_____

_____

ACKNOWLEDGMENT

I would like to thank the School of Electrical Engineering and Computer Science for providing teaching assistant support. I would also like to thank Dr. Anneliese Andrews for not only providing research assistant support, but for the superb guidance on my thesis and in my classes. I would like to acknowledge Jack Hagemeister for all of his support during both my undergraduate and graduate studies. I would also like to thank my other committee members, Dr. Roger Alexander and Dr. Murali Medidi, for positively influencing my experience in graduate school.

COMPONENT–BASED SOFTWARE ENGINEERING: QUALIFICATION OF COMPONENTS DURING DESIGN

Abstract

by Andrew Steven O'Fallon, M.S.
Washington State University
August 2004

Chair: Anneliese A. Andrews

In the software industry component–based software development is becoming an increasingly popular and sought after approach. Integrating components into a software system requires some component–based software engineering practices. Gap fulfillment is a commonly used approach to finding the right component for a software system. This approach has two possible outcomes: usable components are found or no relevant component is found. Component–based software engineering practices require interaction standards, composition standards, and component models. This thesis will focus on a systematic process for selecting and qualifying components during the design phase. Components will be selected and qualified based on interaction and composition information that is found in both static and dynamic UML diagrams. The approach discovers architectural mismatches based on coverage, fit, and coupling measures found in an Integrated UML Model. The Integrated UML Model combines Class and Sequence Diagrams into Object Method Directed Acyclic Graphs (OMDAG). The coverage and fit qualification measures rely on the extraction of information from Constrained Class Tuples (CCTS), which are derived from Class Diagrams in the Integrated UML Model. The Class and Sequence Diagrams are combined into the Constrained Object Method Directed Acyclic Graph (COMDAG)

so that coupling measures can be calculated. Coupling metrics are derived by executing Use Cases, from operational profiles, through the COMDAG. In our approach the selection of candidate components is based solely on available black–box information found in requirements and UML documents. An initial Off–The–Shelf–Option (OTSO) is performed to determine candidate components. For each of the candidate components we use Set Theory to qualify them against our designed components. The combination of the coverage, fit, and coupling measures provides the information for a decision about a candidate component's fitness for a particular software design.

Contents

List of Tables

List of Figures

Dedication

This thesis is dedicated to my entire family for their wonderful support through my graduate studies journey.

Chapter 1

Introduction

Component–based software engineering is a relatively new practice that extends from civil engineering [24]. In the software industry, component–based software development is becoming an increasingly popular and sought after approach. According to Allen, by 2003 or 2004, 70% of new software systems will integrate components [3]. Heineman and Councill define a software component as a "software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard" [24]. Integrating components into a software system requires some component–based software engineering practices. Integration of components follows the component–based software development (CBSD) life–cycle shown in Table 1.1 [24]. The CBSD life–cycle has nineteen steps. The first step requires that the details of the business processes for building the software system are described and modeled. In the second and third steps requirements are gathered from the customers and the system is designed. During the design step the software components are specified in UML and the architecture of the system is chosen. Steps 4 through 7 are exclusive to CBSD. These steps require that the "right" candidate components are chosen and specified, and used to fulfill the

requirements of the software. Steps 8 through 11 describe the implementation steps of the CBSD life–cycle. The software developer must select an integrated development environment (IDE) for building the software. If database, middleware, and/or client components are required, then these components must be developed. Once the components are developed, they must be exhaustively tested (step 12). Unit, integration, and system testing must be performed on the components. Steps 13 through 17 illustrate the activities involved in component update and replacement, along with steps involved in component and system maintenance. Steps 18 and 19 are related to business growth and the merger of business and/or software processes.

Step	**Software Development Life Cycle Phase**
1	Model Business Processes
2	Manage Requirements
3	Model System Design (Components)
4	Gap Fulfillment
5	New Component Specification
6	Component Use
7	Get on Mailing List
8	Select Integrated Development Environment (IDE)
9	Build Database (Component Assembly)
10	Build Middleware (Component Assembly)
11	Build Client Software (Component Assembly)
12	Test
13	Roll–Out
14	Receive Notification, New Component
15	Review New Component
16	Update Design
17	Maintenance
18	Extension
19	Merge Systems

Table 1.1: CBSE Software Life Cycle.

This thesis is specifically interested in presenting a systematic approach to selecting and qualifying components during gap fulfillment. Gap fulfillment is a com-

monly used approach to finding the right component for a software system [24]. This approach has two possible outcomes: usable components are found or no relevant component is found. Component–based software engineering practices require interaction standards, composition standards, and component models [24]. This thesis provides a design engineer with a systematic, quantitative process for selecting and qualifying components during the design phase. Components are selected and qualified based on interaction and composition information that is found in both static and dynamic UML diagrams. The approach presented in this thesis identifies architectural mismatches based on coverage, fit, and coupling measures found in an Integrated UML Model [47, 48]. The Integrated UML Model combines Class and Sequence Diagrams into Object Method Directed Acyclic Graphs (OMDAG). The coverage and fit qualification measures rely on the extraction of information from Constrained Class Tuples (CCTS), which are derived from Class Diagrams in the Integrated UML Model. The Class and Sequence Diagrams are combined into the Constrained Object Method Directed Acyclic Graph (COMDAG) so that coupling measures can be calculated. Coupling metrics are derived by executing Use Cases, from operational profiles, through the COMDAG. In our approach the selection of candidate components is based solely on available black–box information found in requirements and UML documents. An initial Off–The–Shelf–Option (OTSO) is performed to determine candidate components. For each of the candidate components we use Set Theory to qualify them against our designed components [43]. The combination of the coverage, fit, and coupling measures provides a final decision about a candidate component's fitness for a particular software design.

Software design is increasingly including component–based software development [24]. This happens from two perspectives: (1) A software designer wants to use components and needs to define how components fit with the remainder of the design.

Candidate components must then be evaluated as to whether or not they fit into the design. The latter is part of component qualification. (2) A software designer wants to design components for reuse, possibly as part of a product line architecture, or as part of a set of components developed for reuse.

For either case we propose an analysis method that works for designs expressed in UML Class Diagrams and Sequence Diagrams. The analysis method is derived from Pilskalns et al. [47]. In [47], a model is derived from Class Diagrams and Sequence Diagrams that integrates both structural and behavioral characteristics of the design (Integrated UML Model). This model is used to generate and execute tests. The qualification model presented in this thesis uses the Integrated UML Model to define coverage, fit, and coupling measures for a component. The first step in the qualification model is to identify candidate components. Next, the Integrated UML Model is built. Thirdly, the qualification model is built. Fourthly, coupling measures are calculated. Lastly, the candidate components are ranked according to the qualification and coupling measures. Steps 2 through 4 of the model address research problem rp–1.

rp–1 Are we able to qualify components in the design phase based on a Set Theory analysis?

Our qualification method is only useful if it is capable of determining if a candidate component is fit for a particular architecture. The method needs to help a design engineer to correctly identify matches between candidate components and design components. Thus, we need to address rp–2.

rp–2 Is this method effective at qualifying components?

Sometimes an effective method is not always the best solution to a problem. We

need to show that our method is also efficient. The method has to be convenient and cost effective (time saving). Thus, we need to address research problem rp–3.

rp–3 Is this method efficient at qualifying components?

Chapter 2 describes existing work on component selection, evaluation, and qualification. Chapter 3 explains the qualification process. Chapter 4 describes the decision making framework for applying the measures in defined by the qualification process. Chapter 5 applies the qualification method to two examples. Chapter 6 draws conclusions and suggests further work.

Chapter 2

Background and Related Work

2.1 Overview

This chapter presents work that is either the foundation for or related to the work presented in Chapters 3, 4, and 5. This chapter presents a comprehensive literature search of existing research papers. We have searched for papers that have contributed toward our research on qualifying components in the design phase.

Our search started with the identification of research papers related to testing of traditional or component–based software. The foundation of the component qualification method presented in this thesis is based on the Integrated UML Model approach described in [47] and [48]. The approach tries to integrate UML class and sequence diagrams via formal methods. Once the two UML views are integrated, objects in the design phase are tested for incorrectness and inconsistencies. The test adequacy criteria developed for the integrated model are derived from criteria defined in [4]. For this thesis we are concerned with developing a new approach to defining and qualifying components based on Pilskalns' et al. integrated model. The qualification approach focuses on available design information found in the integrated model.

The approach to define and qualify components may be viewed as part of the design phase in the component–based software life–cycle (Section 2.2). The definition of components is partially based on cohesion and coupling measures defined in [10]. We also derive Use Cases from a particular operational profile and set of requirements (Section 2.2.1), to help derive coupling measures. From Class and Sequence Diagrams, we can determine static and dynamic interactions that may naturally qualify components. Our approach to qualifying components may be beneficial in manually determining architectural mismatches (Section 2.2.5) and components that are suitable for a particular product line (Section 2.2.6). Once we have the components designed and are ready for qualification, we first need to select candidate components (Section 2.2.2). We apply the Off–The–Shelf–Option (OTSO) for searching and selecting candidate components [31]. The candidate components may be retrieved from on–site repositories (Section 2.2.4) or through other component sources [26]. After the candidate components have been identified and retrieved, our Set Theory approach to qualification is applied. The related work and foundation for our qualification research is supplied in Chapter 3.

The objective of this research is to determine if we can define a process for qualifying components based on UML design information. The qualification method is a systematic and feasible approach to determining if a component is fit for a particular design or architecture. Our approach is advantageous because it only requires basic UML artifacts to perform component qualification. The method may be automated effectively and efficiently because it extracts signature information from component interfaces and generates coverage and fitness metrics. This thesis proposes original methods qualifying components during design.

2.2 Component–Based Software Development

The Integrated UML Model introduced by Pilskalns et al. in [47] is presented in Section 3.3. The model is the basis for the qualification method that is presented in this thesis. However, before the qualification approach is explained, we must understand how we apply Use Cases, derived from requirements, to defining qualification (coupling) measures.

2.2.1 Software Specification and Requirements of Components

Organizations have different guidelines for specifying requirements of a software system. According to Heineman, defining components and their attributes before the selection and evaluation phases of component–based software engineering, is a critical step toward finding the "right" components [24]. Both functional and non–functional requirements need to be considered for this process. James and Suzanne Robertson in [50] have developed a software requirements specification document that outlines the approaches for collecting requirements for modern software systems, which includes component–based systems. The document provides a section for each of the requirement types commonly used in contemporary software systems. The template emphasizes project drivers, project constraints, functional and non–functional requirements, and project issues. The template follows the Volere requirements process which has been adopted by various organizations throughout the world based on the collected experiences of various software developers. The Robertson's claim this document will gather all requirements needed to specify a complete project. This thesis will focus on deriving use cases from functional requirements. The requirements must be correct in order to derive the correct Use Cases. Steps should be taken to detect

inconsistencies in requirements specifications before Use Cases are defined.

In [25], Heitmeyer et al. focus on detecting inconsistencies in requirements specifications. The team of researchers use a formal method based on the Software Cost Reduction (SCR) tabular notation. The approach exposes inconsistencies between redundant specifications. The two redundant specifications include an operational specification and a property–based specification. The method does not require any mathematical skills and may be applied by most software developers. Abstractions are used to reduce the "state space" in requirement specifications. The focus of this thesis is not to find inconsistencies in requirements specifications, but rather to provide a means to defining qualification measures from Use Cases based on the requirements of the system. However, determining that the requirements are complete is not always trivial.

According to Beus–Dukic and Wellings, selecting components based on supplier information may lead to pitfalls [9]. Comparing a feature list supplied by a vendor to a requirements document is not always effective. A software developer must also perform some design analysis before completing the requirements of a Component–Off–The–Shelf (COTS) based system. The architecture and requirements of the system change drastically when COTS are to be used. Estimating the difficulty associated with COTS–based systems is critical in deciding whether or not a component based approach should be pursued.

Boehm, in [12], describes the next version of COCOMO (COnstructive COst MOdel), COCOMO 2.0. COCOMO 2.0 has added new techniques for estimating software development efforts. The newest version tries to produce a new method for component–based software systems. The method uses object and function points, and source lines of code as factors for determining effort required to build the system. The paper also claims that reusing components is best if viewed from a black–box per-

spective. The effort is expressed in terms of person–months. Ellis in [17] adds another perspective to estimating costs of glue code for COTS–based systems. The model is useful for providing a basis for selection and evaluation of candidate COTS components. The costs of integrating the COTS include such factors as function points of the glue code, the percentage of requirements that are satisfied by the component, and a difficulty of integration factor for each COTS component. The difficulty of integration factor is important in the selection of components for a particular software system. This thesis tries to determine such factors based on the interactions between components and the complexity of the component interfaces. The next section describes various selection and evaluation techniques needed before integrating components.

2.2.2 Component Selection and Evaluation

Component selection and evaluation is an important area for this research because it is subsumed by component qualification. Many of the following papers have interesting techniques for selecting and evaluating components. Many of them, however, are based on functionality as opposed to UML artifacts in the design phase. The research presented in this thesis attempts to evaluate components based on the contents of UML documents. Cardenas–Garcia and Zelkowitz describe an interesting tool called Selector for the evaluation of software designs [21]. The tool considers two cases in evaluating software designs for the selection of components. In the first case, the certainty case, the manager knows the relative rankings of attributes and trade–offs. The tool determines an evaluation measure based on the attribute values of each component supplied by the manager. During the second case, the uncertainty case, a probabilistic determination for each of the candidates is computed. For this case the

relative rankings of attribute values are uncertain. Selector allows for the use of risk analysis and decision theory to effect the outcome of the software design process. We have to understand the risks and difficulties associated with selecting and evaluating COTS components.

Briand in [14] explains the difficulties and problems associated with COTS selection and evaluation. Briand points out that COTS evaluation is difficult because it is an optimization problem [29]. He also suggests using Analytic Hierarchy Process (AHP) when decisions need to be made about multiple COTS components. These decisions are generally based on quality criteria, functional criteria, architectural criteria, and compliance with standards. Many of the decisions for COTS selection are driven by quality attributes.

In [28], Jeanrenaud and Romanazzi describe the Checklist Driven Software Evaluation Methodology (CDSEM). CDSEM builds on using the quality characteristics defined in ISO 9126 as a guide for software evaluation. Six quality characteristics drive the evaluation process including: functionality, reliability, usability, maintainability, portability, and efficiency. The research applies measurements to the characteristics. The sum of the relative importance of each characteristic is gathered and used to make a final decision about a component. The CDSEM presents a semi–systematic approach to evaluating a component. However, the method is not very original. Many methods for evaluating components drive evaluation by quality characteristics. Fenton in [19] believes that a product may be evaluated by external or internal attributes. These attributes can be either directly or indirectly measured. Such measurements may be a the first step in the evaluation of multiple candidate components. Internal attributes such as size, algorithmic complexity, and levels of inheritance may not be available with COTS components. The more realistic approach to measuring attributes for COTS components would be the more conventional black–box methods,

such as measuring functionality, usability, and maintainability. For this thesis we need to be able to understand where our qualification method fits into the whole software development picture. We need to understand that a more complete description of software selection and evaluation is necessary.

In [40], Morisio and Tsoukias introduce the IusWare (IUStitia SoftWARis) formal methodology for evaluating software products. The method consists of selection, assessment, and evaluation of software products. IusWare encompasses two main phases: design and application. The design phase consists of defining the actors involved with the software product. During the design phase the type of evaluation required, either formal or partitioning of products, is defined. The design phase also encompasses defining the quality attributes and associating measures with each of the attributes. Lastly, an aggregate value is computed to determine the recommendation for the component. In the application phase attributes are measured and aggregated to form a recommendation of the product.

Morisio and Tsoukias' method is not the only method that consists of selection, assessment, and evaluation. Lichota et al. introduce the Portable, Reusable, Integrated, Software Modules (PRISM) Program started by the U.S. Air Force [36]. The paper describes the approach on building command centers. PRISM uses the Product Examination Process (PEP), a multi–phased approach for selecting COTS components. PEP contains five phases: identification, screening, stand–alone test, integration test, and field test. The identification phase determines if a candidate component should be selected for further examination. During the screening phase the components selected for examination are filtered based on available documentation. In the stand–alone testing phase each of the COTS components are tested for performance. The integration testing phase tries to determine how effectively a component can be integrated into a given architecture. Components that pass the test

are entered into a reusable component repository. The field testing phase determines the inter–operability of a component for field operational profiles. The component that performs the best during the testing phases is selected for integration into the system. This thesis will select candidate components based on a six phase process. The process is complete for searching, selecting, and evaluating components.

In [31], Kontio describes the Off-The-Shelf-Option (OTSO). This method is considered an extension of the Experience Factory. The process for selecting reusable components is defined. The process contains the capabilities for customization. The OTSO selection process has six phases: search, screening, evaluation, analysis, deployment, and assessment. For each phase the number of alternatives decreases. The first phase, search, accumulates all possible candidate components based on requirements and criteria defined in the criteria definition process. The search phase requires multiple search locations. These locations should be documented and include in-house component repositories, the Internet, magazines, journals, conferences, colleagues, vendors, etc. The search phase should terminate when searching for new alternatives, finds few. During the screening phase the best suitable components are chosen from all alternatives. The components are chosen based on qualifying thresholds (as will our method) which limit the number of possible candidates chosen for evaluation. The evaluation phase analyzes the components based on evaluation criteria and document evaluation results. The evaluation includes analyzing the tools and studying the available features. Time delays for shipping of the product or product releases may be an important factor when evaluating the product. The evaluation of each component is documented for future reference and for later analysis. The analysis phase includes a proper assessment of data collected during the evaluation phase. The qualitative characteristics of the alternatives are analyzed using the Analytic Hierarchy Process (AHP). Kontio applies the OTSO method to a case study in [30].

The paper also describes the pitfalls and problems found while applying the selection method to the case study. The case study reported involved a NASA project under contract to the Hughes corporation. The results indicated that the OTSO method provides an efficient, consistent, and quality evaluation process. The results of the cases study also indicated that the AHP method is more effective at producing relevant information for COTS selection than the weighted scoring method (WSM). Kontio et al. further apply the OTSO method to case studies in [32]. Once again the method seems to be very effective in selecting and evaluating COTS components for use within a component–based software system. In order to find components that will satisfy a given design, our approach uses the search, screening, and evaluation of the OTSO method. However, we modify the six phase process so that we can collect measures to identify candidate components that are suitable for a given design based on an integrated model. We use the AHP to evaluate coverage and fitness metrics during qualification.

The AHP contains six steps. Four of the steps determine the importance and relative weights of factors involved in the analysis [51]. The fifth step is used to check the consistency of the results. The last step is used to calculate an aggregate variable. In the first step a square priority comparison matrix is created. The matrix element at cell i, j, records the relative importance of variable i versus variable j [55]. The second step involves a pairwise comparison of the variables. The comparison consists of contrasting the importance of one variable with another. The granularity and refinement of the importance scale can be adjusted according to the number and types of factors involved. The pairwise comparison tries to determine how much more important one variable is than another. If a variable i is more important than a variable j, then a mark is placed toward the i. The proximity of the mark determines how much more important i is than j. The mark indicates a ranking of the variables. The value in

each cell of the comparison matrix reflects the rankings of the variables. For example, if i was assigned a 5 on the scale, the cell at i, j would contain a 5 and the cell at j, i would contain 1/5. Step three involves computing the eigenvector of the comparison matrix. Saaty uses averaging over normalized columns to compute the eigenvector or the priority vector [55]. The fourth step uses the priority vector calculated in step three to assign importance values to variables. The i^{th} variable is assigned the value in the i^{th} position in the priority vector. The fifth step analyzes the consistency of the rankings of the variables. A consistency index and a consistency ratio are computed. If the consistency ratio is 0.10 or less, the rankings are considered acceptable. Step six is applied when the consistency ratio indicates an acceptable result. Aggregate variables can be computed. Other methods are available for assigning weights to variables. We decided that AHP best fits our needs. A comparison of AHP and the multi–attribute value theory is described below.

In [8], Belton compares and exposes weaknesses of both the AHP method and the multi–attribute value theory (MAVT). According to the author, one of the weaknesses of AHP is that a ratio scale of measurement is required, whereas the MAVT only requires an interval scale. Also, assigning weights to variables with the AHP is perceived as ambiguous. The major weakness of MAVT is that it does not have analytic support for measuring consistency of measurements. AHP quantifies consistency with a consistency index and ratio. A comparison of the two methods also showed that neither one produces the same results. However, as long as the steps involved with both methods are kept consistent, the results of the two methods should be reasonable. We apply the use of AHP to our component qualification approach. In the next section more qualification approaches are described. These approaches, unlike the one to be presented, do not use UML artifacts to qualify components.

2.2.3 Component Qualification

Ideally a qualification method would be performed based on quantitative methods and the design of a system. Our approach applies Set Theory to derive some quantitative metrics for the overall qualification of a candidate component. The following paper applies some qualitative and quantitative methods, but the assessment is based solely on black–box techniques as opposed to addressing the design of the components. Maiden and Ncube in [37] describe the PORE (Procurement–Oriented Requirements Engineering) method for acquiring software requirements. The PORE method is applied to component–based systems. The authors along with two other team members acquired requirements for a software system from various documents and stakeholders. The team then formed a questionnaire and sent it to candidate component suppliers. The supplier's answers to the questionnaire allowed the team of engineers to determine which COTS components most fulfilled the requirements. Six candidate components were selected. Test cases were applied to the candidate components for evaluation. The evaluation of the components were based on qualitative and quantitative assessment. The main stakeholder then added weights of importance to the requirements based on the assessment. The lessons learned from the component evaluation process led to the PORE method. The PORE method is an iterative requirement acquisition process that continues until most if not all of the customer requirements are satisfied. The process has three stages. Each stage provides guidance for acquiring customer and product information. The COTS products are selected based on supplier information, test cases, and customer–led demonstrations of the product.

Beach and Bonewell propose a way to qualify COTS components for medical systems based on quality criteria [7]. There are two major phases to the selection of can-

didate components for medical systems. The first phase includes evaluation planning. The most important step during evaluation planning is to ensure that all software requirements are well defined and complete. Steps must be taken to ensure that the COTS component is fit for the medical system. Criteria for evaluating the COTS components should be established according to adherence to requirements, costs, the vendor's process maturity level, and the suggestions of users of the components. The summation of the quantitative scores collected for each criterion for each COTS software package are then evaluated and the highest score identifies the best component for the system. The second phase of the selection of a component is qualification planning. Qualification planning involves the use of multiple documents including requirements specification, interface specification, test plans, traceability documents, configuration management plans, and hazard analysis. Three qualification activities that should be performed during qualification include acceptance testing, verification testing, and review of the vendor's quality system. The final step in the qualification planning phase is to write a report which leads to a decision about a COTS component. Although the approach in [7] is interesting, the approach does not address the problem of qualifying components in the design phase.

The COTE (COmponent TEsting) project [27] is concerned with developing an integrated environment (IE) for qualifying and testing components. The main goal of the research is to develop the IE which inputs UML as the test description language. The researchers are interested in testing the messages, callbacks, branching, loops, and active objects. Sequence diagrams are used to build the UML test profile. The research does not try to combine UML views to develop a new test execution profile, instead the approach tries to simplify the UML views. Jard and Pickin are not solely focused on the pre–implementation phase. Pickin et al. further describe an integrated approach to testing components using interaction diagrams as the test description

language in [46]. The research is interested in exploiting the UML environment and developing an integrated UML test description language (TeLa) that is applied to black–box tests. The tests will target messages exchanged between the system under test and the tester. The test description language specifies syntax for loops, branching, and concurrency as found in UML sequence diagrams. The authors also describe the informal semantics of UML interaction diagrams and then propose new constructs that may be needed for a testing environment. In [56] sequence and collaboration diagrams are used for testing the interactions between objects. However, the research does not completely address a way to test the components via a systematic approach. They focus on ways to test code as opposed to ways to find faults in the design phase.

Baudry et al. are concerned with a testing–for–trust method that integrates design and test approaches for qualifying components [6]. The method views a component as a set composed of a specification, implementation, and test cases. The testing–for–trust approach uses mutation analysis to check for consistency between specification, implementation, and testing. This method is not concerned with the design and UML modeling phase of the software life-cycle. The approach uses genetic algorithms to determine test cases that may further check for inconsistencies in the components. This approach is not capable of taking advantage of UML diagrams to qualify components. For our approach we realize that retrieving components from vendors or repositories is important, however we will not integrate retrieving methods into our approach. Some methods are acknowledged below.

2.2.4 Component Repositories and Adaptation

When searching for candidate components for gap fulfillment, several locations may be searched. Among the locations are local repositories. Sophisticated search and

retrieval methods are required to ensure that most if not all candidate components for a design may be found. In [35], Lester, Wilkie, and Bustard apply the idea of using stereotypes, class compartments, and association rules for qualifying the reuse of software artifacts. These UML constructs are used to define search criteria for reuse candidates. The use of the stereotype limits the search to those objects which contain the stereotype or are derived from the object with the stereotype. Attribute-Value classification can provide a structured integration of association roles into the search criteria of an object. Using classifications for reusable components does not guarantee that for a given design components with the best fit are identified.

Henninger explains component repositories in [26]. He tries to minimize the classification and domain analysis required to design and support a component repository. The paper tries to reduce the dependence on classifications by using tools, PEEL (Parse and Extract Emacs Lisp) and CodeFinder. In general a classification system relies heavily on the structure of the repository. If the structure of the repository is not properly built at the beginning, then retrieval of components may be very inefficient and difficult. Three main retrieval methods exist. They include enumerated classification, faceted classification, and free–text indexing. Enumerated classification is the method used by the Dewey Decimal system and relies heavily on the structure of the repository [2]. Faceted classification does not enumerate the components into a hierarchy, but instead classifies components based on attributes. For this classification method less structure is required for the repository. The last classification, free–text indexing or automatic indexing, use the text contents of the component or document for indexing them. The queries formed by the users required only keywords that may be found in the documents. Free–text indexing methods do not require any structure to the repository. Enumerated and faceted classification systems do not adapt well to the changing infrastructures of repositories. The free–text indexing methods are

just too simple and naive. Henninger proposes an evolutionary design of repositories. This design of the repository requires little initial effort and structure. The repository used techniques to incrementally improve as it is used. Initially the repository must be populated with components, the PEEL tool then uses automatic extraction to retrieve components from text files and uses semi–automatic methods for indexing into the repository. The resulting repository may contain incomplete and inconsistent information. Thus, query reformulation and soft–matching techniques are used to restructure the information. The CodeFinder system then uses new retrieval techniques to find the components allowing the overall system to adapt to the evolving environment, which become more effective as the repository is used more frequently.

Morel and Alexander present the SPARTACAS framework for automating specification based retrieval and adaptation in [39]. The framework presents adaptation architectures for components that do not completely satisfy the requirements of a system. The first step in the SPARTACAS framework is to input formal specifications into the retrieval engine. The engine then returns partial and total solutions that satisfy the requirements. For the partial solutions, the adaptation engine will generate adaptation architectures that contain subproblems for fulfilling the missing functionality. Once the the requirements of the system have been satisfied, the new components are verified. The new components created from the architectural adaptations are then added to the component repository for future use and solutions to future problems. The component retrieval framework uses a layered architecture approach. Each layer functions as a filter for possible candidate solutions to the required software system. If a component passes through all filters, it has a high probability of satisfying the specification of the system. This tool seems to be very powerful for finding solutions to a particular software architecture and design. The system, however, requires too much information and there is no evidence that the tool is efficient

and effective for finding solutions to a particular problem.

2.2.5 Architectural Mismatches using Components

In [16], Egyed and Gacek describe two integration approaches for detecting mismatches during component based development. One approach detects component feature clashes. Observations and assumptions about the behavior of the component are used in order to detect clashes. The connections between various components are also used in the process. The connections described include call, spawn, shared data, shared repository, data connector, trigger, and shared resources. A set of conceptual features are used to describe the components. The set consists of backtracking, component priorities, concurrency, control unit, distribution, dynamism, encapsulation, layering, preemption, reconfiguration, re-entrance, response time, supported data transfers, and triggering capabilities. This approach attempts to identify mismatches early in the development phase. Incomplete component specifications are handled. Detecting mismatches in the design phase would be more practical and economical. Our approach tries to qualify candidate components for a particular design and architecture. Egyed's and Gacek's approach targets in-house, COTS, and legacy systems. The Architect's Automated Assistant (AAA) tool performs analysis for component mismatches based on the high level descriptions provided. The other approach detects mismatches in UML views of the system. The mismatches are primarily driven by inconsistencies and incompleteness of the views and architecture information. The approach uses both rules and constraints to determine inconsistencies. The view analysis performs three activities: mapping, transformation, and differentiation. Mapping identifies relationships between information through traceability matrices. Transformation extracts the information from the models so that

they can be used in different views. Differentiation identifies the mismatches in the views through the constraints or rules supplied. The power of the approach presented in this thesis is that it attempts to best match components based on UML diagrams and architecture. Our approach tries to define components based on the UML diagrams and is helpful for developing core assets needed for product lines.

2.2.6 Product Lines

Developing software from scratch is expensive and requires diligent effort toward software specification, design, and quality assurance. Product lines try to reduce the effort required to build software by reusing software artifacts. According to Northrop [41] a product line is a set of "software–intensive" systems that share common features. These systems often share core assets which include: architecture, reusable software components, requirements, domain models, documentation, schedules, budget, test plans and test cases, etc. Product line development requires core asset development, product development, and organizational management. Northrop describes 29 practice areas that need to be mastered before a successful product line may be developed. Each of these practice areas is directly related to core asset development, product development, or organizational management. Architecture definition and evaluation, component development, and COTS utilization are some of the required practices for a successful set of core assets and products. Configuration management, data collection, process definition, business cases, and market analysis are some of the practices required for successfully developing product lines. This thesis does not try to remedy any of the problems with developing and maintaining product lines. We acknowledge that our methods for defining and qualifying components are effective and efficient approaches to developing and maintaining core assets for product lines.

2.3 Summary of Literature Search

The purpose of the literature search performed was to find supporting research for this thesis. The other purpose of the search was to ensure that the research presented in this thesis is novel. This thesis provides a systematic and quantitative approach to qualifying components. We match a design component with one or more candidate components. The match is based on available UML design models, such as Class and Sequence Diagrams. We need to be able to apply our qualification approach during the design and gap fulfillment phases of the CBSE Software Life Cycle. The qualification process uses Set Theory Analysis. Other approaches to qualification include applying qualifying tests during the implementation phase of the life cycle. These methods do not take advantage of the UML design documents that may be available for the component. These qualification methods are applied later in the life cycle and thus any changes in the selection of the component is more costly.

Table 2.1 illustrates the result of the literature search and review. Table 2.1 shows the results of the search for papers related to general software development practices. The table addresses the papers that either use UML (w/ UML) or do not use UML (w/o UML) during the software life–cycle. The literature search indicates that component qualification in the design phase, besides one of our papers [43] marked by ++++++, yields few results. Qualification of components during the design phase is the main focus of this thesis. We suggest to use our approach as part of the OTSO method introduced by Kontio [31]. The approach to qualification involves applying Set Theory to constrained class tuples (CCT) developed from Class Diagrams [47]. CCTs are described in Section 3.3.1. The Set Theory approach to qualification of components in the design phase is described in Section 3.4.3.

	Spec. & Requirements	**Selection & Evaluation**	**Qualification**	**Repos. & Adaptation**	**Arch. Mismatches**	**Product Lines**
CBSD w/ UML	NA	[35]	[43] ++++++	[2], [26], [35]		[41]
CBSD w/o UML	[9], [24], [25], [37], [50]	[8], [10], [14], [19], [21], [28], [29], [30], [31], [32], [36], [40], [51], [55]	[6], [7]	[23], [37], [39]	[16]	

Table 2.1: Existing Work in Component–based Software Development using UML.

Chapter 3

Qualification Process

3.1 Overview

The traditional software lifecycle consists of four phases: analysis/specification, design, implementation, and testing [49]. According to Pressman [49] the impact of change is more expensive as a project progresses through the lifecycle. The same concept applies to the component–based software lifecycle shown in Table 1.1. However, the component–based lifecycle requires that components are qualified for a particular design. Components are generally qualified during the beginning of the implementation phase. Companies that have spent a lot of money and effort in making the business case for components have two choices for integrating components [24]. They can either design their system and architecture around existing components or they can write glue code to modify the components for the existing architecture and design of the system. Determining that a component is not fit for a particular design or architecture in the implementation phase is 1.5–6 times more costly than in the design phase [49]. The qualification approach presented in this chapter detects whether a component is fit for a particular architecture in the design phase of the lifecycle. The

earlier in the lifecycle that qualification is performed the less costly it is. Component–based software engineering hinges on the fact that components save costs in software development and testing. This chapter presents our systematic process for qualifying components during the design phase. The first step, Section 3.2, requires that candidate components are identified. The second step, Section 3.3, requires that an Integrated UML Model is constructed for use in defining components and calculating coupling measures. The third step, Section 3.4, requires that the Qualification Model is built. The fourth step, Section 3.5, calculates the coupling measures. The fifth and final step, Section 3.6 ranks the components based on the qualification or coupling measures. Table 3.1 lists the phases in our approach.

Step	**Qualification Phase**
1	Identify candidate components
2	Build the Integrated UML Model
3	Build the Qualification Model
4	Calculate coupling measures
5	Rank components based on qualification or coupling measures

Table 3.1: Steps of Qualification Process.

Figure 3.1 shows the process to build the Integrated UML Model, Figure 3.2 pictures the Qualification Model, and Figure 3.3 illustrates the qualification approach.

3.2 Identifying Candidate Components

The first step in the qualification process is to identify candidate components. In order to identify components we will first follow the phases of the Off–The–Shelf–Option (OTSO) method developed by Kontio [31]. However, we will only use the first two phases of the process: search and screen. During the first phase, we will search for candidate components based on requirements and criteria defined during specification

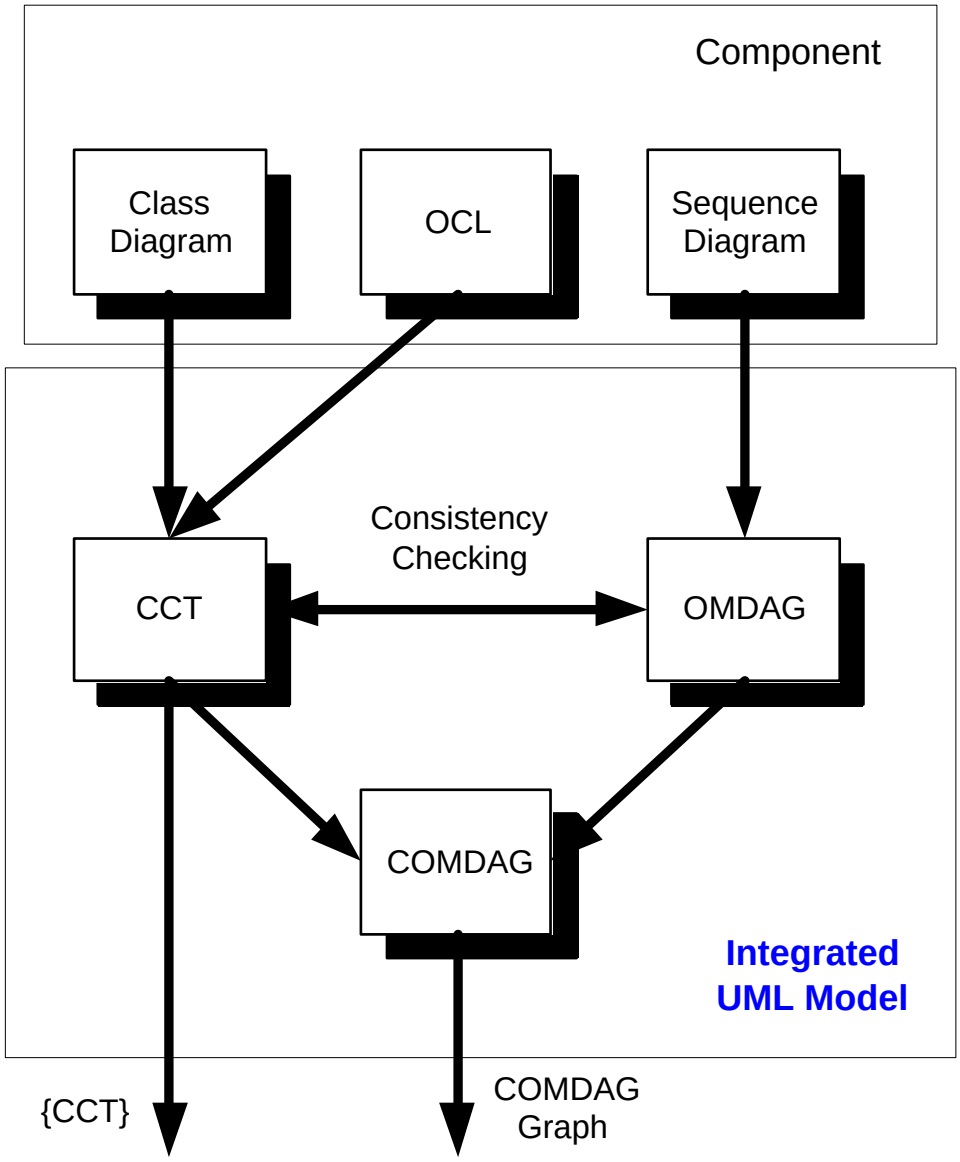


Figure 3.1: The Integrated UML Model.

and architecture definition. During the search phase we will make a first attempt at satisfying most requirements with a variety of candidate components. The search phase accumulates potential components into a list. Each component in the list may be identified by its name, reference to the source of its location (web address, company, etc.), and main characteristics such as features, price, documentation available,

operating platforms, etc. The search domain for candidate components should be well documented for future use. The search domain consists of in–house repositories, the Internet, magazines and journals, conferences, vendors, and other organizations. Searching for candidate components should begin as soon as requirements are defined for the system. According to Kontio [30], the searching phase is complete when the incremental searching for new candidate components finds few new alternatives. Once the searching phase is complete, candidate components must be screened.

The screening phase is based on satisfying requirements and architectural style. These requirements may be extended in order to accommodate a "qualifying threshold" [31]. The "qualifying threshold" is set according to the number of requirements necessarily satisfied and a subjective assessment of whether the candidate component fits the architectural style of the design. If potential components do not satisfy the threshold they are eliminated and documented for potential use in other software systems. The screening phase is complete when evaluation tasks have been assigned to the components that have not been eliminated.

The next phase in the OTSO method is to evaluate each of the alternatives selected during screening [31]. We also evaluate the potential candidates, however, our method is more conscientious of the architecture of the defined system. Our qualification method subsumes the evaluation phase during the identification of the candidate components. During the second step of our qualification process, an Integrated UML Model is built around the designed components in our system as well as the candidate components identified.

3.3 Building the Integrated UML Model

The second step in our qualification process is to build the Integrated UML Model (Figure 3.1). The Integrated UML Model is defined by Pilskalns in [47] and [48]. In order to measure component fitness we need information about attributes and methods which may be found in Class Diagrams. More specifically our fitness measure requires the set of Constrained Class Tuples (CCT) of the component. Our coupling measure requires that both dynamic and static views of the component are available. Essentially Sequence and Class diagrams need to be merged into a directed graph (OMDAG) because the two views alone do not have the information necessary to collect our metrics. Once the constraints on the OMDAG are determined (COMDAG), we use an operational profile to walk through the COMDAG. We collect coupling metrics based on the connections between nodes and the number of activations that are encountered during the walk through the graph. The integrated model contains three steps shown in Table 3.2.

Step	**Phase**
1	Construct Constrained Class Tuples
2	Build the Object Method Directed Acyclic Graph (OMDAG)
3	Add constraints to the OMDAG to form the Constrained OMDAG (COMDAG)

Table 3.2: Steps to Build UML Integrated Model.

3.3.1 Constrained Class Tuples

The first step in building the UML integrated model is to construct Constrained Class Tuples (CCT) from class diagrams and corresponding OCL expressions [47]. CCTs contain static structural information that is used in Pilskalns et al's [47] testing

approach and by our Qualification Method. The CCT flattens the class hierarchy into object instances that contain both attribute and method information. OCL invariants, preconditions, and postconditions constrain the information that is present in the Class Diagrams. A Constrained Class Tuple CCT of a class c has the form in Equation 3.1 [48]:

$$CCT(c) = \langle \{\langle\, Parent\ CCT \rangle\},\ \{\langle\, Attribute \rangle\},\ \{\langle\, Method \rangle\}, \{[invariant]\}\rangle \quad (3.1)$$

where, c is the class name, $\{\langle\, Parent\ CCT \rangle\}$ is a set of parent class CCTs, $\{\langle\, Attribute \rangle\}$ is a set of attribute tuples, $\{\langle\, Method \rangle\}$ is a set of method tuples, and $invariant$ is the set of OCL constraints on the information in the class c.

The set of *Attribute* tuples contains all attribute information for a class. The *Attribute* tuple is defined as follows in Equation 3.2 [48]:

$$Attribute = \langle\, attr\ name,\ attr\ type,\ visibility,\ initial\ value, \{[invariant]\}\rangle \quad (3.2)$$

The *attr name* describes the attribute, the *visibility* describes the accessibility (public, private, or protected) of the attribute, and the *invariant* represents the OCL associated with the attribute. The set of $Method$ tuples defines all of the methods in a class. Each $Method$ tuple may contain parameters. Each parameter is defined with a tuple called the $Parameter$ tuple. The parameter tuple is defined as follows in Equation 3.3 [48]:

$$Parameter = \langle parameter\ name, parameter\ type \rangle \quad (3.3)$$

In the $Method$ tuple, the $Parameter$ tuple is replaced with a attribute CCT if the parameter type is not a base type. The $Method$ tuple is defined as follows in Equation 3.4 [48]:

$$Method = \langle meth\ name,\ ret\ type,\ visibility,\ \{[pre]\}, \{[post]\}, \{Parameter\}\rangle \quad (3.4)$$

The *meth name* is the name of the corresponding method, the *ret type* is the method return type, the *visibility* refers to the the level of protection (public, private, or protected) of the method, the *pre* is the set of OCL preconditions associated with the method, the *post* is the set of OCL postconditions associated with the method, and the $\langle\, Parameter \,\rangle$ tuple represents all of the information about the parameters for the method.

According to Pilskalns et al. [47] one CCT is created for each class used in a sequence diagram. A "non–homomorphic" mapping may occur when not all classes and relationships are used. In our qualification approach we use the CCT to collect coverage and fitness metrics and to determine if a candidate component matches and fits a particular design.

3.3.2 The Object Method Directed Acyclic Graph

The second step in the Integrated UML Model is to define the Object Method Directed Acyclic Graph (OMDAG). The OMDAG is created from the mapping of behavioral information in a sequence diagram to a directed acyclic graph. The original purpose of the OMDAG was to facilitate test sequence generation. Now the OMDAG is used in the Integrated UML Model as an intermediate step for harnessing the information needed to define coupling metrics. The OMDAG is a tuple $\angle V, E, s\rangle$, where V is a set of nodes or vertices, E is a set of edges, and s is the starting node in the graph [48]. Each of the nodes is defined by a tuple $v = \langle o, m, lifeline, \{ARGS\}, [cond], c\rangle$. The o is an object, the m is the method call, the $lifeline$ is the classification of the object as new, exists, or deleted, $ARGS$ is the set of arguments, $[cond]$ is the conditional expression, and c is the class name. The $ARGS$ parameter is a triple: $\langle type, name, value\rangle$. The algorithm for creating an OMDAG is found in [47]. Once

the OMDAG is created, it can be combined with Constrained Class Tuples (CCTS) to form the Constrained Object Method Directed Acyclic Graph (COMDAG).

3.3.3 COMDAG

The last step in the UML integrated model is to combine both the behavioral information found in the OMDAG with the structural information found in the CCT to form the Constrained Object Method Directed Acyclic Graph (COMDAG). Pilskalns et al. describe an algorithm for combining the two pieces of information in [48]. The COMDAG is now used for defining coupling metrics needed during the qualification process.

3.4 Building the Qualification Model

The third step in our qualification process is to build the Qualification Model needed to determine the overall fitness of a candidate component in the design of a system. This step applies Set Theory concepts to the information found in the Integrated UML Model. Essentially this step extracts information from the CCT of each the design components and candidate components to determine if the candidate components properly fit or satisfy the design components. The information that is extracted from the CCTs forms a set of attributes and methods. Thus, we apply set operations to collect coverage and fitness metrics.

3.4.1 Set Theory Definitions

We apply Set Theory to derive metrics about the fitness of a candidate component for a design component. The basic building blocks of the theory are derived from

[18]. In this section, set theory definitions are described for software components. For the definitions we will assume that we have a design component A and a generic candidate component B. A *design component* is defined in terms of a component that is designed for a particular system. A *candidate component* is a component in a repository or a COTS component that may satisfy the design component. For our approach we also assume that one–to–many CCTs define a component.

Elements and Sets

We will define a set on attributes and methods described in the CCT. We perform extraction operations on the sets in Section 3.4.3. Attribute and method *sets* are collections of attribute and method elements. Table 3.3 lists the attribute and method sets. An *attribute element* is defined as a collection of atomic values. The atomic values, attribute type and invariant, are derived through the attribute extraction operation described in section 3.4.3. A *method element* is also defined as a collection of atomic values. The atomic values, return type, invariant, and parameter, are derived through the method extraction operation also described in Section 3.4.3. The following notation will be used to describe an element that belongs to a particular set (Equation 3.5):

$$attribute\ a \in component\ X\ or\ method\ m \in component\ X \tag{3.5}$$

The following notation will be used to describe an element that does not belong to a particular set (Equation 3.6):

$$attribute\ a \notin component\ X\ or\ method\ m \notin component\ X \tag{3.6}$$

Since set theory is applied to components as a technique for qualification, the order of the elements in the design component A and the candidate component B does not matter.

Reference #	Identifier	Definition
1	CCT(class name)	$\langle\{\langle Parent\ CCT\rangle\}, \{\langle Attribute\rangle\}, \{\langle Method\rangle\}, \{[invariant]\}\rangle$
2	Attribute	$\langle aname, attribute\ type, visibility, initial\ value, \{[invariant]\}\rangle$
3	Method	$\langle mname, return\ type, visibility, \{[pre]\}, \{[post]\}, \{Parameter\}\rangle$
4	$\{Attributes\}$	$\{\langle aname_1, attributetype_1, avisibility_1, initialvalue_1, \{[invariant_1]\}\rangle, \langle aname_2, attributetype_2, avisibility_2, initialvalue_2, \{[invariant_2]\}\rangle, \cdots, \langle aname_m, attributetype_m, avisibility_m, initialvalue_m, \{[invariant_m]\}\rangle\}$
5	$\{Methods\}$	$\{\langle mname_1, returntype_1, visibility_1, \{[pre_1]\}, \{[post_1]\}, \{Parameter_1\}\rangle, \langle mname_2, returntype_2, visibility_2, \{[pre_2]\}, \{[post_2]\}, \{Parameter_2\}\rangle, \cdots, \langle mname_n, returntype_n, visibility_n, \{[pre_n]\}, \{[post_n]\}, \{Parameter\}\rangle\}$

Table 3.3: A constrained class tuple and its elements.

Subsets and Venn Diagrams

Assume we have a design component A with an attribute set A_{attr} and a method set B_{meth} and have selected a candidate component B with an attribute set B_{attr} and a method set B_{meth}. Venn diagrams will be used to illustrate the relationships between the attribute and method sets. A_{attr} is a *subset* of B_{attr}, written as in Equation 3.7:

$$AttributeSubset ::= A_{attr} \subseteq B_{attr}, \tag{3.7}$$

if, and only if, every attribute element in A_{attr} is also an attribute element in B_{attr}. The above subset indicates that the candidate component B contains all of the attribute elements needed to satisfy the design of component A. If $B_{attr} \subseteq A_{attr}$, then all of the attribute elements in design component A may not be available in the candidate component B. However, all of component $B's$ attribute elements match attribute elements of component A. Equation 3.7 is presented as a Venn diagram in

Figure 3.4.

A_{meth} is a subset of B_{meth}, written as in Equation 3.8:

$$MethodSubset ::= A_{meth} \subseteq B_{meth}, \tag{3.8}$$

if, and only if, every method element in A_{meth} is also an method element in B_{meth}. The above subset indicates that the candidate component B contains all of the method elements needed to satisfy the design of component A. If $B_{meth} \subseteq A_{meth}$, then all of the method elements in design component A may not be available in the candidate component B. However, all of component $B's$ method elements match method elements of component A. Equation 3.8 is presented as a Venn diagram in Figure 3.5.

A_{attr} is a *proper subset* of B_{attr}, written as in Equation 3.9:

$$ProperAttributeSubset ::= A_{attr} \subset B_{attr}, \tag{3.9}$$

if, and only if, every attribute element in A_{attr} is also an attribute element in B_{attr}, but there is at least one attribute element in B_{attr} that is not in A_{attr}. The above relationship indicates that the candidate component B contains all of the attribute elements needed to satisfy the design of component A. If $B_{attr} \subseteq A_{attr}$, then all of the attribute elements in design component A may not be available in the candidate component B. However, all of component $B's$ attribute elements match attribute elements of component A. Equation 3.9 is presented as a Venn diagram in Figure 3.6. Notice that the same relationship described in Equation 3.9 holds for A_{meth} and B_{meth}.

A_{attr} is *not a subset* of B_{attr}, if, and only if, there exists at least one attribute element in A_{attr} that is not an attribute element in B_{meth}. This relationship indicates that component A has attribute requirements that cannot be fulfilled by component B. Refer to Figure 3.7. If B_{attr} is not a subset of A_{attr}, then it is possible that all attribute elements in B_{attr} match all of the attribute elements in A_{attr}.

Set Equality

Ideally during selection of candidate components for fitness, all attribute sets and methods sets of the candidate component B are exactly the same as the attribute sets and method sets of the design component A. Attribute set equality is defined to examine this idea. A_{attr} *equals* B_{attr}, written in Equation 3.10:

$$AttributeSetEquality ::= A_{attr} = B_{attr} \tag{3.10}$$

if, and only if, every attribute element in A_{attr} is in B_{attr} and every attribute element in B_{attr} is in A_{attr}. The relationship in Equation 3.10 is illustrated in Figure 3.8.
We will also define method set equality to examine the above idea. A_{meth} *equals* B_{meth}, written in Equation 3.11:

$$MethodSetEquality ::= A_{meth} = B_{meth} \tag{3.11}$$

if, and only if, every method element in A_{meth} is in B_{meth} and every method element in B_{meth} is in A_{meth}. The relationship in Equation 3.11 is illustrated in Figure 3.9.

3.4.2 Operations on Sets

In this section the operations that may be performed on attribute and method sets are defined and described. Imagine a universal set U which contains all attribute and method elements. The *attribute union* of two attribute sets A_{attr} and B_{attr}, written in Equation 3.12:

$$AttributeUnion ::= A_{attr} \cup B_{attr} \tag{3.12}$$

is the set of all attribute elements a in U such that a is in A_{attr} or a is in B_{attr}. The shaded region in Figure 3.10 represents $A_{attr} \cup B_{attr}$.

Next the *method union* is defined. The method union of two method sets A_{meth} and B_{meth}, written in Equation 3.13:

$$MethodUnion ::= A_{meth} \cup B_{meth} \tag{3.13}$$

is the set of all method elements m in U such that m is in A_{meth} or m is in B_{meth}. The *attribute intersection* of two attribute sets A_{attr} and B_{attr}, written in Equation 3.14:

$$AttributeIntersection ::= A_{attr} \cap B_{attr} \tag{3.14}$$

is the set of all attribute elements a in U such that a is in A_{attr} and a is in B_{attr}. The shaded region in Figure 3.11 represents $A_{attr} \cap B_{attr}$.

Next the *method intersection* is defined. The method intersection of two method sets A_{meth} and B_{meth}, written in Equation 3.15:

$$MethodIntersection ::= A_{meth} \cap B_{meth} \tag{3.15}$$

is the set of all method elements m in U such that m is in A_{meth} and m is in B_{meth}. The *attribute difference* between two attribute sets A_{attr} and B_{attr}, written in Equation 3.16:

$$AttributeDifference ::= A_{attr} - B_{attr} \tag{3.16}$$

is the set of all attribute elements a in U such that a is in A_{attr} and a is not in B_{attr}. The shaded region in Figure 3.12 represents $A_{attr} - B_{attr}$.

Next the *method difference* is defined. The method difference between two method sets A_{meth} and B_{meth}, written in Equation 3.17:

$$MethodDifference ::= A_{meth} - B_{meth} \tag{3.17}$$

is the set of all method elements m in U such that m is in A_{meth} and m is not in B_{meth}. The *attribute complement* of A_{attr}, written in Equation 3.18:

$$AttributeComplement ::= A^c_{attr} \tag{3.18}$$

is the set of all attribute elements a in U such that a is not in A_{attr}. The shaded region in Figure 3.13 represents A^c_{attr}.

The *method complement* of A_{meth}, written in Equation 3.19:

$$MethodComplement ::= A^c_{meth} \tag{3.19}$$

is the set of all method elements m in U such that m is not in A_{meth}.

The notation defined will now be used throughout the rest of the thesis.

3.4.3 Qualification Measures

The component qualification measures rely on the extraction of information from the CCTs defined. Table 3.3 shows the definition of the CCT as specified in [47]. Here all parts of this definition are not needed as it is not necessary to generate and execute test cases.

The approach determines if an implemented candidate (COTS) component qualifies for the current design and architecture of the system. The analysis is based on comparing the interfaces of the design component as defined in Section 3.3 with a list of implemented candidate components. Interfaces are described in terms of information about the methods, parameters, and attributes as contained in the Constrained Class Tuples (CCT) in Table 3.3.

Interface analysis determines if an implemented candidate component B satisfies the requirements of the system as represented by design component A. The set of attributes and methods is described in the form shown in rows 4 and 5 of Table

3.3, respectively. We assume that a component X may contain multiple attributes and methods, hence many different attribute and method signatures. An interface is defined as comprising multiple attribute and method signatures.

Step 1: The first step to ascertaining that a candidate component is sufficient for a designed component is to extract the necessary information from the signatures of both the designed and candidate components. Information must be extracted from both the method and attribute sets, X_{meth} and X_{attr} of component X, respectively. X is defined as $X = X_{attr} \cup X_{meth}$ similar to Equation 3.12.

Attribute Extraction, $Attr(X)$, is a function that extracts pertinent attribute information, for component qualification from component X. Component X contains a set of attributes of the form seen in Table 3.3 row 4. $Attr(X)$ returns the set of all (attribute type, invariant) pairs of component X. We define a metric $\#Attr(X)$ which is equivalent to $|Attr(X)|$.

Method Extraction, $Meth(X)$, extracts pertinent method information, for component qualification from component X. Component X contains a set of methods of the form given in Table 3.3 row 5. $Meth(X)$ returns the set of all (return type, invariant, parameter) triples of component X. We define a metric $\#Meth(X)$ which is equivalent to $|Meth(X)|$.

The set of method triples Meth(X) and set of attribute pairs Attr(X) of a component are called the signature of component X.

Step 2: Next we determine if the interface of a candidate component B *matches* the interface of a designed component A. Each method triple and attribute pair of A is compared to each method triple and attribute pair of B. A *complete match* is found if the signature of the method or attribute in A has the same return type, invariant, and parameter tuple or attribute type and invariant of the method or attribute, respectively, in B. A *partial match* is found if, and only if, some of the elements of

the method or attribute signature in A match the method or attribute signature in B. For a match we need to recursively search through $A's$ CCT and Parent CCT for methods and attributes whose signatures match the signatures of B (refer to Table 3.3 row 1).

Step 3: The third step is to determine if the interface of a candidate component B *exceeds* the interface of a design component A. The interface of B *safely exceeds* the interface A if it contains enough methods and attributes to match all signatures of methods and attributes in A. Essentially the signature of A, as defined by its attribute pairs Attr(A) and method triples Meth(A), must be a proper subset of the signature of B.

This helps to determine whether or not a given design that requires attributes described by component A can be satisfied by candidate component B. For example, imagine a component A which requires five cards to represent a poker hand. Each of the cards is represented as a String. If candidate component B contains six cards represented as Strings, then component A is a proper subset of component B. If the circles in Figure 3.6 indicate the attribute space for a given component, then Venn diagrams may be defined which visualize the overhead of two components. The shaded area indicates the overhead.

Step 4: We determine a qualification measure. We need to perform operations on sets of attributes and methods of components in order to determine if a candidate component satisfies the requirements. We perform weighted operations on both the attribute and method sets of components. The attribute sets are weighted w_a and the method sets are weighted w_m according to the Analytic Hierarchy Process (AHP) [51]. This reflects relative importance of matching specific parts of a component's signature. In the following sections, the fitnesses and coverages computed for both attribute and method sets are also weighted by AHP.

Given two components A and B, with attributes $A_{attr} = \{aa_1, aa_2, aa_3, \cdots, aa_m\}$ and $B_{attr} = \{ba_1, ba_2, ba_3, \cdots, ba_n\}$ respectively, the *difference* between component $A's$ and component $B's$ attributes is defined in Equation 3.20.

$$\#attrOver(B, A) = |Attr(B) - Attr(A)| \tag{3.20}$$

The difference in Equation 3.20 is the number of attribute elements in $Attr(B)$, but not in $Attr(A)$.

The difference between two components and their attributes is considered the *attribute overhead* of the component. For example the design of a poker game needs a component A which has five cards all represented by Strings,

$$\{card_1, card_2, card_3, card_4, card_5\}$$

Consider a candidate component B which contains six cards all represented by Strings, and a game type represented as an integer number,

$$\{card_1, card_2, card_3, card_4, card_5, card_6, game\_type\}.$$

Applying $Attr(B) - Attr(A)$ (Equation 3.16) to the two components results in:

$$\#attrOver = \{card, game\_type\}$$

The difference in this case represents the *attribute overhead* of the components. Representing the attribute overhead as a percentage is desired, refer to Equation 3.21.

$$attrOver\% = \frac{\#attrOver}{\#Attr(B)} * 100 \tag{3.21}$$

For this example the percentage of overhead is $2/7 = 29\%$.

The difference between two components is shown in Figure 3.12. The shaded area indicates the difference between $Attr(B)$ and $Attr(A)$.

Given two components A and B, with methods $\{am_1, am_2, am_3, \cdots, am_m\}$ and $\{bm_1, bm_2, bm_3, \cdots, bm_n\}$ respectively, the difference between component $A's$ and component $B's$ methods is defined in Equation 3.22.

$$\#methOver(B, A) = |Meth(B) - Meth(A)| \tag{3.22}$$

The method difference is the number of method elements in $Meth(B)$, but not in $Meth(A)$. The difference between two components and their methods is considered the *method overhead* of the component. For example, the design of a poker game needs a component A which has a deal and shuffle method, $\{getHand(), shuffle()\}$. If a candidate component B is available which contains methods for getHand( ), shuffle( ), and getBet(), $\{getHand(), shuffle(), getBet()\}$, then applying $Meth(B) - Meth(A)$ (Equation 3.17) to the two components results in $\#methOver = \{getBet()\}$. The method overhead of the components as a percentage is defined in Equation 3.23.

$$methOver\% = \frac{\#methOver}{\#Meth(B)} * 100 \tag{3.23}$$

For this example the percentage of overhead is $1/3 = 33\%$.

The *intersection* of two components A and B, with attributes $\{aa_1, aa_2, aa_3, \cdots, aa_m\}$ and $\{ba_1, ba_2, ba_3, \cdots, ba_n\}$ respectively, is defined in Equation 3.24.

$$\#attrInt(A, B) = |Attr(A) \cap Attr(B)| \tag{3.24}$$

The attribute intersection is the number of attribute elements in both $Attr(A)$ and $Attr(B)$.

The intersection between two components indicates coverage. Thus all attribute elements that are present in $Attr(A)$ and in $Attr(B)$ represent the *attribute coverage*. Attribute coverage can be calculated as in Equation 3.25.

$$attrCov\% = \frac{\#attrInt}{\#Attr(A)} * 100 \tag{3.25}$$

For example, the design of a poker game needs a component A which has five cards all represented by Strings, $\{card_1, card_2, card_3, card_4, card_5\}$. If a candidate component B is available which contains six cards all represented by Strings, and a game type represented as an integer number ($\{card_1, card_2, card_3, card_4, card_5, card_6, game\_type\}$), then applying $Attr(A) \cap Attr(B)$ (Equation 3.14), results in:

$$I = \{card_1, card_2, card_3, card_4, card_5\}$$

Thus, $|Attr(A)| = 5$ and $\#attrInt = 5$, which indicates that the coverage is $1 - (5 - 5) = 100\%$. However, the attribute coverage must be weighted by w_{ac}. The *attribute fitness* of the component is defined in Equation 3.26.

$$attrFit\% = (1 - \frac{attrOver\%}{100}) * 100 \tag{3.26}$$

The attribute fitness is $1 - 0.29$ or 71%. The attribute fitness must also be weighted by w_{af}. The Venn diagram in Figure 3.11, illustrates by the shaded area, the intersection of attributes between two components.

The intersection of two components A and B, with methods $\{am_1, am_2, am_3, \cdots, am_m\}$ and $\{bm_1, bm_2, bm_3, \cdots, bm_n\}$ respectively, is defined in Equation 3.27.

$$\#methInt(A, B) = |Meth(A) \cap Meth(B)| \tag{3.27}$$

Where method intersection is the number of method elements in both $Meth(A)$ and $Meth(B)$.

The intersection between two components indicates coverage. Thus all method elements that are present in $Meth(A)$ and in $Meth(B)$ represent the *method coverage*. The method coverage is calculated as in Equation 3.28.

$$methCov\% = \frac{\#methInt}{\#Meth(A)} * 100 \tag{3.28}$$

For example, the design of a poker game needs a component A which has a deal and shuffle method, $\{getHand(), shuffle()\}$. If a component B is available which contains methods for getHand( ), shuffle( ), and getBet( ), $\{getHand(), shuffle(), getBet()\}$, then applying $Meth(B) \cap Meth(A)$ (Equation 3.15) to the two components results in $I = \{getHand(), shuffle()\}$. Thus, $|Meth(A)| = 2$ and $\#methInt = 2$, which indicates that the method coverage is $1 - (2 - 2) = 100\%$. However, the method coverage must be weighted by w_{mc} The *method fitness* of the component is defined in Equation 3.29.

$$methFit\% = (1 - \frac{methOver\%}{100}) * 100 \tag{3.29}$$

The method fitness is $1 - 0.33$ or 67%. The method fitness must also be weighted by w_{mf}.

A component B *properly satisfies* a component A if and only if the fitness and coverage is x for both attributes and methods. A component B *satisfies* a component A if and only if the fitness and coverage is at least y for both attributes and methods, where $x > y$ and $x \leq 50\%$. Note the y value is set by the designer. If the fitness and coverage for both attributes and methods is less than y, then a different component should be considered. The sum of attribute fitness and attribute coverage should be as close to 100% as possible to ensure that a given component *attribute qualification* satisfies the requirements of the system. The sum of method fitness and method coverage should be as close to 100% as possible to ensure that a given component *method qualification* satisfies the requirements of the system. Adding the results of weighted attribute qualification and the weighted method qualification measures, results in the *overall qualification* of the component for the system. The weights of the measures are calculated according to the Analytic Hierarchy Process (AHP).

3.4.4 Applying the Analytic Hierarchy Process (AHP)

The Analytic Hierarchy Process (AHP) is necessary for determining the weights applied to the attribute and method qualification measures. The AHP has four steps for determining the relative importance of factors and a fifth step for checking the consistency of the results [51] and [55].

Step 1: Create a square $n \times n$ preference matrix labeled, N, where n is the number of qualification variables. Element n_{ij} is the relative importance between variables i and j. For our Qualification Method, there are four qualification variables ($n = 4$). The variables in the matrix are defined as follows: (1) attrFit%, (2) attrCov%, (3) methFit%, and (4) methCov%.

Step 2: Four qualification variables require six pairwise comparisons. Table 3.4 shows the six pairs and the values in the comparison using AHP. Table 3.4 also shows which of the qualification variables in the comparison is more important along with the value of the entry in the preference matrix. The importance of the variables is based on a subjective measurement. During the qualification of components in the design phase we have decided that the coverage of the component is more important than the fitness of the component. This is because having enough methods ($methCov\%$) and attributes ($attrCov\%$) to fulfill the design requirements is more important than having too many methods ($methFit\%$) or attributes ($attrFit\%$) that do not fulfill the design of the component. We have also decided that attributes are more important in a design than methods. Our decision stems from the notion that if the attributes are inherent in the candidate component, then we can always add methods later. However, if the attributes are not contained within the available component, then the methods within that component that manipulate the attributes will obviously not satisfy the needs of the design component. The results of the pairwise comparison

are shown in Table 3.5.

Pair	**Value**	**More Important**
(1) attrFit% vs. attrCov% (2)	5	attrCov%: $n_{21} = 5$, $n_{12} = 1/5$
(1) attrFit% vs. methFit% (3)	2	attrFit%: $n_{13} = 2$, $n_{31} = 1/2$
(1) attrFit% vs. methCov% (4)	4	methCov%: $n_{41} = 4$, $n_{14} = 1/4$
(2) attrCov% vs. methFit% (3)	5	attrCov%: $n_{23} = 5$, $n_{32} = 1/5$
(2) attrCov% vs. methCov% (4)	2	attrCov%: $n_{24} = 2$, $n_{42} = 1/2$
(3) methFit% vs. methCov% (4)	4	methCov%: $n_{43} = 4$, $n_{34} = 1/4$

Table 3.4: Pairwise Comparison of Qualification Variables.

1	1/5	2	1/4
5	1	5	2
1/2	1/5	1	1/4
4	1/2	4	1

Table 3.5: N Preference Matrix

Step 3: Compute the eigenvector of N. The method proposed by Saaty, in 1980 [51], suggests that the following steps should be followed:

- Calculate the sum of the columns in the N preference matrix, $n_j = \Sigma_{i=1}^{n} n_{ij}$. The sum of columns 1–4 are 10.5, 1.9, 12, and 3.5, respectively.

- Divide each element in a column by the sum of the column, $m_{ij} = n_{ij}/n_j$. The result is in Table 3.6. The new elements are denoted by m_{ij}.

0.095	0.105	0.167	0.071
0.476	0.526	0.417	0.571
0.048	0.105	0.083	0.071
0.381	0.263	0.333	0.286

Table 3.6: M Matrix

- Calculate the sum of each row in the new M matrix in Table 3.6, $m_i = \Sigma_{j=1}^{n} m_{ij}$. The sum of each row, 1–4, in Table 3.6 is 0.438, 1.990, 0.307, and 1.263, respectively.

- The sums of the rows are normalized by dividing by the number of qualification variables n, $P_i = m_i/n$. P is called the priority vector and contains estimates of the eigen values of the matrix in Table 3.6. In this case $P = (0.11, 0.50, 0.08, 0.32)$, for elements P_i, where $i = 1, \cdots, n$.

Step 4: The priority vector, P, indicates that attrFit% accounts for 11% of the qualification variables' importance. The other three factors attrCov%, methFit%, and methCov% account for 50%, 8%, and 32% of the qualification variables' importance, respectively.

Step 5: This step computes the consistency index CI of the eigen values. The first step is to multiply the N preference matrix with the priority vector P ($R = N * P$),R = (0.45, 2.09, 0.32, 1.33). The λ vector is calculated by dividing each element in R with the corresponding element in the priority vector P. In this case $\lambda = (4.09, 4.18, 4.00, 4.16)$. λmax is the average of the values in the λ vector. In particular λmax = 4.11. The consistency index is:

$$CI = \frac{\lambda max - n}{n - 1} = \frac{4.11 - 4}{3} = 0.04 \tag{3.30}$$

The consistency ratio is computed by dividing CI by the RI value for $n = 4$. $RI = 0.90$. The RI can be found in [51]. The CR value is thus $CR = 0.04/0.90 = 0.04$. According to [51] and [55], the CR value is considered acceptable (< 0.10). The weights applied to the attribute qualification (Equation 3.31) and the method

qualification (Equation 3.32) are thus found in the priority vector P. The overall qualification is found in Equation 3.33.

$$attribute\ qualification(B) = 0.11 * attrFit\% + 0.50 * attrCov\% \quad (3.31)$$

$$method\ qualification(B) = 0.08 * methFit\% + 0.32 * methCov\% \quad (3.32)$$

$$overall\ qualification(B) = attribute\ qualification + method\ qualification \quad (3.33)$$

3.4.5 Adjusting Coverage and Fitness Measures to Accommodate Complex Types

When calculating the attribute overhead which directly affects the attribute fitness, we are not only concerned with the percentage of overhead, but also with classifying the complexity of attributes. We would like to identify which attributes lying in the overhead region are in fact complex types and which are simple or base types. For example, if the candidate component B contains extra aggregate attributes such as a list or vector we would like to differentiate the attribute overhead of this component with one that has extra simple types such as string or int. A weight could be applied to the overhead for every extra attribute based on the classification of the attribute. The weight is determined by dividing the attributes into 3 classes defined on an ordinal scale. The *attribute classes* are listed in Table 3.7.

Classification Value	Classification Name
1	SIMPLE
2	STANDARD
3	USERDEFINED

Table 3.7: Attribute Classes.

The SIMPLE classification includes all attributes of type integer (int), real (float), character (char), etc. The STANDARD classification includes all attributes the are

defined in standard libraries such as the file type (FILE), size type (sizet), etc. Lastly, the USERDEFINED classification includes all attributes that may be defined by the programmer such as lists, stacks, queues, vectors, etc. The weights can be used to adjust the overhead or coverage measure. For example, 10 attributes of a candidate component $B's$ 100 attributes match the attributes for a design component A. We know according to Equation 3.21 that the $attrOver\% = 90/100 = 90\%$. Assume that of the extra 90 attributes 80 are SIMPLE, 5 are STANDARD, and 5 are USERDEFINED. With the introduction of the weights for the classification of attributes we could adjust the $attrOver\%$ by multiplying the excess attributes by a normalized value of the classification value. The equation for applying the weights is supplied in Equation 3.34.

$$Attribute\ Weight = 1 + |\frac{1 - Classification\ Value}{MAXCLASSIFICATION}| \tag{3.34}$$

where MAXCLASSIFICATION is the largest classification value that can be obtained. In our scale the largest classification value is 3 or USERDEFINED. After the weights of the classifications are determined, then the number of attributes of type SIMPLE, STANDARD, or USERDEFINED are multiplied by the corresponding weights. For our example, the new $attrOver\% = \frac{80*1+5*1.333+5*1.667}{100} = 95\%$. If the max value of the new $attrOver\%$ cannot exceed 100%.

The attribute coverage may also be adjusted to indicate that one candidate component B covers more complex attributes than another component C. The attribute coverage may need to be adjusted when the attrCov% measure returns the same number for both component B and C. In this case if component B covers the same number of attributes as C, but covers more complex attributes, then the new measure with the new weights will indicate that B is more qualified than C for the particular design.

Not only can weights be applied to adjust the attribute coverage and fitness measures, but weights can be applied to adjust the method coverage and fitness measures in order to accommodate for discrepancies in types. Equation 3.34 can be applied to methods also. However, the classifications of method types is slightly different. The *method classes* are shown in Table 3.8.

Classification Value	Classification Name
1	SIMPLE
2	STANDARD
3	REFSTANDARD
4	USERDEFINED

Table 3.8: Method Classes.

The classifications in Table 3.8 are similar to Table 3.7 except the REFSTANDARD represents the types that are referenced, e.g. a method that has a return type of a pointer to a variable.

3.4.6 Detecting Architecture Co–Evolution

Xing and Stroulia [57] have developed a method for detecting changes between different versions of Class Diagrams. They have envisioned a tool that reveals discrepancies between the implemented design and the desired design. The tool also allows for an approximation of the amount of work that has been accomplished by comparing two snapshots of the Class Models at different points in time. The idea behind the tool is to compare two or more versions of the architecture of a system represented by UML diagrams or XMI code. The tool applies an algorithm that builds change trees. The change trees store information about classes, methods, attributes, and renamings. Successive change trees indicate deletions, additions, and modifications to classes, methods, and attributes in the trees. Information about architectural changes be-

tween versions may be viewed in the change trees.

This thesis is also concerned with detecting differences in the architectures of two entities. It is however not concerned with the evolution of one architecture, but rather with the discrepancies between a design component's architecture and a candidate component's architecture. At present, our qualification measures may be collected via inspections by the designer of the component. However, Xing and Stroulia's tool would allow for measures to be collected through automation. We could walk through the change trees of a design component and a candidate component and collect simple counts about methods and attributes in the trees, which would then allow for the calculation of coverage and fit measures. The tool recognizes matches in types of attributes and parameters of methods. Although the tool is interested in renamings of classes, attributes, and methods, our process ignores names.

3.5 Calculating Coupling Measures

The fourth step in our qualification process is to calculate coupling measures. The measures are calculated by tracing Use Cases through a COMDAG. Use Cases are derived from an operational profile of a software system. The Use Cases may be used to help aid in defining coupling measures for a candidate component.

We define coupling measures by (1) creating a model that merges the static and dynamic information of UML Class and Sequence diagrams and (2) applying an operation profile to the model to collect metrics. The first step in defining coupling measures is to convert the Class Diagrams and Sequence Diagrams into a directed acyclic graph that can be analyzed for coupling (COMDAG). We do this in two steps based on Pilskalns et al. [47]. First we convert classes into constrained class tuples (CCTs) that describe attributes, methods, and inheritance relationships of a class

(Equation 3.1). A CCT is contained in each node of the COMDAG and represents an instantiated class. A set of CCTs can be used to define a component, since all of the interface information is available. The Sequence Diagrams are converted into graphs (COMDAG), starting with the first method call, and following the paths through the Sequence Diagram. Nodes are defined by the method, object, and classes involved. Edges connect method sequences as specified in the Sequence Diagram.

The COMDAG can be constructed by mapping elements of the Sequence Diagrams to a graph. The COMDAG is a triple $< V, E, s >$, where V is a set of vertices, E is the set of edges, and s is the starting vertex. Each vertex, v, is defined by the triple $v = \langle o, \langle M \rangle, CCT(c) \rangle$, where o is an object, $\langle M \rangle$ is a method tuple, c is a class. An edge E, represented by the tuple $\langle v_1, v_2 \rangle$, consists of a pair of vertices that represent the ordering between vertices v_1 and v_2 defined by the sequence diagram.

Before the introduction of the decision making framework, it is necessary to define the coupling measures that are primary factors in the decision making process. By primary factors, we mean the measures that directly affect the outcome of decisions that are made during the design phase. The coupling measures take advantage of the Integrated UML Model introduced in Section 3.3 and in [48]. The measures are indicators of general quality attributes that can be collected during design. The number of connections ($\#conn$) and activations ($\#act$) measured during a walk through the COMDAG defines the coupling of the component.

Once the CCT and COMDAG are defined as in Sections 3.3.1 and 3.3.3, it is possible to define a set of (connected) COMDAG vertices with the smallest number of connections ($\#conn$). To define and design a component and its boundaries select, a set of nodes $\langle A \rangle$ where every node is directly connected to at least one other member in the set (e.g. a connection is define as $\langle v_n, v_{n+1} \rangle$). Since each node contains a CCT, the component contains a complete description of the potential interface. A min–cut

algorithm can be employed or a designer can manually identify potential component boundaries. In addition, we assume an operational profile has been defined for Use Cases; that is, each Use Case is associated with a (relative) frequency. Execution as in [47] or tracing a use case through the COMDAG identifies how often interfaces are activated ($\#act$). A decision on which nodes are part of a defined component are then made based on ($\#conn, \#act$). The designer can then choose either the interface with the fewest connections, or the interface with the fewest activations. Alternatively, the designer may have set a threshold for $\#act$ and then selected the interface with the fewest $\#conn$ that falls within the threshold. The component $\langle A \rangle$ identified with this approach is a subset of the vertices in the COMDAG.

After we collected the number of connections and number of activations, we need to normalize the measures to a scale from 0 to 1. Equation 3.35 shows the *overall coupling* measure:

$$Overall\ Coupling = coupling\% = (w_e * \frac{\#conn}{total\ \#conn} + w_a * \frac{\#act}{total\ \#act}) * 100 \quad (3.35)$$

where $total\ \#conn$ is the total number of connections (edges) between nodes in the COMDAG and $total\ \#act$ is the estimated total number of activations possible for a use case. The weights w_e and w_a are calculated following the AHP described in Section 3.4.4. The weights are assigned according to the relative importance of both the $\#conn$ and $\#act$ and $w_e + w_a = 1$. The coupling should be less than some threshold c_t.

3.6 Ranking the Candidate Components

The candidate components must be ranked based on the results of the qualification and coupling measures. We have three choices on how to rank components (Table

3.9). The first choice is to rank components 1 through n based on the qualification measure, where 1 is the highest ranking and n is the number of candidate components. The second choice is to once again use a ranking of 1 through n, but instead base the ranking on the coupling measure. Thirdly, the rankings of the components could be based on a hybrid of the qualification and coupling measures.

Choice Number	**Choice**
1	Only qualification measures
2	Only coupling measures
3	Hybrid of qualification and coupling measures

Table 3.9: Choices for Ranking Components.

If the first choice is selected, the components are ranked based on the qualification value that is calculated following the Qualification Model. Our model suggests that a candidate component should satisfy a design component with a qualification value at least equal to y (section 3.4.3). If the component does not reach this threshold, the component is automatically discarded. If any of the top components have the same ranking, adjusting the qualification measures to accommodate differences in complex types is desired (section 3.4.5). This should break any ties in rankings of the top components. If the components still have the same qualification measure, the designer should use his/her experience to select the best candidate component from the ones with the same rank for the design.

If the second choice is selected, then the components are ranked based on the coupling metrics collected. The components with the lowest coupling should be considered. For this measure there is no threshold value for which to compare the coupling measure. In general, selecting the best candidate component based on coupling may not be a strong enough measure to guarantee that the component is the best fit for the design. The coupling metrics are most useful for strengthening the case for a

component that has the same qualification measure as another component.

If the third choice is selected then the qualification measure and coupling measures affect the designer's decision about the candidate component. The designer has the choice to weigh the qualification (w_q) or coupling (w_c) measures heavier. If the designer decides the qualification measure is more important than the coupling measure, then $w_q > w_c$. If the designer decides that the coupling measure is more important, then $w_c > w_q$. Otherwise they may be weighted equally $w_q = w_c$. The only requirement is that the weights must satisfy $w_q + w_c = 1$. The weights are calculated using the AHP method. The *hybrid measure* is defined as in equation 3.36:

$$Hybrid\ Measure = w_q * overall\ qualification + w_c * (1 - \frac{coupling\%}{100}) * 100 \quad (3.36)$$

The higher the hybrid measure, the better the component. The highest value for the hybrid measure is 100%. To decide which of the three choices to use to rank components follow the decision making framework in the next Chapter 4. The framework also suggests how to determine the weights of the measures for the third choice.

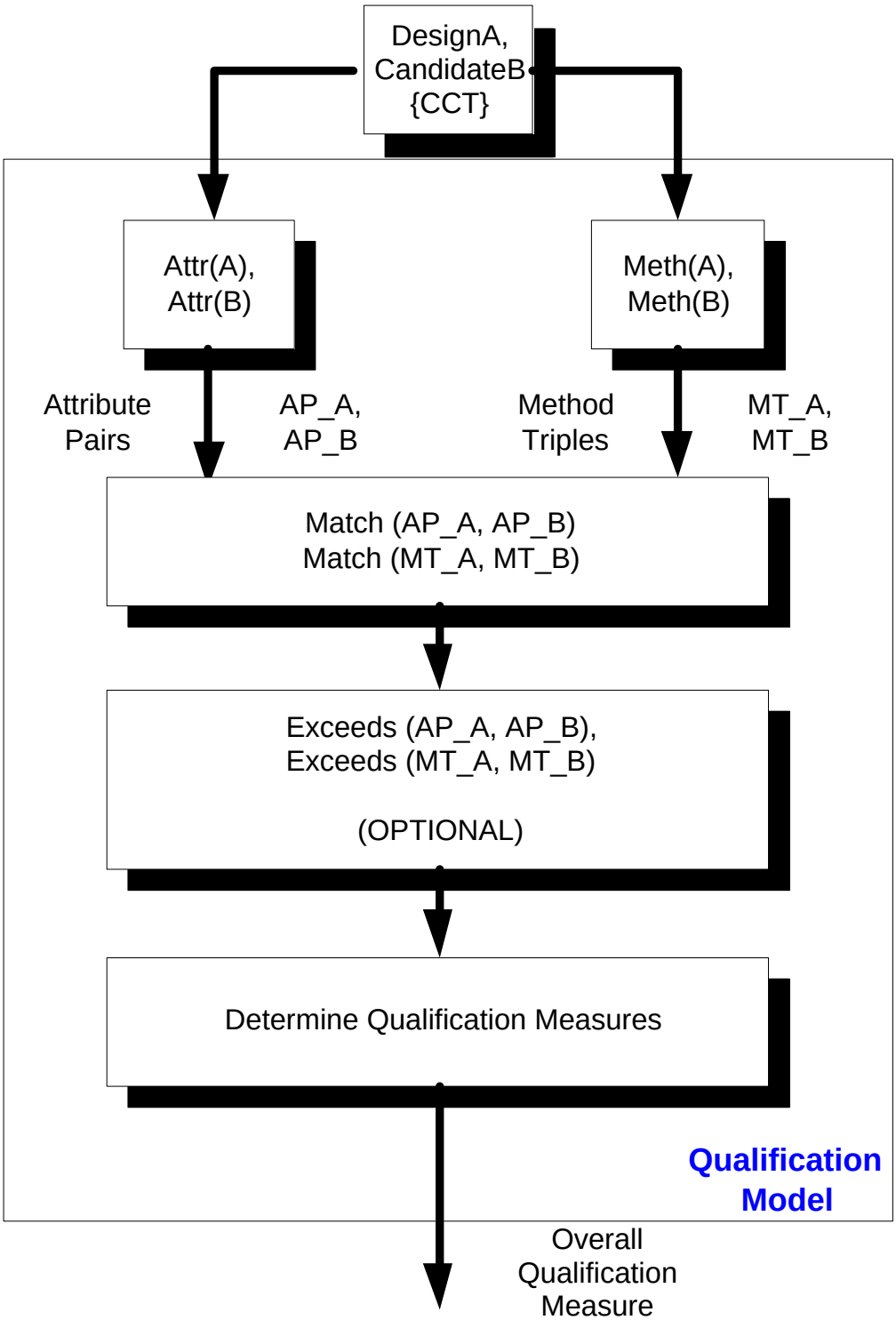


Figure 3.2: The Qualification Model.

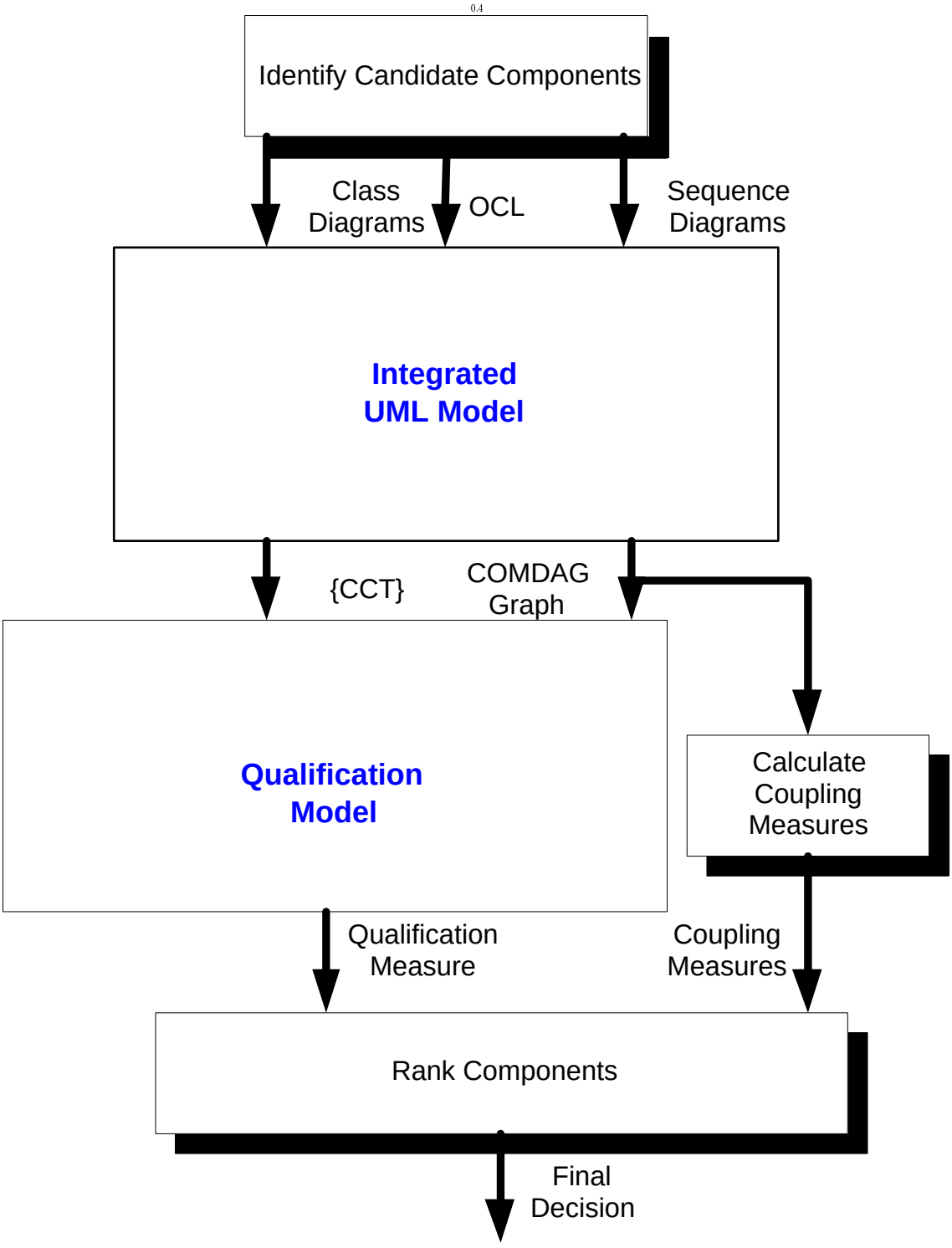


Figure 3.3: The Qualification Approach.

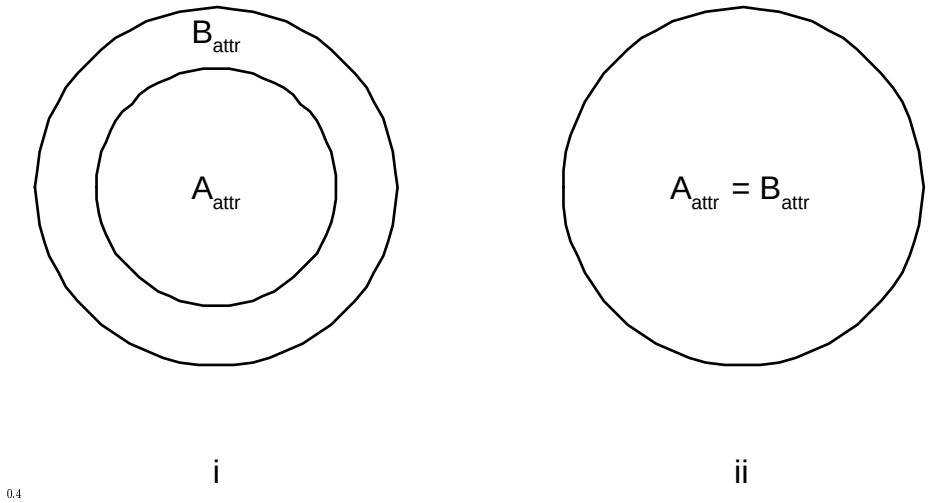


Figure 3.4: Attribute Subset

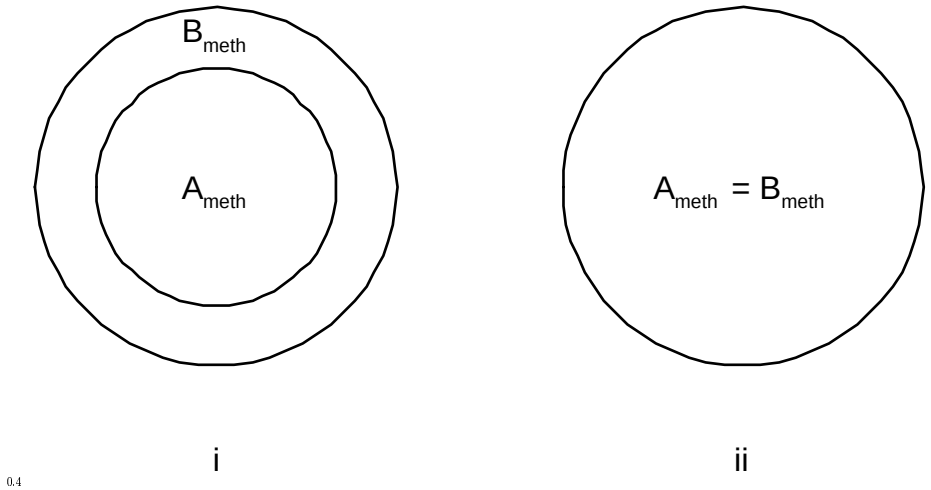


Figure 3.5: Method Subset

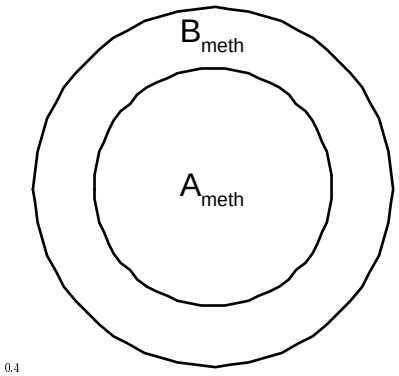


Figure 3.6: Proper Attribute Subset

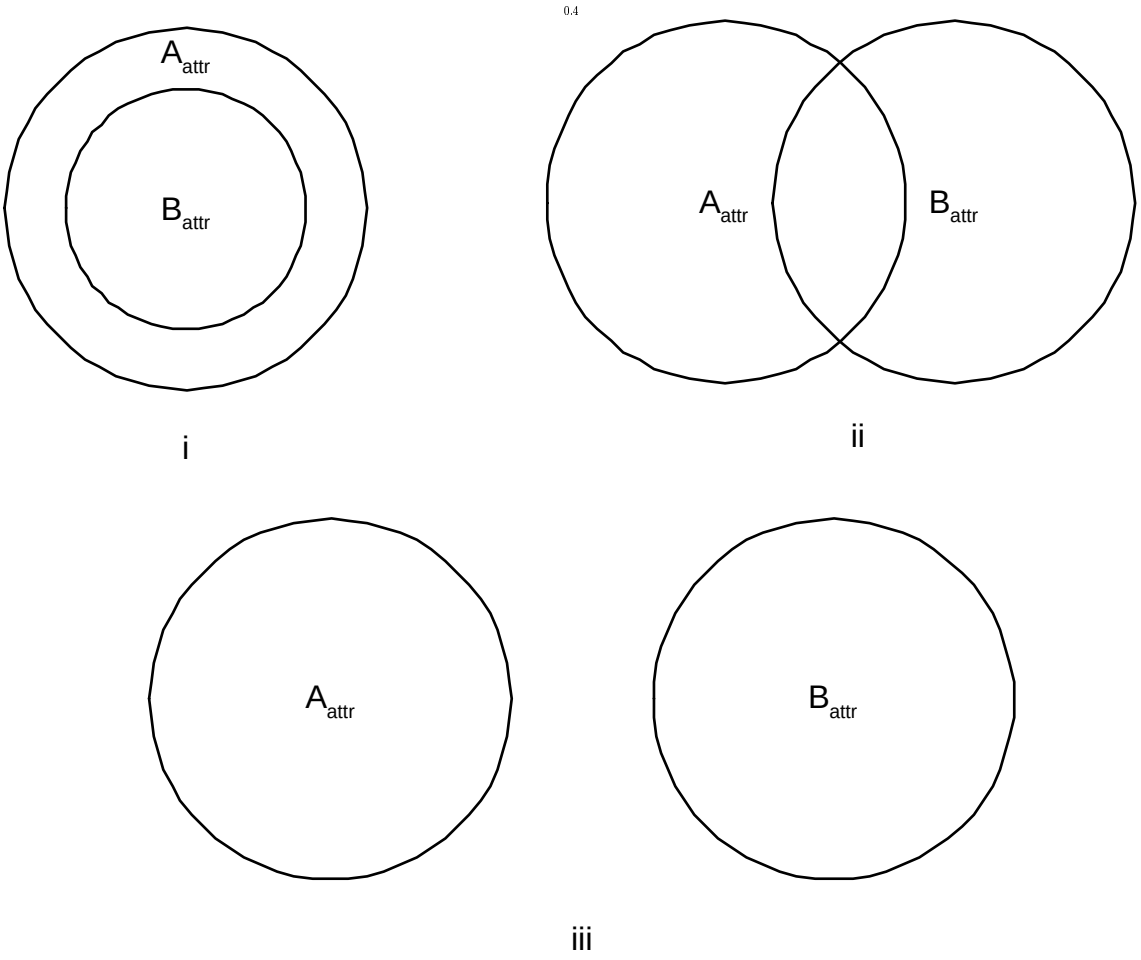
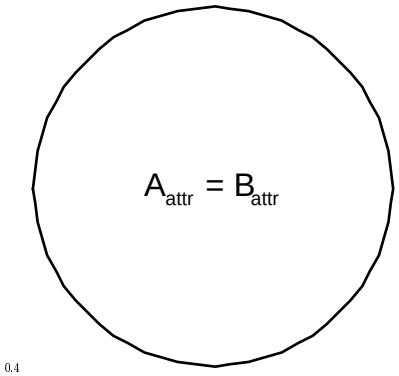
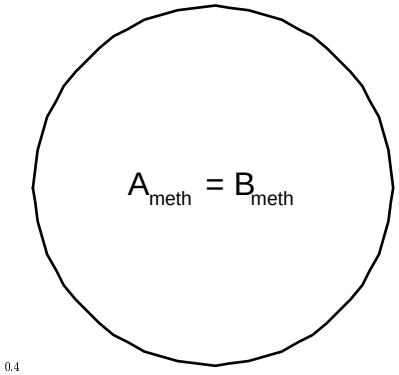


Figure 3.7: Not an Attribute Subset



0.4

Figure 3.8: Attribute Set Equality



0.4

Figure 3.9: Method Set Equality

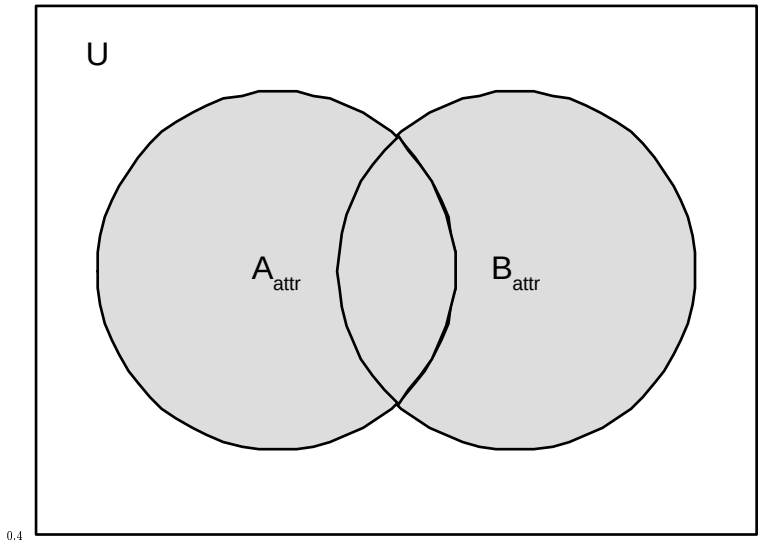


Figure 3.10: Attribute Union

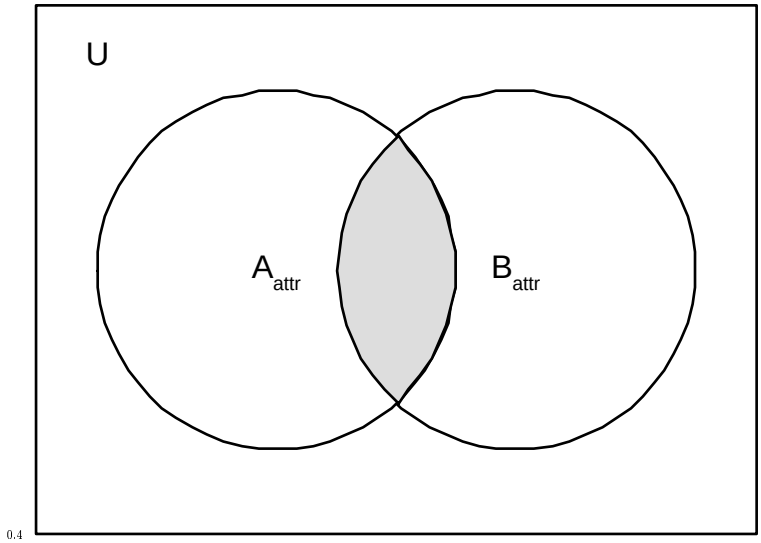


Figure 3.11: Attribute Intersection

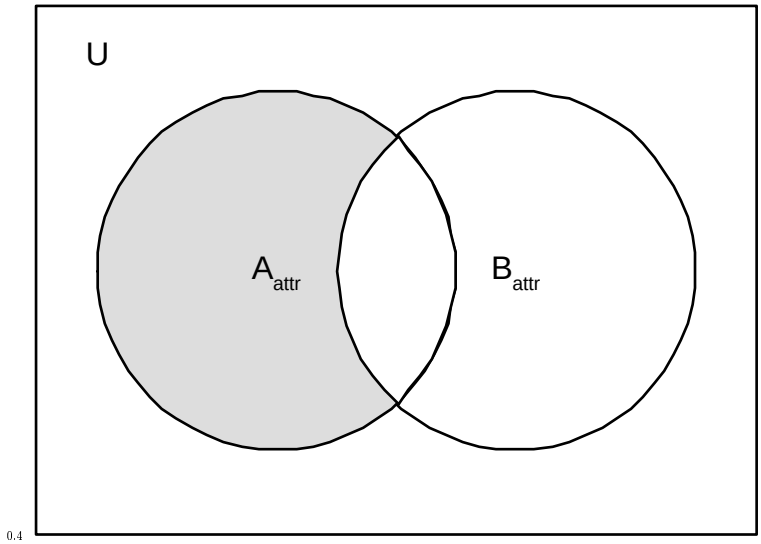


Figure 3.12: Attribute Difference

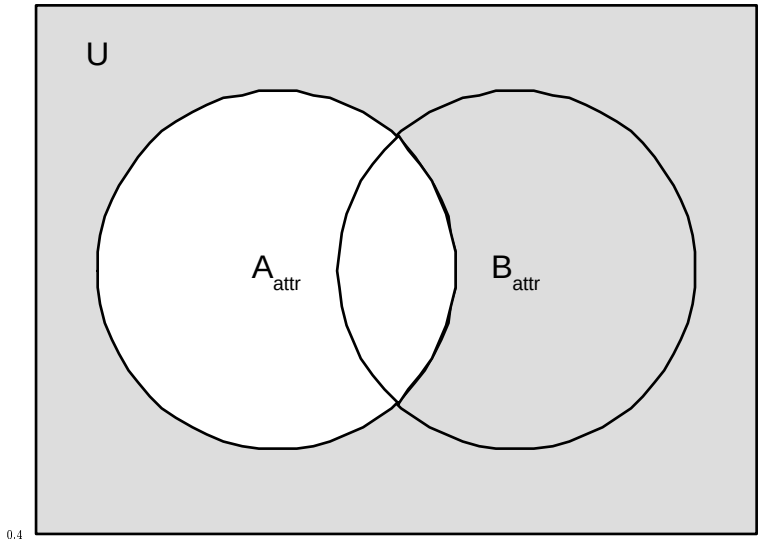


Figure 3.13: Attribute Complement

Chapter 4

Decision Making

Developing component–based software requires that many decisions and trade–offs are made in each of the phases of the lifecycle. During the requirements gathering phase, decisions need to be made about non–functional and functional requirements. During this phase it is necessary to make choices about such requirements as security, performance, and features. During the design phase trade–offs between architecture and available component features are required. During the implementation and testing phases, trade–offs between implementation time and features occur frequently as well as trade-offs between testing time and the number of test cases that are run. The decision making framework presented in this chapter describes how to make decisions on artifacts in the design phase. We will make trade–off decisions between qualification measures of components and coupling measures of components. If a certain component surpasses a threshold for both overall qualification and overall coupling and coupling measures, then trade–off decisions are not required. However, if one of the measures fails to satisfy a particular threshold, then a decision about using the particular component must be made. The software engineer must decide if failing in one of the two areas requires using a completely different component or if the other

measures that have not failed are more important than the ones that did fail. The decision making framework in this chapter will provide the software engineer, with guidelines for making these design choices.

4.1 Decision Making Framework

When determining how to approach the design of a software system, many decisions need to be made. The designer must first decide whether or not to use software components. In order to apply software components to software systems within a company, a well planned business case is required [24]. Risks and benefits of adopting component technology must be well understood within the business case. The costs and payback of using components must also be determined within the business case. Such issues as business goals, organizational readiness, and infrastructure support also help the designer in making a business case for applying components. For this thesis we are not concerned with making decisions for the business case, we assume that we have decided to use components to fill in "gaps" in our software design. Table 4.1 shows the decision making process.

Step	**Decision**
1.	Features of the System
2.	Adoption of Components
3.	Architectural Style
4.	Design Components
5.	Candidate Components
6.	Qualification Measures vs. Coupling Measures
7.	Final Components

Table 4.1: Design Decision Making Process (DDMP).

Item 6 in Table 4.1 is what we are most interested in discussing. We need to

understand how to select the right choice from Table 3.9 for ranking the candidate components. The qualification measures (choice 1) should be used to rank components when coverage and fitness measures are important. The coupling measures (choice 2) should be used when coupling of components is an issue for a particular architecture. The hybrid choice (choice 3) should be selected when the qualification or coupling measures alone do not pass the threshold required to use the candidate component (Section 3.5).

Chapter 5

Application

5.1 Gouraud Polygon

We created a UML designed application to demonstrate the qualification process. The application creates a two-dimensional convex polygon filled with either a solid color or Gouraud shading. The inputs to the program are two filenames: an input file and an image file. The input file contains the size, a list of coordinates, and color values. The image file can have five different formats: BMP, GIF, JPG, PNG, or a TIF.

Use Case: Create Polygon
Intent: Create polygon image from input file
Pre Conditions: Input file is correctly specified
Post Conditions: Polygon image file created
Description:
1. User executes program with 2 command line arguments
2. System returns polygon image file

Figure 5.1: Polygon Use Case

The original Class Diagram contained 14 classes. Figure 5.2 shows a simplified version, with many classes hidden behind the image interface. The Sequence Diagram, Figure 5.3, has been simplified as well. Both diagrams were designed using UML 2.0. The COMDAG, Figure 5.4, was derived from the sequence diagram. The loops in

the COMDAG have been retained to reduce the size, but in practice they would be unrolled into a directed acyclic graph.

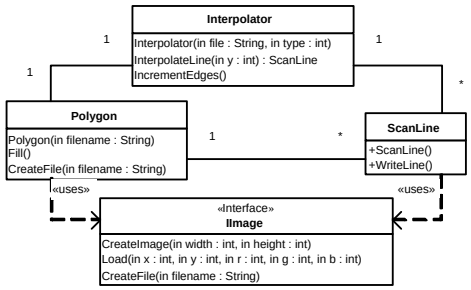


Figure 5.2: Polygon Class Diagram

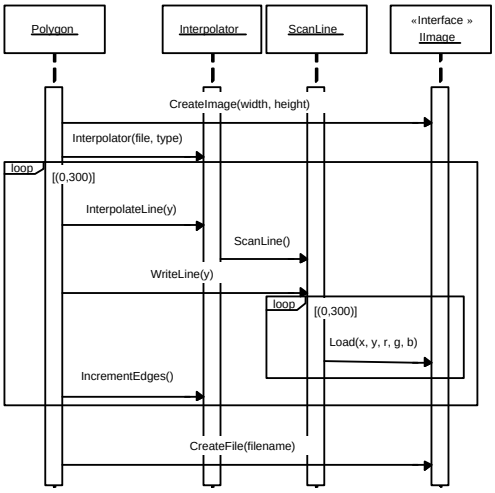
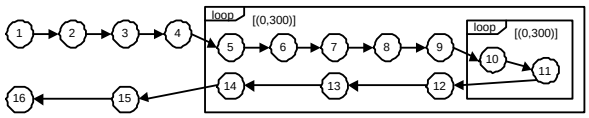


Figure 5.3: Polygon Sequence Diagram

During the collection of coupling measures an Operational Profile consisting of Use Cases and their frequency of execution is used. For this example, we use one of the primary Use-Cases, see Figure 5.1, to simulate execution. There are three nodes directly connected to the Image Component: 1, 10, and 15. Despite only 3 connections, when we simulate the Use-Case there are 90,002 activations.

The purpose of the Image Component is to provide a way for the program to write an image file. We need to be able to specify the image size, the RGB value for each pixel, and create one of five file types. While the tasks the component should perform



1. < p, < CreateImage, < 300, int >, < 300, int > >, Polygon >
2. < IImage, < return >, IImage >
3. < p, < Interpolator, < filename, string >, < type, int > >, Polygon >
4. < i, < return >, Interpolator >
5. < p, < InterpolateLine, < y, int > >, Polygon >
6. < i, < ScanLine, null >, Interpolator >
7. < s, < return >, Scanline >
8. < i, < return >, Interpolator >
9. < p, < WriteLine, < y, int > >, Polygon >
10. < s, < Load, < x, int >, < y, int >, < r, int >, < g, int >, < b, int > >, ScanLine >
11. < IImage, < return >, IImage >
12. < s, < return >, ScanLine >
13. < p, < IncrementEdges, null >, Polygon >
14. < i, < return >, Interpolator >
15. < p, < CreateFile, < filename, string, [Type = bmp, gif, jpg, png, tif] > >, Polygon >
16. < IImage, < return >, IImage >

Figure 5.4: Polygon COMDAG

are quite simple, the component functionality may not conform to our definition. A third party image component may not allow us to create the desired file types, or the interface for loading information may be non-compliant. Other complications arise if the third party component supports extra operations that our program does not use. Although extra functions can be useful, when unused, the excess overhead increases code size without increasing effectiveness.

	Attr (P)	**Attr (IIC)**
Position	int x, int y	int x, int y
Color	int r, int g, int b	int r, int g, int b, *int a*
File Type	bmp, gif, jpg, png, tif	bmp, gif, jpg, png, tif, *pbm, pgm, ppm, tga*

Table 5.1: Attributes Required vs. Provided

We will apply Set Analysis to qualify the Imaginary Image Component (IIC), a COTS component. IIC runs on the Java platform, and the interface is provided as a class. The component allows the specification of a generic file, which can be saved as nine different types. IIC allows the image file to be changed one pixel at a time through the specification of a Position and a Color. Although, neither the position nor color class are used in the Polygon specification, we can use CCTs to resolve each

Meth (P)	Meth (IIC)
CreateImage(int width, int height)	SpecifyImage(int w, int h)
Load(int x, int y, int r, int g, int b)	LoadPixel(Position p, Color c)
CreateFile(String filename)	CreateFile(String name)
	Rotate(int degree)
	Flip()
	Scale(int percentage)
	StretchWidth(int percentage)
	StretchHeight(int percentage)
	Crop(Position tl, Position br)
	Invert()

Table 5.2: Methods Required vs. Provided

structure down to their base types, and perform the comparisons there. Table 5.1 shows the lowest level attributes. After the image has been created, IIC provides a variety of methods to alter the image appearance. Table 5.2 shows a complete list of methods in the IIC component.

Everything required by Polygon that is provided by IIC is listed in normal font. By examining Table 5.1 & 5.2, it is also apparent that IIC is *sufficient* to cover Polygon's requirements. Because IIC is sufficient, coverage is 100% for both attributes and methods.

$$\#attrInt(P, IIC) = |Attr(P) \cap Attr(IIC)| = 10$$

$$attrCov\% = \frac{\#attrInt}{\#Attr(P)} * 100 = \frac{10}{10} * 100 = 100\%$$

$$\#methInt(P, IIC) = |Meth(P) \cap Meth(IIC)| = 3$$

$$methCov\% = \frac{\#methInt}{\#Meth(P)}) * 100 = \frac{3}{3} * 100 = 100\%$$

Although IIC is sufficient, only 2 methods have a *complete match*. LoadPixel is only a *partial match* because of the extra alpha attribute in the color parameter. All other portions of IIC *safely exceed* Polygon's requirements. The overhead is the portion of IIC that safely exceeds Polygon's requirements. In Table 5.1 & 5.2, the

overhead is denoted with italics.

$\#attrOver(IIC, P) = |Attr(IIC) - Attr(P)| = 5$ $attrOver\% = \frac{\#Attr(IIC) - \#Attr(P)}{\#Attr(IIC)} * 100 = \frac{5}{15} * 100 = 33\%$

$attrFit\% = (1 - \frac{attrOver\%}{100}) * 100 = (1 - 0.33) = 77\%$

$\#methOver(IIC, P) = |Meth(IIC) - Meth(P)| = 7$ $methOver\% = \frac{\#Meth(IIC) - \#Meth(P)}{\#Meth(IIC)} * 100 = \frac{7}{10} * 100 = 70\%$

$methFit\% = (1 - \frac{methOver\%}{100}) * 100 = (1 - 0.70) = 30\%$

The component qualification is determined by averaging the coverage and overhead together. The coverage and overhead are weighted equally in this example. In this case the overall qualification is 76.75%, making IIC fairly qualified as a component for the polygon program.

$Overall\ Qualification = \frac{AttrCov + AttrFit + MethCov + MethFit}{4} = 76.75\%$

5.2 Select Ocean Cruises

For the second application we are using an example of a cruise–liner provided in [5]. The cruise–liner is referred to as Select Cruises. Figures 5.6, 5.7, and 5.8 are modified from the figures found in [5]. The cruise line provides adventures for guests on tall ships. The ships take long and short trips from New Zealand and the Caribbean. Each ship has twenty–four berths available for guests and twelve berths available for crew members. Types of berths include bunks, hammocks, cabins, and dormitories. Sales inquiries come from various sources: direct calls, postal, and on–line requests. Access to the cruise schedule, berth availability, and database is required in order to check cruise availability. All inquiries are logged to help aid in future cruise planning.

When reservations are made, confirmation letters detailing the information about the reservation are sent to the customers.

We design the system such that we have one main component. The component is the Design Cruise Component (DCC) (Figure 5.5). The Cruise component contains information about cruises, sales, and customers.

We apply our qualification process to the Select Cruises application. The first step of our process is to identify candidate components (Table 3.1). To identify candidate components we follow the first two phases of the OTSO method (section 3.2). We first search for candidate components. In keeping with the cruise–line example, say we happened to find two candidate components on–line. The two components are the Candidate Sales Component (CSC) in Figure 5.6 and the Candidate Cruise Component (CCC) in Figure 5.7. According to the documentation of the components, the Sales component handles all sales transactions and the Cruise component handles all information about cruises and their availability. Next, we screen the candidate components by looking at available UML documents. If the candidate components do not have static information (Class Diagrams) about the component interface available, the component should be discarded. Otherwise we move to the second step of our qualification process and build the Integrated UML Model.

For this example we will build the CCTS for both the design and candidate components. The CCTS are built for the interfaces of the components. The CCTS are modified for this example to separate the attributes and methods of the component. The attributes for the design component are found in Table 5.3. The methods for the design component are found in Table 5.4.

The attributes for the candidate Sales component are found in Table 5.5. The methods for the candidate Sales component are found in Table 5.6.

The attributes for the candidate Cruise component are found in Table 5.7. The

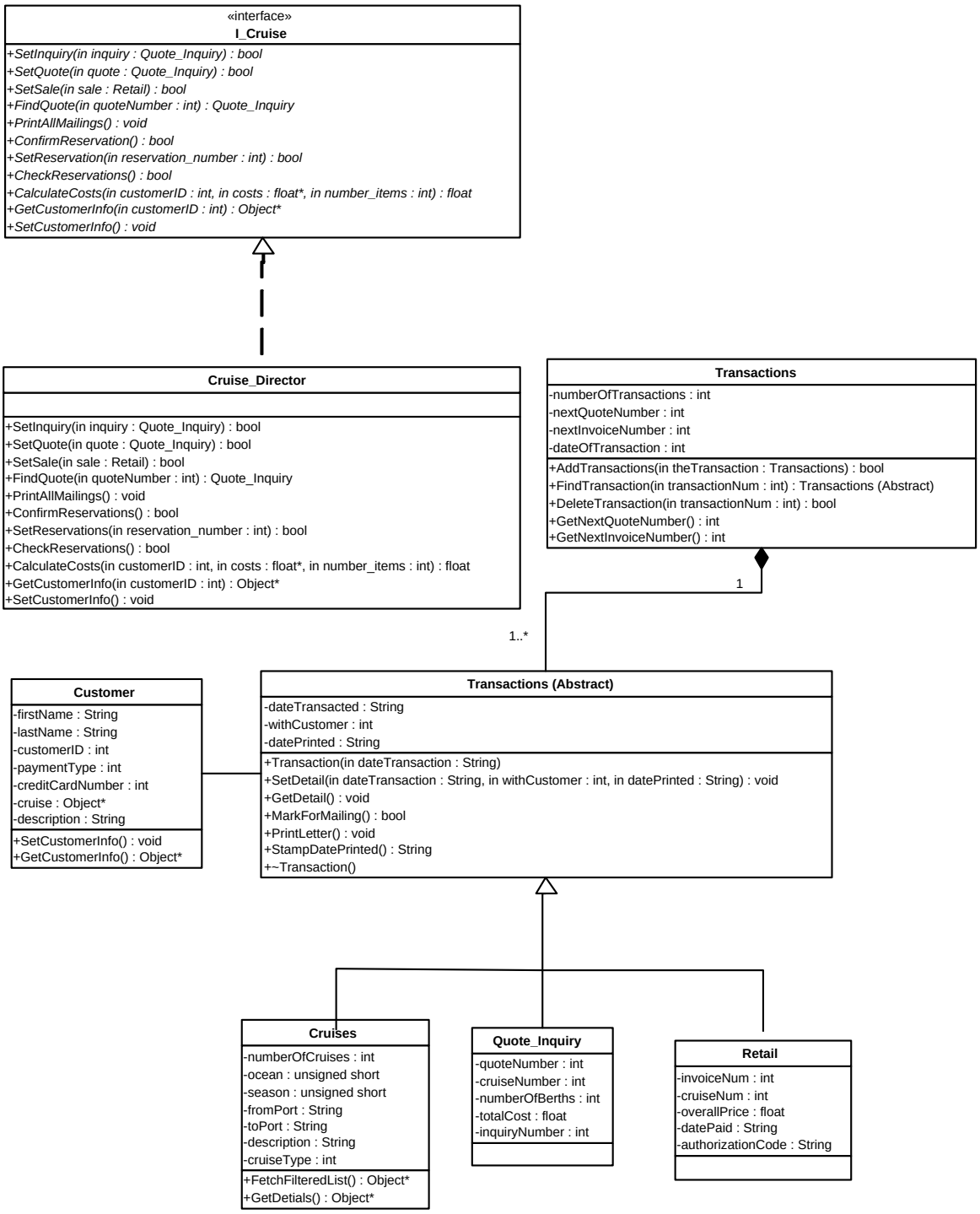


Figure 5.5: Design of Cruise Component for Cruise–Line Application

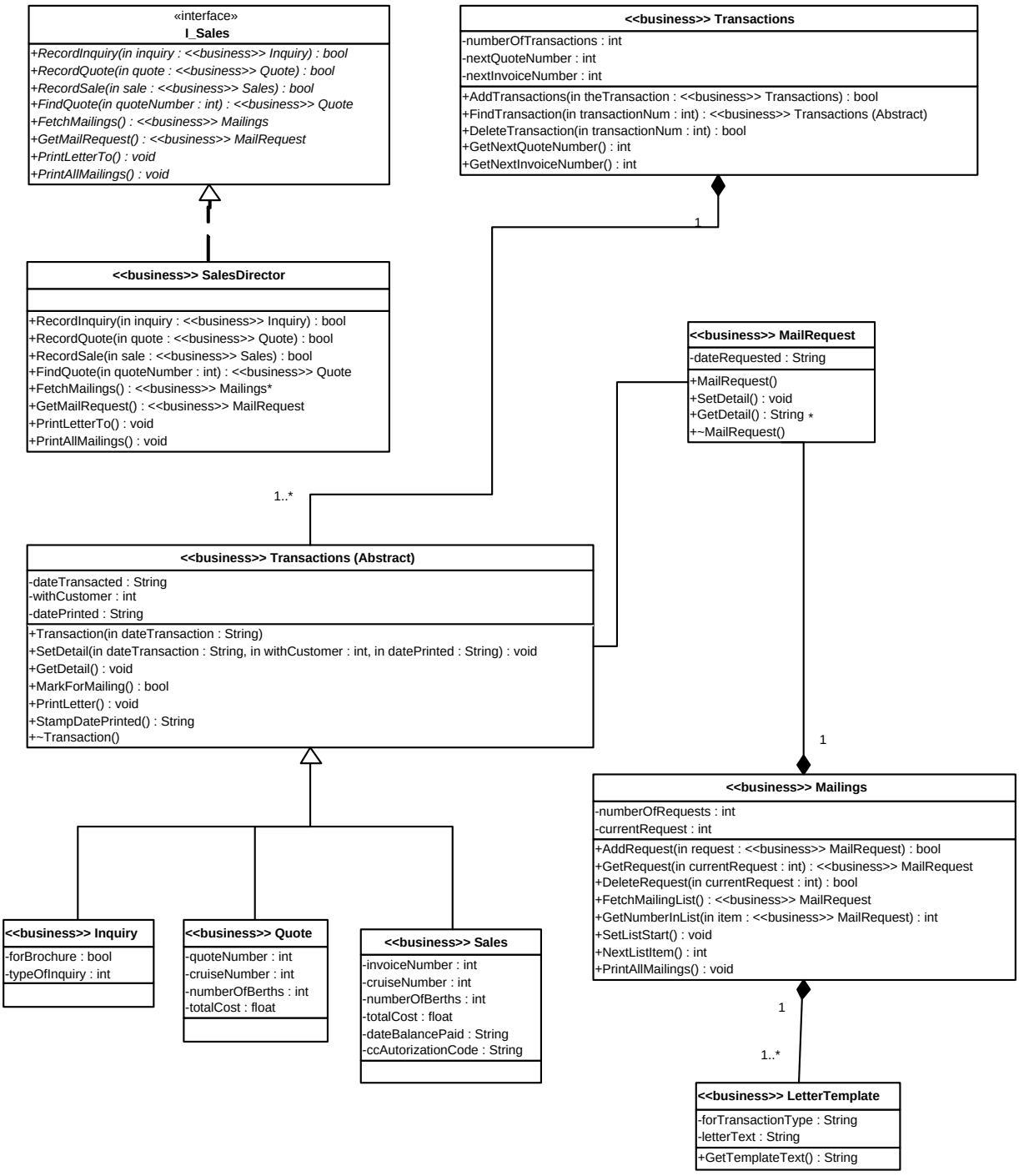


Figure 5.6: Candidate Sales Component

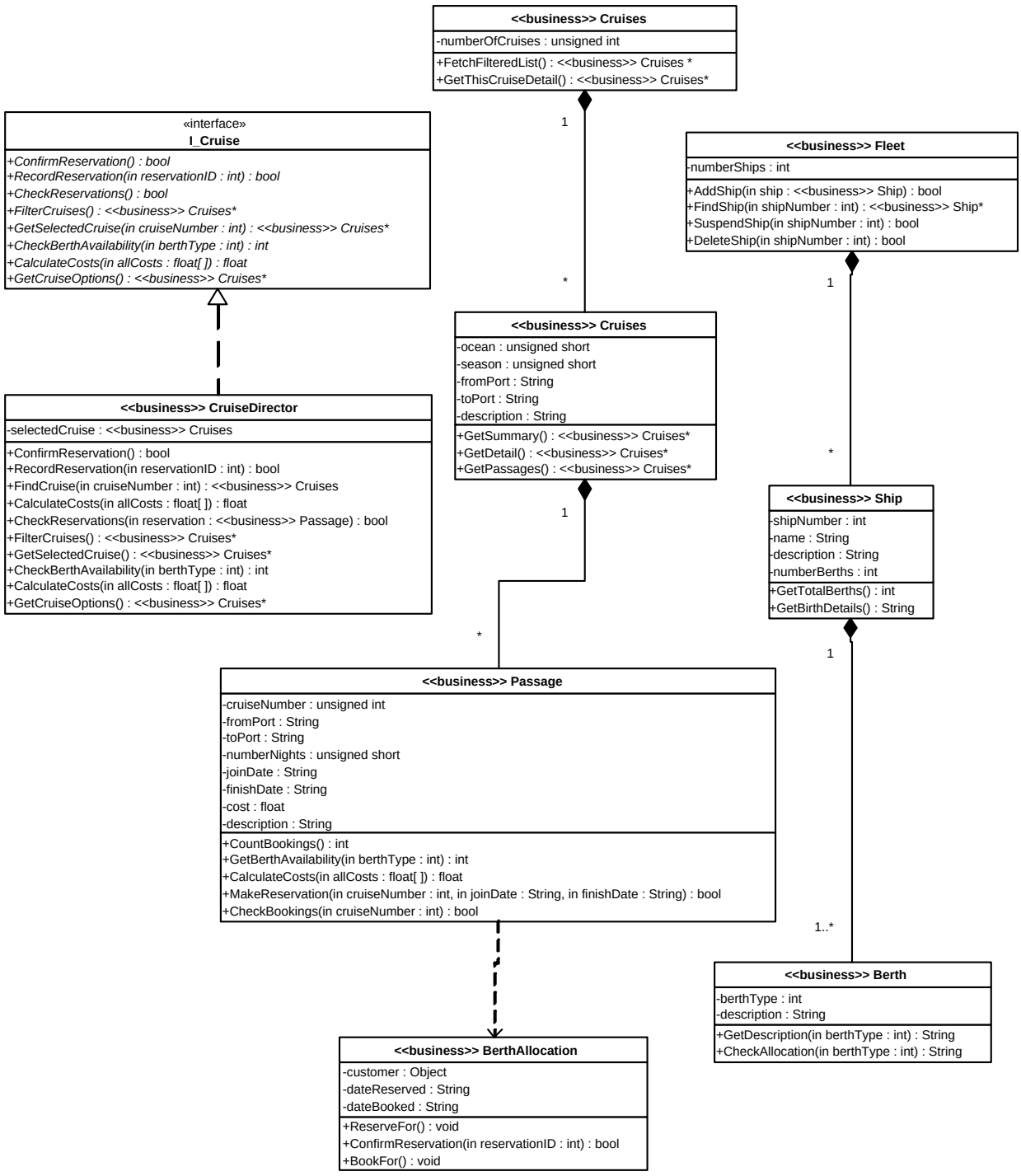


Figure 5.7: Candidate Cruise Component

Number	Attribute	Type
1.	firstName	String
2.	lastName	String
3.	paymentType	int
4.	creditCardNumber	int
5.	cruise	Object *
6.	description	String
7.	numberOfTransactions	int
8.	nextQuoteNumber	int
9.	nextInvoiceNumber	int
10.	dateOfTransaction	int
11.	numberOfCruises	int
12.	ocean	unsigned short
13.	season	unsigned short
14.	fromPort	String
15.	toPort	String
16.	quoteNumber	int
17.	cruiseNumber	int
18.	numberOfBerths	int
19.	totalCost	float
20.	inquiryNumber	int
21.	invoiceNum	int
22.	cruiseNum	int
23.	overallPrice	float
24.	datePaid	String
25.	authorizationCode	String
26.	customerID	int

Table 5.3: Design Cruise Component Attributes.

methods for the candidate Cruise component are found in Table 5.8.

The third step of our qualification process is to build the qualification model. The qualification model requires that we derive the coverage and fitness metrics for the candidate components. The following qualification measures are derived from component DCC and CSC.

$\#attrInt(DCC, CSC) = |Attr(DCC) \cap Attr(CSC)| = 11$

Number	Method	Parameters Types	Return Type
1.	SetInquiry	QuoteInquiry	bool
2.	SetQuote	QuoteInquiry	bool
3.	SetSale	Retail	bool
4.	FindQuote	int	QuoteInquiry
5.	PrintAllMailings	void	void
6.	ConfirmReservation	void	bool
7.	SetReservation	int	bool
8.	CheckReservations	void	bool
9.	CalculateCosts	int, float *, int	float
10.	GetCustomerInfo	int	Object *
11.	SetCustomerInfo	void	void

Table 5.4: Design Cruise Component Methods.

Number	Attribute	Type
1.	numberOfTransactions	int
2.	nextQuoteNumber	int
3.	nextInvoiceNumber	int
4.	dateRequested	String
5.	forBrochure	bool
6.	typeOfInquiry	int
7.	quoteNumber	int
8.	cruiseNumber	int
9.	numberOfBerths	int
10.	totalCost	float
11.	invoiceNumber	int
12.	cruiseNumber	int
13.	dateBalancePaid	String
14.	ccAuthorizationCode	String

Table 5.5: Candidate Sales Component Attributes.

$$attrCov\% = \frac{\#attrInt}{\#Attr(DCC)} * 100 = \frac{11}{26} * 100 = 42\%$$

$$\#methInt(DCC, CSC) = |Meth(DCC) \cap Meth(CSC)| = 5$$

Number	Method	Parameters Types	Return Type
1.	RecordInquiry	Inquiry	bool
2.	RecordQuote	Quote	bool
3.	Sales	Retail	bool
4.	FindQuote	int	Quote
5.	FetchMailings	void	Mailings
6.	GetMailRequest	void	MailRequest
7.	PrintLetterTo	void	void
8.	PrintAllMailings	void	void

Table 5.6: Candidate Sales Component Methods.

Number	Attribute	Type
1.	numberOfCruises	int
2.	numberShips	int
3.	ocean	unsigned short
4.	season	unsigned short
5.	fromPort	String
6.	toPort	String
7.	description	String
8.	shipNumber	int
9.	name	String
10.	numberBerths	int
11.	berthType	int
12.	customer	Object
13.	dateReserved	String
14.	dateBooked	String
15.	cruiseNumber	int
16.	numberNights	int
17.	joinDate	String
18.	finishDate	String
19.	cost	float

Table 5.7: Candidate Cruise Component Attributes.

$methCov\% = \frac{\#methInt}{\#Meth(DCC)}) * 100 = \frac{5}{11} * 100 = 46\%$

Number	Method	Parameters Types	Return Type
1.	ConfirmReservation	void	bool
2.	RecordReservation	int	bool
3.	CheckReservations	void	bool
4.	FilterCruises	void	Cruises *
5.	GetSelectedCruise	int	Cruises *
6.	CheckBerthAvailability	int	int
7.	CalculateCosts	float []	float
8.	GetCruiseOptions	void	Cruises *

Table 5.8: Candidate Cruise Component Methods.

$$\#attrOver(CSC, DCC) = |Attr(CSC) - Attr(DCC)| = 3$$

$$attrOver\% = \frac{\#attrOver(CSC,DCC)}{\#Attr(CSC)} * 100 = \frac{3}{14} * 100 = 21\%$$

$$attrFit\% = (1 - \frac{attrOver\%}{100}) * 100 = (1 - 0.21) * 100 = 79\%$$

$$\#methOver(CSC, DCC) = |Meth(CSC) - Meth(DCC)| = 3$$

$$methOver\% = \frac{\#methOver(CSC,DCC)}{\#Meth(CSC)} * 100 = \frac{3}{8} * 100 = 38\%$$

$$methFit\% = (1 - \frac{methOver\%}{100}) * 100 = (1 - 0.38) = 62\%$$

Next we have to calculate the qualification measures from component DCC and CCC.

$$\#attrInt(DCC, CCC) = |Attr(DCC) \cap Attr(CCC)| = 10$$

$$attrCov\% = \frac{\#attrInt}{\#Attr(DCC)} * 100 = \frac{10}{26} * 100 = 38\%$$

$$\#methInt(DCC, CCC) = |Meth(DCC) \cap Meth(CCC)| = 4$$

$methCov\% = \frac{\#methInt}{\#Meth(DCC)}) * 100 = \frac{4}{11} * 100 = 36\%$

$\#attrOver(CCC, DCC) = |Attr(CCC) - Attr(DCC)| = 9$

$attrOver\% = \frac{\#attrOver(CCC,DCC)}{\#Attr(CCC)} * 100 = \frac{9}{19} * 100 = 47\%$

$attrFit\% = (1 - \frac{attrOver\%}{100}) * 100 = (1 - 0.47) * 100 = 53\%$

$\#methOver(CCC, DCC) = |Meth(CCC) - Meth(DCC)| = 4$

$methOver\% = \frac{\#methOver(CCC,DCC)}{\#Meth(CCC)} * 100 = \frac{4}{8} * 100 = 50\%$

$methFit\% = (1 - \frac{methOver\%}{100}) * 100 = (1 - 0.50) = 50\%$

Next, we calculate the attribute qualification and method qualification measures. These measures are weighted according to weights derived during the AHP (section 3.4.4). Our assumptions during the AHP were that coverage metrics are more important than fitness metrics and that attributes are more important than methods. We apply Equations 3.31 and 3.32 to the results above to calculate the attribute qualification and method qualification, respectively, of components CSC and CCC.

$attribute\ qualification(CSC) = 0.11 * attrFit\% + 0.50 * attrCov\% = 0.11 * 79\% + 0.50 * 42\% = 30\%$

$method\ qualification(CSC) = 0.08 * methFit\% + 0.32 * methCov\% = 0.08 * 62\% + 0.32 * 46\% = 20\%$

$attribute\ qualification(CCC) = 0.11 * attrFit\% + 0.50 * attrCov\% = 0.11 * 53\% + 0.50 * 38\% = 25\%$

$method\ qualification(CCC) = 0.08 * methFit\% + 0.32 * methCov\% = 0.08 * 50\% + 0.32 * 36\% = 16\%$

After the attribute and method qualification measures are calculated the overall qualification is evaluated (Equation 3.33):

$overall\ qualification(CSC) = attribute\ qualification + method\ qualification = 30\% + 20\% = 50\%$

$overall\ qualification(CCC) = attribute\ qualification + method\ qualification = 25\% + 16\% = 41\%$

At first glance, 50% and 41% qualification measures for components CSC and CCC, respectively, may not be considered satisfactory. The designer has some decisions to make. If the attributes and methods of the two candidate components are compared, one can see that most of them do not overlap. Thus, in reality the two candidate components together satisfy most of the design components requirements. Together CSC and CCC match most the attributes and methods found in DCC. Such an observation indirectly suggests that maybe the design component should be split into two components. However, this is a research problem that is out of the scope of this thesis.

If the candidate components overlap a lot in attributes and methods, the fourth step of the qualification process should be executed. Collecting coupling measures may give a clearer picture of whether or not to discard the candidate components and search for different ones.

The fifth and last step of the process is to rank the components. From the overall qualification measures we know that CSC ranks first and CCC ranks second. Given the special circumstances that both of the components cover different parts of the design component (DCC), we can select both of the candidate components to fill "gaps" in the final software system.

For the cruise line example we use the following Use–Case as one of the primary Use–Cases [5]:

> **Use Case:** Find a Cruise **Intent:** To search for cruises that a customer has specified. Calculate costs based on berth types and cruise type **Pre Conditions:** The details for a cruise are available **Post Conditions:** The confirmation of a selected cruise **Description:**
>
> 1. System displays cruise options
> 2. User selects a cruise
> 3. System presents passages of selected cruise
> 4. User selects passage
> 5. System presents berth information
> 6. User selects available berth
> 7. System shows cost of cruise
> 8. User confirms selection of cruise

Figure 5.8 shows the sequence diagram needed for the COMDAG.

The sequence diagram in Figure 5.8 indicates that there are four nodes connected to the Candidate Cruise Component: 3, 5, 7, 9. Hence, $\#conn = 4$. We can simulate the Find a Cruise use case and notice that there are 4 different activations of the I_Cruise interface. Hence, $\#act = 4$. We need to normalize these coupling measures so that we can compare them with the measures collected during step 3 of the qualification process. We normalize the coupling measures by Equation 3.35 in Section 3.5:

$$coupling\% = (0.5 * \frac{\#conn}{total\ \#conn} + 0.5 * \frac{\#act}{total\ \#act}) * 100 = (0.5 * \frac{4}{9} + 0.5 * \frac{4}{16}) * 100 = 35\%$$

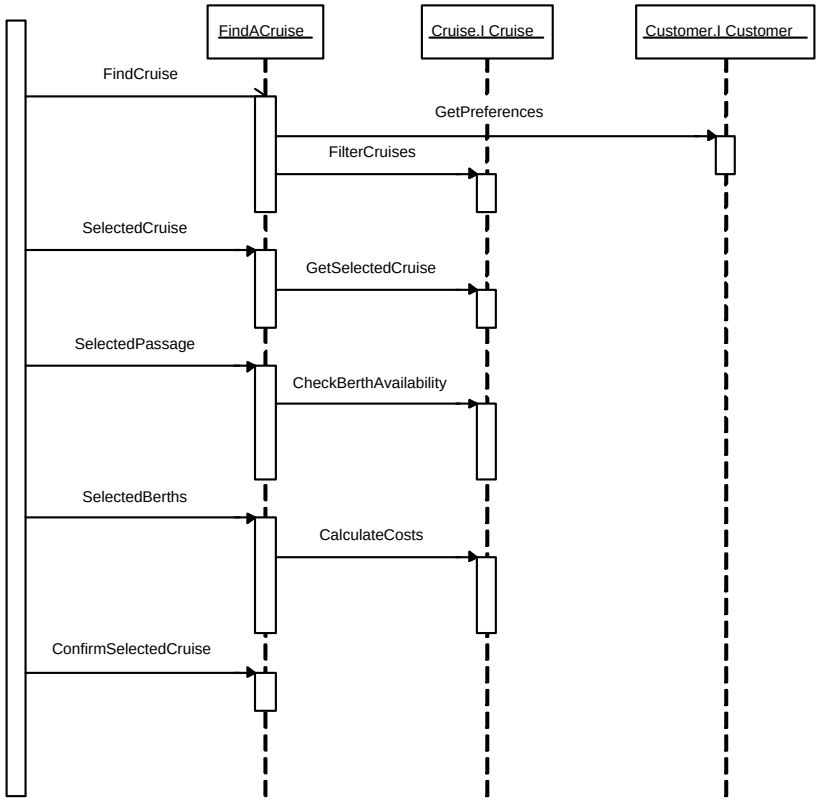
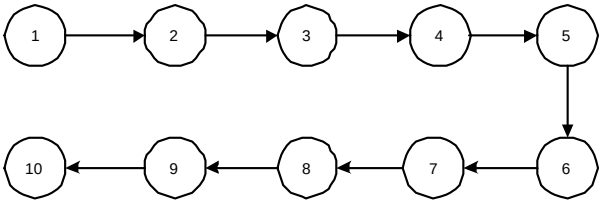


Figure 5.8: Find a Cruise Sequence Diagram

A coupling measure of 35% is slightly high. If the component is to be accepted or rejected based on the coupling measure, it would be rejected. The coupling threshold c_t for this example is 30%. Any component with a measure above this threshold should not be considered unless the designer chooses to calculate a hybrid measure for the qualification and coupling measures. Notice that *total #conn* is 9 because the COMDAG in Figure 5.9 has 9 edges. Also, notice that *total #act* is 16 because tracing the Find a Cruise use case through the COMDAG indicates that we have the possibility to choose among four cruises (cruiseNumber) and four types of berths (berthTypes). We assume that the four cruises are the Caribbean, Pacific, Atlantic, and South American. The four types of berths include a bunk, hammock, cabin, and dormitory. Thus, we have 2^4 possible combinations between cruises and berth types.



1. $< findACruise, GetPreferences, new, \{null\}, [null], FindACruise >$
2. $< customer, return, new, \{null\}, [null], Customer >$
3. $< findACruise, FilterCruises, new, \{null\}, [null], FindACruise >$
4. $< cruise, return, exits, \{null\}, [null], Cruise >$
5. $< findACruise, GetSelectedCruise, new, \{< int, cruiseNumber, null >$
$\}, [null], FindACruise >$ 6. $< cruise, return, exits, \{null\}, [null], Cruise >$
7. $< findACruise, CheckBerthAvailability, new, \{< int, berthType, null >$
$\}, [null], FindACruise >$ 8. $< cruise, return, exists, \{null\}, [null], Cruise >$
9. $< findACruise, CalculateCosts, new, \{< float[], allCosts, null >$
$\}, [null], FindACruise >$ 10. $< cruise, return, exists, \{null\}, [null], Cruise >$

Figure 5.9: Candidate Cruise COMDAG

Note that we are only interested in the parameters that affect the activation of methods in the sequence diagrams. In this case we only have four connections and thus four possible methods that may be activated. These methods are FilterCruises (), GetSelectedCruises (), CheckBerthAvailability (), and CalculateCosts (). FilterCruises () does not have any parameters. GetSelectedCruises () accepts the cruiseNumber and CheckBerthAvailability () accepts the berthType. Lastly, CalculateCosts () accepts the costs of each item. The CalculateCosts () method does not affect the outcome of the activations of the nodes in the COMDAG because it is always activated regardless of the other methods.

After we have calculated the coupling measures we need to rank the components (step 5). Notice that option 3 in Table 3.9 suggests that we can weight both the qualification measure and the coupling measure to give us a better understanding of the candidate components. For this example we only calculated coupling measures

for the Candidate Cruise Components because its qualification measure is less than 50%. We can calculate the hybrid measure as:

$hybrid\ measure = w_q * overall\ qualification + w_c * (1 - \frac{coupling\%}{100}) * 100 = 0.7 * 41\% + 0.3 * 65\% = 48\%$

The overall qualification measure is weighted by w_q and the coupling% is weighted by w_c. For this application the overall qualification measure is weighted 70% because it has the most impact on our decision about the component. The weights for this measure are subjective according to the designers preference.

The results indicate that the Candidate Cruise Component (CCC) should not be considered for the system, but the Candidate Sales Component (CSC) should be. For acceptance the hybrid value is required to satisfy a 50% threshold denoted (h_t).

Chapter 6

Conclusions and Future Work

The purpose of this thesis is to present a qualification process that allows a design engineer to effectively and efficiently evaluate the fitness of a COTS candidate component for a software system. The qualification approach is presented in five steps (Table 3.1) to systematically assist a designer in finding candidate components that fill "gaps" in the designed software system. The first step is to identify all candidate components. In order to identify candidate components, the first two phases of the Off–The–Shelf–Option (OTSO) method are followed [31]. We first search for candidate components and then we screen components according to the requirements and architecture of the software system. The second step of our method is to build a UML integrated model for each candidate component that has passed the screening phase. The integrated model consists of first constructing Constrained Class Tuples (CCTs) from static information provided by the class diagrams and OCL constraints. Next, the Object Method Directed Acyclic Graph (OMDAG) is created from the behavioral information available in the sequence diagrams. Third, the CCTs and OMDAG are combined into a Constrained OMDAG (COMDAG), which is needed for collecting coupling measures. The third step of our qualification method is to define the quali-

fication model for collecting qualification measures. The qualification model uses set analysis to derive overhead, coverage, and fitness measures. The fourth step is to calculate coupling measures. The coupling measures collected include the number of connections (#conn) and the number of activations (#act). These measures are derived from the trace of a use case, for an operational profile, through the COMDAG of the component. The fifth and last step of the process is to rank the candidate components. The components may be ranked according to the qualification measures, coupling measures, or a hybrid of the measures (Table 3.9). If the hybrid method for ranking components is chosen, then the designer must assign subjective weights to the qualification measures and the coupling measures (see section 5.2 for an example).

This thesis addresses the three research problems proposed in the introduction. The qualification and coupling measures that we collect require both class diagrams (static) and sequence (dynamic) diagrams. We know that based on the static information derived from the class diagrams, we can derive metrics from the CCTS (section 3.4). Also, we know that we can derive coupling measures from combined class and sequence diagram views (COMDAG). Hence, we can qualify components in the design phase (rp–1). Our qualification measures compare the signature information of the design component with the signature information of the candidate component. All signature information is based on information available during the design phase. Our application of the measures to real examples have shown our qualification method to be both effective (rp–2) and efficient (rp–3). The method requires little information, just the signature, and we are able to determine whether or not a particular candidate component has the capability to satisfy the requirements of a design component. In the two examples presented in the chapter 5, the method showed to be systematic and effective. Our qualification method is also very efficient compared to other methods. First of all our method is applied during the design phase, one phase before most

other methods. Secondly, our method, except for the match determination part, is relatively easy to automate [57] and requires little information about the design and candidate components involved.

Future work should focus on implementing a tool to execute the qualification process. The tool could be used for not only qualifying candidate components, but for possibly collecting effort measures as follows: the tool could execute on two versions of a component, one earlier version and one later version. If the qualification measure is 100% no effort has been made. If the measure is less than 100% a certain percentage of effort has been made. The tool could be used to indicate the amount of time and effort it would require to add a certain amount of functionality to a certain software system. In the future experiments to test some of the qualifying thresholds should be considered. As of now the thresholds are subjectively set by the design engineer. Experiments could indicate thresholds that corroborate "good" and fit candidate components.

Bibliography

[1] A. Abdurazik and J. Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation", *3rd International Conference on the UML*, pp. 383–395, Oct 2000.

[2] ACM, "The Full Computing Reviews Classification System", ACM, New York, 1992.

[3] P. Allen, "Component-based development for enterprise systems: Applying the SELECT Perspective", Cambridge, UK, New York: Cambridge University Press, 1998.

[4] A. Andrews, R. France, S. Ghosh, and G. Craig, "Test Adequacy Criteria for UML Design Models," *Journal of Software Testing, Verification, and Reliability*, pp. 95–127, April–June, 2003.

[5] H. Apperly, R. Hofman, S. Latchem, B. Maybank, B. McGibbon, D. Piper, and C. Simons, *Service and Component–based Development: Using the Select Perspective and UML*, Addison–Wesley, 2003.

[6] B. Baudry, V. L. Hanh, and Y. L. Traon, "Testing–for–Trust: The Genetic Selection Model Applied to Component Qualification", *Proc. of 33rd International Conference on Technology of Object-Oriented Languages*, pp. 108–119, Jun 5–8, 2000.

[7] J. R. Beach and M. L. Bonewell, "Setting–up a Successful Software Vendor Evaluation/Qualification Process for 'Off-the-Shelve' Commercial Software used in Medical Devices," *Proc. of Sixth Annual IEEE Symposium on Computer-Based Medical Systems*, Ann Arbor, MI, pp. 284-288, Jun 1993.

[8] V. Belton, "A Comparison of the Analytic Hierarchy Process and a Simple Multi-attribute Value Function", *European Journal of Operational Research*, vol. 26, no. 1, pp. 7–21, 1986.

[9] L. Beus–Dukic and A. Wellings, "Requirements for a COTS Software Component: A Case Study", *Conference on European Industrial Requirements Engi-*

neering London, Requirements Engineering, vol. 3, no. 2, Springer–Verlag, pp. 115–120, Oct 1998.

[10] J. Bieman and L. Ott, "Measuring Functional Cohesion", IEEE Trans. Software Eng., vol. 20, no. 8, pp. 644–657, Aug 1994.

[11] R. Binder, *Testing Object-Oriented Systems Models, Patterns, and Tools*, Object Technology Series, Addison Wesley, Reading, Massachusetts, 1999.

[12] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, R. Selby, "Cost models for future software life cycle processes: COCOMO 2.0", *Annals of Software Engineering*, vol.1, Baltzer, Amsterdam, pp. 57–94, 1995.

[13] L. Briand and Y. Labiche, "A UML–based Approach to System Testing", *4th International Conference on the UML*, pp. 194–208, Oct 2001.

[14] L. Briand, "COTS Evaluation and Selection", *Proc. International Conference on Software Maintenance*, Bethesda, pp. 222–223, Nov 1998.

[15] T. Y. Chen, P. L. Poon, and T. H. Tse, "A Choice Relation Framework for Supporting Category–Partition Test Case Generation", *IEEE Transaction on Software Engineering*, vol. 29, no. 7, pp. 577–593, July 2003.

[16] A. Egyed and C. Gacek, "Automatically Detecting Mismatches during Component–Based and Model–Based Development", *IEEE 14th International Conference on Automated Software Engineering*, pp 191–198, Oct 1999.

[17] T. Ellis, "COTS Integration in Software Solutions: A Cost Model", *Systems Engineering in the Global Marketplace, Proceedings of the 5th International Symposium on NCOSE*, vol. 1, pp. 171–177, 1995.

[18] S. S. Epp, *Discrete Mathematics with Applications*, PWS Publishing Company, Boston, MA, 1995.

[19] N. Fenton, *Software Metrics–A Rigorous Approach*, Chapman and Hall, London, 1991.

[20] M. Fowler and K. Scott, *UML Distilled Second Edition*, Addison–Wesley, 2000.

[21] S. Cardenas–Garcia and M. V. Zelkowitz, "A Management Tool for Evaluation of Software Designs", *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 961–971, Sep 1991.

[22] S. Ghosh, R. France, C. Braganza, N. Kawane, A. Andrews, and O. Pilskalns, "Test Adequacy Assessment for UML Design Model Testing", *IEEE 14th International Symposium on Software Reliability Engineering*, pp. 332–343, 2003.

[23] D. Hamlet, D. Mason, and D. Woit, "Theory of Software Reliability Based on Components", *Proc. of the 23rd International Conference on Software Engineering*, pp. 361–370, 2001.

[24] G. Heineman and W. Council, *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, May, 2001.

[25] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj, "Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications", *IEEE Transactions on Software Engineering*, vol. 24, no. 11, pp. 927–948, 1998.

[26] S. Henninger, "Supporting the Construction and Evolution of Component Repositories", *Proc. of the 18th International Conference on Software Engineering*, pp. 279–288, Mar 1996.

[27] C. Jard and S. Pickin, "COTE (COmponent TEsting)", *ERCIM News*, no.48, Jan 2002.

[28] A. Jeanrenaud and P. Romanazzi, "Software Product Evaluation Metrics: Methodological Approach," *Software Quality Management II. Building Quality into Software 2*, pp. 59–69, 1995.

[29] H. Jung and B. Choi, "Optimization Models for Quality and Cost of Modular Software Systems", *European Journal of Operational Research*, pp. 98–117, 1998.

[30] J. Kontio, "A Case Study in Applying a Systematic Method for COTS Selection", *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, March 25-30, 1996, pp. 201-209, Los Alamitos, CA: IEEE Computer Society Press, 1996.

[31] J. Kontio, "OTSO: A Systematic Process for Reusable Software Component Selection", CS-TR-3478, 1995, University of Maryland Technical Reports, University of Maryland. College Park, MD.

[32] J. Kontio, S. F. Chen, K. Limperos, R. Tesoriero, G. Caldiera, and M. Deutsch, "A COTS Selection Method and Experiences of Its Use", 20th Software Engineering Workshop, NASA Software Engineering Laboratory, Greenbelt, MD, 1995.

[33] C. Larman, *Applying UML and Patterns*, Prentice Hall, 2002.

[34] O. Laitenberger, C. Atkinson, M. Schlich, and K. El Emam, "An Experimental Comparison of Reading Techniques for Defect Detection in UML Design Documents," *Journal of Systems and Software*, vol. 53, no. 2, pp. 183–204, 2000.

[35] N. G. Lester, F.G. Wilkie, and D.W. Bustard, "Using UML to Categorise and Specify Criteria for Reusable Artefacts", *UML '98 Beyond the Notation - International Workshop (Preliminary Proceedings)*, 1998.

[36] R. W. Lichota, R. L. Vesprini, and B Swanson, "PRISM Product Examination Process for Component Based Development", *Proceedings of the Fifth International Symposium on Assessment of Software Tools*, pp. 61–69, 1997.

[37] N.A. Maiden and C. Ncube, "Acquiring COTS software selection requirements", *IEEE Software*, vol. 15, no. 2, pp. 46–56, Mar 1998.

[38] W. McUmber and B. Cheng,"A General Framework for Formalizing UML with Formal Languages" *Proc. of the 23rd International Conference on Software Engineering*, pp. 433–442, May 2001.

[39] B. Morel and P. Alexander, "Automating Component Adaptation for Reuse", *Proc. 18th IEEE International Conference on Automated Software Engineering*, pp. 142–151, Oct 2003.

[40] M. Morisio and A. Tsoukias, "IusWare: A Methodology for the Evaluation and Selection of Software Products", *IEEE Proceedings of Software Engineering*, vol. 144, no. 3, pp. 162–174, Jun 1997.

[41] L. M. Northrop, "SEI's Software Product Line Tenets", *IEEE Software*, vol. 19, no. 4, pp. 32–40, July–Aug 2003.

[42] Object Management Group, "UML 2.0 Draft Specification", http://www.omg.org/uml, 2003.

[43] A. O'Fallon, O. Pilskalns, A. Knight, and A. Andrews, "Defining and Qualifying Components in the Design Phase", *Proc. of the Sixteenth International Conference on Software Engineering and Knowledge Engineering*, pp. 129–134, Jun 2004.

[44] J. Offutt and A. Abdurazik, "Generating Test from UML Specifications", *2nd International Conference on the UML*, pp. 416–429, Oct 1999.

[45] T. Ostrand and M. Balcer, "The Category–Partition Method for Specifying and Generating Functional Tests", *Communications of the ACM*, pp. 676–686, June 1988.

[46] S. Pickin, C. Jard, T. Heuillar, J. M. Jezequel, and P. Desfray, "A UML-integrated Test Description Language for Component Testing", *Proc. of Practical UML-Based Rigorous Development Methods Workshop*, pp. 208–223, 2001.

[47] O. Pilskalns, A. Andrews, R. France, and S. Ghosh, "Rigorous Testing by Merging Structural and Behavioral UML Representations", *Sixth International Conference on the Unified Modeling Language*, pp. 234–248, 2003.

[48] O. Pilskalns, A. Andrews, A. Knight, and S. Ghosh, "A Rigorous Systematic Approach to UML Design Evaluation", Submitted to Software and Systems Modeling Journal, Springer Verlag.

[49] R. Pressman, *Software Engineering: A Practitioner's Approach, Fourth Edition*, McGraw–Hill Companies, Inc., New York, NY, 1997.

[50] J. Robertson, S. Robertson, "Volere Requirements Specification Template, Edition 4," Atlantic Systems Guild, London, 1997, (http://www.systemsguild.com/GuildSite/Robs/Template.html).

[51] T. Saaty, *The Analytic Hierarchy Process*, New York, NY, McGraw–Hill, 1980.

[52] T. Schafer, A. Knapp, and S. Merz, "Model Checking UML State Machines and Collaborations", *Electronic Notes in Theoretical Computer Science*, vol. 47, pp. 1–13, 2001.

[53] M. Shaw, "Truth vs. Knowledge: The Difference Between What a Component Does and What We Know It Does", *Proc. 8th International Workshop on Software Specification and Design*, pp. 181–185, March, 1996.

[54] E. Weyuker, "The Evaluation of Program-Based Software Test Data Adequacy Criteria", *Communications of the ACM*, vol. 31, no. 6, pp. 668–675, Jun 1988.

[55] C. Wohlin and A. Andrews, "Prioritizing and Assessing Software Project Success Factors and Project Characteristics using Subjective Data", *Empirical Software Engineering*, vol. 8, pp. 285–308, 2003.

[56] Y. Wu, M.H. Chen, and J. Offutt, "UML–based Integration Testing for Component–based Software", *2nd International Conference on COTS–based Software Systems*, pp. 251–260, Feb 2003.

[57] Z. Xing and E. Stroulia, "Data–mining in Support of Detecting Class Co–evolution", *Proc. of the Sixteenth International Conference on Software Engineering and Knowledge Engineering*, pp. 123–128, Jun 2004.