Sebenta de Desenvolvimento de Software

José Coelho, 2007

Sítio:	Universidade Aberta
Unidade curricular:	Desenvolvimento do Software 08/09
Livro:	Sebenta de Desenvolvimento de Software
Impresso por:	José Pedro Coelho
Data:	Terça, 26 Janeiro 2010, 15:50

Índice

1 - Introdução

A presente sebenta é um texto sobre Desenvolvimento de Software, o qual deve ser complementado pela utilização de um ambiente de desenvolvimento avançado, sendo o Visual C++ 2008 Express Edition o ambiente de desenvolvimento adoptado. Os textos desta sebenta são na sua maioria baseados no livro de texto na bibliografia opcional.

O nosso objectivo nesta unidade curricular não é conseguir desenvolver software, é encontrar a melhor maneira de o fazer, de forma a minimizar os custos e maximizar a qualidade do software, e porque não, maximizar o prazer de o desenvolver. Este texto é acompanhado de exemplos concretos, para assim auxiliar o estudante a assimilar o conteúdo dos textos.

A sebenta começa por dar uma introdução ao Desenvolvimento de Software, nomeadamente as principais fases de desenvolvimento e problemas que se levantam em cada fase e alguns conceitos, continuando no segundo capítulo com algumas notas sobre requisitos e características de um bom desenho. No terceiro capítulo é abordado um tema central, programar bem de forma a escrever e ler facilmente código. São enumerados um conjunto de vantagens tanto na edição como na navegação e no debuguer do Visual C++ 2008 Express Edition, e é apresentada uma norma de escrita de código. O quarto capítulo trata da questão de encontrar falhas, enumerando-se diversos tipos de defeitos, várias abordagens de testes unitários, de testes de integração, e de performance, bem como abordagens para a estimação do número de falhas por descobrir no código. O quinto capítulo descreve uma metodologia para efectuar testes empíricos utilizando o software Engine Tester, acompanhado de um laboratório também disponível em vídeo. Finalmente o sexto capítulo aborda a questão de instalação e manutenção, descrevendo sucintamente o que deve ser feito de forma a colocar o sistema em produção, a ser utilizado e mantido com sucesso.

1.1 Fases de Desenvolvimento

O desenvolvimento de software é uma tarefa complexa, em qualquer uma das suas diferentes fases: especificação; desenho; código; teste; instalação; manutenção. É uma tarefa complexa porque por um lado é longa, e por isso envolve normalmente várias pessoas a trabalhar em simultâneo, mas principalmente porque é fácil fazer-se algo errado, algo que não servirá para nada, uma vez que não há forma de se ter a certeza que o que está feito está bem feito.

1.1.1 Especificação e Desenho

Naturalmente que mesmo com estas condicionantes a especificação de uma aplicação de software tem que ser feita, e ser feita bem. Caso a aplicação fique mal especificada, o erro poderá vir a ser detectado na melhor das hipóteses na fase seguinte, no desenho, ou apenas na fase de codificação, e no pior caso após estas fases, provocando que ainda mais trabalho seja deitado fora.

Será fácil fazer uma especificação bem feita? Desde que esta seja consistente, ou seja, que todas as suas especificações possam ser satisfeitas, e seja completa, ou seja, que todas as situações estejam previstas, será certamente uma boa especificação. Necessita ainda que seja uma solução para o problema que o cliente pretende resolver, assumindo que o cliente conhece bem qual o problema que pretende resolver. Para tal há ferramentas que auxiliam este processo, nomeadamente diagramas UML, e que são matéria da unidade curricular Análise de Sistemas. Nesta unidade curricular esta matéria é abordada superficialmente.

Assumindo que não há problemas na especificação, o desenho deve ser feito correctamente para que se

possa começar a escrever código. O desenho deve especificar uma implementação concreta ao que é pedido pelo cliente, formalizado na fase anterior. Caso não implemente o que é pedido, as restantes fases vão ser feitas para nada. Na fase de teste, ou numa pior situação em que o cliente não está envolvido, na fase de instalação, o cliente vem dizer que não era aquilo que pretendia e parte do trabalho é deitado fora. Os diagramas UML continuam aqui a ter um papel importante, sendo esta matéria também da unidade curricular Análise de Sistemas. Nesta unidade curricular esta matéria é abordada superficialmente.

Os materiais da especificação e desenho, bem como dos aspectos introdutórios de Desenvolvimento de Software, são o 1º, 2º e 3º capítulos da sebenta. Como materiais complementares aconselha-se o livro opcional, capítulos 1, 4, 5 e 6.

1.1.2 Código

Chegamos à fase do código, fase esta que confia que as fases anteriores foram feitas com a máxima atenção. Qualquer erro nessas fases significa que o trabalho a iniciar agora poderá não servir para nada. Mas isso não seria problema, não fosse esta fase ter sempre uma certa incerteza associada: será que serei capaz de implementar o algoritmo especificado? Para além disto há que conhecer a sintaxe da linguagem de programação, as bibliotecas disponíveis, a estrutura de dados e o algoritmo a implementar, e o resto dos métodos que serão implementados pelo próprio programador posteriormente, ou por um colega. Ao programar um algoritmo complexo, o programador tem de ter em mente a estrutura de dados e os seus valores a cada passo da execução do algoritmo. É uma tarefa em que o mais fácil é errar, mas felizmente nem sempre há algoritmos complexos, mas mesmo sem algoritmo complexo, o que necessita de estar na mente do programador ao escrever cada linha de código é um volume considerável de informação.

Não há forma de se saber com certeza que o trabalho feito está correcto. Assim, pode-se ir avançando na implementação de outras partes e ter que se voltar para trás ao descobrir que uma das partes não funciona correctamente. Felizmente que a programação orientada por objectos permite modelar código de forma a tornar as diferentes partes do código mais independentes, ou seja, caso um módulo esteja incorrecto, apenas tem que ser revisto o código desse módulo.

Qual é o problema de se errar na fase do código? Não há problema, qualquer programador é suposto errar. Não lhe é pedido que faça código sem erros dado que tal é impossível garantir. Mas isto não significa que o programador possa não dar a sua máxima atenção ao escrever código. A maior parte dos erros leva a erros de compilação, ou seja, erros que provocam que o código não seja aceite pelo compilador como um código válido. Outros erros tornam os resultados completamente disparatados e são apanhados nos testes preliminares que o programador deve fazer para ter alguma segurança que o código está a funcionar bem. Caso o programador não esteja na sua máxima atenção, irá aperceber-se dos resultados disparatados e ao tentar identificar o que está de errado irá inserir erros pensando que está a corrigir os resultados disparatados. Após várias iterações para a frente e para trás, acaba por conseguir uma versão que funciona. Estará essa versão isenta de erros? Certamente que não, apenas passou os testes de utilização normal que o programador já cansado fez. Há outros erros no código, que não se evidenciam agora, apenas em situações menos normais irão ficar activos, mas que podem mesmo assim inviabilizar a usabilidade do sistema.

Este cenário não é nada agradável para o programador, dir-se-ia que ninguém gostaria de estar na pele de um programador. No entanto as grandes dificuldades atraem pessoas que as fazem, pelo desafio e pelo prazer de ver a dificuldade ultrapassada, e a programação não é excepção. Existem hoje em dia ferramentas para auxiliarem este processo, que são os Ambientes de Desenvolvimento Integrados, os quais são de importância capital em grandes projectos e com todo o mérito são matéria desta unidade

curricular. Estas ferramentas não servem para que o programador não necessite saber fazer isto ou aquilo, servem apenas para poupar na memória e na escrita que o programador necessita para escrever cada linha de código. É natural que um programador que conheça bem o ambiente de desenvolvimento que tem, não gastará tempo a ver as bibliotecas que tem disponíveis, já as conhece, tendo uma produtividade bastante mais elevada que outro que se inicia nesse ambiente de desenvolvimento. Pretende-se nesta unidade curricular analisar um ambiente de desenvolvimento em concreto, Visual C++ 2008 Express Edition, de forma a que o estudante conheça as principais funcionalidades que lhe permitem programar de forma mais confortável, sem necessidade de tanta memorização e atenção, resultando num aumento claro da sua produtividade.

Os materiais da fase do Código, é o 4º capítulo da sebenta. Como materiais complementares aconselha-se o livro opcional, capítulo 7.

1.1.3 Teste

O código após implementado e testado pelo seu programador passa agora à fase de testes, para poder ser devidamente verificado por um colega. O objectivo do colega é encontrar falhas no código, e para tal vai efectuar grande volume de testes na tentativa de encontrar falhas. Ao encontrar uma falha tem de a poder repetir para que esta seja reportada, mas não tem que sequer olhar para o código. Os testes são feitos tanto a cada módulo separadamente (testes unitários), como ao conjunto dos módulos (testes de integração). Esta actividade requer alguma imaginação e criatividade de forma a colocar o sistema perante situações novas, dado que repetir situações normais em que o sistema funciona, de nada adianta nos testes. Caso não exista essa imaginação e criatividade, apenas as falhas mais banais serão identificadas, e as restantes ficarão por identificar. Serão alguma vez identificadas? Concerteza que sim, pelo o utilizador do sistema, que tem normalmente sempre mais criatividade e imaginação para colocar o sistema perante situações não testadas.

Este trabalho não é notoriamente algo que deva ser feito por quem acabou de escrever o código. Primeiro pelo cansaço ao código que já deverá ter acumulado após ter escrito o código, e desta forma não ter a dose de imaginação e criatividade necessárias, e segundo porque é uma tarefa que se falhar tem menos trabalho, e em terceiro porque é uma tarefa que se falhar ninguém dá por isso. Naturalmente que se detectar uma falha não deixará de a reportar. No entanto a falta de imaginação e criatividade do autor do código podem ser aqui determinantes. Além disso, se um erro ou outro passar para os utilizadores descobrirem, qual é o problema para o programador? Até dá um certo gozo, dado que o trabalho e a complexidade do código que os programadores fazem normalmente é desconhecido dos utilizadores. Havendo um erro, o utilizador reconhece a sua total incompetência e nem sequer lê a mensagem de erro porque considera à partida que nunca a iria entender. Decididamente, não deverá ser o programador que escreveu o código a fazer os testes.

Esta fase por vezes não é executada. Porquê? O código está escrito, há um responsável pelo desenho, no caso do sistema não corresponder ao que está na especificação assinada pelo cliente, e cada bloco de código foi escrito por um programador, que é o responsável pelo que fez. Para além deste argumento há também outros dois, talvez até de maior peso: a fase de testes aumenta o tempo e custo de desenvolvimento de software. Ainda por cima cria um mal estar na empresa, colegas a fiscalizarem o trabalho de colegas. Todo este tipo de argumentos são aceites apenas por quem não compreende a impossibilidade de se fazer código sem erros, e não considera normal que o código tenha erros. Deverá ser feita a melhor tentativa para identificar e remover os erros enquanto a aplicação está na empresa, caso contrário serão encontrados fora da empresa com maiores custos não só em termos de tempo gasto nos recursos humanos como também em termos da imagem da empresa. Esta é matéria desta unidade curricular, onde será utilizado um software de testes, o EngineTester.

Os materiais da fase de Teste, é o 5º e 6º capítulo da sebenta e os vídeos de demonstração do Engine Tester. Como materiais complementares aconselha-se o livro opcional, capítulos 8 e 9.

1.1.4 Instalação e Manutenção

Após desenvolvido e testado o software, este tem de ser entregue ao seu destinatário final, o cliente. O software será instalado e utilizado pelos utilizadores sem problemas, e funcionará para sempre, gastando apenas energia eléctrica? Infelizmente não será assim. Os utilizadores irão tentar utilizar o software de acordo com o que acham que este deve de ser, e caso o software não faça o que pretendem, consideram que não funciona e deixam de o utilizar. Caso o software sobreviva ao tempo, será reinstalado várias vezes, quer por actualização de hardware, quer por actualização de sistema operativo. Serão encontrados erros, será necessário implementar novas funcionalidades, mas as pessoas que desenvolveram o software podem já não estar a trabalhar na empresa.

Muitas são as situações que levam a que o software deixe de ser utilizado. As últimas fases no desenvolvimento de software são determinantes para a longevidade deste, e portanto a sua rentabilidade. Como lidar com as situações expostas no parágrafo anterior? Simplesmente com o recurso à boa documentação, e nos sistemas mais complexos e essenciais, à alocação permanente de recursos humanos para a manutenção do sistema.

A documentação deve existir a todos os níveis: interna no próprio código; externa ao código mas técnica; documentação para o utilizador. É destinada a poupar o tempo de quem a lê, seja o utilizador para poder utilizar o sistema, seja o programador (o próprio que desenvolveu o código ou um colega) para poder compreender o código e corrigir um bug, ou implementar uma nova funcionalidade.

No entanto a documentação é normalmente muito pobre. Os manuais de utilizador ninguém os lê. Quem lê só vê frases óbvias, pelo que desiste de ler e vai mas é utilizar o programa directamente. Outros utilizadores ao lerem o manual não percebem nada, e por vezes antes mesmo de ler pedem logo a alguém que lhe explique o que o manual quer dizer. A documentação técnica normalmente é escrita de forma a que apenas o próprio programador a compreende, isto se a ler logo após a ter escrito. Perde-se tempo a escrever documentação de utilidade duvidosa, e quem a escreve pensa por vezes que é um trabalho que qualquer um pode fazer, ao contrário da programação, essa sim, uma actividade difícil. Quando o tempo não existe, tanto devido a pressões do chefe que considera já inaceitável o tempo que está a levar para escrever um pequeno bloco de código, como devido ao facto de que apenas os maus programadores levam muito tempo a programar, a documentação é algo que é normalmente aligeirado.

As tendências para que a documentação não exista ou seja de má qualidade devem ser combatidas, caso contrário o software poderá rapidamente deixar de ser utilizado, ou nunca chegar a ser utilizado. Este problema aumenta em projectos que necessitem de equipas de desenvolvimento de software. Os diferentes programadores não vão estar permanentemente em reuniões a falar sobre o código. O grosso da comunicação é feita através da documentação técnica, que deve estar em boas condições para ser facilmente compreendida por quem a lê, seja por um colega no dia seguinte, seja pelo próprio programador daqui a um ano. Nesta unidade curricular esta matéria é abordada superficialmente.

Um projecto é feito por um conjunto de ficheiros que vai sendo editado ao longo do tempo pela equipa de desenvolvimento. Quando se tem várias pessoas a editar os mesmos ficheiros, pode haver situações em que o mesmo ficheiro esteja a ser editado em simultâneo por duas pessoas, levando a que as alterações feitas pela primeira pessoa a gravar sejam perdidas. Várias outras situações indesejadas podem ocorrer, por exemplo, alguém acidentalmente apaga um ou mais ficheiros do projecto. Para que se possa trabalhar em equipa, é necessário um sistema de controlo de versões, que impede que situações destas ocorram.

Os materiais da fase de Instalação e Manutenção, é o 7º capítulo da sebenta. Como materiais complementares aconselha-se o livro opcional, capítulos 10 e 11.

1.2 Conceitos

Nesta secção descrevem-se alguns conceitos soltos de Desenvolvimento de Software.

1.2.1 Engenharia de Software

Engenharia de software é a área que desenha e implementa soluções de um problema, utilizando para tal computadores e linguagens de programação como meios para atingir os fins pretendidos.

Nem todos os problemas passam por soluções informáticas. Há que analisar o problema e subproblemas envolvidos e suas relações, e construir uma solução. Os problemas surgem da necessidade de melhoria dos processos que decorrem numa empresa e na sua maioria as soluções envolvem informática.

Processos que utilizam muito papel podem ser aligeirados pela informatização de parte do processo, simplesmente por mudar o suporte de parte da documentação para ficheiros em disco e numa segunda fase para base de dados.

Um processo que requeira comunicação intensa entre pessoas pode também ser aligeirada com o simples aumento do uso do email em vez do telefone, e numa segunda fase um sistema informático que tenha o processo implementado, passando o processo automaticamente pelas pessoas que o devem analisar.

Em ambos os exemplos, na primeira solução não são necessários engenheiros informáticos, embora se utilize informática, uma vez que apenas há necessidade de se utilizar email e/ou sistema de ficheiros que pode ser configurado por um técnico de informática, e ser dada formação às pessoas que dela necessitem. Na segunda solução é necessário implementar um sistema informático específico de raiz, ou mesmo que se utilize uma solução global, esta necessita sempre de ser desenhada e customizada por engenheiros informáticos.

1.2.2 Erro / Defeito / Falha

Um erro é uma falha humana, relativamente ao que era suposto fazer-se.

Um defeito é uma consequência do erro no código (ou especificação / desenho) que assim irá funcionar de forma defeituosa. Um defeito irá acontecer após um erro, se quem cometer o erro não der por isso.

Um defeito pode ou não ter consequências finais. No caso de ter, irá provocar uma ou mais falhas no sistema, que são uma não conformidade do sistema com os requisitos.

Exemplos:

- Um programador pode esquecer-se de declarar uma variável. Este erro não teria no entanto grandes consequências, já que seria detectado em tempo de compilação, e seria de imediato corrigido.
- Um programador poderá declarar uma variável do tipo real (double) quando deveria ser do tipo inteira (int). Neste caso o erro passaria a defeito, mas sem consequências de maior dado que um double pode guardar todos os valores de inteiros. Se no entanto o código tiver divisões supostamente inteiras, neste caso seriam feitas divisões reais, e o defeito poderia passar a falha, sendo visível para o utilizador.

Os ambiente de desenvolvimento modernos (exemplo do Eclipse e Vistual Studio), têm tendência a evitar erros ao completar parte do código automaticamente, e a levar o programador a detectar imediatamente erros ao colorir parte do código e sublinhar o código suspeito. Assim diminui-se o número de defeitos.

A filosofia do Visual Basic é no sentido de permitir os tipos de erros mais comuns, e não obrigar o programador a grande escrita. Desta forma diminui o número de defeitos, mas permite de igual forma alguns defeitos que o programador detectaria caso fosse obrigado a maior detalhe no código. A não obrigatoriedade de declarar variáveis e o não distinguir minúsculas de maiúsculas, deixam passar defeitos que não passam em outras linguagens.

Os defeitos que passam desapercebidos, alguns nem tanto uma vez que resultam em alertas de compilador muitas vezes ignorados pelo programador, não são tipicamente simples de descobrir por leitura atenta do código. Com o evoluir dos ambientes de desenvolvimento estes defeitos ficam restritos aos que têm baixa frequência de ocorrência, e não existe nada que os detecte senão uma fase de teste feita correctamente, e mesmo assim nada é garantido.

1.2.3 Participantes

Existem basicamente três participantes no desenvolvimento de software: o cliente, o utilizador e o programador. O cliente é quem paga pelo sistema, o utilizador é quem usa o sistema e o programador é quem implementa o sistema. Estes participantes tanto podem ser uma só pessoa como várias pessoas ou uma organização. Nem sempre são entidades distintas, num caso limite são uma só pessoa, no caso de um programador fazer um programa para ele próprio utilizar.

Esta divisão é útil para ver os diversos pontos de vista. O cliente é que tem o problema que necessita de ser resolvido, não vale a pena resolver um outro problema que não seja o do cliente. O utilizador é que vai utilizar o sistema, ou seja a solução, e deve conseguir fazê-lo, não vale a pena fazer o sistema se os utilizadores a que se destina não o conseguirem utilizar. O programador é que tem que desenvolver o sistema, não deve deixar nenhuma das fases que lhe pertencem para o utilizador fazer, como por exemplo o teste do sistema.

1.2.4 Membros da Equipa

No tópico anterior o programador representa todos os membros da equipa de desenvolvimento de software, mas dependendo da fase há diversos participantes:

- Definição de Requisitos - participa o Analista - tem que saber o que o cliente quer;
- Desenho do Sistema - participa o Desenhador, e o Analista - descrição do sistema de acordo com os requisitos;
- Desenho do Programa - participa o Desenhador, e o Programador - descrição detalhada;
- Implementação - participa o Programador;
- Teste de Unidades - participa o Programador, e o Tester - testes de cada unidade implementada;
- Teste de Integração - participa o Tester - testes das diferentes unidades em conjunto;
- Teste do Sistema - participa o Tester - teste de todo o sistema;
- Instalação / Entrega - participa o Instrutor - mostrar/formar utilizadores do sistema;
- Manutenção - participa o Instrutor - corrigir problemas que ocorram.

Há que ter em atenção que a dimensão do projecto pode justificar ou não a existência de determinadas fases, podendo também a mesma pessoa desempenhar diversos papeis. No caso limite, se a equipa de desenvolvimento de software tiver apenas uma pessoa, esta desempenha todos os papeis. É no entanto conveniente que as diversas fases sejam feitas por pessoas diferentes, de forma a que umas possam

verificar o trabalho das outras.

Manda o bom senso que se atribua às pessoas com maior experiência as primeiras fases, já que as fases seguintes estão normalmente dependentes das anteriores. Algo mal feito numa fase inicial pode condicionar muito mais que algo mal feito nas últimas fases.

Mais informação:
http://pt.wikipedia.org/wiki/Engenharia_de_Software

"Software Engineering, theory and practice", second edition, Prentice Hall, Shari Pfleeger, pág. 2-4, 6, 14-15, 25-27, 136, 141, 145

2 - Especificação

Este capítulo aborda superficialmente a fases da Especificação, e o capítulo seguinte Desenho. Estas matérias são abordadas com o devido detalhe na unidade curricular de Análise de Sistemas.

Um requisito é uma funcionalidade ou característica que o sistema tem de possuir. O conjunto de requisitos do sistema define o que o cliente pretende. É essencial que estes sejam verificados pelo cliente para evitar que se implemente um sistema diferente do pretendido.

Um requisito diz-se **funcional** se se refere a uma interacção entre o sistema e o exterior, definindo o procedimento que o sistema deve fazer para uma ou mais situações. Exemplos: registar todas as entradas e saídas de dados num ficheiro; ter um campo de observações para cada registo de uma lista de contactos; devolver o valor da expressão matemática que o utilizador introduzir; apresentar a página seguinte de texto, após o utilizador ter carregado na tecla PgDn.

Um requisito diz-se **não funcional** se impor apenas uma restrição ao sistema, não definindo nenhuma procedimento concreto. Exemplos: impor um tempo de resposta máximo para determinadas situações; forçar um sistema operativo ou um determinado computador.

Os requisitos são **inconsistentes** caso não exista possibilidade de serem todos satisfeitos em simultâneo. Por exemplo, um requisito diz que, para um conjunto de 10 ecrans sucessivos no preenchimento de um formulário, um utilizador poderá levar entre 10 e 30 segundos em cada ecran, enquanto que outro requisito diz que o sistema deverá abortar caso o utilizador não termine o formulário após 60 segundos. Para cumprir o último requisito, os utilizadores teriam de preencher cada ecran em menos de 10 segundos, o que contraria o primeiro requisito.

Os requisitos são **completos** caso todas as situações em que o sistema possa vir a estar tiverem um procedimento definido. Por exemplo, a definição de uma função factorial, que recebe um inteiro natural, pode deixar de fora o caso do factorial de zero. Embora esta função possa não vir a ser chamada com este argumento, pode ser complicado demonstrar que essa situação não irá ocorrer, pelo que estipulando o valor de retorno nessa situação garante que os requisitos ficam completos.

Os requisitos devem ser realistas e verificáveis. Um requisito impossível na tecnologia actual, não pode ser implementado, e um requisito que não se possa verificar, provavelmente nunca ficaria bem implementado devido a não poder ser testado convenientemente.

Para se conseguir uma boa especificação é necessário conseguir comunicar com o cliente. Primeiro conseguir compreender o problema do cliente. Segundo, conseguir transmitir a solução proposta ao cliente. Normalmente a linguagem técnica não é a melhor forma de o fazer, muito embora seja normalmente a mais fácil. A utilização de diagramas UML são aqui de grande utilidade, em complemento a uma lista de requisitos. Embora o cliente possa vir a validar a especificação de requisitos, é provável que não saiba se a solução proposta resolve o seu problema. É da responsabilidade do analista certificar-se que a solução que propõe resolve o problema do cliente.

Por vezes há uma inversão de papeis nesta fase. O cliente pode até nem sequer falar do seu problema e apresentar ao analista o que pretende que se faça, a solução, na sua própria linguagem, sendo afinado pelo analista de forma a fazer sentido. Tal não será aceitável a não ser que o cliente tenha competências para tal. Se não tiver competências o mais provável é que o que pede não será solução do seu problema. Será implementado e haverá seguramente mais iterações do tipo: "é necessário também isto, caso contrário isto tudo não faz sentido". Fica um projecto condenado a custar muito mais, tanto no preço como na paciência, isto apenas para que o analista não ponha em causa as capacidades

informáticas do cliente.

Exemplo: Poker

Vamos colocar um exemplo concreto para utilizar ao longo da sebenta.

Especificação A:

1. *Pretende-se um sistema para simular jogos de poker entre quatro jogadores, com 7 cartas, com 2 cartas fechadas e 5 abertas (ver: http://en.wikipedia.org/wiki/Texas_hold_%27em).*
2. *Pretende-se para cada jogo obter a melhor mão de cada jogador.*

Quem já tenha jogado poker, provavelmente a especificação a cima é suficiente. No entanto há diversos conceitos indefinidos na frase acima, alguns dos quais são de conhecimento geral, outros nem tanto. É sempre conveniente que na especificação seja especificado tudo com clareza, mesmo conceitos do conhecimento geral. Por exemplo, quem nunca tiver jogado poker, poderá não saber qual o baralho que se utiliza, e quem nunca tiver jogado cartas, poderá não saber que um jogo de cartas implica a utilização e um baralho. Os conceitos de cartas fechadas e abertas, não só estão relacionados com o jogo, pelo que requerem conhecer-se o jogo, como pode eventualmente existir mais que uma denominação para o mesmo conceito. Finalmente, nem sequer é referido que mãos existem, nem a sua ordem de valor.

Se tanto o analista como o cliente se entendem através de uma especificação do nível desta, valerá a pena maior formalização? Vale concerteza, já que o risco envolvido aceitando uma especificação deste tipo é muito elevado. O cliente poderá nem ter consciência do que está a pedir, e muito provavelmente o analista também não sabe o que está a aceitar. Para além disso, se houver conflitos esta especificação de nada serve, cada parte irá elaborar a interpretação que lhe interessar.

Esta poderia ter sido a primeira interacção do analista com o cliente, vamos reflectir um pouco sobre a especificação em outra prespectiva. Quem é o cliente? Eventualmente um jogador de poker, ou um casino. Qual é o problema que o jogador de poker quer resolver? Será realmente a simulação de jogos de poker? Provavelmente não, já que não tem ganho directo por ver um jogo simulado, dado que quando estiver a jogar o jogo não será o mesmo. O que o cliente quer é muito provávelmente informação útil que possa utilizar durante um jogo de poker, pelo que esta especificação irá certamente evoluir. Como para obter informação útil num jogo de poker será necessário simular jogos de poker, não vale a pena complicar a especificação com a informação que se pretende extrair através dos jogos simulados.

Outro ponto a ter em atenção na especificação é a introdução de constantes. Neste caso, o jogo é jogado entre 4 jogadores, com mãos de 7 cartas, com 5 cartas abertas, 2 cartas fechadas. É neste problema que o cliente está interessado, no entanto não há grande ganho em utilizar esta informação como constantes, e o próprio cliente facilmente poderá mudar de ideias e espera que a mudança seja simples, pelo que, a não ser que exista uma vantagem óbvia, as constantes devem ser tratadas como parâmetros, tendo naturalmente os valores iniciais que o cliente especificou. Desta forma estamos a aumentar a reusabilidade do código.

Especificação B:

1. *Utilizar um baralho de 52 cartas (4 naipes e 13 números: 2;3;4;5;6;7;8;9;10;V;D;R;A), e baralhar de forma a distribuir cartas de forma aleatória por NJ=4 jogadores (1<NJ<8);*
2. *Distribuir cartas, colocando CA=5 cartas na mesa, e CF=2 cartas em cada um de quatro jogadores (as primeiras CA+NJ*CF=13 cartas do baralho; 0<=CA<=5;*

0<CF<=5; 5<=CA+CF<=7);

3. *Analisar as cartas de cada jogador juntamente com as duas cartas na mesa, e calcular a melhor mão;*
4. *A mão mais alta é a sequência e côr real, em que consiste em 5 cartas todas do mesmo naipe, com os 5 números mais altos (10;V;D;R;A);*
5. *A segunda mão é a sequência e côr, em que consiste em 5 cartas todas do mesmo naipe, com 5 números seguidos;*
6. *A terceira mão é o poker, em que consiste em 4 cartas com o mesmo número;*
7. *A quarta mão é o fullen, em que consiste em 3 cartas com o mesmo número, e outras 2 cartas com o mesmo número;*
8. *A quinta mão é a côr, em que consiste em 5 cartas todas do mesmo naipe;*
9. *A sexta mão é a sequência, em que consistem em 5 cartas com números seguidos;*
10. *A sétima mão é o trio, em que consiste em 3 cartas com o mesmo número;*
11. *A oitava mão são dois pares, em que consiste em 2 cartas com o mesmo número, e outras duas cartas com o mesmo número;*
12. *A nona mão é o par, em que consiste em 2 cartas com o mesmo número;*
13. *A décima mão é nada, quando não existe nenhuma outra mão;*
14. *Se duas mãos forem iguais, a mão com a carta mais alta é a melhor.*

Nesta especificação já não há grandes conceitos omissos, excepto o conceito de naipe e número, que para quem nunca tenha visto um baralho pode ser confuso. É também uma especificação mais longa, pelo que poderá não ser lida com a mesma atenção que a primeira especificação. É sempre conveniente colocar o mais importante primeiro, de forma a aproveitar não só a maior atenção do leitor, como para ser sempre clara a utilidade de cada requisito.

A passagem de constantes para variáveis força a especificar o domínio das variáveis. Como não faz parte da especificação inicial do cliente, convém colocar domínios que não causem problemas, nem compliquem o resto da especificação, caso contrário perde-se a vantagem de passagem das constantes para variáveis. Por exemplo, ao não incluir a restrição *5<=CA+CF* poderia acontecer que para cada jogador existissem menos de 5 cartas, enquanto que as mãos são vistas em grupos de 5 cartas. Teriamos de definir o que acontece para esses casos, e estariamos provavelmente a seguir por um caminho bem distante do que o cliente pretende.

Especificação C

1. $Numeros = \{2, 3, 4, 5, 6, 7, 8, 9, 10, V, D, R, A\}$
2. $Naipes = \{P, O, C, E\}$
3. $Baralho = Naipes \times Numeros$
4. $Mao5 = \left\{m | m \in 2^{Baralho} \wedge |m| = 5\right\}$
5. $ordem : Baralho \rightarrow \{1, ..., 52\}$
6. $Jogador_{j=1,...,NJ} = \{n | ordem\,(n) \in \{1, 2\} \cup \{i | 4j - 1 \leq i \leq 4j + 3\}\}$
7. $nord : Numeros \rightarrow \{1, ..., 13\}$
8. $SCR = \left\{ \bigcup_{i=1,...,5}^{\{(u,v_i)\}|nord(v_i)=i+8} \right\} \cap Mao5$

9. $$SC = \left\{ \bigcup_{i=1,\ldots,5}^{\{(u,v_i)\}|nord(v_i)=w+i} \right\} \cap Mao5 - SCR$$
10. $$Poker = \left\{ \bigcup_{i=1,\ldots,5}^{\{(u_i,v_i)\}|v_1=v_2=v_3=v_4} \right\} \cap Mao5$$
11. $$Fullen = \left\{ \bigcup_{i=1,\ldots,5}^{\{(u_i,v_i)\}|v_1=v_2=v_3 \wedge v_4=v_5} \right\} \cap Mao5$$
12. $$Trio = \left\{ \bigcup_{i=1,\ldots,5}^{\{(u_i,v_i)\}|v_1=v_2=v_3} \right\} \cap Mao5 - Fullen - Poker$$
13. $$2Pares = \left\{ \bigcup_{i=1,\ldots,5}^{\{(u_i,v_i)\}|v_1=v_2 \wedge v_3=v_4} \right\} \cap Mao5 - Fullen - Poker$$
14. $$Par = \left\{ \bigcup_{i=1,\ldots,5}^{\{(u_i,v_i)\}|v_1=v_2} \right\} \cap Mao5 - 2Pares - Trio - Fullen - Poker$$
15. $$Cor = \left\{ \bigcup_{i=1,\ldots,5}^{\{(u,v_i)\}} \right\} \cap Mao5 - SCR - SC$$
16. $$Seq = \left\{ \bigcup_{i=1,\ldots,5}^{\{(u_i,v_i)\}nord(v_i)=w+i} \right\} \cap Mao5 - SCR - SC$$
17. $$Nada = Mao5 - SCR - SC - Poker - Fullen - Trio - 2Pares - Par - Cor - Seq$$
18. $$TiposMaos = \{SCR, SC, Poker, Fullen, Trio, 2Pares, Par, Cor, Seq, Nada\}$$
19. $$TipoMao : Mao5 \rightarrow TiposMaos$$
20. $$TipoMao = \bigcup_{K \in TiposMaos}^{K} \times \{K\}$$
21. $$MelhorMao \subseteq Mao5^2$$
22. $$MelhorMao = \cdots$$
23. $$MaosJogador_{j=1,\ldots,NJ} = 2^{Jotador_i} \cap Mao5$$
24. $$MelhorMaoJogador_{j=1,\ldots,NJ} = m \in MaosJogador_j | \forall_{m' \in MaosJogador_j} (m', m) \in MelhorMao \vee m' = m$$

Até que fórmula leu antes de passar para esta linha? Se leu todas as fórmulas, é certamente um entre poucos que se sente à vontade na matemática. Esta linguagem pode não ser a melhor forma de comunicar com o cliente, no entanto, este problema tem uma especificação complexa, e dado o background do cliente, jogador de poker, é provável que se sinta à vontade na matemática, e desta

forma fica claro o trabalho envolvido, que tem que ser pago e reconhecido em conformidade.

Vamos percorrer as fórmulas e comentar, construindo o que poderia ser uma segunda versão da especificação C para um cliente adverso à matemática:

1. A numeração é a seguinte: 2,3,4,5,6,7,8,9,10,V,D,R,A
2. Existem os seguintes naipes: P,O,C,E
3. Um baralho é constituído por 52 cartas, cada uma identificada por um número e naipe únicos;
4. São analisadas mãos (conjuntos) de 5 cartas;
5. As cartas são baralhadas, ficando por uma ordem aleatória;
6. Cada um dos jogadores pode contar com as primeiras CA=5 cartas abertas (ficam na mesa), e com CF=2 cartas fechadas (apenas suas), distribuídas por ordem a cada jogador;
7. Os números têm uma ordem de valor, sendo 2 o mais baixo e A o mais alto;
8. Uma mão de 5 cartas é SCR (sequência cor real) se tiver os 5 números mais altos e todas as cartas forem do mesmo naipe (cor);
9. Uma mão de 5 cartas é SC (sequência e cor) se tiver os 5 números seguidos e todas as cartas forem do mesmo naipe, e não é SCR;
10. Uma mão de 5 cartas é Poker se tiver quatro cartas com o mesmo número;
11. Uma mão de 5 cartas é Fullen se tiver três cartas com o mesmo número e outras duas cartas com o mesmo número;
12. Uma mão de 5 cartas é Trio se tiver três cartas com o mesmo número e não for Fullen nem Poker;
13. Uma mão de 5 cartas é Dois Pares se tiver duas cartas com o mesmo número e outras duas cartas com o mesmo número, e não for Fullen nem Poker;
14. Uma mão de 5 cartas é Par se tiver duas cartas com o mesmo número e não for Dois Pares, Trio, Poker nem Fullen;
15. Uma mão de 5 cartas é Cor se tiver cinco cartas todas do mesmo naipe, e não for SCR nem SC;
16. Uma mão de 5 cartas é Sequência se tiver cinco cartas com os números seguidos, e não for SCR nem SC;
17. Uma mão de 5 cartas é Nada se não for nenhuma das mãos anteriormente definidas;
18. Existem apenas os tipos de mãos definidos anteriormente: SCR, SC, Poker, Trio, 2Pares, Par, Fullen, Cor, Sequência, Nada;
19. A função TipoMão, para uma mão de 5 cartas retorna o tipo dessa mão;
20. (definição da função TipoMão)
21. A relação MelhorMão, para duas mãos de 5 cartas A e B, identifica se A é melhor que B. A é melhor que B se é de um tipo de mãos menos frequente, ou no caso de serem do mesmo tipo, tem a carta com o número mais alto;
22. (definição da relação MelhorMão)
23. Cada jogador deve seleccionar de entre as cartas que pode contar, uma mão de 5 cartas;
24. A mão de 5 cartas seleccionadas, deve ser a melhor de entre as mãos disponíveis para o jogador.

A utilidade dos diagramas UML nesta fase é reduzida para este exemplo, dado que é um simulador, e não há grandes interacções com o utilizador, este apenas manda o simulador correr e vê o resultado.

Estes requisitos são todos funcionais, um exemplo de um requisito não funcional seria forçar a linguagem de programação a ser C++. São também consistentes, mas facilmente se poderia ter colocado requisitos inconsistentes, logo no requisito 3, no caso de haver um erro no número de cartas no baralho, se este fosse superior a 52 seria impossível de utilizar o número e naipe para identificar cada carta, dado que há 13 números e 4 naipes. Estes requisitos não são provavelmente completos, ao contrário do que seria ideal. Não estão definidas situações que possibilitem sempre escolher a melhor mão de qualquer conjunto de cartas. Por exemplo, se o conjunto de cartas é todo o baralho, o melhor tipo de mão é SCR, existindo 4 mãos possíveis, de igual valor (uma sequência côr real de cada naipe), pelo que fica indefinido

que mão deve ser escolhida nessa situação.

O cliente disse o que queria, é tempo do analista propôr uma solução, que é o que será feito na fase seguinte.

3 - Desenho

Através de desenhos podem-se representar a duas dimensões objectos reais, pessoas, paisagens, mapas, etc. Esses desenhos têm informação sobre as entidades que representam. Assim também é o desenho em informática, mas destina-se a representar um sistema informático, podendo ser feito a vários níveis de detalhe.

O código não pode ser começado a escrever após a especificação de requisitos. Primeiro é necessário fazer o desenho do sistema para se saber que código é necessário desenvolver.

3.1 Desenho Conceptual e Técnico

Após os requisitos definidos, há que transformá-los num sistema funcional. O desenho conceptual ou desenho do sistema é esse sistema na linguagem do cliente. Após o cliente aprovar o desenho conceptual pode-se produzir o desenho técnico ou desenho do programa, que define em termos técnicos como vai ser implementado o sistema.

O sistema anterior aparentemente simples, é na verdade um processo iterativo. A construção do desenho conceptual e técnico é dependente dos requisitos, podendo estes serem revistos caso se encontrem inconsistências ou lacunas. O cliente pode também rever as suas necessidades ao analisar o desenho conceptual e compreender melhor o seu próprio problema e solução apontada, bem como o analista ao construir o desenho técnico pode se ver obrigado a alterar o desenho conceptual.

3.2 Construção do Desenho

Wasserman em 1995 sugere a construção do desenho em uma das seguintes formas:

- Decomposição Modular - atribui funções a componentes ou módulos, começando pelas funções que o sistema tem de suportar;
- Decomposição Orientada nos Dados - baseia-se na estrutura de dados;
- Decomposição Orientada nos Eventos - baseia-se nos eventos recebidos pelo sistema, e como estes devem ser tratados;
- Desenho baseado na Interface - mapeia toda a interacção do utilizador com o sistema, e tudo o que o sistema deve fazer com essa interacção;
- Desenho Orientado a Objectos - identifica as classes de objectos que o sistema tem, suas relações e atributos.

Actualmente a programação orientada a objectos é dominante, pelo que estas estratégias podem-se fundir numa só, tendo em atenção que uma classe deve: ter uma função clara no sistema (decomposição modular); assentar numa estrutura de dados também bem definida (decomposição orientada nos dados); os eventos ou métodos que trata devem ser claros e naturais relativamente à função principal que implementa.

A programação orientada a objectos é apenas uma boa forma de dividir o programa em componentes, ou módulos, mas resta ainda a tarefa de saber quais os módulos que devem ser criados para um determinado sistema. A resposta a esta pergunta não está na especificação de requisitos, o analista tem de construir alternativas e escolher a melhor.

3.3 Características de um bom Desenho

O desenho do sistema é uma tarefa de alta importância, uma vez que influencia o decorrer das restantes

fases do projecto. É desejável que:

- Os diversos componentes de software sejam independentes;
- Cada componente de software seja coeso.

Apenas se existir independência de componentes estes podem ser desenvolvidos e mantidos em separado. Desta forma o projecto poderá ser tão grande e complexo, que o programador responsável por implementar um componente poderá abstrair-se de tudo o resto e desenvolver o componente sem ter que considerar o resto do projecto. Num projecto grande permite também que os diversos componentes sejam implementados em paralelo por vários programadores. Caso não se consiga a independência dos componentes, não só se limita o tamanho e complexidade do projecto que pode ser implementado por um programador que é um ser humano e portanto tem limitações, como se impede que os diversos componentes sejam implementados em paralelo.

A coesão de um componente é um factor essencial para a razão de ser do componente. Caso um componente não seja coeso, então as suas partes estão juntas por puro acaso. Não há vantagem em serem implementadas em conjunto, pelo que devem ser divididas em vários componentes mais pequenos de forma a serem implementadas em separado.

3.4 Independência de Componentes

A independência de componentes é o grau de independência que os diversos componentes têm entre si. A independência de componentes é uma característica desejável num bom desenho. Podem ocorrer diversos níveis de dependências:

- Conteúdo - quando as variáveis de um componente são modificadas por outro componente;
- Comum - quando os dados estão num local comum e são alterados por diversos componentes;
- Controlo - quando um componente A é controlado através de argumentos invocados de outro componente B, não podendo o componente A funcionar de forma independente de B;
- Selo - quando uma estrutura de dados é passada entre componentes;
- Dados - quando apenas dados são passados entre componentes;
- Não dependentes - quando não há passagem de dados entre componentes.

Na programação orientada por objectos, se todas as variáveis forem privadas evita-se no mínimo a dependência de Selo entre classes, o que é aceitável. Caso não se passe estruturas de dados entre objectos, as dependências entre componentes são quanto muito dependências de Dados, o que é desejável que assim seja.

Um exemplo de um erro em programação orientada por objectos, que tem consequências nos custos de desenvolvimento e manutenção: A dependência de conteúdo de uma classe A com outras classes B1, B2, ... Bn, por exemplo, obriga a sempre que se acrescente algo na classe A, se tenha de analisar e testar as classes B1,...,Bn, ou quando se descobre um valor errado numa variável da classe A, se tenha de ver o código para tentar perceber, não só da classe A como também das classes B1,...,Bn. Impede também o teste de unidades de forma independente da classe A relativamente às classes B1,...,Bn.

3.5 Coesão de um Componente

Um componente diz-se coeso se é constituído por partes que se relacionem entre si. A coesão de componentes é uma característica desejável num bom desenho. Pode ocorrer a diversos níveis:

- Funcional - quando as partes se destinam todas a implementar a mesma função, e apenas essa função;

- Sequencial - quando o resultado de uma parte é entrada para a outra parte, e as partes têm de ser executadas em sequência;
- Comunicacional - quando as partes têm de produzir ou alterar o mesmo conjunto de dados, ou um recurso externo;
- Procedimental - quando as partes estão juntas apenas por deverem ser chamadas em conjunto por uma determinada ordem;
- Temporal - quando as partes estão relacionadas pela altura em que são chamadas, por exemplo, todos os procedimentos de inicialização juntos no mesmo componente;
- Lógica - as partes estão relacionadas apenas ao nível lógico, por exemplo, todos os procedimentos de entrada de dados juntos no mesmo componente;
- Coincidental - as partes do componente não têm qualquer relação entre si.

É ideal que as coesões dos componentes sejam funcionais. Assim, ao dar a um programador uma funcionalidade para implementar ou para testar, irá desenvolver/analisar apenas um componente. Quando há um erro detectado, normalmente é sobre uma determinada funcionalidade que não está a ser verificada, e deverá ser fácil identificar o componente que não está a funcionar bem, uma vez que a coesão é funcional.

No outro extremo, da coesão dos componentes ser coincidental, qualquer das tarefas descritas no parágrafo anterior vão obrigar à análise de todos os componentes. Mesmo que o programador tenha tudo em mente, caso tenha acabado de escrever o código (assumindo que apenas ele escreveu código), mesmo assim há penalidade de ter de alterar/ver vários componentes, que normalmente estão em vários ficheiros.

A coesão sequencial e comunicacional são ainda aceitáveis, ficando a coesão procedimental no limite. Os restantes tipos de coesão são de evitar.

3.6 Melhorar o Desenho

O Desenho deve ser devidamente revisto e optimizado antes de se começar a implementá-lo. Existem as seguintes alternativas:

- Redução da Complexidade - tentar reduzir de alguma forma a complexidade das especificações ou da estrutura de dados, o que pode envolver conhecimentos matemáticos ou da área do problema;
- Protótipo de Desenho - actualmente os ambientes de desenvolvimento permitem ter o sistema pronto a correr, pelo que não é necessário fazer protótipos para deitar fora;
- Desenho por contracto - obsoleto com a programação orientada por objectos;
- Análise de árvore-de-falhas - técnica para detectar incoerências.

O desenho por contracto perde o sentido na programação orientada por objectos, em que cada classe segue essa filosofia. A análise de árvore-de-falhas não é concretamente uma técnica para melhorar um desenho, excepto se se chamar a detecção de uma incoerência e respectiva remoção de incoerência um melhoramento do desenho.

A única real maneira de optimizar um desenho é reduzindo a complexidade do Desenho, através da escolha de um Desenho alternativo. Conhecimentos matemáticos ou conhecimentos da área do problema é a única alternativa. No entanto, atenção! Mais vale um Desenho simples que se sabe estar em condições e de acordo com os Requisitos que um Desenho optimizado mas incerto.

3.7 Verificação e Validação do Desenho

O último passo na fase de Desenho é verificar e validar o Desenho. Para tal há algumas técnicas que se podem utilizar: validação matemática; cálculo de indicadores de qualidade; comparação de desenhos; revisões de desenho.

A **validação matemática** permite em partes mais algorítmicas provar que o algoritmo satisfaz os requisitos. Muitas vezes esta técnica consome demasiado tempo, dependendo da complexidade do que se pretende provar.

O cálculo de **indicadores de qualidade do desenho** permite efectuar a comparação entre desenhos alternativos. Estes indicadores estão desenvolvidos para a programação orientada por objectos. Permitem também a **comparação de desenhos**, que basicamente a partir da mesma especificação se constrói vários desenhos, escolhendo-se o desenho com base numa análise multi-critério.

Uma revisão de desenho é uma reunião com o intuito de verificar e validar o desenho. A **revisão preliminar do desenho** destina-se a verificar se o desenho conceptual corresponde ao que o cliente pretende e está de acordo com os requisitos. A **revisão crítica do desenho** e a **revisão do programa de desenho**, destinam-se a verificar o desenho técnico. As reuniões devem ser feitas com algum formalismo, devendo existir um secretário responsável por redigir uma Acta com os principais pontos apontados escritos de forma clara e concisa. O cliente deve estar presente nas revisões preliminares do desenho. Deve-se convidar colegas não envolvidos no projecto, mas deve-se manter o número de participantes reduzido de forma a facilitar o diálogo.

3.8 Medidas em Programação Orientada por Objectos

Em Programação Orientada por Objectos (POO) existem medidas de esforço de implementação e manutenção de software, mais precisas das que estão disponíveis para programação estruturada. Estas medidas são no entanto válidas para Engenharia de Software no geral, uma vez que se podem adaptar também para a programação estruturada, basta que para tal se pense num módulo ou componente como sendo uma classe. Estas medidas podem estar disponíveis apenas em algumas fases do projecto.

Métricas propostas por Lorenz & Kidd:

- Número de Operações ("number of scenario scripts" / NSS);
- Número de classes principais;
- Número de classes auxiliares;
- Rácio de classes auxiliares por classes principais;
- Número de subsistemas;
- Tamanho da classe (número de métodos e atributos próprios e herdados);
- Número de métodos redefinidos pela subclasse ("number of operations overridden by a subclass" / NOO);
- Número de métodos adicionados pela subclasse;
- Índice de especialização ("specialization index" / SI), SI=(NOOxnível)/(número de métodos).

Nota: As primeiras cinco métricas podem ser utilizadas em fases iniciais do projecto, para decidir entre desenhos alternativos, ou ter uma primeira ideia do esforço de implementação. As últimas 4 métricas são mais centradas na classe, muito embora sejam bastante superficiais, uma vez que consideram os métodos todos com igual contribuição para o tamanho da classe, assim como os atributos.

Exemplo de SI: Uma classe definida a partir de uma só super-classe, com 10 métodos redefinidos de um total de 33 métodos da super-classe, tendo a classe mais 3 métodos novos. O nível da classe é 1 uma vez que só há uma super-classe. Aplicando a fórmula tem-se: **SI**=10*1/36=**0,28**.

Métricas propostas por Chidamber & Kemerer:

- Métodos pesados por classe ("weighted methods per class"/ WMC) - deixa a complexidade de cada método em aberto;
- Nível da árvore de herança ("depth of inheritance tree" / DIT);
- Número de subclasses ("number of children" / NOC) - subclasses imediactas;
- Dependências entre objectos ("coupling between objects" / CBO) - número de classes que interagem com a classe;
- Resposta de uma classe ("response for a class" / RFC) - número de métodos que podem ser executados em resposta a uma mensagem;
- Falta de coesão dos métodos ("lack of cohesion of methods" / LCOM) - max{0;|P|-|Q|}; P - conjunto de pares de métodos cujos atributos que utilizam são disjuntos; Q - conjunto de pares de métodos que utilizam um ou mais atributos em comum.

Notas: estas métricas são mais detalhadas e também relevantes para a estimativa do esforço de implementação/manutenção de código, mas mantendo a simplicidade. A métrica mais complicada de calcular é a falta de coesão. Nesta métrica consideram-se todos os pares de métodos, e vê-se para cada par se utilizam atributos da classe em comum ou não. Caso a maior parte dos pares utilize atributos em comum, a falta de coesão é nula, caso contrário a falta de coesão é o número de pares a mais que não têm atributos em comum.

Exemplo de LCOM: Uma classe tem um atributo, que é utilizado no método m1 e m2, tendo também os métodos m3 e m4 que não utilizam o atributo da classe. O único par de métodos com atributos em comum é o (m1,m2), e os restantes 5 pares não têm nada em comum. A classe tem portanto valor 5-1=4 de falta de coesão.

Métricas propostas por Li & Henry:

- Dependência de mensagens ("message-passing coupling") - número de chamadas definidas na classe;
- Dependência de dados ("data abstraction coupling") - número de dados abstractos utilizados na classe.

Notas: estas duas métricas complementam a dependência de objectos.

Mais informação: http://pt.wikipedia.org/wiki/M%C3%A9tricas_de_software

Exemplo: Poker, Desenho Conceptual

É altura de se dar uma solução ao problema do cliente, mas ainda na linguagem do cliente, de forma a que este valide a solução proposta. Este exemplo, sendo um simulador, temos de indicar de alguma forma o essencial do algoritmo.

Parâmetros:

- semente aleatória;
- número de jogadores;
- número de cartas fechadas;
- número de cartas abertas;
- número de jogos.

Algoritmo:

1. Baralhar o baralho de cartas;
2. Dar cartas a cada jogador;
3. Calcular a melhor mão de cada jogador (ordenar as cartas do jogador com as cartas da melhor mão em primeiro lugar);
4. Mostrar resultado.

O ponto 3 do algoritmo é aqui o desafio, e também o ponto que poderemos necessitar de validação da parte do cliente, pelo que deve ser detalhado. Evidentemente que, se o cliente não conseguir compreender, este ponto pode passar para o desenho técnico.

Calcular melhor mão:

1. Verificar pares, trio, poker e fullen;
2. Verificar sequência*;
3. Verificar cor**;
4. Verificar sequência e cor (incluindo a real);
5. Ordenar as cartas, primeiro as cartas do tipo da mão, e as restantes por ordem de número;
6. Retornar o tipo de mão: 0 - nada; 1 - par; 2 - 2Pares; 3 - trio; 4 - sequência; 5 - cor; 6 - fullen; 7 - poker; 8 - sequência e cor; 9 - sequência e cor real.

* - executar apenas se o resultado for nesse passo inferior a 4
** - executar apenas se o resultado for nesse ponto inferior a 5

Verificar pares, trio, poker e fullen:

1. Ordenar as cartas por números (pares, trios, quadras ficam todas seguidas);
2. Processar todas as cartas, e para cada carta:
    1. Se existirem 3 cartas seguintes com o mesmo número da carta actual, então o resultado passa a poker=7;
    2. Se existirem 2 cartas seguintes com o mesmo número da carta actual, então o resultado passa a trio=3. No entanto, se já tiver sido detectado um par, dois pares ou outro trio, então o resultado passa a fullen=6;
    3. se existir 1 carta seguinte igual, então o resultado passa a par=1. Se já foi detectado um par, ou dois pares, então o resultado é dois pares=2. Caso tenha sido detectado um trio, o resultado passa a fullen=6.

Verificar sequência:

1. Processando as cartas por ordem novamente:
    1. Caso a sequência esteja vazia, guardar o número da carta actual;
    2. Se o último número da sequência é igual à carta actual, não fazer nada, passar para a próxima carta;
    3. Se o último número da sequência é anterior ao número actual, adicionar o número actual à sequência. Se a sequência ficar com 5 cartas, então o resultado é sequência=4, e pára.
    4. Se o último número da sequência difere em mais de uma unidade ao número da carta actual, esfaziar a sequência e inicializar com o número actual.

Verificar Cor:

1. Processar todos os naipes:
    1. Processar todas as cartas:
        1. Se a carta actual é o naipe actual, incrementar contador;

2. se o contador atingiu o valor 5, o resultado é cor=5, e pára.

Verificar sequência e cor:

1. Processar todos os naipes:
    1. Processar todas as cartas:
        1. Se a carta actual é o naipe actual:
            1. Se já estão cartas na sequência+cor e o número da última difere em mais de uma unidade com o número da carta actual, esvaziar as cartas em sequência+cor;
            2. Colocar a carta actual em sequência+cor;
        2. Se o número de catas em sequência+cor é 5:
            1. Se o A está presente, o resultado é SCR=9;
            2. Caso contrário o resultado é SC=8
            3. Parar.

Se o cliente não compreender a descrição do algoritmo, não há nada a fazer, mas se tal não for o caso, a vantagem do cliente validar a solução apresentada é muito importante porque: por um lado retira ao analista parte da responsabilidade de garantir que a solução proposta resolve o problema do cliente, e por outro lado permite ao cliente reflectir se o problema que especificou é realmente o problema do qual necessita de uma solução. Caso o desenho conceptual não seja devidamente compreendido pelo cliente, este só irá reflectir quando tiver a primeira versão da aplicação, altura em que compreende realmente qual tinha sido a solução proposta.

Exemplo: Poker, Desenho Técnico

Nesta fase é necessário dar soluções técnicas, não é necessário ter a preocupação de fazer um documento que o cliente compreenda, mas sim que o programador compreenda, e possa com base nele construir uma aplicação.

Dado que o cliente não especificou nenhuma interface com o utilizador, e para facilitar os testes, faremos uma aplicação de linha de comando e que funciona com o EngineTester. Esta opção não nos restringe futuras utilizações do código em outros contextos, apenas nos facilitará a interface e também os testes. Para tal é necessário reutilizar código do EngineTester com as classes: TEngine; TVector; TRand, sendo fornecido também o código da função main. Tem que se redefinir uma subclasse de TEngine para o problema concreto, e as classes específicas.

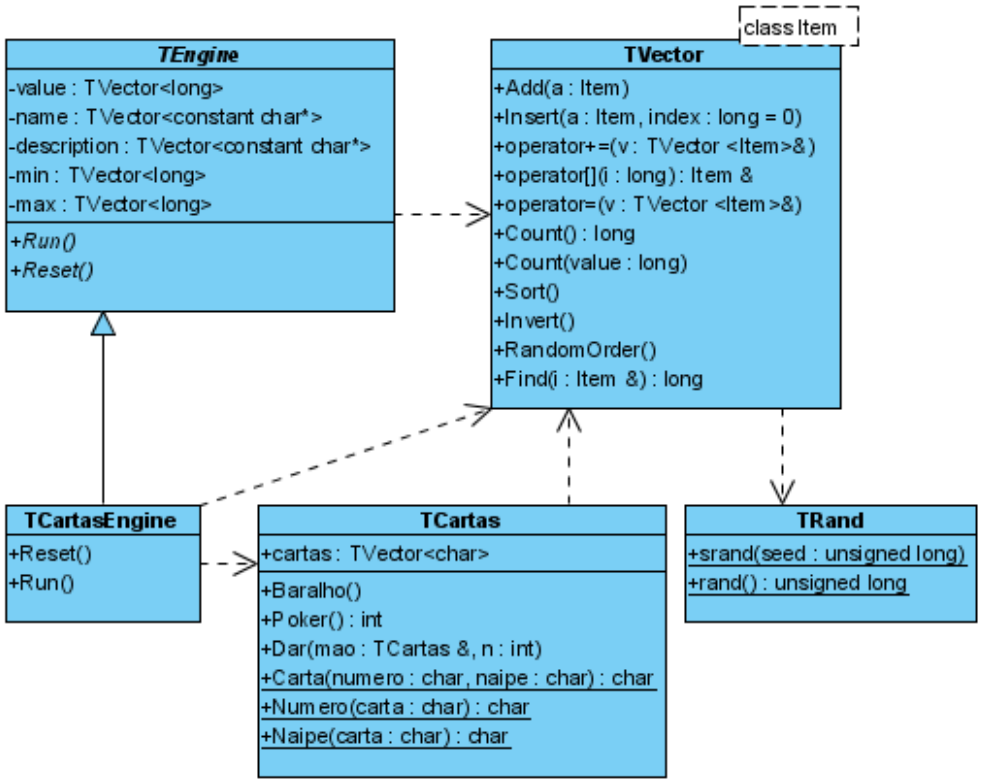
Através da documentação existente podem-se identificar vários candidatos a classes: Carta; Baralho; Mão; TipoMão; Jogador; Jogo. A experiência e o bom senso são essenciais, podendo nesta altura tanto simplificar um problema complexo, como complicar um problema simples. As indicações neste capítulo sobre um bom ou mau desenho devem ser lidas e questionadas principalmente por quem não tenha ainda muita experiência. Neste caso concreto, quais dos candidatos a classe descritos acima valem a pena passar para classe? Nenhum. No entanto, é de toda a conveniência a criação de uma classe. Vamos começar por analisar os candidatos a classe:

- Carta - Tem como atributos o naipe e número. Tem algum método? Não. Os atributos mudam ao longo do tempo? Não. Ainda por cima cabe num só byte, pelo que não faz sentido uma classe.
- Baralho - Esta classe poderia guardar para cada carta o local onde está, e em que posição. Poderia ter um método para baralhar, outro para ordenar as cartas que estão num local, e um método para calculo da melhor mão para as cartas de um determinado local. O único método a utilizar todas as cartas seria o método Baralhar, de resto os restantes só utilizam parte dos dados da classe, pelo que faz mais sentido uma classe que tenha um conjunto de cartas em vez de ter

todas as cartas separadas por local.

- Mão - Esta é a classe mais parecida com um conjunto de cartas, dado que uma mão é um conjunto de cartas, mas o nome deixa de fora a possibilidade de ter todo o baralho, pelo que um nome Cartas é mais geral e indicativo do conceito que representa, um conjunto de cartas.
- TipoMão - Poderia haver um conjunto de classes em que cada uma tinha um método para detectar se uma mão era desse tipo ou não, mas como se viu no desenho conceptual, há vantagem em verificar todos os tipos de mão de uma só vez, numa só função por uma ordem específica. Esta classe a existir teria apenas métodos, o que não é um bom indicador para uma classe.
- Jogador - Os atributos seriam as cartas que um jogador possui, ou seja, um objecto com um conjunto de cartas. A classe contem um só atributo, não há métodos que não sejam aplicados directamente ao único atributo que possui, pelo que a criação desta classe, para já não trás vantagens.
- Jogo - Dado que se pretende simular 1 jogo, não há vantagem aparente. Quando se pretender simular vários jogos e manter os jogos em memória, então nessa altura fará sentido a criação desta classe.

Aconselha-se portanto a criação da classe específica TCartas e à redefinição da classe TEngine, a classe TCartasEngine.



Este diagrama de classes tem para as classes reutilizadas os atributos e métodos que são relevantes para a nossa implementação, e para as classes implementadas os atributos e métodos que se prevê serem necessários.

O *TVector* é uma classe que para gerir a alocação/dealocação de vectores de objectos de uma classe arbitária *Item*. Tem os métodos convenientes a uma abstracção tanto a uma sequência de objectos, como a um conjunto de objectos. São apresentados no diagrama os mais úteis para o nosso problema. A classe *TRand* tem um gerador de números aleatórios implementado, utilizado pelo *TVector* para o método *RandomOrder*, o que permite manter o output constante no caso do código ser compilado num outro compilador, o que não aconteceria se se utilizar o gerador de números aleatórios de uma biblioteca disponível num compilador específico.

O *TEngine* é uma classe abstracta, que tem 5 vectores com informação sobre os parâmetros. Para cada parâmetro tem de existir um valor em cada um dos vectores (mais correcto seria ter feito uma classe para parâmetro, e um vector de parâmetros). Cada parâmetro tem um valor, um nome, uma descrição, um valor máximo e um valor mínimo. As unidades são sempre em inteiros. Os parâmetros devem ser inicializados no método *Reset*. O método *Run* deve ter o algoritmo, que deve aceder ao vector value para obter o valor dos parâmetros, que podem não ser os inicializados no método Reset no caso do utilizador ter alterado o valor de omissão. A classe *TCartasEngine* deve portanto redefinir esses dois métodos.

O método *TCartas* tem um *TVector* com o conjunto de cartas, sendo cada carta um *char*. Para aceder ao número e naipe de uma carta, os métodos estáticos *Numero* e *Naipe* estão definidos, bem como para construir uma carta com o número e naipe, pode-se utilizar o método estático *Carta*. O método *Baralho* inicializa este conjunto de cartas com todas as cartas do baralho, enquanto que o método Poker calcula o valor da melhor mão disponível (o algoritmo principal), existindo ainda um método *Dar*, que retira *n* cartas deste conjunto de cartas, e coloca-as no conjunto de cartas *mao*. Não é necessário um método para baralhar, já que TVector já tem o método *RandomOrder*.

Vamos ficar por aqui, os métodos *Run* e *Poker* estão no desenho conceptual, sendo provavelmente mais penosa a construção de um diagrama de actividades do que propriamente o código. Os métodos *Baralho* e *Dar* têm uma implementação simples, e embora a utilização da classe *TVector* facilite, mesmo sem esta classe apenas se ter cuidado na alocação de memória e implementar um algoritmo de ordenação e outro para baralhar. É portanto altura de começar a escrever o código!

"Software Engineering, theory and practice", second edition, Prentice Hall, Shari Pfleeger, pág. 196-201, 220-225, 231-248, 294-300.

4 - Código

Esta é a parte do processo em que se dá realmente instruções ao computador para que sejam executadas de forma automática: o código. Não se ensina aqui a escrever código, a programar, essa matéria já deve ser do conhecimento do estudante. Neste capítulo estamos interessados em programar bem, não em termos de conseguir implementar um algoritmo complexo e eficiente, mas em termos de facilidade de escrita e de leitura.

Código complexo e brilhante, com documentação interna de forma a ser lida por pessoas inteligentes, com capacidade de ler e apreciar a obra de arte, será lido por um colega apenas após algum bug ter sido reportado, e a sua apreciação será mais do tipo: "que grande baralhada que para aqui anda".

Como se pode tornar mais fácil a escrita? Através de um ambiente de desenvolvimento com auxiliares de memória e de escrita, levando a que o código fique à partida com menos erros e simultaneamente leve menos tempo a ser escrito, contribuindo ambos os pontos para a redução do custo do software, e para o aumento do prazer de programar.

A redução do tempo de escrita do código consegue-se apenas se o código for escrito com calma. Caso o código seja escrito à pressa, independentemente do ambiente de desenvolvimento, o número de erros sobe e a quantidade de trabalho feita que no final não serve para nada aumenta, levando a um aumento no tempo de escrita de código.

Como se pode tornar mais fácil a leitura? Aqui o ambiente de desenvolvimento tem também influência, mas com menor relevo. A boa documentação interna, a utilização de uma norma de escrita de código e a reutilização de código é a base para se conseguir escrever código de fácil leitura e portanto mais facilmente testável e com menores custos de manutenção.

Caso algo corra mal nesta fase, não se pode concluir que a culpa é do programador. Existe também o próprio problema que deveria estar resolvido na fase de desenho. Caso tenha que ser implementado um algoritmo complexo, não completamente definido na fase de desenho, o programador pode não ter bases matemáticas para o compreender e implementar. Ou o módulo a implementar pode ser demasiado grande e o desenho actual não permitir abstracções. Nesse caso o programador poderá não ter capacidade para compreender e implementar o módulo. É portanto necessário que a fase de desenho seja bem realizada para que esta fase também o seja. O bom senso manda também que se atribua os módulos mais complexos aos programadores mais experientes.

4.1 Ambiente de Desenvolvimento

Como pode um ambiente de desenvolvimento facilitar a escrita do código? Para responder a esta pergunta vamos descrever brevemente a escrita do código de um programador com um editor de texto, um compilador e debuguer de linha de comando, que são as ferramentas mínimas para escrever código, após o qual iremos ver os diferentes aspectos em que podem ser melhorados com um Ambiente de Desenvolvimento.

4.1.1 Editor de texto, compilador e debuguer

Com base no desenho técnico, a primeira tarefa é criar os ficheiros necessários, o que depende da linguagem e de opções do programador. Dois ficheiros por cada classe necessária, um com a declaração e outro com a implementação, é a escolha normal. Num caso extremo pode-se criar um só ficheiro com todo o programa, complicando a navegação pelo código.

Num segundo passo tem que se escrever as declarações das classes, e escrever as implementações dos diferentes métodos vazios, bem como tudo o que é necessário para o programa se poder compilar, e finalmente compilar o programa. Após alguns acertos entre nomes das classes e métodos que estão ligeiramente diferentes na declaração e implementação, o programa compila.

No terceiro passo começa-se a escrever código propriamente dito, a implementação de cada método de uma classe. Nessa altura é necessário conhecer para que servem todos os atributos da classe e restantes métodos, bem como ter o conhecimento das classes e bibliotecas que são utilizadas. Após cada método escrito, há que compilar para ver os erros de sintaxe. Nomes de atributos/métodos mal escritos, blocos de código mal fechados, argumentos do tipo errado, são exemplos de erros de sintaxe encontrados e corrigidos. O compilador dá uma lista de defeitos, com a linha em que ocorrem. Infelizmente o defeito no código pode ser originado por um erro em outra zona do código, provavelmente anterior, e o mesmo erro pode originar muitos defeitos. Como se vê a lista de defeitos, convém após a localização do erro voltar a compilar para obter a lista actualizada de defeitos. Este processo deve ser repetido após cada método para limitar a zona que se tem que procurar pelos erros.

Num quarto passo, após uma classe estar completamente escrita, convém preparar um pequeno bloco de teste para ver se a classe está a fazer o que era suposto. O programa compila e corre o bloco de teste. Este passo deve ser feito sem ansiedade, já que tem-se a certeza que haverá problemas. Após confirmada a dura realidade tem que se obter mais informação sobre o que não está correcto, normalmente espalhando informação de debug do que está a ser feito a cada momento para o ecran. Pode-se também utilizar um debuguer de linha de comando, o qual permite correr o programa e parar numa determinada linha de código, podendo nessa altura consultar o estado do programa, nomeadamente consultar o valor das variáveis, saber quais os métodos chamados e com que argumentos, etc. Ao detectar um problema são introduzidas alterações ao código para que este funcione melhor e repete-se o processo. Por vezes as alterações não resolvem e criam elas mais problemas, mas eventualmente todos os problemas ficam resolvidos e o primeiro teste à classe implementada é passado positivamente.

As outras classes vão sendo feitas, algumas delas passando por cima do ponto quatro, isto é, não são testadas em separado de forma a não perder tempo a fazer o código de teste. Uma altura chegará em que tudo está implementado. Nessa altura o código é novamente testado através de um processo idêntico ao ponto quatro mas com código suspeito todo o código produzido e não apenas uma classe. Nesta altura é normal dizer que o código já está feito, só falta testar, sendo de prever que logo que o código corra uma vez correctamente, seja dado como código a funcionar, não havendo lugar a mais demora.

É claro que tudo isto é possível acabar aqui caso o desenho esteja em condições. Se o nosso programador teve a sorte de ter um colega experiente que fez o desenho, então tudo bate certo. Caso o desenho tenha sido feito por alguém sem experiência, descobre agora que o desenho não dava para resolver o problema em questão, era até incoerente, pelo que não consegue completar o programa para fazer o teste final.

4.1.2 Melhoramentos

Pode-se facilitar a escrita de código primeiro no editor de código. O editor deve conhecer a sintaxe da linguagem de programação, de forma a colorir as palavras pertencentes à linguagem, bem como o início e fim de blocos. Deve também ter conhecimento dos restantes ficheiros pertencentes ao projecto, de forma a poder identificar nomes de atributos e métodos válidos, identificando assim os mal escritos. Desta forma, erros de sintaxe são detectados logo em tempo de edição e corrigidos. Este conhecimento é aproveitado também para facilitar a escrita. Ao escrever as primeiras letras de um atributo ou método, o

editor fornece logo as várias possibilidades que existem para completar com um nome válido.

Ao criar uma nova classe, o ambiente de desenvolvimento cria logo os ficheiros de declaração e implementação, e o código essencial, dependente naturalmente da linguagem de programação. Pode-se adicionar um método ou atributo através do ambiente de desenvolvimento, sendo colocada logo uma declaração e implementação nos ficheiros respectivos. Ao chamar-se um determinado método tem que se conhecer o que este faz e quais os seus argumentos. Durante a escrita, ao começar a escrever os argumentos de um método, o editor apresenta de imediacto ao programador não só os argumentos na declaração do método, como também o comentário que está junto da declaração com a descrição do que o método faz.

A navegação pode ser também melhorada. Sem ambiente de desenvolvimento, o programador quando quiser ir para a implementação de um método, tem que saber em que ficheiro está, abrir o ficheiro no editor, e navegar no ficheiro até ao método. O ambiente de desenvolvimento pode listar todas as classes e métodos do projecto, independentemente dos ficheiros em que estão, permitindo ao utilizador ir directamente para a declaração ou implementação do método. Naturalmente que se o programador necessitar, pode efectuar procuras ou substituição de texto num ou em todos os ficheiros do projecto.

A integração do editor, compilador e debuguer, permite por um lado que da lista dos erros de compilação se vá directamente para a linha do código onde o erro foi detectado, e por outro lado que se possa colocar pontos de paragem no código e mandar correr, e ao parar o programa obtem-se logo janelas com o estado do programa. Pode-se nessa altura efectuar algumas alterações ao código e compilar continuando a correr o programa com a actualização, não sendo necessário em muitas situações reiniciar a corrida.

As vantagens a nível do editor, navegação pelo projecto e do debuger, são determinantes na produtividade do programador. No Tuturial do Visual C++ 2008 Express Edition são descritas diversas funcionalidades do ambiente de desenvolvimento. É essencial conhecer e experimentar para decidir entre utilizar ou não. Ficar na ignorância poderá significar uma perda de produtividade e conforto na escrita de código.

Encontram-se com certa facilidade programadores experientes utilizando o ambiente de desenvolvimento editor+compilador+debuguer desintegrado, sendo a integração o sistema operativo. Estes programadores não vêm vantagem nos novos ambientes de desenvolvimento integrados. Difícil é encontrar programadores que tenham utilizado ambientes de desenvolvimento integrados num projecto concreto e tenham voltado ao editor+compilador+debuguer.

As diferentes ajudas nunca devem ser vistas de forma a que o programador não necessite de saber isto ou aquilo. O programador é responsável por todo o código, excepto o código das bibliotecas que utilizar, que tem de o ler e perceber, mesmo que parte deste tenha sido escrito automaticamente pelo ambiente de desenvolvimento.

4.2 Reutilização de Código

A reutilização de código pode dar-se a dois níveis: produção de código reutilizável; consumo de código reutilizável.

Consumidores de código reutilizável devem ter em atenção aos seguintes pontos:

- O componente executa a função e devolve a informação necessária?
- Se for necessária uma pequena modificação, essa modificação é menor que construir o componente de raiz?

- A documentação permite compreender o componente sem ter de se ver o código linha a linha?
- Existe o historial de falhas corrigidas?

Esta última pergunta poderá pensar-se que deva ser ao contrário. Como é natural que o código tenha falhas, a existência do historial de falhas corrigidas apenas vem dar informação que o código foi bastante testado e pode ser utilizado com segurança. Se por outro lado nada se saber sobre falhas, quer dizer que não se encontrou falhas porque se testou pouco.

Os produtores de código reutilizável devem ter em atenção aos seguintes pontos:

- Fazer o componente genérico, utilizando sempre que possível parâmetros em vez de constantes;
- Separar partes que eventualmente possam ter de ser alteradas das que provavelmente nunca serão alteradas;
- Manter historial de falhas corrigidas;
- Respeitar a norma de escrita de código (formatação; comentários; nomes; estilo).

Estes pontos podem ser respeitados o máximo possível, mesmo ao escrever código que não é para ser reutilizado. Isto porque na verdade não se sabe se não irá aparecer no próprio programa uma situação em que o código que se está a escrever não possa ser reutilizado, e depois porque o módulo a construir pode vir a ser repensado e feito reutilizável para outros projectos. Estes pontos facilitam também a fase de testes.

4.3 Norma de Escrita de Código

Esta secção destina-se a normalizar a escrita de código, sendo apenas uma possibilidade entre outras. O intuito da normalização é tornar a leitura de código mais simples, tanto de outras pessoas como do próprio autor. Desta forma reduz-se custos de teste e manutenção.

Esta secção está orientada para terminologia da programação orientada por objectos, mas pode ser utilizado também na programação estruturada e programação funcional. A correspondência de **classe** para programação estruturada poderá ser **módulo** ou **ficheiro** de código, enquanto que a correspondência de **método** será **função** ou **procedimento**.

O ambiente de desenvolvimento em programação orientada por objectos tratará da criação dos ficheiros, devendo os nomes serem iguais aos nomes das classes. Caso assim não seja, deverá ser criado um ficheiro por cada classe, com o mesmo nome da classe, ou dois no caso de existir ficheiro de declaração e implementação. Em programação não orientada por objectos, o projecto deve ser dividido em módulos, ficando cada módulo num ficheiro. Os módulos devem conter funções que estejam relacionadas entre si, e não devem ser muito grandes.

A língua utilizada deve ser sempre a mesma ao longo de todo o código, tanto no nome das variáveis e funções como nos comentários.

1. Formatação

1.1 Indentação: Cada linha de código deve começar por 4 espaços por cada bloco de código em que estiver inserido. É aceitável em vez de 4 espaços utilizar outro valor, desde que seja sempre o mesmo para todo o ficheiro.

Exemplos C++:

```
double TPolinomio::operator[](double x)
{
    double resultado = a.Last();
    for(long i = a.Count()-2; i>=0; i--)
        resultado=resultado*x+a[i];
    return resultado;
}
```

```
double TPolinomio::operator[](double x)
{
  double resultado = a.Last();
  for(long i = a.Count()-2; i>=0; i--)
      resultado=resultado*x+a[i];
  return resultado;
}
```

As duas primeiras linhas e a última, não estão dentro de bloco nenhum, enquanto que a 3ª, 4ª e 6ª linha estão dentro do bloco de código do método, sendo antecedidas de 4 espaços. A 5ª linha está ainda dentro do bloco do ciclo for, pelo que sobe 8 espaços.

Este código está aparentemente bem indentado, no entanto utiliza 2 espaços para a 3ª, 4ª e 6ª linha, enquanto que para a 5ª linha utiliza 6 espaços. Se a indentação é a 2 espaços, a 5ª linha teria de ter 4 espaços e não 6.

1.2 Tamanho das linhas: As linhas tanto código como de comentários não devem exceder os 80 caracteres. Caso tal não seja possível, a linha de código deve ocupar mais que uma linha. Conta para efeitos de indentação a segunda linha e seguintes, como se estivessem dentro de mais um bloco de código. Uma linha de comentário que continue na linha seguinte mantém a indentação.

1.3 Argumentos: As listas de argumentos, tanto na definição como na utilização, não são considerados blocos de código. Assim devem ser escritos na mesma linha, se possível, caso contrário aplica-se o ponto 1.2.

1.4 Início e fim de bloco: A marca de início e de fim de bloco de código deve utilizar uma linha para o início e outra para o fim do bloco. As linguagens funcionais podem ignorar esta regra, uma vez que em geral o primeiro elemento do bloco é um identificador e convém que fique na mesma linha que a abertura de bloco.

Exemplo de C++ e Lisp:

```
double TPolinomio::operator[](double x) {
    double resultado = a.Last();
    for(long i = a.Count()-2; i>=0; i--)
        resultado=resultado*x+a[i];
    return resultado;
}
(defun power (x n)
  (cond ((= n 0) 1)
        ((evenp n) (expt (power x (/ n 2)) 2))
        (t (* x (power x (- n 1))))))
```

O início de bloco deste código está na mesma linha que o nome do método, o que está incorrecto. Deve-se utilizar uma linha separada.

Em Lisp o início de bloco é igual ao início de argumentos e é colocado antes do identificador, pelo que não faz sentido uma linha nova. Ao fechar o bloco é usual não utilizar novas linhas.

A questão da indentação muitas das vezes é vista como uma opção pessoal. Cada um acha que indenta bem o código, o que leva a que cada código que se leia fique na sua indentação própria, perdendo-se parte da vantagem para quem lê. Já o código não indentado por completo torna a leitura tortuosa, dado que força o leitor a prestar grande atenção ao início e fim de blocos, atenção essa que faz parte das obrigações de quem escreve e assim passa também para quem lê o código, caso este não esteja indentado.

2. Comentários

2.1 Localização: Os comentários devem anteceder o código que comentam. Pode-se colocar pequenos comentários ao bloco na própria linha de início ou fim de bloco de código.

2.2 Comentário de classe: No início de cada classe deve existir um comentário de classe. Neste comentário deve conter informação sumária sobre a classe. Dá-se liberdade de incluir mais ou menos campos conforme apropriado, no entanto o comentário de classe deve ter obrigatoriamente os campos (inglês/português):

- Author/Autor - uma ou mais pessoas que contribuíram para a classe
- Last change/Última modificação - data da última modificação
- Description/Descrição - descrição da estrutura de dados e breve descrição dos algoritmos envolvidos, esclarecendo o que os algoritmos fazem e não como fazem

2.3 Comentário de método: Cada método deve ter um comentário inicial, idêntico ao comentário de classe, acrescentando:

- Descrição do algoritmo, não só o que faz mas também como o faz;
- A complexidade temporal do algoritmo, sempre que possível. No início dos blocos de código pode-se colocar também a complexidade temporal do bloco de código;
- Pré e pós-condições de utilização do método, por exemplo, os elementos de uma lista estarem ordenados;
- Incluir eventuais restrições de utilização, por exemplo, o valor máximo para cada argumento do método para o qual o método foi testado e funciona bem. Após serem realizados testes, os resultados dos testes devem ser também incluídos.

2.4 Comentário de linha: Devem ser colocados comentários de linha apenas em zonas complexas, de forma a complementar a descrição do algoritmo feita no comentário de método. Os comentários de linha devem assumir que o leitor conhece a linguagem de programação, pelo que não devem repetir o que está no código.

Exemplo C++:

```
double TPolinomio::operator[](double x)
{
    // a é um vector de números
    // resultado fica com o último número
    double resultado = a.Last();
    // calcula o valor de a3x^2+a2x+a1
    // através da expressão (a3*x+a2)*x+a1
    for(long i = a.Count()-2; i>=0; i--)
        resultado = resultado*x+a[i];
    return resultado;
}
```

O primeiro comentário é desnecessário, uma vez que é suposto antes de ler este método tenha sido lida a definição de classe onde estará definido **a**. O nome do método é também suficientemente sugestivo, pelo que o comando não necessita de comentário.

O segundo comentário é correcto pois explica o algoritmo de forma simples, com um exemplo, e não repete o que já está no código.

Os comentários são sempre complicados de se escrever. Primeiro porque quando o código é complexo é também mais difícil de explicar, e em segundo porque a mente do programador está ocupada com a estrutura de dados e com o algoritmo implementado, está mais preocupada com o facto do algoritmo estar ou não a funcionar do que inspirada para a literatura. Por outro lado, enquanto que o código é necessário para o programa correr, os comentários não, e podem ser sempre adicionados mais tarde, quando houver tempo. Já os comentários curtos junto a atributos e métodos o problema é de outra natureza: encontrar uma frase de uma só linha que adicione algo ao próprio nome do atributo ou método. É complicado, mas tem que ser. Os comentários mais difíceis de escrever são os que não podem ficar para depois. Se isso acontecer o mais provável é que o próprio autor se veja ele próprio a olhar novamente para o código que escreveu dias antes a tentar compreendê-lo, e a considerar a hipótese de re-escrever o código todo de novo.

3. Nomes

3.1 Variáveis iteradoras inteiras: Os nomes das variáveis devem começar por letras minúsculas. As variáveis iteradoras inteiras devem ser chamadas **i**, **j** e **k**. Devem ser reutilizadas, devendo o **k** ser utilizado apenas se o **i** e **j** estiverem em utilização. Caso sejam necessárias mais que três variáveis

iteradoras, adicionar um número à letra i (**i1**, **i2**, ...). Caso as variáveis iteradoras sejam reais ou de outra natureza, utilizar as letras **l**, **m** e **n**.

3.2 Variáveis não iteradoras: Para as variáveis não iteradoras deve-se utilizar nomes elucidativos. Na junção de dois nomes, a letra inicial do segundo nome deve ser maiúscula. A primeira letra deve ser sempre minúscula. Caso não existam nomes elucidativos, utilizar as letras **u**, **v** e **w** para os valores inteiros, **x**, **y** e **z** para os valores reais, **a**, **b** e **c** para nomes (strings), e **p**, **q** e **r** para apontadores ou variáveis de outra natureza.

3.3 Constantes e macros: Constantes ou macros devem ser escritos em maiúsculas, e as palavras separadas por um "underscore": '_'.

3.4 Estruturas e classes: Os nomes das estruturas ou classes têm de ter sempre nomes elucidativos e devem começar por **T** se é um tipo abstracto de dados, **S** se for uma estrutura, e **C** se for uma classe no geral, **D** se for uma classe para caixas de diálogo de uma interface gráfica. Na junção de dois nomes, a letra inicial de cada nome deve ser maiúscula.

3.5 Métodos: Os nomes dos métodos devem ter sempre nomes elucidativos. No caso de programação não orientada por objectos, o nome das funções e procedimentos deve ser antecedida do nome ou iniciais do módulo a que pertencem. Na junção de dois nomes, a letra inicial de cada nome deve ser maiúscula. Não utilizar o mesmo nome para funções ou métodos com argumentos diferentes, a não ser que as funções ou métodos sejam apenas diferentes versões. Em princípio, o nome de um procedimento deve conter um verbo no infinitivo, enquanto que o nome de uma função deverá ter o nome da grandeza que calcula, não necessitando de verbo. As funções com resultado booleano devem conter um verbo no presente ou adjectivo de forma a serem utilizadas dentro de condicionais.

Os nomes são parecidos com os comentários: difíceis de obter, essenciais para a compreensão do código, irrelevantes para a execução. Já o tamanho não afecta a produtividade nem cria problemas de haver enganos a escrever. O ambiente de desenvolvimento auxilia e verifica a escrita dos nomes de atributos e métodos, não há motivo para se utilizar iniciais ou abreviaturas.

4. Estilo

4.1 Blocos de código: Não abrir um bloco de código para um só comando, excepto para evitar confusão por exemplo numa cadeia if/else.

Exemplo C++:

```
double TPolinomio::operator[](double x)
{
    double resultado = a.Last();
    for(long i = a.Count()-2; i>=0; i--)
    {
        resultado = resultado*x+a[i];
    }
    return resultado;
}
```

O bloco de código criado para o ciclo for é desnecessário uma vez que tem apenas um comando.

```
if(aux[level]==inst.Count()-1)
{   // complete path
    if(aux.Count()>sol.Count())
    {   // better solution found
        for(k=0;k<aux.Count();k++)
            sol[k]=aux[k];
        // if maximal solution return
        if(sol.Count()==inst.Count())
            stop=true;
    }
}
else
{   // connections in lig, that are not in aux
    for(k=0;k<lig.Count() && !Stop();k++)
        if(aux.Find(lig[k])<0)
        { // a new node not in current path
            aux.Add(lig[k]);
            Branch(level+1);
            aux.Count(level+1);
        }
}
```

O bloco de código do primeiro if é necessário para esclarecer que o else é do primeiro if e não do segundo. O bloco de código do else é desnecessário, mas considera-se razoável uma vez que o if correspondente tem também um bloco de código para um só comando.

4.2 Repetições: Não repetir uma sequência de comandos. Muitas vezes por facilidade ao escrever código, repetem-se comandos quando poderiam ser postos dentro de um ciclo.

Exemplo em C++:

```
float taxa(float valor) {
    if(valor>10000)
        return 0.4;
    else if(valor>1000)
        return 0.2;
    else if(valor>500)
        return 0.1;
    else return 0.05;
}

float limites[]={ 10000, 1000, 500, 0};
float taxas[]={0.4, 0.2, 0.1, 0.05, 0};
float taxa2(float valor) {
    long k;
    for(k=0;limites[k]>0;k++)
        if(valor>limites[k])
            return taxas[k];
    return taxas[k];
}
```

A segunda implementação da função taxa é preferível à primeira, uma vez que o teste é sempre o mesmo e pode ser colocado dentro de um ciclo. Por vezes não só um comando é repetido mas sim vários comandos que poderiam estar dentro de um ciclo, com eventualmente recurso a um ou mais vectores estáticos com os argumentos que sejam necessários.

4.3 Tamanho dos métodos: Não fazer métodos demasiado grandes (mais de 100 linhas de código). Nesse caso agrupar comandos relacionados em métodos, mesmo que o método venha a ser chamado uma só vez.

4.4 Comandos numa linha: Iniciar sempre uma nova linha para um comando, ou seja, não colocar antes de um comando um condicional, ciclo, ou outro comando.

Exemplo em C++:

```
void TPolinomio::operator+=(TPolinomio &p)
{
    for(long k=1; k<p.a.Count(); k++)
        if(a.Count() <= k) a[k] = p.a[k];
        else a[k] += p.a[k];
}
```

Os dois comandos do if/else deviam estar em linhas diferentes.

Exemplo: Poker

As vantagens de um Ambiente de Desenvolvimento, são difíceis de se colocar em texto, mesmo com um exemplo concreto. É preferível que seja o estudante a descobrir por sí próprio. Para tal há duas vias que se sugere, que deverá optar conforme o tempo disponível para estudo:

1. Implementar o exemplo com base no código reutilizado NewEngine.zip e na documentação existente (é o código genérico de um novo algoritmo, disponível no EngineTester que não é necessário instalar agora);
2. Partir do código implementado Cartas.zip e aplicar as normas de escrita de código.

A segunda via é claramente mais simples e menos real que a primeira via, mas entre não fazer nada no Ambiente de Desenvolvimento, é preverível ao menos re-escrever código, que poderá mesmo assim colocar o estudante em situações em que lhe serão úteis os mecanismos de facilitação de leitura, escrita e depuração do código.

Este exemplo do Poker tem código reutilizado, o TVector, enquanto que o TEngine apenas tenha a sua utilidade clarificada no capítulo do EngineTester. Por agora pode ser visto como uma forma de poupar a escrita da interface com o utilizador, que embora seja de texto, dá sempre algum trabalho. A classe TCartas pode ser vista como uma classe a ser reutilizada em outras aplicações de cartas, pelo que as preocupações em escrever código reutilizável devem ser consideradas para esta classe.

Após realizar esta actividade, pela via 1 ou 2, coloque o código no fórum de estudante, e veja o código dos seus colegas. Pode ver o nível de diferenças que duas implementações com a mesma especificação e desenho podem atingir, e por outro lado, a dificuldade que poderá ter em ler o código de um colega,

mesmo tendo acabado de implementar uma aplicação idêntica.

Exemplo: Poker, alteração 1

Vamos agora simular uma primeira alteração à especificação, e respectiva resolução passo a passo. Se quiser resolva primeiro e só depois veja a resolução.

- Pretende-se calcular a frequência absoluta de ocorrência de cada tipo de mão, devendo essa informação ser guardada de forma a ser retornada no método TCartasEngine::Indicator

De forma a simular uma situação real, esta actividade não deve ser realizada logo após a actividade anterior. Deve fazer uma actividade formativa de outra unidade curricular, e só depois continuar. O objectivo desta actividade não é apenas reflectir a alteração no código, mas fazê-lo do modo mais rápido e seguro, numa altura em que já tem a cabeça em outro projecto, utilizando para tal funcionalidades disponíveis no Ambiente de Desenvolvimento. Quem não fez a actividade anterior, pode realizar esta, partindo do código da via 2 (Cartas.zip).

A - Recepção do Código

Há que ter alguns cuidados ao receber código de outras pessoas, nomeadamente é conveniente estes três passos:

- O código compila? Este é o primeiro passo. Se não compilar pode ser um problema de diferenças entre compiladores, ou então houve um engano nos ficheiros que foram enviados. Apenas muito raramente o motivo será de a pessoa nem sequer ter compilado o código que enviou, e mesmo nesse caso não há problema, uma vez que o tempo gasto foi relativamente reduzido. Se a mesma pessoa reincidir é que é conveniente chamar à atenção.
- O código faz o que é suposto? É claro que esta pergunta não se consegue responder porque o código pode ter bugs, mas basta um pequeno teste para ter como referência, e que verifique as diferentes funcionalidades pelo menos uma vez, e confirmar que o código na origem, para o mesmo input está a ter o mesmo output que o código recebido. Se não for esse o caso, então poderá haver diferenças no compilador e há que localizá-las. Sem exactamente o mesmos resultados, não vale a pena avançar. Caso nem sequer o teste tenha sido enviado, então há que pedí-lo.
- O código está a funcionar, há que guardar um backup com o teste pronto a correr, e passar então para a fase seguinte. Se o código recebido for alterado, é conveniente que se possa facilmente voltar a trás sem ter que passar pelos dois pontos anteriores. Este backup é importante também em alturas em que ao corrigir bugs se desconfie do código recebido, de forma a ter a possibilidade de fazer um teste no código original, colocando o input que gera o bug. Caso não se repetisse do bug apenas no código recebido, ficaria sempre a dúvida se o bug teria realmente origem no código original ou se era devido à sua má utilização ou modificações posteriores.

No trabalho, o código são os ficheiros cpp e h, pelo que no Visual C++ 2008 Express Edition há que criar um novo projecto com estes ficheiros e mandar correr, tal como no tuturial.

Pode-se verificar se os ficheiros estão todos, após o qual manda-se correr (F5). Deverá compilar e de seguida o programa inicia. Caso não tivesse escolhido "Console Application Project", daria um erro e o programa não corria.

Para saber se está tudo bem, escreva:

set seed 1

set information 1
set limit states 1
run

O programa deve apresentar o seguinte output:

Comuns: [♥8 ♣7 ♠R93] nada [R♠ 9♠ 8♥ 7♣ 3♠]
Jogador 1: [♥R8 ♣A7 ♠R93] par [R♥ R♠ A♣ 9♠ 8♥]
Jogador 2: [♥8 ♦5 ♣D7 ♠R93] nada [R♠ D♣ 9♠ 8♥ 7♣]
Jogador 3: [♥8 ♦R8 ♣7 ♠R93] dois pares [R♦ R♠ 8♥ 8♦ 9♠]
Jogador 4: [♥8 ♣7 ♠R9543] c¶r [R♠ 9♠ 5♠ 4♠ 3♠]Run end.

Neste caso não há integração do código, parte-se de um código e pede-se para fazer uma alteração.

B - Alteração ao Código

Tendo um código a funcionar, feito por outra pessoa ou pelo próprio já há bastante tempo, pretende-se a correcção de um bug, ou uma pequena nova funcionalidade.

A abordagem mais convencional é consultar toda a documentação de forma a ter a ideia global da aplicação. Caso a alteração seja pequena não é sequer necessário a compreensão global do código, desde que se mantenha o focus no que é necessário fazer. Por outro lado, como regra geral deve-se manter o código desenvolvido o mais pequeno quanto possível, o que leva a que se reutilize a maior quantidade possível de código existente, pelo que algum conhecimento do que existe deve ser mantido.

Segue-se o raciocínio de uma possível abordagem, que tenta minimizar o código alterado, e também o código consultado. Só é analisado o código que tem de ser, e tudo o resto é ignorado. As alterações não podem no entanto ser cegas, são sempre baseadas em evidências. Alterações cegas dão origem a bugs e muito tempo perdido, pelo que vale sempre a pena consultar mais código para ter uma evidência antes de tomar a decisão de alterar algo.

1. Identificar o que tem que ser feito, a resposta à pergunta: o que deve o sistema fazer de novo? Tem que estar claro este ponto, caso contrário não vale a pena começar. Se for possível assentar a resposta no teste que existe ao código, melhor. No trabalho, tem que se contar quantas mãos ocorreram de cada tipo a cada jogador. Há evidência que já existe uma função que calcula o tipo de mão, uma vez que essa informação é colocada no exemplo (nada, par, etc).
2. Estrutura de dados. O que o sistema tem que guardar? No trabalho é um contador (portanto um inteiro) por cada tipo de mão. Quantas mãos existem? Basta procurar no código caso não tivesse essa informação já no enunciado. Poderia-se procurar por "nada" (Ctrl+F » "nada") e encontrava-se um vector, aliás dois vectores de nome poker com as várias mãos possíveis, ou seja, 10 tipos de mãos. É portanto suficiente um vector de inteiros de tamanho 10.
3. Onde guardar os dados? Tem que ser num local onde se possa aceder nos diversos locais. O enunciado diz-nos que se tem que se devolver o resultado no método TCartasEngine::Indicator, pelo que os dados têm que ser acediveis a partir desse método, mas a simulação não é feita neste método, pelo que os dados não podem ser locais. Só podem ficar então na classe do método, TCartasEngine.
    1. Criação da estrutura de dados. No Visual Studio há a possibilidade de adicionar variáveis a objectos através de wizard (na Class View, botão de contexto em cima da classe: Add » Add variable » variable type = long [10] » variable name = contador).
    2. Caso se reutilize a estrutura de dados presente no código, seria necessário rever a sua utilização. Neste caso a classe TCartasEngine não tinha quaisquer dados, pelo que criou-se a estrutura que era necessária.

3. Convém colocar um comentário em todas as zonas alteradas, referenciar a alteração com um código (ou a data de alteração), para no caso de se querer saber o que foi feito para determinada alteração ao código se localizar facilmente as linhas alteradas.

4. Inicialização da estrutura de dados. Onde se inicializa o contador? A primeira resposta é: no construtor da classe. No entanto, no construtor da classe há um só comando, uma chamada ao método Reset(). A interpretação é clara: esta classe tem a inicialização dos dados num método que não o construtor, de forma a poder reinicializar quando for preciso. Mantendo a filsofia, há que colocar a reinicialização no método Reset, mas se no entanto se colocar no construtor não estaria incorrecto. A inicialização é neste caso a colocação da estrutura a zero: for(int i=0;i<10;i++) contador[i]=0;
5. Actualização dos dados. Onde se incrementa o contador?
    1. Há que localizar o código onde a mão é calculada. Um bom ponto de partida é descobrir o local onde é mostrada a informação para a consola: Ctrl+F » "Jogador", após algumas iterações localizavam-se as linhas onde tal informação é disponibilizada:
       i. fprintf(stdout,"\nJogador %d: ",i+1);
       ii. m[i].Save(stdout);
       iii. m[i].SaveSolution(stdout);
    2. Dado que aparece na consola a mão deste jogador, é evidente que no objecto m[i] está já o resultado da mão, ou então esta é calculada numa destas funções. Que tipo é o objecto m[i]? Basta parar o rato em cima da variável e o Visual Studio diz, não há necessidade de andar para cima e para baixo com a scrollbar: TVector<TCartas>, sendo portanto um elemento do tipo TCartas. Há que investigar esses dois métodos, um deles deverá utilizar o vector poker, que contem os textos dos tipos de mãos.
    3. No método SaveSolution encontra-se a utilização do vector poker, com o argumento sendo a chamada a um método Poker(). Este método deve portanto retornar o tipo de mão que um objecto TCartas tem.
    4. Há agora que localizar a linha de código em que o vector m ficou actualizado com as mãos de cada jogador. Colocando o cursor no código da linha 5.a.i, basta procurar a anterior referência a "m[" (Ctrl+Shift+I » m[, para continuar a recuar, continuar a carregar em Ctrl+Shift+I). Onde a variável aparecer numa situação de acesso não interessa, queremos a última alteração, o que só pode acontecer através de uma atribuição ou chamada a um método. A primeira destas alterações é na linha: m[i].Poker();
    5. Este método já conhecemos, é o que devolve o índice da mão, os comentários e código envolvente clarificam que se localizou o local correcto, pelo que pode-se fazer a alteração da estrutura pretendida, a actualização do contador: contador[m[i].Poker()]++;
6. Utilização dos dados. No método indicador basta retornar o valor do contador com base no indicador recebido, testando eventualmente primeiro para ver se o indicador está dentro dos limites do vector: if(indicador>=0 && indicador<10) return contador[indicador];
7. Compilar e Testar: há que repetir o teste, e utilizar a interface para obter o resultado pretendido. Analisando o texto de ajuda, verifica-se que para pedir o indicador 3 tem que se escrever: get indicator 3. O resultado é a descrição da ajuda novamente, e lendo com mais atenção verifica-se que há um comando para saber o número de indicadores: get indicators. O resultado é 0. Falta portanto definir os 10 indicadores que fizemos.
8. Corrigir. Procurando no código onde é indicado o número de indicadores, acabaria-se por encontrar na função main esse código, em que o número de indicadores é: engine->indname.Count(). O objecto engine é do tipo TEngine, pelo que seguindo para a definição da variável obtem-se na classe a informação que os dois tvectors de strings indname e inddesc, têm os nomes e descrições dos indicadores. Estes dados têm de ser inicializados, e não foram, pelo que tem que se ir novamente ao método TCartasEngine::Reset para inicializar também estes vectores com o tipos das mãos. Bastará reutilizar o ciclo já feito e incluir também a adição de

novas linhas (objectos TVector estão a ser utilizados no mesmo método que serve de exemplo, para adicionar um novo elemento basta chamar o método Add): indname.Add(poker[i]); inddesc.Add(poker[i]);. Se se quiser pode-se fazer um novo vector com as descrições dos tipos das mãos.

9. Compilar e Testar: repetir o teste e após terminar efectuar o comando: get indicator 1, get indicator 2, etc. Verificar que o número devolvido coincide com o número de tipos de mãos obtidos pelos jogadores no teste. Fazer um segundo teste com maior volume, por exemplo:
    1. set information 0
    2. set limit states 1000
    3. run
    4. (aguardar que acabe)
    5. get indicator 1
    6. get indicator 2
10. Há evidência que a alteração foi correctamente bem feita, o próximo passo é enviar o código a alguém que o teste.

Quantas alterações foram necessárias?

- Criar o contador em TCartasEngine
- Inicializar o contador (e posteriormente os indicadores) em TCartasEngine::Reset
- Actualizar o contador em TCartasEngine::Run, alterando uma linha de código
- Retornar o resultado do contador em TCartasEngine::Indicator

"Software Engineering, theory and practice", second edition, Prentice Hall, Shari Pfleeger, pág. 307, 319-320

5 - Testes

5.1 Introdução

A fase de teste não se destina a provar que o código está correcto, uma vez que tal não é possível. O objectivo da fase de teste é encontrar falhas, e por mais cuidadoso que os programadores tenham sido na fase do código, juntamente com os intervenientes anteriores nas fases de especificação e desenho, as falhas vão existir. No entanto não se pode efectuar as restantes fases de forma relaxada, assumindo que na fase de testes qualquer problema será encontrado. Se assim acontecer, a fase de testes encontrará demasiadas falhas, e o custo do projecto será muito superior.

Os testes são feitos de forma a verificar se os requisitos e o desenho são satisfeitos. Podem ser feitos testes de unidades, a cada componente, e testes de integração a diversos componentes, sendo feito um teste final a todo o sistema. Ao encontrar uma falha há que clarificar as condições em que a falha ocorre, para que possa ser repetida e seja relativamente simples de localizar o componente defeituoso. Por esse motivo, manda o bom senso efectuar testes de unidades e só depois os testes envolvendo mais componentes, de forma a ser facilmente localizada a origem da falha. Após estes testes terem sido realizados com sucesso, que são testes de funcionalidades, há que fazer testes de performance, e testes de aceitação. Neste capítulo serão vistas técnicas e boas práticas para a realização de testes.

5.1.1 Classificação Ortogonal de Defeitos

Há duas grandes classes de erros, os sintácticos e os semânticos. Os erros sintácticos são os que derivam da escrita incorrecta da linguagem. Por exemplo, na linguagem C declarar uma função dentro de outra função ou abrir um bloco de código e não o chegar a fechar. Este tipo de erros são detectados no próprio editor do ambiente de desenvolvimento, sendo os de menor importância. Os erros semânticos são erros que apesar de não inviabilizarem o código como um código válido na linguagem de programação em uso, vão fazer com que o código possa não fazer o que é suposto. Por exemplo, utilizar uma variável sem a inicializar, ou ao ordenar um vector deixar o primeiro elemento por ordenar. Este tipo de erros são mais difíceis de apanhar, e é aqui que se centra a nossa preocupação.

Antes começar à procura de defeitos, podemos questionar como são os defeitos? Do que é que andamos à procura? A IBM construiu uma classificação ortogonal para os defeitos, de forma a conhecer-se um pouco mais do que se procura nesta fase, e tentar desta forma orientar os testes:

- Função - defeito que afecta funcionalidades do sistema
- Interface - defeito na interface do utilizador
- Validação - defeito na validação da correcção dos dados correctamente antes de estes serem utilizados
- Atribuição - defeito na estrutura de dados ou inicializações
- Sincronização - defeitos que envolvam recursos em tempo real
- Documentação - defeitos na documentação
- Algoritmo - defeito na eficiência e correcção do algoritmo ou estruturas de dados

Adicionalmente, um defeito diz-se de omissão caso seja derivado de algum aspecto em falta no código (exemplo: variável não inicializada). Um defeito diz-se de não omissão caso seja derivado de algum aspecto errado no código (exemplo: variável inicializada para o valor errado).

Os defeitos de omissão detectados automaticamente pelos ambientes de desenvolvimento, são tipicamente a não inicialização de variáveis, e a não utilização de uma variável após atribuir um valor.

Nestes casos faltará ou uma inicialização ou uma utilização da variável atribuída. Os defeitos de não omissão são a maioria dos erros detectados, tipicamente os sintacticamente incorrectos.

5.1.2 Organização dos Testes

Os testes são feitos a diversos níveis, e por diferentes intervenientes. O primeiro teste é feito pelo programador na fase de implementação. Actualmente os ambientes de desenvolvimento permitem ter o código sempre pronto a correr, de forma a se ir testando o que se vai fazendo. Desta forma é fácil após a escrita de um método verificar se está a funcionar. Um teste maior será feito pelo próprio programador após toda a classe estar implementada. Assim o programador pode detectar problemas cedo e corrigir de imediato, uma vez que sabe que a origem do problema é muito provavelmente no código que acabou de escrever. Um teste a uma classe ou módulo, chama-se teste de módulos, ou teste de componentes, ou ainda teste de unidades. Deve ser realizado também por uma pessoa diferente da que implementou o módulo.

Os componentes vão ficando prontos, e após estes terem passado o teste de unidades, há que executar um teste de integração envolvendo mais que um componente, de forma a verificar que os diversos componentes interagem da forma correcta.

Após os testes de integração há que efectuar testes a todo o sistema. O primeiro é o teste funcional que consiste em testar o sistema para verificar se está de acordo com os requisitos funcionais. De seguida há que efectuar o teste de performance, de forma a verificar os requisitos não funcionais. O teste seguinte é o de aceitação, que envolve o cliente e verifica os requisitos do cliente. Finalmente instala-se o sistema no seu local final e efectua-se um teste de instalação com utilizadores finais.

Uma falha em qualquer documento pode ser encontrada em qualquer altura. Salienta-se o facto de os primeiros documentos a produzir serem os últimos a testar, e portanto a verificar. O pior que poderia acontecer seria a documentação técnica de requisitos não estar coerente com os requisitos do cliente. Caso a incoerência não fosse detectada antes seria apenas no teste de aceitação, ou seja, numa fase muito terminal do projecto, o que poderia obrigar a refazer tudo de novo. É portanto natural que seja atribuída a responsabilidade das fases iniciais do projecto às pessoas com mais experiência.

Quando o componente é testado por outra pessoa, ao encontrar uma falha a preocupação deverá ser a de conseguir repetir a falha e reportá-la para que esta seja corrigida, e não propriamente de culpar a pessoa que desenvolveu o componente.

5.2 Tipos de Testes

5.2.1 Teste de Unidade

Este teste é também chamado de teste de módulo ou teste de componentes. Há basicamente três testes de unidades que se pode fazer:

- Revisão de código;
- Provas matemáticas;
- Testes empíricos.

A primeira via permite envolver recursos humanos no teste, tirando assim proveito em que pessoas experientes detectam falhas muito rapidamente, mas por outro lado têm um custo elevado. A segunda via requer que quem faça o teste tenha muito tempo e bases matemáticas, e estas provas têm também de ser verificadas, uma vez que também podem ter erros. A terceira via é a mais utilizada, também

complementar das anteriores. Consiste em efectuar testes empíricos através de execuções do código em teste, e comparar os argumentos com os resultados de forma a verificar se os resultados estão correctos. O teste analisa uma amostra das possíveis execuções que o código pode originar, permitindo assim verificar se funciona bem nessa amostra e por outro lado efectuar desde logo testes de performance. Convém que a amostra seja representativa da população, pelo que a escolha dos dados de teste é determinante. A dimensão da amostra será dependente do tempo de execução de cada corrida e do tempo disponível para todo o teste do componente.

Uma de duas filosofias pode ser utilizada na escolha de dados de teste: caixa preta (ou caixa fechada); caixa branca (ou caixa aberta). No teste efectuado como caixa preta os dados de teste são escolhidos sem fazer uso da implementação do objecto de teste, e são normalmente gerados aleatoriamente. O teste em modo de caixa branca consiste em escolher os argumentos com base na análise da estrutura interna do componente, permitindo assim escolher uma amostra mais representativa de todas as execuções que podem ser efectuadas com o componente.

Os argumentos do teste são o conjunto de todas as variáveis que se tem de inicializar para testar o componente. No caso de serem muitas variáveis torna-se complicado testar todas as combinações. Normalmente é preferível em caixa preta seleccionar uma distribuição para cada variável. Caso as variáveis tenham um valor mínimo e máximo pode-se utilizar a distribuição uniforme, mas caso não tenham limite pode-se utilizar por exemplo a distribuição exponencial negativa. Esta distribuição tem probabilidade de gerar um valor muito elevado, muito embora tenha mais probabilidades de gerar um valor perto do zero. Tem uma forma simples de ser gerada a partir da distribuição uniforme: -log(Uniforme(0;1))

Em caixa branca o código é analisado e pode-se querer garantir que durante a execução de todo o teste as diversas situações são testadas:

- Teste de comandos: cada comando é executado pelo menos uma vez;
- Teste de ramificações: em cada ponto de decisão no código, cada ramo é tomado pelo menos uma vez;
- Teste de caminhos: cada sequência de comandos (caminho) no código é executada pelo menos uma vez;
- Teste de todas as utilizações: pelo menos um caminho da definição a todas as utilizações da variável tem de ser executado pelo menos uma vez;
- Teste de caminhos definição-utilização: para todas as definições de variáveis e utilizações de variáveis, todos os caminhos da definição à utilização da variável, devem ser executados pelo menos uma vez.

Se o código for complexo poderá ser difícil obter os dados de entrada para se garantir que se verifique qualquer uma destas situações. Pode-se naturalmente combinar a caixa branca com a caixa preta, utilizando parte do teste valores que garantem por exemplo que todos os comandos são executados, e o resto dos valores gerados aleatoriamente.

5.2.2 Revisão de Código

Uma revisão de código é um de teste de unidade em que o código e a documentação associada são analisados por um ou mais especialistas, de forma a se encontrarem erros de qualquer natureza. O objectivo é descobrir erros, não de os corrigir.

Uma revisão de código pode ser de "walkthrough" ou inspecção de código. No "**walkthrough**" quem escreve o código tem de apresentar à equipa de revisão o seu código, conduzindo a discussão de forma

informal, centrando-se a atenção no código e não no programador.

Na **inspecção de código**, a equipa de revisão tem de preparar uma lista de preocupações as quais tem de as verificar sobre a documentação e código, por exemplo: definição da estrutura de dados está consistente com a documentação de desenho; os comentários no código estão consistentes com o código que comentam; estimar a complexidade temporal e espacial do código. O próprio programador pode não fazer parte da equipa de revisão.

As revisões de código permitem a detecção de erros mais cedo, e portanto reduzindo a influência desses erros que teriam um custo muito maior para serem identificados mais tarde, e também um maior custo de correcção. Muitas das vezes não se fazem revisões de código porque as equipas de desenvolvimento de software são pequenas, ou todas as pessoas estão atarefadas, ou ainda por ser muito cansativo analisar código feito por outros. O ganho de revisões de código pode no entanto ser muito elevado, se os programadores passarem a ter mais cuidado no código que escrevem, uma vez que vai ser analisado por outros, e respectiva redução no número de erros.

5.2.3 Testes de Integração

Os diversos componentes de um sistema interagem uns com os outros, criando uma rede de dependências. Como há normalmente muitos componentes, é essencial que exista uma estratégia de testes de integração de forma a testar subsistemas antes de todo o sistema, caso contrário não só aparecem muitas falhas em simultâneo, como se torna difícil localizar a origem das falhas.

A primeira estratégia de teste é a de baixo-para-cima. Nesta estratégia cada componente que é completamente independente dos restantes, é inicialmente testado separadamente. De seguida, os componentes cujos seus componentes dependentes já estão testados, devem ser testados utilizando esses componentes. O processo repete-se até que não no fim todo o sistema será testado. Caso existam dois ou mais componentes que dependam mutuamente entre si, esses componentes devem ser testados em conjunto. Para testar um componente sem ser através dos componentes que o utilizam, é necessário programar uma rotina chamada "driver" (component driver), que irá fazer as inicializações necessárias para correr o componente com os dados de teste. Os drivers são normalmente simples de codificar, e a determinação dos dados de teste é debatida no teste de unidade. Esta estratégia assenta bem na programação orientada por objectos, mas tem o problema de as falhas mais importantes, as que ocorrem nos componentes que utilizam mais componentes, serem descobertas apenas no final. Estas falhas são provavelmente da fase de desenho, e devem ser descobertas o quanto antes, uma vez que podem implicar muito trabalho para as corrigir.

A segunda estratégia de teste é a de cima-para-baixo. Nesta estratégia começa-se por testar o componente mais geral, que não é utilizado por nenhum outro, e de seguida os componentes que são utilizados apenas por componentes já testados. O ciclo continua até que todos os componentes estejam testados. Caso existam dois ou mais componentes que dependam mutuamente entre si, esses componentes devem ser testados em conjunto. Para testar um componente sem ter os componentes que utiliza testados, tem de se construir um "stub" por cada componente utilizado, de forma a simular a sua actividade. A simulação do componente tem de possibilitar os diversos tipos de interacções, pelo que não deve ser algo constante. Os "stubs" são normalmente mais difíceis de construir, e em certos sistemas podem ser requeridos demasiados "stubs" o que pode inviabilizar esta estratégia. A vantagem é de testar o que é mais importante primeiro, permitindo descobrir os erros que podem originar mais mudanças mais cedo. Uma alteração a esta estratégia, consiste em testar sempre cada componente isoladamente antes de ser testado em conjunto. Neste caso são necessários tanto "stubs" como "drivers".

A estratégia "Big-bang" consiste em testar cada componente isoladamente primeiro, e de seguida testar

todo o sistema. Esta estratégia necessita de “stubs” e “drivers” e tem a desvantagem de ser complicado localizar a origem das falhas no teste geral. É possível utilizar-se esta estratégia apenas em pequenos sistemas.

A estratégia “Sandwich” consiste em juntar as estratégias de baixo-para-cima e de cima-para-baixo. Pode-se aplicar uma das estratégias apenas até um determinado nível, e a outra encarrega-se de testar o resto, e desta forma tem-se o melhor de dois mundos. Não se deve descer muito com a estratégia de cima-para-baixo para não ter que codificar uma grande quantidade de “stubs”, mas convém descer até um nível que permita detectar falhas no desenho do sistema.

5.2.4 Teste de Performance

Os testes de unidade e testes de integração são direccionados a verificar se o código está de acordo com o desenho. O teste do sistema destina-se primeiramente a verificar se todo o sistema está de acordo com os requisitos funcionais, de seguida efectuam-se testes de performance para verificar os requisitos não funcionais, seguindo-se o teste de aceitação envolvendo o cliente e teste de instalação no ambiente do utilizador final.

Os testes de performance devem ser efectuados conforme os requisitos não funcionais. Os mais comuns são:

- Teste de stress – o sistema é testado até aos limites especificados, num pequeno espaço de tempo, simulando um pico de utilização;
- Teste de volume – grande volume de dados é fornecido ao sistema, de forma a verificar se responde correctamente quando tem demasiados dados;
- Teste de velocidade – verifica os tempos de resposta do sistema;
- Teste de configuração – são feitos testes com diferentes configurações no software e no hardware;
- Teste de compatibilidade – são executados se o sistema necessitar de interagir com outros sistemas;
- Teste de regressão – necessário quando o sistema substitui outro sistema. Tem que se assegurar que as funcionalidades anteriores estão implementadas no novo sistema;
- Teste de segurança – verifica a segurança do sistema.

Estes testes são chatos de se fazerem, e muitas das vezes são substituídos pela garantia do profissionalismo e competência da empresa que desenvolve o software. Não apresenta ao cliente relatório de testes algum, o próprio cliente não pediu este relatório, apenas quer o sistema a funcionar e contenta-se com uma demonstração do sistema. Se o sistema falhar durante a demonstração, haverá certamente um conjunto de explicações que o cliente não percebe e por isso fica satisfeito.

À medida que o tempo passa, quando o sistema for sendo cada vez mais utilizado o cliente vai-se apercebendo do que está mal no sistema, através das queixas dos utilizadores: "às X horas, quando todos tentam aceder o sistema não responde", "não funciona quando se quer introduzir uma ficha com mais de Y campos", "o sistema é lento", "não funciona em computadores com o Sistema Operativo W", "após a última actualização, já não se consegue introduzir uma ficha do tipo K, nem imprimir", "o utilizador convidado, mesmo sem permissões consegue ver e editar as fichas". Nessa altura o cliente paga à empresa a manutenção que o sistema necessitar.

A imagem da empresa provavelmente não é posta em causa. O sistema simplesmente é muito complexo, e durante os testes não se poderiam prever todas as situações. Esta frase é correcta, mas não invalida que se façam estes testes e se apresente um relatório de testes ao cliente para o comprovar. Sem estes testes o sistema iniciar-se-á com problemas que poderiam ser detectados pela equipa de

desenvolvimento de software.

5.2.5 Teste de Aceitação

Os testes de aceitação são feitos de acordo as preferências do cliente. O cliente pode fazer um conjunto de testes a efectuar, com as situações tipo em que o sistema será utilizado, de forma a verificar não só que o sistema funciona bem como também a avaliar a performance do sistema. Normalmente efectua-se também **testes piloto**, em que os utilizadores utilizam o sistema nas condições finais. Caso os utilizadores sejam da empresa, o teste piloto chama-se de **teste alfa**, se os utilizadores forem os finais, chama-se **teste beta**.

Após o teste de aceitação terminar, o cliente deverá dizer quais os requisitos que não estão satisfeitos, e quais os que devem ser apagados, revistos ou adicionados.

5.2.6 Documentação de Teste

O volume da documentação de teste deve ter uma dimensão adequada aos testes produzidos e ao sistema testado. Um sistema de média grande dimensão deve ter os documentos:

- Plano de teste - deve descrever o sistema e os testes a efectuar para verificar os requisitos;
- Especificação de teste - deve conter os requisitos testados e como são testados;
- Descrição de teste – deve definir os dados de teste e os procedimentos de teste;
- Relatório de teste – deve descrever os resultados obtidos.

5.3 Quando parar de testar

5.3.1 Alternativas

Não havendo forma de saber se um componente tem ou não erros levanta-se a questão de saber quando parar de testar. Essa decisão deve ser de gestão, uma vez que continuar os testes incorre-se em mais custos de recursos humanos, mas aumenta-se a fiabilidade do software. Para suportar essa decisão há que conseguir estimar o número de falhas e o grau de confiança no software de alguma forma.

Uma hipótese é inserir falhas no software, que deve ser feita por alguém da equipa de teste, mas que não participa no teste. Inserindo-se por exemplo S=20 falhas, caso a equipa de teste encontre s=15 dessas falhas e mais n=10 falhas diferentes, então pode-se estimar o número total de falhas N pela fórmula: $N = \frac{Sn}{s}$, assumindo que $\frac{N}{n} = \frac{S}{s}$. Neste caso daria $N = \frac{20*10}{15} = 13,3$ faltas, faltando portanto encontrar 3,3 falhas. O problema deste método é que tem que se inserir muitas falhas para ter algum grau de precisão, e as falhas devem ser da mesma natureza das que estão por descobrir. Há também um aumento no custo de teste.

Uma segunda hipótese é utilizar duas equipas de teste independentes, descobrindo cada equipa um número de falhas f_1 e f_2, das quais algumas falhas são iguais f_{12}. Pode-se assumir que a eficiência de cada equipa relativamente a descobrir o total de falhas n, será de $E_1=f_1/n$ e $E_2=f_2/n$. Pode-se estimar a eficiência de uma equipa a descobrir as falhas que a outra equipa descobriu como sendo $E_1=f_{12}/f_2$ e $E_2=f_{12}/f_1$, e portanto estimar o número total de falhas através da fórmula $n=f_{12}/(E_1*E_2)$. Nesta solução o custo aumenta porque há falhas descobertas por ambas as equipas, mas não duplica propriamente, uma fez que todas as pessoas envolvidas estão a testar o software. Notar no entanto que as equipas nunca podem ser completamente independentes, uma vez que os "testers" têm formações idênticas, e acabam por ter a tendência de descobrir as mesmas falhas.

Uma terceira hipótese é utilizar o modelo Motorola, que nos permite saber o número de horas que se tem de testar sem que nenhuma falha seja encontrada para garantir que o código apenas tem um determinado número de falhas. O modelo tem também assumpções, pelo que o valor aconselhado é meramente indicativo, e não propriamente uma certeza.

5.3.2 Modelo Motorola

O modelo Motorola permite estimar o número de horas de teste necessárias, sem que ocorram falhas, para que se tenha no código apenas um especificado número de falhas. O modelo necessita do número de falhas encontradas até ao momento (*falhas_identificadas*), e do número de horas até à última falha (*horas_até_última_falha*).

O número de horas necessárias é dada pela seguinte fórmula:

$$\frac{\ln(falhas/(0.5+falhas))\times horas\_até\_última\_falha}{\ln((0.5+falhas)/(falhas\_identificadas+falhas))}$$

Exemplo:

Uma falha acabou de ser encontrada, num teste que decorre há 80 horas, sendo já a 10ª falha encontrada. Foi decidido que apenas 1 falha deverá ficar por identificar, e pretende-se saber quanto tempo se terá de testar sem que apareçam novas falhas para assumir que apenas existe uma falha por descobrir.

Com estes dados podemos aplicar o modelo Motorola: ln(1/1.5)x80/ln(1.5/11) = ln(0.67)x80/ln(0.14). Como não é permitido calculadoras nos exames, é fornecida a tabela do logaritmo (ver em baixo), pelo que o número de horas necessário é de: 0.4x80/1.97=16 horas.

ln(x)

x	0	0,01	0,02	0,03	0,04	0,05	0,06	0,07	0,08	0,09
1	0,000	0,010	0,020	0,030	0,039	0,049	0,058	0,068	0,077	0,086
0,9	-0,105	-0,094	-0,083	-0,073	-0,062	-0,051	-0,041	-0,030	-0,020	-0,010
0,8	-0,223	-0,211	-0,198	-0,186	-0,174	-0,163	-0,151	-0,139	-0,128	-0,117
0,7	-0,357	-0,342	-0,329	-0,315	-0,301	-0,288	-0,274	-0,261	-0,248	-0,236
0,6	-0,511	-0,494	-0,478	-0,462	-0,446	-0,431	-0,416	-0,400	-0,386	-0,371
0,5	-0,693	-0,673	-0,654	-0,635	-0,616	-0,598	-0,580	-0,562	-0,545	-0,528
0,4	-0,916	-0,892	-0,868	-0,844	-0,821	-0,799	-0,777	-0,755	-0,734	-0,713
0,3	-1,204	-1,171	-1,139	-1,109	-1,079	-1,050	-1,022	-0,994	-0,968	-0,942
0,2	-1,609	-1,561	-1,514	-1,470	-1,427	-1,386	-1,347	-1,309	-1,273	-1,238
0,1	-2,303	-2,207	-2,120	-2,040	-1,966	-1,897	-1,833	-1,772	-1,715	-1,661

ln(0,71)=ln(0,7+0,01)=-0,342

Exemplo: Poker

Vamos testar no nosso exemplo o método TCartas::Poker, dado que é o método mais complexo, assumindo mãos de 7 cartas. O teste deve verificar se os diferentes tipos de mãos são identificados correctamente.

Caso não tenha feito as actividades anteriores, o código de partida é o Cartas.zip. Se quiser pode fazer os testes apenas com esta informação, e continuar a ler apenas quando acabar, ou aceitar algumas sugestões lendo o próximo parágrafo.

Sugestão: faça um gerador de mãos de determinados tipos, e chame o método para verficar se a resposta é correcta. Por exemplo, para gerar uma mão com um par, retire de um baralho um par gerado

aleatoriamente e adicione-o à mão, e de forma a não haver mais pares nem trios, retire as restantes duas cartas do mesmo número do baralho, e retire também três cartas aleatoriamente de cada um dos restantes números. Para evitar sequências, retire um dos números 5 ou 6 e o número 10 ou V respectivamente. Baralhe as restantes cartas e junte à mão as 5 primeiras. Finalmente, para evitar a cor, conte quantas cartas tem de cada naipe na mão, e no caso de ter 5 ou mais troque o naipe de uma das cartas que não fazem parte do par.

Resolução

Pretende-se testar o método TCartas::Poker, pelo que tem que se ter:

1. Um gerador de mãos (das 10 tipos de mãos);
2. O método TCartasEngine::Run tem que utilizar o gerador para gerar as mãos específicas, e verificar se o valor retornado pelo método TCartas::Poker é igual.

O trabalho resume-se a alterar o método TCartasEngine::Run e a criar um método para gerar as mãos, portanto dois métodos.

Pode-se começar por criar o método para gerar as mãos em TCartasEngine::GerarMao(TCartas&mao,int tmao,int ncartas);

No método TCartasEngine::Run, há que apagar tudo e gerar as mãos por ordem, verificando se estão correctas:

```
void TCartasEngine::Run()
{
   Seed();
   // processar todos os tipos de mãos
   for(int i=0;i<10;i++) {
      // para cada mão testar o número especificado de vezes
      for(long j=0;j<States();j++) {
         // gerar uma mão
         TCartas mao;
         GerarMao(mao,i,7);
         int resultado=mao.Poker();
         if(resultado!=i) {
            // erro, a mão gerada não coincide com a calculada
            printf("\n%s: #%d (%s)",poker[i],j,poker[resultado]);
            mao.Save(stdout);
         }
      }
   }
}
```

O gerador de mãos é mais complexo, tem que se ir desenvolvendo. Foi implementada uma filosofia diferente da sugestão no enunciado, que consiste no seguinte:

- Começar com um baralho com ordem aleatória;
- Ir retirando as cartas que não são necessárias para a mão que se quer, bloqueando as que se quiser manter;
- No final deitar fora as cartas a mais.

Este método acaba por ter duas zonas, as mãos com numerações idênticas, e as mãos com cor e sequência. O código torna-se um pouco mais complexo porque tem que se impedir que na geração de um par não apareça por engano um trio, ou não apareça cor. O código completo da função está comentado de forma a ser melhor interpretado.

```
void TCartasEngine::GerarMao(TCartas&mao,int tmao,int ncartas)
{
   TVector<int> bloquear;
   int estado=0;
   mao.Baralho();
   mao.cartas.RandomOrder();

   if(tmao==0 || tmao==1 || tmao==2 || tmao==3 || tmao==6 || tmao==7)
   { // nada;  par;  2pares;  trio;  fullen;  poker
      for(int i=1;i<mao.cartas.Count();i++) {
         if(estado==tmao && i>ncartas-1)
            break;
         if(i>=4 && tmao!=6 && tmao!=7) {
            // impedir cor
            int count=1;
            for(int j=0;j<i&&j<ncartas;j++)
               if(TC_NAIPE(mao.cartas[i])==TC_NAIPE(mao.cartas[j]))
                  count++;
            if(count>=5) {
               mao.cartas.Delete(i--);
               continue;
            }
            // impedir sequência
            int de;
            bool seq=false;
            for(de=TC_NUMERO(mao.cartas[i])-4;de<=TC_NUMERO(mao.cartas[i]);de++) {
               int j=de;
               for(j=de;j<de+5;j++) {
                  int l;
                  for(l=0;l<=i;l++)
                     if(TC_NUMERO(mao.cartas[l])==j)
                        break;
                  if(l==i+1) break; // não há este número, não há problema
               }
               if(j==de+5) { // sequência
                  mao.cartas.Delete(i--);
                  seq=true;
                  break;
               }
            }
            if(seq) continue;
         }

         for(int j=0;j<i&&j<ncartas;j++)
            if(TC_NUMERO(mao.cartas[j])==TC_NUMERO(mao.cartas[i])) {
               if(   // o estado requerido já foi atingido
```

```
                    estado==tmao ||
                    estado>0 && (
                    // ou quer-se dois pares e estava para vir um trio
                    tmao==2 && bloquear.Find(j)>=0 ||
                    // ou quer-se um trio e estava para vir dois pares
                    (tmao==3 || tmao==7) && bloquear.Find(j)<0 ||
                    // ou quer-se fullen e estava para vir três pares
                    tmao==6 && estado==2 && bloquear.Find(j)<0))
                {
                    mao.cartas.Delete(i--);
                } else {
                    // aceitar esta carta se:
                    if(estado==0 ||
                        (tmao==3 || tmao==7) && bloquear.Find(j)>=0 ||
                        tmao==2 && bloquear.Find(j)<0    ||
                        tmao==6 && (estado<=1 ||
                          estado==2 && bloquear.Find(j)>=0 ||
                          estado==3 && bloquear.Find(j)<0))
                    {
                        if(estado==0) estado=1;
                        else if(estado==1)
                            estado=(bloquear.Find(j)<0?2:3);
                        else if(estado==2) estado=6;
                        else if(estado==3)
                            estado=(bloquear.Find(j)<0?6:7);
                        if(i>ncartas-1) {
                            // arranjar uma carta para trocar
                            for(int l=0;l<ncartas&&l<i;l++)
                                if(l!=j && bloquear.Find(l)<0) {
                                    bloquear.Add(j);
                                    bloquear.Add(l);
                                    mao.cartas[l]=mao.cartas[i];
                                    mao.cartas.Delete(i--);
                                    break;
                                }
                        } else {
                            bloquear.Add(i);
                            bloquear.Add(j);
                        }
                    } else {
                        // não aceitar a carta
                        mao.cartas.Delete(i--);
                    }
                }
                break;
            }
        }
    } else if(tmao==4 || tmao==5 || tmao==8 || tmao==9) {
        // sequência, cor, sequência + cor, sequência + cor real
        for(int i=0;i<mao.cartas.Count();i++) {
```

```
            // coloca as primeiras 5 cartas com o requerido, as outras são à sorte
            if(    // apagar todas as cartas de naipe diferente
               i<5 && (tmao==5 || tmao==8 || tmao==9) &&
               TC_NAIPE(mao.cartas[i])!=TC_NAIPE(mao.cartas[i-1]) ||
               // apagar todas as cartas baixas
               i<5 && tmao==9 && TC_NUMERO(mao.cartas[i])<8 ||
               // apagar os ases para impedir a sequência + cor real
               tmao==8 && TC_NUMERO(mao.cartas[i])==12 ||
               // não deixar sequência passar a cor em 5 cartas
               i==1 && tmao==4 && TC_NAIPE(mao.cartas[0])==TC_NAIPE(mao.cartas[1]) ||
               // não deixar cor passar a sequência em 5 cartas
                                    i==1 && tmao==5 && abs(TC_NUMERO(mao.cartas[0])-
TC_NUMERO(mao.cartas[1]))<=4)
            {
               mao.cartas.Delete(i--);
            } else if(tmao==4 && i>=5) {
               // não deixar a sequência passar a cor
               int count=1;
               for(int j=0;j<i&&j<ncartas;j++)
                  if(TC_NAIPE(mao.cartas[i])==TC_NAIPE(mao.cartas[j]))
                     count++;
               if(count>=5) {
                  mao.cartas.Delete(i--);
                  continue;
               }
            } else if(tmao==5 && i>=5) {
               // não deixar a cor passar a sequência
               int de;
               bool seq=false;
               for(de=TC_NUMERO(mao.cartas[i])-4;de<=TC_NUMERO(mao.cartas[i]);de++) {
                  int j=de;
                  for(j=de;j<de+5;j++) {
                     int l;
                     for(l=0;l<=i;l++)
                        if(TC_NUMERO(mao.cartas[l])==j)
                           break;
                     if(l==i+1) break; // não há este número, não há problema
                  }
                  if(j==de+5) { // sequência
                     mao.cartas.Delete(i--);
                     seq=true;
                     break;
                  }
               }
               if(seq) continue;
            } else if(i<5 && (tmao==4 || tmao==8 || tmao==9))
               for(int j=0;j<i;j++)
                  // apagar todos os números que distam mais de 4 dos anteriores
                  if(TC_NUMERO(mao.cartas[j])==TC_NUMERO(mao.cartas[i]) ||
                     abs(TC_NUMERO(mao.cartas[j])-TC_NUMERO(mao.cartas[i]))>4)
```

```
                {
                    mao.cartas.Delete(i--);
                    break;
                }
            }
        }
        if(mao.cartas.Count()>ncartas)
            mao.cartas.Count(ncartas);
    }
```

Executar e correr. Verificar que não há mãos mal colocadas, mesmo fazendo 1000 testes para cada mão:

set seed 1
set limit states 1000
run
Run end.

Para verificar da necessidade de impedir, por exemplo, a cor na geração das cartas múltiplas, pode-se comentar a zona de código correspondente (após o comentário "impedir cor") e correr novamente:

nada: #7 (cor)[♥8 ♣6 ♠AR543]
nada: #38 (cor)[♥RD10864 ♠A]
nada: #39 (cor)[♥83 ♠AD1054]
nada: #77 (cor)[♥V4 ♦AD1076]
nada: #94 (cor)[♥2 ♣AV10843]
nada: #130 (cor)[♥4 ♦R10953 ♠V]
nada: #156 (cor)[♣R10 ♠V8652]
nada: #168 (cor)[♥4 ♦AD1086 ♣3]
nada: #198 (cor)[♦3 ♣D10874 ♠6]
nada: #250 (cor)[♥98432 ♣A ♠10]
nada: #288 (cor)[♦D10985 ♠63]
nada: #297 (cor)[♥D10754 ♦2 ♣A]
nada: #324 (cor)[♥V4 ♦AD1082]
nada: #351 (cor)[♦10 ♣RV942 ♠7]
nada: #385 (cor)[♥106 ♠ARV83]
nada: #402 (cor)[♥V4 ♠AD965]
nada: #405 (cor)[♥RDV65 ♦2 ♣7]
nada: #417 (cor)[♦RD864 ♣A ♠9]
nada: #434 (cor)[♦D10964 ♣5 ♠R]
nada: #448 (cor)[♦R ♣7 ♠D8632]
nada: #455 (cor)[♥V ♦9 ♣AR1086]
nada: #504 (cor)[♥108743 ♦AR]
nada: #522 (cor)[♣108642 ♠R3]
nada: #550 (cor)[♥7 ♣ARD1054]
nada: #557 (cor)[♦A ♣3 ♠D10752]
nada: #582 (cor)[♥4 ♦9 ♠ARD87]
nada: #585 (cor)[♥D ♣2 ♠ARV63]
nada: #586 (cor)[♥R ♣V7654 ♠9]
nada: #599 (cor)[♥9 ♦3 ♣ARV74]
nada: #618 (cor)[♦32 ♣RDV65]
nada: #679 (cor)[♥R9752 ♦3 ♠D]

nada: #694 (cor)[♦7 ♣V9432 ♠R]
nada: #703 (cor)[♥R8542 ♠A7]
nada: #745 (cor)[♦V9 ♣87532]
nada: #753 (cor)[♦AD852 ♣R10]
nada: #759 (cor)[♥RDV97 ♦8 ♣6]
nada: #772 (cor)[♥6 ♣2 ♠RDV73]
nada: #780 (cor)[♦A ♣V10952 ♠3]
nada: #783 (cor)[♦10 ♣A ♠V9763]
nada: #793 (cor)[♥D ♦10 ♣A8754]
nada: #798 (cor)[♥RV1096 ♦5 ♣4]
nada: #804 (cor)[♣DV1072 ♠86]
nada: #829 (cor)[♦R86542 ♣A]
nada: #852 (cor)[♦84 ♣V10953]
nada: #868 (cor)[♥A5432 ♠97]
nada: #869 (cor)[♥RD975 ♦3 ♠2]
nada: #879 (cor)[♥D10852 ♣7 ♠6]
nada: #885 (cor)[♥R ♦97542 ♠A]
nada: #914 (cor)[♦10 ♣DV764 ♠3]
nada: #925 (cor)[♥A ♣4 ♠109753]
nada: #926 (cor)[♣V10 ♠R8762]
nada: #929 (cor)[♦A2 ♠V8763]
nada: #932 (cor)[♦A4 ♣DV965]
nada: #980 (cor)[♥A ♦V ♣109762]
nada: #982 (cor)[♥6 ♦4 ♠A9853]
par: #109 (cor)[♥R9 ♣ARD75]
par: #155 (cor)[♥4 ♦R10987 ♠4]
par: #163 (cor)[♦A8 ♣108743]
par: #239 (cor)[♥4 ♣AD842 ♠10]
par: #245 (cor)[♥AR853 ♦7 ♣A]
par: #248 (cor)[♦5 ♠D109853]
par: #258 (cor)[♦AR1054 ♣V ♠V]
par: #281 (cor)[♥D ♣R7643 ♠4]
par: #316 (cor)[♦8 ♣10 ♠D10753]
par: #404 (cor)[♥V9754 ♣V ♠3]
par: #421 (cor)[♥DV986 ♦A ♠D]
par: #506 (cor)[♥7 ♣A10982 ♠2]
par: #517 (cor)[♥R ♦RDV94 ♠A]
par: #533 (cor)[♥V ♣D7652 ♠D]
par: #570 (cor)[♦AR864 ♣7 ♠4]
par: #606 (cor)[♥RD965 ♣6 ♠7]
par: #622 (cor)[♦R5 ♣RV1063]
par: #653 (cor)[♦D85432 ♠5]
par: #676 (cor)[♥AV986 ♣8 ♠D]
par: #694 (cor)[♦V ♣ADV98 ♠5]
par: #773 (cor)[♥V ♣A ♠AD1065]
par: #866 (cor)[♥8 ♦V ♠R10986]
par: #892 (cor)[♣D9 ♠AD654]
par: #920 (cor)[♥2 ♦RD6532]
par: #929 (cor)[♦83 ♣R8752]
par: #938 (cor)[♥D ♦A ♠D9654]

par: #954 (cor)[♥AR1042 ♦V ♠A]
2pares: #5 (cor)[♥D10932 ♦9 ♣10]
2pares: #104 (cor)[♥R ♦RV985 ♣8]
2pares: #155 (cor)[♥6 ♣V9864 ♠8]
2pares: #184 (cor)[♥D7432 ♣D7]
2pares: #189 (cor)[♥D8 ♦D10852]
2pares: #199 (cor)[♥V8 ♦V10984]
2pares: #325 (cor)[♥75 ♣A10975]
2pares: #418 (cor)[♥D10832 ♣108]
2pares: #471 (cor)[♥2 ♦4 ♣109742]
2pares: #515 (cor)[♥A ♣R ♠ARV42]
2pares: #833 (cor)[♥5 ♦AD852 ♠A]
2pares: #887 (cor)[♦V7 ♣V10973]
trio: #6 (cor)[♥5 ♦5 ♠AR753]
trio: #88 (cor)[♥R10962 ♣R ♠R]
trio: #190 (cor)[♦3 ♣3 ♠109643]
trio: #303 (cor)[♥AV532 ♣5 ♠5]
trio: #339 (cor)[♥V ♣V ♠ADV103]
trio: #341 (cor)[♥9 ♦9 ♣V9752]
trio: #463 (cor)[♥A10763 ♣10 ♠10]
trio: #527 (cor)[♥D ♣D ♠ADV84]
trio: #589 (cor)[♥DV653 ♣D ♠D]
trio: #639 (cor)[♥RD1073 ♦3 ♠3]
trio: #640 (cor)[♥R10653 ♣6 ♠6]
trio: #724 (cor)[♥5 ♣V10532 ♠5]
trio: #927 (cor)[♥AR982 ♣R ♠R]Run end.

Verifica-se a geração de "cor" acidentalmente quando se pretendia gerar mãos de diversos tipos, pelo que se confirma que o código para impedir a cor nas mãos de numerações idênticas é essencial.

O código não teve nenhum crash, e todas as mãos deram o mesmo resultado que a função geradora, confirmando assim o bom funcionamento do método TCartas::Poker.

Se fosse necessário maior grau de certeza, o passo seguinte, para além de aumentar o número de testes, é a revisão do método manualmente. Ter o teste operacional é muito útil nesta situação porque assim pode-se nessa fase comentar partes do código que não se compreenda de forma a ver se a ausência do código leva a resultados incorrectos.

- "Software Engineering, theory and practice", second edition, Prentice Hall, Shari Pfleeger, pág. 331-362, 371-374, 401-403, 410, 412-414, 417-426

6 - Testes Empíricos com o Engine Tester

A motivação para a construção do software de teste de componentes e a inclusão nesta unidade curricular, é por um lado a relevância que o teste de componentes tem na qualidade do software, e por outro lado por não existirem ferramentas específicas de testes de componentes no mercado.

Um componente bem testado pode ser utilizado com segurança tanto no próprio projecto como ser reutilizado em projectos posteriores. É um componente que irá sobreviver ao tempo. O que é o tempo extra para teste se se irá poupar um tempo indefinido a localizar e corrigir problemas que originem no componente que ficou por testar?

O EngineTester tem duas funcionalidades principais: efectuar testes empíricos em algoritmos e testar componentes. Podem ser analisados parâmetros do algoritmo, comparar vários algoritmos e um ou mais conjuntos de ficheiros de dados. Estas duas funcionalidades não são incompatíveis. Executar um algoritmo com diversos valores nos seus parâmetros com vista a conhecer quais os valores que o optimizam, não é incompatível a executar um algoritmo com diversos valores nos seus parâmetros com vista a saber se funciona bem em todos os valores. Depende dos indicadores que se calcular para cada corrida, podendo até ter-se ambos os objectivos em simultâneo: optimização e teste.

O Visual c++ 2008 Express Edition não tem ferramentas dedicadas a testes de componentes. Pode-se sempre fazer pequenos programas para testar cada componente, mas isso leva a que o custo dos testes aumente e que quem testa o componente tenha de construir código para o testar. Na solução de teste de componentes com o EngineTester, quem desenvolve um componente cria também um engine para correr sobre o EngineTester. Quem testa o componente não necessita olhar para o código mas pode efectuar testes de volume com qualquer número de parâmetros e indicadores de verificação, que se podem incluir na documentação interna do componente e se podem repetir a qualquer momento sem trabalho extra, por exemplo após uma actualização do componente.

O EngineTester permite encadeamento entre algoritmos, ou seja, o resultado de um algoritmo pode ser enviado para outro algoritmo. Todos os resultados podem ser exportados em texto, para serem posteriormente processados, mas pode-se no próprio EngineTester construir análises gráficas a partir dos resultados, sendo essas análises reconstruidas cada vez que os testes sejam repetidos. Eventualmente esta funcionalidade pode ser utilizada para testes de integração, mas como é compreensível este tipo de testes é mais complexo e depende da aplicação concreta.

Todas as parametrizações dos testes são armazenadas em ficheiro, de forma aos testes poderem ser repetidos facilmente. Assim, se se quiser repetir testes já efectuados devido a um melhoramento num dos algoritmos, ou repetir os testes numa nova máquina, ou simplesmente conferir resultados relatados por um colega, não é necessário passar pelo processo todo, basta mandar correr o documento previamente gravado, e todas as chamadas aos algoritmos com as respectivas parametrizações e ficheiros de dados, serão executadas novamente. Se os gráficos forem feitos no EngineTester, até os próprios relatórios são reconstruidos sem qualquer esforço.

Os algoritmos não determinísticos não inviabilizam o processo de repetição, dado que é sempre especificada uma semente. O gerador aleatório utilizado no algoritmo deverá produzir sempre a mesma sequência aleatória para a mesma semente. Caso seja encontrado um problema, as condições em que este ocorreu podem sempre ser repetidas.

O EngineTester não força os algoritmos a serem escritos numa linguagem própria, nem numa linguagem específica, apenas determina que sejam programas de linha de comando e que aceitem uma linguagem de comunicação de texto fixa, esta sim própria, de forma a que o EngineTester possa criar um processo e

comunicar com o algoritmo. O EngineTester está implementado em Windows, pelo que os algoritmos devem ser compilados para este sistema operativo.

A linguagem de comunicação pode ser utilizada de forma manual por um utilizador sem o EngineTester, ao mandar correr o executável correspondente ao algoritmo.

A linguagem de comunicação é definida neste documento, sendo ainda fornecida uma implementação em C++ de forma a que um programador apenas tenha de redefinir uma classe para implementar o algoritmo, sem ter de se preocupar com a linguagem de comunicação. Assim, um algoritmo já implementado pode ser facilmente adaptado para poder funcionar no EngineTester.

Algoritmos codificados em interpretadores que não possam criar executáveis de linha de comando, como é o caso do VBA do Excel, não podem ser utilizados no EngineTester. Aconselha-se no caso do VBA do Excel a fazer migração para o Visual Basic.

No resto da secção está a descrição textual do laboratório indicado nos vídeos, que pode ler em alternativa ou em complemento aos vídeos.

6.1 Construção de um algoritmo

Descreve-se neste ponto a forma de construir um algoritmo de raiz com a respectiva linguagem de comunicação, para funcionar com o EngineTester. A explicação é feita tendo como base um exemplo concreto para resolução do problema "Partition", cuja implementação em C++ é apresentada passo a passo.

6.1.1 Algoritmo (Engine)

Considere-se numa primeira abordagem, um algoritmo um conjunto de instruções para resolver um dado problema. Estas instruções actuam sobre os dados de entrada, produzindo em tempo finito os dados de saída.

Os dados de entrada podem ser de dois tipos:

- Dados do problema a resolver (instância ou dados de teste, em inglês instance);
- Afinações do algoritmo (que são identificados como parâmetros ou em inglês settings).

Os dados do problema são de natureza flexível: assim um problema de rotas terá informação da rede de estradas que se pode utilizar e um problema de afectação terá informação das entidades que se podem afectar e respectivos custos. Por esse motivo os dados de um problema são guardados num ficheiro, num formato próprio ao problema em questão e que o algoritmo deverá conseguir ler.

Os parâmetros são normalmente valores escalares, e em pequeno número. É a parte do algoritmo em que não há certeza do que é melhor para todos os problemas. Nessas situações, há que criar um parâmetro de entrada e definir um valor de omissão que funcione razoavelmente, mas com a possibilidade de variando o valor do parâmetro de entrada estudar qual a melhor opção e para que casos.

De referir que podendo a instância ser descrita por mais de um formato, poderá um dos parâmetros do algoritmo ser a escolha do formato de leitura e/ou escrita da instância.

Também os dados de saída podem ser de dois tipos:

- A solução do problema (que é identificada como solução ou em inglês solution)
- Conjunto de indicadores sobre a solução e a forma como esta foi obtida (que são identificados como indicadores ou em inglês indicators)

O formato de dados da solução é também muito dependente do problema a resolver. A solução pode ser gravada em ficheiro. Os indicadores serão tipicamente valores escalares calculados com base na solução. Por exemplo, num problema de optimização de rotas a solução será um trajecto, sendo um indicador possível o comprimento do trajecto.

Tal como no caso dos parâmetros, pode-se definir o número de indicadores que se pretender. Existem já alguns indicadores pré-definidos, nomeadamente:

- tempo de execução do algoritmo
- número de estados analisados
- nível máximo atingido

O valor dos indicadores, tal como o valor utilizado dos parâmetros do algoritmo, serão utilizados nas análises gráficas. Há que construir o maior número de indicadores possíveis, uma vez que é através deles que se visualizará os resultados. Seria impossível ver todas as soluções, mas facilmente se pode ver uma solução que tenha um determinado indicador muito alto (ou baixo). Convém naturalmente que estes indicadores sejam tão independentes quanto possível, de forma a apanharem aspectos diferentes da solução. A escala a que estes indicadores estão não é relevante, uma vez que se pode construir expressões com base nos indicadores e/ou parâmetros, podendo alterar a escala destes.

Em teste de componentes de software, no caso do algoritmo bloquear, facilmente se identifica as condições em que o erro ocorreu através dos parâmetros de entrada em utilização na altura. Podem-se também criar indicadores, que cuja a função seja verificar que a solução é válida, por forma identificar casos em que o algoritmo não funciona de acordo com as especificações.

Até agora centrámos a atenção em algoritmos para resolução de problemas, mas podemos considerar também algoritmos de outros tipos:

- Algoritmos que transformam instâncias de um problema em instâncias equivalentes de outro problema (interessante por exemplo em casos em que dispomos de um algoritmo para resolução de problemas do 2ºtipo e não do 1º)
- Algoritmos que geram instâncias de um problema (interessante para criar instâncias de teste automaticamente). Neste caso não há instância de entrada.

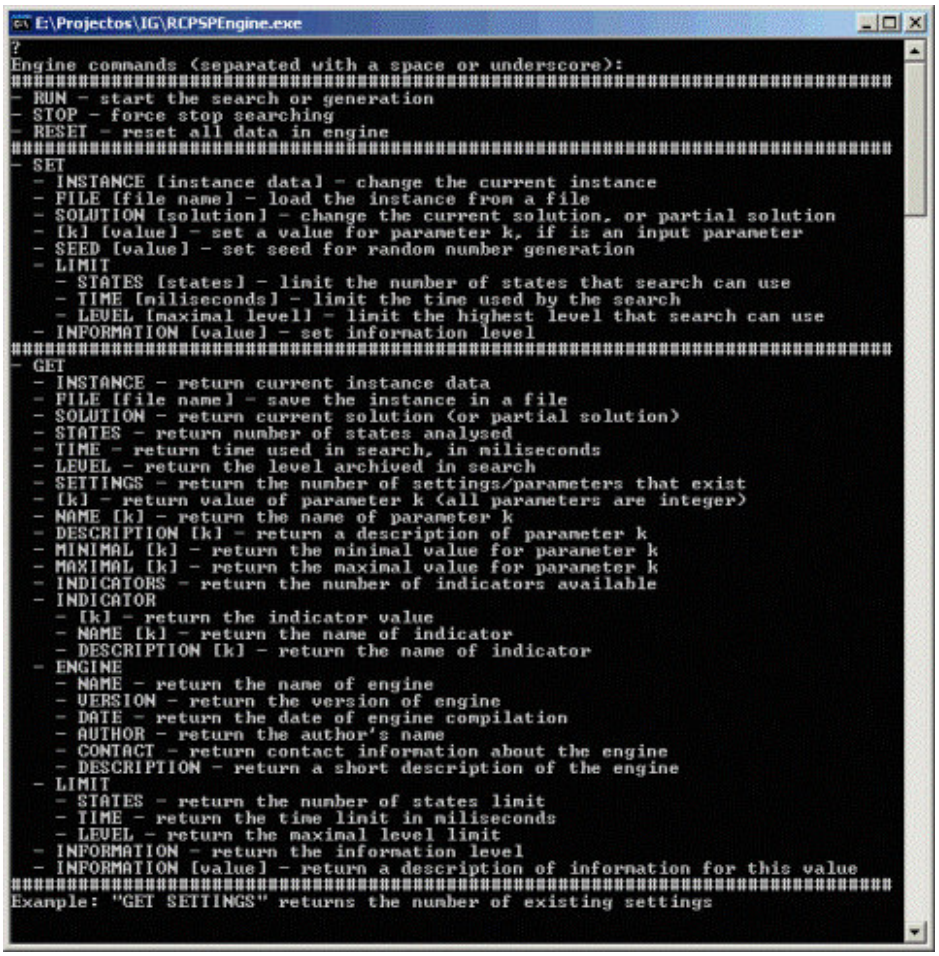
Em resumo podemos ter algoritmos de três tipos diferentes:

- Algoritmos para resolução de problemas
- Algoritmos para transformação de instâncias de um problema, em instâncias equivalentes de outro problema
- Algoritmos de geração de instâncias

6.1.2 Linguagem de Comunicação

Um algoritmo (engine) é um programa de linha de comando, que apenas comunica por texto. Esse texto tem de seguir a linguagem de comunicação aqui definida. Caso programe em C++, não necessita de implementar esta linguagem de comunicação, pode partir do código já feito (ver secção seguinte), e apenas tem de re-definir uma super-classe.

A própria linguagem, no caso de não ser dado um comando válido, envia uma descrição dos comandos, como se pode ver na imagem seguinte.



Descrição detalhada dos comandos:

- RUN - o algoritmo deverá começar a correr neste momento, com os parâmetros de entrada e instância que estiver carregada. Deverá ser criado uma thread à parte para correr o algoritmo, de modo a que este consiga em simultâneo ler comandos, nomeadamente o comando seguinte (STOP)
- STOP - paragem forçada do algoritmo. Ao receber este comando o algoritmo deve retornar o mais breve possível, com a melhor solução caso já tenha encontrado alguma, caso contrário sai sem solução. Como se poderá ver na proposta de código em C++, bastará incluir na condição de paragem dos ciclos principais o método Stop() para que esta funcionalidade seja implementada.
- RESET - limpar todos os parâmetros (ficam com valor de omissão), e descarregar a instância que eventualmente tenha sido carregada.
- SET INSTANCE [instance data] - carregar um ficheiro (instância), directamente pela entrada de dados (acaba com um caracter final de ficheiro)
- SET FILE [file name] - carregar um ficheiro (instância), através do seu nome (com a directoria completa).
- SET SOLUTION [solution] - alterar a solução corrente para a fornecida
- SET [k] [value] - alterar o valor do parâmetro [k] para o valor [value]
- SET SEED [value] - alterar o valor da semente aleatória para o valor [value]
- SET LIMIT STATES [states] - alterar o limite do número de estados permitidos para [states]
- SET LIMIT TIME [miliseconds] - alterar o limite de tempo, em milissegundos para [miliseconds]

- SET LIMIT LEVEL [maximal level] - alterar o limite do número máximo de níveis para [maximal level]
- SET INFORMATION [value] - alterar o nível de informação para [value]. O algoritmo é que define quantos níveis de informação tem. Quanto maior, mais detalhe o algoritmo reportará, sobre o seu comportamento (ideal para saber o porquê de um resultado incorrecto).
- GET INSTANCE - devolver a instância para a consola
- GET FILE [file name] - gravar a instância para o ficheiro [file name]
- GET SOLUTION - devolver a solução actual
- GET STATES - retornar o número de estados analisado
- GET TIME - retornar o tempo utilizado em milissegundos
- GET LEVEL - retornar o nível máximo atingido
- GET SETTINGS - retornar o número de parâmetros que o algoritmo tem
- GET [k] - retornar o valor actual do parâmetro [k]
- GET NAME [k] - retornar o nome do parâmetro [k]
- GET DESCRIPTION [k] - retornar uma descrição do parâmetro [k]
- GET MINIMAL [k] - retornar o valor mínimo possível para o parâmetro [k]
- MAXIMAL [k] - retornar o valor máximo possível para o parâmetro [k]
- GET INDICATORS - retornar o número de indicadores disponível
- GET INDICATOR [k] - retornar o valor do indicador [k]
- GET INDICATOR NAME [k] - retornar o nome do indicador [k]
- GET INDICATOR DESCRIPTION [k] - retornar uma descrição do indicador [k]
- GET ENGINE NAME - retornar o nome do algoritmo (engine)
- GET ENGINE VERSION - retornar a versão do algoritmo
- GET ENGINE DATE - retornar a data de compilação do executável
- GET ENGINE AUTHOR - retornar o nome do autor (ou autores)
- GET ENGINE CONTACT - retornar um ou mais contactos com o autor
- GET ENGINE DESCRIPTION - retornar uma descrição curta do algoritmo
- GET LIMIT STATES - retornar o número de estados permitidos
- GET LIMIT TIME - retornar o tempo em milissegundos permitido
- GET LIMIT LEVEL - retornar o nível máximo permitido
- GET INFORMATION - retornar o nível de informação actual
- INFORMATION [value] - retornar uma descrição sobre o nível de informação [value]. O primeiro nível de informação não deverá mostrar nada.

6.1.3 Implementação C++

A instalação do EngineTester cria uma directoria "Source Code\NewEngine" com o código necessário para começar a implementar o algoritmo sem ter de se preocupar com a linguagem de comunicação.

A directoria "NewEngine" tem os seguintes ficheiros que podem ser consultados, mas não devem ser alterados:

- TEngine.cpp e TEngine.h que correspondem respectivamente à implementação e declaração da super-classe TEngine. Todos os algoritmos devem ser uma subclasse desta classe, que contém todos os métodos que os algoritmos podem e devem redefinir.
- TRand.cpp e TRand.h com a implementação e declaração de uma classe auxiliar geradora de números aleatórios
- TVector.h com a implementação e declaração de uma classe auxiliar de vectores dinâmicos

São também criados os ficheiros que vão ser alterados para conter a implementação do novo algoritmo:

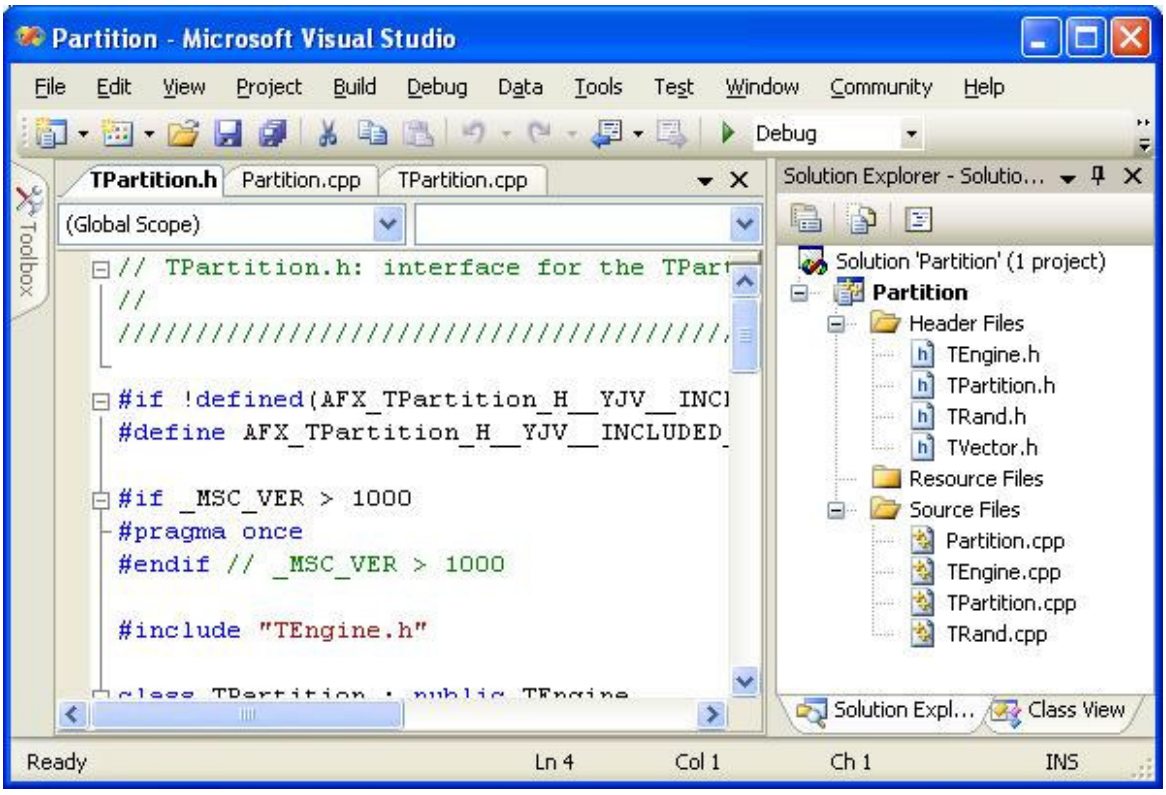
- NewEngine.cpp é o ficheiro tem a função main, e o código da linguagem de comunicação. Aqui há que

modificar apenas o nome da subclasse com o novo algoritmo

- TNewEngine.cpp TNewEngine.h com a implementação e declaração do novo algoritmo numa subclasse de TEngine

Para iniciar um novo algoritmo tem que alterar o nome "NewEngine" pelo nome do seu algoritmo, tanto no nome dos ficheiros como também nas referências internas dentro dos ficheiros "NewEngine.cpp", "TNewEngine.cpp" e "TNewEngine.h". De seguida basta implementar o algoritmo editando apenas os novos ficheiros "TNewEngine.cpp" e "TNewEngine.h".

Para compilar este código o compilador tem de ser posto em modo de multi-thread. O presente código foi testado no Visual Studio 6.0, e de omissão não está neste modo. Se tem dúvidas que o seu compilador tenha um modo multi-thread, compile os ficheiros antes de os alterar, para ver se consegue. Caso não consiga compilar, o problema será na chamada a "_beginthread" e "_endthread". Notar que uma thread é diferente de um processo. Esta implementação necessita de uma thread para a leitura de dados, e cria outra thread para o algoritmo correr. No caso do Visual Studio 2005, não é preciso fazer nada. Na figura está o resultado após construir um projecto "Win32 Console Application", adicionar os ficheiros e substituir o "NewEngine" por "Partition". Está pronto a correr.



Pode fazer uma versão que não suporte multi-thread substituindo o _beginthread pela chamada à própria função e comentando o _endthread, mas tal pode ter efeitos indesejáveis. O EngineTester assume que a comunicação entre a aplicação EngineTester e todos os algoritmos (engines) abertos, deve estar sempre activa, e não haver momentos em que a comunicação é ignorada porque está um algoritmo a correr.

Após a compilação correcta tem um conjunto de tarefas a fazer nos ficheiros TNewEngine.cpp e TNewEngine.h:

- identificar o algoritmo alterando os nomes no método Engine
- criar a estrutura de dados para guardar uma instância em TNewEngine

- criar a estrutura de dados para guardar uma solução
- verificar que construtor e destrutor estão de acordo com a estrutura criada
- implementar Load/Save da instância
- implementar LoadSolution/SaveSolution
- implementar em Reset: identificação de todos os parâmetros (ver código comentado); identificação de todos os indicadores (ver código comentado); adicionar modos de informação, adicionando valores para info_description (ver declaração de TEngine).
- implementar em Run o algoritmo, e adicionar parâmetros se necessário (nesta fase já se pode utilizar o EngineTester para testes). Nos ciclos mais importantes, incluir no critério de paragem a chamada ao método Stop() (de TEngine), para que seja possível parar o algoritmo no caso do utilizador ter mandado parar, ou o tempo estar esgotado (ou qualquer outro critério de paragem). Há que actualizar também o número de estados e nível mais alto (value_states, value_level de TEngine). Por exemplo, o loop base de um algoritmo pode ser: while(!Stop()) { value_states++; /*processar*/ }
- implementar indicadores (método Indicator)
- optimizações, testes de fiabilidade

É tudo. O algoritmo está pronto a ser testado no EngineTester.

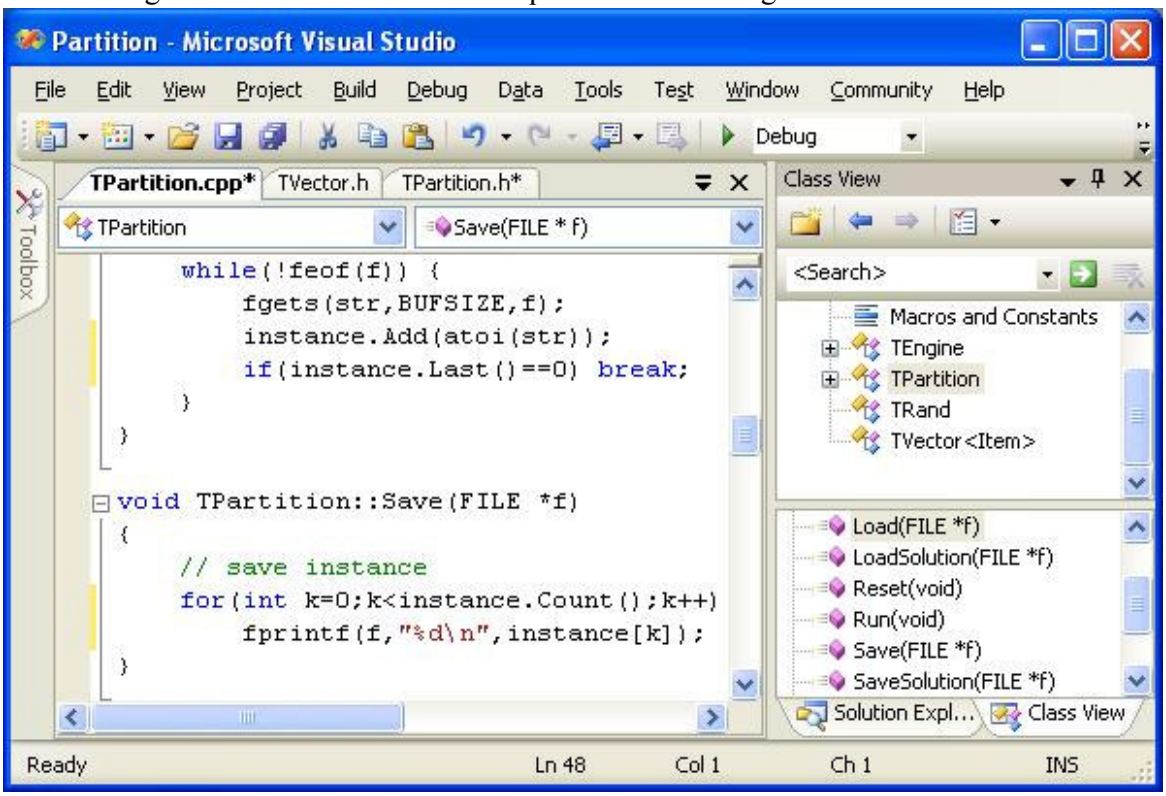
6.1.4 Exemplo (partition)

A partir dos ficheiros da directoria "NewEngine", criar um algoritmo para o problema da partição (partition). Este problema consiste em tendo um conjunto de números inteiros, saber se se podem dividir em dois grupos cuja soma seja exactamente igual. O algoritmo a implementar é determinístico e exaustivo, percorre todas as partições possíveis (algumas indirectamente) e retorna assim que encontrar uma viável.

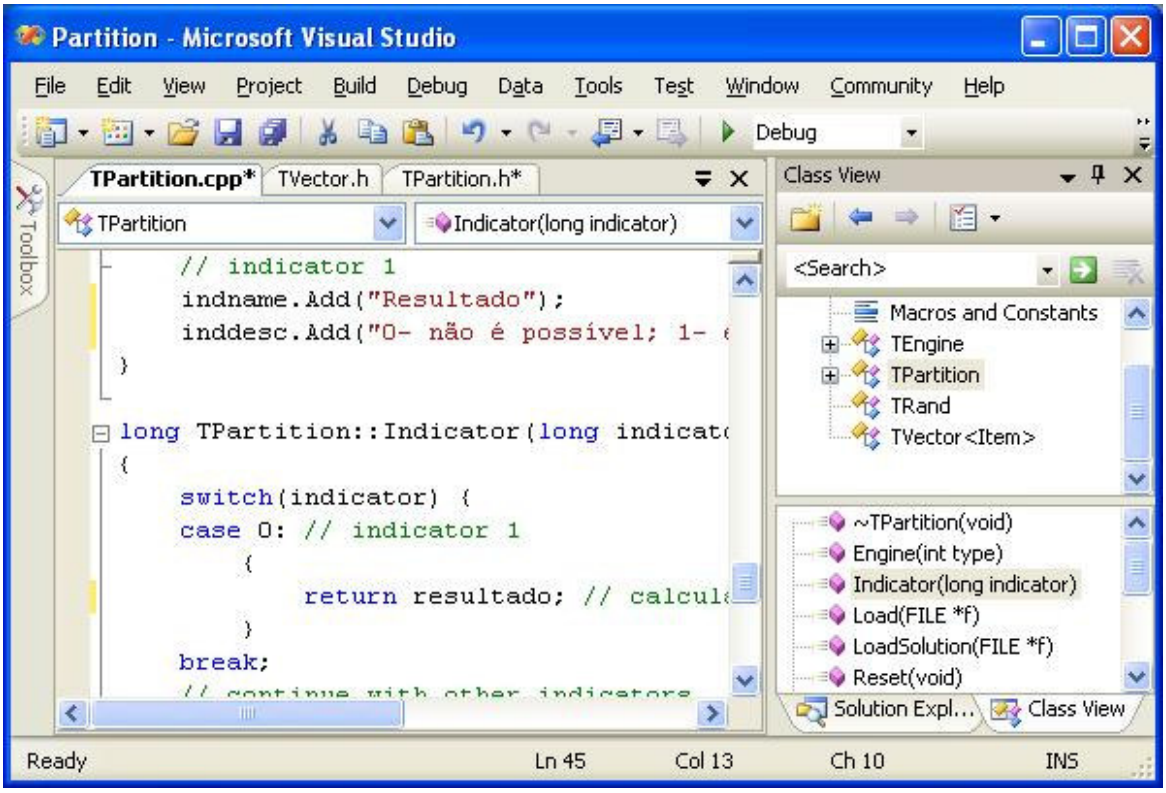
Fazer os seguintes passos:

- criar um projecto do tipo "Console Aplication" utilizando o ambiente de desenvolvimento desejado, e selecionar "empty project" (Visual Studio 2005: projecto de C/C++; template "Win32 Console Application"; nas application settings escolher "console application" como tipo e "empty project" como opções adicionais).
- copiar todos os ficheiros da directoria de "NewEngine" para a nova directoria e alterar o nomes de "NewEngine" para "Partition"
- incluir os ficheiros no projecto e editar Partition.cpp/TPartition.cpp/TPartition.h e fazer substituição automática de "NewEngine" por "Partition"
- alterar configuração do projecto (no Visual Studio 6.0: Project|Settings|C/C++|Code Generation|Multithreaded (ou Debug Multithreaded)). No Visual Studio 2005 não é necessário fazer nada.
- actualizar o método Engine de TPartition e mandar compilar. Se houver problemas, ler a secção anterior, caso contrário correr e escrever na consola "get engine name", ver o resultado e fechar a consola.
- o próximo passo é criar a estrutura de dados para guardar a instância. Como é um vector de números inteiros, e já existe um objecto auxiliar para vectores, TVector, criar uma variável deste tipo, com elementos inteiros (int), em TPartition: TVector<int> instance;
- a solução poderia ser outro vector de inteiros (ou um vector binário), para guardar os números de uma das divisões. Como o problema é apenas de saber se há ou não uma solução, e não de encontrar uma solução caso exista, não vamos guardar a solução.
- o construtor e destrutor estão ok, não precisão de nada porque foi criado apenas um objecto que se destroi ele próprio automaticamente, não deixando lixo
- implementar o método Load/Save: no método Reset adicionar a linha instance.Count(0); ||| aproveitar

o código feito no método Load e na linha para processar a string inserir o comando instance.Add(atoi(str)); (neste caso deverá existir um número em cada linha) e o comando if(instance.Last()==0) break; (para acabar de ler a instância assim que venha um zero) ||| no método save colocar um ciclo a percorrer todos os elementos, em vez da chamada ao fprintf actual: for(int k=0;k<instance.Count();k++) fprintf(f,"%d\n",instance[k]); ||| compilar e correr e escrever: "set instance" e de seguida digitar números (um por linha) e acabar com um 0 (zero), escrevendo o comando "get instance". O engine deverá escrever a instância que acabou de carregar.



- deixar a implementação LoadSolution/SaveSolution por fazer
- no método Reset, não colocar nenhum parâmetro, e colocar apenas um indicador (alterar código: em indname "Resultado"; em inddesc "0 - não é possível; 1 - é possível.")
- criar uma variável inteira resultado em TPartition: int resultado; no método Reset inicializar a zero: resultado=0;
- no método Indicator, no caso do "indicator 1" em vez do return 0; colocar return resultado;

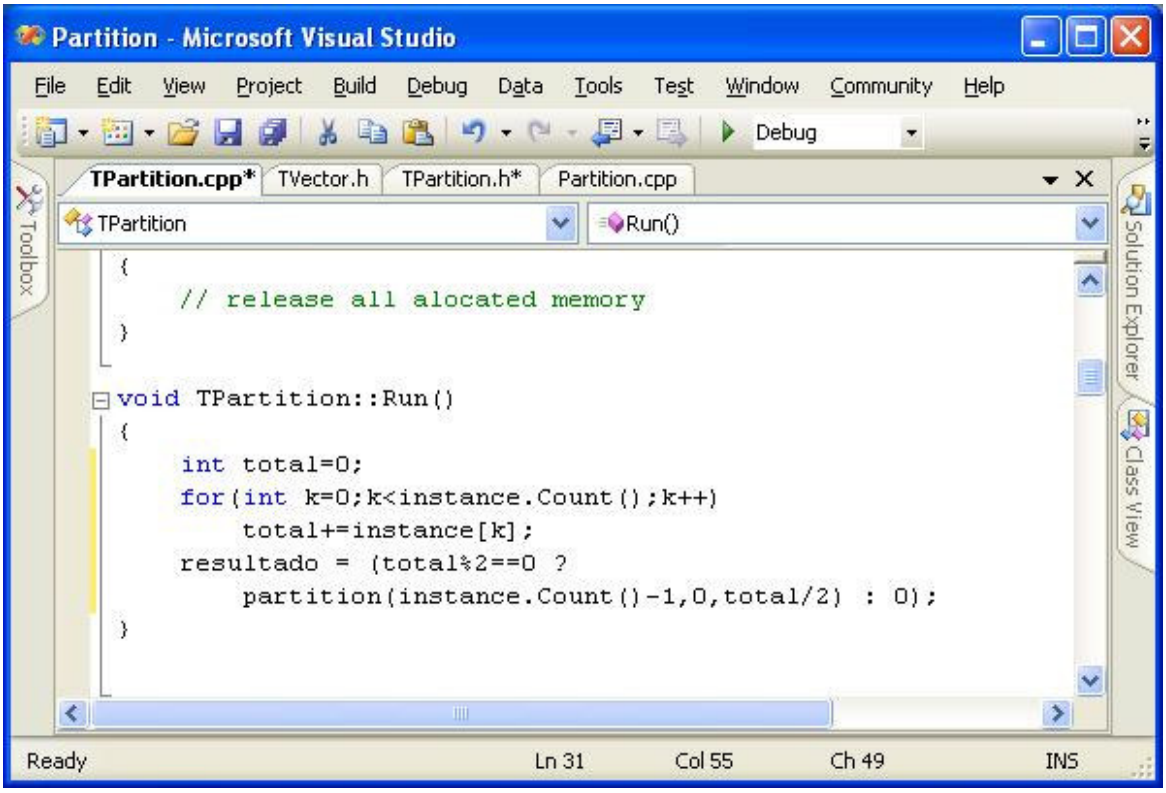


Falta agora implementar o algoritmo. Primeiro há que salientar alguns pontos que nos permitem aliviar algum tempo de processamento:

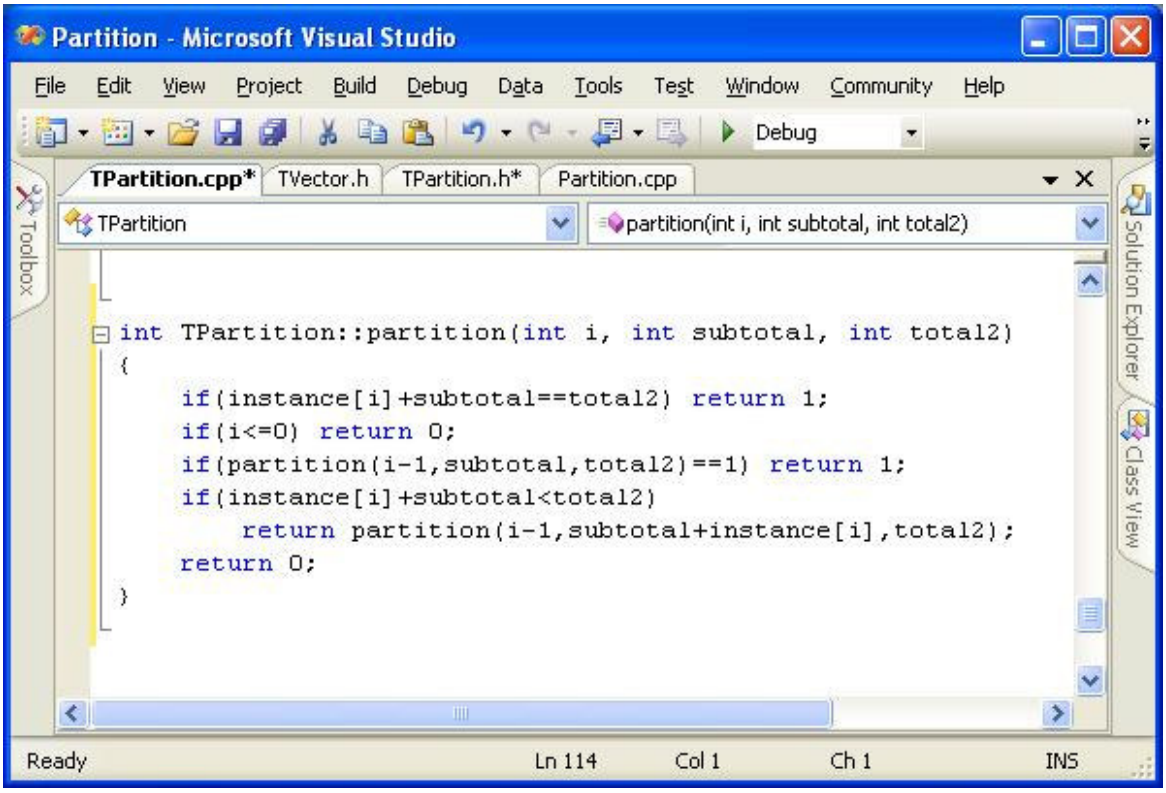
- o algoritmo apenas deve tentar encontrar partições caso a soma total dos números seja par. Caso a soma seja impar é impossível obter uma partição uma vez que os números são todos inteiros, e nessa situação o engine pode retornar logo impossível
- processar os elementos 1 a 1, e ter sempre o valor da soma total a dividir por dois (para não considerar partições com valores superiores a esse valor)
- ter o valor do somatório dos elementos na partição a considerar
- considerar em cada passo um elemento i: se somando o valor de i com os elementos já na partição for igual ao total/2, retornar sucesso; caso seja superior, tentar fazer partições apartir de i+1 sem entrar com i; caso seja inferior, se não houver mais elementos (i é o último), retornar insucesso, caso contrário tentar apartir de i+1 entrando em linha de conta com i, e caso não dê tentar sem entrar com i.

Se for suficiente a ideia, ou se conseguir implementar um algoritmo seu, força. Caso contrário pode fazer os seguintes passos, ao estilo de laboratório:

- criar método partition com os argumentos o valor i, subtotal e total2, retornando inteiro
- no método Run, calcular a soma total
- caso seja divisivel por 2, chamar partition, caso contrário 0, e guardar o resultado na variável resultado



- no método partition, verificar se subtotal com i fica igual a total2, retornar sucesso nesse caso.
- verificar se i==0, nesse caso não há hipótese, retornar insucesso
- verificar se sem este elemento existe uma partição, e retornar sucesso se existir
- verificar se o valor não é ultrapassado se se contar com este elemento, e retornar a chamada à partição
- na última linha inserir return 0; porque nesse caso não é possível



Está feito, agora há que fazer um pequeno teste antes de o testar no EngineTester. Compilar e correr, inserir uma instância divisivel e outra divisivel e mandá-lo correr.

Utilizar os seguintes comandos (no engine):

- set instance - e inserir a instância como anteriormente
- run - e esperar que apareça "Run end."
- get indicator 1 - e ver o resultado

Repetir os comandos escrevendo uma instância que tenha o resultado oposto.

Estes pequenos testes não garantem no entanto que o algoritmo esteja a funcionar correctamente, nem dão ideia da sua complexidade algoritmica. São necessários testes mais extensos, algo que o EngineTester facilita e sem este provavelmente o investigador já fatigado nesta altura, acabaria por dar o trabalho como concluido. Neste caso a complexidade algoritmica é possível de calcular, é de O(2^n) para o pior caso, mas mesmo neste problema não há ideia do tempo médio para um conjunto por exemplo de 1000 inteiros gerados aleatoriamente entre 1 e 1000.

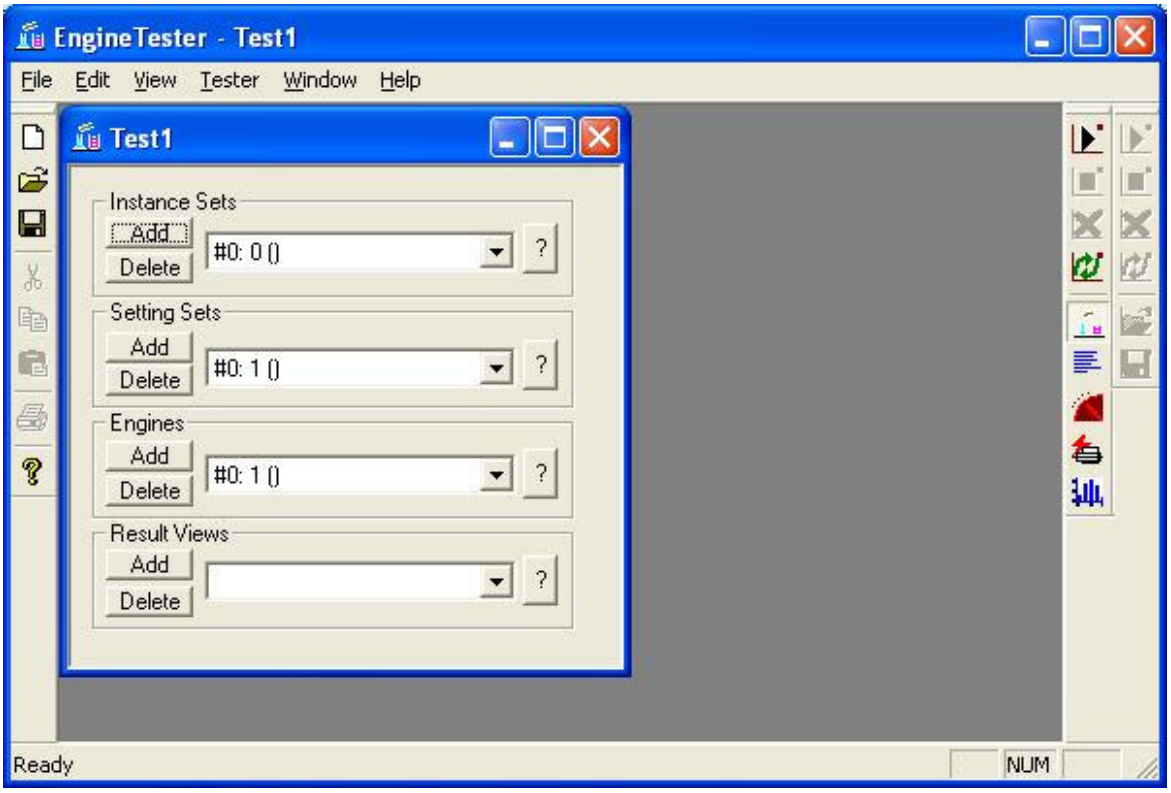
6.2 Utilização do Engine Tester

Um documento no EngineTester guarda informação necessária para executar um conjunto de testes, podendo envolver mais do que um algoritmo, mais do que um conjunto de parâmetros para cada algoritmo, e mais que um conjunto de instâncias, tendo também a informação sobre a construção de gráficos de resultados. Este texto assume que se leu a secção anterior, e se tem conhecimento do que é um engine (algoritmo).

6.2.1 Dialog Base (Main View)

Ao iniciar o EngineTester, um novo documento é criado, sendo aberta a dialog base deste. Esta dialog permite visualizar e editar os elementos de quatro conjuntos, os quais constituem o documento EngineTester. Esses conjuntos são: conjunto de instâncias (instance sets); conjunto de parâmetros (setting sets); conjunto de algoritmos (engines); conjuntos de gráficos (result views).

Cada conjunto tem uma "combobox", em que está seleccionado o elemento desse conjunto activo, e possui os botões Add/Delete para adicionar e remover um elemento a cada conjunto respectivamente. O nome do elemento e respectivas configurações, são editáveis em dialogs específicas.



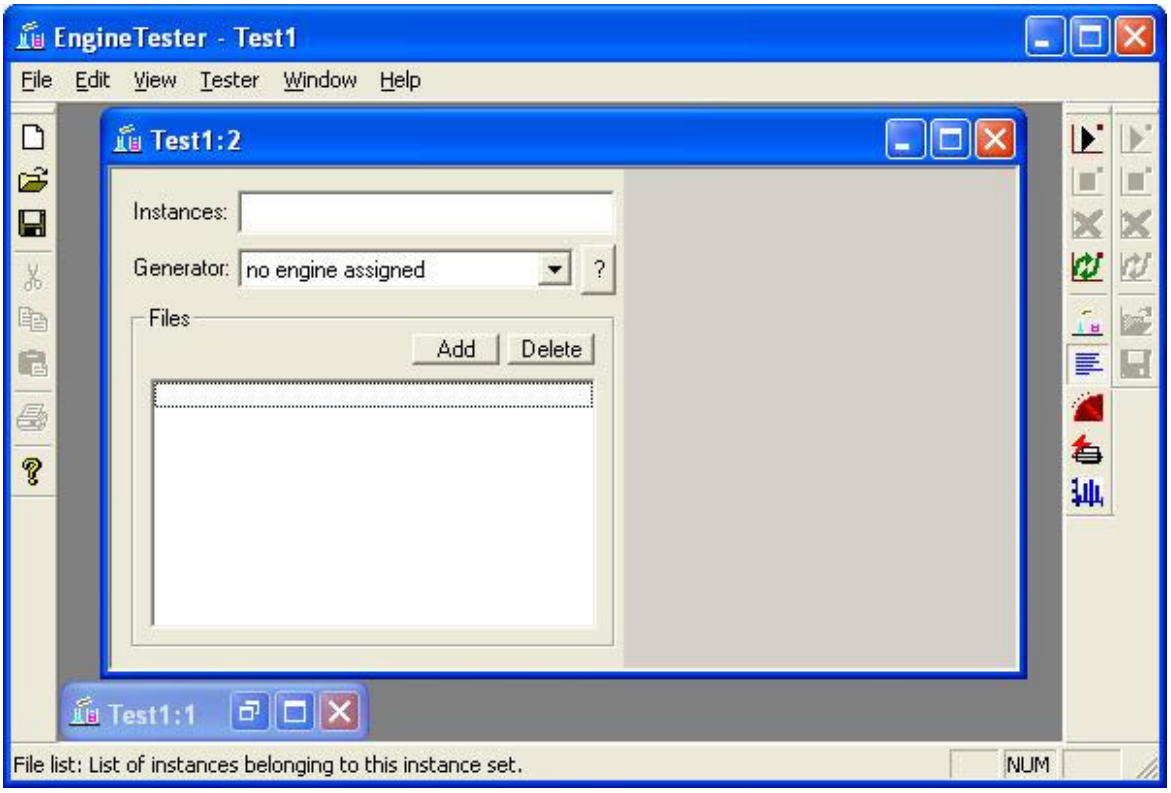
O botão "?" junto de cada conjunto, tal como outros botões com "?" espalhados um pouco por todo o lado, dá uma informação breve do que representa.

Para abrir uma dialog de edição de um dos elementos dos quatro conjuntos, basta que seleccione o elemento. Faça isso para as instâncias, e de seguida minimize a dialog base.

6.2.2 Conjunto de Instâncias (instance sets)

Um conjunto de instâncias é definido pelo seu nome, e um conjunto de ficheiros, em que cada ficheiro tem os dados de uma instância.

Alternativamente a se dar os ficheiros, pode-se fornecer um engine que gera as instâncias, sendo nesse caso passado o resultado desse engine para o engine que necessitar deste conjunto de instâncias.

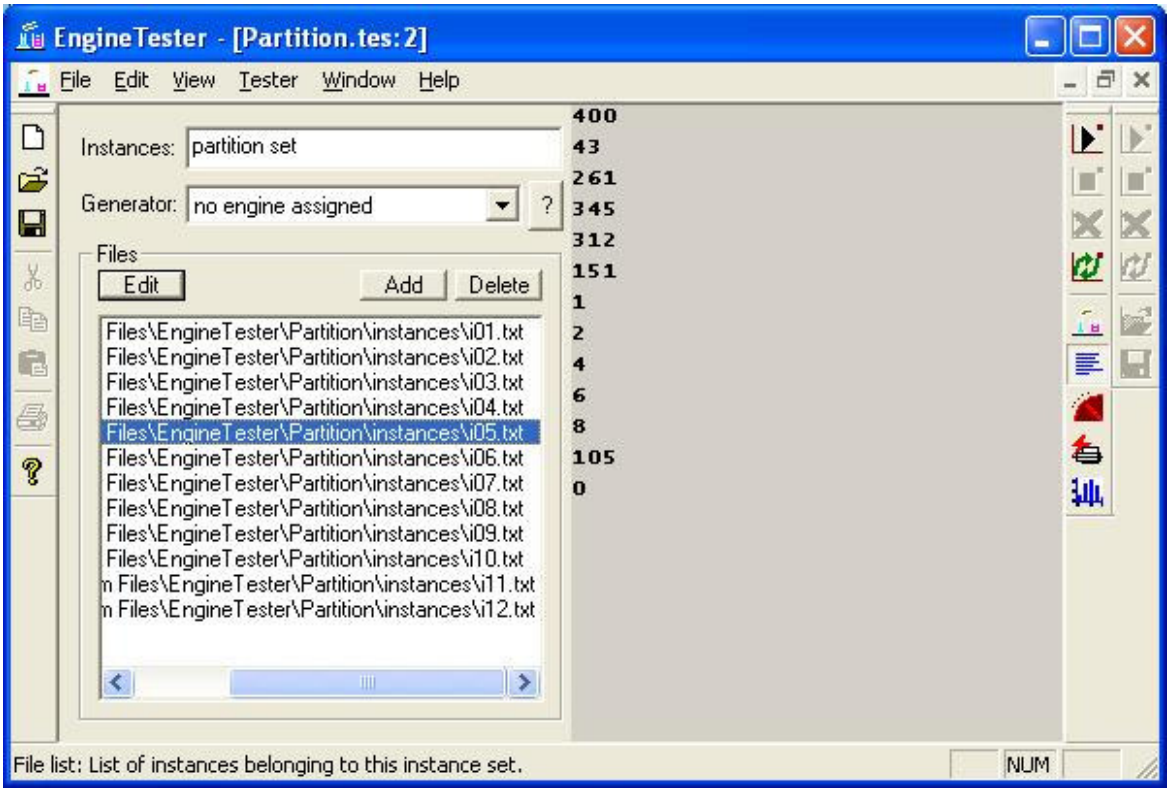


Vamos continuar com o exemplo dado, Partition. Há que construir um conjunto de instâncias de teste para o partition. O formato é simples, um conjunto de números acabando no 0, com os números postos um por linha:

- escreva o nome do conjunto de testes (por ex: "partition set")
- adicione um ficheiro, e crie uma directoria "instances" debaixo da directoria do Partition criado na secção anterior, e dê um nome ao ficheiro
- como o ficheiro não existe, este é criado. Para o editar no notepad, apartir do EngineTester, pode fazer um duplo click sobre a linha do ficheiro. Edite e inicialize com valores que queira.
- repita o processo até existirem uma dúzia de ficheiros de instâncias

É conveniente que todo o conjunto esteja na mesma directoria. No caso de se levar todo o teste para outro computador, incluindo as instâncias, as directorias podem ser diferentes, e nesse caso o EngineTester pergunta a nova directoria. Para as restantes instâncias no conjunto, assume que a directoria trocada se mantêm, mas se não achar o ficheiro pergunta por nova directoria. De forma a poupar trabalho de indicar novas directorias, aconselha-se a que um conjunto de instâncias esteja na mesma directoria.

Ao seleccionar um ficheiro na lista, na zona direita o ficheiro é mostrado. Se não cober na zona completamente, pode navegar no ficheiro através de arrastar o rato, e através de um duplo-click acabando em arrastar o rato.



Na imagem anterior pode-se ver o ficheiro i05.txt seleccionado, sendo o seu conteudo mostrado na zona da direita.

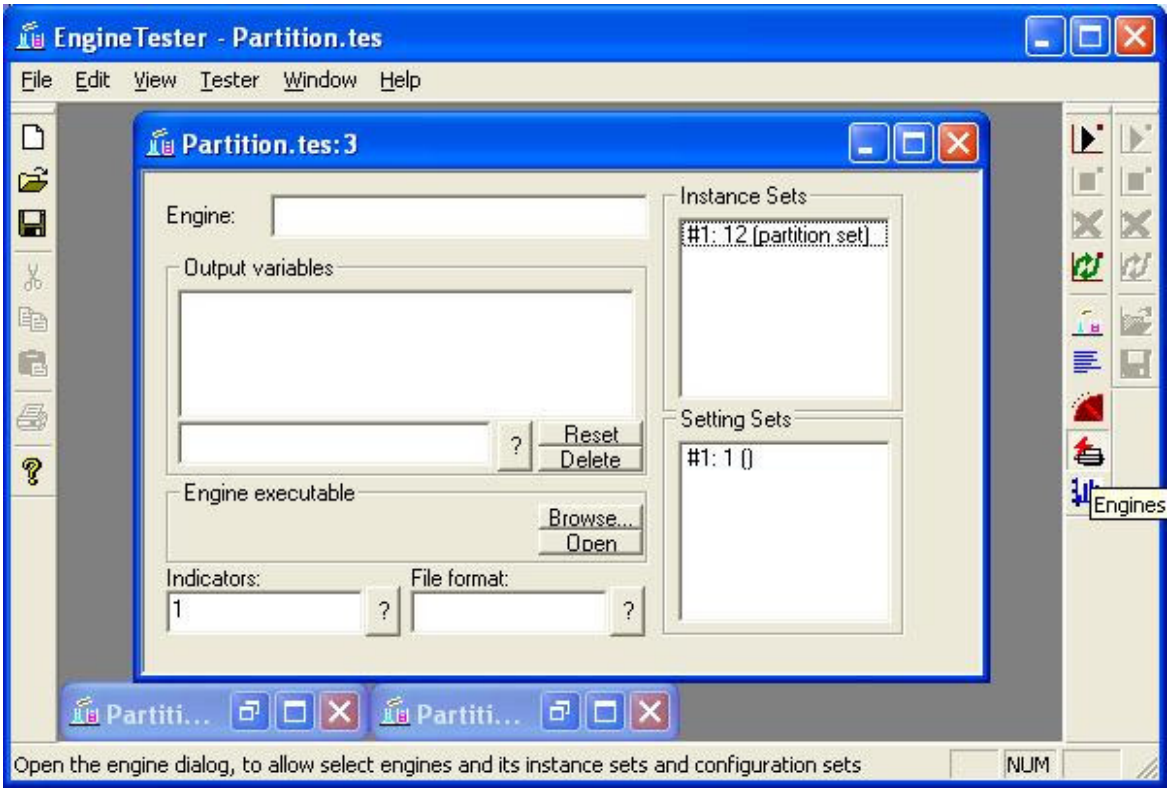
Convem que as instâncias de teste sejam diferentes, de forma a poder ver como o algoritmo se comporta nas diferentes situações. No Partition, pode-se suspeitar ser relevante ao desempenho do algoritmo o número de elementos, e o número de dígitos médio por elemento. Naturalmente que as instâncias cuja soma dos elementos não seja par, é de pouco interesse, já que o algoritmo detecta que é impossível e rejeita de imediacto.

O leitor facilmente concordará que um conjunto construido manualmente dificilmente pode ter instâncias de acordo com o ideal especificado no parágrafo anterior. Primeiro devido a ser trabalhoso construir instâncias grandes, e segundo porque para tirar conclusões são necessárias muitas instâncias, e não apenas uma dúzia.

Apesar destas questões, vamos para já manter o exemplo assim, de forma a este ficar simples, e passar à configuração do nosso engine.

6.2.3 Conjunto de algoritmos (Engines)

Seleccione na toolbar à direita, o icon que se parece com um motor. Esta é um forma rápida de trocar de dialog, passando a editar o engine de seleccionado na dialog base.



Para configurar um engine, há que seleccionar o executavel de linha de comando que implementa o algoritmo. Carregue no botão "Browse" e seleccione o executável Partition.exe construido na secção anterior. Na zona "Engine executable" deverá aparecer o executável. Se necessário, pode criar mais que um engine com o mesmo executável, desde que dê nomes diferentes a cada engine no documento EngineTester.

Para além do nome, um engine no EngineTester tem:

- um subconjunto dos conjuntos de instâncias definidos
- um subconjunto dos conjuntos de parâmetros definidos
- um conjunto de variáveis de saida
- um subconjunto dos indicadores do motor (que se podem utilizar nas variáveis de saída)
- um formato de construção do nome dos ficheiros processados, no caso de se querer gravar após transformar uma instância

A lista do canto superior-direito, tem as instâncias que possui juntamente com o número de ficheiros. A lista permite multipla-selecção, de forma a que se seleccione os conjuntos de instâncias a enviar para este engine. A lista de conjuntos de parâmetros, no canto inferior-direito, segue a mesma filosofia.

O campo dos indicadores permite seleccionar um subconjunto dos indicadores possíveis. Apenas os seleccionados aqui serão pedidos ao engine, após o fim do processamento, e estarão disponíveis para construção das variáveis de saída. Desta forma, caso o engine tenha por exemplo 20 indicadores, e apenas sejam precisos 3, evita-se ter de calcular 20 indicadores para apenas utilizar 3.

O formato do campo dos indicadores é o seguinte:

- [ind]:=[ind1]|[ind1] [ind]
- [ind1]:==[número]|[número]-[número]|[número]-[número]:[número]

Um ou mais [ind1] podem ser inseridos, desde que separados por um espaço. Um [ind1] ou é um

número, ou é do tipo [a]-[b] e nesse caso são todos os números entre [a] e [b] inclusivé, ou então é do tipo [a]-[b]:[c], e nesse caso são todos os números de [a] a [b] de [c] em [c]. Por exemplo: "1 6-9 20-80:15" corresponde aos números 1 6 7 8 9 20 35 50 65 80. Como apenas temos um só indicador, implementado no Partition, deixa-se este campo com o valor de omissão "1".

As variáveis de saída podem ser construídas utilizando as funções matemáticas comuns (sin, cos, sqrt, pow,...) e com base em diversas variáveis:

- I[i] - valor do indicador [i] seleccionado (1 é o primeiro)
- VS, VT, VL - valor utilizado de número de estados, tempo e nível máximo
- PS, PT, PL - valor dos parâmetros de limites do número de estados, tempo e nível máximo
- P[i] - valor do parâmetro na ordem [i] do conjunto de parâmetros
- NE, NI, NS - número do engine, do conjunto de instâncias e do conjunto dos parâmetros

Como temos apenas um indicador, para já podemos colocar as seguintes variáveis: VT; I1; NI. Atenção que para adicionar uma nova variável, a lista de variáveis não pode estar seleccionada, caso contrário é editada a variável seleccionada e não criada nova variável.

No campo de formatação de ficheiros (File format), coloca-se o formato do nome dos ficheiros para serem gravados com novo nome, após serem processados. Caso o campo esteja em branco, os ficheiros não são gravados:

- prefixo*prefixo_apagado*sufixo_apagado*sufixo
- utilizar #i, #s, #e, para substituir pelo número da instância, número do parâmetro, e número do engine respectivamente

Os ficheiros criados para teste ficaram com os nomes: "i01.txt"; ...; "i12.txt". Para os modificar de forma a ficarem "partition01.txt"; ... ;"partition12.txt", bastaria inserir a formatação: "partition*i**", ou seja, apagar o i e adicionar o prefixo "partition".

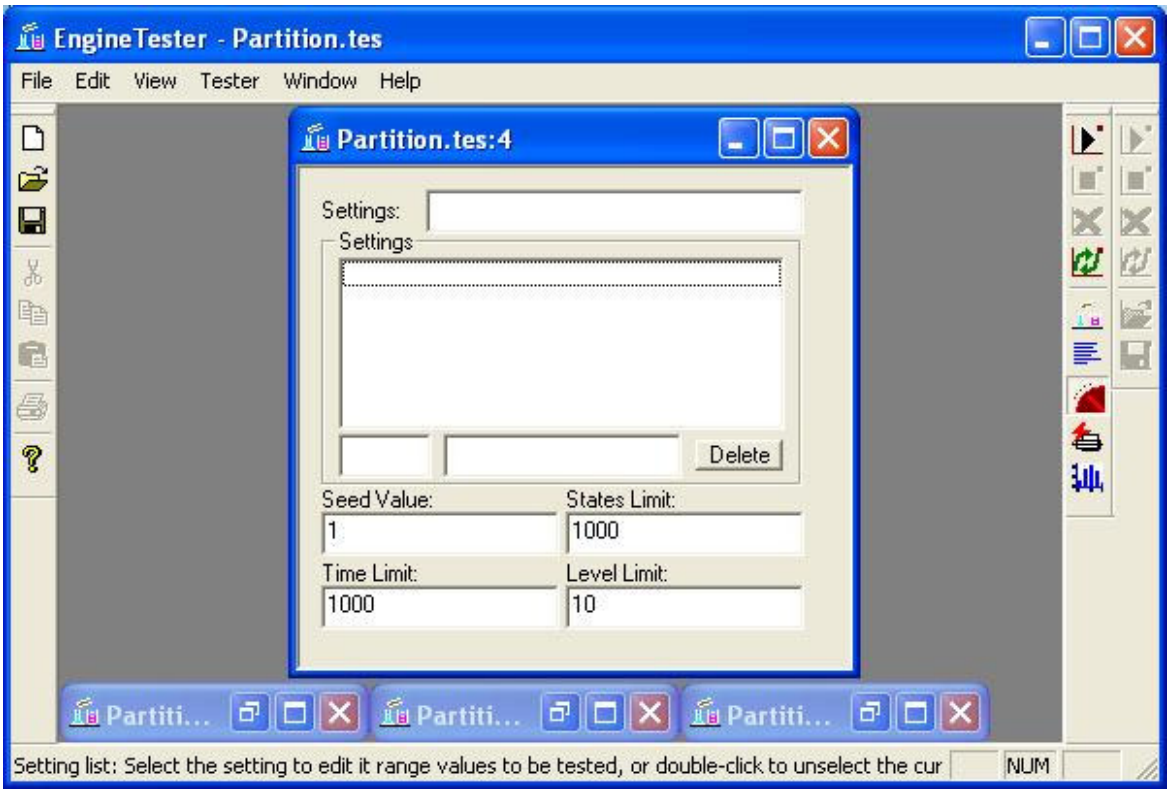
Se um ou mais ficheiros tiverem sido apagados, o número do ficheiro deixa de ter significado. Nesse caso pode-se ignorar o número: "partition#i.txt". Esta formatação ignora o nome original da instância, e distingue as instâncias apenas no número de instância.

6.2.4 Conjunto de parâmetros (Setting Sets)

O conjunto de parâmetros tem de ser identificado por um nome, e os valores dos parâmetros colocam-se da mesma forma que os indicadores na dialog de Engine.

Cada engine pode ter um conjunto de parâmetros, para adicionar novo parâmetro basta escrever o número do parâmetro e os valores, e nova linha é adicionada. Os parâmetros do algoritmo que não ficarem especificados, ficam com o valor de omissão. No nosso exemplo do Partition, não temos de momento nenhum parâmetro.

Quatro parâmetros são fixos: semente do gerador aleatório (Seed Value), de forma a que em algoritmos não determinísticos o resultado possa ser repetido; número máximo de estados analisados (States Limit); tempo máximo em milisegundos (Time Limit), mas apenas nesse instante será enviada ordem para saída, e no caso do algoritmo não chamar com frequência o método Stop (ver secção anterior), este pode ultrapassar este valor; nível máximo a explorar (Level Limit), sendo o nível um conceito mais aplicável a algoritmos de procura em árvore, mas pode ser livremente utilizado. Voltaremos com um exemplo a esta dialog, quando se adicionar parâmetros ao Partition.



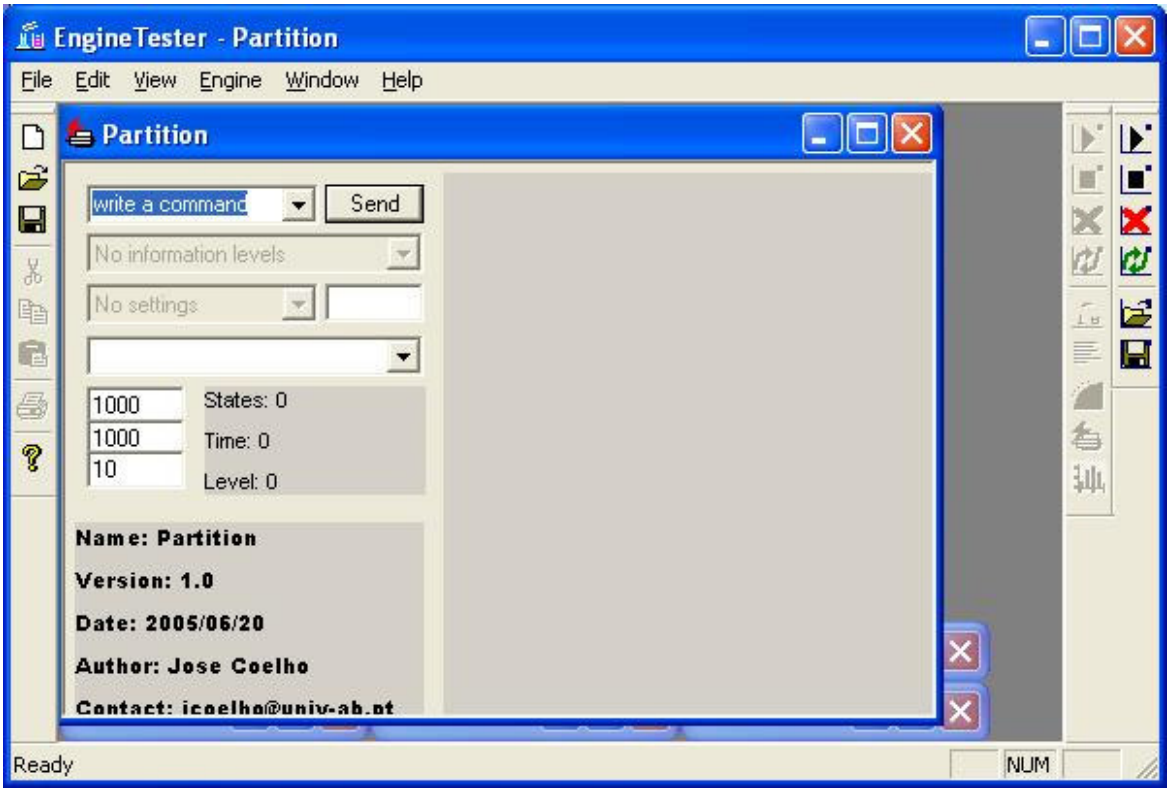
O número de elementos deste conjunto pode crescer rapidamente. Suponhamos que temos 4 parâmetros, e que em cada um queremos testar 10 valores diferentes. Está-se nesta situação a definir 10^4 hipóteses, que são 10.000 corridas. Este valor conjugado com o número de instâncias, por exemplo 100, daria um total de 1.000.000 de corridas.

Aconselha-se a fazer testes pequenos primeiro, antes dos grandes testes. Nos testes pequenos detecta-se rapidamente pequenos problemas com o algoritmo, ou com os resultados, que podem eventualmente levar à escolha de outros parâmetros para o grande teste.

6.2.5 Gráficos de Resultados (Result Views)

Os gráficos de resultados inicialmente não têm elementos. Devem ser construídos no final, uma vez que são dependentes dos objectos inseridos nos restantes conjuntos.

Antes de criar um gráfico, mande o algoritmo correr, através do menu Tester|Start ou do botão "Start". Irá abrir uma outra janela que irá comunicar com o engine. Esta janela também abriria no caso de na dialog Engine carregar no botão "Open".

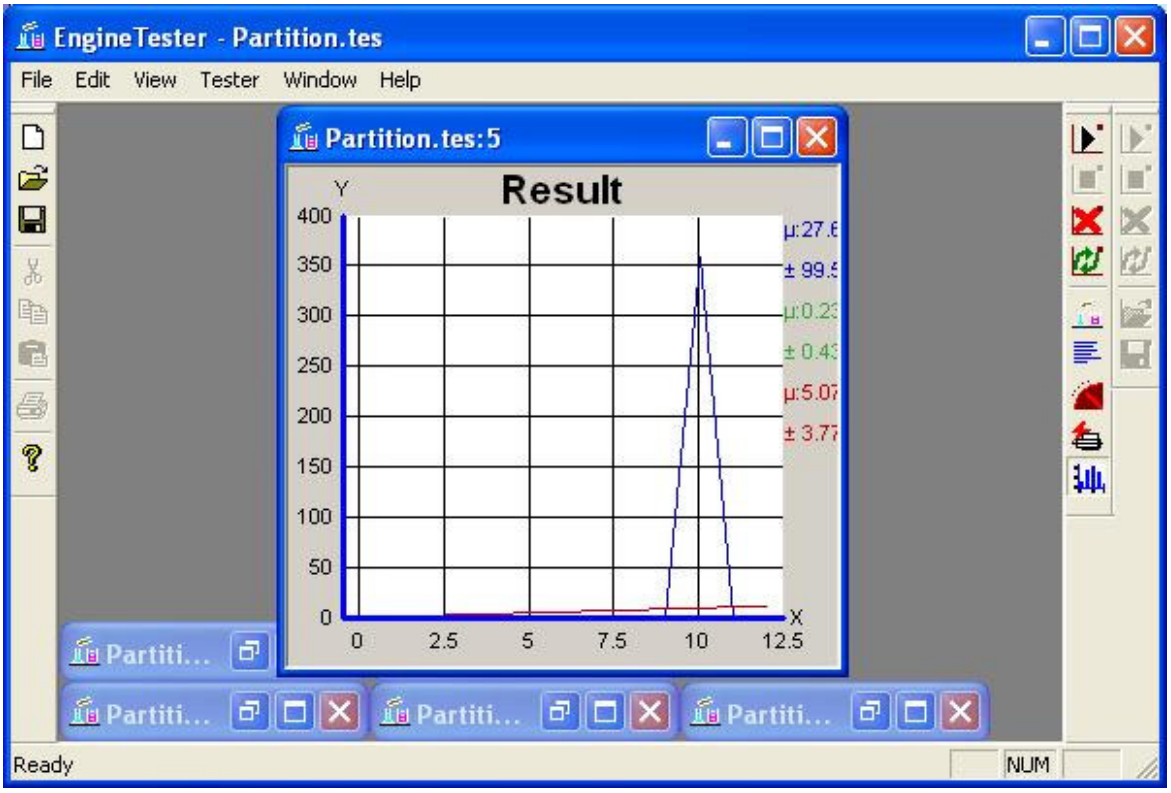


A janela apresenta informação do engine no canto inferior-esquerdo, permite enviar qualquer comando no canto superior-esquerdo. No resto da zona esquerda tem uma "combobox" com os níveis de informação possíveis, outra "combobox" com os parâmetros possíveis e respectivo valor, uma terceira "combobox" com os indicadores. Seleccione o valor "Resultado" nos indicadores, e verá informação descriptiva, que você próprio escreveu quando o construiu. De resto estão os limites do número de estados, tempo em milisegundos e nível máximo.

Caso não esteja um teste a correr, pode utilizar esta dialog para comunicar com um engine, para saber que parâmetros e indicadores tem implementados, ou para seleccionar uma determinada instância que dá um resultado incompreensível, seleccionando um nível de informação adequado para se compreender o porquê do resultado.

Já tendo corrido, faça Tester|Export|Results e grave num ficheiro. Veja o resultado. Deverá ter 12 linhas, caso não tenha abra a dialog Engine e seleccione o conjunto de instâncias. A informação aparece em forma tabelar, com o número de Engine, número de instância e número de parâmetro (neste caso não há parâmetros), seguidos das variáveis de saída (VT, I1, NI).

Crie agora uma vista gráfica através da dialog base, e responda "não" a todas as perguntas. Essas perguntas são relativas a cada elemento escolhido (do conjunto dos engines, das instâncias e dos parâmetros), de forma a fazer um gráfico entrando em linha de conta apenas com resultados desse elemento. Abra agora a vista gráfica criada.



O gráfico à partida mostra os valores seleccionados em série. Tendo 3 variáveis de saída, ficamos com 3 séries com tantos elementos quantas as corridas, neste caso 12. Mesmo neste gráfico podemos ver que a instância 10 houve um tempo muito elevado de 350 milisegundos.

Dado que a escala é igual para todas as séries, a segunda série com o resultado 0 ou 1 não se vê. Pode-se portanto dividir o tempo por 1000 para passar a segundos, e o número da instância por 10 (na dialog Engine). Antes de mandar correr novamente (pode-se manter a vista gráfica sempre aberta), carregue no botão "Clear results", caso contrário os resultados são adicionados aos anteriores.

Neste caso, abrindo a dialog do conjunto de instâncias, foi possível ver a instância que estava a consumir mais tempo, e esta tinha 43 elementos.

A vista gráfica, para além de séries, pode visualizar basicamente histogramas e scatterplots. Com o menu de contexto do DataView, é possível editar a vista gráfica, sendo essas modificações gravadas no documento EngineTester. Vamos mostrar algumas dessas funcionalidades com o evoluir da análise ao nosso algoritmo Partition.

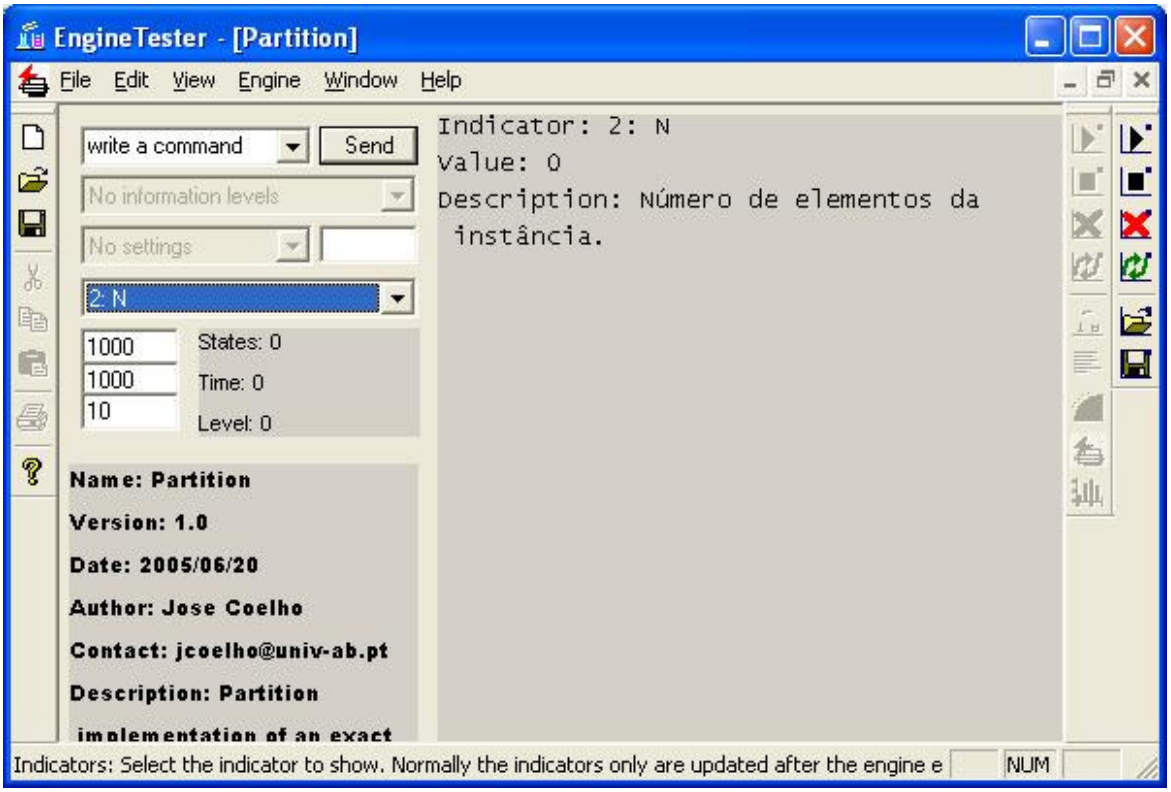
6.2.6 Análise ao Partition

Dos primeiros testes ficou a suspeita de que o tempo de computação depende do número de elementos. Como verificar isso? Primeiro há que construir um indicador no engine que retorne quantos elementos a instância tem. Desta forma podemos utilizar esta informação para cruzá-la com o tempo de computação.

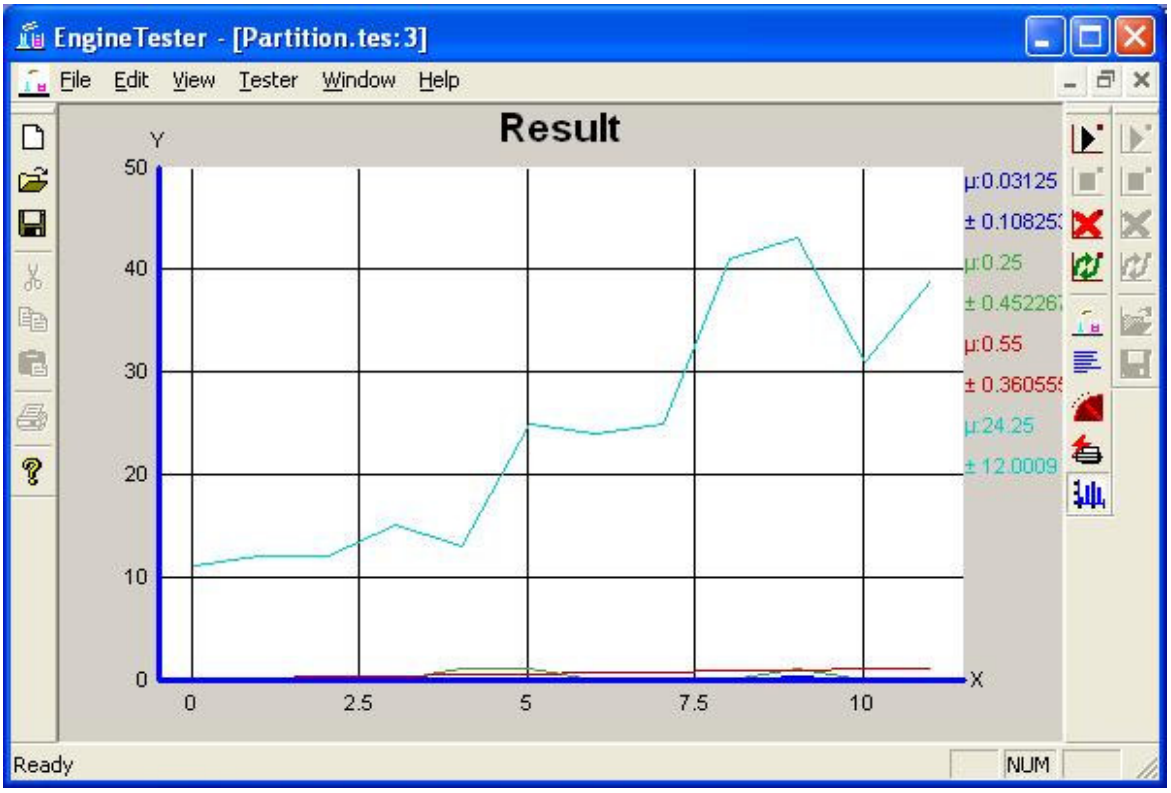
Abra o projecto do Partition e no método Reset adicione um novo indicador "N" com descrição "Número de elementos da instância.". No método Indicator, crie um novo "case" para o indicador e retorne o número de elementos de instance: case 1: return instance.Count(); break;

Compile, mas antes feche o EngineTester gravando o ficheiro, uma vez que pode ter o executável aberto, e volte a abrir o EngineTester novamente após ter o ficheiro compilado.

Para verificar que o novo indicador está funcional, abra a dialog de engine, e carregue no botão "Open". Irá abrir a dialog de comunicação com o engine e um processo onde estará o executável a correr. Se quiser pode abrir o Task Manager e ver o novo processo. Seleccione o valor "N" na "combobox" com os indicadores para ver a sua descrição.



Na dialog de Engine, coloque também o indicador 2, ficando portanto "1 2". Nas variáveis de saída coloque o indicador I2, de forma a ter-se acesso a este valor nas análises. Mande correr e reparará que não há nada de novo na vista gráfica. Abra a dialog base, e apague a vista gráfica refazendo-a novamente da mesma forma, já que adicionou uma variável de saída. Verá agora a nova variável.



Agora há que construir um gerador. Comece por criar um engine apartir de NewEngine, tal como foi feito para criar Partition, e dê o nome de GenPartition. Como a estrutura de dados será igual à do Partition, crie também a variável instance.

O gerador precisa de um gerador de números aleatório. É fornecido a classe TRand para gerar valores, não sendo necessário instanciar a classe já que os seus membros são estáticos. A semente é logo inicializada no TEngine, pelo que não há que preocupar com o assunto, apenas chamar o método Seed(); no início do método Run, e chamar o método TRand::rand() sempre que for preciso um novo valor aleatório.

O gerador necessita de parâmetros de entrada, como o número de elementos a gerar (N). Cada número pode ser gerado de acordo com uma distribuição uniforme, de 1 a M, mas obviamente há instâncias que podem ter os números distribuidos não uniformemente mas sim de forma logaritmica ou outra situação. Abstraindo desse facto, pode-se então criar um parâmetro (M) com o maior número possível.

Crie os dois parâmetros acima, com valores de omissão razoáveis (10 e 1000 respectivamente). No método Run faça a geração dos valores: Reset(); for(long k=0;k<value[0];k++) instance.Add(TRand::rand()%value[1]+1); instance.Add(0);

Compile e corra para efectuar o primeiro teste: no engine, faça "run" e "get instance" para ver a instância gerada. Altere o valor dos settings para verificar que os parâmetros influênciam a instância gerada. Se alterar o valor do Seed antes de mandar correr, verifica que a mesma instância é gerada.

Aparentemente tudo em condições para testar o desempenho do algoritmo Partition, mas na verdade não é possível verificar o resultado sucesso/insucesso, devido a não haver essa informação.

Em todos os problemas, em todas as dimensões, há normalmente instâncias simples. De igual forma, neste caso, muitas vezes as instâncias geradas impossíveis não somam um valor par e o Partition retornará logo, ou uma instância impossível pode ter um número muito mais elevado que os restantes

tornando a instância mais simples de resolver. As instâncias simples inviesam a análise de desempenho nas instâncias dificeis.

Para evitar gerar instâncias demasiado simples, resolveu-se gerar instâncias sempre possíveis, porque aparentemente não haverá muitas destas instâncias demasiado simples, a não ser que um só número seja igual à soma dos restantes. Por isso a solução deverá ter uma selecção de aproximadamente metade dos elementos de forma a que a complexidade seja máxima.

Resta saber como transformar o gerador puro anterior num gerador que garanta que existe uma solução? Normalmente bastará construir uma solução. Neste caso basta ter dois registos de somatórios parciais, a cada número que é gerado é somado num dos registos (o que tiver valor inferior), sendo o último número a diferença de valor entre os dois registos. Assim garante-se que há uma divisão, mas para que o algoritmo que resolver não possa fazer uso do algoritmo de geração para resolver o problema, há que baralhar (ou ordenar) a ordem dos números gerados antes de devolver a instância (instance.RandomOrder();).

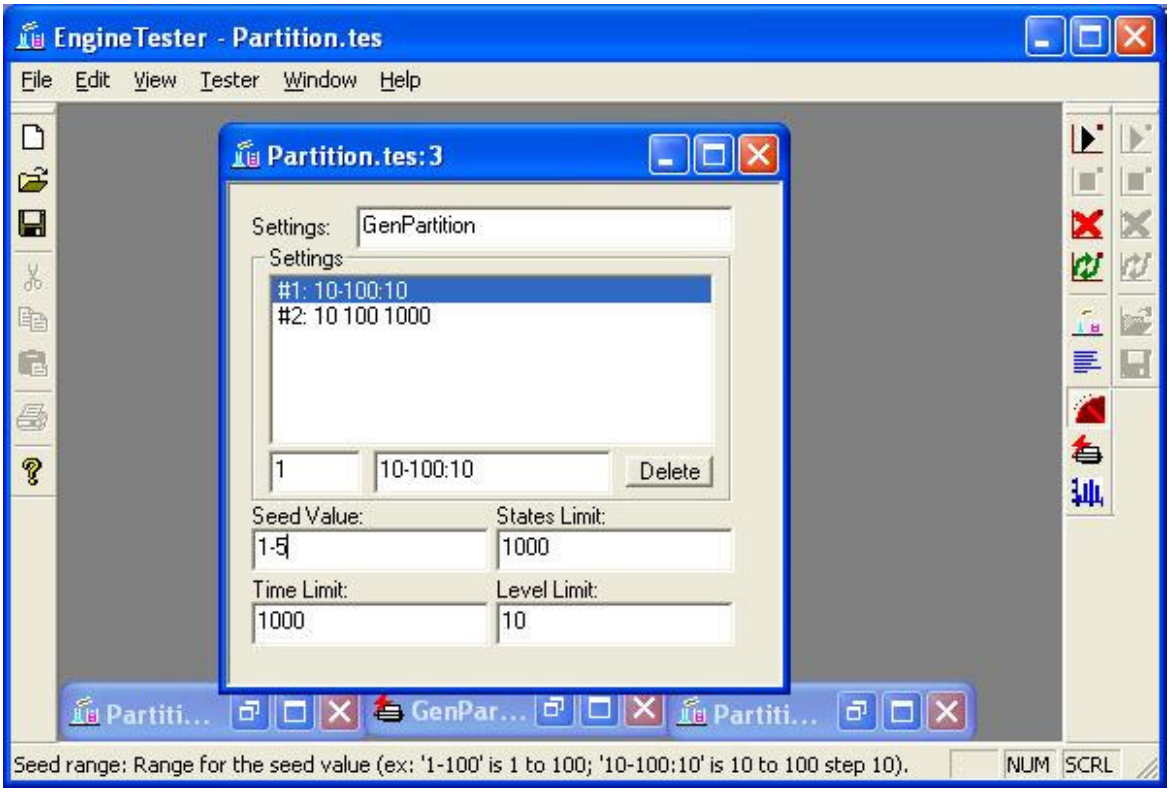
Caso não consiga implementar, siga os seguintes passos, no método Run:

- linha inicial: long soma1=0,soma2=0;
- substituir ciclo: for(long k=0;k<value[0]-1;k++) { instance.Add(TRand::rand()%value[1]+1); if(soma1<soma2) soma1+=instance.Last(); else soma2+=instance.Last(); }
- depois do ciclo: if(soma1<soma2) instance.Add(soma2-soma1); if(soma2<soma1) instance.Add(soma1-soma2);
- baralhar: instance.RandomOrder();
- não esquecer de adicionar 0 no fim: instance.Add(0);

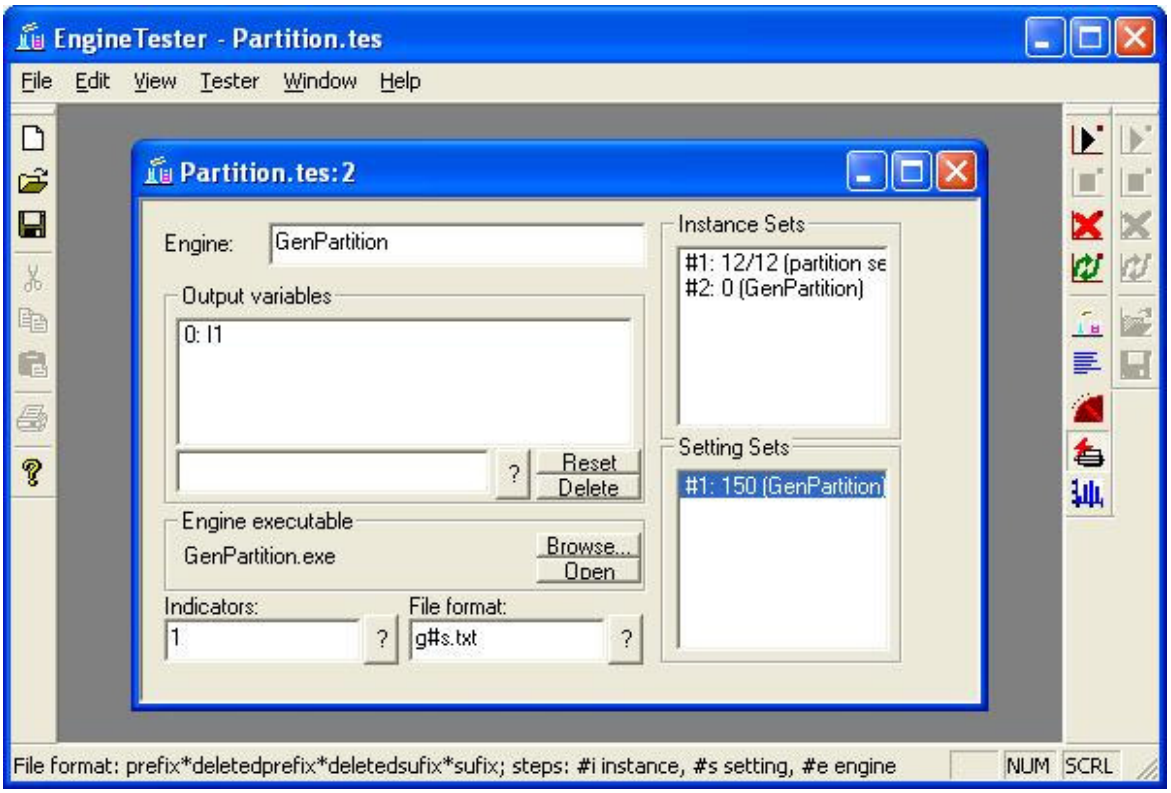
Compile e corra, fazendo novamente os mesmos testes. O gerador está pronto para ser utilizado.

Volte a abrir no EngineTester o mesmo documento, e adicione um novo engine (o GenPartition). O gerador necessita de valores para N e M, e é necessário dizer quantas instâncias se quer. Para fazer esta configuração tem que se criar um conjunto de parâmetros (setting sets).

Edite o conjunto de parâmetros criado de omissão (ou crie um novo), e dê o nome de "GenPartition". Para o valor do primeiro parâmetro (N) colocar "10-100:10", e para o segundo parâmetro colocar "10 100 1000". De forma a ter mais do que uma instância para um conjunto de parâmetros, colocar na semente (Seed Value) 5 valores "1-5".

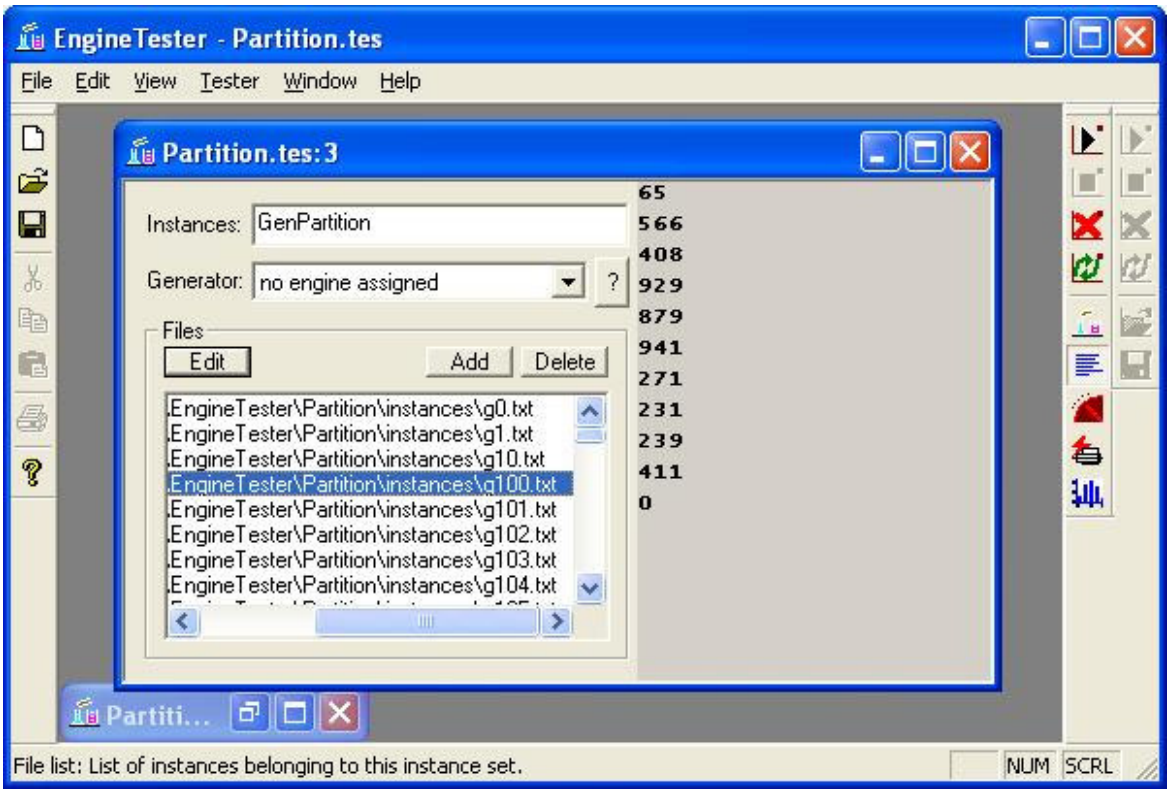


Abra agora o engine GenPartition e seleccione o conjunto de parâmetros criado. Coloque no formato para gravação a string "g#s.txt" de forma a gravar as instâncias.



Mande correr, e os ficheiros devem ficar gravados. Mova os ficheiros criados para a directoria de instâncias do Partition e crie agora um novo conjunto de instâncias, e adicione os ficheiros gravados para

os ver.



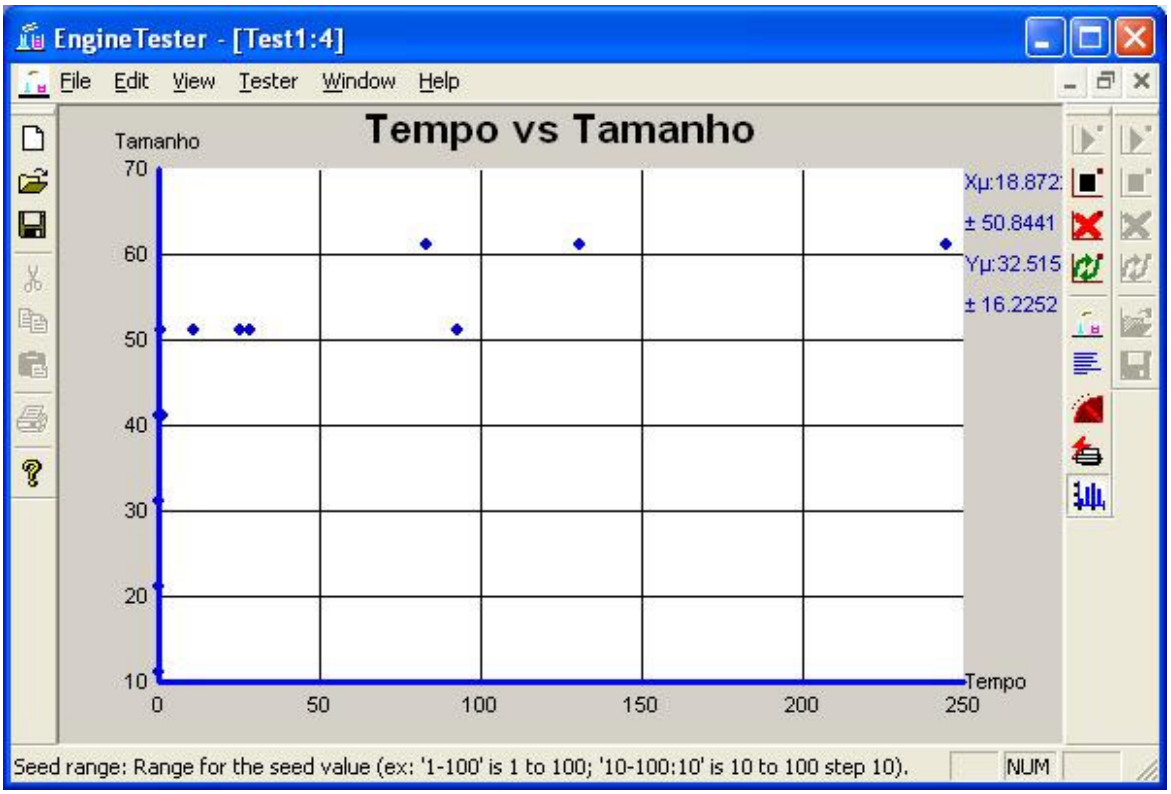
Notar que se utilizou o TRand::rand() para gerar números aleatórios, a instância g100.txt visualizada na imagem anterior, deverá coincidir com a sua, dado que a semente utilizada na sequência aleatória terá sido a mesma. Verifique em qualquer das formas se tem uma chamada no método Run() ao método Reset(), e no caso positivo apague-a, caso contrário os valores dos settings alterados são ignorados e substituidos pelos valores de omissão. Pode confirmar que está tudo em condições relativamente à instância g0.txt que deve ter os valores: 5 6 8 9 9 1 1 5 9 1 0.

Pode agora facilmente repetir os testes do Partition utilizando apenas as instâncias geradas, bastando para isso trocar a selecção do conjunto de instância a utilizar, na dialog de Engine do Partition. Nesse caso, não interessa o resultado do gerador, pelo que há que limpar as variáveis de saída, de forma a que este não corra. Antes de mandar correr, abra a vista gráfica e limpe os resultados entretanto armazenados, de forma a ir vendo os resultados a saírem à medida que são processados.

Enquanto corre, pode abrir o Task Manager para ver a ocupação do CPU, e ir monitorizando os resultados. Haverá 4 variáveis, a primeira o tempo, a segunda o resultado (que deverá ser sempre 1, devido a que todas as instâncias são possíveis), a terceira o número de instância e a quarta o número de elementos.

Será interessante ver o tempo vs tamanho, de forma a verificar se há dependência. Para isso há que primeiramente apagar do relatório as variáveis que não se querem (a 1 e 2, ficando a 0 e 4). As variáveis continuam disponíveis para outros relatórios. Botão da direita, seguido de Edit/Delete/Columns... e escrever "1 2". Apartir desse momento apenas se vê informação de duas variáveis.

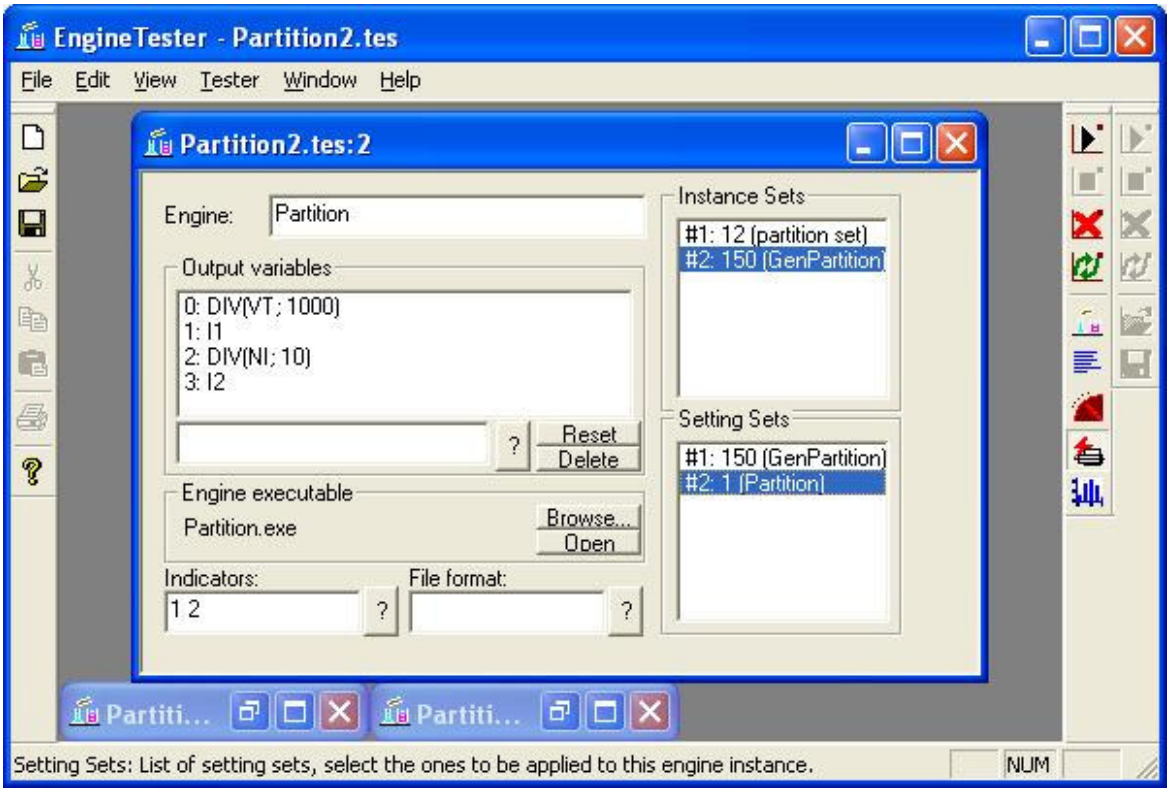
Agora há que passar a vista para tipo Scatterplot: botão da direita, seguido de Settings/Type/Scatterplot. Falta agora dar nomes certos, através também do botão da direita, seguido de Edit/Labels/Axis e Edit/Labels/Caption. Estas alterações ficam guardadas no documento.



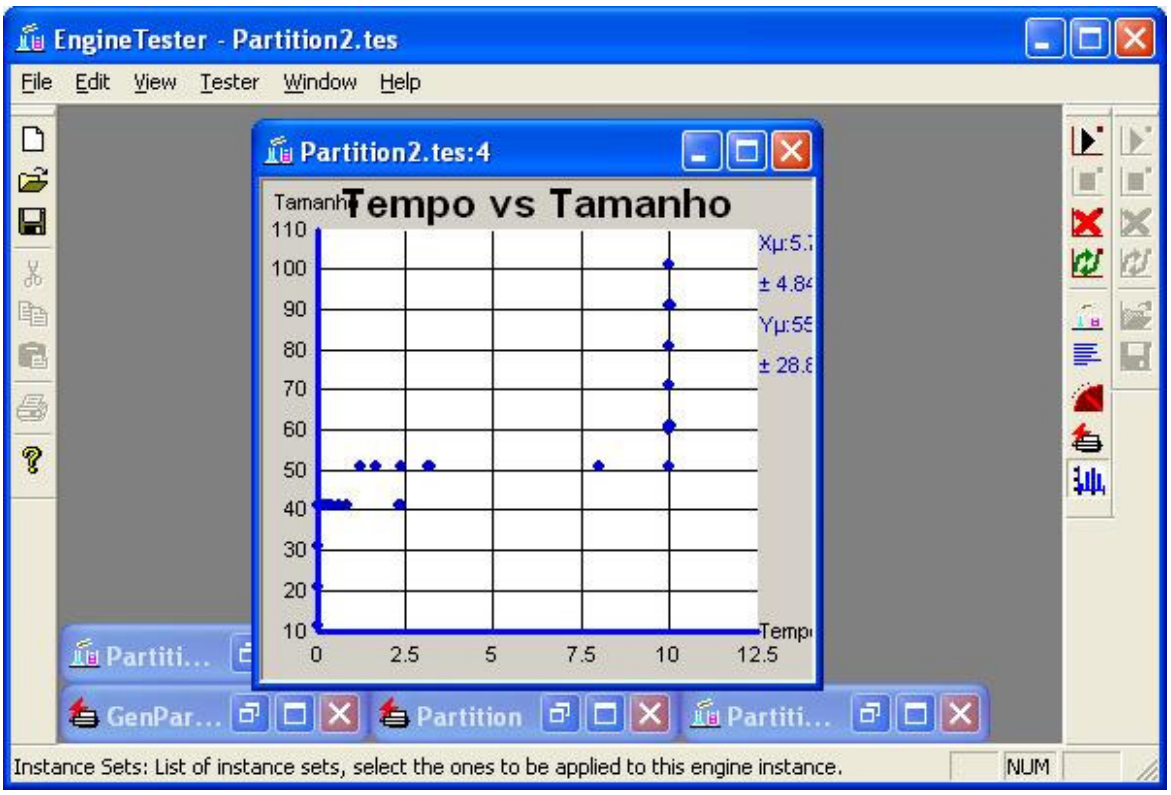
Verifica-se que o algoritmo, nas instâncias de 60 elementos já leva muito tempo. Como o nosso engine não tem verificações para abortar o algoritmo, este não pára senão quando acabar, retornando sucesso ou insucesso.

Na nossa implementação, no método "partition" há que actualizar o número de estados analisados, adicionando logo na primeira linha o comando: value_states++; e no mesmo método, antes da primeira chamada recursiva, executar o comando: if(Stop()) return 0;. Desta forma o algoritmo retornará assim que um dos critérios de paragem forem atingidos, com o valor de insucesso.

Após compilar, volte a abrir o EngineTester no mesmo documento, e crie um novo conjunto de parâmetros Partition, e coloque no limite de tempo 10000 milisegundos, e no número de estados 100000000. Se o número de estados permitido for inferior, é provável que o algoritmo páre não devido ao limite de tempo mas sim ao limite de estados. Corra novamente, mas antes não se esqueça de no Engine Partition seleccionar o respectivo conjunto de parâmetros, e limpar os resultados anteriores.



Se correr de imediacto, verá que o limite de tempo é ignorado, acabando todas as instâncias practicamente instantaneamente. Tal deve-se ao facto de no método "Load" haver uma chamada ao método "Reset", sendo nesse método reposto o tempo limite de omissão (1 segundo) e limite de número de estados (1000 o que é pouco neste caso). Há que trocar a chamada a "Reset" no método "Load" pelos comandos: instance.Count(0); resultado=0;

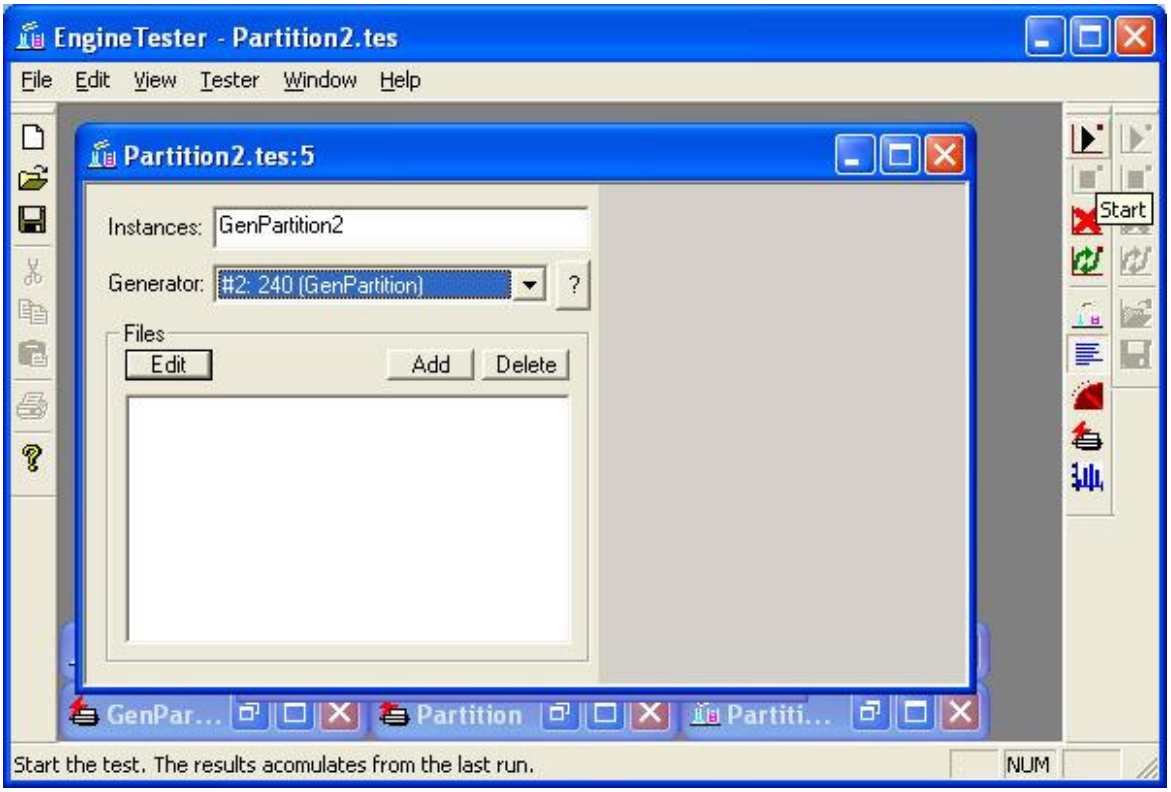


Correndo em outro computador, o gráfico será idêntico, apenas os pontos podem ser ligeiramente deslocados para a esquerda ou direita, conforme o computador é mais rápido ou mais lento, mas devido às características exponenciais do algoritmo, estas diferenças não fariam grande diferença. Caso o limite seja o número de estados, não teria havido diferença nenhuma.

Pode-se observar facilmente pelo gráfico, que de 30 elementos a 60 elementos as instâncias vão de todas simples a todas complexas, ao ponto de nenhuma instância ser resolvida dentro do tempo limite. Após esta análise, há que então fazer outra análise mais fina, de 30 a 60 elementos, e portanto ter que repetir o process.

O EngineTester permite no entanto aliviar este processo, não sendo necessário que as instâncias sejam gravadas. É sempre conveniente gravar e ver as instâncias quando se está a desenvolver o gerador, mas após ter a certeza que este funciona bem, pode-se utilizá-lo apenas para gerar instâncias de determinado tipo, sendo as instâncias utilizadas directamente no gerador.

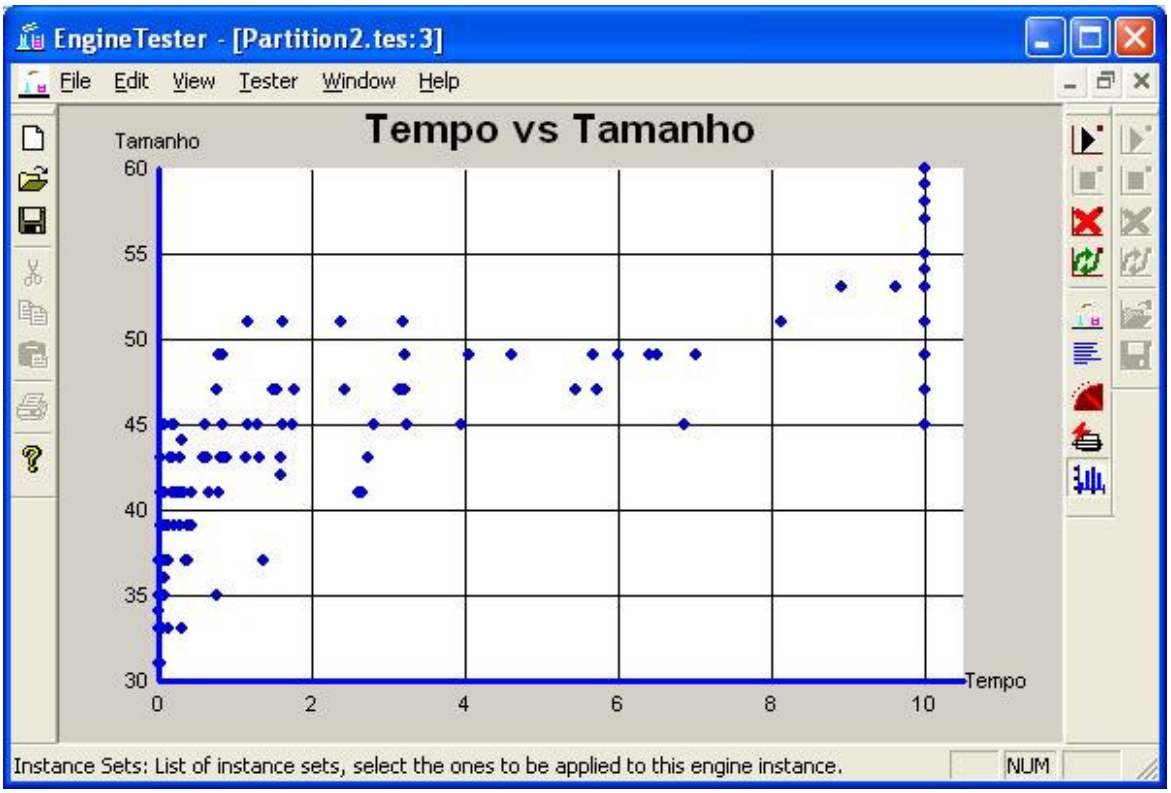
Edite o conjunto de parâmetros GenPartition, e altere os valores do parâmetro 1 para "30-60:2". No engine GenPartion, apague o formato de escrita para não gravar ficheiros, e crie um conjunto de instâncias seleccionando este gerador.



No engine Partition, seleccione o novo conjunto de instâncias GenPartition2, limpe os resultados e mande correr. Os resultados são refeitos automaticamente. A vista gráfica volta a dar informações preciosas sobre a fase de transição entre instâncias dificeis e simples.

Atenção que os resultados podem ser mal lidos. Com 50 elementos no gráfico não aparece nenhuma instância simples, ou seja, que se resolva em menos de 1 segundo. Isso não quer dizer que não existam instâncias simples, apenas que o nosso gerador não as gera, uma vez que tivemos o cuidado de manter equilibrado o número de elementos em cada selecção. Caso tivesse sido utilizada a primeira versão do gerador, teriamos muitas mais instâncias simples, nubelando os gráficos.

Este algoritmo fica validado em termos práticos para resolver qualquer instância até 40 elementos. Este tipo de informação não nos dá uma análise puramente analítica, que resultaria a informação de que o algoritmo tem complexidade O(2^n).

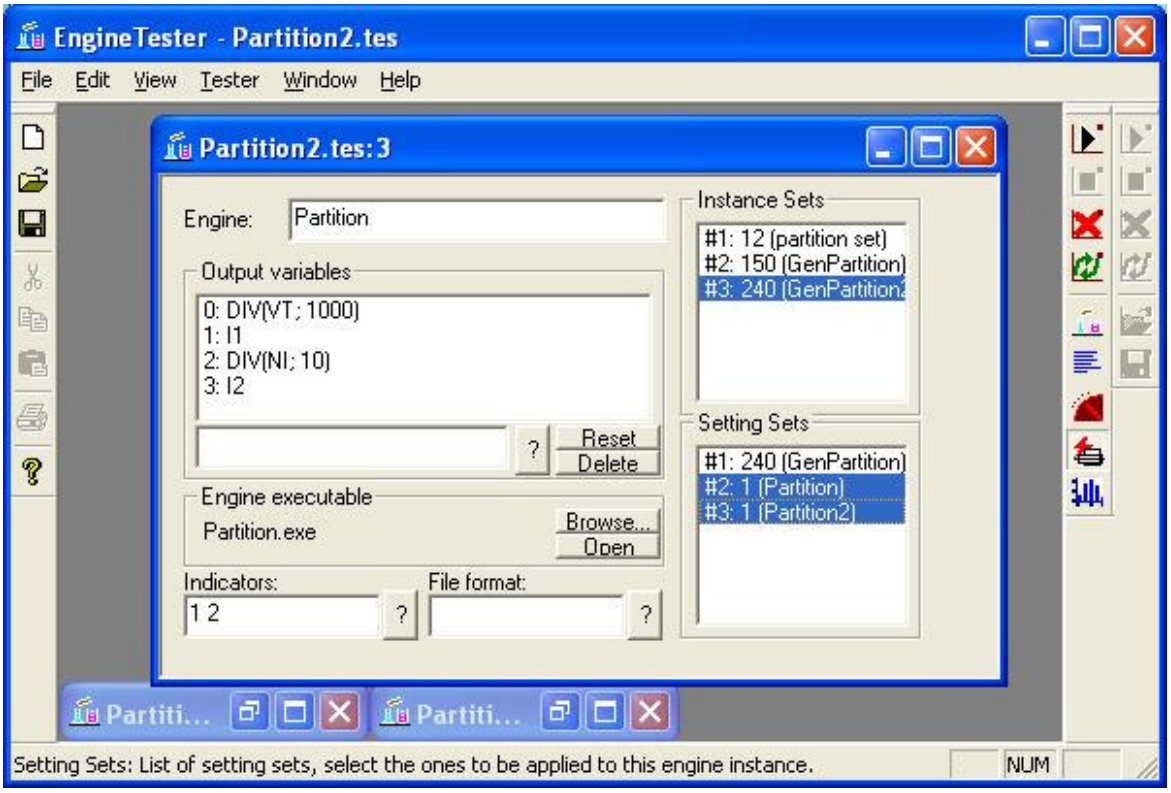


Suponhamos agora que desconfiava que o algoritmo iria correr mais rápido se escolher primeiro os elementos maiores e só depois os menores. Há hipótese de implementar isso facilmente, basta que a instância seja ordenada antes de arrancar com o algoritmo. Tem também que comparar essa opção com a actual, uma vez que a alteração requer mais tempo a ordenar, o que poderia não compensar.

No Partition há que adicionar um parâmetro (no método "Reset"), que de omissão está a 0 e se for 1 a instância é ordenada: value.Add(0); name.Add("Ordenar"); description.Add("0 - não ordenar; 1 - ordenar"); min.Add(0); max.Add(1);

No método "Run", antes de chamar partition, mandar remover os zeros e ordenar. Não esquecer de adicionar o zero no fim, para ficar uma instância válida, no caso de se gravar. O último comando passa a ser antecedido por: if(value[0]==1) { if(total%2==0) { instance.Remove(0); instance.Sort(); resultado=partition(instance.Count()-1,0,total/2); instance.Add(0); } else resultado=0; } else [último comando]

Agora basta compilar e no mesmo ficheiro do EngineTester, no conjunto de parâmetros de Partition colocar o primeiro parâmetro a 0, e duplicar este conjunto de parâmetros, ficando Partition2, e colocar o primeiro parâmetro a 1. No engine Partition, há que colocar ambos os conjuntos de parâmetros.

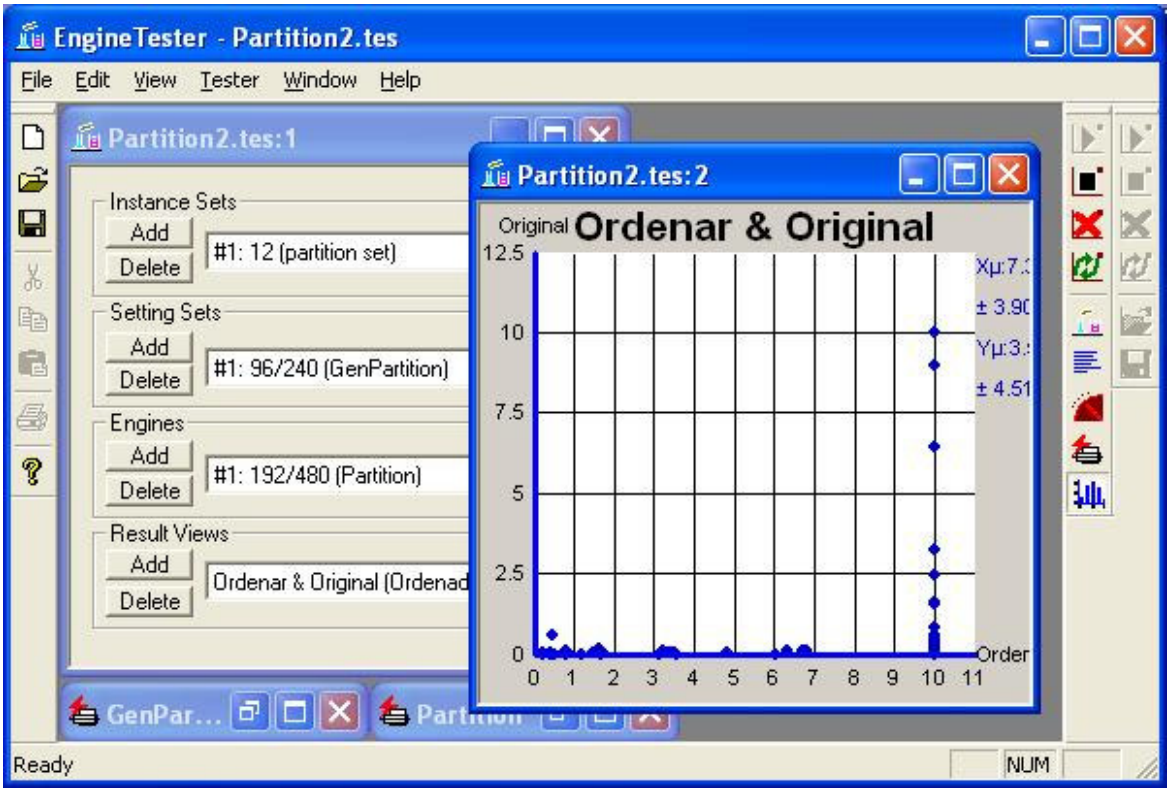


Naturalmente o número de corridas a efectuar duplicam, passa agora a 480 corridas, mas permite fazer a comparação dos desempenhos com e sem a ordenação. Pode mandar correr após limpar os resultados actuais, que levará tempo, e os gráficos fazem-se com as corridas em curso.

Pretende-se agora uma vista tipo a anterior, mas apenas para o caso do conjunto de parâmetros ser Partition. Para tal há que na dialog de base selecionar o conjunto de parâmetros Partition, e criar uma vista, respondendo sim à pergunta de ser restrito ao conjunto de parâmetros Partition. Apague desde já todas as colunas excepto o tempo "1-3".

Seleccione agora o conjunto de parâmetros Partition2 e crie uma vista, respondendo sim não só à pergunta de ser restrito ao conjunto de parâmetros Partition2, como também a fundir com a vista anterior. Apague também aqui as colunas "1-3", ficando apenas com duas linhas, com a informação do tempo gasto em ambas as opções.

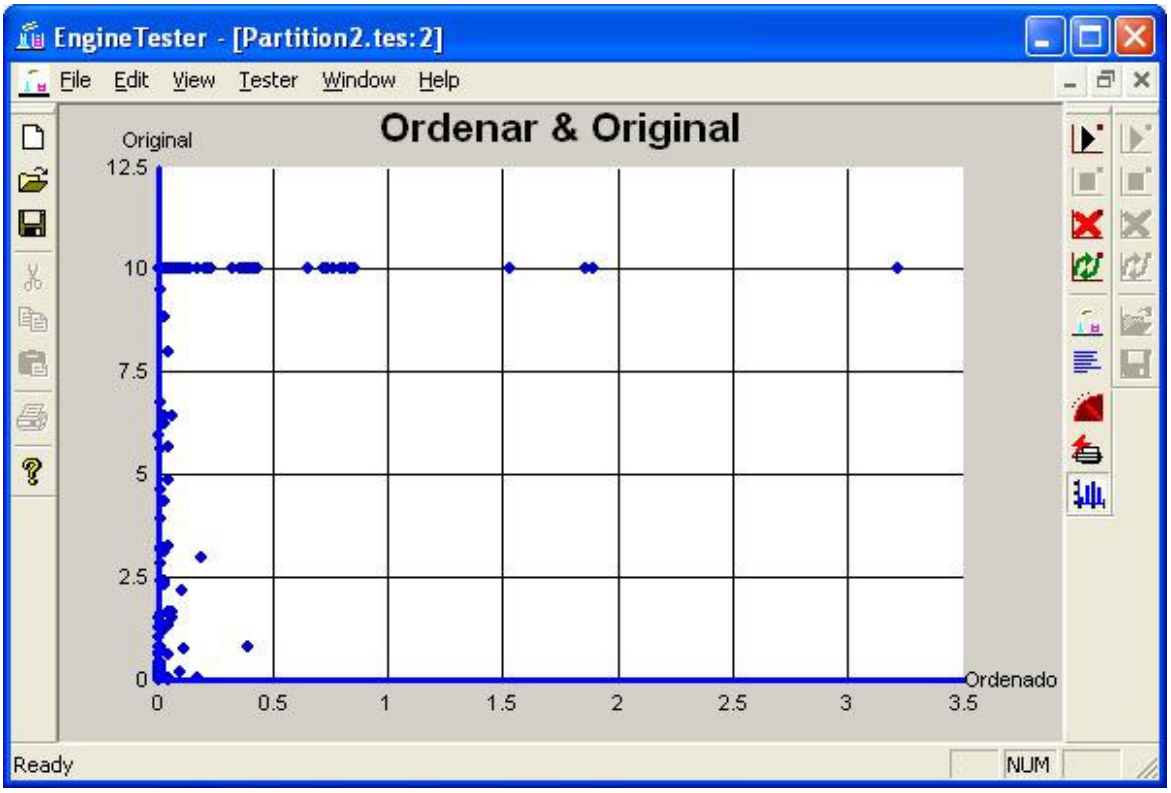
Agora pode construir um scatterplot e comparar directamente os tempos gastos num caso e no outro. Neste caso pode-se ver que o desempenho do algoritmo piora claramente. Se houver dúvidas de qual é o eixo com as novas opções, bastará abrir a vista do tamanho vs tempo e verifica-se que os resultados pioraram na globalidade.



Pode-se parar a corrida, já que agora suspeita-se que invertendo a ideia dê bom resultado, ou seja, processar primeiro os números mais baixos e depois os mais altos. Embora intuitivamente a primeira ideia faça mais sentido, no campo dos testes há que olhar para os resultados.

Dado que a inversão não está implementada, há que incluir após a ordenação (comando instance.Sort();) código para inverter a instância: for(int i=0,j;i<instance.Count()/2;i++) { j=instance[i]; instance[i]=instance[instance.Count()-i-1]; instance[instance.Count()-i-1]=j; }

Ao mandar correr, observa-se que até a actual dimensão das instâncias, de 30 a 60 elementos, não está convenientemente dimensionada para o método ordenar, dado que todas as instâncias são resolvidas, levando a mais difícil pouco mais de 3 segundos. Haverá agora que passar este parâmetro a ter um valor fixo, deixando de ser parâmetro, ou pelo menos o valor de omissão passar a ser 1, em vez de 0.



O processo repete-se, voltando a identificar novo teto do algoritmo, e volta-se a testar novas ideias. Sem EngineTester, quanto tempo levaria a montar as corridas, fazer os gráficos e a reagir? Provavelmente demasiado tempo, e ideias para melhoramento tivessem ficado por testar dado o custo de fazer novamente todas as análises.

Muitas vezes faz-se tantas análises que se perde a origem de algumas análises. No EngineTester poupa-se também no facto de se conseguir explicar como se fez cada análise, dado que o documento EngineTester tem toda a informação sobre como é feita cada análise, e está sempre pronto a refazê-la.

Paramos por aqui, o leitor pode completar a análise de forma a concluir em termos práticos até que instâncias pode ser utilizado este algoritmo com sucesso.

Foram demonstradas as principais funcionalidades do EngineTester, embora algumas delas não com o detalhe desejado. Esperamos no entanto que os exemplos de engines e análises dados no capítulo seguinte sejam suficientes.

7 - Instalação e Manutenção

Após o sistema construído e devidamente testado, resta colocar o sistema em utilização. Para tal há que formar os utilizadores do sistema e construir documentação necessária para que o sistema possa ser utilizado com sucesso.

7.1 Instalação

A instalação não deve ser efectuada sem uma documentação. O sistema pode ser facilmente instalado pela equipa de desenvolvimento que acabou de desenvolver o software, não necessita naturalmente nenhuma documentação. O problema é quando o hardware necessitar de ser actualizado, ou o sistema operativo, e as pessoas envolvidas no desenvolvimento da aplicação estão em outros projectos ou em outra empresa.

A documentação deve descrever o processo de backup, e mais importante, o processo de restore de todo o sistema. Forçar um ou outro restore de supresa, de forma a conhecer o que acontecerá numa situação de emergência, é uma atitude sábia. É boa prática no caso de sistemas essenciais, existir uma máquina com o sistema instalado, pronta a ser utilizada caso exista algum problema com o sistema em produção. Assim é possível testar as actualizações antes de estas serem feitas no sistema em produção.

7.2 Formação de Utilizadores

A formação de um utilizador deverá incidir primeiramente pelas operações básicas e mais frequentes do sistema, evoluindo para as operações mais complexas e menos frequentes. A formação pode relacionar a forma como as operações são efectuadas no sistema actual, e como serão efectuadas no sistema novo. Um utilizador precisa apenas de saber o que pode fazer com o sistema e como utilizá-lo. Não necessita de saber detalhes de como o sistema funciona internamente.

Para além dos utilizadores do sistema, há também os operadores ou administradores, que efectuam operações de administração do sistema. A formação dos administradores deve ser direccionada ao funcionamento interno do sistema, e às operações que têm de ser efectuadas regularmente para que este funcione correctamente. Algumas dessas operações podem ser: dar acessos a utilizadores; fazer backups do sistema; instalar novos módulos. Os administradores têm também de dar apoio aos utilizadores do sistema, de modo que têm também de ter a formação dos utilizadores.

Poderá haver utilizadores que utilizam o sistema apenas esporadicamente, ou apenas para consulta, devendo haver uma formação mais leve para esses utilizadores.

A formação deve ter em atenção os conhecimentos prévios dos utilizadores. Por exemplo, caso existam utilizadores não familiarizados com computadores, é necessário um curso de iniciação a esses utilizadores. A formação deve ser dividida em unidades pequenas.

A formação não deve ser apenas numa sessão na entrega do sistema. Deve estar disponível sempre que o utilizador necessitar, por diferentes formas: manual impresso; manual online; sessões de formação; utilizadores experientes.

7.2.1 Manual Impresso

Os manuais impressos têm de ser completos e portanto acabam por serem volumosos. Normalmente são considerados pelos utilizadores como não essenciais, muitas vezes porque pretendem fazer apenas

operações simples e não sabem como encontrar rapidamente a informação que necessitam, e não querem perder muito tempo a ler o manual. Os manuais devem ser consultados quando há dúvidas, ou antes de uma operação pouco frequente, e devem ter hipótese de serem utilizados tanto em modo de consulta rápida (manual de referência), como em modo de formação (manual tutorial).

Um manual de utilizador deve conter no mínimo os seguintes elementos:

- Um esquema geral com as principais funcionalidades do sistema e de como se relacionam entre si;
- Uma descrição dos dados necessários a cada funcionalidade do sistema;
- Uma descrição dos dados produzidos por cada funcionalidde do sistema;
- Uma descrição de características especiais de cada funcionalidade do sistema.

7.2.2 Manual Online

O manual online, juntamente com nomes de menus e icons sugestivos, e toda a interface do sistema no geral, são outra forma de formação, normalmente a mais utilizada. O manual online pode ser uma cópia do manual impresso ou uma versão reduzida deste, mas deverá ter hipótese de ser acedido de diversos locais com ligações directas à página do manual onde é descrito o comando que poderia ser executado. No manual impresso as referências a outras zonas do manual são feitas por indicação da página, no manual online estas devem ser através de hiper-texto, de forma a obter-se a página à distância de um clique.

7.2.3 Sessões de Formação

Na formação, muitos utilizadores preferem demonstrações do software. As demonstrações podem ser feitas presencialmente, ou gravadas, e normalmente focam um aspecto do sistema de forma a não serem demasiado longas. As demonstrações funcionam melhor para muitas pessoas que mantêm a atenção durante mais tempo em apresentações verbais do que em apresentações escritas. Deve-se encorajar a participação dos utilizadores nas demonstrações.

7.2.4 Utilizadores Experientes

Na entrega do sistema, a formação prévia de utilizadores avançados pode facilitar a formação dos restantes utilizadores, que assim podem ser assistidos pelos utilizadores avançados durante a formação e após a formação. Este tipo de abordagem é aconselhado em sistemas complexos, em que se anteveja que a maior parte dos utilizadores venha a ter grandes dificuldades a utilizar o novo sistema.

7.3 Manutenção

Após o sistema estar em funcionamento há sempre operações de desenvolvimento que se necessita de fazer, para não só corrigir falhas que se encontrem entretanto, como para desenvolver novos módulos que venham a revelar-se importantes. A manutenção é a actividade de correcção ou desenvolvimento feita após o sistema estar em funcionamento. Tudo o que for sendo feito deve ficar registado de forma a poder ser repetido quando se fizer actualizações ao sistema.

Com o sistema em funcionamento, ao longo dos tempos o custo de manutenção pode ir aumentando. Há que decidir o ponto em que se toma a decisão de substituir o sistema actual por um novo. Para tal há que analisar: custo de manutenção; fiabilidade do sistema; capacidade de evolução do sistema; alternativas no mercado.

Para basear as decisões deve-se calcular a fiabilidade, disponibilidade e manutenabilidade do software,

bem como registar outros indicadores de manutenção.

Caso se mantenha o sistema em funcionamento existem os seguintes tipos de manutenção:

- Manutenção correctiva – resolução de falhas que ocorram no dia-a-dia;
- Manutenção adaptativa – implementação de mudanças em componentes necessárias devido a uma outra mudança no sistema;
- Manutenção aperfeiçoativa – melhoramentos efectuados no sistema sem que tenham origem em falhas;
- Manutenção preventiva – alteração do sistema de forma a torná-lo mais robusto, sem que tenha no entanto existido falhas.

Naturalmente que sistemas de grande dimensão e essenciais ao negócio da empresa, requerem recursos humanos atribuídos permanentemente à manutenção.

7.3.1 Fiabilidade, Disponibilidade, Manutenabilidade

A fiabilidade do software está relacionada com a probabilidade de este falhar. Um software altamente fiável tem uma baixa probabilidade de falhar. A disponibilidade do software está relacionada com a probabilidade do sistema estar operacional num dado momento. A manutabilidade do sistema está relacionada com a facilidade de manutenção do software.

Há basicamente dois tipos de incertezas, a primeira é do tipo 1, reflecte o nosso desconhecimento de como o sistema será utilizado, e portanto é impossível conhecer quanto tempo levará até à próxima falha. O segundo tipo de incerteza, do tipo 2, reflecte a nossa falta de conhecimento sobre as implicações no resto do sistema da correcção de uma falha.

Estas grandezas podem ser medidas, desde que para tal se recolha a informação necessária:

- Tempo para falha – o valor médio é "Mean Time to Failure" MTTF;
- Tempo de reparação – o valor médio é "Mean Time to Repair" MTTR.
- Com estes valores pode-se também calcular o valor médio do tempo entre falhas: "Mean Time between Failures" MTBF.

Com base nestes valores médios pode-se calcular rácios para que o valor 1 seja o ideal e o valor 0 indesejado, para as três grandezas fiabilidade, disponibilidade e manutenabilidade:

- $R = \frac{MTTF}{1+MTTF}$ – fiabilidade;
- $A = \frac{MTBF}{1+MTBF}$ – disponibilidade;
- $M = \frac{1}{1+MTTR}$ – manutenabilidade.

7.3.2 Indicadores de Manutenção

Para que exista algum controle sobre a manutenção, devem ser registadas informações básicas das falhas encontradas. No cálculo do tempo de reparação deve ser descriminado as seguintes parcelas:

- tempo que levou a ser reportado um problema;
- tempo perdido em processos administrativos;
- tempo necessário para analisar o problema;
- tempo necessário para especificar as mudanças a fazer;
- tempo necessário para fazer as mudanças;

- tempo necessário para testes;
- tempo necessário para documentar.

Há também indicadores de complexidade do código a manter, que se podem calcular de forma a ter uma ideia dos custos de manutenção futuros. Utiliza-se normalmente o número ciclomático para complexidade de código. Há também a possibilidade de calcular indicadores sobre documentação, de forma a contabilizar a legibilidade do texto. Esses indicadores não estão no entanto muito desenvolvidos.

O número ciclomático é o número de caminhos diferentes no código. Mostrou-se que esse número é igual ao número de comandos de decisão mais 1. É portanto muito fácil de calcular, mesmo para uma grande quantidade de linhas de código.

O número ciclomático permite dar uma ideia da complexidade de um bloco de código, mas ao decidir efectuar uma alteração ao código, o custo estimado nessa alteração poderá passar não só pelo módulo afectado, mas também pelos módulos que este interage, e pela revisão da documentação associada. Os testes que estão ligados aos módulos afectados, têm de ser refeitos, e a documentação actualizada. Todos estes custos têm de ser levados em consideração, para saber se vale a pena efectuar a correcção, ou se por outro lado se deixa como está. Notar que após identificar um problema, a sua resolução poderá passar por corrigir apenas o código, mas pode também passar por alterar o desenho, ou mesmo os requisitos, levando a custos muito superiores nestes dois últimos casos.

7.3.3 Rejuvenescimento de Software

O rejuvenescimento de software destina-se a aumentar a qualidade do software em manutenção com vista em reduzir os custos de manutenção futuros. Há diversas estratégias para tal:

- re-documentação – efectuar uma análise ao código, de forma a produzir documentação explicativa do código;
- re-estruturação – efectuar uma análise ao código, de forma a re-estruturar o código para uma estrutura mais correcta;
- engenharia inversa – efectuar uma análise ao código, de forma a gerar o desenho e especificações que lhe deram origem;
- re-engenharia – efectuar uma análise ao código, de forma a gerar o desenho e especificações que lhe deram origem, e com estas gerar novo desenho e novo código.

"Software Engineering, theory and practice", second edition, Prentice Hall, Shari Pfleeger, pág. 403-407, 448-460, 464-502