Dynamically Tracing Non-Functional Requirements through Design Pattern Invariants

Jane Cleland-Huang and David Schmelzer
DePaul University, Chicago
{jhuang, dschmelzer}@cs.depaul.edu

Abstract

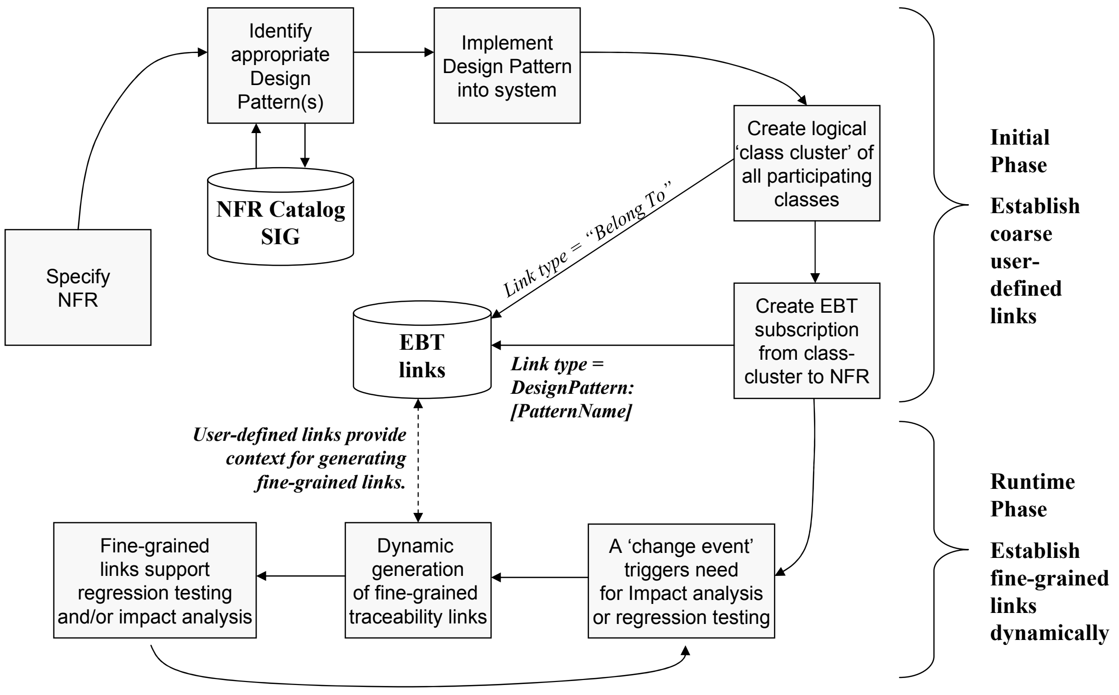
Nonfunctional requirements (NFRs) are critical to the successful implementation of almost every nontrivial software system. This is evidenced by the fact that many documented system failures are directly attributed to the inadequate implementation and maintenance of NFRs. Although tracing NFRs could alleviate this problem through supporting activities such as requirements validation, impact analysis, and regression testing, the task is complicated by their tendency to exhibit complex interactions and to have a global and far-reaching impact upon a software system. This paper describes an approach for establishing traceability between certain types of NFRs and design and code artifacts, through the use of design patterns as intermediary objects. By synergistically utilizing both static and dynamically generated links, EBT_{DP} minimizes the cost and effort of establishing and maintaining traceability links.

1 Introduction

Requirements traceability is a critical component in the long term maintenance of any medium or large scaled software application. Traceability links define relationships between requirements and design artifacts, and support a number of crucial activities related to requirements validation, impact analysis, regression testing, and knowledge management [1,2]. Traditionally these techniques have focused upon the functional requirements of the system, however if non-functional requirements (NFRs) such as performance, reliability, scalability, and safety are not considered, then functional changes may introduce unexpected side effects resulting in both immediate and long-term degradation of the system quality.

NFRs come in many different shapes and sizes, and there is therefore no single traceability technique that can be applied in every case. For example, certain types of requirements that are traditionally considered to be nonfunctional, tend to decompose into lower level requirements that are quite functional in nature, and can be traced using conventional methods. Many security requirements fall into this category. However, other NFRs such as those related to reliability and scalability tend to have a more global impact upon a system and are therefore difficult to trace without creating and maintaining an excessive number of links. Due to these difficulties, many developers entirely fail to trace NFRs, risking exposure to unpredicted side-effects that may adversely impact critical qualities of the system.

This paper builds on our previous work on Event-based traceability (EBT) [3,4,5,6] by describing a new method, $EBT_{DP,}$ for tracing NFRs through the use of design patterns [7]. Several researchers have recognized the fact that NFRs are often fulfilled through the implementation of design patterns [8,9], making this approach applicable to a significant number of requirements. Its primary advantage is that user-defined and dynamic links can be used synergistically to establish finely-grained traceability links supportive of important software engineering tasks such as automated impact analysis and regression testing, without the overhead of maintaining an excessive number of static links. For the purposes of this paper we therefore distinguish between a static user-defined link for which the user explicitly defines a relationship between two artifacts, and a dynamic link which is generated by the system at runtime.



Identify appropriate Design Pattern(s)
Implement Design Pattern into system
Create logical 'class cluster' of all participating classes
Initial Phase
Establish coarse user-defined links
NFR Catalog SIG
Specify NFR
Link type = "Belong To"
EBT links
Create EBT subscription from class-cluster to NFR
Link type = DesignPattern: [PatternName]
User-defined links provide context for generating fine-grained links.
Runtime Phase
Establish fine-grained links dynamically
Fine-grained links support regression testing and/or impact analysis
Dynamic generation of fine-grained traceability links
A 'change event' triggers need for Impact analysis or regression testing

Figure 1 The EBT_{DP} process

1.1. Current Practices

Typical traceability practices [10,2] utilize user-defined approaches such as matrices, hyperlinks, or the traceability features embedded in requirements management tools such as DOORS and Requisite Pro. Unfortunately, although it is easy to create links using these methods, it can be extremely hard to maintain them in an accurate state over time [2,11]. In practice, high-end traceability users attempt to reduce the number of links that need to be maintained by using a mixture of fine and coarse grained links [10], however this inhibits the possible trace automation of the parts of the system for which no fine-grained links exist. Establishing static traceability links for NFRs with global impact upon a system could quickly result in an excessive number of hard to maintain links and should therefore be avoided.

Several researchers have investigated the use of dynamic link generation in place of static links [12,13]. However these approaches are primarily suitable for linking requirements and artifacts that exhibit a high lexical correlation. For example, information retrieval techniques can be used to dynamically link a requirement with artifacts that contain similar words and phrases. However, in the case of NFRs this type of high lexical correlation between the requirement and its implemented artifacts often does not exist.

The use of design patterns as intermediary objects introduces the possibility of rule-based link generation in place of lexical-based generation. Although current techniques for detecting implemented design patterns at the system level, lack the precision required to support traceability, the new approach proposed in this paper describes how a small number of user-defined links can provide a context in which precision can be improved and useful links can be successfully generated. These links can then support activities such as regression testing and impact analysis. This new approach utilizes the functionality of the traceability event engine developed as part of our previous work on Event-Based traceability (EBT) and is therefore called $EBT_{DP.}$ Figure 1 provides an overview of the EBT_{DP} process.

The following section discusses some of the issues surrounding the tracing of NFRs and provides an example that is used throughout the remainder of the paper. Section 3 defines the EBT_{DP} traceability mechanism and section 4 discusses the use of design pattern invariants to drive the dynamic generation of

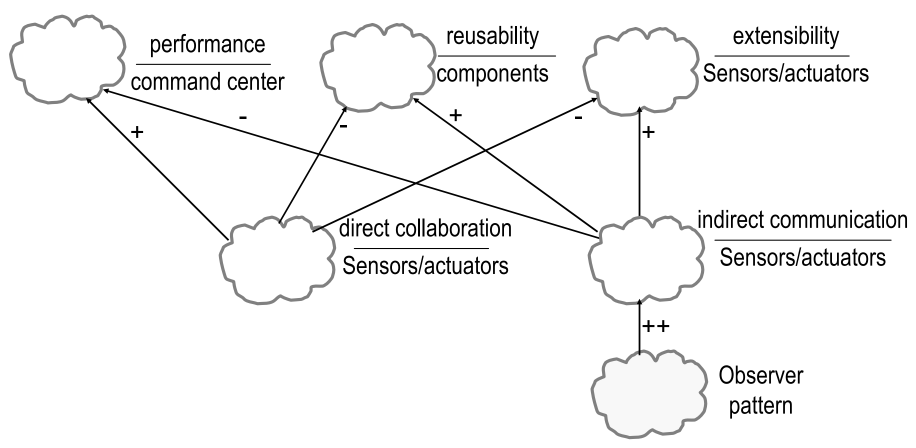


Figure 2 Tracing NFRs to design patterns through a Softgoal Interdependency Graph

links. Section 5 then concludes with a discussion of the findings and an outline for future work.

2. Tracing Non-Functional Requirements

Consider the following three NFRs taken from a critical application in which a Mission Command Center application is responsible for monitoring and controlling a set of instruments:

- **MCC_NFR1** (extensibility)
  "The system shall support the dynamic addition of monitored sensors at runtime"
- **MCC_NFR2** (performance)
  "The system shall respond in a timely manner to all emergency situations."
- **MCC_NFR3** (reusability)
  "The system shall be designed to support reuse of software components"

NFRs are often initially stated in fuzzy or incomplete terms and then refined during the analysis and development process [14].

2.1 Design Patterns as Operationalizations

A softgoal interdependency graph (SIG), as depicted in Figure 2, provides a useful modeling notation for reasoning about and analyzing NFRs, and for determining when an NFR might be fulfilled through the use of a design pattern. Within a SIG, NFRs are represented as 'softgoals', refined into lower level softgoals and 'operationalizations'. An operationalization represents a potential solution for fulfillment of a softgoal. Interdependencies between softgoals and operationalizations are captured as links, and annotated according to whether they *make (++)*, *help (+)*, *hurt (-)*, *break(--)*, or have an *unknown* impact upon their parent node in the graph.

As illustrated in Figure 2, the two NFRs related to extensibility and reusability can be fulfilled through the implementation of the observer design pattern [7]. The observer pattern 'defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically'. This pattern enables dependent observers to be dynamically added at runtime and supports interaction between multiple observers without requiring any direct communication between them.

The objective is to establish links between each NFR and the components of the implemented observer pattern in order to support ongoing compliance monitoring that the design pattern remains implemented in the design. In other words, activities such as regression testing and impact analysis should ensure that the NFRs related to extensibility and reusability remain validated. For the purposes of this paper we use the term 'implemented classes' to refer to both UML classes and coded classes, as the EBT_{DP} traceability techniques are equally applicable to both code and UML classes.

2.2 Traditional User-Defined Links

Utilizing a user-defined traceability technique to establish fine-grained traceability between the extensibility and reusability NFRs and the components of this observer pattern would require

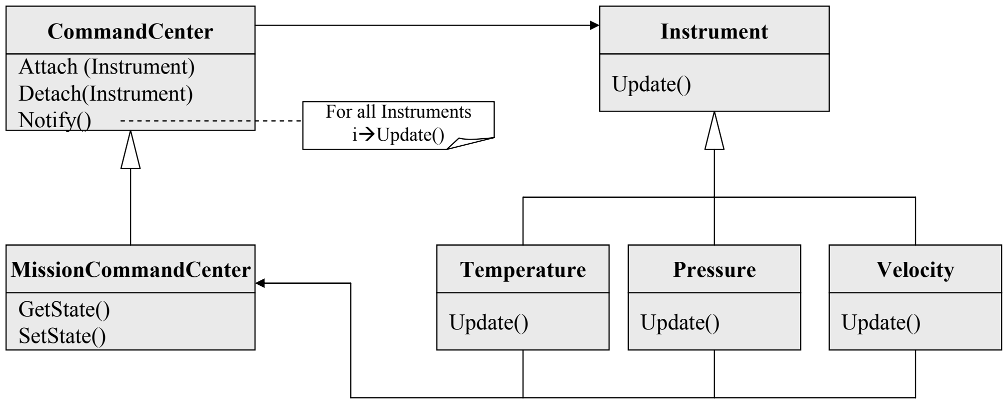
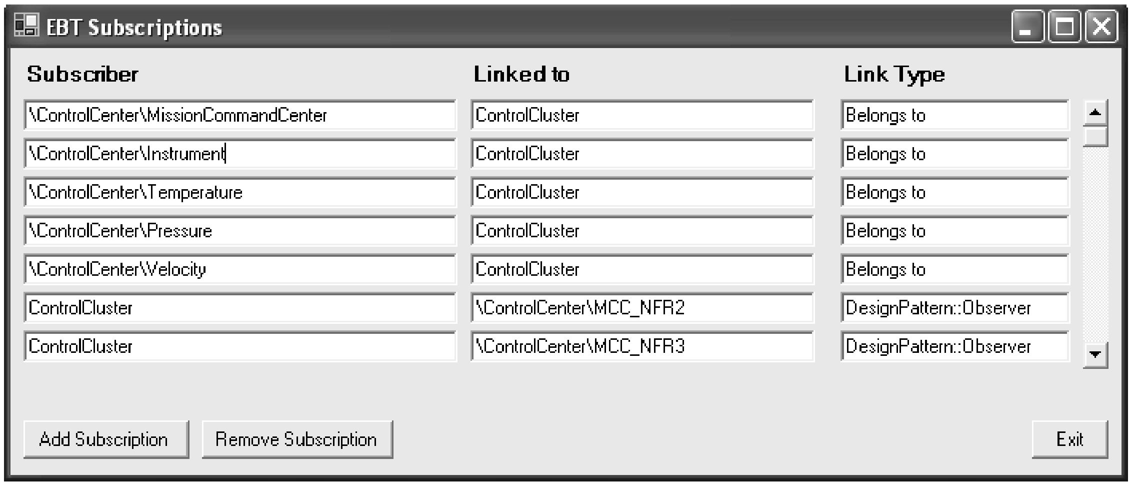


Figure 3 Observer Pattern applied within the MissionCommandCenter



Subscriber	Linked to	Link Type
\ControlCenter\MissionCommandCenter	ControlCluster	Belongs to
\ControlCenter\Instrument	ControlCluster	Belongs to
\ControlCenter\Temperature	ControlCluster	Belongs to
\ControlCenter\Pressure	ControlCluster	Belongs to
\ControlCenter\Velocity	ControlCluster	Belongs to
ControlCluster	\ControlCenter\MCC_NFR2	DesignPattern::Observer
ControlCluster	\ControlCenter\MCC_NFR3	DesignPattern::Observer

Figure 4 User defined links established as EBT subscriptions

links to be established between each NFR and each of the critical observer pattern methods. These are depicted in Figure 3 and would minimally include links to the *Instrument:Update* method in all subclasses of Instrument, *MissionCommandCenter*: *Attach*, Mission-*CommandCenter*:*Detach*, and *Mission-CommandCenter*:*Notify*. In our example, if we assume an oversimplified command center with only ten different types of Instruments (ie ten subclasses of Instrument), simply linking both of the NFRs to their Observer Pattern implementations would require 80 traceability links (2 NFRs x 10 Instruments x 4 Links)!

This creates a non-viable situation in terms of the cost and effort involved in establishing and maintaining these links. Furthermore, although these static links can clearly identify many critical components of the pattern within the implementation, they are unable to monitor more sophisticated elements of the design such as the prohibition of direct links between classes of type *Instrument*.

A user-defined approach to linking NFRs through design patterns using matrices, hyperlinks, or existing requirements management tools is therefore prohibitively expensive to implement and limited in expressiveness.

3. An overview of EBT_{DP}

In EBT, traceability links are established as publish-subscribe relationships, and are attributed according to the type of link. EBT_{DP} requires only a few user-defined links because the remaining fine-grained links will be dynamically generated. In this example, we will assume that the observer pattern is implemented as depicted in the class diagram of Figure 3. In this case the participating classes include *MissionCommandCenter*, *Instrument*, *Temperature*, *Pressure*, and *Velocity*. These classes are packaged into a cluster labeled '*ControlCluster'* through the use of EBT subscriptions. The resulting cluster, which only exists as an abstract entity is then linked to each of the NFRs that are fulfilled by the Observer Pattern, again using EBT subscriptions.

These subscriptions are shown in Figure 4.

In comparison to our previous example of using fine-grained user-defined links, this approach requires one link for each instrument class in order to subscribe to the class cluster, and one link from the class cluster to each NFR that it fulfills. Therefore for the mission command center with 10 instruments, the number of traceability links is reduced from 80 to 14! In a similar situation with 20 instruments the number of links would be reduced from 160 to 24. Furthermore, all of these links are coarse-grained and therefore much simpler to maintain than the fine-grained links that would be required using a non-dynamic approach.

EBT_{DP} consists of two distinct phases. These are illustrated in Figure 1. In the first phase, which occurs during inception, elaboration, and construction of the system, the initial user-defined traceability links are established. In the second phase, which occurs during the ongoing maintenance and refinement of the system, fine-grained links are dynamically generated.

Early in the software development process, the NFRs are elicited and analyzed. A SIG or other method can be used to identify relationships between NFRs and design patterns. During design and construction, the design pattern is implemented into the UML model and related code. The classes in which the elements of the design pattern are instantiated are identified and composed into a logical cluster. A traceability link is established between the cluster and the NFR and assigned the link type of the implemented design pattern. In this case, the link is assigned the type attribute of "DesignPattern:Observer".

3.1 Event Triggers in EBT_{DP}

In any event driven system, event handlers are responsible for handling specific events as they occur. In EBT, events are published as generic event messages to the EBT event server, and then customized according to the type of subscription placed by the dependent artifact. Customized event messages are then forwarded to the event handler for each impacted artifact and handled accordingly. EBT is an example of a process driven environment in which the logic of the enactment domain is distributed amongst the event handlers [15].

In EBT_{DP} the purpose is to trigger the dynamic generation of links when a critical event occurs. For example, if a change is made to a class that implements a design pattern, it would be useful among other things, to trigger an event that ultimately results in the re-execution of regression tests to determine if the change adversely impacted the implementation of the design pattern. In this way, the traceability links would support long-term compliance monitoring of the design to its stated NFRs.

The full EBT methodology is described in [3,4,5,6], however this section illustrates its support for dynamic link generation. As an example, consider the situation that would occur if the class "Temperature" were modified. In this situation the following event message is published: "`ChangeEvent:Class:Temper-ature|Modified`"

This message is received by the EBT event server, which determines that the Temperature class is subscribed to the cluster "ControlCluster". As no event handler is attached to the class 'Temperature', the message is forwarded to 'ControlCluster' as: "`ChangeEvent:Class-Cluster:Temperature|Modified`".

The EBT event server then determines that ControlCluster is subscribed to MCC_NFR2, and MCC_NFR3, and that the link is attributed with the type 'DesignPattern:Observer'.

The logic in the event server then causes the triggering of the mechanism to dynamically generate links between the NFR and the class implementations. These traceability links can then be used to support a full impact analysis or regression testing. The following section discusses the dynamic generation of these links.

4. Dynamic Link Generation

In EBT_{DP} links are generated dynamically according to the invariant rules of the relevant design pattern. The objective is to use the invariants of the design pattern to identify critical components within the implemented classes that should be traceable, and to generate related links. In this section we therefore examine related work in detecting and reverse engineering design patterns.

4.1 Design Pattern Detection

Several researchers have examined the problem of detecting design patterns from design and code artifacts [16,17]. This problem is closely related to that of dynamically generating traceability links, because once a pattern and its implemented components can be identified, then traceability links can be established between the elements of the design pattern and those components. Brown stated that a design pattern "is detectable if its template solution is both *distinctive* and *unambiguous*."[16] He determined that some design patterns such as Interpreter [7] are difficult to detect because they rely upon general principles rather than specific design fragments, while others are quite easy to detect because they exhibit very specific and unique patterns of interaction. In the general case, many design patterns fall somewhere in between these two extremes, with certain parts of the pattern being easily detectable, and others being more obscure. In the general pattern detection problem, detecting individual parts of the pattern may not provide a reasonable level of confidence to clearly demonstrate the use of the whole pattern.

Heuzeroth also investigated the problem of design pattern detection [15]. He defined each design pattern to be detected in terms of its invariant features expressed as a tuple of program elements including classes, methods, and attributes, and representing the restrictions and rules of a design pattern. For example, the invariants of the Observer pattern are represented by the tuple (*Subject.addListener*, *Subject.removeListener*, *Subject.notify*, *Listener. update*). The expected behavior of each of these elements is then defined.

The Subject class must be capable of dynamically adding observers. To accomplish this, it instantiates a *Subject.addListener* method capable of receiving an object as a parameter and either storing that object locally for future use or passing it to another method for storage. Because this method may be called by any name it is necessary to search for a method that matches the required behavioral pattern of the *add.Listener* method.

Heuzeroth's approach is implemented in two stages. During the initial static analysis a set of candidate patterns are identified. In practice, this stage tends to output many false positive patterns, and therefore a second stage dynamic analysis is performed to filter out false patterns. During this stage the simulated runtime behavior of the classes and their interactions is observerd. Heuzeroth found that as long as the dynamic analysis actually caused the execution of the candidate pattern elements, it was generally possible to effectively distinguish between true and false pattern detections.

4.2 Linking Pattern Elements to the NFR

The underlying concept of EBT_{DP} is that if an NFR is implemented through a design pattern, and if that design pattern can be detected, then finely grained traceability links can be generated as needed to support important tasks such as regression testing and impact analysis.

Dynamic link generation utilizes the techniques described in the previous section with a couple of critical differences. First, the precision of the approach is increased because the pattern detection analyzer no longer is searching for an unknown set of design patterns within a large system-wide search space. The search space is constrained by the initial set of user-defined links that specify the class cluster in which the pattern resides. Furthermore the objective changes from the broad task of finding any type of implemented pattern to answering the very specific question of "Where in this class cluster is this specific pattern implemented?" This reduction in scope allows for a much more precise result.

Heuzeroth also pointed out that the dynamic analysis phase is only effective if the candidate pattern elements are actually executed. In EBT_{DP} a single additional traceability link to a specific test case can guarantee execution of the implemented pattern.

Following the dynamic analysis phase the pattern invariants are either linked to their implemented elements or marked as 'missing'. Missing elements indicate a possible problem in ongoing compliance to the NFR because the loss of

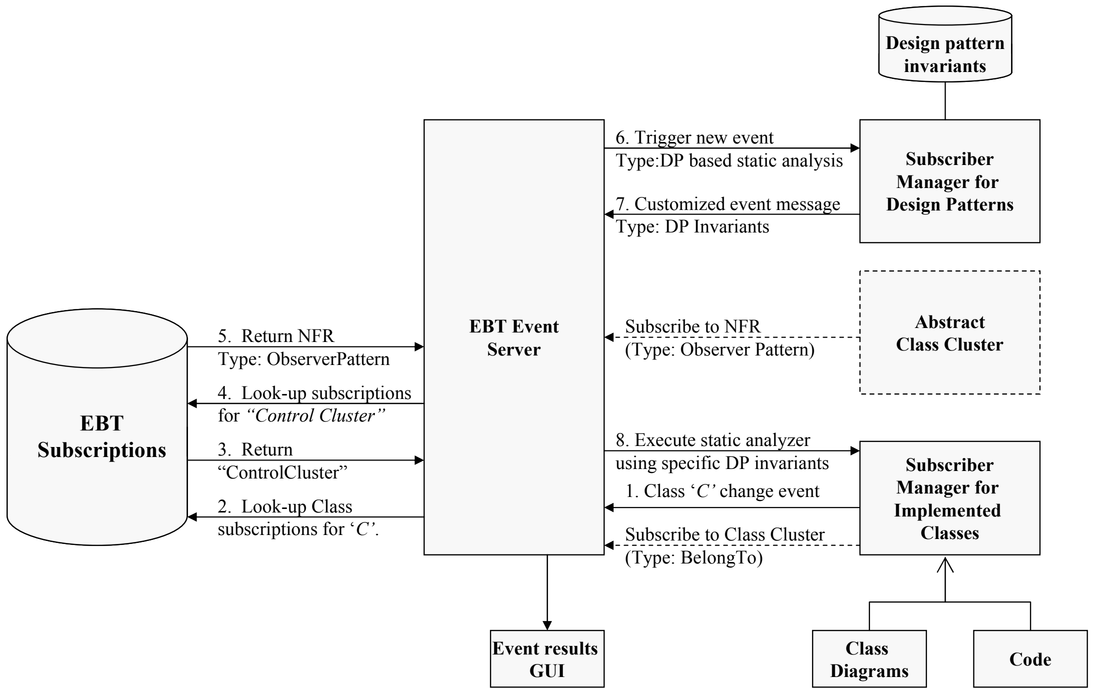

Design pattern invariants
6. Trigger new event
Type:DP based static analysis
7. Customized event message
Type: DP Invariants
Subscriber Manager for Design Patterns
EBT Event Server
Subscribe to NFR
(Type: Observer Pattern)
Abstract Class Cluster
5. Return NFR
Type: ObserverPattern
4. Look-up subscriptions for "Control Cluster"
EBT Subscriptions
3. Return "ControlCluster"
2. Look-up Class subscriptions for 'C'.
8. Execute static analyzer using specific DP invariants
1. Class 'C' change event
Subscriber Manager for Implemented Classes
Subscribe to Class Cluster
(Type: BelongTo)
Event results GUI
Class Diagrams
Code

Figure 5. Supporting traceability of NFRs through EBT$_{DP}$

a critical part of the pattern suggests that it is no longer implemented as originally planned and can no longer serve to validate the NFR. These links are then used to conduct normal software engineering activities such as impact analysis.

4.3 Support for Impact Analysis

The purpose of EBT$_{DP}$ traceability is to support activities such as impact analysis and regression testing. Consider for example a proposed functional change that will impact a class such as the *Temperature* class from the *MissionCommand-Center*. The developer issues an EBT$_{DP}$ query against the *Temperature* class to determine how changing it might impact the overall system. As depicted in Figure 5, an event message is issued to the event server and as the *Temperature* class belongs to the *ControlCluster*, the event message is forwarded to all artifacts to which the *ControlCluster* subscribes. Activating the DesignPattern:Observer link from the Control-Cluster to the NFR triggers the EBT$_{DP}$ mechanism to dynamically generate links from the NFR to the implemented components of the observer pattern. These links clearly identify critical elements of the pattern implemented as classes and methods in the design and code.

The dynamically generated traceability links therefore provide the developer with information about the class that is about to be changed, by clearly identifying critical elements within the class that should remain implemented in order to maintain the integrity of the related NFR. The developer is therefore equipped to make effective decisions concerning the proposed change and its general impact upon the system. Following the change, the link generation process can be repeated as a form of automated regression testing to determine if the design pattern continutes to remain intact. Failure to identify the pattern invariants indicates that the NFR might no longer be adequately fulfilled in the design and code.

5. Conclusions

Establishing traceability from NFRs to design components and code, certainly represents a non-trivial problem for which no single solution is available. The sheer diversity of NFRs and their related implementation techniques indicates that it is unlikely that a single traceability technique will be optimal for all types of NFRs. This paper describes one method applicable to NFRs that are fulfilled through the implementation of a design pattern. EBT$_{DP}$ provides a powerful alternate to establishing traceability through the use of static links. One of its primary advantages is that the known and predefined rules of the design pattern enable fine-grained links from the pattern to specific class implementations to be generated dynamically. This reduces the need for establishing explicit traceability links, and increases both the maintainability and the expressiveness of the approach.

Although the various components have already been demonstrated to work, either within EBT, or through the work of other researchers, we are currently in the process of implementing the full EBT_{DP} approach in a way that would be applicable to a broader spectrum of design patterns and NFRs. To accomplish this, it is necessary to analyze a more extensive set of design patterns to identify their invariants, to demonstrate the use of a generic static analyzer to support identification of design patterns within the EBT linked class clusters, and to investigate the further extension of the NFR catalog so that identifying the use of design patterns to fulfil NFRs is further facilitated. Once this work is completed we will conduct further experiments with this approach utilizing our EBT framework.

As a broader contribution to the advancement of traceability practices, this approach suggests that hybrid approaches utilizing a combination of static and dynamically generated links can potentially provide effective solutions for implementing long-term and maintainable traceability.

Acknowledgments

This work is partially funded by NSF grant CCR-0306303.

References

1. M. Jarke, "Requirements Traceability", *Communications of the ACM,* Vol. 41, No. 12, Dec. 1998, pp. 32-36.

2. O. Gotel and A. Finkelstein, "An Analysis of the Requirements Traceability Problem," *Proc. 1st Int'l Conf. Requirements Engineering.,* 1994, pp. 94-101.

3. J. Cleland-Huang, C.K.Chang, and M. Christensen, 'Event-Based Traceability for Managing Evolutionary Change', IEEE Trans. on Software Eng., Vol. 29, No. 9, September, 2003. pp. 796-810.

4. J. Cleland-Huang, C. K. Chang, and J. Wise, "Automating Performance Related Impact Analysis through Event Based Traceability," *Requirements Engineering Journal,* Springer Verlag, Vol 8, No. 3, August, 2003. pp 171-182.

5. J. Cleland-Huang, C.K. Chang, K. Javvaji, H. Hu, G.Sethi, J.Xia, set al., "Requirements Driven Impact Analysis of System Performance," *IEEE Proc. of the Joint Conf. on Requirements Engineering*, Essen, Germany, Sept. 2002.

6. J. Cleland-Huang and C. K. Chang, "Supporting Event Based Traceability through High-Level Recognition of Change Events," *IEEE Proc. of COMPSAC*, Oxford, England, Aug. 2002, pp. 595-600.

7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Mass., 1995.

8. Daniel Gross and Eric Yu, "From Non-Functional Requirements to Design through Patterns", Requirements Engineering Journal, Vol 6, No. 1, 2001, pp. 18-36.

9. N. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley, 1995. Appendix available online at http:// sunnyday.mit.edu/ accidents/therac.pdf.

10. B. Ramesh, and M. Jarke, "Toward Reference Models for Requirements Traceability", *IEEE Trans. on Software Eng.,* Vol. 27, No. 1, Jan 2001, pp. 58-92.

11. R. Domges and K. Pohl, "Adapting Traceability Environments to Project Specific Needs", *Communications of the ACM,* Vol. 41, No. 12, 1998, pp. 55-62.

12. G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, E. Merlo, "Recovering Traceability Links between Code and Documentation", *IEEE Trans. On Software Engineering*, Vol. 28, No. 10, pp. 970-983.

13. E.Tryggeseth and O. Nytrø, "Dynamic Traceability Links Supported by a System Architecture Description", *Proc. of the IEEE International Conference on Software Maintenance*, Bari, Italy, Oct. 1997.

14. L. Chung, B. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, 2000.

15. K. Pohl, K. Weidenhaupt, R. Domges, P. Haumer, M. Jarke, and R. Klamma, "PRIME – Toward Process-Integrated Modeling Environments", *ACM Transactions on Software Engineering and Methodology,* Vol. 8, No. 4, October 1999, pp. 343-410.

16. Dirk Heuzeroth, Thomas Holl, Gustav Högström, Welf Löwe, *Automatic Design Pattern Detection*, 11th International Workshop on Program Comprehension, co-located with 25th International Conference on Software Engineering, Portland, IEEE, May 2003.

17. K. Brown, "Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk", Master Thesis, University of Illinois at Urbana-Champaign, 1997