Modeling Non-Functional Requirements in Cloud Hosted Application Software Engineering

Santoshi Devata, Aspen Olmsted
Department of Computer Science
College of Charleston, Charleston, SC 29401
e-mail: devatas@g.cofc.edu, olmsteda@cofc.edu

***Abstract*- Functional Requirements are the primary focus of software development projects for both end users and developers. The Non-Functional Requirements (NFR) are treated as a secondary class requirement, ignored until the end of the development cycle. They are often hidden, overshadowed and therefore, frequently neglected or forgotten. NFRs are sometimes difficult to deal with and are the most expensive in certain cases. NFRs become even more important with cloud architectures because the concurrent load and response latency are more vulnerable using public networks than they were on private networks. More wok is needed on mapping NFR models into software code. Developing a cloud based system with functional requirements only is not enough to build a good and useful software. NFRs should become an integral part of software development. In this paper, we focus on the modeling of NFRs and the transformations from UML models into the source code. Specifically, we choose three NFRS: response time, concurrency, user response time for a Theater Booking system.**

Keywords - Non-Functional Requirements; NFRs; Response time; Concurrency

I. INTRODUCTION

The software engineering process is intended to produce software with specific functionality that is delivered on time, within budget and satisfying customer's need. Bruegge and Dutoit [1] dedicate the first chapter of their text book to these outcomes and budget constraints. These demands mean that the software development is focused and driven by the functional requirements. The software market is changing every day increasing its demands for providing best quality software that not only implements all the desired functionality but also satisfies the NFRs. Including NFRs in the model, leads to a complete software capable of handling not just the requirements associated with the product but also provides the usability according to the current standards. NFRs (sometimes also referred to as software qualities) indicate how the system behaves and includes requirements dealing with system performance, operation, required resources and costs, verification, documentation, security, portability, and reliability. Thus, satisfying NFRs is critical to building good software systems and expedites the time-to-the-market process, since errors due to not properly dealing with NFR, which is usually time-consuming and complex, can be avoided. However, software engineers need to know whether the performance cost of the algorithms that deal with the various NFRs will violate the basic performance requirements or conflict among themselves.

Failing to address NFRs in the design phase can lead to a software product that may meet all the functional requirements but fail to be useful because it cannot be used. In the United State, the federal government contracted a 3rd party vendor to develop an application for individuals to register for health care coverage. The designers failed to specify the NFRs for concurrency, and the application could not be used because of the mistake. [2]

NFRs are sometimes not intuitive to the developers, so implementing and including them in the development cycles is challenging. There are different approaches to handling the NFRs, but the best way is to model them and implement for each case. Rahman and Ripon [3] describe a use case and the challenge of integrating the NFRs into the design models.

The organization of the paper is as follows: Section 2 describes the related work and the limitations of current methods. In Section 3 we give a motivating example and explain our proposal, present our algorithms and show how they are used in our research. We conclude and discuss future work in Section 4.

II. RELATED WORK

Functional requirements are defined and represented in many ways. These functional requirements are the basis of software development, but NFRs are the ones that supply the rules when implementing the code. Many authors have looked at NFRs and the problems of their inclusion in the design process. Pavlovski and Zou [4] define NFRs as specific behaviors and operational constraints, such as performance expectations and policy constraints. Though there are many discussions about the NFRs, they are not taken as seriously as they should be. Glinz [5] suggest the notion of splitting both functional and NFRs into a set of categories and make groups of

 ISBN: 978-1-61208-460-2

them so that they are inherently considered while developing the applications. Alexander [6] suggests looking at the language used to describe the requirements. Words ending in '-ility' are often the NFRs. Examples of these words are reliability and verifiability. All of their work focuses on identification of the NFRs. Our work builds on theirs by applying domain specific models using extensibility mechanisms built into standard modeling notations.

Ranabahu and Sheth [7] explore four different modeling semantics required when representing cloud application requirements. These include data, functional, non-functional and system. Their work focuses on functional and system requirements. There is a small overlap with our work, but only with nonfunctional requirements from the system perspective. They build on work done by Stuart [8] in his workshops where he defined semantic modeling languages to model cloud computing requirements in the three phases of the cloud application life cycle. The three phases are development, deployment, and management. Our work adds to the missing semantic category of NFRs.

In Ranabahu and Sheth [7] work they use Unified Markup Language (UML) [9] to model the functional requirements only. UML is a standardized notation for representing interactions, structure and process flow of software systems. UML consists of many different diagram types. Individual diagrams can be linked together to model different perspectives of the same part of a software system. We utilize UML to express the NFRs also.

Additional semantics for models can be added by the integration of the matching UML Activity and Class diagrams. UML provides an extensibility mechanism that allows a designer to add new semantics to a model. A stereotype is one of three types of extensibility mechanisms in the UML that allows a designer to extend the vocabulary of UML to represent new model elements [10]. Traditionally the semantics were consumed by the software developer and manually translated into the program code in a hard coded fashion.

Object Constraint Language (OCL) [11] is part of the official Object Management Group (OMG) standard for UML. An OCL constraint formulates restrictions for the semantics of the UML specification. An OCL constraint is always true if the data is consistent. Each OCL constraint is a declarative expression in the design model that states correctness. Expression of the constraint happens on the level of the class, and enforcement happens on the level of the object. OCL has operations to observe the system state but does not contain any operations to change the system state.

Our contribution in the domain of cloud computing software modeling is in the use of modeling standards such as UML and OCL with their extensibility mechanism of stereotypes to model the NFRs. We demonstrate these models can be transformed into application source code through the application of three application domain constraints.

III. RESEARCH PROPOSAL

Our contribution in this work is to look at application domain specific NFRs that are useful for cloud based application architectures. We model the NFRs using the extensibility mechanisms built into the standard modeling notations of UML and OCL to specify those NFRs. We then auto-generate code for NFRS using Java. We demonstrate the NFRs using a theater booking system example**.** A theater booking system is an online application used by theatres to sell their entrance tickets. Figure 1 shows an activity diagram for a theatre booking system where the NFRs are represented using UML stereotypes. We chose to focus on three NFRs for this study: response time, concurrency, and pick seat time to implement the theater booking system. For each NFR, we model in UML and OCL utilizing stereotypes to apply the additional required semantics. We then generate code from the model to enforce the NFRS. The code is generated for these NFRs for use in a cloud application that uses the threads on the server side for each client.

Request response time is one of the key performance measures in a theater booking system. It is an NFR that is represented as a 'Response time' stereotype in the UML activity diagram (Figure 1). This stereotype is related to every interaction between the client and the server. In general terms, response time can be defined as the amount of time system takes to process a request after it has received one. A control flow in an activity diagram can be assigned the stereotype. In the algorithm that is used to enforce for this stereotype, the time is noted right before the request is sent to the server and the difference is measured once the response is received. The difference between the send and receive time gives the response time for the request. Specific stereotypes are used to represent different latency requirements. Examples are "low latency: and "high latency". Runtime configuration can define the allowable time for each stereotype. Response time for each request from the users is measured, and the average response is calculated for the overall system. This is particularly useful to measure the overall system performance and compare it over time. If the average response time increases, we can further get the average response of each module/type of requests and find the bottlenecks. Algorithm 1 shows the algorithm implemented to guarantee this NFR. In the algorithm, the client notifies the server when the timeout occurs to enable the server to rollback any partial work completed.

 ISBN: 978-1-61208-460-2

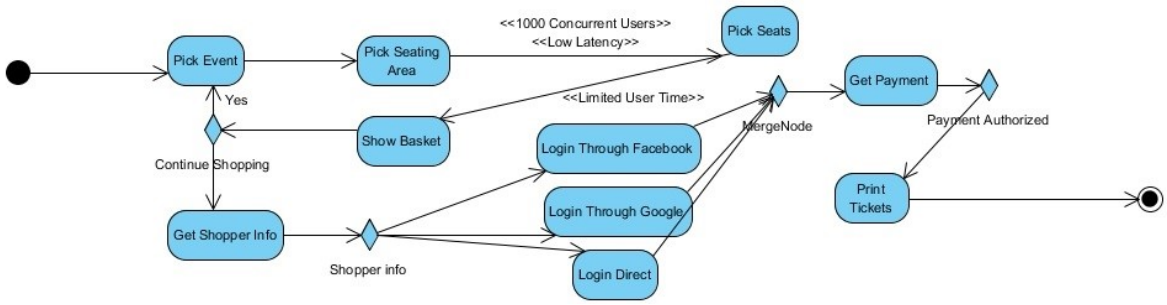

Figure 1 Activity Diagram for A Theater Booking System

Concurrency is a robustness measure of applications, and especially for any online booking system. This is represented as 'Concurrent Users' stereotype in the UML activity diagram (Figure 1). We represent the threshold concurrency as the 'Concurrent Users' stereotype in the UML activity diagram (Figure 1). We implemented this by spawning a thread pool with the size based on stereotype. The server then handles the request by creating a queue. Requests are pulled from the queue and assigned a thread from the pool to process the request. In the implementation of this stereotype, we measure the latency of the request by noting the time right before the request is sent to the server also when the response is received. This difference between the two values gives the latency for that request. This latency for each request is measured and the queue time is appended to the log. The log of measurements is particularly useful to measure the overall system performance and compare it over time. If the average latency time increases, we can further get the average latency of each module/type of requests and find the bottlenecks. When handling the concurrency stereotype, the bottleneck is often caused by a pool of threads that is smaller than the demand on the server. Algorithm 2 shows the algorithm implemented to guarantee this NFR.

Handling the situation where a user does not respond to a form in an appropriate amount of time is another important NFR for many systems. In the theater booking system, while the user is picking a seat, resources are locked from other users. The time the locks are held needs to be minimized. We represent the form response time requirement as 'Limited user time' stereotype in the UML activity diagram (Figure 1). The stereotype is specific to the pick seats activity in the ticketing application domain. The user should be given a limited time to respond when selecting the seats. We implemented this by binding an event to the request submission of the client. When the user tries to pick the seats, the client application polls continuously to check if the request is sent during the specified time. When there is a delay of more than the time specified by the stereotype, then the user gets a message indicating that

Algorithm 1. Request Response Timeout

INPUT: XML of Send to Server, timeout
OUTPUT: XML of response with server

```
Send request to server
Set timer to fire every second
Set timeExpired = 0
Do
   Check if response is received
While timeExpired < timeout or response received
If not response received
   Set response to time expiration error
   Set response to timeout error
    Notify server of timeout
End if
Return response
```

Algorithm 2. Concurrency

INPUT: XML of request, timeout
OUTPUT: XML of data entered or XML with error

```
Check if anythreads in pool
If no threads in pool
   Set timer to fire every second
   Set timeExpired = 0
   Do
      Check if thread available in the pool
   While timeExpired < timeout or thread received
If not thread received
   Set response to timeout error
ELSE
    Execute request in thread
End if
Return response
```

Algorithm 3. User Response Timeout

INPUT: XML of form to display, timeout
OUTPUT: XML of data entered or XML with error

```
Show form to user
Set timer to fire every second
Set timeExpired = 0
Do
   Check if response is received
While timeExpired < timeout or response received
If not response received
   Notify user of time expiration
   Set response to timeout error
End if
Return response
```

there locks have been released. If the request is sent before the specified time, then the user will proceed to next activity. Algorithm 3 shows the algorithm implemented to guarantee this NFR.

IV. CONCLUSION/FURTHER RESEARCH

In this work, we show that it is possible to model many cloud based software NFRs using UML stereotypes. UML diagrams have been used for years to model the functional requirements of the application. We extended the modeling of functional requirements by using UML stereotypes to model the NFRs in the same design model. The UML stereotype is transformed to application code that guarantees the NFR will be enforced. Future work will enhance our work to include OCL constraints to broaden the type of NFRs that can be modeled and transformed into cloud application code.

REFERENCES

[1] B. Bruegge and A. Dutoit, Object-Oriented Software Engineering, Prentice Hall, Inc, 2010.

[2] T. Mullaney, "Obama adviser: Demand overwhelmed HealthCare.gov," USA Today, 06 10 2013. [Online]. Available: http://www.usatoday.com/story/news/nation/2013/10/05/health-care-website-repairs/2927597/. [Accessed 24 02 2016].

[3] M. M. Rahman and S. Ripon, "Elicitation and Modeling Non-Functional Requirements – A POS Case Study," *International Journal of Future Computer and Communication,* vol. 2, no. 5, pp. 485-489, 2013.

[4] C. J. Pavlovski and J. Zou, "Non-functional requirements in business process modeling," *Proceedings of the Fifth on Asia-Pacific Conference on Conceptual Modelling, vol. 79,* 2008.

[5] M. Glinz, "Rethinking the Notion of Non-Functional Requirements," *Third World Congress for Software Quality, Munich, Germany,* 2005.

[6] Alexander, I, "Misuse Cases Help to Elicit Non-Functional Requirements," *Computing & Control Engineering Journal, 14, 40-45,* 2003.

[7] R. Ajith and A. Sheth, "Semantic Modeling for Cloud Computing, Part I," *Computing,* vol. May/June, pp. 81-83, 2010.

[8] S. Charlton, *Model Driven Design and operations for the Cloud,* Towards Best Practices in Cloud Computing Workshop, 2009.

[9] Object Management Group, "OMG Formal Versions of UML," 06 2015. [Online]. Available: http://www.omg.org/spec/UML/. [Accessed 11 09 2015].

[10] Object Management Group, "Unified Modeling Language: Supersturcture," 05 02 2007. [Online]. Available: http://www.omg.org/spec/UML/2.1.1/. [Accessed 08 01 2013].

[11] Object Management Group, "OMG Formally Released Versions of OCL," 02 2014. [Online]. Available: http://www.omg.org/spec/OCL/. [Accessed 09 11 2015].

 ISBN: 978-1-61208-460-2