1
Introdução

Neste trabalho, apresentamos um processo para desenvolvimento de software mais transparentes. As contribuições deste processo foram anexadas a um catálogo, a fim de facilitar seu reuso. Este capítulo descreve a motivação que levou a realização deste trabalho, caracteriza o problema da construção de software transparente e como ele se relaciona com problemas clássicos da engenharia de software, detalha o enfoque da solução proposta e define a organização da tese.

1.1.
Motivação

Transparência é um conceito difuso, que tem sido utilizado em uma grande variedade de âmbitos, como social, econômico e político. Recentemente, este conceito tem sido explorado também no contexto de software (Leite e Cappelli, 2010). Um software transparente é aquele que informa sobre si mesmo, disponibilizando informação sobre o que faz, como e porque o faz. A complexidade inerente ao termo transparência é muito grande em qualquer contexto, e o de software não é uma exceção. Com o propósito de lidar de forma coerente com tal complexidade, utilizou-se a abordagem proposta em (Cappelli, 2009), instanciada para o tópico software. Tal abordagem propõe a divisão da transparência em diversos atributos de qualidade, modelados como metas flexíveis inter-relacionadas, através do *Software Interdenpendency Graph* (SIG), proposto no NFR framework (Chung et al., 2000). A partir desta modelagem, criou-se o Catálogo de Transparência de Software (CTS) (Catálogo, 2013), cujo objetivo é organizar as operacionalizações relacionadas às metas flexíveis presentes no SIG. A variabilidade deste catálogo é proporcionada por alternativas para responder as questões relacionadas às metas flexíveis.

Desenvolver sob os princípios de transparência, de forma geral, é benéfico devido à abertura de informações. A abertura, ideia básica por trás do movimento de Software Livre, é o pilar de sustentação da Lei de Linus[1] - "*given*

[1] Raymond, Eric S. (1999). The cathedral and the Bazaar. O'Reily Media. p.30. ISBN 1-56592-724-9.

enough eyeballs, all bugs are shallow". Não obstante, os princípios de transparência também têm um impacto direto em outros problemas clássicos da Engenharia de Software, como a rastreabilidade entre artefatos de um software, o entendimento de seu código fonte e a definição de critérios para sua organização modular. Isto significa que muitas das características elencadas no SIG já eram desejadas para o software, porém não apareciam sob o conceito de transparência. Esta relação ainda não foi totalmente explorada, mas sabemos, analisando as questões do CTS, que há uma interseção entre as características de transparência e desafios já conhecidos da Engenharia de Software. Neste trabalho, focaremos em três conceitos que fazem parte desta interseção: detalhamento do código fonte, modularização e rastreabilidade.

O detalhamento é uma das metas flexíveis presentes no SIG de transparência. Aplicado ao código fonte do software, esta qualidade pode facilitar a sua leitura por aqueles que não estiveram envolvidos diretamente em sua escrita, expandindo o número de pessoas que o compreendem. Utilizando o mesmo raciocínio que justifica a programação por pares na metodologia XP (Beck e Andres, 2004), a Lei de Linus ou o uso de pontos de vista (Leite e Freeman, 1991), podemos argumentar que a qualidade irá aumentar, pois mais pessoas terão a capacidade de compreender o que ele faz. Esta crença também é confirmada pelo estudo realizado por (Rahman e Devanbu, 2011), onde ficou claro que o código escrito por vários autores apresentava menos erros que aqueles escritos por apenas um. Uma das atividades impactada pelo entendimento do código é a manutenção do software. Segundo (Mayrhauser e Vans, 1995), o conhecimento do código influencia diretamente a eficácia da manutenção e evolução do software. Além disto, a sua compreensão também influencia a atividade de reuso. Segundo (Marshall, 1999), quando o programador encontra o código que deseja reusar, ele pode precisar modificar parte dele, para adaptá-lo a sua necessidade. Ainda que a modificação não seja necessária, ele precisa importá-lo para o software em desenvolvimento. De qualquer forma, ambas as atividades exigem que o programador entenda o código em questão.

O conceito de modularização, embora não esteja explicitamente representado no SIG de transparência, pode ser percebido nas metas flexíveis dependência e divisibilidade (Catálogo, 2013). Entendemos como modularização a disciplina de divisão do software em módulos, que são unidades básicas do software com interface bem definida. Uma boa estrutura modular é aquela que apresenta um alto grau de coesão e baixo grau de acoplamento (Yourdon e

Constantine, 1979; Pressman, 1992; Sommerville, 2001). Acredita-se que um software com uma boa modularização seja mais fácil de desenvolver e manter (Pressman, 1992; Sommerville, 2001). Segundo Parnas, (Parnas, 1972) os benefícios de uma modularização bem feita são: (i) tempo de desenvolvimento reduzido, (ii) flexibilidade do produto e (iii) facilidade de compreensão. A primeira vantagem se deve a possibilidade de paralelizar o desenvolvimento, uma vez que os módulos são independentes. A segunda se deve a possibilidade de alterar uma parte do software, sem que outras sejam afetadas. A última é decorrente da possibilidade de compreender uma parte do software de cada vez. Existem muitos trabalhos com o objetivo de obter uma organização modular, a partir de software já construído (Kiarash et al., 2003; Harman et al., 2005; Praditwong et al., 2011), porém, poucos lidam com os critérios para escolha dos módulos no início do desenvolvimento de um novo software (Parnas, 1972).

A rastreabilidade é outra meta flexível que pertence ao SIG de transparência. Este conceito é central na construção de software, especialmente a ligação entre arquitetura e código e requisitos e arquitetura (De Silva e Balasubramaniam, 2012). A arquitetura de um software é uma abstração, que permite uma visão geral de sua estrutura. Ela oculta os detalhes de implementação e estrutura de dados, permitindo o foco na descrição de elementos importantes, suas propriedades visíveis externamente e o relacionamento entre eles (Bass e Kazman, 2003). Linguagens formais para descrição de arquitetura (ADLs) (Medvidovic e Taylor, 2000), são ferramentas projetadas para ajudar na representação e análise destas estruturas de alto-nível. Embora o desenvolvimento de ADLs esteja avançando, assegurar que a arquitetura do software descreve o que foi implementado, e manter esta arquitetura atualizada, ainda são problemas centrais no processo de desenvolvimento de software (Aldrich et al., 2002). Como dito por Shaw (Shaw e Clements, 2006), duas áreas promissoras no campo da arquitetura de software são: estabelecer um vínculo entre arquitetura e código e investigar o relacionamento entre decisões arquiteturais e atributos de qualidade. O primeiro é importante, pois sem a presença de um elo entre arquitetura e código, ela estará destinada a obsolescência. A segunda é crucial para determinar o impacto dos atributos de qualidade no processo de desenvolvimento de software, de forma que seja possível entregar o software o mais próximo possível das necessidades do usuário. Segundo Garlan (Garlan, 2003), embora existam trabalhos na área de geração automática de código a partir de modelos

arquiteturais, assegurando a conformidade por construção, muito ainda precisa ser feito.

Os esforços para manter a integridade entre código e arquitetura, não devem ser limitados a apenas uma direção, isto é, apenas da arquitetura para o código ou do código para a arquitetura. De acordo com Gurp (Gurp e Bosch, 2002) a erosão da arquitetura, isto é, a diferença do que foi modelado e o que está realmente implementado, é uma condição inevitável, devido a evolução dos requisitos ao longo do tempo, que é uma característica inerente da grande maioria dos software. Portanto, os esforços devem ser concentrados em tratar o sintoma, ao invés de tentar evitá-lo.

Podemos separar os problemas entre arquitetura e implementação em três categorias (Ubayashi et al., 2010): (i) nível de abstração do desenho da arquitetura, (ii) refinamento do desenho para o código e (iii) coevolução entre desenho e código. De acordo com Ubayashi, estes problemas existem, pois não há uma definição explícita da fronteira entre desenho e implementação. O primeiro problema está relacionado ao nível de abstração apropriado para a arquitetura. Um nível muito alto aumenta a diferença entre arquitetura e implementação. Por outro lado, se o nível de abstração for muito baixo, a diferença entre arquitetura e implementação se torna muito sutil. O segundo se refere à correlação das decisões de desenho, com o seu respectivo impacto no código. O último trata da integridade entre código fonte e arquitetura. Para manter esta integridade, é necessária uma rastreabilidade bidirecional (Moriconi et al., 1995; Aldrich, 2002; Buchgeher e Weinreich, 2009; Christensen e Hansen, 2011; Zheng e Taylor, 2011).

Observando a discussão sobre os problemas apresentados, é notória a relevância de abordá-los durante o processo de desenvolvimento do software. Fica claro também, o relacionamento entre algumas características desejadas para um software transparente, e dificuldades há muito discutidas no contexto da engenharia de software. Apesar da ampla discussão, ainda é possível contribuir para amenizá-las, pois nenhuma solução definitiva foi apresentada. O vínculo entre a transparência de software e estes problemas, torna a tarefa de elaborar as soluções ainda mais recompensadora, pois desta forma também estamos colaborando para tornar o software mais transparente.

1.2. Caracterização do Problema

A transparência de software é um conceito complexo e, para ser melhor compreendido, foi descrito através de uma rede de metas flexíveis. Para que seja possível construir software utilizando princípios de transparência, é preciso encontrar operacionalizações para estas metas flexíveis. A forma encontrada para permitir a associação de operacionalizações às metas flexíveis da rede foi a construção de um catálogo, cuja variabilidade, isto é, seu ponto de extensão, são respostas as questões contidas em sua estrutura. Algumas questões do Catálogo de Transparência de Software (CTS) remetem a desafios conhecidos da engenharia de software, que ainda não foram solucionados a contento. Dentre estes desafios, temos o de facilitar o entendimento do código, definir critérios para modularização do software e criar/manter a rastreabilidade entre os artefatos que compõem o software.

A melhor forma de lidar com estes desafios é incorporar ao processo de construção do software meios de atenuá-los. É importante que estes desafios sejam abordados durante o processo de desenvolvimento, pois sabemos que alterações em software em produção são muito mais custosas do que aquelas realizadas durante seu desenvolvimento (Bennett e Rajlich, 2000).

Diante do exposto acima, em relação à operacionalização de metas flexíveis específicas da transparência de software, nos vemos diante dos seguintes desafios:

- Partindo de artefatos de requisitos em linguagem natural, como obter a organização modular da arquitetura do software, de modo que se mantenha a rastreabilidade entre os requisitos e a arquitetura?

- Como utilizar o mesmo artefato de requisitos, integrado com os diferentes artefatos produzidos ao longo do desenvolvimento do software, de modo a obter uma rastreabilidade, em ambas as direções, dos requisitos até o código?

- Como facilitar a compreensão de um software, permitindo uma explicação detalhada de sua estrutura e código, através da utilização de linguagem natural semi-estruturada, de forma que sejam entendidos com esforço reduzido?

- Como incorporar as operacionalizações para rastreabilidade, dependência e detalhamento no Catálogo de Transparência de Software (CTS), a fim de permitir o seu reuso orientado a qualidade?

1.3. Enfoque da Solução

Neste trabalho, propomos um processo de construção de software (PDS+T) (Almentero, 2013b), que incorpora em suas etapas procedimentos específicos para lidar com as questões enunciadas na seção anterior. Este processo utiliza como alicerces a representação de requisitos através de cenários (Leite et al., 1997) e a linguagem do Universo de Informação (UdI), descrita através da técnica Léxico Ampliado da Linguagem (LAL) (Leite e Franco, 1993). Restringimos o escopo do processo a uma categoria específica de software, que são aqueles desenvolvidos para Web. Isto foi feito para que durante o processo, pudéssemos adotar heurísticas específicas aos paradigmas (Web), como, por exemplo, interface construída através da linguagem de marcação HTML, envolvidos na construção do software.

Propomos um processo inteiramente novo para construção de software, pois vislumbramos a oportunidade de aproveitar características específicas das técnicas de requisitos LAL e cenários, a fim de agregar características de transparência ao software, durante seu desenvolvimento. Não encontramos nenhuma outra estratégia que utiliza estas técnicas com este fim. As características que nos interessaram são: (i) o princípio da circularidade do LAL, que permite identificar o relacionamento entre os símbolos do domínio, atributo interessante para abordar a qualidade de dependência, (ii) o modelo semi-estruturado dos cenários, que já foi utilizado com sucesso na descrição de artefatos do software (Silva et al., 2003; Almentero, 2009) e demonstrou ser uma característica interessante para qualidade de detalhamento e (iii) a existência de relacionamentos entre cenários, que nos permite a criação de uma rede de relacionamentos entre artefatos descritos com esta técnica, uma característica muito interessante para qualidade de rastreabilidade.

Na primeira etapa do processo, o objetivo é a organização dos cenários de requisitos em módulos. Esta organização é obtida através de heurísticas de clusterização aplicadas ao LAL do UdI. Estas heurísticas irão identificar os

conceitos mais importantes dentro do LAL, que representarão os módulos do software. A estreita relação entre os cenários e o LAL será utilizada a fim de dispor os cenários de acordo com os módulos identificados. Através desta etapa, propomos a modularização a partir dos requisitos do software (LAL e cenários). A originalidade desta etapa reside no fato de que estas heurísticas se alicerçam em técnicas de requisitos em linguagem natural, que fornecem a base para encontrar os relacionamentos entre seus artefatos. O mesmo não acontece em (Al-Otaiby et al., 2005), onde estes relacionamentos tem que ser inferidos.

A segunda etapa do processo consiste na derivação da arquitetura, segundo o padrão MVC, com base nos cenários de requisitos. Para esta tarefa, foram propostas heurísticas, através das quais cenários específicos de cada camada do MVC são criados a partir dos cenários de requisitos. Esta tarefa é executada de forma que é mantido um rastro bidirecional entre os cenários iniciais e os de camada originados a partir deles. Com isto, obtemos a arquitetura do software, que é explicada através dos cenários de camadas, e pode ser rastreada até os requisitos. Nesta etapa, a contribuição em relação a trabalhos existentes reside no fato de que um rastro bi-direcional é mantido entre os requisitos e a arquitetura. Além disto, a arquitetura é explicada através dos cenários criados, que são escritos em linguagem natural. Isto não acontece em (Aldrich et al., 2000), onde o rastro entre a arquitetura e os requisitos é mantido em apenas uma direção, dos requisitos para a arquitetura, e não é fornecida uma explicação em linguagem natural para arquitetura do software.

Durante a última etapa do processo, o código do software é produzido. Antes de escrever o código propriamente dito, os cenários das camadas são refinados e cenários mais detalhados são produzidos (chamados de cenários operacionais), de forma que se mantém um rastro entre eles e os das camadas. Os cenários operacionais são então codificados. Ao final desta etapa, obtemos um único documento, que contém o código fonte do software integrado com os cenários operacionais. Os cenários operacionais deste documento podem ser rastreados até os de camada (arquitetura), que por sua vez, podem ser rastreados até os iniciais (requisitos). O caminho inverso também pode ser percorrido. Esta última etapa se assemelha a proposta *Model Driven Development* (MDD) (Sheng e Benatallhah, 2005; Hastbacka e Vepsalainen, 2011; Agner et al., 2012), porém, a diferença é que nossa abordagem não é restrita a um domínio específico. Além disto, o código não é gerado a partir de modelos semelhantes à arquitetura, e sim de modelos em linguagem natural. Por fim, nossa abordagem tem como resultado uma rastreabilidade bi-direcional

entre arquitetura e código, enquanto a maioria das técnicas MDD apresenta apenas a rastreabilidade da arquitetura para o código.

Resumidamente, o processo proposto é composto de etapas encadeadas, de forma que o produto de cada uma é verificado antes de ser utilizado na seguinte, procurando com isso atenuar problemas com a qualidade. Nestas etapas, critérios e heurísticas são estabelecidas, com o intuito de padronizar a estrutura do software produzido e criar uma sequência lógica de passos, onde qualidades de transparência (detalhamento, dependência e rastreabilidade) são atendidas através de operacionalizações integradas com as atividades tradicionais do desenvolvimento de software.

1.4. Organização da Tese

No capítulo 2, apresentamos os conceitos e abordagens, que inspiraram e foram utilizados como alicerce de nossa pesquisa. O capítulo 3 apresenta o processo que foi proposto para lidar com as dificuldades enunciadas neste trabalho. Tal processo foi distribuído em etapas, com entradas e saídas bem definidas, com o intuído de aclarar seus procedimentos. No capítulo 3, expomos como as contribuições obtidas através do processo foram anexadas ao CTS. No capítulo 4, descrevemos as avaliações realizadas, através de estudos empíricos, com o intuito de determinar se os objetivos esperados foram alcançados. Finalmente, no capítulo 5, apresentamos o relacionamento da pesquisa com trabalhos anteriores, nossas conclusões, limitações do trabalho e os trabalhos futuros.