4-2008

COMP-REF: A technique to guide the delegation of responsibilities to components in software systems

Subhajit DATTA
Singapore Management University, subhajitd@smu.edu.sg

Robert van Engelen
Florida State University

Citation
DATTA, Subhajit and van Engelen, Robert. COMP-REF: A technique to guide the delegation of responsibilities to components in software systems. (2008). *Fundamental approaches to software engineering: 11th International Conference, FASE 2008: Budapest, Hungary, March 29-April 6: Proceedings*. 4961, 332-346.
Available at: https://ink.library.smu.edu.sg/sis_research/6006

COMP-REF: A Technique to Guide the Delegation of Responsibilities to Components in Software Systems

Subhajit Datta and Robert van Engelen

Department of Computer Science and School of Computational Science
Florida State University, Tallahassee, FL 32306, USA
sd05@fsu.edu, rvaneng@fsu.edu

Abstract. In software systems, components collaborate to collectively fulfill requirements. A key concern of software design is the delegation of responsibilities to components such that user needs are most expediently met. This paper presents the COMP-REF technique based on a set of metrics and Linear Programming (LP) to guide the allocation of responsibilities of a system's components. We define the metrics *Aptitude Index*, *Requirement Set*, and *Concordance Index* to extract some design characteristics and use these metrics in an optimization algorithm. Results from experimental validation of the COMP-REF technique across a range of software systems are reported. We also discuss future directions of work in extending the scope of technique.

1 Introduction

Larman has called the ability to assign responsibilities as a "desert-island skill" [22], highlighting its criticality in the software development process. Indeed, deciding which component does what remains an important challenge for the software designer. Ideally, each component should perform a specialized task and cooperate with other components to deliver the system's overall functionality. But very often responsibilities are delegated to components in an ad-hoc manner, resulting in components that try to do almost everything by themselves or those that depend extensively on other components for carrying out their primary tasks. During initial design, it is not unusual to spawn a new component for every new bit of functionality that comes to light. As design matures, many of these components are best combined to form a compact set of components, whose each member is strongly focused on its task and interacts closely with other components to deliver the overall system functionality. The intrinsically iterative nature of software design offers opportunities for such re-organization of components.

However, this kind of design refinement usually depends on intuition, experience, and nameless "gut-feelings" of designers. In this paper we introduce the COMP-REF technique to guide such refinement of components using a set of metrics and a Linear Programming based optimization algorithm. Upon its application, the technique recommends *merging* of certain components, whose current roles in the system warrant their responsibilities be delegated to other components, and they be de-scoped. Recognizing the deeply reflective nature of software design, COMP-REF seeks to *complement* a designer's judgment by abstracting some basic objectives of component interaction and elucidating some of the design choices.

Before going into the details of our approach it will be helpful to clarify the meaning of certain terms in the context of this paper.

– A *requirement* is described as "... a design feature, property, or behavior of a system" by Booch, Rumbaugh, and Jacobson [5]. These authors call the statement of a system's requirements the assertion of a contract on what the system is expected to do. How the system does that is essentially the designer's call.
– A *component* carries out specific responsibilities and interacts with other components through its interfaces to collectively deliver the system's functionality (of course, within acceptable non-functional parameters).
– A *collaboration* is described in the *Unified Modeling Language Reference Manual, Second Edition* as a "... society of cooperating objects assembled to carry out some purpose" [26]. Components collaborate via messages to fulfill their tasks.
– "Merging" of a particular component will be taken to mean distributing its responsibilities to other components in the system and removing the component from the set of components fulfilling a given set of requirements. So *after* merging, a set of components will be *reduced* in number, but will be fulfilling the same set of requirements as before.
– In this paper "compact" in the context of a set of components will be taken to mean "designed to be small in size ...". [1]

We also assume COMP-REF technique is applicable in an iterative development scenario. This is a reasonable assumption, since even if the iterative and incremental model is not officially being followed, it is widely accepted that software design is an iterative activity.

In the next sections, we present a model for the software development space as a basis for the COMP-REF technique, introduce the ideas of *aptitude* and *concordance*, formally define our set of metrics, discuss the background and intuition behind the COMP-REF technique and present its steps. We then report results of experimental validation of the technique, highlight some related work and conclude with a discussion of open issues and directions of future work.

2 A Model for the Software Development Space

In order to examine the dynamics of software systems through a set of metrics, a *model* is needed to abstract the essential elements of interest.

[1] http://dictionary.reference.com/browse/compact

The development space consists of the set requirements $Req = \{R_1, ..., R_x\}$ of the system, which are fulfilled by the set of components $Comp = \{C_1, ..., C_y\}$.

We take *fulfillment* to be the satisfaction of any user defined criteria to judge whether a requirement has been implemented. Fulfillment involves delivering the *functionality* represented by a requirement. A set of mapping exists between requirements and components, we will call this *relationships.* At one end of a relationship is a requirement, at the other ends are all the components needed to fulfill it. Requirements also mesh with one another – some requirements are linked to other requirements, as all of them belong to the same system, and collectively specify the overall scope of the system's functionality. The links between requirements are referred to as *connections.* From the designer's point of view, of most interest is the interplay of components. To fulfill requirements, components need to collaborate in some optimal ways, this is referred to as the *interaction* of components.

Thus one aspect of the design problem may be viewed as: *given a set of connected requirements, how to devise a set of interacting components, such that the requirements and components are able to forge relationships that deliver the system's functionality within given constraints?*

Based on this model, the COMP-REF technique uses metrics to examine the interaction of components and suggest how responsibilities can be re-aligned. Before the metrics are formally defined, we introduce the notions of *aptitude* and *concordance* in the next section.

3 The Ideas of *Aptitude* and *Concordance*

Every software component exists to perform specific tasks, which may be called its *responsibilities.* The canons of good software design recommend that each component be entrusted with one primary responsibility. In practicality, components may end up being given more than one task, but it is important to try and ensure they have one primary responsibility. Whether components have one or more responsibilities, they can not perform their tasks entirely by themselves, without any interaction with other components. This is specially true for the so-called *business objects* – components containing the business logic of an application. The extent to which a component has to interact with other components to fulfill its core functionality is an important consideration. If a component's responsibilities are strongly focused on a particular line of functionality, its interactions with other components can be expected to be less disparate. Let us take *aptitude* to denote the quality of a component that reflects how coherent its responsibilities are. Intuitively, the *Aptitude Index* measures the extent to which a component (one among a set fulfilling a system's requirements) is coherent in terms of the various tasks it is expected to perform.

As reflected upon earlier, the essence of software design lies in the collaboration of components to collectively deliver a system's functionality within given constraints. While it is important to consider the responsibility of individual components, it is also imperative that inter-component interaction be clearly understood. Software components need to work together in a spirit of harmony if they have to fulfill requirements through the best utilization of resources. Let us take *concordance* to denote such cooperation amongst components. How do we recognize such cooperation? It is manifested in the ways components share the different tasks associated with fulfilling a requirement. Some of the symptoms of less than desirable cooperation are replication of functionality – different components doing the same task for different contexts, components not honoring their interfaces (with other components) in the tasks they perform, one component trying to do everything by itself etc. The idea of concordance is an antithesis to all such undesirable characteristics – it is the quality which delegates the functionality of a system across its set of components in a way such that it is evenly distributed, and each task goes to the component most well positioned to carry it out. Intuitively, the metric *Concordance Index* measures the extent to which a component is concordant in relation to its peer components in the system.

How do these ideas relate to cohesion and coupling? Cohesion is variously defined as "... software property that binds together the various statements and other smaller modules comprising the module" [16] and "... attribute of a software unit or module that refers to the relatedness of module components" [4]. (In the latter quote, "component" has been used in the sense of part of a whole, rather than a unit of software as is its usual meaning in this paper.) Thus cohesion is predominantly an *intra-component* idea – pointing to some feature of a module that closely relates its constituents to one another. But as discussed above, concordance carries the notion of concord or harmony, signifying the spirit of successful collaboration amongst components towards collective fulfillment of a system's requirements. Concordance is an *inter-component idea*; the concordance of a component can only be seen in the light of its interaction with other components.

Coupling has been defined as "... a measure of the interdependence between two software modules. It is an intermodule property" [16]. Thus coupling does not take into account the reasons for the so called "interdependence" – that modules (or components) need to cooperate with one another as they must together fulfill a set of connected requirements. In the same vein as concordance, aptitude is also an intra-component idea, which reflects on a component's need to rely on other components to fulfill its primary responsibility/responsibilities.

Cohesion and coupling are legacy ideas from the time when software systems were predominantly monolithic. In the age of distributed systems, successful software is built by carefully regulating the interaction of components, each of which are entrusted with clearly defined responsibilities. The perspectives of aptitude, and concordance – explored intuitively in this section, with metrics based on them formally defined in the next section – complement cohesion and coupling in helping recognize, isolate, and guide design choices that will lead to the development of usable, reliable, and evolvable software systems.

4 Defining the Metrics

Considering a set of requirements $Req = \{R_1, ..., R_x\}$ and a set of components $Comp = \{C_1, ..., C_y\}$ fulfilling it, we define the metrics in the following subsections:

4.1 Aptitude Index

The *Aptitude Index* seeks to measure how coherent a component is in terms of its responsibilities.

To each component C_m of $Comp$, we attach the following *properties* [12]. A *property* is a set of zero, one or more components.

- *Core* - $\alpha(m)$
- *Non-core* - $\beta(m)$
- *Adjunct* - $\gamma(m)$

$\alpha(m)$ represents the set of component(s) required to fulfill the primary responsibility of the component C_m. As already noted, sound design principles suggest the component itself should be in charge of its main function. Thus, most often $\alpha(m) = \{C_m\}$.

$\beta(m)$ represents the set of component(s) required to fulfill the secondary responsibilities of the component C_m. Such tasks may include utilities for accessing a database, date or currency calculations, logging, exception handling etc.

$\gamma(m)$ represents the component(s) that guide any conditional behavior of the component C_m. For example, for a component which calculates interest rates for bank customers with the proviso that rates may vary according to a customer *type* ("gold", "silver" etc.), an *Adjunct* would be the set of components that help determine a customer's type.

Definition 1. *The Aptitude Index $AI(m)$ for a component C_m is a relative measure of how much C_m depends on the interaction with other components for delivering its core functionality. It is the ratio of the number of components in $\alpha(m)$ to the sum of the number of components in $\alpha(m)$, $\beta(m)$, and $\gamma(m)$*

$$AI(m) = \frac{|\alpha(m)|}{|\alpha(m)| + |\beta(m)| + |\gamma(m)|} \tag{1}$$

4.2 Requirement Set

Definition 2. *The Requirement Set $RS(m)$ for a component C_m is the set of requirements that need C_m for their fulfillment.*

$$RS(m) = \{R_p, R_q, ...\} \tag{2}$$

where C_m participates in the fulfillment of R_p, R_q etc.

Evidently, for all C_m, $RS(m) \subseteq Req$.

4.3 Concordance Index

Definition 3. *The Concordance Index $CI(m)$ for a component C_m is a relative measure of the level of concordance between the requirements being fulfilled by C_m and those being fulfilled by other components of the same system.*

For a set of components $Comp = \{C_1,C_2,...,C_n,...,C_{y-1},C_y\}$ let,
$W = RS(1) \cup RS(2) \cup ... \cup RS(y-1) \cup RS(y)$

For a component C_m $(1 \le m \le y)$, let us define,
$X(m) = (RS(1) \cap RS(m)) \cup ... \cup ((RS(m-1) \cap RS(m)) \cup$
$((RS(m) \cap (RS(m+1)) \cup ... \cup ((RS(m) \cap (RS(y))$

Thus $X(m)$ denotes the set of requirements that are not only being fulfilled by C_m but also by some other component(s).

Expressed as a ratio, the *Concordance Index* $CI(m)$ for component C_m is:

$$CI(m) = \frac{|X(m)|}{|W|} \tag{3}$$

5 COMP-REF: A Technique to Refine the Organization of Components

COMP-REF is a technique to guide design decisions towards allocating responsibilities to a system's components. As in human enterprises, for a successful collaboration, software components are expected to carry out their tasks in a spirit of cooperation such that each component has clearly defined and specialized responsibilities, which it can deliver with reasonably limited amount of support from other components. *Aptitude Index* measures how self sufficient a component is in carrying out its responsibilities, and *Concordance Index* is a measure of the degree of its cooperation with other components in the fulfillment of the system's requirements. Evidently, it is desired that cooperation across components would be as high as possible, within the constraint that each requirement will be fulfilled by a limited number of components. This observation is used to formulate an objective function and a set of linear constraints whose solution gives a measure of how much each component is contributing to maximizing the concordance across the entire set of components. If a component is found to have low contribution (low value of the a_n variable corresponding to the component in the LP solution as explained below), *and* it is not significantly self-sufficient in carrying out its primary responsibility (low *Aptitude Index* value) the component is a candidate for being de-scoped and its tasks (which it was hardly executing on its own) distributed to other components. This results in a more compact set of components fulfilling the given requirements.

The goal of the COMP-REF technique is identified as *maximizing* the *Concordance Index* across all components, for a given set of requirements, in a particular iteration of development, within the constraints of *not* increasing the number of components currently participating in the fulfillment of each requirement.

A new variable a_n $(a_n \in [0,1])$ is introduced corresponding to each component C_n, $1 \le n \le N$, where $N =$ the total number of components in the system. The values of a_n are arrived at from the LP solution. Intuitively, a_n for a component C_n can be taken to indicate the extent to which C_n contributes to maximizing the *Concordance Index* across all components. As we shall see later, the a_n values will help us decide which components to merge.

The LP formulation can be represented as:

$$\text{Maximize} \sum_{n=1}^{y} CI(n) a_n$$

Subject to: $\forall R_m \in Req$, $\sum_{n=1}^{y} a_n \leq p_m/N$, a_n such that $C_n \in CS(m)$. $p_m = |CS(m)|$. (As defined in [13], the *Component Set* $CS(m)$ for a requirement R_m is the set of components required to fulfill R_m.)

So, for a system with x requirements and y components, the objective function will have y terms and there will be x linear constraints.

The COMP-REF technique is summarized as: Given a set of requirements $Req = \{R_1, ..., R_x\}$ and a set of components $Comp = \{C_1, ..., C_y\}$ fulfilling it in iteration I_z of development,

- STEP 0: Review *Req* and *Comp* for new or modified requirements and/or components compared to previous iteration.
- STEP 1: Calculate the *Aptitude Index* for each component.
- STEP 2: Calculate the *Requirement Set* for each component.
- STEP 3: Calculate the *Concordance Index* for each component.
- STEP 4: Formulate the objective function and the set of linear constraints.
- STEP 5: Solve the LP formulation for the values of a_n
- STEP 6: For each component C_n, check:
  - Condition 6.1: a_n has a low value compared to that of other components? (If yes, implies C_n is not contributing significantly to maximizing the concordance across the components.)
  - Condition 6.2: $AI(n)$ has a low value compared to that of other components? (If yes, implies C_n has to rely heavily on other components for delivering its core functionality.)
- STEP 7: **If** *both* conditions 6.1 and 6.2 hold TRUE, GOTO STEP 8, **else** GOTO STEP 10
- STEP 8: For C_n, check:
  - Condition 8.1: Upon merging C_n with other components, in the resulting set $\tilde{Comp}$ of q components (say), $CI(q) \neq 0$ for all q? (If yes, implies resulting set of q components has more than one component).
- STEP 9: **If** condition 8.1 is TRUE, C_n is a candidate for being merged; after merging components C_n GOTO STEP 0, starting with *Req* and $\tilde{Comp}$, **else** GOTO STEP 10.
- STEP 10: Wait for the next iteration.

6 Experimental Validation

In this section we present results from our experimental validation of the COMP-REF technique.

6.1 Validation Strategy

We have applied the COMP-REF technique on the following variety of scenarios to better understand its utility and limitations.

- **A "text-book" example** – *The Osbert Oglesby Case Study* is presented in Schach's software engineering textbook [27] as a software development project across life cycle phases and workflows. Using the Java and database components given as part of the design, we use the COMP-REF technique to suggest a reorganization of components and examine its implication on the design thinking outlined in the study.
- **The Financial Aid Application (FAA) project** – Florida State University's University Computing Services[2] is in charge of meeting the university's computing and networking goals. As a development project in 2006, existing paper based *Financial Aid Application* (FAA) was migrated to an online system. The development team took the previously used paper forms as the initial reference and built a system using JavaServer Pages (JSP), Java classes, and a back-end database to allow students to apply for financial aid over the Web. The COMP-REF technique is applied to suggest the merging of some of the components and its effect discussed on the overall design.
- **Morphbank: A Web-based Bioinformatics Application** – Morphbank[3] serves the biological research community as an open web repository of images. "It is currently being used to document specimens in natural history collections, to voucher DNA sequence data, and to share research results in disciplines such as taxonomy, morphometrics, comparative anatomy, and phylogenetics". The Morphbank system uses open standards and free software to store images and associated data and is accessible to any biologist interested in storing and sharing digital information of organisms. The COMP-REF technique investigates whether the overall design can be streamlined by a re-allocation of responsibilities across components and retiring some of them.
- **FileZilla: An open source project** – "FileZilla is a fast FTP and SFTP client for Windows with a lot of features. FileZilla Server is a reliable FTP server."[4] We use COMP-REF to examine FileZilla's allocation of component responsibilities.
- **The SCIT Workshop** – Symbiosis Center for Information Technology (SCIT)[5] is a leading academic institution in India, imparting technology and management education at the graduate level. Twenty five first-year students of the two year Master of Business Administration – Software Development and Management (MBA-SDM) graduate program participated in an workshop conducted by us. All the students had undergraduate degrees in science or engineering, and about half of them had prior industrial experience in software development. The students were divided into two groups with an even

[2] http://www.ucs.fsu.edu/
[3] http://www.morphbank.net
[4] http://sourceforge.net/projects/filezilla/
[5] http:///www.scit.edu

distribution of experience and exposure to software development ideas. Each group was in turn divided into two teams, *customer* and *developer*. The objective of the workshop was to explore how differently the same software system will be designed, with and without the use of the COMP-REF technique. Accordingly, each group was given the high level requirements of a contrived software project of building a Web application for a bank, where its customers can access different banking services. Within each group, the *developer* team interacted with the *customer* team to come up with a design in terms of interacting components that best met the requirements. The COMP-REF technique was applied in guiding the design choices of one group, which we will call Group A, while the other group, Group B, had no such facility. The workshop provided valuable insights into how COMP-REF can complement (and at times constrain) the intuition behind software design. We wish to thank Ms.Shaila Kagal, Director, SCIT for her help and support in conducting the study.

6.2 Presentation and Interpretation of the Results

Due to space constraints, we can not present each of the above validation scenarios in detail. Instead, we illustrate the application of COMP-REF in the FAA project in detail. The summary of all the validation scenarios are presented in Table 1.

Table 2 gives brief description of the requirements for the first iteration of the FAA project.

The $RS(m)$ column of Table 3 shows the *Requirement Set* for each component. Evidently, $W = \{R_1, R_2, R_3, R_4, R_5\}$ and $|W| = 5$. The $AI(m)$ and $CI(m)$ columns of Table 3 give the *Aptitude Index* and the *Concordance Index* values respectively for each component.

From the design artifacts, we noted that R_1 needs components C_1, C_5, C_{11} ($p_1 = 3$), R_2 needs $C_2, C_6, C_7, C_8, C_9, C_{11}$ ($p_2 = 6$), R_3 needs $C_3, C_6, C_7, C_8, C_9, C_{11}$ ($p_3 = 6$), R_4 needs $C_3, C_6, C_7, C_8, C_9, C_{11}$ ($p_4 = 6$), and R_5 needs $C_4, C_6, C_7, C_{10}, C_{11}$ ($p_5 = 5$) for their respective fulfillment. Evidently, in this case $N = 11$.

Based on the above, the objective function and the set of linear constraints was formulated as:
Maximize
$0.2a_1 + 0.2a_2 + 0.4a_3 + 0.2a_4 + 0.2a_5 + 0.8a_6 + 0.8a_7 + 0.4a_8 + 0.4a_9 + 0.2a_{10} + a_{11}$
Subject to

$a_1 + a_5 + a_{11} \leq 0.27$
$a_2 + a_6 + a_7 + a_8 + a_9 + a_{11} \leq 0.55$
$a_3 + a_6 + a_7 + a_8 + a_9 + a_{11} \leq 0.55$
$a_3 + a_6 + a_7 + a_8 + a_9 + a_{11} \leq 0.55$
$a_4 + a_6 + a_7 + a_{10} + a_{11} \leq 0.45$

Using the automated solver, GIPALS (General Interior-Point Linear Algorithm Solver)[6], the above LP formulation was solved (values in the a_n column of Table 3).

[6] http://www.optimalon.com/

Table 1. Experimental Validation: A Snapshot

System	**Scope and Technology**	**Parameters**	**Findings**
Osbert Oglesby Case Study	A detailed case study across software development life cycle workflows and phases presented in [27], using Java and database components.	Three requirements, eighteen components.	COMP-REF suggested 27% of the components can be merged with other components.
FAA project	Migration of paper based student aid application system to a Web based system, using Java and database components.	Five requirements, eleven components.	COMP-REF suggested 18% of the components can be merged with other components. Detailed calculation and interpretation given in Section 6.2 of this paper.
Morphbank	A Web-based collaborative biological research tool using PHP and database components. We studied the *Browse* functional area.	Seven requirements, eighty-one components.	The results of applying COMP-REF were inconclusive. Almost all the components executing common tasks across functional areas (around 75% of the total number of components) are suggested to be potential candidates for merging.
FileZilla	A fast and reliable cross-platform FTP, FTPS and SFTP client using C/C++.	As this is a software product vis-a-vis a project, there are no user defined requirements; three major lines of functionality and around one thirty eight components (ignoring header files).	While applying COMP-REF, difficulties were faced in correlating requirements with components. Assuming very coarse-grained requirements, COM-REF did not find valid justification for merging a notable percent of components.
SCIT workshop	Two separate groups designed a contrived software system of a Web based banking application using Java and database components. One group (Group A) was allowed the use of the COMP-REF technique, while the other group (Group B) was not. Group A and Group B were oblivious of one another's design choices.	Three requirements; Group A had eight components, Group B had twelve.	Group A's components 33% fewer than Group B's, they also had cleaner interfaces and smaller number of inter-component method calls. It appears COMP-REF helped Group A deliver the same functionality through a more compact set of component by being able to use COMP-REF in intermediate stages of design.

Let us examine how the COMP-REF technique can guide design decisions. Based on the a_n values in Table 3, evidently components $C_5, C_7, C_8, C_9, C_{10}$ have the least contribution to maximizing the objective function. So the tasks performed by these components may be delegated to other components. However, as mandated by COMP-REF, another factor needs be taken into account

Table 2. Requirements for FAA: iteration I_1

Req ID	Brief Description
R_1	Display financial aid information to users.
R_2	Allow users to enter enrollment period and record the information after validation.
R_3	Allow users to enter FSU sessions and record the information after validation.
R_4	Allow users to enter expected summer resources and record the information after validation.
R_5	Display summary of the user's enrollment status.

Table 3. FAA case study: Metrics values and LP solution for iteration I_1

C_m	Component name	$RS(n)$	$\alpha(n)$	$\beta(n)$	$\gamma(n)$	$AI(n)$	$\|X(n)\|$	$CI(n)$	a_n
C_1	summary.jsp	R_1	C_1	C_5, C_{11}	-	0.33	1	0.2	0.25
C_2	summer_instructions.jsp	R_2	C_2	C_8, C_9, C_6,C_{11}	C_7	0.17	1	0.2	0.4
C_3	summer_app.jsp	R_3, R_4	C_3	C_8, C_9, C_6,C_{11}	C_7	0.17	2	0.4	0.4
C_4	alerts_summary.jsp	R_5	C_4	C_{10},C_6,C_{11}	C_7	0.2	1	0.2	0.3
C_5	RetrieveSummerData.java	R_1	C_5	C_8, C_{11}	-	0.33	1	0.2	0
C_6	SummerApplication.java	R_2, R_3, R_4, R_5	C_6	C_8, C_9	C_3	0.25	4	0.8	0.13
C_7	SummerApplicationUtils.java	R_2, R_3, R_4, R_5	C_7	-	-	1	4	0.8	0
C_8	ValidateSummerApplication.java	R_2, R_3, R_4	C_8	-	-	1	2	0.4	0
C_9	SaveSummerApplication.java	R_2, R_3, R_4	C_9	C_{10}, C_{11}	C_3	0.25	2	0.4	0
C_{10}	RetrieveSummerApplication	R_5	C_{10}	-	C_7	0.5	1	0.2	0
C_{11}	StuSummerApp	R_1, R_2, R_3, R_4, R_5	C_{11}	-	-	1	5	1	0.02

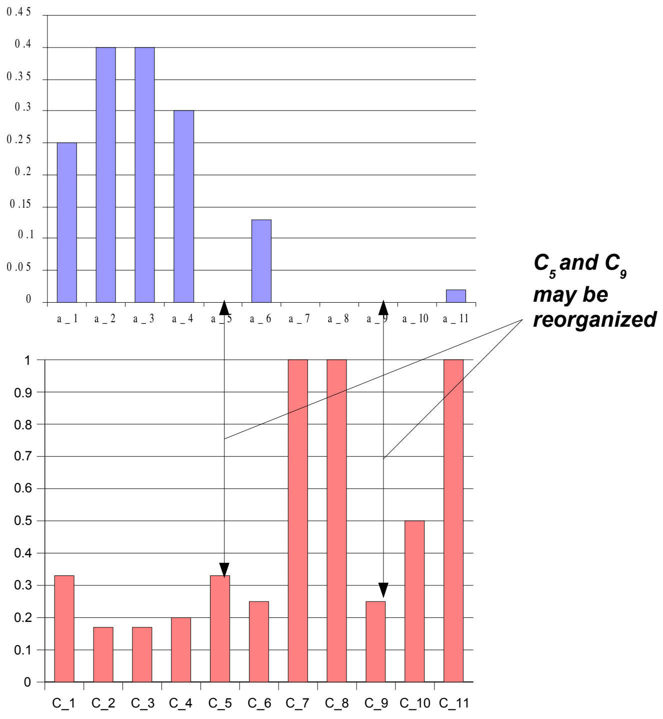

Fig. 1. a_n values from LP solution(top) and $AI(n)$ vs. C_n (bottom)

before deciding on the candidates for merging. How self-sufficient are the components that are sought to be merged? We next turn to $AI(n)$ values for the components in Table 3. We notice, $AI(5) = 0.33$, $AI(7) = 1$, $AI(8) = 1$, $AI(9) = 0.25$, and $AI(10) = 0.5$. Thus C_7, C_8 and C_{10} have the highest *Aptitude Index* values. These are components delivering functionalities of general utility, user input validation and database access logic respectively – facilities used across the application. Thus it is expedient to keep them localized. But C_5 and C_9, as their relatively low values of $AI(n)$ suggest, need to interact significantly with other components to carry out their task. And given their negligible contribution to maximizing concordance; a helpful design choice would be to merge them with other components. A smaller set of high concordance components is preferred over a larger set of low concordance ones, as the former has lesser inter-component interaction, thereby leading to better resilience to modification of particular components due to requirement changes. Figure 1 summarizes these discussions, suggesting reorganization of the two components through merging.

Thus one cycle of application of the COMP-REF technique suggests the reduction of the number of components from eleven to nine (18%) in fulfilling the set of requirements for the first iteration of the FAA project.

7 Related Work

Although it is common to use the terms *measure*, *measurement* and *metrics* in place of one another, some authors have underscored subtle distinctions [25], [2], [17]. For our discussion, we have taken *metrics* to mean "a set of specific measurements taken on a particular item or process" [3]. Metrics for analysis include the closely reviewed function point based approaches [1] and the Bang metric [15]. Card and Glass [6] have proposed software design complexity in terms of *structural complexity*, *data complexity* and *system complexity*. [23] identifies some important uses of complexity metrics. Fenton underscores the challenges of trying to formulate general software complexity measures [17]. Chidamber and Kemerer present a widely referenced set of object oriented software metrics in [7], [8]. Harrison, Counsell and Nithi have evaluated a group of metrics for calibrating object-oriented design [19].

Freeman's paper, *Automating Software Design*, is one of the earliest expositions of the ideas and issues relating to design automation [18]. Karimi et al. [21] report their experiences with the implementation of an automated software design assistant tool. Ciupke presents a tool based technique for analyzing legacy code to detect design problems [9]. O'Keeffe et al. [24] present an approach towards automatically improving Java design. Jackson's group are working on the *Alloy Analyzer* tool that employs "automated reasoning techniques that treat a software design problem as a giant puzzle to be solved" [20].

This current paper extends our ongoing research in understanding the effects of changing requirements on software systems, the role of metrics as design heuristics, and how the development life cycle can tune itself to the challenges of enterprise software development [13],[12], [11], [10], [14]. Particularly, [13] explores the relationship between requirements and components from another perspective.

8 Open Issues and Future Work

From the summary of the experimental results in Table 1, it is apparent COMP-REF is able to give conclusive recommendations in some of the validation scenarios. Let us reflect on the scenarios its suggestions are inconclusive. In the case of Morphbank, the system does not follow a clear separation of functionality in delegating responsibilities to its components. For FileZilla, it is difficult to extract clearly defined requirements and correlate them with corresponding components. This is not unusual for a software product, vis-a-vis a software development project, where a system is built to fulfill user given requirements. From the validation results so far, COMP-REF *appears* to work best for systems that have a clear set of requirements, follows the n-tier architecture paradigm and use object orientation to ensure a clear separation of concerns. We expect to scrutinize this conclusion further through ongoing case studies. The scalability of the technique also needs to be tested on very large scale systems and across many iterations of development.

COMP-REF suggests the merging of components. The in-built safeguards within the technique (STEP 8) ensures it will not lead to a single component *monolithic* system. The underlying assumption behind COMP-REF is that fewer components delivering the same functionality is better than a larger number of components, on grounds of more streamlined inter-component interaction, reduced communication overheads between members of the team developing the software, and better localization of the effects of inevitable changes in requirements [13]. In some cases there may be a need to *split* components instead of merging them. We plan to extend the technique to cover this aspect in future work. We are also working on developing an automated tool using the Eclipse platform [7] that will parse design artifacts (such as Unified Modeling Language diagrams), apply COMP-REF and present a set of recommendations. This tool integrates COMP-REF with our earlier work on a mechanism to track the effects of changing requirements on software systems [13]. Initial results from applying the tool are very promising.

9 Conclusions

In this paper we presented COMP-REF as a promising technique to guide the organization of components in software systems. COMP-REF is meant to complement, and certainly not replace, the intuitive and subjective aspects of software design. Results from applying the technique on a variety of systems were presented. Experimental data suggests COMP-REF works best for object-oriented

[7] http://www.eclipse.org/

systems using n-tiered architecture that fulfill user requirements. We plan to refine the technique through further validation and extend it into a fully automated framework for guiding analysis and design of software systems.

References

1. Albrecht, A.: Measuring Application Development Productivity. In: Proc. Joint SHARE/GUIDE/IBM Application Development Symposium, October 1979, pp. 83–92 (1979)
2. Baker, A.L., Bieman, J.M., Fenton, N., Gustafson, D.A., Melton, A., Whitty, R.: A philosophy for software measurement. J. Syst. Softw. 12(3), 277–281 (1990)
3. Berard, E.V.: Metrics for object-oriented software engineering (1995), `http://www.ipipan.gda.pl/~marek/objects/TOA/moose.html`
4. Bieman, J.M., Ott, L.M.: Measuring functional cohesion. IEEE Trans. Softw. Eng. 20(8), 644–657 (1994)
5. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide, 2nd edn. Addison-Wesley, Reading (2005)
6. Card, D.N., Glass, R.L.: Measuring Software Design Quality. Prentice-Hall, Englewood Cliffs (1990)
7. Chidamber, S.R., Kemerer, C.F.: Towards a metrics suite for object oriented design. In: OOPSLA 1991: Conference proceedings on Object-oriented programming systems, languages, and applications, pp. 197–211. ACM Press, New York (1991)
8. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20(6), 476–493 (1994)
9. Ciupke, O.: Automatic detection of design problems in object-oriented reengineering. In: TOOLS 1999: Proceedings of the Technology of Object-Oriented Languages and Systems, Washington, DC, USA, p. 18. IEEE Computer Society Press, Los Alamitos (1999)
10. Datta, S.: Integrating the furps+ model with use cases - a metrics driven approach. In: Supplementary Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE2005), Chicago, IL, November 7–11, 2005, pp. 4–51–4–52 (2005)
11. Datta, S.: Agility measurement index: a metric for the crossroads of software development methodologies. In: ACM-SE 44: Proceedings of the 44th annual southeast regional conference, pp. 271–273. ACM Press, New York (2006)
12. Datta, S.: Crosscutting score: an indicator metric for aspect orientation. In: ACM-SE 44: Proceedings of the 44th annual southeast regional conference, pp. 204–208. ACM Press, New York (2006)
13. Datta, S., van Engelen, R.: Effects of changing requirements: a tracking mechanism for the analysis workflow. In: SAC 2006, pp. 1739–1744. ACM Press, New York (2006)
14. Datta, S., van Engelen, R., Gaitros, D., Jammigumpula, N.: Experiences with tracking the effects of changing requirements on morphbank: a web-based bioinformatics application. In: ACM-SE 45: Proceedings of the 45th annual southeast regional conference, pp. 413–418. ACM Press, New York (2007)
15. DeMarco, T.: Controlling Software Projects. Yourdon Press (1982)
16. Dhama, H.: Quantitative models of cohesion and coupling in software. In: Selected papers of the sixth annual Oregon workshop on Software metrics, pp. 65–74. Elsevier Science Inc., New York (1995)

17. Fenton, N.: Software measurement: A necessary scientific basis. IEEE Trans. Softw. Eng. 20(3), 199–206 (1994)
18. Freeman, P.: Automating software design. In: DAC 1973: Proceedings of the 10th workshop on Design automation, Piscataway, NJ, USA, pp. 62–67. IEEE Computer Society Press, Los Alamitos (1973)
19. Harrison, R., Counsell, S.J., Nithi, R.V.: An evaluation of the mood set of object-oriented software metrics. IEEE Trans. Softw. Eng. 24(6), 491–496 (1998)
20. Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT Press, Cambridge (2006)
21. Karimi, J., Konsynski, B.R.: An automated software design assistant. IEEE Trans. Softw. Eng. 14(2), 194–210 (1988)
22. Larman, C.: Applying UML and Patterns. Prentice Hall, Englewood Cliffs (1997)
23. McCabe, T.: A software complexity measure. IEEE Trans. Softw. Eng. SE-2, 308–320 (1976)
24. O'Keeffe, M., Cinneide, M.M.O.: A stochastic approach to automated design improvement. In: PPPJ 2003: Proceedings of the 2nd international conference on Principles and practice of programming in Java, pp. 59–62. Computer Science Press, Inc., New York (2003)
25. Pressman, R.S.: Software Engineering: A Practitioners Approach. McGraw-Hill, New York (2000)
26. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley, Reading (2005)
27. Schach, S.: Object-oriented and Classical Software Development, 6th edn., McGraw-Hill International Edition (2005)