

**Instituto Superior de Engenharia de Lisboa**  
**Engenharia Informática e de Computadores**  
**Secção de Análise de Sinais**  
**Compressão e Codificação de Dados**

---

---

***Framework* didático para a compressão de dados: descrição e  
exemplos de utilização.**

Artur Ferreira {arturj@cc.isel.ipl.pt}

3 Março 2004

Versão 1.0

---

---

## Índice

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Compressão de dados</b>	<b>1</b>
2.1	Modelo teórico . . . . .	2
2.2	Tipos de codificação e formas de compressão . . . . .	3
2.3	Modelos estatísticos e baseados em dicionário . . . . .	4
<b>3</b>	<b>Descrição do framework</b>	<b>4</b>
3.1	Modelos . . . . .	4
3.1.1	Estatístico . . . . .	4
3.1.2	Dicionário . . . . .	5
3.2	Codificadores . . . . .	5
3.3	Transformadas . . . . .	7
3.4	Compressores . . . . .	8
3.5	Armazenamento em suporte físico . . . . .	10
<b>4</b>	<b>Utilização do framework: troços de código</b>	<b>10</b>
4.1	Codificação de Huffman nas formas estática e semi-adaptativa . . . . .	10
4.2	Codificação Aritmética nas formas estática, semi-adaptativa e adaptativa . . . . .	11
4.3	Codificação LZSS . . . . .	11
4.4	Transformadas . . . . .	12
4.4.1	Uso de apenas um bloco transformada . . . . .	13
4.5	Compressores compostos . . . . .	13
4.6	Transformadas compostas e compressores compostos . . . . .	14
4.7	Especificação da frequência de ocorrência . . . . .	15
4.8	Uso de alfabetos restritos . . . . .	15
<b>5</b>	<b>Diagramas UML e aspectos de implementação</b>	<b>16</b>
5.1	Codificadores, decodificadores e modelos . . . . .	16
5.2	Compressores . . . . .	17
5.3	Transformadas . . . . .	19
5.4	Aspectos de implementação . . . . .	20
5.4.1	O método compress . . . . .	20
5.4.2	Construção e escrita do modelo . . . . .	21

# 1 Introdução

Este documento apresenta o *framework* para compressão de dados desenvolvido, com objectivos didáticos, de acordo com os conceitos leccionados no âmbito da disciplina Compressão e Codificação de Dados (CCD), do oitavo semestre da licenciatura em Engenharia Informática e de Computadores (LEIC). Pretende-se, com este documento, habilitar os alunos de CCD, com os conhecimentos necessários à utilização do *framework*, tendo por base os conhecimentos entretanto adquiridos em CCD e provenientes das unidades curriculares da área de programação.

O *framework* está escrito na linguagem de programação C++, segundo o paradigma da programação orientada por objectos (POO), utilizando padrões de desenho (*design patterns*). Os critérios seguidos no desenho e implementação do *framework* são:

- correspondência entre os componentes do modelo teórico da compressão de dados e as entidades que o implementam;
- flexibilidade e facilidade de utilização e expansão;
- legibilidade do código;
- infra-estrutura didática.

Não se pretende ter implementações optimizadas dos algoritmos, relativamente ao tempo de execução ou às taxas de compressão.

O desenvolvimento do *framework* foi iniciado no final de 1998/início de 1999, pelos alunos Artur Ferreira, Paulo Pereira e Nuno Pereira, sob a orientação do prof. David Coutinho. Posteriormente foram realizadas expansões por outros alunos de CCD, nomeadamente José Tavares, Nuno Datia, Nuno Leite, Pedro Fazenda, Luís Sousa e João Viegas. Têm sido também realizadas, por alunos e docentes, expansões para a introdução de outras técnicas de codificação de fonte, codificação de canal e cifra, mantendo os objectivos didáticos, no contexto da LEIC. Estas expansões ainda não estão documentadas.

Este documento apresenta a versão inicial do *framework*, também publicada nas Jornadas de Engenharia de Telecomunicações e Computadores (JETC'99), realizadas no ISEL em Outubro de 1999 [1].

A estrutura do *framework* deve reflectir o conceito da codificação de fonte. O facto de um *framework* ser uma estrutura semi-acabada implica a necessidade de algum conhecimento do problema por parte do utilizador. Estas características são determinantes dado o contexto didático da utilização do *framework*, sendo este preferível à utilização de uma biblioteca.

O presente documento está organizado na forma que a seguir se descreve. Na secção 2, introduz-se o problema da compressão de dados, apresentando o modelo teórico da compressão de dados, os tipos de codificação e as formas de compressão. Na secção 3, descreve-se a constituição interna do *framework* e a relação com as entidades do modelo teórico. A secção 4 apresenta troços de código que constituem compressores, ilustrando a utilização do *framework*. Conclui-se com a secção 5, na qual se apresentam os diagramas UML do *framework* e alguns aspectos de implementação.

Sugere-se, como leitura complementar, o artigo [1].

## 2 Compressão de dados

O objectivo da compressão de dados, concretamente da codificação de fonte, é a representação, com mínima redundância, dos símbolos produzidos por determinada fonte. Procura-se que esta

representação tenha menor dimensão do que a original. A figura 1 ilustra o cenário de utilização da codificação de fonte, cifra e codificação de canal.

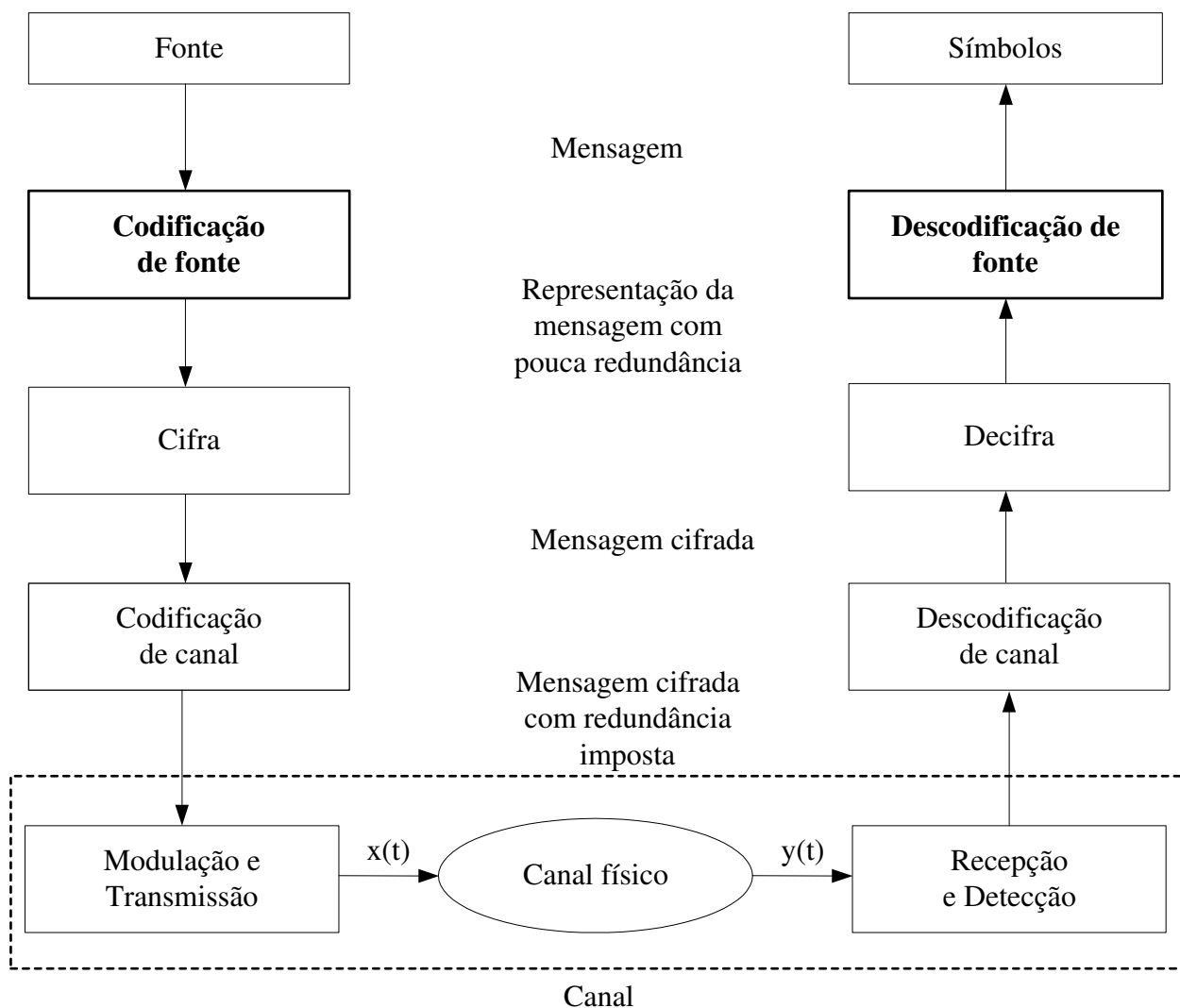


Figura 1: Enquadramento da codificação de fonte, cifra e codificação de canal.

A cifra tem como objectivo, transformar de forma reversível, a informação a transmitir noutra tal que seja imperceptível, excepto para o destinatário da mensagem. A codificação de canal adiciona redundância à mensagem para fazer face aos erros causados pela passagem no canal (de transmissão/armazenamento) ruidoso.

## 2.1 Modelo teórico

O modelo teórico da compressão, sem perda, de dados foi proposto em 1981 por Rissanen e Langdon tendo como característica essencial a separação bem definida entre modelação e codificação. Esta proposta baseia-se no modelo de sistema de comunicação com texto reduzido desenvolvido em 1951 por Shannon. Os dados a comprimir, produzidos por determinada fonte segundo uma distribuição de probabilidades, são constituídos por símbolos que pertencem a determinado alfabeto de dimensão finita. A figura 2 apresenta o modelo teórico da compressão de dados, no qual se identificam as entidades:

- modelo;

- codificador/descodificador;
- compressor/descompressor.

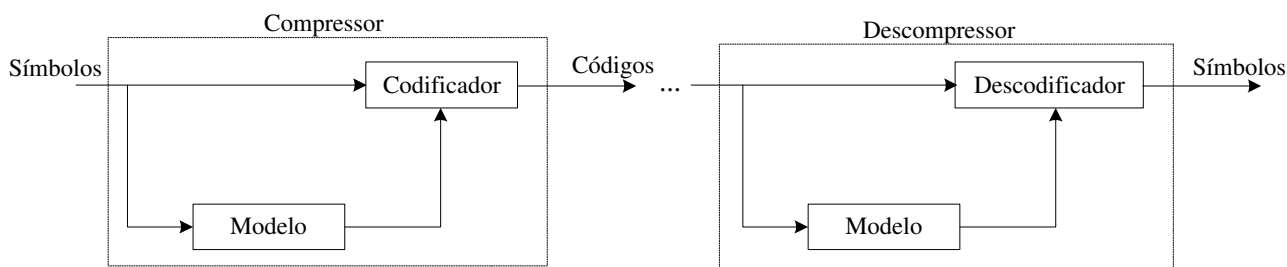


Figura 2: Modelo da compressão de dados. Adaptado de [1].

O modelo detém informação sobre a frequência de ocorrência dos símbolos produzidos pela fonte. O codificador atribui o código correspondente a cada símbolo ou conjunto de símbolos, segundo a informação obtida a partir do modelo. A informação existente no modelo deve ser adequada à estatística dos símbolos da fonte, para que o codificador produza palavras de código com comprimento adequado. Deve existir adaptação entre o modelo e o codificador. De acordo com o Teorema da Codificação de Fonte de Shannon, a adequação da informação existente no modelo determina a taxa de compressão. Designam-se por compressor e descompressor a associação dos pares modelo/codificador e modelo/descodificador, respectivamente. O compressor estabelece o modelo, associa-o ao codificador e trata da actualização e coerência de ambos, ao longo do processamento dos diversos símbolos.

Na descodificação, o descodificador recebe os códigos e converte-os para os símbolos correspondentes em função da informação fornecida pelo modelo. Deve existir coerência entre os modelos usados na codificação e descodificação, de forma a possibilitar a recuperação do texto original. Nalgumas situações, tal implica a transmissão do modelo juntamente com o ficheiro codificado.

## 2.2 Tipos de codificação e formas de compressão

Existem dois tipos de codificação:

- **estatística**, na qual se atribui a cada símbolo um código, com base na sua frequência de ocorrência, tal que símbolos mais frequentes recebem códigos com menor comprimento do que os símbolos menos frequentes;
- **baseada em dicionário** (família de algoritmos Lempel-Ziv) que consiste em representar um conjunto de símbolos através de um código que representa a localização desse conjunto de símbolos no dicionário; entende-se como dicionário um conjunto de frases/caracteres, a partir dos quais se codifica o texto.

Estes tipos de codificação têm características distintas e como tal a sua implementação é intrinsecamente diferente.

Designa-se por forma de compressão, o processo de criação e iniciação do modelo, realizado pelo compressor (descompressor). Consideram-se as seguintes formas de compressão:

- **estática**;
- **semi-adaptativa**;

- **adaptativa.**

A forma de compressão estática caracteriza-se pela criação e utilização de modelo pré-estabelecido (por exemplo, assume-se determinada distribuição para os símbolos). Na compressão semi-adaptativa, estabelece-se o modelo em função da amostra (ficheiro) a codificar, sendo necessário transmitir esse modelo para o descompressor, de modo que a descodificação se possa realizar. Nestas duas formas, o modelo é inalterado ao longo dos processos de compressão e descompressão. Na compressão adaptativa constrói-se um modelo pré-estabelecido que vai sendo actualizado pelo compressor e descompressor ao longo dos processos de compressão e descompressão, respectivamente.

## 2.3 Modelos estatísticos e baseados em dicionário

Dada a distinção vincada entre os dois tipos de codificação, estatística e baseada em dicionário, tem-se que o tipo de informação existente no modelo estatístico é substancialmente diferente da existente no modelo baseado em dicionário. O modelo estatístico é constituído por informação estatística sobre a frequência de ocorrência dos símbolos do alfabeto. No caso da codificação baseada em dicionário, o modelo, geralmente designado de dicionário, é constituído por sequências de símbolos (frases). Neste tipo de codificação, a informação está associada a sequências de símbolos e não a cada símbolo do alfabeto.

Na codificação estatística o modelo pode ser estabelecido de forma genérica para qualquer codificador. No caso da codificação baseada em dicionário, o conteúdo da informação fornecida pelo modelo ao codificador deve estar adequado às características deste último, causando interdependência entre ambos.

## 3 Descrição do framework

De forma a reflectir as características do domínio do problema, a estruturação interna do *framework* deve seguir todos os aspectos anteriormente apontados relativamente ao modelo teórico da compressão de dados.

### 3.1 Modelos

Dada a disparidade entre as características dos modelos estatísticos e baseados em dicionário optou-se por efectuar a implementação separada destes dois tipos de modelo.

#### 3.1.1 Estatístico

Os símbolos a codificar pertencem a determinado alfabeto. A entidade `Alphabet` contém a lista de símbolos produzidos pela fonte, possibilitando assim a implementação dos compressores e codificadores estatísticos de forma genérica, tal que estes manipulem os símbolos independentemente da sua dimensão e da dimensão do alfabeto. A entidade `StatisticalModel` agrega o alfabeto em uso e detém informação sobre a estatística dos símbolos do mesmo, nomeadamente a probabilidade de ocorrência de cada símbolo, armazenada na forma de função cumulativa de distribuição de probabilidade. A combinação destas entidades caracteriza a fonte de símbolos, sendo suficientemente genérica para ser utilizada com qualquer codificador estatístico. A figura 3 ilustra as relações entre as entidades `Alphabet` e `StatisticalModel`. Para permitir ao modelo a navegação sobre o alfabeto, mantendo a independência entre o primeiro e a implementação do segundo, foi utilizado o padrão de desenho `Iterator`.

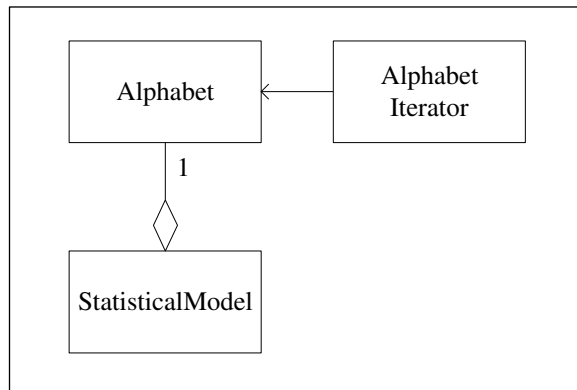


Figura 3: Micro-arquitectura do modelo estatístico. Adaptado de [1].

### 3.1.2 Dicionário

Dadas as características da codificação baseada em dicionário, o alfabeto de símbolos é intrínseco ao modelo. Existe um modelo específico para cada codificador devido à estreita relação entre ambos. O mesmo se verifica para os decodificadores. Estabeleceram-se as interfaces comuns, sob a forma de classes abstractas (`EncoderDictionaryModel` e `DecoderDictionaryModel`) identificadas como ponto de expansão. A figura 4 ilustra a hierarquia de modelos associados à codificação e decodificação, onde se apresentam as entidades `EncoderSlidingWindowModel` e `DecoderSlidingWindowModel` como exemplo de implementações. Estas entidades são compostas por um buffer que armazena a sequência de símbolos (dentro de uma janela pré-definida) processados até ao momento. A entidade `SlidingWindowBuffer` é responsável pela implementação do buffer. Sobre esta foi utilizado o padrão de desenho `Iterator`, através da entidade `SlidingWindowBufferIterator` com os objectivos anteriormente referidos.

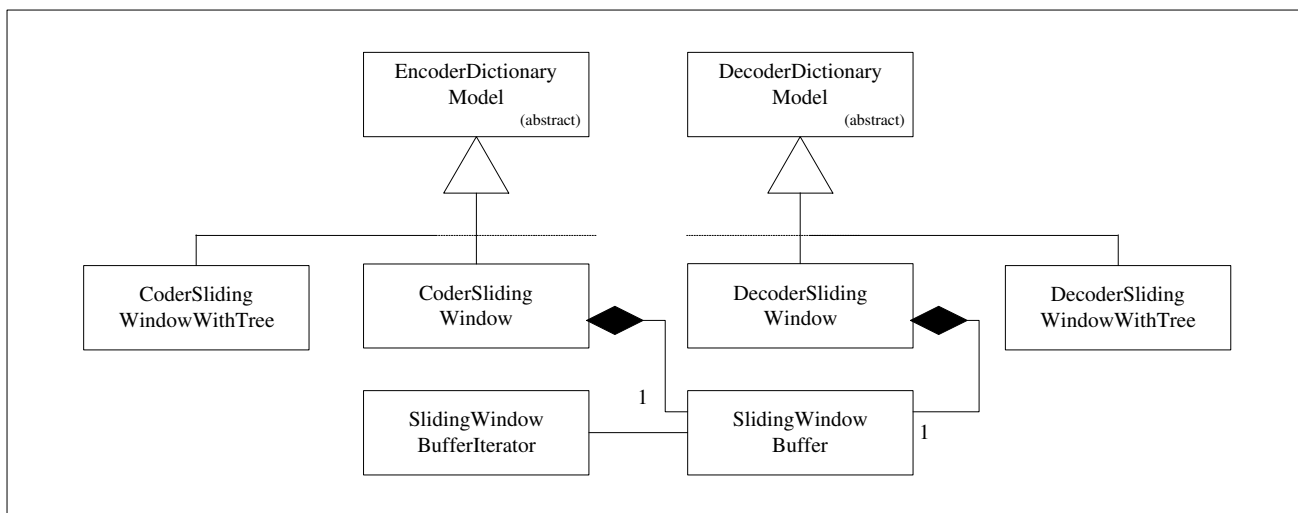


Figura 4: Micro-arquitectura do modelo de dicionário. Adaptado de [1].

## 3.2 Codificadores

A figura 5 representa a estrutura hierárquica dos codificadores implementados e relação com os respectivos modelos, salientando os possíveis pontos de expansão. A entidade `Encoder`

(classe abstracta) é a base de todos os codificadores e estabelece a interface comum. A hierarquia de codificadores reflecte os dois tipos de codificação através de duas classes abstractas (StatisticalEncoder e DictionaryEncoder) que estabelecem a relação com o respectivo tipo de modelo. O processo de codificação tem a seguinte sequência de acções:

- recepção do símbolo ou conjunto de símbolos;
- consulta ao modelo;
- atribuição do código.

Os codificadores estatísticos implementados são: Huffman, Aritmético e CommaCode. No que diz respeito aos codificadores baseados em dicionário existem duas implementações da codificação LZSS (variante do LZ77, da família Lempel-Ziv). Note-se também a existência de codificadores aritméticos adaptativos, designados por AdaptiveArithmeticDecoderSpeed e AdaptiveArithmeticDecoderSize, optimizados em relação ao tempo de execução e à taxa de compressão, respectivamente. Estes codificadores, por cada símbolo codificado, adaptam a informação estatística interna sobre os símbolos do alfabeto.

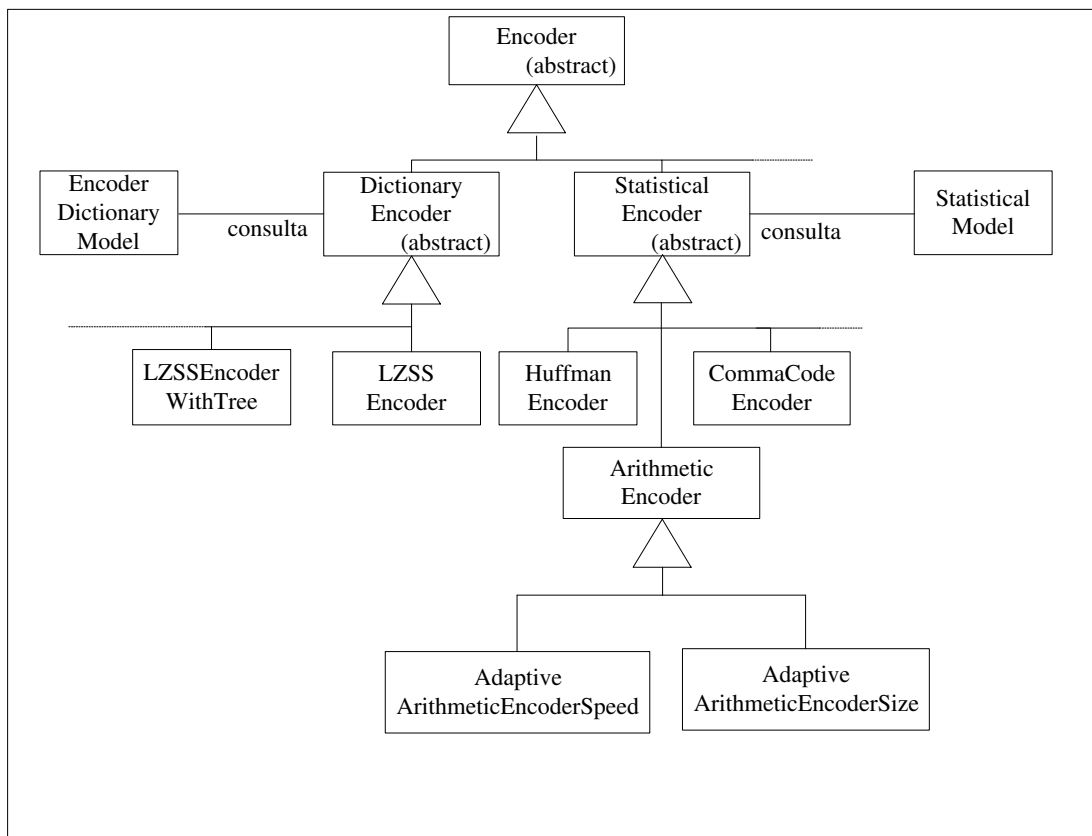


Figura 5: Micro-arquitetura de codificadores. Adaptado de [1].

A hierarquia dos decodificadores é simétrica à dos codificadores, com a excepção da diferenciação do modelo utilizado pelo codificador e decodificador baseado em dicionário. O codificador utiliza a entidade EncoderDictionaryModel e o decodificador utiliza DecoderDictionaryModel. Tal justifica-se pelos diferentes tipos de informação utilizada pelo codificador e pelo decodificador baseado em dicionário, ao contrário do que se passa com os codificadores estatísticos. A figura 6 apresenta a hierarquia de decodificadores.

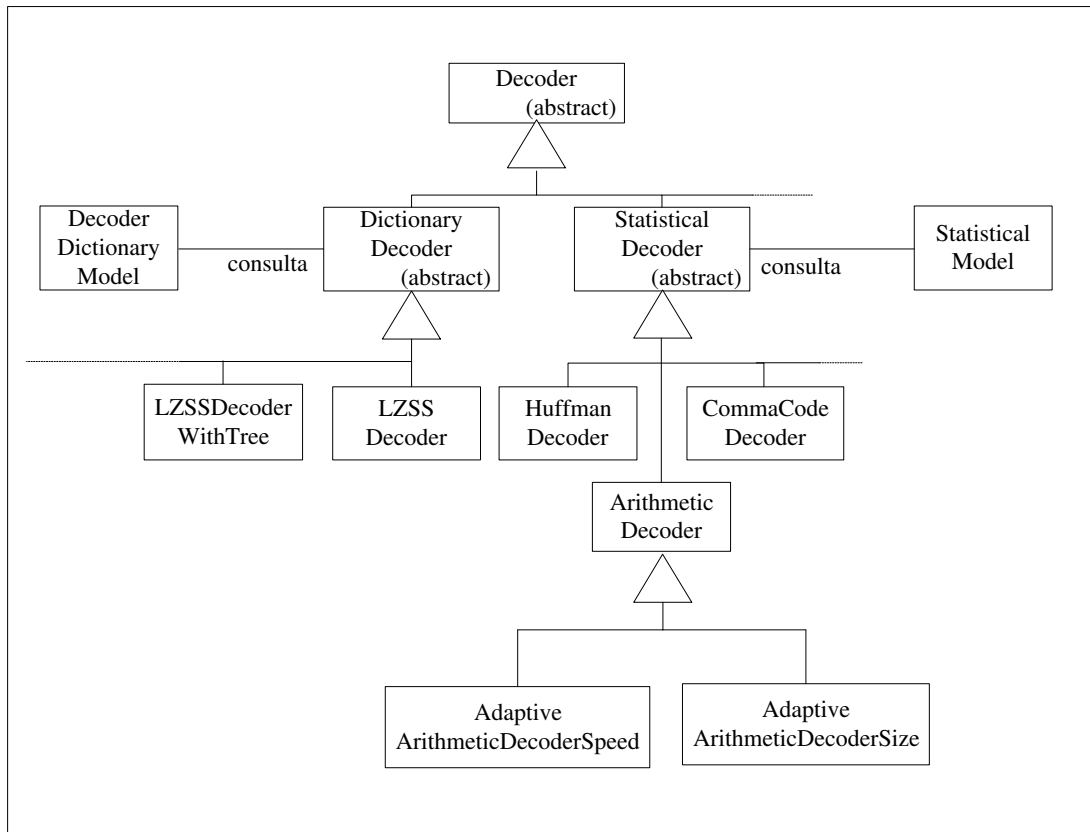


Figura 6: Micro-arquitetura de decodificadores. Adaptado de [1].

### 3.3 Transformadas

Considera-se a expansão do modelo teórico da compressão, apresentado na figura 2, através de transformadas que realizam pré-processamento sobre os símbolos, antes de estes serem submetidos ao processo de codificação. Este pré-processamento consiste na aplicação de uma ou mais transformações, ao nível do símbolo ou do conjunto de símbolos, de forma a evidenciar redundância para que esta possa ser explorada pelo codificador a jusante. A figura 7 ilustra este conceito sobre o modelo teórico da compressão.

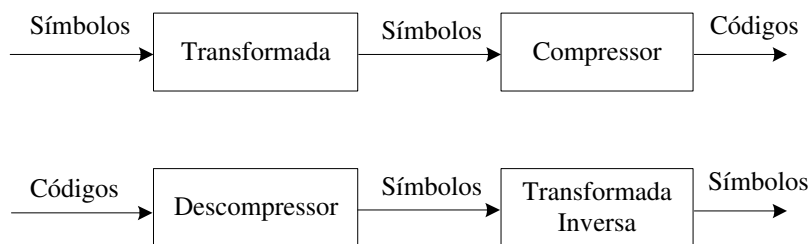


Figura 7: Uso de transformadas sobre o modelo teórico da compressão de dados.

A hierarquia que implementa esta funcionalidade está ilustrada na figura 8. As transformadas implementadas são as seguintes:

- Burrows-Wheeler (BWT - Burrows-Wheeler Transform);
- Move-To-Front (MTF - Move To Front).

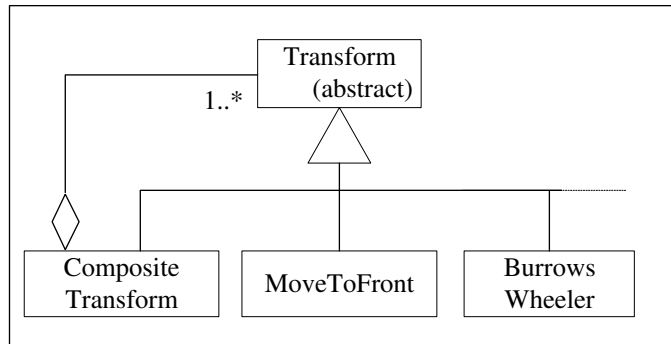


Figura 8: Micro-arquitetura de transformadas. Adaptado de [1].

Sobre a hierarquia de transformadas destaca-se a aplicação do padrão de desenho Composite, que realiza seqüências de transformadas mantendo o tratamento uniforme, dado que implementa a interface definida pela classe base abstracta Transform. A entidade CompositeTransform é responsável pela implementação da seqüência, disponibilizando a interface necessária para adicionar e remover transformadas.

### 3.4 Compressores

As entidades que efectuam a compressão seguem o conceito geral, tal como documentado na figura 2, onde é sugerido que o compressor utiliza determinado codificador em conjunto com o respectivo modelo. Todos os tipos de compressão referidos anteriormente são suportados pela hierarquia de compressores, apresentada na figura 9.

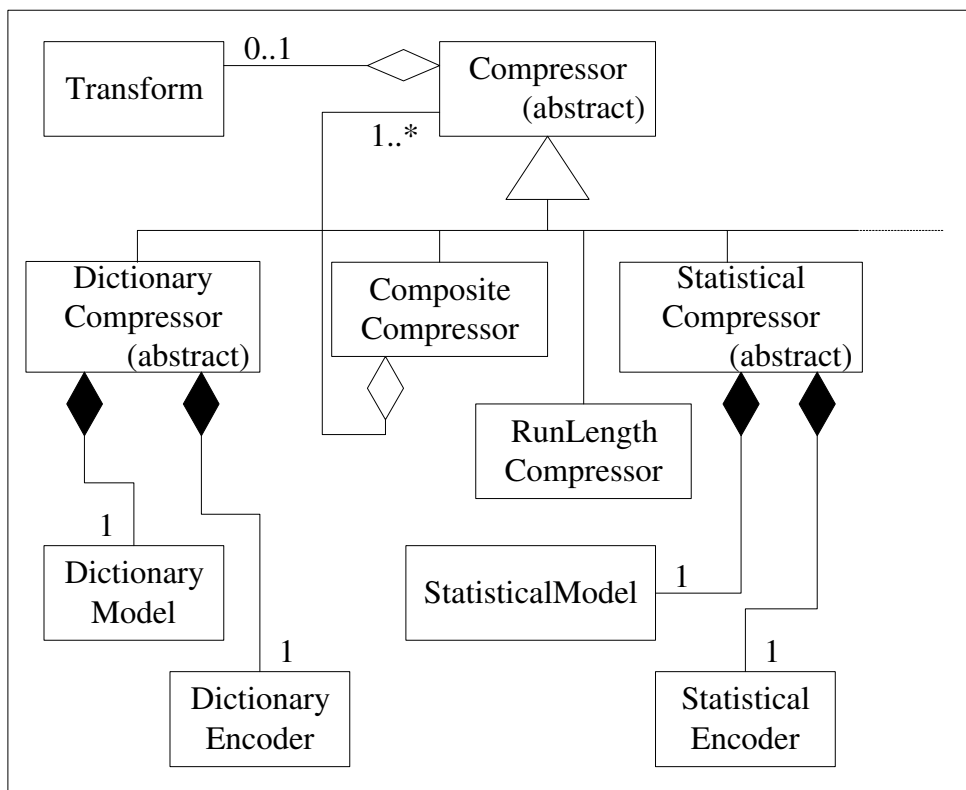


Figura 9: Tipos de compressor e sua relação com modelo e codificador. Adaptado de [1].

A classe abstracta Compressor é a base da hierarquia e para além de definir a interface comum a todos os compressores prevê ainda a utilização de transformadas. A partir desta entidade são estabelecidos os tipos de compressão: estatística, baseada em dicionário e ad hoc. Os dois primeiros são modelados pelas classes abstractas StatisticalCompressor e DictionaryCompressor que garantem a utilização dos pares codificador/modelo correspondentes ao tipo de compressão usado. O tipo de compressão ad-hoc não segue o modelo teórico apresentado. O exemplo mais comum é a codificação de repetições (Run Length Encoding), implementada pela classe RunLengthCompressor.

Prevê-se a utilização de sequências de compressores, através da classe CompositeCompressor, recorrendo ao padrão de desenho Composite, tal como na micro-arquitetura de transformadas. Por sua vez, a classe Compressor também pode utilizar uma composição de transformadas. A partir das entidades StatisticalCompressor e DictionaryCompressor efectua-se a especialização para as três formas de compressão existentes (estática, semi-adaptativa e adaptativa), tal como ilustra a figura 10.

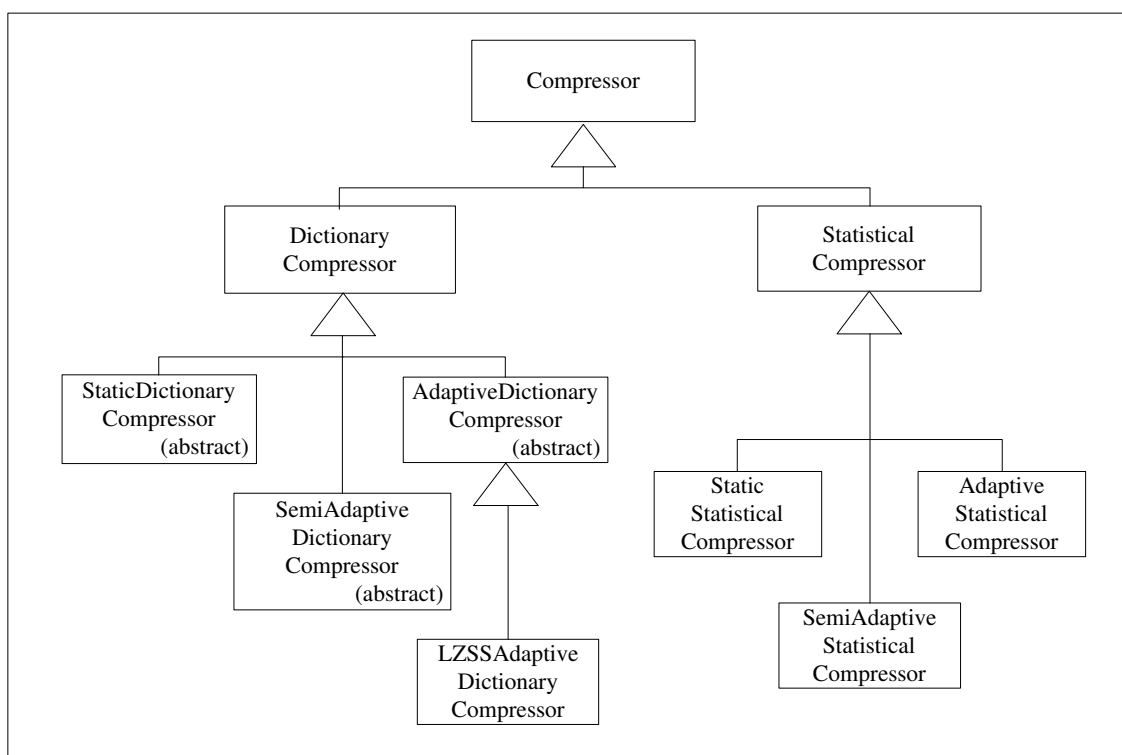


Figura 10: Formas de compressão (estática, semi-adaptativa e adaptativa) para os tipos estatístico e baseado em dicionário. Adaptado de [1].

Dada a interdependência entre codificador e modelo, os compressores baseados em dicionário devem assegurar a coerência na relação entre modelo e codificador para cada implementação de compressor. Dado que o compressor de dicionário está vinculado ao par codificador/modelo que o compõe, este só será definido na altura da criação do compressor em causa. Desta forma justifica-se a necessidade das classes que determinam a forma de compressão serem abstractas.

De forma análoga à relação entre as hierarquias de codificadores e descodificadores, também as hierarquias de compressores e descompressores são simétricas.

### 3.5 Armazenamento em suporte físico

No processo da codificação, os símbolos de entrada são transformados em sequências (códigos) com dimensão múltipla do bit (dígito binário). Foi necessário criar entidades que realizem a leitura e escrita dessas sequências. As entidades responsáveis pela manipulação do acesso ao nível do bit (BitIstream e BitOstream) estão representadas na figura 10. Para além de acrescentar funcionalidade às streams da biblioteca standard do C++, pretende-se restringir a interface disponibilizada pelas mesmas. Para cumprir este objectivo utiliza-se o padrão de desenho Adapter.

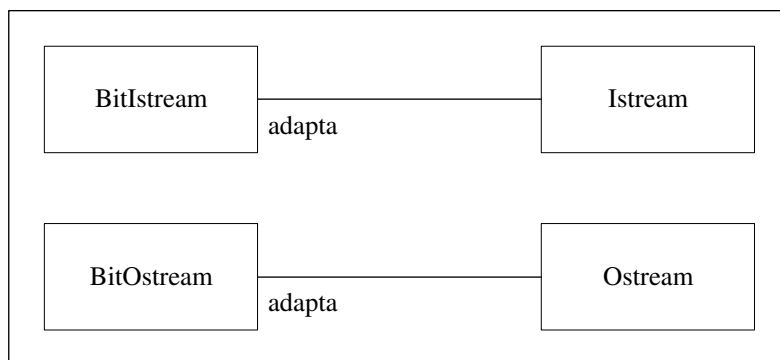


Figura 11: Entidades para manipulação do acesso ao canal ao nível do dígito binário. Adaptado de [1].

## 4 Utilização do framework: troços de código

Nesta secção exemplifica-se a utilização do framework, com diversos compressores, codificadores e modelos.

### 4.1 Codificação de Huffman nas formas estática e semi-adaptativa

Para utilizar o codificador de Huffman, instancia-se um objecto HuffmanEncoder e define-se o alfabeto de símbolos considerado pelo codificador (objecto Alphabet). Por omissão este é constituído pelos 256 símbolos da tabela ASCII.

Para definir a forma de compressão estática (modelo fixo, assumindo equiprobabilidade dos símbolos do alfabeto), instancia-se um compressor do tipo StaticStatisticalCompressor. O modelo (objecto StatisticalModel) é instanciado pelo StaticStatisticalCompressor.

```
void huffmanCompress( const char* inFile, const char* outFile )
{
    Alphabet alphabet;
    HuffmanEncoder huffmanEncoder;
    StaticStatisticalCompressor huffmanCompressor( alphabet, &huffmanEncoder);
    huffmanCompressor.compress( inFile, outFile );
}
```

Para o processo de decodificação, o código apresenta simetria, relativamente à codificação.

```
void huffmanDecompress( const char* inFile, const char* outFile )
{
```

```

Alphabet alphabet;
HuffmanDecoder huffmanDecoder;
StaticStatisticalDecompressor huffmanDecompressor( alphabet, &huffmanDecoder);
huffmanDecompressor.decompress( inFile, outFile );
}

```

## 4.2 Codificação Aritmética nas formas estática, semi-adaptativa e adaptativa

Considere-se agora a codificação aritmética com modelo estático. Relativamente à codificação de Huffman, com modelo estático, a diferença está na utilização de codificador do tipo ArithmeticEncoder, em vez de HuffmanEncoder.

```

void arithCompress( const char* inFile, const char* outFile )
{
Alphabet alphabet;
ArithmeticEncoder arithEncoder;
StaticStatisticalCompressor arithCompressor( alphabet, &arithEncoder);
arithCompressor.compress( inFile, outFile );
}

```

Em relação ao exemplo anterior, passando a considerar a forma de compressão semi-adaptativa, tem-se apenas alteração no objecto Compressor, o qual passa a ser do tipo SemiAdaptiveStatisticalCompressor.

```

void arithCompress( const char* inFile, const char* outFile )
{
Alphabet alphabet;
ArithmeticEncoder arithEncoder;
SemiAdaptiveStatisticalCompressor arithCompressor( alphabet, &arithEncoder);
arithCompressor.compress( inFile, outFile );
}

```

Considerando agora o uso de um codificador aritmético adaptativo, tem-se o troço de código apresentado abaixo. Tanto o codificador como o compressor são adaptativos.

```

void arithCompress( const char* inFile, const char* outFile )
{
Alphabet alphabet;
AdaptiveArithmeticEncoderSpeed arithEncoder;
AdaptiveStatisticalCompressor arithCompressor( alphabet, &arithEncoder );
arithCompressor.compress( inFile, outFile );
}

```

## 4.3 Codificação LZSS

Na codificação baseada em dicionário, basta instanciar um objecto compressor dicionário, do tipo pretendido. O tipo de codificador está implícito pelo compressor utilizado e o modelo é interno a este. O troço de código seguinte ilustra a codificação e descodificação baseada em dicionário.

```

void lzssCompress( const char* inFile, const char* outFile )
{
  unsigned lookBufferDim = 15;
  LZSSAdaptiveDictionaryCompressor lzCompress( lookBufferDim );
  lzCompress.compress(inFile, outFile );
}

void lzssDecompress( const char* inFile, const char* outFile )
{
  unsigned lookBufferDim = 15;
  LZSSAdaptiveDictionaryDecompressor lzDecompress( lookBufferDim );
  lzDecompress.decompress(inFile, outFile );
}

```

#### 4.4 Transformadas

Os compressores admitem o uso de transformadas (eventualmente compostas também) como pré-processamento para a codificação. Na instanciação do compressor, indica-se a transformada a utilizar. No troço de código seguinte, a transformada de Burrows-Wheeler é utilizada como pré-processamento para o codificador aritmético adaptativo (neste último, utiliza-se a versão otimizada em tempo de execução), tal como ilustrado na figura 12.

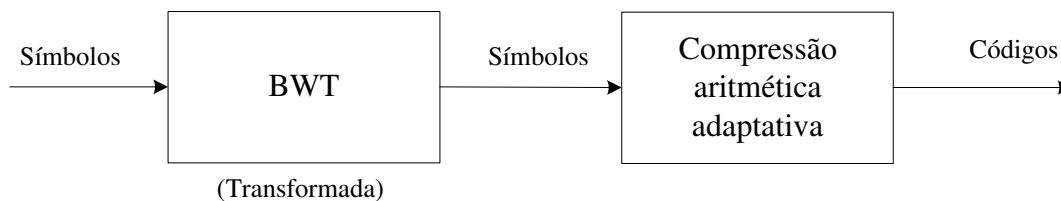


Figura 12: Uso da BWT como pré-processamento para a codificação.

```

void compressWithTransform( const char* inFile, const char* outFile )
{
  BurrowsWheeler bwt;

  Alphabet alphabet;
  AdaptiveArithmeticEncoderSpeed arithEncoder;
  AdaptiveStatisticalCompressor comp ( alphabet, &arithEncoder, 0, &bwt );
  comp.compress( inFile, outFile );
}

```

Na descodificação o código é idêntico ao utilizado na codificação.

```

void decompressWithTransform( const char* inFile, const char* outFile )
{
  BurrowsWheeler bwt;

  Alphabet alphabet;
  AdaptiveArithmeticDecoderSpeed arithDecoder;
  AdaptiveStatisticalDecompressor decomp ( alphabet, &arithDecoder, 0, &bwt );
}

```

```
decomp.decompress( inFile, outFile );
}
```

#### 4.4.1 Uso de apenas um bloco transformada

Para realizar apenas uma transformada ou transformada inversa, utilizam-se os métodos `transform` e `inverseTransform`, respectivamente. Apresenta-se um exemplo com MTF (com a BWT é idêntico).

```
void mtfOnly( const char* inFile, const char* outFile )
{
MoveToFront mtf;
mtf.transform( inFile, outFile );
}
```

```
void unmtfOnly( const char* inFile, const char* outFile )
{
MoveToFront mtf;
mtf.inverseTransform( inFile, outFile );
}
```

### 4.5 Compressores compostos

Os compressores podem ser associados entre si, criando seqüências de compressores, utilizando a entidade `CompositeCompressor`. O trecho de código seguinte, apresenta o uso de RLE (compressor ad hoc) a preceder a utilização de um compressor aritmético adaptativo (ver figura 13). O método `add` adiciona compressores e forma a seqüência. Para executar a seqüência de compressores, é chamado o método `compress` tal como se realiza para um só compressor.

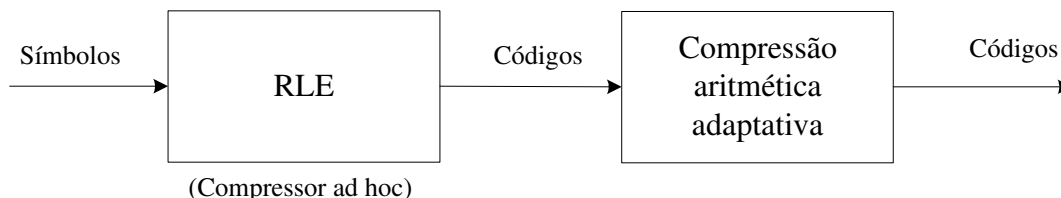


Figura 13: Utilização de compressores compostos.

```
void compositeCompress( const char* inFile, const char* outFile )
{
RunLengthCompressor rlComp;

Alphabet alphabet;
AdaptiveArithmeticEncoderSpeed arithEncoder;
AdaptiveStatisticalCompressor artComp ( alphabet, &arithEncoder );

CompositeCompressor cc;
cc.add( &rlComp );
cc.add( &artComp );
}
```

```
cc.compress( inFile, outFile );
}
```

## 4.6 Transformadas compostas e compressores compostos

A sequência de código abaixo, utiliza transformadas compostas e compressores compostos, ou seja, associa transformadas e codificadores. Realiza a sequência de ações: RLE, BWT, MTF, RLE e compressão aritmética adaptativa, tal como ilustrado na figura 14.

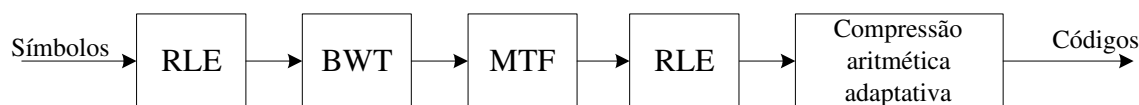


Figura 14: Utilização de transformadas compostas e compressores compostos.

```
void compositeCompress( const char* inFile, const char* outFile )
{
  CompositeTransform ct;
  BurrowsWheeler bwt;
  MoveToFront mtf;
  ct.add( &bwt );
  ct.add( &mtf );

  RunLengthCompressor rlComp;
  RunLengthCompressor rlComp1( &ct );

  Alphabet alphabet;
  AdaptiveArithmeticEncoderSpeed arithEncoder;
  AdaptiveStatisticalCompressor artComp ( alphabet, &arithEncoder );

  CompositeCompressor cc;
  cc.add( &rlComp );
  cc.add( &rlComp1 );
  cc.add( &artComp );

  cc.compress( inFile, outFile );
}
```

O trecho de código abaixo apresenta a sequência inversa da anterior.

```
void compositeDecompress( const char* inFile, const char* outFile )
{
  CompositeTransform ct;
  BurrowsWheeler bwt;
  MoveToFront mtf;
  ct.add( &bwt );
  ct.add( &mtf );

  RunLengthDecompressor rlDecomp;
```

```

RunLengthDecompressor rlDecomp1( &ct );

Alphabet alphabet;
AdaptiveArithmeticDecoderSpeed arithDecoder;
AdaptiveStatisticalDecompressor artDecomp ( alphabet, &arithDecoder );

CompositeDecompressor cc;
cc.add( &artDecomp );
cc.add( &rlDecomp1 );
cc.add( &rlDecomp );

cc.decompress( inFile, outFile );
}

```

## 4.7 Especificação da frequência de ocorrência

Na utilização dos compressores estatísticos estáticos e adaptativos, é possível especificar (assumir) determinada frequência de ocorrência para os 256 símbolos do alfabeto. Para tal, os construtores destas classes recebem um vector com essas frequências de ocorrência, tal como ilustrado no troço de código seguinte, no qual se utiliza codificação aritmética na forma adaptativa.

```

void aarith_model( const char* inFile1, const char* outFile )
{
unsigned long freqs [256];
freqs [0xFF] = 512*(64 - 10);
for (unsigned i=1; i<(sizeof(freqs)/sizeof(freqs[0])); ++i )
    freqs [i] = 512*2;

Alphabet alphabet;
AdaptiveArithmeticEncoderSpeed arithEncoder;
AdaptiveStatisticalCompressor artComp ( alphabet, &arithEncoder, freqs );
artComp.compress( inFile1, outFile );
}

```

## 4.8 Uso de alfabetos restritos

A existência da entidade Alphabet exterior à Compressor, justifica-se para ter alfabetos com número reduzido de símbolos, tal como por exemplo, {A,B} (fonte binária), tal como se apresenta no troço de código seguinte. Recorre-se ao construtor de Alphabet, com três parâmetros: Alphabet(char\* symbolsList[], unsigned symbolSize, unsigned alphabetSize). Esta utilização de alfabetos de dimensão reduzida tem a restrição dos símbolos terem códigos ASCII consecutivos; por exemplo, a sequência {A,C} não é admissível. A imposição desta restrição facilita a escrita do código, mantendo os objectivos didáticos.

```

void restrictedAlphabet( const char* inFile1, const char* outFile )
{
char* symbols = new char[2];
char** symbolsList = new char*[2];

```

```

symbols[0] = 'A';
symbols[1] = 'B';

symbolsList[0] = &symbols[0];
symbolsList[1] = &symbols[1];

Alphabet alphabet( symbolsList,1,2 );
AdaptiveArithmeticEncoderSpeed arithEncoder;
AdaptiveStatisticalCompressor artComp ( alphabet, &arithEncoder, 0 );
artComp.compress( inFile1, outFile );

delete [] symbolsList;
delete [] symbols;
}

```

Utilizando alfabeto restrito, especifica-se agora a frequência de ocorrência para cada símbolo.

```

void restrictedAlphabetFreqs( const char* inFile1, const char* outFile )
{
char* symbols = new char[2];
char** symbolsList = new char*[2];

symbols[0] = 'A';
symbols[1] = 'B';

symbolsList[0] = &symbols[0];
symbolsList[1] = &symbols[1];

unsigned long freqs [2];
freqs[0] = 15; // 'A'
freqs[1] = 5; // 'B'

Alphabet alphabet( symbolsList,1,2 );
AdaptiveArithmeticEncoderSpeed arithEncoder;
AdaptiveStatisticalCompressor artComp ( alphabet, &arithEncoder, freqs );
artComp.compress( inFile1, outFile );

delete [] symbolsList;
delete [] symbols;
}

```

## 5 Diagramas UML e aspectos de implementação

Nesta secção apresentam-se os diagramas UML que descrevem o *framework* e alguns troços de código elucidativos dos conceitos explorados.

### 5.1 Codificadores, decodificadores e modelos

A figura 15 apresenta a hierarquia de codificadores e respectivos modelos. Note-se a separação entre codificadores estatísticos e baseados em dicionário. No caso da codificação estatística,

verifica-se que o conceito de fonte de símbolos (apresentado na figura 1) é modelado através das entidades Alphabet, AlphabetIterator e StatisticalModel. A entidade StatisticalEncoder utiliza o StatisticalModel e o Alphabet.

A classe abstracta Encoder é

```
class Encoder {
public:
virtual ~Encoder() { }
virtual void initialize(BitOStream& ) {}
virtual void finalize(BitOStream&) {}
virtual int encode(BitOStream& , const unsigned char* , unsigned ) = 0;
}; // classe Encoder.
```

O método encode é virtual puro e define a interface comum de todos os codificadores. A partir desta classe, derivam as classes abstractas StatisticalEncoder e DictionaryEncoder que adicionam informação relativa ao tipo de codificação. Derivando a partir destas duas classes, implementam-se os diversos codificadores.

A classe abstracta Decoder tem a seguinte declaração:

```
class Decoder {

public:
virtual ~Decoder() { }

virtual void initialize(BitIStream&) {}
virtual void finalize(BitIStream&) {}
virtual PtrReferencedObject<AllocatedArrayAdapter<unsigned char> >
    decode(BitIStream&) = 0;
}; // classe Decoder.
```

Tanto o codificador como o descodificador são utilizados no âmbito do compressor e do descompressor, nos métodos compress e decompress, respectivamente.

## 5.2 Compressores

A classe abstracta Compressor está declarada como a seguir se apresenta.

```
class Compressor {

protected:

Transform* transform;

public:

// Constantes com os valores dos resultados
// de sucesso ou insucesso na codificacao de uma stream.
enum { CODIFICATION_OK=0, CODIFICATION_ERROR=1 };

// Construction and destruction.
Compressor(Transform* transform=0) {
```

```

this->transform = transform;
}
virtual ~Compressor() { }

// Compressor interface.
virtual int compress ( std::istream& in, std::ostream& out ) = 0;
virtual int compress ( const char* inputFile, const char* outputFile );

// Transform managing.
virtual int add( Transform* transform ) { this->transform = transform; return 0; }

}; // classe Compressor.

```

A partir desta classe, derivam as classes que realizam compressão estatística e baseada em dicionário (StatisticalCompressor e DictionaryCompressor, respectivamente), nas formas estática, semi-adaptativa e adaptativa. O método `compress` define a interface genérica de todos os compressores. O método `add` existe para adicionar transformadas (eventualmente compostas) como pré-processamento. A figura 16 apresenta a hierarquia de compressores. Note-se a especialização de alguns compressores estatísticos, passando a incluir o codificador. Por exemplo, a entidade `SemiAdaptiveArithmeticCompressor` é um compressor semi-adaptativo que utiliza um codificador aritmético. Face aos compressores de uso genérico, tem a diferença de não necessitar de receber o codificador como parâmetro de criação. A sua utilização é a seguinte:

```

void semiAdaptiveArithCompress( char* inFile, char* outFile )
{
Alphabet alphabet;
SemiAdaptiveArithmeticCompressor arithCompressor( alphabet );
arithCompressor.compress( inFile, outFile );
}

```

Outra especialização implementada é o compressor que opera sobre blocos de símbolos, de forma adaptativa. Este é realizado pela entidade `AdaptiveBlockStatisticalCompressor`, sendo esta uma especialização de `AdaptiveStatisticalCompressor`. A sua utilização é idêntica à de outro compressor:

```

void adaptiveHuffmanCompress( char* inFile, char* outFile )
{
Alphabet alphabet;
HuffmanEncoder huffmanEncoder;
AdaptiveBlockStatisticalCompressor huffmanCompressor( alphabet, &huffmanEncoder );
huffmanCompressor.compress( inFile, outFile );
}

```

A declaração da classe `AdaptiveBlockStatisticalCompressor` é a seguinte:

```

class AdaptiveBlockStatisticalCompressor : public AdaptiveStatisticalCompressor {

unsigned dimBlock;

public:

```

```

AdaptiveBlockStatisticalCompressor( Alphabet& alphabet,
StatisticalEncoder* statisticalEncoder,
unsigned dimBl=1024, unsigned long* freqs=0,
Transform* transform=0 )

    :AdaptiveStatisticalCompressor(alphabet,
    statisticalEncoder, freqs, transform ), dimBlock(dimBl) { }

int compress ( std::istream& in, std::ostream& out );
int compress ( const char* inputFile, const char* outputFile ) {
return AdaptiveStatisticalCompressor::compress(inputFile, outputFile );
}
}; // classe AdaptiveBlockStatisticalCompressor.

```

A hierarquia dos descompressores é simétrica relativamente à dos compressores. Apresenta-se de seguida, a classe base Decompressor.

```

class Decompressor {

protected:

Transform* transform;

public:

// Constantes com os valores dos resultados
// de sucesso ou insucesso na codificacao de uma stream.
enum { DECODIFICATION_OK=0, DECODIFICATION_ERROR=1 };

// Construction and destruction.
Decompressor(Transform* transform=0) {
this->transform = transform;
}
virtual ~Decompressor() { }

// Decompressor interface.
virtual int decompress ( std::istream& in, std::ostream& out ) = 0;
virtual int decompress ( const char* inputFile, const char* outputFile );

// Transform managing.
virtual int add( Transform* transform ) { this->transform = transform; return 0; }

}; // classe Decompressor.

```

### 5.3 Transformadas

A hierarquia de transformadas tem em consideração a utilização de transformadas compostas, utilizadas como pré-processamento para a codificação. A figura 17 apresenta a micro-arquitetura de transformadas. A classe base Transform tem a definição apresentada abaixo. Note-se que suporta a transformação directa e inversa.

```

class Transform {

public:

enum ErrorCode { NO_ERROR, STREAM_ERROR, MEMORY_ALLOCATION_ERROR };

virtual ~Transform(){}

ErrorCode getLastError() { return lastError; }
void clearLastError() { lastError = NO_ERROR; }

virtual int transform(std::istream&, std::ostream&) = 0;
virtual int transform( const char* inputFile, const char* outputFile );

virtual int inverseTransform(std::istream&, std::ostream&) = 0;
virtual int inverseTransform( const char* inputFile, const char* outputFile );

protected:

ErrorCode lastError;

}; // classe Transform.

```

## 5.4 Aspectos de implementação

### 5.4.1 O método compress

Apresenta-se de seguida, um troço de código do método `StatisticalCompressor::compress`, no qual se constata a utilização da interface da classe `Encoder`.

```

int StatisticalCompressor::compress ( std::istream& in, std::ostream& out )
{
//....

// Obter a dimensao do simbolo do alfabeto.
unsigned symbolSize = alphabet.getAlphabetSymbolSize();

// Stream de saída bits onde se coloca o resultado da codificacao.
BitOStream bitOstream = BitOStream( out );

// Instanciar o buffer de leitura com dimensao suficiente.
line = new unsigned char [ symbolSize ];

// Iniciar a codificacao.
statisticalEncoder->initialize( bitOstream );

// Ler a stream de entrada ate' ao seu final.
while( 1 ) {

```

```

// Ler simbolo a simbolo.
inPtr->read( (char*)line, symbolSize );

// Verificar se atingiu o final de ficheiro.
if( inPtr->eof() )
break;

// Codificar o simbolo.
statisticalEncoder->encode( bitOstream, line, 1 );
}

// Finalizar a codificacao.
statisticalEncoder->finalize( bitOstream );

// ...
}

```

#### 5.4.2 Construção e escrita do modelo

A classe `SemiAdaptiveStatisticalCompressor` implementa compressão na forma semi-adaptativa. Antes de chamar o método virtual `compress` (para o compressor em causa), constrói o modelo e escreve-o no ficheiro de saída. O modelo é construído pelo método `buildModel` (o qual tem visibilidade *protected*).

```

int SemiAdaptiveStatisticalCompressor::compress ( std::istream& in,
std::ostream& out )
{
// Construir o modelo interno.
setModel( buildModel( &in ) );

// Escrever o modelo na stream de saída.
saveModel( out );

// Efectuar a codificacao.
return StatisticalCompressor::compress( in, out );
}

```

Este modelo é posteriormente lido pelo descompressor e utilizado pelo decodificador. Note-se que na hierarquia de descompressores o método `buildModel` tem a funcionalidade inversa da que tem na hierarquia de compressores.

```

int SemiAdaptiveStatisticalDecompressor::decompress ( std::istream& in,
std::ostream& out )
{
// Construir o modelo interno a partir dessa mesma stream.
setModel( buildModel( &in ) );

// Efectuar a codificacao.
return StatisticalDecompressor::decompress( in, out );
}

```

## Referências

- [1] A. Ferreira, N. Pereira, P. Pereira, and D. Coutinho. Hierarquia de classes para suporte da compressão de dados. In *Jornadas de Engenharia de Telecomunicações e Computadores (JETC'99)*, Lisboa, Portugal, Outubro 1999.

# Hierarquia de Codificadores: Estatísticos e baseados em dicionário

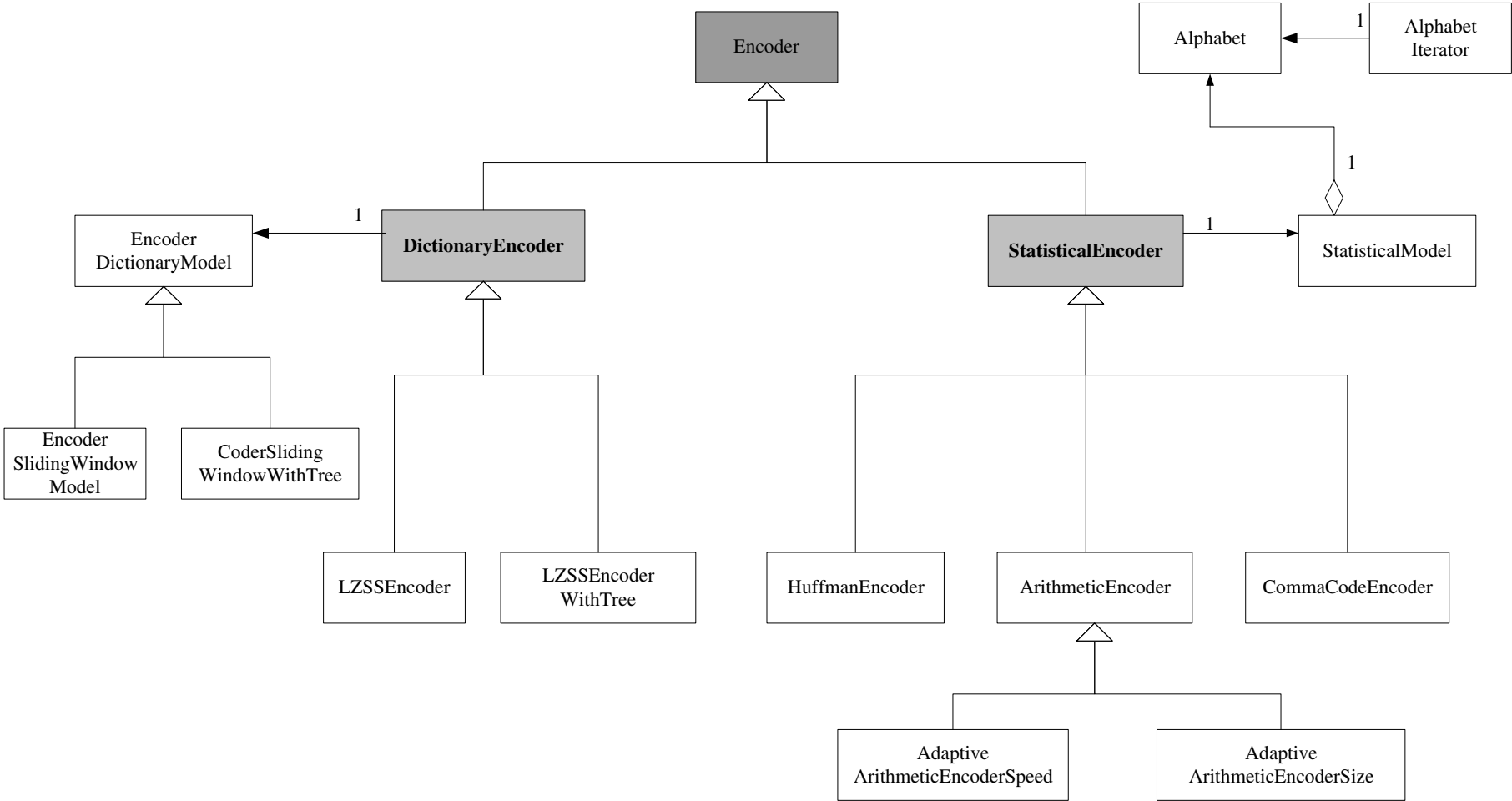


Figura 15: Diagrama dos codificadores.

# Hierarquia de Compressores

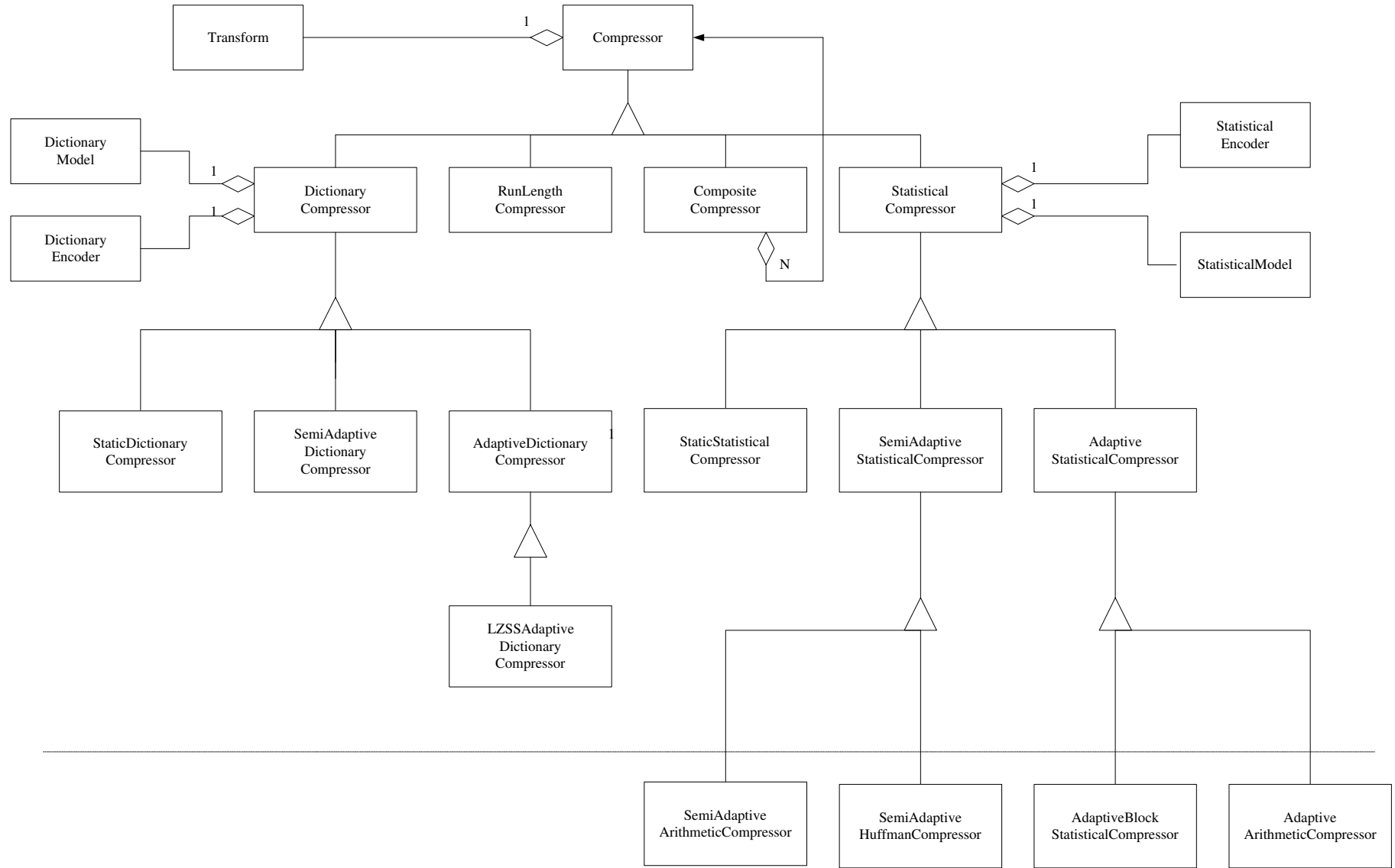


Figura 16: Diagrama dos compressores.

## Hierarquia de Transformadas

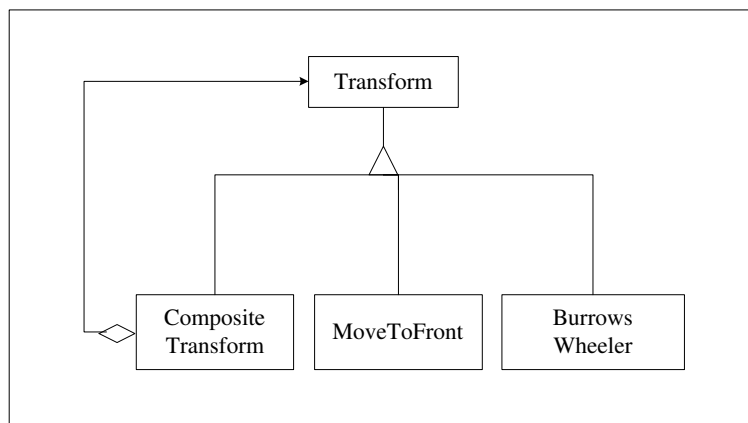


Figura 17: Diagrama das transformadas.