|  |
| --- |
| LogoISCTE-IUL.png  Mestrado em engenharia informática, 2º ano  Programação Avançada 2009/2010 |
| *Visual Tracer* |
| Relatório final |
|  |
|  |
| **14-02-2010** |

|  |  |
| --- | --- |
| **Professor**: | |
| Manuel Menezes de Sequeira | |
| **Alunos**: | |
| Carlos Correia | nº 8298 |
| Rute Oliveira | nº 8299 |

Índice

[1. Introdução 3](#_Toc253914677)

[2. Modelação/Desenho da aplicação 4](#_Toc253914678)

[2.1. Model-View-Controller (MVC) 4](#_Toc253914679)

[2.2. Modelo de dados (Model) 5](#_Toc253914680)

[2.2.1. Estrutura sintáctica 7](#_Toc253914681)

[2.2.2. Estrutura de fluxo de execução 8](#_Toc253914682)

[2.3. Controlo (Controller) 10](#_Toc253914683)

[2.4. Interface gráfica (View) 10](#_Toc253914684)

[2.5. AspectJ – *Tracing* 12](#_Toc253914685)

[2.6. Casos de uso (use cases) 14](#_Toc253914686)

[2.7. Diagramas de sequência 15](#_Toc253914687)

[2.7.1. Exemplo de fluxo 16](#_Toc253914688)

[2.7.2. Exemplo de actualização da interface gráfica 18](#_Toc253914689)

[2.8. Opções tomadas 18](#_Toc253914690)

[3. Implementação da aplicação 19](#_Toc253914691)

[3.1. Decisões tomadas 19](#_Toc253914692)

[3.1.1. Representação das instâncias por *MyWeakReference* 19](#_Toc253914693)

[3.1.2. Identificação de threads 21](#_Toc253914694)

[*3.1.3.* Criação e selecção de *snapshots* 23](#_Toc253914695)

[3.1.4. Interface gráfica – *Standard Widget Toolkit* (SWT) e JFace 24](#_Toc253914696)

[4. Instalação 25](#_Toc253914697)

[5. Utilização 25](#_Toc253914698)

[5.1. Manual do utilizador 25](#_Toc253914699)

[5.2. Exemplos de utilização 26](#_Toc253914700)

[6. Discussão 31](#_Toc253914701)

[7. Trabalho futuro 31](#_Toc253914702)

[Bibliografia 35](#_Toc253914703)

[Glossário 36](#_Toc253914704)

# Introdução

Na programação, o rastreamento de código é uma actividade imprescindível para a percepção de comportamentos não esperados de um programa e para a detecção de erros no código. A título académico, é útil para a compreensão dos resultados da execução de um programa, bem como para entender a ordem de execução das instruções.

O projecto aqui descrito consiste na concepção e construção de uma ferramenta de rastreamento de código, utilizando programação orientada a aspectos (POA). A aplicação pretende mostrar em tempo-real os detalhes da criação de classes e objectos, da alteração de atributos e da execução de métodos, para cada thread criada pela aplicação a ser rastreada. A sua arquitectura foi concebida com o fim de oferecer a possibilidade de extensão para outro tipo de ferramentas. Esta arquitectura está assente no padrão de desenho *Model-View-Controller* (MVC), para permitir precisamente a independência entre interface e o modelo de dados e facilitar a reutilização do código. Para além disso, a arquitectura pretende ser o mais modular possível, isto é, isolar cada funcionalidade num módulo individual. Embora o paradigma da programação orientada a objectos (POO) constitua um avanço neste sentido, ainda possui algumas falhas na modularização e encapsulamento.

A POA surge assim, não como alternativa, mas como complementar à POO. Permite o encapsulamento de certas preocupações transversais como a autenticação, criação e actualização de *logs*, persistência e verificação de asserções que, apenas recorrendo a POO, se repetiriam várias vezes ao longo do código, em vários locais.

Na primeira secção será descrito o modelo conceptual da ferramenta desenvolvida, explicitando o padrão de desenho Model-View-Controller (MVC) e cada uma das suas camadas. A programação por aspectos é depois focada no contexto do *tracing*, que é o objectivo deste projecto. Em seguida são apresentados diagrama UML de casos de uso e de sequência. A primeira secção termina com a descrição das opções tomadas na concepção. A implementação do projecto é o segundo tópico, onde se enumeram as decisões tomadas nesta etapa e se explicitam alguns dos conceitos inerentes às técnicas utilizadas. Em seguida é explicitada a instalação da aplicação e depois a sua utilização através de exemplos práticos de utilização. A secção seguinte enuncia as conclusões retiradas e a discussão do projecto. Por fim, são sugeridas algumas extensões ao projecto para trabalho futuro.

# Modelação/Desenho da aplicação

## Model-View-Controller (MVC)

O padrão de desenho *Model-View-Controller* (MVC) permite o encapsulamento de tarefas de um programa através de camadas isoladas. Desta forma as alterações em cada uma das camadas não implicam a alteração de toda a estrutura da aplicação.

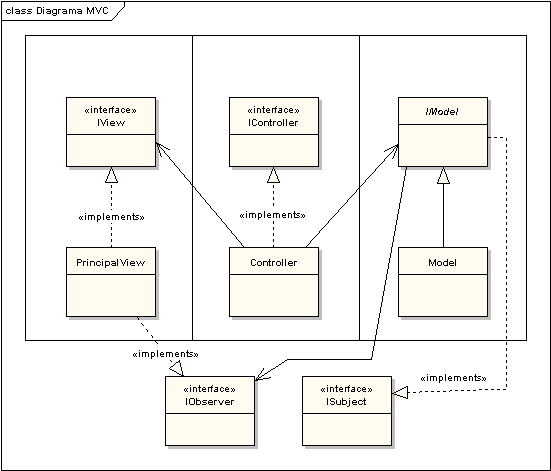
Este modelo é constituído por três camadas:

* *Model* – camada de modelo de dados da qual fazem parte todas as classes referentes ao modelo de dados da aplicação (apresentadas na secção anterior);
* *View* – camada da interface que engloba todas as classes responsáveis pela interacção com o utilizador;
* *Controller* – camada de controlo, responsável pela mediação entre a camada View e a camada M*odel*; esta camada contém as regras de negócio.

O padrão de desenho MVC recorre a três padrões de desenho: *Composite* (que permite a organização da camada de interface gráfica em componentes menores), *Strategy* (cujo objectivo é definir a estratégia na camada de controlo e delegar tarefas) e *Observer* (onde as várias interfaces gráficas subscrevem-se para serem notificadas de alterações na camada de modelo de dados).

Na aplicação *Visual* *Tracer* as três camadas do padrão de desenho MVC correspondem a três pacotes distintos: *model*, *view* e *controller*. Cada pacote contém as classes que constituem cada camada. No Diagrama 1 demonstram-se as principais classes de cada camada. Tanto na camada *View* como na camada *Controller* existe uma interface responsável pela comunicação – *IView* e *IController* – e uma classe que implementa essa interface – *Principal View* e *Controller*. Já na camada *Model*, a classe de comunicação é uma classe abstracta – *IModel* – devido às limitações da linguagem Java por não permitir que uma interface implemente outra interface.

De acordo com o padrão de desenho *Observer*, a classe *IModel* implementa a interface *ISubject* de modo a notificar os interessados acerca da mudança do seu estado. Por sua vez, a classe *PrincipalView* implementa a interface *IObserver* de modo a subscrever-se para receber as notificações de alterações que possam ocorrer na camada do modelo de dados.



***View***

***Controller***

***Model***

Diagrama –MVC

Na camada de interface pode observar-se que a classe *PrincipalView* está relacionada com a interface *IController* para, deste modo, comunicar ao controlador as alterações em si ocorridas. Já na camada de controlo, é a classe *Controller* que comunica tanto com a camada de interface – através da relação com a interface *IView* – e com a camada do modelo de dados – através da relação com a interface *IModel*. A camada de modelo de dados comunica as suas alterações através da relação estabelecida com a interface *IObserver*. Assim, todas as classes que implementam esta interface (sendo neste caso apenas a classe *PrincipalView*) são notificadas das alterações ocorridas na camada de modelo de dados.

Nas secções seguintes, cada uma das três camadas do padrão de desenho MVC será explicitada com mais detalhe.

## Modelo de dados (Model)

O modelo conceptual foi idealizado para que no futuro possa ser expandido e utilizado para criar novas vistas de instrumentação. É composto essencialmente por duas estruturas de dados: a estrutura que representa a parte de fluxo de execução do programa e a estrutura que diz respeito à parte sintáctica do programa. A interligação entre estas duas componentes é feita através da classe *Model*, como podemos observar no Diagrama 2. A classe *Model* implementa a interface *IModel*, conforme as regras da utilização do padrão de desenho *Model-View-Controller* ditam (ver 2.1 Model-View-Controller (MVC)). As duas estruturas interligam-se ainda através da conexão existente entre as classes de topo de cada uma, sendo que uma instância da estrutura de fluxo de execução (*AbstractJoinPoint*) corresponde a uma instância

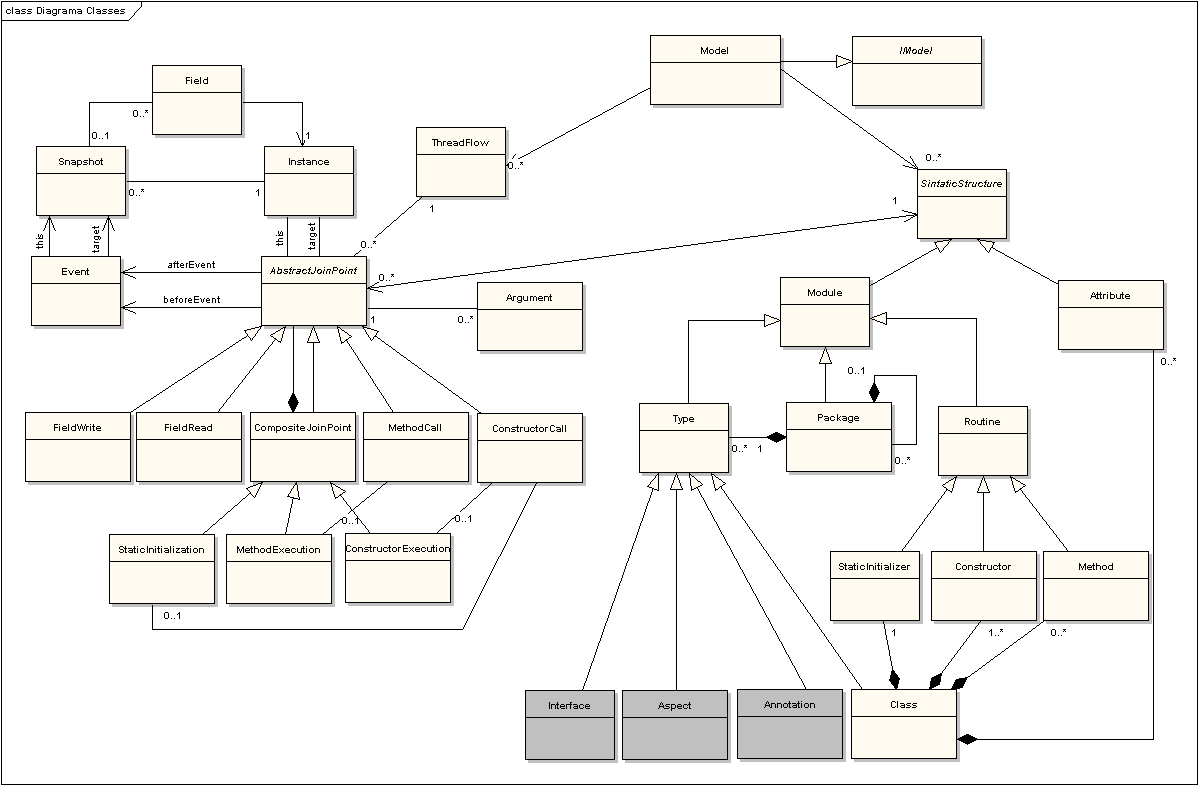


Diagrama – Diagrama de classes UML

da estrutura sintáctica (*SintaticStructure*) e uma instância da estrutura sintáctica irá ter várias instâncias correspondentes da estrutura de fluxo de execução.

As duas estruturas mencionadas serão descritas seguidamente com mais detalhe. A componente sintáctica está relacionada com a representação dos conceitos da programação orientada a objectos. O contexto de execução de um programa é representado pela componente de fluxo de execução, cujos conceitos correspondem a pontos de junção da programação orientada a aspectos.

### Estrutura sintáctica

Com o propósito de, no futuro, ser possível adicionar ao *Visual Tracer* uma vista de instrumentação para avaliar o desempenho da execução de um programa orientado a objectos, foi conceptualizada uma estrutura sintáctica. Esta estrutura pretende representar os componentes existentes na programação orientada a objectos, isto é, a parte sintáctica de um programa.

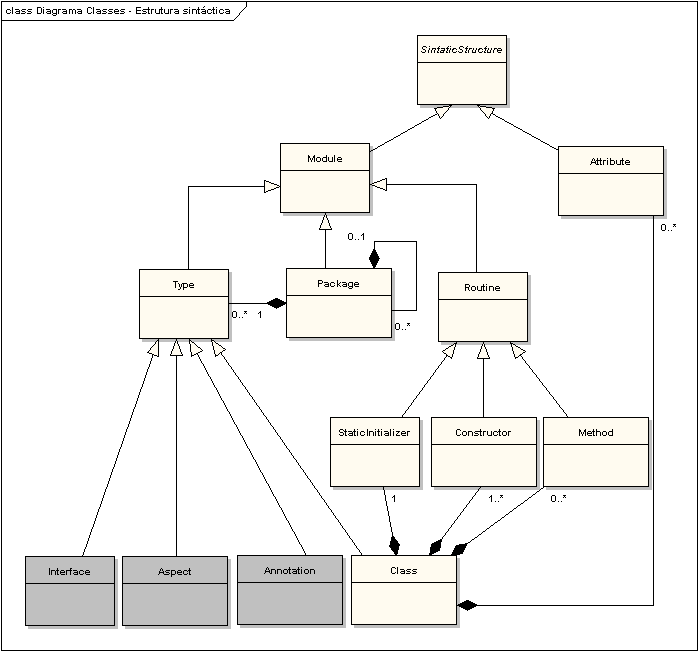


Diagrama – Estrutura sintáctica

A classe de topo desta estrutura é a *SintaticStructure*. Esta classe é uma classe abstracta, da qual estendem as classes *Module* e *Attribute*, sendo este último a representação de um atributo de uma classe, seja ela estática ou não.

A classe *Module*, por sua vez, pode representar um tipo (*Type*), um pacote (*Package*) ou uma rotina (*Routine*). Um pacote poderá ser constituído por outros pacotes ou então constituído por tipos.

As interfaces (*Interface*), aspectos (*Aspect*), anotações (*Annotation*) e classes (*Class*) são as extensões possíveis da classe *Type*, embora este projecto não tenha contemplado as três primeiras na sua implementação. No Diagrama 3 estas classes possuem um fundo de cor cinzenta por esse motivo.

Uma rotina poderá representar um inicializador de uma classe (*StaticInitializer*), um construtor (*Constructor*) ou um método (*Method*). Estas rotinas estão relacionadas com as classes (*Class*), na medida em que uma classe é composta por uma única inicialização de uma classe, por um ou mais construtores e por nenhum ou vários métodos. A classe (*Class*) poderá também conter nenhum ou vários atributos (*Attribute*).

### Estrutura de fluxo de execução

O projecto *Visual Tracer* incidiu sobre a estrutura de fluxo de execução de um programa, uma vez que o seu objectivo é capturar e apresentar as instruções executadas durante a execução do programa a rastrear. As classes que compõem esta estrutura, apresentadas no Diagrama 4, assemelham-se a pontos de junção da programação por aspectos.

Nesta estrutura a classe de topo, ou classe central, é a *AbstractJoinPoint* que, como o nome indica, é uma classe abstracta que pretende representar um ponto de junção (*joinpoint*). As extensões possíveis a esta classe são as classes que representam a leitura de um campo (*FieldRead*), a escrita de um campo (*FieldWrite*), a invocação de um método (*MethodCall*), a invocação de um construtor (*ConstructorCall*) ou um ponto de junção composto (*CompositeJoinPoint*). Este último é a generalização de representações de inicializações de uma classe (*StaticInitialization*), de execuções de métodos (*MethodExecution*) e de execuções de construtores (*ConstructorExecution*). O ponto de junção composto é constituído por pontos de junção abstractos, ou seja, a execução de um método poderá conter dentro dela escritas e leituras de campos ou invocações a métodos e construtores, por exemplo.

Como se pode observar no Diagrama 4, a classe *MethodCall* relaciona-se com uma instância da classe *MethodExecution*, o que significa que uma invocação a um método corresponde a uma execução a um método. O mesmo se aplica aos construtores, com as classes *ConstructorCall* e *ConstructorExecution*, respectivamente. Estas relações contemplam o caso em que uma invocação poderá não ter nenhuma execução associada pelo facto de os aspectos não capturarem algumas bibliotecas do Java, como é o caso da biblioteca de *Swing* por exemplo. No caso da invocação de um construtor poderá ainda ocorrer a inicialização da classe, algo que está reflectido na relação entre as classes *ConstructorCall* e *StaticInitialization*.

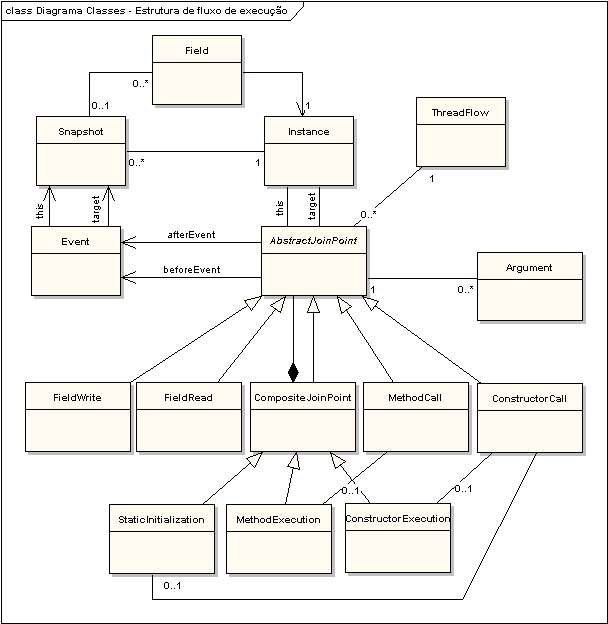


Diagrama – Estrutura de fluxo de execução

Um ponto de junção abstracto possui também uma ligação à classe que representa os argumentos de um método ou construtor e respectivos valores (*Argument*), ligação que apenas fará sentido se a instância se tratar de uma instância de *MethodCall*, *MethodExecution*, *ConstructorCall* e *ConstructorExecution*.

A classe *AbstractJoinPoint* relaciona-se com a classe *Event*, uma vez que um ponto de junção abstracto irá conter duas instâncias da classe *Event* que correspondem aos eventos ocorridos antes (*beforeEvent*) e depois (*afterEvent*) do ponto de junção em questão. A classe *Event*, por sua vez, contém duas instâncias de *Snapshot*, que dizem respeito aos objectos *this* e *target* da instância de *Event* em causa. Por possuir duas instâncias que representam instâncias da execução de um programa (*this* e *target*), o ponto de junção abstracto (*AbstractJoinPoint*) está também relacionado com a classe *Instance*. Uma instância da classe *Instance* tem, por sua vez, vários *snapshots* – instâncias da classe *Snapshot* – associados.

A classe *Field* representa um atributo de uma instância (*Instance*) com um valor atribuído num determinado instante de tempo. Por este motivo, uma instância da classe *Field* poderá ter um *snapshot* (*Snapshot*) associado, caso o seu valor tenha sido alterado. Um *snapshot* pode conter várias instâncias da classe *Field*, pois quando o estado da instância é alterado (isto é, quando um dos seus atributos é alterado) é guardado um *snapshot* com todos os atributos da instância.

A separação dos pontos de junção pelas respectivas *threads* é exequível através da classe *ThreadFlow*, cujo propósito é guardar o histórico de pontos de junção da sua execução. Ou seja, cada instância de *ThreadFlow* está associada a uma única *thread* e guarda as instâncias de *AbstractJoinPoint* relativas a essa *thread*.

## Controlo (Controller)

Esta é a camada responsável pela mediação de dados entre a camada de interface e a camada do modelo de dados. Assim sendo, o controlador pode ser invocado por qualquer uma das camadas.

A camada de controlo poderá conter mais do que um controlador, sendo que um dos controladores terá que ser o principal e o responsável por delegar tarefas a outros controladores (por exemplo, tendo um controlador principal que distribuía tarefas para o controlador responsável pelo *tracing* e para um outro controlador responsável por *profiling*). Neste momento, a aplicação *Visual* *Tracer* apenas contém um controlador na camada de controlo.

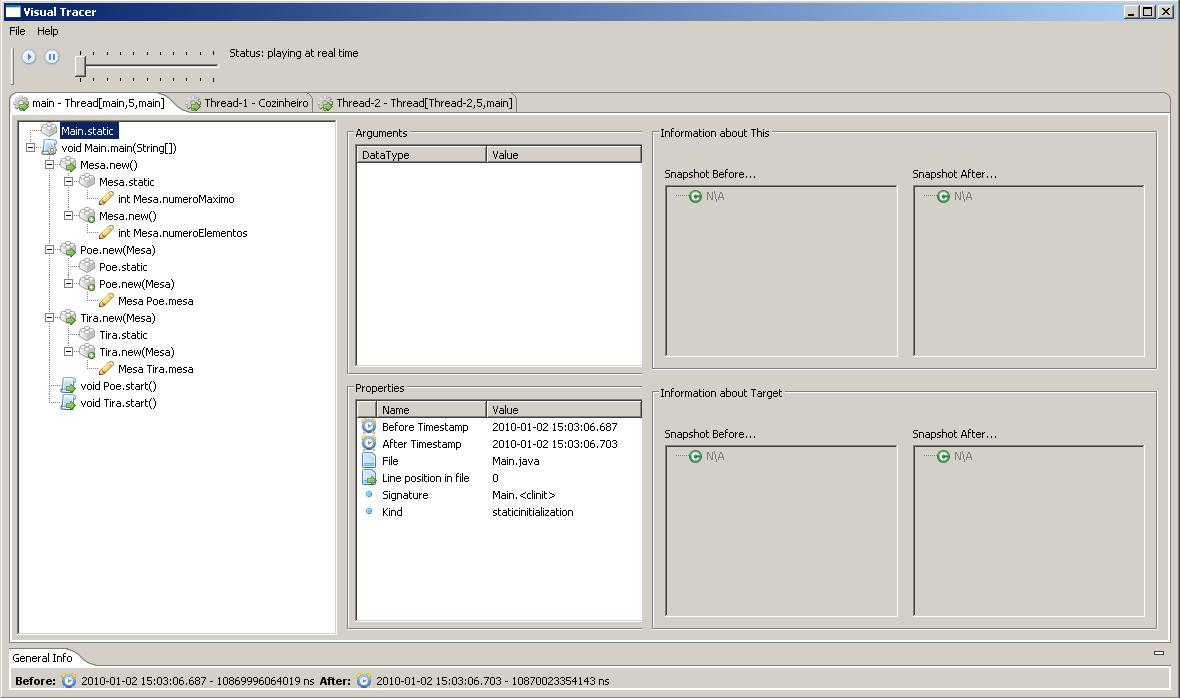
Algumas das funções pelas quais o controlador do *Visual* *Tracer* é responsável são listadas a seguir.

* Tornar a interface gráfica visível;
* Consultar o estado de instâncias no modelo de dados;
* Consultar e ordenar a adição de estruturas sintácticas;
* Ordenar a adição ou alteração de dados no modelo de dados;
* Consultar e solicitar a alteração ao tempo de execução da aplicação;
* Consultar e solicitar a alteração do estado da aplicação (*playing*/*paused*).

## Interface gráfica (View)

A camada de interface gráfica encontra-se dividida em pequenos componentes, recorrendo assim ao padrão de desenho *Composite*. A Imagem 1 ilustra a divisão dos componentes da janela principal da interface através de um *screenshot* da aplicação para melhor percepção por parte de quem pretender evoluir o projecto.

Como se pode observar na Imagem 1, a janela principal (*PrincipalView*) contém no seu interior outros componentes. O componente *TraceView* corresponde ao conteúdo de cada separador (*tab*) da interface que apresente o fluxo de execução de uma *thread*. O componente *DetailTraceView* está contido dentro do componente *TraceView* e consiste numa secção da interface que apresenta informação sobre dois *snapshots* de uma instância (*after* e *before*).



***PrincipalView***

***DetailTraceView***

***DetailTraceView***

***TraceView***

Imagem – Estrutura da interface gráfica

Este componente é replicado para mostrar a informação sobre a instância *this* e sobre a instância *target*, ou seja, existem dois objectos do tipo *DetailTraceView* dentro do *TraceView*.

O componente *InstanceDetailView* consiste no conteúdo de um separador, à semelhança do que acontece com *TraceView*, com a diferença de que apresenta uma vista histórica sobre uma determinada instância.

## AspectJ – *Tracing*

Embora seja possível implementar rotinas de *tracing* através de programação orientada a objectos, o código relacionado com esta tarefa espalhar-se-ia ao longo de todo o projecto, tornando-o difícil de compreender e de alterar. Este tipo de requisitos não passíveis de serem isolados numa rotina em programação por objectos – as preocupações transversais (*cross-cutting concerns*) –, constituíram a grande motivação para o surgimento da programação orientada a aspectos.

O paradigma da programação orientada a aspectos permite o encapsulamento de requisitos transversais – como o *tracing* – num único local, que são depois colocados nos sítios adequados através do *entretecimento (weaving).* A criação de aspectos para implementação do *Visual Tracer* recorreu à extensão da linguagem de programação Java, o *AspectJ*.

Deste modo, todas as actividades referentes ao *tracing* têm início no ficheiro de aspectos da aplicação, *Profiler.aj*, que está contido no pacote dos aspectos, *aspects*.

No arranque da aplicação, na primeira inicialização de uma classe e apenas na primeira, uma nova *thread* é lançada e encarregue de criar a interface gráfica e executá-la, para não interferir com o curso normal da aplicação a rastrear, como ilustra a Diagrama 5.

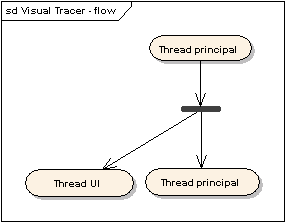


Diagrama – Fluxo de *threads* do *Visual Tracer*

A *Thread UI* estará encarregue de actualizar os dados da interface. Assim, quando o utilizador decide terminar a aplicação *Visual Tracer*, o programa a ser rastreado continua a sua execução normalmente. Recorre-se, então, a programação por aspectos para capturar a inicialização de uma classe para verificar se é a primeira para lançar a *thread UI*, através do seguinte ponto de corte (*pointcut*):

Excerto de código – Ponto de corte de inicialização de classe

**pointcut** classInitialization() : staticinitialization(\*);

Todos os conselhos (*advices*) implementados são do tipo *around*, o que possibilita a execução de tarefas antes e depois do ponto de junção em questão. Isto é especialmente importante, uma vez que a construção dos objectos para guardar informação sobre a execução (objectos do tipo *Event*) necessita de informação anterior e posterior ao ponto de junção. Para além disso, o tipo de conselho *around* permite o controlo do tempo de execução do programa rastreado para ir de encontro à funcionalidade da definição de intervalo de tempo de execução entre pontos de junção, disponível a partir da interface do *Visual Tracer*, como veremos adiante.

Foram definidos conselhos no ficheiro de aspectos da aplicação para capturar cada um dos acontecimentos possíveis na execução do programa a rastrear, sendo esses acontecimentos:

* A inicialização de uma classe;
* A invocação de um construtor;
* A execução de um construtor;
* A invocação de um método;
* A execução de um método;
* A leitura de um atributo;
* A escrita de um atributo.

A captura dos acontecimentos acima descritos exclui, como é óbvio, a execução do próprio *Visual Tracer*, pois o objectivo é observar apenas o comportamento do programa a rastrear, sem qualquer tipo de informação sobre a aplicação designada para o fazer.

Todos os conselhos executam um conjunto de tarefas semelhantes. Aquando da captura de um ponto de junção, verifica-se qual a *thread* correspondente recorrendo, para isso, às instância da classe *ThreadFlow*. Neste momento anterior ao ponto de junção são criadas duas instâncias da classe *Instance*, para representar os objectos *this* e *target* e depois é criada a instância da classe *Event*. De acordo com o ponto de junção capturado, serão criados o objecto da estrutura de fluxo de execução e o objecto correspondente na estrutura sintáctica. A Tabela 1 apresenta a correspondência entre as duas estruturas do modelo da aplicação. De seguida, é invocado o comando proceed() para que o programa rastreado continue a sua execução. No momento após a execução do ponto de junção, uma nova instância da classe *Event* é criada para representar o estado das instâncias da classe *Instance* (*this* e *target*) neste momento. Caso se trate de um ponto de junção de escrita de um atributo, e apenas neste caso, são criadas duas instâncias da classe *Snapshot* para guardar o novo estado dos objectos *this* e *target* (ver ).

|  |  |
| --- | --- |
| Estrutura de fluxo de execução | Estrutura sintáctica |
| *FieldRead* | *Attribute* |
| *FieldWrite* | *Attribute* |
| *StaticInitialization* | *StaticInitializer* |
| *ConstructorCall* | *Constructor* |
| *ConstructorExecution* | *Constructor* |
| *MethodCall* | *Method* |
| *MethodExecution* | *Method* |

Tabela – Correspondência entre classes das estruturas de fluxo de execução e sintáctica

Em suma, todos os pontos de junção de um programa são capturados e apresentados na aplicação, que apresenta uma árvore de eventos para cada *thread* lançada pelo programa rastreado. De realçar que o mecanismo apresentado permite a captura de criação e execução de interfaces gráficas.

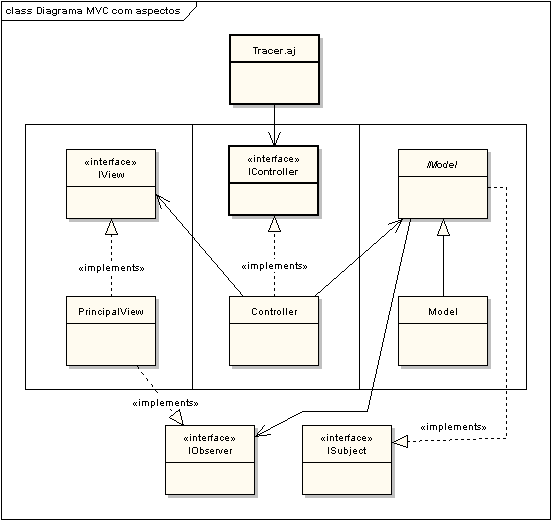


Diagrama – Relação entre aspectos e a arquitectura MVC

Como é visível no Diagrama 6, os aspectos descritos nesta secção comunicam com a camada de controlo, responsável pela comunicação com a interface gráfica e com o modelo de dados. A ligação entre os aspectos e o controlo é feita, mais uma vez, através da interface *IController*, para permitir um melhor encapsulamento das camadas.

## Casos de uso (use cases)

Os utilizadores podem interagir com a aplicação *Visual Tracer* de várias formas. Estas formas de interacção gerais são descritas através de um diagrama de casos de uso (*use* *cases*).

O Diagrama 7 apresenta os casos de uso gerais. O utilizador poderá controlar o tempo de execução da aplicação, isto é, definir o ritmo a que o programa é executado. Para além disso, poderá também alterar o estado da aplicação, pausando a execução do programa e ordenando depois que este se volte a executar. A acção de visualizar evento consiste na selecção de um dos itens na árvore de uma das janelas de execução de uma *thread*, sendo-lhe mostrada a informação relativa a esse mesmo evento.

As instâncias poderão ser observadas individualmente, sendo para isso necessário seleccionar uma instância numa das janelas de execução de uma *thread*. Após a selecção de uma instância, o novo separador que contém o histórico de uma instância permite ver a informação detalhada acerca de um *snapshot* dessa instância. Este separador poderá ainda ser fechado pelo utilizador.

As opções de visualizar ajuda, visualizar a janela “*About*” e de sair da aplicação estão disponíveis a partir do menu da aplicação. A janela “*About*” contém informação acerca da realização do projecto, os seus autores e o âmbito académico.

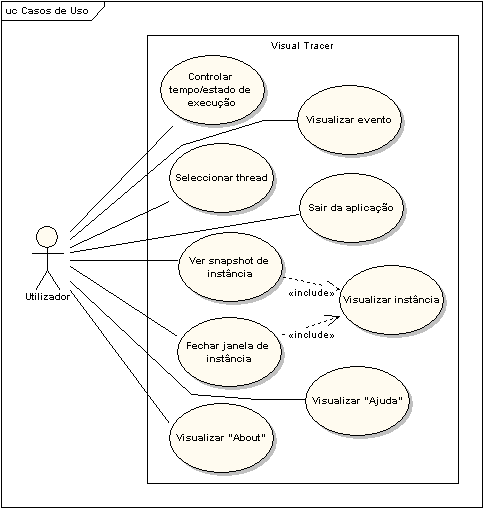


Diagrama – Casos de uso gerais

## Diagramas de sequência

Apresentam-se em seguida dois diagramas de sequência que visam esclarecer o funcionamento do Visual Tracer. Estes diagramas pretendem demonstrar o funcionamento base da aplicação através de um exemplo concreto da invocação de um construtor, sendo ilustrativos de todos os casos de ocorrência de um ponto de junção.

O primeiro pretende dar a conhecer o fluxo normal de execução, desde a ocorrência de um ponto de junção até à adição do novo item. O segundo esclarece de que forma a interface gráfica é notificada e altera em tempo real a árvore do fluxo de execução.

### Exemplo de fluxo

Inicialmente o estado e o tempo de execução são verificados. Depois, procura-se a instância de *ThreadFlow* (que representa uma *thread* do programa rastreado) que despoletou este ponto de junção. É também verificada a existência das instâncias correspondentes aos objectos *this* e *target* envolvidos no ponto de junção.

Como se trata de um exemplo em que o ponto de junção é uma invocação de um construtor, não são criados *snapshots* das instâncias (apenas são criados quando ocorre a escrita de um campo), como será explicado mais à frente. A estrutura sintáctica correspondente ao ponto de junção que ocorreu é devolvida após uma procura, para deste modo verificar se o valor devolvido é nulo e criar a estrutura sintáctica correspondente. No caso ilustrado o valor não era nulo, o que significa que a estrutura sintáctica em causa já estava criada. Em seguida, será recolhida a informação de contexto do ponto de junção, nomeadamente:

* O nome do ficheiro onde este ponto de junção se encontra;
* O nº da coluna no código onde se encontra;
* O nº da linha de código onde se encontra;
* A assinatura do ponto de junção;
* O tipo de ponto de junção.

Esta e toda a informação guardada até este ponto são utilizadas para criar a instância de *AbstractJoinPoint* (classe de topo da estrutura de fluxo de execução, ver 2.2.2). A instância criada é metida no *stack* da instância *ThreadFlow* para fins de controlo de fluxo de execução, como será explicado adiante. A camada de controlo é invocada para guardar no modelo de dados este novo ponto de junção (*AbstractJoinPoint*). A alteração ao modelo de dados irá repercutir uma notificação na camada de interface. Deste modo, a interface poderá mostrar esta nova informação em tempo real. Esta notificação irá dar origem ao diagrama explicitado no tópico seguinte. Nesta altura, o *Visual* *Tracer* indica ao programa rastreado que pode continuar a execução daquele ponto de junção, para depois guardar a informação sobre o estado das instâncias após da ocorrência do ponto de junção. Por fim, resta libertar o ponto de junção do *stack* e adicioná-lo à estrutura que guarda o fluxo de execução daquela *thread*.

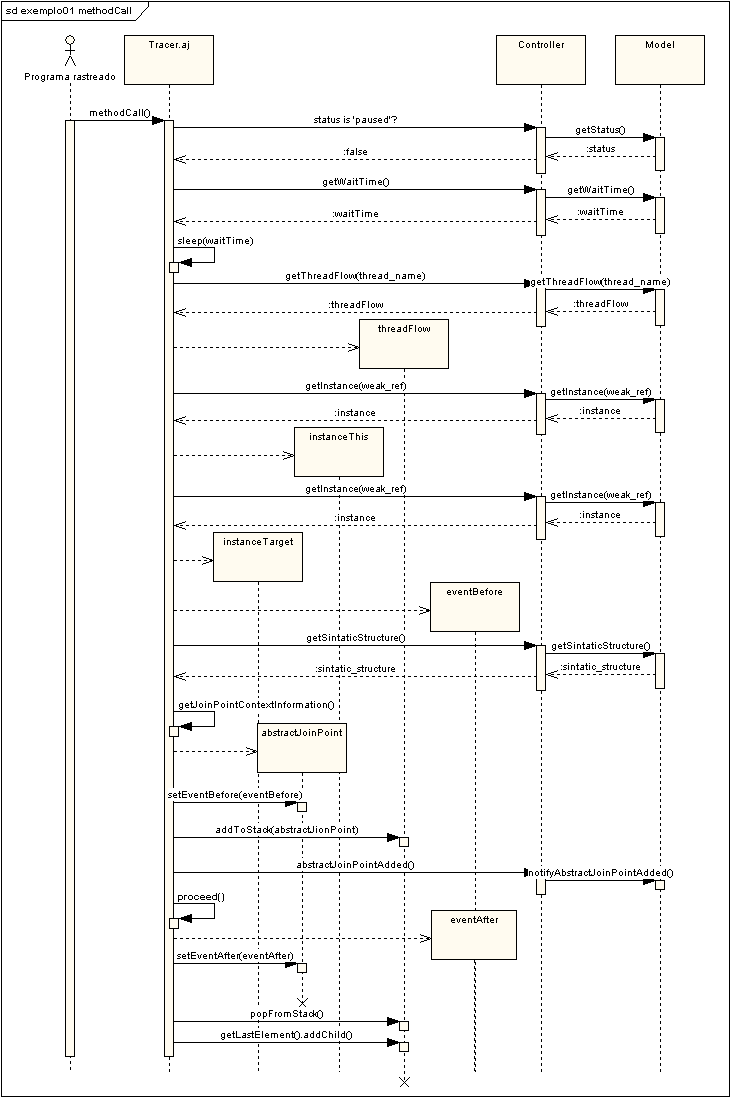


Diagrama – Exemplo de fluxo de execução do *Visual Tracer*

### Exemplo de actualização da interface gráfica

Partindo do exemplo acima e do ponto em que é enviada uma notificação para a camada de controlo, esta por sua vez envia a notificação para o modelo. O modelo envia a notificação para todas as classes registadas para receber esta notificação, ou seja, para todas as classes que implementam a interface *Observer*, como é o caso da classe que representa a janela principal da aplicação – *PrincipalView*. Esta classe da camada de interface gráfica recebe a notificação que, neste caso, implica uma alteração num dos seus componentes – o *TraceView* – que apresenta a informação relativa ao fluxo de execução de uma *thread* (ver Imagem 1). Deste modo, a janela principal comunica ao seu componente que deve inserir um novo item na sua árvore de execução.

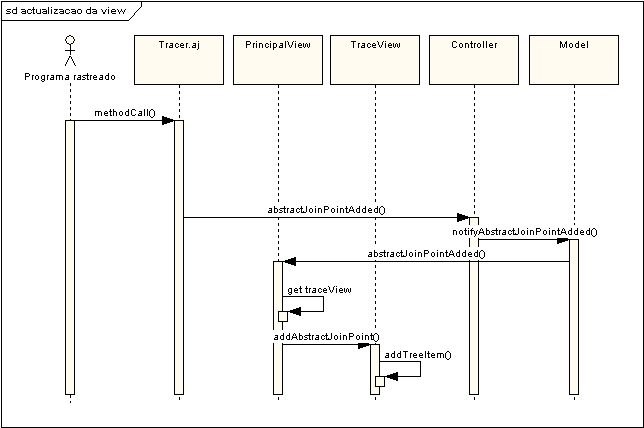


Diagrama – Actualização de interface gráfica

## Opções tomadas

As decisões enunciadas neste tópico estão relacionadas com a definição de âmbito do projecto de acordo com o tempo disponível para a sua realização.

Como já referido anteriormente (em 2.2.1), a estrutura sintáctica do *Visual* *Tracer* não contempla os tipos (ou extensões da classe *Type*) *Interface*, *Aspect* e *Annotation*, pois estes tipos divergem do tipo *Class*, tendo as suas próprias peculiaridades.

As classes embutidas não foram igualmente implementadas na aplicação. Em Java, as classes embutidas, isto é, classes definidas dentro de outras classes, dividem-se da seguinte forma:

* **Classes estáticas embutidas** (*static nested classes*), quando são classes estáticas

**public** **class** Lista {

…

**public** **static** **class** Nó {

**int** i;

Nó próximo, anterior;

}

}

* **Classes internas** (*inner classes*), quando não são classes estáticas,

**public** **class** Botão {

…

**public** **class** SentinelaDoBotão {

…

}

}

e podem ser ainda:

* + **Classes locais**, que são definidas dentro de um método

**public** **class** Animal {

…

**public void** tornaRacional() {

**public class** Racional {…}

}

}

* + **Classes anónimas**, definidas sem nome dentro de um método

**public** **class** Animal {

…

**public** Racional tornaRacional() {

**return new** Racional(){…};

}

}

Esta categorização de classes embutidas está de acordo com a matéria respectiva leccionada na cadeira de Programação Concorrente e Distribuída. Cada tipo de classe embutida contém, para além da definição, um pequeno exemplo ilustrativo em Java.

Esta categorização de classes embutidas é útil para futuramente o *Visual Tracer* conseguir capturar e rastrear os eventos a elas associados.

# Implementação da aplicação

Esta secção pretende explicar todos os mecanismos e decisões tomadas na implementação da aplicação *Visual Tracer*.

## Decisões tomadas

## Representação das instâncias por *MyWeakReference*

A implementação de uma aplicação de *tracing* implica guardar o histórico de eventos que aconteceram na execução do programa rastreado, bem como as instâncias criadas e seu estado ao longo do tempo. O armazenamento das instâncias implica, como se sabe, apontar uma referência para elas. Contudo, através da referenciação normal no Java, uma instância só é finalizada e removida da memória (pelo *Garbage* *Collector* – GB) quando não existe nenhum caminho de referências para chegar até ela, o que implicaria manter todas as instâncias em memória ao rastrear uma aplicação até que esta fosse terminada.

Deste modo, é necessário implementar um mecanismo que recorre a referências fracas que estabelecem uma ligação às instâncias, mas permitem a finalização e eventual desocupação dos objectos em memória. A hierarquia dos quatro tipos de referência em Java encontra-se esquematizada na Imagem 2, sendo a de topo a mais forte e a de base a mais fraca. Como se pode ver, existem três tipos de referências fracas, que são explicadas em seguida de um modo geral:

* **Referências leves** (*SoftReference*)) – instâncias cuja referência mais forte é deste tipo serão limpos consoante os critérios do GB em relação ao consumo de memória;
* **Referências fracas** (*WeakReference*) – instâncias cuja referência mais forte é deste tipo estarão disponíveis para serem finalizados;
* **Referências fantasma** (*PhantomReference*) – instâncias cuja referência mais forte seja deste tipo não são passíveis de retornar a instância (este tipo de referências é geralmente utilizado para conhecer o momento em que uma instância é efectivamente removida da memória).

Imagem – Hierarquia de referências em Java

A aplicação *Visual Tracer* recorre a referências fracas para estabelecer uma ligação às instâncias rastreados, sem implicar que as mesmas fiquem presos em memória. Para tal, recorre à classe *WeakReference<T>*.

Ainda assim, para além da implementação de referências fracas, persiste o problema de como aceder à informação acerca de uma instância após a mesma ter sido removido de memória. Mais concretamente, a comparação de instâncias (melhor dizendo, de referências que apontam para instâncias) torna-se impossível quando a instância é retirada da memória.

A solução encontrada resultou na implementação da classe *MyWeakReference*, que pretende representar a ligação para uma instância, sem a prender na memória, e ainda permitindo a comparação entre instâncias. Para isso, é criado um identificador único para cada instância, que é guardado na instância de *MyWeakReference* respectiva. Os métodos hashCode() e equals() foram sobrescritos de modo a garantir a conformidade com os requisitos já explicados.

## Identificação de threads

A gestão de *threads* rastreadas pela aplicação é feita recorrendo a uma classe que representa cada uma das *threads* e o seu respectivo fluxo de execução – *ThreadFlow*.

Uma instância da classe *ThreadFlow* está, portanto, associada a uma *thread* da aplicação a ser rastreada, existindo sempre pelo menos uma instância que corresponde à *thread* *main*. Cada instância possui um identificador incremental e um nome alternativo, que corresponde ao endereço de memória e poderá ser definido pelo utilizador caso ele sobrescreva o método toString() na implementação da *thread* na sua aplicação.

Cada instância de *ThreadFlow* guarda também o seu fluxo de execução, armazenando para isso um conjunto de pontos de junção abstractos – instâncias de *AbstractJoinPoint* –, que representam os nós raiz da sua árvore de execução (estes nós raiz contêm, por sua vez, os respectivos filhos). A árvore de execução é construída com o auxílio de um *stack* de *AbstractJoinPont*, onde são colocados os eventos que ocorrem e retirados assim que terminam.

Em seguida é apresentado um exemplo ilustrativo do funcionamento do stack que cada instância de ThreadFlow contém para controlar o seu fluxo de execução.

**public** **class** Teste {

**public** **static** **void** main(String[] args)

{

Pessoa pai = **new** Pessoa("José");

}

}

**public** **class** Pessoa {

**private** String primeiroNome;

**public** Pessoa(String primeiroNome)

{

**this**.primeiroNome = primeiroNome;

}

}

Excerto de código – Exemplo de criação de instância de *Pessoa*

O Excerto de código 2 acima ilustra um exemplo simples de um programa cujo objectivo é a criação de uma instância da classe *Pessoa*. A classe *Teste* é a classe que contém o método *main*, que contém a instrução para criar uma nova instância de *Pessoa*.

A classe Pessoa, por sua vez, possui um atributo do tipo *String* *primeiroNome* e um construtor que recebe como argumento uma *String*. Aquando da execução do construtor, o atributo *primeiroNome* será alterado para o valor recebido como argumento.

Vejamos então a evolução do *stack* na Imagem 3 tendo em conta os eventos despoletados pela execução do excerto de código apresentado acima. A instância *ThreadFlow* que contém este *stack* corresponde à instância que representa a *thread* principal do programa composto pelo excerto de código.

Cada evento do *stack* é representado por um bloco. A ordem cronológica do fluxo de execução é de cima para baixo, ou seja, o evento mais acima foi o primeiro a ocorrer e assim sucessivamente. As imagens no canto superior esquerdo de cada disco são as imagens utilizadas na árvore de fluxo de execução para distinguir cada tipo de evento na aplicação *Visual* *Tracer*.

FieldWrite.pngMethodExecution.pngStaticInitialization.pngMethodExecution.pngConstructorCall.pngStaticInitialization.pngConstructorExecution.pngConstructorCall.png

String Pessoa.primeroNome

Pessoa.static

Pessoa.new(String)

Teste.main

(String[])

Pessoa.new(String)

Pessoa.new(String)

Teste.main

(String[])

Teste.static

Imagem – Exemplo de stack da criação de instância de *Pessoa*

O primeiro evento que ocorre naquele exemplo é a inicialização estática da classe *Teste*. O evento é então adicionado ao *stack* da instância *ThreadFlow* correspondente, resultando na primeira pilha da esquerda da Imagem 3. Este evento não tem sucessores e é portanto removido do *stack* logo após a sua execução.

O evento que se segue é a execução do método *main*. Como podemos ver no Excerto de código 2, o método *main* apenas contém a instrução de criação de uma instância da classe *Pessoa*, através da invocação do construtor, evento que será também capturado pelo *Visual* *Tracer* e guardado no *stack*. Este evento ocorre no contexto da execução do método *main*, portanto torna-se “filho” deste e nenhum evento é ainda retirado do *stack*, como se pode observar na segunda pilha da Imagem 3. O seguinte evento é a inicialização estática da classe Pessoa, que não contém nenhum evento dependente de si (nenhum evento é executado no contexto deste). Como tal, este evento é também imediatamente removido do *stack* após a sua execução.

De seguida, ocorre o evento de execução do construtor da classe Pessoa, ainda dentro do contexto do evento de invocação do construtor. Este evento é também adicionado ao *stack*. O evento que se segue é a escrita de um atributo, ou seja, a atribuição de um valor recebido como argumento do construtor ao atributo *primeiroNome*, também adicionado ao *stack*. Tanto as operações de leitura de atributos como de escrita de atributos são operações atómicas, o que significa que não irão ocorrer eventos dento do seu contexto. Assim sendo, após a escrita do atributo *primeiroNome*, o evento de escrita será de imediato removido do *stack*, como é ilustrado na terceira pilha da Imagem 3.

Como após a escrita do atributo não existem mais instruções, a execução do construtor termina, sendo este evento removido do *stack*. É então terminado também o evento de invocação do construtor, que é também removido do *stack*, o que nos leva de volta ao evento de execução do main. Este evento é também removido do *stack*, uma vez que dentro do corpo do *main* não existem mais instruções a executar. A remoção de todos os eventos do *stack* converge com o término da execução da *thread* à qual estes eventos diziam respeito.

## Criação e selecção de *snapshots*

A representação do estado de uma instância num determinado instante de tempo é feita através da criação de *snapshots* dessa instância, isto é, os *snapshots* capturam o estado de uma instância quando esta é alterada.

Uma vez que a função de um *snapshot* é mostrar o estado de uma instância, os *snapshots* (instâncias da classe *Snapshot*) são criados cada vez que ocorre a alteração do estado de uma instância, ou seja, de cada vez que ocorre uma escrita de um atributo de uma instância. Um snapshot contém:

* Um vector de campos (*fields*), que correspondem aos valores dos atributos de uma instância;
* O nome da classe ao qual pertence a instância modificada pelo *snapshot*;
* Um *timestamp* que regista a ocorrência do *snapshot*;
* O valor dos nanossegundos em que o *snapshot* ocorreu;
* A representação da instância a que diz respeito o *snapshot*;
* O ponto de junção referente à ocorrência do *snapshot*.

De qualquer modo, todos os eventos que ocorreram durante a execução de um programa estarão associados a um *snapshot*, correspondente ao estado que a instância possuía naquele momento.

Assim, quando o utilizador selecciona uma instância de um evento da lista de eventos na janela de fluxo de execução de uma thread, a janela de instância irá abrir-se e terá um *snapshot* seleccionado. A selecção do *snapshot* consiste em procurar o *snapshot* com um *timestamp* mais próximo e menor que o do evento da instância. Um *timestamp* posterior ao do evento da instância, ainda que seja mais próximo, implica que o estado da instância daquele *timestamp* seja diferente do estado da instância aquando da ocorrência do evento em questão.

## Interface gráfica – *Standard Widget Toolkit* (SWT) e JFace

Para a implementação da interface gráfica, a escolha recaiu sobre a tecnologia *Standard Widget Toolkit* (SWT), que surge como alternativa às já existentes *toolkits* de desenvolvimento de interfaces em Java – *Abstract Window Toolkit* (AWT) e *Swing*.

O SWT difere das *toolkits* anteriores por estar integrado com o sistema de janelas nativo de cada sistema operativo e por permitir o acesso a funcionalidades como as interacções nativas (como o *drag and drop*) e componentes específicos do sistema operativo (como controlos ActiveX do Windows). O JFace é uma *toolkit* que auxilia o programador na tarefa de desenvolvimento de interfaces gráficas em SWT, tornando assim a tarefa de programação bastante simples.

As interfaces *look*-*and*-*feel* do *Swing* melhoram a interacção com o utilizador ao emular as interfaces gráficas do sistema operativo, mas esta abordagem é problemática tendo em conta que os sistemas operativos têm diferentes temas gráficos que podem ser definidos pelo utilizador. Além disso, não é favorável para o utilizador lidar com diferentes tipos de interface.



Imagem – Janela de upload de imagem SWT [fonte: ]



Imagem – Janela de upload de imagem *Swing* [fonte: ]

Nas imagens ilustradas acima podemos verificar que a interface desenvolvida em SWT para o upload de imagens torna-se mais familiar para o utilizador, neste caso no Windows, possibilitando ainda outro tipo de funcionalidades, como o acesso rápido às propriedades da imagem.

Outra grande vantagem face ao *Swing* é a rapidez dos componentes. Embora o *Swing*, tenha vindo a melhorar neste campo, ainda fica um pouco aquém em comparação com o SWT. A Tabela 2 demonstra uma comparação informal, feita em , das velocidades de três carregamento da janela de upload de imagens em *Swing* e em SWT.



Tabela – Comparação de velocidade *Swing* *vs*. SWT

Como se pode observar, o tempo total utilizado para o carregamento da janela em *Swing*, em média, é muito superior ao do SWT.

Em termos de portabilidade, o SWT é compatível com os sistemas operativos do Windows (98/ME/2000/XP/Vista/7), Linux, Solaris 8, QNX, AIX, HP-UX e Mac OS X. Devido a este amplo conjunto de plataformas compatíveis e outros motivos, alguns deles enunciados anteriormente, o programa de desenvolvimento de aplicações em várias linguagens, Eclipse, recorreu ao SWT para a implementação da sua interface gráfica.

Numa perspectiva de desenvolvimento, o SWT é bastante flexível por permitir um desenvolvimento tradicional de interfaces gráficas ou um desenvolvimento recorrendo ao padrão de desenho MVC. Para isso, o SWT disponibiliza já um conjunto de classes desta arquitectura, como é o caso do *Composite*, utilizado por nós para construir a interface do *Visual* *Tracer*.

# Instalação

Consultar o *Manual de Instalação e Utilização Geral*.

# Utilização

## Manual do utilizador

Para instruções gerais de uso, consultar o *Manual de Instalação e Utilização Geral*.

## Exemplos de utilização

Como até aqui tem sido referido, o *Visual Tracer* é uma ferramenta de rastreio de execução de código de aplicações. Todos os conceitos inerentes à concepção e desenvolvimento já foram atrás explicitados. Porém, a potencialidade desta ferramenta não foi totalmente explorada neste documento, no sentido em que a sua efectiva utilidade ainda não ficou clara.

Nesta secção pretende-se demonstrar casos em que o rastreamento de código é importante e onde o *Visual* *Tracer* se destaca por permitir o rápido acesso à informação necessária de um modo simples e visual.

#### Ordem de execução

Com o Visual Tracer é possível explorar a própria linguagem de programação de uma aplicação, verificando quais as operações que são executadas em primeiro lugar. Este tipo de estudo torna-se bastante útil para compreender melhor a linguagem e o comportamento de cada aplicação nela escrito por parte de programadores principiantes, mas também por parte de programadores avançados que desejam conhecer a linguagem.

A título de exemplo, criou-se uma classe *Pessoa*, com um atributo estático – *numeroPessoa* – e três atributos privados – *primeiroNome*, *ultimoNome*, *pai* e *idade*. A Imagem 6 apresenta o resultado de execução no *Visual* *Tracer* da criação de uma pessoa. Como podemos observar, após a invocação do construtor da classe *Pessoa*, no método *main*, é inicializado o atributo estático daquela classe antes de qualquer outra coisa.

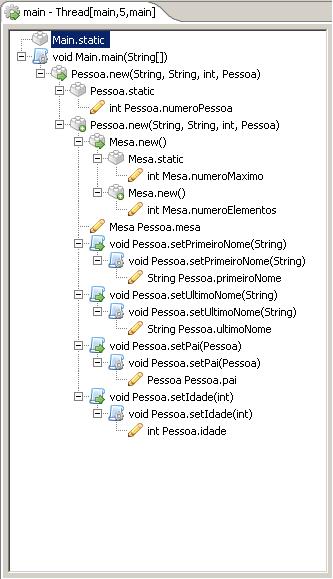


Imagem – Exemplo de ordem de execução

#### Multi-threading

A capacidade de rastrear a execução de múltiplas threads de uma aplicação torna-se uma mais valia para os utilizadores do Visual Tracer. Em primeiro lugar, o utilizador consegue percepcionar qual o número de threads que está em execução a cada instante de tempo, obtendo assim uma visão transparente da execução da aplicação rastreada.

O rastreio deste tipo de aplicações é complicado, tendo que recorrer a escritas na consola que, muitas vezes, não são fiéis à execução real por causa da sincronização das *threads*. Para além disso, as escritas na consola contribuem para um código confuso, difícil de entender e, estando espalhadas por diversos módulos, causam o emaranhamento de código.

Para ilustrar as funcionalidades do *Visual Tracer* em questões de *multi*-*threading*, recorremos a um exemplo académico em que existem dois tipos de entidade, *Cozinheiros* e *Glutões*. Os primeiros são responsáveis por colocar javalis na mesa, enquanto fica a cargo dos últimos comer os javalis disponíveis na mesa. A *mesa* é, portanto, o recurso partilhado entre ambas as entidades. A mesa inicialmente encontra-se vazia, pelo que se um glutão tentar comer irá ter que esperar que um cozinheiro coloque um javali na mesa. A capacidade máxima da mesa é de três javalis, pelo que se um cozinheiro tentar colocar um quarto javali na mesa, terá que esperar que algum dos glutões coma.

Um excerto do resultado do rastreio da aplicação descrita anteriormente através de escrita na consola é ilustrado na Imagem 7.

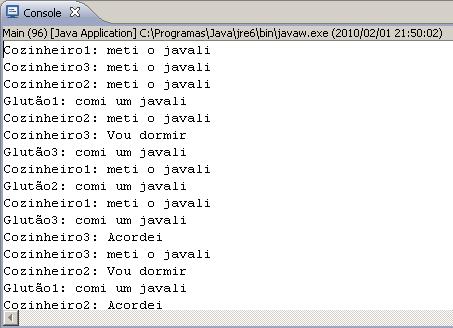


Imagem – Rastreio *multi-threading* através de escrita na consola

Como se constata, a análise da execução torna-se confusa pois todo o output está misturado na consola. Para além disso, perde-se a noção de ordem de execução, sendo impossível saber que acontecimentos sucederam os outros.

Por oposição, a Imagem 8 mostra o resultado do rastreio da execução da mesma aplicação através do *Visual Tracer*. Antes de mais, em tempo real é possível saber quantas threads existem pelo número de separadores de threads presentes no *Visual* *Tracer*. Como podemos ver, no final da execução da aplicação rastreada existiram, no total, sete *threads*: uma *thread* principal (*main*), três *threads* de *Cozinheiro* e três *threads* de *Glutão*.

O separador da instância de *Mesa* revela-se uma grande mais-valia na análise porque, sendo este o recurso partilhado, permite observar a evolução do objecto ao longo do tempo. Neste caso, permite-nos ver em cada momento qual a quantidade total de javalis na mesa.

Tratando-se do rastreio de aplicações com várias *threads*, não poderíamos deixar de ter em conta os problemas que a sincronização acarreta, tais como os *deadlocks*. Através do *Visual* *Tracer*, o utilizador detecta rapidamente a ocorrência de um *deadlock*, quando na execução em tempo real verifica que a *thread* não avança para além da execução do método *wait*, como mostra a Imagem 9. Além disso, os *snapshots* dos objectos *this* e *target* que deveriam ser mostrados após a ocorrência do evento em causa (neste caso, após a execução do método *wait*) não se encontram disponíveis, sendo mostrada na área respectiva da aplicação a mensagem “*Waiting*….”.

Por fim, o rastreio de aplicações *multi*-*threading* recorrendo ao *Visual* *Tracer* permite perceber que *thread* é acordada e ganha o acesso ao recurso partilhado, quando a velocidade de execução definida pelo utilizador é lenta.

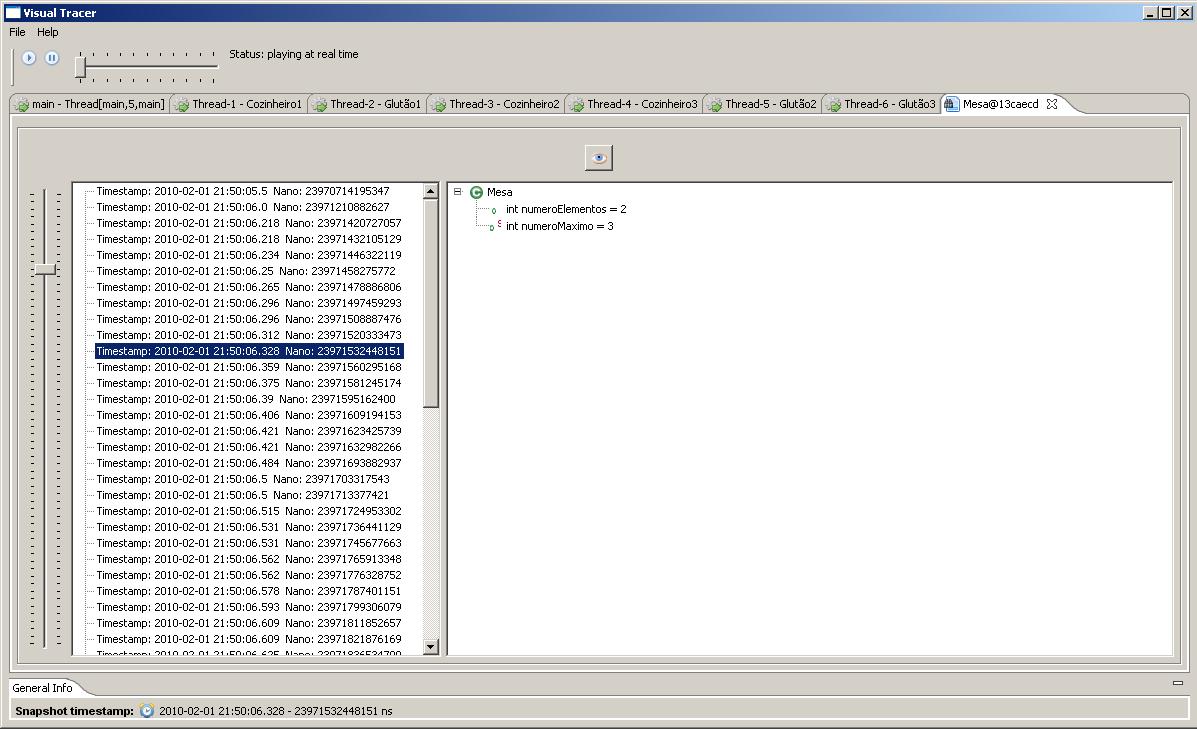


Imagem – Rastreio *multi*-*threading* com o *Visual* *Tracer*

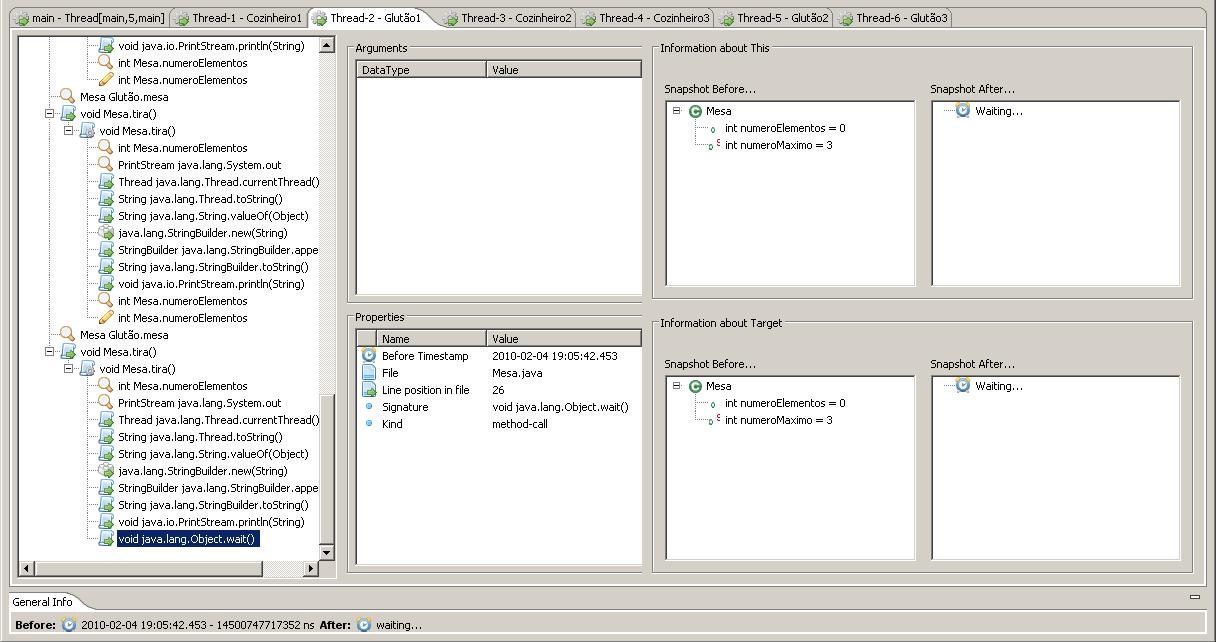


Imagem – Detecção de *deadlock* no *Visual* *Tracer*

# Discussão

A programação orientada a aspectos mostra-se bastante útil para o encapsulamento de certos módulos de código – ou preocupações transversais – como a autenticação, a escrita de *logs* e a instrumentação de código. Mais do que isso, a programação orientada a aspectos surge como um complemento necessário às limitações do paradigma de programação orientado a objectos.

O padrão de desenho MVC, para além de estruturar a aplicação em camadas lógicas, possibilita a implementação de novas extensões facilmente, tanto na camada de interface gráfica, como na camada de modelo de dados. A independência obtida através desta arquitectura é uma grande mais-valia na adição de novas classes em cada camada.

O modelo de dados concebido, correspondente à camada do modelo de dados do padrão de desenho MVC, não só satisfaz os requisitos de um projecto de *tracing* através da estrutura de fluxo de execução, como também suporta outros tipos de instrumentação de código através da estrutura sintáctica, que já se encontra construída. Deste modo, a adição de funcionalidades de *profiling* está facilitada pela estrutura concebida.

Percepcionando o *Visual* *Tracer* como um todo, pensamos ser uma ferramenta muito útil nas tarefas de *debugging* do código de uma aplicação, uma vez que permite analisar todo o fluxo de execução de uma aplicação de uma forma limpa e ordenada. Poderá também auxiliar a aprendizagem da linguagem de programação Java, uma vez que demonstra todos os eventos ocorridos por ordem cronológica e permite a análise de toda a informação de execução recorrendo a uma interface gráfica onde os conteúdos se encontram ordenados segundo o contexto. Em suma, a possibilidade de ter potencialidades presentes na linguagem de programação orientada a aspectos numa aplicação gráfica possibilita aceder a informação valiosa para o utilizador de uma forma rápida e simples, possibilitando ainda o despertar para a descoberta da própria linguagem Java.

# Trabalho futuro

Como foi referido ao longo do documento, o projecto foi concebido e desenvolvido sob o modelo MVC, para permitir o encapsulamento das diferentes camadas do projecto (modelo de dados, controlador e apresentação visual). Desta forma, as extensões à aplicação poderão ser feitas de um modo mais simples.

Nesta secção são descritas algumas extensões à aplicação *Visual Tracer*. As extensões e soluções propostas foram surgindo numa fase final do Project, quando o tempo disponível para a sua execução era já bastante reduzido. Acreditamos que estas extensões melhoram, em muito, a utilização do *Visual Tracer*.

#### Interface básica e interface avançada

Sendo o *Visual Tracer* uma ferramenta de rastreamento de execução de código, deve-se ter em conta que o nível de conhecimento de programação pode variar entre os utilizadores. Para responder às necessidades de grupos de utilizadores em particular, é evidente a criação de perfis de utilização, que serão reflectidos na interface da aplicação.

Poderão, para isso, ser definidos dois tipos de interface gráfica: básica e avançada. Aquando da primeira utilização da aplicação, o utilizador teria que optar pelo tipo de interface a ser apresentado. Esta informação seria guardada para que, posteriormente, o arranque da aplicação fosse feito sempre com o mesmo tipo de interface.

O primeiro tipo de interface seria dirigida a um grupo de utilizadores principiantes na programação, cujas funcionalidades principais seriam apresentar o fluxo de execução de um programa e apresentar o histórico de dados, permitindo a navegação entre estes dois tipos de vista. Deste modo, os utilizadores poderiam observar e compreender a ordem de execução do código e os conceitos básicos inerentes à programação orientada a objectos.

Já a interface avançada iria permitir a análise de outro tipo de questões mais complexas, como é o caso da programação concorrente. Esta interface iria permitir a navegação entre a vista de fluxo de execução e a vista de histórico de dados, como na interface básica, mas conteria ainda a vista de fluxo de execução por cada nova *thread* lançada, para além da *thread* principal (*main*). Estas vistas permitiriam observar o comportamento das *threads* no que diz respeito ao acesso a recursos partilhados, bem como o comportamento de cada *thread* em particular.

Como a evolução natural dos conhecimentos de programação é expectável, seria interessante disponibilizar ao utilizador a opção para alterar o tipo de interface da aplicação.

##### **Solução**

As funcionalidades descritas já estão implementadas, sendo apenas necessário designar quais delas estariam disponíveis em cada uma das interfaces. (…)

Quanto ao registo do tipo de interface escolhida, poderia ser feito recorrendo a um ficheiro de propriedades no formato XML. Na primeira utilização do programa, o ficheiro seria escrito com o tipo de interface seleccionado. O ficheiro XML seria lido em cada arranque da aplicação para que a interface seleccionada fosse correctamente apresentada. De acordo com o sugerido, seria ainda disponibilizada uma opção no menu da aplicação para que o utilizador pudesse alterar o tipo de interface, cuja execução resultaria na escrita dessa opção no ficheiro de propriedades.

#### Integração com Eclipse

Esta funcionalidade está relacionada com a criação de um *plug*-*in* para o Eclipse, de modo a que o utilizador pudesse usufruir das funcionalidades do *Visual* *Tracer* de uma forma transparente. O utilizador apenas teria que instalar o *plug*-*in* e, quando executasse a sua aplicação, apareceria instantaneamente a janela principal do *Visual* *Tracer*, que iria rastrear a aplicação.

Por uma questão de limite de tempo, não foi possível estudar a solução para esta questão da integração com o Eclipse.

#### Diagramas UML dinâmicos

A conceptualização de qualquer estrutura de programação é, geralmente, feita recorrendo a diagramas UML. Esta linguagem auxilia na modelação e especificação de aplicação recorrendo a diagramas “padrão”.

A representação da interligação entre as instâncias criadas na execução de um programa e do próprio fluxo de execução através de diagramas UML seria uma mais-valia para o Visual Tracer. O objectivo seria criar vistas que apresentassem a evolução de diagramas UML conforme a execução do código. Mais precisamente, seria interessante obter visualizações distintas dos diagramas de objectos, de sequência e de comunicação.

##### **Solução**

A grande dificuldade desta extensão reside na manipulação gráfica necessária para a apresentação dos diagramas, uma vez que o modelo de dados construído já serve de suporte para esta vista.

A implementação do diagrama de objectos poderá recorrer à utilização de grafos bidimensionais com algoritmos de disposição dos elementos gráficos num determinado espaço (como o algoritmo do caminho mais curto de *Dijkstra*). *JGraphT* e *JGraphEd* são o exemplo de duas ferramentas desse género.

#### Vista de estrutura sintáctica

Como referido anteriormente, o modelo de dados do *Visual Tracer* é composto por uma estrutura de fluxo de dados e por uma estrutura sintáctica, embora esta última não tenha sido alvo de exploração no projecto aqui retratado.

Para além de apresentar a informação sintáctica sobre a aplicação – como os pacotes, classes, atributos e métodos existentes, por exemplo – as funcionalidades da estrutura sintáctica serviriam para avaliar o desempenho de um programa.

Aqui ficam algumas sugestões:

* A duração da execução total da aplicação a instrumentar;
* O tempo de execução de cada método, com ou sem os tempos de execução de métodos que são invocados por ele;
* A duração média de execução de cada método;
* A evolução do espaço ocupado em memória ao longo do tempo;
* O número de objectos criados e o espaço respectivo que ocupam em memória;
* O número de vezes que um método é executado;
* A possibilidade de executar um determinado método com diferentes argumentos;

##### **Solução**

A implementação da vista de estrutura sintáctica está contemplada no modelo de dados da aplicação, sendo necessário adequar as classes existentes aos requisitos a implementar definidos pela equipa.

A duração de execução de métodos será feita no ficheiro *Profiler.aj*, no ponto de corte methodExecution() e/ou no ponto de corte methodCall(). A medição de tempos ocorrerá no início e no fim do ponto de corte, para depois calcular a diferença entre o tempo final e o tempo inicial. A unidade de medição terá que ser o nanossegundo (ns), recorrendo para isso à função System.nanoTime().

Existem alguns sítios na internet com conteúdos relacionados com a medição de utilização de memória na linguagem Java, sendo possível estimar a ocupação de um objecto em memória (ver Bibliografia).

#### Persistência de informação

Uma funcionalidade bastante útil seria o armazenamento de informação que a aplicação disponibiliza acerca de cada execução de um programa. Isto permitiria guardar os resultados de diferentes execuções para análise, comparação, ou até para envio para outros elementos ou equipas de desenvolvimento.

##### **Solução**

Os dados relativos a uma execução do programa a instrumentar poderiam ser armazenados numa pequena base de dados ou num ficheiro com o formato XML. As opções para gravar e abrir os ficheiros deveriam ser incluídas no menu da aplicação *Visual Tracer*. Esta mecanismo de persistência poderia ser facilmente implementado recorrendo a tecnologias de *Object Relational Mapping.* Por exemplo, utilizando a *framework* Hibernate[[1]](#footnote-1), seria possível criar uma estrutura de ficheiros XML que mapeavam todos os objectos na camada de dados para uma base de dados. A pedido do utilizador, os objectos em memória seriam persistidos automaticamente nas tabelas respectivas.

***Comunicar estados da aplicação***

Uma vez que o estado de execução pode ser controlado pelo utilizador através dos botões de controlo e da barra de velocidades, existe a necessidade de informar o utilizador acerca do estado da aplicação. As velocidades de reprodução e o estado de pausa são já apresentados, mas o utilizador não obtém nenhum feedback de quando termina a execução da aplicação rastreada. Seria também interessante comunicar que uma thread terminou a sua execução e até mesmo que uma instância deixou de existir em memória, tudo isto em tempo real.

##### **Solução**

Para comunicar o estado de término de execução da aplicação rastreada poderá ser utilizada a área onde já são comunicadas as velocidades de execução e o estado de pausa. Para isso, será necessário confirmar que todas as *threads* da aplicação terminaram a sua execução, verificando a cada intervalo de tempo (por exemplo, de 5 em 5 milissegundos) se a *thread* ainda se encontra “viva” (método isAlive() da classe *Thread*). Isto serve também para ir alterando o ícone de cada separador de *thread* assim que for detectado que a *thread* já não está em execução.

Quanto ao estado das instâncias, sugere-se o estudo e a utilização de referências fantasma (*phantom* *references*), uma vez que permitem saber quando o objecto foi finalizado, antes de ser apagado da memória.

# Bibliografia

Coordenação da cadeira de Programação Concorrente e Distribuída. (2007/2008). *Aula 2- Classes Embutidas.* ISCTE- IUL, Departamento de Ciências e Tecnologias de Informação.

*Get size of object in memory*. (s.d.). Obtido de Java Practices: http://www.javapractices.com/topic/TopicAction.do?Id=83

Guojie, J. L. (2005). *Professional Java Native Interfaces with SWT/JFace.* Wrox.

*JGraphT*. (s.d.). Obtido de JGraphT: http://jgrapht.sourceforge.net/

Laddad, R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming* (1ª edição ed.). Manning.

*Memory usage of Java objects: general guide*. (s.d.). Obtido de Javamex: http://www.javamex.com/tutorials/memory/object\_memory\_usage.shtml

Nicholas, E. (2006). *Understanding Weak References.* Obtido de java.net - The Souce for Java Technology Collaboration: http://weblogs.java.net/blog/2006/05/04/understanding-weak-references

# Glossário

**A**

***AspectJ*** – extensão à linguagem Java para permitir a programação orientada a aspectos.

**Aspecto (*aspect*)** – Mecanismo que encapsula as preocupações transversais de um sistema; constituído por pontos de corte e por conselhos.

**C**

**Conselho (*Advice*)** – parte de implementação de um aspecto, composto por um ponto de corte e por um bloco de código a ser executado no ponto de corte definido.

**D**

**Deadlock** – consiste num impasse em que duas ou mais threads estão à espera umas das outras para continuar a sua execução.

**Debug** – processo de análise de uma aplicação com o objectivo de detectar erros ou falhas.

**Diagrama de comunicação** – apresenta o fluxo de mensagens trocadas entre as instâncias de classes na execução de determinada parte de uma aplicação.

**Diagrama de sequência** – demonstra as interacções entre as instâncias de classes em determinada parte de execução de uma aplicação, bem como a sua cronologia.

**Diagrama de objectos** – apresenta as instâncias de classes criadas na execução de uma parte de uma aplicação, bem como as respectivas associações entre elas.

**E**

**Entretecimento (weaving)** – funcionalidade da programação por aspectos que consiste no acto de colocar o código relativo a aspectos nos locais definidos pelos *pointcuts*, no momento de compilação.

***Extensible Markup Language* (XML)** – linguagem de marcação de dados para descrever informação de um modo estruturado.

**M**

***Model-View-Controller* (MVC)** – padrão de desenho que visa encapsular separadamente o modelo de dados, as regras de negócio e a apresentação visual de uma aplicação.

**P**

**Padrão de desenho –** modelação de classes de forma a resolver um determinado tipo de problema de programação.

**Ponto de corte (*pointcut*)** – conjunto de pontos de junção, utilizados para entretecer os aspectos.

**Ponto de junção (*joinpoint*)** – qualquer ponto do fluxo de um programa.

**Preocupações transversais *(cross-cutting concerns)*** – requisitos que representam tarefas transversais às funcionalidades de um programa.

**T**

**Target** – modo de acesso ao objecto alvo do ponto de junção detectado, em AspectJ.

**This** – modo de acesso ao objecto em execução no ponto de junção detectado, em AspectJ.

***Tracing*** **(rastreamento)** – No contexto da programação, designa a descrição e análise do fluxo de execução de um programa.

**U**

**Unified Modeling Language (UML)** – Linguagem de modelação e representação gráfica da concepção de um sistema.

1. <http://www.hibernate.org> [↑](#footnote-ref-1)