An Examination of the Effects of Offshore and Outsourced Development on the Delegation of Responsibilities to Software Components

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

Abstract. Offshore and outsourced development are the latest facts of life of professional software building. The easily identifiable advantages of these trends – such as cost benefits, continuous delivery and support – have already been explored to considerable extent. But how does offshore and outsourced development affect the delegation of responsibilities to components of a software system? In this paper we investigate this question by applying the RESP-DIST technique on a set of real life case studies. Our RESP-DIST technique uses metrics and a linear programming based method to recommend the reorganization of components towards an expedient distribution of responsibilities. The case studies embody varying degrees of offshore and outsourced development. Results from the case studies lead to some interesting observations on whether and how offshore and outsourced development influences software design characteristics.

1 Introduction

The paradigm of offshore and outsourced software development involves distribution of life cycle activities and stakeholder interests across geographical, political, and cultural boundaries. In this paper we will use the phrase *dispersed development* to refer to offshore and outsourced software development. We use the term "dispersed" in the sense of distribution of software development resources and concerns across different directions and wide area.

We seek to examine whether dispersed development has any impact on how responsibilities are delegated to components in a software system. Towards this end, we will apply the RESP-DIST technique across a spectrum of software development projects and analyze the results. RESP-DIST is a mechanism to guide *RESPonsibility DISTribution* in components of a software system such that they are best able to collectively deliver the system's functionality. RESP-DIST uses the metrics *Mutation Index* [6], *Aptitude Index*, and *Concordance Index* [7] and a linear programming (LP) based algorithm [7] to recommend the *merging* or *splitting* (as defined in more detail in the next section) of components

* Corresponding author.

K. Berkling et al. (Eds.): SEAFOOD 2008, LNBIP 16, pp. 73–89, 2009.

to lead to an even distribution of responsibilities and resilience to requirement changes. As discussed in a later section, *aptitude* and *concordance* are the design characteristics which RESP-DIST considers while making its recommendations. By applying RESP-DIST across a set of software projects of varying dispersion in their development, we expect to discern whether and how offshore and outsourced development affects the responsibility delegation aspect of software design.

The remainder of this paper is organized as follows: In the next section we describe a model for the software development space which will serve as a foundation for applying RESP-DIST. The following section informally reviews some key concerns of software design, followed by the introduction of the ideas of aptitude and concordance. We then define the metrics and specify the RESP-DIST technique. Subsequently, results from applying RESP-DIST on five real life software projects are presented. The paper ends with a discussion of related work, open issues and planned future work, and conclusions.

2 A Model for the Software Development Space

The discussion of this paper is based on the following connotations of certain definitions:

- A *requirement* is described as "... a design feature, property, or behaviour of a system" by Booch, Rumbaugh, and Jacobson [2]. These authors refer to the statement of a system's requirements as the assertion of a contract on *what* the system is expected to do; *how* the system does that is essentially for the designer to decide.
- A *component* carries out specific responsibilities and interacts with other components through its interfaces to collectively deliver the system's functionality (within acceptable non-functional parameters).
- A *collaboration* is described in the *Unified Modelling Language Reference Manual, Second Edition* as a "... society of cooperating objects assembled to carry out some purpose" [18]. Components collaborate via messages to fulfil their tasks. In this paper "collaboration" and "interaction" will be used interchangeably.
- *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.
- *Splitting* of a particular component will be taken to mean distributing some its responsibilities to a new component in the system which will interact on its own with other components to collectively deliver the system's functionality. So *after* splitting, a set of components will be *increased* in number, but will be fulfilling the same set of requirements as before.

We now present an abstraction of how requirements are fulfilled by components.

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 [7].

Let the development space of a software system consist 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 *fulfilment* to be the satisfaction of any user defined criteria to judge whether a requirement has been implemented. Fulfilment involves delivering the *functionality* represented by a requirement. A set of mapping exists between requirements and components, we will call these *relationships*. At one end of a relationship is a requirement, at the other ends are all the components needed to fulfil 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 fulfil requirements, components need to collaborate in some useful ways, these are referred to as the *interactions* of components.

Based on this model, an important goal of software design can be stated 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 best deliver the system's functionality within given constraints?

3 Key Concerns of Software Design

To examine whether and how distributed development affects software design, we discuss the act of design in some detail.

We may say that the conception of a particular system's design is instantiated by allocating particular tasks to components and specifying their interaction with other components such that the set of components *collectively* fulfil the system's requirements within acceptable non-functional parameters such as performance etc. In this paper, we focus entirely on the functional aspect of design. How does one decide on allocating tasks and specifying interactions?

Larman has called the ability to assign responsibilities as a "desert-island skill" [16], to underline its criticality in the design process. Indeed, deciding *which component does what* remains a key 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. Whenever a new functionality comes to light by analyzing (new or modified) requirements, the designer's instinct is to spawn a new component and assign it the task for delivering that functionality. This new component acts as something of an initial *placeholder* for the new functionality; to be reconsidered later if necessary. With increasing accretion of functionality a system usually ends up having a large number of fine grained components. Why does the designer instinctively spawn a new component for a new functionality, and not just commandeer an existing component to deliver that functionality?

The instinct perhaps is inspired by one of the lasting credos of effective and elegant software design: shunning large, bloated units of code (the so called "Swiss army knife" or "do-it-all" components) in preference to smaller, more coherent, and closely collaborative ones. However, recognizing the inherently iterative nature of software design, there is always scope – sometimes a pressing need – for deciding to merge some components while splitting others as development proceeds. Merging helps consolidate related responsibilities, thereby decreasing redundant and sometimes costly method calls across components. It is a natural way to refine components and their interactions after new components had been spawned (often indiscriminately) earlier to address new functionalities. But often, splitting a component is an useful way to isolate the implementation of a piece of functionality that is undergoing frequent modifications due to changing requirements. Such isolation helps insulate other components and their interactions from the effects of requirements that change often.

Thus the design of a software system in terms of its components – their individual responsibilities and collective interaction – matures iteratively through merging and splitting. Over such repeated reorganizations, design objectives of expediently fulfilling requirements as well as being resilient to some of their changes, are progressively met. But how does one decide on which component to merge and which to split? This is one the most important concerns of the software designer, usually addressed through experience or intuition or nameless "gut-feelings". RESP-DIST brings in a degree of discipline and sensitivity into such decisions – the technique seeks to complement the best of designers' judgment, and constrict their worst. RESP-DIST leverages certain characteristics of a software system's design which we discuss next.

4 Delegation of Responsibilities in Software Design

Design is usually an overloaded word, even in the software development context. There are no universally accepted features of *good* design, while symptoms of *bad* design are easy to discern. In the model for the software development space presented in an earlier section, we highlighted one aspect of the design problem. Based on this aspect, we introduce *aptitude* and *concordance* [7] as two key characteristics of design.

Every software component exists to perform specific tasks, which may be called its *responsibilities*. Software design canons recommend that each component be entrusted with one primary responsibility. In practice, 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 fulfil 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. We 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.

The *Aptitude Index* [7] 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* [5]. 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 fulfil 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 fulfil 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 behaviour 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 $\gamma(m)$.*

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

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 fulfil 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 honouring 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* [7] measures the extent to which a component is concordant in relation to its peer components in the system.

Definition 2. *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.*

The Requirement Set $RS(m)$ for a component C_m is the set of requirements that need C_m for their fulfilment [7].

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 $(1 \leq m \leq y)$, 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:

$$CI(m) = \frac{|X(m)|}{|W|} \tag{2}$$

How do the ideas of aptitude and concordance 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" [8] and "... attribute of a software unit or module that refers to the relatedness of module components" [1]. (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 fulfilment 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" [8]. 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 fulfil a set of connected requirements. Aptitude is an intra-component idea, which reflects on a component's need to rely on other components to fulfil 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 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.

As mentioned earlier, one of the main drivers of design change is changing requirements. Let us take the term *mutation* to mean any change in a particular requirement that would require a modification in one or more components fulfilling either one or a combination of the *display*, *processing*, or *storage* aspects of the requirement. In keeping with the principle of separation of concerns, it is usually taken to be good design practice to assign specific components to deliver each of the display, processing, and storage aspects. Components (or sets of components) delegated to fulfil the display, processing, and storage aspects of requirement(s) map to the *stereotypes* of analysis classes: *boundary*, *control*, and *entity* [13]. Intuitively, the metric *Mutation Index* [6] measures the extent to which a requirement has changed from one iteration to another, in terms of its display, processing, and storage aspects.

For a system let $Req = \{R_1, R_2, ..., R_m..., R_x\}$ denote the set of requirements. Between iterations I_{z-1} and I_z each requirement is annotated with its $Mutation\ Value$; a combination of the symbols D, P and S. The symbols stand for:

$D \equiv Display(1)$
$P \equiv Processing(3)$
$S \equiv Storage(2)$

The parenthesized numbers denote the $Weights$ attached to each symbol. The combination of more than one symbol signifies the addition of their respective $Weights$, thus:

$PD \equiv DP \equiv 1 + 3 = 4$
$SD \equiv DS \equiv 1 + 2 = 3$
$SP \equiv PS \equiv 3 + 2 = 5$
$SPD \equiv ... \equiv DPS \equiv 1 + 3 + 2 = 6$

The *Weight* assigned to each category of components – *Display*, *Processing* and *Storage* – is a relative measure of their complexities. (Complexity here refers to how intense the design, implementation, and maintenance of a component are in terms of development effort.) *Processing* components usually embody application logic and are most design and implementation intensive. *Storage* components encapsulate the access and updating of application data stores; their level of complexity is usually lower than that of the *Processing* components but higher than *Display* ones. Accordingly, *Display*, *Processing* and *Storage* have been assigned the $Weights$ 1, 3 and 2 respectively. Exact values of $Weights$ may be varied from one project to another; the essential idea is to introduce a quantitative differentiation between the types of components.

Definition 3. *The Mutation Index $MI(m)$ for a requirement R_m is a relative measure of the extent to which the requirement has changed from one iteration to another in terms of the components needed to fulfil it.*

Expressed as a ratio, the $MI(m)$ for requirement R_m :

$$MI(n) = \frac{The\ Mutation\ Value\ for\ R_m}{The\ maximum\ Mutation\ Value} \tag{3}$$

In the next section, we present how the RES-DIST technique uses the metrics *Aptitude Index*, *Concordance Index*, and *Mutation Index* to recommend merging or splitting of components.

5 The RESP-DIST Technique

Software design is about striking a balance (often a very delicate one!) between diverse factors that influence the functioning of a system. The ideas of aptitude, concordance, and mutation as outlined earlier are such factors we will consider now. The RESP-DIST technique builds on a LP formulation to *maximize* 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 fulfilment of each requirement. Results from the LP solution are then examined in the light of the metric values and suggestions for merging or splitting components arrived at. (RESP-DIST is the enhanced version of the COMP-REF technique we proposed in [7] – the latter only guided merging of components without addressing situations where components may require to be split.)

A new variable a_n ($a_n \in [0, 1]$) is introduced corresponding to each component C_n, $1 \leq n \leq N$, 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 [6], the *Component Set* $CS(m)$ for a requirement R_m is the set of components required to fulfil 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, proceed to next step, **else** GO TO 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.
- STEP 10: Let $Comp'$ denote the resulting set of components after above steps have been performed. For each component $C_{n'}$ in $Comp'$:
  - 10.1 Calculate the average $MI(m)$ across all requirements in $RS(n')$. Let us call this $\bar{MI}(m)$.
  - 10.2 Identify the requirement R_m with the highest $MI(m)$ in $RS(n')$. Let us call this $MI(m)_{highest}$.
- STEP 11: For each component $C_{n'}$, check:
  - Condition 11.1: $AI(n')$ has a high value compared to that of other components? (If yes, implies component relies relatively less on other components for carrying out its primary responsibilities.)
  - Condition 11.2: $CI(n')$ has a low value compared to that of other components? (If yes, implies component collaborates relatively less with other components for collectively delivering the system's functionality.)
- STEP 12: **If** both conditions 11.1 and 11.2 hold TRUE for component $C_{n'}$, it is tending to be monolithic, doing all its activities by itself and collaborating less with other components. Thus the $C_{n'}$ is a candidate for being split; proceed to next step, **else** GO TO STEP 14.
- STEP 13: Repeat STEPs 10 to 12 for all components of $Comp'$. For the component for which conditions 11.1 and 11.2 hold TRUE, choose the ones with the highest $\bar{MI}(m)$ and split each into two components, one with the requirement corresponding to the respective $MI(m)_{highest}$ and the other with remaining requirements (if any) of the respective *Requirement Set*. If the component was fulfilling only one requirement, the responsibility for fulfilling the requirement's functionality may now be delegated to two components.
- STEP 14: Wait for the next iteration of development.

6 Experimental Validation

6.1 Validation Strategy

To explore whether or how dispersed development affects the distribution of responsibilities amongst software components, we have studied a number software projects, which vary significantly in their degrees of dispersion. The projects range from a single developer team, to an open source system being developed through a team whose members are located in different continents, a software system built by an in-house team of a large financial organization, and standalone utility systems built through remote collaboration. We discuss results from 5 such projects in the following subsections.

6.2 Presentation of the Results

Due to space constraints, we limit the detailed illustration of the application of RESP-DIST to one project in detail. The summary of all the validation scenarios are presented in Table 1.

Table 2 gives metrics values and the LP solution for an iteration of Project A. Note: The project had 8 requirements: $R_1, R_2, R_3, R_4, R_6, R_7, R_8, R_9$ with requirement R_5 having been de-scoped in an earlier iteration of development. In the table Avg $MI(m)$ denotes $\bar{MI}(m)$ and ${R_m}^h$ denotes the requirement R_m with the highest $MI(m)$ in $RS(n')$. $MI(m)$ and ${R_m}^h$ values are not applicable (NA) for C_4 since RESP-DIST recommends it to be merged as explained later.

From the design artefacts, we noted that R_1 needs components C_3, C_5, C_6 ($p_1 = 3$), R_2 needs C_5, C_7 ($p_2 = 2$), R_3 needs C_1, C_3, C_4 ($p_3 = 3$), R_4 needs C_2, C_3 ($p_4 = 2$), R_6 needs C_1, C_2, C_6 ($p_6 = 3$), R_7 needs C_2, C_6 ($p_7 = 2$), R_8 needs C_7 ($p_8 = 1$), and R_9 needs C_8 ($p_9 = 1$) for their respective fulfilments. Evidently, in this case $|W| = N = 8$.

Based on the above, the objective function and the set of linear constraints was formulated as:

Maximize

$0.25 * a_1 + 0.25 * a_2 + 0.5 * a_3 + 0.13 * a_4 + 0.25 * a_5 + 0.25 * a_6 + 0.13 * a_7 + 0.a_8$

Subject to

$a_3 + a_5 + a_6 \leq 0.38$

$a_1 + a_3 + a_4 \leq 0.38$

$a_2 + a_3 \leq 0.25$

$a_1 + a_2 + a_6 \leq 0.38$

$a_7 \leq 0.13$

$a_8 \leq 0.13$

The *linprog* LP solver of MATLAB [1] was used to arrive at the values of a_n in the Table 2. Let us examine how RESP-DIST can recommend the merging or splitting of components. Based on the a_n values in Table 2, evidently components C_2, C_4, C_6 have the least contribution to maximizing the objective function. So

[1] http://www.mathworks.com/

Table 1. Experimental Validation: A Snapshot

System	Scope and Technology	Salient Features	Findings
Project A	A 5 member team dispersed development project – with 1 member interfacing with the customer and other members located in another continent – to build an automated metrics driven tool to guide the software development life cycle activities. The system was released as an open source product.	8 requirements, 8 components; system developed using Java.	RESP-DIST recommended 1 component be merged, 1 component be split. Detailed calculations are given later in this section.
Project B	A 2 member team dispersed development project – with virtual collaboration between the team members – to build a standalone utility to execute standard text classification algorithms against bodies of text, allowing for different algorithm implementations to be added, configured and used. Among other uses, a spam detection application can use this utility to try out different detection algorithms.	8 requirements, 7 components; system developed using Java. The system was selected from a competition and integrated in a broader application framework. The developers had financial incentives.	RESP-DIST did not recommended merging of any components, but 2 components could be split.
Project C	A 2 member team dispersed development project – with virtual collaboration between the team members – to define, read, and build an object representation of an XML driven business work flow, allowing manipulation and execution of the workflow through a rich API interface for the easy addition of workflow operations.	11 requirements, 13 components; system developed using the .NET platform. The system was selected from a competition and integrated in a broader application framework. The developers had financial incentives.	RESP-DIST recommended merging of 3 components, and splitting of 2 components.
Project D	A 6 member team dispersed development project – with the developers and customers spread across two cities of the same country – to develop an email response management system for a very large financial company. The system allows for emails from users across six product segments to be processed and placed in designated queues for customer associates to respond, and deliver the responded back to the users within prescribed time limits.	5 requirements; 10 components; system developed using Java, Netscape Application Server (NAS), and Lotus Notes. Developers worked on the system as a part of their job responsibilities. The system has been running for several years, with around 100,000 users.	RESP-DIST recommended merging of 1 component, and splitting of 4 components.
Project E	A 1 member team project to build a Web based banking application which allowed users to check their profile and account information, send messages to the bank; and administrators to manage user accounts, transactions, and messages.	12 requirements, 28 components; system developed according to the Model-View-Controller (MVC) architectural pattern with J2EE and a Cloudscape database.	Result from applying RESP-DIST was inconclusive.

Table 2. Details for Project A

C_m	$RS(n)$	Avg $MI(m)$	${R_m}^h$	$\alpha(n)$	$\beta(n)$	$\gamma(n)$	$AI(n)$	$\lvert X(n)\rvert$	$CI(n)$	a_n
C_1	R_3, R_6	0	-	C_1	C_3, C_5, C_7	-	0.25	2	0.25	0.21
C_2	R_4, R_7	0	-	C_2	C_3, C_7	C_6	0.2	2	0.25	0.08
C_3	R_1, R_3, R_4, R_6	0.17	R_1	C_3	C_1, C_5, C_7	-	0.25	4	0.5	0.17
C_4	R_3	NA	NA	C_4	C_3, C_5	-	0.33	1	0.13	0
C_5	R_1, R_2	0.5	R_1	C_5	C_1	-	0.5	2	0.25	0.12
C_6	R_1, R_7	0.34	R_1	C_6	C_2, C_7	-	0.33	2	0.25	0.09
C_7	R_2, R_8	0.5	R_8	C_7	-	-	1	1	0.13	0.13
C_8	R_9	1	R_9	C_8	-	-	1	0	0	0.105

the tasks performed by these components may be delegated to other components. However, as mandated by RESP-DIST, another factor needs be taken into account before merging. How self-sufficient are the components that are sought to be merged? We thus turn to the $AI(n)$ values for the components in Table 2. We notice, $AI(2) = 0.2$, $AI(4) = 0.33$, $AI(6) = 0.33$. Out of these, C_4 is contributing nothing to maximizing concordance ($a_4 = 0$), and its $AI(n)$ value is not very high either (0.33 on a scale of 1). So a_4 can be merged with other components. Now we check for the highest $MI(m)$, which corresponds to C_8. C_8 also has a high $AI(8)$ value of 1 and a low $CI(8)$ value of 0. Thus C_8 is trying to do all its task by itself, without collaborating with other components – this is indeed a candidate for splitting. The R_m with the highest $MI(m)$ in $RS(8)$ is R_9 – in fact R_9 is the only requirement in this particular case fulfilled by C_8. So RESP-DIST recommends C_8 be split into two components, each fulfilling a part of R_9. Relating the recommendations to the actual components and requirements, we find that C_4 is an utility component in charge of carrying out some numerical calculations; whose tasks can very well be re-assigned to components which contain the business logic behind the calculations. On the other hand, R_9 is a requirement for extracting data from design artefacts. This is certainly a requirement of very large sweep and one likely to change frequently, as the data needs of the users change. Thus it is justifiable to have R_9 fulfilled by more than one component, to be able to better localize the effects of potential changes in this requirement. Figure 1 summarizes these discussions, indicating merging for C_4 and splitting for C_8.

6.3 Interpretation of the Results

We examined the factors of dispersed development that could potentially affect the outcome of applying the RESP-DIST technique on these projects.

The *Agile Manifesto* lists the principles behind agile software development – methodologies being increasingly adopted for delivering quality software in large and small projects in the industry, including those utilizing dispersed development [15]. The Manifesto mentions the following among a set of credos: "The most efficient and effective method of conveying information to and within a development team is face-to-face conversation", and "Business people and

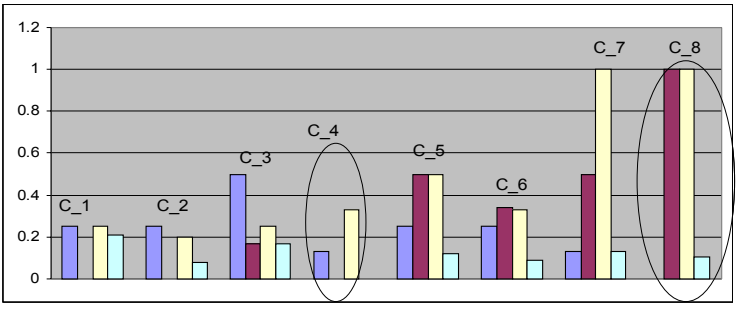

Fig. 1. Values of a_n, $AI(n)$, $\bar{MI}(m)$ and $CI(n)$ corresponding to the components C_1,...,C_8 for Project A. The RESP-DIST technique suggests merging for C_4 and splitting for C_8 – detailed discussion in section 6.3.

developers must work together daily throughout the project" [2]. Evidently, the very nature of dispersed development precludes this kind of interaction between those who commission and use a software system (these two groups may be identical or different, they are often clubbed together as *customers*) and those who develop it, that is, the *developers*.

We identify the key drivers of the effects of dispersed development on software design as *locational asynchrony* (LA), and *perceptional asynchrony* (PA). LA and PA may exist between customers and developers or within the development team. Locational asynchrony arises from factors like differences in geography and time zones. An example of LA would be the difficulty in explaining a simple architectural block diagram over email or telephone conversation, which can be easily accomplished with a white board and markers in a room of people (something similar to the *consequence of distance* highlighted in [10]). Perceptional asynchrony tends to be more subtle, and is caused by the complex interplay of stakeholder interests that dispersed development essentially entails. For example, in dispersed development scenarios, developers who have no direct interaction with the customer often find it hard to visualize the relevance of the module they are working on in the overall business context of the application – this is a manifestation of PA. With reference to Table 1, Project A has high LA but moderate PA; Projects B and C have moderate LA but high PA; Project D has moderate LA and low PA, while Project E has low LA and PA.

Apparently, there is no clear trend in the recommendations from RESP-DIST by way of merging or splitting components in Table 1 that suggests locational asynchrony or perceptional asynchrony have noticeable impact on how responsibilities are delegated. However, Projects B and C have a higher requirement to component ratio compared to others. This not only influences the way RESP-DIST runs on these projects but also indicates that moderate to high perceptional asynchrony may lead to a more *defensive* analysis of requirements – being

[2] http://agilemanifesto.org/principles.html

relatively unsure of the customers' intents developers are more comfortable dealing with finer grained requirements. The inconclusiveness of RESP-DIST's recommendation for Project E is also interesting. Project E's scenario represents by far the most *controlled conditions of development* amongst all the projects studied. It was developed by a single developer – a software engineer with more than 5+ years of industry experience – who had the mandate to refine the responsibility delegations amongst components repeatedly until the system delivered as expected. So naturally, RESP-DIST did not have much scope for suggesting merging or splitting of components. Also, compared to other projects Project E had a relatively unrelated set of requirements and relatively high number components with uniformly distributed responsibilities. Thus from the results related to Project A to D, RESP-DIST is seen to work best on a small set of closely related requirements and components. For a system with many requirements and components, it can be applied separately on *subsystems* that constitute the whole system.

7 Related Work

Freeman's paper, *Automating Software Design*, is one of the earliest expositions of the ideas and issues relating to design automation [9]. Karimi et al. [14] 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 [3]. Jackson's *Alloy Analyzer* tool employs "automated reasoning techniques that treat a software design problem as a giant puzzle to be solved" [12].

Collaboration platforms for offshore software development are evaluated in [17]. Shami et al. simulate dispersed development scenarios [20] and a research agenda for this new way of software building is presented in [19].

Herbsleb and Grinter in their papers [10], [11] have taken a more *social* view of distributed software development. In terms of Conway's Law – *organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations* [4] – Herbsleb and Grinter seek to establish the importance of the match between how software components collaborate and how the members of the teams that develop the software components collaborate.

8 Open Issues and Future Work

From the interpretation of the case study results, it is apparent the recommendations of merging or splitting components from applying the RESP-DIST technique are not significantly influenced by the degree of dispersion in a project's development scenario in terms of their location or perceptional asynchronies. However, factors other than locational or perceptional asynchrony may also

stand to affect the delegation of responsibilities in some dispersed development projects. In future work we plan to develop mechanisms to investigate such situations.

The case studies we presented in this paper range from 1 member to 6 member development teams, 5 to 12 requirements, and 7 to 28 components. Evidently, these are small to medium projects. We expect the execution of the RESP-DIST technique to scale smoothly to larger systems – more requirements and components will only mean more terms and linear constraints, which can be handled easily by automated LP solvers. However, the ramifications of larger systems on the dynamics of dispersed development is something which can only be understood by further case studies, some of which are ongoing.

We also plan to fine-tune the RESP-DIST technique by studying more projects across a diverse range of technology and functional area. The applicability of techniques like RESP-DIST are highly enhanced with tool support. We are working on an automated tool that will take in design artefacts and/or code as input, apply RESP-DIST and suggest the merging or splitting of relevant component.

9 Conclusion

In this paper, we examined whether and how offshore and outsourced development influences the delegation of responsibilities to software components. We applied the RESP-DIST technique – which uses the metrics *Aptitude Index*, *Mutation Index* and *Dependency Index*, and a linear programming based algorithm to recommend reorganization of the responsibilities of a software system's components through merging or splitting – on a range of software projects that embody varying degrees of offshore and outsourced development. It appears that two aspects of offshore and outsourced development – what we call as locational asynchrony and perceptional asynchrony – do not have significant effect on how responsibilities are distributed in the projects studied. However these factors may influence the way requirements are abstracted by the development team, which in turn can influences the application of the RESP-DIST technique. We plan to extend our work in refining the RESP-DIST technique by conducting further case studies. We are also working on developing a software tool to automate the application of the RESP-DIST technique.

Acknowledgements

We wish to thank Sean Campion, Project Manager at TopCoder Inc.; Dr Animikh Sen, Executive Director of Strategic Planning and Program Development at Boca Raton Community Hospital; Kshitiz Goel, Pooja Mantri, Prerna Gandhi, and Sidharth Malhotra, graduate students at Symbiosis Center for Information Technology for their help with acquiring and analyzing some of the case study information. We would also like to thank the anonymous reviewers for their helpful comments and criticism.

References

1. Bieman, J.M., Ott, L.M.: Measuring functional cohesion. IEEE Trans. Softw. Eng. 20(8), 644–657 (1994)
2. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide, 2nd edn. Addison-Wesley, Reading (2005)
3. 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, Los Alamitos (1999)
4. Conway, M.: How do committees invent? Datamation Journal, 28–31 (April 1968)
5. 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)
6. Datta, S., van Engelen, R.: Effects of changing requirements: a tracking mechanism for the analysis workflow. In: SAC 2006: Proceedings of the 2006 ACM symposium on Applied computing, pp. 1739–1744. ACM Press, New York (2007)
7. Datta, S., van Engelen, R.: Comp-ref: A technique to guide the delegation of responsibilities to components in software systems. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 332–346. Springer, Heidelberg (2008)
8. 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., Amsterdam (1995)
9. Freeman, P.: Automating software design. In: DAC 1973: Proceedings of the 10th workshop on Design automation, Piscataway, NJ, USA, pp. 62–67. IEEE Press, Los Alamitos (1973)
10. Herbsleb, J.D., Grinter, R.E.: Architectures, coordination, and distance: Conway's law and beyond. IEEE Softw. 16(5), 63–70 (1999)
11. Herbsleb, J.D., Grinter, R.E.: Splitting the organization and integrating the code: Conway's law revisited. In: ICSE 1999: Proceedings of the 21st international conference on Software engineering, pp. 85–95. IEEE Computer Society Press, Los Alamitos (1999)
12. Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT Press, Cambridge (2006)
13. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley, Reading (1999)
14. Karimi, J., Konsynski, B.R.: An automated software design assistant. IEEE Trans. Softw. Eng. 14(2), 194–210 (1988)
15. Kornstadt, A., Sauer, J.: Mastering dual-shore development - the tools and materials approach adapted to agile offshoring. In: Meyer, B., Joseph, M. (eds.) SEAFOOD 2007. LNCS, vol. 4716, pp. 83–95. Springer, Heidelberg (2007)
16. Larman, C.: Applying UML and Patterns. Prentice Hall, Englewood Cliffs (1997)
17. Rodriguez, F., Geisser, M., Berkling, K., Hildenbrand, T.: Evaluating collaboration platforms for offshore software development scenarios. In: Meyer, B., Joseph, M. (eds.) SEAFOOD 2007. LNCS, vol. 4716, pp. 96–108. Springer, Heidelberg (2007)

18. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley, Reading (2005)
19. Sengupta, B., Chandra, S., Sinha, V.: A research agenda for distributed software development. In: ICSE 2006: Proceeding of the 28th international conference on Software engineering, pp. 731–740. ACM, New York (2006)
20. Shami, N.S., Bos, N., Wright, Z., Hoch, S., Kuan, K.Y., Olson, J., Olson, G.: An experimental simulation of multi-site software development. In: CASCON 2004: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research, pp. 255–266. IBM Press (2004)