Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

MARIANA BUDISKI DE LIMA

O IMPACTO DAS BOAS PRÁTICAS DE PROGRAMAÇÃO NO GERENCIAMENTO DE MEMÓRIA NO JAVA

Assis, SP
2012

MARIANA BUDISKI DE LIMA

O IMPACTO DAS BOAS PRÁTICAS DE PROGRAMAÇÃO NO GERENCIAMENTO DE MEMÓRIA NO JAVA

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Bacharelado em Ciência da Computação.
Orientador: Prof. Fernando Cesar Lima
Área de Concentração: Informática.

Assis, SP
2012

FICHA CATALOGRÁFICA

De LIMA, Mariana Budiski.

O Impacto das Boas Práticas de Programação no Gerenciamento de Memória no Java / Mariana Budiski de Lima. Fundação Educacional do Município de Assis – FEMA – Assis, 2012.

65p.

Orientador: Prof. Fernando Cesar Lima.
Trabalho de Conclusão de Curso – Instituto Municipal de Ensino Superior de Assis – IMESA.

1. Gerenciamento de Memória. 2. Java. 3. Programação

CDU 001.6
Biblioteca FEMA

MARIANA BUDISKI DE LIMA

O IMPACTO DAS BOAS PRÁTICAS DE PROGRAMAÇÃO NO GERENCIAMENTO DE MEMÓRIA NO JAVA

Trabalho de Conclusão de Curso apresentado ao

Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Bacharelado em Ciência da Computação, analisado pela seguinte comissão examinadora:

Orientador: Prof. Fernando Cesar Lima

Analisador : Prof. Dr. Alex Sandro Romeo de Souza Polleto

Assis, SP

2012

AGRADECIMENTOS

A Fernando Lima, que me orientou neste trabalho mostrando sempre onde deveria melhorar e pelas excelentes ideias;

A meus amigos e familiares pela paciência e apoio, em especial Rafael Moraes de Oliveira que sempre me incentivou e ajudou nos momentos de maiores dificuldades;

A todos que direta e indiretamente, contribuíram para a realização deste trabalho.

RESUMO

Gerenciar a memória utilizada não é tarefa fácil, ela exige diversos tratamentos e é um componente finito do computador. Linguagens de programação como o C e C++ exigem do programador que ele aloque e libere toda a memória utilizada, diferentemente do Java que possui gerenciamento de memória automático. O Garbage Collector verifica áreas de memória que não estão mais sendo utilizadas e libera a memória automaticamente, livrando o programador desses detalhes possibilitando que ele se concentre apenas na lógica de negócio de suas aplicações, causando muitas vezes um excesso de confiança nesse mecanismo, porém práticas de programação danificam esse mecanismo muitas vezes causando erros por falta de memória. As soluções foram criadas para o programador das boas práticas. Neste trabalho será apresentado oque fazer quando uma aplicação esta consumindo muita memória e como corrigir esses problemas com boas práticas de programação.

Palavras Chave: Gerenciamento de memória. Java. Programação.

ABSTRACT

Manage the memory used is no easy task, it requires several treatments and is a component of the computer finite. Programming languages like C and C + + programmer who require it to allocate and release all memory used, unlike Java has automatic memory management. The Garbage Collector checks for areas of memory that are no longer being used and automatically frees memory, freeing the programmer of these details enabling him to focus only on the business logic of their applications, often causing an excess of confidence in this mechanism, but practical programming damage this mechanism often causing errors due to lack of memory. The solutions are designed for the programmer good practice. In this work will be presented what to do when an application is consuming too much memory and how to fix these problems with good programming practices.

Keywords: Memory Management. Java. Programming.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1. INTRODUÇÃO

Com o avanço dos *softwares* e da tecnologia da informação, é cada vez maior a necessidade de se processar um grande volume de dados. Para isso, os desenvolvedores de *softwares* necessitam de mais programas rodando simultaneamente para processar tanta informação quanto for possível, sendo assim, o uso da memória para o processamento e o desempenho das aplicações são cada vez mais importantes.

Gerenciar a memória utilizada não é tarefa fácil, ela é um componente finito do computador e envolve diversos tratamentos para que seja utilizada corretamente. O gerenciamento de memória pode ser definido como técnicas para tornar o uso da memória eficiente, para carregar a quantidade máxima de processos possível e diminuir o tempo ocioso do processador. O gerenciamento de memória nas linguagens de programação têm influencia direta no desempenho do sistema e na capacidade do programador em escolher a estrutura de dados adequada para determinada situação (PINTO; LUNA, 2010).

Reconhecendo o impacto dos microprocessadores em dispositivos eletrônicos, em 1991, a Sun Microsystems financiou um projeto interno em sua empresa que resultou no desenvolvimento de uma linguagem de programação orientada a objetos baseada no C++ que recebeu o nome Java.

O Java é uma linguagem simples, robusta e segura, onde não há possibilidade de gerenciar a memória do sistema explicitamente, diferentemente do C e C++, o Java possui o gerenciamento de memória automático, com algoritmos e estratégias para liberação de memória. Gerenciamento de memória automático significa que o programador não precisa se preocupar com as tarefas anteriores, porém, algumas práticas de programação ou até mesmo de desenvolvimento de software podem aumentar em muito o consumo da memória pela aplicação e diminuir drasticamente seu desempenho.

Programadores Java, gerentes de projetos e projetistas estão acostumados a não se preocuparem com o consumo de memória por suas aplicações já que ele é feito

de forma automática. Outro fator, e talvez o mais propício, que influencia na despreocupação dos programadores sem dúvida é o avanço do hardware, hoje em dia os computadores possuem muita memória e muita capacidade de processamento, assim os programadores acham que qualquer *software* vai “rodar” sem problemas e por isso não se preocupam com o quanto a aplicação pode ser “pesada”. O resultado é que em um ambiente de testes o sistema funciona muito bem, porém quando entra em produção pode começar a surgir erros apontando para a falta de memória.

Neste trabalho ficará visível que apesar do gerenciamento de memória dinâmico, estratégias de coleta de lixo e muita memória no computador não significa que um erro causado por falta de memória não possa ocorrer. Além disso, uma das desvantagens do Java é exatamente essa, o consumo de memória, aplicações escritas em Java consomem muita memória quando comparada a aplicações de outras linguagens.

Algumas aplicações são implantadas em ambientes atípicos e precisam de um cuidado maior com a memória e desempenho assim como aplicações WEB que consomem mais memória a cada novo acesso simultâneo. Este trabalho abordará o que fazer quando erros causados por falta de memória ocorrerem, além do funcionamento da máquina virtual do Java e os recursos utilizados por essa linguagem de programação para o gerenciamento de memória, e principalmente como as boas práticas de programação podem impactar positivamente no consumo de memória das aplicações.

1.1. OBJETIVOS

1.1.1. OBJETIVO GERAL

Analisar como as más práticas de programação podem influenciar no gerenciamento de memória no Java.

1.1.2. OBJETIVOS ESPECÍFICOS

a) Estudar o Gerenciamento de memória no Java.

b) Detectar más práticas de programação em aplicações Java.

c) Mostrar o consumo de memória por aplicações Java utilizando o Visual VM.

d) Apresentar alternativas de boas práticas de programação para diminuir o consumo de memória e aumentar o desempenho de aplicações mal implementadas.

1.2. JUSTIFICATIVA

O *hardware* dos computadores vem melhorando a cada ano, hoje em dia há computadores com *terabytes* de armazenamento, 8Gb de memória RAM e vários núcleos de processamento, avanços inimagináveis há algumas décadas quando o programador tinha que se preocupar em como usar a pouca memória disponível e com o desempenho de seus *softwares* nos sistemas monotarefas da época. Componentes como memórias e processadores avançaram bastante, por esse motivo os programadores atuais não sentem a necessidade de se preocuparem tanto com o consumo de recursos por suas aplicações, além disso, a linguagem Java possui gerenciamento de memória automático. Alguns programadores e autores pregam que o programador Java não tem que se preocupar com o gerenciamento de memória já que ela é feita pela própria máquina virtual, mas isso não é absoluto. E se uma aplicação estiver em um ambiente atípico? E se for uma aplicação *web* que tem milhares de acessos diários? Essa aplicação estará consumindo cada vez mais memória a cada novo acesso simultâneo.

Algumas aplicações utilizam banco de dados em memória (database in-memory) que são banco de dados carregados puramente na memória principal, nesse caso o programador tem uma preocupação extra, não só a aplicação em si estará consumindo cada vez mais memória como também o banco de dados carregado.

Aplicações que não são bem implementadas podem consumir muita memória e diminuir seu desempenho, muitas vezes causando erros e etc. Desta constatação surgiu a ideia que este trabalho se propõe investigar: estudos de caso em que as más práticas de programação impactam no gerenciamento de memória no Java comprometendo a qualidade do *software* e mostrar que, mesmo com todos os recursos que a JVM oferece para gerenciar memória não é suficiente quando o programa apresenta más prátcas de programação.

1.3. MOTIVAÇÃO

O Java domina grande parte do mercado. Dentre os vários motivos para a rápida e ampla adoção do Java talvez o mais importante, além da portabilidade, seja o suporte ao desenvolvimento de *software* de qualidade, pois a linguagem possui diversas características que fornecem esse suporte.

As más práticas de programação diminuem a qualidade do software, que é essencial para competitividade e sobrevivência tanto de desenvolvedores quanto consumidores de *software*.

1.4. PERSPECTIVAS DE CONTRIBUIÇÃO

Este trabalho apresentará um estudo sobre os mecanismos de gerenciamento de memória no Java e como as más práticas de programação aumentam o consumo de memória pela aplicação e como isso pode causar problemas, diminuindo a qualidade do *software*. O trabalho pretende auxiliar os programadores Java que não possuem conhecimento de tais mecanismos e alertar sobre essas más práticas que geralmente são comuns apontando boas práticas de programação como alternativa. Também será uma fonte de informações sobre a JVM, e oferecerá subsídios de fundamental interesse para aqueles programadores que desejam ter conhecimento mais avançado sobre a linguagem Java.

1.5. METODOLOGIA DE PESQUISA

A metodologia utilizada para o desenvolvimento desse projeto é a pesquisa bibliográfica de trabalhos acadêmicos e artigos disponibilizados na INTERNET, tutoriais sobre gerenciamento de memória em Java e obras como a de PAUL e HARVEY DEITEL que visa ensinar como programar em Java e atenta aos erros mais comuns de programação e dá dicas de desempenho.

Serão desenvolvidos algumas aplicações em Java para estudos de caso, fazendo um comparativo entre as boas e as más práticas de programação que podem ser comuns para analisar o consumo de memória dessas aplicações e qual o impacto das boas práticas de programação para esse consumo.

Essa análise chama-se *Profiling* e será realizada com o Visual VM, ferramenta disponível na distribuição da JDK (*Java Development Kit*) e assim coletar os dados. Em seguida será apresentado o comparativo entre os dados coletados.

1.6. ESTRUTURA DO TRABALHO

O primeiro capítulo aborda o conteúdo geral do trabalho, quais as justificativas e motivações e a metodologia de pesquisa.

O segundo capítulo apresenta a fundamentação teórica da pesquisa, descrevendo brevemente as tecnologias que serão utilizadas durante a pesquisa.

O terceiro capítulo apresentara o funcionamento do gerenciamento de memória automático do Java, erros causados pela falta dela e o funcionamento da JVM, suas áreas de memória e detalhes sobre a criação e destruição de objetos.

O quarto capítulo detalha uma abordagem para identificação de problemas relacionados à memoria em aplicações Java, chamado *Profiling*, técnica que consiste no monitoramento de aplicações em tempo de execução obtendo detalhes da memória utilizada.

O quinto capítulo apresenta os erros causados pela falta de memória, más práticas de programação que devem ser evitadas no desenvolvimento de aplicações e boas práticas de programação como alternativa para corrigir os possíveis problemas.

O sexto capítulo apresentara o estudo de caso de um sistema real, o Sistema de Gestão Financeira do Projeto Rede Ciranda, onde será feito o Profile no Visual VM de módulos da aplicação que apresentam más práticas de programação com análise e coleta de dados e então, será apresentado boas práticas de programação para correção dos problemas.

2. FUNDAMENTAÇÃO TEÓRICA

Neste capítulo será feita a fundamentação teórica das tecnologias utilizadas para desenvolvimento da pesquisa, bem como as implementações para experimento. Será feita a descrição das tecnologias Java utilizadas, utilitários, implementações, servidor e banco de dados.

2.1. JAVA

Java é uma linguagem de programação de alto nível e uma plataforma de computação lançada pela Sun Microsystems em 1995. É uma linguagem orientada a objetos , segura e independente de plataforma.

Java seduziu os programadores com sua sintaxe amigável, recursos da orientação a objetos, gerenciamento de memória automático e principalmente a portabilidade, escrever programas e executá-los em qualquer sistema operacional era e ainda é um grande atrativo e por isso Java é muito popular (SIERRA; BATES, 2010).

Segundo (METSKER, 2004) Java é importante e provavelmente será a base para as futuras gerações de linguagens de computação.

A plataforma de computação Java é um conjunto de tecnologias e pode ser dividida em três partes:

- JSE (*Java Standard Edition*) é a plataforma padrão com desenvolvimento voltado para *desktops.*
- JEE (*Java Enterprise Edition*) é a plataforma voltada para desenvolvimento de aplicações de grande porte voltadas para WEB.
- JME (*Java Micro Edition*) é a plataforma voltada para dispositivos móveis e embarcados.

A linguagem Java é compilada e diferentemente das demais linguagens, seu código é primeiramente compilado para *bytecode* para depois ser executado pela JVM (*Java Virtual Machine*).

2.1.1. JVM

A JVM é o programa responsável por executar os aplicativos Java, ela converte o os *bytecodes* do código Java em linguagem de máquina que então é executado. A JVM é a grande responsável pela portabilidade do Java, depois de um programa ser compilado para *bytecode* qualquer dispositivo que possuir a JVM instalada pode executar esse programa. Detalhes do funcionamento e arquitetura da JVM serão tratados em detalhes no Capítulo 3.

2.1.2. COMO O JAVA FUNCIONA

A Figura 1 mostra os passos para a execução de um aplicativo Java.

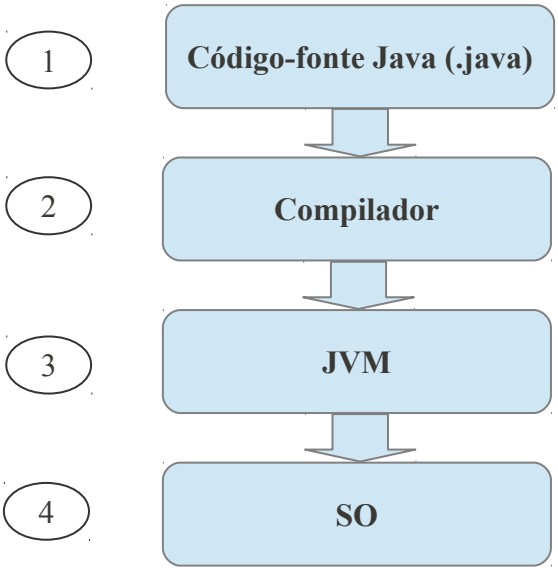


Figura 1 – Etapas para execução de aplicações Java

1. Nesta primeira etapa se dá a criação de arquivos onde são inseridos códigos (código-fonte Java) pelo programador, e salvos com a extensão “.java”. Para criarmos aplicações Java, que contém vários arquivos, geralmente são utilizados ambientes de desenvolvimento conhecidos como IDEs (*Integrated Development Environmen*t). As IDEs fornecem ferramentas que auxiliam no desenvolvimento do software como editores, depuradores de programas e verificadores de lógica e sintaxe do código-fonte.

2. A segunda etapa é a compilação do programa Java em *bytecodes*. O compilador é distribuído junto com o kit de desenvolvimento Java, o JDK, ele converte o código-fonte Java, os arquivos “.java” em *bytecodes*, gerando um outro arquivo com extensão “.class”, ou seja, para cada classe Java que criamos teremos um arquivo com extensão “.class” de mesmo nome contendo esses bytecodes que mais tarde são interpretados pela JVM. Os *bytecodes* são arquivos com codificação binária, eles representam tarefas a serem executadas na fase de execução do programa.

3. A JVM (Java virtual machine) executa os *bytecodes* transformando-os em linguagem de máquina que então podem ser executada pelo dispositivo.

4. Qualquer dispositivo independente do sistema operacional com a JVM instalada é capaz de executar aplicativos Java.

É importante salientar que a portabilidade tem seu preço, programas Java são mais lentos para serem executados devido ao processo que tem que passar para então serem executados pela JVM, em comparação com outra linguagem de programação muito conhecida o qual o Java foi inspirado, o C++, que é apenas compilado para linguagem de máquina.

Outras características importantes do Java são:

- Sintaxe simples,
- Concisa, não contém redundâncias,
- Concorrente, pois suporta *multithread* e monitores em suas aplicações,
- Estável
- E não se prende a nenhuma arquitetura ou empresa.

Devido a suas características praticamente tudo pode ser feio em Java, sistemas críticos, sistemas que precisam de velocidade, bancários e até mesmo sistemas que vão ser executados fora do planeta como a sonda Spirit enviada pela Nasa (Agencia Espacial Norte Americana) para Marte.

2.2. PROGRAMAÇÃO ORIENTADA A OBJETOS (POO)

O termo programação orientada a objetos foi criado por Alan Key, autor da linguagem de programação Smaltalk, porém antes mesmo do termo ser conhecido a ideia já era empregada na linguagem Simula 67 de 1967 e hoje praticamente todas as linguagens mais populares adotaram a orientação a objetos.

Dentre os motivos que influenciaram o desenvolvimento da POO estão o avanço na arquitetura dos computadores que passaram a suportar ambientes com interfaces de programação mais sofisticados e avanços nas linguagens de programação.

O objetivo da POO é aproximar o mundo real do virtual, tentar moldar objetos no computador e fazê-los interagir entre si através de mensagens (David, 2007).

A POO baseia-se nos conceitos de Classe, Objeto, Herança e Polimorfismo. A seguir algumas das principais vantagens no uso da Programação Orientada a Objetos:

- Reutilização de código;
- Maior aproximação do modelo do mundo real;
- Facilidade de manutenção.

2.3. JDK (*JAVA DEVELOPMENT KIT*)

O JDK é o kit de desenvolvimento Java, disponibilizado para download gratuitamente, contendo utilitários que promovem recursos para desenvolver aplicativos Java e é composto por compilador e bibliotecas.

Atualmente a JDK esta na versão 7 e possui gerenciadores de perfis como o JConsole e a Visual VM, que será explorada nesse trabalho, além de diversos utilitários em linha de comando para monitoramento de desempenho de aplicativos Java, que podem ser desconhecidos para a maioria dos programadores. Esses utilitários encontram-se no diretório “/bin” onde a JDK é instalada.

A seguir uma breve descrição de alguns desses utilitários.

2.3.1 JSTAT

O utilitário jstat reúne diversas informações sobre o aplicativo Java de acordo com a opção informada, em sua maioria sobre o coletor de lixo do Java.

2.3.2. JMAP

O jmap imprime mapas de objetos compartilhados de memória ou detalhes de memória *heap* de um processo em um determinado momento e cria outro arquivo que pode ser do tipo texto ou binário com as informações coletadas, processo chamado *Heap Dump.* O *heap* é a área da memória alocada pela JVM para rodar a aplicação Java.

Este utilitário é usado quando uma aplicação está consumindo muita memória, ou a memória utilizada não esta sendo liberada como deveria. Obter informações sobre o *heap* e observar oque há lá ajuda a resolver problemas relacionados a memória.

O jmap é usado para capturar as informações sobre a memória e para analisar os dados capturados é necessário outro aplicativo como a Visual VM ou o utilitário jhat descrito na próxima seção. Neste trabalho será utilizado o jmap para realizar o *Heap Dump* para posterior análise na Visual VM, todo o processo será descrito em detalhes no Capítulo 4.

2.3.3. JHAT

O jhat é utilizado para analisar o arquivo binário criado depois de executar o *Heap Dump*. O utilitário cria um servidor HTTP/HTML que pode ser explorado em um navegador e fornece informações objeto a objeto que está na memória *heap*.

Este utilitário suporta consultas em linguagem OQL para facilitar a análise.

Para usar esses utilitários é necessário ter conhecimento do PID (*Process Identificator*), número do processo do sistema operacional correspondente ao aplicativo Java em execução que deseja monitorar.

2.4. BANCO DE DADOS HSQLDB

Um banco de dados são conjuntos de registros dispostos em uma estrutura regular, no caso dos Banco de Dados Relacionais, tabelas.

O HSQLDB é um banco de dados relacional escrito 100% em linguagem Java e funciona inteiramente na memória principal (*In-Memory Database*). Segundo o *site* do desenvolvedor, o HSQLB é usado em milhares de *softwares open source* e suporta uma ampla gama de recursos do SQL.

Os bancos de dados *in-memory* são uma preocupação a mais para o programador, que além do consumo de memória da aplicação em si, também terá que se preocupar com o consumo de memória do banco de dados, pois todas as suas tabelas e registros são carregadas na memória principal.

Bancos de dados *in-memory* oferecem uma grande vantagem de desempenho, tecnologias *in-memory* chegam a ser 20% a 30% mais rápidas.

2.5. JDBC (*JAVA DATABASE CONNECTIVITY*)

JDBC é uma API incluída dentro da linguagem Java para o acesso a banco de dados. Consiste em um conjunto de classes e interfaces escritas em Java que oferecem uma completa API para a programação com banco de dados (GONÇALVES, 2007).

Por ser escrito 100% em Java, o JDBC também é independente de plataforma.

2.6. HIBERNATE

Os bancos de dados relacionais sem dúvida constituem o núcleo da corporação moderna, enquanto as linguagens mais modernas como o Java fornecem uma visão intuitiva e orientada para objetos. Muitas tentativas foram feitas para criar uma ponte entre a tecnologia relacional e a orientada para objetos e é esse o desafio que o Hibernate assume através do mapeamento Objeto/Relacional (ORM) (BAUER & KING, 2007).

Sendo um dos mapeadores de banco de dados mais populares do mercado, o Hibernate é um *framework* para persistência de dados que deixa o desenvolvedor livre para se concentrar nos problemas da lógica de negócio. O Hibernate é de simples configuração e facilmente incorporado a qualquer sistema Java, comunicando a aplicação com o banco de dados como se fosse feito para a aplicação (GONÇALVES, 2007).

2.7. JPA

O Hibernate é uma ferramenta de mapeamento objeto/relacional como visto na seção anterior e sua API nativa é projetada pelos seus desenvolvedores.

JPA (*Java Persistence API*) é uma API padrão do Java para persistência de dados que deve ser implementada por *frameworks* que queiram seguir o seu padrão.

Diversos *frameworks* de ORM como o Hibernate implementam a JPA.

2.8. JAVASERVER FACES

O JavaServer Faces (JSF) é uma tecnologia visual para construção de páginas WEB através de componentes.

O JSF incorpora características de um *framework* MVC (*Model View Controller*) para WEB e um modelo de interfaces gráficas baseada em eventos. O MVC divide a aplicação em três camadas que serão descritas a seguir.

O *Model* (modelo) representa os objetos de negócio e é responsável pelo controle e acesso a dados. A *View* (visualização) representa a interface com o usuário e é responsável por encaminhar os dados para a camada *Controller.*

O *Controller* (controle) é responsável pela ligação entre a camada de modelo e visualização.

2.9. APACHE TOMCAT

Para que o Java funcione em aplicações WEB é necessário um *Container Servlet* que pode ser um servidor de aplicativos WEB.

O Apache Tomcat é um servidor de aplicações Java para web desenvolvido pela Apache Software Foundation no renomado projeto Apache Jakarta.

O Tomcat funciona como um servidor de aplicações web ou como um c*ontainer servlets* centrado nas tecnologias de Servlets e JSPs (JavaServer Pages). Ele é

responsável basicamente por interpretar o código Java e transformá-los em HTML para que sejam apresentados pelo *browser*.

3. JAVA E MEMÓRIA

O Java é famoso por possuir gerenciamento de memória automático, o *Garbage Collector* procura por áreas de memória inutilizadas por um programa e as recupera utilizando complexos algoritmos. Essas áreas de memórias podem ser variáveis, objetos e blocos que não podem mais ser referenciados.

Gerenciar memória de forma automática significa que o programador não precisa se preocupar com suas tarefas anteriores, variáveis e objetos criados e que já foram utilizados serão em algum momento identificados e retirados da memória pelo *Garbage Collector*, porém, diferentemente do que a maioria dos programadores pensam o *Garbage Collector* não resolve todos os problemas de memória da aplicação, as soluções foram projetadas para os casos mais comuns e principalmente para os programadores das boas práticas, é preciso tomar cuidado com práticas de programação que podem fazer a aplicação aumentar drasticamente o consumo de memória e apresentar erros por falta dela devido aos vazamentos de memória conhecidos como *memory leaks*.

O programador Java está acostumado a não se responsabilizar pela memória consumida pela aplicação por dois motivos, pelo avanço do hardware e pelo famoso *Garbage Collector* que evita, na medida do possível, que ocorram vazamentos de memória, os *memory leaks,* e assim o programador não precisa fazer esse processo manualmente como é o caso de linguagens de programação como o C e C++ onde deve-se especificar explicitamente quando e quais objetos devem ser desalocados da memória.

Segundo o artigo “*Memory leak* em Java” sobre boas práticas de programação publicado no SITE JAVAFREE.org os vazamentos de memória podem ser de dois tipos:

1. Blocos de memória estão alocados e disponíveis mas não podem ser acessados pois a aplicação não tem nenhum ponteiro apontando para essa área da memória.
2. Blocos de memória possuem dados que poderiam ser desalocados mas são inacessíveis pois ainda estão referenciados no código da aplicação.

O primeiro caso é muito comum em C e C++ quando o programador aloca um espaço na memória utilizando uma função chamada *malloc* e em seguida ele faz com que o ponteiro para essa área de memória aponte para outro lugar e assim perdendo a referencia inicial. Isso não acontece em Java pois o *Garbage Collector* é capaz de identificar esse tipo de situação e liberar a memoria.

Já o segundo caso é o que acontece em Java, basicamente quando uma aplicação apresenta *memory leaks* significa que objetos que não deveriam estão permanecendo na memória, fazendo o consumo de memória pela aplicação aumentar.

Aplicações Java que consomem muita memória comprometem seu desempenho e afetam a qualidade do software, podendo apresentar erros por falta dela, nesses casos o programador pode fazer um *Profiling* da aplicação, que consiste em tirar uma “foto” do *heap* em tempo de execução para ver o que há lá. Utilizando *Profiling* é possível identificar objetos que deveriam ter sido retirados da memória pelo *Garbage Collector* mas por algum motivo estão permanecendo lá e assim o programador poderá corrigir seu código devidamente para que esses objetos possam ser liberados da memória. Essa técnica será abordada passo a passo no Capítulo 4.

Softwares com consumo excessivo de memória podem apresentar lentidão e afetar a performance, nas próximas seções, alguns recursos necessários para entender o gerenciamento de memória do Java.

3.1. JVM

A JVM (*Java Virtual Machine*) ou Máquina Virtual Java é o *software* onde são executados os aplicativos Java, ela isola a aplicação do contato direto com o hardware funcionando como uma máquina imaginária, isso garante a conhecida portabilidade e segurança do Java. Qualquer dispositivo que possuir uma JVM instalada poderá executar qualquer aplicação escrita em Java.

Depois que um programa é compilado pelo JDK (*Java Development Kit*) em *bytecodes* ele pode ser executado pela JVM. Os *bytecodes* apresentam uma codificação binária universal dispostos em aquivos de acordo com a classe compilada com extensão *.class* e são interpretados pela JVM ou compilados em linguagem de máquina. A JVM possui uma ferramenta chamada javap para visualização do conteúdo de arquivos *.class*.

As máquinas virtuais buscam adaptar-se o melhor possível aos ambientes onde serão executadas as aplicações e isso funciona bem na maioria dos casos, mas se a aplicação for muito grande, consumir muita memória ou estiver em um ambiente atípico, se for uma aplicação WEB com milhares de acessos simultâneos, a configuração padrão da máquina virtual pode mostrar-se inadequada.

Para melhorar a performance da aplicação fazendo ajustes na máquina virtual ou verificar porque uma aplicação está consumindo muita memória é necessário conhecer as áreas de memória da JVM e como ocorre a alocação de recursos.

3.1.1 ÁREAS DE DADOS

Existem diversas estratégias para alocação de memória, algumas delas são:

- Alocação estática;
- Alocação linear;
- Alocação dinâmica.

A alocação estática é para dados que possuem tamanho fixo e estão organizados de modo sequencial. Na alocação linear a memória fica alocada em fila ou pilha e na alocação dinâmica o tamanho dos dados não é fixo.

O Java utiliza a alocação dinâmica para objetos, área de memória chamada *heap* e alocação linear para procedimentos sequenciais (pilha). Todo o gerenciamento é feito automaticamente pela JVM, ela não oferece opções ao programador para alocação no *heap* ou pilha.

As próximas seções foram baseadas no trabalho de Helder Rocha, um tutorial sobre gerenciamento de memória.

Para executar um programa a JVM define várias áreas da memória:

- Registradores
- Pilhas e Quadros
- *Heap* e área de métodos

3.1.1.1. REGISTRADORES

Threads é a forma de um processo se dividir em duas ou mais tarefas. O Java foi a primeira linguagem a incluir explicitamente o conceito de *Threads* em sua linguagem. *Threads* possuem suas próprias variáveis locais e seu próprio PC (P*rogram Counter*) e quando são de um mesmo processo compartilham memória.

Todo programa Java possui pelo menos uma *thread*, o método *main()*. A JVM possui várias *threads* que realizam tarefas como a coleta de lixo e finalização de objetos. Basicamente cada *thread* executa o código de um único método de um programa Java que são uma lista de instruções. Os registradores contém o endereço de memória da instrução da JVM que está sendo executada.

3.1.1.2. PILHAS E QUADROS

Cada *thread* possui sua própria pilha na JVM. A pilha é criada junto com a *thread,* ela armazena em quadros (*frames*) variáveis locais e resultados parciais de

métodos. Os Quadros ainda lidam com invocação e retorno de métodos e são criados toda vez que um método é executado.

A JVM permite que a pilha possa ter tamanho fixo ou dinâmico, pode ser expandida ou reduzida de acordo com a necessidade do programa. Essa memória pode ser alocada no *heap* e é liberada automaticamente depois que o método termina.

3.1.1.3. HEAP

O *heap* é a área de memória onde todos os dados são alocados e é compartilhado por todas as *threads*. O heap pode ter tamanho fixo ou não, podendo ser expandido e reduzido dinamicamente, ele é alocado quando a máquina virtual é iniciada e seus espaços alocados podem ser reciclados pelo coletor de lixo. Diferentes máquinas virtuais podem oferecer controles para ajustar seu tamanho inicial, máximo e mínimo de acordo com a necessidade do programa.

Outra área de memória particular da JVM é a chamada *permanent generation ou Perm Gem* onde são alocados objetos internos da JVM, objetos Class, Method além do pool de strings (SILVEIRA et.al, 2012).

Essa área de memória contém objetos que raramente são retirados da memória.

3.1.2. CRIAÇÃO E DESTRUIÇÃO DE OBJETOS

Para entender como ocorre a alocação e liberação de memória na JVM é preciso entender como acontece a criação e destruição de objetos.

Quando instancia-se um objeto a JVM automaticamente aloca memória no *heap* com todas as variáveis declaradas na classe e na superclasse e cria uma referencia desse objeto na pilha, lançando um erro caso não haja memória suficiente. Objetos são criados através da expressão *new Classe()* e suas variáveis são inicializadas com valor *default.*

A destruição de objetos é feita pelo *Gargage Collector* através da liberação de blocos de memória que não estão mais sendo utilizados ou não são mais referenciados no código. Para a destruição de objetos o programador possui poucas opções já que o processo é feito de modo automático, são elas:

- Finalização de objetos;
- Sinalização de código para tentativas de execução do coletor de lixo;
- Remoção da referencia de um objeto.

Antes que qualquer objeto seja removido da memória é chamado um finalizador através de uma *thread,* o método *finalize().* O método *finalize()* é opcional ao programador já que ele é chamado automaticamente quando um objeto perde referencia na aplicação.

Alguns cuidados devem ser tomados no uso do método *finalize(),* sua forma correta de uso é através do finalizador da superclasse *super.finalize()*.

O segundo caso é a sinalização de código para tentativas de execução do coletor de lixo. Áreas do código que o programador acha que há memória a ser liberada podem ser marcadas através do método *System.gc()*. Os algoritmos de coleta de lixo então analisarão o local com possíveis casos de objetos que não serão mais usados identificados pelo programador para liberar memória. A chamada explícita ao coletor de lixo porém, não garante que a memória seja efetivamente liberada.

O último caso é a remoção de referencia para um objeto, isso pode ser feito fazendo o objeto em questão receber o valor *null,* assim ele ficará visível ao coletor de lixo.

3.2. GARBAGE COLLECTOR

O *Garbage Collector* é o coletor de lixo do Java, ele procura por objetos que não são mais referenciados liberando a memória utilizada por eles. Não é possível prever quando o coletor de lixo entrará em ação, nem mesmo quando o programador fará chamadas explícitas a ele.

O *Garbage Collector* possui suas vantagens, várias delas já foram vistas anteriormente:

- Faz o gerenciamento de memória automaticamente;
- Reduz as chances de *memory leaks;*
- Reduz as chances de esgotamento da memória.

Porém, é importante ressaltar que o famoso coletor de lixo tem suas desvantagens, os processos e estratégias para identificação de áreas de memória que podem ser liberadas consomem muito recurso computacional, ou seja, podem impactar na performance da aplicação. Além disso muitas chamadas ao coletor de lixo pode comprometer a eficiência do algoritmo.

A liberação de memória *heap* é feita através de algoritmos de coleta de lixo, eles tem o desafio de identificar e tomar a decisão do que é lixo e pode ser liberado e o que não é. As JVMs atuais possuem algoritmos precisos para essa coleta, cujo principal critério para definir o que é lixo é a alcançabilidade.

3.3. CONFIGURAÇÕES DE MEMÓRIA DISPONÍVEIS

Existem diversos algoritmos para coleta de lixo, alguns que podemos citar são: Mark-And-Sweep, generational copying, Serial Collector, Parallel Collector e G1.

Cada um deles tem suas particularidades, como uso do processador, threads e velocidade de execução. A JVM possibilita ao programador sinalizar qual algoritmo de coleta quer utilizar em sua aplicação através de parâmetros de comandos da JVM para adequar o melhor algoritmo para cada ambiente de produção.

Porém o mais importante é saber adequar a aplicação ás hipóteses de geração, adequando o código ou conhecendo melhor as configurações de execução da JVM.

Além disso, não interferir nas coletas de lixo e não depender de seu comportamento imprevisível é uma das melhores boas práticas.

Outra possibilidade, e muito utilizada, é aumentar o tamanho da memória alocada pela JVM afim de evitar erros de memória ou como tentativa de resolver um erro

ocorrido por falta dela. Essa medida pode resolver o problema apenas temporariamente. Se a aplicação está consumindo memória excessivamente é porque a memória utilizada não está sendo coletada, aumentar o tamanho da memória só serviria para “mascarar” o problema por um tempo, não o resolveria. O mais correto é verificar os possíveis *memory leaks.*

4. *PROFILE*

Quando a aplicação Java está consumindo muita memória, apresenta erros por falta dela ou lentidão, uma boa opção para ajudar a corrigir o problema é fazer *Profile* da aplicação. Essa técnica conhecida como *Proffiling* consiste em verificar o comportamento de uma aplicação Java através de um *Profiler*.

Profilers são ferramentas que disponibilizam visualmente dados sobre aplicações Java como utilização de memória, uso de CPU, *threads* e objetos que estão vivos na memória em tempo de execução. Existem diversos *Profilers open source,* alguns deles:

- Visual VM
- Cougar Memoty Profiler
- JTreeProfiler
- NetBeans Profiler
- JRat
- JMP
- TomcatProbe
- Allmon
- Perf4j
- JBoss Profiler

Também existem outros *Profilers* proprietários como o JProfiler. O Visual VM é disponibilizado junto com a distribuição da JDK e será utilizado para estudos de caso de boas e más práticas de programação nessa pesquisa e assim coletar os dados das duas implementações para análise. Outra ferramenta disponível nas

distribuições da JDK para monitoramento de aplicações Java é o JConsole. A Figura 2 mostra uma visão geral do JConsole.

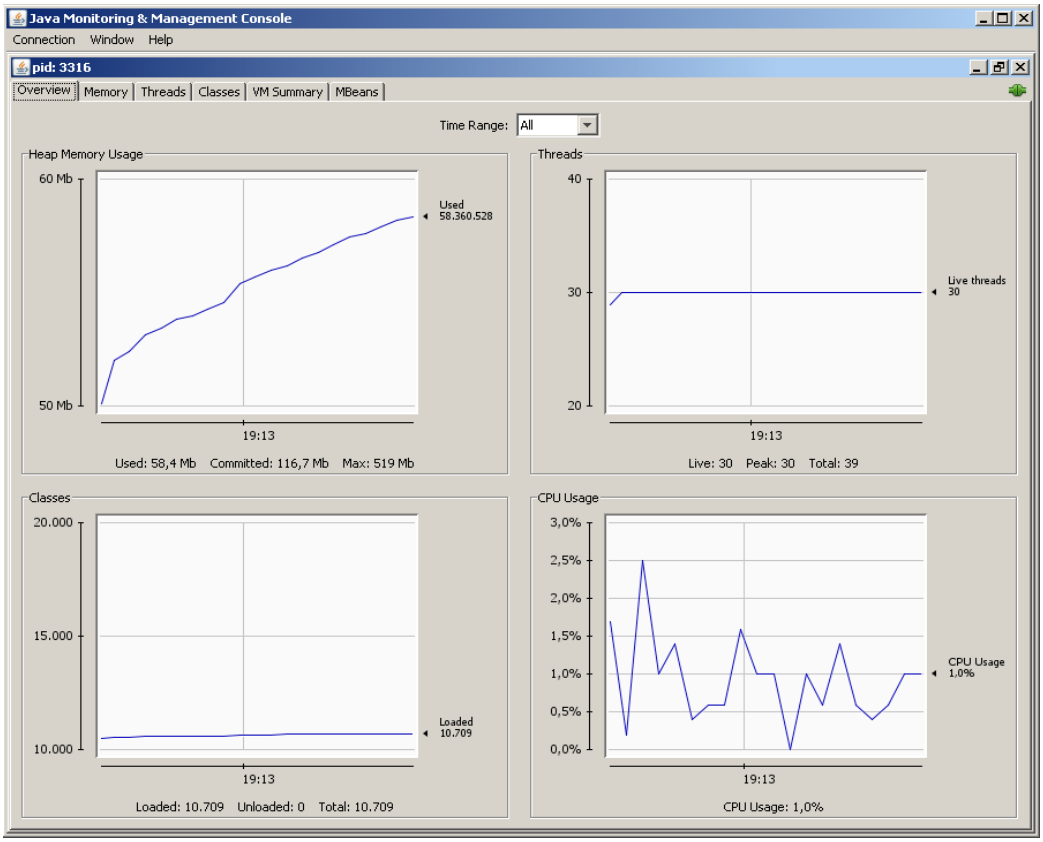


Figura 2 - JConsole

O JConsole monitora o uso da memória hea, uso de CPU, quantidade de threads e classes de uma aplicação.

4.1. *PROFILING* COM VISUAL VM

O JConsole permite fazer o monitoramento de um processo Java, já a Visual VM possui um recurso muito importante adicional bem como os outros *Profilers* citados, o *Profiling*, que permite analisar detalhadamente um processo Java pelo uso de CPU e pelo uso de memória. A Figura 3 mostra a tela do Visual VM.

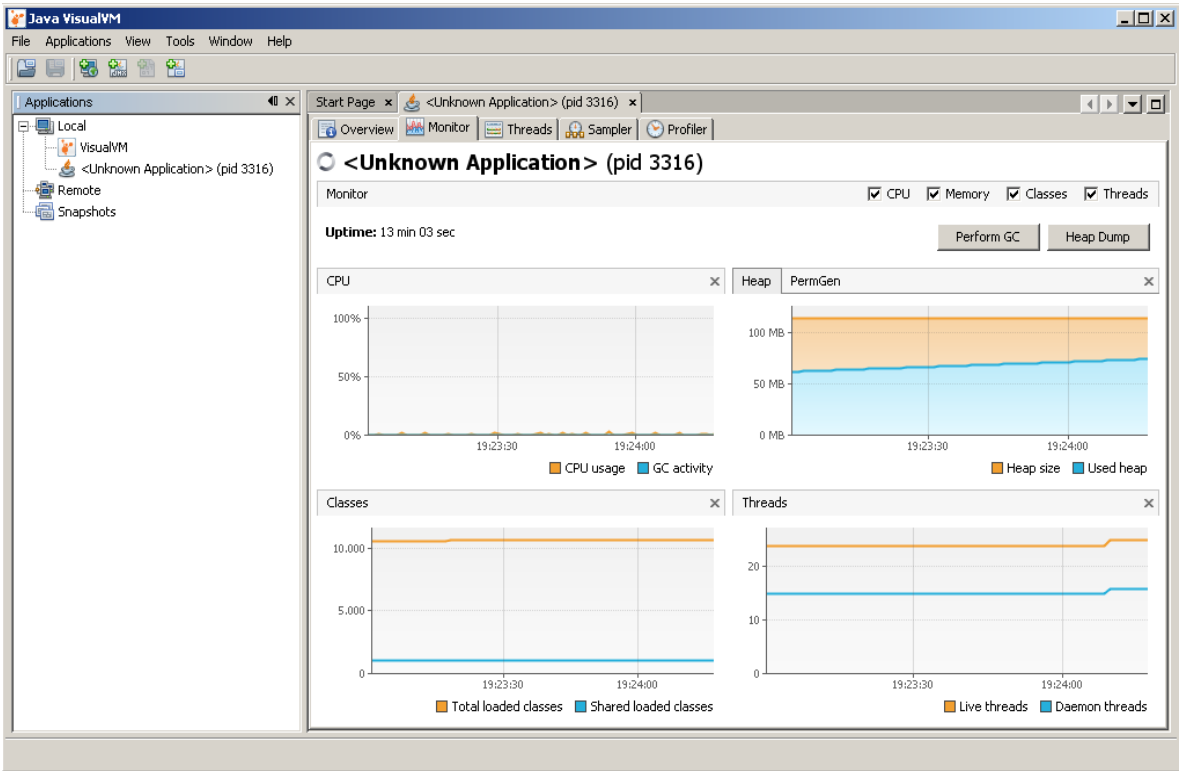


Figura 3 – Tela do Visual VM

A tela do Visual VM possui duas partes: *Aplications* e as abas dos processos. Na primeira parte, *Aplications*, ficam as aplicações Java que estão sendo executadas naquele momento localmente, também é possível adicionar processos para serem monitorados remotamente. Na segunda parte ficam as abas das aplicações que estão sendo analisadas com a Visual VM.

Para melhor descrever como analisar e coletar dados de aplicações através da Visual VM, foi desenvolvido uma aplicação Java chamada “EstudoMemoria” utilizando o Eclipse IDE contendo duas classes, TelaPrincipal.java e Cliente.java, e um pacote chamado “estudoProfile” conforme mostra a Figura 4.

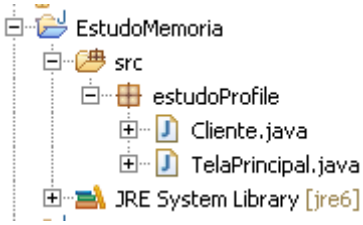


Figura 4 – Estrutura do projeto

A aplicação simula um cadastro de Clientes, a Figura 5 mostra a tela da aplicação.

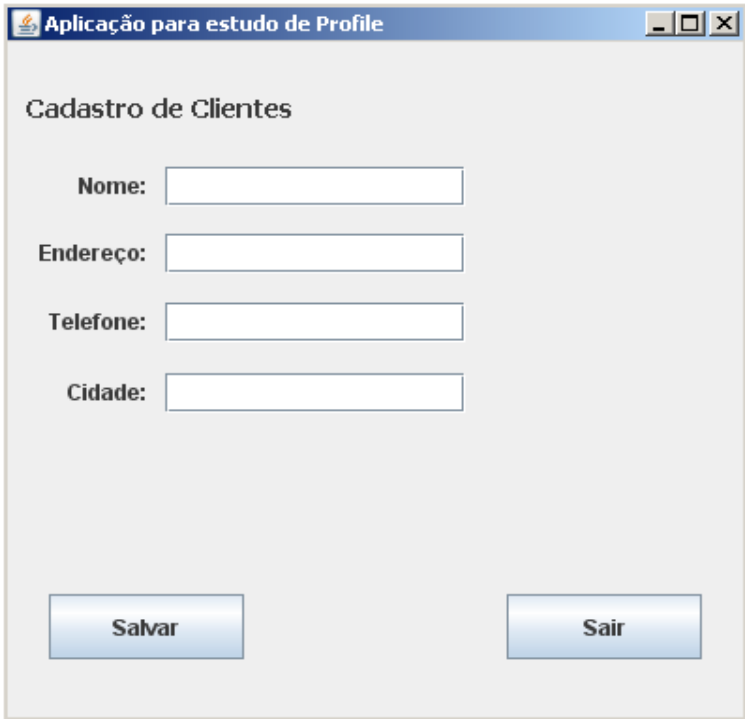


Figura 5 – Tela da aplicação

Quando esta aplicação é executada o sistema operacional atribui um PID (*Process Identificator*) para ela, o PID é o numero de identificação desse processo. Quando está em execução a aplicação já é reconhecida pela Visual VM (Figura 6), para iniciar o monitoramento basta clicar na aplicação.

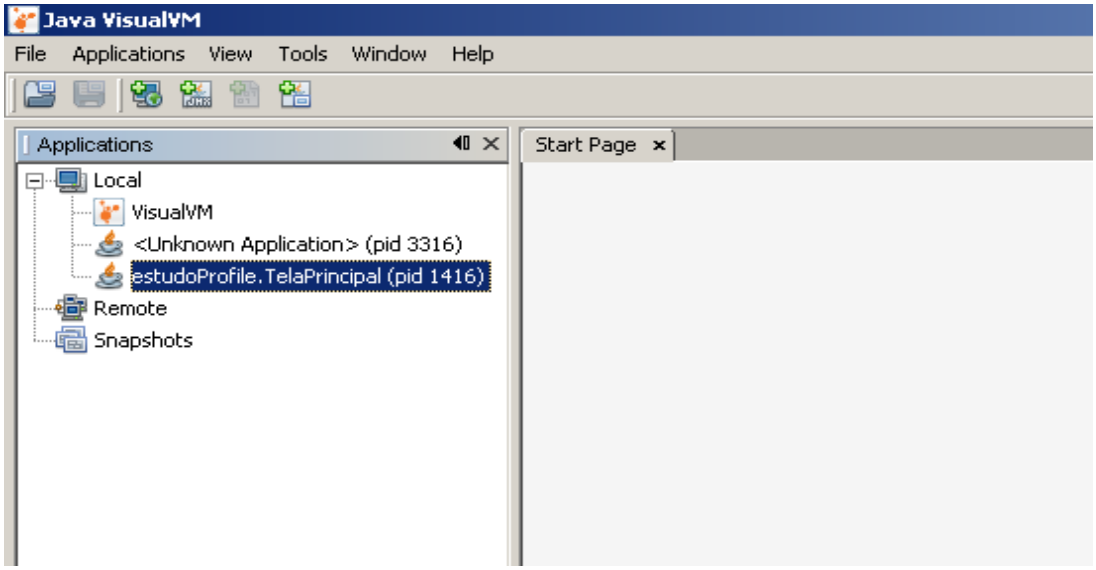


Figura 6 – Aplicação na Visual VM

A primeira aba da Visual VM mostrada é a *Overview,* nela se encontram informações sobre a aplicação como o PID, nome da classe principal precedido pelo nome do pacote, nesse caso “estudoProfile.TelaPrincipal”, entre outras informações como versão da JDK e localização da JVM (Figura 7).

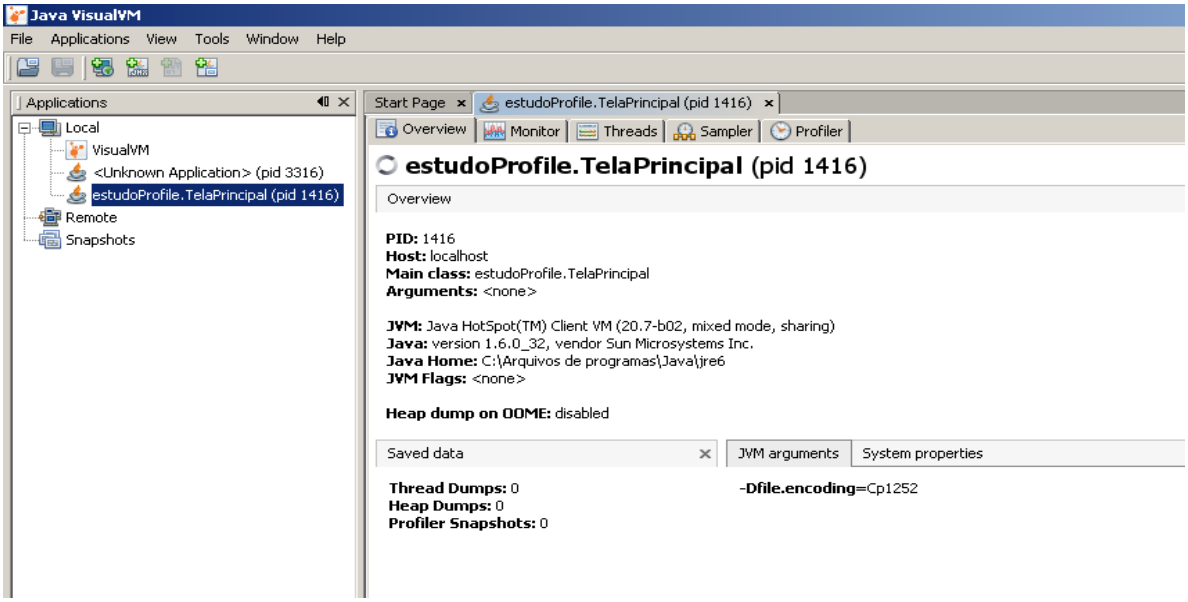


Figura 7 – *Overview*

A visão *Monitor* fornece quatro gráficos para monitoramento do comportamento de uma aplicação:

- Gráfico do uso de CPU.
- Gráfico com a quantidade de classes instanciadas.
- Gráfico de *threads*.
- Gráfico com o uso do *Heap* e *Permgen*.

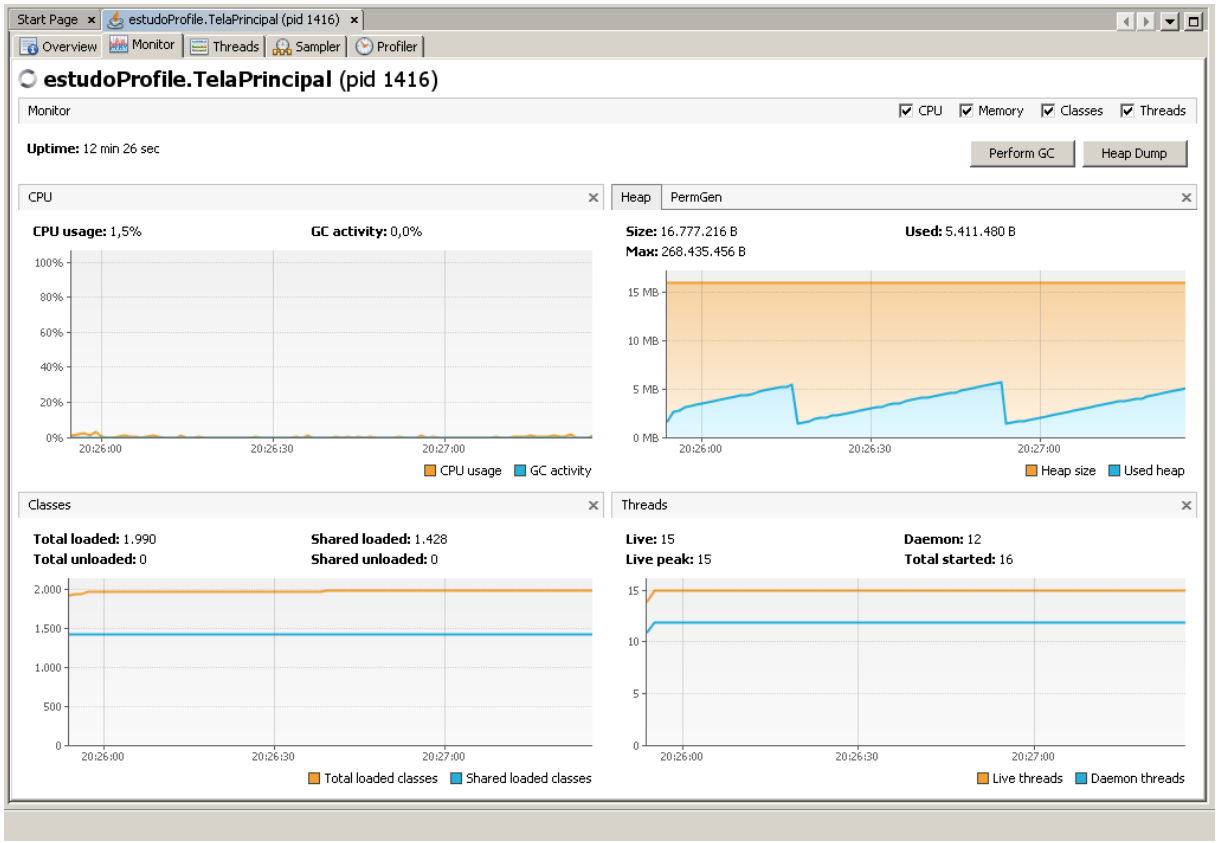


Figura 8 – *Monitor*

Nessa aba é possível identificar se uma aplicação Java pode estar consumindo muita memória através do gráfico do uso de Heap. Nesta visão também pode ser visto o percentual de atividade do Garbage Collector.

A visão *Threads* mostra as *threads* da aplicação e da JVM (Figura 9).

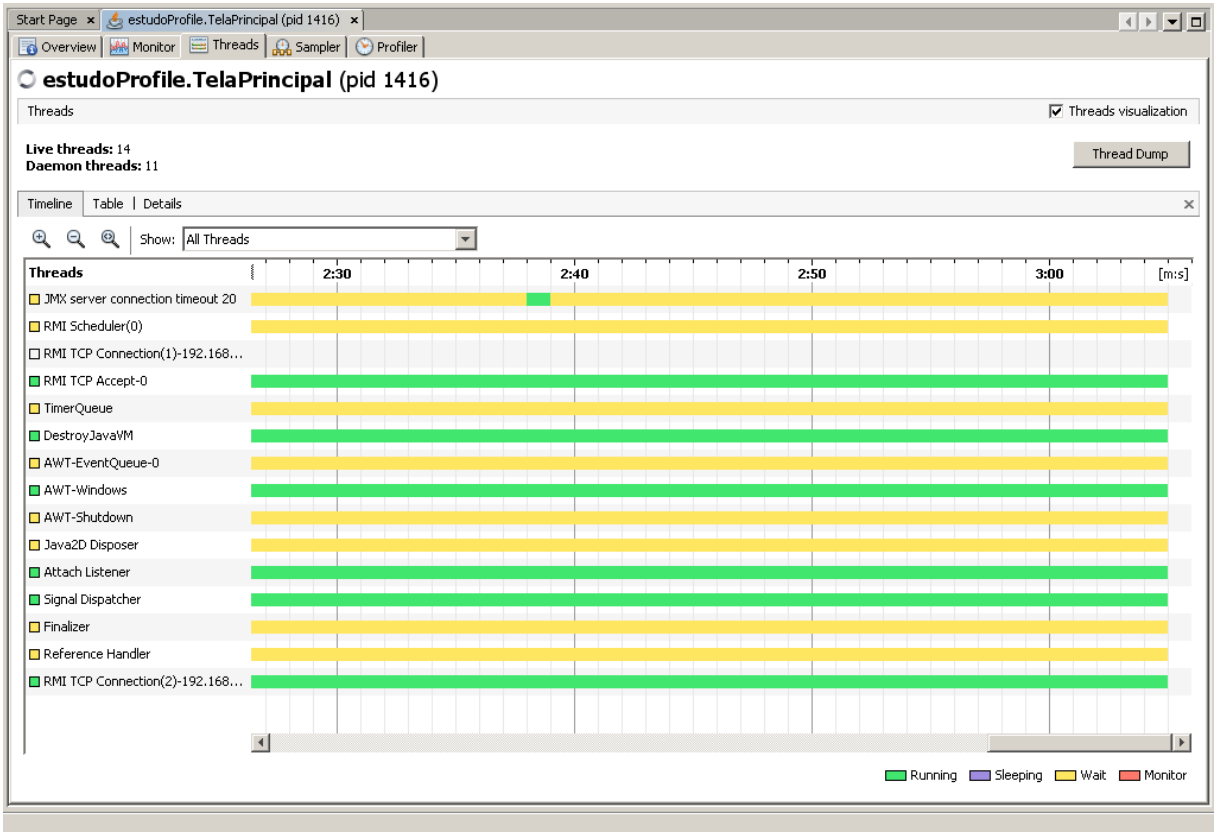


Figura 9 – *Threads*

Na visão *Profiler* é possível observar o processo em questão de duas formas, pelo uso de CPU (Figura 10) e pelo uso de memória (Figura 11). A visão de CPU permite analisar quais métodos do aplicativo estão ocupando mais tempo do processador, percentual em relação aos demais e a quantidade de chamadas de cada um.

Na visão *Memory é* possível verificar o consumo de memória por classe carregada, quantidade de instancias e a quantidade de bytes de cada instancia.

Através da análise de memória da visão *Profiler* é possível observar quantas instancias de cada tipo de objeto está viva no *heap* naquele momento já que a análise é feita em tempo de execução.

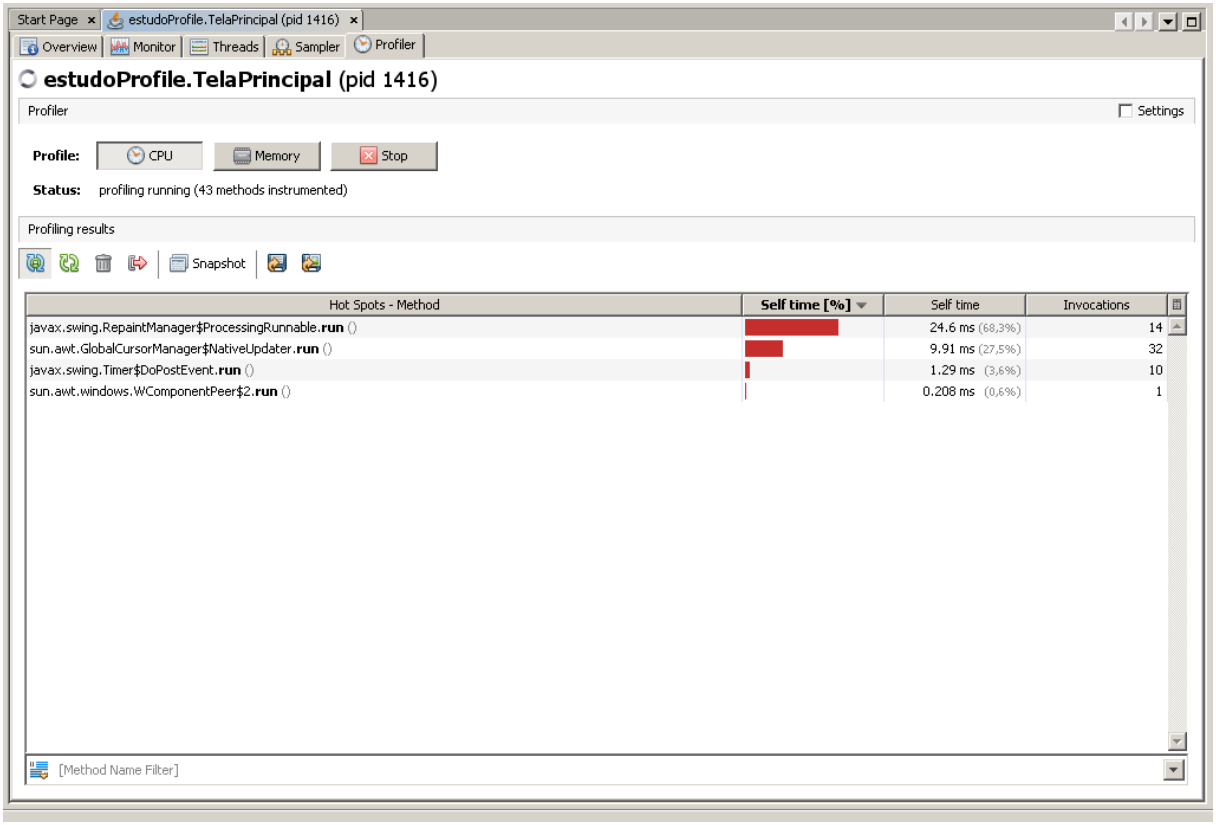


Figura 10 – Uso de CPU

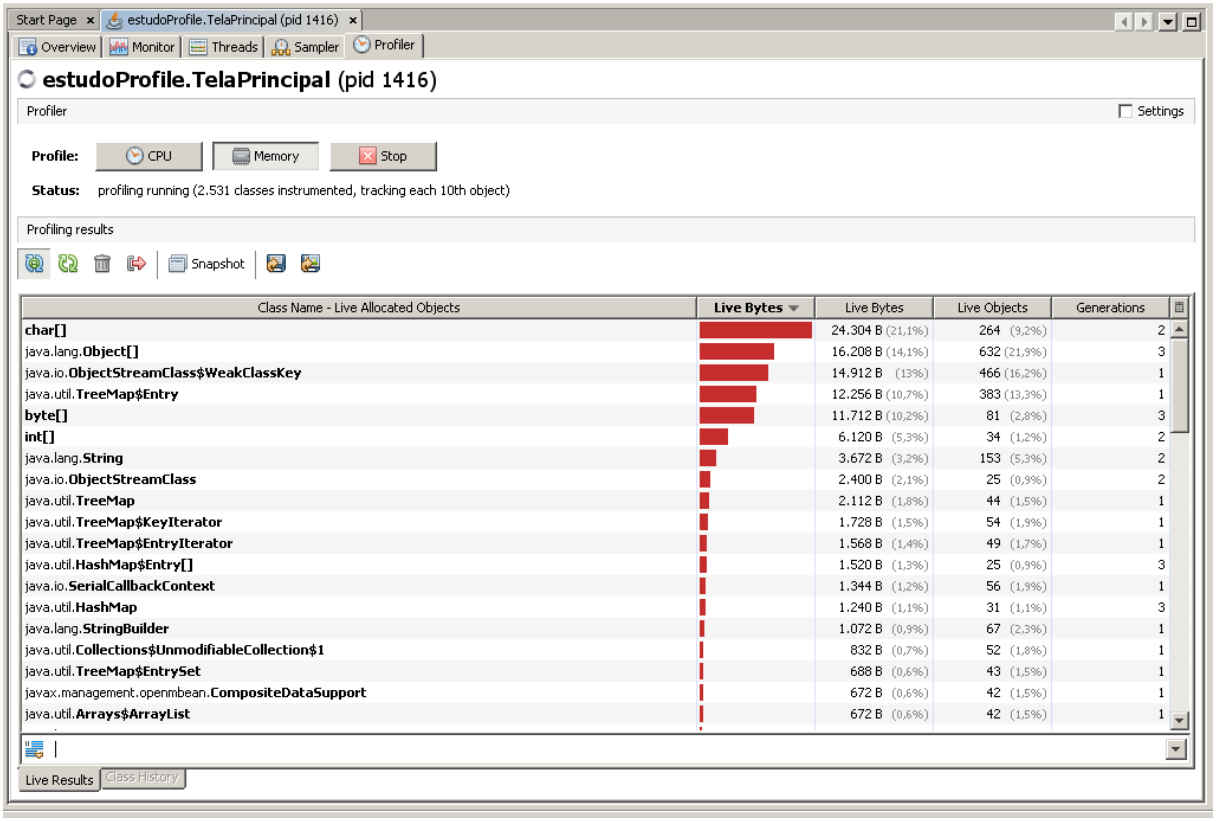


Figura 11 – Uso de memória

Utilizando a Visual VM é possível fazer uma análise de memória ainda mais detalhada realizando um *Heap Dump.* Essa operação grava em um arquivo um mapa de toda a memória *heap* de uma aplicação em execução no momento do *Heap Dump* e pode ser analisada através da Visual VM. O *Heap Dump* pode ser realizado de duas formas:

- Através da própria Visual VM clicando no botão *"Heap Dump";*
- Através do utilitário jmap em linha de comando;
- Através de geração automática do arquivo no momento de um erro.

Para fazer um *Heap Dump* através da Visual VM é preciso clicar no botão *"Heap Dump"* na visão *Monitor.* Essa operação abre uma nova aba parecida com a *Profiler* porém com mais recursos: *Sumary* (Resumo), *Classes*, *Instances* (Instancias) e *OQL Console* (Figura 12). A operação também cria o arquivo com as informações da memória *heap*, é possível salvá-lo através da Visual VM.

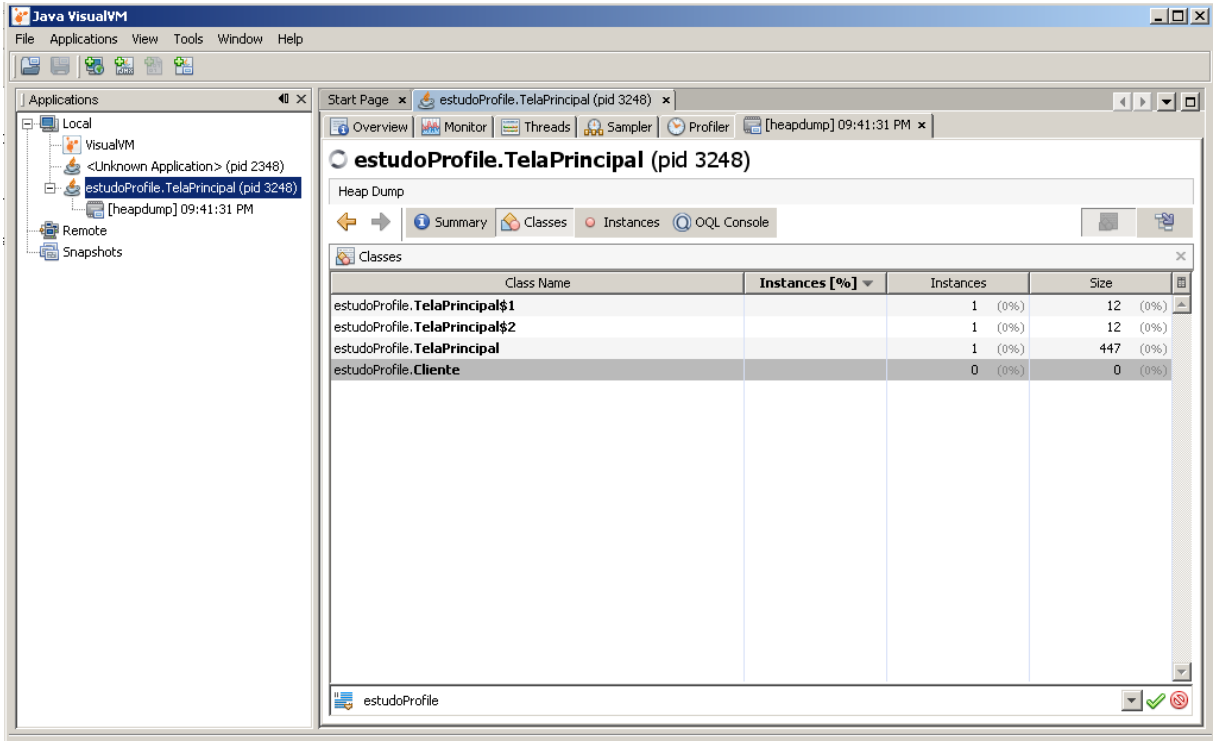


Figura 12 – *Heap Dump*

O painel Resumo exibe informações do momento que o *Heap Dump* foi feito como tamanho do arquivo, localização, quantidade de classes, instancias que faltam ser finalizadas e informações sobre o sistema operacional e JVM.

O painel Classes é parecido com o da visão *Profiler* porém ao dar um duplo clique no nome da classe é possível visualizar informações de todas as instancias existentes daquela classe e navegar por toda a hierarquia de objetos associados, verificar seus valores e referencias.

A Figura 13 ilustra as informações da classe “TelaPrincipal.java” do *Heap Dump* da aplicação “EstudoMemoria”.

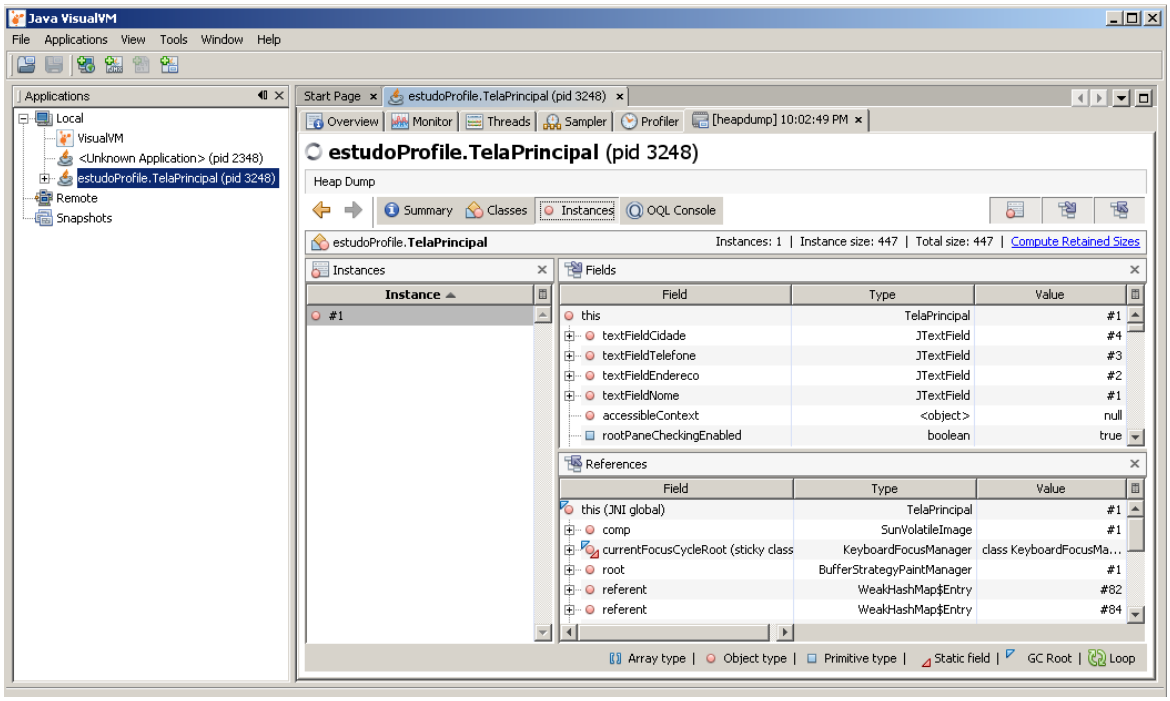


Figura 13 – *Heap Dump* da aplicação “EstudoMemoria”

Além disso clicando no link *Computer Retained Sizes* (Calcular tamanhos retidos) que pode ser visto na imagem anterior, a Visual VM irá calcular para todas as instancias o tamanho em bytes que estavam ocupando no *heap.*

Este recurso é interessante para identificar quais objetos podem estar fazendo a aplicação consumir muita memória.

A segunda forma de fazer *Heap Dump* é criar o arquivo contendo o mapa dos objetos contidos no *heap* através do utilitário jmap e depois carregá-lo na Visual VM para análise ou analisa-lo com o utilitário jhat da JDK. Como visto anteriormente, o *jmap* é um utilitário disponível na distribuição da JDK que imprime mapas de

memória *heap* e cria um arquivo que pode ser binário ou texto com as informações, como se tirasse uma “foto” do *heap*.

A sintaxe do comando jmap pode ser:

- No Windows: jmap -dump:format=b,file=NomeDoArquivo.hprof <pid>
- No Unix: jmap -heap:format=b <pid>

A instrução *format=b r*efere-se a criação de um arquivo binário, *file* ao nome do arquivo e o <pid> é o PID do processo em questão. Depois de criado o arquivo é preciso carregá-lo na Visual VM para análise. A Figura 14 ilustra a criação do arquivo de *Heap Dump* da aplicação “EstudoMemoria” no Windows e a Figura 15 no Linux.

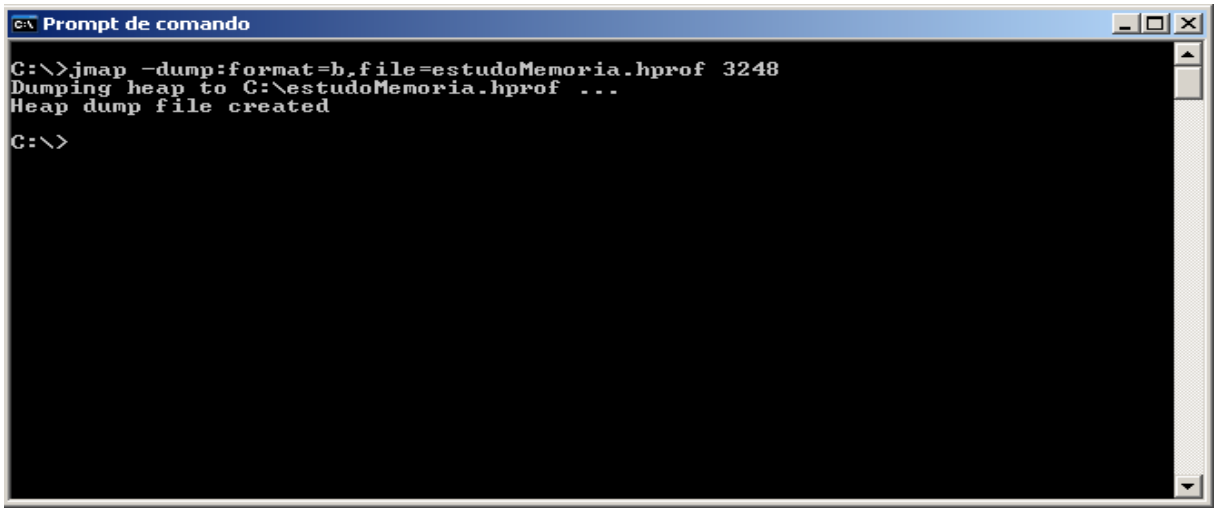


Figura 14 – *Heap Dump* com JMAP no Windows

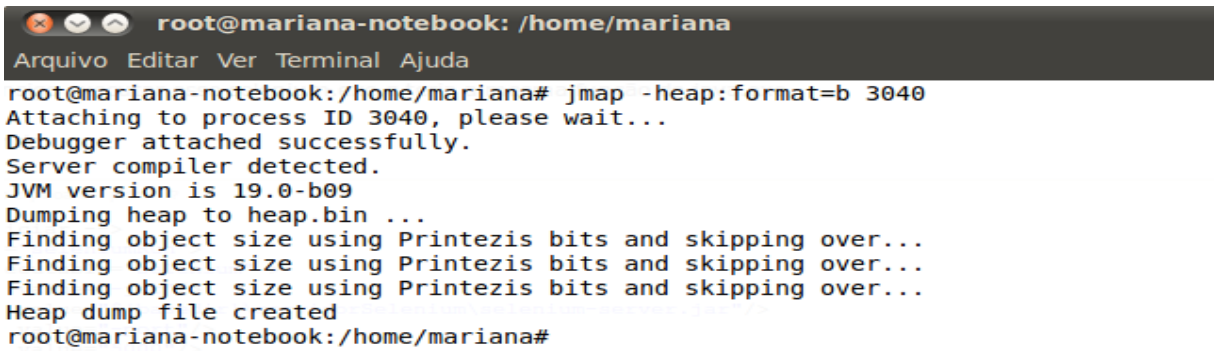


Figura 15 – *Heap Dump* com JMAP no Linux

5. O IMPACTO DAS BOAS PRÁTICAS DE PROGRAMAÇÃO NO GERENCIAMENTO DE MEMÓRIA

Apesar dos mecanismos para coleta de lixo, aplicações Java podem apresentar consumo excessivo de memória e isso é sempre causado por más práticas de programação.

5.1. EXCEÇÕES E ERROS

Uma exceção é um evento que ocorre durante a execução de um programa, que interrompe o fluxo normal de instruções de um programa (ALLES, 2005).

As exceções são instâncias de classe. A raiz de todas as exceções é a classe java.lang.Throwable que possui duas subclasses:

- *java.lang.Exception:* É a raiz de classes derivadas da *Throwable* que indica situações que o programa pode capturar o erro que ocorreu, sua causa e realizar um tratamento através de blocos *try/catch.*
- j*ava.lang.Error:* É a raiz das classes derivadas de Throwable que indica situações que a aplicação não deve tentar tratar. Usualmente indica situações anormais, que não deveriam ocorrer (ALLES, 2005).

5.2. ERROS DE MEMÓRIA

Quando falta memória em um programa Java dois erros podem ocorrer e estão relacionados a operações na pilha e espaço no *heap*:

- *StackOverFlowError* ocorre quando a computação de uma *thread* precisa de uma pilha maior que a permitida, isso ocorre geralmente em métodos que criam muitas variáveis locais e recursivos.

- *OutOfMemoryError* ocorre quando não há memória suficiente para expandir uma pilha que pode crescer dinamicamente e um objeto não pode ser alocado. Esse erro geralmente acontece quando a aplicação está consumindo muita memória e objetos que deveriam ter sido retirados do *heap* pelo *Garbage Collector* estão permanecendo lá.

Uma das desvantagens do Java é justamente essa, o consumo de memória, por isso é preciso tomar cuidado com as más práticas de programação que podem levar a um consumo excessivo de memória. Em alguns casos isso pode passar despercebido, como é o caso de aplicações de pequeno porte ou aplicações beneficiadas por um bom hardware na máquina, mas em outros casos más práticas de programação podem causar problemas como é o caso de aplicações que serão instaladas em ambientes atípicos e aplicações WEB que consomem gradativamente mais memória a cada novo acesso simultâneo.

Se a aplicação está consumindo muita memória ou apresentou erros por falta dela é essencial fazer Profile da aplicação para identificar quais objetos podem estar consumindo muito memória e não estão sendo retirados do *heap* pelo coletor de lixo, assim o programador poderá fazer as devidas alterações no código para eliminar esses problemas.

Manter o código legível e limpo influencia no consumo de memória e também no desempenho da aplicação, além disso boas práticas de programação podem impactar positivamente no consumo de memória e melhorar o desempenho da aplicação. A seguir é apresentada uma listagem de algumas dessas boas práticas e como essas boas práticas podem impactar no gerenciamento de memória.

1. Reduzir ao máximo o número de vezes que a aplicação faz acesso ao banco de dados

Problema: Programadores as vezes fazem consultas ao banco de dados sem antes analisar se é realmente necessário, cada vez que o banco de dados é acessado objetos vão se multiplicando na aplicação. Em aplicações WEB esses objetos vão se multiplicando a cada acesso simultâneo, um erro como o *OutOfMemoryError*

pode surgir, além disso nesse caso o banco de dados é remoto e isso é o que mais degrada a performance de uma aplicação.

Solução: Analisar quantas vezes é realmente necessário fazer acesso ao banco de dados e usar JOINS e SUBQUERYS e diminuir os acessos.

2. Concatenação de objetos String

Problema: Objetos Strings quando são criados nunca mudam, operações com Strings sempre geram outras Strings e seu uso abusivo aumenta o consumo de memória da aplicação. Outro problema com objetos String é a concatenação. Sempre que há uma concatenação de String novos endereços de memória são alocados para essa tarefa e só são liberados quando o método termina, ou seja, a memória usada por essas Strings vai se multiplicando a cada concatenação. A Figura 16 ilustra um exemplo de concatenação incorreta de Strings.

```
public class Principal {

    public static void main(String[] args) {

        String palavra = "FEMA";

        palavra = palavra + "FEMA";

    }
```

Figura 16 – Exemplo de concatenação de String

Solução: Usar *StringBuffer* ou *StringBuilder* para concatenações, assim a operação usará apenas um endereço de memória e não criará mais Strings a cada concatenação (Figura 17).

```
public static void main(String[] args) {

    StringBuilder palavra = new StringBuilder();

    palavra.append("FEMA");
    palavra.append(" - ");
    palavra.append("Fundação ");
    palavra.append("Educacional ");
    palavra.append("do ");
    palavra.append("Municipio ");
    palavra.append("de ");
    palavra.append("Assis.");

}
```

Figura 17 – *StringBuilder* para concatenação de Strings

3. Controlar a criação de objetos

Problema: É importante controlar a criação de objetos durante a execução de uma programa pois a cada chamada o operador *new* aloca fisicamente mais memória no *heap.*

Solução: O programador deve se perguntar o porque alocar mais objetos, se é realmente necessário e quantas vezes a criação será executada. Outra solução é o reuso de objetos.

4. Fechar as conexões com o banco de dados

Existem várias formas de conectar uma aplicação Java com o banco de dados, dentre as mais conhecidas está o uso da API do JDBC (*Java Database Conectivity*) e o uso de *frameworks* para mapeamento objeto-relacional como o Hibernate. Para realizar operações com o banco de dados é preciso abrir conexões. No Hibernate isso é feito através da interface *SessionFactory* que cria *Sessions* para operações com o banco de dados.

Problema: A criação de uma *SessionFactory* consome muita memória devido as várias operações e estruturas que essa interface precisa construir, e para cada operação no banco de dados cria-se uma *Session* do Hibernate. Muitos programadores não tem o hábito de fechar essas conexões deixando-as abertas aumentando o consumo de memória pela aplicação a cada *Session* aberta.

É importante ressaltar que esse é um dos principais motivos por falta de memória em servidores que hospedam aplicações WEB.

Solução: Fechar as conexões depois de usadas conforme mostra a Figura 18.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class Principal{

    @SuppressWarnings("deprecation")
    public static void main(String[] args) {

        //Objeto de configuração para criar a conexão com o banco de dados
        Configuration cfg = new Configuration();

        //Fábrica de Conexões
        SessionFactory sf = cfg.buildSessionFactory();

        Usuario novoUsuario = new Usuario();
        novoUsuario.setNome("Mariana");

        //Nova Session para salvar novoUsuario
        Session session = sf.openSession();
        session.getTransaction().begin();
        session.save(novoUsuario);
        session.getTransaction().commit();

        //Fechar a conexão e a SessionFactory
        session.close();
        sf.close();

    }

}
```

Figura 18 – *Sessions* do Hibernate

5. Laços FOR convencionais

Problema: Em um laço FOR convencional (Figura 19) é necessário criar vários objetos para controle do *loop* e isso consome mais processamento e memória.

```
import java.util.ArrayList;
import java.util.List;

public class Principal{

    public static void main(String[] args) {

        List<Usuario> lista = new ArrayList<Usuario>();

        Usuario user1 = new Usuario();
        Usuario user2 = new Usuario();
        Usuario user3 = new Usuario();
        Usuario user4 = new Usuario();

        lista.add(user1);
        lista.add(user2);
        lista.add(user3);
        lista.add(user4);

        for(int i=0; i<lista.size(); i++){

            lista.get(i).setNome("Maria");

        }

    }

}
```

Figura 19 – Uso de laço FOR convencional

Neste exemplo é criado um *ArrayList* chamado “lista” e adicionado a ele quatro objetos “Usuario”. O laço FOR seta o nome “Maria” nos quatro objetos “Usuario” da lista.

Solução: Usar *Iterator* ou “*For extendido”* ao inves de FOR convencionais, com eles não é necessário criar objetos de controle de loop como mostra a Figura 20.

```
import java.util.ArrayList;
import java.util.List;

public class Principal{

    public static void main(String[] args) {

        List<Usuario> lista = new ArrayList<Usuario>();

        Usuario user1 = new Usuario();
        Usuario user2 = new Usuario();
        Usuario user3 = new Usuario();
        Usuario user4 = new Usuario();

        lista.add(user1);
        lista.add(user2);
        lista.add(user3);
        lista.add(user4);

        for(Usuario user : lista){

            user.setNome("Maria");

        }

    }

}
```

Figura 20 – Uso de *Foreach*

6. Tipos primitivos e objetos

Problema: Objetos consomem mais memória do que tipos primitivos.

Solução: Sempre que possível utilize tipos primitivos como long, *string* e *int* ao inves de *Long, String e Integer*.

7. Referencia de objetos no código

Problema: Como visto anteriormente objetos que possuem referencia no código da aplicação não são eliminados pelo *Garbage Collector* e vão se acumulando na memória *heap*.

Solução: Para remover as referências de objetos no código basta fazê-los receber o valor *null,* assim eles serão visíveis para o *Garbage Collector.*

8. Fechar aplicações *Swing*

Swing é uma API que fornece interface gráfica de usuário (GUI) para programas Java. Ela permite a criação de janelas para a aplicação.

Problema: Quando são criados aplicações Java *Swing* ao clicar no botão para fechar a tela a instancia daquela aplicação permanece na memória.

Solução: Usar o método *"setDefaultCloseOperation()"* em telas como mostra a Figura 21. O método fecha a aplicação ao clicar no botão "Fechar" da Janela.

```
public TelaPrincipal(){

    setBounds(100,100,400,400);
    getContentPane().setLayout(null);
    setTitle("Aplicação para estudo de Profile");

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JLabel lblAplicaoParaRealizar = new JLabel("Cadastro de Clientes");
    lblAplicaoParaRealizar.setFont(new Font("Tahoma", Font.BOLD, 14));
    lblAplicaoParaRealizar.setBounds(10, 23, 229, 26);
    getContentPane().add(lblAplicaoParaRealizar);

    JButton btnSair = new JButton("Sair");
    btnSair.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {

            dispose();

        }
    });
    btnSair.setBounds(266, 305, 104, 36);
    getContentPane().add(btnSair);

    JLabel lblNome = new JLabel("Nome:");
    lblNome.setHorizontalAlignment(SwingConstants.RIGHT);
    lblNome.setBounds(10, 72, 64, 14);
    getContentPane().add(lblNome);
```

Figura 21 – Método *setDefaultCloseOperation()*

6. ESTUDOS DE CASO

O gerenciamento eficiente de memória e recursos é uma necessidade de qualquer aplicação sejam elas *desktop,* ou *web* destinadas a executar continuamente durante muito tempo. Neste capítulo serão apresentadas alguns casos de retenção de recursos que podem causar erros de memória, como diagnosticá-los e algumas boas práticas de programação para evitá-los.

6.1. RECURSOS NECESSÁRIOS

Para realizar os experimento foram utilizados os seguintes recursos.

6.1.1. RECURSOS DE SOFTWARE

- IDE Eclipse Indigo 3.6;
- Sistema Operacional Ubuntu 10.04;
- Banco de dados relacional MySQL 5.1;
- Oracle JDK 1.6;
- VISUAL VM 1.2.2.

1.7.2. RECURSOS DE HARDWARE

- Notebook Acer Aspire T2390;
- 2Gb RAM, 160Gb HD, Intel Pentium Dual-core processor T2390.

6.2. APLICAÇÃO DESKTOP COM *SWING*

Para o desenvolvimento da aplicação foram utilizadas as seguintes tecnologias:

- Plataforma J2SE;
- API de componentes de interface gráfica para usuários *Swing;*
- Sistema Operacional Ubuntu 10.04.

A aplicação era iniciada normalmente e não apresentava problemas ou erros, porém ao finalizar a aplicação ela não saia da memória, e continuava retendo recursos do sistema operacional, isso pode ser observado através do Gerenciador de Tarefas do Windows, todas as vezes que a aplicação era iniciada ela permanecia nos processos mesmo depois de ser finalizada.

Sintoma: Todas as vezes que a aplicação é executada, cria-se uma nova instancia da aplicação nos processos do sistema operacional, porém quando a aplicação é finalizada através do botão finalizar da janela principal a aplicação continua executando como um processo consumindo memória.

Problema: A aplicação não está sendo finalizada corretamente, assim, a cada vez que a aplicação é iniciada e finalizada ela continua nos processos do sistema consumindo memória principal.

Solução: Quando um *JFrame* é finalizado através do botão finalizar da janela é preciso sinalizar ao aplicativo qual é a operação de fechamento da janela acrescentando o método *setDefaultCloseOperation* ao construtor da classe. Esse método recebe um parâmetro que pode ser *EXIT_ON_CLOSE,* assim, a instancia da aplicação será finalizada e retirada da memória.

6.3. SISTEMA DE GESTÃO FINANCEIRA

A Associação Filantrópica Nosso Lar foi fundada em 25 de dezembro de 1949, em uma proposta para conveniar com a Assistência Social do Município de Assis. A associação faz parte das instituições que compõem o Projeto Rede Ciranda da Criança e Adolescente de Assis.

O Nosso Lar possui diversos projetos voltados à comunidade de Assis, um deles, financiado pela Fundação Telefônica, que visa dentre vários trabalhos, o desenvolvimento de um sistema para gestão financeira, para prestação de contas das entidades a seus respectivos financiadores.

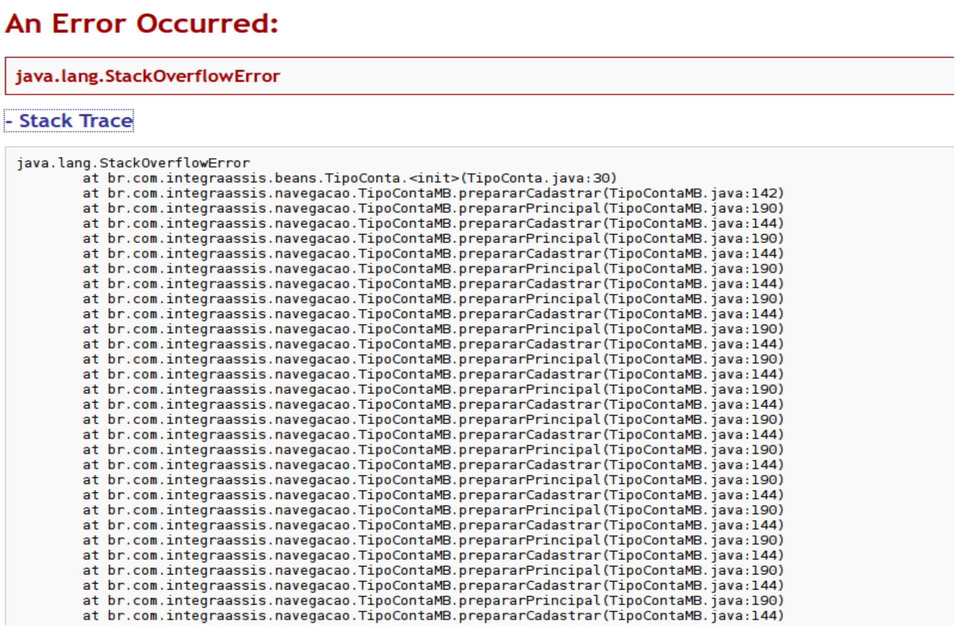
Para o desenvolvimento do sistema foram utilizadas as seguintes tecnologias:

- *Framework* para criação de páginas web JSF2;
- *Framework* para persistencia de dados JPA2;
- Banco de dados relacional Mysql 5;
- Ambiente de desenvolvimento Eclipse Indigo;
- Servidor web Apache Tomcat 7;

A equipe de desenvolvimento era composta por mim, Mariana Budiski e Rafael Moraes. Após meses de desenvolvimento problemas com gerenciamento de memória começaram a surgir.

Sintoma: Em uma das funcionalidades do sistema foi encontrado um erro de estouro de pilha: *StackOverFlowError.*

O erro surgia clicando em um botão no menu para abrir a tela de cadastro de Tipo de Conta. A Figura 22 mostra o erro gerado.


An Error Occurred:

java.lang.StackOverflowError

- Stack Trace

```
java.lang.StackOverflowError
	at br.com.integraassis.beans.TipoConta.<init>(TipoConta.java:30)
	at br.com.integraassis.navegacao.TipoContaMB.prepararCadastrar(TipoContaMB.java:142)
	at br.com.integraassis.navegacao.TipoContaMB.prepararPrincipal(TipoContaMB.java:190)
	at br.com.integraassis.navegacao.TipoContaMB.prepararCadastrar(TipoContaMB.java:144)
	at br.com.integraassis.navegacao.TipoContaMB.prepararPrincipal(TipoContaMB.java:190)
	at br.com.integraassis.navegacao.TipoContaMB.prepararCadastrar(TipoContaMB.java:144)
	at br.com.integraassis.navegacao.TipoContaMB.prepararPrincipal(TipoContaMB.java:190)
	at br.com.integraassis.navegacao.TipoContaMB.prepararCadastrar(TipoContaMB.java:144)
	at br.com.integraassis.navegacao.TipoContaMB.prepararPrincipal(TipoContaMB.java:190)
	at br.com.integraassis.navegacao.TipoContaMB.prepararCadastrar(TipoContaMB.java:144)
	at br.com.integraassis.navegacao.TipoContaMB.prepararPrincipal(TipoContaMB.java:190)
	at br.com.integraassis.navegacao.TipoContaMB.prepararCadastrar(TipoContaMB.java:144)
	at br.com.integraassis.navegacao.TipoContaMB.prepararPrincipal(TipoContaMB.java:190)
	at br.com.integraassis.navegacao.TipoContaMB.prepararCadastrar(TipoContaMB.java:144)
	at br.com.integraassis.navegacao.TipoContaMB.prepararPrincipal(TipoContaMB.java:190)
	at br.com.integraassis.navegacao.TipoContaMB.prepararCadastrar(TipoContaMB.java:144)
	at br.com.integraassis.navegacao.TipoContaMB.prepararPrincipal(TipoContaMB.java:190)
	at br.com.integraassis.navegacao.TipoContaMB.prepararCadastrar(TipoContaMB.java:144)
	at br.com.integraassis.navegacao.TipoContaMB.prepararPrincipal(TipoContaMB.java:190)
	at br.com.integraassis.navegacao.TipoContaMB.prepararCadastrar(TipoContaMB.java:144)
	at br.com.integraassis.navegacao.TipoContaMB.prepararPrincipal(TipoContaMB.java:190)
	at br.com.integraassis.navegacao.TipoContaMB.prepararCadastrar(TipoContaMB.java:144)
	at br.com.integraassis.navegacao.TipoContaMB.prepararPrincipal(TipoContaMB.java:190)
	at br.com.integraassis.navegacao.TipoContaMB.prepararCadastrar(TipoContaMB.java:144)
	at br.com.integraassis.navegacao.TipoContaMB.prepararPrincipal(TipoContaMB.java:190)
	at br.com.integraassis.navegacao.TipoContaMB.prepararCadastrar(TipoContaMB.java:144)
	at br.com.integraassis.navegacao.TipoContaMB.prepararPrincipal(TipoContaMB.java:190)
	at br.com.integraassis.navegacao.TipoContaMB.prepararCadastrar(TipoContaMB.java:144)
	at br.com.integraassis.navegacao.TipoContaMB.prepararPrincipal(TipoContaMB.java:190)
	at br.com.integraassis.navegacao.TipoContaMB.prepararCadastrar(TipoContaMB.java:144)
```

Figura 22 - *StackOverFlowError*

Segundo o *Stack Trace*, aparentemente uma funcionalidade estava criando muitos objetos da classe TipoConta. Pesquisando mais sobre o erro foi possível constatar que ele era causado por um método recursivo que estava em *loop* infinito e estava criando muitos objetos da classe TipoConta.

Problema: Analisando os métodos do *managed bean* da tela de cadastro de tipo de conta, um método implementado errado estava chamando outro e retornava ao primeiro, recursivamente, em um desses métodos era criado um objeto da classe TipoConta. Como os métodos eram executados indefinidamente em um *loop*, muitos objetos da classe TipoConta eram instanciados causando o erro.

Após resolver esse problema, observamos que o Tomcat estava consumindo memória excessivamente. O servidor onde hospedaríamos o sistema já contava com vários sites hospedados para as Entidades Sociais e um Sistema para Gerenciamento de Crianças e Adolescentes que era usado diariamente por várias Entidades que faziam parte do Projeto, por isso o Sistema de Gestão Financeira não poderia consumir muita memória. Foi então que surgiu a necessidade de monitorar a aplicação para ver o comportamento dela com relação a memória utilizada.

A ferramente utilizada foi a Visual VM, disponível nas distribuições da JDK. Através do *Monitor* da ferramente foi possível constatar que o sistema consumia aproximadamente 100Mb ao iniciar na página de login. Após efetuar login no sistema e realizar algumas operações como pesquisas e cadastros, o consumo de memória foi aumento gradativamente como mostra a Figura 23.

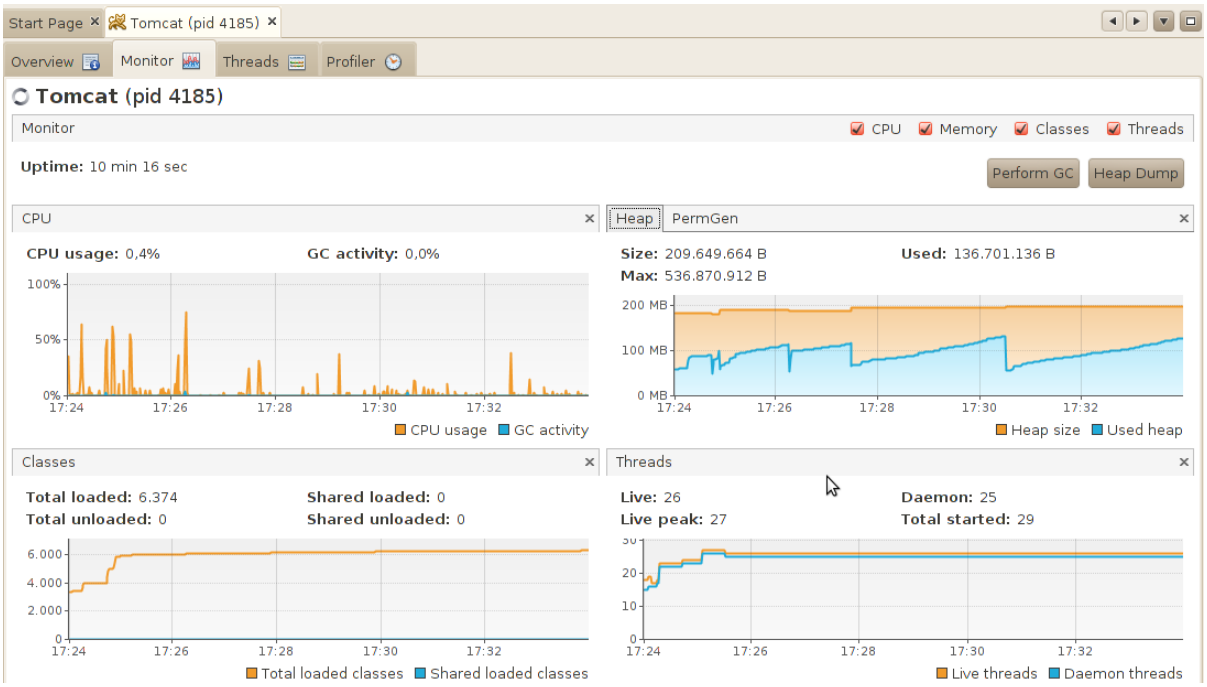


Figura 23 – Uso do *Heap* e *Garbage Collector*

Na Figura 23 é possível observar que o tamanho da memória *heap* para ser utilizado é de 209Mb o tamanho total alocado é de 536Mb e usado 136Mb. É possível observar também que houve grande uso de CPU para os métodos e pouca atividade do *Garbage Collector.*

A Figura 24 mostra o uso do PermGem, outra área importante de memória do Java.

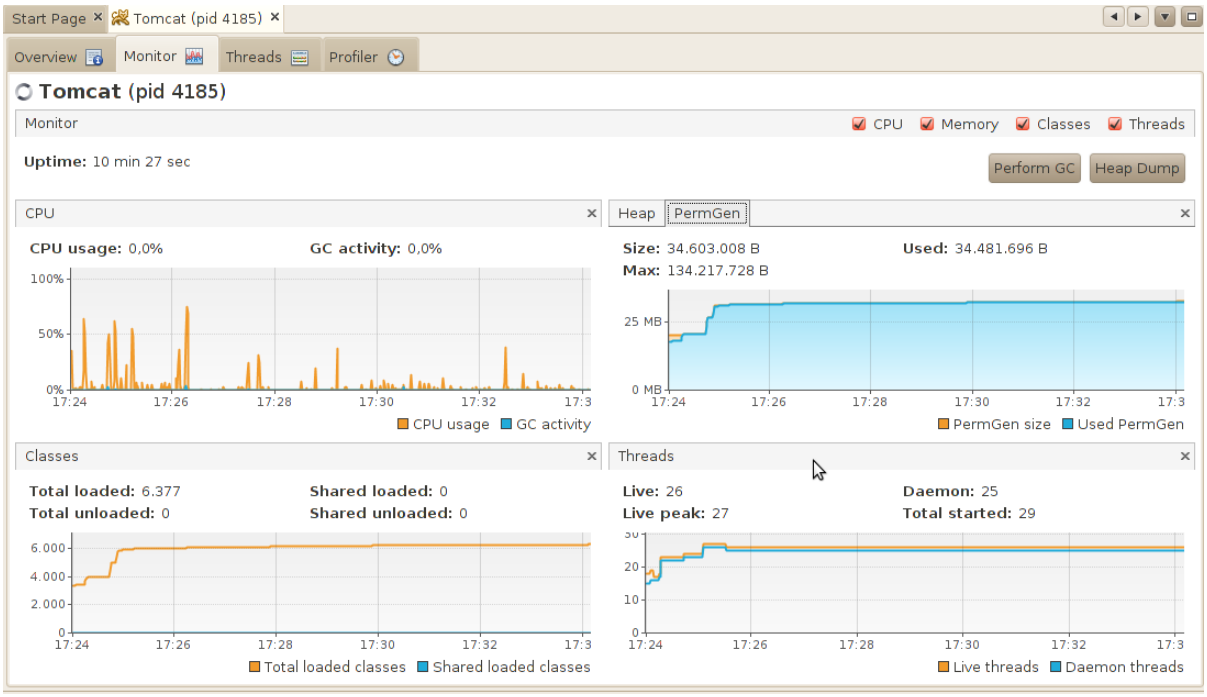


Figura 24 – Uso do *PermGem*

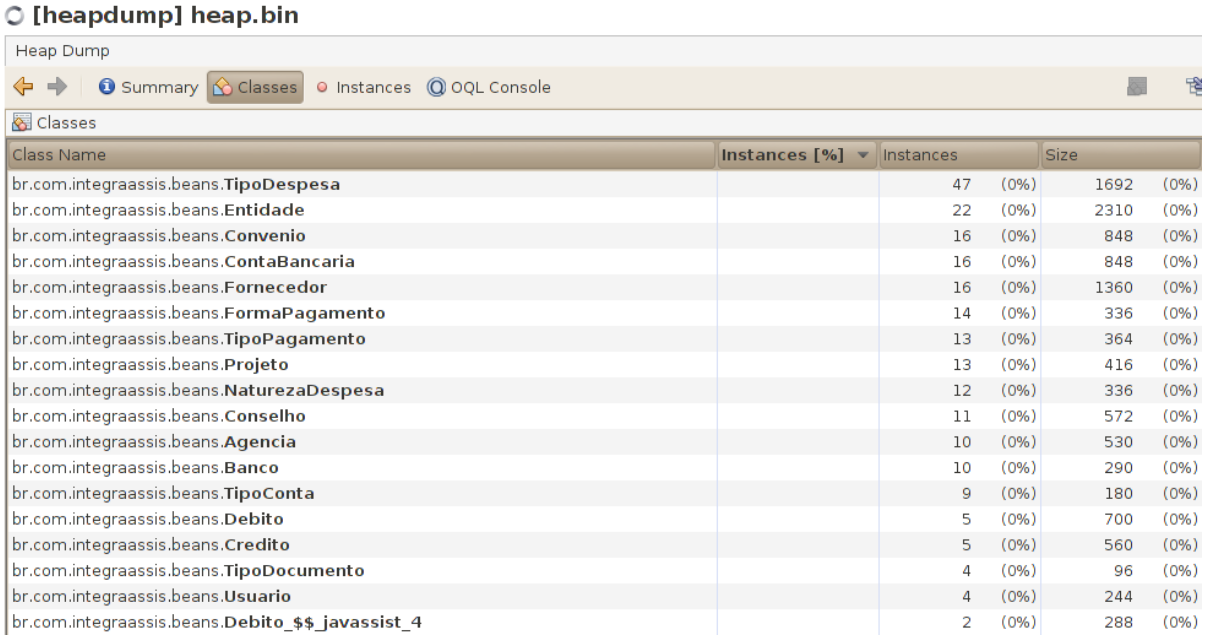
O uso do *PermGem* pelo sistema estava quase no limite do disponível para uso e crescendo cada vez mais.

O consumo de memória estava alto com apenas um usuário conectado, sem dúvidas o consumo aumentaria a medida que mais usuários se conectassem ao sistema e o servidor já contava com outras aplicações, os recursos eram limitados.

Problemas de memória geralmente são difíceis de diagnosticar, mas saber quais objetos estavam permanecendo na memória ajudaria muito a identificar o problema, para isso foi feito o *Profile* da aplicação.

Após executar a aplicação por algum tempo passando por várias funcionalidades como cadastros e pesquisas foi utilizado um utilitário da JDK, o JMAP, para guardar em um arquivo binário detalhes do uso de memória HEAP da aplicação naquele momento. Para isso foi usado o comando jmap -heap:format=b 2791. O comando cria vários arquivos com informações sobre aplicação e uso e memória.

Na Visual VM é possível abrir o arquivo gerado para análise, o heap.bin, com ele foi possível observar os objetos que estavam vivos em memória como pode ser visto na Figura 25.



Class Name	Instances [%]	Instances		Size	
br.com.integraassis.beans.**TipoDespesa**		47	(0%)	1692	(0%)
br.com.integraassis.beans.**Entidade**		22	(0%)	2310	(0%)
br.com.integraassis.beans.**Convenio**		16	(0%)	848	(0%)
br.com.integraassis.beans.**ContaBancaria**		16	(0%)	848	(0%)
br.com.integraassis.beans.**Fornecedor**		16	(0%)	1360	(0%)
br.com.integraassis.beans.**FormaPagamento**		14	(0%)	336	(0%)
br.com.integraassis.beans.**TipoPagamento**		13	(0%)	364	(0%)
br.com.integraassis.beans.**Projeto**		13	(0%)	416	(0%)
br.com.integraassis.beans.**NaturezaDespesa**		12	(0%)	336	(0%)
br.com.integraassis.beans.**Conselho**		11	(0%)	572	(0%)
br.com.integraassis.beans.**Agencia**		10	(0%)	530	(0%)
br.com.integraassis.beans.**Banco**		10	(0%)	290	(0%)
br.com.integraassis.beans.**TipoConta**		9	(0%)	180	(0%)
br.com.integraassis.beans.**Debito**		5	(0%)	700	(0%)
br.com.integraassis.beans.**Credito**		5	(0%)	560	(0%)
br.com.integraassis.beans.**TipoDocumento**		4	(0%)	96	(0%)
br.com.integraassis.beans.**Usuario**		4	(0%)	244	(0%)
br.com.integraassis.beans.**Debito_$$_javassist_4**		2	(0%)	288	(0%)

Figura 25 – Heap dump da aplicação

Na Figura 25 é visto que muitas instâncias de objetos estavam permanecendo na memória, a classe TipoDespesa por exemplo mantinha 47 instâncias vivas, já a classe Entidade 22 objetos. Esses objetos deveriam estar sendo coletados pelo *Garbage Collector*.

Problema: Muitos objetos estavam sendo instanciados e permanecendo na memoria.

Algumas funcionalidades estavam carregando muitos objetos na memória que não estavam sendo liberados pelo *Garbage Collector* e então permanecendo lá. A medida que mais usuários se conectassem ao sistema simultaneamente o consumo aumentaria e possíveis erros por falta de memória poderiam ocorrer.

Através do *Heap dump* foi possível observar que as conexões com o banco de dados estavam sendo fechadas corretamente pois não haviam instâncias de conexões do Hibernate vivas, o fechamento das conexões na aplicação é feito através de um *Web Filter,* porém muitos objetos estavam permanecendo na memória como *DAOs, Managed Beans* e principalmente objetos que fazem parte do pacote *bean* da aplicação, ou seja, objetos do modelo. Esses objetos poderiam estar em *ArrayLists* que as telas consumiam e não estavam sendo liberados como deveriam, ou seja, depois que a tela que consumia esses *ArrayLists* fosse fechada os objetos deveriam ser retirados da memória mas isso não acontecia.

O problema era causado porque a aplicação estava com todos os *Managed Beans* mapeados com escopo de sessão *(Session Scoped)* e por isso permanecendo na memória, esse escopo define que os backing beans teriam o tempo de vida que a aplicação estivesse online, por isso não estavam sendo coletados, além disso objetos não mais usados ainda tinham referência no código e por isso não estavam sendo liberados também.

Uma das soluções seria aumentar a memória alocada pela JVM através de parâmetros, mas isso não resolveria o problema, mas sim o esconderia por algum tempo, já que a aplicação precisaria ficar meses a fio online, inevitavelmente a memória se esgotaria com o tempo.

Soluções:

- Alterar o escopo de vida de objetos para que sejam visíveis mais rapidamente pelo coletor de lixo;
- Fazer objetos ficarem visíveis pelo coletor e lixo logo após o uso atribuindo *null* a eles;
- Aplicar o *singleton* para objetos com única funcionalidade ao invés de recriá-los;
- Reutilizar objetos;
- Usar a abordagem de paginação ao invés de criar um alto volume de objetos resultantes de interações com banco de dados.

7. CONCLUSÃO

O gerenciamento de memória é um campo complexo da computação e exige diversos tratamentos e técnicas para torná-lo mais eficiente e é uma necessidade de qualquer aplicação, especialmente aquelas destinadas a serem executadas por meses a fio. O problema mais comum é o vazamento de memória que ocorre quando uma quantidade de memória é alocada e não liberada.

O gerenciamento de memória automático do Java permite ao programador se preocupar apenas com a lógica de negócio da aplicação livrando-o de detalhes de alocação e liberação de memória, porém esse também é o motivo para a excessiva confiabilidade dos programadores e despreocupação com suas práticas de programação ao desenvolver uma aplicação. Gerenciamento de memória automático significa que os desenvolvedores tendem a estar mais desatentos com detalhes de memória mas esse mecanismo apresenta limitações, pois a forma que o coletor de lixo entra em ação é justamente onde reside o problema, a coleta pode acontecer a qualquer momento e pode levar um longo tempo para se completar, e isso toma uma porcentagem valiosa de tempo de processamento da aplicação. As soluções foram projetadas para o programador das boas práticas.

Além disso, o avanço do hardware tem sido crucial para a despreocupação dos programadores com o consumo de memória por suas aplicações, mas o *Garbage Collector* não faz “milagre”, e algumas práticas comuns de programação podem danificar esse mecanismo de liberação de blocos de memória não mais utilizados.

Boas práticas de programação podem fazer toda a diferença tanto no consumo de memória quanto no desempenho da aplicação.

REFERENCIAS BIBLIOGRÁFICAS

DEITEL, P. J.; DEITEL, H. M. Java como programar 8ª edição. São Paulo: Pearson Prentice Hall, 2001. 1114 p.

METSKER, Steven John. Padrões de Projeto em Java. São Paulo: Editora Bookman, 2004. 407 p.

GONÇALVES, Edson. Desenvolvendo Aplicações WEB com JSP, Servlets, JavaServer Faces, Hibernate, EJB Persistence e Ajax. Rio de Janeiro: Editora Ciência Moderna, 2007. 736 p.

SIERRA, Kathy; BATES, Bert. Use a Cabeça! Java. Rio de Janeiro: Editora Alta Books, 2010. 484 p.

PINTO, Flávio; LUNA, Hadley. Gerenciamento de Memória em Java. Disponível em:< http://flaviopintoblog.wordpress.com/2010/06/24/gerenciamento-de-memria-em-java/> Acesso em: 20 de mar. 2012.

ROCHA, Helder. Tutorial de Gerência de Memória em Java. Disponível em:< http://www.argonavis.com.br/cursos/java/j190/TutorialGerenciaMemoriaJava.pdf

> Acesso em: 14 de abr. 2012.

Documentation Java VisualVM. Disponível em:< http://docs.oracle.com/javase/6/docs/technotes/guides/visualvm/index.html

> Acesso em: 10 de abr. 2012.

JESUINO Memory Leak em Java. Disponível em:<http://javafree.uol.com.br/artigo/878208/Memory-leak-em-Java.html>. Data de acesso: 20 de abr. 2012.

LUCKOW, Décio Heinzelmann. Monitoramento e Análise Detalhada de Memória no Java. Disponível em:<http://www.mgjug.com.br/index.php?option=com_content&view=article&id=56:analisememoria&catid=36:mateirais&Itemid=58>. Data de acesso: 6 de jun. 2012.

DAVID, Marcio Frayze. Programação Orientada a Objetos: Uma introdução. Disponível em:<http://www.mgjug.com.br/index.php?

option=com_content&view=article&id=56:analisememoria&catid=36:mateirais&Itemid=58>. Data de acesso: 7 de jun. 2012.

Cinco coisas que você não sabia sobre monitoramento de desempenho Java Parte 2: Monitoramento de processo Java com os gerenciadores de perfis da jdk. Disponível em:< http://docs.oracle.com/javase/6/docs/technotes/guides/visualvm/index.html

Documentation HyperSQL. Disponível em:<http://hsqldb.org/web/hsqlDocsFrame.html>

OLIVEIRA, Éric C. M. Garbage Collector e as Armadilhas na Performance de Aplicações. Disponível em:<http://www.linhadecodigo.com.br/artigo/568/garbage-collector-e-as-armadilhas-na-performance-de-aplicacoes.aspx>. Data de acesso: 12 de out. 2012.

FRANZINI, Fernando. A Famosa Falta de Memória *OutOfMemoryError*. Disponível em:<http://fernandofranzini.wordpress.com/2011/12/13/a-famosa-falta-de-memoria-java-lang-outofmemoryerror/>. Data de acesso: 12 de out. 2012.

DEODERLEIN, Osvaldo P. Gerenciando Memória e Recursos. Java Magazine DIGITAL, São Paulo, v. 75, p. 188-225.

ALLES, Vanderlei. Tratamento de Exceções e Erros. Disponível em: <http://fortium.edu.br/blog/vanderlei_alles/files/2010/05/excecao2.pdf>. Data de acesso: 20 de out. 2012.

SILVEIRA, Paulo; SILVEIRA, Guilherme; LOPES, Sérgio; MOREIRA, Guilherme; STEPPAT, Nico; KUNG, Fabio. Introdução à Arquitetura e Design de Software: Uma visão sobre a plataforma JAVA. Rio de Janeiro: Elsevier, 2012.