Desenvolvimento, Prototipagem e Teste de Projectos Em Java5 e BlueJ

F. Mário Martins

DI/UM

2006/2007

PROJECTO 1

APLICAÇÃO DE REGISTO E CONSULTA DE INFORMAÇÃO SOBRE UMA TURMA DE ALUNOS (VERSÃO 1)

REQUISITOS

Pretende-se desenvolver uma aplicação que permita realizar um conjunto de operações básicas sobre as informações registadas acerca dos alunos que frequentam uma dada **Turma** (aqui assumida como uma Disciplina), ou seja, sobre todos os alunos inscritos em tal Turma.

Sobre cada aluno deve possuir-se a seguinte informação: número, nome, curso, ano do curso, média actual e lista com os códigos das disciplinas a que está inscrito.

Designando esta informação referente a cada aluno por **FichaAluno**, pretende-se que, para além dos construtores, das usuais operações de consulta e modificação, e ainda de `toString()` e `clone()`, sejam disponibilizadas também as operações seguintes:

- *Operação de modificação do nome do aluno;*
- *Operação de alteração da média actual do aluno;*
- *Operação que devolve uma lista com os códigos das disciplinas a que o aluno se encontra inscrito;*
- *Operação que determina o número de disciplinas a que um aluno está inscrito;*
- *Operação que determina se o aluno está inscrito a uma disciplina cujo código é dado como parâmetro;*
- *Operação que inscreve o aluno a uma nova disciplina;*

Designando por **Turma** uma colecção de fichas de alunos inscritos, à qual se associa também um código de disciplina e o seu nome (note-se que todos os alunos devem ter esta disciplina na sua lista de inscrições), pretende-se agora desenvolver o seguinte conjunto de operações sobre tal colecção de fichas de aluno:

- *Inserir uma nova ficha de aluno na turma;*
- *Verificar se um aluno cujo número é dado está inscrito na turma;*
- *Remover a ficha do aluno cujo número é dado como parâmetro;*
- *Determinar o número actual de alunos inscritos;*
- *Operação que devolve a lista com os números dos alunos inscritos;*
- *Operação que devolve um conjunto contendo os números dos alunos com média superior à média dada como parâmetro;*
- *Operação que determina a maior média da turma;*
- *Operação que devolve a ficha completa de um aluno cujo número é dado;*

ANÁLISE DOS REQUISITOS

Da análise do problema torna-se evidente a necessidade de começarmos por definir a classe que irá permitir criar *fichas de aluno*. Designaremos tal classe por **`FichaAluno`**.

`FichaAluno`

Tal classe, muito bem caracterizada nos requisitos do problema não oferece grandes dúvidas quanto à sua estruturação. Assim, a classe **`FichaAluno`**, possuirá a seguinte estrutura:

```
// variáveis de instância
private String numero;   // número do Aluno
private String nome;     // nome do aluno
private String curso;    // código do Curso
private String anoCurso; // ano actual
private double media;    // média actual do aluno
private ArrayList<String> discp; // lista disciplinas
```

A lista das disciplinas foi representada como sendo um `ArrayList<String>` e não um `TreeSet<String>` porque se admite que na criação de cada ficha de aluno tal lista nunca contém duplicados, e que depois nunca admitiremos que possa ter, e também porque se admite que a ordem das disciplinas na inscrição possa ser importante. Se assim não fosse, o `TreeSet<String>` teria vantagens de representação.

Analisemos agora os construtores que fará sentido que sejam desenvolvidos. Sem dúvida que deveremos redefinir o construtor predefinido de JAVA `FichaAluno()`. Ainda que não sendo uma grande semântica, uma possibilidade seria:

```
// Construtores
public FichaAluno() {
   numero = "?"; nome = "?"
   curso = "?"; anoCurso = "?";
   media = 0.0;
   discp = new ArrayList<String>();
}
```

O segundo construtor, em geral também óbvio e muito comum, é aquele que recebe todas as partes que irão constituir a instância, e as atribui *correctamente* às respectivas variáveis de instância. Atribuir correctamente significa, sobretudo, garantir que não há entre *parâmetros* e *variáveis de instância* qualquer partilha de objectos, ou seja, que por erro de atribuição, nenhuma variável de instância fica a referenciar um objecto qualquer que já está a ser referenciado por uma outra variável, neste caso a que foi usada como parâmetro na invocação do construtor. Tais precauções não fazem sentido para tipos simples – que são valores -, devendo aplicar-se apenas a tipos referenciados, exceptuando as instâncias da classe `String` (e `Integer`, `Double`, etc.) que em JAVA são tratadas como valores, ou seja, são copiadas e são imutáveis portanto.

Assim, o segundo construtor terá o seguinte código:

```
public FichaAluno(String num, String nom,
                  double media, String curso,
                  String anoc,
                  ArrayList<String> cod_discps) {
   numero = num; nome = nom;
   this.media = media; this.curso = curso;
   anoCurso = anoc;
   // copiar o ArrayList parâmetro para discp
   discp = new ArrayList<String>();
   for(String cod : cod_discps) { discp.add(cod); }
}
```

Preferiu-se neste caso realizar uma cópia explícita do `ArrayList` parâmetro a realizar uma operação de *clone* do mesmo, já que a mesma implicaria realizar *casting*, dado que o método `clone()` de qualquer colecção devolve um `Object`, cf. em

```
/** copiar o ArrayList parâmetro para discp */
discp = new ArrayList<String>();
discp = (ArrayList<String>) cod_discp.clone();
```

Esta operação é, em JAVA5, considerada *"unsafe"* porque o compilador não pode garantir que o "casting" esteja correcto. Por outro lado, o método `clone()` da classe `Object` não nos serve pois não realiza uma cópia adequada, antes uma partilha de referências. Por se tratarem de *strings*, objectos imutáveis, neste caso até funcionaria bem, mas trata-se de uma questão de metodologia, por um lado, e por outro para evitar problemas de *casting* e código desnecessariamente *unsafe*.
Como vimos, um simples iterador `for()` resolve-nos elegantemente o problema.

O terceiro construtor, em geral também óbvio e muito comum, é o construtor de cópia, que copia toda a informação da ficha parâmetro para a ficha a criar. Os construtores de cópia baseiam-se na utilização dos métodos `getX()`. Não tendo estes sido ainda apresentados, serão usados os métodos `getX()` definidos imediatamente a seguir.

```
public FichaAluno(FichaAluno ficha) {
   numero = ficha.getNumero();
   nome = ficha.getNome();
   media = ficha.getMedia; curso = ficha.getCurso();
   anoCurso = ficha.getCurso();
   discp = ficha.getDiscp();  // ver código; faz cópia
}
```

Analisemos agora o conjunto de métodos de instância que pretendemos implementar sobre uma instância de `FichaAluno`. Em primeiro lugar os usuais interrogadores e modificadores básicos, tendo-se poupado o uso (aconselhado) de comentários para documentação `/** */` apenas por questões de espaço.

Os métodos que consultam e devolvem o número, nome, curso, ano e média do aluno, são,

```
public String getNumero() { return numero; }
public String getNome() { return nome; }
public double getMedia() { return media; }
```

```
public String getAnoCurso() { return anoCurso; }
public String getCurso() { return curso; }
```

Os métodos que permitem modificar o nome e a média actual de uma ficha de aluno são,

```
public void setNome(String nom) { nome = nom; }
public void setMedia(double nvMed) { media = nvMed;}
```

A operação que devolve um `ArrayList<String>` das disciplinas a que o aluno está inscrito tem a mesma particularidade da partilha de referências anteriormente referida. Programar apenas um `return discp` em tal método, consistiria em devolver uma referência (apontador ou "ponteiro") para o *arraylist* de códigos das disciplinas do objecto ficha de aluno. Quem quer que no exterior recebesse tal referência e a atribuísse a uma qualquer variável, por exemplo `disc_Ana`, caso em seguida com tal variável realizasse operações de inserção, como `disc_Ana.add("Física I")`, estaria a modificar o estado interno do *arraylist* referenciado por `disc_Ana` mas também, dado serem partilhados, o estado da variável de instância `discp`, modificando assim o estado interno da ficha do aluno (não haveria pois encapsulamento e protecção!). Assim, temos que codificar tal método de forma a que não haja partilha de endereços, o que faremos através da criação de uma cópia que será dada como resultado.

```
/** Devolve uma cópia dos códigos das disciplinas a que
  o aluno está inscrito
*/
public ArrayList<String> getDiscp(){
     ArrayList<String> dsp = new ArrayList<String>();
     for(String cod : discp) { dsp.add(cod); }
     return dsp;
}
```

Mais uma vez poderíamos ter devolvido como resultado um `HashSet<String>` ou um `TreeSet<String>`. Mas devolvendo um `ArrayList<String>` mantemos a ordem das inscrições, tal como existe na ficha. Apenas não é necessário fazer `clone()` de `cod` por se tratar de uma *string*. Por outro lado, devolvendo um `ArrayList<String>` este método pode ser directamente usado no construtor por cópia.

A operação de verificar se o aluno (a cuja ficha diz respeito) está inscrito numa dada disciplina, consiste em utilizar-se o método `contains` da API de `ArrayList<E>` sobre a variável `discp`, devolvendo tal método um valor lógico como resultado.

```
/** Verifica se a disciplina pertence
    às inscrições do aluno
*/
public boolean inscritoA(String codDiscp) {
      return discp.contains(codDiscp);
}
```

Devolver o número de disciplinas a que o aluno está inscrito é calcular o tamanho actual do *arraylist* de códigos de disciplinas a que o aluno está inscrito segundo a sua ficha.

```
/** Total de disciplinas a que o aluno está inscrito */
public int numInscricoes() { return discp.size(); }
```

O método `void inscreveA(String novaDiscp)` será o método a desenvolver para se inscrever um aluno a mais uma *nova* disciplina. Assim, é importante que, antes que tal método seja invocado, se possa ter a certeza de que o aluno não está inscrito a tal disciplina. O método ou programa invocador do método `inscreveA(novaD)`, deverá, antes de realizar a invocação, usar o método `inscritoA(novaD)` para verifica se tal disciplina já pertence ou não ao conjunto das disciplinas a que o aluno está inscrito. Se já existir, então não será necessário realizar a invocação do método pois tal código *está errado*. Trata-se de uma verificação prévia que detecta um erro e evita a invocação. O método, *que deste modo será sempre invocado em situações correctas*, limita-se a usar o método `add(E elem)` de `ArrayList<E>`, inserindo o código no fim do *arraylist*.

```
/**Junta um novo código de disciplina às inscrições do aluno */
    public void inscreveA(String novaDiscp) {
       discp.add(novaDiscp);
    }
```

Operação que permite obter uma representação completa sob a forma de texto de um qualquer objecto do tipo `FichaAluno`, para que possa ser visualizado em ecrã ou até ser escrito num ficheiro de texto.

```
public String toString() {
  StringBuilder s = new StringBuilder();
  s.append(" --- FICHA DO ALUNO Nº: ");
  s.append(numero); s.append("\n");
  s.append("NOME : "); s.append(nome); s.append("\n");
  s.append("MEDIA : ");
  s.append(media); s.append("\n");
```

```
        s.append("CURSO : ");
        s.append(curso); s.append("\n");
        s.append("ANO : ");
        s.append(anoCurso); s.append("\n");
        s.append("-----  INSCRITO A -------------------\n");
        for(String cod : discp) {
            s.append(cod) ; s.append("\n");
        }
        return s.toString();
    }
```

A operação que realiza a criação de uma cópia de uma `FichaAluno` será realizada, tal como já fizemos em projectos anteriores e continuaremos a fazer em projectos futuros, criando uma nova instância da classe `FichaAluno` usando o *construtor de cópia*, o que torna a operação sistemática e sempre muito simples.

```
/** Clonagem – criação de instância que é uma cópia
 */
 public FichaAluno clone() {
     return new FichaAluno(this);
 }
```

Vamos em seguida analisar a estrutura e o comportamento da classe que irá estruturar estas fichas, a classe que designaremos por **`TurmaList`**.

TurmaList

Nesta primeira versão deste projecto, e de forma propositada, *vamos cometer um erro muito grave*, não de análise mas já mesmo de implementação, ao considerarmos que a classe **Turma** estrutura as fichas de aluno sob a forma de uma ***lista de fichas sem ordem especial a não ser a sua ordem de entrada***, tendo-se assim decidido utilizar a classe `ArrayList<E>`, que implementa uma estrutura sequencial indexada de 0 até n – uma lista -, em "arrays" dinâmicos em tamanho e virtualmente infinitos de elementos de tipo `E`, possuindo um enorme conjunto de métodos para operar sobre tal estrutura (ver API).

Assim, para além do seu código e do seu nome, a classe turma possuirá um *arraylist* de `FichaAluno`, cf. as variáveis de instância a seguir apresentadas.

```
// Variáveis de Instância
 private String codigo; // código da Disciplina/Turma
 private String nomeDiscp;
 private ArrayList<FichaAluno> turma;
```

Vamos criar os construtores usuais.

Note-se que o construtor completo assume que o `ArrayList` de fichas de aluno que lhe é passado como parâmetro foi anteriormente validado, ou seja, o construtor não assume qualquer responsabilidade por copiar fichas que possam conter incoerências, isto é, não cumpram as propriedades (invariantes, requisitos, regras, etc.).

```
// Construtor por omissão
public TurmaList() {
    codigo = "?"; nome = "?";
    turma = new ArrayList<FichaAluno>();
}

/** Construtor completo
  Insere uma Colecção de Fichas válidas em turma
*/
public TurmaList(String codigo, String nome,
                 Collection<FichaAluno> colFichas) {
   codigo = "?"; nome = "?";
    turma = new ArrayList<FichaAluno>();
    turma.addAll(colFichas);
}
```

Em primeiro lugar, usamos como parâmetro o tipo `Collection<FichaAluno>` por uma questão de generalização. Deste modo o parâmetro *actual* tanto poderá ser um `ArrayList<FichaAluno>`, como uma `LinkedList<FichaAluno>`, como um `HashSet<FichaAluno>`, etc. Abstraímos e, assim, generalizamos portanto o método.

Porém, este construtor aparentemente correcto, não funciona correctamente, dado que o método `addAll()` das colecções de JAVA5 não faz, nem poderia fazer, uma cópia de cada objecto a adicionar à colecção receptora, e apenas lhe passa o endereço do objecto. Ou seja, cada objecto passa a estar partilhado pelas duas colecções: a receptora e a colecção parâmetro (violação do encapsulamento mais uma vez … e sempre !).

Assim, e mais uma vez porque o método `clone()` de colecções também não garante a cópia e obriga ainda por cima a *castings unsafe*, cada `FichaAluno` oriunda do parâmetro de entrada deverá ser por nós explicitamente copiada para a colecção receptora, iterando sobre a colecção parâmetro usando o iterador `for()` e usando o método `clone()` definido na classe `FichaAluno`.
Este será, sempre que tivermos que trabalhar com colecções de JAVA, o nosso método correcto de copiarmos colecções para as nossas variáveis de instância, por forma a garantirmos a preservação do encapsulamento.

```
/** Construtor completo.
   Copia correctamente uma colecção de Fichas válidas
 */
public TurmaList(String cod, String nom,
                 Collection<FichaAluno> colFichas) {
    codigo = cod; nome = nom;
    turma = new ArrayList<FichaAluno>();
    for(FichaAluno ficha : colFichas)
         turma.add(ficha.clone());
}
```

O construtor de cópia usa o método getFichas() que é apresentado em seguida.

```
/** Construtor de cópia */
public TurmaList(TurmaList t) {
    codigo = t.getCodDisciplina();
    nome = t.getNomeDiscp();
    turma = t.getFichas();
}
```

Vamos agora analisar os métodos de instância começando pelos usuais métodos de consulta e de modificação.

```
// Métodos de Instância

 /**  Devolve o nome da Disciplina/Turma */
public String getNomeDiscp() { return nomeDiscp; }

 /** Muda o nome da Disciplina/Turma */
public void mudaNomeDiscip(String novoNome) {
    nomeDiscp = novoNome;
}

 /** Devolve o código da Disciplina/Turma */
public String getCodDisciplina() { return codigo; }

/** Determina o número de alunos da Turma */
public int numAlunos() { return turma.size(); }
```

```
/** Devolve uma cópia do ArrayList<FichaAluno> */
public ArrayList<FichaAluno> getFichas() {
   ArrayList<FichaAluno> aux =
                 new ArrayList<FichaAluno>();
   for(FichaAluno f : turma) aux.add(f.clone());
   return aux;
}
```

Criar uma lista com os números dos alunos da turma, sendo os números dos alunos da turma do tipo `String`, pode ser resolvido usando um `ArrayList<String>` que se inicializa a vazio e para o qual se copiam, um a um, todos os números dos alunos obtidos de cada uma das fiches encontradas no *arraylist* `turma`. Para percorrer o *arraylist* vamos mais uma vez usar o iterador sobre coleccões `for(..)` (ler "para cada …obtida de ...").

```
/** Cria uma lista com os números dos alunos da turma  */
public ArrayList<String> codigos() {
    ArrayList<String> cods = new ArrayList<String>();
    for(FichaAluno ficha : turma) {
        cods.add(ficha.getNumero());
    }
    return cods;
 }
```

Estando algoritmicamente tal solução perfeitamente correcta, tanto mais que não existem dois alunos com o mesmo número e, portanto, tal lista, ainda que sendo uma lista, é de facto um conjunto pois não contém duplicados, a verdade é que, em informática, existe uma certa tendência para nos requisitos dos projectos nos pedirem *listas de coisas* que, de facto, não são listas mas sim conjuntos. No exemplo anterior, e embora o que de facto nos tenha sido pedido tenha sido a *lista dos números dos alunos* e tenha sido isso que tenha sido documentado e programado, mais correcto seria termos programado e documentado de forma explícita que o resultado do método é um *conjunto de números de alunos*.

Procurando realizar tal correcção, que é bastante simples, teríamos apenas que procurar saber quais as classes de JAVA5 que implementam conjuntos. Teríamos duas possíveis implementações genéricas: `TreeSet<E>` e `HashSet<E>`. A classe TreeSet<E> não só implementa conjuntos mas também permite definir uma ordenação dos seus elementos. A classe HashSet<E> pressupõe eficácia na implementação. Vamos usar esta segunda e, dado que o método que permite juntar um elemento a um conjunto se designa também por `add(..)`, o código anterior ficaria:

```
/** Cria um conjunto com os números dos alunos da turma */
public HashSet<String> codigos() {
    HashSet<String> cods = new HashSet<String>();
    for(FichaAluno ficha : turma) {
        cods.add(ficha.getNumero());
    }
    return cods;
}
```

Verificar se um dado aluno cujo código é dado está inscrito na turma, consiste em realizar uma operação de pesquisa sequencial sobre as fichas dos alunos da turma, quer até encontrar tal número em cujo caso o aluno está inscrito, quer até esgotar o conjunto das fichas, em cujo caso o aluno não está inscrito. Dado que não se trata de um algoritmo que necessite garantidamente de percorrer todo o espaço de procura (todas as fichas), pois pode terminar a qualquer momento, ou seja, não é exaustivo ou de "varrimento", deve ser usado um `Iterator` porque o ciclo `for(..)` sobre colecções não permite incluir condições de paragem da iteração.

```
/** Verifica se um aluno de número dado existe */
public boolean existeAluno(String numAluno) {
  Iterator<FichaAluno> it = turma.iterator();
  FichaAluno ficha; String numero;
  boolean existe = false;
  while(it.hasNext() && !existe) {
    ficha = it.next(); numero = ficha.getNumero();
    if(numero.equals(numAluno)) existe = true;
  }
  return existe;
}
```

Inserir um novo aluno na turma depois de se ter a garantia de que não está ainda inscrito (usando o método anterior) consiste simplesmente em *juntar a cópia* da ficha dada como parâmetro à turma.

```
/** Insere um novo aluno na turma */
public void insereAluno(FichaAluno ficha) {
     turma.add(ficha.clone());
 }
```

Determinar a maior média da turma trata-se mais uma vez de codificar um algoritmo que cai na classe dos algoritmos que têm por essência a iteração completa da colecção de fichas representadas neste caso no *arraylist*. O que pretendemos neste caso de cada uma delas? Obter a média do aluno. Para quê? Para a comparar com a maior média até então encontrada e guardar a deste aluno se for superior. Finalmente, com que média deve ser comparada a média do primeiro aluno? Com uma média suficientemente baixa para que a dele seja superior de certeza (algoritmo de máximo).

```
/** Determina a maior média de todos os alunos */
public double maiorMediaTurma() {
  double maiorMedia = Double.MIN_VALUE;
  double mediaAluno = 0.0;
  for(FichaAluno ficha : turma) {
     mediaAluno = ficha.getMedia();
     if(mediaAluno > maiorMedia)
           maiorMedia = mediaAluno;
  }
  return maiorMedia;
}
```

Criar a lista ou o conjunto dos números dos alunos com média superior à média dada como parâmetro continua a ser um problema que necessita de uma iteração sobre a colecção de fichas (resolvida com o `for(..)`) . De cada ficha vamos consultar a média que vamos comparar com a média dada como parâmetro, e caso a média do aluno seja maior consultamos da ficha o seu número, que guardamos ou num `ArrayList` ou num `HashSet` que no final da iteração devolvemos como resultado do método.

```
 /**  Cria a lista com os códigos dos alunos com média
      superior à dada como parâmetro (exº média > 12).
 */
public TreeSet<String> codAlsMedSup(double notaRef){
    TreeSet<String> cods = new TreeSet<String>();
    for(FichaAluno ficha : turma) {
       if(ficha.getMedia() > notaRef)
            cods.add(ficha.getNumero()) ;
  }
  return cods;
}
```

A operação de remover a ficha de um aluno garantidamente existente, tal como a operação de inserir um novo aluno, pressupõe uma validação prévia da existência de tal

aluno na turma. Sendo certo portanto que tal aluno existe, a operação vai consistir, antes de mais, na procura da posição ocupada pela ficha de tal aluno no *arraylist*, ou seja o valor do índice (entre 0 e o nº de fichas – 1) do *arraylist* em que se encontra guardada. Encontrado tal valor, e caso se tratasse de um *array* normal (cf. [] de C ou [] JAVA), em seguida deveríamos implementar algoritmos para eliminar tal ficha procurando reaproveitar o seu espaço da melhor maneira. A classe `ArrayList<E>` possui um método `remove(int i)` que, felizmente, faz tudo isto automaticamente, ou seja, remove o elemento que se encontra no índice dado como parâmetro e automaticamente faz o "shift-up" de todos os elementos em índices superiores ao removido.

```
/** Remove a ficha de um aluno garantidamente existente */
public void removeAluno(String numAluno) {
    Iterator<FichaAluno> it = turma.iterator();
    FichaAluno ficha = null;
    String numero; int index = 0;
    boolean encontrado = false;
    while(it.hasNext() && !encontrado) {
      ficha = it.next();
      numero = ficha.getNumero();
      if(numero.equals(numAluno)) encontrado = true;
      else index++;
    }
    turma.remove(index);
}
```

No entanto existe uma outra forma mais simples de remover de uma colecção um objecto que já foi por nós identificado como sendo o que queremos remover. Um dos métodos de `Iterator<E>` é o método `remove()` que elimina da colecção o último objecto iterado. Usando tal método nem sequer necessitamos de conhecer o seu índice. O código final, mais simples ficaria então:

```
/** Remove a ficha de um aluno garantidamente existente */
public void removeAluno1(String numAluno) {
    Iterator<FichaAluno> it = turma.iterator();
    FichaAluno ficha = null; String numero;
    boolean encontrado = false;
    while(it.hasNext() && !encontrado) {
      ficha = it.next(); numero = ficha.getNumero();
      if(numero.equals(numAluno)) it.remove();
    }
}
```

O método que vai devolver a ficha do aluno cujo número é dado, começa por criar um iterador sobre as fichas da turma e enquanto existirem fichas e não tiver encontrado a ficha do aluno com tal número vai percorrendo a colecção. A variável `ficha`, que vai guardar cada uma das fichas percorrida, é inicializada a `null` pelo que, se não for encontrada nenhuma ficha com tal número, o método devolverá o valor `null`.

Se a ficha de tal aluno for encontrada, então o ciclo `while` será abandonado e na variável `ficha` temos a ficha do aluno cujo número foi dado como parâmetro. Porém coloca-se a questão. Temos uma *cópia* dessa ficha ou temos uma *referência* (apontador ou ponteiro) para ela? A resposta é que temos uma referência porque um iterador não copia a colecção que itera apenas referencia – aponta para - os respectivos elementos. Assim, se escrevêssemos return ficha estaríamos a enviar para o exterior de `TurmaList` o endereço de memória da ficha deste aluno e quem o recebesse poderia modificar tal ficha de aluno e, portanto, a turma (encapsulamento, sempre !).

A linha de `return` testa o valor da variável booleana de teste e devolve um dos dois valores possíveis: `null` ou *a cópia da ficha*.

```
/** Devolve a Ficha completa do aluno de número dado */
public FichaAluno procuraAluno(String numAluno) {
    Iterator<FichaAluno> it = turma.iterator();
    FichaAluno ficha = null; String numero;
    boolean existe = false;
    while(it.hasNext() && !existe) {
        ficha = it.next();
        numero = ficha.getNumero();
        if(numero.equals(numAluno)) existe = true;
    }
    return existe ? ficha.clone() : null;
}
```

```
/** Representação textual da Turma */
public String toString() {
     StringBuilder s = new StringBuilder();
     s.append("---------  TURMA --------------------\n");
      s.append(“DISCIPLINA : “); s.append(nomeDiscp\n);
      s.append(“CODIGO : “); s.append(codigo\n);
      s.append("-------- ALUNOS --------------------\n");
     for(FichaAluno ficha : turma) {
          s.append(ficha.toString());
```

```
        }
        s.append("-----------------------------------\n")
        return s.toString();
    }
```

```
    /** Clone de TurmaList */
    public TurmaList clone() {
       return new TurmaList(this);
    }
```

Para classes complexas como `TurmaList`, não faz, em geral, sentido codificar o método equals(), podendo-se sempre usar o método `equals()` de `Object` que testa os endereços dos objectos em comparação.

Tendo codificado todos os construtores e métodos de instância das duas classes que constituem as classes principais do projecto, resta apenas ter em consideração que tais classes, quer venham a ser completadas em ambiente JDK puro ou em ambiente de suporte BlueJ, deverão ter os seguintes "templates" de definição no respectivo ficheiro com o seu nome e extensão *.java*:

```
import java.util.*;
import java.io.*;
public class FichaAluno {
  …….

}
```

```
import java.util.*;
import java.io.*;
public class TurmaList {
  …….

}
```

PROTOTIPAGEM EM BLUEJ

CLASSE DE TESTE PARA BLUEJ

```
/**
 * Classe de teste que constrói uma pequena instância da
 * classe devolvida como resultado da execução de main().
 * No ambiente BlueJ a execução do método main() da classe
 * TestTurmaList criará tal instancia de TurmaList de modo
 * a ser de imediato usada para teste.
 *
 * @author F. Mário Martins
 * @version 1.0/01/2005
 */
import java.util.*;
import java.io.*;
public class TestTurmaList {

  public static TurmaList main() {
      FichaAluno ficha1, ficha2, ficha3;
      FichaAluno ficha4, ficha5, ficha6;
      ArrayList<String> discp = new ArrayList<String>();

      discp.add("66223"); discp.add("66220");
      discp.add("66112"); discp.add("66233"); discp.add("66291");

      ficha1 = new FichaAluno("1211", "Rita", "66", "2", 15.2, discp);
      ficha2 = new FichaAluno("1222", "Paula", "66", "2",14.6, discp);

      discp.clear(); // limpa o ArrayList anterior e insere novos códigos

      discp.add("66244"); discp.add("66376");
      discp.add("66255"); discp.add("66220");
      discp.add("66346"); discp.add("66377");

      ficha3 = new FichaAluno("6633", "Joao", "66","3",12.7, discp);
      ficha4 = new FichaAluno("6644","Artur", "66","3",14.2, discp);
      ficha5 = new FichaAluno("6655", "Ana", "66","3",12.2, discp);
      ficha6 = new FichaAluno("6666", "Rui", "66", "3",14.8, discp);

      // Cria uma TurmaList vazia e insere os alunos
      TurmaList turma1 = new TurmaList();

      turma1.insereAluno(ficha1);turma1.insereAluno(ficha2);
      turma1.insereAluno(ficha3);turma1.insereAluno(ficha4);
      turma1.insereAluno(ficha5);turma1.insereAluno(ficha6);
      return turma1; // devolve a turma criada
  }
 }
```

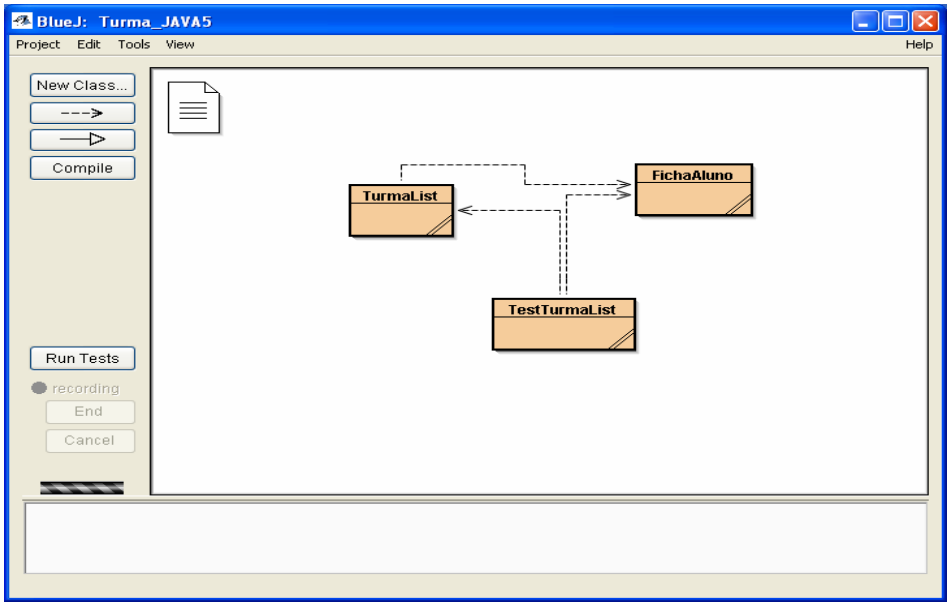
A prototipagem em BlueJ segue uma metodologia muito simples:

- *Criar uma classe com as suas definições mínimas, tipicamente, um construtor vazio, alguns métodos get() e um método de inserção;*
- *Criar uma classe de teste com um método main() que devolva uma instância dessa classe;*
- *Escrever os métodos de instância um a um e, para cada um deles, criar a instância e invocar o método para testar a sua correcção, eventualmente usando INSPECT.*

Neste exemplo, e porque tal exigiria uma apresentação de imagens passo a passo, apresentamos apenas os testes finais globais do projecto completo.

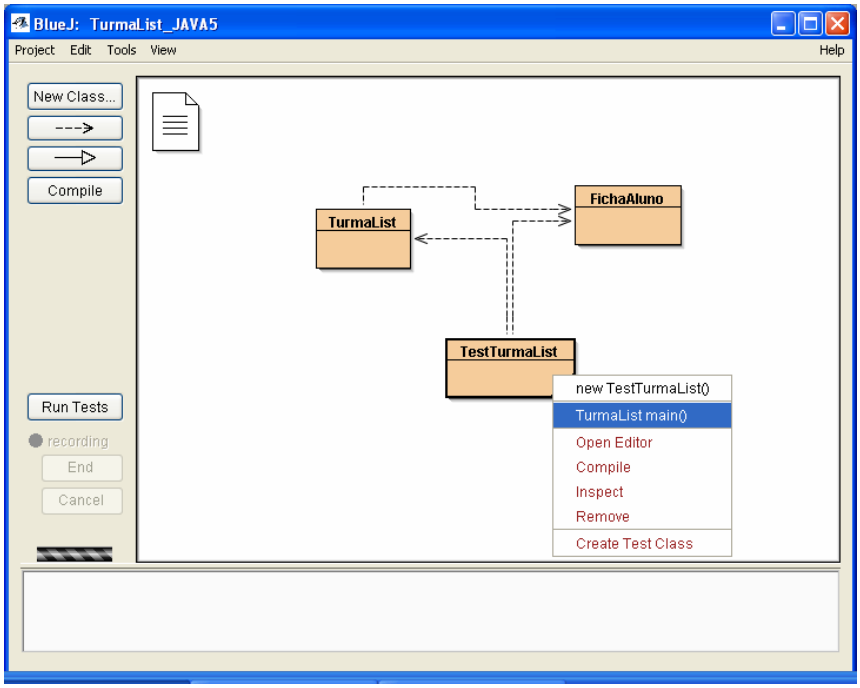
No entanto, o desenvolvimento incremental com prototipagem é a metodologia de desenvolvimento que naturalmente deve ser seguida.

Criado o projecto **Turma_JAVA5** e, através do editor do BlueJ, criadas as classes `FichaAluno`, `TurmaList` e `TestTurmaList` , o **Diagrama de Classes** do projecto, gerado automaticamente pelo BlueJ, é o que se apresenta na figura seguinte.

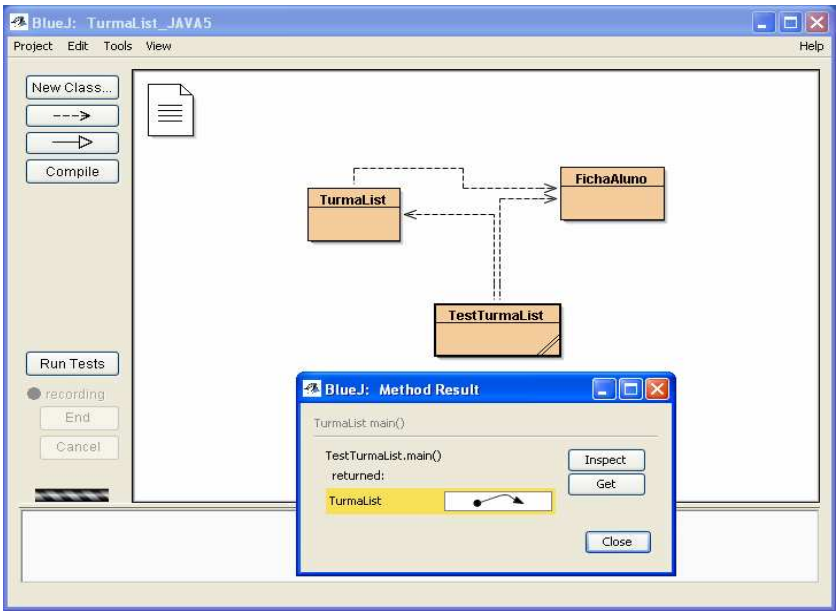


Podemos a partir deste momento criar instâncias de qualquer das classes `FichaAluno` e `TurmaList` usando os seus construtores. Quer num caso quer noutro, a criação de tais instâncias a partir do zero, e sendo tais classes compostas por variáveis de instância que são estruturadas, implica algum trabalho paralelo. A criação de classes auxiliares que fazem tal trabalho revela-se nestes projectos compensador, tal como no exemplo a classe `TestTurmaList`.

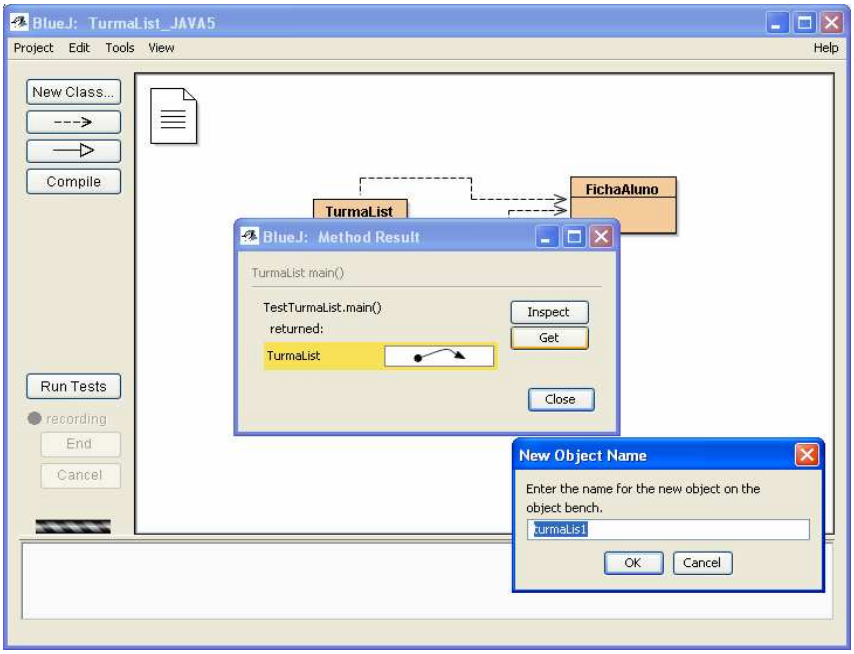
Seleccionada a classe `TestTurmaList` e accionado ◗ surge o menu contendo todas as operações realizáveis sobre esta classe. Dentre estas, estamos particularmente interessados em seleccionar a execução do método `main()` que, como a sua assinatura indica, devolverá como resultado uma instância de `TurmaList`.

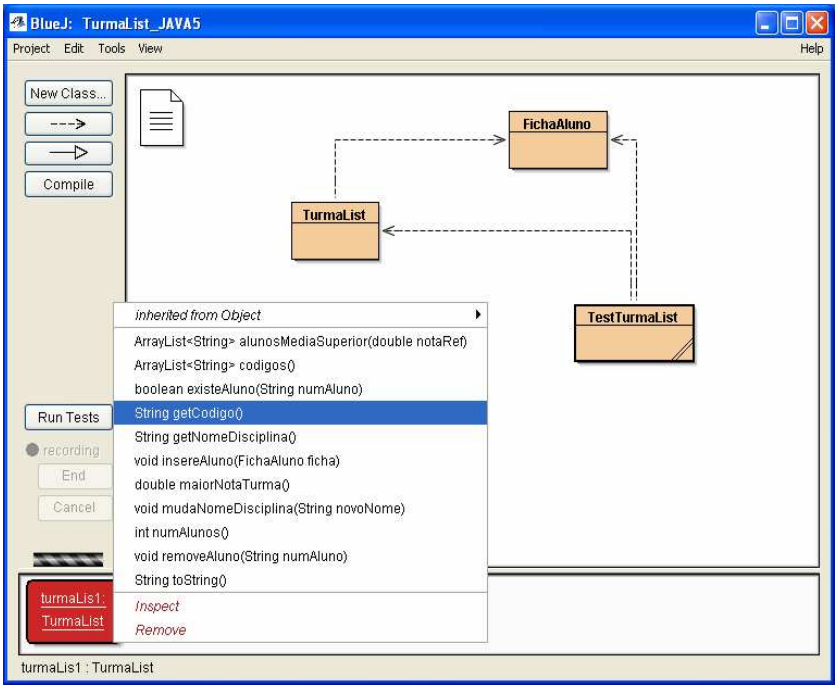


Para tal, depois de seleccionada a operação basta fazermos ◖ para que a mesma seja executada e neste caso surja no ecrã uma janela com o resultado da execução do método, ou seja, um objecto do tipo `TurmaList` ao qual deveremos atribuir um nome.



Vamos fazer o **Get** do objecto e atribuir-lhe um nome ou aceitar o nome proposto pelo BlueJ.



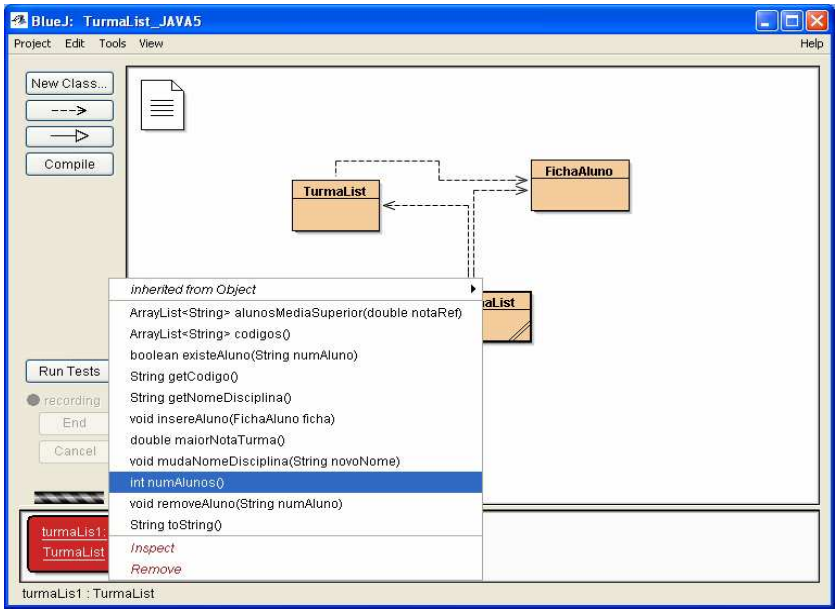


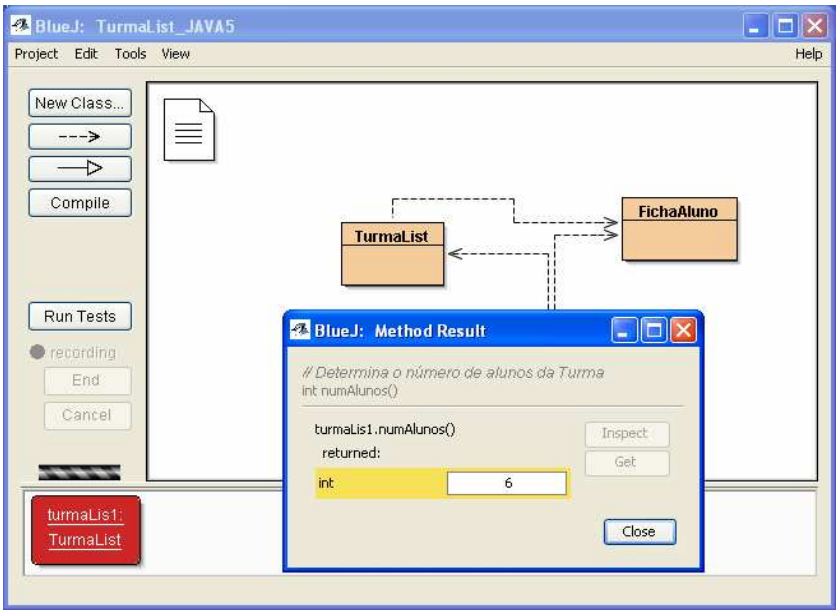
Criada desta forma simples a instância de `TurmaList` – note-se como seria muito mais complexo criar 6 fichas de aluno, colocá-las no espaço de instâncias uma a uma, criar uma turma vazia e inserir as fichas uma a uma, etc.-, e colocada no espaço de instâncias, podemos agora, tal como poderíamos em qualquer momento do projecto desde que já tivéssemos pelo menos os construtores, testar cada um dos métodos de instância da classe `TurmaList` (admite-se que já o havíamos feito com `FichaAluno`).

Na imagem acima é também visível o conjunto de mensagens a que a instância de `TurmaList` é capaz de responder, ou seja, o conjunto de métodos implementados.

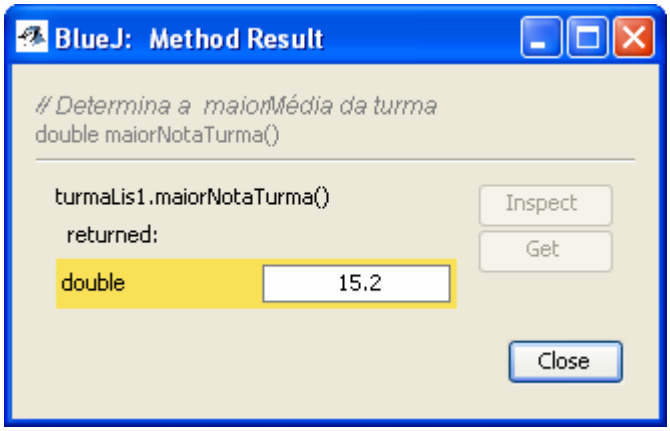
Vamos testar agora as principais operações do projecto. Para cada operação iremos mostrar a sua invocação ou já a fase de introdução de parâmetros se tal for o caso, e o resultado da mesma. Caso o resultado seja mais estruturado utilizaremos *Inspect* para analisarmos os conteúdos dos objectos resultantes.

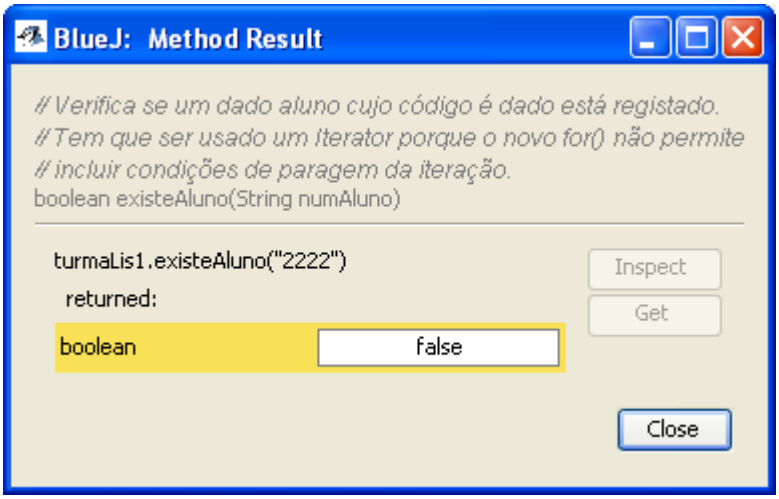
NÚMERO DE ALUNOS

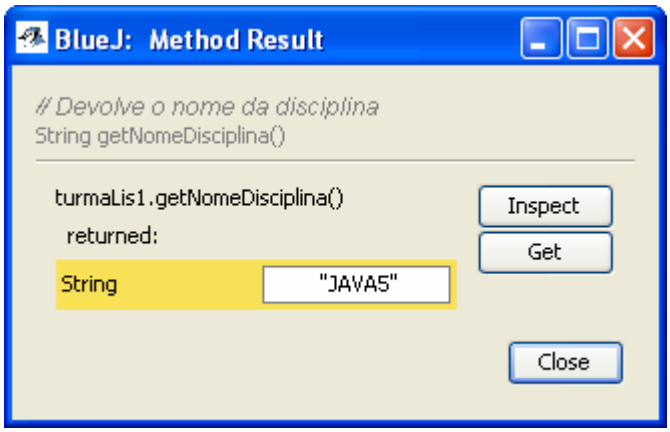




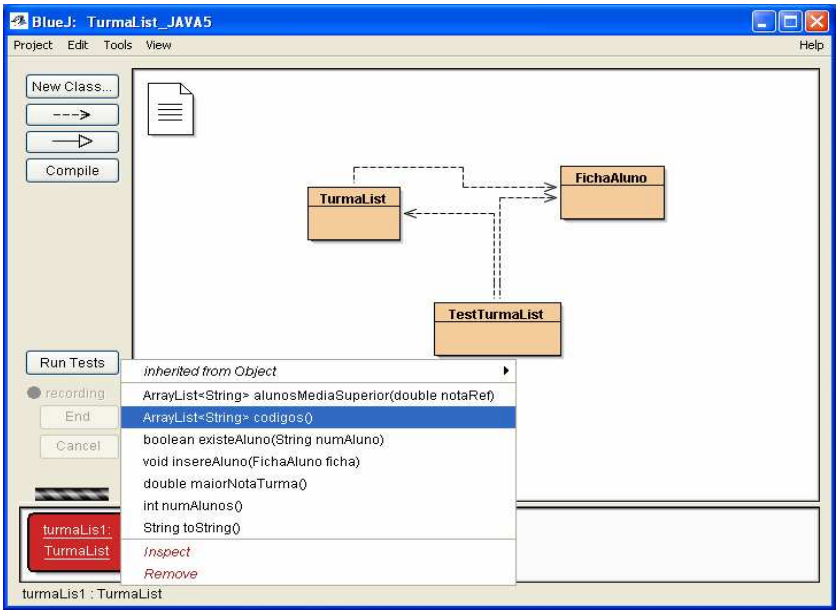
OUTRAS CONSULTAS

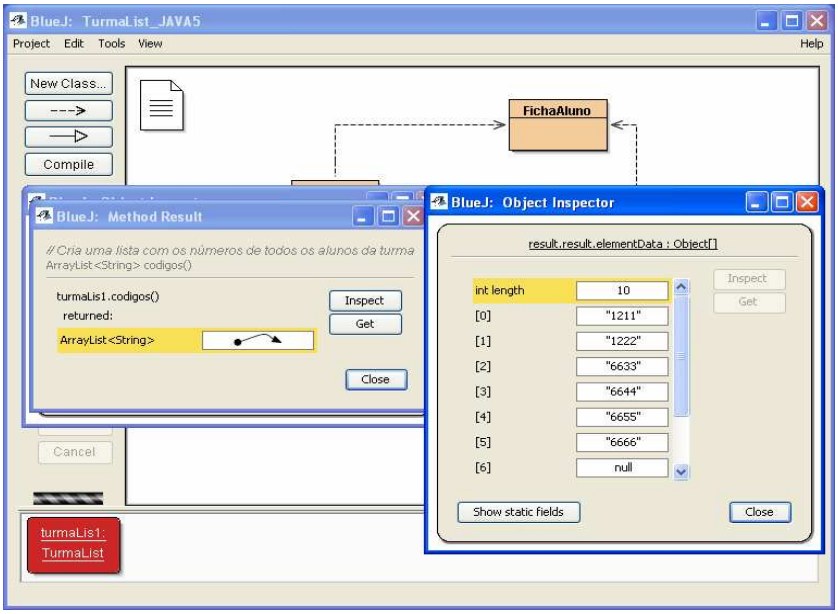




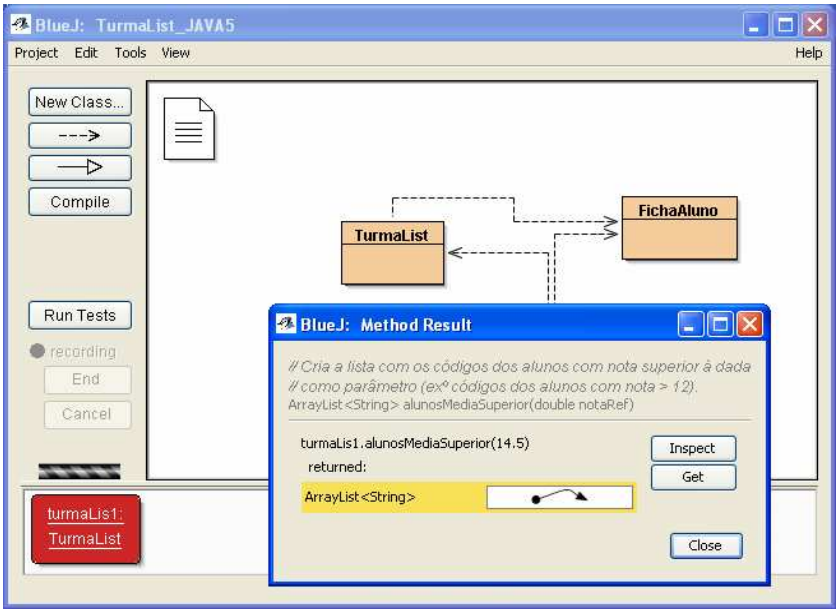


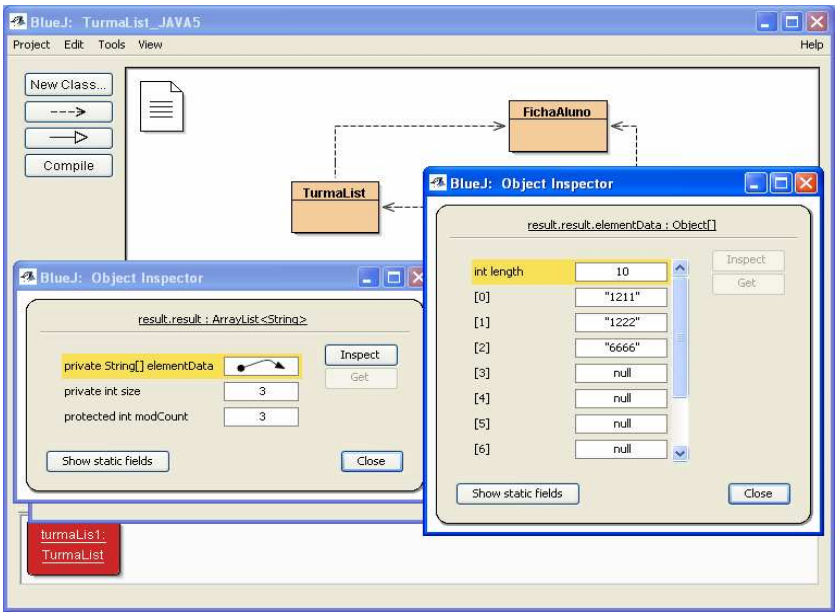
CÓDIGOS DOS ALUNOS





CÓDIGOS DOS ALUNOS COM MÉDIA SUPERIOR A X



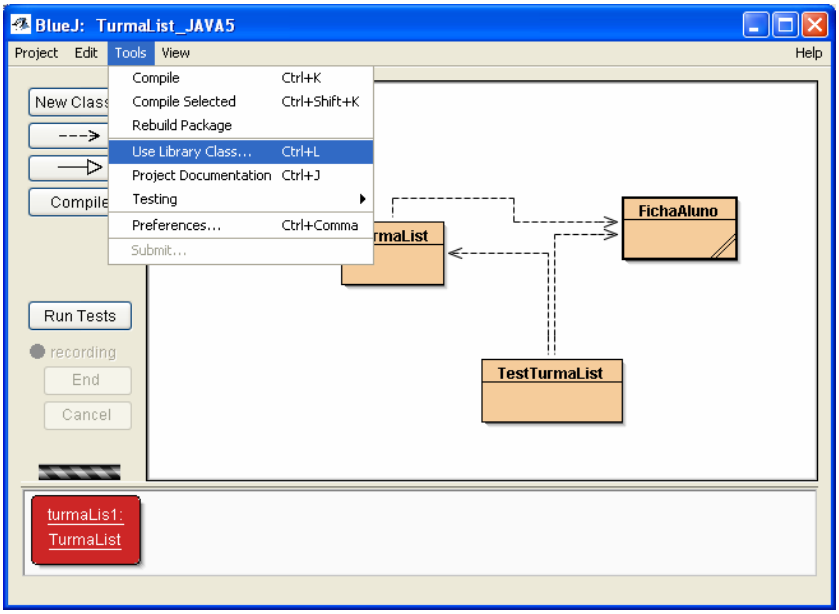


INSERE NOVO ALUNO

A) CRIAR ArrayList<String> com códigos das disciplinas;

B) CRIAR a FichaAluno com número, nome, ano, curso e disciplinas

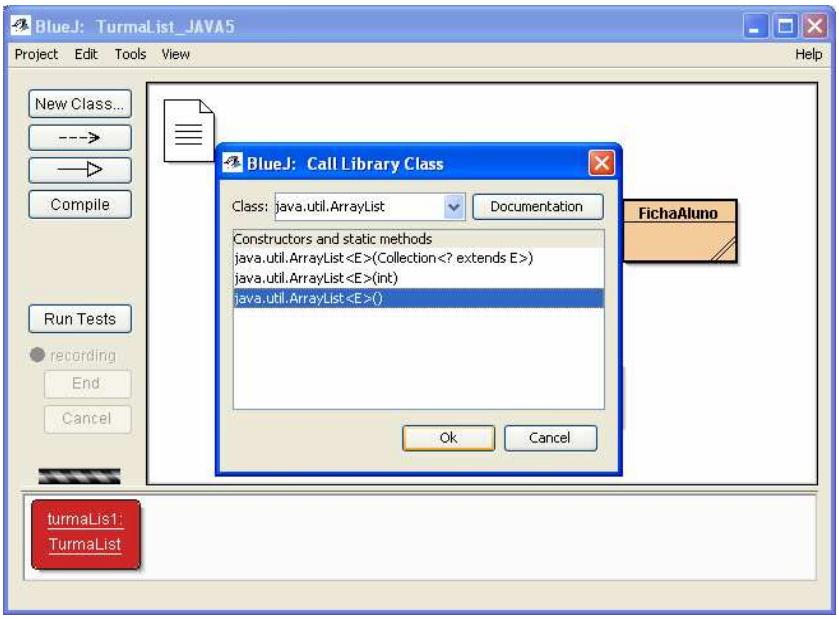
C) INSERIR na turma

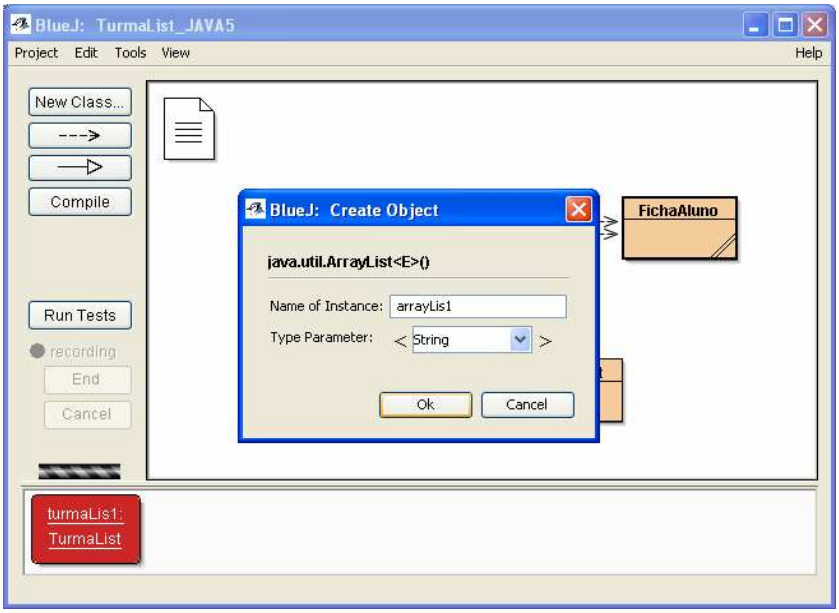


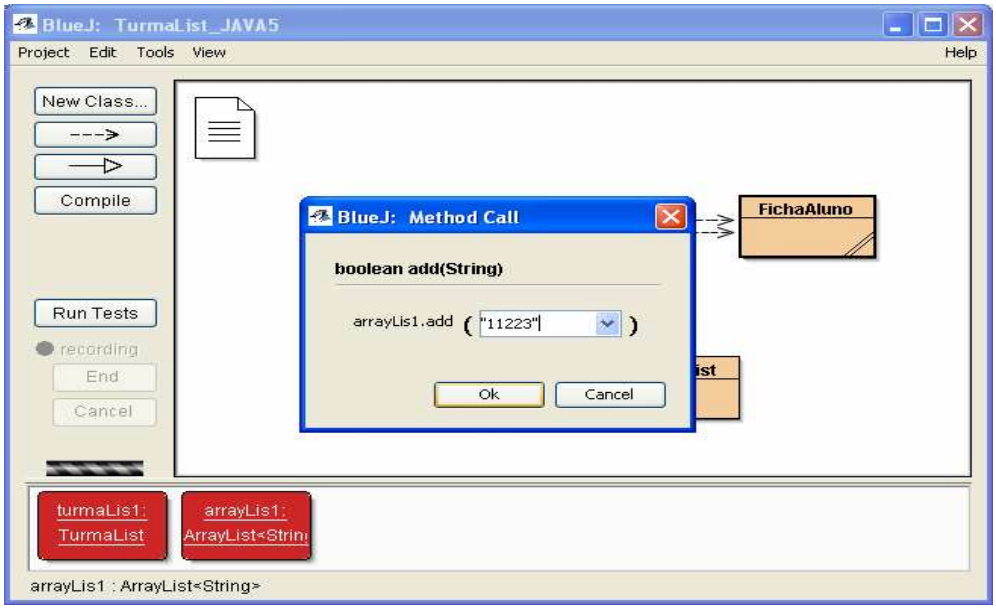
Preparar o `ArrayList<String>` com os códigos das disciplinas, implica usar a biblioteca de classes de JAVA para criar uma instância de ArrayList. Neste caso precisamos de usar o *package* java.util onde reside a classe `ArrayList`.

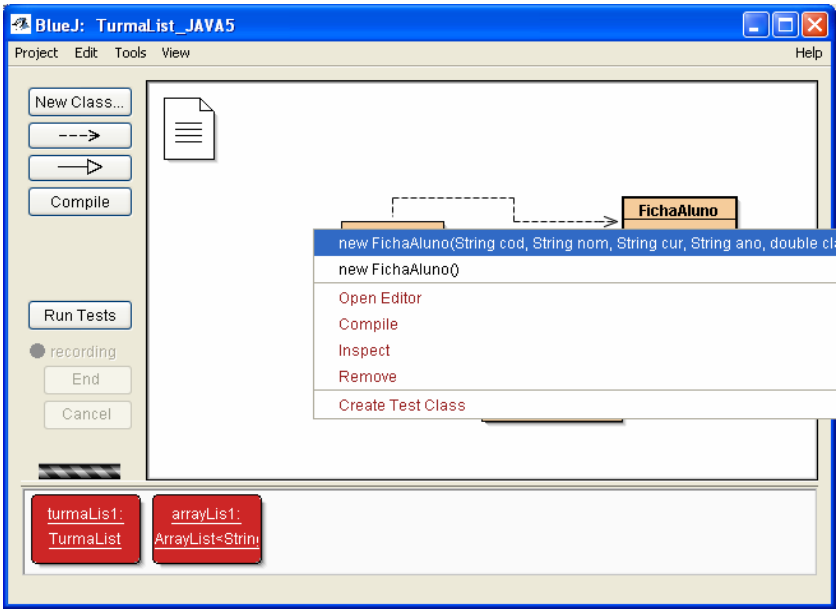
Como o `ArrayList` é parametrizado, depois de seleccionar a classe `ArrayList` o BlueJ pede naturalmente o parâmetro. Este diálogo é apresentado nas duas figuras seguintes.

Depois de termos uma instância vazia do `ArrayList<String>` e usando o método `add()` da API de `ArrayList`, inserimos os códigos das disciplinas do aluno do qual iremos criar a ficha.

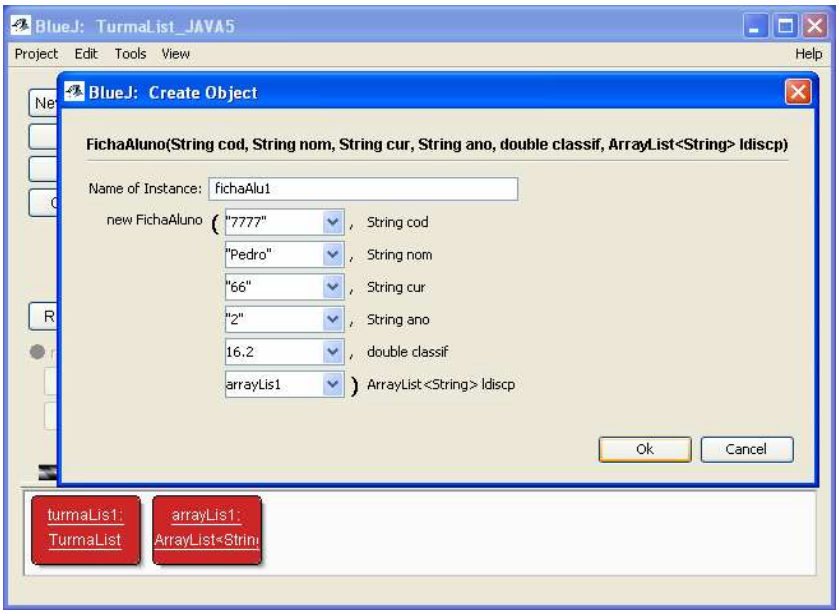

BlueJ: TurmaList_JAVA5
Project Edit Tools View Help
New Class...
Compile
Run Tests
recording
End
Cancel
BlueJ: Call Library Class
Class: java.util.ArrayList
Documentation
Constructors and static methods
java.util.ArrayList<E>(Collection<? extends E>)
java.util.ArrayList<E>(int)
java.util.ArrayList<E>()
Ok
Cancel
FichaAluno
turmaLis1:
TurmaList


BlueJ: TurmaList_JAVA5
Project Edit Tools View Help
New Class...
Compile
Run Tests
recording
End
Cancel
BlueJ: Create Object
java.util.ArrayList<E>()
Name of Instance: arrayLis1
Type Parameter: < String >
Ok
Cancel
FichaAluno
turmaLis1:
TurmaList

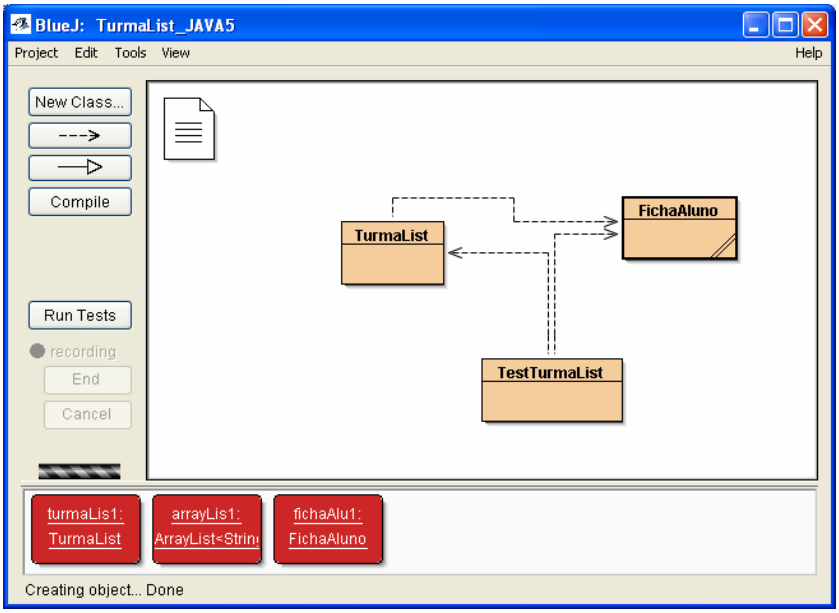




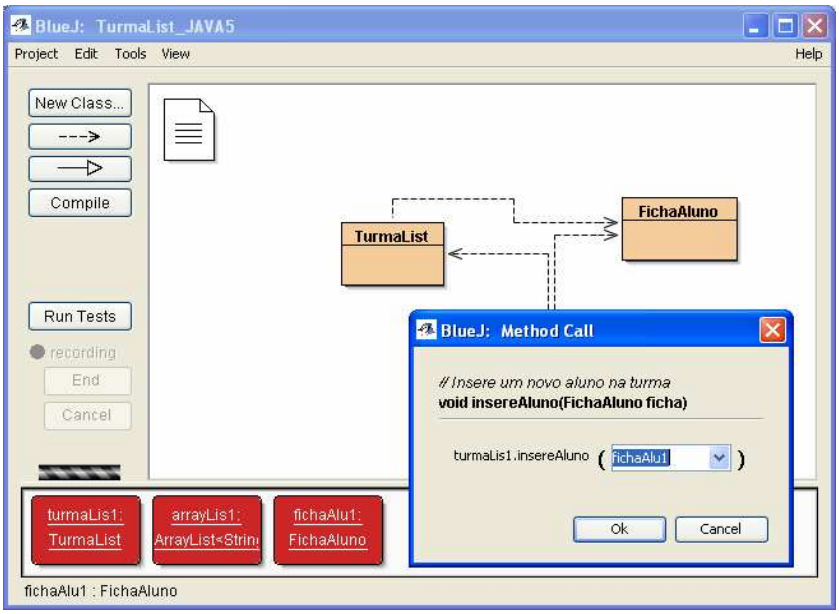
Com o `ArrayList<String>` contendo os códigos das disciplinas já pronto, invoca-se o construtor de `FichaAluno` para criar a ficha do novo aluno.

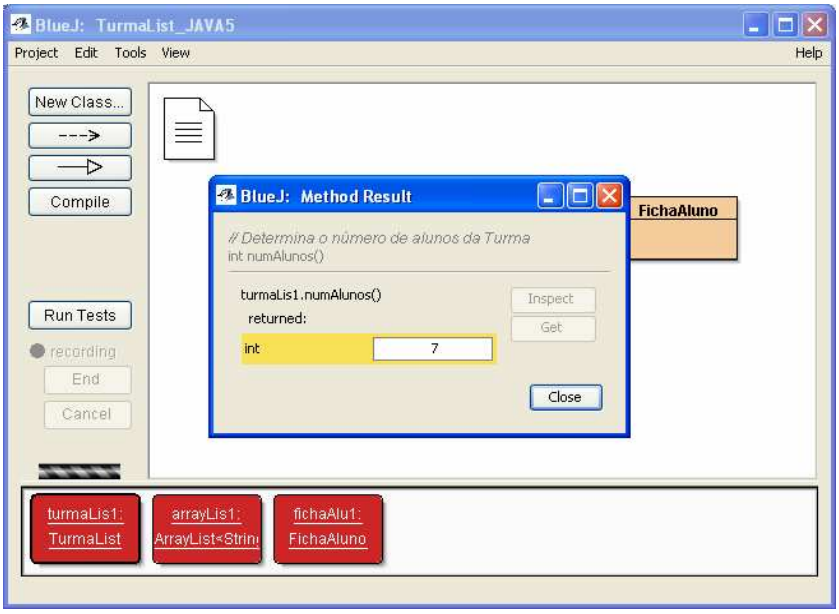


São preenchidos os campos necessários e no parâmetro correspondente à lista de disciplinas, ou via nome ou via na instância arrayList1, introduzimos o identificador do `ArrayList` que criámos.



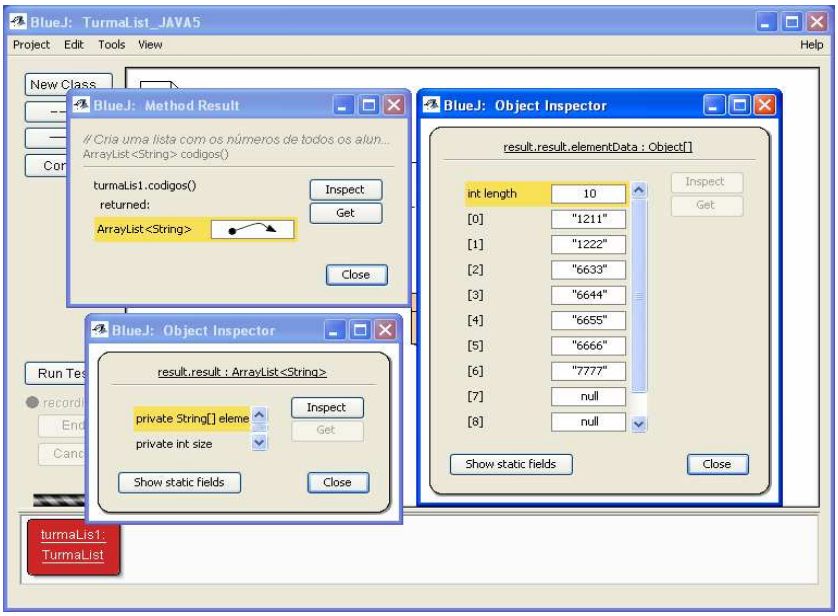
A instância de `FichaAluno` foi criada e está na área de instâncias pronta para ser usada como parâmetro na operação de inserção de um aluno em `turmaList1`.





Inserção do novo aluno na turma e verificação do novo número de alunos da turma para confirmação do resultado da operação.

Nova listagem dos códigos dos alunos inscritos para reconfirmação da inscrição do aluno "7777" que havíamos inscrito anteriormente. Como a ordem de inserção é a de chegada, é naturalmente o último elemento do `ArrayList<FichaAluno>`.



PROJECTO 2

APLICAÇÃO DE REGISTO E CONSULTA DE INFORMAÇÃO SOBRE UMA TURMA DE ALUNOS (VERSÃO 2)

REQUISITOS

Os requisitos deste projecto são exactamente os mesmos do Projecto 1.

ANÁLISE DOS REQUISITOS

Dá análise do problema torna-se evidente a necessidade de começarmos por definir a classe que irá permitir criar *fichas de aluno*. Tal classe não oferece grandes dúvidas quanto à sua estruturação, pelo que a classe **`FichaAluno`** possuirá a mesma estrutura e comportamento que foi apresentada e desenvolvida para o Projecto 3.

Em seguida vamos reanalisar a estrutura e o comportamento da classe que irá estruturar estas fichas, a classe **`Turma`**, sabendo que as operações a implementar são:

- *Inserir uma nova ficha de aluno na turma;*
- *Verificar se um aluno cujo número é dado está inscrito na turma;*
- *Remover a ficha do aluno cujo número é dado como parâmetro;*
- *Determinar o número actual de alunos inscritos;*
- *Operação que devolve a lista com os números dos alunos inscritos;*
- *Operação que devolve um conjunto contendo os números dos alunos com média superior à média dada como parâmetro;*
- *Operação que determina a maior média da turma;*
- *Operação que devolve a ficha completa de um aluno cujo número é dado;*

Na primeira versão deste projecto, ainda que de forma propositadamente pouco correcta, considerou-se que a classe Turma estruturaria as fichas de aluno sob a forma de uma ***lista de fichas sem ordem especial a não ser a sua ordem de entrada***, tendo-se assim decidido utilizar a classe `ArrayList<E>` de JAVA5, que implementa uma estrutura sequencial genérica, indexada de 0 até n – *uma lista* -, em "*arrays*" dinâmicos em tamanho e virtualmente infinitos, contendo elementos de tipo `E`.

Esta decisão tornou relativamente complexas e também pouco eficazes as operações de procura, *tornando-as* sequenciais. Ou seja, encontrar uma ficha de aluno implicou percorrer toda a estrutura linear até que o seu código fosse encontrado, ou não.

Ora se os alunos possuem números que são únicos (*chaves únicas*), então a cada número de aluno poderíamos fazer corresponder, de forma única também a sua ficha, de tal forma que, dado o seu número (*chave/código*) pudessemos ter acesso directo à ficha associada (*valor*).

Matematicamente, garantir que a cada *chave* se associa um e um só *valor* não é fácil. As funções de *hashing* e *rehashin*g são funções matemáticas que em geral nos dão preciosas ajudas nas implementações das designadas *tabelas de hashing*, onde tais pares *chave-valor* são guardados de alguma forma que facilite a sua manipulação.

Como sabemos, em informática estas associações CHAVE-VALOR são, em especial em Sistemas de Bases de Dados, mas não só, extraordinariamente comuns e importantes. Por isso, torna-se muito importante conhecer em detalhe uma classe que implemente o que se pretende.

Em JAVA5 a classe `HashMap<K,V>` implementa uma *tabela de hashing*, onde a cada elemento *chave* absolutamente único do tipo `K` (de *key*) está associado um elemento *valor* do tipo `V`. Um conjunto de métodos (ver APIs) permitirão que se realizem inúmeras operações sobre um `HashMap`, entre outras, inserir um par `<K, V>`, dada uma chave determinar o valor associado, saber o actual conjunto de chaves, saber se uma chave já existe, determinar a lista de valores, percorrer as chaves uma a uma (iterar), percorrer os valores um a um (iterar), etc.

De notar que uma `TreeMap<K,V>` é apenas uma implementação em árvore binária destas mesmas correspondências, e tem, no essencial, a mesma API de `HashMap<K,V>` (ambas são do tipo `Map<K,V>`, sendo apenas implementações distintas). Usaremos para já `HashMap<K,V>`.

Assim, para além do seu código e do seu nome, a classe turma possuirá um *HashMap* que irá associar a cada número de aluno (`String`) o seu valor (`FichaAluno`), cf. as variáveis de instância a seguir apresentadas. Designaremos esta classe por **`TurmaHash`**.

TurmaHash

```
// Variáveis de Instância
private String codigo;        // código da Disciplina
private String nomeDiscp;
// Número de Aluno → FichaAluno
private HashMap<String, FichaAluno> turma;
```

```
/** Construtor que redefine o predefinido */
public TurmaHash() {
    codigo = "?"; nomeDiscp = "?";
    turma = new HashMap<String,FichaAluno>();
}

/** Insere uma colecção de fichas válidas em turma */
public TurmaHash(String cod, String nome,
                 Collection<FichaAluno> colfichas) {
    codigo = codigo; nomeDiscp = nome;
    turma = new HashMap<String,FichaAluno>();
    for(FichaAluno ficha : colfichas)
      turma.put(ficha.getNumero(), ficha.clone());
}
```

```
/** Insere uma colecção de fichas em turma */
public TurmaHash(String codigo, String nome,
                 Map<String,FichaAluno> colfichas) {
    this.codigo = codigo; nomeDiscp = nome;
    turma = new HashMap<String,FichaAluno>();
    turma.putAll(colfichas);
}
```

Este método `putAll()` implementado por qualquer classe `Map` (ou seja `HashMap`, `WeakHashMap` ou `TreeMap`, as três implementações possíveis de *mappings* ou *tabelas de hashing* de JAVA) copia as associações do *mapping* parâmetro para o *hashmap* turma, sobrepondo os seus valores se as chaves já existirem em turma, ou seja, alterando-as, e inserindo as associações novas.

Falta no entanto saber se tal se trata de uma verdadeira cópia ou se, uma vez mais em turma são apenas colocados os endereços das chaves e dos valores do `Map` parâmetro. Neste caso, mesmo consultando a API tal não é esclarecido, ainda que a mesma fale de uma cópia. Mas a palavra "cópia" é neste contexto ambígua. Porém, a consulta do código fonte é esclarecedora. Quer as chaves quer os valores do `Map` parâmetro `colfichas`

são copiados por referência para `turma`, pelo que passam a estar partilhados. Não é feito qualquer `clone()` de tais objectos, o que seria quase impossível dada a generalidade do código de tal método.

Assim, e em circunstâncias em que tivermos que garantir a máxima segurança do código, ou seja, em que houver a possibilidade de que o `Map` parâmetro possa posteriormente ser usado e, em consequência modificar o `HashMap turma`, teremos que garantir a realização de tal cópia de segurança no construtor. O código ficaria então:

```
/** Insere uma colecção de fichas em turma */
public TurmaHash(String codigo, String nome,
                 Map<String,FichaAluno> colfichas) {
    this.codigo = codigo; nomeDiscp = nome;
     turma = new HashMap<String,FichaAluno>();
     for(FichaAluno ficha : colfichas.values())
     turma.put(ficha.getNumero(),ficha.clone());
}
```

```
// Métodos de Instância

/** Devolve o nome da Disciplina */
public String getNomeDisciplina() { return nomeDiscp; }

/** Muda o nome da Disciplina */
public void mudaNomeDisciplina(String novoNome) {
    nomeDiscp = novoNome;
}

/** Devolve o código da Disciplina */
public String getCodDisciplina() { return codigo; }

/** Determina o número de alunos da Disciplina */
public int numAlunos() { return turma.size(); }
```

```
/** Devolve um conjunto com os códigos dos alunos */
public HashSet<String> codigos() {
   HashSet<String> cods = new HashSet<String>();
   for(FichaAluno ficha: turma.values()) {
      cods.add(ficha.getNumero());
  }
  return cods;
}
```

O método que insere uma nova ficha de aluno na turma vai consistir na inserção da chave e do respectivo valor no HashMap. A chave é o *número do aluno* e o valor a respectiva *Ficha do aluno*. O método vai usar como CHAVE, por razões de coerência, o próprio número de aluno existente na ficha a ser inserida. Tal poderá de momento parecer algo redundante, mas, tal como veremos posteriormente, esta pequena redundância trar-nos-á de futuro inúmeras vantagens. Assim, dada uma FichaAluno como parâmetro, dela vamos consultar o número do respectivo aluno através de `getNumero()` e, para que não haja partilha de referencias, realizamos um `clone` de tal ficha usando o método clone definido na classe FichaAluno fazendo o *casting* de `Object` para `FichaAluno`. Através do método `put(K,V)` inserimos então no `HashMap` tal número de aluno como chave e associamo-lo ao clone da ficha parâmetro, cf. o código.

```
/* Insere um novo aluno na turma. Existe a garantia
    prévia da existência de tal número de aluno
*/
public void insereAluno(FichaAluno ficha) {
       turma.put(ficha.getNumero(), ficha.clone());
}
```

Tal como vimos anteriormente, este código apenas deverá ser invocado se se verificar que este aluno ainda não se encontra inscrito, o que deverá ser feito pela invocação prévia do método `existeAluno(String numAluno).`

Este método consiste em determinar se o número do aluno pertence ou não ao conjunto das chaves do *hashMap*, o que se codifica de forma simples da forma

```
/** Verificar se um aluno está inscrito */
public boolean existeAluno(String numAluno) {
    return turma.keySet().contains(numAluno);
}
```

onde `keyset()` devolve um Set<E>, isto é, uma qualquer implementação JAVA5 de um conjunto de elementos de tipo `E` (designadamente HashSet<E> ou TreeSet<E>). De momento, o que interessa sabermos é que qualquer conjunto responde à mensagem `contains(e)` indicando se tal elemento é ou não seu membro, neste caso, se pertence ou não ao conjunto das chaves do *hashMap*, tal como pretendíamos.

A operação que remove um aluno da turma dado o seu número, baseia-se no método `remove(chave)` da classe `HashMap<K,V>`, método que recebe por parâmetro uma chave garantidamente existente - logo aplicam-se-lhe as considerações anteriores para

`insereAluno(…)` -, e remove do *hashMap* a associação (o par) criada tendo tal chave como chave de acesso. No nosso caso tal chave será o número do aluno, sendo pois removido o seu número e a sua ficha correspondente, ou seja o par completo.

```
/** Remove da turma o aluno de número dado */
public void removeAluno(String codAluno) {
     turma.remove(codAluno);
}
```

A operação que vai determinar qual a maior média da turma pode ser implementada, no nosso caso, de duas formas possíveis. A primeira implementação, que se apresenta a seguir, consiste em criar a colecção de chaves da turma - os números dos alunos-, realizar a sua iteração com um `for(..)`, usar o método `get(chave)` de *hashmap* para aceder ao respectivo valor, neste caso uma `FichaAluno` e dela consultar a média de tal aluno para realizar a comparação. Ou seja, *fazemos o "varrimento" ou iteração pelo conjunto das chaves e pelas chaves acedemos aos respectivos valores.*

```
/** Maior média da turma (via chaves) */
public double maiorMediaTurma() {
   double maiorMedia = Double.MIN_VALUE;
   FichaAluno ficha;
      for(String numero : turma.keyset()) {
         ficha = turma.get(numero);
         if( ficha.getMedia() > maiorMedia )
           maiorMedia = ficha.getMedia();
 }
 return maiorMedia;
}
```

Foi nossa decisão de projecto que na ficha de cada aluno, ainda que redundantemente, o seu número fosse também aí registado. Assim, tal número não só se encontra no hashmap como chave mas também num dos campos da respectiva ficha. Ora se a classe HashMap possui uma forma de ser iterada pelos seus valores, então não seria necessário no nosso caso (em que o número do aluno está registado no ficha valor) usar as *chaves* para aceder aos *valores*. Vamos pois refazer o código realizando a iteração usando o método *values()*.

```
/** Maior média da turma (via valores) */
      for(FichaAluno ficha : turma.values) {
         if( ficha.getMedia() > maiorMedia )
           maiorMedia = ficha.getMedia(); }
      return maiorMedia;
}
```

A redundância de dados em especial a redundância de colocação da *chave de acesso* junto do registo valor quando se usam `HashMap<K,V>` ou outros *mappings*, acaba por trazer algumas vantagens resultantes da possibilidade de, dada a necessidade de iterarmos pelos valores e termos a chave a tal valor associada acessível no próprio valor, termos de imediato toda a informação disponível. Note-se a propósito que num *mapping* não existe função inversa, ou seja, dado um valor, quando muito poderíamos ter um conjunto de chaves a ele associadas (o que seria uma relação e não uma função). Com isto pretende-se dizer que, se não fossemos redundantes ao colocar o número do aluno na sua ficha, através da ficha nunca poderíamos saber directamente a que aluno ela pertenceria.

```
    /** Cria o conjunto com os códigos dos alunos
        com nota superior à dada como parâmetro
    */
    public HashSet<String> codAlsMedSup(double notaRef) {
     HashSet<String> cods = new HashSet<String>();
     for(FichaAluno ficha: turma.values())
      if( ficha.getMedia() > notaRef )
          cods.add(ficha.getNumero()) ;
     return cods;
  }
```

A operação que devolve a ficha de um aluno dado o seu número baseia-se, naturalmente, no método `get(k)` da classe `HashMap<K,V>`. O resultado do método será `null` se a chave não existir.

```
    /** Devolve a ficha do aluno dado ou null se não existe */
    public FichaAluno daFichaAluno(String numAluno) {
         return turma.get(numAluno);
    }
```

```
  public String toString() {
      StringBuilder s = new StringBuilder();
      s.append("-------  TURMA --------------------\n");
      s.append("NOME : "); s.append(nomeDiscp);
      s.append("\nCODIGO : "); s.append(codigo);
      s.append("------ ALUNOS  --------------------\n");
     for(FichaAluno ficha : turma.values)
         s.append(ficha.toString());
     s.append("-----------------------------------\n");
    return s.toString();
  }
```

Tendo codificados todos os construtores e métodos de instância das duas classes que constituem as classes principais do projecto, resta apenas ter em consideração que tais classes, quer venham a ser completadas em ambiente JDK puro ou em ambiente de suporte BLUEJ, deverão ter os seguintes "templates" de definição no respectivo ficheiro com o seu nome e extensão *.java*:

```
import java.util.*;
import java.io.*;

public class FichaAluno {
  …….

}
```

```
import java.util.*;
import java.io.*;

public class TurmaHash {
  …….

}
```

PROTOTIPAGEM EM BLUEJ

CLASSE DE TESTE PARA BLUEJ

```
import java.util.*;
import java.io.*;

/**
 * Classe de teste que constrói uma pequena turma
 * a partir da criação de 6 Fichas de Aluno que
 * são depois inseridas numa instância da classe TurmaHash
 * que no final é devolvida como resultado da execução
 * do método main().
 *
 * No ambiente BlueJ, a execução do método main() desta
 * classe criará tal instancia de TurmaHash de modo a poder
 * ser de imediato usada.
 *
 * @author F. Mário Martins
 * @version 1.0/01/2005
 */
```

```
public class TestTurmaHash {

   public static TurmaHash main() {

        FichaAluno f1, f2, f3, f4, f5, f6;
        ArrayList<String> discp = new ArrayList<String>();
        discp.add("66223”); discp.add("66220"); discp.add("66112");
        discp.add("66233");
        discp.add("66291");
        f1 = new FichaAluno("1211", "Rita", "66", "2", 15.2, discp);
        f2 = new FichaAluno("1222", "Paula", "66", "2",14.6, discp);
        discp.clear();
        discp.add("66244"); discp.add("66376");
        discp.add("66255"); discp.add("66220");
        discp.add("66346"); discp.add("66377");
        f3 = new FichaAluno("6633", "Joao", "66","3",12.7, discp);
        f4 = new FichaAluno("6644", "Artur", "66", "3",14.2, discp);
        f5 = new FichaAluno("6655", "Ana", "66","3",12.2, discp);
        f6 = new FichaAluno("6666", "Rui", "66", "3",14.8, discp);

        TurmaHash turma1 = new TurmaHash();

        turma1.insereAluno(f1);  turma1.insereAluno(f2);
        turma1.insereAluno(f3);  turma1.insereAluno(f4);
        turma1.insereAluno(f5);  turma1.insereAluno(f6);

        return turma1;
  }

}
```

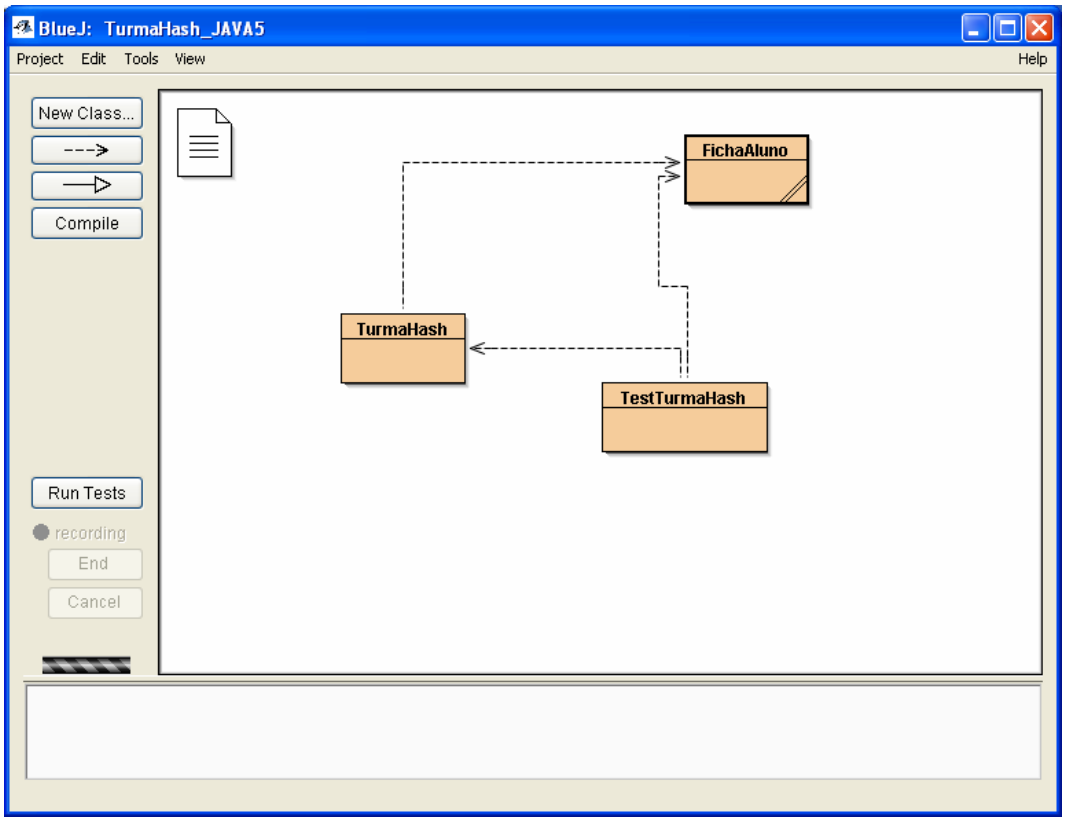


Diagrama de Classes do Projecto 2

Dado que a API das classes `TurmaList` e `TurmaHash` desenvolvidas nos projectos 1 e 2 são iguais, ou seja, a linguagem para o utilizador é a mesma, sendo as implementações distintas, o que para o utilizador – via abstracção – é irrelevante, até porque não as conhece, tal significa que para o programador a *prototipagem* seria também idêntica já que os métodos são os mesmos, os parâmetros são do mesmo tipo, restando-lhe testar, como é óbvio, os resultados do seu código que, agora, trabalha sobre um `HashMap`.