UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

UM ESTUDO DE CASO USANDO O MÉTODO FORMAL Z PARA ESPECIFICAÇÃO DE UM SISTEMA

BRUNO BIRIBIO WOERNER

BLUMENAU
2009

2009/1-03

BRUNO BIRIBIO WOERNER

UM ESTUDO DE CASO USANDO O MÉTODO FORMAL Z PARA ESPECIFICAÇÃO DE UM SISTEMA

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. José Roque Voltolini da Silva – Orientador

BLUMENAU
2009

2009/1-03

UM ESTUDO DE CASO USANDO O MÉTODO FORMAL Z PARA ESPECIFICAÇÃO DE UM SISTEMA

Por

BRUNO BIRIBIO WOERNER

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

	_____
Presidente:	Prof. José Roque Voltolini da Silva – Orientador, FURB
	_____
Membro:	Prof. Joyce Martins – FURB
	_____
Membro:	Prof. Everaldo Artur Grahl – FURB

Blumenau, 8 de julho de 2009

Dedico este trabalho a pessoa que esteve este todo o tempo a meu lado, minha noiva, Anne.

AGRADECIMENTOS

A Deus e a Cristo por sempre terem estado comigo e especialmente terem me carregado durante este trabalho.

À minha noiva, Anne Caroline Varela, por ter acreditado, dado apoio, atenção e incentivo durante todo o trabalho, e pelo carinho e amor.

À minha família, que possibilitou minha dedicação total a este trabalho.

Aos meus amigos, pelo interesse, pelo ouvir e pelas críticas.

Ao meu orientador, José Roque Voltolini da Silva, pela cobrança e por ter acreditado na conclusão deste trabalho.

O problema neste mundo, é que os idiotas têm imensas certezas e as pessoas sensatas imensas dúvidas.

Bertrand Russell

RESUMO

Este trabalho apresenta uma especificação formal de um sistema utilizando a notação Z. A especificação de um sistema de biblioteca é apresentada. O sistema possui as operações básicas *Create*, *Retrieve*, *Update*, *Delete* (CRUD), modeladas em Z. Para cada símbolo e construção utilizada na especificação, é apresentado o mapeamento para a respectiva implementação do software. A linguagem de programação utilizada para a implementação do software foi o Java. Foram ainda implementados testes utilizando a especificação construída. Os testes foram implementados utilizando a biblioteca JUnit.

Palavras-chave: Especificação formal. Notação Z. Sistema de biblioteca.

ABSTRACT

This work presents a formal specification of a system using the Z notation. The specification of a library system is presented. The system has the basis operations Create, Retrieve, Update, Delete (CRUD), modeled with Z. For each symbol and construction used in a specification, a mapping to software implementation is showed. The programming language used to software implementation is Java. Still test are implemented using the specification builded. The tests are implemented using JUnit library.

Key-words: Formal Specification. Z notation. Library system.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Moura (2001, p. 25-27) descreve o uso de métodos formais como meio de evitar falhas em módulos de software, além de proporcionar auxílio no processo de desenvolvimento. Atualmente a tecnologia de projeto e produção de hardware possui um elevado grau de sofisticação, resultando em um número muito baixo de falhas nestes componentes. Há uma rigorosidade muito maior na produção de hardware com relação a sua especificação. Nos sistemas computacionais o limitador de progresso não está atrelado ao hardware, o software é o maior responsável por este título. Erros e comportamentos indesejados no software servem de embasamento para esta afirmação. O método formal por sua vez objetiva evitar ao máximo que falhas ocorram, permitindo que o comportamento de um programa possa ser conhecido e analisado sem que haja a necessidade de executar seu código fonte.

Davies e Woodcock (1996, p. 1-2) apresentam os métodos atuais de especificação de software como ambíguos, imprecisos e focados em detalhes não importantes ao produto em si. Esta ambigüidade acarreta em descobrir alguns problemas muito tardiamente e muitas vezes inviabiliza o projeto em plena construção. Para resolver este problema é necessário utilizar-se de uma linguagem sem ambigüidade, precisa e com um bom nível de abstração. Estas características remetem à matemática. Os métodos formais são fortemente baseados em toda a ciência matemática.

Conforme Moura (2001, p. 30-31), a notação formal Z é apropriada para este cenário. Baseada fortemente na teoria dos conjuntos demonstra uma maneira precisa e simples de especificar um software. A notação Z não é uma simples tradução de texto para fórmulas matemáticas, mas sim uma maneira de apresentar o comportamento e propriedades do software. Z não é executável, a definição pode ser construída de diversas formas a fim de proporcionar diferentes métodos de construção. Ainda, segundo Spivey (1992, p. 128), a mais complexa das definições realizadas em Z não é nada mais que teoria matemática organizada de uma maneira estruturada.

Potter, Sinclair e Till (1996, p. 309) explicam que o ganho em precisão na especificação utilizando Z proporciona uma redução considerável na quantidade de erros no processo de produção do software, que provavelmente seriam descobertos em fases futuras do processo de desenvolvimento ou de implantação. Isso é facilmente evidenciado em virtude do desenvolvedor possuir o prévio conhecimento do software e do que necessita ser testado.

A partir do exposto, são apresentadas neste trabalho a especificação e a implementação

de um estudo de caso utilizando a notação Z, detalhando cada passo, objetivando demonstrar a validade e consistência dos requisitos de um sistema. Ainda, segundo Moura (2001, p. 32) pode-se "agregar à especificação técnicas para prova de propriedades acerca do objeto especificado, caminhando na direção de um cálculo de programas." Também será disponibilizado um roteiro de uso enfocando um sistema desde a sua especificação até a implementação.

O sistema de biblioteca foi a aplicação escolhida para ser desenvolvida utilizando a notação Z. Hall (1990, p. 15) afirma que métodos formais podem ser aplicados em qualquer tipo de software.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é especificar e implementar um sistema utilizando o método formal Z.

Os objetivos específicos do trabalho são:

a) desenvolver um sistema de biblioteca;
b) realizar a implementação a partir da especificação definida através do método formal;
c) disponibilizar um roteiro para a construção de um software utilizando o método formal Z, desde a especificação até a implementação na linguagem Java.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está dividido em quatro capítulos. O segundo capítulo apresenta os assuntos estudados durante o desenvolvimento do trabalho. Nele, de forma geral, são discutidos os aspectos como processo de desenvolvimento de sistema, especificação formal, notação Z e mapeamento da especificação formal para código Java.

O terceiro capítulo apresentada o desenvolvimento do trabalho, iniciando pelos requisitos que o sistema de biblioteca deve atender e seguido pela especificação formal em Z do mesmo. Também é mostrado o mapeamento das construções em Z para o código Java

implementado. Em seguida são discutidos os resultados do desenvolvimento.

Por fim, o quarto capítulo apresenta as conclusões do trabalho e sugestões para extensões.

2 FUNDAMENTAÇÃO TEÓRICA

A especificação de um sistema em Z requer o conhecimento de vários conceitos e métodos a respeito deste assunto. Evidencia-se a necessidade de introduzir estes conceitos ao leitor antes de realizar uma análise mais profunda no estudo de caso em si. Portanto, este capítulo visa elucidar os conceitos, metodologias e ferramentas utilizadas no desenvolvimento do trabalho, sendo eles: processo de desenvolvimento, especificação formal, notação Z, mapeamento da especificação formal para código Java e a ferramenta de especificação Z/EVES. Na última seção são apresentados alguns trabalhos correlatos.

2.1 PROCESSO DE DESENVOLVIMENTO

Potter, Sinclair e Till (1996, p. 5-6) afirmam que o ciclo de vida de um software ou hardware geralmente é identificado por inicialmente conceber uma idéia, seguir o desenvolvimento do produto, realizar as vendas, atingir sua maturidade e podendo futuramente torna-se obsoleto. O ciclo de vida, pensado de uma maneira organizada, é definido como um processo. A cada momento nascem e se adaptam processos para suprir a necessidade de melhorar a qualidade final e proporcionar clareza e agilidade no processo de construção como um todo. De uma maneira geral, as atividades contempladas no processo de desenvolvimento de software são:

a) análise de requisitos: o cliente e o fornecedor do software trabalham em conjunto para entender os problemas existentes e definir uma solução para o mesmo. Neste ponto o cliente expõe o que espera que o software realize;
b) especificação do sistema: os requisitos a serem atendidos tomam forma. Pontos como *performance,* confiabilidade e usabilidade, por exemplo, começam a ser definidos neste momento. A utilização de requisitos funcionais e não funcionais se torna mais evidente nesta parte do processo;
c) projeto ou *design*: é a ponte entre o que se planeja fazer para o como fazer. Nesta fase que são definidos quais os meios a serem utilizados para contemplar a especificação. O trabalho em arquitetura é bastante evidente nesta parte. Os módulos do sistema também começam a ganhar forma;

d) implementação: a partir do projeto definido ocorre a implementação, gerando um programa executável. Geralmente é conhecida também como etapa de codificação ou programação;
e) testes e integração: o sistema é testado a fim de validar a implementação com o proposto, além de efetuar verificações de integração com os outros softwares, garantido a sua funcionalidade como um todo;
f) suporte e otimizações: ao entregar o software ao cliente, o mesmo sofre correções de eventuais erros encontrados e alterações devido a necessidades específicas de cada usuário.

Ince (1992, p. 7) apresenta o processo de desenvolvimento de software problemático em virtude da natureza das notações utilizadas em cada parte do processo de desenvolvimento de software. Utiliza-se da linguagem natural para descrever os requisitos do sistema, notação gráfica pesadamente anotada com linguagem natural para projetar o sistema e enviá-lo a fase de programação. Como a linguagem natural não é precisa, cada desenvolvedor possui uma forma de se expressar, construindo sua própria notação, o que causa um problema de comunicação entre as fases do ciclo de desenvolvimento de software.

2.1.1 Problemas no processo de desenvolvimento

Segundo Ince (1992, p. 10), o problema já inicia no processo de levantamento de requisitos, tanto nos requisitos funcionais, como requisitos não funcionais. Alguns dos problemas relatados são:

a) imprecisão: um requisito escrito pode ter tamanho consideravelmente grande, porém mesmo assim é quase impossível contemplar um grande nível de precisão na escrita. No pior dos casos pode-se obter um requisito como: a interface utilizada pelo operador de radar deve ser amigável. Este exemplo contém uma abstração muito grande, por conseqüência não é de muita utilidade no decorrer do processo de desenvolvimento;
b) contradição: requisitos funcionais e não funcionais não estão agrupados em um mesmo universo na técnica de construção de requisitos. Algumas contradições podem ser encontradas ao analisar alguns requisitos contra outros. Um exemplo de um requisito neste molde é: o nível de água dos últimos três meses deve ser armazenado em fita magnética. O outro requisito é definido como: o comando

`imprimir_nivel` imprime a média de água durante os últimos 3 meses. O sistema não pode demorar mais de três segundos para retornar a pesquisa. Obviamente acessar uma fita magnética irá consumir facilmente estes 3 segundos;

c) imperfeito: a imperfeição na construção dos requisito é um dos maiores fatores de erro na entrega de um software. Considerando um requisito como: a função `obter_media` imprime a temperatura média do reator em determinado dia. O que aconteceria caso fosse informado uma data para a qual não haveria temperatura mensurada naquele dia? Deveria acontecer um erro? O que imprimiria na tela?;
d) requisitos misturados: em vários requisitos pode-se identificar a mescla entre requisitos funcionais, não funcionais, dados e estruturas. Funcionalidades do sistema e estruturas de dados podem estar tão misturadas dentro da aplicação que pode parecer extremamente confuso identificar o comportamento do software;
e) ambigüidade: requisitos escritos na linguagem natural podem em muitos casos conter ambigüidades. A linguagem natural é um meio ideal para fazer romances ou poesias, aonde o leitor interpreta o que o autor estava sentido no momento em que escreveu aquela frase. Entretanto, nem sempre é um dos melhores meios para especificar um sistema computacional, a linguagem não é precisa. Um exemplo seria: a identidade do operador do sistema deve consistir do nome e senha. A senha deve conter seis dígitos. Isso deve ser inserido na tela de *logon* do sistema. Ao utilizar a palavra "isso" pode gerar diferentes interpretações, seria a senha que deveria ser inserida, ou o nome na tela de *logon*?;
f) diferentes níveis de abstração: um nível de detalhe diferente entre vários requisitos pode causar confusão. Um primeiro nível de abstração seria: o sistema deve gerar relatórios para o gerenciamento das mercadorias transportadas entre os armazéns. Outro exemplo em um nível diferente de abstração tem-se como exemplo: o sistema deve permitir que o gerente acesse o relatório de movimentações em um determinado armazém em um específico dia. As mercadorias podem ser identificadas em categorias que estão definidas em outra seção. Este certo nível de abstração acaba por gerar dependências entre requisitos. Os documentos passam a necessitar de organização estrutural, conexões internas, parágrafos e subparágrafos. Isto tudo torna a manutenção mais complicada e o repasse de trabalho fica tão ou mais complicado quanto à própria estrutura do documento.

De acordo com Ince (1992, p. 10), a utilização dos métodos formais pode ser aplicada em qualquer parte do processo de desenvolvimento de software. Trocar as técnicas de

especificação já utilizadas em cada processo pode parecer algo trabalhoso, porém muitas técnicas de validação e verificação tornam-se muito mais apropriadas, práticas, simples e precisas utilizando a notação formal.

2.2 ESPECIFICAÇÃO FORMAL

Moura (2001, p. 24-25) apresenta a especificação formal como uma busca da necessidade existente na computação em desenvolver softwares precisos e concisos; de fácil manipulação e entendimento; algo que se adapte ao processo de desenvolvimento de software. Algumas áreas já possuem este mecanismo incorporado, como é o exemplo da arquitetura e engenharia. As razões para este fato parte do princípio que as estruturas utilizadas pela engenharia utilizam a matemática juntamente com a física, há séculos. Não é surpresa que esta parte da matemática esteja tão bem adaptada ao cálculo de estruturas.

Contemplando a área de computação, os métodos matemáticos utilizados, a matemática discreta e a lógica matemática, estavam em seu estado inicial no início do século XX. Aplicações diretas do uso destes métodos não chegam a 50 anos. Neste ponto o que não é surpresa é que haja dificuldade em se utilizar das especificações formais. Realizar uma descrição rigorosa das funcionalidades de um software como um todo é, em geral, uma tarefa não trivial (MOURA, 2001, p. 25).

Segundo Hall (1990, p. 13), as especificações formais além de eliminarem grandes quantidades de erros, possibilitam uma melhor identificação de um eventual erro. Com a metodologia de especificação informal, um possível erro pode ser interpretado de maneira divergente entre diferentes pessoas. O senso crítico geralmente é utilizado para resolver se determinado comportamento está correto. Com a especificação formal identificar um erro é muito mais rápido e todos podem concordar que determinado comportamento é um erro.

Spivey (1992, p. 2) afirma que a especificação formal por possuir uma boa abstração e clareza do comportamento do software, além de não estar atrelada ao código fonte, proporciona uma maior interação com o cliente ainda na fase de especificação. Afora o tempo ganho em descobrir novas necessidades do cliente, a compreensão do sistema como um todo torna-se mais fácil.

2.3 NOTAÇÃO Z

Segundo Davies e Woodcock (1996, p. 2), um dos maiores casos de sucessos da utilização de Z foi a realizada no projeto *Customer Information Control System* (CICS) da *International Business Machines* (IBM). Por volta de 1980 alguns módulos do CICS foram refeitos utilizando a linguagem Z. A linguagem Z mostrou-se de fácil aprendizado e simples uso, mesmo para programadores sem prévia experiência com matemática, além de aumentar a qualidade e confiabilidade do código produzido.

O primeiro produto construído com Z foi o CICS/*Enterprise Systems Architecture* (ESA) *version* 3, disponível em 1989. A partir do sucesso obtido, em abril de 1992, o *Queen's Award for Technological Achievement* foi entregue a IBM United Kingdom Laboratories Limited e a Oxford University Computing Laboratory, segundo o título "desenvolvimento e uso de método de programação avançada que reduz o custo de desenvolvimento e aumenta significativamente a qualidade e confiabilidade", denominado Z (DAVIES; WOODCOCK, 1996, p. 3).

Potter, Sinclair e Till (1996, p. 309) explicam que uma especificação escrita em Z é uma mistura do formal com o informal. A formalidade está presente nas fórmulas matemáticas e a informalidade é encontrada nos textos informais. Ambos são importantes: a especificação formal fornece uma descrição precisa do software especificado, enquanto os textos informais proporcionam uma melhor compreensão da especificação realizada, aproximando a notação matemática do mundo real.

Spivey (1992, p. 2) afirma que Z possui um mecanismo denominado *schema*. *Schemas* em Z são decomposições das especificações em partes menores. Tendo esta divisão realizada, é possível ter um sistema definido peça a peça. Cada peça pode ser ligada com um comentário que explica informalmente o comportamento matemático. Em Z os *schemas* são utilizados para definir aspectos estáticos e dinâmicos de um sistema.

Os aspectos estáticos dividem-se em duas outras categorias (SPIVEY, 1992, p. 2):

a) o estado que cada *schema* pode representar;
b) o comportamento do sistema ao ocorrer a transição de um estado para outro estado.

Os aspectos dinâmicos dividem-se em três outras categorias (SPIVEY, 1992, p. 2):

a) as operações possíveis a serem realizadas;
b) a relação entre suas entradas e suas saídas;
c) as mudanças de estados que acontecem.

Davies e Woodcock (1996, p. 3) apresentam como característica do Z a utilização de tipos para cada objeto definido. Isso auxilia no processo de desenvolvimento da especificação, tanto que existem ferramentas que realizam a checagem de tipo em Z. Outro fator de grande relevância em Z é a questão do refinamento. A especificação realizada deve ser refinada até que o código fonte esteja pronto. Ainda deve ser observado que o refinamento deve estar o mais próximo possível do código executável.

Segundo Barden, Cooper e Stepney (1994, p. 5), mesmo Z sendo uma linguagem formal (baseada na matemática) não há garantias da não existência de erros no software. Erros podem ocorrer em diversas fases do desenvolvimento. A fase de levantamento de requisitos é uma delas. Não entender o problema do cliente é um dos erros mais graves e custosos. Entretanto, a utilização de Z na necessidade de especificar formalmente força o cliente a esclarecer melhor os requisitos, até mesmos um requisito considerado um pouco vago é mais facilmente identificado.

A Internation Organization for Standardization (2002, p. vi) define que uma especificação do sistema deve ajudar a entender o sistema, ajudando na manutenção e desenvolvimento. Deve conter apenas propriedades abstratas, não se atendo a detalhes de implementação ou algum algoritmo específico. Deve ser livre ao ponto de permitir ser refinada para diferentes tipos de implementação. Uma especificação nestes moldes pode ser escrita na notação Z.

Z utiliza notação matemática. Consequentemente uma especificação escrita em Z herda as propriedades da matemática. A formalidade que existe na matemática também é incorporada em Z. Outro ponto é que as provas de propriedades de um sistema são realizadas de modo racional, sendo consequência da especificação.

A notação Z é caracterizada pelo seu extenso ferramental matemático, tratamento de tipos e utilização de *schema* para construção da estrutura da especificação.

Exemplos de tipos de especificações realizadas em Z incluem:

a) sistemas não tolerante a falhas, como sinalização férrea, aparelhos médicos, sistemas de energia nuclear;
b) sistemas seguros, como sistemas de transação bancária, comunicação;
c) sistemas em geral, como linguagem de programação, processadores de ponto flutuante;
d) formalização de semânticas de outras linguagens, especialmente em documentos de padrões.

Para o estudo da linguagem Z ser mais produtivo, na próxima seção são apresentados

alguns símbolos, definições e construções utilizados na linguagem.

2.3.1 Símbolos

Nas sub-seções a seguir é apresentado um apanhado dos símbolos utilizados ao construir uma especificação em Z.

2.3.1.1 Símbolos básicos

No Quadro 1 são apresentados os símbolos dos operadores lógicos em Z. No Quadro 2 são apresentados os símbolos dos operadores relacionais em Z. No Quadro 3 são apresentados os símbolos dos operadores sobre conjuntos em Z.

Símbolo	Descrição
$\wedge$	E
$\vee$	Ou
$\Rightarrow$	Se
$\Leftrightarrow$	Se e somente
$\neg$	Não

Fonte: adaptado de International Organization for Standardization (2002, p. 22).

Quadro 1- Símbolos dos operadores lógicos em Z

Símbolo	Descrição
$=$	Igualdade / Atribuição
$\neq$	Negação de igualdade
$\leqslant$	Menor igual
$<$	Menor
$\geqslant$	Maior igual
$>$	Maior

Fonte: adaptado de International Organization for Standardization (2002, p. 22).

Quadro 2- Símbolos dos operadores relacionais em Z

Símbolo	Descrição
∀	Para todo
∃	Existe (pelo menos um)
∈	Pertence
∉	Não pertence
⊆	Subconjunto ou igual
⊂	Subconjunto
∪	União
∩	Intersecção

Fonte: adaptado de International Organization for Standardization (2002, p. 22-23).

Quadro 3- Símbolos dos operadores de conjunto em Z

2.3.1.2 Definições de construções

Para utilizar construções em Z é necessário ter conhecimento de algumas das definições dos símbolos utilizados. Estes símbolos também são utilizados nas definições axiomáticas e declarações de *schemas.*

No Quadro 4 é apresentada a definição do símbolo de representação de inteiros em Z. No Quadro 5 é apresentada a definição do símbolo de representação de naturais em Z. No Quadro 6 é apresentada a definição do símbolo de abreviação em Z. No Quadro 7 é apresentada a definição do símbolo de abreviação livre em Z. No Quadro 8 é apresentada a definição do símbolo de separador declarativo em Z. No Quadro 9 é apresentada a definição dos símbolos de delimitadores declarativos em Z. No Quadro 10 é apresentada a definição do símbolo de declaração de tipos em Z. No Quadro 11 é apresentada a definição de separador em Z.

Símbolo	Nome	Função/Definição	Exemplo
$\mathbb{Z}$	Inteiros	Representação dos números inteiros.	$\{x: \mathbb{Z}\}$

Fonte: adaptado de International Organization for Standardization (2002, p. 24).

Quadro 4 - Definição de inteiros em Z

Símbolo	Nome	Função/Definição	Exemplo
$\mathbb{N}$	Naturais	Representação dos números naturais. Os números naturais estão compreendidos pelos números inteiros maiores ou igual a zero.	$\mathbb{N} == \{x : \mathbb{Z} \mid x \geqslant 0\}$

Fonte: adaptado de International Organization for Standardization (2002, p. 21).

Quadro 5 - Definição de naturais em Z

Símbolo	Nome	Função/Definição	Exemplo
==	Abreviação	Denota um novo símbolo para o termo. Como já utilizado na definição de ℕ, o mesmo pode ser realizado para outras situações.	RGB=={Red, Green, Blue}

Fonte: adaptado de International Organization for Standardization (2002, p. 22).

Quadro 6 - Definição de abreviação em Z

Símbolo	Nome	Função/Definição	Exemplo
::=	Abreviação livre	Denota um novo símbolo para o termo. Os elementos presentes na declaração são livres, não necessitando estar previamente especificados.	Result ::= Error | Fail | Success

Fonte: adaptado de International Organization for Standardization (2002, p. 22).

Quadro 7 - Definição de abreviação livre em Z

Símbolo	Nome	Função/Definição	Exemplo
|	Separador declarativo	Separa a parte declarativa da parte predicativa. A parte anterior é a parte declarativa e a parte posterior a parte predicativa.	{declarativa | predicativa} {x : ℤ | x ⩾ 0}

Fonte: adaptado de International Organization for Standardization (2002, p. 22).

Quadro 8 - Definição de separador declarativo em Z

Símbolo	Nome	Função/Definição	Exemplo
{ }	Delimitadores declarativos	Delimitadores nos quais estão contidas as partes predicativas e declarativas da função definida.	{declarativa | predicativa} {x : ℤ | x ⩾ 0}

Fonte: adaptado de International Organization for Standardization (2002, p. 21).

Quadro 9 - Definição de delimitadores declarativos em Z

Símbolo	Nome	Função/Definição	Exemplo
[ ]	Declaração de tipo	Declaração de tipos em Z. Para quesito de organização e consistência da notação, a utilização de tipos é extremamente pertinente.	[CLIENT, ADDRESS]

Fonte: adaptado de International Organization for Standardization (2002, p. 21).

Quadro 10 - Definição de declaração de tipos em Z

Símbolo	Nome	Função/Definição	Exemplo
•	Separador	Fornecer mecanismo que utilize dos valores da parte declarativa em uma nova expressão. Sendo a função {a : b | c • d }, pode-se definir como: conjunto das expressões `d` que `a` é do tipo `b` e satisfaz as condições em `c`.	{ x : CLIENT | x is blond • phone(x) }

Fonte: adaptado de Martini (2008, p. 7).

Quadro 11 - Definição de separador em Z

2.3.1.3 Definições e operações em conjuntos

Algumas das construções mais relevantes em Z são apresentadas a seguir, sendo elas as operações sobre conjuntos.

No Quadro 12 é apresentada a definição do símbolo de conjunto vazio em Z. No Quadro 13 é apresentada a definição do símbolo de conjunto potência em Z. No Quadro 14 é apresentada a definição do símbolo de conjunto finito em Z. No Quadro 15 é apresentada a definição do símbolo de união entre conjuntos em Z. No Quadro 16 é apresentada a definição do símbolo de remoção de conjunto em Z. No Quadro 17 é apresentada a definição do símbolo de contador em Z. No Quadro 18 é apresentada a definição do símbolo de mapeamento em Z. No Quadro 19 é apresentada a definição do símbolo de produto cartesiano em Z. No Quadro 20 é apresentada a definição do símbolo de relação em Z. No Quadro 21 é apresentada a definição do símbolo de domínio em Z. No Quadro 22 é apresentada a definição do símbolo de imagem em Z. No Quadro 23 é apresentada a definição do símbolo de restrição de domínio em Z. No Quadro 24 é apresentada a definição do símbolo de restrição de imagem em Z. No Quadro 25 é apresentada a definição do símbolo de subtração de domínio em Z. No Quadro 26 é apresentada a definição do símbolo de subtração de imagem em Z.

Símbolo	Nome	Função/Definição	Exemplo
$\varnothing$	Conjunto vazio	Representação de conjunto vazio.	$x = \varnothing$

Fonte: adaptado de International Organization for Standardization (2002, p. 23).

Quadro 12 - Definição de conjunto vazio em Z

Símbolo	Nome	Função/Definição	Exemplo
$\mathbb{P}$	Conjunto potência	Representa um conjunto com todos os subconjuntos de x.	Se $x = \{a,b\}$ $\mathbb{P}x = \{\varnothing, \{a\}, \{b\}, \{a,b\}\}$

Fonte: adaptado de Martini (2008, p. 8).

Quadro 13 - Definição de conjunto potência em Z

Símbolo	Nome	Função/Definição	Exemplo
$\mathbb{F}$	Conjunto finito	Representa um conjunto finito de elementos de um tipo x.	Se $x = \mathbb{N}$ $\{ x : \mathbb{F}\mathbb{N} \mid x \geqslant 1 \land x \leqslant 10\}$

Fonte: adaptado de International Organization for Standardization (2002, p. 21).

Quadro 14 - Definição de conjunto finito em Z

Símbolo	Nome	Função/Definição	Exemplo
$\cup$	União	Realiza a união entre conjuntos. Quando uma nova entrada necessita ser adiciona ao conjunto, é realizada utilizando o símbolo de união de conjunto.	Se x = $\mathbb{FN}$ x = x $\cup$ {1,2,3}

Fonte: adaptado de International Organization for Standardization (2002, p. 23).

Quadro 15 - Definição de união entre conjuntos em Z

Símbolo	Nome	Função/Definição	Exemplo
\	Remoção de conjunto	Remove conjunto de entradas y sobre um conjunto x. Quando uma entrada necessita ser removida do conjunto é realizada utilizando o símbolo de exclusão de conjunto.	Se x = $\mathbb{FN}$ x = x \ {1,2,3}

Fonte: adaptado de International Organization for Standardization (2002, p. 23).

Quadro 16 - Definição de remoção de conjunto em Z

Símbolo	Nome	Função/Definição	Exemplo
#	Contador	Obtém o tamanho de um determinado conjunto.	#{1,2,3} : como resultado, é apresentado 3 (três).

Fonte: adaptado de International Organization for Standardization (2002, p. 24).

Quadro 17 - Definição de contador em Z

Símbolo	Nome	Função/Definição	Exemplo
$\mapsto$	Mapeamento	Relaciona um elemento x a outro elemento y. Possui o mesmo comportamento de relação existente em um par ordenado (x, y).	x $\mapsto$ y

Fonte: adaptado de International Organization for Standardization (2002, p. 23).

Quadro 18 - Definição de mapeamento em Z

Símbolo	Nome	Função/Definição	Exemplo

$\times$	Produto Cartesiano	Representação de um produto cartesiano entre dois conjuntos.	Singer=={Waters, Gilmour}
			Music=={Dogs, Mother, Money}
			Singer $\times$ Music==
			{
			{Waters,Dogs},{Gilmour,Dogs},
			{Waters,Mother},{Gilmour,Mother},
			{Waters,Money},{Gilmour Money}
			}

Fonte: adaptado de Martini (2008, p. 9).

Quadro 19 - Definição de produto cartesiano em Z

Símbolo	Nome	Função/Definição	Exemplo
$\leftrightarrow$	Relação	Conjunto de todas as relações entre x e y.	A relação é uma abreviação de: $X \leftrightarrow Y == \mathbb{P}\ \{ X \times Y \}$ Se X = {a,b} e Y = {1,2} $X \leftrightarrow Y == \{\emptyset,\ \{a,1\},\ \{a,2\},\{b,1\},\ \{b,2\},$ {{a,1},{a,2}}, {{a,1},{b,1}}, {{a,1},{b,2}}, {{a,2},{b,1}}, {{a,2},{b,2}}, {{b,1},{b,2}}, ...}

Fonte: adaptado de Martini (2008, p. 11).

Quadro 20 - Definição de relação em Z

Símbolo	Nome	Função/Definição	Exemplo
Dom	Domínio	Domínio de uma função. O domínio de uma função sendo $X \leftrightarrow Y$ é denotado pelos elementos em X associados aos elementos em Y.	dom R = $\{ x : X; y : Y \mid x \mapsto y \in R \bullet x \}$ Se x = {{1↦A},{2↦B},{3↦C}} dom(x) = {1,2,3}

Fonte: adaptado de Martini (2008, p. 12).

Quadro 21 - Definição de domínio em Z

Símbolo	Nome	Função/Definição	Exemplo
Ran	Imagem	Imagem de uma função. A imagem de uma função sendo $X \leftrightarrow Y$ é denotado pelos elementos em Y associados ao elementos em X.	ran R = $\{ x : X; y : Y \mid x \mapsto y \in R \bullet y \}$ Se x = {{1↦A},{2↦B},{3↦C}} ran(x) = {A,B,C}

Fonte: adaptado de Martini (2008, p. 12).

Quadro 22 - Definição de imagem em Z

Símbolo	Nome	Função/Definição	Exemplo
$\lhd$	Restrição de domínio	Se R: $X \leftrightarrow Y$ e $A \subseteq X$ então $A \lhd X$ denota a restrição de domínio de R para A. Funcional para quando há interesse em obter apenas uma parte da relação com o filtro no domínio.	$\{ x : X; y : Y \mid$ $x \mapsto y \in R \land x \in A \bullet x \mapsto y \}$ Se x = {{1↦A},{2↦B},{3↦C}} $\{1\} \lhd x$ = {{1↦A}}

Fonte: adaptado de Martini (2008, p. 13).

Quadro 23 - Definição de restrição de domínio em Z

Símbolo	Nome	Função/Definição	Exemplo
$\rhd$	Restrição de imagem	Se R: $X \leftrightarrow Y$ e $B \subseteq Y$ então $R \rhd B$ denota a restrição de imagem de R para B. Funcional para quando há interesse em obter apenas uma parte da relação com o filtro na imagem.	$\{ x : X; y : Y \mid x \mapsto y \in R \land y \in B \bullet x \mapsto y \}$ Se x = {{1↦A},{2↦B},{3↦C}} $x \rhd \{A\} = \{\{1 \mapsto A\}\}$

Fonte: adaptado de Martini (2008, p. 13).

Quadro 24 - Definição de restrição de imagem em Z

Símbolo	Nome	Função/Definição	Exemplo
$\lhd$	Subtração de domínio	Para excluir um subconjunto A do domínio X de uma relação $R : X \leftrightarrow Y$, considera-se a restrição de domínio: $(X \setminus A) \lhd R$. Por ser uma operação muito comum é definido símbolo para a subtração de domínio.	$A \lhd R = \{x : X; y : Y \mid x \mapsto y \in R \land x \notin A \bullet x \mapsto y\}$ Se x = {{1↦A},{2↦B},{3↦C}} $\{1\} \lhd x = \{\{2 \mapsto B\},\{3 \mapsto C\}\}$

Fonte: adaptado de Martini (2008, p. 14).

Quadro 25 - Definição de subtração de domínio em Z

Símbolo	Nome	Função/Definição	Exemplo
$\rhd$	Subtração de imagem	Para excluir um subconjunto A da imagem Y de uma relação $R : X \leftrightarrow Y$, considera-se a restrição de imagem: $R \rhd (Y \setminus A)$. Por ser uma operação muito comum é definido um símbolo para a subtração de imagem.	$R \rhd B = \{x : X; y : Y \mid x \mapsto y \in R \land y \notin B \bullet x \mapsto y\}$ Se x = {{1↦A},{2↦B},{3↦C}} $x \rhd \{A\} = \{\{2 \mapsto B\},\{3 \mapsto C\}\}$

Fonte: adaptado de Martini (2008, p. 14).

Quadro 26 - Definição de subtração de imagem em Z

2.3.1.4 Declaração de funções

A definição de função se dá por: se cada objeto de um conjunto está relacionado a no máximo um objeto de um outro, então a relação entre estes dois conjuntos é chamada de função.

No Quadro 27 são apresentados os tipos de funções em Z. No Quadro 28 é apresentada a função parcial em Z. No Quadro 29 é apresentada a função total em Z. No Quadro 30 é apresentada a função parcial injetora em Z. No Quadro 31 é apresentada a função total

injetora em Z. No Quadro 32 é apresentada a função parcial sobrejetora em Z. No Quadro 33 é apresentada a função total sobrejetora em Z. No Quadro 34 é apresentada a função total bijetora em Z.

Símbolo	Descrição
$\pfun$	função parcial
$\fun$	função total
$\pinj$	função parcial injetora
$\inj$	função total injetora
$\psurj$	função parcial sobrejetora
$\surj$	função total sobrejetora
$\bij$	função total bijetora

Fonte: adaptado de International Organization for Standardization (2002, p. 23).

Quadro 27 – Tipos de funções em Z

Função	Definição
Função parcial	Considerando a função X $\pfun$ Y é parcial porque nem todos os elementos de X possam estar associados a elementos de Y.

Fonte: adaptado de International Organization for Standardization (2002, p. 23).

Quadro 28 – Função parcial em Z

Função	Definição
Função total	Considerando a função X $\fun$ Y é total porque todos os elementos de X estão associados a elementos de Y.

Fonte: adaptado de International Organization for Standardization (2002, p. 23).

Quadro 29 – Função total em Z

Função	Definição
Função parcial injetora	Considerando a função X $\pinj$ Y é injetora parcial porque os elementos de X que possuam associação estão associados a apenas um elemento distinto em Y. Pode haver elementos em X que não estão associados a elementos de Y.

Fonte: adaptado de International Organization for Standardization (2002, p. 23).

Quadro 30 – Função parcial injetora em Z

Função	Definição
Função total injetora	Considerando a função X $\inj$ Y é injetora total porque todos os elementos de X estão associados a apenas um elemento distinto em Y.

Fonte: adaptado de International Organization for Standardization (2002, p. 23).

Quadro 31 – Função total injetora em Z

Função	Definição
Função parcial sobrejetora	Considerando a função X $\psurj$ Y é sobrejetora parcial porque todos os elementos em Y que possuam associação estão associados a apenas um elemento distinto em X. Pode haver elementos em Y que não estão relacionados com elementos em X.

Fonte: adaptado de International Organization for Standardization (2002, p. 23).

Quadro 32 – Função parcial sobrejetora em Z

Função	Definição
Função total sobrejetora	Considerando a função X $\twoheadrightarrow$ Y é sobrejetora total porque todos os elementos em Y estão associados a um elemento distinto em X.

Fonte: adaptado de International Organization for Standardization (2002, p. 23).

Quadro 33 – Função total sobrejetora em Z

Função	Definição
Função total bijetora	Considerando a função X $\rightarrowtail\!\!\!\!\!\twoheadrightarrow$Y é bijetora total porque todos os elementos de X possuem apenas um elemento distinto associado em Y.

Fonte: adaptado de International Organization for Standardization (2002, p. 23).

Quadro 34 – Função total bijetora em Z

2.3.1.5 Definições axiomáticas

As definições axiomáticas possuem uma parte declarativa e outra predicativa. Acima da linha divisória fica a parte declarativa e abaixo a parte predicativa. As restrições definidas na parte predicativa são identificadas pelos símbolos definidos na parte declarativa. O Quadro 35 apresenta a definição axiomática de uma variável `maxSize` que contenha apenas números inteiros maiores que zero.

$$maxSize: \mathbb{Z}$$
$$\hline$$
$$maxSize > 0$$

Fonte: Davies, Woodcock (1996, p. 78).

Quadro 35 – Definição axiomática em Z

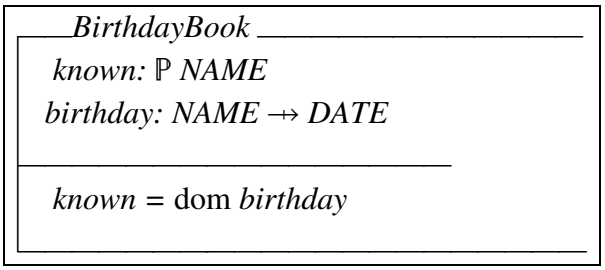
2.3.1.6 *Schema*

Segundo Harry (1996, p. 201), *schemas* são a espinha dorsal de uma especificação em Z. Ajudam a estruturar e modularizar a especificação, possibilitam descrever estados, operações, tipos, predicados e teoremas.

2.3.1.6.1 Estrutura de um *schema*

Spivey (1992, p. 3-4) demonstra um exemplo utilizando a notação formal Z para

especificar uma agenda de aniversários. Para tanto, o nome e a data de aniversário são cadastrados. Em Z inicialmente é necessário definir um *state space* (estado do sistema). Para realizar esta representação será utilizada a definição de *schema*. Um exemplo de *schema* pode ser visto no Quadro 36.



Fonte: Spivey (1992, p. 3).

Quadro 36 - Exemplo de um *schema* especificado em Z

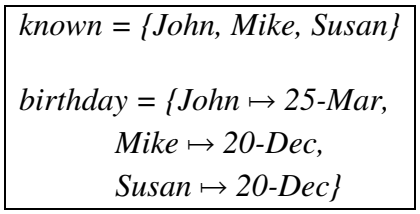
A maioria dos *schemas* possuem este padrão de especificação. Acima da linha divisória central são declaradas algumas variáveis e na parte abaixo da linha divisória central está o relacionamento entre as variáveis. Os elementos presentes no Quadro 36 são:

a) `known` (acima da linha divisória central): conjunto de nomes com aniversários registrados;
b) `birthday`: função que quando aplicada a determinado nome retorna a data de aniversário;
c) `known` (abaixo da linha divisória central): relacionamento definindo que a partir de `birthday` é possível obter `known`. A variável `known` é o domínio da função `birthday`. A função é parcial, o que indica que pode haver nomes sem data de aniversário relacionada.

Alguns pontos foram objetivados com esta especificação formal do *schema*, sendo eles:

a) abstração dos tipos: a partir do Quadro 36 pode-se visualizar que os tipos definidos como `NAME` e `DATE` não tem o formato definido. Inicialmente um valor de qualquer formato pode ser atribuído a estas variáveis. Isso proporciona agilidade no desenvolvimento, sem preocupar-se com estes detalhes iniciais;
b) criação de invariante: ao definir `known` como domínio da função `birthday`, determina-se a criação de uma invariante no sistema. O retorno da função `birthday` é sempre `known`.

O Quadro 37 apresenta um possível estado com os valores de `known` definidos e seus respectivos aniversários associados pela função `birthday`.



Fonte: Spivey (1992, p. 4).

Quadro 37 - Representação do estado de um *schema* em Z

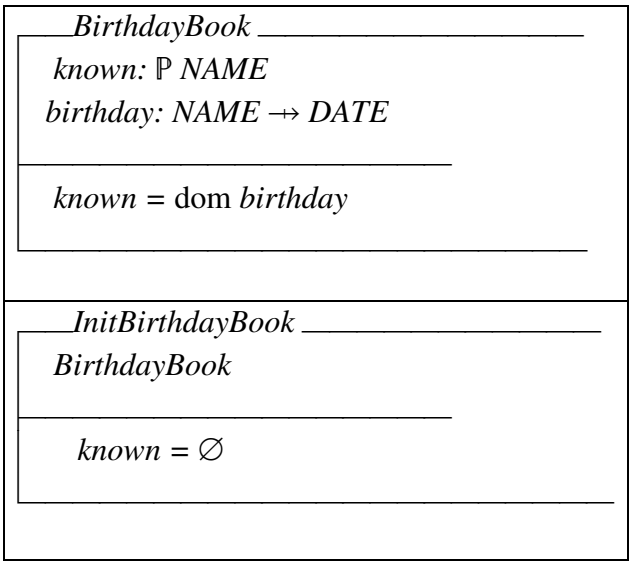
É possível visualizar que nesta especificação não foi definida a quantidade máxima de registros possíveis a ser armazenada no livro de aniversário, tampouco a ordem de armazenamento das entradas definidas. O intuito da representação de um estado do *schema* foi objetivar os seguintes tópicos:

a) validação de invariantes: a invariante definida no Quadro 37 é aqui satisfeita. Os registros em `birthday` possuem todos os nomes presentes em `known`;
b) retorno definido: o retorno da função `birthday` está bem definido. Retorna apenas uma data de aniversário para cada pessoa associada.

2.3.1.6.2 Inclusão de *schemas*

Uma possibilidade para utilização de *schemas* de forma mais produtiva é utilizar a inclusão de *schemas*. Um *schema* previamente definido pode ser incluído em um *schema* definido posteriormente. A parte declarativa e parte predicativa é adicionada no novo *schema* definido.

No Quadro 38 é definido o modelo de como incluir um *schema* em outro. O *schema* `BirthdayBook` é o inicialmente definido. O *schema* `InitBirthdayBook` inclui o *schema* `BirthdayBook`. O conceito de inclusão de *schemas* pode ser relacionado ao conceito de herança em orientação a objetos.

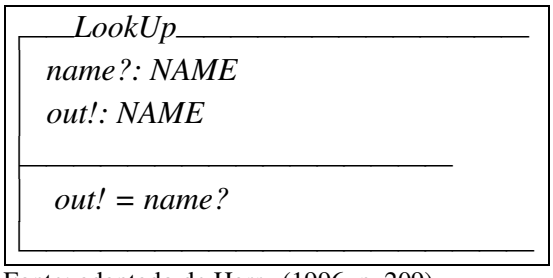


Fonte: Spivey (1992, p. 3, 6).

Quadro 38 - Exemplo de inclusão de *schema* em Z

2.3.1.6.3 Descritores de estado do *schema*

Harry (1996, p. 209) afirma que para realizar operações sobre os *schemas* é necessário especificar como o estado do sistema é afetado. Deve ser possível definir as entradas, o efeito das entradas no *schema* e as suas saídas. As entradas são identificadas adicionando um ponto de interrogação (?) ao nome da variável. As saídas são identificadas adicionando um ponto de exclamação (!) ao nome da variável. No Quadro 39 a variável `name?` é uma entrada e `out!` é uma saída. No exemplo a variável de saída recebe o valor de entrada.



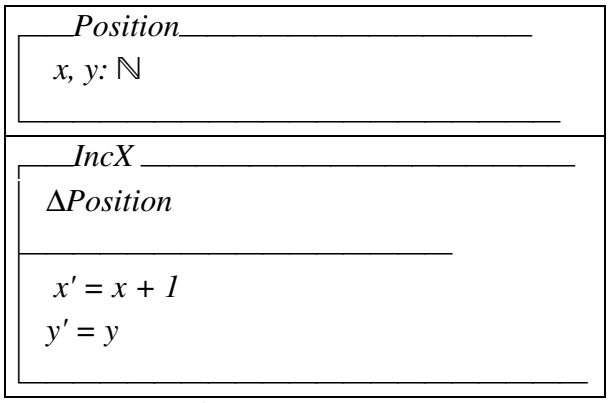
Fonte: adaptado de Harry (1996, p. 209).

Quadro 39 – Símbolo dos decoradores de variáveis de entrada e saída

2.3.1.6.4 Convenções

Harry (1996, p. 210-211) apresenta as operações como ações que modificam o estado das variáveis. Para representar os estados das variáveis são utilizadas as convenções para

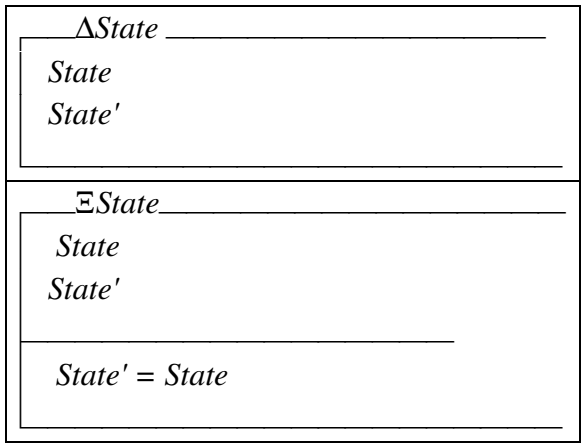
estado anterior e estado posterior. O estado anterior é representado pelo próprio nome da variável. Já o estado posterior é representado pelo nome da variável sufixado com um apóstrofo (’). No Quadro 40 a variável `x` representa o estado anterior e a variável `x'` representa o estado posterior. Outro ponto é que a especificação garante que no *schema* `IncX` a variável y não sofre alteração, o seu estado posterior é o mesmo que o estado anterior.



Fonte: Harry (1996, p. 210).

Quadro 40 – Símbolo de decorador de estado de variáveis

Outra convenção de estado é aplicada à definição de *schemas*. São as definições delta e xi. A convenção delta é representada pela letra grega ‘Δ’ (Quadro 41). Ao incluir a letra grega delta ao nome do *schema* define-se que o estado pode ser alterado. A convenção xi é representada pela letra grega ‘Ξ’ (Quadro 41). Ao incluir a letra grega xi ao nome do *schema* define-se que o estado não pode ser alterado. No exemplo é descrito um *schema* de nome `State`. O decorador apóstrofo (‘) após o nome define que todas as variáveis do *schema* também possuem o decorador. No exemplo existe apenas uma variável de nome `State`. Portanto, o *schema* com as convenções delta e xi possui as variáveis que definem o estado anterior (variável sem apóstrofo) e posterior (variável com apóstrofo).



Fonte: Harry (1996, p. 209).

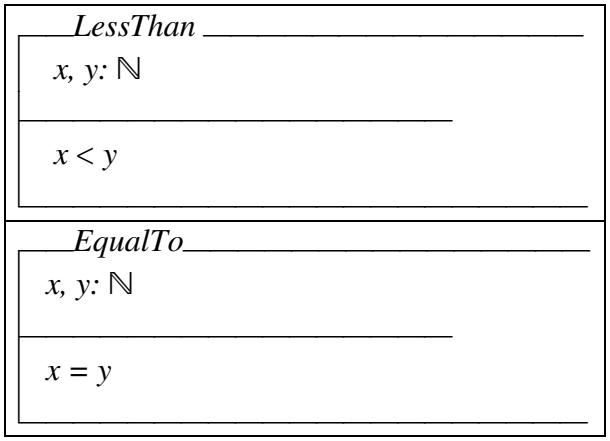
Quadro 41 – Símbolos decoradores de *schema*

2.3.1.6.5 União de *schemas*

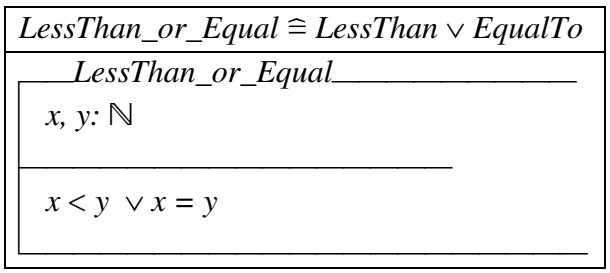
Segundo Harry (1996, p. 215), é possível utilizar os operadores proposicionais lógicos para operar sobre *schemas* previamente definidos, gerando assim novos *schemas*. Não há limite no número de *schemas* a serem unidos. Os operadores podem ser vistos no Quadro 1. Esta utilização pode operar sobre um ou mais *schemas*. A parte inicial do comando é o identificador do novo *schema*, seguido do símbolo de união de *schema* (≙) e por final os demais *schemas* participantes na definição do novo schema.

A parte declarativa deve ser compatível entre os *schemas* que se deseja operar. Variáveis de mesmo nome devem ser de tipos compatíveis. As partes predicativas envolvidas têm suas partes conexas pelo operador definido na união dos *schemas*.

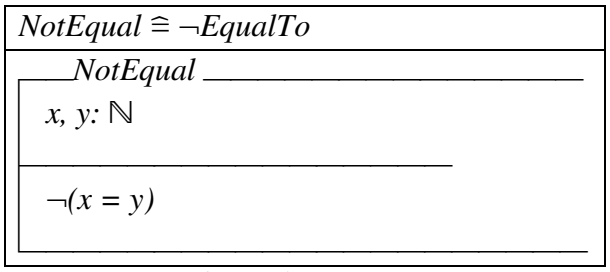
Para exemplificar, no Quadro 42 são apresentados dois *schemas* iniciais. No Quadro 43 é apresentada a disjunção destes *schemas*. No Quadro 44 o *schema* é negado.



Fonte: Harry (1996, p. 215).
Quadro 42 – Definições de *schemas* a serem unidos



Fonte: Harry (1996, p. 215).
Quadro 43 – Disjunção de *schemas*

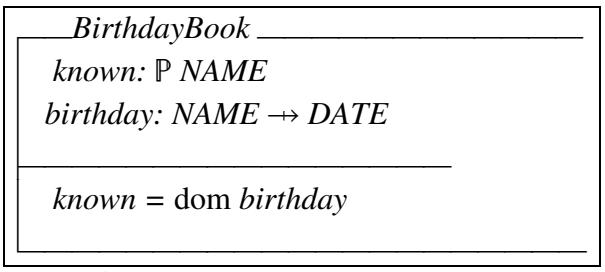


Fonte: Harry (1996, p. 215).

Quadro 44 – Negação de *schema*

2.4 MAPEAMENTO DA NOTAÇÃO Z PARA JAVA

Miyazawa (2008, p. 82) apresenta a ferramenta JZed-Gen que a partir de uma especificação em Z, com algumas parametrizações é possível gerar um código executável em Java. Para exemplificar é apresentada uma conversão utilizando a ferrramenta. É apresentada para conversão a especificação apresentada no Quadro 45.



Fonte: Miyazawa (2008, p. 77).

Quadro 45 - Exemplo de um *schema* a ser traduzido

Segundo Miyazawa (2008, p. 77-78), os conjuntos `NAME` e `DATE` são conjuntos dos quais não se têm informação, por este motivo, devem ser associados a alguma classe. Foi utilizada a classe `String`. Desta forma a tradução dos conjuntos associados são realizadas para as variáveis `known` e `birthday`, respectivamente as classes `BasicSet<String>` e `BasicPFunction<String,String>`. O predicado contido neste *schema* pode ser traduzido para a seguinte expressão: `known.equals(birthday.domain())`. A partir destas regras é obtida a classe Java presente no Quadro 46. A implementação completa da classe `BasicSet` e `BasicPFunction` é apresentada no Anexo A.

```
	public class BirthdayBook extends Schema {

	@AVariableDeclaration
	public BasicSet<String> known;
	@AVariableDeclaration
	public BasicPFunction<String, String> birthday;

	public BirthdayBook(BasicSet<String> known,
	BasicPFunction<String, String> birthday) {
		super(" BirthdayBook ");
		this.known = known;
		this.birthday = birthday;
	}

	@Override
	protected void checkSpecificationInvariants() throws InvariantViolation {
		Assert.isTrue(known.equals(birthday.domain()));
	}

	@Override
	protected void checkUserInvariants() throws InvariantViolation {
		// nothing
	}

	@Override
	protected Object clone() throws CloneNotSupportedException {
		return new BirthdayBook((BasicSet<String>) known.clone(),
		(BasicPFunction<String, String>) birthday.clone());
	}

	@Override
	protected void execute() {
		// nothing
	}
}
```

Fonte: Miyazawa (2008, p. 77-78).

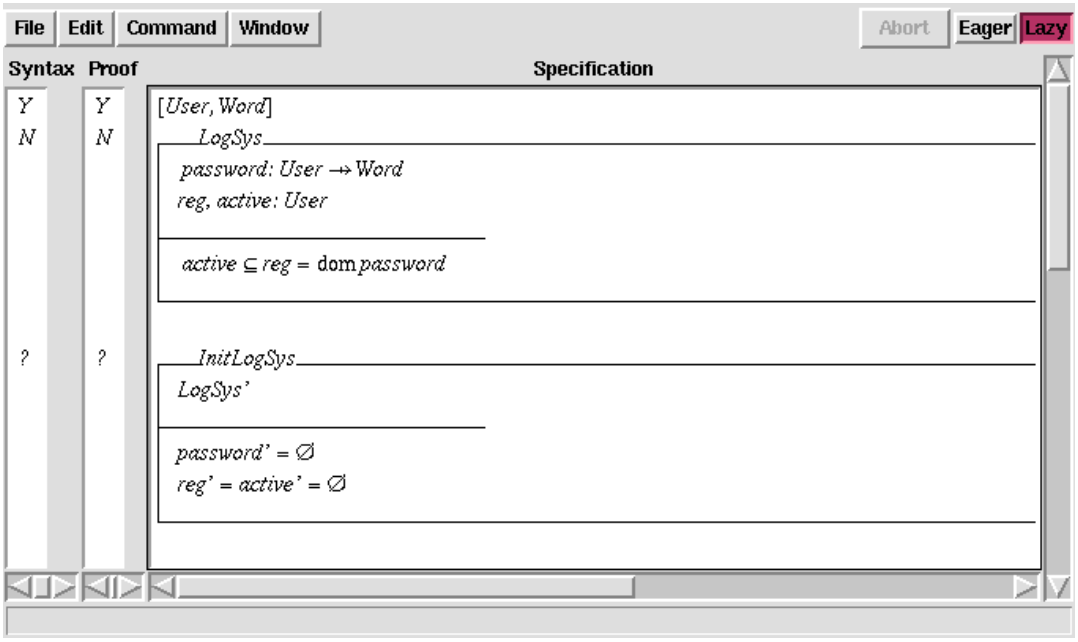
Quadro 46 – Classe gerada pela ferramenta JZed-Gen

2.5 FERRAMENTA DE ESPECIFICAÇÃO Z/EVES

Meisels e Saaltink (1995, p. 1) apresentam Z/EVES como uma ferramenta para analisar especificações em Z. Pode ser utilizada para realizar *parse*, checagem de tipo, checagem de domínios, expansão de *schemas*, realizar cálculo de pré-condições, refinamento de provas e prova de teoremas.

Segundo Saaltink (1999, p. 9), Z/EVES possibilita criar e editar especificações pelo editor de especificações. Também aceita expressões em LaTeX para trabalhar com a especificação em Z.

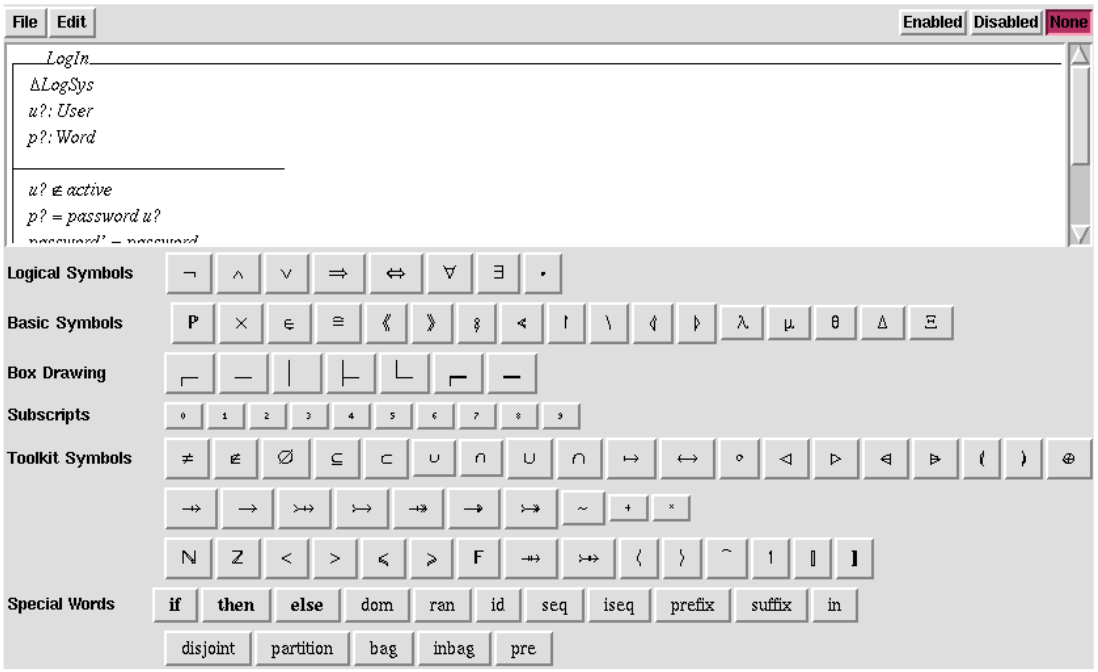
Saaltink (1999, p. 3-4) demonstra a ferramenta Z/EVES apresentando as janelas utilizadas para construir a especificação. Na Figura 1 é apresentada a janela principal de especificação. Na coluna Specification são apresentados os parágrafos construídos. Na coluna *Syntax* é apresentada a checagem da sintaxe do parágrafo. Na coluna *Proof* são verificadas se as provas geradas pelo Z/EVES estão corretas.



Fonte: Saaltink (1999, p. 4).

Figura 1 - Janela de especificação no Z/EVES

Na Figura 2 é apresentada a tela de edição. Nela é possível construir os parágrafos da especificação, sejam eles *schemas*, declarações axiomáticas, ou quaisquer outras definições e construções em Z.



Fonte: Saaltink (1999, p. 8).

Figura 2 - Janela de edição no Z/EVES

Na Figura 3 é apresentada a janela de provas. É apresentado o parágrafo a ser testado e a sua respectiva validade na coluna *Run*.

Fonte: Saaltink (1999, p. 6).

Figura 3 - Janela de provas no Z/EVES

2.6 TRABALHOS CORRELATOS

Hammer e Peleska (1995) apresentam o processo de desenvolvimento do módulo *Cabin ILlumination function* (CIL) presente no *Cabin Intercommunication Data System* (CIDS). O CIDS é um sistema tolerante a falhas que implementa as funções de comunicação da cabine de um avião da família Airbus A330/340. Por ser um sistema de alto risco, a implementação foi rigorosamente acompanhada. Foram utilizados os métodos clássicos de especificação de software, munido de uma ferramenta *Computer-Aided Software Engineering* (CASE) comercial (o nome desta ferramenta não foi apresentado no trabalho). Esta maneira de especificar o sistema tornou-se muito complexa e pouco confiável. Para resolver este problema, as partes críticas do software utilizaram Z como método de especificação. Ao final do desenvolvimento pode-se comprovar que as ferramentas CASE se tornaram insuficiente para lidar com a complexidade do sistema. Foi atribuída a notação Z o sucesso do projeto. É relatado que a especificação do sistema em Z ajudou em muito para criar o *design* da implementação com uma complexidade muito menor. Além destes benefícios, a utilização de Z proporcionou uma redução de custo no desenvolvimento e integração do sistema.

Findeiss (1999) demonstra a especificação de um sistema de compras de *Compact Disc* (CD) utilizando a notação formal *Vienna Development Method* (VDM) e sua aplicação no desenvolvimento de software. Após realizada a especificação, é implementado o referido sistema utilizando o ambiente Uniface. Por fim, é demonstrada a forma de mapeamento realizada entre a especificação formal e a estratégia de implementação. O trabalho é uma apresentação superficial sobre o uso da linguagem de especificação VDM e seu mapeamento para a linguagem Uniface. Outra característica do trabalho é mapear a especificação VDM para estrutra do banco de dados. O trabalho descreve que a utilização da notação VDM é identificada como de “extrema complexidade”.

Stocks (1993) desenvolveu um *framework* que visa introduzir os métodos formais no desenvolvimento de testes. A partir de uma especificação formal em Z, o *framework* gera várias notações formais em Z definindo a forma de como esta deve ser testada. A especificação formal em Z gerada são os casos de testes que o desenvolvedor deve contemplar ao realizar a implementação da especificação formal inicial. A especificação baseada em testes que é apresentada pretende garantir que a especificação seja escrita de forma concisa e que contemple os requisitos definidos. Para isso, o *framework* trabalha com a especificação baseada em *templates*. Vários *templates* são disponibilizados para definir e formalizar a

estrutura das especificações baseadas em teste.

3 DESENVOLVIMENTO

As seguir é apresentado o desenvolvimento de um sistema para biblioteca. Para tanto, são apresentados os requisitos do sistema e a sua especificação formal em Z. Ainda na implementação é descrito como foi mapeada uma especificação formal em Z para Java. Implementação de partes da especificação para Java são apresentadas. A construção das telas do sistema e sua implementação é mostrada. Testes realizados a partir da especificação construída também são apresentados. A operacionalidade da implementação também é descrita. A utilização de provas também é mostrada de uma maneira breve e sintética. Na última seção são feitas discussões sobre os resultados alcançados.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Os requisitos do estudo de caso são:

a) ser especificado utilizando a notação formal Z (Requisito Não Funcional - RNF);
b) implementar testes utilizando a ferramenta JUnit (RNF);
c) utilizar o ambiente de desenvolvimento Eclipse (RNF);
d) permitir inclusão, alteração e remoção de leitores da biblioteca (Requisito Funcional - RF);
e) permitir inclusão, alteração e remoção das diversas categorias de leitores. As categorias de leitores são: aluno de graduação, aluno de pós-graduação, professor, funcionário e usuário externo (RF);
f) permitir inclusão, alteração e remoção das diversas categorias de obras literárias. As categorias literárias são: livro, revista, jornal, trabalho de conclusão de curso, dissertação de mestrado e tese de doutorado (RF);
g) permitir inclusão, alteração e remoção das obras literárias da biblioteca (RF);
h) permitir inclusão, alteração e remoção de funcionários da biblioteca (RF);
i) permitir processamento da reserva da obra literária (RF);
j) permitir empréstimo de obras literária para um leitor (RF);
k) permitir devolução de uma obra literária por um leitor (RF);
l) permitir geração de um relatório com as obras que estão em empréstimo (RF);

m) permitir geração de um relatório com as obras que estão em atraso (RF);
n) permitir consulta da situação de uma obra literária (RF).

3.2 ESPECIFICAÇÃO

Por especificamente utilizar-se de uma metodologia e linguagem de especificação um pouco diferenciada das usuais, a própria especificação torna-se parte relevante do trabalho. Neste ponto é apresentada a forma como a notação Z opera sobre os requisitos identificados.

A especificação das regras do negócio é realizada utilizando o método Z. A especificação foi escrita no ambiente de desenvolvimento Z/EVES. A especificação da interface é apresentada utilizando a técnica de orientação a objetos.

3.2.1 Especificação das regras de negócio

A especificação completa das regras de negócio do sistema de biblioteca é apresentada nesta sub-seção. Os *schemas*, tipos e definições axiomáticas usadas na especificação estão expostos em quadros para melhor entendimento.

No Quadro 47 é apresentada a especificação de tipos básicos utilizados na especificação do sistema.

$$[ID, String]$$

Quadro 47 – Definição dos tipos `ID` e `String`

No Quadro 48 são apresentadas todas as mensagens de erros especificadas no sistema.

```
Report ::= MsgCopyExistsError
         | MsgLiteracyWorkExistsError
         | MsgLibrarianExistsError
         | MsgReaderExistsError
         | MsgReaderTypeExistsError
         | MsgAuthorExistsError
         | MsgCopyNotExistsError
         | MsgLiteracyWorkNotExistsError
         | MsgLibrarianNotExistsError
         | MsgReaderNotExistsError
         | MsgAuthorNotExistsError
         | MsgReaderTypeNotExistsError
         | MsgAuthorHasLiteracyWorkError
         | MsgReaderTypeHasReadersError
         | MsgReaderHasReservationError
         | MsgReaderHasIssueError
         | MsgCopyHasReservationError
         | MsgCopyHasIssueError
         | MsgLiteracyWorkHasCopyError
         | MsgCopyNotAvailableError
         | MsgReaderHasMaxLoansError
         | MsgCopyIsIssuedError
         | MsgCopyIsReserverError
         | MsgCopyIsAvailableError
         | MsgCopyIsNotIssuedError
         | MsgCopyIsReservedError
```

Quadro 48 – Definição das mensagens de erro

No Quadro 49 é apresentada a especificação utilizando a definição axiomática para definir um tipo de dado. Neste caso é declarada uma variável global com o nome `leapYear`, que representa os anos bissextos. Como é demonstrado na especificação, os anos bissextos são todos os divisíveis por 4 e que não sejam divisíveis por 100 ou sejam divisíveis por 400. A inclusão do símbolo de finito ($\mathbb{F}$) junto a declaração do tipo identifica que o conjunto é finito. Por ser finito, apenas são utilizados os anos compreendidos entre 1900 até 9999.

$$
\begin{array}{|l}
leapYear: \mathbb{F}\ \mathbb{N} \\
\hline
\forall year: leapYear \\
\quad \bullet\ year \geqslant 1900 \\
\quad \land\ year \leqslant 9999 \\
\quad \land\ year \bmod 4 = 0 \\
\quad \land\ (year \bmod 100 \neq 0 \lor year \bmod 400 = 0)
\end{array}
$$

Quadro 49 – Definição de ano bissexto

No Quadro 50 é apresentada a especificação do tipo `Date` (data). É realizado nas quatro primeiras linhas da parte predicativa as consistências dos valores das datas. A partir da quinta linha na parte predicativa é realizado o cálculo do valor da data. A variável `value` é utilizada para realizar a comparação de datas.

Date

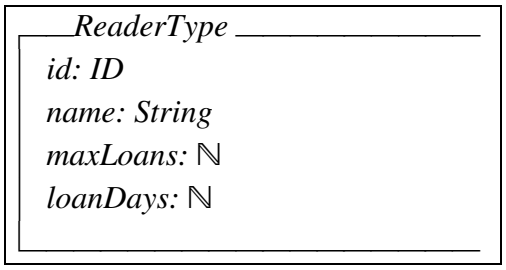
day: 1 .. 31
month: 1 .. 12
year: 1900 .. 9999
value: $\mathbb{N}$

$month \notin \{2, 4, 6, 9, 11\}$
$\lor\ month \in \{4, 6, 9, 11\} \land day \leqslant 30$
$\lor\ month = 2 \land year \notin leapYear \land day \leqslant 28$
$\lor\ month = 2 \land year \in leapYear \land day \leqslant 29$
$value = value + (year - 1900) * 366 \Leftrightarrow year \in leapYear$
$value = value + (year - 1900) * 365 \Leftrightarrow year \notin leapYear$
$value = value + month * 30 \Leftrightarrow month \in \{4, 6, 9, 11\}$
$value = value + month * 31 \Leftrightarrow month \notin \{2, 4, 6, 9, 11\}$
$value = value + month * 29 \Leftrightarrow month = 2 \land year \in leapYear$
$value = value + month * 28 \Leftrightarrow month = 2 \land year \notin leapYear$
$value = value + day$

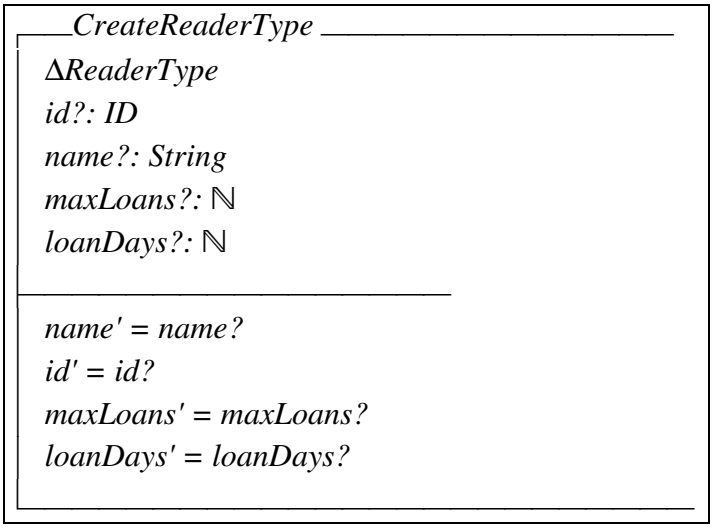
Quadro 50 – Definição de data

No Quadro 51 é descrita a especificação do tipo `ReaderType` (tipo de leitor). `ReaderType` é definido contento um identificador `id` (identificador) do tipo `ID`, `name` (nome) do tipo `String`, `maxLoans` (número máximo de locações) do tipo natural e `loanDays` (número de dias da locação) do tipo natural.

No Quadro 52 é apresentada a especificação do construtor de `ReaderType`. Esta especificação define as entradas que são necessárias para construir um objeto do tipo `ReaderType`. As variáveis de entrada possuem o mesmo nome das variáveis de destino na especificação, apenas incluindo o caractere decorador ponto de interrogação (?) para entrada e apóstrofo (‘) para estado posterior do *schema*.



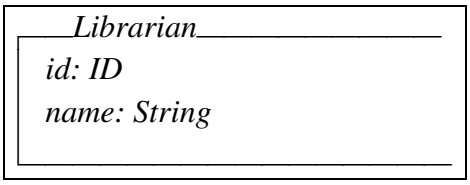
Quadro 51 – Definição de tipo de leitor



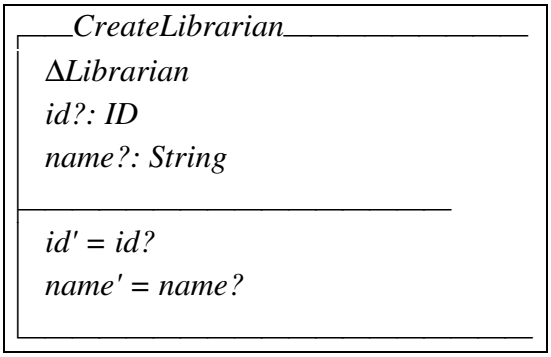
Quadro 52 – Definição de criação de tipo de leitor

No Quadro 53 é descrita a especificação do tipo `Librarian` (bibliotecário). `Librarian` é definido contento um identificador `id` (identificador) do tipo `ID` e `name` (nome) do tipo `String`.

No Quadro 54 é apresentada a especificação do construtor de `Librarian`. Esta especificação define as entradas que são necessárias para construir um objeto do tipo `Librarian`.



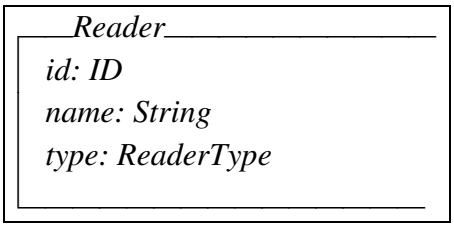
Quadro 53 – Definição de bibliotecário



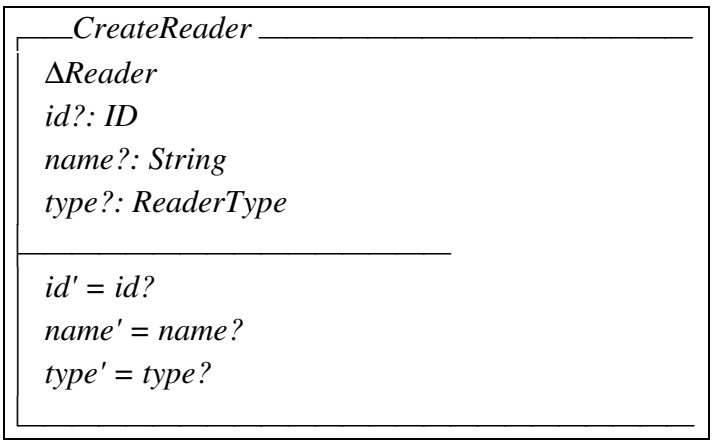
Quadro 54 – Definição de criação de bibliotecário

No Quadro 55 é descrita a especificação do tipo `Reader` (leitor). `Reader` é definido contento um identificador `id` (identificador) do tipo `ID`, `name` (nome) do tipo `String` e `type` (tipo) do tipo `ReaderType`.

No Quadro 56 é apresentada a especificação do construtor de `Reader`. Esta especificação define as entradas que são necessárias para construir um objeto do tipo `Reader`.



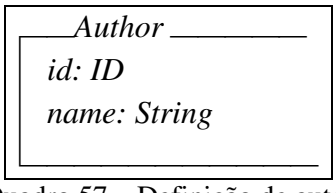
Quadro 55 – Definição de leitor



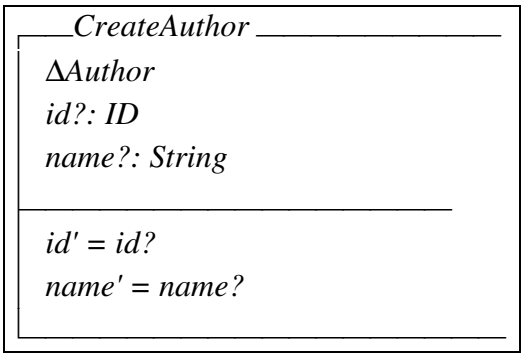
Quadro 56 – Definição de criação de leitor

No Quadro 57 é descrita a especificação do tipo `Author` (autor). `Author` é definido contento um identificador `id` (identificador) do tipo `ID` e `name` (nome) do tipo `String`.

No Quadro 58 é apresentada a especificação do construtor de `Author`. Esta especificação define as entradas que são necessárias para construir um objeto do tipo `Author`.



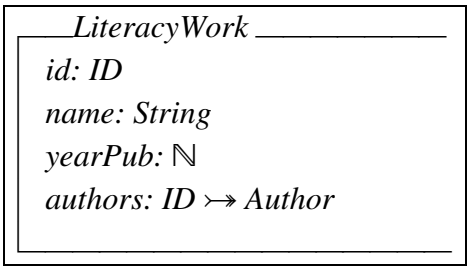
Quadro 57 – Definição de autor



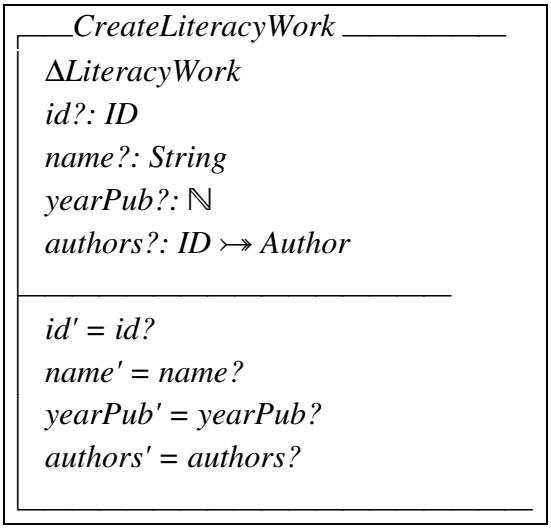
Quadro 58 – Definição de criação de autor

No Quadro 59 é descrita a especificação do tipo `LiteracyWork` (obra literária). `LiteracyWork` é definido contento um identificador `id` (identificador) do tipo `ID`, `name` (nome) do tipo `String`, `yearPub` (ano de publicação) do tipo natural e `authors` (autores) do tipo função bijetora relacionando `ID` e `Author`.

No Quadro 60 é apresentada a especificação do construtor de `LiteracyWork`. Esta especificação define as entradas que são necessárias para construir um objeto do tipo `LiteracyWork`.



Quadro 59 – Definição de obra literária

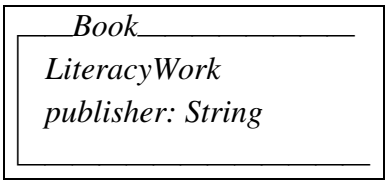


Quadro 60 – Definição de criação de obra literária

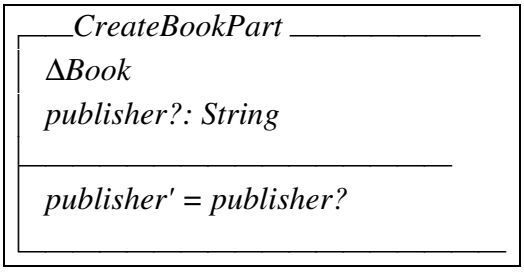
No Quadro 61 é descrita a especificação do tipo `Book` (livro). `Book` é definido como uma extensão de `LiteracyWork`, acrescido de `publisher` (editora) do tipo `String`.

No Quadro 62 é apresentada a especificação do construtor de `Book`. Esta especificação define as entradas que são necessárias para construir um objeto do tipo `Book`.

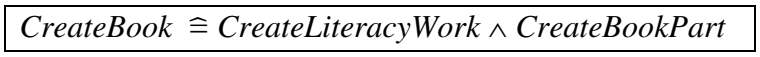
No Quadro 63 é apresentada a especificação unindo os *schemas* especificados de `LiteracyWork` e `Book`, formando deste modo a construção do objeto `Book`.



Quadro 61 – Definição de livro



Quadro 62 – Definição de criação do livro

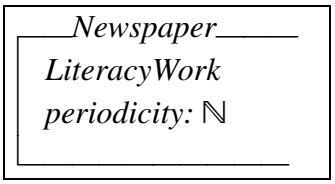


Quadro 63 – Definição completa de criação do livro

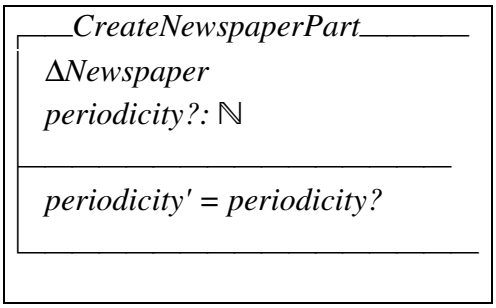
No Quadro 64 é descrita a especificação do tipo `Newspaper` (jornal). `Newspaper` é definido como uma extensão de `LiteracyWork`, acrescido de `periodicity` (periodicidade) do tipo natural.

No Quadro 65 é apresentada a especificação do construtor de `Newspaper`. Esta especificação define as entradas que são necessárias para construir um objeto do tipo `Newspaper`.

No Quadro 66 é apresentada a especificação unindo os *schemas* especificados de `LiteracyWork` e `Newspaper`, formando deste modo a construção do objeto `Newspaper`.



Quadro 64 – Definição de jornal



Quadro 65 – Definição de criação de jornal

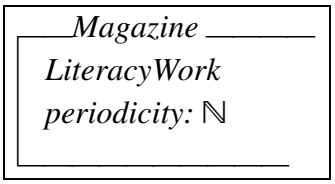
$$CreateNewspaper \mathrel{\widehat=} CreateLiteracyWork \land CreateNewspaperPart$$

Quadro 66 – Definição completa de criação de jornal

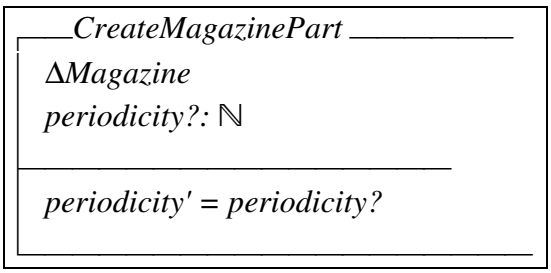
No Quadro 67 é descrita a especificação do tipo `Magazine` (revista). `Magazine` é definido como uma extensão de `LiteracyWork`, acrescido de `periodicity` (periodicidade) do tipo natural.

No Quadro 68 é apresentada a especificação do construtor de `Magazine`. Esta especificação define as entradas que são necessárias para construir um objeto do tipo `Magazine`.

No Quadro 69 é apresentada a especificação unindo os *schemas* especificados de `LiteracyWork` e `Magazine`, formando deste modo a construção do objeto `Magazine`.



Quadro 67 – Definição de revista



Quadro 68 – Definição de criação de revista

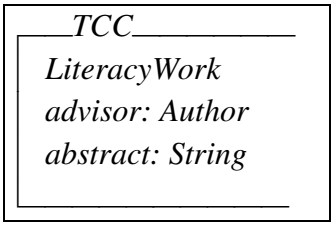
$$CreateMagazine \mathrel{\widehat=} CreateLiteracyWork \land CreateMagazinePart$$

Quadro 69 – Definição completa de criação de revista

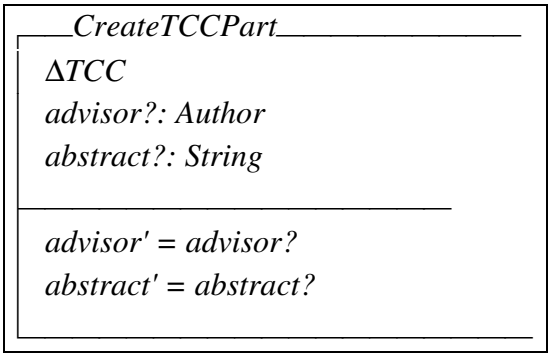
No Quadro 70 é descrita a especificação do tipo `TCC` (trabalho de conclusão de curso). `TCC` é definido como uma extensão de `LiteracyWork`, acrescido de `advisor` (orientador) do tipo `Author` e `abstract` (*abstract*) do tipo `String`.

No Quadro 71 é apresentada a especificação do construtor de `TCC`. Esta especificação define as entradas que são necessárias para construir um objeto do tipo `TCC`.

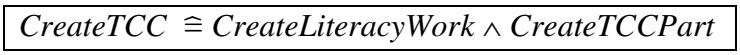
No Quadro 72 é apresentada a especificação unindo os *schemas* especificados de `LiteracyWork` e `TCC`, formando deste modo a construção do objeto `TCC`.



Quadro 70 – Definição de `TCC`



Quadro 71 – Definição de criação de `TCC`

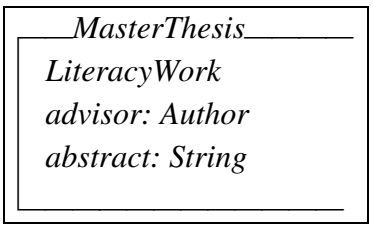


Quadro 72 – Definição completa de criação de `TCC`

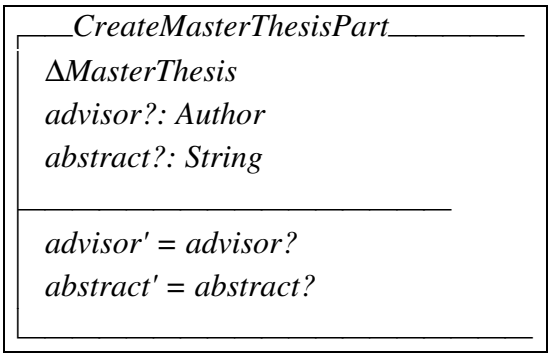
No Quadro 73 é descrita a especificação do tipo `MasterThesis` (dissertação de mestrado). `MasterThesis` é definido como uma extensão de `LiteracyWork`, acrescido de `advisor` (orientador) do tipo `Author` e `abstract` (*abstract*) do tipo `String`.

No Quadro 74 é apresentada a especificação do construtor de `MasterThesis`. Esta especificação define as entradas que são necessárias para construir um objeto do tipo `MasterThesis`.

No Quadro 75 é apresentada a especificação unindo os *schemas* especificados de `LiteracyWork` e `MasterThesis`, formando deste modo a construção do objeto `MasterThesis`.



Quadro 73 – Definição de dissertação de mestrado



Quadro 74 – Definição de criação de dissertação de mestrado

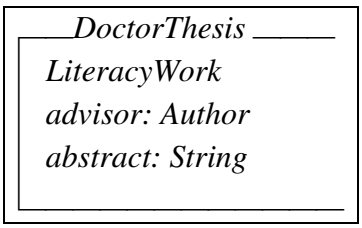
$$CreateMasterThesis \;\widehat{=}\; CreateLiteracyWork \wedge CreateMasterThesisPart$$

Quadro 75 – Definição completa de criação de dissertação de mestrado

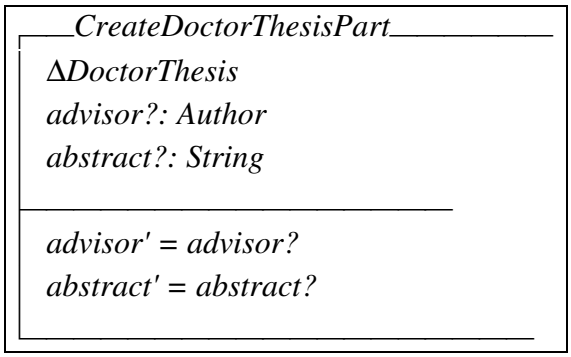
No Quadro 76 é descrita a especificação do tipo `DoctorThesis` (tese de doutorado). `DoctorThesis` é definido como uma extensão de `LiteracyWork`, acrescido de `advisor` (orientador) do tipo `Author` e `abstract` (*abstract*) do tipo `String`.

No Quadro 77 é apresentada a especificação do construtor de `DoctorThesis`. Esta especificação define as entradas que são necessárias para construir um objeto do tipo `DoctorThesis`.

No Quadro 78 é apresentada a especificação unindo os *schemas* especificados de `LiteracyWork` e `DoctorThesis`, formando deste modo a construção do objeto `DoctorThesis`.



Quadro 76 – Definição de tese de doutorado



Quadro 77 – Definição de criação de tese de doutorado

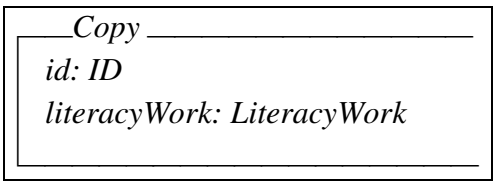
$$CreateDoctorThesis \;\widehat{=}\; CreateLiteracyWork \wedge CreateDoctorThesisPart$$

Quadro 78 – Definição completa de criação de tese de doutorado

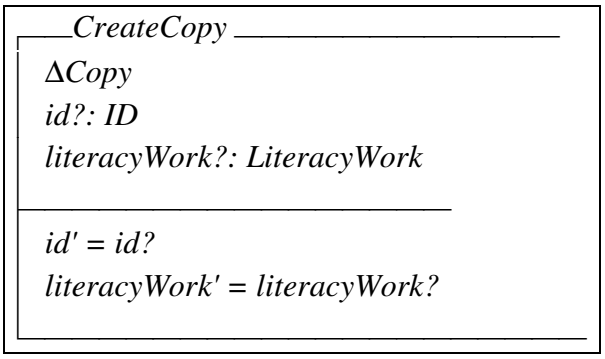
No Quadro 79 é descrita a especificação do tipo `Copy` (cópia). `Copy` é definido contento

um identificador `id` (identificador) do tipo `ID e literacyWork` (obra literária) do tipo `LiteracyWork`.

No Quadro 80 é apresentada a especificação do construtor de `Copy`. Esta especificação define as entradas que são necessárias para construir um objeto do tipo `Copy`.



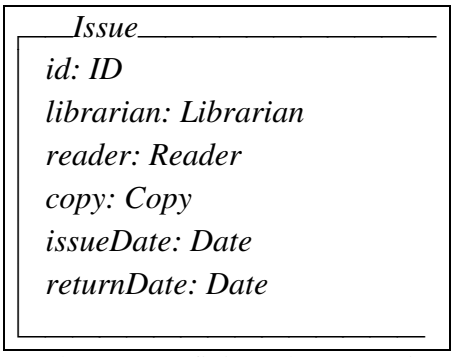
Quadro 79 – Definição de cópia



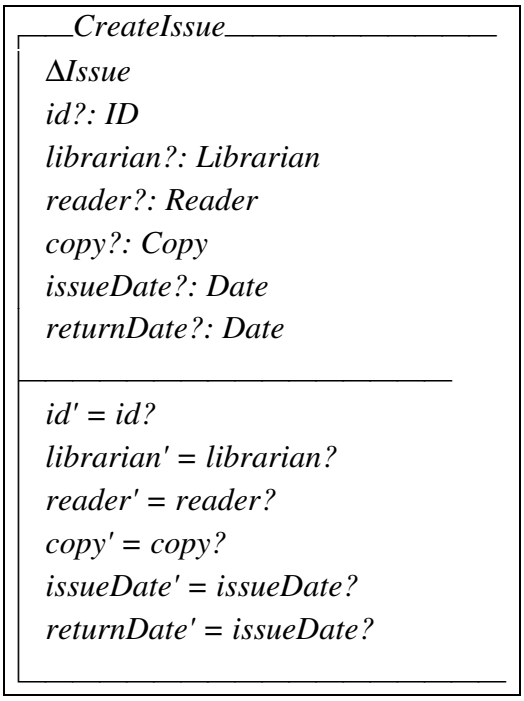
Quadro 80 – Definição de criação de cópia

No Quadro 81 é descrita a especificação do tipo `Issue` (empréstimo). `Issue` é definido contento um identificador `id` (identificador) do tipo `ID, librarian` (bibliotecário) do tipo `Librarian, reader` (leitor) do tipo `Reader, copy` (cópia) do tipo `Copy, issueDate` (data de empréstimo) do tipo `Date` e `returnDate` (data de retorno) do tipo `Date`.

No Quadro 82 é apresentada a especificação do construtor de `Issue`. Esta especificação define as entradas que são necessárias para construir um objeto do tipo `Issue`.



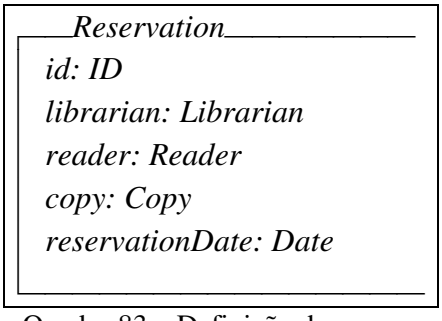
Quadro 81 – Definição de empréstimo



Quadro 82 – Definição de criação de empréstimo

No Quadro 83 é descrita a especificação do tipo `Reservation` (reserva). `Reservation` é definido contento um identificador `id` (identificador) do tipo `ID`, `librarian` (bibliotecário) do tipo `Librarian`, `reader` (leitor) do tipo `Reader`, `copy` (cópia) do tipo `Copy` e `reservationDate` (data de reserva) do tipo `Date`.

No Quadro 84 é apresentada a especificação do construtor de `Reservation`. Esta especificação define as entradas que são necessárias para construir um objeto do tipo `Reservation`.



Quadro 83 – Definição de reserva

CreateReservation

Δ*Reservation*
id?: ID
librarian?: Librarian
reader?: Reader
copy?: Copy
reservationDate?: Date

---

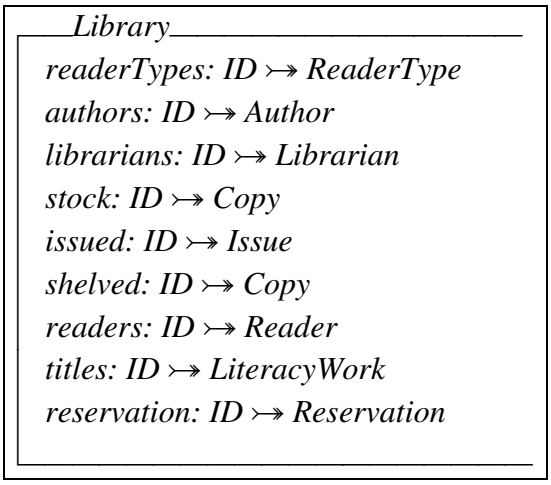
id' = id?
librarian' = librarian?
reader' = reader?
copy' = copy?
reservationDate' = reservationDate?

Quadro 84 – Definição de criação de reserva

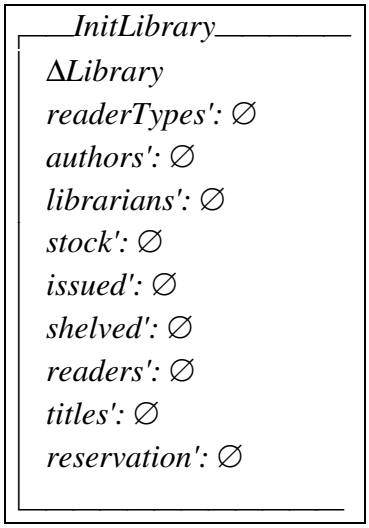
No Quadro 85 é descrita a especificação do tipo `Library` (biblioteca). `Library` é definido contento:

a) `readerTypes` (tipos de leitores): uma função bijetora relacionando `ID` com `ReaderType`;
b) `authors` (autores): uma função bijetora relacionando `ID` com `Author`;
c) `librarians` (bibliotecários): uma função bijetora relacionando `ID` com `Librarian`;
d) `stock` (estoque): uma função bijetora relacionando `ID` com `Copy`;
e) `issued` (emprestados): uma função bijetora relacionando `ID` com `Issue`;
f) `shelved` (prateleira): uma função bijetora relacionando `ID` com `Copy`;
g) `readers` (leitores): uma função bijetora relacionando `ID` com `Reader`;
h) `titles` (títulos): uma função bijetora relacionando `ID` com `LiteracyWork`;
i) `reservation` (reserva): uma função bijetora relacionando `ID` com `Reservation`.

No Quadro 86 é apresentada a especificação que inicializa o *schema* `Library`. Esta especificação inicializa todas as coleções com um conjunto vazio.



Quadro 85 – Definição de biblioteca



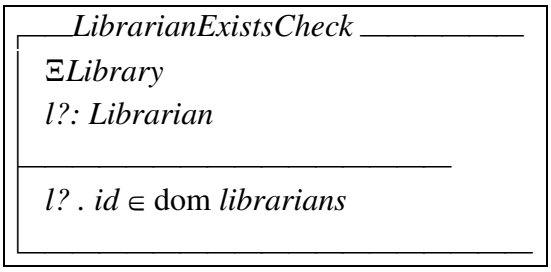
Quadro 86 – Definição de inicialização da biblioteca

No Quadro 87 é descrita a especificação de verificação da existência de um bibliotecário na biblioteca. O *schema* `LibrarianExistsCheck` verifca se um objeto do tipo `Librarian` contém um `ID` que pertença a coleção `librarians` definida em `Library`.

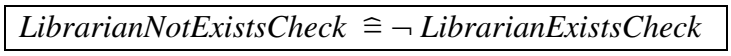
No Quadro 88 é descrita a especificação de verificação da não existência de um bibliotecário na biblioteca. O *schema* `LibrarianNotExistsCheck` verifca se um objeto do tipo `Librarian` contém um `ID` que não pertença a coleção `librarians` definida em `Library`.

No Quadro 89 é descrita a especificação de erro ao encontrar um bibliotecário cadastrado na biblioteca. O *schema* `LibrarianExistsError` retorna a mensagem de erro `rep!` do tipo `Report`.

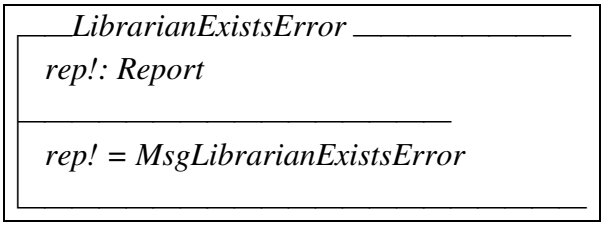
No Quadro 90 é descrita a especificação de erro ao não encontrar um bibliotecário cadastrado na biblioteca. O *schema* `LibrarianNotExistsError` retorna a mensagem de erro `rep!` do tipo `Report`.



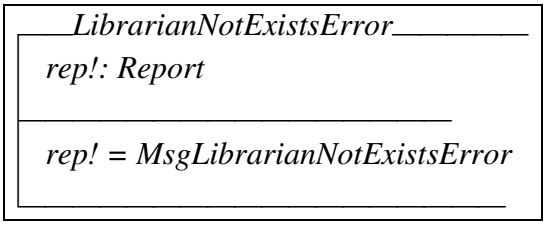
Quadro 87 – Definição de checagem de existência de bibliotecário



Quadro 88 – Definição de checagem de não existência de bibliotecário

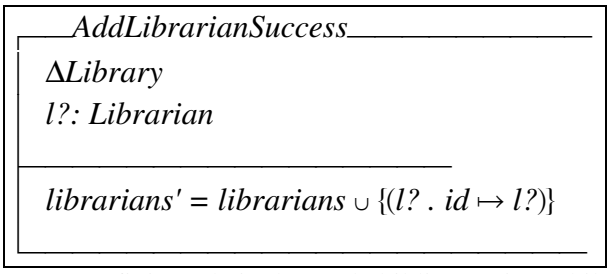


Quadro 89 – Definição de erro de bibliotecário existente

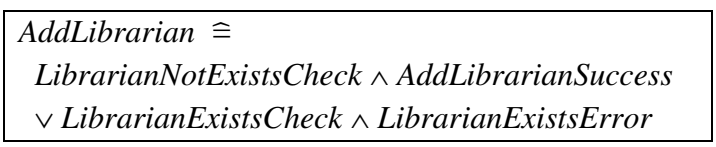


Quadro 90 – Definição de erro de bibliotecário não existente

No Quadro 91 é descrita a especificação de adição com sucesso de um bibliotecário a biblioteca. O *schema* `AddLibrarianSuccess` adiciona um objeto do tipo `Librarian` a coleção `librarians` definida em `Library`. A operação de inclusão de um bibliotecário é definida no Quadro 92, no *schema* `AddLibrarian`. A inclusão é realizada caso não exista um bibliotecário cadastrado ou ocorre erro caso exista um bibliotecário cadastrado.



Quadro 91 – Definição de inclusão de bibliotecário na biblioteca



Quadro 92 – Definição completa de inclusão de bibliotecário na biblioteca

No Quadro 93 é descrita a especificação de remoção com sucesso de um bibliotecário

da biblioteca. O *schema* `RemoveLibrarianSuccess` remove um objeto do tipo `Librarian` da coleção `librarians` definida em `Library`. A operação de remoção de um bibliotecário é definida no Quadro 94, no *schema* `RemoveLibrarian`. A remoção é realizada caso exista um bibliotecário cadastrado ou ocorre erro caso não exista um bibliotecário cadastrado.

RemoveLibrarianSuccess

$\Delta Library$
$l?: Librarian$

$librarians' = \{l?\ .\ id\} \lhd librarians$

Quadro 93 – Definição de remoção de bibliotecário na biblioteca

$$RemoveLibrarian \mathrel{\widehat=} LibrarianExistsCheck \land RemoveLibrarianSuccess \lor LibrarianNotExistsCheck \land LibrarianNotExistsError$$

Quadro 94 – Definição completa de remoção de bibliotecário na biblioteca

No Quadro 95 é descrita a especificação de verificação da existência de um autor na biblioteca. O *schema* `AuthorExistsCheck` verifica se um objeto do tipo `Author` contém um `ID` que pertença a coleção `authors` definida em `Library`.

No Quadro 96 é descrita a especificação de verificação da não existência de um autor na biblioteca. O *schema* `AuthorNotExistsCheck` verifica se um objeto do tipo `Author` contém um `ID` que não pertença a coleção `authors` definida em `Library`.

No Quadro 97 é descrita a especificação de erro ao encontrar um autor cadastrado na biblioteca. O *schema* `AuthorExistsError` retorna a mensagem de erro `rep!` do tipo `Report`.

No Quadro 98 é descrita a especificação de erro ao não encontrar um autor cadastrado na biblioteca. O *schema* `AuthorNotExistsError` retorna a mensagem de erro `rep!` do tipo `Report`.

AuthorExistsCheck

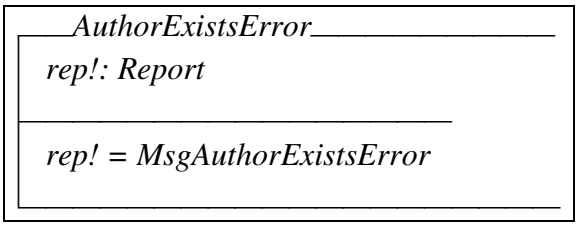
$\Xi Library$
$a?: Author$

$a?\ .\ id \in \mathrm{dom}\ authors$

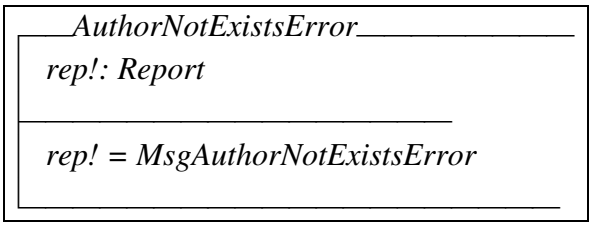
Quadro 95 – Definição de checagem de existência de autor

$$AuthorNotExistsCheck \mathrel{\widehat=} \neg\ AuthorExistsCheck$$

Quadro 96 – Definição de checagem de não existência de autor

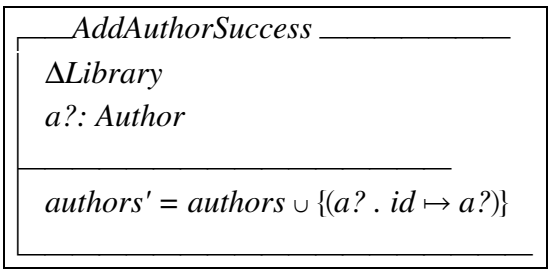


Quadro 97 – Definição de erro de autor existente

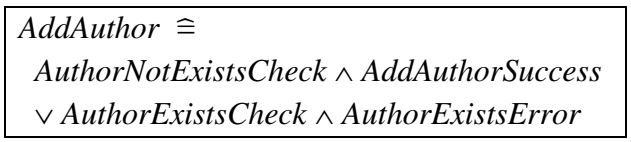


Quadro 98 – Definição de erro de autor não existente

No Quadro 99 é descrita a especificação de adição com sucesso de um autor a biblioteca. O *schema* `AddAuthorSuccess` adiciona um objeto do tipo `Author` a coleção `authors` definida em `Library`. A operação de inclusão de um autor é definida no Quadro 100, no *schema* `AddAuthor`. A inclusão é realizada caso não exista um autor cadastrado ou ocorre erro caso exista um autor cadastrado.



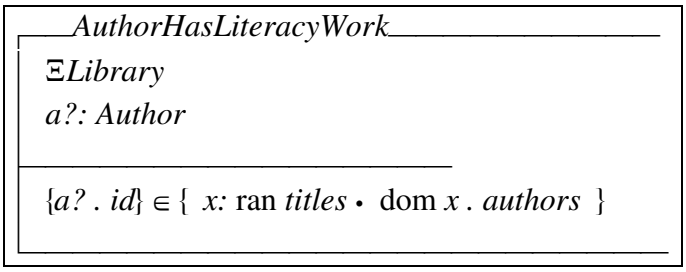
Quadro 99 – Definição de inclusão de autor na biblioteca



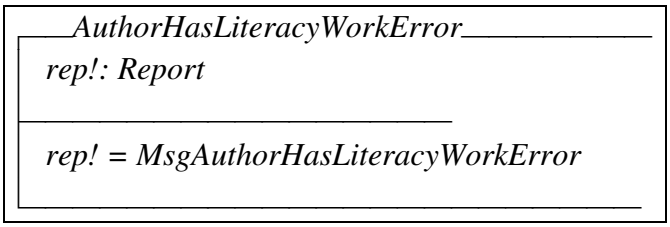
Quadro 100 – Definição completa de inclusão de autor na biblioteca

No Quadro 101 é descrita a especificação de verificação da existência de um autor associado a uma obra literária na biblioteca. O schema `AuthorHasLiteracyWork` verifica se o `ID` do objeto do tipo `Author` pertence ao conjunto de `authors` presente em `titles`.

No Quadro 102 é descrita a especificação de erro ao encontrar um autor associado a uma obra literária na biblioteca. O *schema* `AuthorHasLiteracyWorkError` retorna a mensagem de erro `rep!` do tipo `Report`.

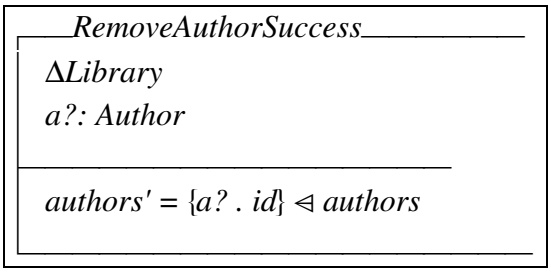


Quadro 101 – Definição de checagem de existência de autor associado a obra literária

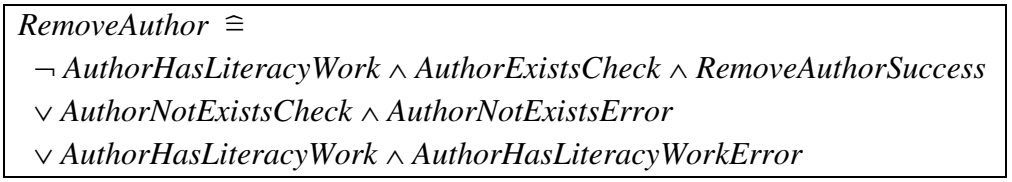


Quadro 102 – Definição de erro ao existir autor associado a obra literária

No Quadro 103 é descrita a especificação de remoção com sucesso de um autor da biblioteca. O *schema* `RemoveAuthorSuccess` remove um objeto do tipo `Author` da coleção `authors` definida em `Library`. A operação de remoção de um autor é definida no Quadro 104, no *schema* `RemoveAuthor`. A remoção é realizada caso exista um autor cadastrado e este autor não esteja associado a uma obra literária. Ocorre erro caso não exista um autor cadastrado ou este possua obra literária associada.



Quadro 103 – Definição de remoção de autor na biblioteca



Quadro 104 – Definição completa de remoção de autor na biblioteca

No Quadro 105 é descrita a especificação de verificação da existência de um tipo de leitor na biblioteca. O *schema* `ReaderTypeExistsCheck` verifica se um objeto do tipo `ReaderType` contém um `ID` que pertença a coleção `readerTypes` definida em `Library`.

No Quadro 106 é descrita a especificação de verificação da não existência de um tipo de leitor na biblioteca. O *schema* `ReaderTypeNotExistsCheck` verifica se um objeto do tipo `ReaderType` contém um `ID` que não pertença a coleção `readerTypes` definida em `Library`.

No Quadro 107 é descrita a especificação de erro ao encontrar um tipo de leitor

cadastrado na biblioteca. O *schema* `ReaderTypeExistsError` retorna a mensagem de erro `rep!` do tipo `Report`.

No Quadro 108 é descrita a especificação de erro ao não encontrar um tipo de leitor cadastrado na biblioteca. O *schema* `ReaderTypeNotExistsError` retorna a mensagem de erro `rep!` do tipo `Report`.

ReaderTypeExistsCheck

$\Xi Library$
$r?: ReaderType$

$r?\ .\ id \in \text{dom}\ readerTypes$

Quadro 105 – Definição de checagem de existência de tipo de leitor

$ReaderTypeNotExistsCheck \mathrel{\widehat=} \neg\ ReaderTypeExistsCheck$

Quadro 106 – Definição de checagem de não existência de tipo de leitor

ReaderTypeExistsError

$rep!: Report$

$rep! = MsgReaderTypeExistsError$

Quadro 107 – Definição de erro de tipo de leitor existente

ReaderTypeNotExistsError

$rep!: Report$

$rep! = MsgReaderTypeNotExistsError$

Quadro 108 – Definição de erro de tipo de leitor não existente

No Quadro 109 é descrita a especificação de adição com sucesso de um tipo de leitor a biblioteca. O *schema* `AddReaderTypeSuccess` adiciona um objeto do tipo `ReaderType` a coleção `readerTypes` definida em `Library`. A operação de inclusão de um tipo de leitor é definida no Quadro 110, no *schema* `AddReaderType`. A inclusão é realizada caso não exista um tipo de leitor cadastrado ou ocorre erro caso exista um tipo de leitor cadastrado.

AddReaderTypeSuccess

$\Delta Library$
$r?: ReaderType$

$readerTypes' = readerTypes \cup \{(r?\ .\ id \mapsto r?)\}$

Quadro 109 – Definição de inclusão de tipo de leitor na biblioteca

$$AddReaderType \mathrel{\widehat=} ReaderTypeNotExistsCheck \land AddReaderTypeSuccess \lor ReaderTypeExistsCheck \land ReaderTypeExistsError$$

Quadro 110 – Definição completa de inclusão de tipo de leitor na biblioteca

No Quadro 111 é descrita a especificação de verificação da existência de um tipo de leitor associado a uma obra literária na biblioteca. O schema `ReaderTypeHasReaders` verifica se o `ID` do objeto do tipo `ReaderType` está referenciado por `type` da coleção de `readers`.

No Quadro 112 é descrita a especificação de erro ao encontrar um tipo de leitor associado a um leitor na biblioteca. O *schema* `ReaderTypeHasReadersError` retorna a mensagem de erro `rep!` do tipo `Report`.

ReaderTypeHasReaders

$\Delta Library$
$r?: ReaderType$

$r?.id \in \{ x: \text{ran } readers \bullet x.type.id \}$

Quadro 111 – Definição de checagem de existência de tipo de leitor associado a leitor

ReaderTypeHasReadersError

$rep!: Report$

$rep! = MsgReaderTypeHasReadersError$

Quadro 112 – Definição de erro ao existir tipo de leitor associado a leitor

No Quadro 113 é descrita a especificação de remoção com sucesso de um tipo de leitor da biblioteca. O *schema* `RemoveReaderTypeSuccess` remove um objeto do tipo `ReaderType` da coleção `readerTypes` definida em `Library`. A operação de remoção de um tipo de leitor é definida no Quadro 114, no *schema* `RemoveReaderType`. A remoção é realizada caso exista um tipo de leitor cadastrado e este não esteja associado a uma obra literária. Ocorre erro caso não exista um tipo de leitor cadastrado ou este possua obra literária associada.

RemoveReaderTypeSuccess

$\Delta Library$
$r?: ReaderType$

$readerTypes' = \{r?.id\} \lhd readerTypes$

Quadro 113 – Definição de remoção de tipo de leitor na biblioteca

$$RemoveReaderType \mathrel{\widehat=} \neg\, ReaderTypeHasReaders \land ReaderTypeExistsCheck \land RemoveReaderTypeSuccess \lor ReaderTypeNotExistsCheck \land ReaderTypeNotExistsError \lor ReaderTypeHasReaders \land ReaderTypeHasReadersError$$

Quadro 114 – Definição completa de remoção de tipo de leitor na biblioteca

No Quadro 115 é descrita a especificação de verificação da existência de um leitor na biblioteca. O *schema* `ReaderExistsCheck` verifica se um objeto do tipo `Reader` contém um `ID` que pertença a coleção `readers` definida em `Library`.

No Quadro 116 é descrita a especificação de verificação da não existência de um leitor na biblioteca. O *schema* `ReaderNotExistsCheck` verifica se um objeto do tipo `Reader` contém um `ID` que não pertença a coleção `readers` definida em `Library`.

No Quadro 117 é descrita a especificação de erro ao encontrar um leitor cadastrado na biblioteca. O *schema* `ReaderExistsError` retorna a mensagem de erro `rep!` do tipo `Report`.

No Quadro 118 é descrita a especificação de erro ao não encontrar um leitor cadastrado na biblioteca. O *schema* `ReaderNotExistsError` retorna a mensagem de erro `rep!` do tipo `Report`.

ReaderExistsCheck

$$\Xi Library$$
$$r?: Reader$$

$$r?\,.\,id \in \text{dom } readers$$

Quadro 115 – Definição de checagem de existência de leitor

$$ReaderNotExistsCheck \mathrel{\widehat=} \neg\, ReaderExistsCheck$$

Quadro 116 – Definição de checagem de não existência de leitor

ReaderExistsError

$$rep!: Report$$

$$rep! = MsgReaderExistsError$$

Quadro 117 – Definição de erro de leitor existente

ReaderNotExistsError

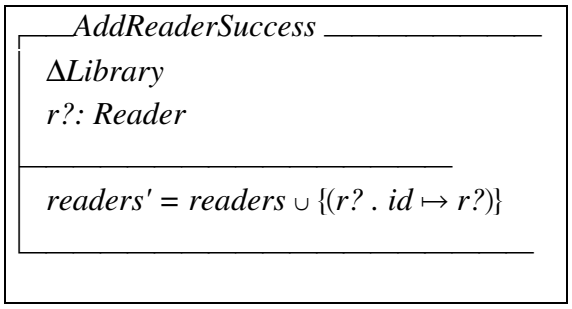
$$rep!: Report$$

$$rep! = MsgReaderNotExistsError$$

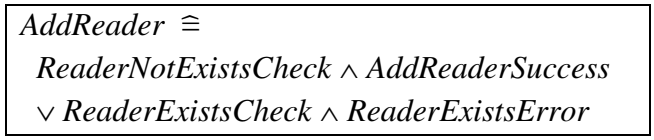
Quadro 118 – Definição de erro de leitor não existente

No Quadro 119 é descrita a especificação de adição com sucesso de um leitor a

biblioteca. O *schema* `AddReaderSuccess` adiciona um objeto do tipo `Reader` a coleção `readers` definida em `Library`. A operação de inclusão de um leitor é definida no Quadro 120, no *schema* `AddReader`. A inclusão é realizada caso não exista um leitor cadastrado ou ocorre erro caso exista um leitor cadastrado.



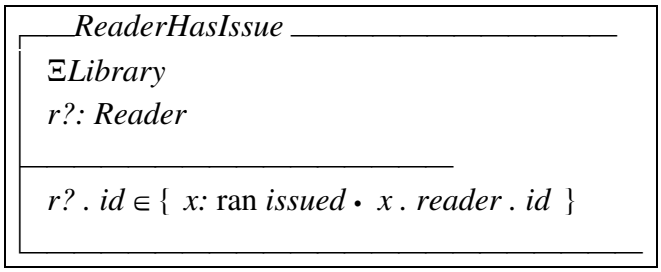
Quadro 119 – Definição de inclusão de leitor na biblioteca



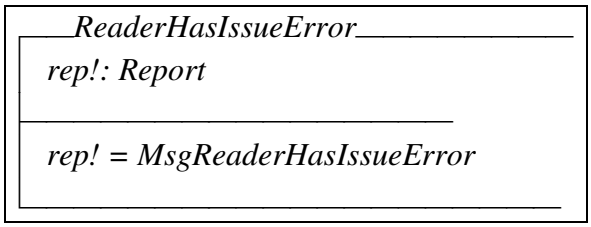
Quadro 120 – Definição completa de inclusão de leitor na biblioteca

No Quadro 121 é descrita a especificação de verificação da existência de um leitor associado a um empréstimo na biblioteca. O *schema* `ReaderHasIssue` verifica se o `ID` do objeto do tipo `Reader` está referenciado por `reader` da coleção de `issued`.

No Quadro 122 é descrita a especificação de erro ao encontrar um leitor associado a um empréstimo na biblioteca. O *schema* `ReaderHasIssueError` retorna a mensagem de erro `rep!` do tipo `Report`.



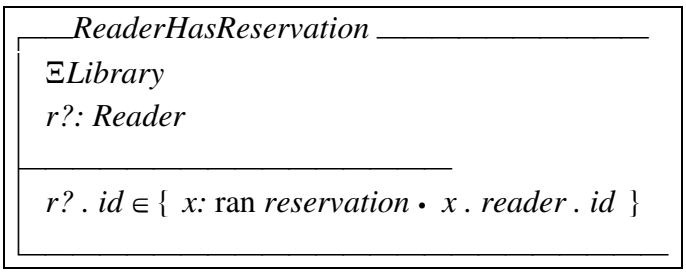
Quadro 121 – Definição de checagem de existência de empréstimo associado ao leitor



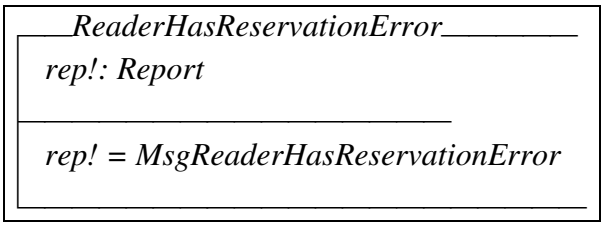
Quadro 122 – Definição de erro ao existir empréstimo associado ao leitor

No Quadro 123 é descrita a especificação de verificação da existência de um leitor associado a uma reserva na biblioteca. O *schema* `ReaderHasReservation` verifica se o `ID` do objeto do tipo `Reader` está referenciado por `reader` da coleção de `reservation`.

No Quadro 124 é descrita a especificação de erro ao encontrar um leitor associado a uma reserva na biblioteca. O *schema* `ReaderHasReservationError` retorna a mensagem de erro `rep!` do tipo `Report`.

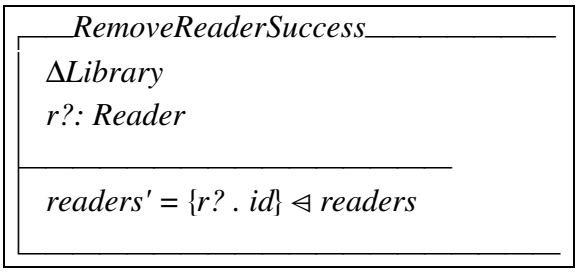


Quadro 123 – Definição de checagem de existência de reserva associada ao leitor



Quadro 124 – Definição de erro ao existir reserva associada ao leitor

No Quadro 125 é descrita a especificação de remoção com sucesso de um leitor da biblioteca. O *schema* `RemoveReaderSuccess` remove um objeto do tipo `Reader` da coleção `readers` definida em `Library`. A operação de remoção de um leitor é definida no Quadro 126, no *schema* `RemoveReader`. A remoção é realizada caso exista um leitor cadastrado e este não tenha empréstimos em aberto ou reserva. Ocorre erro caso não exista um leitor cadastrado, possua empréstimo em aberto ou tenha reserva.



Quadro 125 – Definição de remoção de leitor na biblioteca

$$
\begin{aligned}
&RemoveReader \mathrel{\widehat=} \\
&\quad \neg\, ReaderHasIssue \\
&\quad \land \neg\, ReaderHasReservation \\
&\quad \land\, ReaderExistsCheck \\
&\quad \land\, RemoveReaderSuccess \\
&\quad \lor\, ReaderNotExistsCheck \land ReaderNotExistsError \\
&\quad \lor\, ReaderHasReservation \land ReaderHasReservationError \\
&\quad \lor\, ReaderHasIssue \land ReaderHasIssueError
\end{aligned}
$$

Quadro 126 – Definição completa de remoção de leitor na biblioteca

No Quadro 127 é descrita a especificação de verificação da existência de uma obra literária na biblioteca. O *schema* `LiteracyWorkExistsCheck` verifica se um objeto do tipo `LiteracyWork` contém um `ID` que pertença a coleção `titles` definida em `Library`.

No Quadro 128 é descrita a especificação de verificação da não existência de uma obra literária na biblioteca. O *schema* `LiteracyWorkNotExistsCheck` verifica se um objeto do tipo `LiteracyWork` contém um `ID` que não pertença a coleção `titles` definida em `Library`.

No Quadro 129 é descrita a especificação de erro ao encontrar uma obra literária cadastrada na biblioteca. O *schema* `LiteracyWorkExistsError` retorna a mensagem de erro `rep!` do tipo `Report`.

No Quadro 130 é descrita a especificação de erro ao não encontrar uma obra literária cadastrada na biblioteca. O *schema* `LiteracyWorkNotExistsError` retorna a mensagem de erro `rep!` do tipo `Report`.

$$
\begin{array}{|l}
\hline
\textit{LiteracyWorkExistsCheck} \\
\hline
\Xi \textit{Library} \\
\textit{l?: LiteracyWork} \\
\hline
\textit{l?} \,.\, \textit{id} \in \text{dom}\ \textit{titles} \\
\hline
\end{array}
$$

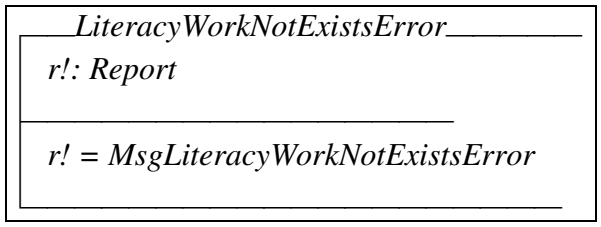
Quadro 127 – Definição de checagem de existência de obra literária

$$
\textit{LiteracyWorkNotExistsCheck} \mathrel{\widehat=} \neg\, \textit{LiteracyWorkExistsCheck}
$$

Quadro 128 – Definição de checagem de não existência de obra literária

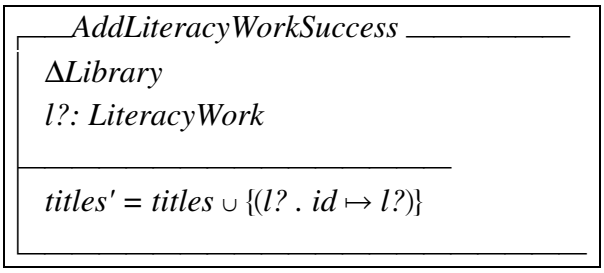
$$
\begin{array}{|l}
\hline
\textit{LiteracyWorkExistsError} \\
\hline
\textit{r!: Report} \\
\hline
\textit{r!} = \textit{MsgLiteracyWorkExistsError} \\
\hline
\end{array}
$$

Quadro 129 – Definição de erro de obra literária existente

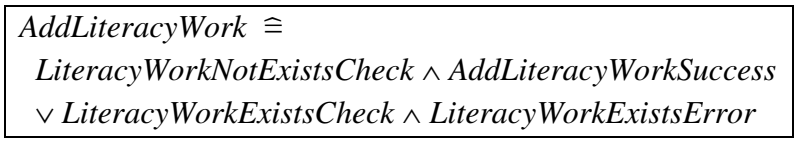


Quadro 130 – Definição de erro de obra literária não existente

No Quadro 131 é descrita a especificação de adição com sucesso de uma obra literária à biblioteca. O *schema* `AddLiteracyWorkSuccess` adiciona um objeto do tipo `LiteracyWork` a coleção `titles` definida em `Library`. A operação de inclusão de uma obra literária é definida no Quadro 132, no *schema* `AddLiteracyWork`. A inclusão é realizada caso não exista uma obra literária cadastrada ou ocorre erro caso exista uma obra literária cadastrada.



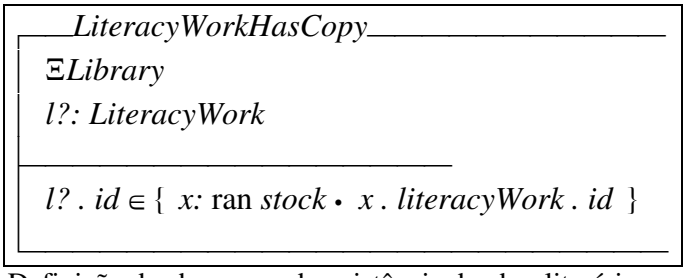
Quadro 131 – Definição de inclusão de obra literária na biblioteca



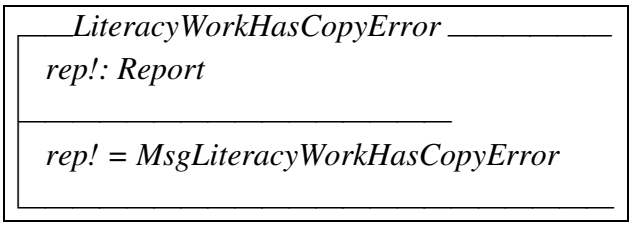
Quadro 132 – Definição completa de inclusão de obra literária na biblioteca

No Quadro 133 é descrita a especificação de verificação da existência de uma obra literária associada a uma cópia na biblioteca. O *schema* `LiteracyWorkHasCopy` verifica se o `ID` do objeto do tipo `LiteracyWork` está referenciado por `literacyWork` da coleção de `stock`.

No Quadro 134 é descrita a especificação de erro ao encontrar uma obra literária associada a uma cópia na biblioteca. O *schema* `LiteracyWorkHasCopyError` retorna a mensagem de erro `rep!` do tipo `Report`.

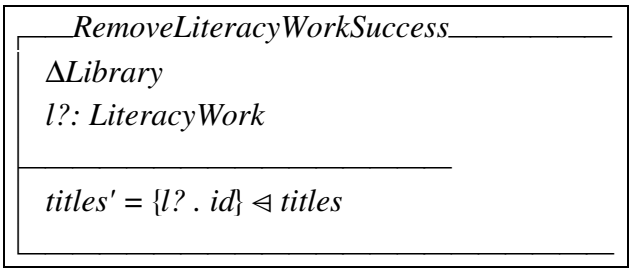


Quadro 133 – Definição de checagem de existência de obra literária associada a cópia



Quadro 134 – Definição de checagem de erro ao existir obra literária associada a cópia

No Quadro 135 é descrita a especificação de remoção com sucesso de uma obra literária da biblioteca. O *schema* `RemoveLiteracyWorkSuccess` remove um objeto do tipo `LiteracyWork` da coleção `titles` definida em `Library`. A operação de remoção de uma obra literária é definida no Quadro 136, no *schema* `RemoveLiteracyWork`. A remoção é realizada caso exista uma obra literária cadastrada e esta não tenha cópia associada. Ocorre erro caso não exista uma obra literária ou possua cópia associada.



Quadro 135 – Definição de remoção de obra literária na biblioteca

$$RemoveLiteracyWork \mathrel{\widehat=} \neg\, LiteracyWorkHasCopy \land LiteracyWorkExistsCheck \land RemoveLiteracyWorkSuccess \lor LiteracyWorkNotExistsCheck \land LiteracyWorkNotExistsError \lor LiteracyWorkHasCopy \land LiteracyWorkHasCopyError$$

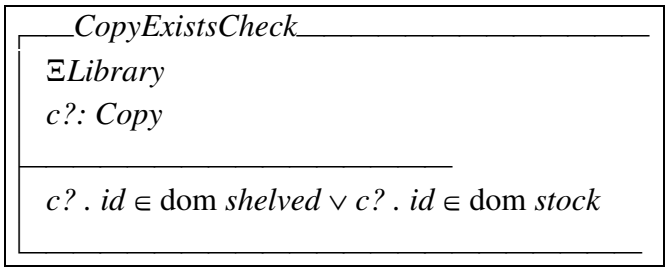
Quadro 136 – Definição completa de remoção de obra literária na biblioteca

No Quadro 137 é descrita a especificação de verificação da existência de uma cópia na biblioteca. O *schema* `CopyExistsCheck` verifica se um objeto do tipo `Copy` contém um `ID` que pertença a coleção `shelved` ou `stock` definida em `Library`. Os que estão disponíveis estão na coleção `shelved` (prateleira). Todos os existentes estão na coleção `stock` (estoque).

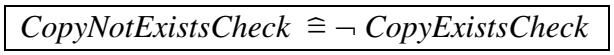
No Quadro 138 é descrita a especificação de verificação da não existência de uma cópia na biblioteca. O *schema* `CopyNotExistsCheck` verifica se um objeto do tipo `Copy` contém um `ID` que não pertença a coleção `shelved` ou `stock` definida em `Library`.

No Quadro 139 é descrita a especificação de erro ao encontrar uma cópia cadastrada na biblioteca. O *schema* `CopyExistsError` retorna a mensagem de erro `rep!` do tipo `Report`.

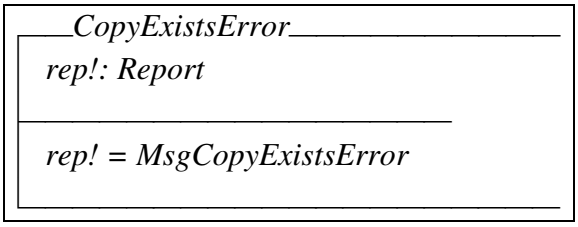
No Quadro 140 é descrita a especificação de erro ao não encontrar uma cópia cadastrada na biblioteca. O *schema* `CopyNotExistsError` retorna a mensagem de erro `rep!` do tipo `Report`.



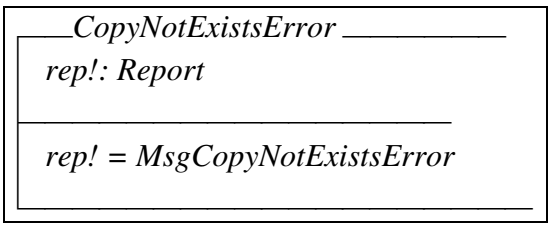
Quadro 137 – Definição de checagem de existência de cópia



Quadro 138 – Definição de checagem de não existência de cópia

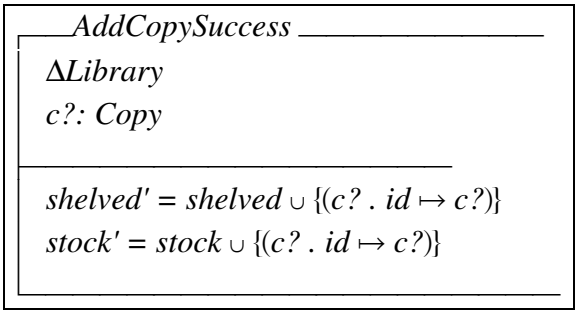


Quadro 139 – Definição de erro de cópia existente

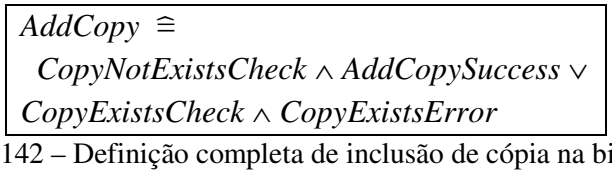


Quadro 140 – Definição de erro de cópia não existente

No Quadro 141 é descrita a especificação de adição com sucesso de uma cópia a biblioteca. O *schema* `AddCopySuccess` adiciona um objeto do tipo `Copy` a coleção `shelved` e `stock` definida em `Library`. A operação de inclusão de uma cópia é definida no Quadro 142, no *schema* `AddCopy`. A inclusão é realizada caso não exista uma cópia cadastrada ou ocorre erro caso exista uma cópia cadastrada.



Quadro 141 – Definição de inclusão de cópia na biblioteca

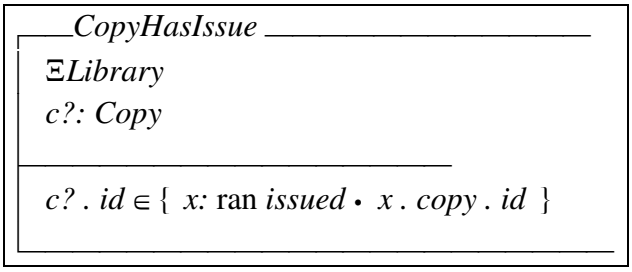


Quadro 142 – Definição completa de inclusão de cópia na biblioteca

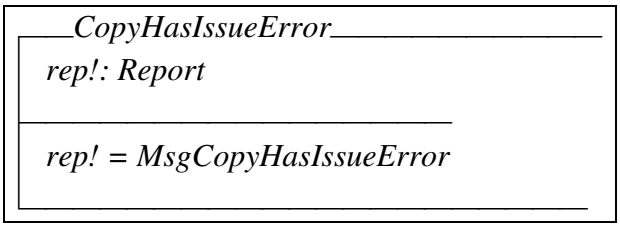
No Quadro 143 é descrita a especificação de verificação da existência de um leitor

associado a um empréstimo na biblioteca. O *schema* `CopyHasIssue` verifica se o `ID` do objeto do tipo `Copy` está referenciado por `copy` da coleção de `issued`.

No Quadro 144 é descrita a especificação de erro ao encontrar um leitor associado a um empréstimo na biblioteca. O *schema* `CopyHasIssueError` retorna a mensagem de erro `rep!` do tipo `Report`.



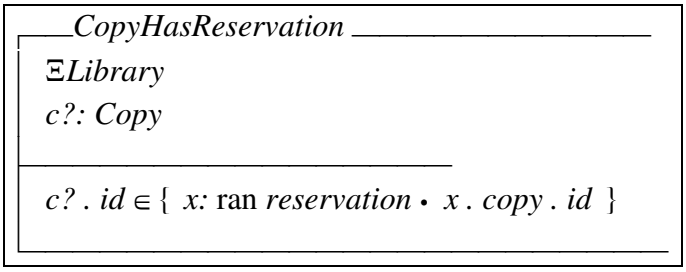
Quadro 143 – Definição de checagem de existência de cópia associada a empréstimo



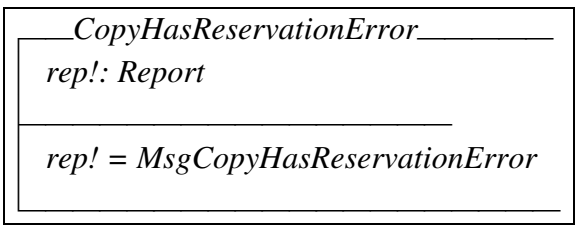
Quadro 144 – Definição de erro ao existir cópia associada a empréstimo

No Quadro 145 é descrita a especificação de verificação da existência de um leitor associado a uma reserva na biblioteca. O *schema* `CopyHasReservation` verifica se o `ID` do objeto do tipo `Copy` está referenciado por `copy` da coleção de `reservation`.

No Quadro 146 é descrita a especificação de erro ao encontrar um leitor associado a uma reserva na biblioteca. O *schema* `CopyHasReservationError` retorna a mensagem de erro `rep!` do tipo `Report`.



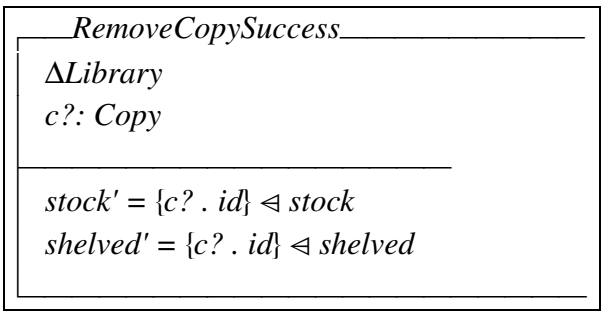
Quadro 145 – Definição de checagem de existência de cópia associada a reserva



Quadro 146 – Definição de erro ao existir cópia associada a reserva

No Quadro 147 é descrita a especificação de remoção com sucesso de uma cópia da

biblioteca. O *schema* `RemoveCopySuccess` remove um objeto do tipo `Copy` da coleção `shelved` e `stock` definida em `Library`. A operação de remoção de uma cópia é definida no Quadro 148, no *schema* `RemoveCopy`. A remoção é realizada caso exista uma cópia cadastrada, não tenha empréstimo ativo, nem reserva. Ocorre erro caso não exista uma cópia, exista empréstimo ativo ou reserva associada a cópia.

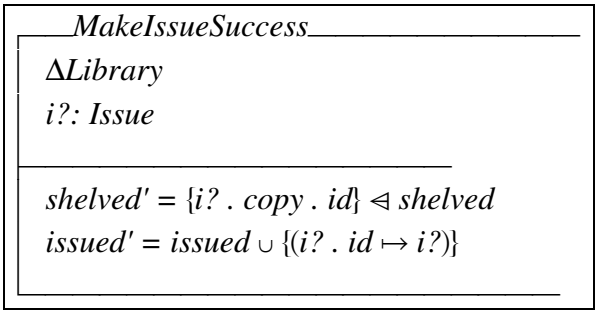


Quadro 147 – Definição de remoção de cópia na biblioteca

$$
\begin{aligned}
&\mathit{RemoveCopy} \mathrel{\widehat=} \\
&\quad \neg\, \mathit{CopyHasReservation} \land \neg\, \mathit{CopyHasIssue} \land \mathit{CopyExistsCheck} \land \mathit{RemoveCopySuccess} \\
&\quad \lor\, \mathit{CopyNotExistsCheck} \land \mathit{CopyNotExistsError} \\
&\quad \lor\, \mathit{CopyHasReservation} \land \mathit{CopyHasReservationError} \\
&\quad \lor\, \mathit{CopyHasIssue} \land \mathit{CopyHasIssueError}
\end{aligned}
$$

Quadro 148 – Definição completa de remoção de cópia na biblioteca

No Quadro 149 é descrita a especificação de efetivação de um empréstimo. O *schema* `MakeIssueSuccess` retira um objeto do tipo `Copy` da coleção `shelved` e adiciona um objeto do tipo `Issue` a coleção `issued`.



Quadro 149 – Definição de empréstimo

No Quadro 150 é descrita a especificação do *schema* que faz as checagens para realizar um empréstimo. Ao realizar um empréstimo a cópia deve existir no estoque e estar na prateleira. A cópia não pode estar em empréstimo, nem reservada. O leitor e o bibliotecário devem estar cadastrados na biblioteca. O número de locações do leitor não pode exceder o máximo de locações do respectivo tipo do leitor.

CanMakeIssue

$\Delta Library$
$i?: Issue$

$i?.copy.id \in \text{dom } stock$
$i?.copy.id \in \text{dom } shelved$
$i?.copy.id \notin \{ x: \text{ran } issued \bullet x.copy.id \}$
$i?.copy.id \notin \{ x: \text{ran } reservation \bullet x.copy.id \}$
$i?.reader.id \in \text{dom } readers$
$i?.librarian.id \in \text{dom } librarians$
$i?.reader.type.maxLoans$
$> \#\{ x: \text{ran } issued \mid x.reader.id = i?.reader.id \}$

Quadro 150 – Definição de checagens ao efetuar um empréstimo

No Quadro 151 é descrita a especificação do *schema* que verifica as situações de erro ao realizar um empréstimo. As situações de erro são:

a) cópia não pertencente ao estoque;
b) cópia não presente na prateleira;
c) cópia em empréstimo;
d) leitor não cadastrado;
e) bibliotecário não cadastrado;
f) número de empréstimo superior ao máximo definido pelo tipo do leitor.

MakeIssueError

$\Delta Library$
$i?: Issue$
$rep!: Report$

$rep! = MsgCopyNotExistsError \Leftrightarrow i?.copy.id \notin \text{dom } stock$
$rep! = MsgCopyNotAvailableError \Leftrightarrow i?.copy.id \notin \text{dom } shelved$
$rep! = MsgCopyIsIssuedError \Leftrightarrow i?.copy.id \in \{ x: \text{ran } issued \bullet x.copy.id \}$
$rep! = MsgReaderNotExistsError \Leftrightarrow i?.reader.id \notin \text{dom } readers$
$rep! = MsgLibrarianNotExistsError \Leftrightarrow i?.librarian.id \notin \text{dom } librarians$
$rep! = MsgReaderHasMaxLoansError \Leftrightarrow i?.reader.type.maxLoans$
$\leqslant \#\{ x: \text{ran } issued \mid x.reader.id = i?.reader.id \}$

Quadro 151 – Definição de erro ao efetuar um empréstimo

No Quadro 152 é descrita a especificação do *schema* `MakeIssueFromReservation` que define um empréstimo de uma cópia reservada pelo próprio usuário. É necessário que a cópia que está sendo emprestada seja a mesma que o leitor efetuou a reserva.

MakeIssueFromReservation

$\Delta Library$

$i?: Issue$

$i?\ .\ copy\ .\ id$

$\in \{\ x:$ ran $reservation$

$\quad \mid x\ .\ reader\ .\ id = i?\ .\ reader\ .\ id \land x\ .\ copy\ .\ id = i?\ .\ copy\ .\ id$

$\quad \bullet\ x\ .\ copy\ .\ id\ \}$

$reservation'$

$= \{\ x:$ ran $reservation$

$\quad \mid x\ .\ reader\ .\ id = i?\ .\ reader\ .\ id \land x\ .\ copy\ .\ id = i?\ .\ copy\ .\ id$

$\quad \bullet\ x\ .\ id\ \}$

$\lhd\ reservation$

Quadro 152 – Definição de empréstimo a partir de uma reserva do leitor

No Quadro 153 é descrita a especificação do *schema* `MakeIssueFromReservationError` que define o erro que ocorre ao efetuar o empréstimo de uma cópia reservada. Ocorre erro ao identificar que a cópia que se deseja ser emprestada esteja reservada a outro usuário.

MakeIssueReservationError

$\Delta Library$

$i?: Issue$

$rep!: Report$

$i?\ .\ copy\ .\ id$

$\in \{\ x:$ ran $reservation$

$\quad \mid x\ .\ reader\ .\ id = i?\ .\ reader\ .\ id \land x\ .\ copy\ .\ id = i?\ .\ copy\ .\ id$

$\quad \bullet\ x\ .\ copy\ .\ id\ \}$

$rep! = MsgCopyIsReservedError$

Quadro 153 – Definição de erro ao realizar um empréstimo a partir de uma reserva do leitor

No Quadro 154 é descrita a especificação do *schema* `MakeIssue` que define o empréstimo como um todo. O processo de empréstimo pode ocorrer pela efetivação do empréstimo sem obra reservada, pelo empréstimo de uma obra que esteja reservada para este leitor ou pela ocorrência de erro.

$$
\begin{aligned}
&MakeIssue \mathrel{\widehat=} \\
&\quad MakeIssueFromReservation \land CanMakeIssue \land MakeIssueSuccess \\
&\quad \lor CanMakeIssue \land MakeIssueSuccess \\
&\quad \lor MakeIssueReservationError \\
&\quad \lor MakeIssueError
\end{aligned}
$$

Quadro 154 – Definição completa de empréstimo

No Quadro 155 é descrita a especificação de efetivação de uma reserva. O *schema* `MakeReservationSuccess` adiciona um objeto do tipo `Reservation` a coleção `reservation`.

MakeReservationSuccess

$$
\begin{aligned}
&\Delta Library \\
&r?: Reservation \\
&\hline
&reservation' = reservation \cup \{(r?\,.\,id \mapsto r?)\}
\end{aligned}
$$

Quadro 155 – Definição de reserva

No Quadro 156 é descrita a especificação do *schema* que faz as checagens para realizar uma reserva. Ao realizar uma reserva a cópia deve existir no estoque e estar indisponível para empréstimo (não está na prateleira). A cópia tem que estar em empréstimo, mas não pode estar reservada. O leitor e o bibliotecário devem estar cadastrados na biblioteca. O número de locações do leitor não pode exceder o máximo de locações do respectivo tipo do leitor.

CanMakeReservation

$$
\begin{aligned}
&\Delta Library \\
&r?: Reservation \\
&\hline
&r?\,.\,copy\,.\,id \in \mathrm{dom}\ stock \\
&r?\,.\,copy\,.\,id \notin \mathrm{dom}\ shelved \\
&r?\,.\,copy\,.\,id \in \{\ x: \mathrm{ran}\ issued \bullet x\,.\,copy\,.\,id\ \} \\
&r?\,.\,copy\,.\,id \notin \{\ x: \mathrm{ran}\ reservation \bullet x\,.\,copy\,.\,id\ \} \\
&r?\,.\,reader\,.\,id \in \mathrm{dom}\ readers \\
&r?\,.\,librarian\,.\,id \in \mathrm{dom}\ librarians \\
&r?\,.\,reader\,.\,type\,.\,maxLoans \\
&\quad > \#\{\ x: \mathrm{ran}\ issued \mid x\,.\,reader\,.\,id = r?\,.\,reader\,.\,id\ \}
\end{aligned}
$$

Quadro 156 – Definição de checagens ao efetuar uma reserva

No Quadro 157 é descrita a especificação do *schema* que verifica as situações de erro ao realizar uma reserva. As situações de erro são:

a) cópia não pertencente ao estoque;

b) cópia está presente na prateleira;

c) cópia não está em empréstimo;

d) cópia está reservada;

e) leitor não cadastrado;

f) bibliotecário não cadastrado;

g) número de empréstimo superior ao máximo definido pelo tipo do leitor.

MakeReservationError

$\Delta Library$
r?: Reservation
rep!: Report

$rep! = MsgCopyNotExistsError \Leftrightarrow r?.copy.id \notin \text{dom } stock$
$rep! = MsgCopyIsAvailableError \Leftrightarrow r?.copy.id \in \text{dom } shelved$
$rep! = MsgCopyIsNotIssuedError \Leftrightarrow r?.copy.id \notin \{ x: \text{ran } issued \bullet x.copy.id \}$
$rep! = MsgCopyIsReserverError \Leftrightarrow r?.copy.id \in \{ x: \text{ran } reservation \bullet x.copy.id \}$
$rep! = MsgReaderNotExistsError \Leftrightarrow r?.reader.id \notin \text{dom } readers$
$rep! = MsgLibrarianNotExistsError \Leftrightarrow r?.librarian.id \notin \text{dom } librarians$
$rep! = MsgReaderHasMaxLoansError \Leftrightarrow r?.reader.type.maxLoans \leqslant \#\{ x: \text{ran } issued \mid x.reader.id = r?.reader.id \}$

Quadro 157 – Definição de erro ao efetuar uma reserva

No Quadro 158 é descrita a especificação do *schema* `MakeReservation` que define a reserva como um todo. O processo de reserva pode ocorrer pela reserva com sucesso da obra a ser reservada, ou pela ocorrência de erro.

$MakeReservation \mathrel{\widehat=}$
$CanMakeReservation \land MakeReservationSuccess \lor$
$MakeReservationError$

Quadro 158 – Definição completa de reserva

No Quadro 159 é descrita a especificação de efetivação de retorno de um empréstimo. O *schema* `ReturnIssueSuccess` adiciona um objeto do tipo `Copy` a coleção `shelved` e remove um objeto do tipo `Issue` da coleção `issued`.

ReturnIssueSuccess

$\Delta Library$
i?: Issue

$shelved' = shelved \cup \{(i?.copy.id \mapsto i?.copy)\}$
$issued' = \{i?.id\} \lhd issued$

Quadro 159 – Definição de retorno de empréstimo

No Quadro 160 é descrita a especificação do *schema* faz as checagens para realizar o

retorno de um empréstimo. Ao realizar o retorno de um empréstimo a cópia deve existir no estoque e não estar na prateleira. A cópia deve estar em empréstimo. O leitor deve estar cadastrados na biblioteca.

CanReturnIssue

$\Xi Library$
$i?: Issue$

$i?.copy.id \in \text{dom } stock$
$i?.copy.id \notin \text{dom } shelved$
$i?.copy.id \in \{ x: \text{ran } issued \bullet x.copy.id \}$
$i?.reader.id \in \text{dom } readers$

Quadro 160 – Definição de checagens ao efetuar retorno de empréstimo

No Quadro 161 é descrita a especificação do *schema* que verifica as situações de erro ao realizar o retorno de um empréstimo. As situações de erro são:

a) cópia não pertencente ao estoque;
b) cópia está presente na prateleira;
c) cópia não está em empréstimo;
d) leitor não cadastrado.

ReturnIssueError

$\Xi Library$
$i?: Issue$
$rep!: Report$

$rep! = MsgCopyNotExistsError \Leftrightarrow i?.copy.id \notin \text{dom } stock$
$rep! = MsgCopyIsAvailableError \Leftrightarrow i?.copy.id \in \text{dom } shelved$
$rep! = MsgCopyIsNotIssuedError \Leftrightarrow i?.copy.id \notin \{ x: \text{ran } issued \bullet x.copy.id \}$
$rep! = MsgReaderNotExistsError \Leftrightarrow i?.reader.id \notin \text{dom } readers$

Quadro 161 – Definição de erro ao efetuar retorno de empréstimo

No Quadro 162 é descrita a especificação do *schema* `ReturnIssue` que define o retorno de empréstimo como um todo. O processo de retorno de empréstimo pode ocorrer pela efetivação do retorno do empréstimo ou pela ocorrência de erro.

$ReturnIssue \mathrel{\widehat=} CanReturnIssue \land ReturnIssueSuccess \lor ReturnIssueError$

Quadro 162 – Definição completa de retorno de empréstimo

No Quadro 163 é descrita a especificação de efetivação de cancelamento de uma reserva. O *schema* `CancelReservationSuccess` remove um objeto do tipo `Reservation` da

coleção `reservation`.

CancelReservationSuccess

$\Delta Library$

$r?: Reservation$

$reservation' = \{r?\ .\ id\} \lhd reservation$

Quadro 163 – Definição de cancelamento de reserva

No Quadro 164 é descrita a especificação do *schema* que faz as checagens para realizar o cancelamento de uma reserva. Ao realizar o cancelamento de uma reserva a cópia deve estar reservada. O leitor deve estar cadastrado na biblioteca.

CanCancelReservation

$\Xi Library$

$r?: Reservation$

$i?\ .\ copy\ .\ id \in \{\ x: \text{ran}\ reservation \bullet\ x\ .\ copy\ .\ id\ \}$

$i?\ .\ reader\ .\ id \in \text{dom}\ readers$

Quadro 164 – Definição de checagens ao efetuar o cancelamento de reserva

No Quadro 165 é descrita a especificação do *schema* que verifica as situações de erro ao realizar uma reserva. As situações de erro são:

a) cópia está reservada;

b) leitor não cadastrado.

CancelReservationError

$\Xi Library$

$r?: Reservation$

$rep!: Report$

$rep! = MsgCopyIsReservedError \Leftrightarrow r?\ .\ copy\ .\ id \notin \{\ x: \text{ran}\ reservation \bullet\ x\ .\ copy\ .\ id\ \}$

$rep! = MsgReaderNotExistsError \Leftrightarrow r?\ .\ reader\ .\ id \notin \text{dom}\ readers$

Quadro 165 – Definição de erro ao efetuar o cancelamento de reserva

No Quadro 166 é descrita a especificação do *schema* `CancelReservation` que define o cancelamento de reserva como um todo. O processo de cancelamento de reserva pode ocorrer pelo cancelamento da reserva com sucesso, ou pela ocorrência de erro.

$$CancelReservation \mathrel{\widehat=} CanCancelReservation \land CancelReservationSuccess \lor CancelReservationError$$

Quadro 166 – Definição completa de cancelamento de reserva

No Quadro 167 é descrito a implementação de obtenção das cópias em empréstimo. É retornado um conjunto finito do tipo `Copy` na variável `result`. Na variável está contida todas as cópias presentes em `copy` da coleção `issued` definida na biblioteca.

$$\begin{array}{|l} \textit{ReturnIssuedCopies} \\ \hline \Xi Library \\ result! : \mathbb{F}\ Copy \\ \hline result! = \{\ x : \text{ran}\ issued \bullet x.copy\ \} \\ \hline \end{array}$$

Quadro 167 – Definição de obtenção de cópias em empréstimo

No Quadro 168 é descrito a implementação de obtenção das cópias em atraso. É retornado um conjunto finito do tipo `Copy` na variável `result`. Na variável está contida todas as cópias presentes em `copy` da coleção `issued` definida na biblioteca, onde a data de devolução do empréstimo é menor que a data definida na variável `today`.

$$\begin{array}{|l} \textit{ReturnOverdueCopies} \\ \hline \Xi Library \\ today : Date \\ result! : \mathbb{F}\ Copy \\ \hline result! = \{\ x : \text{ran}\ issued \mid x.returnDate.value < today.value \bullet x.copy\ \} \\ \hline \end{array}$$

Quadro 168 – Definição de cópias em atraso

No Quadro 169 é descrito as situações que uma cópia pode estar. Uma cópia pode estar emprestada, reservada, disponível ou em situação de erro desconhecido.

$$CopyStatus ::= Issued \mid Reserved \mid Available \mid Unknow$$

Quadro 169 – Definição de estados possíveis de uma cópia

No Quadro 170 é descrito a especficação de obtenção de estado de cópia. É retornado o estado da cópia definido em `CopyStatus` na variável `result`.

ReturnCopyStatus

$\Xi Library$
$c?: Copy$
$result!: CopyStatus$

$c? . id \in \text{dom } issued \Rightarrow result! = Issued \lor$
$c? . id \in \text{dom } shelved \Rightarrow result! = Available \lor$
$c? . id \in \text{dom } reservation \Rightarrow result! = Reserved \lor$
$result! = Unknow$

Quadro 170 – Definição de estado em que se encontra uma cópia

3.2.2 Provas de propriedades

Potter, Sinclar e Till (1996, p. 312) afirmam que a introdução ao Z está naturalmente definida em três fases. Na primeira fase as especificações aprendidas e aplicadas, inicialmente em problemas pequenos e já conhecidos, num projeto piloto por exemplo. Na segunda fase Z é aplicado em maior escala utilizando de conceitos mais complexos, como cálculo de pré-condições, por exemplo, torna-se mais frutífero a utilização da formalidade. No fim da segunda fase a codificação torna-se mais rigorosa, o que faz valer a pena o tempo despendido na especificação. Finalmente na terceira fase é obtido o foco na rigorosidade da especificação, onde provas começam a ser construídas. Construir provas deve ser algo meticulosamente estudado. Provas são trabalhosas para serem desenvolvidas e se não houver domínio na sua utilização seu custo pode ser alto. No caso dos sistemas críticos, como em sistemas de aviação ou de usinas nucelares, o custo não vem em primeiro lugar, mas sim a segurança, o que remete a utilizar provas ignorando trabalho e custo em virtude de ganhos maiores.

Não foi utilizado provas de propriedades de uma forma ampla no desenvolvimento do estudo de caso, porém é demonstrado um exemplo de prova na especificação construída, a qual pode ser vista no Quadro 171. O *schema* de prova chamado de `theorem` define neste quadro que no *schema* `AddLibrarian` o retorno em `rep!` será `MsgLibrarianExistsError` sempre que existir um bibliotecário já cadastrado com este `id`. No Quadro 172 é apresentado um pedaço da prova gerada pelo Z/EVES. No Apêndice B é demonstrada a prova completa gerada pelo Z/EVES. Na Figura 4 é apresentada a execução da prova no ambiente Z/EVES.

theorem *AddLibrarianNotOk*
$\forall AddLibrarian \mid l? . id \in \text{dom } librarians \bullet rep! = MsgLibrarianExistsError$

Quadro 171 – Definição de teorema

$rep! \in Report$
$\land\ l? . id \in \text{dom } librarians$
...
$\land\ rep! = MsgLibrarianExistsError$
...
$\Rightarrow rep! = MsgLibrarianExistsError$

Quadro 172 – Pedaço da prova gerada pelo Z/EVES a partir do teorema

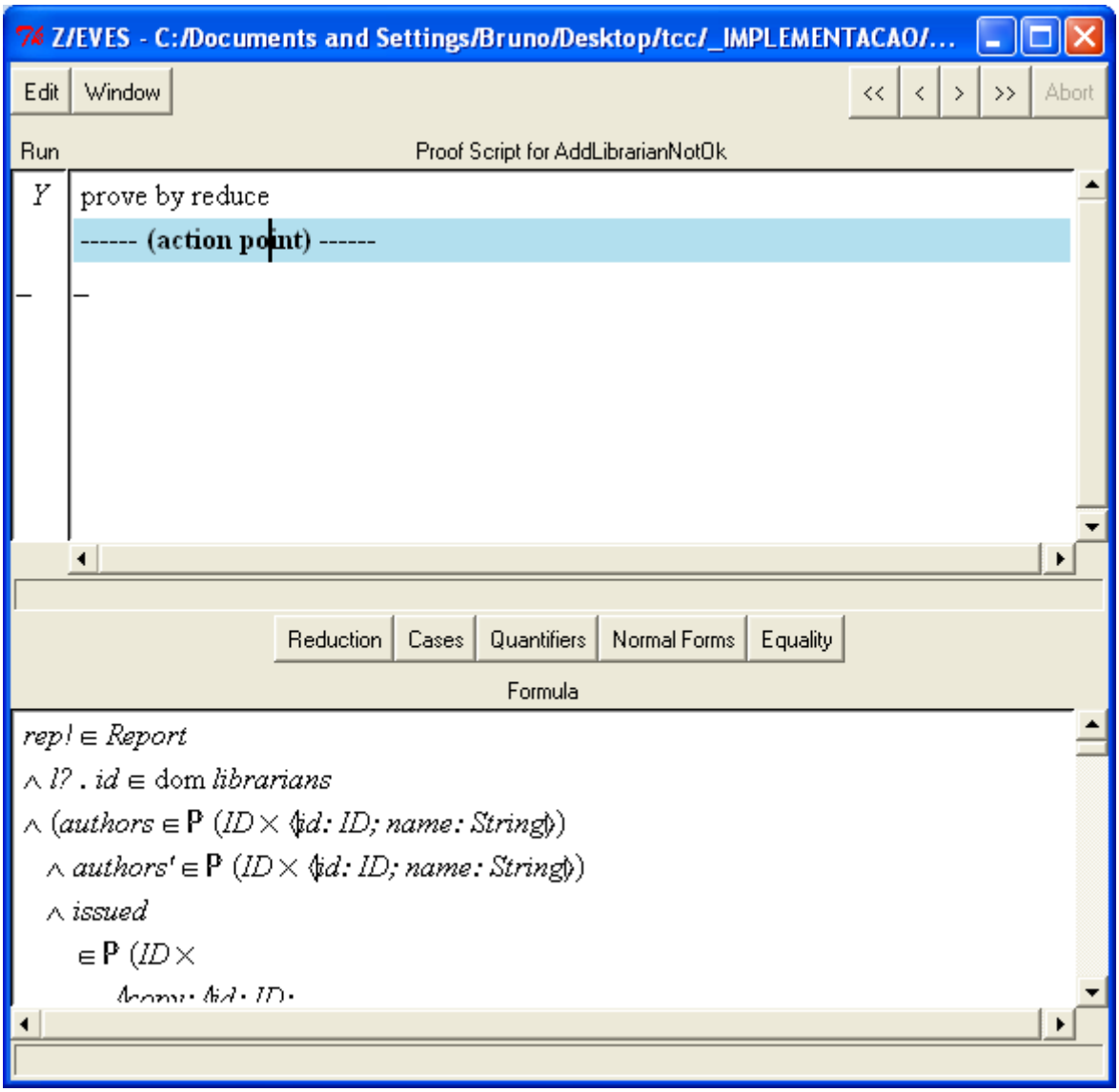


Figura 4 - Execução da prova no Z/EVES

3.3 IMPLEMENTAÇÃO

A implementação apresentada nesta seção demonstra como foi realizada a codificação de toda a especificação realizada em Z.

3.3.1 Técnicas e ferramentas utilizadas

A implementação da especificação foi escrita na linguagem Java utilizando o ambiente de programação Eclipse. Os testes realizados sobre a especificação e implementação foram escritos utilizando JUnit. Para a construção das telas foi utilizado uma classe genérica, utilizando a construção de *generics* disponível a partir da versão 1.5 do Java. Para o mapeamento de coleções foi utilizada a classe `ZSet` e para o mapeamento de relações foi utilizada a classe `ZRelation`, ambas presente no Anexo A. Por fim, as classes construídas no estudo de caso estão presentes no Apêndice A.

3.3.2 Mapeamento das construções em Z para Java

Para demonstrar a conversão das construções da especificação realizadas em Z para a linguagem Java se faz necessário realizar um mapeamento entre as duas tecnologias. São apresentadas apenas as construções utilizadas na especificação do sistema de biblioteca. É apresentado o mapeamento das estruturas de uma forma genérica e após o mapeamento de partes da especificação para a respectiva implementação em Java.

3.3.2.1 Mapeamentos das estruturas

No Quadro 173 é apresentado o mapeamento dos tipos definidos em Z para Java. Os tipos apresentados são os utilizados na especificação do sistema. Alguns tipos possuem seu mapeamento para uma estrutura simples, porém tipos complexos definidos em *schemas* necessitam de uma implementação mais detalhada em Java.

Z	Descrição	Java

`ID`	Este tipo não foi detalhado na especificação. Poderia ser transcrito para qualquer construção.	Criada uma classe `ID` com valor interno do tipo `int`. Realizada checagem do valor, não pode ser menor que zero.
		`if(value < 0){`
		`  throw new IllegalArgumentException(`
		`  "Id must be greater than zero.");`
		`}`

`String`	Este tipo não foi detalhado na especificação. Poderia ser transcrito para qualquer construção.	`String`
$\mathbb{N}$	Naturais, os quais são os inteiros maiores que zero.	`int`. Realiza a mesma checagem de valor que a transformação transcrita em `ID`.
$\mathbb{F}\,\mathbb{N}$	Conjunto finito de naturais.	`ZSet<Integer> set;`
$X \rightarrowtail\!\!\!\!\!\rightarrow Y$	Função bijetora	`ZRelation<X,Y> rel;`
Outros tipos. Ex: (`Reader`, `Author`)	Tipos construídos na especificação.	Criada uma classe própria para cada tipo

Quadro 173 – Mapeamento dos tipos definido em Z para Java

No Quadro 174 é apresentado o mapeamento da definição livre definido em Z para Java.

Z	Java
`a ::= b`	Criada uma enumeração (`enum`) com nome a.

`A ::=`
`  | a`
`  | b`
`  | c`
`  | d`
	Criado os elementos da enumeração com o nome de cada elemento que está separado pelo caractere |.
	
	`enum A {`
	`  a,`
	`  b,`
	`  c,`
	`  d;`
	`}`

Quadro 174 – Mapeamento da definição livre definido em Z para Java

No Quadro 175 é apresentado o mapeamento de definição axiomática definido em Z para Java.

Z	Java

`a`
`b`
	Criada uma variável `a` com a restrição imposta em b, sendo a do tipo `A` e aplicando uma restrição b qualquer.
	
	`A a;`
	`if(b){`
	`  //do something;`
	`}`

Quadro 175 – Mapeamento de definição axiomática definido em Z para Java

No Quadro 176 é apresentado o mapeamento dos símbolos relacionais definido em Z para Java.

Z	Java
*	*
+	+
−	−
Mod	%
⩾	>=
⩽	<=
<	<
>	>
=	=
≠	!=
∨	||
∧	&&
¬	!

a ⇔ b	if(a){
	  assert b;
	}

a ⇒ b	if(a) {
	  b;
	}

Quadro 176 – Mapeamento dos símbolos relacionais definido em Z para Java

No Quadro 177 é apresentado o mapeamento de declaração de *schemas* simples definido em Z para Java. Esta construção é a base dos tipos definidos na especificação construída e apresentada.

Z	Java

SchemaA
a: A
b: B
	public class SchemaA {
	  public A a;
	  public B b;
	}

Quadro 177 – Mapeamento de declaração de *schemas* simples definido em Z para Java

No Quadro 178 é apresentado o mapeamento de *schemas* derivados definido em Z para Java. Este mapeamento é utilizado para mapear as classes derivadas na especificação. Este conceito foi traduzido com a utilização de herança em Java.

Z	Java

SchemaB
SchemaA
c: C
	public class SchemaB extends SchemaA{
	  public C c;
	}

Quadro 178 – Mapeamento de *schemas* derivados definido em Z para Java

No Quadro 179 é apresentado o mapeamento de utilização de um *schema* definido em Z para Java. Esta utilização de *schema*, como inicializador das variáveis, remete ao conceito de construtor em Java.

Z	Java

CreateSchemaA
ΔSchemaA
a?: A
b?:B
--- 	
a' = a?
b' = b?
	`public SchemaA(A a, B b) {`
	`  this.a = a;`
	`  this.b = b;`
	`}`

Quadro 179 – Mapeamento de utilização de um *schema* definido em Z para Java

No Quadro 180 é apresentado o mapeamento de utilização de um *schema* derivado definido em Z para Java. Aqui a junção dos *schemas* derivados e inicialização de ambas as partes, remete ao conceito de construtor com classe superior.

Z	Java

CreateSchemaBPart
ΔSchemaB
c?: C
--- 	
c' = c?
--- 	
CreateB ≙ *CreateSchemaBPart* ∧ *CreateSchemaA*
	`public SchemaB(A a, B b, C c) {`
	`  super(a, b);`
	`  this.c = c;`
	`}`

Quadro 180 – Mapeamento de utilização de um *schema* derivado definido em Z para Java

No Quadro 181 é apresentado o mapeamento de utilização de retorno definido em Z para Java. Um *schema*, como neste mapeamento, pode ser mapeado para um método. Caso exista um retorno (variável decorada com o caracter exclamação (!)), este é o retorno do método. As variáveis de entrada são os parâmetros.

Z	Java

ReturnA
schema?: Schema
ret! : A
--- 	
ret! = schema?.a
	`public A returnA(Schema schema) {`
	`  return schema.a;`
	`}`

Quadro 181 – Mapeamento de utilização de retorno definido em Z para Java

No Quadro 182 é apresentado o mapeamento dos operadores sobre conjuntos definido em Z para Java. Os conjuntos apresentados são as coleções e as relações. Cada símbolo é relacionado com seu respectivo mapeamento em Java.

Z	Java (Coleção)	Java (Relação)
$\in$	`ZSet set;` `set.contains(a);`	`ZRelation rel;` `rel.contains(a,b)`
$\notin$	`ZSet set;` `!set.contains(a);`	`ZRelation rel;` `!rel.contains(a,b);`
$\cup$	`ZSet set;` `set.union(a, b);`	`ZRelation rel;` `rel.union(a, b);`
{}	`ZSet set = new BasicSet();`	`ZRelation rel = new BasicPFunction();`
#	`ZSet set;` `set.size();`	`ZRelation rel;` `set.size();`

Quadro 182 – Mapeamento dos operadores sobre conjuntos definido em Z para Java

No Quadro 183 é apresentado o mapeamento dos operadores sobre relações definido em Z para Java.

Z	Java
$\mapsto$	`ZRelation rel;` `rel.put(a, b);`
$\lhd$	`ZRelation rel;` `rel = rel.domain_restriction(a, b);`
dom	`ZRelation rel;` `rel.domain();`
ran	`ZRelation rel;` `rel.range();`

{ a: b | c }
(É desejado retornar um objeto `a` do tipo `b` com restições em `c`.)
	Considerando a um objeto do tipo `ZRelation<X,Y>` (tipo `b`). Aplicando-se uma restrição `c` qualquer. Adiconar dentro da construção condicional `c` as regras para manipulação do conjunto.
	`private ZRelation<X,Y> methodX(ZRelation<X,Y> a){`
	`  if(c){`
	`  //do something;`
	`  }`
	`  return a;`
	`}`

{ a: b • c }
(É desejado retornar um objeto `c` a partir de um objeto `a` do tipo `b`)
	Considerando `a` um objeto do tipo `B` (tipo `b`). Retornando um objeto do tipo `C` (objeto `c`).
	`private C methodX(B b){`
	`  return b.c;`
	`}`

{ a: b | c • d }
(É desejado retornar um objeto `d` a partir de um objeto `a` do tipo `b` com restrições em `c`.)
	Considerando `a` um objeto do tipo `ZRelation<B,D>` (tipo `b`). Aplicando uma restrição `c` qualquer e retornando um objeto do tipo `ZSet<D>` (objeto `d`).
	`private ZSet<D> methodX(ZRelation<B,D> a){`
	`  ZSet<D> zd = a.range();`
	`  ZSet<D> ret;`
	`  for (D d : zd) {`
	`  //do something;`
	`  }`
	`  return ret;`
	`}`

Quadro 183 – Mapeamento dos operadores sobre relações definido em Z para Java

3.3.2.2 Implementação dos mapeamentos

A seguir é apresentada a técnica de mapeamento das definições do sistema definidas em Z para Java utilizando o mapeamento demonstrado na seção anterior. Não é apresentado todo o mapeamento da implementação da especificação em Z para a linguagem Java. É apresentado apenas as construções chaves para que se possa compreender como um todo a especificação apresentada.

No Quadro 184 é apresentado o mapeamento da implementação de um tipo livre definido em Z para Java. É apresentado o mapeamento do tipo `Report`, que contém as variáveis de exceção utilizadas na especificação. Nem todas as exceções estão listadas neste quadro. Apenas algumas são mostradas visando demonstrar o mapeamento desta construção. Este mapeamento tem como resultado de sua especificação uma enumeração em Java.

Z	Java

Report ::=
| CopyAlreadyExists
| LiteracyWorkAlreadyExists
| LibrarianAlreadyExists
| ReaderAlreadyExists
	`enum Report {`
	`  CopyAlreadyExists,`
	`  LiteracyWorkAlreadyExists,`
	`  LibrarianAlreadyExists,`
	`  ReaderAlreadyExists;`
	`}`

Quadro 184 – Mapeamento da implementação de um tipo livre definido em Z para Java

No Quadro 185 é apresentado o mapeamento da implementação de uma definição axiomática definido em Z para Java. A definição apresentada é a de ano bissexto. A definição de ano bissexto, realizado na forma de definição axiomática, foi transformada em uma variável que tem seus valores definido em um bloco de código que contemplam a parte predicativa.

Z	Java

leapYear: $\mathbb{F}\ \mathbb{N}$
———————————
$\forall$ *year: leapYear*
• *year* $\geqslant$ 1900
$\wedge$ *year* $\leqslant$ 9999
$\wedge$ *year* mod 4 = 0
$\wedge$ (*year* mod 100 $\neq$ 0
$\vee$ *year* mod 400 = 0)
	`ZSet<Integer> leapYear = new BasicSet<Integer>();`
	
	`for (int year = 1900; year <= 9999; year++) {`
	`  if (year % 4 == 0 &&`
	`  (year % 100 != 0 || year % 400 == 0)) {`
	`  leapYear.add(year);`
	`}`

Quadro 185 – Mapeamento da implementação de uma definição axiomática definido em Z para Java

No Quadro 186 é apresentado o mapeamento da implementação do construtor de data definido em Z para Java. O conceito de definição de tipos e construtor podem ser claramente vistos neste quadro. Nesta especificação, além da declaração do tipo, é definido um conjunto

de valores. Para contemplar as restrições definidas nos tipos definidos, é necessário realizar uma checagem adicional ao instanciar um objeto. Esta checagem foi implementada no construtor do objeto.

Z	Java

day: 1 .. 31
month: 1 .. 12
year: 1900 .. 9999
value: ℕ
	`public class Date {`
	
	`public final int year;`
	`public final int month;`
	`public final int day;`
	`public final int value;`
	
	`public Date(int year, int month, int day) {`
	`this.year = year;`
	`this.month = month;`
	`this.day = day;`
	`checkInput();`
	`checkDate();`
	`value = getValue();`
	`}`
	
	`private void checkInput() {`
	`if (year < 1900 || year > 9999) {`
	`throw new IllegalArgumentException("Invalid value for year: " + year);`
	`}`
	`if (month < 1 || month > 12) {`
	`throw new IllegalArgumentException("Invalid value for month: " + month);`
	`}`
	`if (day < 1 || day > 31) {`
	`throw new IllegalArgumentException("Invalid value for day: " + day);`
	`}`
	`}`

Quadro 186 – Mapeamento da implementação do construtor de data definido em Z para Java

No Quadro 187 é apresentado o mapeamento da checagem dos valores de data definido em Z para Java. A utilização de símbolos relacionais fica bem evidente no mapeamento presente neste quadro. A implementação da checagem de datas é realizada por meio de bloco de código inserido no corpo da classe `Date`. Estas checagens são definidas dentro de um método. A utilização de um método serve apenas para organizar o código.

Z	Java

$month \notin \{2, 4, 6, 9, 11\}$
$\lor month \in \{4, 6, 9, 11\}$
$\land day \leqslant 30$
$\lor month = 2 \land year$
$\notin leapYear \land day \geqslant 28$
$\lor month = 2 \land year$
$\in leapYear \land day \geqslant 29$
	`private void checkDate() {`
	`  ZSet<Integer> setNon31 = new BasicSet<Integer>();`
	`  setNon31.add(2);`
	`  setNon31.add(4);`
	`  setNon31.add(6);`
	`  setNon31.add(9);`
	`  setNon31.add(11);`
	`  ZSet<Integer> set30 = new BasicSet<Integer>();`
	`  set30.add(4);`
	`  set30.add(6);`
	`  set30.add(9);`
	`  set30.add(11);`
	`  if (!setNon31.contains(month)) {`
	`  return;`
	`  } else if (set30.contains(month)) {`
	`  if (day <= 30) {`
	`  return;`
	`  }`
	`  }`
	`  if (month == 2) {`
	`  if (!LeapYear.leapYear.contains(year)) {`
	`  if (day <= 28) {`
	`  return;`
	`  }`
	`  } else if (LeapYear.leapYear.contains(year)) {`
	`  if (day <= 29) {`
	`  return;`
	`  }`
	`  }`
	`  }`
	`  throw new IllegalArgumentException("Invalid Date");`
	`}`

Quadro 187 – Mapeamento da checagem dos valores de data definido em Z para Java

No Quadro 188 é apresentado o mapeamento da variável `value` na definição de data definido em Z para Java. A utilização de símbolos matemáticos e de atribuição é exposta no mapeamento existente neste quadro. A implementação de um valor para uma data é realizada por meio de bloco de código inserido no corpo da classe `Date`. Estas atribuições são definidas dentro de um método. A utilização de um método serve apenas para organizar o código.

Z	Java

value = *value* + (*year* - 1900) * 366 ⇔ *year* ∈ *leapYear*
value = *value* + (*year* - 1900) * 365 ⇔ *year* ∉ *leapYear*
value = *value* + *month* * 30 ⇔ *month* ∈ {4, 6, 9, 11}
value = *value* + *month* * 31 ⇔ *month* ∉ {2, 4, 6, 9, 11}
value = *value* + *month* * 29 ⇔ *month* = 2 ∧ *year* ∈ *leapYear*
value = *value* + *month* * 28 ⇔ *month* = 2 ∧ *year* ∉ *leapYear*
value = *value* + *day*
	`private int getValue() {`
	`  ZSet<Integer> setNon31 = new BasicSet<Integer>();`
	`  setNon31.add(2);`
	`  setNon31.add(4);`
	`  setNon31.add(6);`
	`  setNon31.add(9);`
	`  setNon31.add(11);`
	`  ZSet<Integer> set30 = new BasicSet<Integer>();`
	`  set30.add(4);`
	`  set30.add(6);`
	`  set30.add(9);`
	`  set30.add(11);`
	
	`  int ret = 0;`
	`  if (LeapYear.leapYear.contains(year)) {`
	`  ret = ret + (year - 1900) * 366;`
	`  }`
	`  if (!LeapYear.leapYear.contains(year)) {`
	`  ret = ret + (year - 1900) * 365;`
	`  }`
	`  if (set30.contains(month)) {`
	`  ret = ret + month * 30;`
	`  }`
	`  if (!setNon31.contains(month)) {`
	`  ret = ret + month * 31;`
	`  }`
	`  if (month == 2 &&`
	`LeapYear.leapYear.contains(year)) {`
	`  ret = ret + month * 29;`
	`  }`
	`  if (month == 2 &&`
	`!LeapYear.leapYear.contains(year)) {`
	`  ret = ret + month * 28;`
	`  }`
	`  ret = ret + day;`
	`  return ret;`
	`}`

Quadro 188 – Mapeamento da variável `value` na definição de data definido em Z para Java

Schemas são as construções mais utilizadas na especificação do sistema. O objeto `Copy` será apresentado na especificação, justamente por possui uma maior utilização em todo o sistema e estar relacionado a grande parte das construções utilizadas. Antes de apresentar as implementações de `Copy` em si, é necessário apresentar a classe base `LiteracyWork`. Também é necessário apresentar o *schema* `Library`, pois é onde os objetos do tipo `Copy` são manipulados. No Quadro 189 é apresentado o mapeamento de `LiteracyWork` definido em Z para Java. Esta construção apresenta o mapeamento de um tipo definido em um *schema* para a sua implementação em Java. A implementação é uma classe com seus respectivos atributos. A estrutura utilizada é semelhante a uma `struct` na linguagem C.

Z	Java

LiteracyWork
id: ID
name: String
yearPub: ℕ
authors: ID ⤖ Author
	`public class LiteracyWork {`
	`  public ID id;`
	`  public String name;`
	`  public int yearPub;`
	`  public ZRelation<ID, Author> authors;`
	`}`

Quadro 189 – Mapeamento de `LiteracyWork` definido em Z para Java

No Quadro 190 é apresentado o mapeamento de `Book` definido em Z para Java. `Book` é um tipo derivado de `LiteracyWork`. A respectiva inclusão da classe base na especificação é a inclusão da classe base na classe `Book` em Java.

Z	Java

Book
LiteracyWork
publisher: String
	`public class Book extends LiteracyWork {`
	`  public String publisher;`
	`}`

Quadro 190 – Mapeamento de `Book` definido em Z para Java

No Quadro 191 é apresentado o mapeamento do construtor de `LiteracyWork` definido em Z para Java. O mapeamento define a criação de um construtor onde os parâmetros são dos mesmos tipos dos atributos. Os atributos recebem o valor das variáveis dos parâmetros do construtor.

Z	Java

CreateLiteracyWork
ΔLiteracyWork
id?: ID
name?: String
yearPub?: ℕ
authors?: ID ⤖ Author
———
id' = id?
name' = name?
yearPub' = yearPub?
authors' = authors?
	`public class LiteracyWork {`
	`  public ID id;`
	`  public String name;`
	`  public int yearPub;`
	`  public ZRelation<ID, Author> authors;`
	
	`  public LiteracyWork(ID id, String name, int yearPub, ZRelation<ID, Author> authors) {`
	`    this.id = id;`
	`    this.name = name;`
	`    this.yearPub = yearPub;`
	`    this.authors = authors;`
	`  }`
	`}`

Quadro 191 – Mapeamento do construtor de `LiteracyWork` definido em Z para Java

No Quadro 192 é apresentado o mapeamento do construtor de `Book` definido em Z para Java. Também é realizada a criação de construtor no mapeamento de um tipo derivado, porém ao invés de associar diretamente aos atributos da classe Java é realizada a invocação pelo método `super`.

Z	Java

CreateBookPart
ΔBook
publisher?: String
--- 	
publisher' = publisher?
 	
CreateBook ≙
CreateLiteracyWork ∧
CreateBookPart
	`public class Book extends LiteracyWork {`
	`  public String publisher;`
	
	`  public Book(ID id, String name, int yearPub,`
	`ZRelation<ID, Author> authors, String publisher) {`
	`  super(id, name, yearPub, authors);`
	`  this.publisher = publisher;`
	`  }`
	`}`

Quadro 192 – Mapeamento do construtor de `Book` definido em Z para Java

No Quadro 193 é apresentado o mapeamento do schema `Library` definido em Z para Java. O *schema* `Library` é quem centraliza todos os dados. Aqui são cadastradas, manipuladas e consultadas as cópias de quais estão sendo apresentadas.

Z	Java

Library
readerTypes: ID ⤖ ReaderType
authors: ID ⤖ Author
librarians: ID ⤖ Librarian
stock: ID ⤖ Copy
issued: ID ⤖ Issue
shelved: ID ⤖ Copy
readers: ID ⤖ Reader
titles: ID ⤖ LiteracyWork
reservation: ID ⤖ Reservation
	`public class Library {`
	
	`  public ZRelation<ID, Librarian> librarians;`
	`  public ZRelation<ID, ReaderType> readersTypes;`
	`  public ZRelation<ID, Author> authors;`
	`  public ZRelation<ID, Copy> stock;`
	`  public ZRelation<ID, Issue> issued;`
	`  public ZRelation<ID, Copy> shelved;`
	`  public ZRelation<ID, Reader> readers;`
	`  public ZRelation<ID, LiteracyWork> titles;`
	`  public ZRelation<ID, Issue> overdue;`
	`  public ZRelation<ID, Reservation> reserved;`
	`...`
	`}`

Quadro 193 – Mapeamento do *schema* `Library` definido em Z para Java

No Quadro 194 é apresentado o mapeamento do schema `Copy` definido em Z para Java.

Z	Java

Copy
id: ID
literacyWork: LiteracyWork
	`public class Copy {`
	`  public ID id;`
	`  public LiteracyWork literacyWork;`
	`}`

Quadro 194 – Mapeamento do *schema* `Copy` definido em Z para Java

No Quadro 195 é apresentado o mapeamento do schema `AddCopySuccess` definido em Z para Java. A especificação é mapeada como um método na classe `Library`.

Z	Java

AddCopySuccess
$\Delta Library$
$c?: Copy$
--- 	
$shelved' = shelved \cup \{(c?\ .\ id \mapsto c?)\}$
$stock' = stock \cup \{(c?\ .\ id \mapsto c?)\}$
	`public class Library {`
	`  ...`
	`  private void addCopySuccess(Copy c) {`
	`  shelved.put(c.id, c);`
	`  stock.put(c.id, c);`
	`  }`
	`}`

Quadro 195 – Mapeamento do *schema* `AddCopySuccess` definido em Z para Java

No Quadro 196 é apresentado o mapeamento do schema `CopyExistsCheck` definido em Z para Java. É adicionado um método à classe `Library` que realiza a parte predicativa do *schema* definido. Por convenção, ao mapear a implementação tendo a parte predicativa envolvendo apenas construções lógicas é criado um método com retorno de tipo lógico.

Z	Java

CopyExistsCheck
$\Xi Library$
$c?: Copy$
--- 	
$c?\ .\ id \in \text{dom}\ shelved \lor$
$c?\ .\ id \in \text{dom}\ stock$
	`public class Library {`
	`  ...`
	`  private boolean copyExistsCheck(Copy c) {`
	`  return shelved.domain().contains(c.id) ||`
	`  stock.domain().contains(c.id);`
	`  }`
	`}`

Quadro 196 – Mapeamento do *schema* `CopyExistsCheck` definido em Z para Java

No Quadro 197 é apresentado o mapeamento do schema `CopyNotExistsCheck` definido em Z para Java. Ao negar um *schema* a implementação também nega a sua relativa implementação.

Z	Java

$CopyNotExistsCheck \mathrel{\widehat=} \neg$
CopyExistsCheck
	`public class Library {`
	`  ...`
	`  private boolean copyNotExistsCheck(Copy c) {`
	`  return !copyExistsCheck(c);`
	`  }`
	`}`

Quadro 197 – Mapeamento do *schema* `CopyNotExistsCheck` definido em Z para Java

No Quadro 198 é apresentado o Mapeamento do schema `CopyExistsError` definido em Z para Java. É adicionado um método a classe `Library` que realiza o *schema* definido, retornando o que está contido na variável com o sufixo ponto de exclamação (!). Mesmo o retorno aqui sendo `void`, o retorno é caracterizado pela mensagem de erro lançada ao invocar o método.

Z	Java

CopyExistsError
rep!: Report
--- 	
rep! = MsgCopyExistsError
	`public class Library {`
	`  ...`
	`  private void copyExistsError() {`
	`  throw new ReportException(Report.CopyAlreadyExists);`
	`  }`
	`}`

Quadro 198 – Mapeamento do *schema* `CopyExistsError` definido em Z para Java

No Quadro 199 é apresentado o mapeamento do schema `AddCopy` definido em Z para Java. É adicionado um método a classe `Library` que implementa a especificação em Z. Ao verificar o conceito de inclusão de *schema*, pode-se ver que a semântica é idêntica a lógica dos conectores entre os *schema*. Os estados possíveis do *schema* são cópia não existente e inclusão de cópia, ou cópia existente e erro de cópia existente. Qualquer outro estado deste *schema* não é válido. A definição em Java, além de seguir a mesma lógica, também não permite outro estado do *schema*.

Z	Java

AddCopy ≙
CopyNotExistsCheck ∧
AddCopySuccess ∨
CopyExistsCheck ∧
CopyExistsError
	`public class Library {`
	`  ...`
	
	`  public void addCopy(Copy c) {`
	`  if (copyNotExistsCheck(c)) {`
	`  addCopySuccess(c);`
	`  } else if (copyExistsCheck(c)) {`
	`  copyExistsError();`
	`  } else {`
	`  throw new IllegalStateException();`
	`  }`
	`  }`
	`}`

Quadro 199 – Mapeamento do *schema* `AddCopy` definido em Z para Java

No Quadro 200 é apresentado o mapeamento do schema `RemoveCopySuccess` definido em Z para Java. Neste *schema* é apresentada mais uma utilização e mapeamento de relações em Z para Java.

Z	Java

RemoveCopySuccess
ΔLibrary
c?: Copy
--- 	
stock′ = {c? . id} ⩤ stock
shelved′ = {c? . id} ⩤ shelved
	`public class Library {`
	`  ...`
	`  public void removeCopy(Copy c) {`
	`  stock = stock.domain_restriction(stock, c.id);`
	`  shelved = shelved.domain_restriction(shelved, c.id);`
	`  }`
	`}`

Quadro 200 – Mapeamento do *schema* `RemoveCopySuccess` definido em Z para Java

No Quadro 201 é apresentado o mapeamento do schema `CopyHasIssue` definido em Z para Java. Aqui é apresentado um grande poder e flexibilidade do Z, a utilização de coleções e navegações que a notação proporciona.

Z	Java

CopyHasIssue
$\Xi Library$
$c?: Copy$
--- 	
$c?.id \in \{ x: \text{ran } issued \bullet x.copy.id \}$
	`public class Library {`
	`  ...`
	`  public boolean copyHasIssue(Copy c) {`
	`  ZSet<Issue> issues = issued.range();`
	`  for (Issue issue : issues) {`
	`  if (issue.copy.id.equals(c.id)) {`
	`  return true;`
	`  }`
	`  }`
	`  return false;`
	`  }`
	`}`

Quadro 201 – Mapeamento do *schema* `CopyHasIssue` definido em Z para Java

No Quadro 202 é apresentado o mapeamento do schema `ReturnIssuedCopies` definido em Z para Java. Este é um *schema* que especifica uma consulta e retorna uma coleção de cópias em empréstimo. A mesma funcionalidade pode ser também facilmente identificada na implementação em Java.

Z	Java

ReturnIssuedCopies
$\Delta Library$
$result!: \mathbb{F}\ Copy$
--- 	
$result! = \{ x: \text{ran } issued \bullet x.copy \}$
	`public ZSet<Copy> returnIssuedCopies() {`
	`  ZSet<Issue> issued = library.issued.range();`
	`  ZSet<Copy> copies = new BasicSet<Copy>();`
	`  for (Issue issue : issued) {`
	`  Copy copy = issue.copy;`
	`  copies.add(copy);`
	`  }`
	`  return copies;`
	`}`

Quadro 202 – Mapeamento do *schema* `ReturnIssuedCopies` definido em Z para Java

3.3.3 Implementação da interface

A implementação da interface ocorreu de forma tradicional ao desenvolvimento orientado a objetos. Na Figura 5 é demonstrado o modelo de classes utilizadas na implementação da interface.

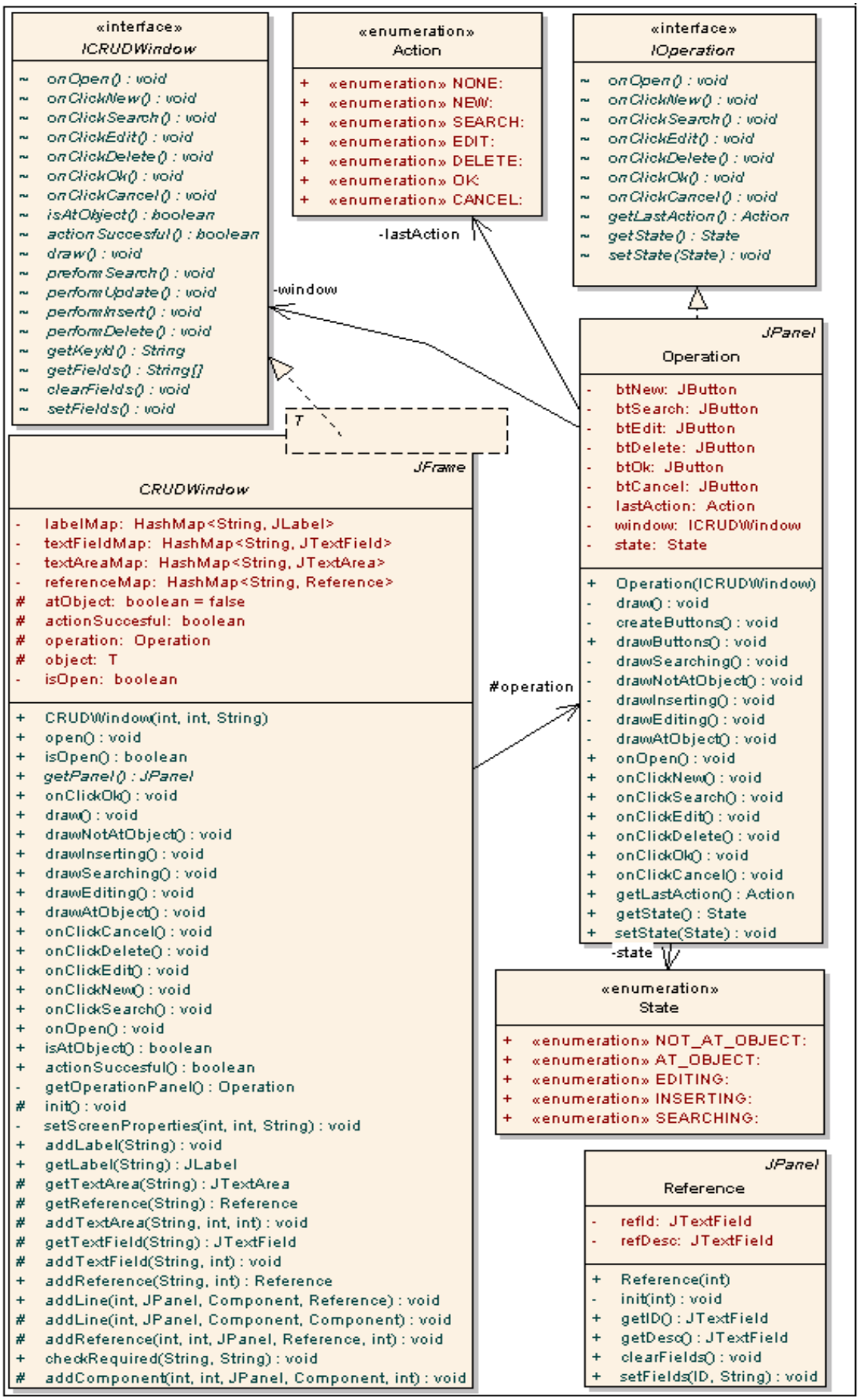


Figura 5 - Diagrama das classes utilizadas na definição da interface

As classes (Figura 5) são as que possibilitam criar uma tela para cada objeto especificado. Grande parte da implementação necessária para as telas do sistema é definida nestas classes bases. Juntando a utilização da estrutura de *generics* do Java é possível construir um código enxuto para as classes que necessitam derivar da implementação base. No Quadro 203 é apresentada a descrição das classes bases para utilização de tela.

Classe	Descrição
`ICRUDWindow`	Interface contendo todas as operações que uma tela deve contemplar ao ser implementada.
`CRUDWindow`	Classe abstrata que implementa a interface `ICRUDWindow`. Funções comuns a todas as telas são implementadas nesta classe.
`IOperation`	Interface contendo todas as operações realizadas. São as operações de interação com o usuário.
`Operation`	Implementação da interface `IOperation` que associada a uma implementação de `ICRUDWindow`. Orquestra as ações do usuário e o comportamento da tela.
`Action`	Enumeração contendo todas as ações do usuário. As ações possíveis são: novo, editar, excluir, procurar, confirmar e cancelar.
`State`	Estado da tela em determinado momento. Os estados da tela podem ser: posicionado no objeto, não posicionado no objeto, editando, iserindo ou procurando.
`Reference`	Classe auxiliar para tratar atributos do tipo referência na tela.

Quadro 203 – Classes utilizadas para construir uma tela

Após apresentada a estrutura básica para a construção de uma tela, é mostrada uma implementação de tela que manipula o objeto especificado `Copy` (Quadro 204). A implementação estende a classe base `CrudWindow` e utilizando da construção de *generics* define o objeto `Copy`. No construtor uma instância da biblioteca é associada e o tamanho da tela é definida. Os métodos `getFields`, `getKeyID`, `getPanel` e `createComponents` têm reponsabilidade de definir o que será apresentado na tela e de que forma serão construídos. Os métodos `clearFields` e `setFields` são eventos que tratam a apresentação das informações na tela.

```
Public class CopyView extends CRUDWindow<Copy> {

  private static final String ID = "ID";
  private static final String LITERACY_WORK = "Literacy Work";
  private static final String[] fields = { LITERACY_WORK };
  private Library library;
  private Reference rfLiteracyWork;

  public CopyView(Library library) {
    this(500, 300, "Copy");
    this.library = library;
  }

  public CopyView(int width, int height, String title) {
    super(width, height, title);
  }

  @Override
  public String[] getFields() {
    return fields;
  }

  @Override
  public String getKeyId() {
    return ID;
  }

  @Override
  public void clearFields() {
    getTextField(ID).setText("");
    rfLiteracyWork.clearFields();
  }

  @Override
  public void setFields() {
    getTextField(ID).setText(String.valueOf(object.id.id));

    LiteracyWork literacyWork = object.literacyWork;
    rfLiteracyWork.setFields(literacyWork.id, literacyWork.name);
  }

  @Override
  public JPanel getPanel() {
    createComponents();
    int gridx = 0;
    JPanel panel = new JPanel();
    panel.setLayout(new GridBagLayout());
    addLine(gridx++, panel, getLabel(ID), getTextField(ID));
    addLine(gridx++, panel, getLabel(LITERACY_WORK), rfLiteracyWork);
    return panel;
  }

  private void createComponents() {
    addLabel(ID);
    addLabel(LITERACY_WORK);

    addTextField(ID, 80);
    rfLiteracyWork = addReference(LITERACY_WORK, 200);
  }
...
}
```

Quadro 204 – Fonte Java tratando aspectos da interface

Os eventos de manipulação do objeto são apresentados no Quadro 205. Os eventos presentes nos métodos `performDelete`, `performSearch` e `performInsert` somente invocam os métodos presentes na biblioteca. O método `performUpdate` associa diretamente a variável manipulada na tela (neste caso `rfLiteracyWork`) ao atributo `literacyWork` do tipo `Copy`.

```
public class CopyView extends CRUDWindow<Copy> {

  private static final String ID = "ID";
  private static final String LITERACY_WORK = "Literacy Work";
  private static final String[] fields = { LITERACY_WORK };
  private Library library;
  private Reference rfLiteracyWork;

  ...

  @Override
  public void performDelete() {
    library.removeCopy(object);
  }

  @Override
  public void preformSearch() {
    int id = Integer.parseInt(getTextField(ID).getText());
    object = library.seachCopy(new ID(id));
  }

  @Override
  public void performUpdate() {
    LiteracyWork literacyWork = getLiteracyWork(rfLiteracyWork);
    object.literacyWork = literacyWork;
  }

  @Override
  public void performInsert() {
    int id = Integer.parseInt(getTextField(ID).getText());
    LiteracyWork literacyWork = getLiteracyWork(rfLiteracyWork);

    Copy obj = new Copy(new ID(id), literacyWork);
    library.addCopy(obj);
    object = obj;
  }

  private LiteracyWork getLiteracyWork(Reference ref) {
    int id = Integer.parseInt(ref.getID().getText());
    return library.getLiteracyWork(new ID(id));
  }
}
```

Quadro 205 – Fonte Java tratando eventos de manipulação dos objetos

3.3.4 Implementação de testes

As especificações realizadas em Z podem ajudar na definição de testes a serem implementados. Em Java existe a biblioteca JUnit que facilita programar os testes. No Quadro 206 é apresentada a implementação de uma classe de testes para a especificação de data. No

Quadro 206 também são apresentados métodos auxiliares para checagem de datas inválidas. Um erro ocorre caso uma data inconsistente seja instanciada com sucesso.

```
public class TestDate {
  ...
  private void checkInvalidDate(int year, int month, int day) {
    try {
      new Date(year, month, day);
      fail();
    } catch (IllegalArgumentException e) {
      assertEquals("Invalid Date", e.getMessage());
    }
  }

  private void checkInvalidInput(int year, int month, int day) {
    try {
      new Date(year, month, day);
      fail();
    } catch (IllegalArgumentException e) {
      //ok
    }
  }
}
```

Quadro 206 – Métodos auxiliares para checagem de datas inválidas

No Quadro 207 é apresentado um teste que verifica o valor interno de uma data válida. O valor interno é utilizado na comparação de datas, o qual é um inteiro que contém a quantidade de dias de uma data deste o ano de 1900.

```
public class TestDate {
  ...
  @Test
  public void testDate() throws Exception {
    Date date = new Date(2001, 9, 11);
    assertEquals(37146, date.value);

    Date date2 = new Date(2001, 9, 12);
    assertEquals(37147, date2.value);
  }
}
```

Quadro 207 – Teste de valor interno de data válida

No Quadro 208 é apresentado um teste onde os limites dos valores de uma data (ano, mês e dia) são verificados. É utilizado um valor mínimo e um valor máximo possível para a data.

```
public class TestDate {
  ...
  @Test
  public void testDateLimits() throws Exception {
    Date date = new Date(1900, 1, 1);
    assertEquals(32, date.value);

    Date date2 = new Date(9999, 12, 31);
    assertEquals(2956538, date2.value);
  }
}
```

Quadro 208 – Teste de limites dos valores de uma data

No Quadro 209 é apresentado um teste onde os limites dos valores de uma data (ano, mês e dia) são extrapolados. Todas as unidades da data têm os valores definidos fora do limite

especificado, tanto para mais como para menos.

```
Public class TestDate {
  ...
  @Test
  public void testDateOutOfLimits() throws Exception {
    checkInvalidInput(1899, 12, 31);
    checkInvalidInput(10000, 1, 1);
    checkInvalidInput(2000, 0, 1);
    checkInvalidInput(2000, 13, 1);
    checkInvalidInput(2000, 1, 0);
    checkInvalidInput(2000, 1, 32);
  }
}
```

Quadro 209 – Teste de data com valores fora do limite

No Quadro 210 é apresentado um teste onde são verificadas as datas válidas em anos bissextos.

```
public class TestDate {
  ...
  @Test
  public void testDateInLeapYear() throws Exception {
    Date date = new Date(2008, 2, 28);
    assertEquals(39614, date.value);

    Date date2 = new Date(2008, 2, 29);
    assertEquals(39615, date2.value);

    Date date3 = new Date(2400, 2, 28);
    assertEquals(183086, date3.value);

    Date date4 = new Date(2400, 2, 29);
    assertEquals(183087, date4.value);
  }
}
```

Quadro 210 – Teste de data válida em anos bissextos

No Quadro 211 é apresentado um teste onde são verificadas datas inválidas em anos bissextos.

```
public class TestDate {
  ...
  @Test
  public void testDateNotInLeapYear() throws Exception {
    Date date = new Date(1900, 2, 28);
    assertEquals(84, date.value);

    checkInvalidDate(1900, 2, 29);

    Date date3 = new Date(2009, 2, 28);
    assertEquals(39869, date3.value);

    checkInvalidDate(2009, 2, 29);
  }
}
```

Quadro 211 – Teste de data inválida em anos bissextos

No Quadro 212 é apresentado um teste verificando várias datas inválidas.

```
public class TestDate {
  ...
  @Test
  public void testInvalidDate() throws Exception {
    checkInvalidDate(2000, 2, 30);

    checkInvalidDate(2000, 2, 31);
    checkInvalidDate(2000, 4, 31);
    checkInvalidDate(2000, 6, 31);
    checkInvalidDate(2000, 9, 31);
    checkInvalidDate(2000, 11, 31);
  }
}
```

Quadro 212 – Teste de com um apanhado de datas inválidas

O resultado da execução deste teste pode ser visualizado na Figura 6.

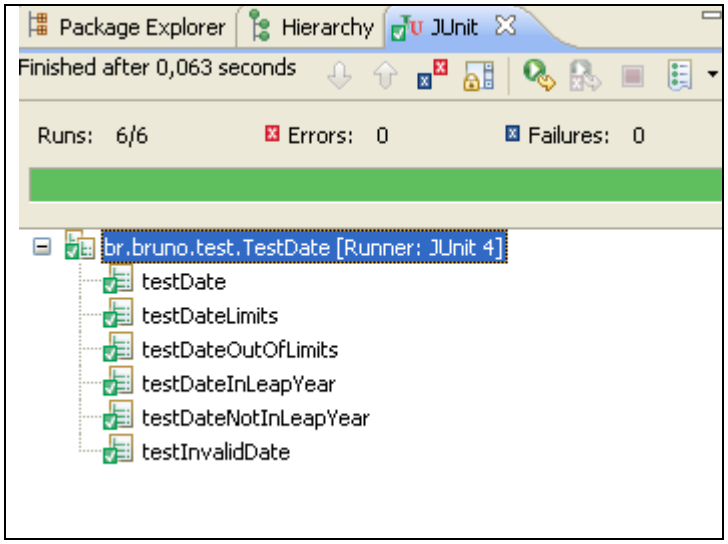


Figura 6 - Resultado da execução do JUnit de datas

3.3.5 Operacionalidade da implementação

Para a melhor compreensão dos conceitos apresentados, desde a especificação até o mapeamento das construções é apresentado o estudo de caso implementado na linguagem de programação Java.

3.3.5.1 Estudo de caso

O estudo de caso trata-se de um sistema de biblioteca. O sistema possibilita realizar o cadastro e manipulação de autores, tipos de leitores, leitores, bibliotecários, obras literárias e cópias das obras literárias. São obras literárias os livros, as revistas, os jornais, os trabalhos de conclusão de curso, as dissertações de mestrado e as teses de doutorado. O sistema também

permite realizar empréstimos, devolução de empréstimo e reserva. Por fim, o sistema possui geração de relatórios que demonstram: cópias em atraso, cópias em empréstimo e estado de uma cópia específica (emprestada, disponível ou reservada). Na Figura 7 é apresentado o menu dos cadastros na biblioteca. Na Figura 8 é apresentado o menu das operações na biblioteca. Na Figura 9 é apresentado o menu dos relatórios na biblioteca. Na Figura 10 é apresentada a tela de cadastro de livro na biblioteca. Na Figura 11 é apresentada a tela de empréstimo na biblioteca. Na Figura 12 é apresentada a tela de consulta de estado de uma cópia e na Figura 13 é apresentado o resultado da consulta do estado da obra. Nem todas as telas do sistema são apresentadas, apenas as necessárias para poder compreender o sistema como um todo. A interface do programa foi escrita na língua inglesa.

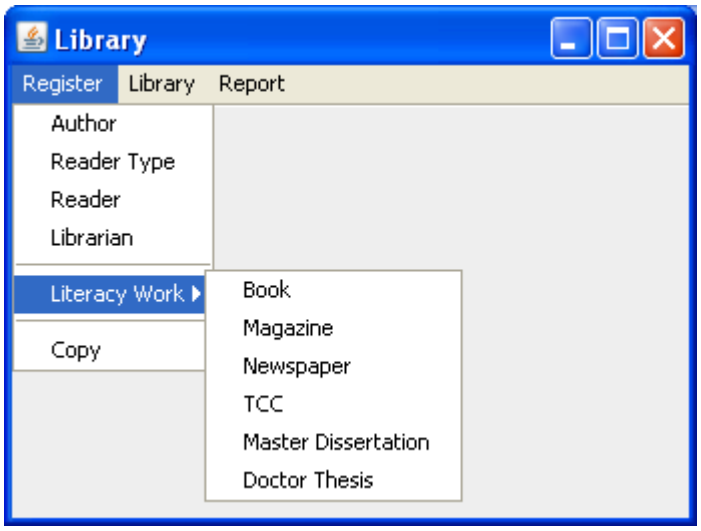


Figura 7 - Tela de cadastro na biblioteca

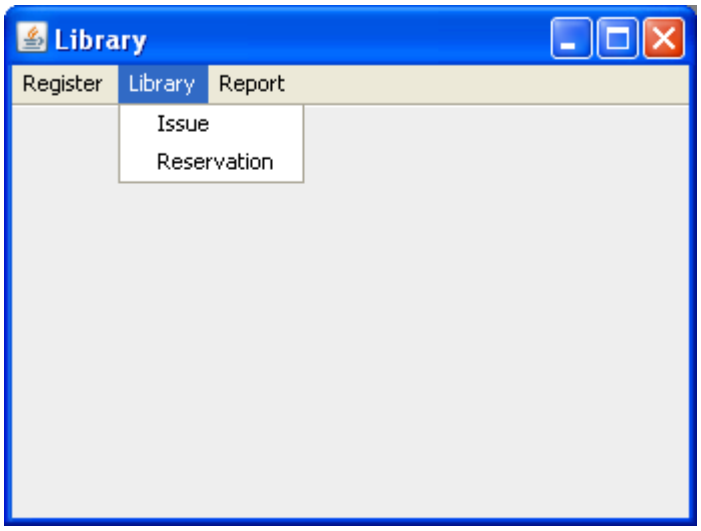


Figura 8 - Tela de operações da biblioteca

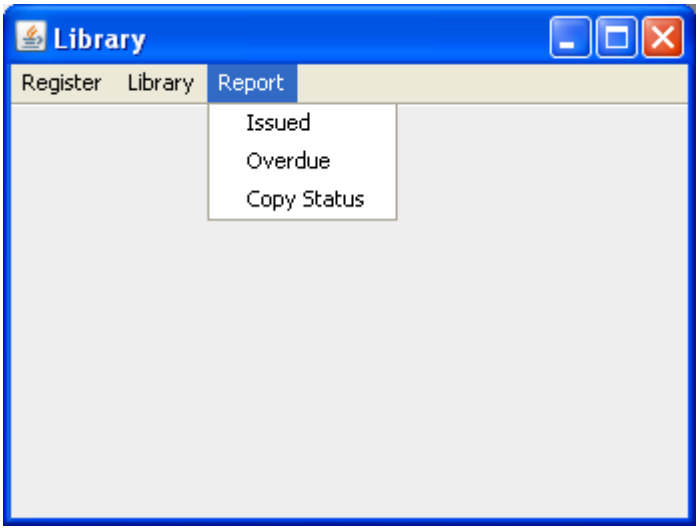


Figura 9 - Tela de relatórios na biblioteca

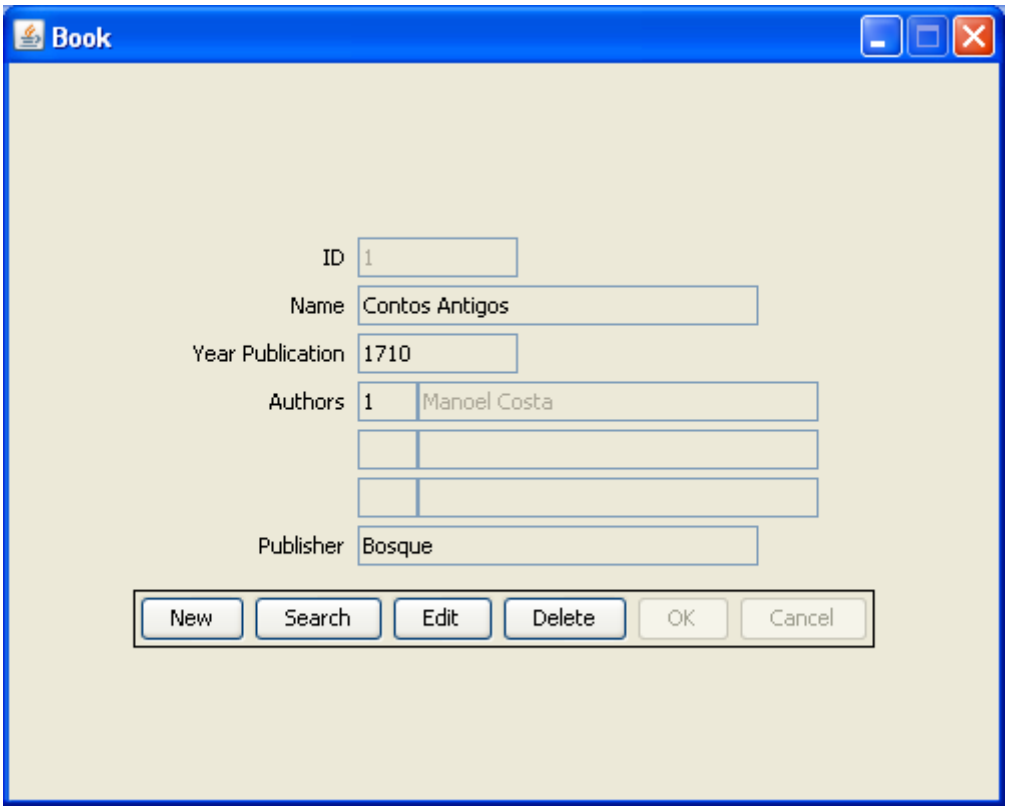


Figura 10 - Tela de cadastro de livro na biblioteca

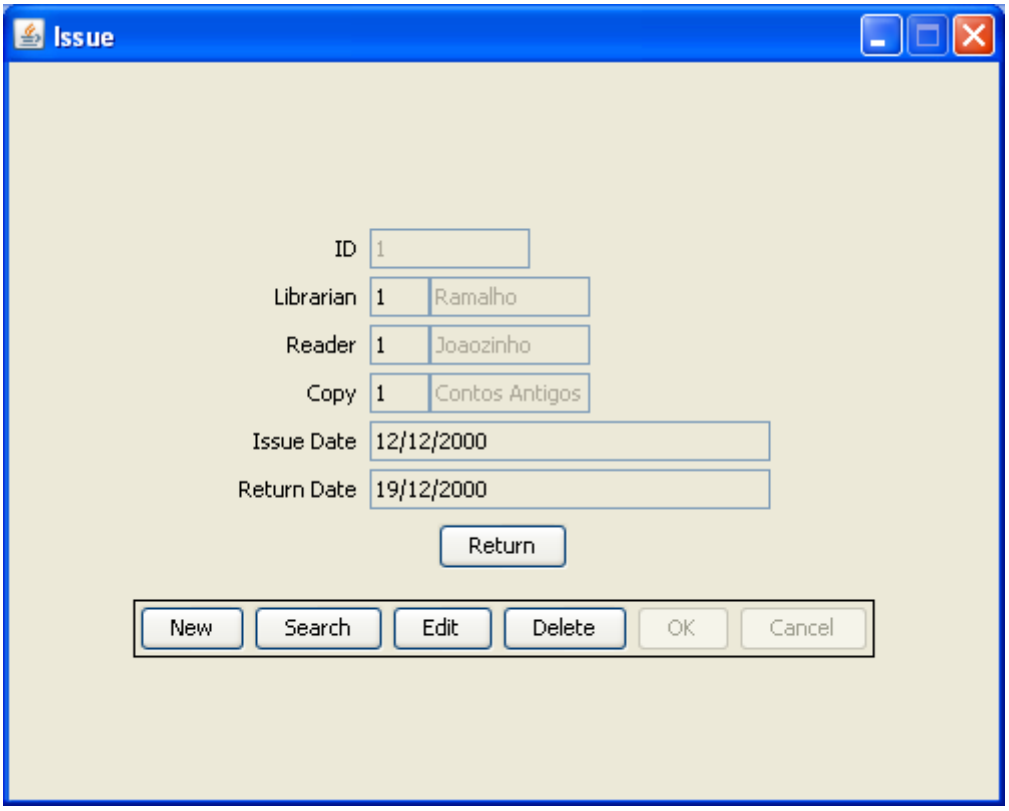


Figura 11 - Tela de empréstimo na biblioteca

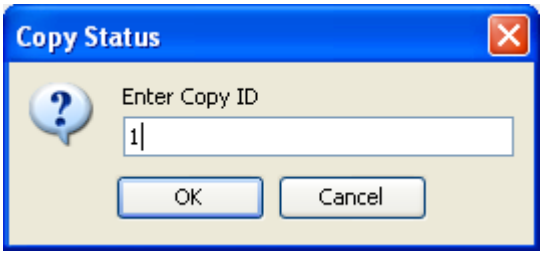


Figura 12 – Soliticação de estado de cópia

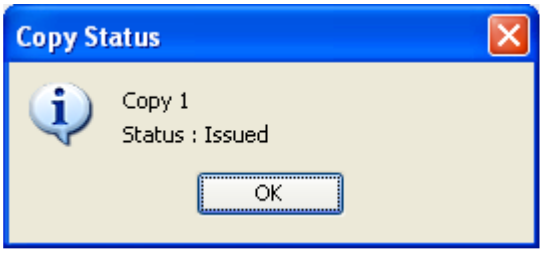


Figura 13 – Estado da cópia solicitada

3.4 RESULTADOS E DISCUSSÃO

O estudo de caso apresentado pode ser considerado simples. Desenvolvedores com grande experiência em Java podem até questionar a utilização de métodos formais para desenvolver simples funções de cadastro e consulta. Porém, nem sempre o problema mostra-

se tão simples de se resolver. O que geralmente ocorre, mesmo sendo um sistema simples, são dúvidas sobre alguma funcionalidade, ou erros após a entrega do software. Normalmente espera-se "pequenas alterações" no software entregue, que são consideradas "normais", o que é minimizado utilizando uma especificação formal.

Para melhor demonstrar os resultados do trabalho, as considerações são apresentadas em tópicos segundo os sete mitos da especificação formal (Hall, 1990), os quais são:

a) mito um: métodos formais garantem que o software é perfeito: a utilização da especificação formal conforme já apresentado não é a panacéia do desenvolvimento do software. Pode-se comprovar a grande ajuda da especificação construída no processo de desenvolvimento. A grande contribuição, que pode ser atribuída aos *schemas*, é a aproximação dos requisitos com o projeto da implementação. Ao escrever um *schema*, requisitos e modelos de desenvolvimento estão no mesmo local, possibilitando aproximar requisitos, especificação e implementação. À linguagem matemática pode-se atribuir a clareza existente na especificação de um *schema*. Nos métodos tradicionais os requisitos, regras de negócio e modelos de projeto estão espalhados em modelos semi-formais (normalmente através de diagramas);
b) mito dois: métodos formais são para realizar provas de programas: este trabalho não explorou fortemente o uso de provas que as especificações formais proveem. Mesmo assim, conforme explicado no mito um, houve grande utilidade dos métodos formais no processo de desenvolvimento. O método tradicional não possui qualquer tipo de prova formal para o desenvolvimento de software;
c) mito três: métodos formais somente são úteis para programas onde a segurança é crítica: conforme apresentado neste trabalho, a utilidade de métodos formais em sistemas que não tem características tão grande de segurança também é de grande valia. Não deixa dúvida que indiferente das características do software, métodos formais veem para somar no processo de desenvolvimento;
d) mito quatro: métodos formais exigem matemáticos altamente treinados: certamente é necessário conhecimento matemático para utilizar métodos formais e a notação Z. Um grande tempo de estudo foi necessário antes de efetivamente utilizar todo o poder da notação Z. Utilizando de uma análise mais crítica, Z combina teoria dos conjuntos e lógica proposicional, ambas amplamente discutidas e utilizadas no ambiente de desenvolvimento de software. Fica claro que não é necessário ser um matemático para aplicar os métodos formais, o que é necessário é que o

desenvolvedor tenha interesse e vontade em utilizar esta metodologia;

e) mito cinco: métodos formais aumentam o custo do desenvolvimento: não há dados neste trabalho que possam comprovar, ou refutar este argumento. Realizando uma especulação pode-se dizer que foi gasto um grande tempo no estudo da especificação, em aprendizado da notação e em mapeamento das construções para linguagem Java. Mesmo com todo o tempo despendido, o tempo de programação na parte de regras de negócio foi bem menor que o despendido para codificar as telas. Claro que não se pode comparar implementações distintas para tirar conclusões seguras, porém é possível compartilhar que os objetivos na parte da codificação aparentam-se mais bem definidos utilizando de métodos formais. Neste trabalho não se tem como mensurar o número de erros gerados, porém se seguir o que é citado a respeito de métodos formais (que a quantidade de erros é menor) o processo é válido;
f) mito seis: métodos formais não são aceitados pelo usuário: não é verdade que os métodos não são aceitos. Os métodos formais não são amplamente difundidos, e como resultado surge preconceito sobre o chamado "método matemático". Somente será possível de verificar o aceite do método a partir do momento que o método estiver difundido nas instituições de ensino, ou no mercado de trabalho;
g) mito sete: métodos formais não são utilizados na maioria dos softwares: realmente é difícil encontrar desenvolvimento em larga escala utilizando métodos formais. Além dos aspectos apresentados no mito seis, outro problema deve-se pelas poucas ferramentas e falta de processos bem definidos. Não foi constatada a existência de uma definição ou corrente que defina como realizar uma implementação utilizando métodos formais. Apesar de Z/EVES ser um dos mais populares *frameworks* para especificação formal, sua utilização é complexa e sua interface é pobre, não intuitiva e de difícil utilização.

Após apresentar resultados do trabalho utilizando-se dos sete mitos, outros pontos merecem alguma atenção. Ao visualizar uma especificação pode aparentar que será necessário despender um grande esforço para transformar todo o arcabouço matemático inserido na especificação e obter algo executável transcrito para uma linguagem orientada a objetos. Este mapeamento que aparenta ser complexo, não se apresenta deste modo. Conforme apresentado no mapeamento da especificação para o código Java, não é algo que seja extremamente complexo, porém exige certo conhecimento nas duas tecnologias. Após isso, a codificação tornou-se muito mais simples que qualquer outro tipo de especificação ou modelo de

desenvolvimento utilizado.

A percepção é que uma especificação mapeada poderia ser gerada automaticamente a partir das construções em Z. A falta de ferramentas é um dos principais problemas. Não foi constatada a existência de ferramentas que consigam trabalhar com a especificação e código de forma conjunta objetivando uma maior produtividade.

A utilização da especificação para construir testes foi de grande valia, acima do que era esperado. A certeza de testar algo que está definido de forma exata dá uma maior segurança. Os testes a serem realizados são mais facilmente identificados em virtude da clareza da especificação. O confrontamento com os requisitos, que são escritos em linguagem informal, também são mais facilmente verificados.

4 CONCLUSÕES

Desenvolver uma especificação formal a partir de uma notação não conhecida, não é uma das tarefas mais triviais. A curva de aprendizado para utilizar os conceitos e construções é relativamente grande e morosa. Mesmo utilizando da ferramenta Z/EVES para construção da especificação, a especificação do sistema é trabalhosa e exige muita dedicação.

O objetivo principal deste trabalho foi alcançado. A utilização de uma especificação formal no desenvolvimento de sistemas foi largamente discutido no trabalho. A implementação torna-se mais compreensiva e provê um maior dinamismo ao implementar um sistema a partir de uma especificação escrita em Z, mostrando que esta foi de grande valia na fase de implementação. A notação Z seria mais bem aproveitada caso existisse um ferramental mais completo que proporcionasse a construção de especificações e *templates* de códigos para a especificação construída.

Os requisitos do sistema de biblioteca foram executados com sucesso, tanto as funções de cadastro e manipulação dos objetos, como os relatórios e funções. Todos foram especificados com a notação Z e implementados em Java.

Na parte de testes, a especificação em Z mostrou-se de grande viabilidade. Desenvolver testes utilizando da especificação formal mostrou-se muito mais eficiente do que técnicas tradicionais. A especificação trouxe clareza e segurança ao implementar os testes.

A utilização de prova de propriedades não foi amplamente utilizada, porém foi demonstrado que é possível sim utilizar das propriedades matemáticas. A prova possibilita garantir que o especificado é ou não consistente.

O roteiro de uso que se propôs inicialmente não foi construído. Para isso, é necessário um maior aprofundamento no assunto para não ser apenas uma compilação de boas práticas no processo de desenvolvimento. Certamente, a construção de uma metodologia para desenvolvimento utilizando especificação formal é mais viável e agrega mais valor.

Resta concluir que especificação formal é sim viável e útil. Com o aprendizado, experiência adquirida e ferramental adequado é muito mais produtivo e simples realizar a implementação de um sistema.

4.1 EXTENSÕES

Algumas melhorias são propostas para este trabalho são:

a) implementar uma ferramenta para realizar a construção de especificações em Z, tendo em vista que as atuais deixam um pouco a desejar. Criar funções na ferramenta para gerar código executável e testes a partir a especificação;
b) implementar a especificação de prova utilizando a especificação do sistema construído;
c) implementar o refinamento da especificação. As literaturas falam em especificação, prova e refinamento;
d) focar a especificação de um sistema para um ambiente específico. Verificar como seria especificar um sistema que utilize uma conexão com o banco de dados, *webservices*, processos distribuídos ou outra tecnologia atualmente difundida;
e) construir uma metodologia para o desenvolvimento de software utilizando especificação formal.

REFERÊNCIAS BIBLIOGRÁFICAS

BARDEN, R.; COOPER, D.; STEPNEY S. **Z in practice.** Hemel Hempstead: Prentice Hall International (UK), 1994.

DAVIES, J.; WOODCOCK J. **Using Z.** [S.l.]: [s.n.], 1996. Disponível em: <http://www.usingz.com/text/zedbook.zip>. Acesso em: 13 maio 2009.

FINDEISS, J. P. **Método formal de especificação VDM e sua aplicação no desenvolvimento de um sistema.** 1999. 62 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

HALL, J. A. Seven myths of formal methods. **IEEE Software**, [S.l.], v. 7, n. 5, p. 11-19, sept. 1990. Disponível em: <http://lml.ls.fi.upm.es/rsd/Papers/seven.myths.pdf.>. Acesso em: 16 set. 2008.

HAMER, U.; PELESKA, J. **The CIDS A330/340 cabin communication system**: a Z application. Englewood Cliffs: Prentice Hall International, 1995. Disponível em: <http://www.informatik.uni-bremen.de/agbs/jp/papers/airbus.ps.gz>. Acesso em: 17 set. 2008

HARRY, A. **Formal methods fact file.** Chichester: John Wiley & Sons, 1996.

INCE, D. C. **An introduction to discrete mathematics, formal system specification, and Z.** 2nd. ed. New York: Oxford University Press, 1992.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO/IEC 13568**: information technology — Z formal specification notation — syntax, type system and semantics. Geneva, 2002. Disponível em: <http://webstore.iec.ch/preview/info_isoiec13568%7Bed1.0%7Den.pdf>. Acesso em: 05 maio 2009.

MARTINI, A. **A notação Z**: uma introdução a notação Z. Porto Alegre: PUCRS, 2008. Disponível em: <http://www.inf.pucrs.br/~alfio/ParadigmasFormais/intro_Z-new.pdf>. Acesso em: 13 maio 2009.

MEISELS, I.; SAALTINK, M. **The Z/EVES reference manual**. Ottawa: ORA Canada, 1995. Disponível em: <http://azalea.icu.ac.kr/courses/ice532/download_files/941_3_Z-refmanual.pdf>. Acesso em: 3 jun. 2009.

MIYAZAWA, A. H. **Geração parcial de código Java a partir de especificações formais Z.** 2008. 144 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo.

MOURA, A. V. **Especificações em Z**: uma introdução. Campinas: Unicamp, 2001.

POTTER, B.; SINCLAR, J.; TILL, D. **An introduction to formal specification and Z**. 2nd. ed. Harlow: Prentice Hall, 1996.

SAALTINK, M. **The Z/EVES 2.0 user's guide**. Ottawa: ORA Canada, 1999. Disponível em: <http://azalea.icu.ac.kr/courses/ice532/download_files/941_2_Z-usersguidegui.pdf>. Acesso em: 3 jun. 2009.

SPIVEY, J. M. **The Z notation**: a reference manual. Oxford: Prentice Hall International (UK), 1992. Disponível em: <http://spivey.oriel.ox.ac.uk/mike/zrm/zrm.pdf>. Acesso em: 20 ago. 2008.

STOCKS, P. A. **Applying formal methods to software testing.** 1993. 180 f. Thesis (Doctor of Phylosophy) – Departament of Computer Science, University of Queensland, Brisbane. Disponível em: <http://www.4shared.com/file/50425758/b97607a7/Project_-_Applying_Formal_Methods_to_Software_Testing.html?s=1>. Acesso em: 14 set. 2008.

ANEXO A – Implementação das coleções e relações

Nesta seção é apresentada a implementação completa das interfaces e classes utilizadas para definir coleções e relações, sendo elas: `ZSet.java` (Quadro 213), `BasicSet.java` (Quadro 214), `ZRelation.java` (Quadro 215) `e BasicPFuncition.java` (Quadro 216).

```
package jzed.mtoolkit;

import java.util.Collection;
import java.util.Set;

public interface ZSet<X> extends Iterable<X>, Collection<X>, Set<X>, Cloneable {
  /* this methods checks whether this set is equal to set S */
  public Boolean equals(ZSet<X> S);

  /* this method checks whether this set contains element e */
  public Boolean contains(X e);

  /* this method checks whether this set contains set S */
  public Boolean contains(ZSet<X> S);

  /* this method checks whether this set contains set S properly */
  public Boolean contains_properly(ZSet<X> S);

  /* this method turns this set into the union of S1 and S2 */
  public ZSet<X> union(ZSet<X> S1, ZSet<X> S2);

  /* this method turns this set into the intersection of S1 and S2 */
  public ZSet<X> intersection(ZSet<X> S1, ZSet<X> S2);

  /* this method turns this set into the difference of S1 and S2 */
  public ZSet<X> difference(ZSet<X> S1, ZSet<X> S2);

  /* this method turns this set into the symmetric difference of S1 and S2 */
  public ZSet<X> symmetric_difference(ZSet<X> S1, ZSet<X> S2);

  /* this method turns this set into the generalized union of the sets in S */
  public ZSet<X> generalized_union(ZSet<ZSet<X>> S);

  /*
   * this method turns this set into the generalized intersection of the sets
   * in S
   */
  public ZSet<X> generalized_intersection(ZSet<ZSet<X>> S);
}
```

Quadro 213 – Definição de `ZSet.java`

```
package jzed.mtoolkit;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;

public class BasicSet<X> extends HashSet<X> implements ZSet<X> {
  public BasicSet() {
    super();
  }

  public BasicSet(Collection<X> S) {
    super();
    addAll(S);
  }

  @Override
  public Boolean contains(X e) {
    return super.contains(e);
  }

  @Override
  public Boolean contains(ZSet<X> S) {
    return super.containsAll(S);
  }

  @Override
  public Boolean contains_properly(ZSet<X> S) {
    return !(!contains(S) || equals(S));
  }

  @Override
  public ZSet<X> difference(ZSet<X> S1, ZSet<X> S2) {
    clear();
    addAll(S1);
    for (X i : S2) {
      this.remove(i);
    }
    return this;
  }

  @Override
  public Boolean equals(ZSet<X> S) {
    return super.equals(S);
  }

  @Override
  public ZSet<X> generalized_intersection(ZSet<ZSet<X>> S) {
    generalized_union(S);
    for (Iterator<X> it = iterator(); it.hasNext();) {
      X x = it.next();
      for (ZSet<X> s : S) {
        if (!s.contains(x))
          it.remove();
      }
    }
    return this;
  }

  @Override
  public ZSet<X> generalized_union(ZSet<ZSet<X>> S) {
    clear();
    for (ZSet<X> s : S) {
      addAll(s);
    }
    return this;
  }
```

```
  @Override
  public ZSet<X> intersection(ZSet<X> S1, ZSet<X> S2) {
    clear();
    addAll(S1);
    for (Iterator<X> it = iterator(); it.hasNext();) {
      if (!S2.contains(it.next()))
        it.remove();
    }
    return this;
  }

  @Override
  public ZSet<X> symmetric_difference(ZSet<X> S1, ZSet<X> S2) {
    clear();
    addAll(S1);
    for (X i : S2) {
      if (contains(i))
        remove(i);
      else
        add(i);
    }
    return this;
  }

  @Override
  public ZSet<X> union(ZSet<X> S1, ZSet<X> S2) {
    clear();
    addAll(S1);
    addAll(S2);
    return this;
  }
}
```

Quadro 214 – Definição de `BasicSet.java`

```
package jzed.mtoolkit;

public interface ZRelation<X, Y> extends Cloneable {
  /*
   * methods that must be implemented because of the set aspect of partial
   * functions
   */
  /* this methods checks whether this relation is equal to relation R */
  public Boolean equals(ZRelation<X, Y> R);

  /* this method checks whether this relation contains the pair x, y */
  public Boolean contains(X x, Y y);

  /* this method checks whether this relation contains relation R */
  public Boolean contains(ZRelation<X, Y> R);

  /* this method checks whether this relation contains relation R properly */
  public Boolean contains_properly(ZRelation<X, Y> R);

  /* this method turns this relation into the union of R1 and R2 */
  public ZRelation<X, Y> union(ZRelation<X, Y> R1, ZRelation<X, Y> R2);

  /* this method turns this relation into the intersection of R1 and R2 */
  public ZRelation<X, Y> intersection(ZRelation<X, Y> R1, ZRelation<X, Y> R2);

  /* this method turns this relation into the difference of R1 and R2 */
  public ZRelation<X, Y> difference(ZRelation<X, Y> R1, ZRelation<X, Y> R2);

  /*
   * this method turns this relation into the symmetric difference of R1 and
   * R2
   */
  public ZRelation<X, Y> symmetric_difference(ZRelation<X, Y> R1, ZRelation<X, Y>
R2);
```

```
  /*
   * this method turns this relation into the generalized union of the
   * relations in R
   */
  public ZRelation<X, Y> generalized_union(ZSet<ZRelation<X, Y>> R);

  /*
   * this method turns this relation into the generalized intersection of the
   * relations in /* methods particular to relations
   */
  public ZRelation<X, Y> generalized_intersection(ZSet<ZRelation<X, Y>> R);

  /* this method returns the domain of this relation */
  public ZSet<X> domain();

  /* this method returns the range of this relation */
  public ZSet<Y> range();

  /*
   * this method turns this relation into the relational composition of
   * relations R1 e R2
   */
  public <T> ZRelation<X, Y> relational_composition(ZRelation<X, T> R1,
ZRelation<T, Y> R2);

  /*
   * this method turns this relation into the functional composition of
   * relations R1 e R2
   */
  public <T> ZRelation<X, Y> functional_composition(ZRelation<X, T> R1,
ZRelation<T, Y> R2);

  /*
   * this method turns this relation into relation R with domain restricted to
   * set S
   */
  public ZRelation<X, Y> domain_restriction(ZRelation<X, Y> R, ZSet<X> S);

  public ZRelation<X, Y> domain_restriction(ZRelation<X, Y> R, X S);

  /*
   * this method turns this relation into relation R with range restricted to
   * set S
   */
  public ZRelation<X, Y> range_restriction(ZRelation<X, Y> R, ZSet<Y> S);

  /*
   * this method turns this relation into relation R with domain subtracted by
   * set S
   */
  public ZRelation<X, Y> domain_subtraction(ZRelation<X, Y> R, ZSet<X> S);

  /*
   * this method turns this relation into relation R with range subtracted by
   * set S
   */
  public ZRelation<X, Y> range_subtraction(ZRelation<X, Y> R, ZSet<Y> S);

  /* this method turns this relation into the inverse of relation R */
  public ZRelation<Y, X> relational_inversion();

  /*
   * this method returns the set equivalent to the relational image of this
   * relation and set S
   */
  public ZSet<Y> relational_image(ZSet<X> S);
```

```
  /* this method turns this relation into relation R1 overridden by R2 */
  public ZRelation<X, Y> overriding(ZRelation<X, Y> R1, ZRelation<X, Y> R2);

  /*
   * this method turns this relation into the transitive reflexive closure of
   * relation R
   */
  public ZRelation<X, X> reflexive_transitive_closure(ZRelation<X, X> R);

  /*
   * this method implements application of this relation to value X and
   * returns the sole range value i such that the pair (x,i) is in the set of
   * pairs that is R. If there is no value or more than one value associated
   * with x, then the application has a value but what it is is not specified (
   * from ISO/IEC 13568)
   */
  public Y apply(X x);

  /* these methods are auxiliary methods */
  /* this method return the set of elements associated with element y */
  public ZSet<X> yapply(Y y);

  /* this method return the set of elements associated with element x */
  public ZSet<Y> xapply(X x);

  public void clear();

  /* this method puts a pair in a relation */
  public Y put(X x, Y y);

  /* this method removes a pair from a relation */
  public Y remove(X x, Y y);

  public int size();

  public Object clone();
}
```

Quadro 215 – Definição de `ZRelation.java`

```
package jzed.mtoolkit;

import java.util.HashMap;
import java.util.Iterator;

public class BasicPFunction<X, Y> extends HashMap<X, Y> implements ZRelation<X, Y>
{
  @Override
  public Y apply(X x) {
    return get(x);
  }

  @Override
  public Boolean contains(X x, Y y) {
    Y aux = apply(x);
    if (aux == null)
      return false;
    if (aux.equals(y))
      return true;
    else
      return false;
  }

  @Override
  public Boolean contains(ZRelation<X, Y> R) {
    for (X x : R.domain()) {
      Y y = R.apply(x);
      if (!contains(x, y))
        return false;
```

```
    }
    return true;
  }

  @Override
  public Boolean contains_properly(ZRelation<X, Y> R) {
    return !(!contains(R) || equals(R));
  }

  @Override
  public ZRelation<X, Y> difference(ZRelation<X, Y> R1, ZRelation<X, Y> R2) {
    clear();
    for (X x : R1.domain()) {
      Y y = R1.apply(x);
      if (!R2.contains(x, y))
        put(x, y);
    }
    return this;
  }

  @Override
  public ZSet<X> domain() {
    return new BasicSet<X>(keySet());
  }

  @Override
  public ZRelation<X, Y> domain_restriction(ZRelation<X, Y> R, X S) {
    ZRelation<X, Y> ret = new BasicPFunction<X, Y>();
    for (X x : R.domain()) {
      if (!x.equals(S)) {
        Y y = R.apply(x);
        ret.put(x, y);
      }
    }
    return ret;
  }

  @Override
  public ZRelation<X, Y> domain_restriction(ZRelation<X, Y> R, ZSet<X> S) {
    clear();
    for (X x : R.domain()) {
      Y y = R.apply(x);
      if (S.contains(x))
        put(x, y);
    }
    return this;
  }

  @Override
  public ZRelation<X, Y> domain_subtraction(ZRelation<X, Y> R, ZSet<X> S) {
    clear();
    for (X x : R.domain()) {
      Y y = R.apply(x);
      if (!S.contains(x))
        put(x, y);
    }
    return this;
  }

  @Override
  public Boolean equals(ZRelation<X, Y> R) {
    if (!domain().equals(R.domain()))
      return false;

    for (X x : R.domain()) {
      if (!apply(x).equals(R.apply(x)))
        return false;
    }
```

```
    return true;
  }

  @Override
  public <T> ZRelation<X, Y> functional_composition(ZRelation<X, T> R1,
ZRelation<T, Y> R2) {
    clear();
    for (X x : R1.domain()) {
      Y y = R2.apply(R1.apply(x));
      put(x, y);
    }
    return this;
  }

  @Override
  public ZRelation<X, Y> generalized_intersection(ZSet<ZRelation<X, Y>> R) {
    clear();
    Iterator<ZRelation<X, Y>> it = R.iterator();
    if (!it.hasNext())
      return this;
    else
      union(it.next());
    while (it.hasNext()) {
      intersection(it.next());
    }
    return this;
  }

  @Override
  public ZRelation<X, Y> generalized_union(ZSet<ZRelation<X, Y>> R) {
    clear();
    for (ZRelation<X, Y> r : R) {
      union(r);
    }
    return this;
  }

  @Override
  public ZRelation<X, Y> intersection(ZRelation<X, Y> R1, ZRelation<X, Y> R2) {
    clear();
    union(R1);
    intersection(R2);
    return this;
  }

  public ZRelation<X, Y> intersection(ZRelation<X, Y> R) {
    for (X x : domain()) {
      Y y = apply(x);
      if (!R.contains(x, y)) {
        remove(x, y);
      }
    }
    return this;
  }

  @Override
  public ZRelation<X, Y> overriding(ZRelation<X, Y> R1, ZRelation<X, Y> R2) {
    clear();
    union(R2);
    ZSet<X> dom2 = R2.domain();
    for (X x : R1.domain()) {
      if (!dom2.contains(x)) {
        Y y = R1.apply(x);
        put(x, y);
      }
    }
    return this;
  }
```

```
  @Override
  public ZSet<Y> range() {
    return new BasicSet<Y>(values());
  }

  @Override
  public ZRelation<X, Y> range_restriction(ZRelation<X, Y> R, ZSet<Y> S) {
    clear();
    for (X x : R.domain()) {
      Y y = R.apply(x);
      if (S.contains(y))
        put(x, y);
    }
    return this;
  }

  @Override
  public ZRelation<X, Y> range_subtraction(ZRelation<X, Y> R, ZSet<Y> S) {
    clear();
    for (X x : R.domain()) {
      Y y = R.apply(x);
      if (!S.contains(y))
        put(x, y);
    }
    return this;
  }

  @Override
  public ZRelation<X, X> reflexive_transitive_closure(ZRelation<X, X> R) {
    throw new IllegalStateException(" This class does not" + " implement method
reflexive_transitive_closure ");
  }

  @Override
  public <T> ZRelation<X, Y> relational_composition(ZRelation<X, T> R1,
ZRelation<T, Y> R2) {
    return functional_composition(R1, R2);
  }

  @Override
  public ZSet<Y> relational_image(ZSet<X> S) {
    ZSet<Y> rimg = new BasicSet<Y>();
    for (X x : S) {
      rimg.add(apply(x));
    }
    return rimg;
  }

  @Override
  public ZRelation<Y, X> relational_inversion() {
    ZRelation<Y, X> f = new BasicPFunction<Y, X>();
    for (X x : domain()) {
      Y y = apply(x);
      if (f.domain().contains(y))
        throw new NotInjectionException();
      else
        f.put(y, x);
    }
    return f;
  }

  @Override
  public Y remove(X x, Y y) {
    Y aux = apply(x);
    if (y.equals(aux))
      return remove(x);
    else
```

```
      return null;
  }

  @Override
  public ZRelation<X, Y> symmetric_difference(ZRelation<X, Y> R1, ZRelation<X, Y>
R2) {
    clear();
    union(R1);
    for (X x : R2.domain()) {
      Y y = R2.apply(x);
      if (contains(x, y))
        remove(x, y);
      else
        put(x, y);
    }
    return this;
  }

  @Override
  public ZRelation<X, Y> union(ZRelation<X, Y> R1, ZRelation<X, Y> R2) {
    clear();
    union(R1);
    union(R2);
    return this;
  }

  public ZRelation<X, Y> union(ZRelation<X, Y> R) {
    if (!(new BasicSet<X>()).intersection(domain(), R.domain()).isEmpty()) {
      // the result is not a function
      throw new NotFunctionException();
    } else {
      for (X x : R.domain()) {
        for (Y y : R.xapply(x)) {
          put(x, y);
        }
      }
    }
    return this;
  }

  @Override
  public ZSet<Y> xapply(X x) {
    ZSet<Y> set = new BasicSet<Y>();
    set.add(apply(x));
    return set;
  }

  @Override
  public ZSet<X> yapply(Y y) {
    ZSet<X> set = new BasicSet<X>();
    if (range().contains(y)) {
      for (X x : domain()) {
        if (y.equals(apply(x)))
          set.add(x);
      }
    }
    return set;
  }
}
```

Quadro 216 – Definição de `BasicPFuncition.java`

APÊNDICE A – Diagramas de pacotes do estudo de caso

As classes construídas no estudo de caso (sistema de biblioteca) são apresentadas na Figura 14 e Figura 15. Estes diagramas foram gerados após a construção do estudo de caso. O diagrama de componentes do sistema de biblioteca é apresentado apenas para ilustrar a organização do sistema.

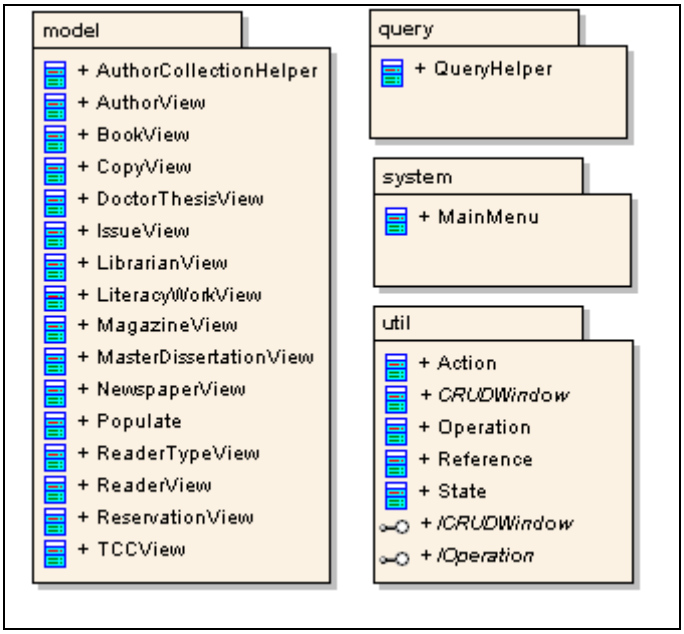


Figura 14 - Diagrama de pacotes das telas do sistema

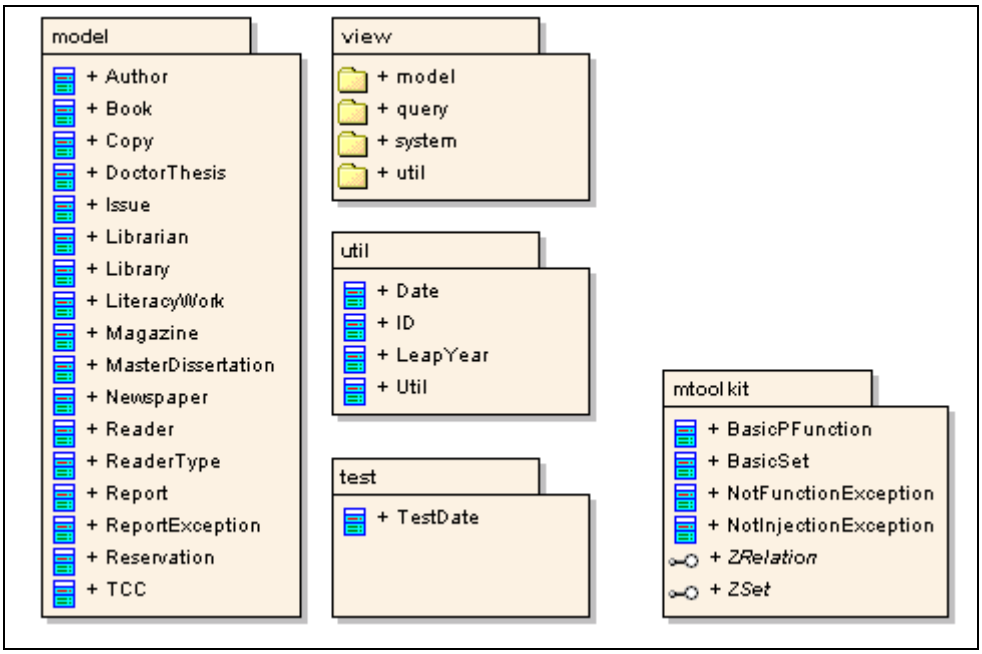


Figura 15 - Diagrama de pacotes do sistema de biblioteca

APÊNDICE B – Prova de propriedades utilizando o Z/EVES

No Quadro 217 é apresentada a prova completa gerada pelo Z/EVES após a especificação do teorema definido no Quadro 171.

```
Rep! ∈ Report
∧ l? . id ∈ dom librarians
∧ (authors ∈ ℙ (ID × ⟨id: ID; name: String⟩)
  ∧ authors' ∈ ℙ (ID × ⟨id: ID; name: String⟩)
  ∧ issued
    ∈ ℙ (ID ×
      ⟨copy: ⟨id: ID;
            literacyWork: ⟨authors: ℙ (ID × ⟨id: ID; name: String⟩);
                          id: ID;
                          name: String;
                          yearPub: ℤ⟩⟩;
       id: ID;
       issueDate: ⟨day: ℤ; month: ℤ; value: ℤ; year: ℤ⟩;
       librarian: ⟨id: ID; name: String⟩;
       reader: ⟨id: ID;
             name: String;
             type: ⟨id: ID; loanDays: ℤ; maxLoans: ℤ; name: String⟩⟩;
       returnDate: ⟨day: ℤ; month: ℤ; value: ℤ; year: ℤ⟩⟩)
  ∧ issued'
    ∈ ℙ (ID ×
      ⟨copy: ⟨id: ID;
            literacyWork: ⟨authors: ℙ (ID × ⟨id: ID; name: String⟩);
                          id: ID;
                          name: String;
                          yearPub: ℤ⟩⟩;
       id: ID;
       issueDate: ⟨day: ℤ; month: ℤ; value: ℤ; year: ℤ⟩;
       librarian: ⟨id: ID; name: String⟩;
       reader: ⟨id: ID;
             name: String;
             type: ⟨id: ID; loanDays: ℤ; maxLoans: ℤ; name: String⟩⟩;
       returnDate: ⟨day: ℤ; month: ℤ; value: ℤ; year: ℤ⟩⟩)
  ∧ l? ∈ ⟨id: ID; name: String⟩
  ∧ librarians ∈ ℙ (ID × ⟨id: ID; name: String⟩)
  ∧ librarians ∈ ID ⤖ Librarian
  ∧ l? ∈ Librarian
  ∧ librarians' = librarians ∪ {(l? . id, l?)}
  ∧ readerTypes ∈ ℙ (ID × ⟨id: ID; loanDays: ℤ; maxLoans: ℤ; name: String⟩)
```

$\wedge$ *readerTypes'* $\in \mathbb{P}$ (*ID* $\times$ ⟨*id: ID; loanDays:* $\mathbb{Z}$*; maxLoans:* $\mathbb{Z}$*; name: String*⟩)
$\wedge$ *readers*
  $\in \mathbb{P}$ (*ID* $\times$
    ⟨*id: ID;*
    name: String;
    type: ⟨*id: ID; loanDays:* $\mathbb{Z}$*; maxLoans:* $\mathbb{Z}$*; name: String*⟩⟩)
$\wedge$ *readers'*
  $\in \mathbb{P}$ (*ID* $\times$
    ⟨*id: ID;*
    name: String;
    type: ⟨*id: ID; loanDays:* $\mathbb{Z}$*; maxLoans:* $\mathbb{Z}$*; name: String*⟩⟩)
$\wedge$ *reservation*
  $\in \mathbb{P}$ (*ID* $\times$
    ⟨*copy:* ⟨*id: ID;*
      literacyWork: ⟨*authors:* $\mathbb{P}$ (*ID* $\times$ ⟨*id: ID; name: String*⟩);
        id: ID;
        name: String;
        yearPub: $\mathbb{Z}$⟩⟩;
    id: ID;
    librarian: ⟨*id: ID; name: String*⟩;
    reader: ⟨*id: ID;*
      name: String;
      type: ⟨*id: ID; loanDays:* $\mathbb{Z}$*; maxLoans:* $\mathbb{Z}$*; name: String*⟩⟩;
    reservationDate: ⟨*day:* $\mathbb{Z}$*; month:* $\mathbb{Z}$*; value:* $\mathbb{Z}$*; year:* $\mathbb{Z}$⟩⟩)
$\wedge$ *reservation'*
  $\in \mathbb{P}$ (*ID* $\times$
    ⟨*copy:* ⟨*id: ID;*
      literacyWork: ⟨*authors:* $\mathbb{P}$ (*ID* $\times$ ⟨*id: ID; name: String*⟩);
        id: ID;
        name: String;
        yearPub: $\mathbb{Z}$⟩⟩;
    id: ID;
    librarian: ⟨*id: ID; name: String*⟩;
    reader: ⟨*id: ID;*
      name: String;
      type: ⟨*id: ID; loanDays:* $\mathbb{Z}$*; maxLoans:* $\mathbb{Z}$*; name: String*⟩⟩;
    reservationDate: ⟨*day:* $\mathbb{Z}$*; month:* $\mathbb{Z}$*; value:* $\mathbb{Z}$*; year:* $\mathbb{Z}$⟩⟩)
$\wedge$ *shelved*
  $\in \mathbb{P}$ (*ID* $\times$
    ⟨*id: ID;*
    literacyWork: ⟨*authors:* $\mathbb{P}$ (*ID* $\times$ ⟨*id: ID; name: String*⟩);
      id: ID;
      name: String;
      yearPub: $\mathbb{Z}$⟩⟩)
$\wedge$ *shelved'*

$$
\begin{aligned}
&\in \mathbb{P}\,(ID \times \\
&\quad \langle id\!:\, ID; \\
&\quad literacyWork\!:\, \langle authors\!:\, \mathbb{P}\,(ID \times \langle id\!:\, ID;\ name\!:\, String\rangle); \\
&\qquad\qquad id\!:\, ID; \\
&\qquad\qquad name\!:\, String; \\
&\qquad\qquad yearPub\!:\, \mathbb{Z}\rangle\rangle) \\
\wedge\ &stock \\
&\in \mathbb{P}\,(ID \times \\
&\quad \langle id\!:\, ID; \\
&\quad literacyWork\!:\, \langle authors\!:\, \mathbb{P}\,(ID \times \langle id\!:\, ID;\ name\!:\, String\rangle); \\
&\qquad\qquad id\!:\, ID; \\
&\qquad\qquad name\!:\, String; \\
&\qquad\qquad yearPub\!:\, \mathbb{Z}\rangle\rangle) \\
\wedge\ &stock' \\
&\in \mathbb{P}\,(ID \times \\
&\quad \langle id\!:\, ID; \\
&\quad literacyWork\!:\, \langle authors\!:\, \mathbb{P}\,(ID \times \langle id\!:\, ID;\ name\!:\, String\rangle); \\
&\qquad\qquad id\!:\, ID; \\
&\qquad\qquad name\!:\, String; \\
&\qquad\qquad yearPub\!:\, \mathbb{Z}\rangle\rangle) \\
\wedge\ &titles \\
&\in \mathbb{P}\,(ID \times \\
&\quad \langle authors\!:\, \mathbb{P}\,(ID \times \langle id\!:\, ID;\ name\!:\, String\rangle); \\
&\quad id\!:\, ID; \\
&\quad name\!:\, String; \\
&\quad yearPub\!:\, \mathbb{Z}\rangle) \\
\wedge\ &titles' \\
&\in \mathbb{P}\,(ID \times \\
&\quad \langle authors\!:\, \mathbb{P}\,(ID \times \langle id\!:\, ID;\ name\!:\, String\rangle); \\
&\quad id\!:\, ID; \\
&\quad name\!:\, String; \\
&\quad yearPub\!:\, \mathbb{Z}\rangle) \\
\wedge\ &readerTypes \in ID \rightarrowtail\!\!\!\!\!\rightarrow ReaderType \\
\wedge\ &authors \in ID \rightarrowtail\!\!\!\!\!\rightarrow Author \\
\wedge\ &stock \in ID \rightarrowtail\!\!\!\!\!\rightarrow Copy \\
\wedge\ &issued \in ID \rightarrowtail\!\!\!\!\!\rightarrow Issue \\
\wedge\ &shelved \in ID \rightarrowtail\!\!\!\!\!\rightarrow Copy \\
\wedge\ &readers \in ID \rightarrowtail\!\!\!\!\!\rightarrow Reader \\
\wedge\ &titles \in ID \rightarrowtail\!\!\!\!\!\rightarrow LiteracyWork \\
\wedge\ &reservation \in ID \rightarrowtail\!\!\!\!\!\rightarrow Reservation \\
\wedge\ &readerTypes' \in ID \rightarrowtail\!\!\!\!\!\rightarrow ReaderType \\
\wedge\ &authors' \in ID \rightarrowtail\!\!\!\!\!\rightarrow Author \\
\wedge\ &stock' \in ID \rightarrowtail\!\!\!\!\!\rightarrow Copy \\
\wedge\ &issued' \in ID \rightarrowtail\!\!\!\!\!\rightarrow Issue
\end{aligned}
$$

$$
\begin{array}{l}
\quad \land\ shelved' \in ID \rightarrowtail\!\!\!\!\!\!\rightarrow Copy \\
\quad \land\ readers' \in ID \rightarrowtail\!\!\!\!\!\!\rightarrow Reader \\
\quad \land\ titles' \in ID \rightarrowtail\!\!\!\!\!\!\rightarrow LiteracyWork \\
\quad \land\ reservation' \in ID \rightarrowtail\!\!\!\!\!\!\rightarrow Reservation \\
\quad \land\ librarians \cup \{(l?\ .\ id, l?)\} \in ID \rightarrowtail\!\!\!\!\!\!\rightarrow Librarian \\
\quad \land\ (readerTypes = readerTypes' \\
\qquad \land\ authors = authors' \\
\qquad \land\ stock = stock' \\
\qquad \land\ issued = issued' \\
\qquad \land\ shelved = shelved' \\
\qquad \land\ readers = readers' \\
\qquad \land\ titles = titles' \\
\qquad \land\ librarians = librarians \cup \{(l?\ .\ id, l?)\} \\
\qquad \Rightarrow \neg\ reservation = reservation') \\
\quad \lor\ readerTypes' \in ID \rightarrowtail\!\!\!\!\!\!\rightarrow ReaderType \\
\qquad \land\ readerTypes = readerTypes' \\
\qquad \land\ authors' \in ID \rightarrowtail\!\!\!\!\!\!\rightarrow Author \\
\qquad \land\ authors = authors' \\
\qquad \land\ librarians' \in ID \rightarrowtail\!\!\!\!\!\!\rightarrow Librarian \\
\qquad \land\ librarians = librarians' \\
\qquad \land\ stock' \in ID \rightarrowtail\!\!\!\!\!\!\rightarrow Copy \\
\qquad \land\ stock = stock' \\
\qquad \land\ issued' \in ID \rightarrowtail\!\!\!\!\!\!\rightarrow Issue \\
\qquad \land\ issued = issued' \\
\qquad \land\ shelved' \in ID \rightarrowtail\!\!\!\!\!\!\rightarrow Copy \\
\qquad \land\ shelved = shelved' \\
\qquad \land\ readers' \in ID \rightarrowtail\!\!\!\!\!\!\rightarrow Reader \\
\qquad \land\ readers = readers' \\
\qquad \land\ titles' \in ID \rightarrowtail\!\!\!\!\!\!\rightarrow LiteracyWork \\
\qquad \land\ titles = titles' \\
\qquad \land\ reservation' \in ID \rightarrowtail\!\!\!\!\!\!\rightarrow Reservation \\
\qquad \land\ reservation = reservation' \\
\qquad \land\ l? \in Librarian \\
\qquad \land\ rep! = MsgLibrarianExistsError) \\
\Rightarrow rep! = MsgLibrarianExistsError
\end{array}
$$

Quadro 217 – Prova gerada pelo Z/EVES