Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Título da monografia

Thiago Gomes Cavalcanti
Vinícius Correa de Almeida

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2015

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Computação — Licenciatura

Coordenador: Prof. Dr. Wilson Henrique Veneziano

Banca examinadora composta por:

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador) — CIC/UnB
Prof. Dr. Professor I — CIC/UnB
Prof. Dr. Professor II — CIC/UnB

CIP — Catalogação Internacional na Publicação

Cavalcanti, Thiago Gomes.

Título da monografia / Thiago Gomes Cavalcanti, Vinícius Correa de Almeida. Brasília : UnB, 2015.
45 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2015.

1. language design, 2. language evolution, 3. refactoring,
4. microrefactoring, 5. java

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Título da monografia

Thiago Gomes Cavalcanti
Vinícius Correa de Almeida

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Prof. Dr. Professor I
CIC/UnB

Prof. Dr. Professor II
CIC/UnB

Prof. Dr. Wilson Henrique Veneziano
Coordenador do Curso de Computação — Licenciatura

Brasília, 31 de março de 2015

Dedicatória

Dedico a....

Agradecimentos

Agradeço a....

Abstract

Resumo.....

Palavras-chave: language design, language evolution, refactoring, microrefactoring, java

Abstract

Abstract.....

Keywords: language design, language evolution, refactoring, microrefactoring, java

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Com o passar do tempo as linguagens de programação evoluem entretanto não sabe-se ao certo como os softwares projetados e implementados há alguns anos acompanharam tais atualizações. Conforme explicado por [1], tal evolução faz com que características obsoletas sejam mantidas e raramente são removidas de uma linguagem o que acarreta em um aumento da complexidade, aprendizagem e da manutenção do software. Isso naturalmente aumenta a dificuldade de desenvolvimento o que resulta em um aumento de dificuldade de aprendizagem de determinada versão já ultrapassada de uma linguagem e faz com que a equipe alterne entre propriedades atuais e antigas as quais passam a ser quase um dialeto da linguagem implicando no aumento de tempo para conceber um projeto e consequentemente gerindo aumento no custo final projeto.

Uma decisão não é tão simples de manter uma porção do código congelado, sem evolução, ao longo projeto devido alguma restrição técnica. O que infelizmente acarreta em uma estagnação de todo um sistema pois não é somente o projeto afetado, mas sim uma toda infraestrutura como compiladores, banco de dados e sistema operacional e que se de alguma forma vierem a ser atualizados com esta porção código estagnado pode ocasionar problemas como uma queda significativa de desempenho ou até mesmo o sistema parar de funcionar. Devido a esses problemas de código não atualizado com as versões com estruturas mais atuais a proposta da realização de refatoração através de ferramenta a serem desenvolvidas que visem atacar esse gargalo deixado por código obsoleto.

A base para tal trabalho será o desenvolvimento de algumas ferramentas que auxiliem em mudanças em um código legado para reduzir suas estruturas obsoletas e com baixa performance. Essas ferramentas tem como base a sua construção em linguagem *Java* com o intuito de tornar o processo de construção mais ágil e posteriormente aberto para o acoplamento de novos módulos. A construção de uma arvore sintática é um passo onde é feito para cada arquivo de código .Java e posteriormente de todos os arquivos .Java contidos em qualquer projeto para posterior análise. Este parser implica em listar todos os arquivos Java e gerar uma Árvore Sintática Abstrata (AST) para que posterirormente haja um percorrimento de blocos de codigo afim de compará-los como estes com a versão atual.

O processo de refatoração tem como motivação a reestruturação de codigo, de forma que o codigo considerado pelo processo, morto,duplicado ou com perca de desempenho

não haja codigo morto, duplicado ou com perca de desempenho em um determinado trecho de codigo. Esse processo não tem como premissa a atualizaçao do codigo para novas estruturas de versões da liguagem mais recentes. Essa nova abordagem de refatoração tem com motivaçao a retirada de codigo obsoleto, devido a novas abordagens e estruturas das novas versões, e com baixo desempenho de sistemas sem prejuizo na sua engenharia e funcionamento.

Devido a esse tipo de refatoração de código visa a evolução do código para a uma mais recente com estruturas onde não haja perca de desempenho devido a mudança e também a atualização do código legado para estruturas modernas. Tais estruturas antigas com a ação dessa proposta de refatoração tendem com o tempo sair das estruturas providas pela linguagem Java como aconteceu com Fortan 90 como visto em [1] e essa abordagem tem o intuito de diminuir a quantidade de estruturas antigas que não fazem mais sentido pois novas estruturas realizam as mesmas funções no interpretador da linguagem Java. As modificações proposta pela ferramenta de refatoração não deverá trazer com que haja perca de desempenho ou aumento da complexibilidade do código portanto deixando como uma sugestão a alteraçao do codigo ou segmento de codigo para o usuário a adesão das propostas de refatoração ou não.

A árvore de sintaxe abstrata (AST) é uma estrutura de dados que representa estruturas de cadeias sintaticas onde é respresentada por um esqueleto semantico da liguagem em questão. É constituída atravez de um framework do ambiente de desenvolvimento integrado chamado Eclipse onde o nome desse framework é chamado de EclispeJDT onde traz métodos implementados pelo próprio framework para o percorrimento e ações na árvore sintática. A ideia é transformar inicialmente qualquer código fonte java em uma árvore sintática e devido a isso, o mapeamento da arvore já construída que é muito conveniente para inspecionar o código fonte de um arquivo ou de um projeto com vários arquivos em diferentes diretórios. Com isso é possível realizar ou sugerir ao usuário modificações no código fonte através desta árvore construída e isto seria referenciado automaticamente no código fonte.

A proposta é criar ferramentas de análise estática para códigos fonte da linguagem Java para que possa-se apurar projetos pequenos e posteriormente em projetos *open-sources* para a verificação da existência de alguma defasagem de estruturas entre qualquer versão da linguagem Java que fora concebido para a versão atual e estável na qual a linguagem se encontra. O desenvolvimento das ferramentas será com a versão mais atual da linguagem Java que neste momento é o Java versão 8 para verificar se os softwares desenvolvidos está versão nesta versão e como acompanharam a evolução de decadas de novas versões sem atualização do codigo por motivo de engenharia outro qualquer.

Capítulo 2

Java

2.1 Historia da linguagem

No começo da decada de 90 um pequeno grupo de engenheros da Oracle chamados de "Green Team" acreditava que a próxima onde de na area da computação seria a união de equipamentos eletroeletrônicos com os computadores. O "Green Team" liderado por James Gosling, demonstraram que a linguagem de programaçao Java, que foi desenvolvida pela equipe e originalmente era chamado de Oak, foi desenvolvida para dispositivos de entretenimento como aparelhos de tv a cabo, porem não foi bem aceita no meio. Em 1995 com a massificação da Internet foi quando a linguagem Java teve sua primeira grande aplicação o navegador Netscape.

Java é uma linguagem de programação de propósito geral orientada a objetos, concebida especificadademente para ter poucas dependencias de implementação que isso acarreta que uma vez que a aplicação fora desenvolvida ela poderá ser executada em qualquer lugar.

Na sua primeira versão chamada de Java 1 (JDK* 1.0.2) onde introduziram oito pacotes básicos do java como: java.lang, java.io, java.util, java.net, java.awt, java.awt.image, java.awt.peer e java.applet. Foi usado para o desenvolvimento de ferramentas populares na epoca como o Netscape 3.0 e o Internet Explorer 3.0.

Sua segunda versão foi o JDK* 1.1 onde trouxe ganhos em funcionalidades, desempenho e qualidade. Novas aplicações tambem surgiram como : JavaBeans, aprimoramento do AWT*, novas funcionalidades como o JDBC*, acesso remoto ao objeto (RMI*) e suporte ao padrão Unicode 2.0.

Na terceira versão Java 2 (JDK* 1.2) ofereceu melhorias significativas no desempenho, um novo modelo de segurança, flexível e um conjunto completo de aplicações de programação interfaces (APIs). Os novos recursos da plataforma Java 2 incluiram:

- O modelo de "sandbox"foi ampliado para dar aos desenvolvedores, usuários e administradores de sistema a opção de especificar e gerenciar um conjunto de políticas de segurança flexíveis que governam as ações de uma aplicação ou applet que pode ou não ser executada.

- Suporte nativo a thread para o ambiente operacional Solaris. Compressão de memória para classes carregadas. Alocação de memória com mais desempenho e melhor para a coleta de lixo. Arquitetura de máquina virtual conectável para outras máquinas virtuais, incluindo a Java HotSpot VMNew. Just in Time (JIT*). Java Native Interface (JNI*) de conversão.
- O conjunto de componentes de projeto, GUI* (Swing). API* Java 2D que fornece novos recursos gráficos 2D e AWT*, bem como suporte para impressão. O Java *look and fell*. Uma nova API de acessibilidade.
- Framework de entrada de caracteres (suporte a japonês, chinês e coreano). Complexo de saída usando a API* do Java 2D para fornecer um *display* bi-direcional, de alta qualidade de japonês, árabe, hebraico e outras línguas de caracteres.
- Java Plug-in para navegadores da web, incluída na plataforma Java 2, fornecendo um tempo de execução totalmente compatível com a máquina virtual Java amplamente implantadas em navegadores.
- Invocação das operações ou serviços de rede remoto. Totalmente compatível com Java ORB* e incluído no tempo de execução.
- JDBC* que fornece um acesso mais fácil aos dados para consultas mais flexíveis. Melhor desempenho e estabilidade são promovidos por cursores de rolagem e suporte para SQL3* de tipos.

Em 8 de maio de 2000 foi anunciado o Java 2 versão 1.3 que trouxe ganho de desempenho em relação a primeira versão da JS2E* de cerca de 40% no tempo de *start-up* e de 20%. Tambem trouxe novas funcionaliadades como:

- O Java HotSpot VM* de cliente e suas bibliotecas atentando ao desempenho ao fazer o J2SE* versão 1.3 a *realease* o mais rápido até à data.
- Novos recursos, como o *caching applet* e instalação do pacote opcional Java através da tecnologia Java *Plug-in* para aumentar a velocidade e a flexibilidade com que os *applets* e aplicativos baseados na tecnologia Java pode ser implantado. Java *Plug-in* tecnologia é um componente do ambiente de execução Java 2 que permite Java *applets* e aplicativos para a execução.
- O novo suporte para RSA* assinatura eletrônica, gerenciamento de confiança dinâmico, certificados X.509, e verificação de arquivos o que significa o aumento das possibilidades que os desenvolvedores tem para proteger dados eletrônicos.
- Uma série de novos recursos e ferramentas de desenvolvimento da tecnologia J2SE* versão 1.3 que permite o desenvolvimento mais fácil e rápido de aplicações baseadas na tecnologia *web* ou Java *standalone* de alto desempenho.
- A adição de RMI*/IIOP* e o Jndi (JNDI*) para a versão 1.3, melhora na interoperabilidade J2SE*. RMI*/IIOP* melhora a conectividade com sistemas de *back-end* que suportam CORBA*. JNDI* fornece acesso aos diretórios que suportam o populares LDAP* Lightweight Directory Access Protocol, entre outros.

No ano de 2000 no dia 6 de Fevereiro, foi lançado a J2SE* versão 1.4. Com a versão 1.4, as empresas puderam usar a tecnologia Java para desenvolver aplicativos de negócios mais exigentes e com menos esforço e em menos tempo. As novas funcionalidades como a nova I/O* e suporte a 64 bits. A J2SE* se tornou plataforma ideal para a mineração em grande escala de dados, inteligência de negócios, engenharia e científicos. A versão 1.4 forneceu suporte aprimorado para tecnologias padrões da indústria, tais como SSL*, LDAP* e CORBA* a fim de garantir a operacionalidade em plataformas heterogêneas, sistemas e ambientes. Com o apoio embutido para XML*, a autenticação avançada, e um conjunto completo de serviços de segurança, está versão forneceu base para padrões de aplicações Web e serviços interoperáveis. O J2SE* avançou o desenvolvimento de aplicativos de cliente com novos controles de GUI, acelerou Java 2D, a performance gráfica, internacionalização e localização expandida de apoio, novas opções de implantação e suporte expandido para o até então Windows XP.

Com a chegada da JSE2* versão 1.5 (Java 5.0) em 6 de ferevereiro de 2002, impulsionou benefícios extensivos para desenvolvedores, incluindo a facilidade de uso, desempenho global e escalabilidade, monitoramento do sistema e gestão e desenvolvimento. O Java 5 foi derivado do trabalho de 15 componentes Java Specification Requests (JSRs) englobando recursos avançados para a linguagem e plataforma. Os líderes da indústria na época que participam no grupo de peritos J2SE 5.0 incluiram: Apache Software Foundation, Apple Computer, BEA Systems, Borland Software Corporation, Cisco Systems, Fujitsu Limited, HP, IBM, Macromedia, Nokia Corporation, Oracle, SAP AG, SAS Institute, SavaJe Technologies e Sun Microsystems.

Novas funcionalidades foram implementadas como:

- Facilidade de desenvolvimento: os programadores da linguagem Java pode ser mais eficiente e produtivos com os recursos de linguagem Java 5 que permitiram a codificação mais segura. Nesta versão surgiu *Generics*, tipos enumerados, metadados e autoboxing de tipos primitivos permitindo assim uma fácil e rápida codificação.

- Monitoramento e gestão: Um foco chave para a nova versão da plataforma, a aplicativos baseados na tecnologia Java *Virtual Machine* que passou a ser monitorado e gerenciado com o *built-in* de suporte para Java *Management Extensions*. Isso ajudou a garantir que seus funcionários, sistemas de parceiros do cliente permanecessem em funcionamento por mais tempo. Suporte para sistemas de gestão empresarial baseados em SNMP* também é viável.

- Um olhar novo aplicativo, mais moderna, baseada na tecnologia Java padrão e proporciona uma sensação GUI para aplicativos baseados na tecnologia Java. A J2SE* 5.0 teve suporte completo a internacionalização e também possuindo suporte para aceleração de hardware por meio da API* OpenGL* e tambem para o sistema operacional Solaris e sistemas operacionais da distribuição Linux.

- Maior desempenho e escalabilidade: A nova versão incluiu melhorias de desempenho, tais como menor tempo de inicialização, um menor consumo de memória e JVM* auto ajustável para gerar maior desempenho geral do aplicativo e desenvolvimento em J2SE* 5.0 em relação às versões anteriores.

Java 1.6 (Java 6) foi divulgado em 11 de dezembro de 2006. Tornou o desenvolvimento mais fácil, mais rápido e mais eficiente em termos de custos e ofereceu funcionalidades para serviços web, suporte linguagem dinâmica, diagnósticos e aplicações desktop. Com a chegada dessa nova versão do Java houve combinação com o NetBeans IDE 5.5 fornecendo aos desenvolvedores uma estrutura confiável, de codigo aberto e compatível, de alta performance para entregar aplicativos baseados na tecnologia Java mais rápido e mais fácil do que nunca. O NetBeans IDE* fornece uma fonte aberta e de alto desempenho, modular, extensível, multi-plataforma Java IDE* para acelerar o desenvolvimento de aplicações baseadas em software e serviços *web*. Novas funcionalidades foram implementadas como:

- O Java 1.6 ajudou a acelerar a inovação para o desenvolvedor, aplicativos de colaboração *online* e baseadas na *web*, incluindo um novo quadro de desenvolvedores APIs para permitir a mistura da tecnologia Java com linguagens de tipagem dinâmica, tais como PHP, Python, Ruby e tecnologia JavaScript. A Sun também criou uma coleção de mecanismos de script e pré-configurado o motor JavaScript Rhino na plataforma Java. Além disso, o software inclui uma pilha completa de clientes de serviços web e suporta as mais recentes especificações de serviços *web*, como JAX-WS* 2.0, JAXB* 2.0, STAX* e JAXP*.
- A plataforma Java 1.6 forneceu ferramentas expandidas para o diagnóstico, gestão e monitoramento de aplicações e também inclui suporte para o novo NetBeans Profiler 5.5 para Solaris DTrace e, uma estrutura de rastreamento dinâmico abrangente que está incluído no sistema operacional Solaris 10. Além disso, o software Java SE 6 aumenta ainda mais a facilidade de desenvolvimento com atualizações de interface ferramenta para o Java Virtual Machine (JVM) e o Java Platform Debugger Architecture (ACDP)*.

Java 7 foi lançado no dia 28 de julho de 2011. Essa versão foi resultado do desenvolvimento de toda a indústria envolvendo uma revisão de codigo aberto e extensa colaboração entre os engenheiros da *Oracle* e membros do ecossistema Java em todo o mundo através da comunidade *OpenJDK** e do *Java Community Process* (JCP)*. Compatibilidade com versões anteriores de Java 7 com versões anteriores da plataforma a fim de preservar os conjuntos de habilidades dos desenvolvedores de software Java e proteger os investimentos em tecnologia Java.

Com essa versão novas funcionalidades foram adicionadas:

- As alterações de linguagem ajudaram a aumentar a produtividade do desenvolvedor e simplificar tarefas comuns de programação, reduzindo a quantidade de código necessário, esclarecendo sintaxe e tornar o código com mais legibilidade.
- Melhor suporte para linguagens dinâmicas incluindo: Ruby, Python e JavaScript, resultando em aumentos substanciais de desempenho no JVM*.
- Uma nova API* *multicore-ready* que permite aos desenvolvedores para se decompor mais facilmente problemas em tarefas que podem ser executadas em paralelo em números arbitrários de núcleos de processador.
- Uma interface de I/O* abrangente para trabalhar com sistemas de arquivos que podem acessar uma ampla gama de atributos de arquivos e oferecem mais informações quando ocorrem erros.

- Novos recursos de rede e de segurança. Suporte expandido para a internacionalização, incluindo suporte a Unicode 6.0. Versões atualizadas das bibliotecas padrão.

Com o lançamento do Java SE* 8 em 18 de Março de 2014, permitiu uma maior produtividade e desenvolvimento de aplicativos significativos aumentos de desempenho através da redução de linhas de código, *collectons* melhoradas, modelos mais simples de programação paralela e uso mais eficiente de processadores multi-core modernos.As principais características do JDK* 8 são o Projeto Lambda, Nashorn JavaScript Engine, um conjunto de perfis compactas e a remoção da "geração permanente"do HotSpot Java Virtual Machine (JVM*).A JDK* 8 alcançou desempenho recorde mundial para 4 sistemas de soquete em servidores baseados em Intel e NEC por 2 sistemas de soquete em servidores SPARC da Oracle T5, com uma melhoria de desempenho de 12% para 41% em comparação com o JDK* 7 na mesma configuração de Oracle. O JDK* 8 adicionou novas funcionalidades como:

- As expressões lambda são suportados pelas seguintes características: As referências a metodos são compactos, maior legibilidade expressões lambda para métodos que já têm um nome. Métodos padrão que permitem adicionar novas funcionalidades para as interfaces de suas bibliotecas e assegurar a compatibilidade binária com o código escrito para versões mais antigas dessas interfaces. Eles são os métodos de interface que têm uma aplicação e a palavra-chave padrão no início da assinatura do método. Além disso, pode-se definir métodos estáticos em interfaces. Novos e aprimorados APIs que se aproveitam de expressões lambda e dos *streams* em Java 8 descrevem as classes novos e aprimorados que se aproveitam de expressões lambda e *streams*.

- O compilador Java aproveita digitação alvo para inferir os parâmetros de tipo de um método de invocação genérica. O tipo de destino de uma expressão é o tipo de dados que o compilador Java espera, dependendo de onde a expressão aparece. Por exemplo, você pode usar o tipo de destino de uma instrução de atribuição para o tipo de inferência em Java 7. No entanto, em Java 8, você pode usar o tipo de destino para a inferência de tipos em mais contextos.

- Anotações sobre tipos Java. Agora é possível aplicar uma anotação em qualquer lugar onde um tipo é usado. Utilizado em um conjunto com um sistema de tipo de conector, isso permite a verificação de tipo mais forte de seu código.

- Repetindo Anotações. Agora é possível aplicar o mesmo tipo de anotação mais de uma vez para a mesma declaração ou o tipo de utilização.

2.2 Aspectos evolutivos da liguagem Java

2.2.1 Java 2

A primeira versão do Java Security, disponível no JDK* 1.1, contém um subconjunto dessa funcionalidade, incluindo APIs para:

- Assinaturas Digitais: Algoritmos de assinatura digital, como DSA* ou MD5* com RSA*. A funcionalidade inclui a geração de chaves público/privado , bem como assinatura e verificação de dados digitais.
- Gerenciamento de Chaves: Um conjunto de abstrações para o gerenciamento de "diretores" (entidades como usuários individuais ou grupos), suas chaves, e os seus certificados. Ele permite que aplicativos para projetar seu próprio sistema de gerenciamento de chaves, e para interoperar com outros sistemas em alto nível.
- Lista de controle de acesso: Um conjunto de abstrações para o gerenciamento de "diretores" e suas permissões de acesso.
- A obtenção de um objeto de assinatura:

```
import java.security.Signature;
import java.security.NoSuchAlgorithmException;

public class SignFile {
    Signature signature;

    private void init(String algorithm)
    throws NoSuchAlgorithmException{
        signature = Signature.getSignature(algorithm);
    }
}
```

- Em versões anteriores, Java suportava apenas *top-level* classes, que devem ser membros de pacotes. Na versão 1.1, o programador Java pode agora definir classes internas como membros de outras classes, localmente dentro de um bloco de instruções, ou (anonimamente) dentro de uma expressão.

```
  public class FixedStack {
    ...
    public java.util.Enumeration elements() {
        return new FixedStack$Enumerator(this);
    }
}

class FixedStack$Enumerator implements java.util.Enumeration {
    private FixedStack this$0;
    FixedStack$Enumerator(FixedStack this$0) {
        this.this$0 = this$0;
        this.count = this$0.top;
    }

    int count;
    public boolean hasMoreElements() {
```

```
        return count > 0;
    }
    public Object nextElement() {
        if (count == 0)
            throw new NoSuchElementException("FixedStack");
        return this$0.array[--count];
    }
}
```

- Para escrever um objeto remoto (RMI*), você escrever uma classe que implementa uma ou mais interfaces remotas.

```
package examples.hello;
public interface Hello extends java.rmi.Remote {
    String sayHello() throws java.rmi.RemoteException;
}
```

- HelloImpl.java

```
package examples.hello;

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject implements Hello{
    private String name;

    public HelloImpl(String s) throws RemoteException {
            super();
            name = s;
}

    public String sayHello() throws RemoteException {
           return  "Hello World!";
}

    public static void main(String args[]){
           System.setSecurityManager(new RMISecurityManager());
           try {
                     HelloImpl obj = new HelloImpl("HelloServer");
                     Naming.rebind("//myhost/HelloServer", obj);
                     System.out.println("HelloServer bound in registry");
           }catch (Exception e) {
                     System.out.println("HelloImpl err: " + e.getMessage());
                     e.printStackTrace();
```

```
                    }
            }
}
```

2.2.2 Java 4

- *Assertion Facility.* As *assertions* são expressões booleanas que o programador acredita ser verdade sobre o estado de um programa de computador. Por exemplo, depois de ordenar uma lista o programador pode afirmar que a lista está em ordem crescente. Avaliando as afirmações em tempo de execução para confirmar a sua validade é uma das ferramentas mais poderosas para melhorar a qualidade do código, uma vez que rapidamente se descobre equívocos do programador sobre o comportamento de um programa.

2.2.3 Java 5

- *Generics.* Este novo recurso para o sistema de tipo permite que um tipo ou método operare em objetos de vários tipos, proporcionando em tempo de compilação tipo de segurança. Acrescenta em tempo de compilação um tipo de segurança para as *collections* e elimina o trabalho penoso de *casting.* Um exemplo do uso de *colletions* e *generics* respectivamente:

```
static void expurgate(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
      if (((String) i.next()).length() == 4)
        i.remove();
}

static void expurgate(Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext(); )
      if (i.next().length() == 4)
        i.remove();
}
```

- *For-Each Loop.* Esta nova estrutura de linguagem elimina o trabalho e erro de propensão de iteradores e variáveis de índice quando a iteração ocorre sobre coleções e arrays. Como a construção evoluiu com o advento dessa nova estrutura:

```
void cancelAll(Collection<TimerTask> c) {
    for (Iterator<TimerTask> i = c.iterator(); i.hasNext(); )
        i.next().cancel();
```

```
}

void cancelAll(Collection<TimerTask> c) {
    for (TimerTask t : c)
        t.cancel();
}
```

- *Varargs*. Esta nova estrutura tende a eliminar a necessidade de passagem manual de listas de argumentos em um array ao invocar métodos que aceitam de um comprimento variável de uma lista de argumentos. Nas versões anteriores, um método levava um número arbitrário de valores necessários a criar uma matriz e colocar os valores para a matriz antes de chamar o método.

```
public class Test {
    public static void main(String[] args) {
        int passed = 0;
        int failed = 0;
        for (String className : args) {
            try {
                Class c = Class.forName(className);
                c.getMethod("test").invoke(c.newInstance());
                passed++;
            } catch (Exception ex) {
                System.out.printf("%s failed: %s%n", className, ex);
                failed++;
            }
        }
        System.out.printf("passed=%d; failed=%d%n", passed, failed);
    }
}
```

- *Autoboxing/Unboxing*. Esta nova estrutura elimina o trabalho de conversão manual entre tipos primitivos (como *int*) e os tipos de classes *wrapper*

2.2.4 Java 6

Não ocorreu mudanças ou introdução de novas estruturas na linguagem Java

2.2.5 Java 7

- *Multi Catch* e lançamento de exceções com melhora na verificação de tipos. Um único bloco *catch* poderá lidar com mais de um tipo de exceção. Além disso, o

compilador executa a análise mais precisa das excepções. Isso permite que o programador especifique tipos de exceção mais específicos na cláusula de uma declaração método. Um exemplo de como era as estruturas que usavam *cacths* e com a introdução de *multi cacth* com o Java 7, respectivamente.

```
  catch (IOException ex) {
     logger.log(ex);
     throw ex;
} catch (SQLException ex) {
     logger.log(ex);
     throw ex;
}
```

```
  catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

- O *try-with-resouces*. A declaração *try-with-resouces* é uma instrução *try* que declara um ou mais recursos. Um recurso é um objeto que deve ser fechada após o programa terminar com ele. Essa declaração garante que cada recurso é fechada no final da declaração.

```
public static void writeToFileZipFileContents(String zipFileName,
String outputFileName)
    throws java.io.IOException {

    java.nio.charset.Charset charset =
    java.nio.charset.StandardCharsets.US_ASCII;
    java.nio.file.Path outputFilePath = java.nio.file.Paths.get(outputFileName);

    try (
      java.util.zip.ZipFile zf = new java.util.zip.ZipFile(zipFileName);
      java.io.BufferedWriter writer =
      java.nio.file.Files.newBufferedWriter(outputFilePath, charset);
    ) {

    for (java.util.Enumeration entries = zf.entries();
    entries.hasMoreElements();) {

      String newLine = System.getProperty("line.separator");
      String zipEntryName = ((java.util.zip.ZipEntry)
      entries.nextElement()).getName() + newLine;
      writer.write(zipEntryName, 0, zipEntryName.length());
```

```
      }
    }
  }
```

- Inferência de tipos para criação de instâncias em *generics*. Com o Java 7 pode-se substituir os argumentos de tipo necessários para invocar o construtor de uma classe genérica com um conjunto vazio de parâmetros de tipo (<>), desde que o compilador infira os argumentos de tipo a partir do contexto. Este par de colchetes angulares é informalmente chamado de diamante.

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();

Map<String, List<String>> myMap = new HashMap<>();

List<String> list = new ArrayList<>();
list.add("A");

  // The following statement should fail since addAll expects
  // Collection<? extends String>

list.addAll(new ArrayList<>());

class MyClass<X> {
  <T> MyClass(T t) {
    // ...
  }
}
```

2.2.6 Java 8

- Melhoria na inferência de tipos. O compilador Java aproveita digitação para inferir os parâmetros de tipo de uma invocação de método genérica. O tipo de destino de uma expressão é o tipo de dados que o compilador Java espera, dependendo de onde a expressão aparece. Por exemplo, pode-se usar o tipo de destino de uma instrução de atribuição para o tipo de inferência em Java 7. No entanto, em Java 8, pode-se usar o tipo de destino para a inferência de tipos em mais contextos. O exemplo mais proeminente está usando tipos de destino de um método de invocação para inferir os tipos de dados dos seus argumentos.

```
List<String> stringList = new ArrayList<>();
```

```
stringList.add("A");
stringList.addAll(Arrays.asList());
```

- Expressões lambda. Permitem encapsular uma única unidade de comportamento e passá-lo para outro código. Pode-se usar uma expressãos lambda, se quiser uma determinada ação executada em cada elemento de uma *collection*, quando o processo for concluído, ou quando um processo encontra um erro.

```
public class Calculator {

    interface IntegerMath {
        int operation(int a, int b);
    }

    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String... args) {

        Calculator myApp = new Calculator();
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
            myApp.operateBinary(20, 10, subtraction));
    }
}
```

Referências

[1] Jeffrey L. Overbey and Ralph E. Johnson. Regrowing a language: refactoring tools allow programming languages to evolve. (10):493–502, October 2009. 1, 2