Aplicação de padrões de projeto no desenvolvimento de software para a melhoria de qualidade e manutenibilidade

Junior Sturm, Madalena Pereira da Silva

Programa de Pós-Graduação em Engenharia de Software
Universidade do Planalto Catarinense (UNIPLAC) – Lages, SC – Brazil

{juniorsturm, madalenapereiradasilva}@gmail.com

Abstract. *With the increasing demand for the development of Information Systems, which tend to be scalar, complex and constantly changing, there is the need to implement an efficient control to improve the quality and maintainability of the source code. This article proposes to use design patterns to improve the code quality and the benefits found during maintenance or implementation of new features. The methodology consisted of an integrative review of design patterns, MVC (Model - View - Controller) and software maintenance. As a result it is presented a case study to evaluate the benefits of using design patterns and MVC in maintenance and software quality.*

Resumo. *Com a crescente demanda pelo desenvolvimento de Sistemas de Informação, que tendem a serem escalares, complexos e em constantes mudanças, surge a necessidade da implantação de um controle eficiente para a melhoria da qualidade e manutenibilidade do código fonte. Este artigo tem como finalidade empregar padrões de projetos para a melhoria da qualidade do código e os benefícios encontrados durante a manutenção ou implementação de novas funcionalidades. A metodologia consistiu numa revisão integrativa sobre padrões de projeto, MVC (Model – View – Controller) e manutenção de software. Como resultado é apresentado um estudo de caso para avaliar as vantagens do uso padrões de projeto e MVC na manutenção e qualidade do software.*

1. Introdução

Com a complexidade e a escalabilidade das funções dos softwares, faz-se necessário a adoção de uma padronização durante o processo de geração de código, evitando-se assim, problemas recorrentes encontrados durante a execução do projeto. A partir deste contexto os Padrões de Projetos começaram a ser aplicados a problemas clássicos encontrados durante o processo de desenvolvimento de softwares (VALENTIM, 2005).

A adoção de padrões de projetos tem como finalidade aumentar o entendimento do software, sendo considerada uma parte da documentação do sistema. O uso de padrões tem como intuito a produção de sistemas eficientes com características de reusabilidade, extensibilidade e manutenibilidade, fornecendo uma solução completa e melhor estruturada ao invés de uma solução imediata (LINO, 2011).

Silva (2009) destaca que para que um projeto alcance o sucesso, além do uso de padrões, faz-se necessário o emprego de uma arquitetura no processo de criação e desenvolvimento do sistema. A arquitetura irá auxiliar na organização em componentes da

aplicação, promover a independência dos módulos, alcançando objetivos como: eficiência, escalabilidade, reutilização e facilidade na manutenção do sistema (SILVA, 2009).

Diante do contexto exposto, este trabalho tem como objetivo o emprego de padrões de projetos no desenvolvimento de software sobre a arquitetura MVC (*Model–View–Controller*) para a melhoria da qualidade e a manutenibilidade do sistema. O artigo está organizado como segue. A seção 2 descreve os padrões de projetos. A seção 3 apresenta a arquitetura MVC. A seção 4 descreve a manutenção do software. A seção 5 apresenta o estudo de caso. A seção 6 apresenta as considerações.

2. Padrões de Projeto

Durante o desenvolvimento de software, a equipe procura aplicar técnicas que melhorem a qualidade do código, deixando-o legível, organizado e de fácil manutenção. Como resultado, há grandes chances de se obter um sistema livre de defeitos e muito próximo do desejado.

2.1. Categorias dos padrões de projetos

A classificação dos padrões *Gang of Four* (GoF) (GAMA et al., 2006) descreve um catálogo de vinte e três padrões de projetos, dividindo-os em três categorias:

- *Criação*: são voltados à criação dos objetos, sendo criado um padrão de implementação para cada política necessária, por exemplo, criação de uma única instância de um objeto.
- *Estruturais*: definem uma forma comum de organizar classes e objetos em um sistema.
- *Comportamento*: definem o comportamento de objetos e como eles irão se relacionar, oferecendo comportamentos especiais para uma variedade de situações.

2.2. Padrões de projetos usados no estudo de caso

Para o desenvolvimento deste trabalho foram adotados os padrões de projeto *Singleton*, *Factory Method* e *Façade*, descritos a seguir.

2.2.1. Singleton

O *Singleton* garante que uma classe ou objeto tenha uma única instância acessível de maneira global (LINO, 2011). O *Singleton* necessita do método responsável pela instanciação do objeto. Este método tem como objetivo verificar se o objeto requisitado foi instanciado. Se o mesmo tiver sido instanciado, o método apenas devolve ao solicitante a instância previamente feita na classe. Do contrário, o método se responsabiliza pela nova instanciação da classe. Para assegurar que a classe tenha somente uma instância, deve-se declarar o seu construtor como privado ou protegido (SHALLOWAY, 2004).

A classe, com dois métodos, ilustrada na Figura 1 simplifica o funcionamento do *Singleton*. O método privado *Singleton()* é o construtor da classe. O método público *getInstance()* é responsável por realizar a instanciação da própria classe a qual pertence.

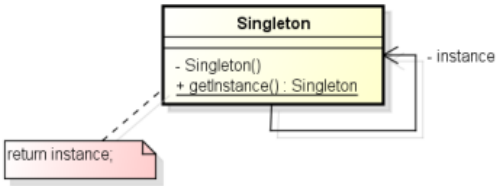

Figura 1. Classe com ilustração do funcionamento do *singleton* (FONTE: LINO, 2011)

2.2.2. Factory method

O *Factory Method* tem como proposta ajudar na atribuição de responsabilidade para a criação de objetos. Segundo a GoF, o *Factory Method* deve definir uma interface para a criação de um objeto, permitindo que uma classe possa delegar a instanciação para subclasses, e essas decidem qual classe deve ser instanciada (SHALLOWAY, 2004).

Lino (2011) destaca que o *Factory Method* proporciona maior abstração ao sistema, não trabalhando diretamente com a classe que será instanciada e podendo ser adotado como uma fábrica de objetos genéricos, criando de maneira transparente os objetos.

A Figura 2 apresenta parte de um diagrama de classes para ilustrar o funcionamento deste padrão. O diagrama mostra: (i) a classe *Creator* abstrata, a qual possui o método *FactoryMethod()* que será implementado nas subclasses, com o objetivo de criar uma classe *Product*. Nela também podem ser encontrados um ou mais métodos, cada qual com seu comportamento e que invocarão o *FactoryMethod*. (ii) A classe *ConcreteCreator* sobrescreve o método *FactoryMethod()*, retornando o objeto da classe *ConcreteProduct*.

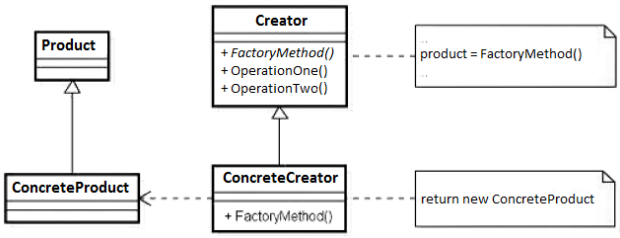

Figura 2. Diagrama de classe do padrão de projeto *factory method* (FONTE: LINO, 2011)

2.2.3. Façade

O *Façade* tem como objetivo evitar que códigos e regras de interface visual se misturem com as regras de negócios. O uso deste padrão torna o código legível e de fácil manutenção (SOUZA, 2011). O *Façade* foi definido pela GoF para fornecer uma interface de mais alto nível, com objetivo de unificar um conjunto de interfaces de subsistemas para facilitar a utilização das mesmas (SHALLOWAY, 2004).

Com a proposta de se ter uma classe que exerça o papel de interface entre as classes da aplicação e seus subsistemas, o *Façade* irá receber da aplicação todas as chamadas que se referem aos subsistemas e através de um método desta classe, encaminhará a requisição a um método correspondente, presente dentro de um subsistema (KROTH, 2000).

A Figura 3 (lado esquerdo) representa um sistema que não utiliza o padrão *Façade*. Nota-se que cada classe da aplicação que usa métodos presentes em diferentes subsistemas necessita realizar a instanciação de cada classe.

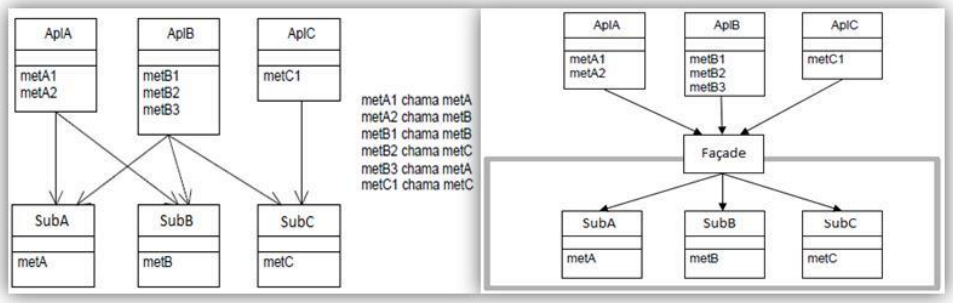

Figura 3. Associação entre aplicação e subsistemas com e sem uso do *façade* (FONTE: KROTH, 2000)

Com o uso do *Façade,* cada classe da aplicação que utilizar mais de um subsistema necessitará somente da instância da classe do *Façade*, a qual servirá de intermediador entre as partes, e terá acesso à implementação presente nos subsistemas, retirando a necessidade de instanciar cada um deles dentro da aplicação. Na Figura 3 (lado direito) é apresentado o padrão de projeto *Façade,* que realiza a intermediação entre cada parte do sistema.

3. A Arquitetura MVC

O MVC (*Model – View – Controller*) é definido como um padrão arquitetural que oferece uma visão de mais alto nível da estrutura do sistema. O MVC uso os padrões de projeto (*Observer*, *Composite* e *Strategy*) combinados para tratamento de efeitos globais do sistema (GAMMA *et al.*, 2006). A arquitetura MVC (Figura 4), tem como objetivo isolar a lógica do negócio da entrada e apresentação dos dados, permitindo assim a independência no desenvolvimento, testes e manutenção de cada camada do sistema (ORLANDO 2009).

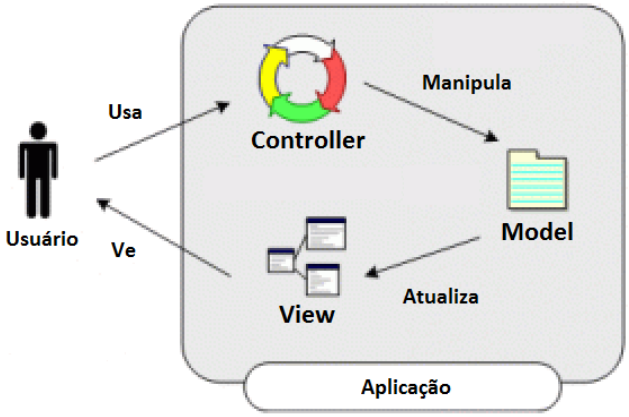

Figura 4. Modelo da arquitetura MVC (FONTE: ORLANDO, 2009)

A arquitetura é composta por três tipos de objetos que aumentam a flexibilidade e a reutilização da aplicação (GAMMA *et al.,* 2006).

- Model – implementa a lógica de negócio do domínio de dados, sendo considerado o núcleo da aplicação. Não possui conhecimento específico sobre o controlador, nem sobre a visão, sendo o próprio sistema responsável por comunicar a visão qualquer alteração que ocorra no estado do modelo. Em pequenas aplicações o modelo é muitas vezes apenas uma separação conceitual, em vez de física, fazendo com que ele assuma o papel de modelo de objetos.
- *View* – serve de interface com o usuário, disponibilizando dados produzidos a partir do modelo para a criação, edição e visualização. Pode ser representada através de vários tipos de componentes, sendo geralmente, páginas que geram conteúdo dinamicamente. Estas páginas podem incluir conteúdos HTML, arquivos PDF ou XML, imagens, gráficos, entre outros.
- *Controller* – responsável por lidar com a interação do usuário, trabalhando com o modelo a fim de selecionar os dados visíveis na interface. Este componente interliga a visualização e o modelo, controlando todas as requisições realizadas na visualização e as consultas ao modelo.

Segundo Orlando (2009), o MVC proporciona uma clara separação de interesses, o que possibilita o desenvolvimento paralelo das visões e dos controladores, o aumento da produtividade da equipe de desenvolvimento e também da manutenção.

4. Manutenção de software

A manutenção de software é consequência de procedimentos de gestão de qualidade realizadas na fase de desenvolvimento do sistema, sendo considerada qualquer mudança após o software estar em operação (SANTOS, 2007). Segundo as leis de Lehman e Belady (1985) essas mudanças estão ligadas a fatores como a mudança contínua, aumento da complexidade, evolução de software de porte, estado saturado e mudanças incrementais.

4.1. Manutenibilidade de Software

A manutenibilidade é caracterizada como um conjunto de atributos que evidencia o esforço necessário para realizar modificações específicas no produto de software, sendo ela dividida em cinco sub-características (LINO, 2011):

- *Analisabilidade*: capacidade do produto de software permitir o diagnóstico de deficiências, causas de falhas, ou a identificação de partes a serem modificadas.
- *Modificabilidade*: capacidade do produto de software permitir que uma modificação específica seja implementada.
- *Estabilidade*: capacidade do produto de software evitar efeitos inesperados decorrentes de modificações realizadas no software.
- *Testabilidade*: capacidade do produto de software ser validado quando modificado.
- *Conformidade*: capacidade do produto de software estar de acordo com normas ou convenções relacionadas à manutenibilidade.

Segundo Santos (2007) para que um produto de software seja manutenível, as sub-características da manutenibilidade devem ser incorporadas desde o inicio do seu processo de desenvolvimento. Mas, muitos produtos de software são desenvolvidos sem a preocupação com o seu tempo de vida e consequentemente não são projetados para facilitar sua manutenção, acarretando custos temporais e financeiros na manutenção. Isso, muitas vezes, tem maior ônus que desenvolver um novo sistema (PRESSMAN 2006).

O uso e a aplicação de padrões de projeto pode ter grande influência na manutenibilidade, reusabilidade e extensibilidade do software, ajudando na melhoria da legibilidade do código, o que facilita o entendimento dos seus mantenedores, além de ser considerada como uma parte da documentação do sistema (LINO, 2011).

5. Estudo de caso

Essa seção apresenta o sistema desenvolvido, com e sem a adoção de padrões de projetos. Como resultados, são evidenciados os ganhos que a equipe pode obter, ao adotar os padrões de projetos.

5.1. Visão geral do sistema

O estudo de caso aborda um sistema de venda de passagem de ônibus, cujo objetivo consiste em permitir que o usuário, visualize, através de uma aplicação web, as viagens disponibilizadas pela empresa, bem como fazer a aquisição de passagens de forma ágil e simplificada. Para o gerenciamento da aplicação, o usuário administrador será o único que terá acesso aos cadastros que compõem uma viagem, tendo ele a possibilidade de cadastrar novos ônibus e rotas. O usuário passageiro que acessar o sistema, terá acesso a todas as viagens que a empresa disponibiliza, sendo possível visualizar os seus dados mais

relevantes. Entretanto, somente passageiros cadastrados no sistema poderão realizar a compra de uma passagem. O usuário, após logar-se no sistema verá um histórico com as passagens já adquiridas, tendo a possibilidade de efetuar o cancelamento de uma compra, com até 24 horas de antecedência da partida do ônibus.

5.2. Modelagem do Sistema

A Figura 5 (lado esquerdo) ilustra o diagrama de caso de uso da aplicação desenvolvida, que representa, a história estilizada sobre como um usuário final interage com o sistema.

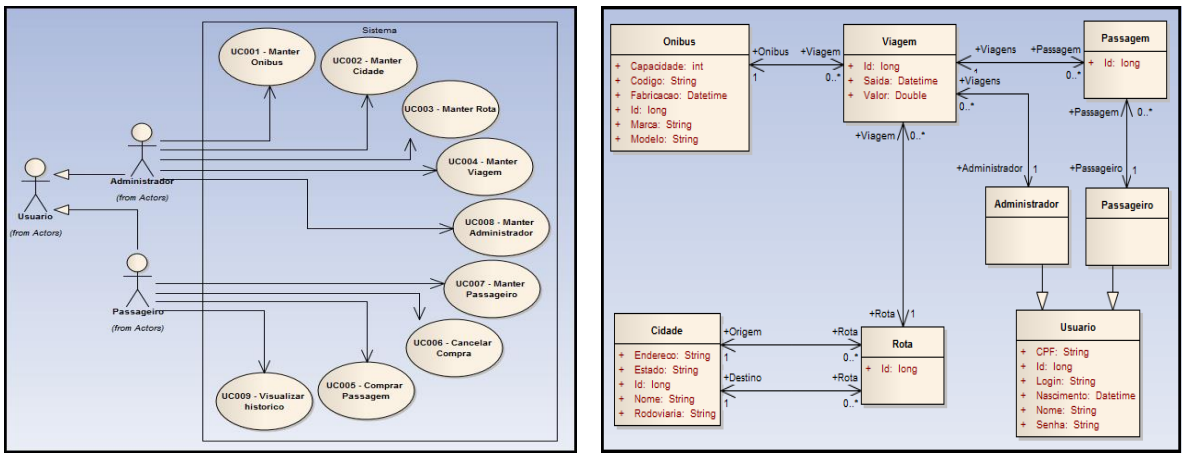

Figura 5. Diagramas de caso de uso e de classes

Para desenvolver o sistema também foi criado um diagrama de classes. Esse diagrama (lado direito da Figura 5) ilustra as entidades do mundo real, com seus atributos, relacionamentos e métodos.

5.3. Implementação do Sistema

Com objetivo de demostrar as melhorias do código fonte ao adotar os padrões de projetos, foram desenvolvidos dois sistemas, sem e com a adoção de padrões de projeto. Para o desenvolvimento foram usados os *frameworks* MVC e *Entity Framework*. A adoção dos padrões de projeto permite uma melhor separação de responsabilidade dentro do sistema, possibilitando a modularidade do mesmo. A modularidade permite que cada parte do projeto seja aproveitada no desenvolvimento de outro. Um exemplo são os objetos modelos que contém a definição de suas entidades e regras de negócio. O Quadro 1 mostra a entidade Rota com a definição de seus atributos e a implementação de uma regra de negócio.

Quadro 1. Entidade rota, presente no projeto *model*

```
1  public class Rota : EntidadeBase{
2      public virtual Cidade Origem { get; set; }
3      public Int64 OrigemId { get; set; }
4      public virtual Cidade Destino { get; set; }
5      public Int64 DestinoId { get; set; }
6      public void Validar(){
7          if (Origem == Destino)
8              throw new ArgumentException
9  ("Cidade de Origem e Destino devem ser diferentes", "Erro");
10     }
11 }
```

As validações do sistema serão chamadas sempre a partir do serviço da entidade (Quadro 2), que além desta função, caso necessário, irá aplicar regras que envolvam a sua

construção. Caso todo o processo seja efetuado corretamente é chamado o repositório do mesmo (Quadro 3) para aplicação das alterações no banco de dados.

Quadro 2. Método criar, presente no serviço rota

```
1  public Rota Criar(Rota rota){
2      rota.Validar();
3      return RotaRepositorio.Criar(rota);
4  }
```

Quadro 3. Método criar, presente no repositório rota

```
1  public Rota Criar(Rota rota){
2      ArticleContext db = new ArticleContext();
3      db.Rotas.Add(rota);
4      db.SaveChanges();
5      db.Dispose();
6      return rota;
7  }
```

A partir destas implementações, obtém-se um sistema organizado, tendo cada um dos projetos, papéis específicos na aplicação. E, tendo entre outros benefícios, uma melhor separação do código, permitindo que diferentes equipes trabalhem no mesmo projeto, mas com focos distintos. Enquanto uma equipe trabalha na parte visual, a partir dos controladores e visões, outra equipe poderá trabalhar nas regras de negócios do sistema e na conexão com a base de dados. Sem a separação de papéis, o sistema acaba tendo todas estas funcionalidades presentes no controlador, como mostra o Quadro 4. Não havendo a separação clara dos papéis, muitas vezes, pode haver junções de regras de negócios com implementações específicas sendo encaminhadas às *views* do sistema.

Quadro 4. Método *create,* presente no controlador rota

```
1  public ActionResult Create(Rota rota){
2      if (ModelState.IsValid){
3          rota.Validar();
4          db.Rotas.Add(rota);
5          db.SaveChanges();
6          return RedirectToAction("Index");
7      }
8      return View(rota);
9  }
```

A partir das novas implementações, o controlador terá apenas a função de buscar e encaminhar os dados a *view,* bem como busca-los na tela e encaminhá-los ao serviço. Entretanto, é possível se deparar com situações em que um controlador necessita buscar informações em mais de um serviço. Isto irá acarretar em um custo a mais no momento da instanciação deste controlador, além de requerer que o programador tenha conhecimento de todos os serviços. Para evitar este problema, foi adotado o padrão de projeto *Façade*.

Quadro 5. Método deletar viagem, presente no *façade*

```
1  public void Delete(Viagem viagem){
2    IEnumerable<Passagem> passagens =
3      BuscarTodasPassagens().Where(x => x.Viagem == viagem);
4
5    if(passagens.Any())
6      throw new ArgumentException
7       ("Viagem não pode ser excluída, pois já possui
8         passagens vendidas.", "Erro")
9
10   ViagemServico.Delete(viagem);
11 }
```

No *Façade* é adotada uma superclasse que contém uma referência a todos os métodos implementados nos serviços. Nesta superclasse, além das chamadas aos serviços,

pode-se ter a implementação de regras que envolvam mais de um serviço, visto que cada serviço é responsável unicamente por sua entidade. Um exemplo é a validação realizada antes da exclusão de uma viagem (Quadro 5), a qual verifica as passagens vendidas.

Com a implementação do *Façade* é possível resolver apenas o problema da instanciação de vários serviços em um único controlador, ainda sendo necessária a instanciação do serviço cada vez que for utilizá-lo em um controlador. Para resolver este problema, foram utilizados os padrões de projeto *Factory Method* e *Singleton*.

O padrão *Factory* (Quadro 6) é responsável por retornar a instância do *Façade*. Este padrão foi adotado com o intuito de no futuro, com a expansão do sistema, caso seja necessário à chamada de outro *Façade* ou serviço presente em outro módulo e que não trabalhe com o *Façade* implementado, seja possível acrescentá-lo ao método abaixo, tendo sua instancia facilitada em todo o sistema, além de se manter um padrão na instanciação de interfaces da aplicação. Em adição, foram utilizados os conceitos do padrão *Singleton* para que somente uma vez seja instanciado o *Façade*.

Quadro 6. Método deletar viagem, presente no *façade*

```
1  public class ConcreteCreator : Creator{
2      private ICoreFacade _coreFacade;
3      public override IServico GetInstance(EnumServico enumServico){
4          switch (enumServico){
5              case EnumServico.CoreFacade:
6                  if (_coreFacade == null)
7                      _coreFacade = new CoreFacade();
8                  return _coreFacade;
9              default:
10                 throw new ArgumentException("Serviço não
11                 encontrado.", "Erro");
12         }
13     }
14 }
```

O padrão *Singleton* (Quadro 7) será responsável por assegurar que cada serviço seja instanciado somente uma vez para todo o sistema, evitando que este trabalho seja repetido a todo instante, garantindo que o *Façade* tenha uma única instância acessível de maneira global no sistema.

Quadro 7. Classe *singleton* responsável por receber e gerenciar as instâncias

```
1  public class ApplicationSingleton{
2      private static ApplicationSingleton _instance;
3      public static ApplicationSingleton Instance{
4          get{
5              if (_instance == null)
6                  _instance = new ApplicationSingleton();
7              return _instance;
8          }
9      }
10
11     private Creator _factory;
12     public Creator Factory{
13         get{
14             if (_factory == null)
15                 _factory = new ConcreteCreator();
16             return _factory;
17         }
18     }
19 }
```

Com todas estas implementações realizadas, o método *criar*, presente no controlador passa a ficar extremamente simples do que o apresentado no Quadro 4, tendo a partir das novas implementações somente a responsabilidade de encaminhar as

informações vindas da *view* para o método presente no *Facade* (Quadro 8), o qual tem sua instanciação realizada pelos padrões *Singleton*, linha 5.

Quadro 8. Método *create* no controlador após aplicar os padrões de projetos

```
1  public ActionResult Create(Rota rota){
2      if (ModelState.IsValid){
3          rota.Validar();
4          db.Rotas.Add(rota);
5          db.SaveChanges();
6          return RedirectToAction("Index");
7      }return View(rota);
8  }
```

6. Conclusão

O presente trabalho teve como objetivo o estudo e a aplicação de padrões de projetos e da arquitetura MVC com foco na melhoria de qualidade e manutenibilidade do código fonte da aplicação. O uso destes padrões na implementação de sistemas acaba se tornando um diferencial na qualidade do produto final, uma vez que sua correta adoção e aplicação, em muitos casos, pode trazer outros benefícios, além da melhoria do código. Esses benefícios se estendem na melhoria do controle da memória utilizada e até mesmo no ganho de desempenho. E principalmente, permite que membros de uma mesma equipe possam trabalhar paralelamente no mesmo projeto, porém em atividades distintas, de forma organizada e ágil. Este é um dos grandes diferenciais que encontramos entre o desenvolvimento dos dois sistemas do estudo de caso. Vimos que no sistema sem padrão de projetos, toda a lógica é centralizada, o que pode dificultar a compreensão em alguns casos, diferentemente do exemplo com a adoção dos padrões, onde temos cenários bem definidos e com responsabilidades claras.

Com base nos sistemas desenvolvidos para este trabalho, também notamos algumas melhorias a serem adotadas continuamente pelas equipes de desenvolvimento de sistemas.

A primeira melhoria consiste na organização do código fonte, alcançada através da adoção do padrão *Façade,* pois, ele é a única interface para aplicação das alterações realizadas em tela na base de dados. A adoção deste padrão, retira qualquer interação realizada pelo controlador da aplicação sobre a base de dados, deixando esta responsabilidade para os serviços implementados e que são acessíveis via *Façade*. Desta forma, o controlador acaba tendo seu trabalho simplificado, sendo sua responsabilidade somente a de solicitar informações para serem mostradas em tela ou encaminhá-las aos métodos do *Façade* para que se apliquem as regras e validações necessárias.

Outra melhoria que percebemos e que tem grande impacto no desenvolvimento do sistema é quanto à utilização dos padrões *Singleton* e *Factory Method* na instância do *Façade*. Enquanto, o *Singleton* nos assegura que teremos somente uma instância do *Façade* durante a utilização do sistema, evitando o desgaste de inúmeras instanciações, o padrão *Factory Method* nos permite realizar a instância das interfaces de forma clara e centralizada, permitindo que futuras expansões do sistema, como a implementação de um novo *Façade*, sejam realizadas facilmente, melhorando consideravelmente a manutenabilidade do sistema.

Com a adoção destas práticas associadas à arquitetura MVC, a qual já proporciona uma clara separação de papéis, tem-se uma melhor divisão de responsabilidades, o que proporciona diversos benefícios que vão desde a identificação facilitada de erros, até a implementação de novas ações. Estas características tornam o sistema mais manutenível, pois, a partir da identificação de alguma inconformidade, pode-se encontrar o local desta

falha facilmente, seja ela regra de negócio, acesso a dados ou simplesmente visual, o que facilita também novas implementações, visto que se conhece o que implementar e onde implementar. Além do mais, estas características possibilitam que a equipe de desenvolvimento se divida, tendo cada grupo a responsabilidade de manter uma parte do sistema, se especializando nas ferramentas que as compõem. Aliado a todas estas características, a adoção dos padrões de projeto impacta diretamente sobre a qualidade do código fonte, uma vez que se tem regras específicas para a implementação de cada ação; o *Façade* para centralização dos acessos aos serviços, o *Singleton* para gerenciar que uma interface seja instanciada uma única vez e o *Factory Method* para a padronização de como estas interfaces serão instanciadas.

A adoção destas práticas faz com que tenhamos uma equipe de desenvolvimento em sintonia, onde todos sabem como o sistema funciona e deve se comportar, facilitando também na integração de novos colaboradores ao projeto, uma vez que teremos um código legível, o que facilita o entendimento de quem o mantem, além de ser considerada uma parte da documentação do projeto. Mas, a sua adoção deve ser estudada previamente pela equipe de desenvolvimento, pois em alguns casos, dependendo do contexto do sistema a ser implementado, o custo de sua aplicação será alto. Entretanto, a sua correta adoção, aliadas a outras técnicas tendem a ser benéficas ao projeto sendo seu uso sempre recomendado, desde que seja de forma correta e no contexto certo.

Referencias

GAMMA, Erich *et al*. Padrões de Projetos: Soluções Reutilizáveis de Software Orientado a Objetos. São Paulo: Bookman, 2006. 364 p. 0-201-63361-2.

KROTH, Eduardo. Arquitetura de software para reuso de componentes. 2000. 69 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação, Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2000.

LEHMAN, M. M.; BELADY, L. Program Evolution: Processes of Software Change. Academic Press, 1985, 538p.

LINO, Carlos Eduardo. Reestruturação de software com adoção de padrões de projeto para a melhoria da manutenibilidade. 2011. 68 f. Monografia (Bacharelado) - Curso de Sistemas de Informação, Departamento de Ciência da Computação, Universidade Federal de Lavras, Lavras, 2011.

ORLANDO, Alex Fernando. Uma infraestrutura computacional para o gerenciamento de programas de ensino individualizado. 2009. 181 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, Departamento de Computação, Universidade Federal de São Carlos, São Carlos, 2009.

PRESSMAN, Roger S. Engenharia de Software. São Paulo: Mcgraw-hill Interamericana, 2006. 752p.

SANTOS, Rodrigo Pereira. Critérios de manutenibilidade para construção e avaliação de produtos de software orientados a aspectos. 2007. 103 f. Monografia (Graduação) – Departamento de Ciência da Computação, Departamento de Computação, Universidade Federal de Lavras, Lavras, 2007.

SHALLOWAY, Alan; TROTT, James R. Explicando padrões de projeto: uma nova perspectiva em projeto orientado a objeto. Porto Alegre: Bookman, 2004.

SILVA, Patrícia F.; PENHA, José A. M.; ALVES, Gabriel M. Estudo do padrão de projeto observer no desenvolvimento de softwares utilizando a arquitetura MVC. In: mostra nacional de iniciação científica e tecnológica interdisciplinar, 3., 2009, Camboriú.

SOUZA, Marcio B. Do DAO ao Facade. Java Magazine, Natal, vol. 97, no. 1, nov. 2011.

VALENTIM, Ricardo A. M.; NETO, Plácido A. S. O impacto da utilização de design patterns nas métricas e estimativas de projetos de software: a utilização de padrões tem alguma influência nas estimativas?. Revista da Farn, Natal, vol. 4, no. 1, p.63-74, dez. 2005.