A Survey on Conflict and Dependency Analysis in the context of Software Development and Release Management

Guru Rajesh Kaustubh[1], Pratiba D[2]

[1]UG Student, Dept. of Computer Science Engineering, RV College of Engineering, Bangalore, Karnataka, India
[2]Assistant Professor, Dept. of Computer Science Engineering, RV College of Engineering, Bangalore, Karnataka, India

---***---

Abstract - *The Agile model of Software Engineering is widely regarded as the most popular model due to its multiple merits, but the adoption of the same to large scale software development has proven to be counter-productive due to various reasons. With the practice of documentation having been abandoned almost completely, we are in need of new approaches to keep everyone on the same page, in order for large scale enterprises to run smoothly with all of its diverse and spread out workforce and resources.*

Key Words: Software Development, Software Release, Conflict, Dependency, Model, Request for Change, Configuration Item

1. INTRODUCTION

With enterprises expanding beyond geographic boundaries at a rapid rate, thanks to the opportunities presented by the onset of the trend of globalization, most activities in companies involve multiple functional teams communicating across the globe and working hand in hand. We focus on two such activities in this paper – Software Development and Release Management. Software Development is the activity of constructing software to satisfy and cater to certain specified needs of the market (although the culture of documentation of requirements has drastically reduced in recent years due to the observed tendency of clients' requirements to constantly change as development progresses). Software Development is done by teams which generally have some common roles such as Project Manager, Product Owner, Development and Testing and so on. Some popular models of Software Engineering are Waterfall model, Iterative model and Agile model, of which Agile model is the most frequently used nowadays.

Software Release refers to the completely constructed version of a software, that is to be deployed and made available to the intended end user. As simple as it may sound, it is a complex process that involves many steps and sign-offs from multiple teams. A lot of stress testing and integration testing is done to make sure that the new features and extensions being included in the upcoming release do not break the functionalities of the current working version of the software. Virtually all companies are continuously upgrading and optimizing their products and services in order to stay relevant and competent in today's competitive market economy, which is what motivated us to probe into this topic and survey the research that has been done to deal with the various problems faced by Software Development and Release Management teams.

2. CHALLENGES FACED

2.1 Dependency

Dependency is a side effect of component-based software development. It is a relationship between modules due to which one of them cannot function without one or more of the other modules. Dependencies in software can get very complex as it so happens very often that different components are developed by different independent organizations, each of which controls the release of newer versions of its components [1]. Hence it is important to document dependencies, especially in component-based development, so that they don't create unnecessary hiccups further in the development process. Dependencies create a lot of unpredictability as it increases the possibility of future issues and failures during development. Dependencies can exist between different activities in the development process, between software artefacts and across teams and between team members [6].

2.2 Communication Challenges

This is another side effect of component-based software development, when development work is distributed among various teams across the globe. There is a limit to the amount of improvement in productivity brought about by adding more people to a project team, due to the increase in coordination costs. Developer teams generally prefer to work alone, and often go up to 2-3 months without knowing what is happening in the main branch [6]. This makes the process of integration very messy, as many conflicts pop up. Also, once a developer is done with one task, he/she generally prefers to move on to the next assigned task, rather than document what he/she did on the task which was just finished. This makes it hard for the developer's future replacement to understand parts of the code, in case that code needs to be reviewed or backtracked (perhaps in order to resolve a conflict).

Another important issue that the Communication Challenge gives rise to is that of conflicting priorities. It often happens that a high priority development component of one team is dependent on a component which is of a low priority of another team, and possibly in its backlog. Knowledge sharing between different teams is extremely crucial in order to be aware of and communicate technical dependencies.

2.3 Changes in Requirements and Time-Plan

Conflicts that pop up, depending on their severity, often cause the developer teams to deviate from their initially laid out plan and re-schedule some of their activities in order to accommodate the unforeseen activities in the timeline, that are now required to be carried out in order to resolve the conflict. It also happens many a time, that due to an unplanned technical dependency of a component on another component in the team's backlog, the team needs to perform that backlogged task earlier than initially planned in order to resolve the dependency, or even a task that isn't in the backlog, and this upsets the time-plan [6].

2.4 The Domino Effect

As one can understand by reading the previous challenges stated, it is evident that the likelihood of challenges is largely affected by the presence and occurrence of other challenges. They create a vicious cycle, aggravating the existing challenges, as well as giving rise to new challenges, if not dealt with properly. Communication challenges lead to a lack of knowledge about technical dependencies, which creates problems during the integration of different components. The conflicts that arise may then lead to a change in requirements and/or re-prioritization of tasks, which then invariably results in the upset of the organization's intended plans and schedules. This illustrates the domino effect of challenges, which majorly hampers the levels of productivity, and causes a delay in deploying the Software Release.

3. UNDERSTANDING CONFLICTS

In the context of software development, conflicts occur when two or more parties try to access and modify or make changes to a common file or resource. If a change in the order of serialization of the different changes results in a change of output then it indicates a conflict [8]. In collaborative software development, developers access the same code asynchronously, and this gives rise to a lot of inconsistencies, due to these conflicting concurrent changes by the different developers [7].

In the context of software releases, conflicts happen when two or more teams or bodies have possibly conflicting tasks planned for a common configuration item. For example, a few of the changes that are to be executed during a release may involve ramping down of a certain configuration item, and a few other changes require the same configuration item to be functional during the same time period. This results in a conflict between the two sets of changes that are planned for that release. In both cases, it essentially results from a lack of coordination and communication between developers and teams, in the contexts of software development and software release management respectively.

Conflict detection is about determining if two sets of simultaneous changes are isolated from each other, or if they interfere with each other. Most conflicts need to be resolved by manual merge operations, which is why it's important to detect conflicts as fine grained as possible. It is also important to not lose any information or violate the integrity of the model [8].

4. APPROACHES EXPLORED

4.1 Model Driven Development

Models are layouts or abstractions that are used to conceptualize and/or represent software systems. They serve as the blue prints used to develop software. They are helpful in encapsulating the relationships between different modules as well. They also serve as interfaces through which developers can interact with and interpret the model. Thus, one of the main outcomes of this approach is to spare the users and developers from the complexities of the underlying functioning of the system [3]. Thus, having accurate models would be helpful in understanding the relationships between modules, and highlighting possible dependencies between them, which is better than having complications due to the dependencies pop up during run time.

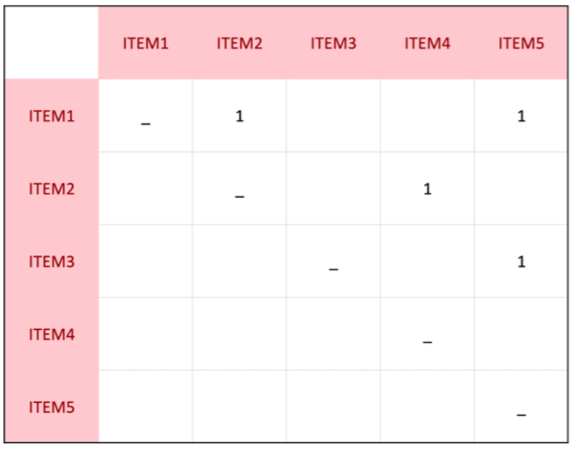
4.2 Autonomic Computing

Automating the detection of (at least the commonly faced) problems and triggering self-healing and self-optimization mechanisms could greatly reduce the cost of human resources, and thus result in increased efficiency. There are various degrees of autonomic computing depending on the complexities of systems as well as the available opportunities for automation, ranging from completely manual computing, to partially autonomic computing to complete autonomic computing. An autonomic computing system should help in dealing with RFCs (Request for Change). It should maintain a library containing descriptions of common RFC triggers, and refer to this library when exceptions or conflicts are thrown, and create a suitable RFC. It would also be responsible for allotting a priority to the detected RFC, and judge whether it is a normal RFC or not. By testing out the RFC and making an impact and requirements assessment, it helps in the planning and execution of the RFC, as well as any related decision making [4].

4.3 Design Structure and Domain Mapping Matrices

Quantifying the quality of the architecture with some measurable criteria can provide many insights which could be valuable for guiding engineering decisions, as well as evaluation of the development work carried out. Reference [5] explores a few such metrics that help in analyzing the quality of architectures, and also in bringing out other insightful trends. A Design Structure Matrix maps the dependencies between items in a particular domain. It is a square matrix as shown in Figure 1, with the list of items on both axes. For example, say there is an entry '1' in a particular cell, it indicates the presence of a dependency between the items in the row and column corresponding to that cell.

Depending on the nature of the directions of the dependencies, the resulting matrix could either be symmetric (in case of bidirectional dependency) or asymmetric (in case of unidirectional dependency). When dependencies occur across different domains, they are mapped using Domain Mapping Matrices. If two domains are largely interdependent, this relationship can be identified by clusters of entries in the Domain Mapping matrix. Design Structure Matrices and Domain Mapping Matrices are combined in order to study the intra-domain and inter-domain dependencies in the context of two related domains.



	ITEM1	ITEM2	ITEM3	ITEM4	ITEM5
ITEM1	_	1			1
ITEM2		_		1	
ITEM3			_		1
ITEM4				_	
ITEM5					_

Fig-1: Design Structure Matrix

Representing the dependencies helps in assessing the total impact on the system when certain changes are made, as some changes have a cascading effect due the dependencies between items. The cost of Propagation, denoted by P_c, is a metric which helps us quantify the potential extent of the impact of a change made to a randomly selected item. It is calculated as the density of the mapping matrix [5]

$$P_c = \frac{\sum_i M^i}{n^2}$$

Where M^i are the cells of the matrix having an entry, and n is the size of the matrix.

The total cost T_n of a Release Event n, is the summation of two components: the Implementation cost I_n, which is the cost of adding new elements to the system in the release, and the Rework cost R_n, which refers to the cost of modifying the existing architecture in order to accommodate the new components which are being added to it during the release.

$$T_n = I_n + R_n$$

This metric is used to estimate the costs which are expected to be incurred in a Release Event n.

5. CONCLUSIONS

As systems and their development keep scaling up, the management and unification of the different teams and components involved pose a huge difficulty due to the friction caused by conflicts in processes and decisions, and the roadblocks that dependencies cause. It is thus critical to develop systems and practices that promote communication and knowledge sharing, and help keep track of development and release related activities, in order to help keep all the concerned teams on the same page.

It is also important to develop analytical tools that represent dependencies in models, and help identify trends as well as assess the impact of potential conflicts.

ACKNOWLEDGEMENT

I would like to personally thank my guide Ms. Pratiba D and the Department of Computer Science and Engineering of RV College of Engineering for guiding me through the process of writing this paper, with their valuable suggestions and insights.

REFERENCES

[1] M. Ramakrishnan, "Software Release Management", Bell Labs Technical Journal 9(1), 205-210 (2004)

[2] Par Carlshamre and Bjorn Regnel, "Requirements Lifecycle Management and Release Planning in Market-Driven Requirements Engineering Process ", 2000 IEEE

[3] R. France, B. Rumpe, "Model-Driven Development of Complex Software: A Research Roadmap", Future of Software Engineering (FOSE'07), 2007 IEEE

[4] S. Chen, H. Jiang, "Integrating Change and Release Management towards Autonomic Computing", 2008 IEEE

[5] N. Brown, R. L. Nord, I. Ozkaya, M. Pais, "Analysis and Management of Architectural Dependencies in Iterative Release Planning", Proceedings of the Ninth Working IEEE/IFIP Conference on Software Architecture (WICSA) 2011

[6] N. Sekitoleko, F. Evbota, E. Knauss, A. Sandberg, M. Chaudron, H. H. Olsson, "Technical Dependency Challenges in Large-Scale Agile Software Development", G. Cantone and M. Marchesi (Eds.): XP 2014, LNBIP 179, pp. 46-61, 2014

[7] P. Dewan, R. Hegde, "Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development", ECSCW'07: Proceedings of the Tenth European Conference on Computer Supported Cooperative Work, 24-28 September 2007

[8] M. Koegel, J. Helming, S. Seyboth, "Operation-based conflict detection and resolution", CVSM'09, May17, 2009