PATI-MVC: Padrões MVC para Sistemas de Informação

Gabriela T. De Souza*
gabi@atlantico.com.br
Instituto Atlântico

Carlo Giovano S. Pires*
cgiovano@atlantico.com.br
Instituto Atlântico

Márcio de Oliveira Barros
marcio@cos.ufrj.br
Universidade Federal do Rio de Janeiro

Resumo

Sistemas de informação são baseados em manutenção e consulta de dados. Usualmente, estes sistemas têm funcionalidades simples, mas estas funcionalidades se repetem várias vezes. O objetivo dos padrões PATI-MVC é apresentar soluções para o projeto de funcionalidades de manutenção de dados em sistemas de informação.

Abstract

Information systems are based on registering, searching, and maintaining data. Usually, they have simple functionality, but these functions are repeated several times within an information system. The aim of PATI-MVC patterns is to present design solutions for data maintaining in information systems.

1. Introdução

Em sistemas de informação, as funcionalidades são muito concentradas em cenários de uso baseados em manutenção e consulta de dados, gerando vários problemas recorrentes durante o desenvolvimento do sistema. Esses problemas são comumente relacionados a aspectos de apresentação, interação e tratamento de eventos de usuário; separação entre aspectos de apresentação, regras de negócio e persistência de dados. Neste artigo serão apresentados os padrões CRUD-MVC e Registro com Busca que são baseados no modelo CRUD e no padrão de arquitetura MVC (Modelo-Visão-Controle) [3] e tem o objetivo de fornecer soluções para problemas recorrentes em sistemas de informação.

O modelo CRUD (*Create, Retrieve, Update, Delete*) é aplicado em cenários de consulta e manutenção de entidades em sistemas de informação. Esses cenários são

* Este trabalho foi suportado pelo Instituto Atlântico (*www.atlantico.com.br*).

tipicamente construídos com base em operações de criação (ou inclusão) de novos dados em um repositório, localização (ou consulta de dados), atualização e exclusão dos dados. Os dados mantidos pelas operações CRUD estão relacionados a uma ou mais entidades de negócio. Uma forma bastante comum para armazenar as entidades é através de tabelas em bancos de dados relacionais.

O padrão de arquitetura MVC determina a separação de responsabilidades de uma aplicação em componentes de modelo, visão e controle. O modelo é formado por entidades de negócio que representam os dados do sistema. Componentes de visão (telas gráficas ou páginas HTML, por exemplo) têm o objetivo de apresentar as informações pertencentes ao modelo e capturar eventos de usuário. Componentes de controle tratam os eventos capturados, notificam e coordenam componentes de modelo. Os componentes de controle fazem a ponte entre componentes de visão e do modelo.

Os padrões CRUD-MVC e Registro com Busca participam de uma família de padrões dedicada a construção de sistemas de informação. O padrão CRUD-MVC apresenta os relacionamentos e interações entre classes do modelo, visão e controle para realização de operações CRUD e suporte para outras operações de negócio. O padrão Registro com Busca concentra-se nas classes da visão, indicando padrões de interação com usuário de acordo com a complexidade do objeto e cenário de negócio.

2. Padrão CRUD-MVC

2.1 Contexto

Sistemas de informação requerem funcionalidades de negócio implementadas através de interface humano-computador (IHC), componentes para tratamento de regras de negócio e acesso a dados para operações CRUD e operações de negócio.

2.2 Problema

Como tratar funcionalidades recorrentes de criação, consulta, atualização e exclusão de dados em sistemas de informação considerando aspectos de apresentação e tratamento de eventos, regras de negócio e persistência?

2.3 Forças

- Deve fornecer baixo acoplamento, facilitando a troca de mecanismos de interface e acesso a dados
- Deve permitir reuso de estrutura e comportamentos de componentes de IHC, e tratamento de eventos CRUD

2.4 Solução

A idéia básica da solução é criar uma camada de classes abstratas que capturam a estrutura, comportamento e colaborações genéricas entre modelo, visão e controle e criar várias implementações (para cada cenário de uso CRUD) herdando as características da camada abstrata e definindo as características concretas da aplicação. O projetista deve utilizar o padrão definindo, as características gerais de interface, negócio e persistência na camada abstrata.

As classes abstratas de visão determinam operações para tratamento de eventos CRUD com enfoque em questões de interface com usuário, por exemplo, a operação *tratarExclusao* pode sempre exibir uma interface de diálogo para confirmar a operações antes de acionar o controlador. Outros eventos de interface com usuário e outras características que possam ser reutilizadas também são de responsabilidade das classes abstratas de visão. As classes abstratas de controle permitem a fabricação de entidades, tratamento de eventos CRUD e também suportam outras operações genéricas que o projetista possa definir. O tratamento de eventos CRUD nas classes de controle é voltado para o fluxo de negócio, ao contrário do tratamento de eventos CRUD nas classes de visão. A classe abstrata de entidade permite definir as operações CRUD propriamente ditas ou outras operações genéricas de entidades. As classes concretas implementam as operações definidas na camada abstrata e complementam comportamentos e características específicas de negócio.

2.5 Estrutura

O padrão apresenta uma camada de classes abstratas para visão, modelo e controle. As classes abstratas utilizam métodos de fabricação definidos de acordo com o padrão *Factory Method* [2], e assim, permitem que as subclasses concretas determinem as implementações das classes abstratas. Essa estrutura permite que classes do nível abstrato possam implementar as interações entre visão, modelo e controlador e as operações CRUD que são reutilizadas por todas as classes concretas da aplicação. Os relacionamentos do padrão MVC existem no nível das classes abstratas (para tratamento de eventos CRUD) e no nível das classes concretas (para tratamento de eventos de negócio específicos).

A classe de visão notifica *ControladorCRUD* sobre eventos de criação, alteração, exclusão e consulta de entidades de negócio. Por sua vez, *ControladorCRUD* é uma classe de controle que define mecanismos para a criação da entidade de negócio e operações reutilizáveis para o tratamento dos eventos CRUD sobre uma interface genérica para uma entidade de negócio. *Entidade* é uma classe de modelo que define a interface genérica de entidade de negócio, com contratos para as operações CRUD.

Para cada uma das classes genéricas, podem existir várias implementações concretas de acordo com as diversas entidades de negócio do sistema. As implementações concretas especializam as classes genéricas, definindo as características e comportamentos concretos de IHC (*ComponenteIHCConcreto*), fabricando controladores concretos (*ControladorConcreto*) que criam entidades concretas (*EntidadeConcreta*) e implementam tratamentos de outros eventos de negócio (que não operações CRUD). As entidades concretas definem os atributos (informação), implementam as operações CRUD e outras operações de negócio. A Figura 1 apresenta o diagrama de classes do padrão e seus componentes.

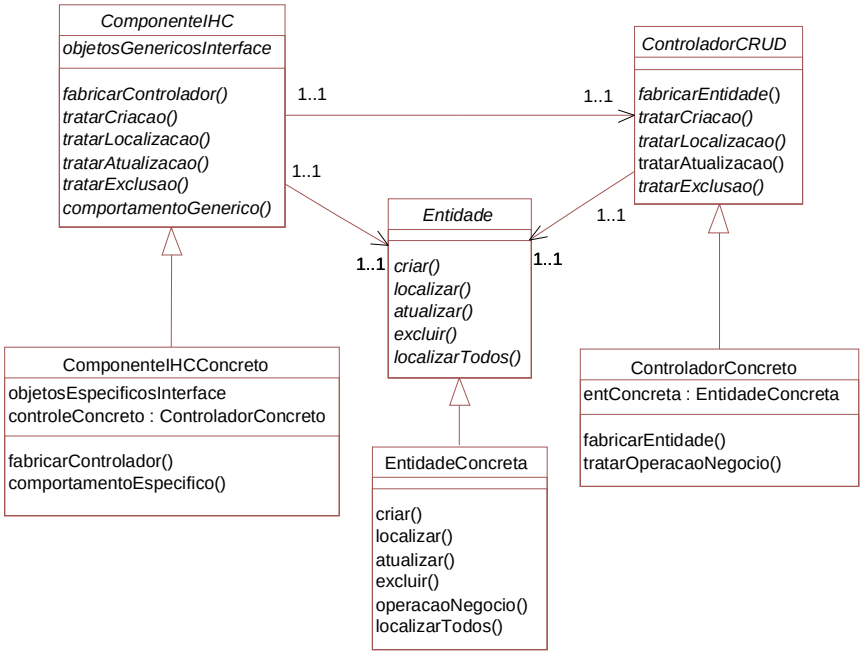

Figura 1 – Diagrama de classes do Padrão CRUD-MVC

2.6 Participantes

- **ComponenteIHC**
  - Pode ser formado por vários componentes de interface com usuário
  - Define as características de interação com usuário (*objetosGenericosInterface*) presente em todo componente de IHC de um sistema. Pelo menos, comandos para criação, localização, atualização e exclusão. Pode definir outras características a serem reutilizadas (por exemplo, ajuda e comando para fechar/ocultar interface)
  - Define comportamento existente em toda interface de usuário (*comportamentoGenerico*). Por exemplo, controle de acesso a operações e alertas
  - Fornece método *Factory* [2] *fabricarControlador* para criação do objeto de controle e armazena referência através de interface *ControladorCRUD*
  - Fornece métodos genéricos para tratamento dos eventos de interface com usuário (*tratarCriacao*, *tratarLocalizacao*, *tratarLocalizacao*, *tratarExclusao*). Os métodos de evento notificam seu observador (*ControladorCRUD*) das ações tomadas pelo usuário. Os métodos genéricos permitem definir ações de pré-processamento e pós-processamento das operações de criação, localização, atualização e exclusão. Por exemplo, antes de solicitar exclusão, a interface pode exibir um diálogo de confirmação de operação (pré-processamento de exclusão)

- **ComponenteIHCConcreto**
  - Define os campos de interface com o usuário para visualização e preenchimento das informações da entidade de negócio. Define também como os campos serão sincronizados com os atributos de *EntidadeConcreta*

- Define as características específicas de interação com usuário para a entidade de negócio: comandos para ações de negócio, pastas para manter relacionamentos com outras entidades e notificação de eventos de negócio para o *ControladorConcreto*
- Implementa o método *fabricarControlador*, criando *ControladorConcreto* para implementar *ControladorCRUD*

- **ControladorCRUD**
  - Fornece método *Factory* [2] *fabricarEntidade* para criação da entidade de negócio
  - Define as operações para tratamento de eventos: *tratarCriacao, tratarLocalizacao, tratarAtualizacao, tratarExclusao* que operam sobre a interface genérica (*Entidade*) implementada pelo objeto criado pelo método *fabricarEntidade*

- **ControladorConcreto**
  - Implementa método *Factory* [2] *fabricarEntidade* para criação da entidade de negócio (*EntidadeConcreta*)
  - Define as operações para tratamento de eventos de negócio

- **Entidade**
  - Define interface genérica para entidade de negócio com operações para criação, localização, atualização e exclusão
  - Pode ser composta por objetos que concentram a informação e objetos para acesso ao repositório de dados. No entanto, fornece interface única para o controlador. Três padrões podem ser combinados para gerar a entidade de negócio: *Data Access Object* [1], *ValueObject* [1] e Fachada [2]. Assim, utiliza-se um *ValueObject* representando a informação, um *Data Access Object* responsável por realizar a persistência do *ValueObject* e um objeto fachada para coordenar o *ValueObject* e o *Data Access Object*, expondo para os clientes uma interface de negócio unificada: atributos, métodos do ciclo de vida – criar, atualizar, excluir, consultarPorId, consultarTodos e métodos de negócio. A fachada seria a *Entidade* propriamente dita

- **EntidadeConcreta**
  - Implementa a interface de *Entidade* para os métodos de criação, localização, atualização e exclusão
  - Define os atributos que são sincronizados com os campos de *ComponenteIHCConcreto*
  - Define e implementa outros métodos de pesquisa
  - Define e implementa métodos de negócio utilizados por *ControladorConcreto*

2.7 Dinâmica

A seqüência de eventos abaixo descreve a dinâmica do padrão:

1. O usuário aciona algum comando CRUD em *ComponenteIHCConcreto;*
2. Se evento de atualização ou inclusão, a informação é sincronizada de *ComponenteIHCConcreto* para *EntidadeConcreta;*

3. O comportamento (métodos de tratamento) herdado de *ComponenteIHC* é executado, realizando algum pré-processamento e enviando uma notificação sobre o evento para o *ControladorConcreto*;
4. Em *ControladorConcreto*, o método adequado para tratamento do evento (*tratarCriacao*, *tratarLocalizacao*, *tratarAtualizacao*, ou *tratarExclusao*) é executado pelo código herdado de *ControladorCRUD*, executando o método CRUD adequado em *EntidadeConcreta* (através da interface de *Entidade*);
5. O método CRUD de *EntidadeConcreta* é executado e o controle retorna para o código herdado de *ComponenteIHC*, permitindo pós-processamento;
6. Se evento de localização, a informação é sincronizada de *EntidadeConcreta* para *ComponenteIHCConcreto*.

Os eventos de negócio específicos das classes concretas são tratados de maneira similar, utilizando, no entanto, código implementado nas classes concretas. A dinâmica do padrão é exemplificada na figura 2. A figura 2 apresenta o tratamento para o evento *criar*. Os outros eventos ocorrem de maneira análoga.

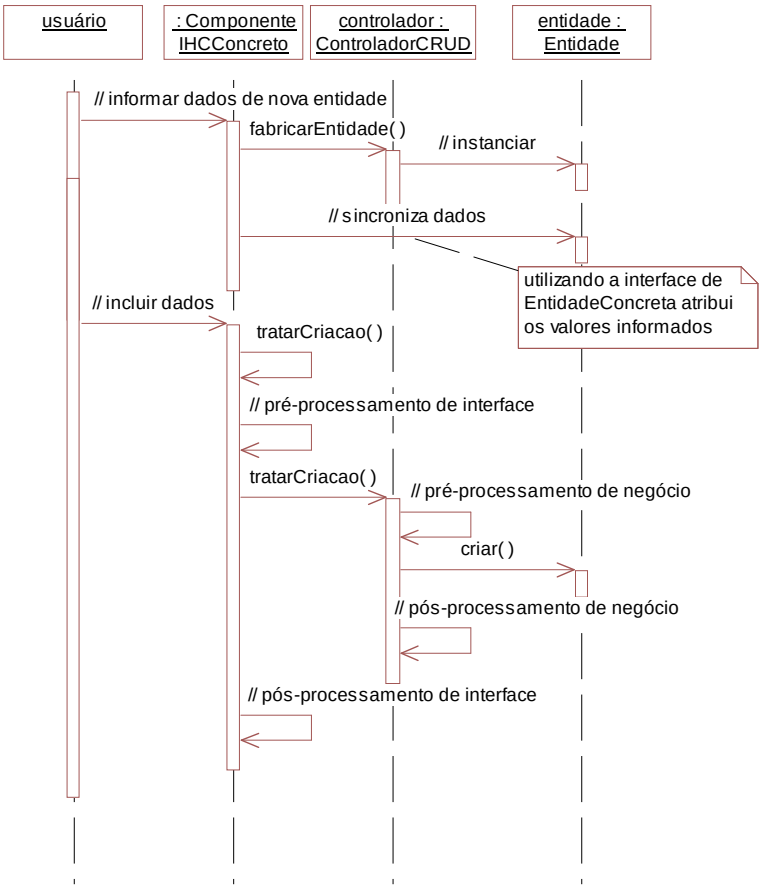

Figura 2: Diagrama de seqüência para o subfluxo de evento criar

2.8 Conseqüências

O padrão CRUD-MVC oferece os seguintes benefícios:

- A classe *ComponenteIHC* permite padronização e reuso de interface de usuário e de tratamento de eventos CRUD através de herança;

- Separação de código entre eventos CRUD e outros eventos de negócio;
- Permite a rápida extensão/manutenção de recursos de interface de usuário e tratamento de eventos através da centralização em classes abstratas que servem de base para o restante das classes da aplicação;
- Permite independência da interface de usuário com relação ao mecanismo de persistência através da interface de persistência de entidade;
- Permite a troca de componentes de IHC sem afetar o restante do código.

As seguintes desvantagens merecem ser mencionadas:

- Apesar de separar aspectos de apresentação, controle, regras de negócio e persistência, o número de classes da aplicação pode aumentar bastante com o uso do padrão. Se essa separação não é muito crítica na aplicação, as variantes 1 e 2 podem ser consideradas;
- Com a utilização do padrão a complexidade da solução torna-se maior, envolvendo relacionamento e comunicação entre vários objetos.

2.9 Padrões relacionados

- *Factory Method* [2]:
  - Utilizado na criação de objetos de Entidade e de Controle
- *MVC* (Modelo-Visão-Controle):
  - Utilizado para fornecer baixo acoplamento entre os componentes da aplicação
- *Data Access Object* [1], *ValueObject* [1] e Fachada [2]:
  - Esse três padrões são utilizados para construção de *Entidade*. Os padrões *Data Access Object* (DAO) e *ValueObject* são bastante utilizados em aplicações J2EE, mas podem ser utilizados em qualquer tipo de linguagem orientada a objetos

2.10 Variantes

De acordo com as características da linguagem utilizada na implementação do sistema ou requisitos não-funcionais, o padrão pode ser utilizado com as seguintes variantes:

1. Componente IHC e Controlador unificados

A unificação dessas duas classes aumenta o acoplamento, mas simplifica o projeto, reduz o número de classes e ainda continua fornecendo características como reutilização do tratamento de eventos e padronização dos componentes de interface e comportamento. Nesse caso, os tratamentos dos eventos com relação à interface e regras de negócio estariam juntos na classe *ComponenteIHC* e as respectivas classes concretas também seriam unificadas. Algumas regras de negócio do controle poderiam eventualmente passar para a entidade.

2. Simplificação da *Entidade*

A separação da entidade em fachada, *ValueObject* e DAO oferece uma boa separação entre operações de negócio, operações de acesso e leitura de dados e persistência, mas aumenta o número de classes do sistema. Se essa separação não for importante para a aplicação, pode-se utilizar apenas uma classe de entidade que tem os atributos com

seus métodos para leitura e escrita, métodos de negócio relacionados aos dados e código para persistência e comunicação com o repositório de dados todos juntos.

3. Utilização do padrão *Observer*
Algumas linguagens e *frameworks* (como Java *Swing*) trazem suporte natural para o padrão *Observer* [2]. Em outras linguagens a implementação do padrão pode ser mais complicada. No entanto, o uso do padrão *Observer* pode sofisticar a implementação do padrão MVC, diminuindo o acoplamento entre visão, controlador e modelo. Dois tipos de estratégias MVC podem ser adotados com o padrão *Observer*: Modelo Ativo ou Passivo. No modelo Ativo, a classe *Entidade* seria observada por *ComponenteIHC* através da interface *Observable* e qualquer mudança na *Entidade* levaria a uma notificação de *ComponenteIHC* através da interface *Observer*. Com o modelo passivo, o *ControladorCRUD* é que seria observado por *ComponenteIHC*, e quando fizesse alterações na *Entidade* notificaria *ComponenteIHC*.

2.11 Usos Conhecidos

Por motivos de confidencialidade, mais detalhes dos usos conhecidos abaixo não podem ser fornecidos.

- Sistema Imobiliário
- Sistema de controle de Processos Jurídicos
- Sistema de Administração de Recursos Humanos
- Sistema de CallCenter
- Sistemas Bancários

3. Padrão Registro com Busca

3.1 Contexto

Utilizado para as principais entidades de negócio, com muitos campos e/ou relacionamentos. Em geral, essas entidades são objetos complexos com muitos atributos e relacionamentos. O enfoque do cenário é de uma busca eficiente seguida de visualização e edição de uma entidade.

3.2 Problema

Como criar componentes de cadastro e manutenção de entidades de negócio complexas, atendendo a requisitos de interface humano-computador, permitindo a reutilização de interação com usuário e estrutura comuns aos vários tipos de entidades complexas existentes em uma aplicação de sistema de informação?

3.3 Forças

- Deve suportar reuso de características de interação com usuário, recorrentes na manutenção de entidades complexas
- Necessita de manipulação individual do objeto, mas com busca eficiente de entidades de negócio

- Deve permitir fácil visualização de dados e relacionamentos de uma entidade complexa
- Deve fornecer baixo acoplamento entre interfaces de usuário para entrada e consulta de dados, tratamento de eventos CRUD e entidade de negócio

3.4 Solução

Utilizar classe *Registro* para criação, atualização e exclusão de entidades, permitindo que uma única entidade seja visualizada e editada por vez. A classe *Registro* apresenta a informação e relacionamentos da entidade. Ela também fornece um mecanismo de localização eficiente, colaborando com a classe *Busca*. A classe *Busca* fornece critérios de pesquisa e lista de resultados. Além de buscar a entidade complexa, ela pode ser utilizada para buscar entidades para preenchimento de relacionamentos.

O padrão *Registro com Busca* utiliza o padrão *CRUD-MVC* (ver seção 2) para definir como as classes de modelo, visão e controle podem se relacionar. Dessa forma, o padrão mantem seu foco na interação com usuário para manutenção de entidades complexas e utiliza a estrutura de *CRUD-MVC* para fornecer características como extensibilidade e baixo acoplamento. As classes *Registro* e *Busca* funcionam de acordo com a classe *ComponenteIHC*. As classes *RegistroConcreta* e *BuscaConcreta* funcionam de acordo com a classe *ComponenteIHCConcreto*. As classes de controle e modelo são utilizadas também de acordo com o padrão *CRUD-MVC*. Assim, o padrão *Registro com Busca* apresenta uma solução de interação com usuário para cenários de manutenção de entidades complexas e utiliza o padrão *CRUD-MVC* para estruturar suas classes.

3.5 Estrutura

A Figura 3 apresenta o diagrama de classes do padrão e seus componentes. As classes de controle e modelo não são apresentadas e seguem o padrão *CRUD-MVC*.

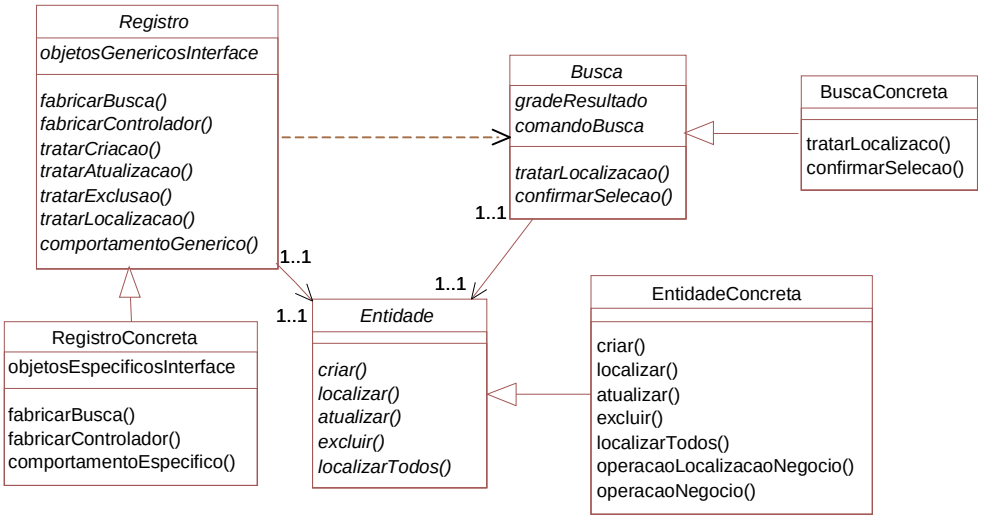

Figura 3 – Diagrama de Classes do Padrão Registro com Busca

3.6 Participantes

- **Registro**
  - Define as características de interação com usuário presente em toda classe para manutenção de entidades complexas. Fornece botão para acionar uma interface para realizar pesquisas elaboradas sobre a entidade complexa
  - Permite a edição, criação e exclusão de apenas uma entidade complexa por vez.
  - Fornece método *Factory* [2] *fabricarBusca* para criar uma instância de classe de *BuscaConcreta*. Após a criação o objeto registro exibe a tela de busca

- **Busca**
  - Fornece uma interface gráfica genérica para preenchimento de critérios de pesquisa e de apresentação de resultados
  - Fornece botão para ação de pesquisa e grade para exibição de resultados
  - Executa comunicação genérica com o objeto *Registro* para seleção e apresentação da entidade de negócio

- **RegistroConcreta**
  - Define os campos para visualização e preenchimento das informações da entidade de negócio complexa e como os campos serão sincronizados com os atributos de *EntidadeConcreta*
  - Define as características específicas de interação com usuário para a entidade de negócio: comandos para ações de negócio, pastas para manter relacionamentos com outras entidades e notificação de eventos de negócio para o *ControladorConcreto*. As pastas de entidades relacionadas indicam relacionamentos *um-para-muitos* da entidade de negócio para entidades dependentes
  - Implementa o método *fabricarControlador*, criando *ControladorConcreto* para implementar *ControladorCRUD*

- **BuscaConcreta**
  - Define os critérios concretos para a pesquisa
  - Define os campos que são apresentados na grade de resultados
  - Faz sincronização da coleção de entidades com a grade
  - Pode ser utilizada para buscar entidades para preenchimento de campos de relacionamentos da entidade complexa

3.7 Dinâmica

Os eventos de CRUD se comportam de acordo com o padrão *CRUD-MVC*. O detalhe ocorre no evento de localização. Para localizar e visualizar uma entidade, o seguinte fluxo é executado:

1. Usuário dispara comando de localização;
2. O código herdado de *Registro* é executado e o método *fabricarBusca* é executado, utilizando a implementação fornecida por *RegistroConcreta*. Em seguida à criação do objeto de busca, o mesmo é disponibilizada para o usuário;

3. O usuário informa os parâmetros da pesquisa e solicita a localização;
4. O objeto *BuscaConcreta* notifica o *ControladorConcreto* para tratamento do evento de pesquisa;
5. *O ControladorConcreto* utiliza um método de negócio de pesquisa, *localizar(params)*, da *EntidadeConcreta*, disponiblizando uma coleção de entidades para *BuscaConcreta;*
6. *BuscaConcreta* exibe alguns dados das entidades encontradas e o usuário seleciona a entidade que deseja visualizar ou editar;
7. *BuscaConcreta* disponibiliza a entidade selecionada para *RegistroConcreta* que exibe seus dados.

Em seguida, o usuário pode editar ou excluir a entidade. A criação pode ser feita a qualquer momento. A dinâmica do padrão é exemplificada na figura 4.

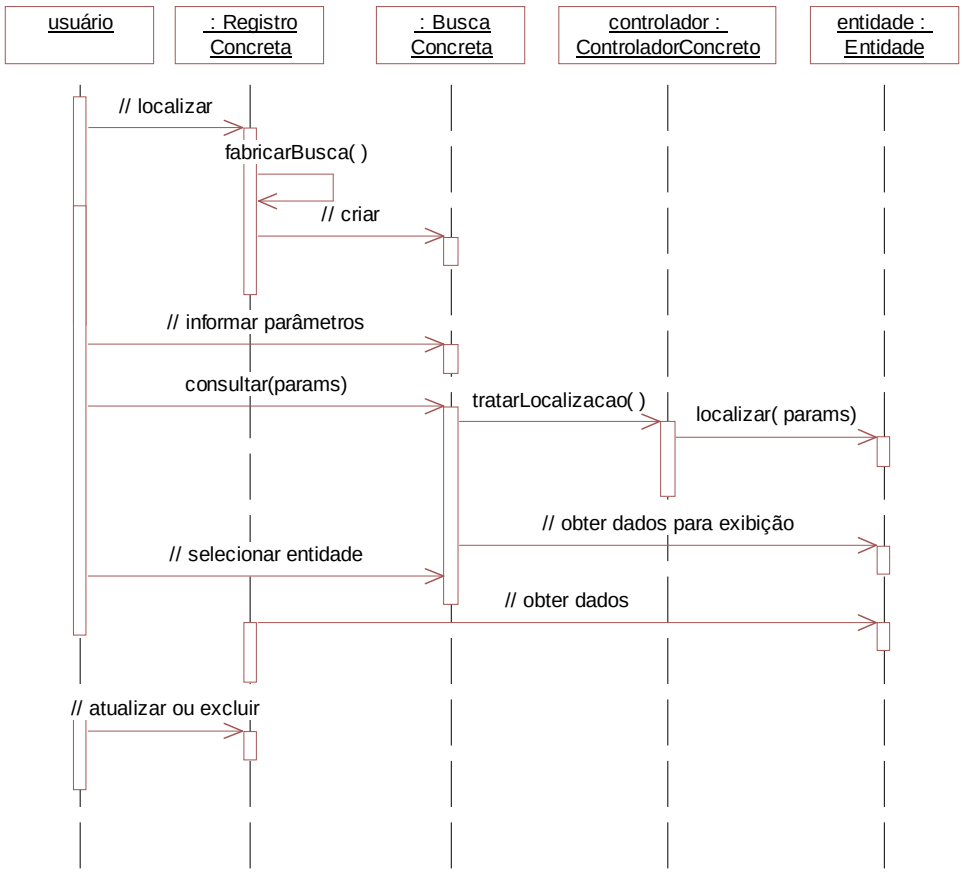

Figura 4: Diagrama de seqüência do padrão Registro com Busca

3.8 Conseqüências

O padrão Registro com Busca oferece os seguintes benefícios:

- Fornece mecanismo eficiente de busca e visualização/edição de uma única entidade, aumentando a eficiência do sistema. Economiza recursos do sistema evitando alocação em memória de coleções de entidades complexas que consomem muitos recursos;

- Permite maior usabilidade do sistema, fornecendo para o usuário uma interface adequada para visualização e edição de entidade complexa;
- O reuso de *Registro* através de herança fornece interface de usuário padronizada para registro de entidades de negócio complexas;
- Permite reuso de tratamento de eventos CRUD e interação com objeto de busca.

As seguintes desvantagens merecem ser mencionadas:

- Apesar de separar aspectos de apresentação, controle, regras de negócio e persistência, o número de classes da aplicação pode aumentar bastante com o uso do padrão;
- Com a utilização do padrão a complexidade da solução torna-se maior, envolvendo relacionamento e comunicação entre vários objetos.

3.9 Padrões relacionados

- *Factory Method* [2]:
  - Utilizado na criação de objetos de Entidade e de Busca
- *CRUD-MVC* (ver seção 2):
  - Utilizado para definir interação entre as classes do modelo, visão e controle

3.10 Variantes

O padrão *Registro com Busca* também pode ser utilizado sem a estrutura do *CRUD-MVC*, considerando apenas seu principal foco, a interação com usuário para manutenção de entidades complexas. Essa situação pode ser aplicada quando a aplicação não possui fortes requisitos de extensibilidade e baixo acoplamento.

3.11 Usos Conhecidos

- Ver *CRUD-MVC*

4. Agradecimentos

Este trabalho foi suportado pelo Instituto Atlântico.

Os autores agradecem aos responsáveis pelo processo de revisão, em especial ao Márcio de Oliveira Barros, pelas contribuições realizadas no aprimoramento do artigo.

5. Referências

[1] D. Alur, J. Crupi, D. Malks. *Core J2EE Patterns As melhores práticas e estratégias de design*, Editora Campus, 2002.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Michael. *Pattern-Oriented Software Architecture*, John Wiley & Sons, 2001.