Identificação de Padrões de Características Estruturais em Software Orientado a Objetos

Kecia A. M. Ferreira[1], **Roberta C. N. Moreira**[1], **Mariza A. S. Bigonha**[2]

[1]Departamento de Computação – CEFET-MG
Av. Amazonas, 7675 – Nova Gameleira – Belo Horizonte-MG

[2]Departamento de Ciência da Computação – UFMG
Av. Antônio Carlos, 6627 – Pampulha – Belo Horizonte-MG

kecia@decom.cefetmg.br, robertacoelineves@gmail.com, mariza@dcc.ufmg.br

Resumo. *Manter os softwares criados em funcionamento e com qualidade é um grande desafio. A compreensão profunda do software pode favorecer o controle dos impactos das modificações realizadas e a gestão do processo de manutenção de software. Todavia, para alcançar isso, são necessários recursos que permitam a avaliação apropriada do software. Buscando contribuir com um recurso desta natureza, um trabalho anterior das autoras do presente artigo define um modelo, denominado Little House, que consiste em uma figura macroscópica genérica das estruturas de software orientado a objetos. Little House modela o software como um grafo com cinco vértices, denominados componentes, que correspondem a conjuntos de classes conectados entre si. O presente trabalho tem por objetivo caracterizar qualitativamente esses componentes por meio de estudos de caso com cinco softwares Java. O estudo consistiu em inspecionar manualmente todas as classes a fim de identificar seus propósitos. A análise dos resultados revela como as classes dos softwares são distribuídas entre os componentes de Little House conforme os propósitos delas. Os resultados deste trabalho identificam padrões de características estruturais de software que indicam como os softwares com os quais temos que lidar são estruturados.*

Abstract. *Keeping the software systems working well and with high quality is a major challenge. A deep understanding of the software systems can improve the control of change impacts and the management of the software maintenance process. However, to achieve this goal, resources are needed to enable a proper evaluation of the software systems we have deal with. Aiming to contribute to overcome this issue, a previous work of the authors of this paper has defined a model, called Little House, which consists of a macroscopic picture of the object-oriented software structures. Little House represents a software system as a graph with five nodes, called components, which correspond to interconnected sets of classes. The present work aims to characterize qualitatively these components by means of five case studies with Java software systems. The study consisted in the manual inspection of all classes in order to identify their purposes. The analysis reveals how the classes are distributed among the components of Little House, according to their purposes. These results identify patterns of structural features of the programs, and show how the software systems we have to deal with are structured.*

1. Introdução

A manutenção de software é um dos problemas mais críticos no contexto da Engenharia de Software. Realizar modificações em um software envolve ter domínio das técnicas e ferramentas utilizadas na sua construção, bem como conhecer seus requisitos e seu projeto. Idealmente, ao fazer uma modificação em determinada parte do código, o desenvolvedor deve ter conhecimento exato dos impactos da modificação. Para isso, é necessário que ele tenha uma visão ampla do software alvo da modificação. Mesmo em sistemas de pequeno porte, essa tarefa é difícil. Esses, dentre outros aspectos, tornam a manutenção de software a fase mais cara do ciclo de vida de um sistema.

A compreensão detalhada da estrutura do software pode favorecer um melhor controle dos impactos das modificações realizadas, bem como uma melhor gestão do processo de manutenção. Todavia, para alcançar isso, são necessários recursos que permitam a avaliação apropriada do software objeto da manutenção. Esse problema tem motivado pesquisas que, dentre outros objetivos, buscam prover formas de visualizar softwares [Wettel and Lanza 2007, Sharafi 2011] e caracterizar as estruturas dos softwares desenvolvidos [Louridas et al. 2008, Wen et al. 2009, Yao et al. 2009]. Um trabalho anterior das autoras do presente artigo [Ferreira et al. 2011] define um modelo, denominado Little House, que consiste em uma figura macroscópica genérica das estruturas de software orientado a objetos. Little House modela o software como um grafo com cinco vértices, denominados componentes, que correspondem a conjuntos de classes conectados entre si. Esse modelo é proposto com o objetivo de ser utilizado para melhorar tarefas como manutenção e teste de software. Entretanto, o modelo precisa ser profundamente investigado para que se possa identificar características dos seus componentes e determinar suas implicações. O presente trabalho tem por objetivo caracterizar qualitativamente o modelo Little House. Para isso, foram realizados cinco estudos de caso com softwares desenvolvidos em Java. O estudo consistiu em inspecionar manualmente todas as classes de cada software para identificar o propósito de cada uma delas. O objetivo dos estudos de caso é observar como as classes são distribuídas entre os componentes de Little House em função de seus propósitos. Os resultados deste trabalho identificam padrões de características estruturais dos software que indicam como os softwares com os quais temos que lidar são estruturados.

O restante deste artigo está organizado da seguinte forma: Seção 2 discute trabalhos relacionados e descreve o modelo Little House; Seção 3 descreve a metodologia aplicada ao estudo; Seção 4 apresenta os estudos de caso; Seção 5 discute os resultados observados; Seção 6 traz as conclusões e indicações de trabalhos futuros.

2. Trabalhos Relacionados

Identificar recursos e ferramentas que propiciem uma melhor compreensão das estruturas dos softwares desenvolvidos tem sido motivação para muitos trabalhos. Com o objetivo de formar um modelo mental para as estruturas de software, muitas metáforas têm sido propostas. Wettel e Lanza (2007), por exemplo, definem uma abordagem 3D de visualização de software que representa um software como uma cidade. O objetivo dessa abordagem é mostrar uma imagem do software na qual os prédios são as classes, e os pacotes são os quarteirões; todavia não se identificou como essa abordagem pode ser utilizada na avaliação das estruturas de software. Grafos direcionados parecem ser a abordagem mais

comumente aplicada para visualização de software [Sharafi 2011]. Aplicando essa abordagem em um software orientado a objetos, as classes são representadas pelos vértices, e os relacionamentos entre as classes, pelas arestas. Por exemplo, se uma classe *A* usa uma classe *B*, há uma aresta de *A* para *B*. Uma dificuldade relacionada a essa abordagem é que desenhar grafos é uma tarefa complexa, pois os vértices e arestas devem ser exibidos de tal forma que a estrutura do software possa ser visualizada com clareza. Esse problema torna-se ainda mais difícil em sistemas de grande porte.

Outros trabalhos têm explorado as estruturas dos softwares desenvolvidos à luz de conceitos de Redes Complexas [Newman 2003]. Um achado comum desses trabalhos [Baxter et al. 2006, Louridas et al. 2008] é que as estruturas dos softwares têm características de redes de escala livre (*scale-free networks*). A principal característica dessas redes é que uma pequena parcela dos vértices tem grau elevado, enquanto a maior parte dos vértices têm grau pequeno. O grau de um vértice corresponde ao número de arestas conectadas a ele. Wen et al. (2009) identificaram características de redes de escala livre em oito programas Java. O trabalho de Yao et. al. (2009) concluiu que essas características tornam-se mais evidentes à medida que o software cresce. Esses trabalhos identificam características importantes da formação das estruturas de software, entretanto não são suficientes para prover uma compreensão satisfatória dessas estruturas. Com o objetivo de avançar neste contexto, Ferreira et. al (2011) realizaram um estudo sobre evolução de software e também identificaram características de redes de escala livre em software. Um dos resultados do trabalho de Ferreira et al. (2011) foi a definição de um modelo denominado Little House para representar a topologia genérica de software orientado a objetos. Esse modelo é uma adaptação do modelo Bow-Tie [Broder et al. 2000], que é usado para representar a topologia do grafo formado pelas páginas da Web. Ferreira et al. (2009) definiram o modelo Little House e verificaram que ele é aplicável a todos os softwares analisados no estudo realizado por eles. Entretanto, as características e implicações do modelo precisam ser investigadas. Por exemplo, é preciso identificar como as classes de um software são distribuídas entre os componentes de Little House. Como o objetivo de detalhar as características do modelo Little House, o presente trabalho visa identificar como as classes de um software são distribuídas entre os componentes de Little House em função de seus respectivos propósitos.

2.1. O Modelo Little House

O modelo Little House [Ferreira et al. 2011] é um grafo no qual um vértice corresponde a um grupo específico de classes. O modelo é mostrado na Figura 1. Cada vértice do modelo é denominado componente. Em cada componente, as classes são conectadas entre si livremente. Little House possui os seguintes componentes:

- LSCC (*largerst strongly connected component*): é o componente fortemente conectado do software. Em *LSCC*, uma classe é alcançável a partir de qualquer outra classe de *LSCC*. Com isso, esse componente pode ser difícil de ser entendido, testado e modificado.
- In: classes desse componente podem usar classes de qualquer componente, exceto *Disconnected*. Entretanto elas não são usadas por classes de outros componentes.
- Out: classes desse componente podem ser usadas por classes de qualquer outro componente, exceto *Disconnected*. Entretanto elas usam somente classes de *Out*.

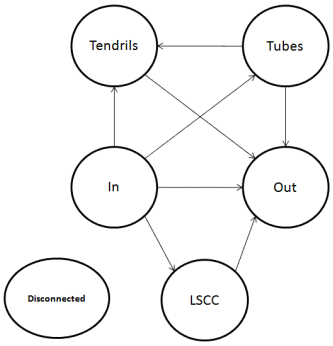


Figura 1. *Little House* – um modelo genérico para as estruturas de software

- Tendrils: classes de *Tendrils* usam apenas classes de *Out* e podem ser usadas por classes de *Tendrils*, *Tubes* ou *In*.
- Tubes: classes de *Tubes* usam apenas classes desse componente, de *Out* ou de *Tendrils*. Além disso, elas podem ser usadas apenas por classes de *Tubes* ou *In*.
- Disconnected: uma classe neste componente não possui conexões com classes de outros componentes.

O modelo Little House é uma visão macroscópica da estrutura de software. Pretende-se que com a aplicação desse modelo a compreensão das estruturas de um software seja facilitada. Todavia, para isso, é necessário prover detalhamento sobre a forma como cada um de seus componentes é constituído. Este trabalho apresenta os resultados de cinco estudos de caso realizados com esse objetivo.

3. Metodologia

O seguintes softwares desenvolvidos em Java foram analisados neste trabalho:

1. JHotDraw (versão 5.2)[1]: é um *framework* customizável com interface gráfica de usuário que permite o desenvolvimento de aplicações de desenho gráfico. A equipe original de seu desenvolvimento contava com Erich Gamma e sua implementação utiliza fortemente padrões de projeto. Muitos trabalhos têm indicado que a qualidade estrutural de JHotDraw é alta [Seng et al. 2006, Czibula and Czibula 2008, Jancke 2010, Kessentini et al. 2010]. Por esta razão, este software foi selecionado para ser analisado neste estudo. Considera-se que a sua análise pode indicar como as classes de um software bem estruturado são distribuídas entre os componentes de Little House.
2. JUnit (versão 4.8.1)[2]: é uma biblioteca amplamente conhecida para a implementação de testes de software Java. Por ser um software intensamente utilizado na indústria, pressupõe-se que sua qualidade estrutural tenha uma qualidade tal que permita seu uso com grande sucesso. Desta forma, a análise de sua estrutura pode indicar como as classes de bibliotecas bem estruturadas são distribuídas entre os componentes de Little House.

[1]http://www.jhotdraw.org/
[2]http://www.junit.org/

3. DBUnit (versão 2.0): é uma extensão do JUnit para automatizar manipulação de banco de dados durante as operações de teste. Tem como objetivo, por exemplo, evitar possíveis danos causados aos dados armazenados durante a realização dos testes. Como o JUnit também é analisado neste estudo, a análise do DBUnit permitirá verificar como a distribuição de classes no modelo Little House é afetada quando um software é estendido.
4. Hotel: é um software desenvolvido por um aluno de graduação em Engenharia de Computação como trabalho prático realizado em uma disciplina de Programação Orientada a Objetos. O software tem por objetivo gerenciar processos em um hotel, tais como registro de acomodações, hóspedes, reservas e hospedagens. O programa foi desenvolvido de acordo com o padrão MVC. A avaliação qualitativa prévia do programa mostra que ele é bem construído e aplica boas práticas de projeto de software.
5. Connecta [Ferreira 2006]: é uma ferramenta acadêmica que realiza medições em software Java. A ferramenta coleta métricas de software, além de gerar um arquivo em formato texto contendo informações sobre o grafo que representa o software analisado. *Connecta* foi selecionada para ser um dos estudos de caso por ter sido desenvolvida por uma das autoras do presente trabalho. O fato de sua estrutura ser muito bem conhecida pelas autoras favorece uma melhor análise do padrão de distribuição de suas classes entre os componentes de Little House.

O trabalho baseia-se na inspeção manual de todas as classes de cada software. Desta forma, o tamanho do software foi um critério relevante na seleção das versões dos softwares analisados. Por esta razão, as versões de JHotDraw, JUnit e DBUnit não são as mais recentes. Foram analisadas as versões cujos tamanhos viabilizassem a inspeção manual das classes.

A coleta dos dados foi realizada de acordo com o seguinte método. Primeiramente, os dados do grafo que representa o software foram coletados pela ferramenta *Connecta*. O aplicativo *Pajek*[3] foi usado para desenhar o grafo que corresponde ao modelo Little House, tendo como entrada os arquivos gerados por *Connecta*. *Pajek* não automatiza o desenho de Little House, ele apenas particiona o grafo conforme o modelo Bow-Tie, no qual o modelo Little House é baseado. O desenho gerado pela ferramenta é, então, manipulado manualmente para obter-se a figura do software conforme o modelo Little House.

3.1. Classificação das Funcionalidades das Classes

As classes de cada software foram inspecionadas manualmente com o objetivo de se identificar o papel de cada uma delas. Conforme a análise feita do código-fonte dos softwares utilizados no estudo, pode-se classificar as funcionalidades das classes nos seguintes grupos principais:

- Interface com Usuário: são as classes responsáveis por manter a interação com usuário, seja por meio de interface gráfica ou de interface do tipo caractere.
- Interface: corresponde ao recurso de *interface* da linguagem Java. Esse recurso é utilizado em Java para estabelecer os serviços que determinadas classes devem implementar.

[3]http://vlado.fmf.uni-lj.si/pub/networks/pajek/

- Classe Abstrata: esse tipo de classe consiste em uma estrutura que serve de modelo para uma classe concreta, não sendo possível instanciar objetos a partir dela. Esta classe permite explorar o recurso de herança em Java, constituindo superclasses das quais as subclasses herdam a implementação.
- Classe de Exceção: a denominação "classe de exceção" foi dada a toda classe implementada para representar uma exceção no programa. Em Java, essas classes são herdeiras da superclasse Exception.
- Classe para Armazenamento de Informações: são classes do tipo estruturas de dados, que possuem apenas atributos e métodos de acesso a eles.
- Classe para Manipulação de Informações: implementam os objetos elementares do domínio do problema do software. Tanto essas classes quanto as de armazenamento de informações são consideradas classes de objetos de negócio.
- Classe para Controle de Informações: são classes que implementam principalmente as regras de negócio do programa.
- Classe para Consulta a Banco de Dados: classes deste tipo permitem utilizar recursos do pacote java.sql para inserir dados ou realizar consultas em banco de dados no padrão SQL (*Structured Query Language*).
- Classe para Manipulação de Arquivos: classes que realizam operação de escrita ou de leitura em arquivo.
- Classe para Entrada e Saída de Dados: esse tipo de classe possibilita empregar recursos do pacote java.io, controlando as informações que são veiculadas pelos dispositivos de entrada e direcionando os dados apropriados aos periféricos de saída. Esse tipo de classe foi encontrado somente do software JHotDraw, para manipulação de dados provenientes do *mouse*.
- Classe de Serviços: consiste na classe que apresenta métodos estáticos e públicos para utilização por várias classes no programa.
- Classe Depositária: a denominação "classe depositária" foi dada a toda classe que possui somente atributos que atuam como constantes no programa.

4. Estudos de Caso

Esta seção descreve os resultados dos estudos de caso dos cinco software analisados. Os dados das distribuições das classes entre os componentes de Little House para cada software são mostrados nas Tabelas 1, 2, 3, 4 e 5.

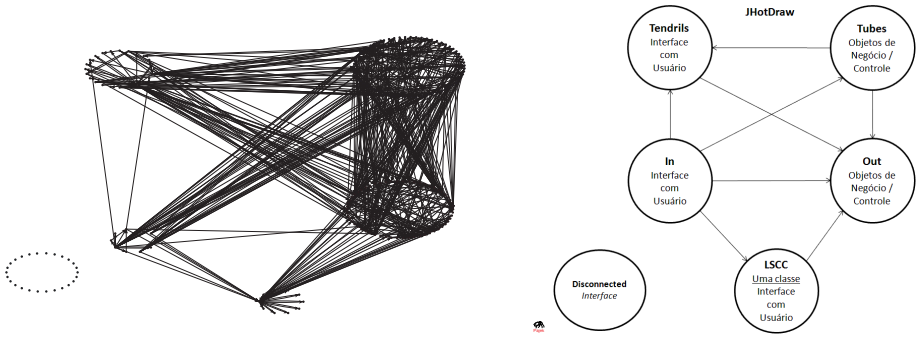
4.1. JHotDraw

O componente *Disconnected* do JHotDraw apresenta predominantemente classes de interface, embora duas classes distintas, uma de armazenamento de informações e outra de controle, também estejam presentes. Esse resultado deve-se ao fato de que *Connecta* considera somente as conexões entre classes para gerar o grafo. Desta forma, é esperado que as interfaces façam parte deste componente. A mesma característica foi observada nos demais softwares analisados neste trabalho.

Na Figura 2a, que mostra a estrutura de JHotDraw conforme Little House, o componente *LSCC* apresenta dez vértices. Entretanto, esses vértices correspondem aos arquivos de bytecode de uma mesma classe. Desta forma, JHotDraw possui apenas uma classe no componente *LSCC*, que é do tipo interface gráfica de usuário.

Tabela 1. Distribuição das classes de JHotDraw

Propósito da Classe	*IN*	*LSCC*	*OUT*	*TUBES*	*TEND.*	*DISC.*
Interação com Usuário	6	1	3		10	
Interface						23
Classe Abstrata			6	2		
Classe de Exceção						
Armazenamento de Informações			22	25	6	1
Manipulação de Informações			22	27	1	
Controle			2	3		1
Manipulação de Banco de Dados						
Manipulação de Arquivos						
Entrada e Saída de Dados				2	1	
Serviços			2	1	1	
Depositária						



(a) (b)

Figura 2. JHotDraw - (a) Modelado por Little House (b) Distribuição das classes em Little House

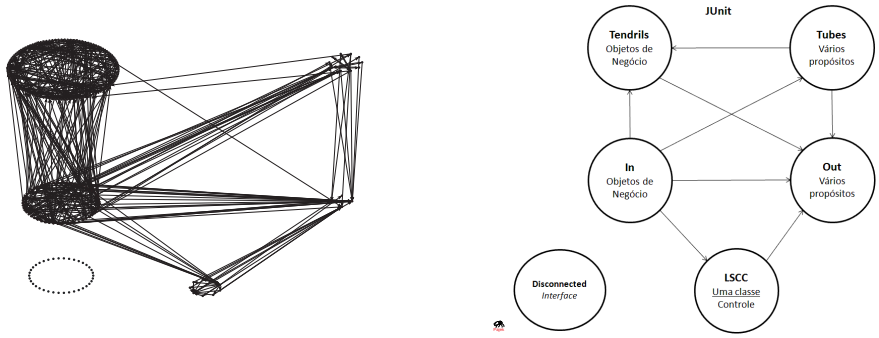
No componente *In*, todas as classes presentes são classes de interface gráfica. Isso indica que o componente *In* caracteriza-se, neste caso, por pertencer à camada de apresentação do software. As funcionalidades das classes do componente *Out* são as seguintes: interface com o usuário, classes abstratas, armazenamento e manipulação de informações, controle e serviços. Entretanto, a maior parte dessas classes são de armazenamento e manipulação de informações, isto é, implementam objetos de negócio.

Tubes também apresenta classes de diferentes propósitos: classe abstrata, armazenamento e manipulação de informações, serviços, controle, e entrada e saída de dados. Assim como *Out*, as classes de *Tubes* também são na maioria classes que implementam objetos de negócio. Em *Tendrils*, há classes de serviço, armazenamento e manipulação de informações, e entrada e saída de dados. Entretanto, a maioria das classes é de interface gráfica de usuário.

Em resumo, a estrutura de JHotDraw está assim distribuída: as classes de interface gráfica concentram-se em *In* e *Tendrils*; as classes que implementam objetos de negócio e as de controle estão localizadas em *Out* e *Tubes*; as classes de serviços são poucas e estão localizadas em *Out*, *Tubes* e *Tendrils*. A Figura 2 mostra JHotDraw modelado por Little

Tabela 2. Distribuição das classes de JUnit

Propósito da Classe	*IN*	*LSCC*	*OUT*	*TUBES*	*TEND.*	*DISC.*
Interação com Usuário Interface						32
Classe Abstrata	3			2	14	
Classe de Exceção			1		17	1
Armazenamento de Informações	7		2	1	18	1
Manipulação de Informações	26			2	34	
Controle	1	1				
Manipulação de Banco de Dados						
Manipulação de Arquivos				1	2	
Entrada e Saída de Dados						
Serviços	2		1	2	8	
Depositária						



(a) (b)

Figura 3. JUnit - (a) Modelado por Little House (b) Distribuição das classes em Little House

House e a distribuição de suas classes entre os componentes de Little House.

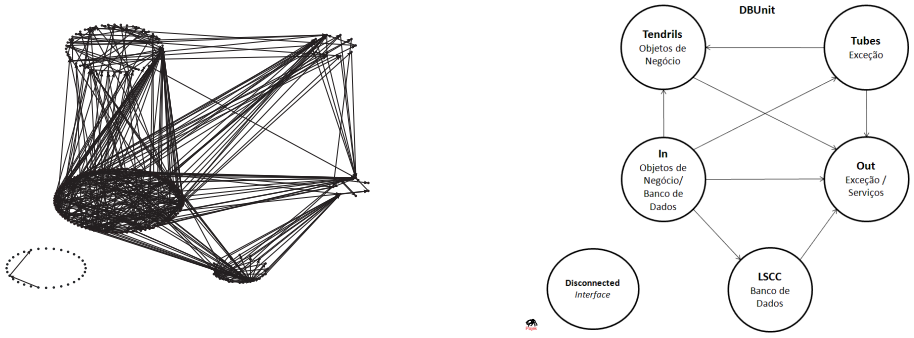
Uma característica muito importante desse software é que *LSCC* possui apenas uma classe. Isso mostra que não há em JHotDraw um componente fortemente conectado. Como a estrutura de JHotDraw tem sido avaliada como boa, esse resultado sugere que um *LSCC* pequeno pode ser um indicador de software bem construído.

4.2. JUnit

O componente *LSCC* do JUnit possui somente uma classe, que é uma classe de controle. A Figura 3a mostra a estrutura de JUnit conforme Little House. Nesta figura, o componente *LSCC* apresenta nove vértices. Entretanto, esses vértices correspondem aos arquivos de bytecode gerados para uma mesma classe. O componente *In* possui classes com diferentes funcionalidades: classes abstratas, armazenamento e manipulação de informações, controle e serviços. Todavia, cerca de 85% das classes são de armazenamento e manipulação de informações. *Out* é um componente pequeno, possuindo somente três classes, sendo uma de exceção, uma de serviços e duas de armazenamento de informações. *Out* é um componente importante em Little House, pois todos os demais

Tabela 3. Distribuição das classes de DBUnit

Propósito da Classe	*IN*	*LSCC*	*OUT*	*TUBES*	*TEND.*	*DISC.*
Interação com Usuário						
Interface					1	20
Classe Abstrata	9	2		1	3	2
Classe de Exceção			4	7	2	1
Armazenamento de Informações	26			2	24	5
Manipulação de Informações	9				12	
Controle						
Manipulação de Banco de Dados	30	14			2	1
Manipulação de Arquivos	4				2	
Entrada e Saída de Dados						
Serviços	6		3		2	2
Depositária						



(a) (b)

Figura 4. DBUnit - (a) Modelado por Little House (b) Distribuição das classes em Little House

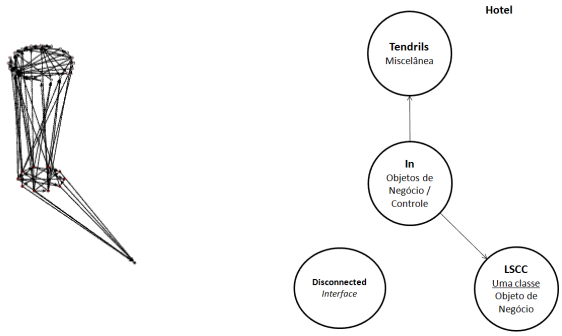
componentes são dependentes dele. O fato de JUnit possuir *Out* pequeno indica que não há nele um grupo grande de classes do qual o restante do software dependa.

Tubes é também um componente pequeno. Possui oito classes igualmente distribuídas com os seguintes propósitos: classe abstrata, armazenamento e manipulação de informação, manipulação de arquivos e serviços. *Tendrils* é o componente que agrega a maior parte das classes. Possui 93 classes, o que corresponde a 52% do software. Possui classes abstratas, classes de exceção, classes de armazenamento e manipulação de informações, manipulação de arquivos e serviços. A maior parcela das classes de *Tendrils*, cerca de 53%, são de armazenamento e manipulação de informações.

As classes de JUnit são assim distribuídas: classes abstratas e de exceção concentram-se em *Tendrils*; a maior parte das classes de objetos de negócio está em *Tendrils*, mas uma grande parte está também em *In*; classes de controle estão em *In* e *LSCC*; classes de manipulação de arquivos e de serviços concentram-se em *Tendrils*. A Figura 3 mostra JUnit modelado por Little House e a distribuição de suas classes entre os componentes de Little House.

Tabela 4. Distribuição das classes de Hotel

Propósito da Classe	*IN*	*LSCC*	*OUT*	*TUBES*	*TEND.*	*DISC.*
Interação com Usuário	1				14	
Interface						
Classe Abstrata					1	
Classe de Exceção						
Armazenamento de Informações	4	1			4	
Manipulação de Informações						
Controle	3				2	
Manipulação de Banco de Dados						
Manipulação de Arquivos						
Entrada e Saída de Dados						
Serviços	1					
Depositária						



(a) (b)

Figura 5. Hotel - (a) Modelado por Little House (b) Distribuição das classes em Little House

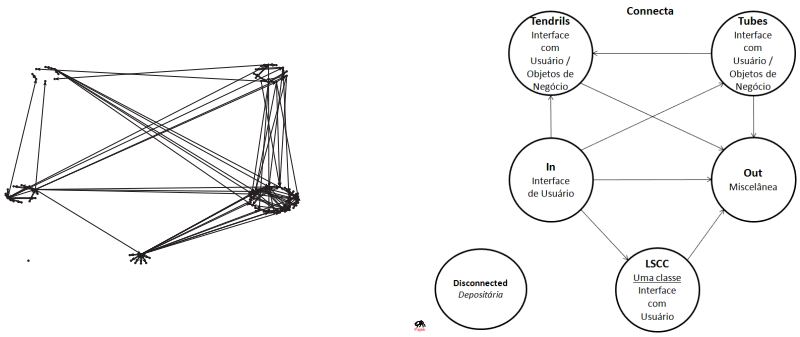
4.3. DBUnit

Em DBUnit, a maior parte das classes do componente *LSCC* é para manipulação de banco de dados. O maior componente é *In*, com 43% das classes do software. Esse componente concentra classes para armazenamento de informações e para manipulação de banco de dados. Entretanto, possui também classe de outros propósitos: serviços, classes abstratas, manipulação de informações e de arquivos. *Out* é um componente pequeno e possui apenas classes de exceção e de serviços. *Tubes* também é pequeno e a maior parte de suas classes são de exceção. *Tendrils* agrega grande parcela das classes do software, sendo que a maior parte delas, que corresponde a cerca de 75% do componente, são para armazenamento e manipulação de informações.

As classes de DBUnit estão assim distribuídas: há classes abstratas em todos os componentes, exceto em *Out*; as classes de exceção estão na maioria em *Tendrils*; classes de armazenamento e manipulação de informações estão distribuídas entre *In* e *Tendrils*; classes de manipulação de banco de dados estão concentradas em *In*, mas uma grande parte delas está em *LSCC*; classes de manipulação de arquivos estão em *In* e *Tendrils*; classes de serviços estão distribuídas entre *In*, *Out*, *Tendrils* e *Disconnected*, sendo que *In*

Tabela 5. Distribuição das classes de Connecta

Propósito da Classe	IN	LSCC	OUT	TUBES	TEND.	*DISC.*
Interação com Usuário	2	1	3	3	3	
Interface						
Classe Abstrata						
Classe de Exceção						
Armazenamento de Informações			4			
Manipulação de Informações			3	3	1	
Controle						
Manipulação de Banco de Dados			5			
Manipulação de Arquivos			1			
Entrada e Saída de Dados						
Serviços						
Depositária						1



(a) (b)

Figura 6. Connecta - (a) Modelado por Little House (b) Distribuição das classes em Little House

é o componente que possui a maior quantidade desse tipo de classe. A Figura 4 mostra DBUnit modelado por Little House e a distribuição de suas classes entre os componentes de Little House.

4.4. Hotel

Este software apresenta apenas três dos componentes de Little House: *LSSC*, *In* e *Tendrils*. No componente *LSCC*, a única classe presente permite o armazenamento de informações, no caso, os dados do objeto do tipo *Cliente*. Em *In*, a maior parte das classes é de armazenamento de informações e de controle. Esse componente possui também uma classe de serviços e uma classe de interação com o usuário, que é a classe principal (main) da aplicação. Em *Tendrils* está a maior parcela das classes do programa, sendo que a maior parte deste componente corresponde a classes de interface com o usuário. *Tendrils* possui também classes de armazenamento de informações, de controle e uma classe abstrata.

Este estudo de caso mostra que o grafo resultante de Little House pode não ter alguns componentes quando o software é muito pequeno. No caso específico desse software, não se implementou persistência de dados. Ou seja, o programa é uma versão preliminar de um software, pois, para ser usado na prática ele necessitaria ser comple-

mentado. Essa análise sugere que a formação completa de Little House possivelmente é resultante da evolução do software. A Figura 5 mostra o software Hotel modelado por Little House e a distribuição de suas classes entre os componentes de Little House.

4.5. Connecta

O componente *Disconnected* de Connecta possui apenas uma classe, que foi classificada como depositária. Esta classe foi definida para ser utilizada em funcionalidades que ainda não foram implementadas em Connecta. Por esta razão, ela não possui qualquer conexão com outras classes do programa.

A Figura 6a mostra a estrutura de Connecta conforme Little House. Nesta figura, o componente *LSCC* apresenta seis vértices. Entretanto, esses vértices correspondem aos seis arquivos de bytecode gerados para uma mesma classe. Desta forma, o *LSCC* de Connecta possui apenas uma classe, que é uma classe de interface com usuário responsável por apresentar os dados resultantes da coleta de métricas na ferramenta.

O componente *In* possui duas classes que são de interface com usuário. Uma é a classe principal, a partir da qual o usuário aciona uma segunda interface, que também faz parte de *In*, para selecionar os arquivos do programa para o qual deseja-se coletar as métricas. *Out* concentra a maior parte das classes do programa. Esse componente possui classes de propósitos variados, que são utilizadas pelas demais classes do software. São classes de interface gráfica, de armazenamento e manipulação de informações, e de manipulação de banco de dados. *Tubes* e *Tendrils* possuem classes de interface de usuário e de manipulação de informações.

As classe de Connecta estão distribuídas da seguinte forma entre os componentes de Little House: classes de interface estão distribuídas entre todos os componentes, exceto *Diconnected*; classes de armazenamento de informações fazem parte de *Out*; classe de manipulação de informação estão em *Out* e em *Tubes*; classe de manipulação de banco de dados e de arquivos estão em *Out*. A Figura 6 mostra Connecta modelado por Little House e a distribuição de suas classes entre os componentes de Little House.

Embora existam pontos a serem melhorados na estrutura de Connecta, avalia-se que a sua qualidade estrutural seja boa. O componente de *LSCC* possui apenas uma classe, o que é um ponto favorável à sua manutenibilidade. Com a análise de Connecta à luz de Little House fica evidente que o modelo pode ser útil para auxiliar a identificar classes centrais no sistema. Por exemplo, as classes de *Out* têm papel central em *Connecta*. Neste componente há classes que implementam objetos elementares manipulados pelo software. A modificação em uma dessas classes certamente acarretaria necessidade de modificação em outras classes do sistema. Todavia, para que o modelo Little House possa ser aplicado na prática para prover análise de estrutura de software, é necessária a criação de ferramentas que permitam a visualização gráfica do software aplicando Little House, bem como o seu detalhamento.

5. Discussão

Pelos estudos de caso realizados não é possível afirmar que exista um padrão genérico de distribuição de classes entre os componentes de Little House, considerando-se os propósitos das classes. Todavia, a análise dos softwares traz importantes conhecimen-

tos sobre as características estruturais dos softwares quando avaliados sob a perspectiva de Little House.

Dois dos estudos de caso foram realizados com softwares cujas estruturas são reconhecidas como de boa qualidade: JHotDraw e JUnit. O software JHotDraw tem sido considerado como exemplo de software bem estruturado em outros estudos. JUnit é uma biblioteca de uso industrial amplamente disseminado. Desta forma, considera-se também que o JUnit tenha um bom nível de qualidade estrutural. Embora os achados desses estudos de caso não possam ser generalizados, a análise deles é importante pois revela características de Little House que podem ser consideradas como referência para a qualidade estrutural de software. Uma característica entre os dois softwares, JHotDraw e JUnit, é a constituição do componente LSCC. Em ambos os softwares, LSCC possui apenas uma classe. Essa é uma característica relevante, pois LSCC é um componente crítico, uma vez que suas classes são fortemente conectadas entre si, o que pode representar dificuldades de manutenção e de compreensão do software. Possuir apenas uma classe em LSCC indica que não há no software um grupo de classes fortemente conectadas entre si.

JHotDraw é um software baseado em interface gráfica com o usuário. Tendo como premissa a alta qualidade de sua estrutura, a forma da distribuição das suas classes entre os componentes de Little House pode ser tomada como parâmetro para softwares semelhantes: os componentes *In* e *Tendrils* agregam predominantemente classes de interface com usuário, enquanto *Tubes* e *Out* concentram as classes de objetos de negócio e de controle. Um raciocínio semelhante pode ser aplicado a JUnit, que é uma biblioteca. Nesse software, os componentes *In* e *Tendrils* agregam predominantemente classes de negócio, enquanto *Tubes* e *Out* possuem classes de propósito diversos: classes abstratas, classes de exceção, classes de objetos de negócio, manipulação de arquivos e serviços.

Com relação à proporção de classes distribuídas entre os componentes de Little House, JUnit e DBUnit têm formações similares. Há também coincidências em relação à distribuição das classes quanto ao propósito delas: classes de exceção concentram-se em *Tendrils*, classes de objetos de negócio concentram-se em *In* e *Tendrils*, e boa parte das classes de manipulação de arquivos estão também em *Tendrils*. A diferença principal se dá em relação às classes de manipulação de banco de dados, que constituem a extensão de DBUnit a partir de JUnit. Em DBUnit, essas classes concentram-se em *In* e *LSCC*. Essas observações sugerem que a configuração de Little House de um software estendido é semelhante à do software original.

O software Hotel foi o único que não apresentou todos os componentes de Little House. Esse programa pode ser considerado como uma versão inicial de um sistema para gerenciamento de hotel, já que não realiza persistência de dados alguma. Os resultados observados nesse estudo de caso sugere que possivelmente a formação de Little House é resultante da evolução do software, ou seja, softwares muito pequenos ou em estágios iniciais têm estruturas simples e podem não ser perfeitamente ajustados a Little House.

A ferramenta Connecta é baseada em interface gráfica com usuário e a distribuição de suas classes em Little House assemelha-se à de JHotDraw. A diferença principal é que o componente *Out* de Connecta agrega classes de propósitos diversos: interface gráfica, de objetos de negócio e de banco de dados. O componente *LSCC* de Connecta também possui somente uma classe. A estrutura da ferramenta é profundamente conhecida pelas

autoras. Sabe-se que, embora algumas melhorias possam ser realizadas, a estrutura do software tem um bom nível de qualidade. Desta forma, esse resultado reforça a ideia de que um *LSCC* pequeno pode ser um indicador de boa qualidade estrutural do software.

6. Conclusões

Realizar manutenções em um software demanda conhecer bem sua estrutura para que se possa, dentre outros aspectos, gerenciar os impactos das modificações e alocar recursos apropriados às atividades de manutenção. Essa tarefa pode ser laboriosa mesmo em softwares pequenos, e pode ser extremamente difícil em softwares de grande porte.

Um trabalho anterior das autoras realizou um estudo de caracterização das estruturas de software e da forma como elas evoluem. Um dos achados do trabalho foi o modelo denominado Little House, que representa a rede formada pelas conexões das classes de um software orientado a objetos como um macro-grafo com cinco componentes, denominados *In*, *LSCC*, *Out*, *Tendrils* e *Tubes*. Cada um desses componentes corresponde a grupos de classes conectadas entre si. Little House fornece uma visão macroscópica da estrutura do software.

O presente trabalho realizou estudos de caso com o objetivo de caracterizar o modelo Little House. Foram analisados cinco softwares Java. As classes dos softwares foram inspecionadas manualmente para se identificar a forma como elas são distribuídas entre os componentes de Little House de acordo com seus propósitos. Embora os estudos de caso não tenham evidenciado a existência de uma distribuição genérica das classes, os seus resultados sugerem características importantes dos componentes de Little House e dos softwares analisados. Em particular os resultados de JHotDraw e JUnit, que são softwares bem conhecidos e de alta qualidade, podem ser tomados como parâmetro de comparação e avaliação de estruturas de softwares similares. Tanto JHotDraw quanto JUnit têm LSCC com apenas uma classe. LSCC é o componente cujas classes são fortemente conectadas entre si. Desta forma, ele é um componente crítico do ponto de vista de manutenção de software, uma vez que uma modificação em uma classe pode gerar impacto nas demais classes de LSCC. O resultado da análise desses dois softwares sugere, então, que um LSCC pequeno pode ser considerado como um indicador de software bem estruturado. Os resultados dos estudos de caso também mostram que software muito pequeno ou em estágio inicial de desenvolvimento têm estruturas simples e podem não ser modelados perfeitamente por Little House. Isso indica que a formação de Little House possivelmente é resultante da evolução do software.

Outros trabalhos de caracterização do modelo estão em andamento, por exemplo a análise da distribuição dos valores de métricas entre os componentes, bem como a evolução da qualidade das classes dos componentes do modelo. Pretende-se que o modelo Little House possa ser aplicado na avaliação da estrutura de softwares. Por fornecer uma visão macroscópica da estrutura do software, o modelo poderia ser usado como uma espécie de "Raio X" do software. Entretanto, para que Little House possa ser aplicado na prática, é necessário construir ferramentas apropriadas para o uso do modelo para, por exemplo, permitir a visualização gráfica do software aplicando-se Little House, bem como o detalhamento das informações de cada componente do software e de suas classes.

Referências

Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., and Tempero, E. (2006). Undertanding the shape of Java software. In *OOPSLA'06*, Oregon, Portland, USA.

Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., and Wiener, J. (2000). Graph structure in the web. In *WWW9 Conference*, pages 309–320.

Czibula, I. G. and Czibula, G. (2008). Clustering based automatic refactorings identification. In *Proceedings of the 2008 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 253–256, Washington, DC, USA. IEEE Computer Society.

Ferreira, K. A. M. (2006). *Avaliação de Conectividade em Sistemas Orientados por Objetos*. Master Thesis - Federal University of Minas Gerais. Belo Horizonte, Brazil.

Ferreira, K. A. M., Bigonha, M. A., Bigonha, R. S., and Gomes, B. M. (2011). Software evolution characterization - a complex network approach. In *X Brazilian Simposium on Software Quality - SBQS'2011*, Curitiba, Paraná, Brazil.

Jancke, S. (2010). *Smell Detection in Context*. Diploma thesis. University of Bonn. Bonn, Germany., http://dirkriehle.com/computer-science/research/dissertation/.

Kessentini, M., Vaucher, S., and Sahraoui, H. (2010). Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 113–122, New York, NY, USA. ACM.

Louridas, P., Spinellis, D., and Vlachos, V. (2008). Power laws in software. 18(1).

Newman, M. E. J. (2003). The structure and function of complex networks. In *SIAM Reviews*, volume 45, pages 167–256.

Seng, O., Stammel, J., and Burkhart, D. (2006). Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1909–1916, New York, NY, USA. ACM.

Sharafi, Z. (2011). A systematic analysis of software architecture visualization techniques. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 254–257, Washington, DC, USA. IEEE Computer Society.

Wen, L., Dromey, R. G., and Kirk, D. (2009). Software engineering and scale-free networks. *Trans. Sys. Man Cyber. Part B*, 39:648–657.

Wettel, R. and Lanza, M. (2007). Visualizing software systems as cities. In *In Proc. of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99. Society Press.

Yao, Y., Huang, S., Ren, Z.-p., and Liu, X.-m. (2009). Scale-free property in large scale object-oriented software and its significance on software engineering. In *Proceedings of the 2009 Second International Conference on Information and Computing Science - Volume 03*, pages 401–404, Washington, DC, USA. IEEE Computer Society.