On the Impact of Evolving Requirements-Architecture Dependencies: An Exploratory Study

Safoora Shakil Khan, Phil Greenwood, Alessandro Garcia, Awais Rashid

Lancaster University, UK
{shakilkh, greenwop, garciaa, marash}@comp.lancs.ac.uk

Abstract. Architecture design plays a significant role in the evolution of software systems, as it provides the prime realization of the driving requirements and their inter-dependencies. With the increasing volatility of software requirements nowadays, it is necessary to understand the correlation between evolving classical requirements dependencies and their impact on the architectural decomposition. In the context of this analysis, two questions arise: (i) what are the conventional categories of requirements dependencies that are more architecturally significant in terms of change impact? and (ii) to what extent those evolving dependencies tend to generate ripple effects through architectural modules and interfaces. In order to address these two questions, this paper first presents an analysis model that categorizes requirements dependencies. Second, we have performed an exploratory study, based on the change history analysis of a real-life Web-based information system, in order to gather the most architecturally-significant requirements dependencies from our model. We have systematically analyzed ten system releases, based on some qualitative and quantitative indicators, with respect to how the requirements-architecture dependencies and compositions evolved.

Keywords: Dependency analysis, traceability, software architecture, change impact analysis.

1 Introduction

Software architecture of information systems is the pivotal realization of requirements and their inter-dependencies as it encompasses the driving design decisions in order to satisfy stakeholders' needs [19]. Software architectures are often decomposed into a set of modules (components) and interfaces, typically designed for future evolution in order to avoid that the system succumbs in the presence of changes [19][20]. Thus, architectural design forms the backbone of the target system, and frequent non-systematic modifications to requirements dependencies can make the architecture fragile and cause short-term design degeneration [15][16][17]. As a result, software architects often need to understand, predict and trace the impact of evolving requirements dependencies on architectural designs [1][2][3].

However, there is not much empirical knowledge on the influence of different categories of changing requirements on architecture elements [1][2][3] so that designers can forecast change impact. As a result, they cannot thereafter effectively trace key problem-solution dependencies. Not surprisingly, requirements-architecture traceability and change impact analysis techniques are still in its infancy and limited support is provided to software architects. There is a growing body of traceability techniques emerging in the literature [15][16][17][22], but they do not provide an adequate end-to-end tracing between categories of requirements dependencies and architectural elements. Most of these approaches are based on mapping requirements to architecture using trace matrix or trace graph [24][25]. There are a few approaches [2][5][6][8][9] that focus on characterizing requirements dependencies but these approaches have not been extended to cope with evolving requirements-architecture relations. Most empirical studies in the literature focus on characterizing architectural changes [18], keeping architecture and implementation in synchronization [15][17] [22], or supporting change impact analysis on implementation artifacts [23].

This paper presents a first exploratory study in order to identify the potential factors associated with evolving requirements dependencies and their corresponding effects or architectural changes. In order to be able to perform the investigation, we have defined a requirements dependency model (Section 2); we also discuss how each category of requirements dependency is likely to affect the architectural decomposition in the presence of change to it. The dependency model is based on a systematic analysis of classical requirements engineering techniques [1][2][3][5]. In a second step, we have undergone a set of experimental procedures (Section 3) in order to analyze the impact of evolving requirements dependencies and architecture changes through the releases of a real-life Web-based information system called Health Watcher (HW). The goal of the analysis was to characterize (Section 4):

(i) how the nature of requirements dependencies can lead to tight or loose interconnections with architectural elements;

(ii) the most architecturally-significant requirements dependencies; and

(iii) how the requirements dependencies tend to evolve in a typical Web-based information system.

Our analysis was based on qualitative and quantitative indicators. Classical change impact metrics have also been used to quantify the requirements and architecture co-relations. We also contrast our findings with related work (Section 5), and provide some concluding remarks (Section 6).

2 From Requirements to Architecture: A Dependency Model

This section presents a dependency model that provides support to software analysts to understand how the requirements are being realized to the architecture components and their compositions. This model assists a number of software evolution tasks, such as: (i) the identification of dependencies that lead to tight or loose interconnections among requirements elements and architectural decompositions, and (ii) basic support for understanding significant architecture implications from the perspective of

requirements changes. We have defined a dependency taxonomy based on a systematic analysis of conventional requirements engineering approaches [7][14]. The analyst must keep in mind that more than one dependency can hold between requirements to architecture and it may be rare that individual dependency exist due to requirements characteristics. We have discussed six types of dependencies from our dependency model:

Goal Dependency. Goal dependency relates system's quality attributes at problem domain to their realization in solution domain (architecture and implementation). This dependency has been adapted from the conventional goal-oriented requirements engineering approaches [1][2][3] that define characteristics of systems. Dependency relates to requirements specifying quality of service (security, availability, performance, etc) and development (compatibility, adaptability, interoperability, etc) to component at architecture level. Consider the requirement 'system must be flexible in terms of the storage format [4]', i.e., to enhance variability and provide the user with the multiple options of storing data, such as arrays or different databases. This requirement will be linked to component providing persistence at the architecture level as it defines the objective of the system under construction.

Service Dependency.. Service dependency relates the requirements expressing behavioral or functional characteristics of the system to corresponding operations and functions at architecture. The trace connection among requirements-level operations to architecture will usually be intricate or fine grained, as it will relate to classes, operations, or interfaces at the architecture level. A service dependency may coexist with goal, conditional, temporal, etc., dependency as it provides operation to perform when certain goal or condition is met. Consider the requirement '...system provides user with the queried data …' [4], forms a service dependency with architecture. Operation for searching and retrieving the requested data of a particular query type is invoked at architecture level.

Conditional Dependency. Conditional dependency defines events that trigger services, processes, and tasks based on certain conditions, constraints, or decisions taken at the requirements level to their realization at the architectural level. The triggering can be autonomous or non-autonomous reaction to a condition, constraints, or decisions, for example, when smoke is detected the autonomous reaction of the system is to open the doors. This dependency has been inspired from programming and Meyer [26] 'Design by Contract'. Consider the requirement 'employee can make changes to when authenticated by the system as an employee [4]'. This requirement has conditional and service dependency with components of the architecture. The conditional dependency exists as the employee will not be granted access to perform restricted operations unless they have been verified as a valid employee.

Temporal Dependency. Temporal dependency relates requirements specifying time frame of an event to occur, processes to complete, or condition to hold true, to their realization at architecture. Temporal dependencies manifest often in requirements associated with real-time systems and distributed systems. Temporal dependency is closely related to conditional dependency and may usually co-exist. Consider the requirement 'terminate user's request if system does not respond within 5 seconds [4]', has temporal and conditional dependency with the architecture. The dependencies hold as condition needs to be true within the specified time frame for the system to proceed further.

Task Dependency. Task dependency traces the connection between artifacts which require response, input, or feedback from user for their completion. Task dependency forms a medium between user and system, allowing user to request for systems services. Consider the requirement 'employee chooses one of the given options (review, update, delete)' has task dependency as the system needs users input to invoke the corresponding service based on users request. The common notion between conditional and task dependency is that systems halts for response. The distinguishing notion between the two dependencies is that conditional dependency depends on input, response or feedback from other operations/services in the system. Retrospectively, a task dependency depends on input, response or feedback from user.

Infrastructure Dependency. Infrastructure defines the hardware and software of system. Infrastructure dependency relates the resources, infrastructures (networks, telecommunications, mobile, etc.), technical standard/details, and compatibility issues specified in stakeholder's requirement to the architecture conception/construction. This dependency has been adapted from Ramesh and Jarke [8] *resource dependency* and Grady [9] *implementation requirements*. Consider the requirement, 'application should be accessible via internet' , it has infrastructure dependency with architecture components as .the Web service is implemented using servlets.

3 Case Study and Evaluation Procedures

This section describes in detail the requirements and architectural characteristics of the application used in our exploratory study (Sections 3.1), and the evaluation and analysis procedures (Sections 4.1 to 4.2). Health Watcher (HW) [10][11][12] is a typical Web information system and a real-life application that was chosen to support the empirical analysis in our exploratory study. The reason for selecting the HW case study is threefold. First, a rich set of HW artifacts and their releases were made available. For instance, for the analysis we have requirements specification (available from [13]), both use case descriptions and goal models. The architecture design is specified according to two fundamental architectural views, the module and component-connector views. Also, deployment-related decisions are embedded in these views. As a result, these requirements and architecture artifacts (Sections 3.1) capture respectively a rich, complementary set of requirements-architecture dependencies according to our model (Section 2).

Second, the original HW implementation and its ten releases [4] are available in three programming languages, Java, AspectJ, and CaesarJ. As a consequence, HW architecture is basically realized according to two architectural designs: a layered OO version and a layered AO version. None of these artifacts have been specially prepared or modified for our exploratory study, which in turn makes the analyzed base of changes more representative from realistic software maintenance scenarios. Third, this application has been used for implementation-level maintenance analyses [10][11][12] allowing us to correlate our findings with their results. Finally, in the investigation reported in [10], additional changes were applied to the HW application leading to ten implementation releases.

The study is divided as follows: (i) definition of change metrics (section 3.3), (ii) identification of requirements to architecture dependencies (section 4.1), (iii) application of change scenarios to assess change impact (architectural models and code are visited) and measure change propagation to identify the architecture-level changes in terms of classes, interfaces and compositions (section 4.2), and (iv) analysis of the assessments in order to identify the architecturally-significant dependencies (section 4.3) in the presence of changes.

3.1 Health Watcher Requirements and Architecture

The Health Watcher application is a Web-based information system which allows online access to register complaints, read health notices, and query regarding health issues. Employees can record, update, delete, print, search, change the records stored in the HW repository (in form of tables) after being authenticated as HW employee, i.e., by providing correct login name and password. Citizen can register complaints that system registers in the repository and generates a complaint code. The initial version of the HW system lacked flexibility and incapability to support generic Web applications as it was bound to specialized Web services. Also, the initial version provided limited functionality for a limited set of data, for example the system only allowed to query and update health units and complaints.

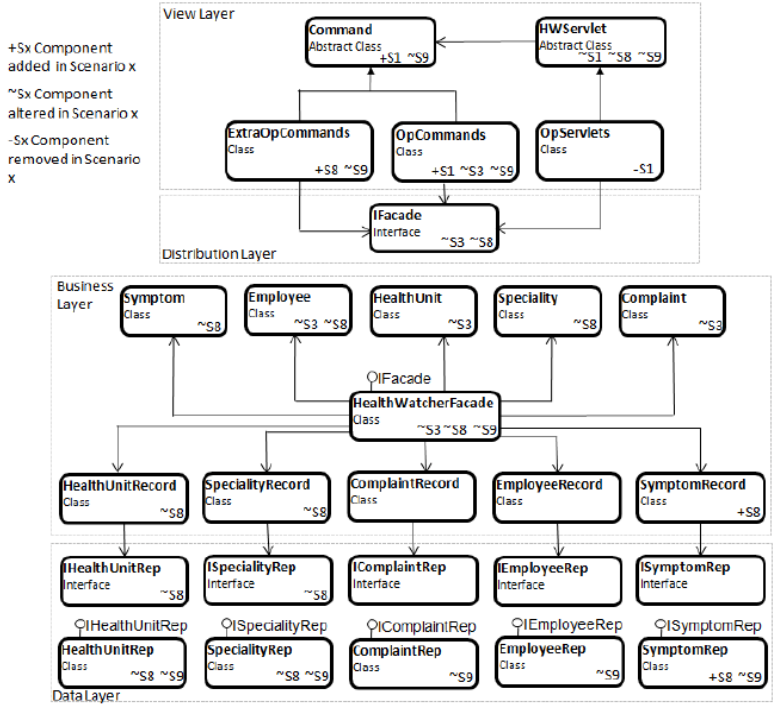

Fig.1. Module view of the Health Watcher system [10]

We have analyzed both OO and AO architectures of the HW system. The AO architecture modularizes concurrency, distribution, and persistence as aspects. Due to space limitation, our description focuses on the OO architectural design; detailed discussion about the AO architecture version is available at multiple sources [4][10][14][21]. Figure 1 shows the module view of the OO version for the HW system [10] that realizes the layered architectural style. It comprises of four layers: view, distribution, business, and data. Citizens access Web pages to query and/or register complaints. Multiple users can access the HW system simultaneously through Java Servlets, captured by the view layer (Figure 1), which decomposes into two main modules `HWServlet` and `OpServlets`. The Distribution module provides the interface `IFacade` to enable the access the HW services implemented in Business layer. The latter comprises of a number of modules, such as: `HealthUnitRecord`, `SpecialtyRecord`, `ComplaintRecord`, `EmployeeRecord`, and `SymptomRecord`. Each of these modules is invoked for specific operations requested by citizen/employee. For example, if the citizen has requested to query a health unit then `IFacade` provides access to business layer module `HealthUnitRecord` which accesses the HW database `HealthUnitRep` using interface `IHealthUnitRep`. A complete description of the HW architecture and implementation is available elsewhere [4][10][11][12]. Figure 1 also represents the implemented change scenarios. In particular, it points out the impacted modules for each scenario, which are sub-scripted by change type and scenario number.

3.2 Change Metrics

Our quantitative assessment is based on a metrics suite to identify propagation of requirements change on the elements of the OO and AO architecture design. The architectural views, the module and component-connector views consists of module/components that have class(es), operation(s), and port(s)/interface(s). The metrics will give quantitative values to analyze the dependency from perspective of change and architectural significance. The quantitative metrics to access change at architecture level are:

Concentration (C): it measures requirements dependency to the architecture components and their composition. Set of requirements may trace to a layer or more than one layer comprising of components. In the AO architecture version an aspect will be treated as a layer. The lowest value of *C* is one, which is also an indicator that the change will be occurring at intra-level, i.e., only affecting a layer. The highest value of *C* is the total number of layers in the architecture design, which is also an indicator that the change may cause ripple effect in the architecture.

Dispersion (D): it measures the percentage of components impacted by change in a layer or multiple layers. In equation 1, *Ec:* is number of effected classes, operations, and interfaces in architecture due to change and *Tc:* is the total number of components in a layer.

$$D = \left(\frac{Ec}{Tc}\right) \times 100 \quad \textbf{(1)}$$

We will calculate *D* for each layer and then take an average. For example, if change is concentrated (*C*) in two layers, then *D* will individually calculated for each layer and then an average will be taken to calculate the final value of *D*. If dispersion percentage is lower than 25% (1/4) it may be considered as a mild dispersion. If inter-component dispersion percentage is greater than 33% (1/3), even if change is concentrated in a layer it is considered severe.

Inclusion (I): it measures the number of components added when change is incorporated. It is not mandatory that each change introduces a new component, therefore, *I* can be 0 or any number of components added.

4 Empirical Results and Constraints

This section reports the evaluation outcomes and discussion, based on a systematic analysis of the nine change scenarios implemented in Java and AspectJ, which has led to ten releases for the HW architecture (Section 3.1).

4.1 Requirements-Architecture Dependency Analysis

This section provides a summary of requirements-architecture dependency analysis for HW using the model described in Section 2. This analysis involved the trace of the requirements to their module and composition counterparts in the architectural models. Figure 2 shows a representative set of examples on dependencies that are likely to exist from the HW requirements elements to the architecture components and their compositions. A few of requirements have been shortened due to space limitations:

R1 to R5 form *goal* and *infrastructure dependencies* with architecture layers of HW system. For example, R2 have *goal* and *infrastructure dependencies* as one of a few goals of health watcher is ease of access (i.e., available online) and usability, which are satisfied by implementing health watcher as a Web-based online application and servlet for GUI. R6, R7, and R9 have *task dependency* with view layer as user interacts with the system providing input or feedback to system to proceed further.

R8 forms *service dependency* with business and data layer. *Service dependency* holds as health watcher performs operations to store information entered by the user, parse the data entered by the user, creates a new instance of the appropriate complaint type, generates a unique identifier and assigns this to the new complaint, complainers address is parsed and saved.

R13 forms *service dependency* with business and data layer as `searchComplaint(int code)` and `search(String login)` classes are invoked to list the complaints and employees to be updated of HW system. R11 forms *conditional and service dependencies* because user can not access the restricted operations: update and register unless system verifies (user's login and password) them as valid user.

R15 and R16 are representative example of a few errors occurrence during the operation of the HW system. Error handling has *service and conditional dependencies* with view, distribution, business, and data layer, depending where the error occurred. Distribution, persistence, and concurrency are modularized as aspects in the AO architecture. Therefore, the requirements which did not have explicit trace dependency for OO version form dependencies in AO architecture design. R20 forms *service and conditional dependency* with concurrency component (`HWManagedSync` and `HWTimeStamp`). Timestamp provides functionality to avoid data inconsistency by applying timestamp field on the most recently modified data, storing it in the persistence mechanism.

Requirements	Dependency	Architecture
R1: System should be an online Web-based service	Goal & Infrast.	View
R2: System should have an easy to use GUI	Goal & Infrast.	View
R3: System must provide flexible storage mechanism	Goal & Infrast.	Business & Data
R4: System should be capable of running on separate machines	Goal & Infrast.	Distribution
R5: System must be able to handle 20 simultaneous users	Goal & Infrast.	Distribution
R6: … to register complaint citizen choose a complaint type: animal, food, or special	Task	View
R7: …user provides complaint details, place, and date/time	Task	View
R8: …complaint is stored on the server assigning an identification number to each stored complaint	Service	Business & Data
R9: … to query any information user selects query type: healthunit, specialty, or complaint	Task	View
R10: … based on selection of query type system retrieves the list	Service	Business & Data
R11: … to access restricted operations employee verifies themselves	Conditional & Service	View, Business & Data
R12: … verified employee selects healthunit, specialty, or complaint to update	Conditional & Task	View
R13: For particular selection: healthunit, specialty, complaint data is retrieved	Service	Business & Data
R14: updates for healthunit, specialty, or complaint are stored at server	Service	Business & Data
R15: … raise error message if invalid data is entered	Cond. & Service	View
R16: … if the system does not respond in 5sec raise an exception	Temp., Cond., & Service	View & Distribution

Fig. 2. Functional and non-functional requirements of Health Watcher system [13][14]

4.2 Evolving Dependencies' Impact on the Architecture

Our findings report how changing requirements dependencies (Section 4.1) tend to entail four categories of architecture-level changes, namely: adaptive restructuring, perfective modifications, incremental changes, and behavioral modifications. The analysis was guided by using an existing categorization of architectural modifications in the OO and AO module view [18]. This reference model systematically characterizes both the level of impact and severity of each type of architectural change. However, it provides an investigation on the correlation of requirements dependencies and architecture change categories, which is the key aim of our analysis.

Adaptive Restructuring of Layers. There were two change scenarios that implied adaptive restructurings. First, change scenario 1, involved restructuring of HW software to provide extensibility by separating servlets and promote GUI decoupling. This scenario evolved *goal and infrastructure dependencies* of R2 to service dependency with view layer. The changes only spanned over the inner architectural elements of the view layer. On the other hand, there was an evidence of high change dispersion (Table 1) as there were changes in `HWServlet` and the removal of `OpServlets` in view layer. This also led to the inclusion (I) of two components `Command` and `OpCommand` in view layer of the OO architecture.

Second, change scenario 5 led to deployment restructuring of the persistence mechanism in order to allow data storage in memory or database repository. *Goal and infrastructure dependencies* of R3 with business and data layer evolved to *goal, conditional, and infrastructure dependencies* in OO architecture, whereas in AO architecture *goal, conditional, and infrastructure dependencies* are formed with persistence (an aspectual component) and data layer. The reason for *conditional dependency* to emerge is because the change scenario has introduced a strategic design choice that altered the architecture design based on storage mechanism chosen at deployment time.

Table 1. Change scenarios and change assessment

	Description	**Type of Change**	**Evolving Req. dependencies**	**Assessment of Impacted Architecture**			
				OO/AO	**C**	**D**	**I**
1	Restructure Web based health watcher system to improve extensibility	Adaptive	Goal and Infrastructure	OO	1	50%	2
				AO	2	50%	1
2	Disable multiple updates once complaint state is CLOSED	Corrective	Services	OO	1	25%	1
			Conditional and Service	AO	1	25%	1
3	Improve maintainability by disassociating Update functionality from HW functions: health unit, specialty, and complaint	Perfective	Service	OO	1	28%	-
			Conditional and Service	AO	1	33%	1
4	HW system should support use of different distribution configurations	Perfective	Goal and Infrastructure	OO	2	39%	-
				AO	-	-	-
5	System must flexible in term of data storage	Adaptive	Goal and Infrastructure	OO	2	22%	1
				AO	2	33%	1
6	Ease the process of adding GUI	Perfective	Service	OO	1	33%	-
				AO	1	33%	-
7	Generalize distribution mechanism	Perfective	Goal and Infrastructure	OO	1	22%	1
				AO	-	-	-
8	Provide functionality to query more data types: symptoms and disease	Perfective	Service	OO	3	83%	3
			Conditional and Service	AO	6	50%	-
9	Modularize error handling strategies and provide better error recovery mechanisms	Perfective	Conditional and Service	OO	3	54%	-
				AO	4	57%	1

Perfective Modifications of Pivotal Architectural Services. Change scenario 3 is a perfective change modularizing update function, decoupling it from the rest of HW system, such as the disassociation of the health unit, specialty, and complaint entities in business layer. The set of requirement had *service dependency* with business layer and data layer. Change is concentrated in business layer with a dispersion of 28%, decoupling update functionality from `HealthWatcherFacade`, `Employee`, `HealthUnit`, and `Complaint` components in business layer. For the AO architecture, update function formed *service and conditional dependencies* with concurrency (aspect) and business layer. The reason to form dependencies with concurrency (aspect) is to achieve synchronization and maintain data consistency, which is achieved by applying timestamp field to the retrieved/updated complaint, health unit, and specialty data.

Change scenario 9, modularizes error and exception handling, decoupling it from the rest of the health watcher system. Similarly, change scenario 4 performs a perfective change as it modularizes the distribution mechanism, decoupling it from the rest of the health watcher system for OO architecture in order to facilitate deployment of different distribution configurations. But change scenario 4 is not applicable as distribution mechanism was modularized as an aspect in the initial AO architecture design.

Incremental Change of Component Interfaces. Change scenario 6 is an increment of release 2 that implements change scenario 1. It generalizes the request and response servlet parameters to enable inclusion of new operations and GUI. R2 entails *service dependency* to the view layer. *Dependency* remains unchanged due to the perfective nature of the modifications. Change is concentrated on the view layer with change dispersion lower than of change scenario 1 as it impacts a few operations in component interfaces within the view layer.

Change Scenario 7 is an increment of release 5 which implements change scenario 4. Incorporated change allows number of different distribution mechanisms to be configured for multiple servlets. Change is concentrated on the business layer of OO architecture with change dispersion lower than of release 5 (change scenario 4). Change scenario 7 has no effect/applicable as distribution mechanism was modularized as an aspect in the initial design of AO architecture.

Behavioral Modifications. Change scenarios 2, 8, and 9 modified/changed the intended behavior of system functions. Change scenario 2 refined the update complaint functionality, by allowing the complaint status to be set to close once complaint had been modified by employee. R14 implied on a *service dependency* with business and data layers for OO architecture, in AO architecture it formed *service and conditional dependencies* with concurrency (aspect) and business layer. For AO architecture the dependencies remained the same, whereas for OO architecture the dependency evolved to *conditional and service dependencies*, as every time a complaint is requested for update its status was checked. From the architectural perspective, a few operations in classes of `Complaint` component underwent some impact. In fact, the dispersion percentage was relatively low, i.e. 25%. The change has also encompassed the introduction of `State` component in business layer of OO and AO architecture.

The initial version of HW system only provided the option to query the health unit, complaint, and specialty. New functionality and querying options are provided with

respect to the symptoms and diseases entities in scenario 8. Querying functionality had *service dependency* with business layer in OO architecture. Interestingly, the change has propagated to 3 layers and rippled through the system. The change added `SymptomRecord` class which formed *service* dependency with business layer and `SymptomRep` class which formed *goal and infrastructure dependencies* with data layer of OO architecture. While, querying functionality forms *service and conditional dependencies* with concurrency (aspect), persistence (aspect), and business layer of AO architecture. Change to query behaviour impact 6 layers and added observer pattern. `SymptomRecord` and `SymptomRepositoryRDB` classes were added in AO architecture.

Scenario 9 comprises of two architecture-level changes: a) perfective and b) behavioral. The changes in scenario 9 had been incorporated in parallel therefore they were not considered under the incremental change category. The change introduced new exceptions that were not in the initial intended OO and AO architecture design: `CommunicationException`, `SQLPersistenceMechanismException`, and `RepositoryException`.

4.3 On the Architecturally-Significant Dependencies

This section discusses some findings on which types of changing requirements dependencies (Section 2) are likely to have widely-scoped, moderate or localized impact at the architecture design. The previous quantitative and qualitative analysis (Section 4.1) are used as the basis.

Architectural Pull. An analysis of Table 1, externalizes the fact that dependencies are orthogonal in nature (Section 2) and it is impossible to have strict separation between the dependencies due to nature of requirements. These dependencies may form weak or strong interconnection with the architecture elements, which we refer to as architecture pull. The dependencies pull the architecture in various directions which may provide an insight of dominant dependencies at architectural level from perspective of change, as discussed below.

Dominant Dependency for a Particular Type of Change. The orthogonal nature of dependencies raises some questions: how coexisting dependencies evolve? which dependency has dominant impact during change? does the impact degree of a dependency category vary based on heterogeneous change types?

Table 1, shows *goal and infrastructure dependencies* co-existed, they were impacted by adaptive change (scenario 1 and 5). The adaptive modifications led to high dispersion of architecture-level change. It is well understood that a goal dependency is significant at architecture level [1] but for the specific change *infrastructure dependency* was dominant as the changes were focused on the software architecture being adapted to new standards/techniques in order to improve both server and client performance and this change did not cause the system to deviate from its original design and objectives and, as a result, the *goal dependency* was of limited impact. Change in *goal dependency* may lead to significant changes as the system's objectives are changed which will lead to degeneration of architecture design.

According to Table 1, *usability and service dependencies* (scenario 6) were affected by perfective change. For the specific change scenario *usability dependency* played a minimum role in the architectural modifications, the dominant dependency was *service dependency* as it refined the operations provided by system. Based on the assumption of corrective change in scenario 2, dependencies might have equal importance and may not be dominant over the other from perspective of change.

Many requirements formed *conditional and service dependencies* with OO and AO architecture, as seen in Table 1. In our analysis, we observed that *service dependency* led to high dispersion when it involved changes in system functionality/behavior. We have noticed the similar trend for the *service dependency* co-existing with *conditional dependency*. *Conditional dependency* captures the behavior and structure in form of architectural design choices, decisions, and constraints, which form core of the architecture, which are implied on number of architectural elements. When architectural decisions and/or choices change it may lead to addition of new structure or behavior in the architecture causing a break down, degradation, or enhancement in the architecture. Therefore a *conditional dependency* plays a dominant role and qualifies as a significant dependency when it co-exists with *service dependency* as all the dependant process or services are checked to see if they satisfy the new condition, decision, or constraint.

Independent Dependencies. Up to this point, we have discussed the orthogonal or overlapping dependencies. Now the question arises: if it is possible for dependencies to exist independently. A careful analysis of the outcomes in section 4.1and 4.2 makes it evident that *goal dependency* is unlikely to exist independently. *Goal dependency* defines the system objectives or quality attributes that are achieved by operations, services, software, and technical infrastructure, which have clear realization at architecture.

Dependencies with Minimum Architectural Impact. From analysis of the change scenarios we identified dependencies that are least likely to impact the architecture. The least significant dependency is *task dependency*. As task *dependency* facilitates user's interaction with the system through a medium (e.g., Web-browser, command prompt, etc.). Even if the backend software is enhanced/ modified (as in scenario 1) or service is modified/added (scenario 6) the front end remains the same, i.e., a Web browser. Based on scenario 1, if the GUI is changed, i.e., addition of radio buttons, drop down list, or check boxes, it will not introduce any change at architecture level, but at code level.

4.4 Study Constraints

Even though this study fully satisfies our initial goal of providing a first empirical investigation on the impact of evolving requirements dependencies on architectural changes, our procedures have some limitations. These limitations will contribute to further explorations using other experimental procedures and systems as targets.

Our analysis concentrates on the change history of one software system providing a single point of observation. Our target case study is a representative choice of Web-based information systems for several reasons: the HW realizes n-tier architecture that is one of the most common design alternatives implemented by deployed Web

systems [10][21], HW functional and non-functional requirements are consistently part of applications in this specific domain, changes are heterogeneous and representative of real change requests within software projects from this nature. More importantly, the design of the HW system realizes the best design practices [4] and have being systematically being enhanced through the years [10]. It provides evidence that the requirements-architecture co-changes are not merely a matter of lack of systematic design choices.

Of course, strong conclusion can not be drawn by analyzing a single system as dependencies may (or may not) vary depending based on web-based information system's quality attributes. For example, an online banking system has security and privacy as its key quality attributes. Similarly an online comparison system has completeness and correctness as its quality attributes, b) we analyze a single architectural view, therefore, we are unsure of the dependencies that may exist for other views, how the architecture will be affected, on incorporating change and will similar set of dependencies be architecturally significant, and c) we only focused on analyzing the dependencies that evolved due to incorporated change scenarios. This does not give us a clear idea on change impact on other dependencies.

5 Related Work

A few conventional requirements modeling languages and methodologies [1][2][3] explicitly attempt to support a more straightforward derivation of architecture designs. They rely on the provision of means to enable the requirements engineers to reason about the system goals and how they are operationalized in terms of architectural elements. However, the sole use of these approaches is not sufficient to estimate the impact of requirements changes on the derived architectures. Requirements change is inevitable and incorporating these changes involve: huge cost and effort, risk of architecture degeneration, and undesirable system deviation from its original design.

One way to assess effect of change is to apply impact analysis techniques. There are many code-level change impact analysis approaches, but a scarcity of impact analysis support targeted at architectural evolution. Only a few authors [15][16][17] define impact analysis techniques for architecture designs, but their focus is restricted to the scope of architecture-level changes. They are oblivious to the way changes to the requirements and their inter-dependencies impact architectural decompositions. As a consequence, it makes it difficult to architects to estimate how specific change requests on requirements will affect the architecture elements. For understanding impact of requirements evolution on architecture it is fundamental to understand the trace dependencies that may hold between entities of the two artifacts.

Ramesh and Jarke [8] defined requirements trace dependencies: *goal, task, resource, and temporal* for trace managing requirements on the basis of nature of dependency. Grady [9] took a step further to define architectural requirements by defining *design, implementation, interface, and physical* requirements, which only provide a mechanism to analyze how these requirements are realized at architecture. William and Carver [18] characterized architectural change and specified their affect on logical and runtime architectural structure. Inspiring from works of Ramesh and

Jarke [8], Chung et. al. [2], William and Carver [18], and Grady [9] we have defined requirements to architecture dependency model to predict impact of requirements change based on dependencies and change type (corrective, adaptive, and perfective) on architecture. This has help predict change impact of dependencies and identify architecturally significant dependencies.

6 Conclusion and Future Work

This paper presents the outcomes of an exploratory study aimed at assessing the impact of typical changes in requirements dependencies on architecture design. From our exploratory study we have externalized the fact that requirement dependencies are often orthogonal in nature and, in many cases, there is no strict separation between the dependencies. We have analyzed how co-existing dependencies evolve, which dependencies have dominant impact during change and led to different "architectural pulls". Our study has shown that evolution of co-existing *conditional and services dependencies* is architecturally significant as for evolving *service dependency* it needs to be checked if the corresponding condition, constraints, and decision are satisfied or for evolving *conditional dependency* it needs to be checked if the process or services correspond to the condition. Similarly, co-existing *goal and infrastructure dependencies* are architecturally significant as change to system objectives may lead to architectural degeneration. Strong conclusion can not be drawn by analyzing a single system as dependencies may (or may not) vary depending based on Web-based information system's quality attributes. In order to validate our dependency model and identify other architecturally-significant dependencies, other studies should further analyze applications from different domains.

Acknowledgements. This work was partially supported by the European Commission grant IST-33710 - Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE), and the TAO project, funded by Lancaster University Research Committee.

References

1. Lamsweerde, v. A.: From System Goals to Software Architecture. Formal Methods for Software Architectures, M. Bernardo & P. Inverardi (eds), LNCS 2804, Springer-Verlag, 2003, 25-43
2. Chung, L. Nixon, A. B.,Yu, E., and Mylopoulous, J.: Non-functional Requirements in Software Engineering. Kluwer Academic Publishing, 1999.
3. Herold, S., et. al.: Towards Bridging the Gap between Goal-Oriented Requirements Engineering and Compositional Architecture Development. SHARK-ADI 2007..
4. Greenwood, P., et al.: Aspect Interaction and Design Stability: An Empirical Study (2007), Available from: http://www.comp.lancs.ac.uk/computing/users/greenwop/ecoop07
5. Jacobson, I., Chirsterson, M., Jonsson, P., and Overgaard, G.: Object-Oriented Software Engineering: A Use Case Driven Approach. 4 ed: Addison-Wesley, 1992.

6. Chitchyan, R., et. al.,: Semantics-based Composition for Aspect-Oriented Requirements Engineering. AOSD 2007, Vancouver, Canada. pp. 36-48
7. Chitchyan, R., et. al.: Survey of Aspect-Oriented Analysis and Design. AOSD-Europe Project Deliverable No: AOSD-Europe-ULANC-9. Editor(s): R. Chitchyan, A. Rashid.
8. Ramesh, B., Jarke, M.: Towards Reference Models for Requirements Traceability. IEEE Transactions on Software Engineering. 27(1), Jan 2001.
9. Grady, R.: Practical Software Metrics for Project Management and Process Improvement. Prentice-Hall, 1992.
10. Phil Greenwood et al: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. ECOOP 2007, pp. 176-200.
11. Sant'Anna, C., et. al.: On the Modularity of Software Architectures: A Concern-Driven Measurement Framework. ECSA 2007, pp. 207-224.
12. Cacho, N., et. al.: Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. AOSD 2006, pp. 109-121.
13. TAO: A testbed for Aspect Oriented Software Development (2007), available from: http://www.comp.lancs.ac.uk/~greenwop/tao/
14. Khan, S. S., et. al.: On On the Interplay of Requirements Dependencies and Architecture Evolution: An Exploratory Study (2007), Available from: http://www.comp.lancs.ac.uk/~shakilkh/caise08
15. Feng, T. and Maletic, I. J.: Applying Dynamic Change Impact Analysis in Component-based Architecture Design. 7th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'06)
16. Riebisch, M. and Wohlfarth, S.: Introducing Impact Analysis for Architectural Decisions. ECBS 2007, pp:381 – 392
17. Zhao, J., et. al.: Change Impact Analysis to Support Architectural Evolution.. Software Maintenance: Research and Practice. Vol. 14, No. 5. (2002), pp. 317-333.
18. Williams, J. B. and Carver, C. J.: Characterizing Software Architecture Changes: An Initial Study. 1st Intl. Symposium on Empirical Software Engineering and Measurement (ESEM 2007) pp. 410-419.
19. Clement, P., et. al.: Documenting Software Architectures: Views and Beyond. SEI Series in Software Engineering, Addison Wesley (2002).
20. Shaw, M. and Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Inc. 1996.
21. Soares, S., et. al.: Distribution and Persistence as Aspects. Software Practice and Experience (2006).
22. Murta,.G. P. L., et. al.: ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links. ASE 2006, pp. 135-144.
23. Lee, M. and Offutt, J.: Change Impact Analysis of Object-Oriented Software. Published by George Mason University. Pages: 193,(1998).
24. Browning, T. R., et. al.: Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions. IEEE Transaction on Engineering Management. Vol. 48, Issue. 3 (2001), pp. 292-306.
25. Sangal, N., et. al.: Using Dependency Models to Manage Complex Software Architecture. OOPSLA 2005, San Diego, California, USA.
26. Meyer, B.: Applying "Design by Contract". Computer, Vol. 25, (10), pp. 40-51, 1992.