Engenharia de Software

Capítulo 7

Linguagens de Programação

1. Introdução

Todas as etapas discutidas até o momento no contexto do curso são desenvolvidas visando a obtenção de um código executável que será o elemento central do produto de software. Sob esta ótica, o código do programa nada mais é do que uma formalização, numa dada linguagem, das idéias analisadas na etapa de engenharia do sistema, dos requisitos estabelecidos na etapa de análise e dos modelos gerados na etapa de projeto de modo a que o sistema computacional alvo possa executar as tarefas definidas e refinadas nas etapas anteriores.

Na etapa de Codificação, que é o nome que daremos ao conjunto de atividades relacionadas a esta formalização, é feito uso de linguagens de programação como forma de expressar, numa forma mais próxima da linguagem processada pela máquina, os aspectos definidos nas etapas precedentes.

Num processo de desenvolvimento de software onde os princípios e metodologias da Engenharia de Software sejam aplicados corretamente, a codificação é vista como uma conseqüência natural do projeto. Entretanto, não se deve deixar de lado a importância das características da linguagem de programação escolhida nesta etapa nem o estilo de codificação adotado.

2. A Tradução

Na etapa de codificação, o programador procura mapear corretamente as representações (ou modelos) obtidos no projeto detalhado nas construções suportadas por uma dada linguagem de programação. O resultado deste mapeamento constitui o **código-fonte**, o qual será, na seqüência, traduzido por um compilador gerando o **código-objeto**. Finalmente, o código-objeto será mapeado nas instruções executáveis pelo microprocessador que compõe o sistema computacional, resultando no **código de máquina**.

Ainda em grande parte dos casos, o primeiro nível de mapeamento é obtido de forma manual ou semi-automatizada, estando sujeito, portanto, a "ruídos" inerentes do processo. Alguns exemplos de ruídos são:

- interpretação inadequada de aspectos da especificação de projeto;
- restrições ou complexidades características da linguagem de programação adotada;
- etc...

As restrições impostas por uma dada linguagem de programação ou o conhecimento incompleto das suas potencialidades pode conduzir a raciocínios (e conseqüentes projetos) relativamente limitados.

3. CARACTERÍSTICAS DAS LINGUAGENS DE PROGRAMAÇÃO

Como já foi mencionado, as linguagens de programação funcionam no processo de desenvolvimento de software como o agente de formalização das representações de projeto perante o sistema computacional. As características de uma linguagem de programação podem exercer um impacto significativo no processo de codificação segundo diferentes ângulos, os quais serão abordados nos parágrafos que seguem.

3.1. Características Psicológicas

Sob a ótica psicológica, aspectos tais como a facilidade de uso, simplicidade de aprendizagem, confiabilidade e baixa freqüência de erros são características que devem ser apresentadas pelas linguagens de programação; estes fatores podem causar impacto significativo no processo de desenvolvimento de um software, considerando a codificação como uma atividade intrinsecamente humana. As características de ordem psicológica são apresentadas abaixo.

3.1.1. Uniformidade

Esta característica está relacionada à consistência (coerência) da notação definida para a linguagem, a restrições arbitrárias e ao suporte a exceções sintáticas e semânticas. Um exemplo (ou contra-exemplo) significativo desta característica está na linguagem Fortran, que utiliza os parênteses para expressar dois aspectos completamente diferentes na linguagem — a precedência aritmética e a delimitação de argumentos de subprogramas).

3.1.2. Ambigüidade

Esta é uma característica observada pelo programador na manipulação da linguagem. Embora o compilador interprete a instrução de um único modo, o leitor (programador) poderá ter diferentes interpretações para a mesma expressão. Um caso típico de problema relacionado a esta característica é a precedência aritmética implícita. Por exemplo, a expressão **`X = X1/X2*X3`** pode levar a duas interpretações por parte do leitor: **`X = (X1/X2)*X3`** ou **`X = X1/(X2*X3).`**

3.1.3. Concisão

Esta é uma outra característica psicológica desejável nas linguagens de programação que está relacionada à quantidade de informações orientadas ao código que deverão ser recuperadas da memória humana. Os aspectos que influem nesta característica são a capacidade de suporte a construções estruturadas, as palavras-chave e abreviações permitidas, a diversidade com relação aos tipos de dados, a quantidade de operadores aritméticos e lógicos, entre outros.

3.1.4. Localidade

A localidade está relacionada ao suporte a construções que possam ser identificadas em blocos. O leitor identifica um conjunto de instruções a partir da visualização de um de seus componentes. Por exemplo, as construções do tipo ***IF-THEN-ELSE***, REPEAT-UNTIL, entre outras, são exemplos de construções que aumentam a característica de localidade numa linguagem.

3.1.5. Linearidade

A linearidade está relacionada à manutenção do domínio funcional, ou seja, a percepção é melhorada quando uma seqüência de instruções lógicas é encontrada. Programas caracterizados por grandes ramificações violam esta característica.

3.2. Características de Engenharia

Numa visão de engenharia de software, as características das linguagens de programação estão relacionadas às necessidades do processo de desenvolvimento do software. Algumas características que podem ser derivadas segundo esta ótica são:

3.2.1. Facilidade de derivação do código-fonte

Esta característica está ligada à proximidade da sintaxe da linguagem de programação com as linguagens de representação de projeto. Na maior parte dos casos, as linguagens que suportam construções estruturadas, estruturas de dados relativamente complexas, possibilidade de manipulação de bits, construções orientadas a objetos e construções de entrada/saída especializadas permitem mapear mais facilmente as representações de projeto

3.2.2. Eficiência do compilador

Esta característica contempla os requisitos de tempo de resposta e concisão (exigência de pouco espaço de memória) que podem caracterizar determinados projetos.

A maioria dos compiladores de linguagens de alto nível apresenta o inconveniente de geração de código ineficiente.

3.2.3. Portabilidade

Este aspecto está relacionado ao fato do código-fonte poder migrar de ambiente para ambiente, com pouca ou nenhuma alteração. Por ambiente, pode-se entender o processador, o compilador, o sistema operacional ou diferentes pacotes de software.

3.2.4. Ambiente de Desenvolvimento

Um aspecto fundamental é, sem dúvida, a disponibilidade de ferramentas de desenvolvimento para a linguagem considerada. Neste caso, deve ser avaliada a disponibilidade de outras ferramentas que as de tradução, principalmente, ferramentas de auxílio à depuração, edição de código-fonte, bibliotecas de subrotinas para uma ampla gama de aplicações (interfaces gráficas, por exemplo).

3.2.5. Manutenibilidade

Outro item de importância é a facilidade em alterar o código-fonte para efeito de manutenção. Esta etapa, em muitos casos negligenciada nos projetos de desenvolvimento de software, só poderá ser conduzida eficientemente a partir de uma completa compreensão do software.

Embora a existência de documentação relativa ao desenvolvimento do software seja de fundamental importância, a clareza do código-fonte vai ser fator determinante para o sucesso das tarefas de manutenção.

3.3. Características técnicas

As duas classes de características apresentadas anteriormente são, de certo modo, conseqüência da forma como as linguagens de programação foram concebidas, tanto do ponto de vista sintático como semântico. A seguir, serão apresentadas as principais características "técnicas" das linguagens de programação, as quais exercem forte impacto na qualidade do software e nas condições sob as quais este é desenvolvido.

3.3.1. Suporte a tipos de dados

Quanto mais poderosos são os programas atuais, mais complexas são as estruturas de dados que estes manipulam. Nas linguagens de programação atuais, o suporte à representação dos dados é caracterizado por diferentes categorias de dados, desde estruturas extremamente simples até estruturas com alto grau de complexidade. A seguir são enumeradas algumas destas categorias:

- tipos numéricos (inteiros, complexos, reais, bit, etc...);
- tipos enumerados (valores definidos pelos usuários... cores, formas, etc...);
- cadeias de caracteres (strings);
- tipos booleanos (verdadeiro/falso);
- vetores (uni ou multidimensionais);
- listas, filas, pilhas;
- registros heterogêneos.

Na maior parte dos casos, a manipulação lógica e aritmética de dados é controlada por mecanismos de verificação de tipos embutidos nos compiladores ou interpretadores. O objetivo deste mecanismo é garantir a coerência dos programas com relação às estruturas de dados e às operações que podem ser realizadas sobre eles.

3.3.2. Os subprogramas

Os subprogramas correspondem a componentes de programa contendo estruturas de controle e dados que podem ser compilados separadamente. Com relação aos elementos de um software definidos na etapa de projeto, pode-se, eventualmente, considerar subprogramas como o mapeamento natural dos módulos de um software, embora não necessariamente os módulos (que correspondem a uma definição mais genérica) serão mapeados em subprogramas. Os subprogramas recebem diferentes denominações, dependendo da linguagem de programação considerada (subrotinas, procedimentos, funções, módulos, agentes, processos, objetos).

Independentemente da linguagem de programação considerada, os subprogramas são caracterizados pelos seguintes elementos:

- uma seção de especificação (declaração), contendo o seu identificador e uma descrição da interface;
- uma seção de implementação, onde é descrito o seu comportamento através das estruturas de controle e dados;
- um mecanismo de ativação, o qual permite invocar o subprograma a partir de um ponto qualquer do programa (por exemplo, uma instrução ***call***).

3.3.3. Estruturas de controle

Praticamente todas as linguagens de programação oferecem suporte à representação das estruturas de controle mais clássicas, como seqüências, condições e repetições. A maior parte destas linguagens oferece um conjunto de estruturas sintáticas

quase que padronizadas que permitem exprimir estruturas como IF-THEN-ELSE, DO-WHILE, REPEAT-UNTIL e CASE.

Além das estruturas clássicas mencionadas, outros fatores podem ser interessantes como suporte a aspectos de programação não contemplados nestas estruturas. Abaixo são citados alguns destes fatores:

- a ***recursividade***, que permite que um subprograma contenha em seu código uma ativação a si próprio;
- a ***concorrência***, que permite a criação de múltiplas tarefas e oferece suporte à comunicação e sincronização destas tarefas;
- o ***tratamento de exceções***, que permite ativar a execução de um conjunto de instruções a partir da ocorrência de um evento significativo do sistema computacional (erro de execução, relógio de tempo-real, etc...) ou definido pela aplicação (interrupção causada pela pressão de uma tecla, chegada de uma mensagem, etc...).

3.3.4. Suporte a abordagens orientadas a objeto

Não obstante o resultado da aplicação de uma abordagem orientada a objeto possa ser codificada a partir de qualquer linguagem de programa, o mapeamento dos elementos especificados no projeto fica mais simples se a linguagem de programação oferece suporte a esta abordagem.

Sendo assim, o suporte a conceitos como classe, herança, encapsulamento e troca de mensagens pode ser um aspecto interessante numa linguagem utilizada na implementação de um projeto orientado a objetos.

3.4. Critérios de Escolha de uma Linguagem de Programação

As características apresentadas acima devem ser levadas em conta no momento da escolha de uma linguagem de programação no contexto de um projeto de software. Por outro lado, outros critérios podem auxiliar na decisão, entre eles:

- a área de aplicação para a qual o software está sendo construído;
- a complexidade computacional e algorítmica do software;
- o ambiente no qual o software vai executar;
- os requisitos de desempenho;
- a complexidade da estrutura de dados;
- o conhecimento da equipe de desenvolvimento;
- a disponibilidade de boas ferramentas de desenvolvimento.

4. Classes de Linguagens

Devido à grande diversidade das linguagens de programação existentes nos dias atuais, é interessante estabelecer uma classificação que permita situá-las no contexto do desenvolvimento de programas. Embora os critérios de classificação possam ser os mais diversos, adotou-se aqui uma ótica cronológica, onde as linguagens posicionam-se, de certo modo, em função de suas características técnicas.

4.1. Linguagens de Primeira Geração

Nesta classe de linguagens, desenvolvidas nos primórdios da computação, enquadram-se as linguagens em nível de máquina. O código de máquina representa a linguagem interpretada pelos microprocessadores, sendo que cada microprocessador é caracterizado por uma linguagem de máquina própria, podendo, eventualmente,

estabelecer-se certa compatibilidade em software entre microprocessadores de uma mesma família (fabricante).

Embora tendo seu uso bastante limitado, alguns programas ainda são realizados utilizando o código de máquina ou, mais precisamente, seu equivalente "legível", a linguagem Assembly.

Numa ótica de Engenharia de Software, a programação através de linguagens desta classe limitam-se a situações onde a linguagem de alto nível adotada para a programação não suporte alguns requisitos especificados.

4.2. Linguagens de Segunda Geração

As linguagens de segunda geração constituem as primeiras linguagens de "alto nível", que surgiram entre o final da década de 50 e início dos anos 60. Linguagens como Fortran, Cobol, Algol e Basic, com todas as deficiências que se pode apontar atualmente, foram linguagens que marcaram presença no desenvolvimento de programas, sendo que algumas delas têm resistido ao tempo e às críticas, como por exemplo Fortran que ainda é visto como uma linguagem de implementação para muitas aplicações de engenharia. Cobol, por outro lado, ainda é uma linguagem bastante utilizada no desenvolvimento de aplicações comerciais.

4.3. Linguagens de Terceira Geração

Nesta classe, encaixam-se as chamadas linguagens de programação estruturada, surgidas em meados dos anos 60. O período compreendido entre a década de 60 e a de 80 foi bastante produtivo no que diz respeito ao surgimento de linguagens de programação, o que permitiu o aparecimento de uma grande quantidade de linguagens as quais podem ser organizadas da seguinte forma:

- as ***linguagens de uso geral***, as quais podem ser utilizadas para implementação de programas com as mais diversas características e independente da área de aplicação considerada; encaixam-se nesta categoria linguagens como Pascal, Modula-2 e C;
- as ***linguagens especializadas***, as quais são orientadas ao desenvolvimento de aplicações específicas; algumas das linguagens que ilustram esta categoria são Prolog, Lisp e Forth;
- as ***linguagens orientadas a objeto***, que oferecem mecanismos sintáticos e semânticos de suporte aos conceitos da programação orientada a objetos; alguns exemplos destas linguagens são Smalltalk, Eiffel e C++.

4.4. Linguagens de Quarta Geração

As linguagens de quarta geração surgiram como forma de aumentar o grau de abstração na construção de programas. Uma característica comum nestas linguagens, é a especificação do programa sem a necessidade de expressão de detalhes algorítmicos de processamento.

É evidente que, devido a esta característica, é impossível que uma linguagem possa ser utilizada para a programação de uso geral, sendo normalmente destinadas a aplicações em domínios específicos.

Dentro desta classe, é possível encontrar diferentes categorias de linguagens:

4.4.1. Linguagens de consulta

Nesta categoria, encaixam-se as linguagens orientadas a aplicações conjuntas de bancos de dados, permitindo que o usuário especifique operações sobre os registros num alto nível de sofisticação. Algumas linguagens nesta categoria incorporam recursos de

interpretação de linguagem natural, de modo que o usuário pode manipular a informação no nível mais alto de abstração possível.

4.4.2. Linguagens geradoras de programas

Nesta categoria estão as linguagens que permitem que o programador especifique o programa considerando construções e instruções num elevado nível de abstração, sendo que os programas serão obtidos em código-fonte de uma linguagem de programação de terceira geração.

4.4.3. Outras linguagens

Nesta categoria, pode-se considerar as linguagens desenvolvidas nos últimos anos como por exemplo as linguagens de especificação formal (referenciadas no capítulo 6), as linguagens de prototipação, e os ambientes de programação presentes em computadores pessoais (planilhas, bancos de dados, hypertexto, etc...).

5. ESTILO DE CODIFICAÇÃO

A escolha de uma linguagem de programação adequada, que suporte os requisitos de projeto estabelecidos e que apresente boas características entre o conjunto apresentado anteriormente é, sem dúvida, um grande passo na direção da obtenção de um software de qualidade.

Entretanto, um fator que é essencial para a obtenção de um código claro e que ofereça facilidade de manutenção é a boa utilização dos mecanismos presentes na linguagem adotada, o que vamos rotular aqui por “estilo de codificação”.

Abaixo, serão discutidos os principais fatores determinantes à obtenção de código-fonte bem escrito.

5.1. A Documentação de Código

O código-fonte não deve ser considerado, em nenhuma hipótese, como um resultado intermediário irrelevante no processo de desenvolvimento de um software. Ele se constitui num elemento essencial tanto para atividades de validação do software como (e principalmente) para as tarefas de manutenção.

Desta forma, o aspecto de documentação é um aspecto que deve ser bastante considerado na etapa de codificação.

Um primeiro passo na documentação do código fonte é a escolha dos identificadores de variáveis e procedimentos. O uso de “siglas” ou caracteres para identificar os elementos de um programa é uma tendência herdada das linguagens de segunda geração que não encontra mais lugar nos dias atuais. Assim, é importante que os identificadores escolhidos tenham significado explícito para a aplicação considerada.

Outro aspecto bastante considerado é a introdução de comentários ao longo do código-fonte. Na realidade este é um aspecto bastante polêmico, mas este mecanismo pode ser bastante útil se utilizado eficientemente, podendo transformar-se no principal aliado às tarefas de manutenção. Uma sistemática interessante de uso dos comentários é a compatibilização destes com a documentação de projeto do software.

Um último aspecto importante na documentação é a formatação do código-fonte. Um mecanismo importante neste contexto é o uso da endentação como meio de melhor posicionar as instruções no contexto de trechos significativos do código. A endentação permite explicitar melhor a combinação dos blocos básicos de uma linguagem de programação (as estruturas clássicas de controle) para a composição de componentes mais complexos. No que diz respeito a este aspecto muitas ferramentas CASE oferecem os chamados formatadores automáticos de código, os quais liberam o programador desta preocupação.

5.2. A Declaração de Dados

Este aspecto nem sempre é objeto de preocupação por parte dos programadores, mas, à medida que o programa vai se tornando complexo no que diz respeito à definição de estruturas de dados, o estabelecimento de uma sistemática padronizada de declaração de tipos de dados e variáveis vai assumindo um nível cada vez maior de importância.

A linguagem Pascal deu um impulso importante neste aspecto, com a possibilidade do usuário construir tipos de dados mais complexos a partir dos tipos básicos e de seus construtores. Este mecanismo está presente em praticamente todas as linguagens em uso nos dias de hoje.

5.3. A Construção de Instruções

O fluxo lógico de instruções é definido normalmente durante a etapa de projeto (projeto detalhado). Entretanto, o mapeamento deste fluxo lógico em instruções individuais faz parte do trabalho de codificação.

Um aspecto que deve ser explorado durante a construção das instruções é a simplicidade. Utilizar as possibilidades que algumas linguagens oferecem de escrever mais de uma instrução por linha, por exemplo, é sem dúvida uma decisão que vai contra a clareza do código, independente da economia de espaço (e de papel) que isto pode representar.

Abaixo, são apresentadas algumas regras importantes no que diz respeito a este aspecto:

- evitar o uso de testes condicionais complicados ou que verifiquem condições negativas;
- evitar o intenso aninhamento de laços ou condições;
- utilizar parênteses para evitar a ambigüidade na representação de expressões lógicas ou aritméticas;
- utilizar símbolos de espaçamento para esclarecer o conteúdo de uma instrução.

5.4. Entradas/Saídas

As entradas de dados num software podem ser realizadas de forma interativa ou em batch. Independente da forma como as entradas serão efetuadas, é importante considerar algumas regras:

- validação de todas as entradas de dados;
- simplicidade e padronização no formato da entrada;
- no caso de entradas envolvendo múltiplos dados, estabelecer um indicador de final de entrada (ENTER, ponto final, etc...);
- entradas de dados interativas devem ser rotuladas, indicando-se eventuais parâmetros, opções e valores default;
- rotulação de relatório de saída e projeto.

6. CODIFICAÇÃO E EFICIÊNCIA

Embora os fatores determinantes da eficiência na execução de software possam ser determinados fundamentalmente por aspectos de hardware como o processador utilizado e o espaço de memória disponível, a codificação pode ter uma participação, mesmo modesta, na redução do tempo de processamento de um programa ou na menor ocupação de espaço na memória.

De modo geral, existem basicamente três regras a serem observadas no tocante à eficiência:

- a eficiência é um requisito de desempenho e deve, portanto, ser contemplada desde a etapa de análise e definição dos requisitos;
- a eficiência pode ser melhorada com um bom projeto;
- a eficiência e a simplicidade do código devem ser requisitos bem equilibrados; em hipótese alguma deve-se sacrificar a clareza e a legibilidade do código em nome da eficiência.

A codificação pode ser encaminhada de modo a obter eficiência segundo diferentes visões, as quais serão discutidas a seguir.

6.1. Eficiência de Código

Este aspecto está relacionado diretamente à eficiência dos algoritmos que serão implementados nos componentes do software. Embora estes sejam especificados na fase de projeto detalhado, o estilo de codificação pode trazer uma contribuição importante, particularmente com a observação das seguintes regras:

- expressões aritméticas e lógicas devem ser simplificadas como uma tarefa que anteceda a codificação;
- malhas aninhadas devem ser cuidadosamente analisadas para verificar a possibilidade de moves instruções para fora delas;
- quando possível, privilegiar o uso de estruturas de dados simples, evitando estruturas do tipo vetores multidimensionais, ponteiros e listas complexas;
- adotar operações aritméticas rápidas;
- evitar a mistura de diferentes tipos de dados (mesmo que a linguagem utilizada permita isto);
- utilizar expressões booleanas e aritmética de números inteiros sempre que possível.

6.2. Eficiência de Memória

Este item refere-se ao fato do programa ocupar o menor espaço possível na memória da máquina. Embora no que se refere a grandes computadores e mesmo em computadores pessoais (em alguns casos) a eficiência de memória não seja uma questão fundamental, a preocupação é verdadeira no caso de sistemas dedicados.

Não existem propriamente regras explícitas para se atingir a eficiência de memória, mas o parâmetro fundamental para obter isto é a simplicidade de software. Além disso, algumas das regras seguidas para obtenção da eficiência de código podem resultar igualmente em eficiência de memória. O caso típico para isto é a utilização de estruturas de dados simples.

6.3. Eficiência de Entradas/Saídas

Este é outro aspecto importante a ser considerado no momento da codificação. As principais regras a serem seguidas para se atingir este parâmetro são:

- minimizar a quantidade de solicitações de E/S;
- fazer uso de buffers para reduzir o acúmulo de operações de comunicação;
- no caso de E/S para dispositivos secundários deve adotar uma organização dos dados em blocos e métodos de acesso simples;
- as E/S para terminais e impressoras devem levar em conta características dos dispositivos que melhorem sua velocidade e qualidade;
- a clareza da E/S deve prevalecer sobre as questões de eficiência.