Universidade Federal da Bahia

Universidade Salvador

Universidade Estadual de Feira de Santana

TESE DE DOUTORADO

Caracterização da Complexidade Estrutural em Sistemas de Software Livre

Antonio Soares de Azevedo Terceiro

Programa Multiinstitucional de

Pós-Graduação em Ciência da Computação – PMCC

Salvador - BA

2012

PMCC-DSc-0006

ANTONIO SOARES DE AZEVEDO TERCEIRO

CARACTERIZAÇÃO DA COMPLEXIDADE ESTRUTURAL EM SISTEMAS DE SOFTWARE LIVRE

Tese apresentada ao Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, Universidade Estadual de Feira de Santana e Universidade Salvador, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientadora: Prof. Dra. Christina von Flach Garcia Chavez
Co-orientador: Prof. Dr. Manoel Gomes de Mendonça Neto

Salvador
2012

Ficha catalográfica.

Terceiro, Antonio Soares de Azevedo

Caracterização da Complexidade Estrutural em Sistemas de Software Livre/ Antonio Soares de Azevedo Terceiro– Salvador, 2012.

135p.: il.

Orientadora: Prof. Dra. Christina von Flach Garcia Chavez.
Co-orientador: Prof. Dr. Manoel Gomes de Mendonça Neto.
Tese (doutorado)– Universidade Federal da Bahia, Instituto de Matemática, 2012.

1. Complexidade Estrutural. 2. Manutenção de Software. 3. Fatores Humanos em Engenharia de Software. 4. Mineração de Repositórios de Software. 5. Teorias em Engenharia de Software. 6. Engenharia de Software Experimental. 7. Projetos de Software Livre. .
I. Chavez, Christina von Flach Garcia. II. Mendonca, Manoel Gomes de. III. Universidade Federal da Bahia. Instituto de Matemática. IV. Título.

CDD 20.ed. 005.2

TERMO DE APROVAÇÃO

ANTONIO SOARES DE AZEVEDO TERCEIRO

CARACTERIZAÇÃO DA COMPLEXIDADE ESTRUTURAL EM SISTEMAS DE SOFTWARE LIVRE

Esta tese foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da UFBA-UEFS-UNIFACS.

Salvador, 23 de Março de 2012

Profa. Dra. Christina von Flach Garcia Chavez
Universidade Federal da Bahia

Prof. Dr. Dalton Dario Serey Guerrero
Universidade Federal de Campina Grande

Prof. Dr. Guilherme Horta Travassos
Universidade Federal de Rio de Janeiro

Prof. Dr. Claudio Nogueira Sant'Anna
Universidade Federal da Bahia

Prof. Dr. Eduardo Santana de Almeida
Universidade Federal da Bahia

AGRADECIMENTOS

A minha esposa, Joselice, pelo apoio incondicional e suporte durante o desenvolvimento deste trabalho.

A meus pais, referenciais de postura na minha vida.

À professora Christina Chavez, a melhor orientadora que alguém poderia querer.

A Manoel Mendonça e Daniela Cruzes, pelos ensinamentos e colaboração.

A Paulo Meirelles e Fabio Kon, pela parceria frutífera que estabelecemos.

A Gail Murphy, por ter me recebido em seu laboratório em Vancouver, Canadá.

A Roberto Bittencourt, por nos receber e ajudar em Vancouver.

Aos amigos da Colivre, por tocarem o barco enquanto tive de me afastar para concluir este trabalho.

RESUMO

Esta tese apresenta uma caracterização da complexidade estrutural em sistemas de software livre, com objetivo de identificar (i) a contribuição de diversos fatores para a variação da complexidade estrutural e (ii) os efeitos da complexidade estrutural sobre projetos de software. Possíveis fatores na variação da complexidade estrutural incluem: fatores humanos, como experiência geral dos desenvolvedores e a sua familiaridade com as diferentes partes do sistema; fatores relacionados às mudanças realizadas no sistema, como variação no tamanho, espalhamento das mudanças; e fatores organizacionais, como maturidade do processo de desenvolvimento e a estrutura de comunicação do projeto. Efeitos da complexidade estrutural incluem maior esforço, e consequentemente maior custo, em atividades de compreensão e manutenção de software.

Para testar as possíveis causas da complexidade estrutural, foram realizados quatro estudos experimentais, utilizando mineração de dados em repositórios de projetos de software livre. Foram analisados dados históricos de mudanças realizadas em 13 sistemas de diferentes domínios de aplicação e escritos em diferentes linguagens de programação. Os resultados dos estudos realizados são sintetizados através de uma teoria que descreve causas e consequências da complexidade estrutural.

Os resultados indicaram que todos os fatores estudados influenciaram a variação da complexidade estrutural em pelo menos um dos projetos, mas projetos diferentes foram influenciados por conjuntos diferentes de fatores. Modelos construídos foram capazes de descrever até 93% da variação na complexidade estrutural nos projetos estudados.

Palavras chave: Complexidade Estrutural, Manutenção de Software, Fatores Humanos em Engenharia de Software, Mineração de Repositórios de Software, Teorias em Engenharia de Software, Engenharia de Software Experimental, Projetos de Software Livre.

ABSTRACT

This thesis presents a characterization of structural complexity in free software systems to identify (i) the contribution of several factors to the structural complexity variation and (ii) the effects of structural complexity in software projects. Possible factors in the structural complexity variation include: human factors, such as general experience of the developers and their familiarity with the different parts of the system; factors related to the changes performed on the system, such as size variation and change diffusion; and organizational factors, such as the maturity of the software development process and the communication structure of the project. Effects of structural complexity include higher effort, and consequently higher cost, in software comprehension and maintenance activities.

To test possible causes of structural complexity, four empirical studies were performed, mining data from free software project repositories. We analyzed historical data from changes performed in 13 systems from different application domains and written in different programming languages. The results of these studies were summarized by means of a theory that describes causes and consequences of structural complexity.

The results indicated that all the factors studied influenced the structural complexity variation in at least one of the projects, but different projects were influenced by different sets of factors. The models obtained were capable of describing up to 93% of the structural complexity variation in the projects analyzed.

Keywords: Structural Complexity, Software Maintainance, Human factors in Software Engineering, Mining Software Repositories, Theories in Software Engineering, Empirical Software Engineering, Free/Open Source Software Projects.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

Este capítulo descreve a motivação do trabalho, seus objetivos, a metodologia adotada e os resultados obtidos.

INTRODUÇÃO

Manutenção e evolução de software são atividades que consomem uma grande parte dos custos associados ao ciclo de vida de um sistema de software (Bennett; Rajlich, 2000). Consequentemente, a compreensão, avaliação e controle de fatores que as influenciam são de fundamental importância.

A complexidade dos sistemas de software (Mccabe, 1976; Darcy et al., 2005; Sangwan; Vercellone-smith; Laplante, 2008) é um fator que possivelmente influencia as atividades de menutenção e evolução. Quanto maior a complexidade de um sistema de software, maior é o esforço para compreendê-lo, modificá-lo e evoluí-lo (Darcy et al., 2005; Midha, 2008). Em especial, a *complexidade estrutural*, uma medida de complexidade definida em termos de acoplamento e coesão (Darcy et al., 2005), é um indicativo de problemas na manutenibilidade de sistemas de software (Darcy et al., 2005; Meirelles et al., 2010).

Darcy e colegas realizaram um experimento controlado com desenvolvedores profissionais com o intuito de avaliar se a complexidade estrutural poderia exercer influência na atividade de manutenção de software e, mais especificamente, causar um aumento no esforço necessário para atividades de manutenção. Os autores observaram que, de fato, um aumento nessa medida levou a um aumento no tempo necessário para realizar atividades de manutenção de software (Darcy et al., 2005). Como para uma boa parte dos sistemas de software a necessidade de mudança é inevitável (Lehman; Belady, 1985; Lehman et al., 1997), minimizar o esforço necessário para atividades de manutenção é algo desejável.

Os efeitos do excesso de complexidade em projetos de software podem ser verificados em episódios de conhecimento público. `gnome-session` e `eog` são dois projetos de software livre, componentes do GNOME[1], um sistema de área de trabalho para computadores e dispositivos embarcados. Num determinado ponto da evolução destes sistemas, seus

[1] `http://gnome.org/`

mantenedores resolveram reescrevê-los completamente. Em conversas pessoais com um desenvolvedor envolvido nos processos de reescrita dos dois sistemas, foi relatado que um dos motivos que levaram à decisão de reimplementá-los a partir do zero foi o fato do seu código fonte ter se tornado "complexo demais": consertar defeitos passou a demandar cada vez mais esforço, e adicionar novas funcionalidades ainda mais.

O esforço de reescrita de sistemas inteiros é algo que idealmente se quer evitar, de forma que os desenvolvedores possam dedicar o seu esforço a atividades que tragam retorno ao projeto, como a correção de defeitos no código existente e a satisfação de novos requisitos através da implementação de novas funcionalidades.

Neste contexto, a compreensão, avaliação e controle de fatores que influenciam a complexidade estrutural de sistemas de software também adquirem importância.

1.1 PROPOSTA

Estudos experimentais sobre a medição da complexidade estrutural e sua evolução através do tempo (Darcy et al., 2005; Stewart; Darcy; Daniel, 2006; Darcy; Daniel; Stewart, 2010) não fornecem evidências sobre os fatores que influenciam a evolução da complexidade estrutural no decorrer do ciclo de vida de um projeto de software.

Assim, este trabalho tem o objetivo geral de melhorar o entendimento sobre a complexidade estrutural e os fatores que a influenciam ao longo do tempo, de modo a contribuir para mitigar os seus efeitos em projetos de software.

Assim, este trabalho tem os seguintes objetivos específicos:

1. Identificar os fatores que influenciam a complexidade estrutural em sistemas de software.

2. Identificar os efeitos da complexidade estrutural sobre o esforço necessário para atividades de manutenção em projetos de software.

3. Testar a influência de um subconjunto dos fatores identificados como possíveis causas da complexidade estrutural através de estudos experimentais.

O conhecimento sobre quais fatores influenciam a complexidade estrutural pode ser útil para que desenvolvedores e gerentes de projeto possam controlar o crescimento da complexidade, ou buscar a sua redução. Isto pode contribuir para uma maior capacidade de compreensão e mudança no sistema, reduzindo o esforço de manutenção e consequentemente os custos.

Com base em um *brainstorm* inicial, identificou-se que a complexidade estrutural de um projeto de software pode aumentar em função de diferentes fatores, como o nível de maturidade do projeto, o aumento no tamanho do seu código fonte, o tipo de manutenção

ao qual o projeto é submetido, práticas de desenvolvimento, decisões de projeto tomadas no início do projeto, e mesmo características dos desenvolvedores que trabalham no projeto.

Com o objetivo de estudar fatores relacionados conjuntamente, estes fatores foram classificados em três grupos: fatores relacionados à organização desenvolvedora, fatores relacionados à manutenção a qual o sistema é submetido, e fatores relacionados aos desenvolvedores envolvidos no projeto. Em função no limite de tempo para o desenvolvimento do trabalho e do escopo definido – projetos de software livre, foi dada ênfase aos fatores relacionados à manutenção e aos fatores relacionados aos desenvolvedores; os fatores relacionados à organização não foram abordados. A figura 1.1 ilustra o escopo do presente trabalho.

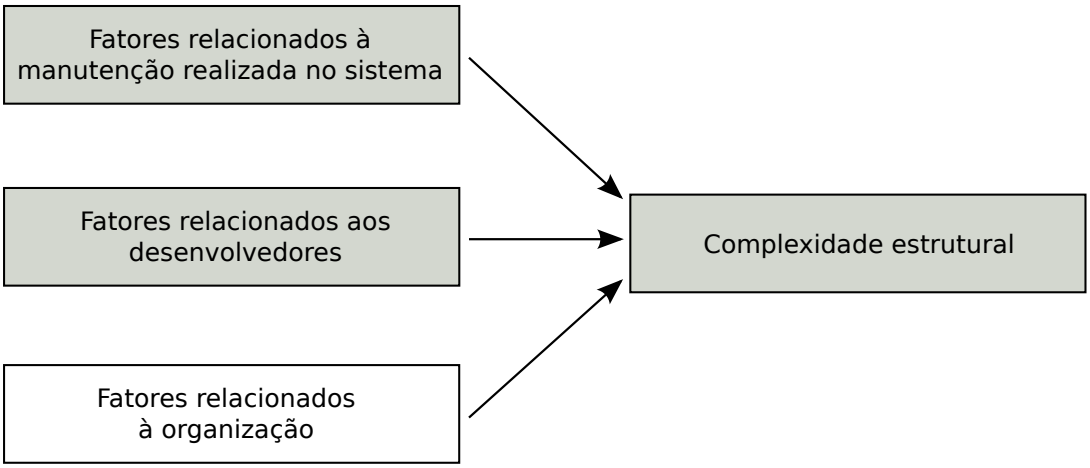


Figura 1.1 Grupos de fatores que podem influenciar a complexidade estrutural. Os grupos de fatores abordados diretamente neste trabalho estão destacados em cinza.

1.2 METODOLOGIA DE TRABALHO

Nesta tese, foi investigada a influência de diversos fatores sobre a variação da complexidade estrutural, bem como os efeitos da complexidade estrutural sobre a atividade de manutenção de software.

Os fatores identificados como potenciais influenciadores da complexidade estrutural foram classificados em três diferentes grupos. Estes grupos reuniram fatores conceitualmente relacionados, de forma que os fatores pertencentes a a cada grupo pudessem ser avaliados em conjunto. Esta classificação foi a seguinte:

1. Fatores relacionados aos desenvolvedores: experiência do desenvolvedor no projeto, grau de autoria sobre módulos do sistema, nível de participação em projetos de software livre.

2. Fatores relacionados à manutenção realizada sobre o sistema: variação de tamanho do sistema, difusão de mudança, tipo de mudança.

3. Fatores relacionados à organização onde o projeto de software se insere: nível de maturidade, estrutura organizacional.

Para validar as proposições da teoria correspondentes aos fatores dos grupos 1 e 2, foi utilizada uma abordagem experimental (Wohlin et al., 2000; Kitchenham et al., 2002; Stol; Babar, 2009).

Quatro estudos experimentais foram realizados através da mineração de repositórios de projetos de software livre. As unidades de análise utilizadas nos estudos foram as mudanças armazenadas nos repositórios de controle de versão dos projetos. Para cada uma destas mudanças, o seguinte procedimento foi realizado:

- Obtenção do código fonte correspondente ao estado atual do sistema exatamente após aquela mudança.

- Extração de métricas do código fonte.

- Cálculo dos valores das variáveis correspondentes aos fatores estudados.

- Cálculo da variação na complexidade estrutural causada por cada mudança. Para isso, o valor atual da complexidade estrutural é subtraído do valor da complexidade estrutural correspondente ao estado do código fonte do sistema antes da realização da mudança.

O primeiro estudo realizado foi um estudo exploratório no qual desejava-se experimentar esta abordagem de análise da evolução da complexidade estrutural através da mineração de dados semiautomática em repositórios de projetos de software livre. Para isso, foram analisadas 21 versões, abrangendo 15 meses de desenvolvimento, de um projeto de pequeno porte escrito em C.

O segundo estudo investigou a influência do nível de participação de desenvolvedores em projetos de software livre na complexidade estrutural adicionada ou removida por eles. O nível de participação de um desenvolvedor em um projeto de software livre indica a posição social de um desenvolvedor naquele projeto(Mockus; Fielding; Herbsleb, 2002; Crowston; Howison, 2005): desenvolvedores centrais realizam a maior parte das atividades e possuem maior poder de decisão, enquanto os desenvolvedores periféricos realizam uma menor parte das atividades e possuem menor poder de decisão. Foram analisados 7 projetos de servidores web escritos em C, que totalizavam mais de 13.000 mudanças em seus repositórios. O desenvolvedor responsável por cada mudança foi classificado como central ou periférico no momento da realização da mudança, e comparou-se a variação na complexidade causada por desenvolvedores centrais e periféricos.

O terceiro estudo investigou três fatores de uma vez: experiência dos desenvolvedores, variação de tamanho e difusão da mudança. Neste trabalho, a experiência dos desenvolvedores é um indicador da quantidade de atividades realizadas anteriormente pelo desenvolvedor no projeto (Mockus; Weiss, 2000); a variação de tamanho indica o número de linhas de código adicionadas ou removidas por cada mudança, e a difusão da mudança indica o número de arquivos adicionados ou modificados por cada mudança (Mockus; Fielding; Herbsleb, 2002). Foram analisados 5 projetos escritos em C, C++ e Java, cada um com pelo menos 18 meses de desenvolvimento, que totalizaram mais de 5.000 mudanças. Foram construídos modelos de regressão linear, onde a variação na complexidade estrutural foi modelada como uma função dos fatores estudados. Mudanças que aumentavam a complexidade estrutural foram analisadas separadamente de mudanças que reduziam a complexidade estrutural, por considerarmos que representam atividades de manutenção de naturezas distintas.

O quarto estudo experimental teve o objetivo de refinar os modelos de regressão linear obtidos anteriormente. Para isso, foram analisados os mesmos projetos utilizados no estudo anterior. Os modelos foram refinados através da introdução do grau de autoria do desenvolvedor sobre os módulos alterados e da separação da difusão de mudança em número de módulos alterados e número de módulos adicionados. O grau de autoria mensura a interação de um desenvolvedor com um determinado artefato de software (Fritz et al., 2010).

Para sintetizar os resultados dos estudos, foi formulada uma teoria que identifica causas da complexidade estrutural e suas consequências sobre o processo de manutenção de sistemas de software livre.

Para apoiar a realização dos estudos, foi desenvolvida uma ferramenta chamada Analizo. A Analizo é capaz de analisar código fonte escrito em C, C++ e Java, calcula um conjunto de mais de 20 métricas de software, e suporta analisar cada revisão armazenada em um repositório de controle de versão, armazenando tanto métricas quanto informações estruturais do código fonte correspondente a cada mudança.

1.3 RESULTADOS

Os principais resultados deste trabalho incluem:

- Obtenção de resultados experimentais acerca da influência de fatores humanos e características da manutenção realizada sobre a variação da complexidade estrutural em sistemas de software livre, e consequentemente, sobre a manutenibilidade destes sistemas. Em especial,
  - O estudo da evolução da complexidade estrutural do sistema pode ser utilizado para identificar momentos em que a arquitetura do sistema é alterada de forma substancial.

  - Todos os fatores estudados influenciaram a variação na complexidade estrutural em pelo menos um dos projetos estudados.
  - Em cada projeto estudado, a variação na complexidade estrutural foi influenciada por um conjunto diferente de fatores.
  - Além disso, dentro de um mesmo projeto, as mudanças que aumentam a complexidade estrutural e as mudanças que reduzem a complexidade estrutural foram influenciadas por conjuntos diferentes de fatores.
  - Os modelos obtidos são capazes de explicar até 93% da variação na complexidade estrutural como uma função de um subconjunto dos fatores estudados, no contexto dos sistemas estudados.

- Uma proposta de teoria para a complexidade estrutural em projetos de software livre. Apesar da teoria proposta ainda ter pouca evidência experimental e baixa generalidade, a avaliação realizada indica que de ela é de alta utilidade para projetos de software. Além disso, esta teoria pode ser refinada mediante a realização de estudos que forneçam mais evidências experimentais.

- A construção de uma ferramenta para análise de código fonte que pode ser usada para estudos experimentais em larga escala que envolvam análise de código fonte em diferentes linguagens de programação.

1.4 ORGANIZAÇÃO DO TEXTO

O capítulo 2 apresenta conceitos fundamentais sobre projetos de software livre, sua forma de funcionamento, suas diferenças e semelhanças em relação a projetos de software proprietário (ditos "convencionais").

No capítulo 3, é feita uma revisão da literatura sobre complexidade em sistemas de software, e descrita a medida de complexidade estrutural que foi utilizada nos estudos realizados como parte deste trabalho.

Os fatores estudados como possíveis causas da variação da complexidade estrutural em projetos de software são descritos no capítulo 4.

Os estudos experimentais desenvolvidos para avaliar os diversos fatores como causas da variação da complexidade estrutural em sistemas de software são apresentados e discutidos no capítulo 5.

O capítulo 6 apresenta uma teoria para a complexidade estrutural em sistemas de software livre, indicando principalmente causas e consequências da complexidade estrutural.

O capítulo 7 elenca as contribuições e limitações deste trabalho, e relaciona possíveis trabalhos futuros.

Capítulo

2

Este capítulo discute projetos de software livre. São discutidas definições e as principais caracterítiscas destes projetos, e em seguida faz-se uma discussão da utilização de projetos de software livre como objeto de estudo em Engenharia de Software.

PROJETOS DE SOFTWARE LIVRE

Nesta tese, adotamos uma abordagem experimental que utiliza repositórios de software livre para estudar a influência de alguns fatores na complexidade estrutural de sistemas de software. Assim, neste capítulo, apresentamos uma breve caracterização de projetos de software livre (seção 2.1), bem como uma discussão sobre o uso de projetos de software livre como objeto de estudo na pesquisa em Engenharia de Software (seção 2.2). A seção 2.3 conclui o capítulo, ressaltando os principais conceitos apresentados.

2.1 DEFINIÇÃO E CARACTERÍSTICAS

Software livre representa uma classe de sistemas de software distribuídos sob licenças cujos termos permitem aos seus usuários usar, estudar e modificar o software. Para isso, necessariamente o código-fonte deve estar disponível. Software não considerado como software livre é conhecido como software proprietário.

Do ponto de vista da Engenharia de Software, um aspecto importante do software livre é o seu processo de desenvolvimento. Um projeto de software livre começa quando um desenvolvedor individual ou uma organização decide tornar um produto de software disponível ao público através da internet, de forma que ele possa ser livremente utilizado, modificado e redistribuído.

Depois que uma versão inicial é lançada e divulgada nos canais apropriados, os primeiros usuários começam a usar o software. Alguns desses primeiros usuários podem também ser desenvolvedores, que vão analisar o código fonte e eventualmente propor mudanças para reparar defeitos, adaptar o software para melhor funcionamento nos seus ambientes específicos ou adicionar novas funcionalidades que atendem às suas próprias demandas.

Essas mudanças são enviadas de volta aos desenvolvedores originais na forma de *patches*, arquivos que descrevem as mudanças e podem ser usados pelos desenvolvedores originais para aplicar as mudanças propostas na versão oficial do programa.

Os líderes do projeto vão revisar as mudanças propostas e, caso concordem com o objetivo da mudança e com a sua implementação, irão aplicá-las (ou não) à versão oficial. No lançamento da próxima versão, os usuários finais terão então acesso às novas funcionalidades ou reparos que foram contribuídos de volta para o projeto.

Com o passar do tempo, cada nova versão do produto possui mais funcionalidades e é mais portável do que a versão anterior, devido às contribuições de desenvolvedores externos. Os colaboradores mais frequentes normalmente podem ganhar a confiança dos fundadores do projeto e receberem acesso direto de escrita ao código fonte oficial do projeto, passando assim a poderem fazer mudanças diretamente na versão oficial.

O processo pelo qual desenvolvedores se juntam a projetos de software livre e ganham responsabilidades – e reconhecimento – varia conforme o projeto. Projetos pequenos normalmente possuem procedimentos bastante informais para isso. Por exemplo, um dos desenvolvedores atuais pode simplesmente oferecer uma conta de acesso ao repositório de controle de versão para o novo desenvolvedor. Projetos maiores, por outro lado, podem ter processos mais "burocráticos" para aceitar novos desenvolvedores com acesso privilegiado aos recursos do projeto. Essa "burocracia" pode envolver, entre outras atividades, o preenchimento de formulários de inscrição e assinatura de termos de transferência de direitos autorais (Jensen; Scacchi, 2005, 2007).

O processo de desenvolvimento de projetos de software livre normalmente faz uso de um sistema de controle de versão, como ilustrado na figura 2.1. Enquanto o repositório de controle de versão está disponível publicamente para leitura, o acesso de escrita é restrito a um grupo limitado de desenvolvedores. Os outros desenvolvedores precisam ter os seus *patches* revisados por um desenvolvedor com as permissões necessárias para que as suas contribuições sejam adicionadas ao repositório.

Em geral, projetos de software livre usam ambientes bastantes parecidos para colaboração entre os seus desenvolvedores: além do sistema de controle de versão, normalmente são utilizados um sistema de gestão de atividades para receber relatos de defeitos dos usuários e organizar as atividades pendentes, em andamento e futuras; uma ou mais listas de discussão para coordenação e discussão entre a equipe; e um ou mais sistemas *web* de gestão de conteúdo para publicação do *website* do projeto e escrita de documentação colaborativa.

As seguintes características fazem projetos de software livre interessantes objetos de estudo e os diferenciam significantemente de projetos de software ditos "convencionais":

- **Disponibilidade do código fonte.** O código fonte de projetos de software livre está sempre disponível na internet, e a maioria dos projetos possui um repositório de controle de versão público.

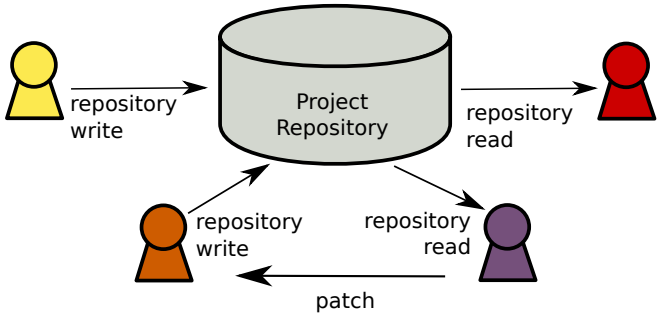


Figura 2.1 Desenvolvimento de software livre através de um repositório de controle de versão.

- **Simbiose desenvolvedor/usuário.** Em boa parte dos projetos de software livre, os desenvolvedores são também usuários do software, e portanto fontes de requisitos. Talvez por isso, muitos projetos não possuam documentos de requisitos de forma explícita, e o desenvolvimento segue um ritmo no qual os desenvolvedores conseguem satisfazer as suas próprias necessidades.

- **Trabalho não contratual.** Uma grande parte do trabalho realizado em projetos de software livre é feita de forma não contratual. Isso não significa que os desenvolvedores sejam necessariamente voluntários contribuindo no seu tempo livre, mas sim que não existe uma gestão central com poder de decisão formal sobre as atividades dos desenvolvedores.

- **O trabalho é auto atribuído.** A ausência de uma gestão central com poder de decisão sobre as atividades dos desenvolvedores promove a auto atribuição de atividades: os desenvolvedores voluntários irão trabalhar nas partes do projeto que mais os atraem, e os desenvolvedores contratados trabalharão nas partes do projeto que mais interessam aos seus empregadores.

- **Distribuição geográfica.** Na maioria dos projetos de software livre, os desenvolvedores estão espalhados pelo mundo. Em projetos com alta dispersão geográfica, a maioria – ou mesmo a totalidade – da comunicação entre os desenvolvedores é realizada através de meios eletrônicos.

Apesar da literatura de Engenharia de Software nos últimos anos ter tratado o software livre como um fenômeno homogêneo (Østerlie; Jaccheri, 2007), as características apresentadas anteriormente não se aplicam a todos os projetos de software livre, e algumas delas se manifestam de formas bastante diferentes, de projeto para projeto.

2.2 PROJETOS DE SOFTWARE LIVRE COMO OBJETO DE ESTUDO EM ENGENHARIA DE SOFTWARE

Dados de projetos privados de desenvolvimento de software são normalmente escassos e podem impor custos para sua aquisição (por exemplo, a necessidade de deslocamento até a organização desenvolvedora). Dados de projeto de software livre, por outro lado, são abundantes e estão disponíveis para qualquer pesquisador com uma conexão à internet. Dados privados normalmente não podem ser publicados para que outros pesquisadores possam replicar os estudos realizados com eles em função de acordos de sigilo, enquanto dados de projetos de software livre já são públicos de qualquer forma (Scacchi, 2007).

A maioria dos projetos de software livre possui todos os seus meios de comunicação – repositórios de controle de versão, sistemas de gestão de atividades, listas de discussão, *websites* de documentação – disponíveis publicamente na internet. Através desses repositórios de informação, pesquisadores podem monitorar e analisar o trabalho da equipe de desenvolvimento sobre um determinado artefato. Este tipo de dado tem sido largamente utilizado em estudos de Engenharia de Software com a assistência de ferramentas que automatizam a extração e análise de dados de repositórios públicos acessíveis pela *web* (Kon et al., 2011).

Projetos de software livre têm produzido com sucesso sistemas de software sofisticados, usando metodologias e processos por vezes substancialmente diferentes daqueles documentados nos livros textos clássicos de Engenharia de Software (Pressman, 2009; Sommerville, 2010). Entender como o software livre funciona e quais as suas diferenças e similaridades com o desenvolvimento de software "convencional" pode ajudar pesquisadores e profissionais da prática a melhorar a prática da Engenharia de Software em geral.

A pesquisa sobre software livre trouxe uma série de novos tópicos interessantes à agenda de pesquisa em Engenharia de Software, como a estrutura social de comunidades de desenvolvimento (Mockus; Fielding; Herbsleb, 2002; Crowston et al., 2006; Jensen; Scacchi, 2005, 2007), comunicação e fluxo de trabalho em projetos de software livre (Masmoudi et al., 2009; Scialdone et al., 2009), evolução e participação de desenvolvedores em projetos de software livre (Costa; Santana; Souza, 2009; Terceiro; Rios; Chavez, 2010), e atratividade de projetos de software livre (Meirelles et al., 2010; Santos Jr. et al., 2011).

Pesquisa sobre software livre não se limita às subáreas existentes da Engenharia de Software: o estudo sobre o fenômeno do software livre em si é também uma área de pesquisa promissora para as ciências sociais em geral, administração, economia e filosofia. A principal conferência internacional sobre pesquisa em software livre, a *International Conference on Open Source Systems*, é um evento multidisciplinar que reúne anualmente pesquisadores de diversas áreas do conhecimento.

Vale ressaltar, no entanto, que o estudo de projetos de software livre possui algumas limitações. Primeiro, a disponibilidade imediata de informações sobre o processo de desenvolvimento não traz consigo acesso direto às pessoas envolvidas no processo, o que

não facilita a realização de estudos qualitativos que necessitem de dados não registrados eletronicamente. Os pesquisadores precisam, assim como no caso de projetos privados, entrar em contato com os participantes do projeto e solicitar as informações necessárias.

Segundo, não se tem controle sobre os participantes do projeto, o que torna infactível a realização de estudos controlados. Os estudos que envolvem projetos de software livre, especialmente na área de Engenharia de Software, são quase sempre baseados em dados históricos obtidos de repositórios *online*.

Por último, os resultados obtidos não podem ser generalizados para o caso geral de projetos de desenvolvimento de software. Apesar da grande quantidade de dados disponíveis sobre projetos de software livre, ainda assim, eles não são representativos da totalidade de projetos de software – assim como projetos de software proprietário específicos não o são. Mas, se por um lado, projetos de software livre não são representativos do caso geral de projetos de desenvolvimento de software, eles também não são mais diferentes de projetos de software proprietário do que projetos de software proprietário distintos são diferentes entre si (Wilson; Aranda, 2011).

Apesar das limitações discutidas, projetos de software livre têm servido à comunidade de pesquisa em Engenharia de Software como objeto de estudo de forma crescente nos últimos anos (Kon et al., 2011).

2.3 CONCLUSÃO

Um projeto de software livre é caracterizado por termos de uso que permitem aos seus usuários usar, modificar e redistribuir o software; consequentemente, seu código fonte é disponibilizado para todos os seus usuários. A disponibilidade do código fonte viabiliza a colaboração de diferentes desenvolvedores através da internet de forma a adaptar o software às suas diversas necessidades.

Uma vez que uma grande parte das atividades de colaboração que compõem o desenvolvimento de um projeto de software livre é registrada eletronicamente e de forma pública, estes projetos representam uma oportunidade sem precedentes para pesquisadores da Engenharia de Software. Apesar disso, é necessário reconhecer as limitações na utilização de projetos de software livre em pesquisas de Engenharia de Software. Por exemplo, a grande heterogeneidade entre projetos de software livre faz com que não haja um projeto de software livre arquetípico, e com que projetos de software livre não possam ser considerados como representativos do caso geral de projetos de desenvolvimento de software. O mesmo acontece com projetos de software proprietário, no entanto.

Neste trabalho, projetos de software livre foram analisados com o objetivo de caracterizar a influência de um conjunto de fatores sobre a variação da sua complexidade estrutural. O capítulo seguinte apresenta os principais conceitos sobre complexidade estrutural.

Capítulo

3

Este capítulo descreve o conceito de complexidade em geral, fazendo um breve apanhado do campo de sistemas complexos. Em seguida, o conceito de complexidade em sistemas de software é discutido e diferentes propostas de medidas para complexidade de sistemas de software são apresentadas, entre elas a medida conhecida como complexidade estrutural, utilizada no restante deste trabalho. O capítulo é concluído com uma discussão sobre os efeitos da complexidade estrutural em sistemas de software e sobre o comportamento evolucionário da complexidade estrutural.

COMPLEXIDADE ESTRUTURAL

Este capítulo é dedicado à complexidade estrutural, um conceito fundamental neste trabalho.

A seção 3.1 introduz de forma geral o conceito de complexidade de acordo com o campo de estudo de sistemas complexos, bem como características comuns de sistemas que são ditos "sistemas complexos", e uma caracterização de sistemas de software como sistemas complexos. A seção 3.2 discute especificamente a complexidade em sistemas de software em termos de conceitos da Engenharia de Software como módulos, abstração, e arquitetura de software. Em seguida, a seção 3.3 discute três abordagens diferentes para a medição de complexidade de software, que fornece subsídios para a escolha de uma abordagem de medição a ser utilizada neste trabalho. A seção 3.4 discute os efeitos da complexidade estrutural sobre projetos de software, e a seção 3.5 discute a evolução histórica da complexidade estrutural em projetos de software. Este capítulo é concluído pela seção 3.6, onde são discutidos os principais resultados identificados.

3.1 SISTEMAS COMPLEXOS

Sistemas complexos são comuns em diversas áreas do conhecimento. Esta seção apresenta alguns destes sistemas e discute as características comuns a eles, segundo Mitchell (Mitchell, 2009).

As formigas são criaturas relativamente simples quando obervadas individualmente. Elas possuem instintos básicos como procurar alimento, responder a estímulos químicos

vindos de outras formigas, combater intrusos, etc. No entanto, quando observadas coletivamente em uma colônia, as formigas aparentam ser muito mais sofisticadas. Elas são capazes de se organizar em diferentes atividades, criar estruturas complexas dentro de seu formigueiro, e de encontrar o caminho mais curto para uma fonte de alimento.

Outros exemplos de sistemas considerados complexos incluem os sistemas nervoso e imunológico, sistemas econômicos, ou sistemas de software, como a *World Wide Web* (Mitchell, 2009). Algo que se pode notar nesta lista é a existência de dois tipos de sistemas complexos: sistemas naturais e sistemas artificiais.

Sistemas naturais são aqueles cuja constituição não tem participação humana. É importante notar que "biológico" não é necessariamente o mesmo que "natural": uma floresta é natural, mas uma fazenda não é (Simon, 1996). As ciências naturais buscam compreender os sistemas naturais e suas propriedades.

Sistemas artificiais são projetados por humanos, com *objetivos* e *funções* definidos. Sistemas artificiais podem ou não serem projetados à imagem de um sistema natural, e durante a sua concepção eles são discutidos em termos tanto de suas *características* (o que eles *são*) como de *necessidades* que eles devem satisfazer (o que eles *deveriam ser*) (Simon, 1996).

Segundo Mitchell, as seguintes características são comuns a todos os sistemas complexos, naturais ou artificiais:

Comportamento coletivo complexo. Apesar de serem compostos por elementos bastante simples individualmente, sistemas complexos podem exibir comportamentos coletivos bastante sofisticados, que são imprevisíveis se considerada unicamente a análise do comportamento individual de seus componentes.

Troca de sinais e processamento de informação. Em cada um destes sistemas, seus componentes individuais consomem e produzem informação entre si. Parte do comportamento do sistema envolve transformação dessa informação que, em alguns casos, pode até ser considerada como uma forma de computação. De fato, o conhecimento existente sobre colônias de formigas e sobre o funcionamento do cérebro humano inspiraram respectivamente os algoritmos de colônias de formigas e as redes neurais artificiais, dois exemplos de técnicas algorítmicas inspiradas em sistemas naturais.

Adaptação Sistemas complexos adaptam-se a novas situações de forma a aumentar suas chances de sobrevivência diante de novas condições em seu ambiente.

A partir dessas características, Mitchell propõe a seguinte definição para sistemas complexos: *um sistema no qual grandes redes de componentes sem controle central e regras simples de operação dão origem a um comportamento coletivo complexo, processamento sofisticado de informação, e adaptação através de aprendizado ou evolução.*

A partir da definição acima, apresentamos a seguir uma caracterização de sistemas de software como um sistema complexo artificial.

Primeiro, sistemas de software exibem *comportamento coletivo complexo*. Sistemas de software são compostos por componentes, em geral chamados de *módulos*, que possuem tanto estado quanto comportamento próprios. Em geral, módulos individuais de um sistema de software são simples quando comparados com o sistema como um todo e, certamente, é mais fácil de entender o comportamento de um módulo do que o de todo o sistema de software. Os módulos são compostos para formar um sistema cujo comportamento é bastante sofisticado em relação ao comportamento individual de seus componentes.

Segundo, sistemas de software exibem *trocas de sinais e processamento de informação*. Módulos produzem informação para outros módulos através de parâmetros em chamadas de sub-rotinas e consomem informação através dos valores de retornos das chamadas realizadas a sub-rotinas de outros módulos e dos parâmetros de suas próprias sub-rotinas quando estas são invocadas por outros módulos.

Por último, sistemas de software de fato passam por *adaptação* para se adequar a mudanças em seu ambiente. O fluxo contínuo de novos requisitos e de mudanças no ambiente operacional de sistemas de software força-os a se manter em constante evolução em busca de"sobrevivência". Sistemas de software, no entanto, se diferenciam dos sistemas complexos naturais pelo fato de serem projetados; consequentemente, o seu processo evolucionário não é intrinsecamente parte do seu comportamento, mas fruto da ação consciente de seus desenvolvedores.

É importante ressaltar que esta caracterização de sistemas de software como sistemas complexos diz respeito à estrutura interna dos sistemas, ou seja, aos componentes que o constituem e ao relacionamento entre estes componentes. Não foram considerados outros aspectos importantes de sistemas complexos, como por exemplo o seu relacionamento com o ambiente externo.

3.2 COMPLEXIDADE EM SISTEMAS DE SOFTWARE

A complexidade de um sistema de software está relacionada a fatores que afetam o custo associado a desenvolvê-lo e mantê-lo (Tegarden; Sheetz; Monarchi, 1995). Segundo Tegarden *et al*, três tipos de complexidade afetam a capacidade que desenvolvedores possuem de compreender sistemas de software: a complexidade do problema, a complexidade procedural, e a complexidade do projeto do sistema. Esta última é o foco deste trabalho.

A *complexidade do problema* está relacionada ao domínio do problema. De forma simplória, assume-se que problemas mais complexos são mais difíceis de compreender do que problemas mais simples. Normalmente, não possuímos qualquer tipo de controle sobre este tipo de complexidade, o que faz com que o seus efeitos sobre a capacidade de compreensão de desenvolvedores seja inevitável.

A *complexidade procedural* está relacionada à estrutura lógica da programa, em especial do seu comprimento, em termos de número de *tokens*, linhas de código fonte, ou estruturas de controle.

A *complexidade do projeto do sistema* está relacionada ao mapeamento entre o domínio do problema e o domínio da solução, na forma de um sistema de software. Tegarden *et al* diferenciam o que eles chamaram de *complexidade estrutural*, associada ao acoplamento entre módulos, e *complexidade de dados*, associada à coesão interna dos módulos. Esta tese trata deste tipo de complexidade, a qual chamamos como um todo de *complexidade estrutural*. A complexidade estrutural, portanto, está relacionada ao projeto do sistema, ou seja, à sua divisão em *módulos*.

Módulos são unidades de um sistema que possuem uma independência relativa entre si, mas que trabalham em conjunto para produzir o comportamento resultante do sistema (Parnas, 1972). O sistema como um todo deve prover uma arquitetura que dá suporte tanto à independência de estrutura como à integração de função (Baldwin; Clark, 1999, p. 63).

A *arquitetura* de um software é a estrutura que determina quais tipos de módulos existem, quais propriedades cada um desses módulos são externamente visíveis, e como os módulos se relacionam entre si (Clements et al., 2002; Bass; Clements; Kazman, 2003).

O ato ou efeito de se projetar um sistema como um conjunto de módulos é chamado de *modularização*. A modularização pode ser considerada como uma das fases da definição da arquitetura do sistema. Segundo Baldwin e Clark (Baldwin; Clark, 1999), a modularização deve seguir dois princípios básicos, o da *ocultação de informação* e o da *abstração*.

Ocultação de informação consiste em esconder detalhes de um módulo dos demais, de forma que estes detalhes possam ser alterados sem que os demais módulos também precisem ser alterados (Parnas, 1972). Parnas propõe que módulos encapsulem decisões que estejam propensas a mudar, de forma que mudanças relacionada a estas decisões não afetem as demais partes do sistema. Desta forma, módulos são projetados para expor uma interface que fornece acesso à sua funcionalidade a outros módulos, mas sem expor os detalhes da sua implementação.

A *abstração* consiste em fazer com que módulos apenas interajam com a interface dos outros módulos. Desde que sua interface seja mantida, cada módulo pode ter a sua implementação alterada sem que isso afete outros módulos.

A modularização é, portanto, uma forma de amenizar os efeitos da complexidade, permitir que módulos independentes sejam desenvolvidos em paralelo, e isolar a incerteza confinando-a dentro de módulos (Baldwin; Clark, 1999, p. 90-91).

Mesmo dividido em módulos, um sistema de software ainda possui uma complexidade com a qual seus desenvolvedores precisam lidar. Além da complexidade do problema, que é natural, o projeto do sistema como um conjunto de módulos inter-relacionados possui uma complexidade característica que afeta a capacidade de desenvolvedores de

compreender e realizar manutenção sobre o sistema. Esta complexidade está relacionada ao relacionamento entre os diferentes módulos do sistema, bem como à estrutura de cada módulo individual.

3.3 MEDIDAS DE COMPLEXIDADE EM SISTEMAS DE SOFTWARE

No campo de sistemas complexos, existem diversas propostas de medidas de complexidade que capturam diferentes aspectos da complexidade de um sistema (Mitchell, 2009). Da mesma forma, na área de Engenharia de Software, existem diferentes propostas de medidas de complexidade de software (Mccabe, 1976; Sangwan; Vercellone-smith; Laplante, 2008; Darcy et al., 2005) que capturam diferentes aspectos da complexidade de um sistema de software.

Esta seção descreve três abordagens para medição de complexidade de sistemas de software. Estas abordagem são caracterizadas em termos dos seguintes critérios:

C1 A medida leva em consideração tanto a complexidade interna de cada módulo como a complexidade inerente ao relacionamentos entre os diferentes módulos do sistema?

Este critério visa garantir que a abordagem captura aspectos importantes da complexidade estrutural de um sistema (Tegarden; Sheetz; Monarchi, 1995).

C2 A medida representa uma dimensão distinta do que se pode obter medindo o tamanho do sistema?

Este critério visa garantir que a abordagem escolhida seja justificável em termos de esforço: uma vez que medir o tamanho do sistema usualmente requer menos esforço do que medir propriedades da sua estrutura, procura-se abordagens que meçam propriedades significantemente não relacionadas ao tamanho.

C3 A medida pode ser usada de forma independente do paradigma de programação e da estrutura escolhida pelos desenvolvedores para organizar os módulos do sistema?

Este critério visa privilegiar as abordagens que possam ser aplicadas no maior número possível de contextos diferentes.

Os critérios representam questões de interesse para este trabalho e, de certo modo, justificam a utilização de uma medida de complexidade de sistemas de software – a complexidade estrutural baseada em acoplamento e coesão (Darcy et al., 2005) – nos estudos experimentais realizados.

3.3.1 Complexidade ciclomática

A partir dos anos 70, surgiram diversas propostas para medir a complexidade de sistemas de software no nível de subrotinas (Mccabe, 1976; Sedlmeyer et al., 1985; Roca,

1996). Destas, a que se tornou mais amplamente usada foi a complexidade ciclomática de McCabe (Mccabe, 1976), que ainda é utilizada em trabalhos recentes (Barry; Kemerer; Slaughter, 2007; Zhang; Baddoo, 2007; Midha, 2008).

A complexidade ciclomática de McCabe é uma medida da complexidade de uma sub-rotina, e calcula o número de possíveis traços de execução de uma sub-rotina. Seja G um grafo que representa o fluxograma de uma sub-rotina, onde cada vértice é uma instrução do programa, e cada aresta indica uma possível transferência de controle entre duas instruções. Seja n o número de vértices de G, e o número de arestas, e p o número de componentes conectados de G, a complexidade ciclomática da sub-rotina representada por G é dado por $v(g) = e - n + 2p$.

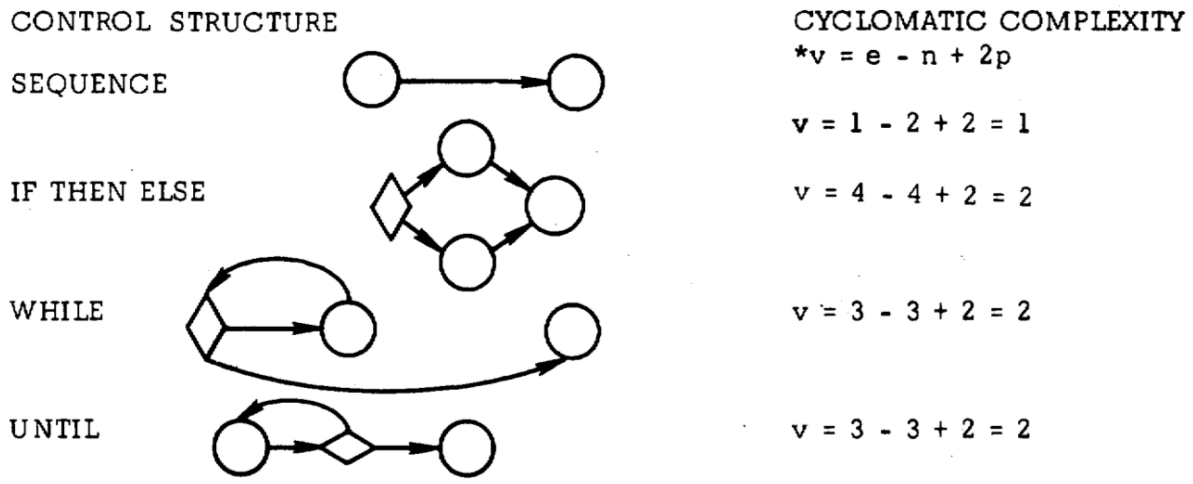


Figura 3.1 Complexidade ciclomática de McCabe para algumas estruturas de controle (Mccabe, 1976).

A figura 3.1 exibe alguns exemplos de complexidade ciclomática para estruturas de controle comuns. Podemos notar que a complexidade ciclomática de código sequencial é 1, e a de estruturas de controle simples como condicionais e laços é 2. Uma forma simples de determinar a complexidade ciclomática de uma sub-rotina é simplesmente contar o número de estruturas de controle explícitas como condicionais e laços e somar 1.

A simplicidade de cálculo da complexidade ciclomática de McCabe é provavelmente um dos fatores que potencializam o seu uso pela comunidade de Engenharia de Software.

Uma das principais desvantagens de se usar complexidade ciclomática como medida de complexidade é que ela só se aplica a sub-rotinas, e portanto, não captura a complexidade proveniente do relacionamento entre diferentes sub-rotinas, a organização das sub-rotinas dentro de um módulo, e o relacionamento entre diferentes módulos num sistema de software.

Outro problema com a complexidade ciclomática é o fato dela estar fortemente cor-

relacionada a medidas de tamanho (Meulen; Revilla, 2007; Jay et al., 2009; Herraiz; Hassan, 2011). Uma vez que medidas de tamanho são usualmente mais simples de calcular, utilizar complexidade ciclomática para medir complexidade de sistemas de software é pouco justificável.

No entanto, a complexidade ciclomática indica o número de possíveis caminhos na execução de uma sub-rotina, e portanto é uma medida bastante útil para a atividade de teste de software (Herraiz; Hassan, 2011).

3.3.2 Complexidade Estrutural como combinação de graus de violação e tamanho

Sangwan e colegas (Sangwan; Vercellone-smith; Laplante, 2008) definem complexidade estrutural de um software em termos da combinação de três características:

1. Grau de violação devido a entrelaçamento.
2. Grau de violação devido a "gordura".
3. Tamanho.

Um estudo sobre a evolução deste tipo de complexidade estrutural foi realizado por Sangwan e colegas (Sangwan; Vercellone-smith; Laplante, 2008) com apoio do framework de medição do *Structure 101*, um pacote de software proprietário comercializado pela *Headway Software*.

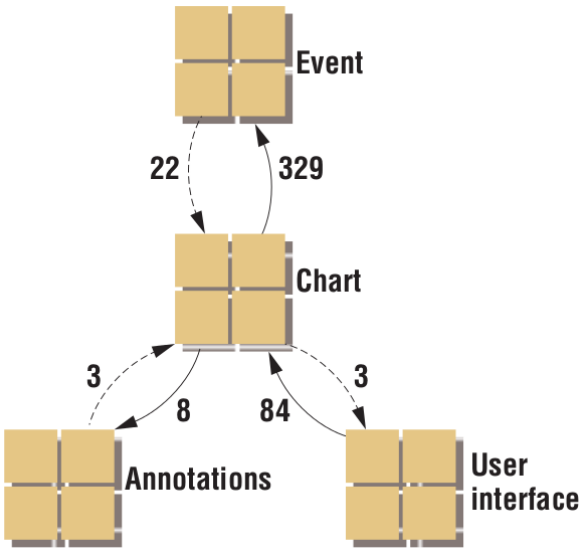


Figura 3.2 Grafo de dependência ilustrando o cálculo do MFS (*minimal feedback system*) (Sangwan; Vercellone-smith; Laplante, 2008)

O *grau de violação por entrelaçamento* é calculado por meio da identificação no grafo de dependências do sistema de feedback mínimo (*minimal feedback system - MFS*) – o menor conjunto de arestas que precisa ser removido para não haver dependências circulares. Por exemplo, na figura 3.2, o *MFS* é composto das arestas com pesos 3, 3 e 22. O grau de violação devido a entrelaçamento é um valor normalizado entre 0 e 1, calculado como a razão entre a soma dos pesos do *MFS* pela soma dos pesos de todas as dependências. No exemplo da figura 3.2, o grau de violação devido a entrelaçamento é $(3 + 3 + 22)/(3 + 3 + 22 + 8 + 84 + 329) \simeq 0.0623$.

A forma de medir a "gordura" depende do nível de abstração analisado. Em níveis de abstração altos, como no nível de pacotes, a "gordura" é representada pelo número de dependências entre seus sub-pacotes, ignorando o peso das dependências. No exemplo da figura 3.2, a "gordura" do pacote representado tem valor 6 (número de dependências entre os sub-pacotes). No nível de pacotes que contêm classes, a "gordura" é o número total de dependências entre as classes do pacote. A "gordura" de uma classe é o número total de dependências entre seus componentes (métodos e atributos), e a "gordura" de um método é definido com sendo a sua complexidade ciclomática (Mccabe, 1976).

O *grau de violação devido a "gordura"* é um valor normalizado entre 0 e 1 dado pela expressão $max\{0, [(value - threshold)/value]\}$, onde $threshold$ é um limiar arbitrário.

A determinação do *tamanho* também depende do nível de abstração: o tamanho de um pacote de alto nível é a soma dos tamanhos de seus sub-pacotes, o tamanho de um pacote que contém classes é a soma dos tamanhos das classes, e assim por diante, até chegar no tamanho de um método, definido como 1 mais o número de instruções contidas no método.

O *Structure 101* então mede a complexidade estrutural de um sistema de software como o produto entre o grau de violação devido a entrelaçamento, o grau de violação devido à "gordura", e o tamanho. A complexidade estrutural do sistema é definida de forma recursiva, somando-se a complexidade estrutural dos elementos de mais baixo nível, e dividindo-se pela soma de seus tamanhos.

A principal desvantagem dessa abordagem é que ela depende de um paradigma específico na organização dos elementos do sistema. Por exemplo, sistemas que não utilizam pacotes como forma de organização hierárquica (seja em função da sua linguagem de implementação, seja por escolha dos desenvolvedores) não se encaixam na estrutura de medição do *Structure 101*.

Outra desvantagem é o fato de que o limiar utilizado no cálculo do grau de violação devido à "gordura" é arbitrário. Isto faz com que a medição dependa de um elemento subjetivo, e com que medições usando limiares diferentes não sejam compatíveis entre si.

3.3.3 Complexidade Estrutural como combinação de Acoplamento e Coesão

Darcy, Kemerer, Slaughter e Tomayko (Darcy et al., 2005) propõem que complexidade estrutural seja medida através da combinação de acoplamento e coesão. Estes são dois conceitos complementares: enquanto o acoplamento reflete o relacionamento entre módulos, a coesão nos fornece uma visão da organização dos componentes internos de um módulo e seus relacionamentos. A literatura clássica para profissionais da prática de projeto de software defende que módulos com baixo acoplamento e alta coesão são mais fáceis de entender e modificar (Martin, 2003; Mcconnell, 2004).

Em termos de complexidade, a combinação de acoplamento e coesão numa única medida captura tanto a complexidade criada pelo excesso de dependências entre diferentes módulos quanto a complexidade causada pela presença de módulos com múltiplas responsabilidades não relacionadas.

Uma formalização da métrica proposta por Darcy et al. (Darcy et al., 2005) pode ser como se segue. A complexidade estrutural de um módulo m é dada pelo produto das suas métricas de acoplamento e falta de coesão. Dado um projeto p e seu conjunto de módulos $M(p)$, a complexidade estrutural de p é dada por $SC(p)$, definida da seguinte forma:

$$SC(p) = \frac{\sum_{m \in M(p)} A(m) \times FC(m)}{|M(p)|}$$

Na definição acima, $A(m)$ representa o acoplamento do módulo m, e $FC(m)$ representa a falta de coesão do módulo m.

Esta medida de complexidade estrutural é portanto a complexidade estrutural média entre todos os módulos do sistema. Quando um sistema cresce em tamanho e também tem a sua complexidade estrutural média aumentada, pode-se inferir que não só o sistema como um todo está maior e portanto, mais difícil de compreender e modificar, mas também que cada módulo, em média, também é mais difícil de compreender e modificar do que antes.

A métrica proposta por Darcy *et al.* para complexidade estrutural possui diversos pontos fortes. Primeiro, ela foi derivada de uma revisão da literatura, na qual acoplamento e coesão foram identificados como fatores que poderiam fornecer uma medida significativa de complexidade estrutural. Segundo, acoplamento e coesão não estão limitados ao paradigma da orientação a objetos. A maioria dos paradigmas de desenvolvimento de software possui uma ou mais construções que fazem o papel de *módulo* – "classes", "aspectos", "tipos abstratos de dados", ou "arquivos-fonte" – para as quais medidas de acoplamento e coesão podem ser obtidas. Finalmente, Darcy *et al.* validaram a suposição de que a complexidade estrutural está associada a maior esforço de manutenção por meio de um experimento controlado. Os autores descobriram que nem acoplamento nem falta

de coesão individualmente explicavam a diminuição do desempenho dos desenvolvedores numa atividade de compreensão; apenas quando acoplamento e falta de coesão foram combinados e considerados como fatores que interagem, foi possível observar uma associação com maior esforço de manutenção.

Ao contrário da complexidade de McCabe, a complexidade estrutural proposta por Darcy *et al* não está necessariamente relacionada ao tamanho do software (Darcy; Daniel; Stewart, 2010; Terceiro et al., 2012), e representa uma dimensão da complexidade de sistemas de software que é independente do tamanho.

O aumento do tamanho é inevitável durante e evolução de sistemas de software num ambiente real, onde existe um fluxo constante de novos requisitos a serem satisfeitos e mudanças frequentes no ambiente externo ao sistema (Lehman; Belady, 1985; Lehman et al., 1997). O desafio da modularização neste tipo de sistema é, portanto, conseguir crescer em tamanho sem que a complexidade estrutural também cresça necessariamente.

3.3.4 Avaliação das medidas

A tabela 3.1 apresenta uma avaliação das diferentes medidas descritas anteriormente em função dos critérios apresentados no início da seção 3.3.

Tabela 3.1 Comparação entre medidas de complexidade para sistemas de software

Proposta	**C1**	**C2**	**C3**
CC (Mccabe, 1976)	Não	Não	Sim
SC no Structure 101 (Sangwan; Vercellone-smith; Laplante, 2008)	Sim	?	Não
SC por Darcy *et al* (Darcy et al., 2005)	Sim	Sim	Sim

Não foi possível avaliar o critério C2 para a medida de complexidade estrutural do Structure 101, pois não foi encontrada na literatura técnica nenhuma evidência sobre a sua relação com o tamanho de sistemas de software.

A medida de complexidade estrutural proposta por Darcy e colegas (Darcy et al., 2005) é a única que atende a todos os critérios colocados, isto é,

- atende a C1, pois leva em consideração a complexidade interna de cada módulo e a complexidade inerente ao relacionamentos entre os diferentes módulos do sistema;
- atende a C2, pois representa uma dimensão distinta do que se pode obter medindo o tamanho do sistema; e
- atende a C3, pois pode ser usada de forma independente da linguagem de programação e da estrutura escolhida pelos desenvolvedores do sistema.

Dessa forma, a medida de complexidade estrutural proposta por Darcy e colegas (Darcy et al., 2005) foi a escolhida para uso nos estudos experimentais realizados neste trabalho.

3.4 EFEITOS DA COMPLEXIDADE ESTRUTURAL EM PROJETOS DE SOFTWARE

Um dos efeitos diretos da complexidade estrutural em projetos de software é o aumento no esforço necessário para atividades de manutenção. No mesmo estudo em que propuseram a medida de complexidade estrutural utilizada neste trabalho, Darcy e colegas (Darcy et al., 2005) realizaram um experimento controlado no qual desenvolvedores profissionais foram expostos a dois sistemas com diferentes valores de complexidade estrutural com o objetivo de realizarem atividades de manutenção de software. As atividades de manutenção no sistema com maior complexidade estrutural levaram mais tempo para serem realizadas. Esse resultado confirma o entendimento de que um aumento da complexidade faz com que sistemas de software sejam mais difíceis de compreender e consequentemente de modificar.

No caso específico de projetos de software livre, a complexidade estrutural traz uma consequência especialmente ruim. Num estudo com mais de 6000 projetos de software livre escritos em C e disponíveis no *Sourceforge*[1], Meirelles *et al* (Meirelles et al., 2010) descobriram uma correlação negativa entre complexidade estrutural e a atratividade dos projetos, que representa a sua capacidade de atrair usuários e desenvolvedores. Os projetos com maior complexidade estrutural atraíram menos usuários e também menos desenvolvedores. Para projetos de software livre que não possuem apoiadores corporativos e dependem apenas de desenvolvedores independentes para evoluir, este resultado demonstra a necessidade de manter a complexidade estrutural do sistema sob controle.

Resultados semelhantes aos descritos anteriormente também estão disponíveis para outras medidas de complexidade: Midha (Midha, 2008) estudou 450 projetos escritos em C e C++ disponíveis no *Sourceforge* e verificou que a complexidade de McCabe estava positivamente correlacionada com o número de defeitos e com o tempo levado para correção de defeitos, e negativamente correlacionada com o número de contribuições de novos desenvolvedores. Ainda que usando uma medida diferente da utilizada neste trabalho, esses resultados reforçam a percepção que que um aumento na complexidade em projetos de software é algo a ser evitado.

3.5 COMPORTAMENTO EVOLUCIONÁRIO DA COMPLEXIDADE ESTRUTURAL

Stewart *et al* (Stewart; Darcy; Daniel, 2006) estudaram 59 projetos de software livre escritos em Java disponíveis no *Sourceforge* e mediram a complexidade estrutural da forma proposta por Darcy *et al* (Darcy et al., 2005). Entre estes projetos, eles verificaram quatro diferentes padrões na evolução da complexidade estrutural: dois deles apresentavam tendência de crescimento ao final do período observado, enquanto outros dois apresentaram estabilização da complexidade estrutural. Nenhum dos padrões encon-

[1] O Sourceforge (`http://sourceforge.net/`) é um dos maiores sites de hospedagem de projetos de software livre.

trados, no entanto, apresentou tendência de diminuição da complexidade estrutural ao final do período de observação: aparentemente, a tendência natural da complexidade estrutural é sempre aumentar com o tempo, ou na melhor das hipóteses, se manter estável. Um estudo preliminar realizado como parte deste trabalho também identificou tendência de crescimento da complexidade estrutural em um projeto de software livre pequeno e escrito em C (Terceiro; Chavez, 2009).

A tendência de crescimento da complexidade não é uma observação recente: o trabalho seminal de Lehman sobre evolução de software já identificara este fenômeno, formalizado em sua segunda lei[2] da evolução de software, a "Lei da Complexidade Crescente" (Lehman; Belady, 1985; Lehman et al., 1997). Formulada no contexto de projetos de software proprietário, a segunda lei de Lehman sugere que na medida que um sistema de software evolui, a sua complexidade irá aumentar a não ser que sejam realizadas atividades específicas para evitar que isso aconteça.

De fato, existem projetos nos quais essas atividades específicas para conter o aumento da complexidade são realizadas. Darcy e colegas (Darcy; Daniel; Stewart, 2010) realizaram um segundo estudo sobre padrões de evolução da complexidade estrutural. Eles analisaram 108 projetos em C++ do *Sourceforge* e os agruparam conforme os seus diferentes padrões de evolução da complexidade estrutural. Desta feita, foram identificados 3 diferentes padrões de evolução: um grupo de projetos apresentou crescimento na complexidade, outro apresentou uma complexidade estável e o último uma diminuição na complexidade. No mesmo estudo, os projetos também foram agrupados de acordo com diferentes padrões de evolução de tamanho (em linhas de código), mas nenhum dos padrões identificados apresentava diminuição de tamanho. Este resultado sugere que ainda que um aumento no tamanho seja inevitável durante a evolução de sistemas de software, o aumento na complexidade não está necessariamente associado ao aumento do tamanho. Faz-se necessário, então, identificar fatores que influenciam a variação na complexidade estrutural, de forma que desenvolvedores e líderes de projetos sejam capazes de manipulá-los para manter a complexidade estrutural dos seus projetos sob controle.

3.6 CONCLUSÃO

Este capítulo apresentou os principais conceitos sobre complexidade estrutural utilizados neste trabalho. Inicialmente, foi feita uma analogia com o campo dos sistemas complexos. Em seguida, foi discutido o conceito de complexidade para sistemas de software. Além da complexidade estrutural, tratada no restante deste trabalho, a complexidade em sistemas de software pode ser encarada por dois outros pontos de vista: a complexidade do problema (ou complexidade natural) e a complexidade procedural.

Três medidas para complexidade de sistemas de software foram apresentadas e avaliadas, ao fim do que a medida de complexidade estrutural baseada em acoplamento e

[2]Ainda que as "leis de Lehman" não sejam propriamente leis científicas, a comunidade de Engenharia de Software convencionou chamá-las desta forma. Seria mais correto chamá-las de "hipóteses de Lehman", no entanto.

coesão proposta por Darcy *et al* foi selecionada para utilização neste trabalho. Resultados existentes associam esta medida a consequências negativas em projetos de software, e indicam que sua tendência evolutiva natural parece ser de crescimento, ou no melhor caso, de estagnação, mas nunca de redução.

O capítulo seguinte descreve fatores identificados como possíveis causas para a variação da complexidade estrutural em sistemas de software.

Capítulo

4

Este capítulo descreve fatores identificados como possíveis causas da variação na complexidade estrutural em projetos de software, classificados em três grupos: fatores relacionados aos desenvolvedores, fatores relacionados à manutenção realizada no sistema, e fatores relacionados à organização onde o sistema é desenvolvido. Por fim, são discutidos estudos relacionados, que investigam a influência destes fatores sobre outros atributos de qualidade de sistemas de software.

FATORES DE INTERESSE PARA O ESTUDO DA EVOLUÇÃO DA COMPLEXIDADE ESTRUTURAL

A complexidade estrutural é um dos fatores que influenciam a evolução de sistemas de software (Darcy et al., 2005; Midha, 2008; Meirelles et al., 2010; Darcy; Daniel; Stewart, 2010). A compreensão de *fatores que exercem influência sobre a complexidade estrutural* passa a ser um assunto de interesse para gerentes e líderes de projeto em geral, de forma que se possa buscar meios para controlá-los, mitigando o crescimento da complexidade estrutural e seus efeitos.

Durante o desenvolvimento deste trabalho, foi identificado um conjunto de fatores que podem influenciar a complexidade estrutural. Estes fatores foram então organizados em três grupos:

- *Fatores relacionados aos desenvolvedores envolvidos no projeto.* Uma vez que o desenvolvimento de software é uma atividade humana, mesmo com a adoção de padrões no processo de desenvolvimento, é de se esperar que características dos desenvolvedores que compõem a equipe influenciem na qualidade e, mais especificamente, na complexidade estrutural do código-fonte produzido.

  Neste grupo estão fatores como nível de participação dos desenvolvedores, experiência no projeto e grau de autoria sobre o código-fonte. A seção 4.1 discute fatores relacionados aos desenvolvedores.

- *Fatores relacionados à manutenção a qual o software é submetido.* Um sistema de software precisa ser alterado para se adaptar a novas necessidades de seus usuários,

para corrigir falhas existentes, para se adaptar a um novo ambiente de operação e mesmo para facilitar mudanças futuras. Ao longo desse processo, o sistema sofre alterações que se refletem em diversos atributos de qualidade interna, incluindo a complexidade estrutural. Desta forma, esperamos que fatores relacionados às mudanças realizadas no sistema possuam uma influência sobre a complexidade estrutural.

Neste grupo estão fatores como variação de tamanho do sistema causado por uma mudança, difusão de mudança, tipo de manutenção realizada, etc. A seção 4.2 discute fatores relacionados ao processo de manutenção.

- *Fatores relacionados à organização.* Além das características dos desenvolvedores, esperamos que o ambiente organizacional sob o qual um sistema é desenvolvido também possua alguma influência sobre a complexidade estrutural.

  Neste grupo estão fatores como maturidade do processo de desenvolvimento, estrutura organizacional, etc. A seção 4.3 discute fatores relacionados à organização.

A seção 4.4 discute trabalhos relacionados, nos quais fatores apresentados neste capítulo são explorados. A seção 4.5 conclui o capítulo com um resumo do que foi apresentado nas seções anteriores.

4.1 FATORES RELACIONADOS AOS DESENVOLVEDORES

4.1.1 Experiência no Projeto

A *experiência do desenvolvedor* é abordada na literatura técnica em termos de duas dimensões distintas. A primeira dimensão está associada com a experiência prévia de trabalho dos desenvolvedores: eles são caracterizados em termos de "anos de experiência", ou de "nível de habilidade", normalmente representadas em escalas ordinais como "baixo, médio ou alto" ou em escalas numéricas simples como "de 1 a 5".

Esta dimensão de experiência já foi usada, por exemplo, para predizer produtividade de projetos (Banker; Datar; Kemerer, 1991), esforço de manutenção (Darcy et al., 2005) e atributos de qualidade de software (Beaver; Schiavone, 2006). Modelos de capacidade de desenvolvimento de software como SPI e CMMI são criticados por não levarem em conta aspectos de experiência prática anterior (Steen, 2007).

Uma segunda dimensão de experiência dos desenvolvedores está relacionada com sua atividade em um dado projeto. Ao invés de utilizar medidas subjetivas e estáticas como descrito anteriormente, esta dimensão é caracterizada pela contagem de mudanças que um desenvolvedor realiza no projeto. Considera-se que, de forma geral, a experiência do desenvolvedor naquele projeto aumenta na medida em que ele realiza mudanças.

A experiência de um desenvolvedor no contexto de um projeto pode ser medida com o uso de informações de *commits* em sistemas de controle de versão, em termos de (i)

número de *commits* realizados pelo desenvolvedor (Matsumoto et al., 2010) e (ii) número de dias desde o primeiro *commit* realizado pelo desenvolvedor naquele projeto (Zhou; Mockus, 2010).

A informação sobre experiência dos desenvolvedores no projeto pode ser usada, por exemplo, para identificar especialistas em determinadas partes de um projeto como módulos ou subsistemas individuais (Mockus; Herbsleb, 2002), para assegurar que equipes de manutenção possuem ao menos um membro com conhecimento prévio em um dado projeto (Banker; Datar; Kemerer, 1991), ou como uma variável preditora para a ocorrência de defeitos (Mockus; Weiss, 2000; Matsumoto et al., 2010).

4.1.2 Grau de Autoria

Um aspecto importante da participação de um desenvolvedor num projeto de desenvolvimento de software diz respeito às partes do projeto com as quais ele interage. A partir de informações sobre mudanças armazenadas em um sistema de controle de versão, é possível determinar a parcela de autoria de um desenvolvedor com relação a qualquer arquivo do projeto.

O *grau de autoria* de um desenvolvedor com relação a um elemento do sistema (Mockus; Herbsleb, 2002; Fritz et al., 2010) é uma medida que indica tal parcela, de forma que se possa caracterizar um desenvolvedor em termos dos elementos sobre os quais ele tem mais conhecimento, ou mesmo comparar dois desenvolvedores em termos da probabilidade de terem conhecimento sobre um determinado elemento.

Mockus e Herbsleb (Mockus; Herbsleb, 2002) utilizaram informações de autoria no Expertise Browser para identificar *experts* em determinadas partes do código. Um dos problemas com a abordagem utilizada nesse trabalho, no entanto, foi apontada por Fritz e colegas (Fritz et al., 2010): a função de *expertise* para Mockus e Herbsleb era uma função crescente monotônica, isto é, a *expertise* de um desenvolvedor com relação a um determinado módulo sempre aumenta na medida que ele altera este módulo. Suponha que um desenvolvedor A possui uma alta *expertise* sobre um determinado módulo, e que um desenvolvedor B reescreva todo o código daquele módulo. De acordo com a métrica proposta por Mockus e Herbsleb, assim que A voltasse a modificar aquele módulo, sua *expertise* naquele módulo seria ainda maior do que antes dele ser reescrito por B.

A proposta de Fritz e colegas (Fritz et al., 2010) para calcular o grau de autoria resolve esta questão: na medida em que um desenvolvedor altera um módulo, o grau de autoria dos desenvolvedores que haviam alterado o mesmo módulo anteriormente irá decair. Fritz e colegas utilizaram o grau de autoria em conjunto com informações sobre interação.

O *grau de conhecimento* (*DOK – degree of knowledge*) de um desenvolvedor com relação a um determinado elemento do código (arquivo, classe, método) é determinado da seguinte forma:

$$DOK = \alpha_{FA} \times FA + \alpha_{DL} \times DL + \alpha_{AC} \times AC + \beta_{DOI} \times DOI$$

Na equação acima, FA representa *autoria inicial* (*first authorship*), que pode ser 1 ou 0, caso o desenvolvedor seja (ou não seja, respectivamente) o autor original do elemento em questão. DL (*deliveries*) representa o número de vezes que o desenvolvedor alterou o elemento em questão. AC (*acceptances*) representa o número de alterações realizadas por outros desenvolvedores no elemento em questão. DOI representa o *grau de interação* (*degree of interaction*), medido através do monitoramento das atividades dos desenvolvedores com suporte de um ambiente de desenvolvimento instrumentado para isso.

Uma das características deste modelo é que, para aplicá-lo na sua totalidade, é necessária a obtenção do grau de interação, o que só é possível em estudos onde se tenha acesso aos desenvolvedores, e principalmente, onde se possa instrumentar o seu ambiente de desenvolvimento. Por outro lado, ao realizarem uma regressão linear deste modelo a partir de graus de conhecimento determinados através de entrevistas com desenvolvedores, Fritz e colegas verificaram que apenas os termos ligados à autoria (FA, DL e AC) apresentaram coeficientes significativos estatisticamente. Desta forma, na prática o modelo obtido por Fritz et al. pode ser simplificado para:

$$DOK' = 1.098 \times FA + 0.164 \times DL - 0.321 \times ln(1 + AC)$$

Pode-se notar que o coeficiente associado a AC é negativo, o que confirma a noção de que o grau de conhecimento sobre um elemento do sistema – e portanto, a *expertise* do desenvolvedor com relação aquele elemento – diminui na medida que aumenta o número de alterações realizadas por outros desenvolvedores naquele elemento.

Com tal modelo simplificado, que elimina a necessidade de se obter o grau de interação, o grau de conhecimento de um desenvolvedor sobre um determinado elemento do sistema pode ser medido em estudos que apenas contemplem dados obtidos de sistemas de controle de versão.

Desse modo, torna-se viável a utilização do grau de autoria como um *proxy* para o grau de conhecimento em estudos experimentais quantitativos em larga escala.

A informação sobre autoria pode ser utilizada, por exemplo, para identificar experts em determinadas áreas do projeto (Mockus; Fielding; Herbsleb, 2002) ou para identificar quais desenvolvedores podem ter interesse em revisar mudanças a depender de quais módulos são alterados (Fritz et al., 2010). Nós esperamos que também se possa utilizar informações de autoria para predizer variações na complexidade estrutural de projetos de software.

4.1.3 Nível de Participação

O *nível de participação* de um desenvolvedor em um projeto de software livre indica a posição social que o desenvolvedor tem naquele projeto. O nível de participação é determinado pela quantidade de contribuições que o desenvolvedor realiza, e está associado ao poder de decisão que o desenvolvedor possui sobre o projeto.

O "modelo cebola" (Mockus; Fielding; Herbsleb, 2002; Crowston; Howison, 2005), ilustrado na figura 4.1, se tornou uma representação amplamente aceita do que acontece num projeto de software livre, ao indicar a existência de níveis concêntricos de contribuição. Um pequeno grupo de desenvolvedores centrais ou do núcleo (*core*) do projeto realizam a maior parte do trabalho. Um grupo um pouco maior faz contribuições diretas, porém menos frequentes, na forma de correções de defeitos, documentação, etc. Um grupo ainda maior relata defeitos encontrados durante o uso do software. Por fim, um grupo ainda maior é composto pelos usuários silenciosos, que apenas utilizam o software mas nunca fornecem nenhum tipo de contribuição ou *feedback* para o projeto.

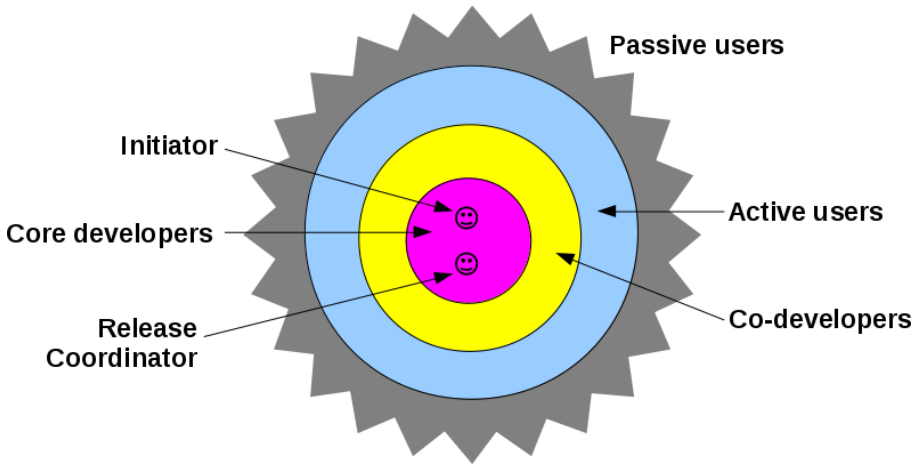


Figura 4.1 O "modelo cebola". Adaptado de (Crowston; Howison, 2005).

Numa cultura meritocrática como a de projetos de software livre, uma maior centralidade dentre as camadas da "cebola" implica maior respeito por parte da comunidade, e consequentemente maior poder de influência sobre o projeto. Mesmo que o projeto seja composto majoritariamente por desenvolvedores independentes, sem obrigações contratuais com relação à coordenação do projeto, os membros mais centrais possuem a capacidade de determinar os rumos do projeto e influenciar o trabalho dos demais.

O processo pelo qual participantes de um projeto migram de um grupo para outro é diferente em cada projeto: algumas comunidades adotam um processo formal com procedimentos explícitos para a obtenção de posições mais centrais; outras comunidades possuem um processo informal, onde desenvolvedores mais centrais simplesmente oferecem acesso aos recursos do projeto a novos desenvolvedores que se destacam. Independente do procedimento específico utilizado por cada comunidade, um elemento comum é que a

conquista de papéis centrais se dá com base no mérito: um desenvolvedor se torna um líder através da sucessão de contribuições de valor para a comunidade (Jensen; Scacchi, 2007).

Como a maior parte do trabalho no projeto é realizado pela equipe central (*core team*), a manutenção de uma equipe central ativa é uma questão importante para a qualidade e sobrevivência dos projetos. Alguns projetos conseguem manter sua equipe central com pouca ou nenhuma mudança durante toda a sua história, enquanto outros acabam passando por uma sucessão de diferentes gerações de desenvolvedores centrais (Robles; Gonzalez-barahona, 2006; Robles; Gonzalez-barahona; Herraiz, 2009).

O relacionamento entre membros centrais e periféricos de uma comunidade por vezes é conturbado: existem situações em que os desenvolvedores centrais tendem a satisfazer suas próprias demandas e dar pouca atenção às demandas da periferia (Dalle; Besten; Masmoudi, 2008; Masmoudi et al., 2009). Individualmente, membros do núcleo e da periferia podem também exibir comportamentos distintos durante a discussão de assuntos importantes para o projeto (Scialdone et al., 2009) e durante as atividades de relato e triagem de defeitos (Masmoudi et al., 2009).

4.2 FATORES RELACIONADOS À MANUTENÇÃO

4.2.1 Variação de Tamanho

O *tamanho do software* é um atributo que tem sido recorrentemente observado por pesquisadores e engenheiros de software. Muitos acreditam que há uma relação entre tamanho do software e sua complexidade. Parnas (Parnas, 1994) menciona o tamanho do sistema como um dos fatores que tornam sistemas difíceis de alterar e evoluir. Em seus estudos iniciais sobre evolução de software, Lehman incluiu entre suas leis de evolução de software tanto "Crescimento Contínuo" como "Complexidade Crescente" (Lehman et al., 1997). A evidência existente de que a complexidade ciclomática (Mccabe, 1976) possui uma forte correlação com o tamanho (Meulen; Revilla, 2007; Jay et al., 2009; Herraiz; Hassan, 2011) fortalece essa corrente de pensamento.

Apesar de ser criticada, o número de linhas de código (*LOC*, de *lines of code*) tem sido uma das medidas mais utilizadas para tamanho de sistemas de software. Há várias formas de contar linhas de código, e normalmente ignora-se linhas em branco e comentários.

Para medir a variação no tamanho de um sistema de software, pode-se usar as informações de mudanças registradas nos sistemas de controle de versão e determinar a diferença entre o tamanho total do sistema depois da mudança e o seu tamanho antes da mudança. Desta forma, a variação no tamanho pode ser positiva, se a mudança acarretou no crescimento do sistema, ou negativa, no caso em que a mudança fez o sistema diminuir de tamanho.

4.2.2 Difusão de Mudança

A difusão de uma mudança representa a abrangência do seu escopo, e de certa forma é um indicativo da sua complexidade. Existem diversas formas possíveis de se medir difusão de mudança: o número de subsistemas alterados, o número de módulos alterados, ou o número de arquivos alterados.

A difusão da mudança aumenta na medida que aumentam o número de elementos diferentes envolvidos na mudança. Ao analisar o histórico de mudanças de qualquer projeto de software, pode-se notar que existe uma grande variação na difusão de uma mudança para outra: algumas mudanças são pontuais, afetando poucas linhas de código em um único módulo; outras mudanças podem afetar diversos módulos diferentes.

Uma alta difusão de mudança pode indicar de que o *design* atual do software não comporta aquela mudança de uma forma modular, e por isso é necessário alterar uma grande quantidade de módulos diferentes para atingir o resultado desejado.

Como a cognição humana tem limites (Miller, 1956; Gigerenzer; Selten, 2002), uma mudança que envolve uma quantidade de módulos maior do que aquela com a qual um desenvolvedor pode lidar ao mesmo tempo exige um maior esforço cognitivo por parte dos desenvolvedores. Isso poderia acarretar, por exemplo, num maior risco de violação de decisões arquiteturais ou de introdução de defeitos.

De fato, a difusão de mudança se mostrou como um bom preditor para defeitos em sistemas de software. Mockus e Weiss relataram que mudanças mais difusas entre os módulos do sistema incorreram numa maior probabilidade de introdução de defeitos (Mockus; Weiss, 2000).

Hassan propôs medidas para a complexidade de mudanças baseadas na sua entropia, um conceito similar à difusão de mudança, que superaram outras métricas usadas anteriormente, como número de modificações prévias e número de defeitos prévios, na previsão de defeitos (Hassan, 2009).

Intuitivamente, é de se esperar que mudanças que afetam mais módulos também aumentem o risco de que ao modificar um *design* de software existente, os desenvolvedores acabem tornando-o mais complexo. Por outro lado, a recíproca pode também ser verdadeira: talvez um *design* mais complexo leve a mudanças mais difusas.

4.2.3 Tipo de Mudança

As atividades de manutenção num sistema de software possuem diferentes naturezas. Por exemplo, a manutenção pode ter o objetivo de corrigir um defeito ou de adaptar o software a mudanças ocorridas no seu ambiente de operação.

Swanson, Lientz e Tompkins (Swanson, 1976; Lientz; Swanson; Tompkins, 1978) forneceram as bases para a categorização de atividades de manutenção, categorizando-as em corretiva, adaptativa e perfectiva. Esta classificação foi adotada pela norma ISO/IEC

14764 e expandida da seguinte forma (Abran et al., 2004):

- *Manutenção corretiva*: realizada para corrigir problemas descobertos após o lançamento de um produto de software. É uma atividade desempenhada de forma reativa.
- *Manutenção adaptativa*: realizada para manter um produto de software usável num ambiente sujeito a mudanças.
- *Manutenção perfectiva*: realizada para melhorar o desempenho ou manutenibilidade.
- *Manutenção preventiva*: realizada para detectar e corrigir defeitos latentes antes que eles desencadeiem falhas.

Kitchenham e colegas argumentam que essa classificação é baseada no motivo que levou à mudança, e que ela não leva em conta características objetivas da mudança (Kitchenham et al., 1999). Desta forma eles propõem a seguinte classificação:

- *Correções*: mudanças que corrigem defeitos, isto é, discrepâncias entre o comportamento esperado do sistema e o que é observado. Correções não alteram os requisitos nem a implementação do sistema.
- *Melhorias*: mudanças que alteram o comportamento ou a implementação do sistema. Melhorias podem ser ainda divididas em melhorias que alteram requisitos existentes, melhorias que adicionam novos requisitos, e melhorias que alteram a implementação mas não os requisitos.

Esta classificação alternativa possui a vantagem de ser mais fácil de ser posta em prática em estudos observacionais com dados históricos de repositórios de engenharia de software. Por exemplo, diversos sistemas de gestão de atividades armazenam explicitamente esse tipo de informação para cada requisição de mudança.

Espera-se que o tipo de mudança influencie a variação na complexidade estrutural. Correções de defeitos provavelmente não afetam a complexidade estrutural. Já melhorias provavelmente afetam, pois podem envolver a adição de novos módulos e a modificação substancial de módulos existentes.

4.3 FATORES RELACIONADOS À ORGANIZAÇÃO

Neste trabalho, o termo "organização" é utilizado num sentido amplo, abrangendo coletivos de pessoas que colaboram para a construção de um sistema de software. Exemplos de organizações são empresas de desenvolvimento de software e projetos de software livre. As subseções a seguir apresentam aspectos de organizações de desenvolvimento de software que podem ter influência sobre a complexidade estrutural de sistemas de software.

4.3.1 Nível de maturidade

Assim como os desenvolvedores ganham experiência durante o decorrer de um projeto de software, as organizações também evoluem na medida em que realizam projetos, acumulando práticas e conhecimentos que são aproveitados de um projeto para o outro.

O *nível de maturidade* de uma organização desenvolvedora de software indica a presença de alguns elementos considerados importantes para a obtenção de produtos de software de qualidade, como por exemplo a sistematização do processo de desenvolvimento, a presença de boas práticas e a instituição de um programa de melhoria do processo de desenvolvimento.

Existem diferentes modelos para aferição do nível de maturidade no desenvolvimento de software. Internacionalmente, o principal modelo é o CMMI (*Capability Maturity Model Integration*) (CMMI Product Team, 2010). As organizações podem ser certificadas desde "CMMI nível 1", a certificação básica, até "CMMI nível 5", o nível mais alto de maturidade neste modelo.

O MPS-BR (Melhoria do Processo de Software Brasileiro) (Softex, 2011) é uma iniciativa para fornecer um modelo de maturidade mais condizente com a realidade específica da indústria de software brasileira. Assim como o CMMI, ele apresenta uma escala crescente de níveis de maturidade dentro da qual as organizações podem ser certificadas.

A partir da percepção de que projetos de software livre são organizações substancialmente diferentes das organizações desenvolvedoras de software para as quais modelos como CMMI e MPS-BR foram desenvolvidos, o projeto QualiPSo desenvolveu o OMM (*Open Source Maturity Model* (QualiPSo Project, 2008)). O OMM tem o objetivo de fornecer um modelo de maturidade adaptado à realidade de projetos de software livre, contendo elementos específicos do desenvolvimento de software livre e deixando de lado aspectos que apenas se aplicam a organizações privadas de desenvolvimento de software.

É de se esperar que mais altos níveis de maturidade incorram em software com menos complexidade estrutural. Na medida em que alcançam níveis mais altos de maturidade, as organizações podem tomar medidas para melhoria do produto e do processo que beneficiem a manutenibilidade dos produtos. Em especial, espera-se que impeçam o aumento explosivo da complexidade estrutural.

4.3.2 Estrutura organizacional

Na década de 60, Melvin Conway cunhou o que hoje é conhecido como a "Lei de Conway"[1] (Conway, 1968):

> Qualquer organização que projete um sistema ... irá produzir um projeto cuja estrutura é uma cópia da estrutura de comunicações da organização.

[1] A "Lei de Conway" também não é uma lei científica propriamente dita.

Em outras palavras, Conway propôs que a organização de um sistema de software tende a imitar a estrutura organizacional na qual ele é desenvolvido. Brooks (Brooks.jr, 1995) defendeu que a estrutura organizacional afeta fortemente a qualidade do produto.

Nagappan, Murphy e Basili propuseram um conjunto de métricas relacionadas à estrutura organizacional envolvida no desenvolvimento de um produto de software. Estas métricas incluiam, por exemplo, número de desenvolvedores envolvidos, número de desenvolvedores que haviam participado do desenvolvimento e que deixaram a organização e número de diferentes organizações que haviam contribuído com o produto. Num estudo de caso com dados relativos ao desenvolvimento do Windows Vista, eles descobriram que todas as oito métricas propostas contribuíam para a predição de defeitos pós-lançamento, que alcançou altos níveis de precisão e revocação (Nagappan; Murphy; Basili, 2008). O modelo obtido obteve uma desempenho melhor do que outros modelos baseados em métricas de código-fonte, número de mudanças prévias e número de defeitos prévios.

Também esperamos que a estrutura organizacional na qual um sistema de software é desenvolvido influencie a sua complexidade estrutural. Em especial, se a lei de Conway estiver certa, a complexidade da estrutura de comunicação de uma organização deve refletir-se na complexidade do sistema produzido por ela.

4.4 ESTUDOS RELACIONADOS

Esta seção apresenta brevemente alguns trabalhos relacionados, divididos em subseções que os agrupam em tópicos que consideramos associados ao tema deste capítulo. Assim como neste tese, nos estudos apresentados a seguir, é investigada a influência dos fatores discutidos neste capítulo sobre atributos de qualidade de sistemas de software.

4.4.1 Impacto de fatores organizacionais sobre atributos de qualidade de software

Capra e colegas realizaram um estudo quantitativo com 75 projetos de software livre e chegaram à conclusão de que projetos com uma estrutura de governança mais aberta exibiam melhor qualidade de design, definida pelos autores em termos de métricas de projeto orientado a objetos. Por um lado, argumentam os autores, módulos menos acoplados permitem que diferentes desenvolvedores trabalhem em áreas diferentes do projeto sem a necessidade de coordenação explícita. Por outro lado, uma estrutura de governança mais aberta oferece aos desenvolvedores mais liberdade para melhorar a qualidade do *design*, uma vez que eles não precisam se ater a prazos estritos ou outros tipos de pressão por parte de um gerente superior ou de clientes (Capra; Francalanci; Merlo, 2008).

Bargallo e colegas realizaram um estudo com 56 projetos de software livre, e relataram que a popularidade de um projeto leva a um *design* de pior qualidade. Eles modelaram "popularidade" em termos de número de *downloads*, tráfego no *website* do projeto e volume de atividades de desenvolvimento, e "qualidade de design" em termos de métricas de projeto orientado a objetos. Os autores sugerem que na medida que um projeto de torna

popular, os líderes do projeto podem ter que redirecionar seus esforços de programação para outras atividades, como responder usuários em fóruns e listas de discussão, revisar contribuições, etc (Barbagallo; Francalenei; Merlo, 2008). O afastamento dos desenvolvedores principais do projeto da atividade de desenvolvimento pode levar a uma perda de integridade conceitual (Brooks.jr, 1995).

4.4.2 Caracterização de mudanças em projetos de software

William e Carver realizaram uma revisão sistemática com o objetivo de determinar os tipos de mudanças que impactam na arquitetura de software (Williams; Carver, 2010). Com base nos resultados, eles propuseram um esquema para caracterização de mudanças na arquitetura de software. Nesse esquema, as mudanças são avaliadas em termos de diferentes características, de forma a guiar o desenvolvedor no processo de decisão sobre a viabilidade da mudança e na previsão dos seus impactos (Williams; Carver, 2010). Algumas das características incluídas no esquema de caracterização de mudanças proposto pelos autores coincidem com os fatores utilizados neste trabalho.

4.4.3 Impacto de mudanças sobre atributos de qualidade

Mockus e Weiss propuseram um modelo para predizer o risco de mudanças em projetos de software incorrerem em defeitos, utilizando como preditores propriedades das próprias mudanças (Mockus; Weiss, 2000). Estes preditores incluíram diversos fatores que também foram discutidos neste trabalho, como variação de tamanho, difusão da mudança, experiência dos desenvolvedores, e tipo de mudança (correções sobre código existente ou código novo). Os autores aplicaram este modelo a mudanças de software numa empresa, e concluíram que difusão de mudança e experiência do desenvolvedor são essenciais para predizer falhas (Mockus; Weiss, 2000).

Hassan propôs medidas para a complexidade de mudanças baseadas na entropia, um conceito similar à difusão de mudança. Em um estudo de caso com dados históricos de 6 projetos de software livre de grande porte, foi verificado que essas métricas superavam outras métricas usadas anteriormente, como número de modificações prévias e número de defeitos prévios, na previsão de defeitos (Hassan, 2009).

4.4.4 Impacto de fatores humanos sobre atributos de qualidade de software

O estudo de Mockus e Weiss mencionado anteriormente concluiu que a experiência dos desenvolvedores era um dos fatores essenciais para predizer falhas (Mockus; Weiss, 2000).

Ostrand e colegas realizaram um estudo para verificar se informações sobre quais desenvolvedores modificaram quais arquivos podem ser usadas para melhorar predição de defeitos. Eles concluíram que o número de desenvolvedores diferentes que já alteraram um dado arquivo pode ajudar a melhorar predições de defeitos (Ostrand; Weyuker; Bell,

2010).

Matsumoto e colegas estudaram os efeitos de caracteríticas de desenvolvedores na confiabilidade de projetos de software. Eles analisaram o relacionamento entre o número de falhas e métricas de desenvolvedores como "número de *commits* por desenvolvedor", "número de desenvolvedores por módulo". Os autores concluíram que módulos alterados por mais desenvolvedores continham mais falhas, e que o uso de métricas relacionadas aos desenvolvedores melhoravam o desempenho de modelos de predição (Matsumoto et al., 2010).

4.5 CONCLUSÃO

Este capítulo apresentou um conjunto de fatores que podem influenciar a complexidade estrutural em sistemas de software, divididos em três grupos: fatores relacionados aos desenvolvedores, fatores relacionados à manutenção realizada no sistema, e fatores relacionados à organização de desenvolvimento de software.

Foi apresentada uma série de estudos relacionados, nos quais é investigada a influência destes fatores sobre atributos de qualidade de software.

O capítulo seguinte descreve os estudos realizados durante o desenvolvimento deste trabalho, nos quais investigou-se a influência dos fatores relacionados aos desenvolvedores e dos fatores relacionados à manutenção realizada sobre a variação da complexidade estrutural em projetos de software livre.

Capítulo 5

Este capítulo descreve os estudos experimentais realizados durante o desenvolvimento deste trabalho. São apresentados quatro estudos e é feita uma discussão geral sobre os resultados obtidos, em termos dos fatores que influenciam a complexidade estrutural em sistemas de software livre.

ESTUDOS EXPERIMENTAIS REALIZADOS

Para validar a teoria proposta no capítulo 6, foi realizada uma série de estudos experimentais. Nestes estudos, a complexidade estrutural foi utilizada como variável dependente e diversos dos fatores descritos no capítulo 4 foram utilizados como variáveis independentes.

Este capítulo descreve estes estudos, e está organizado como segue. A seção 5.1 tece considerações gerais sobre os estudos. A seção 5.2 descreve um estudo exploratório inicial sobre a evolução da complexidade estrutural em um projeto de software livre pequeno. A seção 5.3 apresenta um estudo sobre a influência do nível de participação (seção 4.1.3) na variação da complexidade estrutural. A seção 5.4 descreve um estudo sobre a influência da experiência do desenvolvedor no projeto (seção 4.1.1), da variação de tamanho (seção 4.2.1) e da difusão de mudaça (seção 4.2.2) sobre a variação da complexidade estrutural. A seção 5.5 contém um estudo que expande o anterior na busca de melhores modelos para a variação da complexidade estrutural. O estudo inclui os componentes do grau de autoria (seção 4.1.2) e melhora a definição operacional da difusão de mudança (seção 4.2.2). Por fim, a seção 5.6 conclui este capítulo e apresenta uma discussão sobre os fatores que foram identificados nos estudos como relevantes com relação à variação da complexidade estrutural nos projetos analisados.

5.1 CONSIDERAÇÕES GERAIS

A seguir descrevemos pontos em comum entre os diversos estudos apresentados neste capítulo.

5.1.1 Cálculo da complexidade estrutural

Nos estudos realizados, a complexidade estrutural foi calculada conforme a proposta de Darcy, Kemerer, Slaughter e Tomayko (Darcy et al., 2005). Desta forma, a complexidade estrutural de um sistema de software é definida como:

$$SC(p) = \frac{\sum_{m \in M(p)} A(m) \times FC(m)}{|M(p)|}$$

Na equação acima, $M(p)$ representa o conjunto de módulos de um projeto p, $A(m)$ representa uma medida de acoplamento do módulo m, e $FC(m)$ representa uma medida de falta de coesão.

Para medir acoplamento, utilizamos a métrica CBO proposta por Chidamber e Kemerer (Chidamber; Kemerer, 1994). O valor de CBO para um dado módulo é calculado como o número de módulos dos quais o módulo em questão depende.

Para medir falta de coesão, utilizamos LCOM4 proposta por Hitz e Montazeri (Hitz; Montazeri, 1995). O valor de LCOM4 para um módulo é igual ao número de componentes conexos de um grafo não dirigido onde os vértices são as sub-rotinas (métodos, funções, etc) de um módulo, e as arestas indicam que duas sub-rotinas usam pelo menos um atributo ou variável em comum, ou que uma das sub-rotinas chama a outra. Estes componentes conexos representam partes independentes de um módulo, ou seja, que implementam responsabilidades distintas.

Para linguagens orientadas a objeto como C++ e Java, classes são considerads como módulos. Para C, pares de arquivos cabeçalho/implementação (`.h`/`.c`) são considerados como um módulo. Ainda que as métricas utilizadas para acoplamento e coesão tenham sido definidas no contexto da orientação a objetos, consideramos que elas podem ser aplicadas no contexto de programas em C com as devidas adaptações, considerando as funções e variáveis de um módulo como análogas a métodos a atributos de uma classe.

5.1.2 A ferramenta Analizo

Durante o desenvolvimento deste trabalho, identificamos a necessidade de realizar análise de código fonte para extração de métricas. As ferramentas encontradas que estavam disponíveis sob licenças de software livre ou suportavam uma única linguagem de programação ou estavam obsoletas, que dificultava a sua utilização. Identificamos a necessidade de realizar análise de código fonte em programas escritos em diferentes linguagens de programação. Por isso, foi desenvolvida uma ferramenta para realizar análise de código fonte chamada de Analizo. Analizo é uma suíte de ferramentas de análise e visualização de código fonte que suporta diferentes linguagens de programação. Analizo foi utilizada para extração de dados em todos os estudos experimentais realizados. Esta seção descreve

brevemente a evolução da Analizo em função da realização dos estudos experimentais.

Analizo foi baseada num programa chamado `egypt`, desenvolvido por Andreas Gustafsson[1]. `egypt` era um programa monolítico que lia arquivos intermediários gerados pelo processo de compilação com o compilador GNU C e produzia grafos de dependência entre funções.

Para a realização do primeiro estudo experimental (seção 5.2), `egypt` foi estendido para detectar uso de variáveis dentro de funções, o que possibilitava calcular métricas de coesão. Além disso, o programa original foi refatorado para um *design* orientado a objetos de forma que diferentes estratégias de extração de dados a partir do código fonte pudessem ser implementadas.

Joenio Costa (Universidade Católica do Salvador) implementou um extrator baseado no Doxygen[2], um sistema para documentação de software cujo *parser* padrão suporta código-fonte C, C++ e Java (Costa, 2009). Isso possibilitou analisar projetos sem a necessidade de compilar o seu código fonte, o que traz duas vantagens. Primeiro, o processo de extração passou a exigir menos tempo. Segundo, não ter que compilar o código possibilita a análise de código legado cuja compilação não é mais possível, o que acontece quando o código depende de versões obsoletas do compilador ou de versões de bibliotecas que não estejam mais disponíveis.

A partir desse ponto, não fazia mais sentido continuar chamando a ferramenta pelo nome original, uma vez que praticamente nenhum código do `egypt` havia restado. Foi então escolhido o nome Analizo, que significa "análise" em esperanto.

Para o segundo estudo experimental (seção 5.3), Luiz Romário Rios (UFBA) colaborou com a criação de programas para automatizar o processo de invocar Analizo em todas as versões disponíveis em um repositório de controle de versão. Estes programas foram reunidos num pacote chamado de `analizo-utils`. Apesar da adoção do *parser* do Doxygen, que teoricamente suportaria C, C++ e Java, havia um defeito na interface da Analizo com o Doxygen que impedia a correta análise de código orientado a objetos. Por essa razão, o estudo descrito na seção 5.3 precisou ser realizado apenas com projetos escritos em C.

Na mesma época, Paulo Meirelles, João Miranda e Lucianna Almeida (IME-USP) começaram a colaborar no desenvolvimento de Analizo, em especial, na implementação de novas métricas e no desenvolvimento de um novo extrator para determinação do tamanho em linhas de código com a ajuda de uma ferramenta chamada `sloccount`[3].

O defeito na interface com o Doxygen foi resolvido, de forma que para o estudo apresentado na seção 5.4, foi possível analisar projetos escritos em C, C++ e Java. Além disso, os programas auxiliares presentes no pacote `analizo-utils` foram incorporados à Analizo, oficializando o suporte à mineração de dados em repositórios de controle de

[1] `http://www.gson.org/egypt/`
[2] `http://www.doxygen.org/`
[3] `http://www.dwheeler.com/sloccount/`

versão. Nesse cenário, a saída disponível para o processo de mineração de dados era uma única tabela, que continha uma linha para cada versão analisada. Cada uma destas linhas continha métricas globais relativas àquela versão (por exemplo, tamanho em linhas de código, número de módulos), e métricas de módulo agregadas (por exemplo, acoplamento médio, falta de coesão média, e complexidade estrutural média – que era nossa variável dependente de interesse no estudo). Apesar de viabilizar o estudo, esta saída era ainda inadequada para diversos outros tipos de estudos.

Para o estudo descrito na seção 5.5, foram feitas duas grandes mudanças na Analizo. Primeiro, foi adicionado suporte ao armazenamento dos dados obtidos mediante mineração de repositórios em um banco de dados relacional. Isso permitiu armazenar informações individualizadas dos módulos do projeto e, desse modo, realizar uma análise muito mais rica da evolução do projeto, uma vez que se pode analisar a evolução das métricas relativas a cada módulo individualmente. Segundo, foi adicionado suporte à utilização de múltiplos processadores. Esta mudança representou um passo importante para a realização de estudos em larga escala. Uma vez que o processamento de repositórios grandes pode levar horas ou mesmo dias, utilizar toda a capacidade de computadores com mais de um processador – que na atualidade já são a regra e não mais a exceção – reduz significativamente o tempo necessário para analisar estes repositórios.

O apêndice D reproduz um artigo publicado na sessão de ferramentas do I Congresso Brasileiro de Software (Terceiro et al., 2010), onde é feita uma descrição detalhada da Analizo à época. Mais informações sobre a ferramenta podem ser encontradas no seu *website*, em `http://analizo.org/`.

5.1.3 Reproducibilidade dos estudos

As seguintes medidas foram tomadas para garantir a reproducibilidade dos estudos experimentais apresentados neste capítulo:

- Todo código fonte e *datasets* utilizados nos estudos estão disponíveis no *website* `http://deb.li/tt` para download. Isto inclui o código fonte utilizado para extração e análise estatística dos dados, geração de gráficos, etc, bem como os arquivos de dados obtidos após o processamento dos repositórios dos projetos e o código fonte dos artigos resultantes.

- Todas as ferramentas de software utilizadas são livres e estão disponíveis para download gratuito na internet.

- Todos os projetos analisados têm seus repositórios de controle de versão públicos e disponíveis na internet.

- Nos artigos produzidos, procurou-se descrever com o máximo de detalhe possível os procedimentos utilizados nos estudos.

5.2 EVOLUÇÃO DA COMPLEXIDADE EM UM PROJETO DE SOFTWARE LIVRE

O primeiro estudo realizado foi um estudo exploratório com dois objetivos principais:

- Experimentar uma abordagem para o estudo da evolução da complexidade estrutural em projetos de software que possa ser usada por desenvolvedores em seus próprios projetos. Isto incluía realizar uma prova de conceito de Analizo.
- Obter resultados iniciais sobre a evolução da complexidade estrutural num projeto de software livre escrito em C, de forma a complementar os resultados obtidos no estudo de Stewart e colegas (Stewart; Darcy; Daniel, 2006), realizados com projetos de software escritos em Java.

Para isso, foi realizado um estudo de caso com a evolução de um projeto de software livre chamado Ristretto. Ristretto é um visualizador de imagens, parte do ambiente *desktop* Xfce; foi escrito em C e usa o *toolkit* de interface gráfica GTK+.

Este estudo foi publicado nos anais do *Joint Workshop on Quality and Architectural Concerns in Open Source Software (QACOS) and Open Source Software and Product Lines (OSSPL) – QACOS/OSPL 2009* com o título "*Structural Complexity Evolution in Free Software Projects: A Case Study*" (Terceiro; Chavez, 2009). O apêndice A contém o texto na íntegra.

5.2.1 Hipóteses

Este estudo investigou duas hipóteses com relação à evolução do projeto Ristretto:

H_1 : o tamanho do código fonte aumenta com a passagem do tempo

H_2 : a complexidade estrutural do projeto aumenta com a passagem do tempo.

5.2.2 Projeto experimental

A população do estudo experimental incluiu 21 *releases* do projeto, lançadas durante um período de 15 meses. Para cada release do projeto, foram extraídas as seguintes variáveis:

- *RD* – dia do lançamento da versão (*release day*). Esta é a variável independente.
- *SLOC* – tamanho do projeto em linhas de código.
- *SC* – complexidade estrutural média, conforme (Darcy et al., 2005). No estudo publicado, esta variável foi originalmente chamada "CplXLCoh", que era o nome utilizado no estudos do grupo proponente (Darcy et al., 2005; Stewart; Darcy; Daniel, 2006). Posteriormente decidiu-se usar um nome mais simples (*SC*).

As hipóteses foram testadas de acordo com a seguintes formalizações:

H_1: existe uma correlação positiva entre RD e $SLOC$.

H_2 : existe uma correlação positiva entre RD e SC.

5.2.3 Resultados

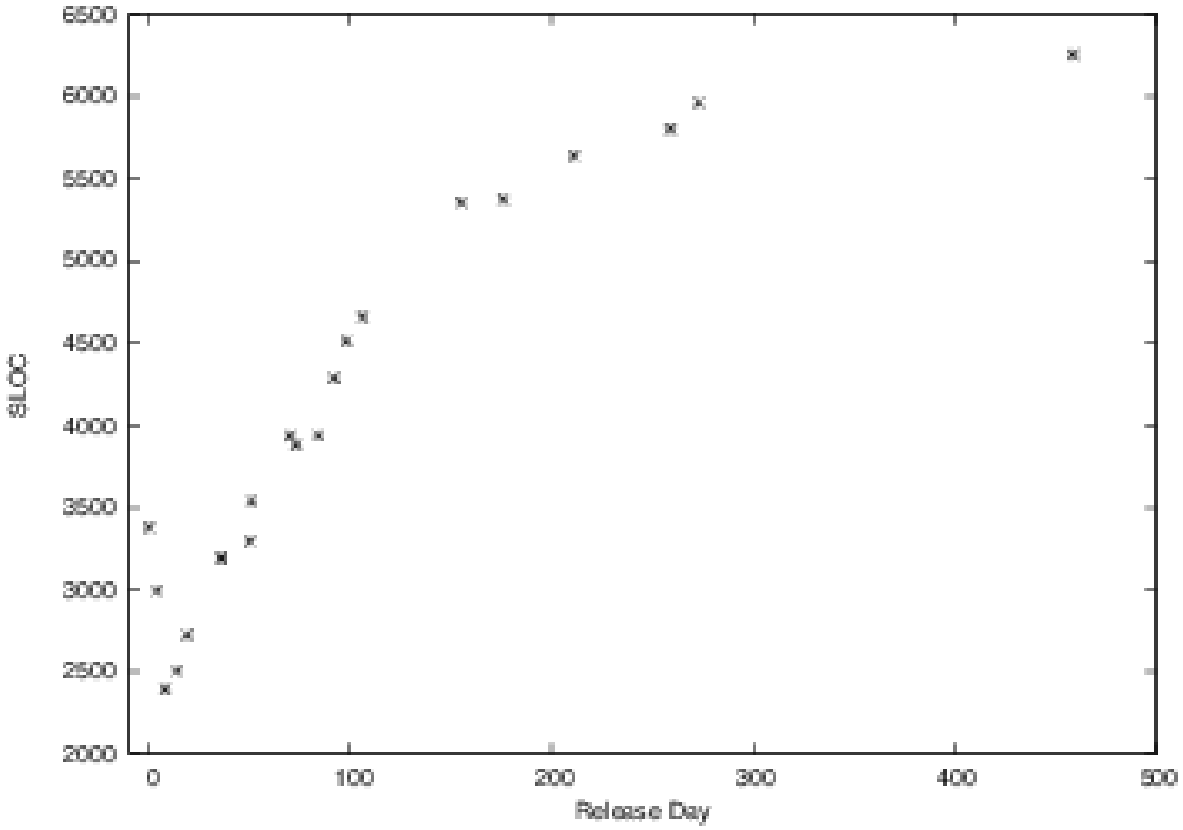


Figura 5.1 Ristretto: evolução do tamanho

H_1 foi confirmada: o teste de correlação de Pearson detectou uma correlação entre de 0.9041602 entre RD e $SLOC$, com $p < 0.01$. Como pode-se verificar na figura 5.1, de fato o tamanho estava fortemente relacionado ao tempo.

O teste de correlação de Pearson também confirmou H_2, e detectou uma correlação de 0.8636375 entre RD e SC, com $p < 0.01$. A figura 5.2 mostra uma plotagem da complexidade estrutural média contra o dia de lançamento.

Apesar de tanto tamanho quanto complexidade estrutural apresentarem uma forte correlação com o tempo de lançamento das versões, nota-se claramente que esse crescimento não é uniforme. Na figura 5.2, em especial, pode-se notar descontinuidades

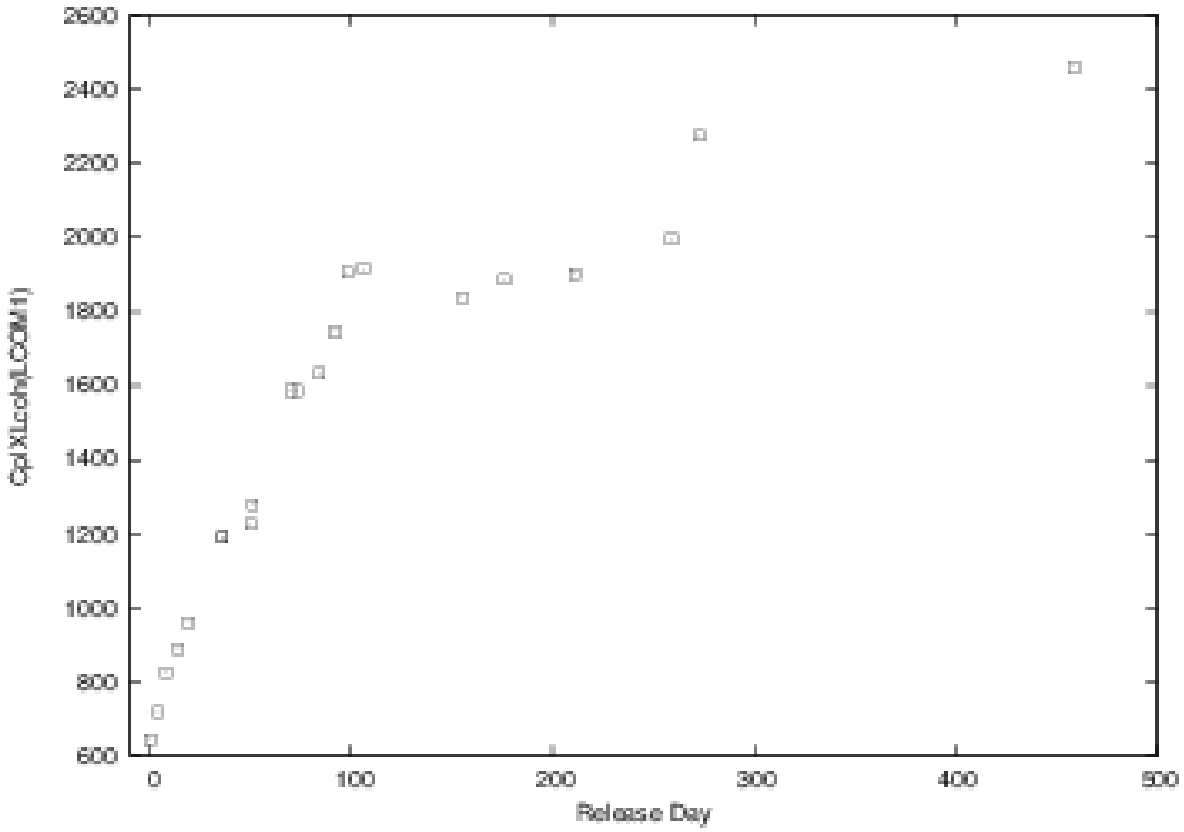


Figura 5.2 Ristretto: evolução da complexidade estrutural

próximas de $RD = 40$, $RD = 150$ e $RD = 450$. Estas descontinuidade representam pontos em que a complexidade passa a crescer em ritmo muito menor do que observado anteriormente, ou mesmo diminui (o caso para $RD = 150$).

Observando uma representação da arquitetura do projeto (figura 5.3), verificamos que essas descontinuidades coincidiam com o lançamento das versões 0.0.6, 0.0.16 e 0.0.21. Cada uma dessas versões exibiu uma mudança na arquitetura do projeto em relação à versão imediatamente anterior, ou seja, as descontinuidades no aumento da complexidade estrutural coincidiram com momentos de reorganização da arquitetura do projeto.

5.2.4 Ameaças à validade

Com relação à *validade de conclusão*, a utilização do teste de correlação de Pearson não foi totalmente adequada: ainda que $SLOC$ e SC apresentem distribuição normal, o mesmo não vale para a variável independente RD. Desta forma, a utilização do teste de Pearson não é adequada, por ele ser indicado para variáveis normais (Wohlin et al., 2000). No entanto, verificou-se posteriormente que o teste de Spearman, adequado para variáveis cuja distribuição não é normal, confirma os resultados relatados e até apresenta

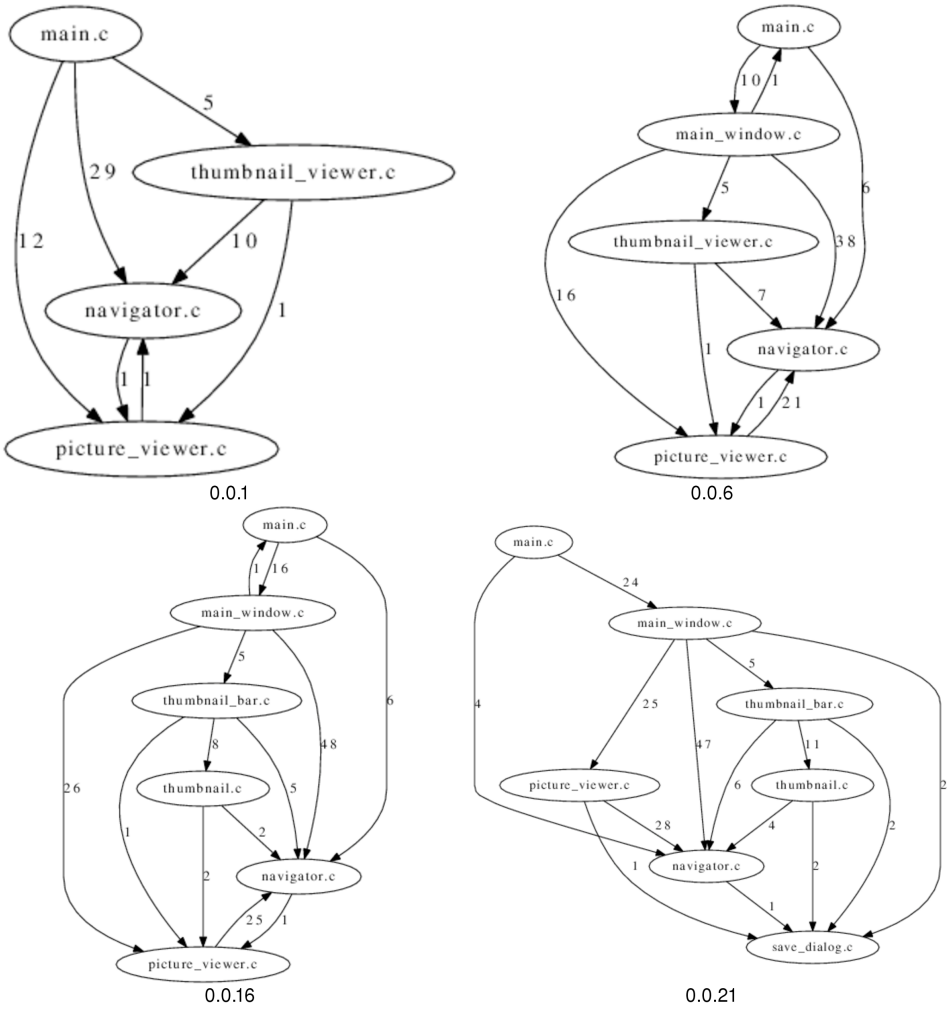


Figura 5.3 Ristretto: evolução da arquitetura

correlações mais fortes.

Este estudo também apresenta ameaças à *validade externa*, porque o projeto estudado (Ristretto) não é representativo de sistemas de software nos quais este tipo de análise se faz necessário. Primeiro, é um sistema bastante pequeno (menos de dez mil linhas de código). Segundo, porque possui apenas um desenvolvedor. Apesar do repositório de controle de versão do projeto apresentar 13 diferentes autores de contribuições ao projeto, foi descoberto posteriormente que apenas 1 desenvolvedor fazia modificações no código fonte, enquanto os outros 12 realizavam mudanças apenas em arquivos relacionados à tradução do programa para diferentes idiomas.

5.2.5 Conclusões

Neste estudo identificamos que mudanças na tendência evolutiva de aumento da complexidade estrutural, como descontinuidades e redução da taxa de crescimento, normalmente estão associadas a mudanças estruturais na arquitetura do projeto.

Concluímos também que a abordagem desenvolvida para análise automatizada de projetos de software era viável para estudos de grande escala e que poderíamos automatizar a execução para um grande número de versões.

De forma similar, uma vez que é possível automatizar a execução da Analizo, ela poderia ser introduzida no processo de compilação de projetos, viabilizando a análise e visualização de métricas durante o processo de desenvolvimento.

5.3 COMPLEXIDADE E NÍVEL DE PARTICIPAÇÃO DE DESENVOLVEDORES

Este estudo teve o objetivo de investigar a influência do nível de participação dos desenvolvedores (seção 4.1.3) sobre a evolução da complexidade estrutural em projetos de software livre.

Este estudo foi publicado nos anais do XXIV Simpósio Brasileiro de Engenharia de Software com o título "*An Empirical Study on the Structural Complexity Introduced by Core and Peripheral Developers in Free Software Projects*" (Terceiro; Rios; Chavez, 2010). O apêndice B contém o texto na íntegra.

5.3.1 Hipóteses

Existem diferenças entre desenvolvedores centrais e periféricos no que diz respeito ao volume de trabalho (Crowston; Howison, 2005) realizado e também no que diz respeito ao seu comportamento em discussões sobre o projeto de software (Masmoudi et al., 2009) e durante a atividade de gestão de defeitos (Dalle; Besten; Masmoudi, 2008). Isto nos levou a questionar se a qualidade da sua contribuição – levando-se em conta a complexidade estrutural do software – também seria diferente. Queríamos avaliar se a quantidade de complexidade estrutural introduzida pelos dois grupos de desenvolvedores seria diferente. Intuitivamente, considerando que os desenvolvedores centrais possuem um conhecimento mais aprofundado sobre a arquitetura do projeto, esperávamos que eles fossem capazes de realizar mudanças que introduziriam menos complexidade estrutural do que os desenvolvedores periféricos. Dessa forma, a primeira hipótese definida foi:

H_1: mudanças realizadas por desenvolvedores centrais introduzem menos complexidade estrutural do que mudanças realizadas por desenvolvedores periféricos.

Da mesma forma, imaginávamos que o mesmo valeria para atividades nas quais há redução na complexidade estrutural. Ainda que ambos os grupos pudessem participar

desta atividade, o maior conhecimento por parte dos desenvolvedores centrais também nos levava a crer que estes seriam mais bem sucedidos do que os desenvolvedores periféricos em atividades com redução na complexidade estrutural. Assim, formulamos a segunda hipótese:

H_2: Dentre as mudanças que reduzem a complexidade estrutural, aquelas realizadas por desenvolvedores centrais causam uma maior redução do que aquelas realizadas por desenvolvedores periféricos.

5.3.2 Projeto experimental

Este estudo foi realizado por meio da coleta de dados em repositórios de controle de versão de 7 projetos de software livre no domínio de servidores *web*, conforme apresenta a tabela 5.1. A unidade experimental escolhida foi o *commit*, como são conhecidas as mudanças registradas em sistemas de controle de versão. Para cada *commit*, foram obtidas as seguintes variáveis:

- Variável independente:
  - L, o nível de participação no projeto associado ao desenvolvedor no momento de realização do *commit*. Esta variável possui dois valores: "central" e "periférico".
- Variáveis dependentes:
  - SC, a complexidade estrutural do projeto após a realização do *commit*.
  - ΔSC, a variação de complexidade estrutural causada pelo *commit*. Obtida pela diferença entre o valor de SC de um *commit* e o valor de SC para o *commit* imediatamente anterior.
  - $|\Delta SC|$, o valor absoluto da variação de complexidade estrutural causada pelo *commit*.

Foram analisados 7 projetos de servidores web escritos em C. A tabela 5.1 apresenta algumas informações descritivas dos projetos analisados neste estudo.

5.3.3 Resultados

No total, 13553 *commits* foram analisados, dos quais 9944 (73.36%) haviam sido realizados por desenvolvedores centrais, e 3609 (26.63%) por desenvolvedores periféricos. Para testar as hipóteses colocadas, foram excluídos da análise os *commits* que não alteraram a complexidade estrutural (i.e. $\Delta SC = 0$). Os *commits* restantes estavam distribuídos de forma similar entre os dois grupos: de 2513, 1994 (79.35%) haviam sido realizados por

Tabela 5.1 Projetos analisados

Projeto	Início	Final	Commits	Desenvolvedores
aolserver	2000/05	2009/05	1125	22
apache	1999/06	2009/11	9663	72
cherokee	2005/03	2009/10	1545	8
fnord	2001/08	2007/11	99	2
lighttpd	2005/02	2009/10	775	6
monkeyd	2008/01	2009/06	207	4
weborf	2008/07	2009/10	139	3

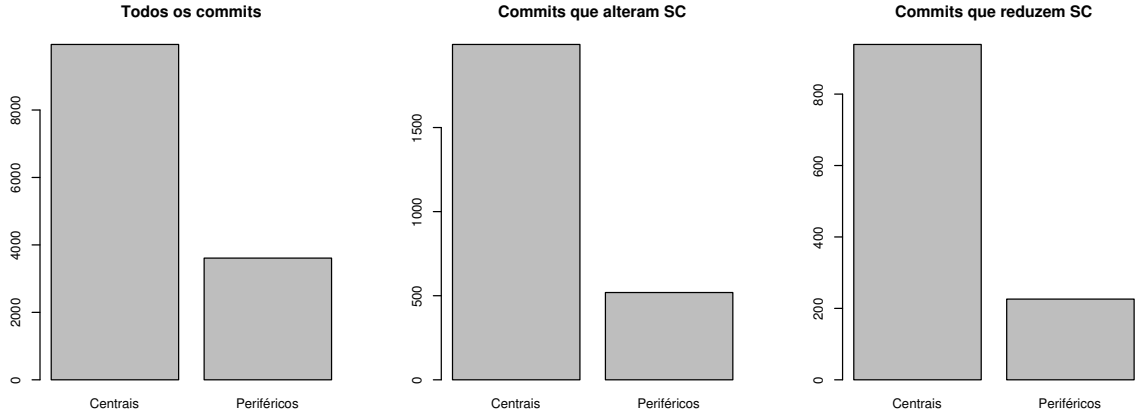


Figura 5.4 Distribuição de *commits* entre desenvolvedores centrais e periféricos.

desenvolvedores centrais, enquanto 519 (20.65%) haviam sido realizados por desenvolvedores periféricos (figura 5.4).

Para testar H_1 (mudanças realizadas por desenvolvedores centrais introduzem menos complexidade estrutural do que mudanças realizadas por desenvolvedores periféricos), comparamos ΔSC para os subconjuntos de *commits* realizados por desenvolvedores centrais e periféricos. H_1 foi formalizada como se segue:

$$H_1 : \mu_{\Delta SC_{centrais}} < \mu_{\Delta SC_{perif}}$$

Foi utilizado um teste-t para verificar H_1, que aceitou a hipótese com $p < 0.05$ ($p = 0.01515265$). Os dados confirmaram H_1, ou seja, nesta amostra de projetos, as mudanças realizadas pelos desenvolvedores centrais introduziram menos complexidade do que as mudanças realizadas peles desenvolvedores periféricos.

Para testar H_2 (entre as mudanças que reduzem a complexidade estrutural, aquelas realizadas por desenvolvedores centrais causam uma maior redução do que aquelas realizadas por desenvolvedores periféricos), o conjunto de *commits* foi filtrado novamente para

manter apenas os *commits* que haviam causado redução na complexidade, i.e. aqueles para os quais $\Delta SC < 0$. Neste conjunto reduzido, tínhamos 1165 *commits*, dos quais 939 (80.60%) foram realizados por desenvolvedores centrais e 226 (19.40%) por desenvolvedores periféricos (novamente, uma divisão semelhante à divisão do conjunto total de *commits* – ver figura 5.4). H_2 foi formalizada como se segue:

$$H_2 : \mu_{|\Delta SC|_{centrais}} > \mu_{|\Delta SC|_{perif}}$$

O teste-t para H_2 também nos permitiu aceitar H_2 com $p < 0.05$ ($p = 0.01091324$). Desta forma, os dados também suportaram a nossa hipótese de que nas atividades de redução da complexidade, os desenvolvedores centrais são capazes de remover mais complexidade (em média) do que os desenvolvedores periféricos.

5.3.4 Ameaças à validade

Ainda que projetado cuidadosamente, este estudo possui algumas limitações que representam ameaças à sua validade.

Do ponto de vista da *validade de conclusão*, o fato da variável testada (ΔSC) não possuir uma distribuição normal e ainda assim ter sido utilizado um teste-t representa um risco, uma vez que o teste-t normalmente requer variáveis com distribuição normal. Apesar disso, Wohlin nota que o teste-t é robusto o suficiente para suportar algum distanciamento das suas pré-condições. Em especial, como a amostra é grande o suficiente, o teste-t pode ser usado sem problemas (Wohlin et al., 2000). Para ter certeza, foi aplicado também um teste de Wilcoxon/Mann-Whitney – um teste não paramétrico utilizado em substituição ao teste-t para amostras não distribuídas normalmente – que nos forneceu resultados semelhantes.

Com relação à *validade externa*, a escolha de projetos escritos em C e dentro de um único domínio de aplicação não produziu uma amostra representativa de sistemas de software livre nem de sistemas de software em geral.

Do ponto de vista da *validade interna*, o fato de ter sido utilizada uma única variável independente representa uma ameaça. A nossa análise levou em consideração apenas o fato do desenvolvedor responsável pela mudança ser um desenvolvedor central ou periférico. Em especial, não foi analisada a natureza de cada mudança, ou outros atributos dos desenvolvedores e das mudanças. Desta forma, é possível que haja outros fatores fora do nosso controle que influenciem a variação na complexidade estrutural causada por mudanças.

Do ponto de vista da *validade de construção*, uma limitação causada pela escolha de usar apenas informações armazenadas de forma estruturada nos repositórios de controle de versão é a possibilidade de considerar o usuário responsável pelo *commit* como sendo o autor do *commit*. Como os projetos analisados utilizavam sistemas de controle de versão que não armazenam a informação de autoria explicitamente, é possível que uma parte

dos *commits* realizados estejam sendo atribuídos a desenvolvedores que apenas aplicaram no repositório oficial mudanças que na verdade foram realizadas por outros desenvolvedores. Para mitigar essa possibilidade, nos estudos posteriores foram utilizados apenas informações de projeto cujo controle de versão armazena explicitamente a informação de autoria de cada *commit*.

Ainda com relação à *validade de construção*, não foram analisadas mudanças que não causaram mudança na complexidade estrutural (i.e. aquelas onde $\Delta SC = 0$). Tais mudanças podem revelar atividade de *design* interessantes, mas que pelo fato de não alterarem a complexidade estrutural, foram desconsideradas neste estudo.

5.3.5 Conclusões

Neste estudo encontramos evidências que realçam a dicotomia entre desenvolvedores centrais e periféricos, desta vez em uma atividade eminentemente técnica. Desenvolvedores centrais e periféricos não só desempenham um volume de trabalho distinto e se comportam de forma distinta; eles também produzem código de complexidade diferente.

Descobrimos que os desenvolvedores centrais introduzem menos complexidade no código e removem mais complexidade, o que realça a importância da equipe de desenvolvedores centrais para o bem-estar do projeto. De certa forma, a equipe central é responsável por manter a integridade conceitual do projeto, como sugerido por Brooks (Brooks.jr, 1995).

Apesar disso, líderes de projetos de software livre não devem evitar contribuições de novos colaboradores. Considerando que o novo colaborador de hoje pode se tornar um dos desenvolvedores centrais no futuro (Robles; Gonzalez-barahona, 2006), líderes de projeto devem criar mecanismos para incentivar novos colaboradores e ao mesmo tempo minimizar o possível impacto negativo de suas ações sobre o código do projeto. Possíveis mecanismos para isso incluem:

- Processos de revisão explícita de código. Se as contribuições de desenvolvedores periféricos introduzirem complexidade desnecessária, a revisão por parte de desenvolvedores mais experientes pode ajudar a atingir o mesmo objetivo através de mudanças mais simples e que introduzam menos complexidade.

- Sistemas de *plugins*. Arquiteturas extensíveis por meio de *plugins* permitem que novas funcionalidades sejam adicionadas ao software sem que o seu núcleo precise ser alterado e, consequentemente, sem que a sua complexidade precise aumentar.

Uma limitação deste estudo foi o fato de que os projetos estudados utilizavam sistemas de controle de versão que não armazenam separadamente as identificações do desenvolvedor que aplicou a mudança no repositório do projeto e do desenvolvedor que é de fato o autor da mudança. Desta forma, uma certa quantidade de *commits* pode estar atribuída

ao desenvolvedor que aplicou a mudança no repositório, mas na verdade foi desenvolvida por outro desenvolvedor.

5.4 ANÁLISE QUANTITATIVA DA EVOLUÇÃO DA COMPLEXIDADE ESTRUTURAL

Neste estudo, analisamos 5 projetos de software livre em 3 diferentes domínios de aplicação, com o objetivo de estudar a influência de três fatores sobre a variação de complexidade estrutural: experiência dos desenvolvedores, variação de tamanho e difusão da mudança.

Este estudo foi publicado nos anais da *16th European Conference on Software Maintenance and Reengineering* com o título "*Understanding Structural Complexity Evolution: a Quantitative Analysis*" (Terceiro et al., 2012). O apêndice C contém o texto na íntegra.

5.4.1 Hipóteses

Ao formularmos as hipóteses para este estudo, resolvemos formular duas hipóteses associadas a cada fator: uma com relação a mudanças que aumentam a complexidade estrutural e outra com relação a mudanças que diminuem a complexidade estrutural. Essa distinção é justificada pela crença de que mudanças que aumentam a complexidade estrutural representam atividades de natureza diferente das atividades representadas por mudanças que reduzem a complexidade estrutural e, portanto não esperávamos que os fatores necessariamente influenciassem essas duas atividades de forma semelhante.

Assim, as hipóteses relativas ao aumento da complexidade estrutural foram as seguintes:

H_1: A experiência dos desenvolvedores influencia negativamente o aumento da complexidade estrutural. Espera-se que os desenvolvedores mais experientes introduzam menos complexidade estrutural.

H_2: A variação de tamanho influencia positivamente o aumento da complexidade estrutural.

H_3: A difusão da mudança influencia positivamente o aumento da complexidade estrutural.

Da mesma forma, as hipóteses relativas à redução da complexidade estrutural foram:

H_4: A experiência dos desenvolvedores influencia a redução da complexidade estrutural positivamente.

H_5: A variação de tamanho influencia a redução da complexidade estrutural negativamente. Espera-se que quanto mais o tamanho diminua, maior seja a redução causada na complexidade estrutural.

Tabela 5.2 Descritivo dos projetos estudados

Projeto	Linguagem	Tempo	Desenv.	Módulos	Domínio
Clojure	Java	5 anos	58	666	Compilador
Node	C++	2 anos	164	62	Compilador
Redis	C	2 anos	32	48	Banco de Dados
Voldemort	Java	2 anos	35	434	Banco de Dados
Zeromq2	C++	1.75 anos	39	106	Mensageria

H_6**:** A difusão de mudança influencia a redução da complexidade positivamente.

5.4.2 Projeto experimental

Neste estudo, a unidade experimental são *commits* registrados em sistemas de controle de versão. Nem todos os *commits* causam variação na complexidade estrutural, e por isso este estudo limitou-se a analisar apenas os *commits* que de fato levaram a uma variação, para cima ou para baixo, da complexidade estrutural.

- Exp – o número de *commits* previamente realizados pelo autor do *commit* desde que ele começou a trabalhar no projeto. Esta foi a medida utilizada para representar a experiência do desenvolvedor no projeto. No estudo publicado, esta variável foi chamada de n.
- ΔLOC – variação de tamanho, medida em número linhas de código.
- CF – difusão de mudança medida como o número de arquivos alterados pelo *commit*.

As variáveis dependentes foram as seguintes:

- ΔSC - variação na complexidade estrutural causada pelo *commit*.
- ΔSC_i - aumento na complexidade estrutural. Nos casos em que essa variável foi utilizada em testes estatísticos, apenas os *commits* que apresentaram aumento na complexidade estrutural (i.e. $\Delta SC > 0$) foram considerados.
- ΔSC_d - diminuição na complexidade estrutural. Nos casos em que essa variável foi utilizada em testes estatísticos, apenas os *commits* que apresentaram diminuição na complexidade estrutural (i.e. $\Delta SC < 0$) foram considerados.

ΔSC_i e ΔSC_d são ambos definidos como $|\Delta SC|$, mas considerados para conjuntos disjuntos de *commits*.

Para mitigar a limitação identificada no estudo apresentado na seção 5.3, desta vez buscamos projetos que utilizassem um sistema de controle de versão que armazena explicitamente a identificação do autor do *commit*. Desta forma, neste estudo foram analisados

5 projetos que utilizavam Git[4], cujos repositórios, portanto, apresentam informação de autoria das mudanças mais confiável. Os projetos selecionados são apresentados na tabela 5.2.

Clojure é uma linguagem funcional derivada de Lisp, e sua principal implementação, que também se chama Clojure, é desenvolvida em Java e compila código Clojure para *bytecode* de máquinas virtuais Java. Node é uma plataforma para programação em Javascript baseada no motor V8 Javascript com entrada e saída baseada em eventos, usada principalmente para programação de aplicações servidoras em Javascript. Redis e Voldemort são sistemas de armazenamento de pares chave/valor, normalmente consideradas como pertencendo à categoria de "bancos de dados NoSQL". Zeromq2 é uma biblioteca de mensagens usada para comunicação entre processos em aplicações distribuídas.

A figura 5.5 exibe a evolução dos projetos durante o período analisado em termos de sua complexidade estrutural e tamanho. Ambas medidas foram normalizadas em relação ao seu máximo histórico para poderem ser apresentadas no mesmo gráfico, mas vale a pena notar que tamanho e complexidade estrutural estão em escala diferentes. Ainda assim, essa visualização é bastante útil para colocar a evolução de uma métrica em perspectiva, em relação à evolução de outra.

De acordo com o senso comum, espera-se que tamanho e complexidade estrutural evoluam de forma parecida, o que é o caso, por exemplo, para o projeto Voldemort: suas linhas de tamanho e complexidade estrutural têm formas bastante parecidas. No entanto, isso não acontece em outros projetos. Node, por exemplo, parece demonstrar um comportamento contrário: enquanto o tamanho cresce consistentemente, a complexidade estrutural parece se estabilizar.

Existem ainda artefatos nos gráficos, demonstrando aumentos ou quedas abruptos tanto no tamanho quanto na complexidade estrutural, e nem sempre um salto no tamanho vem acompanhado de um salto da complexidade estrutural (e vice-versa). Esses saltos não foram analisados de forma especial neste estudo.

5.4.3 Resultados

Para testar as hipóteses H_1 a H_3, ΔSC_i foi modelado como uma combinação linear dos fatores em estudo: a experiência dos desenvolvedores, a variação no tamanho e a difusão de mudança:

$$\Delta SC_i = \alpha_0 + \alpha_1 Exp + \alpha_2 \Delta LOC + \alpha_3 CF$$

A tabela 5.3 exibe os resultados obtidos através deste modelo de regressão linear. Os coeficientes de regressão considerados significativos estão destacados em negrito.

Podemos ver que H_1 (a experiência do desenvolvedor no projeto influencia negativa-

[4] `http://git-scm.org/`

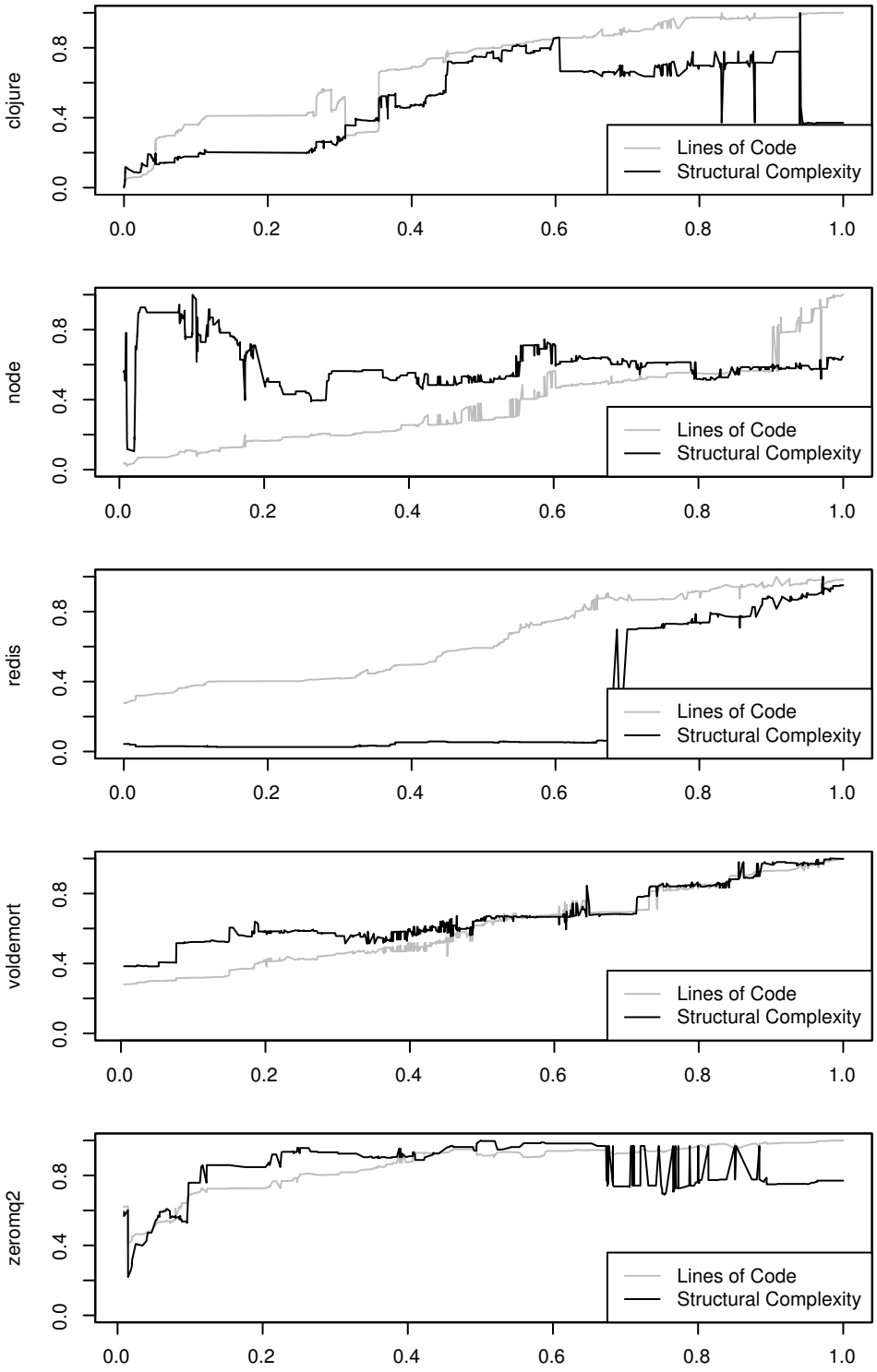


Figura 5.5 Evolução dos projetos analisados em termos da complexidade estrutural e do tamanho

Tabela 5.3 Modelos de regressão para *commits* que **aumentam** a complexidade estrutural.

Projeto	Exp (H_1)	ΔLOC (H_2)	CF (H_3)	Adj. R^2
clojure	0.000108	**0.000147 *****	**0.014175 *****	0.20
node	**-0.000479 ****	0.000369	**0.025465 ***	0.10
redis	0.004697	0.00244	**0.236758 *****	0.31
voldemort	-5e-06	**0.000195 ****	**0.002326 *****	0.22
zeromq2	-0.000399	**0.000746 ****	0.004965	0.21

***: $p < 0.001$; **: $p < 0.01$; *: $p < 0.05$

mente a variação da complexidade estrutural) foi confirmada apenas para o Node, que é o único projeto no qual a variável de experiência Exp teve um coeficiente com significância estatística e sinal negativo. Nossa interpretação foi de que as mudanças feitas por desenvolvedores com menos experiência introduzem mais complexidade estrutural, ou ao contrário, de que desenvolvedores mais experientes introduzem menos complexidade. Apesar do coeficiente parecer ter uma pequena magnitude, deve-se notar que esta variável está numa escala diferente em relação à variável dependente.

H_2 (variação de tamanho influencia o aumento da complexidade estrutural positivamente) foi confirmada para Clojure, Voldemort e Zeromq2. Em todos estes três projetos, mudanças que introduziram mais linhas de código também introduziram mais complexidade estrutural. A mesma observação anterior sobre a magnitude do coeficiente aplica-se também neste caso: ΔLOC e ΔSC_i são medidos em escalas diferentes.

Por último, descobrimos que H_3 (a difusão das mudanças influencia positivamente o aumento da complexidade estrutural) foi confirmada para todos os projetos, exceto Zeromq2. Nestes projetos as mudanças que abrangem mais arquivos também são aquelas que introduzem mais complexidade estrutural.

Para testar as hipóteses H_4 a H_6, foram construídos modelos de regressão linear similares aos discutidos acima, mas com a redução de complexidade estrutural como variável dependente (e portanto levando em consideração apenas os *commits* nos quais a variação de complexidade é negativa). A redução de complexidade estrutural foi modelada como uma combinação linear da experiência do desenvolvedor que realizou o *commit*, da variação de tamanho causada pelo *commit* e pela sua difusão:

$$\Delta SC_d = \alpha_0 + \alpha_1 n + \alpha_2 \Delta LOC + \alpha_3 CF$$

Os resultados obtidos com estes modelos de regressão são apresentados na tabela 5.4. Novamente, os coeficientes de regressão significativos estão destacados em negrito.

H_4 (a experiência do desenvolvedor influencia a redução na complexidade estrutural positivamente) foi confirmada apenas para Clojure. Nossa interpretação é de que neste projeto, as mudanças feitas por desenvolvedores com mais experiência são aqueles que causam uma maior redução na complexidade estrutural. É possível que a maior parte

Tabela 5.4 Modelos de regressão para *commits* que **diminuem** as complexidade estrutural.

Projeto	Exp (H_4)	ΔLOC (H_5)	CF (H_6)	Adj. R^2
clojure	**0.000422 ****	0.000204	0.011493	0.04
node	-0.00038	-0.000308	**0.036385 ****	0.10
redis	0.000952	0.000443	**0.163235 ****	0.18
voldemort	2.3e-05	**0.00013 ****	0.001125	0.07
zeromq2	-0.000372	**-0.001258 *****	**0.021416 *****	0.88

***: $p < 0.001$; **: $p < 0.01$; *: $p < 0.05$

das atividades anti-regressivas são realizadas pelos desenvolvedores mais experientes.

H_5 (variação do tamanho influencia a redução na complexidade estrutural nagativamente) foi confirmada apenas para Zeromq2. Naquele projeto, os *commits* que produziram uma maior redução foram aqueles com menores variações de tamanho, ou seja, *commits* que incluem um número pequeno de linhas de código ou mesmo que removem linhas de código. Voldemort também apresentou uma coeficiente significativo para este fator, ΔLOC, mas com coeficiente positivo: neste projeto, os *commits* que adicionaram mais linhas de código foram aqueles que proporcionaram maiores reduções na complexidade estrutural, o que é anti-intuitivo. Talvez os desenvolvedores do projeto tenham conseguido reduzir a complexidade quebrando módulos existentes em novos módulos maiores, mas menos acoplados ou mais coesos.

H_6 (difusão das mudanças influencia a redução da complexidade estrutural positivamente) foi confirmada para Node, Redis e Zeromq2. Nossa interpretação é que nestes projetos os *commits* que afetam mais arquivos são aqueles que causam uma maior redução na complexidade estrutural.

5.4.4 Ameaças à validade

Apesar de ter sido projetado com cuidado, este estudo não está livre de ameaças à sua validade. Checamos o projeto experimental contra a lista de ameaças à validade por Wohlin *et al* (Wohlin et al., 2000) e as questões identificadas são relatadas nesta seção.

Com relação à *validade externa*, o fato de terem sido avaliados apenas projetos de software livre faz com que não possamos generalizar os resultados para projetos de software em geral. Também não podemos generalizar para o caso geral de projetos de software livre, uma vez que apenas 5 projetos foram analisados e projetos de software livre arbitrários também apresentam bastante heterogeneidade entre si.

Com relação à *validade de construção*, uma limitação inerente à métrica usada para experiência dos desenvolvedores é o fato dela ser monotônica, ou seja, dela sempre crescer e nunca decair. Isto poderia ser compensado com a introdução de outra medida representando o grau de conhecimento do desenvolvedor com relação ao código, que crescesse na medida que o desenvolvedor trabalhe no código, e diminua em períodos de inatividade

ou quando outros desenvolvedores modificam o código.

Ainda com relação à *validade de construção*, não foi levada em consideração a natureza das mudanças introduzidas pelos *commits*: o fato de um *commit* representar o conserto de um defeito, a implementação de uma nova funcionalidade ou uma refatoração pode fazer uma grande diferença. Por exemplo, a natureza da complexidade estrutural introduzida pela implementação de uma nova funcionalidade pode ser diferente da complexidade estrutural introduzida por uma refatoração. Uma refatoração pode aumentar a complexidade estrutural por estar introduzindo características típicas de *frameworks*, o que pode possibilitar que mudanças futuras sejam realizadas com menores aumentos na complexidade estrutural.

Outra ameaça à *validade de construção* é o fato de que *commits* que não alteraram a complexidade estrutural – ou seja, aqueles para os quais $\Delta SC = 0$ – não foram incluídos na análise. É possível que alguns destes *commits* acabem na realidade influenciando futuros *commits* que alteram a complexidade estrutural. Por exemplo, um *commit* que introduz um defeito sério pode levar os desenvolvedores a perceber que uma grande refatoração é necessária para evitar defeitos semelhantes no futuro. Incluir estes *commits* na análise realizada neste estudo não seria tão simples, no entanto: como todos eles possuem o mesmo valor para a variável dependente ($\Delta SC = 0$!), eles não iriam fornecer resultados úteis nos modelos de regressão. Precisaríamos de um projeto experimental diferente para aproveitar estes dados adicionais.

Outra ameaça à *validade de contrução* é a premissa de que cada *commit* representa uma mudança auto-contida. Se houverem mudanças lógicas que tenham sido realizadas como uma série de *commits*, esta informação foi perdida e cada *commit* foi analisado como sendo uma mudança independente.

Com relação à *validade de conclusão*, o presente estudo apresenta algumas limitações relacionadas à violação de premissas das regressões lineares múltiplas realizadas:

- Nenhuma das variáveis independentes possui uma distribuição normal. Da mesma forma, as variáveis dependentes também não possuem distribuição normal.
- Os resíduos não são distribuídos normalmente, e na maioria dos modelos eles não são homoscedásticos, ou seja, não possuem a mesma variância.
- Alguns dos modelos apresentam violação de independência dos resíduos.
- Como a maioria dos modelos apresentaram um baixo coeficiente de determinação, é possível que regressão linear não seja uma técnica adequada para este estudo.

Outra limitação à *validade de conclusão* é o fato de que não foi feito um ajuste nos níveis de significância em função do fato de terem sido realizados múltiplos testes de significância (Bland; Altman, 1995). Os níveis de significância (p) associados a cada coeficiente dos modelos de regressão linear deveriam ter sido ajustados, usando por exemplo o método de Bonferroni.

5.4.5 Conclusões

O método aplicado neste estudo pode ser usado para suportar decisões de atribuição de trabalho em atividades de manutenção de software. Por exemplo, gerentes que encontrarem nos seus projetos uma associação entre experiência dos desenvolvedores e aumento na complexidade estrutural podem querer alocar os desenvolvedores mais experientes para trabalhar em mudanças arriscadas. Do contrário, se essa associação não foi encontrada no histórico do projeto, provavelmente não faz diferença alocar desenvolvedores específicos.

Gerentes podem também monitorar as mudanças e decidir se uma mudança específica precisa ser revisada explicitamente por um segundo desenvolvedor quando ela ultrapassa um limiar de número de linhas de código ou de número de arquivos alterados – se estes fatores forem relevantes para o projeto em questão.

Com relação aos resultados, tiramos importantes lições deste estudo, que são discutidas a seguir.

Todos os fatores estudados apresentaram influência sobre a variação da complexidade estrutural em pelo menos 2 modelos de regressão. Isto sugere que experiência dos desenvolvedores no projeto, variação de tamanho e difusão de mudanças devem ser levados em consideração quando se estiver monitorando métricas de software para decidir como priorizar recursos para revisão de código.

Difusão de mudanças é o mais influente entre os fatores estudados. Em 7 dos 10 modelos de regressão, mudanças que tocaram mais arquivos foram aquelas que adicionaram/removeram mais complexidade estrutural. Variação de tamanho foi o segundo fator mais influente, sendo significante em 5 de 10 modelos. Experiência do desenvolvedor no projeto teve uma influência significativa apenas em 2 dos 10 modelos, mas em um deles nenhum dos outros fatores foi significante.

O crescimento do sistema não está necessariamente associado a aumento na complexidade. Contrariando o senso comum, é possível crescer em tamanho sem um aumento correspondente em complexidade.

Projetos diferentes são influenciados por diferentes fatores. Uma implicação direta para gerentes de projeto é que para aplicar estes resultados, é necessário primeiro estudar o histórico do projeto e descobrir quais fatores possuem influência no projeto em questão.

Fatores que influenciam o aumento da complexidade não são os mesmos que influenciam a diminuição. Dependendo dos seus objetivos, gerentes de projeto podem considerar conjuntos diferentes de fatores. Se o objetivo principal é evitar o crescimento da complexidade estrutural, um conjunto de fatores deve ser levado em conta; se o objetivo é trabalhar para reduzir a complexidade estrutural, então um conjunto diferente de fatores deve ser trabalhado.

Provavelmente existem outros fatores influenciando a complexidade estrutural que não foram levados em conta neste estudo. Nossos modelos de regressão

não alcançaram um coeficiente de determinação (R^2) alto o suficiente. Isso nos leva a crer que podem existir outros fatores não abordados neste estudo que influenciam a complexidade estrutural. Mais pesquisas são necessárias para identificar estes fatores.

5.5 REFINAMENTO DE MODELOS PARA EVOLUÇÃO DA COMPLEXIDADE ESTRUTURAL

No estudo anterior (conforme visto na seção 5.4), foram construídos modelos de regressão linear para a variação da complexidade estrutural, nos quais foram utilizados como variáveis independentes a experiência do desenvolvedor, variação de tamanho e difusão de mudança. A maioria dos modelos obtidos, no entanto, possuía baixo coeficiente de determinação (R^2) e, portanto, explicam uma pequena parcela da variabilidade existente na variável dependente, no caso, a variação na complexidade estrutural.

O objetivo principal deste estudo é de obter melhores modelos por meio da introdução de novas variáveis. Para isso, introduzimos dois refinamentos nos modelos:

1. A difusão de mudança foi reformulada, e passamos a considerar separadamente o número de módulos modificados e o número de módulos adicionados em cada *commit*, ao invés de considerar esses dois números conjuntamente.

   Intuitivamente, imaginamos que alterar módulos existentes e adicionar novos módulos afetam o design do sistema de formas diferentes, e por isso espera-se que essa separação possa melhorar a predição da variação na complexidade estrutural.

2. Foram incluídas variáveis correspondentes aos componentes do grau de autoria (ver seção 4.1.2).

   O grau de autoria é um dos fatores relacionados no capítulo 4, e reflete a expertise dos desenvolvedores com módulos específicos do sistema. Esta é uma dimensão de experiência diferente daquele obtida com a medida de experiência onde conta-se o número de *commits* previamente realizados pelo desenvolvedor.

5.5.1 Hipóteses

As hipóteses deste estudo estão relacionadas aos refinamentos realizados nos modelos para a variação da complexidade estrutural. Estamos interessados em identificar se estes refinamentos produzem modelos com maior coeficiente de determinação. Temos então as seguintes hipóteses:

H_1: Considerar separadamente número de módulos adicionados e número de módulos modificados produz modelos com maior coeficiente de determinação.

H_2: A introdução dos componentes do grau de autoria incorre em modelos com maior coeficiente de determinação.

H_3**:** A introdução de ambos os refinamentos nos leva a modelos com maior coeficiente de determinação.

Uma vez que as novas variáveis introduzidas representam dimensões diferentes sobre os desenvolvedores e sobre a mudança realizada, esperamos que incluí-las nos modelos nos levem a modelos com melhor coeficiente de determinação.

Além disso, também estamos interessados em verificar se esses novos fatores, de fato, influenciam a variação na complexidade estrutural. Mais uma vez, analisaremos os *commits* que aumentam a complexidade estrutural separadamente dos *commits* que reduzem a complexidade estrutural, por considerarmos que eles representam atividades distintas no ciclo de vida do projeto.

Temos então as seguintes hipóteses relacionadas aos *commits* que aumentam a complexidade estrutural:

H_4**:** A primeira autoria influencia negativamente o aumento da complexidade estrutural.

Desenvolvedores que alteram módulos criados por eles devem ser capazes de realizar mudanças que introduzam menos complexidade estrutural.

H_5**:** O número de modificações posteriores influencia negativamente o aumento na complexidade estrutural.

Ao realizarem mudanças em módulos com os quais estão acostumados, desenvolvedores tentem a introduzir menos complexidade estrutural.

H_6**:** O número de modificações por outros desenvolvedores influencia positivamente o aumento na complexidade estrutural.

Ao realizarem mudanças em módulos que outros desenvolvedores realizaram mudanças, desenvolvedores tentem a introduzir mais complexidade estrutural.

H_7**:** O número de módulos modificados influencia positivamente o aumento na complexidade estrutural.

Quanto mais módulos precisarem ser alterados para realizar uma mudança, mais será a aumento na complexidade estrutural do sistema.

H_8**:** O número de módulos adicionados influencia positivamente o aumento na complexidade estrutural.

Quanto mais módulos forem adicionados, maior será o aumento na complexidade estrutural.

Para os *commits* que reduzem a complexidade estrutural, temos as seguintes hipóteses análogas:

H_9: A primeira autoria influencia positivamente a redução na complexidade estrutural.

Desenvolvedores alterando módulos criados por eles devem ser capazes de causar maiores reduções na complexidade estrutural.

H_{10}: O número de modificações posteriores influencia positivamente a redução na complexidade estrutural.

Desenvolvedores modificando módulos que estão acostumados a modificar tendem a causar maiores reduções na complexidade estrutural.

H_{11}: O número de modificações por outros desenvolvedores influencia negativamente a redução na complexidade estrutural.

Desenvolvedores modificando módulos onde outros desenvolvedores realizaram mudanças tendem a causar menores reduções na complexidade estrutural.

H_{12}: O número de módulos modificados influencia positivamente a redução na complexidade estrutural.

Quanto mais módulos foram alterados, maior será a redução na complexidade estrutural.

H_{13}: O número de módulos adicionados influencia negativamente a redução na complexidade estrutural.

Em *commits* que reduzem a complexidade estrutural, espera-se que módulos adicionados sejam menos complexos do que a média atual, que causam uma redução na complexidade estrutural.

5.5.2 Projeto experimental

Para cada *commit* analisado, as seguintes variáveis foram extraídas:

- Exp – experiência do desenvolvedor no projeto. Calculada como o número de *commits* anteriores ao *commit* em questão.
- FA – primeira autoria (*first authorship*). $FA = 1$ se o desenvolvedor foi o autor original de um módulo, e $FA = 0$ caso contrário. Para o *commit* como um todo, foi considerada a soma dos valores de FA para todos os módulos envolvidos no *commit*.
- DL – número de modificações (*deliveries*). DL representa o número de modificações realizadas por um desenvolvedor num dado módulo. Para o *commit* como um todo, foi considerada a soma dos valores de DL para os módulos envolvidos no *commit*.
- AC – número de modificações por outros desenvolvedores (*acceptances*). AC representa o número de mudanças que outros desenvolvedores realizaram num módulo até aquele ponto no tempo. Para o *commit* como um todo, foi considerada a soma dos valores de AC para os módulos envolvidos no *commit*.

Tabela 5.5 Quantidade de *commits* analisados, divididos entre os *commits* que aumentam a complexidade estrutural e os que a reduzem.

Projeto	$\Delta SC > 0$	$\Delta SC < 0$
clojure	291	239
node	146	117
redis	65	75
voldemort	223	184
zeromq2	77	63

- ΔLOC – variação de tamanho causada pelo *commit*, em número de linhas de código.
- AM – número de módulos adicionados (*added modules*) pelo *commit*.
- CM – número de módulos modificados (*changed modules*) pelo *commit*.

Os projetos analisados foram os mesmos do estudo anterior (ver tabela 5.2 na página 53 e figura 5.5 na página 55). O número de *commits* analisados para cada projeto neste estudo pode ser visto na tabela 5.5. Novamente, modelamos a variação na complexidade estrutural como uma combinação linear. Desta vez, no entanto, nós testamos 4 modelos diferentes.

M_1 designa o modelo original obtido no estudo anterior, onde $CF = AM + CM$.

$$M_1 : \Delta SC = \alpha_0 + \alpha_1 Exp + \alpha_2 \Delta LOC + \alpha_3 CF$$

M_2 designa um modelo no qual os termos componentes de CF são considerados separadamente.

$$M_2 : \Delta SC = \alpha_0 + \alpha_1 Exp + \alpha_2 \Delta LOC + \alpha_3 AM + \alpha_4 CM$$

M_3 designa um modelo que inclui os termos relacionados ao grau de autoria:

$$M_3 : \Delta SC = \alpha_0 + \alpha_1 Exp + \alpha_2 FA + \alpha_3 DL + \alpha_4 AC + \alpha_5 \Delta LOC + \alpha_6 CF$$

Por fim, M_4 designa um modelo completo, que apresenta as duas modificações testadas neste estudo: CF é separado em AM e CM, e são introduzidos os termos que compõem o grau de autoria.

$$M_4 : \Delta SC = \alpha_0 + \alpha_1 Exp + \alpha_2 FA + \alpha_3 DL + \alpha_4 AC + \alpha_5 \Delta LOC + \alpha_6 AM + \alpha_7 CM$$

Tabela 5.6 Diferenças nos coeficientes de determinação dos novos modelos em comparação ao modelo original M_1, entre os *commits* que aumentam a complexidade estrutural.

Projeto	$R^2_{M_1}$	$R^2_{M_2} - R^2_{M_1}(H_1)$	$R^2_{M_3} - R^2_{M_1}(H_2)$	$R^2_{M_4} - R^2_{M_1}(H_3)$
clojure	0.08	0.00	0.02	0.02
node	0.14	0.01	0.20	0.21
redis	0.82	0.11	0.02	0.11
voldemort	0.47	0.18	0.04	0.18
zeromq2	0.20	0.10	0.00	0.13

Tabela 5.7 Diferenças nos coeficientes de determinação dos novos modelos em comparação ao modelo original M_1, entre os *commits* que reduzem a complexidade estrutural.

Projeto	$R^2_{M_1}$	$R^2_{M_2} - R^2_{M_1}(H_1)$	$R^2_{M_3} - R^2_{M_1}(H_2)$	$R^2_{M_4} - R^2_{M_1}(H_3)$
clojure	0.43	0.00	0.26	0.26
node	0.06	-0.01	-0.02	-0.02
redis	0.23	-0.01	0.13	0.14
voldemort	0.05	-0.01	0.02	0.02
zeromq2	0.44	0.00	0.04	0.05

5.5.3 Resultados

Para analisar os resultados relativos a H_1, H_2 e H_3 , realizamos as regressões lineares para os quatro modelos, e comparamos o coeficiente de determinação (R^2 ajustado) de M_1, nosso modelo original, com os coeficientes de determinação dos novos modelos M_2, M_3 e M_4.

A tabela 5.6 apresenta as comparações entre os coeficientes de determinação dos modelos testados para os commits que aumentam a complexidade estrutural, arredondados para 2 casas decimais. Neste estudo, consideramos como significativas diferenças de pelo menos 0.05.

Na tabela 5.6, pode-se verificar que M_2 obteve um coeficiente de determinação melhor do que M_1 em 3 dos 5 projetos: Redis, Voldemort e Zeromq2. M_3 superou M_1 em apenas 1 dos 5 projetos: Node. Finalmente, M_4 forneceu os melhores resultados, com um coeficiente de determinação significativamente melhor do que M_1 em 4 dos 5 projetos: apenas para Clojure M_4 não obteve uma melhora significativa no coeficiente de determinação. Com relação à magnitude dessas melhorias, M_4 conseguiu melhorias entre 0.11 e 0.21 no coeficiente de determinação.

A tabela 5.7 apresenta os resultados relativos aos *commits* que reduzem a complexidade estrutural. M_2 não obteve um coeficiente de determinação significantemente melhor do que M_1 em nenhum dos projetos. M_3 obteve melhor coeficiente de determinação em 2 dos 5 projetos, Clojure e Node. No caso de Node, o coeficiente de determinação obtido por M_3 R^2 foi menor do que o do o do modelo original M_1, mas não é uma diferença considerada significativa neste estudo. M_4 novamente apresentou os melhores resultados.

Tabela 5.8 Resumo dos resultados para $H1$, H_2 e H_3

	$\Delta SC > 0$			$\Delta SC < 0$		
Projeto	H_1	H_2	H_3	H_1	H_2	H_3
clojure					✓	✓
node		✓	✓			
redis	✓		✓		✓	✓
voldemort	✓		✓			
zeromq2	✓		✓			✓

M_4 foi significativamente superior a M_1 em 3 dos 5 projetos (Clojure, Redis e Zeromq2), obtendo uma melhoria no coeficiente de determinação entre 0.05 e 0.26. M_4 também obteve um resultado pior no caso de Node, porém a diferença não é considerada significativa neste estudo.

A tabela 5.8 apresenta um resumo dos resultados para as hipóteses H_1 a H_3. Pode-se notar que os novos modelos obtiveram melhores resultados com relação aos dois grupos de *commits*, e M_4 foi o modelo que obteve o melhor desempenho.

Para testar as hipóteses H_4 a H_{13}, portanto, utilizamos os coeficientes de regressão obtidos a partir de M_4. A tabela 5.9 apresenta os resultados da regressão linear baseada no modelo M_4.

H_4 (a primeira autoria influencia negativamente o aumento na complexidade estrutural) não foi confirmada. No caso do projeto Node, obtivemos um coeficiente significativo mas positivo, o que indica que, naquele projeto, os desenvolvedores que realizaram mudanças nos módulos cuja primeira autoria foi deles estavam propensos a causar maiores aumentos na complexidade estrutural.

H_5 (o número de modificações posteriores influencia negativamente o aumento na complexidade estrutural) não foi confirmada. Clojure apresentou um coeficiente significativo, mas com valor zero. Na nossa interpretação, o número de modificações posteriores **não** influencia o aumento da complexidade estrutural. Zeromq2 apresentou um coeficiente positivo, o que contradiz a nossa hipótese. Os desenvolvedores que realizaram mudanças em módulos que eles já haviam modificado antes causaram maiores aumentos na complexidade estrutural.

H_6 (o número de modificações por outros desenvolvedores influencia positivamente o aumento na complexidade estrutural) foi confirmada apenas para Clojure.

H_7 (o número de módulos modificados influencia positivamente o aumento na complexidade estrutural) foi confirmada apenas para Voldemort.

H_8: (o número de módulos adicionados influencia positivamente o aumento na complexidade estrutural) foi confirmada para 3 dos 5 projetos: Redis, Voldemort e Zeromq2.

Os resultados relativos aos *commits* que reduzem a complexidade estrutural são apresentados na tabela 5.10 e descritos a seguir.

Tabela 5.9 Regressões lineares para *commits* que aumentam a complexidade estrutural

Projeto	Exp	FA (H_4)	DL (H_5)	AC (H_6)
clojure	3e-05	0.003812	**0 ***	**8e-06 ****
node	**-0.000483 ****	**0.176619 *****	-0.000482	0.001365
redis	0.000306	0.044213	0.001175	0.001393
voldemort	3.4e-05	-0.000295	-1e-06	-5e-06
zeromq2	-0.000187	0.005243	**0.001098 ***	-0.002774
Projeto	ΔLOC	CM (H_7)	AM (H_8)	R^2
clojure	**0.000168 ****	0.001829	-0.015597	0.11
node	**0.000589 ****	-0.001794	0.068049	0.35
redis	-0.001718	-0.004448	**0.926037 *****	0.93
voldemort	**0.000142 *****	**0.001498 *****	**0.013466 *****	0.65
zeromq2	0.000602	-0.005991	**0.114908 *****	0.33

***: $p < 0.001$; **: $p < 0.01$; *: $p < 0.05$

H_9 (a primeira autoria influencia positivamente a redução na complexidade estrutural) foi confirmada apenas para Clojure. Naquele projeto, as mudanças realizadas por desenvolvedores que foram os autores originais dos módulos alterados são propensas a causar maiores reduções na complexidade estrutural. Redis também teve um coeficiente significativo mas negativo: isso quer dizer que as mudanças feitas por desenvolvedores que foram os autores originais dos módulos alterados foram propensas a causar menores reduções na complexidade estrutural.

H_{10} (o número de modificações posteriores influencia positivamente a redução na complexidade estrutural) foi confirmada para 2 dos 5 projetos: Clojure e Redis. Nesses projetos, as mudanças realizadas por desenvolvedores que haviam realizado um maior número de modificações prévias naqueles módulos foram as que causaram maiores reduções na complexidade estrutural. Zeromq2 também teve um coeficiente significativo, mas no sentido contrário ao da nossa hipótese.

H_{11} (o número de modificações por outros desenvolvedores influencia negativamente a redução na complexidade estrutural) foi confirmada para Clojure e Redis. Nesses projetos, quanto maior o número de modificações que outros desenvolvedores realizaram, menor foi a redução de complexidade estrutural alcançada com as mudanças.

H_{12} (o número de módulos modificados influencia positivamente a redução na complexidade estrutural) não foi confirmada. Clojure e Voldemort apresentaram coeficientes significativos, mas negativos, o que é o contrário do que esperávamos.

H_{13} (o número de módulos adicionados influencia negativamente a redução na complexidade estrutural) não foi confirmada em nenhum dos projetos.

5.5.4 Ameaças à validade

Por se tratar de uma extensão do estudo apresentado na seção anterior, este estudo compartilha todas as suas ameaças à validade, apresentadas na seção 5.4.4.

Tabela 5.10 Regressões lineares para *commits* que reduzem a complexidade estrutural

Projeto	Exp	FA (H_9)	DL (H_{10})	AC (H_{11})
clojure	-0.000193	**0.025424 *****	**1e-06 *****	**-1.4e-05 *****
node	0.000714	-0.038055	-9.2e-05	-0.000985
redis	-0.000248	**-0.307591 *****	**0.001816 ****	**-0.003596 ****
voldemort	3e-05	0.00187	3e-06	6e-06
zeromq2	-0.001182	0.029765	**-0.000707 ***	0.000577
Projeto	ΔLOC	CM (H_{12})	AM (H_{13})	R^2
clojure	-0.000125	**-0.021474 *****	-0.010159	0.69
node	**0.000524 ***	-0.01082	-0.082005	0.04
redis	-0.000741	0.012442	-0.176974	0.36
voldemort	-2.6e-05	**-0.001856 ****	-0.000906	0.07
zeromq2	**0.0016 *****	-0.004717	-0.118971	0.48

***: $p < 0.001$; **: $p < 0.01$; *: $p < 0.05$

Tabela 5.11 Resumo sobre as hipóteses relacionadas à influência dos diversos fatores sobre a variação na complexidade estrutural

Projeto	H_4	H_5	H_6	H_7	H_8	H_9	H_{10}	H_{11}	H_{12}	H_{13}
clojure			✓			✓	✓	✓		
node										
redis					✓		✓	✓		
voldemort					✓					
zeromq2				✓	✓					

5.5.5 Conclusões

A tabela 5.11 apresenta um resumo dos resultados dos testes de hipótese de H_4 a H_{13}. Apenas 3 hipóteses foram confirmadas em mais de 1 projeto, a saber:

- H_8: o número de módulos adicionados influencia positivamente o aumento na complexidade estrutural.
- H_{10}: o número de modificações posteriores influencia positivamente a redução na complexidade estrutural.
- H_{11}: o número de modificações por outros desenvolvedores influencia negativamente a redução na complexidade estrutural.

Façamos agora uma análise dos fatores que influenciam a variação da complexidade estrutural em cada projeto, apresentados na tabela 5.12.

Novamente, podemos notar que diferentes fatores influenciam diferentes projetos. Além disso, para cada projeto os fatores que influenciam o aumento da complexidade estrutural são diferentes dos fatores que influenciam a redução da complexidade estrutural.

Tabela 5.12 Resumo dos fatores que influenciam a variação da complexidade estrutural em cada projeto

Projeto	Fatores para $\Delta SC > 0$	R^2	Fatores para $\Delta SC < 0$	R^2
clojure	AC, ΔLOC	0.11	FA, DL, AC, CM	0.69
node	E, FA, ΔLOC	0.35	ΔLOC	0.04
redis	AM	0.93	FA, DL, AC	0.36
voldemort	ΔLOC, CM, AM	0.65	CM	0.07
zeromq2	DL, AM	0.33	DL, ΔLOC	0.48

O fator mais recorrente foi a variação de tamanho (ΔLOC), presente em 5 dos 10 casos. Em seguida temos o número de modificações prévias (DL), presente em 4 dos 10 casos. Autoria prévia (FA), modificações por outros desenvolvedores (AC) e número de módulos adicionados (AM) são influentes em 3 dos 10 casos.

É importante notar que todos os componentes do grau de autoria (FA, DL, AC) influenciaram a variação da complexidade estrutural em pelo menos 3 dos 10 casos.

Os modelos apresentaram uma grande variação nos seus coeficientes de determinação (R^2). Por exemplo, 93% da variação no aumento da complexidade estrutural no projeto Redis pode ser explicado por apenas uma variável, o número de módulos adicionados (AM). Já no caso da redução da complexidade estrutural nos projetos Node e Voldemort, o nosso melhor modelo explica apenas 4 e 7% da variação, respectivamente. Isto significa que ainda existe uma grande variabilidade na complexidade estrutural destes projetos que não pode ser explicada com as variáveis que utilizamos até agora, e que ainda é necessário identificar outros fatores que influenciam a variação da complexidade estrutural naqueles projetos.

5.6 CONCLUSÕES

Uma das principais preocupações deste trabalho foi identificar os fatores que poderiam ser considerados como causa da complexidade estrutural. Uma vez conhecidos os fatores que influenciam o aumento e a diminuição da complexidade estrutural, gerentes de projeto podem adicionar aos seu processos atividades para contenção da complexidade estrutural, de forma que a manutenção futura do sistema seja menos custosa.

Conforme discutido no capítulo 4, identificamos três grupos de fatores candidatos a influenciar a complexidade estrutural em sistemas de software: fatores relacionados aos desenvolvedores envolvidos no projeto, fatores relacionados à manutenção do sistema, e fatores relacionados à organização desenvolvedora de software. Neste trabalho, exploramos fatores dos dois primeiros grupos (desenvolvedores e manutenção) em uma série de estudos experimentais descritos ao longo deste capítulo.

Experiência no projeto foi utilizada em dois estudos, descritos nas seções 5.4 e 5.5. No primeiro estudo, a experiência no projeto teve uma influência significativa em 1

modelo de 5 no caso de *commits* que aumentavam a complexidade, e em 1 de 5 no caso de *commits* que reduziam a complexidade. No segundo estudo, a experiência no projeto teve uma influência significativa em 1 de 5 modelos para *commits* que aumentavam a complexidade e 2 de 5 para *commits* que reduziam a complexidade estrutural.

Grau de autoria foi introduzido no estudo descrito na seção 5.5, e teve seus 3 componentes (autoria inicial – FA, alterações subsequentes – DL e alterações por outros desenvolvedores – AC) usados como fatores independentes. Autoria inicial (FA) foi significativa em 1 dos 5 modelos para *commits* que aumentavam a complexidade, e em 2 de 5 modelos para *commits* que reduziam a complexidade estrutural. Alterações subsequentes (DL) foi significativa em 1 dos 5 modelos para *commits* que aumentavam a complexidade, e em 3 de 5 modelos para *commits* que reduziam. Alterações por outros desenvolvedores (AC) foi significativa em 1 dos 5 modelos para *commits* que aumentavam a complexidade, e em 2 de 5 modelos para *commits* que reduziam. Vale notar que os componentes do grau de autoria foram significativos mais vezes entre os *commits* que reduziam a complexidade do que no caso dos *commits* que aumentavam a complexidade.

Nível de participação foi utilizado no estudo descrito na seção 5.3 para classificar os autores dos *commits* entre desenvolvedores centrais ou periféricos, e verificou-se que, em geral, os desenvolvedores centrais introduziam menos – e removiam mais – complexidade do que os periféricos. No entanto, como o nível de participação não foi utilizado na construção dos modelos de regressão linear, não podemos verificar em que parcela dos caso ele se aplicaria.

Variação de tamanho se mostrou como um fator importante no estudo da variação da complexidade estrutural. Em ambos os estudos onde foi utilizada (seções 5.4 e 5.5), a variação de tamanho influenciou significativamente a variação na complexidade em 3 de 5 modelos para *commits* que aumentavam a complexidade, e em 2 de 5 modelos para *commits* que reduziam a complexidade.

Difusão de mudança também foi um fator influente em uma grande quantidade de casos. No primeiro estudo (5.4), ele foi significativo em 4 entre 5 modelos para *commits* que aumentavam a complexidade, e em 3 entre 5 modelos para *commits* que reduziam a complexidade. No segundo estudo, a difusão de mudança foi estudada separadamente a partir de módulos alterados (CM) e módulos adicionados (AM). Verificou-se então que CM foi significativo em 1 dos 5 modelos para *commits* que aumentavam a complexidade, e em 2 entre 5 modelos para os *commits* que reduziam a complexidade. AM foi significativo em 3 entre 5 modelos para *commits* que aumentavam a complexidade, e não foi significativo em nenhum dos 5 modelos para os *commits* que reduziam a complexidade.

Podemos usar a frequência com que cada fator foi significativo como uma métrica que indique a relevância deste fator no estudo da variação da complexidade estrutural. Dessa forma, pode-se priorizar os fatores durante o planejamento de estudos futuros. Também é possível que gerentes de projeto utilizem essa métrica para priorizar a implantação de medições desses fatores nos seus projetos.

A figura 5.6 apresenta os fatores em ordem crescente de frequência, para *commits* que

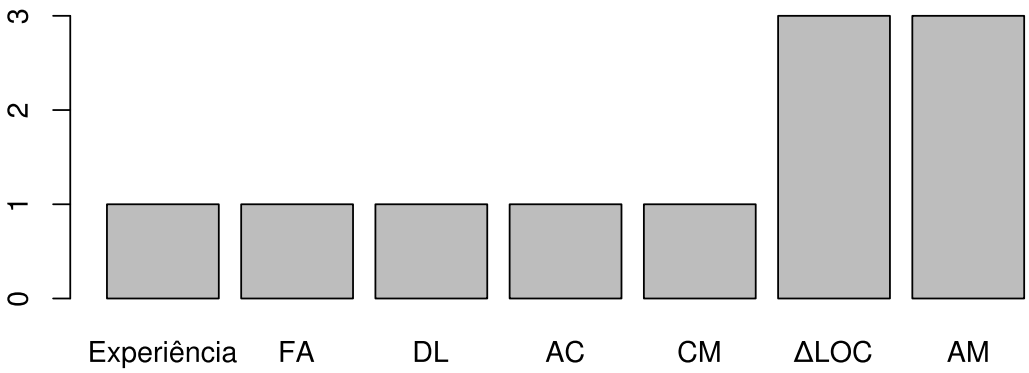


Figura 5.6 Fatores em ordem de relevância para *commits* que aumentam a complexidade estrutural.

aumentam a complexidade estrutural. Pode-se ver que os fatores mais frequentes foram ΔLOC e AM. Em atividades cujo objetivo seja controlar o aumento da complexidade estrutural, esses dois fatores podem ser priorizados em relação aos demais.

Entre os *commits* que aumentam a complexidade, pode-se notar que todos os fatores humanos (experiência e os componentes do grau de autoria FA, DL e AC) foram significativos em apenas um projeto. Fatores relacionados à evolução/manutenção do sistema (ΔLOC e módulos adicionados, AM) foram significativos em 3 projetos dos 5 analisados.

Os fatores que influenciam a variação da complexidade estrutural para os *commits* que a reduzem são representados na figura 5.7. DL foi o fator mais relevante nos estudos realizados. Gestores implantando atividades com objetivo de reduzir a complexidade estrutural podem priorizar a alocação de desenvolvedores às áreas do projeto nas quais eles estão acostumados a realizar mudanças.

Podemos notar que no caso da redução de complexidade, o fator que foi significativo mais vezes foi um fator humano: o número de alterações prévias (DL). Também é válido notar que nesse caso, os demais fatores humanos – experiência e os outros componentes do grau de autoria – foram significativos no mesmo número de projetos que fatores relacionados à evolução/manutenção, como ΔLOC e número de módulos modificados (CM).

O capítulo seguinte descreve apresenta uma teoria que relaciona causas e consequências da complexidade estrutural em projetos de software livre. Esta teoria sintetiza os resultados experimentais obtidos com este trabalho.

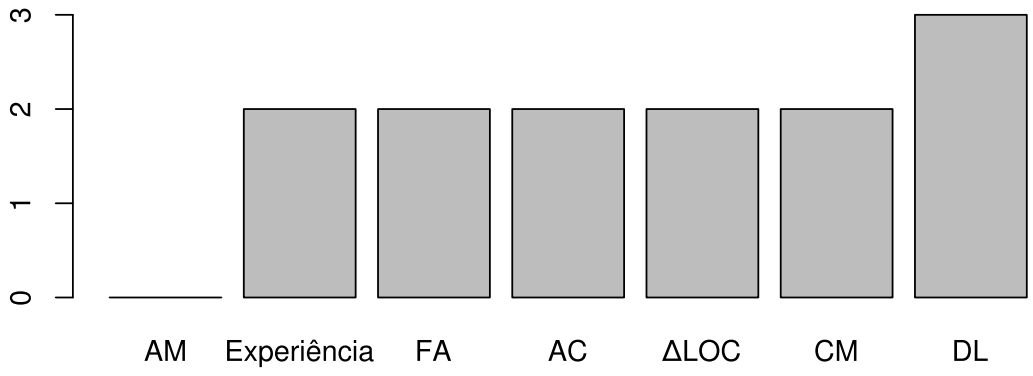


Figura 5.7 Fatores em ordem de relevância para *commits* que reduzem a complexidade estrutural.

Capítulo

6

Este capítulo apresenta uma teoria proposta para sintetizar os resultados obtidos nesta tese. Esta teoria busca principalmente identificar causas e consequências da complexidade estrutural em sistemas de software livre.

UMA TEORIA PARA A COMPLEXIDADE ESTRUTURAL EM PROJETOS DE SOFTWARE LIVRE

Este capítulo apresenta uma teoria para a complexidade estrutural em projetos de software livre, construída utilizando a metodologia proposta por Sjøberg Dybå, Anda, e Hannay (Sjøberg et al., 2008). Esta teoria relaciona possíveis causas e consequências da complexidade estrutural no contexto de projetos de software livre, e foi formulada como uma síntese dos resultados obtidos neste trabalho.

O restante deste capítulo está organizado da seguinte forma: a seção 6.1 introduz os conceitos fundamentais sobre teorias em Engenharia de Software segundo Sjøberg et al. A seção 6.2 apresenta uma visão geral da teoria para a complexidade estrutural em projetos de software aqui proposta. Os construtos, proposições e explicações, e escopo da teoria proposta são apresentados em detalhe nas seções 6.3, 6.4 e 6.5, respectivamente. A seção 6.6 conclui o capítulo realizando uma avaliação da teoria proposta.

6.1 TEORIAS EM ENGENHARIA DE SOFTWARE

Teorias descrevem fenômenos observáveis, de forma que se possa explicar o motivo pelo qual eles acontecem e eventualmente fazer previsões sobre ocorrências futuras desses fenômenos. Em Engenharia de Software, uma teoria deve explicar ou predizer fenômenos observáveis em Engenharia de Software (Sjøberg et al., 2008).

As previsões feitas precisam ser testáveis, de forma que as teorias possam ser refutáveis. A refutabilidade é uma propriedade importante de teorias, pois a possibilidade de serem refutadas é um incentivo para que elas sejam aperfeiçoadas com o tempo.

A descrição de uma teoria é composta da descrição de seus diferentes elementos: construtos, proposições, explicações, e escopo (Sjøberg et al., 2008).

Construtos são as entidades em termos das quais uma teoria explica um determinado fenômeno. Algumas tradições científicas assumem que os construtos de uma teoria devem ser necessariamente objetos diretamente observáveis, enquanto outras permitem a representação de conceitos teóricos não diretamente observáveis, mas cuja existência é assumida pela teoria. Os construtos presentes nas teorias em Engenharia de Software são normalmente propriedades associadas a pessoas, organizações, tecnologias, atividades, técnicas, métodos e sistemas de software. São exemplos de construtos: a experiência de desenvolvedores, o tamanho de um sistema de software, o nível de maturidade em desenvolvimento de software de uma organização, uma tecnologia em particular etc.

Proposições consistem de relacionamentos entre construtos que descrevem de que forma estes construtos interagem. Por exemplo, uma teoria em Engenharia de Software pode conter proposições como: "o uso de metodologias ágeis diminiu o tempo necessário para lançar produtos de software"; "a linguagem de programação Ruby aumenta a produtividade no desenvolvimento de aplicações web"; "o uso da instrução *goto* torna a compreensão de programas mais difícil".

Explicações se dedicam a esclarecer o motivo pelo qual as proposições são verdadeiras, isto é, a razão pelo qual determinados construtos interagem daquela forma.

O *escopo* determina as condições sob as quais a teoria se aplica. Em Engenharia de Software, por exemplo, é possível que uma determinada teoria só se aplique a organizações desenvolvedoras de software num determinado nível de maturidade de processo, ou apenas a desenvolvedores experientes, ou apenas a sistemas de software num determinado domínio de aplicação.

Segundo Sjøberg et al. (Sjøberg et al., 2008), teorias em Engenharia de Software comumente se dedicam a explicar fenômenos relacionados a situações que seguem o padrão "um *ator* aplica *tecnologias* para desempenhar *atividades* em um *sistema de software*". Os conceitos acima destacados (ator, tecnologia, atividade e sistema de software) são chamados de "classes arquetípicas" ou "arquétipos". Os autores sugerem que os elementos representados na teoria sejam associados a essas classes arquetípicas por meio de especialização e composição. Por exemplo, um projeto de desenvolvimento de software é uma especialização do arquétipo *ator*, e é composto de um ou mais desenvolvedores.

Em síntese, uma teoria em Engenharia de Software pode ser definida como uma teoria em que ao menos um construto é específico à Engenharia de Software. Os construtos numa teoria em Engenharia de Software usualmente estão associados a elementos que são, ou compõem, *atores*, *tecnologias*, *atividades*, ou *sistemas de software*.

6.2 VISÃO GERAL DA TEORIA PROPOSTA

O objetivo principal da teoria proposta é identificar as causas e consequências da evolução da complexidade estrutural em projetos de software, isto é: (i) identificar construtos que influenciam a evolução da complexidade estrutural, e (ii) identificar construtos que são influenciados pela evolução da complexidade estrutural.

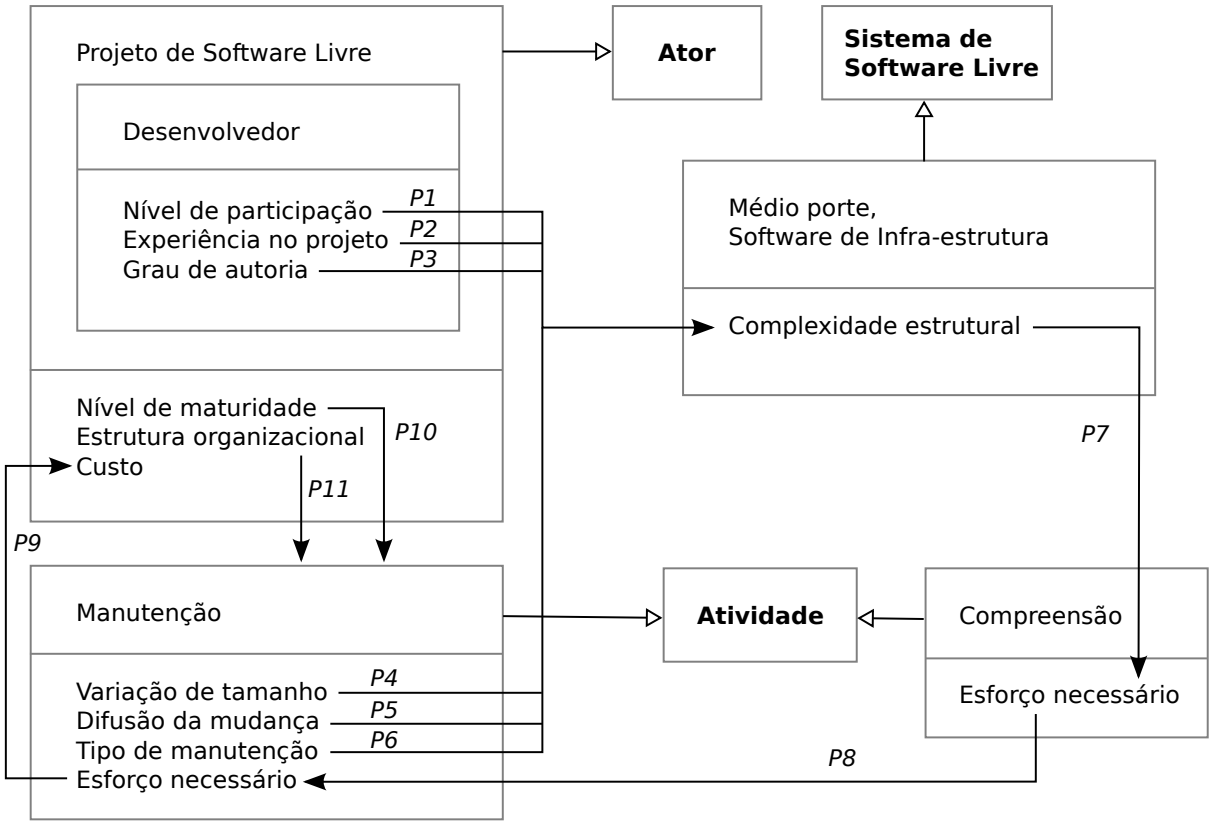


Figura 6.1 Uma teoria para a complexidade estrutural em projetos de software

A figura 6.1 traz uma representação gráfica da teoria proposta neste trabalho. A representação usa a notação proposta por Sjøberg *et al.*, parcialmente baseada em diagramas de classe da UML.

Os construtos presentes na teoria são representados por classes ou atributos de classe. Uma classe pode ser uma subclasse de outra, e nesse caso elas são conectadas pela seta de herança da UML. Por exemplo, "Projeto de Software Livre" é um tipo de ator na teoria, e isso é representado no diagrama fazendo de "Projeto de Software Livre" uma subclasse da classe arquétipo "Ator".

Um construto também pode ser componente de outro construto. Por exemplo, na teoria proposta, um desenvolvedor faz parte de um projeto de software livre. No diagrama, este fato é representado apresentando a classe "Desenvolvedor" como uma classe interna

à classe "Projeto de Software Livre".

Se um construto corresponde a um valor específico de uma variável, isto é modelado como uma subclasse, e.g. a subclasse "Médio porte, Software de infraestrutura" da classe "Sistema de Software". Já no caso de construtos para os quais se está interessado na variação de valores de uma determinada variável, então o construto é representado por um atributo de uma classe e colocado na parte de baixo da caixa onde a classe é representada, e.g. "Complexidade Estrutural".

Proposições são representadas por setas com ponta preenchida ligando dois construtos. A direção da seta representa a direção da influência entre os construtos. Por exemplo, uma seta de A para B representa a proposição "A influencia B".

6.3 CONSTRUTOS

Na teoria proposta, os atores principais são *Projetos de Software*, entendidos enquanto organizações de pessoas cujo objetivo é produzir, manter ou evoluir sistemas de software livre.

Construtos relacionados aos projetos de software livre são:

Nível de maturidade indica o nível de maturidade da organização onde o projeto é desenvolvido. Existem diferentes modelos de maturidade publicados, como por exemplo, CMMI (*Capability Maturity Model Integration*) (CMMI Product Team, 2010), MPS-BR (Melhoria do Processo de Software Brasileiro) (Softex, 2011) e, para projetos de software livre, OMM (*Open Source Maturity Model* (QualiPSo Project, 2008). Estes modelos cobrem diferentes aspectos das organizações. Por exemplo, o MPS-BR cobre o contexto específico de organizações desenvolvedoras de software brasileiras, enquanto o OMM envolve questões específicas de projetos de software livre, cujas organizações são substancialmente diferentes do que tradicionalmente se considera como uma organização desenvolvedora de software. Este construto foi descrito na seção 4.3.1.

Estrutura Organizacional na qual o projeto é desenvolvido. Determinadas estuturas organizacionais podem facilitar o desenvolvimento e a manutenção do projeto, e outras podem dificultar. Este construto foi descrito na seção 4.3.2.

Custo do projeto. O custo normalmente é considerado como uma função do esforço necessário para realizar as atividades do projeto, e pode ser mensurado em "homens-hora", "recursos", etc. O custo de um projeto é um dos principais fatores considerados na análise de sua viabilidade (Pressman, 2009).

Projetos de software livre são compostos por *desenvolvedores*, relacionados aos seguintes construtos:

Grau de Autoria indica o quanto um desenvolvedor está familiarizado com uma determinada parte de um sistema de software (Fritz et al., 2010). Pode ser usado tanto para identificar mudanças sobre as quais um determinado desenvolvedor teria interesse, como para identificar desenvolvedores que devem ser consultados sobre mudanças em determinados subsistemas. O grau de autoria foi descrito na seção 4.1.2.

Experiência no Projeto indica o volume de trabalho realizado previamente pelo desenvolvedor no projeto em questão. Na medida que um desenvolvedor ganha experiência no projeto, espera-se que ele compreenda melhor o sistema de software e consiga realizar mudanças com menor risco de introdução de defeitos (Mockus; Weiss, 2000). A experiência no projeto foi descrita na seção 4.1.1.

Nível de Participação indica o grau de envolvimento de um desenvolvedor e a sua posição social num projeto de software livre (Crowston; Howison, 2005). Desenvolvedores centrais (*core*) são aqueles que desempenham a maior parte da atividade do projeto e usualmente possuem maior poder de decisão no projeto. Desenvolvedores periféricos fazer contribuições esporádicas e exercem menor influência sobre os rumos do projeto. Este construto foi descrito na seção 4.1.3.

Manutenção é uma atividades na qual os desenvolvedores realizam mudanças num sistema de software. Os resultados desta atividade ficam registrados em sistemas de controle de versão, e nos referimos a eles como *commits*, *checkins*, etc.

Em alguns casos, as mudanças realizadas alteram a arquitetura do sistema, e o fazem ficar mais ou menos complexo. Outras mudanças não alteram a estrutura do sistema, e portanto não possuem qualquer efeito sobre a complexidade estrutural.

Os seguintes construtos estão associados à atividade de manutenção:

Variação de Tamanho indica o quanto o sistema de software cresceu (ou diminuiu) em função da realização de uma dada mudança. Conforme visto na seção 4.2.1, o tamanho de um sistema é considerado um dos fatores que influenciam a dificuldade em mudar e evoluir sistemas de software (Parnas, 1994).

Difusão da mudança representa o quanto uma dada mudança está espalhada sobre os módulos de um sistema. Espera-se que mudanças restritas a poucos módulos de um sistema sejam mais fáceis de serem realizadas, e que mudanças que envolvem uma grande quantidade de módulos apresentem um risco à futura compreensão do sistema. A difusão de mudança, descrita na seção 4.2.2, é um fator relevante na predição de riscos de uma mudança causar defeitos (Mockus; Weiss, 2000).

Tipo de manutenção indica a natureza da mudança realizada. Atividades de manutenção de software podem ser classificadas de acordo com a sua motivação como manutenção corretiva, adaptativa, perfectiva ou preventiva (Abran et al., 2004), ou de acordo com os seus resultados como correções ou melhorias (Kitchenham et al., 1999). Este construto foi descrito na seção 4.2.3.

Esforço necessário para a realização de uma mudança é um componente do custo de um projeto de software. Quanto maior for o custo para realizar mudanças num sistema, maior será o seu custo de manutenção.

Compreensão é uma atividade que faz parte da atividade de manutenção. Uma parte considerável do esforço de manutenção é dedicado à compreensão do software a ser modificado. Antes de realizar uma mudança em um sistema de software como parte de uma atividade de manutenção, o desenvolvedor precisa entender o sistema em termos da sua estrutura e funcionamento, formar um modelo mental do sistema, e então decidir como exatamente essa mudança precisa ser realizada de forma a atingir os objetivos da manutenção – por exemplo, adicionar uma nova funcionalidade, consertar um defeito ou tornar uma determinada parte do sistema mais flexível para mudanças futuras.

O seguinte construto está associado à atividade de compreensão:

Esforço necessário para a realização de uma atividade de compreensão está relacionada a diversos fatores, tais como tamanho, complexidade, familiaridade do desenvolvedor com o sistema e com o domínio da aplicação, etc.

Por fim, o construto principal da teoria proposta está relacionado a *Sistemas de Software Livre*:

Complexidade estrutural é uma propriedade de um sistema de software relacionada à sua arquitetura. Influencia a capacidade de compreensão que desenvolvedores têm sobre o sistema. A complexidade estrutural foi descrita no capítulo 3.

6.4 PROPOSIÇÕES E EXPLICAÇÕES

A seguir são listadas as proposições que compõem a teoria proposta, juntamente com suas respectivas explicações.

P1 O nível de participação dos desenvolvedores influencia a variação de complexidade estrutural causada pelas mudanças realizadas por estes desenvolvedores.

Explicação: Desenvolvedores centrais conhecem mais profundamente a arquitetura do sistema, e por isso são capazes de realizar mudanças que introduzem menos complexidade estrutural do que os desenvolvedores periféricos. Além disso, eles também são capazes de remover mais complexidade em atividades que incorrem na redução da complexidade.

P2 A experiência de desenvolvedores num dado projeto influencia a variação de complexidade estrutural causada pelas mudanças realizadas por estes desenvolvedores.

Explicação: Na medida em que o desenvolvedor ganha experiência no projeto, ele aprende sobre a arquitetura do sistema e passa a ser capaz de realizar mudanças que (i) introduzem menos complexidade ou (ii) removem mais complexidade.

P3 O grau de autoria de desenvolvedores sobre um sistema de software influencia a variação de complexidade estrutural causada pelas mudanças realizadas por estes desenvolvedores.

Explicação: Na medida em que os desenvolvedores adquirem conhecimento sobre determinados elementos do software, eles se tornam capazes de modificar aqueles elementos de forma a introduzir menos complexidade estrutural, ou de remover mais complexidade estrutural.

P4 A variação no tamanho de um sistema de software causada por uma mudança influencia a variação na complexidade estrutural causada pela mesma mudança.

Explicação: A adição de código – para incluir novas funcionalidades ou corrigir defeitos – além de tornar o sistema maior, pode também torná-lo Já uma redução no tamanho do sistema tente a torná-lo menos complexo.

P5 A difusão de uma mudança influencia a variação na complexidade estrutural causada por ela.

Explicação: Mudanças que tocam uma grande quantidade de elementos de um sistema podem ser consideradas como mudanças para as quais a arquitetura do sistema não está preparada. Essas mudanças podem levar a uma modificação na arquitetura do sistema, tornando-o mais ou menos complexo do que antes.

P6 O tipo de manutenção ao qual uma mudança está associada influencia a variação na complexidade estrutural causada por aquela mudança.

Explicação: Mudanças caracterizadas como correções tendem a ser localizadas e portanto, não alteram significantemente a organização dos elementos do software e a complexidade estrutural. Já mudanças que modificam a arquitetura do sistema – caracterizadas como melhorias, que modificam o comportamento do sistema ou a sua implementação – podem alterar a sua complexidade estrutural.

P7 A complexidade estrutural influencia positivamente o esforço necessário para a atividade de compreensão de software.

Explicação: Sistemas de software complexos dificultam a compreensão dos desenvolvedores. O ser humano tem uma capacidade limitada de compreensão, especialmente no que diz respeito a manter um grande número de elementos na memória de curto prazo (Miller, 1956; Gigerenzer; Selten, 2002). Desta forma, um aumento na complexidade do software implica em um aumento no esforço necessário para que desenvolvedores consigam compreender a arquitetura do software.

P8 O esforço necessário para compreensão de um sistema de software influencia positivamente o esforço necessário para realizar atividades de manutenção.

Explicação: Antes de realizar uma mudança sobre um sistema de software, o desenvolvedor precisa primeiramente compreender o sistema. Desta forma, um aumento no esforço para compreensão acarreta um aumento no esforço necessário para realizar atividades de manutenção.

P9 O esforço necessário para atividades de manutenção influencia positivamente os custos de projetos de software.

Explicação: Se os desenvolvedores têm maior dificuldade para compreender e modificar sistemas de software, os custos do projeto de manutenção de tal sistema tendem a subir. Os desenvolvedores precisam de mais tempo para realizar as mudanças, o que, por consequência, faz com que o projeto consiga alcançar menos resultados com os recursos disponíveis.

P10 O nível de maturidade do processo de desenvolvimento de software de uma organização influencia as mudanças realizadas no sistema de software.

Explicação: Organizações com nível mais alto de maturidade possuem processos e práticas padronizados para lidar com as mudanças, de forma a poder realizá-las com menor impacto sobre a qualidade interna e possivelmente, a complexidade estrutural do software.

P11 A estrutura organizacional onde um projeto é desenvolvido influencia as mudanças realizadas num sistema de software

Explicação: Determinadas estruturas organizacionais proporcionam uma comunicação eficiente entre os desenvolvedores de acordo com a divisão das responsabilidades do projeto facilita o processo de desenvolvimento, e portanto deve levar a uma melhor qualidade interna. Já outras estruturas organizacionais podem se mostrar inadequadas e dificultar o trabalho dos desenvolvedores, não contribuindo para uma melhor qualidade interna.

A situação das proposições acima com relação à sua validação é resumida na tabela 6.1. As proposições P1 a P5 foram exploradas durante o desenvolvimento desta tese através de estudos experimentais, que são descritos no capítulo 5.

P7 é considerada como proposição válida com base na literatura existente (Darcy et al., 2005; Midha, 2008) (capítulo 3), e é considerada como justificativa para os estudos realizados.

P8 e P9, propondo que o esforço de compreensão influencia positivamente no esforço de manutenção e que este influencia nos custos do projeto, são consideradas axiomas. Isto é, elas são assumidas como verdadeiros dentro do nosso domínio de análise, e servem de ponto de partida para inferir outros fatos na nossa teoria.

P6, P10 e P11 não puderam ser exploradas nesta tese devido a restrições de tempo.

6.5 ESCOPO DA TEORIA

Os estudos experimentais foram realizados com dados de projetos de software livre, o que faz com que os resultados não sejam generalizáveis para projetos de software em geral. Como o rótulo de "projeto de software livre" não implica nenhuma metodologia

Tabela 6.1 Situação das proposições com relação à sua validação

Proposição	Situação
P1	Explorada (seção 5.3)
P2	Explorada (seções 5.4 e 5.5)
P3	Explorada (seção 5.5)
P4	Explorada (seções 5.4 e 5.5)
P5	Explorada (seções 5.4 e 5.5)
P6	Não explorada
P7	Validada por outros pesquisadores
P8	Axioma
P9	Axioma
P10	Não explorada
P11	Não explorada

ou ferramenta específica, os resultados também não podem ser generalizados diretamente para projetos de software livre em geral.

Além disso, em função das limitações da ferramenta de análise de código fonte utilizada, os projetos analisados se limitaram a sistemas escritos em C, C++ e Java.

Os critérios de seleção de projetos para os estudos experimentais implicaram em restrições (não premeditadas) ao escopo em que a teoria foi testada. Com relação a domínios de aplicação, a grande maioria dos projetos analisados são classificados como software de infraestrutura, como servidores web, compiladores, sistemas de bancos de dados etc. Com relação ao tamanho, os sistemas analisados são de médio porte, contendo até 35.000 linhas de código.

6.6 AVALIAÇÃO DA TEORIA

Segundo Sjøberg, Dybå, Anda, e Hannay (Sjøberg et al., 2008), uma teoria em Engenharia de Software deve ser avaliada em termos dos critérios descritos a seguir.

Testabilidade. O quanto a teoria é formulada de forma a permitir refutação experimental.

Suporte experimental. O quanto a teoria é suportada por estudos experimentais que confirmam a sua validade.

Poder de explicação. O quanto a teoria abrange, e até que ponto ela prediz todas as observações dentro do seu escopo. Também, o quanto a teoria é simples, contém poucas premissas arbitrárias, e está relacionada ao que já se sabe.

Parsimônia. O quanto a teoria é formulada de forma econômica, com um mínimo de conceitos e proposiçoes.

Generalidade. A amplitude do escopo da teoria, e o quanto ela é independente de

Tabela 6.2 Resumo da avaliação da teoria proposta

Critério	Avaliação
Testabilidade	Alta
Suporte experimental	Baixo
Poder de explicação	Baixo
Parcimônia	Média
Generalidade	Baixa
Utilidade	Alta

situações específicas.

Utilidade. O quanto a teoria dá suporte a questões relevantes para a indústria de software.

A teoria sobre a complexidade estrutural em sistemas de software proposta neste trabalho foi avaliada de acordo com cada um dos critérios colocados acima. Essa avaliação, no entanto, foi realizada pelo próprio autor deste trabalho, o que representa uma ameça à sua validade. Um resumo desta avaliação é apresentado na tabela 6.2.

Testabilidade. Todos os construtos presentes na teoria são claros e precisos, de forma que são imediatamente compreensíveis e não ambíguos. A partir das proposições colocadas na figura 6.1 (capítulo 6, página 75) podemos facilmente deduzir hipóteses para teste experimental. O escopo definido para a teoria está colocado claramente. Portanto, consideramos a testabilidade da teoria como *alta*.

Suporte experimental. A tabela 6.3 apresenta o suporte experimental para as proposições presentes na teoria.

P1 (*Nível de participação influencia a complexidade estrutural*) foi confirmada no estudo descrito na seção 5.3.

O estudo apresentado na seção 5.4 apresentou suporte experimental para P2 (*Experiência do desenvolvedor no projeto influencia a complexidade estrutural*) em 2 de 10 casos, P4 (*Variação de tamanho influencia complexidade estrutural*) em 5 de 10 casos, e P5 (*Difusão de mudanças influencia a complexidade estrutural*) em 7 de 10 casos.

O estudo apresentado na seção 5.5 nos forneceu evidência experimental para P3 (*Grau de autoria do desenvolvedor influencia a complexidade estrutural*).

P7 (*Complexidade estrutural influencia o esforço necessário para compreensão de software*) foi confirmada por dois estudos de outros autores (Darcy et al., 2005; Midha, 2008), e representa a nossa justificativa para trabalhar nos fatores que influenciam a complexidade estrutural.

Como não foi realizada uma revisão sistemática da literatura técnica, a quantidade de estudos que suportam essa teoria é bastante limitada. A maioria das proposições confirmadas possuem apenas um estudo associado, exceto no caso de P7 que possui dois

Tabela 6.3 Suporte experimental às proposições.

Proposição	Suporte experimental
P1	(Terceiro; Rios; Chavez, 2010)
P2	2/10 (Terceiro et al., 2012)
P3	3-4/10 (seção 5.5)
P4	5/10 (Terceiro et al., 2012)
P5	7/10 (Terceiro et al., 2012)
P6	Não explorada.
P7	(Darcy et al., 2005; Midha, 2008)
P8	Assumida como axioma.
P9	Assumida como axioma.
P10	Não explorada.
P11	Não explorada.

estudos. Além disso, três proposições não puderam ser exploradas e outras duas foram assumidas como axioma. Desta forma, avaliamos o suporte experimental como *baixo*.

Poder de explicação. Conforme discutido nas seções 5.4 e 5.5, os modelos testados para os fatores *Experiência*, *Grau de autoria*, *Variação de Tamanho* e *Difusão da Mudança* apresentam uma grande amplitude de variação nos seus coeficientes de determinação (R^2). Em alguns projetos, uma grande parte da variação na complexidade estrutural pode ser explicada a partir dos modelos obtidos. Para outros projetos apenas uma pequena parte da variação encontra explicação, o que significa que a maior parte da variação da complexidade estrutural ainda não pode ser explicada pelos fatores incluídos nesta teoria. Além disso, o teste de hipótese utilizado no estudo sobre o *Nível de participação* (seção 5.3) foi uma comparação de médias, e envolveu uma única variável independente (o nível de participação, que indicava desenvolvedores centrais ou periféricos). Desta forma, avaliamos o poder de explicação da teoria como *baixo*.

Parcimônia. Todos os construtos e proposições apresentados são relevantes no contexto da teoria, pois representam representam aspectos importantes que devem ser levados em consideração na análise da evolução da complexidade estrutural em sistemas de software. No entanto, alguns construtos e proposições não puderam ser explorados, e ainda assim foram mantidos na teoria, mesmo sem suporte experimental. Desta forma, consideramos a parcimônia da teoria proposta como *média*.

Generalidade. Considerando o escopo proposto — projetos de software livre em C, C++ e Java, com até 35000 linhas de código e em domínios de aplicação ligados a software de infraestrutura – avaliamos que a teoria proposta possui generalidade *baixa*.

Utilidade. Conforme estudos que mostraram uma associação de complexidade estrutural com maior esforço de manutenção (Darcy et al., 2005; Midha, 2008), e o fato de que maior esforço de manutenção implica em maiores custos e maior risco, a questão da complexidade estrutural num projeto de software é de vital importância. Desta forma, avaliamos a utilidade desta teoria como *alta*.

Capítulo 7

Este capítulo conclui este trabalho discutindo os principais resultados obtidos, suas limitações e as oportunidades identificadas para trabalhos futuros.

CONCLUSÃO

Neste capítulo, são discutidas as contribuições deste trabalho (seção 7.1), bem como suas as limitações (seção 7.2). Por fim, são apresentados alguns possíveis trabalhos futuros (seção 7.3).

7.1 CONTRIBUIÇÕES

Evolução de software pode ser vista como *equilíbrio pontuado*, isto é, períodos de estabilidade e mudanças pontuais intercalados por grandes fluxos de mudanças (Wu, 2006). O estudo apresentado na seção 5.2 concluiu que pontos de descontinuidade na evolução da complexidade estrutural são um bom indicador de mudanças relevantes na arquitetura do sistema. No estudo de caso realizado, em todos os pontos onde a complexidade estrutural sofria variações bruscas, ou deixava de aumentar no mesmo ritmo, aconteceram mudanças na arquitetura do sistema. Este resultado sugere que mudanças abrutas na complexidade estrutural ou no seu ritmo de mudança podem ser usadas para identificar estes diferentes períodos na evolução de um sistema de software.

O estudo apresentado na seção 5.3 concluiu que desenvolvedores centrais introduzem menos complexidade do que desenvolvedores periféricos. Também foi verificado que desenvolvedores centrais causam maiores reduções na complexidade estrutural do que os desenvolvedores periféricos. Este resultado demonstra a importância da manutenção de uma equipe de desenvolvedores centrais num projeto de software livre. Este resultado contribui para a compreensão de uma nova dimensão da dicotomia núcleo/periferia em projetos de software livre: núcleo e periferia não só possuem interesses diversos (Dalle; Besten; Masmoudi, 2008; Masmoudi et al., 2009), comportamentos diversos na discussão de assuntos importantes para o projeto (Scialdone et al., 2009) e na atividade de relato e triagem de defeitos (Masmoudi et al., 2009), como eles também introduzem níveis diferentes de complexidade estrutural no sistema durante a atividade de programação.

Os resultados a seguir representam um primeiro passo no entendimento das causas da complexidade estrutural em sistemas de software.

O estudo apresentado na seção 5.4 alcançou diversos resultados interessantes. Um deles foi a constatação de que a variação na complexidade estrutural é influenciada por diferentes fatores em diferentes projetos. Além disso, para todos os projetos, também são diferentes os fatores que influenciam o aumento e a redução da complexidade estrutural.

Tanto o estudo apresentado na seção 5.4 quanto o estudo apresentado na seção 5.5 indicaram que todos os fatores estudados apresentaram influência sobre a variação da complexidade estrutural em pelo menos um dos modelos construídos.

No seção 5.6, foi identificado que, no caso de *commits* que aumentam a complexidade estrutural, a variação de tamanho e o número de módulos adicionados – fatores ligados à manutenção – apresentaram influência significativa sobre um número maior de projetos do que os fatores humanos como a experiência do desenvolvedor e os componentes do grau de autoria (figura 5.6, página 70).

Para os *commits* que reduziram a complexidade estrutural, os fatores relacionados à manutenção e os fatores humanos foram significativos na mesma quantidade de projetos (figura 5.7, página 71).

O estudo apresentado na seção 5.3 foi realizado com 7 servidores *web* escritos em C. Os estudos descritos nas seções 5.4 e 5.5 foram realizados com 5 projetos de diversos domínios de aplicação, escritos em C, C++ e Java. Todos os projetos analisados eram projetos de software livre. Apesar disso, acreditamos que estes resultados podem ser úteis também para projetos de software em geral. Como identificamos que projetos diferentes – e atividades diferentes – são influenciados por fatores diferentes, estes resultados não devem ser considerados de forma prescritiva, mas sim como orientações de procedimento. Para aplicá-los a um outro projeto, este deverá ser estudado previamente, seguindo uma metodologia similar à utilizada neste trabalho. Tal estudo poderá contribuir para identificar os fatores específicos que influenciam a variação da complexidade estrutural naquele projeto.

7.2 LIMITAÇÕES

A identificação dos fatores que podem influenciar a complexidade estrutural não foi feita de forma sistemática. Foi feita uma pesquisa prévia na literatura, mas não foram encontradas referências nas quais se pudesse basear esta lista de fatores. Além disso, esta pesquisa não foi realizada de forma sistemática, de forma que ela não pode ser reproduzida. Assim, os fatores foram elencados de acordo com a intuição e experiência prévia com desenvolvimento de software, e não foram fruto de uma pesquisa na literatura. Isso se deve em parte também ao fato de que a pesquisa realizada não conseguiu identificar estudos que abordassem especificamente as causas da complexidade estrutural em sistemas de software.

Teorias em Engenharia de Software podem ser construídas a partir de evidência existente através de Revisão Sistemática da Literatura, Metanálise ou através de uma abordagem baseada num Portal de Experiências (Shull; Feldmann, 2008). A teoria proposta nesta tese foi baseada em evidência existente, mas ela não foi construída com base em nenhuma destas abordagens. Nossa teoria foi construída a partir do conjunto de fatores que havia sido determinado previamente como possíveis causas da complexidade estrutural, bem como da reunião das hipóteses existentes nos estudos experimentais.

Os estudos experimentais realizados exibiram uma grande amplitude nos coeficientes de determinação (R^2) dos modelos obtidos. Para alguns projetos, conseguimos explicar uma boa parte da variação na complexidade estrutural em em função dos fatores explorados neste trabalho. Para outros projetos, por outro lado, uma pequena parcela da variação na complexidade estrutural pode ser explicada pelos fatores utilizados. Isto quer dizer que para aqueles projetos, existem outros fatores que ainda não foram explorados e que influenciam a variação na complexidade estrutural.

Fatores que merecem ser explorados podem incluir tanto aqueles que foram identificados neste trabalho, mas ainda não puderam ser explorados como, por exemplo, os fatores organizacionais, quanto outros fatores que sequer foram considerados durante a realização deste trabalho. Estudos qualitativos provavelmente são necessários para identificar outros fatores.

7.3 TRABALHOS FUTUROS

A replicação dos estudos realizados em diferentes contextos é fundamental para refinar a teoria proposta. Podem-se realizar estudos que levem em consideração fatores relacionados à organização, que não trabalhados nesta tese, como por exemplo, o nível de maturidade no desenvolvimento de software ou a estrutura organizacional da equipe de desenvolvedores. Ainda seria interessante expandir a caracterização das mudanças, como por exemplo considerar o tipo de manutenção associada a cada *commit*. Correções e melhorias podem apresentar características diferentes com relação à variação na complexidade estrutural. Além disso, ampliar o número de domínios de aplicação e linguagens de programação e diversificar o tamanho dos sistemas estudados também iria contribuir para enriquecer a teoria proposta.

Outra possibilidade de trabalho futuro é explorar diferentes aspectos da complexidade estrutural. A métrica utilizada neste trabalho para complexidade estrutural de um projeto representa a complexidade estrutural média entre os seus módulos. O uso apenas da média não leva em consideração a distribuição da complexidade entre os módulos, ou seja, a variância. Dois projetos, ou dois momentos no histórico de um mesmo projeto, que possuam a mesma complexidade estrutural média mas variâncias diferentes, podem apresentar desafios diferentes à atividade de manutenção. Por exemplo, uma baixa variância implica que a complexidade é distribuída de forma mais uniforme entre os módulos, o que pode indicar um problema estrutural grave cuja resolução demandaria bastante esforço. Já uma variância alta pode indicar que o excesso de complexidade estrutural está loca-

lizado num subconjunto do módulos, o que pode eventualmente ser solucionado através de um esforço localizado de reengenharia. Este tipo de análise pode ajudar a identificar problemas de *design* em sistemas de software.

Outro possível caminho a se seguir é aplicar esses modelos num estudo de caso com um projeto existente, e verificar de que forma modelos similares aos obtidos nos estudos quantitativos podem contribuir com o dia-a-dia de um projeto de desenvolvimento de software. O ideal seria conseguir que um projeto independente adotasse a metodologia empregada nos nossos estudos quantitativos para monitorar as suas mudanças e apoiar a tomada de decisão com relação à priorização de esforço para revisão de código e com relação à alocação de desenvolvedores para tarefas. A utilização dos resultados deste trabalho no desenvolvimento de projetos reais nos traria uma visão totalmente nova sobre o problema e provavelmente ajudaria a enriquecer a teoria.

Outra possibilidade de trabalho futuro envolve tentar identificar associação entre a métrica utilizada para complexidade estrutural e outros atributos de qualidade de software, em especial atributos de qualidade externa, como defeitos, usabilidade, etc. Diversos outros atributos de qualidade interna e externa já são utilizados com preditores de defeitos, mas não se tem notícia da complexidade estrutural ser utilizada para esta finalidade.

REFERÊNCIAS BIBLIOGRÁFICAS

Abran, A. et al. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. 2004 version. ed. Piscataway, NJ, USA: IEEE Press, 2004. 1–202 p. ISBN 0769510000. Disponível em: <http://www.swebok.org/ironman/pdf/SWEBOK_Guide_2004.pdf>.

Baldwin, C. Y.; Clark, K. B. *Design Rules: The Power of Modularity Volume 1*. Cambridge, MA, USA: MIT Press, 1999. ISBN 0262024667.

Banker, R. D.; Datar, S. M.; Kemerer, C. F. A model to evaluate variables impacting the productivity of software maintenance projects. *Manage. Sci.*, INFORMS, Institute for Operations Research and the Management Sciences (INFORMS), Linthicum, Maryland, USA, v. 37, n. 1, p. 1–18, January 1991. ISSN 0025-1909.

Barbagallo, D.; Francalenei, C.; Merlo, F. The impact of social networking on software design quality and development effort in open source projects. In: *ICIS 2008 Proceedings*. [s.n.], 2008. Disponível em: <http://aisel.aisnet.org/icis2008/201>.

Barry, E. J.; Kemerer, C. F.; Slaughter, S. A. How software process automation affects software evolution: a longitudinal empirical analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, v. 19, n. 1, p. 1–31, 2007. ISSN 1532-0618.

Bass, L.; Clements, P.; Kazman, R. *Software Architecture in Practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321154959.

Beaver, J. M.; Schiavone, G. A. The effects of development team skill on software product quality. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 31, n. 3, p. 1–5, May 2006. ISSN 0163-5948.

Bennett, K. H.; Rajlich, V. T. Software maintenance and evolution: a roadmap. In: *Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA: ACM, 2000. (ICSE '00), p. 73–87. ISBN 1-58113-253-0. Disponível em: <http://doi.acm.org/10.1145/336512.336534>.

Bland, J. M.; Altman, D. G. Multiple significance tests: the bonferroni method. *BMJ*, v. 310, n. 6973, p. 170, 1 1995.

Brooks.jr, F. P. The mythical man month: Essays on software engineering. In: _____. [S.l.]: Addison-Wesley, 1995. cap. "Aristocracy, Democracy, and System Design".

Capra, E.; Francalanci, C.; Merlo, F. An empirical study on the relationship between software design quality, development effort and governance in open source projects. *IEEE Transactions on Software Engineering*, v. 34, n. 6, p. 765–782, Nov.-Dec. 2008. ISSN 0098-5589.

Chidamber, S.; Kemerer, C. A metrics suite for object oriented design. *IEEE Trans. Sftware Eng.*, v. 20, n. 8, p. 476–493, 1994.

Clements, P. et al. *Documenting Software Architecture : Views and Beyond*. Boston: Addison-Wesley, 2002. (The SEI series in software engineering.).

CMMI Product Team. *CMMI® for Development, Version 1.3*. [S.l.], nov 2010. Disponível em: <http://www.sei.cmu.edu/library/abstracts/reports/10tr033.cfm>.

Conway, M. E. How do committees invent? *Datamation*, v. 14, n. 4, p. 28–31, 1968.

Costa, J. *Extração de Informações de Dependência entre Módulos de Programas C/C++*. [S.l.], 2009.

Costa, J. M. d. R.; Santana, F. W.; Souza, C. R. B. d. Understanding open source developers' evolution using transflow. In: *Groupware: Design, Implementation, and Use, 15th International Workshop, CRIWG 2009, Peso da Régua, Douro, Portugal, September 13-17, 2009. Proceedings*. [S.l.: s.n.], 2009. p. 65–78.

Crowston, K.; Howison, J. The social structure of free and open source software development. *First Monday*, v. 10, n. 2, 2005.

Crowston, K. et al. Core and periphery in free/libre and open source software team communications. *Hawaii International Conference on System Sciences*, IEEE Computer Society, Los Alamitos, CA, USA, v. 6, p. 118, 2006. ISSN 1530-1605.

Dalle, J.-M.; Besten, M. d.; Masmoudi, H. Channeling firefox developers: Mom and dad aren't happy yet. In: Russo, B. et al. (Ed.). *Open Source Development, Communities and Quality, IFIP 20th World Computer Congress, Working Group 2.3 on Open Source Software, OSS 2008, September 7-10, 2008, Milano, Italy*. [S.l.]: Springer, 2008. v. 275, p. 265–271. ISBN 978-0-387-09683-4.

Darcy, D. P.; Daniel, S. L.; Stewart, K. J. Exploring complexity in open source software: Evolutionary patterns, antecedents, and outcomes. *Hawaii International Conference on System Sciences*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 1–11, 2010.

Darcy, D. P. et al. The structural complexity of software: An experimental test. *IEEE Transactions on Software Engineering*, v. 31, n. 11, p. 982–995, Nov. 2005. ISSN 0098-5589.

Fritz, T. et al. A degree-of-knowledge model to capture source code familiarity. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA: ACM, 2010. (ICSE '10), p. 385–394. ISBN 978-1-60558-719-6.

Gigerenzer, G.; Selten, R. (Ed.). *Bounded rationality: the adaptive toolbox*. [S.l.]: MIT Press, 2002.

Hassan, A. E. Predicting faults using the complexity of code changes. In: *Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009. (ICSE '09), p. 78–88. ISBN 978-1-4244-3453-4. Disponível em: <http://dx.doi.org/10.1109/ICSE.2009.5070510>.

Herraiz, I.; Hassan, A. E. Beyond lines of code: Do we need more complexity metrics? In: Oram, A.; Wilson, G. (Ed.). *Making Software: What Really Works, and Why We Believe It*. Sebastopol, CA, USA: O'REILLY, 2011.

Hitz, M.; Montazeri, B. Measuring coupling and cohesion in object-oriented systems. In: *Proceedings of the International. Symposium on Applied Corporate Computing*. [S.l.: s.n.], 1995.

Jay, G. et al. Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship. *Journal of Software Engineering and Applications*, v. 2, n. 3, p. 137–143, 2009.

Jensen, C.; Scacchi, W. Modeling recruitment and role migration processes in ossd projects. In: *In Proceedings of the 6th International Workshop on Software Process Simulation and Modeling*. [S.l.: s.n.], 2005.

Jensen, C.; Scacchi, W. Role migration and advancement processes in ossd projects: A comparative case study. In: *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007. p. 364–374. ISBN 0-7695-2828-7.

Kitchenham, B. A. et al. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 28, n. 8, p. 721–734, August 2002. ISSN 0098-5589. Disponível em: <http://portal.acm.org/citation.cfm?id=636196.636197>.

Kitchenham, B. A. et al. Towards an ontology of software maintenance. *Journal of Software Maintenance*, John Wiley & Sons, Inc., New York, NY, USA, v. 11, n. 6, p. 365–389, November 1999. ISSN 1040-550X. Disponível em: <http://dl.acm.org/citation.cfm?id=334928.334929>.

Kon, F. et al. Free and open source software development and research: Opportunities for software engineering. In: . [S.l.: s.n.], 2011.

Lehman, M.; Belady, L. *Program Evolution: Processes of Software Change*. London: London Academic Press, 1985.

Lehman, M. M. et al. Metrics and laws of software evolution-the nineties view. In: *Proceedings of the 4th International Symposium on Software Metrics*. [S.l.: s.n.], 1997.

Lientz, B. P.; Swanson, E. B.; Tompkins, G. E. Characteristics of application software maintenance. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 6, p. 466–471, June 1978. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359511.359522>.

Martin, R. C. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003. ISBN 0135974445.

Masmoudi, H. et al. "peeling the onion": The words and actions that distinguish core from periphery in bug reports and how core and periphery interact together. In: Boldyreff, C. et al. (Ed.). *OSS: Diverse Communities Interacting, 5th IFIP WG 2.13 International Conference on Open Source Systems, OSS 2009, Skövde, Sweden, June 3-6, 2009. Proceedings*. [S.l.]: Springer, 2009. v. 299, p. 284–297. ISBN 978-3-642-02031-5.

Matsumoto, S. et al. An analysis of developer metrics for fault prediction. In: *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. New York, NY, USA: ACM, 2010. (PROMISE '10), p. 18–1. ISBN 978-1-4503-0404-7.

Mccabe, T. J. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2, n. 4, p. 308–320, dec. 1976. ISSN 0098-5589.

Mcconnell, S. *Code complete*. [S.l.]: Microsoft Press, 2004. (DV-Professional). ISBN 9780735619678.

Meirelles, P. et al. A study of the relationships between source code metrics and attractiveness in free software projects. In: *CBSOFT-SBES2010*. Los Alamitos, CA, USA: IEEE Computer Society, 2010. p. 11–20. ISBN 978-0-7695-4273-7.

Meulen, M. J. P. van der; Revilla, M. A. Correlations between internal software metrics and software dependability in a large population of small c/c++ programs. In: *Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on*. [S.l.: s.n.], 2007. p. 203–208. ISSN 1071-9458.

Midha, V. Does complexity matter? the impact of change in structural complexity on software maintenance and new developers' contributions in open source software. In: *ICIS 2008 Proceedings*. [S.l.: s.n.], 2008.

Miller, G. A. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, v. 63, p. 81–97, 1956. Available online at `http://www.musanim.com/miller1956/`.

Mitchell, M. *Complexity - A Guided Tour*. [S.l.]: Oxford University Press, 2009.

Mockus, A.; Fielding, R. T.; Herbsleb, J. D. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM, New York, NY, USA, v. 11, n. 3, p. 309–346, 2002. ISSN 1049-331X.

Mockus, A.; Herbsleb, J. D. Expertise browser: a quantitative approach to identifying expertise. In: *Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM, 2002. (ICSE '02), p. 503–512. ISBN 1-58113-472-X.

Mockus, A.; Weiss, D. M. Predicting risk of software changes. *Bell Labs Tech. J.*, Wiley Subscription Services, Inc., A Wiley Company, Software Production Research Department, Bell Labs, Naperville, Illinois, v. 5, n. 2, p. 169–180, 2000. ISSN 1538-7305.

Nagappan, N.; Murphy, B.; Basili, V. The influence of organizational structure on software quality: an empirical case study. In: *Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008. (ICSE '08), p. 521–530. ISBN 978-1-60558-079-1. Disponível em: <http://doi.acm.org/10.1145/1368088.1368160>.

Ostrand, T. J.; Weyuker, E. J.; Bell, R. M. Programmer-based fault prediction. In: *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. New York, NY, USA: ACM, 2010. (PROMISE '10), p. 19–1. ISBN 978-1-4503-0404-7.

Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, ACM, New York, NY, USA, v. 15, n. 12, p. 1053–1058, December 1972. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/361598.361623>.

Parnas, D. L. Software aging. In: *ICSE '94: Proceedings of the 16th international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994. p. 279–287. ISBN 0-8186-5855-X.

Pressman, R. *Software Engineering: A Practitioner's Approach*. 7th edition. ed. [S.l.]: McGraw-Hill, 2009.

QualiPSo Project. *OMM – The QualiPSo Open Maturity Model*. 2008. Meio eletrônico. Disponível em `http://qualipso.icmc.usp.br/OMM/`. Último acesso em 16 de novembro de 2011.

Robles, G.; Gonzalez-barahona, J. Contributor turnover in libre software projects. *Open Source Systems*, p. 273–286, 2006.

Robles, G.; Gonzalez-barahona, J. M.; Herraiz, I. Evolution of the core team of developers in libre software projects. In: *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*. [S.l.: s.n.], 2009. p. 167–170.

Roca, J. L. An entropy-based method for computing software structural complexity. *Microelectronics and Reliability*, v. 36, n. 5, p. 609–620, 1996. ISSN 0026-2714.

Sangwan, R. S.; Vercellone-smith, P.; Laplante, P. A. Structural epochs in the complexity of software over time. *Software, IEEE*, v. 25, n. 4, p. 66–73, july-aug. 2008. ISSN 0740-7459.

Santos Jr., C. D. et al. Intellectual property policy and attractiveness: a longitudinal study of free and open source software projects. In: *CSCW*. [S.l.: s.n.], 2011. p. 705–708.

Scacchi, W. Free/open source software development: recent research results and emerging opportunities. In: *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007, Companion Papers*. [S.l.]: ACM, 2007. p. 459–468.

Scialdone, M. J. et al. Group maintenance behaviors of core and peripherial members of free/libre open source software teams. In: Boldyreff, C. et al. (Ed.). *OSS: Diverse Communities Interacting, 5th IFIP WG 2.13 International Conference on Open Source Systems, OSS 2009, Skövde, Sweden, June 3-6, 2009. Proceedings*. [S.l.]: Springer, 2009. v. 299, p. 298–309. ISBN 978-3-642-02031-5.

Sedlmeyer, R. L. et al. Problems with software complexity measurement. In: *Proceedings of the 1985 ACM thirteenth annual conference on Computer Science*. New York, NY, USA: ACM, 1985. (CSC '85), p. 340–347. ISBN 0-89791-150-4.

Shull, F.; Feldmann, R. L. Building theories from multiple evidence sources. In: Shull, F.; Singer, J.; Sjøberg, D. I. K. (Ed.). *Guide to Advanced Empirical Software Engineering*. [S.l.]: Springer London, 2008. p. 337–364. ISBN 978-1-84800-044-5.

Simon, H. A. *The Sciences of the Artificial*. Cambridge, Massaschusetts, London, England: The MIT Press, 1996.

Sjøberg, D. I. K. et al. Building theories in software engineering. In: Shull, F.; Singer, J.; Sjøberg, D. I. K. (Ed.). *Guide to Advanced Empirical Software Engineering*. [S.l.]: Springer London, 2008. p. 312–336. ISBN 978-1-84800-044-5.

Softex. *MPS.BR - Melhoria de Processo do Software Brasileiro*. 2011. Meio eletrônico. Disponível em `http://www.softex.br/mpsbr/`. Último acesso em 16 de novembro de 2011.

Sommerville, I. *Software Engineering*. 9th edition. ed. New York: Addison Wesley, 2010.

Steen, O. Practical knowledge and its importance for software product quality. *Inf. Softw. Technol.*, Butterworth-Heinemann, Newton, MA, USA, v. 49, n. 6, p. 625–636, June 2007. ISSN 0950-5849.

Stewart, K. J.; Darcy, D. P.; Daniel, S. L. Opportunities and challenges applying functional data analysis to the study of open source software evolution. *Statistical Science*, v. 21, p. 167, 2006.

Stol, K.-J.; Babar, M. A. Reporting empirical research in open source software: The state of practice. In: Boldyreff, C. et al. (Ed.). *OSS: Diverse Communities Interacting, 5th IFIP WG 2.13 International Conference on Open Source Systems, OSS 2009, Skövde, Sweden, June 3-6, 2009. Proceedings*. [S.l.]: Springer, 2009. v. 299, p. 156–169. ISBN 978-3-642-02031-5.

Swanson, E. B. The dimensions of maintenance. In: *Proceedings of the 2nd international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976. (ICSE '76), p. 492–497. Disponível em: <http://dl.acm.org/citation.cfm?id=800253.807723>.

Tegarden, D. P.; Sheetz, S. D.; Monarchi, D. E. A software complexity model of object-oriented systems. *Decis. Support Syst.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 13, n. 3-4, p. 241–262, mar 1995. ISSN 0167-9236. Disponível em: <http://dx.doi.org/10.1016/0167-9236(93)E0045-F>.

Terceiro, A.; Chavez, C. Structural complexity evolution in free software projects: A case study. In: Babar, M. A.; Lundell, B.; Linden, F. van der (Ed.). *QACOS-OSSPL 2009: Proceedings of the Joint Workshop on Quality and Architectural Concerns in Open Source Software (QACOS) and Open Source Software and Product Lines (OSSPL)*. [S.l.: s.n.], 2009.

Terceiro, A. et al. Analizo: an extensible multi-language source code analysis and visualization toolkit. In: *CBSOFT-Ferramentas*. [S.l.: s.n.], 2010.

Terceiro, A. et al. Understanding structural complexity evolution: a quantitative analysis. In: *16th European Conference on Software Maintenance and Reengineering*. [S.l.: s.n.], 2012.

Terceiro, A.; Rios, L. R.; Chavez, C. An empirical study on the structural complexity introduced by core and peripheral developers in free software projects. *Brazilian Symposium on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 21–29, sep 2010.

Williams, B. J.; Carver, J. C. Characterizing software architecture changes: A systematic review. *Inf. Softw. Technol.*, Butterworth-Heinemann, Newton, MA, USA, v. 52, n. 1, p. 31–51, January 2010. ISSN 0950-5849. Disponível em: <http://dl.acm.org/citation.cfm?id=1645441.1645568>.

Wilson, G.; Aranda, J. Empirical software engineering. *American Scientist*, v. 99, n. 6, p. 466, 2011.

Wohlin, C. et al. *Experimentation in Software Engineering: an Introduction*. [S.l.]: Kluver Academic Publishers, 2000.

Wu, J. *Open source software evolution and its dynamics*. Tese (Doutorado) — University of Waterloo, Waterloo, Ont., Canada, Canada, 2006.

Zhang, M.; Baddoo, N. Performance comparison of software complexity metrics in an open source project. In: *Proceedings of Software Process Improvement, 14th European Conference, EuroSPI 2007, Potsdam, Germany, September 26-28, 2007*. [S.l.: s.n.], 2007. p. 160–174.

Zhou, M.; Mockus, A. Developer fluency: achieving true mastery in software projects. In: *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2010. (FSE '10), p. 137–146. ISBN 978-1-60558-791-2.

Østerlie, T.; Jaccheri, L. A critical review of software engineering research on open source software development. In: Stanislaw, W. (Ed.). *Proceedings of the 2nd AIS SIGSAND European Symposium on Systems Analysis and Design*. [S.l.]: Gdansk University Press, 2007. p. 12–20. ISBN 978-83-7326-447-2.

Apêndice A

Este apêndice contém uma reprodução de artigo publicado com o mesmo título, nos anais do Joint Workshop on Quality and Architectural Concerns in Open Source Software (QACOS) and Open Source Software and Product Lines (OSSPL) – QACOS/OSPL 2009.

STRUCTURAL COMPLEXITY EVOLUTION IN FREE SOFTWARE PROJECTS: A CASE STUDY

Structural Complexity Evolution in Free Software Projects: A Case Study

Antonio Terceiro and Christina Chavez

Computer Science Department – Universidade Federal da Bahia
{terceiro,flach}@dcc.ufba.br

Abstract. The fundamental role of Free Software in contemporary IT Industry is sometimes threatened by the lack of understanding of its development process, which sometimes leads to distrust regarding quality of Free Software projects. While Free Software projects do have quality assurance activities, there is still room for improvement and introduction of new methods and tools. This paper presents a case study of structural complexity evolution in a small project written in C. The approach used can be used by Free Software developers in their own projects to understand and control complexity, and the results provide initial understanding on the study of structural complexity evolution.

1.1 Introduction

Free Software[1] plays a fundamental role in the IT Industry in the contemporary society. Companies, governments and non-profit organizations recognize the potential of Free Software and are at least considering it. But Free Software is developed in a way too much different from the "conventional" software these organizations are used to: teams are distributed throughout the world, most of the time with no contractual obligations; normally, there are no formal requirements specifications, in the sense of those we expect in traditional software development organizations; quality assurance does happen, while not in the way an outsider expects (and perhaps it's not as good as it could be). Quoting Scacchi [Scacchi, 2007], Free Software Development is not Software Engineering done poorly, it's just different.

While there is quality assurance (QA) activities in Free Software projects [Zhao and Elbaum, 2003], there is a lot of room for improvement in this area. One of the areas in which Free Software QA can advance is at the use of project metrics to inform, guide and control development. It's not uncommon to find projects in which the lead developer(s) lost the interest and there are users/contributors willing to continue the development, but the complexity of the code makes it more practical for them to start a new project as a replacement to the original. Sometimes the lead developer(s) realize that the code

[1] also referred to in the research community as "Open Source Software" (OSS), "Free/Open Source Software" (FOSS), "Free/Libre/Open Source Software" (FLOSS) etc.

became so complex that it's more cost-effective for them to rewrite large parts of the software, or even to rewrite it entirely from scratch, than investing time in enhancing existing code.

So, if Free Software developers have tools and methods to understand and tame the complexity of their code, there will be a more healthy Free Software ecosystem. Less complex code may promote maintainability and help Free Software projects to get and retain new contributors.

This paper goals are twofold: first, to experiment an approach for studying the evolution of structural complexity in Free Software projects that can be used by Free Software developers in their own projects; second, to provide initial results on the study of structural complexity evolution in Free Software projects written in C (so we can e.g. compare them with the Java projects studied in [Stewart et al., 2006]). For that, we present a case study in which we analyze the evolution of structural complexity in a small Free Software project during a period of approximately 15 months.

The remainder of this paper is organized as follows: related work is described in section 1.2; section 1.3 briefly describes the tool infrastructure used to extract structural complexity metrics from projects written in the C language; section 1.4 presents the case study; and finally, we provide final remarks and discuss future work in section 1.5.

1.2 Related Work

Stewart and colleagues studied 59 projects written in Java that were available on Sourceforge [Stewart et al., 2006], using as a software complexity measure the metric "CplXLCoh", the product of Coupling ("Cpl") and Lack of Cohesion ("LCoh"). They found four patterns on software complexity evolution among those projects: 1) early decreasers, in which the complexity starts to decrease in the very beginning of the project's public availability, then gets stable for a period, and then starts a slight growing trend; 2) early increasers, where the complexity starts to increase just during the beginning of the project, and after some time gets stable; 3) midterm increasers, which experiences a faster growing of the complexity several months after the start of the project; and 4) midterm decreasers, that continued to decrease complexity during the middle of the observed period before stabilizing.

Capiluppi and Boldyreff [Capiluppi and Boldyreff, 2007] presented an approach to use coupling information to indicate potentially reusable parts of projects, which could be distributed as independent projects and reused by other software in the same application domain or with similar non-functional requirements. Their approach was based on a instability metric, as in the work of Martin [Martin, 2003]. This metric is defined in terms of afferent coupling (number of modules calling the module in question, C_a) and efferent coupling (number of modules that the module in question calls, C_e), as $I = \frac{C_e}{C_a + C_e}$. They show that modules (represented in their study by folders) with low insta-

bility (i.e. stable modules) are good candidates to be turned into independent, external modules.

Wu [Wu, 2006] studied the dynamics of Free Software projects evolution, and argues that it happens in the form of punctuated equilibrium: the projects alternate between periods of localized and incremental changes and periods with deep architectural changes.

1.3 `egypt`: tool support for extracting coupling and cohesion data from C programs

`egypt` is program originally developed by Andreas Gustafsson[2]. It works by reading intermediate files generated by the GNU C Compiler and producing as output a call graph in the format used by the Graphviz graph visualization software [3], so we can visualize the call dependencies between functions in C source code.

We made the following main modifications in `egypt` to use in this study:

- Implementation of variables usage detection, to identify which functions use which variables. This is used for calculating both coupling between modules (in the case where modules use variables from other modules) and lack of cohesion for a given module.
- Addition of an option to group the calls and variable usages by module, so that we can have a module dependency view. The original `egypt` only produced graphs at the function level, which makes it impossible to understand the structure of non-trivial software.
- Refactoring of the `egypt` script into an object-oriented design to be able to plug different extraction and reporting modules.
- Implementation of a metrics output which, instead of producing Graphviz input files, produces a metrics report on the extracted design including coupling and cohesion data.

Our modified version is available as a git repository at `http://github.com/terceiro/egypt`.

Since `egypt` extracts coupling data, it can also be used to carry studies like the one by Capiluppi and Boldyreff [Capiluppi and Boldyreff, 2007] to identify potentially reusable modules in Free Software Projects. This is not our interest in this paper, though.

1.4 Case study: the Ristretto project

Ristretto[4] is a fast and lightweight picture-viewer for the Xfce desktop environment. It's written in C, uses the GTK+ user interface toolkit and is licensed

[2] `http://www.gson.org/egypt/`
[3] `http://www.graphviz.org/`
[4] `http://goodies.xfce.org/projects/applications/ristretto`

under the GNU General Public License, version 2 or later. In this study we **analyze** the Ristretto project **with the goal of** characterizing it **with respect to** size and complexity over time **from the perspective** of developers.

1.4.1 Planning

In this study, our "population" is the series of releases the Ristretto project had since the beginning of its development. This includes 21 consecutive releases, from 0.0.1 to 0.0.21, spanning a period of approximately 15 months.

For each release of the project, the following data was extracted:

- Independent variables:
  - Release day (RD): the number of days that has passed since the first release. The first release itself has $RD = 0$.
- Dependent variables:
  - Physical Source Lines of Code ($SLOC$).
  - The product of average module coupling and average module lack of cohesion ($CplXLCoh$), as in [Stewart et al., 2006], as a measure of complexity. We have used the classic coupling and lack of cohesion metrics by Chidamber and Kemerer [Chidamber and Kemerer, 1994].

We formulated two hypothesis for this study, which are described below.

Hypothesis 1. We want to test whether the project presents a consistent growth, as reported in the literature for both "conventional" Software Engineering [Lehman et al., 1997] and for Free Software projects [Koch, 2007]. Since we are interested only in testing for consistent growth and don't need a precise prediction of project size, we consider enough to test for a linear correlation between the date of release and the size of the project. Our null hypothesis H_0^1 is that there is no linear correlation between time and size of the project, and our alternative hypothesis H_A^1 stands for a positive linear correlation between the variables:

$$H_0^1 : r_{RD,SLOC} = 0 \qquad H_A^1 : r_{RD,SLOC} > 0$$

Hypothesis 2. To test whether the project becomes more complex as the time passes, we want to verify how does our complexity metric evolve. The theory suggests that software projects tend to become more complex through time, unless explicit actions are taken to prevent it [Lehman et al., 1997]. Our null hypothesis H_0^2, then, says there is no linear correlation between time and complexity; our alternative hypothesis H_A^2 is that there is a positive linear correlation between them:

$$H_0^2 : r_{RD,CplXLCoh} = 0 \qquad H_A^2 : r_{RD,CplXLCoh} > 0$$

1.4.2 Data Extraction

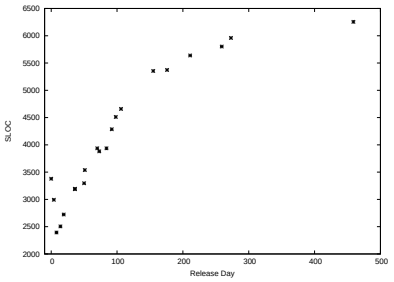
For measuring *SLOC*, we used David A. Wheeler's `sloccount` tool, available at `http://www.dwheeler.com/sloccount/`. For each release, `sloccount` is run and only the total Physical Source Lines of Code count is taken.

For measuring *CplXLCoh*, we used our modified version of `egypt`.

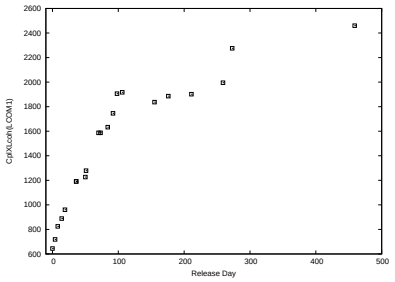
Ristretto's Subversion repository was imported into a `git`[5] repository. We then used a script that iterates through all the releases, gets the release date from the version control data, checks out the source code of the given release, invokes `sloccount` to get the size of the project at that release, builds the project so the GNU C Compiler generates the needed intermediate files, and invokes `egypt` to extract the design information from the GCC intermediate files.

1.4.3 Data analysis

Figure 1(a) presents the evolution of size in the Ristretto project, using the day of release as data in the X axis (while using the version string as label), and *SLOC* as the Y-axis. The plot shows that Ristretto is consistently growing: it goes from approximately 2500 *SLOC* in the first release to more than 6000 *SLOC* in the last observed release. On the other hand, looking at the latest releases makes us wonder if the growth rate isn't decreasing.



(a) Growth of Ristretto project over time



(b) Evolution of the complexity metric in the Ristretto project

Fig. 1.1. Ristretto evolution data

The correlation test, using Pearson's method, gives us $r_{RD,SLOC} = 0.9041602$, with $p < 0.01$. This way we are able to reject the null hypothesis H_0^1 and accept the alternative hypothesis H_A^1: the current data allows us to say that there is a positive linear correlation between the release day and the size, measured in Physical Source Lines of Code.

[5] `http://www.git-scm.org/`

Figure 1(b) shows the data for our complexity metric. The plot shows that although the complexity in increasing, there are specific releases in which either the complexity does not increase significantly or even the complexity decreases in comparison with the previous release. This behavior is discussed in more detail in the section 1.4.4

The correlation test for RD and $CplXLCoh$ gave us $r_{RD,CplXLCoh} = 0.8636375$ with $p < 0.01$, also using Pearson's method. This way we can reject our null hypothesis H_0^2 and accept our alternative hypothesis: there is a linear correlation between release day and complexity.

1.4.4 Interpreting the Results

The growth data allows us to assert that the project is consistently growing since it's first release. This suggests it's being actively developed and is receiving new features as an effect of new user requirements. But growth is not out main interest in this study.

The complexity data reveals interesting issues. In the long term, the complexity grows as the time passes, but the curve we could draw through the data points shows discontinuities (see figure 1(b)):

1. Approximately in the 40th day, in the 6th release of Ristretto, the complexity increase is attenuated.
2. Approximately in the 150th day, in the 16th release, the complexity *decreases* in comparison with the previous release.
3. Approximately in the 450th day, in the 21th and last release, the complexity also seems to increase less in comparison with the previous release than it increased in the previous release in comparison with the release before it.

All these discontinuities coincide with major architectural changes in Ristretto: releases 0.0.6, 0.0.16 and 0.0.21 introduced new modules in comparison with their respective previous releases. Figure 1.2 shows the four different architectures Ristretto had during its life cycle: the leftmost graph shows the architecture of the first release, and them the architecture in releases 0.0.6, 0.0.16 and 0.0.21.

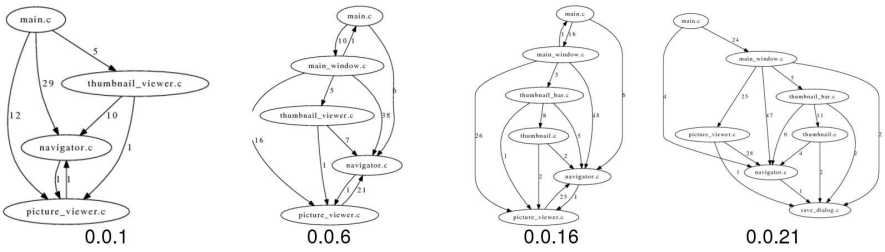


Fig. 1.2. Ristretto's architectural evolution. Graphs by Graphviz from `egypt` output.

It is easy to understand why the introduction of new modules has an impact in complexity increase. As the newly introduced module tends to be less complex than the previously existing ones, and the sum of the complexities is now divided by $n + 1$ instead of n, the new module brings down the average complexity metric.

Ristretto exhibits the behavior described by Wu [Wu, 2006]: it alternates periods of incremental change, in which the complexity increases, with moments of rupture in which the architecture changes. In the case of Ristretto, these architecture changes made it less complex, or at least attenuated the complexity increase trend at that moment.

1.4.5 Limitations of this study

We chose a small project to study on purpose, since we wanted to do a exploratory study and experiment the approach we are developing. A larger project may not exhibit a similar behavior as Ristretto.

Although the version control history data lists 13 different contributors to the Ristretto source folder, we later identified that actually only one developer made changes to the source code. The other contributors' changes were mainly updates to user interface translations, which are separated from the C source code. This way we have no data that allows us to investigate the relationship between the structural complexity and the number of developers who contributed in a given period (for example).

1.5 Conclusions

This paper presented a case study on structural complexity evolution. We analyzed 21 versions of the Ristretto project, and concluded it grows consistently, and its structural complexity increases as time passes. Both size and complexity metrics have a high correlation with the release day. We identified that specific releases where structural complexity decreases or starts to increase more slowly than in the previous release seem to be related to significant architectural changes. These changes, in the case of this small project, were additions of new modules.

We believe that our approach can be used to make larger-scale studies. These include individual studies with projects larger than Ristretto and studies comparing the structural complexity evolution of different projects. By comparing several different projects, perhaps we'll be able to associate different patterns of structural complexity evolution with characteristics of the projects. In special, it would be interesting to compare the results of studying C projects with the results of the Java projects studied by Stewart et al [Stewart et al., 2006]. Ristretto, as this paper has shown, seems to belong to the *early increasers* group described by them. Other type of study that may produce interesting results is studying structural complexity evolution in a more fine-grained scale: instead

of analyzing only the released source code, we can use the history stored in the version control system and analyze every single revision to identify the exact changes that introduced either an increase in complexity or a refactoring that made the software less complex.

The approach used in this paper can also be used by Free Software developers to monitor the structural complexity of their C projects. By using the `egypt` tool to obtain both design graphs and metrics, they can verify whether a specific change increases the overall system complexity or if a refactoring reduced the complexity in comparison with a given previous state of the code. They can also inspect the history of the projects in points in which the complexity decreased to learn important lessons about their own projects.

References

[Capiluppi and Boldyreff, 2007] Capiluppi, A. and Boldyreff, C. (2007). Coupling patterns in the effective reuse of open source software. In *FLOSS '07: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, page 9, Washington, DC, USA. IEEE Computer Society.

[Chidamber and Kemerer, 1994] Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

[Koch, 2007] Koch, S. (2007). Software evolution in open source projects—a large-scale investigation. *J. Softw. Maint. Evol.*, 19(6):361–382.

[Lehman et al., 1997] Lehman, M., Ramil, J., Wernick, P., and Perry, D. (1997). Metrics and laws of software evolution-the nineties view. In *Proceedings of the 4th International Symposium on Software Metrics*.

[Martin, 2003] Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[Scacchi, 2007] Scacchi, W. (2007). Free/open source software development: Recent research results and methods. In Zelkowitz, M. V., editor, *Advances in Computers*, volume 69, pages 243–269. 2007 edition.

[Stewart et al., 2006] Stewart, K. J., Darcy, D. P., and Daniel, S. L. (2006). Opportunities and challenges applying functional data analysis to the study of open source software evolution. *Statistical Science*, 21(2):167–178.

[Wu, 2006] Wu, J. (2006). *Open source software evolution and its dynamics*. PhD thesis, University of Waterloo, Waterloo, Ont., Canada, Canada.

[Zhao and Elbaum, 2003] Zhao, L. and Elbaum, S. (2003). Quality assurance under the open source development model. *J. Syst. Softw.*, 66(1):65–75.

Apêndice B

Este apêndice contém uma reprodução de artigo publicado com o mesmo título nos anais do XXIV Simpósio Brasileiro de Engenharia de Software em 2010.

AN EMPIRICAL STUDY ON THE STRUCTURAL COMPLEXITY INTRODUCED BY CORE AND PERIPHERAL DEVELOPERS IN FREE SOFTWARE PROJECTS

An Empirical Study on the Structural Complexity Introduced by Core and Peripheral Developers in Free Software Projects

Antonio Terceiro, Luiz Romário Rios, Christina Chavez
Software Engineering Lab (LES)
Computer Science Department
Federal University of Bahia
{terceiro,luizromario,flach}@dcc.ufba.br

***Abstract*—Background: Several factors may impact the process of software maintenance and evolution of free software projects, including structural complexity and lack of control over its contributors. Structural complexity, an architectural concern, makes software projects more difficult to understand, and consequently more difficult to maintain and evolve. The contributors in a free software project exhibit different levels of participation in the project, and can be categorized as core and peripheral developers.**

Research aim: This research aims at characterising the changes made to the source code of 7 web server projects written in C with respect to the amount of structural complexity added or removed and the developer level of participation.

Method: We performed a observational study with historical data collected from the version control repositories of those projects, recording structural complexity information for each change as well as identifying each change as performed by a core or a peripheral developer.

Results and conclusions: We have found that core developers introduce less structural complexity than peripheral developers in general, and that in the case of complexity-reducing activities, core developers remove more structural complexity than peripheral developers. These results demonstrate the importance of having a stable and healthy core team to the sustainability of free software projects.

I. Introduction

The problem of software aging, introduced by Parnas in 1994, is a well-known problem in the Software Engineering field [1]. It has two causes: the first is "lack of movement", when a software project fails to deliver an updated product that fulfils the changing needs of its users. The second cause of software aging is "ignorant surgery": the software is subsequently changed by people who do not fully understand its design, and after some time even the creators of the design do not understand it anymore; changing the software becomes harder and more error prone.

According to Parnas, software aging is inevitable, just like human aging. While there are measures that can be taken to slow down the effects of software aging, even successful projects will get old and suffer the consequences of aging: older software requires more effort for being updated as time passes, bug fixes tend to cause new bugs, and can even exhibit unsatisfactory performance.

The aging phenomenon can be also observed in the context of free software[1] projects. For instance, GNOME is a worldwide project that provides the GNOME desktop environment, a desktop system, and the GNOME development platform, a framework for building applications that integrate into the rest of the desktop. GNOME had two of its components, `eog`, GNOME's image viewer and `gnome-session`, GNOME's session management software, completely rewritten from scratch.[2]

In older projects, the code base becomes so difficult to maintain that their maintainers decide that rewriting them will require less effort than to keep maintaining it. Sometimes the developers believe that rewriting their project (or large parts of it) is absolutely necessary for them to be able to evolve the project. Less complex code may facilitate the addition of new features and bug fixing in a sustainable way and therefore supports maintainability.

These rewrites take time and effort, so every project leader would certainly prefer not to deal with them. If the factors that contribute to added complexity in free software projects can be identified and characterized, project leaders would possibly be able to avoid such rewrites.

Several factors may impact the process of software maintenance and evolution. The internal quality of the software that is to be evolved and maintained is one of the most important: the lower the quality of the source code and other artifacts, the larger is the effort to change them. In this context, the *structural complexity* of the source code is one of the dimensions of the internal quality.

One of the characteristics of free software projects is the lack of control over the project developers. They usually join

[1] in our work, "free software" is used as a synonym for "open source software" (OSS), 'free/open source software" (FOSS) and "free/libre/open source software"(FLOSS).

[2] These rewrites were described in their wiki pages, respectively http://live.gnome.org/EyeOfGnome/EogNg and http://live.gnome.org/SessionManagement/NewGnomeSession.

and leave projects based on their motivation or needs, and present different *levels of participation*. The convention is to categorize them as either *core developers* or *peripheral developers*. The former are the most active developers, who perform most of the work, and are in general in charge of the important decisions regarding the project. The later contribute less often, and normally have little decision power in the project [2], [3].

In this paper, we explore the different levels of developers participation as a factor that may influence the amount of structural complexity in free software projects. We want to verify whether core and peripheral developers introduce different amounts of structural complexity in the code, and also whether they remove different amounts of complexity during activities that reduce the complexity of the source code, such as refactorings.

The remainder of this paper is organized as follows: section II presents the background for this study; section III presents the hypotheses that we investigate in this paper; section IV describes our research design; in section V we analyse the results obtained; section VI discusses threats to the validity of this study; section VII discusses similar studies and compares our study with them; finally, in section VIII we discuss our results as well as possibilities of future work.

II. Background

The following sections present background in the main subjects addressed by this paper: free software projects, structural complexity of software and the concept of Core and Periphery as different levels of contribution in free software projects.

A. Free Software

We call "free software" every software product that is available to its users under a license that allows it to be freely[3] studied, modified, and redistributed, according to the "Free Software Definition" [4], published by the Free Software Foundation.

The most interesting aspect of the nature of free software, from a Software Engineering point of view, is its development process. A free software project starts when an individual developer, or an organization, decides to make a software publicly available on the internet so that it can be freely used, modified and distributed. After an initial version is released, and with some effort in advertising it in the appropriate channels, independent users start using it, and those users who are also software developers are able to inspect the software's source code and propose changes to it. These changes are sent back to the original developer(s) in the form of *patches*[4]. The project leader(s) review this change proposals and apply (or not) them to their own version, so that when a new version of the software is released, end users will have access to the new functionalities of bug fixes proposed by these contributors.

In the course of time, the most frequent contributors gain the trust from the initial developer(s), and can receive direct write access to the official source code of the project. That way they will be able to make changes directly instead of having to pass through the review of the original developers. This process of developers joining free software projects may range from very informal to very formal, depending on the project. Small projects normally have informal procedures for that, e.g. one existing developer just offers an account in the version control system to the new contributor. Larger projects, on the other hand, may have more "bureaucratic" processes for accepting a new developer with privileged access to the project's resources, such as filling forms with an application, signing terms of copyright transference and other procedures.

The development is normally driven with the help of a version control system (VCS), where the latest version of the source code is stored in a source code repository. The VCS records every change ever made to the source code, as well as the author and date of the changes. While the repository is often publicly available for read access, write access is restricted to a limited group of developers. Other developers will need their changes (*patches*) to be reviewed by a developer with the needed privileges in order to get their contributions into the project's official repository. This process is illustrated by figure 1.

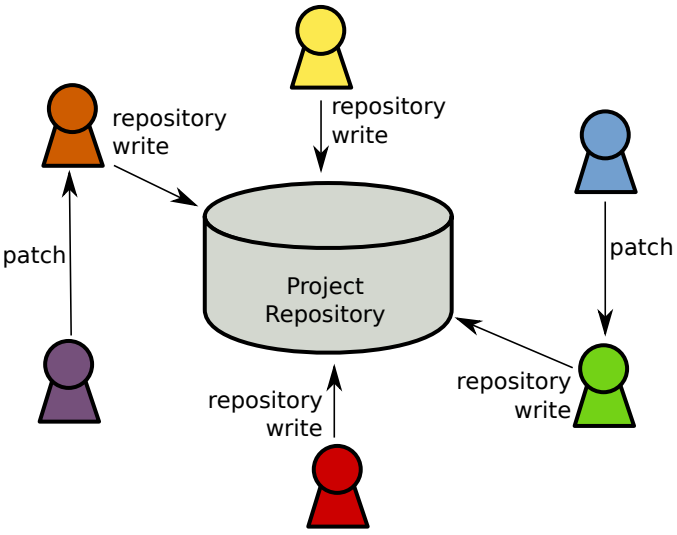


Fig. 1. Free software development by means of a VCS repository.

There are variants of this process. For example, recently the usage of distributed version control became quite popular. With distributed VCS, there is no need for a central repository. Each developer has its own repository, which may be published for other to access. But the concept of an official repository, blessed by the project leaders, still exist.

The following characteristics make free software projects different enough from "conventional" software projects, and

[3]As free software advocates are used to remind, the "free" here is "free" as in "freedom", not as in "free lunch" ("gratis").

[4]A patch is a file that describes the changes to one or more files, normally of textual content, by describing which lines to remove and which lines to add. After receiving a patch the developer can reproduce the changes proposed by the sender in his/her own copy of the source code.

also makes them an interesting object of study:

- **Source code availability.** Source code of free software projects is always available on the internet. Although most of the projects have a publicly-accessible version control repository, there are projects that do not have one (e.g. the Lua Programming language [5]).
- **User/developer symbiosis.** In most free software projects the developers are also users of the software, and they also provide requirements. Maybe because of that, several free software projects do not have explicit requirement documents, and the development flows on a rhythm in which the developers are able to satisfy their own needs.
- **Non-contractual work.** A large amount of work in free software projects is done in a non-contractual fashion. This does not imply that the developers are necessarily volunteers, but only that there is no central management with control over all of the developers' activities.
- **Work is self-assigned.** Since most free software projects don't have a central central management with control over the contributors' activities, the work of these contributors is normally self-assigned: volunteer developers work on the parts of the project that most appeal to them.
- **Geographical Distribution.** In most Free Software projects the developers are spread among several different locations in the world. In the projects with high geographical dispersion, almost all communication is performed through electronic means.

Although the Software Engineering literature tends to portrait Free Software as a homogeneous phenomenon[6], most of these characteristics do not apply to all free software projects, and some of them may be manifest in different ways across projects.

B. Structural complexity

Structural complexity is an architectural concern: it involves both the internal organization of software modules, as well as how these modules relate to each other [7], [8]. structural complexity influences the developer's time: a more complex software is expected to require more effort from developers to be comprehended in maintenance activities [9].

Several aspects of Software Design can be considered when evaluating structural complexity. We can consider, among others, coupling [10], [11], cohesion [10], and inheritance [10], [11].

While inheritance is specific to the object-oriented paradigm, coupling and cohesion are more generally applicable. Every programming paradigm has a notion of *module*, whether it is called "module", "class", "aspect", "abstract data type", "source file", etc. Having modules, one can always analyse a program and identify which other modules a module refers to and thus have a notion of coupling, and also verify how the subparts of a given module interact with each other to evaluate the cohesion of such a module.

In an experimental setting with professional software developers, Darcy *et al* found that more complex software requires more effort for maintenance activities [9]. Moreover, they verified that neither coupling nor lack of cohesion by themselves could explain the decrease in comprehension performance of the developers; only when considered together (by multiplying the two) they presented an association with higher maintenance effort. The authors claim that when considering structural complexity, one must consider coupling and cohesion together.

In this work, we follow Darcy *et al* and consider both coupling and cohesion in our definition of structural complexity. We provide, however, a formal definition of this metric, as follows.

The structural complexity of a module is given by multiplying coupling (Chidamber and Kemerer's CBO [10]) and lack of cohesion (Hitz and Montazeri's LCOM4 [12]) metrics. A project-wide value for structural complexity can be obtained by taking the average structural complexity among all modules. Given $M(p)$, the set of modules in a project p, we have:

$$SC(p) = \frac{\sum_{m \in M(p)} CBO(m) \times LCOM4(m)}{|M(p)|}$$

C. Structural Complexity in free software projects

Midha [13] studied projects from sourceforge.net and verified that increases in complexity leads to increase in the number of bugs in the source code, decrease in contributions from new developers and increase in the time taken to fix bugs. Although using a different concept of structural complexity by considering MacCabe's Cyclomatic Complexity and Halstead's Effort[5], these results demonstrate that complexity has nasty effects on Free Software projects. We speculate that an increase in structural complexity as defined in this study here has similar effects in free software projects, as it did in the controlled experiment by Darcy *et al* [9].

Stewart *et al* studied 59 projects written in Java that were available on Sourceforge [14], using the product of coupling and lack of cohesion as their structural complexity measure. They verified 4 different patterns of structural complexity evolution, of which 2 presented growing trend in the end of the period. The other 2 presented stabilization in the end of the period: none of the identified patterns featured a complexity reduction trend. A previous study of ours also indicated a growing trend in structural complexity on another (but smaller) project written in C [15].

Increasing complexity trends are not an exclusive feature of free software projects, and are not recent news: the seminal work of Lehman on software evolution already identified it, and that led to the formulation of the second law of software evolution the Law of Increasing Complexity [16]. That law, formulated in the context of studies on proprietary software systems, states that as systems evolve, their complexity increases unless work is done to maintain or reduce it.

[5]These two measures represent respectively the internal complexity of subroutines and the overall vocabulary size of the code base. They reflect a different aspect of structural complexity, at the subroutine level. Here we are looking at structural complexity at the design/architecture level, considering the relationship between modules and between the sub-parts of each module.

For now, we know that i) software complexity is associated with undesirable effects (more maintenance effort, more bugs, less attraction of new developers) and ii) structural complexity tends to not decrease, and in a reasonably large amount of cases, it tends to grow. That leads us to the following question: why does structural complexity increase in the context of Free Software projects?

In this paper we investigate whether developer attributes can explain the variation in structural complexity in free software projects, specifically whether core and periphery developers contribute differently to such variation. The concept of core and periphery an important aspect in the study of the free software development process, and is described in the next section.

D. Core and Periphery in free software Projects

Normally, a Free Software project is started by a single developer, or by a group of developers, in need of addressing a particular need. After there is a usable version, it is released to the public under a Free Software license which allows anyone to use, change and distribute a copy of that software. As new users get interested in the project, some of them may start to contribute to it in several possible ways: with code for new features or bug fixes, with translations into their native languages, with documentation, or with other types of contribution. At some point, then, the project has a vivid and active community: a group of people that gravitate around a project, with varied levels of involvement and contribution.

The "onion model" [2], [3] became a widely accepted representation of what happens in a Free Software project, by indicating the existence of concentric levels of contribution: a small group of core developers do the largest part of the work; a larger group makes direct, but less frequent contributions in the form of bug fixes, patches, documentation, etc; an even larger group reports problems as they use the software, and the largest group is formed by the silent users who only use the software but never provide any type of feedback.

The processes by which participants migrate from one group to another are very different from one community to the other: communities may adopt more formal and explicit procedures for that, or use a more relaxed approach and let things flow "naturally". But in general the achievement of central roles (and thus more responsibility, respect and decision power) are merit-based: a developer becomes a leader by means of continuous valuable contributions to the community [17].

Since most of the work is done by a core team, it is important for projects to keep a healthy and active core team. Some projects are able to keep its core team with few or no changes across its entire history, while others experience a succession of different generations of core developers [18], [19].

The relationship between core contributors and peripheral (non-core) members of a community are not always smooth: sometimes the core tends to work on their own demands and to give little attention or even to ignore completely the demands of the periphery [20], [21]. From an individual point of view, core and periphery members also exhibit different behaviour while debating subjects related to the project [22] or in the bug reporting activity [21].

III. Research hypotheses

As discussed earlier, structural complexity raises the maintenance cost of a software project, because the code becomes harder to understand, and in consequence harder to modify. In free software projects, such an increase in effort may represent an extra difficulty for gathering new contributors. Failing to attract contributors represents a threat to the project sustainability [23], specially those which are not mainly funded by a single organization and rely on the contributions of volunteers.

There are differences between core and peripheral contributors with respect to the volume of work done and behaviour in communication inside the project. This begs the question as if the quality of their contributions could be also different. We want to evaluate the amount of complexity that they introduce into the code. Since core developers have deeper knowledge of the software architecture, it is expected that their changes to the source code do not add as much structural complexity as the changes made by peripheral developers do. Thus the first hypothesis we want to test in this paper is the following:

> H_1: *changes made by core developers introduce less structural complexity than those made by periphery developers.*

Several projects also undertake development effort in order to refactor the code and thus reduce complexity [24]. As formulated by Lehman [16], this is needed in order to keep the project under a sustainable level of complexity. While both core and periphery developers can participate in such a task, core developers are probably more successful at it than periphery developers, and we want to verify that empirically. Our second hypothesis is then related to the reduction of structural complexity in the source code:

> H_2: *among the changes that reduce structural complexity, the ones made by core developers achieve greater structural complexity reduction than those made by periphery developers.*

IV. Research Design

In order to test our hypotheses, we designed and executed an empirical study, which is described in this section. Care was taken in order to provide all information needed by the reader to assess the quality of the study and the applicability of its results [25].

The research method used was an *observational study*, in which a phenomenon is observed in its natural setting (as opposed to a controlled lab environment, used in a true experiment).

Our data collection approach was to mine the version control systems of a selected group of free software projects from the web server application domain, and to collect data from each change to the project source code. For each change, we registered the date of the change, the variation of structural

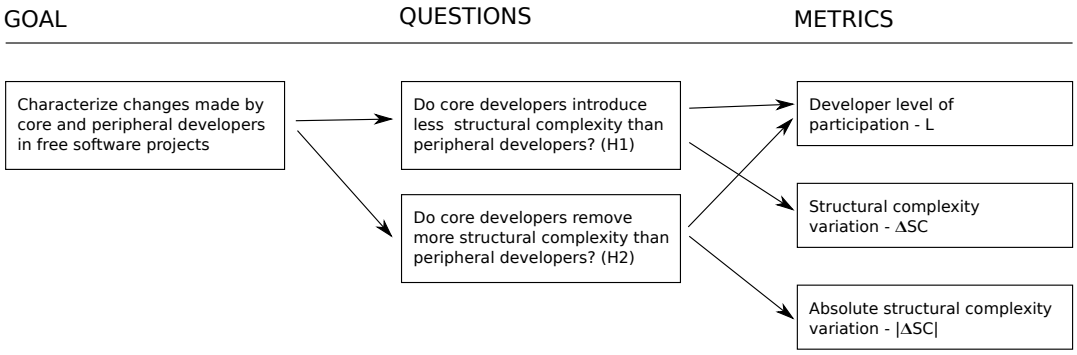


Fig. 2. GQM diagram of the study

complexity accomplished by the change, and whether the change was made by a core or a peripheral developer.

The study definition, using a GQM template [26] is as follows: In this study we **analyse** changes made to the source code of free software projects **for the purpose of** of characterization **with respect to** structural complexity added or removed and level of developer participation, **from the perspective** of the researcher **in the context of** the web server application domain.

Figure 2 presents a GQM diagram of the study. In such diagram, the first column, "Goals", identifies the problems that a given study is trying to solve. "Questions" identifies the questions that, when answered, will provide a solution to the problems, and "Metrics" identifies which metrics (or "variables") need to be measured so that we achieve answers to the related questions. Arrows connect goals and their associated questions, as well as questions and their associated metrics. In the case of the current study, there is a single goal with two questions. Our questions are directly mapped to the research hypotheses presented in section III.

The metrics associated to each research question in figure 2 are presented in section IV-A. Our data collection approach and the obtained sample are described in section IV-B.

A. Variables and operational definitions

The following variables are considered in the study:

- Independent variable
  - L, the level of involvement of the author of the given change in the project at that point in time. This is determined by splitting the entire studied period in 20 periods of equal duration, and for each period identifying the 20% top committers as the core team (cf. [18], [19]). The reader should note that by according to this definition, the same developer can be considered as a core developer in some periods and a peripheral developer in other periods. This is coherent with reality: since developers often do not have any formal responsibilities with with the projects, they may reduce or increase their activity in the project in specific periods, and thus can shift from the core to the periphery and vice-versa.
- Dependent variables
  - SC, the overall structural complexity of the project after each change, as described in section II-B.
  - ΔSC, the increment in structural complexity in each change. For each change, this value is obtained by subtracting its SC value from the previous change's SC value. This variable represents how much the structural complexity changed after a given change was applied to the project source code.
  - $|\Delta SC|$, the absolute change in structural complexity (i.e. the absolute value of ΔSC).

B. Sample and data collection

We started by identifying all web server packages in the Debian GNU/Linux archive. This decision was made in order to select projects that were being actually used: if a web server software is properly packaged and maintained in Debian, it means that there is interest in it to the point that someone volunteers to maintain an automatic installation package so that users can install it without the need for manual configuration[6].

This provided us 21 web server projects as a starting point. Since we needed to harvest the version control history of the projects in order to characterize the changes made to the source code, we needed that the projects have an accessible version control repository. Due to a temporary limitation in our tooling for source code analysis, we had to stick to projects written in the C language. Since C is a language commonly used for infrastructure software, most of the web servers identified are written with it. After applying both restrictions, we could identify 7 projects to work with. They are listed in table I.

The source code repository of each project was imported locally in a `git`[7] repository in order to facilitate fast and off-line history browsing. We used a set of scripts developed by us to mine this repository as follows:

- Determine the list of relevant commits, by identifying the commits that changed source code files. This way we

[6]Although the first author of this paper is a Debian maintainer, he is not maintainer of any of the web server packages evaluated. Also, none of the authors is affiliated with any of the studied projects.

[7]http://git-scm.org/. `git` has support for importing repositories from CVS and Subversion.

TABLE I
PROJECTS SELECTED FOR ANALYSIS

Project	Start	End	Commits	Developers
aolserver	2000/05	2009/05	1125	22
apache	1999/06	2009/11	9663	72
cherokee	2005/03	2009/10	1545	8
fnord	2001/08	2007/11	99	2
lighttpd	2005/02	2009/10	775	6
monkeyd	2008/01	2009/06	207	4
weborf	2008/07	2009/10	139	3

avoided analysing subsequent states of the source code that were no different from each other.

- Checkout each relevant version and run a static source code analysis tool in order to calculate the source code metrics used, namely CBO [10] and LCOM [10] (we used the improved version from Hitz and Montazeri [12], though).
- Accumulate the results for each change in a single data file per project.

After processing all the projects, we loaded their raw data in a relational database in order to facilitate the evaluation of L, SC, ΔSC and $|\Delta SC|$. During this process we excluded 7 commits that had no previous commit to compare to (i.e. the very first commit of each project).

Table I lists some aggregated information about the data sample. The "Start" and "End" columns show year and month of the first and last changes considered, respectively, according to the project's version control system. The "Commits" column displays the number of changes (commits) considered for each project, and the "Developers" column counts how many different developers were responsible for these considered changes.

The following tools were used to mine the repositories:

- The static source code analysis was performed with `analizo`[8], a multi-language source code analysis tool we have been working on.
- The history analysis of the version control repositories was made by the `analizo-metrics-history` script from the `analizo-utils` package[9]
- The analysis specific to this study was done by some ad-hoc Ruby code.

The complete package for this study, with data, local scripts etc is available on the web at the following address: http://www.dcc.ufba.br/~terceiro/papers/cpsc.tar.gz.

V. DATA ANALYSIS AND RESULTS

The full dataset contained 13553 changes, of which 9944 (73.36%) were made by core developers, and 3609 (26.63%) by peripheral developers. In order to test our hypotheses, we excluded from the analysis the changes that did not make any changes to the structural complexity metric SC (i.e. $\Delta SC = 0$). The remaining changes are similarly distributed between the two groups: of 2513 changes, 1994 (79.35%) were made by developers considered as core developers, while the other 519 (20.65%) were made by peripheral developers.

To test H_1, we need to compare ΔSC for the subset of changes made by core developers and the ones made by periphery developers. H_1 can then be formalized as follows:

$$H_1 : \mu_{\Delta SC_{core}} < \mu_{\Delta SC_{periphery}}$$

We used a t-test to verify the hypothesis, and were able to reject the null hypothesis that there is no difference between the means and accept the alternative hypothesis H_1, with $p < 0.05$ ($p = 0.01515265$). This demonstrates that our hypothesis that *changes made by core developers introduce less structural complexity than those made by periphery developers* is supported by the data.

To test H_2, we must consider only the cases in which there is a decrease in structural complexity. To do that, we filtered the dataset again and kept only the changes in which $\Delta SC < 0$. In this filtered dataset we have 1165 changes, of which 939 (80.60%) were made by core developers and 226 (19.40%) by periphery developers. We want then to verify whether the amount of structural complexity removed by core developers is greater than the amount removed by periphery developers, what can be formalized as follows:

$$H_2 : \mu_{|\Delta SC|_{core}} > \mu_{|\Delta SC|_{periphery}}$$

The t-test for H_2 allowed us to reject the null hypothesis of the two variables being equal, and accept the alternative H_2 with $p < 0.05$ ($p = 0.01091324$). The data support our second hypothesis as well: *among the changes that reduce structural complexity, the ones made by core developers achieve greater structural complexity reduction than those made by periphery developers*.

Table II presents descriptive statistics of the dataset used in these results.

The data analysis was performed with the R system [27] and RKWard, a frontend to R[10].

VI. THREATS TO VALIDITY

While carefully designed, this study has some limitations that represent threats to its generalisability.

The careful reader will notice that although all variables tested in section V are not normally distributed (see table II), we still used the t-test for comparing them. The t-test usually requires that the variables have a normal distribution, but as noted by Wohlin *et al* [28], it is robust enough to support some deviation from these preconditions. In particular, since our sample is large enough, we can use the t-test without problems. To be sure, we also performed a Wilcoxon/Mann-Whitney test (a non-parametric test indicated as replacement for the t-test when the samples are not normally distributed) that provided similar results.

With respect to the choice of data sample, by considering only one application domain and only projects written in C we do not address the wide diversity of free software projects.

[8] http://github.com/terceiro/analizo
[9] http://github.com/terceiro/analizo-utils
[10] http://rkward.sourceforge.net/

TABLE II
DESCRIPTIVE STATISTICS OF THE VARIABLES TESTED

Variable	Mean	Std. Dev.	Min.	Max.	n
ΔSC_{core}	0.001660474	0.3334254	-5.967357	5.355073	1994
$\Delta SC_{periphery}$	0.03426117	0.2970714	-2.023467	3.021991	519
$\lvert\Delta SC_{core}\rvert$	0.1291047	0.3092991	5.939609e-05	5.967357	939
$\lvert\Delta SC_{periphery}\rvert$	0.09200304	0.1891808	0.002171662	2.023467	226

In order to have results that can be properly generalized, we need to study a more diverse population. It may be the case that the communities working on different application domains or different programming languages have different design and programming practices, and that could affect the obtained results.

From a construction validity point of view, by having a single independent variable (the level of involvement of the developer) we are not considering other factors that influence the addition of structural complexity to the source code on those projects.

A limitation caused by our choice to use only the version control metadata directly provided by the repositories in structured form is that we may be masking the reality by considering the commit author as being the same developer who actually developed the change. In several projects there is a limited set of developers, known as *committers*, who have write access to the repositories. This way, contributions from developers who are not committers need to be reviewed and applied by a committer, and the version control repository stores the committer name as the author of the change. In those cases the committer usually gives credit to the original author of the change in free-form text inside the commit log message, but we did not consider this in the data extraction. On the other hand, in such cases the committer explicitly decided to approve a change proposed by another developer and to apply that change to the source code; one can also argue that by doing that the committer took part in a design decision that affects the structural complexity of the project, even not having written the code herself.

We did not analyse the nature of the changes that reduce structural complexity, so we cannot claim that they are similar. Complexity-reducing changes may actually represent corrective, adaptive, perfective and preventive maintenance activities. Those changes can also be localized in few software modules, or systemic changes that touch a large number of modules. They can be defect corrections, or implementation of new features. For example, a change that adds a new module will usually make the structural complexity metric drop: since we used the average structural complexity per module, adding a new module that is not as complex as the current average will make the average value fall down. In such case, however, adding a new module that is not as complex as the average may be considered a good thing, since we are adding new functionality to the software without making the part that implements that functionality as complex as the rest of the system. The bottom line is that the differences in structural complexity reduction may actually be caused by the type of maintenance being performed while core and periphery developers happen to perform different types of maintenance.

We also excluded from the analysis changes that do not change the structural complexity metric (i.e. those in which $\Delta SC = 0$). These changes may also reveal interesting design activities, such as removing a dependency from module A to module B and making A depend on C instead. While this will not change A's coupling, being able to analyse changes like this one may provide relevant information about the project's design activity. By not considering this type of change we are probably ignoring events that may influence the others that actually change the structural complexity metric.

VII. RELATED WORK

Capra *et al* [24] claim that free software projects with a more open governance structure exhibit a better design quality. From one side, a higher design quality enables a more open governance: less coupled modules allow different developers to work on their own parts of the project without explicit coordination activities. Having a more open governance gives developers more freedom to enhance the design quality instead of having to keep up with deadlines or another types of pressures from higher management or customers.

Bargallo *et al* found that project popularity[11] may be associated with a lower design quality. They argue that as a project becomes more popular, their lead developers may redirect their efforts from programming to other activities such as answering users in forums or mailing lists, reviewing contributions etc. Such projects, having their main (and more experienced) developers change their attention to non-programming activities, may suffer from a decrease in design quality[29].

These works try to find factors that influence what they call "good design" in free software projects, in terms of different sets of object-oriented metrics from the suites by Chidamber amd Kemerer [10] and Brito e Abreu [11]. Our work differs from the above by i) considering the structural complexity construct, which is based on concepts applicable to both OO and non-OO software (coupling and cohesion) and ii) studying developers' attributes as factors (as opposed to looking at organizational and project-wide aspects).

VIII. CONCLUSIONS

In this paper we have investigated the relationship between the different levels of developer involvement in free software projects and the amount of structural complexity added by changes recorded in version control systems. We thus provide

[11]measured as a function of the number of downloads, web traffic and development activity

relevant results on the technical side of the core/periphery dichotomy: core and periphery do not only behave differently [21], [22] and contribute different amounts of effort [3], [2], but they also provide code of different complexity.

The data obtained supports both our hypothesis: the core developers are able to make changes to the source code without introducing as much structural complexity as the peripheral developers; and they also remove more structural complexity than the other developers. Finding empirical evidence for such hypotheses highlights the importance of a healthy core team for a free software project: in a certain sense, the core team is responsible for keeping the project's conceptual integrity, as suggested by Brooks [30].

It is important to note, however, that these results cannot be taken as an incentive to not accept contributions from non-core developers. Not all projects are able to keep the same core team during its entire lifespan [18], so receiving new contributors is fundamentally important for the project's sustainability. In several projects, non-core developers are responsible for a healthy ecosystem of extensions, plugins and other types of add-ons that can be used together with the main product. The source code for these add-ons sometimes does not even reside in the main project repository, and sometimes the changes they propose to the core product are needed to enable an entire new line of possibilities for extension developers.

Attracting as much contributors as possible is important to have a thriving free software project. Project leaders will most likely not want to send their contributors away even if their contributions are not perfect. The results presented here can be seen as a opportunity for project leaders to qualify their contributors: since non-core developers tend to produce more complex code, it is perhaps a good idea to explicit review their code. If code review practices are adopted, core developers can work together with non-core developers looking for less complex solutions. This code review activity can even leverage documented guidelines of good design practices for the project contributors.

The work reported in this paper is part of a larger research project, in which we investigate how developer characteristics influence the variation of structural complexity in free software projects. Considering the developer's level of participation is just the first step towards a more comprehensive understanding of the phenomenon. That said, in the following paragraphs we outline directions that may be taken for future work.

We plan to test other developer attributes in order to build a more comprehensive model of how the developers influence the variation of structural complexity in these projects. Such attributes can be divided in two groups: i) attributes related to the developers themselves, such as skill level, programming ability, experience in general; ii) attributes related to the participation of the developer in the project, such as level of participation (already explored in this paper), experience in the project, experience with the modules being changed, and whether the developer is a specialist or not in the area he/she is working on (with regard to its own past activity in the project).

Another alternative is testing the same hypotheses tested in this study on each project individually. This may provide us with stronger results for specific projects, and for other projects their data may not support the hypotheses at all. If that is the case, doing a richer characterization of the projects would support us on identifying other factors that enable or prevent the results achieved in this study. We can also investigate other factors that may impact the introduction of structural complexity in the source code by constructing a richer characterization of the changes themselves.

Extending our dataset to include projects from other application domains and written in different programming languages will also help us generalize the results to a wider range of free software projects.

The information stored in the version control repositories can also be explored better. There is a lot of possibilities for analysing the combination of the metadata from the version control system *and* information about the actual changes in the structure of the source code (as opposed to mere information on lines added/removed).

It is also unclear, and worth investigating, how the developer's design ability advances as the developer advances in the community. It would be interesting to learn how individual developers evolve in terms of complexity added to the source code as a developer moves from the periphery to the core, or the other way around.

ACKNOWLEDGMENTS

The authors are thankful for the insightful contributions from Dr. Daniela Cruzes from the Norwegian University of Science and Technology[12] and from their colleagues at the Software Engineering Lab (LES) at Federal University of Bahia[13], in special Dr. Manoel Mendonça. Antonio Terceiro is supported by the Brazilian National Council of Research and Development (CNPq)[14]. Luiz Romário is supported by *Fundação de Amparo à Pesquisa do Estado da Bahia (FAPESB)*[15]. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES)[16], funded by CNPq, grant 573964/2008-4.

We are also grateful to the anonymous reviewers for their comments and contributions. All of their remarks that could not be addressed in this paper will certainly be taken into account in future studies.

REFERENCES

[1] D. L. Parnas, "Software aging," in *ICSE '94: Proceedings of the 16th international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 279–287.

[2] K. Crowston and J. Howison, "The Social Structure of Free and Open Source Software Development," *First Monday*, vol. 10, no. 2, 2005.

[3] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 3, pp. 309–346, 2002.

[12] http://www.idi.ntnu.no/
[13] http://les.dcc.ufba.br/
[14] http://www.cnpq.br/
[15] http://www.fapesb.ba.gov.br/
[16] http://www.ines.org.br/

[4] Free Software Foundation, "The Free Software Definition," 2009, Available at http://www.gnu.org/philosophy/free-sw.html, last accessed on January 1st, 2010.

[5] The Lua Programming Language, "Frequently Asked Questions," 2009, Available at http://www.lua.org/faq.html#1.8, last access on January 5th, 2010.

[6] T. Østerlie and L. Jaccheri, "A Critical Review of Software Engineering Research on Open Source Software Development," in *Proceedings of the 2nd AIS SIGSAND European Symposium on Systems Analysis and Design*, W. Stanislaw, Ed. Gdansk University Press, 2007, pp. 12–20.

[7] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architecture : Views and Beyond*, ser. The SEI series in software engineering. Boston: Addison-Wesley, 2002.

[8] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[9] D. P. Darcy, C. F. Kemerer, S. A. Slaughter, and J. E. Tomayko, "The Structural Complexity of Software: An Experimental Test," *IEEE Transactions on Software Engineering*, vol. 31, no. 11, pp. 982–995, Nov. 2005.

[10] S. Chidamber and C. Kemerer, "A metrics Suite for Object Oriented Design," *IEEE Trans. Sftware Eng.*, vol. 20, no. 8, pp. 476–493, 1994.

[11] F. B. e Abreu, "The MOOD Metrics Set," in *Proc. ECOOP Workshop Metrics*, 1995.

[12] M. Hitz and B. Montazeri, "Measuring coupling and cohesion in object-oriented systems," in *Proceedings of the International. Symposium on Applied Corporate Computing*, 1995.

[13] V. Midha, "Does Complexity Matter? The Impact of Change in Structural Complexity on Software Maintenance and New Developers' Contributions in Open Source Software," in *ICIS 2008 Proceedings*, 2008.

[14] K. J. Stewart, D. P. Darcy, and S. L. Daniel, "Opportunities and Challenges Applying Functional Data Analysis to the Study of Open Source Software Evolution," *Statistical Science*, vol. 21, p. 167, 2006.

[15] A. Terceiro and C. Chavez, "Structural Complexity Evolution in Free Software Projects: A Case Study," in *QACOS-OSSPL 2009: Proceedings of the Joint Workshop on Quality and Architectural Concerns in Open Source Software (QACOS) and Open Source Software and Product Lines (OSSPL)*, M. Ali Babar, B. Lundell, and F. van der Linden, Eds., 2009.

[16] M. M. Lehman, J. F. Ramil, P. D. Wernick, and D. E. Perry, "Metrics and Laws of Software Evolution-The Nineties View," in *Proceedings of the 4th International Symposium on Software Metrics*, 1997. [Online]. Available: {citeseer.ist.psu.edu/lehman97metrics.html}

[17] C. Jensen and W. Scacchi, "Role Migration and Advancement Processes in OSSD Projects: A Comparative Case Study," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 364–374.

[18] G. Robles and J. Gonzalez-Barahona, "Contributor Turnover in Libre Software Projects," *Open Source Systems*, pp. 273–286, 2006. [Online]. Available: {http://dx.doi.org/10.1007/0-387-34226-5_28}

[19] G. Robles, J. M. Gonzalez-Barahona, and I. Herraiz, "Evolution of the core team of developers in libre software projects," in *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, May 2009, pp. 167–170.

[20] J.-M. Dalle, M. d. Besten, and H. Masmoudi, "Channeling Firefox Developers: Mom and Dad Aren't Happy," in *Open Source Development, Communities and Quality, IFIP 20th World Computer Congress, Working Group 2.3 on Open Source Software, OSS 2008, September 7-10, 2008, Milano, Italy*, B. Russo, E. Damiani, S. A. Hissam, B. Lundell, and G. Succi, Eds., vol. 275. Springer, 2008, pp. 265–271.

[21] H. Masmoudi, M. d. Besten, C. d. Loupy, and J.-M. Dalle, ""Peeling the Onion": The Words and Actions that Distinguish Core from Periphery in Bug Reports and How Core and Periphery Interact Together." in *OSS: Diverse Communities Interacting, 5th IFIP WG 2.13 International Conference on Open Source Systems, OSS 2009, Skövde, Sweden, June 3-6, 2009. Proceedings*, C. Boldyreff, K. Crowston, B. Lundell, and A. I. Wasserman, Eds., vol. 299. Springer, 2009, pp. 284–297.

[22] M. J. Scialdone, N. Li, R. Heckman, and K. Crowston, "Group Maintenance Behaviors of Core and Peripherial Members of Free/Libre Open Source Software Teams," in *OSS: Diverse Communities Interacting, 5th IFIP WG 2.13 International Conference on Open Source Systems, OSS 2009, Skövde, Sweden, June 3-6, 2009. Proceedings*, C. Boldyreff, K. Crowston, B. Lundell, and A. I. Wasserman, Eds., vol. 299. Springer, 2009, pp. 298–309.

[23] A. Capiluppi and M. Michlmayr, "From the Cathedral to the Bazaar: An Empirical Study of the Lifecycle of Volunteer Community Projects," in *Open Source Development, Adoption and Innovation*, J. Feller, B. Fitzgerald, W. Scacchi, and A. Silitti, Eds. Springer, 2007, pp. 31–44.

[24] E. Capra, C. Francalanci, and F. Merlo, "An Empirical Study on the Relationship Between Software Design Quality, Development Effort and Governance in Open Source Projects," *IEEE Transactions on Software Engineering*, vol. 34, no. 6, pp. 765–782, Nov.-Dec. 2008.

[25] K.-J. Stol and M. A. Babar, "Reporting Empirical Research in Open Source Software: The State of Practice," in *OSS: Diverse Communities Interacting, 5th IFIP WG 2.13 International Conference on Open Source Systems, OSS 2009, Skövde, Sweden, June 3-6, 2009. Proceedings*, C. Boldyreff, K. Crowston, B. Lundell, and A. I. Wasserman, Eds., vol. 299. Springer, 2009, pp. 156–169.

[26] V. Basili, G. Caldiera, and D. H. Rombach, "The Goal Question Metric Approach," in *Encyclopedia of Software Engineering*, J. Marciniak, Ed. Wiley, 1994.

[27] R Development Core Team, *R: A Language and Environment for Statistical Computing*, Vienna, Austria, 2009, ISBN 3-900051-07-0. [Online]. Available: {http://www.R-project.org}

[28] C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: an Introduction*. Kluver Academic Publishers, 2000.

[29] D. Barbagallo, C. Francalenei, and F. Merlo, "The Impact of Social Networking on Software Design Quality and Development Effort in Open Source Projects," in *ICIS 2008 Proceedings*, 2008. [Online]. Available: {http://aisel.aisnet.org/icis2008/201}

[30] F. P. Brooks.Jr, *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley , April 1995, ch. "Aristocracy, Democracy, and System Design".

Apêndice C

Este apêndice contém uma reprodução de artigo publicado com mesmo título nos anais da 16th European Conference on Software Maintenance and Reengineering*, em 2012.*

UNDERSTANDING STRUCTURAL COMPLEXITY EVOLUTION: A QUANTITATIVE ANALYSIS

Understanding Structural Complexity Evolution: a Quantitative Analysis

Antonio Terceiro, Manoel Mendonça, Christina Chavez
Software Engineering Lab (LES)
Department of Computer Science (DCC)
Federal University of Bahia (UFBA)
{terceiro,mgmendonca,flach}@dcc.ufba.br
Daniela S. Cruzes
Department of Computer and Information Science (IDI)
Norwegian University of Science and Technology (NTNU)
dcruzes@idi.ntnu.no

Abstract*—*Background: **An increase in structural complexity makes the source code of software projects more difficult to understand, and consequently more difficult and expensive to maintain and evolve. Knowing the factors that influence structural complexity helps developers to avoid the effects of higher levels of structural complexity on the maintainability of their projects.**
Aims: **This paper investigates factors that might influence the evolution of structural complexity.**
Method: **We analyzed the source code repositories of 5 free/open source software projects, with commits as experimental units. For each commit we measured the structural complexity variation it caused, the experience of the developer who made the commit, the size variation caused by the commit, and the change diffusion of the commit. Commits that increased structural complexity were analyzed separately from commits that decreased structural complexity, since they represent activities of distinct natures.**
Results: **Change diffusion was the most influential among the factors studied, followed by size variation and developer experience; system growth was not necessarily associated with complexity increase; all the factors we studied influenced at least two projects; different projects were affected by different factors; and the factors that influenced the increase in structural complexity were usually not the same that influenced the decrease.**
Conclusions: **All the factors explored in this study should be taken into consideration when analysing structural complexity evolution. However, they do not fully explain the structural complexity evolution in the studied projects: this suggests that qualitative studies are needed in order to better understand structural complexity evolution and identify other factors that must be included in future quantitative analysis.**

I. Introduction

Software maintenance and evolution are labor-intensive activities that consume a large part of the overall software lifecycle costs [1]. In this context, the assessment, comprehension and control of factors that impact maintenance and evolution activities are of paramount importance.

One such a factor, the *structural complexity* of a software system [2], [3], [4], [5], [6], [7], may exert strong influence on the maintenance effort: the higher is the structural complexity of the source code, the greater is the effort to comprehend, change and evolve it [6], [8]. In the specific case of free software[1], projects with higher structural complexity are less likely to attract developers [8], [9]. This situation is specially harmful for projects that do not have corporate sponsors and depend solely on independent and volunteer developers to evolve.

There are empirical studies on the measurement of structural complexity and its evolution over time [6], [10], [11], but they fail to assess and provide empirical evidence about which factors influence the evolution of structural complexity itself. The structural complexity of a project may increase due to different factors, such as the maturity level of the project, growth of its codebase, the type of maintenance (perfective, corrective, evolutionary) the project has been subject to, development practices, past design decisions, and characteristics of the developers working on the project. The knowledge about the factors that impact structural complexity could be useful for developers and project managers to control the growth of structural complexity, or to drive its decrease by means of anti-regressive activities. This would contribute to keep an acceptable level of understandability and changeability and, therefore, to reduce maintainance effort and costs.

Our work aims to reduce this gap by studying different factors that may influence the variation in structural complexity, for instance, developers' experience within a project, size variation and change diffusion. The choice of *developers' experience within the project* as a factor reflects the importance of human factors on software development projects. We expect that developers' experience influences software design activities, and in special, the evolution of structural complexity. Experience has been used previously to identify experts [12] and to predict defects [13], [14]. *System size* is an important factor in software evolution studies, and this work is not an exception. Besides, system size if often associated with software complexity [15], [16], [17]. Change diffusion – the number of different elements (subsystems, modules, files etc) involved in a change – represents how wide a change is with respect to a given codebase. Change diffusion has

[1] a.k.a. "open source software".

been characterized as a good predictor for defects [13], where the authors argued that "large diffusion indicates that the modularity of the code is not compatible with the change, because several modules have to be touched to implement the change. [...] The diffusion of a change reflects the complexity of the implementation [...]". Based on their ideas, we expect that change diffusion may have some influence over structural complexity evolution as well.

This paper reports the results of an empirical study performed in order to assess whether and how developers' experience within a project, size variation and change diffusion influence the evolution of structural complexity. These three factors were studied in the context of historical data from five well-established free software development projects written in different programming languages. Our study can be regarded as a preliminary step towards the systematic identification of factors that can be used in the selection of treatments for taming structural complexity during maintenance activities.

The rest of the paper is organized as follows. Section II introduces basic concepts related to structural complexity, developer experience within the project, size variation and change diffusion. Section III presents the design of our study. Results are presented in Section IV, and discussed in Section V. Threats to validity are presented in Section VI. Section VII presents our closing remarks.

II. Background

A. Structural Complexity

The concept of software complexity has been extensively studied in Software Engineering [2], [3], [4], [18], [19], [5], [6], [11], [10], [7], and yet there is no widely-accepted approach for measuring it. Since the 70's, there were several proposals for measuring complexity at the subroutine level [2], [3], [4]. McCabe's cyclomatic complexity [2] eventually became widely used and it's still in use in more recent work [20], [8], [21].

Software complexity can also be studied at a higher level of abstraction. Researchers have proposed structured approaches for software architecture planning and documentation [18], [19], Design Structure Matrices [5] to represent complex software systems, highlighting dependencies and modules, measures for structural complexity based on inter-module coupling and intra-module cohesion [6], [11], [10], and the use of recursive aggregation to calculate structural complexity of higher levels of abstraction in terms of their own organization and the structural complexity of lower levels of abstraction [7].

Darcy *et al.* [6] propose that structural complexity should be measured by combining coupling and cohesion metrics together. Coupling and cohesion are complementary concepts: while coupling addresses inter-module relationships, cohesion is concerned with the relationship between the intra-module elements and their organization. The software design practitioners' literature advocates that modules with low coupling and high cohesion are easier to understand and modify [22], [23].

The combination of coupling and cohesion into a structural complexity metric captures both the complexity created by excessive inter-module dependency and the complexity that results from modules with multiple, unrelated responsibilities. Moreover, coupling and cohesion are not limited to the object-oriented paradigm. Most programming paradigms have one or more concepts that stand for *module* – "classes", "aspects", "abstract data types", or "source files" – for which coupling and cohesion measures can be calculated.

Our work uses Darcy's metric for structural complexity for two reasons. First, their approach was based on an comprehensive literature review, in which they identified coupling and cohesion as factors that could be used to provide a meaningful measure of structural complexity. Second, they validated the assumption that higher structural complexity is associated with greater maintenance effort by means of a controlled experiment. The authors found out that neither coupling nor lack of cohesion alone could explain the decrease in comprehension performance of the developers; only when coupling and lack of cohesion were considered together as interacting factors, an association with greater maintenance effort could be observed.

A formalization of the metric proposed by Darcy *et al.* [6] can be given as follows. Given a project p and its set of modules $M(p)$, the structural complexity of p is given by $SC(p)$, which is defined as follows.

$$SC(p) = \frac{\sum_{m \in M(p)} CBO(m) \times LCOM4(m)}{|M(p)|}$$

This structural complexity measure is the average structural complexity across all the modules of the system. When a system grows in size and also has its structural complexity increased, this means that not only the system as a whole is larger and therefore more difficult to understand and modify, but every individual module, on average, is also more difficult to understand and modify.

For measuring cohesion, we use Chidamber and Kemerer's CBO [24], calculated as the number of modules a given module depends on. For measuring lack of cohesion, we use Hitz and Montazeri's LCOM4 [25]. The LCOM4 value for a module is the number of connected components of an undirected graph, where the nodes are the module's subroutines (methods, functions etc.), and the edges indicate that two subroutines use at least one attribute/variable in common, or that one subroutines calls the other. These connected components represent independent parts of a module, and modules that have more than one of them have independent, distinct responsibilities.

Growing in size is inevitable for evolving software in a real-world setting, where there is a constant flow of new user requirements to be satisfied and environment changes to deal with. The challenge of modularization is then being able to grow in size while maintaining complexity under control.

B. Developers' Experience

Developer's experience has been covered in past literature in terms of two different dimensions. The first dimension is associated with developers' previous work experience: developers are characterized in terms of "years of experience", or in terms of "skill level", which are represented in ordinal scales such as "low/medium/high" or in simple numeric scales such as "1 to 5". This dimension of experience has been used, for example, as a predictor for project productivity [26], maintenance effort [6], and software product quality attributes [27]. Software development capability methods such as SPI and CMM have been criticized for not taking previous practical experience into account [28].

A second dimension of developer experience is related to developers' activity in a given project. This dimension is characterized by using historical project information to provide a measure of how much experience each developer has on the ongoing project. This can be used, for example, to identify experts in certain parts of the project such as individual modules or subsystems [12], to make sure maintenance teams have at least someone knowledgeable in the maintained project [26], and a as predictor variable for defects [13], [14].

The experience of a developer within a project has been measured using commit data recorded in version control repositories, in terms of: (i) number of commits performed by the developer [14]; or (ii) the number of days since the first commit performed by the developer in the project's version control repository [29].

C. System Size Variation

System size is definitively an attribute that must be observed by software researchers and practitioners. Besides the common sense observation that larger systems are more difficult to deal with, many believe that there is an association between size and complexity. Parnas [15] mentions system size as a factor that makes systems hard to change and evolve. In early software evolution studies, Lehman stated among his laws of sofware evolution both "Continuous Growth" and "Increasing Complexity" [16]. This association is also suggested in a paper by Jay *et al.* [17] that found a very strong correlation between system size and cyclomatic complexity across different programming languages, development methodologies and application domains.

Despite being often criticized, lines of code (LOC) remains as the most widely used measure of system size. Approaches for counting lines of code vary, but ignoring blank and comment lines is a common practice. System size variation may also be computed in terms of lines of code by calculating the difference in the LOC count between two versions of a given codebase.

D. Change diffusion

Small changes or changes that involve a small number of modules tend to require less cognitive effort from developers, while larger changes may involve more modules than a developer can easily reason about. Therefore, we expect that changes that span several modules increase the risk of developers modifying existing software design in a way that makes it more complex.

Change diffusion can be measured as the number of subsystems touched, the number of modules touched, or the number of files touched. It has been characterized as a good predictor for software failures [13]. Change diffusion has also been used to construct prediction models based on change entropy, which outperformed models based on previous changes and previous faults for predicting the occurrence of faults in large free software projects [30]

III. Experimental Setup

In this study, our unit of analysis are *changes made by developers to the project source code*, also known as *commits*. Not all commits made by developers result in variation of structural complexity. Therefore, we have limited ourselves to examine only commits that increased or decreased the structural complexity of the studied systems.

In light of the three factors presented in Section II, the following research questions guided our study:

- RQ1: How does the developer's experience within a project influence the increase or decrease in structural complexity caused by his commits?
- RQ2: How does the size variation introduced by commits influence the increase or decrease in structural complexity?
- RQ3: How does the change diffusion introduced by commits influence the increase or decrease in structural complexity?

A. Research Hypotheses

Our first hypotheses relate the three factors to commits that *increase* structural complexity.

H_1: *Developer experience influences the increase in structural complexity negatively*. We expect commits made by more experienced developers to introduce less structural complexity in the codebase.

H_2: *Size variation influences the increase in structural complexity positively*. Our intuition is that commits that add more code also add more structural complexity.

H_3: *Change diffusion influences the increase in structural complexity positively*. We expect that broad changes touching many files introduce more structural complexity than narrow, localized changes.

The last hypotheses relate the three factors to commits that *decrease* structural complexity:

H_4: *Developer experience influences the decrease in structural complexity positively*. When developers work in complexity-reducing activities, we expect that more experienced developers remove more structural complexity.

H_5: *Size variation influences the decrease in structural complexity negatively*. In complexity-reducing activities, we expect that removing code should cause larger decrease in structural complexity.

H_6: *Change diffusion influences the decrease in structural complexity positively*. In complexity-reducing activities, we expect broader changes to be large reorganizations that make the code less complex, and therefore commits that touch more files are expected to produce a larger decrease in structural complexity.

B. Instrumentation and Measurement

In this study we are interested in the process through which a developer gains experience within a given project, and thus on the second dimension of experience mentioned in Section II-B. We are not interested in making absolute comparisons between developers, but rather to analyze developers' behavior as they get more experienced in the context of a project they work on. Thus, we do not consider any previous experience the developer may have had, and represent experience in terms of time the developer has spent in that given project. We used the following measures of developer experience within the project:

- n – the number of commits the author had made to the project's source code before the current commit (since he started working in the project).
- d – the number of days in which the author of the change had been contributing to the project before the given commit. This is measured by counting the number of days between the first change made by the developer and the date of the current change.

LOC is used for measuring size variation, since it is the most widespread size measure. For change diffusion, we use the number of files that are touched by a commit. They were represented by the following variables:

- ΔLOC – the variation in size of the project, measured in lines of code.
- CF – changed files, a measure of change diffusion. This variable is measured as the number of files affected (added, changed, removed) by the commit, as recorded by the version control system.

For measuring structural complexity, we use the metric by Darcy *et al* [6]. Since we are intested in both commits that increase and that decrease structural complexity, we represent structural complexity increase and decrease in their own variables:

- ΔSC – structural complexity variation. This is the difference between the structural complexity after the commit and the structural complexity before it. This is an auxiliary variable used to calculate the next two.
- ΔSC_i – increase in structural complexity. Defined as ΔSC, if $\Delta SC > 0$. Whenever this variable was used in a test, only the data points where structural complexity actually increased in comparison with the previous commit (i.e. $\Delta SC > 0$) were considered.
- ΔSC_d – decrease in structural complexity. Defined as $|\Delta SC|$, if $\Delta SC < 0$. Whenever this variable was used in a test, only the data points where the structural complexity actually decreased in comparison with the previous commit (i.e. $\Delta SC < 0$) were considered.

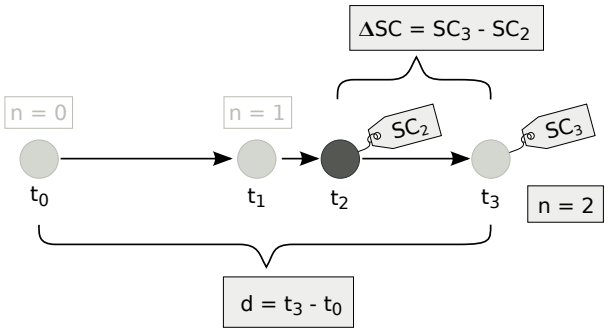


Fig. 1. Variable values for commit at t_3.

Figure 1 describes visually the operational definitions for n, d and ΔSC, exemplifying their calculations for the commit at t_3. Each circle represents a commit in the project's source code. We have information about its date (written below it), its author (represented by the different shades of gray), and also information about the state of the code after the commit (such as metrics values), represented by an attached tag. The commit at t_3, for example, is the third commit made by the light gray developer, so it has $n = 2$, which means that before making that commit, the developer had done two other commits. The value of ΔSC for that commit is then calculated by the difference between the current SC value and the SC value at the previous state of the code, represented by the commit in dark gray. ΔLOC is calculated similarly, but omitted in the figure. d is then calculated by the difference between the current time (t_3) and the time of the first commit that developer made to the project (t_0). CF is also not shown, but is obtained by querying the version control system for how many files were touched by each commit.

Commits that increase and decrease structural complexity were analyzed separately, since we believe they represent conceptually different activities. Commits that increase structural complexity represent the addition of new features and the correction of defects, while commits that decrease structural complexity are the result of anti-regressive work such as refactorings and other source code reorganizations.

C. Analyzed projects

We have studied historical data from five free software projects. The criteria for the project selection was the following: projects should be written in C, C++ or Java; projects must have had at least 18 months of development history available; the version control repository used by the project must be Git. Git is a distributed version control system (VCS) that stores explicitly the author name of each commit done in the project. Older VCS such as CVS and Subversion only store the login name of the user that added the commit to the VCS repository (committer), which is often not the same person who wrote the code (author). Since Git stores both author and committer data, the authorship information is more trustworthy, what is

TABLE I
DESCRIPTIVE INFORMATION FOR STUDIED PROJECTS

Project	Language	Dev. time	#Devs	$ELOC_0$	$ELOC_f$	Modules	Avg. module size	Domain
Clojure	Java	5 years	58	59	31830	666	47.79	Compiler
Node	C++	2 years	164	617	15692	62	253.10	Compiler
Redis	C	2 years	32	4718	16801	48	350.02	Database
Voldemort	Java	2 years	35	9762	34663	434	79.87	Database
Zeromq2	C++	1.75 years	39	7260	11757	106	110.92	Messaging

crucial for a study like the present one.

Projects were selected from GitHub[2]. Descriptive information about the selected projects is presented in table I. Clojure is a dynamic, functional programming language that is a dialect of Lisp. Its main implementation, also called Clojure, compiles Clojure code to JVM bytecode. Node is a platform for JavaScript programming with event-driven I/O based on the V8 JavaScript engine, mainly used for writing server-side applications in JavaScript. Redis and Voldemort are key-value storage systems, often considered to be in the category of "NoSQL databases". Zeromq2 is a messaging library used for process communication in distributed applications.

All projects are infrastructure software, i.e. we did not analyze projects targeted at end users. This was not planned, and is probably due to our choice of looking for projects on GitHub: several free software applications targeted at end users, specially those part of umbrella projects like GNOME, KDE and Xfce, are hosted on their own servers.

Some of these projects started very small, such as Clojure with 59 LOC and Node with 617 LOC (column named $ELOC_0$). Others started their version control history with 60% of their final size, such as Zeromq2.

We consider them to be medium-sized projects: their final size ($ELOC_f$) ranges between 11,000 and 35,000 LOC. When we analyze their size in terms of number of modules, they present a high variance. Clojure is the project with the largest number of modules, containing 666 of them with an average module size of approximately 48 LOC/module. Redis is the project with the largest average module size, containing 48 modules with an average of 350 LOC/module.

These projects also had a significantly high number of distinct developers. Node was the project with more developers among them, with 164 developers recorded in its version control database. The one with fewer developers was Redis, with 32 developers. The developers do not have a uniform volume of participation in the projects, though.

Figure 2 shows the evolution of the studied projects in terms of their size in Lines of Code and its average structural complexity. The X axis shows the project time, starting at the first commit recorded in the version control repository and ending at the last commit that was considered in this paper (approximately until February/2011). Size and average structural complexity are depicted in the Y axis, with both measures normalized against their respective maximum values, so that they range from 0 to 1 in the chart. All projects have been growing in size since the beginning of the observed history, which indicates that they are being subject to an active evolution process. It is also interesting to note that all projects have increasing structural complexity over time, but with very different patterns.

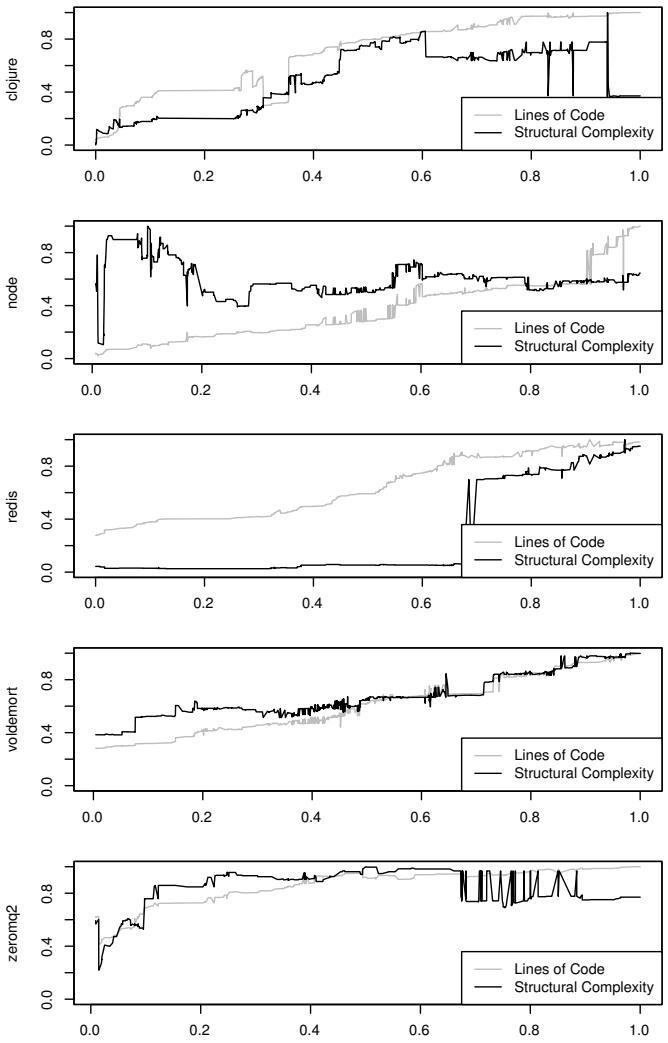


Fig. 2. Evolution of project size and structural complexity for all 5 studied projects. Both measures are normalized against their maximum value for that project

From these charts we can also see that, while in some periods the structural complexity seems to fluctuate together

[2]GitHub (github.com) is a Git-based hosting website largely used by free/open source software projects.

with the system size, that is not always the case. For example, we can notice how the Voldemort structural complexity curve looks very similar to its size curve. On the other hand, during some periods, the structural complexity drops or stays stable even though the system size is increasing: notice Node between time 0.6 and 1.0.

One can also notice periods of instability, in which even though one of the measures is evolving smoothly, the other is subject to sharp increases or decreases. In Redis, for example, between time 0.6 and 0.8 the size was growing approximately in a linear way, while the structural complexity exploded at approximately time 0.7.

D. Data extraction process

The version control repositories of the selected projects were processed with analizo[3], a multi-language source code analysis and visualization toolkit. The procedure for each project is described below.

First, the project Git repository was cloned locally. Then, we listed all commit author names in the repository and created a mapping file to normalize their names. Since some authors use slightly different names during the project history, by doing this we avoid considering commits done by the same person as being made by different developers.

Then we determined programming language inclusion rules, to avoid analyzing code that is not part of the main project. For example Clojure and Node are compilers, and contain a lot of code in the language they compile (resp. Clojure and Javascript). We only analyzed their main code: Java for Clojure and C++ for Node. After that, we determined directory exclusion rules, to exclude directories that should not be included in the source code analysis, such as embedded dependencies, test code, etc. This was done by listing all the different source code tree configurations with analizo's `tree-evolution` tool and compiling a list of directories to be excluded for each project.

The repositories were then processed with analizo's `metrics-history` tool using the language and directory filter rules described previously, to obtain an initial project-specific dataset. This data set contained, for each commit that changed the source code, one row containing meta-information about the commit plus source code metrics for the corresponding source code (i.e. the resulting source code, *after* the commit was done). After that, the datasets were imported into a common relational database and post-processed to calculate the variables described previously (n, d, ΔLOC, CF, ΔSC, ΔSC_i, and ΔSC_d).

When calculating structural complexity for C++ and Java source code, classes were considered as "modules". In the case of C, a "module" is a pair of .c/.h source files.

IV. Results

In this section, we present and interpret the experimental results according to the research hypotheses presented in Section III-A. All data analysis and statistical tests were performed using the R system [31].

An initial interesting fact that resulted from our data analysis was that the different ways of measuring experience within a project we propose in this paper are practically equivalent. Table II shows, for each project, the Spearman's ρ correlation between n (number of commits done by the developer so far) and d (number of days since the developer started in the project). One can see that there is a very strong linear correlation between these variables, what means we only need to look at one of them. For the rest of this study, we will only present and discuss the hypotheses using n as independent variable: their results are similar to the ones where we used d as independent variable.

TABLE II
LINEAR CORRELATIONS BETWEEN n AND d

Project	Commits	$\rho(n, d)$
clojure	522	0.996
node	240	0.961
redis	117	0.984
voldemort	394	0.875
zeromq2	128	0.884

All correlations statistically significant with $p < 0.001$

To test hypotheses H_1 to H_3, we built regression models in which the structural complexity increase is modeled as a linear combination of the experience of the developer making that change, the size variation and the change diffusion:

$$\Delta SC_i = \alpha_0 + \alpha_1 n + \alpha_2 \Delta LOC + \alpha_3 CF$$

Table III presents the resulting models for the commits that increase structural complexity.

We can see that H_1 (developer experience influences the increase in structural complexity negatively) holds only for Node, which is the only project in which the experience variable n has a statistically significant coefficient with a negative sign. Our interpretation is that the changes made by less experienced developers introduce more structural complexity in the project codebase, or alternatively that more experienced developers introduce less structural complexity. Even though that coefficient seems to have a low magnitude, one must note that the measures are in different scales.

H_2 (size variation influences the increase in structural complexity positively) holds for Clojure, Voldemort and Zeromq2. In all three projects, changes that introduce more lines of code also introduce more structural complexity. The same observation about the magnitude of the coefficient applies here: ΔLOC and ΔSC_i are measured in different scales.

Last, we have found that H_3 (change diffusion influences the increase in structural complexity positively) holds for all projects, except Zeromq2. In these projects changes that touch more files tend to introduce more structural complexity.

To test our hypotheses H_4 to H_6, we have built linear regression models similar to the ones presented above. For each project, we modeled the decrease in structural complexity

[3]More information about Analizo can be found at analizo.org.

TABLE III
REGRESSION MODELS FOR COMMITS THAT **INCREASE** STRUCTURAL COMPLEXITY

Project	#Commits	Intercept	n (H_1)	ΔLOC (H_2)	CF (H_3)	Adj. R^2
clojure	287	0.031637	0.000108	**0.000147** ***	**0.014175** ***	0.20
node	133	**0.340228** ***	**-0.000479** **	0.000369	**0.025465** *	0.10
redis	72	-1.848899	0.004697	0.00244	**0.236758** ***	0.31
voldemort	217	0.026552	-5e-06	**0.000195** **	**0.002326** ***	0.22
zeromq2	71	**0.224476** **	-0.000399	**0.000746** **	0.004965	0.21

***: $p < 0.001$; **: $p < 0.01$; *: $p < 0.05$

TABLE IV
REGRESSION MODELS FOR COMMITS THAT **DECREASE** STRUCTURAL COMPLEXITY

Project	#Commits	Intercept	n (H_4)	ΔLOC (H_5)	CF (H_6)	Adj. R^2
clojure	235	-0.064833	**0.000422** **	0.000204	0.011493	0.04
node	107	**0.313235** *	-0.00038	-0.000308	**0.036385** **	0.10
redis	45	-0.359334	0.000952	0.000443	**0.163235** **	0.18
voldemort	177	**0.029518** *	2.3e-05	**0.00013** **	0.001125	0.07
zeromq2	57	0.10271	-0.000372	**-0.001258** ***	**0.021416** ***	0.88

***: $p < 0.001$; **: $p < 0.01$; *: $p < 0.05$

as a linear combination of experience of the developer, size variation and change diffusion:

$$\Delta SC_d = \alpha_0 + \alpha_1 n + \alpha_2 \Delta LOC + \alpha_3 CF$$

The results provided by these regression models are presented in table IV.

H_4 (developer experience influences the decrease in structural complexity positively) holds only for Clojure. Our interpretation is that in Clojure, the changes made by developers with more experience are the ones that cause larger decreases in structural complexity. In that project, anti-regressive work is done by the most experienced developers.

H_5 (size variation influences the decrease in structural complexity negatively) holds only for Zeromq2. In that project, the commits that produce larger decreases in structural complexity are the ones with lower number of lines of code added, or commits that actually remove lines of code. Voldemort also has a significant coefficient for ΔLOC, but that coefficient is negative: in that project commits that add more lines of code are the ones producing larger decreases in structural complexity. This sounds counter-intuitive: maybe the Voldemort developers were able to reduce complexity by splitting existing modules into new ones that are larger but less coupled and/or more cohesive.

H_6 (change diffusion influences the decrease in structural complexity positively) holds for Node, Redis and Zeromq2. Our interpretation is that in those projects the commits that touch more files are the ones that cause a greater reduction in structural complexity.

V. DISCUSSION

The projects we analyzed in this study provided widely different results for all the studied factors. In order to identify differences between the projects and conjecture about the reasons for their structural complexity variation being influenced by some of the factors and not by the others, we want to look back at their descriptive data in table I.

Node was the only project in which higher levels of experience within the project were associated with smaller increases in structural complexity in complexity-increasing changes. The only notable difference between Node and all other projects is the fact that Node had a higher number of distinct contributors than the other projects. A possible explanation is that more experience only leads to smaller increases in structural complexity in projects with large teams, i.e. as far as structural complexity is concerned, larger teams would benefit more from having more experienced developers.

When looking at complexity-decreasing changes, we only found significant influence of experience in Clojure: in that project, changes made by developers with higher experience within the project produced a larger reduction in structural complexity. Clojure is the project with the longest development history among all of the studied projects: 5 years, against mostly 2 years for the other projects. A possible explanation is that in long-running projects, the more experienced developers are the ones that end up performing anti-regressive activities, such as refactorings.

Node and Clojure are the projects in which experience did play a role in influencing the amount of structural complexity added or removed by developers. Comparing them with the other projects, there are two aspects that called our attention:

- Both Node and Clojure started from very few lines of code ($ELOC_0$) in comparison with the their final size ($ELOC_f$), contrary to what happened to the other projects. A reasonable explanation for that would be that those projects had their important achitectural decisions taken after they were first imported in the version control system. The same developers that made those decisions in the beginning would be the ones that now are able to introduce new code without adding so much complexity (in the case of Node), or to remove more complexity when doing reorganizations in the code (in the case of Clojure). The other projects could have had their important architectural decisions taken before they could be

TABLE V
FACTORS INFLUENCING THE STRUCTURAL COMPLEXITY VARIATION.

Project	$\Delta SC > 0$	Adj. R^2	$\Delta SC < 0$	Adj. R^2
Clojure	ΔLOC, CF	0.20	Experience	0.04
Node	Experience, CF	0.10	CF	0.10
Redis	CF	0.31	CF	0.18
Voldemort	ΔLOC, CF	0.22	ΔLOC	0.07
Zeromq2	ΔLOC	0.21	ΔLOC, CF	0.88

tracked, and the developers were left with no resources, no matter how experienced they were, to contain the structural complexity increase (or to drive its decrease).

- Both Node and Clojure are in the compilers domain, while the other projects in which experience did not influence structural complexity variation significantly are in other domains. It is possible that experience only plays a significant role in specific applications domains (compilers in the case of the present study) and not in others.
- It is also possible that some characteristics exhibited by Node and Clojure that was not captured in this study might cause their results to be different from the ones in the other projects.

Table V presents a summary of the obtained regression models, with the factors that had a statistically significant influence over the structural complexity variation for both types of change.

We can see that CF is a recurring factor, being present in 7 out of 10 cases. The fact that a commit needs to change a large number of files may be a sign that functionality is not well-factored in independent modules: whenever a developer needs to add a new functionality or correct a defect, she needs to change several separate modules. When changing lots of different modules is also associated with larger increases in structural complexity, this is an indicator of a poor modularization that is harmful to the understandability of the codebase.

ΔLOC is also a significant factor in 5 out of 10 cases. In the case of changes that decrease complexity, it is understandable why removing code makes the software less complex. In the case of changes that increase structural complexity, the fact that an increase in size necessarily implies an increase in structural complexity may also be a sign of a suboptimal modularization.

Structural complexity is an average across all modules, and so it may increase with size in two ways. First, maybe new functionality was added to existing modules while making them more complex; second, maybe functionality was added as new modules that are more complex than the average. In both cases, this may be a sign of poor modularization.

On other hand, we did *not* find a significant influence of ΔLOC over structural complexity variation in 5 out of 10 cases. While system size variation must be looked after, there are cases in which systems complexity does not vary according to how size does. That is the case for the Node project. As we saw before in figure 2, Node structural complexity evolution curve presents no similarity whatsoever with the corresponding size evolution curve.

Software developers and project managers can use the results discussed above by inspecting the past commits that changed the structural complexity and had a large number of changed files or a large size variation to figure out the reason why those commits make the software more complex. They can them make informed decisions on where to direct their perfective maintenance efforts. They can also use these results to monitor future changes. If the team does not have enough resources to explicitly review every commit, they can give priority to commits that add more than a specified number of lines, or that touch more than a specified number of files, since those commits may be more likely to cause larger variations in the structural complexity of the codebase.

Another important fact is that most of the models do not have a large coefficient of determination (adjusted R^2). This coefficient can be interpreted as the amount of variation in the dependent variables (ΔSC_i and ΔSC_d) that is predicted by the independent variables, or how likely the dependent variable might be predicted by the resulting regression model. 0 means no prediction at all, and 1 means that the independent variables predict the dependent variable perfectly.

For example, in the case of changes that decrease complexity in Zeromq2, 88% of the variation is predicted by just ΔLOC and CF: in that project, decreasing the complexity necessarily means removing code, what can also be a sign of a suboptimal modularization.

In the other cases, we did not obtain such a good a model: coefficients of determination range from 0.04 to 0.31, what leaves room for plenty of future work in order to obtain improved models that can predict the structural complexity variation in those projects with higher accuracy.

VI. THREATS TO VALIDITY

Although it was carefully designed, the present study is not free from threats to its validity. We checked our study against the list of threats to validity by Wohlin *et al* [32] and the issues we found are reported in this section.

Since we only looked at free software projects, we cannot assume that the results hold for the general case of software development projects. Strictly speaking, the results also cannot be generalized to other free software projects, since that classification does not imply the use of any specific development methodology, platform, tool, or technique.

An inherent limitation of the metric we used for developer experience within the project is that it always increases and never decays. This could have been compensated by introducing another measure representing the developers' actual expertise with the codebase, which would increase as the developer works in the project and decay in periods of inactivity, or when other developers change the code they wrote.

We also did not look at the nature of the change introduced by commits: the fact that a commit represents a bug fix, the implementation of a new feature or a refactoring may make a big difference. For example, the nature of structural complexity that is introduced by the implementation of a new

feature may be different from the structural complexity that is introduced by a refactoring. A refactoring could increase the structural complexity because it is adding framework-like capabilities to the code base, what would in turn allow future changes to be made with a smaller increase in structural complexity.

Commits that did not change structural complexity – i.e. those commits for which $\Delta SC = 0$ – were not included in our analysis. It is possible for a commit that does not change the structural complexity to actually influence future commits that do alter the structural complexity. For example, a commit that introduces a serious defect might lead developers to realize that they need a large refactoring to properly avoid the same type of defect in the future. Including these commits in the analysis performed in this study would not help, though: since they all exhibit the same value for the dependent variable ($\Delta SC = 0$!), they would provide us no useful statistical results. We would need a different research design to take advantage of those additional data points.

Another limitation is the assumption that each commit represents a self-contained change. If there were logical changes that were performed as a sequence of separate commits, this information was lost and each commit was analyzed as being an independent change.

VII. Conclusions

This paper reports on a study to identify factors that influence the structural complexity evolution. We investigated the influence of developer experience within the project, size variation and change diffusion in the variation of structural complexity, using commit data from 5 well-established free/open source projects in different application domains and written in different programming languages.

The method applied and results found can be used as guidance to support decisions on how to assign developers for maintenance work. For example, managers that find an association between lower developer experience within a project and structural complexity increase might want to allocate their most experienced developers to work on risky changes. If no such association is found in the history of the project, maybe there is no point in allocating a specific developer to perform the change.

Managers can also monitor changes and decide whether a specific change needs to be explicitly reviewed by a second developer when it crosses a threshold of number of lines of code or number of files changed, if those are relevant factors for a given project.

This study taught us important lessons, which should be taken into consideration by practitioners looking for applying these results in their software development projects.

All of the studied factors have influence over structural complexity evolution in at least 2 regression models. This means that all of developer experience within the project, size variation and change diffusion might be taken into consideration by project managers when monitoring software metrics, or to decide on how to prioritize resources for code review.

Change diffusion is most influential factor among the ones we studied. In 7 out of 10 regression models, changes that touched more files were the ones that added/removed more structural complexity and removed. Software size was the second most influential factor, being significant in 5 out of 10 regression models. Developer experience only had a significant influence in 2 out of 10 regression models, but in one of them neither size variation nor change diffusion were significant.

System growth is not necessarily associated with structural complexity increase. Contrary to the common sense, it is possible for a project to grow without having a corresponding increase in complexity.

Different projects are influenced by different factors. A direct implication for project managers is that to apply these results on software development projects, previous project history must be studied to identify which factors influence specific projects.

Factors that influence the increase of structural complexity are different from the ones that influence the decrease. Depending on their goals, project managers might consider a different set of factors: if the main goal is to control or avoid the increase in structural complexity, a different set of factors must be considered in comparison with when the goal is to actively work in the reduction of structural complexity by means of activities such as refactorings and re-engineering in general.

There are other influencing factors that we did not investigate in this study. Our regression models did not achieve a high coefficient of determination, what means that there are other factors that influence structural complexity variation that were not considered in the present study. Future work might involve performing qualitative studies and look in detail at specific changes, trying to identify other factors that might be included in a quantitative analysis such as the one presented in this work.

We also plan to investigate other factors that may influence the increase (or reduction) of structural complexity in software projects. This includes both developer characteristics, such as familiarity with the modules being changed, and factors associated with the nature of changes performed, such as the type of maintenance (corrective, perfective etc) that is taking place at each commit.

Another venue for future work is analyzing change in structural complexity in a more detailed context: when we analyze a given change in the context of a large code base, it is perceived to make a smaller difference than if it was analyzed only in the context of e.g. the modified subsystem.

Yet another idea for future work is to perform a qualitative study on the different amounts of structural complexity variation. In figure 2 we can see that there are several points in time where projects have very sharp increases or decreases in structural complexity, and sometimes these large variations don't come together with a comparable variation in size. A closer observation of those changes might provide a deeper insight about factors that influence structural complexity. Such a study

would also help to clarify counter-intuitive results such as the ones for the Voldemort project (a negative influence of size variation in structural complexity reduction and $R^2 = 0.88$) presented in section IV.

ACKNOWLEDGMENTS

The authors would like to thank colleagues from the Software Practices Lab at the University of British Columbia for the insightful comments. Special thanks to Gail Murphy, Nicholas Sawadsky and Robin Salkeld who provided precious feedback on a draft version of this paper. This work has been partially supported by INES (National Institute of Science and Technology for Software Engineering), funded by CNPq (Brazilian National Council of Research and Development), grant 573964/2008-4.

REFERENCES

[1] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 73–87. [Online]. Available: http://doi.acm.org/10.1145/336512.336534

[2] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, pp. 308–320, dec. 1976.

[3] R. L. Sedlmeyer, J. K. Kearney, W. B. Thompson, M. A. Adler, and M. A. Gray, "Problems with software complexity measurement," in *Proceedings of the 1985 ACM thirteenth annual conference on Computer Science*, ser. CSC '85. New York, NY, USA: ACM, 1985, pp. 340–347.

[4] J. L. Roca, "An entropy-based method for computing software structural complexity," *Microelectronics and Reliability*, vol. 36, no. 5, pp. 609–620, 1996.

[5] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005, pp. 167–176.

[6] D. P. Darcy, C. F. Kemerer, S. A. Slaughter, and J. E. Tomayko, "The structural complexity of software: An experimental test," *IEEE Transactions on Software Engineering*, vol. 31, no. 11, pp. 982–995, Nov. 2005.

[7] R. S. Sangwan, P. Vercellone-Smith, and P. A. Laplante, "Structural epochs in the complexity of software over time," *Software, IEEE*, vol. 25, no. 4, pp. 66–73, july-aug. 2008.

[8] V. Midha, "Does complexity matter? the impact of change in structural complexity on software maintenance and new developers' contributions in open source software," in *ICIS 2008 Proceedings*, 2008.

[9] P. Meirelles, C. S. Jr., J. Miranda, F. Kon, A. Terceiro, and C. Chavez, "A study of the relationships between source code metrics and attractiveness in free software projects," in *CBSOFT-SBES2010*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2010, pp. 11–20.

[10] K. J. Stewart, D. P. Darcy, and S. L. Daniel, "Opportunities and challenges applying functional data analysis to the study of open source software evolution," *Statistical Science*, vol. 21, p. 167, 2006.

[11] D. P. Darcy, S. L. Daniel, and K. J. Stewart, "Exploring complexity in open source software: Evolutionary patterns, antecedents, and outcomes," *Hawaii International Conference on System Sciences*, vol. 0, pp. 1–11, 2010.

[12] A. Mockus and J. D. Herbsleb, "Expertise browser: a quantitative approach to identifying expertise," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 503–512.

[13] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Tech. J.*, vol. 5, no. 2, pp. 169–180, 2000.

[14] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '10. New York, NY, USA: ACM, 2010, pp. 18–1.

[15] D. L. Parnas, "Software aging," in *ICSE '94: Proceedings of the 16th international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 279–287.

[16] M. M. Lehman, J. F. Ramil, P. D. Wernick, and D. E. Perry, "Metrics and laws of software evolution-the nineties view," in *Proceedings of the 4th International Symposium on Software Metrics*, 1997.

[17] G. Jay, J. E. Hale, R. K. Smith, D. Hale, N. A. Kraft, and C. Ward, "Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship," *Journal of Software Engineering and Applications*, vol. 2, no. 3, pp. 137–143, 2009.

[18] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architecture : Views and Beyond*, ser. The SEI series in software engineering. Boston: Addison-Wesley, 2002.

[19] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[20] E. J. Barry, C. F. Kemerer, and S. A. Slaughter, "How software process automation affects software evolution: a longitudinal empirical analysis," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 1, pp. 1–31, 2007.

[21] M. Zhang and N. Baddoo, "Performance comparison of software complexity metrics in an open source project," in *Proceedings of Software Process Improvement, 14th European Conference, EuroSPI 2007, Potsdam, Germany, September 26-28, 2007*, 2007, pp. 160–174.

[22] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[23] S. McConnell, *Code complete*, ser. DV-Professional. Microsoft Press, 2004.

[24] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Sftware Eng.*, vol. 20, no. 8, pp. 476–493, 1994.

[25] M. Hitz and B. Montazeri, "Measuring coupling and cohesion in object-oriented systems," in *Proceedings of the International. Symposium on Applied Corporate Computing*, 1995.

[26] R. D. Banker, S. M. Datar, and C. F. Kemerer, "A model to evaluate variables impacting the productivity of software maintenance projects," *Manage. Sci.*, vol. 37, no. 1, pp. 1–18, January 1991.

[27] J. M. Beaver and G. A. Schiavone, "The effects of development team skill on software product quality," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 3, pp. 1–5, May 2006.

[28] O. Steen, "Practical knowledge and its importance for software product quality," *Inf. Softw. Technol.*, vol. 49, no. 6, pp. 625–636, June 2007.

[29] M. Zhou and A. Mockus, "Developer fluency: achieving true mastery in software projects," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 137–146.

[30] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070510

[31] R Development Core Team, *R: A Language and Environment for Statistical Computing*, Vienna, Austria, 2009, iSBN 3-900051-07-0. [Online]. Available: http://www.R-project.org

[32] C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: an Introduction*. Kluver Academic Publishers, 2000.

Apêndice D

Este apêndice contém uma reprodução de artigo publicado nos anais da sessão de ferramentas do I Congresso Brasileiro de Software, em 2010.

ANALIZO: AN EXTENSIBLE MULTI-LANGUAGE SOURCE CODE ANALYSIS AND VISUALIZATION TOOLKIT

Analizo: an Extensible Multi-Language Source Code Analysis and Visualization Toolkit

Antonio Terceiro[1], **Joenio Costa**[2], **João Miranda**[3], **Paulo Meirelles**[3], **Luiz Romário Rios**[1], **Lucianna Almeida**[3], **Christina Chavez**[1], **Fabio Kon**[3]

[1] Universidade Federal da Bahia (UFBA)

{terceiro,luizromario,flach}@dcc.ufba.br

[2]Universidade Católica do Salvador (UCSAL)

joenio@perl.org.br

[3]Universidade de São Paulo (USP)

{joaomm,paulormm,lucianna,fabio.kon}@ime.usp.br

Abstract. *This paper describes Analizo, a free, multi-language, extensible source code analysis and visualization toolkit. It supports the extraction and calculation of a fair number of source code metrics, generation of dependency graphs, and software evolution analysis.*

1. Introduction

Software engineers need to analyze and visualize the software they create or maintain in order to better understand it. Software Engineering researchers need to analyze software products in order to draw conclusions in their research activities. However analyzing and visualizing large individual software products or a large number of individual software products is only cost-effective with the assistance of automated tools.

Our research group have been working with empirical studies that require large-scale source code analysis, and consequently we resort to source code analysis tools in order to support some of our tasks. We have defined the following requirements for the tool support we needed:

- Multi-language. The tool should support the analysis of different programming languages (in particular, at least C, C++ and Java), since this can enhance the validity of our studies.
- Free software. The tool should be free software[1], available without restrictions, in order to promote the replicability of our studies by other researchers.
- Extensibility. The tool should provide clear interfaces for adding new types of analyzes, metrics, or output formats, in order to promote the continuous support to our studies as the research progresses.

In this paper, we present Analizo, a toolkit for source code analysis and visualization, developed with the goal of fulfilling these requirements. Section 2 describes related work. Section 3 describes Analizo architecture. Section 4 presents Analizo features. Section 5 presents Analizo use cases. Finally, Section 6 concludes the paper and discusses future work.

[1]In our work, we consider the terms “free software” and “open source software” equivalent.

2. Related work

While evaluating the existing tools to use in our research, we analyzed the following ones: CCCC [Littlefair 2010], Cscope [Steffen et al. 2009], LDX [Hassan et al. 2005], CTAGX [Hassan et al. 2005], and CPPX [Hassan et al. 2005]. Besides the research requirements described, we have included two practical requirements:

- The tool must be actively maintained. This involves having active developers who know the tool architecture and can provide updates and defect corrections.
- The tool must handle source code that cannot be compiled anymore. For example, the code may have syntax errors, the libraries it references may be not available anymore, or the used libraries changed API. This is important in order to be able to analyze legacy source code in software evolution studies.

The requirements evaluation for the tools are presented in Table 1. Since we only looked at tools that were free software, the table does not have a line for that requirement.

Requirement	**CCCC**	**Cscope**	**LDX**	**CTAGX**	**CPPX**
Language support	C++, Java	C	C, C++	C	C, C++
Extensibility	No	No	No	No	No
Maintained	Yes	Yes	No	No	No
Handles non-compiling code	Yes	No	No	No	No

Table 1. Found tools versus posed requirements

As it can be seen in Table 1, none of the existing tools we found fulfills all of our requirements. In special, none of the tools were able to analyze source code in all three needed languages, and none of them had documented extension interfaces that could be used to develop new analysis types or output formats.

3. Architecture

Analizo architecture is presented in Figure 1, using a Layered style [Clements et al. 2002]. Each layer in the diagram uses only the services provided by the layers directly below it.

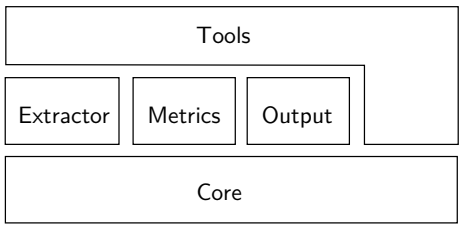


Figure 1. Analizo architecture, using the Layered Style [Clements et al. 2002]

The *Core* layer contains the data structures used to store information concerning the source code being analyzed, such as the list of existing modules[2], elements inside each module (attributes/variables, or methods/functions), dependency information (call,

[2]we used the "module" concept as a general term for the different types of structures used in software development, as classes and C source files

inheritance, etc). This layer implements most of Analizo business logic, and it does not depend on any other layer.

The *Extractors* layer comprises the different source code information extraction strategies built in Analizo. Extractors get information from source code and store them in the *Core* layer data structures. It requires only the creation of a new subclass to add a new type of extractor that interfaces with another external tool or provides its own analysis directly. Currently, there are two extractors. Both are interfaces for external source code parsing tools:

- *Analizo::Extractors::Doxyparse* is an interface for Doxyparse, a source code parser for C, C++ and Java developed by our group [Costa 2009]. Doxyparse is based on Doxygen[3], a multi-language source code documentation system that contains a robust parser.
- *Analizo::Extractors::Sloccount* is an interface for David A. Wheeler's Sloccount[4], a tool that calculates the number of effective lines of code.

The other intermediate layers are *Metrics* and *Output*. The *Metrics* layer processes *Core* data structures in order to calculate metrics. At the moment, Analizo supports a fair set of metrics (listed in Section 4). The *Output* layer is responsible for handling different file formats. Currently, the only output format implemented is the DOT format for dependency graphs, but adding new formats is simply a matter of adding new output handler classes.

The *Tools* layer comprises a set of command-line tools that constitute Analizo interface for both users and higher-level applications. These tools use services provided by the other layers: they instantiate the core data structures, one or more extractors, optionally the metrics processors, an output format module, and orchestrate them in order to provide the desired result. Most of the features described in Section 4 are implemented as Analizo tools.

Those tools are designed to adhere to the UNIX philosophy: they accomplish specialized tasks and generate output that is suitable to be fed as input to other tools, either from Analizo itself or other external tools. Some of the tools are implemented on top of others instead of explicitly manipulating Analizo internals, and some are designed to provide output for external applications such as graph drawing programs or data analysis and visualization applications.

4. Features

4.1. Multi-language source code analysis

Currently, Analizo supports source analysis of code written in C, C++ and Java. However, it can be extended to support other languages since it uses Doxyparse, which is based on Doxygen and thus also supports several different languages.

4.2. Metrics

Analizo reports both project-level metrics, which are calculated for the entire project, and module-level metrics, which are calculated individually for each module. On the

[3] doxygen.org/
[4] dwheeler.com/sloccount/

project-level, Analizo also provides basic descriptive statistics for each of the module-level metrics: sum, mean, median, mode, standard deviation, variance, skewness and kurtosis of the distribution, minimum, and maximum value. The following metrics are supported at the time of writing[5]:

- Project-level metrics: Total Coupling Factor, Total Lines of Code, Total number of methods per abstract class, Total Number of Modules/Classes, Total number of modules/classes with at least one defined attributes, Total number of modules/classes with at least one defined method, Total Number of Methods.
- Module-level metrics: Afferent Connections per Class, Average Cyclomatic Complexity per Method, Average Method LOC, Average Number of Parameters per Method, Coupling Between Objects, Depth of Inheritance Tree, Lack of Cohesion of Methods, Lines of Code, Max Method LOC, Number of Attributes, Number of Children, Number of Methods, Number of Public Attributes, Number of Public Methods, Response For a Class.

4.3. Metrics batch processing

In most quantitative studies on Software Engineering involving the acquisition of source code metrics on a large number of projects, processing each project individually is impractical, error-prone and difficult to repeat. Analizo can process multiple projects in batch and produce one comma-separated values (CSV) metrics data file for each project, as well as a summary CSV data file with project-level metrics for all projects. These data files can be easily imported in statistical tools or in spreadsheet software for further analysis. This can also be used to analyze several releases of the same project, in software evolution studies.

4.4. Metrics history

Sometimes researchers need to process the history of software projects on a more fine-grained scale. Analizo can process a version control repository and provide a CSV data file with the metrics values for each revision in which source code was changed in the project. Git and Subversion repositories are supported directly, and CVS repositories must be converted into Git ones beforehand.

4.5. Dependency Graph output

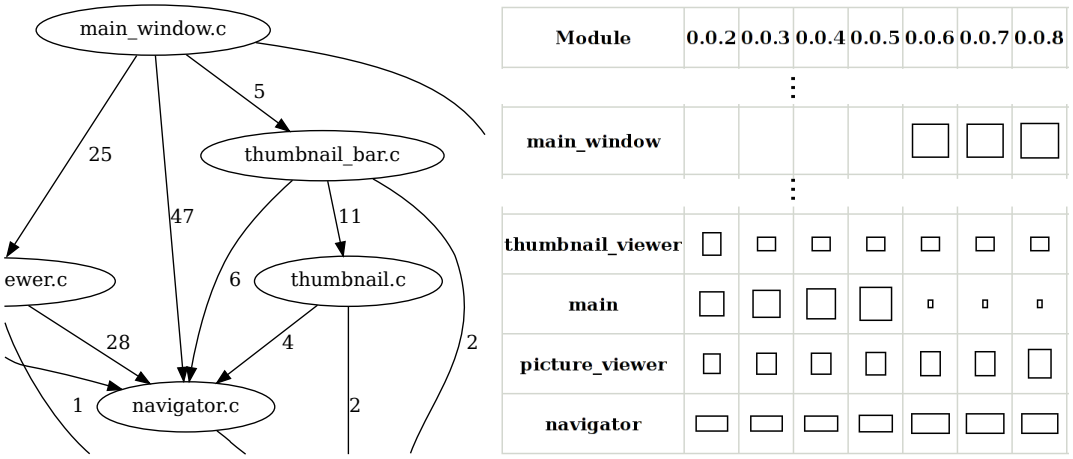
Analizo can output module dependency information extracted from a source code tree in a format suitable for processing with the Graphviz[6] graph drawing tools. Figure 2(a) presents a sample dependency graph obtained by feeding Graphviz' *dot* tool with Analizo graph output.

4.6. Evolution matrix

Another useful Analizo feature is generating evolution matrices [Lanza 2001]. After processing each release of the project (see Section 4.3), the user can request the creation of an evolution matrix from the individual data files. Figure 2(b) shows an excerpt of a sample evolution matrix produced by Analizo.

[5]References to literature on each metric were omitted because of space constraints.

[6]`graphviz.org/`



(a) Sample module dependency graph (b) Sample evolution matrix

Figure 2. Examples of Analizo features.

5. Usage in research work

Analizo has been extensively used by our group to support research projects:

- [Amaral 2009] used Analizo module dependency graph output to produce an evolution matrix for a case study on the evolution of the VLC project. Later on, an evolution matrix tool was incorporated in Analizo itself.
- [Costa 2009] did a comparison between different strategies for extracting module dependency information from source code, leading to the development of Doxyparse – the Analizo Doxygen-based extractor.
- [Terceiro and Chavez 2009] used the metrics output on an exploratory study on the evolution of structural complexity in a free software project written in C.
- [Morais et al. 2009] used the Analizo metrics tool as a backend for Kalibro[7], a software metrics evaluation tool. Later on, Kalibro Web Service[8] was developed, providing an integration with Spago4Q[9] – a free platform to measure, analyze and monitor quality of products, processes and services.
- [Terceiro et al. 2010] used the metrics history processing feature to analyze the complete history of changes in 7 web server projects of varying sizes.
- [Meirelles et al. 2010] used Analizo metrics batch feature to process the source code of more than 6000 free software projects from the Sourceforge.net repository.

Most of the work cited above contributed to improvements in Analizo, making it even more appropriate for research involving source code analysis.

6. Final remarks

This paper presented Analizo, a toolkit for source code analysis and visualization that currently supports C, C++ and Java. Analizo has useful features for both researchers working with source code analysis and professionals who want to analyze their source code in order to identify potential problems or possible enhancements.

[7] `softwarelivre.org/mezuro/kalibro/`
[8] `ccsl.ime.usp.br/kalibro-service`
[9] `spago4q.org`

Future work includes the development of a web-based platform for source code analysis and visualization based on Analizo. This project is current under development.

Analizo is free software, licensed under the GNU General Public License version 3. Its source code, as well as pre-made binary packages, manuals and tutorials can be obtained from `softwarelivre.org/mezuro/analizo`. All tools are self-documented and provide an accompanying UNIX manual page. Analizo is mostly written in Perl, with some of its tools written in Ruby and Shell Script.

This work is supported by CNPQ, FAPESB, the National Institute of Science and Technology for Software Engineering (INES), Qualipso project, and USP FLOSS Competence Center (CCSL-USP).

References

Amaral, V. (2009). Análise de evolucao de projetos de software livre através de matrizes de evolucao. Undergraduation course conclusion project, Universidade Federal da Bahia.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. (2002). *Documenting Software Architecture : Views and Beyond*. The SEI series in software engineering. Addison-Wesley, Boston.

Costa, J. (2009). Extração de informações de dependência entre módulos de programas c/c++. Undergraduation course conclusion project, Universidade Católica do Salvador.

Hassan, A. E., Jiang, Z. M., and Holt, R. C. (2005). Source versus object code extraction for recovering software architecture. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE'05)*.

Lanza, M. (2001). The evolution matrix: recovering software evolution using software visualization techniques. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 37–42, New York, NY, USA. ACM.

Littlefair, T. (2010). CCCC - C and C++ Code Counter. Available at `http://cccc.sourceforge.net/`. Last access on June 3rd, 2010.

Meirelles, P., Jr., C. S., Miranda, J., Kon, F., Terceiro, A., and Chavez, C. (2010). A Study of the Relationship between Source Code Metrics and Attractiveness in Free Software Projects. *Submitted*.

Morais, C., Meirelles, P., and Kon, F. (2009). Kalibro: Uma ferramenta de configuração e interpretação de métricas de código-fonte. Undergraduation course conclusion project, Universidade de São Paulo.

Steffen, J., Hans-Bernhard, and Horman, B. N. (2009). *Cscope*. http://cscope.sourceforge.net/.

Terceiro, A. and Chavez, C. (2009). Structural Complexity Evolution in Free Software Projects: A Case Study. In Ali Babar, M., Lundell, B., and van der Linden, F., editors, *QACOS-OSSPL 2009: Proceedings of the Joint Workshop on Quality and Architectural Concerns in Open Source Software (QACOS) and Open Source Software and Product Lines (OSSPL)*.

Terceiro, A., Rios, L. R., and Chavez, C. (2010). An Empirical Study on the Structural Complexity introduced by Core and Peripheral Developers in Free Software projects. *Submitted*.

Programa Multiinstitucional de
Pós-Graduação em Ciência da Computação – PMCC

PMCC-IM-UFBA, Campus de Ondina
Av. Adhemar de Barros S/N, Salvador - Bahia, CEP 40.170-110
dmcc@ufba.br http://dmcc.dcc.ufba.br