Pontifícia Universidade Católica de São Paulo - PUC-SP

Carolina Cavalcanti Moya

Diego Martins de Pinho

Hugo Bretz Cabral Hennies

Projeto de TCC: Ambiente para testes e análise de códigos

São Paulo - SP

2013

Carolina Cavalcanti Moya

Diego Martins de Pinho

Hugo Bretz Cabral Hennies

Projeto de TCC: Ambiente para testes e análise de códigos

Trabalho de Conclusão de Curso de alunos de Ciência da Computação da Pontifícia Universidade Católica de São Paulo.

Orientador:

Professor Francisco Supino Marcondes

São Paulo - SP

2013

À minha compreensiva família

Aos meus amigos

A todos os que me apoiaram

*Carolina Cavalcanti Moya*

À minha família

Aos meus amigos

Aos meus companheiros de grupo

*Diego Martins de Pinho*

À minha família

Aos meus amigos

Aos meus companheiros de grupo

Aos meus professores

*Hugo Bretz Cabral Hennies*

**AGRADECIMENTOS**

Carolina Cavalcanti Moya: Agradeço à minha família pelo apoio e compreensão, em especial à minha irmã, que esteve sempre ao meu lado. Aos meus primeiros professores, que me forneceram a base para tudo o que sei hoje. Aos professores do curso, que contribuíram muito para a minha formação. E aos meus amigos, pela motivação e apoio.

Diego Martins de Pinho: Gostaria de agradecer primeiramente à minha família por me proporcionar a oportunidade de estudar em uma excelente universidade e poder construir um bom futuro. A minha companheira, Cayla Ortega, que sempre apoiou e não me deixou desistir. Sou grato a todos os professores que contribuíram e enriqueceram meus conhecimentos durante toda a minha formação acadêmica, em especial aos professores Ítalo Santiago Vega, Fernando Giorno e a professora Lisbete Madsen Barbosa. Ao nosso orientador Francisco Supino Marcondes, que sempre se propôs a ajudar e acreditou no nosso trabalho. Aos meus colegas de classe, que também nos ajudaram muito. E a Pontifícia Universidade Católica pelo fornecimento de materiais e equipamentos.

Hugo Bretz Cabral Hennies: Em primeiro lugar gostaria de agradecer à minha família, que sempre me apoiou, me incentivou e me manteve motivado, além de proporcionar os recursos necessários para a minha educação e para a formação do meu futuro. Agradeço também meus amigos, com quem sempre pude contar, e que me proporcionaram bons momentos dos quais nunca vou esquecer. Sou muito grato também a todos os meus professores, em especial aqueles do curso, pois me proporcionaram conhecimento inestimável que será fundamental na minha experiência daqui em diante. Por último, um agradecimento especial ao nosso professor orientador Francisco Supino Marcondes, por estar sempre à disposição de ajudar e por acreditar em nós e no nosso trabalho.

*“Ciência da Computação está tão relacionada aos computadores quanto a Astronomia aos telescópios, Biologia aos microscópios, ou Química aos tubos de ensaio. A Ciência não estuda ferramentas. Ela estuda como nós as utilizamos, e o que descobrimos com elas.”*  Edsger Dijkstra

**RESUMO**

A proposta deste trabalho é desenvolver um ambiente que auxilie o professor da área de tecnologia da informação no ensino da complexidade de funcionamento de um software para alunos iniciantes na área de desenvolvimento de software. Este ambiente fornecerá ao professor uma maneira de abordar a complexidade de funcionamento de modo que seja possível ensinar o que é necessário dentro do tempo limitado de aula. O ambiente permitirá submeter o código-fonte dos alunos a análises e testes, elaborados pelos professores, que servirão de suporte para ensinar aos alunos que o fato de ter o código compilado, não significa necessariamente que ele esteja funcionando.

**Palavras-chave:** Teste, Complexidade de Funcionamento, Software, Desenvolvimento

**ABSTRACT**

This project aims to develop a programing and learning environment which helps professors to teach the operation complexity of a software system to newcomers to the software development area. By operation complexity, we understand it as the difference between industry grade software and software made by students to understand the constructors of the language. Our environment gives the professor a way to approach the operation complexity subject in the limited class time, by submitting the student’s source code to various object oriented metrics, and professor defined tests, in a way that makes clear that the source code is not intended to be an industry grade code.

**Key Words:** Test, Operation Complexity, Software, Development

**LISTA DE SIGLAS**

CBO– *Coupling Between Objects* (Acoplamento Entre Objetos)

CDU – Casos de uso

DIT – *Depth of Inheritance Tree* (Profundidade da Árvore de Herança)

DOM – *Domain Model* (Modelo de domínio)

ICA – Interface com o Ambiente

IDE – Integrated development enviroment (Ambiente integrado de desenvolvimento)

GOF – *Gang of Four*

LCOM– *Lack of Cohesion in Methods* (Falta de Coesão em Métodos)

NOC– *Number of Children* (Número de Filhos)

PUC – Pontifícia Universidade Católica

RFC – *Response For a Class* (Resposta De uma Classe)

RUP – *Rational Unified Process*

TCC – Trabalho de conclusão de curso

TI – Tecnologia da informação

UML – *Unified Modeling Language*

WMC– *Weighted Methods Per Class* (Métodos Ponderados Por Classe)

XP – *Extreme Programming*

**LISTA DE FIGURAS E TABELAS**

Gráfico 1 - Percentuais por tipos de insucesso no desenvolvimento de software

Tabela 1 – Cronograma de atividades do primeiro semestre de 2013

Tabela 2 – Cronograma de atividades para o segundo semestre de 2013

Figura 1 – Processo ICONIX adaptado

Figura 2 – ICA – Interface com ambiente da aplicação

Figura 3 – DOM – Modelo de domínio da aplicação

Figura 4 – CDU – Casos de uso

Figura 5 – Análise de robustez

Figura 6 – Diagrama de classes

Figura 7 – Diagrama de classes

Figura 8 – Diagrama de classes

Figura 9 - Diagrama de sequência de mensagens

**SUMÁRIO**

1. **INTRODUÇÃO**...................................................................................................12
2. **PROGRAMAÇÃO EXPLORATORIA E ORIENTADA AO APRENDIZADO   
   X PRODUCAO**....................................................................................................14
3. **FUNDAMENTAÇÃO TEÓRICA**..........................................................................18
4. **METODOLOGIA**..................................................................................................22  
   4.1 – **LINGUAGEM JAVA**....................................................................................25  
   4.2- **FERRAMENTAS**...........................................................................................25  
    4.2.1 – Eclipse................................................................................................25  
    4.2.2 – GIT......................................................................................................27  
    4.2.3 – Junit.....................................................................................................27  
    4.2.4 – Gerrit...................................................................................................27  
    4.2.5 – Outras Ferramentas............................................................................28
5. **ITERAÇÃO I**.........................................................................................................29  
   5.1 – **CRONOGRAMA DO PRIMEIRO SEMESTRE**............................................29  
   5.2 – **ATIVIDADES**...............................................................................................30  
    5.2.1 – Análise.................................................................................................30  
    5.2.2 – Desenho..............................................................................................34  
    5.2.3 – Implementação e Testes.....................................................................36
6. **ITERAÇÃO II**........................................................................................................37  
   6.1 – **CRONOGRAMA DO SEGUNDO SEMESTRE**............................................37  
   6.2 – **ATIVIDADES**................................................................................................37  
    6.2.1 – Análise.................................................................................................38  
    6.2.2 – Desenho..............................................................................................44  
    6.2.3 – Implementação e Testes.....................................................................45
7. **RESULTADOS E DISCUSSÃO**...........................................................................46
8. **CONCLUSÃO**......................................................................................................48
9. **REFERÊNCIAS**....................................................................................................49
10. **APÊNDICE A**........................................................................................................51

**1. INTRODUÇÃO**

Um software é um instrumento que automatiza e/ou apoia a realização de atividades humanas através do processamento de informações realizado por meio de uma sequencia finita de instruções. Este processamento é representado através de modelos, uma vez que o software é um elemento intangível. Estes modelos são construídos baseados em uma abstração e, portanto, ignora detalhes que não são importantes para ele.

Os modelos são utilizados para projetar um software e, desta maneira, prever o seu comportamento. No entanto, devido à própria natureza do software, não é possível prever todo o comportamento que ele apresentará diante da realidade.

O mundo real contém inúmeros elementos que interagem com o software, tais como: Interação humana, problemas de hardware, etc. E estes influenciam no funcionamento do software. É necessário que o software esteja preparado para lidar com as influências externas para que possa funcionar de maneira previsível, assim como é preciso que ele seja flexível, para se adaptar a mutabilidade de requisitos e escalável, para poder evoluir.

Para um aluno ingressante na área de desenvolvimento de software, o funcionamento de um software se limita ao fato de ser seu código ter sido compilado, ou não, e ter realizado uma tarefa específica – como realizar a soma de dois números inteiros, por exemplo - com sucesso. Entretanto, esta concepção está errada e cabe ao professor ensinar isto a ele. No entanto, é possível ensinar a complexidade de funcionamento de um software, para um aluno ingressante, dentro do tempo de aula?

O ensino dos fundamentos da engenharia de software - necessários para que seja possível entender a complexidade envolvida no funcionamento de software - dentro ambiente acadêmico, é uma atividade muito limitada para os professores, uma vez que a restrição do tempo de aula e de duração do curso impede que o professor ensine todos os aspectos fundamentais necessários. Cabe ao profissional, escolher cuidadosamente quais serão os principais assuntos que ele irá apresentar aos seus alunos, tornando a ausência de outros obrigatória.

Entretanto, apesar de todo cuidado por parte do professor ao escolher o conteúdo de seu curso, os assuntos que o mesmo decidiu não abordar, acabam se tornando lacunas quando os alunos partem do mundo acadêmico para o mercado de trabalho.

Nosso projeto tem como objetivo a construção de um ambiente que facilite o ensino da complexidade de funcionamento dentro do tempo de aula. Com ela, o professor estará apto a realizar testes e análises nos códigos dos alunos e mostrar que, ao contrário do que os alunos ingressantes têm em mente, o funcionamento do software é algo complexo.

**2. PROGRAMAÇÃO EXPLORATÓRIA E ORIENTADA AO**

**APRENDIZADO X PRODUÇÃO**

O processo de desenvolvimento de um software envolve diversas atividades: desde o levantamento de requisitos, modelagem, seguida da codificação e dos testes, tudo isso iterativamente, até que se tenha obtido uma versão do software que é entregue ao cliente. No entanto, entender os requisitos da aplicação, criar um modelo que atenda a estes requisitos, implementa-lo de forma que ele permita testes de validação e que seja possível fazer a manutenção, é uma tarefa que demanda muito tempo e esforço.

Para auxiliar nestas atividades, utiliza-se a técnica de programação exploratória e programação orientada ao aprendizado.

A programação exploratória é uma das técnicas utilizadas por engenheiros de software para analisar as necessidades do cliente, juntamente com outras técnicas, como o ICA, DOM e CDU; para transforma-las em:

* Requisitos funcionais: Requisitos que restringem o que a aplicação faz.
* Requisitos regulatórios: Requisitos que representam as regras que a aplicação deve obedecer (regras de negócio).

Esta exploração trabalha na construção de um código fonte de forma que este ato de construção seja objeto de análise, pesquisa e estudo. O código resultado desta atividade se caracteriza por ser artesanal, ou seja, construído de acordo com o entendimento e qualificação do programador e/ou da equipe; e monolítico, e por estes motivos não pode ser considerado como o código final da produção de um software.

O código exploratório é um artefato da atividade de análise dentro do processo de desenvolvimento de software. Este artefato sofre alterações ao longo do tempo de estudo e desenvolvimento, isto porque sua essência envolve a incerteza em relação aos requisitos da aplicação.

De acordo com Cristiano R. R. Portella, a incerteza acontece em três níveis: conceitual, funcional e aspectos externos [1].

Este nível de incerteza nos requisitos acontece quando:

* Desenvolvedores não possuem experiência, tanto na área de aplicação do sistema, quanto em desenvolvimento de sistemas em geral.
* Quando o sistema possui um escopo muito amplo.
* Quando o sistema é muito complexo.
* Quando as necessidades do cliente não são claras.
* Quando existe dificuldade em traduzir as necessidades do cliente em requisitos para o sistema.

Unificadamente com a programação exploratória, temos o que definimos como programação orientada ao aprendizado. Este tipo de programação tem como objetivo o aprendizado tanto da tecnologia utilizada (linguagem de programação, framework utilizado, IDE) quanto da lógica de programação. Este tipo de programação é muito comum em cursos básicos de programação, onde o foco é:

* Ensinar a sintaxe da linguagem, ou seja, como usa-la.
* Ensinar como utilizar o ambiente de desenvolvimento.
* Ensinar lógica de programação.
* Ensinar como utilizar os recursos da linguagem para resolver problemas.

Este tipo de programação é utilizada ao mesmo tempo em que se quer resolver um problema, mas o programador não sabe como utilizar a tecnologia para isso. À medida que os requisitos da aplicação são descobertos, a orientação ao aprendizado auxilia nesta passagem da solução em nível conceitual, ou seja, da ideia, para uma solução tecnológica. O principal objetivo desta programação é resolver um problema de forma simples e direta, sem se preocupar com detalhes relevantes sobre o contexto geral da aplicação, como acoplamento entre linhas de código, por exemplo. Assim como a programação exploratória, é um processo inteiramente artesanal e por isso não oferece um código de produção da aplicação.

A programação exploratória, unida a programação orientada ao aprendizado, apresenta diversos problemas quando é utilizada de forma incorreta, por exemplo, como atividade principal na produção de software. Isso por que:

* Mudanças sucessivas tende a produzir um código e, consequentemente, um sistema, cuja estrutura é desorganizada. Consequência disso, a manutenção tende a ser mais difícil e cara.
* As mudanças sucessivas desmotiva uma documentação, uma vez que o sistema é modificado frequentemente.
* Impossibilita o trabalho em grandes equipes.

O Standish Group, desde 1995, realiza estudos relacionados a projetos de software ao redor do mundo. O resultado destes estudos é um relatório chamado *Chaos Report*. Este relatório identifica as principais características dos projetos na área de TI que falham, os principais fatores que levam ao fracasso destes projetos e as ações necessárias para minimizar as possibilidades de falha.

De acordo com o *Chaos Report* de 2011 [2], somente 32% dos projetos são completados dentro do prazo do custo, enquanto 44% deles ultrapassam estimativas de prazo e custo e os demais 24% são cancelados antes de serem completados.

Ainda de acordo com este mesmo relatório, os principais motivos para o fracasso e atraso dos projetos estão representados no gráfico 1.

Gráfico 1 – Percentuais por tipos de insucesso no desenvolvimento de software.

Fonte: Chaos Report, Standish Group, 2011

No gráfico 1 é revelado que grande parte do insucesso no desenvolvimento estão relacionados aos requisitos e ao analfabetismo tecnológico, ou seja, fica notável a necessidade do pleno entendimento das necessidades do cliente e consequentemente dos requisitos (funcionais e regulatórios) da aplicação, assim como da tecnologia que será usada. É necessário saber solucionar os problemas de maneira correta. Para isso, as programações exploratórias e orientadas ao aprendizado formam uma boa solução, pois elas permitem um melhor entendimento do problema a ser tratado e dos requisitos, através de códigos exploratórios, que são estudados e analisados, possibilitando um melhor planejamento, por vez, um bom modelo, que será refletido em um código de produção que atenderá aos requisitos e manutenível.

**3. FUNDAMENTAÇÃO TEÓRICA**

Antes de analisarmos como vamos lidar com a questão de ensinar a complexidade de funcionamento à alunos ingressantes, é fundamental definirmos o que é entendido por complexidade de funcionamento.  
 Um sistema computacional funciona manipulando representações, e representações são sempre formuladas através de modelos. O importante para nós, no entanto, é o fato que de todo modelo lida com seu tema em um determinado nível de abstração, e portanto ignora detalhes que não são essenciais para sua representação [4]. O problema surge quando determinado sistema, que se baseia em modelos limitados e abstratos, entra em contato com o mundo físico, e toda sua infinidade de detalhes. Devido a própria natureza do software, não é possível prever todo o comportamento que ele apresentará diante da realidade. Porem, supondo que fosse possível prever da forma mais minuciosa possível seu comportamento, isso não seria ainda prova suficiente do seu funcionamento.

O funcionamento do software é relativo ao que se é esperado dele, à sua especificação. Esta pode ser diferente para o cliente, para o gerente de projetos e para o engenheiro do programa. De forma geral, exige-se de um software de produção a realização das funcionalidades requisitadas pelo cliente, ou usuário, e adaptabilidade. O software deve ser capaz de adaptar-se a mudanças de requisitos, e ser maleável o suficiente para realizar as funcionalidades requisitadas em qualquer ambiente que seja necessário.

É importante notar também que são esperadas as falhas de um software, elas são inevitáveis, principalmente pela natureza abstrata e limitada do software, no entanto, tais falhas devem ser minimizadas para torna-lo mais confiável em face da especificação.O aluno ingressante, portanto, utilizará nosso ambiente de forma a testar e analisar o software criado por ele de acordo com o que é esperado de forma geral do funcionamento de um software de produção.

Para que possamos analisar a questão de funcionamento utilizaremos o conjunto de métricas OO (orientadas a objeto) apresentadas por Chidamber e Kemerer [3] e um conjunto de testes definidos pelo professor que utilizará o ambiente, característico da atividade dada em questão na sua aula. As métricas possibilitarão medir a adaptabilidade e maleabilidade do código que for analisado com base nelas, pois medem princípios amplamente aceitos e fornecidos por desenvolvedores de software OO experientes [1]. E os testes representarão o que seria a exigência de funcionalidade do cliente em um software de produção.

As medidas apresentadas por Chidamber e Kemerer que servirão como base para a construção do ambiente, são:

* Métrica 1: *Weighted Methods Per Class* (WMC) (Métodos Ponderados Por Classe)

Considere a classe C1, com métodos m1,...mn. Sendo c1,...cn a complexidade estática dos métodos. Então o número de métodos e suas respectivas complexidades indicam quanto tempo e esforço é necessário para desenvolver e manter determinado objeto. Também implicam no grau de potencial impacto de subclasse, uma vez que elas herdarão todos os métodos definidos.



* Métrica 2: *Depth of Inheritance Tree* (DIT) (Profundidade da Árvore de Herança)

O DIT é definido pela altura da classe na árvore de herança, a classe raiz tem DIT = 0, suas imediatas subclasses DIT = 1 e assim por diante. Ele serve como medida de quantas classes ancestrais estão possivelmente influenciando a classe analisada. Quanto maior o DIT, mais métodos serão herdados, mais dependências existirão e portanto maior a complexidade do elemento.

* Métrica 3: *Number of Children* (NOC) (Número de Filhos)

O NOC é definido pelo numero de subclasses imediatas à classe analisada, tem relação com o escopo das propriedades de determinado elemento. É uma medida de quantas classe vão herdar os métodos da classe ancestral. O NOC passa a idéia da possível influencia que a classe tem no design geral, caso ela tenha um NOC alto, seus métodos são mais amplamente utilizados, e portanto devem ter uma importância maior.

* Métrica 4: *Coupling Between Objects* (CBO) (Acoplamento Entre Objetos)

O CBO é definido pelo pela quantidade de dependências que não sejam de herança de determinada classe. Considerando dependência como a relação existente entre dois elementos que agem um sobre o outro, ou seja, utilizam-se dos métodos ou variáveis de instância do outro. Um CBO muito alto é prejudicial ao reuso de código e a um design modular, uma vez que a modificação de determinada classe possivelmente interfere no comportamento de suas classes dependentes. Portanto o CBO deve ser mantido em um mínimo para melhorar a modularidade e promover o encapsulamento.

* Métrica 5: *Response For a Class* (RFC) (Resposta De uma Classe)

Sendo RS o conjunto de resposta de uma classe, definido por RS = { Mi } U {Ri}, onde Mi é o conjunto de todos os métodos da classe, e Ri é o conjunto de todos os métodos que são chamados por Mi. RFC = I RS I, ou seja, o tamanho do conjunto RS. Essa métrica mede a quantidade de métodos que podem ser chamados em determinado objeto, como também os métodos que são chamados por ele em outros objetos, por isso serve também como métrica de comunicação entre objetos. Quanto maior o RFC, maior a complexidade do elemento, sendo necessário mais tempo para desenvolve-lo, entende-lo e testa-lo.

* Métrica 6: *Lack of Cohesion in Methods* (LCOM) (Falta de Coesão em Métodos)

Sendo C1 uma classe com métodos M1, M2,... Mn, e { Ii } o conjunto de variáveis de instância usado pelo método Mi. LCOM = número de conjuntos disjuntos formados pela intercessão dos n conjuntos I ( { Ii },... { In } ). Essa medida fornece um parâmetro para a discrepância entre os métodos de uma classe, com um LCOM alto podemos supor a grande diferença entre métodos, enquanto que com um LCOM alto, podemos supor que os métodos são bastante semelhantes. É desejável uma coesão entre os métodos de uma classe, uma vez que, ela promove o encapsulamento de objetos. Caso tenhamos um LCOM muito baixo, é possível que seja melhor dividir a classe para acomodar melhor cada função que ela representa. [3]

Um software de produção precisa ser confiável, e cumprir as suas funcionalidades sem apresentar problemas para o usuário. No entanto, devido à sua complexidade e mutabilidade, um software nunca deve ser visto como completamente livre de erros. Como não é possível garantir que um software funciona perfeitamente em todas as situações, devemos reduzir o risco de erros. Os testes são uma garantia, de que ao menos aquilo que passou nos testes funciona da maneira especificada por eles.

No entanto, não devemos esquecer de que os testes, assim como o resto do código, ainda podem tem erros, ou falhas de compreensão sobre aquilo que deveria ser testado.

As vantagens de um software devidamente testado ficam mais claras quando é necessário lidar com softwares maiores, e consequentemente mais complexos, e com mudanças em um código entregue, para acrescentar, modificar ou corrigir algo.

Alunos iniciantes não tem experiência e dificilmente adquirem ela em experiências práticas ao longo do curso. O intuito da funcionalidade de testes é apresentar e incentivar o uso de testes aos alunos desde o início de seu aprendizado, apontando aos alunos a frequência em que erros aparecem mesmo em softwares pequenos, feitos para uso didático.

A funcionalidade de métricas e testes juntas complementam uma à outra, e tornam o ensino de ambas mais eficiente. Enquanto os testes indicam ao aluno que é necessário garantir que um software faz o que devia fazer, as métricas apontam que só isso não é suficiente, e que para um software de produção estar bom, é necessário produzi-lo de maneira estruturada, organizada de modo a facilitar futuras alterações.

Por ser uma ferramenta para o apoio de ensino a alunos iniciantes, que geralmente produzem códigos simples, e devido à restrição de tempo, foi decidido que os testes oferecidos se limitarão a métodos simples, que recebam de entrada e tenham como saída apenas parâmetros dos tipos definidos pela linguagem Java.

**4. METODOLOGIA**

Existem vários processos que podem ser utilizados para o desenvolvimento de software, como: *Rational Unified Process* (RUP) [5], *Extreme Programming* (*XP*) [7], ICONIX.

O processo ICONIX foi o escolhido para orientar o projeto, por ser mais adequado em termos de documentação do que o processo RUP. O RUP exige uma documentação extensa, que se tornaria inviável, dado o tamanho da equipe. Outro motivo para a escolha do ICONIX é que ele não exige uma equipe altamente especializada como o XP, uma vez que os membros da equipe, como estudantes, não possuem essa experiência. O ICONIX está entre esses processos, visando o equilíbrio entre o planejamento e a objetividade. O objetivo dele é ser o mais objetivo possível ainda suficiente para atender as necessidades do projeto com eficiência [8].

Os artefatos a serem produzidos e utilizados são a programação exploratória (PXP), a interface com o ambiente (ICA), o modelo de domínio, os modelos de casos de uso, o diagrama de classes, o diagrama de sequências, o código e unidades de teste.

O processo ICONIX divide o modelo em duas partes: estática e dinâmica. Na parte estática ficam os modelos de domínio e diagramas de classe, e na parte dinâmica, os casos de uso e os diagramas de sequência [8].

O processo define um ciclo de vida iterativo incremental, o que possibilita alterações no projeto durante o seu desenvolvimento e um refinamento sucessivo. A cada iteração são executadas as mesmas atividades, divididas em quatro estágios: análise de requisitos, análise e design preliminar, design detalhado e implementação. No fim de cada estágio existe uma *milestone*, para a verificação do que foi realizado até um dado momento [8].

A seguir são detalhados os estágios, com as atividades executadas em cada um deles.

Análise de requisitos:

* Identificar objetos de domínio do mundo real – Nessa etapa são feitos ICAs, PXP e o modelo de domínio. São identificados objetos de domínio do mundo real, e interações entre eles, dentro do escopo do problema que o projeto pretende atender. O modelo de domínio servirá como base para o desenvolvimento do projeto. Ele é constantemente atualizado, conforme o maior entendimento do problema [7].
* Definir requisitos comportamentais – Os requisitos comportamentais nessa fase são representados através dos casos de uso, portanto nessa etapa são elaborados os casos de uso, organizados em grupos, e descrições para cada caso de uso [8].

Análise e design preliminar:

* Fazer análise de robustez – O objetivo da análise de robustez é eliminar casos de ambiguidade nos casos de uso e identificar possíveis distanciamentos entre os casos de uso e o modelo de domínio. Para isso é feito um design preliminar, a partir dos casos de uso obtidos anteriormente, são identificados objetos necessários que não estão no modelo de domínio, e então o modelo de domínio é atualizado. Caso haja ambiguidade, os textos dos casos de uso também são atualizados [8].

Design detalhado:

* Alocar comportamentos para os objetos – Essa etapa marca o início do design. O principal diagrama a ser desenvolvido aqui é o diagrama de sequência, mas podem ser desenvolvidos outros, conforme a necessidade, como o diagrama de atividades ou diagrama de estados. Cada caso de uso da origem a um diagrama de sequência [8].
* Concluir o modelo estático – O modelo estático é um detalhamento do modelo de domínio, com diagramas de classes detalhados, com um nível mais baixo de abstração, e que reflitam os diagramas de sequência. O modelo estático deve conter informações detalhadas do design, e satisfazer todos os requisitos encontrados [8].

Implementação:

* Escrever o código – Nessa etapa é feita a implementação do sistema a partir do design que foi realizado anteriormente. São desenvolvidos e executados testes para avaliar o funcionamento do sistema e apontar erros [8].
* Teste de desempenho do sistema e de aceitação do usuário – O sistema é testado com base nos casos de uso, para a avaliação de desempenho e aceitação do usuário [8].

A figura 1 representa os principais estágios, atividades e artefatos envolvidos no processo ICONIX.

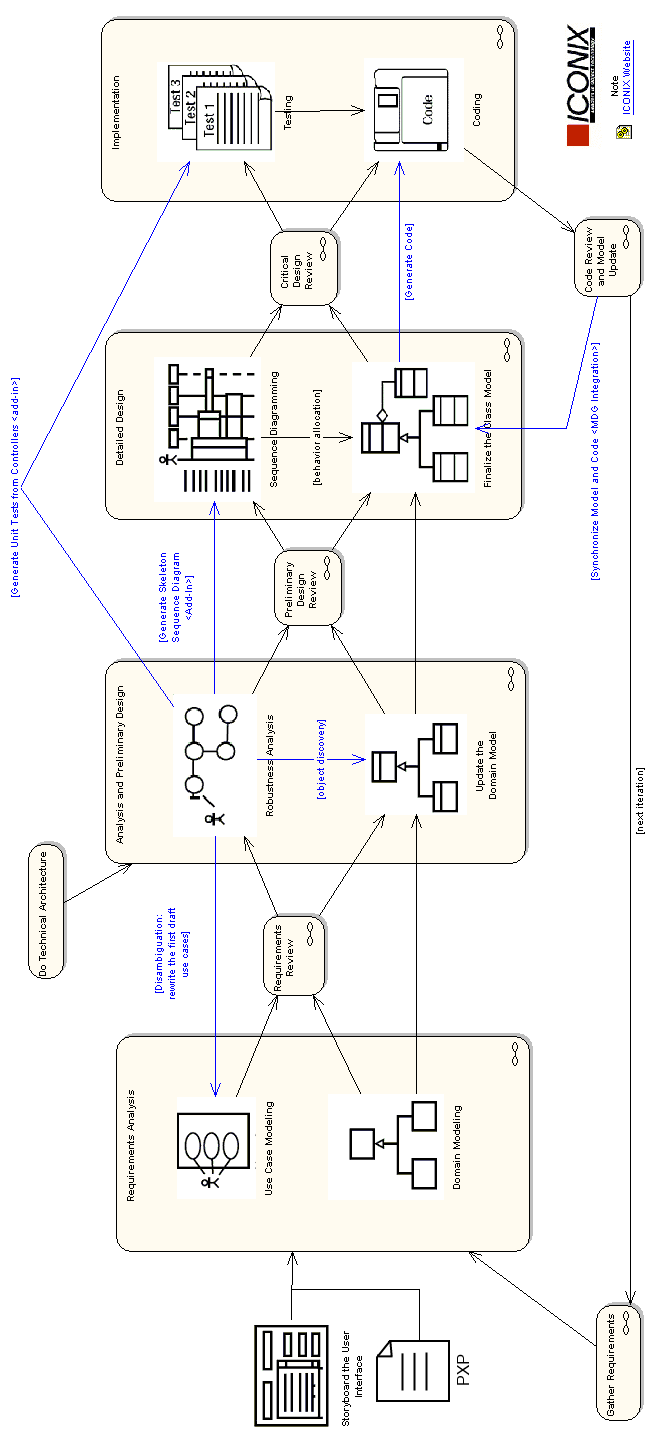


Figura 1 - Processo ICONIX adaptado [8]

**4.1 – LINGUAGEM JAVA**

O paradigma escolhido para o desenvolvimento do projeto é o paradigma orientado a objetos, devido ao processo escolhido, que é iterativo incremental. Java foi a linguagem escolhida para a implementação do projeto, por ser a linguagem de maior domínio de todos os membros, e por ser uma linguagem adequada para programação orientada a objetos.

**4.2 - FERRAMENTAS**

Ao longo do desenvolvimento deste projeto foram utilizadas diversas ferramentas, todas disponíveis gratuitamente na internet. Foram utilizadas ferramentas para gerenciamento de projeto, desenvolvimento (ambiente), diagramação UML, revisão e controle de versão de código.

**4.2.1 - Eclipse**

Eclipse é uma IDE para desenvolvimento Java, sendo expansível, através de plug-ins para outras linguagens de programação, tais como: C/C++ e Phyton. O mesmo foi desenvolvido pela empresa IBM, com um orçamento inicial de 40 milhões de dólares. O Eclipse foi escrito na linguagem Java e atualmente é um projeto *open-source*. Atualmente, o Eclipse é a IDE Java mais utilizada do mundo, competindo com IDE’s como NetBeans e IntelliJ.

Escolhemos o Eclipse pela sua facilidade de uso, familiaridade com a sua interface e funcionalidades, e pela possibilidade de expandi-lo utilizando plug-ins. Durante o desenvolvimento deste projeto, foram utilizadas duas versões: Juno (4.2) e Kepler (4.3). Também foram utilizados três plug-ins na plataforma:

1. EGit: Plug-in para integração com o GIT. Permite que *merges*, *commits*, *fetchs* e outras funcionalidades do GIT sejam feito direto do Eclipse, fazendo desnecessário o uso do terminal.
2. EClemma: Plug-in para cobertura de código com teste unitários. Mostra a quantidade de código que foi coberto pelos códigos unitários. Isso permite ter uma melhor ideia dos lugares do código que ainda não foram testados.
3. JUnit Plug-in Test Launcher: Plug-in para facilitar escrever e realizar testes unitários utilizando o JUnit.

**4.2.2 - GIT**

GIT é um sistema de controle de versão distribuído e um sistema de gerenciamento de código-fonte, onde não existe um repositório central e cada cópia do projeto se torna um repositório local ou remoto. O GIT foi inicialmente projetado e desenvolvido por Linus Torvalds, criador do Linux. O GIT foi escolhido pela característica de não existir um repositório central. Isso permitiu que o trabalho fosse desenvolvido em diversas máquinas de modo separado e posteriormente, unificado em um só. Para facilitar o uso do GIT, foi utilizado o GitHub.

**4.2.3 - JUnit**

O JUnit é um framework *open-source*, criado por Erich Gamma e Kent Beck, com suporte à criação de testes automatizados na linguagem de programação Java. No entanto, atualmente, foi adaptado também para outras linguagens de programação, tais como: C#, Phyton, Fortran e C++.

Este framework permite a criação de testes unitários, ou seja, pequenos testes que testam o menor dos componentes de um sistema de maneira isolada. Escolhemos o JUnit por ser um consagrado framework para teste muito fácil de se utilizar. Unificadamente com o plug-in Eclemma, permitiu que cobríssemos grande parte do nosso código fonte com testes. Foi utilizada a versão 4.x.

**4.2.4 – Gerrit**

O Gerrit é uma ferramenta web gratuita, focada em equipes, para revisão de código. Cada um dos integrantes da equipe pode rever e comentar as modificações de cada um no código-fonte e por meio de um navegador, aprovar ou rejeitar essas modificações. Ele foi criado no Google por Shawn Pearce durante o desenvolvimento do sistema operacional Android.

O Gerrit foi adotado, pois ele permite ver as modificações no código antes que ele pudesse ser enviado para o repositório. Outra vantagem levada em consideração é que ele se integra ao GIT, o que facilitava ainda mais ou controle de versão. Foi utilizada a versão 2.7.

**4.2.5 – Outras ferramentas**

Durante o desenvolvimento deste projeto também foram utilizadas outras ferramentas:

* Dropbox: serviço para armazenamento e partilha de arquivos.
* Trello: ferramenta colaborativa para gerenciamento de projetos.
* Cacco: ferramenta de desenho online para criação de diagramas.
* Astah: software para modelagem UML.
* Google Drive: serviço de armazenamento e sincronização de arquivos.
* Pencil Project: ferramenta para a criação de protótipos de interface.

**5. ITERAÇÃO I**

A primeira iteração ocorreu durante o primeiro semestre do ano de 2013.

**5.1 - CRONOGRAMA DO PRIMEIRO SEMESTRE**

Para o primeiro semestre, foi seguido o seguinte cronograma:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Atividades** | **Fev.** | **Mar.** | **Abr.** | **Mai.** | **Jun.** |
| **Formulação do projeto** |  |  |  |  |  |
| **Leitura Bibliografia** |  |  |  |  |  |
| **Redação da 1ª versão do texto** |  |  |  |  |  |
| **Análise do código** |  |  |  |  |  |
| **Design do código** |  |  |  |  |  |
| **Implementação** |  |  |  |  |  |
| **Apresentação parcial e discussão dos resultados** |  |  |  |  |  |

Tabela 1 – Cronograma de atividades do primeiro semestre de 2013

O primeiro semestre foi caracterizado por uma grande fase de pesquisas. Durante os três primeiro meses de projeto, foi realizada uma grande pesquisa por referências bibliográficas dentro do tema de complexidade de funcionamento. Também foi feita uma grande pesquisa relacionada às métricas OO.

A partir do quarto mês, iniciou-se a etapa de análise. Nesta etapa foram elaborados diversos artefatos: códigos experimentais (PXP), diagramação e especificação de casos de uso, diagramas de robustez e ICAs. Todos estes artefatos nos auxiliaram a elaborar a lista de requisitos da aplicação.

A partir da elaboração da lista de requisitos, o projeto partiu para a fase de design. Nesta fase, foram discutidas diversas estratégias para montar a aplicação da maneira mais flexível possível. Para isso, foram elaborados mais artefatos: diagramas de classe e diagramas de sequência de mensagens. A fase de implementação foi caracterizado pela construção do código e dos testes.

**5.2 – ATIVIDADES**

Na primeira iteração, o principal objetivo foi implementar a funcionalidade de métricas. Para que isso fosse possível, passamos por todas as etapas sugeridas pelo processo ICONIX, produzindo todos os artefatos: dos casos de uso até os diagramas de sequência.

**5.2.1 – Análise**

* Interface com ambiente (ICA)

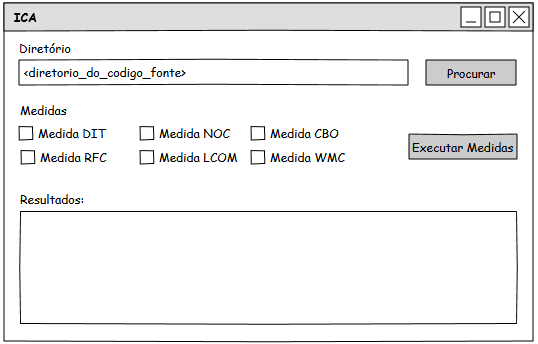


Figura 2 – Interface com ambiente da aplicação

A construção do ICA nos deu uma ideia das funcionalidades básicas da aplicação e de como aconteceria o fluxo de informações. Também foi possível perceber algumas limitações.

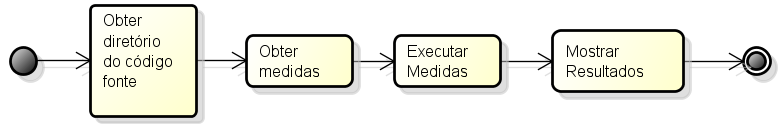


Figura 3 – Fluxo básico da aplicação

* Modelo de Domínio (DOM)

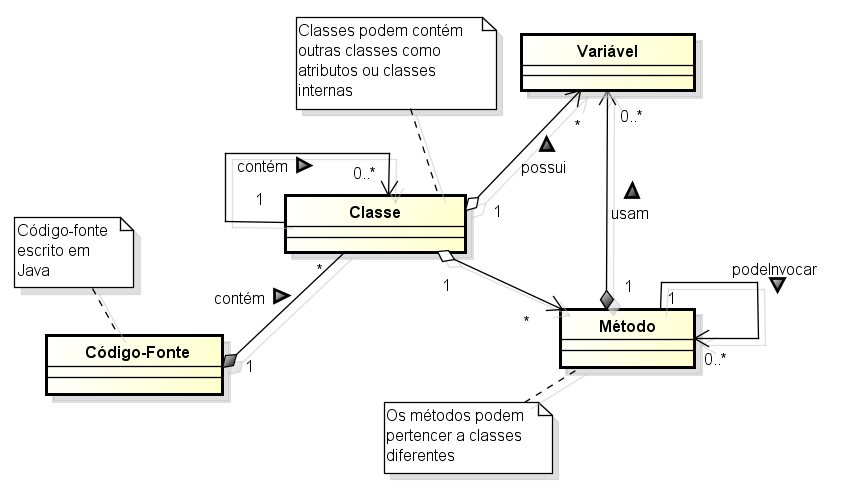


Figura 4 – Modelo de domínio da aplicação

O modelo de domínio deu uma ideia inicial das possíveis classes que poderiam ser implementadas. Como se trata de uma linguagem específica que estamos tratando no trabalho, o modelo domínio auxiliou a enxergar as particularidades da linguagem Java (estrutura).

* Casos de Uso (CDU)

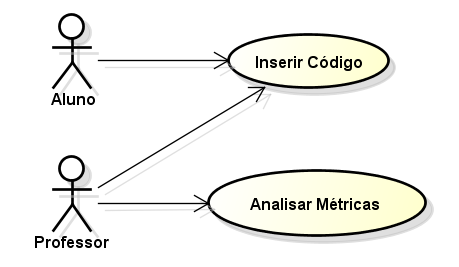


Figura 5 – Casos de uso

Baseado no ICA, foram identificados dois atores e dois casos de uso. Os atores são o Aluno e o Professor. Os principais casos de uso identificados foram: inserir código e analisar métricas.

Inserir Código – versão 1

|  |  |
| --- | --- |
| Atores | Professor ou Aluno |
| Pré-Condição | Ter código compilável escrito em Java |
| Fluxo de Eventos  Básico | 1. Clicar no botão “Inserir Código”  2. Selecionar o diretório do código-fonte  3. Clicar em “Ok” |
| Fluxo de eventos secundário | O ator pode cancelar a entrada de código a qualquer momento. |
| Pós-Condição | \_ |

Analisar Métricas – versão 1

|  |  |
| --- | --- |
| Atores | Professor |
| Pré-Condição | Ter um código escrito em Java inserido no programa |
| Fluxo de Eventos  básico | 1. Selecionar medidas que deseja realizar no código  2. Clicar no botão “Analisar”  3. Ver os resultados |
| Fluxo de eventos secundário | O ator pode selecionar qualquer medida e realizar várias análises |
| Pós-Condição | \_ |

* Análise de Robustez

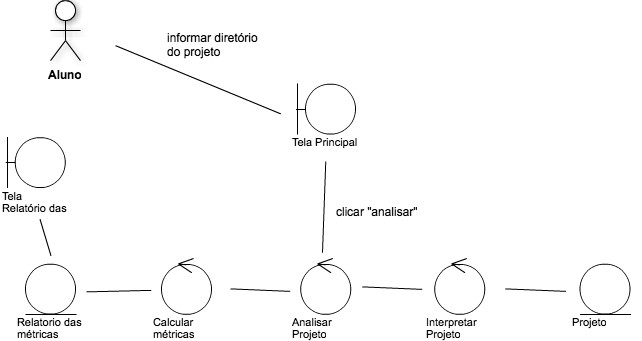


Figura 6 – Análise de robustez

O diagrama de robustez foi criado por recomendação do processo ICONIX com o objetivo de captar os principais objetos e respectivas relações de comunicação estabelecidas entre os mesmos.

* Lista de Requisitos

A partir dos artefatos produzidos, foi obtida a seguinte lista de requisitos:

|  |  |
| --- | --- |
| Requisito | Especificação |
| R1 | A aplicação deve permitir selecionar qual o código deve ser analisado. |
| R2 | O código que vai ser analisado deve estar escrito em Java. |
| R2 | A aplicação deve permitir quais as medidas devem ser realizadas. |
| R3 | A aplicação deve ser capaz de analisar o código selecionado, utilizando as medidas selecionadas e mostrar os resultados. |
| R4 | A aplicação deve oferecer pelo menos seis medidas diferentes: WWC, DIT, NOC, CBO, RFC e LCOM. |
| R5 | Todos os resultados das métricas são representados por números naturais |
| R6 | A aplicação deve realizar as métricas de acordo com as especificações apresentadas por Chidamber e Kemerer. |

Tabela 2 – Especificação dos requisitos da primeira iteração

Neste momento, não houve preocupação em separar os requisitos em categorias (funcional e não funcional), mas sim em identifica-los. Uma vez obtida à lista de requisitos, iniciou-se a atividade de desenho.

**5.2.2 – Desenho**

* Diagramas de Classes (DCL)

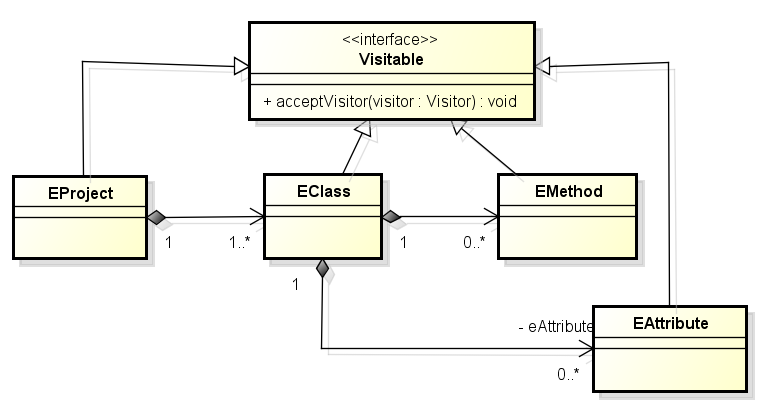


Figura 7 – Diagrama de classes

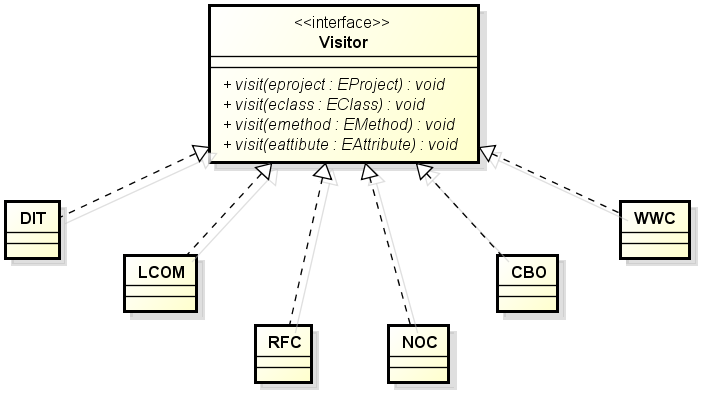


Figura 8 – Diagrama de Classes

Para realizar as métricas, é necessário interagir com os elementos contidos nas entidades (Classe, Atributo, Método). A realização de operações sobre os elementos da estrutura de um objeto caracteriza um padrão de projeto *Gang of Four* (GoF), da categoria de padrões comportamentais, o padrão Visitor [11].

Para utilizar este padrão no projeto, foi criada uma interface Visitor que é realizada por todas as classes de medidas. Para que essas classes de medidas pudessem realizar operações nas entidades, foi criada uma interface com o método que aceita um Visitor. Uma vez que essas classes estendem a interface, ela permite a “visita” de uma medida que realizará as operações necessárias para fazer a medida.

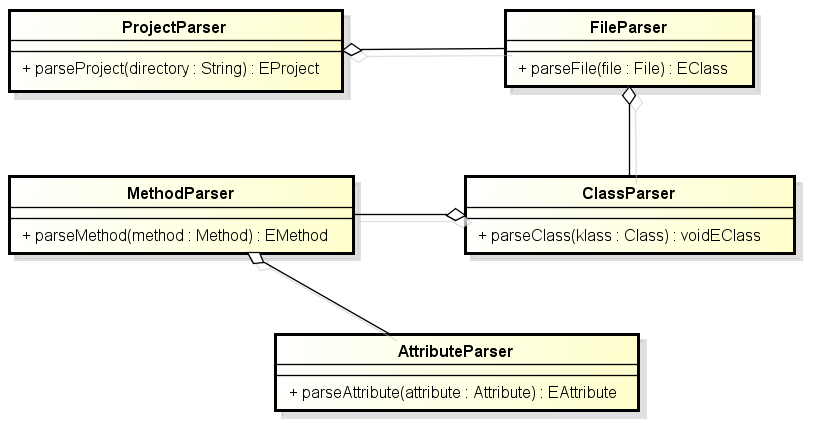


Figura 9 – Diagrama de classes

Como o modelo de domínio revelou que os elementos estavam contidos um dentro do outro, foi pensada uma estratégia de delegação para realizar os *parsers*. Nesta estratégia, cada um dos *parsers* delega o processamento da entidade que está no nível abaixo dela. Uma vez que a delegação chega ao nível mais baixo, ela vai subindo até chegar ao primeiro nível novamente com todas as informações processadas.

* Diagrama de sequência de mensagens (DSM)

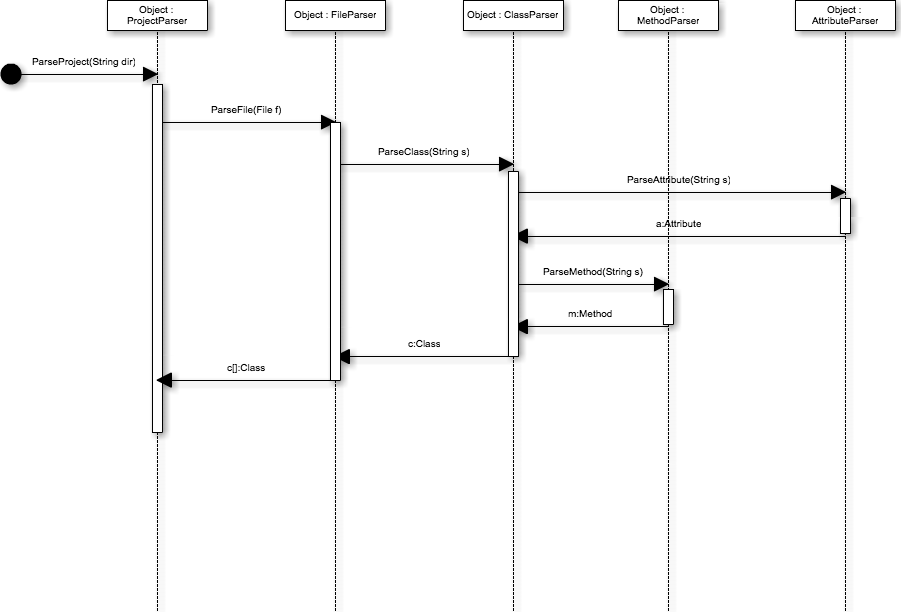


Figura 10 – Diagrama de sequência de mensagens

**5.2.3 - Implementação e Testes**

A codificação e os testes unitários foram realizados separadamente e anexados a este documento. O código PXP produzido para auxiliar na busca de requisitos também está anexado a este documento, juntamente com a biblioteca cjkm utilizada como apoio [9]. O ambiente FitNesse começou a ser estudado para ser utilizado futuramente também para testes [10].

**6. ITERAÇÃO II**

A segunda iteração aconteceu durante o segundo semestre do ano de 2013.

**6.1 - CRONOGRAMA DO SEGUNDO SEMESTRE**

Para o segundo semestre de 2013, temos o seguinte cronograma:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Atividades** | **Jul.** | **Ago.** | **Set.** | **Out.** | **Nov.** | **Dez.** |
| **Redação da 2ª versão do texto** |  |  |  |  |  |  |
| **Revisão do texto** |  |  |  |  |  |  |
| **Continuação Desenvolvimento do código** |  |  |  |  |  |  |
| **Apresentação final e discussão dos resultados** |  |  |  |  |  |  |

Tabela 3 – Cronograma de atividades para o segundo semestre de 2013.

O foco principal do segundo semestre foi a especificação e implementação da funcionalidade de testes e integração com a funcionalidade de métricas. Todas as atividades realizadas na primeira iteração foram repetidas.

**6.2 – Atividades**

Na segunda iteração, o principal objetivo foi implementar a funcionalidade de testes, para complementar a primeira funcionalidade de métricas.

**6.2.1 – Análise**

* Interface com ambiente (ICA)

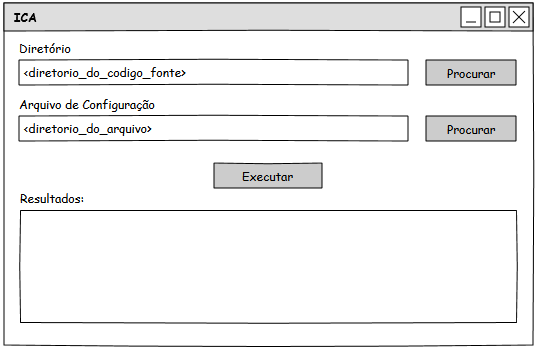


Figura 11 – Interface com ambiente da aplicação

A atualização do ICA ajudou a identificar um novo comportamento. Ao invés de selecionar as métricas, a aplicação deve receber um arquivo de configurações que contém as métricas selecionadas e as informações necessárias para os testes.

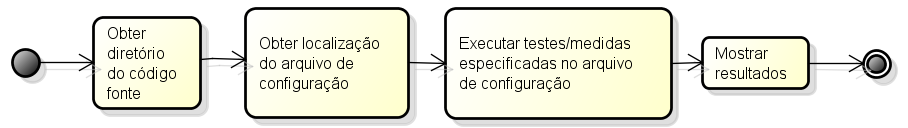
****

Figura 12– Fluxo básico da aplicação baseado no ICA

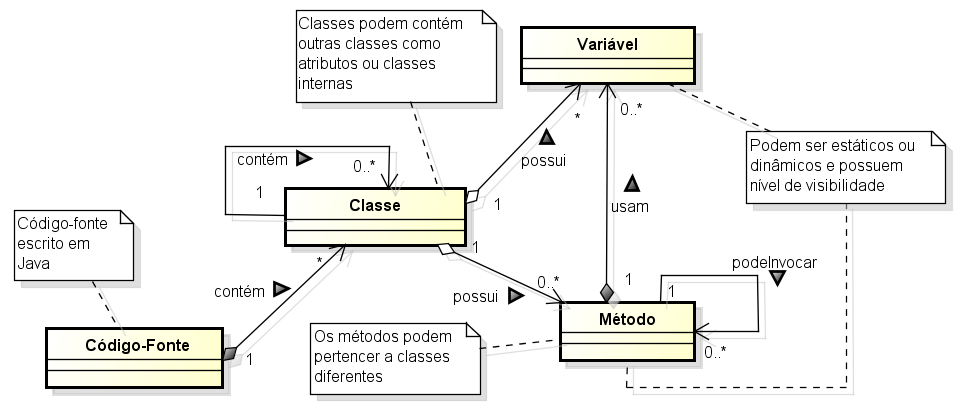
* Modelo de Domínio (DOM) 

Figura 13 – Modelo de domínio da aplicação

Da primeira para a segunda iteração, o modelo de domínio sofreu poucas alterações.

* Casos de Uso (CDU)

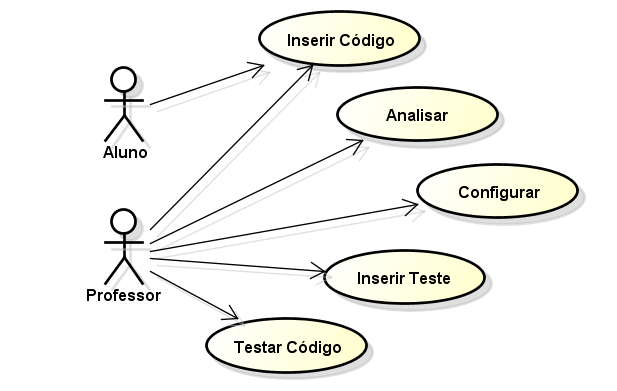


Figura 14 – Caso de uso

A adição da funcionalidade de teste adicionou mais casos de uso a aplicação, mas os atores continuaram os mesmos que foram identificados na primeira iteração.

Descrição dos casos de uso:

Inserir Código – versão 2

|  |  |
| --- | --- |
| Atores | Professor ou Aluno |
| Pré-Condição | Ter código compilável escrito em Java (\*.java) |
| Fluxo de Eventos  Básico | 1. Clicar no botão “diretório do projeto”  2. Selecionar o diretório do código-fonte  3. Clicar em “Ok” |
| Fluxo de eventos secundário | O ator pode cancelar a entrada de código a qualquer momento. |
| Pós-Condição | \_ |

Analisar – versão 2

|  |  |
| --- | --- |
| Atores | Professor |
| Pré-Condição | 1. Ter um código escrito em Java inserido no programa  2. Ter um arquivo de configuração inserido no programa |
| Fluxo de Eventos  Básico | 1. Clicar no botão “Analisar”  2. Ver resultados |
| Fluxo de eventos secundário | O ator pode selecionar qualquer medida e realizar várias análises |
| Pós-Condição | \_ |

Configurar – versão 1

|  |  |
| --- | --- |
| Atores | Professor |
| Pré-Condição | Ter escrito um código de configurações |
| Fluxo de Eventos  Básico | 1. Clicar no botão “Buscar Arquivo”  2. Selecionar o arquivo de configurações no file system  3. Clicar em “Ok”  4. Receber a mensagem de confirmação |
| Fluxo de eventos secundário | 1. O ator pode cancelar a entrada de um arquivo de configurações a qualquer momento.  2. Se acontecer algum problema na hora de inserir o arquivo de configuração no programa, ele será exibido no passo 4. |
| Pós-Condição | As configurações serão salvas e só serão alteradas caso o ator insira outro arquivo de configuração. |

Inserir Teste – versão 1

|  |  |
| --- | --- |
| Atores | Professor |
| Pré-Condição | Ter escrito um código de configurações que contenha a descrição de pelo menos um teste |
| Fluxo de Eventos  Básico | 1. Clicar no botão “Buscar Arquivo”  2. Selecionar o arquivo de configurações no file system  3. Clicar em “Ok”  4. Receber a mensagem de confirmação |
| Fluxo de eventos secundário | 1. O ator pode cancelar a entrada de um arquivo de configurações com testes a qualquer momento.  2. Se acontecer algum problema na hora de inserir o arquivo de configuração no programa, ele será exibido no passo quatro. |
| Pós-Condição | As configurações e testes serão salvas e só serão alteradas caso o ator insira outro arquivo de configuração. |

Testar código – versão 1

|  |  |
| --- | --- |
| Atores | Professor |
| Pré-Condição | Ter inserido no programa pelo menos um teste. |
| Fluxo de Eventos  Básico | 1. Clicar no botão “Testar”.  2. Ver os resultados do teste. |
| Fluxo de eventos secundário | O ator pode executar vários testes diferentes quantas vezes for necessário. |
| Pós-Condição | As configurações serão salvas e só serão alteradas caso o ator insira outro arquivo de configuração. |

* Análise de Robustez

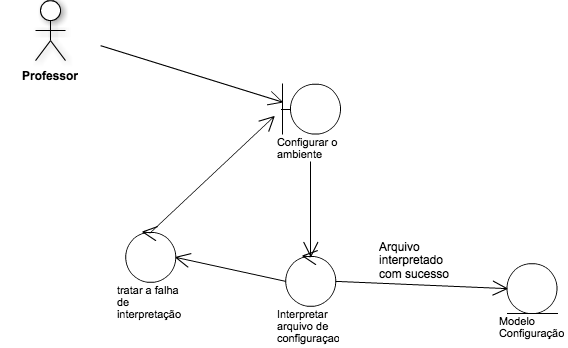


Figura 15 – Diagrama de Robustez

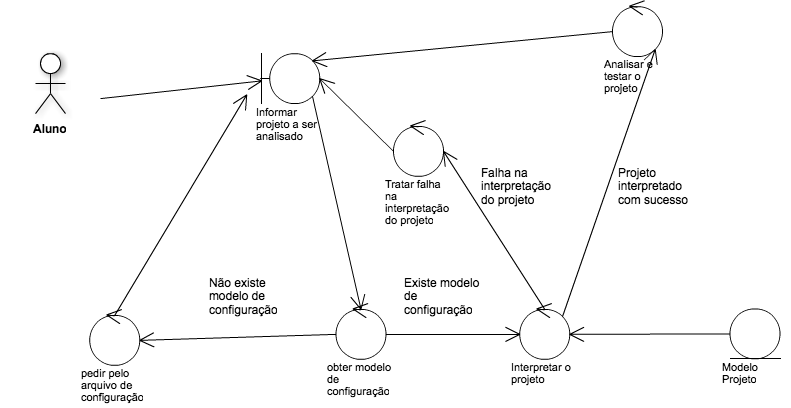


Figura 16 – Diagrama de Robustez

* Lista de requisitos

A lista de requisitos foi atualizada de acordo com as modificações feitas na análise, para adicionar a funcionalidade de testes.

|  |  |
| --- | --- |
| Requisito | Especificação |
| R1 | A aplicação deve permitir selecionar qual o código deve ser analisado. |
| R2 | O código deve estar escrito na linguagem Java. |
| R3 | A aplicação deve permitir quais as medidas devem ser realizadas. |
| R4 | A aplicação deve ser capaz de analisar o código selecionado, utilizando as medidas selecionadas e mostrar os resultados. |
| R5 | A aplicação deve oferecer pelo menos seis medidas diferentes: WWC, DIT, NOC, CBO, RFC e LCOM. |
| R6 | Todos os resultados das métricas são representados por números naturais |
| R7 | A aplicação deve realizar as métricas de acordo com as especificações apresentadas por Chidamber e Kemerer. |
| R8 | A aplicação deve ser capaz de receber um arquivo de configurações em XML, contendo informações sobre quais métricas usar. |
| R9 | A aplicação deve permitir inserir testes. |
| R10 | A aplicação deve ser capaz de realizar os testes carregados no código selecionado. |
| R11 | A aplicação deve exibir os resultados dos testes. |

Tabela 4 – Especificação dos requisitos da segunda iteração.

**6.2.2 – Desenho**

* Diagramas de Classe (DCL)

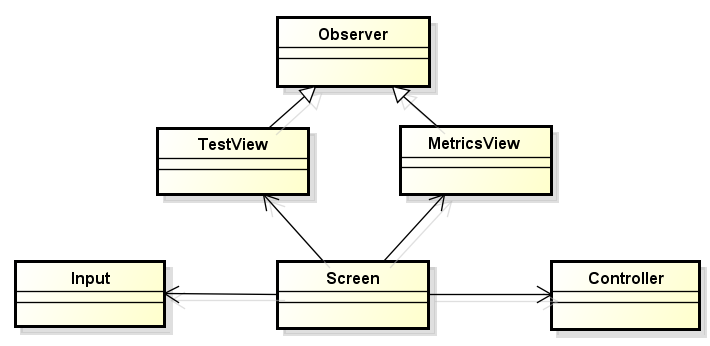


Figura 17 – Diagrama de classes

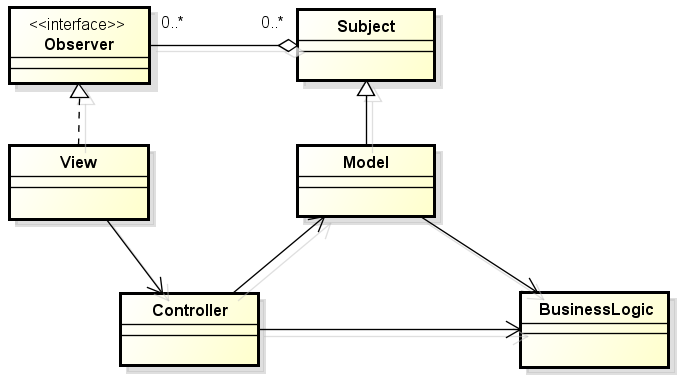


Figura 18 – Diagrama de Classes

Para evitar acoplamento entre as linhas de código, foi aplicada a arquitetura *Model-View-Controller* (MVC) na construção da tela. Deste modo, a interface do usuário, a lógica de controle e os dados ficaram separados, permitindo um melhor controle do fluxo de informações e de manutenção do código.

Para aplicar a arquitetura MVC, também foi utilizado mais um projeto GoF do tipo comportamental, o padrão Observer [11].

**6.2.3 - Implementação e Testes**

**<texto\_sobre\_implementação\_testes\_aqui>**

|  |
| --- |
| <configuration>  <class classpath="br.edu.exemplo.Exemplo">  <metric name="DIT">  <metric name ="CBO">  <metric name ="WWC">  <method name ="executar(int a, int b)">  <parameter type ="int" value ="3" />  <parameter type ="int" value ="3" />  </method>  </class>  </configuration> |

Figura **19** – Exemplo de arquivo de configuração de teste/medidas

**7. RESULTADOS E DISCUSSÃO**

Durante o primeiro semestre, foi desenvolvida a parte de métricas do ambiente proposto. Somente isso ainda não era suficiente para o propósito visado, as métricas sozinhas podem parecer um conceito vago e sem muita importância para um aluno iniciante em programação.

Foram encontradas dificuldades na implementação do componente interpretador do código a ser medido, pois a linguagem Java não oferece recursos práticos para tal. A maneira encontrada para realizar essa tarefa foram expressões regulares (REGEX), no entanto essa opção se mostrou mais trabalhosa do que o esperado. É difícil prever como o código a ser analisado vai estar formatado pelo aluno que irá submeter o seu código aos testes e medidas do nosso ambiente, uma vez que, nada o impede que ele esteja escrito em uma formatação completamente diferente da que se costuma utilizar em ambientes profissionais e/ou mais desenvolvidos conceitualmente dentro do contexto de engenharia de software. No entanto, apesar das dificuldades, a evolução na construção do ambiente não sofreu grandes problemas e seguiu como planejado.

No segundo semestre foi desenvolvida a funcionalidade de testes e a interface para o usuário. A funcionalidade de testes apresentou alguns problemas de implementação inicialmente, principalmente no uso do javax.reflection, um pacote Java que permite a chamada de métodos sem o conhecimento de classes e objetos envolvidos, mas eles foram superados dentro do prazo previsto.

Outra funcionalidade acrescentada foi a leitura de um arquivo de configuração XML com as informações sobre as métricas e os testes que devem ser realizados. Facilitando desse modo o trabalho do professor, que pode selecionar as métricas e testes uma vez só no arquivo de configuração, e disponibilizar para todos os alunos.

As métricas e os testes juntos oferecem uma base melhor para o aluno compreender a importância de um código mais bem elaborado. O aluno consegue visualizar os problemas que podem surgir se um código não for bem testado, e ao mesmo tempo aprende que só isso não é suficiente, só passar nos testes não significa que um código está bom, ainda é preciso se preocupar com a estrutura dele.

Essa ferramenta então, pode ajudar professores a ensinar a alunos iniciantes conceitos importantes sobre desenvolvimento de software dentro do tempo de aula, ao mesmo tempo que ensina uma linguagem, ou lógica de programação.

Infelizmente o tempo para o desenvolvimento desse projeto não era suficiente para testar a aplicação em um ambiente real, então não é possível afirmar com certeza a eficácia da aplicação, no entanto acredita-se que o seu efeito em um ambiente real seria positivo.

**8. CONCLUSÃO**

O desenvolvimento do ambiente está seguindo como planejado. Embora a funcionalidade referente às métricas esteja sendo implementada, a ferramenta ainda não cumpre o seu propósito, que é auxiliar o professor no ensino da complexidade de funcionamento, uma vez que as métricas sozinhas não significam muito para um aluno ingressante - já que ele não possui a carga de conhecimento necessária para entender a importância delas. É importante que os testes também existam, para que a questão do funcionamento do software possa ser trabalhada de maneira mais clara, abrangente e completa pelo professor.

**9. REFERÊNCIAS**

**Norma ABNT**:

ABNT – Associação Brasileira de Normas Técnicas. NBR 14724: Informação e documentação. Trabalhos Acadêmicos - Apresentação. Rio de Janeiro: ABNT, 2002.

**Ferramentas**:

Dropbox: *https://www.dropbox.com/*

Eclemma: *http://www.eclemma.org/*

Eclipse Juno/Kepler: *http://www.eclipse.org/*

E-Git: *http://www.eclipse.org/egit/*

Gerrit: *https://code.google.com/p/gerrit/*

GIT: *http://git-scm.com/*

GitHub: *https://www.github.com*

Google-Drive: *https://drive.google.com/‎*

Pencil Project: *http://pencil.evolus.vn/*

JUnit: *http://junit.org/*

Trello: *https://trello.com/*

**Artigos, livros e sites**:

[1] PORTELA, C. Prototipação exploratória. Disponível em:

< *http://www.cesarkallas.net/arquivos/faculdade/engenharia\_de\_software/11-Prototipa%87\_o%20Explorat%A2ria/Prototipa%87\_o%20Explorat%A2ria.pdf* >

Acesso em: 18 de junho de 2013.

[2] STANDISH GROUP. Standish Group Blog. Disponível em:

< *http://blog.standishgroup.com/* >. Acesso em: 18 de junho de 2013.

[3] CHIDAMBER, S., KEMERER, C., *Towards A Metrics Suite For Object Oriented Design*. Massachusetts Institute of Technology; Cambridge; Nov. 1991

[4] SMITH, BRIAN C., *The Limits of Correctness*; 1996. Disponível em:

< *https://www.student.cs.uwaterloo.ca/~cs492/11public\_html/p18-smith.pdf* >. Acesso em: 18 de junho de 2013.

[5] IBM. *Rational Unified Process, Best Practices for Software Development Teams*. Disponível em: <[*http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251\_bestpractices\_TP026B.pdf*](http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf)>. Acesso: 18 de junho de 2013.

[6] WELLS, D., *Extreme Programming: A gentle introduction*. Disponível em:

< *http://www.extremeprogramming.org/* >

Data de acesso: 18/06/2013.

[7] STEPHENS, M., ROSENBERG, D., COLLINS-COPE, M., *Agile Development with ICONIX Process: People, Process and Pragmatism*. APRESS, 2005.

[8] STEPHENS, M., ROSENBERG, D., *ICONIX Roadmap*. Disponível em:

< *http://iconixprocess.com/iconix-process/roadmap/* > Acesso em: 18 de junho de 2013.

[9] Chidamber and Kemerer Java Metrics. Disponível em:

< [*http://www.spinellis.gr/sw/ckjm/*](http://www.spinellis.gr/sw/ckjm/) >. Acesso em 22 de Abril de 2013.

[10] FitNesse. Disponível em: < [*http://fitnesse.org/*](http://fitnesse.org/) >. Acesso em: 22 de Abril de 2013.

[11] GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J., *Design Pattern – Elements of Reusable Object-Oriented Software*; Addison-Wesley; 1994

**APÊNDICE A**

AMBIENTE PARA TESTES E ANÁLISE DE CÓDIGOS

Carolina Cavalcanti Moya1, Diego Martins de Pinho2, HugoBretz Cabral Hennies3, Francisco Supino Marcondes4

1, 2, 3, 4 Pontifícia Universidade Católica de São Paulo

carolc.moya@hotmail.com – fmarcondes@pucsp.br

***1. Introdução***

Para um aluno ingressante, o funcionamento de um software se limita ao fato de ser seu código ter sido compilado, ou não, e ter realizado uma tarefa específica – como realizar a soma de dois números inteiros, por exemplo - com sucesso. Entretanto, esta concepção está errada e cabe ao professor ensinar isto a ele. No entanto, é possível ensinar a complexidade de funcionamento de um software, para um aluno ingressante, dentro do tempo de aula?

O ensino dos fundamentos da engenharia de software - necessários para que seja possível entender a complexidade envolvida no funcionamento de software - dentro ambiente acadêmico, é uma atividade muito limitada para os professores, uma vez que a restrição do tempo de aula e de duração do curso impede que o professor ensine todos os aspectos fundamentais necessários. Cabe ao profissional escolher cuidadosamente quais serão os principais assuntos que ele irá apresentar aos seus alunos, tornando inevitável a ausência de outros.

Entretanto, apesar de toda meticulosidade por parte do professor ao selecionar o conteúdo de seu curso, os assuntos que o mesmo decidiu não abordar, acabam se tornando lacunas quando os alunos partem do mundo acadêmico para o mercado de trabalho.

Para evitar esta situação, nosso projeto tem como objetivo a construção de um ambiente que facilite o ensino da complexidade de funcionamento dentro do tempo de aula. Com ela, o professor estará apto a realizar diversos testes nos códigos dos alunos e mostrar que, ao contrário do que os alunos ingressantes têm em mente, o funcionamento do software é algo complexo.

***2. Complexidade de funcionamento***

Um sistema computacional funciona manipulando representações formuladas através de modelos. E todo modelo lida com seu tema em um determinado nível de abstração, ignorando portanto, detalhes que não são essenciais para sua representação [1]. O problema surge quando determinado sistema, que se baseia em modelos limitados e abstratos, entra em contato com o mundo físico, e toda sua infinidade de detalhes. Devido a própria natureza do software, não é possível prever todo o comportamento que ele apresentará diante da realidade. Porem, supondo que fosse possível prever da forma mais minuciosa possível seu comportamento, isso não seria ainda prova o suficiente do seu funcionamento.

O funcionamento do software é relativo ao que se é esperado dele, à sua especificação. Esta pode ser diferente para o cliente, para o gerente de projetos e para o engenheiro do programa. De forma geral é exigido de um software de produção a funcionalidade que é requisitada pelo cliente ou usuário, se ele é adaptável a mudanças de requisitos e se ele é maleável o suficiente para adaptar-se a qualquer ambiente que seja necessário.

É importante notar também que são esperadas as falhas de um software, elas são inevitáveis, principalmente pela natureza abstrata e limitada do software, no entanto, tais falhas devem ser minimizadas para torná-lo mais confiável em face da especificação. O aluno ingressante, portanto, utilizará nosso ambiente de forma a testar e analisar o software criado por ele contra o que é esperado de forma geral do funcionamento de um software de produção.

***3. Proposta***

Para que possamos analisar a questão de funcionamento utilizaremos o conjunto de métricas OO (orientadas a objeto) apresentadas por Chidamber e Kemerer [2] e um conjunto de testes definidos pelo professor que utilizará o ambiente, característico da atividade dada em questão na sua aula. As métricas possibilitarão.

***4. Conclusões***

O desenvolvimento do ambiente está seguindo como planejado. Embora a funcionalidade referente às métricas esteja sendo implementada, a ferramenta ainda não cumpre o seu propósito, que é auxiliar o professor no ensino da complexidade de funcionamento, uma vez que as métricas sozinhas não significam muito para um aluno ingressante - já que ele não possui a carga de conhecimento necessária para entender a importância delas. É importante que os testes também existam, para que a questão do funcionamento do software possa ser trabalhada de maneira mais clara, abrangente e completa pelo professor.

***5. Referências***

[1]Smith, Brian Cantwell: The Limits of Correctness; 1996

[2] Chidamber, Shyam; Kemerer, Chris: Towards A Metrics Suite For Object Oriented Design; Massachusetts Institute of Technology; Cambridge; Nov. 1991

***Agradecimentos***

À Pontifícia Universidade Católica de São Paulo, pela disponibilização de ferramentas e laboratórios para o desenvolvimento do projeto.