Reuso na Atividade de Teste para Reduzir Custo e Esforço de VV&T no Desenvolvimento e na Reengenharia de Software

Maria Istela Cagnin[1*], José Carlos Maldonado[1] , Alessandra Chan[1†],
Rosângela Penteado[2] , Fernão Germano[1]
[1]Instituto de Ciências Matemática e de Computação – Universidade de São Paulo,
Av. do Trabalhador São-Carlense, 400, São Carlos-SP 13560-970
[2]Departamento de Computação – Universidade Federal de São Carlos,
Rod. Washington Luís, Km 235, São Carlos-SP 13.565-905
e-mail: [istela,jcmaldon,fernao]@icmc.usp.br, ale@grad.icmc.usp.br,
rosangel@dc.ufscar.br

Resumo

Linguagens de padrões estão sendo cada vez mais utilizadas no desenvolvimento e na reengenharia de software, e permitem o reuso de soluções de problemas em um determinado contexto. Atividades de VV&T consomem grande parte dos esforços despendidos nos projetos. A carência dessas atividades agregadas a linguagens de padrões, dificultam o reuso de requisitos de teste e, conseqüentemente, a redução do tempo e esforços gastos nos projetos. Para reduzir essa carência este artigo apresenta uma abordagem composta por: a) uma estratégia, que define e associa requisitos de teste a padrões de linguagens de padrões de análise; b) diretrizes, que apóiam o engenheiro de software na decisão de quais requisitos de teste disponíveis devem ser reusados e instanciados para casos de teste concretos. A estratégia que define os requisitos de teste da abordagem proposta foi aplicada aos padrões de uma linguagem de padrões de análise, utilizados na reengenharia de um sistema legado de biblioteca. Este artigo apresenta a aplicação dessa estratégia em um desses padrões.

Abstract

Pattern languages are used in the software development and reengineering, and they allow the reuse of problems solutions in a certain context. Most part of the efforts, in the projects, are spent with VV&T activities. The lack of those activities aggregated to pattern languages, makes difficult the reuse of testing requirements and, consequently, time and efforts reduction in the projects. For reducing that lack, this paper presents an approach composed by: a) a strategy, that defines and associates testing requirements to patterns of analysis pattern languages; b) guidelines, that support the software engineer to make decisions about which available testing requirements may be reused and instantiated for real test cases. The strategy, which defines the testing requirements of the proposed approach, has been applied to patterns of an analysis pattern language, used in the library legacy system reengineering. This paper presents the usage of this strategy in one of these patterns.

1. Introdução

A garantia da qualidade de software é a principal atividade com a qual uma equipe de desenvolvimento/manutenção/reengenharia deve se preocupar a fim de entregar um produto confiável a seus usuários. No entanto, essa atividade não é freqüentemente praticada, pois consome a maior

*Apoio Financeiro da FAPESP - nr. 00/10881–4
†Apoio Financeiro da CAPES

parte dos custos e esforços despendidos, tanto no desenvolvimento [27, 20] quanto na reengenharia de software [9], aumentando o custo do produto e o tempo de entrega.
Muitos processos de desenvolvimento/manutenção/reengenharia utilizam padrões de software [8] em diferentes contextos [17, 13, 19, 21, 24]: de análise, de projeto, de implementação, de arquitetura, etc. A qualidade do produto proveniente desses processos é imprescindível para a satisfação dos clientes e usuários. Assim, como discutido por Rocha et al. [28], a garantia da qualidade deve ser conduzida em todas as fases do processo, para que os defeitos introduzidos sejam eliminados na própria fase em que surgiram, a fim de evitar o aumento dos custos, se deixados para etapas posteriores.
PARFAIT [1] [11] é um processo ágil de reengenharia, que utiliza uma linguagem de padrões de análise, visando a facilitar e reduzir o tempo de documentação e entendimento do sistema legado. Em alguns estudos de caso de reengenharia, observou-se que atividades relacionadas a VV&T estavam consumindo mais de 80% de todo o tempo e esforço despendido. Isso evidenciou a necessidade de agregar técnicas de VV&T no contexto de padrões de software, permitindo que o engenheiro de software reuse não somente as soluções dos padrões, mas também os testes a eles associados.
Este artigo tem como objetivo apresentar uma abordagem de reuso de teste que possui uma **estratégia** para apoiar a definição de requisitos de teste funcionais para linguagens de padrões de análise e **diretrizes** que permitem o reuso dos requisitos definidos e a instanciação dos casos de teste. Essa abordagem foi motivada e criada com o apoio dos resultados das atividades de VV&T em um estudo de caso de reengenharia usando PARFAIT.
Na Seção 2, discutem-se os trabalhos relacionados. Na Seção 3 apresenta-se a abordagem de reuso de teste. A experiência de uso da estratégia de definição de requisitos de teste, da abordagem proposta, em um dos padrões de uma linguagem de padrões de análise é relatada na Seção 4. Na Seção 5, discutem-se as considerações finais e os trabalhos futuros.

2. Trabalhos Relacionados

Um padrão é um pedaço de informação instrutiva e nomeada, que captura a estrutura essencial e "*insights*" de uma família bem sucedida de soluções aprovadas para um determinado problema, o qual surge em um determinado contexto. Uma linguagem de padrões é uma coleção de tais soluções que agem juntas para resolver um problema, de acordo um um objetivo pré-definido [2]. Uma linguagem de padrões pode também ser considerada como uma coleção estruturada de padrões que transformam requisitos e restrições em arquitetura de software [14].
Tsai et al. [29] discutem testes em padrões de projeto implementados em frameworks orientados a objetos. Esse trabalho difere da estratégia proposta neste artigo quanto ao nível de abstração, pois preocupa-se em definir cenários de teste baseados na implementação das funcionalidades por meio de padrões de projeto e não se preocupa com o nível de análise. No trabalho de Tsai et al., é apresentada uma técnica denominada *Message Framework Sequence Specifications, (MfSS)*, para apoiar a geração de gabaritos de cenários que podem ser utilizados para gerar vários tipos de cenários de teste tais como, cenários de teste de partição e cenários de teste randômico. Esses cenários apóiam o teste de aplicações geradas a partir de frameworks orientados a objetos que usam padrões de projeto extensível, ou seja, padrões que permitem adicionar novas classes e métodos ao framework em tempo de compilação ou em tempo de execução. *MfSS* é uma extensão das técnicas *Method Sequence Specifications (MtSS)*, e *Message Sequence Specifications (MgSS)*, as quais especificam a interação de mensagem entre objetos de aplicações orientadas a objetos por meio de expressões regulares. *MtSS* especifica as mensagens que um determinado método pode enviar e *MgSS* especifica a seqüência de mensagens que podem ser enviadas para uma classe. Adicionalmente, *MfSS* especifica restrições de seqüência nas interações entre objetos do framework e considera comportamentos dinâmicos, ou seja, tipagem e ligação dinâmica.

[1] Processo Ágil de Reengenharia baseado em FrAmework no domínio de sistemas de Informação com VV&T.

Existem também padrões de teste [5, 16] que fornecem soluções para auxiliar os testadores na avaliação da qualidade do produto, diferentemente da estratégia proposta neste artigo.
Algumas práticas de Testes Ágeis [2] [15], foram observadas na abordagem proposta neste artigo, pois ajuda a: **identificar suposições ocultas de teste** (Seção 3.3), antes mesmo da engenharia avante (tanto no desenvolvimento quanto na reengenharia do sistema), ou seja da escrita do código; **definir teste para cada funcionalidade** (Seções 3.2.1, 3.2.2 e 3.2.3) que pertence ao domínio da linguagem de padrões de análise sendo aplicada a abordagem, **focar os testes em "o que" e não "como"** (Seções 3.2.3) e possibilita **aumentar a velocidade das atividades de teste** (Seção 3.3), com o apoio das diretrizes de reuso dos requisitos definidos na linguagem de padrões.

3. Abordagem de Reuso de Teste

Esta Seção apresenta a abordagem que apóia a definição de requisitos de teste[3] para linguagens de padrões de análise (ETAPA 1, Figura 1) e, também, o reuso desses requisitos definidos e a instanciação dos casos de teste dos sistemas, tanto no desenvolvimento quanto na reengenharia de software (ETAPA 2, Figura 1). A criação dessa abordagem foi motivada e apoiada pelos resultados de um estudo de caso de reengenharia de um sistema de biblioteca de uma Universidade [9, 12], utilizando o processo ágil PARFAIT.

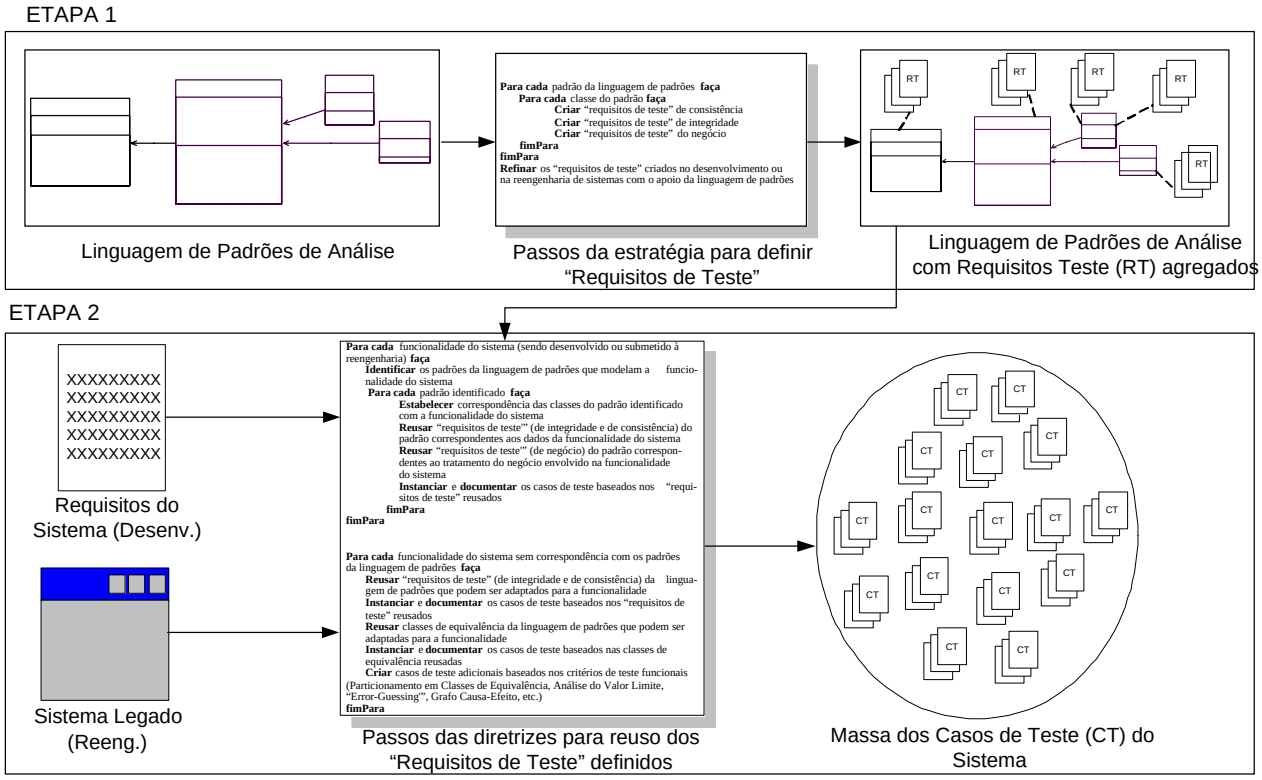

Figura 1: Visão geral da abordagem de reuso de teste em linguagens de padrões

O principal propósito do processo PARFAIT é apoiar a migração de sistemas procedurais de pequeno e médio porte para o paradigma orientado a objetos, e garantir que o produto de software resultante seja confiável e aceitável por seus usuários. Para isso usa:

- a linguagem de padrões GRN (Gestão de Recursos de Negócio) [7], para apoiar a elaboração da documentação e o entendimento do sistema legado. Essa linguagem fornece conhecimento do domínio do sistema legado, que deve pertencer ao mesmo domínio da linguagem de padrões.

[2] *Agile Testing*, em inglês.
[3] de agora em diante utilizaremos o termo "requisito de teste" para denominar "requisito de teste funcional".

Ela é formada por quinze padrões de análise que fornece aos desenvolvedores inexperientes informação suficiente para desenvolver novos sistemas no domínio de "Gestão de Recursos de Negócios", juntamente com soluções alternativas. Essa linguagem possui um domínio específico e bem definido, concentrado no aluguel, comércio e manutenção de recursos de negócios. Os padrões da GRN estão representados em UML (*Unified Modeling Language*) [18, 23];

- o framework GREN [6] como apoio computacional: a) para as atividades de engenharia reversa: na elicitação de novos requisitos e na identificação de regras de negócio do sistema legado; e b) para as atividades de engenharia avante, na criação do protótipo do sistema. A construção desse framework foi baseada na linguagem de padrões GRN, portanto facilita a reengenharia dos sistemas legados, no domínio de "Gestão de Recursos de Negócios". O framework foi construído utilizando a linguagem de programação Smalltalk e o SGBD MySQL [26];
- a participação dos usuários e clientes do sistema durante a aplicação do processo para que o produto seja validado à medida que evolui;
- testes baseados em critérios de teste funcionais, como Particionamento em Classes de Equivalência e Análise do Valor Limite [25], que apóiam a identificação das regras e funcionalidades específicas do sistema legado e também a validação do sistema produzido.

PARFAIT é incremental e iterativo, pois o engenheiro de software tem a flexibilidade de retornar a passos do processo anteriormente executados, a fim de refinar os artefatos produzidos. Além disso, é considerado um processo ágil, pois atende a diversas características e *práticas* de metodologias ágeis [1, 4, 30]: a) fornece aos usuários uma versão do novo sistema o mais rápido possível - *prática do XP (eXtreme Programming) pequenas releases* -; b) os usuários participam ativamente da maioria das atividades do projeto de reengenharia e aprovam o produto à medida que o projeto evolui. Em cada interação com os usuários, melhorias no produto são realizadas - *prática do XP clientes presentes* -; c) testes no novo sistema são realizados constantemente e comparados com os resultados dos testes do sistema legado - *prática do XP testes constantes* -; d) o planejamento da reengenharia é refeito a cada iteração do processo - *prática do XP jogo do planejamento*-; e e) incentiva a *prática do XP programação em pares*.

A Subseção 3.1 apresenta um panorama do estudo de caso de reengenharia prospectivo. A Subseção 3.2 apresenta a estratégia (da abordagem proposta) que apóia a criação e associação de requisitos de teste em linguagens de padrões de análise e a Subseção 3.3 apresenta as diretrizes (da abordagem proposta) que apóiam o reuso dos requisitos de teste criados. Quando necessário, resultados do estudo de caso de reengenharia, que foram utilizados para apoiar a criação da estratégia e das diretrizes, serão retomados.

3.1. Estudo de Caso de Reengenharia Prospectivo para a Criação da Abordagem Proposta

O sistema legado do estudo de caso de reengenharia é um sistema que controla a biblioteca de uma determinada universidade e foi originalmente implementado em linguagem Clipper com aproximadamente 6 KLOC. Um usuário do sistema legado, que é professor da universidade, participou durante todo o estudo de caso. Maiores detalhes sobre esse estudo de caso podem ser encontrados em [9, 12]. A Tabela 1 mostra a seqüência das atividades do PARFAIT[4] executadas no estudo de caso e destaca com fonte negrito e itálico a atividade *"Desenvolver o diagrama de casos de uso e documentar os casos de teste"*, que é inerente a criação dos casos de teste e que consumiu 81% (552:50 hs do total de 675:29 hs) de todo o tempo da reengenharia. Nessa atividade, o sistema legado é executado para que o engenheiro de software se familiarize com suas funcionalidades, as quais são documentadas em um diagrama de casos de uso [18]. Todos os casos de teste utilizados para exercitar o sistema legado são documentados para serem posteriormente utilizados a fim de identificar regras do negócio

[4]maiores detalhes de cada atividade podem ser encontrados em [11].

e funcionalidades específicas do sistema legado, por meio de teste comparativo do sistema legado com o protótipo do sistema criado pelo framework, e a fim de validar o novo sistema. Os casos de teste foram criados com o apoio dos critérios de teste funcionais Particionamento em Classes de Equivalência e Análise do Valor Limite [25].

Tabela 1: Seqüência das atividades do PARFAIT executadas no estudo de caso de reengenharia [9]

Seqüência das Atividades Executadas	Atividades do PARFAIT	Tempo Gasto por Atividade
1	Familiarizar-se com o domínio do framework	2:00
2	Observar o domínio do sistema legado em relação ao do framework	0:40
3	Confrontar as características não funcionais do framework com as do sistema legado	0:50
4	Elaborar o planejamento do projeto de reengenharia	1:00
5,18,25	***Desenvolver o diagrama de casos de uso e documentar os casos de teste***	***552:50***
6,8,10,15,22	Desenvolver o diagrama de classes do sistema	16:10
7,9,11,16,23	Documentar as modificações realizadas no diagrama de classes	8:16
12	Desenvolver um protótipo do sistema no paradigma orientado a objetos	14:10
13,19,26	Executar os casos de teste no protótipo do sistema	21:30
14,21	Documentar as regras do negócio do sistema	1:15
17,20,24	Adaptar o protótipo do sistema	42:38
27	Elaborar o manual do usuário do sistema	3:00
28	Converter a base de dados do sistema legado	3:20
29	Testar o sistema no paradigma orientado a objetos	12:20
não executada	Treinar os usuários finais	0:00
	Total	**675:29**

Para a atividade em questão, foram realizadas 3 iterações (5, 18, 25), primeira coluna da Tabela 1, sendo criadas 174 classes de equivalência com o apoio do critério de teste Particionamento em Classes de Equivalência e enumeradas como mostradas parcialmente na Tabela 2. A partir dessas classes e também do uso do critério de teste Análise do Valor Limite, 362 casos de teste foram derivados, e estão parcialmente mostrados na Tabela 3.

Tabela 2: Documentação parcial das classes de equivalência para a funcionalidade empréstimo de livro [10]

Restrições de Entrada	Classes Válidas	Classes Inválidas
código do estudante	tamanho da seqüência de caracteres numéricos = 8 caracteres **(1)**	seqüência de caracteres **(2)**
		caracteres numéricos tamanho da seqüência diferente de 8 **(3)**
existência do estudante	existência **(4)**	não existência **(5)**
código do exemplar	seqüência de até 6 caracteres numéricos **(8)**	seqüência de caracteres **(9)**
		seqüência maior do que 6 caracteres numéricos **(162)**
situação do exemplar para empréstimo	exemplar não emprestado **(10)**	exemplar emprestado **(11)**
Data do empréstimo	Dia e mês válidos **(16)**	Dia e mês inválidos **(17)**
	Ano maior ou igual a 0100 **(18)**	Ano menor do que 0100 **(19)**
Legenda: () = número da classe de equivalência		

O diagrama de classes do sistema de biblioteca foi criado durante cinco iterações da atividade *"Desenvolver o diagrama de classes do sistema"*, como pode ser observado pela Tabela 1. A construção desse diagrama é apoiada pela linguagem de padrões GRN, em que partes (*chuncks*) da documentação do sistema legado são identificadas por meio dos padrões dessa linguagem, como mostra a Figura 2. Foram utilizados os seguintes padrões da GRN: 1 - *Identificar o Recurso*, 2 - *Quantificar o Recurso*, and 4 - *Locar o Recurso*. As classes que compõem os padrões estão ilustradas na parte superior da figura e as classes do sistema estão ilustradas na parte inferior, e são representadas como subclasses das classes dos padrões, com atributos e métodos específicos do sistema. Anotações UML contidas no diagrama de classes indicam o papel das classes do padrão, quando necessário, e a implementação

Tabela 3: Documentação parcial dos casos de teste para a funcionalidade empréstimo de livro

Id.Caso de Teste	Id.Classe de Equivalência	Entrada	Saída Esperada
CT1	1, 4, 8, 10, 14, 16, 18, 139, 141	código do estudante = 20000101, código do exemplar = 1000, data do empréstimo = 12/07/2003	exibe data de devolução, empréstimo é criada
CT2	2	código do estudante = “a”	empréstimo não criado
CT3	3	código do estudante = 0	empréstimo não criado
CT7	5	código do estudante = 9999 (não existe)	empréstimo não criado
CT9	9	código do estudante = 20000101, código do exemplar = “teste”	empréstimo não criado
CT10	8	código do estudante = “b”, código do exemplar = 4444 (não existe)	empréstimo não criado
CT11	1, 4, 8, 10, 14, 17, 18	código do estudante = 20000101, código do exemplar = 1000, data do empréstimo = 32/17/2003	empréstimo não criado
CT12	1, 4, 8, 10, 14, 16, 19, 139	código do estudante = 20000101, código do exemplar = 1000, data do empréstimo = 12/07/2003, empréstimo cancelado antes de ser confirmado	empréstimo não criado
CT13	1, 4, 8, 11, 14	código do estudante = 20000101, código do exemplar = 1010 (emprestado)	empréstimo não criado
CT316	1, 4, 162	código do estudante = 20000101, código do exemplar = 99.999	empréstimo não criado
CT352	1, 4, 8, 10, 14, 16, 18, 139, 141	código do estudante = 20000101, código do exemplar = 1000, data do empréstimo = 12/07/2003, solicita impressão do recibo do empréstimo	data da devolução, empréstimo é criado, recibo do empréstimo é impresso
…	…	…	…

das regras do negócio, quando existirem. Alguns tipos de dados (*ListaDiscreta*[5], *ListaTabela*[6], e *ListaMultivalorada*[7]), específicos do framework GREN, foram utilizados nesse diagrama de classes para representar alguns tipos de dados não suportados pela linguagem de padrões.

3.2. Estratégia para Associar Requisitos de Teste a Linguagens de Padrões de Análise: ETAPA 1 da Abordagem Proposta

Os dados processados pelos sistemas utilizam algum tipo de sistema de armazenamento (ou seja, arquivos texto, banco de dados relacional, banco de dados orientado a objetos, etc), que garantem a sua integridade para que possam ser corretamente recuperados. Assim, para criar uma massa de requisitos de teste mais completa é necessário que as regras de integridade, atendidas pelos sistemas de armazenamento, sejam também consideradas. Nesta estratégia particularidades de integridade de banco de dados relacional são consideradas.
Inicialmente, foram definidos três tipos de requisitos de teste: de consistência, de integridade e do negócio:

1. **requisito de teste de consistência**: baseado nas diretrizes para identificar as classes de equivalência do critério funcional Particionamento em Classes de Equivalência [25] e do Teste de Validação de Entrada, proposto por Hayes e Offutt [22], que identifica os dados de teste que tentam mostrar a presença ou a ausência de falhas específicas, pertinentes a “tolerância a entradas” (ou seja, habilidade da aplicação processar adequadamente valores de entrada esperados e inesperados). Tal requisito também foi observado pelos resultados da atividade *“Desenvolver o diagrama de casos de uso e documentar os casos de teste”* do estudo de caso de reengenharia, Tabela 2, primeira, terceira e quinta linha. Esse tipo de requisito preocupa-se com a consistência dos dados, como: tamanho e valor (mínimo e máximo), tipo (`integer`, `float`, `string`, `date`, etc).

[5]similar a tipo enumerado.
[6]similar a tipo enumerado, mas os dados são provenientes de uma classe.
[7]simular a atributo multivalorado.

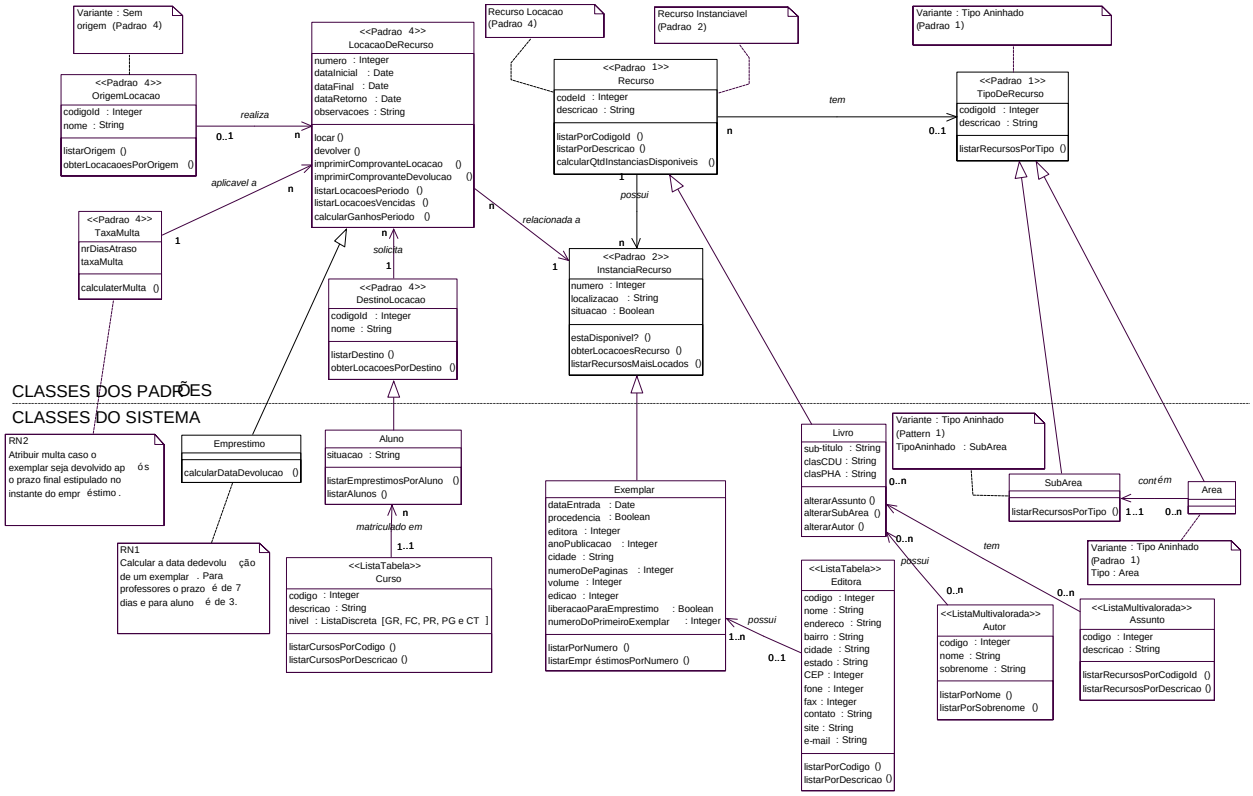

Figura 2: Diagrama de classes do sistema de biblioteca [10]

2. **requisito de teste de integridade**: baseado e limitado a integridade de bancos de dados relacionais e preocupa-se com a integridade dos dados armazenados fisicamente, como: existência de valor único de chave primária, garantia da existência da chave primária, estabelecida como valor de chave estrangeira em outra tabela, etc. A criação desse tipo de requisito também pôde ser observada pelos resultados da atividade *"Desenvolver o diagrama de casos de uso e documentar os casos de teste"*, Tabela 2, segunda linha.
3. **requisito de teste do negócio**: baseado nas particularidades do funcionamento do negócio. Esse tipo de requisito preocupa-se com as funcionalidades do negócio embutidas no padrão. A criação desse tipo de requisito também pôde ser observada pelos resultados da atividade *"Desenvolver o diagrama de casos de uso e documentar os casos de teste"*, Tabela 2, quarta linha.

Em seguida, foram estabelecidos os passos da estratégia para criar cada um dos tipos de requisitos de teste, para cada padrão das linguagens de padrões de análise. A definição desses passos foi baseada na experiência de uso dos critérios funcionais Particionamento em Classes de Equivalência e Análise do Valor Limite no estudo de caso de reengenharia discutido. A Tabela 4 apresenta a seqüência de passos criada para definir e associar requisitos de teste a linguagens de padrões. Os mesmos critérios de teste funcionais utilizados pelo PARFAIT (Particionamento em Classes de Equivalência e Análise do Valor Limite) também são utilizados para a criação dos requisitos de teste pois são os mais utilizados e difundidos. A seguir, os passos da estratégia são apresentados em detalhes.

3.2.1. PASSO 1 - Criar requisitos de teste de consistência.

- **Para cada** atributo da classe do padrão que está sendo considerada, verificar o seu tipo:
  - **Caso** tipo do atributo seja:
    * `Integer` ou `Float` criar:
      · classes de equivalência (válidas e inválidas) que considerem: valor máximo e mínimo que o atributo pode ter, valor do atributo dentro do

Tabela 4: Passos da Estratégia para Associar Requisitos de Teste

Para cada padrão da linguagem de padrões de análise
- **Para** cada classe do padrão
  - **Criar** requisitos de teste de consistência (PASSO 1)
  - **Criar** requisitos de teste de integridade (PASSO 2)
  - **Criar** requisitos de teste do negócio (PASSO 3)

Refinar os requisitos de teste criados (PASSO 4)

intervalo válido estabelecido, valor nulo (ou zero) para o atributo; tamanho máximo e mínimo do atributo e tamanho do atributo dentro do limite determinado.

* `String` criar:
  - classes de equivalência (válidas e inválidas) que considerem: tamanho máximo e mínimo, tamanho dentro do limite estabelecido e valor nulo (vazio).
* `Date` criar:
  - classes de equivalência (válidas e inválidas) que considerem: dia com valor maior que o número de dias permitido para o mês, dia com valor menor que zero, dia com valor igual a zero, dia com valor dentro do esperado para o mês, mês maior que 12, mês menor que zero, mês igual a zero, mês entre 1 e 12, ano com valor de milênio maior que o atual, ano menor que zero, ano igual a zero, ano com a formatação estabelecida (2 dígitos ou 4 dígitos), data menor do que a data atual, data maior do que a data atual, data atual e ano bissexto.

– **Caso contrário**:
* verificar as possíveis classes de equivalência (válidas e inválidas) que podem ser abstraídas a partir das sugestões estabelecidas para os tipos `integer`, `float`, `string` e `date` e criar outras, que considerem características específicas do tipo em questão.

- **Criar** requisitos de teste baseados nas classes de equivalência (válidas e inválidas) definidas.
- **Criar** requisitos de teste baseados no critério Análise do Valor Limite a partir das classes de equivalência definidas.
- **Para cada** requisito de teste de consistência criado
  – **Classificar** o requisito de teste (SUB-PASSO A).
  – **Documentar** o requisitos de teste (SUB-PASSO B).

SUB-PASSO A - Classificar requisitos de teste: Este sub-passo trata da classificação dos requisitos de teste criados, que é baseada nas operações de manipulação de dados (inserção, alteração, remoção ou recuperação) em que o requisito de teste deve ser considerado:

- **Para cada** requisito de teste criado, classifique-o de acordo com as operações de manipulação de dados, considerando:
  – **operação de inserção**: valor do atributo, que é chave primária, não pode existir no banco de dados (sugestão: alguns requisitos de teste de integridade pertencem a essa classificação); todos os demais atributos durante a inserção devem ser consistidos (sugestão: requisitos de teste de consistência pertencem a essa classificação);
  – **operação de alteração**: valor do atributo, que é chave primária, deve existir no banco de dados (sugestão: alguns requisitos de teste de integridade pertencem a essa classificação); todos os demais atributos, que podem sofrer alterações, devem ser consistidos (sugestão: requisitos de teste de consistência pertencem a essa classificação).
  – **operação de remoção**: valor do atributo, que é chave primária, deve existir no banco de dados e, em geral, não pode ser valor de chave estrangeira de outra tabela (sugestão: alguns requisitos de teste de integridade pertencem a essa classificação).
  – **operação de recuperação**: valor do atributo, que está sendo considerado na recuperação, deve existir no banco de dados.

SUB-PASSO B - Documentar requisitos de teste: Este sub-passo trata da documentação dos requisitos de teste criados, que devem ser documentados com as seguintes informações (sugere-se que a documentação seja feita em formato tabular):

- **número do requisito de teste**;
- **tipo** do requisito de teste (ou seja, de integridade, de persistência ou do negócio);
- nome do **padrão da linguagem de padrões**;
- nome da **classe do padrão**;
- tipo de **operação** de manipulação de dado em que o requisito de teste é considerado (ou seja, inserção, alteração, remoção ou recuperação);
- **tabela do SGBD** correspondente à classe que o requisito de teste pertence;
- número das **classes de equivalência utilizadas** para criar o requisito de teste;
- **especificação do requisito de teste**, que descreve o que o dado de entrada do teste a ser instanciado futuramente deve considerar;
- **condições válidas** que serão consideradas para analisar o requisito de teste e estabelecer o retorno esperado;
- **validações anteriores**, ou seja, requisitos de teste que são considerados como pré-condição do requisito de teste que está sendo documentado;
- **observação**, que informa alguma informação relevante a respeito do requisito de teste, e;
- **retorno esperado**.

3.2.2. PASSO 2 - Criar requisitos de teste de integridade.

- **Identificar** cada atributo da classe que será mapeado como chave primária no banco de dados relacional:
  - **Criar** classes de equivalência (válidas e inválidas), que considerem: valor da chave primária existe no banco de dados, valor da chave primária não existe no banco de dados, valor da chave primária existe no banco de dados e é chave estrangeira de outra(s) tabela(s) do banco de dados relacional.
- **Identificar** os relacionamentos, como chaves estrangeiras no banco de dados relacional, entre a classe que está sendo analisada e as demais classes dos padrões:
  - **Criar** classes de equivalência (válidas e inválidas), que considerem: valor da chave estrangeira existe na tabela de origem, valor da chave estrangeira não existe na tabela de origem.
- **Criar** requisitos de teste baseados nas classes de equivalência (válidas e inválidas) definidas.
- **Criar** requisitos de teste baseados no critério Análise do Valor Limite a partir das classes de equivalência definidas.
- **Para cada** requisito de teste de integridade criado
  - **Classificar** o requisito de teste (SUB-PASSO A, Subseção 3.2.1).
  - **Documentar** o requisitos de teste (SUB-PASSO B, Subseção 3.2.1).

3.2.3. PASSO 3 - Criar requisitos de teste do negócio. Identificar funcionalidades específicas do negócio (regras de negócio do domínio ao qual a linguagem de padrões pertence), por exemplo, em uma locação, o recurso só pode ser locado se estiver disponível no momento.

- **Para cada** funcionalidade específica do negócio identificada:
  - **Criar** classes de equivalência (válidas e inválidas).
  - **Criar** requisitos de teste baseados nas classes de equivalência definidas.
  - **Criar** requisitos de teste baseados no critério Análise do Valor Limite a partir das classes de equivalência definidas.
- **Para cada** requisito de teste do negócio criado:
  - **Classificar** o requisito de teste (SUB-PASSO A, Subseção 3.2.1).
  - **Documentar** o requisitos de teste (SUB-PASSO B, Subseção 3.2.1).

3.2.4. PASSO 4 - Refinar os requisitos de teste criados. Os requisitos de teste criados devem ser refinados por meio de seu reuso no desenvolvimento ou na reengenharia de sistemas com o apoio de linguagens de padrões. Diretrizes para apoiar o reuso dos requisitos de teste pelo engenheiro de software foram definidas na abordagem proposta e estão apresentadas na Seção 3.3.

3.3. Diretrizes para Instanciar Casos de Teste a partir dos Requisitos Definidos: ETAPA 2 da Abordagem Proposta

Diretrizes foram estabelecidas (Tabela 5) para que os requisitos de teste, agregados aos padrões de linguagens de padrões de análise, possam ser reusados. Essas diretrizes motivam também o reuso de requisitos de teste para funcionalidades do sistema que não possuem correspondência com os padrões da linguagem de padrões de análise. As diretrizes inerentes a **instanciação** e **documentação** dos casos de teste foram baseadas nos resultados da criação dos casos de teste do estudo de caso de reengenharia prospectivo, Tabela 3. A criação das demais diretrizes foi apoiada pela experiência da aplicação da linguagem de padrões GRN no estudo de caso de reengenharia.

Tabela 5: Diretrizes para instanciar casos de teste a partir dos requisitos definidos

Para cada funcionalidade do sistema (sendo desenvolvido ou submetido à reengenharia)
- **Identificar** os padrões da linguagem de padrões de análise que modelam a funcionalidade do sistema
  - **Para** cada padrão identificado
    - **Estabelecer** correspondência das classes do padrão identificado com a funcionalidade do sistema
    - **Reusar** requisitos de teste de integridade e de consistência do padrão correspondentes aos dados da funcionalidade do sistema
    - **Reusar** requisitos de teste do negócio do padrão correspondentes ao tratamento do negócio envolvido na funcionalidade do sistema
    - **Instanciar**[a] e **documentar** os casos de teste baseados nos requisitos de teste reusados

Para cada funcionalidade do sistema sem correspondência com os padrões da linguagem de padrões de análise
- **Reusar** requisitos de teste de integridade e de consistência da linguagem de padrões de análise que podem ser adaptados para a funcionalidade
- **Instanciar** e **documentar** os casos de teste baseados nos requisitos de teste reusados
- **Reusar** classes de equivalência da linguagem de padrões de análise que podem ser adaptadas para a funcionalidade
- **Instanciar** e **documentar** os casos de teste baseados nas classes de equivalência reusadas
- **Criar** casos de teste adicionais baseados nos critérios de teste funcionais Particionamento em Classes de Equivalência e Análise do Valor Limite.

[a] o verbo **instanciar** nestas diretrizes significa criar casos de teste baseados na especificação do requisito de teste e no retorno esperado.

A documentação recomendada para os casos de teste instanciados a partir dos requisitos de teste reusados, consiste das seguintes informações: nome da funcionalidade, número do requisito de teste reusado, número do caso de teste instanciado, entrada (contém informação utilizada como dado de entrada do teste) e saída esperada (informação que representa a saída esperada do teste).
Um outro estudo de caso de reengenharia do sistema legado de controle de biblioteca está sendo planejado para que as diretrizes de reuso possam ser aplicadas, a fim de estimar a redução do tempo de reengenharia.

4. Uso da Estratégia da Abordagem Proposta para Associar Requisitos de Teste na Linguagem de Padrões GRN

A estratégia (Tabela 4) da abordagem proposta foi aplicada aos seguintes padrões da GRN, pois foram os utilizados na reengenharia do sistema de biblioteca:

- *Padrão 1: Identificar o Recurso* - apóia a identificação dos recursos do negócio envolvidos nas transações processadas pelo sistema. Por exemplo, livro é um recurso em um sistema de biblioteca;
- *Padrão 2: Quantificar o Recurso* – estabelece como a aplicação quantifica o recurso do negócio. Por exemplo, o livro é quantificado como um recurso instanciável, pois os seus exemplares é que são tratados nas transações do sistema;

- *Padrão 3: Armazenar o Recurso* – identifica como a aplicação controla o armazenamento do recurso do negócio. Por exemplo, os livros são armazenados fisicamente em corredores e/ou prateleiras; e
- *Padrão 4: Locar o Recurso* – identifica como os aluguéis dos recursos são gerenciados pela aplicação. Por exemplo, empréstimo e devolução de livros, no caso de uma aplicação de controle de biblioteca.

Para exemplificar um resumo da aplicação da estratégia, o Padrão 4 da GRN foi tomado como exemplo (os detalhes dos passos da estratégia utilizados estão detalhados na Subseção 3.2.1, Subseção 3.2.2, e Subseção 3.2.3), por tratar de uma das principais funcionalidades do domínio de "Gestão de Recursos de Negócios".
A Figura 3 apresenta o diagrama de classes do padrão 4 da GRN, mostrado inicialmente na Figura 2, e o mapeamento da classe "Locação de Recurso" para o banco de dados relacional MySQL.

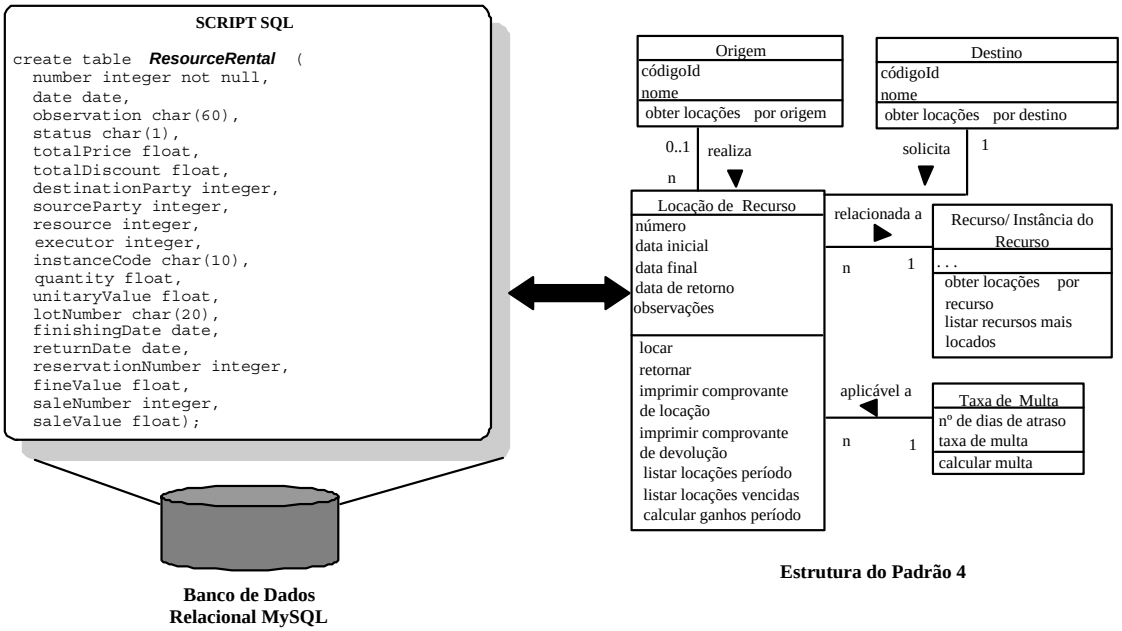

Figura 3: Padrão 4 da GRN – Locar o Recurso

PASSO 1: as classes de equivalência referentes a consistência dos dados foram criadas com o apoio do script SQL das tabelas correspondentes às classes do padrão 4, Figura 3, pois o tipo e o tamanho de cada atributo das classes não estavam especificados na estrutura do padrão. Esse script está representado na língua inglesa, como foi originalmente criado na implementação do framework GREN [6]. A Tabela 6 ilustra uma das classes de equivalência criadas para a classe "Locação de Recurso", padrão 4 da GRN, obtida com a realização do Passo 1.

Tabela 6: Exemplo de uma classe de equivalência para requisitos de teste de consistência

Operação	Classe do Padrão	Tabela Correspondente	Atributo	Classes Válidas	Classes Inválidas
Inserção	Locação de Recurso	Rental Resource	número	número inteiro (3); número entre 1 e 2147483647 (5); número preenchido (10)	número não numérico (63); número $<$ 1 (6); número $>$ 2147483647 (7); número vazio (12)

O número entre parênteses, que aparece nas colunas "classes válidas" e "classes inválidas" das Tabelas 6, 7 e 8, identificam cada classe de equivalência criada.
PASSO 2: a criação das classes de equivalência de integridade foi parcialmente apoiada pelo script SQL das tabelas correspondentes às classes do padrão. Por meio desse script, o engenheiro de software pôde verificar se as chaves primárias e estrangeiras identificadas estavam corretas. A Tabela 7 ilustra

Tabela 7: Exemplo de duas classes de equivalência para requisitos de teste de integridade

Operação	Classe do Padrão	Tabela Correspondente	Atributo	Classes Válidas	Classes Inválidas
Inserção	Locação de Recurso	Rental Resource	número	número não cadastrado (16);	número cadastrado (15)
Remoção	Locação de Recurso	Rental Resource	número	número cadastrado (15); número não cadastrado como chave estrangeira em outras tabelas (17)	número não cadastrado (16); número cadastrado como chave estrangeira em outras tabelas (18)

duas das classes de equivalência criadas para a classe "Locação de Recurso", padrão 4 da GRN, obtida com a realização do Passo 2.

PASSO 3: a criação das classes de equivalência do negócio foi apoiada pelo conhecimento do engenheiro de software no domínio ao qual a linguagem de padrões pertence. Esse conhecimento pôde também ser obtido pelo estudo da própria linguagem. A Tabela 8 ilustra uma das classes de equivalência criada para a classe "Locação de Recurso", padrão 4 da GRN, obtida com a realização do Passo 3.

Tabela 8: Exemplo de uma classe de equivalência para requisito de teste do negócio

Operação	Classe do Padrão	Tabela Correspondente	Atributo	Classes Válidas	Classes Inválidas
Inserção	Locação de Recurso	Rental Resource	–	código da instância cadastrado (15); código da instância disponível (57)	código da instância não cadastrado (16); código da instância não disponível (58)

A Tabela 9, Apêndice I, apresenta a documentação de alguns requisitos de teste criados a partir das classes de equivalência definidas. Para os quatro primeiros padrões da linguagem de padrões GRN foram definidas 63 classes de equivalência e 1272 requisitos de teste, os quais estão disponíveis para serem reusados. Para a criação desses requisitos foram gastas aproximadamente 60 horas.

5. Considerações Finais e Trabalhos Futuros

A estratégia da abordagem proposta que define e associa requisitos de teste a linguagens de padrões mostrou-se efetiva e eficiente quando aplicada a alguns padrões da linguagem de padrões GRN. Porém, é necessário que ela seja usada em outras linguagens de padrões de análise, em diferentes domínios, para se avaliar a sua aplicabilidade e flexibilidade. Um outro esforço de reengenharia será realizado no sistema legado de controle de biblioteca para estimar a redução do tempo de reengenharia, com o reuso dos requisitos de teste definidos para a linguagem de padrões GRN.

Sabe-se que cada critério de teste contribui com um conjunto de casos de teste particular, mas nenhum deles oferece um conjunto completo. Portanto, é necessário utilizar critérios funcionais e estruturais para que se possa alcançar tal conjunto, uma vez que, de acordo com Myers [25], critérios de teste funcionais e estruturais devem ser utilizados conjuntamente para que um complemente o outro.

Neste trabalho foram tratados apenas requisitos de teste funcionais, uma vez que somente a especificação do produto, representada pelos padrões de linguagens de padrões de análise, foi considerada. Assim, pretende-se definir outras estratégias para criar requisitos de teste, baseados em critérios estruturais e em outras técnicas, como por exemplo a *MfSS*, proposta por Tsai et al. [29], para associá-los às classes de frameworks baseados em linguagens de padrões, como é o caso do GREN. Dessa forma, poder-se-á obter uma cobertura maior dos testes e, conseqüentemente, a garantia do produto com maior qualidade.

Para que o engenheiro de software possa usar não somente os requisitos de teste como também a sua implementação, é necessário que casos de teste sejam instanciados e implementados em frameworks de teste, como *SUnit* [8] [3] e *JUnit* [9]. Isso permitirá ao engenheiro de software avaliar automaticamente os testes que falharam e observar a cobertura dos testes utilizados, a fim de verificar quantativamente a qualidade do sistema e decidir pela criação ou não de mais testes. A implementação de testes

[8] http://sunit.sourceforge.net/

[9] http://junit.sourceforge.net/

no framework *SUnit*, instanciados a partir dos requisitos de teste criados no padrões 1, 2, 3 e 4 da linguagem de padrões GRN, já foi iniciada.

Referências

1. Abrahamsson, P.; Salo, O.; Ronkainen, J.; Warsta, J. Agile software development methods. review and analysis. ESPOO (Technical Research Centre of Finland)' 2002. VTT Publications n. 478, `http://www.inf.vtt.fi/pdf/publications/2002/P478.pdf`. Acessado em: Dezembro, 2003.
2. Appleton, B. Patterns and software: Essential concepts and terminology. site, `http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html`. Acessado em: Dezembro,2003.
3. Beck, K. Simple smalltalk testing: With patterns. site, `http://www.xprogramming.com/testfram.htm`. Acessado em: Março, 2003.
4. Beck, K. Extreme programming explained: Embrace change. Second ed. Addison-Wesley, 2000.
5. Binder, R. V. Testing object-oriented systems: Models, patterns, and tools. First ed. Addison-Wesley, 1999.
6. Braga, R. Um processo para construção e instanciação de frameworks baseados em uma linguagem de padrões de domínio específico. Tese de Doutoramento, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2003.
7. Braga, R. T. V.; Germano, F. S. R.; Masiero, P. C. A pattern language for business resource management. In: PLOP'1999, Conference on Pattern Languages of Programs, 1999, p. 1–33.
8. Buschmann, F.; al Pattern-oriented software architeture. In: European Conference on Object-Oriented Programming, 1997.
9. Cagnin, M. I.; Maldonado, J. C.; Germano, F. S.; Chan, A.; Penteado, R. D. Um estudo de caso de reengenharia utilizando o processo PARFAIT. In: SDMS'2003, Simpósio de Desenvolvimento e Manutenção de Software da Marinha, Niterói, RJ, CD-ROM, 2003.
10. Cagnin, M. I.; Maldonado, J. C.; Germano, F. S.; Masiero, P. C.; Chan, A.; Penteado, R. D. An agile reverse engineering process based on a framework. In: WER'2003, 6th International Workshop on Requirements Engineering, 2003, p. 240–252.
11. Cagnin, M. I.; Maldonado, J. C.; Germano, F. S.; Penteado, R. D. PARFAIT: Towards a framework-based agile reengineering process. In: ADC'2003, Agile Development Conference, IEEE, 2003, p. 22–31.
12. Chan, A.; Cagnin, M. I.; Maldonado, J. C. Aplicação do processo de reengenharia PARFAIT em um sistema legado de controle de biblioteca, Documento de Trabalho, ICMC-USP, 2003.
13. Coad, P.; North, D.; Mayfield, M. Object models: Strategies, patterns and applications. Second ed. Yourdon Press, 1997.
14. Coplien, J. O. The patterns handbook: Techniques, strategies, and applications, cáp. Software Design Patterns: Common Questions and Answers Cambridge University Press, p. 311–320, 1998.
15. Crispin, L.; House, T. Testing extreme programming. The XP Series. Addison-Wesley, 2003.
16. DeLano, D. E.; Rising, L. Pattern Languages of Program Design 3, Capítulo: Software Design Patterns: Common Questions and Answers, Reading-MA, p. 503–525, 1998.
17. F. Buschmann97 et al. Pattern-oriented software architecture - a system of patterns. Wiley, 1996.
18. Fowler, M.; Scott, K. UML Distilled - Applying the Standard Object Modeling Language. First ed. Addison-Wesley, 1997.
19. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design patterns: Elements of reusable object-oriented software. Addison Wesley, 1995.

20. Harrold, M. J. Testing: A roadmap. In: 22th International Conference on Software Engineering, 2000.
21. Hay, D. Data model patterns: conventions of thought. New York, USA: Dorset House Publishing, 1996.
22. Hayes, J. H.; Offutt, A. J. Increased software reliability through input validation analysis and testing. In: ISSRE'1999, 10th International Symposium on Software Reliability Engineering, Boca Raton, FL, USA, 1999, p. 199–209.
23. IBM Rational Software. `http://www-306.ibm.com/software/rational/`. Acessado em: Abril, 2004.
24. Larman, C. Applying uml and patterns: An introduction to object-oriented analisys and design. Prentice-Hall, 1997.
25. Myers, G. J. The art of software testing. Wiley, 1979.
26. MySQL. Mysql reference manual. `http://www.mysql.com/doc/en/index.html`. Acessado em: Dezembro, 2003.
27. Pressman, R. Software engineering: A practitioner's approach. Fifth ed. McGraw-Hill, 2001.
28. Rocha, A. R.; Maldonado, J.; Weber, K. Qualidade de software: Teoria e prática. First ed. Prentice Hall, 2001.
29. Tsai, W.; Tu, Y.; Shao, W.; Ebner, E. Testing extensible design patterns in object-oriented frameworks through scenario templates. In: Twenty-Third International Computer Software and Applications Conference, 1999.
30. Turk, D.; France, R.; Rumpe, B. Limitations of agile software processes. In: Third International Conference on Extreme Programming and Agile Processes in Software Engineering (XP'2002), Alghero, Sardinia, Italy, 2002, p. 43–46.

Apêndice I

Tabela 9: Documentação parcial dos requisitos de teste da classe “Locação de Recurso” do Padrão 4 da GRN

Nr. Requisito de Teste	Tipo	Padrão da Linguagem de Padrões	Classe do Padrão	Operação	Tabela correspondente no SGBD	Classes de Equivalência utilizadas	Especificação do Requisito de Teste	Condições Válidas	Validações Anteriores	Obs.	Retorno Esperado
RT851	Integridade e Consistência	Padrão 4: Locar o Recurso	Locação de Recurso	Inserção	Resource Rental	3, 5, 10, 16	número inteiro, entre 1 e 2147483647, preenchido, não cadastrado	número é valor inteiro (entre 0 e 2147483648), não vazio, não cadastrado	–	–	sucesso na verificação da integridade e da consistência, permitindo continuar com a operação
RT853	Consistência	Padrão 4: Locar o Recurso	Locação de Recurso	Inserção	Resource Rental	63	número não numérico	número numérico	–	–	falha na verificação da consistência, não permitindo continuar com a operação
RT1254	do Negócio	Padrão 4: Locar o Recurso	Locação de Recurso	Inserção	Resource Rental	15, 57	código da instância cadastrado e instância disponível, ou seja, não retirada para locação	código da instância cadastrado e instância disponível	número da locação e código da instância válidos (consistidos)	–	sucesso na verificação da funcionalidade do negócio, permitindo continuar com a operação
…	…	…	…	…	…	…	…	…	…	…	…