UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE TECNOLOGIA E GEOCIÊNCIAS

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

JUCIMAR MAIA DA SILVA JR

$_{oo}$Erlang

Uma Extensão de Erlang Orientada a Objetos

Recife, agosto de 2013.

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE TECNOLOGIA E GEOCIÊNCIAS

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

$_{oo}$*Erlang*

Uma Extensão de Erlang Orientada a Objetos

por

JUCIMAR MAIA DA SILVA JR

Tese submetida ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Pernambuco como parte dos requisitos para a obtenção do grau de Doutor em Engenharia Elétrica.

Prof. Dr. Rafael Dueire Lins (orientador)

Prof. Dr. Francisco Heron de Carvalho Jr. (co-orientador)

Recife, julho de 2013.

Catalogação na fonte
Bibliotecária: Rosineide Mesquita Gonçalves Luz / CRB4-1361 (BCTG)

S586e Silva Jr., Jucimar Maia da.
$_{oo}$*Erlang* uma extensão de Erlang Orientada a Objetos / Jucimar Maia da Silva Jr. – Recife: O Autor, 2013.
xviii, 108f. il., figs., gráfs., tabs.

Orientador: Prof. Dr. Rafael Dueire Lins.
Co-orientador: Francisco Heron de Carvalho Jr.
Tese (Doutorado) – Universidade Federal de Pernambuco. CTG. Programa de Pós-Graduação em Engenharia Elétrica, 2013.
Inclui Referências.

1. Engenharia Elétrica. 2. Erlang.. *3*. Linguagens de Programação. 4. Programação Orientada a Objetos. I. Lins, Rafael Dueire (Orientador). II. Carvalho Jr. Francisco Heron de (Co-orientador). III. Título.

621.3 CDD (22.ed) UFPE/BCTG-2014 / 030

Universidade Federal de Pernambuco
Pós-Graduação em Engenharia Elétrica

PARECER DA COMISSÃO EXAMINADORA DE DEFESA DE TESE DE DOUTORADO

JUCIMAR MAIA DA SILVA JUNIOR

TÍTULO

"*ooERLANG* – UMA EXTENSÃO DE ERLANG ORIENTADA A OBJETOS"

A comissão examinadora composta pelos professores: RAFAEL DUEIRE LINS, CIN/UFPE; VALDEMAR CARDOSO DA ROCHA JÚNIOR, DES/UFPE; JOÃO ALEXANDRE BAPTISTA VIEIRA SARAIVA, DI/UM; PAULO HENRIQUE MONTEIRO BORBA, CIN/UFPE; FERNANDO JOSÉ CASTOR DE LIMA FILHO, CIN/UFPE e FRANCISCO HERON DE CARVALHO JÚNIOR, DC/UFC, sob a presidência do primeiro, consideram o candidato **JUCIMAR MAIA DA SILVA JUNIOR** **APROVADO**.

Recife, 31 de julho de 2013.

CECÍLIO JOSÉ LINS PIMENTEL
Coordenador do PPGEE

RAFAEL DUEIRE LNS
Orientador e Membro Titular Interno

JOÃO ALEXANDRE BAPTISTA VIEIRA SARAIVA
Membro Titular Externo

FRANCISCO HERON DE CARVALHO JUNIOR
Co-Orientador e Membro Titular Externo

PAULO HENRIQUE MONTEIRO BORBA
Membro Titular Externo

VALDEMAR CARDOSO DA ROCHA JÚNIOR
Membro Titular Interno

FERNANDO JOSÉ CASTOR DE LIMA FILHO
Membro Titular Externo

Dedico este trabalho a meus pais, Jucimar Maia da Silva e Selma Maria de Souza e Silva.

Aos meus filhos Kaio César e Anna Karollina

A Cristina

AGRADECIMENTOS

Ao meu orientador Prof. Rafael Dueire Lins, pela paciência, perseverança e tudo mais.

Ao meu co-orientador Heron carvalho, pela orientação e direcionamento.

Aos professores da UFPE e da UEA.

Aos meus colegas do DINTER, em especial Raimundo Oliveira, Ricardo Barboza, Rodrigo Choji , Ernande Melo e Neide Alves.

Aos meus alunos e orientandos.

A coordenação do DINTER em Manaus, Prof. Francis Wagner e Prof. Antenor.

Aos meus pais. Sem vocês eu não teria conseguido.

A Cristina, pelo amor e paciência.

A todos que me ajudarem de maneira direta ou indireta.

A Deus

O seu tempo é limitado, então não o gaste vivendo a vida de um outro alguém. Não fique preso pelos dogmas, que é viver com os resultados da vida de outras pessoas. Não deixe que o barulho da opinião dos outros cale a sua própria voz interior. E o mais importante: tenha coragem de seguir o seu próprio coração e a sua intuição. Eles de alguma maneira já sabem o que você realmente quer se tornar. Todo o resto é secundário.

- Steve Jobs
Discurso para os formandos da Universidade de Stanford
Junho de 2005

Resumo da Tese apresentada à UFPE como parte dos requisitos necessários
para a obtenção do grau de Doutor em Engenharia Elétrica.

$_{oo}$*Erlang*
Uma Extensão de Erlang Orientada a Objetos

JUCIMAR MAIA DA SILVA JR

Agosto/2013

Orientador: Prof. Dr. Rafael Dueire Lins.
Co-orientador: Prof. Dr. Francisco Heron de Carvalho Jr.
Área de Concentração: Telecomunicações.
Palavras-chave: Erlang, linguagens de programação, programação orientada a objetos.
Número de Páginas: 125.

RESUMO:

Jogos via Internet, redes sociais e as novas aplicações web demandam acesso simultâneo e interativo de milhares (às vezes milhões) de pessoas. Esses sistemas são quase sempre desenvolvidos usando linguagens de *script* como PHP ou usando *frameworks* baseados em linguagens como Java, Ruby ou Pyhton. À proporção que o acesso a esses sistemas cresce, os fornecedores de tais serviços necessitam atender a novas demandas por meio da substituição de hardware por modelos mais potentes, aumentando seus custos operacionais. Quando o nível de acesso cresce drasticamente, o projetista se vê forçado a reprojetar toda a arquitetura do sistema migrando para soluções complexas usando Java Enterprise Edition (JEE) ou Node.js. Essas soluções também demandam mais e mais servidores. O problema possui uma raiz mais profunda: as linguagens de programação usadas para o desenvolvimento de sistemas não foram projetadas para suportar concorrência massiva. Linguagens com suporte a concorrência baseadas no modelo de memória compartilhada não possuem a escalabilidade necessária para atender a demanda. Para resolver os problemas ocasionados pela concorrência massiva, os desenvolvedores estão optando por usar linguagens funcionais como Scala e Erlang na arquitetura do sistema ao contrário de linguagens orientadas a objetos como Java. Mas Erlang não possui uma sintaxe própria para programação orientada a objetos. Este trabalho mostra o desenvolvimento de uma extensão

orientada a objetos para a linguagem Erlang, chamada $_{oo}$*Erlang*, que possui uma melhor expressividade para resolução de problemas "do mundo real" e que não degrade o bom desempenho da linguagem em aplicações que demandam alto tráfego de dados e fina granularidade computacional, tal qual em programas Web 2.0. Assim sendo, o nicho da extensão aqui apresentada é o mesmo de Erlang: desenvolver sistemas *backend* para grandes aplicações onde a concorrência massiva e tolerância a falhas são requeridas.

Abstract of the Thesis presented to the UFPE as part of the necessary requirements for the title of Doctor in Electrical Engineering.

$_{oo}$Erlang

An Object-oriented Extension of Erlang

JUCIMAR MAIA DA SILVA JR

August/2013

Supervisor: Prof. Dr. Rafael Dueire Lins.
Co-supervisor: Prof. Dr. Francisco Heron de Carvalho Jr.
Concentration Area: Telecommunications.
Keywords: Erlang, programming languages, object-oriented programming.
Number of Pages: 125.

ABSTRACT:

Games via the Internet, social networks and new web applications claim for the simultaneous and interactive access of thousands (sometimes millions) of people. Such systems are almost always developed using script languages such Java, Ruby or Pyhton. As the number of accesses to such systems grows, their developers need to meet the new demands, in general by replacing the existing hardware by a more powerful platform, increasing the operating costs.

If the access level grows suddenly, the system engineer is forced to redesign the whole platform using either the Java Enterprise Edition (JEE) or Node.js. Such solution also claims for more and more servers. This problem has deeper roots: the programming languages used for the development of those systems were not designed to support massive concurrency. Languages with support to concurrency based on the shared memory model have not the scalability needed to meet the growth in demand. To solve such problems functional programming languages such as Scala and Erlang are being adopted, instead of object-oriented languages such as Java. Erlang has no adequate object-oriented syntax, and this narrows its applicability.

This thesis presents and object-oriented extension to Erlang, called *$_{oo}$Erlang*, which is more expressive and more suitable for the solution of “real-world” problems. *$_{oo}$Erlang* does not degrade the performance of Erlang in applications that need high data communication traffic and has fine computational granularity, as in current Web 2.0 applications. Thus the niche of the extension

presented here is the same as in Erlang: the development of back-end systems for large applications in which massive concurrency and fault-tolerance are required.

SUMÁRIO

AGRADECIMENTOS ..... **V**

1. INTRODUÇÃO **1**

1.1. LINGUAGENS FUNCIONAIS PARA O DESENVOLVIMENTO DE APLICAÇÕES WEB 3
1.2. MODELOS DE SUPORTE À CONCORRÊNCIA 4
1.2.1. Concorrência baseada em memória compartilhada e threads 4
1.2.2. Concorrência baseada em troca de mensagens 4
1.2.3. Comparação de desempenho do suporte a concorrência 5
1.3. A LINGUAGEM ERLANG 6
1.3.1. Concorrência em Erlang 7
1.3.2. Erlang e a orientação a objetos 8
1.4. MOTIVAÇÃO 10
1.5. OBJETIVO 12
1.5.1. Objetivos específicos 12
1.5.2. Questões de pesquisa 12
1.6. CONTRIBUIÇÕES 13
1.7. ESTRUTURA DESTA TESE 13

2. SINTAXE, SEMÂNTICA E IMPLEMENTAÇÃO **14**

2.1. EXEMPLOS 14
2.1.1. Instanciação de objetos 14
2.1.2. Herança 16
2.2. COMPILAÇÃO 18
2.3. SINTAXE E SEMÂNTICA 19
2.3.1. EBNF 19
2.3.2. Classes e Objetos em $_{oo}$Erlang 23
2.3.3. Herança em $_{oo}$Erlang 24
2.3.4. Vinculação dinâmica em $_{oo}$Erlang 24
2.3.5. Métodos em $_{oo}$Erlang 25
2.3.6. Atributos em $_{oo}$Erlang 27
2.3.7. Tratamento de erros em $_{oo}$Erlang 27
2.3.8. Palavras-chave e símbolos de $_{oo}$Erlang 28
2.4. CONCLUSÃO 30

***3.* EXPRESSIVIDADE** **31**

3.1. EXPRESSIVIDADE 31
3.2. EXEMPLO 1 – SINGLETON 32
3.3. EXEMPLO 2 – BRIDGE 35
3.4. EXEMPLO 3 – STRATEGY 37

3.5. EXEMPLO 5 - PINGPING 39
3.6. EXEMPLO 6 - PINGPONG 49
3.7. EXEMPLO 7 – SENDREC 51
3.8. CONCLUSÕES 54

4. PADRÕES DE PROJETO 55

4.1. CLASSIFICAÇÃO DOS PADRÕES DE PROJETO 56
4.1.1. Padrões de criação 56
4.1.2. Padrões estruturais 56
4.1.3. Padrões comportamentais 57
4.2. OBSERVER 58
4.3. DECORATOR 62
4.4. FACTORY METHOD 64
4.5. ABSTRACT FACTORY 68
4.6. COMMAND 71
4.7. ADAPTER 74
4.8. FAÇADE 76
4.9. TEMPLATE METHOD 78
4.10. CONCLUSÕES 81

5. ANÁLISE DE DESEMPENHO DE $_{OO}$*ERLANG* E OUTRAS LINGUAGENS 82

5.1. INTEL MPI BENCHMARK (IMB) 82
5.2. AMBIENTE DOS EXPERIMENTOS 83
5.3. TESTE PINGPING 85
5.3.1. PingPing com mensagens de 5kB 85
5.3.2. PingPing com mensagens de 10kB 86
5.3.3. PingPing com mensagens de 50kB 87
5.3.4. PingPing com mensagens de 100kB 88
5.4. TESTE PINGPONG 89
5.4.1. PingPong com mensagens de 5kB 89
5.4.2. PingPong com mensagens de 10kB 90
5.4.3. PingPong com mensagens de 50kB 91
5.4.4. PingPong com mensagens de 100kB 92
5.5. TESTE SENDREC– THREADRING 93
5.5.1. Testes com 1000 processos 93
5.5.2. Testes com 10 mil processos 95
5.5.3. Testes com 50 mil processos 97
5.6. CONCLUSÕES 99

6. CONCLUSÕES E TRABALHOS FUTUROS 100

6.1. TRABALHOS FUTUROS 102

7. PUBLICAÇÕES 103

8. REFERÊNCIAS.. 104

LISTA DE FIGURAS

Figura 2.1 : Instanciação de objetos 15
Figura 2.2 : Herança 17
Figure 2.3 : Fluxo interno da compilação no $_{oo}$*Erlang* 19
Figura 2.4 : Diagrama de sintaxe do *form* 20
Figura 2.5 : Diagrama de sintaxe do oo_attributes 21
Figura 2.6 : Diagrama de sintaxe do *attributes_1* 21
Figura 2.7 : Diagrama de sintaxe do *oo_attribute* 21
Figura 2.8 : Diagrama de sintaxe do *oo_method* 22
Figura 2.9 : Diagrama de sintaxe do *function* 22
Figura 2.10 : Diagrama de sintaxe do *function_clauses* 22
Figura 2.11 : Diagrama de sintaxe do *function_clause* 22
Figura 2.12 : Diagrama de sintaxe do *expr_800* 23
Figura 2.13 : Objetos em $_{oo}$*Erlang* 23
Figure 2.14 : Vinculação dinâmica 25
Figure 2.15 : Estrutura de um arquivo .cerl 28
Figura 3.1: Exemplo do *Design Pattern Singleton* 32
Figura 3.2: Exemplo do *Design Pattern Bridge* 35
Figura 3.3: Exemplo do *Design Strategy* 37
Figura 3.4 : PingPing *Benchmark* 39
Figura 3.5 : PingPing 40
Figura 3.6 : PingPong *Benchmark* 49
Figura 3.7**:** Pingpong 50
Figura 3.8: SendRec *Benchmark* 51
Figura 3.9 : Classes doSendRec Threadring 52
Figura 4.1 : Exemplo do padrão Observer 59
Figura 4.2 : Exemplo do padrão Decorator 62
Figura 4.3 : Padrão *Factory Method* 64
Figura 4.4 : Exemplo do *Design Pattern Abstract Factory* 68
Figura 4.5 : Exemplo do *Design Pattern Command* 71
Figura 4.6**:** Exemplo do *Design Pattern Adapter* 74
Figura 4.7 : Exemplo do *Design Pattern Façade* 76
Figura 4.8 : Exemplo do *Design Pattern Template Method* 79

LISTA DE TABELAS

Tabela 2.1 : Principais módulos do ooErlang 18
Tabela 4.1 : Padrões de criação 56
Tabela 4.2 : Padrões estruturais 57
Tabela 4.3 : Padrões comportamentais 57
Tabela 5.1 **:** Classificação dos testes 83
Tabela 5.2 **:** Ambiente de teste 84
Tabela 5.3 : Configurações das linguagens 84

LISTA DE GRÁFICOS

Gráfico 5.1**:** PingPing com mensagens de 5kB 86
Gráfico 5.2: PingPing com mensagens de 10kB 87
Gráfico 5.3**:** PingPing com mensagens de 50kB 88
Gráfico 5.4 : PingPing com mensagens de 100kB 89
Gráfico 5.5 :PingPong com mensagens de 5kB 90
Gráfico 5.6**:** PingPong com mensagens de 10 kB 91
Gráfico 5.7**:** PingPong com mensagens de 50kB 92
Gráfico 5.8 : PingPong com mensagens de 100kB 93
Gráfico 5.9: SendRec de 1000 processos 94
Gráfico 5.10 : SendRec com 10.000 processos 96
Gráfico 5.11 : SendRec com 50.000 processos 98

LISTA DE PROGRAMAS

Programa 1.1: “Hello World” em Erlang 6
Programa 1.2: Quicksort em Erlang 6
Programa 1.3: Quicksort concorrente em Erlang 7
Programa 1.4: Quicksort concorrente em Erlang 8
Programa 1.5: Classe Bike em Java 9
Programa 1.6: Classe Bike em Erlang 10
Programa 1.7: Classe Animal em ECT 11
Programa 1.8: Herança em ECT 11
Programa 1.9: Instanciação de objetos em ECT 11
Programa 2.1: Declarações iniciais do fonte pessoa.cerl 15
Programa 2.2 : Classe gerenciadorDePessoas.cerl 16
Programa 2.3: Código-fonte da classe empregado.cerl 17
Programa 2.4: Declarações iniciais da classe horista.cerl 17
Programa 2.5: EBNF do $_{oo}$*Erlang* 20
Programa 2.6 : Compilação do constructor 24
Programa 2.7 : Vinculação dinâmica 25
Programa 2.8: Exemplo de execução de programa 26
Programa 2.9 : Compilação dos atributos 27
Programa 3.1: Classe ChocolateBoiler em Java 33
Programa 3.2: Classe ChocolateBoiler em Erlang 34
Programa 3.3 : Classe ChocolateBoiler 34
Programa 3.4 : Classe Bike em Java 35
Programa 3.5 : Classe Bike em Erlang 36
Programa 3.6 : Classe Bike em Erlang 37
Programa 3.7 : Classe MallardDuck em Java 38
Programa 3.8 : Classe MallardDuck em Erlang 38
Programa 3.9 : Classe MallardDuck em ooErlang 39
Programa 3.10 : PingPing em Java 43
Programa 3.11 : PingPing em Erlang 45
Programa 3.12 : PingPing em $_{oo}$*Erlang* 47
Programa 3.13: Método run(DataSize,R) da classe pingping.cerl 47
Programa 3.14: Método run(DataSize,R,OutFile) da classe pingping 48
Programa 3.15: Método finalize(Pid) da classe pingping.cerl 48
Programa 3.16: Classe procping 49

Programa 3.17: Método run(DataSize,R,OutFile) da classe pingpong.cerl 50
Programa 3.18: Método run/1 da classe threadring.cerl..52
Programa 3.19: Método create_procs(QtdProcs) da classe threadring.cerl 52
Programa 3.20:Método sender_ring_node/3 de threadring.cerl..53
Programa 3.21: Classe proc...53
Programa 4.1 : Classe weatherData...60
Programa 4.2: Classe statisticsDisplay 61
Programa 4.3: Classe houseBlend..62
Programa 4.4: Classe milk 63
Programa 4.5: Classe starbuzzCoffee 63
Programa 4.6: Classe dependentPizzaStore..65
Programa 4.7: Classe pizzaStore..65
Programa 4.8: Classe pizza 66
Programa 4.9: ClassechicagoStylePepperoniPizza...66
Programa 4.10: Classe nyPizzaStore 67
Programa 4.11: Classe nyPizzaStore 67
Programa 4.12: Interface PizzaIngredientFactory..69
Programa 4.13: Interface nyPizzaIngredientFactory..69
Programa 4.14: Classe pizzaIngredientFactory...70
Programa 4.15: Classe cheesePizza 70
Programa 4.16: Classe pizzaTestDrive 71
Programa 4.17: Interface Command 72
Programa 4.18: Classes que implementam Command...72
Programa 4.19: Classes com métodos encapsulados 72
Programa 4.20: Classe simpleRemoteControl 73
Programa 4.21: Classe remoteControlTest 73
Programa 4.22: Classes adaptadoras..74
Programa 4.23: Classes com métodos encapsulados 75
Programa 4.24 : Classes com métodos encapsulados 76
Programa 4.25: HomeTheaterFacade...78
Programa 4.26: Classe caffeineBeverage...79
Programa 4.27 : Classes concretas tea e coffee..80
Programa 4.28: Classe caffeineBeverageWithHook..80
Programa 4.29 : Classes concretas teaWithHook e coffeeWithHook...81
Programa 4.30: Classe beverageTestDrive 81

1. INTRODUÇÃO

Eu acredito na intuição e na inspiração. A imaginação é mais importante que o conhecimento.
O conhecimento é limitado, enquanto a imaginação abraça o mundo inteiro, estimulando o progresso, dando à luz à evolução.
Ela é, rigorosamente falando, um fator real na pesquisa científica

Albert Einstein
No livro Sobre a Religião Cósmica e Outras Opiniões e Aforismo publicado em 1931

No paradigma de orientação a objetos, um problema no mundo real pode ser representado por meio de objetos que interagem entre si. Dessa maneira, pode-se pensar no paradigma da orientação a objetos como uma evolução dos paradigmas imperativo e funcional. Uma gama de metodologias e ferramentas para o desenvolvimento orientado a objetos surgiu para acompanhar essa demanda. A *Unified Modeling Language* (UML) **[39]** tornou-se um padrão da indústria de software. De acordo com Tiobe **[18]** em julho de 2013, em um escala das dez linguagens de programação mais usadas, oito delas são linguagens orientadas a objeto. Essas são as linguagens em que são desenvolvidas os

grandes sistemas. Acontece que sistemas não pararam de crescer em complexidade e quantidade de usuários.

Jogos via Internet, redes sociais e as novas aplicações web demandam acesso simultâneo e interativo de milhares (às vezes milhões) de pessoas. Esses sistemas são quase sempre desenvolvidos usando linguagens de *script* como PHP [3] ou usando *frameworks* baseados em linguagens como Java [2], Ruby [4] ou Python [5]. À proporção que o acesso a esses sistemas cresce, os fornecedores de tais serviços necessitam atender a novas demandas por meio da substituição de hardware por modelos mais potentes, aumentando seus custos operacionais. Quando o nível de acesso cresce drasticamente, o projetista se vê forçado a reprojetar toda a arquitetura do sistema migrando para soluções complexas usando Java Enterprise Edition (JEE) [11] ou Node.js [42]. Essas soluções também demandam mais e mais servidores. O problema possui uma raiz mais profunda: as linguagens de programação usadas para o desenvolvimento de tais sistemas não foram projetadas para suportar concorrência massiva. Linguagens com suporte a concorrência baseadas no modelo de memória compartilhada não possuem a escalabilidade necessária para atender a demanda [48][55].Três exemplos importantes são apresentados a seguir.

LinkedIn e Java

LinkedIn [75] é uma rede social lançada em 5 de maio de 2003. Ela é utilizada principalmente para contatos profissionais. Em junho de 2012, LinkedIn alcançou 175 milhões de usuários registrados em mais de 200 países e territórios. Eishay Smith em [52] diz que "*LinkedIn é escrito 99% em Java usando uma arquitetura baseada em Spring/Jetty/Tomcat*".

Twitter e Scala

Twitter é um sistema de *microblog* que em março de 2013possuiaaproximadamente 200 milhões de usuários [47]. Nele o usuário escreve uma mensagem curta de no máximo 140 caracteres (*tweet*) e envia para seus seguidores. Quanto mais "famoso" é o usuário, maior é a quantidade de seguidores. Ao receber uma mensagem, o seguidor pode comentá-la e retransmití-la para os seus próprios seguidores. Dependendo do assunto e/ou do usuário, a mensagem pode torna-se viral e ser reenviada milhões de vezes. Isso aconteceu por exemplo, quando o cantor Michael Jackson morreu. O tráfego de mensagens foi tão intenso que todos os servidores do Twitter ficaram sobrecarregados e o sistema como ficou temporariamente indisponível.

As primeiras versões do Twitter foram desenvolvidas em Ruby On Rails [43], um *framework* para desenvolvimento web baseado na linguagem Ruby. À medida que o serviço foi ganhando mais e mais usuários, um problema surgiu: sua arquitetura não suportava a demanda mesmo com o acréscimo de novos servidores. Não adiantava colocar máquinas mais potentes. Para resolver o

problema, os desenvolvedores trocaram a arquitetura do sistema por uma outra desenvolvida em Scala [48], uma linguagem de programação funcional com suporte a concorrência massiva e que funciona sobre a Java Virtual Machine (JVM) [10].

Facebook e Erlang

Facebook é uma rede social que em outubro de 2012 alcançou um bilhão de usuários cadastrados [53]. Aproximadamente 618 milhões de usuários acessam o sítio todos os dias [49]. Cada usuário pode ter até 5000 contatos. Um usuário-empresa pode ter mais de 5000 contatos. Cada vez que um usuário envia uma mensagem (*post*), dependendo da política de retransmissão configurada, ela é retransmitida para todos seus contatos. Cada contato pode "curtir" a mensagem e retransmití-la para seus próprios contatos. Além desse tipo de mensagem, Facebook tem um sistema de bate-papo. Em 2009, o sistema de bate-papo processou aproximadamente 1 bilhão de mensagens por dia [50]. As primeiras versões do Facebook foram desenvolvidas em LAMP (Linux [44]/PHP [3]/Apache [45]/MySQL [46]) [51]. À medida que a quantidade de usuários foi crescendo, os desenvolvedores tiveram de modificar o *backend* do sistema, criar um compilador de PHP para C++ chamado HipHop [51] e portar o sistema de bate-papo (*chat*) para Erlang [1][54][55].

1.1. Linguagens funcionais para o desenvolvimento de aplicações Web

Nos casos do Twitter e Facebook, observa-se um domínio de problema parecido: existe um sistema distribuído [28][29] com uma imensa quantidade de acessos de usuários que trocam mensagens entre si. Esses usuários acessam o sistema via *desktop*, *laptops*, *tablets* e *smartphones*. Além disso, há um "ecossistema" de aplicações que foram criadas para acessar o Twitter e Facebook. A comunicação entre essas aplicações e os sistemas é feita a partir de *web services* SOAP [40] e principalmente RESTful [41]. Para resolver os problemas ocasionados pela concorrência massiva, os desenvolvedores optaram por usar linguagens funcionais como Scala [6] e Erlang [1] na arquitetura do sistema ao invés de linguagens orientadas a objetos como Java. Vinoski [64]observou a volta da popularidade das linguagens funcionais para desenvolvimento Web por causa da expressividade e facilidade que algumas novas linguagens tem de implementar concorrência massiva. Apesar de oferecerem modelos de concorrência diferentes, Java e Erlang [54] são usadas para o desenvolvimento de grandes aplicações web que precisem de concorrência massiva [75][76].

1.2. Modelos de suporte à concorrência

A comunicação entre componentes concorrentes pode ser invisível ao programador ou ser manipulada explicitamente. No Modelo de Memória Compartilhada Virtual (*Virtual Shared Memory*), por exemplo, uma camada de software cuida da comunicação de uma maneira transparente. Nesse modelo de comunicação, variáveis são endereçadas uniformemente usando um endereço virtual "global". Por outro lado, a comunicação manipulada explicitamente pode ser dividida em duas classes: comunicação por memória compartilhada e comunicação por troca de mensagens. Memória compartilhada e troca de mensagens possuem diferentes características de desempenho. Tipicamente, mas nem sempre, a sobrecarga computacional per-processos e para o roteamento de tarefas é menor no modelo de passagem de mensagens, mas a demanda computacional na troca de mensagens é maior por si só que uma chamada de procedimentos. Essas diferenças são encobertas por outros fatores de desempenho.

1.2.1. Concorrência baseada em memória compartilhada e *threads*

Linguagens como Java possuem suporte à concorrência baseado em memória compartilhada e *threads* **[35]**. *Threads* são processos leves que compartilham arquivos e memória, mas cada uma delas possui seu próprio contador de programas, pilha e variáveis locais. *Threads* são a parte central da linguagem Java; elas são usadas desde o gerenciamento da biblioteca de interface gráfica até a implementação do coletor de lixo da Java Virtual Machine (JVM). Goetz **[34]** afirma que os pontos fortes de usar *threads* são: a simplicidade de modelagem, o tratamento simples de eventos assíncronos e o poder de explorar as arquiteturas de computadores multiprocessados. *Threads* não são processos. Portanto, eles não são protegidos uns dos outros pelo sistema operacional. Como os *threads* compartilham memória, uma série de abstrações são usadas (semáforos, monitores, *mutex*) para evitar os problemas comuns de concorrência tais como inanição, impasses, etc. Se não for programado com cuidado, um *thread* poderá interferir com outros, podendo provocar erros difíceis de serem encontrados em tempo de desenvolvimento. Como os *threads* Java contam com os *threads* do sistema operacional, é limitada a quantidade de *threads* que um usuário do sistema operacional pode criar. Além de todos esses problemas já citados, Lee **[56]** diz que *threads*, como modelo de computação, são intrinsicamente não determinísticos.

1.2.2. Concorrência baseada em troca de mensagens

A troca de mensagens pode ser realizada assincronamente **[63]**, ou usar um estilo *rendezvous*, onde o processo que envia uma mensagem espera até a mensagem ser recebida. A passagem de mensagens assíncronas pode ser confiável ou não confiável. Troca de mensagens é mais fácil de ser

entendida que o modelo de memória compartilhada e é tipicamente considerada uma forma mais robusta de programação concorrente. Há uma variedade de teorias matemáticas para entender e analisar os sistemas baseados em troca de mensagens, incluindo o Modelo de Atores (*Actor Model*) **[58]**. Passagem de mensagens também é implementada através de algumas bibliotecas como a *Message Passing Interface* (MPI) **[80]**.

O Modelo de Atores se baseia em abstrações onde atores são entidades isoladas que se comunicam assincronamente umas com as outras. Um ator pode enviar mensagens para outro ator. Um ator pode criar outro ator dinamicamente. As linguagens Scala e Erlang se baseiam no Modelo de Atores **[57][6]**.

Erlang implementa atores como processos **[57]**. Os processos Erlang funcionam sobre a máquina virtual Erlang (*Erlang Virtual Machine – EVM*) e são completamente isolados uns dos outros e também do sistema operacional **[65]**. Erlang possui variáveis de atribuição única (*single assignment*): uma vez que um valor é atribuído, ele não pode ser modificado. A atribuição única e o isolamento de processos afastam problemas comuns de concorrência, como condição de corrida, sem a necessidade de introduzir semáforos e monitores.

De acordo com a referência **[6]**, Scala é uma linguagem de propósito geral projetada para expressar padrões de programação de uma maneira concisa, elegante e tipada. Seus desenvolvedores falam que Scala integra funcionalidades de linguagens orientadas a objetos e linguagens funcionais, permite que programadores Java e outros se tornem mais produtivos. Os tamanhos dos códigos são tipicamente reduzidos a um fator de dois ou três comparados com a linguagem Java.

1.2.3. Comparação de desempenho do suporte a concorrência

A suíte de programas de teste conhecida como Intel MPI Benchmark (IMB) **[19]** oferece uma maneira justa de comparar o desempenho do suporte à concorrência provido por linguagens de programação. Esses testes foram desenvolvidos originalmente para medir o desempenho da programação paralela em arquiteturas de redes locais (*clusters*) e sistemas multiprocessados. Para implementar o *benchmark* em uma determinada linguagem, o programador é obrigado a usar as primitivas de concorrência que a linguagem oferece para criação de processos, sincronização e troca de mensagens entre processos.

Jan Schäfer **[59]** desenvolveu um *framework* baseado em atores para a Máquina Virtual Java e utiliza o *benchmark* IMB para comparar o desempenho entre Scala, Clojure **[7]**. O Computer Language Benchmark Game apresenta uma análise comparativa entre várias linguagens, incluindo Java e Erlang, usando uma versão do IMB SendRec chamada *Threadring* **[20]**. Outros testes podem ser encontrados em **[20]** para o mesmo programa de teste, porém utilizando versões mais antigas das linguagens.

1.3. A linguagem Erlang

Erlang é uma linguagem de programação funcional com suporte a concorrência originalmente desenvolvida pela Ericsson para programação de centrais telefônicas [1].Ela saiu de seu nicho inicial e tornou-se uma alternativa para o desenvolvimento de grandes sistemas para Internet [55][66][78]. A habilidade de suportar concorrência massiva, alta taxa de transferência, tolerância a falhas e a fácil integração com outras linguagens são seus pontos fortes. De acordo com Tiobe [18], em junho de 2013, Erlang está na 31ª posição entre as linguagens mais usadas. Em abril de 2012, ela estava na 44ª posição. O Programa 1.1 mostra um exemplo de código em Erlang. No caso, o "hello world".

```
% welcome to the jungle
-module(helloworld).
-export([hello_world/1]).

hello_world(Name) ->
io:format("Hello World, ~s ~n", [Name]).
```

Programa 1.1: "Hello World" em Erlang

Erlang é dividido em módulos. Cada módulo é um arquivo texto com a extensão `.erl`. Cada módulo inicia com a palavra-chave `-module(nome_modulo)`. Os módulos são divididos em formas e estas terminam em um ponto ("."). Funções são um ponto chave da linguagem. As funções públicas são assinaladas pelo `export([listaFuncoes])`. Listas em Erlang iniciam e terminam com colchetes. Tuplas em Erlang iniciam e terminam com chaves (ex: `{nome,"Jucimar"}`). No exemplo, a função `hello_world` é pública e possui um parâmetro (`hello_world/1`). A assinatura das funções Erlang seguem a regra nome_funcao/cardinalidade. Funções com a mesma assinatura são consideradas a mesma função. Dois pontos(":") são usados para invocar funções de um módulo. No exemplo, a função hello_world invoca a função `format/2` do módulo `io`. A função `io:format/2` é similar a função `printf()` da linguagem C. Neste caso, ~s é substituído por Name e formatado como string. O "`~n`" indica uma quebra de linha semelhante ao"`\n`" da linguagem C.

O Programa 1.2 mostra algumas características da sintaxe de Erlang através da implementação do quicksort.

```
-module(quicksort).
-export([qsort/1]).

qsort([])->[];
qsort([Pivot|Rest])->
qsort([X||X<-Rest,X<Pivot])++
[Pivot]++
qsort([Y||Y<-Rest,Y>=Pivot]).
```

Programa 1.2: Quicksort em Erlang

No caso, a função `qsort` precisa de um parâmetro do tipo lista. Se o parâmetro for uma lista vazia (`[]`), a função retorna uma lista vazia. Senão, a função quebra a lista em duas partes: `Pivot`, primeiro item da lista e `Rest`, o resto da lista sem o primeiro item e aplica o processamento. O processamento desta função é baseado em compreensão de lista, concatenação de listas e recursão. A expressão `[X || X <- Rest, X < Pivot]` compara `X` com o `Pivot` e se `X` for menor que o `Pivot`, acrescenta na lista resultante. A lista resultante é processada recursivamente por `qsort`. A expressão `++` concatena listas. A expressão .`[Y || Y <- Rest, Y >= Pivot]` compara `X` com o `Pivot` e se `X` for maior ou igual que o `Pivot`, acrescenta na lista resultante, a qual é processada recursivamente por `qsort`.

1.3.1. Concorrência em Erlang

Erlang provê um modelo de concorrência simples, baseado no Modelo de Atores, que permite a criação de processos leves. A comunicação entre os processos é realizada por meio de troca de mensagens: um processo envia uma mensagem assincronamente para outro processo que armazena a mensagem em uma "caixa de correio" (*mailbox*). Cada processo possui ainda um dicionário interno (tabela *hash*) para armazenar informações. É possível criar facilmente milhares de processos. É possível fazer processos hospedados em máquinas diferentes se comunicarem de maneira transparente e com bom desempenho. A sintaxe para o uso de processos é bem simples e intuitiva. O Programa 1.2 mostra uma versão concorrente do quicksort em Erlang, extraído de **[31]**.

```
-module( quicksort ).
-export([ pqsort/1 ]).

pqsort([]) -> [];
pqsort([Pivot]) -> [Pivot];
pqsort([Pivot|Rest]) ->
    io:format("+", []),
    Left = [X || X <- Rest, X < Pivot],
    Right = [Y || Y <- Rest, Y >= Pivot],
    [SortedLeft, SortedRight] = plists:pmap(fun pqsort/1, [Left, Right]),
    io:format("-", []),
    SortedLeft ++ [Pivot] ++ SortedRight.
```

Programa 1.3: Quicksort concorrente em Erlang

O Programa 1.2 é muito parecido com a versão apresentada em Programa 1.1. A principal diferença é apresentada no uso da função `pmap/2`. O código-fonte dessa função é apresentado no Programa 1.3.

```
-module(plists).
-export([pmap/2]).

pmap(F, L) ->
    S = self(),
    Pids = lists:map(fun(I) -> spawn(fun() -> pmap_f(S, F, I) end) end, L),
    pmap_gather(Pids).

pmap_gather([H|T]) ->
    receive
        {H, Ret} -> [Ret|pmap_gather(T)]
    end;
pmap_gather([]) ->
    [].

pmap_f(Parent, F, I) ->
    Parent ! {self(), (catch F(I))}.
```

Programa 1.4: Quicksort concorrente em Erlang

A função `pmap/2` é semelhante à função `map/2`, porém executando cada chamada de função em um processo. Os processos são executados em paralelo e no final uma nova lista é criada. A função interna `spawn/1` cria um processo. Neste caso, ela cria um processo para a chamada da função `pmap_f/3`. A função `pmap_f/3` executa uma determinada função passada como parâmetro. Uma mensagem é enviada usando o símbolo de exclamação (`!`). Na expressão `Parent ! {self(), (catch F(I))}.`, `Parent ! {self(), (catch F(I))}` é enviado como mensagem para `Parent`. A caixa postal do processo é gerenciada pela função `pmap_gather/1`: ao receber uma mensagem, ela vai concatenando a lista. A função `catch/1` captura qualquer erro de execução que ocorra.

1.3.2. Erlang e a orientação a objetos

Joe Armstrong, criador da linguagem Erlang, argumenta que, usando o contexto de Alan Kay sobre orientação a objetos **[77]**, Erlang seria a única linguagem verdadeiramente orientada a objetos **[83]**. Porém, a linguagem não possui uma sintaxe com uma boa expressividade para a orientação a objetos. É necessário um grau de experiência elevado do desenvolvedor para modelar o mundo real com a "programação orientada a processos" típica do Erlang e depois pensar isso como objetos. Na orientação a objetos, o "abismo semântico" entre o mundo real e o programa é reduzido. Isso não acontece quando se tenta modelar "orientado a processos", como é feito em Erlang. Um programador não consegue olhar para uma determinada cena e ver processos se comunicando. Ele vê objetos se comunicando. Para exemplificar essa falta de expressividade, um pequeno exemplo foi modelado de acordo com a Figura 1.1. Ele é um exemplo do padrão de projeto *Bridge* **[38]**

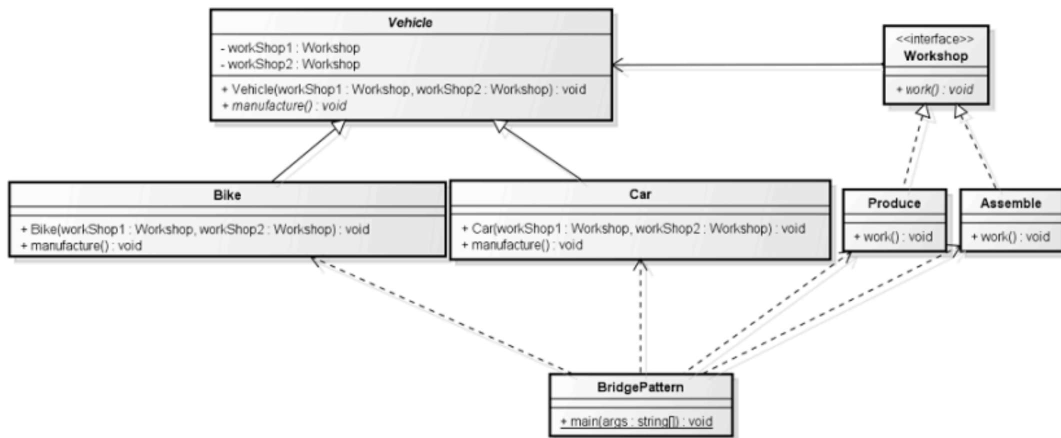


Figura 1.1: Exemplo do Padrão de Projeto *Bridge*

Neste exemplo, o objetivo é modelar e codificar a construção de bicicletas e carros. Ambos possuem métodos de fabricação diferentes. Deve-se evitar repetição de código que aconteceria pela herança [67]. Para resolver isso, os comportamentos são descritos através da interface Workshop e implementados pelas classes Produce e Assemble. A implementação da classe Bike em Java é apresentada no Programa 1.5

```
public class Bike extends Vehicle{

      public Bike(WorkshopworkShop1,WorkshopworkShop2){
      super(workShop1,workShop2);
      }

      public void manufacture(){
      System.out.print("Bike ");
      workShop1.work();
      workShop2.work();
      }
}
```

Programa 1.5: Classe Bike em Java

O código apresentado no Programa 1.5 é simples e fácil de entender. Java possui uma sintaxe expressiva para escrever programas orientados a objetos. O Programa 1.6 mostra a classe Bike implementada em Erlang.

```
-module(bike).
-export([new/2,manufacture/0]).

new(WorkShop1,WorkShop2)->
      Dict=orddict:new(),
      Dict2=orddict:store(workshop1,WorkShop1,Dict),
      Dict3=orddict:store(workshop2,WorkShop2,Dict2),
      PidOfObject=spawn(bike,object_running,[Dict3]),
      PidOfObject.
```

```
manufacture()->
      io:format("Bike ").

object_running(Dict)->
      receive
        {Sender,manufacture}->
                 manufacture(),
             caseorddict:find(workshop1,Dict)of
                 {ok,Value}->
                    Value!{self(),work};
            _->
                    io:format("")
             end,
                 caseorddict:find(workshop2,Dict)of
                 {ok,Value2}->
                    Value2!{self(),work};
            _->
                    io:format("")
             end,
             object_running(Dict)
    end.
```

Programa 1.6: Classe Bike em Erlang

Apesar de ser possível construir a referida classe, a sintaxe de Erlang não facilitou a programação. Vários artifícios de programação foram necessários para implementar uma classe simples. Isso dificultaria o desenvolvimento de programas orientados a objeto em Erlang. Além disso, neste exemplo não foi possível implementar herança.

Portanto, por não possuir uma sintaxe expressiva para o desenvolvimento de programas orientados a objetos, para a grande maioria dos desenvolvedores, Erlang não é orientado a objetos. E por não ser uma linguagem orientada a objetos, afasta uma boa parcela de programadores.

1.4. Motivação

Uma linguagem orientada a objetos deve possuir expressividade para facilitar o trabalho dos desenvolvedores. Dotar Erlang de um suporte a orientação a objetos pode ser uma estratégia para levar a linguagem a um público que hoje está acostumando a programar em Java e outras linguagens. De fato, não há uma incompatibilidade fundamental entre Erlang e a programação orientada a objetos.

Existem algumas implementações de extensões de orientação a objetos para Erlang. Corrado Santoro desenvolveu um framework chamado eXAT **[60]** para trabalhar com agentes inteligentes. Richard Carlsson criou o "Parameterized modules" **[61]**, onde os módulos Erlang são criados através de parâmetros. Isso é similar à orientação a objetos, mas não há suporte a herança e os objetos não possuem identificadores únicos. Gábor Fehér e Andras Békér desenvolveram o ECT **[62]**, a mais nova extensão orientada a objetos para Erlang. Ela cria objetos usando *records* de

Erlang. Seus testes indicaram um desempenho melhor que outras abordagens, mas sua sintaxe não é expressiva, o que dificulta a legibilidade de programas. Um exemplo de código em ECT retirado de [81] é apresentado no Programa 1.7:

```
-include_lib("ect/include/ect.hrl").
-class(animal).
?FIELDS({weight, color = red}).
?METHODS([method1/2, method2/2]).

method1(This, A) ->
   A.
method2(This, A) ->
   A+1.
```

Programa 1.7: Classe Animal em ECT

Este código mostra a criação de uma classe *Animal* com os atributos *weight*, *color* e os métodos *method1* e *method2*. O Programa 1.8 mostra um exemplo de herança em ECT

```
-include("class1.class.hrl").
2
3 -class(class2).
4 -superclass(class1).
5 ?FIELDS({x, y, z}).
6 ?METHODS([method2/2]).
7
8 method2(_, _) -> ok.
```

Programa 1.8: Herança em ECT

Neste exemplo, a *Class1* é subclasse de *Class2*. O uso de classes é mostrado no Programa 1.9

```
...
demo() ->
X = #classb{b = 6},
#classb{a = A1, b = B1} = X,
#classa{a = A2, b = B2} = X,
Y1 = X#classb{a = 1, b = 2},
Y2 = X#classa{a = 1, b = 2},
C1 = Y#classb.c,
C2 = Y#classa.c,
...
```

Programa 1.9: Instanciação de objetos em ECT

Esse programa instancia uma *classb*, depois extrai o valor de seus atributos, modifica o valor desses atributos e depois extrai o valor de um atributo único. Apesar de a sintaxe ser familiar a um programador Erlang, expressar classes e objetos através de registros e tuplas não é muito atraente para um programador acostumado com uma linguagem orientada a objetos mais expressiva como Java. ECT também não possui suporte a primitivas importantes como interfaces, métodos abstratos e chamada a métodos de superclasses.

Existem linguagens que funcionam sobre a máquina virtual do Erlang como Elixir **[8]** e Efene **[9]**. Porém, elas não usam a sintaxe do Erlang, o que obriga um desenvolvedor a aprender mais uma linguagem. Para programação paralela com MPI existem vários dialetos de C e FORTRAN **[80]**. Dentre eles, podemos destacar o PObC++ **[79]**, uma extensão da linguagem C++, portanto com suporte a orientação a objetos, voltada para programação paralela. Existe uma implementação do Erlang na Java Virtual Machine, o Erjang **[82]**.

1.5. Objetivo

Esta tese apresenta uma extensão orientada a objetos para a linguagem Erlang, chamada $_{oo}$*Erlang*, que possui uma melhor expressividade para resolução de problemas do "mundo real" e que não degrade o desempenho da linguagem. As ideias básicas por trás de $_{oo}$*Erlang* foram mostradas em **[71][72]** e são detalhadas nos capítulos seguintes desta tese. O nicho da extensão é o mesmo de Erlang: desenvolver sistemas *backend* para grandes aplicações onde a concorrência massiva e tolerância a falhas são requeridas. Um dos pontos mais importantes é que a extensão seja conservativa, de forma que qualquer programa Erlang deva ser passivo de compilação por $_{oo}$*Erlang* sem a menor alteração.

1.5.1. Objetivos específicos

Os objetivos específicos deste trabalho são:

- Comparar o desempenho da extensão $_{oo}$*Erlang* com a própria linguagem Erlang, Java, Scala, Python e Ruby por meio dos testes Intel MPI Benchmark. Um teste com um subconjunto destas linguagens foi apresentado nas referências **[70][73]**.
- Criar uma sintaxe para a extensão mais próxima possível de Java **[13]** sem contudo se distanciar do Erlang. A linguagem Java foi escolhida por ser popular e ter uma boa expressividade.
- Avaliar a expressividade da extensão comparando códigos escritos em Erlang com códigos escritos em $_{oo}$*Erlang*.

1.5.2. Questões de pesquisa

O projeto de pesquisa realizado no trabalho de doutorado aqui relatado teve como guia um conjunto de questionamentos de pesquisa que serviram para a definição dos objetivos e elaboração de hipóteses. As perguntas de pesquisa relacionadas são:

- Para o desenvolvimento real de sistemas, um atributo muito importante de uma linguagem orientada a objetos é o suporte a implementação a Padrões de Projetos **[39]**.

Através desses padrões, os sistemas podem ser programados com baixo acoplamento e alta coesão, características fundamentais para manutenção e reuso de código. Como implementar os Padrões de Projeto em $_{oo}$*Erlang*?

- É possível criar uma extensão que não degrade o desempenho da linguagem Erlang?
- É possível criar uma extensão que seja expressiva?

O presente trabalho descreve as atividades de investigação que buscaram responder essas questões e seus resultados.

1.6. Contribuições

As principais contribuições desta tese são:

- Uma extensão orientada a objetos de alto desempenho para a linguagem Erlang;
- A implementação dos testes Intel MPI benchmark em Erlang, Java, Scala, Python e $_{oo}$*Erlang*;
- A implementação dos 23 padrões de Projeto do *"Gang of Four*" em $_{oo}$*Erlang.*

1.7. Estrutura desta tese

Esta tese é composta, além desta introdução, por cinco capítulos. O Capítulo 2apresenta $_{oo}$*Erlang* em detalhes: a sintaxe, semântica e implementação do $_{oo}$*Erlang*. O Capítulo 3 mostra a implementação de programas desenvolvidos em $_{oo}$*Erlang* que exemplificam o uso de suas características de orientação a objetos. O Capítulo 4 descreve a implementação de Padrões de Projeto em $_{oo}$*Erlang*. O Capítulo 5 mostra os resultados da análise de desempenho entre Java, Erlang, Scala, Pyhton e Ruby e $_{oo}$*Erlang* utilizando a suíte de testes Intel MPI Benchmark (IMB). O Capítulo 6 conclui este trabalho com as considerações finais e propostas de trabalhos futuros.

2. Sintaxe, Semântica e Implementação

Do princípio forte da herança, toda a variedade selecionada tenderá a propagar a sua nova e modificada forma.

Charles Darwin em
A Origem das Espécies. 1859

$_{oo}$*Erlang* é uma extensão da linguagem Erlang para o suporte a orientação a objetos. Este capítulo mostra alguns exemplos simples de $_{oo}$*Erlang*. Depois mostra a sintaxe, semântica e implementação. A extensão da linguagem está implementada em toda a sua funcionalidade, e pode ser obtida em no sítio `https://sites.google.com/site/ooerlang1/`.

2.1. Exemplos

Esta seção mostra alguns exemplos simples modelados em UML [37] que demonstram a sintaxe de $_{oo}$*Erlang*. A semântica e decisões de projetos são apresentadas em 2.3.

2.1.1. Instanciação de objetos

Este exemplo mostra a instanciação de objetos em $_{oo}$*Erlang*. As classes utilizadas podem ser vistas na Figura 2.1:

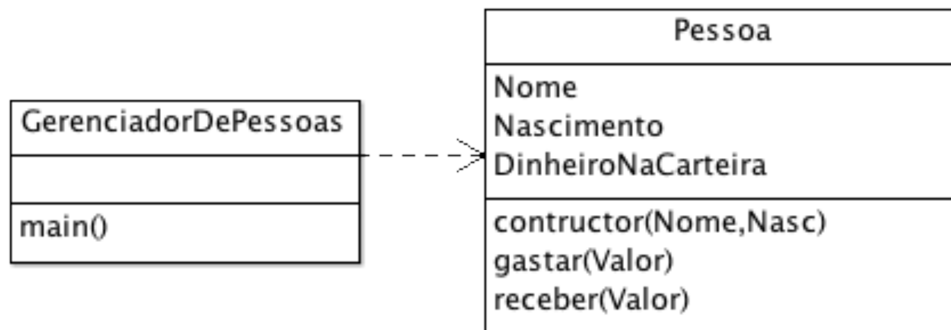


Figura 2.1 : Instanciação de objetos

A classe `pessoa.cerl` é uma classe com três atributos e três métodos. Os atributos são `Nome`, `Nascimento`, `DinheiroNaCarteira`. Os métodos são `gastar()`, `receber()`. O construtor da classe é `constructor()`. A classe `gerenciadorDePessoas.cerl` possui o método `main()`, podendo instanciar objetos do tipo pessoa e utilizar os métodos `gastar()` e `receber()`. O Programa 2.1 mostra a conversão para $_{oo}$*Erlang*.

```
-class(pessoa).
-constructor([constructor/2]).
-export([gastar/1, receber/1]).

attributes.
Nome;
Nascimento;
DinheiroNaCarteira.

methods.
constructor(Nome, Nasc) ->
self::Nome = Nome,
self::Nascimento = Nasc,
self::DinheiroNaCarteira = 0.

gastar(Valor) ->
self::DinheiroNaCarteira = self::DinheiroNaCarteira - Valor.

receber(Valor) ->
self::DinheiroNaCarteira = self::DinheiroNaCarteira + Valor.
```

Programa 2.1: Declarações iniciais do fonte pessoa.cerl

A primeira linha define o nome da classe. Essa declaração é semelhante ao uso do “`module`” em Erlang. O construtor da classe é definido na segunda linha através da palavra “`constructor`”. Nesse caso, ele foi nomeado como *constructor*, mas poderia ser qualquer nome de função válida em

Erlang. A palavra reservada `“exports”` declara os métodos que serão exportados, ou seja, serão públicos e visíveis para outras classes. Utilizando a palavra reservada `“attributes”` inicia-se a seção de declaração de atributos. Os nomes dos atributos seguem as regras das variáveis em Erlang, tais como iniciar com letra maiúscula e tipagem dinâmica. A separação entre as variáveis é feita usando ponto-e-vírgula (“;”). A última variável marca o final da seção de atributos com um ponto (“.”). Como em Erlang uma variável pode receber uma atribuição de qualquer tipo, não é necessário tipificar cada atributo. Se uma classe não possui atributos, não é necessário criar a seção de atributos.

Para iniciar a seção de declaração de métodos, é utilizada a palavra reservada `“methods”`. Os métodos podem ser definidos, sem diferenças em relação à forma de definir funções em Erlang. A principal diferença desses métodos para uma função Erlang é o uso da palavra reservada `“self”`. Seu uso é muito semelhante à palavra reservada `“this”` da linguagem Java, servindo para diferenciar variáveis que possuem nomes parecidos com os nomes dos atributos. O dois pontos duplos (“`::`”) é utilizado como o ponto (“`.`”) na linguagem Java, servindo para que um objeto acesse seus métodos e atributos. Dois pontos duplos(“`::`”) são utilizados por Erlang já utilizar dois pontos(“`:`”), tornando mais fácil a adoção por programadores Erlang. A classe `gerenciadorDePessoas.cerl` é apresentada no Programa 2.2:

```
-class(gerenciadorDePessoas).
-export([main/0]).

class_methods.

main() ->
    PVitor = pessoa::constructor("Vitor Fernando Pamplona", "07/11/1983"),
    PVitor::receber(1000.00),
    PJoao = pessoa::constructor("João da Silva", "18/02/1970"),
PJoao::receber(500.00),
    PJoao::gastar(100.00).
```

Programa 2.2 : Classe gerenciadorDePessoas.cerl

Observa-se que a função `main/0` é uma função estática, pois fica dentro da seção `class_methods`. Uma função estática é uma função da classe e não pode ser acessada por instâncias dos objetos. Neste caso, ela é usada como uma função `public static main()` do Java para instanciar e utilizar outras classes. Os objetos `“PVitor”` e `“PJoao”` são instanciados e seus métodos utilizados.

2.1.2. Herança

Este exemplo mostra a instanciação de objetos em $_{oo}$*Erlang*. As classes utilizadas podem ser vistas na Figura 2.2.

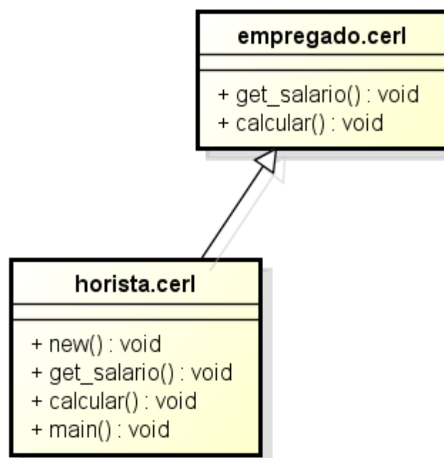


Figura 2.2 : Herança

A classe `empregado.cerl` possui os métodos `get_salario()`e `calcular()`. A classe horista.cerl é uma subclasse de `empregrado.cerl` e redefine `get_salario()` e `calcular()`. A codificação da classe `empregado.cerl` em $_{oo}$*Erlang* é mostrada no Programa 2.3:

```
-class(empregado).
-export([get_salario/0, calcular/0]).

methods.
get_salario() -> -1.
calcular() -> io:format("Salario da superclasse:~p ~n", [self::get_salario()] ).
```

Programa 2.3: Código-fonte da classe empregado.cerl

A classe `horista.cerl` é mostrada no Programa 2.4:

```
-class(horista).
-extends(empregado).
-export([get_salario/0, main/0,new/0,calcular/0]).
-constructor([new/0]).

methods.
new() -> ok.
get_salario() ->    -2.

calcular() ->
    super::calcular(),
io:format("Salario da classe:~p ~n", [ self::get_salario() ] ).

class_methods.
main() ->
    H = horista::new(),
H::get_salario(),
H::calcular().
```

Programa 2.4: Declarações iniciais da classe horista.cerl

A herança é definida pela palavra reservada "extends". No exemplo, a classe horista.cerl estende a classe empregado.cerl. A função new/0é definida como o construtor.

Os métodos calcular/0 e get_salario/0 são redefinidos na subclasse. O método calcular/0 redefine o método calcular/0 da classe empregado.cerl. Além disso, ele invoca o método da superclasse usando a palavra reservada "super". Já o método estático main/0 instancia um objeto do tipo horista e chama o método get_salario/0 da própria classe (retorna -2) e depois chama calcular/0, que chama o get_salario/0 da superclasse e da própria classe novamente.

2.2. Compilação

As classes $_{oo}$*Erlang* são escritas em arquivos texto com extensão .cerl. Em um programa orientado a objetos, todas as classes pertencentes ao projeto deverão estar no mesmo diretório ou um diretório apontado na configuração do compilador. O compilador $_{oo}$*Erlang* extrai os tokens e a árvore sintática por meio do analisador léxico (*scanner*)e do analisador sintático (*parser)*. O analisador sintático **[21][22]** foi construído utilizando a ferramenta Yecc **[16]**, um gerador LALR-1 de analisadores sintáticos similar ao YACC, estendendo o analisador léxico/sintático do Erlang **[14]**.Yecc não possui *front-ends* para evolução automática. A árvore sintática e os *tokens* são passados para o pré-processador que converte a árvore sintática do $_{oo}$*Erlang* para a árvore sintática do Erlang **[15]**, gerando um arquivo .erl. O compilador Erlang processa o arquivo .erl e gera o .beam, um arquivo *byte-code* Erlang **[30][31][32]**. Os principais módulos do $_{oo}$*Erlang* são descritos na Tabela 2.1

Tabela 2.1 : Principais módulos do ooErlang

Módulo	**Descrição**
ast.erl	Geração e manipulação da árvore sintática do $_{oo}$*Erlang*
core.erl	Geração e manipulação da árvore sintática do Erlang
st.erl	Manipulação de informações sobre as classes, variáveis e objetos
ooe.erl	Manipulação objetos durante a execução do programa
ooel.erl	Compilação dos programas .cerl invocando os outros módulos
ooel_parse.yrl	Descrição da sintaxe do $_{oo}$*Erlang* no formato do YECC
ooel_parse.erl	Geração do *parser*
ooel_scan.erl	Geração do *scanner*
ooel_errors.erl	Manipula dos erros de compilação

A Figura 2.3 mostra um esquema com detalhes internos da compilação:

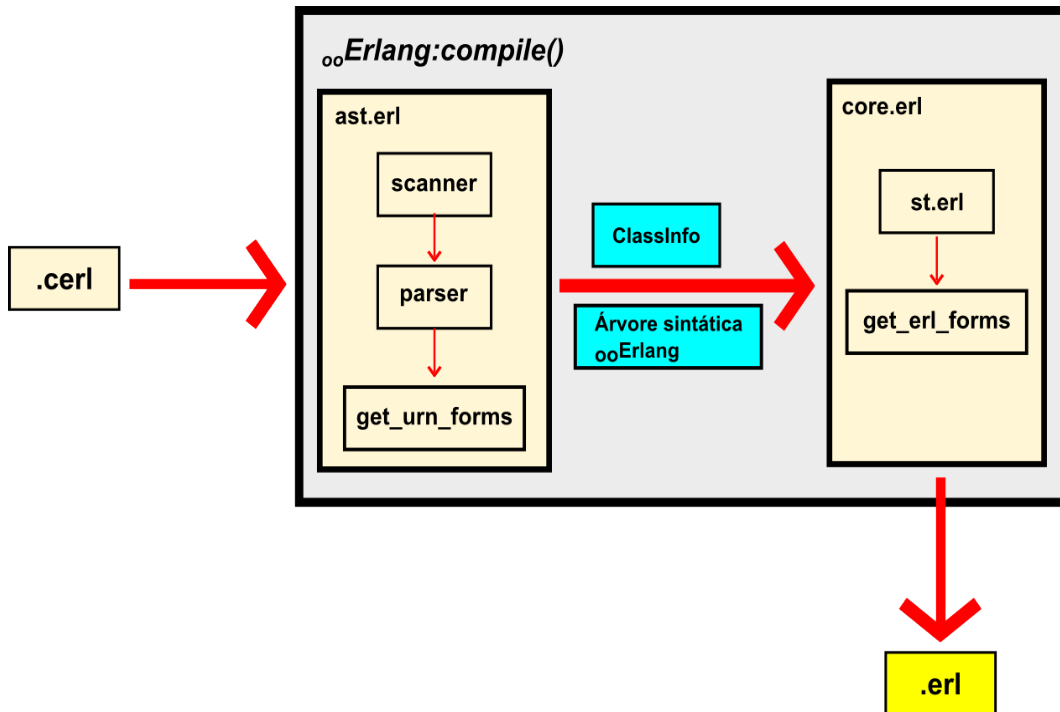


Figure 2.3 : Fluxo interno da compilação no *ooErlang*

O arquivo `.cerl` é passado como parâmetro para a função `ooel:compile()`. Dentro dessa função, o arquivo é passado para o módulo `ast.erl`. O módulo extrai os *tokens* e as regras de sintaxe e forma a árvore sintática abstrata do *ooErlang* (OOAST). São coletadas também todas as informações referentes às classes. A OOAST e as informações das classes são passadas para o módulo `core.erl`. O módulo `st.erl` é uma estrutura de dados que utiliza um dicionário (tabela *hash*) para manipular e verificar as classes, variáveis e outros dados importantes relativos à semântica. Esses dados e OOAST são processados e convertidos em código Erlang válido e compilável.

2.3. Sintaxe e semântica

Erlang é uma linguagem simples e de fácil legibilidade [30][31][32]. *ooErlang* foi projetada para acrescentar o suporte a orientação a objeto sem ferir essas qualidades.

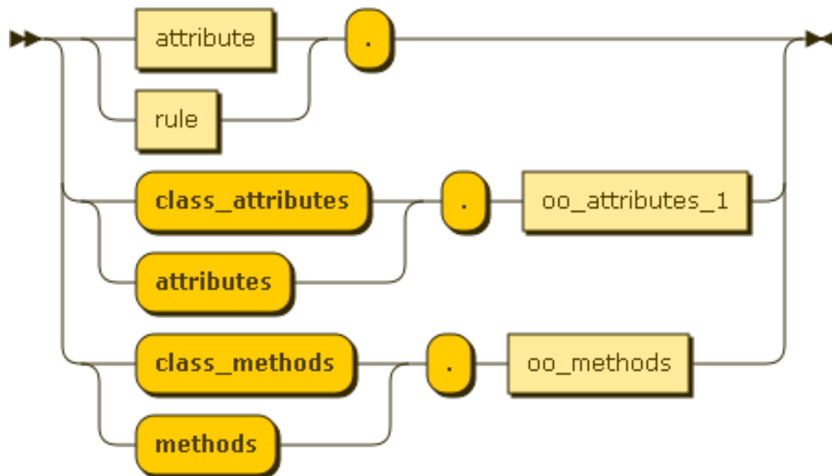
2.3.1. EBNF

O Programa 2.5 mostra a notação *Extended Backus–Naur Form*(EBNF) [21][23] para a gramática do *ooErlang*. Esse trecho é adicionado a especificação Erlang [14] para o suporte às novas palavras-chave e *tokens* [22][24]. Os gráficos são construídos por meio de uma ferramenta gráfica [17]. A versão completa da EBNF encontra-se no sítio do projeto [74].

```
form ::=
    (attribute | rule) '.' |
    (('class_attributes' | 'attributes') '.'  oo_attributes_1) |
    (('class_methods' | 'methods') '.' oo_methods)

oo_attributes_1 ::= (oo_attribute ( '.' |';' oo_attributes) | )
oo_attributes ::= oo_attribute (';' oo_attributes | '.')

oo_attribute ::= var ( |'=' exprs)

oo_methods ::= (function '.' oo_methods | )

function ::= function_clauses

function_clauses ::= function_clause ( | function_clauses )

function_clause ::= atom clause_args clause_guard (clause_body|)

expr_800 ::= expr_900 ((':'|'::') expr_max |)
```

Programa 2.5: EBNF do $_{oo}$*Erlang*

A regra inicial da gramática Erlang é o `form` e seu diagrama de sintaxe é apresentado na Figura 2.4. A partir dele, deriva-se o `attribute`, utilizado para palavras-chaves, tais como `module` e `exports`, do Erlang e as novas palavras-chave do $_{oo}$*Erlang*, tais como `class`, `extends`, `implements`. Do `form`, também derivam-se os atributos e métodos do $_{oo}$*Erlang*.

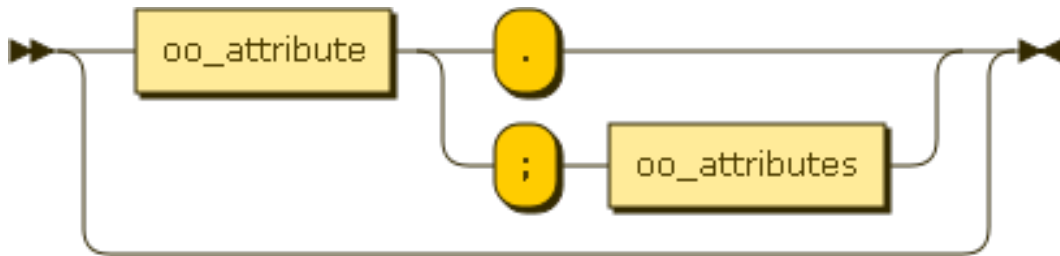



```
form ::=
    (attribute | rule) '.' |
    (('class_attributes' | 'attributes') '.'  oo_attributes_1) |
    (('class_methods' | 'methods') '.' oo_methods)
```

Figura 2.4 : Diagrama de sintaxe do *form*

Os atributos são tratados pela regras `oo_attributes1` e `oo_attributes`. Optou-se por seguir a regra do Erlang para separação das `forms` e então os atributos são separados entre si por um ponto-e-vírgula. Depois do último atributo é colocando um ponto (“.”). A regra `oo_attributes1` é

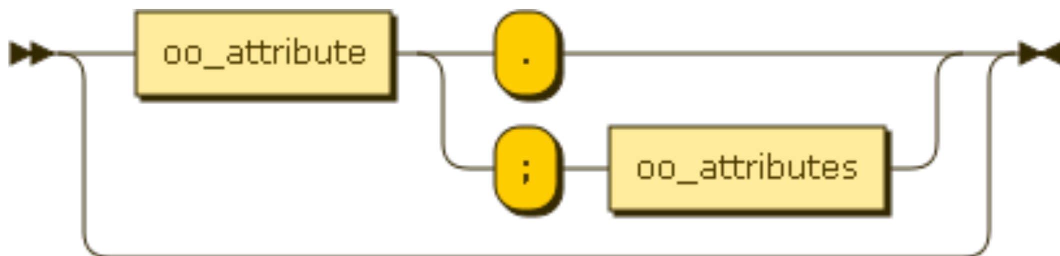
criada para evitar ambiguidade, e seu diagrama é apresentado na Figura 2.5.O atributo pode ser vazio.




```
oo_attributes ::= oo_attribute (';' oo_attributes | '.')
```

Figura 2.5 : Diagrama de sintaxe do oo_attributes

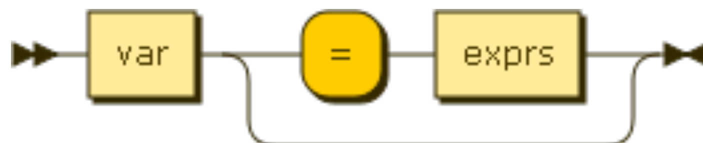
A regra `oo_attributes` é apresentada na Figura 2.6.




```
oo_attributes_1 ::= (oo_attribute ( '.' |';' oo_attributes) | )
```

Figura 2.6 : Diagrama de sintaxe do *attributes_1*

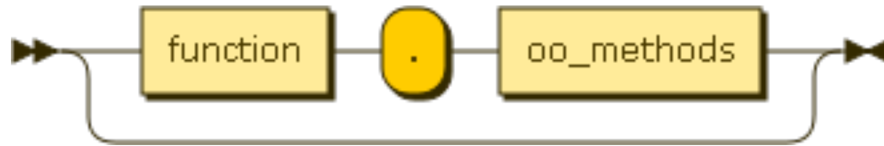
A regra `oo_attribute` é apresentada na Figura 2.7. Cada atributo pode ser uma variável ou uma variável inicializada com um valor (`exprs`).




```
oo_attribute ::= var ( |'=' exprs)
```

Figura 2.7 : Diagrama de sintaxe do *oo_attribute*

A regra `oo_method` é apresentada na Figura 2.8. Os métodos são como funções Erlang e são separados por pontos.



`oo_methods ::= (function '.' methods | )`

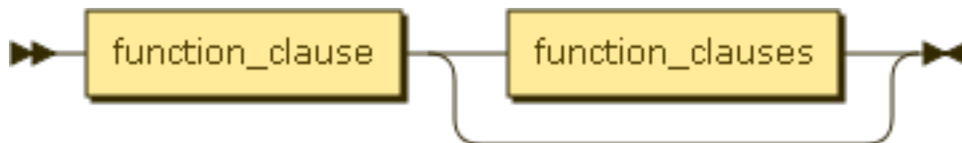
Figura 2.8 : Diagrama de sintaxe do *oo_method*

A regra `function` é apresentada na Figura 2.9. As funções são formadas por `function_clauses`.

`function ::= function_clauses`

Figura 2.9 : Diagrama de sintaxe do *function*

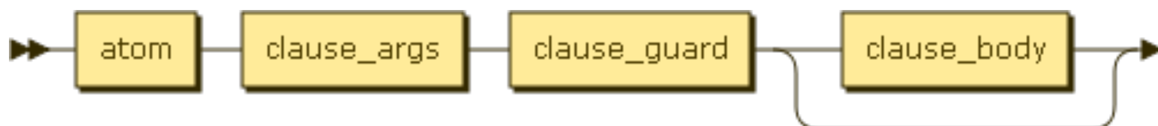
A regra `function_clauses` é apresentada na Figura 2.10. As cláusulas são formadas por nenhuma, uma ou mais `function_clause`.



`function_clauses ::= function_clause ( | function_clauses )`

Figura 2.10 : Diagrama de sintaxe do *function_clauses*

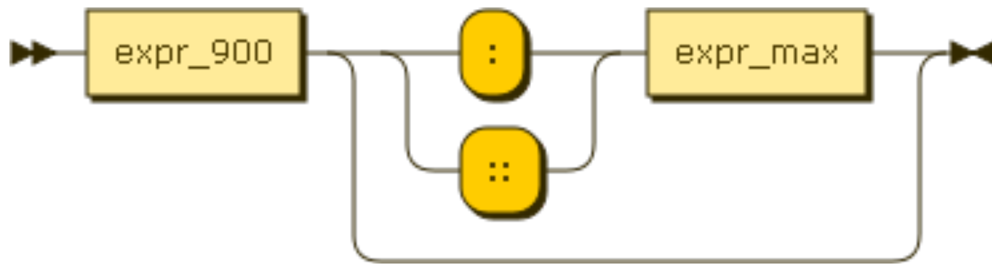
A regra `function_clause` é apresentada na figura 2.11. Uma cláusula (função) tem seus argumentos e pode ter ou não um corpo (no caso de interfaces).



`function_clauses ::= function_clause (|function_clauses)`

Figura 2.11 : Diagrama de sintaxe do *function_clause*

Os dois-pontos-duplos ("::") são acrescentados pela alteração da regra *expr_800* do Erlang e seu diagrama é apresentado na Figura 2.12.




```
expr_800 ::= expr_900 ((':'|'::') expr_max |)
```

Figura 2.12 : Diagrama de sintaxe do *expr_800*

2.3.2. Classes e Objetos em *$_{oo}$Erlang*

Na compilação, as classes são transformadas em módulos Erlang. As informações das classes são extraídas das árvores sintáticas e armazenadas em um dicionário (ST) através do módulo `st.erl`. Para cada classe é criada uma entrada na ST. A chave dessa entrada é o próprio nome da classe e, se houver, o valor é uma tupla com o nome da superclasse, incluindo: uma lista com dados dos atributos, uma lista com dados dos métodos e uma lista com dados dos construtores.

$_{oo}$Erlang implementa objetos como processos Erlang **[32]**. Cada objeto possui um identificador único, o ObjectID, que corresponde ao identificador do processo Erlang. Os atributos são armazenados dentro do dicionário de processos **[31]** de cada processo. Em tempo de compilação, os objetos são referenciados conforme Figura 2.13. Em tempo de execução, os objetos comunicam-se através de troca de mensagens.

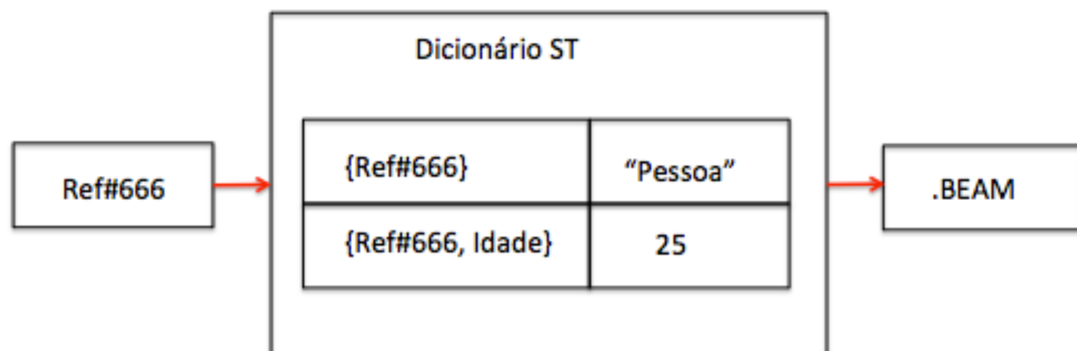


Figura 2.13 : Objetos em *$_{oo}$Erlang*

O construtor cria uma instância da classea que pertence e guarda a referência ao objeto criado na variável de nome ObjectID na ST. As expressões definidas no corpo do construtor são compiladas como um método de objeto comum. Ao final do corpo da função, também é acrescido o nome da variável ObjectID, pois assim a função sempre retornará a referência ao objeto criado. Por

exemplo, o Programa 2.6 é compilado da seguinte forma:

```
-class(pessoa).
-constructor([new/1]).

attributes.
Idade.

methods.
new(MinhaIdade) ->
self::Idade = MinhaIdade.
```

Programa 2.6 : Compilação do constructor

- O parâmetro "MinhaIdade" é declarado como uma variável e seu valor é armazenado.
- É criada uma referência para o objeto (ex: ref#666) e declara-se o atributo "Idade" (ex: : {ref#666, Idade}). A tupla é armazenada no dicionário ST.
- O "self::" indica ao compilador que deve ser usada referencia do objeto atual, no caso "ref#666". Então, é buscada no dicionário a entrada "{ref#666,Idade}".
- O valor da variável "MinhaIdade" é buscado e atribuído a "{ref#666,Idade}".
- A referência do objeto (ex: ref#666) é retornada para quem chamou o constructor.

Em tempo de execução, os objetos são gerenciados pelo módulo `ooe.erl`. Como cada objeto é um processo Erlang, cada objeto possui seu próprio dicionário interno. Esse dicionário é utilizado para armazenar os dados dos objetos e seu ObjectID. Os objetos são reciclados por meio do *garbage collector* da máquina virtual do Erlang.

2.3.3. Herança em $_{oo}$*Erlang*

$_{oo}$*Erlang* não aceita herança múltipla para evitar o aumento da complexidade e ambiguidade. Em contrapartida, $_{oo}$*Erlang* aceita implementações de múltiplas interfaces. Para dar suporte à herança, as informações das classes são mescladas com as informações da superclasse. Primeiramente, as informações da classe são inseridas na ST. Depois, o compilador percorre a superclasse e insere na ST os atributos e métodos que não constam na classe. Se for encontrada a palavra-chave "`super::`", ela é substituída pelo nome da superclasse para verificação da existência dos métodos solicitados. A classe pode sobrescrever os métodos e atributos da superclasse.

2.3.4. Vinculação dinâmica em $_{oo}$*Erlang*

Durante a execução, $_{oo}$*Erlang* é capaz de fazer vinculação dinâmica. A Figura 2.14 apresenta um exemplo.

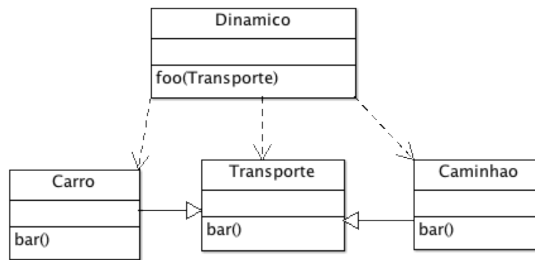


Figure 2.14 : Vinculação dinâmica

A classe transporte possui um método `bar()`. As classes `carro` e `caminhão` redefinem `bar()` de acordo com suas particularidades. A classe Dinamico possui um método `foo()` que aceita como parâmetro um objeto do tipo Transporte e invoca seu método `bar()`. O método `foo()` pode aceitar Transporte e qualquer uma de suas subclasses e decidir, em tempo de execução, qual é o método `bar()` correto que deve ser invocado. O código-fonte da classe `Dinamico` pode ser visto no Programa 2.7.

```
-class(dinamico).
-export([foo/0]).

class_methods.

foo(Transporte) ->
io:format("Transporte: ~p~n", [Transporte::bar()]).

main() ->
Transp = transporte::new_(),
Caminhao = caminhao::new_(),
Carro = carro::new_(),
foo(Transporte), %% chama Transporte::bar()
foo(Caminhao), %% chama Caminhao::bar()
foo(Carro). %% chama Carro::bar()
```

Programa 2.7 : Vinculação dinâmica

O vínculo dinâmico é facilitado por Erlang possuir tipagem dinâmica e os objetos possuírem seus dados armazenados no dicionário interno de cada processo.

2.3.5. Métodos em $_{oo}$*Erlang*

Na compilação, cada método é acrescido de um parâmetro ObjectID que é uma referência ao objeto (ex: ref#666 na figura 2.13). Por exemplo, um método "`pessoa:get_name()`" tornar-se-ia "`get_name(ObjectID)`"e seria chamado invocado como "`pessoa:get_name(ObjectID)`". Um método pode possuir corpo ou não. Os métodos sem corpo são métodos abstratos. Os métodos de uma interface são obrigatoriamente abstratos e devem ser redefinidos pela classe que implementar a interface.

Em tempo de execução, ao invocar um método de um outro objeto, uma mensagem é enviada ao processo. Para exemplificar, são utilizadas as classes apresentadas no Programa 2.8:

<table>
<tr><th>Classe A</th><th>Classe B</th><th colspan="2">Classe Main</th></tr>
<tr><td><pre>-class(classea).
-export([get_value/0]).

attributes.
Value = 5.

methods.
get_value() ->
self::Value.</pre></td><td><pre>-class(classeb).
-export([print_A/1]).

methods.
print_A (ObjA) ->
io:format(
  "~p~n",
  [ObjA]
).</pre></td><td><pre>1
2
3
4
5
6
7
8</pre></td><td><pre>-class(main).
-export([main/0]).

class_methods.
main() ->
A = classea::new_(),
B = classeb::new_(),
B::print_A(A::get_value()).</pre></td></tr>
<tr><td colspan="4">Programa 2.8: Exemplo de execução de programa</td></tr>
</table>

A Classe A e a Classe B são instanciadas pela Classe Main. A Classe B executa um método da Classe A. Internamente ocorre:

Classe Main: linha 6 – Classe A é instanciada

- É criado um processo correspondente ao objeto A.
- Como a Classe A possui um atributo `Value` inicializado com o valor 5, este é armazenado no dicionário interno do processo.

Classe Main: linha 7 – Classe B é instanciada

- É criado um processo correspondente ao objeto B.

Classe Main: linha 8 – Objeto B invoca um método do objeto A.

- Objeto B envia uma mensagem para o objeto B indicando que quer utilizar o método `get_value/0`.
- Objeto A identifica o método e descobre que o mesmo consulta o valor de atributos.
- Objeto A consulta o valor do atributo em seu dicionário interno e o retorna para o `get_value/0`.
- Objeto A retorna o valor de `get_value/0` para o objeto B.
- Objeto B recebe o valor do retorno do objeto A.
- Objeto B imprime o valor recebido.

2.3.6. Atributos em $_{oo}$*Erlang*

Em Erlang, as variáveis são de atribuição única. Isso ajuda a evitar problema de concorrência. As variáveis em $_{oo}$*Erlang* também são de atribuição única. Os atributos não. Os atributos são sempre privados e só podem ser modificados pelos métodos do próprio objeto para fins de encapsulamento e um melhor controle sobre o estado do objeto. Apesar de $_{oo}$*Erlang* permitir a modificação dos valores dos atributos, para evitar problemas de concorrência, o desenvolvedor deve preferir a criação de objetos que usem e modifiquem os atributos apenas uma vez na sua criação (objetos imutáveis [27]).Por exemplo, o Programa 2.9 é compilado da seguinte forma:

```
-class(pessoa).
-export([mudar_idade/1,main/0]).

attributes.
Idade.

methods.
mudar_idade(MinhaIdade) ->
self::Idade = MinhaIdade.

class_methods.
main() ->
Joao = pessoa::new_(),
Joao::mudar_idade(10).
```

Programa 2.9 : Compilação dos atributos

- É criada uma referência para o objeto “Joao” (ex: ref#666).
- É invocado o método “mudar_idade()” com a referência “ref#666” e o valor, gerando internamente “`pessoa:mudar_idade(ref#666,10)`”
- Dentro do objeto “ref#666”, o parâmetro “MinhaIdade” é declarado como uma variável e o valor 10 é atribuído a ele.
- O valor da variável “MinhaIdade” é buscado e atribuído a “{ref#666,Idade}”.

Em tempo de execução, os atributos são armazenados no dicionário interno do objeto e acessados quando chamados pelos métodos.

2.3.7. Tratamento de erros em $_{oo}$*Erlang*

Os erros de compilação específicos da extensão $_{oo}$*Erlang* são tratados no módulo `ooel_error.erl` . Erlang possui um robusto sistema de tratamento de erros de execução baseado na estratégia “deixe quebrar”: se um processo deixar de funcionar, um outro processo deverá detectar isso e recriar o processo que caiu. Erlang provê essa abordagem, mas não obriga o desenvolvedor a

segui-la, ou seja, depende exclusivamente da maneira que um determinado sistema é programado. $_{oo}$*Erlang* segue a mesma abordagem, deixando a cargo do programador projetar seu sistema de modo a mitigar os riscos.

2.3.8. Palavras-chave e símbolos de $_{oo}$*Erlang*

Para o suporte a orientação a objetos, $_{oo}$*Erlang* introduziu novas palavras-chaves, símbolos e semânticas à linguagem Erlang descritos a seguir. O esquema geral de uma classe dentro de um arquivo `.cerl` é apresentado na figura 2.5.

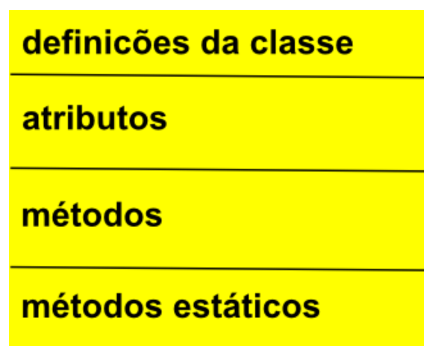


Figure 2.15 : Estrutura de um arquivo .cerl

A primeira parte é dedicada às definições da classe: nome, superclasse, implementações, construtor e interface. A segunda parte é utilizada para definição de atributos. A terceira parte para definição de métodos. A quarta parte é usada para definição de métodos estáticos. Cada uma dessas seções é iniciada por uma palavra-chave. As palavras-chaves utilizadas em $_{oo}$*Erlang* são:

- Palavra-chave **–class(*nome_da_classe*)**
  Utilizada no início da codificação de uma classe. Cada arquivo `.cerl` corresponde a apenas uma classe. Pode-se utilizar qualquer nome de módulo válido em Erlang.

- Palavra-chave **-interface(*nome_da_interface*)**
  Utilizada no início da codificação de uma interface. Cada arquivo `.cerl` corresponde a apenas uma interface. Pode-se utilizar qualquer nome de módulo válido em Erlang. Uma interface é uma abstração para criação em $_{oo}$*Erlang* de classes abstratas. Em uma interface, todos os métodos possuem apenas sua assinatura. O corpo dos métodos é vazio para ser obrigatoriamente redefinido nas classes que implementam a interface. O nome interface foi escolhido por ser mais expressivo para um programador acostumado

a orientação a objetos[1] usando Java. Sua compilação ocorre do mesmo jeito das classes, acrescentando a verificação de que seus métodos possuem apenas a assinatura e não o corpo.

- Palavra-chave **-extends(*nome_da_superclasse*)**
  Indica que a classe é uma subclasse. $_{oo}$*Erlang* aceita apenas herança simples.

- Palavra-chave **-implements(nome_da_interface1, nome_da_interface2...)**
  Indica que a classe implementará a interface indicada. $_{oo}$*Erlang* aceita que uma classe implemente múltiplas interfaces ao mesmo tempo. Sempre que uma classe implementar uma interface, ela é obrigada a redefinir todos os métodos da interface. O compilador mescla as informações da classe com a interface que ela implementa. Ao fazer essa mesclagem, é verificado se a classe implementa todos os métodos da interface. Se não o fizer, um erro de compilação é lançado.

- Palavra-chave **-constructor(nome_do_constructor1, nome do construtor2...)**
  Indica qual método será usado como construtor. $_{oo}$*Erlang* permite que vários métodos sejam declarados como construtores. Se o construtor não for definido, $_{oo}$*Erlang* assumirá o construtor padrão `new_/0`.

- Palavra-chave **methods.**
  Indica que a partir daquele ponto iniciará a seção de métodos. Os métodos obedecem às regras das funções em Erlang para escrita e nomeação.

- Palavra-chave **attributes.**
  Indica que a partir daquele ponto iniciará a seção de atributos. Os atributos são separados entre si por um ponto-e-vírgula (";"). Depois do último atributo, coloca-se um ponto ("."). Os atributos podem ser inicializados diretamente nessa seção ou então por meio do construtor. Os atributos obedecem as regras das variáveis Erlang.

- Palavra-chave **class_methods.**
  Indica que a partir daquele ponto iniciará a seção de métodos estáticos, ou seja, métodos da classe. Os métodos obedecem às regras das funções em Erlang para escrita e

[1]Foi observado que alguns desenvolvedores de nível intermediário em Java e iniciantes em Erlang consideram ser mais simples de entender o conceito de "classe abstrata pura" usando a palavra-chave *interface* ao invés de *abstract_class*

nomeação. Métodos estáticos não podem acessar diretamente os atributos do objeto, pois estes são acessíveis apenas pelos métodos do objeto. Eles são compilados como funções Erlang. Por exemplo, um método "Pessoa::main()" será compilado como "pessoa:main()".

- Palavra-chave **self::**

  A palavra-chave "`self::`" é utilizada para acessar atributos e métodos do objeto. Corresponde ao "`this`" do Java **[11]**. O nome `self` foi selecionado pelo Erlang já possuir uma função "self()", tornando assim mais fácil de ser lembrado pelos programadores Erlang. Seu uso é descrito na seção 2.3.6.

- Palavra-chave **super::**

  A palavra-chave "`super::`" é utilizada para acessar métodos da superclasse. Corresponde ao "`super`" do Java **[11][25]**.

2.4. Conclusão

Este capítulo mostra a sintaxe, semântica e implementação do *$_{oo}$Erlang*. Um pré-requisito de projeto foi a total compatibilidade com Erlang. Por isso foi necessário ampliar os analisadores léxico e sintático para introdução de novas palavras-chave. Ainda assim, a sintaxe manteve-se simples, expressiva e facilmente reconhecível por um programador acostumado com Erlang. O Capítulo 3 mostra uma análise da expressividade de *$_{oo}$Erlang* e exemplos de programas. O Capítulo 4 apresenta uma avaliação da expressividade de *$_{oo}$Erlang* por meio de exemplos mais elaborados.

3. EXPRESSIVIDADE

To me programming is more than an
important practical art.
It is also a gigantic undertaking in
the foundations of knowledge.

Grace Hooper citada em
Management and the Computer of
the Future

Este capítulo mostra a expressividade de $_{oo}$*Erlang*, comparando códigos desenvolvidos em $_{oo}$*Erlang* com códigos em Java e Erlang. Depois apresenta exemplos de programas para destacar características importantes do suporte a orientação a objetos implementadas por $_{oo}$*Erlang*.

3.1. Expressividade

A expressividade de uma linguagem caracteriza quão naturalmente uma ideia pode ser concretizada em forma de um programa. Segundo Sebesta **[23]**, uma linguagem expressiva especifica computações de forma conveniente, em vez de deselegante de modo a aumentar a facilidade de escrita. Apesar de Joe Armstrong e outros considerarem Erlang orientada a objetos **[83]**, a linguagem carece de uma sintaxe específica que facilite a programação orientada a objetos. $_{oo}$*Erlang* veio suprir esta carência. Para fins de comparação, são escritos três programas orientados

a objeto usando Java, Erlang e $_{oo}$*Erlang*. Os exemplos de Padrões de Projetos foram retirados de [26] e os demais exemplos de [19].

3.2. Exemplo 1 – Singleton

Singleton é um Padrão de Projeto que garante que uma classe só tenha uma única instância e fornece um ponto global de acesso a ela [38].O exemplo do uso do padrão Singleton é apresentado na Figura 3.1:

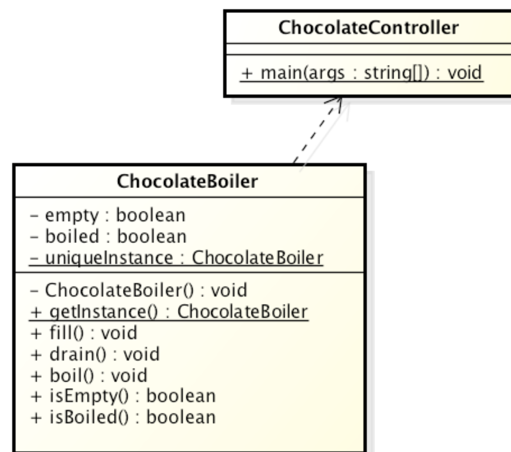


Figura 3.1: Exemplo do *Design Pattern Singleton*

Na Figura 3.1, uma caldeira de chocolate é monitorada por um controlador. Só uma caldeira pode ser instanciada. A ideia do padrão é forçar a criação de apenas uma única instancia do objeto. Um trecho da implementação em Java da classe ChocolateBoiler é mostrado no Programa 3.1

```
publicclassChocolateBoiler{
        private boolean empty;
        private boolean boiled;
        private static ChocolateBoiler uniqueInstance;

      private ChocolateBoiler(){

                 empty =true;
                 boiled =false;
        }

        public static ChocolateBoiler getInstance(){
                 if(uniqueInstance ==null){
                          System.out.println("Creating unique instance of Chocolate
Boiler");
                          uniqueInstance =new ChocolateBoiler();
                 }
                 System.out.println("Returning instance of Chocolate Boiler");
```

```
                    return uniqueInstance;
            }
...
```

Programa 3.1: Classe ChocolateBoiler em Java

Observa-se que foi fácil concretizar a ideia do problema em Java. O método `getInstance()` garante a instanciação de um único objeto. O mesmo exemplo agora é codificado em Erlang e o trecho que garante a única instância é apresentado no Programa 3.2.

```
-module(chocolateBoiler).
-compile(export_all).

new() ->
    Dict=orddict:new(),
    Dict2=orddict:store(empty, true, Dict),
    Dict3=orddict:store(boiled, false, Dict2),
    Dict4=orddict:store(uniqueInstance, [], Dict3),
    PidOfObject=spawn(chocolateBoiler, object_running, [Dict4]),
    ListOfProcesses=erlang:registered(),
    Return=lists:member(new_object, ListOfProcesses),
    if
        (Return== false) ->
            erlang:register(new_object, PidOfObject),
            new_object;
        true ->
            PidOfObject
    end.

object_running(Dict) ->
    receive
        {update, Key, NewValue} ->
            case orddict:find(Key, Dict) of
                {ok, KeyValue} ->
                 Dict2=orddict:store(KeyValue, NewValue, Dict);
                _->
                 Dict2=Dict
            end,
            object_running(Dict2)
    end.


get_instance() ->
    ListOfProcesses=erlang:registered(),
    Return=lists:member(unique_instance, ListOfProcesses),
    if
                (Return== false) ->
                io:format("Creating unique instance of Chocolate Boiler ~n"),
         NewPid= new(),
```

```
                                update_attribute(new_object, uniqueInstance, NewPid),
                                erlang:register(unique_instance, NewPid);
                      true ->
                                io:format("")
    end,
    io:format("Returning instance of Chocolate Boiler ~n"),
    unique_instance.

update_attribute(Id, Key, NewValue) ->
    Id! {update, Key, NewValue}.
```

Programa 3.2: Classe ChocolateBoiler em Erlang

Observa-se que foi necessário o uso de diversos artifícios de programação Erlang. Inclusive o uso de dicionários e o controle explícito de processos. Apesar de funcionar corretamente, o código não é nada intuitivo. Ele não exprime bem a ideia do padrão. O mesmo problema agora é codificado em $_{oo}$*Erlang* e o trecho que garante a instanciação única é apresentado no Programa 3.3

```
-class(chocolateBoiler).
-export([new/0, get_instance/0, fill/0, drain/0, boil/0]).
-export([is_empty/0, is_boiled/0]).
-constructor([new/0]).

attributes.

Empty;
Boiled;
UniqueInstance.

methods.

new() ->
      self::Empty = true,
      self::Boiled = false.

get_instance() ->
      ListOfProcesses = erlang:registered(),
      % unique_instance é o atom que identificará a instancia unica do Singleton
      Return = lists:member(unique_instance, ListOfProcesses),
      if
            (Return == false) ->
                  io:format("Creating unique instance of Chocolate Boiler ~n"),
                  self::UniqueInstance = chocolateBoiler::new(),
                  erlang:register(unique_instance, ObjectID);
            true ->
                  io:format("")
      end,
      io:format("Returning instance of Chocolate Boiler ~n"),
      Unique = {chocolateBoiler, whereis(unique_instance)},
      Unique::UniqueInstance.
…
```

Programa 3.3 : Classe ChocolateBoiler

O Figura 3.1 foi codificada de maneira simples e direta usando $_{oo}$*Erlang*. A única instância é garantida usando o a capacidade do Erlang em registrar processos. Como cada objeto é um processo, basta verificar se ele já existe.

3.3. Exemplo 2 – Bridge

Bridge é um Padrão de Projeto cujo objetivo é desacoplar uma abstração de sua implementação para que ambos possam variar independentemente [38]. O exemplo do uso do padrão Bridge é apresentado na Figura 3.2:

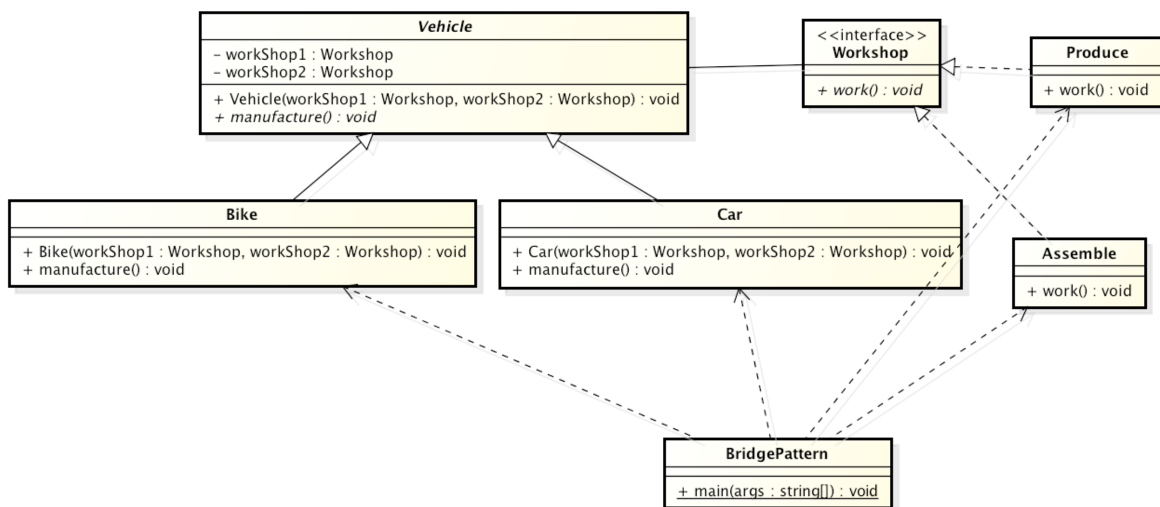


Figura 3.2: Exemplo do *Design Pattern Bridge*

Na Figura 3.2 são desenvolvidos dois tipos de veículos (“Bike” e “Car”) e estes podem ser construídos de duas formas distintas (Produce e Assemble). Criar subclasses BikeProduce, BikeAssemble, CarProduce e CarAssemble ocasionaria duplicação de código. Em vez disso, utilizou-se a interface Workshop para servir de “ponte” e unir os veículos e seus métodos de produção. Um trecho da implementação em Java da classe Bike é mostrada no Programa 3.4

```
public class Bike extends Vehicle {
        public Bike(Workshop workShop1, Workshop workShop2){
        super(workShop1, workShop2);
        }

        public void manufacture(){
            System.out.print("Bike ");
            workShop1.work();
        workShop2.work();
        }
}
```

Programa 3.4 : Classe Bike em Java

O programa foi escrito em Java sem maiores problemas. O código-fonte Erlang para o mesmo problema é apresentado no Programa 3.5

```
-module(bike).
-export([new/2, manufacture/0]).

new(WorkShop1, WorkShop2) ->
        Dict=orddict:new(),
        Dict2=orddict:store(workshop1, WorkShop1, Dict),
        Dict3=orddict:store(workshop2, WorkShop2, Dict2),
        PidOfObject=spawn(bike, object_running, [Dict3]),
        PidOfObject.

manufacture() ->
        io:format("Bike ").

object_running(Dict) ->
        receive
       {Sender, manufacture} ->
                        manufacture(),
          case orddict:find(workshop1, Dict) of
                {ok, Value} ->
                 Value! {self(), work};
                _->
                 io:format("")
          end,
                        (workshop2, Dict) of
                {ok, Value2} ->
                 Value2! {self(), work};
                _->
                 io:format("")
          end,
          object_running(Dict)
    end.
```

Programa 3.5 : Classe Bike em Erlang

Foi necessário utilizar dicionários e controlar explicitamente a criação de processos. Há uma preocupação maior com a utilização correta da sintaxe e artifícios de programação do que com resolução do problema em si. O mesmo problema agora é codificado em $_{oo}$*Erlang* e apresentado no Programa 3.6

```
-class(bike).
-extends(vehicle).
-export([new/2, manufacture/0]).
-constructor([new/2]).

methods.

new(WorkShop1, WorkShop2) ->
         self::WorkShop1 = WorkShop1,
         self::WorkShop2 = WorkShop2.

manufacture() ->
         io:format("Bike "),
         Work1 = self::WorkShop1,
         Work2 = self::WorkShop2,
         Work1::work(),
         Work2::work().
```

Programa 3.6 : Classe Bike em Erlang

Ao nosso ver, a codificação é limpa, direta e intuitiva.

3.4. Exemplo 3 – Strategy

"Strategy" tem o objetivo de definir uma família de algoritmos, encapsulando-os e fazendo-os intercambiáveis [38].O exemplo do uso do padrão Strategy é apresentado na Figura 3.3:

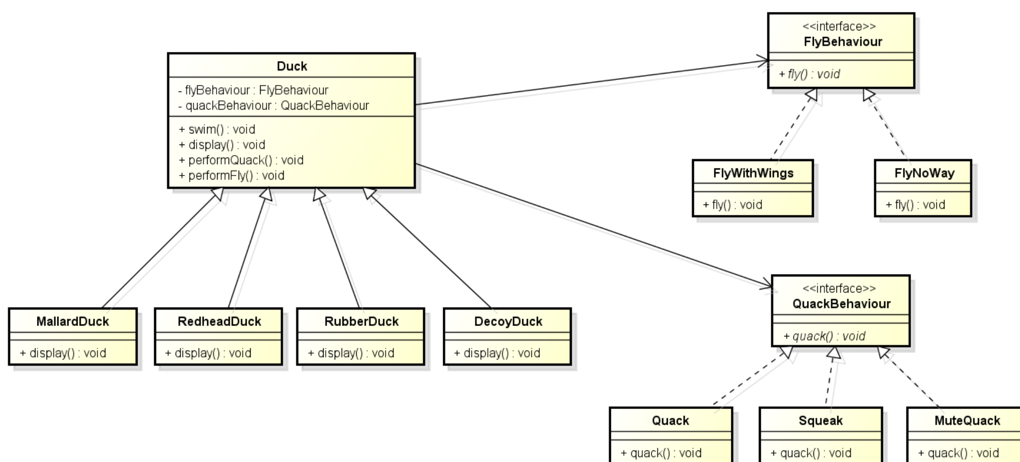


Figura 3.3: Exemplo do *Design Strategy*

Nesse exemplo, o objetivo é construir um simulador de patos. Os patos podem grasnar (Quack) e voar (Fly). Existem maneiras diversas de grasnar e voar. Existem vários tipos de patos que grasnam e voam de maneiras diferentes. Utilizou-se o padrão Strategy para combinar as classes de modo a

diminuir o acoplamento e evitar duplicação de código. Um trecho da implementação em Java da classe "MallardDuck" é mostrado no Programa 3.4

```
-public class MallardDuck extends Duck {
        public MallardDuck(){
                quackBehaviour =new Quack();
                flyBehaviour =new FlyWithWings();
        }

        public void display(){
                System.out.println("I'm a real Mallard Duck!");
        }
}
```

Programa 3.7 : Classe MallardDuck em Java

O código em Java mostra claramente a solução do problema. O código em Erlang é apresentado no Programa 3.8

```
-module(mallardDuck).
-export([new/0, display/0]).

new() ->
        Dict=orddict:new(),
        Dict2=orddict:store(quackBehaviour, quack:new(), Dict),
        Dict3=orddict:store(flyBehaviour, fly:new(), Dict2),
        IdOfObject=spawn(mallardDuck, object_running, [Dict3]),
        IdOfObject.
display() ->
        io:format("I'm a real Mallard Duck!~n").

object_running(Dict) ->
        receive
                {Sender, update} ->
                        io:format("")
...

        end.
```

Programa 3.8 : Classe MallardDuck em Erlang

Ficou complicado de reconhecer a implementação do "MallardDuck". Novamente, houve a necessidade de usar artifícios de programação Erlang e dicionários para programar uma classe simples. O código em $_{oo}$*Erlang* é apresentado no Programa 3.9

x bytes

P1 P2

x bytes

```
-class(mallardDuck).
-extends(duck).
-export([new/0, display/0]).
-constructor([new/0]).

methods.

new() ->
        self::QuackBehaviour = quack::new_(),
        self::FlyBehaviour = flyWithWings::new_().

display() ->
        io:format("I'm a real Mallard Duck!~n").
```

Programa 3.9 : Classe MallardDuck em ooErlang

O código em $_{oo}$*Erlang* ficou simples de entender e bastante limpo.

3.5. Exemplo 5 - PingPing

Esse exemplo mostra a implementação do *benchmark* IMB PingPing [19]. O foco desse *benchmark* é medir a eficiência no tratamento de bloqueios que acontecem quando um processo recebe uma mensagem no momento que ele envia outra. O teste também registra o tempo de latência do processo e a vazão do sistema (processamento de cada mensagem). A Figura 3.8 esquematiza a comunicação nesse teste:

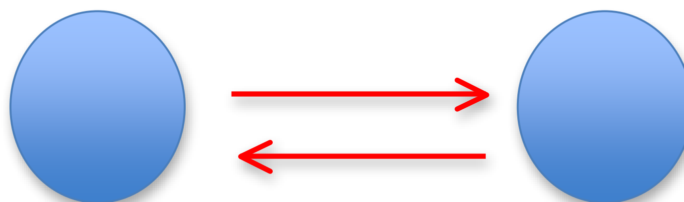


Figura 3.4 : PingPing *Benchmark*

Um processo P1 envia uma mensagem de *x* bytes de tamanho para o processo P2. Assincronamente, P2 envia a mesma mensagem para P1. São coletados o tempo de latência, o tempo de inicialização do processo e o tempo total de envios e recebimentos. A modelagem deste problema é apresentada na Figura 3.5. A vantagem de usar orientação a objetos para sua resolução é facilitar o entendimento do problema e sua modelagem.

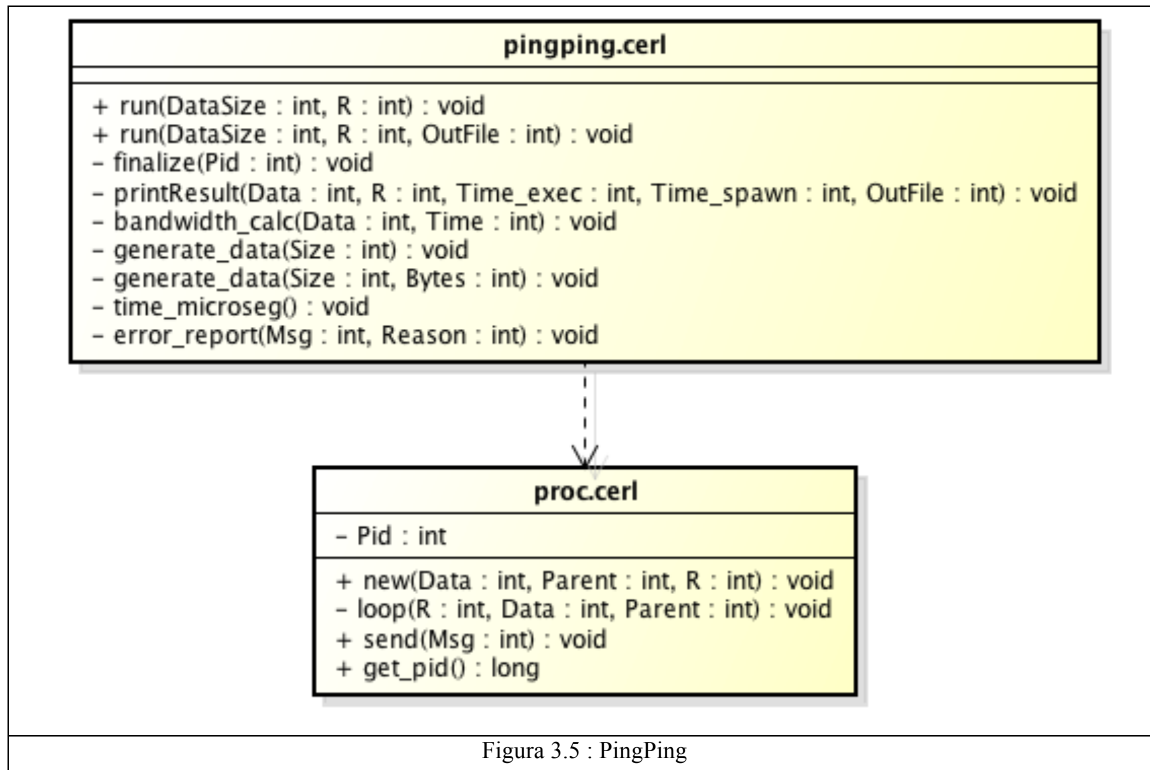


Figura 3.5 : PingPing

A classe `pingping` gerencia o comportamento de troca de mensagens entre os processos `proc`. Ela inicializa os processos, calcula o tempo de inicialização, o tempo de comunicação e salva os dados em um arquivo de saída. A solução desse problema em Java é mostrada no Programa 3.10.

```
public class PingPing extends Thread{

        private final int QTD_PROC_TOTAL = 2;
        // a medida que os processos finalizarem
        // qtdTerminados é incrementados
        private int qtdProcTerminados = 0;

        private long timeSpawn;
        private long timeExec;

        public long timeStart;
        public long timeEnd;

        private int tamDados;
        private int qtdRept;
        private String outLocation;

        public PingPing(int tamDados, int qtdMsg, String outLocation) {
                this.tamDados = tamDados;
                this.qtdRept = qtdMsg;
                this.outLocation = outLocation;
        }
```

```
	public void run() {
		byte[] dado = generateData(tamDados);

		// cria os processos

		timeStart = timeMicroSeg();
		ProcPing p1 = new ProcPing("1", dado, this, qtdRept);
		ProcPing p2 = new ProcPing("2", dado, this, qtdRept);
		timeEnd = timeMicroSeg();

		// armazena tempo de criação
		timeSpawn = timeEnd - timeStart;

		// inicia execução dos processos
		timeStart = timeMicroSeg();
		p1.setPeer(p2); // seta o par
		p1.start();
		p2.setPeer(p1); // seta o par
		p2.start();

		dormirAteTerminar();

		// após todos terminarem, executa a prox linha
		timeEnd = timeMicroSeg(); // captura tempo atual

		//FINALIZA TESTE
		timeExec = timeEnd - timeStart; // armazena tempo de execução

		// escreve a saída
		Salvar.writeResultPeer(outLocation, tamDados, qtdRept, timeExec,
timeSpawn);
	}

	private synchronized void dormirAteTerminar() {
		while(qtdProcTerminados!= QTD_PROC_TOTAL){
			try {
				wait();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}

	public synchronized void acordar() {
		qtdProcTerminados++;
		notifyAll();
	}

	private byte[] generateData(int tamDados) {
```

```
		byte[] dado = new byte[tamDados];
		return dado;
	}

	private long timeMicroSeg() {
		return System.nanoTime()/1000;
	}

}

public class ProcPing extends Thread {
	public byte[] mailbox;
	private ProcPing peer;
	private PingPing parent;
	private int qtdMsg;
	private byte[] dado;

	public ProcPing(String name, byte[] dado, PingPing parent, int qtdMsg) {
		this.setName(name);
		this.dado = dado;
		this.parent = parent;
		this.qtdMsg = qtdMsg;
	}

	public void setPeer(ProcPing peer) {
		this.peer = peer;
	}

	private synchronized void send(byte[] msg) {
			peer.mailbox = msg.clone();
	}

	private void recv() {
		// verifica mailbox até ter mensagem
		while (true) {
			synchronized (this) {
				if (mailbox != null) {
					mailbox = null;
					break;
				}
			}		}
	}


	public void run() {
		for (int i = 1; i <= qtdMsg; i++) {
			send(dado);
			if(i!=qtdMsg){
				recv();
			}
```

```
                    }
                    parent.acordar();
          }
}
```

Programa 3.10 : PingPing em Java

O código em Java ficou relativamente longo para um problema simples porque ele envolve troca de mensagens entre *threads* e isso não é o padrão de concorrência da linguagem. O problema codificado em Erlang é apresentado no Programa 3.11

```
-module(pingping).
-export([run/2, run/3]).
-include("conf.hrl").

run(DataSize, R) ->
   OutFileLocation = "../../docs/erlang/out_erl_pingping.txt",

   case file:open(OutFileLocation, [append]) of
         {error, Why} ->
             ?ERR_REPORT("Falha ao criar arquivo!", Why);

         {ok, OutFile} ->
             run(DataSize, R, OutFile)
end.

run(DataSize, R, OutFile) ->
    Data = generate_data(DataSize),
    Self = self(),
    SpawnStart = time_microseg(),
    P1 = spawn(fun() -> pingping(Data, Self, R) end),
    P2 = spawn(fun() -> pingping(Data, Self, R) end),
    SpawnEnd = time_microseg(),
    TimeStart = time_microseg(),
P1 ! {init, self(), P2},
P2 ! {init, self(), P1},
    finalize(P1),
    finalize(P2),
    TimeEnd = time_microseg(),
    TotalTime = TimeEnd - TimeStart,
    SpawnTime = SpawnEnd - SpawnStart,
printResult(Data, R, TotalTime, SpawnTime, OutFile).

pingping(_,Parent, 0) ->
Parent ! {finish, self()};

pingping(Data, Parent, R) ->
    receive
```

```
        {init, Parent, Peer} ->
        Peer ! {self(), Data},
            pingping(Data, Parent, R-1);

        {Peer, Data} ->
        Peer ! {self(), Data},
            pingping(Data, Parent, R-1)
end.

finalize(P1) ->
    receive
        {finish, P1} ->
            ok
end.

%%-------------------------------------------------------------------------------
%% printResult( Data, R, Time, OutFile )
%% imprime o resultado
%%
%% Data (binary) = os dados usados no teste
%% R             = quantidade de repetições
%% Time (µs)  = o tempo total do teste
%% OutFile    = o arquivo de saída

printResult(Data, R, Time_exec, Time_spawn, OutFile) ->
FormatH = "~-9s\t ~-13s\t ~-17s\t ~-11s\t ~-10s~n",
    Header = ["#bytes", "#repetitions", "exec_time[µsec]", "MBytes/sec",
"spawn_time"],
    io:format(OutFile, FormatH, Header),
    MBps = bandwidth_calc(Data, Time_exec),
    FormatD = "~-9w\t ~-13w\t ~-17.2f\t ~-11.6f\t ~-15.2f~n",
io:format(OutFile, FormatD, [size(Data), R, Time_exec, Time_spawn, MBps]).

%%-------------------------------------------------------------------------------
%% bandwidth_calc(Data, Time)
%% calcula o tráfego na rede baseado no tamanho dos dados e quanto tempo levou
%%
%% Data (binary) = os dados usados no teste
%% Time (µs)  = o tempo total do teste

bandwidth_calc(Data, Time) ->
    Megabytes = (size(Data) / math:pow(2, 20)),
    Seconds = (Time * 1.0e-6),
    Megabytes / Seconds.

%%-------------------------------------------------------------------------------
%% generate_data(Size)
%%   gera um dado de tamanho Size bytes
%%
%%   Size = integer
```

```
generate_data(Size) -> generate_data(Size, []).

generate_data(0, Bytes) ->
    list_to_binary(Bytes);
generate_data(Size, Bytes) ->
generate_data(Size - 1, [1 | Bytes]).

%%-------------------------------------------------------------------------------
%% time_microseg()
%% captura o tempo atual em microssegundos

time_microseg() ->
    {MS, S, US} = now(),
(MS * 1.0e+12) + (S * 1.0e+6) + US.
```

Programa 3.11 : PingPing em Erlang

O código ficou com menos linhas em Erlang, porém não foi possível modularizar: o pingping e o proc ficaram em apenas um programa. Isso atrapalha uma futura manutenção. O código em $_{oo}$*Erlang* é apresentado no Programa 3.12

```
-class(pingping).
-export([run/2, run/3]).

class_methods.

run(DataSize, R) ->
   OutFileLocation = "out_cerl_pingping.txt",

   case file:open(OutFileLocation, [append]) of
      {error, Why} ->
          error_report("Falha ao criar arquivo!", Why);

      {ok, OutFile} ->
          run(DataSize, R, OutFile)
    end.

run(DataSize, R, OutFile) ->
    Data = generate_data(DataSize),
    Self = self(),
    SpawnStart = time_microseg(),
    P1 = procping::new(Data, Self, R),
      P2 = procping::new(Data, Self, R),
    SpawnEnd = time_microseg(),
    TimeStart = time_microseg(),
    P1::send({init, self(), P2::get_pid() }),
    P2::send({init, self(), P1::get_pid()}),
      finalize(P1::get_pid()),
      finalize(P2::get_pid()),
    TimeEnd = time_microseg(),
    TotalTime = TimeEnd - TimeStart,
```

```
    SpawnTime = SpawnEnd - SpawnStart,
    printResult(Data, R, TotalTime, SpawnTime, OutFile).

%%------------------------------------------------------------------------------
%% finalize( Pid )
%% espera termino da execucao do processo
%%
%% Pid           = o pid do processo
finalize(Pid) ->
      receive
            {finish, Pid} ->
                  ok
      end.

%%------------------------------------------------------------------------------
%% printResult( Data, R, Time, OutFile )
%% imprime o resultado
%%
%% Data (binary) = os dados usados no teste
%% R             = quantidade de repetições
%% Time (µs)  = o tempo total do teste
%% OutFile    = o arquivo de saída

printResult(Data, R, Time_exec, Time_spawn, OutFile) ->
    FormatH = "~-9s\t ~-13s\t ~-17s\t ~-11s\t ~-10s~n",
    Header = ["#bytes", "#repetitions", "exec_time[µsec]", "MBytes/sec",
"spawn_time"],
    io:format(OutFile, FormatH, Header),
    MBps = bandwidth_calc(Data, Time_exec),
    FormatD = "~-9w\t ~-13w\t ~-17.2f\t ~-11.6f\t ~-15.2f~n",
    io:format(OutFile, FormatD, [size(Data), R, Time_exec, Time_spawn, MBps]).

%%------------------------------------------------------------------------------
%% bandwidth_calc(Data, Time)
%% calcula o tráfego na rede baseado no tamanho dos dados e quanto tempo levou
%%
%% Data (binary) = os dados usados no teste
%% Time (µs)  = o tempo total do teste

bandwidth_calc(Data, Time) ->
    Megabytes = (size(Data) / math:pow(2, 20)),
    Seconds = (Time * 1.0e-6),
    Megabytes / Seconds.

%%------------------------------------------------------------------------------
%% generate_data(Size)
%%   gera um dado de tamanho Size bytes
%%
%%   Size = integer

generate_data(Size) -> generate_data(Size, []).

generate_data(0, Bytes) ->
    list_to_binary(Bytes);
generate_data(Size, Bytes) ->
    generate_data(Size - 1, [1 | Bytes]).

%%------------------------------------------------------------------------------
%% time_microseg()
%% captura o tempo atual em microsegundos
time_microseg() ->
    {MS, S, US} = now(),
    (MS * 1.0e+12) + (S * 1.0e+6) + US.
```

```
%%-----------------------------------------------------------------------------
%% error_report(Msg, Reason)
%% formata a msg de erro
error_report(Msg, Reason) ->
      io:format("%%%%%%%%%%%%%% ERRO %%%%%%%%%%%%%%\n" ++
                              "Msg: ~w\n" ++
                              "Reason: ~w\n\n", [Msg, Reason]).

-class(procping).
-export([new/3, send/1]).
-constructor([new/3]).

attributes.

Pid.

methods.

new(Data, Parent, R) ->
      self::Pid = spawn( fun() -> loop(R, Data, Parent) end ).

send(Msg) ->
      self::Pid ! Msg.

get_pid() ->
      self::Pid.

class_methods.

loop(0, _, Parent) ->
    Parent ! {finish, self()};

loop(R, Data, Parent) ->
      receive
            {init, Parent, Peer} ->
            Peer ! {self(), Data},
            loop(R-1, Data, Parent);

            {Peer, Data} ->
            Peer ! {self(), Data},
            loop(R-1, Data, Parent)
   end.
```

Programa 3.12 : PingPing em $_{oo}$*Erlang*

O código em $_{oo}$*Erlang* ficou um pouco mais longo que o de Erlang porém foi possível a modularização. Foram criadas as classes PingPing e ProcPing. Nesse exemplo, o usuário passa os parâmetros para o método `run(DataSize,R)`, mostrado Programa 3.13:

```
run(DataSize, R) ->
OutFileLocation = "out_cerl_pingping.txt",
case file:open(OutFileLocation, [append]) of
{error, Why} ->
error_report("Falha ao criar arquivo!", Why);
{ok, OutFile} ->
run(DataSize, R, OutFile)
end.
```

Programa 3.13: Método run(DataSize,R) da classe pingping.cerl

Esse método recebe o tamanho dos dados a serem enviados entre os processos e a quantidade de repetições de envio de mensagens. Em seguida, é chamado o método `run(DataSize,R,OutFile)`, mostrado no Programa 3.14:

```
run(DataSize, R, OutFile) ->
Data = generate_data(DataSize),
Self = self(),
    SpawnStart = time_microseg(),
    P1 = procping::new(Data, Self, R),
    P2 = procping::new(Data, Self, R),
    SpawnEnd = time_microseg(),
    TimeStart = time_microseg(),
    P1::send({init, self(), P2::get_pid() }),
    P2::send({init, self(), P1::get_pid()}),
    finalize(P1::get_pid()),
    finalize(P2::get_pid()),
    TimeEnd = time_microseg(),
    TotalTime = TimeEnd - TimeStart,
    SpawnTime = SpawnEnd - SpawnStart,
    printResult(Data, R, TotalTime, SpawnTime, OutFile).
```

Programa 3.14: Método run(DataSize,R,OutFile) da classe pingping

Inicialmente, esse método gera o tamanho do pacote a ser enviado e recebido entre os processos. Em seguida, cria dois processos, "`P1`" e "`P2`", calculando o tempo de criação desses processos. A seguir, cada um dos processos envia uma mensagem um para o outro, iniciando o teste. Assim que um processo recebe a mensagem de início, responde enviando novamente a mensagem ao remetente. Como ambos mandam a mensagem inicial, ambos recebem e respondem, de acordo com a quantidade de repetições passada no parâmetro "`R`" do método `run(DataSize,R,OutFile)`.Quando `R` chegar a 0 (zero) é utilizado método `finalize(P1::Pid)`, mostrado no Programa 3.15, para que os processos terminem, após todos os envios de mensagem.

```
finalize(Pid) ->
receive
{finish, Pid} ->ok
end.
```

Programa 3.15: Método finalize(Pid) da classe pingping.cerl

A classe `procping.cerl` implementa a abstração de um processo ping. Seu código-fonte é mostrado no Programa 3.16:

```
-class(procping).
-export([new/3, send/1]).
-constructor([new/3]).

attributes.

Pid.
```

X

X

```
methods.

new(Data, Parent, R) ->
self::Pid = spawn( fun() -> loop(R, Data, Parent) end ).


send(Msg) ->
      self::Pid ! Msg.

class_methods.

loop(0, _, Parent) ->
    Parent ! {finish, self()};
loop(R, Data, Parent) ->
    receive
        {init, Parent, Peer} ->
            Peer ! {self(), Data},
            loop(R-1, Data, Parent);
        {Peer, Data} ->
            Peer ! {self(), Data},
            loop(R-1, Data, Parent)
    end.
```

Programa 3.16: Classe procping

Essa classe encapsula todos os métodos e atributos utilizados por um processo `ping`. Mesmo sendo uma classe, nota-se o estilo Erlang de programar. O código tornou-se mais simples, expressivo e de fácil manutenção.

3.6. Exemplo 6 - PingPong

Este exemplo mostra uso do $_{oo}$*Erlang* para implementação do *benchmark* IMB PingPong **[19]**. Os códigos em Java e Erlang são apresentados no sítio deste projeto **[74]**. PingPong funciona de maneira similar ao *benchmark* PingPing e seu esquema de comunicação é apresentado na Figura 3.6.

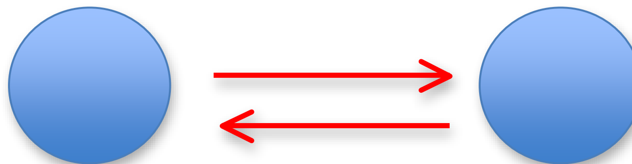


Figura 3.6 : PingPong *Benchmark*

A diferença entre o PingPing e o PingPong é a natureza síncrona deste último: um processo P1 envia uma mensagem de x bytes de tamanho para um processo P2 e entra em estado de espera. O processo P2 recebe a mensagem de P1 e responde com outra mensagem de tamanho x bytes. O problema foi modelado usando duas classes apresentadas na Figura 3.7:

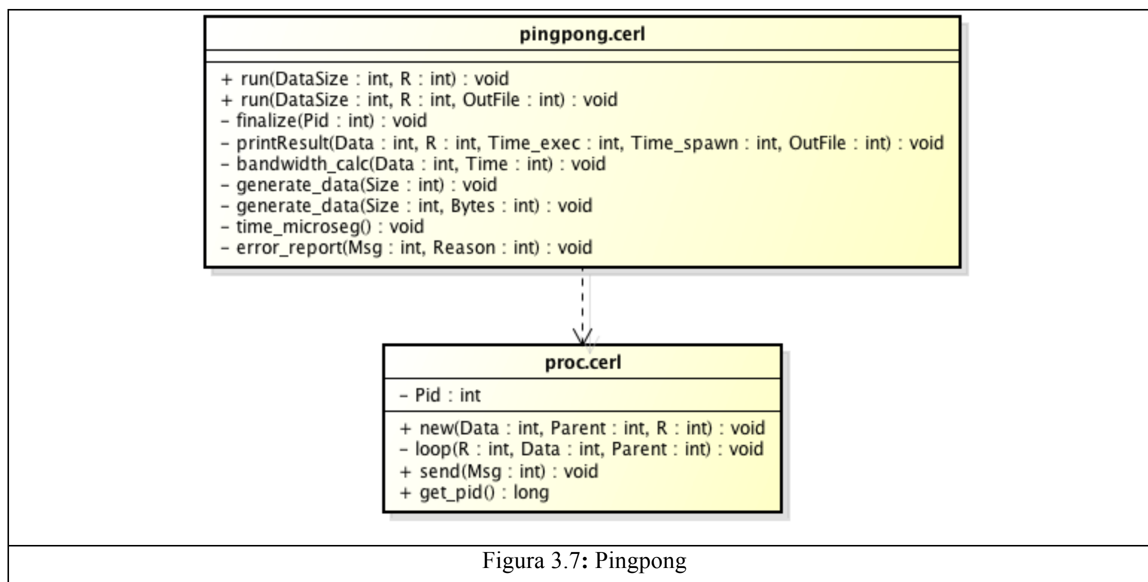


Figura 3.7: Pingpong

A classe `proc.cerl` é reutilizada do pingping. A classe `pingpong.cerl` gerencia o comportamento de troca de mensagens entre os processos `procping.cerl`. Ela inicializa os processos, calcula o tempo de inicialização, o tempo de comunicação e salva os dados em um arquivo de saída. O usuário passa os parâmetros para o método `run(DataSize,R)`.O código-fonte do método é apresentado no Programa 3.17:

```
run(DataSize, R, OutFile) ->
    Data = generate_data(DataSize),
    Self = self(),
    SpawnStart = time_microseg(),
    P1 = procpong::new(Data, Self, R),
    P2 = procpong::new(Data, Self, R),
    SpawnEnd = time_microseg(),
    TimeStart = time_microseg(),
    P1::send({init, self(), P2::get_pid()}),
    finalize(P1::get_pid()),
    finalize(P2::get_pid()),
    TimeEnd = time_microseg(),
    TotalTime = TimeEnd - TimeStart,
    SpawnTime = SpawnEnd - SpawnStart,
    printResult(Data, R, TotalTime, SpawnTime, OutFile).
```

Programa 3.17: Método run(DataSize,R,OutFile) da classe pingpong.cerl

A diferença entre os métodos `run/3` dos testes PingPing e PingPong é que, no `pingpong.cerl`, o início de troca de mensagens entre os processos é feito apenas do processo "`P1`" enviando a mensagem "`init`" para o processo "`P2`". E assim que "`P2`" recebe a mensagem, ele responde ao "`P1`", iniciando-se a troca de mensagens entre os processos.

3.7. Exemplo 7 – SendRec

Este exemplo mostra uso do $_{oo}$*Erlang* para implementação do *benchmark* IMB SendRec **[19]**. Os códigos em Java e Erlang são apresentados no sítio deste projeto **[74]**. O *benchmark* Send-Receive (SendRec) exercita a transferência paralela de dados. A atividade em um processo está ocorrendo simultaneamente em outros processos. Então, é medida a eficiência no envio de mensagens sob uma carga global. Os processos são criados e dispostos em forma de "anel" como mostra a Figura 3.8. Cada processo envia uma mensagem de x bytes para o processo à sua direita. A versão implementada foi a *Threadring*, onde uma mensagem é enviada e percorre N vezes o anel de processos. Ao receber a mensagem, o processo a reenvia para o processo à sua direita e assim por diante.

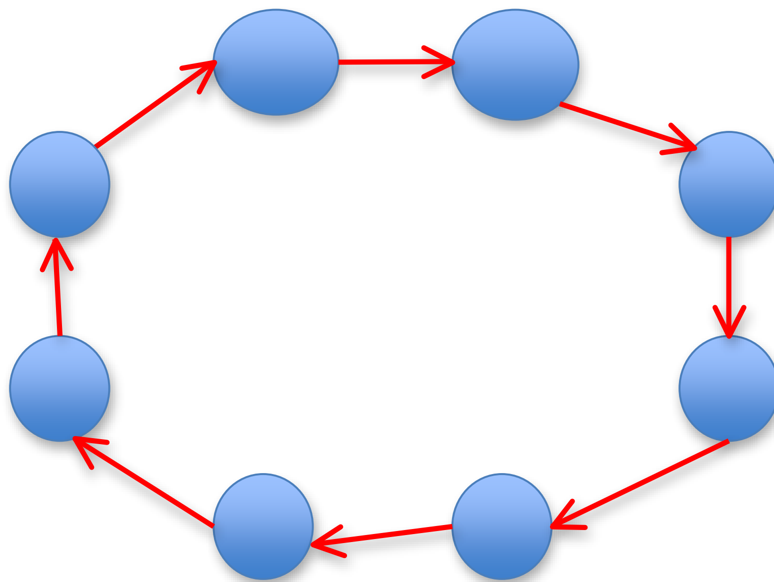


Figura 3.8: SendRec *Benchmark*

O problema foi modelado utilizando as classes apresentadas na Figura 3.9:

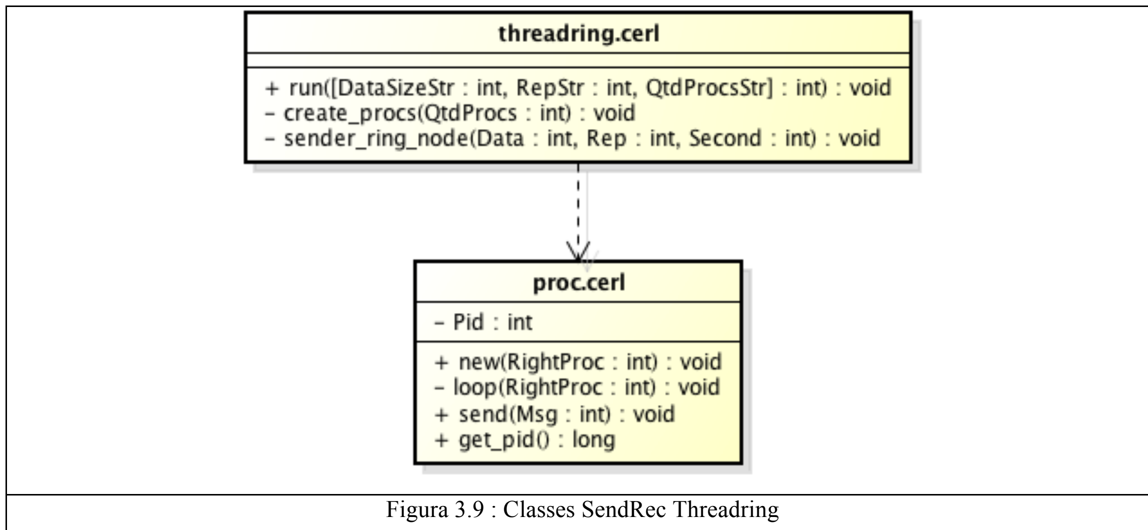


Figura 3.9 : Classes SendRec Threadring

A classe `proc.cerl` modela o processo que será usado como "elo" do anel. A classe `threadring.cerl` é a responsável por criar os processos e iniciar o ciclo de envio de mensagens. Isto é feito por meio do método `run/1` mostrado no programa 3.18:

```
run([DataSizeStr, RepStr, QtdProcsStr]) ->
    DataSize = list_to_integer(DataSizeStr),
    Rep = list_to_integer(RepStr),
    QtdProcs = list_to_integer(QtdProcsStr),

    Data = generate_data(DataSize),

    SpawnStart = time_microseg(),
    Second = create_procs(QtdProcs),
    SpawnEnd = time_microseg(),

    ExecStart = time_microseg(),
    sender_ring_node(Data, Rep, Second),
    ExecEnd = time_microseg(),

    TotalTime = ExecEnd - ExecStart,
    SpawnTime = SpawnEnd - SpawnStart,

    OutFile = createOrOpen_file("./out_erl_threadring.txt"),
    printResult(Data, Rep, QtdProcs, TotalTime, SpawnTime, OutFile),
    erlang:halt().
```

Programa 3.18: Método run/1 da classe threadring.cerl

Os processos são criados e o tempo é calculado durante essa criação. O método que cria os processos é o `create_procs/1` apresentado no Programa 3.19:

```
create_procs(QtdProcs) ->
lists:foldl(
fun(_Id, RightPeer) -> ObjProc = proc::new(RightPeer),ObjProc::get_pid() end,
      self(),
      lists:seq(QtdProcs, 2, -1) ).
```

Programa 3.19: Método create_procs(QtdProcs) da classe threadring.cerl

Esse método cria a quantidade de processos recebida no parâmetro `“QtdProcs”` e cada processo possui o Pid de seu processo criador. Todo processo ao ser criado fica esperando receber uma mensagem para enviar essa mensagem ao seu processo criador.

Após o anel de processos ser criado, é utilizado o método `sender_ring_node/3` que inicializa o envio de mensagens entre os processos do anel. O código-fonte do método é apresentado no Programa 3.20 :

```
sender_ring_node(_,0,_) -> ok;
sender_ring_node(Data, Rep, Second) ->
    Second ! Data,
    receive
      Data ->
          sender_ring_node(Data, Rep-1, Second)
    end.
```

Programa 3.20:Método sender_ring_node/3 de threadring.cerl

Pode ser observado que esse método funciona recursivamente. Cada vez que `sender_ring` é chamado, o contador de voltas (Rep) é decrementado. Quando a quantidade de repetições chega ao valor 0 (zero), o método é finalizado. O processos são modelados através da classe `proc.cerl` mostrada no Programa 3.21:

```
-class(proc).
-export([new/1,send/1]).
-constructor([new/1]).

attributes.

Pid.

methods.

new(RightProc) ->self::Pid = spawn( fun() -> loop(RightProc) end ).

send(Msg) ->    self::Pid ! Msg.

class_methods.

loop(RightProc) ->
      receive
      Data ->
      RightProc ! Data,
          loop(RightProc)
    end.
```

Programa 3.21: Classe proc

A classe possui um atributo chamado `“Pid”` e trata-se do Identificador do Processo (PId) do processo atual. Este PId é utilizado no método `new/1`. Este método cria um processo, armazenando seu PId no atributo `“Pid”` e faz com que o processo execute o método `loop/1`.

O método `loop/1` é o executado pelos processos do anel. Cada processo fica esperando o recebimento de uma mensagem, e assim que isto ocorre, ele envia a mesma mensagem para o processo seguinte do anel. Quando todos executarem este envio, todo o anel foi percorrido.

A classe `threadring.cerl` também importa métodos de outros módulos em Erlang, e as funções utilizadas são para armazenar os dados de tempo de criação de processos, tempo de envio de mensagens e tráfego na rede em um arquivo de saída.

Essa é a forma com a qual podemos modelar um anel de envio de mensagens entre processos utilizando a extensão $_{oo}$*Erlang*, tendo a possibilidade de utilizar todas as funções já presentes na linguagem Erlang, e com uma modelagem orientada a objetos.

3.8. Conclusões

Este capítulo mostra o uso do $_{oo}$*Erlang* para solução de problemas em orientação a objetos e como sua expressividade facilita a programação. Nos exemplos com padrões de projeto, que ilustram situações normais no dia-a-dia de um programador, Erlang "puro" mostrou-se inadequado, pois não possui uma sintaxe expressiva o suficiente. $_{oo}$*Erlang* conseguiu expressar os programas de uma maneira simples e intuitiva, semelhante a linguagem Java. Nos exemplos IMB, $_{oo}$*Erlang* foi capaz de modularizar o código de modo que um programador acostumado com Java por exemplo, conseguisse entender um típico programa Erlang. O Capítulo 4 desta tese, mostra o uso do $_{oo}$*Erlang* na programação dos Padrões de Projeto **[38]** para verificar o uso em programas orientados a objetos mais complexos.

4. PADRÕES DE PROJETO

The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer.

Edsger W. Dijkstra em
The Humble Programmer. 1972

Um desenvolvedor inexperiente pode se complicar devido à grande quantidade de opções e caminhos a seguir durante o desenvolvimento de um programa orientado a objetos. Uma maneira de minimizar os riscos é adotar boas práticas de projeto, ou padrões de projeto. Segundo **[39]**, em orientação a objetos, um *padrão* é uma descrição denominada de um problema e solução que pode ser aplicada a novos contextos. Para um problema repetidamente encontrado em projetos de *software*, é indicada uma "receita de bolo" que o soluciona. Em 1994, foi publicado o livro "Design Patterns" de autoria de Gamma, Helm, Johnson e Vlissides (os quatro autores ficaram conhecidos como *A Gangue dos Quatro – The Gang of Four – GoF*) **[38]**. Esse livro, que se tornou a "bíblia" dos padrões de projeto, definiu vinte e três diferentes tipos de padrões de projetos, subdivididos em diferentes tipos, de acordo com sua aplicação, e devidamente documentados para serem utilizados corretamente. Esses padrões foram desenvolvidos com base em diferentes problemas, quando uma

solução trivial não é aplicável a uma determinada situação. Deitel **[25]** afirma que os Padrões de Projetos são arquiteturas comprovadas para construir *software* orientado a objetos flexível e fácil de manter.

Um projeto para ser flexível precisa utilizar os padrões de projeto. A linguagem em que esse *software* será desenvolvida precisa suportar a sintaxe e as semânticas necessárias para criação de um padrão de projeto. Java, Scala, Python e Ruby possuem as primitivas necessárias para o desenvolvimento dos padrões.

4.1. Classificação dos padrões de projeto

Os padrões de projetos foram classificados em três tipos: *padrões de criação*, *padrões estruturais* e *padrões comportamentais*.

4.1.1. Padrões de criação

São padrões que ajudam a fazer um sistema independente de como seus objetos são criados, construídos e representados. A Tabela 4.1 descreve os padrões conforme foram definidos em **[38]**.

Tabela 4.1 : Padrões de criação

Padrão	Definição pela GoF
Abstract Factory	Fornece uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.
Builder	Separa a construção de um objeto complexo de sua representação de modo que o mesmo processo de construção possa criar diferentes representações.
Factory Method	Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe ser instanciada. O Factory Method permite postergar a instanciação às subclasses.
Prototype	Especifica os tipos de objetos a serem criados usando uma instância prototípica e cria novos objetos copiando este protótipo.
Singleton	Garante que uma classe só tenha uma única instância e fornece um ponto global de acesso a ela.

4.1.2. Padrões estruturais

Padrões estruturais consideram como classes e objetos são compostos para formar estruturas maiores. A Tabela 4.2 descreve os padrões conforme foram definidos em **[38]**.

Tabela 4.2 : Padrões estruturais

Padrões Estruturais	**Definição pela GoF**
Adapter	Converte a interface de uma classe em outra interface esperada pelos clientes. *Adapter* permite que certas classes trabalhem em conjunto, pois de outra forma seria impossível por causa de interfaces incompatíveis.
Bridge	Separa uma abstração de sua implementação, de modo que as duas possam variar independentemente.
Composite	Compõe objetos em estruturas de árvore para representar hierarquias do tipo todo-parte. *Composite* permite que clientes tratem objetos individuais e composições de objetos de maneira uniforme.
Decorator	Atribui responsabilidades adicionais a um objeto dinamicamente. *Decorators* fornecem uma alternativa flexível a subclasses para estender uma funcionalidade.
Façade	Fornece uma interface unificada para um conjunto de interfaces em subsistema. *Façade* define uma interface de nível mais alto que torna o subsistema mais fácil de usar.
Flyweight	Usa compartilhamento para suportar grandes quantidades de objetos, de granularidade fina, de maneira eficiente.
Proxy	Fornece um objeto representante (*surrogate*), ou um marcador de outro objeto, para controlar o acesso ao mesmo.

4.1.3. Padrões comportamentais

Padrões comportamentais concernem algoritmos e a atribuição de responsabilidades entre objetos, descrevendo não apenas padrões de objetos ou classes, mas também padrões de comunicação entre eles. A Tabela 4.3 descreve os padrões conforme foram definidos em **[38]**.

Tabela 4.3 : Padrões comportamentais

Padrões Comportamentais	**Definição pela GoF**
Chain of Responsibility	Evita o acoplamento do remetente de uma solicitação ao seu destinatário, dando a mais de um objeto a chance de tratar a requisição. Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto a trate.
Command	Encapsula uma solicitação como um objeto, desta forma permitindo a parametrização de clientes com diferentes solicitações, enfileiramento, registro de solicitações e suporte a operações reversíveis.

Interpreter	Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nessa linguagem.
Iterator	Fornece uma maneira de acessar sequencialmente os elementos de uma agregação de objetos sem expor sua representação subjacente.
Mediator	Define um objeto que encapsula a forma como um conjunto de objetos que interagem. *Mediator* promove o acoplamento fraco ao evitar que os objetos se refiram explicitamente uns aos outros, permitindo a variação de suas interações independentemente.
Memento	Sem violar o encapsulamento, captura e externaliza um estado interno de um objeto, de modo que o mesmo possa posteriormente ser restaurado para esse estado.
Observer	Define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto mude de estado, todos os seus dependentes são automaticamente notificados e atualizados.
State	Permite que um objeto altere o seu comportamento quando o seu estado interno muda. O objeto parecerá ter mudado de classe.
Strategy	Define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. *Strategy* permite que o algoritmo varie independentemente entre clientes que o utilizam.
Template Method	Define o esqueleto de um algoritmo em uma operação, postergando alguns passos para as subclasses. *Template Method* permite que as subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura.
Visitor	Representa uma operação a ser executada sobre os elementos de uma estrutura de objetos. *Visitor* permite definir uma nova operação sem mudar as classes dos elementos nos quais opera.

Esta tese mostra a implementação dos principais padrões de projetos utilizados no livro “Use a Cabeça – Padrões de Projetos” [26]. Os padrões do livro representam um conjunto significativo dos padrões mais comuns usados no dia-a-dia de um programador Java. Os exemplos são em inglês e foram implementados na linguagem Java. Os exemplos serão portados para $_{oo}$*Erlang* a fim de comparar a sintaxe da extensão com Java e mostrar que os padrões podem ser implementados de maneira eficiente.

4.2. Observer

O exemplo do uso do padrão *Observer* [38]é apresentado na Figura 4.1. Neste exemplo, uma estação meteorológica *weatherData* coleta informações dos sensores *weatherStation* e

weatherStationHeatIndex. Os dados coletados serão apresentados nos mostradores *statisticDisplay*, *forecastDisplay* e *heatIndexDisplay*. Toda vez que um dado é recebido pelos sensores, os mostradores devem automaticamente ser avisados. Esse problema é resolvido através do uso do padrão *Observer*.

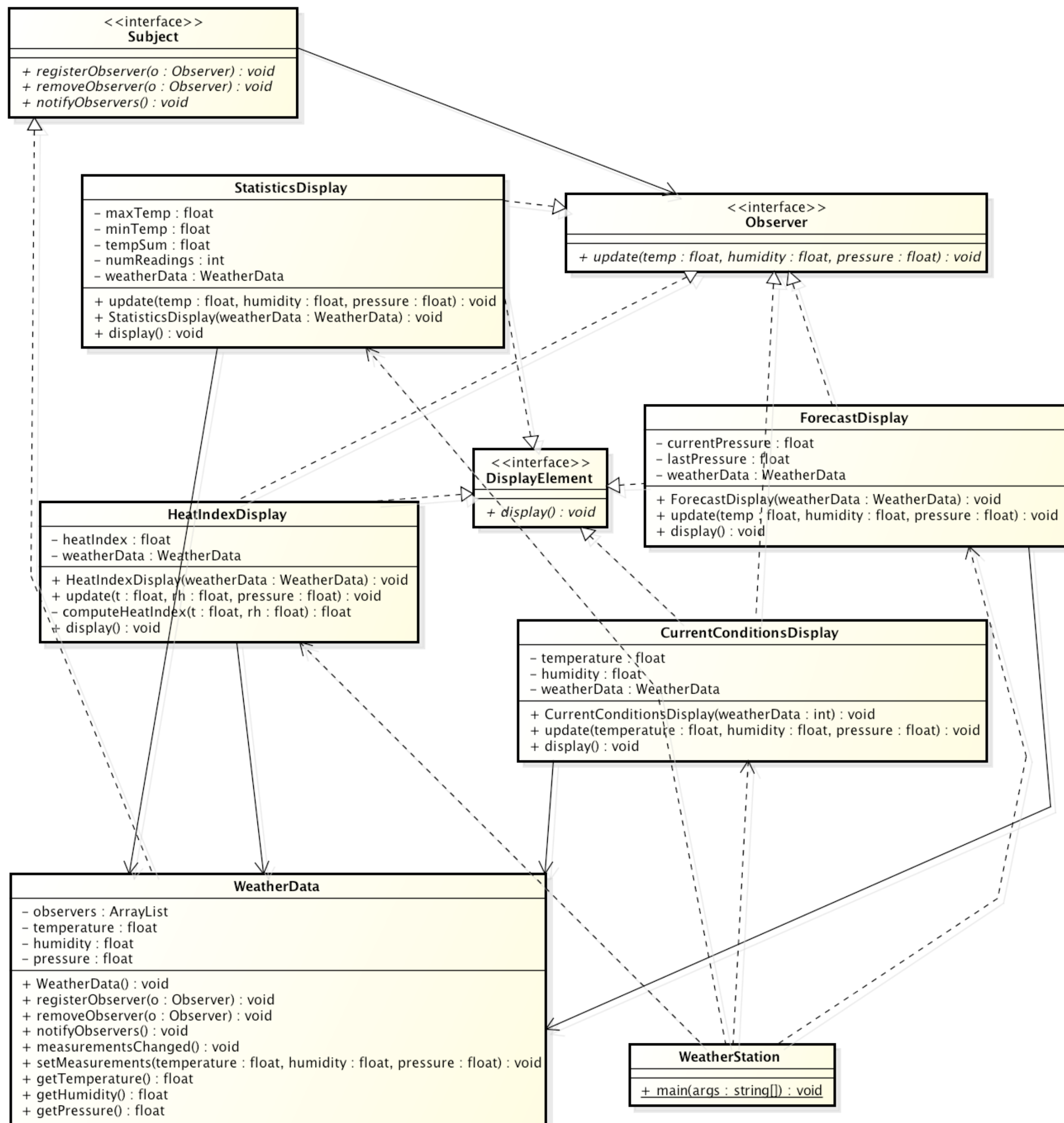


Figura 4.1 : Exemplo do padrão Observer

A classe *weatherData* implementa a interface *subject* (assunto a ser distribuído) e as classes *statisticDisplay*, *forecastDisplay* e *heatIndexDisplay* implementam a interface *observer* (os interessados em receber atualização do assunto). O código-fonte da *WeatherData* em $_{oo}$*Erlang* é apresentado no Programa 4.1:

```
-class(weatherData).
-implements(subject).
-export([new/0, measurements_changed/0, set_measurements/3, get_temperature/0,
get_humidity/0]).
-export([register_observer/1, remove_observer/1, notify_observers/0,
get_pressure/0]).
-constructor([new/0]).

attributes.

Observers;
Temperature;
Humidity;
Pressure.

methods.

new() ->
      self::Observers = [].

register_observer(Observer) ->
      self::Observers = [Observer | self::Observers].

remove_observer(Observer) ->
      lists:delete(Observer, self::Observers).

notify_observers() ->
      Observers = self::Observers,
      notify_aux(Observers).

notify_aux([]) -> ok;
notify_aux([Obs | Observers]) ->
      Obs::update(self::Temperature, self::Humidity, self::Pressure),
      notify_aux(Observers).

measurements_changed() ->
      notify_observers().

set_measurements(Temperature, Humidity, Pressure) ->
      self::Temperature = Temperature,
      self::Humidity = Humidity,
      self::Pressure = Pressure,
      measurements_changed().

get_temperature() ->
      self::Temperature.

get_humidity() ->
      self::Humidity.

get_pressure() ->
      self::Pressure.
```

Programa 4.1 : Classe weatherData

Esta classe possui o atributo `observers` que é uma lista dos observadores cadastrados para receber as notificações de mudança de estado. *statisticDisplay* é um observador e seu código-fonte é apresentado no Programa 4.8:

```
-class(statisticsDisplay).
-implements([observer, displayElement]).
-export([new/1, update/3, display/0]).
-constructor([new/1]).

attributes.

MaxTemp;
MinTemp;
TempSum;
NumReadings;
WeatherData.

methods.

new(WeatherData) ->
      self::MaxTemp = 0,
      self::MinTemp = 200,
      self::TempSum = 0,
      self::NumReadings = 0,
      self::WeatherData = WeatherData,
      Temp = self::WeatherData,
      Temp::register_observer({statisticsDisplay, ObjectID}).

update(Temp, Humidity, Pressure) ->
      self::TempSum = self::TempSum + Temp,
      self::NumReadings = self::NumReadings + 1,

      Maximum = self::MaxTemp,
      Minimum = self::MinTemp,

      if
            (Temp > Maximum) ->
                  self::MaxTemp = Temp;
            true ->
                  io:format("")
      end,
      if
            (Temp < Minimum) ->
                  self::MinTemp = Temp;
            true ->
                  io:format("")
      end,

      display().

display() ->
      io:format("Avg/Max/Min temperature = ~p/~p/~p ~n",
[(self::TempSum/self::NumReadings),

            self::MaxTemp,self::MinTemp]).
```
Programa 4.2: Classe statisticsDisplay

A classe *statisticsDisplay* implementa a interface *observer* e a interface *displayElement*, evitando assim os problemas de acoplamento da herança.

4.3. Decorator

O exemplo do uso do padrão *Decorator* é apresentado na Figura 4.2. Neste exemplo, o objetivo é criar um cardápio personalizável. Um cliente escolhe um tipo de café (*houseBlend*, *darkRoast*, *expresso*, *decaf*) e pode incrementá-lo com “condimentos” (*mocha*, *milk*, *whip*, *soy*). O custo total da bebida será de acordo com o tipo de café e os condimentos escolhidos. As bebidas são subclasses da classe abstrata *beverage*. Os condimentos são subclasses de *condimentDecorator* que também é subclasse de *beverage*. Cada condimento possui um atributo do tipo *beverage* de modo a poder encapsular novos condimentos.

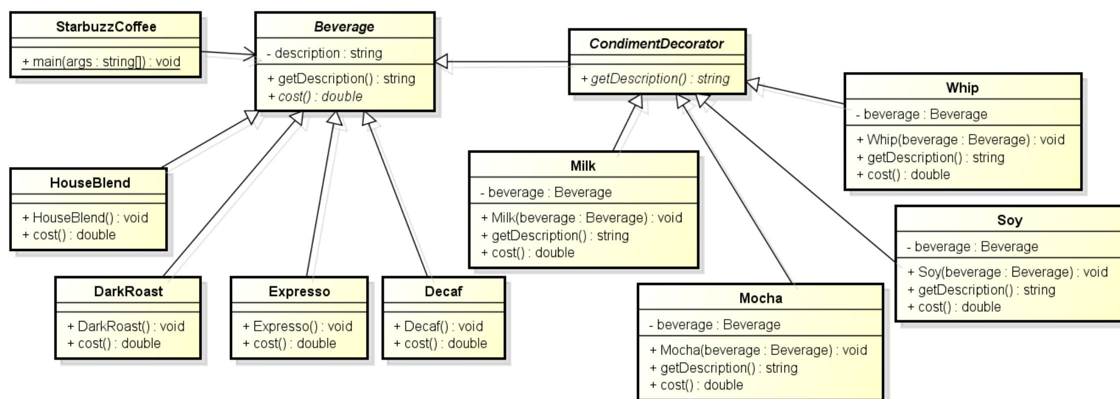


Figura 4.2 : Exemplo do padrão Decorator

O código-fonte da classe HouseBlend é apresentada no Programa 4.3

```
-class(houseBlend).
-extends(beverage).
-export([new/0, cost/0]).
-constructor([new/0]).

methods.

new() ->
      self::Description = "House Blend Coffee".

cost() -> 0.89.
```

Programa 4.3: Classe houseBlend

A classe *milk* é um tipo de condimento seu código-fonte é mostrado no Programa 4.4:

```
-class(milk).
-extends(condimentDecorator).
-export([new/1, get_description/0, add_cost/0]).
-constructor([new/1]).

attributes.

Beverage.

methods.

new(Beverage) ->
      self::Beverage = Beverage.

get_description() ->
      Temp = self::Beverage,
      Temp::get_description() ++ ", Milk".

add_cost() ->
      Temp = self::Beverage,
      0.10 + Temp::cost().
```

Programa 4.4: Classe milk

A classe *starbuzzCoffee* é encarregada do funcionamento do programa. Seu código-fonte é apresentado no Programa 4.5

```
-class(starbuzzCoffee).
-export([main/0]).

class_methods.

main() ->
      Beverage = expresso::new(),
      io:format("~p $ ~p~n", [Beverage::get_description(), Beverage::cost()]),

      %% um DarkRoast com mocha duplo e whip
      BevTemp1 = darkRoast::new(),
      BevTemp2 = mocha::new(BevTemp1),
      BevTemp3 = mocha::new(BevTemp2),
      BevTemp4 = whip::new(BevTemp3),
      io:format("~p $ ~p~n", [BevTemp4::get_description(), BevTemp4::cost()]),

      %% um HouseBlend com Soy, Mocha e Whip
      BevTemp5 = houseBlend::new(),
      BevTemp6 = soy::new(BevTemp5),
      BevTemp7 = mocha::new(BevTemp6),
      BevTemp8 = whip::new(BevTemp7),

      io:format("~p $ ~p~n", [BevTemp8::get_description(), BevTemp8::cost()]).
```

Programa 4.5: Classe starbuzzCoffee

A classe *starBuzzCoffee* cria dois pedidos: um *darkRoast* com mocha duplo e whip e um *houseBlend* com *soy*, *mocha* e *whip*.

4.4. Factory Method

O exemplo do uso do padrão *Factory Method* é apresentado na Figura 4.4. Neste exemplo, o objetivo é criar uma pizzaria onde vários tipos de pizza podem ser acrescentados sem a necessidade de modificações na estrutura do programa.

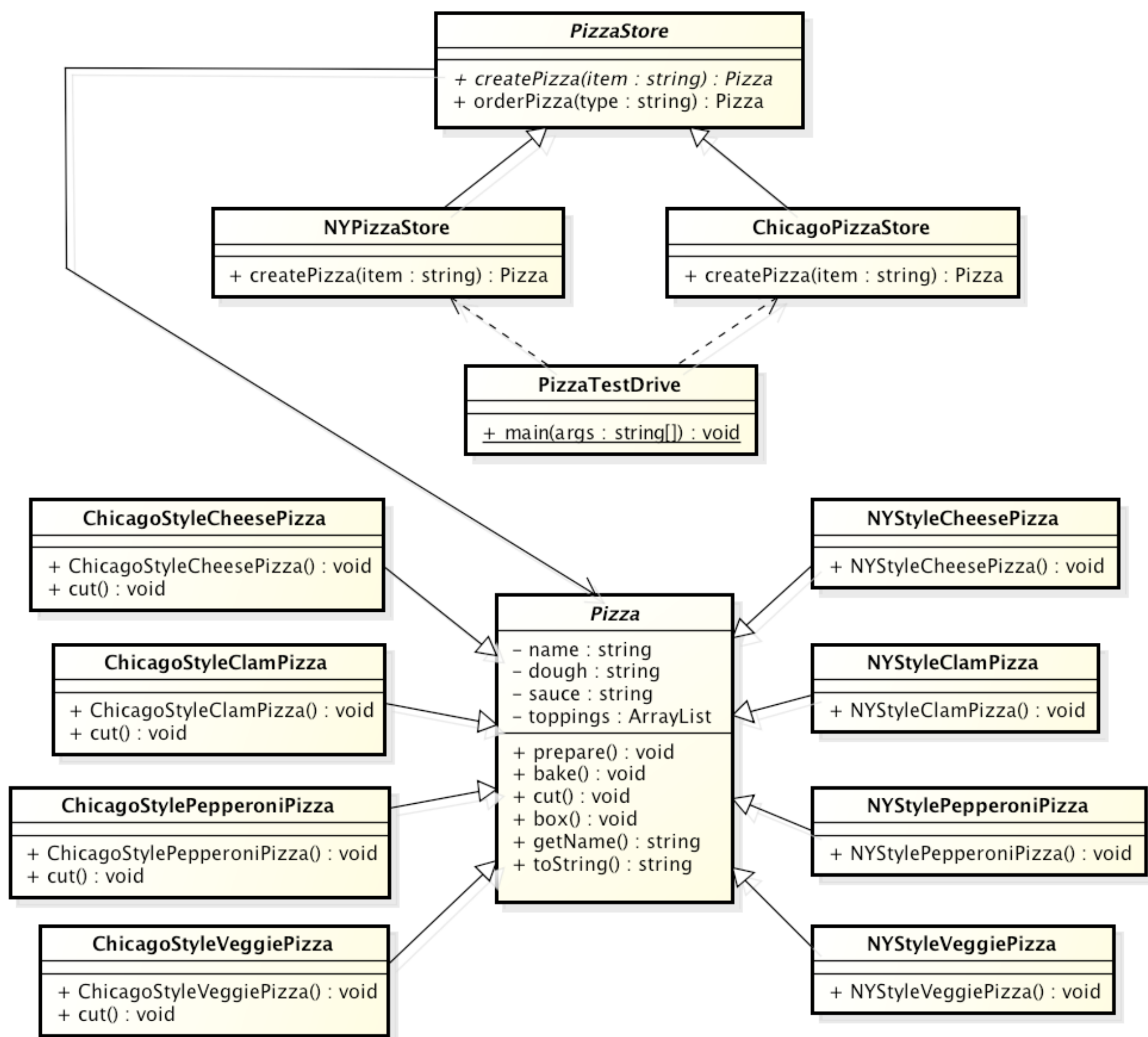


Figura 4.3 : Padrão *Factory Method*

Um exemplo de classe *pizzaStore* com dependências é apresentado no Programa 4.6.

```
-class(dependentPizzaStore).
-export([create_pizza/2]).

methods.

create_pizza(Style,Type) ->
```

```
    if
        (Style == "NY") ->
            if
                (Type == "cheese") ->
                    Pizza = nyStyleCheesePizza::new();
                (Type == "veggie") ->
                    Pizza = nyStyleVeggiePizza::new();
                (Type == "clam") ->
                    Pizza = nyStyleClamPizza::new();
                (Type == "pepperoni") ->
                    Pizza = nyStyleClamPizza::new()
            end;
        (Style == "Chicago") ->
            if
                (Type == "cheese") ->
                    Pizza = chicagoStyleCheesePizza::new();
                (Type == "veggie") ->
                    Pizza = chicagoStyleVeggiePizza::new();
                (Type == "clam") ->
                    Pizza = chicagoStyleClamPizza::new();
                (Type == "pepperoni") ->
                    Pizza = chicagoStyleClamPizza::new()
            end;
        _ ->
            io:format("Error: invalid type of pizza ~n"),
            null
    end,
    Pizza::prepare(),
    Pizza::bake(),
    Pizza::cut(),
    Pizza::box(),
    Pizza.
```

Programa 4.6: Classe dependentPizzaStore

A cada novo tipo de pizza criado, o código de *dependentePizzaStore* precisará ser modificado. Isso não é o ideal. Deve apenas criar um novo tipo de pizza (uma subclasse de *pizza*) e acrescentar seu "uso" na classe *pizzaTestDrive*. O código-fonte da classe *pizzaStore* é apresentado no Programa 4.7

```
-class(pizzaStore).
-export([create_pizza/1, order_pizza/1]).

methods.

create_pizza(Item) -> null.

order_pizza(Type) ->
    Pizza = pizza::new_(),
    Pizza::create_pizza(Type),
    io:format("--- Making a ~p --- ~n", [Pizza::get_name()]),
    Pizza::prepare(),
    Pizza::bake(),
    Pizza::cut(),
    Pizza::box(),
    Pizza.
```

Programa 4.7: Classe pizzaStore

A classe *pizzaStore* é responsável pela criação das pizzas por meio do método `create_pizza/1`. O método criará qualquer tipo de pizza. As pizzas são subclasses de *pizza*, apresentada no Programa 4.8

```
-class(pizza).
-export([prepare/0, bake/0, cut/0, box/0, get_name/0]).

attributes.

Name;
Dough;
Sauce;
Toppings.

methods.

prepare() ->
      io:format("Preparing ~p ~n", [self::Name]),
      io:format("Tossing dough... ~n"),
      io:format("Adding Sauce... ~n"),
      io:format("Adding toppings: ~n"),
      Tops = self::Toppings,
      prepare(Tops).

prepare([]) -> null;
prepare([Top|Toppings]) ->
      io:format("   ~p~n", [Top]),
      prepare(Toppings).

bake() ->
      io:format("Bake for 25 minutes at 350 ~n").

cut() ->
      io:format("Cutting the pizza into diagonal slices ~n").

box() ->
      io:format("Place pizza in official PizzaStore box").

get_name() ->
      self::Name.
```

Programa 4.8: Classe pizza

chicagoStylePepperoniPizza, uma subclasse de *pizza*, é apresentada no Programa 4.9.

```
-class(chicagoStylePepperoniPizza).
-extends(pizza).
-export([new/0, cut/0]).
-constructor([new/0]).

methods.

new() ->
      self::Name = "Chicago Style Pepperoni Pizza",
      self::Dough = "Extra Thick Crust Dough",
      self::Sauce = "Plum Tomato Sauce",

      self::Toppings = ["Shredded Mozzarella Cheese"|self::Toppings],
      self::Toppings = ["Black Olives"|self::Toppings],
      self::Toppings = ["Spinach"|self::Toppings],
      self::Toppings = ["Eggplant"|self::Toppings],
      self::Toppings = ["Sliced Pepperoni"|self::Toppings].

cut() ->
      io:format("Cutting the pizza into square slices ~n").
```

Programa 4.9: Classe chicagoStylePepperoniPizza

Utilizando pizzaStore como superclasse, pode-se especializar e criar “pizzarias” específicas por região. Essas subclasses criariam pizzas através da redefinicação do método `create_pizza/1`, o método que fabrica objetos (*Factory Method*). *nyPizzaStore* é apresentada no Programa 4.10.

```
-class(nyPizzaStore).
-extends(pizzaStore).
-export([create_pizza/1]).

methods.

create_pizza(Item) ->
      if
            (Item == "cheese") ->
                  nyStyleCheesePizza::new();
            (Item == "veggie") ->
                  nyStyleVeggiePizza::new();
            (Item == "clam") ->
                  nyStyleClamPizza::new();
            (Item == "pepperoni") ->
                  nyStylePepperoniPizza::new();
            _ ->
                  null
      end.
```

Programa 4.10**:** Classe nyPizzaStore

pizzaStoreTestDrive é a classe que invoca a aplicação. Ela é apresentada no Programa 4.11.

```
-class(pizzaTestDrive).
-export([main/0]).

class_methods.

main() ->
      NyStore = nyPizzaStore::new_(),
      ChicagoStore = chicagoPizzaStore::new_(),

      Pizza1 = NyStore::order_pizza("cheese"),
      io:format("Ethan ordered a ~p ~n", [Pizza1::get_name()]),

      Pizza2 = ChicagoStore::order_pizza("cheese"),
      io:format("Joel ordered a ~p ~n", [Pizza2::get_name()]),

      Pizza3 = NyStore::order_pizza("clam"),
      io:format("Ethan ordered a ~p ~n", [Pizza3::get_name()]),

      Pizza4 = ChicagoStore::order_pizza("clam"),
      io:format("Joel ordered a ~p ~n", [Pizza4::get_name()]),

      Pizza5 = NyStore::order_pizza("pepperoni"),
      io:format("Ethan ordered a ~p ~n", [Pizza5::get_name()]),

      Pizza6 = ChicagoStore::order_pizza("pepperoni"),
      io:format("Joel ordered a ~p ~n", [Pizza6::get_name()]),

      Pizza7 = NyStore::order_pizza("veggie"),
      io:format("Ethan ordered a ~p ~n", [Pizza7::get_name()]),

      Pizza8 = ChicagoStore::order_pizza("veggie"),
      io:format("Joel ordered a ~p ~n", [Pizza8::get_name()]).
```

Programa 4.11**:** Classe nyPizzaStore

4.5. Abstract Factory

O exemplo do uso do padrão *Abstract Method* é apresentado na Figura 4.5:

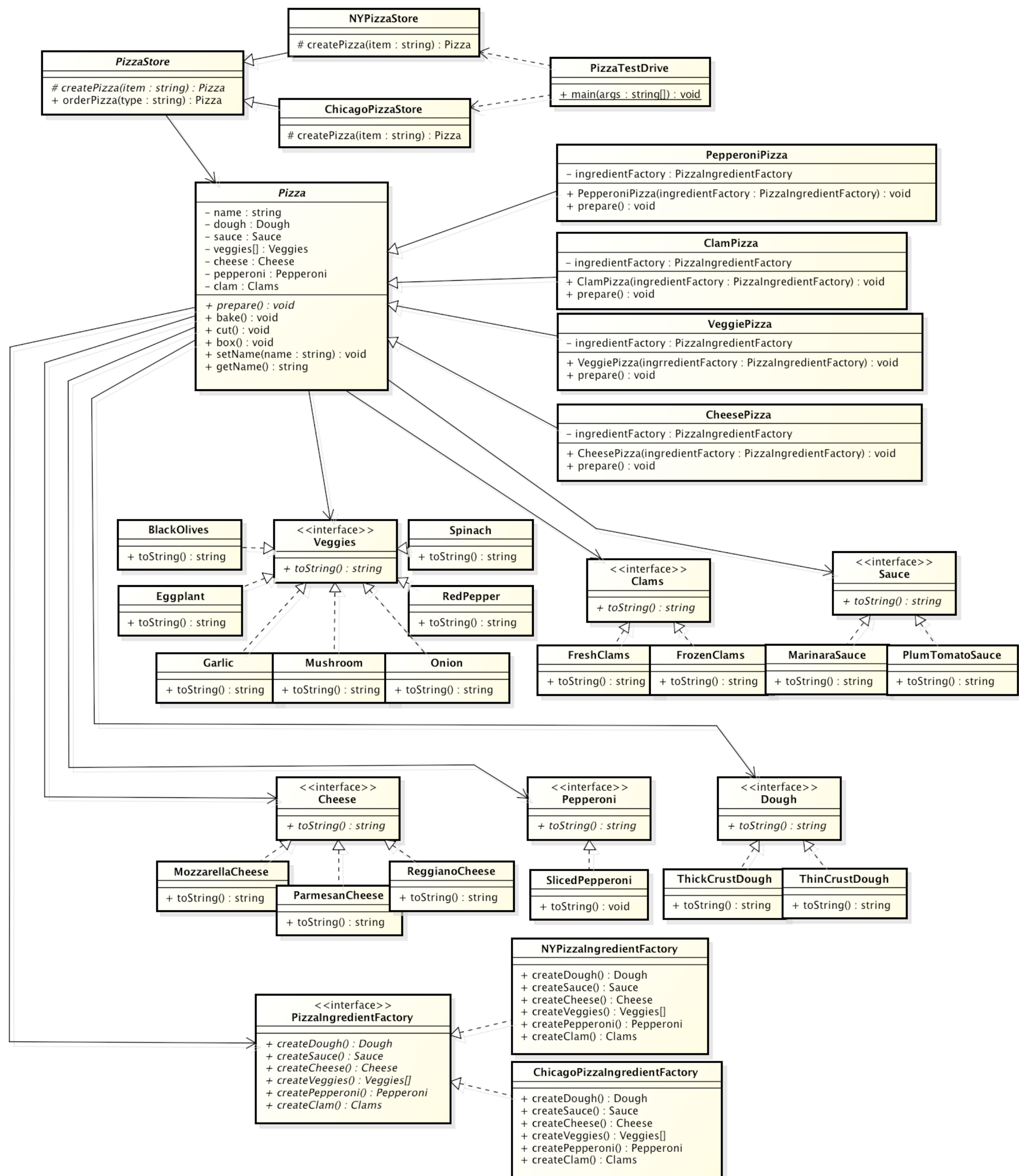


Figura 4.4 : Exemplo do *Design Pattern Abstract Factory*

Este exemplo é uma continuação do exemplo do *Factory Method*. Na *Abstract Factory* são criadas linhas de produtos, no caso do exemplo ingredientes das pizzas. A interface *pizzaIngredientFactory* é apresentada no programa 4.12

```
-interface(pizzaIngredientFactory).
-export([create_dough/0, create_sauce/0, create_cheese/0]).
-export([create_veggies/0, create_pepperoni/0, create_clam/0]).

methods.

create_dough().
create_sauce().
create_cheese().
create_veggies().
create_pepperoni().
create_clam().
```

Programa 4.12: Interface PizzaIngredientFactory

A classe *nyPizzaIngredientFactory*, cujo código-fonte é apresentado no Programa 4.13. Ela implementa a interface *pizzaIngredientFactory*.

```
-class(nyPizzaIngredientFactory).
-implements(pizzaIngredientFactory).
-export([create_dough/0, create_sauce/0, create_cheese/0, create_veggies/0]).
-export([create_pepperoni/0, create_clam/0]).

methods.

create_dough() ->
      Return = thinCrustDough::new_(),
      Return.

create_sauce() ->
      Return = marinaraSauce::new_(),
      Return.

create_cheese() ->
      Return = reggianoCheese::new_(),
      Return.

create_veggies() ->
      Garlic = garlic::new_(),
      Onion = onion::new_(),
      Mushroom = mushroom::new_(),
      RedPepper = redPepper::new_(),
      Veggies = [Garlic, Onion, Mushroom, RedPepper],
      Veggies.

create_pepperoni() ->
      Return = slicedPepperoni::new_(),
      Return.

create_clam() ->
      Return = freshClams::new_(),
      Return.
```

Programa 4.13: Interface nyPizzaIngredientFactory

Cada fábrica prepara os ingredientes de acordo com sua necessidade. A classe *nyPizzaStore*, que utiliza a fábrica, é apresentada no Programa 4.14.

```
-class(nyPizzaStore).
-extends(pizzaStore).
-exports([create_pizza/1]).

methods.

create_pizza(Item) ->
      IngredientFactory = nyPizzaIngredientFactory::new_(),

      if
            (Item == "cheese") ->
                  Pizza = cheesePizza::new(IngredientFactory),
                  Pizza::set_name("New York Style Cheese Pizza");
            (Item == "veggie") ->
                  Pizza = veggiePizza::new(IngredientFactory),
                  Pizza::set_name("New York Style Veggie Pizza");
            (Item == "clam") ->
                  Pizza = clamPizza::new(IngredientFactory),
                  Pizza::set_name("New York Style Clam Pizza");
            (Item == "pepperoni") ->
                  Pizza = pepperoniPizza::new(IngredientFactory),
                  Pizza::set_name("New York Style Pepperoni Pizza")
      end,
      Pizza.
```

Programa 4.14: Classe pizzaIngredientFactory

O atributo `IngredientFactory` recebe uma *nyPizzaIngredientFactory* e a repassa para uma subclasse de pizza. A classe *cheesePizza* é apresentada no Programa 4.15.

```
-class(cheesePizza).
-extends(pizza).
-export([new/1, prepare/0]).
-constructor([new/1]).

attributes.

IngredientFactory.

methods.

new(IngredientFactory) ->
      self::IngredientFactory = IngredientFactory.

prepare() ->
      io:format("Preparing ~p~n", [self::Name]),
      self::Dough = IngredientFactory::create_dough(),
      self::Sauce = IngredientFactory::create_sauce(),
      self::Cheese = IngredientFactory::create_cheese().
```

Programa 4.15: Classe cheesePizza

A classe *pizzaTestDrive* é responsável por fazer funcionar a aplicação. Seu código-fonte é apresentado no Programa 4.16.

```
-class(pizzaTestDrive).
-export([main/0]).

class_methods.
```

```
main() ->
      NyStore = nyPizzaStore::new_(),
      chicagoStore = chicagoPizzaStore::new_(),

      Pizza1 = NyStore::order_pizza("cheese"),
      io:format("Ethan ordered a ~p ~n", [Pizza1::get_name()]),

      Pizza2 = ChicagoStore::order_pizza("cheese"),
      io:format("Joel ordered a ~p ~n", [Pizza2::get_name()]),

      Pizza3 = NyStore::order_pizza("clam"),
      io:format("Ethan ordered a ~p ~n", [Pizza3::get_name()]),

      Pizza4 = ChicagoStore::order_pizza("clam"),
      io:format("Joel ordered a ~p ~n", [Pizza4::get_name()]),

      Pizza5 = NyStore::order_pizza("pepperoni"),
      io:format("Ethan ordered a ~p ~n", [Pizza5::get_name()]),

      Pizza6 = ChicagoStore::order_pizza("pepperoni"),
      io:format("Joel ordered a ~p ~n", [Pizza6::get_name()]),

      Pizza7 = NyStore::order_pizza("veggie"),
      io:format("Ethan ordered a ~p ~n", [Pizza7::get_name()]),

      Pizza8 = ChicagoStore::order_pizza("veggie"),
      io:format("Joel ordered a ~p ~n", [Pizza8::get_name()]).
```

Programa 4.16**:** Classe pizzaTestDrive

4.6. Command

O exemplo do uso do padrão *Command* é apresentado na Figura 4.5:

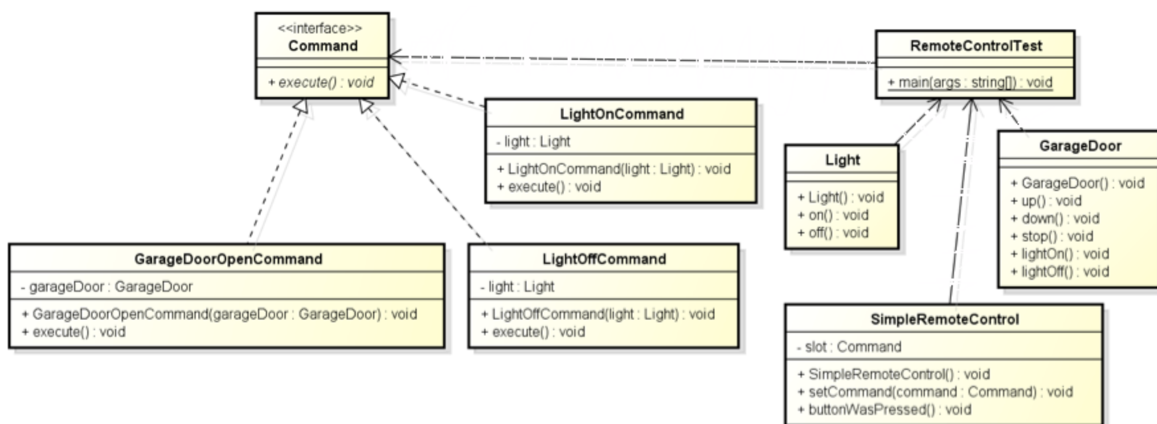


Figura 4.5 : Exemplo do *Design Pattern Command*

O objetivo do padrão é encapsular chamadas a métodos. Neste exemplo, a meta é projetar um controle remoto para os diversos aparelhos eletrônicos de uma casa. A interface *command* é apresentada no Programa 4.17.

```
-interface(command).
-export([execute/0]).

methods.

execute().
```

Programa 4.17: Interface Command

A interface é implementada pelas classes *garageDoorOpenCommand*, *lightOnCommand*, *lightOffCommand*. Elas são apresentadas no Programa 4.18.

GarageDoorOpenCommand	LightOnCommand	LightOffCommand

-class(garageDoorOpenCommand).
-implements(command).
-export([new/1, execute/0]).
-constructor([new/1]).
 	
attributes.
 	
GarageDoor.
 	
methods.
 	
new(GarageDoor) ->
      self::GarageDoor =
GarageDoor.
 	
execute() ->
      Temp = self::GarageDoor,
      Temp::up().
	-class(lightOnCommand).
	-implements(command).
	-export([new/1, execute/0]).
	-constructor([new/1]).
	
	attributes.
	
	Light.
	
	methods.
	
	new(Light) ->
	      self::Light = Light.
	
	execute() ->
	      Temp = self::Light,
	      Temp::on().
		-class(lightOffCommand).
		-implements(command).
		-export([new/1,
		execute/0]).
		-constructor([new/1]).
		
		attributes.
		
		Light.
		
		methods.
		
		new(Light) ->
		      self::Light =
		Light.
		
		execute() ->
		      Temp =
		self::Light,
		      Temp::off().

Programa 4.18: Classes que implementam Command

As classes *garageDoor* e *light* têm seus métodos encapsulados, apresentadas no Programa 4.19.

GarageDoor	Light

-class(garageDoorOpenCommand).
-implements(command).
-export([new/1, execute/0]).
-constructor([new/1]).
 	
attributes.
 	
GarageDoor.
 	
methods.
 	
new(GarageDoor) ->
      self::GarageDoor = GarageDoor.
 	
execute() ->
      Temp = self::GarageDoor,
      Temp::up().
	-class(lightOffCommand).
	-implements(command).
	-export([new/1, execute/0]).
	-constructor([new/1]).
	
	attributes.
	
	Light.
	
	methods.
	
	new(Light) ->
	      self::Light = Light.
	
	execute() ->
	      Temp = self::Light,
	      Temp::off().

Programa 4.19: Classes com métodos encapsulados

As classes *garageOpenDoor*, *lightOnCommand*e *lightOffCommand*são embarcadas na classe *simpleRemoteControl*, apresentada no Programa 4.20

```
-class(simpleRemoteControl).
-export([set_command/1, button_was_pressed/0]).

attributes.

Slot.

methods.

set_command(Command) ->
      self::Slot = Command.

button_was_pressed() ->
      Temp = self::Slot,
      Temp::execute().
```

Programa 4.20: Classe simpleRemoteControl

A classe *remoteControlTest*, encarregada do funcionamento da aplicação, é apresentada no Programa 4.21.

```
-class (remoteControlTest)
-exports([main/0]).

class_methods.

main() ->
           Remote = simpleRemoteControl::new_();
           Light = light::new_();
           GarageDoor = garageDoor::new_();

LightOn = lightOnCommand::new(Light);
           GarageOpen = garageDoorOpenCommand::new(GarageDoor);

           Remote::setCommand(lightOn);
           Remote::buttonWasPressed();
           Remote::setCommand(garageOpen);
           Remote::buttonWasPressed();
    }
}
```

Programa 4.21: Classe remoteControlTest

4.7. Adapter

O exemplo do uso do padrão *Adapter* é apresentado na Figura 4.6:

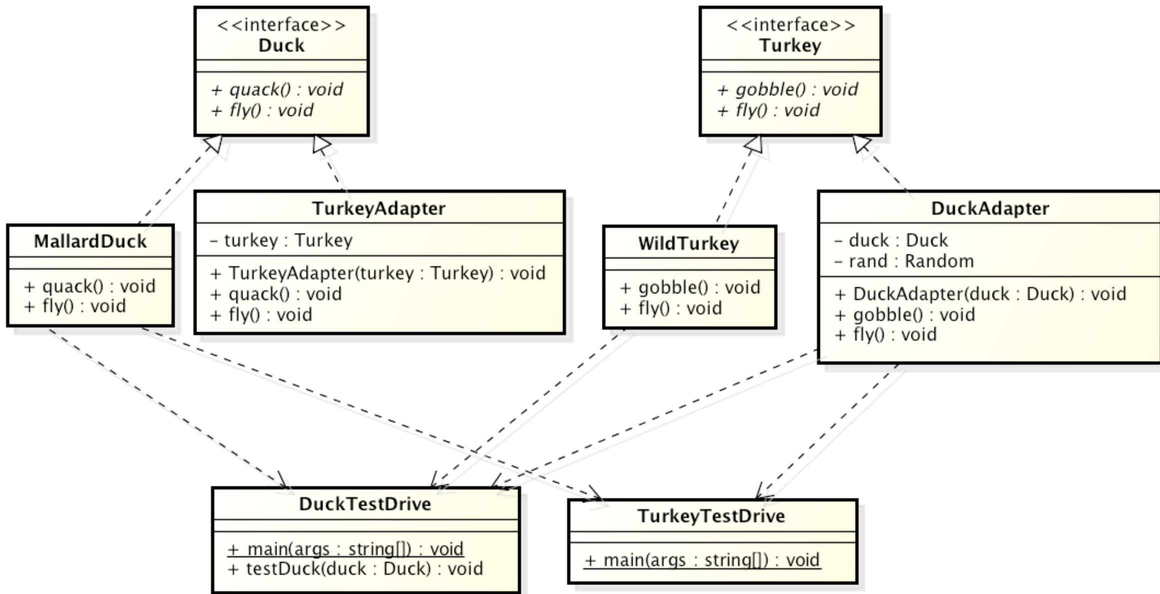


Figura 4.6**:** Exemplo do *Design Pattern Adapter*

O objetivo deste padrão é fazer com que classes com interfaces incompatíveis trabalhem em conjunto. No exemplo, um pato é adaptado a um peru (*turkeyAdapter*) e um peru é adaptado a um pato (*duckAdapter*). As classes são mostradas no Programa 4.22.

TurkeyAdapter

```
-class(turkeyAdapter).
-implements(duck).
-export([new/1, quack/0, fly/0]).
-constructor([new/1]).

attributes.
Turkey.

methods.
new(Turkey) ->
	self::Turkey = Turkey.
quack() ->
	Temp = self::Turkey,
	Temp::gobble().
fly() ->
	Temp = self::Turkey,
	Temp::fly(),
	Temp::fly(),
	Temp::fly(),
	Temp::fly(),
	Temp::fly().
```

DuckAdapter

```
-class(duckAdapter).
-implements(turkey).
-export([new/1, gobble/0, fly/0]).
-constructor([new/1]).

attributes.
Duck.

methods.
new(Duck) ->
	self::Duck = Duck.
gobble() ->
	Temp = self::Duck,
	Temp::quack().
fly() ->
	Random = random:uniform(5),
	if
		(Random == 0) ->
			Temp = self::Duck,
			Temp::fly();
		true ->
			io:format("")
	end.
```

Programa 4.22**:** Classes adaptadoras

As classes *duck* e *turkey* possuem interfaces incompatíveis. Elas são apresentadas no programa 4.23.

TurkeyTestDrive

```
-class(wildTurkey).
-implements(turkey).
-export([gobble/0, fly/0]).

methods.

gobble() ->
      io:format("Gobble gobble ~n").

fly() ->
      io:format("I'm flying a short
distance ~n").
```

MallardDuck

```
-class(mallardDuck).
-implements(duck).
-export([quack/0, fly/0]).

methods.

quack() ->
      io:format("Quack ~n").

fly() ->
      io:format("I'm flying ~n").
```

Programa 4.23: Classes com métodos encapsulados

A classe *turkeyTestDrive* e *duckTestDrive* fazem a aplicação funcionar. Elas são apresentadas no Programa 4.24.

TurkeyTestDrive

```
-class(turkeyTestDrive).
-export([main/0]).

class_methods.

main() ->
      Duck = mallardDuck::new_(),
      DuckAdapter =
duckAdapter::new(Duck),

      io:format("The DuckAdapter
says..."),
      DuckAdapter::gobble(),
      DuckAdapter::fly(),
      io:format("The DuckAdapter
says..."),
      DuckAdapter::gobble(),
      DuckAdapter::fly(),
      io:format("The DuckAdapter
says..."),
      DuckAdapter::gobble(),
      DuckAdapter::fly(),
      io:format("The DuckAdapter
says..."),
      DuckAdapter::gobble(),
      DuckAdapter::fly(),
      io:format("The DuckAdapter
says..."),
      DuckAdapter::gobble(),
      DuckAdapter::fly(),
      io:format("The DuckAdapter
says..."),
      DuckAdapter::gobble(),
```

DuckTestDrive

```
-class(duckTestDrive).
-export([main/0, test_duck/1]).

class_methods.

main() ->
      Duck = mallardDuck::new_(),
      Turkey = wildTurkey::new_(),
      TurkeyAdapter =
turkeyAdapter::new(Turkey),

      io:format("The turkey says... ~n"),
      Turkey::gobble(),
      Turkey::fly(),

      io:format("~nThe Duck says... ~n"),
      test_duck(Duck),

      io:format("~nThe TurkeyAdapter says...
~n"),
      test_duck(TurkeyAdapter).

test_duck(Duck) ->
      Duck::quack(),
      Duck::fly().
```

```
      DuckAdapter::fly(),
      io:format("The DuckAdapter
says..."),
      DuckAdapter::gobble(),
      DuckAdapter::fly(),
      io:format("The DuckAdapter
says..."),
      DuckAdapter::gobble(),
      DuckAdapter::fly(),
      io:format("The DuckAdapter
says..."),
      DuckAdapter::gobble(),
      DuckAdapter::fly(),
      io:format("The DuckAdapter
says..."),
      DuckAdapter::gobble(),
      DuckAdapter::fly().
```

Programa 4.24 : Classes com métodos encapsulados

4.8. Façade

O exemplo do uso do padrão Façade é apresentado na Figura 4.7:

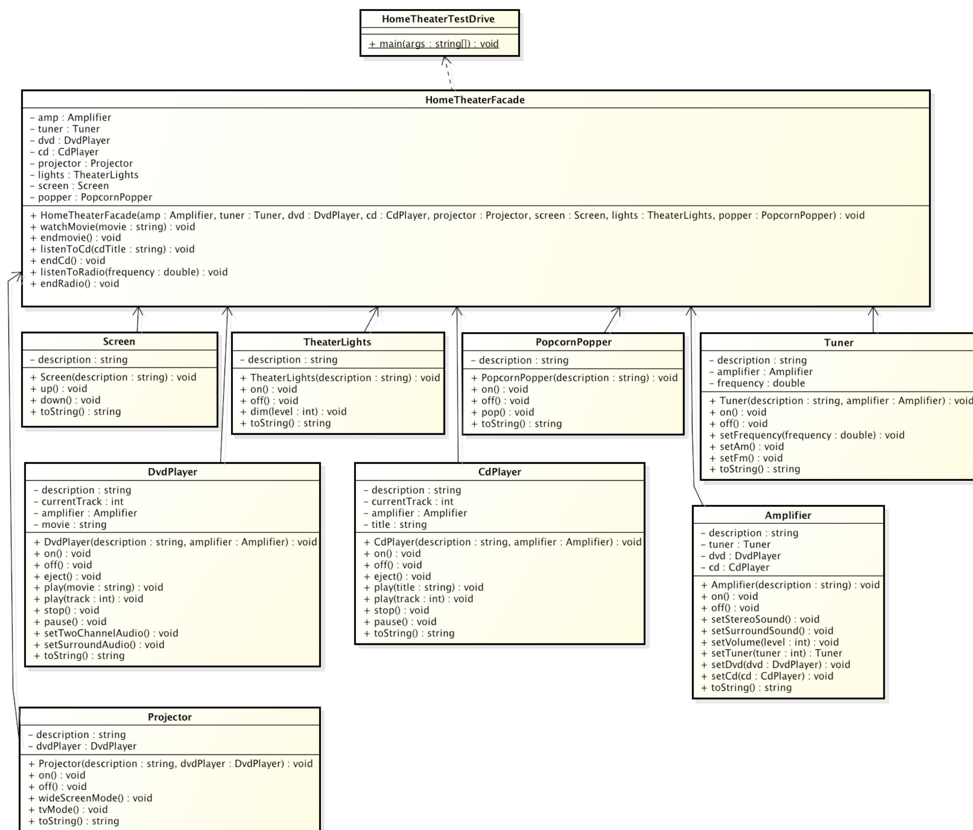


Figura 4.7 : Exemplo do *Design Pattern Façade*

Esse padrão fornece uma interface integrada para um conjunto de interfaces de um subsistema. No exemplo, as interfaces de diversos aparelhos eletrônicos são integradas em uma interface simplificada chamada *homeTheaterFacade*. Seu código-fonte é apresentado no Programa 4.25

```
-class(homeTheaterFacade).
-export([new/8, watch_movie/1, end_movie/0, listen_to_cd/1, end_cd/0]).
-export([listen_to_radio/1, end_radio/0]).
-constructor([new/8]).

attributes.

Amp;
Tuner;
Dvd;
Cd;
Projector;
Lights;
Screen;
Popper.

methods.

new(Amp,Tuner,Dvd,Cd,Projector,Screen,Lights,Popper) ->
      self::Amp = Amp,
      self::Tuner = Tuner,
      self::Dvd = Dvd,
      self::Cd = Cd,
      self::Projector = Projector,
      self::Screen = Screen,
      self::Lights = Lights,
      self::Popper = Popper.

watch_movie(Movie) ->
      io:format("Get ready to watch a movie...~n"),
      Temp1 = self::Popper,
      Temp1::on(),
      Temp1::pop(),
      Temp2 = self::Lights,
      Temp2::dim(10),
      Temp3 = self::Screen,
      Temp3::down(),
      Temp4 = self::Projector,
      Temp4::on(),
      Temp4::widescreen_mode(),
      Temp5 = self::Amp,
      Temp5::on(),
      Temp5::set_dvd(self::Dvd),
      Temp5::set_surround_sound(),
      Temp5::set_volume(5),
      Temp6 = self::Dvd,
      Temp6::on(),
      Temp6::play(Movie).

end_movie() ->
      io:format("Shutting movie theater down...~n"),
      Temp1 = self::Popper,
      Temp1::off(),
      Temp2 = self::Lights,
      Temp2::on(),
      Temp3 = self::Screen,
      Temp3::up(),
```

```
      Temp4 = self::Projector,
      Temp4::off(),
      Temp5 = self::Amp,
      Temp5::off(),
      Temp6 = self::Dvd,
      Temp6::stop(),
      Temp6::eject(),
      Temp6::off().

listen_to_cd(CdTitle) ->
      io:format("Get ready for an audiopile experience...~n"),
      Temp1 = self::Lights,
      Temp1::on(),
      Temp2 = self::Amp,
      Temp2::on(),
      Temp2::set_volume(5),
      Temp2::set_cd(),
      Temp2::set_stereo_sound(),
      Temp3 = self::Cd,
      Temp3::on(),
      Temp3::play(CdTitle).

end_cd() ->
      io:format("Shutting down CD...~n"),
      Temp1 = self::Amp,
      Temp1::off(),
      Temp1::set_cd(self::Cd),
      Temp2 = self::Cd,
      Temp2::eject(),
      Temp2::off().

listen_to_radio(Frequency) ->
      io:format("Tunning in the airwaves...~n"),
      Temp1 = self::Tuner,
      Temp1::on(),
      Temp1::set_frequency(Frequency),
      Temp2 = self::Amp,
      Temp2::on(),
      Temp2::set_volume(5),
      Temp2::set_tuner(self::Tuner).

end_radio() ->
      io:format("Shutting down the tuner...~n"),
      Temp1 = self::Tuner,
      Temp1::off(),
      Temp2 = self::Amp,
      Temp2::off().
```

Programa 4.25: HomeTheaterFacade

4.9. Template Method

O exemplo do uso do padrão *Template Method* é apresentado na Figura 4.8:

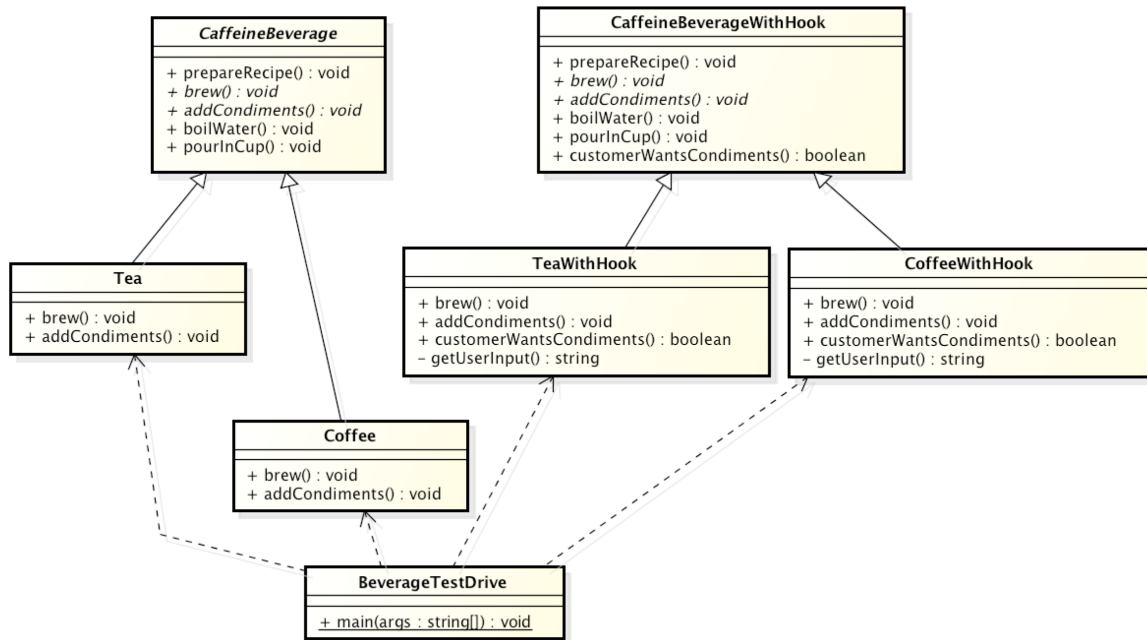


Figura 4.8 : Exemplo do *Design Pattern Template Method*

O objetivo do padrão é encapsular pedaços de algoritmos, para que as subclasses possam se conectar diretamente a um processo computacional sempre que precisarem. No exemplo, a preparação dos cafés (*coffee*) e chás (*tea*) são muito parecidas. Programá-las em separado causaria duplicação de código. A preparação então é encapsulada utilizando o *Template Method*. A classe *caffeineBeverage* é apresentada no Programa 4.26.

```
-class(caffeineBeverage).
-export([prepare_recipe/0, brew/0, add_condiments/0]).
-export([boil_water/0, pour_in_cup/0]).

methods.

prepare_recipe() ->
      boil_water(),
      brew(),
      pour_in_cup(),
      add_condiments().

brew() -> null.

add_condiments() -> null.

boil_water() ->
      io:format("Boiling water~n").

pour_in_cup() ->
      io:format("Pouring into cup~n").
```

Programa 4.26: Classe caffeineBeverage

As classes *tea* e *coffee* são apresentadas no Programa 4.27.

Tea	Coffee

-**class**(tea).
-**extends**(caffeineBeverage).
-**export**([brew/0, add_condiments/0]).
 	
methods.
 	
brew() ->
      io:format("Steeping the tea~n").
 	
add_condiments() ->
      io:format("Adding Lemon~n").
	-**class**(coffee).
	-**extends**(caffeineBeverage).
	-**export**([brew/0, add_condiments/0]).
	
	methods.
	
	brew() ->
	      io:format("Dripping Coffee through filter~n").
	
	add_condiments() ->
	      io:format("Adding Sugar and Milk~n").

Programa 4.27 : Classes concretas tea e coffee

A classe *caffeineBeverageWithHook* é apresentada no Programa 4.28.

```
-class(caffeineBeverageWithHook).
-export([prepare_recipe/0, brew/0, add_condiments/0]).
-export([boil_water/0, pour_in_cup/0, customer_wants_condiments/0]).

methods.

prepare_recipe() ->
      boil_water(),
      brew(),
      pour_in_cup(),
      Return = customer_wants_condiments(),
      if
            (Return == true) ->
                  add_condiments();
            true ->
                  io:format("")
      end.

brew() -> null.

add_condiments() -> null.

boil_water() ->
      io:format("Boiling water~n").

pour_in_cup() ->
      io:format("Pouring into cup~n").

customer_wants_condiments() ->
      true.
```

Programa 4.28: Classe caffeineBeverageWithHook

As classes *teaWithHook* e *coffeeWithHook* estendem a classe *caffeineBeverageWithHook*. São apresentadas no Programa 4.29.

TeaWithHook	CoffeeWithHook

`-class(tea).`
`-extends(caffeineBeverage).`
`-export([brew/0, add_condiments/0]).`
 	
`methods.`
 	
`brew() ->`
`    io:format("Steeping the tea~n").`
 	
`add_condiments() ->`
`    io:format("Adding Lemon~n").`
	`-class(coffee).`
	`-extends(caffeineBeverage).`
	`-export([brew/0, add_condiments/0]).`
	
	`methods.`
	
	`brew() ->`
	`    io:format("Dripping Coffee through filter~n").`
	
	`add_condiments() ->`
	`    io:format("Adding Sugar and Milk~n").`

Programa 4.29 : Classes concretas teaWithHook e coffeeWithHook

A classe *beverageTestDrive*, apresentada em 4.30, faz a aplicação funcionar.

```
-class(beverageTestDrive).
-export([main/0]).

class_methods.

main() ->
      Tea = tea::new_(),
      Coffee = coffee::new_(),

      io:format("~nMaking Tea...~n"),
      Tea::prepare_recipe(),

      io:format("~nMaking Coffee...~n"),
      Coffee::prepare_recipe(),

      TeaHook = teaWithHook::new_(),
      CoffeeHook = coffeeWithHook::new_(),

      io:format("~nMaking Tea...~n"),
      TeaHook::prepare_recipe(),

      io:format("~nMaking Coffee...~n"),
      CoffeeHook::prepare_recipe().
```

Programa 4.30: Classe beverageTestDrive

4.10. Conclusões

Os padrões de projeto são um conjunto de boas práticas descobertas e catalogadas para resolver problemas recorrentes de programação orientada a objetos. Uma linguagem com suporte a orientação a objetos para ser amplamente utilizada deve prover todos os mecanismos necessários para implementação desses padrões. Este capítulo mostra que $_{oo}$*Erlang* é capaz de implementar os padrões de projeto. Implementar esses padrões em Erlang seria inconveniente devido à complexidade conforme apresentado no Capítulo 3 desta tese. Implementar em ECT não seria possível conforme visto na Introdução desta tese **[62]**.

5. ANÁLISE DE DESEMPENHO DE $_{oo}$*Erlang* E OUTRAS LINGUAGENS

The world is parallel.

Joe Armstrong no livro
Programming Erlang – Software For
a Concurrent World

Este capítulo apresenta a comparação de desempenho entre Java **[2]**, Scala **[6]**, Python **[5]**, Ruby **[4]**, Erlang **[1]** e a extensão $_{oo}$*Erlang*. Essas linguagens foram escolhidas porque são utilizadas para criação de grandes aplicações Web 2.0 em um contexto de concorrência massiva **[47]-[50][52][53]**. A comparação é realizada por meio de testes criados pela Intel, conhecidos como Intel MPI Benchmark (IMB) **[19]**, já apresentados no Capítulo 3 desta tese. Os testes simulam o uso em situações de concorrência massiva. O esperado é que a extensão não cause degradação de desempenho no Erlang, ou seja, o tempo de execução dos programas em $_{o}$*oErlang* e Erlang devem ser praticamente iguais.

5.1. Intel MPI Benchmark (IMB)

O Intel MPI Benchmark (IMB) é um conjunto de programas de teste desenvolvidos pela Intel para avaliar de maneira eficiente as mais importantes funcionalidades da biblioteca *Message*

Passing Interface (MPI), bem como o desempenho de um conjunto de processadores executando algoritmos concorrentemente [19].

Os testes são divididos em três categorias:

1. **Transferência simples** – Somente uma mensagem é trocada entre dois processos.
2. **Transferência paralela** – Somente uma mensagem é trocada entre dois processos, mas muitos pares de processos se comunicam simultaneamente.
3. **Transferência coletiva** – Muitos processos trabalham juntos para realizar uma determinada tarefa.

A Tabela 5.1 apresenta os testes classificados nessas três categorias:

Tabela 5.1 **: Classificação dos testes**

Transferência simples	Transferência paralela	Transferência coletiva
PingPong	SendRec	BCast
PingPongSpecifiedSource	Exchange	Allgather, Allgatherv
PingPing	Multi-PingPong	Alltoall, Alltoallv
PingPingSpecifiedSource	Multi-PingPing	Scatter, Scatterv
	Multi-SendRec	Gather, Gatherv
		Multi-Exchange Reduce
		Reduce_scatter
		Allreduce
		Barrier

Os testes escolhidos foram PingPing, PingPong e SendRec. PingPing e PingPong foram selecionados por exemplificar a passagem síncrona e assíncrona de mensagens entre apenas dois processos, ou seja, os dois casos mais simples de comunicação. O SendRec foi selecionado por requerer a criação massiva de processos e a comunicação entre eles e ser o mais simples teste de transferência paralela. Como algumas linguagens não completaram os testes SendRec, não houve necessidade de executar os demais testes de transferência paralela e nem os testes de transferência coletiva pois esses utilizam muito mais recursos que os de transferência paralela.

5.2. Ambiente dos Experimentos

Cada teste foi executado dez vezes e depois calculadas a média, variância, desvio padrão e intervalo de confiança (~95%) do tempo. Esses dados se encontram no sítio deste projeto [74]. Todos os testes são executados na plataforma indicada na Tabela 5.2:

Tabela 5.2 **:** Ambiente de teste

Sistema Operacional	Ubuntu 11.04 (natty) GNU/Linux 2.6.38-8-server x86 64
Hardware	Processador Intel Core 2 Duo CPU E7400 - 2.8 GHz Memória RAM: 4 Gb DDR2 HD: 250 Gb - SATA 2
Linguagens de programação	Java Oracle Version - 1.6.0 25 - Java HotSpot (TM) 64-Bit Server - Build 20.1-b02 Erlang R14B03 - (64-bits) - Eshell Version - 5.8.4 Scala version 2.9.0.1. Python 2.7.3 Ruby 1.8.7

As máquinas alocadas para os testes estão dedicadas exclusivamente aos mesmos e funcionam sobre o modo terminal, não executando nenhum outro processo de relevante concorrência, nem mesmo o ambiente gráfico do Ubuntu.

As linguagens de programação funcionam sobre máquinas virtuais **[10][57]**. Cada uma delas foi configurada conforme mostra a Tabela 5.3

Tabela 5.3 : Configurações das linguagens

Linguagem **Legenda**	**Linha de comando**	**Descrição**
Erlang	`erl -noshell -eval ...`	Execução *default* de um programa em Erlang
$_{oo}$*Erlang*	`erl -noshell -eval ...`	Execução *default* de um programa em Erlang
Java Normal	`java -verbose:gc -Xloggc:java.log ...`	Execução de um programa Java sem modificações nas configurações *default* da máquina virtual e gravação de log. Sem modificações no JIT.
Java MaxHeapSize	`java -verbose:gc -Xloggc:java.log -Xms512m -Xmx1g ...`	Execução otimizada do Java com o tamanho da memória *heap* variando de 512MB a 1GB. Sem modificações no JIT.

Scala Normal	JAVA_OPTS="-verbose:gc -Xloggc:scala.log" scala ...	Execução de um programa em Scala sem modificações nas configurações *default* da máquina virtual
Scala MaxHeapSize	JAVA_OPTS="-verbose:gc -Xloggc:scala.log -Xms512g -Xmx1g" scala	Execução otimizada do Scala com o tamanho da memória *heap* variando de 512MB a 1GB
Python	./pingpong.pyc $h $j	O programa Python foi inicialmente compilado e depois executado.
Ruby	ruby programa.rb datasize rep	Execução normal de um programa Ruby

Os códigos-fontes e dados dos testes encontram-se disponíveis para *download* no site do projeto[2]. Os programas foram desenvolvidos de modo a haver honestidade nos critérios de comparação, utilizando os melhores recursos de programação concorrente de cada linguagem **[28]** e havendo semelhança estrutural entre eles e o código desenvolvido para $_{oo}$*Erlang*. Os gráficos dos experimentos estão em escala logarítmica.

5.3. Teste PingPing

O *benchmark* PingPing mostra uma clara superioridade do Erlang em relação a Java e Scala. $_{oo}$*Erlang* herda o desempenho do Erlang e é mais eficiente que Java e Scala.

5.3.1. PingPing com mensagens de 5kB

O Gráfico 5.1 apresenta os resultados para mensagens de tamanho 5kB. Como pode ser observado, até alcançar 100 mil repetições, todas as linguagens exibem o mesmo desempenho. O desempenho do Python e Ruby cai drasticamente após 1 milhão de repetições.

[2]https://sites.google.com/site/ooerlang1/

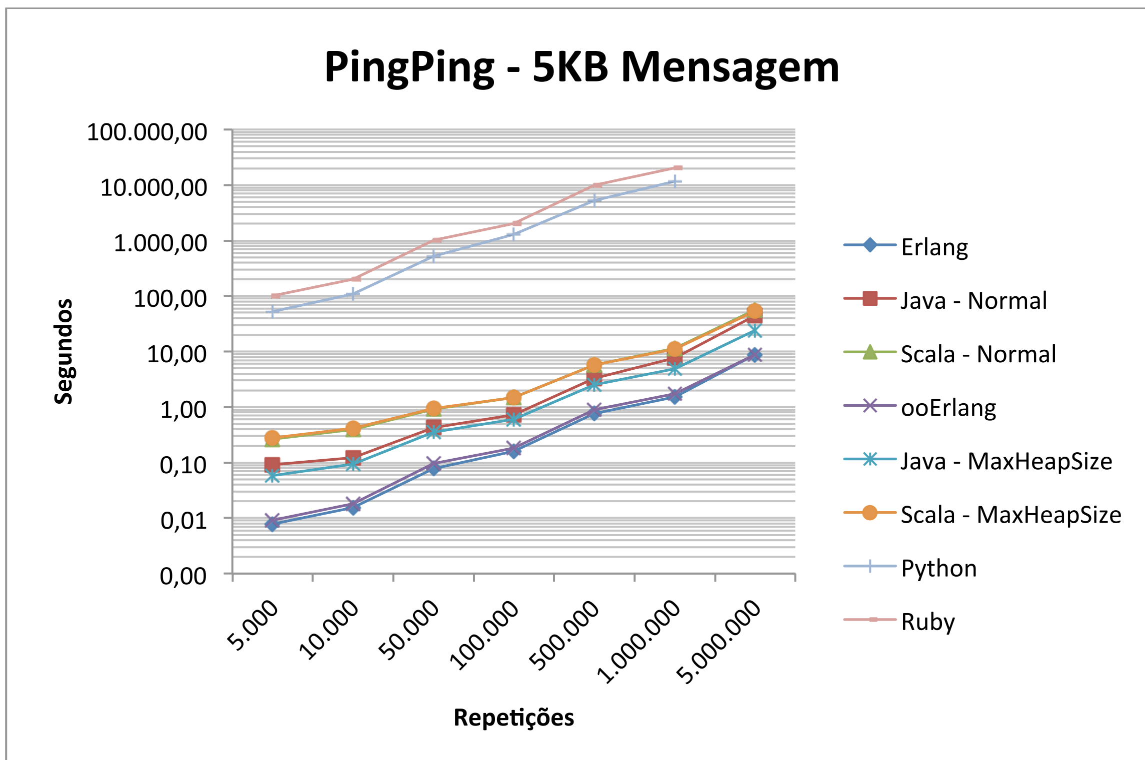


Gráfico 5.1: PingPing com mensagens de 5kB

5.3.2. PingPing com mensagens de 10kB

O Gráfico 5.2 apresenta os resultados para mensagens de tamanho 10kB. Como pode ser observado, até alcançar 100 mil repetições, todas as linguagens exibem o mesmo desempenho. O desempenho de Python e Ruby cai bastante após 100 mil repetições.

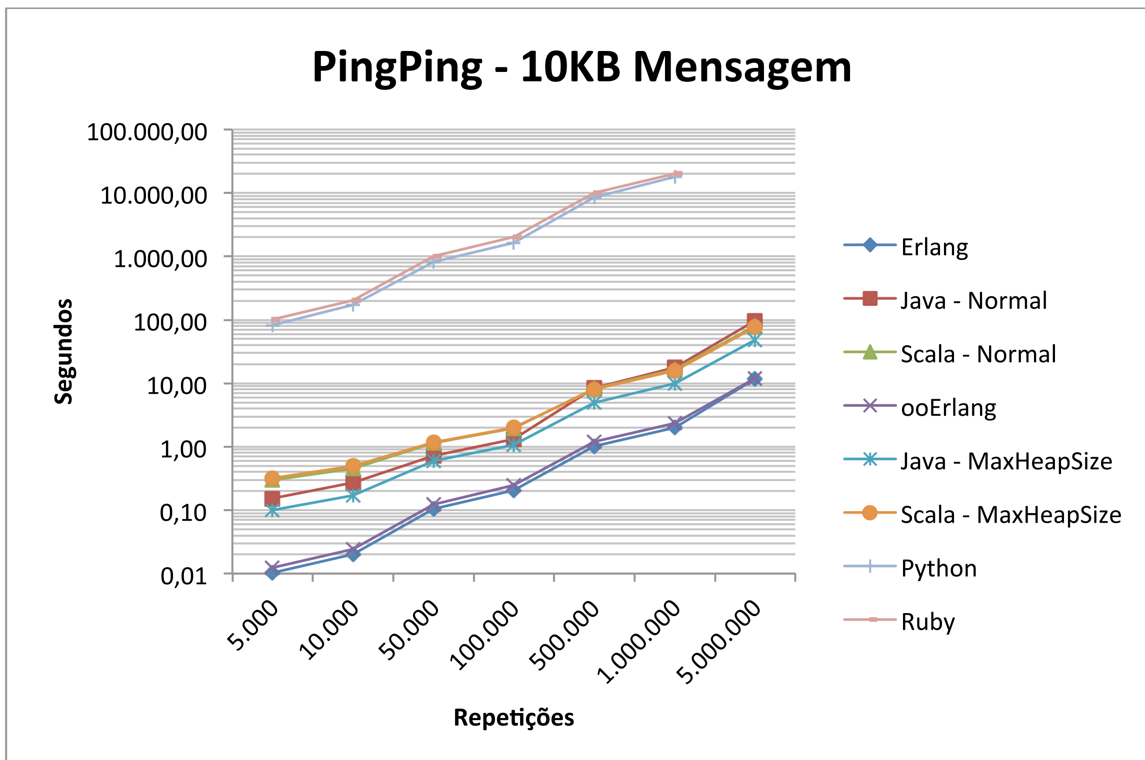


Gráfico 5.2: PingPing com mensagens de 10kB

5.3.3. PingPing com mensagens de 50kB

O Gráfico 5.3 apresenta os resultados para mensagens de tamanho 50kB. O desempenho das linguagens continua seguindo o mesmo padrão.

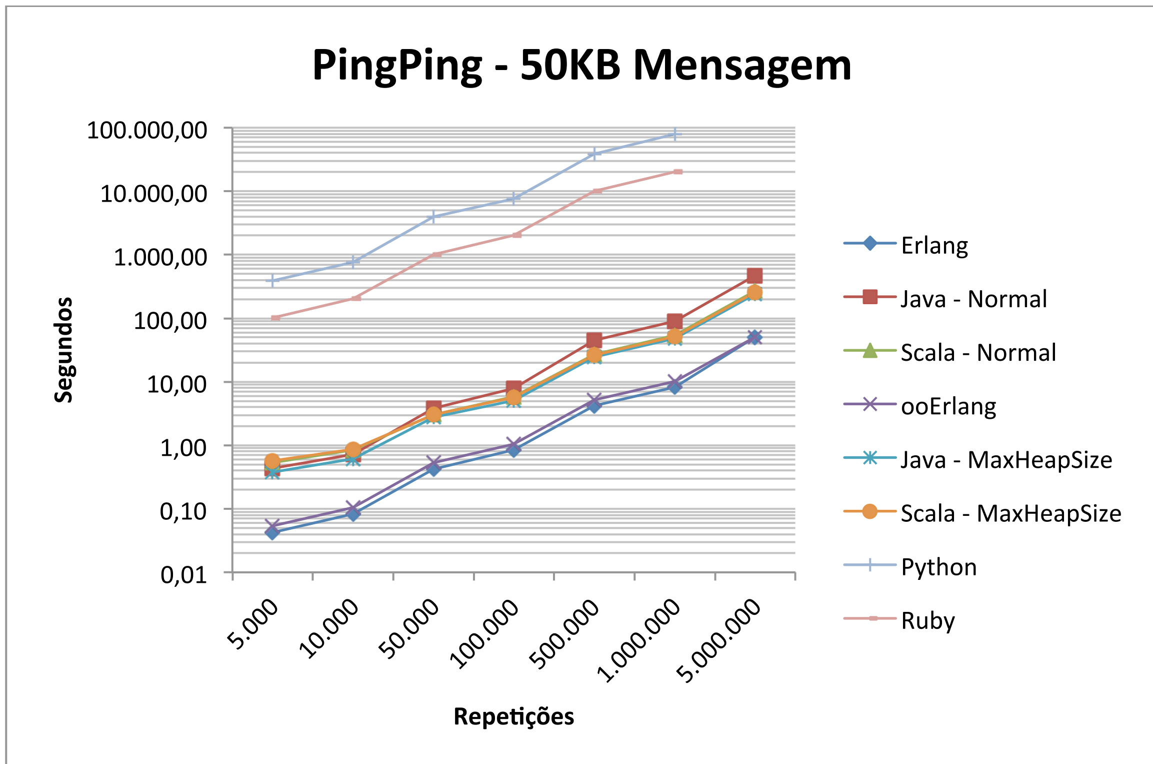


Gráfico 5.3: PingPing com mensagens de 50kB

5.3.4. PingPing com mensagens de 100kB

O Gráfico 5.4 apresenta os resultados para mensagens de tamanho 100kB. O padrão de desempenho das linguagens permanece o mesmo.

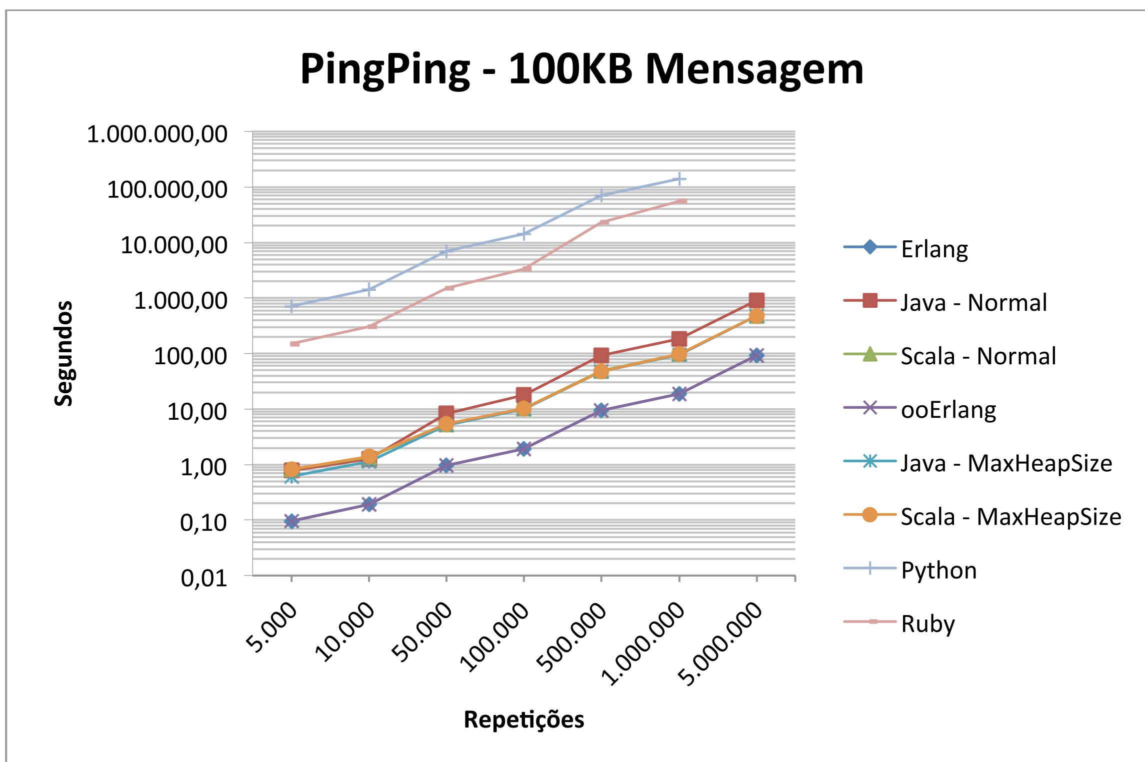


Gráfico 5.4 : PingPing com mensagens de 100kB

5.4. Teste PingPong

Assim como no PingPing, o *benchmark* PingPong mostra uma clara superioridade do Erlang em relação as outras linguagens. $_{oo}$*Erlang* herda o desempenho do Erlang.

5.4.1. PingPong com mensagens de 5kB

O Gráfico 5.17 apresenta os resultados para mensagens de tamanho 5kB. O resultado do teste segue o mesmo padrão do PingPing.

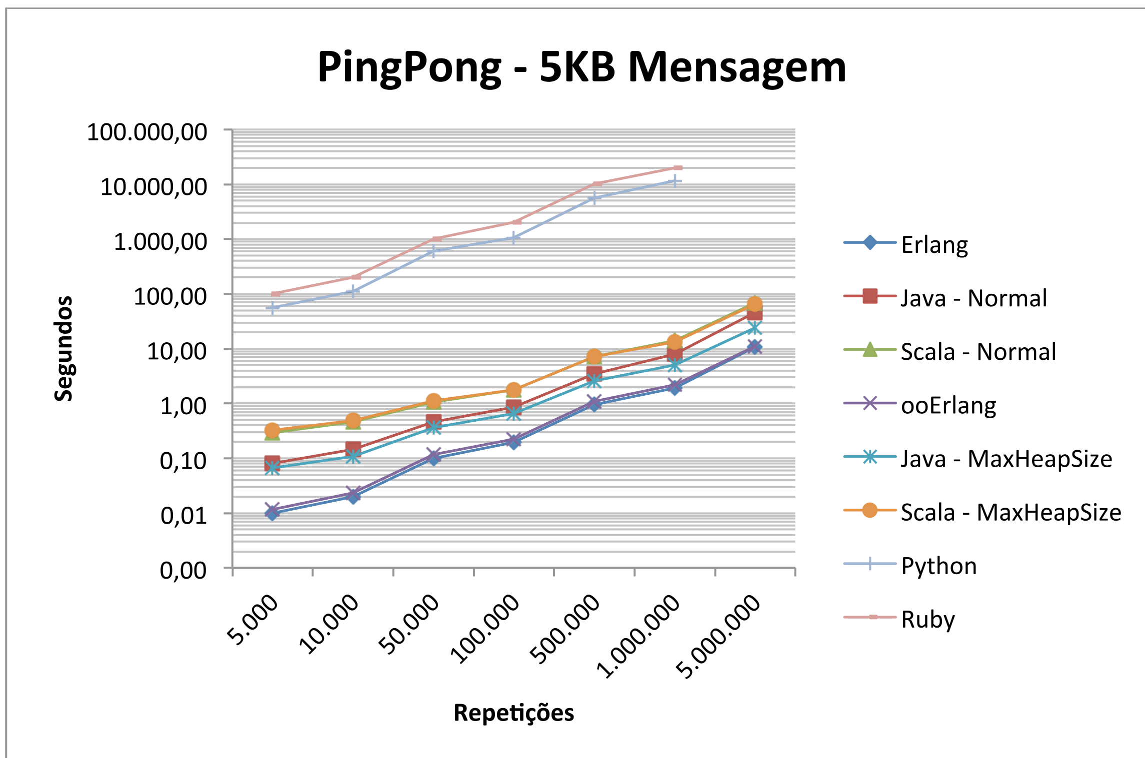


Gráfico 5.5 :PingPong com mensagens de 5kB

5.4.2. PingPong com mensagens de 10kB

O Gráfico 5.6 apresenta os resultados para mensagens de tamanho 10kB. Os resultados seguem, como esperado, o padrão do PingPing.

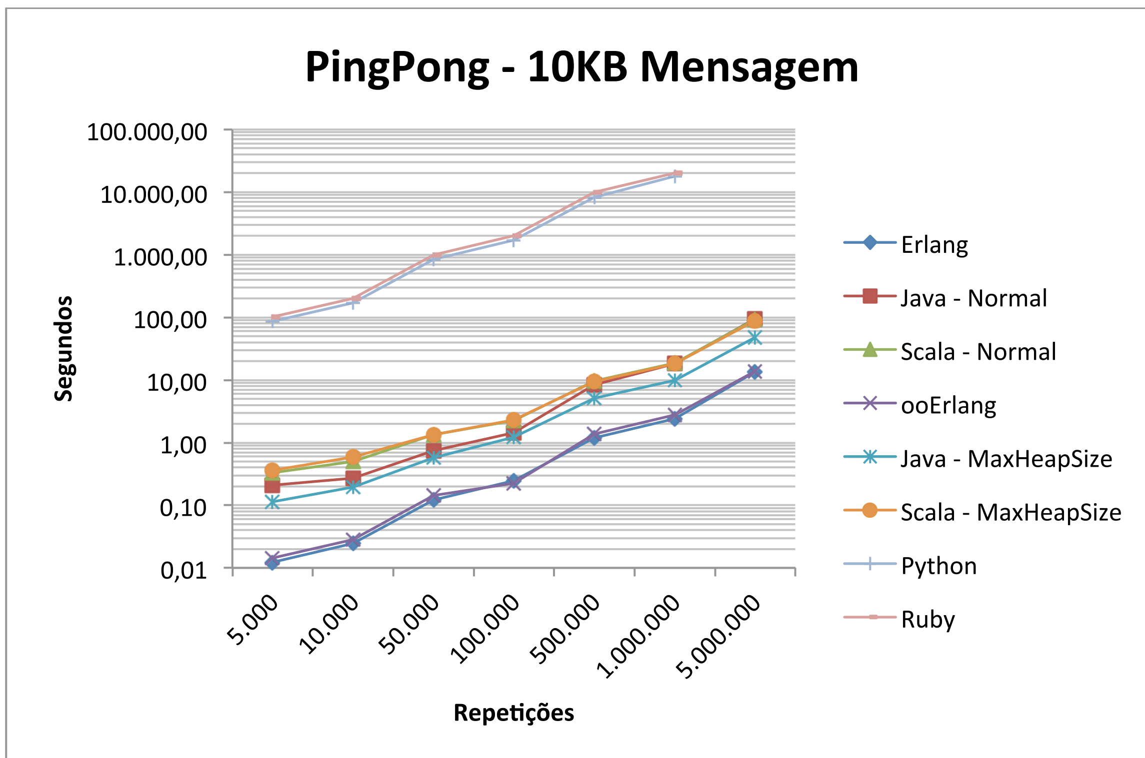


Gráfico 5.6**:** PingPong com mensagens de 10 kB

5.4.3. PingPong com mensagens de 50kB

O Gráfico 5.7 apresenta os resultados para mensagens de tamanho 50kB. O desempenho de Python e Ruby continua deixando muito a desejar comparado ao desempenho das outras linguagens.

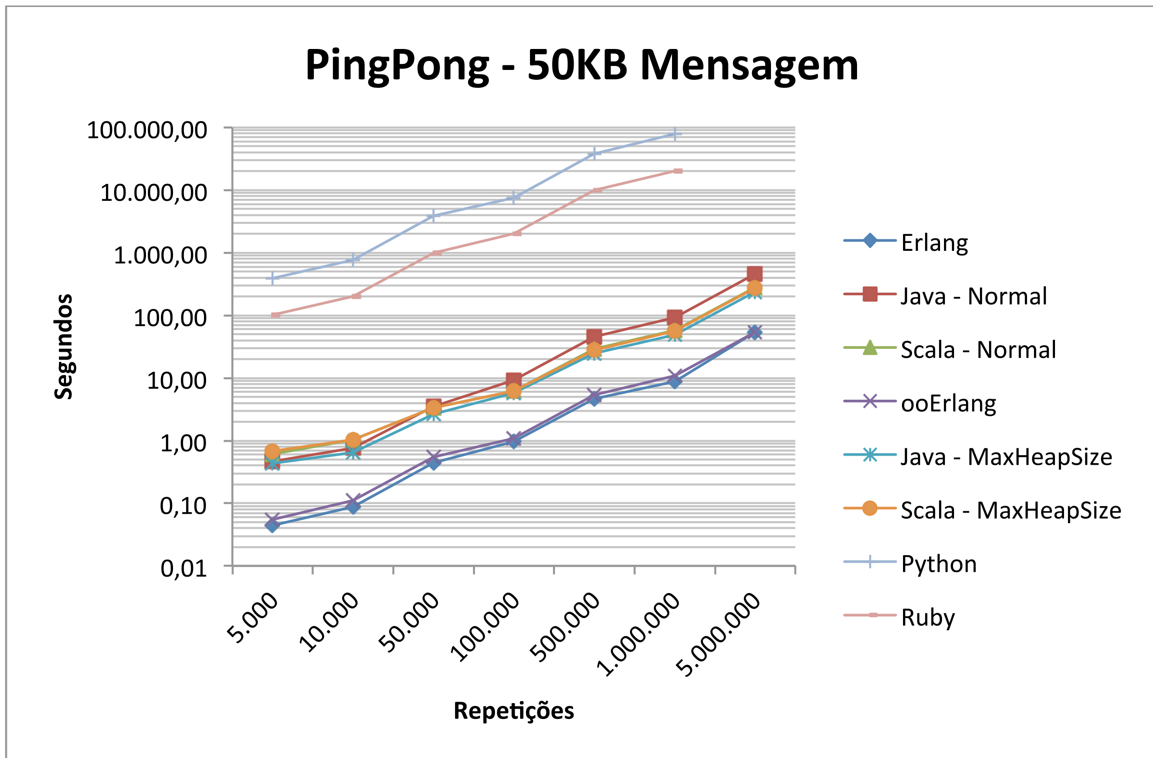


Gráfico 5.7: PingPong com mensagens de 50kB

5.4.4. PingPong com mensagens de 100kB

O Gráfico 5.8 apresenta os resultados para mensagens de tamanho 100kB. Python e Ruby continuam muito mais lentos que as demais linguagens.

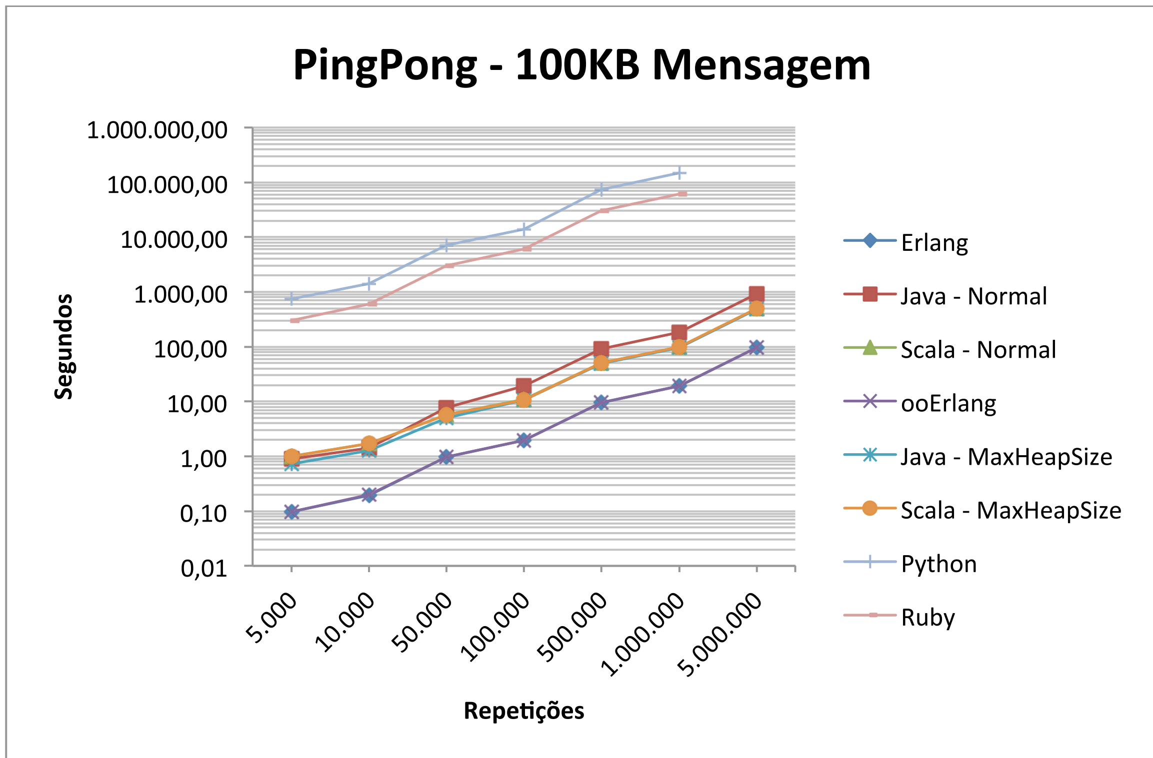


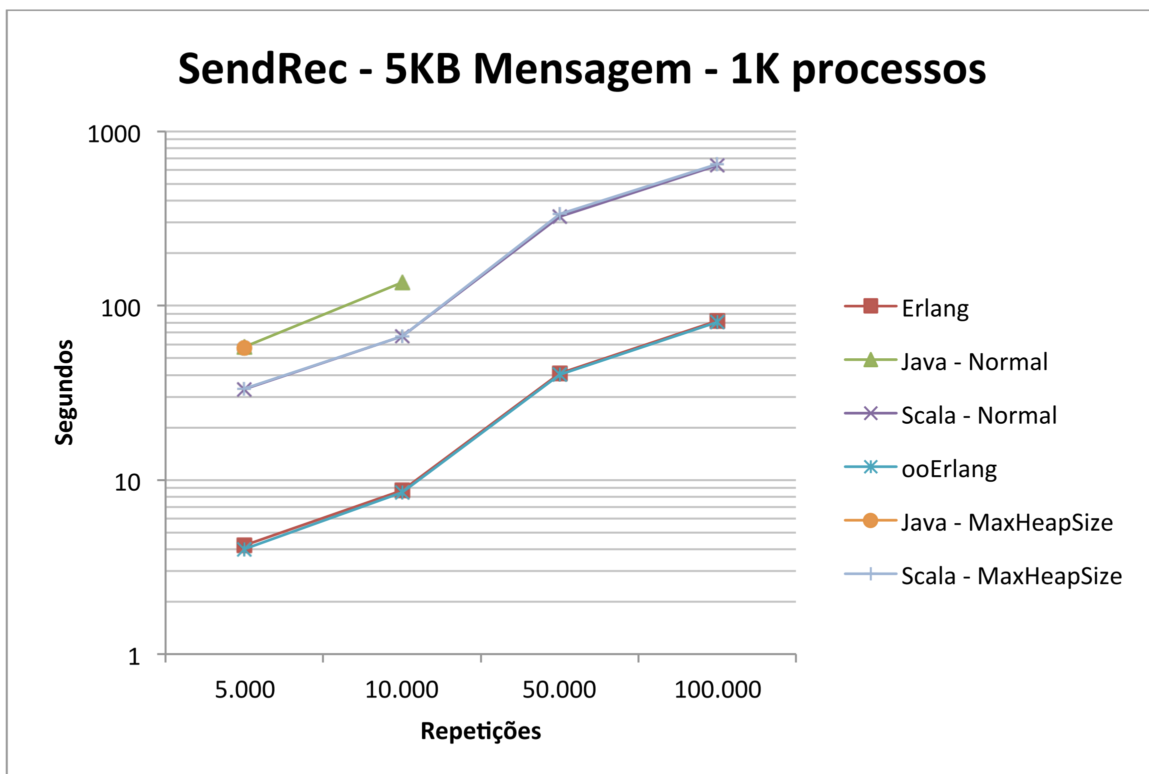
Gráfico 5.8 : PingPong com mensagens de 100kB

5.5. Teste SendRec– Threadring

Os testes para o *benchmark* SendRec Threadring mostram uma diferença maior de desempenho entre as linguagens Erlang, Java e Scala e a extensão $_{oo}$*Erlang*. A linguagem Python não conseguiu criar os 1000 processos para iniciar o primeiro teste SendRec e portanto foi excluída deste e dos demais testes. Ruby não completou o primeiro teste após uma semana funcionando e por isso foi excluída.

5.5.1. Testes com 1000 processos

O *benchmark* com mensagens de tamanho 5 kB, 10kB e 50kB enviadas através de um anel de 1000 processos é mostrado no Gráfico 5.9. Pode ser observado que Java MaxHeapSize "quebra" após 5 mil voltas no anel, pois o programa interrompe sua execução mesmo a máquina não tendo alcançado seu limite. O mesmo acontece com Java Normal após 10 mil voltas. A JVM aparentemente não consegue lidar com essa carga de processos se comunicando. Scala tem um desempenho moderado até 50 mil voltas e depois cai drasticamente. Erlang e $_{oo}$*Erlang* mostram um desempenho bom até 50 mil voltas, mas que degrada linearmente.



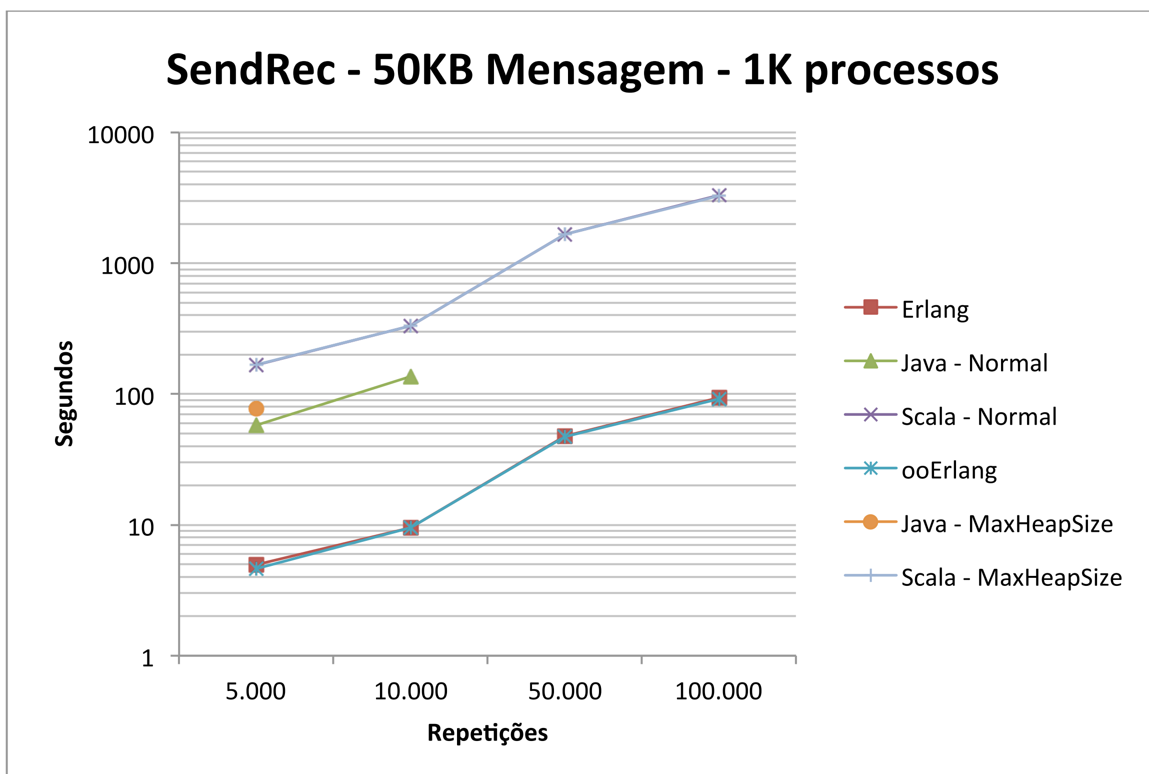
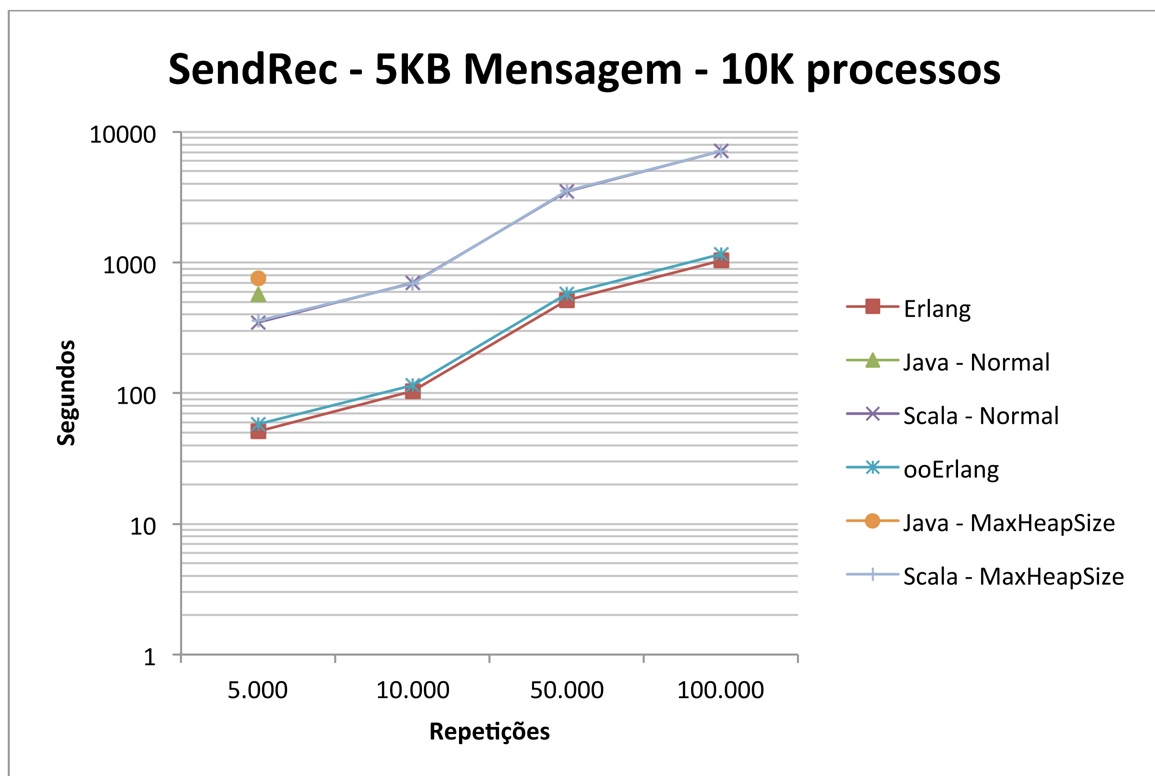


Gráfico 5.9: SendRec de 1000 processos

5.5.2. Testes com 10 mil processos

O *benchmark* com mensagens enviadas através de um anel de 10 mil processos é mostrado no Gráfico 5.10. Pode ser observado que Java não consegue passar de 5.000 voltas no anel e não completa o teste. Scala consegue completar o teste, mas a partir de 10 mil voltas, seu desempenho começa a degradar. Erlang e $_{oo}$*Erlang* conseguem completar o teste mas a partir de 10 mil voltas, seu desempenho vai caindo linearmente em uma taxa menor que Scala.



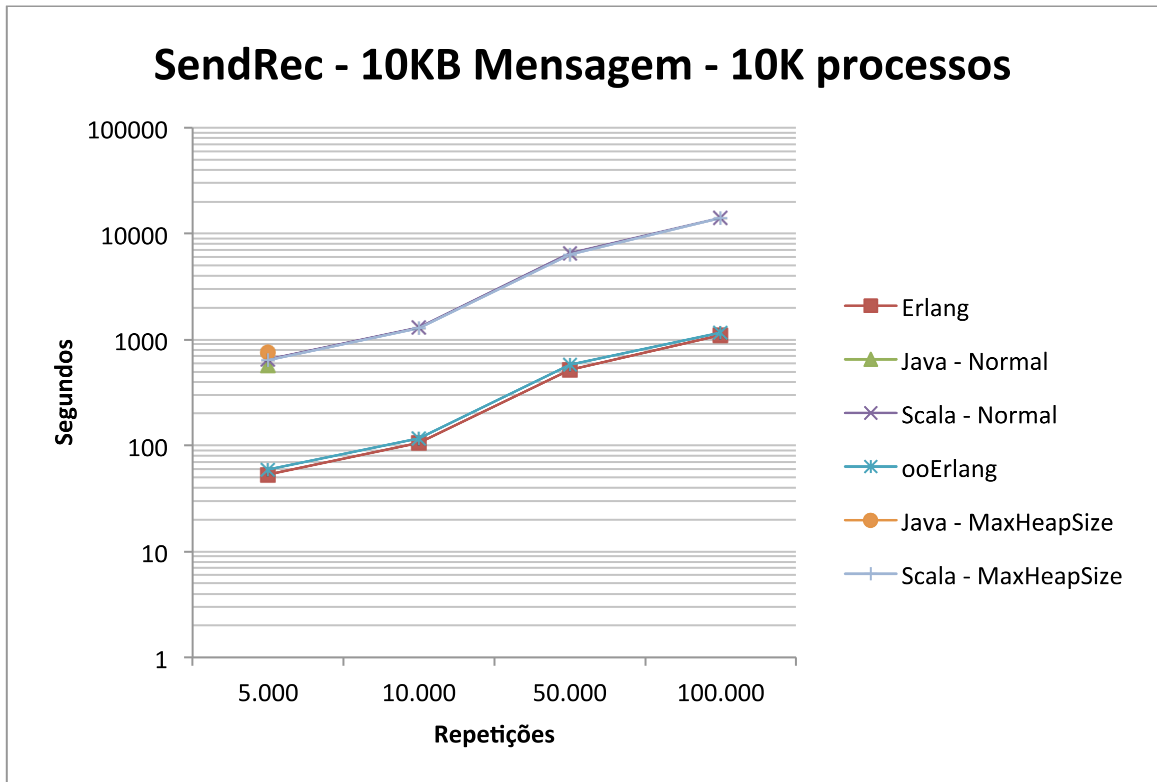
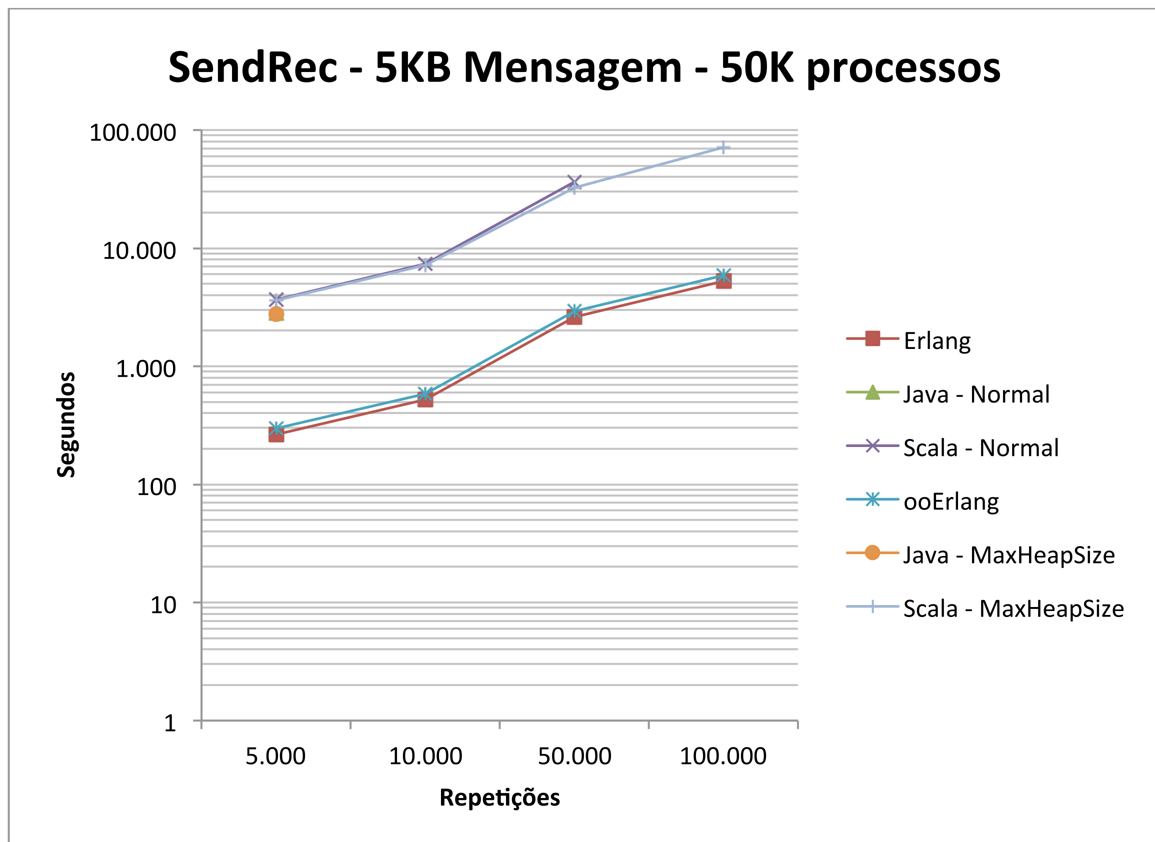
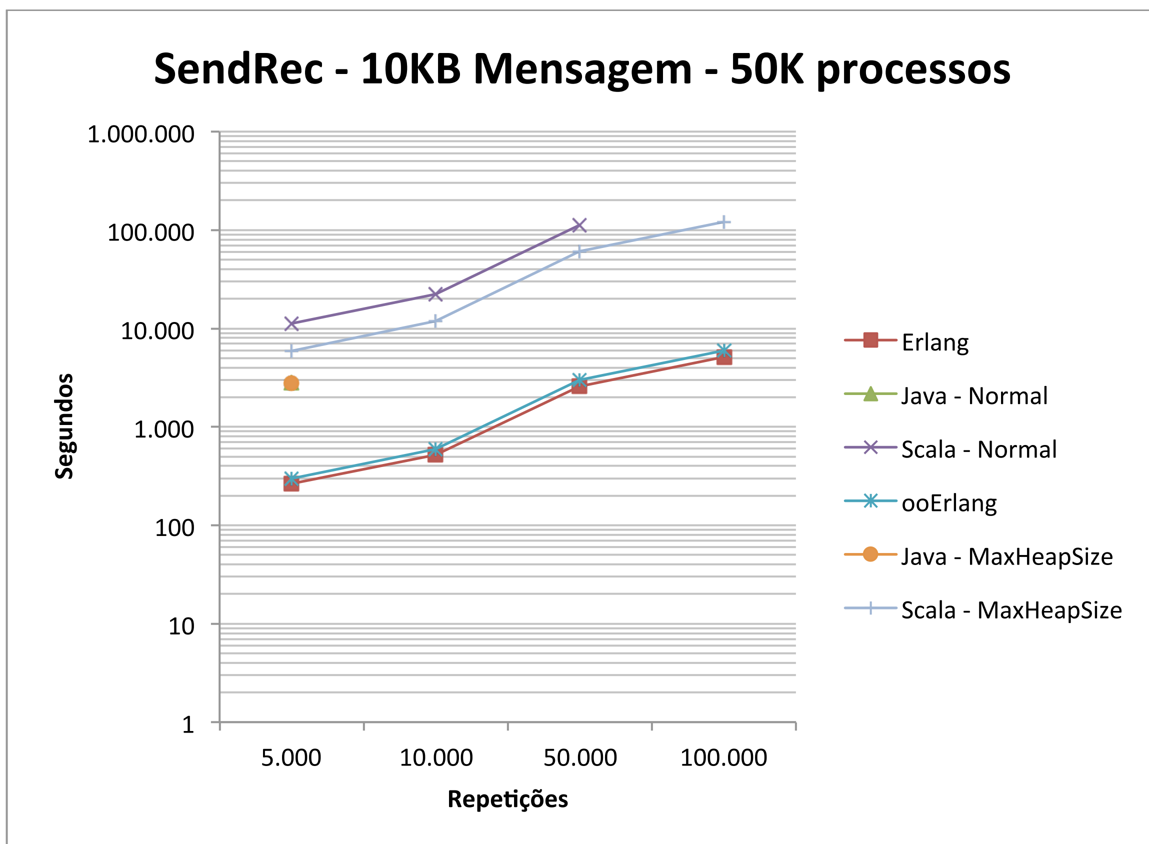


Gráfico 5.10 : SendRec com 10.000 processos

5.5.3. Testes com 50 mil processos

O *benchmark* com mensagens de tamanho 5 kB, 10kB e 50kB enviadas através de um anel de 50 mil processos é mostrado no Gráfico 5.11. Observa-se que Java não consegue passar de 5.000 voltas no anel e não completa o teste. Scala-normal e Scala-MaxHeapSize conseguem completar o teste, mas, a partir de 10 mil voltas, seus desempenhos começam a degradar. Scala-Normal para de funcionar em 50 mil voltas.





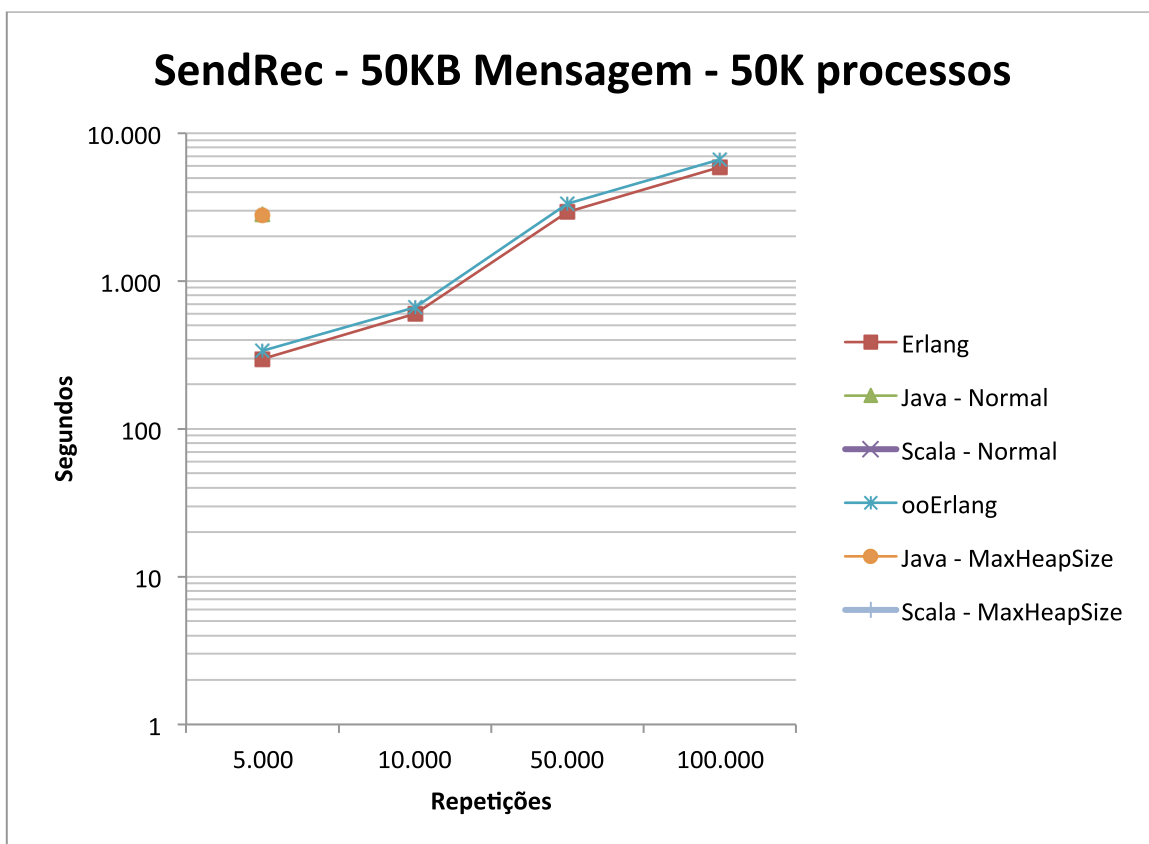


Gráfico 5.11 : SendRec com 50.000 processos

Com mensagens de 50kB somente Erlang e $_{oo}$*Erlang* conseguem fechar o teste. Todas as outras falham.

5.6. Conclusões

Este capítulo mostra o desempenho superior do Erlang e $_{oo}$*Erlang* em relação a Ruby, Python, Java e Scala para tarefas concorrentes em todos os testes. Nos testes de Pingping e Pingpong, Ruby, Python, Java e Scala completaram o *benchmark*, apesar do desempenho muito ruim de Python e Ruby. No SendRec, Python não conseguiu criar a quantidade de processos necessários para iniciar o teste. Ruby passou uma semana processando e não conseguiu finalizar nem o primeiro teste. Ainda no SendRec, mesmo modificando parâmetros de uso da máquina virtual, as linguagens Java e Scala não conseguiram completar os testes. Essas linguagens possuem suporte a concorrência, mas não o suficiente para criação e uso de milhares de processos. $_{oo}$*Erlang* mostrou-se com um desempenho semelhante ao Erlang em todos os testes. Com isso, pode-se afirmar que o suporte a orientação a objetos não afetou negativamente a linguagem, ou seja, o a sobrecarga dos objetos criados e manipulados por $_{oo}$*Erlang* é baixa.

6. CONCLUSÕES E TRABALHOS FUTUROS

Stay hungry, stay foolish.

Steve Jobs
Discurso na graduação da Universidade de
Stanford em 12 de Junho de 2005.

A orientação a objetos facilita a modelagem e criação de programas, pois aproxima o desenvolvedor do "mundo real" por meio do uso de uma linguagem associada ao dia-a-dia do usuário final. Segundo **[25]**, a orientação a objetos é um modo natural de pensar sobre o mundo e de escrever programas de computador.

Erlang é uma linguagem de programação criada pela Ericson originalmente para a gerência de centrais telefônicas **[30][31]**. Ela se tornou uma opção para criação de aplicações Web 2.0 devido a sua habilidade de suportar concorrência massiva, alta taxa de vazão, tolerância a falhas e integração com outras linguagens de programação **[57]**. Erlang, porém, é uma linguagem funcional, e isso afasta uma parcela de programadores que tem dificuldade de modelar e programar sistemas por meio desse paradigma.

$_{oo}$*Erlang* é uma extensão que acrescenta o suporte a orientação a objetos à linguagem Erlang. Programadores acostumados a linguagens orientadas a objetos podem utilizar mecanismos como classes, objetos, interfaces, herança, polimorfismo e as boas práticas dos Padrões de Projeto por

meio de uma sintaxe simples e expressiva. Desenvolvedores que trabalham com Erlang podem facilmente reconhecer a sintaxe, a semântica e o desempenho dessa linguagem na extensão. $_{oo}$*Erlang* visa aliar o desempenho do Erlang com a facilidade de modelar domínios de problema da orientação a objetos. Fatores podem ser destacados para adoção do $_{oo}$*Erlang*:

- **Legibilidade.** A sintaxe facilita a escrita e leitura de programas orientados a objetos através de uma sintaxe simples e expressiva, parecida com a linguagem Java;
- **Aumento do suporte a abstração.** A inclusão de classes, objetos e interfaces facilita a modelagem e criação de programas de um determinado domínio de problema;
- **Desempenho.** $_{O}$*oErlang* possui um desempenho muito semelhante ao Erlang para o desenvolvimento de aplicações onde o Erlang é sempre mais rápido que as linguagens Java e Scala;
- **Código-fonte aberto.** $_{O}$*oErlang* é licenciado como Código Aberto. Isso permite que outros programadores possam colaborar com o desenvolvimento da extensão e usar livremente em seus projetos de código aberto ou não. Ele pode ser obtido e testado no sítio do projeto **[74]**.

Esta tese apresenta uma análise comparativa do desempenho de Ruby, Python, Java, Scala, Erlang e $_{oo}$*Erlang* em tarefas com troca intensiva de mensagens como as usadas em aplicações Web 2.0,por meio de três programas de teste da Intel. Java foi desenvolvida para tarefas de média granularidade, baixa troca de mensagens e memória compartilhada. Erlang foi projetado para tarefas onde há uma intensa troca de mensagens e que não precisem de memória compartilhada, como uma central telefônica. Scala é uma linguagem de propósito geral que, de acordo com seus desenvolvedores, foi projetada para expressar padrões comuns de programação de uma maneira concisa e elegante. Eles alegam que Scala integra funcionalidades das linguagens orientadas a objetos e das linguagens funcionais, habilitando os programadores a serem mais produtivos.

Os experimentos do Capítulo 5 desta tese mostram que não importa se a plataforma tiver muita memória ou vários processadores, Scala e Java não conseguem atender a demanda dos testes e nem usar todo o poder da máquina. Java e Scala pararam de funcionar quando 10 mil processos foram criados. Por outro lado, o desempenho de Erlang e $_{oo}$*Erlang* são limitados apenas pela máquina hospedeira. Essas linguagens atendem ao crescimento da demanda usando todos os recursos disponíveis. Serviços Web desenvolvidos em Java e Scala precisam monitorar a quantidade de *threads* criados e se um determinado limiar de tráfego foi alcançado. Esse limiar é imposto pela linguagem e pelos *frameworks* utilizados, não pela arquitetura da máquina.

6.1. Trabalhos futuros

O resultado esperado com a versão final desta tese é a ampla utilização da extensão por programadores Erlang e por programadores acostumados a linguagens orientadas a objetos. Portanto, o projeto $_{oo}$*Erlang* pode ser continuado de modo a incentivar essa adoção:

- **Inclusão de "*namespaces*".** Os *namespaces* são um mecanismo adotado pelas linguagens modernas para organização de código-fonte através de uma estrutura hierarquizada de arquivos. Em Java, por exemplo, existem os *packages* **[12]**. Em Erlang não existem *namespaces*. Essa adição ajudaria os programadores $_{oo}$*Erlang* e Erlang na organização e manutenção do código-fonte. Em $_{oo}$*Erlang*, seriam *packages*;
- **Otimização do *garbage collector* da máquina virtual do Erlang para processos e objetos.** A máquina virtual do Erlang é otimizada para reciclar processos **[69]**. Poderia ser estudada a inclusão de um novo algoritmo de *garbage collector* **[33]**como o apresentado em **[68],** na VM, e analisado qual seria o impacto no desempenho junto a criação/limpeza de processos e objetos do Erlang e do $_{oo}$*Erlang*.
- **Teste de usabilidade com a extensão $_{oo}$*Erlang*.** Para verificar com mais precisão a legibilidade da extensão, seria interessante fazer um teste de usabilidade com até cinco programadores. Eles seriam filmados durante uma sessão de programação na extensão e responderiam um questionário.

7. PUBLICAÇÕES

A relação das publicações associadas a esta tese:

Silva Jr., J. and Lins, R.D. 2012, *$_{oo}$Erlang*: Another Object Oriented Extension to Erlang. Proceedings of the 11th ACM SIGPLAN Workshop on Erlang (Erlang '12), pp 65-66. ACM, New York. doi: 10.1145/2364489.2364502

Silva Jr, J ; Lins, R; Aquino, D. H. B. . ooErlang: Object-oriented Erlang. In: IADIS International Conference on Applied Computing, 2012, Madrid. International Conference on Applied Computing 2012, 2012. p. 155-162.

Silva Jr, J. M.; Lins, Rafael Dueire ; Santos, L. M. . Assessing the Performance of Java and Erlang in Web 2.0 Applications. In: IADIS International Conference WWW/Internet 2012, 2012, Madrid. IADIS International Conference WWW/Internet 2012, 2012. p. 315-322.

Silva Jr., J. M. ; Lins, R. D. ; Santos, L.M. . Comparing the Performance of Java, Erlang and Scala in Web 2.0 Applications. IADIS International Journal on WWWInternet, v. 10, p. 121-137, 2012.

8. REFERÊNCIAS

[1] Linguagem de programação Erlang - http://www.erlang.org/, acessado em março de 2013;

[2] Linguagem de programação Java - http://www.oracle.com/technetwork/java/javase/overview/index.html, acessado em março de 2013;

[3] Linguagem de programação PHP -http://www.php.net/, acessado em março de 2013;

[4] Linguagem de Programação Ruby - http://www.ruby-lang.org, acessado em março de 2013;

[5] Linguagem de programação Python -http://www.python.org, acessado em março de 2013;

[6] Linguagem de programação Scala -http://scala-lang.org/, acessado em março de 2013;

[7] Linguagem de programação Clojure -http://clojure.org/, acessado em março de 2013;

[8] Linguagem de programação Efene - http://marianoguerra.com.ar/efene/, acessado em março de 2013;

[9] Linguagem de programação Elixir - http://elixir-lang.org/, acessado em março de 2013;

[10] Linguagem de programação Java: Máquina Virtual Java SE Hotspot - http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html, acessado em fevereiro de 2013;

[11] Linguagem de programação Java: Java Enterprise Edition - http://www.oracle.com/technetwork/java/javaee/overview/index.html, acessado em março de 2013;

[12] Linguagem de programação Java: Packages - http://docs.oracle.com/javase/tutorial/java/package/managingfiles.htmll, acessado em março de 2013;

[13] Java Especificação - http://docs.oracle.com/javase/specs/jls/se5.0/html/j3TOC.html, acessado em março de 2013;

[14] Erlang Especificação - www.erlang.org/download/erl_spec47.ps.gz, acessado em fevereiro de 2013;

[15] Erlang AST - http://www.erlang.org/doc/apps/erts/absform.html, acessado em março de 2013;

[16] Erlang YECC- http://www.erlang.org/doc/man/yecc.html, acessado em março de 2013;

[17] Ferramenta Visual de EBNF - http://railroad.my28msec.com/rr/ui, acessado em março de 2013;

[18] TIOBE Index - http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, acessado em março de 2013;

[19] Intel MPI Benchmark, http://software.intel.com/en-us/articles/intel-mpi-benchmarks/, acessado em março de 2013;

[20] Computer Language Benchmark Games -http://shootout.alioth.debian.org/, acessado em março de 2013;

[21] Aho, A e Ravi, S. ,Compiladores: princípios, técnicas e ferramentas, *Pearson Addison-Wesley*, 2ª edição, 2008;

[22] Louden, K. Compiladores – Princípios e Práticas, *Thomson*, 2004;

[23] Sebesta, R. Conceitos de Linguagens de Programação, 9ª edição, *Bookman*, 2011

[24] Linguagens de Programação – Princípios e Paradigmas, *McGrall-Hill*, 2009;

[25] Deitel, P e Deitel, H. Java – Como Programar, 8ª edição, *Prentice-Hall*, 2010;

[26] Freeman, E. Freeman, E. Use a cabeça – Padrões de Projeto, 2ª edição, *Alta Books*, 2009

[27] Silveira, P. Et al. Introdução à Arquitetura e Design de Software, *Elsevier*, 2012

[28] Tanembaum, A. e Van Steen, M. Sistemas Distribuídos – Princípios e Paradigmas, *Prentice-Hall*, 2ª edição, 2008;

[29] Dantas, M. Computação Distribuída de Alto Desempenho: Redes, Clusters e Grids Computacionais, *Axcel Books*, 2005

[30] Cesarini, F. e Thomson, S. Erlang Programming, *O'Reilly*, 2009;

[31] Armstrong, J. Programming Erlang – Software for a concurrent world, *Pragmatic Bookshelf*, 2007;

[32] Learn you some Erlang for great good - http://learnyousomeerlang.com/content, acessado em fevereiro de 2013

[33] Jones, R. e Lins, R. Garbage Collection: Algorithms for Dynamic Memory Management, *Wiley*, 1996;

[34] Goetz, B. *et al*. Java Concurrency in Practice. *Addison-Wesley,* 2009;

[35] Bloch, J. Effective Java. 2ª edição. *Addison-Wesley*, 2008;

[36] Fowler, M. Domain-Specific Language. *Addison-Wesley*, 2010;

[37] Fowler, M. UML Distilled: a brief guide to the standard object modeling language, 2ª edição. *Addison-Wesley*, 2010;

[38] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: elements of reusable object-oriented software. *Addison-Wesley*, 1994;

[39] Larman, C. Applying UML and Patterns: an introduction of object-oriented analysis and design and interactive development, 3ª edição. *Prentice Hall*, 2004

[40] SOAP Specifications -http://www.w3.org/TR/soap/, acessado em março de 2013;

[41] REST : Representational State Transfer - http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm, acessado em março de 2013;

[42] Node.js -http://nodejs.org/, acessado em março de 2013;

[43] Ruby on Rails, 2012. Ruby on Rails, http://www.rubyonrails.org, acessado em março de 2013;

[44] What is Linux: An Overview Of The Linux Operating System - http://www.linux.com/learn/new-user-guides/376-linux-is-everywhere-an-overview-of-the-linux-operating-system, acessado em março de 2013;

[45] Apache HTTP Server Project - http://httpd.apache.org/, acessado em março de 2013;

[46] MySQL - http://www.mysql.com/why-mysql/, acessado em março de 2013;

[47] Celebrating #Twitter7 -https://blog.twitter.com/2013/celebrating-twitter7, acessado em junho de 2013;

[48] Why Scala? - http://www.web2expo.com/webexsf2009/public/schedule/detail/6110, acessado em março de 2013;

[49] About Facebook -http://newsroom.fb.com/content/default.aspx?NewsAreaId=22, acessado em março de 2013;

[50] Chat reaches 1 billion messages sent per day - https://www.facebook.com/note.php?note_id=91351698919, acessado em março de 2013;

[51] HipHop for PHP: Move Fast - https://developers.facebook.com/blog/post/2010/02/02/hiphop-for-php--move-fast/, acessado em março de 2013;

[52] LinkedIn is 99% Java but 100% Mac - http://blog.linkedin.com/2008/06/24/linkedin-is-99-java-but-100-mac/, acessado em março de 2013;

[53] One Billion People on Facebook-http://newsroom.fb.com/News/457/One-Billion-People-on-Facebook, acessado em março de 2013;

[54] Erlang OTP- http://www.erlang.org/doc/design_principles/des_princ.html, acessado em março de 2013;

[55] Piro, C. and Letuchy, E. Functional programming at Facebook. *Proceedings of the 2009 Video Workshop on Commercial Users of Functional Programming: Functional Programming as a Means, Not an End (CUFP '09),ACM,*2009;

[56] Lee, E.A. The Problem with Threads. *Computer, vol. 39 issue 5, pp. 33-42. IEEE Computer Society Press,* 2006;

[57] Armstrong, J. Erlang. *Communications of ACM, vol. 53 issue 9, pp. 68-75. ACM, New York,* 2010

[58] Agha, A.G., Mason I.A., Smith, S.F., Talcott, C.L. A foundation for actor computation. *Journal of Functional Programming, vol. 7, pp. 1-72. Cambridge University Press*, 1997

[59] Schäfer, J. and Poetzsch-Heffter, A. JCoBox: generalizing active objects to concurrent components. *Proceedings of the 24th European conference on Object-oriented programming. Theo D'Hondt (ed.). Springer-Verlag, Berlin, Heidelberg*, 2010;

[60] Santoro, C. and Stefano,A. 2004. Designing Collaborative Agents with eXAT. In Proceedings of the 13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '04). pp15-20. IEEE Computer Society, Washington, DC, USA.

[61] Carlsson, R. Parameterized modules in Erlang. *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang (ERLANG '03). pp 29-35. ACM, New York, NY, USA,* 2003.

[62] Feher, G. and Bekes. ECT: an object-oriented extension to Erlang. *Proceedings of the 8th ACM SIGPLAN workshop on ERLANG (ERLANG '09). pp 51-62. ACM, New York, NY, USA*, 2003.

[63] Berry, G., Ramesh and S., Shyamasundar, R. K. Communicating reactive processes. *Proceedings of the 20th Symposium on Principles of Programming Languages (POPL '93), pp. 85-98. ACM, New York,* 1993;

[64] Vinoski, S. Welcome to "The Functional Web" - Steve Vinoski, Welcome to "The Functional Web". *IEEE Internet Computing, vol. 13, no. 2, pp. 104,102-103, doi:10.1109/MIC.2009.49*, Março-Abril 2009;

[65] Vinoski, S., Concurrency and Message Passing in Erlang. Computing in Science & Engineering , *vol.14, no.6, pp.24,34,*
doi: 10.1109/MCSE.2012.67, Nov.-Dec. 2012

[66] Bryson, D.; Vinoski, S. Build Your Next Web Application with Erlang. *Internet Computing, IEEE , vol.13, no.4, pp.93,96,* Julho-Agosto, 2009
doi: 10.1109/MIC.2009.74

[67] Venners, B. Inheritance versus Composition: Whitch one you should you choose? http://www.javaworld.com/jw-11-1998/jw-11-techniques.html, acessado em março 2013

[68] Formiga, A. Algoritmos para contagem de referencias cíclicas, Tese de Doutorado, Universidade Federal de Pernambuco, 2011;

[69] Erlang Garbage Collector - http://www.erlang.org/faq/academic.html#id55848, acessado em março de 2013.

[70] Silva Jr., J., Lins, R.D., and Santos, L.M. Assessing the Performance of Java and Erlang in Web 2.0 Applications. *WWW and Internet 2012. Madrid. IADIS Press,* 2012

[71] Silva Jr., J. and Lins, R.D. *ooErlang*: Another Object Oriented Extension to Erlang. *Proceedings of the 11th ACM SIGPLAN Workshop on Erlang (Erlang '12), pp 65-66. ACM, New York. doi: 10.1145/2364489.2364502*, 2012;

[72] Silva Jr., J., Lins, R.D., and Aquino, D.A. *ooErlang*: Object Oriented Erlang. *Applied Computing 2012. Madrid. IADIS Press,* 2012;

[73] Silva Jr., J. ; Lins, Rafael Dueire ; Santos, L.M. . Comparing the Performance of Java, Erlang and Scala in Web 2.0 Applications. IADIS International Journal on WWWInternet, v. 10, p. 121-137, 2012

[74] *ooErlang* - https://sites.google.com/site/ooerlang1/, acessado em março 2013;

[75] About LinkedIn – http://press.linkedin.com/about, acessado em março 2013;

[76] Super charges git-deamon - https://github.com/blog/112-supercharged-git-daemon, acessado em março de 2013;

[77] Alan Kay's Definition Of Object-Oriented - http://c2.com/cgi/wiki?AlanKaysDefinitionOfObjectOriented, acessado em março de 2013

[78] SmallTalk - http://en.wikipedia.org/wiki/Smalltalk, acessado em março de 2013

[79] Pinho, G. e Carvalho, H. An object-oriented parallel programming Language for distributed-memory parallel computing plataforms - http://www.sciencedirect.com/science/article/pii/S0167642313000750, acessado em março 2013;

[80] The Message Passing Interface - http://www.mcs.anl.gov/research/projects/mpi/, acessado em março 2013;

[81] Message Passing Interface - http://www.mcs.anl.gov/research/projects/mpi/, acessado em março 2013;

[82] Erjang - https://github.com/trifork/erjang/wiki/, acessado em maio 2013;

[83] Ralph Johnson, Joe Armstrong on the State of OOP - http://www.infoq.com/interviews/johnson-armstrong-oop, acessado em maio 2013