UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GUILHERME PEREIRA SCHUELER
RICARDO COSTA ABDALLA

PROPOSTA PARA O DESENVOLVIMENTO DE APLICAÇÕES WEB UTILIZANDO O FRAMEWORK ASP.NET MVC, DESENVOLVIMENTO ORIENTADO A TESTES E DESIGN ORIENTADO AO DOMÍNIO

NITERÓI
2010

GUILHERME PEREIRA SCHUELER
RICARDO COSTA ABDALLA

PROPOSTA PARA O DESENVOLVIMENTO DE APLICAÇÕES WEB UTILIZANDO O FRAMEWORK ASP.NET MVC, DESENVOLVIMENTO ORIENTADO A TESTES E DESIGN ORIENTADO AO DOMÍNIO

Trabalho de Conclusão apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

ORIENTADOR: PROF. ESTEBAN WALTER GONZALEZ CLUA

NITERÓI
2010

GUILHERME PEREIRA SCHUELER
RICARDO COSTA ABDALLA

PROPOSTA PARA O DESENVOLVIMENTO DE APLICAÇÕES WEB UTILIZANDO O FRAMEWORK ASP.NET MVC, DESENVOLVIMENTO ORIENTADO A TESTES E DESIGN ORIENTADO AO DOMÍNIO

Trabalho de Conclusão apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

_____
Profº. Esteban Walter Gonzalez Clua

_____
Profº. Leonardo Gresta Paulino Murta

_____
Profª. Vanessa Braganholo Murta

Niterói
2010

RESUMO

Este trabalho consiste em desenvolver uma aplicação Web utilizando o *framework* ASP.NET MVC, em conjunto com o *framework* de testes NUnit, agregando na construção do aplicativo conceitos oriundos da metodologia de desenvolvimento orientada a testes (*Test-Driven Development*) e do *design* orientado ao domínio da aplicação (*Domain-Driven Design*). A aplicação proposta para implementação consiste em um sistema baseado em internet capaz de gerir a arrecadação de recursos destinados a campanhas de doações ou micro financiamentos.

Palavras-chave: Aplicações Web; ASP.NET MVC; Desenvolvimento Orientado a Testes; Design Orientado ao Domínio.

ABSTRACT

This works consists in the development of a Web Application using the ASP.NET MVC Framework, combined with the NUnit test framework, also aggregating in the construction concepts extracted from the Test-Driven Development and the Domain-Driven Design methodologies. The proposed application can be defined as an online system which manages resources for donation or micro-credit campaigns.

Keyword: Web Application; ASP.NET MVC; Test-Driven Development; Domain-Driven Design.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

MVC	Model-View Controller
TDD	Test-Driven Development
DDD	Domain-Driven Design
URL	Uniform Resource Locator
HTTP	HyperText Transfer Protocol
API	Application Programming Interface

SUMÁRIO

1. INTRODUÇÃO

O presente trabalho trata sobre um tema caro ao desenvolvimento de aplicações baseadas na Web: a ainda escassa adoção dos avanços obtidos nas técnicas da engenharia de software, como por exemplo, a modelagem orientada ao domínio do problema e o desenvolvimento orientado a testes. Tais avanços, embora já aplicados com sucesso em outras áreas do desenvolvimento de software, ainda possuem escassa adoção no ambiente de desenvolvimento para aplicativos Web.

1.1 VISÃO GERAL

A proposta aqui discutida busca complementar a utilização da arquitetura *Model-View-Controller* (MVC), já bastante reconhecida como uma base eficaz para o desenvolvimento de aplicações Web, através da incorporação de alguns dos conceitos da modelagem orientada ao domínio da aplicação e das práticas de desenvolvimento orientado a testes.

1.2 OBJETIVO

Projetar uma aplicação para ambiente Web baseada na arquitetura MVC, mantendo o foco da atividade de desenvolvimento na satisfação dos requisitos impostos pelas regras do negócio, ou seja, o domínio da aplicação, que é a camada do software realmente capaz de gerar valor aos consumidores do produto final.

Como forma de garantir que, a cada momento do ciclo de vida do produto, as especificações das regras de negócio implementadas correspondam de fato àquelas definidas pelo domínio da aplicação, os comportamentos programados serão avaliados através de testes unitários, objetivando garantir que um determinado conjunto de comportamentos funcione corretamente.

1.3 CONTRIBUIÇÕES

A principal contribuição deste trabalho reside em exemplificar como o desenvolvimento de uma aplicação projetada para ser executada sobre o protocolo HTTP (ambiente Web) pode ser organizado através da adoção da arquitetura MVC, em conjunto com técnicas do desenvolvimento orientado a testes e da modelagem orientada ao domínio da aplicação.

No exemplo aqui oferecido, a implementação da arquitetura MVC é auxiliada pela utilização de um dos inúmeros frameworks MVC *open-source* disponíveis atualmente, o ASP.NET MVC 2.0.

1.4 ESTRUTURA DA MONOGRAFIA

Esta monografia está estruturada da seguinte forma:

No capítulo 2, estão apresentados alguns dos conceitos básicos para o entendimento dos capítulos posteriores. Descreve-se o quadro atual encontrado pelos profissionais que atuam no desenvolvimento de aplicações para a Web, e uma breve introdução aos conceitos-chaves deste trabalho: Arquitetura MVC, *Domain-Driven Design* e *Test-Driven Development*.

No capítulo 3, são apresentadas as duas principais ferramentas utilizadas nas tarefas de implementação propostas por este trabalho, os frameworks ASP.NET MVC e NUnit.

No capítulo 4, discute-se a arquitetura proposta através da apresentação de um estudo de caso, onde estão sendo consideradas questões como o processo de modelagem do domínio e a infra-estrutura de software que dará suporte à implementação do modelo elaborado.

O capítulo 5 aborda a implementação da aplicação sugerida no estudo de caso do capítulo anterior. São definidos os componentes que compõem tal aplicação, e apresentados dois exemplos de Histórias de Usuário.

No capítulo 6, estão expostas as conclusões obtidas, assim como uma breve discussão acerca da viabilidade da proposta oferecida e seus possíveis desdobramentos futuros.

2. CONCEITOS BÁSICOS

Neste capítulo, são apresentados os conceitos fundamentais para o entendimento deste trabalho.

2.1. O QUADRO ATUAL DO DESENVOLVIMENTO DE APLICAÇÕES WEB

A partir do seu processo de popularização, iniciado em 1993 com o lançamento do navegador Mosaic [1], a *World Wide Web* (WWW) foi gradativamente sendo convertida de um modelo estático, baseado na exibição de documentos de conteúdo estático interligados, para um modelo dinâmico, alimentado por dados oriundos de fontes diversas, como bases de dados corporativas ou os próprios usuários da rede. A informação a ser disponibilizada passou a ser processada por software, gerando sites capazes de atender uma crescente demanda por serviços on-line, mais acessíveis e baratos, transformando a Web em uma plataforma para execução de aplicações deste tipo.

Os avanços obtidos foram conseqüência do surgimento e da evolução de tecnologias [2] capazes de estender o modelo sem-estado (*stateless*) tradicional do *HyperText Transfer Protocol* (HTTP), adicionando capacidade de processamento e armazenamento de dados, ou seja, uma nova camada de software, no intervalo existente entre a requisição feita por um computador cliente e a resposta fornecida por um computador servidor.

A primeira destas tecnologias, definida pelo padrão *Common Gateway Interface* (CGI), possibilitava aos desenvolvedores desviar a tarefa da geração da resposta a uma requisição HTTP para uma aplicação baseada em texto, conhecida como Script CGI, que seria então interpretada e executada fora da aplicação servidora, gerando um código de saída para que esta mesma aplicação servidora respondesse a requisição do cliente.

Novas tecnologias, como o *PHP:HyperText Processor* (PHP) e o *Active Server Pages* (ASP), aprimoraram o fluxo de processamento das requisições capazes de gerar conteúdo dinâmico, trazendo o processamento de scripts para dentro das aplicações servidoras, sendo então interpretadas e executadas por módulos de extensão (*runtimes*) específicos.

O desenvolvimento posterior destas tecnologias, proporcionado principalmente

pelo surgimento das plataformas Java e ASP.NET, aproximou o ambiente de desenvolvimento de aplicações Web do ambiente de desenvolvimento tradicional, substituindo a interpretação de scripts por um modelo compilado, com a execução do código gerenciado por uma máquina virtual, além de substituir as linguagens procedurais até então utilizadas por linguagens mais modernas, baseadas no paradigma de Orientação a Objetos (OO).

Atualmente, as principais plataformas de desenvolvimento para aplicações Web disponibilizam uma série de ferramentas e recursos aos programadores, sendo os mais importantes [3]:

- Ampla variedade de interfaces para acesso a dados;
- Ambientes de execução gerenciados, mais seguros e com limpeza automática de memória;
- Troca de dados entre aplicações on-line heterogêneas, através da publicação e do consumo de serviços Web (*Web Services*);
- Suporte a vários fluxos de execução simultâneos (*Multithreading*);
- Requisições assíncronas ao servidor (Ajax);

2.2. ARQUITETURA MVC

O padrão de arquitetura Modelo-Visão-Controlador (*Model-View-Controller*) surgiu em meados da década de 70 [4], no laboratório Xerox PARC, durante o projeto de desenvolvimento da linguagem orientada a objetos *SmallTalk*. Na época, o laboratório também desenvolvia um pioneiro projeto de Interface Gráfica de Usuário (*Graphical User Interface* – GUI), e havia a necessidade de melhor organizar a arquitetura das aplicações GUI, através da separação das camadas de software, de acordo com as suas respectivas responsabilidades.

Basicamente, as funcionalidades que compõem o núcleo do modelo de negócios são separadas de sua apresentação e da lógica de controle que as utiliza. Esta separação possibilita com que múltiplas visões do modelo de negócio sejam geradas, além de facilitar a legibilidade, a compreensão do código gerado e a sua testabilidade.

As camadas da arquitetura MVC podem ser sucintamente descritas da seguinte forma [5]:

- Modelo: a camada de Modelo é a responsável pela lógica do domínio da aplicação, o que inclui as regras do negócio, suas entidades e operações. É

independente de quaisquer vínculos com a camada de visão, além de centralizar as funcionalidades referentes à persistência dos dados.

- Controle: a camada de Controle é a responsável pela lógica da aplicação, traduzindo as interações com as visões em ações a serem executadas pelo Modelo, alterando assim o seu estado e, posteriormente, sua visualização. Dentro do modelo de aplicações Web, as atividades da camada controladora começam com o processamento das requisições recebidas do cliente, seguida por operações que são executadas sob o Modelo, com os dados resultantes sendo então preparados para serem passados à visão adequada.

- Visão: a camada de Visão possui a lógica de apresentação da arquitetura, convertendo os dados do Modelo em uma Interface de Usuário. É dela a responsabilidade em manter a consistência dos dados apresentados, quando o Modelo sofre alterações. No ambiente de aplicações Web, isto se traduz em uma requisição, gerada pela Visão, solicitando ao Modelo seu estado atual.

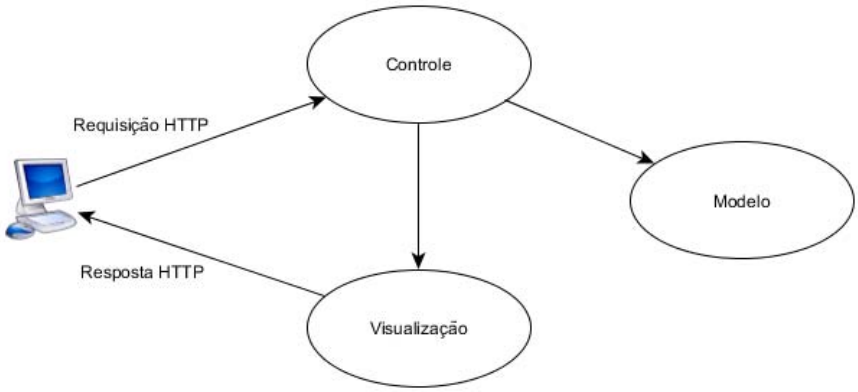


Figura 1 - Arquitetura MVC para Web [4]

2.3. DESIGN ORIENTADO AO DOMÍNIO

2.3.1. Introdução

Popularizada pelo autor Eric Evans [6], o Design Orientado ao Domínio (DDD) pode ser sucintamente descrito como uma forma disciplinada de criar um Modelo de Domínio. O seu objetivo é controlar a complexidade inerente ao processo de

desenvolvimento de software e atingir o que o autor chama de "coração do software", ou seja, a capacidade que um software tem de resolver problemas relacionados ao seu domínio para os usuários.

O DDD não pretende, em sua proposta, ser uma nova metodologia para o desenvolvimento de software, mas sim um conjunto de práticas estruturais e terminologias para a realização de decisões de design, acelerando projetos de software que possuam domínios complexos.

Para isto, as práticas propostas pelo DDD focam o núcleo do domínio, ou seja, a atividade fim ou a área a que o software se destina, construindo designs complexos através de modelos, tidos como abstrações organizadas e seletivas do conhecimento que se tem de um domínio, constantemente aperfeiçoados e refinados durante o processo.

2.3.2. O Modelo do Domínio

Para que um software aplicado a uma determinada atividade possua valor, é necessário que os seus desenvolvedores sejam capazes de considerar um conjunto de conhecimentos específicos relacionados a esta atividade. Um modelo é uma forma de conhecimento estruturada e seletivamente simplificada, capaz de dar forma ao design de um software que busque oferecer valor aos seus usuários [7].

No contexto do DDD, a elaboração de um modelo acontece de forma iterativa. Os modeladores do domínio devem ser capazes de digerir conhecimento do domínio, aplicando ao modelo somente aquilo que é de fato relevante para a implementação. O DDD propõe uma colaboração estreita entre especialistas do domínio e desenvolvedores, o que leva a necessidade da elaboração de uma linguagem comum. Esta linguagem tem a sua gênese no próprio modelo de domínio, e deve ser capaz de comunicar o fluxo de conhecimento existente entre o modelo e a sua implementação.

O modelo e a implementação estão intimamente relacionados, e é este relacionamento que torna o modelo relevante e garante que o conhecimento obtido na modelagem seja aplicado ao produto final. De forma recíproca, o conhecimento gerado a partir da implementação contribui para o refinamento do modelo. Este mecanismo exprime o fato de que, ao desenvolver um software complexo, nunca se sabe o suficiente em seus estágios iniciais. O conhecimento que a equipe de desenvolvimento possui do domínio é gradativamente aumentado ao longo do processo, e deve ser estimulado através de técnicas de aprendizado contínuo.

2.3.3. A Linguagem Onipresente

A necessidade de se desenvolver um modelo do domínio através da cooperação entre especialistas de domínio e desenvolvedores de software enfrenta uma barreira natural: a comunicação. Cada um destes grupos de profissionais possui o seu próprio vocabulário, composto por terminologias e conceitos distintos, dificultando o intercâmbio de idéias e comprometendo a construção conjunta de conhecimento entre os envolvidos no projeto, o que é fundamental para o bom desenvolvimento do modelo.

Torna-se, portanto, crítica a construção de uma linguagem comum, baseada no modelo, que é o ponto de encontro entre o software e o domínio. O DDD a denomina Linguagem Onipresente (*Ubiquitous Language*), e propõe que os desenvolvedores a utilizem para a descrição dos artefatos, tarefas e funcionalidades do sistema. Os especialistas do domínio devem utilizá-la para comunicarem-se uns com os outros, mas também para especificar requisitos e o planejamento do projeto junto aos desenvolvedores.

A Linguagem Onipresente deve permear o código e a sua documentação, eliminando, na medida do possível, a necessidade de tradução entre conceitos técnicos e conceitos pertencentes ao domínio. Quanto maior for a sua permeabilidade no projeto, mais suave será o fluxo de conhecimento.

2.3.4. Design Orientado ao Modelo (DOM)

Em muitos projetos de software, é prática comum iniciar a elaboração de um modelo do domínio através da geração de um modelo de análise, elaborado por especialistas do negócio. Este tipo de modelo costuma ser utilizado para melhor compreender o domínio e, durante a sua elaboração, não são levados em consideração questões relacionadas ao software e a sua futura implementação. Uma vez concluído, este modelo será entregue aos desenvolvedores responsáveis pela elaboração do modelo de projeto. Mas, visto que tal modelo não considerou questões relacionadas ao design, existirá, provavelmente, a necessidade de adaptação ou reformulação do modelo por parte dos desenvolvedores, e o resultado final será o abandono do modelo de análise tão logo a codificação comece.

Outro sério problema encontrado nesta abordagem é a incapacidade dos analistas de negócio de preverem todos os defeitos existentes em sua modelagem, além de ignorarem detalhes importantes do domínio que serão notados somente durante o

processo de implementação.

A abordagem proposta pelo DDD consiste em relacionar intimamente a modelagem do domínio e o design do software. O modelo deve ser elaborado levando em consideração questões relacionadas a implementação do software, e os desenvolvedores devem fazer parte do processo de modelagem. A idéia central é optar por um modelo que possa ser apropriadamente descrito em software, de modo que o processo de desenvolvimento possa evoluir de acordo com o modelo, tornando o modelo relevante e dando significado ao código. O código deve ser uma expressão do modelo, de modo que mudanças no código sejam também mudanças no modelo, com suas conseqüências propagadas naturalmente para as demais atividades do projeto.

2.3.5. Os elementos de construção do DOM

Abaixo estão apresentados os principais elementos utilizados na modelagem de software, segundo a visão proposta pelo DDD.

2.3.5.1. Arquitetura em Camadas

Ao desenvolver um software, uma grande parte do código gerado não está diretamente relacionada ao seu domínio, sendo parte da infra-estrutura e da lógica de funcionamento da aplicação, como por exemplo, o acesso a dados, a redes e a interface de usuário. Se o código relacionado ao domínio encontra-se diluído entre as demais partes do código da aplicação, torna-se difícil pensar e atuar sobre ele. Neste contexto, mudanças superficiais na lógica da aplicação podem influenciar negativamente a lógica do negócio, e a manutenção desta mesma lógica torna-se difícil e dispendiosa, além de inviabilizar a automação dos testes.

Assim sendo, todo o código relacionado ao modelo do domínio deve permanecer em uma camada isolada [8], de modo que os seus objetos, livres de quaisquer outras obrigações, estejam focados na correta expressão do modelo.

Os demais aspectos do sistema devem ser organizados em camadas, de modo que qualquer elemento de uma camada dependa somente de outros elementos da própria camada, ou de elementos de camadas inferiores. O design em camadas deve buscar o mínimo grau possível de acoplamento com as camadas superiores.

Basicamente, um design elaborado sob os preceitos do DDD [9] possuem as camadas exibidas na figura 2.

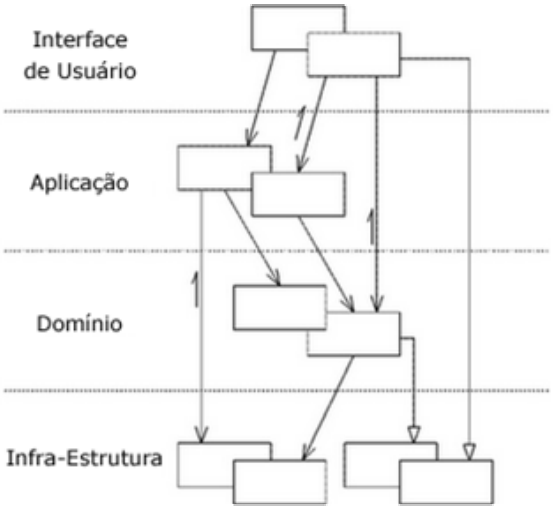


Figura 2 – Arquitetura em Camadas [6, Pág. 64]

- UI (*User Interface*): exibe informações ao usuário do sistema, e interpreta os seus comandos.
- Aplicação: define as funções que o sistema deve executar e direciona os comandos recebidos pelo sistema, coordenando as tarefas e delegando o trabalho para os objetos do domínio, responsáveis pela solução efetiva dos problemas.
- Domínio: representa os conceitos e regras do negócio, além de possuir as informações sobre o estado corrente dos objetos do domínio. É o coração do software.
- Infra-estrutura: fornece os recursos técnicos genéricos que dão suporte às camadas superiores, como por exemplo, a persistência de estado do domínio.

2.3.5.2. Entidades

Na visão da DDD, as entidades são uma categoria de objetos que possuem uma identidade própria e que permanece inalterada durante as mudanças de estado do objeto [10]. Um objeto representativo de uma entidade precisa poder ser distinguido de outro objeto que, eventualmente, possua os mesmos atributos. É, portanto, imprescindível a existência de um identificador único. Este identificador pode ser um atributo específico, ou um conjunto de atributos que juntos possuam um único valor, ou ainda um

identificador arbitrário criado pelo sistema.

2.3.5.3. Objetos Valor

Diferentemente das entidades, objetos valor são usados para descrever aspectos do domínio que não possuam uma identidade própria [11]. É recomendado que sejam imutáveis, ou seja, não sejam alterados durante o seu ciclo de vida, pois é comum a necessidade de compartilhá-los.

Objetos valor podem conter outros objetos valor, e também referências para entidades. São usados para armazenar atributos dos objetos do domínio que devam ser agrupados de modo a formar um todo conceitualmente coerente, como por exemplo, o endereço de uma entidade que represente um cliente.

2.3.5.4. Agregados

Um agregado é um padrão do domínio utilizado para definir a posse e as fronteiras de um objeto, garantindo a integridade dos objetos nele contido [12]. É, basicamente, um grupo de objetos associados, considerados como uma unidade do ponto de vista das mudanças no estado dos dados.

Todo agregado possui uma raiz, que é uma entidade, sendo esta raiz o único objeto do grupo acessível ao mundo exterior. Uma raiz pode possuir referências a qualquer um dos objetos agregados, e estes objetos podem possuir referências entre si. No entanto, um objeto exterior à agregação pode somente referenciar o objeto raiz.

Um agregado pode ser composto por objetos valor e outras entidades além da entidade raiz, mas estas possuem identidade local, fazendo sentido somente dentro da agregação. Os objetos agregados podem referenciar outros agregados, desde que esta referência seja feita à entidade raiz da agregação.

2.3.5.5. Serviços

Ao desenvolver uma modelagem, nem todos os aspectos do domínio podem ser facilmente mapeados em objetos. Objetos são caracterizados por possuírem atributos, um estado interno e um comportamento. Na construção da linguagem onipresente, os nomes da linguagem são mapeados em objetos; seus verbos, associados ao nome correspondente, transformam-se em parte do comportamento destes objetos. No entanto, algumas ações do domínio não pertencem a nenhum objeto, embora ainda assim

representem algum comportamento importante do domínio. Quando este tipo de comportamento é identificado, deve ser declarado como um serviço.

O propósito de um serviço é prover funcionalidades para o domínio [13]. A sua identificação na modelagem do domínio passa pelo reconhecimento de três características importantes:

1. As operações executadas pelos serviços se referem a conceitos do domínio que não pertencem naturalmente a entidades ou objetos valor;
2. Estas operações fazem referência a outros objetos do domínio;
3. Não possuem estado.

2.3.5.6. Fábricas

Na modelagem orientada a objetos, uma fábrica é, basicamente, um objeto utilizado para criar outros objetos. São utilizadas principalmente para encapsular o conhecimento necessário para a criação de objetos complexos, como por exemplo, agregados definidos no modelo do domínio [14].

2.3.5.7. Repositórios

O padrão de Fábrica, como dito anteriormente, é utilizado para a criação de objetos. Quando é preciso recuperar objetos já criados, utiliza-se o padrão de Repositório.

Um repositório atua como local de armazenamento para objetos do domínio globalmente acessíveis, encapsulando toda a lógica necessária para obter referência a tais objetos [15]. A utilização de repositórios também tem por objetivo evitar que objetos do domínio tenham que interagir diretamente com a infra-estrutura, mantendo, portanto, o isolamento da camada de modelo.

2.4. DESENVOLVIMENTO ORIENTADO A TESTES

O Desenvolvimento Orientado a Testes (*Test-Driven Development*, TDD) é uma técnica de desenvolvimento de software baseada em ciclos de curta duração, onde a ênfase é dada primeiramente na elaboração e escrita de casos de testes automatizados, para só então se prosseguir para o estágio posterior, onde será efetivamente escrito código capaz de atender aos requisitos definidos pelo caso de teste e, conseqüentemente, ser verificado por ele [16].

O TDD encoraja o design simples e modularizado, buscando com isto produzir componentes de softwares com baixa dependência, sendo, portanto, testáveis. Sua implementação requer a criação de testes unitários automatizados, que deverão definir os requisitos do software, antes de sua escrita. A verificação do código elaborado através da aprovação obtida em um teste confirma o comportamento adequado, na medida em que o código se desenvolve e é refatorado. A implementação de um teste automatizado, dentro do contexto do TDD, deve primeiro obter um resultado falho, visto que a funcionalidade a ser testada ainda não foi implementada. A partir deste ponto, o trabalho do desenvolvedor consiste em elaborar código capaz de obter um resultado positivo no teste, para finalmente refatorá-lo para alcançar padrões aceitáveis.

Alguns dos principais benefícios da aplicação das técnicas do TDD são [17]:

- Ao iniciar pelos testes, o TDD proporciona a redução de custo dos defeitos do software (*bugs*), visto que estes costumam ter um custo proporcional ao tempo que demoram a serem detectados;
- Testes unitários provam que o código escrito realmente funciona, mas não substituem testes de integração e aceitação;
- A existência de testes unitários automatizados permite com que o design do código seja aperfeiçoado (*refactoring*) sem comprometer, no entanto, a funcionalidade implementada;
- O conjunto de testes de um projeto é capaz de fornecer uma idéia bastante razoável do progresso de sua execução. Em um determinado momento, é possível saber o percentual de código escrito que é aprovado nos testes;
- Proporciona ganhos de produtividade, visto que parte considerável do

tempo aplicado no desenvolvimento de um software é gasto em correção de defeitos. O desenvolvimento orientado por testes reduz o número de defeitos, diminuindo, portanto, o tempo gasto em tarefas de depuração.

O ciclo de desenvolvimento orientado a testes pode ser sucintamente descrito da forma como se segue:

1. Adicionar um teste.
   Cada nova funcionalidade deve ser precedida da escrita de um teste. O teste deve, necessariamente, falhar em um primeiro momento, pois a funcionalidade em questão ainda não foi descrita. O desenvolvedor deve compreender a especificação e os requisitos da funcionalidade, através de casos de uso e histórias do usuário.
2. Executar o conjunto de testes, e verificar se o teste recém-adicionado falha.
   É preciso verificar se o novo teste não é aprovado erroneamente, ou seja, deve-se certificar de que sua implementação não acuse positivo sempre, o que anularia o valor do teste. Também é necessário verificar que o teste falha pela razão esperada.
3. Escrever código
   Esta etapa consiste em escrever código capaz de ser aprovado pelo teste, sem grande ênfase em seu design.
4. Executar novamente o conjunto de testes, e observá-los sendo aprovados.
   Caso todos os testes até então escritos sejam aprovados, cresce a confiança do programador em que os requisitos estão sendo obtidos.
5. Refatorar o código.
   Na escrita de código descrita pelo passo 3, o design não era prioritário, visto que a ênfase estava na obtenção de código capaz de ser aprovado pelo teste. Nesta etapa, o código já é capaz de passar no teste, e a ênfase desloca-se para a obtenção de melhorias na sua implementação. No término desta atividade, é preciso executar novamente o conjunto de testes, para verificar se as atividades de refatoração não alteraram o sentido do código.
6. Repetir o ciclo.

É importante notar que o Desenvolvimento Orientado a Testes não apresenta somente vantagens. Seguindo as orientações da metodologia, ao iniciar a implementação de um novo comportamento, o teste é criado antes de se iniciar a codificação. Como conseqüência, o primeiro teste sempre falha.

Após este primeiro momento, código é gerado visando atingir o comportamento desejado e a aprovação no teste, deixando de lado, por um momento, questões relacionadas ao desempenho, design e a qualidade do código gerado. Faz-se necessário, portanto, uma etapa de refatoração.

Em ambientes de desenvolvimento muito acelerados, onde mudanças acontecem com alta freqüência e as entregas são feitas em intervalos de tempo muito curtos, o código que primeiro implementa a funcionalidade, atingindo o objetivo do teste, pode acabar sendo o código final a ser entregue, ignorando a etapa de refatoração, e comprometendo a qualidade do produto final como um todo.

3. TECNOLOGIAS RELACIONADAS

Este capítulo faz uma breve introdução das duas principais ferramentas utilizadas na execução do presente trabalho: os frameworks ASP.NET MVC e NUnit.

3.1. O FRAMEWORK ASP.NET MVC

O ASP.NET MVC é um framework para o desenvolvimento de aplicações Web, baseado na plataforma Microsoft ASP.NET, que implementa o padrão *Model-View-Controller* [18]. Sua primeira versão foi lançada em abril de 2009, sob a licença *Microsoft Public License* (MS-PL). Atualmente, encontra-se na versão 2.0.

O framework foi projetado de modo a separar as tarefas de uma aplicação Web (lógica de entrada de dados, lógica do negócio e lógica de apresentação) em camadas distintas, acopladas através de contratos baseados em interfaces, viabilizando, portanto, a sua testabilidade. Além disto, seus componentes foram elaborados para serem "plugáveis" e extensíveis: é possível, por exemplo, substituir o módulo responsável pela geração das visualizações, ou o módulo que implementa a política de roteamento das URLs (*Uniform Resource Locators*).

O coração da arquitetura do framework está em suas controladoras. Ao receber uma requisição HTTP vinda de um navegador, o módulo responsável pelo roteamento de URL (*UrlRoutingModule*) realiza a análise do endereço solicitado. Em seguida, é feito o mapeamento para uma controladora específica tratar a requisição. No ASP.NET MVC, uma controladora é basicamente uma classe escrita em código C# ou VB.NET. Dentro de uma classe controladora, métodos de ação (*action methods*) são implementados para responder a requisições originárias do navegador ou chamar uma determinada visualização. Uma controladora pode passar dados obtidos do modelo à visualização, utilizando uma estrutura de dados denominada ViewData.

O processo de execução de uma requisição no ASP.NET MVC está representado na figura 3.

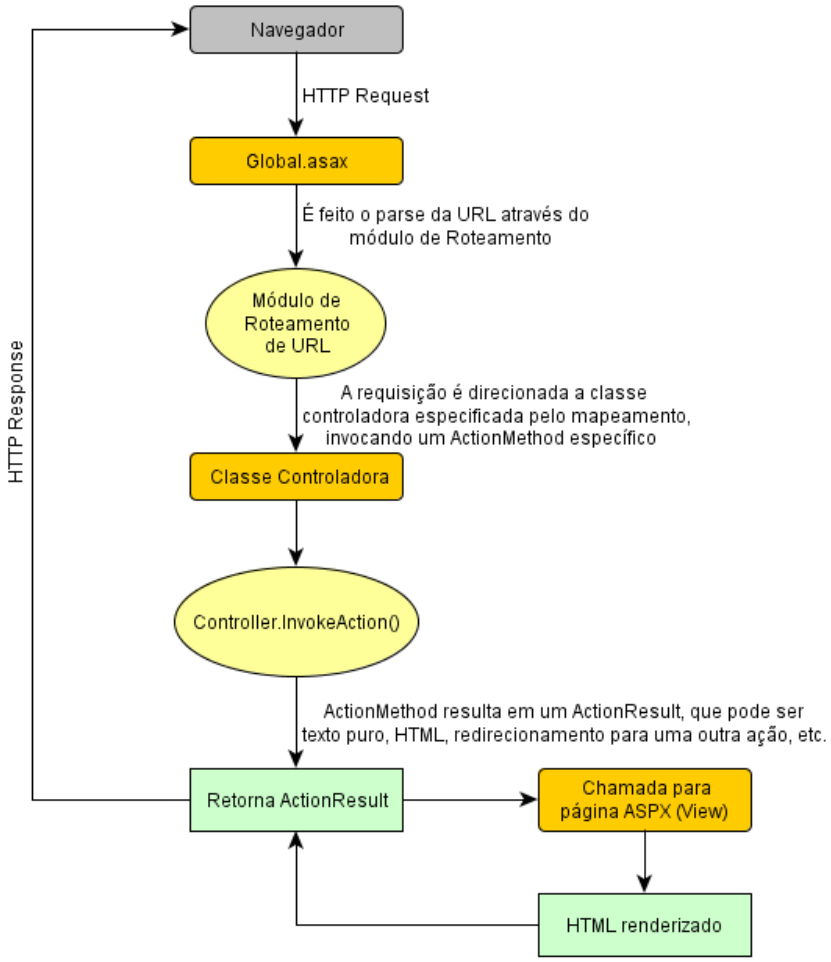


Figura 3 - O processo de execução do ASP.NET MVC [19]

O modelo de requisição padrão é o seguinte:

Erro! A referência de hiperlink não é válida., onde

- [hostname] é o endereço da aplicação
- [controller] é a classe controladora
- [action] é o método de ação a ser invocado na classe

Fisicamente, o framework separa os arquivos da aplicação Web seguindo a estrutura representada na figura 4.

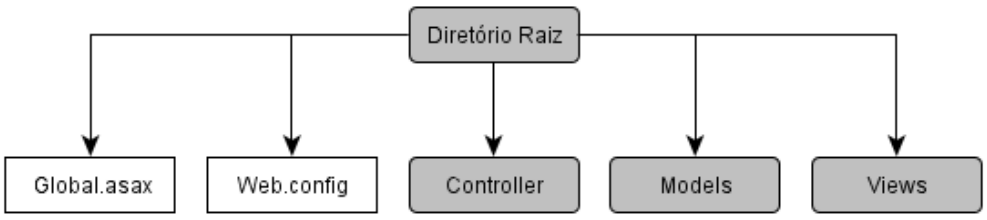


Figura 4 - Estrutura física da aplicação ASP.NET MVC

No diretório Controller, ficam armazenadas as classes controladoras, que devem seguir o padrão de nomenclatura *NomeDaControladora*Controller.

No diretório Views estão os arquivos ASPX e HTML que compõem a camada de visualização da aplicação.

O diretório Models, embora seja criado por padrão na estrutura da aplicação ASP.NET MVC, não é obrigatório. Em uma aplicação ASP.NET MVC, a camada de modelo é a única que não é controlada pelo framework. Os desenvolvedores tem, desta forma, total liberdade para implementá-la da forma que pensarem ser mais adequada. Na verdade, em grande parte dos casos esta camada estará implementada em um módulo a parte, sem vínculos com o projeto da aplicação.

3.2. O FRAMEWORK DE TESTES NUNIT

O NUnit é um framework *open-source* para testes unitários [20], projetado para operar em conjunto com as linguagens da plataforma Microsoft .NET. Foi portado a partir do seu equivalente para a plataforma Java, o JUnit, e encontra-se atualmente na versão 2.5.

Basicamente, para realizar testes unitários utilizando o framework é preciso escrever código de teste, encapsulado em um componente (DLL) a parte, que então é "marcado" com os atributos especiais *TestFixture* (para identificar uma classe que contenha métodos de teste) e *Test* (para identificar um método de teste). O código de teste desenvolvido deve conter asserções capazes de demonstrar o correto funcionamento da funcionalidade testada. As asserções são disponibilizadas ao programador através da classe *Assert*. Existem inúmeros tipos de asserções, sendo as principais as asserções de igualdade, comparação, condição, identidade e tipos.

O framework possui uma interface gráfica de usuário, exibida na figura 5, de onde é possível criar um projeto de testes, carregar componentes de testes desenvolvidos, executá-los e então acompanhar os seus resultados. Também é possível a execução dos testes através de um console de linhas de comando (nunit-console.exe).

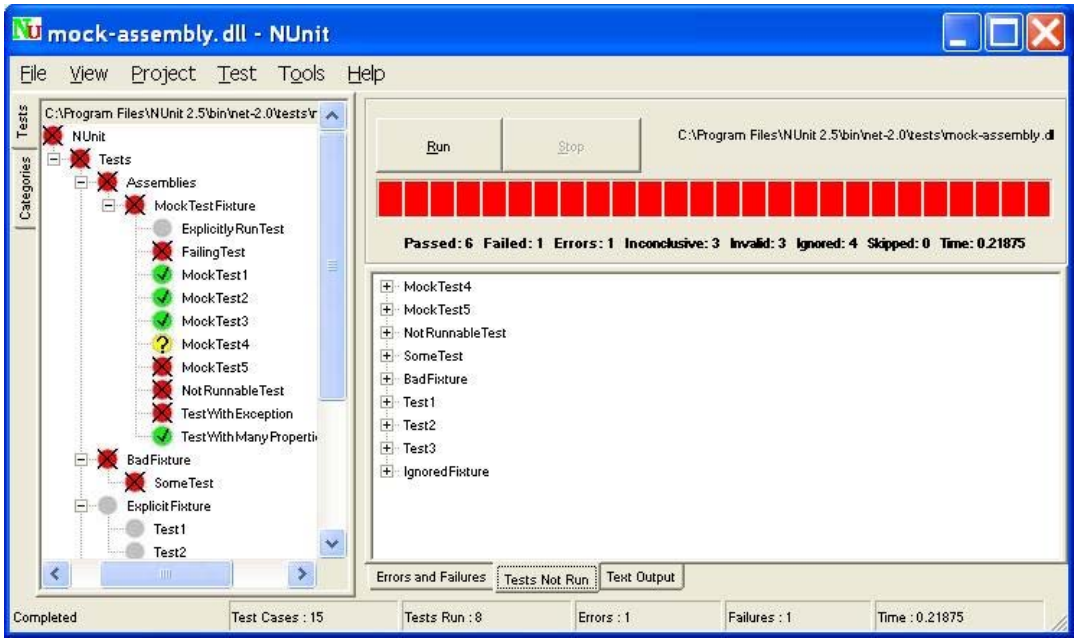


Figura 5 - Interface gráfica do NUnit (nunit.exe)

4. ARQUITETURA PROPOSTA DA APLICAÇÃO

Este capítulo pretende aplicar os conceitos de design de modelo vistos no Capítulo 2, na elaboração de uma proposta de arquitetura para uma aplicação-exemplo, sugerida no estudo de caso apresentado a seguir.

4.1. ESTUDO DE CASO

Este estudo de caso originou-se com a participação dos autores na competição acadêmica de inovação tecnológica Microsoft Imagine Cup, edição de 2008. Os autores concorreram com um projeto de aplicação de tema ecológico, desenvolvido em parceria com alguns membros da Secretaria de Agricultura do Estado do Rio de Janeiro, especialistas do domínio em questão. O objetivo do projeto era disponibilizar uma solução em software para viabilizar a arrecadação de recursos para aplicação em áreas rurais, situadas próximos às bacias hidrográficas do estado fluminense, objetivando a preservação dessas áreas em cooperação com as comunidades locais.

A proposta de aplicação aqui discutida é uma generalização do projeto piloto, e denomina-se Sistema Gestor de Microcrédito. Esta aplicação deve ser capaz de viabilizar o fluxo de recursos financeiros entre os seus usuários, que poderão ser doadores ou financiadores de idéias empreendedoras, expostas publicamente em um site através de campanhas criadas e gerenciadas por seus usuários. Tanto os usuários criadores de campanhas como aqueles que nelas investem recursos são identificados através de um perfil único (*profile*), também público. Os usuários, ao se registrarem para obter um perfil, devem apresentar o nome completo, endereço, telefone, documento de identificação e um email para contato.

Conforme dito anteriormente, as campanhas são o meio de se obter recursos através da aplicação. Uma campanha pretende arrecadar uma determinada quantia-objetivo, em um determinado prazo, através de uma justificativa determinada por seu criador. Esta justificativa pode ser tanto a necessidade de se arrecadar fundos para a manutenção de um bem público, ou mesmo financiar uma pequena empresa recém-saída da incubadora.

Sendo a campanha o meio de se obter recursos, no término do seu período de arrecadação, os recursos obtidos são direcionados para o proprietário da campanha. O período de arrecadação de uma campanha é definido por uma data inicial e uma data

final, que pode ser prolongada, de acordo com certos limites, caso o seu proprietário assim o deseje.

No ato de sua criação, uma campanha pode ser caracterizada de acordo com a forma pela qual busca ser financiada: através de doações ou de empréstimos. Para as campanhas de empréstimos, é possível ao seu criador escolher valores para os juros (sobre o valor arrecadado) a serem pagos, o número de parcelas para pagamento total da dívida, e o tempo, medido em dias após o término do período de arrecadação, para que sejam iniciados os pagamentos do financiamento.

Uma campanha possui um ciclo de vida, ou seja, passa por várias fases ou estados ao longo do tempo. Quando é criada, torna-se visível na aplicação, porém, os usuários do sistema só poderão realizar contribuições quando a data de inicio determinada em sua criação for atingida. Após a obtenção integral dos recursos pretendidos ou ao atingir a data de término definida, uma campanha de doação é finalizada. No entanto, caso seja uma campanha do tipo empréstimo, a mesma entra em uma fase de espera, até que se atinja o prazo para início da quitação do empréstimo recebido. Ao atingir a data do primeiro pagamento, que é calculado a partir dos dias informados em sua criação, a campanha tem seu estado alterado, entrando em uma fase de quitação do empréstimo recebido. Nesta fase, o responsável pela campanha deverá realizar os pagamentos do empréstimo obtido, acrescidos dos juros calculados de acordo com a taxa pré-estabelecida. Se todas as parcelas do financiamento forem quitadas corretamente, a campanha é finalizada com sucesso.

Nos casos em que ocorrem atrasos no pagamento, a campanha é suspensa até que os pagamentos sejam normalizados. Se esta condição não for atingida em um prazo de tempo máximo pré-determinado, é identificada uma situação de inadimplência por parte do criador da campanha, e a campanha é encerrada com um status que indique sua posição devedora.

A transferência dos recursos para doações ou empréstimos, assim como os pagamentos referentes a quitação das dívidas de uma campanha, serão realizados através da própria aplicação, por meio da integração com mecanismos on-line de pagamentos, como os oferecidos pelas operadoras de cartão de crédito, *PagSeguro* ou *PayPal*. Todos os usuários do sistema possuem uma conta cadastrada de onde são creditados ou debitados os pagamentos.

Cabe notar que o empréstimo de recursos financeiros esta sujeito a questões legais e, por questões de simplicidade e por não fazerem parte do foco deste trabalho,

não são aqui discutidas, mas certamente não poderiam ser ignoradas em um ambiente real de desenvolvimento.

Por fim, os administradores da aplicação possuem a capacidade de executar operações de manutenção do bom funcionamento do sistema, como por exemplo, a remoção de campanhas que contenham conteúdos inapropriados e o bloqueio de usuários com histórico de inadimplência.

4.2. O MODELO DE DOMÍNIO

A partir da proposta de aplicação descrita na seção anterior, foi elaborado um primeiro modelo do domínio (Figura 6). Este modelo reflete o conhecimento obtido a partir da análise inicial do problema, e procura identificar as principais classes e relações existentes no domínio, construindo a sua nomenclatura a partir dos elementos da linguagem utilizada na descrição da proposta.

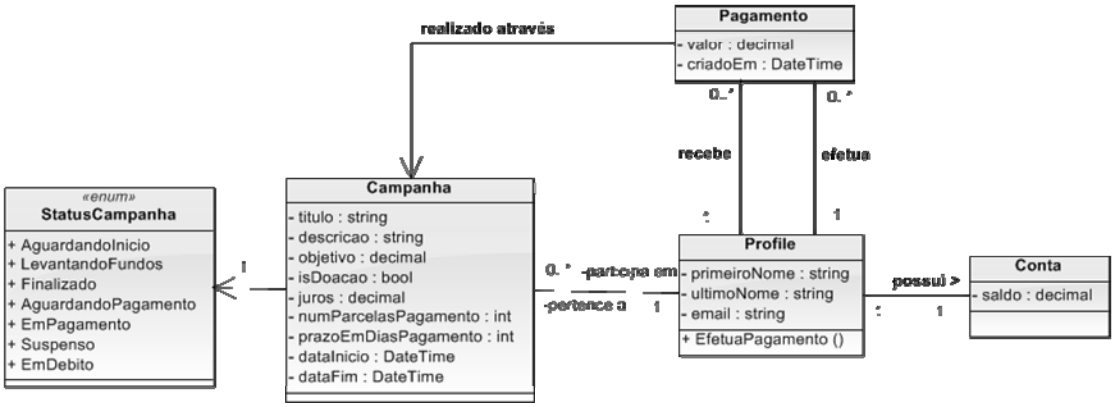


Figura 6 - Modelo nº 1

O modelo inicial já apresenta, mesmo que de forma simplificada, conhecimentos essenciais para o processo de implementação. Através de sua visualização, é possível identificar regras de negócio do domínio, como por exemplo:

> "*... uma **Campanha** pertence a um **Profile**...*"
>
> "*... um **Profile** pode doar, emprestar ou pagar um empréstimo a outros **profiles** realizando **Pagamentos** vinculados a uma **Campanha**.*"
>
> "*... os recursos arrecadados vão para uma **Conta** do Usuário.*"

A partir do modelo inicial, cada objeto do domínio presente é classificado, de acordo com o seu papel no domínio, como uma entidade ou objeto-valor.

Campanhas distinguem-se umas das outras, possuindo, portanto, uma identidade própria. O estado de uma campanha também varia de acordo com o tempo, sendo por isso um objeto mutável, classificado como uma entidade. Para identificar uma campanha, foi criado um identificador único no sistema.

Profiles são os usuários registrados do sistema, e representam uma pessoa ou empresa. Como sabemos, toda pessoa, física ou jurídica, tem um identificador próprio, como o CPF ou o CNPJ. *Profiles* são, desta forma, classificados como uma entidade e são identificados no sistema por um atributo único gerado automaticamente.

Um objeto ***Conta***, embora possua uma identidade global exclusiva (o número da conta), tem a sua existência no domínio vinculada a um *Profile*. Uma conta é mutável, e não pode ser intercambiável, ou seja, não pode ser trocada com outros *profiles*. Sendo assim, a conta é uma entidade e poderia ser identificada através do identificador do *Profile* a qual pertence.

Os objetos ***Pagamento,*** apesar de serem imutáveis, ou seja, não terem os seus valores alterados ao longo do tempo, possuem identidade própria, visto que mesmo na eventualidade de possuírem o mesmo valor e serem realizados pelo mesmo *Profile*, associado a uma mesma campanha, podem ser diferenciados pela data ou hora de sua realização. Logo, são classificados como entidade. Para facilitar a tarefa de identificação de um pagamento, um identificador numérico foi criado.

Um objeto ***StatusCampanha*** é imutável, não possui identidade e pode ser compartilhado entre os demais objetos do domínio, sendo classificado como um objeto-valor.

Uma vez concluída a classificação dos objetos identificados no modelo do domínio, o próximo passo a ser dado consiste na localização de possíveis agregados, ou seja, agrupamentos lógicos de classes tratados como uma unidade, capazes de definir fronteiras e responsabilidades, além de garantir o acesso único e a integridade dos dados.

Analisando o modelo, procura-se identificar quais objetos serão necessários no sistema, separadamente. Os objetos *Campanha* e *Profile* são partes essenciais da lógica do domínio. Constantemente serão feitas buscas diretas para acessá-los, independente de outros objetos. Sendo assim, duas agregações surgem em torno destes dois objetos do domínio, que são as suas entidades raízes.

O agregado *Profile* é composto pelas entidades *Conta* e *Pagamento*, cujas referências tornam-se acessíveis somente através da entidade raiz. No caso da entidade

Pagamento, suas referências são armazenadas na entidade *Profile* através de duas listas acessíveis ao mundo exterior, *PagamentosRecebidos* e *PagamentosEfetuados*.

O segundo agregado orbita em torno da entidade *Campanha*, e é composto pelo objeto-valor *StatusCampanha*. O agregado *Campanha*, embora seja referenciado por entidades do tipo Pagamento, não referencia diretamente instâncias desta entidade, o que violaria a fronteira do agrupamento. Visto que uma campanha pertence a um *Profile*, é apenas a este *Profile* que ela guarda referência. Desta forma, os pagamentos efetuados a uma campanha podem ser acessados sem violar as fronteiras dos agregados, simplesmente acessando a lista de pagamentos recebidos do *Profile* raiz do agrupamento referenciado pela *Campanha*, e filtrando tais objetos de acordo com a identidade da *Campanha* em questão.

O diagrama exibido na figura 7 reflete as transformações discutidas no modelo inicial.

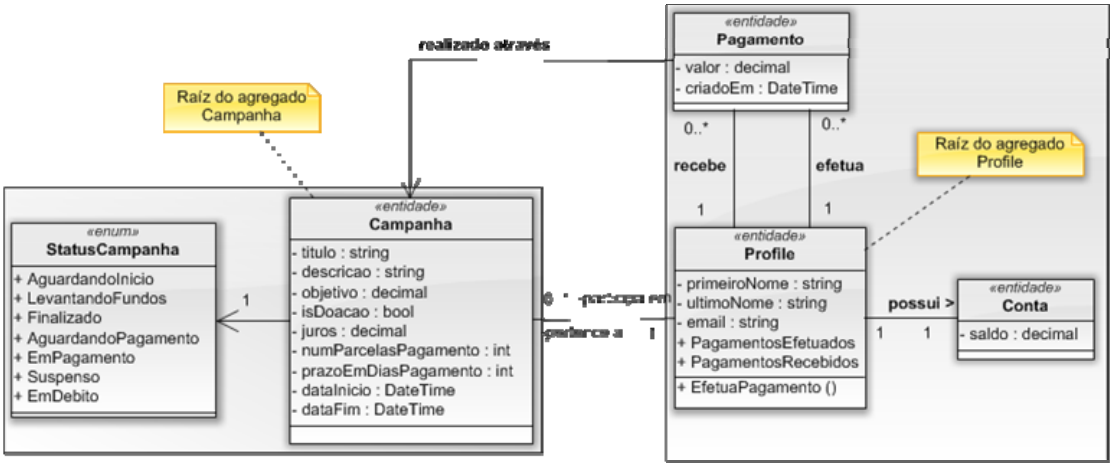


Figura 7 - Modelo nº 2

Uma vez definido as entidades, os objetos-valor, e os agregados, foram definidas as fábricas necessárias para criação dos agregados. Como dito anteriormente, o padrão de fábrica pretende encapsular a criação de objetos complexos, ocultando o conhecimento necessário para a tarefa, buscando com isto simplificar o processo.

O modelo atual possui apenas dois agregados, cujas raízes são *Campanha* e *Profile*. Sendo assim, foram criadas duas fábricas: *FactoryCampanha* e *FactoryProfile*. No *FactoryCampanha*, por exemplo, uma das complexidades inerentes a criação do

objeto é a definição do estado inicial válido da campanha, *AguardandoInício*. Este e outros conhecimentos importantes são responsáveis por garantir a integridade do objeto, e por isto estão encapsulados em sua fábrica.

A figura 8 exibe o modelo após a criação das fábricas.

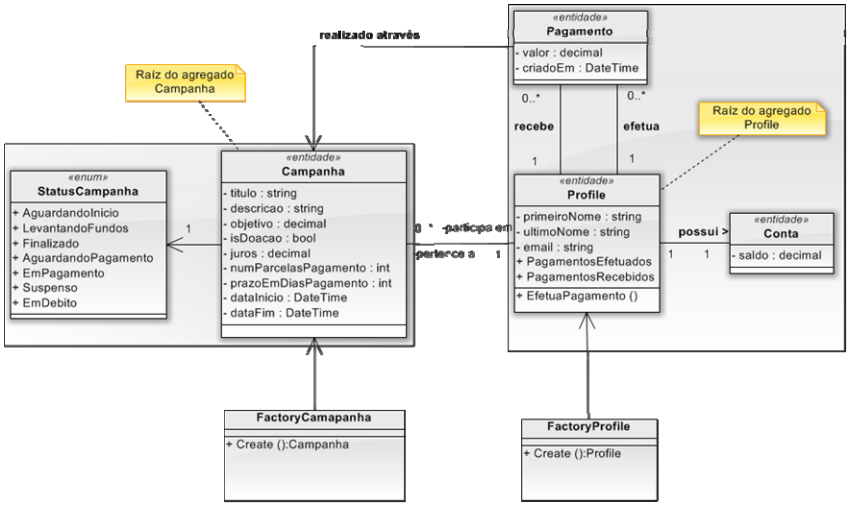


Figura 8 - Modelo nº 3

O próximo refinamento do modelo envolve o padrão repositório, que será o responsável pelo gerenciamento do ciclo de vida dos objetos. O ciclo de vida de um objeto do domínio é composto por vários estágios, mostrados na figura 9.

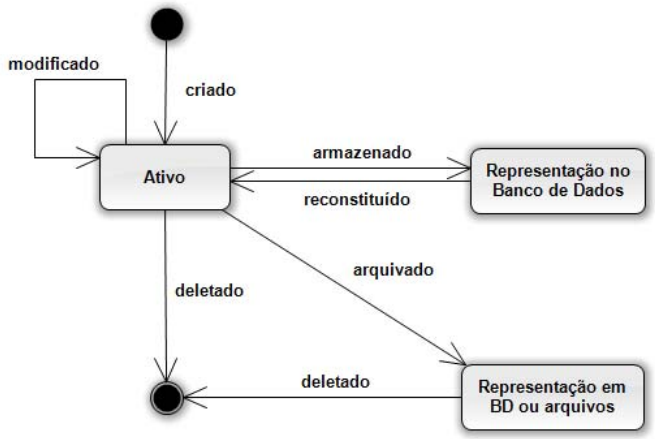


Figura 9 - Ciclo de vida de um objeto do domínio [6, Pág. 117]

Ao ser criado, um objeto do domínio passa para o estado Ativo. Neste estado, ele pode ser modificado, armazenado, arquivado ou excluído. Ao ser modificado, o objeto permanece no estado Ativo; ao ser armazenado, ele passa para o estado

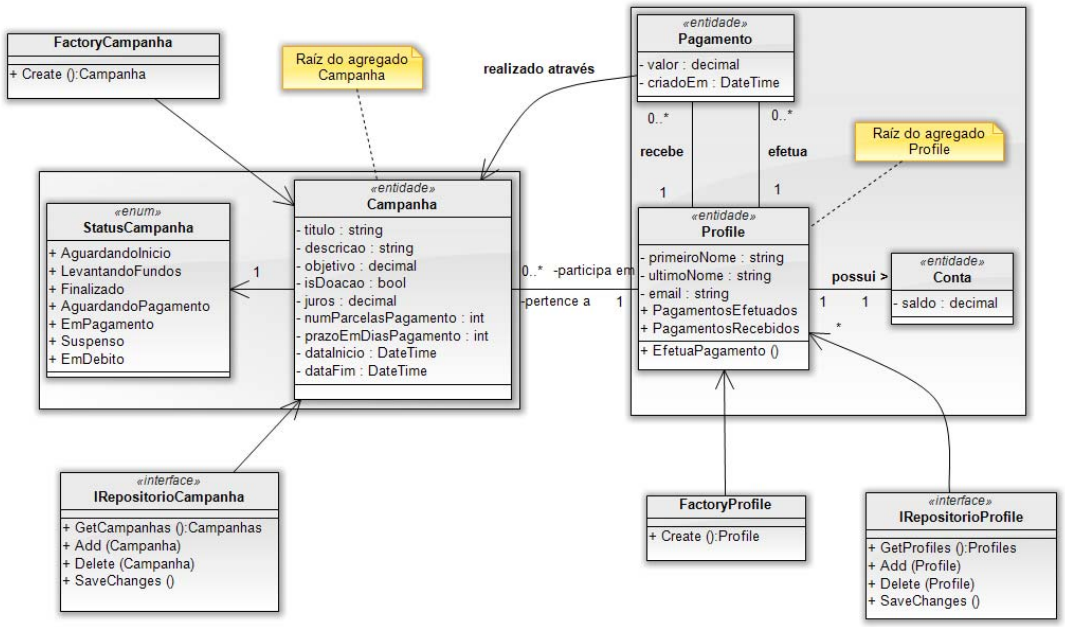
Armazenado; ao ser arquivado, ele passa para o estado Arquivado; finalmente, ao ser excluído, o objeto é de fato eliminado do domínio.

Quando um objeto encontra-se no estado Arquivado, a única operação que se pode executar sobre ele é a exclusão do domínio.

Os objetos no estado Armazenado podem ser reconstituídos, e então se tornam Ativos novamente. Na reconstituição dos objetos, entram em cena os repositórios. Os repositórios são os responsáveis pela recuperação e reativação dos objetos, atuando nos objetos do domínio globalmente acessíveis.

Desta forma, foram criados dois repositórios no modelo de domínio atual para fins de reconstituir os agregados previamente identificados, *Campanha* e *Profile*.

Na figura 10, é apresentado o diagrama do modelo do domínio com as entidades,



os objetos-valor, os agregados, as fábricas e os repositórios.

Figura 10 - Modelo nº 4

Como é possível verificar no diagrama da figura 10, os repositórios foram representados no modelo através de interfaces. Isto se deve a necessidade de expor dados e comportamentos do domínio sem comprometer o isolamento da camada de domínio, evitando que objetos do domínio interajam diretamente com a infra-estrutura. Caso ocorra, a interação direta entre os objetos do domínio e a infra-estrutura afetaria negativamente o baixo acoplamento desejado.

Outros artefatos do domínio, não menos importantes, são os serviços. Serviços são criados geralmente quando se identifica algum comportamento que não é facilmente associado aos outros objetos existentes.

No modelo corrente, foram identificados três comportamentos que não estão associados a nenhum objeto do domínio.

O comportamento *VerificarCampanhas* tem por objetivo identificar e supervisionar as mudanças de estado ocorridas nas campanhas. Uma campanha que tenha a sua data de inicio atingida, deverá sofrer uma mudança em seu estado, passando de *AguardandoInício* para *LevantandoFundos*, por exemplo. Este comportamento referencia várias instâncias de objetos Campanha, além de consultar dados dos seus proprietários e, por isso, não poderia ser um comportamento da entidade *Campanha*. Então, criou-se o serviço *ServicoCampanha* para implementar este comportamento.

Os outros dois comportamentos, *ValidarPagamento* e *Executar* (um pagamento), também foram incluidos em um servico denominado *ServicoPagamento*. O comportamento *ValidarPagamento* precisa verificar algumas regras do negócio, como por exemplo a verificação se o valor que esta sendo doado ou emprestado está dentro dos limites do objetivo (valor) da campanha, e se o usuário doador possui saldo suficientes para executar a operação. O comportamento em questão acessa vários objetos do domínio e por este motivo foi implementado como um serviço. A ação *Executar* tem por objetivo efetivar o pagamento. A implementação concreta do mesmo poderia ser realizada através de boleto bancário, cartão de crédito, ou outros mecanismos de pagamento *on-line*, acessando para isto APIs externas. Este comportamento, embora inerente ao domínio, precisa conhecer detalhes de tecnologias específicas. Desta forma, a sua especificação abstrata está definida em um serviço do domínio, mas o seu comportamento concreto deverá ser implementado na camada de infra-estrutura, para não comprometer o isolamento da camada de modelo.

Na figura 11, é exibido o diagrama de classes do modelo, já acrescido dos serviços, e na figura 12, um segundo diagrama, representando a transição de estados de uma entidade campanha, formando o modelo de domínio gerado a partir do estudo de caso apresentado.

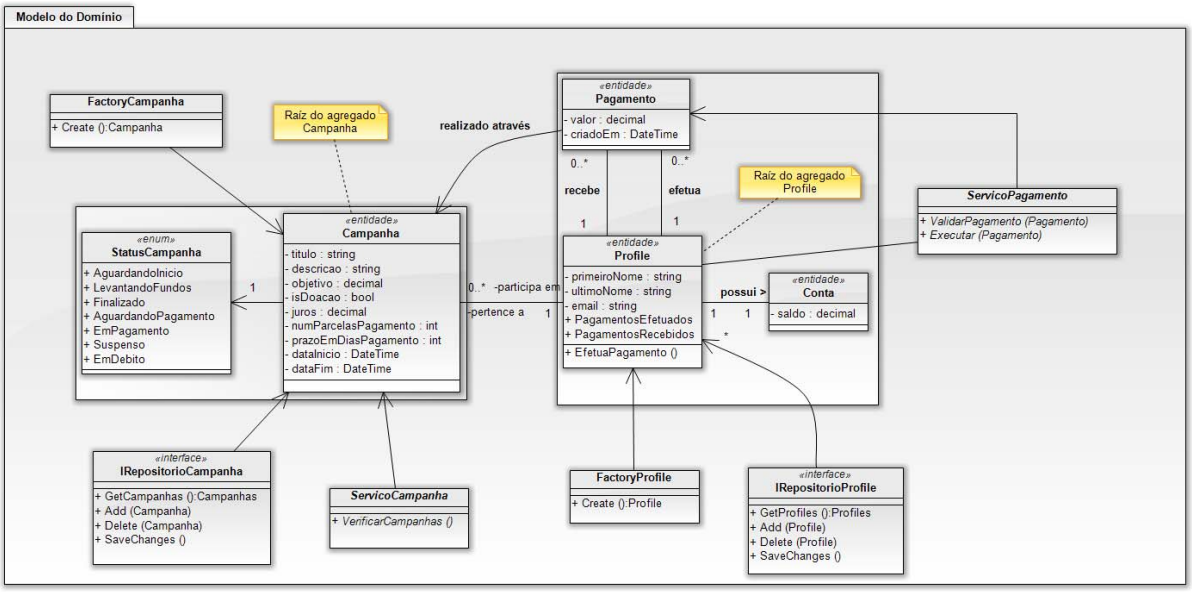


Figura 11 - Modelo nº 5

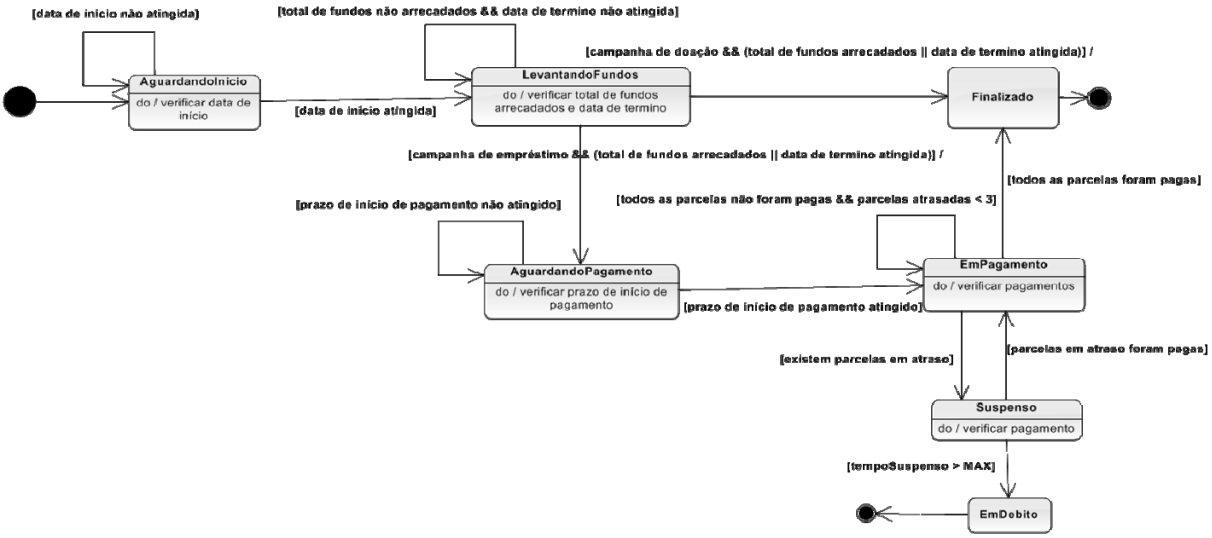


Figura 12 - Diagrama de Estados da entidade Campanha

4.3. A INFRA-ESTRUTURA

Neste ponto, o modelo de domínio elaborado transmite todo o conhecimento que se conseguiu extrair do estudo de caso apresentado anteriormente. No entanto, para não perder o foco na essência do domínio, algumas abstrações foram realizadas durante o processo de modelagem, de modo que seus detalhes concretos, dependentes das tecnologias utilizadas para implementação, ficaram a cargo da camada de infra-estrutura.

Desta forma, são os objetos da camada de infra-estrutura, estendendo e implementando os objetos abstratos definidos no domínio, que conhecem e se relacionam com as classes, módulos e APIs de acesso aos dados presentes no .NET Framework, e também outras bibliotecas e serviços disponíveis na internet.

A figura 13 exibe um novo diagrama, agora já com a camada de domínio e a camada de infra-estrutura presentes. Os pacotes presentes demonstram a separação das camadas.

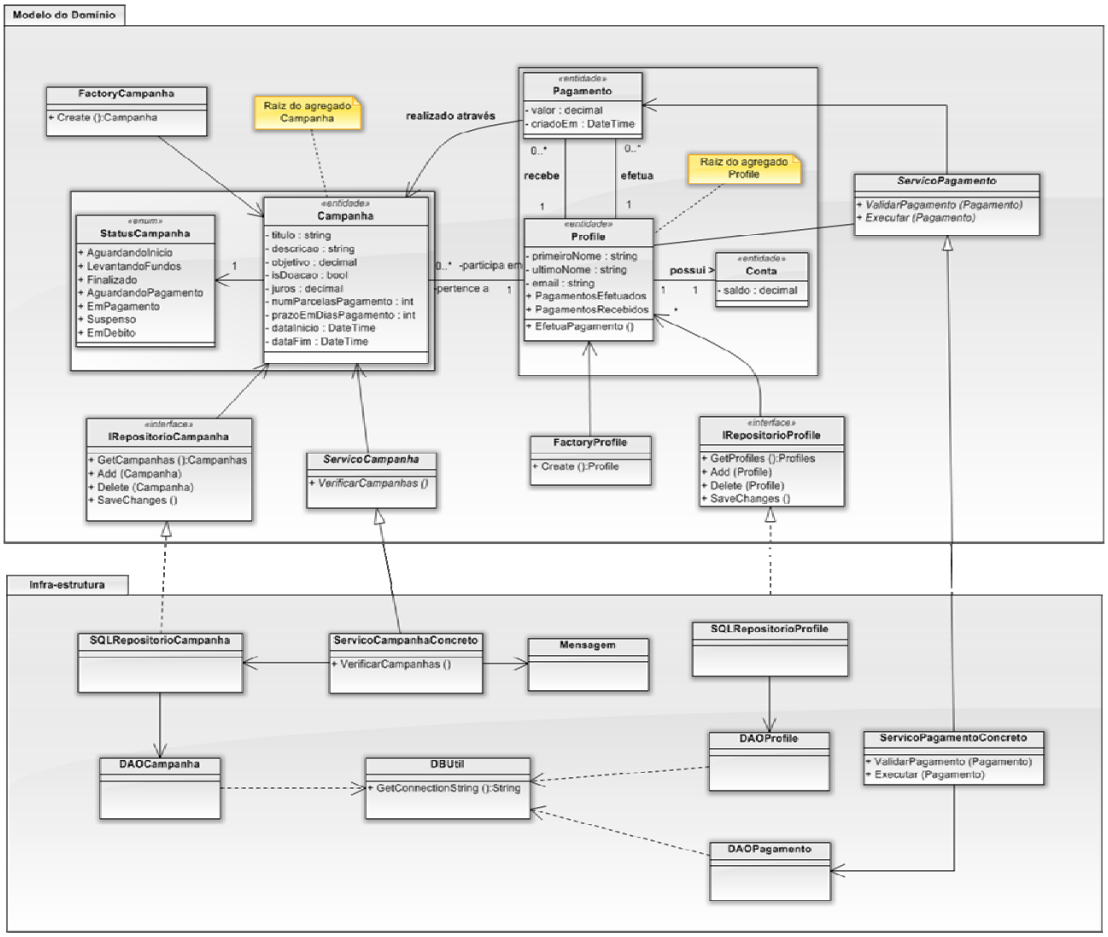


Figura 13 - Modelo nº 6

Na classe *DBUtil*, encontram-se presentes funcionalidades utilitárias para as classes que realizam acesso a dados, como por exemplo a recuperação da *string* de conexão com o servidor do software de banco de dados.

As classes DAO, *Data Access Objects*, são as classes que executam diretamente as operações no banco de dados, mapeando os objetos em dados relacionais e vice-versa.

As classes *SQLRepositorioProfile* e *SQLRepositorioCampanha* são as classes concretas que implementam o padrão repositório definido no modelo do domínio, servindo-se para isto das classes DAO.

A classe *ServicoPagamentoConcreto* implementa o comportamento do serviço de pagamentos definido na camada do domínio, validando e executando uma ação de pagamento de acordo com o mecanismo de pagamento adotado. Um exemplo possível seria pagamento por transferência on-line via *PayPal*. Neste caso, uma classe concreta se responsabilizaria por implementar os comportamentos definidos pelo serviço, encapsulando os detalhes do uso da API disponibilizada aos programadores pelo *PayPal*.

A classe *ServicoCampanhaConcreto* implementa o comportamento do serviço de campanhas, também definido na camada do domínio, checando e atualizando o status das campanhas em funcionamento e enviando mensagens aos usuários do sistema, toda vez que o status for alterado e existam pagamentos pendentes em uma campanha.

O isolamento da camada de domínio e a separação dos detalhes concretos da implementação em uma camada de infra-estrutura evitam que o modelo seja "contaminado" por uma tecnologia específica, e garantem ao software melhor capacidade de manutenção e expansão, visto que o código de uma funcionalidade qualquer pode ser reescrito para dar suporte a novas tecnologias sem com isto impactar na lógica do domínio.

4.4. VISÃO GERAL DA ARQUITETURA

Com a conclusão do estudo acerca das camadas de modelo e de infra-estrutura propostos pelo DDD, é chegado o momento de relacionar estas camadas com a arquitetura proposta para a aplicação, o MVC.

O MVC propõe uma camada de Visão, que pode ser diretamente relacionada com a camada de Interface do Usuário proposta pelo DDD, visto que definem as mesmas responsabilidades.

A camada de controle do MVC é responsável pela lógica da aplicação, mesma responsabilidade da camada do Aplicativo do DDD.

A camada de Modelo do MVC é responsável pela lógica do domínio, persistência dos dados e outros serviços. No DDD, estas responsabilidades são separadas em duas camadas, que são o Domínio e a Infra-estrutura. Como conseqüência, a camada de Modelo do MVC engloba as responsabilidades das camadas de Domínio e de Infra-estrutura do DDD.

O diagrama ilustrado na figura 14 representa conceitualmente a disposição das camadas identificadas dentro da arquitetura MVC, utilizada no desenvolvimento da aplicação proposta.

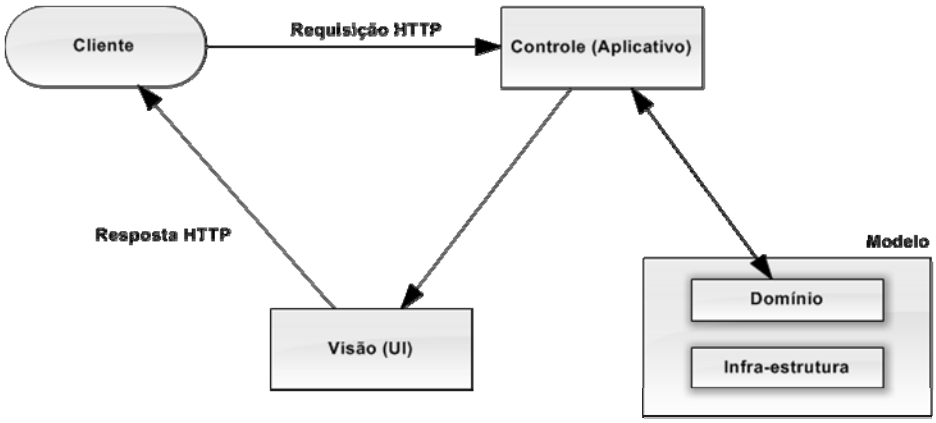


Figura 14 – Visão geral da arquitetura da aplicação proposta

5. IMPLEMENTAÇÃO

Após analisar o estudo de caso, elaborar um modelo que refletisse os conceitos do domínio, e por fim definir a arquitetura a ser utilizada, iniciou-se o processo de implementação da aplicação proposta.

Neste capítulo, são apresentados os detalhes desta implementação, através do exemplo de algumas dos principais histórias de usuário cobertas pelo software.

5.1. PROPOSTA DE IMPLEMENTAÇÃO

A plataforma escolhida para implementar o aplicativo foi o ASP.NET, utilizando para isto a linguagem C#. Como base para o desenvolvimento da aplicação, utilizou-se o framework ASP.NET MVC, e para o desenvolvimento dos testes unitários foi utilizado o framework NUnit, como já citado em capítulos anteriores.

A figura 15 mostra a estrutura lógica dos arquivos que compõem o projeto, organizados em quatro projetos distintos, representando as camadas identificadas ao longo deste estudo.

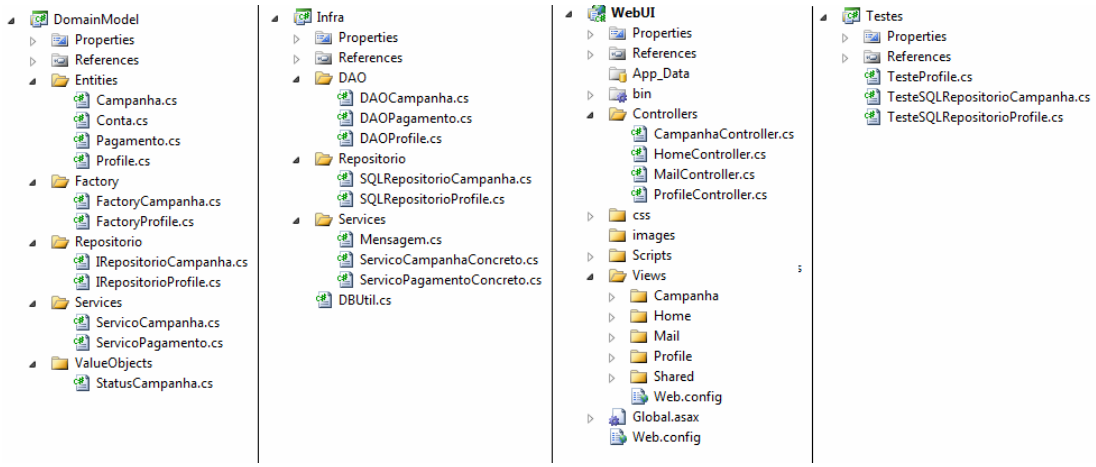


Figura 15 - Os projetos que compõem a aplicação

O projeto *DomainModel* é do tipo biblioteca de classes, e contém a implementação do modelo do domínio. Nele foram criadas todas as classes e interfaces especificadas no modelo. Como conseqüência do isolamento do modelo, este projeto não faz referência a nenhum dos outros três projetos existentes na aplicação.

O projeto *Infra* também é do tipo biblioteca de classes, e agrupa os arquivos responsáveis pelas classes que dão suporte ao modelo, implementando as suas

especificações e oferecendo serviços como o acesso aos dados. Este projeto faz referência ao projeto *DomainModel*, e também a bibliotecas de acesso a dados, como o ADO.NET.

O projeto *WebUI* é do tipo ASP.NET MVC Web Application, e é composto pelos arquivos que implementam a camada de controle e a camada de visão da aplicação. Como exposto na figura 15, as classes que em conjunto formam a camada controladora do MVC estão organizadas na pasta *Controllers*. Já as páginas e classes que compõem a camada de visão encontram-se organizadas na pasta *Views.* Este projeto faz referência tanto ao projeto *DomainModel* como ao projeto *Infra*.

O último projeto, *Testes*, agrupa todas as classes que compõem o conjunto de testes unitários desenvolvidos ao longo do processo de desenvolvimento. Como foram desenvolvidos testes unitários para cobrir funcionalidades do modelo e da infra-estrutura, este projeto faz referência aos projetos *DomainModel* e *Infra*.

O diagrama da figura 16 mostra como ficou a arquitetura com as camadas desenvolvidas.

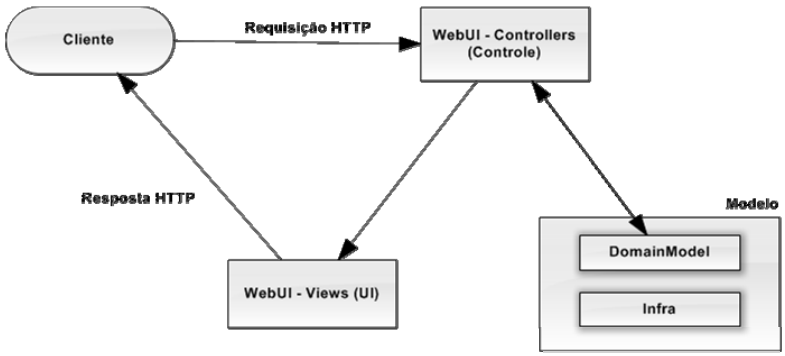


Figura 16 - Os projetos que compõem a solução e a arquitetura

5.2. HISTÓRIAS DE USUÁRIOS

A seguir, são apresentadas as principais histórias de usuários presentes na aplicação. O objetivo é ilustrar a sua implementação, dando ênfase a aspectos relacionados ao tema deste trabalho, através da descrição do fluxo de execução das histórias selecionadas e dos testes realizados para garantir o cumprimento do objetivo de cada uma.

História #1: Cadastrar uma campanha

Descrição: O usuário acessa a aplicação para cadastrar uma campanha. É preciso informar o nome e a descrição da campanha, o valor (objetivo) que se deseja arrecadar, o tipo de campanha (empréstimo ou de doação), as condições de pagamento do valor arrecadado, no caso de empréstimo, e o período de duração da campanha. Após a criação da campanha, o usuário deverá visualizá-la para verificar se os dados cadastrados estão corretos.

Fluxo de execução:

Figura 17 - Tela inicial

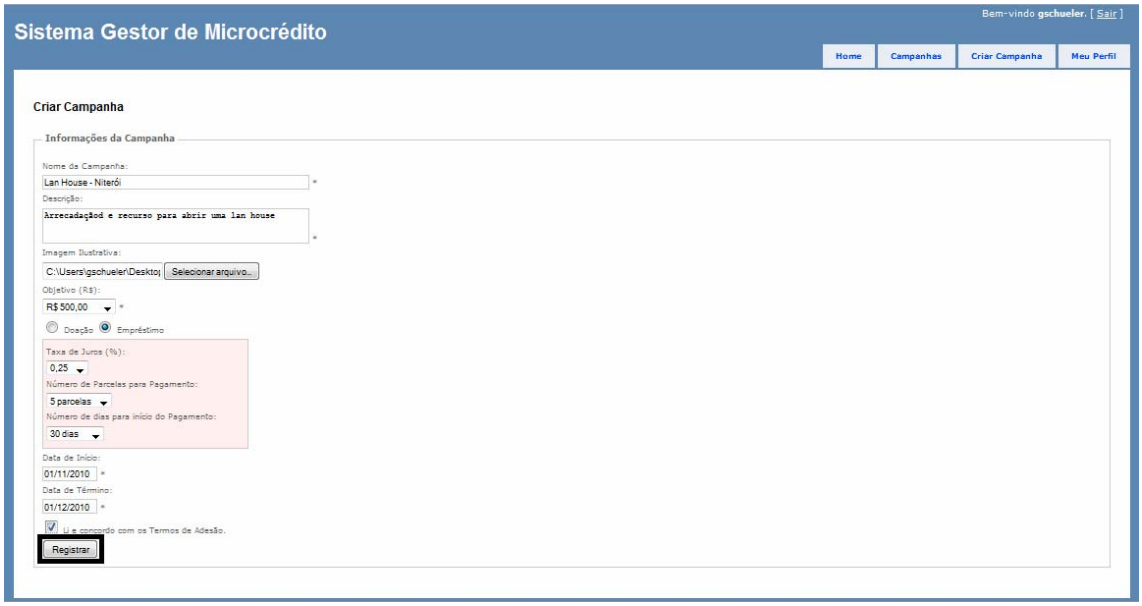


Figura 18 - Formulário para criação de campanha

Figura 19 - Tela de confirmação

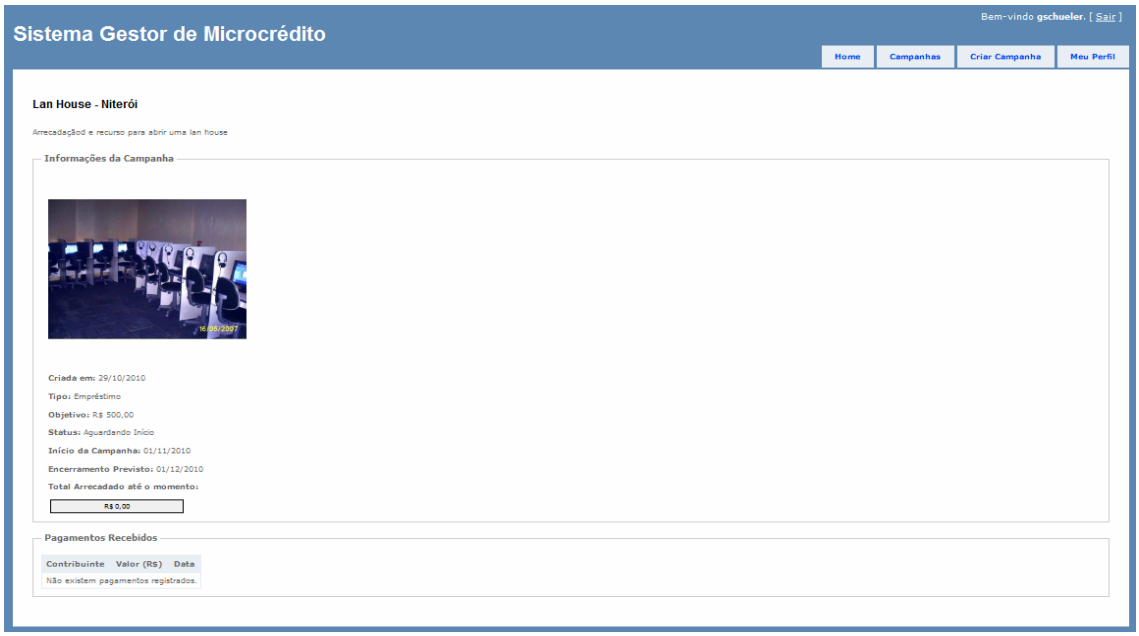


Figura 20 - Detalhes da campanha recém-criada

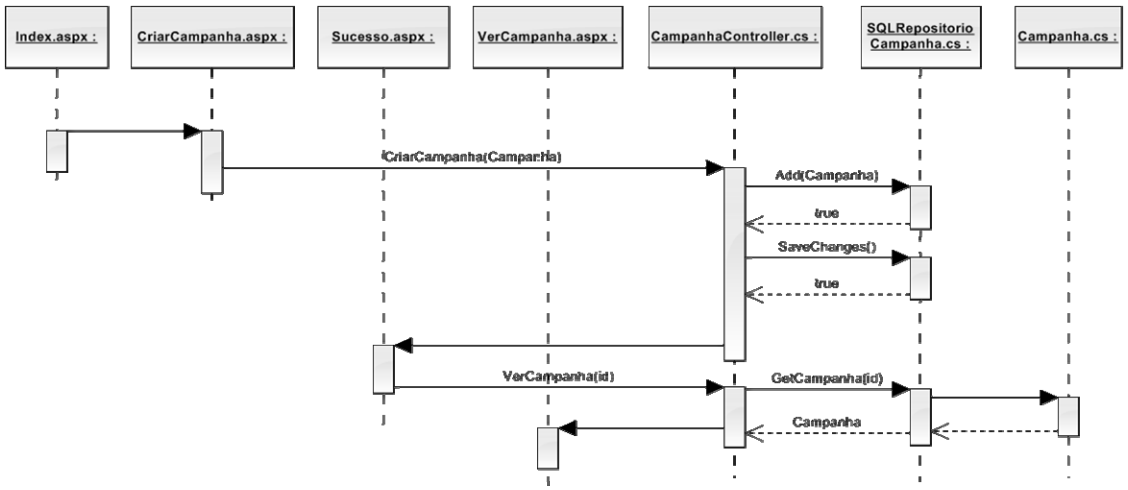


Figura 21 - Diagrama de Seqüência da história 'Criar Campanha'

No diagrama ilustrado na figura 21, percebe-se claramente o fluxo existente entre as camadas, bem como o papel centralizador e direcionador exercido pela camada de controle do padrão MVC. A classe *CampanhaController* possui a lógica de aplicação que constitui o elo de ligação entre as camadas de modelo e visão, selecionando e cadenciando a execução dos algoritmos necessários para a resolução do problema identificado na história do usuário.

Durante a implementação, para garantir que o código implementado atingisse o objetivo da história, foi elaborado o seguinte teste:

```
[SetUp]
public void Init()
{
    /* Arrange */
    string userId = "112452A1-9ED4-4BC4-8E18-929F09434771";
    Guid userIdGuid = new Guid(userId);
    FactoryCampanha factoryCampanha = new FactoryCampanha();
    FactoryProfile factoryProfile = new FactoryProfile();
    this.campanha = factoryCampanha.Create();
    this.campanha.Proprietario = factoryProfile.Create(userIdGuid);
    this.campanha.Titulo = "Teste de Título";
    this.campanha.Descricao = "Teste de Descrição";
    this.campanha.IsDoacao = true;
    this.campanha.Objetivo = 1000;
    this.campanha.Status = StatusCampanha.Aguardando;
    this.campanha.DataInicio = DateTime.Today;
    this.campanha.DataFim = DateTime.Today;
    this.campanha.CriadoEm = DateTime.Now;
    this.campanha.AtualizadoEm = DateTime.Now;
    this.campanha.Views = 0;
}
```

Figura 22 - Método de inicialização do teste

```
[Test]
public void TestaRepositorioAddCampanha()
{
    /* Act */
    this.repositorio.Add(campanha);
    bool result = repositorio.SaveChanges();

    /* Assert */
    Assert.IsTrue(result, "Deveria ser true, mas é false.");
}
```

Figura 23 - Código do teste unitário para Adicionar Campanha

O teste cuja listagem está exibida na figuras 22 e 23 salva uma campanha no banco de dados utilizando o *SQLRepositorioCampanha*. Como se pode ver, o esperado é que o resultado deste teste seja "*true*", caso contrário, o código não está executando como deveria. Abaixo é mostrada a execução do teste com o framework de testes NUnit.

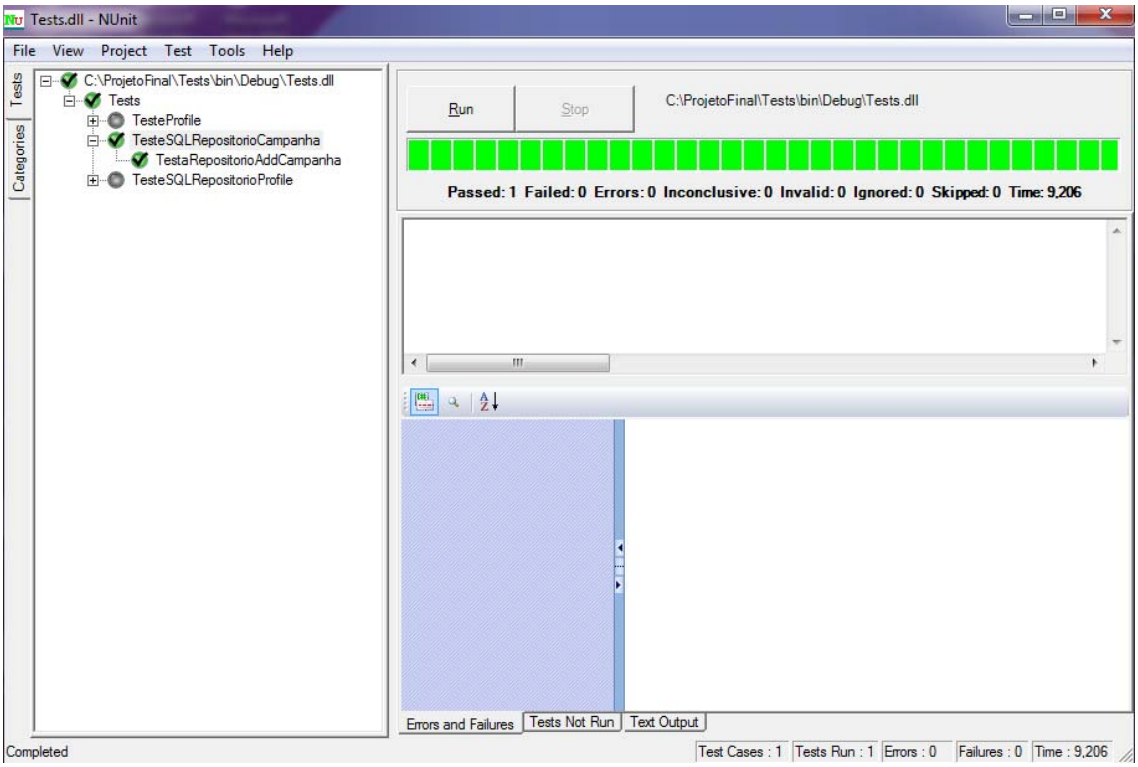


Figura 24 - Execução do teste 'Adicionar Campanha' no NUnit

História #2: Emprestar ou doar recursos a uma campanha.

Descrição: O usuário acessa a aplicação para doar ou emprestar recursos através de uma campanha. Para isto, deverá escolher uma campanha dentre todas as que estão cadastradas no sistema e possuam as condições adequadas. Uma condição é adequada quando a campanha tenha atingido a sua data de início para a etapa de arrecadação. Após a escolha, o usuário deverá informar o valor e os dados para pagamento, e em seguida confirmá-los. Após efetuar o pagamento, a campanha é atualizada mostrando o total de recursos que já foram arrecadados.

Fluxo de execução:

Figura 25 – Tela inicial

Figura 26 – Lista de Campanhas

Figura 27 – Detalhes da campanha selecionada

Figura 28 – Tela de contribuição

Figura 29 – Tela para confirmar a contribuição

Figura 30 – Tela para efetuar o pagamento

Figura 31 – Confirmação do pagamento

Figura 32 – Dados da Campanha atualizados após contribuição

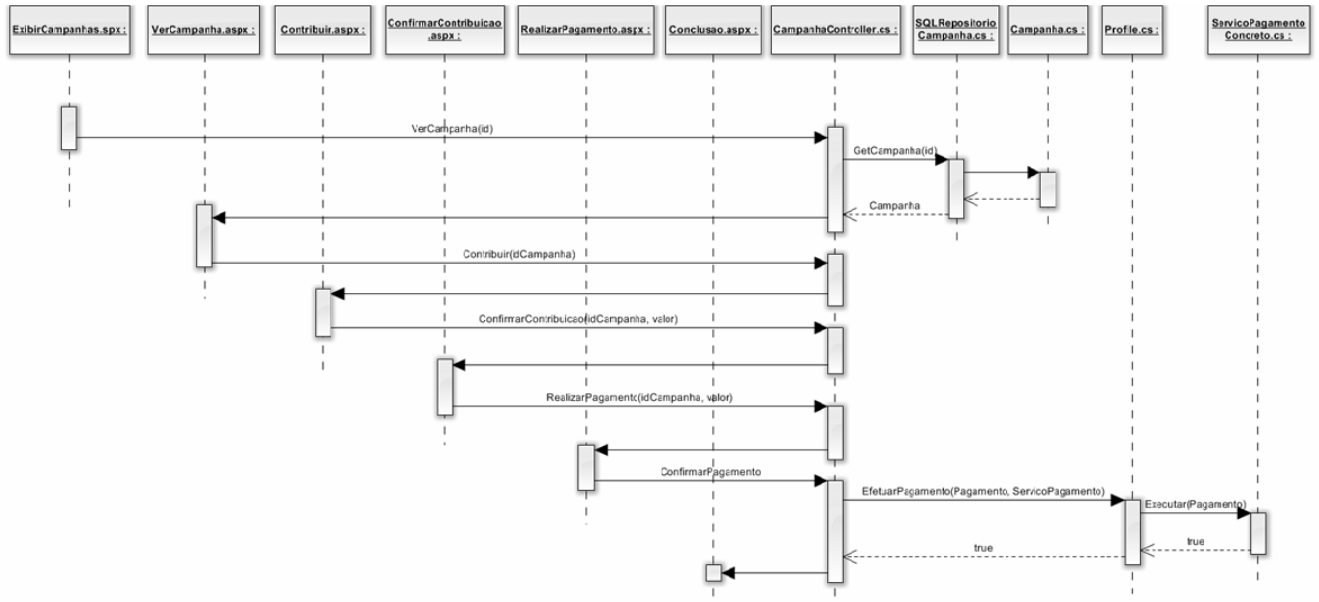


Figura 33 – Diagrama de seqüência da história 'Emprestar ou Doar recursos a uma Campanha'

Nas figuras 34 e 35, estão o código e a tela de execução do teste unitário desenvolvido para esta história, respectivamente.

```
[Test]
public void TestaEfetuaPagamento()
{
    /* Arrange */
    Pagamento pagamento = new Pagamento(this.campanha,
        this.profileOrigem, this.profileDestino,
        100.25m, 0, "0000999988887777", "512", "American Express",
        "Ricardo Costa Abdalla", "09/2010", DateTime.Now );

    ServicoPagamento servico = new ServicoPagamentoConcreto();

    /* Act */
    bool result = this.profileOrigem.EfetuaPagamento(pagamento, servico);

    /* Assert */
    Assert.IsTrue(result, "Retornou false.");
}
```

Figura 34 - Código do teste unitário para a realização de um pagamento

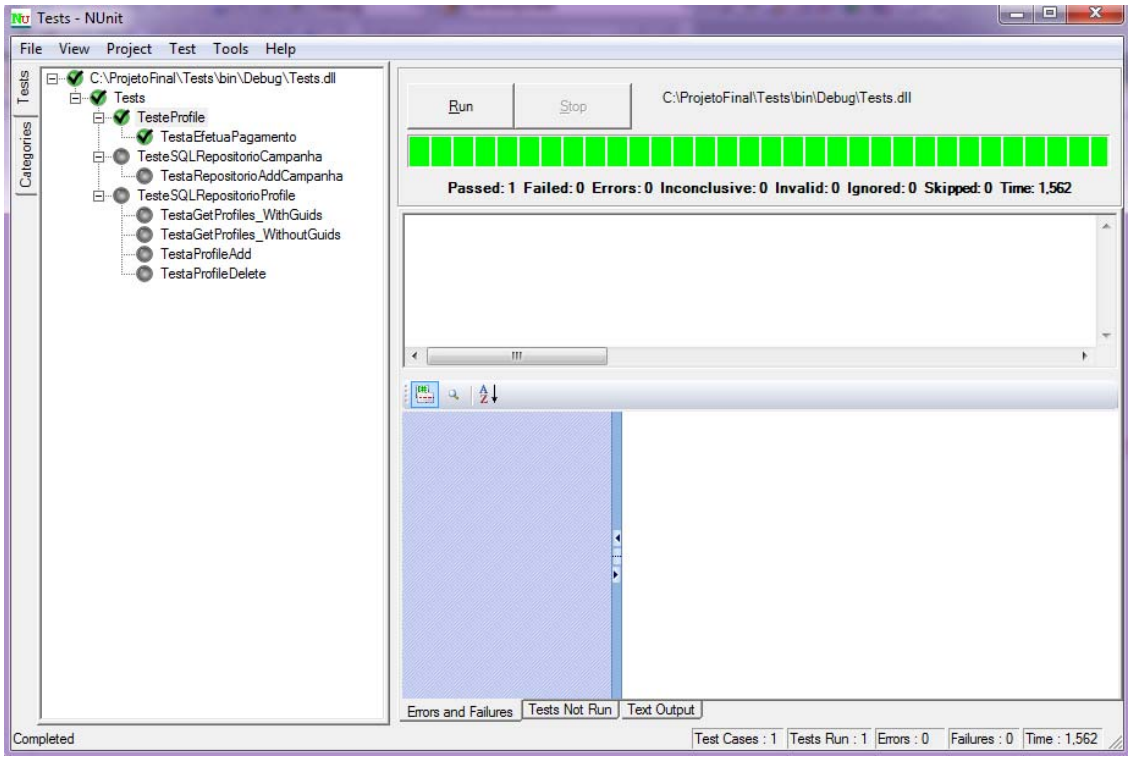


Figura 35 - Execução do teste 'Efetua Pagamento' no NUnit

História #3: Pagar um empréstimo obtido através de uma campanha.

Descrição: Quando a data estipulada para o início dos pagamentos referentes ao empréstimo obtido por uma campanha é atingida, o usuário proprietário visualiza uma lista dos pagamentos a serem realizados. O sistema gera as informações dos pagamentos com os valores já calculados (incluindo taxas). O usuário seleciona a parcela atual do pagamento, informa os dados do meio de pagamento (Ex.: cartão de crédito) e confirma a sua realização. O sistema então distribui o valor pago entre os credores da campanha, depositando em suas contas as quantias pertinentes a cada um. Após a conclusão do pagamento, o usuário proprietário pode visualizar todos os pagamentos efetuados e os ainda pendentes na tela de sua campanha.

Fluxo de execução:

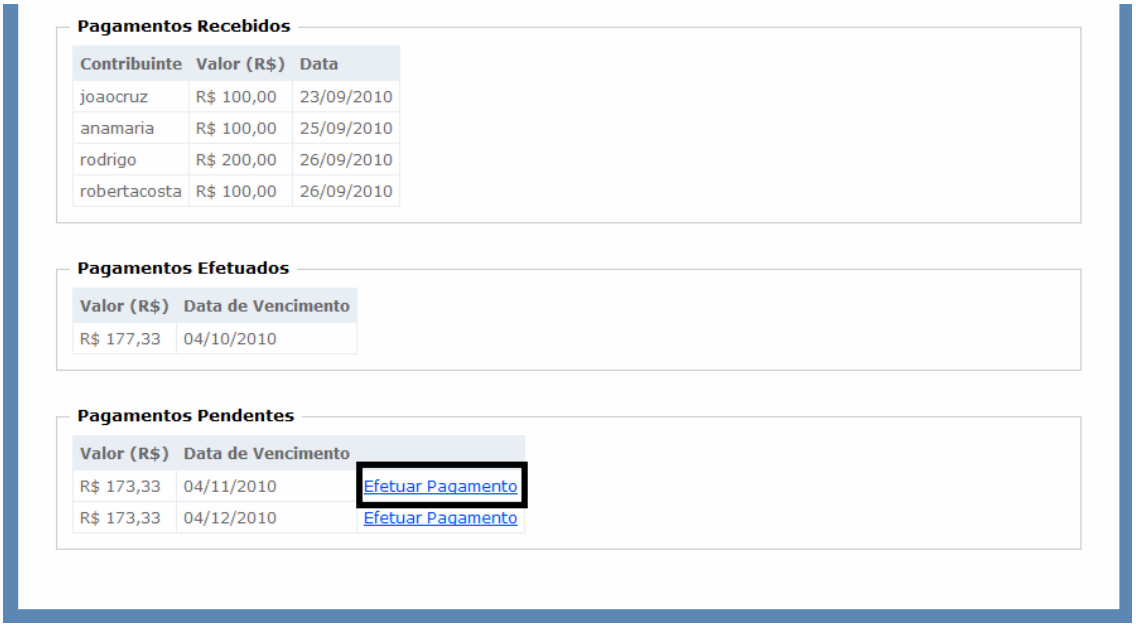


Figura 36 - Lista de pagamentos de uma campanha

Figura 37 - Tela para efetuar pagamento

Figura 38 - Tela Pagamento de empréstimo concluído

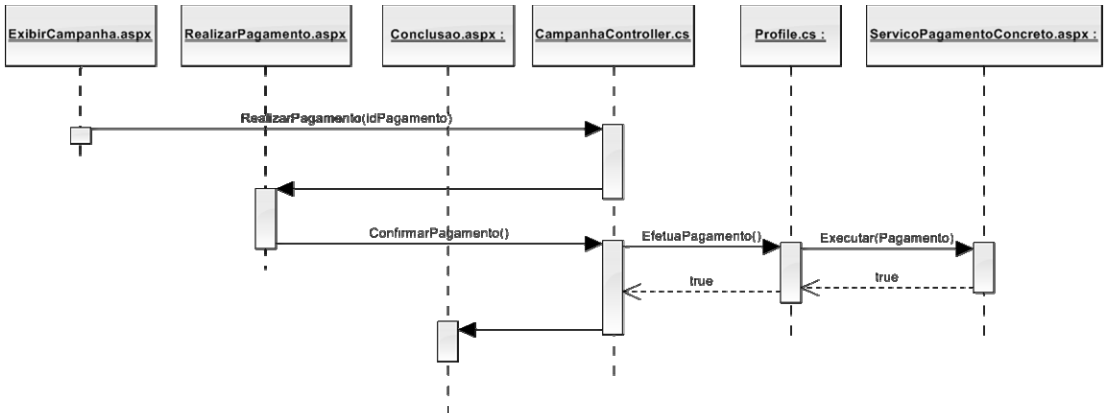


Figura 39 - Diagrama da história 'Pagar Empréstimo obtido através de uma Campanha'

Observação: o teste unitário que cobre a execução de um pagamento nesta história é o mesmo utilizado no item anterior (História #2).

6. CONCLUSÃO

Este trabalho propôs, ao longo de seus capítulos, uma prática de desenvolvimento para aplicações Web capaz de reunir e integrar os benefícios da modelagem orientada ao domínio e do desenvolvimento orientado a testes ao já consagrado padrão de arquitetura MVC.

6.1. VIABILIDADE DO MODELO PROPOSTO

O modelo de desenvolvimento proposto mostrou-se bastante adequado para o ambiente volátil e dinâmico das aplicações Web. Muitas destas aplicações são desenvolvidas por equipes cujas formações e habilidades são bastante diferenciadas, como por exemplo, designers, programadores, arquitetos de software, administradores de banco de dados, especialistas do negócio, entre outros, e a separação do software em camadas bem definidas torna-se um requisito fundamental para viabilizar projetos desta natureza. Também não são raros os exemplos de aplicações Web que atuam como *front-ends* de sistemas mais complexos, onde o modelo do domínio precisa ser compartilhado com outros subsistemas, quase sempre operando a partir de computadores remotos utilizando sistemas operacionais e tecnologias diferentes.

Outro ponto a ser mencionado é a rapidez com que as aplicações Web precisam responder as novas demandas dos usuários. O geralmente curto intervalo de tempo existente entre o lançamento de versões faz com que o processo evolutivo deste tipo de software seja mais acelerado do que o encontrado em outros ambientes de desenvolvimento. Cabe lembrar que a *World Wide Web* é uma plataforma aberta e heterogênea, e, por este motivo, mudanças constantes nos requisitos constituem praticamente uma regra geral.

Desta forma, os autores entendem que o modelo aqui discutido é viável. A separação em camadas viabiliza a realização de trabalho paralelo, o emprego de equipes multidisciplinares e também facilita a manutenção durante o ciclo de vida da aplicação. O design orientado ao modelo é capaz de melhorar a absorção de conhecimento durante o processo de desenvolvimento, enriquecendo gradativamente o modelo do domínio e, com isso, gerando mais valor aos usuários do software. O desenvolvimento orientado a testes contribui com ganho de confiabilidade, na medida em que partes consideráveis do código passam a ser cobertas por pelo menos um dos testes unitários, além de buscar

atacar os defeitos existentes antes que estes ganhem maior proporção, tornando-se mais caros e difíceis de serem corretamente tratados.

Por fim, é importante notar que este trabalho não apresenta uma solução completa para o problema do desenvolvimento de aplicativos para a Web. A adoção de uma metodologia adequada faz-se necessária. Pelas particularidades inerentes ao ambiente de desenvolvimento Web, já anteriormente citadas, e de acordo com o estudo comparativo entre metodologias ágeis e metodologias ditas tradicionais realizado por Engels, Lohman e Wagner [21], os autores acreditam que o modelo aqui discutido pressupõe a utilização de metodologias ágeis de desenvolvimento, mais adequadas ao ritmo das mudanças imposto pela plataforma.

6.2. TRABALHOS FUTUROS

As questões abordadas neste trabalho deram ênfase a alguns aspectos importantes do processo de desenvolvimento de software, como a escolha de uma arquitetura apropriada e o projeto de modelos eficazes e testáveis. São questões de cunho técnico, diretamente relacionadas ao produto final, o software, mas que pouco dizem sobre os aspectos gerenciais presentes no processo de desenvolvimento.

Temas como o gerenciamento das equipes e dos recursos disponíveis, o controle do fluxo de execução das atividades, a adaptabilidade e o tempo de resposta das equipes às mudanças, são algumas das questões sempre presentes no cotidiano dos projetos de software, e por isto constituem um campo natural de continuidade do presente trabalho.

Surgido inicialmente nas indústrias automotivas, o Scrum destaca-se atualmente como metodologia de gerenciamento de projetos de software. O Scrum é um processo ágil, iterativo e incremental, baseado em equipes pequenas e multidisciplinares, portando bastante adequado para a grande maioria dos projetos de software direcionados para a Web. Entre as suas principais características, o Scrum propõe uma atuação ativa do cliente junto à equipe de desenvolvimento, reuniões diárias entre os membros da equipe para discutir o andamento das tarefas e eventuais empecilhos para a sua execução, elaboração e constante atualização de um plano para mitigação de riscos e entregas freqüentes de funcionalidades plenamente operacionais.

A característica do Scrum de buscar aproximar o cliente da equipe de desenvolvimento é particularmente interessante, pois apresenta uma relação de complementaridade com proposições do DDD, como por exemplo, a estreita colaboração entre especialistas do negócio (o cliente) e a equipe do desenvolvimento, o que, como já visto no capítulo 2, leva a necessidade de uma linguagem comum, capaz de viabilizar a comunicação entre ambos os grupos. Outro ponto de contato entre o DDD e o Scrum é o processo incremental e iterativo, proposto como técnica na construção do modelo do domínio pelo DDD, e estendido a todo o projeto de software pelo Scrum.

Assim sendo, os autores do presente trabalho entendem que a sua continuidade deverá estudar a integração, no modelo proposto, das metodologias ágeis de desenvolvimento, tendo preferência pelo Scrum em decorrência do que foi exposto no parágrafo anterior, sem, no entanto, ignorar a contribuição que outras metodologias

ágeis, como o XP, podem vir a oferecer.

7. REFERÊNCIAS BIBLIOGRÁFICAS

[1] Mosaic Web Browser History, NCSA, Marc Andreessen, Eric Bina. Disponível em <http://www.livinginternet.com/w/wi_mosaic.htm>. Acesso em 9 de junho de 2010.

[2] A Brief History of Developing Web Applications, Intelliware Development Inc.. Disponível em < http://i-proving.ca/space/Technologies/A+Brief+History+of+Developing+Web+Applications>. Acesso em 9 de junho de 2010.

[3] Daniel M. Brandon. Software Engineering for Modern Web Applications – Methodologies and Technologies, IGI Global, 2008. Pág. 30.

[4] Wikipedia. Model-View-Controller. Disponível em <http:// en.wikipedia.org/wiki/Model–View–Controller>. Acesso em 20 de junho de 2010.

[5] Ibidem.

[6] Eric Evans, Domain-Driven Design – Atacando as complexidades no coração do software, Alta Books, 2009.

[7] Ibidem. Páginas 41-57.

[8] Ibidem. Pág. 64.

[9] Ibidem. Páginas 65-66.

[10] Ibidem. Páginas 84-88.

[11] Ibidem. Páginas 92-94.

[12] Ibidem. Páginas 119-128.

[13] Ibidem. Páginas 99-103.

[14] Ibidem. Páginas 129-132.

[15] Ibidem. Páginas 140-153.

[16] Scott W. Ambler. Introduction to Test Driven Development (TDD). Disponível em <http://www.agiledata.org/essays/tdd.html>. Acesso em 22 de junho de 2010.

[17] Ibidem.

[18] Microsoft, ASP.NET MVC 2. Disponível em <http://msdn.microsoft.com/en-us/library/dd394709.aspx>. Acesso em 6 de junho de 2010.

[19] The Code Project, ASP.NET MVC Architecture. Disponível em <http://www.codeproject.com/KB/dotnet/MVC_architecture.aspx>. Acesso em 10 de junho de 2010.

[20] NUnit. Disponível em <http://www.nunit.org/ >. Acesso em 18 de junho de 2010.

[21] Gerti Kappel [et al.]. Web Engineering – The Discipline of Systematic Development of Web Applications, John Wiley & Sons, 2006. Capítulo 10, The Web Application Development Process.