4 - BLUEJ

4.1 INTRODUÇÃO

Designam-se por **ambientes integrados de desenvolvimento de aplicações**, um conjunto de ferramentas *software* actualmente disponíveis associadas a certas linguagens de programação, que permitem aos programadores criar as suas aplicações em tais linguagens usando um único contexto de trabalho no qual podem fazer praticamente tudo o que é necessário para a criação das mesmas, desde a gestão do projecto, a edição, compilação e teste dos vários componentes, até à concretização final da aplicação.

Estes ambientes, designados em inglês por *Integrated Development Environments*, IDE, visaram desde sempre colmatar as deficiências dos *kits* de desenvolvimento das próprias linguagens, que, em geral, não se preocupam com a integração dos diversos programas que é necessário utilizar desde o início até à conclusão de uma aplicação, designadamente, editores, compiladores, *debuggers*, eventualmente, interpretadores, programas de arquivo, de geração de documentação, etc.

JAVA sempre teve os seus respectivos *kits* de desenvolvimento (SDK) que foram sendo usados até ao aparecimento dos primeiros IDE para JAVA, que, no seu início, pelo menos auxiliavam os programadores através de editores estruturados dirigidos pela sintaxe que detectavam erros à medida que o código era introduzido.

Existem actualmente muitos e muito sofisticados IDE para JAVA, comerciais e não só, como Eclipse, NetBeans, JBuilder, IntelliJ IDEA, JDDE, SlickEdit, JCreator, BlueJ, etc., alguns com características próprias para desenvolvimento de aplicações de tipo e dimensão empresarial.

Tendo surgido a necessidade de, ao longo deste livro, apresentar de alguma forma os resultados da execução de partes de código JAVA de certos programas, ilustrar erros de compilação, apresentar a estruturação das classes de alguns projectos, etc., e não sendo necessário usar todas as capacidades de alguns sofisticados (mas "pesados") IDE para JAVA, optámos por escolher **BlueJ**, por ser "pequeno" (5MB) e muito simples de aprender e rapidamente usar.

BlueJ é um IDE para JAVA que foi desenvolvido especificamente para propósitos de ensino por uma equipa conjunta das Universidades de Monash, Melbourne, Austrália, e de Southern Denmark, Odense, Dinamarca, e cujo © é pertencente a M. Kölling e J. Rosenberg.

O ambiente BlueJ pode ser obtido livremente a partir de http://www.bluej.org, e é muito fácil de instalar em qualquer sistema operativo, apenas exigindo que já se possua instalado o que em todo caso teríamos que ter, ou seja, um ou mais JDK. Na fase de instalação o BlueJ *launcher* pesquisa automaticamente os JDK existentes, permitindo até ter diferentes ambientes BlueJ instalados sobre diferentes JDK (ex.: um para JDK1.4 e outro para JDK1.5).

Vamos neste capítulo introduzir, passo a passo, os aspectos fundamentais da criação de aplicações JAVA usando o IDE BlueJ, o que faremos tomando como exemplo de referência a classe `PMMB` desenvolvida no capítulo anterior, que, apesar de ser muito simples, nos permitirá abordar as funcionalidades mais importantes para a utilização de BlueJ em qualquer outro tipo de projecto.

Importante também será observar a metodologia da utilização do BlueJ, dado que, esta ferramenta de auxílio ao projecto pode permitir-nos criar projectos de forma incremental, ou seja, passo a passo, usando uma metodologia de prototipagem rápida de *software*.

Definida a estrutura básica, as variáveis de instância, e os construtores, cada um dos métodos de instância é codificado e introduzido através do editor na definição da respectiva classe. Esta será compilada, uma instância é

criada, tal método é executado uma ou várias vezes e a sua correcção de execução verificada. Assim, método a método, seguindo um processo iterativo, poderemos com a ajuda do BlueJ convergir mais facilmente para a solução final.

O facto de editor, compilador e interpretador estarem integrados no mesmo ambiente, juntamente com a sua interface gráfica e mecanismos de inspecção rápida dos estados internos dos objectos manipulados, facilitam em muito a adopção desta metodologia de trabalho, que vivamente se aconselha.

O ambiente BlueJ que vamos utilizar possuir para além das ferramentas indicadas, uma *Terminal Window* que simula um normal monitor no qual se escreve usando as instruções de *output*, e ainda um *debugger* para execução de um programa passo a passo.

Aconselha-se igualmente que este capítulo seja, mais do que lido, seguido passo a passo no próprio ambiente BlueJ, o que em muito facilitará a sua compreensão.

4.2 PASSOS NA CRIAÇÃO DE APLICAÇÕES

4.2.1 JANELA DE TRABALHO

A janela inicial de trabalho do BlueJ, aberta via ícone respectivo, é simples (Figura 4.1). Possui duas zonas rectangulares vazias, que irão constituir duas zonas especiais de trabalho, apresenta na barra de opções, para além do **HELP**, as quatro operações principais que estão sempre disponíveis para o utilizador, **PROJECT**, **EDIT**, **TOOLS** e **VIEW**, e ainda na sua parte lateral esquerda um conjunto de cinco botões que têm funcionalidades para realizar sobre a área para trabalho com as classes.

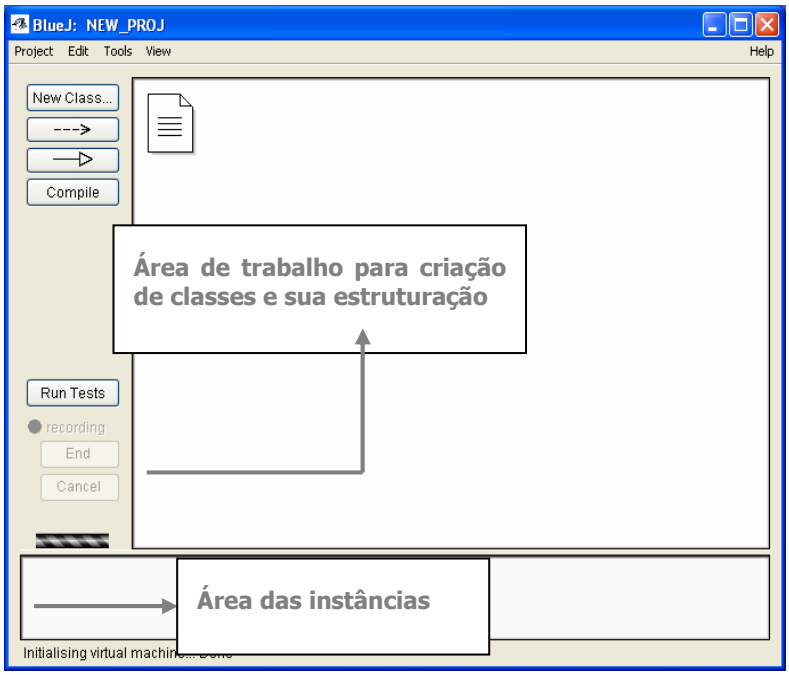

Figura 4.1 – Janela inicial

Um **projecto** BlueJ pode ser visto como uma **pasta especial** do sistema de ficheiros, onde são guardados ficheiros com código JAVA correspondentes a classes e a outras entidades da linguagem JAVA, que o BlueJ reconhece e é capaz de estruturar e representar.

As classes são apresentadas estruturadas sob a forma de um diagrama de classes, que as relaciona entre si distinguindo agregação de composição, e que é apresentado ao utilizador na área rectangular de classes acima apresentada. Cada uma das classes pode ser editada, compilada, executada para criar instâncias, etc., tudo dentro do mesmo ambiente proporcionado por todas as operações apresentadas nesta janela (Figura 4.2).

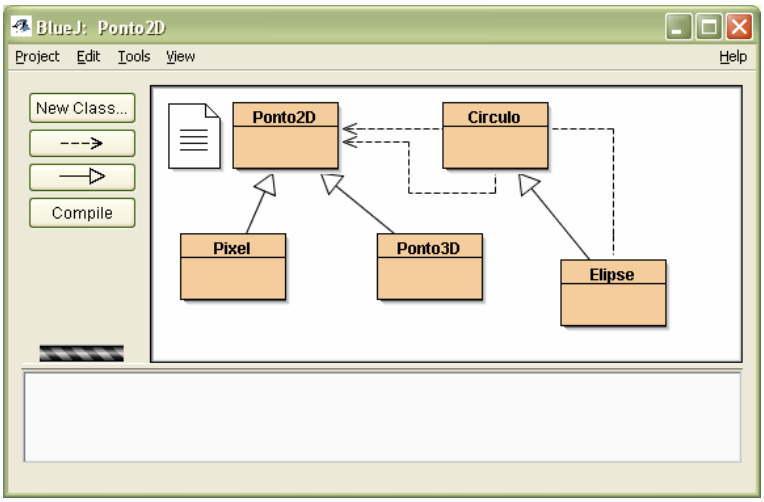

Figura 4.2 – Apresentação diagramática das classes de um projecto

A Figura 4.2 exemplifica como um diagrama de classes apresenta as várias classes que foram desenvolvidas no contexto de um dado projecto. Os seus nomes e os relacionamentos que, de forma automática, o BlueJ estabeleceu entre elas ao realizar a sua compilação.

Estudaremos em seguida cada uma das operações associadas à opção **PROJECT** (exceptuando a opção **EDIT**), usando como exemplo concreto a classe `PMMB` desenvolvida no capítulo anterior.

PROJECT

A opção **PROJECT** vai permitir-nos realizar várias acções, quer para criar um **NOVO PROJECTO** quer relativas ao projecto em curso, para além de nos oferecer algumas operações adicionais básicas (Figura 4.3)

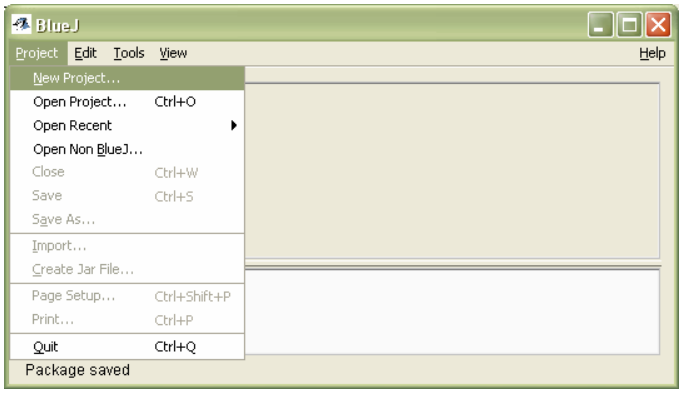

Figura 4.3 – Subopções associadas à opção PROJECT

Todas as operações apresentadas têm a sua semântica associada à noção de **projecto**, ou seja, directório onde se encontram todos os ficheiros BlueJ constituintes do projecto, que podem ser ficheiros com código fonte de JAVA*(*.java*), ficheiros já compilados *(*.class*), ficheiros BlueJ com extensão *(*.pk)* e o ficheiro de projecto *bluej.pkg* com o ícone respectivo, que, a partir desse directório, permite abrir todo o projecto fazendo dois cliques com o botão esquerdo do rato no seu ícone ( ).

A Tabela 4.1 sintetiza as acções associadas a cada uma destas operações.

PROJECT	Acção
Create Project	Cria uma nova pasta de projecto
Open Project	Abre projecto existente numa pasta
Open Recent	Selecciona um dos recentemente usados
Open Non-BlueJ	Abre directório com fontes Java

Close	Fecha projecto
Save	Fecha e guarda projecto actual
Save As	Fecha projecto actual e renomeia
Import	Importa ficheiros de directório
Create Jar File	Cria arquivo Jar do projecto

Tabela 4.1 – Acções relativas a projectos

Sendo de momento a opção mais interessante de seguir, accionemos a subopção **NEW PROJECT** para criarmos o nosso primeiro projecto BlueJ, a que vamos dar o nome de PROJ_JAVA5 (Figura 4.4).

Figura 4.4 – Abertura de um novo projecto

4.2.2 CRIAÇÃO DE UMA NOVA CLASSE

Dos quatro botões laterais (Figura 4.5), os botões com setas serão usados para estabelecerem certos relacionamentos especiais entre classes que abordaremos mais tarde, o botão **COMPILE** servirá para invocar o compilador sobre uma classe seleccionada ou todas, e o botão **NEW CLASS**, usado neste caso, acciona a abertura de uma janela que vai permitir criar uma classe nova de um dos vários tipos apresentados, e que, no nosso caso, escolhemos como sendo uma classe "standard" (as que até agora estudamos) a que demos o nome de `PMMB` (Figura 4.5).

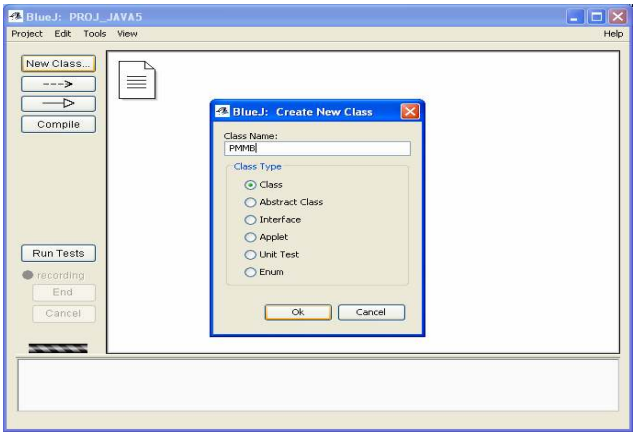

Figura 4.5 – Criação de nova classe

Como se pode verificar, existem vários outros tipos de entidades JAVA que podem ser criadas em BlueJ, para além das classes definidas pelo utlizador, que serão por nós estudadas mais tarde em capítulos próprios.

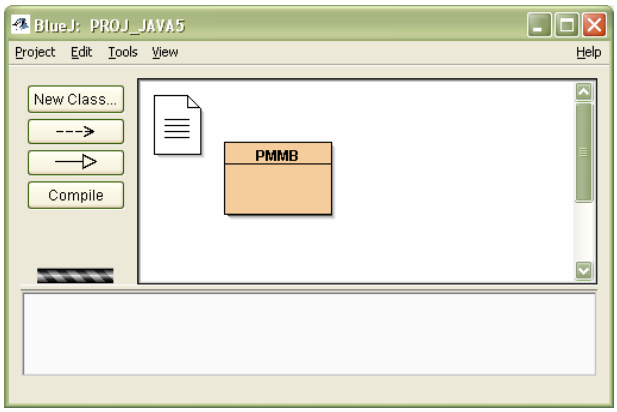

Figura 4.6 – Classe na área de projecto

Um pequeno rectângulo a amarelo com um identificador dentro será o símbolo representativo de uma classe no ambiente BlueJ (Figura 4.6).

Fazendo ❶❶ sobre o símbolo com o nome da classe, abre-se automaticamente a janela do editor de texto, contendo, para além de um *template* por omissão que serve de guião para que o programador insira a sua descrição da classe, autoria, versão e outros dados, um conjunto de botões auxiliares de edição, normais num editor de texto, como **UNDO**, **CUT**, **COPY**, **PASTE**, **FIND**, **FIND NEXT** e **CLOSE**, com os significados óbvios. A Figura 4.7 mostra a janela de edição e respectivos botões.

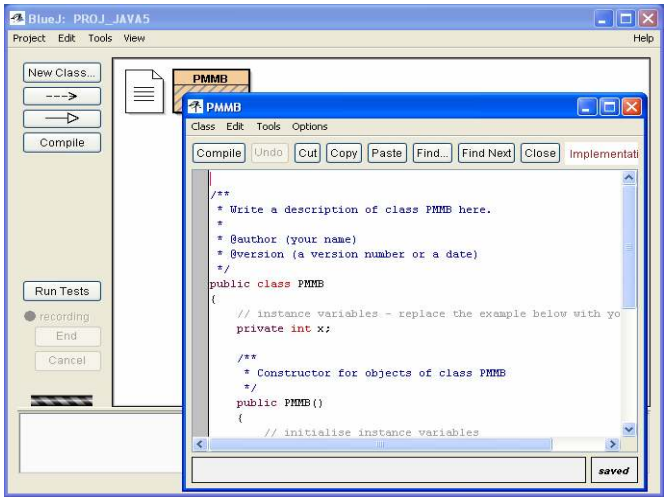

Figura 4.7 – *Template* para edição de uma classe

Não sendo um editor estruturado, este apresenta no entanto, a cores diferentes, os vários elementos constitutivos da sintaxe da linguagem para melhor distinção dos mesmos. Tipos de dados a vermelho vivo, modificadores a vermelho rubi, comentários que vão ser usados na documentação final a azul (cf. /** .. */), comentários normais a cinzento, certas palavras reservadas como `this` a azul claro, *strings* a verde, etc.

Os comentários a azul, delimitados pelos símbolos /** .. */, podem ser comentários que se estendem por mais de uma linha, e são muito importantes pois irão fazer parte da documentação final do projecto, que pode ser, a pedido do utilizador, automaticamente gerada pelo BlueJ. Como veremos adiante, são geradas páginas HTML contendo todas as classes do projecto, e para cada uma destas a descrição completa das suas variáveis e métodos, documentados com os comentários que o programador introduziu no código.

Assim, em geral, é boa prática inscrever um destes comentários antes da definição de cada classe, de cada variável e de cada método programado. Para além das vantagens em termos de documentação, veremos que, durante a execução dos métodos em BlueJ, estes comentários aparecem ao utilizador para o auxiliar na introdução dos valores dos respectivos parâmetros.

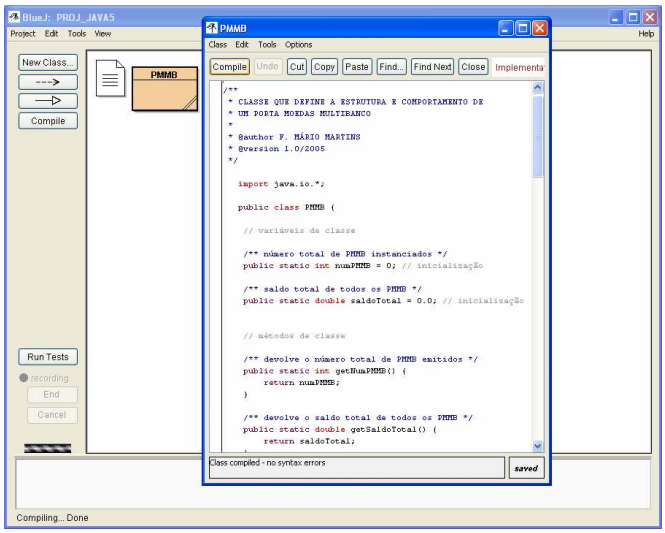

Figura 4.8 – Texto introduzido e compilado

A qualquer momento do processo de edição pode fazer-se **Save** a partir do menu **Class**. Accionado o botão **Compile** a classe é compilada (Figura 4.8).

4.2.3 ERROS DE COMPILAÇÃO

Um erro de compilação é indicado a cor na linha (ou zona) onde é detectado (Figura 4.9).

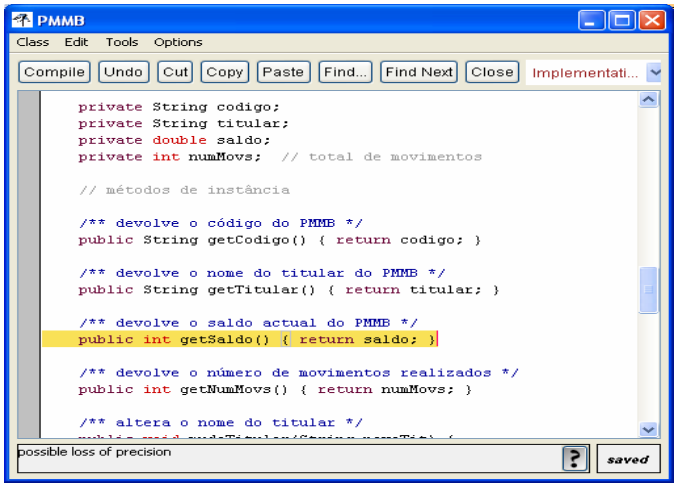

Fig 4.9 – Erro de compilação 1

Caso o erro não seja óbvio, poder-se-á solicitar alguma ajuda sobre o mesmo usando o botão **?** (Figura 4.10). Corrigido o erro e recompilado o programa, a mensagem de compilação com sucesso é apresentada na área de mensagens da janela (Figura 4.11).

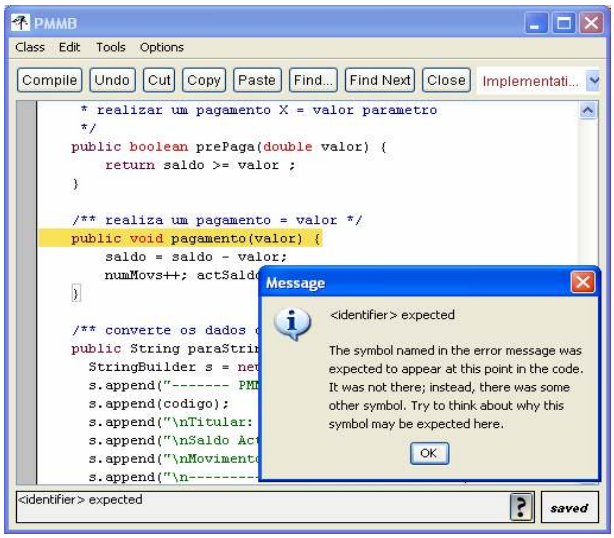

Fig 4.10 – Erro de compilação 2 e ajuda

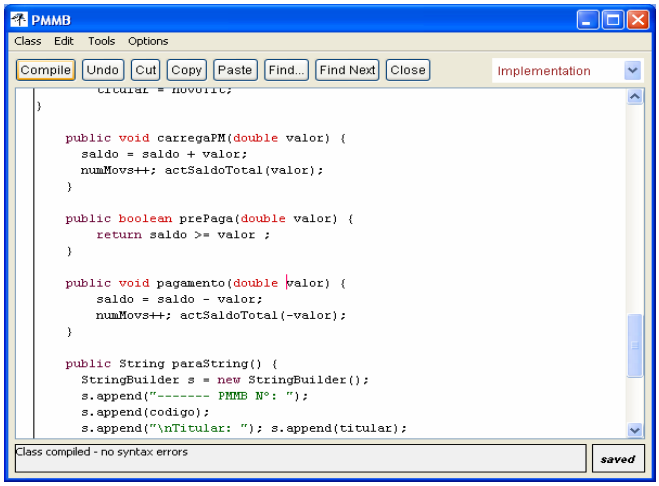

Fig 4.11 – Erro de compilação 2 corrigido

Existem erros de compilação que são muito comuns e frequentes em fases iniciais de programação. O erro **"**incompatible types**"** é muito comum pois pode acontecer em múltiplas situações (parâmetros e atribuições). A Figura 4.12 ilustra uma destas situações.

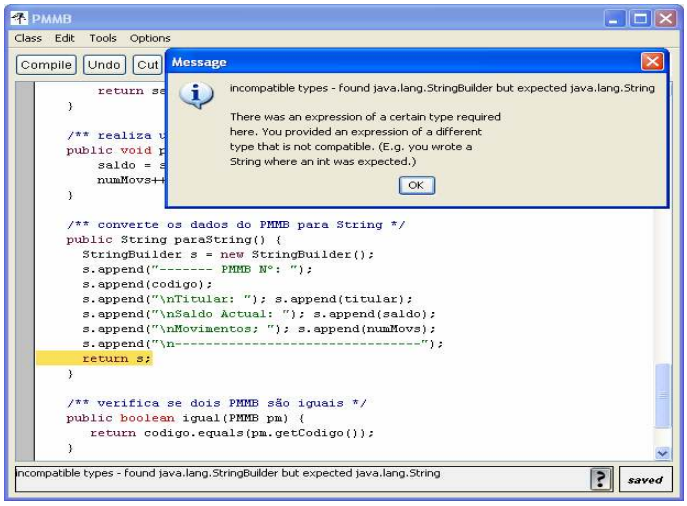

Fig 4.12 – Erro de compilação 3: Tipo de resultado errado

O erro anterior torna-se fácil de corrigir pois resultou de um erro na instrução `return` de um método. O compilador indica que estaria à espera de um valor de tipo `String` e que encontrou um valor do tipo `StringBuilder`. É um erro comum: esquecêmo-nos de converter a `StringBuilder` em `String` usando `toString()`.

Imediatamente editamos a linha onde o erro ocorreu, e recompilamos a classe, agora com sucesso tal como indicado pelo compilador (Figura 4.13).

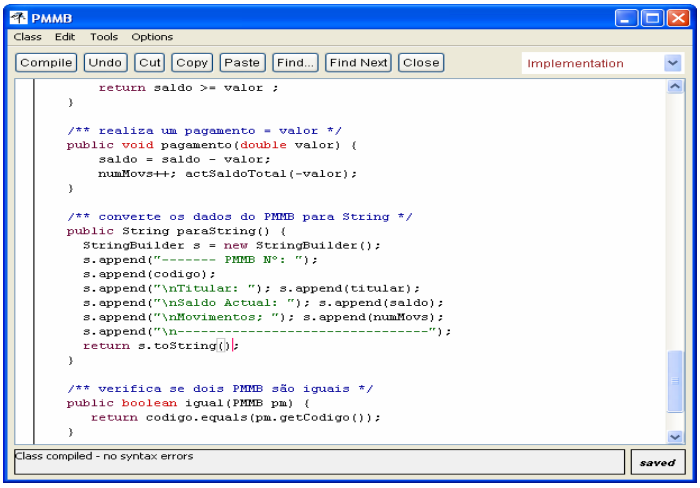

Fig 4.13 – Erro de compilação corrigido

4.2.4 CRIAÇÃO DE INSTÂNCIAS

A partir do momento em que uma classe tenha definidos construtores, torna-se possível de imediato criar instâncias suas. Porém, se ainda não tivermos definidos quaisquer métodos de instância, não poderemos enviar mensagens às instâncias, pelo que estas de nada servem.

Em BlueJ, é possível criar uma classe método a método, criando instâncias só com os métodos actualmente definidos e testar tais métodos, voltar a editar a classe, juntar mais métodos de instância e assim sucessivamente. Não o fizemos neste exemplo, no qual criámos de uma vez só a classe `PMMB`, porque precisamente o fizemos no capítulo anterior. Mas esta metodologia de utilização do BlueJ pode tornar-se muito útil dado que, em conjunto naturalmente com manuais, papel e lápis (e borracha!), **o BlueJ serve de verdadeira ferramenta de prototipagem**, pois permite-nos o desenvolvimento e teste incremental, no sentido de que, método a método, classe a classe, podemos ir desenvolvendo o projecto, codificando e testando peça a peça.

Antes de criarmos algumas instâncias, vamos verificar junto da classe `PMMB` quais os métodos que esta nos disponibiliza. Ainda que a nossa atenção esteja completamente focada na criação das instâncias, não nos podemos esquecer que as classes também podem ter a sua estrutura e comportamento, como é até o caso da que temos por exemplo.

Ao fazermos ◗ sobre o símbolo da classe `PMMB` aparecerá em ecrã a lista de todos os **métodos públicos** (a preto) e **operações** (a vermelho) disponíveis sobre tal classe, tal como se mostra na Figura 4.14.

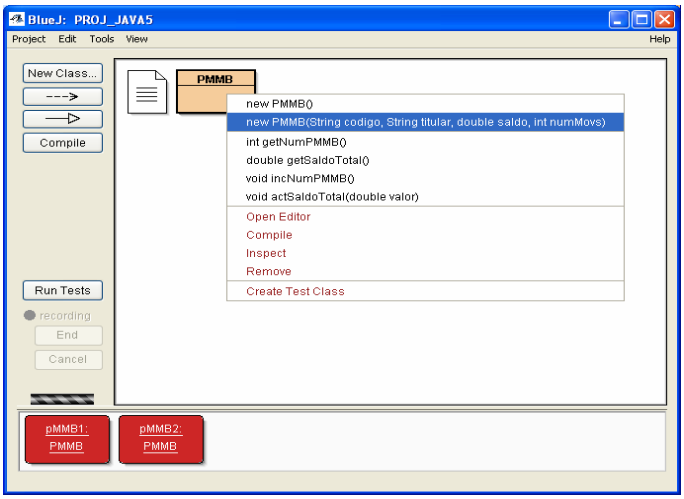

Figura 4.14 – Operações sobre a classe e duas instâncias

Na Figura 4.13, mostra-se já um estado em que verificamos ter disponíveis seis operações que foram programadas para a classe `PMMB`, duas das quais são **construtores** (as que começam por `new`). As quatro outras operações, estando associadas à classe `PMMB`, são os métodos de classe, que aparecem na API da classe porque os declarámos como sendo `public`. Porventura agora se possa melhor compreender a importância de declarar ou não um método `public`. Neste caso, tal corresponde a deixar que alguém, via BlueJ, possa invocar o método de classe `incNumPMMB()`, deixando-o aumentar o número de `PMMB`, sem que tal corresponda à criação efectiva de nenhuma instância. Tal não abona em favor da coerência de dados do sistema, mas este é o resultado de se ter decidido que tais métodos seriam `public`.

Todas as outras operações a vermelho são operações que o ambiente BlueJ disponibiliza para serem executadas sobre uma qualquer classe, entre elas **INSPECT** que permite observar os valores das suas variáveis de classe, tal como apresentadas em janela própria (como se pode observar na Figura 4.15).

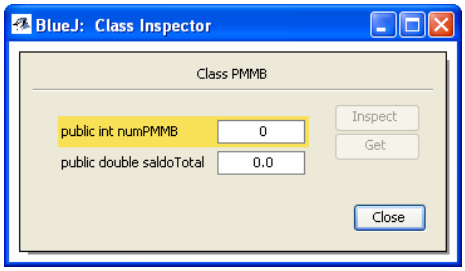

Figura 4.15 – Variáveis de classe de `PMMB`

Vamos agora invocar o construtor completo (Figura 4.14) e ver como poderíamos criar cada uma das instâncias que na mesma figura aparecem já no espaço destinado às várias instâncias criadas pelas possíveis diferentes classes do projecto.

A invocação do construtor conduziria à abertura das janelas de diálogo apresentadas a seguir para introdução dos respectivos dados. Repare-se que comentários, assinaturas, tipos de dados e nomes de parâmetros, são informações passadas do código ao utilizador.

Repare-se ainda que BlueJ sugere identificadores para as instâncias, iniciados por letras minúsculas, mas que o utilizador poderá substituir no momento da criação de cada uma das instâncias (não depois).

Cada instância tem sempre a si associado o respectivo tipo, isto é, a classe a que pertence (ver Figuras 4.16 e 4.17). Em todas as operações de introdução de dados, o BlueJ fornece informação sobre os nomes e os tipos de cada um dos valores a serem lidos, e, fornece ainda informação sobre a sintaxe da operação que está a ser realizada (assinatura dos métodos, por exemplo).

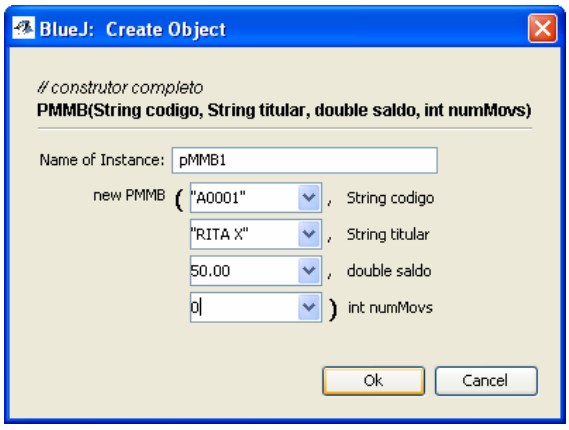

Figura 4.16 – Diálogo para introdução dos valores parâmetro

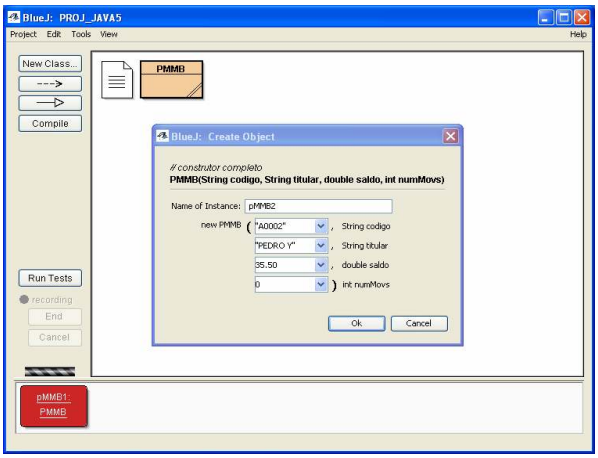

Figura 4.17 – Criação de outra instância

Note-se, nestes exemplos, que cada campo da *form* de introdução de dados apresenta o respectivo identificador e tipo, tal como no nosso código os escrevemos. Os dados devem ser introduzidos usando o seu formato normal tal como o compilador de JAVA os aceitaria. Dados errados são validados pelo BlueJ, que envia mensagens de erro que são apresentadas na própria *form* de introdução de dados.

Se, num dado método, necessitarmos de um parâmetro que não seja de tipo primitivo mas uma instância de uma dada classe, podemos introduzi-lo em BlueJ tal como tal instância seria criada num programa normal. Por exemplo, se um método necessita de um parâmetro de tipo `Ponto`, podemos escrever textualmente `new Ponto(-1, 3)` no respectivo campo de introdução de dados.

4.2.5 INTERACÇÃO COM AS INSTÂNCIAS

As instâncias criadas, são colocadas na parte inferior da janela de projecto, e cada uma possui um identificador (como se fosse o nome da variável que a referencia no programa) e o nome da classe a que pertence (Figura 4.18).

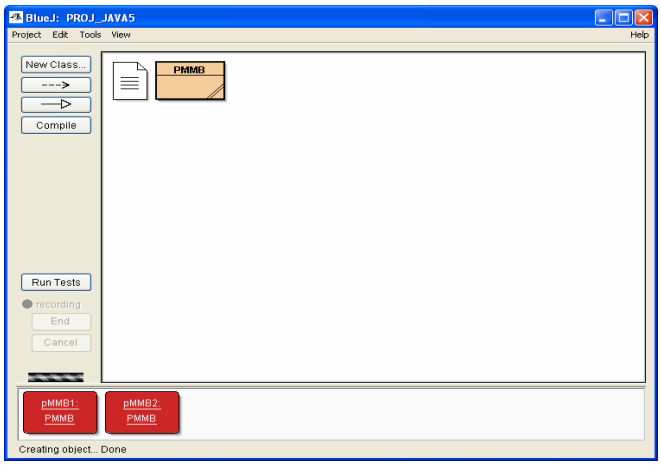

Figura 4.18 – Classe e Duas Instâncias a Testar

Fazendo ◗ sobre uma qualquer instância aparece-nos a lista dos métodos de instância que podemos invocar e respectivas assinaturas (Figura 4.19).

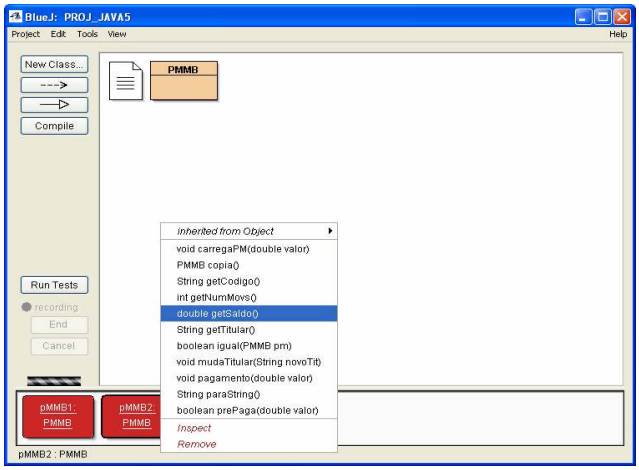

Figura 4.19 – Métodos de Instância disponíveis

Como se pode ver pela Figura 4.19, seleccionámos o método `getSaldo()` e o resultado da invocação deste método será o apresentado na Figura 4.20 (onde se apresenta apenas a "janela resultado"). Note-se a apresentação, na janela, do texto correspondente aos comentários de documentação que incluímos no código (usando /** ... */).

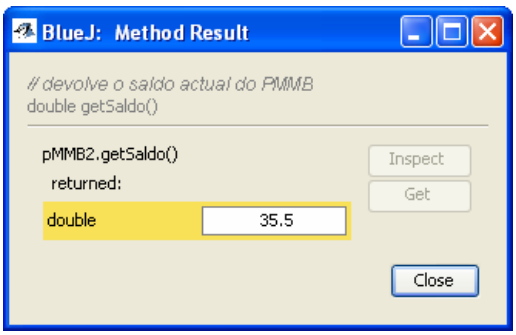

Figura 4.20 – Janela com resultado do método

A operação **INSPECT** pode ser invocada a partir da lista de operações ou fazendo ◖ sobre a instância, abrindo-se uma janela com os valores das variáveis de instância, tal como se mostra na Figura 4.21.

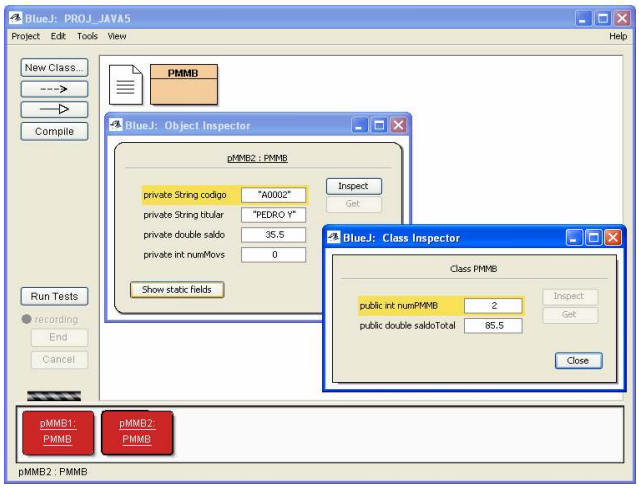

Figura 4.21 – Resultado de INSPECT sobre uma instância

Repare-se que a Figura 4.21 é o resultado de se realizar **INSPECT** sobre a instância, o que produz uma janela que contém as suas variáveis de instância e seus valores, e de termos também accionado o botão **SHOW STATIC FIELDS** (variáveis estáticas ou de classe) na janela resultado. Estas duas acções deram origem à janela onde são mostrados em conjunto os valores das variáveis da classe `PMMB` e da instância `pMMB2`.

A operação **REMOVE** elimina a instância sobre a qual foi accionada.

Vamos agora realizar uma operação que altera o estado interno da instância `pMMB2`, aumentando o seu saldo, e em seguida verificar se tudo funciona bem, ou seja, se o seu saldo é actualizado e se o saldo total também. Apresentando apenas as janelas de diálogo e de resultados teríamos a sequência apresentada nas Figuras 4.22 a 4.24.

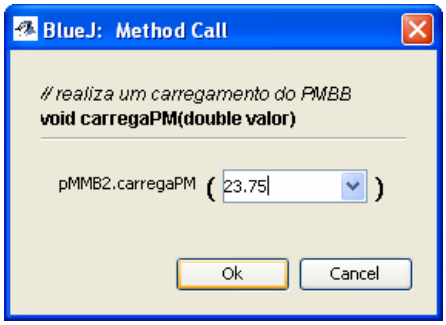

Figura 4.22 – Operação de modificação do estado

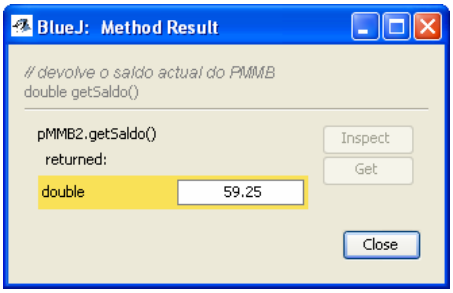

Figura 4.23 – Operação de consulta do estado

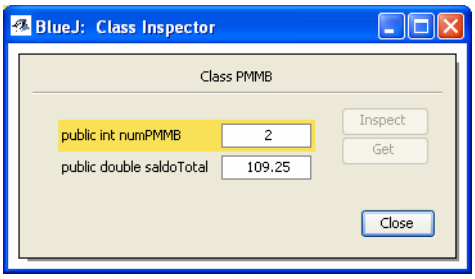

Figura 4.24 – Resultado de Inspect sobre a classe

Analisando o valor apresentado na Fig. 4.24, pode assim comprovar-se que a alteração foi feita correctamente (saldo anterior de 85.5 + 23.75 = 109.25 de saldo total). São operações deste tipo apoiadas na opção **INSPECT**, que justificam a utilização de um IDE como BlueJ, não só para criação das aplicações mas, sobretudo, para verificação.

Algumas operações necessitam de parâmetros que são eles próprios instâncias, por vezes complexas, de certas classes, quer nossas quer predefinidas de JAVA. Como criar tais parâmetros e atribuí-los como parâmetros aos métodos que deles necessitam?

Vamos começar por um exemplo que visa comparar dois `PMMB`. Ao invocarmos o método `igual()` sobre, por exemplo, a instância `pMMB2`, surge a janela de diálogo da Figura 4.25, onde é pedido um parâmetro do tipo `PMMB`.

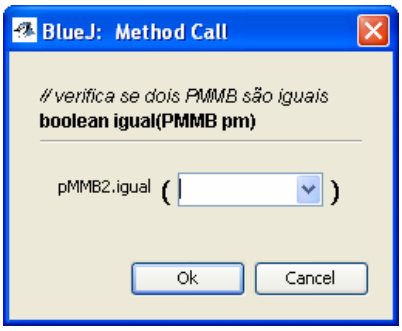

Figura 4.25 – Diálogo para introdução de um parâmetro objecto

Ora, ou comparamos `pMMB2` com ele próprio, ou o comparamos com `pMMB1`, pois não temos mais instâncias de `PMMB` criadas, ou então criamos um outro `PMMB` apenas para comparação (tipo objecto temporário). Para usar uma instância que já existe, temos duas formas: introduzir textualmente o seu nome ou fazer ◖◖ sobre a instância que queremos para parâmetro. Se pretendêssemos um novo objecto, usávamos o construtor, criávamos o objecto, tal como atrás, usávamo-lo como parâmetro e no fim removíamo-lo. Em qualquer dos casos, o resultado seria o esperado, os `PMMB` são diferentes, pois têm códigos diferentes (Figura 4.26).

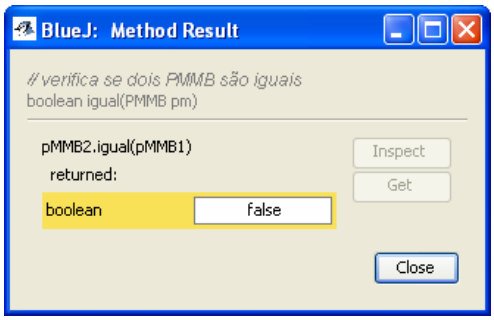

Figura 4.26 – Resultado da comparação entre dois PMMB

Note-se que esta operação de comparação de cartões poderia ter que ser melhor pensada caso pretendêssemos comparar um `PMMB` existente com um `PMMB` fictício, criado apenas para efeito de teste, em cujo caso não deveria

afectar a variável de classe que conta o número total de emitidos. Precisaríamos, se assim fosse pedido, de um novo construtor para satisfazer casos destes que não incrementasse `numPMMB` na classe.

Vamos agora verificar se a cópia de instâncias de `PMMB` está bem implementada. A Figura 4.27 dá a entender que já invocámos o método cópia sobre `pMMB1`, e temos já um resultado.

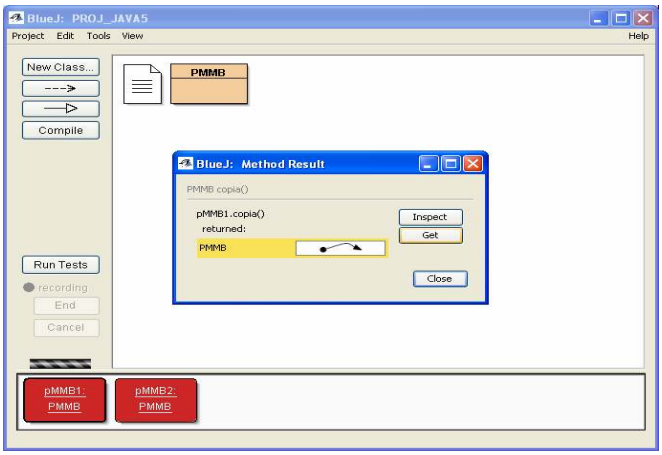

Figura 4.27 – Cópia de um `PMMB`

A **seta** indica que é de um **tipo referenciado**, ou seja, que o que foi criado como resultado é um **apontador para um PMMB**. Agora é a metáfora do "papagaio voador": ou larga e ele voa (**Close**) ou dá a mão (**Get**) e tem que dar um nome. Vamos "adquirir" o objecto criado usando o botão **GET** e dando-lhe o nome **`copiaDe1`**.

Poderíamos usar os métodos consultores um a um para verificar se a cópia foi de facto realizada, mas o **INSPECT** faz isso de uma só vez (Figuras 4.28 e 4.29).

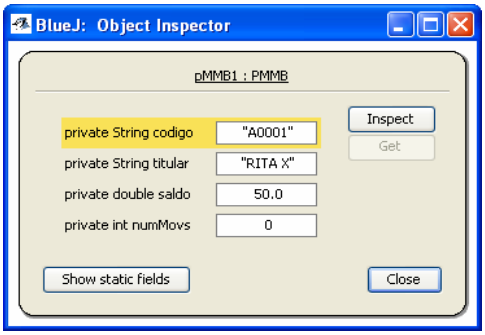

Figura 4.28 – `pMMB1` original

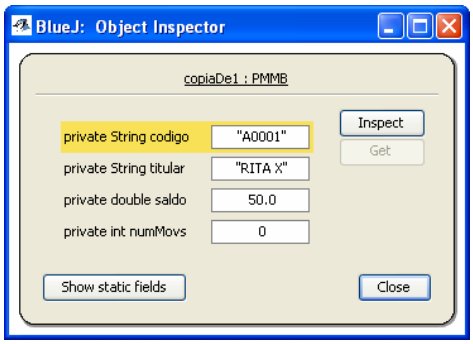

Figura 4.29 – Cópia perfeita; objectos distintos

Para testarmos se são ou não distintos vamos usar um deles para pagar 35.50 euros numa compra e voltar a inspeccionar as suas variáveis de instância. Não apresentando aqui a sequência de operações que foi já anteriormente realizada mas apenas os seus novos estados internos, qual foi o `PMMB` usado para realizar tal pagamento?

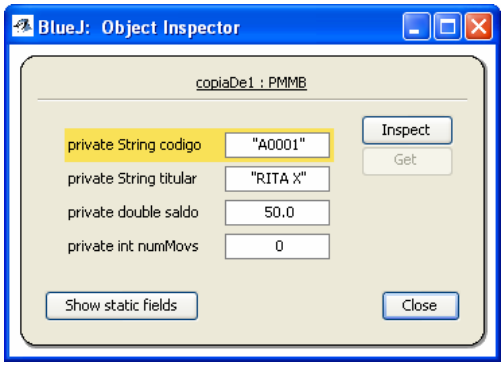

Figura 4.30 – Objecto copiado

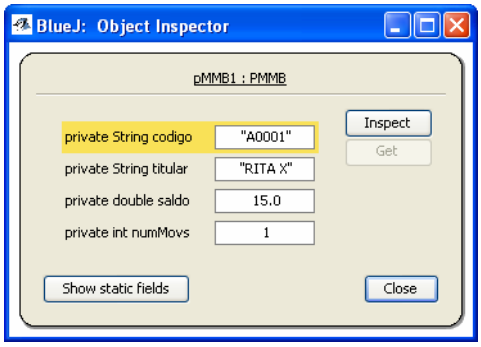

Figura 4.31 – Objecto inicial – saldo alterado

A compra, como se pode ver, foi realizada sobre `pMMB1` que viu o seu saldo alterado em 35 Euros. Prova-se também, mais uma vez, que os objectos não são partilhados pois `pMMB2` não se alterou após a alteração de estado de `pMMB1`.

4.3 TOOLS

A opção **TOOLS** da barra principal de opções oferece-nos um conjunto de operações relativas às classes que constituem o projecto actual sintetizadas na tabela seguinte.

TOOLS	Acção
Compile	Compila todas as classes do projecto
Compile Selected	Compila a classe seleccionada
Rebuild Package	Reconstrói a estrutura de classes
Use Library Class	Abre biblioteca de classes JAVA
Project Documentation	Cria ou regenera a documentação do projecto em HTML
Testing ▶	Permite criar unidades de teste
Preferences	Definição de preferências do utilizador

Tabela 4.2 – Acções relativas a TOOLS

Por exemplo, há a necessidade de criarmos instâncias de classes predefinidas de JAVA a partir dos seus construtores e métodos de inserção, ou apenas relembrar métodos de instância definidos para as mesmas. A opção

USE LIBRARY CLASS permite-nos ter acesso à biblioteca de classes predefinidas de JAVA, tal como se ilustra a seguir, onde vamos exemplificar a criação de uma `String` para modificar o nome do titular.

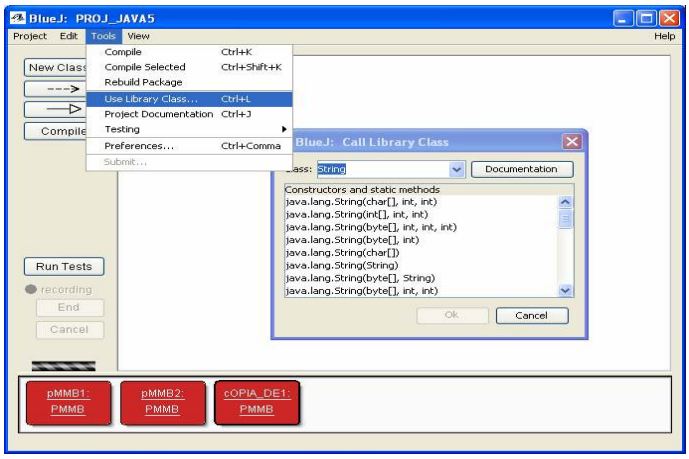

Figura 4.32 – Consulta da classe `String`

Para tal, vamos aceder à classe predefinida `String`, e criar uma instância com o novo nome do titular usando um construtor apropriado. A instância criada vai para o espaço de trabalho de instâncias.

Dos vários construtores de `String` (ou de qualquer outra possível classe), seleccionamos o que mais nos convém, introduzimos os parâmetros, carregamos no botão de **OK** e temos uma instância criada preparada para servir de argumento para o método que se pretenda (Figuras 4.33 e 4.34).

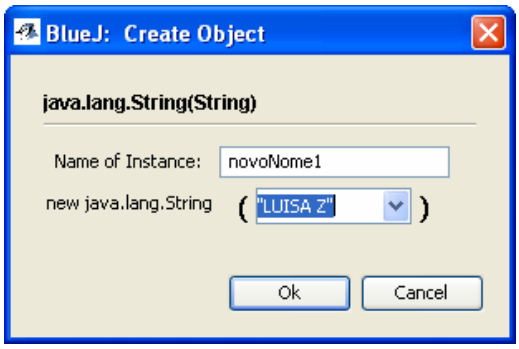

Figura 4.33 – Diálogo para criação de uma instância de `String`

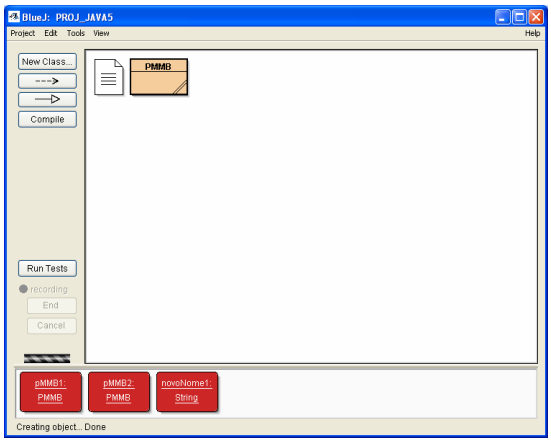

Figura 4.34 – Instância de `String` criada

Este exemplo dado com a classe `String` não faz muito sentido já que uma `String` poderia ser escrita directamente como parâmetro, quer usando um construtor, como em `new String("abc")`, quer de forma textual, apenas como `"abc"`. O importante, porém, é perceber-se que, qualquer que fosse a classe complexa da qual pretendêssemos criar instâncias, os passos seriam os mesmos, e as bibliotecas estão disponíveis.

Usando de novo **INSPECT** podemos confirmar a efectiva alteração do nome do titular na instância `pMMB2`, tal como se mostra na Figura 4.35.

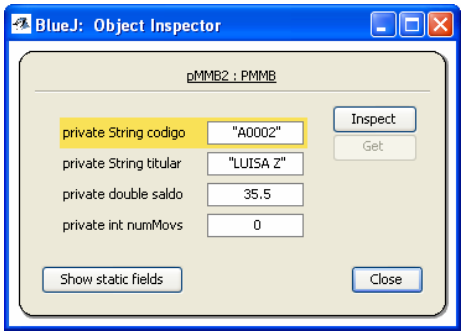

Figura 4.35 – Confirmação de mudança de titular em `pMMB2`

Uma forma de se verificar que as `Strings` são objectos de facto diferentes em JAVA, é que enquanto em qualquer variável que referencia um objecto aparece uma seta, indicativa de que a variável contém uma referência ou apontador, quando se trata de uma *string* aparece explicitamente o seu valor na forma "….".

A opção de **TOOLS** de nome **COMPILE SELECTED**, permite que em projectos complexos, envolvendo muitas classes, se possa seleccionar uma dada classe (por exemplo uma das classes que importámos de outro projecto), e se faça a sua compilação imediata sem ter que abrir o editor de texto..

Se nesta compilação realizada em modo *batch*, a classe apresentar erros de compilação, o erro é imediatamente assinalado e o modo de projecto comutado automaticamente para o modo de edição.

A opção **REBUILD PACKAGE** é também usada quando em projectos contendo elevado número de classes, e em especial quando acabámos de realizar uma importação de novas classes, há a necessidade de recompilar todas as classes e, mais do que isso, reestabelecer os relacionamentos entre elas. Em geral, esta opção é usada num projecto quando pretendemos que o diagrama de classes, contendo as várias setas que representam relações entre as classes, seja actualizado.

A operação **PRINT** da opção **PROJECT**, permite imprimir várias componentes do projecto, como código fonte, ficheiro *read.me* e o **diagrama de classes**.

A notação usada em BlueJ para o diagrama de classes, obedece a uma notação *standard* da área da modelação de *software* "orientada aos objectos" designada UML (*Unified Modelling Language*), a que nos voltaremos a referir adiante.

A opção **PROJECT DOCUMENTATION** permite-nos obter de forma automática a documentação final do nosso projecto em formato HTML, com estrutura semelhante à descrição das próprias classes predefinidas de JAVA (Figura 4.36).

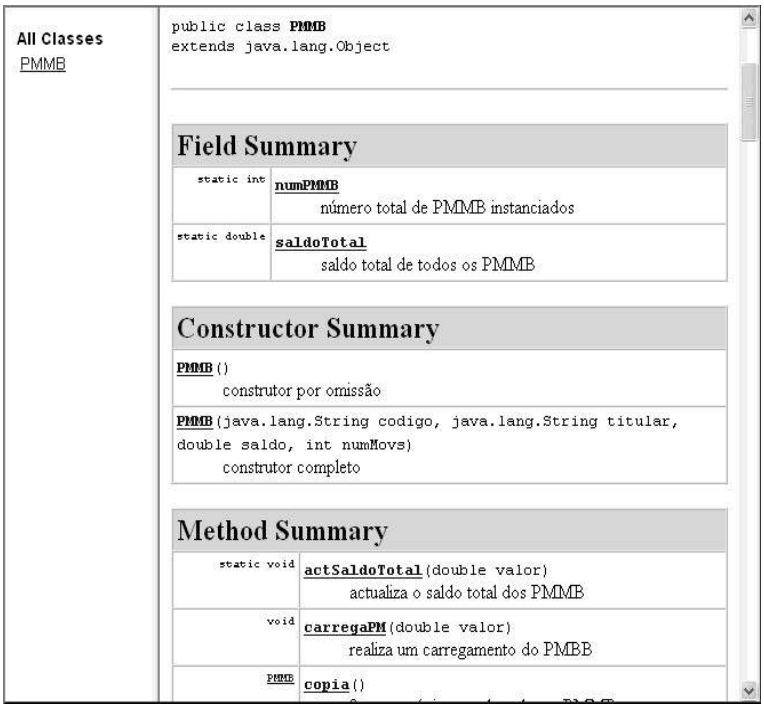

Figura 4.36 – Documentação do projecto em HTML

Depois de termos todo o projecto concluído e documentado, podemos criar uma pequena classe de teste, a classe `TstPMMB`, que contém o método `main()` no qual apenas se cria uma instância de um `PMMB` que se devolve como resultado. Este resultado será devolvido ao ambiente BlueJ, e esta instância será de imediato colocada no ambiente de trabalho, na zona das instâncias, para que se possa testar o seu código.

Neste caso trata-se de um pequeno projecto, mas, em grandes projectos, como veremos, esta possibilidade será de grande utilidade pois evitará todos os pequenos passos anteriores (que foram úteis para estudarmos o BlueJ) de criação das instâncias passo a passo e uma a uma.

Por outro lado, e como vamos ver em seguida, poderemos deste modo criar aplicações que podem ser arquivadas e enviadas a terceiros em estado executável e prontas a serem intensamente testadas. Note-se que a classe `PMMB` não é executável mas `TstPMMB` é dado conter um método `main()`.

Estas nossas ***classes de teste*** não devem ser confundidas com as *Test Units* e *Test Class* de BlueJ que não serão sequer por nós utilizadas. A classe `TstPMMB` é apresentada na Figura 4.37).

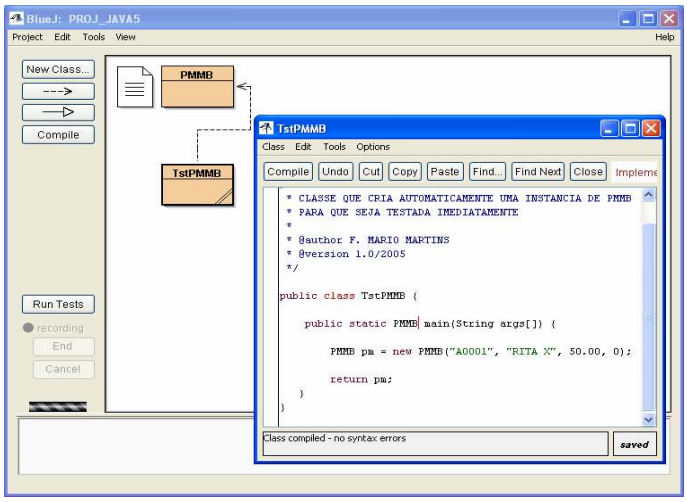

Figura 4.37 – Criação da classe de teste de `PMMB`

Note-se nesta figura, o aparecimento pela primeira vez de **uma seta a tracejado** a ligar as classes `TstPMMB` e `PMMB`. Trata-se da primeira semântica de relacionamento entre classes que é apresentada no **diagrama de classes**, e que significa que a classe `TstPMMB` **usa** a classe `PMMB`, ou seja, está dependente dela.

O *diagrama de classe* do BlueJ segue o mais de perto possível a notação da linguagem de modelação por objectos designada UML (*Unified Modelling Language*), notação que é hoje bastante utilizada em projectos de *software* e não só.

O significado de **usa** é evidente já que, para definirmos o código da classe `TstPMMB`, tal como se pode ver até pela Figura 4.37, necessitamos de criar variáveis do tipo `PMMB` e, consequentemente, da classe `PMMB`.

A seta a cheio representa um outro tipo de relacionamento entre classes que ainda não estudamos e que as classes exemplo não têm. Trata-se do relacionamento em que uma classe é definida como sendo subclasse de uma outra, relacionamento que implica que a subclasse inclui a classe superior na sua definição.

Este relacionamento hierárquico entre classes será por nós estudado no capítulo seguinte, elo que, a partir de então será melhor entendido. No entanto, e como exemplo da forma como tal relação é representada no diagrama de classes, na Figura 4.2 é apresentado um exemplo em que a classe `Ponto3D` é subclasse da classe `Ponto2D`.

Voltando à classe de teste `TstPMMB`, vamos executar o seu método `main()` que vai devolver uma instância de `PMMB` (Figuras 4.38 e 4.39). O método estático `main()` devolve um `PMMB`, foi seleccionado e vai ser executado.

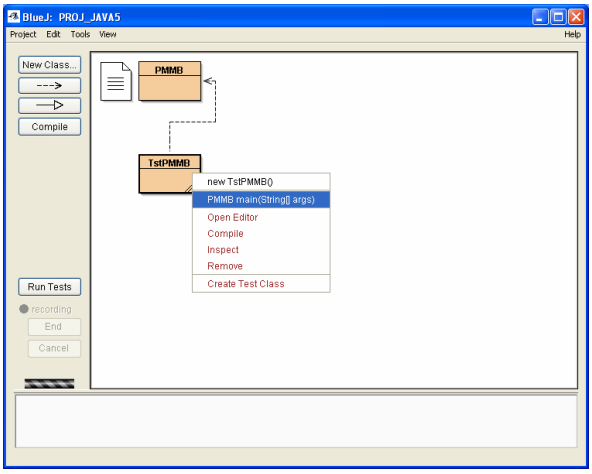

Figura 4.38 – Execução do método main() em TstPMMB

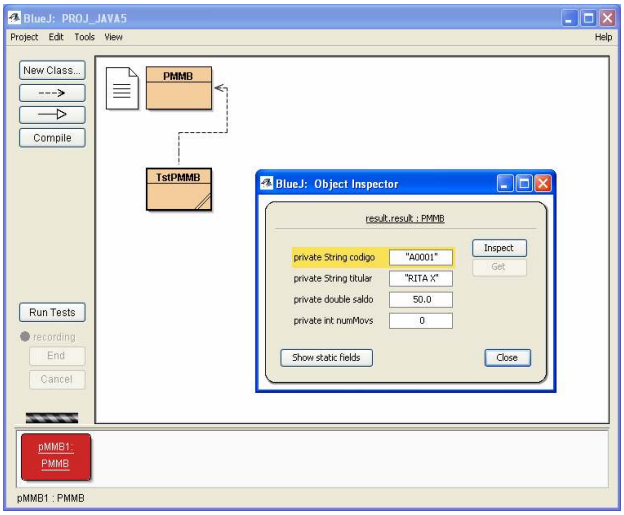

Figura 4.39 – Execução do método `main()` em `TstPMMB`

Antes de realizarmos a última e importante operação relativamente ao nosso primeiro projecto, vamos ver a derradeira operação sobre classes que há para analisar, a operação **REMOVE** (Figura 4.40).

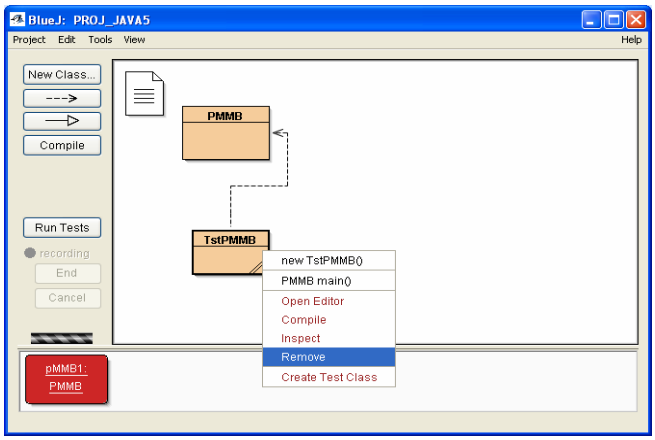

Figura 4.40 – Remoção de uma classe

Fazendo ◗ sobre o símbolo da classe respectiva, surge a lista de operações. Seleccionando a operação **REMOVE,** a classe é removida permanentemente. Note-se que a classe é removida não apenas do diagrama de classes do projecto mas também do directório, pelo que esta operação pode tornar-se delicada.

Assim, no menu de preferências do utilizador, **TOOLS ► PREFERENCES,** é conveniente configurar as operações de **Save** para que possam criar *backups*, evitando situações delicadas de perda do código de classes.

Finalmente, vamos criar um ficheiro de arquivo JAVA (*JAVA Archive File – jar*) do projecto, a partir do menu **PROJECT ► CREATE JAR FILE**, ao qual podemos dar o nome de *PMMB.jar* e que, neste caso, será um executável a partir do método `main()` da classe `TstPMMB` (Figura 4.41).

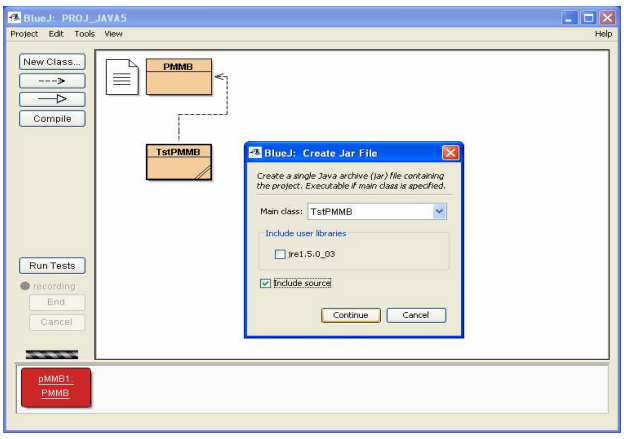

Figura 4.41 – Criação do ficheiro PMMMB.jar

Assim completámos o nosso projecto PMMB agora em ambiente BlueJ.

No directório BlueJ associado a um projecto, existem diversos ficheiros (ou pastas até) com diferentes extensões com significados que se torna importante clarificar. Vão existir ficheiros com código fonte JAVA, de extensão *.java*, classes já compiladas de extensão *.class*, ficheiros do Bluej de extensão *.pkh*, eventualmente uma pasta *doc* contendo a documentação HTML do projecto e um ficheiro **bluej.pkg** executável.

Na Figura 4.42 apresenta-se o directório PROJ_JAVA5 correspondente ao projecto da classe `PMMB` que acabámos de desenvolver, para que, em concreto, se possam ver os vários tipos de ficheiros que foram criados (incluindo o arquivo executável).

Figura 4.42 – Directório final do projecto

A partir desta pasta de projecto, fazendo ❹❹ sobre o ícone correspondente ao ficheiro seleccionado na figura (*bluej.pkg*), o projecto é de imediato reaberto para trabalho.

O ficheiro README corresponde ao ícone da **folha em branco** que permanentemente aparece na área de trabalho de projecto, e que pode ser preenchido com indicações adicionais, por exemplo, sobre os autores, de versão, de requisitos de instalação ou outras.

4.4 VIEW

Para terminarmos a apresentação do BlueJ falta apenas referirmo-nos à opção **VIEW** que permite ao utilizador introduzir, temporária ou permanentemente, no ambiente de trabalho, janelas adicionais dedicadas à visualização de operações ou tarefas específicas, correspondentes a funcionalidades adicionais oferecidas pelo ambiente BlueJ.

Por exemplo, a Figura 4.43 mostra a janela designada *Terminal Window* que simula o monitor, e para onde é direccionado o normal *output*, como `out.println("abc")`, bem como mensagens de erro, etc.

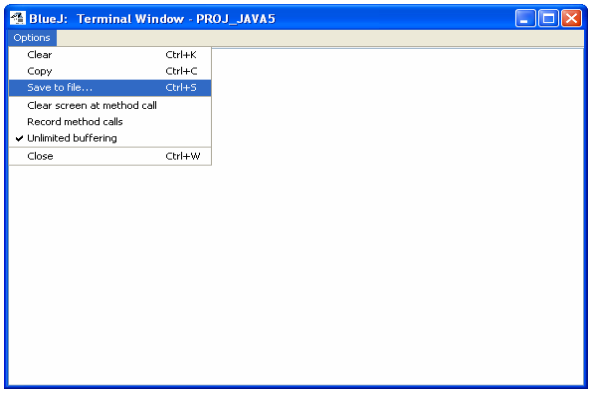

Figura 4.43 – *Terminal Window*

O BlueJ possui também incorporada uma ferramenta de *debugging*. O programador pode introduzir *breakpoints* no código fonte e realizar a execução do código passo a passo. A janela apresentada na Figura 4.44 permite analisar os valores que certas variáveis vão assumindo e controlar a execução do código.

Figura 4.44 – *Debugger Window*