SBA: Controle de Automação, Vol. 1, Nº 2, pp 99-109

AUTOMAÇÃO DO DESENVOLVIMENTO DE SOFTWARE
PARTE I: CONCEITOS FUNDAMENTAIS, FERRAMENTAS E AUTOMAÇÃO

Mario Jino
UNICAMP - FEE - DCA
Caixa Postal 6101 - Campinas, SP

Mario Bento de Carvalho
CTI - IA - DEI
Caixa Postal 6162 - Campinas, SP

Caetano Traina Júnior
ICMSC - USP
Caixa Postal 369 - São Carlos,SP

Resumo

Neste artigo apresenta-se um panorama das tendências atuais no desenvolvimento de ferramentas destinadas a automatizar o desenvolvimento de software. Na Parte I analisam-se os conceitos e problemas mais importantes desse desenvolvimento e apresenta-se uma motivação histórica do surgimento de ferramentas. Na Parte II descrevem-se sucintamente algumas abordagens e sistemas existentes que têm por objetivo aumentar o grau de automatização da produção de software.

Automation of Software Development
Part I: Fundamental Concepts, Tools and Automation.

Abstract

In this paper an overview of the current trends in the development of the tools intended to extend the automation of software development is presented. In Part I concepts and major problems related to software development are analysed and a historical perspective of the tools is presented. In Part II approaches and existent systems which aim the increase of the level of automation of software production are described briefly.

1. INTRODUÇÃO

A informática vem se tornando cada vez mais indispensável a um número cada vez maior de atividades humanas, apresentando soluções eficientes a necessidades antigas, e até mesmo possibilitando que novas atividades, antes inexequíveis, possam ser empreendidas. Por outro lado, a própria informática cria novas necessidades, algumas das quais ela própria deve solucionar.

Quando os primeiros computadores foram construídos, a sua limitada capacidade de memória e recursos permitia que a sua programação fosse feita sem que fosse necessário um empreendimento muito grande de recursos e pessoas, visto que esses programas eram necessariamente pequenos. No entanto, a tendência de se aumentar a capacidade dos computadores, dotando-os de mais memória e maiores recursos (por exemplo, memória virtual, proteção de páginas e periféricos, interrupção, etc), longe de tornar a programação mais fácil (como seria de se esperar à primeira vista), fez com que o esforço necessário à elaboração do software que aproveitasse esses recursos fosse cada vez maior. Além disso, com o aumento de sua capacidade e com o seu barateamento, os computadores passaram a ser exigidos com recursos de software muito mais sofisticados, o que também contribuiu para tornar o desenvolvimento de software cada vez mais laborioso.

A programação dos computadores, que nos primórdios de sua existência era em geral feita por pessoas que tinham formação de engenheiros, físicos, etc, passou gradualmente a requerer pessoal especializado, que conhecesse técnicas de programação sofisticadas, que pudesse analisar um problema a ser solucionado por computador e, de maneira metódica, conseguir usar o computador de forma consistente para se obter um resultado satisfatório. Além disso, a tarefa de se tratar de programas de computador, desde a sua concepção, projeto e implementação, até a sua manutenção e desativação, passou a requerer a participação de várias pessoas, às vezes até um número bastante grande delas. Assim, os problemas envolvidos em cada uma das várias fases por que passam os programas durante o seu desenvolvimento, e o próprio gerenciamento das ativida-

des de desenvolvimento, tiveram que ser estudados e dispor de soluções que as pessoas envolvidas nessas tarefas deveriam conhecer.

Dessa forma, surgiu uma nova disciplina, a **Engenharia de Software**, e um novo profissional que a exerce, o qual tem como razão de ser a de estabelecer métodos e indicar soluções para conceber, produzir e manter todo o software que existe nos computadores atuais. A percepção da necessidade de se organizar essa disciplina surgiu do reconhecimento de que o software é o produto de uma atividade que apresenta muitas características e necessidades semelhantes àquelas apresentadas pelas atividades das várias modalidades de engenharia tal como a engenharia civil, mecânica, elétrica, etc. Esse reconhecimento ocorreu pela primeira vez no fim da década de 60 e deu origem à tendência de sistematização das atividades de concepção e implementação de software, que hoje constituem a Engenharia de Software.

A Engenharia de Software apresenta-se em várias facetas distintas das modalidades tradicionais de engenharia. Talvez os dois pontos que apresentam maior diferença sejam a sua relativa juventude e a qualidade de ser impalpável de seu produto final, o software.

A juventude dessa disciplina manifesta-se sobretudo pela inexistência de metodologias, simbologia, normas, etc, que sejam consagradas. Em geral, o que ocorre é a existência de várias metodologias semelhantes, simbologias independentes, normas conflitantes, etc, cada uma com suas próprias vantagens e desvantagens em relação às outras, sem que tenha ainda surgido, em relação a muitas dessas facetas, uma solução universalmente aceita ou definitiva.

A qualidade do software de ser altamente impalpável tem um papel importante nesse estado de coisas, pois dificulta a obtenção de um padrão de representação, dado que não existe, em geral, nenhuma forma de se visualizar concretamente as facetas notadamente abstratas do software. Um exemplo típico pode ser dado com o gerenciamento da produção do software que, por não ter o que medir, torna o acompanhamento da evolução do produto extremamente difícil por qualquer método tradicional. Os conceitos tradicionais de controle de produção que, com poucas variações, podem ser aplicados às modalidades tradicionais de engenharia não se aplicam à Engenharia de Software, a qual necessita de métodos e recursos bastante diferentes.

Porém, como todas as modalidades de engenharia, a Engenharia de Software precisa de recursos específicos em termos de ferramentas, para que ela possa ser exercida. E, em sua grande maioria, essas ferramentas são ferramentas de software. Elas têm sido criadas até naturalmente, pela própria necessidade de sua existência. Editores, compiladores, sistemas de gerenciamento de arquivos, etc, nada mais são do que ferramentas que auxiliam a implementação do software.

E da mesma maneira como a informática vem permitindo a automação de muitas atividades de modalidades tradicionais de engenharia, tais como sistemas de automação de produção, auxílio computadorizado ao projeto, etc, também a informática vem se beneficiando da automação de muitas atividades da Engenharia de Software.

2. CONCEITOS FUNDAMENTAIS

A Engenharia de Software tem por objetivo estabelecer normas e fornecer recursos para que a tarefa de se usar computadores para auxiliar a resolução de problemas seja bem sucedida. Essa colocação: "usar computadores para auxiliar a resolução de problemas", apresenta vários pontos a serem considerados. O primeiro deles é que, para se usar computadores, eles devem ser programados. Então, para se usar computadores, é necessário que se escrevam programas de computadores. Porém, a mera existência de programas não significa que os computadores resolvam problemas; então, o segundo ponto a ser considerado é que os programas devem ser usados, e isso significa que alguém deve escolher qual programa será usado para resolver um problema e como esse programa será usado. O terceiro ponto é que, se o problema muda um pouco, e isso ocorre muito frequentemente, o programa já escrito pode ou não ser capaz de acompanhar essa mudança sem ser reescrito; isso significa que um programa pode ter sua utilidade limitada ou estendida (se for possível alterá-lo com facilidade). Se for muito difícil conseguir que o computador acompanhe as alterações de um problema, longe de ser auxílio à solução do problema, o computador torna-se o problema.

Dessa forma, é importante que se conceitue o que significa "**Software**". É comum que se considere software apenas os programas de computadores; porém, dentro do escopo da Engenharia de Software, essa palavra significa muito mais. Seguindo os três pontos que foram considerados no parágrafo anterior, em primeiro lugar é "Software" **programa de computador**, em suas várias formas: código executável da máquina, código fonte em uma linguagem de programação, módulos objeto em bibliotecas, etc. Em segundo lugar, é "Software" também a documentação de como esses programas podem ser usados, por exemplo, através de manuais de uso, cursos estruturados que ensinem o seu uso, etc. E em terceiro lugar, é "Software" também a documentação da própria estrutura de funcionamento desses programas, a explicação lógica de suas estruturas internas, os detalhes e decisões de projeto que levaram à sua concepção, etc.

Portanto, é "Software" um programa, seus manuais de uso, a explicação documentada de seu funcionamento, etc, e nenhuma dessas formas de software é mais ou menos importante do que as demais, visto que a inexistência de qualquer uma delas prejudica a própria razão de sua existência: permitir "usar computadores para auxiliar a resolução de proble-

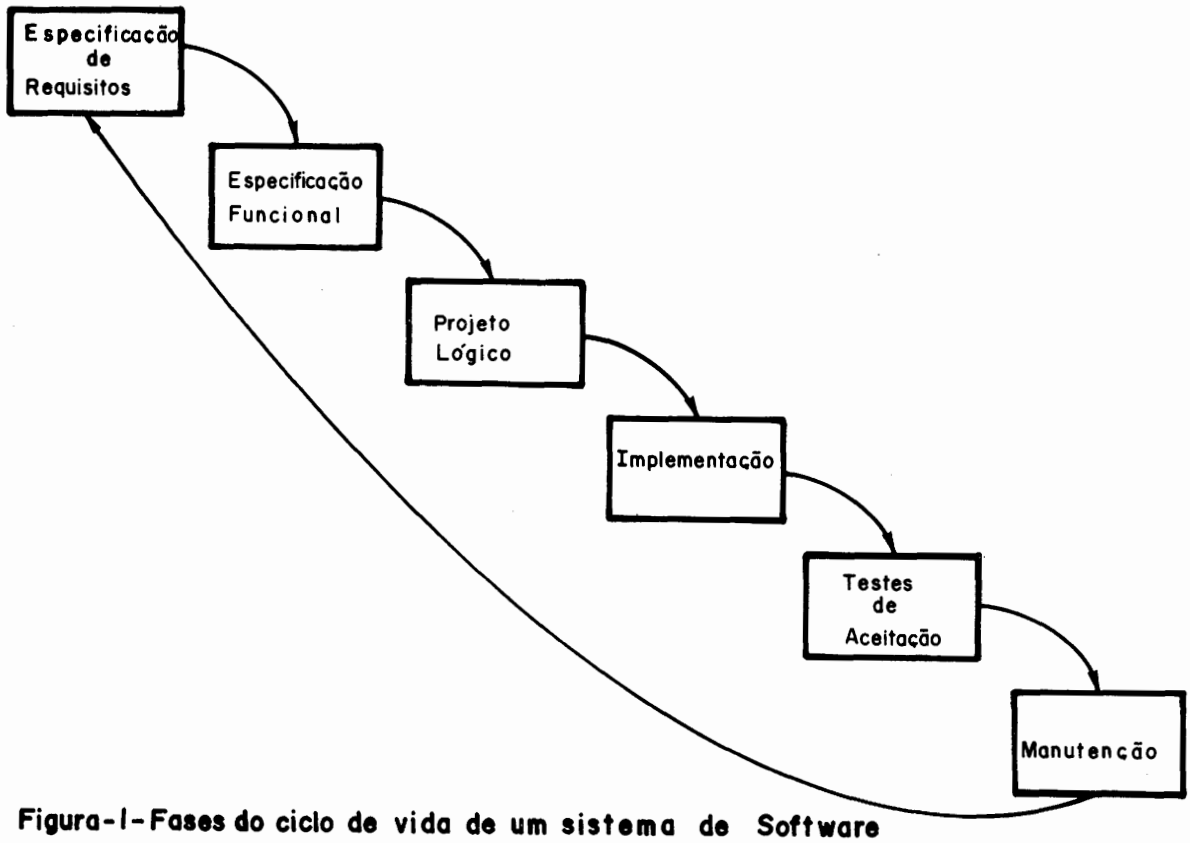

Figura-1-Fases do ciclo de vida de um sistema de Software

mas".

É claro que tudo isso tem um significado maior quanto maior for a complexidade do problema que se quer resolver. A idéia comum de se considerar software apenas programas chega a fazer sentido quando o "programa" é pequeno, foi escrito num final de tarde para resolver um problema simples. Mas quando uma equipe de profissionais tem que trabalhar muito tempo, de forma bem coordenada, para conseguir obter o "Software" necessário, então o "programa" é somente parte do produto. O "Software" que é o produto deste trabalho consiste de programas, manuais, documentação organogramas de cursos, etc. Nesses casos, é comum então que se denomine o produto de um **Sistema de Software.**

Ciclo de Vida de Software

O desenvolvimento de um sistema de software, tal como o de um produto de qualquer modalidade de engenharia, passa por várias fases, desde a primeira idéia que alguém tem da necessidade desse produto até a sua existência como um produto final. Além disso, se for um produto que satisfaz uma necessidade que existe de fato, então pode-se esperar que esse produto venha a sofrer modificações durante a sua existência, sendo que cada modificação passa aproximadamente pelas mesmas fases por que o produto como um todo deve passar, desde a percepção da necessidade dessa modificação até a existência do produto com a mudança incorporada. Assim, pode-se pensar que o produto passa por um ciclo, onde várias fases se sucedem uma à outra, iniciando-se uma vez com a concepção inicial do produto e somente terminando quando o produto eventualmente deixar de existir.

O reconhecimento da existência desse ciclo deu origem ao conceito de **Ciclo de Vida de um Sistema de Software,** o qual, uma vez reconhecido, serve como um referencial para todas as atividades da Engenharia de Software. Pode-se a grosso modo dizer que as fases que compõem esse ciclo são:

Especificação de Requisitos - consiste no reconhecimento de que a existência de um determinado recurso ou produto pode contribuir para satisfazer uma necessidade existente e elaboram-se especificações que descrevem como esse recurso ou produto deveria ser;

Especificação Funcional - é feita uma análise de quais funções o produto deverá ter, em termos de necessidades gerais, técnicas que poderiam ser empregadas, etc;

Projeto Lógico - as indicações das características desse produto, geradas na fase anterior, são detalhadamente estudadas, tendo-se em vista os recursos disponíveis, gerando descrições detalhadas de como sua implementação deve ser;

Implementação - baseado nas descrições obtidas na fase anterior, a tarefa de implementação do produto propriamente dito é encetada;

Teste de Aceitação - dispondo do produto recém-produzido, este deve ser testado para verificar se ele atende às necessidades originais e às várias especificações e descrições geradas nas fases anteriores;

Manutenção - consiste em se manter o produto disponível para atender às solicitações de uso, no ambiente que estiver disponível para a sua execução.

Na Figura 1 mostram-se fases ocorrendo em sequência, formando efetivamente um ciclo. No entanto, esse estado bem comportado nunca

ocorre realmente sendo que, ao final de cada fase, em geral tem que se voltar um pouco atrás e repetir alguma (ou algumas) fase anterior, antes de se prosseguir. Além disso, em geral um produto está em várias fases ao mesmo tempo, porque cada componente que o constitui evolui em velocidade diferente das demais, a distinção entre as fases não é claramente delimitada, etc, e se for tentado traçar a sequência de um projeto real, ao invés de um diagrama como o da Figura 1, o que se acaba obtendo é a representação de um novelo de lã embaraçado.

Metodologias e Ferramentas

A Engenharia de Software tem a obrigação de colocar uma ordem no caos e procura fazer isso estabelecendo normas para o desenvolvimento de software. Para isso, uma conceituação adequada dos termos empregados deve ser conhecida. Por exemplo, os termos "estratégia", "metodologia" e "método"; e também "técnica" e "ferramenta" devem ter seus significados bem compreendidos para que uma discussão geral sobre a Engenharia de Software possa ser feita. Segundo Davis (1982), **estratégia** é uma abordagem genérica com a qual se pretende atingir um objetivo. Um **método** é um procedimento ordenado ou sistemático para isso. Já uma **metodologia** é um conjunto de métodos e técnicas que permitem atingir um objetivo bem definido. Assim, o termo estratégia se refere a abordagens genéricas, enquanto métodos e metodologias são meios detalhados de executá-las.

Já o termo **técnica** pode ser entendido como a maneira de se executar as ações preconizadas nos métodos, e as **ferramentas** são recursos existentes que se destinam a possibilitar ou facilitar a execução dessas ações. Uma ferramenta pode ser automatizada ou manual. Por exemplo, um gabarito que facilite a elaboração de um fluxograma pode ser visto como uma ferramenta manual, que auxilia a execução daquela tarefa. Já um editor de fluxogramas em execução num computador, que respeite as regras de confecção de fluxogramas, pode ser visto como uma ferramenta automatizada para auxiliar a execução da mesma tarefa.

Existem metodologias que indicam como cada fase do ciclo de vida deve ser executada, e como deve ser a sequência de desenvolvimento de um projeto dentro de seu ciclo de vida. É muito frequente que as metodologias estabeleçam regras que ditem como as várias fases devem se encadear, por exemplo, permitindo que a volta possa ocorrer apenas a determinadas fases e, além disso, que essas voltas somente possam ser feitas ao final ou em pontos específicos de cada fase.

A maneira como cada metodologia estabelece os critérios para se considerar encerrada uma fase varia de metodologia para metodologia; em geral, baseia-se na existência de documentos ou "software" em geral, que deve estar pronto, e sobre os quais alguns tópicos pré-especificados ("check-list") devem estar satisfeitos, tal como a verificação de que determinada informação foi incluída em um relatório.

Gerenciamento de Projeto e de Configuração de Software

A existência de pontos de referência padronizados para a delimitação de fases dentro do ciclo de vida de um projeto é especialmente interessante por permitir que se estabeleçam métodos para o gerenciamento do desenvolvimento do projeto. O **Gerenciamento de Projeto** consiste em se estabelecer critérios para se alocar pessoal e recursos para as várias tarefas de desenvolvimento do projeto. Isto é feito através do acompanhamento do estado de desenvolvimento dos diversos ítens de software. Itens de software podem estar em desenvolvimento ou "congelados", indicando que não devem mais ser alterados; no entanto, ítens já congelados podem ser liberados para novas modificações, se a evolução do projeto assim o requerer.

A existência de pontos de referência padronizados entre as fases do ciclo de vida permite também que a configuração do software possa ser gerenciada, através da disciplina conhecida como **Gerenciamento de Configuração de Software** (Bersoff, 1984). Seu objetivo é: facilitar o gerenciamento de versões distintas do sistema, que possam vir a ser produzidas a partir de componentes comuns e que podem coexistir durante toda a vida do sistema; e permitir que a sua existência seja metodicamente controlada. Além disso, indica quando deve ser efetuada a verificação de que o produto de uma fase confere com as especificações e requisitos produzidos em fases anteriores, e permite que se mantenha um registro de todas as atividades exercidas durante o desenvolvimento do software, bem como a razão das decisões tomadas.

O Gerenciamento de Projeto e o Gerenciamento de Configuração de Software são duas atividades que devem ser exercidas durante o desenvolvimento de um sistema de software, atuando em todas as fases de seu ciclo de vida, mantendo-se, porém, as atividades do próprio desenvolvimento de software. Existem metodologias específicas para apoiar essas duas atividades que são, na medida do possível, independentes das metodologias que apóiam o desenvolvimento do software.

3. METODOLOGIAS E ABORDAGENS DE DESENVOLVIMENTO

A inexistência de metodologias universalmente aceitas, que permeia toda a Engenharia de Software, manifesta-se também com respeito ao conceito de ciclo de vida, não existindo um padrão consagrado para a sua representação (Yau e Tsai, 1986). Desta forma, também as fases do ciclo de vida reconhecidas em cada metodologia não são necessariamente aquelas descritas anteriormente. O que ocorre é que cada metodologia define a sua própria concepção do que constitui cada fase.

Por exemplo, pode-se considerar que o Projeto Detalhado de cada produto seja obtido através de uma fase de Projeto Lógico, onde não se levam em conta os recursos disponíveis, e outra fase de Projeto Físico, onde o projeto evolui considerando-se os recursos disponíveis. Cada metodologia em geral define a sua concepção de quais fases constituem o ciclo de vida, qual o resultado que deve ter sido obtido ao final de cada fase e como o produto pode evoluir percorrendo as várias fases.

Teoricamente poder-se-ia esperar a existência de uma metodologia que fosse capaz de indicar como obter um produto de software a partir do reconhecimento de sua necessidade, cobrindo todo o ciclo de vida desse produto. Embora algumas abordagens recentes pretendam atingir esse objetivo, não existe ainda disponível uma tal metodologia. O que ocorre é a existência de metodologias que enfocam uma ou duas fases do ciclo de vida (sendo o escopo de cada fase definido precisamente apenas dentro da própria metodologia). A grande maioria das metodologias procuram cobrir alguma (ou algumas) das fases de Especificação Funcional, Projeto Lógico e Implementação.

Especificação Funcional

Dentro do escopo da Especificação Funcional, duas estratégias têm sido adotadas, que correspondem às **metodologias orientadas por processos** e às **metodologias orientadas por dados** (Yau e Tsai, 1986).

As metodologias orientadas por processos enfatizam a decomposição e estruturação dos processos envolvidos em um sistema de software, aplicando sucessivamente a técnica de dividir um processo complicado em outros mais simples (técnica da **decomposição "top-down"**), ou então a técnica que emprega a composição de processos sucessivamente mais complexos a partir de outros mais simples (técnica da **composição "bottom-up"**). Exemplos de metodologias que adotam essa estratégia são a "Análise Estruturada" (Gane e Sarson, 1984), a metodologia "SADT" (Structured Analysis Design Technique) (Ross, 1977), a Metodologia de Projeto de Jackson (Jackson, 1975), a Metodologia de Warnier (Warnier, 1983), etc. Não cabe aqui a descrição dessas metodologias; a bibliografia indicada pode auxiliar o leitor interessado. Vale ressaltar que todas elas procuram atacar os mesmos problemas, que são: analisar quais funções o produto deverá ter; e fornecer a indicação de técnicas que poderiam ser empregadas.

As metodologias orientadas por dados enfatizam o projeto dos componentes de dados que constituem um sistema de software. As técnicas mais importantes dessa estratégia são as **técnicas de projeto orientadas por objetos**, e as **técnicas de projeto conceitual de bases de dados**. As primeiras procuram tratar os dados como se cada um por si só pudesse se modificar seguindo solicitações externas a ele. Por exemplo, poder-se-ia solicitar que os salários de todos os funcionários de uma empresa se elevassem de determinada porcentagem. As técnicas de projeto conceitual de bases de dados procuram orientar o projetista no processo de analisar as especificações de requisitos e, a partir deles, obter esquemas de estruturas que modelem os dados envolvidos no sistema de software.

Apesar de já existirem metodologias apoiadas na estratégia de projeto orientado por dados, como a SLAN-4 ("Software Language-4") (Beichter, Herzog e Petzsch, 1984) que emprega a técnica de projeto orientada por objetos, em geral as metodologias que usam esta estratégia não estão ainda tão bem estruturadas como as que existem para estratégias orientadas por processos. O tratamento matemático de métodos que exploram essa estratégia é no entanto mais fácil, feito através da formalização das estruturas de dados e das operações que os envolvem. Assim, existem métodos de especificação de "Tipos Abstratos de Dados" (Guttag, 1977; Liskov e Zilles, 1975), "Projeto de Objetos" (Booch, 1986), "Modelagem Conceitual de Dados" (Hammer e McLeod , 1977), "Ocultamento de Informações" (Parnas, 1972), etc, todos apoiados em conceitos e métodos de especificação formal.

Projeto Lógico

As metodologias que procuram cobrir a fase do projeto lógico do desenvolvimento de um sistema de software estão frequentemente associadas às metodologias que cuidam da fase anterior. Porém, continuam sendo metodologias independentes, que podem ou não ser utilizadas em conjunto com seus pares da fase de especificação funcional. Quase todas as técnicas empregadas nessa fase procuram de alguma forma usar recursos visuais para facilitar a compreensão de como os programas devem vir a ser implementados. A técnica que emprega o uso de fluxogramas é sem dúvida a mais antiga, tendo sido desenvolvida por Von Neumann com o intuito de servir como um meio de se documentar programas. Outras técnicas mais modernas têm sido propostas, tais como os Diagramas de Nassi-Shneiderman (Nassi e Shneiderman, 1973; Chapin, 1974), Português Estruturado, etc. Nessa área, já existem técnicas que procuram usar ferramentas computadorizadas para auxiliar o seu emprego, tais como o PDL ("Program Design Language") (Caine e Gordon, 1975) e o HOS ("Higher Order Software") (Hamilton e Zeldin, 1976).

Implementação

As metodologias que procuram cobrir a fase de implementação têm recebido maior divulgação, principalmente porque essa é a fase em que o resultado começa a tomar forma e, portanto, em projetos não muito extensos, costuma caracterizar a maior parte (se não a totalidade) do tempo de duração da produção de software. São razoavelmente bem conhecidas as metodologias de Programação Modular (Maynard, 1977) e Programação Estruturada (Wirth, 1978). O objetivo dessas metodologias é fornecer técnicas de programação (usando uma linguagem de programação) que facilitem o desenvolvimento e o entendimento de programas. As razões que

orientam cada uma dessas metodologias são, respectivamente, a maior facilidade de se produzir módulos de programas com uma função e conjunto de dados manipulados bem delimitados, e a noção de que o emprego organizado de estruturas de controle básicas facilitam a concepção, o entendimento e a manutenção de programas.

A grande maioria das ferramentas que utilizam computadores são destinadas a apoiar esta fase do ciclo de vida. Porém, como a sua necessidade foi sentida desde os primórdios da existência de computadores, o seu desenvolvimento ocorreu em geral antes que surgissem os conceitos de métodos e metodologias, que dessem uma orientação precisa para o seu emprego. Assim, editores de textos, compiladores, depuradores, etc, são ferramentas que apóiam a implementação de sistemas de software; porém, a sua forma tradicional não está associada a uma metodologia de implementação. Historicamente pode-se perceber que as próprias linguagens de programação acompanham o surgimento de metodologias. Por exemplo, linguagens antigas como COBOL e FORTRAN não refletem nenhuma metodologia, uma vez que não estavam estruturadas para isso. Com a consagração da programação estruturada, tornaram-se também difundidas as linguagens que as suportavam, tal como PASCAL e C. Atualmente, com a consecução de novas técnicas, novas linguagens vêm ganhando aceitação, tais como ADA (Ocultamento de Informação), SMALLTALK (Manipulação de Objetos), etc.

Contudo, além do surgimento de linguagens que incorporam recursos adequados para apoiar técnicas modernas de implementação, outras ferramentas têm se beneficiado desse desenvolvimento. Exemplos disso são os editores sensíveis à linguagem, depuradores de alto nível e geradores automáticos de telas. É interessante notar que o seu desenvolvimento é tão recente quanto o das ferramentas computadorizadas que procuram apoiar o projeto lógico de sistemas de software; porém, a maior popularidade das técnicas utilizadas para a implementação de programas fazem dessas ferramentas indubitáveis campeãs de uso e aceitação.

A consciência de que o processo de desenvolvimento de software não compreende apenas a codificação de instruções de programas tem feito surgir ferramentas que procuram manter à disposição do programador tudo o que ele possa necessitar durante a tarefa de implementar programas. Uma só ferramenta integra o editor com todos os recursos necessários, o compilador, o ligador de módulos, o gerenciador de biblioteca e o depurador, recebendo a designação genérica de **Ambientes de Programação**. Essas ferramentas vêm tornando-se rapidamente populares, principalmente com o advento das estações de trabalho apoiadas em microprocessadores e em Microcomputadores Pessoais.

A evolução dos ambientes de programação indica duas perspectivas que devem e vêm sendo estudadas para continuar o desenvolvimento de técnicas de apoio às tarefas de desenvolvimento de software. Uma delas corresponde à automação desse desenvolvimento, no sentido de que operações repetidas com frequência, onde o elemento "criatividade" não se manifesta, podem e devem ser realizadas pela própria máquina, retirando do programador o ônus de sua execução e, em alguns casos, até mesmo indicando ou sugerindo ao programador as alternativas de prosseguimento. Essa perspectiva será considerada em mais detalhe nas seções seguintes deste artigo.

A segunda constitui-se na constatação de que linguagens de alto nível podem ser compreendidas e manipuladas pelo próprio computador. Dessa forma, se linguagens de nível cada vez mais alto forem sendo desenvolvidas, então pode-se pensar que a colocação de um problema em linguagem coloquial poderá um dia ser diretamente compreendida e executada por computadores.

Transformação de Software

A expressão usada acima de que determinada linguagem é ou será "compreendida por computador" significa que existem **compiladores**, ou em última instância algum tipo de **tradutor**, que pode analisar documentos escritos nessa linguagem e traduzí-la para a linguagem interna que o computador executa. Encarado dessa forma, um compilador é uma ferramenta de transformação que, baseado em um documento de entrada escrito em determinada linguagem, produz um outro documento de saída que representa a mesma informação escrita em uma outra linguagem. Esse conceito não é diferente daquele existente em modalidades tradicionais de engenharia de que a aplicação de algum sinal ou objeto na entrada de um equipamento (que pode ser chamado de ferramenta) será manipulado por este e produzir-se-á em sua saída o sinal ou objeto modificado pela ação dessa ferramenta.

A especificação de requisitos que descrevem um novo sistema de software, ou descrevem uma mudança em um sistema já existente, é a informação de como se pretende que o produto seja, expressa em uma determinada linguagem, usualmente um documento escrito em linguagem coloquial com alguma estrutura e precisão. A tarefa da fase seguinte à obtenção desse documento, a de especificação funcional, consiste em transformar esse documento em outro contendo indicações mais precisas de como implementar o sistema requisitado. Na fase seguinte, a de projeto lógico, também faz-se a mesma coisa, refinando ainda mais essas indicações. Finalmente, tendo-se obtido as informações necessárias à implementação, esta passa a ser efetuada. Nesse momento uma nova transformação tem lugar, pois as informações indicando como a implementação será efetuada serão convertidas no programa executável pelo computador. Essa transformação em geral é a composição de duas outras transformações: a codificação das informações de implementação em linguagem de programação; e a compilação desse código para a obtenção do código executável.

A fase do ciclo de vida seguinte à de im

plementação pode também ser vista como consistindo de transformações e comparações, pois o produto final obtido pode ser transformado em especificações de requisitos, indicações de como ele foi emplementado, etc, e o resultado dessas transformações comparado com as informações originais correspondentes. Se o resultado da comparação for positivo, o produto é aprovado. De uma certa maneira, pode-se considerar que o processo de desenvolvimento de software consiste na **transformação** evolutiva dos requisitos iniciais do sistema, existindo métodos específicos que orientam o desenvolvimento de software apoiados nessa premissa (Balzer, 1981).

Como Automatizar o Desenvolvimento ?

Se tudo o que ocorre em cada uma das fases anteriores à implementação é a transformação de documentos em outros contendo a mesma informação cada vez mais detalhada, pode-se pensar que se as linguagens compreendidas pelo computador forem de nível cada vez mais alto, então poder-se-ia automatizar as atividades de todo o ciclo de vida. Considere-se, por exemplo, que num dado instante sejam dadas ao computador uma estrutura de processos e a descrição do que são esses processos, que é genericamente o conteúdo de um documento gerado por uma metodologia típica da fase de especificação funcional. Considere-se então que o próprio computador pudesse obter as indicações detalhadas de implementação que a fase seguinte (projeto lógico) deveria fornecer, gerando, por exemplo, um fluxograma da implementação. E que a seguir o próprio computador pudesse , a partir do fluxograma por ele mesmo gerado, realizar a codificação em linguagem de programação e, finalmente, produzir o programa executável. O que teríamos então seria o próprio computador gerando os programas a partir tão somente daquela descrição inicial de estruturas e processos.

Isso poderia ser feito efetivamente se apenas transformações estivessem envolvidas. Contudo, além do analista transformar as informações em cada fase, ele contribui aumentando a quantidade de informações que o documento de saída contém, adicionalmente às obtidas do documento de entrada. Isso porque , ao trabalhar em um problema, o analista coloca nele muito conhecimento que é, ou do próprio analista, ou obtido do cruzamento de seu conhecimento com as informações contidas no documento de entrada. Essas informações não podem ser obtidas por uma ferramenta totalmente automática e, assim, o sonho de se ter um computador que possa gerar um sistema de software a partir apenas das especificações desse problema não pode se concretizar.

Pode-se no entanto aumentar muito o nível de automação dessas atividades. Para isso, algumas questões chaves como as que se seguem devem ser respondidas: que tipo de informação um analista deve suprir para passar do início da fase de especificação funcional para o início da fase de projeto lógico? e desta para a de implementação? que tipo de representação dessas informações poderia transportar a maior quantidade possível de informação entre o computador e o analista?

Essas questões são hoje amplamente estudadas e os resultados de muitos trabalhos de pesquisa são frequentemente colocados em novas ferramentas, que auxiliam desta maneira não somente o desenvolvimento de sistemas de software como também, através da avaliação dessas ferramentas, o desenvolvimento da própria Engenharia de Software.

4. AUTOMAÇÃO DO DESENVOLVIMENTO DE SOFTWARE

Conforme foi visto na seção anterior, a automação do desenvolvimento de software é conseguida com a criação de ferramentas que auxiliem as tarefas que devem ser executadas para a produção de software. De um ponto de vista histórico, nota-se que as ferramentas mais antigas são destituídas de uma motivação metodológica, tendo sido criadas para suprir uma necessidade específica e sendo, quase invariavelmente, destinadas a auxiliar a fase de implementação dos programas.

Com o reconhecimento da existência do ciclo de vida dos sistemas de software, outras fases foram contempladas com ferramentas. Logo após esse reconhecimento, nos primeiros anos da década de 70, as primeiras metodologias foram concebidas, destinadas especificamente a determinadas fases do ciclo de vida. Algumas delas eram manuais, ou seja, estabeleciam métodos de trabalho que não necessitavam usar computadores; outras (mais raramente) eram automatizadas, ou seja, eram baseadas em métodos que usavam ferramentas computadorizadas.

Metodologias Automatizadas

Um exemplo de ferramenta automatizada que surgiu nessa época foi o sistema PSL/PSA ("Problem Specification Language/Problem Specification Analyser") (Teichroew e Hershey , 1977), desenvolvido na Universidade de Michigan. Esse sistema, com ampla aceitação e em uso até hoje, procura atingir a fase de Especificação Funcional e se apóia em uma metodologia desenvolvida especificamente para ele. A ferramenta e a metodologia estão tão intimamente ligadas que torna-se impossível separar uma da outra. Usualmente refere-se à metodologia como sendo representada pela linguagem, PSL, e à ferramenta como sendo representada pelo seu analisador, PSA. A metodologia procura identificar os objetos do mundo real e classificá-los segundo um conjunto de tipos reconhecidos pela linguagem PSL, e estabelecer entre esses objetos relacionamentos de tipos padrão também reconhecidos pela linguagem. A descrição de um sistema feita nessa linguagem pode ser submetida ao PSA, que armazena a descrição feita em uma base de dados e pode fornecer ao usuário um amplo conjunto de relatórios e análises sobre os dados armazenados.

A idéia geral por trás desse sistema - a de se reconhecer objetos do sistema e classi

ficá-los segundo um conjunto pré-determinado de tipos de objetos, e reconhecer entre eles um conjunto pré-determinado de relacionamentos - é universalmente aplicável a qualquer tipo de representação de informações, bastando para isso que se estabeleçam os tipos de objetos e de relacionamentos adequadamente . Assim, o PSL/PSA tem no seu conjunto de tipos de objetos aqueles que são significativos para a fase do ciclo de vida que ele se propõe apoiar. Sendo destinado à Especificação Funcional, os tipos de objetos são, por exemplo, PROCESS, INPUT (dados de entrada), OUTPUT (dados de saída), GROUP (Estruturas de Dados), etc. Substituindo-se os conjuntos de tipos de objetos e de relacionamentos, a linguagem que permite a descrição de sistemas passa a poder representar o sistema em outras fases do ciclo de vida. Por exemplo, se forem estabelecidos tipos de objetos como PROGRAMA, SUBROTINA, PARÂMETRO-DE-ENTRADA, etc, a linguagem passaria a se dedicar à fase de projeto lógico. A dificuldade maior para isso está no analisador dessa linguagem, pois as análises e relatórios gerados são construídos especificamente para uma determinada linguagem.

No entanto, apesar dessas diferenças, uma vez tendo sido implementado o PSL/PSA, a criação de outras ferramentas semelhantes, apoiadas em linguagens estruturalmente iguais (apenas com os conjuntos de tipos de objetos e de relacionamentos modificados), ficou muito facilitada. Seguindo o rastro do PSL/PSA surgiram várias outras metodologias e ferramentas, baseadas na mesma idéia de identificação de objetos e de relacionamentos entre eles, porém destinadas a aspectos diferentes da produção de software. Assim, surgiram várias metodologias/ferramentas, tais como: RSL/REVS ("Requirements Specification Language/Requirements Engineering and Validation System") (Davis e Vick, 1977), para a descrição de especificações; DAS ("Design Analysis System") (Willis, 1978), para a simulação de sistemas de controle; ESPRESO (Ludewig, 1983) e EPOS (Lauber, 1982), para a descrição de sistemas de tempo real; ISDT ("Interactive Software Development Tool") (Azuma e outros, 1981), para a descrição lógica de programas comerciais (especialmente para os escritos em COBOL); etc.

O maior desenvolvimento de técnicas e metodologias que, na fase de projeto funcional, adotam a estratégia de projeto orientado por processos reflete-se nas ferramentas que as automatizam, tal como é o caso das ferramentas/metodologias citadas acima. No entanto, ferramentas que apóiam a estratégia de projeto orientado por dados também têm sido desenvolvidas, tal como é o caso da ferramenta INCOD-DTE ("INteractive COnceptual Design of Data, Transactions and Events"), descrita em Atzeni e outros (1982), baseada no Modelo Entidade-Relacionamento, expandido para incluir abstrações (chamadas de Hierarquias) de agregação (hierarquias de relacionamento de subconjunto) e de generalização. A INCOD-DTE tem como propósito ser usada durante a fase de especificação funcional como uma formalização de transações envolvendo bases de dados independentes de implementações particulares; ao mesmo tempo, durante a fase de execução do sistema, pode ser usada como uma Linguagem de Manipulação de Dados de alto nível.

Ferramentas Integradas

Além de ferramentas/metodologias destinadas a apoiar o desenvolvimento do próprio software, surgiram ferramentas específicas para o apoio às tarefas de Gerenciamento de Projeto. A execução dessa tarefa normalmente é feita em conjunto com o desenvolvimento do software em si e, portanto, deve se adptar às metodologias de desenvolvimento de software empregadas. Qualquer metodologia que se empregue para o gerenciamento do projeto é independente daquelas que apóiam o desenvolvimento do software, uma vez que se destinam a objetivos diferentes e são normalmente desenvolvidas de maneira independente. O seu uso simultâneo em um projeto é conseguido porque as pessoas que as exercem são bastante flexíveis para identificar e aproveitar os pontos em comum que permitem a sua execução em cooperação.

Já o mesmo nao ocorre quando se procura uma automatização desse processo. Nesse caso, é comum que se empreguem duas ferramentas independentes e não integradas, cada uma cuidando de uma das atividades. Existem, no entanto, ferramentas/metodologias que procuram dar apoio tanto ao desenvolvimento do software quanto ao gerenciamento do projeto.

Esse é o caso do sistema EPOS - Engineer and Process-Oriented development support System, que possui uma metodologia intrínseca para o apoio ao desenvolvimento de sistemas em tempo real, ao mesmo tempo que apóia o gerenciamento de projetos. Para isso, possui uma linguagem de especificação de requisitos, EPOS-R, através da qual podem ser especificados os requisitos gerais do sistema, e uma linguagem de especificação detalhada, EPOS-S, onde as componentes de um sistema podem ser descritas, usando para isso a metodologia de desenvolvimento intrínseca ao próprio EPOS . O Gerenciamento de Projeto, bem como algumas funções do Gerenciamento de Configuração de Software são apoiados através de construções sintáticas que permitem interrelacionar os componentes dos vários detalhamentos do sistema, feitos através da EPOS-S, tanto entre si como com os requisitos gerais do projeto, efetuados com a EPOS-R.

Para automatizar a metodologia espelhada por essas linguagens, o EPOS possui três ferramentas: EPOS-A, que permite analisar as descrições feitas nessas linguagens e armazená-las em uma base de dados interna; EPOS-D, que permite gerar relatórios em vários formatos sobre as informações armazenadas (diagrama de blocos, diagramas hierárquicos, diagramas de estruturas de dados, redes de Petri, diagramas de Nassi-Shneiderman, etc); e EPOS-C, que permite o acesso às demais ferramentas.

O sistema EPOS é um bom exemplo de uma ferramenta que integra o desenvolvimento do

software através de várias fases, uma vez que possibilita ao usuário a documentação de seu sistema desde as fases iniciais de especificação até a fase de projeto lógico, além de ter associada uma metodologia de apoio ao gerenciamento de projeto.

É interessante ressaltar as principais vantagens que existem em se dispor de um ferramental integrado para o desenvolvimento de um projeto:

- rastreabilidade, pois permite associar ítens de software pertencentes a representações de fases diferentes do ciclo de vida;
- não redundância de informações, pois cada ítem de software possui uma única representação interna;
- uniformidade de operações e de uso pelo usuário, através de todas as fases suportadas.

A capacidade de apoiar o desenvolvimento através de um ferramental que integre várias fases do ciclo de vida (ou mesmo idealmente todo o ciclo), pode ser conseguida, ou através da concepção de uma metodologia que seja capaz de abranger todas essas fases, ou então integrando metodologias já existentes, de maneira que o conjunto integrado cubra todas as fases desejadas. A integração de metodologias já existentes, desenvolvidas independentemente umas das outras, nem sempre pode ser feita diretamente, sendo em geral necessário fazer alterações para que a integração seja possível ou melhorada.

Tem sido mais comum o desenvolvimento de ferramentas integradas que se apóiem em metodologias especialmente desenvolvidas, tal como é o caso dos sistemas SARA (Campos e Estrin, 1977), DRACO (Neighbors, 1984), Aspect (Earl e Whittington, 1985), etc, principalmente devido à possibilidade de que a ferramenta automática tenha a sua especificação ditada pela metodologia especialmente criada, o que facilita o seu desenvolvimento. O problema associado a esse enfoque é que, por necessitar cobrir um espectro muito amplo de situações, em geral determinados aspectos do desenvolvimento são fracamente suportados.

Meta-Sistemas

Com a proliferação de ferramentas automatizadas para o desenvolvimento de software, começou a se perceber o quanto existia de comum nos vários enfoques em termos de ferramentas para apoio computadorizado, e também que os núcleos dos vários sistemas implementados tinham muitas partes semelhantes. Assim, passou-se a considerar a possibilidade de se construir ferramentas mais flexíveis, que pudessem aceitar a definição da linguagem de descrição de sistemas, centralizada na definição dos conjuntos de tipos de objetos de projeto, e dos tipos de relacionamentos que existem entre eles.

As novas ferramentas construídas segundo esse conceito são destituídas de uma metodologia associada, uma vez que esta fica atrelada à linguagem de descrição e não à ferramenta em si. Esses sistemas passaram a ser conhecidos com a designação de **Meta-Sistemas**, uma vez que, além da descrição dos sistemas, eles são capazes de receber a própria linguagem de descrição de sistemas. Desta forma, uma única ferramenta pode atender a várias linguagens de descrição de sistemas.

Exemplos de Meta-Sistemas são: SEMS ("System Encyclopedia Management System") (Demetrovics, Knuth e Radó, 1982), desenvolvido dentro do mesmo projeto que produziu o PSL/PSA, como seu sucessor Meta-Sistema direto e com o qual é compatível; SDS ("Software Development System") (Levene e Mullery, 1982), que permite a classificação dos tipos de objetos em categorias pré-determinadas, que facilitam a elaboração de análises e relatórios; SDLA ("System Descriptor and Logical Analyser") (Knuth, Halász e Radó, 1982), cuja principal diferença em relação aos demais Meta-Sistemas é a de admitir a especificação de Subtipos de objetos que define uma linguagem; etc. As diferenças entre os vários Meta-Sistemas recaem principalmente no tipo de análise sintática que pode ser feita nas linguagens definidas e nos relatórios que podem ser produzidos por eles.

Note-se que os relatórios ou análises produzidos por um Meta-Sistema são relativamente genéricos, visto que o gerador de relatórios deve ser construído a partir da especificação de como uma linguagem pode ser e não de como ela é, pois durante a construção da ferramenta a linguagem não é conhecida. Isso leva à necessidade de se conseguir estender um meta-sistema incorporando-lhe novos recursos para a geração e análise de outros tipos de relatórios, específicos para determinadas linguagens que lhes são associadas. Assim, pode-se ter Meta-Sistemas que forneçam relatórios genéricos para as linguagens que são livremente criadas; porém, para uma determinada linguagem pré-definida e imutável, a ferramenta pode incluir relatórios específicos. Um exemplo disso é o SREM ("Software Requirements Engineering Methodology") (Alford,1977) que, embora possa aceitar a definição de linguagem de descrição do usuário, tem uma capacidade de análise muito maior se for usado com a linguagem RSL (a mesma do sistema RSL/REVS).

A maior flexibilidade dos Meta-Sistemas é, dessa forma, um tanto limitada, pois a impossibilidade de se gerar os relatórios adequadamente para uma linguagem definida pelo usuário faz que, por exemplo, esses sistemas não se prestem para a automatização de metodologias criadas independentemente deles. Por exemplo, um Meta-Sistema não pode ser usado para automatizar uma metodologia manual já existente, a menos que esta seja modificada para se adaptar às características do Meta-Sistema. O que ocorre quando isso é tentado, é que as mudanças são tão profundas que descaracterizam a metodologia original e o meta-sistema não é bem aceito pelas pessoas que estavam habituadas com o uso da metodologia manual, provocando a rejeição da automatização.

Flexibilidade e Integração

Um outro problema que ainda persiste com o uso de Meta-Sistemas é que, se duas linguagens sao desenvolvidas e colocadas em um mesmo Meta-Sistema, as descrições efetuadas em uma das linguagens não podem ser confrontadas com as feitas na outra, pois não existe um meio de se integrar as duas. Assim, um problema que existe nas ferramentas que não se apóiam em Meta-Sistemas, que é a total não integração das descrições mantidas em cada sistema, permanece nos Meta-Sistemas.

Essa integração é altamente desejável, e começam a ser exploradas várias alternativas para se conseguí-la. A sua necessidade fundamenta-se em vários motivos que serão descritos na segunda parte deste artigo; porém, pode-se perceber a priori que, em termos de automatização da produção de software, não tem sentido lógico um analista fazer a especificação de um sistema usando um sistema de software cujo resultado não possa ser usado por esse mesmo sistema para continuar o trabalho na fase seguinte. E isso é o que ocorre hoje, não somente obrigando o analista a repassar a mesma informação novamente para o computador (agora usando outro sistema de apoio ao projeto), como também tendo que mudar a forma de representação da informação, uma vez que os sistemas de apoio ao projeto são frequentemente incompatíveis até mesmo quanto à representação da informação.

Os sistemas integrados para produção de software são a resposta que o desenvolvimento da informática pretende dar para a automatização da produção de software; contudo, esses sistemas ainda estão, hoje, em fase de desenvolvimento. Na segunda parte deste artigo serão apresentadas novas abordagens que podem permitir a sua disponibilidade, e os esforços que estão sendo desenvolvidos em vários países com o objetivo de se conseguir trazê-los para a realidade.

5. REFERÊNCIAS BIBLIOGRÁFICAS

Alford, M. W. (1977). "A Requirements Engineering Methodology for Real-Time Processing Requirements". IEEE Trans. on Software Engineering, Vol. SE-3, Nº 1: 60-69.

Atzeni, P.; Batini, C.; De Antonellis, V.; Lenzerini, M.; Villanelli, F.& Zonta, B. (1982). "A Computer Aided Tool for Conceptual Data Base Design". Automated Tools for Information Systems Design. H.J. Schneider & A.I. Wasserman (Eds.), North-Holland Publishing Company, pp.85-107.

Azuma, M.; Takahashi, M.; Kamya, S.& Minomura, K. (1981). "Interactive Software Development Tool: ISDT". Proc. 5th International Conf. on Software Engineering, San Diego, California, pp.153-163.

Balzer, R. (1981). "Transformational Implementation: An Example". IEEE Trans. on Software Engineering, Vol. SE-7, Nº 1: 3-14.

Beichter, F.W.; Herzog, O.; Petzsch, H.(1984). "SLAN-4 - A Software Specification and Design Language". IEEE Trans. on Software Engineering, Vol.SE-12, Nº 2: 155-161.

Bersoff, E. (1984). "Elements of Software Configuration Management". IEEE Trans.on Software Engineering, Vol.SE-10, Nº 1: 79-87.

Booch, G. (1986). "Object-Oriented Development". IEEE Trans. on Software Engineering, Vol.SE-12, Nº 2: 211-221.

Caine, S.H.& Gordon, E.K. (1980). "PDL - A Tool for Software Design". Tutorial on Software Design Techniques, 3rd Edition, P.Freeman & A.I. Wasserman (Eds.), IEEE Computer Society Press, pp. 380-385.

Campos, I.M. & Estrin, G. (1977). "Concurrent Software System Design Supported by SARA at the Age of One". Proc. of the 3rd Int. Conf. on Software Engineering, pp. 230-242.

Chapin, N. (1974). "New Format for Flowcharts". Software Practice and Experience, Vol.4: 341-357.

Davis, C.G. & Vick, C.R. (1977). "The Software Development System". IEEE Trans. on Software Engineering, Vol.SE-3, Nº 1: 69-84.

Demetrovics, J.; Knuth, E.& Radó, P. (1982). "Specification Meta Systems". IEEE Computer, Vol. 15, Nº 5: 29-35.

Davis, G.B. (1982). "Strategies for Information Requirements Determination". IBM Systems Journal, Vol. 21, Nº 1.

Earl, A.N. & Whittingtom, R.P. (1985). "Capturing the Semantics of an IPSE Database". Data Processing, Vol. 27, Nº 9: 33-43.

Gane, C. & Sarson, T. (1984). Análise Estruturada de Sistemas. Rio de Janeiro, Livros Técnicos e Científicos Editora S.A..

Guttag, J. J. (1977). "Abstract Data Types and the Development of Data Structures". Commun, ACM, Vol. 20, Nº 6: 396-404.

Hamilton, M. & Zeldin, S. (1976). "Higher Order Software - A Methodology for Defining Software". IEEE Trans. on Software Engineering, Vol.SE-2, Nº 1: 9-32.

Hammer, M.& McLeod, D. (1981). "Database Description with SDM: A Semantic Database Model". ACM Trans. on Database Systems , Vol. 6, Nº 3: 351-386.

Jackson, M. A. (1975). Principles of Program Design. New York, Academic Press.

Knuth, E.; Halász, F. & Radó, P. (1982). "SDLA - System Descriptor and Logical Analyser". Information Systems Design Methodologies: A Comparative Review, T.W. Olle, H.G. Sol & A. A. Verrijn-Stuart (Eds.), North-Holland Publishing Company, pp. 143-171.

Lauber, R.J. (1982). "Development Support Sys

tems". IEEE Computer, Vol. 15, Nº 5 :36-46.

Levene, A. A. & Mullery, G.P. (1982)."An Investigation of Requirement Specification Languages: Theory and Practice". IEEE Computer, Vol. 15, Nº 5: 50-59.

Ludewig, J. (1983). "ESPRESO - A System for Process Control Software Specification". IEEE Trans. on Software Engineering, Vol. SE-9, Nº 4: 427-436.

Liskov, B.H. & Zilles, S.N. (1975). "Specification Techniques for Data Abstraction". IEEE Trans. on Software Engineering, Vol.SE-1: 7-19.

Maynard, J. (1977). Programação Modular. Rio de Janeiro, Livros Técnicos e Científicos Editora S.A..

Nassi, I.& Shneiderman, B. (1973). "Flowchart Techniques for Structured Programming". ACM SIGPLAN Notices.

Neighbors, J.M. (1984). "The DRACO Approach to Constructing Software from Reusable Components". IEEE Trans. on Software Engineering, Vol.SE-10, Nº 5: 564-574.

Parnas, D.L. (1972). "On the Criteria to Be Used in Decomposing Systems into Modules". Comm. of the ACM, Vol. 15, Nº 12: 1053-1058.

Ross, D. T.& Schoman Jr., K.E. (1977). "Structured Analysis for Requirements Definition". IEEE Trans. on Software Engineering, Vol.SE-3, Nº 1: 6-15.

Ross, D.T. (1977). "Structured Analysis (SA): A Language for Communicating Ideas". IEEE Trans. on Software Engineering, Vol. SE-3, Nº 1: 16-34.

Teichroew, D. & Hershey III, E.A. (1977). "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems". IEEE Trans. on Software Engineering, Vol.SE-3, Nº 1: 41-48.

Warnier, J.-D. (1983). LCP: Lógica de Construção de Programas: Um Método de Programação Estruturada, 2a. Ed., Rio de Janeiro, Ed. Campus Ltda.

Willis, R.R. (1978). "DAS - An Automated System to Support Design Analysis". Proc. 3rd Int. Conf. on Software Engineering.

Wirth, N. (1978). Programação Sistemática. Rio de Janeiro, Ed. Campus Ltda.

Yau, S.S. & Tsai, J. J.-P. (1986)- "A Survey of Software Design Techniques". IEEE Trans. on Software Engineering, Vol.SE-12, Nº 6: 713-721.