A Systematic and Lightweight Method to Identify Dependencies Between User Stories

Arturo Gomez[1], Gema Rueda[1] and Pedro P. Alarcón[2]

[1] Blekinge Institute of Technology
Sweden
{argo09,geru09}@student.bth.se
[2] E.U. Informática
Technical University of Madrid (UPM), Madrid, Spain
pedrop.alarcon@eui.upm.es

Abstract. The order in which user stories are implemented can have a significant influence on the overall development cost. The total cost of developing a system is non commutative because of dependencies between user stories. This paper presents a systematic and lightweight method to identify dependencies between user stories, aiding in the reduction of their impact on the overall project cost. Initial architecture models of the software product are suggested to identify dependencies. Using the method proposed does not add extra load to the project and reinforces the value of the architecture, facilitates the planning and improves the response to changes.

Key words: User Stories Dependencies, Agile Development, Dependencies Identification Method, Non Commutative Implementation Cost

1 Introduction

The elements that comprise the system under construction interact with each other, establishing dependencies among them [1]. In Figure 1, element A requires element B, generating a dependency between them. Such dependencies are naturally inherited by the user stories (US_i cannot be implemented until US_j is implemented). Therefore, the natural dependencies between User Stories (US from now on) should be accepted as inevitable. In fact, only a fifth of the requirements can be considered with no dependencies [2]. The existence of dependencies between USs makes necessary to have some implemented before others [2] [3] [1] [4]. If the order of user stories implementation does not take into account these dependencies it may have a large number of preventable refactoring, increasing the total cost of the project needlessly. Identifying beforehand the dependencies increases the ability to effectively deal with changes. Therefore light systematic mechanisms, as shown in this paper, are needed to help identify dependencies between USs.

The rest of the paper is structured as follows. The second section describes the problem of dependencies. The third section defines the concept of dependency

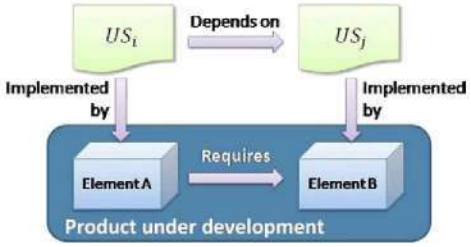

Fig. 1. Inherited dependencies by user stories

between user stories. The fourth section describes the method to identify dependencies. The fifth section presents an example applying the method proposed. The sixth section presents related work. Finally, the conclusions are listed.

2 Problem Description

The existence of dependencies between USs hampers planning [5] [4]. Not considering them increases the chances of not complying with the release plans [6]. Therefore, the sequencing of USs is seen as a challenge [7]. Depending on the established implementation order of USs the number of refactoring may increase. For example, suppose that at time t, once the user story US_i has been implemented, there is a database (DB) in production with the entity T_1 and primary key k_1. At time t+1, after implementing US_j, the data model shown in Figure 2 is obtained, in which the primary key of the entity T_2 is k_2. Given the cardinality, the primary key attributes from T_2 become part of the table generated for entity T_1. This will require a refactoring of the DB and all components that access T_1 and an update of all rows of table T_1. If US_j had been implemented before US_i there would be no need to refactor, so the refactoring cost would be zero. Hence, due to the existing dependencies, the total cost of developing a system

Fig. 2. DB in time = t+1

depends on the order in which the USs are implemented. Therefore, the total cost of developing a system is non commutative. Generalizing, if US_j depends on US_i and being C the cost function of implementing a user story in a given time t, considering RC as the cost of carrying out a determined refactoring j, then: $C(US_j)_t + C(US_i)_{t+1} = C(US_i)_t + C(US_j)_{t+1} + RC_j$. Note that refactoring can become a complex process with a very high cost [8], which is directly proportional to the number of implemented user stories [9].

3 User Stories Dependency Concept

This section defines the concepts: Dependency on key (Definition 1) and dependency on service (Definition 2).

Definition 1. Considering an agile project P, and E as the data model of P. Given that US_i and US_j are user stories from P that respectively require data represented in the entities E_i and E_j belonging to E. If after E is transformed into the target model (usually relational model) the data structure generated for the entity E_i adds the primary key attributes of the entity E_j, then US_i has a dependency on key with US_j, and it is expressed as: $US_j \rightarrow US_i$. In Figure 3, the following dependencies on key are found: $K=\{US_2 \rightarrow US_1\}$.

Fig. 3. Example of simplified conceptual data and component diagram

Definition 2. Considering an agile project P which has been represented by a component diagram C. Given that US_i and US_j are user stories from P, which are implemented respectively in the components C_i and C_j included in C. The user history US_i has a dependency on service with respect to US_j, if and only if C_j implements at least one service used by US_i in C_i, expressed as: $US_j \rightarrow US_i$. In Figure 3, the following dependencies on service are found: $S=\{US_2 \rightarrow US_1\}$.

Based on the above definitions, the complete set of depencencies is defined as: $D=\{K \cup S\}$. Note that D can vary because of changes in user stories.

4 User Stories Dependencies Identification Method

The dependencies cannot be clearly inferred from the definition of USs. Building an initial architecture (data and component models) helps to identify them. Both models are transversal to the USs, see architectural models boxes at Figure 4. The evaluation of the interaction of each user story with both models allows the identification of possible dependencies. The proposed method identifies USs dependencies. Its duration depends on the size of the project and the presence of the whole team is recommended during its application to gain a project overview. It is lightweight in the sense that it does not add load to the project, since the activities or products needed are carried out in initial stages. If the USs or models change, the identification method should be executed before starting the next iteration (see Figure 4).

To identify dependencies between USs: First, a quick study of user stories defined so far is suggested, generating a simplified data model (without attributes). The use of the entity-relationship model is recommended since it helps to generate an overall view of the system. It is usualy generated in software projects and therefore it does not add additional load. Notice that this diagram is not an

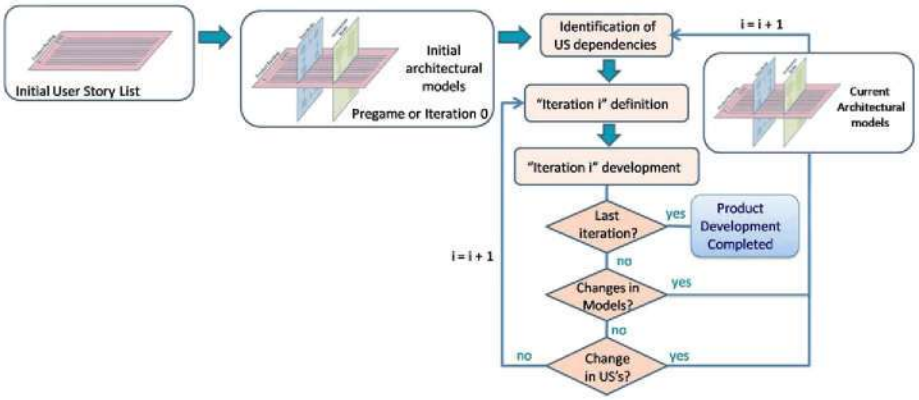

Fig. 4. Proposed method in iterative life cycle

objective in itself. Its purpose is to identify the elements from the data model that each user story requires to be implemented, writing its identifier next to the data element required. For example, brackets can be used as shown in Figure 3. Second, establish the set of dependencies on key from the diagram, according to Section 3. To do so, for example, the transformation rules from an entity-relationship model to relational model can be used. Thus, given two elements A and B of a model M, if element A migrates the primary key attributes to element B, then the user stories related to B will have dependency on key of the user stories related to A.

To identify dependencies on service it is proposed: First, use a simplified component model which will represent the list of user stories identified so far. This diagram will include the components identified as well as the service relationship between them. It has a high level of abstraction that allows to easily identify the dependencies on service. Its creation provides a global perspective of the system to the team, which is important for understanding the dependencies. As in the previous case, this diagram is not a goal in itself. It can be replaced by any other that allows identification of such dependencies. The USs involved in the implementation of each component should be written within brackets (see Figure 3). Second, identify the set of dependencies on service from the diagram, according to Section 3. Thus, given two elements A and B of a model M, if the element A implements a service required by B, then the user stories related to B will depend on the user stories related to A.

The mechanism to register dependencies is to record them using a directed graph like the one shown in Figure 5. Initially, all the USs are represented as disconnected vertices. As soon as $US_j \rightarrow US_i$ is identified, an edge pointing US_i is drawn between vertices US_j and US_i. This representation informs quickly about the dependencies among USs. Additionally, it helps to quickly identify dependency chains between USs. The graph generated can be used as basis to support planning or as an input for well known algorithms [1] [3] to generate an implementation sequence that reduces the impact of dependencies.

When interpreting the results, a vertex without incoming edges means that this user story has no dependencies. If a vertex (US_i) has incoming edges but these edges come from vertices representing USs already developed, it is also considered that US_i has no dependencies. From the technical perspective, a user story without dependencies can be implemented at any time or assigned at any release and business value would be the main factor when prioritizing and planning it. When planning, the development team must be aware that if a user story (US_i) is developed and it depends on other USs not developed yet, there could be additional costs associated with refactoring and other technical risks. The customer should be warned with this information before prioritizing the user story. When a user story changes or a new one is introduced, the directed graph must be checked to identify the USs that depend on the changed or new user story. The architectural elements associated to these dependent user stories are more likely to be impacted by this change. Therefore, the set of architectural elements that are likely to change is reduced, facilitating the response to change.

5 Example of Use

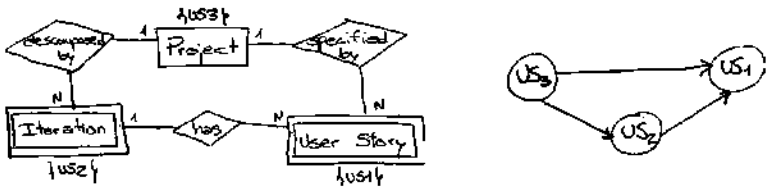

Fig. 5. Original scanned data model from selected US

This section focuses on a subset of USs extracted from a real project in which the authors of this paper participated. This project included the development of a software tool called Agile Management Tool (AMT). The subset of user stories selected from AMT project is: US_1 (Create User Stories); US_2 (Create Iterations); US_3 (Create Projects). Due to the paper's size restrictions this section focuses only on the data model (see Figure 5). Following the identification method proposed, references to USs related to each model element have been included. Notice that when the simplified data model is transformed into relational tables, the primary key attributes from the entity *Project* (related to US_3) will migrate into the entity *User Story* (related to US_1), which implies that US_1 depends on US_3, therefore an edge from US_3 vertex pointing to US_1 vertex must be drawn. This way the team will continue identifying dependencies, generating at the end a graph like the one showed on Figure 5. Based on it, the dependency set is: $D=\{US_3 \rightarrow US_1;\ US_3 \rightarrow US_2;\ US_2 \rightarrow US_1\}$. Then, from the technical point of view, since every user story depends on US_3, the recommendation to the customer would be implementing US_3 first. Otherwise, the cost of refactoring should be added to the cost of developing US_1, US_2 and US_3.

6 Related Work

Some well known methods consider dependencies such as IFM [1] and Evolve [4][3]. Nevertheless none of them provide a systematic mechanism for identifying dependencies between user stories as the method proposed in this paper. In [2] is proposed a method to identify dependencies but it relies on pairwise assessment among the requirements. This is applicable for a small number of requirements but requires too much effort facing a large number of requirements. Mike Cohn states that if two user stories are dependent they must merge [5]. However, in practice it has been seen that large user stories that cannot be completed in one iteration, hinder the feeling of progress and therefore team motivation [7].

7 Conclusions

The implementation cost is non commutative due to the existence of dependencies between user stories. If this fact is obviated, it could generate overrun in the development of a product. This overrun comes from unnecessary refactoring that could have been avoided with a different implementation order. Two definitions of dependencies have been provided: dependency on key and dependency on service. This paper contributes with a very lightweight method that identifies dependencies between user stories, helping the planning and reducing the technical risks of the project, while reinforcing the architectural value as a lateral effect. Furthermore, if this method is applied at the beginning of the project, it helps to create a common perspective of the system. This method has been designed to fit in an agile environment, following the agile values and principles.

Refereuces

1. Denne, M., Cleland-Huang, J.: The incremental funding method: Data-driven software development. IEEE Software **21**(3) (2004) 39–47
2. Carlshamre, P., Sandah, K., et al: An industrial snrvey of requirements interdependencies in software product release planning. In: RE'01. (2001) 84–91
3. Greer, D., Rnhe, G.: Software release planning: an evolntionary and iterative approach. Information and Software Technology **46** (2004) 243–253
4. Logue, K., McDaid, K.: Handling uncertainty in agile requirement prioritization and scheduling using statistical simulation. In: Agile 2008, IEEE CS (2008) 73–82
5. Cohn, M.: User Stories Applied: For Agile Software Development (The Addison-Wesley Signature Series). Addison-Wesley Professional (March 2004)
6. Babinet, E., Ramanathan, R.: Dependency management in a large agile environment. AGILE Conference (2008) 401–406
7. Ton, H.: A strategy for balancing business value and story size. In: Proceedings of the AGILE 2007, Washington, DC, USA, IEEE Computer Society (2007) 279–284
8. Ambler, S.W., Sadalage, P.J.: Refactoring Databases: Evolutionary Database Design. Addison-Wesley Professional (March 2006)
9. Boehm, B., Turner, R.: Balancing Agility and Discipline: A Gnide for the Perplexed. Addison-Wesley Professional (Augnst 2003)