Análise da utilização de padrões de projeto para a melhoria da qualidade no desenvolvimento de sistemas orientados a objeto

Ademir Constantino Filho
Prof. Djone Kochanski
Faculdade Metropolitana de Blumenau – FAMEBLU
Bacharelado em Sistemas de Informação – Trabalho de Graduação II
20/06/2010

RESUMO

Como atividade de alta demanda nos últimos anos, o desenvolvimento de software utilizando tecnologias orientadas a objeto tem sido popularmente adotado pelos benefícios de reuso, facilidade na manutenção e qualidade na arquitetura. O presente artigo faz uma análise da utilização de alguns padrões de projeto na busca da melhoria da qualidade de software.

Palavras-chave: Orientação a Objetos; Padrões de Projeto; Qualidade de Software.

1 INTRODUÇÃO

A produção de software cresce a cada ano na medida em que as grandes, pequenas e médias empresas decidem automatizar seus processos buscando competitividade, redução de custos e melhoria dos processos de negócio.

Apesar de um software desenvolvido inicialmente atender os requisitos do negócio, na grande maioria dos casos as alterações são inevitáveis, considerando que um processo dentro da empresa pode mudar devido por uma nova lei, um novo processo de qualidade ou uma nova funcionalidade.

Pressman (2006) define que: manutenção de software custa mais esforço que qualquer outra atividade de engenharia de software. Manutenção de software é a atividade que um programa pode ser corrigido se um erro é encontrado, adaptado caso haja mudança no ambiente, ou aprimorado se o cliente desejar uma mudança nos requisitos.

Se considerarmos os processos de desenvolvimento de software, onde as metodologias de desenvolvimento são aplicadas buscando a excelência no processo através da Engenharia de Software, Pressman (2006) afirma que não há uma forma para estimar a fase de manutenção de um software, porém pode ser utilizada uma métrica simples para analisar a mudança solicitada, projetar uma situação apropriada, programar a mudança, testá-la e distribuí-la para todos os usuários. Esta fase normalmente toma a grande maioria do tempo do projeto, pois as alterações e adaptações são constantemente solicitadas pelos clientes. A qualidade da arquitetura em nível de desenvolvimento para que além de um processo de desenvolvimento definido ajude na manutenção, a qualidade do software seja atingida através soluções já documentadas com o intuito de reduzir o código duplicado e para que as alterações geradas sob demanda sejam executadas mantendo a qualidade do projeto e ainda permitindo que futuras alterações possam ser feitas sem muitos esforços.

Segundo Fowler (2004), um sistema mal projetado normalmente precisa de mais código para fazer as mesmas coisas muitas vezes porque o mesmo código foi duplicado em diversos lugares diferentes. Quanto mais difícil é visualizar o projeto a partir do código, mais difícil será preservá-lo e mais rapidamente ele se desestruturará. A eliminação de código duplicado é um aspecto importante na melhora do projeto, sendo que reduzindo a quantidade de código faz uma grande diferença sobre a manutenção desse projeto.

Para Gang of Four (1995), uma coisa que os melhores projetistas sabem que não devem fazer é resolver cada problema a partir de princípios elementares ou do zero, em vez disso eles reutilizam soluções que funcionaram no passado. Os padrões de projetos introduzidos pela Gang of Four são amplamente conhecidos por projetistas, arquitetos e experientes desenvolvedores que os aplicam na resolução de problemas de arquitetura de software fazendo com que o software possa ser alterado para atender problemas e requisitos futuros.

Os capítulos seguintes irão abordar os padrões de projeto introduzidos pela equipe de cientistas da computação que foram denominados de Gang dos Quatro (Gang of Four), por ser formada pelos integrantes Erich Gamma, Richard Heml, Ralph Johnson, John Vlissides, todos cientistas da computação com o foco na melhoria na qualidade do desenvolvimento de software. Soluções atuais como Arquitetura orientada ao domínio do cliente também será abordada.

2 PADRÕES DE PROJETOS

O conceito de padrão de projeto foi criado pelo arquiteto Cristopher Alexander em 1977, onde define (Alexander, 1977 apud Gang of Four 1995, p. 19) “cada padrão descreve um problema no nosso ambiente e o cerne de sua solução, de tal forma que você possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira”. Sendo posteriormente apresentado pela primeira vez na computação no evento OOPSLA (1987) pelos programadores Kent Beck e Ward Cunningham, onde apresentaram cinco pequenos padrões de projeto para a criação de janelas utilizando a linguagem orientada a objetos Smalltalk.

Pimentel (2010), mostra que em geral os padrões de projeto podem ser classificados em três diferentes tipos:

- **Padrões de criação:** abstraem o processo de criação de objetos a partir da instanciação de classes.

- **Padrões estruturais:** tratam da forma como classes e objetos estão organizados para a formação de estruturas maiores.

- **Padrões comportamentais:** preocupam-se com algoritmos e a atribuição de responsabilidade.

Os padrões de projetos passaram a ser amplamente conhecidos mundialmente com a popularização da linguagem C++, Java e posteriormente as linguagens da tecnologia ponto net e Ruby. Normalmente os padrões de projeto são aplicados na busca pela melhoria na qualidade de software, reuso, redução de custos e facilidade na manutenção do código.

Partindo do princípio que padrões propõem soluções para problemas de design, a seguir serão apresentados três problemas de design em projetos orientados a objetos e posteriormente suas respectivas soluções.

3 PROBLEMAS COMUNS DE ARQUITETURA

1. **Serviço para conexão com banco de dados:**

Considerando um caso de um objeto ou unidade que trata de conexão com o banco de dados, normalmente um método se faz responsável por iniciar a conexão com o SGDB, permitindo então que uma transação, uma consulta ou uma chamada de função seja executada. Uma solução seria acessar o SGDB iniciando uma conexão para cada operação, fazendo com que para cada consulta do banco de dados uma conexão se inicia, a consulta é executada, os dados são retornados, a conexão é fechada e em seguida os dados são exibidos no sistema.

Considere que esta conexão não precisa ser fechada até que o programa seja finalizado, pois se o programa dispusesse de uma conexão aberta, além do sistema ter um melhor desempenho diminuindo o overhead, obtendo o máximo de throughput de rede e ainda possíveis problemas de limitação na quantidade de conexões permitidas pelo banco de dados, este problema poderia ser resolvido aplicando um padrão de projeto que permita apenas uma instância de um objeto de conexão com o banco de dados seja utilizada, fazendo com que toda vez que o programa necessite uma conexão, um objeto provê a conexão que apenas seria finalizada quando a instância do programa fosse finalizada ou quando solicitada pelo programa.

2. Arquitetura orientada ao domínio do cliente:

Uma equipe de estudantes acadêmicos, desenvolvendo um trabalho para a disciplina de orientação a objetos tiveram de criar um domínio onde Pessoa pode enviar mensagens, não especificando o tipo de mensagem. Foi definido pelo grupo que Pessoa e Mensagem seriam definidas pelas suas respectivas classes. Basicamente depois de criado o domínio, surgiu a necessidade da criação da implementação da mensagem, então o grupo criou uma classe responsável por esta operação, encapsulando a regra de negócio da pessoa na classe PessoaService.

Segue abaixo um exemplo em linguagem Java para esta funcionalidade:

```
public enum TipoPessoa {
      FISICA,
      JURIDICA;
}

public class Pessoa {

      private String nomePessoa;
      private TipoPessoa tipoPessoa;
      private List<Mensagem> listaMensagem;
```

```
      // getters e setters
}

public class Mensagem {

      private String dados;

      // getters e setters
}

public class PessoaService {

   private Pessoa pessoa;

      public PessoaService(Pessoa p) {
            // set pessoa = p
      }

      public void enviaMensagem(Mensagem mess) {
            // implementação do envio da mensagem
      }

}
```

Tabela 1. Código Fonte de um domínio anêmico

Este tipo de arquitetura é muito discutido nos dias de hoje, Fowler (2003), afirma que o sintoma básico do modelo de domínio anêmico é que a primeira impressão é que isto parece uma coisa real. Eles são objetos, muitos nomeados como substantivos no domínio e estes objetos são interligados com relacionamentos ricos e estruturas que modelos de domínio reais têm. A verdade vem quando você olha para o comportamento, e você percebe que dificilmente existe comportamento para estes objetos, fazendo deles um pouco mais que objetos com getters e setters. Normalmente estes modelos são regras de arquitetura que dizem que você não pode colocar nenhuma lógica nos objetos de domínio. Em vez disso alguns objetos chamados de Service capturam toda a lógica do domínio. Estes serviços vivem no topo do modelo de domínio e utilizam o modelo de domínio para dados.

Para Weisfeld (2004), a diferença chave entre a programação orientada a objetos e a programação procedural, é que atributos e comportamentos são contidos em um único objeto, porém no paradigma procedural os atributos e comportamentos são normalmente separados.

Portanto pode ser identificado não apenas uma necessidade de aplicar um padrão de projeto, mas uma má prática que vem ocorrendo ao longo dos anos.

3. **Necessidade da eliminação de redundância quando aplicado o princípio da abstração (programando para interface).**

É comum que o domínio de um cliente possa mudar. Considere um caso onde uma empresa transportadora de veículos com carga deseja rastrear seus veículos em viajem. Neste sistema existe um escalonador de processos batch que são executados de dois em 2 minutos.

Todo processo é chamado de alerta, portanto quando um veículo está fora de rota, o processo é identificado pelo alerta chamado AlertaForadeRota, na situação de parada indevida o alerta é o AlertaParadaIndevida, podendo existir “n” novos tipos de alertas.

Os projetistas do sistema definiram então o seguinte design das classes:

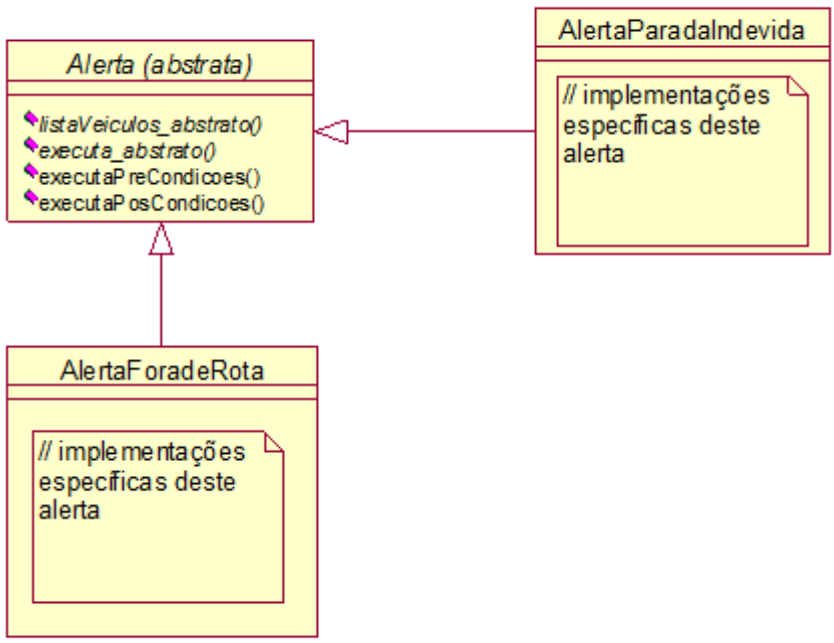

Figura 1. Diagrama UML para representação do domínio de Alertas

Kheng-Khoon Khor (1995) define que abstração permite ao desenvolvedor separar a implementação do objeto de seu comportamento. Esta separação cria uma "caixa-preta" onde o usuário é isolado das alterações na implementação. Como a interface sempre continua a mesma, qualquer mudança na implementação interna é transparente ao usuário.

Considere a seguinte analogia para entendimento do parágrafo anterior: Imagine duas classes, Usuário, Computador(Classe abstrata). Quando o usuário trabalha com um computador, este computador pode ser abstrato, pois se trata de uma generalização, mas cada tipo específico computador estende a classe abstrata Computador, por exemplo: ComputadorLaptop ou ComputadorDesktop. Neste caso, um Usuário que possui uma ligação com o Computador, ou seja: Usuário tem um (has one) Computador, abstrato, ou seja, ele sabe trabalhar com todos os tipos de computadores, pois ele sabe que todos os computadores possuem o botão ligar, possui um teclado e um monitor, pois estas características básicas são definidas no contrato, onde estão todas as características básicas de um Computador, e esta é a caixa preta referida no parágrafo anterior. A Pessoa só precisa saber que ele sabe trabalhar com um computador, isto faz com que o Usuário trabalhe com diferentes computadores, sendo estes, alterados mesmo em momento de execução, mas o grande benefício aqui é que novos computadores poderão ser adicionados neste sistema e o usuário não precisará sofrer sequer uma alteração para trabalhar com os novos computadores, pois todos compartilham as mesmas características básicas.

O ponto aqui é que na condição de diferentes tipos de alertas poderem ser criados, hoje em uma classe que controla o escalonamento dos processos, considere que no futuro existirá mais uma ou muitas classes que farão o uso dos objetos. A questão é que quando se trabalha com objetos abstratos, na grande maioria das vezes é necessária a instanciação dos objetos concretos a partir de condições do sistema. Considere o seguinte código:

```
public Map<Veiculo, Alerta>
     listarAlertasPorVeiculo(Collection<Veiculo> listaVeiculos) {

      Map<Veiculo,  Alerta>  alertasExecucao  =  new  HashMap<Veiculo,
Alerta>();
      for (Veiculo veicAtual: listaVeiculos) {

            if(veicAtual.isMonitorado()) {
                  for(ServicoRastreamento servico:
                              veicAtual.getServicosHabilitados()) {

                  switch(servico) {

                        case FORA_DE_ROTA:

                              Alerta alertaServicoFR =
                                          new AlertaForadeRota();
                              alertasExecucao.put(veicAtual,
                                                alertaServicoFR);
                              break;

                        case PARADA_INDEVIDA:
```

```
                              Alerta alertaServicoPI =
                                          new AlertaParadaIndevida();
                              alertasExecucao.put(veicAtual,
                                                alertaServicoPI);
                              break;
                              // futuras implementações

                  default:
                        break;

             } // fim do switch
      } // fim do foreach de servicos habilitados

      } // fim do foreach de veiculos
}
```

Tabela 2. Código Fonte utilizando programação para Interface

O método acima além de criar a lista de Alertas necessários para os veículos de acordo com seus serviços habilitados retorna uma lista com os alertas que posteriormente deverão ser executados. Considere a situação em que diversas partes do sistema fazem o uso dos alertas, e isto, apesar de no início ser pouco provável deve ser considerado, pois um requisito futuro pode fazer com que os alertas sejam utilizados em outras partes do sistema, exemplo: O cliente solicitou que quando acessar o sistema na web, ele pode disparar o alerta manualmente. O impacto aqui será que o seguinte código seria duplicado, fazendo com que o código que verifica como o alerta deve ser criado exista em mais de uma parte do sistema, impactando ainda que quando um novo alerta fosse criado, seria necessária a alteração em mais de uma parte do projeto.

4 SOLUÇÕES PROPOSTAS UTILIZANDO PADRÕES DE PROJETOS

Maioriello (2003) afirma que: desde o início, os softwares foram criados para resolver problemas específicos de um problema. Quanto mais os arquitetos de softwares tem um entendimento claro do problema e os requisitos, o sistema pode ser criado para atender as necessidades e expectativas do usuário. Uma vez que os sistemas estão em uso e na mão dos usuários, os sistemas de sucesso devem suportar e atender as mudanças dos requisitos.

Para Wroblewski (2006), os padrões de projeto descrevem soluções. Soluções que nós conhecemos podem trabalhar “positivamente” problemas específicos. As soluções são documentadas como um “padrão” (pattern) onde todos os aspectos são descritos, até mesmo detalhes de implementação caso auxiliem e sejam relevantes.

Baseado nos 4 princípios do paradigma de orientação a objetos, Maioriello (2003) afirma que: os objetivos fundamentais baseiam-se em três estratégias básicas para criar boas arquiteturas orientadas a objetos, elas são:

- Programar para a interface;
- Preferir composição a herança;
- Identificar e encapsular o que varia.

O que pode ser observado neste cenário é que os princípios e objetos da orientação a objetos e os padrões de projeto trabalham juntamente quando se trata de uma boa arquitetura. Alguns padrões de projetos podem combinar juntos ou um padrão conduzir a outro, podendo ainda ser similares ou alternativos, portanto não existe escolha certa de padrão ou solução de um problema. Algumas soluções são propostas nos itens seguintes para a exemplificação da aplicação dos padrões selecionados, embora não sejam a única ou a solução ideal para todos os problemas de arquitetura de software orientado a objetos.

1. **Singleton para acesso ao banco de dados.**

Para criar uma classe que possua a funcionalidade de prover a mesma conexão para o uso na aplicação, seria primeiramente utilizado o padrão singleton, categorizado como Padrão de Criação pela Gang of Four (1995), onde define:

É importante para algumas classes ter uma, e apenas uma instância. Por exemplo, embora possam existir muitas impressoras em um sistema, deveria haver somente um sistema de arquivos de e um gerenciador de janelas.

A aplicabilidade para o uso de um singleton neste caso é definida por Gang of Four (1995) como: Quando for preciso haver apenas uma instância de uma classe, e essa instância tiver que dar acesso aos clientes através de um ponto bem conhecido.

O gráfico abaixo representa o funcionamento do serviço de um objeto que aplica o padrão singleton:

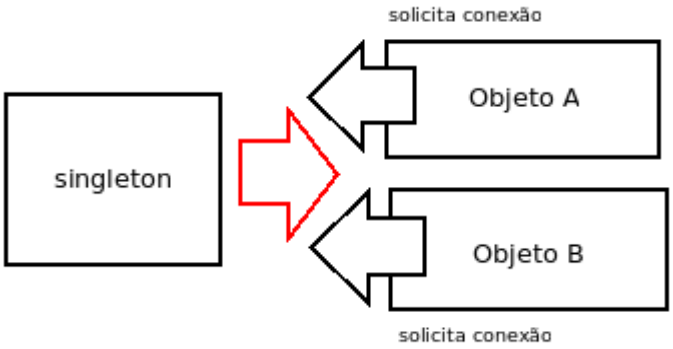

Figura 2. Diagrama para exemplificação de um serviço Singleton

Segue abaixo o código exemplo em linguagem Java:

```
public class ConnectionService {

	private ConnectionService instance;
	private Connection conexao;

	private ConnectionService() {
		this.initDB();
	}

	public static synchronized
		ConnectionService getInstance() throws Exception {

		try {
			if(instance == null) {
				instance =  new ConnectionService();

			} else {

				// assegura que sera retornada
				//com uma conexao valida

				if(instance.conexao.isClosed()) {
					initDB();
				}
			}

			return instance;

		} catch (Exception e) {
			// Registra um evento no log de erro
			throw new Exception(e);
		}
	}

	private void initDB() {
		// metodo responsavel por realizar a conexao
		// conexao = estabelecer conexao com o banco de dados
	}
```

```
}
```

3. Código Fonte com implementação de um padrão Singleton

A classe exemplificada acima ConnectionService será utilizada pelas classes que necessitarem acesso ao programa, conforme o exemplo abaixo:

```
public class PessoaDAO {

      private Connection conexao;

      public List<Pessoa> recuperaListaPessoas() {

          List<Pessoa> clRetorno = new ArrayList<PessoaFisica>();

          conexao = ConnectionService.getInstance();
          Statement stmt = conexao.createStatement();
          ResultSet rs = stmt.executeQuery("SELECT * FROM Pessoa");

          while(rs.next()) {
            Pessoa p = new PessoaFisica();
            p.setId(rs.getInt("id"));
            p.setNome(rs.getString("nome"));
            clRetorno.add(p);
      }

      return clRetorno;
}
```

Tabela 4. Código fonte da utilização do objeto que utiliza o padrão Singleton

O código acima mostra que através da utilização de um padrão que dispõe uma conexão única e centralizada quando necessária, além do benefício da facilidade da utilização, traz benefícios de desempenho de rede e do sistema.

Embora o modificador synchronized tenha sido utilizado, fazendo com que o serviço da conexão seja utilizado de forma sincronizada, evitando problemas de concorrência, observa-se que esta solução não é viável em sistemas web, pois apenas uma conexão com o banco de dados estaria disponível para uma grande quantidade de usuários. A biblioteca básica Java Database Connectivity (JDBC) 2.0, disponibiliza classes que aplicam o conceito de pool de conexões que podem ser utilizadas como soluções para sistemas concorrentes ou centralizados.

2. Arquitetura de Software baseada no domínio do negócio.

Por anos a modelagem de software e a o código caminharam junto no processo de desenvolvimento de software, apesar de na maioria das vezes não refletirem o mesmo sentindo quando gerados os artefatos finais.

Moreira (2010) define que: o domain-driven design é um conjunto de princípios que auxiliam no desenvolvimento de software voltado para o domínio, ou seja, voltado para o processo que existe no mundo real, não ignorando o fato de que este domínio será expresso através de código.

Na busca pela integração do domínio do negócio (análise e projeto do sistema) e na codificação, Eric Evans introduziu em 2004 em seu livro intitulado “Domain Driven Design – Tackling Complexity in Heart of Software” (Arquitetura Orientada ao Domínio – Combatendo a Complexibilidade no Coração do Software), onde define técnicas de engenharia de software, incluindo direcionamento para a metodologia de processo de desenvolvimento e para a arquitetura e projeto em si.

Evans (2004) afirma que: Para criar software validamente envolvido na atividade do usuário, o time de desenvolvimento de software deve trazer para si um corpo de conhecimento relacionado à atividade do usuário.

Um simples exemplo seria um projeto de software de contabilidade, onde a equipe de desenvolvimento através de entrevistas ou outras formas de comunicação devem entender e trazer este conhecimento para o dia-dia, utilizando-o também na definição do domínio do usuário.

Evans (2004) define ainda que: O modelo de domínio não é simplesmente um diagrama; isto é uma abstração rigorosamente organizada e seletiva do conhecimento. Um diagrama pode representar e comunicar um modelo como pode cuidadosamente um código fonte escrito, assim como a língua Inglesa.

Os envolvidos no projeto neste caso podem ser classificados como a equipe dos especialistas (domain-experts) no domínio (considerando o exemplo do sistema de contabilidade, os contadores) que devem se esforçar para entender a parte técnica a fim de auxiliar na geração dos artefatos, e a equipe técnica (desenvolvedores de software) que além de dominar as tecnologias do projeto devem absorver o máximo do conhecimento sobre o domínio.

O modelo nem sempre precisa ser um código fonte ou um diagrama da UML, podendo ser esboços de especificações de requisitos, anotações de reuniões, esboços de diagramas ou outra forma de representar um conhecimento.

Evans (2004), explica que os especialistas no domínio têm entendimento limitado dos jargões técnicos do desenvolvimento de software, mas eles usam jargões de suas áreas específicas – provavelmente de diversas formas. Desenvolvedores, por outro lado, podem discutir e entender em termos funcionais descritivos desprovidos dos termos de linguagem utilizados pelos especialistas na área de negócio.

O maior problema aqui é que partindo desta premissa, os desenvolvedores de software vagamente entendem os termos e jargões utilizados pelo domínio, criando suas próprias definições baseadas em entendimentos e aplicando nos artefatos do software, como nos códigos e diagramas, criando assim duas linguagens distintas entre os especialistas do domínio e os desenvolvedores de software. Estas distintas terminologias fazem com que os artefatos de software estejam desconectados da realidade do negócio do usuário.

Moreira (2010) afirma que: As duas equipes vão conseguir atingir um modelo de qualidade através de muita conversa transmissão de conhecimento, nas duas vias. Mas não é qualquer conversa que será aproveitável. Uma conversa produtiva será aquela cujo resultado seja algo integrável ao código, sem nenhum intermediário. Os termos utilizados durante as reuniões devem ser amplamente aceitos, usados e entendidos, e a linguagem que os dois times usam deve ser a mesma.

Para definir a linguagem em comum para ambas as equipes, Eric Evans definiu como conceito de linguagem ubíqua os termos que devem ser aplicados em classes, operações, atributos entre itens que compõe os artefatos de software. Esta linguagem inclui termos para discutir regras que tem sido explícitas no modelo do domínio.

No exemplo do problema anterior, onde a pessoa envia mensagens através de uma classe responsável pela operação de negócio, a classe pessoa deveria encapsular a operação de envio de mensagem, pois no negócio em questão Pessoa envia mensagem, porém para isto seria necessária a utilização de um objeto responsável por realizar o envio da mensagem, fazendo assim um efeito

cascata positivo, pois através de um dispositivo específico a mensagem seria enviada, fazendo necessária também a utilização de camadas de software.

3. Necessidade da eliminação de redundância quando aplicado o princípio da abstração.

Quando utilizada a programação para interface ou o princípio da abstração conforme Figura 1 e Tabela 2 deve ser tomada atenção às diversas mudanças que podem ocorrer no sistema, pois a interface define o comportamento sem saber que tipo de objetos está sendo utilizado e para isto existe um padrão que pode auxiliar na criação dos objetos de tipos em comum.

Conforme citado anteriormente o sistema deve estar preparado para atender necessidades futuras, ou seja, suportar os novos alertas do sistema sem que seja necessária uma grande alteração no sistema.

Freeman (2009) define que: ao codificar para uma interface, você sabe que pode se isolar de várias alterações que podem acontecer em um sistema, porque se seu código é escrito para uma interface, ele irá funcionar com qualquer classe que implemente esta interface por meio do polimorfismo. No entanto, quando você tem um código que utiliza mais classes concretas, está procurando problemas porque o código pode ter que ser alterado à medida que novas classes concretas são adicionadas. Assim e em outras palavras, seu código não será “fechado para modificação”. Para estendê-lo com novos tipos concretos, ele terá que ser reaberto.

O problema do código apresentado anteriormente é que o método listarAlertasPorVeiculo(Collection<Veiculo> listaVeiculos), retorna os alertas permitidos para um veículo específico passado como parâmetro e nestes casos, em toda a situação envolvendo a criação de alertas o código seria duplicado, tornando difícil a manutenção do projeto.

No caso dos Objetos de alertas serem criados a partir dos serviços habilitados, poderia ser criado um método utilizando o padrão fábrica de objetos abstratas, padrão de criação definido pela Gang of Four (1995) com a seguinte intenção:

Fornecer uma interface para a criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

Uma solução aqui seria criar uma classe chamada de FabricaAlerta, responsável por criar Alertas através do método criarAlerta(ServicoRastreamento servico), onde o serviço de rastreamento serve como parâmetro para a criação do alerta do caso específico, fazendo então com que todo o processo de criação estivesse centralizado neste objeto, tornando possível a simples alteração em apenas uma classe.

O código abaixo apresenta uma simples solução para o caso exemplificado em linguagem Java:

```
public class FabricaAlerta {

    public Alerta criarAlerta(ServicoRastreamento servico) {

       Alerta alertaRetorno =  Alerta.ALERTA_NULO;

        switch(servico) {

            case FORA_DE_ROTA:

                  Alerta  alertaServicoFR  =  new  AlertaForadeRota();

                  alertaRetorno = alertaServicoFR;
                  break;

            case PARADA_INDEVIDA:

                  Alerta alertaServicoPI = new AlertaParadaIndevida();

                  alertaRetorno = alertaServicoFR;
                  break;

                  // futuras implementações
            default:
                  break;
        }
        return alertaRetorno;
  }

}
```

Tabela 5. Código Fonte exemplificando o padrão Abstract Factory

5 CONCLUSÃO

O uso de padrões de projetos tornou-se obrigatório quando aplicado o paradigma da programação orientada a objetos com a utilização de bibliotecas e principalmente frameworks, que fornecem benefícios de reuso e qualidade arquitetural. Como conhecimento trivial de arquitetos,

projetistas e desenvolvedores de software, os padrões de projetos já documentados formam uma base de conhecimento que se aplicada desde o início do projeto garantem eficiência na etapa da manutenção e na qualidade na perspectiva de desenvolvimento.

Uma definição simples para os padrões de projetos é simplesmente o conhecimento de especialistas em software para a resolução de problemas no desenvolvimento de software orientado a objetos.

O uso dos padrões de projetos permitiu que frameworks de desenvolvimento reutilizáveis, ou seja, sistemas base para para outros sistemas fossem desenvolvidos na busca do re-uso permitindo maior produtividade.

Linguagens atuais como Java, Ruby e C#, altamente populares e baseadas em orientação a objetos fazem o uso de padrões e técnicas modernas, entre elas os design patterns para o de desenvolvimento de software de alta qualidade.

6 REFERÊNCIAS

BECK, K; Cunningham, W. **Using Pattern Languages for Object-Oriented Programs.** Orlando: Adele Goldberg and Chet Wisinski, 1985. Disponível em: <http://c2.com/doc/oopsla87.html> Acesso em: 25 mai. 2010.

EVANS, Eric. **Domain-driven design:** tackling complexity in the heart of software. Addison-Wesley, 2004.

FREEMAN, Eric; FREEMAN, Elisabeth. **Use a cabeça! Padrões de Projetos: design patterns.** Alta Books, 2009.

FOWLER, Martin. **Anemic Domain Model.** Chicago, 2003. Disponível em: <http://martinfowler.com/bliki/AnemicDomainModel.html> Acesso em: 25 mai. 2010.

FOWLER, Martin. **Refatoração:** aperfeiçoando o projeto de código existente. Bookman, Porto Alegre, 2004.

GAMMA, Erich et. al. **Padrões de Projeto:** soluções reutilizáveis de software orientado a objetos. Bookman, Porto Alegre, 2000.

Kheng-Khoon Khor. **IBM Smalltalk Tutorial.** North Carolina, 1995. Disponível em: <http://www.inf.ufsc.br/poo/smalltalk/ibm/> Acesso em: 25 mai. 2010.

MAIORIELLO, James. **Applying Design Patterns to Solve Design Issues.** Foster City, 2003. Disponível em: <http://www.developer.com/design/article.php/1577941/Applying-Design-Patterns-to-Solve-Design-Issues.htm> Acesso em: 27 mai. 2010.

MOREIRA, G; PRADO, G. «**Domain-Driven Design: Além dos Patterns**», Revista Mundo Java, 01.05.2010, p. 18.

PIMENTEL, A. R. **Modelos, Especificações, Modelos de Projeto.** Curitiba, 2010. Disponível em: <http://www.inf.ufpr.br/andrey/ci221/SOFTua11.pdf> Acesso em: 27 mai. 2010.

PRESSMAN, R. S. **Software Engineering:** a practitioner's approach. 6a Edição. McGraw-Hill Science, 2004.

WEISFIELD, M. A. **The object-oriented thought process.** 2a Edição. Sams Publishing, 2004.

WROBLESKI, Luke. **What Do We Mean by "Design Patterns".** San Francisco, 2006. Disponível em: <http://www.lukew.com/ff/entry.asp?348> Acesso em: 27 mai. 2010.