Quality-Based Software Reuse

Julio Cesar Sampaio do Prado Leite[1], Yijun Yu[2], Lin Liu[3],
Eric S. K. Yu[2], John Mylopoulos[2]
[1]Departmento de Informatica, Pontifícia Universidade Católica do Rio de Janeiro, RJ 22453-900, Brasil
[2]Department of Computer Science, University of Toronto, M5S 3E4 Canada
[3]School of Software, Tsinghua University, Beijing, 100084, China

Abstract. Work in software reuse focuses on reusing artifacts. In this context, finding a reusable artifact is driven by a desired functionality. This paper proposes a change to this common view. We argue that it is possible and necessary to also look at reuse from a non-functional (quality) perspective. Combining ideas from reuse, from goal-oriented requirements, from aspect-oriented programming and quality management, we obtain a goal-driven process to enable the quality-based reusability.

1 Introduction

Software reuse has been a lofty goal for Software Engineering (SE) research and practice, as a means to reduced development costs[1] and improved quality. The past decade has seen considerable progress in fulfilling this goal, both with respect to research ideas and industrial practices (e.g., [1–3]).

Current reuse techniques focus on the reuse of software artifacts on the basis of desired functionality. However, non-functional properties (qualities) of a software system are also crucial. Systems fail because of inadequate performance, security, reliability, usability, or precision, to name a few. Quality concerns, therefore, should also be front and centre in methods for software reuse. For example, in designing for the NASA Mars Spirit spacecraft, one would not adopt a "cosine" function from an arbitrary mathematical library. Instead, one must look for, and possibly adopt, a reusable component that meets stringent requirements in precision, performance and reliability.

Despite this practical need, few methods for reuse have focused on non-functional requirements (NFRs). The typical object of software reuse as surveyed in [1], is an artifact, initially executable code, and more recently large-scale components, software architectures, designs, frameworks, and software product lines. All of these are predominantly reused on the basis of functionality. One will not find precision, performance or reliability as components ready-made for reuse. Needless to say, it will be invaluable to reuse the knowledge about these critical requirements accumulated from the design of software for other spacecrafts, or from other domains.

Why is it hard to incorporate quality requirements into reuse methods? One important reason for this is that software artifacts include both functional and quality frag-

[1] Improved software productivity and reduced development costs result from building *with* reuse; building *for* reuse actually has an overhead cost.

ments. Some of the quality fragments are hard to recognize since they are mingled with the functional fragments in order to be executable.

Our goal is to focus on qualities as reusable assets. Would it be possible to separate knowledge about how to achieve a quality, such as "performance" from a specific function, say "cosine"? Would it be possible and reasonable to look for knowledge on performance, instead of looking for different implementations of "cosine"? Would it be possible to retrieve useful knowledge relative to a concern that is applicable in several domains? This is exactly the issue that we want to address in this paper.

Unfortunately, the cross-cutting nature of quality attributes in software makes them hard to classify. Cataloging them in terms of taxonomies [4, 5] is not sufficient support for proper software quality reuse. On the other hand, in traditional function-oriented classification, quality information is not discernible in the reusable artifact. This difficulty increases when there are multiple quality concerns being dealt within one artifact, which is often the case.

To overcome these difficulties, we combine insights and techniques from research in non-functional requirements [6, 7], goal-oriented requirements engineering [8], aspect-oriented programming [9], software reuse [1] and quality management. In particular we rely on the results of combining aspect-oriented programming with goal-oriented requirements engineering [10].

This combination proves to be effective because it unites a goal refinement and classification strategy with a packing strategy provided by aspect-oriented programming, making use of well-defined relations among functional and quality fragments, we provide mechanisms for weaving those fragments together. We define a coherent process that uses an asset library to find quality characteristics and apply those to a software functional description. We show that it is possible to store qualities, retrieve it for reuse, specialize it for different contexts and integrate it with functional descriptions. The process works both for graphs holding implementation information, as the detailed operationalization of goals (into tasks), or for graphs without such detail. The reuse process is therefore applicable at requirements as well as implementation level.

The paper is organized as follows. Section 2 provides some background on the concepts of goal-oriented requirements, on aspect-oriented programming, and on reuse. Section 3 reviews the obstacles to quality reusability, presents the concept of goal aspects, and proposes a language to support software quality reuse. Section 4 describes the overall process we foresee for automating part of the quality reuse process. Section 5 illustrates the process by demonstrating how the quality Usability can be reused. We conclude in Section 6 by noting the contributions and limitations of this work, positioning it with respect to the literature, and discuss future directions.

2 Goals, Aspects and Reuse

As stated in the introduction, we are using insights from different areas of research in software engineering as well as in management. Our aim is to provide means to make it possible to reuse software quality.

The starting point of our approach is the work on non-functional requirements [7] (NFRs) that treats them as softgoals in a goal dependency graph. This graph depicts interactions among goals where one goal can influence positively or negatively other goals. The similarity among NFRs and the concepts of aspect-oriented programming has been discussed in the literature and was exploited in [10]. Since both lines of work structure software with respect to qualities, they were central to our approach which uses goal graphs as a medium to organize software in relation to qualities, and in relation to functionality.

Goals A goal represents a stakeholder intention. A goal can be either fulfilled or not [11], and may depend on sub-goals through AND or OR refinements. Goal-oriented requirements engineering [8, 11] focuses on goals which are "roughly speaking, precursors of requirements" [12]. Some goal-based modeling approaches, such as i* [13, 14], also model the actors who hold these intentions.

Most variations of goal models in the literature use AND/OR trees to represent goal decomposition [6, 11] and define a space of alternative solutions to the problem of satisfying a root-level goal. There are several proposals for goal analysis techniques. For example, obstacle analysis [15] explores possible obstacles to the satisfaction of a goal. Along a different dimension, qualitative goal analysis [16] allows qualitative contributions from one goal to another, and shows how to formalize and reason with them. In whatever form, goal-oriented requirements engineering has been attracting considerable attention within the software engineering community [17–20].

In [6], the concept of a softgoal has been proposed as a means for modeling and analyzing NFRs. Softgoals, unlike goals, can be partially satisfied or denied, and may depend on other goals and softgoals through Make($++$), Help($+$), Hurt($-$), and Break($--$) relations, also known as *contribution links* [6]. With goal models, software development proceeds by refining goals, identifying collections of leaf goals that together fulfill root-level goals, and assigning responsibilities for the fulfillment of leaf-level goals. Figure 1 provides an example of a goal model. In this paper, we use an ellipse, a rectangle and a cloud to represent a goal, a task, and a softgoal (quality) respectively. Each node has a type and a topic. A type describes a generic function or a generic NFR (a quality attribute). A topic, denoted in between "[" and "]", describes contextual information. For instance, the goal "contact" refers to a "friend", the softgoal "reliable" refers to "reply".

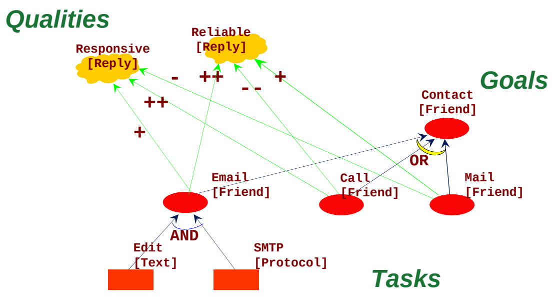

Fig. 1. A requirements goal model.

Aspects Factoring or factorization involves the decomposition of an object into a structure of smaller objects, or factors, which when combined together give the original. For example, the number 15 factors into primes as 3×5; and the polynomial $x^2 - 1$ factors as $(x - 1)(x + 1)$. The principle has been echoed in the software refactoring community [21] where refactoring is the process of rewriting material to improve its readability or structure, while preserving its meaning or behavior. For example, refactoring $x^2 - 1$ as $(x + 1)(x - 1)$, reveals an internal structure that was previously not visible (such as the two roots of the polynomial at +1 and -1). Similarly, in software refactoring, the change in visible structure can often reveal the "hidden" internal structure of the original code. Extracting commonalities can also simplify the representation of potentially complex artifacts, e.g. re-expressing $20 + 20 + 20$ as $(1 + 1 + 1) \times 20$ or 3×20 .

Aspect-Oriented Programming (AOP) [9, 22, 23] gives a different perspective to the factoring principle. Figure 2 illustrates how AOP is related to, and yet different from Structured Design [24]. The commonality may be modularized or refactored into a module C to avoid duplication. In Structured Design it is the responsibility of the user of a function to hold the address/name of the called module, whereas in AOP the responsibility of knowing where an aspect is needed relies on the aspect. One of the goals in structured design is to increase the fan-in of a module, a motivation shared with reuse.

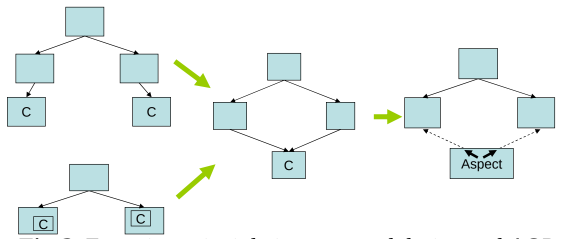

Fig. 2. Factoring principle in structured design and AOP

An aspect [9] names the address of where it is needed as a pointcut. Pointcuts are not absolute addresses; they are virtual ones, making it possible that an aspect be applied in several places where the same conditions apply. An aspect keeps the information of what it does, as an advice. Through separation of crosscutting concerns, aspect-oriented languages offer simpler and more readable code structures. In order to execute the factored code, aspect-oriented environments use a reverse process known as weaving.

It is important to stress that the factoring principle as implemented in structured design and in aspect-oriented programming is to help reuse by consolidating similar information in just one place, thus making it easy to store and retrieve information.

An example aspect expressed in AspectJ syntax is as follows:

```
aspect DisplayUpdating {
  pointcut move(): call(void FigureElement.moveBy(int, int)) ||
   call(void Line.setP1(Point)) || call(void Line.setP2(Point)) ||
   call(void Point.setX(int))   || call(void Point.setY(int));
  after() returning: move() { Display.update(); } }
```

The aspect `DisplayUpdating` includes the advice `Display.update()` that will be weaved into the component code after the `move()` pointcut. A pointcut is a virtual address for the inclusion of the advice in a component. This virtual address is resolved through matching. For example, every time a `Line.setP1(Point)` appears in a component, the advice, `Display.update()` will be weaved in that component.

Reuse Aspects that crosscut different parts of the system arise likely to address global concerns of quality attributes represented by softgoals in the requirements goal model. The link among softgoals and aspects brings the possibility of using these concepts as basic entities to represent and organize qualities. On top of that we use a framework from quality management to better organize qualities.

We are anchoring our understanding of software reusability in Krueger's taxonomy for software reuse processes [1]. Krueger lists five key processes that should happen for a software artifact to be reused: classification, abstraction, selection, specialization and integration. Classification organizes the stored information to help future queries and updates, both by those who build for reuse, and by those who build with reuse. Abstraction helps understandability by hiding low-level details and implementation. Selection is the process where the actor building with reuse chooses what to reuse from the available reusable artifacts. Specialization is necessary in white box reuse, where an artifact needs to be changed to become reusable. To contrast, in black box reuse, an artifact is used as is. Finally, integration is necessary to make the artifact being reused fit into the context where it is going to operate.

Although this process taxonomy is primarily concerned with functional reuse, we will use it to highlight the obstacles facing the reuse of qualities next.

3 A Goal- and Aspect- Driven Representation

This section outlines the key challenges when attempting to reuse qualities. We conclude that a better representation language is needed to achieve an effective reuse process. The primary insight is that a goal based representation allows qualities to be formally related to functional tasks through softgoal refinements and operationalizations.

3.1 Obstacles to Reusing Software Qualities

As we have noted before, cataloging quality requirements as taxonomies are not yet sufficient to support proper quality reuse as it is not clear about which functionalities are bound to the quality concerns. On the other hand, software representation languages are known to lack non-functional concepts [12], which makes NFRs hard to be traced in the different representations used along the software construction process. As quality concerns impact both high-level architectural changes and low-level code changes, they create difficulties in reuse when different levels of abstraction are related.

The selection of a particular incarnation of a given NFR is possible only if there is a way of linking the different incarnations with the required NFR. Since these incarnations are embedded in functional implementations, we also need to know how much these implementations satisfice [2] the given NFR. As such selection, from the non-functional perspective is a problem if the proper linkage and correlations among functions and qualities are not bound together.

[2] Herbert Simon [25] used the term *satisfice* to denote the idea of "good enough" solutions to an untractable problem. The NFR framework [7] is founded on the premise that NFRs (softgoals) are "satisficed" when they admit a partial, but good enough solution.

We do not see an easy way for black-box reuse in the context of quality reusability. The key aspect in reusing qualities is how the selected instance of a given quality will be specialized into a new context. Specialization of a quality concern is hard, mainly due to its cross-cutting characteristic.

Last but not least, integrating a quality concern that was selected and specialized is another obstacle. The need for well-defined interfaces among the reusable and the new context is more complex than when dealing with functional concerns only.

3.2 Goal Aspects

Goal aspects were proposed in [10] to relate goal models representing functional requirements to softgoal models representing NFRs. Goals, softgoals and tasks are related by means of a V-graph, which is a graph with an overall shape of the letter V representing the three types of nodes (Figure 3). The top two vertices of the V represent respectively functional and non-functional requirements in terms of goal models. Following [7] we represent NFRs in terms of softgoals, i.e., goals with no clear-cut satisfaction. Both models are AND/OR trees with lateral correlation links. The bottom vertex of the "V" represents a set of tasks that contribute to the satisfaction of both goals and softgoals.

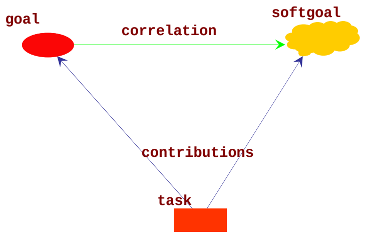

Fig. 3. A V-shape goal model

A systematic requirements engineering process [10] uses the V-graph to elicit aspects, as in AOP terminology. We call these aspects, goal aspects, since they simplify the V-graph by removing the correlation links and putting functional and non-functional issues into separate AND/OR decomposition hierarchies. We have used it in the Media Shop case study [26]. The advantage of having a systematic process for discovering goal aspects is that finding them early on, makes it easier to trace quality concerns to aspect-oriented implementations. Although the V-graph representation helps the traceability of requirements and as such helps the processes of integration, specialization and selection, it does not fully support the classification and abstraction processes that are necessary for reusability. Next, we detail our proposal for a goal-oriented representation language to support quality reusability.

3.3 Q7: A Language for Organizing Qualities

As we have seen before, one of the key challenges in quality reusability is the multidimensional characteristic of quality issues. Classification of quality requirements and abstraction mechanisms to deal with them are obstacles to be overcome. These would require a language that could handle not only the characteristics of the quality knowledge, but that could relate those with functional descriptions as well. As such, we would need proper representation for the following concepts: functions, topics, quality types, pre-conditions, pointcuts (relations among functions, topics and quality types), contribution structures and quality operationalizations.

The source of inspiration for coming up with the abstract language was an analogy involving natural science and automobile design. In designing a sports car, a dominant quality to strive for is speed. If we think about speed in the context of marine life, we will observe that the fastest swimming animals have a common streamline shape. Further we will recognize that the streamline shape is manifested at different parts of the animal: the tail, the body and the front. If we based our automobile design on this concept we would need a car that would have special attention to the shapes of the rear part, the body and the front part. Although there is a huge gap in "reusing" this shape information, the analogy helped us in understanding that to locate a quality issue we would need to know why we need it, where it is applicable, and how to implement it. So, in the car as in the fish, when we need speed, the quality of speed needs to be applied to different parts of the fish or car, which when operationalized, are implemented as streamline shapes.

We could paraphrase the above as: having a reason (that is why), a place to apply the reason (where), and the details of the implementation to attain the reason(which is how). Once we made this connection, it came to us that the structure of the 5W2H used in quality management could be useful in classifying qualities.

Let's see how the 5W2H fits into our context of quality reusability by examining each of the 7 questions (also known as Q7).

– *Why?* This question is central to a quality view; it addresses intentionality and focuses on the rationale of an intention. Non-functional requirements was initially proposed to describe quality attributes [6] to answer "Why an artifact needs a quality attribute?". So the "why" question refers to the soft-goal or the quality information we want to reuse. In the NFR framework this is also known as "type".
– *Who?* The "who" characterizes the main target of the quality attribute. In our analogy the fish and the car would be the target or the artifact to receive the speed attribute. So in our case, the "who" is representing the artifact associated with the soft-goal or the quality we would like to attain.
– *What?* The "what" characterizes contextual information of a given "who", that will be the target of a quality attribute ("why"). It is a necessary triggering characteristic that the artifact must have to reuse the software quality. In the NFR framework this is also known as "topic".
– *Where?* The "where" is the specific addresses of the quality concern in the artifact. In our V-graph goal model it is the pointcuts where the goal aspects will point to. This address is discovered by examining the correlations, in the NFR framework sense, found in a V-graph. "Where" is exactly the point in the V-graph that a goal aspect ("why") will be weaved. To be applied to this point the goal aspect has to comply with the "who" and "what" related to the reuse task at hand.
– *When?* The "when" is used to indicate a pre-condition that needs to hold before the operationalization ("how") could be applied in a given pointcut ("where"). In the NFR framework is also known as a "claim".

– *How?* This question addresses the refinement of the quality concern into a functional description. In the NFR framework this is known as the operationalization of a NFR [6]. It is how the NFRs will be implemented. In our model it will be the advices in our goal aspects.
– *How much?* The impacts and side-effects of applying the operationalizations ("how") to the artifact. In the NFR framework it is the set of contributions links that relate the operationalizations with NFRs. Impacts can be implicit when they relate the operationalization and its parent softgoal.

Table 1 summarizes what is listed above and gives an example of the questions for the Media Shop case study.

Table 1. Classifying the NFRs knowledge, such as the "Usability" aspect in Media Shop

artifacts	quality topics	quality types	claims	pointcuts	operationalizations	contributions
Who	What	Why	When	Where	How	How much
MediaShop	interface	usability	lang.	conventions	communicative	-productivity
MediaShop	interface	usability	operations	memorizability	operability	-productivity
MediaShop	interface	usability	usage	always	training	-productivity
MediaShop	lang.	communicative	words	natural lang.	lang. customization	-productivity
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Based on the above intuitions, we define the BNF grammar for the Q7 language, which organizes the knowledge for the purpose of software quality reuse.

```
START := Advice*
Advice := [When] [Who] Why [What] [Where] [How] [HowMuch]
When:= "(" Expr ")" "=>"
Who:= "<" id ">" "::"
Why:= id
What := "[" id { "," id}* "]"
Where:= "<=" Pointcut { "," Pointcut }*
How:='{' BoolOp Advice* '}'
HowMuch:= "=>" Effect { "," Effect }*
Expr:= "true" | "false" | id | Expr BoolOp Expr
Effect:= HowMuchOp [ Who "::" ] Why [ "[" What "]" ]
Pointcut:= HowMuchOp [("*"|Who) "::"] ("*" | Why) [ ("[" "*" "]" | What ) ]
BoolOp := "&" | "|"
HowMuchOp:="++"|"+"|"-"|"--"
```

We have designed a parser to convert a Q7 program into the Telos knowledge representation in our OpenOME tool.

Next we describe how the Q7 language is central to our process for reusing qualities. If we look at Q7 just from the point of view of an organization scheme, it may look similar to a set of fixed facets. Faceted classification was proposed by Prieto-Diaz [27] to better organize a library of software components, where each component would have a description written as a set of facets. Although the facets may be defined at will, usually the examples shown in the literature did focus on the functional part of the components. Q7 goes beyond facets, by providing specific relationships among functional and quality concerns using an AND/OR graph as the basic representation scheme.

4 A Process for Quality-based Reuse

Having framed the obstacles found in quality reuse in terms of the 5W2H framework, we now present a partially automated process to support quality reuse.

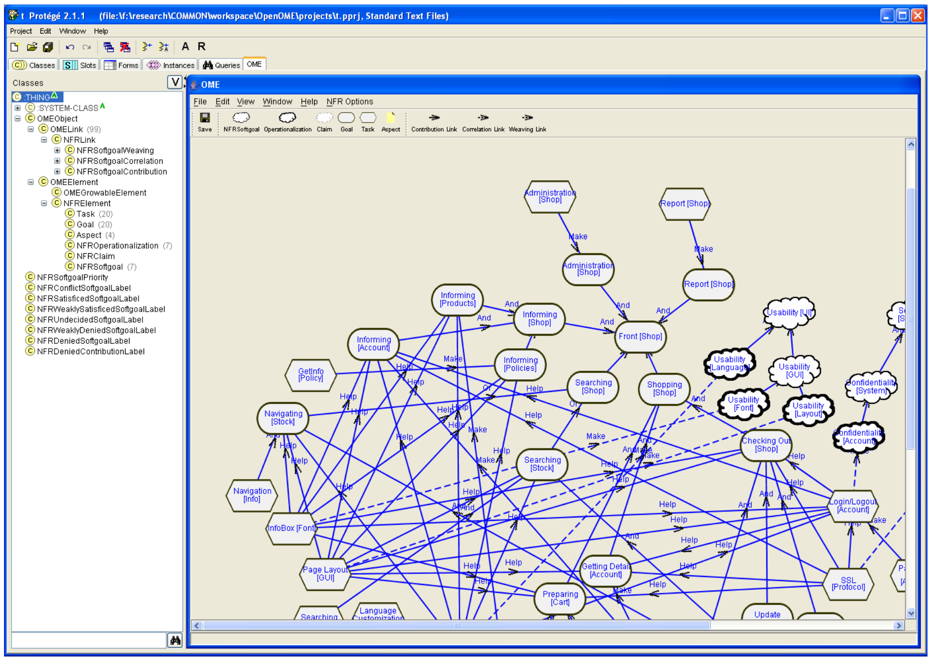

Fig. 4. The Media Shop build for reuse in OpenOME: the asset library is shown as the Protégé ontology to the left; the V-graph is viewed to the right.

Figure 5 shows that our process is centered around the idea of an asset library. We have implemented this library by using a knowledge base approach with Protégé [28] on top of OWL [29, 30]. We have developed an OME/Telos (a tool for modeling NFRs) plugin for the Protégé 2.1.1 (Figure 4), which is capable of populating (TELL) and querying(ASK) the asset library. Figure 5 shows on the left hand side the input to the asset library – V-graphs for software systems (products) containing both functional and non-functional information, written in Q7. This description was produced from the point of view of building *for* reuse. The right part of the figure shows the products that are necessary in order to achieve quality reuse. We start with a general query on available assets for a given quality, retrieve a candidate for reuse, specialize the candidate by manual enumeration of pointcuts for a given functional description, and finally integrate the resulting goal aspects for the given functional description.

Abstraction and Classification The processes of abstraction and classification are tasks for building for reuse, which we do not detail here. As said before we have developed basic infrastructure to support these tasks, by means of a parser for Q7 and the integration of Protégé with OME.

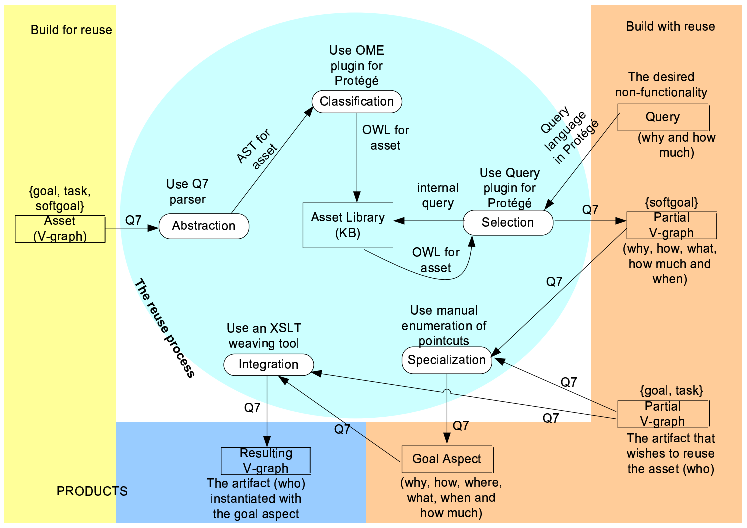
Build for reuse
Build with reuse
Use OME plugin for Protégé
Classification
AST for asset
OWL for asset
Query language in Protégé
The desired non-functionality
Query
(why and how much)
{goal, task, softgoal} Asset (V-graph)
Q7
Use Q7 parser
Abstraction
Asset Library (KB)
internal query
Use Query plugin for Protégé
Selection
{softgoal} Partial V-graph
(why, how, what, how much and when)
OWL for asset
The reuse process
Use an XSLT weaving tool
Integration
Use manual enumeration of pointcuts
Specialization
{goal, task} Partial V-graph
The artifact that wishes to reuse the asset (who)
Resulting V-graph
The artifact (who) instantiated with the goal aspect
Goal Aspect
(why, how, where, what, when and how much)
PRODUCTS

Fig. 5. A process for quality reusability

Selection The selection process is performed by the software engineer using a Protégé plug-in to query the asset library. A query is performed to retrieve a needed quality that a developer is trying to reuse. This is done with the "why" operand, that is the query will return a partial V-graph with the softgoal sub-graph. This graph may be pruned by performing queries that narrow the search using the "when" and "how much" operands. It is possible to check for preconditions on the softgoal graph by the "when" operand, and to check for the satisficing levels (contribution links) of a given sub-graph by the "how" operands, to check the effects on other quality attributes by the "how much" operands.

Specialization The process of specialization uses two V-graphs as inputs to produce the goal aspect graph for the chosen quality. The inputs are the V-graph retrieved from the selection process and the target V-graph, that is the V-graph representing the functional part where the quality has to be applied. As of now, we are using a manual inspection of both graphs to compose the resulting goal aspect graph. This task has to identify "where" in the functional representation the goal aspects must be weaved. Doing this we are looking for the "what" or topic to which the advice of the goal aspect will be weaved into. Note, that although Figure 5 does not explicitly have a feedback loop to the selection process, it occurs as we try to find a better candidate due to difficulties in the specialization process. We foresee several automation strategies to lessen the burden of a manual specialization: using matching patterns for topics in the inputs; using an automatic tool to locate bottlenecks of the artifact that needs operationalized quality improvements (e.g., using a profiling compiler that detects performance bottlenecks of the execution that needs the tuning advices); or using an external intelligent agent that performs the selection and specialization as a black-box reuse.

Integration Once we have the V-graph for the selected and specialized goal aspect, we can use an automatic procedure, similar to an AOP weaver, to integrate the quality into the functional description. Unlike AOP, the weaving here not only insert the advices before/after/around the existing functionalities, but also allows a modification of the existing tasks by more advanced semantics, such as goal satisfactions, or function-preserving transformations.

In this paper, we studied a simple form of pointcuts specialization and weaving based on the goal satisfaction at the high-level requirements [10]. We believe it is extensible to low-level code weaving using existing tools such as AOP, compiler and transformation systems. The simple weaving is done as follows. The goal aspect V-graph produced by the specialization step and the functional V-graph provided by the software engineer are encoded in Q7, which are the inputs to the weaver, as described by the following procedure:

```
for each softgoal $s$
  $P = \text{WHERE\_POINTCUTS}(s) = \text{test}(\text{WHEN\_CONDITION}(s), \text{all functional goals})$
  for each pointcut $p$ of $P$
    for each functional goal $g$ /* a candidate join point */
      if $\text{match}(\text{WHO}(g), \text{WHO}(p))$ and $\text{match}(\text{WHY}(g), \text{WHY}(p))$
        and $\text{match}(\text{WHAT}(g), \text{WHAT}(p))$ then
          $\text{weave}(\text{HOWMUCH\_OP}(p), \text{HOW}(g), \text{HOW}(s))$ end if
```

The routine `test` uses the guard condition in the “when” clause to test whether any functional goal can apply the quality advice. These functional goals will be enumerated as the pointcuts in the quality aspect. To be more useful, these pointcut expressions can use wildcards to keep the virtual addresses.

The routines `match` checks whether a hard goal g matches the specifications of a pointcut softgoal s. They match if g has the same “who”, “why” and “what” as those of the pointcut of s. Wildcard “*” in the pointcut specification can match any name.

The operation `weave` combines g and s using the pointcut operator (similar to “howmuch” operator), which is one of $++$, $+$, $-$ and $--$. First, a correlation link $g \Rightarrow$ op s is created as an obligation on the joinpoint g. To fulfill this obligation, if the operator is $++(--)$, then all the subgoals of g (how) must add the operationalized tasks (anti-tasks) of s (how). If the operator is $+$ $(-)$, then at least one of the subgoals of g must add the operationalized tasks (anti-tasks) of s. The semantic of the addition can be implemented using one of the “before”, “after”, “around” semantics in AOP.

In the next Section we show, with an example, how we have applied this process to retrieve usability from the asset library and reuse it in a different software system.

5 Reusing the Usability asset

This section uses two software systems, Media Shop and Web Based Training (WBT), in order to illustrate the feasibility of our reuse process. The goal model describing the Media Shop case study was obtained using a goal aspect discovery process [10] and the goal model describing the WBT case study was obtained from an i* model presented in [31]. Our aim is to reuse one of the qualities “usability” present in the Media Shop, and apply it to a different system – the WBT system.

For the Media Shop study we used both a requirements level description [26] and a real implementation, osCommerce [32], to trace the goals and softgoals to tasks and operationalized tasks. The goal aspect discovery process was applied on a V-graph merging the requirements level description with the recovered abstraction of the implementation. This V-graph is the asset we classified and stored in the asset library, which contains operationalizations for qualities such as security, usability, responsiveness and integrity [10]. An abstraction of the asset library is stored in the nested Q7 format. We only show the necessary parts for illustration purposes:

```
<MediaShop>::Front[Shop] { &
   Shopping[Shop] { ShoppingCart[product, item] ... }
   Informing[Shop] { ... }  Managing[Shop] { ... }
} => ++Security, ++Usability, ++Integrity, ++Responsiveness
Security[System] { ... }
Usability[UI] { & Usability[lang.] { & Communicative[Language] { &
 (NaturalLanguages) => LangCustomization[Words] <= ++<MediaShop>::ShoppingCart,
... } ... } ... } Integrity[Data] { ... } Responsiveness[Transaction] { ... }
```

We rewrote the functional part for the WBT system [31] using Q7. Below we list a partial description of the resulting goal model.

```
<WBT>::Build[System] { &
 CoursePattern[System] { |
  CoursePattern[InstructorLed] { &
   SchedulePresentation[Instructor] OptionalTopics[Learner]
  } CoursePattern[LearnerLed] { &
   ActAsLearningResource[Instructor] SetCoursePace[Learner]
 }} Collaboration[System] { |
  Collaboration[Email] Collaboration[NewsGroupForum] Collaboration
    [ChatRoom] Collaboration[SharingScreen] Collaboration[AVConf]
 } CommonLessonStructure[System] { |
   Classic[Tutorial] ActivityCentered[Lessons] LearnerCustomized[Tutorial]
   KnowledgePaced[Tutorial] Exploratory[Tutorial] Generated[Lessons]
}}
```

Given the functional description, our aim is to implement an interface for WBT that considers aspects of usability. Following the process in Figure 5, we reuse the usability asset as in our library. First we select from the asset library a softgoal hierarchy using a query (`why`=”Usability”), resulting in an aspect without pointcuts in Q7:

```
Usability[UI] { &
  (Conventions) => Communicative[Language] { &
    (NaturalLanguage) => LangCustomization[Words]
  } (Conventions) => Communicative[Custom] { &
    (Classifications) => Customization[WordsOrder]
  } (Memorizability) => Operability[Operations] { &
    (MultipleWidgets) => Similar[LookAndFeel]
    (MultipleFonts) => Stylized[Font]
    (MultipleActions) => HierarchicalMenus[Navigation]
  } Training[Usage] { &
    ProvideUserManual[UseScenarios]
    ProvideContextSensitiveHelp[Actions]
    LearnByExamples[Tutorial]
} } => -Productivity
```

For the root advice (why=“Usability”), we have (what=“UI”, when=“Conventions”, howmuch=“-Productivity”). The decomposition of the goal is nested inside the braces as detailed advices (how). We discard information stored in the asset library that is specific only to the asset. For example, (where = “ShoppingCart [product, item]”) is a goal in the (who=“MediaShop”) domain that needs language customization for usability. To reuse the Usability in the “WBT” domain, however, “ShoppingCart” is irrelevant. Therefore we would only retrieve information that can be applied to any domains, such as (when=“NaturalLanguage”).

We perform the process of specializing the reusable asset as a goal aspect by updating the pointcuts. Currently, a human agent has to manually identify pointcut functional goals in the new domain according to the “when” condition in the queried aspect. For example, in WBT, any functional goal that involves “natural language” may consider the “language customization” advice. Therefore, one may enumerate the topics “Email”, “ChatRoom”, “NewsgroupForum”, “Tutorial” and “Lessons” into the pointcut:

```
<WBT>::Usability[UI] { &
 (Conventions) => +Communicative[Language] { &
  (NaturalLanguage) => LangCustomization[Words] <= ++*[Email],
     ++*[ChatRoom], ++*[NewsgroupForum], ++*[Tutorial], ++*[Lessons]
}  ... /* omitted */ } => -Productivity
```

The integration process is performed automatically and the resulting product is a Q7 description of WBT weaved with the goal aspect of Usability. According to the Usability goal aspect, the operationalized task “LangCustomization[Words]” is only weaved with the functional goals that match the pointcut.

```
<WBT>::Build[System] { &
  CoursePattern[System] { |
    CoursePattern[InstructorLed] { &
      SchedulePresentation[Instructor] OptionalTopics[Learner]
    } CoursePattern[LearnerLed] { &
      ActAsLearningResource[Instructor] SetCoursePace[Learner]
  } } Collaboration[System] { |
      Collaboration[Email] => ++LangCustomization[Words]
      Collaboration[NewsGroupForum] => ++LangCustomization[Words]
      Collaboration[ChatRoom] => ++LangCustomization[Words]
      Collaboration[SharingScreen] Collaboration[AVConf]
  }  CommonLessonStructure[System] { |
      Classic[Tutorial] => ++LangCustomization[Words] }
      ActivityCentered[Lessons] => ++LangCustomization[Words]
      LearnerCustomized[Tutorial] => ++LangCustomization[Words]
      KnowledgePaced[Tutorial] => ++LangCustomization[Words]
      Exploratory[Tutorial] => ++LangCustomization[Words]
      Generated[Lessons] => ++LangCustomization[Words]
} }  => -Productivity
```

Since the V-graph of Media Shop also had the actual implemented goal aspects (given by osCommerce as explained in [32]) the final reuse will be the reuse of the code that implements the goal aspect, that is the desired quality. As such we rely on an operational semantics as to validate our final result. The fitness of the integration will depend on the quality of the specialization that was performed. Note that, by merging the two graphs, the semantics of the composed parts are preserved.

6 Conclusions

We have presented a method for making quality requirements a prominent dimension in software reuse. It is based on combining results from different research directions: requirements reuse and aspect-oriented programming

There are still several problems to be addressed, both from the point of view of supporting mechanisms as well as the feasibility of dealing with a large number of assets. The problem of scalability, dealing with huge graphs, is not itself the prime concern here, but how different strategies for partitioning the result of selection queries would be handled. It is also not clear where the strategy may break and how it will deal with very general quality concerns, for instance reusability. However this problem is general and also applies to the AOP view. Further research and experiments are needed.

As stated up front, reusing qualities is not an issue that received much attention in the literature. Two works, from different perspectives did approach the issue indirectly. One, [33], deals with the problem from the perspective of design patterns, while the other, [34], from the perspective of aspects. While Clarke and Walker [34] focus on parameterizing aspects to make them more flexible, Gross and Yu [33] propose to explicitly deal with quality concerns in design patterns and, as such, propose an explicit encoding of the intentionality for each pattern. In our proposal we provide a broader view of the problem and address all the five software reuse key processes.

References

1. Krueger, C.: Software reuse. ACM Computer Survey **24** (1992) 131–183
2. Prieto-Diaz, R.: Status report: Software reusability. IEEE Software **10** (1993) 61–66
3. van Vliet, H.: Software Engineering: principles and practice, 2nd Ed. John Wiley (2000)
4. Sommerville, I.: Software Engineering, 4th Ed. Addison-Wesley (1992)
5. Boehm, B.W., Brown, J.R., Lipow, M.: Quantitative evaluation of software quality. In: ICSE, International Conference on Software Engineering, IEEE Computer Society Press (1976) 592–605
6. Mylopoulos, J., Chung, L., Nixon, B.: Representing and using nonfunctional requirements: A process-oriented approach. IEEE Trans. Softw. Eng. **18** (1992) 483–497
7. Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishing (2000)
8. Mylopoulos, J., Chung, L., Yu, E.: From object-oriented to goal-oriented requirements analysis. CACM **42** (1999) 31–37
9. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect oriented programming. LNCS **1241** (1997) 220–242
10. Yu, Y., do Prado Leite, J.C.S., Mylopoulos, J.: From goals to aspects: Discovering aspects from goal models. In: RE 2004 International Conference on Requirements Engineering, IEEE Computer Society Press (2004) 38–47
11. van Lamsweerde, A.: Goal-oriented requirements engineering: From system objectives to UML models to precise software specifications. In: ICSE 2003. International Conference on Software Engineering, IEEE Computer Society Press (2003) 744–745
12. Feather, M.S., Menzies, T., Connelly, J.R.: Relating practitioner needs to research activities. In: RE 2003. International Conference on Requirements Engineering, IEEE Computer Society Press (2003) 352–361

13. Yu, E.S.K., Mylopoulos, J.: From E-R to A-R – modelling strategic actor relationships for business process reengineering. Int. Journal of Intelligent and Cooperative Information Systems **4** (1995) 125–144
14. Liu, L., Yu, E., Mylopoulos, J.: Security and privacy requirements analysis within a social setting. In: RE 2003. International Conference on Requirements Engineering, IEEE Computer Society Press (2003) 151–161
15. van Lamsweerde, A., Letier, E.: Handling obstacles in goal-oriented requirements engineering. IEEE Trans. Softw. Eng. **26** (2000) 978–1005
16. Giorgini, P., Mylopoulos, J., Nicchiarelli, E., Sebastiani, R.: Reasoning with goal models. LNCS **2503** (2002) 167–181
17. Anton, A.I., Carter, R.A., Dagnino, A., Dempster, J.H., Siege, D.F.: Deriving goals from a use-case based requirements specification. Requirement Engineering **6** (2001) 63–73
18. Rolland, C., Prakash, N.: From conceptual modelling to requirements engineering. Annals of Software Engineering **10** (2000) 151–176
19. Kaiya, H., Horai, H., Saeki, M.: Agora: Attributed goal-oriented requirements analysis method. In: RE 2002. International Conference on Requirements Engineering, IEEE Computer Society Press (2002) 13–22
20. Bolchini, D., Paolini, P., Randazzo, G.: Adding hypermedia requirements to goal-driven analysis. In: RE 2003. International Conference on Requirements Engineering, IEEE Computer Society Press (2003) 127–137
21. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
22. Murphy, G.C., Walker, R.J., Baniassad, E.L.A., Robillard, M.P., Lai, A., Kersten, M.A.K.: Does aspect-oriented programming work? CACM **44** (2001) 75–77
23. Robillard, M.P., Murphy, G.C.: Concern graphs: finding and describing concerns using structural program dependencies. In: Proceedings of the 24th International Conference on Software Engineering (ICSE-02), New York, ACM Press (2002) 406–416
24. Yourdon, E., Constantine, L.L.: Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, 1st ed. Prentice-Hall (1979)
25. Simon, H.A.: The Science of the Artificial, 3rd Edition. MIT Press (1996)
26. Castro, J., Kolp, M., Mylopoulos, J.: Towards requirements-driven information systems engineering: the tropos project. Information Systems **27** (2002) 365–389
27. Diaz, R.P.: Implementing faceted classification for software reuse. Commun. ACM **34** (1991) 88–97
28. Noy, N.F., Sintek, M., Decker, S., Crubezy, M., Fergerson, R.W., Musen, M.A.: Creating semantic web contents with Protege-2000. IEEE Intelligent Systems **16** (2001) 60–71
29. W3C: Web ontology language, http://www.w3.org/2004/owl (2004)
30. Fensel, D., van Harmelen, F., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F.: OIL: an ontology infrastructure for the semantic web. IEEE Intelligent Systems **16** (2001) 38–45
31. Liu, L., Yu, E.: Design web-based systems in social context: A goal and scenario based approach. In: CAiSE 2002. Volume 2348., Springer-Verlag (2002) 37–51
32. : (Open Source E-Commerce Solutions, http://www.oscommerce.com)
33. Gross, D., Yu, E.S.K.: From Non-Functional Requirements to Design through Patterns. Requirements Engineering **6** (2001) 18–36
34. Clarke, S., Walker, R.J.: Composition patterns: An approach to designing reusable aspects. In: ICSE 2001. International Conference on Software Engineering, IEEE Computer Society Press (2001) 5–14