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

Resumo

A proposta deste trabalho é desenvolver um ambiente que auxilie o professor 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 testes e avaliações que servirão de suporte ao professor para ensinar aos seus alunos que o fato de ter o código compilado, não significa 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

UML – *Unified Modeling Language*

PUC – Pontifícia Universidade Católica

TCC – Trabalho de conclusão de curso

ICA – Interface com o Ambiente

CDU – Casos de uso

DOM – *Domain Model* (Modelo de domínio)

IDE – Integrated development enviroment (Ambiente integrado de desenvolvimento)

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

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

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

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

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

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

RUP – *Rational Unified Process*

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....................................................................................................6
2. Programação Exploratória e Orientada ao Aprendizado x Produção.........7
3. Fundamentação Teórica............................................................................10
4. Metodologia...............................................................................................13
5. Cronograma do Primeiro Semestre...........................................................16
6. Especificação............................................................................................16
   1. Análise...............................................................................................17
   2. Desenho............................................................................................19
   3. Implementação e Testes.....................................................................21
7. Resultados e Discussão............................................................................22
8. Conclusão..................................................................................................22
9. Cronograma...............................................................................................23
10. Referências ...............................................................................................23

1. Introdução

Um software é um instrumento que automatiza e/ou apoia a realização de atividades humanas através do processamento de informações. 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, 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, assim como é preciso que ele seja flexível, para se adaptar a mutabilidade de requisitos e escalável, para poder evoluir.

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 a ausência de outros obrigatória.

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. 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 implementação 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, a programação exploratória e orientada 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 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 à 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 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 contra 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 contra elas, 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]

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 diagramas de sequência [8].

Ele 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 performance do sistema e de aceitação do usuário – O sistema é testado com base nos casos de uso, para a avaliação de performance e aceitação do usuário [8].

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

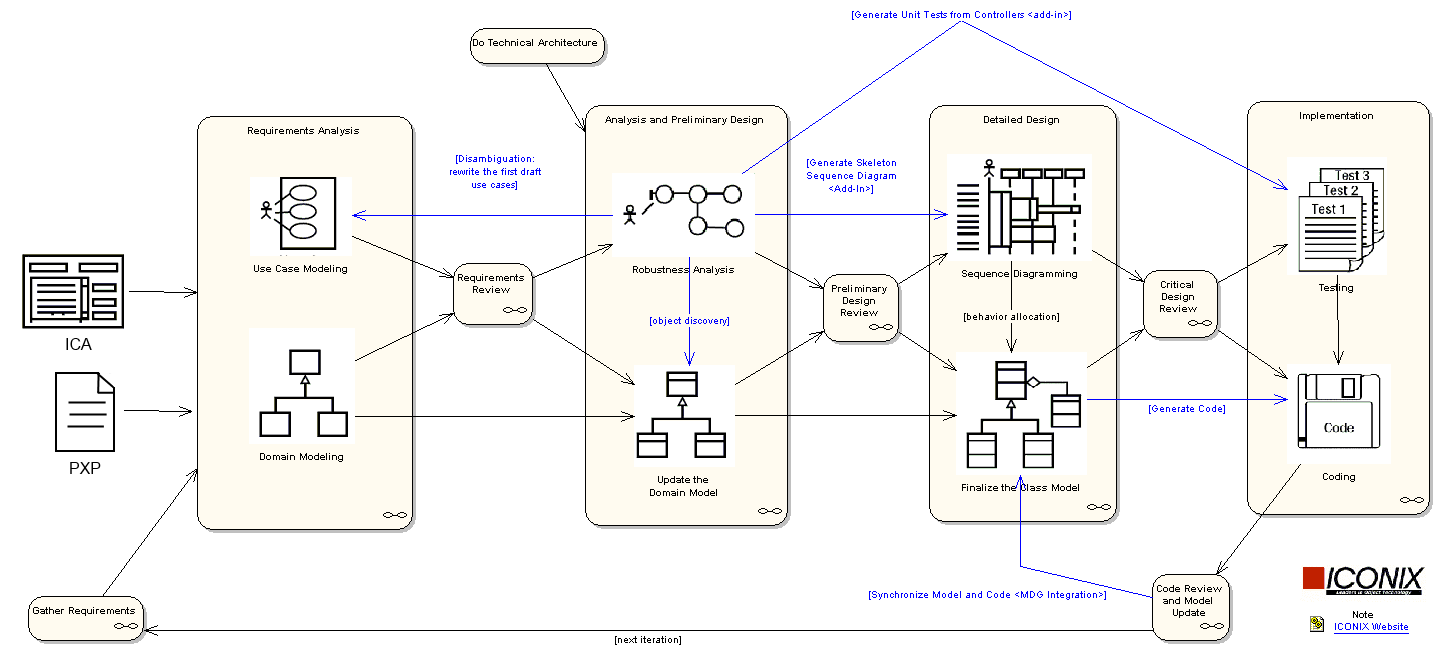


Figura 1. Processo ICONIX adaptado [8]

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.

As ferramentas de desenvolvimento a serem utilizadas são a IDE Eclipse para programação em Java, GIT, para controle de versão e JUnit para testes unitários.

5. Cronograma do Primeiro Semestre

No primeiro semestre, realizamos as seguintes atividades:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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.

6. Especificação

Neste período foram realizadas duas iterações, a primeira com o objetivo de oferecer a funcionalidade de calcular as métricas mencionadas anteriormente [7], e a segunda com o objetivo de acrescentar a funcionalidade referente aos testes.

6.1 – Iteração I

6.1.1 – Análise

* Interface com ambiente (ICA)

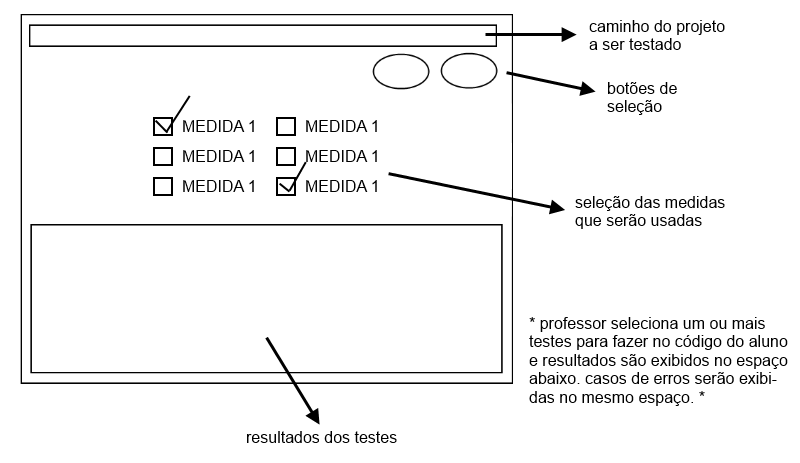


Figura 2 – Interface com ambiente da aplicação.

* Modelo de Domínio (DOM)

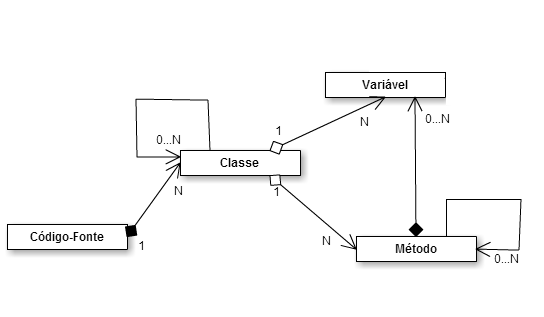


Figura 3 – Modelo de domínio da aplicação.

* Casos de Uso (CDU)

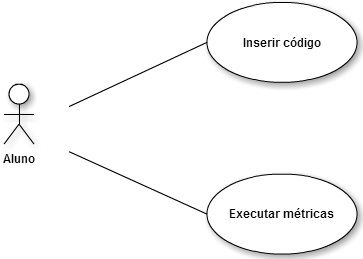


Figura 4 – Casos de uso.

* Análise de Robustez

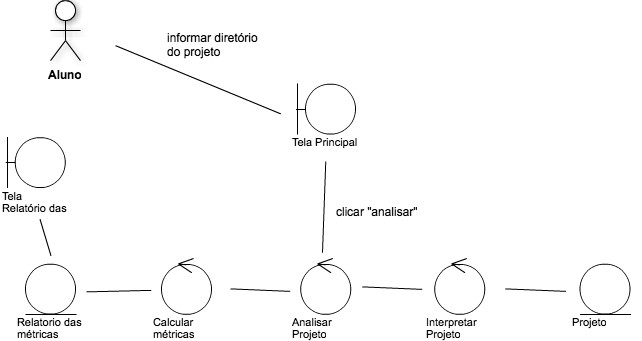


Figura 5 – Análise de robustez.

A partir do ICA, do PxP, do modelo de domínio e do caso de uso, foi obtida a seguinte lista de requisitos.

Lista de requisitos:

R1 - A aplicação deve permitir ao aluno selecionar qual o código ele quer analisar.

R2 - A aplicação deve permitir ao professor selecionar quais medidas ele deseja realizar no código do aluno.

R3 - A aplicação deve analisar o código selecionado, usando as medidas selecionadas pelo professor e mostrar o resultado no espaço reservado.

R4 - A aplicação deve oferecer pelo menos 6 medidas diferentes: WMC, DIT, NOC, CBO, RFC e LCOM.

R5 - Todos os resultados das métricas são representadas por números inteiros positivos. (>=0)

R6 - A aplicação deve realizar as métricas de acordo com as especificações apresentadas por Chidamber e Kemerer.

R7 - A aplicação deve receber um arquivo de configurações, contendo informações sobre quais métricas usar.

R8 - A aplicação deve ser capaz de ler as configurações fornecidas pelo professor.

6.1.2 – Desenho

* Diagramas de Classes (DCL)

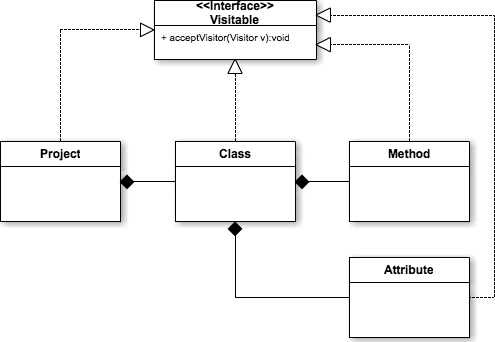


Figura 6 – Diagrama de classes UML.

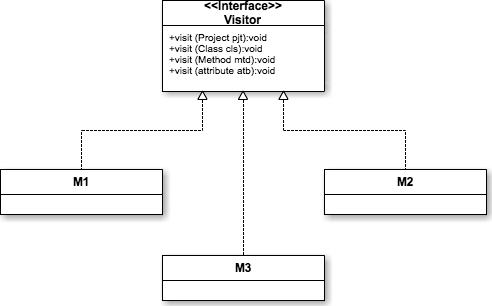


Figura 7 – Diagrama de Classes UML.

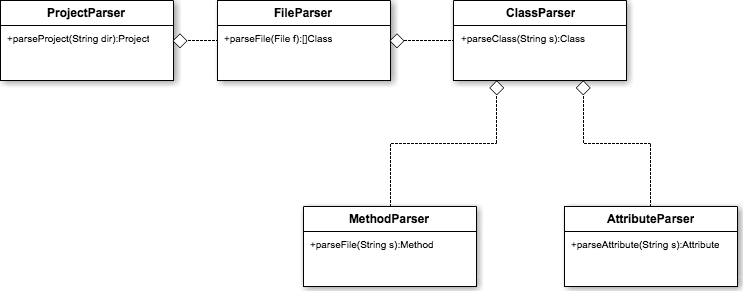


Figura 8 – Diagrama de classes

* Diagrama de sequência de mensagens (DSM)

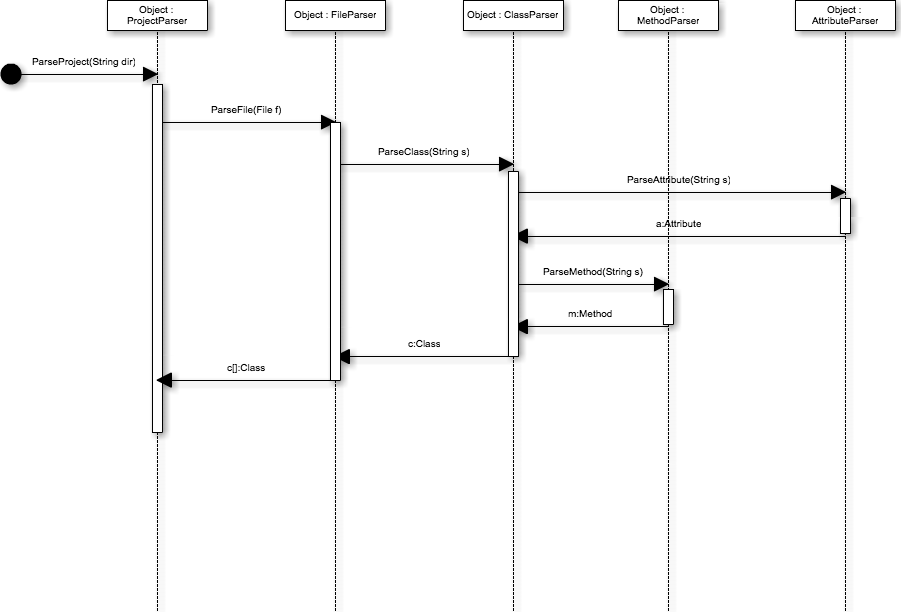


Figura 9 – Diagrama de sequência de mensagens

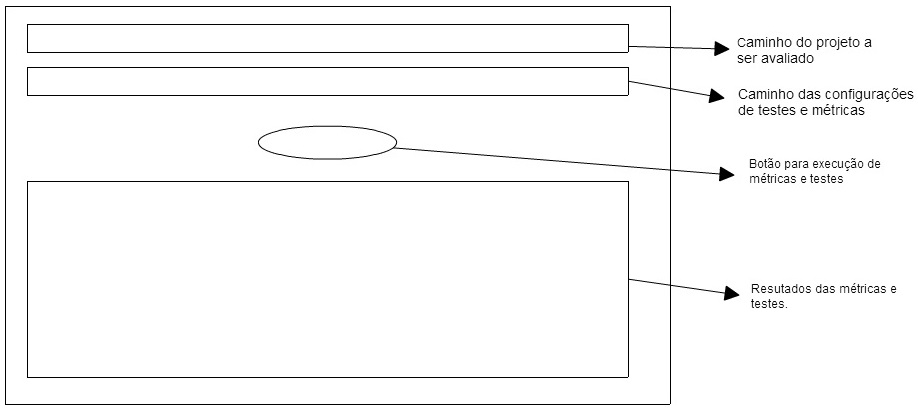
6.1.3. Implementação e Testes

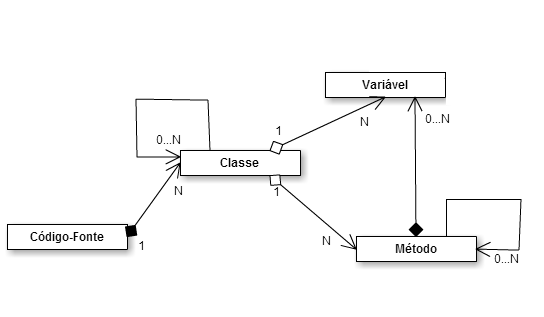
A codificação e os testes unitários foram realizados separadamente e anexados a este documento. O código PxP produzido para auxiliar na busca de requisitos também está anexado a este documento, juntamente com a biblioteca cjkm utilizada como apoio [9]. O ambiente FitNesse começou a ser estudado para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

6.2 – Iteração II

6.2.1 – Análise

* Interface com ambiente (ICA)

Figura 10 – Interface com ambiente da aplicação.

* Modelo de Domínio (DOM)
* Figura 3 – Modelo de domínio da aplicação.
* Casos de Uso (CDU)

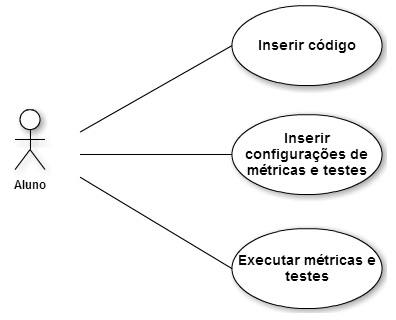


Figura 12 – Caso de uso.

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

Lista de requisitos:

R1 - A aplicação deve permitir ao aluno selecionar qual o código ele quer analisar.

R2 - A aplicação deve permitir ao professor selecionar quais medidas ele deseja realizar no código do aluno.

R3 - A aplicação deve analisar o código selecionado, usando as medidas selecionadas pelo professor e mostrar o resultado no espaço reservado.

R4 - A aplicação deve oferecer pelo menos 6 medidas diferentes: WMC, DIT, NOC, CBO, RFC e LCOM.

R5 - Todos os resultados das métricas são representadas por números inteiros positivos. (>=0)

R6 - A aplicação deve realizar as métricas de acordo com as especificações apresentadas por Chidamber e Kemerer.

R7 - A aplicação deve receber um arquivo de configurações, contendo informações sobre quais métricas usar.

R8 - A aplicação deve ser capaz de ler as configurações fornecidas pelo professor.

R9 - A aplicação deve permitir ao professor inserir os testes a serem realizados.

R10 - A aplicação deve ser capaz de realizar os testes carregados pelo professor, no código do aluno.

R11 - A aplicação deve exibir o resultado dos testes realizados para o aluno.

7. Resultados e Discussão

Durante esse semestre, foi desenvolvida a parte de métricas do ambiente proposto. Somente isso ainda não é 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 é continua e acreditamos que até o final do cronograma, o ambiente estará pronto e cumprirá seu objetivo.

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. Cronograma

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

10. Referências

**Norma ABNT**:

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

**Ferramentas**:

Eclipse Juno: <http://www.eclipse.org/>

GIT: <https://www.github.com>

**Artigos, livros e sites**:

[1] Professor Cristiano R.R. Portela. 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>

Data de acesso: 18/06/2013

[2] Standish Group. Standish Group Blog. Disponível em: <http://blog.standishgroup.com/>. Data de acesso: 18/06/2013

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

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

[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>

Data de acesso: 18/06/2013

[6] Don Wells. Extreme Programming: A gentle introduction. Disponível em: <http://www.extremeprogramming.org/>

Data de acesso: 18/06/2013

[7] Rosemberg, D.; Stephens, M.; Collins-Cope, M.: Agile Development with ICONIX Process: People, Process and Pragmatism. APRESS, 2005.

[8] Matt Stephens and Doug Rosenberg. Roadmap. Disponível em: <http://iconixprocess.com/iconix-process/roadmap/>

Data de acesso: 18/06/2013

[9] Chidamber and Kemerer Java Metrics. Disponível em: <http://www.spinellis.gr/sw/ckjm/>

Data de acesso: 22/05/2013

[10] FitNesse. Disponível em: <http://fitnesse.org/> .Data de acesso: 22/05/2013