Construção de Software para Cálculo Matemático

Miguel Oliveira e Silva, Francisco Vaz

***Resumo* - Este artigo faz uma abordagem introdutória ao problema da construção de "software" para cálculo matemático, propondo uma técnica de estruturação diferente seguindo o paradigma orientado a objectos. Faz-se um estudo sucinto sobre a estruturação funcional usada "classicamente" nesta área, salientando as suas qualidades e defeitos. É feita uma análise e projecto seguindo uma estruturação orientada a objectos, à qual se segue uma enumeração das exigências colocadas sobre as linguagens de programação que a queiram implementar. No fim refere-se as vantagens e desvantagens desta nova estrutura. Por forma a facilitar a compreensão deste trabalho, faz-se também uma introdução aos paradigmas de construção de "software" aqui referenciados.**

***Abstract* - This paper studies the problem of software construction for mathematical applications, and proposes a different structuring technique following the Object-Oriented paradigm. A short study about the "classic" functional structure used in this area is made, pointing out its qualities and drawbacks. An Object-Oriented analysis and design is performed, and from it is enumerated a set of requirements to impose on programming languages to use in implementation. At the end, the main advantages and disadvantages of this new structure are enumerated. To help to understand this work, one section is dedicated to an introduction to the usual software construction paradigms.**

I. INTRODUÇÃO

O desenvolvimento de *software* para cálculo matemático foi a primeira aplicação prática de relevo dos computadores.

Desde o início, quer pelo cariz funcional do cálculo matemático, quer pela estrutura funcional do *hardware* dos próprios computadores e das linguagens que lhe servem de *interface*, a estruturação deste *software* tem assentado no *paradigma funcional*.

Neste artigo vamos fazer um pequeno estudo sobre uma estruturação seguindo o *paradigma orientado a objectos*.

Todo o projecto desta estruturação é feito pensando em facilitar a extensão do *software* quer para novas funções matemáticas, quer para novos tipos numéricos. Nas bibliotecas e aplicações matemáticas "clássicas" é relativamente fácil (na maioria dos casos) a sua extensão com novas funções. É, no entanto, quase impossível a extensão para novos tipos de números.

Começaremos por definir e caracterizar, resumidamente, alguns paradigmas de programação. De seguida far-se-á uma análise sobre a estrutura geralmente usada no software desenvolvido para cálculo matemático. Depois iniciar-se-á o projecto da estruturação orientada a objectos fazendo uma análise sobre o cálculo matemático. Assente nessa análise, lançar-se-á as bases fundamentais para a estrutura do *software*. No fim concluir-se-á este artigo salientando as principais características desta estruturação.

Este artigo assume um conhecimento mínimo sobre engenharia de *software*, nomeadamente no respeitante aos factores de qualidade, princípios e metodologias de programação, aplicáveis ao *software* (ver [1]–[3]).

Não sendo obrigatório, algum conhecimento (básico) sobre análise e projecto orientado a objectos torna mais fácil a leitura deste artigo (ver [2], [4]–[7]).

Uma abordagem bastante mais completa, incluindo uma implementação e respectiva avaliação prática, pode ser encontrada em [1].

II. PARADIGMAS DE PROGRAMAÇÃO

A forma como se analisa, decompõe, compõe, e estrutura um qualquer problema de programação define uma certa filosofia de programação e é usualmente denominada por *paradigma de programação*. Sendo uma classificação necessáriamente subjectiva, existem diferentes noções de paradigmas de programação.

Nesta secção, iremos apresentar as noções dos paradigmas referidos neste artigo. Desta forma espera-se evitar confusões resultantes de diferentes interpretações dos paradigmas enunciados.

A. Funcional

Stroustrup [8] define-o da seguinte forma:

> "Decidir quais os procedimentos que se quer; usar os melhores algoritmos possíveis."

Neste paradigma o elemento básico de estruturação do *software* é a função ou procedimento.

Esta é, tendo em vista o funcionamento dos computadores, a forma mais "natural" e fácil de desenvolver uma linguagem de programação. A linguagem máquina das unidades de processamento segue o mesmo paradigma. Um programa em linguagem máquina é constituido por sequências de instruções, em que cada instrução define uma acção, ou procedimento, a realizar pelo processador.

Desta forma torna-se natural que as primeiras linguagens de programação de "alto nível"[1] tenham seguido a mesma filosofia, facilitando o desenvolvimento de compiladores e interpretadores (tradutores) para a linguagem máquina de cada processador.

[1] O nível duma linguagem é tanto mais alto quanto menos esforço tenha de ser feito para desenvolver aplicações. É uma medida necessariamente subjectiva e relativa entre linguagens de programação.

A grande fraqueza deste paradigma, resulta da dificuldade que tem em lidar (estruturar) com problemas complexos [1], [6].

B. Encapsulamento de Dados

Ao longo dos anos a importância foi passando dos procedimentos para a organização dos dados. Esta situação resultou da constatação de que os dados eram menos sujeitos a mudanças do que os procedimentos. Donde, se a estrutura básica do *software* assentasse nestes, ficava menos sujeita a alterações, por vezes drásticas, tão comuns na sua manutenção.

Aparece assim um novo paradigma, definido da seguinte forma por Stroustrup [8].

> "Decidir quais os módulos que se quer; partir o programa de forma a que os dados sejam escondidos nos módulos."

Um módulo aqui é identificado com um conjunto de procedimentos e funções (serviços), partilhando dados internos.

Este paradigma representa, em termos de resolução de problemas complexos, uma melhoria significativa em relação ao paradigma anterior. Os princípios e critérios de modularidade [2], [1] são cumpridos quase na totalidade.

É mesmo possível, desde que os módulos sejam tipos de dados, implementar tipos de dados abstractos (*Abstract Data Types*) [2], [1], [8].

C. Orientado a Objectos

Segundo um estudo de Lientz [9], estima-se que cerca de 70% do custo do *software* é dispendido na sua manutenção. Nesta fase, cerca de dois quintos devem-se a extensões e modificações requeridas pelos seus utilizadores.

Coloca-se assim o seguinte problema: como construir *software* por forma a facilitar a sua extensão e adaptação?

Nestes aspectos, o paradigma orientado a objectos revoluciona as metodologias de programação. Fazendo uso de um mecanismo - a *herança* - e das duas técnicas a ele associadas - o *polimorfismo* e a *ligação dinâmica* - este paradigma consegue, de uma forma admirávelmente simples, ir de encontro aos princípios da antecipação de mudanças e da incrementabilidade [2], minimizando o problema anterior.

Este novo paradigma assenta directamente sobre o paradigma de encapsulamento de dados, extendendo a sua usabilidade de duas formas. Primeiro permite, por herança, a definição de novos módulos (*Abstract Data Types*) por extensão e/ou ajustamento (*refinement*) de módulos já existentes. Por fim, permite também, por polimorfismo e ligação dinâmica, a substituição (dinâmica) de módulos sem afectar o código do cliente. Estas características permitem um uso verdadeiramente *abstracto* dos tipos de dados, onde os módulos são usados sem necessidade de qualquer conhecimento prévio da sua implementação, podendo mesmo ser usados módulos sem implementação (em tempo de execução, óbviamente terão de ser substituidos por algum módulo implementado), é necessário sim, a definição das *interfaces* ou comportamento, de cada módulo.

Na terminologia da maioria das linguagens orientadas a objectos, a especificação de cada módulo é designada por *classe*, e as entidades que em tempo de execução instanciam as classes são os *objectos*.

A herança deve ser vista como um método de *classificação*, estabelecendo uma relação *é-um* (*is-a*) entre a classe filha e a(s) classe(s) progenitora(s).

C.1 Covariância

Havendo a possibilidade de redefinir serviços herdados de uma classe progenitora, que liberdade dar aos tipos dos seus parâmetros de entrada e, se existir, ao tipo da entidade de retorno?[2]

Das várias soluções possíveis para este problema iremos referir apenas duas delas, por serem, práticamente, as únicas usadas.

A primeira, consiste em permitir que os tipos da assinatura dos serviços redefinidos possam ser alterados *contra* o sentido normal da herança. Esta é a solução conhecida por *contra-variância*.

Esta é a solução preconizada pelo *C++*. Tem a vantagem de facilitar enormemente o trabalho do compilador, pois evita "buracos" no sistema de tipos. Nunca há o risco ao usar a ligação dinâmica, de invocar um serviço com entidades cujos tipos não sejam conformes com os da sua assinatura. A sua desvantagem é ter pouca aplicação prática (o autor ainda não conhece nenhum exemplo prático que leve vantagem pela aplicação desta solução).

A segunda solução, consiste em permitir que os tipos da assinatura dos serviços redefinidos possam ser alterados *no sentido* normal da herança. Esta é a solução conhecida por *covariância*.

Esta é a solução usada pelo *EIFFEL*. A vantagem desta solução reside na sua maior proximidade com os problemas reais. Por exemplo, uma situação prática nitidamente covariante, é a seguinte:

> Os herbívoros comem plantas. As vacas *são* herbívoros. A erva *é* uma planta. As vacas comem erva mas não outras plantas.

A desvantagem da covariância reside na complexidade imposta ao compilador, pois esta solução pode gerar "buracos" no sistema de tipos da linguagem.

C.2 O Problema do Encaminhamento Múltiplo

Inerente ao próprio paradigma orientado a objectos é o envio de mensagens[3] de objectos para outros objectos. O problema que aqui se coloca tem a ver com o encaminhamento das mensagens. Já vimos que a herança fornece um mecanismo privelegiado (ligação dinâmica) para que esse encaminhamento se faça em função do tipo do objecto (em tempo de execução) para o qual queremos enviar a mensagem.

Sendo um mecanismo simples, nem sempre resulta numa aplicação simples e directa de alguns problemas reais.

Vejamos, por exemplo, a operação de soma entre dois números

$$a + b.$$

[2] A este conjunto costuma-se chamar *assinatura* do serviço.

[3] Sejam elas implementadas por procedimentos (*EIFFEL* e *C++*), ou explicitamente por mensagens (*SMALLTALK*).

Qual será o objecto ao qual devemos enviar a mensagem de soma? Ao *a*? Ao *b*? Ou a outro objecto qualquer?

Se formos rigorosos e aplicarmos a semântica matemática envolvida na soma de dois números, não pode haver efeitos colaterais (*side effects*) nem em *a* nem em *b* pela aplicação desta operação. Assim sendo a escolha deve recair sobre a terceira alternativa: a mensagem deve ser enviada para um terceiro objecto.

O problema que aqui queremos discutir é sobre qual o tipo de objectos a atribuir a esse terceiro objecto. O de *a*? O de *b*? Ou um outro tipo qualquer?

Se *a* e *b* forem do mesmo tipo a solução é simples: o tipo do terceiro objecto deve ser o mesmo.

Se *a* e *b* forem de tipos diferentes (expressão heterogénea) colocam-se três situações: ou *a* é conforme[4] com *b*; ou *b* é conforme com *a*; ou nenhuma das situações anteriores.

Na terceira hipótese, coloca-se a questão da validade da aplicação da própria operação de soma (que sentido tem somar dois tipos de números não relacionados?). Esta é uma situação que, em princípio, deve ser rejeitada pelo sistema de tipos da linguagem (em tempo de compilação ou em tempo de execução).

Para as outras duas hipóteses, se observarmos com algum cuidado, a solução também é imediata: o tipo do terceiro objecto deve ser o tipo do objecto mais genérico dos dois (o tipo do objecto "pai").

Vejamos um exemplo. Se somarmos um número real com um número inteiro, o resultado tem de ser evidentemente um número real (tipo mais genérico).

Esta solução resulta directamente de relação *is-a* imposta pela herança. O tipo de resultado da operação tem de englobar o tipo dos dois objectos aos quais se aplica a operação, donde tem de ser o tipo mais genérico.

C.3 O Problema das Expressões Heterogéneas

Intimamente ligado ao problema anterior existe o problema da implementação dos serviços elementares com pelo menos um argumento, ao qual é necessário ter acesso directo ao seu estado interno.

Não sendo esta uma situação muito frequente, aparece inevitavelmente na implementação dos vários operadores aritméticos, como sejam o operador da soma e da multiplicação.

Voltando ao exemplo da subsecção anterior, vamos supor que queremos implementar o operador soma entre dois números reais. Como é evidente, o serviço relativo ao operador soma terá de ser um serviço elementar, pois não é possível fazer a soma de dois números reais sem ter acesso à sua representação interna.

Se esta situação não traz nenhum problema quando os dois números corresponderem a objectos que sejam instâncias de uma mesma classe, o mesmo já não se passa se um deles pertencer a uma classe descendente diferente, por exemplo, a uma classe de números inteiros. Nesta situação, como implementar este serviço sem obrigar a que essa implementação tenha conhecimento de todas as classes descendentes da classe dos números reais?

[4] Descendente.

A solução só pode ser uma, tem de ser possível converter o estado interno dos objectos das classes descendentes para o estado interno dessa classe, no caso, para a classe dos números reais. Aqui a solução ideal talvez passasse por um mecanismo de coerção (*casting*) da linguagem de programação, definível para cada classe, aplicado automaticamente pelo compilador sempre que necessário. Infelizmente, nenhuma das linguagens de programação conhecidas pelo autor implementa este mecanismo, nem oferece qualquer solução para este problema (*C++*, *EIFFEL*, *SMALLTALK*, etc.). Nestas linguagens, ou se limita o uso de expressões matemáticas a expressões homogéneas (só com objectos de um só tipo), ou se implementam as classes descendentes mantendo o estado interno da classe pai, alterando-o (sempre que possível) paralelamente ao novo estado da classe filha (pode-se dizer que, nesta situação, a coerção é feita sistematicamente em cada alteração do estado interno da classe descendente).

III. ESTRUTURAÇÃO "CLÁSSICA"

Em geral, dois tipos de soluções têm sido usadas para facilitar o desenvolvimento do *software* para grupos específicos de problemas: *especialização* de linguagens de programação; e o desenvolvimento de *bibliotecas* de *software* em linguagens de programação de aplicação geral, para este domínio de problemas.

A primeira solução, faz com que as entidades manuseáveis da linguagem de programação se aproximem das entidades e termos usados nessa área de aplicação, do que resulta uma maior simplicidade e compreensibilidade do *software*. Esta solução facilita o desenvolvimento de ambientes e ferramentas altamente especializados para a resolução deste tipo de problemas, fazendo com que, em geral, se obtenham tempos de desenvolvimento baixos.

Como exemplos de aplicações seguindo esta solução na área do cálculo matemático temos: o MATLAB e o MATHEMATICA.

Comum à maioria destas aplicações é o facto de assentarem em interpretadores das suas linguagens, do que resulta, em princípio, uma perda do desempenho.

Outro aspecto comum a estas aplicações (pelo menos as mencionadas) é o facto de assentarem numa estrutura funcional, onde os módulos do sistema são funções e dados pré-definidos (reais, complexos, etc.). Daqui resulta de imediato uma coesão forte [2] entre os dados e as funções que operam sobre eles.

Estas características fazem com que este tipo de aplicações sejam ideais para *prototipagem*.

A segunda solução, além de em princípio levar a um melhor desempenho, tem outras vantagens importantes: são mais facilmente integráveis em aplicações maiores (compostas não só por cálculo matemático) e, em princípio, são mais facilmente extensíveis para novos problemas não previstos nos sistemas especializados.

Como exemplo de uma biblioteca seguindo esta solução temos a *NAG*, desenvolvida para a linguagem *FORTRAN*. Inúmeras outras bibliotecas existem para outras linguagens como sejam o *C* e o *C++*. A *NAG* surge no entanto como

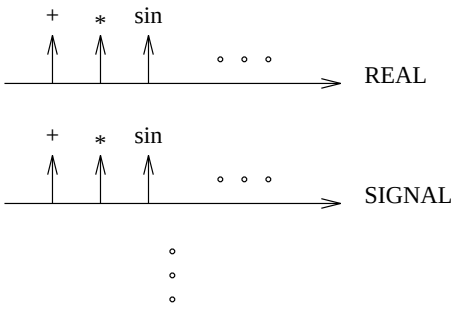

Fig. 1 - Estruturação funcional.

referência essencial pelo seu uso generalizado nesta área.

Estas características fazem com que estes sistemas sejam mais indicados para desenvolvimento de *aplicações em grande escala*, ou quando o desempenho é um factor importante.

Fazendo um apanhado sobre algumas das bibliotecas e aplicações existentes para cálculo matemático, constata-se que a estruturação base que estas usam também assenta no paradigma funcional.

Os problemas deste tipo de estrutura, nesta área, são essencialmente dois. Em primeiro lugar, como já foi referido, há uma coesão forte entre os tipos numéricos e as funções que os operam. Assim, por exemplo, o uso da função *seno* não pode ser visto como o cálculo do seno de um número real, mas sim como o cálculo do seno de uma (única) implementação particular de números reais. Esta característica dificulta enormemente a extensão e a adaptação da biblioteca a novos números.

O segundo problema assenta na não classificação dos vários tipos de números e funções. Por exemplo, um número real *é* um número complexo, assim como um número inteiro *é* um número real, estas relações estabelecem claramente uma hierarquia entre estes tipos de números, no entanto essa hierarquia não é, de forma alguma, implementada nesta estrutura. Se na aparência este problema não parece ser particularmente importante, na prática pode ter implicações profundas. Por exemplo, pelo facto de um número real ser um complexo, é perfeitamente válido fazer a sua soma com um número complexo, obtendo-se sempre como resultado um outro número complexo. Pelo contrário, a soma de um real com um complexo, em geral não tem como resultado um número real. Não havendo uma hierarquização destes tipos numéricos, como garantir estas regras?

Classicamente, estas regras de implementação de expressões heterogéneas, são definidas de uma forma *AD HOC*. Desta forma torna-se muito difícil extender estas regras a novos tipos de números. Outra consequência negativa desta não classificação, é a menor compreensibilidade do *software*, o que pode afectar a sua fiabilidade.

A grande vantagem deste tipo de estrutura (pelo menos nas bibliotecas de *software*) é o seu excelente desempenho. Essa eficiência deve-se, óbviamente, à relação muito directa feita por este *software* com os recursos disponibilizados pelo *hardware* dos computadores.

IV. CÁLCULO MATEMÁTICO

Um dos pilares duma boa estruturação orientada a objectos consiste em assentar a estrutura de classes do *software* nas abstracções essenciais da área do problema que se está a resolver. Impõe-se assim, um estudo e caracterização das entidades e conceitos com que se lida no Cálculo Matemático em geral.

Comum a todo o cálculo numérico é a necessidade de lidar com entidades representando quantidades e entidades que calculam, objectiva e rigorosamente, novas quantidades através de regras, leis e axiomas formais.

As entidades que representam quantidades são designadas pelo conceito abstracto de *número*. As entidades que transformam ou calculam números são designadas por *funções* ou *operadores*. Associadas a estas entidades existe um conjunto de *propriedades* que caracterizam o comportamento dessas entidades (a sua semântica).

A. Números

Comecemos pelos números. O que é que os caracteriza e define? Antes de mais, cada tipo de números implementa uma forma particular de representar "quantidades". Por exemplo, para representar quantidades discretas e unidimensionais usam-se os números inteiros, para quantidades também unidimensionais mas contínuas usam-se os números reais, etc. Além desta característica óbvia, a cada tipo de números pode-se aplicar uma infinidade de operadores e funções, do que resultam novos números (não necessariamente do mesmo tipo). É o caso, por exemplo, dos operadores aritméticos básicos atrás referidos, das funções trignométricas, etc. Da semântica (comportamento) desses operadores ou funções é possível extrair um conjunto de propriedades sempre observáveis na aplicação dos mesmos (comutatividade da soma, etc.).

Associadas a cada tipo de números existe um conjunto de funções ou operadores básicos que permitem caracterizar as propriedades básicas de cada tipo de números. Por exemplo, é esse o caso dos operadores de soma, multiplicação, igualdade, e da relação de ordem dos números reais.

B. Funções

As funções (e operadores), servem para processar números. Apesar de estarem sempre ligados a estes, podem ter uma identidade muito para além de um determinado tipo de número. É o caso, por exemplo, do operador soma, aplicável a uma enorme variedade de tipos de números, como sejam os números inteiros, reais, complexos, etc. Essa identidade refere-se, não só à aplicação da mesma função a vários tipos de números, mas também, no caso geral, pela verificação das mesmas propriedades no seu uso. A comutatividade do operador soma, por exemplo, verifica-se quer seja soma de inteiros, complexos ou mesmo matrizes. Já o caso do operador *maior-do-que*, restringe o tipo de números a que é aplicável: estes têm de ser ordenáveis.

Na maioria dos casos a definição de novas funções é feita à custa de outras já existentes. Por exemplo, a soma de matrizes é definida à custa do produto e da soma dos seus elementos, quer estes sejam números inteiros, reais ou comple-

xos. Neste exemplo, vemos mesmo que o algoritmo para o cálculo dessa operação é o mesmo para todos esses tipos de números.

C. Propriedades

As propriedades servem para disciplinar o uso de funções e números uns com os outros. Funcionam assim para validar o uso da função, ou número, no contexto onde são usados. Por exemplo a propriedade *ordenável* de um determinado tipo de número serve para validar o uso da função *maior-do-que* a entidades (ou instâncias) nesse tipo de número.

É notória, a similaridade entre o uso das propriedades em matemática e o uso de um sistema de tipos em linguagens de programação. Tal como as propriedades matemáticas, a atribuição de tipos aos objectos duma linguagem orientada a objectos, tem o objectivo de validar a utilização desses objectos no contexto onde são usados. Desta constatação resulta a implementação óbvia de cada propriedade matemática numa linguagem orientada a objectos. Associar cada propriedade matemática a um tipo (classe) de objectos distinto. Desta forma, para atribuir propriedades a cada tipo de números ou função basta fazer com que esses números ou funções herdem das propriedades que os caracterizam.

Infelizmente, embora se possa sempre associar uma classe a cada tipo de propriedade, nem todas as propriedades são facilmente implementáveis por classes. Como exemplos das duas situações temos as propriedades da *ordenabilidade* e da *comutatividade*. A ordenabilidade é facilmente implementável, basta definir para a respectiva classe os serviços associáveis às relações de ordem: maior; menor; igual; menor ou igual; e maior ou igual (estas duas últimas podem ser obtidas das três anteriores). Assim só as classes que herdem da classe da ordenabilidade podem fazer uso desses serviços. O caso da comutatividade é, no entanto, completamente diferente. A validação desta propriedade não se faz pelo uso de serviços distintos, mas sim apenas pela semântica no uso de serviços já existentes. Na prática, implementar esta propriedade tornar-se-ia pesado pois obrigaria, para cada operação de soma efectuada, a efectuar a mesma operação com os argumentos trocados e a verificar se o resultado era igual. Teríamos assim um *overhead* excessivo por cada operação efectuada, o que tornaria o *software* necessariamente bastante ineficiente.

Em rigor, este problema é também transportável para a ordenabilidade (e em geral para qualquer outra propriedade matemática), pois o facto de termos as operações das relações de ordem definidas não nos garante que funcionem adequadamente, conforme o sentido semântico matemático que a elas associamos.

Aqui apresentam-se duas alternativas. Ou se tenta garantir ao máximo a observância estrita de todas as propriedades matemáticas usadas, o que obrigaria à execução de inúmeros testes por cada uso de funções e números. Ou se tenta limitar a verificação das propriedades aos casos que não ponham em risco o desempenho do *software*.

A escolha recai obviamente pela segunda alternativa, pois o objectivo do *software* é o seu uso na prática, e não o garantir a correcção absoluta na utilização das entidades matemáticas.

D. Representação de Números

Um dos problemas graves que se coloca na implementação de números em computadores, é o problema da sua representação ter de ser feita por um número finito de estados. Esta situação leva a que possa haver efeitos colaterais "inesperados" na aplicação de funções a esses números. Esta situação pode gerar perdas de precisão e saturação no processamento de números, afectando a fiabilidade de todo o *software*.

Se pouco ou nada se pode fazer em termos de implementar números sem limites máximos ou mínimos e de precisão infinita, pode-se, no entanto, não obrigar ao uso de uma única representação para cada tipo de números usados. Ou seja, arranjar uma estrutura do *software*, em que se possam implementar novas funções de processamento numérico sem que estas obriguem ao uso de um único tipo de dados.

Esta situação reforça as conclusões do estudo atrás feito sobre as vantagens de haver independência entre os números e as funções que operam sobre eles.

Desta forma pode-se desenvolver destemidamente novas função de cálculo numérico, sem o risco de mais tarde elas terem de ser reimplementadas só pelo facto de a representação dos tipos numéricos usada ser insuficiente (ou mesmo boa demais, limitando o desempenho) para as exigências colocadas sobre o *software* de cálculo desenvolvido.

E. Funções e Operadores Elementares

Este problema da representação de números em computadores, salienta outra questão importante. Cada tipo de números em particular, terá, como é óbvio, de lidar com o tipo de representação usada para o mesmo. Isso só pode ser feito ligando fortemente um certo conjunto básico de operadores e funções a esse tipo de números, por forma a que, por exemplo, a soma de 3 com 5 dê 8 em cada tipo de números inteiros existentes.

Tem assim de haver um conjunto de funções dependentes de cada tipo numérico existente. Essas funções ditas elementares têm de fazer parte da especificação de cada tipo numérico, e é só fazendo uso delas que se pode processar números.

Os critérios de escolha entre fazer com que uma função faça parte, ou não, da ADT[5] de cada tipo numérico prendem-se essencialmente com duas situações: ou a sua implementação depende da representação interna do número, ou por questões de desempenho.

F. Excepções Numéricas

Qual é o comportamento que o *software* deve ter se houver uma divisão por zero, ou se a multiplicação de dois números não poder ser representada nesses tipos numéricos?

Esta é uma situação razoavelmente frequente, que resulta de, por vezes, não ser possível efectuar correctamente um

[5] *Abstract Data Type.*

qualquer cálculo numérico. As excepções matemáticas surgem sempre da impossibilidade de expressar com correcção o resultado numérico de uma qualquer função.

Podem-se identificar duas origens para esta situação. A primeira diz respeito a questões puramente matemáticas, como é o caso de se tentar usar uma função com valores que não pertencem ao seu domínio. A segunda, resulta de limitações na representação de números em computadores, podendo gerar problemas de precisão ou de saturação nos cálculos.

Não prever, ou achar-se irrelevantes, este tipo de situações é de todo inaceitável, pois além de elas serem relativamente frequentes, nem sempre exprimem erros no *software* desenvolvido. O exemplo clássico desta situação é o da inversão de matrizes. Esta operação só é válida se as matrizes não forem singulares (determinante diferente de zero), no entanto, o algoritmo que testa se uma matriz é singular é igual ao que faz a inversão. É assim preferível tentar a inversão e lidar com uma eventual excepção de singularidade *a posteriori*.

Estas duas situações levam também a que não possa haver uma única forma de lidar com excepções. Por vezes deve-se indicar ao programador um erro no *software* (situação normal). Outras vezes deve-se permitir que se efectue um processamento especial qualquer na sua existência. Pior que isso, não compete à própria função onde a excepção se deu, lidar com essa situação, pois uma mesma excepção pode corresponder a causas muito diferentes, podendo o tratamento a dar às mesmas ser completamente diverso. Por exemplo, uma inversão de uma matriz singular pode mostrar, de facto, um erro no *software*, devendo-se nesta situação terminar "suavemente"[6] a aplicação e indicar ao programador a localização do código fonte onde a excepção se deu. Esta mesma situação pode, como já foi referido, resultar de uma tentativa consciente, de inversão da matriz.

Assim, deve competir ao cliente dessa função o tratamento a dar em caso de excepções.

G. Tipos Numéricos Básicos

Fundamental numa qualquer biblioteca matemática é a existência dos tipos numéricos básicos, nomeadamente os números inteiros, reais e complexos.

Já vimos que uma qualquer implementação desses tipos numéricos pode levar a problemas por limitações na sua representação. Vimos também o tipo de estruturação que se deve usar para minorar esse problema. Falemos agora um pouco sobre a essência desses números e como se relacionam entre si.

Os números inteiros servem para representar quantidades discretas.

Os números reais representam quantidades contínuas, e, em geral, são implementados por um de dois tipos de representações: vírgula fixa ou vírgula flutuante.

Como o conjunto dos números reais contém o conjunto dos números inteiros, a biblioteca deve permitir o uso de um número inteiro em substituição de um número real, não devendo, em princípio, permitir a situação inversa.

Os números complexos são definidos à custa de uma quantidade abstracta designada por j, cujo valor é igual a $\sqrt{-1}$, e à custa dos números reais. Para estes números há também duas representações frequentes: cartesiana e polar.

[6] Sem mais eventos catastróficos.

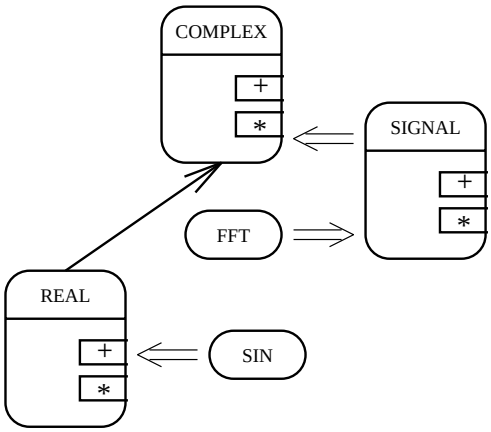

Fig. 2 - Estruturação Orientada a Objectos.

Como os números complexos se implementam à custa dos números reais, em princípio, a biblioteca só precisa de uma implementação deste tipo de números ligada a um tipo real abstracto redefinível para outros quaisquer tipos reais (incluindo os próprios números inteiros).

Estando o conjunto dos números reais contido no conjunto dos números complexos, a biblioteca deve permitir, como acontecia com os números inteiros em relação aos números reais, o uso de números reais em substituição de números complexos. Já o uso de números complexos como reais, não deve ser permitido, já que só sob certas condições (parte imaginária nula) é que um número complexo pode ser visto como real, o que tornaria, pela complexidade envolvida, muito diícil, e mesmo pouco desejável, a implementação de um tal mecanismo.

V. ESTRUTURAÇÃO ORIENTADA A OBJECTOS

Pegando na análise da secção anterior, vamos agora projectar a estrutura básica do *software*, baseada no paradigma orientado a objectos.

Uma estruturação orientada a objectos deve assentar em relações comportamentais entre os vários módulos da biblioteca. Isso é feito recorrendo a *classes* para descrever os vários módulos e ao uso da *herança* para atribuição dos comportamentos associados a cada classe.

Como vimos na secção anterior, as abstracções base do cálculo matemático são: *número*, *função* e *propriedade*. Duas delas são generalizáveis para abstracções mais simples (e uteis): uma *função* pode ser vista como um caso particular de um *sistema*, e um *número* é uma espécie de *dados* (memória).

A abstracção *propriedade* parece ser suficientemente geral. Para se conseguir que as propriedades validem o uso de números e funções faz-se uso do sistema de tipos das linguagens orientadas a objectos, ou seja, por herança.

Por forma a facilitar o armazenamento heterogéneo de quaisquer objectos da biblioteca (e não só), torna-se útil definir uma classe progenitora de todas as classes da biblioteca. Convencionou-se chamar ANY a essa classe.

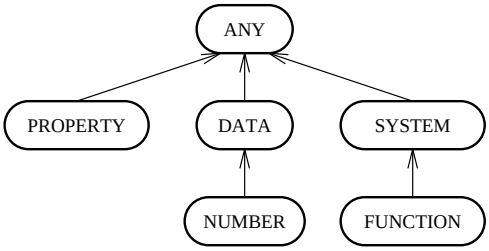

Fig. 3 - Classificação básica da biblioteca

A. *Classificação básica*

Iremos assim assentar toda o *software* em três classes base: PROPERTY, DATA e SYSTEM. Dessas classes base a classe NUMBER herda da classe DATA e a classe FUNCTION herda da classe SYSTEM.

Nesta estruturação surge uma particularidade curiosa e aparentemente paradoxal, há uma classe que abstrai o conceito de função!

Meyer [2] num capítulo sobre técnicas de projecto orientado a objectos, descreve aquilo que designa de *o grande erro* no projecto de classes:

> "O papel fundamental do projecto orientado a objectos é o de construir módulos à volta de tipos de objectos, não de funções. (...)"

Sendo esta afirmação verdadeira na quase totalidade das situações, falha neste exemplo em particular, pois o que se pretende abstrair é precisamente o conceito de função (o próprio Meyer [10] identifica uma situação análoga, a classe COMMAND pertencente ao grupo das classes de *interface* com o utilizador). Um pouco mais à frente nesse livro, ele generaliza o problema dando-lhe, na opinião do autor, o contexto adequado:

> "(..) Este erro é fácil de evitar uma vez estando consciente do risco. O remédio é, como era anteriormente, garantir que cada classe corresponde a uma abstracção de dados com sentido."

Nesta situação, sem dúvida que a abstracção de função é uma abstracção com sentido, pois é uma das entidades fundamentais com que se lida em matemática.

A validação de propriedades, por questões de desempenho e simplicidade, é feita usando o sistema de tipos da linguagem de programação (ou seja através da herança).

B. *Tipos Numéricos Básicos*

Pretende-se projectar classes para os tipos de números básicos, de tal forma que seja possível redefinir qualquer um desses tipos sem afectar o resto da biblioteca, e possibilitando que um número real substitua um complexo, e que um inteiro substitua qualquer um desses dois.

A solução é clara, tornar a classe dos números reais herdeira da classe dos números complexos, e fazer o mesmo para a classe dos números inteiros em relação à classe dos números reais. É necessário também garantir que há redefinição dos operadores da classe dos complexos pelos operadores da classe dos reais, assim como para a classe dos inteiros.

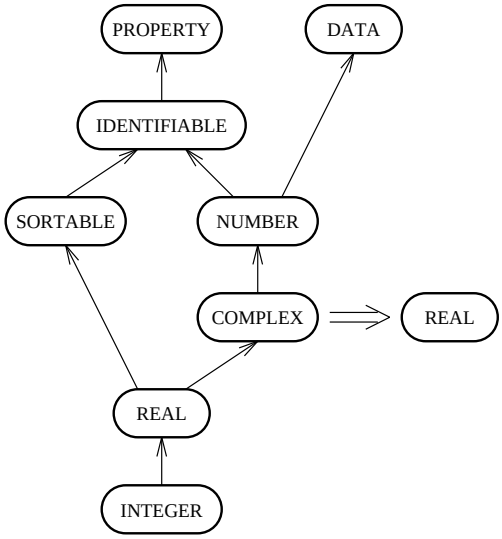

Fig. 4 - Classes numéricas básicas.

A capacidade de substituir os tipos numéricos mais genéricos por tipos numéricos mais especializados, obriga a uma redefinição covariante dos seus operadores comuns. Por exemplo, a soma de dois números complexos é um número complexo, assim como a soma de números reais resulta também num número real, logo este operador tem de ser, obrigatoriamente, redefinido de uma forma covariante.

Apesar deste processo de especialização desde os números complexos até aos números inteiros, todos os serviços dos números complexos continuam a ser aplicáveis, com resultados apropriados aos novos números, nos seus descendentes. Assim continua a ser válida uma invocação ao serviço de conjugação, por exemplo, não sendo, no caso, tomada nenhuma acção uma vez que o conjugado de um número inteiro ou real, é igual ao próprio.

C. *Expressões Aritméticas*

Como resultado das exigências colocadas na estrutura dos tipos numéricos básicos pode acontecer não ser possível usar directamente os tipos numéricos básicos postos à disposição pela linguagem de programação. Essa eventualidade, quando somada à exigência de covariância, pode inviabilizar o uso de expressões aritméticas na sua forma mais natural, ou seja usando directamente os operadores aritméticos.

Se à partida isso pode não parecer muito importante, pois o uso de expressões aritméticas não é mais do que uma forma alternativa de invocar funções numa linguagem de programação, as consequências resultantes da sua não utilização são mais sérias do que isso.

Em primeiro lugar resulta numa menor compreensibilidade do *software*, logo, temos uma menor fiabilidade do *software* que é, evidentemente, o factor de qualidade mais importante a ter em consideração num produto de *software*.

A menor compreensibilidade que esta situação traz, é bem visível nas duas implementações (em pseudo-código) que

em seguida se faz da convolução de dois sinais.

```
(1)-> from k = h.low to h.high loop
        tmp->assign(x[k]);
        tmp->rmult(h[n - k]);
        y[n]->radd(tmp);
      end;

(2)-> from k = h.low to h.high loop
        y[n] = y[n] + x[k] * h[n - k];
      end;
```

Sendo a simplicidade um dos pilares essenciais de uma estruturação orientada a objectos, seria incoerente obrigar os utilizadores do *software* a usarem uma notação tão pouco natural para implementarem expressões matemáticas.

A implementação de expressões aritméticas respeitando as exigências colocadas sobre os tipos numéricos básicos tem também os problemas decorrentes do encaminhamento múltiplo e das expressões heterogéneas.

D. Exigências sobre a Linguagem de Programação

Nesta subsecção vamos agrupar os requesitos necessários ou aconselháveis, da linguagem de programação a usar para implementação de uma biblioteca para cálculo matemático seguindo esta estruturação.

Alguns destes requesitos devem ser fornecidas pela própria linguagem de programação a utilizar, outras podem ser implementadas sobre a mesma (irá depender evidentemente, das facilidades postas à disposição pela linguagem).

- Herança, polimorfismo e ligação dinâmica.
- Herança múltipla (ver [1]).
- Sistema de tipos estático (ver [1]).
- Covariância.
- Programação por contrato (ver [1]).
- Tipos numéricos básicos implementados como classes.
- Relações entre tipos numéricos básicos como descrito na figura 4.

VI. CONCLUSÕES

Como conclusões deste pequeno estudo sobre uma estruturação orientada a objectos de *software* para cálculo matemático, pode-se dizer o seguinte:

- É óbvio que a grande vantagem prática desta estruturação é a sua extensibilidade a novos tipos numéricos. Por acréscimo, congrega também as vantagens inerentes às técnicas orientadas a objectos, embora estas só sobresaiam caso o *software* a construir seja suficientemente complexo.
- A desadequação de todas as linguagens de programação orientadas a objectos existentes. Aqui a que mais se aproxima é a linguagem *EIFFEL*, falhando na hierarquia que impõe aos tipos numéricos básicos, e a forma como lida com o encaminhamento múltiplo e as expressões heterogéneas.
- Por fim, e na sequência do ponto anterior, implementações de uma biblioteca com esta estrutura numa linguagem de programação existente, leva necessariamente a um mau desempenho. A razão principal deve-se ao desconhecimento pelo compilador dos tipos numéricos básicos, impedindo optimizações dos cálculos matemáticos usando esses tipos.

Esta estruturação foi testada numa biblioteca denominada *Calculus Object Oriented Library* (COOL), implementada sobre a linguagem *C++*, e também numa linguagem de utilização dessa biblioteca designada por *COOL Usage Language* (CUL). Ambas as ferramentas foram desenvolvidas na tese de mestrado do autor (ver [1]).

AGRADECIMENTOS

Este trabalho teve o apoio da bolsa de mestrado da JNICT no âmbito do programa CIÊNCIA: BM/1733/91-IA

REFERENCES

[1] Miguel Oliveira e Silva, "Biblioteca de software para cálculo matemático e processamento digital de sinal", Master's thesis, Departamento de Electrónica e Telecomunicações – Universidade de Aveiro, Sept. 1994.

[2] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, N.J., 1988.

[3] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli, *Fundamentals of Software Engineering*, Prentice-Hall, 1991.

[4] Peter Coad and Edward Yourdon, *Object Oriented Analysis*, Prentice-Hall, 1990.

[5] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[6] Grady Booch, *Object Oriented Design with Applications*, The Benjamin/Cummings Series in Ada and Software Engineering. The Benjamin/Cummings Publishing Company, Inc., 1991.

[7] Sally Shlaer and Stephen J. Mellor, *Object-Oriented Systems Analysis – Modeling the World in Data*, Prentice-Hall, 1988.

[8] Bjarne Stroustrup, "What is object-oriented programming?", *IEEE Software*, pp. 10–20, May 1988.

[9] B.P. Lientz and E.B. Swanson, "Software maintenance: A user/management tug of war", *Data Management*, pp. 26–30, Apr. 1979.

[10] Bertrand Meyer, *Reusable Software: The Base Object-Oriented Component Libraries*, The Object-Oriented Series. Prentice-Hall, 1994.