GenQA: Automated Addition of Architectural Quality Attribute Support for Java Software

Sarthak Grover
Business Objects, an SAP company
332 Front S S, 3rd Fl
La Crosse, WI 54601 USA
sarthak.grover@gmail.com

Nigamanth Sridhar
Electrical and Computer Engineering
Cleveland State University
Cleveland OH 44115
n.sridhar1@csuohio.edu

ABSTRACT
Non-functional requirements for software systems are typically specified using informal notions such as quality attribute scenarios. Further, implementation strategies for such non-functional attributes are frequently common across systems with quite different functional requirements. In such cases, the time invested in implementing these quality attributes could be salvaged, thereby reducing the project lifetime and increasing software quality. In this paper, we present the design and prototype implementation of a tool and associated framework that enables software engineers to effectively capture non-functional requirements, and then automatically generate implementations of these requirements to be added to the application being built. The quality attribute implementations are generated as aspects (AspectJ in the prototype) that can be woven in with the application code (Java) with minimal development effort.

Categories and Subject Descriptors
D.2.11 [Software Architectures]: Languages; D.2.12 [Design Tools and Techniques]: Computer-aided software engineering

General Terms
Design

Keywords
Aspects, automated generation, software architecture

1. INTRODUCTION
Maintainability, usability, traceability, security, fault tolerance, etc., are properties of software systems that are not directly related to the functional behavior of the system. Instead, such non-functional properties are considered external to the functional specification. Architectural decisions often affect non-functional properties of software systems.

The preferred solution to this problem is to move the specification of non-functional requirements out of the system specification, and instead include them as part of the architectural description of the system. By way of making architectural decisions, the software engineer can now endow a system with the requisite non-functional property. Non-functional requirements are described as quality attribute scenarios as part of the architecture description [1]. These quality attributes get into the design specification in the form of tactics, and are finally implemented in conjunction with the implementation of the functional system.

Our view in this paper is that implementation of architectural tactics to incorporate quality attributes should involve an activity that is separate from the implementation of the functional specification. Moreover, the property that the implementation of such tactics does not depend upon the functional behavior of the system under consideration should be exploited as much as possible.

To support this view, we present in this paper the design and a prototype implementation of GenQA, a tool and framework that allows software engineers to:

1. Capture non-functional requirements and automatically place them in context with respect to the functional specification,
2. Generate code to implement these architectural tactics in such a way as to cleanly separate the functional concerns of the system from the non-functional concerns.

The rest of this paper is organized as follows. In Section 2, we present the design of GenQA, and describe the prototype implementation for Java and AspectJ. In Section 3, we present some examples to demonstrate our system. After presenting some related research in Section 4, we conclude with a summary of our contributions in Section 5.

2. THE GENQA FRAMEWORK
The GenQA framework includes several components that collaborate to capture non-functional requirements, and generate the
aspect code to implement such requirements. The overall workflow of using the GENQA framework is presented in Figure 1.

2.1 Capturing Non-Functional Requirements

Non-functional properties are described at two levels. At a generic level, it is possible to give a fairly concrete description of a property itself independent of the system under construction. For example, a message logging service can be described as one that records and maintains a log of all inter-object activity in a software system. Such a description alone, however, is not sufficient for the property’s realization in a particular system. The second level of description is necessary for this, where the generic description is specialized to reference individual and particular parts of the system being built. For example, it may be necessary to identify particular classes in the system under construction, and methods in these classes as ones to be logged for the message logging service.

In order to allow developers to describe non-functional attributes of a software system, GENQA provides for both levels of descriptions. The generic description of the non-functional attribute is encoded in the tool itself, and using this, the tool provides a wizard-like graphical interface using which the developer can specialize the attribute to the system under development. At this point, GENQA supports three kinds of services: (1) services that only monotonically add behavior to the system, (2) services that cause redirection of method invocations, and (3) services that may cause the execution of a method invocation to be deferred. In Section 3, we present an example of these kinds of services.

2.2 Specifying Non-functional Requirements

For specifying non-functional requirements, we use an adaptation of a behavioral model for specifying container services [9]. The specification approach presented in [9] provides a way to capture the additive behavior introduced by a container service. As such, application services provided by a container architecture such as the J2EE [10] can be viewed as a delivery mechanism to augment a software system with non-functional properties. It is therefore appropriate to use this specification scheme for modeling non-functional requirements in GENQA.

We present a brief overview of the specification approach to provide context for the rest of the presentation here. The basic idea is that the behavior of a non-functional property is specified in terms of how the said property modifies the contract of each method in the underlying system it affects. Accordingly, the specification of a generic non-functional service contains four parts: (1) additional state introduced into the base component, (2) a predicate, pre-modifier, that is conjunctively combined with the pre-condition of every method in the base component, (3) a predicate, post-modifier, that is conjunctively combined with the post-condition of every method in the base component, and (4) an optional set of methods that are introduced into the base component. Note that the additional state and new methods are only usable from the implementation of the non-functional attribute; these state variables and methods are not available to the base component.

In order for the specification to be usable in our automated code generation infrastructure, we represent the specification using XML. The following is an excerpt of the generic XML schema that we use to specify and represent non-functional requirements:

```xml
<xs:schema>
  <xs:element name="service" type="NFService"/>
  <xs:complexType name="NFService">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="state" type="State"/>
      <xs:element name="pre-modifier" type="Predicate"/>
      <xs:element name="post-modifier" type="Predicate"/>
      <xs:element name="method" type="Method"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

For each non-functional attribute that needs to be added, the above specification needs to be created. This is a necessarily manual, one-time effort. Once this activity has been completed once, and there is confidence that the specification is complete and correct, GENQA can use it over and over again to generate tactic implementations to provide these attributes for different systems. This part of the specification remains static, and does not need to change from one system to the next.

Further, based on these specifications, GENQA can be extended to include and support new kinds of quality attributes. The only requirement is that the specification of new quality attributes conform to the specification scheme presented above. In addition to specifying new quality attributes using the above scheme, GENQA also needs to be extended to add support for contextualizing the new quality attribute for specific systems. Our design of GENQA has extensibility as a core goal. As such, GENQA supports the addition of new quality attributes by way of a seamless plug-in architecture. Only the requirement capture part (front-end) needs to be implemented for each new quality attribute. The rest of the system, including the aspect code generation modules, remain the same.

2.3 Contextualizing Quality Attribute Specs

Once the quality attribute(s) for a particular system have been

Figure 1: The GENQA work flow
identified, they need to be contextualized to that system. This means that the definitions of the quality attribute need to be re-stated in terms of the classes and methods that are present in the system under development. GENQA provides support for this step as well.

GENQA uses a wizard-like graphical interface to collect information about the system under construction, and for specializing a generic quality attribute definition to a particular system. The set of questions that the wizard asks a system developer is based on the specification of the service itself. For this Message Logging service example, the specification is shown in Figure 2.

Figure 2: Specification of Logger tactic, reproduced from [9].

The state of Logger consists of two strings — ILog and OLog. ILog is the log of all method invocations on their way from the caller to the target. Each element in the string is a pair consisting of a method name, and the sequence of actual parameter values passed to the named method. OLog is the log of all method invocations on their way from the target object back to the caller. OLog contains the final values of method parameters and the return value.

The pre-modifier of Logger adds to ILog a new pair with this method (method name), and the actual values of each element in thisthismethod.args (method arguments). @ILog here refers to the value of ILog in the state immediately preceding the start of the body of Logger during the target method call (LoggerBodyPre). The post-modifier of Logger creates a new pair with the name of thismethod and the sequence of return values (the actual values of elements in thismethod.args), concatenated with thismethod retval. This new pair is added to OLog.

Based on this description of the service, the wizard screen collects information about which classes in the system are to be impacted by this quality attribute tactic. The developer can use a class

```java
public class MethodLoggingHelperCurrentMethod {
    String methodName;
    ArrayList methodArguments = new ArrayList();
    String methodReturnValue;
    
    public aspect MethodLoggingAspect {
        ArrayList ILog = new ArrayList();
        ArrayList OLog = new ArrayList();
        MethodLoggingHelperCurrentMethod currMethod;
        pointcut ImpactMethods69 : execution(* BankAccount.*(...));
        before(): ImpactMethods69 {
            currMethod.methodName = thisJoinPoint.methodName();
            currMethod.methodReturnValue = null;
            for (k : thisJoinPoint.Parameters())
                [ currMethod.methodArguments.add(k.name); ]
            ILog.add(currMethod);
        }
        after(): ImpactMethods69 {
            currMethod.methodName = thisJoinPoint.methodName();
            currMethod.methodReturnValue = thisJoinPoint.returnValue();
            for (k : thisJoinPoint.Parameters())
                [ currMethod.methodArguments.add(k.name); ]
            OLog.add(currMethod);
        }
    }
}
```

Figure 3: The Method Logging aspect generated by GENQA. Note that the code has been re-formatted slightly for presentation.

The translation process is based on that presented in [8]. The XML tree is parsed and methodically processed to produce AspectJ output. Figure 3 shows the AspectJ code generated by GENQA based on the inputs provided. The “helper” type MethodLoggingHelperCurrentMethod (lines 1–5) defines the type as described in the XML attribute specification for storing method data (name, arguments, return value). The additional state that the aspect adds are shown on lines 7–9. Commonly used types (such as ArrayList) are known to GENQA, and the code generator inserts the appropriate initial state for variables of such types.

The pointcut ImpactMethods69 defined on line 10 captures all methods in BankAccount. Depending on the level of specificity the developer chooses to provide in the specification wizard, there may be one or more pointcuts that target methods more narrowly. The aspect contains two advices corresponding to the specification.
The first advice is a before advice that records a method’s information before it is invoked, and the second is an after advice that analogously records information upon return of control.

3. OTHER EXAMPLES

The logging attribute that we have seen so far only monotonically adds behavior to the underlying component’s own behavior. Another example of a service that falls in the same category is object Visualization. These services do not interfere with the normal interaction between an object and its clients. They are simply “observers”. In the remainder of this section, we will look at two more examples of services, ones that are not simply observers but may modify the interaction pattern between the object and its clients.

3.1 Fault Masking

In systems that require high availability, an important quality attribute that is desired is fault masking. When a client invokes a method on a given object, the client must always receive a response\(^2\). As such, the system must be able to tolerate faults without allowing such faults to become system failures. One tactic that is useful to augment a system with the fault tolerance quality attribute is to be able mask faults from clients. That is, even when a particular object fails for some reason, the client must be insulated from the fault. In this manner, the client does not get affected by the fault — the system still works.

As Figure 4 shows, the Fault Masking service causes the redirection of a method invocation to an object instance other than the intended receiver. The service can, in a manner that is transparent to the client, prevent method invocations from being sent to object instances that have failed. This kind of redirection only applies to components that are identity-less—the method call cannot depend on the component’s identity.

The fault masking service needs the set of objects that are currently failed (R1), and for each failed object, the set of objects to which calls can be re-directed (R2). There are different strategies for obtaining R1. In synchronous systems, simple timeouts can be used. In asynchronous systems, however, it is not possible to place such time bounds. We can, however, abstract away that detail and leave it to some failure detector [2]. FMask maintains a set suspects—objects that the failure detector suspects to be failed. If a failed object is remembered, the invocation is directed to an object from the set suspects.

To satisfy R2, the service maintains a set of alternate objects (alt_objs) for every object obj in the container. This way, when the container does encounter a method call whose target is failed, it can look up an alternate object and forward the call to that object. The method setAlternates() can be invoked on a hosted object with a set of alternate objects. The argument to setAlternates() is a Set parameterized by the type of target. All objects in a_objs are added to the set H of container-hosted objects.

\(^2\)Or at least, receive a response with a very high likelihood.

When FMask intercepts a method call to a suspected target, the call is directed to an object from alt_objs(target) that is still alive. If no such alternate object can be found by FMask, the invocation fails. On the return direction, the service does nothing, and therefore does not modify the post-condition.

An abbreviated version of the aspect generated by GenQA to implement the Fault Masking tactic is shown in Figure 5. The aspect implementation ignores several details, and is shortened for presentation. In particular, a gaping omission is that the service always trusts that the first alternate object for a failed object is alive and ready to receive invocations. In reality, it is necessary to check that the chosen alternate object is alive, and if not, the process of finding alternates must continue.

3.2 Atomic Transaction Support

Let us now look at the third class of quality attribute tactics — deferred execution. The client may not immediately see the effect of a method invoked, but will see the effect at a later time. The service we consider here is atomic transaction support. Some components require that certain groups of methods be called in succession; either all of these methods should succeed, or they should all fail. A partial execution may result in an inconsistent state.

Consider a class with four methods m1(), m2(), m3(), and m4(). Further, suppose that the four methods must be executed in an atomic transaction, which is triggered by m1() and finalized by m4(). The behavior that we would like to see is that whenever m1() is invoked, a transaction is initiated. However, the method is not executed yet. Instead the invocation is cached for execution in the future, when the transaction can be executed in its entirety.

When subsequent invocations arrive, they are added to the transaction that is currently open. When the finalizer method, m4() finally arrives, the transaction is now ready to be committed. All the four methods are executed in sequence. If something were to go wrong with any of the four methods, then the entire transaction needs to be rolled back. To enable this, before the transaction begins executing, a checkpoint is recorded (the local state of the object is remembered).

A simplified version of the AspectJ code generated by GenQA for the preceding example is presented in Figure 6. Notice that in this example, there are three pointcuts to capture (1) the trigger method, (2) all methods to be included in the transaction, and (3) the finalizer method. In the advices that correspond to the first two pointcuts, notice that there is no proceed() statement. This means that the call that the client made is never forwarded to the object for execution. It is simply recorded for future execution after all the other methods in the transaction have also been invoked by the client. Eventually, when the finalizer method (m4()) is invoked, the
Another approach [3] aims at accommodating Aspect Oriented Software Development (AOSD) in existing software development processes by adapting and extending current OOP methodology in an aspectual context. This study also aims at identifying and modeling cross-cutting concerns at an early stage in the software development life cycle. The authors emphasize that requirements elicitation is crucial since it is important to identify NFR’s early.

5. CONCLUSION

In this paper, we have presented the design and prototype implementation of GENQA, a tool that supports the automated addition of architectural quality attributes for Java software. The fundamental observation behind this work is that while functional requirements and their implementations are closely tied to individual systems, non-functional requirements are typically cross-cutting, and their implementations look similar across different systems.

However, it is not simply a matter of implementing non-functional properties in a purely generic manner — they must be contextualized to the system under consideration. Accordingly, we have presented a way to specify non-functional quality attribute tactics at two levels: a generic description, and a contextualized description that is specific to a particular system. Also, we have presented tool support to automatically transform and specialize general descriptions of quality attribute tactics into specialized descriptions.

Finally, GENQA includes code generation capabilities to translate the specification of a quality attribute tactic into a corresponding implementation in AspectJ. This aspect can now be weaved with the system code (Java) to augment the underlying system with the desired quality attribute.

GENQA currently supports three classes of non-functional requirements: (1) monotonic addition of behavior, (2) redirection, and (3) deferred execution. We have designed GENQA to be extensible to add support for other classes of quality attributes by way of a plugin architecture. Future research in this direction will include the development and addition of new classes of quality attributes.

6. REFERENCES