Date 03/11/2015

Authors:
Essa Imhmed, Ruth Torres Castillo, Vijaya K Pandey, Hitesh N. Sharma, Sultan Alharthi

Software Architecture Document

Jupiter

A User story Application

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

This document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made on the system.

1.2 Scope

This Software Architecture Document provides an architectural overview of the user story management application (Jupiter). Jupiter is being developed by the project team to create and manage user stories, which create a coherent whole "requirements document" from them, and to manage their connections to actual code.

1.3 Definitions, Acronyms and Abbreviations

- Java - Programming Language
- SAD - Software Architecture Document
- UML – Unified Modeling Language
- Jupiter - The user story management application name
- User - This is any user who uses the application

2. Architectural Representation

This document presents the architecture as a series of views; use case view, logical view and process view. There is no separate implementation or deployment view described in this document.

3. Architectural Constraints

There are some key requirements and system constraints that have a significant bearing on the architecture. They are:

1. The System is intended to be used for a single user without the use of heavy databases but data stored through the system should be consistent and reliable.
2. This system is developed as a small class project, we are not considering performance, stress testing, size or growth.
3. The system must maintain data locally in the computer it is installed in.
4. The system should be able to generate requirements documents in a consistent standard format.

4. Rationale of the Architecture

4.1. Standalone Desktop Application

The main rationale of using the standalone architecture is the need for having a lightweight application that can run from a user's computer without the need of any web browser or without needing to connect to any remote systems. As the application must maintain data locally in the machine that it is installed in, a desktop based standalone application is a good option to have. The system doesn't require providing access to multiple users, so a centralized server or database is not required. However, management of data generated by the application will be done through flat Comma Separated files (CSV files). Moreover, upgrading and scalability becomes very easy as the user just needs to update the application. Further, for the user to be able to link his code to the user stories, a desktop application can efficiently access local file system to manage linkages between them.

4.2 Data Storage

The main rationale of using flat files for storing data lies is that fact that the data storage requirement of the system is not considerably large and doesn't require the management of complex data relations. The data requirement of the system can be fulfilled by using a flat file without having to use heavy database servers. Moreover, the system is developed keeping in mind that user doesn't need to install any database servers for the application.

4.3 Significant Requirements That Drove the Rationale

The architecturally significant requirements that drove the rationale are:

1. *Adding user stories*: The rationale behind this requirement is to give the user the ability to create and edit user stories so that he can efficiently capture client's requirements and store them locally for developing his system based on those requirements.

2. *Linking user stories to actual code*: The rationale behind this requirement is to give an easy and effective way to link a user story to the program code. It gives the user a view that allows him to see linkages between his user stories and the actual code that was created to implement that user story. This way a user can know which files are most linked to a particular user story or which file in the system is most significant in terms of user story coverage. Further, as the code is generally located in the client computer, desktop application can easily access code which is stored in local files in the machine.

3. *Generate requirements documents*: The rationale behind this requirement is to create a coherent whole "requirements document" from the user stories to be able to track progress of user requirements that are completed or ongoing.

5. Use-Case View

The Use Case View describes the set of scenarios and/or use cases that represent some significant, central functionality of the system. Figure 1 shows the use cases of our system, Jupiter.

5.1 Architecturally-Significant Use Cases

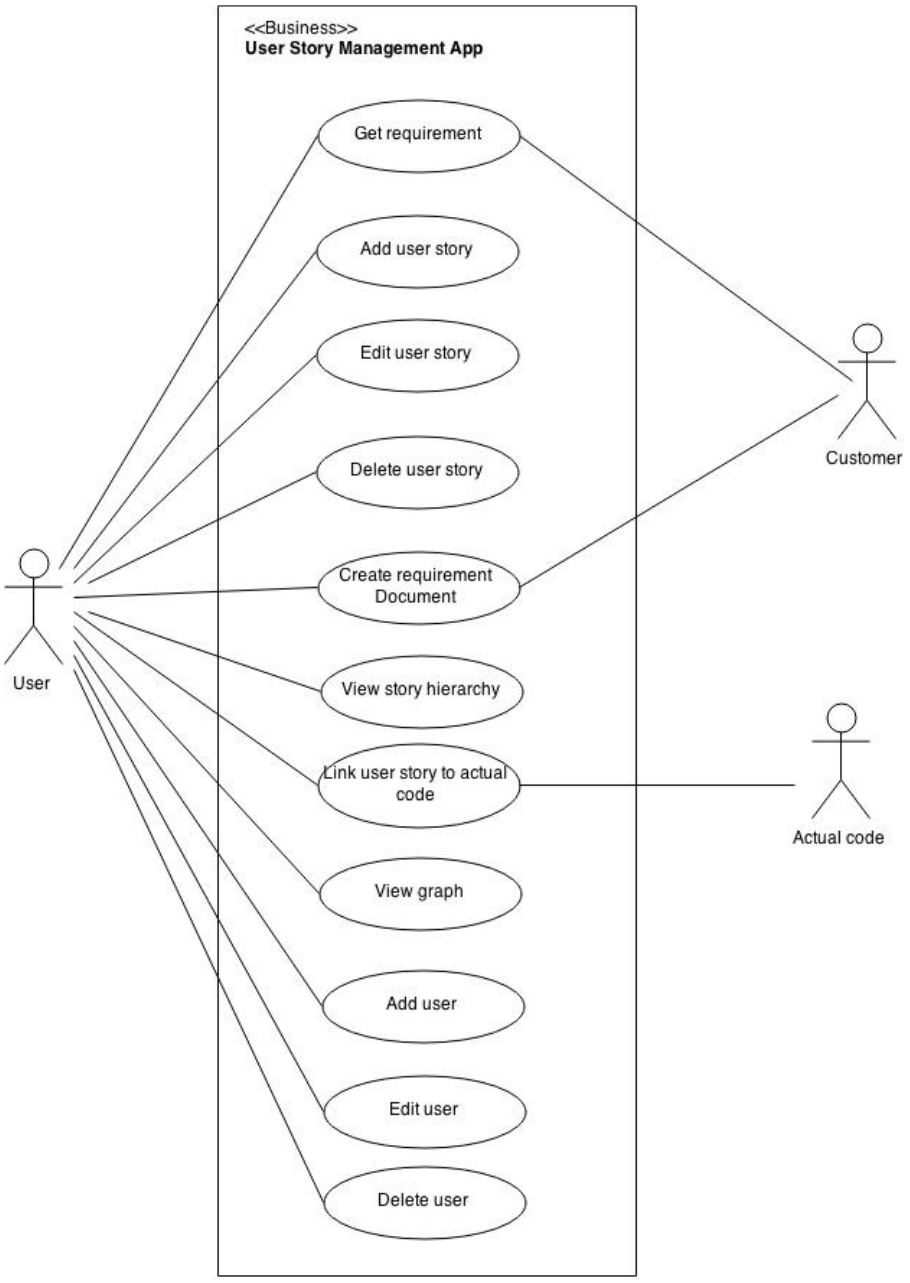

Figure1: Architecturally Significant Use Case diagram

5.1.1 Get requirement

Brief Description: This use case allows the user to get a requirement from customer.

5.1.2 Add user story

Brief Description: This use case allows the user to create new user story.

5.1.3 Edit user story

Brief Description: When the user creates a new user story, it is stored in a file. This use case allows the user to edit the existing user story.

5.1.4 Delete user story

Brief Description: This use case allows the user to delete the user story.

5.1.5 Create requirement document

Brief Description: This use case allows the user to generate requirement documents from the user stories.

5.1.6 View story hierarchy

Brief Description: This use case allows the user view a hierarchical structure of the user stories.

5.1.7 Link user story to actual code

Brief Description: This use case allows the user to link user story to actual code. This includes adding and deleting tags.

5.1.8 View graph

Brief Description: This use case allows the user to view and print progress report of user stories.

5.1.9 Add user

Brief Description: This use case allows the user to create a new user.

5.1.10 Edit user

Brief Description: This use case allows the user to modify the details of an existing user.

5.1.11 Delete user

Brief Description: This use case allows the user to delete an existing user.

6. Conceptual Architecture

In this case the Layered architecture is the one that best defines Jupiter system. Because as this architecture works each layer interacts with the layer below it and delivers services for the layer immediately above. The upper layers can concentrate on implementing functions needed by its users and relay on lower layers to handle the more basic system needs. For the Jupiter system the 3 layered that will be define are:

Components of 3-Layer Architecture

Presentation or Interphase

Business
Data

Tier indicates a physical separation of components
Layer indicates a logical separation of components with the help of namespaces and classes

And because these components are physically separated we are going to call them

Interphase Tier: as the name stands for, is the UI with which the user will interact
Business Tier: This works as a bridge between the interphase and the data Tiers. It collects raw data from the Interphase Tier, checks for validations, converts them to a standard format and finally sends them to the Data Tier. Similarly it collects data from the Data Tier, purifies it and sends it to the Interphase Tier for display.
Data Tier: Where actual data is stored.

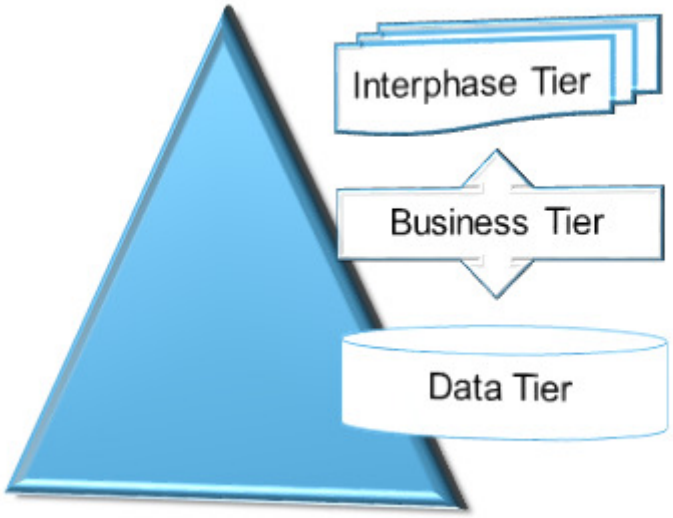

Figure 2: the conceptual architecture diagram

Figure 2 describes the Jupiter's architecture. In this diagram, the Interphase Tier, responsible for displaying information and handling interaction is using services from the Business Tier that is re-

sponsible of performing actual system functions. The business Tier is using services on its own from the data layer who is responsible with storing and retrieving data.

6.1 Informal Component Specification

The tables represented below show the Class Responsibility Collaboration (CRC) Card used for informal component specification.

CRC-R.1

Component Name	User
Responsibilities	Adds requirements in form of user stories
Collaborators	User Story
Rationale	We need an object that adds user stories to the system and for this there should be a user component
Issues and Notes	

CRC-R.2

Component Name	User Story
Responsibilities	Knows user requirements in the form of a user story, knows user role, priority, duration, reason, requirement, requirement type, title.
Collaborators	User, Requirement Document, Code Linker
Rationale	User story should have complete information about particular user requirements. It should contain time it takes to finish a user story so that it is easy for planning and estimation, should give a reason why a user story is needed. It should also have a priority so that the user story can be prioritized based on this.
Issues and Notes	

CRC-R.3

Component Name	Requirement Document
Responsibilities	Get generated from User stories
Collaborators	User Story

Rationale	A requirement document is generated because the user requires a requirement document based on user stories in the system, so this is like a mapping from user stories to requirements.
Issues and Notes	

CRC-R.4

Component Name	Code Linker
Responsibilities	Links user story to the code in a file
Collaborators	User Story
Rationale	It is a common scenario when a user wants to make changes to code based on any user story and is having trouble finding the code related to the user story. So, if the user story is linked to the code then user can easily find particular codes to change and will save a lot of time.
Issues and Notes	

7. Logical Architecture

7.1 Structure View

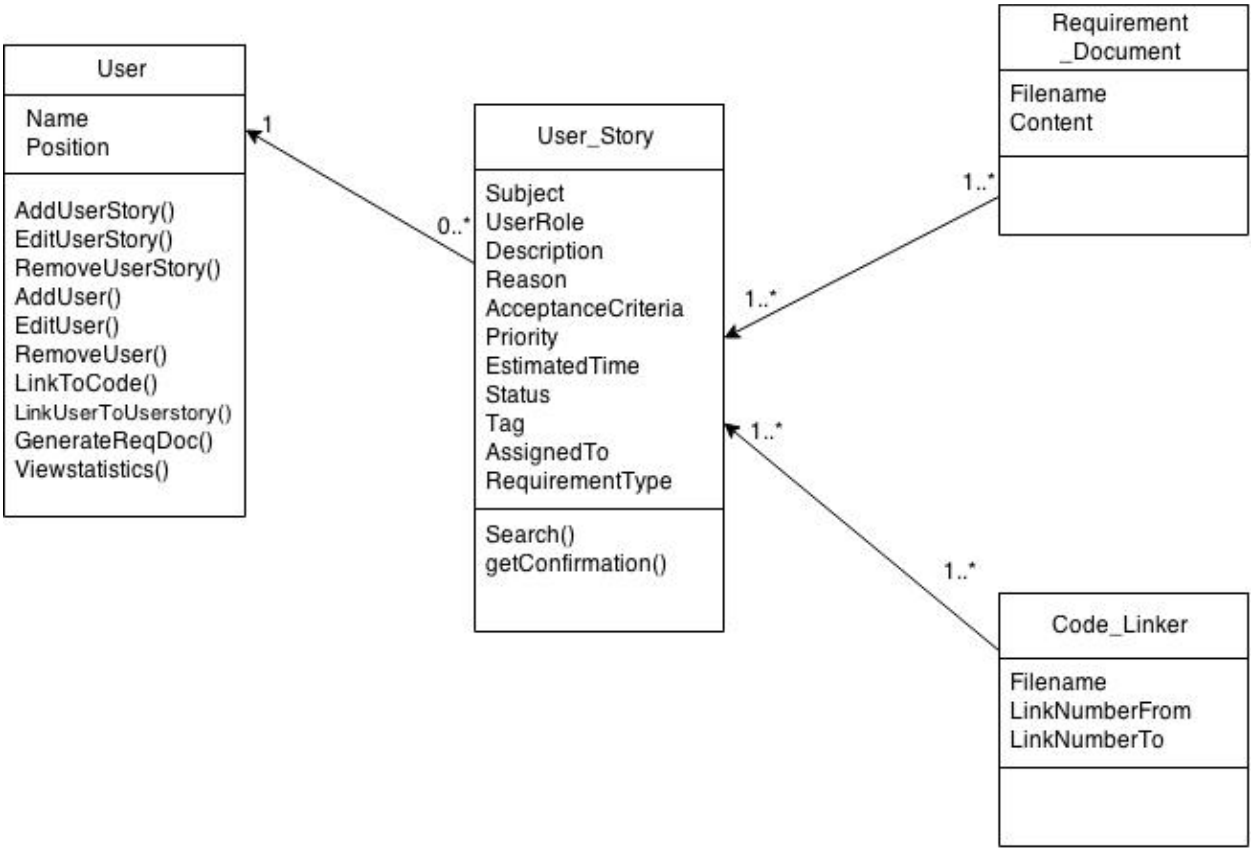

Figure 3: UML Class diagram of Jupiter's structure elements

Figure 1 shows a UML class diagram contains some structure elements that represents Jupiter's main attributes and their functions: User, User Story, Requirement Document, and Code linker. While the User Story class is shown as depending on User class, the other two classes, Requirement Documents and Code Linker, are shown as depending on User Story class.

7.2 Behavior View

As well as defining structural elements, architecture defines the interactions between these structural elements. It is these interactions that provide the desired system behavior. Figures 4, 5, 6, for example, show UML sequence diagrams showing a number of interactions that, together, allow the system to support the creation and managing of a user stories.

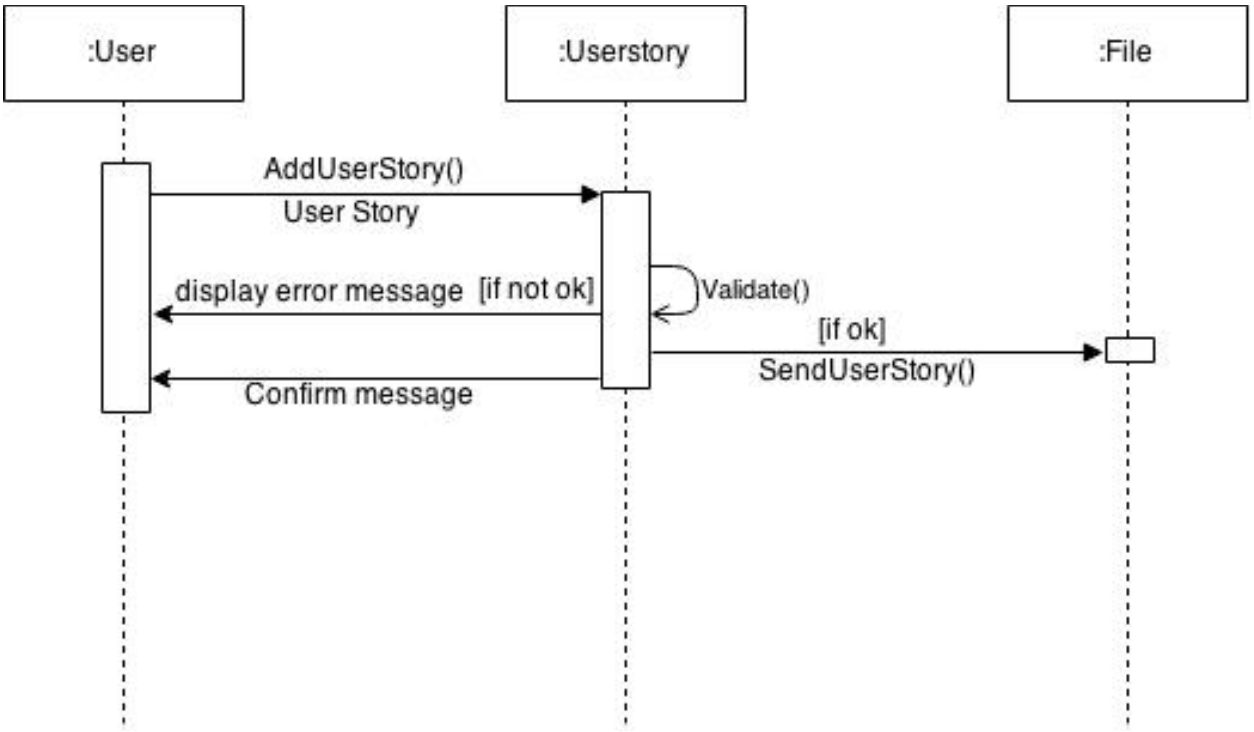

Figure 4: UML Sequence diagram of adding user story

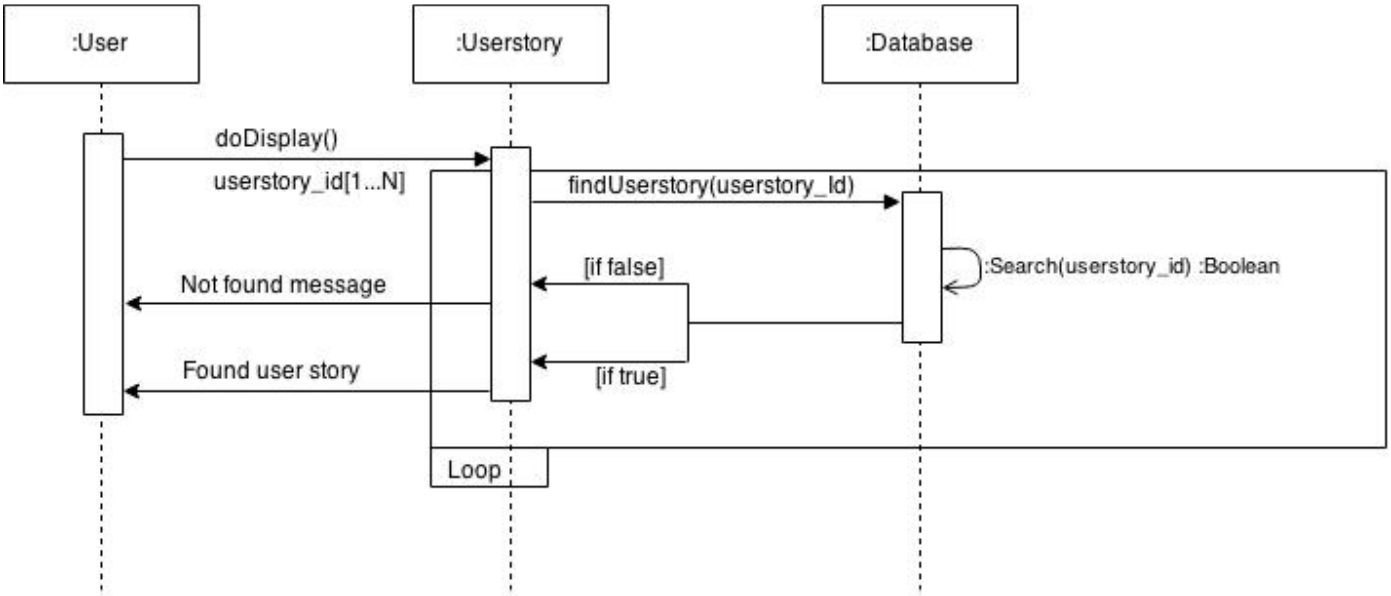

Figure 5: UML Sequence diagram of displaying user story

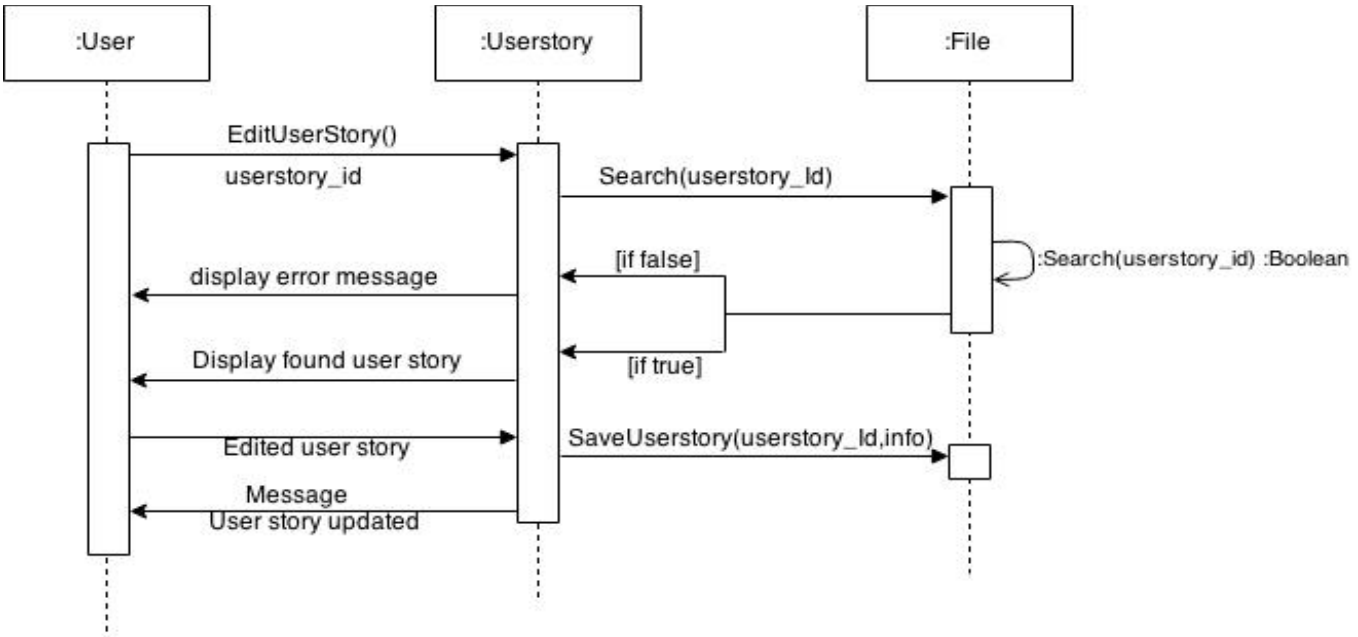

Figure 6: UML Sequence diagram of editing user story

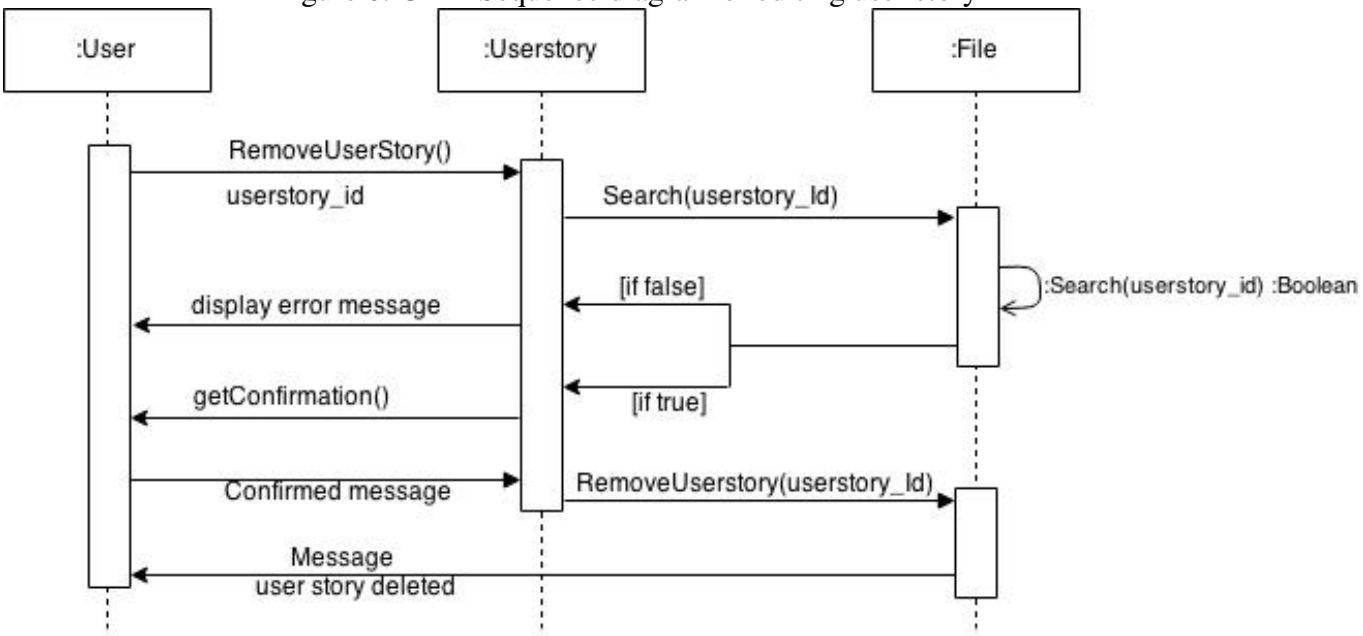

Figure 7: UML Sequence diagram of deleting user story

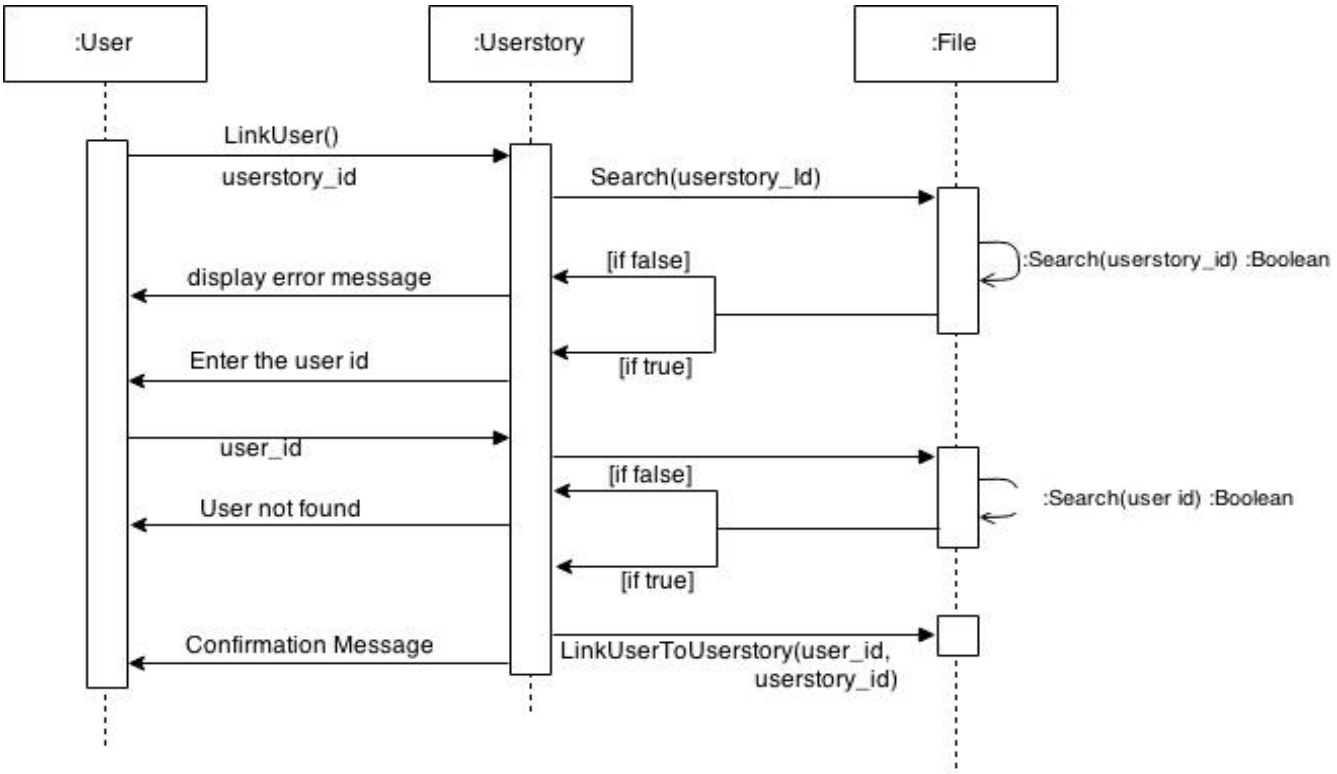

Figure 8: UML Sequence diagram of linking user story to actual code

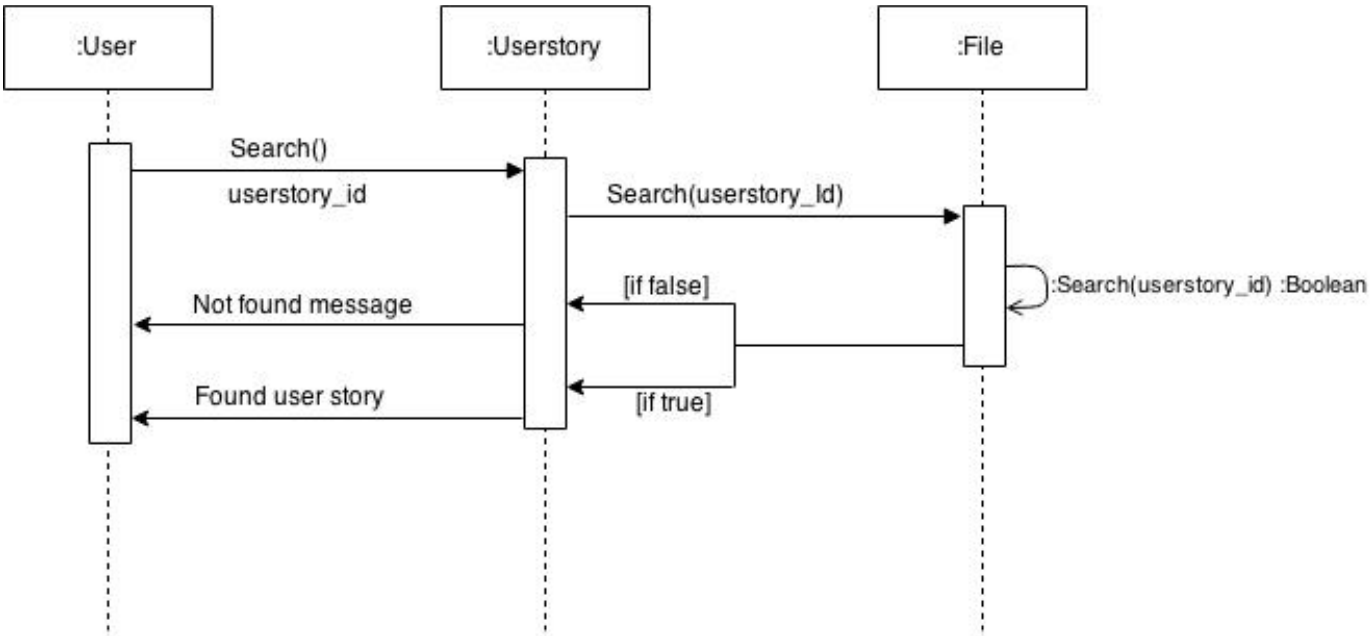

Figure 9: UML Sequence diagram of searching user story

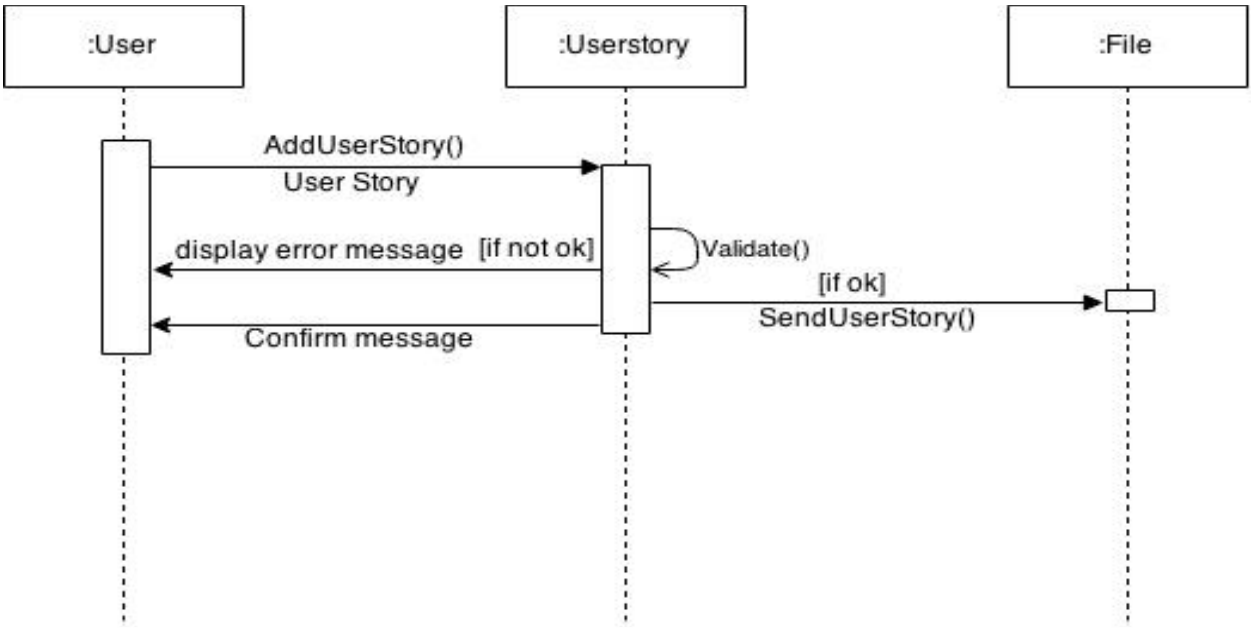

Figure 10: UML Sequence diagram of generating requirement document

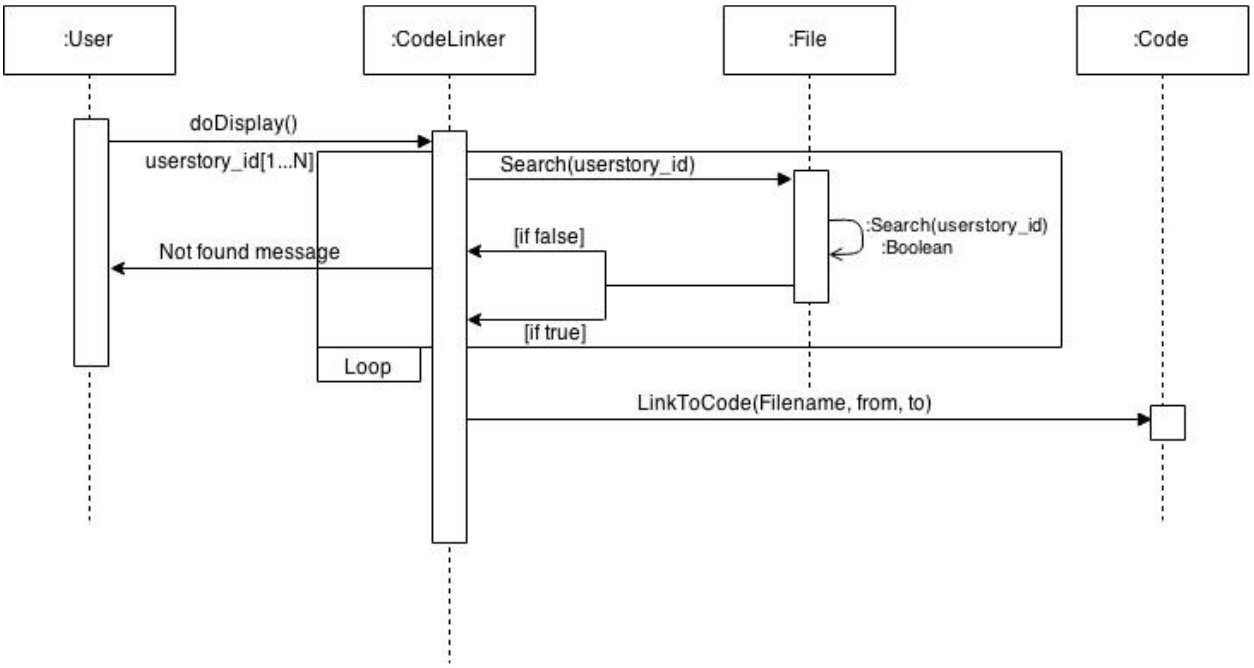

Figure 11: UML Sequence diagram of linking user stories to actual code

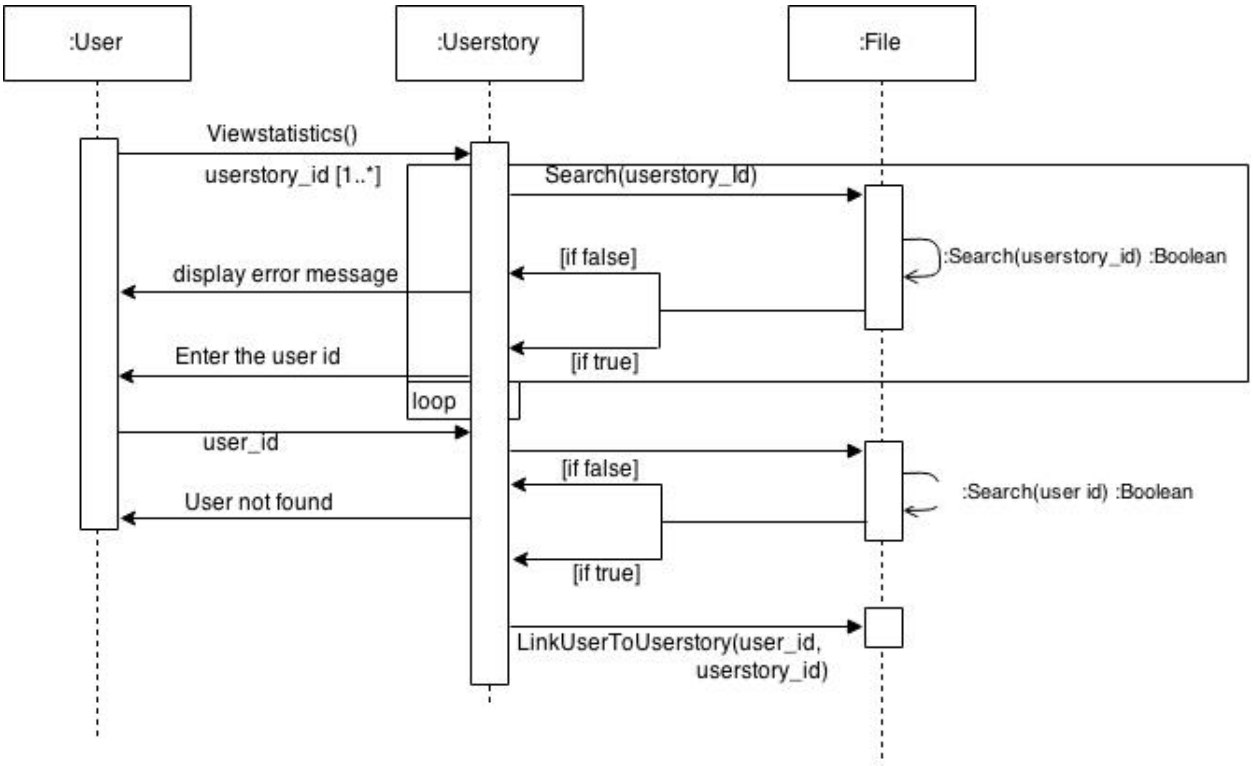

Figure 12: UML Sequence diagram of viewing statistics

8. Architecture Scope and Impact

Because our system will be based on managing user stories, high impact decisions as the creation, generation of requirements and linking to the code are part of the systemic decisions that we should take into consideration.

	Low impact	**High impact (High priority, Important to business)**
Systemic (broad scope)	• Create system statistics based on application's data. • Create a new type of criteria for classification for user stories • Add users to the system	• Add user stories
Local	• Display User Stories	• Link user story to the code • Generate requirement document

9. Size and Performance

Volumes:

- Estimated user story addition: 5 a day (Assuming a project team has 5 members and work on one user story a day)

Performance:

- Time to process to add user story: less than 5 seconds approximate

10. Quality

- The user interface of Jupiter shall be designed for ease-of-use and shall be appropriate for a computer-literate user community with no additional training on the System.

11. References

1. Eeles, Peter. "Capturing Architectural Requirements." Capturing Architectural Requirements. IBM, n.d. Web. 12 Mar. 2015.

2. Clements, Paul, and Len Bass. Relating Business Goals to Architecturally Significant Requirements for Software Systems. Ft. Belvoir: Defense Technical Information Center, 2010. Web.

3. Malan, Ruth, and Dana Bredemeyer. "Software Architecture: Central Concerns, Key Decisions." *Software Architecture: Central Concerns, Key Decisions, 2005.*

4. Malan, Ruth, and Dana Bredemeyer. "Conceptual Architecture Action Guide." *Concep tual Architecture Action Guide, 2005.*