2
Fundamentação

O objetivo deste trabalho é contribuir com pesquisas relacionadas à detecção de anomalias de modularidade em código orientado a objetos. Esses problemas normalmente trazem impactos negativos à evolução e manutenção de sistemas e frequentemente podem ser solucionados por meio de refatorações. Este capítulo apresenta conceitos importantes relacionados a este trabalho. A Seção 2.1 aborda a manutenção de sistemas, a qual motiva estudos como o nosso. Na Seção 2.2, é apresentado o processo de refatoração e alguns exemplos de anomalias de modularidade de código. Para finalizar o capítulo, na Seção 2.3, explicamos o uso da terminologia "sensível à história", frequentemente citada neste trabalho.

2.1
Manutenção e Evolução de Sistemas

2.1.1
Manutenção e suas Categorias

Os sistemas de software frequentemente necessitam passar por mudanças, mesmo após terem sido entregues para seus clientes (Sommerville 2006). Os motivos para essas mudanças podem ser: defeitos não detectados nos testes, evolução da plataforma de software/hardware, alteração de requisitos já implementados ou ainda necessidade de implementar novas funcionalidades.

À atividade de efetuar essas mudanças de forma controlada e organizada dá-se o nome de manutenção de software. As razões pelas quais ocorrem tais mudanças após a liberação dos sistemas para uso determinam a categoria ou tipo de manutenção. Algumas categorias frequentemente apresentadas pela literatura (Sommerville 2006) são:

- manutenção corretiva;
- manutenção adaptativa;
- manutenção evolutiva.

Mesmo efetuando-se a atividade de teste de forma correta, é comum que alguns erros só sejam descobertos com o uso do sistema pelo usuário final.

A manutenção corretiva é o processo que inclui o diagnóstico e a correção de erros do programa após o programa já ter sido entregue. Além disso, as plataformas de software e hardware estão em rápida evolução. A cada ano são lançadas novas gerações de computadores, periféricos, sistemas operacionais e aplicativos com os quais o sistema interage. A manutenção adaptativa modifica o software para que ele tenha uma interface adequada com este ambiente de inovações de hardware e software.

As características do negócio e as necessidades dos usuários se modificam ao longo da vida útil do sistema. Novas capacidades e novas funcionalidades são requeridas. Se o sistema não evolui para atender a essas mudanças ele se torna obsoleto. A manutenção evolutiva é a atividade de modificar o sistema para atender a essas requisições. Esta atividade é responsável pela maior parte do esforço de manutenção.

Além dessas categorias, também é bastante comum chamar de manutenção preventiva (Pressman 2006) quando o software é modificado para melhorar características de confiabilidade ou facilitar a realização de alterações futuras. Esta atividade é caracterizada pelo uso das técnicas de reestruturação de sistemas, onde está inserido o contexto de refatorações de código (Seção 2.2).

Como destaca Sommerville (2006), embora esses diferentes tipos de manutenção sejam geralmente reconhecidos, autores diferentes, algumas vezes, lhes dão nomes diferentes. Ele destaca, por exemplo, que para alguns a manutenção evolutiva significa aperfeiçoar o software implementando novos requisitos e, para outros, se refere a melhorar a estrutura e desempenho do software. Em razão dessa nomenclatura incerta, muitos autores, como Sommerville, preferem evitar utilizar uma taxonomia para os diferentes tipos de manutenção. No nosso trabalho manutenção e evolução de código são tratadas como sinônimos, tomando como base o fato de que ambas resultam em alterações no código.

A realidade é que a grande maioria dos sistemas é feita para refletir comportamentos do mundo real (Gall et al. 1997). Como o mundo real está em constante mudança, sistemas de software frequentemente precisarão sofrer mudanças. Neste trabalho, consideraremos como evolução de sistemas qualquer mudança que gere uma nova versão do sistema. A geração desse conjunto de versões resultará no que chamamos de histórico ou história do sistema. Esse histórico será o principal recurso considerado na abordagem de detecção discutida neste trabalho.

2.1.2 Manutenibilidade de Sistemas

Manutenibilidade de software pode ser definida qualitativamente como a facilidade com que um software pode ser entendido, corrigido, adaptado e/ou aumentado (Pressman 2006). Um processo de desenvolvimento mal feito tem um impacto negativo sobre a manutenibilidade de um software. Uma configuração de software ruim tem um impacto negativo semelhante, assim com a falta de documentação e diversos outros fatores. Não são apenas fatores relacionados à estrutura do código que podem afetar a manutenibilidade de sistemas. Entretanto, neste trabalho nos limitamos basicamente em considerar sintomas presentes no código os quais são difundidos como prejudiciais à manutenibilidade de sistemas (Seção 2.2).

Apesar de não ser uma atividade simples, manter e alterar sistemas trata-se de uma realidade cada vez mais comum entre desenvolvedores. A necessidade de sistemas de software sofrer mudanças, assim como as dificuldades para realização dessas mudanças são temáticas discutidas há anos entre pesquisadores (Lehman e Belady 1985, Lehman et al. 1997) As Leis de Evolução criadas por Lehman apresentam algumas dessas discussões (Lehman e Belady 1985).

Por exemplo, a lei de mudança contínua (primeira lei) aborda a necessidades do software de sofrer mudanças. Segundo ela, "um software deve ser continuamente adaptado, caso contrário se torna progressivamente menos satisfatório". Já as leis de complexidade crescente (segunda lei) e de qualidade decrescente (sétima lei) abordam alguns problemas possivelmente resultantes de processos de manutenção. Uma destaca que "à medida que um software é alterado, sua complexidade cresce, a menos que um trabalho seja feito para mantê-la ou diminuí-la". A outra considera que "programas apresentarão qualidade decrescente ao longo de sua evolução a menos que sejam rigorosamente mantidos e adaptados às mudanças no ambiente operacional".

Assim como Lehman, outros pesquisadores (Parnas 1994) são enfáticos em mencionar que não acompanhar as mudanças requeridas a um sistema pode implicar em perda de qualidade ou até mesmo no fim de sua vida útil. É o que Parnas (1994) chama de envelhecimento de software. Diante desse contexto, muitas pesquisas têm se preocupado em descobrir mecanismos que contribuam com a manutenibilidade de sistemas (Gamma et al. 1995, Riel 1996, Fowler et al. 1999, Lanza e Marinescu 2006) e que possam controlar o aumento da complexidade e queda da qualidade discutidos nas leis de Lehman. Conhecer anomalias comuns na estrutura do código e eliminá-las através de refatorações de código são propostas considerados por Fowler (1999) para evitar a degeneração da manutenibilidade de sistemas de software.

2.2 Refatoração e Anomalias de Modularidade de Código

Segundo pesquisas (Lehman et al. 1997), sistemas evoluem e cada modificação adicional se torna mais difícil ao longo dos anos. Além disso, se nada é feito ao longo do tempo de vida do sistema, sua complexidade tende a aumentar enquanto a sua qualidade tende a diminuir (Lehman et al. 1997). De acordo com estudiosos, conhecer as estruturas possivelmente problemáticas no código, identificá-las e corrigí-las são atitudes que podem controlar a manutenibilidade do código. O trabalho de (Fowler et al. 1999) fornece importantes contribuições em tal contexto.

Martin Fowler e Kent Beck (1999) catalogaram diversos problemas de código denominados por eles como "*code smells*". Tal termo pode ser entendido como uma metáfora normalmente usada para descrever sintomas observados nos módulos que potencialmente prejudicarão a manutenibilidade ou mesmo o reúso do sistema. Tratam-se de sintomas frequentemente resultantes de indevida modularidade. Por isso, nesta dissertação usamos o termo anomalias de modularidade (ou, simplesmente, anomalias) como sinônimo ao termo em inglês *code smells*. De acordo com (Fowler et al. 1999), a identificação dessas anomalias direciona a realização de atividades de reestruturação de código conhecidas como refatorações.

Refatorações são mudanças feitas na estrutura do software no sentido de deixá-lo mais fácil de ser entendido e menos custoso para ser mantido, sem, contudo, modificar seu comportamento observável (Fowler et al. 1999). A idéia central é distribuir classes, variáveis e métodos a fim de facilitar futuras adaptações ou extensões. O processo de refatoração consiste em uma sequência de passos (Mens e Tourwé 2004) em que o primeiro é a identificação das anomalias de código candidatas a sofrer um determinado tipo de refatoração. Entretanto, a localização manual dessas anomalias não é uma tarefa trivial e algumas razões disso podem ser citadas, como:

- sistemas a serem mantidos tendem a ser grandes, o que dificulta a inspeção de código de todos os módulos que integram esses sistemas;
- sistemas são desenvolvidos por diferentes desenvolvedores e equipes. Nesse caso, anomalias podem não ser identificadas por desenvolvedores ou inspetores devido à falta de compreensão de alguns módulos da aplicação;
- a falta de conhecimento sobre anomalias de modularidade de código é bastante comum entre desenvolvedores, principalmente iniciantes. Nesse

caso, esses desenvolvedores dificilmente conseguem identificar oportunidades de refatoração no código que implementam.

Por razões como essas, estudiosos têm pesquisado recursos (Capítulo 3) que possam contribuir com a detecção de anomalias de código. A pesquisa apresentada nesta dissertação visa contribuir com tal etapa. Alguns exemplos de anomalias de modularidade a serem localizadas e que potencialmente resultarão na necessidade de refatorações de código são: *God Class* (GC), *Shotgun Surgery* (SS) e *Divergent Change* (DC).

God Classes tendem a ser classes grandes que centralizam muitas funcionalidades de um sistema. *Shotgun Surgery* é uma classificação para classes que tendem a ocasionar diversas alterações em cascata no sistema. *Divergente Change* é caracterizada por classes que tendem a sofrer alterações provenientes de naturezas divergentes. Discutimos cada uma dessas anomalias no Capítulo 5, precedendo a apresentação das estratégias que se propõem a contribuir com a detecção desses problemas.

2.3 Avaliação Sensível à História

Antes de apresentarmos os demais capítulos desta dissertação, gostaríamos de deixar registrado exatamente o que consideramos uma abordagem sensível à história (SH) para detecção de anomalias. Utilizamos tal terminologia para classificar recursos de detecção que em avaliações de código não consideram apenas a análise da versão atual do sistema, de forma isolada. Adicionalmente a isso, são analisadas características evolutivas do conjunto (ou subconjunto) de versões que integram o histórico de evolução de um sistema. Como recursos de detecção nos referimos a métricas (Seção 3.1), estratégias de detecção (Seção 3.2) ou ferramentas relacionadas que tenham esse propósito de contribuir com a identificação de anomalias de código.

De forma geral, avaliações sensíveis à história se distinguem de avaliações convencionais pelas informações consideradas durante a análise do código. Em avaliações sensíveis à história é possível identificar, por exemplo, a instabilidade de uma dado módulo em relação a quantidade de vezes que seu tamanho foi alterado, ou outras propriedades relacionadas à evolução do código que se deseja avaliar. Avaliações convencionais se limitam a analisar apenas as características de código de uma única versão do sistema, ou seja, são agnósticas ao comportamento evolutivo do mesmo.