



Universidade Federal de Pernambuco
Centro de Informática
Graduação em Ciência da Computação

Swicity: Visualizando Sistemas de *Software* em Swift como cidades

Rafael Nunes Galdino da Silveira

Trabalho de Graduação

Recife, dezembro de 2016

Universidade Federal de Pernambuco
Centro de Informática
Graduação em Ciência da Computação

Swicity: Visualizando Sistemas de *Software* em Swift como cidades

*Trabalho apresentado ao Programa de Graduação em
Ciência da Computação do Centro de Informática da
Universidade Federal de Pernambuco como
requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação.*

*Aluno: Rafael Nunes Galdino da Silveira
(rngs@cin.ufpe.br)*

*Orientador: Fernando José Castor de Lima Filho
(fjclf@cin.ufpe.br)*

Recife, dezembro de 2016

Agradecimentos

Eu dedico essa seção a todas as pessoas que tornaram, de alguma maneira, a experiência dos últimos 5 anos única e excepcional. As sensações de gratidão, de dever cumprido e de saudades antecipadas, são ao mesmo tempo angustiantes e confortadoras.

Gostaria de agradecer primeiramente aos meus amados familiares, meu pai, minha mãe e meu irmão. Meus pais sempre me estimularam, desde pequeno, a estudar, a ser uma pessoa esforçada, a lutar por meus sonhos e objetivos. Entretanto, nunca deixando isso enfraquecer meu caráter como cidadão, nem deixando de lado o respeito com os outros, colocando-se sempre no lugar de quem precisa, enfim, de me formar como pessoa. Agradeço todo o apoio deles nos momentos de dificuldade e apesar de muitas vezes não compreenderem o que eu realmente estudo, acreditam fielmente que sou capaz de vencer quaisquer dificuldades, sejam elas desde as questões das listas de algoritmos até a escrita deste trabalho de conclusão. Ao meu irmão eu agradeço por sempre estar do meu lado e por xingar o *design* dos meus trabalhos, fazendo com que eu sempre melhorasse.

Agradeço também à minha namorada, Camila, que acompanhou toda a luta em busca de um tema e posteriormente na execução deste trabalho. Sempre serena e paciente comigo, se fez calma para poder me acalmar. Revisou este documento e apontou todas as evidências da minha dislexia, não diagnosticada. Que não se importava em passar horas ao meu lado, vendo-me escrever e programar, que sempre foi proativa em colaborar com dicas, ideias e soluções simples para minhas ansiedades, dificuldades e problemas durante o desenvolvimento do TCC. Agradeço pelo seu amor e carinho que a cada "oi" dado, a cada ligação feita, mensagem enviada, encontro vivido, a cada dia passado junto, renovavam as minhas energias e me proviam força para seguir em frente. Você me inspira e me faz querer ser cada vez melhor, para que se orgulhe de mim!

Também agradeço ao Professor Fernando Castor por ser um excelente professor, desde a matéria de PLC. Nesse momento, em especial, por me orientar e acolher no trabalho, sempre disposto a ajudar e tirar dúvidas. Tornou este trabalho possível. Também agradeço a Marcel Rebouças que foi fundamental na pesquisa, junto com Castor. Tanto na concepção do projeto, como fornecendo o suporte nos meus momentos de dúvidas, colaborando com fontes a serem estudadas e os dados do trabalho.

Agradeço a todos os meus amigos, que sempre me fizeram sentir confortável, bem vindo e amado. Amigos que compartilharam bons e maus momentos, da greve à noite que viramos para terminar Comunicação. São amigos que me dão orgulho e me inspiram por sua competência e inteligência, por irem trabalhar em grandes empresas lá fora, e que me deixam tristes por sua ausência próxima e agradável. Agradecimento especial a Jesus, que aguentou em muitos projetos o meu "mimimi", sempre cético e brincalhão, também dando dicas e consultas grátis sobre *web development* neste trabalho. A Lucas Tenório que sempre ganhou de mim nas discussões sobre tecnologia, o que me fez aprender bastante. Todos colaboraram de maneira direta ou indireta para que esse trabalho fosse possível.

Agradecimento ao Centro de Informática que proporcionou um ambiente não só de aprendizado, mas de oportunidades de vida, onde pude fazer grandes amizades. É um centro de excelência acadêmica, mas também "pessoal". Sinto-me orgulhoso por ter sido parte dele.

Agradeço por fim, mas não menos importante ao grupo PET-Informática. Um grupo que é de pesquisa, ensino e extensão, mas também de experiências de vida, formação pessoal e profissional. Um grupo que reúne pessoas sensacionais, com qual a gente aprende muito, se sente motivado a sempre estudar, grupo com que a gente se diverte bastante e que me permitiu fazer grandes amizades que levo para a vida! Sentirei muita falta de tudo que se pode viver nesse grupo. Será que não dá pra continuar no PET depois de formado?

Resumo

O esforço gasto por organizações de *software* com a manutenção e retrabalho no desenvolvimento de *software* pode variar, em média, entre 40% e 50% do esforço total do desenvolvimento. Com o crescimento dos sistemas e de suas complexidades, a habilidade natural do ser humano de realizar manutenção fica comprometida. Para mitigar isso, diferentes processos e propostas são aplicadas no contexto de manutenção para se tentar identificar focos de possíveis inspeções de código, como: verificação manual de código, testes, validação, análise estática, entre outros. Visualização de *software*, porém, também é uma área que fornece uma alternativa para se mitigar esse problema. Essa área reúne técnicas e abordagens para a construção de visões claras, objetivas e úteis na análise de sistemas de *software*, seja no contexto de evolução, controle de versão, inspeção de código, entre outros. Existem diversas abordagens para se construir visualizações de sistema e cada uma possui suas vantagens e desvantagens. Swift é uma linguagem nova com notória e crescente adoção pela comunidade de desenvolvedores. Apesar desse fato, poucos estudos e iniciativas são focadas em análise de código Swift. O objetivo deste trabalho, então, é construir uma ferramenta para visualização de sistemas escritos em Swift, baseada na metáfora das cidades (Wettel, 2007). Além disso, esta pesquisa tem por objetivo, também, colaborar com as pesquisas sobre a utilização de Swift realizadas pelo Professor Fernando Castor. Para atingir esses objetivos foram feitos estudos sobre a construção e técnicas de visualização de *software* para entender as dificuldades e necessidades de uma boa visualização. Mas também estudos relacionados ao mapeamento dos elementos da linguagem Swift para o desenvolvimento da metáfora das cidades. Desse modo pôde-se oferecer uma plataforma útil, fácil de se utilizar e que permite explorar sistemas de código Swift em busca de conclusões não óbvias a respeito do mesmo.

Palavras-chave: Engenharia de *Software*, Visualização de *Software*, Manutenção de *Software*, Inspeção de código, Swift, Metáfora das Cidades.

Abstract

The effort spent by software organizations with maintenance and rework in software development can vary, on average, between 40% and 50% of the total development effort. With the growth of systems and their complexities, human's natural ability to perform maintenance is compromised. To mitigate this, different processes and proposals are applied in maintenance context to try to identify targets of possible code inspections, such as: manual code verification, testing, validation, static analysis, and others. Software visualization, however, is also a field that provides an alternative to mitigate this problem. This field assembles techniques and approaches to construct clear, objective and useful visions in the analysis of software systems, whether in the context of evolution, version control, code inspection, and others. There are several approaches to building system visualizations, and each one has its advantages and disadvantages. Swift is a new language with notorious and growing adoption by the developer community. Despite this fact, few studies and initiatives are focused on Swift code analysis. The purpose of this work, then, is to construct a tool for visualization of systems written in Swift, based on the city metaphor (Wettel, 2007). In addition, this research also aims to collaborate with research on the use of Swift carried out by Professor Fernando Castor. To achieve these objectives, studies were done on the construction and visualization techniques of software to understand the difficulties and needs of a good visualization. But also, studies related to the mapping of the elements of the Swift language to the development of the city metaphor. In this way, it was possible to offer a platform that is useful, easy to use and allows to explore Swift code systems in search of non-obvious conclusions about it.

Keywords: Software Engineering, Software Visualization, Software Maintenance, Code Inspection, Swift, City Metaphor.

Sumário

CAPÍTULO 1	2
Introdução	2
CAPÍTULO 2	5
Fundamentos	5
2.1 Visualização de Dados	5
2.2 Visualização de Software	5
2.3 A Metáfora das Cidades	7
2.4 Swift	9
2.4.1 Classes e Structures	11
2.4.2 Protocols	11
2.4.3 Enumerations	12
2.4.4 Extensions	12
CAPÍTULO 3	14
Trabalhos relacionados	14
3.1 Trabalhos em visualização	14
CAPÍTULO 4	18
Desenvolvimento	18
4.1 A Pesquisa e Coleta de Dados	18
4.2 As Tecnologias de Desenvolvimento	19
4.3 O Processo de Desenvolvimento	22
CAPÍTULO 5	25
A ferramenta	25
5.1 Elementos da Metáfora das Cidade em Swift	25
5.2 Funcionalidades, Navegação e Interatividade	28
5.2.1 Buscar e Executar	29
5.2.2 Configurações de Visualização	29
5.2.3 Sistema de Navegação	30
5.3.4 Detalhes do Projeto	31
Seleção e detalhamento dos blocos	31
5.2.5 Screenshot	32
CAPÍTULO 6	33

Casos de estudo	33
6.1 CVCalendar	33
6.2 Alamofire	35
6.3 RxSwift	37
CAPÍTULO 7	39
Conclusão	39
7.1 Contribuições	40
7.2 Trabalhos futuros	40
7.3 Considerações Finais	41
Apêndice A	42
Guia de usabilidade	42
Referências	44

CAPÍTULO 1

Introdução

O processo de desenvolver *software* é complexo e envolve coordenação de equipes de desenvolvimento, com os objetivos internos, também as exigências dos clientes, contando com a limitação de infraestrutura, tempo e recurso humano.

O crescimento da complexidade do código e da complexidade em sistemas de *software* modernos sobrecarregam a capacidade natural dos engenheiros e times de desenvolvimento de compreender, gerenciar e identificar o comportamento do sistema apenas olhando para o código (Waller; Wulf, 2013). Uma pesquisa feita por Lientz e Swanson(1980), ainda na década de 80, apontou que 75% do esforço na manutenção de *software* vem das mudanças do ambiente de desenvolvimento e de novas exigências dos clientes. Além disso, segundo o mesmo trabalho, 21% era relacionado a correção de erro. Problema que tende a abranger cada vez mais pessoas, pois sistemas de *software* já estão em todo lugar, sob demanda contínua de novas aplicações e suporte das antigas.

O esforço gasto por organizações de *software* com retrabalho pode variar, em média, entre 40% e 50% do esforço total do desenvolvimento de um projeto.¹ Observando esses números fica clara a importância que a manutenção e design do *software* no ciclo de desenvolvimento. Por conta disso existem diferentes iniciativas e processos que visam aumentar a qualidade do *software*, como por exemplo: documentação, inspeção, validação, verificação e teste. Aplicadas individualmente ou de maneira combinada tais processos tem como objetivos maiores o monitoramento, rastreamento e manutenção do produto.

Uma das maneiras de explorar o problema é utilizar visualização de dados, área que visa facilitar o entendimento do comportamento, identificar fenômenos e extrair insights do conjunto de dados explorado. No contexto de sistemas e programas, as ferramentas proporcionam uma representação visual do código e dos sistemas, utilizadas com propósito de auxiliar tanto nas fases do desenvolvimento, mas também na manutenção, inspeção, depuração desses códigos e sistemas (Fyock, 1997).

Técnicas de visualização 2D, por exemplo, já são amplamente utilizadas e difundidas como em (Voinea, 2015), (Wilhelm, 2015), (Ducasse, 2015), normalmente arriscadas a sofrer com sobrecarga de informação, sobreposição, granularidade dos objetos analisados, etc.

¹ Kalinowski (2016) Kalinowski, Marcos. "Introdução à Inspeção de Software". , 2016. Disponível em: <<http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-inspecao-de-software/8037>>. Acesso em: 16 jun. 2016.

Também existem técnicas de visualização em 3D, agregando uma dimensão, o que possibilita exibir mais informações, mas também trazendo diferentes problemas que precisam ser mitigados como: navegação, oclusão e comparação de elementos, dentre outros. Utilizar-se dessas ferramentas traz vantagens para os desenvolvedores, resultando em uma maior qualidade do produto de *software*, melhor rendimento do time e redução do custo do projeto, como apontado por Fyock (1997).

Wettel em seu trabalho propõe uma visualização de *software* 3D seguindo uma nova abordagem que ele chama de "metáfora das cidades", onde os elementos do código são mapeados e visualizados no contexto de vizinhança, blocos, prédios (Wettel, 2007). Tal metáfora já está sendo utilizada e adotada para o estudo e visualização de sistemas e aspectos de código como pode ser visto em (Viana, 2015), (Waller, 2013).

O foco da análise deste trabalho será em sistemas de *software* em Swift². Swift vem em adoção crescente e teve sua primeira versão lançada em 2014. É uma linguagem de programação multiparadigma desenvolvida pela Apple para substituir Objective-C como a linguagem de desenvolvimento para suas plataformas iOS, OS X, watchOS, tvOS, além de ter também uma versão disponível para o sistema operacional Linux.

Apesar da visualização de software ser uma ótima ferramenta para o entendimento do desenvolvimento e evolução do código, pouco disso foi aplicado e explorado no contexto da análise de código Swift, em particular, e no ecossistema da Apple, em geral. Sendo assim, é interessante explorar uma abordagem de visualização tal como a "metáfora de cidades" pondo Swift como contexto de estudo. Por conta da crescente adoção da tecnologia e da ausência de trabalhos em visualização de *software* para Swift, mas também devido às peculiaridades da linguagem como o fato dela ser multiparadigma. Por conter construções de tipos mais flexíveis onde, por exemplo, um enumerador pode conter propriedades e métodos, tipos podem estender outros tipos, dentre outros aspectos ainda não explorados em visualização e utilizando a metáfora proposta.

Os objetivos deste trabalho são:

- Fornecer uma plataforma que possibilite uma visão diferenciada do código, podendo gerar *insights* e observações das consequências práticas, comportamentais e teóricas da implementação de sistemas em Swift.
- Sendo assim será desenvolvida uma ferramenta de visualização de software 3D que mapeie os elementos de Swift para a metáfora de cidades (Wettel, 2007).
- Esta ferramenta visa também colaborar com as pesquisas do aluno de mestrado Marcel Rebouças e do seu mentor professor Fernando Castor. Os dados dos sistemas que serão analisados virão como produto de trabalho de Marcel, que desenvolve uma ferramenta para extração de métricas de sistemas Swift.

² <https://swift.org>

Será necessário, então, estudar as peculiaridades dos elementos e código de Swift para fazer um mapeamento dos mesmos para o conceito da metáfora das cidades, estudar também os aspectos relevantes e interessantes para o contexto da visualização e estudar uma maneira de implementá-los em uma tecnologia que torne a plataforma acessível, interessante e útil.

Este documento é dividido em capítulos e está estruturado da seguinte forma. No capítulo 2 são explorados os fundamentos necessários das áreas envolvidas na pesquisa, assim como conceitos gerais tais como: visualização de *software* e Swift e seus elementos. No capítulo 3 são apresentados trabalhos relacionados, que também fazem uso do conceito da metáfora das cidades, aplicados a temas relacionados a engenharia de sistemas. No capítulo 4 descreve-se o método de desenvolvimento da ferramenta, desde a pesquisa, utilização dos dados, tecnologias escolhidas, etc. No capítulo 5 encontra-se a descrição do ciclo de desenvolvimento, as decisões tomadas tanto de implementação, como decisões do mapeamento entre conceitos da metáfora para Swift, escolhas no modo de visualização e do projeto, e também uma explicação do funcionamento da ferramenta. No capítulo 6 faz-se uma seleção de projetos para serem utilizados como Estudo de Caso da ferramenta. O capítulo 7 apresenta as contribuições realizadas por este trabalho, conclusões obtidas do estudo e os possíveis trabalhos futuros. Por fim os apêndices e as referências.

CAPÍTULO 2

Fundamentos

Neste capítulo serão apresentados fundamentos importantes para o entendimento do trabalho. Primeiramente conceitos gerais de visualização de dados e visualização de *software*. Depois o conceito da metáfora das cidades, modelo utilizado para produção da visualização deste trabalho, e depois conceitos de Swift que são importantes para o entendimento do mapeamento e construção da visualização para a linguagem.

2.1 Visualização de Dados

Na era da informação as pessoas se deparam com uma quantidade massiva de dados que sobrecarrega muitas vezes a capacidade humana de absorção e processamento. É um desafio moderno trabalhar com todos os dados que são produzidos diariamente, pois hoje não só os humanos produzem dados, mas programas coletam dados do ambiente, assim como programas geram dados a partir de dados e programas são dados.

Visualização de dados é um campo de estudo que visa permitir a análise de dados quando as pessoas não conhecem exatamente, a priori, as perguntas que precisam ser feitas a respeito dos dados (Tamara 2014). Dessa forma, visualização de dados estuda um conjunto de técnicas, práticas e modelagem da informação para análise dos dados.

Entretanto, é necessário que as visualizações sejam construídas de maneira honesta. Scott Murray em seu livro *Interactive Data Visualization for the Web-O'Reilly Media 2013* fala que é claro que visualizações, assim como palavras, podem ser usadas para mentir e distorcer a verdade. Porém, quando praticadas honestamente e com cuidado, o processo de visualização pode nos ajudar a ver o mundo de uma nova maneira, revelando padrões inesperados e tendências que estão escondidas nas informações que nos cercam. Ou seja, nos ajuda a compreender a disposição dos dados no mundo, identificar padrões e comportamento e extrair *insights* não triviais.

2.2 Visualização de *Software*

Price (93, 98) diz que visualização de *software* é o uso da construção de tipografias, design gráfico, animações e cinematografia com uma moderna interação humano computador e tecnologias de computação gráfica para facilitar tanto a compreensão humana quanto a efetividade do uso do sistema de *software*.

Pode-se entender, então, que visualização de *software* analisa informações relacionadas a sistemas e programas de *software*, ou seja, tendo como objeto de estudo desde a arquitetura, como o próprio código fonte, métricas de execução, processo de desenvolvimento, evolução do sistema etc. Tudo isso facilitado por sistemas gráficos de representação.

A área pode ser dividida em algumas categorias de estudo³, como por exemplo:

- Visualização de Sistema de *Software*
 - Visualização de Estruturas de *Software*
 - Visualização de código fonte
- Visualização de dados do *Software*
 - Visualização do rastreamento dos dados de execução
 - Visualização de controle de versão de código
- Visualização de Algoritmos
- Linguagens de programação visual

Existem diversas iniciativas que objetivam explorar esses segmentos da visualização de *software* como, por exemplo, uma ferramenta para análise de fluxo de dados em programas feitos em linguagens funcionais (Weck, 2016), como também ferramentas de visualização do foco de desenvolvimento do desenvolvedor no código, StarGate (Ogawa, 2008).

Beyer (2006), por exemplo, explora a evolução do código do sistema através do que ele chamada de *storyboards*⁴ evolutivos. Ele apresenta um grafo evolutivo do código e documentação, onde a visão exibe *clusters*⁵ do código fonte e documentação que evoluem com o tempo, da esquerda para a direita, como apresentado na Figura 1 abaixo.

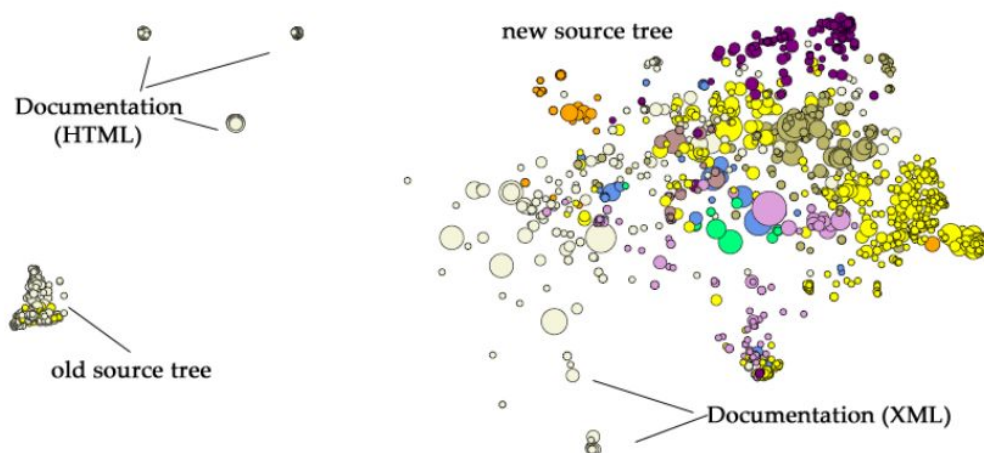


Figura 1. Storyboard aplicado ao projeto ArgoUML, mostrando o antigo cluster (esquerda) e o novo cluster (direita) de arquivos de código fonte e documentação. Fonte: Beyer 2006.

³ <http://www.cs.kent.edu/~jmaletic/cs63903/Lecture-%20Soft%20Vis.pdf>

⁴ <https://en.wikipedia.org/wiki/Storyboard>

⁵ en.wikipedia.org/wiki/Cluster_diagram

De maneira similar Collberg (2003) descreve a GEVOL, um sistema para análise da evolução de *software* baseado em grafos. Em seu sistema, Figura 2, ele fornece uma visão evolutiva de alguns aspectos do código como: gramática⁶, grafo de herança do sistema⁷ e grafo de chamadas⁸.

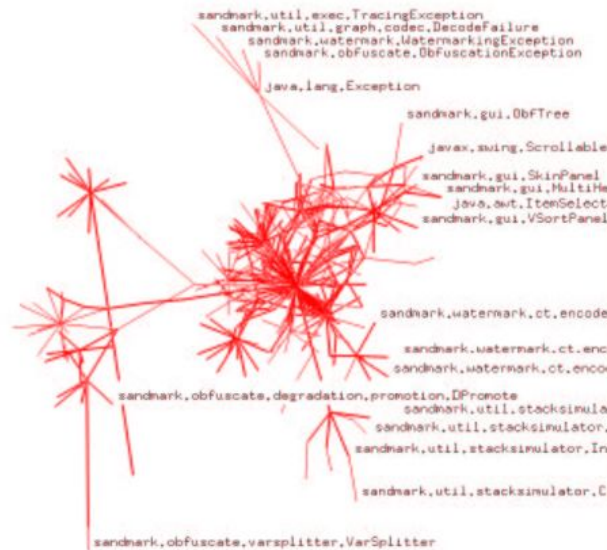


Figura 2. Snapshot do GEVOL visualizando o grafo de herança do sistema SandMark's com algumas classes nomeadas na visualização. Fonte: Collber 2003.

Ambos fornecem uma abordagem para visualização evolutiva do código do sistema baseada em grafos. Existem, porém, outras técnicas e estudos para tentar fornecer uma melhor visualização de *software* como a metáfora baseada em ondas, EVOWAVE (Magnavita, 2015), como também o YARN (Hindle, 2007) e outros.

2.3 A Metáfora das Cidades

A visualização escolhida para o desenvolvimento deste trabalho é conhecida como a metáfora das cidades, introduzida por Wettel (2007). Como o próprio nome diz, a metáfora tenta fornecer uma atmosfera de cidade como representação do código. O objetivo é fornecer ao usuário um senso de localidade para a fácil compreensão do programa.

A metáfora mapeia conceitos do código e do sistema para conceitos estudados de *layouts*, topologias e navegação, que melhor exaltem a atmosfera urbana.

⁶ https://en.wikipedia.org/wiki/Context-free_grammar

⁷ [https://en.wikipedia.org/wiki/Inheritance_\(object-oriented_programming\)](https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming))

⁸ https://en.wikipedia.org/wiki/Call_graph

Muitas ferramentas de visualização 3D falham em comunicar informações relevantes do sistema e em fornecer suporte ao programador a ter compreensão do mesmo. Uma razão pela qual isso acontece é que acaba-se por explorar demais a terceira dimensão, adicionando muita complexidade a visão que leva rapidamente ao sobrecarregamento de informação. A metáfora da cidade por sua vez tenta explorar de forma simples as métricas, mapeando-as para elementos básicos do 3D.

A visão tem foco em sistemas orientados a objeto⁹, onde cada bloco na cidade que tomará forma de um prédio ou construção é representado por uma classe do sistema. Essa visão explora duas métricas simples para sua execução, a primeira é chamada de NOA, acrônimo para *Number of Attributes*, e NOM, acrônimo para *Number of Methods*.

A NOA é utilizada para definir a base do bloco que representa um prédio da cidade. Já o NOM é utilizado para definir a altura do mesmo bloco, a razão para isso é que como grandes prédios são normalmente relacionados a grandes empresas e negócios o número de métodos diria quanta funcionalidade e importância o bloco tem. Logo uma classe com muitos métodos, mas com poucos atributos seria representada por uma alta e fina construção. Assim, como no caso complementar uma classe com muitos atributos e poucos métodos, seria representado por uma plataforma ou um estacionamento. Isso para dar uma noção de extremos ao sistema, prover impressão de magnitude e também da distribuição do mesmo.

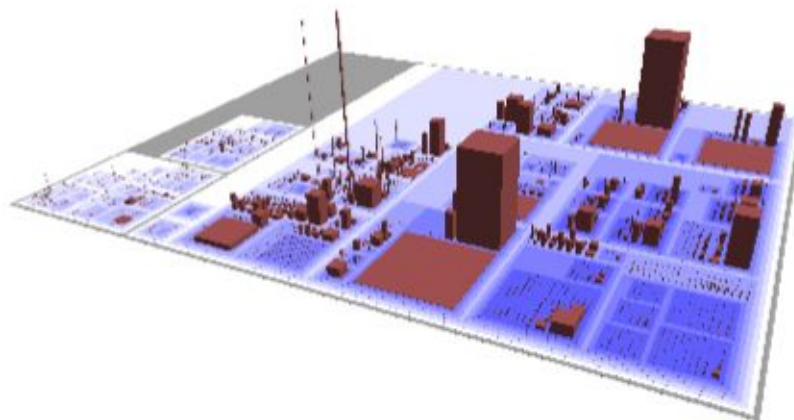


Figura 3. Metáfora das cidades representando a cidade do sistema ArgoUML. Fonte: Wettel 2007.

A Figura 3 mostra um exemplo da metáfora aplicado ao sistema ArgoUML. Wettel, em seu trabalho, também introduz conceitos de normalização dos dados para uma melhor representação visual, bem como fundamenta conceitos de topologia e *layout* para representação da cidade. Além disso, ele apresenta um conjunto de funcionalidades relacionadas a interatividade e navegação como zoom, seleção, filtros, *tags*, entre outros. O intuito é garantir a experiência completa de imersão na região urbana e não perder o foco de análise.

⁹ https://en.wikipedia.org/wiki/Object-oriented_programming

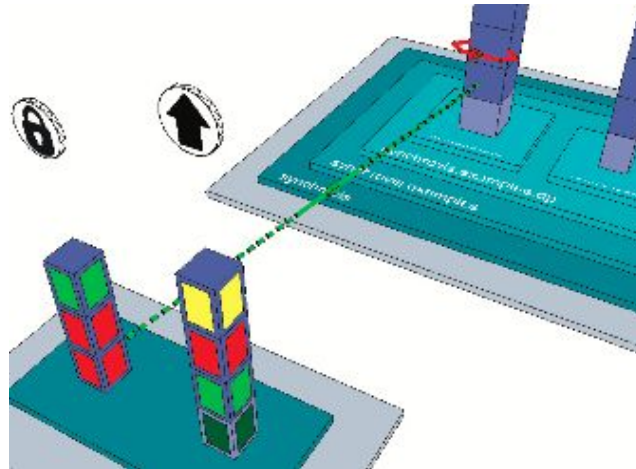


Figura 4. Visualizando a execução do problema do jantar dos filósofos. Fonte: Waller 2014.

Esse conceito já tem adeptos e trabalhos inspirados em seus conceitos. Como na Figura 4, onde Waller (2013) apresenta a ferramenta SynchroVis, uma ferramenta para monitoramento e rastreamento para análise de concorrência baseado na metáfora das cidades. Tal sistema fornece um entendimento de ambas propriedades estáticas e dinâmicas do sistema com o foco em exibir o comportamento concorrente. O trabalho de Viana (2015) por sua vez é uma adaptação da metáfora para sistemas em *Javascript*¹⁰, linguagem interpretada popular no desenvolvimento de aplicações web.

Por ser uma metodologia de construção de visualização diferente, simples e interessante, a metáfora da cidade foi escolhida como objetivo de implementação deste trabalho.

2.4 Swift

Este trabalho propõe a análise de sistemas Swift, a qual teve sua primeira versão lançada em 2014 e é uma linguagem de programação multiparadigma, pois orquestra diferentes paradigmas como funcional, orientado a protocolo, orientado a objetos e imperativo. Swift, atualmente na versão 2.2.1, vem sendo desenvolvida pela Apple para substituir Objective-C¹¹ como a linguagem de desenvolvimento para suas plataformas iOS, OS X, watchOS, tvOS, além de ter também uma versão disponível para o sistema operacional Linux.

Apesar de nova, Swift tem uma curva de adoção bastante significativa segundo o GitHub¹² e segundo pesquisas recentes sobre a utilização de linguagens de programação. Diversas aplicações *iOS* que são *open-source* já foram desenvolvidas em Swift e estão disponíveis no GitHub. Em uma lista¹³ que contém cerca de 400 aplicativos *iOS*, com mais de 1,000 *commits* e 88 contribuidores, 162 desses aplicativos foram desenvolvidos em Swift.

¹⁰ <https://en.wikipedia.org/wiki/JavaScript>

¹¹ <https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html>

¹² <http://github.info>

¹³ <https://github.com/dkamsing/open-source-ios-apps>

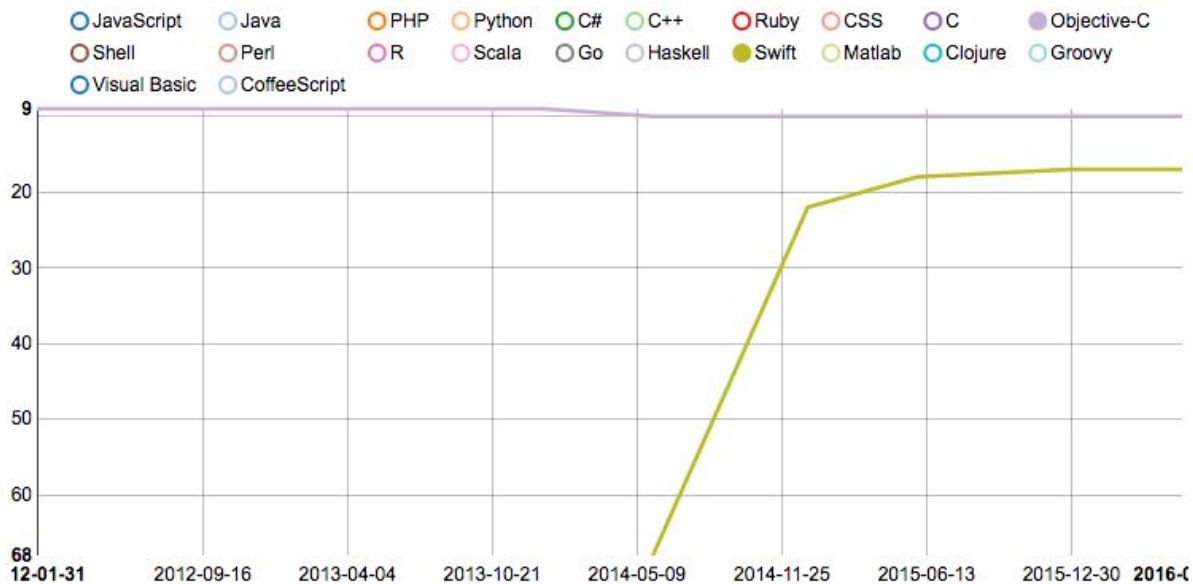


Figura 5. *Programming Language Rank: Position vs. Dates over the years. The RedMonk. Fonte: Programming Language Rankings: June 2016.*

Na Figura 5, podemos observar alguns resultados da pesquisa feita por O'Grady¹⁴ em Junho de 2016. É notória a crescente adoção de Swift, que partiu em 2014 da 68ª posição, já aparece no top 20 linguagens mais utilizadas do GitHub, agora na 17ª posição. Enquanto isso, Objective-C manteve-se estável e isso condiz com a proposta da Apple de dar suporte aos programas desenvolvidos em suas antigas plataformas, assim como o crescimento de Swift mostra que cada vez mais a linguagem cumpre seu papel de substituir a utilização de Objective-C. Apesar da crescente e notória adoção da linguagem, poucos iniciativas e estudos trouxeram até agora alternativas para análise de código Swift. Por essa razão uma abordagem de visualização de *software* voltada para sistemas desenvolvidos em Swift é uma interessante e importante colaboração para a comunidade e seus desenvolvedores.

Como a linguagem apresenta estruturas e conceitos de diferentes do usual por ser multiparadigma, não tem uma estrutura hierárquica bem definida, apresenta não só elementos de linguagens orientadas a objeto. Se faz necessário um estudo dos elementos presentes na linguagem para uma melhor mapeamento para a visualização, compreensão e representação da mesma. Alguns desses elementos são *Classes e Structures*¹⁵, *Protocols*¹⁶, *Enumerations*¹⁷ e *Extensions*¹⁸ e são descritos brevemente abaixo.

¹⁴ <http://redmonk.com/sogady/2016/07/20/language-rankings-6-16/>

¹⁵ https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/ClassesAndStructures.html

¹⁶ https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html

¹⁷ https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Enumerations.html

¹⁸ https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Extensions.html

2.4.1 Classes e Structures

Classes e *Structures* (Figura 6), são estruturas flexíveis que permitem definições de propriedades e métodos para adicionar funcionalidade a parte do seu código. Ambas são estruturas de funcionamento similar e estão presentes na maioria das linguagens.

```
1  struct Resolution {
2      var width = 0
3      var height = 0
4  }
5  class VideoMode {
6      var resolution = Resolution()
7      var interlaced = false
8      var frameRate = 0.0
9      var name: String?
10 }
```

Figura 6. Definição um *Struct* e uma *Class*. Fonte: *The Swift Programming Language (Swift 3.0.1)*.

2.4.2 Protocols

Protocols definem um modelo de métodos, propriedades e outros requisitos necessários para uma estrutura adequar-se a uma tarefa ou funcionalidade. *Protocols* podem ser utilizados por *classes*, *enumerations* ou *structs* para prover uma implementação em conformidade com tais requerimentos. Qualquer tipo na linguagem que satisfaça os requerimentos de um determinado *protocol* é dito que está conforme o *Protocol*.

```
1  protocol SomeProtocol {
2      var mustBeSettable: Int { get set }
3      var doesNotNeedToBeSettable: Int { get }
4  }
```

Figura 7. Definição um *protocol*. Fonte: *The Swift Programming Language (Swift 3.0.1)*.

São muito similares a interfaces de Java e outras linguagens orientadas a objeto, mas possuem particularidades como:

- Podem também especificar propriedades que precisam ser implementadas (funcionalidade não presente nas interfaces de Java¹⁹).
- Precisam lidar com valor e referência das variáveis/propriedades por conta do uso de *mutating keyword* que Swift permite.

¹⁹ <https://docs.oracle.com/javase/7/docs/api/>

- Pode-se combinar *protocols* em qualquer ponto com a palavra chave *protocol<>*. Por exemplo, no momento de declarar um parâmetro de uma função para que ele tenha que estar em conformidade com o *protocol* X e Z, como:

```
func foo ( parameter : protocol<X, Z> ){  
    ...  
}
```

Figura 8. Combinação de *protocols* com a palavra-chave *protocol<>*. Fonte: O autor.

2.4.3 Enumerations

Uma *enumerator* é um tipo comum para um conjunto de valores relacionados e permite que você trabalhe com esses valores de uma maneira segura de tipo dentro de seu código. Esse tipo de estrutura é comum e presente em diversas linguagens, porém em Swift sua construção é mais flexível e permite dar mais contexto aos valores do tipo criado. Por exemplo, o *enumerator* não fornece valor associado a cada caso do seu tipo, cada caso é o valor por si mesmo. Entretanto, caso queira dar um valor associado a cada caso, esse valor poderá ser um *int*, ou *string*, ou *char*, dentre outros. A seguir um exemplo de criação de um tipo chamado *CompassPoint*, exemplo retirado da documentação da linguagem.

```
1  enum CompassPoint {  
2      case north  
3      case south  
4      case east  
5      case west  
6  }
```

Figura 9. Criação de um *enumerator*. Fonte: *The Swift Programming Language (Swift 3.0.1)*.

Além disso esse tipo de estrutura pode ser implementado para obedecer um *protocol*, ter suas funcionalidades estendidas, criação de enumeração indireta (recursiva). Também pode ter um inicializador para fornecer valores iniciais, métodos de instâncias para prover funcionalidades aos casos e propriedades computadas para fornecer contexto adicional ao caso.

2.4.4 Extensions

Extensions adicionam um novo comportamento à estruturas já definidas, um tipo primitivo, uma *class* implementada... Como o nome sugere, elas podem estender as funcionalidades de *classes*, *structs*, *protocols*, *enumerations*. Abaixo uma definição simples de *extension* presente na documentação, mostrando a definição de uma *struct*, chamada *Rect*, e a *extension* provendo um inicializador adicional que recebe parâmetros *center* e *size*.

```

1  struct Size {
2      var width = 0.0, height = 0.0
3  }
4  struct Point {
5      var x = 0.0, y = 0.0
6  }
7  struct Rect {
8      var origin = Point()
9      var size = Size()
10 }

1  extension Rect {
2      init(center: Point, size: Size) {
3          let originX = center.x - (size.width / 2)
4          let originY = center.y - (size.height / 2)
5          self.init(origin: Point(x: originX, y: originY), size: size)
6      }
7  }

```

Figura 10. Implementação de um extension da struct Rect. Fonte: *The Swift Programming Language* (Swift 3.0.1).

Esse tipo de componente não possui nome, ao contrário das *categories*²⁰ em Objective-C, e pode adicionar propriedades computadas, métodos e tipos de métodos, inicializadores, definição de tipos aninhados, dentre outros.

Além de que pode fazer um tipo existente entrar em conformidade com um *protocol*, dada sua implementação. *Extensions* são interessantes, pois são blocos de código que pertencem a um componente já definido cuja implementação é feita de forma separada. É uma construção de Swift bastante utilizada, principalmente para modularização dos componentes, porém deve-se saber utilizá-la corretamente porque também pode acarretar em uma fragmentação desnecessária do código.

Essas estruturas compõem os principais componentes de bloco de código da linguagem. É importante, então, considerar suas naturezas e diferenças na implementação para que a visualização seja fiel, interessante e útil, mas também porque impactarão as decisões de desenvolvimento descritas no capítulo 5 deste trabalho.

²⁰<https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/Category.html>

CAPÍTULO 3

Trabalhos relacionados

Diversos trabalhos, como alguns dos já citados neste documento, usam uma abordagem de visualização para análise de sistemas de *software*. Neste capítulo destacam-se possíveis limitações e vantagens, dos trabalhos relacionados, visando um entendimento melhor do tema que orientará a construção da ferramenta proposta no trabalho.

3.1 Trabalhos em visualização

Os trabalhos de visualização de *software* exploram diferentes abordagens, ideias e técnicas para montar sua visão de análise. Muitas vezes revolucionando os conceitos utilizados, modelos seguidos e visões construídas, Ball (1996) constata isso em seu estudo que analisa inovações em ferramentas de visualização de *software*. E essas inovações acabam iniciando tendências como, por exemplo, forte adoção e utilização dos conceitos e ferramentas construídas.

Apesar desse fato, visualização e análise devem ser feitas com cuidado, pois as abordagens costumam não ser genéricas o suficiente para serem usadas sem o estudo adequado. Existem prós e contras de se utilizar cada tipo de visão, e os objetos de análise são diferentes. É ideal que se estude as visões e o que se quer analisar para combinar para, então, adequar ambos na construção de uma ferramenta interessante.

Muitas ferramentas têm por objetivo a análise do código e sua evolução, Erick, ainda em 1992, introduziu uma ferramenta chamada Seesoft. Ferramenta essa baseada em visualização da evolução das linhas de código no sistema.

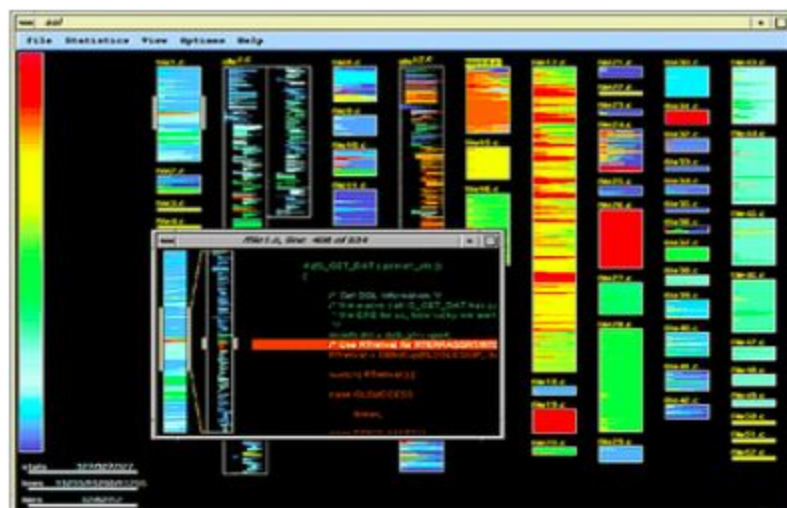


Figura 11. Evolução das linhas de código do sistema. Fonte: Erick, 1992.

O objeto de análise de Erick são as linhas de código e a intenção dele é fazer um rastreamento, colorindo e criando escalas para identificar mudanças no código e a frequência dessas mudanças. Lanza, em 2001, propõe uma análise na evolução do código, mas com base nas classes de um sistema. Ele extrai métrica das classes e constrói o que chama de matriz de evolução, a ideia é que cada coluna da matriz representa uma versão do código e cada linha representa uma classe. Logo teria-se classe ao longo das versões, e cada célula contém um quadrado cuja largura é dada pelo número de métodos e a altura dada pelo número de instâncias de variáveis.

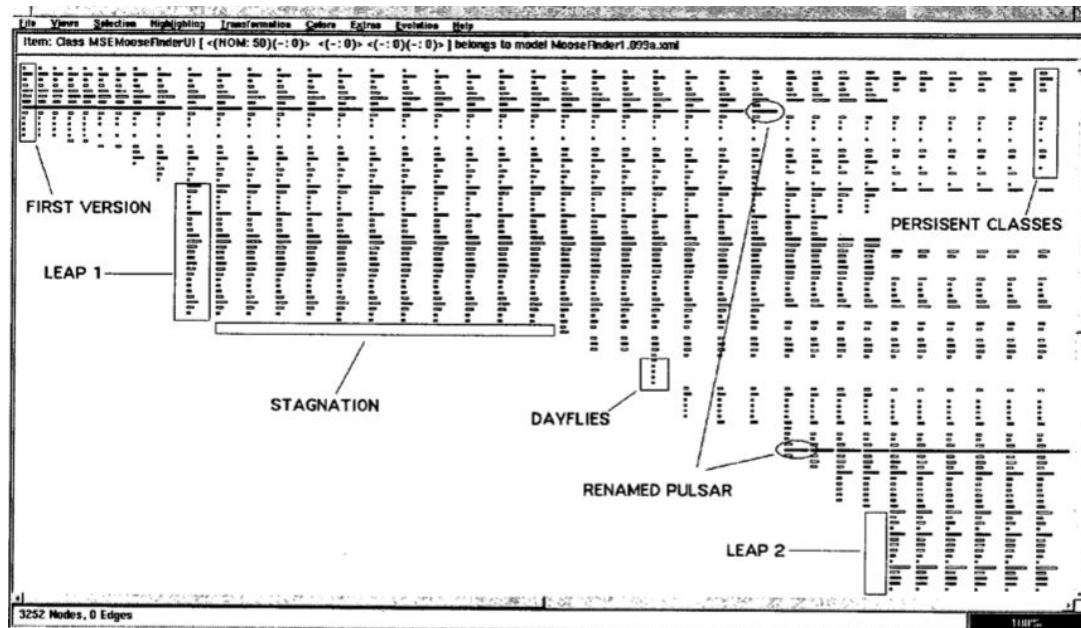


Figura 12. A matriz de evolução do MooseFinder. Fonte: Lanza 2001.

Apesar da proposta de Lanza ser diferente, o resultado da visão é similar ao de Erick, o que muda é o objeto em análise. Essa visão é interessante para acompanhar a evolução das classes, mas a análise do impacto de cada componente no sistema fica prejudicada por conta da quantidade de informação exibida.

Ainda sobre evolução do sistema, no capítulo 2 deste trabalho foram apresentados as ferramentas de Beyer (2006) e Collberg (2003), que com suas visões baseadas em grafos tentam explorar de maneira diferente o impacto dos componentes no sistema ao longo do tempo. Porém, dada a quantidade de elementos presentes na visão a sobreposição e distribuição dos elementos fica prejudicada. Por conta disso, além de entender a visão e os dados que se quer analisar, é interessante implementar sistemas de navegação e interação que permitam uma melhor experiência.

Treemap²¹ é uma visão diferente, porém conhecida na área de visualização de dados. Balze (2005), sugere na construção de sua ferramenta uma abordagem que alia uma variação do Treemap com a visualização de sistema de *software*, Figura 13.

²¹ <https://en.wikipedia.org/wiki/Treemapping>

O objetivo é apresentar uma visão hierárquica e navegável do sistema baseado em definições e métricas a respeito do mesmo.

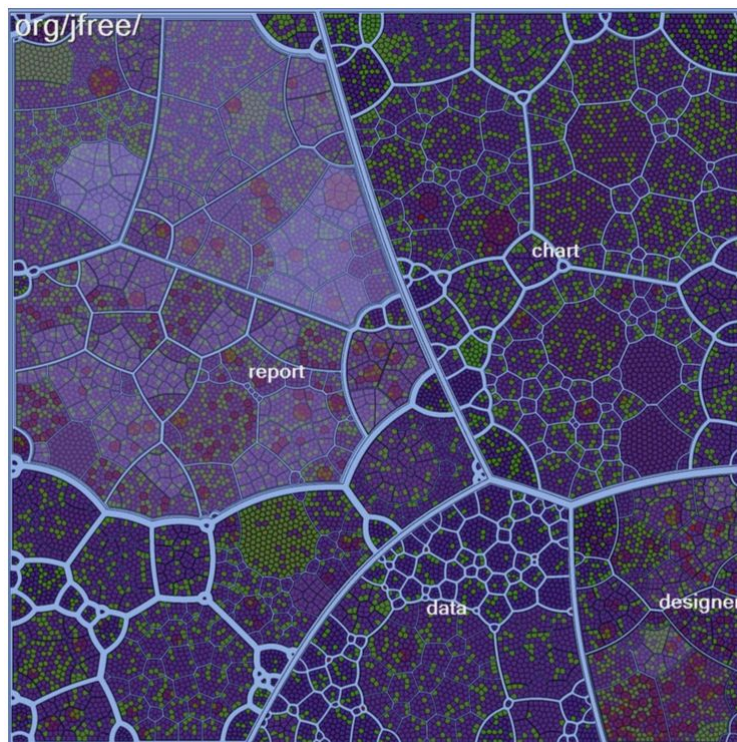


Figura 13. Voronoi Treemap da estrutura estática do sistema JFree. Fonte: Balze 2005.

A ferramenta pode fazer análise de diferentes aspectos do programa. Sua representação permite ao usuário identificar quais subsistemas são os maiores e densos. Entretanto, é necessário navegar nos níveis para analisar os componentes de menor granularidade como classes. Dependendo da linguagem isso é razoável. Em Java, por exemplo, cada arquivo normalmente está associado a uma classe, a implementação de maior parte do código se dá em classes e a hierarquia de pacotes e subsistemas é razoavelmente bem definida. Entretanto, em Swift a implementação de por parte do código se distribui dentre os componentes citados no capítulo 2 (*classes*, *protocols*, *structs*, *enumerations* e *extensions*), um arquivo contém mais de um componente desse e com hierarquia de sistema imprecisa.

Este trabalho propõe a visualização baseada na metáfora das cidades e o conceito utiliza elementos 3D para a composição da visão. Com isso tem-se uma dimensão a mais para agregar informação a respeito do sistema em análise, porém também significa que é preciso ter cuidado com a forma de representação, sobreposição e distribuição dos elementos, navegação, granularidade, que podem ser potencializados com a utilização da terceira dimensão. Além de outros problemas que surgem com a adição de uma dimensão, como por exemplo a oclusão de elementos, a difícil representação das escalas e comparação dos elementos.

Apesar da complexidade e, possivelmente, de problemas diferentes a se mitigar com a abordagem da visualização, é possível construir uma ferramenta interessante, como nos exemplos já citados no capítulo 2. Além disso, de maneira muito similar a ferramenta JSCity (Figura 14), desenvolvida por Viana (2015), é uma ferramenta para visualização de sistemas de *software* desenvolvidos em JavaScript utilizando o conceito da metáfora das cidades. Também tendo como o objetivo de analisar e representar bem as particularidades dos sistemas, em sua linguagem, numa plataforma simples e interessante.

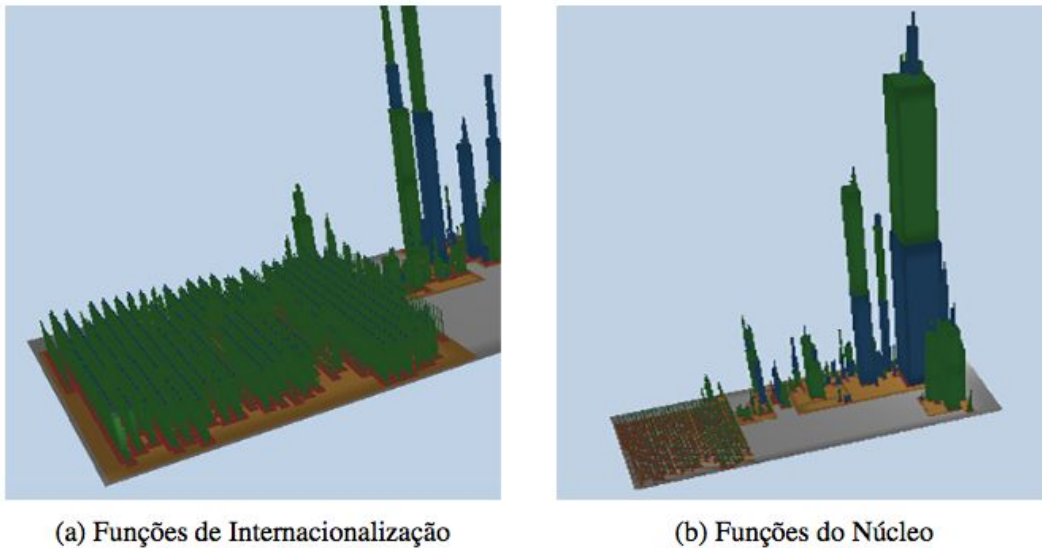


Figura 14. Cidade do sistema AngularJS. Fonte: Viana 2005.

CAPÍTULO 4

Desenvolvimento

Este capítulo aborda o método de desenvolvimento da ferramenta. Descreve-se, portanto, como se deu a pesquisa, como a coleta de dados e métricas foi realizada, quais tecnologias foram escolhidas para o desenvolvimento e processo de desenvolvimento.

4.1 A Pesquisa e Coleta de Dados

Este trabalho é fruto da pesquisa colaborativa sob orientação do professor Fernando Castor e colaboração do mestrando Marcel Rebouças. Fernando e Marcel estão engajados na pesquisa e análise de projetos Swift, na análise da utilização da linguagem pela comunidade, mas também estudo das particularidades de seus tipos e impacto de suas construções. Sendo assim, um dos objetivos deste trabalho é colaborar com a pesquisa, fornecendo uma ferramenta que possibilite o estudo de projetos desenvolvidos na linguagem e que retrata bem suas nuances.

Para a coleta de dados utilizou-se a ferramenta construída por Marcel. A ferramenta utiliza o compilador de Swift, em um modo chamado *-dump-ast*. Nesse modo, quando o projeto é compilado, imprime-se no console uma representação da AST²² do mesmo. Marcel modificou o modo do compilador para imprimir o resultado em um arquivo e sua ferramenta utiliza a representação da AST como *input*. Sendo assim, usando tal *input*, realiza-se um *parsing*²³ desse arquivo para extrair as *classes*, *extensions*, métricas, e outros elementos do projeto. Por fim, produz, como *output*, um arquivo no formato JSON²⁴ com os metadados resultantes da análise da AST. O arquivo contém informações sobre as estruturas encontradas no projeto, como: nome, tipo, *source path*, linhas de código, número de métodos, métodos, etc. O *output* da ferramenta desenvolvida por Marcel, serve então, como *input* para a que foi desenvolvida nesta pesquisa. A figura abaixo exibe um trecho do JSON, ilustrando algumas informações que ele contém.

```
5  "structs": [  
6    {  
7      "name": "CVSet",  
8      "node_type": "struct",  
9      "type": "CVSet.Type",  
10     "number_of_lines": 28,  
11     "source_path": "/CVCalendarz/CVCalendar/CVSet.swift",
```

Figura 15. Trecho de um arquivo JSON, output para o projeto CVCalendar. Fonte: O autor.

²² https://en.wikipedia.org/wiki/Abstract_syntax_tree

²³ <https://en.wikipedia.org/wiki/Parsing>

²⁴ <http://www.json.org>

4.2 As Tecnologias de Desenvolvimento

A proposta da ferramenta é que ela seja fácil de se acessar e utilizar. Murray (2013) em seu livro, "*Interactive data visualization for the Web*" exalta que visualizações só são verdadeiramente visuais se elas podem ser vistas, a maneira com que publica-se um trabalho e o produto é crítica e publicar na *web* é a maneira mais rápida de se atingir audiência global.

Seguindo os princípios SAAS²⁵, descritos por Fox (2013), o ambiente escolhido para execução da ferramenta é o *web*. Ideia do SSAS é oferecer o *software* como serviço ao invés de produto. Como produto, um *software* é comprado na loja, instalado em cada máquina para, então, começar a usar. Como serviço a proposta é que não se compre o produto, ao invés disso, o produto é acessado pelo usuário, mas não instalado, através da *web*. Com isso, não existe a necessidade instalar as aplicações, facilita o processo de lançamento de uma nova versão, dentre outras vantagens.

Para que seja possível implementar a ferramenta em ambiente *web* é necessário utilizar tecnologias que navegadores como, por exemplo, Google Chrome²⁶, FireFox²⁷, Internet Explorer²⁸, deem suporte e sejam capazes de executar. Sendo assim, diferentes tecnologias de desenvolvimento para essa plataforma foram estudadas para diferentes finalidades dentro do projeto. Utilizou-se HTML²⁹ que é a linguagem de marcação de texto, é a padrão para a construção de aplicações e sistemas *web*. Elementos HTML são blocos de construção de páginas, pode-se formatar imagens, blocos de texto, mídia e outros objetos. Além disso a linguagem pode incorporar trechos de linguagens de *script* para prover funcionalidade e comportamento nas páginas.

Para se incorporar estilo, utilizou-se CSS³⁰, que é a linguagem de estilo de folha padrão para *web*. Essa linguagem é utilizada para descrever a apresentação de um documento escrito em uma linguagem de marcação de texto. Além disso, o MaterializeCSS³¹ é um *framework* de CSS que foi incorporado ao projeto. Esse é um projeto de código aberto³² que tem um grande conjunto de funções de estilo, atualizado constantemente e uma grande comunidade que colabora para ele, possuindo mais de 3300 *forks* e mais de 23 mil estrelas no GitHub. A utilização desse *framework* foi essencial para a implementação dos estilos da aplicação, garantindo uma unidade e uniformidade do *layout* da ferramenta.

²⁵ https://en.wikipedia.org/wiki/Software_as_a_service

²⁶ <https://www.google.com/chrome/>

²⁷ <https://www.mozilla.org/en-US/firefox/new/?gclid=CPfOhP2X1tACFcYHkQodcwMHOg>

²⁸ <https://www.microsoft.com/en-gb/download/internet-explorer.aspx>

²⁹ <https://en.wikipedia.org/wiki/HTML>

³⁰ https://en.wikipedia.org/wiki/Cascading_Style_Sheets

³¹ <https://github.com/Dogfalo/materialize>

³² https://pt.wikipedia.org/wiki/C%C3%B3digo_aberto

Para se incorporar comportamento e funcionalidade às estruturas da página foi utilizado JavaScript, que é uma linguagem de *script* e juntamente com HTML e CSS forma o conjunto de tecnologias padrão para desenvolvimento para *web*. A maioria dos *websites* a incorpora e todos os navegadores atuais suportam a linguagem sem necessidade extensões ou instalação.

JavaScript possui um vasto conjunto de funções e bibliotecas padrão e também é amplamente utilizada como base para diversos sistemas, extensões e frameworks. Uma das funções da linguagem nos permite importar um arquivo JSON, via *upload*, sem que haja necessidade de um servidor. Nesse caso, o próprio navegador processa o arquivo para que ele esteja disponível para utilização na ferramenta. Essa funcionalidade é extremamente importante, pois remove a necessidade de se criar um servidor apenas para a submissão do arquivo e logo a aplicação apenas com o *front-end* é suficiente para as funcionalidades e necessidades descritas no capítulo seguinte deste trabalho.

Por essas razões JavaScript é suficiente e foi escolhida como linguagem da ferramenta a ser desenvolvida. Entretanto, HTML/CSS não possuem elementos 3D nativamente, algo que é essencial para a proposta de construção da metáfora das cidades. É necessário, então, utilizar algum *framework* ou linguagem que construa e gerencie a renderização de elementos 3D em aplicações *web*. Avaliou-se a utilização de três APIs³³ que dão suporte a esse requisito, que são: WebGL³⁴, Three.js³⁵, x3dom³⁶.

WebGL é uma API, baseada no OpenGL ES³⁷, para JavaScript compatível com navegadores, sem a necessidade de extensões, e permite uso de GPU para processar imagens, cálculos físicos e efeitos como parte do *canvas* das páginas *web*. Também permite que seus elementos sejam misturados a outros, de HTML por exemplo. Entretanto, WebGL é uma API considerada de baixo nível e por isso difícil de se lidar, além disso a documentação é dispersa e difícil de se encontrar informações úteis na prática. Three.js e x3dom são duas alternativas, de *frameworks*, baseadas em WebGL que abstraem detalhes e dificuldades de implementação através de construções e elementos simples.

No x3dom, a cena, objeto que contém a visão 3D, é diretamente escrita através de marcações HTML. Para funcionar, nenhuma extensão é necessária, apenas que seja importado um *script* JavaScript que é o código da API. Ela usa um sistema de *fallback* para a renderizar os elementos, por exemplo, tenta utilizar WebGL, caso não consiga, tenta OpenGL e assim por diante por alguns níveis. O projeto é de código aberto, está disponível no GitHub, e conta com mais de 390 estrelas e 170 *forks*. A proposta é que elementos 3D pareçam nativos, por serem escritos através de elementos HTML.

³³ https://en.wikipedia.org/wiki/Application_programming_interface

³⁴ <https://en.wikipedia.org/wiki/WebGL>

³⁵ <https://threejs.org>

³⁶ <http://www.x3dom.org>

³⁷ https://en.wikipedia.org/wiki/OpenGL_ES



Figura 16. Marcação HTML para um cubo e resultado da renderização. Fonte: *x3dom.org*.

Entretanto, ela não é uma plataforma prática porque a gerência dos objetos em cena acaba tendo de ser feita através de *scripts* que: selecionem, removam, adicionem e alterem os elementos HTML. Além disso, as funções de interação são limitadas e, como os elementos são representados por elementos HTML, é necessário, muitas vezes, vários níveis de blocos para a implementação de um objeto.

Three.js por sua vez é uma solução baseada em *scripts* escritos em Javascript, em que os elementos são objetos JavaScript adicionados a um objeto de cena, e um objeto de renderização avalia e atualiza a cena a cada instante. Assim como o x3dom nenhuma extensão é necessária, basta adicionar o *script* da biblioteca para ter acesso às funcionalidades e que o navegador esteja habilitado para WebGL. O projeto também é de código aberto e está disponível no GitHub, porém é um projeto que recebe bem mais contribuições, com uma vasta comunidade, contribuições diárias e conta com mais de 29 mil estrelas e 10 mil *forks* no GitHub. Por ser um *framework* que gerencia os elementos, cena e a renderização de maneira programática, o desenvolvedor tem muito mais controle sobre a aplicação. Three.js conta com uma documentação bem detalhada, exemplos bem definidos e informações práticas e úteis na *web*. Além disso a biblioteca tem uma grande quantidade de funcionalidades e construções para a renderização, como funções de controle de câmera e mouse, *raycaster*³⁸, importar modelos 3D, texturas, animações, dentre outras. Por ser uma plataforma programática, uma comunidade bem ativa e uma documentação vasta e útil, Three.js foi escolhida como biblioteca para a renderização dos elementos 3D deste trabalho.

Por fim, utilizou-se também o D3.js (Data-Driven Documents), uma biblioteca também escrita em JavaScript especializada para manipulação e visualização de dados. É uma API de código aberto com 57 mil estrelas e 15 mil *forks* no GitHub. Bem documentada e fortemente adotada, a biblioteca fornece um conjunto de funções para tratamento de dados como, por exemplo, carregamento de dados, escalas, funções estatísticas, funções matemáticas. Também apresenta funções para visualização e renderização de elementos 2D, caso que não será explorado no trabalho. Além de funções de seleção e manipulação de blocos HTML, o que permite adicionar, de forma programática, comportamento, elementos, estilos e fazer edições na aplicação como um todo.

³⁸ https://en.wikipedia.org/wiki/Ray_casting

Em suma, as tecnologias adotadas no desenvolvimento da ferramenta foram: HTML/CSS para estrutura e estilo, MaterializeCSS como *framework* de estilo, Three.js como biblioteca de renderização e gerência da visualização 3D e D3.js para manipulação dos dados e dos elementos da aplicação.

4. 3 O Processo de Desenvolvimento

O processo de desenvolvimento da ferramenta se deu em fases. Elegeram-se os requisitos ³⁹ iniciais da visualização com base na metáfora, por exemplo, renderização, navegação, interação. Junto a isso, os requisitos de utilização da ferramenta, como *upload* do arquivo de entrada, hospedagem da aplicação, entre outros aspectos. Caso surgissem novas ideias e requisitos da pesquisa, discussão e implementação, eles eram adicionados ao conjunto de requisitos do sistema. Para cada novo requisito sua prioridade de implementação era definida dentro do projeto, os requisitos que fossem essenciais para a metáfora recebiam uma prioridade mais alta, já os requisitos considerados importantes, porém opcionais recebiam uma prioridade mais baixa. Dessa forma, foi possível gerenciar os requisitos para que se atingisse um equilíbrio entre os requisitos essenciais e os importantes.

A cada fase foram selecionados os requisitos dada sua ordem de prioridade, discutiu-se com o abordar tal requisito, implementou-se e então discutiu-se o resultado do mesmo para realizar mudanças ou melhorias. As reuniões para discussão dos resultados, quando não podiam acontecer pessoalmente, eram feitas de forma online ou aconteciam via *email*.

Para a discussão de como se usaria a ferramenta, inicialmente realizou-se uma fase de prototipação. Prototipar⁴⁰ é uma fase importante no desenvolvimento de *software*, pois ajuda a eliminar ambiguidades e melhorar a precisão na interpretação dos requisitos do sistema, também ajuda a garantir que a solução faça o que é supostamente deveria fazer, entre outros benefícios⁴¹.

A princípio construiu-se um *mockup*⁴² (mock-up) da ferramenta, que é uma representação estática de média a alta fidelidade de um design. O propósito era enxergar e tentar organizar a estrutura da informação, visualiza o conteúdo e demonstra as principais funcionalidades de uma forma estática e procurar revisar a parte visual da aplicação. Na Figura 17 temos o *mockup* feito durante a idealização do sistema. Para fins de alcance global, ou seja, que a aplicação possa atingir e ser útil a desenvolvedores do mundo todo, o design interface da aplicação foi projetado em inglês.

³⁹ https://en.wikipedia.org/wiki/Software_requirements

⁴⁰ <https://pt.wikipedia.org/wiki/Protótipo>

⁴¹ http://www.sqa.org.uk/e-learning/IMAuthoring01CD/page_06.htm

⁴² <https://en.wikipedia.org/wiki/Mockup>

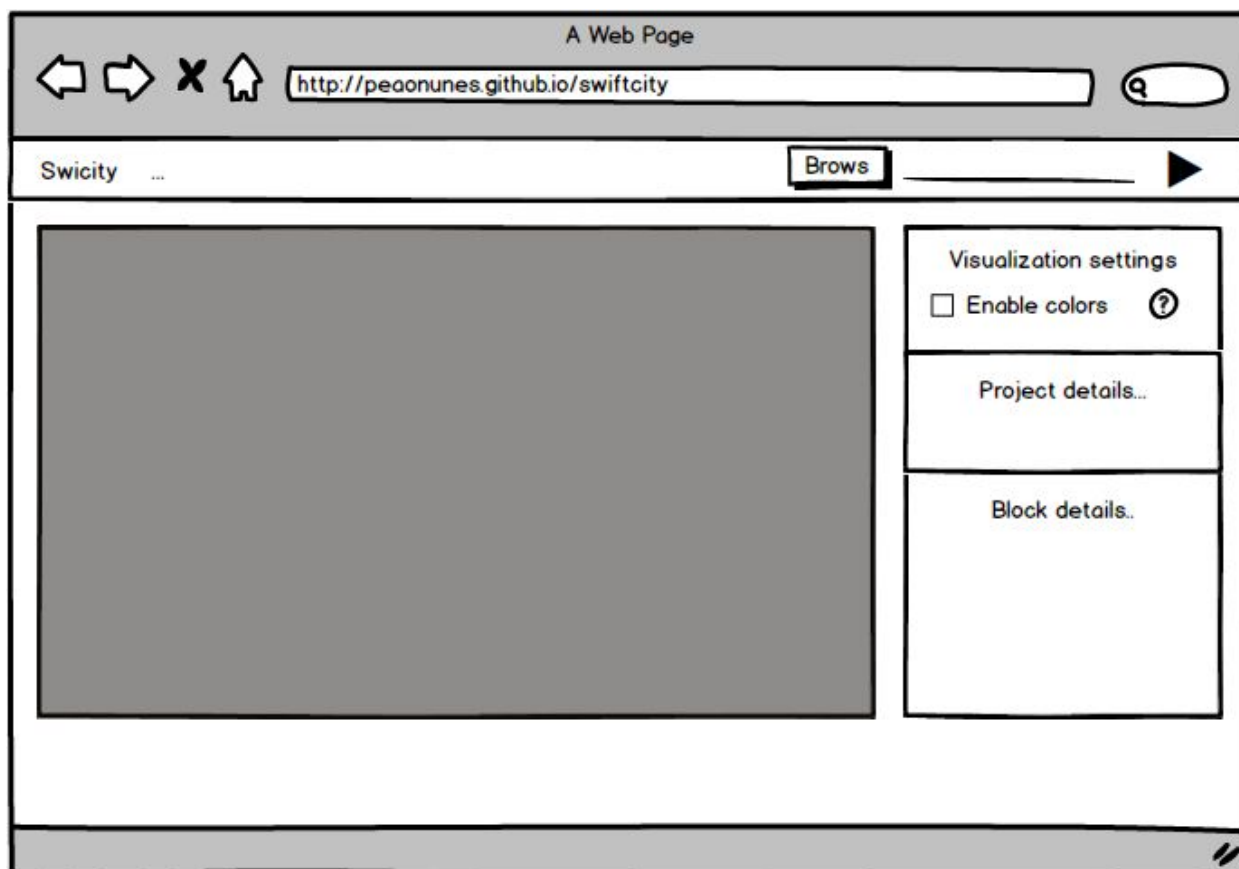


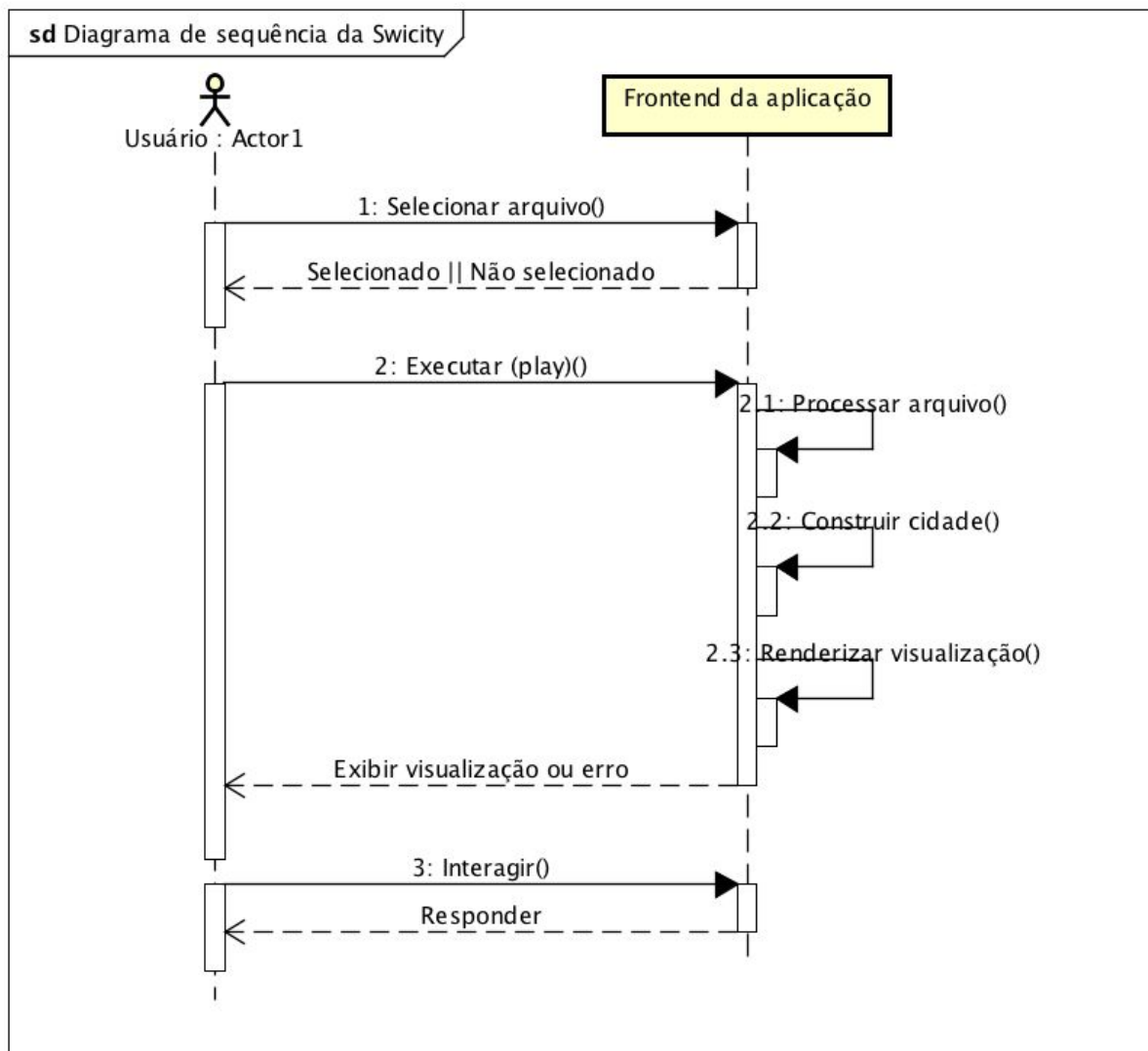
Figura 17. Mockup da ferramenta. Fonte: O autor.

O *mockup* foi construído utilizando-se a ferramenta Balsamiq⁴³, uma ferramenta que provê um conjunto de elementos visuais para a criação de protótipos. Na Figura 16 nota-se que a interface é bem simples, a barra de navegação foi idealizada para que o usuário possa submeter o arquivo de entrar e então apertar no botão de *play* para executar a ferramenta. O quadro em cinza escuro é o maior elemento, pois representa o campo de renderização, ou seja, onde a visão da cidade será construída e o usuário irá interagir. Na coluna visível a direita existe um menu para configurações dos elementos da visualização, até então habilitar cor dos elementos era a única opção imaginada. Mas também, na coluna existem ainda dois espaços, o primeiro que se imaginou conter um resumo das informações do projeto. O segundo um espaço com o resumo das informações de um determinado bloco, complementando o requisito da metáfora que é a interação e seleção de um objeto.

Além do *mockup* acima, protótipos de média e alta fidelidade da ferramenta foram desenvolvidos. À medida que os requisitos foram sendo implementados, eles eram testados a partir de uma versão da interface que permitiria a interação e a execução de tal requisito.

⁴³ <https://balsamiq.com/?gclid=CLiB3pSy4NACFYWakQodJOUAkQ>

A utilização da ferramenta também foi analisada, de maneira simples, através de uma diagrama de sequência. Como as funções de JavaScript removeram a necessidade de um servidor o fluxo da aplicação dá-se totalmente no lado do cliente (o navegador), pode ser visto na Figura 18.



powered by Astah

Figura 18. Diagrama de sequência da ferramenta, ilustrando seu fluxo. Fonte: O autor.

No diagrama, o fluxo se dá todo no cliente da aplicação, após a submissão do arquivo pelo usuário, as funções internas da aplicação leem o arquivo, projetam a cidade e a renderizam na tela. Após a renderização a aplicação torna para um estado de interação onde o usuário pode interagir diretamente com a cidade e a aplicação responderá às suas ações. Além disso, o usuário seria capaz de voltar ao fluxo inicial, selecionando um outro arquivo e podendo voltar ao fluxo da execução, quando as configurações da visualização forem alteradas.

CAPÍTULO 5

A ferramenta

Neste capítulo explica-se os elementos que compõem a visualização e como foi o processo da tradução da metáfora das cidades para sistemas em Swift. E, por fim, as funcionalidades disponíveis atualmente na aplicação, também discutindo as abordagens tomadas.

5.1 Elementos da Metáfora das Cidade em Swift

Nesta seção apresenta-se a tradução da metáfora das cidades para Swift, revisando e explorando com cuidado as particularidades e as dificuldades de se representar um sistema de *software* em Swift como cidade. O bloco é o elemento fundamental da metáfora, ele representa um prédio da cidade e originalmente seria mapeado para representar uma classe do sistema. Entretanto, Swift possui diversos tipos, como visto no capítulo 2. Por isso, foi necessário escolher os que devem ser traduzidos para a visualização, foram eles: *classes*, *extensions*, *protocols*, *structs* e *enumerations*. Logo, cada bloco na cidade representa uma construção de um dos tipos.

Para se construir um bloco é necessário utilizar métricas que definem sua altura, sua largura e seu comprimento. Diversas e diferentes métricas podem ser aplicadas ao contexto e tornar isso parametrizável também é algo possível na visualização. Entretanto, para este trabalho, a altura do bloco é dada pelo LOC (*Lines of Code*), número de linhas de código do elemento. A largura e comprimento do bloco, por sua vez, são dados pelo NOM (*Number of methods*), número de métodos do elemento. Essas métricas são normalmente utilizadas em ferramentas de análise como visto por Lincke (2008) e são interessantes para o contexto de análise de sistemas Swift. A NOM, principalmente, pelo fato de que é possível que os tipos de Swift, até mesmo *enumerations*, podem ter métodos e instâncias de métodos implementadas.

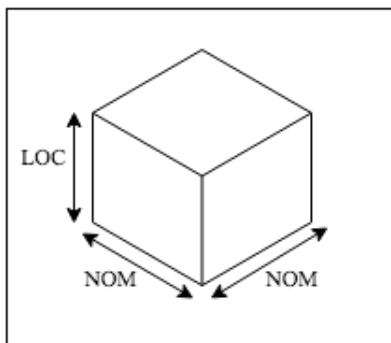


Figura 19. Representação do bloco com base nas métricas. Fonte: O autor.

A metáfora aplicada a sistemas em Java, onde a principal unidade de código é uma classe, faz com que os blocos tenham uma cor uniforme. Entretanto, em Swift um bloco que seja um *protocol*, que só contém definições de métodos, será um bloco bem largo e baixo, por outro lado uma *class* pode ser um bloco alto e pouco largo. Logo, caso não sejam facilmente identificados, ou seja, sendo não sejam vistos como blocos dos seus respectivos tipos, pode acarretar a uma conclusão equivocada a respeito da complexidade do elemento. Por esse motivo, foi estudado a possibilidade de representar as categorias dos tipos através de cores ou formas. Zanolie, (2008) aponta em seu estudo vantagens e desvantagens de cada abordagem: formas são usualmente associadas à representação de elementos de natureza diferente, porém todos os elementos são tipos de Swift apenas pertencem a categorias diferentes. Além disso, a utilização de formas diferentes iria contra a metáfora, traduzindo os elementos a objetos que não necessariamente criam o ambiente de imersão em uma cidade.

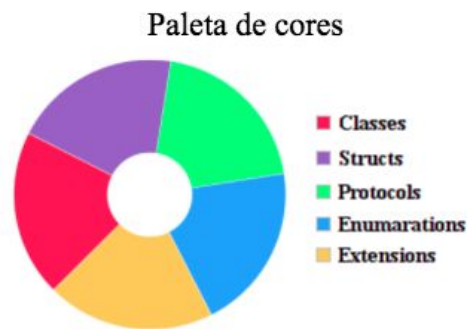


Figura 20. Paleta de cores escolhidas. Fonte: O autor.

A Figura 20 mostra a paleta escolhida, um processo que deve ser estudado com cuidado, como Maureen Stone cita em seu guia *Choosing Colors for Data Visualization*⁴⁴, as cores são fundamentais para garantir clareza, legibilidade, guiar o usuário, etc. A paleta define o grupo de cores de maneira qualitativa, com contrastes que mapeiam uma cor a um tipo diferente de elemento de Swift. A montagem da paleta foi feita utilizando o guia, ColorBrewer2.0⁴⁵, e também a ferramenta de composição de cores, Adobe Color CC⁴⁶.

Outro elemento importante na construção da cidade é a topologia, normalmente mapeada para elementos da hierarquia do sistema. Como Swift tem uma hierarquia imprecisa, no sentido que não existem pacotes⁴⁷, namespaces⁴⁸ ou equivalentes, o mapeamento limita-se dois níveis claros de hierarquia, são eles: o projeto e os arquivos. Primeiro e mais baixo nível representa os limites da cidade como sendo o projeto todo, e cada elemento do segundo nível representa um distrito que é arquivo dentro do projeto. A Figura 21 mostra o primeiro nível (cinza escuro), o segundo nível (cinza claro), e os blocos dentro dos seus respectivos arquivos.

⁴⁴ https://www.perceptualedge.com/articles/b-eye/choosing_colors.pdf

⁴⁵ <http://colorbrewer2.org>

⁴⁶ <https://color.adobe.com>

⁴⁷ <https://docs.oracle.com/javase/tutorial/java/concepts/package.html>

⁴⁸ <https://en.wikipedia.org/wiki/Namespace>

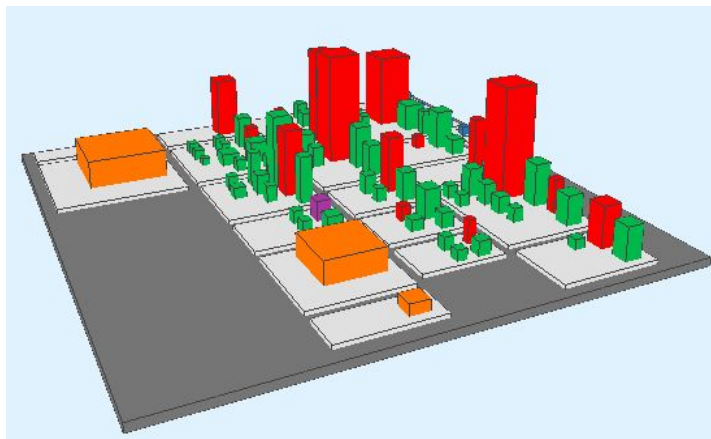


Figura 21. Topologia do projeto CVCalendar, na versão 0.5 da visualização. Fonte: O autor.

A topologia apresentada na Figura 21 retrata bem a hierarquia nos níveis de topologia que se pode extrair do projeto. Porém, nessa figura, uma versão mais antiga da aplicação, as *classes* estão em vermelho e as *extensions* estão em verde. Com isso identificou-se um problema do estado da visualização.

Muitas das *extensions*, nesse projeto, estendem o comportamento das *classes*, ou seja, alguns blocos representam implementações, semanticamente do mesmo elemento, porém feitas em blocos de código diferentes. Entretanto, a informação de qual bloco estende e quem é o bloco estendido por ele não é representada, isso acontece porque esses blocos não estão relacionados na visualização de alguma maneira. É fundamental, então, estudar uma formas de representar esse relacionamento para que fique claro o impacto e a complexidade de utilizar esses tipos de construções no código.

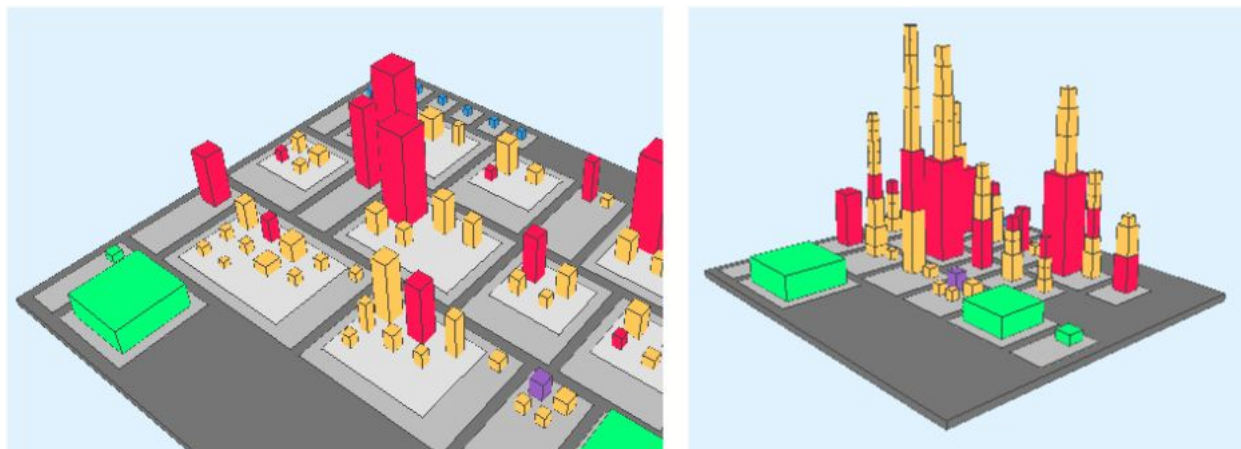


Figura 22. Lado esquerdo representação da relação com adição de um nível na topologia. Lado direito adição da relação empilhando os blocos relacionados. Versão 0.7 da ferramenta. Fonte: O autor.

A Figura 22, mostra as duas abordagens que foram discutidas para adaptar a metáfora a esse requisito, aplicando-as no mesmo projeto, CVCalendar. Nessa versão da ferramenta, as *classes* estão em um tom de vermelho e as *extensions* em tom de laranja.

A primeira, do lado esquerdo, propõe exibir a relação através do agrupamento entre o bloco que estendem e os bloco que por ele são estendidos. Esse agrupamento adiciona um novo nível na topologia, representado pelo cinza mais claro na visualização. Dessa maneira, poderia-se traduzir o novo nível para uma representação de um bairro na metáfora da cidade. A segunda abordagem, lado direito da figura, apresenta as *extensions* e os blocos relacionados empilhados. Nessa forma, empilha-se os elementos relacionados ordenando de baixo para cima, a partir da maior base até a menor base, e tendo a altura como o critério de desempate.

As duas abordagens foram consideradas satisfatórias no que diz respeito a cobrir o requisito de representar o relacionamento. Existem, entretanto, vantagens e desvantagens de cada uma. A primeira, por exemplo, tem uma visão mais limpa porque aumenta o tamanho da cidade dado que os objetos mais espalhados. Por outro lado, a segunda, traduz melhor a complexidade dos elementos, pois a construção fica mais alta, representando o que Wettel (2007) chama de "prédio de negócios". Porém pode causar problema de oclusão entre os blocos, por conta da quantidade de blocos próximos e suas alturas parecidas, o que dificulta a visão e comparação. Para mitigar isso, existem sistemas de navegação e interação na visão que permitem o usuário explorar a cidade. Considerando que a visualização depende muito do projeto analisado, as duas abordagens foram mantidas e são selecionáveis como uma configuração da visualização. Dessa maneira, o usuário terá liberdade de escolher a forma que mais lhe agrade e ache interessante para o visualizar seu projeto.

5.2 Funcionalidades, Navegação e Interatividade

Esta seção descreve as funcionalidades e as decisões relacionadas a construção de algumas delas. A Figura 23, apresenta uma visão geral da aplicação quando acessada. Um guia de utilização da ferramenta está disponível como apêndice A.

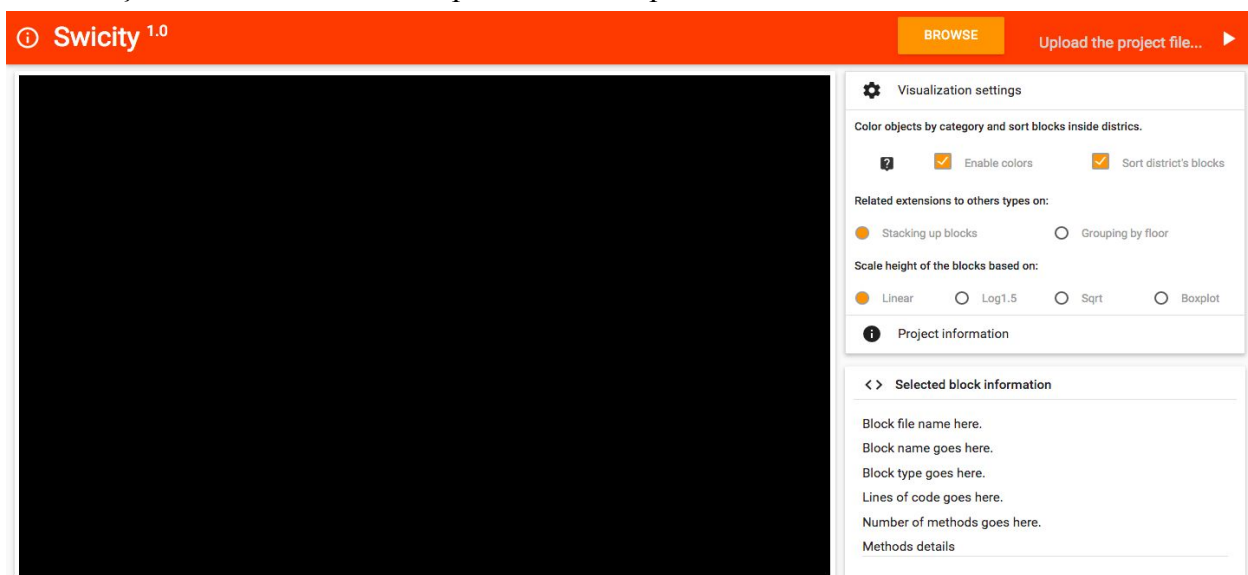


Figura 23. Visão geral da ferramenta na versão 1.0. Fonte: O autor.

5.2.1 Buscar e Executar

Na barra de navegação da ferramenta, encontram-se as opções de busca e execução. O botão *browse* permite que usuário selecione o arquivo de entrada no seu computador, o arquivo, como já citado, deve ser o *output* fornecido pela ferramenta de Marcel. Ao lado do direito do botão *browse*, existe um botão de *play*, que, quando acionado, lê o arquivo selecionado e executa a aplicação, renderizando a cidade do projeto escolhido. A cidade, então, é exibida no *canvas* que estava inicialmente em preto.

5.2.2 Configurações de Visualização

No lado direito da aplicação existe uma coluna. O primeiro elemento dessa coluna é um menu responsável por controlar as configurações de visualização. A aplicação fornece um conjunto de opções para que o usuário possa explorar a visão de acordo com as suas preferências, entretanto respeitando a metáfora e os conceitos estudados. Para o usuário compreender o efeito e o que cada opção realiza, existe um botão de ajuda sinalizado por um balão com uma interrogação logo na primeira linha do menu de configurações.

Color objects by category and sort blocks inside districts.

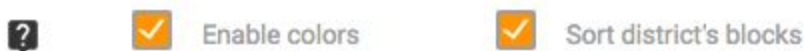


Figura 24. Primeira linha das configurações de visualização. Fonte: O autor.

Na primeira linha existem duas outras opções. Uma delas é uma opção chamada *Enable colors*, essa opção quando ativada habilita a renderização de cores com base na paleta já mencionada, caso desabilitado a renderização utiliza uma cor uniforme para todos os blocos. A outra opção é chamada *Sort district's blocks*, essa opção quando habilitada ordena os blocos dentro de um distrito pela altura do bloco e utiliza a largura como critério de desempate. Quando desabilitada essa ordenação não acontece. O objetivo dessa ordenação é enxergar de maneira simples os elementos mais complexos dentro de um arquivo (distrito).

Related extensions to others types on:



Figura 25. Segunda linha das configurações de visualização. Fonte: O autor.

Na segunda linha de opções, está presente a opção em que o usuário pode escolher entre uma das abordagens já citadas para representação da relação dos blocos estendidos e, então, a renderização se atualizará para entrar em conformidade com a escolha.



Figura 26. Terceira linha das configurações de visualização. Fonte: O autor.

Por fim, na terceira linha, existe uma opção que permite ao usuário escolher a escala da altura dos blocos. Wettel (2007), quando introduz a metáfora, observa que a escala com que se produz a altura dos blocos é importante, pois dependendo do projeto a visão funciona melhor quando normalizada. A aplicação desenvolvida neste trabalho fornece quatro funções de escalas, são elas:

- *Linear*, a altura de cada bloco é o retorno da função linear para o LOC do bloco, cujo domínio está entre o mínimo e o máximo número de LOC, entre os blocos do projeto.
- $\log_{1.5}$, a altura de cada bloco é o retorno da função log na base 1.5 para o LOC do bloco, cujo domínio está entre o mínimo e o máximo número de LOC, entre os blocos do projeto.
- *Sqrt*, a altura de cada bloco é o retorno da função *Sqrt* para o LOC do bloco, cujo domínio está entre o mínimo e o máximo número de LOC, entre os blocos do projeto.
- *Boxplot*⁴⁹, a altura de cada bloco é o retorno da função *Boxplot* para o LOC do bloco, cujo domínio divide-se em faixas de valores dados pelas métricas do *Boxplot* (mínimo, primeiro quartil, mediana, terceiro quartil e máximo) extraídos dos blocos do projeto.

A função de *boxplot* é um pouco diferente e foi citada como uma alternativa interessante no próprio trabalho de Wettel e por isso também foi incluída neste trabalho. As funções mapeiam o domínio para a imagem (que é constituída de valores na unidade da biblioteca Three.js). Os valores para a imagem foram escolhidos de forma arbitrária, num processo experimental, avaliando o resultado de cada conjunto imagem. E isso finaliza o conjunto de configurações de visualização disponíveis para o usuário.

5.2.3 Sistema de Navegação

É essencial que seja possível interagir e navegar na visualização. Por isso, a ferramenta permite que o usuário navegue na cidade de duas maneiras, utilizando o *mouse* ou o teclado. Utilizando-se do *mouse*, o usuário ao pressionar o botão esquerdo pode rotacionar a cena em torno do eixo origem (0,0,0). Utilizando o *scroll* do *mouse*, que pode ser feito através da bola ou com dois dedos em alguns *trackpads*, é possível dar mais zoom ou menos zoom no *canvas*. Além disso, utilizando o botão direito é possível arrastar toda a cena e visão dentro do *canvas*.

Com o teclado, usando as setas direcionais o usuário pode mover a câmera para cima ou baixo e para esquerda ou direita. Além disso, pressionando "+" e "-" é possível dar mais zoom e menos zoom respectivamente. Essa informações ficam em um menu abaixo da visualização quando ela é renderizada.

⁴⁹ https://en.wikipedia.org/wiki/Box_plot

5.3.4 Detalhes do Projeto

Na coluna da direita, abaixo do menu de configurações, ficam os detalhes do projeto, Figura 27. Esse menu só é carregado quando a aplicação executa o arquivo. Nele constam informações gerais sobre o projeto, são elas: Uma legenda da visão, associando a cor a cada tipo de elemento e quantidade desse tipo presente no projeto. Também apresenta a quantidade total de LOC do projeto, assim como o número mínimo e máximo para as métricas, número LOC e NOM, dado todos os blocos do projeto.






Project information		
	Number of Classes: 15	Total project's LOC: 1533
	Number of Structs: 1	Min LOC in a block: 1
	Number of Extensions: 53	Max LOC in a block: 248
	Number of Protocols: 4	Min NOM in a block: 0
	Number of Enums: 7	Max NOM in a block: 32

Figura 27. Menu de informações gerais do projeto. Fonte: O autor.

Esse menu é interessante porque permite o usuário observar dados gerais do projeto e entender a distribuição de seus elementos implementados de acordo com o tipo. Informações como essas podem ser úteis e incentivar *insights* para a manutenção do sistema.

Seleção e detalhamento dos blocos

Essa funcionalidade provê interação entre o usuário e os elementos da visualização em cena. Com o botão esquerdo do *mouse* é possível selecionar um bloco presente na cena para obter mais informações sobre o mesmo. A figura abaixo apresenta o resultado de uma seleção.

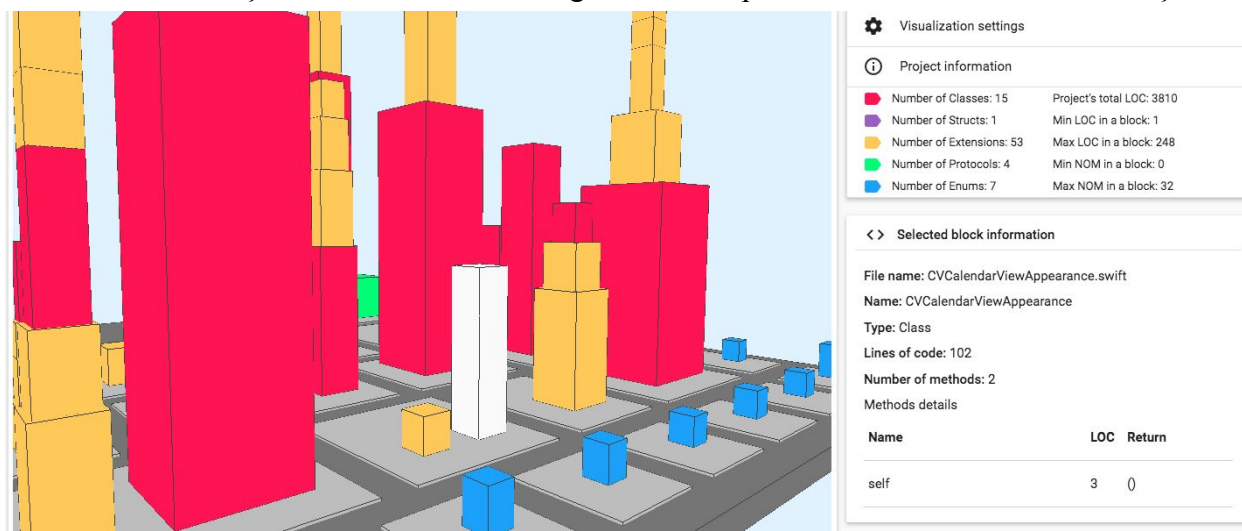


Figura 28. Resultado ao selecionar um objeto em cena. Fonte: O autor.

Até então, na versão atual, ao selecionar um bloco, o mesmo fica evidenciado na cena ao receber a cor branca. Após isso, o menu chamado *Selected block information* exibe um resumo das informações daquele bloco. As informações são: o nome do arquivo que o elemento se encontra, o nome, o tipo, número de LOC, o NOM, e uma tabela com o resumo de informações a respeito dos métodos do bloco selecionado. A tabela apresenta o nome do método, número de LOC e o tipo do retorno do método. Com esse detalhamento é possível entender um pouco mais sobre o que a visão fornece, assim como sobre os elementos do projeto. É possível então questionar a complexidade dos elementos como, por exemplo, um método que seja responsável por quase a totalidade do número de LOC do bloco. Fato parecidos com esse, que podem existir de maneira proposital ou não, são agora, possíveis e fácil de serem identificados.

5.2.5 Screenshot

Além das funções já citadas, que proveem navegação, interação, exibição de informação, entre outros. A aplicação fornece uma função de *screenshot* que é acionada a partir da tecla 'P' do teclado. Quando pressionada, executa uma função do Three.js responsável por recuperar o *buffer*⁵⁰ de desenho da cena, exporta as informações do *buffer* para um arquivo de imagem no formato *png*⁵¹ e abre uma outra aba no navegador com a imagem. Por extrair a imagem direto do *buffer* do renderizador, garante-se uma imagem de alta qualidade, melhor que possíveis funções de *screenshots* normais para navegadores. Essa função é útil caso o usuário queira compartilhar a visualização ou algum aspecto particular observado na cena. As figuras das visões utilizadas neste trabalho foram extraídas utilizando a essa função.

Em suma, esses são os conjuntos de funções e elementos da visualização implementados que tem por objetivo traduzir o sistema em Swift para a metáfora das cidades e fornecer uma plataforma de análise.

⁵⁰ <https://threejs.org/docs/api/renderers/WebGLRenderer.html>

⁵¹ https://en.wikipedia.org/wiki/Portable_Network_Graphics

CAPÍTULO 6

Casos de estudo

Neste capítulo foram selecionados alguns projetos de sistema em Swift para executar a visualização proposta pela ferramenta e avaliar o resultado. Em discussão, três projetos foram escolhidos, são eles: CVCalendar⁵², Alamofire⁵³ e RxSwift⁵⁴. Projetos de código aberto, populares entre desenvolvedores da comunidade, com tamanhos considerados médio, médio e grande, respectivamente.

6.1 CVCalendar

O CVCalendar, criado por Mozharovsky, é um calendário que permite customização visual para iOS 8 ou superior escrito em Swift (2.0). Ele tem 2,307 estrelas e 422 *forks* no GitHub. Na figura abaixo temos a visão geral do projeto carregado na aplicação.

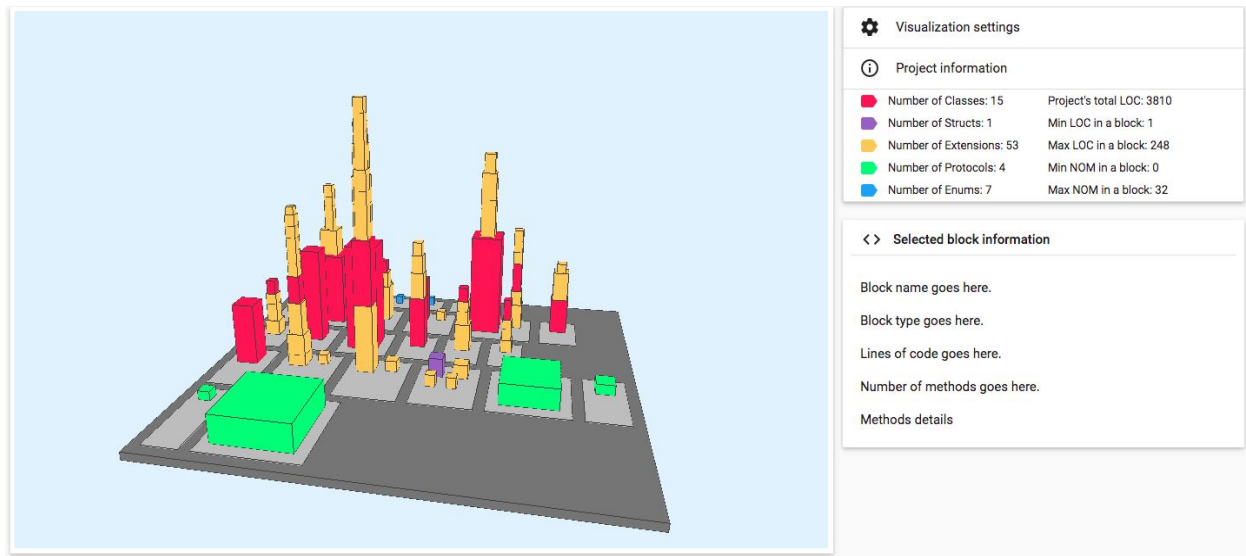


Figura 29. Visão geral da aplicação com o projeto CVCalendar carregado. Fonte: O Autor.

Com o menu de informações do projeto descobrimos que o mesmo tem 3810 linhas de código no total e 80 blocos referentes a implementação dos tipos selecionados para investigação, na adaptação da metáfora. É possível observar facilmente que os *protocols* (em verde) são as implementações que possuem mais métodos, porém poucas linhas de código. Esse comportamento é esperado uma vez que *protocols* apenas possuem as definições dos métodos com os quais os tipos devem entrar em conformidade.

⁵² <https://github.com/Mozharovsky/CVCalendar>

⁵³ <https://github.com/Alamofire/Alamofire>

⁵⁴ <https://github.com/ReactiveX/RxSwift>

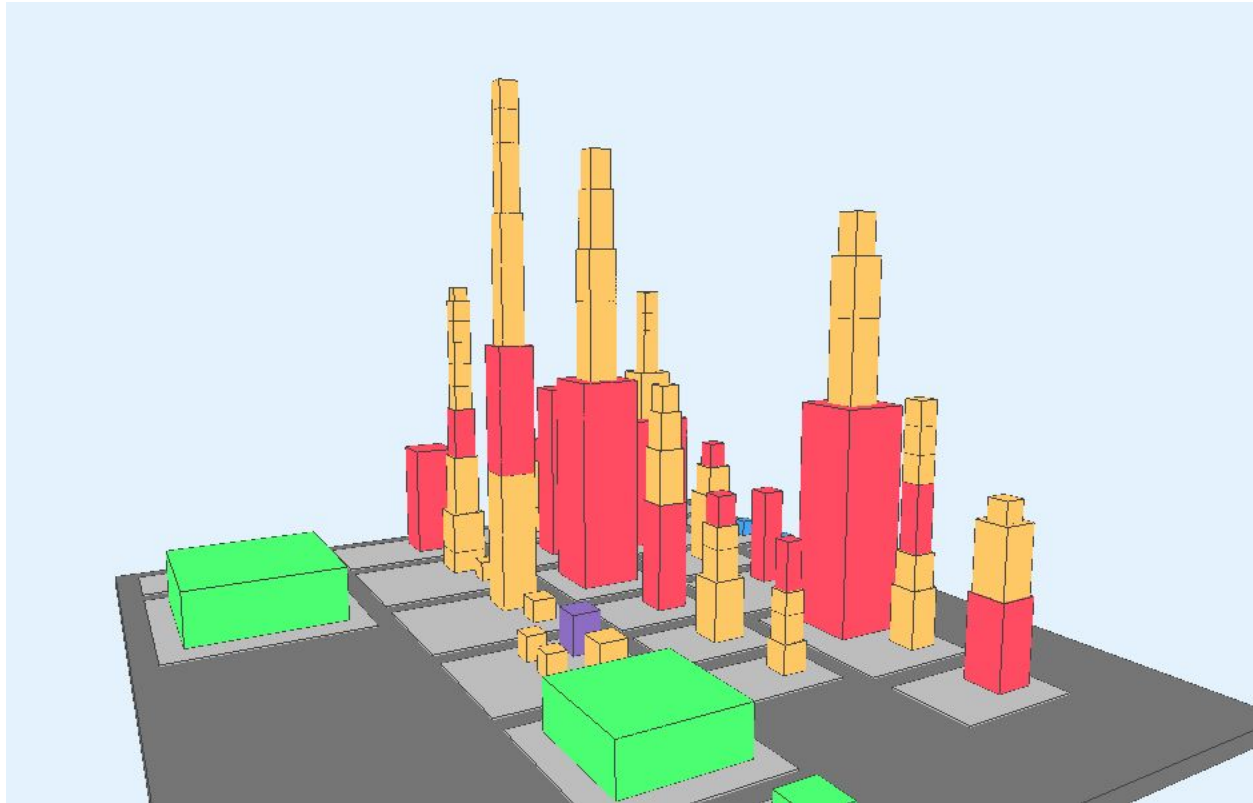


Figura 30. Cidade do projeto CVCalendar. Fonte: O autor.

Analisando o resultado com a visão onde empilham-se os blocos relacionados, é possível observar que a maior construção possui esse tamanho devido a existência das *extensions* que o estendem. Selecionando um dos blocos da maior torre, obtém-se o nome da classe que é CVCalendarDayView. Dada as alturas dos blocos empilhados sobre ela, é possível observar, também, que duas das *extensions* da CVCalendarDayView possuem praticamente a mesma quantidade LOC, sendo a da base tão complexa quanto a própria *class* que ela estende.

Type: Extension			Methods details		
Lines of code: 148					
Number of methods: 5					
Methods details					
Name	LOC	Return	Name	LOC	Return
labelSetup()	39	()	labelSetup()	39	()
preliminarySetup()	8	()	preliminarySetup()	8	()
supplementarySetup()	7	()	supplementarySetup()	7	()
			topMarkerSetup()	33	()
			setupDotMarker()	54	()

Figura 31. Detalhes dos métodos da extension. Fonte: O autor.

Selecionando a *extension*, que é a base da torre, temos no menu detalhes suas informações como mostra a Figura 31. A parte da direita foca nos métodos e com isso nota-se que seus métodos são relacionados à implementação dos *markers* e *labels* da *CVCalendarDayView*. Isso diz, a priori, que essas funcionalidades são as mais complexas da *view*⁵⁵, mas também pode indicar alguma má prática que precise de um refatoramento, dentre outros fatores. Pode-se explorar mais a cidade do *CVCalendarView*, como, por exemplo, observando a segunda maior torre. Ao selecionar a base da torre que representa uma *class* se tem a informação que é a implementação da *CVCalendarWeekContentViewController*. *ViewControllers*⁵⁶ são normalmente mais complexos porque dizem respeito a infraestrutura de gerência de uma *view*. Pode-se fazer perguntas então, como: Por que o *CVCalendarWeekContentViewController* possui menos *extensions* do que o *CVCalendarView*? É necessário refatorar o código? É importante, então, reconhecer que com uma rápida análise do resultado da visualização pôde-se fazer perguntas e levantar discussões sobre a implementação da biblioteca.

6.2 Alamofire

O Alamofire é uma biblioteca para a gerência de requisições HTTP⁵⁷ escrita em Swift. Possui 1117 *commits*, 20786 estrelas e 3589 *forks* no GitHub. A visão da cidade do Alamofire pode ser vista na figura abaixo.

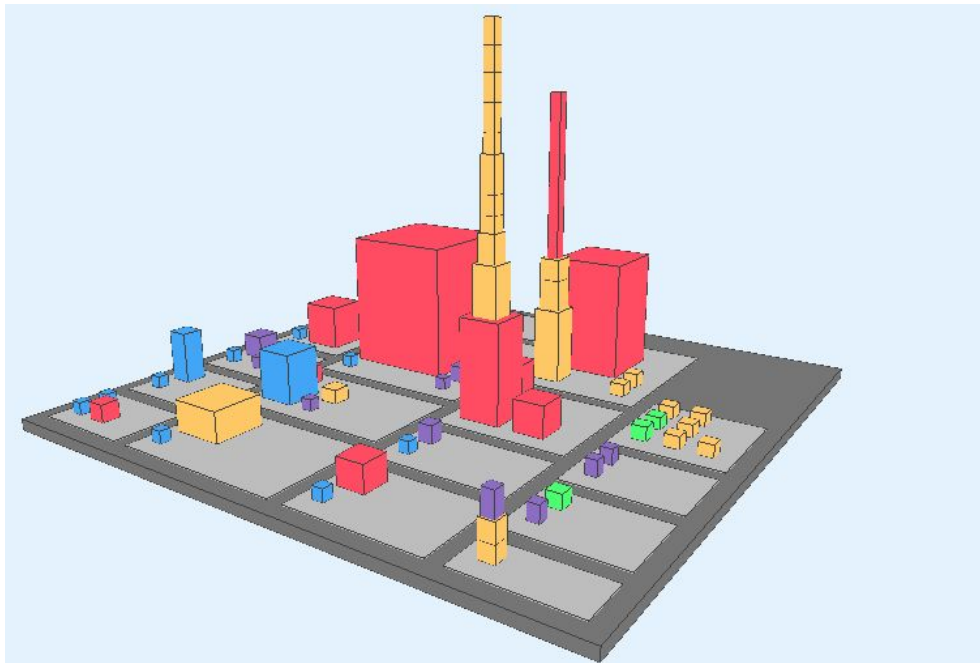


Figura 32. Cidade do Alamofire. Fonte: O autor.

⁵⁵ <https://developer.apple.com/reference/uikit/uiview>

⁵⁶ <https://developer.apple.com/reference/uikit/uiviewController>

⁵⁷ https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

O projeto tem mais de 5434 linhas de código e apesar de ter cerca de 1600 a mais que o CVCalendar, possui 15 blocos a menos, referentes a implementação dos tipos selecionados para investigação pela adaptação da metáfora, com total de 65. É notória a variedade de construções de diferentes tipos, diferentemente do caso de estudo anterior, em que predominam *classes* e *extensions*. Análogo ao análise feita com o CVCalendar, é interessante investigar o porquê da existência dessa quantidade de *extensions* nas duas torres mais altas do projeto. Entretanto, outros fatos interessantes são evidenciados na Figura 33.

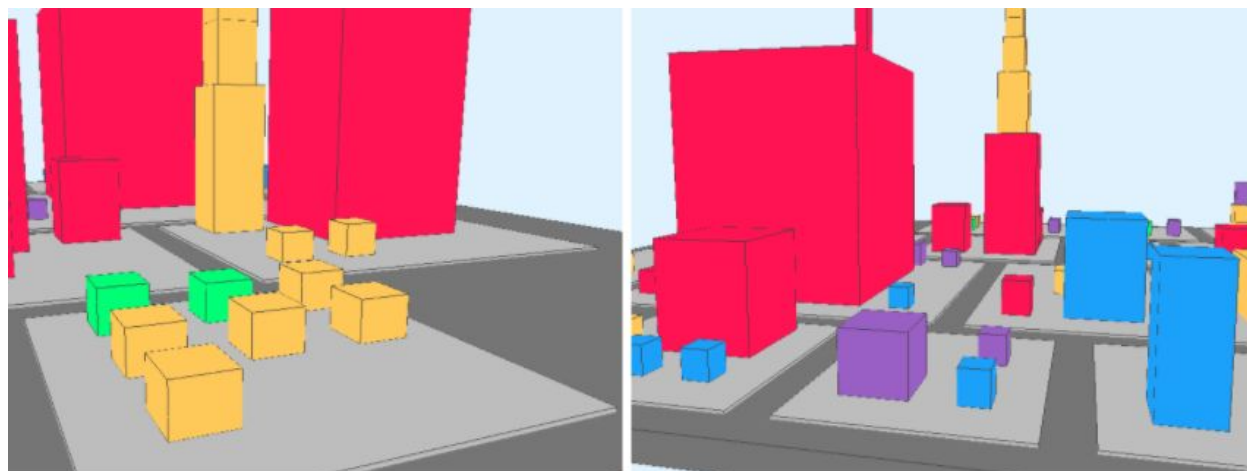


Figura 33. Zoom em duas regiões diferentes da cidade do Alamofire. Fonte: O autor.

O primeiro deles, no lado esquerdo, é a presença de um grupo de *extensions* aparentemente isolada no código. Ao selecionar entende-se que quatro, das cinco, estendem classes relacionadas a requisição a *urls*, por exemplo *NSURL*⁵⁸. Essas classes são de implementação nativa do Swift, explicando o porquê das *extensions* não estarem empilhadas e faz sentido que elas existam (já que se referem a uma biblioteca de conexão HTTP).

No lado direito da figura existem dois *enumerations*, representados por grandes torres azuis. Apesar de *enumerations* poderem conter instâncias de métodos e propriedades pré-computadas, é difícil observar uma implementação desse tipo com cerca de 200 linhas de código. Portanto, tal implementação pode ser alvo de inspeção. Além disso, existem duas *classes* que possuem tamanhos que claramente divergem dos demais elementos da cidade. São elas, *MultipartFormData* e *SessionDelegate*, que são *classes* do *core* da biblioteca, fato que pode justificar o seu tamanho. Ainda sim, são candidatos interessantes a se fazer uma análise mais aprofundada.

⁵⁸ <https://developer.apple.com/reference/foundation/nsurl>

6.3 RxSwift

O RxSwift é a versão Swift da biblioteca Reactive Extensions⁵⁹. Originalmente feita para compor programas assíncronos e baseados em eventos usando sequências observáveis, além de operadores de consulta em estilo LINQ.

No GitHub o projeto tem 2064 *commits*, 6965 estrelas e 960 *forks*. É um projeto constantemente atualizado que conta com 12180 linhas de código no total, 214 *classes*, 12 *structs*, 104 *extensions*, 21 *protocols* e 8 *enumerations*, totalizando 359 blocos. A cidade do projeto pode ser visto na figura abaixo.

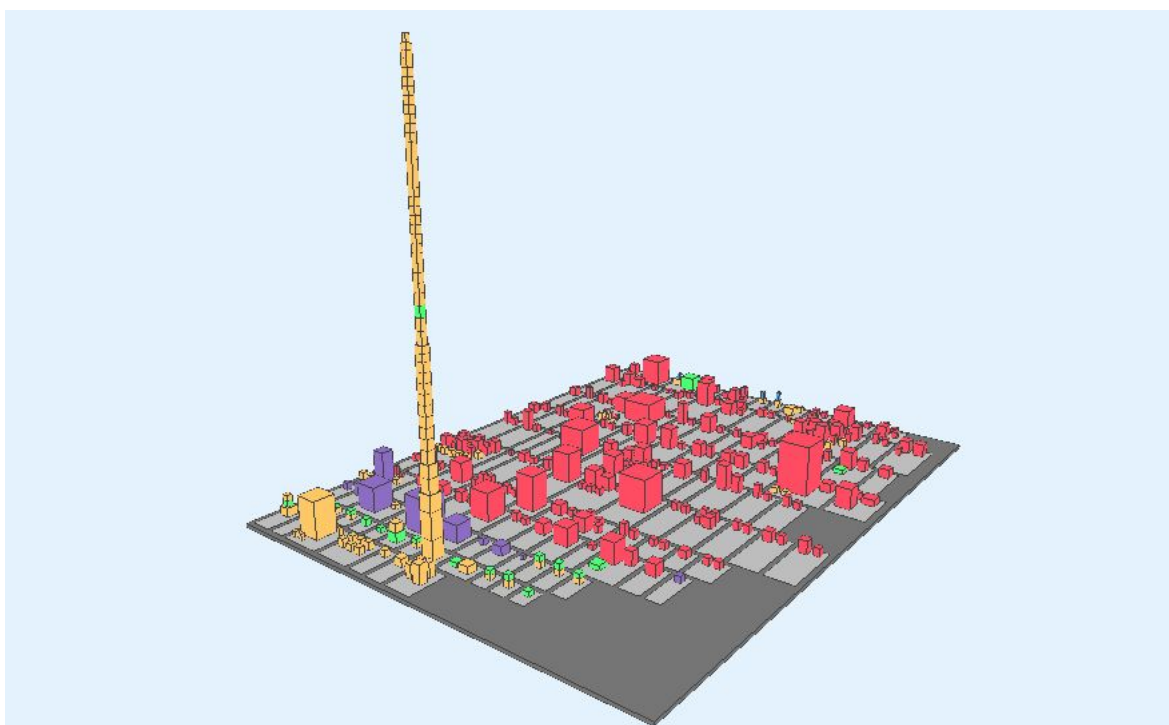


Figura 33. Cidade do projeto RxSwift. Fonte: O autor.

É nítido, apenas olhando para a visão da cidade, que o projeto é consideravelmente maior que os dois exemplos já citados. Outro fato curioso é a grande torre presente na cidade. Ela é exageradamente maior quando comparada aos demais elementos da cidade. A torre é um conjunto de *extensions* que estendem a definição de um *protocol* chamado ObservableType. Esse *protocol* define um novo tipo que é o *core* e o objetivo de toda a biblioteca, isso pode justificar o tamanho e a quantidade de *extensions* nela presente.

⁵⁹ <https://github.com/Reactive-Extensions/Rx.NET>

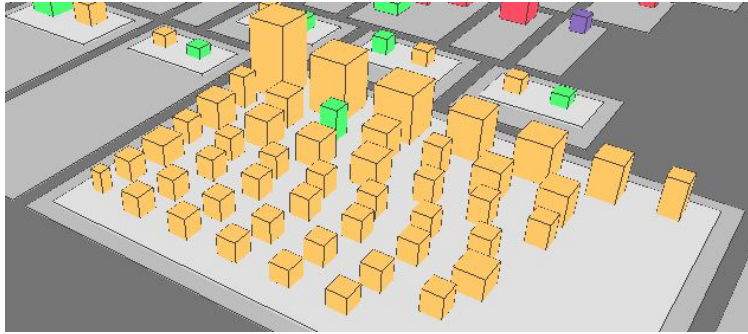


Figura 34. Zoom na cidade do projeto RxSwift com a visão agrupando *extensions* num nível de topologia.

Fonte: O autor.

A figura 34 mostra a mesma definição do tipo `ObservableType`, agora na abordagem que agrupa *extensions* adicionando um nível na topologia. Observando a figura se tem uma noção melhor da quantidade de elementos que compõem essa definição: São 48 blocos no total, 17 a menos que a quantidade total de blocos do projeto Alamofire. A complexidade desse elemento deve-se, provavelmente, à necessidade de entrar em conformidade com inúmeros *protocols* para poder-se definir um tipo, inclusive aos próprios *protocols* desenvolvidos dentro do sistema, que totalizam 20 (excluindo o que define o tipo). Também observa-se uma quantidade bem maior de *classes*, em proporção com o tamanho do projeto, quando comparado com os outros dois exemplos.

Uma dificuldade encontrada ao se analisar cidades de sistemas Swift é perceber a que partes do sistema cada distrito e blocos correspondem, principalmente em se tratando de projetos maiores como o analisado neste exemplo. Por conta da imprecisa hierarquia de sistema, é necessário explorar e navegar na cidade, selecionando elementos, para procurar entender que elementos estão relacionados ao núcleo do sistema, ou a parte de dados, ou de eventos, entres outros.

Os metadados dos projetos analisados nesta seção estão disponíveis *online* no repositório GitHub⁶⁰ do projeto. Em suma, visualizando as cidades das três bibliotecas alguns padrões foram observados, são eles:

1. **Prédios amarelos por toda a cidade:** *Extensions* são amplamente utilizadas nos projetos, provavelmente para desacoplamento e modularização;
2. **Prédios altos contém muitas *extensions*:** Justamente pelo fato apresentado em 1, muitos elementos são visualizados como grande prédios quando a abordagem de empilhamento é utilizada.
3. **Pequenos blocos verdes:** A grande maioria das definições de *protocols* são pequenas e contém poucos métodos. Existe, porém, *outliers* e que são grandes.
4. **Variedade nas cores dos blocos:** Os projetos realmente fazem uso de todas as construções dos tipos padrões fornecidos.

⁶⁰ <https://github.com/peaonunes/swiftcity/tree/master/docs/data>

CAPÍTULO 7

Conclusão

O objetivo deste trabalho foi criar uma ferramenta de visualização de sistemas de *software* escritos em Swift. Fornecendo, então, uma plataforma fácil e intuitiva que possibilitaria extração de conhecimento, *insights*, estudo e auxílio na manutenção dos sistemas.

Inicialmente foi discutido a importância da manutenção de *software*, principalmente com o crescimento da complexidade dos sistemas modernos, e que visualização de *software* é uma alternativa para se tentar mitigar esse problema. Em seguida apresentou-se os fundamentos necessários para entender o que é visualização de *software*, bem como os fundamentos da metáfora das cidades (Wettel, 2007), que foi a utilizada neste projeto. Além disso, justificou-se a escolha de Swift como linguagem alvo, a ser estudada neste trabalho, comentando o cenário atual de crescimento e utilização de Swift, bem como suas particularidades e construções interessantes.

Logo em seguida, discutiu-se sobre trabalhos relacionados, o que já foi e vem sendo feito na área de visualização de *software*. Assim como, as vantagens, desvantagens e dificuldades pertinentes a implementação e construção de cada abordagem. Após essa etapa, explicou-se um pouco do processo da pesquisa, de onde foram retirados os dados para os testes da visualização, bem como as tecnologias necessárias para o desenvolvimento e a razão de escolher cada uma e, por fim, como se deu o processo de desenvolvimento.

Foi apresentada, então, a tradução dos elementos de Swift para a construção da metáfora da cidade. Com decisões que tiveram que ser tomadas para traduzir de maneira fiel os conceitos propostos pela metáfora, bem como algumas adaptações e abordagens escolhidas para que as particularidades de Swift fossem bem representadas. Por exemplo, as abordagens de representação das extensões e seus elementos relacionados. Assim, relatou-se as funcionalidades do sistema, o que impacta na visualização, o que se traduz delas para a metáfora e porque elas são úteis. Como, por exemplo, as opções de escalas diferentes.

Por fim, aplicou-se a visualização em três projetos selecionados, CVCalendar⁶¹, Alamofire⁶² e RxSwift⁶³. Discutindo-se o resultado da visão, procurou-se entender, de maneira breve, as particularidades de cada projeto, analisar possíveis alvos de inspeção de código, manutenção e refatoramento. Buscando também, analisar a distribuição dos elementos que constituem o projeto, o aspecto da visualização e identificar padrões na implementação de sistemas em Swift.

⁶¹ <https://github.com/Mozharovsky/CVCalendar>

⁶² <https://github.com/Alamofire/Alamofire>

⁶³ <https://github.com/ReactiveX/RxSwift>

7.1 Contribuições

De maneira resumida, pode-se dizer que as principais contribuições deste trabalho para a comunidade foram:

1. O estudo sobre diferentes técnicas de visualização de *software*, suas vantagens e desvantagens, limitações e as dificuldades de garantir uma visão que seja: limpa, interessante, navegável, personalizável, selecionável, entre outros.
2. O estudo de quais tecnologias são interessantes para produção de *software web*, principalmente no quesito de visualização. Neste quesito, este trabalho recomenda fortemente o uso de Three.js para implementações de visões em 3D.
3. O mapeamento das características, particularidades e elementos da linguagem Swift para a metáfora das cidades. Sendo uma fonte de estudo ou um ponto de partida para outros trabalhos relacionados ao assunto.
4. Uma aplicação interessante, de fácil acesso, uso e útil, que permite que desenvolvedores do mundo todo visualizem seu sistema de *software* escrito em Swift como uma cidade. Facilita a análise do sistema e de sua implementação, permitindo a extração de conhecimento do programa e geração de *insights* sobre melhoria do mesmo.
5. Uma breve análise sobre três projetos escritos em Swift, de código aberto, que são bastante populares e utilizados pela comunidade desenvolvedora.

7.2 Trabalhos futuros

Baseado-se nos conceitos estudados e dificuldades apontadas neste trabalho, as seguintes propostas são oportunidades de melhoria à implementação atual:

1. Aprofundar o estudo sobre o mapeamento dos elementos de Swift para a metáfora das cidades, visando traduzir mais elementos e maneiras de representar mais informações dos sistemas na metáfora.
2. Integrar a ferramenta atual com a implementada por Marcel Rebouças. Dessa forma, a aplicação seria autossuficiente em gerar os metadados e a visualização.
3. Implementar outras funcionalidades que melhorem a experiência de interação e navegação na cidade, por exemplo:
 - a. Focar exibição em determinado distrito do sistema (ocultando os demais).
 - b. Filtrar elementos da visualização por nome, tipo ou propriedade.
 - c. Buscar elemento do sistema através do nome.
 - d. Realizar *Tag* de um ou mais elementos do sistema para que permaneçam selecionados e não sejam esquecidos durante a navegação.
4. Executar experimentos de análise da eficácia e de usabilidade da ferramenta, contra técnicas de inspeção manual e inspeção utilizando outras ferramentas de análise.

7.3 Considerações Finais

O resultado obtido com o trabalho foi considerado satisfatório pelos envolvidos na pesquisa. Acredita-se que o objetivo de fornecer uma visualização interessante para sistemas em Swift foi atingido com êxito, é conhecido porém, que existem melhorias necessárias e funcionalidades interessantes a se agregar. A ferramenta foi nomeada Swicity pelo autor e o código fonte está inteiramente disponível em um repositório do GitHub⁶⁴, ainda não estando sob uma licença de código aberto.

⁶⁴ <https://github.com/peaonunes/swiftcity>

Apêndice A

Guia de usabilidade

1. O primeiro passo é acessar o sistema através do site:
<https://peaonunes.github.io/swiftcity/>
 - a. O usuário deve estar munido de seu arquivo de metadados gerados pela ferramenta de Marcel.



Figura 1. Barra de navegação.

2. Após isso, o usuário deve procurar seu arquivo clicando em "BROWSE", botão disponível na barra de navegação como mostra a imagem, e acessando o sistema de arquivos do seu computador.
3. Quando o arquivo estiver selecionado o usuário deve apertar no botão "play" ao lado direito do seletor de arquivos.
4. A aplicação carregará o arquivo e exibirá a cidade. Então o usuário poderá interagir com a aplicação utilizando-se dos comandos esclarecidos no menu de controles, figura abaixo.

Basic Controls				
Mouse				
Zoom out	Zoom In	Rotation	Move scene	Select block
Scroll up	Scroll down	Pan and move	Right click	Left click

Keyboard			
Move up/down	Move left/right	Zoom In/Out	Screenshot
UP/Down Arrows	Left/Right Arrows	+/-	P

Figura 2. Menu de controles.

5. A qualquer momento o usuário pode trocar as opções da visualização através do menu de configurações, figura abaixo. A qualquer mudança a visualização será reconstruída. O botão de ajuda exibe explicações a respeito das opções.

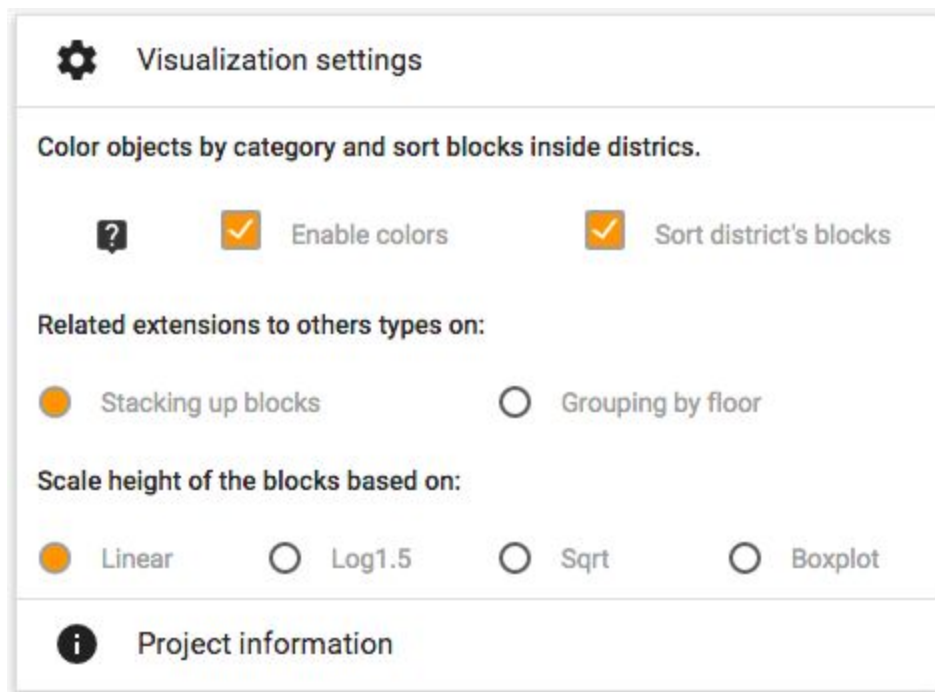


Figura 3. Menu de configurações.

6. O usuário também pode selecionar qualquer prédio da cidade para obter mais informações que serão exibidas no menu de detalhes do bloco.

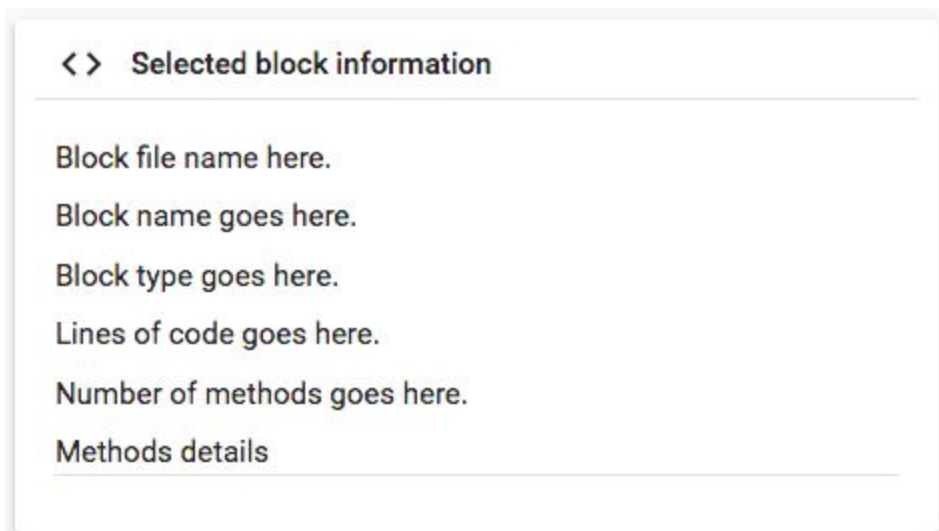


Figura 4. Menu de mais detalhes da seleção.

Referências

- Ball, Thomas, and Stephen G. Eick. "Software visualization in the large." *Computer* 29.4 (1996): 33-43.
- Balzer, Michael, Oliver Deussen, and Claus Lewerentz. "Voronoi treemaps for the visualization of software metrics." *Proceedings of the 2005 ACM symposium on Software visualization*. ACM, 2005.
- Beyer, Dirk, and Ahmed E. Hassan. "Animated Visualization of Software History using Evolution Storyboards." *WCRE*. Vol. 6. 2006.
- Collberg, Christian, et al. "A system for graph-based visualization of the evolution of software." *Proceedings of the 2003 ACM symposium on Software visualization*. ACM, 2003.
- Ducasse, S. and Lanza, M. The class blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, Jan. 2005.
- Eick, S. C., Joseph L. Steffen, and Eric E. Sumner. "Seesoft-a tool for visualizing line oriented software statistics." *IEEE Transactions on Software Engineering* 18.11 (1992): 957-968.
- Fox, Armando, David A. Patterson, and Samuel Joseph. *Engineering software as a service: an agile approach using cloud computing*. Strawberry Canyon LLC, 2013.
- Fyock, Daniel E. *Using Visualization to Maintain Large Computer Systems*. IEEE Computer Graphics and Applications, Los Alamitos, 1997.
- Hindle, Abram, et al. "Yarn: Animating software evolution." *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 2007.
- Lanza, Michele. "The evolution matrix: Recovering software evolution using software visualization techniques." *Proceedings of the 4th international workshop on principles of software evolution*. ACM, 2001.
- Lientz B. P., Swanson E. B. *Software Maintenance Management*. Addison Wesley, Reading, MA, 1980.

Lincke, Rüdiger, Jonas Lundberg, and Welf Löwe. "Comparing software metrics tools." Proceedings of the 2008 international symposium on Software testing and analysis. ACM, 2008.

L. Voinea, A. Telea, and J. J. van Wijk. CVSscan: visualization of code evolution. In Proceedings of 2005 ACM Symposium on Software Visualization (Softviz 2005), pages 47–56, St. Louis, Missouri, USA, May 2005.

Magnavita, Rodrigo Chaves, Renato Lima Novais, and Manoel G. Mendonça. "Using EVOWAVE to Analyze Software Evolution." ICEIS (2). 2015.

Murray, S. .Interactive data visualization for the Web. " O'Reilly Media, Inc.", 2013.

Ogawa, Michael, and Kwan-Liu Ma. "Stargate: A unified, interactive visualization of software projects." 2008 IEEE Pacific Visualization Symposium. IEEE, 2008.

Price, B.A., Baecker, R.M., and Small, I.S. A Principled Taxonomy of Software Visualization, Journal of Visual Languages and Computing 4(3), September 1993.

Stasko, John, ed. Software visualization: Programming as a multimedia experience. MIT press, 1998.

Tamara, M. Visualization Analysis and Design. CRC Press, 2014.

Viana, M., Moraes, E., Barbosa, G., Hora, A., & Valente, M. JSCity – Visualização de Sistemas JavaScript em 3D. Congresso Brasileiro de Software (CBSOFT), Set. 2015.

Waller, J., Wulf, C., Fittkau, F., Dohring, P., & Hasselbring, W. (2013). Synchronis: 3D visualization of monitoring traces in the city metaphor for analyzing concurrency. 2013 First IEEE Working Conference on Software Visualization (VISOFT).
doi:10.1109/vissoft.2013.6650520

Weck, Tobias, and Matthias Tichy. "Visualizing Data-Flows in Functional Programs." 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). Vol. 1. IEEE, 2016.

Wettel, R. and Lanza, M. (2007). Visualizing Software Systems as Cities. 2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis.
doi:10.1109/vissof.2007.4290706

Wilhelm, M. and Diehl, S. Dependencyviewer - a tool for visualizing package design quality metrics. In VISSOFT, 2005.

Zanolie, Kiki, et al. "Switching between colors and shapes on the basis of positive and negative feedback: An fMRI and EEG study on feedback-based learning." *cortex* 44.5 (2008): 537-547.