UNIVERSIDADE DE SÃO PAULO

ESCOLA POLITÉCNICA

Marilza Antunes de Lemos

Uma Abordagem Baseada em Padrões Elementares para Aprendizado de Programação

São Paulo

2004

Marilza Antunes de Lemos

Uma Abordagem Baseada em Padrões Elementares para Aprendizado de Programação

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Doutor em Engenharia.

Área de Concentração:
Sistemas Eletrônicos

Orientadora:
Profa. Dra. Roseli de Deus Lopes

São Paulo

2004

Aos meus filhos Rodrigo e Alex

Agradecimentos

Agradeço a Deus pela oportunidade de aprendizado. Quanto mais nos aprofundamos no conhecimento mais temos a consciência de que somos um grão de areia no universo infinito do conhecimento. E assim, o aprendizado é uma lição para o cultivo da humildade.

Agradeço a minha orientadora, Profa. Roseli, em primeiro lugar, pela confiança no meu trabalho e pelo apoio irrestrito em todos os aspectos que envolveram esta tese.

À profa. Leliane, agradeço pela dedicação em longas discussões sobre padrões de programação, planejamento e outros aspectos da área de Inteligência Artificial, sempre procurando me mostrar, o caminho acadêmico, a busca pela qualidade apurada na realização dos trabalhos.

Aos meus alunos de iniciação científica, André, Diego, Maria Cara e Ramão, o meu eterno agradecimento por tornarem as minhas idéias concretas através do esforço, a oito mãos, na implementação dos sistemas.

Agradeço aos meus alunos da Faculdade de Engenharia de Sorocaba, em especial, as turmas da Engenharia Elétrica de 2003, que possibilitaram a aplicação das propostas desta tese.

Finalmente, agradeço a duas pessoas muito especiais, minha mãe e tia Olga, que em qualquer momento de minha vida, não somente estão disponíveis, mas acima de tudo são capazes de me fazer feliz.

Resumo

A pesquisa em Ciência Cognitiva mostra que programadores experientes possuem a habilidade em resolver problemas, identificando metas e criando planos mentais de programação. Por outro lado, pesquisadores concordam que a maior dificuldade enfrentada por estudantes no aprendizado de programação é a de construir planos mentais de programação. Aprendizes, dispondo de seu conhecimento prévio e da linguagem de programação para construírem programas, são capazes de criar apenas soluções naturais, próprias para execução fora do contexto do computador. Tais soluções diferem de planos de programação por não considerarem aspectos computacionais. Assim, os problemas mais proeminentes no aprendizado de programação parecem estar situados ao longo de dimensões cognitivas. Como uma proposta para minimizar o problema apresenta-se, nesta tese, um Modelo de Processo de Construção de Programas para Aprendizes de Programação. O modelo é proposto como uma ferramenta cognitiva, definida na literatura como aquela que auxilia pessoas a executarem atividades mentais que não podem ser observadas diretamente ou que podem ser pouco observáveis, tais como ajudar a pensar, conhecer ou aprender. Para representação do modelo é definida a linguagem LPC (Linguagem de Programação Cognitiva), baseada em planejamento hierárquico da Inteligência Artificial. A LPC descreve componentes e estratégias de programação que, manipulados pelo aprendiz, visam promover a construção de conhecimento cognitivo em programação. Os elementos-chave da linguagem LPC são os planos de programação que, combinados, podem representar soluções para problemas de programação. Planos de programação são modelados a partir de padrões elementares de programação da área de Padrões Pedagógicos (PP, 2001). O modelo foi implementado por meio de dois sistemas: (i) a BibPC, Biblioteca para Programação Cognitiva, a qual é capaz de capturar conhecimento de planejamento em programação de forma reutilizável para uso e expansão por educadores; e (ii) o TutorC, um ambiente para aprendizagem de programação em linguagem C, o qual disponibiliza ao aprendiz, componentes de planejamento e de implementação de programas. O modelo proposto foi aplicado em sala de aula e laboratório para um grupo de alunos criteriosamente selecionado. Esta aplicação teve como objetivo realizar uma avaliação preliminar do potencial do Modelo de Processo de Construção de Programas proposto para aprendizes. Análises estatísticas por meio de histogramas e gráficos de distribuição normal das notas obtidas, antes e depois da aplicação, mostraram um aumento no desempenho dos estudantes.

Palavras-chave: Aprendizagem, Sistema Tutor Inteligente, Padrões de Programação

Abstract

The research in Cognitive Science shows that experienced programmers have ability in solving problems, identifying goals and creating programming mental plans. On the other hand, researchers agree the most difficulty faced for students in programming learning is to construct programming mental plans. Apprentices, making use of its background knowledge and the programming language to construct programs, are capable to create only natural and proper solutions for execution outside of computer context. Such solutions differ from programming plans for not considering computational aspects. So, the most prominent problems in programming learning seemed to fall along the cognitive dimensions. As a proposal to overcome this problem it is presented, in this thesis, a Process Model of Program Construction for Apprentices. The model is proposed as a cognitive tool, defined in literature as one that assists people to execute mental activities that can not directly be observed or that they can little be observed, such as to help to think, to know or to learn. A language based in Hierarchical Planning of Artificial Intelligence, the Language of Cognitive Program (LPC), is specified to represent the Model. The LPC describes components and strategies of programming for manipulation by the apprentice and thus, to contribute for construction of programming cognitive knowledge. The elements-key of LPC language are the programming plans that arranged, can represent solutions for programming problems. Programming plans are modeled from elementary programming patterns of the Pedagogical Patterns research area. The model was implemented by means of two systems: (i) the BibPC, Library for Cognitive Programming, which is be able to capture reusable planning knowledge of programming for use and expansion by educators; e (ii) the TutorC, an environment for C programming learning, which supplies programming components for apprentices plan and implement programs. The proposed model was applied in classroom and laboratory for a selected group. This application became possible a preliminary evaluation of the Process Model of Program Construction for Apprentices. Statistical analyses by means of histograms and normal distribution graphs of tests notes, before and after application, had shown increase in the student's performance.

Keywords: Programming Learning, Intelligent Tutoring Systems, Programming Patterns

Lista de Figuras

Lista de Problemas

Lista de Programas

Lista de Tabelas

Lista de Abreviaturas e Siglas

ACT-R	*Adaptive Control of Thought-Rational* Controle Adaptativo de Pensamento Racional
AI-ED	*Artificial Inteligence in Education* Inteligência Artificial em Educação
BibPC	Biblioteca para Programação Cognitiva
CAI	*Computer-Aided Instruction* Instrução Auxiliada por Computador
CTA	*Cognitive Task Analysis* Análise de Tarefas Cognitivas
FACENS.....	Faculdade de Engenharia de Sorocaba
HCI	*Human Computer Interaction* Interação Homem-Máquina
ILE	*Intelligent Learning Environment* Ambiente de Aprendizagem Inteligente
ITS	*Intelligent Tutoring System* Sistema Tutor Inteligente
LPC	Linguagem de Programação Cognitiva
MEC	Ministério de Educação
OOPSLA	*Object-Oriented Programming, Systems, Languages and Applications* Programação, Sistemas, Linguagens e Aplicações Orientadas a Objetos
OOT	*Object Oriented Technology* Tecnologia Orientada a Objetos
PLOP	*Pattern Languages of Programs* Linguagens de Padrão de Programas
PPIG	*Programming Psicology Interest Group* Grupo de Interesse da Psicologia da Programação
PPP	*Pedagogical Patterns Project* Projeto de Padrões Pedagógicos
STI	Sistema Tutor Inteligente
TutorC	Tutor para Aprendizagem de Programação em C

Lista de Símbolos

$\wedge$ - conjunção

$\neg$ - negação

$\in$ - pertinência

$\prec$ - ordenação

Notação

Os capítulos desta tese são numerados seqüencialmente a partir do 1. Dentro de cada capítulo, são definidas seções que recebem uma numeração composta do número do capítulo seguido de um número seqüencial a partir do 1. Quando houver referência à alguma seção, por exemplo 3.2, fica subentendido tratar-se do capítulo 3, parte 2.

Algumas partes do texto da tese recebem formatação diferenciada: texto em *itálico* indica palavras e expressões em língua estrangeira. Texto *itálico* em **negrito** indicam procedimentos (tarefas) em algoritmos. Texto em **negrito** indica ênfase e texto em *"itálico"* entre aspas indica citação de outros autores. As primeiras letras em maiúsculas no meio de uma frase podem indicar: área de pesquisa, siglas ou órgãos. Exemplos de problemas e soluções (programas) estão alocados em caixas de texto cinza.

Sumário

1 INTRODUÇÃO

As dificuldades que um estudante encontra no aprendizado de linguagens e lógica de programação têm sido relatadas em diferentes épocas e em diferentes áreas de pesquisa, tais como a da Educação, Inteligência Artificial, Ciência Cognitiva e Psicologia da Programação.

Da intersecção das áreas de Inteligência Artificial e Educação surgiu a área de Inteligência Artificial aplicada à Educação (AI-ED) que propõe o desenvolvimento de Sistemas Tutores Inteligentes (STI) (SHUTE;PSOTKA, 1994) para guiar o estudante, com o auxílio do computador, no aprendizado de vários domínios do conhecimento, dentre eles, a programação de computadores.

A Ciência Cognitiva, em particular a área de Psicologia da Programação divulgada pelo Grupo de Interesse da Psicologia da Programação (PPIG) (PPIG,1987), preocupa-se com o desenvolvimento de teorias sobre como programadores experientes e como aprendizes compreendem e constróem programas. Estudos empíricos têm resultado em propostas de modelos mentais de programadores e de como diferenciar programadores experientes de aprendizes (VON MAYRHAUSER;VANS,1994).

A pesquisa sobre Padrões de Programação (PP,2001), inclui um grupo de educadores da Ciência da Computação que tem documentado os chamados **padrões elementares de programação** (PEP,2001) para ensinar programação orientada a objetos e outros paradigmas, tais como programação procedimental e funcional. Wallingford (1998) afirma que padrões elementares de programação são apropriados para aprendizes, uma vez que buscam capturar o conhecimento de programação e de projeto no baixo-nível (código). Padrões elementares de programação, em geral, correspondem ao tipo de conhecimento que programadores experientes possuem e usam de forma subconsciente (WALLINGFORD,2002).

Esta tese integra algumas idéias e iniciativas dessas diferentes áreas de pesquisa para propor um trabalho que visa minimizar alguns dos principais e reconhecidos problemas enfrentados por programadores novatos.

1.1 Desafios do Aprendiz de Programação

> *"Os alunos aprendem a contar usando os dedos da mão. Ensinar computação deve partir de um modelo de computação abstrato ou de um modelo mais real?"* (CEEINF,1999)

Um primeiro passo no sentido de se compreender o processo de aprendizagem em programação é identificar as habilidades e competências necessárias para execução da atividade de programação, bem como as dificuldades tipicamente encontradas, dentro de um curso introdutório. Segundo Weber et al. (1996), aprender a programar se constitui numa atividade complexa que inclui:

- adquirir habilidade em resolução de problemas, que envolve entender o problema, identificar suas metas e construir planos de programação que alcancem essas metas;
- aprender a sintaxe de uma linguagem de programação;
- aprender sobre lógica de programação, ou seja, entender o comportamento do computador na execução de um programa, o que inclui compreender as semânticas da linguagem de programação;
- utilizar um ambiente de programação;
- transladar planos de programação em algoritmos executáveis;
- realizar testes e depuração de programas.

Cada uma dessas atividades envolve um certo grau de dificuldade. Porém, há um consenso de que a maior dificuldade nos cursos introdutórios de programação se encontra no desenvolvimento da habilidade em resolução de problemas dentro do contexto do computador. Ou seja, ainda que um aluno saiba resolver muito bem um problema, como por exemplo, uma equação de segundo grau, pode encontrar dificuldade em resolvê-lo no computador.

O Ministério da Educação (MEC), em seu documento de Diretrizes Curriculares (CEEINF,1999), caracteriza a resolução de problema como a principal atividade na programação de computadores. Esse documento define a programação como:

> A programação, entendida como programação de computadores, é uma atividade voltada à solução de problemas. Nesse sentido ela está relacionada com uma variada gama de outras atividades como especificação, projeto, validação, modelagem e estruturação de programas e dados, utilizando-se das linguagens de programação propriamente ditas, como ferramentas. Portanto, o estudo de programação não se restringe ao estudo de linguagens de programação. As linguagens de programação constituem-se em uma ferramenta de concretização de software, que representa o resultado da aplicação de uma série de conhecimentos que transformam a especificação da solução de um problema em um programa de computador que efetivamente resolve aquele problema. (CEEINF,1999).

Assim, pode-se dizer que aprender uma linguagem de programação é considerada apenas uma pequena parte do aprendizado global da programação. Um fato comum é o de professores concluírem que um programador com alguma experiência numa linguagem de programação aprende rapidamente uma nova linguagem de mesmo paradigma, enquanto que a maioria dos aprendizes encontram muita dificuldade nesse aprendizado. Há uma forte indicação de que o conhecimento anterior sobre resolução de problemas dentro do contexto da computação é um pré-requisito importante no aprendizado de linguagens de programação. Ao aprendermos uma segunda linguagem, considerando-se o mesmo paradigma, o que muda é a sintaxe da nova linguagem e o ambiente de programação (PEGG,2003).

Estudos empíricos da Psicologia da Programação realizados nos anos 80 por Soloway e Ehrlich (1984) e Bonar e Soloway (1983) mostraram que programadores experientes possuem a habilidade de resolver problemas, identificando **metas do problema** e criando **planos de programação** (ou planos mentais de programação), para somente depois, traduzí-los num algoritmo executável usando alguma linguagem de programação. Metas e planos de programação estão, de alguma forma, ligados às características computacionais, sejam do próprio computador ou específicas da linguagem de programação. Por outro lado, aprendizes de uma primeira linguagem de programação, dispõem apenas de seu conhecimento prévio e da sintaxe da linguagem para construírem seus programas. Usando o conhecimento prévio, aprendizes são capazes de criar apenas **planos naturais**, os quais diferem de **planos de programação** por não considerarem aspectos computacionais. Um plano natural é uma solução correta para um dado problema, porém não necessariamente executável no computador.

Pesquisas mostram que planos naturais podem ser aproveitados se forem corretamente mapeados para planos de programação. Como essas relações não são claras para o aprendiz, apesar de planos naturais satisfazerem as metas do problema, eles podem criar confusões. É neste contexto que surgem as principais dificuldades no aprendizado de programação: o aprendiz acaba por inserir erros em seu programa ao usar somente seu conhecimento prévio (BONAR;SOLOWAY,1983).

A habilidade em mapear planos naturais para planos de programação não é uma aquisição de conhecimento trivial. Sua complexidade pode variar de acordo com o paradigma de programação empregado.

É reconhecido por professores que, mesmo entre programadores experientes, a principal dificuldade encontrada no aprendizado de um novo paradigma de programação reside em mapear as metas em planos de programação neste novo paradigma. Encontrar um plano de programação que realize uma meta significa:

- encontrar um determinado conjunto de ações;
- estabelecer uma ordem de execução para essas ações e;
- codificá-las usando uma linguagem de programação.

Seja, por exemplo, o problema 1.

Calcular a soma dos números inteiros de 1 a 10.

Problema 1

Uma solução ou plano natural para o problema 1 pode ser dado pela geração de todos os elementos e depois a realização da soma entre todos os elementos:

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55$$

Se tal plano natural fosse transladado para um plano de programação na linguagem C, por um aprendiz, é possível que a seguinte solução dada pelo programa 1 fosse apresentada.

```
main()
{
        int soma;
        soma = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10;
        printf("\nA soma é %i", soma);
}
```

Programa 1

Uma solução ou plano de programação correto para o problema 1 pode ser dado pelo programa 2.

```
main()
{
        int i, soma=0;
        for(i=1; i<=10; i++)
                soma=soma+i;
        printf("\nA soma é %i", soma);
}
```

Programa 2

Apesar de se tratar de poucas linhas de código C, um aprendiz teria que executar uma série de inferências para obter o programa final, tais como:

- identificar quantas e quais variáveis serão necessárias;
- identificar que a geração dos números e do cálculo da soma requer o uso de uma estrutura de repetição, ou seja, a cada número gerado uma soma parcial deve ser realizada;
- configurar a estrutura de repetição identificando, no enunciado do problema, informações como o início e o fim da seqüência numérica a ser gerada, definindo como a seqüência irá crescer ou decrescer;
- inicializar as variáveis;
- definir o que deve ser feito após a estrutura de repetição.

Se o problema a ser resolvido se torna um pouco mais complexo, o número de inferências aumenta e a identificação de suas relações e ordem também se tornam mais complexas.

Considere agora o problema 2:

Dado pelo usuário, um número inteiro positivo n, calcular e mostrar o valor da seguinte soma: 1 + 1/2 + 1/3 + ... + 1/n

Problema 2

Uma possível lista de ações de programação a serem realizadas em busca de uma solução para o problema 2 é apresentada na tabela 1.

Tabela 1 - Ações de Programação do Problema 2

Calcular soma iterativa
Implementar leitura do teclado
Declarar variáveis
Identificar início de seqüência
Identificar fim de seqüência
Emitir mensagem para o usuário
Definir expressão que estabelece o incremento/decremento da seqüência
Definir expressão para geração de um termo da série
Definir se a seqüência será gerada de forma crescente ou decrescente
Identificar a variação entre dois termos da série
Gerar sequência numérica
Inicializar variável para soma iterativa
Definir expressão geral da soma
Mostrar resultado na tela
Selecionar variável que contém o resultado

Uma vez que as ações sejam combinadas e traduzidas corretamente para uma linguagem de programação, é possível obter um programa correto e completo (programa 3).

```
main()
{
float n, d, f, soma;
soma=0;
printf("Digite um número: ");
scanf("%f", &n);
for(d=1; d<=n; d++)
      {
      f=1/d;
      soma=soma+f;
      }
printf("A soma total é %f", soma);
}
```

Programa 3

Além da dificuldade em identificar as ações de programação, a combinação delas é uma atividade cognitiva complexa. A fim de compor um plano solução para o problema, no nível mental, denominado na literatura de **modelo mental do programa** (VON MAYRHAUSER;VANS,1994), um programador precisa conhecer as relações hierárquicas ou de decomposição entre ações de programação. A figura 1, fornece uma idéia do modelo do processo de construção de programas tipicamente realizado por programadores experientes, adaptado da área de Psicologia da Programação.

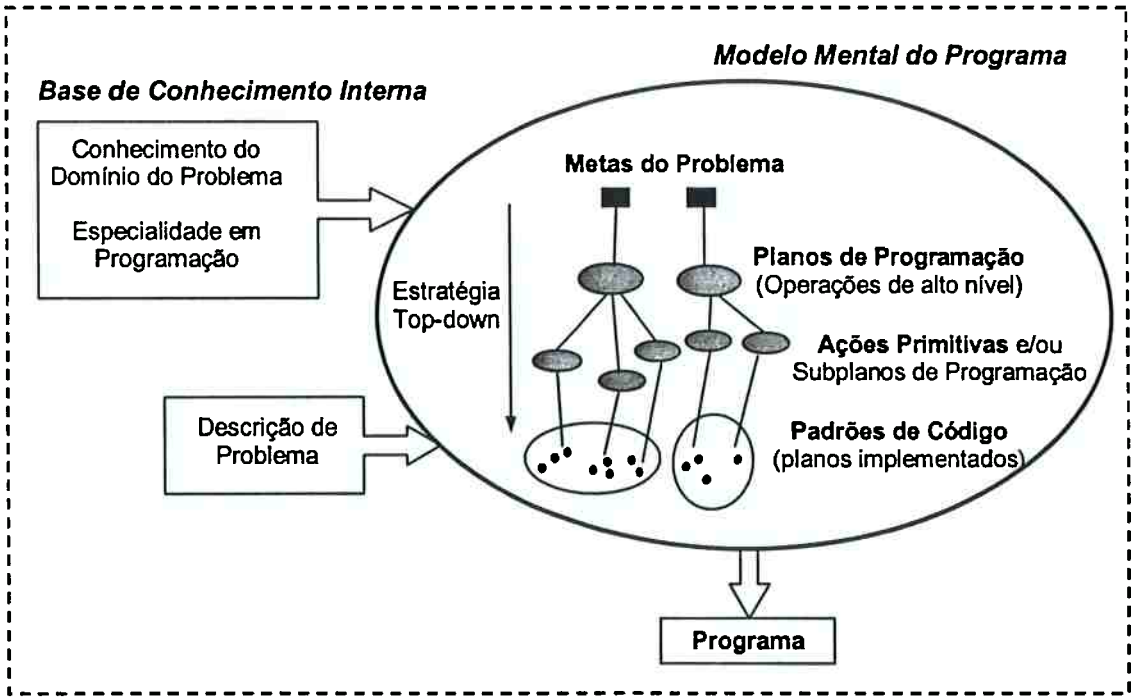


Figura 1 – Modelo do Processo de Construção de Programas por Programadores Experientes (adaptado de Von Mayrhauser e Vans (1994))

Acredita-se que, com a experiência, programadores agrupam fragmentos de código criando estruturas denominadas **planos mentais de programação**, os quais condensam ações primitivas (DU BOULAY,1989;WALLINGFORD,1998). Essas estruturas são armazenadas na base de conhecimento interna do programador ao longo de sua experiência. Dessa forma, programadores experientes conseguem mapear metas do problema diretamente nesses planos de programação. Componentes, estratégias e mapeamentos resumidos na figura 1, encontram-se detalhados no Capítulo 2.

Reclamações comuns de aprendizes da primeira linguagem de programação se referem a eles não conseguirem criar o conjunto de inferências que resolvem um problema sem a ajuda de um professor. Em sala de aula, os alunos têm a impressão de compreenderem como desenvolver um programa que resolva um problema e até acompanham o processo de raciocínio guiado pelo professor. Porém, quando se encontram sozinhos diante de um novo problema, não conseguem resolvê-lo. Um dos motivos que explica este fato é a falta de prática do aprendiz em reconhecer metas do problema a partir do enunciado do problema e mapeá-las em planos de programação. Essa atividade, por sua vez, requer conhecimento sobre planos de programação, os quais são desconhecidos pelo aprendiz.

Entre o enunciado do problema e o código de uma linguagem de programação, que implementa o algoritmo, existe um abismo e o aprendiz não sabe como construir uma ponte que una ambos os lados, como mostra a ilustração da figura 2.

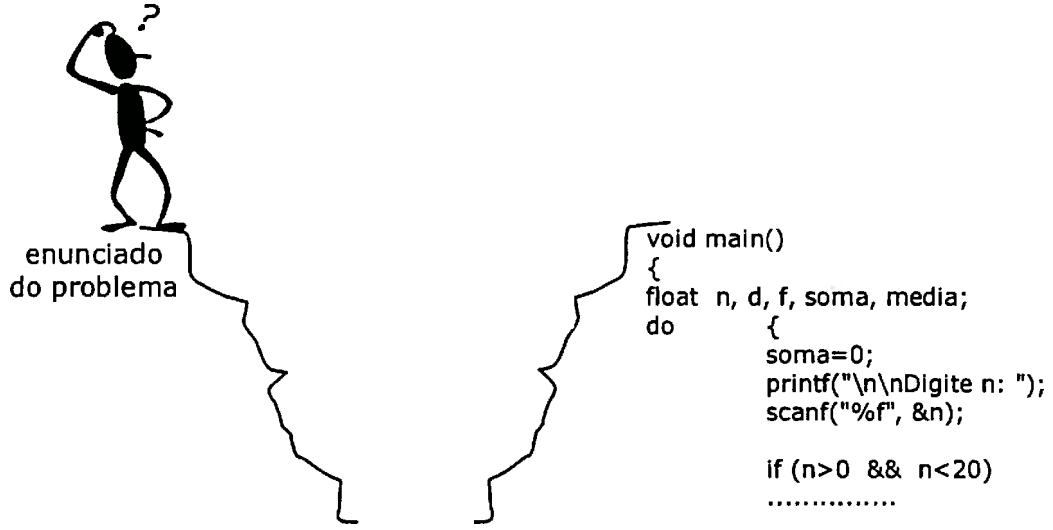


Figura 2 – O abismo entre planos naturais e planos de programação

Algumas estratégias têm sido adotadas para minimizar as dificuldades enfrentadas por aprendizes em programação. Por exemplo, adota-se freqüentemente, em sala de aula e/ou laboratório, atividades e avaliações com consulta. O estudante pode consultar exercícios, livros e apostilas durante a resolução de novos exercícios e avaliações.

Uma outra prática comum em disciplinas introdutórias é a de resolver exercícios de simulação de programas prontos, isto é, o aluno deve fazer o papel do computador executando programas que ele ainda não é capaz de construir. A adoção dessas estratégias são indícios de que a consulta a soluções prontas parece ajudar o aprendiz, principalmente no que diz respeito à memorização. Um estudante pode compreender bem um programa, porém não consegue memorizá-lo a fim de reutilizá-lo. Apesar da adoção destas estratégias, para muitos, a dificuldade ainda permanece. Eles têm a sua disposição a sintaxe e a semântica da linguagem, exercícios resolvidos, mas não possuem a habilidade em resolução de problemas. Nesse caso, o aprendiz não é capaz de criar um modelo mental do programa, identificando as metas do problema, as ações que as resolvem, sua ordenação, e finalmente sua tradução para uma linguagem de programação. A dificuldade em resolução de problemas ocorre também em outras disciplinas e a atividade de programação é útil para forçar o desenvolvimento dessa habilidade.

Na criação e tradução de planos de programação para um programa propriamente dito, as fases de identificação e ordenação das ações devem levar em consideração algumas estruturas da linguagem. Uma ferramenta tradicional que tem o objetivo de facilitar a atividade de programação é o fluxograma. Porém, é uma ferramenta que, por enfatizar o fluxo de controle do programa, não oferece um grau de abstração adequado para o aprendiz construir planos de programação, deixando-o com as mesmas dificuldades para programar. Linguagens de pseudocódigo estão, também, muito mais próximas da linguagem de programação do que de planos de programação, não oferecendo assim, apoio ao aprendiz na construção de programas num nível de abstração mais alto.

Os ambientes tradicionais de desenvolvimento de programas, os compiladores, também não colaboram no sentido de facilitar o desenvolvimento de programas por aprendizes. Compiladores são ferramentas tipicamente projetadas para uso por programadores experientes visando mais o aumento da produtividade de implementação de software do que propriamente fornecer um ambiente para favorecer o aprendizado. Por este motivo, compiladores podem gerar dificuldades quando utilizados para fins de ensino-aprendizagem, independente da linguagem de programação a qual estão associados.

Um outro problema, específico da realidade brasileira, diz respeito à heterogeneidade no nível de conhecimento dos alunos que iniciam um curso superior em que são ministradas disciplinas de linguagens de programação. Assim, é possível encontrar alunos provenientes de cursos tecnológicos, já com algum conhecimento de programação, outros que podem ser considerados simples usuários de computadores e ainda uma

parcela que não teve qualquer contato com computadores. Apesar da defasagem de tempo, Rocha (1995) confirma essa configuração com estatísticas não animadoras:

> Dentro do âmbito da universidade, têm sido oferecidos cursos introdutórios de programação a praticamente todas as carreiras. Existe uma evidente necessidade e interesse em computação. Entretanto os cursos introdutórios de programação, oferecidos para alunos das áreas de exatas e humanas, iniciam suas turmas com uma média de 60 alunos e em poucos meses estão com 20. Obtém-se, portanto, uma média de 60% de desistência, sugerindo pouco interesse por esta disciplina, o que sem dúvida é uma contradição.

Este panorama onde os alunos possuem conhecimento prévio (*background knowledge*) muito diferentes exige ritmos diferentes de ensino e aprendizagem. Por outro lado, ensino individualizado numa sala de aula convencional é impraticável.

Todos os fatores mencionados acabam por levar o aluno a sentir extrema necessidade de acompanhamento e orientação individual, que vai além da disponibilidade do professor e do que os métodos de ensino tradicionais oferecem. O MEC (CEEINF,1999) relata que não se conhece ainda a maneira correta de se introduzir os conhecimentos de computação e coloca a investigação de novos métodos de ensino como tópico de estudo na sub-área Formação Tecnológica dos cursos de Computação e Informática.

Nesta tese, acredita-se que existe uma lacuna na documentação, tradicionalmente, utilizada no ensino da **programação algorítmica**. Entenda-se por programação algorítmica, a parte procedimental embutida em programas desenvolvidos sob paradigmas de programação tais como estruturado e orientado a objetos. Fisher (2001) considera as estruturas básicas de programação como repetição, seleção, comparação, etc., como uma sub-camada básica presente no âmago dos dois paradigmas. Adotou-se, aqui, o termo **programação algorítmica** para tornar clara a existência desse tipo de programação em diferentes paradigmas.

O professor, ao construir programas numa abordagem tradicional, em sala de aula, esforça-se em descrever oralmente suas atividades mentais. Porém, esse tipo de informação não se encontra documentada em livros ou apostilas da área. Em conseqüência, parte dos alunos não conseguem assimilar tais atividades abstratas. Identificando a necessidade de formalizar esse vocabulário, esta pesquisa caminhou na direção de investigar, na literatura, elementos que possam contribuir para a modelagem de componentes capazes de representar tais atividades. Esses componentes são denominados, nesta tese, de **componentes cognitivos de programação**.

Conforme mencionado anteriormente, resultados de pesquisa das áreas da Psicologia da Programação e de Padrões Pedagógicos foram integrados e utilizados para esse propósito.

1.2 Objetivos

Esta pesquisa tem como principal objetivo elaborar um Modelo do Processo de Construção de Programas para Aprendizes de Programação. O modelo é proposto como uma ferramenta cognitiva, definida na literatura como aquela que auxilia pessoas a executarem atividades cognitivas: atividades mentais que não podem ser observadas diretamente ou que podem ser pouco observáveis, tais como ajudar a pensar, conhecer ou aprender. O modelo proposto contém componentes e estratégias definidos com a intenção de apoiar a aprendizagem de programação. Usuários desse modelo podem ser educadores da área que desejam enfatizar ensino e aprendizado de modelos mentais de programas ou ainda por desenvolvedores de sistemas de aprendizagem de programação algorítmica.

Para representação do modelo é definida a linguagem LPC (Linguagem de Programação Cognitiva), baseada em planejamento hierárquico da Inteligência Artificial. Os elementos-chave da LPC são os planos de programação que, combinados, podem representar soluções para problemas de programação. Planos de programação são obtidos a partir de padrões elementares de programação da área de Padrões Pedagógicos. O objetivo da especificação de LPC é facilitar a implementação de dois sistemas propostos na tese e fornecer um possível caminho para realização de planejamento de programas, de forma automática.

O modelo foi implementado por meio de dois sistemas: a BibPC, Biblioteca para Programação Cognitiva, e o TutorC, ambiente para aprendizagem de programação em linguagem C. A BibPC é um sistema de autoria via WEB para apoiar ensino de programação algorítmica em linguagem C. É capaz de capturar conhecimento de planejamento em programação de forma reutilizável para uso e expansão por educadores ou sistemas de aprendizagem.

O TutorC disponibiliza ao aprendiz, componentes de planejamento e de implementação de programas em linguagem C. A intenção é permitir que o aprendiz, apoiado pelos recursos do ambiente, possa ser estimulado a construir modelos mentais de programas, como ocorre com programadores experientes.

Propõe-se que o TutorC seja utilizado por aprendizes antes do uso de um compilador, que em geral é acompanhado por um ambiente de desenvolvimento projetado especificamente para produtividade de software.

Outro objetivo desta tese é modelar um conjunto de componentes para planejamento em programação por meio da BibPC e utilizá-los na resolução de problemas simples no TutorC, com um grupo de alunos do curso de Engenharia Elétrica da Faculdade de Engenharia de Sorocaba (FACENS) a fim de testar o potencial do Modelo proposto.

1.3 Justificativas

A comunidade de software tem desenvolvido vários paradigmas e linguagens de programação a fim de diminuir a distância entre o modelo mental de um programa e a representação externa, o código. Entretanto, sabe-se que o conhecimento de construção de programas para a abordagem algorítmica, focalizado nesta tese, é necessário em mais de um paradigma de programação tal como programação estruturada e programação orientada a objetos. Mesmo poderosas ferramentas *case* tais como ModelMaker (MODELMAKER,2004) e Rational Rose (IBM,2004), as quais geram código automaticamente a partir de modelos orientados a objetos, não são capazes de gerar a parte algorítmica de um software em desenvolvimento, fornecendo apenas um esqueleto do código o qual deve ser preenchido por programadores experientes.

O trabalho de Moström e Carr (1998) faz um estudo sobre como novatos tentam criar seus modelos mentais e sugere o uso de um paradigma de programação que seja mais próximo à maneira de pensar de não-programadores. O trabalho aqui proposto segue outra direção. O objetivo não é o de excluir a necessidade da criação dos modelos mentais, mas criar ferramentas que estimulem a criação desses modelos por aprendizes, tal como ocorre com programadores experientes.

Conforme mencionado na seção 1.1, estudos empíricos da Psicologia da Programação mostram que programadores experientes, quando tentam compreender ou construir programas, buscam por planos mentais de programação (*chunks*). Gellenbeck, Cook e Wiedenbeck apud Pane e Myers (1996) afirmam que aprendizes não conseguem fazer uso dessa técnica porque não aprenderam ainda a relacionar tais estruturas de código às ações de alto nível.

Apesar de pesquisadores como Perkins, Gilmore e Samurçay apud Pane e Myers (1996) sugerirem a exploração desse recurso em ambientes de programação, tanto para compreensão como construção de programas, Fix e Wiedenbeck (1996) e Gellenbeck e Cook[1] apud Pane e Myers (1996) constataram que pouca pesquisa foi dedicada ao desenvolvimento e uso de planos de programação em sistemas educacionais.

Por outro lado, Moström e Carr (1998) relatam que o conhecimento adquirido sobre os modelos mentais de programadores experientes é um meio valioso para ensinar aprendizes a construírem seus próprios modelos. Ryan e Al-Qaimari (1996) acreditam que esse conhecimento pode ser usado como base para formar técnicas instrucionais efetivas a serem aplicadas em situações tradicionais e mais importante ainda, aplicadas a ambientes de aprendizagem. De acordo com Norman (1983, p.7):

> In interacting with the environment, with others, and with the artifacts of technology, people form internal mental models of themselves and of the things with which they are interacting. [...] A person, through interaction with the system, will continue to modify the mental model in order to get to a workable result.

Com base em tais idéias, esta tese propõe um ambiente de aprendizagem que permite a manipulação de um conjunto de componentes de planejamento e implementação de programas que representam modelos e estratégias cognitivas de programadores experientes (VON MAYRHAUSER;VANS,1994; SOLOWAY;EHRLICH,1984). A modelagem dos componentes propostos considera ainda aspectos pedagógicos da área de Padrões Pedagógicos (PP,2001). Assim, acredita-se que aprendizes possam formar seus próprios modelos mentais de programação promovidos pela interação com esse ambiente.

Pode-se encontrar na literatura diversas ferramentas de apoio ao aprendizado de programação. A pesquisa em ambientes para aprendizado inteligente (ILE - *Intelligent Learning Environment*) de programação reporta um extenso e diversificado leque de técnicas e ferramentas. Porém, Brusilovsky (1995) observou que "After 20 years of research there are very few ITS or ILE used regularly for teaching programming in a real classroom, and most of them can not be used outside the university where they were created."

Brusilovsky (1995) e Pane e Myers (2000) destacam algumas características ausentes em ambientes de aprendizagem, em particular em Sistemas Tutores Inteligentes, que impedem seu uso em sala de aula:

---

[1] Gellenbeck, E.M.; Cook,C.R. Does Signaling Help Professional Programmers Read and Understand Computer Programs?. Tech. Report 91-60-3, Oregon State University,1991.

1. **Reusabilidade** - Muitos Sistemas Tutores para programação possuem componentes que implementam idéias interessantes, porém, tais componentes não são projetados de forma a permitir sua reutilização. Um projeto de Sistema Tutor Inteligente é considerado muito complexo, onde cada componente seu requer vários anos de pesquisa de uma pessoa.

2. **Autoria** - Cada sistema suporta uma parte muito pequena do curso oferecendo muito poucas atividades. Assim, por serem sistemas incompletos, não podem ser usados por professores nas salas de aula.

3. **Visibilidade** - Sobrecarga de memória é um problema de todos os programadores, porém, este problema é particularmente incômodo para iniciantes, uma vez que eles ainda não desenvolveram mecanismos para aliviar essa sobrecarga. Um ambiente de programação deveria, em sua interface, possuir informação visível ou facilmente acessível a qualquer momento que ela fosse necessária.

4. **Proximidade de mapeamento** (*closeness of mapping*) - Estudos de Pane e Myers (2000), sobre o perfil de aprendizes de programação, relatam que aspectos tratáveis sobre aprendizagem de programação não foram ainda explorados de maneira intensa, apontando que os maiores problemas envolvem aspectos cognitivos. Uma vez que programação é um processo de criar e traduzir um plano natural para outro que seja executável no computador, um ambiente de aprendizagem deveria minimizar a dificuldade dessa tradução fornecendo primitivas de alto nível para compor planos mentais de programação de alto nível. Se o ambiente não fornece elementos de alto-nível, aprendizes são forçados a compor soluções compostas por primitivas de baixo nível, manipulando a linguagem de programação diretamente, a fim de atingir metas de alto-nível. Esse problema é considerado uma das maiores barreiras cognitivas para programação.

Em relação aos itens 1 e 2, pretende-se conduzir o projeto de TutorC, mencionado na seção 1.2 e detalhado no Capítulo 7, como um conjunto de módulos independentes e reutilizáveis. Brusilovsky (1995) distingui dois tipos de módulos: **módulo baseado em conhecimento** e **módulo de interface** (ferramenta que interage diretamente com o usuário, seja o professor ou o estudante). Um Sistema Tutor Inteligente pode ter vários desses tipos de módulos, os quais interagem um com o outro. Nesta pesquisa, foi adotada a filosofia de componentes reutilizáveis. A BibPC, mencionada na seção 1.2 e detalhada no Capítulo 7, pode ser vista como um sistema de autoria para professores, o qual contém dois módulos: a interface WEB e o banco de dados (módulo que contém o conhecimento cognitivo de programação).

O sistema TutorC reutiliza o módulo de conhecimento da BibPC, via Internet, para implementar outros módulos tais como a interface com o estudante e módulo diagnóstico.

Em relação ao item 3, um aspecto que se objetiva salientar neste trabalho é o de identificar quais são os elementos que compõem o que Pane e Myers (2000) chamam de **informação visível**. Deseja-se tratar tais elementos como **componentes de planejamento e implementação** para programação, ou seja, componentes que professores experientes acreditam ser de fácil compreensão e que possam ser úteis em atividades cognitivas de programação, disponibilizando-os, para o aprendiz, através de uma interface.

Em relação ao item 4, deseja-se, nesta tese, propor um trabalho inovador no sentido de contribuir para o problema da dificuldade do aprendiz em construir modelos mentais de programas, considerando a programação algorítmica. Os sistemas propostos podem ser vistos como ferramentas cognitivas. Segundo Shute e Psotka (1994), o projeto de um STI, tende a ser trocado por uma coleção de ferramentas educacionais especializadas ou ferramentas cognitivas: aquelas que ajudam pessoas a executarem atividades cognitivas tais como ajudar a pensar, conhecer ou aprender. Para Woolf[2] (1988) apud Shute e Psotka (1994) um tutor pode ser considerado inteligente se possui mecanismos que modelam o processo de pensamento de especialistas do domínio e de estudantes, definindo o tutor como ferramenta cognitiva.

De acordo com Gray e Altmann (2001) ferramentas cognitivas envolvem modelagem cognitiva, a qual é definida como a aplicação de teoria cognitiva em problemas aplicados. Neste trabalho, utiliza-se da teoria cognitiva proposta por diversos pesquisadores da área da Psicologia da Programação, os quais investigaram o processo de compreensão e construção de programas por programadores experientes e aprendizes. A teoria é aplicada ao problema da dificuldade de construção do modelo mental de programa pelo aprendiz em programação. Shute e Psotka (1994) coletaram a opinião de alguns pesquisadores sobre o significado da inteligência num Sistema Tutor Inteligente e Wayne Gray define como:

> I concede a wide latitude on the application of the term "ITS" in regard to instructional systems. However, at some level and to some degree, there should be some sort of "cognitive modeling" technology involved. The modeling can be of an ideal student, instructor, or grader, or of a less-than-ideal problem solver [...]

[2] Woolf, B. Intelligent Tutoring Systems: a survey. H. Schrobe, ed. Exploring artificial intelligence, 1-44. Los Altos, CA: Kaufmann, 1988.

A modelagem cognitiva leva em conta a natureza do conhecimento. Self (1995) afirma que muitos domínios têm sido modelados com conhecimento objetivo, entretanto logo se torna aparente que os problemas reais de aprendizado não residem no conhecimento objetivo, mas na sua relação com o conhecimento menos objetivo. No domínio da programação o foco se move da sintaxe da linguagem para aspectos de projeto de programação. É nesse contexto que esta pesquisa se insere e oferece suas contribuições. Embora o escopo desta tese se limite ao ensino e aprendizado de resolução de problemas em programação algorítmica, considera-se que o aprendizado de programação se completa através de um ciclo de conhecimento que envolve (i) conceitos da linguagem de programação; (ii) resolução de problemas e (iii) comportamento dinâmico de programas (figura 3).

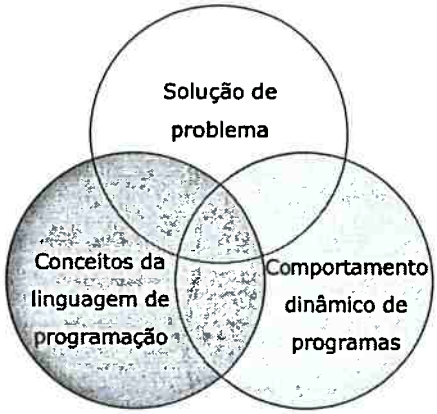


Figura 3 – O ciclo do aprendizado de programação

1.4 Metodologia

Esta pesquisa teve origem a partir das dificuldades encontradas ao ministrar aulas de Introdução à Computação na Faculdade de Engenharia de Sorocaba (FACENS) para alunos iniciantes em programação algorítmica nos cursos de Engenharia Elétrica e Civil, desde o ano de 1998. O levantamento bibliográfico inicial incentivou a continuidade da pesquisa uma vez que mostrava a concordância entre pesquisadores de que os métodos tradicionais de ensino e aprendizado de programação, assim como as soluções propostas na literatura, não atacavam as principais dificuldades encontradas por aprendizes em programação.

A experiência em sala de aula e laboratório apontava a necessidade de formalizar, as atividades cognitivas que envolviam o processo de programação algorítmica. Uma primeira observação foi a de que o professor, ao construir programas em sala de aula para aprendizes, tende a descrever oralmente suas atividades mentais, as quais não são

documentadas. Em conseqüência disso, parte dos alunos não conseguem assimilar tais atividades abstratas sendo incapazes de repetir o processo em situações posteriores ou ainda se lembrar do processo e adaptá-lo para novas situações. Uma vez detectada a existência de uma linguagem informal, no nível cognitivo, a pesquisa caminhou na direção de identificar e representar os elementos que envolviam tal linguagem.

Assim, a área da Psicologia da Programação foi estudada, a qual relata diversos resultados empíricos sobre compreensão e construção de programas por programadores experientes e aprendizes. Essa revisão na literatura permitiu:

(i) identificar quais componentes cognitivos são manipulados por programadores experientes quando constróem programas;

(ii) identificar quais são as dificuldades no processo de construção de programas por aprendizes;

(iii) propor um modelo de processo de construção de programas adequado às limitações de conhecimento dos aprendizes em programação.

Um primeiro trabalho em direção ao modelo foi o de identificar trechos mínimos de código (partes de uma linha de código) em programas e associá-los à ações primitivas de programação. E ainda agrupar ações nesse nível para compor ações num nível de abstração mais alto, formando assim, planos e sub-planos de programação. Houve também uma tentativa de identificar metas gerais de programação que pudessem ser alcançadas por esses planos de programação.

Um primeiro protótipo de um ambiente para aprendizagem foi implementado (OLIVEIRA;LEMOS, 2002). Nele, o aprendiz poderia navegar em diversas estruturas em árvore que representavam o conhecimento cognitivo de programação. A raiz de cada árvore correspondia a uma meta, nós intermediários correspondiam a planos e sub-planos de programação e as folhas correspondiam às ações primitivas. O código correspondente à uma ação selecionada pelo aprendiz era adicionado, automaticamente, na seqüência do programa sob construção. Para o sistema compor o código das ações, o aprendiz deveria informar dados específicos do problema.

A generalidade das metas e a alta granularidade das ações primitivas pareceram tornar o ambiente complexo para aprendizes e também de difícil modelagem. Na busca por uma modelagem e representação do conhecimento cognitivo mais adequados, duas outras áreas foram incluídas na pesquisa: a área de Padrões Pedagógicos para Programação e Planejamento Hierárquico (*Hierarchical Planning*) da Inteligência Artificial.

A pesquisa recente em padrões para programação permitiu avanços significativos na modelagem dos planos de programação desejados. A área de planejamento hierárquico da IA contribuiu para definir uma linguagem para representar atividades cognitivas de programação, as quais podem ser implementadas em ambientes de aprendizagem.

Na seqüência, foram implementados sistemas de apoio ao ensino e aprendizado cognitivo de programação. Nessa proposta não foi considerada a modelagem de **planos naturais** e seu mapeamento aos **planos de programação**. Acredita-se que a interação do aprendiz com componentes de planejamento para programação colaboram na formação do conhecimento sobre como mapear planos naturais a planos de programação. Mas acima disto, acredita-se que interagindo com tais componentes, o aprendiz possa criar e expandir seu conhecimento sobre componentes cognitivos de programação, assim como sua habilidade em resolução de problemas. Finalmente, foi realizada uma avaliação preliminar do modelo do processo de construção de programas proposto, com um grupo de alunos de primeiro ano de Engenharia Elétrica da FACENS. Para tal, o conteúdo modelado em BibPC foi utilizado nas aulas teóricas e o sistema TutorC, em laboratório.

1.5 Organização da Tese

No Capítulo 2 é apresentada a teoria cognitiva da área da Psicologia da Programação que condensa estudos sobre modelos mentais de programadores experientes quando executam atividades de construção e compreensão de programas. Conceitos e estratégias da área de pesquisa são definidos, os quais formam a base para compor a proposta de um Modelo de Processo de Construção de Programas para Aprendizes.

No Capítulo 3 é apresentada uma síntese da pesquisa em Padrões de Programação, particularmente Padrões Elementares de Programação. Trabalhos de modelagem de padrões elementares realizados por professores dessa comunidade são coletados e estudados. O objetivo é observar que tipos de conhecimento de programação estão explícitos nos padrões, para posterior aproveitamento na construção de planos de programação.

No Capítulo 4, ambientes de aprendizagem para programação, da literatura, são resumidos e discutidos.

No Capítulo 5 é proposto um Modelo de Processo de Construção de Programas para Aprendizes, baseado nos componentes e estratégias estudados das áreas da Psicologia

da Programação e de Padrões Pedagógicos. São descritos os componentes do modelo, e em particular, os planos e estratégias de programação. Discussões sobre formalização, implementação e validação do modelo são apresentadas.

No Capítulo 6 é definida uma linguagem para planejamento hierárquico em programação, a Linguagem de Programação Cognitiva (LPC), baseada na área de planejamento da Inteligência Artificial. A LPC permite representar os componentes e estratégias do Modelo de Processo de Construção de Programas para Aprendizes proposto no Capítulo 5 e fornece a base para implementação de duas ferramentas: a BibPC, uma biblioteca de componentes cognitivos de programação e TutorC, um sistema tutor inteligente para aprendizado de programação algorítmica.

No Capítulo 7 é apresentada a implementação dos sistemas BibPC e TutorC. BibPC é projetada como uma ferramenta de autoria, acessível pela WEB, capaz de capturar conhecimento especialista de programação algorítmica, de forma estruturada e reutilizável. Algumas telas são apresentadas a fim de mostrar como um professor deve utilizar o ambiente BibPC. Da mesma forma, aspectos de implementação do TutorC são apresentados, relacionando-os com a arquitetura geral de Sistemas Tutores Inteligentes. Algumas telas são apresentadas a fim de mostrar como um estudante trabalha em TutorC, durante a construção de um programa.

No Capítulo 8 é relatada uma experiência preliminar da utilização do Modelo proposto com alunos da disciplina de introdução à programação do curso de Engenharia Elétrica da FACENS.

No Capítulo 9 são apresentadas as conclusões finais da tese, suas contribuições, e trabalhos futuros.

2 MODELOS COGNITIVOS DE PROGRAMAÇÃO

> *"O saber de Jack sobre A nada mais é além de modelos mentais .... que Jack pode utilizar para responder perguntas sobre A"* (MINSKY, 1985, pg.303)

No fim dos anos 70, dentro da área da Psicologia começaram a ser investigados os primeiros problemas existentes no campo da programação, dando origem à área da Psicologia da Programação. Em 1987, formou-se um grupo de interesse em Psicologia da Programação (PPIG - *Psychology of Programming Interest Group*) que reúne pessoas de diversas comunidades, tanto das universidades como da indústria, para explorar interesses comuns nos aspectos psicológicos da programação. Encontros anuais têm ocorrido ao longo dos últimos quinze anos. Os temas que têm sido abordados incluem:

- estruturas cognitivas em programação envolvendo aspectos cognitivos de aquisição de habilidades por aprendizes, representações mentais sobre programação, particularmente, modelos cognitivos de programadores experientes;
- protocolos de coleta e análise de comportamento de programação, assim como estudos empíricos de construção, compreensão de programas e estudos de erros cometidos por programadores novatos e experientes;
- avaliação de linguagens, ambientes de programação e técnicas para desenvolvimento, compreensão e depuração de programas, particularmente, estudos de efeitos de ferramentas sobre aprendizado e compreensão de software;
- similaridades e diferenças entre linguagens de programação e linguagem natural com objetivos de programação para usuários finais;
- ensino, aprendizado e transferência de conhecimento em várias linguagens de programação e paradigmas;
- aspectos sobre manutenção e reuso de software;
- processos colaborativos em desenvolvimento de software.

A programação é considerada uma atividade cognitiva, portanto, a maioria dos estudos empíricos da literatura descrevem a **Análise de Tarefa Cognitiva** (CTA - *Cognitive Task Analysis*) em programação. Esse tipo de análise é um processo sistemático pelo qual os elementos cognitivos são identificados durante o desempenho da tarefa. Assim, Análise de Tarefa Cognitiva foca atividades mentais que não podem ser observadas diretamente ou que podem ser pouco observáveis. O esforço na criação de modelos cognitivos de programação tem sido motivado para apoiar o processo de manutenção de software, através da construção de ferramentas de exploração de software a fim de facilitar a complexa e demorada atividade de compreensão de programas pobremente documentados (STOREY ET AL.,1999).

Essas ferramentas embutem recursos que facilitam a construção do modelo mental do programa pelo programador, tais como:

- Indicam relações entre objetos do software.
- Reduzem o efeito de planos deslocalizados, usando realces nas declarações. Um **plano deslocalizado** é um algoritmo ou plano de programação fragmentado (espalhado) no código fonte (SOLOWAY;LETOVSKY,1986;LAMPERT ET AL.,1988).
- Fornecem mecanismos para facilitar a construção e visualização de abstrações.
- Recursos de navegação para explorações no código.

Outra motivação é dirigida ao apoio no projeto de novas linguagens de programação e novos ambientes de programação capazes de facilitar a atividade de construção de software por programadores experientes e não-programadores. Essas ferramentas objetivam ser, por exemplo, ambientes de programação com linguagens mais naturais onde detalhes de programação se tornam cada vez mais implícitos ou ambientes poderosos para depuração de programas. Todas essas aplicações são, tipicamente, para uso por programadores experientes ou, no caso, de linguagens naturais para não-programadores (usuários finais).

Apesar da pesquisa sobre modelos mentais ter se tornado popular no domínio da programação para as finalidades acima, Pane e Myers (1996) ressalta que poucos esforços foram realizados para explorar sua aplicação em propostas educacionais. Ryan & Al-Qaimari (1996) afirmam que a identificação dos modelos cognitivos de programadores experientes pode contribuir significativamente para o desenvolvimento de ferramentas cognitivas para ensino de construção de programas. Segundo Eberts[3] (1994) apud Ryan e Al-Qaimari (1996) "[...] knowledge acquired about experts' mental models is a method to communicate an accurate mental model to novices."

Neste capítulo, o interesse está particularmente focado na pesquisa sobre como programadores novatos e experientes, constroem e entendem programas. Não foi encontrada, na literatura, uma clara distinção entre aprendizes de programação e programadores novatos. Os dois termos parecem ser utilizados de forma equivalente. Ainda assim, é possível definir **aprendizes** como aqueles que pretendem aprender programação, porém desconhecem qualquer conceito da área. **Programadores novatos** podem ser considerados como aqueles que conhecem conceitos isolados de programação, mas ainda não conseguem integrá-los a fim de compor uma solução para um problema, ou seja, um programa.

---

[3] Eberts, R. E. *User Interface Design.* Prentice-Hall, 1994.

Exemplos de conceitos isolados são: variável, atribuição, estrutura de decisão, estrutura de repetição e outros. **Programadores experientes** são aqueles que já adquiriram habilidade em resolução de problemas por programação e possuem habilidade em construir o modelo mental do programa e mapeá-lo sobre conceitos da linguagem. Uma vez que o foco nesta tese é aprendizagem em resolução de problemas de programação, o alvo atinge tanto aprendizes em programação como programadores novatos. Assim, não será feita distinção entre os termos neste trabalho.

Como mencionado anteriormente, sob o ponto de vista cognitivo, a programação é freqüentemente definida como um processo de transformar um plano que está numa forma familiar (plano natural) para um plano que seja compatível com o computador (plano de programação). Na execução desse processo muitas dificuldades se originam porque existe uma grande distância entre esses dois planos.

Segundo teorias cognitivas da Psicologia da Programação, construção e compreensão de programas são dois processos que usam conhecimento prévio para adquirir novo conhecimento: o plano mental do programa a ser construído ou do programa já existente. Tal plano corresponde a uma representação mental ou explicitada em um ambiente apropriado, do programa em questão. Em ambos os processos, Von Mayrhauser e Vans (1994) e Welty (1995), entre outros pesquisadores, identificam dois tipos de conhecimento empregados pelos programadores experientes:

Conhecimento do Domínio do Problema

Refere-se ao conhecimento do problema a ser resolvido pelo programador, independente da linguagem ou ambiente de programação. Existem muitas evidências empíricas de que o conhecimento do domínio de problema é essencial para compreensão, projeto e implementação de software (RUGABER,2000), (BROOKS, 1983; CURTIS ET AL., 1988; SOLOWAY; ADELSON, 1985). Além da familiaridade com o domínio no qual o software irá operar, o programador deve saber o que o software deverá fazer naquele domínio, ou seja, entender qual é a atividade específica a ser realizada no domínio. Durante o processo de compreensão de um programa existente ou de construção de um programa, os alunos adquirem mais conhecimento específico do domínio.

Conhecimento de Programação

Há diferentes tipos de conhecimento de programação que têm sido notados em programadores experientes, entre eles:

- conhecimento de linguagens de programação,

- manipulação de ambientes de programação,
- princípios e estilos de programação,
- habilidade em resolução de problemas de programação,
- conhecimento de planos de programação,
- conhecimento de algoritmos.

A representação mental do programa em consideração, que o programador elabora durante o processo de entendimento ou de construção é chamada **Modelo Mental do Programa** e o processo global que conduz à criação do Modelo Mental do Programa é chamado **Modelo Cognitivo de Programação** (VON MAYRHAUSER;VANS,1994); (STOREY AT AL.,1999). A maioria das pesquisas empíricas sobre modelos mentais de programadores utilizou o cenário da compreensão de programas, propondo os chamados **Modelos Cognitivos de Compreensão de Programas**. Nesta seção, focaliza-se o estudo do processo de compreensão de programas a fim de se propor um modelo do processo de construção de programas para aplicá-lo na construção de um ambiente de aprendizagem.

Para maior clareza, o chamado Modelo Cognitivo de Compreensão de Programas será denominado, aqui, de **Modelo do Processo de Compreensão**. Para o processo cognitivo de construção de programas será denominado **Modelo do Processo de Construção**.

Apesar dos dois processos possuírem muitas interseções, eles devem ser descritos de forma independente uma vez que diferem quanto ao tipo de conhecimento disponível e quanto ao tipo de conhecimento produzido, conforme figuras 4 e 5. Por exemplo, no processo de entendimento de um programa, o código do programa está disponível para análise pelo programador, porém ele não conhece o que o programa faz, ou seja, não conhece as metas do programa, uma vez que este é o conhecimento a ser descoberto. Por outro lado, no processo de construção de um programa, as metas a serem alcançadas estão disponíveis no enunciado do problema, porém não existe o código, uma vez que este deve ser ainda construído.

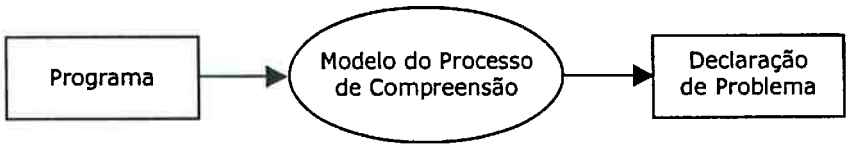


Figura 4 - Entrada e Saída do Processo de Compreensão de Programa

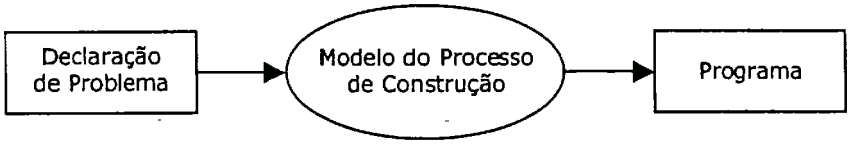


Figura 5 - Entrada e Saída do Processo de Construção de Programa

A seguir, descreve-se resumidamente os modelos dos dois processos:

Modelo do Processo de Compreensão

O processo de compreensão de programas pode ser definido como um processo de formular metas hipotéticas, verificá-las se são verdadeiras ou falsas e revisá-las quando necessário. Programadores tentam encontrar no código, planos que alcancem essas metas hipotéticas. Metas existem em todos os níveis de abstração, assim como planos.

Modelo do Processo de Construção

O programador elabora um modelo mental através de uma hierarquia de várias entidades: *metas*, *planos* e *estruturas de código* (VON MAYRHAUSER;VANS, 1994). Trata-se de um processo dirigido pelas metas do problema. A fim de alcançar uma meta, um plano é construído ou selecionado da base de conhecimento do programador. Esse plano é decomposto em subplanos mais detalhados até estar completamente implementado por estruturas de código.

Nos anos 80 e 90 foram desenvolvidos importantes estudos sobre compreensão de programas. As próximas seções resumem os principais conceitos cognitivos descobertos pela área.

2.1 Modelo do Processo de Compreensão de Programas

A pesquisa em como programadores entendem programas resultou no desenvolvimento das várias teorias cognitivas que descrevem o processo. Embora tais teorias tenham suas diferenças, elas compartilham elementos e conceitos que envolvem atividades básicas no entendimento de programas. Os experimentos foram realizados, em sua maioria, utilizando-se linguagens procedurais tais como: Pascal, C, Basic, Cobol e Fortran. Trabalhos mais recentes envolvem estudos cognitivos sobre compreensão de software orientado a objetos (RAMALINGAM;WIEDENBECK,1997;RYAN;AL-QAIMARI,1996).

O'Brien at al. (2001) declara que cada um dos modelos representa importantes aspectos da atividade de compreensão de programas, que podem ser resumidos em três tipos básicos de processos usados pelos programadores para entender programas de computadores: **hipótese pré-gerada** (ou abordagem *top-down*), **hipótese baseada em inferência** e **compreensão *bottom-up***.

Programadores podem usar um ou mais tipos de processo para concluírem seu entendimento. Usando um processo *top-down*, programadores podem começar com uma hipótese geral sobre a natureza do programa e buscar refinar o entendimento do programa propondo e testando incrementalmente hipóteses específicas. Num processo *top-down* as hipóteses são baseadas no conhecimento previamente obtido pelo programador, ou seja, originárias de sua própria base de conhecimento. Num processo do tipo **hipótese baseada em inferência**, o programador faz uma declaração sobre a natureza do software baseada em informação específica, obtida do programa. A compreensão *bottom-up* ocorre quando o programador examina o programa diretamente, linha por linha, construindo um entendimento geral do mesmo.

O primeiro modelo que surgiu de estudos empíricos foi proposto em 1979 por Shneiderman apud von Mayrhauser e Vans (1994). Segundo seu modelo, programas são entendidos de maneira *bottom-up*, ou seja, lendo-se o código fonte e mentalmente, agrupando pedaços do software de baixo nível e relacionando-os com abstrações de nível mais alto. De acordo com essa teoria, as abstrações são também agregadas formando uma estrutura semântica interna para representar o programa. A estrutura mental é construída através do reconhecimento da função dos componentes abstratos do programa e dos fragmentos de código. Essa estratégia de agregações é denominada de *chunking*. Experimentos posteriores realizados em 1987 por Pennington apud von Mayrhauser e Vans (1994) reforçaram a idéia de que programas são entendidos de maneira *bottom-up*.

Um modelo diferente foi proposto por Brooks (1983). Nesse modelo um programador tenta compreender um programa utilizando-se de uma estratégia *top-down*, reconstruindo conhecimento do domínio e mapeando-a em direção ao código fonte. Essa estratégia começa com a hipótese de uma meta global que o programa alcança. Essa meta é refinada dentro de uma hierarquia de hipóteses secundárias (sub-metas). Essa cascata continua até produzir uma hipótese que seja específica o suficiente para que o programador possa verificá-la no código do programa.

A verificação de uma hipótese começa quando ela trata com operações que possam estar associadas a fragmentos de código conhecidos. A identificação desses fragmentos confirmam a presença de uma estrutura ou operação particular. Brooks denominou tais estruturas de código de ***beacons***, os quais são acumulados na memória do programador, ao longo de sua experiência, por meio da formação de ***chunks* de programação**. Aceitar ou rejeitar uma hipótese depende fortemente da presença ou ausência desses fragmentos de código. *Beacons* são importantes porque eles formam o primeiro conjunto de mapeamentos entre hipóteses e código do programa. Tiemens (1989) afirma que pesquisas apontam que programadores experientes podem localizar *beacons* mais eficientemente do que novatos e ainda que programadores experientes recordam-se de *beacons* melhor do que linhas de código não pertencentes a estes.

O modelo proposto por Soloway e Ehrlich (1984) é um modelo baseado em inferência, onde o programador realiza buscas no código e a partir do conhecimento incompleto a respeito do código, infere uma hipótese abstrata. Essa hipótese corresponde à intenção por trás do código, ou seja, a meta desejada. Em seguida, o programador busca, dentro do programa, por um plano capaz de realizar a meta. O plano é decomposto em submetas e, novamente, sub-planos tem que ser encontrados para realizar as submetas. O programador continua nessa abordagem *top-down* até reconhecer um bloco de código para um subplano. O bloco de código representa a implementação do subplano.

Os planos são classificados segundo seu nível de abstração. **Planos estratégicos**, os quais localizam-se no nível mais alto, descrevem a estratégia global usada num programa e são decompostos como um conjunto de **planos táticos**. Planos táticos descrevem estratégias locais de solução de problema, contêm especificações de algoritmos independentes da linguagem de programação, podem incluir estruturas de dados abstratas e localizam-se num nível intermediário de abstração. **Planos de implementação** são usados para implementar planos táticos. Esses planos contêm fragmentos de código adquiridos através da experiência do programador.

Durante o processo de compreensão, o programador busca no programa em estudo, por fragmentos de código conhecidos. Um modelo mental é construído durante a compreensão *top-down* e consiste de uma hierarquia de metas e planos. Esse modelo foi utilizado para implementar PROUST (JOHNSON;SOLOWAY,1985), um tutor inteligente para ensino da linguagem LISP. A figura 6 mostra parte de uma hierarquia de metas e planos modelados em PROUST.

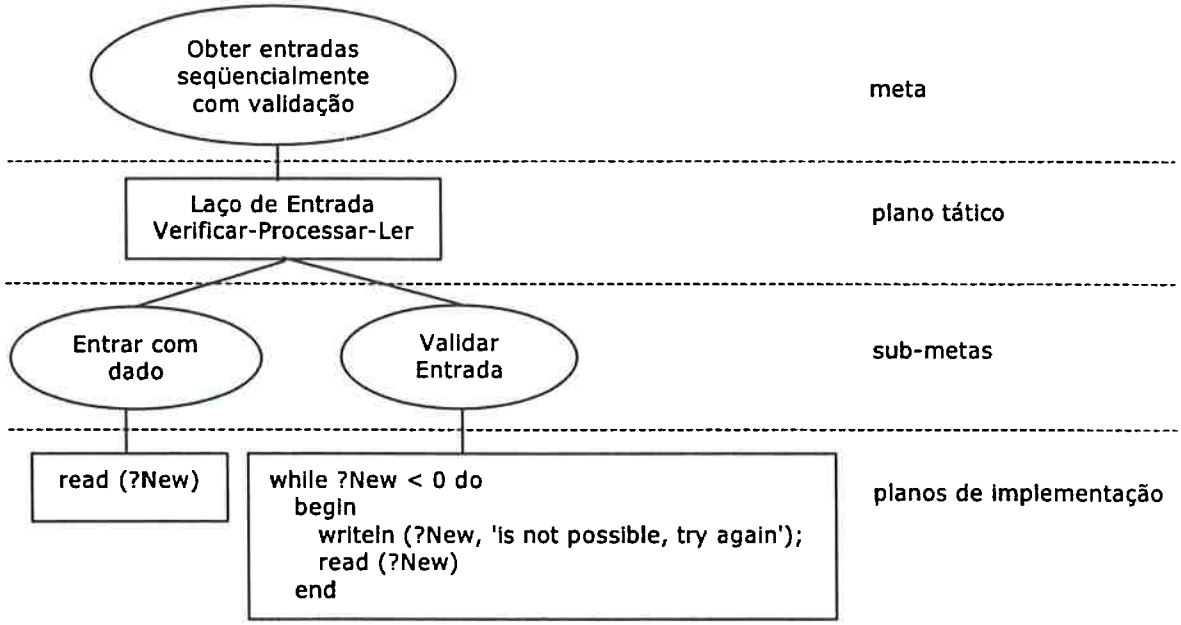


Figura 6 – Exemplo de uma hierarquia de metas e planos de PROUST.

Experimentos posteriores de Letovsky apud Von Mayrhauser e Vans (1994) realizados em 1986 e confirmados por estudos mais recentes de Von Mayrhauser e Vans (1994) indicaram que a compreensão de programas é construída concorrentemente em vários níveis de abstração e alternando livremente entre as estratégias *top-down* e *bottom-up.* Durante o processo de compreensão do programa qualquer uma das estratégias podem se tornar ativas a qualquer tempo. *A* estratégia *top-down* é ativada quando a linguagem ou código é familiar. A partir de algum entendimento do código é gerada uma hipótese de alto nível sobre alguma funcionalidade do programa. A hipótese abstrata vai sendo decomposta até chegar no código novamente. Isto corresponde às propostas de Brooks (1983) e Soloway e Ehrlich (1984). *A estratégia bottom-up* observada por Shneiderman e Pennington é ativada quando o código e a aplicação são completamente não-familiares.

Um exemplo que descreve a troca de estratégias do programador durante a construção do modelo mental do programa é: um programador pode usar a estratégia *bottom-up* e reconhecer um padrão de código (*beacon*) que indica a existência de uma tarefa de ordenação no programa. Para isso ele usa seu conhecimento prévio sobre planos de programação. Isso conduz à hipótese de que o programa faz ordenação, causando um salto para o modelo *top-down*, uma vez que essa é uma funcionalidade abstrata. Então, o programador decompõe a tarefa de ordenação em submetas e busca no código por novos padões de código que confirmem essas submetas. Se durante a busca, ele encontra uma seção de código desconhecido, ele novamente pode preferir saltar para a abordagem *bottom-up* para construir o modelo de programa.

2.1.1 Elementos do Modelo do Processo de Compreensão

Cada um dos autores dos modelos do processo de compreensão, descritos na seção anterior, utiliza uma terminologia para explicar seu modelo. Porém, há intersecções entre conceitos e estratégias nesses modelos, os quais são reunidos no trabalho de von Mayrhauser e Vans (1994) e resumidos nesta seção. Tais modelos são descritos através de um conjunto de **estruturas de conhecimento** e de **estratégias** utilizadas no processo. Estruturas de conhecimento correspondem aos elementos criados pelo programador e que constituem o modelo mental do programa sob consideração. Estratégias são utilizadas pelo programador para obter o modelo mental de seu entendimento sobre esse programa.

2.1.1.1 Estruturas de Conhecimento

Estruturas de texto - Correspondem a trechos ou blocos de código. O conhecimento de estruturas de texto é construído através da experiência e combinação de elementos da linguagem de programação. Exemplos dessas estruturas são: sub-partes de estruturas de controle (por exemplo, estruturas para laços e estruturas condicionais) tais como início e fim, variáveis de controle, expressões de teste; definições de variáveis. Essa micro-estrutura do texto do programa consiste de declarações (*statements*) e seus relacionamentos.

***Chunks* de programação** - são estruturas de texto ou ainda fragmentos de código, para as quais é possível fazer uma hipótese de meta, ou seja, a estrutura de texto embute uma funcionalidade. Como já mencionado, Brooks (1983) denominou tais fragmentos de *beacons*. Gellenbeck e Cook[4] (1991) apud Pane e Myers (1996) reafirmam que *beacons* são estruturas úteis na compreensão de programas para confirmar ou sugerir o que o código sob estudo faz ou então, servem como blocos de construção na elaboração de um programa. Pane e Myers (1996) cita um exemplo simples de *chunk* existente num certo tipo de tarefa de ordenação: a operação de troca de valores entre duas variáveis:

```
temp = x;
x = y;
y = temp;
```

Pennington (1987) define tais agrupamentos como macro-estruturas. A macro-estrutura é uma abstração do bloco de código e pode ser rotulada, neste exemplo, de "ordenação".

[4] Gellenbeck, E.M.; Cook,C.R. Does Signaling Help Professional Programmers Read and Understand Computer Programs?. Tech. Report 91-60-3, Oregon State University,1991.

Chunks de programação de níveis mais baixos podem formar *chunks* de nível mais alto. Estes últimos podem ser descritos totalmente por código ou pode consistir de vários rótulos e de relacionamentos entre eles.

Planos de programação – da mesma forma que *chunks*, planos de programação são elementos que satisfazem metas de um problema de programação. O termo surgiu na mesma época dos estudos empíricos sobre o comportamento de programadores e aprendizes, durante o desenvolvimento de sistemas tutores para programação. *Chunks* de vários níveis de abstração foram modelados e embutidos em alguns desses sistemas e denominados de **planos de programação**. Um exemplo clássico foi o tutor PROUST (JOHNSON;SOLOWAY,1985). A figura 7 mostra um plano de programação de PROUST.

```
PLANO DE REPETIÇÃO LER-PROCESSAR-VERIFICAR_SENTINELA
        Constantes: ?Stop
        Variáveis: ?New
        Template:
                repeat
                        sub-plano Entrada (?New)
                        sub-plano Verificar-Sentinela (?New, ?Stop)
                until ?New = ?Stop
```

Figura 7 - Plano de repetição do sistema PROUST

Os planos podem ser de alto, intermediário ou baixo nível de conceitos de programação. Em níveis mais altos, planos são constituídos por esquemas com objetos genéricos e sub-planos de programação (planos de nível de abstração mais baixo e, portanto, mais simples). Objetos genéricos descrevem, por exemplo, qual a função que uma instância do programa desempenha quando uma hipótese é provada. Planos de níveis mais baixos contêm trechos de código que implementam sub-planos de programação. Em resumo, planos de programação são **artefatos** construídos por projetistas para serem embutidos num sistema de software enquanto que *chunks* é uma **estrutura cognitiva** criada pelo programador. Neste trabalho não será feita distinção entre os dois termos, uma vez que um dos objetivos da tese é descobrir *chunks* de programação ou utilizar *chunks* propostos por pesquisadores para serem usados como planos de programação num sistema de software.

Hipóteses - São suposições ou hipóteses de metas do programa que surgem durante o processo de compreensão de um programa e que devem ser provadas. Metas de nível de abstração mais alto podem ser decompostas em submetas. Hipóteses de metas compreendem suposições sobre o que alguma parte do programa pode ser, como por exemplo, uma função (VON MAYRHAUSER;VANS,1994).

Uma maneira de programadores experientes confirmarem suas hipóteses é buscando *beacons* no programa (CROSBY;STELOVSKY,1990). Brooks (1983) concluiu que hipóteses podem não ser provadas por duas razões principais: (i) o código completo para provar uma hipótese não pode ser encontrado ou (ii) um único pedaço de código satisfaz duas hipóteses diferentes. Geração e discriminação de hipóteses sobre o código são mecanismos importantes em compreensão de programas.

Planos do domínio do problema - Tratam com elementos do mundo real, ou seja, incorporam o conhecimento sobre a área do problema. Por exemplo, planos úteis no desenvolvimento de uma ferramenta de software para projetar automóveis inclui esquemas sobre a função ou aparência de um carro genérico. Tais esquemas devem conter campos (*slots*) para inserir elementos do domínio do problema tais como rodas, portas, motores. Em compreensão de programa esses planos de domínio são indispensáveis para o entendimento da funcionalidade da ferramenta.

2.1.1.2 Estratégias para Compreensão de Programas

Programadores usam estratégias para formar o modelo mental de seu entendimento sobre um programa. As estratégias identificadas pelos estudos empíricos são:

Estratégia sistemática - Consiste da leitura e entendimento sistemáticos do código do programa, ou seja, linha por linha, enquanto o programador constrói uma representação mental de níveis mais altos de abstração.

Estratégia oportunística - O programador estuda o código de uma maneira aleatória ou acidental, sem seguí-lo sistematicamente. Essa estratégia pode ser classificada ainda em:

- *top-down* quando o estudo ocorre das metas em direção ao código,
- *bottom-up* quando o estudo ocorre do código em direção as metas.

Estratégia de comparação - o programador realiza comparações entre *chunks* conhecidos e código. A comparação pode requerer raciocínio superficial ou profundo (SOLOWAY;EHRLICH,1984). No primeiro caso, programadores reconhecem planos familiares, tirando vantagem do seu conhecimento prévio de *chunks de programação*. Essas estruturas, quando reconhecidas, podem dar dicas sobre o que o programa faz. No segundo caso, programadores realizam estudos e comparações sobre relações causais entre elementos ou funções.

Chunking - Compreende à estratégia de criação de novas estruturas de níveis de abstração mais alto a partir de agrupamentos de código. Soloway e Ehrlich (1984) constataram, através de evidências empíricas, que programadores experientes usam tal estratégia formando *chunks* ou planos de programação. Planos (*chunks*) de nível mais baixo representam tarefas simples, bem definidas como **troca-de-valores**, **soma-acumulada**, as quais podem ser agregadas em planos (*chunks*) maiores e mais abstratos capazes de alcançar metas num nível mais alto. Exemplos de tais planos: plano **valor-máximo** capaz de encontrar o valor máximo de uma seqüência fornecida pelo usuário; plano **média** capaz de calcular a média de uma seqüência. Neste último caso o plano **soma-acumulada** está embutido no plano **média**. Soloway e Ehrlich (1984) consideraram essas observações como uma evidência empírica convincente de que há uma forte correlação entre **programação** e **planejamento**. Para estes pesquisadores, *chunks de programação* podem ser vistos como planos. Assim, *chunks* mais simples são vistos como sub-planos que são componentes de planos maiores. Os planos conhecidos por um programador para resolver certas (sub)metas são reusados quando essas (sub)metas aparecem em outros contextos (SOLOWAY apud WELTY,1995).

Referência-cruzada - Relaciona os elementos dos diferentes níveis de abstração. Por exemplo, uma meta mapeada sobre um plano e este por sua vez mapeado sobre partes do código do programa. Referencia cruzada é assim uma parte integral na construção da representação completa do modelo mental interligando todos os níveis de abstração.

2.2 Modelo do Processo de Construção de Programas

Os processos de construção e de compreensão de programas possuem muitas semelhanças no que diz respeito ao conhecimento cognitivo criado pelo programador em ambas as atividades (VON MAYRHAUSER;VANS,1994). O processo de construção de programas consiste da construção de mapeamentos a partir de uma atividade de alto nível, passando por níveis de abstração intermediários até o nível da implementação. Nesses diversos níveis, elementos são criados. A pesquisa em compreensão de programas define o processo de compreensão de programa como o de recriar tais elementos bem como o conjunto de mapeamentos que foram usados para desenvolver o programa (SOLOWAY Apud WELTY,1995). A próxima seção descreve as habilidades cognitivas usadas por programadores experientes durante a construção de programas.

2.2.1 Habilidades Cognitivas de Programadores Experientes

A fim de elaborar um programa, o programador necessita criar vários elementos para compor seu modelo mental. Inicialmente, necessita de um modelo do problema definido aqui como a descrição de problema na forma de metas a serem alcançadas. Esse modelo é criado durante o entendimento do que o programa deve fazer. Segundo, necessita criar planos de programação que alcancem as metas. Especialistas possuem uma biblioteca de tais estruturas à sua disposição, desenvolvida através da prática (DU BOULAY,1989). Terceiro, necessita refinar o modelo definindo sub-metas e selecionando sub-planos a fim de finalizar o modelo mental. Ryan e Al-Qaimari (1996) resumem as principais habilidades que programadores experientes desenvolveram e utilizam ao realizar a atividade de programação:

Abstração - Programar depende fortemente de um processo cognitivo de abstração. A inerente complexidade, mesmo de pequenos programas, requer que o programador faça abstrações constantemente durante sua construção;

Representação do Conhecimento - Para que conhecimento seja eficientemente armazenado, modificado e recuperado, ele deve ser representado internamente de uma maneira altamente organizada. Está claro, a partir de estudos empíricos, que programadores usam seu conhecimento de programação e seu conhecimento do domínio para guiar seu projeto de software, o que indica que programação envolve um processo de representação de conhecimento (WELTY,1995).

Mapas Mentais – Mapeamento é necessário para relacionar os elementos dos vários níveis de abstração. O processo de mapeamento mental apresenta uma sobrecarga cognitiva significativa e assim seus efeitos devem ser considerados em qualquer estudo cognitivo.

Simulações Mentais - Correspondem a estratégias mentais da execução de programas.

Estudos empíricos da Psicologia da Programação realizados por Soloway & Ehrlich (1984) e Bonar e Soloway (1983) mostraram que programadores experientes possuem a habilidade de resolver problemas, identificando **metas do problema** e criando **planos mentais de programação**, para somente depois, traduzí-los num algoritmo executável usando alguma linguagem de programação. A figura 8 apresenta uma proposta de representação do processo de construção de programas por especialistas, baseada nos elementos e estratégias descobertos pela pesquisa em compreensão de programas.

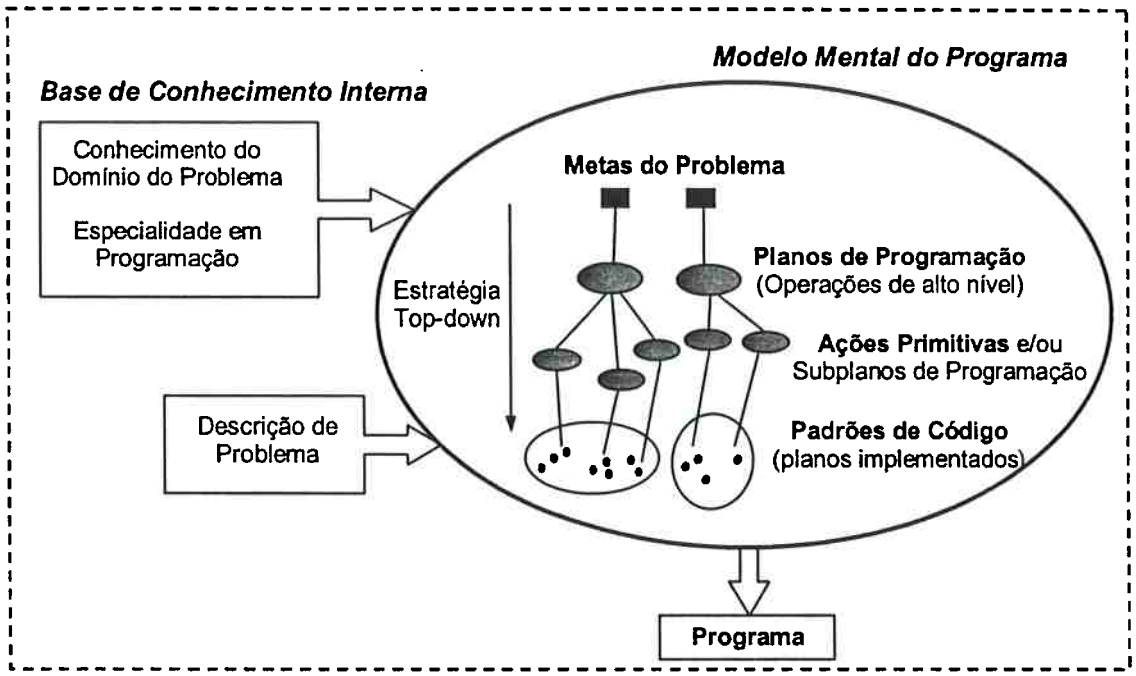


Figura 8 – Modelo do Processo de Construção de Programas por Programadores Experientes (adaptado de Mayrhauser e Vans (1994))

Nesse modelo, programadores experientes utilizam sua **base de conhecimento interna**, adquirida ao longo da sua experiência em programação. Essa base contém *chunks* de diversos níveis de abstração, os quais são selecionados segundo as metas e sub-metas que o programador deseja alcançar. As relações entre os *chunks* são conhecidas e o conhecimento encontra-se hierarquicamente organizado. A identificação de metas e sub-metas numa descrição textual de problema é também uma habilidade de programadores experientes, adquirida com a prática e apoiada pelo conhecimento da funcionalidade dos *chunks* de programação. A cada *chunk* incorporado no programa, o programador o preenche com dados do domínio do problema, além de encaixá-lo no código já existente, formando a lógica do programa.

A estratégia mais amplamente utilizada na construção de programas é a **estratégia *top-down***, onde o programador parte das metas em direção à codificação, passando por componentes intermediários tais como *chunks* de programação. Essa estratégia é realizada diversas vezes, de acordo com as metas e sub-metas a serem alcançadas. Wallingford (2002) complementa que o uso dessas estruturas e estratégias pelo programador experiente acontece, muitas vezes, de forma subconsciente.

Obviamente estratégias adicionais são utilizadas para obter o programa correto e completo. Uma delas é a inspeção dinâmica do comportamento do programa, por exemplo simulações mentais ou depuração no compilador. Consultas a programas

prontos e desconhecidos podem ainda auxiliar na expansão do conhecimento cognitivo do programador experiente.

Observando a figura 8, nota-se que a dificuldade para aprendizes em utilizar tal modelo, é a pouca ou nenhuma **base de conhecimento** adquirida. Na maioria das vezes, o único conhecimento disponível é a sintaxe da linguagem. Considerando que aprendizes não possuem o mesmo conhecimento que programadores experientes, a idéia nesta tese consiste em propor um modelo onde uma **biblioteca de conhecimento cognitivo em programação** está disponível para o aprendiz. Nessa proposta, os componentes da biblioteca possuem uma organização interna e o acesso a um certo conjunto de componentes é habilitado ao aprendiz em momentos diferentes durante o processo de construção do programa. Assim, as habilidades cognitivas do aprendiz são expandidas pela disponibilidade de fontes de conhecimento cognitivo externo.

2.3 Resumo do Capítulo

Este capítulo resume os principais conceitos que descrevem o comportamento de programadores experientes durante a atividade de compreensão e construção de programas, apontando algumas razões que dificultam a realização dessas atividades por aprendizes. Fornece ainda a definição dos termos tipicamente usados para descrever estruturas de conhecimento cognitivas e estratégias de raciocínio utilizadas pelos programadores ao realizarem a atividade de programação.

De um modo geral, os modelos dos processos de compreensão e construção de programas, por programadores experientes, possuem elementos em comum: (1) uma representação interna criada pelo programador: o **modelo mental do programa**; (2) uma **base de conhecimento** que corresponde ao conhecimento prévio do programador; (3) uma representação externa do programa que corresponde ao **código do programa** e possíveis documentações; (4) uma ou mais **estratégias** para combinar conhecimento prévio com o código do programa a ser construído ou compreendido, resultando no modelo mental do programa.

O processo de compreensão de programas é visto como o de recriar os elementos bem como o conjunto de mapeamentos que foram usados para desenvolver o programa (SOLOWAY apud WELTY,1995). Assim, muitas das estruturas e estratégias identificadas em atividades de compreensão de programas são atribuídas, por pesquisadores, à atividade de construção de programas.

A observação da atividade de compreensão é mais favorecida do que a de construção, uma vez que o programador atua sobre um programa já construído. Com relação ao processo de construção de programas, a pesquisa aponta que programadores experientes possuem a habilidade de resolver problemas, usando seu conhecimento prévio para identificar **metas do problema** e buscar por **planos de programação** que as implemente.

Nota-se ainda que as propostas de modelos de compreensão mais recentes (VON MAYRHAUSER;VANS,1994) são expansões ou agrupamentos de modelos mais antigos tais como os de Brooks (1983) e Soloway e Ehrlich (1984). De acordo com Storey at al. (1999) uma das justificativas das diferenças entre os modelos é devido aos diferentes fatores adotados nos experimentos tais como: tamanho e complexidade do programa, linguagem de programação utilizada, domínios de problemas diferentes, diferenças individuais do entendedor. Neste último caso, podem existir diferentes graus de conhecimento sobre o domínio da aplicação, o domínio da programação ou mesmo a criatividade do entendedor.

3 PADRÕES PEDAGÓGICOS PARA CIÊNCIA DA COMPUTAÇÃO

Uma vez que, neste trabalho, deseja-se disponibilizar componentes cognitivos para aprendizes enquanto executam a atividade de construção de programas, planos ou *chunks* de programação são elementos de interesse nesta pesquisa. Embora esses termos sejam muito citados nas áreas da Psicologia da Programação e de Sistemas Tutores Inteligentes para Programação, não constam trabalhos de modelagem desses elementos nessas áreas de pesquisa. O uso de planos de programação surgiu, inicialmente, no desenvolvimento dos primeiros Sistemas Tutores Inteligentes (BONAR; CUNNINGHAM,1988;JOHNSON;SOLOWAY,1985), porém, por terem sido usados como conhecimento interno ao sistema e não para uso explícito pelo aprendiz, não constam especificações de tais planos na literatura. Uma comunidade nova, interessada em documentar conhecimento especialista em programação, é a comunidade de projeto de **padrões pedagógicos para programação** (BERGIN,2001;ECKSTEIN ET AL.,2001; WALLINGFORD,1998;ASTRACHAN;WALLINGFORD,1998). Pesquisadores dessa recente área de pesquisa propõem uma forma de documentar tal conhecimento sob a denominação de **padrões de programação**.

Este capítulo descreve o estado-da-arte em padrões pedagógicos para programação, com a intenção de investigar uma maneira de formalizar planos de programação para uso explícito em ambientes de aprendizagem que enfatizam atividades cognitivas de programação.

Quando um programador se torna um especialista, seu conhecimento de programação se torna independente da sintaxe de linguagens de programação. Um especialista usa um vocabulário de alto nível para descrever um problema e sua solução. Esse vocabulário é formado por sentenças tais como **laço de busca**, **ação alternativa**, **escolha seqüencial** e permitem que o especialista pense sobre o planejamento do programa e represente uma solução num alto nível de abstração. As sentenças estão associadas a construções de código, denominados **idiomas** (*idioms*) na comunidade de padrões para programação, que o programador especialista conhece. O aprendizado dessas novas construções, assim como sua relação com termos de alto nível, são passos importantes para novatos se tornarem especialistas (ANDERSON, 1995).

Padrões de programação definem conceitos de alto nível relacionados com construções de código. E ainda, descrevem a aplicabilidade da construção de código num contexto.

Padrões, de um modo geral, tentam encapsular conhecimento especialista numa forma acessível e utilizável. Bergin (2004) define padrões como "A pattern is a solution to a problem in a context".

No domínio da programação, padrões são como blocos de construção para projeto e construção de software (COAD,1992). A comunidade de projeto de padrões os utiliza como um meio para armazenar conhecimento de solução de problema de um especialista num domínio. Estendendo a definição, um padrão é uma tentativa de estabelecer instruções sobre as possíveis soluções de um problema ou de uma classe de problemas (BERGIN,2004).

3.1 Origem dos Padrões Pedagógicos

Padrões pedagógicos vão além dos domínios da Ciência da Computação. O primeiro conjunto de padrões apareceu como parte de um livro chamado "*A Pattern Language – Towns, Buildings, Constructions*" do arquiteto Christopher Alexander et al.[5] (1977 apud PP, 2001). Alexander introduziu 253 padrões no domínio da arquitetura. Ele apresenta padrões para projetar cidades, prédios, salas com o objetivo de fornecer uma maneira consistente de criar um ambiente confortável para as pessoas morarem. Relacionando esses padrões com um espaço de problemas, o domínio, ele transforma essa coleção numa **linguagem de padrões** para esse domínio.

No início dos anos 90, a comunidade de software começou a capturar e comunicar conhecimento especializado no desenvolvimento de software, usando a técnica de Alexander. O movimento se iniciou na OOPSLA (*Object-Oriented Programming, Systems, Languages and Applications*), a maior conferência sobre programação orientada a objetos. Um primeiro livro publicado foi "*Design Patterns*" de Gamma et al.[6] (1995) apud Marinho (2003), o qual apresentou um catálogo de 23 padrões sobre como projetar sistemas de software orientado a objetos. Nessa época surgiu um conjunto de conferências próprias: PLOP, EuroPLOP, ChiliPLOP and KoalaPLOP (PLOP - *Pattern Languages of Programs*). Recentemente, o escopo das linguagens de padrões se expandiu incluindo os chamados padrões pedagógicos, os quais tentam capturar conhecimento especialista da prática de ensinar e aprender (ECKSTEIN ET AL.,2001).

---

[5] CHRISTOPHER, A. A Pattern Language: Towns - Buildings - Construction. Oxford University Press. 1977.

[6] Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Element of Reusable Object-Oriented Software.* Reading, Massachusetts: Addison-Wesley, 1995.

A maioria dos professores no mundo da tecnologia da informação não possui formação pedagógica. Em geral, o ensino de nível superior prioriza o conhecimento do assunto a despeito das habilidades pedagógicas. Conhecer o assunto é importante, mas não suficiente. Apesar do esforço em desempenhar da melhor maneira seu papel de educador, muitas pessoas ainda lutam com o problema de como ensinar, com sucesso, assuntos complexos. Alguns desses educadores são conscientes de suas próprias faltas de habilidades pedagógicas e tentam estratégias diferentes no ensino de um assunto específico. Fazendo isso, eles podem descobrir padrões pedagógicos de sucesso (ECKSTEIN;VOELTER, 2001).

O esforço internacional em escrever e coletar padrões para projeto de software, iniciou-se na comunidade de Tecnologia Orientada a Objetos. Foram duas as razões principais dessa iniciativa: (1) havia a necessidade de se documentar técnicas de ensino de sucesso porque os conceitos eram muito complexos e muitos treinadores/educadores tinham pouca ou nenhuma habilidade pedagógica; (2) a abordagem de padrões estava começando a ser amplamente usada para documentar tópicos técnicos no mundo OOT (*Object Oriented Technology*). Padrões provaram ser uma boa maneira de documentar conhecimento pedagógico. Hoje, pessoas interessadas em padrões pedagógicos ainda pertencem em sua maioria à comunidade OOT, mas há tendências para ampliar o projeto para todos os tipos de professores, educadores e treinadores (BERGIN,2004).

Desde o final da década de 90, acadêmicos da Ciência da Computação começaram a explorar o uso de padrões em seus cursos. Wallingford (1998) lembra que essa exploração foi uma continuação natural de trabalhos anteriores originários principalmente da Psicologia da Programação tais como os de Soloway e Ehrlich (1984); Johnson e Soloway (1985); Bonar e Cunningham (1988); Linn e Clancey (1992) e outros. Um exemplo de uso de padrões pedagógicos em cursos de programação é *Roundabout* de Wallingford (1997), um conjunto de padrões usados para ensinar a escrever programas recursivos numa linguagem funcional.

Wallingford (2002) define um padrão pedagógico para programação, em sua forma mais simples, como uma regra constituída de três partes:
(1) Um contexto que descreve um conjunto de situações, nas quais o padrão se aplica;
(2) Um problema que é descrito por um conjunto de requisitos e metas a serem alcançadas dentro do contexto;
(3) Uma solução para o problema que constitui-se de um texto explicativo de como obter uma configuração de software que resolve o problema. Um padrão explica ainda porque essa solução é suficiente e descreve como implementar a solução, fornecendo exemplos.

Dependendo da complexidade do padrão, as dificuldades que envolvem o contexto e a busca pela solução, são também relatadas no padrão. Meszaros e Doble (2000) descrevem as relações entre as partes de um padrão e as relações com o usuário, como mostra a figura 9. Por ser um documento de texto, as partes mencionadas nem sempre estão claramente separadas. Descrições de contexto, problemas e dificuldades podem aparecer mescladas no padrão.

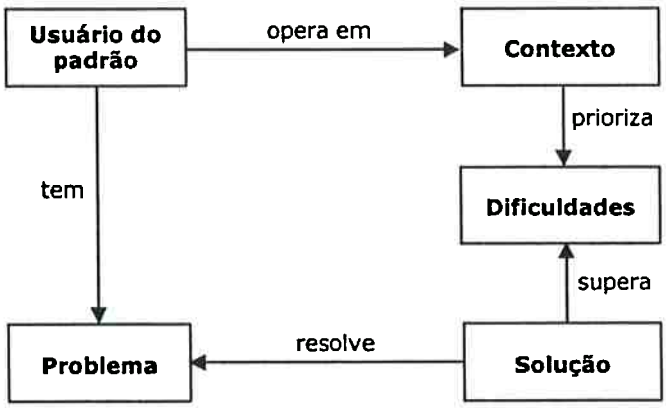


Figura 9 – Relações que envolvem um padrão pedagógico (MESZAROS;DOBLE,2000)

Lendo o contexto de padrões, o usuário pode reconhecer se ele está numa situação relatada num certo contexto. Porém, nem sempre o usuário tem a clara noção das dificuldades a serem enfrentadas em busca da solução para o seu problema. O padrão torna explícita as dificuldades e fornece uma solução que resolve o problema e supera, da melhor maneira, tais dificuldades.

3.2 Classificação de Padrões para Desenvolvimento de Software

Riehle e Züllighoven (1996) observaram que o termo *design pattern* é freqüentemente usado para se referir a qualquer padrão que enderece (i) **temas de arquitetura de software**, (ii) **temas de projeto de software** ou (iii) **temas de implementação de software**. Mas há uma importante distinção entre esses três níveis conceituais. A diferença entre esses três tipos de padrões está nos seus níveis de abstração e detalhes. Appleton (2000) apresenta uma classificação aceita na área, a qual classifica padrões segundo três tipos:

(i) **Padrões de Arquitetura** descrevem estratégias de alto nível que se relacionam com componentes em grande escala, mecanismos e propriedades globais de um sistema de software. Padrões de arquitetura podem ser vistos em Meszaros (1997), Lalanda (1997), Kerth e Whitenack apud Riehle e Züllighoven (1996), Buschmann et. al apud Marinho et al. (2003);

(ii) **Padrões de Projeto** descrevem estratégias que definem estrutura e comportamento de entidades e seus relacionamentos dentro do sistema estabelecendo micro-arquiteturas de subsistemas e componentes, sem afetar a estrutura do global do sistema de software. Exemplos de padrões de projeto de software podem ser vistos no trabalho de Gamma et al. (1995);

(iii) **Padrões de Programação** estão relacionados com técnicas de programação especificadas pela linguagem de programação sendo, portanto, descritos por meio de construções da linguagem. Esse tipo de padrão descreve como implementar aspectos particulares de componentes ou relacionamentos entre eles. Padrões de programação são conhecidos também pelo termo **Idiomas** (*Idioms*). Exemplos de padrões de programação podem ser vistos nos trabalhos de Coplien (1992) apud Riehle e Züllighoven (1996); Astrachan & Wallingford (1998) e Bergin (1999).

A maioria dos padrões publicados na literatura documentam técnicas avançadas em arquitetura, projeto e implementação de software. Sem dúvida, eles fornecem um vocabulário de projeto poderoso e facilitam o desenvolvimento de software complexo. Porém, esses padrões pressupõem um alto nível de maturidade e experiência do programador. Programadores novatos não conseguem tirar proveito desses padrões uma vez que não possuem o nível de competência necessário em programação. Frente a isso, padrões de programação mais simples, denominados **padrões elementares de programação**, têm surgido nos últimos anos e são pretendidos para estudantes novatos no domínio da programação. Wallingford (1998) cita vários aspectos do ensino da programação, para os quais padrões elementares de programação são úteis:

- princípios de projeto de software,
- escrita de programas,
- avaliação de programas,
- leitura e compreensão de programas,
- vocabulário para comunicação entre professor e estudante.

Em resumo, o propósito dos padrões elementares de programação é o de ajudar educadores e estudantes da Ciência da Computação no aprendizado da arte de programar. Eles consistem de um meio para comunicar conhecimento, metodologia e cultura de Ciência da Computação para aprendizes, sendo descritos num nível apropriado para estudantes novatos. Além de desempenhar um importante papel no aprendizado de programação, servem também como fundamento para padrões mais avançados no mundo do software (Wallingford,1998).

Acredita-se que instrutores novatos alcançam mais sucesso adotando padrões elementares de programação em sala de aula. Na verdade, instrutores são as pessoas que podem fazer o melhor uso de tais padrões em seus cursos introdutórios de programação. Neste trabalho, o interesse é sobre os padrões elementares de programação, os quais condensam conhecimento sobre domínios específicos da programação.

3.3 Padrões Elementares para Programação

Na construção de um sistema de software, dois modelos são integrados: modelo do domínio da aplicação e o modelo de projeto de software. Isso resulta num terceiro modelo: o modelo de implementação. Linguagens de programação fornecem a notação para esse último modelo. É nesse nível que os padrões elementares de programação estão situados (RIEHLE;ZÜLLIGHOVEN,1996). Eles capturam conhecimento que programadores especialistas têm, e usam subconscientemente, para implementar programas (WALLINGFORD,2002). Eles são baseados sobre experiência em programação e variam dependendo do paradigma e linguagem de programação. Um exemplo de um padrão elementar de programação de Bergin (1999) é dado na tabela 2.

Tabela 2 – Exemplo de padrão elementar de programação de Bergin (1999)

Nome	Escolha Seqüencial
Contexto / Problema	Você está numa situação em que precisa escolher exatamente uma de várias ações possíveis, mas essa ação não depende do valor de uma única expressão. Em vez disso, cada ação depende de uma condição testável separadamente. Após uma condição ser verificada verdadeira, você quer executar a ação associada e então finalizar. Você deseja um *layout* agradável para ambos: olhos e mente. Você quer uma estrutura que seja fácil de ler e entender.
Solução	Escreva uma seqüência de IF's, na qual cada IF tem uma parte ELSE. Cada ELSE, exceto o último, contém o outro IF integralmente.

Exemplo	`int participants = myParty.size();`
	`if (participants > 15000)`
	`{ rentTheSuperdome();`
	`}`
	`else if (participants > 1500)`
	`{ rentTheCivicCenter();`
	`}`
	`else if (participants > 150)`
	`{ rentATent();`
	`}`
	`else`
	`{ rentAMovie();`
	`}`

Basicamente, padrões elementares de programação podem ser considerados como ferramentas para ler e escrever programas. Segundo Wallingford (1998) os padrões contêm conhecimento para guiar o projeto de um programa, conduzindo às respostas das seguintes perguntas:

- Qual é o problema a ser resolvido?
- Quais as dificuldades que envolvem a resolução do problema ?
- Qual é a solução?
- Porque é melhor essa solução em vez das soluções alternativas?
- Como integrar construções de código mencionadas nos padrões, para formarem um programa?

Nos últimos anos, a comunidade de padrões e a comunidade de educação em Ciência da Computação têm demonstrado interesse na idéia de construir e usar padrões elementares de programação. Vários trabalhos têm sido publicados sobre o assunto sendo, em sua maioria, padrões elementares para ensino de programação orientada a objetos. Constam ainda alguns poucos trabalhos em outros paradigmas de programação. A tabela 3 resume alguns exemplos desses trabalhos.

Tabela 3 - Padrões Elementares para Programação

Domínio da Programação	Autores
Padrões de seleção	(BERGIN, 1999)
Padrões para laços	(ASTRACHAN;WALLINGFORD, 1998)
Padrões para programação funcional	(WALLINGFORD, 2002)
Padrões para programação recursiva	(WALLINGFORD, 1997)
Padrões para programação orientada a objetos	(KUEHNE, 1999)
Padrões para programas Prolog	(HANMER, 1996)
Padrões para construção e uso de variáveis locais em OOP (construtores e métodos para objetos)	(BRADY, 2000)
Padrões para auxiliar programadores JAVA selecionarem objetos do tipo *collections*[7] para projetos	(SANDU, 2001)
Padrões para construir e usar estruturas de dados indexadas	(BRADY, 2001)

Brady (2001) apresenta padrões elementares de programação para construir e usar estruturas de dados indexadas linearmente, tais como vetores e matrizes. Um exemplo resumido de um dos padrões pode ser visto na tabela 4. Outros padrões propostos por Brady (2001) são: *Appended Item* (adiciona um elemento), *Indexed Random Access* (acessa um elemento arbitrariamente), *Linear Indexed Traversal* (caminha por todos os elementos), *Reverse Linear Indexed Traversal* (caminha por todos os elementos em ordem reversa).

[7] *Collections* fornecem estruturas de dados e comportamento para armazenar, recuperar e manipular elementos similares. Exemplos de *colletions* são: array, vetor, hashtable.

Tabela 4 – Padrão para criar estruturas de dados indexadas

Nome	Estrutura de Dado Indexada Linearmente (Tamanho fixo)
Contexto/ Problema	Você necessita criar uma estrutura para armazenar uma coleção de elementos. Os elementos são de mesmo tipo e você sabe quantos elementos devem ser armazenados. Além disso, uma das seguintes condições é verdade: • você quer acessar elementos, arbitrariamente, numa coleção. • a ordem dos elementos na coleção não é importante. • a ordem é importante e elementos serão adicionados em ordem na coleção. • elementos não serão, necessariamente inseridos em ordem, mas devem ser rearranjados numa certa ordem antes de qualquer processamento.
Solução	Especifique o tipo e o número de itens na construção da estrutura. Para maiores detalhes sobre declaração de variáveis em geral, consulte o padrão "*Declare-Construct-Initialize*". Este padrão pode não ser a melhor escolha se: • A ordem dos elementos na coleção é importante mas elementos não serão adicionados na coleção em ordem e sua localização na ordem final depende de outros objetos na coleção. Nesse caso, uma lista ligada (*Linear Linked Data Structure*) ou uma árvore de busca binária (*Binary Search Tree*) pode ser uma escolha melhor. • Você sempre acessará os elementos na coleção em ordem (do primeiro ao último ou do último para o primeiro). Nesse caso, uma fila (*Queue*) ou uma pilha (*Stack*) pode ser uma escolha melhor.

Exemplos	Em C e C++, o tamanho para um *array*, definido estaticamente, deve ser uma expressão constante conhecida em tempo de compilação. A sintaxe para criar *arrays* de tipo T é:
	`T myArray[const_expr];` `// C ou C++`
	`T[ ] myArray = new T[nbrItems];` `// Java`

Astrachan e Wallingford, (1998) propõem padrões para laços, como por exemplo, laços para processamento de itens numa coleção. A tabela 5 apresenta um padrão que explica como implementar código para encontrar valores extremos numa coleção.

Tabela 5 – Padrão para encontrar valores extremos numa coleção

Nome	Valores Extremos
Contexto / Problema	Você está escrevendo código para encontrar valores extremos, por exemplo o máximo e o mínimo em uma coleção ou seqüência de valores.
Dificuldades	As principais dificuldades em escrever código para encontrar valores extremos é determinar o melhor tipo de laço e os valores iniciais para as variáveis que representam o valor extremo, tais como, min e max. Min deve representar o valor mínimo de todos os valores processados até o momento.
Solução	Você deve processar todos os itens da coleção para assegurar que os valores extremos são encontrados. Se o tamanho da coleção for conhecida a priori, use uma estrutura for (consulte o padrão "definite process all items"; senão use a estrutura while (consulte o padrão "iterator process all items"). Inicialize os valores extremos com o primeiro valor da coleção a partir do qual os valores extremos serão determinados. Se possível, use índices, ponteiros ou referências em vez dos valores dos objetos ou variáveis, isto é, mantenha o índice do valor mínimo corrente em um *array* em vez do próprio valor mínimo corrente. O valor pode ser determinado a partir do índice, ponteiro ou referência.

Exemplos	Encontre o maior e o menor valor num vetor de strings.
	
	`int minIndex = 0;  //index of minimum values between 0 and k`
	`int maxIndex = 0;  //index of maximum values between 0 and k`
	`int k;`
	`for(k=1; k < a.size(); k++)`
	`{`
	`if (a[k] < a[minIndex]) minIndex = k;`
	`if (a[k] > a[maxIndex]) maxIndex = k;`
	`}`

Padrões de seleção podem ser encontrados no trabalho de Bergin (1999). A tabela 6 apresenta um resumo desses padrões. Pode-se notar que esses trabalhos são relativamente recentes e, portanto, não são encontrados com facilidade.

Tabela 6 - Padrões elementares de programação para seleção (Bergin, 1999)

Nome	Contexto / Problema	Exemplo

se-ou-não	Alguma ação pode ou não ser apropriada, dependendo de uma condição a ser testada	`if (measuredHeat() > subBoilThreshold)`
		`{ shutDownGenerator();`
		`}`

ação alternativa	Uma de duas ações é apropriada dependendo de uma condição. Quando a condição é verdadeira deseja-se executar uma condição, quando falsa deseja-se executar uma ação diferente	`if ( numericGrade > 60 )`
		`{ output ("passing");`
		`}`
		`else`
		`{ output("failing");`
		`}`

expressão condicional	Existe a necessidade de calcular um valor para usá-lo numa expressão ou atribuição. O valor depende de uma única condição.	`int temp;`
		`if(emp.dueBonus())`
		`temp = dept.standardBonus();`
		`else`
		`temp = 0;`
		`salary = emp.basicSalary() + temp;`

escolha-sequencial	Existe a necessidade de escolher uma entre várias ações possíveis, mas cada ação possível depende de uma condição testada separadamente.	`int participants = myParty.size();`
		`if (participants > 15000)`
		`{ rentTheSuperdome();`
		`}`
		`else if (participants > 1500)`
		`{ rentTheCivicCenter();`
		`}`
		`else if (participants > 150)`
		`{ rentATent();`
		`}`
		`else`
		`{ rentAMovie();`
		`}`

escolha não relacionada	Existem várias ações e várias condições. Pode ser desejável executar várias das ações se as condições associadas forem verdadeiras. Cada ação tem uma condição que determina se ela deve ser executada.	`if (roofIsLeaking())`
		`{ cal'Roofer();`
		`}`
		`if (sinkIsLeaking())`
		`{ callPlumber();`
		`}`
		`if (floorIsLeaking())`
		`{ callExcavator();`
		`}`

escolha independente	Uma, entre duas ações, deve ser escolhida. Mas a escolha de cada uma das ações dependem ainda de vários fatores. Os fatores são independentes.	`if (a.isRectangle())`
		`{ if(a.isFilled())`
		`{ hatchBox();`
		`}`
		`else`
		`{ openBox();`
		`}`
		`}`
		`else`
		`{ if(a.isFilled ())`
		`{ EasterEgg();`
		`}`
		`else`
		`{ justABall();`
		`}`
		`}`

Podemos observar, nos exemplos apresentados, que nem todos os padrões requerem os mesmos tipos de informação. Para um padrão ser verdadeiramente útil, ele deve ter um conjunto mínimo de informações essenciais (MESZAROS;DOBLE, 2000). O volume dessas informações fornece uma idéia da complexidade do padrão e do problema que ele resolve.

A chamada a outros padrões também indica uma complexidade relativa do padrão. Por exemplo, padrões de seleção não possuem nenhuma indicação de consulta a outros padrões, ao contrário do padrão "valores extremos". Esse último incorpora outro tipo de padrão, mais simples: o padrão "*Definite process all items*" ou o padrão "*Iterator process all item*".

A indicação de consulta é útil se o usuário necessita compreender padrões que são pré-requisitos para compreensão e utilização de um padrão mais complexo. Por exemplo, o padrão para criação de estruturas de dados indexadas linearmente, mostrado na tabela 4, indica a consulta ao padrão "*Declare-Construct-Initialize*", para compreensão de declaração de variáveis em geral.

3.4 Escrita de Padrões para Desenvolvimento de Software

A atividade de descoberta e escrita de bons padrões (*pattern mining*) é considerada uma tarefa difícil (APPLETON,2000). Essa afirmação é particularmente verdade para padrões de arquitetura e projeto. Padrões elementares de programação são mais simples, uma vez que fazem parte do conhecimento básico de programação e específico do paradigma e linguagem de programação. Padrões de arquitetura e de projeto descrevem estratégias de alto-nível que relacionam componentes em grande escala, propriedades e mecanismos globais de um sistema ou ainda estrutura e comportamento de entidades do sistema. Padrões elementares de programação preenchem detalhes da estrutura ou comportamento de um componente do sistema de software.

Appleton (2000) afirma que "The best way to learn how to recognize and document useful patterns is by learning from others who have done it well!". Meszaros e Doble (2000) propõem um padrão para escrever padrões (tabela 7), que resume os critérios de Buschmann et. al (1996)[8] apud Appleton (2000):

> **Focus on practicability:** Patterns should describe *proven solutions* to recurring problems rather than the latest scientific results.
>
> **Aggressive disregard of originality:** Pattern writers do *not* need to be the original inventor or discoverer of the solutions that they document.
>
> **Non-anonymous review:** Pattern submissions are *shepherded* rather than reviewed. The shepherd contacts the pattern author(s) and discusses with them how the patterns might be clarified or improved upon [....]

[8] Buschmann, F. et. al. Pattern-Oriented Software Architecture: A System of Patterns. Volume 1. Publisher: John Wiley & Sons, 1 edition, 476 p., 1996.

Assim, o foco para escrever padrões está na prática, uma vez que padrões devem descrever soluções provadas para problemas que se repetem, em vez de inovadores resultados científicos. A atividade de escrever padrões não está relacionada com originalidade. Escritores de padrões não necessitam ser inventores ou descobridores de novas soluções a serem documentadas e sim descobridores de soluções já utilizadas e provadas como boas soluções. Além disso, padrões devem ser submetidos a grupos de interesse para gerar um canal de comunicação que serve como um guia para melhorá-los, em vez de simplismente serem revisados.

Tabela 7 – Um padrão para escrever padrões (MESZAROS;DOBLE,2000)

Padrão	Padrão
Contexto	Você é um especialista em programação no campo. Você notou que tem usado uma certa solução para um problema que comumente ocorre. Você gostaria de compartilhar sua experiência com outros.
Problema	Como compartilhar com outros, uma solução para um problema que tem se repetido, para que esta solução possa ser reutilizada?
Solução	Escreva a solução usando o formato de um padrão. Capture ambos: o problema e a solução, assim como as razões pelas quais a solução é aplicável. Distribua o padrão resultante para uma audiência que você sinta que poderá ajudá-lo a melhorar o padrão.

A escrita de padrões pedagógicos para programação, particularmente padrões elementares de programação, levam em conta todos os aspectos mencionados acima. Porém, tais padrões são tipicamente escritos por professores experientes os quais embutem aspectos didáticos na especificação dos padrões, uma vez que são destinados para uso por programadores aprendizes. Padrões destinados a programadores experientes podem enfatizar, por exemplo, aspectos de otimização e compactação da solução exposta no padrão em detrimento da sua facilidade de compreensão.

3.5 Resumo do Capítulo

O principal objetivo dos padrões dentro da comunidade de software é criar um corpo de literatura para auxiliar desenvolvedores de software a resolverem problemas difíceis e comuns encontrados no desenvolvimento e engenharia de software (APPLETON,2000). Somente recentemente nasceu a idéia de escrever padrões elementares de programação para apoiar o ensino e aprendizado de programação para estudantes.

Segundo Appleton (2000), uma das primeiras necessidades de qualquer ciência ou disciplina de engenharia é um vocabulário para expressar seus conceitos e uma linguagem capaz de relacioná-los. Por muito tempo e ainda nos dias de hoje, a linguagem formal utilizada no ensino introdutório da programação algorítmica tem sido a própria linguagem de programação ou pseudocódigo ou ainda a linguagem de fluxogramas. Embora professores também utilizem, informalmente, um vocabulário mais abstrato ao ensinar construção de programas, a falta de padronização e documentação desse vocabulário, não permitem que aprendizes tirem proveito desse conhecimento.

O interesse recente que a área de Padrões Pedagógicos para Programação vem demonstrando para apoiar professores e aprendizes em programação, faz o elo entre o esforço realizado pela área da Psicologia da Programação nos anos 80 e 90, e o problema da difícil transformação de um aprendiz em um programador experiente. De um lado a Psicologia da Programação realizou uma série de estudos empíricos para estabelecer modelos dos processos de compreensão e construção de programas por programadores experientes e aprendizes, os quais permitiram detectar as razões das dificuldades existentes no aprendizado de programação. Do outro, a área de Padrões Pedagógicos vem propondo soluções para o problema através da modelagem de padrões elementares de programação.

Padrões elementares de programação definem conceitos de alto nível relacionando-os com construções de código, tal como *chunks* de programação da área da Psicologia da Programação, apresentada no Capítulo 2. Além disso, descrevem um contexto de aplicação do padrão nas soluções de problemas. Eles buscam tornar explícito o conhecimento que especialistas em programação possuem. Isto é feito de uma forma didática e considerando alguns princípios importantes sobre aquisição de habilidades, uma vez que padrões têm sido desenvolvidos por professores da área da Computação.

Bransford (1999) cita alguns desses princípios oriundos do estudo sobre pessoas que desenvolveram habilidades em algum domínio e suas implicações para aprendizado e instrução: (1) especialistas notam características e padrões significativos de informação que não são notadas por novatos; (2) especialistas adquirem grande conteúdo de conhecimento que está organizado em estruturas que refletem um entendimento profundo do assunto; (3) conhecimento especialista não pode ser reduzido a conjuntos de fatos isolados ou proposições, mas contextos de aplicabilidade, ou seja, o conhecimento é condicionado a um conjunto de circunstâncias.

Padrões estão classificados de acordo com a complexidade do software a ser desenvolvido. Padrões mais complexos fornecem um vocabulário de projeto poderoso e facilitam o desenvolvimento de software complexo. Tais padrões são destinados para uso por programadores experientes e são os mais facilmente encontrados na literatura. Padrões mais simples, chamados padrões elementares de programação, descrevem conhecimento básico em programação. Seu propósito é o de ajudar educadores e estudantes no aprendizado da arte de programar. Porém, essa é uma preocupação recente dentro de uma área também recente, a de Padrões Pedagógicos, sendo poucos os trabalhos existentes.

O próximo capítulo empresta as idéias das duas áreas estudadas nestes dois últimos capítulos (capítulos 2 e 3), a fim de propor um processo de construção de programas especialmente dirigido a aprendizes.

4 AMBIENTES DE APRENDIZAGEM EM PROGRAMAÇÃO

Fischer (2003) e Rocha (1995) dividem o ato de programar em três etapas: (i) planejamento, codificação e testes. "O planejamento compõe a compreensão do problema e a elaboração de uma solução. Esta solução pode ser inicialmente mental, mas já deve estar voltada a implementação, ou seja o novato deve aprender a expressar sua visão do problema como um programa." (FISCHER,2001). Neste trabalho o interesse está em identificar ambientes de aprendizagem para programação que propõem recursos para superar a dificuldade do aprendiz no nível do planejamento, ou seja, ambientes que forneçam apoio ao aprendiz para a construção do modelo mental do programa (MAYRHAUSER; VANS, 1994; SOLOWAY; EHRLICH, 1984).

4.1 Sistemas Tutores Inteligentes

Uma área que propõe soluções para ensino e aprendizagem em ferramentas computacionais é a área de Sistemas Tutores Inteligentes. Essa é uma especialização dentro do campo de Instrução Auxiliada por Computador (CAI - *Computer-Aided Instruction*) e que integra técnicas de diversas áreas como mostra a figura 10.

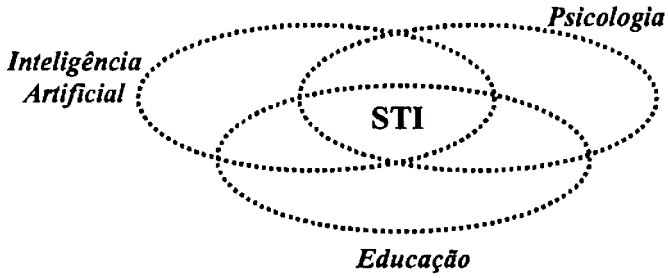


Figura 10 – Multidisciplinaridade dos Sistemas Tutores Inteligentes

Sistemas Tutores Inteligentes são ferramentas de software para guiar a aprendizagem de novatos em algum domínio e que são caracterizados por uma arquitetura típica em que o conhecimento a ser aprendido é representado (Módulo Especialista) e comunicado ao estudante através de uma interface apropriada (Interface do Estudante). E ainda, estratégias de monitoração e diagnóstico (Módulo Diagnóstico) das ações do estudante são implementadas com o objetivo de construir uma representação interna de seu desempenho (Modelo do Estudante). Assim, o sistema pode tomar decisões tutoriais individualizadas para cada estudante (Módulo Pedagógico). Sistemas tutores têm sua origem nos anos 70, porém somente na década de 80, os tutores apresentaram uma arquitetura rmais completa. Esta seção resume os trabalhos relevantes dessa área.

4.1.1 Basic Instructional Program

BIP (*Basic Instructional Program*) desenvolvido em 1976 por Barr et al.[9] apud Wenger (1987) consta como a primeira ferramenta para ensino de programação. Sua base de conhecimento consiste de uma rede que relaciona tópicos da ementa a serem ensinados com tarefas simples a serem executadas pelo aprendiz. O modelo do estudante é composto dos tópicos estudados pelo aluno. Cada tópico possui dois valores associados que indicam o nível adquirido pelo estudante, no tópico. O primeiro valor é atualizado conforme o desempenho do estudante na realização do exercício, usando uma técnica simples. O segundo valor é dado pelo estudante, através de uma pequena entrevista após a realização de cada tarefa, e corresponde a uma estimativa a respeito de seu próprio entendimento. Essas estratégias caracterizam uma combinação de diagnóstico inferencial e interativo, porém extremamente limitado pois o sistema não possue conhecimento de projeto, codificação ou depuração. Por ser incapaz de diagnosticar erros do estudante, o sistema não consegue fornecer realimentação para o estudante.

4.1.2 Meno-II

Outros sistemas que caracterizaram a fase inicial do desenvolvimento de tutores foram LAURA (ADAM;LAURENT,1980), um sistema projetado para apoiar depuração de programas de estudantes e MENO-II de Soloway et al.[10] (1983) apud Wenger (1987). MENO-II é um sistema de diagnóstico especializado na análise de laços em programas PASCAL. Nesse tutor, o estudante escreve seu programa na forma de código e o sistema divide-o transformando-o numa árvore. O objetivo é dar uma forma à solução do estudante a fim de compará-la (*matching*) com uma solução simples pré-armazenada no sistema. Isto é feito com a ajuda de conhecimento especializado sobre tipos de laços e ainda uma biblioteca de erros conhecidos. Quando testado no contexto de um curso introdutório real, MENO-II falha para diagnosticar corretamente uma grande variedade de evidências de erros (*bugs*) em programas de aprendizes. Seu esquema de *matching* é simples, o qual ignora o processo de desenvolvimento do programa.

4.1.3 PROUST

A partir da década de 80, houve uma preocupação em se modelar o conhecimento de programação, além de conceitos isolados da linguagem. Pesquisas sobre como programadores experientes e aprendizes modelam, mentalmente, a solução de problema

---

[9] Barr, A. et al. The computer as tutorial laboratory: the Stanford BIP project. *International Journal on Man-Machine Studies, 8,* 5, 567-596, 1976.

[10] Soloway et al. MENO-II: Na AI-based programming tutor. J. Comput.-Based Instruction 10 (1), 1983.

foram realizados pela área da Psicologia da Programação. O foco foi projetar tutores que realizassem a tarefa de diagnóstico com maior sucesso. O sistema mais representativo, dessa época, foi o sistema PROUST (JOHNSON;SOLOWAY,1985;JOHNSON,1990). Esse sistema realiza diagnóstico em programas PASCAL que envolvem as estruturas WHILE e IF. PROUST reporta, ao estudante, o local provável do erro no código escrito pelo estudante. Esse local corresponde a um trecho do programa o qual é responsável por uma sub-tarefa de programação. Porém, tal sub-tarefa é uma suposição do sistema a respeito do processo de raciocínio do estudante. A abordagem de diagnóstico utilizada foi denominada pelos autores de **diagnóstico baseado em intenções**, também conhecido como **reconhecimento de plano** ou **diagnóstico reconstrutivo**, pois realiza em paralelo: (i) a construção interna do programa, denominada reconstrução das intenções do programador e (ii) o diagnóstico.

Para inferir tal processo, PROUST usa uma base de conhecimento formada por metas, planos de programação, e erros comuns associados a eles. Com esse conhecimento, o sistema tenta construir uma interpretação para o programa em análise. PROUST começa a diagnosticar um programa do estudante selecionando uma meta (interna ao sistema) a partir da descrição de problema. Em seguida, recupera da base de conhecimento um conjunto de planos que implementam essa meta e compara cada um deles com o código. Esse processo é recursivo, uma vez que um plano pode ter submetas. Wenger (1987) conclui que a ampla variabilidade de programas de aprendizes e da possibilidade de erros torna o espaço de interpretações muito grande mesmo para programas relativamente simples.

As decomposições de metas propostas por PROUST, normalmente, não casam com o código do estudante. PROUST possui conhecimento heurístico sobre diferenças de planos na forma de regras (*plan-difference rules*). O catálogo de regras de diferenças de planos foi construído como resultado de uma análise de numerosos programas de aprendizes, em PASCAL.

A abordagem utilizada em PROUST torna o sistema de difícil expansão, uma vez que é baseado em regras de produção. A estratégia de diagnóstico é complexa e imprecisa, uma vez que o catálogo é incompleto e não há nenhum indicativo sobre o processo de raciocínio do estudante. Em configurações reais estudantes reportaram que as explicações de PROUST, a respeito das evidências de erros, eram difíceis de serem entendidas, sugerindo a necessidade de um módulo de *tutoring* ativo, isto é, com maior interatividade durante o processo de diagnóstico.

4.1.4 BRIDGE

O problema da distância entre a descrição de problema e a geração do código pelo estudante foram consideradas no sistema BRIDGE de Bonar e Cunningham (1988), projetado para aprender programação em linguagem Pascal. Esse sistema foi baseado nas teorias descobertas pela Psicologia da Programação, na época (BONAR;SOLOWAY, 1983). Em BRIDGE o estudante constrói planos de programação a partir de frases informais, descritas em quatro níveis de abstração. O sistema está embasado na idéia de que existem, incrementalmente, estágios detalhados na maneira que alguém pode definir um plano, movendo-se de uma simples declaração de problema em linguagem natural para um detalhado programa executável. A tabela 8 mostra alguns exemplos do vocabulário utilizado em cada estágio.

Tabela 8 – Níveis de abstração no tutor BRIDGE (WENGER,1987)

Estágio	*Representação de Conhecimento*	*Exemplos*
1	declarações não-procedurais	"computar a média de" "um conjunto de inteiros"
2	descrição de agregações	"somar todos os inteiros" "contá-los"
3	descrição de passos	"adicionar o próximo inteiro em" "somar até"
4	pseudo linguagem de programação	"repeat" "read an integer"

Em cada nível o estudante dispõe de um conjunto de frases informais a serem selecionadas por ele. O tutor acompanha os passos do estudante, realizando diagnóstico quando necessário, a fim do estudante conseguir passar para o nível seguinte. O tutor impede que o estudante passe para um próximo nível enquanto sua solução, no nível corrente, estiver errada. A transformação da solução obtida no quarto estágio (pseudocódigo) para um programa, ocorre pela seleção de expressões PASCAL para cada declaração em pseudocódigo.

BRIDGE é um dos únicos tutores que dispõe de representações definidas no nível acima da implementação para apoiar o processo mental do aprendiz, em programação algorítmica. Porém, o desenvolvimento de tais representações foram feitas sem qualquer formalismo e sem possibilidade de reuso, como propõe a recente área de pesquisa em Padrões Pedagógicos. Não consta, na literatura, as relações entre as descrições nos quatro níveis de abstração, as quais devem estar representadas internamente ao tutor a fim de realizar diagnóstico.

Essas relações também não ficam explícitas para o estudante em nenhum nível, uma vez que em BRIDGE, o objetivo é que o próprio estudante realize tais mapeamentos. Supondo uma possível expansão do currículo de programação do tutor, para aplicações reais, as listas de frases podem se tornar extensas, dificultando a tarefa do aprendiz.

4.1.5 LISP-Tutor

LISP-Tutor (ANDERSON;REISER,1985) é um sistema para ensinar linguagem de programação LISP com características instrucionistas que guiam estritamente os aprendizes. O tutor foi construído com o objetivo de testar a teoria de cognição ACT-R de Anderson (*Adaptive Control of Thought-Rational*). Na teoria ACT-R pensar é governado por sistemas de produção: (i) conhecimento declarativo é armazenado através de observação e instrução e; (ii) habilidade cognitiva é a habilidade de converter o conhecimento declarativo em regras de produção e combiná-las para formar conhecimento procedimental.

Em LISP-Tutor, a base de conhecimento contém dois modelos: (1) um **modelo ideal** de como estudantes deveriam resolver os problemas propostos pelo tutor; (2) um **modelo de falhas** típicas de estudantes considerando essa classe de problemas. Os modelos contém conhecimento codificado em regras de produção. O conhecimento declarativo necessário para resolução dos problemas não é fornecido pelo tutor. Assume-se que esse conhecimento é fornecido previamente através de aulas tradicionais ou livros.

O modelo de falhas é usado para reconhecer e reparar erros. O modelo ideal é usado para guiar estudantes ao longo de um caminho de solução correto, quando necessário. Esse dois modelos definem a chamada abordagem *model-tracing*. Nessa abordagem, o tutor acompanha o caminho de solução do estudante e interrompe-o sempre que o estudante se desvia do caminho correto. A partir de uma declaração de problema, o estudante inicia a codificação de seu programa. O sistema acompanha a inserção de cada declaração e interage com o estudante através de perguntas objetivas a serem respondidas com um sim/não ou através de múltipla escolha.

4.1.6 ELM-PE

O sistema ELM-PE (WEBER ET AL.,1996); (WEBER;MÖLLENBERG,1994) suporta aprendizado da linguagem LISP. Segundo o autor, o sistema pode ser usado tanto para aprendizes como para programadores experientes. Por ser um sistema que permite completa liberdade ao aprendiz, ele se aproxima muito de um ambiente tradicional de desenvolvimento. Sua vantagem é que possui facilidades como visualização da execução do programa e explicação de exemplos.

O sistema adota a abordagem *model-tracing* passivo para realizar diagnóstico. Assim, durante a construção de um programa, o sistema permite que estudantes cometam erros de programação; somente erros de sintaxe são apontados imediatamente. Um editor estruturado colabora com a redução de erros de sintaxe. A interface contém botões que chamam esqueletos de funções típicas em LISP, onde o estudante pode preenchê-las. Estudantes podem pedir ajuda ao sistema se não são capazes de encontrar um erro no código ou se desejam saber se sua solução está correta. O diagnóstico resulta numa explicação de como o código submetido poderia ter sido implementado pelo aprendiz.

O conhecimento especialista de ELM-PE consiste de conhecimento sobre linguagem de programação LISP, esquemas de algoritmos comuns e conhecimento de solução de problema (ex. recursão). Adicionalmente, há regras de evidência de falhas (*bugs*) descrevendo as causas (erros). A análise cognitiva do código do estudante emprega o método generalização baseada em explicação (*Explanation Based Generalization* - EBG) de Mitchell et al.[11] (1986) apud Weber e Möllenberg (1994). Uma vez que, a representação do conhecimento interno de ELM-PE é diferente da representação usada pelo estudante, para compor seu programa na interface do tutor, ELM-PE usa uma estratégia bastante complexa para diagnosticar erros na solução do aluno. A existência de esquemas de algoritmos comuns disponíveis tanto na interface como internamente ao sistema, minimizam os problemas encontrados no sistema PROUST. Uma evolução de ELM-PE é ELM-ART[12] (WEBER;SPECHT,1997), a versão de ELM-PE para Internet.

4.1.7 ADAPT

ADAPT (FIX;WIEDENBECK,1996) é um sistema idealizado para apoiar aprendizagem de uma segunda linguagem de programação, a linguagem ADA, por estudantes de níveis intermediário e avançado com conhecimentos em alguma linguagem procedimental tal como Pascal ou C. ADAPT auxilia estudantes escolherem planos para implementar tipos de dados abstratos. A estratégia utilizada é a de completar exercícios. No início da solução de problema, uma janela contém algumas palavras-chave e esqueletos (*templates*). Esqueletos contêm planos de alto-nível, os quais devem ser expandidos pelo estudante na criação da solução de problema. O estudante pode trocar um plano de alto-nivel por um plano mais detalhado. Nesse caso um menu com subplanos alternativos ficam disponíveis para escolha. Assim, planos são elaborados num refinamento passo-a-passo até que eles possam ser trocados por um pequeno segmento de código ADA, a ser escrito pelo estudante.

---

[11] Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. Explanation-based generalization: a unifying view. *Machine Learning, 1*, 47-80, 1986.

[12] ELM- ART: http://www.contrib.andrew.cmv.edu/~plb/home.html

A terminologia usada na linguagem de menus foi desenvolvida a partir de várias fontes: terminologia usada na cognição de programação para descrever planos, exemplos de materiais pedagógicos e de especialistas em ADA. Da avaliação do protótipo ADAPT algumas conclusões importantes foram obtidas:

(1) A abordagem de menus de planos foi usada previamente no tutor BRIDGE (BONAR;CUNNINGHAM,1988) para programadores novatos, mas sua aceitabilidade pelos estudantes não foi reportada. Assim, a aceitabilidade por programadores experientes foi tema de discussão no projeto ADAPT. Os estudantes sentiram-se incentivados com a disponibilidade dos planos e reportaram, em entrevistas, que não indicariam, para uma nova versão de ADAPT, a transformação de planos de alto nível diretamente em código. Assim, os autores concluíram que os estudantes não se sentiram restringidos ou desmotivados pelo número relativamente pequeno de planos e ausência de outros planos alternativos;

(2) As descrições em linguagem natural usada para representar os planos nos menus foram fonte de alguma dificuldade para os estudantes. Foram registrados indícios de que alguns estudantes tiveram dificuldades para compreender o significado dos planos, ou seja, o que o plano realizava. Dessa forma a seleção do plano ficava prejudicada. Os autores reportaram que a literatura é escassa em informações para descrever planos de programação e que esse foi o ponto mais fraco do protótipo. Consideram ainda que uma linguagem adequada para descrever planos de programação é de extrema importância para o sucesso pedagógico de uma ferramenta para ensino de programação.

4.1.8 C-Tutor

C-Tutor (HAHN AT AL.,1997) está baseado no tutor PROUST, onde ambos consideram as intenções do programador para reconhecer erros no programa do aprendiz. Assim como PROUST, C-Tutor não disponibiliza sua representação de conhecimento interna na interface para o aprendiz. Apesar de C-tutor possuir uma rica representação de conhecimento, o aprendiz dispõe apenas da linguagem de programação para construir seu programa. A base de conhecimento interna sobre programação é representada como grafos genéticos (GOLDSTEIN,1982) formados de metas e planos (grafos and/or). C-tutor difere de PROUST pela existência de GOES (*GOal Extraction System*) (HAHN,1995), um sistema de engenharia reversa.

Em C-Tutor, novos problemas são inseridos pelo professor na forma de código C e GOES extrai planos implementados a partir do código. Metas são extraídas a partir desses planos para construção da descrição de problema. Baseado nessa estrutura, o programa do estudante é reconhecido pelo sistema para diagnóstico, o que se assemelha a estratégia de PROUST.

A vantagem de C-Tutor é que o estudante parte das mesmas metas do sistema para construir seu programa, minimizando os problemas encontrados em PROUST. Porém, a partir das metas não há registro algum sobre qualquer passo cognitivo que o estudante tenha realizado durante a codificação.

4.2 Ambientes Baseados em Padrões de Programação

Padrões elementares de programação têm sido apresentados na literatura como uma proposta para apoiar ensino e aprendizado de programação. Porém, tais padrões são apresentados ao aluno como uma teoria a ser aprendida, uma vez que são descritos na forma de documentos do tipo texto. A preocupação em criar padrões de programação para ensino e aprendizado é muito recente, o que explica o fato de não ter sido encontrado, na literatura, ferramentas de aprendizagem baseadas em padrões elementares de programação. Nesta tese, os padrões elementares de programação formam a base para modelagem de planos de programação em uma ferramenta de autoria, a BibPC. Os planos são propostos para utilização com aprendizes em construção de programas, por meio do sistema TutorC.

Nesse contexto, alguns ambientes foram encontrados, embora sejam destinados a programadores experientes durante o ciclo de desenvolvimento de software. São os chamados **repositórios de padrões de programação** (DABOUS,2003) tais como os propostos por Kroth e Pfanfeseller (2001); Sun Microsystems (2004) e Microtool (2004). Repositórios têm sido propostos devido ao crescimento do número de padrões de software para análise, projeto e implementação e ainda, devido à necessidade de reutilização dos mesmos no desenvolvimento de sistemas. A função do repositório é facilitar a busca e aplicação dos padrões nas diferentes fases de desenvolvimento de software (MARINHO ET.AL, 2003).

Alguns desses repositórios estão sendo integrados em ferramentas *case*, tanto para acesso pelo desenvolvedor (DABOUS,2003), como para acesso pela própria ferramenta *case*, na geração de código automático (BUDINSKY,1996). No primeiro caso, a integração do padrão com o software que está sendo desenvolvido, não acontece automaticamente. O engenheiro de software é responsável por interpretar e adaptar o padrão de acordo com as suas necessidades (MARINHO,2003). Assim, tal tipo de recurso não é apropriado a um aprendiz. No segundo caso, padrões são utilizados para definir esqueletos do software, automaticamente.

Um exemplo, no qual o desenvolvedor interage com o repositório, é o trabalho proposto por Marinho (2003), o qual consiste na integração de um repositório de padrões de software com o *Rational Unified Process* (RUP), um modelo de processo de desenvolvimento de software difundido atualmente. O UMLStudio (PRAGSOFT,2004) é uma ferramenta comercial de modelagem de classes na metodologia *Unified Modeling Language* (UML) que possui um catálogo de padrões, os quais podem ser selecionados de uma lista e inseridos na modelo em construção.

A maioria desses repositórios não permite autoria, ou seja, o desenvolvedor não pode inserir novos padrões (MARINHO,2003). Exemplos de ferramentas *case* que utilizam padrões para geração automática de código são o Rational Rose e o Websphere Studio da IBM (2003); ModelMaker da ModelMaker Tools (2004), o Eclipse e o CodePro Studio da Instanciations (2004).

BibPC e TutorC, propostos nesta tese, possuem algumas características em comum com tais repositórios e *cases*, tais como (1) modelagem e armazenamento de conhecimento em programação baseado em padrões de software; (2) acesso automático, a esse conhecimento, por outro sistema e (3) atividades de programação. Uma das diferenças reside no tipo de usuário dos sistemas. BibPC é proposta para uso por professores e ambientes de aprendizagem, tal como TutorC, o qual tem o aprendiz como usuário final. Repositórios e Cases são destinados a programadores experientes. A figura 11, mostra uma analogia entre sistemas e atores nos dois contextos.

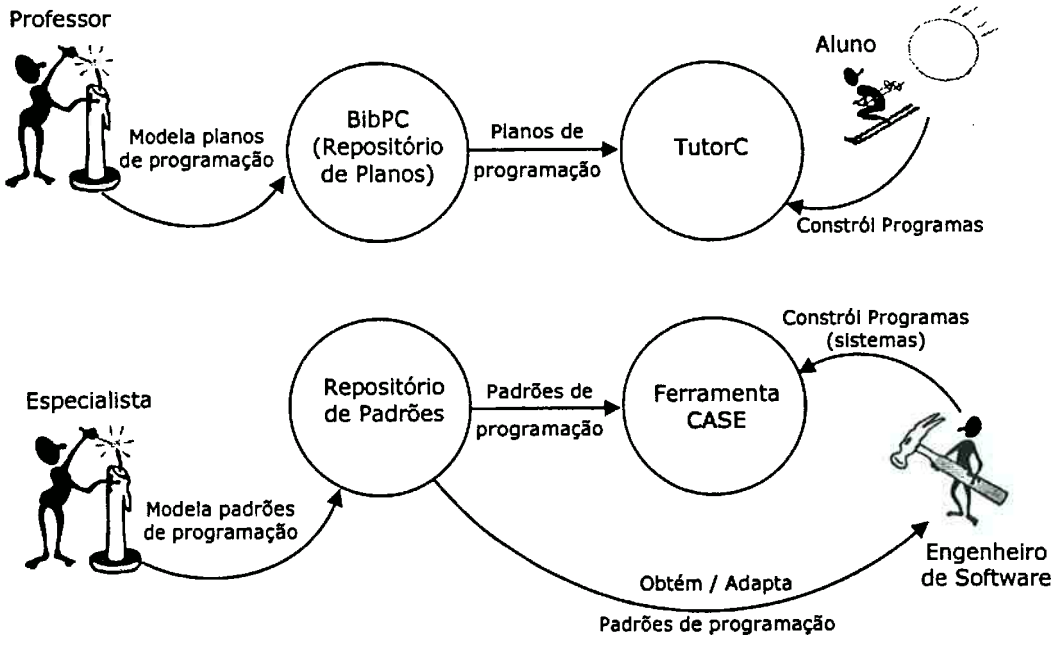


Figura 11 – Analogia dos contextos BibPC/TutorC x Repositório/CASE

4.3 Outros Ambientes de Aprendizagem

A literatura apresenta ainda uma ampla variedade de ambientes para aprendizagem de programação, as quais não se configuram como Sistemas Tutores Inteligentes. Algumas delas são resumidas nesta seção.

Ellis e Lund (1994) projetaram um sistema e uma linguagem chamados G2, que promovem o aprendizado da linguagem C para programadores novatos. A estratégia de ensino utilizada é composta de três estágios:
1° estágio: o estudante aprende e cria seu programa com uma linguagem abstrata (G2), submete-o a um tradutor automático para C e, então compila e executa o programa transladado num compilador C;
2° estágio: estudantes são encorajados a comparar e estudar o código C produzido pelo tradutor com seu programa escrito em G2;
3° estágio: o tradutor é eliminado e o estudante translada seu programa G2 para C.

Na linguagem G2 o estudante tem disponível uma biblioteca de rotinas promovendo idéias de reusabilidade. No desenvolvimento de seu projeto, o estudante tem que pensar sobre os dados de uma maneira separada do método de resolução a ser projetado. O código principal pode ser representado numa ordem que é natural para o método de solução usado pelo estudante em vez da ordem forçada pelo compilador. O sistema G2 funciona bem se o estudante nunca comete erros ao usar a linguagem G2. Se o programa contém erros, estes são apontados na fase de compilação e na execução do programa. O problema é que erros apontados nessas fases são sobre C e não sobre G2. Isso causa sérias dificuldades pois os estudantes não sabem onde estão seus erros no programa em G2.

Eerola e Malmi (1994) propõem KELVIN, uma ferramenta para analisar estilo e estrutura de programas escritos em linguagem C, a qual poderia ser integrada a um sistema tutor. KELVIN espera que o usuário forneça programas sintaticamente corretos para serem analisados. Para programas incorretos, o autor indica o uso de compiladores. Assim como KELVIN, existem várias outras ferramentas semelhantes. Esses sistemas não analisam a sintaxe do programa ou se a lógica está correta. Outros exemplos desse tipo de sistema são: ACCESS e AUTOMARK (REDISH;SMYTH,1986).

De um modo geral, tais sistemas se utilizam de vários fatores métricos para calcular uma pontuação final para um programa em análise. As métricas avaliam, por exemplo, tamanho das linhas, uso profundidade de identação, uso de comentários, uso de linhas brancas como separadores, uso de espaços em brancos, variedade de palavras

reservadas, tamanho e variedade de nomes de identificadores, uso de rótulos (*labels*) e declarações goto, modularização do programa (uso de funções).

George (2000) realizou um trabalho recente no ensino de modelos mentais de execução de programas recursivos. Aprendizes utilizaram EROSI, uma ferramenta de visualização de modelos de execução, que auxilia-os a desenvolverem cópias de modelos mentais de recursão baseadas em modelos mentais de programadores experientes. A ferramenta auxilia aprendizes a simularem mentalmente o comportamento de um programa antes de escreverem o fragmento de código correspondente.

O Mac-Multimídia[13] (FICSHER,2001) implementado no Departamento de Ciência da Computação do IME-USP tem como proposta apoiar o aluno em seu processo cognitivo de programação em linguagem C nas disciplinas de Introdução à Computação (MAC-11x). O projeto consiste de um site na Internet que reúne material didático e software educativo utilizando-se de recursos de som, imagem, vídeo e animação para apresentar soluções de problemas por meio de simulações. A fim de auxiliar o aluno na resolução de um problema, o software educativo fornece apoio em três passos:

1. **Entendimento do Problema -** Este passo tem o objetivo de produzir, no aluno, o entendimento do que o programa deve fazer. O software educativo exibe através de uma animação do professor em sala de aula, a resolução do problema num nível alto de abstração.
2. **Codificação -** O aluno é conduzido por meio de perguntas interativas (testes), a pensar a respeito da ação a ser codificada. A cada teste corresponde, em geral, a implementação de uma linha de código do programa. Deste modo o aluno pode ter uma noção de como ocorre a passagem da linguagem natural para a linguagem de programação.
3. **Simulação -** Nesta etapa o programa é simulado passo a passo. Os valores assumidos pelas variáveis são mostrados a cada interação, promovendo um maior entendimento da lógica do programa.

4.4 Discussão

Existe uma diversidade de propostas para ensino e aprendizagem de linguagens de programação, na literatura. Sistemas Tutores Inteligentes parecem oferecer uma arquitetura mais completa e adequada para o desenvolvimento de um ambiente de apoio ao processo cognitivo do aprendiz em programação.

---

[13] Projeto Mac Multimídia: http://www.ime.usp.br/~macmulti

Assim, as discussões nesta seção se resumem aos sistemas tutores. Do estudo realizado nesta tese e resumido neste capítulo, é possível concluir que grande parte da pesquisa no projeto de sistemas tutores foi dedicada muito mais a aspectos de representação interna do conhecimento sobre programação (para fins de capacitar o sistema a realizar diagnóstico) do que para comunicar o conhecimento ao aprendiz e apoiá-lo durante o processo de programação, o qual não envolve somente codificação. Exceto em alguns sistemas, tal como BRIDE (BONAR;CUNNINGHAM,1988), não houve preocupação em identificar a natureza do conhecimento a ser manipulada pelo aprendiz, na interface dos tutores. Em geral, esse conhecimento ficou restrito ao nível da implementação. Isto pode ser bem compreendido pela análise de como o aprendiz trabalha na interface desses tutores. Na grande maioria dos sistemas, o estudante constrói seu programa manipulando somente a linguagem de programação ou, em alguns casos, dispõe de um editor estruturado e um sistema de ajuda *on-line* sobre conceitos e exemplos da linguagem. Quando o estudante constrói seu programa no nível da implementação, seu comportamento não pode ser observado, o tutor recebe apenas o produto final: o programa do estudante.

Essa classe de tutores implementa um processo de diagnóstico bastante complexo, uma vez que precisa comparar representações diferentes: o modelo interno (módulo especialista) e o modelo externo (interface do aluno). Particularmente, domínios de solução de problema, tal como o da programação, soluções tendem a ser objetos complexos cuja forma não pode ser entendida independentemente do caminho pelo qual elas foram produzidas (WENGER,1987). Conseqüentemente, diagnóstico requer a reconstrução do raciocínio de solução de problema, o que exige estratégias complexas, ou uma forma idealizada do caminho da solução de problema, o que gera imprecisões nos resultados. A fim de diminuir tal imprecisão, vários tutores foram desenvolvidos baseados em biblioteca de erros. A desvantagem é que essa abordagem está baseada sobre a noção de erros cognitivos em procedimentos específicos tornando difícil o reuso e manutenção do sistema. Assim, somente erros armazenados são reconhecidos, erros novos são ignorados.

A maioria dos sistemas tutores se preocupa e realiza muito mais diagnóstico e decisões tutoriais, do que promovem expansões cognitivas no estudante. Poucos tutores disponibilizam na interface, representações de componentes cognitivos e/ou estratégias para programação nos níveis acima da implementação. Porém, como mostram os capítulos 1 e 2, a maior dificuldade do aprendiz em programação, reside na construção do modelo mental do programa.

Neste trabalho, o interesse está em identificar, modelar e tornar visíveis, num sistema tutor, componentes cognitivos para uso pelo aprendiz durante a atividade de programação. Tais componentes devem estender ou complementar a linguagem de programação de tal forma que o aprendiz, interagindo com o ambiente, possa construir o seu modelo mental de programa. Para atingir esses objetivos, várias questões precisam ser respondidas para o projeto de um sistema tutor inteligente (WENGER,1987):

1. Qual o conhecimento que se deseja construir no estudante?
2. Qual o conhecimento a ser embutido no sistema para alcançar o objetivo anterior?
3. Qual o conhecimento que se deseja utilizar para comunicação entre estudante e tutor?
4. Como representar o conhecimento para as diversas finalidades?
5. Outras questões referem-se às técnicas de diagnóstico. Mas diagnóstico é dependente de tipos de conhecimento e sua representação. Nos capítulos que se seguem podem ser encontradas respostas para essas questões.

5 PROPOSTA DE MODELO DE CONSTRUÇÃO DE PROGRAMAS PARA APRENDIZES

A proposta deste trabalho é modelar e disponibilizar ao aprendiz componentes de planejamento para programação, representando conhecimento especialista em programação, tais como metas, planos, ações e estratégias de programação. Esses componentes devem estar organizados de uma maneira especial a fim de que o iniciante em programação possa aprender a criar sua própria organização mental para uso posterior. Considere o problema 2, apresentado no capitulo 1:

Dado pelo usuário, um número inteiro positivo n, calcular e mostrar o valor da seguinte série: 1 + 1/2 + 1/3 + ... + 1/n

Uma possível solução, inicia-se com a identificação das metas globais a serem alcançadas:

Meta-A: Obter um número *n* do usuário

Meta-B: Obter a soma de uma série gerada automaticamente

Meta-C: Mostrar o resultado da soma

Essas metas podem ser alcançadas por meio da execução de um determinado conjunto de ações, como as propostas na tabela 9.

Tabela 9 – Ações de programação para o problema 2

1. Calcular soma iterativa
2. Implementar leitura do teclado
3. Declarar variáveis
4. Identificar início de seqüência
5. Identificar fim de seqüência
6. Emitir mensagem para o usuário
7. Definir expressão que estabelece o incremento/decremento da seqüência
8. Definir expressão para geração de um termo da sequência
9. Definir se a seqüência será gerada de forma crescente ou decrescente
10. Identificar a variação entre dois termos da sequencia
11. Gerar sequência numérica
12. Inicializar variável para soma iterativa
13. Definir expressão geral da soma
14. Mostrar resultado na tela
15. Selecionar variável que contém o resultado

Existe uma relação hierárquica entre essas ações (figura 12): ações de níveis mais altos na hierarquia são denominadas **ações abstratas** ou **ações compostas** ou ainda **planos de programação**, uma vez que podem ser decompostas em ações mais simples. Ações de nível mais baixo na hierarquia são denominadas **ações primitivas**, uma vez que podem ser diretamente mapeadas para uma linguagem de programação.

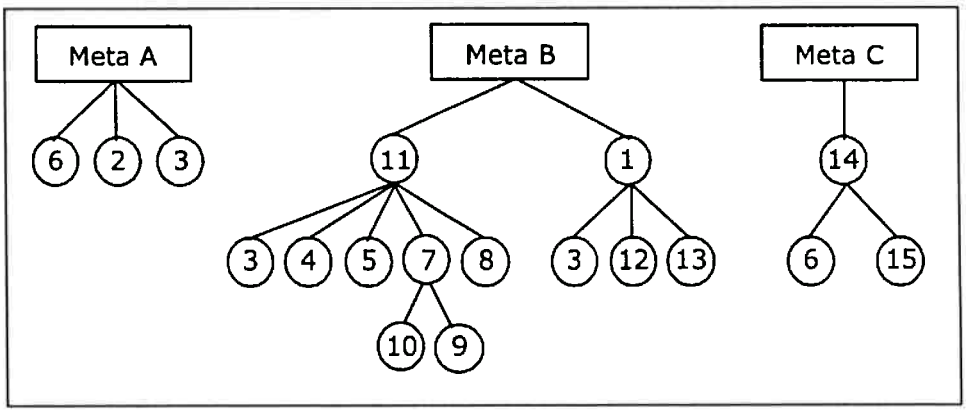


Figura 12 – Hierarquia entre ações de programação

O próximo passo consiste em traduzir as ações primitivas em código. Para realizar esse passo devem ser associados conhecimento da sintaxe da linguagem de programação e dados fornecidos no enunciado do problema. A tabela 10 relaciona ações primitivas (folhas das árvores) ao código instanciado com dados do problema.

Tabela 10 - Ações de programação mapeadas ao código

Ações Primitivas	Código
2. Implementar leitura do teclado	`scanf("%f", &n);`
3. Declarar variáveis	`float n, d, f, soma;`
4. Identificar início de seqüência	`d=1`
5. Identificar fim de seqüência	`d<=n`
6. Emitir mensagem para o usuário	`printf("Digite um número: ");` `printf("A soma total é %f", );`
8. Definir expressão para geração de um termo da seqüência	`f=1/d`
9. Definir se a seqüência será gerada de forma crescente ou decrescente	`++`
10. Identificar a variação entre dois termos da sequencia	`1`
12. Inicializar variável para soma iterativa	`soma=0`
13. Definir expressão geral da soma	`soma=soma+f`
15. Selecionar variável que contém o resultado	`soma`

É importante notar que um único comando (declaração) de uma linguagem de programação pode implementar (realizar) várias ações primitivas. A tabela 11 mostra as ações compostas 11 e 14, as quais são composições de várias ações primitivas.

Tabela 11 – Declarações em C que agrupam ações primitivas

Ações Primitivas	Código
11. Gerar seqüência numérica	for(d=1 ;d<=n ; d++) { }
14. Mostrar resultado na tela	printf("A soma total é %f", soma);

Como já mencionado, a pesquisa em Ciência Cognitiva mostra que programadores formam **planos de programação** agrupando trechos de código e associando-os a operações de alto nível. Por exemplo, a tabela 11 pode ser reduzida a um conjunto de quatro planos instanciados para o problema 2, como mostra a tabela 12.

Tabela 12 – Planos de programação para o Problema 2

Planos de Programação	

Soma de seqüência automática	soma=0;
	for(d=1; d<=n; d++)
	{
	soma=soma+1/d;
	}

Entrada por teclado	printf("Digite um número: "); scanf("%f", &n);
Declaração de variáveis	float n, f, soma; int d;
Saída de dados	printf("A soma total é %f", soma);

Dessa forma, programadores experientes conseguem mapear metas diretamente para planos de programação. Finalmente, o programa é obtido ordenando-se os planos de programação (Programa 4). Isso requer conhecimento sobre a ordem correta em que as metas devem ser alcançadas para que o problema seja resolvido.

```
main()
{
float n, d, f, soma;
soma=0;
printf("Digite um número: ");
scanf("%f", &n);
for(d=1; d<=n; d++)
        {
        f=1/d;
        soma=soma+f;
        }
printf("A soma total é %f", soma);
}
```

Programa 4

Rocha (1995) observou, em seus experimentos sobre aprendizagem de programação, que a maioria das pessoas não conseguia sintetizar o funcionamento de um procedimento. Efetuavam uma leitura comando a comando, explicando-os isoladamente. Assim, concluiu que o entendimento de como funcionam as construções de uma linguagem não é suficiente para compreender programação. Um aspecto que deve ser enfatizado no ensino da programação é que comandos, quando agrupados, deixam de ter um significado e estrutura descontextualizada. Segundo Rocha, sem essa visão o aluno não consegue, por exemplo, adquirir estratégias de solução de problemas tal como a de dividir um problema em partes.

O modelo de processo de construção de programas proposto neste capitulo focaliza a aprendizagem de programação por meio da resolução de problemas. Para esse fim, são modelados planos de programação e sua decomposição em ações. Cada um desses componentes é visto como um bloco (de comandos) definido segundo uma funcionalidade. A próxima seção mostra como padrões elementares de programação contribuíram para essa modelagem.

5.1 Planos: Generalizando Padrões Elementares de Pprogramação

A estrutura de planos proposta, herda algumas características dos Padrões Elementares de Programação (PEP, 2001). Neste trabalho, deseja-se usar as estruturas de código dos padrões elementares de programação como blocos para construir soluções diversas para uma classe de problemas. Porém, as estruturas de código existentes nos padrões da literatura são descritos a partir de exemplos. Uma vez que a intenção aqui é definir um processo de programação e implementá-lo num ambiente de aprendizagem, há a necessidade de se generalizar tais exemplos para serem instanciadas durante a construção de uma solução de problema em particular. Uma maneira encontrada para tornar o código genérico foi a de criar um conjunto de **meta-variáveis**. Uma meta-variável fornece a semântica para um certo elemento a ser inserido num plano de programação. Para o aprendiz, as meta-variáveis servem como um guia para instanciar planos de programação com dados do problema. Por exemplo, uma leitura de teclado, codificada em C, pode ser:

```
scanf("%i", &a);
```

Do protótipo da função *scanf*, da linguagem C, sabe-se que o primeiro argumento trata-se de uma cadeia de caracteres formatada (*string*) que especifica o tipo de dado a ser lido do teclado. O segundo argumento é o endereço da variável onde o dado será armazenado. Generalizando, a linha de código pode ser escrita como:

```
scanf($formato-de-entrada, $endereço);
```

na qual, as palavras precedidas por **$** são **meta-variáveis** a serem instanciadas quando essa linha de código for utilizada para construir uma solução de problema particular. Note que o nome escolhido para uma meta-variável indica o papel que dados do problema desempenham na linha de código, quando assumirem o lugar da meta-variável. Outros exemplos de meta-variaveis são: $operador, $resultado, $tamanho, $valor, etc.

O código genérico que compõe um plano de programação foi denominado de **esquema do plano**. A tabela 13 apresenta exemplos particulares que constam na descrição dos padrões "escolha-sequencial" e "escolha-não-relacionada" de Bergin (1999) e os esquemas dos planos correspondentes.

Tabela 13 – Generalizando Padrões Elementares

Nome do padrão	Exemplo particular	Esquema do plano
escolha-sequencial	`if (participants > 15000) { rentTheSuperdome(); } else if (participants > 1500) { rentTheCivicCenter(); } else if (participants > 150) { rentATent(); } else { rentAMovie(); }`	`if ($condição1) { $ação; } else if ($condição2) { $ação; } else if ($condição3) { $ação; } else { $ação; }`
escolha não relacionada	`if (roofIsLeaking()) { callRoofer(); } if (sinkIsLeaking()) { callPlumber(); } if (floorIsLeaking()) { callExcavator(); }`	`if ($condição1) { $ação; } if ($condição2) { $ação; } if ($condição3) { $ação; }`

5.1.1 Componentes de um Plano de Programação

Para que o estudante possa "aprender" sobre determinado plano de programação, é importante que ele possa visualizar como o **esquema do plano** é formado. Ou seja, quais as ações primitivas embutidas no plano. Como apresentado na solução do problema 2, uma ação cognitiva de programação pode ser tão mínima a ponto de sua implementação corresponder a uma pequena parte de uma linha de código. Para fins de aprendizado pode não ser interessante decompor um plano em partes tão mínimas.

Assim, adotou-se como ação primitiva (ação mínima de programação) àquela que pode ser implementada por uma declaração, ou seja, uma linha de código. Por exemplo, na tabela 11, apresentada no início deste capítulo, a linha

```
for(d=1; d<=n; d++)
```

corresponde à implementação da ação 11 ("gerar sequência numérica") da tabela 10, a qual condensa as ações 4, 5, 8, 9 e 10 da tabela 10. Ou seja a ação 11 é uma ação composta, porém, quando implementada corresponde a apenas uma linha de código. Assim, o plano "Entrada por teclado" da tabela 14, pode ser decomposto em duas ações primitivas: "Emite mensagem para o usuário" e "Obtém dados do usuário".

Tabela 14 – Decomposição do plano "Entrada por teclado"

Ações Primitivas	Esquema primitivo
Emite mensagem para o usuário	`printf($formato-de-saída);`
Obtém dados do usuário	`scanf($formato-de-entrada; $enderecos);`

A figura 13 apresenta a estrutura genérica que define um plano de programação para o processo de construção de programas proposto para aprendizes.

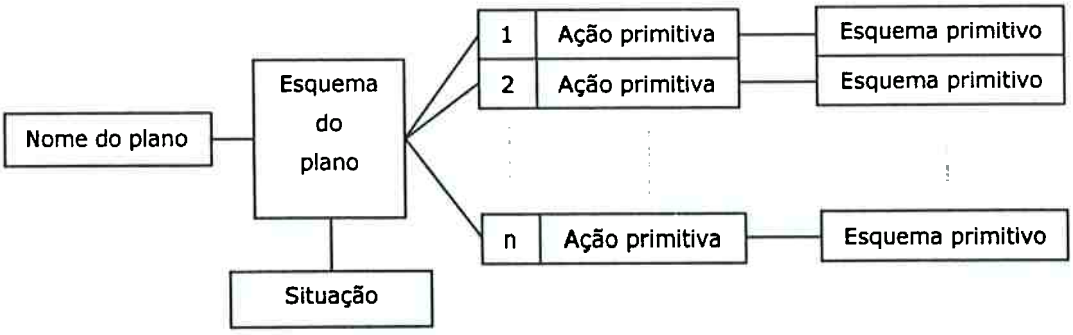


Figura 13 – Estrutura genérica de um plano de programação

Todo plano tem um **nome** que o identifica e a descrição de uma **situação** em que o uso do plano é adequado, tal como os padrões propostos pela área de Padrões Pedagógicos. Essa informação serve como um guia para o aprendiz selecionar um plano correto para resolver um problema de programação. Cada plano está associado a um esquema que representa um *chunk* de programação. Como mencionado anteriormente, esse esquema é instanciado sempre que o plano for utilizado numa solução em particular.

Um conjunto de **ações primitivas** explicam a lógica existente no esquema do plano de programação. Cada ação primitiva possui a linha de código genérico (com meta-variáveis) que a implementa, o qual é denominado de **esquema primitivo**. A enumeração, que pode ser vista na figura 13, define a ordem das ações e dos esquemas primitivos, no plano. O conjunto de esquemas primitivos e sua respectiva ordenação definem o **esquema do plano**.

Utilizando planos e suas partes para compor programas, espera-se que o aprendiz possa "aprender" temas interessantes sobre programação no nível cognitivo, tais como:

- Aprender sobre identificação de metas que devem ser alcançadas para resolver um problema, uma vez que planos são soluções para metas de problemas. Por outro lado, aprender sobre as situações em que determinado plano é útil.
- Aprender sobre a ordem das ações primitivas no plano, especificada pelo especialista em programação.
- Aprender a formar planos de programação através da agregação lógica de linhas de código, tal como nos planos propostos.
- Aprender a formar planos de programação em diferentes níveis de abstração, visualizando planos mais complexos da biblioteca, os quais são agregações de planos mais simples.
- Aprender sobre ordenação de planos a fim de compor uma solução completa para um problema.
- Aprender sobre como utilizar dados do problema no código que implementa a solução desse problema. Para auxiliar nesse aprendizado, as meta-variáveis definem a função que um certo valor ou variável desempenha no código.

A tabela 15 mostra três planos de programação que realizam as três metas do problema 2.

Tabela 15 – Planos de Programação para o Problema 2

Metas do Problema	Plano de Programação
Obter um número n do usuário	Entrada por teclado
Obter a soma de uma série gerada automaticamente	Soma de seqüência automática
Mostrar o resultado da soma	Saída de dados

A tabela 16 apresenta o esquema e as ações primitivas que compõem cada plano. Note que cada ação primitiva corresponde a uma linha de código do esquema do plano.

Tabela 16 – Esquemas e Ações Primitivas dos planos do problema 2

Nome	Esquema	Ação Primitiva
Entrada por teclado	printf($formato-de-saída);	Emite mensagem para o usuário
	scanf($formato-de-entrada, $endereço);	Obtém dados do usuário
Soma de sequência automática	soma=0;	Inicializa soma com elemento neutro
	for(k=$inicio; k<=$fim; k++)	Configura parâmetros da seqüência
	{	Constrói ações a serem repetidas
	soma=soma+$termo;	Adiciona termo na soma anterior
	}	Finaliza laço
Saída de dados	printf($formato-de-saída, $variaveis);	Mostra dados na tela

A tabela 17 contém as descrições de aplicabilidade dos planos (**situação**), tal como os padrões elementares de programação. Essa descrição auxilia o aprendiz a perceber em qual situação ele se encontra quando pretende alcançar uma meta de problema.

Tabela 17 – Descrição da aplicabilidade dos planos do problema 2

Nome do Plano	Situação
Entrada por teclado	Você está numa situação onde o usuário (pessoa que utiliza o programa) precisa interagir com o programa, informando dados. Esses dados podem ser numéricos ou caracteres e devem ser descritos num certo formato.
Soma de sequência automática	Você deseja somar uma sequência de números. O início e tamanho (ou fim) da seqüência são conhecidos. Os números k são gerados automaticamente de forma crescente. A distância entre um número e outro da seqüência é de uma unidade. Pode ser desejável realizar algum processamento extra com o valor k, para gerar o termo específico da seqüência.
Saída de dados	Você está numa situação onde o usuário deseja visualizar, na tela, o resultado de um processamento ou dados em geral, os quais estão armazenados em variáveis. Uma mensagem de texto pode acompanhar os dados, na tela.

5.2 Componentes do Modelo de Construção de Programas Proposto

O modelo proposto utiliza duas bases de conhecimento:

- A **Biblioteca Cognitiva**, a qual contém **planos de programação** e **descrições de problemas**. Um plano na biblioteca é definido por:
    - uma **situação**,
    - uma estrutura de código generalizada, a qual foi chamada de **esquema** e,
    - um conjunto de **ações primitivas**. Uma descrição de problema é constituída de um **enunciado** (texto) e um **modelo de problema** que consiste de um conjunto de metas a serem alcançadas para resolver o problema.

- A **Base de Conhecimento Interna** que corresponde ao conhecimento prévio (*background knowledge*) do aprendiz, constituída pelo conhecimento de domínio de problemas e o conhecimento mínimo de programação, pré-existente no aprendiz.

A figura 14 resume o Modelo de Construção de Programas para Aprendizes proposto: os retângulos contêm as fontes de conhecimento e as elipses representam o **Modelo do Programa** e o **Programa** propriamente dito.

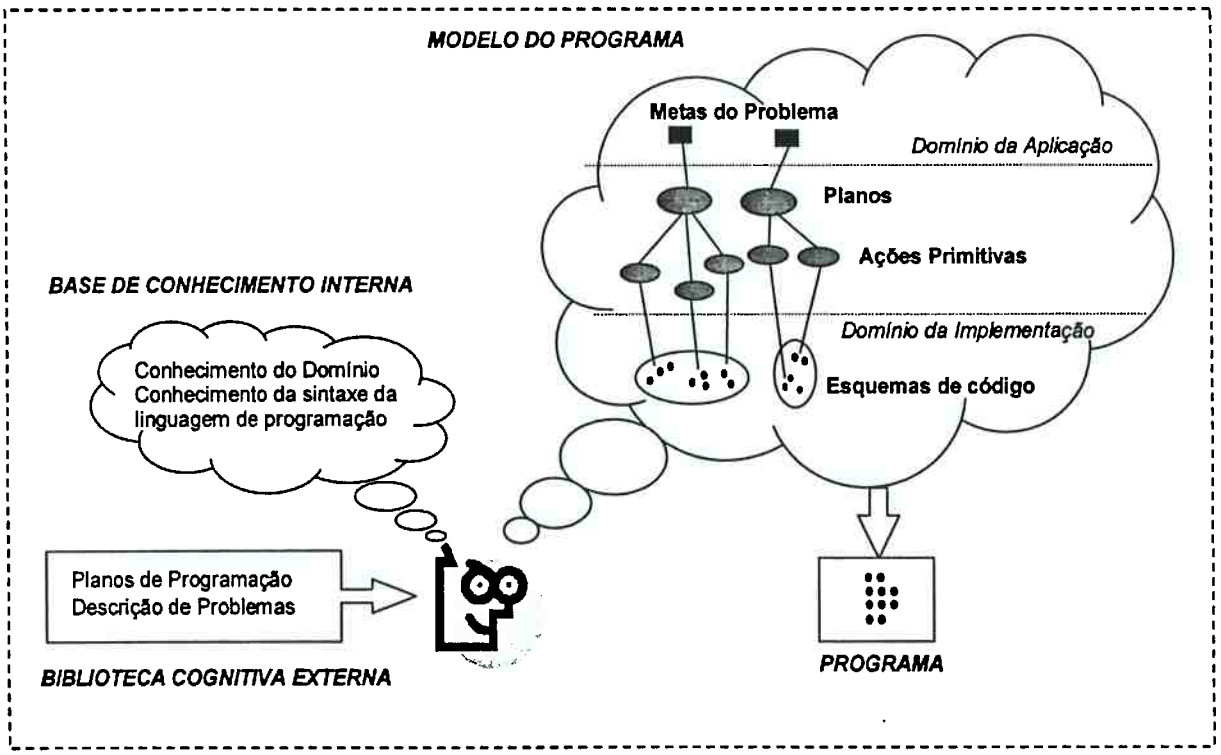


Figura 14 - Modelo do Processo de Construção de Programas para Aprendizes

O **Modelo do Programa** e o **Programa** são obtidos pela manipulação dos componentes cognitivos e de implementação existentes nas fontes de conhecimento. A figura 15 apresenta as principais relações entre os componentes do modelo cognitivo de construção de programas.

Nela, observa-se que um modelo de problema, descrito por um conjunto de metas, é resolvido implementando-se o modelo do programa. Este, por sua vez, é composto por uma hierarquia de metas do problema, planos e ações de programação. A hierarquia define o plano que realiza determinada meta do problema, assim como as ações primitivas que o compõe. Cada ação primitiva é implementada por um esquema primitivo. O conjunto dos esquemas primitivos ordenados formam o esquema do programa, o qual é considerado também, um elemento do modelo do programa. O esquema do programa, quando instanciado com dados do domínio do problema, forma o programa que implementa o modelo abstrato. A próxima seção explica como um programa pode ser construído usando o modelo proposto.

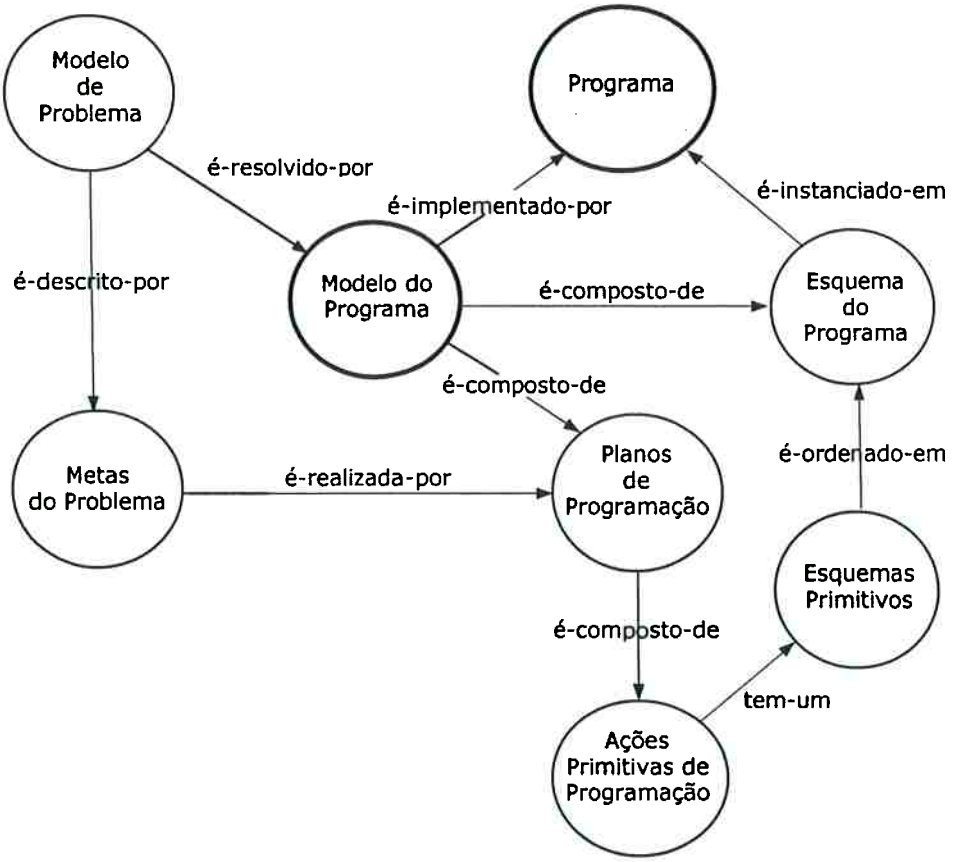


Figura 15 – Relações entre os componentes do modelo de construção proposto

5.3 Estratégia de Construção de Programas no Modelo Proposto

A seção anterior apresentou os principais componentes do Modelo de Construção de Programas proposto, os quais se deseja disponibilizar para apoiar a atividade de programação e promover a formação do novo conhecimento no aprendiz. Esta seção tratará a respeito de como um aluno poderá acessar e utilizar os componentes da Biblioteca Cognitiva para resolver problemas. No Problema 3, selecionado da Biblioteca Cognitiva, o modelo de problema apresenta o conjunto de metas que devem ser satisfeitas.

Problema

Enunciado: Faça um programa que permita que o usuário forneça um conjunto de 15 números inteiros (positivos e negativos). O programa deve mostrar na tela o maior número do conjunto de números fornecidos.
Modelo do Problema:
Meta-1: alocar espaço na memória para os dados
Meta-2: obter primeiro número do usuário
Meta-3: ler e identificar o maior número da seqüência
Meta-4: mostrar o maior número da seqüência na tela

Problema 3

A descrição de problema pode ser apresentada ao aluno de maneiras diferentes. Dependendo do grau de conhecimento do aluno, um sistema de aprendizagem que implementa o Modelo de Construção de Programas proposto, pode apresentar:

- O enunciado e as metas do problema (modelo do problema) já ordenadas, como na descrição do problema 3, ou parcialmente ordenadas. Esse seria o nível de menor dificuldade para o aluno, uma vez que a ordem das metas, estabelece a ordem dos planos de programação na composição do programa;
- O enunciado e as metas do problema, propositalmente desordenadas;
- Somente o enunciado do problema. Nesse caso o aluno necessita decompor o enunciado em sub-problemas para definir o conjunto de metas a serem realizadas com planos de programação da Biblioteca Cognitiva.

Uma vez definido o Modelo do Problema, , o aprendiz seleciona um plano na **Biblioteca Cognitiva** para cada meta do Modelo de Problema. Essa seleção é guiada, principalmente, pela descrição de aplicabilidade do plano, ou seja, a **situação**. Porém, o **esquema do plano** pode, também, auxiliar nessa escolha. A descrição de aplicabilidade de um plano e seu esquema podem ser vistos na tabela 18.

Tabela 18 – Características do Plano Máximo-ou-Mínimo

Situação	Esquema

Você deseja encontrar o valor máximo ou mínimo de uma seqüência numérica onde o usuário fornece um número de cada vez. O tamanho da seqüência é conhecido a priori.	$maxmin= №
	for (contador=1; contador<$tamanho; contador++)
	{ printf($formato-de-saída);
	scanf($formato-de-entrada, &numero);
	if (numero $relacional $maxmin)
	{ $maxmin = numero;
	}
	}

Por exemplo, para alcançar a **meta-3**, o plano **máximo-ou-mínimo** pode ser selecionado. A criação desse plano foi baseada no padrão elementar de programação "Valores Extremos" do trabalho de Astrachan & Wallingford (1998). A tabela 19 mostra os planos selecionados para todas as metas do problema 3.

Tabela 19 – Seleção de planos para o problema 3

Metas do Problema	Planos
Meta-1	P1: Declaração de variáveis
Meta-2	P2: Entrada por teclado
Meta-3	P3: Máximo-ou-mínimo
Meta-4	P4: Saída de dados

Uma vez que os planos tenham sido selecionados, o aprendiz deve reconsiderar as metas do problema e definir a ordem correta de execução destas. Isto define, também, a ordem em que os planos devem ser arranjados no programa. A descrição textual das ações primitivas dos planos e a visualização dos esquemas auxiliam na construção da lógica que deve estar embutida na solução. Esse passo de ordenação gera o chamado **Esquema do Programa**, o qual se constitui dos esquemas primitivos ordenados. As ações primitivas ordenadas geram a explicação (formalização dos conhecidos comentários do programa) de cada linha do esquema do programa e conseqüentemente da lógica do programa. A tabela 20 mostra o Esquema do Programa para o problema 3.

Tabela 20 – Esquema do Programa para o Problema 3

Plano	Esquema do Programa	Ações Primitivas
P-1	$tipo $variaveis;	Declaração de variáveis simples
P-2	printf($formato-de-saída);	Emite msg. para o usuário fornecer primeiro nº
	scanf($formato-de-entrada, &numero);	Obtém número
P-3	$maxmin= $numero;	Assume que 1º número é o maior (menor) de todos
	for (contador=1; contador<$tamanho; contador++)	Configura condição para repetição
	{	Constrói ações a serem repetidas
	printf($formato-de-saída);	Emite msg. para o usuário fornecer próximo nº
	scanf($formato-de-entrada, &numero);	Obtém número
	if (numero $relacional $maxmin)	Verifica se número é maior (menor) que anterior
	{	Constrói ações para condição verdadeira
	$maxmin = numero;	Atualiza máximo (mínimo)
	}	Finaliza if
	}	Finaliza laço
P-4	printf($formato-de-saída, $variaveis);	Saída de dados

Neste ponto, o **Modelo do Programa** é concluído, constituindo-se de um conjunto de componentes formado por metas, planos, ações, esquemas de planos e o esquema do programa. Porém, para obter o programa propriamente dito, o aprendiz deve ainda instanciar o esquema do programa. Usando seu conhecimento prévio este deverá ser capaz de instanciar as meta-variáveis do esquema do programa com conhecimento do domínio do problema, obtendo o **programa** (Programa 5) propriamente dito.

```
void main()
{
int contador, numero, maior;
printf("Digite um número: ");
scanf("%i", &numero);
maior=numero;
for (contador=1; contador<15; contador++)
      {
      printf("Digite um número: ");
      scanf("%i", &numero);
      if (numero > maior)
            maior=numero;
      }
printf("O maior é %i", maior);
}
```

Programa 5

A atividade de ordenação é simples quando o número de planos é pequeno e não existe a necessidade de se implementar planos deslocalizados. Um plano deslocalizado é um algoritmo ou plano de programação fragmentado (espalhado) no código fonte. Ou seja, seu código aparece distribuído em lugares diferentes ao longo do programa. Os primeiros pesquisadores a definirem um plano de programação deslocalizado, foram Soloway & Letovsky (1986) e Lampert et al. (1988).

A tabela 21 mostra um exemplo de uso de plano deslocalizado numa solução de problema. Note que o plano P2 está fragmentado, na solução.

Tabela 21 – Exemplo de ocorrência de planos deslocalizados

Meta	Plano	Esquema do Programa
M1	P1	`$tipo $variaveis;`
M2	P1	`$tipo $variaveis;`
M3	P2	`do{`
M4	P3	`printf ($formato-de-entrada);` `scanf ($formato-de-entrada, &$variavel);`

M5	P4	`contador=0;`
		`while (contador <= $vezes)`
		`{`
		`resultado=contador * $variavel;`
		`printf ($formato-de-entrada, resultado);`
		`contador++;`
		`}`

M6	P2	`printf($formato-de-saída);`
		`letra=getch();`
		`} while ($condição);`

A figura 16 apresenta um algoritmo que descreve a atividade do aprendiz no modelo proposto.

algoritmo: constroi_programa
objetivo: Construir um programa para um problema
variáveis_de_entrada: BP: base de problemas
BCC: base de conhecimento cognitivo
BCI: base de conhecimento interna
variáveis_de_saída: Programa:conjunto ordenado de linhas de código
funções: seleciona, obtém, ordena, instancia
variáveis_internas: M: conjunto de metas do problema a ser resolvido
P: conjunto de planos selecionados para resolver as metas
EP: esquema do programa
meta, plano, ação-primitiva, esquema-primitivo

estrutura_de_controle:

constroi_programa (BP, BCC, BCI → MMP, Programa) =
seleciona (BP → M)
Para cada meta ∈ M
seleciona (BCC → plano)
P ← P ∪ plano
Para cada ação-primitiva ∈ P
obtém (ação-primitiva → esquema-primitivo)
ordena (esquema-primitivo, EP → EP)
Para cada esquema-primitivo ∈ EP
instancia (BCI, esquema-primitivo → código)
Programa ← Programa ∪ código

Figura 16 – Algoritmo das Atividades do Aprendiz no Modelo Proposto

5.4 Construção da Biblioteca Cognitiva

Conforme mencionado anteriormente, programadores experientes possuem uma biblioteca de estruturas cognitivas à sua disposição, desenvolvida através da prática. Novatos não possuem tal recurso e isto dificulta seus esforços para resolver problemas (DU BOULAY,1989). Com o objetivo de implementar e testar o modelo proposto, junto a um grupo de alunos da Faculdade de Engenharia de Sorocaba, surgiu a necessidade de se modelar um conjunto de componentes para dar início à construção da Biblioteca Cognitiva do modelo (LEMOS ET AL., 2003b). Como visto na seção 5.1, um meio para se modelar planos de programação pode ser a descoberta[14] e generalização de padrões elementares de programação, tipicamente propostos por professores experientes. Porém, a maioria dos padrões, disponíveis na literatura, foi escrita visando a modelagem orientada a objetos; poucos têm sido propostos em outras linguagens, tais como linguagens funcionais ou procedimentais (WALLINGFORD,1998;WALLINGFORD,2002). No entanto, o interesse, nesta tese, é sobre aprendizado de introdução à programação numa abordagem algorítmica.

Um curso introdutório de programação algorítmica com duração de um semestre compreende, no mínimo, um currículo que envolve tópicos tais como:

- conceitos básicos (tipos de dados, variáveis, operadores, expressões, funções básicas de biblioteca);

- estruturas de fluxo tais como seleção (if-else) e repetição (while, do-while, for);

- manipulação de variáveis estruturadas homogêneas (vetores e matrizes).

Uma vez que a abordagem proposta focaliza resolução de problemas, espera-se que o conjunto de conceitos básicos seja fornecido, por exemplo, pelo ensino tradicional.

Neste trabalho, optou-se por desenvolver planos de programação que envolvessem estruturas de seleção e de repetição, uma vez que correspondem ao Módulo-1 (meio semestre, incluindo conceitos básicos) da ementa da disciplina de Introdução à Programação da Faculdade de Engenharia de Sorocaba. Os tópicos e sub-divisões (níveis) que envolveram os planos modelados, assim como problemas e suas soluções são mostrados na tabela 22.

---

[14] Segundo Buschmann et. al (1996) apud Appleton (2000), escritores de padrões não necessitam ser inventores ou descobridores de novas soluções a serem documentadas e sim descobridores de soluções já utilizadas e provadas como boas soluções.

Tabela 22 – Currículo utilizado na construção de planos de programação

Currículo	
Tópico	**Nível**
Básico (sem estruturas)	Manipulação de números inteiros
	Manipulação de números reais
Seleção (IF-ELSE)	Uso de estruturas de seleção simples
	Uso de estruturas de seleção aninhadas
Repetição Geral (WHILE, DO-WHILE)	Uso de estruturas de repetição geral simples
	Uso de estruturas de repeticao geral simples e de seleção
	Uso de estrutras de repetição geral aninhadas
Repetição Limitada (FOR)	Uso de estruturas de repetição limitada simples
	Uso de estruturas de repeticao limitada e de seleção
	Uso de estruturas de repetição aninhadas

Dessa forma, foi necessário realizar um trabalho de modelagem de novos padrões elementares de programação e adaptação de padrões existentes para os tópicos acima. Para esse fim, foram desenvolvidas as seguintes atividades:

- Construção de planos de programação baseados nos padrões elementares de programação dos trabalhos de (BERGIN, 1999; BRADY, 1999; ASTRACHAN; WALLINGFORD, 1998);
- Construção de planos de programação baseados na modelagem de novos padrões elementares para programação algorítmica;
- Seleção de termos apropriados para rotular planos de programação e descrever sua aplicabilidade;
- Seleção de um conjunto de problemas que: (1) possam ser resolvidos com os planos obtidos e, (2) sejam suficientes e adequados para a prática num curso introdutório de programação. Para esse fim, foram utilizados materiais didáticos dos cursos de Introdução à Computação do Instituto de Matemática da USP e da Faculdade de Engenharia de Sorocaba;

5.4.1 Modelagem dos Planos de Programação

Na seção 3.4 do Capítulo 3 foram apresentados os requisitos básicos para a atividade de escrita de padrões de programação e, particularmente, padrões elementares de programação. A idéia básica nessa atividade, consiste em capturar problemas e suas soluções. Assim, com o objetivo de criar planos de programação, foram reunidos e analisados exercícios dos materiais didáticos dos cursos de Introdução à Computação do Instituto de Matemática da USP e da Faculdade de Engenharia de Sorocaba. Esse estudo consistiu em descobrir sub-problemas e suas soluções embutidas no código dos inúmeros exercícios analisados.

Por exemplo, o problema de se implementar a entrada de dados pelo usuário, surge em inúmeros exercícios e quase sempre a solução envolve o *chunk* mostrado no Programa 6.

```
printf("Entre com dois valores: ");
scanf("%i,%i", &a, &b);
```

Programa 6

Outro exemplo é o problema de se validar números fornecidos pelo usuário, o qual tem como uma das possíveis soluções, o *chunk* mostrado no Programa 7.

```
printf("Entre com um valor: ");
scanf("%i", &numero);
while (numero>0)
{
printf("Entre com um valor: ");
scanf("%i", &numero);
}
```

Programa 7

Note que o processamento dos números não está especificado no *chunk*. Outras linhas de código ou *chunks* devem ser inseridos para compor uma solução completa para um problema maior. Isto mostra bem a diferença entre funções e padrões elementares de programação ou entre funções e planos de programação. Padrões elementares de programação e planos de programação podem ser vistos como componentes, que corretamente combinados, formam o código de uma função num sistema de software.

Se uma função é simples, então o código interno da função pode corresponder ao código de um único plano. Porém, padrões e planos de programação como propostos aqui, são mais ricos em conhecimento de programação do que uma função. Padrões descrevem sua aplicabilidade, assim como as dificuldades envolvidas na solução do problema proposto no padrão. Planos, como os propostos neste trabalho, além de sua aplicabilidade, descrevem o conhecimento sobre ações primitivas que compõem o plano. Além disso, planos podem estar deslocalizados num programa completo, enquanto que uma função é uma unidade inseparável dentro do sistema de software.

Seguindo essa trajetória, uma série de planos foram construídos como uma proposta inicial da Biblioteca Cognitiva (LEMOS ET AL., 2003a). Certamente, os planos poderão ser melhorados por grupos de interesse, como sugere a área de Padrões Pedagógicos. A tabela 23 apresenta uma amostra dos planos obtidos. No Apêndice A, pode ser visto o conjunto completo dos planos, na forma de uma apostila, a qual foi utilizada com alunos da Faculdade de Engenharia de Sorocaba.

No Apêndice B, podem ser vistos alguns dos problemas propostos para serem resolvidos por meio dos planos de programação da apostila. Os problemas podem ser resolvidos utilizando, unicamente, os planos propostos, isto é, sem a necessidade de se incluir linhas de código adicionais ou planos de programação inexistentes na apostila. Porém, os planos construídos podem também serem utilizados na resolução de inúmeros outros problemas.

Tabela 23 – Amostra de Planos Construídos para a Biblioteca

Nome do plano	Tópico	Situação	Esquema

Escolha não Relacionada	Seleção	Existem várias ações e várias condições. Pode ser desejável executar várias das ações se as condições associadas forem verdadeiras. Cada ação tem uma condição que determina se ela deve ser executada.	`if ($condição1)`
			`  $ação;`
			`if ($condição2)`
			`  $ação;`
			`if ($condição3)`
			`  $ação;`

Verificar Sentinela	Repetição Geral	Você deseja receber números do usuário, um de cada vez, e validá-los segundo uma certa condição de entrada. Enquanto a condição for verdadeira, o próximo número deve ser recebido. A condição pode verificar, por exemplo, se o número recebido do usuário está dentro de uma faixa. Consulte os planos: "Entrada por teclado" e "Repetição com condição de entrada".	`printf ("Entre c/o número:");`
			`scanf ($formato-de-entrada, &numero);`
			`while($condição)`
			`{`
			`printf ("Entre c/o número:");`
			`scanf ($formato-de-entrada, &numero);`
			`}`

Soma de Seqüência Interativa	Repetição Limitada	Você quer somar uma seqüência de números. O tamanho da seqüência é conhecido. Os números da seqüência são fornecidos pelo usuário. Consulte os planos: "Entrada por teclado" e "Gera sequência".	`soma=0;`
			`for(k=1; k<$tamanho; k++)`
			`{`
			`printf($formato-de-saída);`
			`scanf($formato-de-entrada,&$variavel);`
			`soma=soma+$variavel;`
			`}`

5.4.2 Meta-Variáveis

A generalização dos padrões elementares de programação, a qual constituiu os esquemas dos planos de programação propostos, conduziu à criação de uma lista de meta-variáveis. Como mencionado na seção x, meta-variáveis definem uma hierarquia conceitual (ou mapas conceituais) sobre os parâmetros e variáveis dos planos de programação. Assim, as meta-variáveis estão classificadas segundo o tipo de conhecimento que elas representam: (a) **meta-variável da linguagem de programação** e; (b) **meta-variável do domínio de problema**.

A nomenclatura utilizada para nomear as meta-variáveis que representam conhecimento da linguagem de programação não é nova. Vários desses termos encontram-se documentados em livros, porém de forma isolada, como conceitos da linguagem e fora do contexto da atividade de resolução de problemas. A tabela 24 apresenta uma amostra das meta-variáveis da linguagem de programação.

Tabela 24 – Meta-variáveis da linguagem de programação

Meta-Variável	Significado
$tipo	indica que a instância deve ser um tipo de dado. Por exemplo: int, float, char
$endereços	indica que a instância deve ser um ou mais endereços de memória. Por exemplo, &x
$condição	indica que a instância deve ser uma expressão condicional que pode envolver tanto operadores relacionais como lógicos
$ação	indica que a instância deve ser um ou mais planos de programação (ação composta), os quais devem também serem instanciados

Meta variáveis do domínio do problema representam dados especificados ou solicitados no domínio do problema, os quais devem ser embutidos no código. A tabela 25 apresenta uma amostra desse tipo de meta-variável.

Tabela 25 – Meta-variáveis do domínio de problema

$resultado	indica que a instância deve ser uma variável que armazena o resultado de uma expressão.
$maximo	indica que a instância deve ser um valor (ou variável) que representa o máximo de uma seqüência de elementos.
$vezes	indica que a instância deve ser um valor (ou variável) que representa um certo número de vezes que uma ação deve ser executada.
$quantidade	indica que a instância deve ser um valor numérico (ou variável numérica) que representa uma certa quantidade

Todos esses termos fazem parte do vocabulário do professor ao construir programas na abordagem tradicional de ensino. Porém, uma vez que a carga cognitiva exigida na atividade de programação é muito grande, esse conhecimento transmitido apenas verbalmente, dificulta a assimilação pelo aprendiz. O objetivo de se documentar esse vocabulário nos planos de programação, como proposto aqui, não é o de criar uma segunda linguagem, mas sim formalizar atividades cognitivas que são mal explicadas em sala de aula ou não memorizadas pelo aluno. Mais importante do que a nomenclatura, meta-variáveis inseridas nos planos de programação formalizam o mapeamento entre conceitos da linguagem e sua aplicação na resolução do problema, assim como, o mapeamento entre dados do enunciado do problema e sua aplicação no código.

O Apêndice C apresenta o conjunto completo das meta-variáveis inseridas nos planos de programação propostos. Note que, um professor ou especialista em programação pode modificar ou criar novas meta-variáveis ao alterar ou construir novos planos de programação, de acordo com a necessidade de seu grupo de alunos.

5.5 Suposições e Restrições do Modelo

Para que o aprendiz possa encontrar uma solução para um problema existente na biblioteca, três suposições devem ser verdadeiras no modelo proposto:

(1) Para cada meta existente na biblioteca deve existir, pelo menos, um plano da biblioteca que a resolva.

(2) Todo e qualquer problema da biblioteca deve possuir, pelo menos, um **Modelo de Problema**, ou seja, um conjunto de metas que, uma vez alcançadas por planos da biblioteca, geram a solução do problema no nível acima da implementação (o esquema do programa).

(3) Planos são considerados corretos, uma vez que são para ser especificados por um professor ou programador experiente.

***Definição**: Um conjunto de problemas de programação Σ é resolvido por uma biblioteca cognitiva B se para qualquer problema $p \in \Sigma$, há pelo menos um programa implementado que possa ser gerado a partir dos componentes cognitivos da biblioteca B.*

A definição garante que há planos suficientes para compor, pelo menos, uma solução correta para um problema da biblioteca. Porém, à medida em que problemas aumentam em complexidade, o número de soluções corretas, para um mesmo problema, também aumenta. É possível que o processo cognitivo do aprendiz possa não estar contemplado na biblioteca. A fim de diminuir essa restrição, uma ferramenta de autoria é proposta no Capítulo 7 para que professores e especialistas possam expandir e reutilizar os componentes modelados. Nela, planos de programação, modelos de problemas e soluções alternativas para um mesmo problema podem ser modelados.

O Modelo proposto não garante que todas as soluções para um problema estarão modeladas na biblioteca. A proposta do modelo é fornecer um método de aprendizado, no qual o estudante desenvolve a habilidade de construir seus próprios modelos mentais de programas, os quais podem ser testados num compilador.

Na medida em que o conhecimento de programação do estudante evolui e este deseja criar suas próprias metas, planos e soluções para um problema, a biblioteca pode ser vista como uma fonte de consulta na qual planos existentes podem ser adaptados através da inserção ou remoção de linhas de código ou agrupados para gerar novos planos.

5.6 Tópicos sobre Implementação do Modelo Proposto

Apesar do modelo de processo proposto poder ser utilizado de forma manual, através do uso de apostilas impressas, um dos propósitos da tese é facilitar essas atividades implementando ferramentas de software. Esta seção propõe a implementação de duas dessas ferramentas e discute propostas para validação do modelo.

5.6.1 Ferramenta de Autoria

A criação dos componentes cognitivos de programação propostos, não é uma tarefa fácil. Ela requer conhecimento especializado em programação e uma quantia considerável de trabalho sobre modelagem, tanto para produzir modelos de problemas como planos de programação. Um sistema que possa capturar tal conhecimento de uma forma estruturada e reutilizável torna-se útil para toda a comunidade interessada, como por exemplo desenvolvedores de ferramentas cognitivas para aprendizado de programação ou educadores da área. Assim, idealizou-se a BibPC (Biblioteca para Programação Cognitiva), um sistema proposto para apoiar modelagem e reuso de conhecimento em programação algorítmica, através da WEB.

A idéia nesse sistema é a de que um professor ou especialista em programação possa modelar e inserir planos de programação, da forma apresentada na seção 5.1.1. E ainda, possa inserir problemas e resolvê-los diretamente no sistema, conforme estratégia de programação delineada na seção 5.3. Assim, não apenas planos de programação estarão disponíveis para reuso, como também problemas e resoluções completas: partindo-se de enunciados até o código final. Nessa trajetória, todos os componentes da figura 15 apresentada na seção 5.2, ficarão explícitos, através da BibPC.

BibPC caracteriza-se como uma ferramenta de autoria, uma vez que permite usuários, neste caso professores, inserirem material didático de seu interesse. Essa ferramenta, disponível na WEB para a comunidade acadêmica ou restrita a uma instituição, poderá promover a expansão e melhoria desse raro material didático. Pretende-se ainda que a base de dados seja facilmente acessível, via WEB, por desenvolvedores de ambientes de

aprendizado. A próxima seção descreve a intenção de projeto de um sistema tutor que acessa a BibPC, automaticamente. Assim, o sistema tutor poderá promover aprendizado de acordo com o material didático modelado pelo professor na BibPC. O Capítulo 7 apresenta os detalhes da implementação da BibPC.

5.6.2 Um Ambiente para Aprendizagem de Programação

Um dos objetivos da tese é implementar um ambiente de ensino e aprendizagem de programação baseado numa abordagem nova, distinto da maioria dos ambientes propostos na literatura, os quais, tipicamente, implementam alguma abordagem tradicional, tal como ensinar conceitos da linguagem ou propor exercícios de programação sem apoio às atividades cognitivas. A arquitetura escolhida para implementar o modelo proposto é a tipicamente utilizada para Sistemas Tutores Inteligentes (STI). Essa é uma especialização dentro do campo de instrução auxiliada por computador (CAI - *Computer-Aided Instruction*). Tutores Inteligentes são ferramentas de software para guiar a aprendizagem de novatos em algum domínio e que se utilizam de técnicas da área de pesquisa denominada Inteligência Artificial em Educação (AIED - *Artificial Intelligence in Education*) a qual inclui temas de áreas tais como Educação, Psicologia e Inteligência Artificial.

A arquitetura de um Sistema Tutor Inteligente é complexa e abrangente, porém permite um desenvolvimento gradual do sistema, de tal forma que pesquisadores podem focalizar o módulo de interesse, separadamente, em sua pesquisa. Módulos típicos encontrados em sistemas tutores inteligentes são: **módulo especialista**, o qual contém o conhecimento que o tutor deseja ensinar; **módulo interface do estudante**, o qual controla a interação de comunicação entre tutor e estudante; **módulo diagnóstico**, o qual é responsável por detectar erros cometidos pelo estudante; **modelo do estudante**, o qual contém alguma representação do conhecimento corrente do estudante e até mesmo o histórico das sessões de aprendizado e finalmente o **módulo pedagógico**, o qual regula a interação instrucional entre tutor e estudante, tal como propor revisão de um conceito ou apresentar um novo tópico para estudo.

O Modelo de Processo de Construção de Programas para Aprendizes, proposto na tese, focaliza o conhecimento especialista e estratégias de programação situadas no nível cognitivo. Assim, pretende-se propor um ambiente de aprendizagem, denominado TutorC, que utiliza a BibPC para implementar o **módulo especialista** e o **módulo interface do aluno**. Algumas idéias iniciais para o **módulo diagnóstico** e **modelo do estudante** são também implementadas.

O sistema TutorC encontra-se descrito no Capítulo 7. A figura 17 mostra as relações entre os atores (professor e estudante) e os sistemas propostos (BibPC e TutorC).

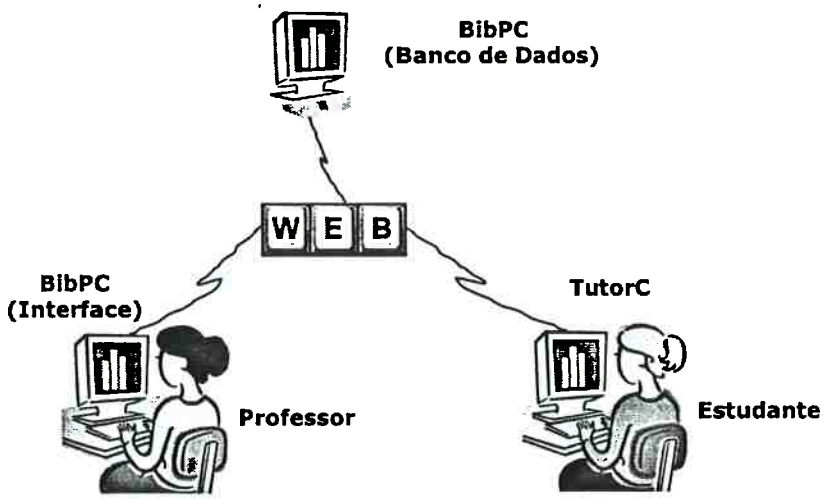


Figura 17 – Atores e Sistemas

5.6.3 Validação do Modelo

Uma preocupação que surge na proposta deste modelo, diz respeito à sua validação. A fim de provar, conceitualmente, componentes e atividades cognitivas descritos até agora e também, para facilitar a implementação das ferramentas, é proposta uma linguagem que formaliza o modelo proposto: a LPC (Linguagem de Programação Cognitiva), descrita no próximo capítulo.

Evidências empíricas mostram forte correlação entre as atividades de programação e de planejamento da área de Inteligência Artificial (*AI Planning*). Assim, LPC está baseada numa abordagem proposta nessa área de pesquisa: a abordagem para redes de tarefas hierárquicas de planejamento (*HTN - Hierarchical Task Network*) de Erol, Hendler e Nau (1992).

A BibPC, descrita no Capítulo 7, tem também um importante papel na validação do modelo. Além de ser uma ferramenta de autoria, ela é um meio para se gerar programas a partir de metas e planos de programação. Cada programa gerado como solução para um problema da biblioteca, pode ser compilado e executado, num compilador, a fim de testar o modelo de metas e os planos de programação envolvidos em cada problema.

Alguns fatores podem impossibilitar a geração de um programa para um dado problema ou ainda gerar um programa que não é uma solução correta para o problema. Exemplos desses fatores são: um modelo de metas errado ou incompleto e a ausência de um plano correto para uma certa meta.

Assim, a possibilidade de se gerar o programa para um problema, na BibPC, e testá-lo num compilador, poderá garantir as suposições e restrições discutidas na seção 5.5, além de prevenir a respeito de erros cometidos pelos usuários da BibPC. Uma das suposições é a de que um usuário da BibPC seja um especialista em programação, porém distrações tais como a seleção errada de um plano para uma meta, ordenação incorreta de ações primitivas ou instanciação incorreta do esquema do programa podem ocorrer. Essas falhas podem ser descobertas executando-se, num compilador, o programa gerado na BibPC.

Uma vez que o modelo cognitivo é proposto como uma abordagem nova para aprendizado de programação, além da prova conceitual, deseja-se realizar alguma avaliação empírica sobre o uso do modelo com alunos. Isto pode ser feito de duas maneiras: (1) de forma manual, oferecendo enunciados de problemas e modelos de metas assim como um conjunto de planos de programação para habilitar a atividade de programação; (2) de forma automática, oferecendo um ambiente de aprendizagem no computador que implemente o modelo proposto, tal como TutorC. O capitulo 7 descreve a metodologia utilizada e os resultados obtidos no uso preliminar da abordagem.

5.7 Resumo do Capítulo

Neste capítulo foi proposto um **Modelo do Processo de Construção de Programas para Aprendizes** baseado na pesquisa sobre compreensão e construção de programas da Psicologia da Programação e Padrões Pedagógicos. Esse modelo baseia-se ainda nas principais dificuldades observadas e relatadas por educadores durante o ensino da programação. Desta forma, a proposta focaliza a dificuldade em resolução de problemas algorítmicos, em vez de propor o aprendizado de conceitos básicos isolados tais como definições de tipos, variáveis, estruturas de fluxo, etc., como é feito no ensino tradicional.

O processo de construção de programas proposto, não supõe que este seja uma representação fiel do comportamento de programadores experientes, mas um meio para promover o desenvolvimento cognitivo de aprendizes em programação. O modelo habilita a construção da solução abstrata para um problema, denominada **modelo do programa**, assim como sua implementação, ou seja, o programa propriamente dito. Planos de programação são considerados elementos principais nessa abordagem. Eles são usados como blocos de construção pelo aprendiz a fim de construir programas para problemas de programação.

Planos consistem de um conjunto de ações primitivas ordenadas, as quais estão associadas a linhas de código genéricas denominadas **esquemas primitivos**. O conjunto dessas linhas formam o chamado **esquema do plano**, o qual corresponde a um *chunk* de programação tipicamente conhecido por um programador experiente. A definição do plano é complementada por um texto que descreve sua **aplicabilidade**. Atividades de planejamento tais como seleção, ordenação e instanciação de planos são deixadas para o aprendiz a fim de que este obtenha o programa final desejado.

Para ilustrar e avaliar o modelo proposto, um conjunto de planos foram modelados, baseados nos padrões propostos pela área de Padrões Pedagógicos e no material didático dos cursos introdutórios de programação da FACENS e do IME-USP. E ainda, um conjunto de problemas foram selecionados para resolução através do modelo proposto. Idealizou-se a implementação de uma ferramenta de autoria, a BibPC, que possa facilitar a modelagem de planos, modelos de problemas e suas soluções, constituindo-se assim, num repositório de material didático a ser utilizado na aplicação da abordagem proposta.

Uma vez que o modelo propõe componentes e estratégias que facilitam as atividades cognitivas de programação, acredita-se que sua utilização pelo aprendiz possa incentivá-lo a criar seus próprios modelos mentais. Uma das razões para essa crença é que os componentes modelados formalizam e documentam o vocabulário informal utilizado por educadores quando realizam atividades de construção de programas em sala de aula.

Neste capítulo, discute-se ainda, a importância do desenvolvimento de um ambiente de aprendizagem, o TutorC, que implemente o modelo proposto. Sistemas Tutores Inteligentes são sistemas com a proposta de embutir e aplicar conhecimento a ser aprendido. Tais sistemas utilizam técnicas da IA a fim de fornecer benefícios de instrução automática um-a-um. Acredita-se que um sistema tutor para programação pode se tornar uma ferramenta efetiva para aprendizagem de construção de programas se conhecimento cognitivo possa ser integrado e disponibilizado, ao aprendiz, em sua interface.

O próximo capítulo estuda um meio para representação de conhecimento cognitivo e estratégias de programação para facilitar a construção de sistemas que implementem o modelo proposto no todo ou parte dele.

6 UMA LINGUAGEM PARA PLANEJAMENTO EM PROGRAMAÇÃO

Conforme apresentado em capítulos anteriores, evidências empíricas mostram forte correlação entre a tarefa de programação e a tarefa de planejamento. Assim, um algoritmo em programação pode ser representado por um conjunto finito de ações ordenadas, isto é, um plano (OLIVEIRA;BORATTI,1999). Segundo Ehrlich e Soloway (1984), programadores experientes formam *chunks* definidos como planos bem conhecidos, os quais são reusados em atividades de compreensão e construção de programas. Combinando e instanciando tais planos, programadores obtêm o **plano-solução**, ou seja, o programa-solução.

Neste capítulo é proposta uma linguagem de representação formal para o modelo proposto no Capítulo 5, baseada na área de Planejamneto em Inteligência Artificial (IA) (RUSSEL;NORVIG,1995). Em particular, na linguagem empregada no planejamento hierárquico (HTN *planning*) (EROL,1995).

A área de planejamento em IA estuda a criação e aplicação de dois componentes principais de planejamento: (1) linguagens para representação de problemas e (2) planejadores (algoritmos) para resolver, automaticamente, esses problemas usando as informações disponíveis na sua representação (RUSSEL;NORVIG,1995). Linguagens de planejamento servem para descrever problemas, estados e ações num domínio.

A linguagem proposta aqui é denominada **Linguagem de Programação Cognitiva (LPC)**, uma linguagem para planejamento em programação capaz de representar componentes que generalizam uma linguagem tradicional de programação, num nível mais abstrato. A idéia é facilitar a atividade de planejamento do aprendiz de programação, disponibilizando planos de programação que, corretamente combinados, formam uma solução de problema. Dessa forma, o aprendiz torna-se muito mais um **agente de planejamento** do que um **agente de solução de problema**. Um agente de solução de problema busca por um conjunto de ações que, quando executadas, conduzem ao estado-meta: o estado final desejado. Um agente de planejamento faz busca num espaço de planos para alcançar a mesma finalidade. O domínio da programação pode ter infinitas ações e estados, tornando a tarefa do aprendiz extremamente difícil. Disponibilizando planos de programação, obtém-se uma sensível redução no espaço de busca, facilitando a tarefa de programação (EROL,1995).

Como mencionado, a linguagem LPC proposta neste capítulo, é um meio para representar componentes e atividades de planejamento em programação. Tais atividades são propostas para professores e alunos, ou seja, professor e aluno são os agentes que constróem o plano-solução. O objetivo da modelagem de solução de problemas, pelo professor, é habilitar o diagnóstico da solução proposta pelo aluno em sistemas de aprendizagem. Como já mencionado, um dos objetivos da área de pesquisa em Planejamento de IA é o de produzir planejadores: software que encontra o plano-solução, automaticamente, a partir da definição de um problema de planejamento. A idéia de formalizar ações de programação baseado em planejamento de IA tem a intenção de mostrar um caminho para automatizar a atividade do professor em sistemas que realizam diagnóstico, tais como Sistemas Tutores Inteligentes.

6.1 A Linguagem de Planejamento Hierárquico de IA

Um problema de planejamento clássico é caracterizado por (i) um domínio descrito por um conjunto de ações, (ii) um estado inicial e (iii) um estado meta, o qual se deseja alcançar executando-se uma seqüência de ações do domínio. O problema de planejamento consiste em encontrar esta seqüência, definida como: "um plano de ações que conduzem, a partir do estado inicial, ao estado-meta" (RUSSEL;NORVIG,1995). Planejamento hierárquico (Planejamento HTN) da Inteligência Artificial (*AI Planning*) define a atividade de planejamento como a de decompor as metas de problemas complexos em sub-metas, cujas soluções podem ser combinadas para fornecer um plano-solução, para o problema completo. Um domínio muito utilizado para exemplificar planejamento, é o chamado **Mundo dos Blocos**, o qual é utilizado aqui para exemplificar planejamento hierárquico. A linguagem de planejamento HTN de Erol (1995) é definida como uma tupla < V, C, P, F, T, N >, na qual os conjuntos são:

- V é um conjunto infinito de símbolos de **variáveis**.

- C é um conjunto de símbolos constantes que representam os **objetos** do mundo. Por exemplo: A, B e C são objetos do tipo bloco.

- P é um conjunto de símbolos **predicados** que representam as relações entre esses objetos. Por exemplo, **sobre**, **livre** e **sobre-a-mesa** são predicados para blocos, os quais podem ser usados como:
  - sobre(A, B) que significa "A está sobre B"
  - livre(A) que significa "A está livre"
  - sobre-a-mesa(C) que significa "C está sobre a mesa"

- F é um conjunto de símbolos de **tarefas primitivas** (*primitive tasks*) que representam as ações que podem ser realizadas no mundo. Tarefas primitivas são aquelas que podem ser alcançadas diretamente pela execução da ação correspondente. Uma tarefa-primitiva tem a forma do[$f(x_1, ..., x_i)$] onde $f \in F$ e $x_1, ..., x_i$ são termos. Exemplos:
  - mover(v_1,v_2,v_3) – "mover v_1 de v_2 para v_3"
  - empilhar(v_1,v_2) – "mover o bloco v_1 da mesa para v_2"
  - desempilhar(v_1,v_2) – "mover o bloco v_1 que está sobre v_2, para a mesa"

- T é um conjunto de símbolos de **tarefas compostas** (*compound tasks*), as quais permitem representar mudanças desejadas no mundo e que não podem ser representadas como uma única tarefa primitiva. Uma tarefa composta tem a forma $t(x_1,....,x_k)$, onde $t \in T$ e x_i são termos. Exemplo: t(livre(A), sobre(A,B)).

- $N = \{n_1, n_2, ...\}$ é um conjunto infinito de símbolos para rotular tarefas.

Definição 1: Um **estado** é uma lista de propriedades do mundo. Exemplo: **sobre-a-mesa**(A), **livre**(A) – "A está sobre a mesa e A está livre"

Definição 2: Um **operador** tem a forma $(f(v_1,....,v_k),l_1,...,l_m)$, onde f é uma tarefa primitiva, $v_1,....,v_k$ são variáveis e $l_1,...,l_m$ são literais que representam os efeitos da tarefa primitiva, capazes de gerar um estado novo no mundo.

Definição 3: Um **plano** é uma seqüência de tarefas primitivas instanciadas.
Exemplo:

do(desempilhar(B,C))
do(desempilhar(C,A))
do(empilhar(B,C))
do(mover(B,C,A))

Definição 4: **Tarefas-meta** (*goal tasks*) são propriedades as quais deseja-se que se tornem verdadeiras no mundo. Uma tarefa-meta tem a forma achieve(l), onde l é um literal.

Exemplo: achieve(livre(v_1)).

Definição 5: Uma **rede de tarefas** (*task network*) é uma coleção de tarefas e suas restrições na forma $((n_1 : \alpha_1),..., (n_m : \alpha_m), \phi)$ onde cada α_i é uma tarefa rotulada com n_i e ϕ é um conjunto de restrições. O rótulo distingui as possíveis ocorrências de uma mesma tarefa α_i na rede.

Três tipos de restrições podem ser modeladas numa rede de tarefas:

(i) **Restrições de ordem temporal** onde $n_i \pi n_j$ significa que a tarefa n_i deve ser realizada antes da tarefa n_j;

(ii) **Restrições de verdade** onde (n_i, l) significa que o estado l deve existir após a realização da tarefa n_i ou ainda, (n_i, l, n_j) que especifica que o estado l deve ser verdade após a realização da tarefa n_j e antes da tarefa n_i;

(iii) **Restrições de unificação** entre variáveis tal como $v_i = v_j$ e $v_i = A$.

A figura 18 mostra uma rede de tarefas do Mundo dos Blocos na qual n_1 é uma tarefa-meta "alcance v_1 livre", n_2 é uma tarefa-meta "alcance v_2 livre" e n_3 é uma tarefa primitiva "mova v_1 de v_3 para v_2". A linha 2 descreve as restrições de ordem de execução entre tarefas declarando que n_1 e n_2 devem ser realizadas antes de n_3. A linha 3 descreve as restrições de verdade que diz: v_1 deve estar livre depois da realização da tarefa n_1 e antes de n_3, v_2 deve estar livre depois da realização da tarefa n_2 e antes de n_3 e finalmente, v_1 deve estar sobre v_3 antes da realização de n_3. A linha 3 descreve as restrições de unificação que garante que v_1, v_2 e v_3, quando instanciadas, correspondem a três blocos distintos.

```
1:  (n1: achieve(livre(v1))) (n2: achieve(livre(v2))) (n3: do(mover(v1,v3,v2)))
2:  (n1 π n3) ∧ (n2 π n3) ∧
3:  (n1, livre(v1), n3) ∧ (n2, livre(v2), n3) ∧ (sobre(v1, v3), n3) ∧
4:  ¬ (v1=v2) ∧¬ (v1=v3) ∧¬ (v2=v3))
```

Figura 18 – Uma rede de tarefas do mundo dos blocos (Erol et al., 1992)

Definição 6: Uma rede de tarefas contendo somente tarefas primitivas é chamada uma **rede de tarefa primitiva**.

Definição 7: Um **método** consiste da especificação de uma das maneiras de se realizar uma tarefa não-primitiva e tem a forma (α, d) na qual α é uma tarefa não primitiva e d é uma rede de tarefas. O método especifica que para realizar α, a rede d deve ser executada, ou seja, executar suas tarefas, respeitando-se as restrições.

Sejam, por exemplo, o **estado inicial do mundo** e o **estado meta** a ser alcançado como apresentado na figura 19. A tarefa composta t consiste em alcançar A sobre B e A livre.

Figura 19 – Descrição de um problema no mundo dos blocos

Um método capaz de realizar a tarefa composta t é (t, *d*), onde *d* é a rede de tarefas da figura 18. Realizando as tarefas da rede *d*, segundo suas restrições, é possível encontrar pelo menos um plano que realiza a tarefa composta t.

Finalmente um problema de planejamento hierárquico é a tripla <d, I, D> em que, d é uma rede de tarefas de entrada, I é o estado inicial do mundo e D é um domínio de planejamento consistindo de um conjunto de métodos (tarefas e redes de tarefas). Assim, um planejador hierárquico (sofware) é capaz de executar a rede d, a partir do estado inicial e encontrar uma seqüência de ações (um plano) que conduzem ao estado-meta desejado.

6.2 Planejamento no Domínio de Programação

A proposta para habilitar planejamento em programação consiste em modelar ações de planejamento hierárquico, segundo o formalismo de Erol, para o domínio específico da programação algorítmica. Isto pode ser realizado a partir da linguagem de planejamento hierárquico para programação proposta nesta seção, denominada **Linguagem de Programação Cognitiva** (**LPC**). Os componentes que constituem o sistema TutorC, apresentado no Capítulo 7, estão baseados na linguagem LPC.

A **Linguagem de Programação Cognitiva** define o domínio de programação algorítmica por meio de três tipos de tarefas: tarefas problema, tarefas meta e tarefas primitivas. **Tarefas problema** correspondem às tarefas compostas de Erol e que no mundo de programação representam os problemas de programação. **Tarefas meta** correspondem aos objetivos a serem alcançados para resolver um problema de programação, tal como "um número obtido do usuário" ou "uma mensagem de erro na tela". **Tarefas primitivas** descrevem as ações primitivas de programação, tal como "ler um número do teclado" ou "emitir uma mensagem na tela". Tarefas problema e tarefas meta são realizadas por **métodos.** Segundo a linguagem proposta por Erol, métodos são pares que relacionam tarefas à rede de tarefas. Na LPC, rede de tarefas são de dois tipos:

(1) **Rede de Tarefas Meta** que consiste da decomposição de uma tarefa problema em tarefas meta. Um problema de programação pode ser decomposto de diferentes maneiras, onde cada uma delas define um modo diferente de alcançar a solução do problema. **Rede de tarefas meta** é o termo formal do chamado **Modelo de Problema** definido no Capítulo 5.

(2) **Rede de Tarefa Primitiva** ou **Plano de Programação** que consiste de um conjunto de tarefas primitivas ou ações primitivas de programação seqüencialmente ordenadas. Uma rede de tarefa primitiva define um modo de alcançar uma tarefa meta.

A figura 20 apresenta uma tarefa problema decomposta segundo o planejamento hierárquico.

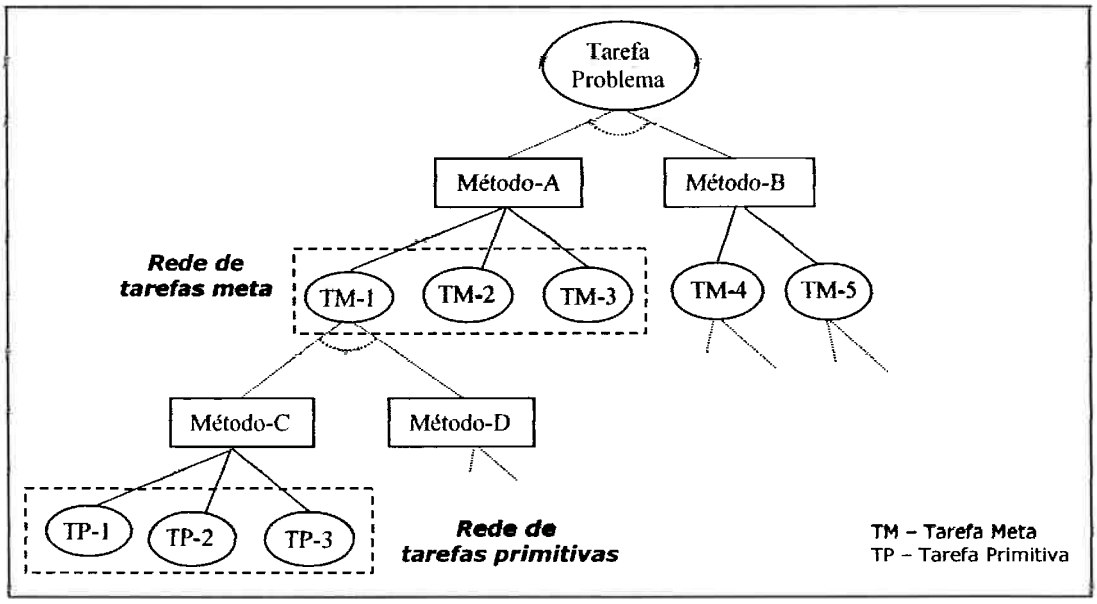


Figura 20 – Conceitos e relações na LPC

Na figura 20, arcos denotam caminhos alternativos. No topo da árvore está a tarefa problema que pode ser resolvida por duas maneiras diferentes, usando o método-A ou o método-B. O método-A decompõe a tarefa problema em três tarefas meta e o método-B decompõe a mesma tarefa problema em duas tarefas meta. Assim, a tarefa problema da figura 20 possui duas redes de tarefas meta. Para compor o plano-solução (o programa) para a tarefa problema é necessário realizar uma das redes de tarefas meta. Isto significa realizar todas as tarefas meta da rede de tarefas selecionada. Por exemplo, a tarefa meta 1 (TM-1) pode ser realizada, alternativamente, por dois métodos: método-C ou método-D. Cada um dos métodos define uma rede de tarefas primitivas de programação ou plano de programação. Os conceitos da LPC, apresentados até aqui, estão formalizados na próxima seção.

6.3 Descrição da Linguagem de Programação Cognitiva

O vocabulário da Linguagem de Programação Cognitiva, é definida como:

$\mathcal{L}$ = < V, C, P, T_M, T_P, N > em que,

V = $\{v_1, v_2, ...\}$ é um conjunto infinito de variáveis

C = {if, else, for,..} é um conjunto de símbolos da linguagem C

P = $\{p_1, p_2, ...\}$ é um conjunto de símbolos de tarefas problemas

T_M = $\{t_{M1}, t_{M2}, ...\}$ é um conjunto de símbolos de tarefas meta

T_P = $\{t_{P1}, t_{P2}, ...\}$ é um conjunto de símbolos de tarefas primitivas de programação

N = $\{n_1, n_2, ...\}$ é um conjunto infinito de símbolos para rotular tarefas

Com a LPC, deseja-se modelar duas partes principais no domínio da programação: o problema e a solução para esse problema. Tal solução, como é objetivo nesta tese, deve ser descrita desde o nível de planejamento até o nível da implementação, onde obtém-se o código do programa. A seção 6.3.1 apresenta como modelar descrições de problemas e a seção 6.3.2 apresenta como modelar soluções de problema.

6.3.1 Modelando a Descrição de Problema

Considere o seguinte problema:

Elabore um programa que calcula e imprime a tabuada t desejada pelo usuário.

Problema 4

Definição 1: Uma tarefa problema tem a forma $\mathbf{p_i(t_1,....,t_k)}$ onde $p_i \in P$ e t_i é um parâmetro de entrada do problema.

Para o problema 4, a tarefa problema pode ser especificada pelo professor como:

calcular_tabuada (t, 0, 10)

Definição 2: Uma **rede de tarefas meta** é uma coleção de tarefas meta e suas restrições, na forma $((n_1 : \text{achieve } (m_1)) \wedge ... \wedge (n_k : \text{achieve } (m_k)) \wedge \phi)$ em que cada achieve(m_i) é uma tarefa meta rotulada com n_i e ϕ é um conjunto de restrições. M_i é a declaração de uma das metas que devem ser realizadas para alcançar a solução do problema.

Uma possível **rede de tarefas meta** para o problema 4 é:

$(n_1$:achieve$(m_1) \wedge n_2$:achieve$(m_2) \wedge n_3$:achieve$(m_3) \wedge$
$(n1 \pi n2) \wedge (n2 \pi n3))$
em que,

- m_1 é "variáveis simples declaradas"
- m_2 é "tipo de tabuada obtida do usuário"
- m_3 é "tabuada calculada"
- o conjunto ϕ de restrições informa que a tarefa meta n_1 deve ser realizada antes de n_2 e que n_2 deve ser realizada antes de n_3.

Assim, o problema 4 é especificado pela tarefa problema e pela rede de tarefas meta, tal como:

(1) (calcular_tabuada (t, 0, 10),
(2) $(n_1$:achieve$(m_1)) \wedge (n_2$:achieve$(m_2)) \wedge (n_3$:achieve$(m_3))$
(3) $(n_1 \pi n_2) \wedge (n_2 \pi n_3))$

Segundo a linguagem proposta por Eroi, **métodos** são pares que relacionam tarefas às redes de tarefas na forma (α, *d*), na qual α é uma tarefa não primitiva e *d* é uma rede de tarefas. Assim a descrição do problema 4, acima, define um método para resolver esse problema. Para o propósito da implementação dos sistemas de ensino e aprendizagem propostos no Capítulo 7, um professor deve especificar pelo menos uma rede de tarefas meta para um problema. Porém, mais redes de tarefas meta podem ser especificadas para o mesmo problema. Dessa forma, o estudante pode escolher qual das redes deseja implementar. Note que o conjunto de restrições ϕ (linha 3), o qual declara a ordem em que as metas devem ser alcançadas, é uma informação a ser utilizada apenas para diagnosticar a solução do estudante, isto é, esta informação não é apresentada ao estudante, uma vez que a aquisição desse conhecimento é um dos objetivos de aprendizagem.

6.3.2 Modelando a Solução de Problema

Conforme proposto no modelo de processo de construção de programas descrito no Capítulo 5, deseja-se que o estudante construa a solução para um problema de programação a partir das metas de um problema, as quais devem ser realizadas por meio de ações primitivas de programação especificadas em planos de programação. Assim, LPC fornece algumas definições adicionais que habilitam a representação formal desses conceitos.

Definição 3: Uma **tarefa primitiva de programação** tem a forma do(t_{Pk}(p_{pi}, ação, esquema)) em que $\mathbf{t_{Pk}}$ corresponde à identificação da tarefa primitiva ($t_{pk} \in T$), $\mathbf{p_{pi}}$ é o plano de programação ao qual a tarefa primitiva pertence, **ação** descreve a ação de programação da tarefa primitiva em português e **esquema** corresponde à implementação da ação numa linguagem de programação particular.

Exemplos de tarefas primitivas de programação são:

do (t_{P1} (p_{p3}, *emitir mensagem para o usuário,* printf ($formato-de-entrada)))
do (t_{P2} (p_{p3}, *obtém dado do usuário,* scanf ($formato-de-entrada, $endereço)))

Definição 4: Uma **rede de tarefas primitivas** ou **plano de programação** define um meio de realizar uma tarefa meta e é descrito na forma p_{pi}: (n_1 : do(t_{P1}),..., n_k: do (t_{Pk}), ϕ) em que do(t_{Pi}) é uma tarefa primitiva rotulada com n_i e ϕ é um conjunto de restrições entre essas tarefas.

No problema 4, a tarefa meta **achieve($\mathbf{m_2}$)** pode ser realizada pelo plano "entrada por teclado", descrito como:

(n_1: do (t_{P1})) $\wedge$ (n_2: do (t_{P2})) $\wedge$ (n_1 π n_2) em que t_{P1} e t_{P2} são:

t_{P1} (p_p, *emitir mensagem para o usuário,* printf ($formato-de-entrada))
t_{P2} (p_p, *obtém dado do usuário,* scanf ($formato-de-entrada, $endereço))

A restrição (n_1 π n_2) garante a ordenação correta das linhas de código para o plano "entrada por teclado". Assim, a realização dessa rede de tarefas primitivas conduz ao seguinte esquema de código:

```
printf ($formato-de-entrada);
scanf ($formato-de-entrada, $endereço);
```

Na implementação do sistema TutorC, Capítulo 7, planos de programação possuem informações adicionais tais como tópico, nome e situação, os quais encontram-se detalhados no Capítulo 5. Considere agora o problema 5 e a rede de tarefas meta da figura 21.

Elabore um programa que calcula a tabuada t desejada pelo usuário. Após o cálculo de uma tabuada, o usuário poderá solicitar nova tabuada, enquanto não digitar a letra F.

Problema 5

n_1:achieve(m_1) ∧ n_2:achieve(m_2) ∧ n_3:achieve(m_3) ∧ n_4:achieve(m_4) ∧ n_5:achieve(m_5)
(n_1 π n_3) ∧ (n_2 π n_3) ∧ (n_3 π n_4) ∧ (n_4 π n_5)

Figura 21 – Rede de Tarefas Meta para o Problema 5

Na rede de tarefas meta da figura 21, a descrição das metas é:

- m1: "Espaço alocado na memória para armazenar inteiros"
- m2: "Espaço alocado na memória para armazenar caracter"
- m3: "Controle da execução de nova tabuada, através do usuário"
- m4: "Tipo de tabuada obtida do usuário"
- m5: "Tabuada calculada por processamento repetitivo"

Para resolver o problema, o estudante deve selecionar planos de programação para a rede de tarefas meta escolhida. Exemplos de planos de programação para a realização da rede de tarefas meta do problema 5 são apresentados na tabela 26.

Tabela 26 – Planos de Programação para a Rede de Tarefas Meta do Problema 5

PLANOS DE PROGRAMAÇÃO	
P_{p1}:	Declaração de variáveis simples do (t_{P1} (p_{p1}, declara variaveis, $tipo $variaveis))
P_{p2}:	Repetição com condição de continuação controlada pelo usuário do (t_{P2} (p_{p2}, inicia corpo de ações para repetição, do {)) do (t_{P3} (p_{p2}, pergunta se o usuário (não) deseja parar repetição, printf ($formato-de-entrada))) do (t_{P4} (p_{p2}, obtém caracter digitado pelo usuário, tecla=getch())) do (t_{P5} (p_{p2}, insere condição para repetição geral, } while ($condição))) ($t_{P2}$ π t_{P3} π t_{P4} π t_{P5})
P_{p3}:	Entrada por teclado do (t_{P6} (p_{p3}, emitir mensagem para o usuário, printf ($formato-de-entrada))) do (t_{P7} (p_{p3}, obtém dado do usuário, scanf ($formato-de-entrada, $endereços))) ($t_{P6}$ π t_{P7})
P_{p4}:	Processamento seqüencial conta vezes do (t_{P8} (p_{p4}, inicia contagem, contador=0)) do (t_{P9} (p_{p4}, insere condição para repetição geral, while (contador < $vezes))) do ($t_{P10}$ (p_{p4}, inicia corpo de ações para condição verdadeira, {)) do (t_{P11} (p_{p4}, insere expressão para cálculo, $resultado = $expressão)) do ($t_{P12}$ (p_{p4}, mostra resultado, printf ($formato-de-entrada, $resultado))) do ($t_{P13}$ (p_{p4}, incrementa contador, contador++;)) do (t_{P14} (p_{p4}, finaliza corpo de ações while, })) (t_{P8} π t_{P9} π t_{P10} π t_{P11} π t_{P12} π t_{P13} π t_{P14})

A associação de uma tarefa meta a um plano de programação define um método para

resolver essa tarefa. Com relação aos sistemas propostos no próximo capítulo, métodos para tarefas meta são também para serem declarados pelo professor a fim de habilitar diagnóstico da solução do estudante, no TutorC. Trabalhando no TutorC, o aluno realiza as associações desejadas, as quais são comparadas com os métodos declarados pelo professor na BibPC. Os métodos obtidos para a rede de tarefas do problema 5, usando os planos de programação da tabela 26, são:

- (achieve(m_1), P_{p1})
- (achieve(m_2), P_{p1})
- (achieve(m_3), P_{p2})
- (achieve(m_4), P_{p3})
- (achieve(m_5), P_{p4})

Uma vez que o aluno tenha associado planos de programação para a rede de tarefas meta escolhida, este deve, então, ordenar os planos de programação entre si, a fim de estabelecer uma ordem correta que resulta no chamado **esquema do programa**, mencionado no Capítulo 5.

Como mencionado anteriormente, analisando soluções de problemas (programas) no domínio da programação, especialmente na construção de algoritmos, observa-se a existência dos chamados **planos deslocalizados** (SOLOWAY;LETOVSKY,1986; LAMPERT ET AL.,1988). A solução parcial do problema 5, apresentada na tabela 27, permite verificar essa característica no plano P_{P2}.

Tabela 27 – Esquema do Programa para o Problema 5

MÉTODOS		**ESQUEMA DO PROGRAMA**	
Tarefa Meta	**Plano**	**T. Primitiva**	**Esquema Primitivo**
n1: achieve(m_1)	P_{P1}	t_{P1}	`$tipo $variaveis;`
n2: achieve(m_2)	P_{P1}	t_{P1}	`$tipo $variaveis;`
n3: achieve(m_3)	P_{P2}	t_{P2}	`do{`
n4: achieve(m_4)	P_{P3}	t_{P6} t_{P7}	`printf ($formato-de-entrada);` `scanf ($formato-de-entrada, &$variavel);`

n5: achieve(m_5)	P_{P4}	t_{P8}
		t_{P9}
		t_{P10}
		t_{P11}
		t_{P12}
		t_{P13}
		t_{P14}
			`contador=0;`
			`while (contador <= $vezes)`
			`{`
			`resultado=contador * $variavel;`
			`printf ($formato-de-entrada, resultado);`
			`contador++;`
			`}`

n3: achieve(m_3)	P_{P2}	t_{P3}
		t_{P4}
		t_{P5}
			`printf($formato-de-saída);`
			`letra=getch();`
			`} while ($condição);`

Pode-se notar que o plano de programação P_{P2} é um plano deslocalizado nessa solução

de problema. P_{P2} está intercalado pelos planos P_{P3} e P_{P4}. Para representar ordenação de planos, LPC modela restrições de ordem temporal de dois tipos:

- **Restrições entre tarefas-meta**: usada para especificar redes de tarefas meta como exemplificado na figura 21. Uma vez que tarefas-meta estão associadas a planos de programação, tais restrições são válidas para a ordenação dos planos.
- **Restrições entre tarefas primitivas e tarefa-meta**: usada para ordenar planos deslocalizados. Indica entre quais tarefas primitivas localiza-se o plano associado à uma certa tarefa-meta.

Para representar, por exemplo, o esquema do programa apresentado na tabela 27, as restrições a serem modeladas são: $(n_1 \pi n_3) \wedge (n_2 \pi n_3) \wedge (n_3 \pi n_4) \wedge (n_4 \pi n_5) \wedge (t_{P2} \pi n_4 \pi t_{P8}) \wedge (t_{P7} \pi n_5 \pi t_{P3})$

A restrição $(t_{P2} \pi n_4 \pi t_{P8})$ indica que o plano P_{P3} deve estar entre as tarefas primitivas t_{P2} e t_{P8}. A restrição $(t_{P7} \pi n_5 \pi t_{P3})$ indica que o plano P_{P4} deve estar entre as tarefas primitivas t_{P7} e t_{P3}. Apesar desta proposta de representação, tanto professor como aluno, especificam esquemas de programa nos sistemas propostos no Capítulo 7, ordenando diretamente as linhas de código por meio de uma interface. Tal como a linguagem de Erol, LPC define um plano como uma seqüência de tarefas primitivas instanciadas. Assim, o esquema do programa instanciado conduz ao plano-solução, ou seja, o programa propriamente dito. A tabela 28 apresenta o programa para o problema 5.

Tabela 28 – O Plano-Solução no nível da implementação em LPC

TP	ESQUEMA DO PROGRAMA	PLANO-SOLUÇÃO: O PROGRAMA
t_{P1}	`$tipo $variaveis;`	`int contador, numero, resultado;`
t_{P1}	`$tipo $variaveis;`	`char tecla;`
t_{P2}	`do{`	`do{`
t_{P6} t_{P7}	`printf ($formato-de-entrada);` `scanf ($formato-de-entrada, &$variavel);`	`printf("Qual tabuada ?");` `scanf ("%f", &numero);`

t_{P8}
t_{P9}
t_{P10}
t_{P11}
t_{P12}
t_{P13}
t_{P14}
	`contador=0;`
	`while (contador <= $vezes)`
	`{`
	`resultado=contador*$variavel;`
	`printf ($formato-de-entrada, resultado);`
	`contador++;`
	`}`
		`contador=0;`
		`while (contador<=10)`
		`{`
		`resultado=contador*numero;`
		`printf ("%i", resultado);`
		`contador++;`
		`}`

t_{P3}
t_{P4}
t_{P5}
	`printf($formato-de-saída);`
	`letra=getch();`
	`} while ($condição);`
		`printf("Digite F para sair");`
		`letra=getch ( );`
		`} while (letra!='F');`

Uma vez que o objetivo desta proposta é apoiar a criação de estruturas cognitivas no

aprendiz, deseja-se enfatizar a construção do plano-solução no nível cognitivo, isto é, acima do nível da implementação. Assim, a definição 5 trata do chamado **Plano Cognitivo.**

Definição 5: Plano Cognitivo é o plano-solução representado como uma seqüência ordenada de ações primitivas de programação, descritas no nível acima da implementação.

Uma vez que tarefas primitivas são compostas de descrições de ações primitivas de programação, na forma de texto, as restrições modeladas para as tarefas primitivas, permitem representar o plano-solução, no nível cognitivo, como mostra a tabela 29. O vocabulário, utilizado informalmente pelo professor em sala de aula, é utilizado aqui para construir o plano cognitivo.

Tabela 29 – O Plano-Solução no Nível Cognitivo em LPC

PLANO COGNITIVO
t_{P1}: declara variáveis
t_{P1}: declara variáveis
t_{P2}: inicia repetição
t_{P6}: emite mensagem para o usuário
t_{P7}: obtém dado do usuário
t_{P8}: inicia contagem
t_{P9}: insere condição para repetição geral
t_{P10}: inicia corpo de ações para condição verdadeira
t_{P11}: insere expressão para cálculo
t_{P12}: mostra resultado
t_{P13}: incrementa contador
t_{P14}: finaliza corpo de ações while
t_{P3}: pergunta se o usuário (não) deseja parar repetição
t_{P4}: obtém caracter digitado pelo usuário
t_{P5}: insere condição para repetição geral

A tabela 30 mostra a relação entre o **plano cognitivo** e o **programa** construídos para o problema 5.

Tabela 30 – Esquema do Programa para a Tarefa Problema 5

TP	PLANO COGNITIVO	PROGRAMA
t_{P1}	Declara variáveis	`int contador, numero, resultado;`
t_{P1}	Declara variáveis	`char tecla;`
t_{P2}	Inicia repetição	`do{`
t_{P6}	Emite mensagem para o usuário	`printf("Qual tabuada ?");`
t_{P7}	Obtém dado do usuário	`scanf ("%f", &numero);`
t_{P8}	Inicia contagem	`contador=0;`
t_{P9}	Insere condição para repetição geral	`while (contador<=10)`
t_{P10}	Inicia corpo de ações para condição verdadeira	`{`
t_{P11}	Insere expressão para cálculo	`resultado=contador*numero;`
t_{P12}	Mostra resultado	`printf ("%i", resultado);`
t_{P13}	Incrementa contador	`contador++;`
t_{P14}	Finaliza corpo de ações while	`}`
t_{P3}	Pergunta se o usuário (não) deseja parar repetição	`printf("Digite F para sair");`
t_{P4}	Obtém caracter digitado pelo usuário	`letra=getch ( );`
t_{P5}	Insere condição para repetição geral	`} while (letra!='F');`

Finalmente um problema de planejamento hierárquico no domínio da programação é definido pela tripla <d, I, D> em que, d é uma rede de tarefas meta, I é o estado inicial do mundo (um programa vazio) e D é um domínio de planejamento para programação consistindo de um conjunto de redes de tarefas primitivas ou planos de programação.

6.4 Resumo do Capítulo

Neste capítulo é apresentada a LPC (Linguagem de Programação Cognitiva), uma linguagem para representação de componentes e tarefas no domínio de programação baseada na abordagem de Planejamento Hierárquico da IA, proposta por Erol (1995). Um dos objetivos da área de pesquisa em Planejamento de IA é a de produzir planejadores: software que encontra o plano-solução, automaticamente, a partir da definição de um problema de planejamento. Nesta tese, LPC é proposta como um meio para representação formal do Modelo de Processo de Construção de Programas descrito no Capítulo 5, com a intenção de automatizar tal processo em sistemas de aprendizagem. O modelo contempla atividades de planejamento em programação propostas para professor e aluno, ou seja, ambos constróem o plano-solução. O objetivo da modelagem de métodos de solução de problemas de programação, pelo professor, é habilitar o diagnóstico da solução proposta pelo aluno. A idéia de formalizar tais atividades baseado em planejamento de IA tem a intenção de mostrar um caminho para automatizar a atividade do professor no projeto de sistemas de aprendizagem que realizam diagnóstico, tais como Sistemas Tutores Inteligentes (STI).

A linguagem LPC é capaz de representar componentes de programação num nível alto de abstração, acima do nível da implementação. Dessa forma é possível construir a solução para um problema de programação utilizando-se componentes cognitivos, que vão além do código de uma linguagem de programação. **Planos de programação** são componentes chave representados na linguagem LPC. Eles desempenham o papel de subplanos que podem ser combinados a fim de representar um **plano-solução** para um problema de programação, ou seja, o **programa**. A decomposição dos planos de programação, assim como a decomposição de um problema num conjunto de metas, como apresentado no Capítulo 5, estão representadas pela linguagem LPC por meio das chamadas **Redes de Tarefas Meta** e **Redes de Tarefas Primitivas**. Uma das razões da representação das decomposições é a necessidade de se representar **planos deslocalizados**, característica existente no domínio da programação. Outra razão é a de permitir a representação do processo de solução de um problema em vários níveis de abstração:

- **Nível das Metas do Problema**, representadas em LPC, por uma Rede de Tarefas Meta;
- **Nível dos Planos de Programação** que realizam tarefas metas. Em LPC a associação de um plano a uma tarefa meta é denominado **método** da tarefa meta;

- **Nível das Ações Primitivas** dos planos que representam a decomposição do plano na forma de uma **Rede de Tarefas Primitivas** e finalmente;

- **Nível da Implementação**, no qual a solução do problema, o plano-solução, é representado por uma seqüência de instruções instanciadas, escritas por meio de uma linguagem de programação.

Nesta tese não foi implementado um software que encontre o plano-solução automaticamente, porém a linguagem LPC é proposta neste trabalho como um meio para habilitar tal automatização. Por exemplo, o professor poderia inserir na BibPC, somente a Rede de Tarefas Meta e as restrições entre tarefas-meta e tarefas primitivas. A partir dessas informações, BibPC poderia executar as restrições obtendo o esquema do programa automaticamente.

7 IMPLEMENTAÇÃO DO MODELO PROPOSTO: BIBPC E TUTORC

No Capítulo 5 é proposto um modelo do processo de construção para aprendizes, o qual define componentes e estratégias para planejamento e implementação de programas numa abordagem algorítmica. O capítulo 6 define a Linguagem de Programação Cognitiva que permite formalizar os componentes e estratégias do modelo, fornecendo uma base sólida para sua implementação. Este capítulo descreve os aspectos de implementação do modelo proposto.

Uma vez que os atores do modelo do processo de construção são de dois tipos: professor e aprendiz, idealizou-se dois sistemas: a BibPC (LEMOS ET AL., 2003b) e o TutorC (LEMOS; BARROS, 2003), conforme mencionado no Capítulo 5. O usuário de BibPC é o professor, o qual modela componentes de programação cognitiva a serem utilizados, automaticamente, pelo TutorC. O aprendiz é usuário de TutorC que disponibiliza em sua interface, componentes da BibPC, além de guiar o aprendiz numa estratégia de programação definida no modelo proposto. A seção 7.1 detalha a implementação de BibPC e a seção 7.2 detalha a implementação de TutorC.

7.1 A Biblioteca para Programação Cognitiva

BibPC é composta por (1) uma interface WEB, denominada **Interface do Professor** e (2) uma estrutura de banco de dados SQL. A interface permite que professores ou programadores experientes possam modelar componentes de planejamento e de implementação de programas, os quais são armazenados no banco de dados.

Para a implementação de BibPC foram escolhidas linguagens e ambientes de programação para WEB: PHP, Java Script, HTML, SQL. A interface está localizada no servidor Júpiter da FACENS em conjunto com o banco de dados SQL e acessível pelo link http://www.li.facens.br/~tutor. Para visualização do conteúdo atual da BibPC deve ser usado o nome de usuário **visitante** e a senha **visita**.

O conhecimento modelado em BibPC pode ser utilizado, de forma manual, por professores em sala de aula ou acessado automaticamente por um ambiente de aprendizagem, tal como TutorC. No trabalho descrito nesta tese, o conhecimento especialista de BibPC foi utilizado de ambas as formas. Os principais componentes modelados por um professor na BibPC são:

- Planos de Programação,
- Problemas,
- Soluções de problemas.

7.1.1 O Modelo de Banco de Dados

O banco de dados, representado na figura 22 por um diagrama Entidade-Relacionamento (E-R) (KORTH;SILBERCHATS,1995), foi estruturado para armazenar os componentes de planejamento e de implementação de programas mencionados nos capítulos 5 e 6.

Tabela 31 – Tabelas do Banco de Dados da BibPC

Tabela	Informações que armazena
Tópico	tópicos de um currículo desejado para estudo
Nível	subdivisões dos tópicos existentes
Plano	descrições de planos de programação e sua classificação no currículo
T_Primitiva	as tarefas primitivas que compõem um plano
T_Problema	o enunciado e a classificação dos problemas propostos
Método_PB	métodos para um problema
Rede_TM	redes de tarefas meta de um problema
Método_TM	métodos para uma meta de uma rede de tarefas meta
Soluções	dados auxiliares para definir um método para um problema
Esquema_PG	ordem dos esquemas primitivos de um método para um problema
Código_Fonte	*esquemas primitivos instanciados para um problema*

As tabelas **Plano** e **T_Primitiva** consistem das Redes de Tarefas Primitivas formalizadas no Capítulo 6. A tabela **Método_PB** e a tabela **Rede_TM** definem soluções alternativas para um problema. Um enunciado pode ser decomposto de maneiras diferentes pelo professor, tal como um conjunto de metas mais globais ou um conjunto maior de metas mais detalhadas. Como visto no Capítulo 6, cada método proposto para um problema define uma maneira diferente de resolver esse problema, ou seja, as diferentes visões que o professor pode ter a respeito da decomposição de metas para um problema.

As tabelas **Esquema_PG** e **Código_Fonte** contêm as definições restantes do método de resolução de um problema até o nível da implementação. Isto inclui, principalmente, códigos que localizam elementos em outras tabelas e que permitem formar o **plano cognitivo** (descrições textuais das tarefas primitivas ordenadas), **o esquema do programa** e o **programa** propriamente dito.

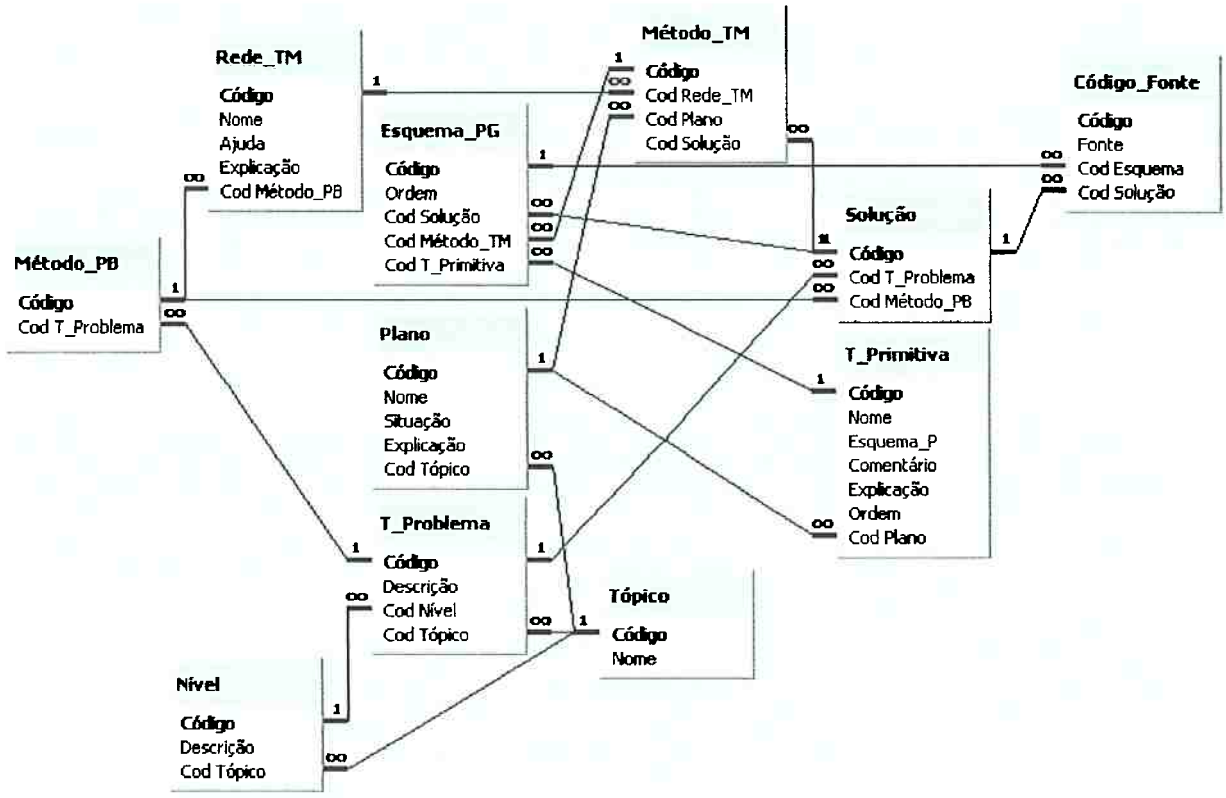


Figura 22 - Modelo E-R do Banco de Dados do sistema BibPC

Neste trabalho, o conceito de um método de solução para um problema envolve não apenas o código do programa (seqüência de linhas de código), mas também um conjunto relacionado de metas, planos e ações primitivas de programação que caracterizam a solução num nível de abstração acima da implementação. A próxima seção descreve de que forma um professor ou usuário da BibPC pode acessá-la para fins de modelagem e consulta.

7.1.2 A Interface do Professor

A função da **Interface do Professor** é facilitar as operações básicas de inserção, alteração e exclusão de planos de programação, problemas e suas soluções no banco de dados. Com um visual amigável, a **Interface do Professor** está disponível para acesso por qualquer computador conectado à **Internet**, desde que os usuários estejam cadastrados. Para uma idéia da interface, algumas funções são discutidas a seguir e algumas telas apresentadas.

7.1.2.1 Atividades do Professor-Coordenador na BibPC

Um professor-coordenador é responsável por três importantes tarefas na BibPC:

- Cadastramento de usuários, possivelmente um conjunto de professores;
- Divisão do curriculum a ser estudado, uma vez que planos e problemas formam o material instrucional e devem ser enquadrados em tais divisões;
- Inserção e manutenção de planos de programação;
- Manutenção de problemas e suas soluções.

O professor-coordenador possui habilitação, através de senha, para executar as tarefas acima. No cadastramento de usuários, o professor-coordenador pode definir tipos diferentes de usuários segundo o nível de permissão atribuída (1, 2, 3 ou 4). Os usuários típicos da BibPC numa instituição de ensino são professores da área de Computação. A figura 23 mostra a tela de cadastramento de usuários da BibPC.

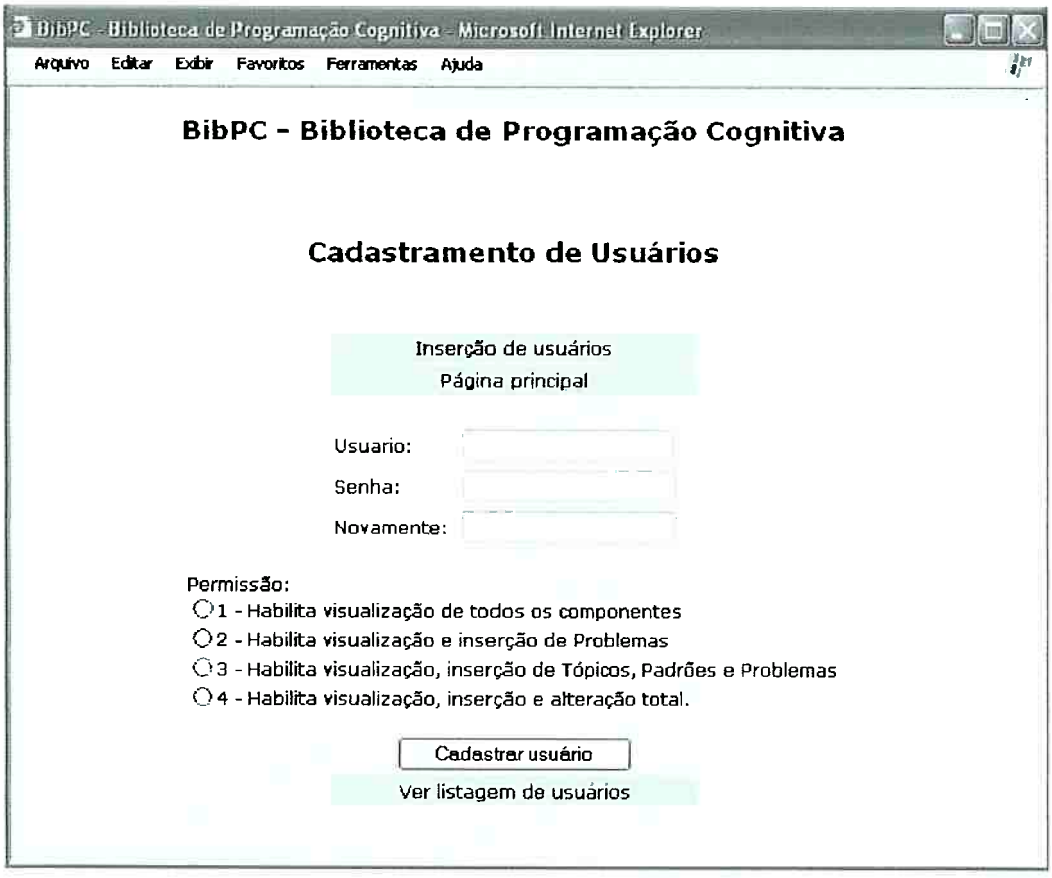


Figura 23 – Tela para Cadastramento de Usuários da BibPC

Outra tarefa atribuída ao professor-coordenador se refere à divisão do currículo a ser estudado. Sempre que um novo plano ou um novo problema é adicionado na BibPC, ambos devem ser classificados nesse currículo.

O currículo pode ser dividido em tópicos e estes em níveis. Os tópicos são também enquadrados em classes numéricas de um a dez, as quais podem ser interpretadas e usadas como um indicativo de grau de dificuldade. Essa divisão mostrou-se suficiente para classificar planos e problemas de programação, uma vez que a abordagem de aprendizado proposta é a abordagem baseada em problemas (PBL - *Problem Based Learning*) (BEAUMONT;SACKVILLE, 2002). Nessa abordagem, a tarefa básica do aprendiz é a de resolver problemas, ou seja, construir programas (VALENTE,1995).

Os componentes da BibPC têm a finalidade de serem utilizados como apoio ao processo de resolução de problemas de programação. Assim, acredita-se que tais componentes são melhor aproveitados se o aprendiz possuir conhecimentos básicos de conceitos isolados de programação tais como tipos de dados, variáveis, operadores, funcionamento de estruturas de fluxo e outros temas. A figura 24 mostra a tela de inserção de tópicos de curriculum e seus níveis.

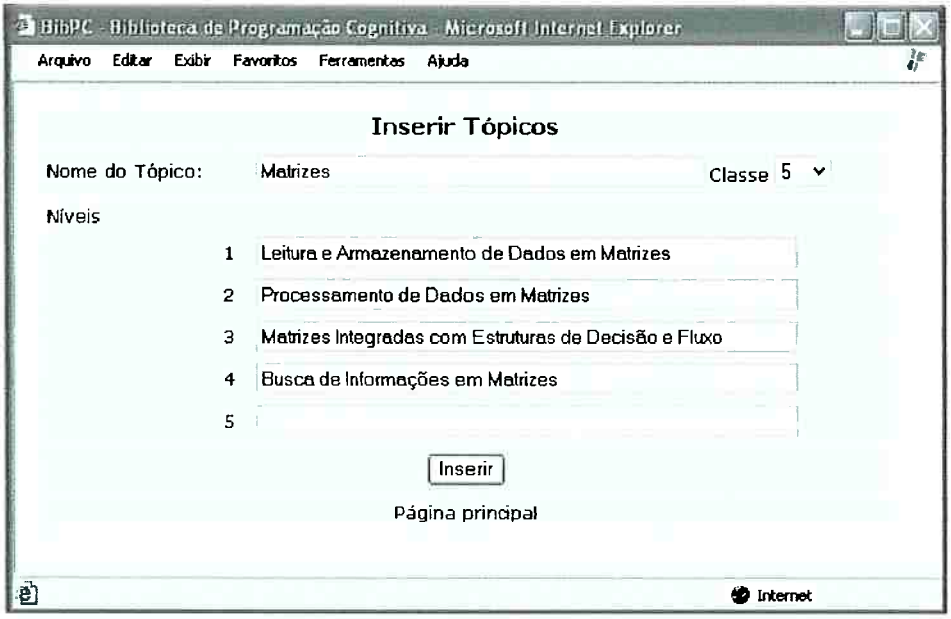


Figura 24 – Tela de Inserção de Tópicos do Curriculum

Outra tarefa do professor-coordenador refere-se à inserção e manutenção dos planos de programação no sistema. Uma vez que a exclusão ou modificação de um plano exige a exclusão dos problemas que o utilizam, essas tarefas devem ser atribuídas a um professor que administra o sistema. A tabela 32 apresenta o algoritmo das ações principais do professor-coordenador na tarefa de inserção de planos de programação.

Tabela 32 - Algoritmo das Ações do Professor-Coordenador na BibPC

Tarefa	***inserir_plano***
Objetivo	algoritmo que realiza a inserção de um plano de programação
Papéis de entrada	nome; situação; $Esquema_{plano}$: chunk de programação; Topico: conjunto de tópicos do curriculum
Papéis de saída	plano; RTP: Rede de Tarefas Primitivas
Sub-tarefas	***criar_plano, decompor, criar_tarefa_primitiva***
Papéis internos	RTP: Rede de Tarefas Primitivas; esquema $_{primitivo}$: parte do esquema do plano; nome: descrição em linguagem natural da ação da tarefa primitiva
Estrutura de Controle	***inserir_plano*** (nome, situação, $Esquema_{plano}$, Tópico → plano, RTP) = ***criar_plano*** (nome, situação, Tópico, Nível → plano) ***decompor*** ($Esquema_{plano}$ → RTP) = para cada $esquema_{primitivo} \in Esquema_{plano}$ ***criar_tarefa_primitiva*** (nome, $esquema_{primitivo}$ → tarefa-primitiva) RTP ← RTP ∪ tarefa-primitiva

As figuras 25 e 26 mostram telas para inserção de planos de programação assim como sua decomposição em tarefas primitivas. O exemplo mostra a inserção do plano **escolha-não-relacionada** do tópico **Seleção**. O campo "explicação" é utilizado para armazenar explicações a serem mostradas, ao aprendiz, durante a fase de diagnóstico de sua solução.

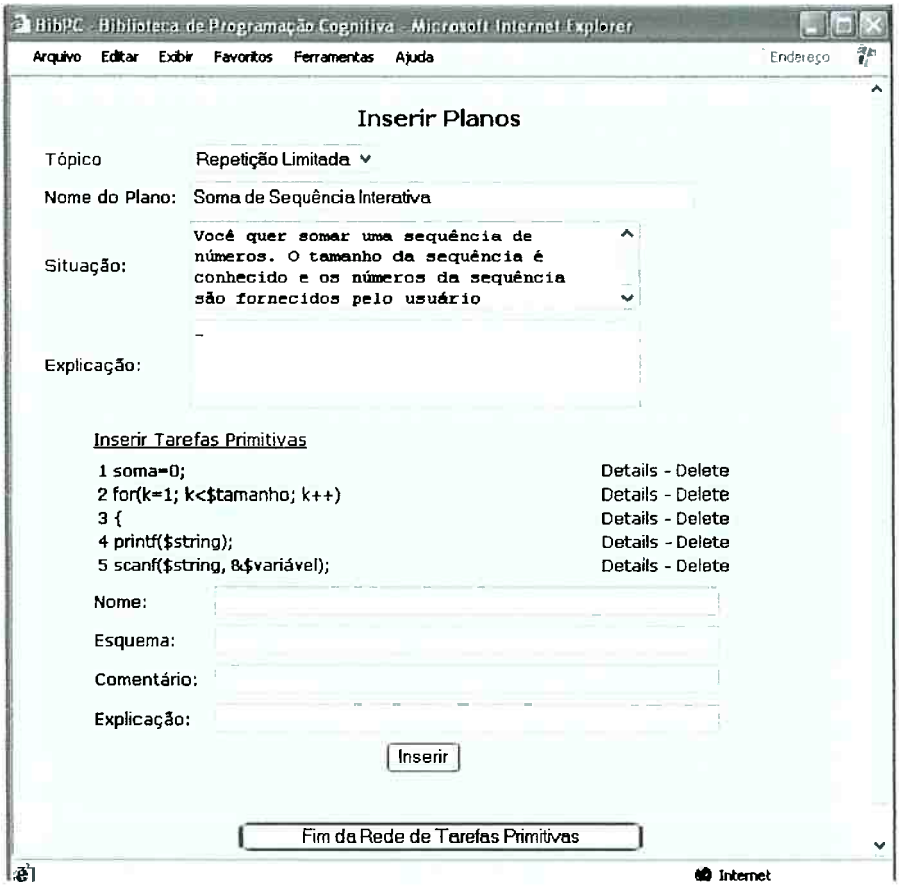


Figura 25- Inserção do Plano "Escolha Não Relacionada"

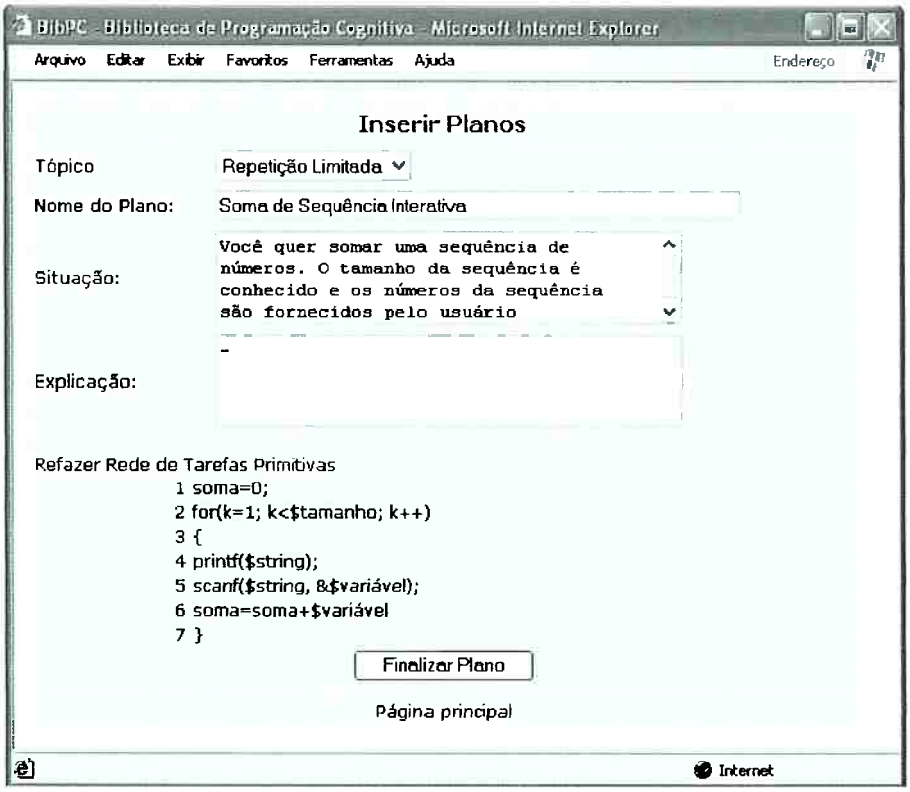


Figura 26 – Passo Final da Inserção de um Plano de Programação

7.1.2.2 Atividades do Professor na BibPC

A tarefa do professor, na Biblioteca para Programação Cognitiva, é a de modelar problemas e métodos, tal como descrito no Capítulo 6. A existência de soluções corretas e alternativas para problemas (métodos), na BibPC, permite sua utilização em sistemas de aprendizagem que realizam diagnóstico, tais como sistemas tutores inteligentes. A tabela 33 apresenta o algoritmo das ações principais do professor para a inserção de um problema na BibPC.

Tabela 33 – Algoritmo da Atividade de Inserção de um Problema

Tarefa	***inserir_problema***
Objetivo	Inserir um novo problema
Papéis de entrada	enunciado, Tópico:conjunto de tópicos do curriculum
Papéis de saída	RTM: Rede de Tarefas Meta
Sub-tarefas	***classificar, decompor***
Papéis internos	tarefa-problema
Estrutura de Controle	***inserir_problema*** (enunciado, Tópico → tarefa_problema, RTM) = ***classificar*** (enunciado, Tópico → tarefa-problema) ***decompor*** (enunciado → RTM)

Inicialmente o professor seleciona o tópico e o nível para classificar o problema. Em seguida insere o enunciado. Na sequência do processo, o professor cria uma Rede de Tarefas Meta para o problema. Uma janela "**Construir Tarefa Meta**" permite a inserção das características de cada tarefa meta. A figura 27 apresenta alguns desses passos na tela principal de inserção de problemas da BibPC. Nesse ponto, o professor estabelece um método para resolver a tarefa meta em construção, selecionando um plano de programação para a tarefa. A fim de decidir pelo plano ideal, o professor tem a possibilidade de visualizar os detalhes de cada plano (nome, situação, tarefas primitivas), conforme figura 28. A definição dos planos de programação para a Rede de Tarefas Meta marcam o início da solução do problema.

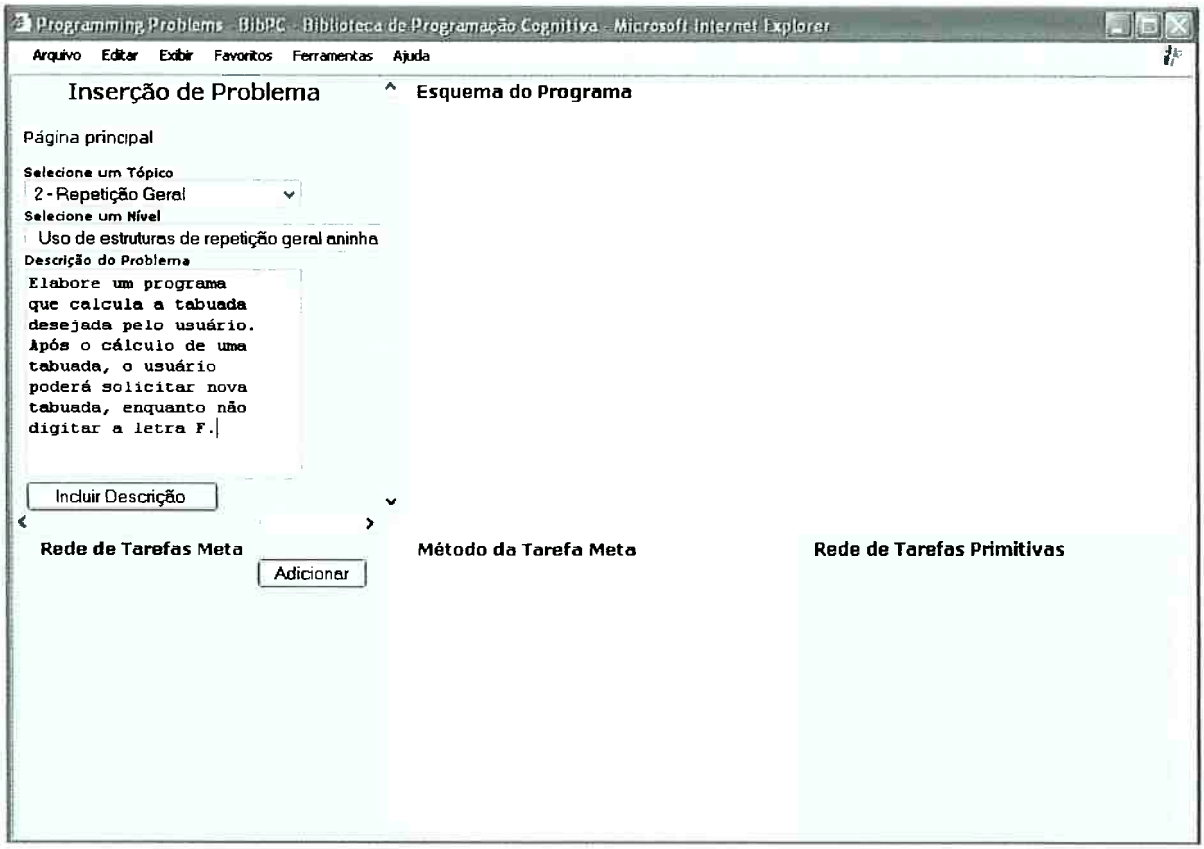


Figura 27 – Inserindo um problema na BibPC

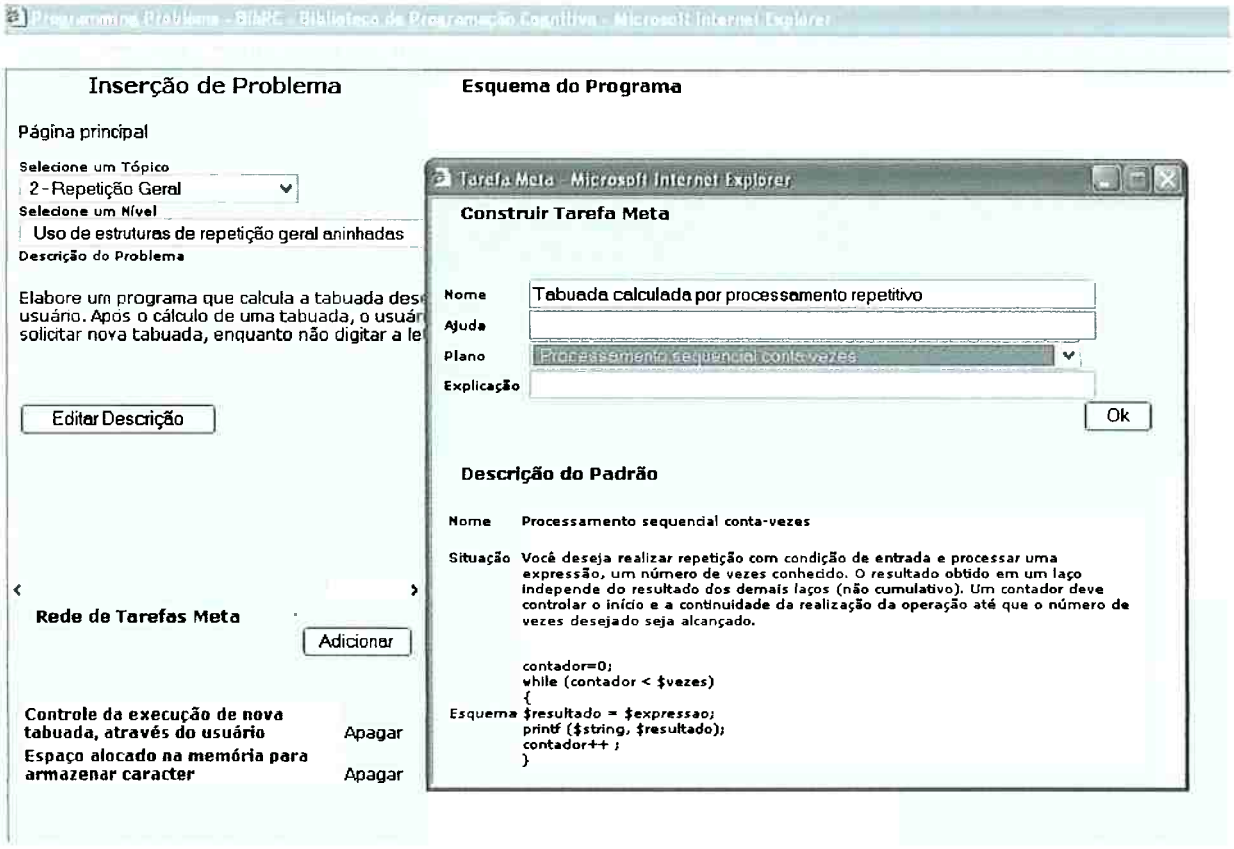


Figura 28 – Seleção de um plano para uma tarefa meta

Após a criação da Rede de Tarefas Meta e a definição de métodos para as tarefas meta da rede, o professor inicia a ordenação das tarefas primitivas dos planos selecionados. Pressionando com o mouse sobre o nome de uma tarefa meta na janela **Rede de Tarefas Meta**, o método da tarefa meta correspondente é apresentado, ao lado, na janela **Método da Tarefa Meta**.

Pressionando com o mouse sobre o nome do plano estabelecido no método, a Rede de Tarefas Primitivas do plano é apresentada, ao lado, na janela **Rede de Tarefas Primitivas**. Cada linha da Rede de Tarefas Primitivas corresponde a uma tarefa primitiva.

Pressionando com o mouse no botão "?", os detalhes de uma tarefa primitiva podem ser observados. Dessa forma, o professor visualiza todos os componentes que envolvem a solução do problema. A figura 29 apresenta alguns desses componentes.

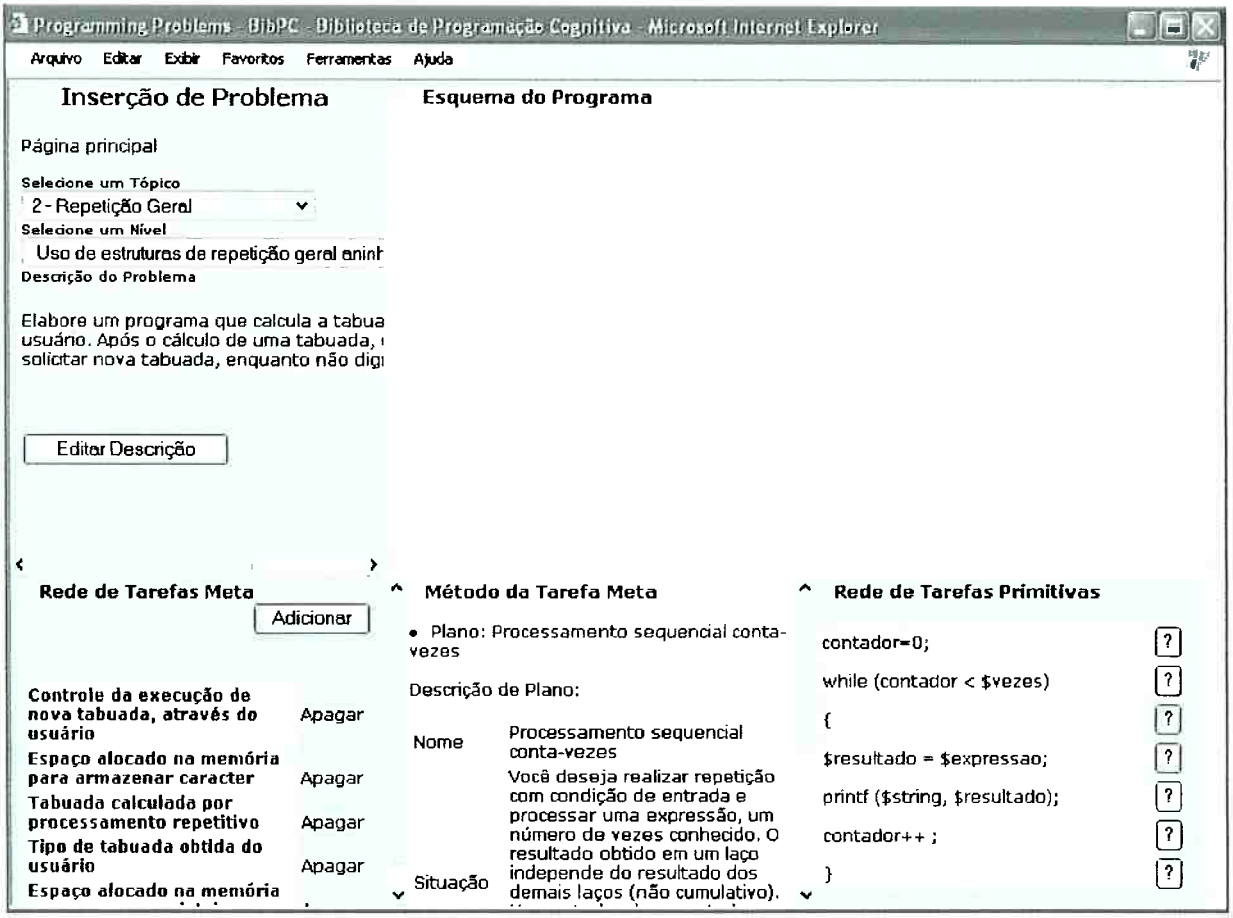


Figura 29 – Componentes cognitivos para elaboração do Programa

O próximo passo consiste em ordenar as tarefas primitivas dos planos para formar o chamado **Esquema do Programa**. Pressionando com o mouse sobre o nome de uma tarefa primitiva, o esquema primitivo é adicionado na janela superior direita denominada **Esquema do Programa**. Dessa forma, o esquema do programa vai sendo construído, seqüencialmente, pelo professor. A figura 30 fornece uma visão parcial desse processo.

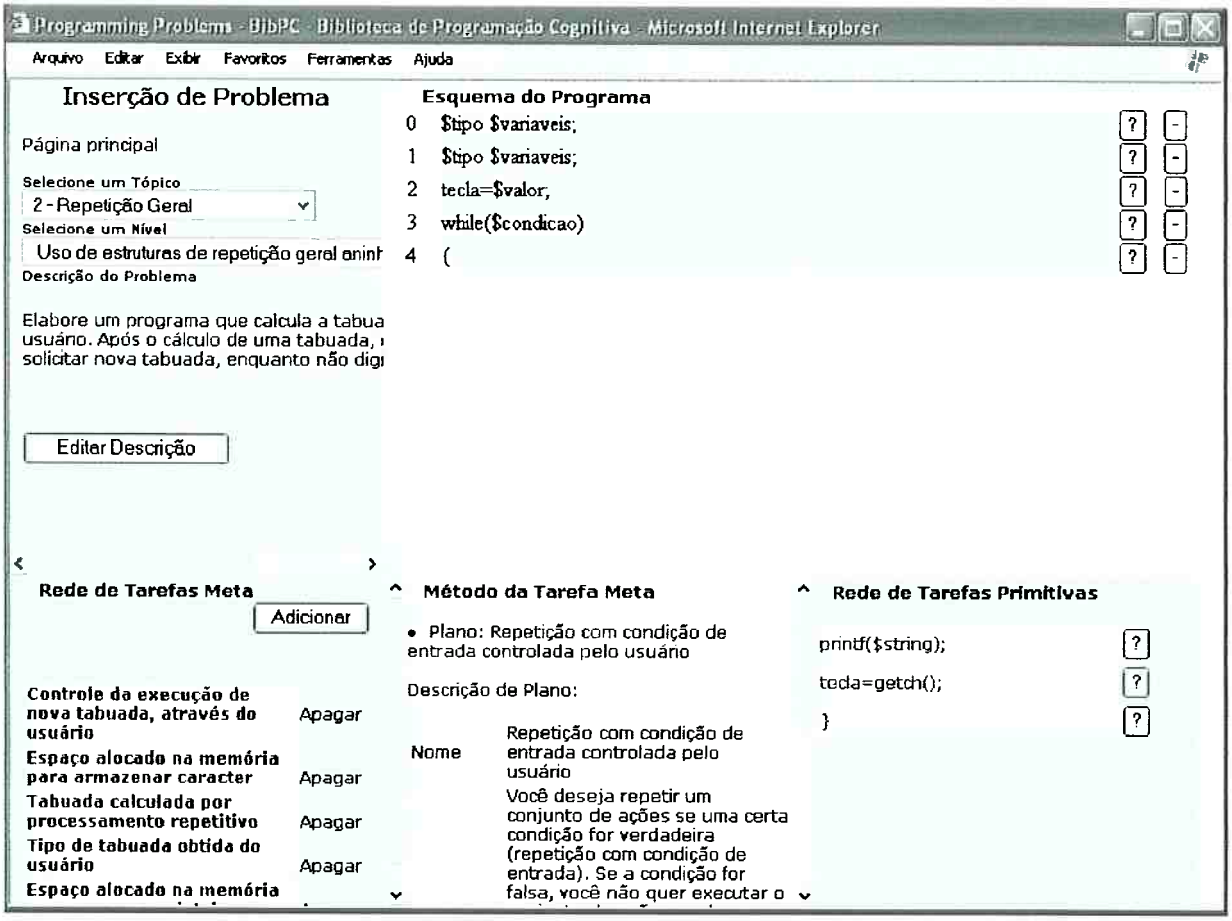


Figura 30 – Iniciando a ordenação de tarefas primitivas

O último passo para o professor completar a solução do problema consiste em instanciar o Esquema do Programa. Para isso, as meta-variáveis são substituídas por caixas de entradas de dados, nas quais o professor insere elementos tais como variáveis, valores, testes, operadores e frases para o usuário, finalizando a construção do programa. A figura 31 mostra a interface usada pelo professor e a figura 32 mostra o programa finalizado.

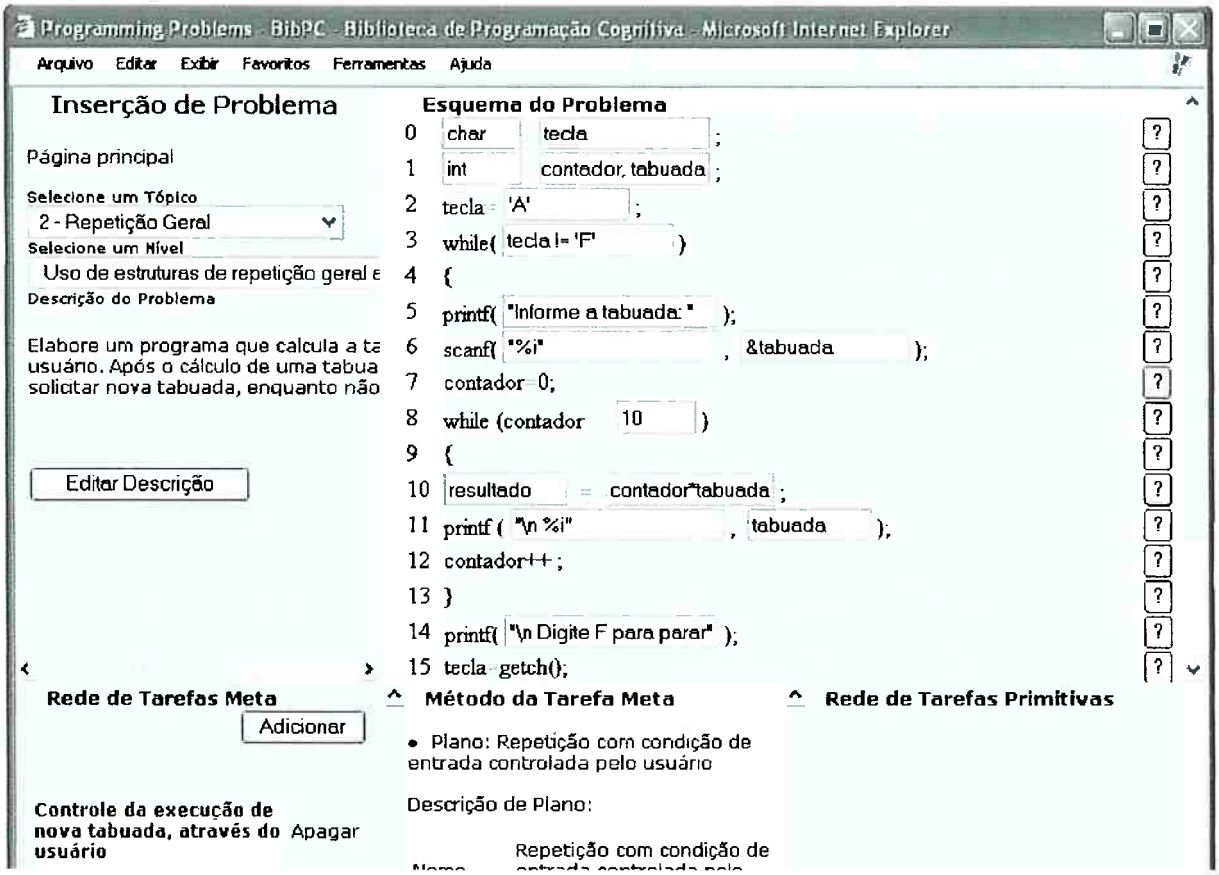


Figura 31 – *O Programa: Esquema do Programa Instanciado*

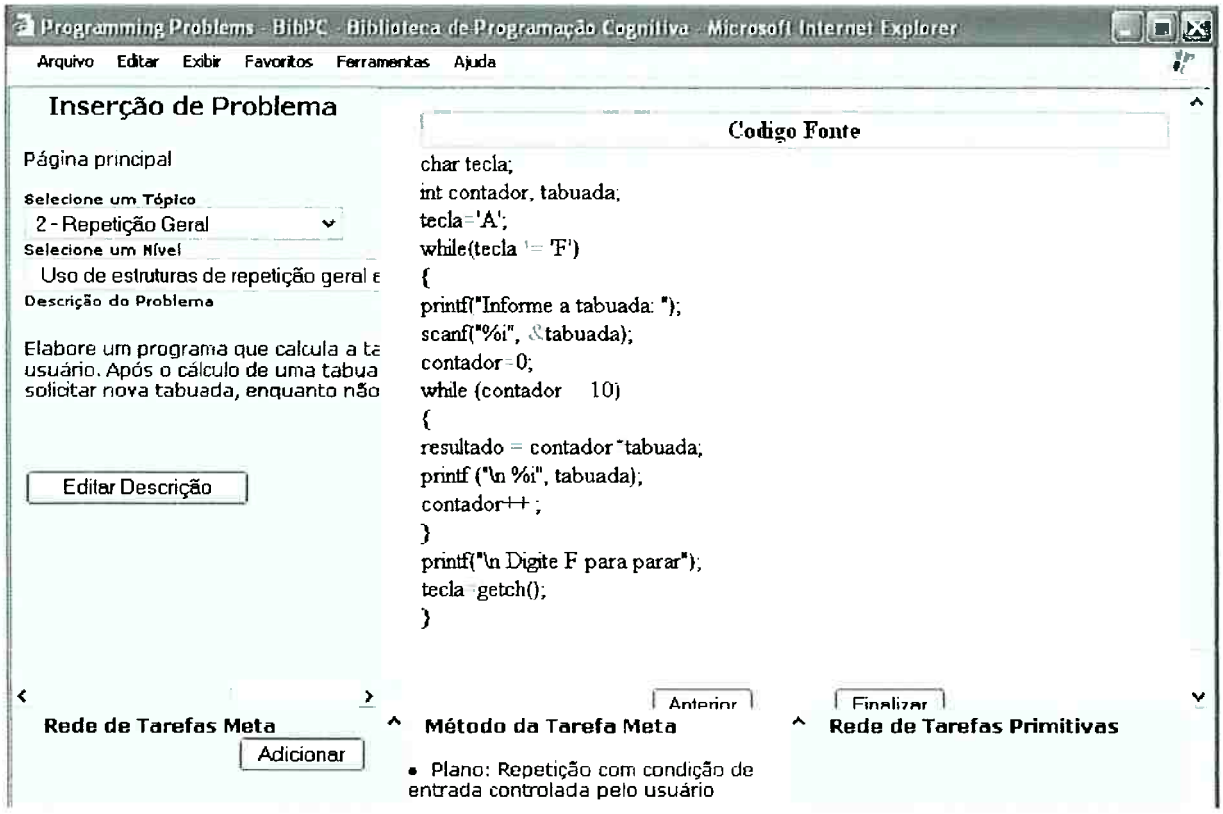


Figura 32 – Código Fonte da Solução de Problema

A tabela 34 apresenta o algoritmo das ações do professor que resume a construção de uma solução de problema na BibPC, conforme descrito.

Tabela 34 - Algoritmo da Atividade de Solução de Problema

Tarefa	***construir_solução_de_problema***
Objetivo	Modelar um método de solução para um problema
Papéis de entrada	RTM: Rede de Tarefas Meta; P: conjunto de planos do sistema
Papéis de saída	Esquema: sequência de esquemas primitivos ordenados; Programa: seqüência de esquemas primitivos instanciados.
Sub-tarefas	***construir_métodos_RTM, construir_esquema-do-programa, construir_programa, selecionar, associar, obter, ordenar, instanciar***
Papéis internos	tarefa-meta; plano; método; tarefa-primitiva; esquema-primitivo; código-fonte; MTM: conjunto de métodos para a Rede de tarefas Meta do problema; RTP: Rede de Tarefas Primitivas; CRTP: conjunto de Redes de Tarefas Primitivas.
Estrutura de Controle	***construir_solução_de_problema*** (RTM, P → Esquema, Programa)= ***construir_métodos_RTM*** (RTM, P → MTM) = para cada tarefa-meta ∈ RTM ***selecionar***(P → plano) ***associar***(tarefa-meta, plano → método) MTM ← MTM ∪ método ***construir_esquema_do_programa*** (MTM → $Esquema_{programa}$)= para cada método ∈ MTM ***obter*** (plano → RTP) CRTP ← CRTP ∪ RTP para cada tarefa-primitiva ∈ CRTP ***ordenar*** (tarefa-primitiva → $Esquema_{programa}$) ***construir_programa*** ($Esquema_{programa}$ → Programa) = para cada esquema-primitivo ∈ $Esquema_{programa}$ ***instanciar*** (esquema-primitivo → código-fonte) Programa ← Programa ∪ código-fonte

BibPC permite ainda que o professor possa inserir outros métodos de solução para um mesmo problema. Nesse caso, o professor tem várias facilidades disponíveis para realizar a construção de uma nova solução para um problema já existente na BibPC. Antes de criar uma nova solução, o professor pode visualizar as redes de tarefas meta já existentes para o problema. Em cada uma das redes, ele pode visualizar os métodos estabelecidos para cada uma das metas de uma determinada rede, pressionando sobre o botão "?". Além disso, ele pode visualizar o código fonte gerado a partir de uma rede. Na BibPC, a seleção de um plano alternativo para uma meta de uma Rede de Tarefas Meta, estabelece uma nova solução para o problema. A figura 33 mostra a janela principal para visualização de soluções existentes para os problemas da BibPC.

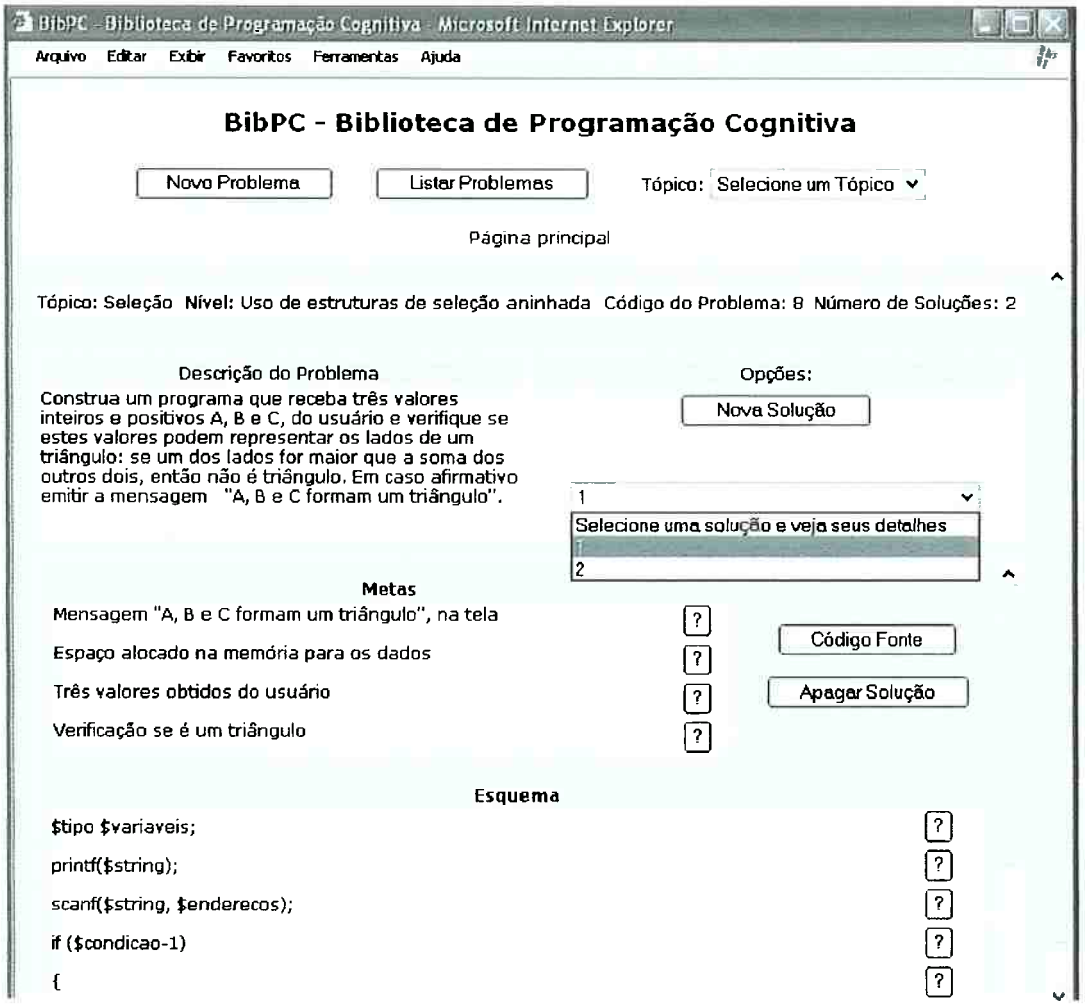


Figura 33 – Janela para Visualização de Métodos Alternativos dos Problemas

7.2 TutorC: Um Ambiente para Planejamento em Programação

O conhecimento e habilidades necessárias para realizar programação algorítmica, discutidos até aqui, torna claro que saber programar vai além do conhecimento da sintaxe de uma linguagem de programação. Além dos conceitos básicos do mundo da programação (tipos de dados, variáveis, operadores, estruturas de fluxo, etc), a especialidade em programar requer:

1. Identificação de metas a serem alcançadas para resolver um problema;
2. Conhecimento sobre planos ou *chunks* de programação (SOLOWAY;EHRLICH,1984);
3. Construção de métodos para realizar as metas, ou seja, conhecimento para associar metas à planos conhecidos ou adaptá-los para metas não-familiares (SOLOWAY;EHRLICH,1984);
4. Conhecimento sobre ordenação de planos, a fim de atingir as metas na ordem correta

(EROL,1995);

5. Conhecimento sobre planos deslocalizados, o que requer conhecimento sobre a rede de ações primitivas que envolve um plano (SOLOWAY; LETOVSKY, 1986; LAMPERT, 1988).
6. Conhecimento da sintaxe de uma linguagem de programação;
7. Instanciação de elementos da linguagem de programação com dados do domínio do problema (SOLOWAY;ADELSON,1985;RUGABER,2000).
8. Entendimento sobre as características dinâmicas de um programa (GEORGE,2000);
9. Prática em depurar programas.

Assim, um tutor inteligente proposto para facilitar o aprendizado dessas tarefas, deve não somente possuir esse conhecimento como torná-lo explícito para o estudante. O TutorC (LEMOS; BARROS, 2003) apresentado neste capítulo foi implementado com tal objetivo tendo, nesta versão, conhecimento de programação referente aos itens de 1 a 6. Em relação ao item 1, TutorC mostra, além do enunciado do problema, o conjunto de metas a serem realizadas para atingir a solução do problema. Em relação ao item 2, TutorC oferece uma variedade de planos de programação que podem ser selecionados pelo estudante através dos critérios de adequabilidade. Planos, no TutorC, são representados por fragmentos de código generalizados com meta-variáveis. Em relação ao item 3, TutorC realiza diagnóstico da seleção de planos do estudante, comparando com métodos modelados pelo professor na BibPC. Em relação ao item 4, TutorC oferece planos em vários níveis de complexidade, onde planos mais complexos agregam planos mais simples. O modo como o professor decompõe as metas de um problema pode conduzir à necessidade do aluno trabalhar com planos mais simples ou mais complexos. Em conseqüência, o estudante tem que raciocinar sobre hierarquia de planos e metas. Em relação ao item 5, ações de programação são representadas pelas tarefas primitivas no tutor, descritas na forma de texto, o que permite ao estudante aprender o significado de cada parte do plano, facilitando aplicações deslocalizadas. Em relação aos itens 6 e 7, TutorC mostra, em ordem alfabética, o conjunto de esquemas primitivos corretos, instanciados com dados do domínio do problema, ou seja as declarações em C corretas. A tarefa do estudante consiste em realizar o mapeamento entre as ações primitivas (tarefas primitivas na forma de texto) e as declarações em C.

A arquitetura do TutorC está baseada na arquitetura proposta pela área de Sistemas Tutores Inteligentes da IA (WENGER,1987; SHUTE;PSOTKA,1994; SELF,1995), sendo composto por cinco módulos que se relacionam como mostra a figura 34. TutorC foi implementado em linguagem C++ com acesso a banco de dados SQL.

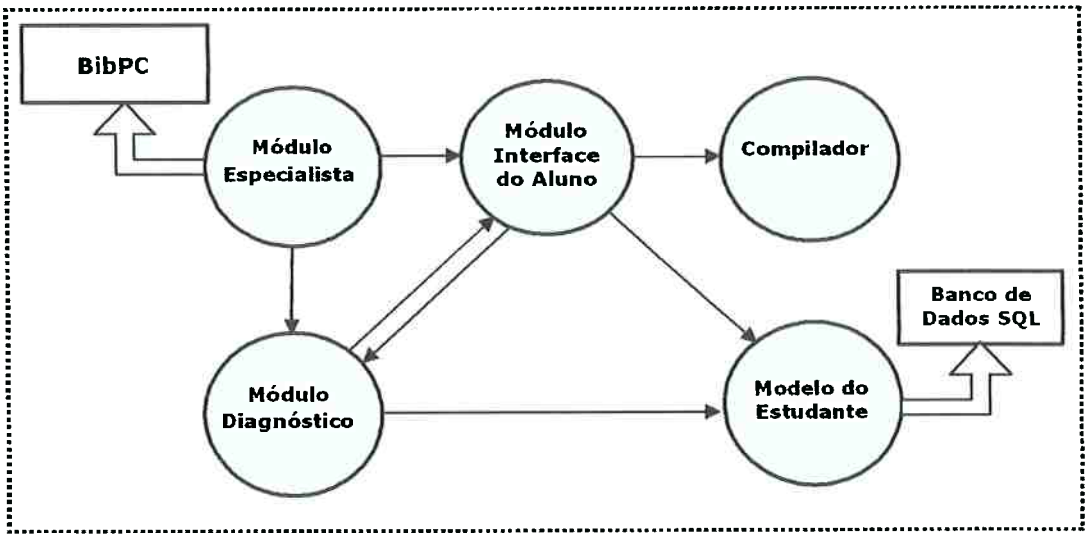


Figura 34 - Arquitetura de TutorC

O conhecimento do **Módulo Especialista** do TutorC é obtido a partir do sistema BibPC. Assim, TutorC disponibiliza em sua interface com o estudante, o **Módulo Interface do Aluno**, um conjunto de componentes de planejamento para programação que permitem a construção de programas de uma forma independente dos detalhes de sintaxe da linguagem C. TutorC realiza diagnóstico (**Módulo Diagnóstico**) em níveis de abstração diferentes, sendo capaz de orientar o aluno em: (1) seleção de planos de programação para metas de problema, (2) ordenação de planos e tarefas primitivas e (3) mapeamento de tarefas primitivas sobre declarações em C. A qualquer momento, tipicamente após ter seu programa finalizado e diagnosticado, o estudante pode disparar, a partir de TutorC, o **Compilador** Turbo C++ Versão 3.0 da Borland (2003). Nesse novo ambiente, o aluno pode compilar e executar seu programa, assim como fazer quaisquer alterações que vão além dos planos de programação propostos no TutorC. O **Módulo do Estudante** do tutor, identifica unicamente cada usuário mantendo um histórico de informações de cada aluno. Essas informações podem ser vistas na WEB pelo professor, na forma de gráficos, para fins de acompanhamento do desempenho e evolução do aprendizado do aluno. As próximas seções detalham cada um dos módulos do TutorC.

7.2.1 Módulo Especialista

O Módulo Especialista é o componente do sistema que contém o conhecimento do domínio, isto é, o material instrucional modelado segundo técnicas da Inteligência Artificial baseando-se, nesta pesquisa, numa linguagem de Planejamento Hierárquico, Padrões Elementares da comunidade de Padrões Pedagógicos para Computação e na Ciência Cognitiva, especialmente a área da Psicologia da Programação.

A essência do conhecimento especialista do TutorC origina-se do banco de dados da BibPC, descrita na seção 7.1. O Módulo Especialista disponibiliza, para a Interface do Aluno, descrições de problemas na forma de redes de tarefas meta e planos de programação na forma de redes de tarefas primitivas. Para o Módulo Diagnóstico, o Módulo Especialista disponibiliza os métodos modelados pelo professor na BibPC.

Uma vez que BibPC é a fonte de conhecimento de TutorC, o tutor torna-se uma ferramenta de aplicações amplas que pode tanto apoiar cursos introdutórios como cursos avançados de linguagem C. Cada professor em sua instituição pode caracterizar TutorC com o nível de conhecimento desejado, modelando componentes introdutórios ou avançados, por meio de BibPC. Segundo Brusilovsky (1995), essa é uma caracteristica desejável em sistemas tutores e um dos motivos pelo qual a aplicação desses sistemas no mundo real, ou seja em salas de aula, é ainda raro.

7.2.2 Módulo Interface do Aluno

O módulo do TutorC que permite a interação entre estudante e tutor é o chamado Módulo Interface do Aluno. Por meio dessa interface, o aluno pode trabalhar no desenvolvimento de programas estimulado pela presença de componentes tais como planos e ações de programação, metas de problemas e declarações em linguagem C. Assim, o estudante manipula componentes de diferentes níveis de abstração gerando soluções abstratas até alcançar o nível da implementação, no qual o programa codificado é obtido.

7.2.2.1 Arquitetura da Interface

A interface é constituída de um conjunto de janelas que são apresentadas seqüencialmente, ao estudante, durante a tarefa de construção de um programa. As telas principais são:

- Janela Login,
- Janela Seleção de Problema,
- Janela Seleção de Modelos,
- Janela Seleção de Planos,
- Janela Construção de Programa,
- Janela Diagnóstico,
- Janela Compilador.

As próximas seções descrevem as operações que o estudante pode realizar através da Interface do Aluno, detalhando a funcionalidade das Janelas.

7.2.2.2 Efetuando o acesso ao TutorC

Para ter acesso à utilização do TutorC, o aluno deve estar cadastrado com um nome de usuário e uma senha. Isso é necessário para que dados específicos de cada aluno possam ser registrados no banco de dados do tutor, mantendo assim o Modelo do Estudante. Para o nome do usuário foi adotado o próprio número de registro do aluno (RA) na FACENS. A figura 35 apresenta a janela inicial de entrada no tutor.

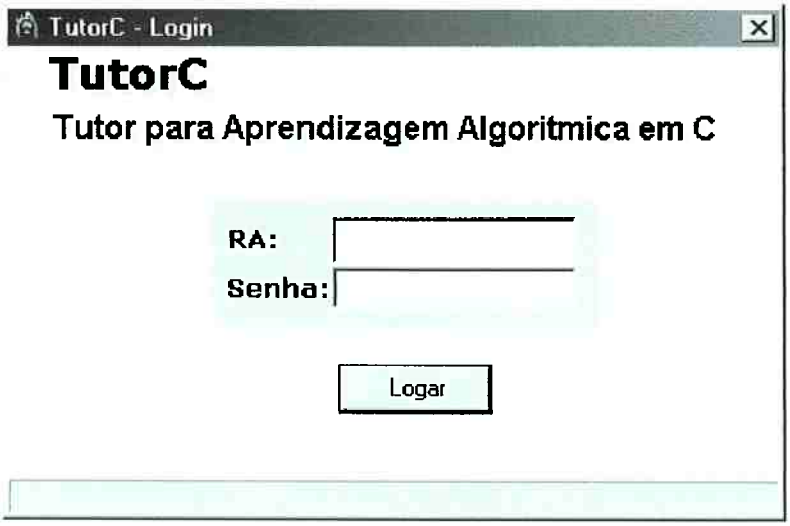


Figura 35 – Janela de Login do TutorC

7.2.2.3 Selecionando um Problema no TutorC

Após a identificação do aluno, o sistema apresenta opções para seleção de tópico, nível e problema para o qual o aluno deseja compor uma solução (figura 36). Essa base de problemas é a existente na BibPC, introduzida pelo professor. No tutorC o aluno decide, livremente, o tema que gostaria de praticar, assim como o problema específico que deseja resolver. A fim de exemplificar as ações de planejamento de programação no TutorC, o seguinte problema é utilizado:

Faça um programa que recebe uma temperatura em graus fahrenheit (ºF) e converte para celsius (ºC). A fórmula para conversão é C=5.0/9*(F-32). Uma vez calculada a temperatura, informe para o usuário se está frio (abaixo de 15 graus), normal (entre 15 e 30 graus) ou quente (acima de 30 graus). O programa deve repetir o cálculo até que o usuário informe que deseja parar. Considere a letra S, como desejo de continuar executando o programa, qualquer outra letra deve interromper a execução do programa.

Problema 6

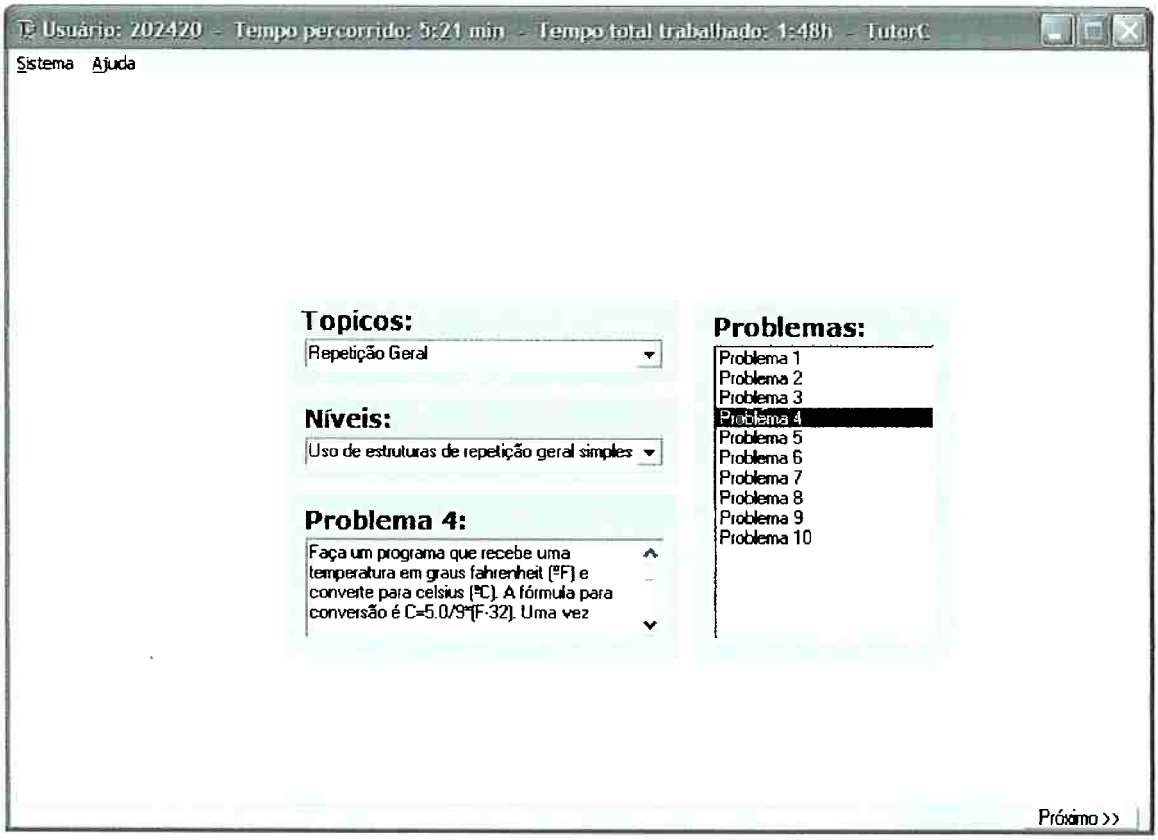


Figura 36 – Janela Seleção de Problema do TutorC

7.2.2.4 Selecionando uma Rede de Tarefas Meta

Uma vez que um problema é selecionado, o tutor apresenta, na janela **Modelos do Problema**, *todas as redes de tarefas metas existentes na BibPC para o problema*. Conforme detalhado no Capítulo 6, uma rede de tarefas meta contém um conjunto de metas que devem ser realizadas para resolver o problema. Cada rede corresponde a uma visão de decomposição do problema e estabelece um método de resolução para o problema. Para o aluno, a Rede de Tarefas Meta é denominada **Modelo do Problema**. A figura 37 apresenta a janela para seleção de modelos do problema selecionado.

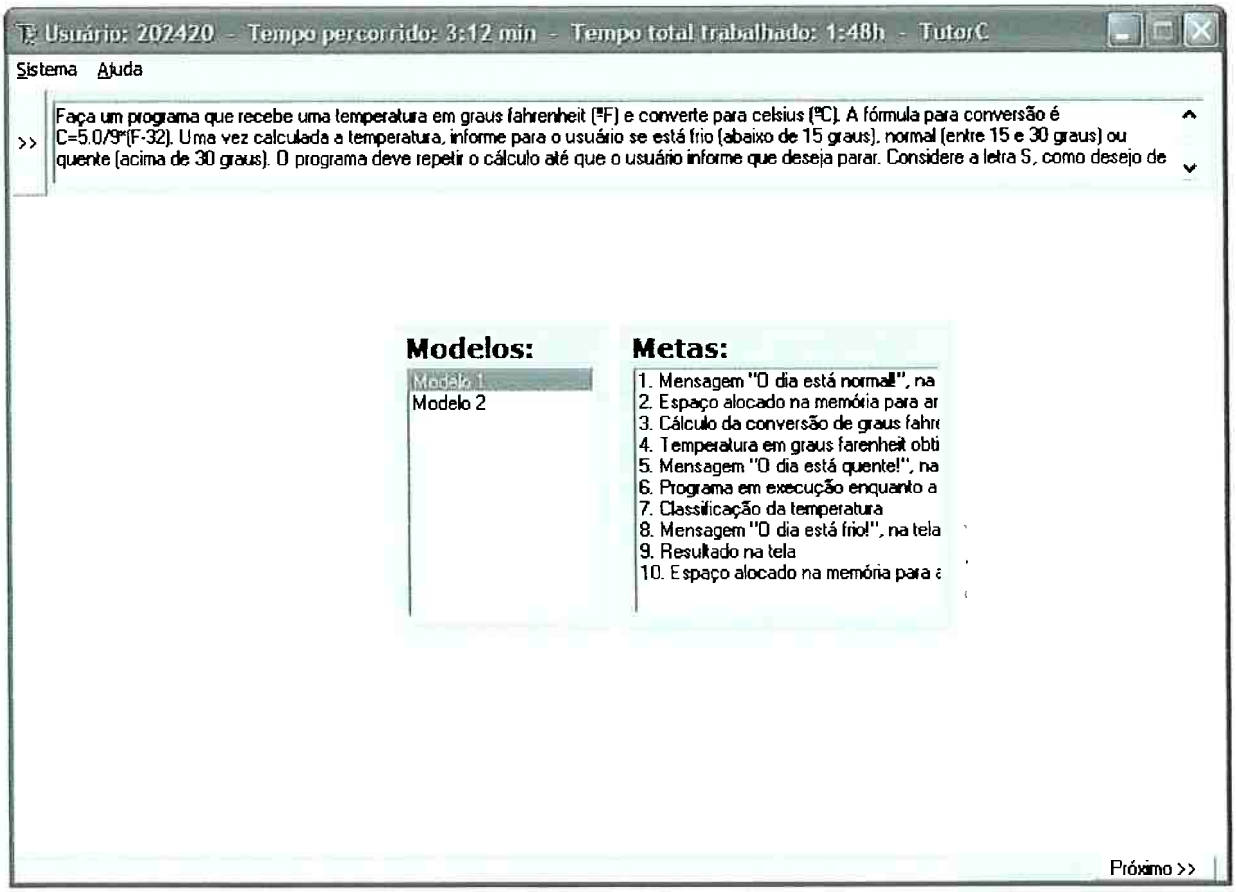


Figura 37 – Janela Modelos do Problema

7.2.2.5 Construindo Métodos para Realizar a Rede de Tarefas Meta

Após a seleção do modelo do problema, TutorC apresenta a janela **Seleção de Planos**. Nessa janela o aluno pode visualizar todas as metas do problema e todos os planos de programação disponíveis para o tópico do problema. Guiando-se pelos critérios de aplicabilidade dos planos e da visualização de esquemas de código, o aluno seleciona um plano para cada uma das metas, estabelecendo assim um conjunto de métodos. A figura 38 apresenta a construção do conjunto de métodos para um modelo do problema exemplo. Na parte superior dessa janela, a tarefa meta 6, "Programa em execução enquanto a letra S não for digitada", foi associada ao plano "Repetição com condição de entrada controlada pelo usuário". Na parte inferior, o aluno pode navegar nos planos disponíveis, aprendendo sobre adequabilidade e esquemas (*chunks* de programação) de planos. Todas as metas devem estar associadas a um plano a fim de habilitar o botão **Próximo**, no canto inferior direito.

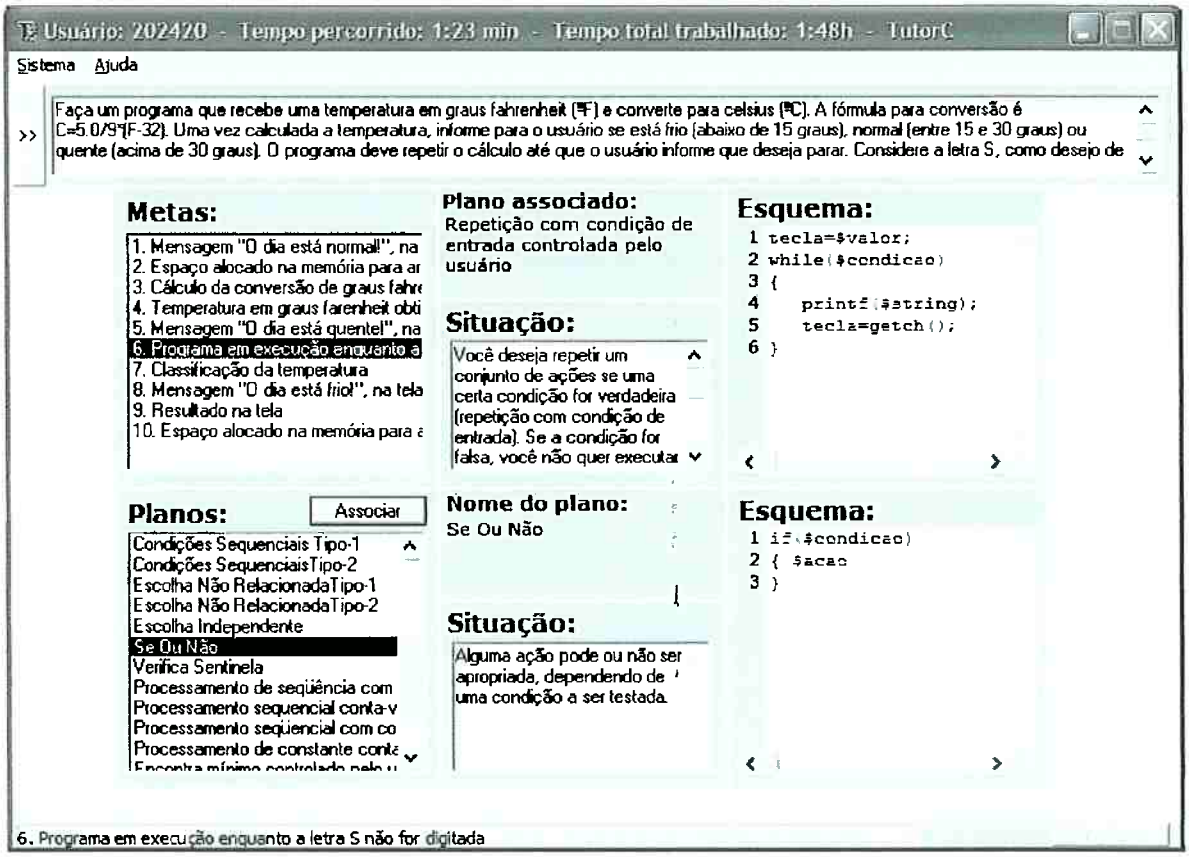


Figura 38 – Janela Planos do TutorC

7.2.2.6 Construindo o Programa

Na janela **Construção de Programa**, é possível visualizar o enunciado do problema conforme mostra a parte superior da figura 39. Imediatamente abaixo, uma árvore representa os métodos para o modelo do problema: as metas do problema (raiz da árvore) e os planos de programação associados (folhas da árvore). Pressionando com o mouse sobre o plano associado à uma meta, é possível visualizar, na janela **Tarefas Primitivas**, a decomposição do plano em tarefas primitivas. O aluno pode selecionar o modo de visualização da decomposição: **texto** ou **esquema**. No modo **texto**, a decomposição pode ser vista por meio da descrição das ações na forma de texto, como mostra a figura 39. No modo **esquema** a decomposição pode ser vista na forma de código e meta-variáveis, como mostra a figura 40. Além disso, a janela **Construção de Programa**, apresenta ao aluno o conjunto de declarações C corretas para o modelo do problema selecionado (janela Fonte). O aluno pode, então, ordenar as tarefas primitivas dos planos e mapeá-las nas declarações em C a fim compor o programa para o problema. A janela **Fonte Usado** permite exclusão (botão **Apagar**) de linhas de código inserido no programa sob construção, assim como alteração da posição das linhas de código do programa (botões **Acima** e **Abaixo**).

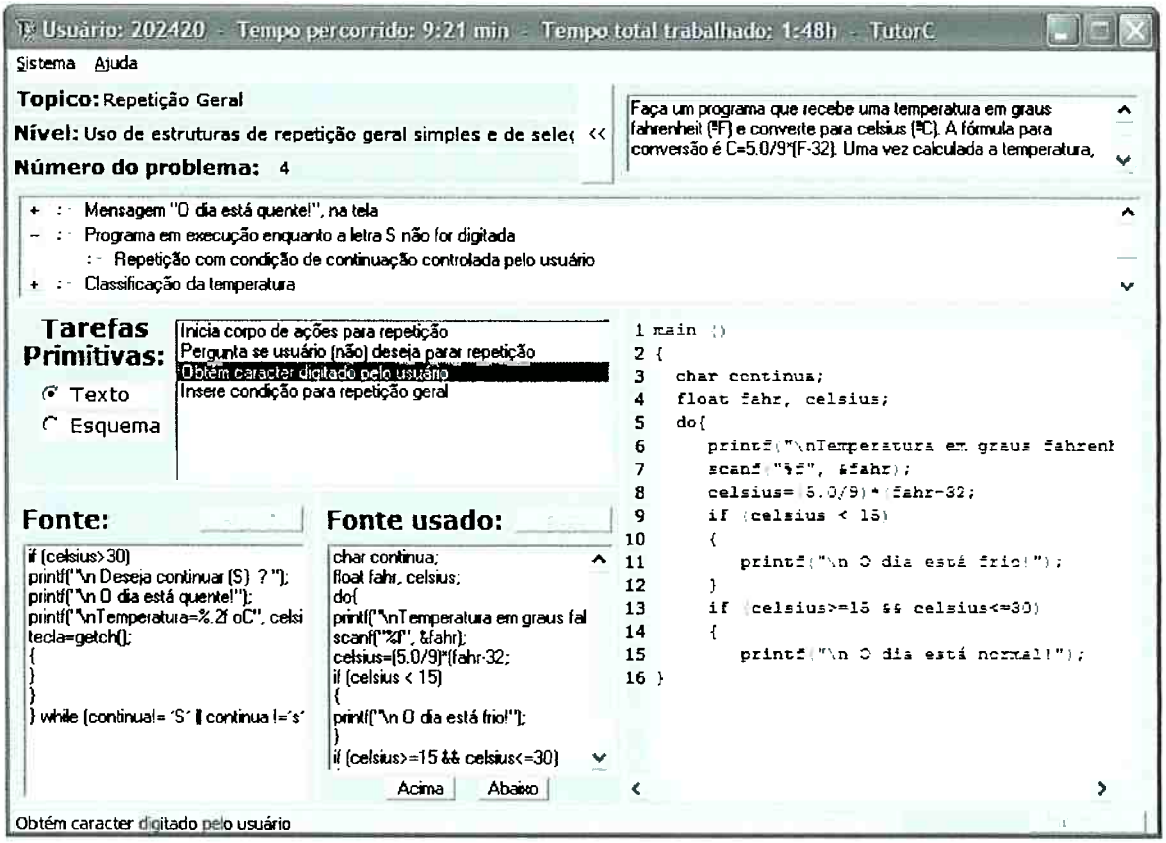


Figura 39 – Ordenando e Mapeando Ações Primitivas

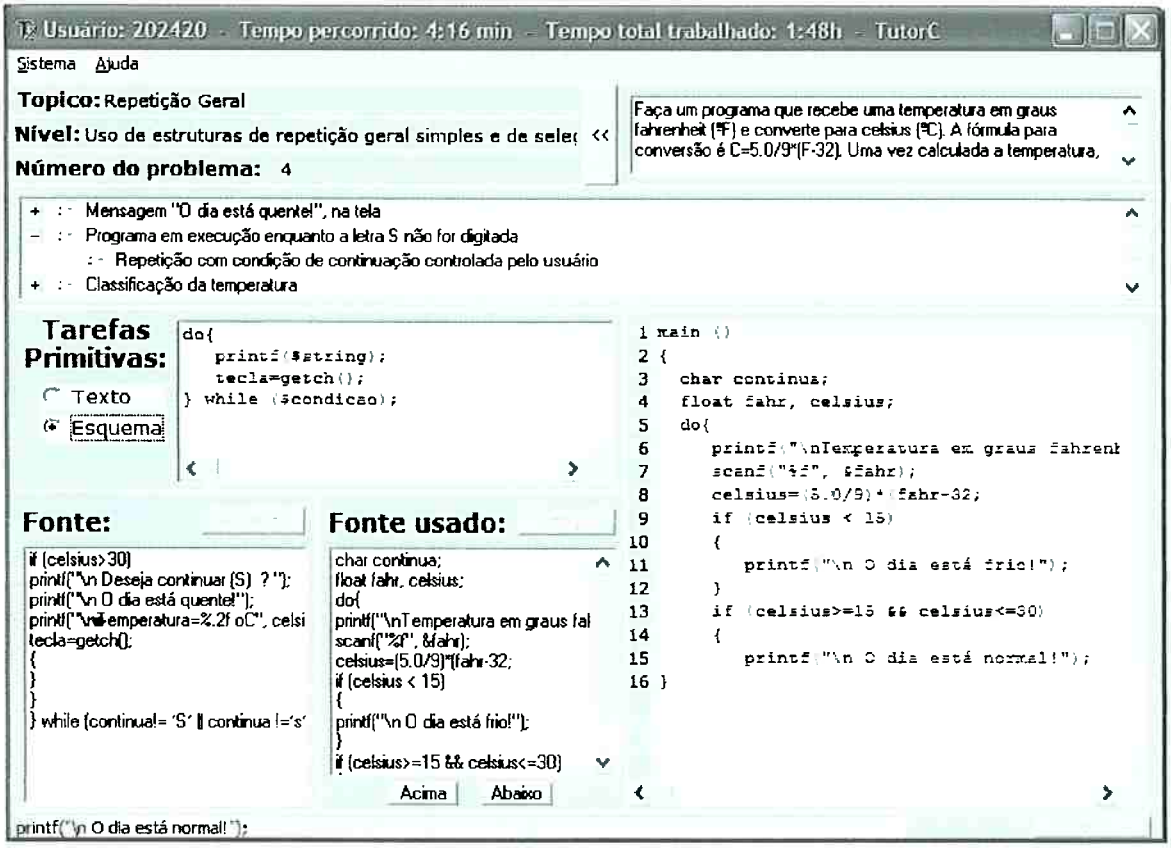


Figura 40 – Janela Construção de Programa no modo Esquema

O botão **Próximo** torna-se habiblitado somente após todas as linhas de código corretas (janela Fonte) terem sido utilizadas na construção do programa. A figura 41 mostra o programa completo, ou seja, todas as tarefas primitivas dos planos foram ordenadas e mapeadas. Nesse estágio, é possível observar a janela **Fonte** vazia e o botão **Próximo** habilitado.

No TutorC, o estudante vai adquirindo familiarização e conhecimento sobre a sintaxe e a semântica da linguagem por associação. Para cada plano visualizado na interface, seu esquema (bloco de código generalizado) é também disponibilizado. Para cada tarefa primitiva selecionada pelo estudante, na fase de ordenação, seu esquema e descrição em texto podem ser visualizados. Finalmente, todas as declarações corretas da solução ficam disponíveis, em ordem alfabética, para que o aluno faça o mapeamento de cada esquema de tarefa primitiva com o código C correspondente. Assim, o aluno pode aprender sobre como instanciar dados do domínio do problema nas estruturas da linguagem de programação. As meta-variáveis dos esquemas contribuem para esse aprendizado, uma vez que seus nomes refletem o papel que o dado do domínio desempenha na solução de problema. O aluno não precisa se preocupar com a sintaxe da linguagem, a qual é colocada na janela, automaticamente. Erros de sintaxe, no início da aprendizagem, são apontados como causadores de confusão em aprendizes.

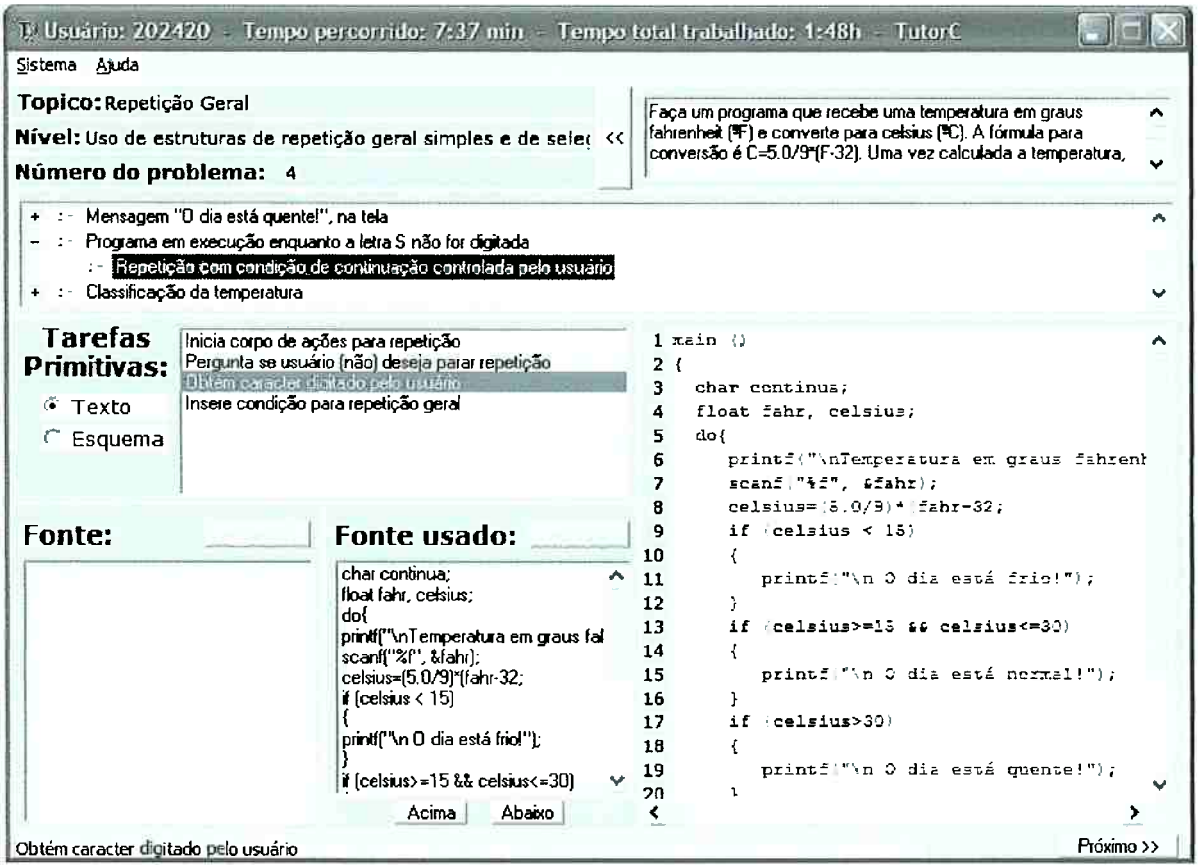


Figura 41 – Programa Finalizado

Além da estratégia de construção de programa implementada nesta versão do TutorC, outras estratégias podem ser implementadas, por exemplo:

1. O aluno visualiza somente as instâncias das meta-variáveis que o professor forneceu quando modelou a solução de problema na BibPC, em vez da declaração em C completa.
2. O aluno realiza a ordenação das tarefas primitivas obtendo o **esquema do programa** e instancia-o sem apoio algum, da mesma maneira que o professor constrói a parte final de uma solução na BibPC.
3. O aluno constrói o código C visualizando apenas a descrição textual das ações primitivas. Nesse caso espera-se que o aluno tenha conhecimentos da sintaxe da linguagem de programação.

7.2.3 Módulo Diagnóstico

O diagnóstico da solução do estudante, é realizada em três níveis de abstração:

- **Diagnóstico de Planos** que verifica a seleção correta de métodos para a Rede de Tarefas Meta (Modelo do Problema);
- **Diagnóstico Procedimental** que verifica a ordenação entre tarefas primitivas;
- **Diagnóstico de Mapeamento** que verifica o conhecimento sobre instanciação de dados do domínio nas estruturas da linguagem.

Durante o diagnóstico, possibilidades para reparo são oferecidas ao estudante. Somente, após algumas tentativas a solução correta é apresentada. Os três processos de diagnóstico levam em conta o conhecimento correto existente na BibPC. Os algoritmos de diagnóstico são relativamente simples, quando comparados com alguns tutores da literatura, uma vez que os componentes usados pelo professor para modelar soluções na BibPC são os mesmos manipulados por um aprendiz na interface do tutor. TutorC possui fidelidade epistêmica (WENGER,1987).

7.2.3.1 Diagnóstico de Planos

Após a seleção de planos para as metas de um problema, TutorC realiza automaticamente, o diagnóstico das escolhas do estudante. O sistema compara os **métodos** criados pelo estudante com todos os métodos armazenados na BibPC para a Rede de Tarefas Meta escolhida. Se algum método do estudante for considerado incorreto, o sistema avisa o estudante de que existe erro, porém não aponta qual método está errado, oferecendo uma chance para reparo (figura 42).

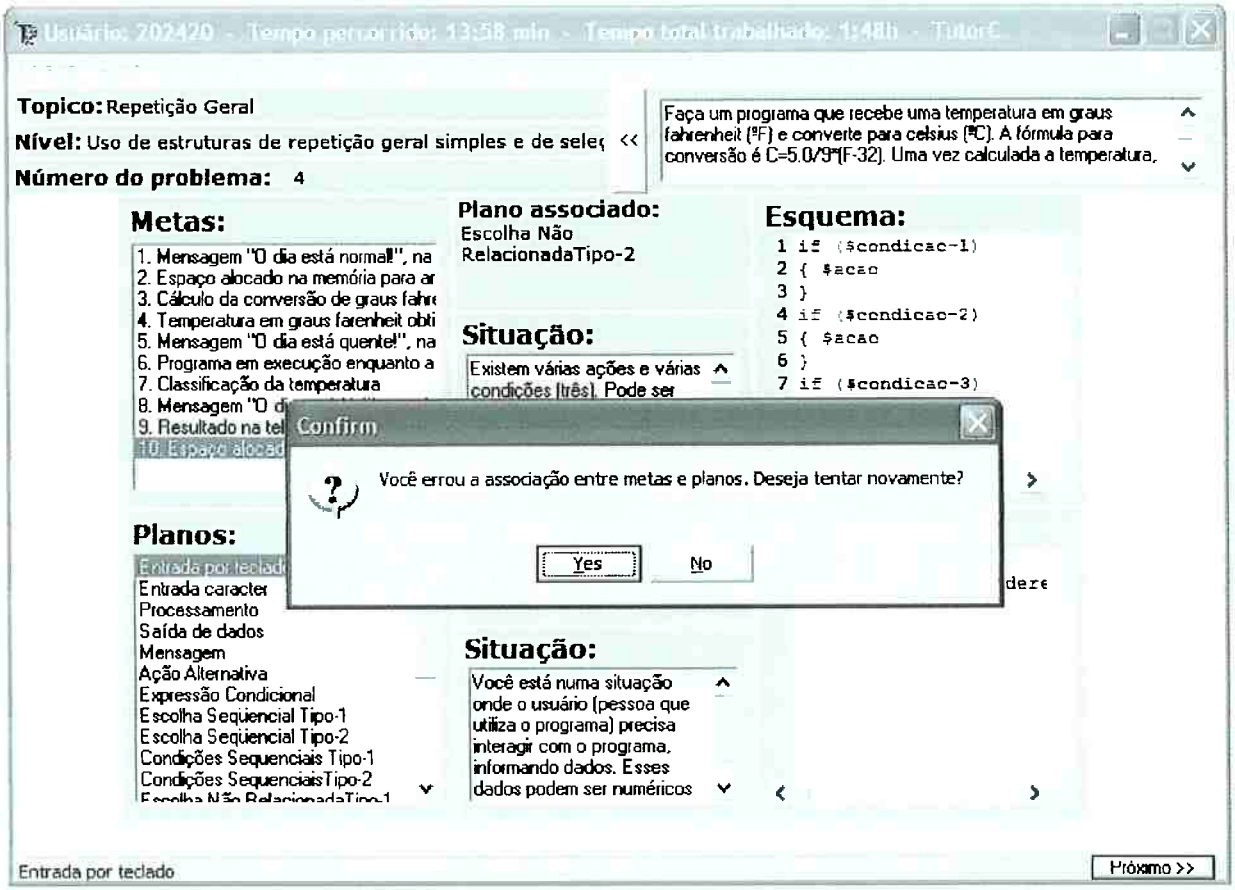


Figura 42 – Diagnóstico de Planos: Primeira Tentativa

Se houve erro numa segunda tentativa, o sistema aponta cada meta que não pode ser realizada com o plano selecionado pelo estudante. O professor quando estabele métodos para metas, pode armazenar uma explicação ou dica sobre como a meta pode ser realizada. Essa dica pode ser mostrada ao aluno nesse momento para mais uma tentativa de acerto. Se houve erro na terceira tentativa, o sistema apresentará um relatório e um conjunto de métodos válido para que o estudante prossiga na construção do programa. A qualquer momento, o estudante pode desistir do problema. A figura 43 mostra um relatório recebido na terceira tentativa com erro.

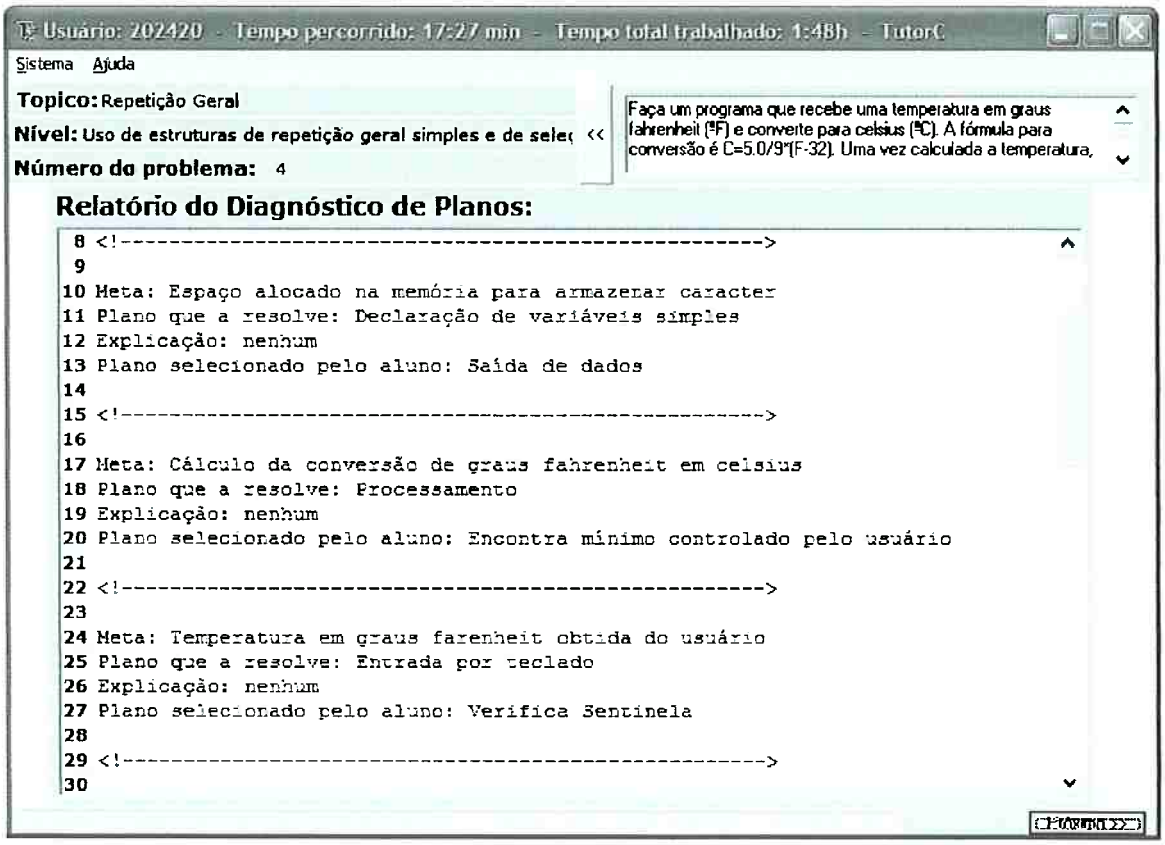


Figura 43 – Relatório de Diagnóstico de Planos

Após a finalização do Diagnóstico de Planos, o estudante prossegue na construção do programa conforme descrito na seção 7.2.2.6.

7.2.3.2 Diagnóstico Procedimental

Após o término da construção de um programa, a **Janela Diagnóstico** é apresentada ao estudante, como mostra a figura 44. Nela podem ser vistos: o código fonte do programa construído pelo aluno, o botão **Diagnosticar** e o botão **Compilar**.

Se o aluno decidir solicitar o diagnóstico, o primeiro a ser realizado é o **Diagnóstico Procedimental**. Esse processo compara a ordenação de tarefas primitivas da solução do estudante com a ordenação correta existente no sistema. Se existe alguma tarefa primitiva ordenada incorretamente, o sistema avisa o estudante de que existe erro, porém não o aponta, como mostra a figura 45.

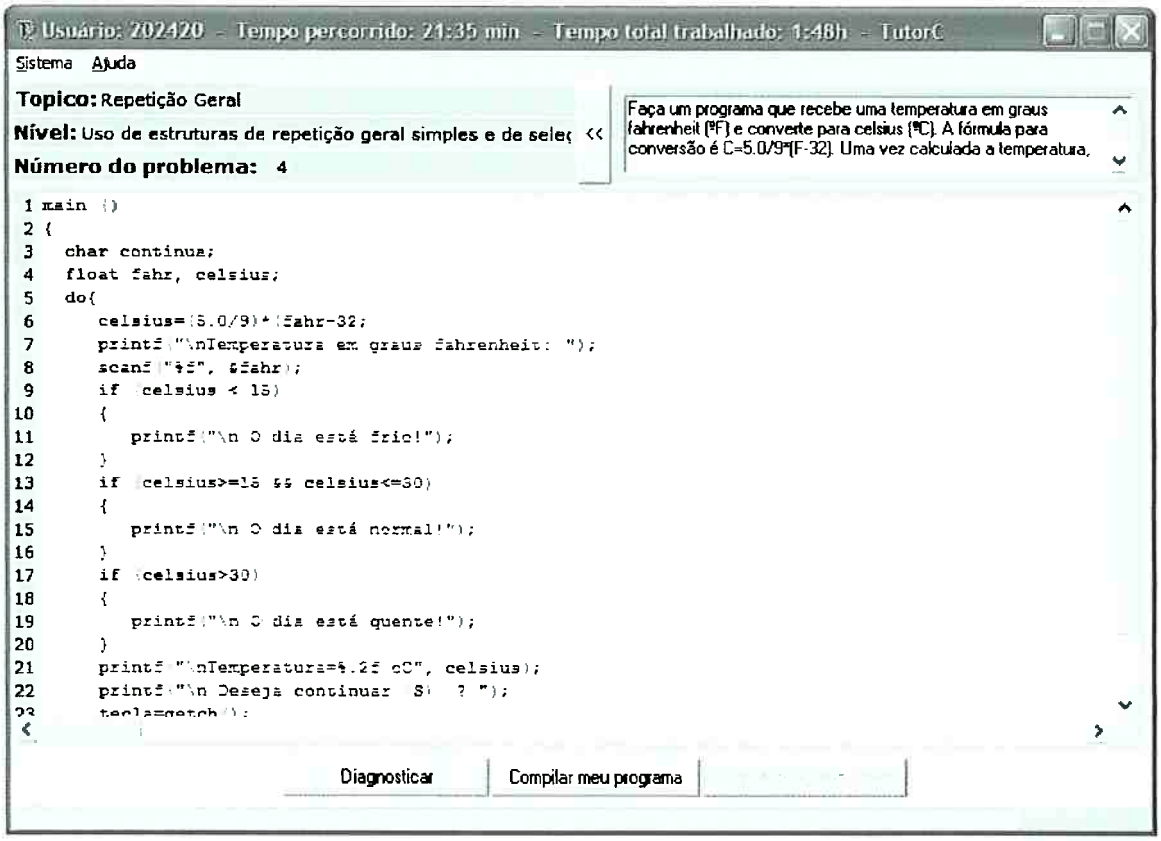


Figura 44 – Janela Diagnóstico

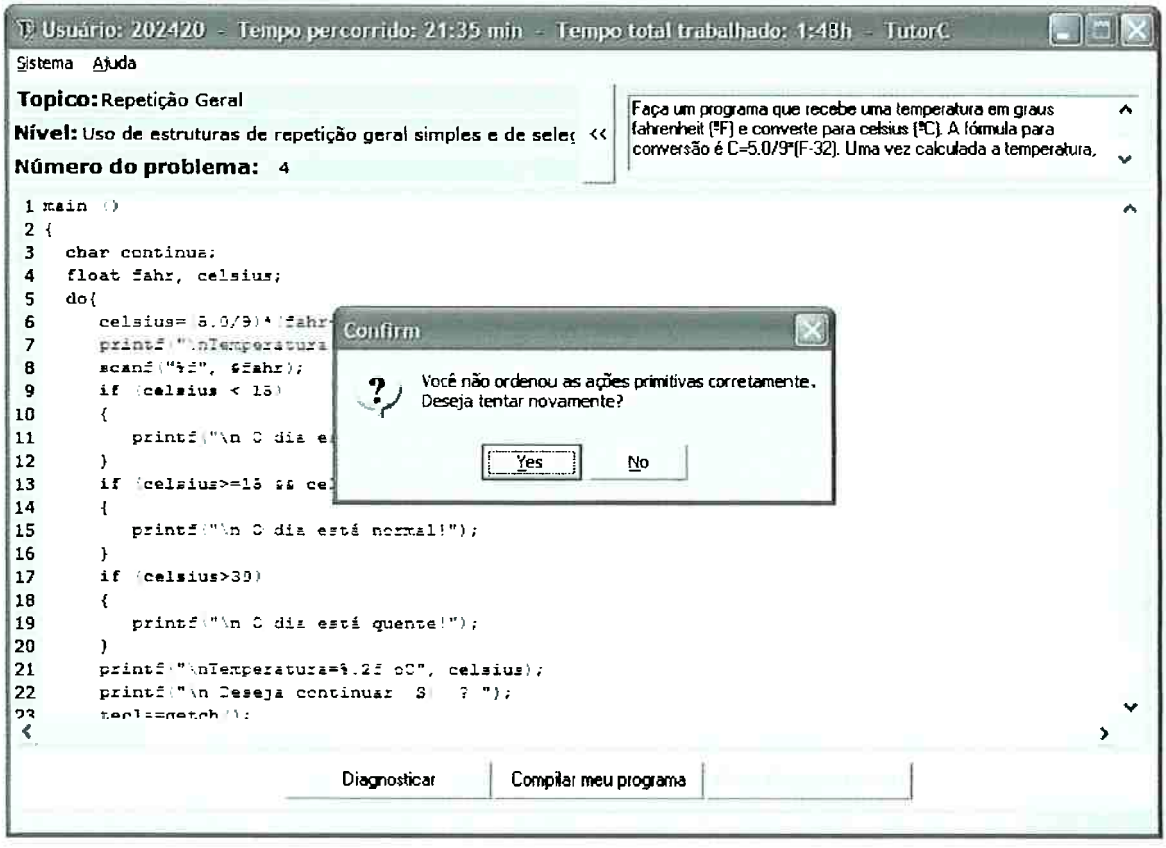


Figura 45 – Diagnóstico Procedimental: Primeira Tentativa

Se houve erro numa segunda tentativa, o sistema apresentará o Plano Cognitivo do aluno e o Plano Cognitivo correto (obtido do sistema BibPC) para que este faça as devidas comparações. Conforme definido no Capítulo 6, a solução de um problema na forma de um conjunto de ações ordenadas é chamado **Plano Cognitivo**. Uma vez que a ênfase do aprendizado está na construção da lógica da solução de problema, TutorC apresenta, inicialmente, os Planos Cognitivos. Porém, o aluno pode optar por comparar as soluções na forma de código, como mostra a figura 46.

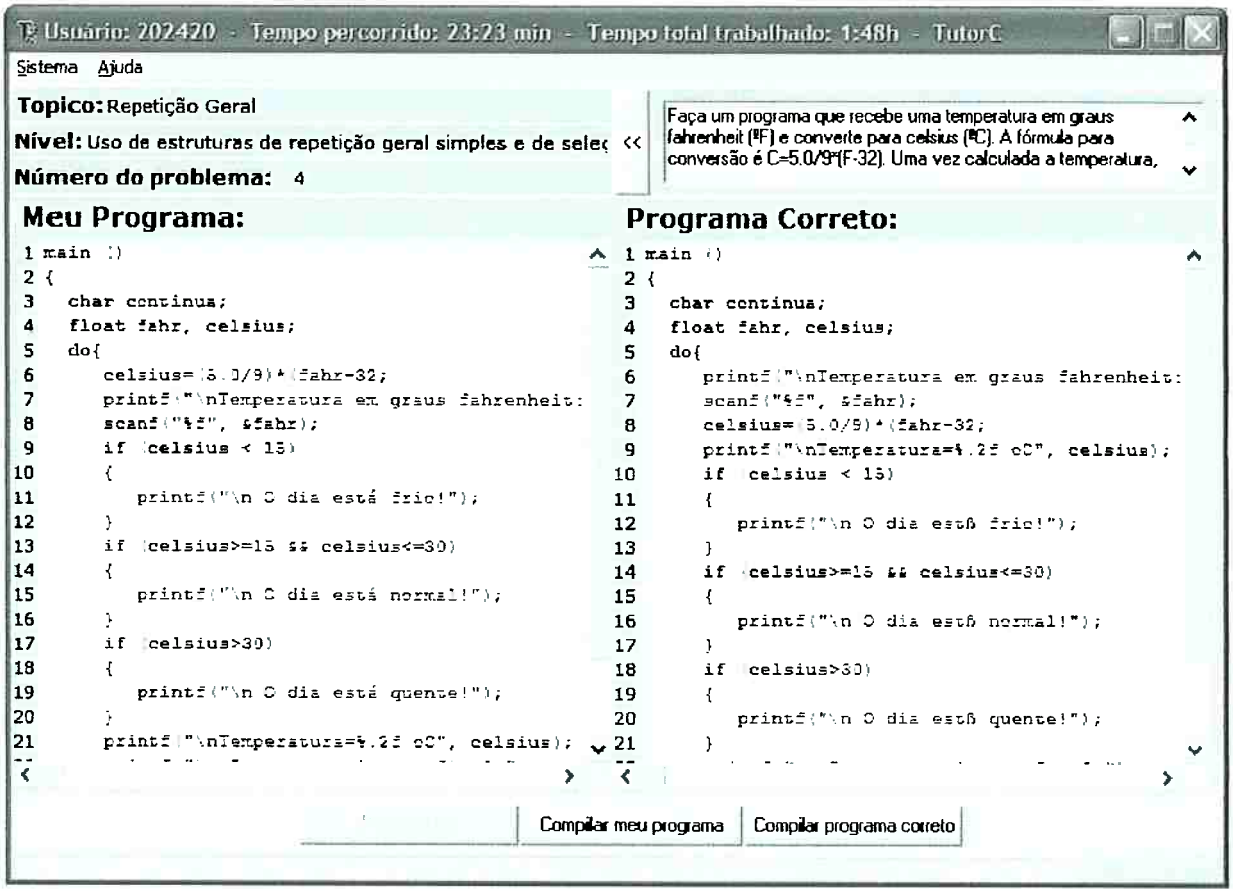


Figura 46 - Diagnóstico Procedimental: Segunda Tentativa

Acionando o botão **Próximo**, o tutor prossegue na realização do Diagnóstico de Mapeamento.

7.2.3.3 Diagnóstico de Mapeamento

Nessa fase, o tutor verifica o mapeamento entre ações primitivas e o nível da implementação, o código C. Os erros e a solução correta são apresentados num relatório conforme mostra a figura 47.

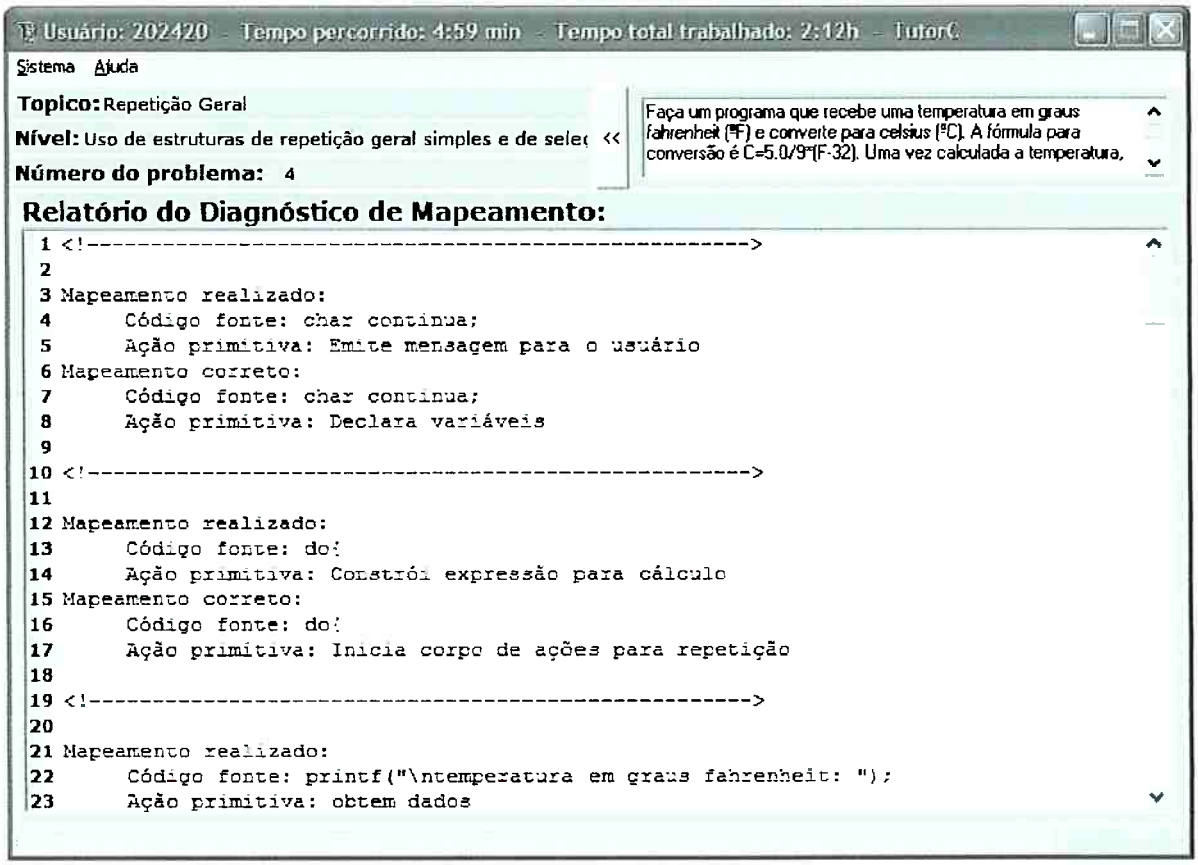


Figura 47 – Relatório de Diagnóstico de Mapeamento

Após a leitura do relatório, o estudante pode retornar à janela da figura 46 para compilar o seu *programa ou o programa correto. Nessa fase, dois botões são apresentados:* **Compilar meu programa** e **Compilar programa correto**.

Quando o aluno decide compilar seu programa ou a solução correta do sistema, o compilador TurboC (BORLAND,2003) é carregado com o código do programa, automaticamente. No compilador, o aluno pode realizar edição, compilação e execução do programa, expandindo suas experiências a respeito do problema em estudo. Pode ainda decidir por compilar seu programa sem ter realizado o diagnóstico procedimental e de mapeamento no tutorC.

7.2.4 Modelo do Estudante

O Modelo do Estudante num sistema tutor mantém, dinamicamente, alguma representação do conhecimento corrente do estudante. O conhecimento armazenado é específico de cada estudante. Pelo menos informações locais - informações sobre o conhecimento da tarefa que está sendo executada na sessão corrente - compõem o Modelo do Estudante. Um modelo mais complexo armazena ainda conhecimento global

sobre o desempenho do estudante, levando em consideração várias sessões de aprendizado. TutorC foi concebido para conter, no Modelo do Estudante, o histórico de todas as sessões de aprendizado do aluno.

O Modelo do Estudante, na maioria dos Sistemas Tutores Inteligentes propostos na literatura, foram projetados para fins de tomada de decisões tutoriais para o aluno. Nos últimos anos, esse tem sido um tema de discussão na área. Shute e Psotka (1994) cita alguns pesquisadores que defendem a liberdade para o aprendiz descobrir, por ele mesmo, o novo conhecimento e outros que defendem que aprendizado guiado é mais produtivo.

TutorC está projetado para ser uma ferramenta cognitiva, definida como aquela que promove expansões cognitivas no aluno (SHUTE;PSOTKA,1994). Assim, o Modelo do Estudante, neste trabalho, tem a finalidade de auxiliar na avaliação preliminar da ferramenta do que habilitar tomadas de decisões tutoriais.

Para utilizar o TutorC, o aluno deve se identificar por meio de um nome de usuário e uma senha, conforme mencionado na seção 7.2.2.2. Isto é necessário para criar e atualizar o seu Modelo de Estudante, ou seja, para que dados específicos de cada aluno possam ser registrados no banco de dados do tutor para posterior avaliação.

7.2.4.1 O Banco de Dados

As principais informações que compõem o Modelo do Estudante são:

- Identificação do aluno;
- Data e tempo de uso do sistema pelo aluno;
- Número de problemas trabalhados por sessão e tópico do curriculum estudado;
- Número de tentativas sem sucesso, por sessão, segundo o tópico e tipo de erro;
- Número de tentativas com sucesso, por sessão, segundo o tópico e tipo de erro.

Segundo as estratégias de diagnóstico implementadas no TutorC, os tipos de erros considerados são:

- Erro de seleção de planos (Diagnóstico de Planos);
- Erro de ordenação (Diagnóstico Procedimental);
- Erro de mapeamento (Diagnóstico de Mapeamento).

As informações do Modelo do Estudante são disponibilizadas, na WEB, por meio de um conjunto de gráficos. A figura 48 apresenta o gráfico da quantidade de problemas

trabalhados pelo estudante até seu último acesso. A quantidade de problemas é apresentada segundo os tópicos do curriculum existente no tutor. No caso da figura 48, o aluno trabalhou em 6 problemas do tópico Básico, 6 problemas do tópico Seleção, 1 problema do tópico Repetição Geral e nenhum problema do tópico Repetição Limitada.

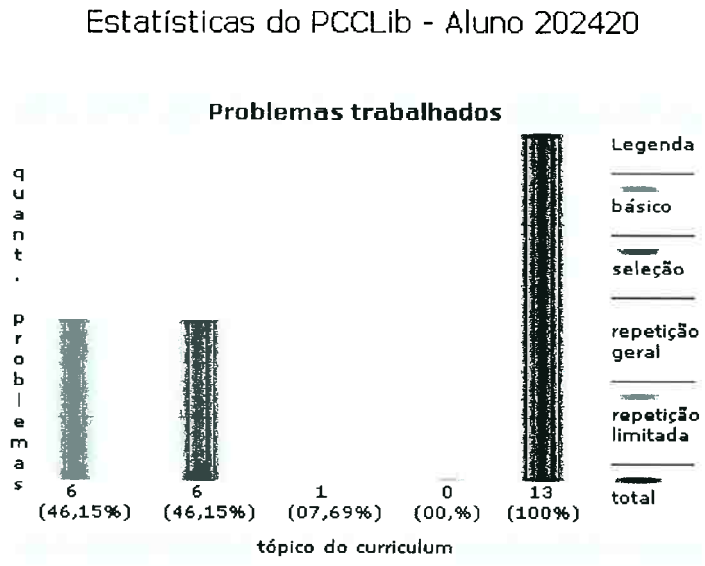


Figura 48 – Problemas trabalhados por um aluno por tópicos do curriculo

A figura 49 apresenta o gráfico que mostra a duração de cada sessão de aprendizado de um dado aluno. Por exemplo, no segundo acesso o aluno esteve trabalhando durante, aproximadamente, dez minutos. O tempo de trabalho na resolução de um problema somente é computado se o aluno submeteu a solução para diagnóstico.

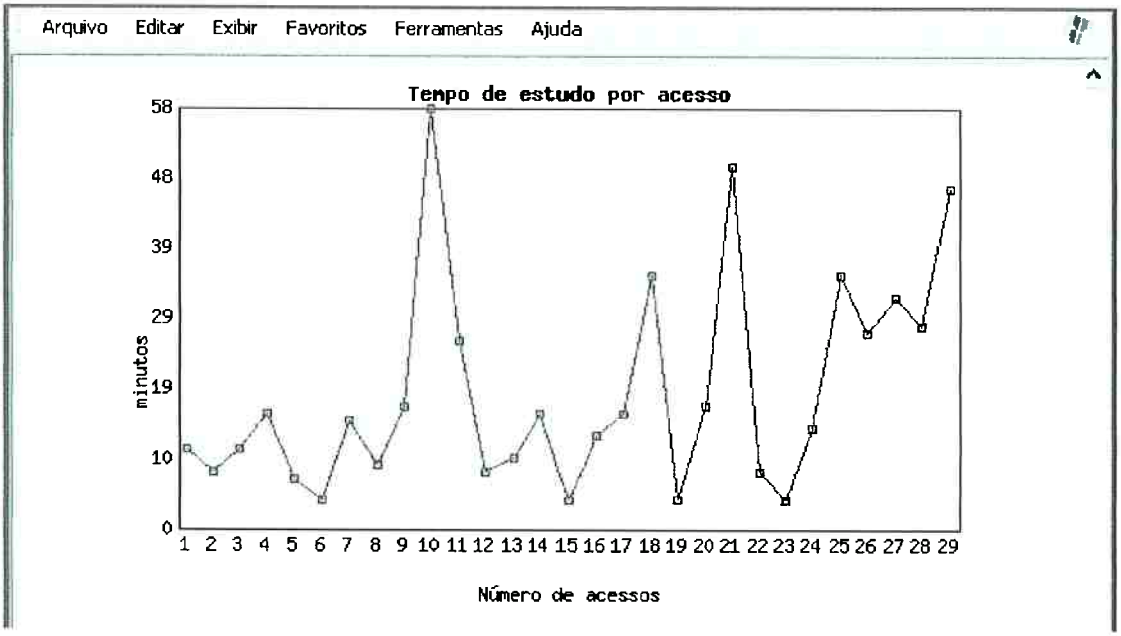


Figura 49 – Gráfico do tempo de estudo de um aluno por sessão

A figura 50 apresenta o gráfico que mostra a evolução do tempo de uso do TutorC pelo aluno. Ou seja, cada acesso, no gráfico, está associado ao tempo total de uso do sistema até esse acesso (tempo acumulado). Por exemplo, na figura 50, o tempo acumulado do aluno desde o primeiro acesso até o 11º acesso é de 183 minutos.

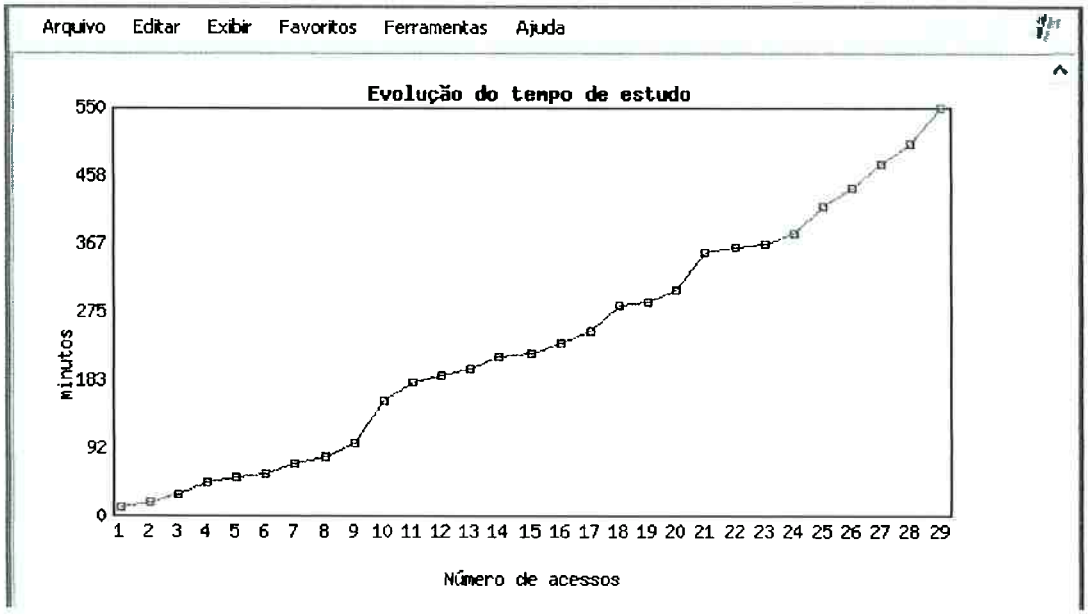


Figura 50 – Evolução do tempo de uso do TutorC por um aluno

O primeiro gráfico da figura 51 mostra a evolução dos erros cometidos pelo aluno desde a primeira sessão até a última. Ou seja, cada sessão no gráfico está associada com a quantidade total de erros cometidos até essa sessão (erros acumulados). O segundo gráfico mostra a evolução dos acertos obtidos pelo aluno desde a primeira sessão até a última. Ou seja, cada sessão no gráfico está associada com a quantidade total de acertos até essa sessão (acertos acumulados).

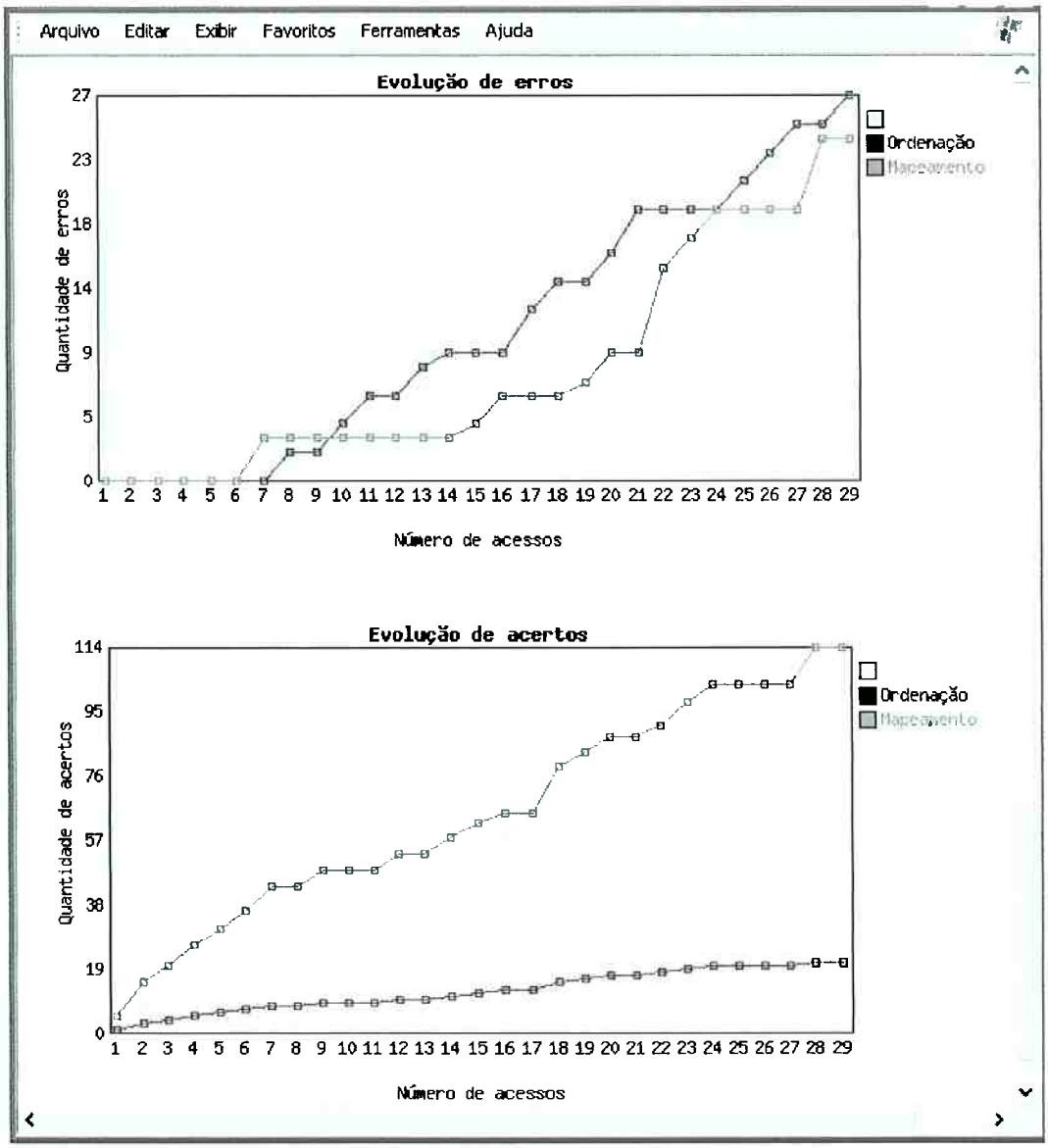


Figura 51 – Evolução dos erros e acertos de um aluno ao longo da utilização do TutorC

7.3 Discussão

O conjunto de ferramentas, BibPC e TutorC, implementam e automatizam o modelo proposto no Capítulo 5. Nesta seção, alguns aspectos relevantes dos sistemas são comparados com sistemas existentes na literatura. Sistemas para ensino da programação, inteligentes ou não, são sistemas muito complexos. Segundo Brusilovsky (1995) cada componente requer vários anos de pesquisa por uma pessoa e somente se o pesquisador fizer parte de um time grande terá a chance de projetar algo interessante que possa ser usado dentro do contexto de um Sistema Tutor Inteligente completo. Ainda assim, pesquisadores num grupo concentram esforços em componentes inteligentes; componentes fora dos interesses são deixados de lado, mesmo que sejam importantes para o usuário.

Um dos objetivos desta tese é a utilização dos sistemas BibPC e TutorC em sala de aula, ainda que de forma preliminar. Assim, a maioria dos módulos que compõem um Sistema Tutor Inteligente foram implementados, porém alguns com aspectos mais relevantes do que outros. De acordo com o modelo de processo proposto no Capítulo 5, a relevância de TutorC concentra-se no Módulo Especialista, Interface do Aluno e Módulo Diagnóstico.

7.3.1 Autoria

BibPC é uma ferramenta de autoria que permite a manutenção e expansão do conhecimento especialista do TutorC, tornando os sistemas desvinculados de interesses específicos da instituição na qual se originam. Usando BibPc e TutorC, cada professor em sua instituição de ensino pode modelar planos e problemas de programação de seu interesse.

Segundo Brusilovsky (1995), a ausência dessa característica na grande maioria dos sistemas tutores da literatura é um dos motivos pelo qual a aplicação desses sistemas, em salas de aula, não melhorou em relação ao que estava há vinte anos atrás. Brusilovsky (1995) cita algumas justificativas pela ausência de aplicações desses sistemas em contextos reais:

- Para um sistema ser útil ele tem que suportar alguma parte significativa do trabalho do professor ou do aluno. A maioria dos tutores suporta uma parte mínima e insuficiente para uso em sala de aula;
- As ferramentas são construídas orientadas estritamente para uso na entidade onde foi desenvolvida, impossibilitando implementar estruturas de cursos diferentes;
- Muitos sistemas criados de forma incompleta, mesmo contendo idéias interessantes mas impossibilitados de reuso, não chegaram em salas de aula porque as idéias morreram com os sistemas quando pesquisadores mudaram seus interesses de pesquisa.

7.3.2 Fidelidade Epistêmica

Vários tutores propostos na literatura têm sua representação interna na forma de planos e metas de programação (JOHNSON;SOLOWAY,1985;HAHN ET AL.,1997). Porém, o modelo interno é tratado como um artefato de uso específico do tutor, o qual não é utilizado para comunicar-se com o aluno através da interface como é a proposta do TutorC. A abordagem desses tutores traz várias conseqüências tais como:

- A representação do conhecimento interno é superficialmente relatada na literatura (JOHNSON;SOLOWAY,1985; BONAR;CUNNINGHAM,1988);
- Não é possível reuso de planos de programação pela comunidade (Wallingford, 1998);
- O processo de diagnóstico dos tutores que utilizam modelo interno (módulo especialista) diferente do modelo externo (interface do aluno) é bastante complexo uma vez que precisam comparar representações diferentes;
- A maioria dos sistemas tutores dispõem, para o estudante, somente um editor, às vezes estruturado, e um sistema de ajuda *on-line* sobre conceitos e exemplos da linguagem. Assim, é possível que o desenvolvimento das habilidades cognitivas de alunos seja semelhante aos resultados obtidos no ensino tradicional. O aluno programa como se estivesse diante de um compilador, tendo como apoio somente a própria linguagem de programação, não tirando proveito de outras possíveis representações externas (COX;BRNA,1995; GEORGE,2000).

Diferentemente desse panorama, TutorC implementa uma abordagem diferente da tradicional, a abordagem descrita no Modelo de Processo de Construção de Programas para Aprendizes. Fazendo uso de uma representação para planejamento em programação foi possível trazer para o modelo externo do TutorC (a interface), os mesmos componentes internos ao tutor (BibPC). Essa é uma característica importante, denominada por Wenger (1987) de fidelidade epistêmica, dificilmente encontrada em tutores inteligentes em geral.

Segundo Wenger (1987), num sistema tutor ideal que possui fidelidade epistêmica, a interface é estritamente uma representação externa da especialidade que o sistema possui internamente. Quando representações internas (ao tutor) e externas (interface) possuem fidelidade epistêmica, o sistema adquire um poder de comunicação que lhe garante características de inteligência:

- O sistema pode comunicar seu conhecimento de solução de problema;
- O sistema pode controlar cada passo de comunicação e monitorar seus efeitos.
- Atividades de solução de problema do estudante podem ser seguidas pelo sistema e facilmente comparadas com decisões do modelo interno;

Para alcançar fidelidade epistêmica, o projeto da representação externa necessita de um comprometimento com o entendimento profundo do domínio. Interfaces expandem a comunicabilidade do modelo de domínio e fornecem influência para aprendizado, portanto seu projeto pode ser considerado uma tarefa pedagógica. A coexistência de um modelo interno e uma interface que possuam o mesmo grau de fidelidade epistêmica abre a possibilidade de transformar fenômenos não observáveis ou implícitos no tutor,

em objetos que podem ser visualizados e estudados. Uma conseqüência importante dessa idéia é que o aprendizado da interface não é algo separado do aprendizado da especialidade embutida no sistema.

No TutorC, a solução do problema pode ser representada em dois níveis pelo aprendiz: (i) no nível mental utilizando os componentes de planejamento e (ii) no nível da implementação, no qual a representação da solução é o programa codificado. O mapeamento entre as duas representações é um caminho percorrido explicitamente pelo estudante. Isto permite a comparação da intenção com a implementação final da solução do problema (VALENTE,1995).

7.3.3 Diagnóstico

Na maioria dos sistemas tutores para programação, o estudante constrói seu programa manipulando a própria linguagem de programação. Quando o estudante constrói seu programa no nível da implementação, seu comportamento não pode ser observado, o tutor recebe apenas o produto final: o programa do estudante. Tipicamente, nessa situação os tutores utilizam o chamado diagnóstico por **reconhecimento de plano** ou **reconstrutivo** como no caso dos tutores PROUST, LAURA, MENO-II e C-Tutor, resumidos no Capítulo 4, a qual é considerada uma abordagem complexa e imprecisa para alguns domínios. Particularmente, domínios de solução de problema, tal como o da programação, soluções tendem a ser objetos complexos cuja forma não pode ser entendida independentemente do caminho pelo qual elas foram produzidas. Consequentemente, diagnóstico requer a reconstrução do raciocínio de solução de problema ou uma forma idealizada do caminho da solução de problema.

Wenger (1987) ressalta que essa abordagem exige estratégias de diagnóstico complexas e ainda a presença de modelo de falhas ou uma teoria de bugs. Vários tutores foram desenvolvidos baseados sob a abordagem de biblioteca de erros. A desvantagem é que essa abordagem está teoricamente baseada sobre a noção de erros cognitivos em procedimentos específicos tornando difícil a reusabilidade e manutenção do sistema. Assim, somente erros armazenados são reconhecidos, erros novos são ignorados.

Saber sobre o comportamento do estudante, significa conhecer os passos intermediários que o estudante tomou para construir sua solução final. A observação de passos intermediários na construção da solução de problema somente é útil se eles podem ser entendidos pelo sistema.

Num domínio como programação, esse entendimento pode ser muito problemático. Seguir a história do desenvolvimento de um programa, somente a partir do código, pode requerer muita engenhosidade. A interpretação precisa de passos intermediários requer um modelo de raciocínio completo. Sem tal modelo esses passos não podem ser interpretados, uma vez que eles refletem um processo e não um produto (WENGER,1987).

TutorC adota um modelo de raciocínio para programação, o modelo de processo detalhado no Capítulo 5. Tal modelo é proposto ao aluno na interface do TutorC, a qual dispõe de estratégias e primitivas de alto nível, que vão além da linguagem de programação. Como já mencionado, o professor introduz conhecimento em BibPC baseado nesse mesmo modelo.

Assim, TutorC pode acompanhar, com precisão, o comportamento do estudante durante todo o processo de programação até a obtenção do produto final: o programa. Isto garante estratégias relativamente simples para diagnosticar erros na solução final proposta pelo estudante como também diagnosticar erros durante o processo de programação.

7.3.4 Ferramenta Cognitiva e Construtivismo

Alguns pesquisadores apoiam a tendência de mover a construção tradicional dos Sistemas Tutores Inteligentes em direção ao projeto de sistemas como ferramentas cognitivas (SHUTE;PSOTKA,1994). Nessa visão, ferramentas cognitivas manipuladas pelos estudantes são vistas como instrumentos que promovem pensamento construtivo. Isso significa fazer com que o estudante supere suas limitações cognitivas disponibilizando um ambiente onde ele possa se engajar em operações cognitivas as quais ele ainda não seria capaz de realizar sozinho. A maioria dos tutores inteligentes se preocupam e realizam muito mais diagnóstico e decisões tutoriais, do que promovem expansões cognitivas no estudante, não podendo assim serem consideradas ferramentas cognitivas (WENGER,1987). Ainda, de acordo com a noção de computadores como ferramentas cognitivas, aprendizes deveriam ter, eles próprios, a opção de alterar o grau de controle, como por exemplo usar um ambiente didático mais restritivo e direcionado ao aprendizado de uma habilidade cognitiva em particular ou um ambiente de descoberta.

De acordo com tais definições, TutorC trata-se de uma ferramenta cognitiva uma vez que disponibiliza ao aprendiz em programação, componentes que representam a especialidade de programadores experientes.

TutorC implementa a idéia de que se um aprendiz não conhece planos de programação, como poderá criá-los e utilizá-los para compor um programa, somente a partir da linguagem de programação?

TutorC é projetado baseado em premissas da Ciência Cognitiva que afirma que pessoas quando interagem com um ambiente, por exemplo um artefato de tecnologia, formam modelos mentais internos a respeito dos elementos com os quais estão interagindo (NORMAN,1983). TutorC propõe metas, planos, ações e estratégias de programação que manipulados por aprendizes podem estimulá-los a criarem seus próprios modelos mentais a respeito desses componentes.

Ben-Ari (2001) observou uma característica peculiar na área da Ciência da Computação que difere das ciências naturais: o aprendiz dificilmente tem alguma estrutura de conhecimento ou paradigma que permita "construir", sobre ela, a estrutura da Ciência da Computação. Uma vez que, no construtivismo (PIAGET,1977), o aproveitamento do aprendiz ocorre por meio da construção de conhecimento, a partir de estruturas de conhecimento que este já possui anteriormente, o conhecimento modelado na BIbPC e disponibilizado em TutorC, preenche parte dessa lacuna de conhecimento, tipicamente existente em aprendizes de programação.

7.4 Trabalhos Relacionados

Dos sistemas propostos para aprendizagem de programação algoritmica, da literatura, o trabalho mais próximo ao TutorC é o tutor BRIDGE (BONAR;CUNNINGHAM,1988), descrito no Capítulo 4. Porém, algumas diferenças importantes podem ser apontadas. Apesar de nenhum artigo sobre BRIDGE descrever um paralelo desse sistema com a tarefa de planejamento (*Planning*) da IA, é possível relacionar as descrições de BRIDGE com componentes dessa área de pesquisa. Esse paralelo permite comparar BRIDGE com TutorC. As descrições não-procedurais manipuladas no estágio 1 de BRIDGE podem ser vistas como metas de programação. Um subconjunto dessas metas, a ser descoberto pelo aprendiz, são pretendidas para serem alcançadas na solução de um problema. Descrições de agregações do estágio 2 podem ser vistas como nomes de planos a serem implementados, pelo aprendiz, no estágio 3, por meio de ações de programação. O estágio 4 é apenas uma representação alternativa (pseudo-código) do estágio 3, mais próxima da linguagem de programação.

Diferentemente de BRIDGE, em TutorC, metas são propostas no nível do domínio do problema e são fornecidas explicitamente ao aprendiz. As metas do problema devem ser associadas a planos de programação, ou seja, o plano não é um artefato a ser criado pelo aprendiz como em BRIDGE. A associação é guiada pelas descrições explícitas dos planos, constituídas pelos critérios de adequabilidade e ainda pelos blocos de código que implementam os planos. O plano pode, também, ser visualizado, pelo aprendiz, como uma rede de tarefas primitivas. No TutorC, o aprendiz compõe um programa, ordenando não somente planos, mas também ações primitivas, no caso de planos deslocalizados. Em BRIDGE, não é mencionado como um plano deslocalizado é implementado pelo aprendiz. Ações de programação, em TutorC, também diferem de BRIDGE. Em TutorC, uma ação de programação é parte de uma tarefa primitiva, que por sua vez é parte de um plano. Assim, uma tarefa primitiva é constituída de uma descrição em linguagem natural (a ação), um esquema (código com meta-variável) e uma posição dentro do plano, o qual faz parte. Essas relações ficam explícitas na interface do TutorC, para o aprendiz. A instanciação do programa com dados do domínio do problema, em TutorC, ocorre por meio das meta-variáveis as quais apóiam o aprendiz nessa tarefa. Não foram encontradas informações de como código é instanciado no sistema BRIDGE.

Por um lado, BRIDGE parece permitir maior liberdade ao aprendiz, em suas atividades de programação, do que em TutorC. Porém de outro lado, é restritivo no sentido de que o aprendiz não pode alcançar um programa incorreto e aprender com a experiência de seus erros no nível da codificação e execução. Essa característica de BRIDGE está diretamente relacionada com a ausência de conhecimento de planejamento explícito na interface de BRIDGE, como descrito acima. Nesse sistema, o aprendiz associa (mentalmente) metas de um problema a ser resolvido com possíveis metas de programação (estágio 1). Porém enquanto o aprendiz não escolher o conjunto de metas correto, o tutor não permite prosseguir. Assim, o aprendiz entra num ciclo de tentativa-e-erro, uma vez que não há nenhum recurso para guiá-lo nessa escolha. O mesmo ocorre no estágio 2 (planos) e no estágio 3 (ações).

Com relação a BRIDGE e demais tutores que possuem conhecimento especialista, modelado na forma de planos e metas, TutorC se mostra um sistema expansível e que permite autoria do currículo a ser trabalhado. Usuários (professores) podem inserir material didático de seu interesse através da BibPC.

7.5 Resumo do Capítulo

Este capítulo descreve os aspectos de implementação do modelo proposto no capitulo 4. Dois sistemas são implementados: BibPC (Biblioteca para Programação Cognitiva) e TutorC (Sistema Tutor Inteligente para Aprendizagem de Programação em C). BibPC é um sistema capaz de capturar, através da WEB, conhecimento de planejamento em programação de uma forma estruturada e reutilizável. Uma vez que esse tipo de conhecimento é muito difícil de ser encontrado, o sistema torna-se útil para toda a comunidade interessada, como por exemplo desenvolvedores de ferramentas cognitivas para aprendizado de programação e educadores da área. BibPC é composta por (1) uma interface WEB, denominada **Interface do Professor** e (2) uma estrutura de banco de dados SQL. A interface permite que professores ou programadores experientes possam modelar componentes de planejamento e de implementação de programas. Os principais componentes modelados por um professor na BibPC são: Planos de Programação, Problemas e Soluções de Problemas.

A interface está disponível para acesso por qualquer computador conectado à **Internet**, desde que os usuários estejam habilitados. A habilitação é realizada em níveis de acesso. O usuário denominado de **professor-coordenador** é responsável por três tarefas:

- Cadastramento de usuários, possivelmente um conjunto de professores;
- Divisão do curriculum a ser estudado, uma vez que planos e problemas formam o material instrucional e devem ser enquadrados em tais divisões;
- Inserção e manutenção de planos de programação;
- Manutenção de problemas e suas soluções.

O usuário denominado simplesmente de professor, é responsavel por modelar problemas e métodos, tal como descrito no capítulo 6. Uma amostra das janelas de BibPC, são apresentadas, assim como algoritmos descrevendo as ações principais do professor-coordenador e demais usuários do sistema. A existência de soluções corretas e alternativas para problemas (métodos), na BibPC, permite sua utilização em sistemas de aprendizagem que realizam diagnóstico, tais como sistemas tutores inteligentes.

TutorC, o segundo sistema apresentado neste capítulo, é proposto para promover aprendizado da construção de modelos mentais de programas. A aquisição dessa habilidade é apontada como uma das maiores dificuldades no aprendizado de programação. TutorC disponibiliza ao aprendiz, componentes de planejamento e de implementação de programas, minimizando a dificuldade do aprendiz em criar e traduzir planos mentais para planos executáveis no computador. O sistema é implementado segundo a arquitetura proposta pela área de Sistemas Tutores Inteligentes de IA

(WENGER,1987; SHUTE;PSOTKA,1994; SELF,1995), sendo composto por cinco módulos: Módulo Especialista, Interface do Estudante, Módulo Diagnóstico, Modelo do Estudante e Compilador. TutorC foi implementado em linguagem C++ com acesso a banco de dados SQL. O conhecimento do **Módulo Especialista** do TutorC é obtido a partir do sistema BibPC. Assim, TutorC disponibiliza em sua interface com o estudante, o **Módulo Interface do Aluno**, um conjunto de componentes de planejamento para programação que permitem a construção de programas de uma forma independente dos detalhes de sintaxe da linguagem C. Sem tais elementos, como ocorre com a maioria dos tutores da literatura, aprendizes são forçados a compor soluções usando apenas a linguagem de programação, a fim de atingir metas de alto-nível. TutorC guia o estudante numa estratégia *top-down* durante a composição de um programa. Ou seja, a solução é alcançada partindo-se das metas do problema até atingir o nível da implementação.

TutorC mostra, além do enunciado do problema, o conjunto de metas a serem realizadas para atingir a solução do problema. Na construção do modelo do programa, o sistema oferece uma variedade de planos de programação que podem ser selecionados pelo estudante através dos critérios de adequabilidade. Planos, no TutorC, são representados por uma rede de tarefas primitivas, compostas de descrições textuais e fragmentos de código generalizados com meta-variáveis. O modo como o professor decompõe as metas de um problema, em BibPC, pode conduzir à necessidade do aluno trabalhar com planos mais simples ou mais complexos. Em conseqüência, o estudante é incentivado a raciocinar sobre hierarquia de planos e metas. Ordenando tarefas primitivas de programação entre planos e, ainda, mapeando-as em declarações em C corretas fornecidas pelo tutor, o aprendiz finaliza a construção do seu programa.

Em resumo, a solução de um problema, no TutorC, pode ser representada em dois níveis pelo aprendiz: (i) no nível mental utilizando os componentes de planejamento e (ii) no nível da implementação, no qual a representação da solução é o programa codificado. O mapeamento entre as duas representações é um caminho percorrido explicitamente pelo estudante. Isto permite a comparação da intenção com a implementação final da solução do problema (VALENTE,1995).

TutorC realiza diagnóstico em níveis de abstração diferentes, sendo capaz de orientar o aluno em: (1) seleção de planos de programação para metas de problema (**Diagnóstico de Planos**); (2) ordenação de planos e tarefas primitivas (**Diagnóstico Procedimental**) e (3) mapeamento de tarefas primitivas sobre declarações em C (**Diagnóstico de Mapeamento**). A qualquer momento, tipicamente após ter seu programa finalizado e diagnosticado, o estudante pode disparar, a partir de TutorC, o

Compilador Turbo C++ Versão 3.0 da Borland (2003). Nesse novo ambiente, o aluno pode compilar e executar seu programa, assim como fazer quaisquer alterações que vão além dos planos de programação propostos no TutorC. O **Módulo do Estudante** do tutor, identifica unicamente cada usuário mantendo um histórico de informações de cada aluno. Essas informações podem ser vistas na WEB pelo professor, na forma de gráficos, para fins de acompanhamento do desempenho e evolução do aprendizado do aluno.

8 APLICAÇÃO DO MODELO PROPOSTO EM SALA DE AULA

Neste capítulo é descrita a aplicação em sala de aula e laboratório da abordagem proposta na tese para um grupo de alunos selecionado. Esta aplicação teve como objetivo realizar uma avaliação preliminar do potencial do Modelo de Processo de Construção de Programas para Aprendizes proposto.

8.1 Metodologia da Aplicação

Na FACENS, o aluno com nota de avaliação inferior a cinco, é obrigado a realizar a chamada prova de reavaliação, sob o risco de ter sua primeira nota dividida por dois. O conceito dessa prova difere da chamada prova substitutiva adotada em várias instituições. Na FACENS, se o aluno obtiver uma nota de reavaliação inferior à prova correspondente, sua nota final será a média das duas notas, caso contrário sua nota final será a nota da reavaliação.

Estima-se que numa classe de 48 alunos de Engenharia Elétrica do período noturno, 41% destes recebem notas menores do que cinco na primeira prova de introdução à programação (Módulo 1). A tabela 35 mostra essa relação nas classes de Engenharia Elétrica dos dois últimos anos.

Tabela 35 – Estimativa de alunos com dificuldade no aprendizado de programação algorítmica por meio da abordagem tradicional

Turma	Ano	Total de alunos	Alunos sob reavaliação no Módulo 1	
			Quantidade	*%*
784	2002	47	19	40
780	2002	55	25	46
374	2003	45	15	33
387	2003	46	22	47
Média		**48**	**20**	**41**

A aplicação consistiu de um mini-curso de introdução à programação C baseado na Abordagem Cognitiva proposta e utilização do TutorC pelos alunos sob reavaliação no Módulo 1 de duas turmas do curso noturno de Engenharia Elétrica da FACENS (turmas 374 e 387 de 2003). O currículo anual da disciplina ICCN, na FACENS, está dividido em quatro módulos. Para cada módulo é realizada uma prova de teoria e uma de laboratório, assim como suas respectivas reavaliações no final do semestre.

O currículo abordado no mini-curso correspondeu ao primeiro módulo (Módulo 1) da disciplina, o qual compreende os conceitos básicos de programação, compreensão de programas através de simulações e resolução de problemas usando estruturas de decisão e de repetição. Porém, o mini-curso envolveu somente resolução de problemas usando planos de programação, conforme a proposta da tese.

A duração do mini-curso foi de cinco semanas nos meses de agosto e setembro de 2003. Os alunos dos cursos noturnos da FACENS caracterizam-se por possuírem pouca disponibilidade de tempo, uma vez que a grande maioria trabalha em período integral e à noite freqüenta as aulas dos cursos de Engenharia. Assim, foram disponibilizados três horários diferenciados para o mini-curso: terças-feiras das 18h20min às 19h10min, sábados das 10h às 11h40min e das 13h às 14h40min. As presenças dos alunos foram controladas, assim como o tempo de uso do TutorC utilizando os recursos disponíveis no tutor.

Do grupo de alunos que participaram do mini-curso, vinte e seis realizaram a prova de teoria e quinze realizaram também prova de laboratório. Durante a prova de teoria, foi permitida a consulta ao conjunto de planos de programação na apostila distribuída aos alunos (apêndice A). Na prova de laboratório esta consulta não foi permitida. O apêndice D mostra os modelos de provas utilizados.

Os critérios adotados para utilização do resultado do desempenho de um dado aluno na avaliação preliminar descrita neste capítulo foram:

- O aluno obteve uma ou mais notas inferiores a cinco nas primeiras duas provas (teoria e/ou laboratório) da disciplina ICCN (Introdução à Computação e Cálculo Numérico).
- O aluno assistiu um mínimo de dez horas de aulas presenciais do mini-curso;
- O aluno realizou um mínimo de oito horas de trabalho individual no TutorC, em horários de sua livre escolha. Além do tempo de uso do tutor, outras condições foram impostas: a seleção diversificada de problemas no tutor, assim como uma quantidade mínima de problemas resolvidos (quatro problemas de cada tópico totalizando um mínimo de dezesseis problemas).

8.2 Abordagem Tradicional x Abordagem Cognitiva

Deseja-se definir, inicialmente, o que é a abordagem tradicional de ensino da programação. Tradicionalmente, numa sala de aula, o professor enquanto explica a construção ou compreensão de um programa, identifica objetivos a serem alcançados, extraídos a partir do enunciado, relembra conceitos da linguagem de programação e

mostra o mapeamento entre objetivos e estruturas da linguagem. Fornece ainda o significado das linhas de código utilizada no programa, explica instanciações de dados do problema no código, entre outras atividades. Porém, todo esse conhecimento é transmitido ao aluno de maneira informal, ou seja, não é documentado. Uma vez que, nessa situação, a carga cognitiva exigida do estudante é grande, este acaba por não assimilar todo o conhecimento, com sucesso. Esse contexto define a chamada abordagem tradicional.

A abordagem cognitiva proposta neste trabalho, formaliza alguns tipos de conhecimento manipulados na abordagem tradicional: decomposição de problemas em metas a serem realizadas, planos e ações primitivas de programação, mapeamentos entre metas e planos, mapeamentos entre dados do problema e ações primitivas de programação. Em resumo, a grande diferença entre as abordagens reside no aspecto da formalização e documentação dos tipos de conhecimento necessários à atividade de programação.

8.3 Resultados

Os dados utilizados nesta seção foram obtidos da secretaria *on-line* acessível pelo *site* da FACENS e os mais relevantes podem ser vistos no anexo A. A tabela 36 fornece a média e desvio padrão das notas de avaliação e reavaliação de teoria e laboratório dos módulos 1 e 2 das turmas de ICCN dos últimos dois anos.

Tabela 36 – Média e desvio-padrão das notas de teoria e laboratório dos módulos 1 e 2 das turmas de ICCN de 2002 e 2003

	Ano	Turma	No. alunos	Alunos sob reavaliação			Avaliação na Abordagem Tradicional		Reavaliação		
				No.	%	G*	$\overline{\mu}$	σ	$\overline{\mu}$	σ	Ab*
Teoria Módulo 1	2002	780	55	24	43	-	1,18	1,59	2,40	2,59	A.T.
		784	47	19	40	-	1,71	1,60	2,88	2,07	A.T.
	2003	374	45	17	37	10	2,7	2,13	5,5	1,82	A.C.
		387	46	22	47	16	2,44	1,79	6,4	1,95	A.C.
Laboratório Módulo 1	2003	374	45	15	33	11	2,5	2,51	6,9	3,21	A.C.
		387	46	20	43	16	2,13	1,96	6,31	2,76	A.C.
Teoria Módulo 2	2003	374	45	24	53	-	1,21	1,68	3,06	2,72	A.T.
Laboratório Módulo 2	2003	387	46	25	54	-	1,96	1,92	3,58	3,70	A.T.

* A.T. – Abordagem Tradicional; A.C. – Abordagem Cognitiva.

* alunos participantes do grupo selecionado

A figura 52 apresenta o histograma das notas obtidas na Avaliação de Teoria do Módulo 1 após o uso da Abordagem Tradicional e na Reavaliação após o uso da Abordagem Cognitiva. Por exemplo, na abordagem tradicional seis alunos e na abordagem cognitiva três alunos tiraram notas entre 3,00 e 3,99 representados no quarto intervalo de notas do histograma.

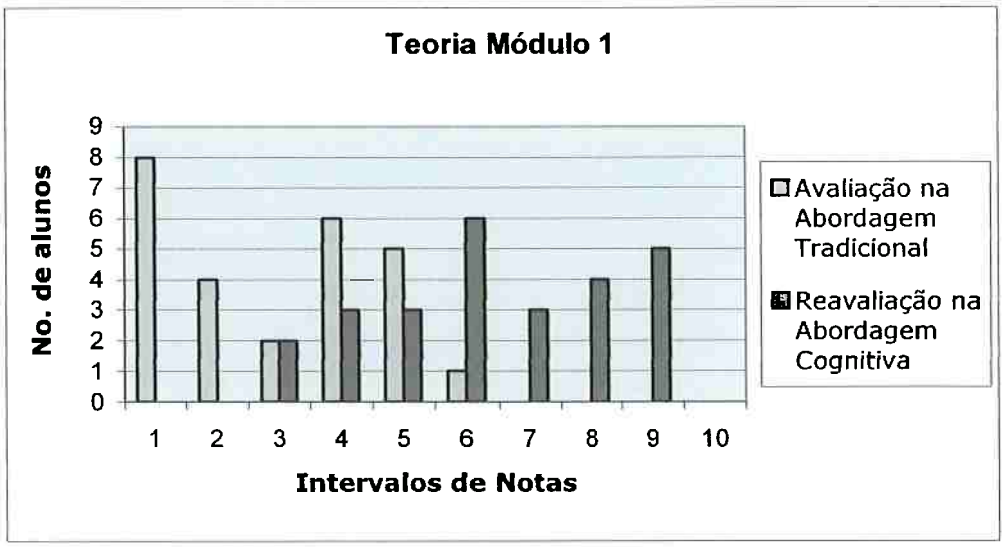


Figura 52 – Histograma das notas da prova de teoria M1 após o uso da abordagem tradicional e após o uso da abordagem cognitiva

A figura 53 apresenta o histograma das notas obtidas na Avaliação de Laboratório do Módulo 1 após o uso da Abordagem Tradicional e na Reavaliação após o uso da Abordagem Cognitiva.

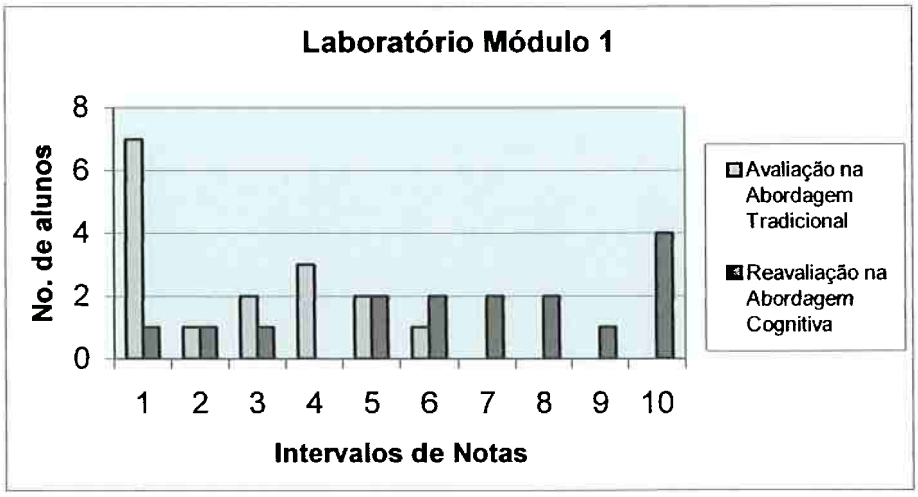


Figura 53 – Histograma das notas da prova de laboratório M1 após o uso da abordagem tradicional e após o uso da abordagem cognitiva

Para facilitar a comparação dos resultados obtidos com o uso das duas abordagens, foram calculados a média, o desvio-padrão e a distribuição normal das notas nas duas situações. Quantitativamente, a dispersão das notas pode ser caracterizada pelo desvio-padrão (σ) do conjunto de notas, definido como:

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{\mu})^2}$$

onde x_i é o resultado da i-ésima nota, N é o número total de notas e $\overline{\mu}$ é a média aritmética das notas dada por:

$$\overline{\mu} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

Assim, o desvio-padrão fornece uma medida do grau de dispersão das notas em relação à média. A tabela 37 apresenta a média e o desvio-padrão das notas do grupo para cada uma das provas.

Tabela 37 - Média e desvio-padrão das notas de provas do grupo de alunos

	Teoria Módulo 1		Laboratório Módulo 1	
	$\overline{\mu}$	σ	$\overline{\mu}$	σ
Avaliação na Abordagem Tradicional	2,54	1,94	2,31	2,18
Reavaliação na Abordagem Cognitiva	6,10	1,89	6,63	2,91

A partir da média e desvio-padrão obtidos em cada uma das abordagens, foram gerados os gráficos da função de distribuição normal, f(x), das notas (x) dada por:

$$f(x) = \frac{1}{\sqrt{2\pi}\,\sigma}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

A figura 54 mostra os gráficos das funções de distribuição normal das notas de provas de teoria realizadas com as duas abordagens.

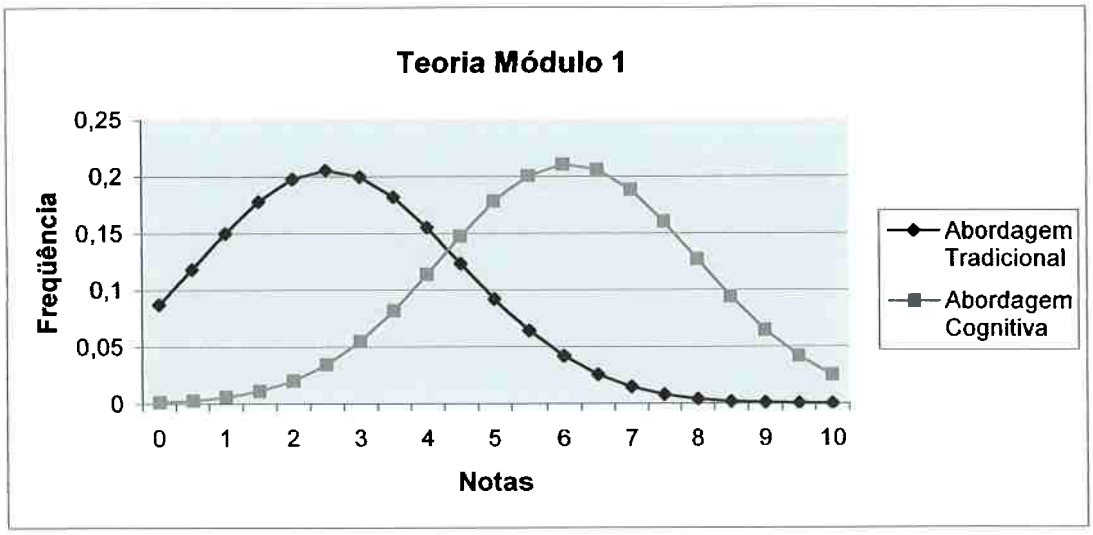


Figura 54 – Distribuição normal das notas de teoria com o uso da abordagem tradicional e com o uso da abordagem cognitiva

A figura 55 mostra os gráficos das funções de distribuição normal das notas de provas de laboratório realizadas com as duas abordagens.

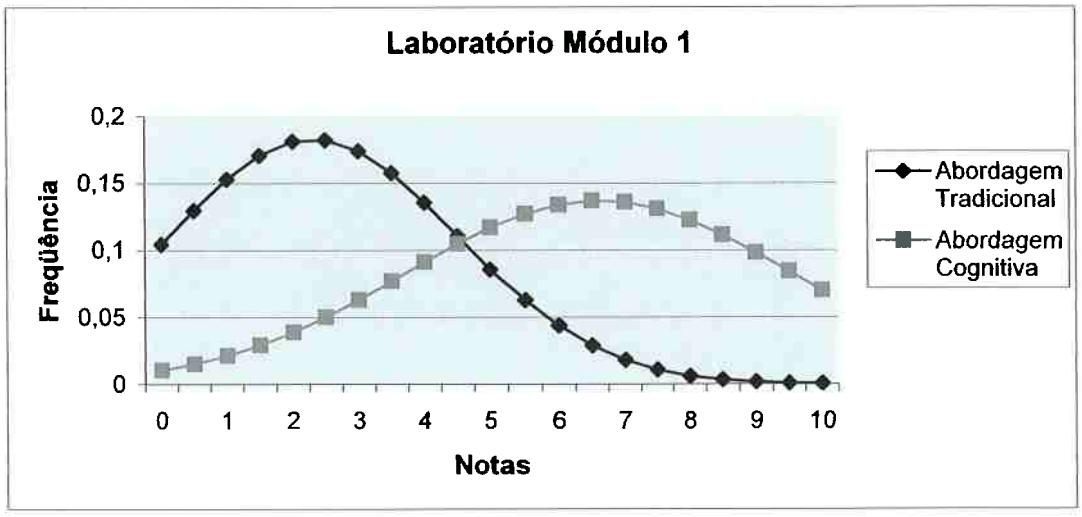


Figura 55 – Distribuição normal das notas de laboratório com o uso da abordagem tradicional e com o uso da abordagem cognitiva

Outro levantamento de dados realizado diz respeito à porcentagem de acertos na prova de reavaliação de teoria pelo grupo de alunos que cumpriu o mínimo de horas exigido (26 alunos), denominado Grupo Participante, e os que não cumpriram (7 alunos), denominado Grupo Não-participante. A prova teve como objetivo verificar três tópicos de conhecimento no aluno: (1) compreensão de programas, (2) construção de programas e (3) identificação de metas e padrões de programação. A tabela 38 apresenta, em porcentagem, a média de acertos para cada um dos tópicos.

Tabela 38 – Porcentagem de acertos na Reavaliação de Teoria Módulo 1 na Abordagem Cognitiva

	Grupo Participante	Grupo Não-participante
Compreensão de programas	58,0%	13,5%
Construção de programas	63,2%	27,4%
Identificação de metas e padrões	61,0%	27,0%

8.4 Discussão

Os gráficos apresentados mostram uma melhora significativa no desempenho do grupo nas provas, após o mini-curso. Pode-se atribuir essa melhora à utilização da nova abordagem uma vez que dados do ano passado (2002) mostram que poucos alunos se recuperam nas reavaliações, como mostra a tabela 36. Essa tabela apresenta médias de duas classes de 2002 do curso de Engenharia Elétrica, as quais mostram-se consideravelmente abaixo dos valores obtidos em 2003. As médias correspondem à prova de avaliação e à prova de reavaliação de teoria do módulo-1 realizadas no ano de 2002. Essas médias foram obtidas a partir das notas dos alunos que obtiveram nota inferior a cinco na prova de avaliação (critério para realização da prova de reavaliação). No ano passado, o critério adotado para o laboratório, não considerava realização de provas de reavaliação. Assim, as notas de laboratório não foram utilizadas nesta comparação. Os anos anteriores a 2002 também não foram considerados, uma vez que a ementa e critérios de avaliação eram diferentes dos atuais. Essa diferença se deve principalmente pela mudança de duração dos cursos noturnos de engenharia, os quais passaram de seis para cinco anos de duração a partir de 2002.

Como as reavaliações após o mini-curso ocorreram em outubro, poderia haver a suposição de que os alunos tiveram mais tempo para estudar, uma vez que as reavaliações, tradicionalmente, ocorrem no final do primeiro semestre. Porém, no início do mini-curso em agosto, o grupo de alunos foi submetido a algumas tarefas de programação que indicaram a ausência de progresso (mas sim retrocesso, por esquecimento) no conhecimento do grupo. Esse fato era esperado, uma vez que o grupo pôde optar por realizar reavaliação no final de junho ou participar do mini-curso e realizar as reavaliações posteriormente. Outro fato a ser levado em consideração é que todos os alunos em reavaliação no Módulo 2 (vetores e matrizes) puderam também optar por realizar reavaliações em outubro. Teoricamente, esses alunos tiveram mais de três meses para estudar, incluindo um mês de férias escolares. Apesar disso, o desempenho desse grupo nas reavaliações do Módulo 2, foi fraco como mostra a tabela 36.

Analisando o gráfico de distribuição normal da figura 55, observamos que houve progresso no desempenho dos alunos no laboratório, porém a dispersão em relação à média é maior do que a apresentada na figura 54 que corresponde às provas de teoria, em sala de aula. Alguns fatores explicam essa diferença. No laboratório, o aluno necessita de habilidades adicionais tais como familiarização com o compilador, prática em tratar erros nas várias fases do desenvolvimento do programa: compilação, *link* e execução. Além disso, na prova de laboratório, o aluno não pôde consultar os planos de programação, exigindo dele, uma carga cognitiva maior. A dispersão mostra que esses fatores adicionais são diferenciados para cada aluno.

Apesar da aplicação realizada ser preliminar, os resultados indicam uma tendência de que a abordagem proposta contribuiu para a melhoria no aprendizado em resolução de problemas de programação. Os dados indicam ainda uma tendência de que o progresso promovido envolveu não somente construção de programas como compreensão de programas. Observando os dados da tabela 38, o grupo participante obteve 58% de acertos em compreensão de programas contra 13,5% do grupo não participante. Em construção de programas o grupo participante obteve 63,2% de acertos contra 27,4% do grupo não participante. Na abordagem tradicional enfatiza-se, tipicamente, a prática de simulações (inspeção de variáveis) para compreensão de programas, enquanto que a abordagem utilizada neste trabalho enfatiza o conhecimento sobre ações primitivas de programação e seu mapeamento com o código.

Outra observação possível a partir dos dados da tabela 38 diz respeito à identificação de metas do problema e sua associação com planos de programação. Os dados indicam que tal conhecimento está diretamente relacionado com a capacidade de construção de programas. Os dois grupos mostram essa relação. O grupo participante obteve 61% de acertos ao identificar metas e seus planos e 63,2% em construção de programas. O grupo não participante obteve 27% de acertos ao identificar metas e seus planos e 27,4% em construção de programas. Esse é um resultado importante que corrobora com resultados de estudos empíricos da Psicologia da Programação, relatados no Capítulo 2.

Além dos dados apresentados, os quais parecem comprovar o progresso no conhecimento em programação do grupo sob teste, em vários momentos durante a realização do mini-curso, alunos emitiram comentários positivos em relação à nova abordagem e ao TutorC. Vários alunos lamentaram o fato de TutorC não estar disponível desde o início do ano e também de não incluir problemas que envolvesse tópicos do módulo 2: vetores e matrizes.

9 CONCLUSÕES E CONSIDERAÇÕES FINAIS

9.1 Conclusões

Nesta tese, foram apresentadas propostas inovadoras para ensino e aprendizado de programação que reuniram esforços de diferentes áreas de pesquisa: Tutores Inteligentes e Planejamento da área de Inteligência Artificial, Psicologia Cognitiva e Padrões Pedagógicos de Programação. O Modelo de Processo de Construção de Programas para Aprendizes proposto, assim como os sistemas de software implementados atacam um dos maiores problemas no aprendizado de programação: a dificuldade na aquisição de habilidades cognitivas para resolução de problemas de programação. A falta de sistemas e metodologias de ensino que apóiem o desenvolvimento dessas habilidades, assim como as queixas ainda atualmente relatadas na literatura, foram o grande incentivo para o desenvolvimento desta pesquisa.

A dificuldade em encontrar conhecimento sobre planejamento em programação algorítmica modelado na literatura e com possibilidade de reuso, incentivou a implementação da ferramenta de autoria denominada Biblioteca para Programação Cognitiva (BibPC). A área de pesquisa em Sistemas Tutores Inteligentes, apresenta, em sua maioria, sistemas com representações internas complexas, porém sem possibilidade de reuso em aplicações externas, tais como a interface de comunicação com o aluno. Apesar da comunidade de Padrões Pedagógicos trabalhar no sentido de formalizar conhecimento especialista em programação, pouca pesquisa foi dedicada à programação procedimental, sendo a grande maioria dos trabalhos, voltados ao paradigma orientado a objetos. Nesse sentido, a utilização de planejamento hierárquico da Inteligência Artificial mostrou-se bem adequado para representar planejamento em programação algorítmica, facilitando a especificação e implementação dos sistemas.

Usuários do modelo proposto podem ser educadores da área que desejam enfatizar ensino e aprendizado de modelos mentais de programas, desenvolvedores de sistemas de aprendizagem de programação algorítmica e finalmente, os próprios aprendizes em programação. Esses três papéis foram desempenhados durante a pesquisa:

- o do professor, uma vez que foi modelado na BibPC, um material didático para planejamento introdutório em programação C e as ferramentas cognitivas aplicadas junto a um grupo de alunos;

- o do desenvolvedor, uma vez que TutorC foi projetado como um sistema tutor que se comunica com o sistema BibPC para alimentar seu Módulo Especialista;

- o do aprendiz, representado pelo grupo de alunos submetidos à nova abordagem e ferramentas.

O processo de construção de programas embutido no modelo, não supõe que este seja uma representação fiel do comportamento de programadores experientes, mas sim um meio para promover o desenvolvimento cognitivo de aprendizes em programação. O modelo e sua automação por meio da BibPC e de TutorC são propostas como ferramentas cognitivas, definida na literatura como aquelas que auxiliam pessoas a executarem atividades cognitivas: atividades mentais que não podem ser observadas diretamente ou que podem ser pouco observáveis, tais como ajudar a pensar, conhecer ou aprender. O programa que o aprendiz define em TutorC pode ser visto como uma descrição do seu processo de pensamento. Ou seja, existe uma proposta de solução do problema no nível mental (o modelo mental do programa) e uma descrição da solução no nível da implementação (o programa). O mapeamento entre as duas representações é um caminho percorrido explicitamente pelo estudante, no TutorC. Isto permite não somente ao estudante, como também ao sistema, a comparação precisa da intenção com a implementação final da solução do problema.

O modelo proposto, implementado em TutorC, configura-se como uma abordagem híbrida para aprendizado uma vez que, de um lado TutorC guia o estudante na construção de seu novo conhecimento fornecendo explicitamente, na interface, componentes para planejamento e implementação de programas. Ou seja, TutorC fornece ao aprendiz uma representação explícita dos componentes cognitivos, os quais deseja-se que o estudante aprenda a criar. Por outro lado, em TutorC o aprendiz é livre para realizar estratégias cognitivas de programação, tais como seleção de planos de programação, ordenação de planos e ações de programação, assim como a instanciação desses componentes.

Esse equilíbrio entre controle e liberdade é desejável quando se aplica o construtivismo em Ciência da Computação. Ben-Ari (2001) observou uma característica peculiar na área da Ciência da Computação que difere das ciências naturais e que foi considerada nesta tese: o aprendiz dificilmente tem alguma estrutura de conhecimento ou paradigma que permita "construir", sobre ela, a estrutura da Ciência da Computação (BEN-ARI,2001). Uma vez que, no construtivismo, o aproveitamento do aprendiz ocorre por meio da construção de conhecimento, a partir de estruturas de conhecimento que este já possui

anteriormente, o conhecimento modelado na BIbPC e disponibilizado em TutorC, preenche parte dessa lacuna de conhecimento, tipicamente existente em aprendizes de programação.

A experiência tem sugerido que o efetivo uso do paradigma do construtivismo no ensino de Ciência da Computação tem se viabilizado somente após o aluno ter conseguido construir uma estrutura de conhecimento básica de uma máquina computacional e alguma percepção da importância dos níveis de abstração (BEN-ARI, 2001). No TutorC, o aprendiz tem o apoio de componentes e estratégias, em diferentes níveis de abstração, e é sobre essa base de conhecimento explícita que o aprendiz constrói seu novo conhecimento sobre programação.

Os resultados obtidos com a avaliação preliminar realizada, fornecem indicativos de que a adoção do modelo proposto em conjunto com a abordagem tradicional em cursos de introdução à programação pode facilitar o aprendizado dos alunos que, tipicamente, apresentam dificuldade no aprendizado da programação, tanto no que diz respeito à construção de solução de problemas como compreensão de programas.

9.2 Contribuições

Esta tese traz contribuições científicas uma vez que propõe uma hipótese – a proposta de um modelo de processo para construção de programas para aprendizes em programação – formaliza e aplica tal modelo num estudo de caso, obtendo resultados preliminares positivos. A formalização do modelo proposto produziu ainda uma linguagem para planejamento em programação algorítmica, denominada Linguagem de Programação Cognitiva (LPC), a qual caracteriza-se por uma contribuição nova na literatura. Por outro lado, a pesquisa traz também contribuições no campo da engenharia: a construção de duas ferramentas de software, BibPC, uma ferramenta de autoria, e TutorC, um sistema tutor inteligente para aprendizado de programação algorítmica. As ferramentas são inovadoras, uma vez que baseiam-se na recente e promissora área de Padrões Pedagógicos para Ciência da Computação. Além disso, este trabalho rendeu a formação de recursos humanos nas atividades de implementação dos sistemas por meio de vários projetos de iniciação científica realizados na Faculdade de Engenharia de Sorocaba e publicados em anais de congressos da área (BARBOSA,2003; DELGADO,2003; ROLINO, 2003; OLIVEIRA,2002).

9.3 Trabalhos Futuros

Vários aspectos desta pesquisa podem ser expandidos ou modificados. Algumas idéias são:

- Automação da tarefa de programação do professor na BibPC. Nessa idéia, o sistema constrói o programa automaticamente a partir de algumas informações dadas pelo professor. São elas: (i) as restrições entre tarefas meta, tal como a Rede de Tarefas Meta proposta na Linguagem de Programação Cognitiva; (ii) as restrições entre tarefas meta e tarefas primitivas, a fim do sistema ser capaz de construir planos deslocalizados e (iii) as instâncias para as meta-variáveis. Com os dois primeiros tipos de informação, BibPC constrói o chamado Esquema do Programa. Com a terceira informação, BibPC instancia o esquema produzindo o programa. Dessa forma o sistema BibPC teria uma representação mais fiel da linguagem LPC.

- Inserção de recursos adicionais no TutorC, tais como:
    - Implementação de uma estratégia alternativa para codificação do esquema do programa, de tal forma que o aluno tenha a liberdade (e responsabilidade) de instanciar as meta-variáveis do esquema do programa. Atualmente a estratégia implementada permite que o aluno associe uma ação primitiva da solução de problema com uma das linhas de código correto disponibilizado pelo tutor.
    - Implementação do chamado Módulo Pedagógico, o qual utiliza informações do Modelo do Estudante para tomar decisões pedagógicas. Um exemplo de tomada de decisão pelo tutor é a seleção da estratégia de codificação adequada ao conhecimento do aluno. Se o aluno comete poucos erros de mapeamento de ações primitivas sobre código correto, está apto a instanciar o esquema do programa, sem apoio do tutor.
    - Incorporação de um sistema de ajuda *on-line* para consulta a conceitos da linguagem de programação.
    - Incorporação de um simulador gráfico de programas, possibilitando ao aluno, a visualização da execução do programa.

- As ferramentas implementadas são específicas para aprendizado da linguagem C. Uma proposta é a de tornar tais ferramentas configuráveis para aprendizado de outras linguagens procedimentais. A linguagem LPC, torna clara a separação dos diversos tipos de conhecimento em programação, o que facilita a implementação desta idéia.

- Uma versão para WEB do TutorC é também um avanço desejável, assim como a exploração de recursos multimídia. Dessa forma, a disponibilidade para estudo do aluno é ampliada, uma vez que poderá acessar o sistema fora dos domínios da instituição.

- Modelagem de planos de programação adicionais na BibPC, a fim de habilitar o uso de TutorC em cursos mais avançados de programação em C.

REFERÊNCIAS

ADAM, A.; LAURENT, J.-P. LAURA: A system to debug student programs. *Journal of Artificial Intelligence*, v.15, 75-122, 1980.

ANDERSON, J. R.; REISER, B. J. The LISP tutor. *BYTE*, v. 10, n. 4, p. 159-175. April, 1985.

ANDERSON, J. R. *Cognitive Psychology and its Implications.* Capítulo 9, 4th Edition Freeman, 1995. Disponível em: http://www.scism.sbu.ac.uk/inmandw/review/cogpsy/author/index.html. Acesso em: Jan.2004.

ANDERSON, J. R.; CORBETT, A. T.; KOEDINGER, K.; PELLETIER, R. Cognitive tutors: Lessons learned. *The Journal of Learning Sciences,* v.4, p. 167-207, 1995. Disponível em: <http://act.psy.cmu.edu/ACT/papers/Lessons_Learned.html>. Acesso em: Abr.2000.

ANDERSON, J.; BOYLE, C.; COBBERT, A.; LEWIS, W. Cognitive modeling and intelligent tutoring. *Artificial Intelligence*, v.42, p.7-50, 1990.

APPLETON, B. Patterns and Software: Essential Concepts and Terminology. 2000. Disponível em <http://www.cmcrossroads.com/bradapp/docs/index.html>. Acesso em: Jan.2004.

ASTRACHAN, O.; WALLINGFORD, E. Loop Patterns. In: FIFTH PATTERN LANGUAGES OF PROGRAMS CONFERENCE, Allerton Park, Illinois, 1998. PloP'98: proceedings, 1998. Disponível em: <http://www.cs.uni.edu/~wallingf/research/>. Acesso em: Agosto 2001.

BARBOSA, M. C. Estudo da Evolução do Aprendizado de Programação no TutorC. In: 11º SIMPÓSIO INTERNACIONAL DE INICIAÇÃO CIENTÍFICA DA USP, São Carlos, 2003.

BECK, J.; STERN M.; HAUGSJAA, E. Applications of AI in Education. *ACM Crossroads Student Magazine.* Eletronic Publication, 1996. Disponível em: <http://www.acm.org/crossroads/xrds3-1/aied.html>.

BEN-ARI, M. Constructivism in Computer Science Education. *Journal of Computers in Mathematics & Science Teaching.* Association for Computing Machinery, Inc. 20(1), p.45-73, 2001. Disponível em: <http://stwww.weizmann.ac.il/g-cs/benari/articles/cons.pdf>. Acesso em: Jan. 2004.

BERGIN, J. *A Pattern for Teaching Patterns.* Disponível em: <http://csis.pace.edu/~bergin/patterns/index.html/>. Acesso em: Jan. 2004.

BERGIN, J. *Patterns for Selection*. 1999. Disponível em <www.csis.pace.edu/~bergin/ patterns>. Acesso em: Jan. 2004.

BERGIN, J. Selection Patterns. In: FOURTH EUROPEAN PATTERN LANGUAGES OF PROGRAMS CONFERENCE, Bad Irsee, Germany, 1999. EuroPLoP'99: proceedings. Disponível em: <http://csis.pace.edu/~bergin/patterns/Patternsv4.html>. Acesso em: Agosto 2001.

BONAR, J.; CUNNINGHAM, R. BRIDGE: An intelligent tutor for thinking about programming. In: Self, J. (Ed.). *Artificial Intelligence and Human Learning*, chapter 24, p. 391-409. London. Chapman and Hall, 1988.

BONAR, J.; CUNNINGHAM, R. BRIDGE: Tutoring the programming process. In: J. Psotka, L. Massey, S. Mutter (Eds.). *Intelligent tutoring systems: Lessons learned.* Hillsdale: Lawrence Erlbaum Associates, 1988.

BONAR, J.; SOLOWAY, E. Uncovering Principles of Novice Programming. *ACM*, p. 10-13, 1983.

BONAR, J.; CUNNINGHAM, R. Intelligent Tutoring with Intermediate Representations. In:

Conference on Intelligent Tutoring Systems, Montreal, 1988. ITS-88: proceedings. Montreal, 1988.

BORLAND. Compilador TurboC. Disponível em: < http://www.borland.com>. Acesso em: Fev. 2003.

BRADY, A. F. Patterns for Constructing and Using Local Variables. 2000. Disponível em: <http://max.cs.kzoo.edu/~abrady/research/>. Acesso em: 2001.

BRADY, A. F. *Patterns for Repetition*. Kalamazoo College, Kalamazoo, MI, 1999. Disponível em: <http://max.cs.kzoo.edu/~abrady/patterns/Repetition.shtml>. Acesso em: Agosto 2001.

BRADY, A. F. *Patterns for Linear Indexed Data Structures*. Kalamazoo College, Kalamazoo, MI, 2001. Disponível em: <http://max.cs.kzoo.edu/~abrady/patterns/LinearIndexDS.shtml>. Acesso em: Março 2002.

BRANDÃO, L. O. *Notas de Aula*. Departamento da Ciência da Computação, Instituto de Matemática, Universidade de São Paulo, USP, 2002.

BRANSFORD, J.; BROWN, A. L.; COCKING, R. R. *How People Learn: Brain, Mind, Experience, and School*. ISBN 0-309-06557-7, p.346, 1999. Livro Eletrônico. Disponível em: http://www.nap.edu/catalog/6160.html. Acesso em: Dez. 1999.

BROOKS, R. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, v.18, p. 543-554, 1983.

BRUSILOVSKY, P. Intelligent learning environments for programming: The case for integration and adaptation. In: WORLD CONFERENCE ON ARTIFICIAL INTELLIGENCE IN EDUCATION, 7th, Washington, DC, August 1995. AI-ED'95: Proceedings, AACE, J. Greer (ed.), 1995. P.16-19. Disponível em: <http://www.contrib.andrew.cmu.edu/~plb/papers/AIED-95.html>. Acesso em: Maio 1999.

BRUSILOVSKY, P.; SCHWARZ, E.; WEBER, G. ELM-ART: An intelligent tutoring system on World Wide Web. In: Frasson, C.; Gauthier, G.; Lesgold, A. (Eds.), *Intelligent Tutoring Systems* (Lecture Notes in Computer Science, Vol. 1086). Berlin: Springer Verlag. 261-269, 1996.

BUDINSKY, F. J.; Finnie, M. A.; Vlissides, J. M.; Yu. P. S. Automatic code generation from design patterns. *IBM Research Journal*, v.35, n.2, 1996. Disponivel em: <http://researchweb.watson.ibm.com/journal/ dj/352/budinsky.html>. Acesso em: Jun. 1999.

CEEINF (Comissão de Especialistas de Ensino de Computação e Informática). *Diretrizes curriculares de cursos da área de computação e informática*, MEC, Secretaria de Educação Superior, 1999. Disponível em: <http://www.mec.gov.br/sesu/ftp/curdiretriz/computacao/co_diretriz.rtf>. Acesso em: Jan. 2001.

COAD, P. Object-oriented patterns. *Communications of the ACM*, v.35 n.9 p152, Sept. 1992.

COX, R.; BRNA, P. Supporting the use of external representations in problem solving: the need for flexible learning environments, 1995, p.239-302. Disponível em: http://www.cbl.leeds.ac.uk/~paul/papers/jaiedERpaper/jaiedpaper.ps.gz. Acesso em: Set. 2000.

CROSBY, M.E.; STELOVSKY, J. How Do We Read Algorithms? A Case Study. *Computer*, v.23, n.1, p. 24-35, 1990.

CURTIS, B.; ISCOE, N.; KRASNER, H. A Field Study of the Software Design Process for Large Systems. *Communications of the ACM,* v. 31, n.11, p. 1268-1287, Nov, 1988.

DABOUS, F. T. A Pattern-Oriented Software Development Approach. University of New South Wales, Sydney, Australia, 2003. Disponivel em: <http://www.sistm.unsw.edu.au/people/FDABOUS/proposal/>. Acesso em: Fev.2004.

DELGADO, D. L. Implementação do Modelo do Estudante no TutorC. SIMPÓSIO INTERNACIONAL DE INICIAÇÃO CIENTÍFICA DA USP, 11º, São Carlos, 2003.

ECKSTEIN, J.; MANNS, M. L.; MARQUARDT, K.; WALLINGFORD; E. Patterns for Experiential Learning. *EuroPLoP* 2001. Irsee, Germany, 2001. Disponível em: <http://www.jeckstein.com/>. Acesso em: 18 junho 2001.

ECKSTEIN, J.; VOELTER, M. Patterns and Pedagogy - a winning team: Learning to teach, learning to learn. In: *IBM developerWorks*, January 2001. Disponível em: <http://www.jeckstein.com/>. Acesso em: 10 agosto 2001.

ECLIPSE Project. Disponível em: <http://www.instantiations.com/codepro/ws/docs/default.htm>. Acesso em: FEV2004.

EROLA, A.; MALMI, L. KELVIN - A System for Analysing and Teaching C Programming Style. Helsinki University of Technology, Finland. In: *Proceedings of CLCE'94, Complex Learning in Computer Environment*, University of Joensuu, Finland, pp. 112-117, 1994. Disponível em <http://www.cs.hut.fi/~lma/papers/>. Acesso em: 2000.

ELLIS, G. P. ; LUND, G. R. G2- A design language to help novice C programmers. COMPUTERS IN TEACHING INITIATIVE ANNUAL CONFERENCE, CTI, 1994. Disponível em <http://www.ulst.ac.uk/misc/cticomp/gpellis.html>. Acesso em: 2000.

EROL, K.; NAU, D.; HENDLER, J. HTN planning: Complexity and Expressivity. In: TWELFTH NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE. Anais. Seattle, WA: AAAI Press, 1994, p. 1123-1128.

FISCHER, M. C. ESTUDO DE REQUISITOS PARA UM SOFTWARE EDUCATIVO DE APOIO AO ENSINO DA INTRODUÇÃO À COMPUTAÇÃO. 172p. Dissertação, IME-USP, 2001.

FIX, V. ; WIEDENBECK, S. Using Cognition of Programming Literature in the Design of a Tool for Learning a Second Programming Language. In: INTERNATIONAL CONFERENCE OF INTELLIGENT TUTORING SYSTEMS, 3th, ITS'96, Montreal, Canada, June, 1996.

GEORGE, C. E. Evaluating a Pedagogic Innovation: Execuction Models and Program Construction Ability. In: Proceedings of 8th ANNUAL CONFERENCE ON THE TEACHING OF COMPUTING, 2000. Disponível em: <http://www.ics.ltsn.ac.uk/pub/conf2000/Papers/George.pdf>. Acesso em: 2001.

GRAY, W. D.,; ALTMANN, E. M. Cognitive modeling and human-computer interaction. *International Encyclopedia of Ergonomics and Human Factors*, New York, v.1, p. 387-391, 2001. New York: W. Karwowski (Ed.). Disponível em: <http://hfac.gmu.edu/%7Egray/pubs/papers/Encycl.htm>. Acesso em: 16 janeiro 2002.

HAHN, S. H. Automatic problem description from a model program for a knowledge-based programming tutor. In: Proceedings 1st JOINT CONFERENCE ON INTELLIGENT TECHNOLOGY, p. 82-90, Seoul, Korea, 1995.

HAHN, S. H.; SONG, J. S.; TAK, K. Y.; KIM, J. H. An intelligent tutor system for introductory C language course. *Computers & Education*, v. 28, n.2, Feb, p. 93-102, 1997.

HANMER, R. Patterns of Efficient Prolog Programs. In: Proceedings of Third PATTERN LANGUAGES OF PROGRAMS CONFERENCE, Allerton Park, Illinois, 1996.

IBM. Rational Rose. Disponivel em: <http://www-306.ibm.com/software/awdtools/developer/technical/>. Acesso em: Fev.2004.

IBM. Websphere Studio. Disponível em: <http://www-3.ibm.com/software/info1/websphere>. Acessado em: 02/07/2003.

INSTANTIATIONS Inc. CodePro Studio. Disponível em: <http://www.instantiations.com/codepro/ws/docs/default.htm>. Acesso em: Fev.2004.

JOHNSON, W. L. Understanding and Debugging Novice Programs. *Artificial Intelligence*, v.42, p. 51-97, 1990.

JOHNSON, W. L. ; SOLOWAY, E. Proust: An automatic debugger for pascal programs. *Byte*, v.10, n.4, p. 179-190, 1985.

KROTH, E.; PFANFESELLER, M., Uma ferramenta de apoio ao desenvolvimento de software baseado em componentes. XV SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, Sessão de Ferramentas, Rio de Janeiro-RJ, outubro, 2001. Disponível em: <http://www.inf.ufrgs.br/~kroth/papers.htm>. Acesso em: Fev. 2004.

KUEHNE, T. A Functional Pattern System for Object-Oriented Design, Verlag Dr. Kovac, Hamburg, Germany, 1999.

LALANDA, P. Two complementary patterns to build multi-expert systems. In: Proceedings of the *PloP'97 Conference*, 1997.

LAMPERT, R.; LITTMAN, D.; PINTO, J.; SOLOWAY, E. ; LETOVSKY, S. Designing Documentation to Compensate for Delocalized Plans. *Communications of the ACM,* v.31, n.11, Nov, 1988.

LEMOS, M. A.; BARROS, L.N.; LOPES, R. D. Modeling plans and goals of Programming in an Intelligent Tutoring System. In: WORKSHOP ON KNOWLEDGE REPRESENTATION AND AUTOMATED REASONING FOR E-LEARNING SYSTEMS (KRR-5) held on 18th International Joint Conference on Artificial Intelligence (IJCAI). Acapulco, México, August 9-15, 2003a.

LEMOS, M.A.; BARROS, L.N. A Didatic Interface in a Programming Tutor. In: WORKSHOP ON INNOVATIONS IN TEACHING PROGRAMMING held on 11th International Conference on Artificial Intelligence in Education (AIED2003), Sidney, Australia, July 20-24, 2003.

LEMOS, M.A.; BARROS, L.N.; LOPES, R.D. Uma Biblioteca Cognitiva para o Aprendizado de Programação. In: XI WORKSHOP DE EDUCAÇÃO EM COMPUTAÇÃO (WEI). Simpósio Brasileiro de Computação (SBC), Campinas, 02-08 Agosto, 2003b.

LINN, M. C.; CLANCEY, M. J. The Case for Case Studies of Programming Problems. *Communications of the ACM*, v.35, n.3, p. 121-132, 1992.

MARINHO, F.; SANTOS, M.; PINTO, R. N.; ANDRADE, R. Uma Proposta de um Repositório de Padrões de Software Integrado ao RUP. In: Proceedings of *The Third Latin American Conference on Pattern Languages of Programming.* Porto de Galinhas, Pernambuco, Brasil. August 12-15, 2003. Disponível em: <http://www.cin.ufpe.br/~sugarloafplop/acceptedPapers.htm>. Acesso em: Fev. 2004.

MESZAROS, G. Archi-Patterns: A Process Pattern Language for Defining Architectures. In Proceedings of the *PloP'97 Conference*, 1997.

MESZAROS, G.; DOBLE, J. Pattern Language for Pattern Writing. Disponível em <http://hillside.net/patterns/writing/patternwritingpaper.htm>. Acesso em: 2000.

MICROTOOL. Objectif's pattern manager. Disponivel em: <http://www.microtool.de/objectiF/en/sp_pattern.htm>. Acesso em: Jan. 2004.

MINSKY, M. (1985), The Society of Mind, Simon & Shuster, NY.

MODELMAKER TOOLS. ModelMaker. Disponível em: <http://www.modelmaker.demon.nl/mm_design_patterns.htm>. Acesso em: Jul.2003.

MOSTRÖM, J. E. ; CARR, D. A. Programming Paradigms and Program Comprehension by Novices. PSYCHOLOGY OF PROGRAMMING INTEREST GROUP. PPIG'10, 1998. Disponível em:<http://www.mostrom.pp.se/folk/jem/art01.pdf>. Acesso em: Nov. 1999.

NORMAN, D. Some Observations on Mental Models. MENTAL MODELS, Erlbaum, p. 7-14, 1983. D. Gentner; A. L. Stevens (Eds), LEA. Disponível em: <http://www.cogs.susx.ac.uk/users/christ/crs/atcs/norman.html>. Acesso em: Jan. 2004.

O'BRIEN, M. P.; SHAFT, T.M.; BUCKLEY, J. An Open-Source Analysis Schema for Identifying Software Comprehension Processes. In: WORKSHOP OF THE PSYCHOLOGY OF PROGRAMMING

INTEREST GROUP, 13th, Bournemouth, UK, April, 2001. Anais. Bournemouth: G. Kadoda (Ed), 2001, p. 129-146. Disponível em: <http://www2.umassd.edu/SWComprehension/compdocs/ppig13/obrien.pdf>. Acesso em: Julho 2002.

OLIVEIRA, R. T. D.; LEMOS, M. A. Construção de um Modelo de Domínio num Tutor Inteligente para Ensino da Linguagem C. 10º SIMPÓSIO INTERNACIONAL DE INICIAÇÃO CIENTÍFICA DA USP, São Carlos, 2002.

PANE, J. F. ; MYERS, B. A. Usability Issues in the Design of Novice Programming Systems. Pittsburgh: Human-Computer Interaction Institute, Carnegie Mellon University, 1996. (*Technical Report* Carnegie Mellon University, CMU-HCII-96-101). Disponível em <http://www-2.cs.cmu.edu/~pane/publications.html>. Acesso em: Janeiro 2001.

PANE, J. F. ; MYERS, B. A. The Influence of the Psychology of Programming on a Language Design: Project Status Report. In: ANNUAL MEETING OF THE PSYCHOLOGY OF PROGRAMMERS INTEREST GROUP, 12th, Corigliano Calabro, April 10-13. Anais. Italy: A. F. Blackwell and E. Bilotta, Eds., Edizioni Memória, 2000, p. 193-205. Disponível em <http://www-2.cs.cmu.edu/~pane/publications.html>. Acesso em: Janeiro 2001.

PEGG, K. Prior Programming Experience and the Influence It has on Students' Performance in an Introductory Programming. *Thesis*. University of Melbourne, Australia, 107p., 2003. Disponível em: http://uob-community.ballarat.edu.au/staff/~kpegg/kyliePeggHonours.pdf. Acesso: Jan. 2004.

PIAGET, J. A teoria de Piaget. In P. Mussen and S. Pfromm Netto (Eds.). *Manual de Psicologia da Criança* (translated to portuguese from Carmichael's Manual of Child Psychology), S.P.:EDUSP, pp. 72-117, 1977.

PEP. Padrões Elementares de Programação. *Homepage*. Disponível em: <http://www.cs.uni.edu/~wallingf/patterns/elementary/>. Acesso em: Jun. 2001.

PPIG: Psychology of Programming Interest Group, 1987. *Homepage*. Apresenta recursos e atividades desenvolvidas. Disponível em: <http://www.ppig.org/>. Acesso em: 2000.

PP. Programming Patterns. *Homepage*. Disponível em: <http://www.hillside.net>. Acesso em: Jun. 2001.

PRAGSOFT Corporation. UMLStudio. 2003. Disponível em: http://www.pragsoft.com. Acesso em: Fev. 2004.

RAMALINGAM, V. ; WIEDENBECK, S. An empirical study of novice program comprehension in the imperative and object-oriented styles. In: SEVENTH WORKSHOP ON EMPIRICAL STUDIES OF PROGRAMMERS, October, 1997. Anais. Disponível em: <http://www.acm.org/pubs/contents/proceedings/chi/266399/>. Acesso em: Fev. 2002.

REDISH, K. A. ; SMYTH, W. F. Program Style Analysis: A Natural By-product of Program Compilation. *Communications of the ACM*, v. 29, n.2, 1986.

RIEHLE, D.; ZÜLLIGHOVEN, H. Understanding and Using Patterns in Software Development, 1996. Disponível em: <http://www.citeseer.nj.nec.com/riehle96undertanding.html>. Acesso em: 2000.

ROCHA, H. V. Representações computacionais auxiliares ao entendimento de conceitos de programação. Capítulo do livro *Computadores e Conhecimento: Repensando a Educação*. Gráfica Cultural Unicamp. Separata 16, 1995. Disponível em: <http://www.nied.unicamp.br/publicacoes/>. Acesso em: Jan. 2004.

ROLINO, A. Explicando Erros para o Estudante no TutorC. 11º SIMPÓSIO INTERNACIONAL DE INICIAÇÃO CIENTÍFICA DA USP, São Carlos, 2003.

RUGABER, S. The use of domain knowledge in program understanding. *Software Engineering*, 9, p. 143-192, 2000. Disponível em <http://citeseer.nj.nec.com/rugaber00use.html>. Acesso em: Jan. 2004.

RYAN, C. ; AL-QAIMARI, G. A Cognitive Perspective on Teaching Object-Oriented Analysis and Design. In: COMPUTER SCIENCE POSTGRADUATE CONFERENCE, RMIT University, Melbourne, Australia, 1996. (Technical Report RMIT University, TR-96-36). Disponível em: <http://citeseer.nj.nec.com/13036.html>. Acesso em: Jun. 1999.

SANDU, D. Collection Patterns. *PLoP 2001 Conference*, 2001.

SELF, J. Computational Mathetics: Towards a Science of Learning Systems Design. Computer Based Learning Unit, University of Leeds, 1995. Disponível em: <http://www.cbl.leeds.ac.uk/~jas/cm.html>. Acesso em: Abr. 1999.

SHUTE, V. J.; PSOTKA, J. Intelligent Tutoring Systems: Past, Present and Future. *Handbook of Research on Educational Communications and Technology*, Scholastic Publications, D. Jonassen (Ed.), 1994.

SOLOWAY, E. ; ADELSON, B. The Role of Domain Experience in Software Design. *IEEE Transactions on Software Engineering.* V. 11, n.11, p. 1351-1360, 1985.

SOLOWAY, E. ; EHRLICH, K. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, IEEE Computer Society, v. SE-10, n. 5, p. 595-609, September, 1984.

SOLOWAY, E. ; LETOVSKY, S. Delocalized Plans and Program Comprehension. *IEEE Software.* V.3, n.3, May, 1986.

STOREY, M.A.D.; FRACCHIA, F. D.; MULLER, H. A. Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration. *Journal of Systems and Software*, 1999. Disponível em: <http://citeseer.nj.nec.com/270012.html>. Acesso em: 2002.

SUN MICROSYSTEMS. Core J2EE Pattern Catalog. Disponível em: <http://java.sun.com/blueprints/ corej2eepatterns/Patterns/index.html>. Acesso em: Fev. 2004.

TIEMENS, T. Cognitive Models of Program Comprehension. Software Engineering Research Center, 1989. Disponível em <http://citeseer.nj.nec.com/98434.html>. Acesso em: Fevereiro 2002.

VALENTE, J. A. Diferentes usos do Computador na Educação. Capítulo do livro *Computadores e Conhecimento: Repensando a Educação*. Gráfica Cultural Unicamp. Separata 1, 1995. Disponível em: <http://www.nied.unicamp.br/publicacoes/>. Acesso em: Jan. 2004.

VON MAYRHAUSER, A. ; VANS, A. M. Program Understanding: A Survey. *Technical Report*, CS-94-120. Colorado: Colorado State University, August 23, 1994

VON MAYRHAUSER, A. ; VANS, A. Program comprehension during software maintenance and evolution. *IEEE Computer*, p. 44–55, August, 1995.

WALLINGFORD, E. Elementary Patterns and their Role in Instruction. In: OOPSLA'98 EDUCATORS SYMPOSIUM NOTES, 1998. Disponível em: <http://www.cs.uni.edu/~wallingf/patterns/elementary/chiliplop98/summary.html>. Acesso em: 20 julho 2001.

WALLINGFORD, E. Functional Programming Patterns and Their Role in Instruction. Functional and Declarative Programming in Education Workshop held on International Conference on Functional Programming, Pittsburgh, 2002. Disponível em: <http://www.cs.uni.edu/~wallingf/patterns/functional/>. Acesso em: 09 dezembro 2002.

WALLINGFORD, E. Roundabout: A Pattern Language for Recursive Programming. In: Proceedings Fourth PATTERN LANGUAGES OF PROGRAMS CONFERENCE, Allerton Park, Illinois, 1997. Disponível em: <http://www.cs.uni.edu/~wallingf/research/>. Acesso em: 2000.

WALLINGFORD, E. Using Patterns in the CS Curriculum. 5th Northeastern CONFERENCE OF THE CONSORTIUM FOR COMPUTING IN SMALL COLLEGES, NSF/SIGCSE-Sponsored Tutorial, 2000.

WEBER, G.; MÖLLENBERG, A. ELM-PE: A knowledge-based programming environment for learning

LISP. In T. Ottmann & I. Tomek (Eds.), *Proceedings of ED-MEDIA '94*, p. 557-562. Charlottesville, VA: AACE, 1994. Disponível em: <http://www.psychologie.uni-trier.de:8000/projects/ELM/elmpe.html>. Acesso em: Out.1999.

WEBER, G. ; SPECHT, M. User modeling and adaptive navigation support in WWW-based tutoring systems. In: Proceedings of *User Modeling '97,* p. 289-300, 1997. Disponível em: <http://www.psychologie.uni-trier.de:8000/projects/ELM/elmart.html>. Acesso em: 2000.

WEBER, G.; BRUSILOVSKY, M. S.; STEINLE, F. ELM-PE: An Intelligent Learning Environment for Programming, 1996. Disponível em: <http://www.psychologie.uni-trier.de:8000/projects/ELM/elm.html>. Acesso em: 2000.

WELTY, C. An Integrated Representation for Software Development and Discovery. 1995. *Thesis* (PhD.), Computer Science Dept., Rensselaer Polytechnic Institute, 1995. Disponível em <http://www.cs.vassar.edu/faculty/welty/papers/phd/>. Acesso em: 2001.

WENGER, E. *Artificial Intelligence and Tutoring Systems*. Los Altos, CA: Morgan Kaufmann, 1987.

GLOSSÁRIO

Ação Primitiva de Programação - ação mínima de programação que pode ser implementada por uma linha de código.

Algoritmo – conjunto finito de ações ordenadas.

Atividade Cognitiva - Atividade mental que não pode ser observada diretamente ou que pode ser pouco observável. (Ciência Cognitiva)

Base de Conhecimento Interna - Corresponde ao conhecimento prévio do aprendiz ou programador experiente, constituída pelo conhecimento de domínio de problemas e conhecimento sobre programação. (Psicologia da Programação)

Beacon – *Chunk* de programação. (Psicologia da Programação)

Biblioteca Cognitiva – Conjunto de componentes para planejamento e implementação de programas.

***Chunk* de Programação** – estrutura cognitiva criada por programadores experientes, na forma de fragmentos de código, os quais possuem uma funcionalidade. São úteis na compreensão de programas para sugerir o que o código faz ou servem como blocos de construção na elaboração de um programa. (Psicologia da Programação)

Ciência Cognitiva – Ciência que estuda a natureza, estrutura e processos de organização do pensamento. Em particular, estuda a criação de modelos computacionais para a modelagem do comportamento inteligente. (Ciência Cognitiva)

Componente Cognitivo de Programação - Elemento ou estratégia proposta para uso pelo aprendiz ao realizar atividades cognitivas de programação. Exemplos: metas, planos e ações de programação.

Componente de Planejamento em Programação – Componente Cognitivo de Programação

Esquema do Plano – Código, generalizado por meta-variáveis, que compõe (implementa) um plano de programação.

Esquema do Programa – Solução para um problema de programação descrita no nível acima da implementação, a qual constitui-se de esquemas de planos ordenados.

Esquema Primitivo - Linha de código, generalizada por meta-variáveis, a qual implementa uma ação primitiva.

Ferramenta Cognitiva – Software ou método que auxilia pessoas a executarem atividades cognitivas tais como ajudar a pensar, conhecer ou aprender. (Inteligência Artificial)

Idioma – Padrão elementar de programação. (Padrões Pedagógicos)

Meta do Problema – Alvo a ser alcançado na resolução de um problema de programação.

Meta-Variável – Variável definida no nível acima da implementação, a qual fornece a semântica do elemento que a instancia no plano de programação.

Modelo do Programa – Proposta de representação do modelo mental de um

programa.

Modelo Cognitivo de Compreensão de Programa - Processo que conduz à criação do Modelo Mental do Programa durante a atividade de compreensão de programas. (Psicologia da Programação)

Modelo Cognitivo de Construção de Programa - Processo que conduz à criação do Modelo Mental do Programa durante a atividade de construção de programas. (Psicologia da Programação)

Modelo Cognitivo de Programação - Processo que conduz à criação do Modelo Mental do Programa. (Psicologia da Programação)

Modelo de Compreensão – Abreviação do termo "Modelo Cognitivo de Compreensão de Programa".

Modelo de Construção - Abreviação do termo "Modelo Cognitivo de Construção de Programa".

Modelo de Problema - Descrição de problema na forma de metas do problema a serem alcançadas.

Modelo Mental do Programa - Representação mental do programa, a qual o programador elabora durante o processo de compreensão ou de construção do programa (Psicologia da Programação).

Programação Algorítmica – Abordagem de programação que promove a construção de programas na forma de seqüência de ações ou comandos.

Padrão de Programação - Regra que descreve conhecimento especialista em programação, no nível da implementação. Consiste de três partes: (1) o contexto de aplicação, (2) um problema, (3) uma solução. (Padrões Pedagógicos)

Padrão Elementar – Padrão Elementar de Programação. (Padrões Pedagógicos)

Padrão Elementar de Programação – Padrão de Programação mais simples, destinado ao uso por aprendizes. (Padrões Pedagógicos)

Padrão Pedagógico - Meio para armazenar conhecimento especialista da prática de ensinar e aprender em qualquer domínio. (Padrões Pedagógicos)

Plano de Programação[1] – Terminologia usada por alguns pesquisadores e/ou áreas de pesquisa como sinônimo de *chunk* de programação. (Psicologia da Programação)

Plano de Programação[2] - Artefatos, que representam *chunks* de programação, construídos por projetistas para serem embutidos em sistemas de software. (Sistemas Tutores Inteligentes)

Plano Deslocalizado - Plano de programação fragmentado e espalhado no código fonte. (Psicologia da Programação)

Plano Natural - Solução correta para um dado problema, porém não necessariamente executável no computador. (Psicologia da Programação)

Programação algorítmica – atividade relacionada com a parte procedimental do software em desenvolvimento sob paradigmas de programação tais como estruturado e orientado a objetos.

Situação do Plano – Parte de um plano de programação que descreve o contexto no qual o plano se aplica.

APÊNDICE A – APOSTILA: PROGRAMAÇÃO COGNITIVA

Este apêndice apresenta a apostila utilizada no mini-curso mencionado no Capítulo 8, a qual apresenta o conjunto de planos de programação modelados na BibPC e disponíveis no TutorC para manipulação pelos alunos.

FACENS

FACULDADE DE ENGENHARIA DE SOROCABA

Disciplina: Introdução à Programação e Cálculo Numérico

Profa. Marilza Antunes de Lemos

Resolução de Problemas usando Programação Cognitiva

Sorocaba - SP
Agosto - 2003

1. O que é Resolução de Problemas de Programação no nível cognitivo?

Resolver problemas de programação no nível cognitivo é uma abordagem diferente da abordagem tradicional no ensino da programação. Nessa nova abordagem, a solução para um problema numa linguagem de programação, tal como C, é constituída de alguns componentes, além do código do programa:

- **Metas do problema**:
  - aquilo que desejamos alcançar com a construção do programa;
- **Planos de programação**:
  - fragmentos de código que realizam as metas do problema;
- **Ações primitivas**:
  - sub-partes de um plano de programação;
- **Programa**:
  - o código do programa, propriamente dito.

2. O que são Planos de Programação ?

Um plano é constituído de:

- **Nome**
  - Identificação do plano;
- **Esquema**
  - Consiste de um bloco de código generalizado através de **meta-variáveis**, as quais devem ser substituídas por dados específicos do problema durante a criação do programa;
- **Situação**
  - Texto que explica em que situação é interessante aplicar o plano.
- **Ações primitivas**
  - É o conjunto de ações mínimas de programação que formam o plano.

Um exemplo de plano de programação em C é o plano *entrada-por-teclado*. A tabela abaixo mostra a descrição de aplicabilidade do plano (situação) e o esquema do plano (bloco de código genérico).

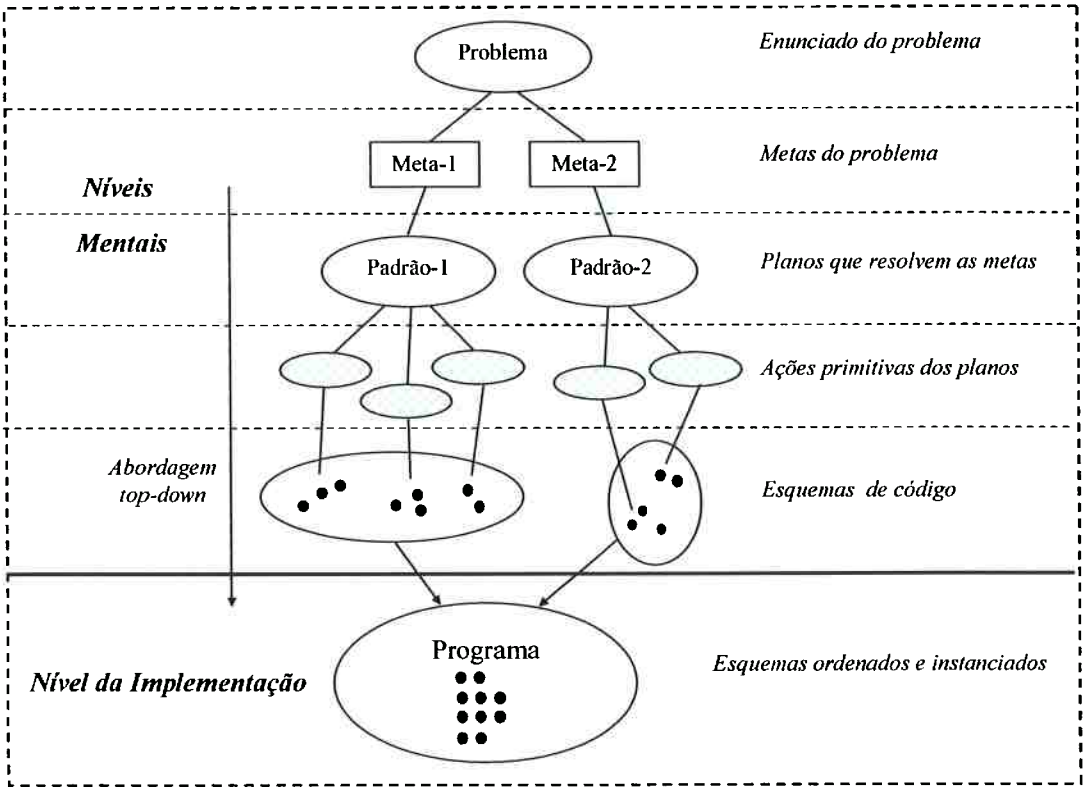
Situação	Esquema

Você está numa situação onde o usuário (pessoa que utiliza o programa) precisa interagir com o programa, informando dados pelo teclado. Esses dados podem ser numéricos ou caracteres.	`printf($formato-de-saída);`
	
	`scanf($formato-de-entrada; $ENDEREÇOS);`

Quando usamos um **plano de programação** na resolução de um problema, as palavras precedidas do caracter $ devem ser substituídas pelos dados específicos do problema em questão. Por isso tais palavras são chamadas de **meta-variáveis**. Chamamos esta operação de **instanciação**. A tabela abaixo mostra a decomposição do plano *entrada-por-teclado* em ações primitivas. Através das ações primitivas é possível entender como o esquema é formado.

Ações Primitivas	Esquema
• emite mensagem para o usuário	`printf($formato-de-saída);`
• obtém dados do usuário	`scanf($formato-de-entrada; $ENDEREÇOS);`

A seção 4 contém um conjunto de planos de programação que podem ser usados para resolver uma série de problemas básicos. A figura abaixo mostra o processo de construção de programas no nível cognitivo.



A estratégia de construção do programa é chamada estratégia *top-down* pois inicia-se a partir das metas do problema em direção ao código, ou seja, de um nível de abstração mais alto em direção ao mais baixo. O programa pronto está situado no nível da implementação, ou seja no nível da linguagem de programação. Para construir um programa nesta abordagem, é necessário passar por vários níveis de abstração até chegar no nível mais baixo: o da implementação.

3. Como construir um programa usando a Abordagem Cognitiva?

Seja a seguinte descrição de problema:

"Faça um programa para ler dois números e mostrar o resultado do quadrado da diferença do primeiro pelo segundo. Use a fórmula: $(a-b)^2 = a^2 - 2ab + b^2$."

As metas do problema são:

Meta-1: Quadrado da diferença
Meta-2: Resultado na tela
Meta-3: Variáveis simples alocadas na memória
Meta-4: Dois números obtidos do usuário

Resolução:

O primeiro passo consiste em escolher um plano de programação para cada meta do conjunto de metas do problema. Lembre-se: **um plano resolve uma meta**. Para fazer uma escolha correta, é necessário consultar a **situação** do plano, pois ela descreve em que situação o plano é importante. Na seção 4, podemos selecionar os seguintes planos para resolver as metas do problema:

Meta-1 → P1: Processamento
Meta-2 → P2: Saída de dados
Meta-3 → P3: Declaração de variáveis simples
Meta-4 → P4: Entrada por teclado

O segundo passo é obter as ações primitivas dos planos selecionados:

Plano	Ações	Esquema
P1	Constrói expressão para cálculo	$variavel=$expressao;
P2	Mostra o resultado na tela	printf($formato-de-saída, $variaveis);
P3	Declara variáveis simples	$tipo $variaveis;
P4	Emite mensagem para o usuário Obtém dados do usuário	printf($formato-de-saída); scanf($formato-de-entrada, $enderecos);

O próximo passo consiste em ordenar logicamente, as ações primitivas e os esquemas dos planos para formar o **esquema do programa**. Para obter a ordenação correta, lembre-se de recordar as metas do problema:

Planos	Esquema do programa
P3	$tipo $variaveis;
P4	printf($formato-de-saída);
P4	scanf($formato-de-entrada, $enderecos);
P1	$variavel=$expressao;
P2	printf($formato-de-saída, $variaveis);

Agora vamos instanciar o **esquema do programa** usando os dados do domínio do problema e, assim, obter o programa em C. Instanciar significa substituir as **meta-variáveis** por dados específicos do domínio do problema.

Esquema do programa	Programa
$tipo $variaveis;	int x, y, resultado;
printf($formato-de-saída);	printf("\nDigite dois números inteiros (x,y): ");
scanf($formato-de-entrada, $enderecos);	scanf("%i, %i", &x, &y);
$variavel=$expressao;	resultado= x*x - 2*x*y + y*y;
printf($formato-de-saída, $variaveis);	printf("\n\n O quadrado da diferença é %i", resultado);

Pronto! Agora é só criar uma função main e compilar o programa abaixo no TurboC:

```
main()
{
int x, y, resultado;
printf("\n\n Digite dois numeros inteiros (x,y):   ");
scanf("%i, %i", &x, &y);
resultado= x*x - 2*x*y + y*y;
printf("\n\n O quadrado da diferenca e %i", resultado );
}
```

4. Planos para Programação Algorítmica

4.1 Aplicabilidade dos Planos

Tabela 1 - Planos Básicos

NOME DO PLANO	ESQUEMA DO PLANO	SITUAÇÃO
Declaração de variáveis simples	$tipo $variaveis;	Você precisa alocar espaço na memória para armazenar dados do programa. Esses dados podem ser dados de entrada, dados intermediários e dados de saída. Você deseja armazenar somente um dado (número ou caracter) de cada vez.
Inicialização	$variavel=$valor;	Existe a necessidade de atribuir um valor inicial para uma variável. Alguns casos comuns são: (1) quando essa variável é usada para contar elementos; (2) fazer soma cumulativa; (3) forçar a entrada num laço.
Entrada por teclado	printf($formato-de-saída); scanf($formato-de-entrada; $enderecos);	Você esta numa situação onde o usuário (pessoa que utiliza o programa) precisa interagir com o programa, informando dados. Esses dados podem ser numéricos ou caracteres.
Entrada caracter	printf($formato-de-saída); $variable = getch( );	Você esta numa situação onde o usuário (pessoa que utiliza o programa) precisa interagir com o programa, fornecendo um único caracter.
Processamento	$variavel=$expressao;	Existe a necessidade de realizar um processamento matemático com os dados do programa. Você deseja implementar uma expressão e armazenar seu resultado numa variável.
Saída de dados	printf($formato-de-saída, $variaveis);	Você está numa situação onde o usuário deseja visualizar, na tela, o resultado de um processamento ou dados em geral, os quais estão armazenados em variáveis. Uma mensagem de texto pode acompanhar os dados, na tela.
Mensagem	printf($formato-de-saída);	Existe a necessidade de enviar uma mensagem de texto, na tela, para o usuário.

Tabela 2 – Planos de Seleção

NOME DO PLANO	ESQUEMA	SITUAÇÃO

se-ou-não	if ($condição)
	{$ação;
	}
		Alguma ação pode ou não ser apropriada, dependendo de uma condição a ser testada

ação alternativa	if ($condição)
	{ $ação;
	}
	else { $ação;
	}
		Uma de duas ações é apropriada dependendo de uma condição. Quando a condição é verdadeira deseja-se executar uma condição, quando falsa deseja-se executar uma ação diferente

expressão condicional	if($condição)
	{
	temp=$expressao1;
	}
	else {
	temp=$expressão2;
	}
	$variavel=$expressão3;
		Existe a necessidade de calcular um valor para usá-lo numa expressão (expressão3) ou atribuição. O valor depende de uma única condição.

Escolha seqüencial tipo-1	if ($condição1)
	{ $ação;
	}
	else {
	if ($condição-2)
	{ $ação;
	}
	else { $ação;
	}
	}
		Existe a necessidade de escolher uma entre várias ações possíveis, mas cada ação possível depende de uma condição testada separadamente. A escolha é seqüencial, isto é, se a ação anterior não pôde ser escolhida, então verifica a próxima. Pode ser necessário verificar duas condições.

Escolha seqüencial tipo-2	if ($condição-1)
	{ $ação;
	}
	else {
	if ($condição-2)
	{ $ação;
	}
	else {
	if ($condição-3)
	{ $ação;
	}
	else { $acao;
	}
	}
	}
		Existe a necessidade de escolher uma entre várias ações possíveis, mas cada ação possível depende de uma condição testada separadamente. A escolha é seqüencial, isto é, se a ação anterior não pôde ser escolhida, então verifica a próxima. Pode ser necessário verificar três condições.

continua

continuação

NOME DO PLANO	ESQUEMA	SITUAÇÃO

condições sequenciais tipo-1	if ($condição-1)
	{
	if ($condição-2)
	{ $ação;
	}
	}
		Existe a necessidade de verificar se uma ação pode ser executada. Isso depende de varias (duas) condições.

condições sequenciais tipo-2	if ($condição-1)
	{
	if ($condição-2)
	{
	if ($condição-3)
	{ $ação;
	}
	}
	}
		Existe a necessidade de verificar se uma ação pode ser executada. Isso depende de varias (três) condições.

escolha não relacionada tipo-1	if ($condição-1)
	{ $ação;
	}
	if ($condição-2)
	{ $ação;
	}
		Existem várias ações e várias (duas) condições. Pode ser desejável executar várias das ações se as condições associadas forem verdadeiras. Cada ação tem uma condição que determina se ela deve ser executada.

escolha não relacionada tipo-2	if ($condição-1)
	{ $ação;
	}
	if ($condição-2)
	{ $ação;
	}
	if ($condição-3)
	{ $ação;
	}
		Existem várias ações e várias condições (três). Pode ser desejável executar várias das ações se as condições associadas forem verdadeiras. Cada ação tem uma condição que determina se ela deve ser executada.

escolha independente	if ($condicao-1)
	{
	if ($condicao-2)
	{ $acao-x;
	}
	else { $acao-y;
	}
	}
	else {
	if ($condicao3)
	{ $acao-x;
	}
	else { $acao-y;
	}
	}
		Uma, entre duas ações, deve ser escolhida. Mas a escolha de cada uma das ações dependem ainda de vários fatores. Os fatores são independentes.

conclusão

Tabela 3 – Planos de Repetição Geral

SITUAÇÃO	ESQUEMA
Plano: Verifica sentinela	

Você deseja receber números do usuário, um de cada vez, e validá-los segundo uma certa condição de entrada. Enquanto a condição for verdadeira, o próximo número deve ser recebido. A condição verifica, por exemplo, se o número recebido do usuário está dentro de uma faixa ou se ele é maior ou menor ou diferente que um certo valor.	printf("Entre com o numero: ");
	scanf($formato-de-entrada, &numero);
	while($condição)
	{
	printf("Entre com o numero:");
	scanf($formato-de-entrada, &numero);
	}

Plano: Processamento seqüencial com condição de entrada	

Você deseja repetir um conjunto de ações se uma certa condição for verdadeira (repetição com condição de entrada de Brandão (2003). Se a condição for falsa, você não quer executar nenhuma vez o conjunto de ações. O conjunto de ações refere-se a receber e processar uma seqüência de itens, mas a quantidade de itens não é conhecida a priori e os itens são fornecidos um de cada vez, pelo usuário. Um item pode ser um ou mais números. A cada item obtido, você deseja validá-lo a fim de verificar se uma sentinela não foi alcançada, para então processá-lo. O resultado de cada item processado independe do resultado dos demais itens. Esse tipo de plano é chamado "step equals init" (Brady, 1999) porque parte das ações do corpo devem existir fora da estrutura como pré-condição: uma primeira entrada de dados deve ser prevista antes da estrutura de repetição.	printf ($formato-de-entrada);
	scanf ($formato-de-entrada, $endereço);
	while ($condição)
	{
	$resultado = $expressao;
	printf ($formato-de-entrada, $resultado);
	printf ($formato-de-entrada);
	scanf ($formato-de-entrada, $endereço);
	}

Plano: Processamento sequencial conta-vezes	

Você deseja realizar repetição com condição de entrada e processar uma expressão, um número de vezes conhecido. O resultado obtido em um laço independe do resultado dos demais laços (não cumulativo). Um contador deve controlar o início e a continuidade da realização da operação até que o número de vezes desejado seja alcançado.	contador=0;
	while (contador < $vezes)
	{
	$resultado = $expressao;
	printf ($formato-de-entrada, $resultado);
	contador++);
	}

Plano: Processamento de seqüência com condição de entrada	

Esse plano é semelhante ao plano "processamento seqüencial com condição de entrada" exceto em que o resultado desejado é aquele obtido ao fim de todas as repetições. Ele é obtido de uma mesma operação entre todos os elementos da seqüência, como por exemplo, a soma entre todos os números de uma seqüência. Esse tipo de plano é também chamado "step equals init" (Brady, 1999) porque parte das ações do corpo devem existir fora da estrutura como pré-condição: uma primeira entrada de dados deve ser prevista antes da estrutura de repetição.	printf ($formato-de-entrada);
	scanf ($formato-de-entrada, &$variavel);
	while($condição)
	{
	$resultado=$resultado$operador$variavel;
	printf ($formato-de-entrada);
	scanf ($formato-de-entrada, &$variavel);
	}

continua

continuação

SITUAÇÃO	ESQUEMA
Plano: Processamento de constante conta vezes	

Você deseja realizar repetição com condição de entrada e processar um valor constante, um número de vezes conhecido. Um contador deve controlar o início e a continuidade da realização da operação até que o número de vezes desejado seja alcançado.	contador=0;
	while(contador < $vezes)
	{
	$variavel=$variavel$operador$valor;
	contador++;
	}

Plano: Encontra mínimo controlado pelo usuário	

Você deseja encontrar o valor mínimo de uma seqüência numérica onde o usuário fornece um numero de cada vez. Porém, o tamanho da seqüência não é conhecido a priori. Portanto o término da seqüência é indicado pelo usuário através da digitação de uma letra especial. Esse tipo de plano é também chamado "step equals init" (Brady,) porque parte das ações do corpo devem existir fora da estrutura como pré-condição: inicialmente, o primeiro número da seqüência é assumido como sendo o menor.	printf($formato-de-saída);
	scanf($formato-de-entrada, &$variavel);
	$minimo=$variavel;
	do {
	printf($formato-de-saída);
	scanf($formato-de-entrada, &$variavel);
	if ($variavel < $minimo)
	{
	$minimo=$variavel;
	}
	printf($formato-de-saída);
	tecla=getch();
	} while ($condicao);
	printf ($formato-de-entrada, $minimo);

Plano: Encontra máximo conta vezes	

Você deseja encontrar o valor máximo de uma seqüência numérica onde o usuário fornece um numero de cada vez. Porém, o tamanho da seqüência é conhecido a priori. A condição para repetição depende do termino da seqüência. Um contador controla se a seqüência chegou ao fim ou não. Esse tipo de plano é também chamado "step equals init" (Brady,) porque parte das ações do corpo devem existir fora da estrutura como pré-condição: inicialmente, o primeiro número da seqüência é assumido como sendo o maior.	printf($formato-de-saída);
	scanf($formato-de-entrada, &$variavel);
	$maximo=$variavel;
	$contador=0;
	while ($contador != $tamanho)
	{
	printf($formato-de-saída);
	scanf($formato-de-entrada, &$variavel);
	if ($variavel > $maximo)
	{
	$maximo = $variavel;
	}
	contador++;
	}
	printf ($formato-de-entrada, $maximo);

continua

continuação

SITUAÇÃO	ESQUEMA
Plano: Repetição com condição de entrada controlada pelo usuário	

Você deseja repetir um conjunto de ações se uma certa condição for verdadeira (repetição com condição de entrada (Brandão, 2003). Se a condição for falsa, você não quer executar o conjunto de ações nenhuma vez. O conjunto de ações não interfere na continuidade da repetição. A repetição é controlada pelo usuário, através de uma letra especial (sentinela). A condição é um teste com respeito a essa letra especial.	while($condição)
	{
	printf($formato-de-saída);
	tecla=getch;
	}

Plano: Repetição com condição de continuação controlada pelo usuário	

Você quer executar um conjunto de ações pelo menos uma vez (repetição com condição de continuação (Brandão, 2003). Se uma certa condição for verdadeira você quer continuar repetindo essas ações até que a condição se torne falsa. O conjunto de ações não interfere na continuidade da repetição. A repetição é controlada pelo usuário, através de uma letra especial (sentinela). A condição é um teste com respeito a essa letra especial.	do {
	printf($formato-de-saída);
	tecla=getch;
	} while ($condicao);

conclusão

Tabela 4 – Planos de Repetição Limitada

SITUAÇÃO	ESQUEMA
Plano: Conta vezes	

Você precisa repetir uma ação (ou ações) um numero conhecido (fim) de vezes	for (contador=1; contador<=$fim; contador++)
	{ $ação
	}

Plano: Gera sequencia	

Você quer gerar uma seqüência numérica crescente, com inicio e fim conhecidos. A distancia entre um numero e outro (passo) da seqüência é constante e conhecida. Ex.: 3, 6, 9, 12....	for (k=$inicio; k<=$fim; k= k + $passo)
	{ $ação
	}

Plano: Gera sequência reversa	

Você quer gerar uma seqüência numérica, em ordem reversa (do maior para o menor numero), com inicio e fim conhecidos. A distancia entre um numero e outro da seqüência é constante e conhecida. Exemplo: 3, 6, 9, 12....	for (k=$fim; k>=$inicio; k= k - $passo)
	{ $ação
	}

Plano: Gera duas sequências	

Você quer gerar duas seqüências ao mesmo tempo, uma crescente e a outra decrescente. O ultimo valor da seqüência crescente é informado pelo usuário. A distancia entre um numero e outro da seqüência é de uma unidade.	printf($formato-de-saída);
	scanf("%i",$fim);
	for (p=1, q=$fim; p<= $fim; p++, q--)
	{
	$ação;
	}

Plano: Soma de sequência iterativa	

Você quer somar uma seqüência de números. O tamanho da seqüência é conhecido. Os números da seqüência são fornecidos pelo usuário.	soma=0;
	for(k=1; k<$tamanho; k++)
	{
	printf($formato-de-saída);
	scanf($formato-de-entrada, &$variavel);
	soma=soma+$variavel;
	}

Plano: Soma de sequência automática	

Você quer somar uma seqüência de números. O início e o tamanho da seqüência são conhecidos. Os números k são gerados automaticamente de forma crescente. A distância entre um número e outro da seqüência é de uma unidade. Pode ser desejável realizar algum processamento com o valor de k antes de realizar a soma.	soma=0;
	for(k=$inicio; k<=$tamanho; k++)
	{
	soma=soma+$numero;
	}

Plano: Tabuada	

Você quer multiplicar um valor fixo com cada número k de uma sequência gerada automaticamente (tipo tabuada). A seqüência é gerada de forma crescente.	for (k=$inicio; k <= $fim; k++)
	{
	resultado=k*$valor;
	printf ($formato-de-entrada, $variaveis);
	}

continua

continuação

SITUAÇÃO	ESQUEMA
Plano: Produto de sequência automática	

Você quer multiplicar todos os números de uma sequência numérica entre si, onde início e fim da seqüência são conhecidos. A multiplicação começa do menor para o maior número da sequência. Ex.: 2*3*4*5*...	produto=1;
	for (k=$inicio; k<=$fim; k++)
	{
	produto=produto*k;
	}
	printf ($formato-de-entrada, $variaveis);

Plano: Produto de seqüência automática reversa	

Você quer multiplicar todos os números de uma seqüência numérica entre si. A multiplicação começa a partir de um número positivo (fim) fornecido pelo usuário até chegar a 1. Ex.: 5*4*3*2*1	produto=1;
	for (k=$fim; k>=1; k--)
	{
	produto=produto*k;
	}
	printf ($formato-de-entrada, $variaveis);

Plano: Média de seqüência iterativa	

Você deseja obter a média de uma certa quantidade de valores. A quantidade é conhecida a priori. Os valores são obtidos do usuário, um de cada vez. Os valores podem ser usados diretamente ou após algum processamento.	$soma=0;
	for (k=1; k<=$quantidade; k++)
	{
	printf($formato-de-saída);
	scanf($formato-de-entrada, $endereço);
	$soma=$soma+$variavel;
	}
	media=$soma/$quantidade;
	printf ($formato-de-entrada, media);

Plano: Máximo ou mínimo	

Você deseja encontrar o valor máximo ou mínimo de uma seqüência numérica onde o usuário fornece um numero de cada vez. O tamanho da seqüência é conhecido a priori. Inicialmente, o primeiro numero da seqüência é assumido como sendo o maior ou o menor.	printf($formato-de-saída);
	scanf($formato-de-entrada, &numero);
	$maxmin= numero;
	for (cont=1; cont<$tamanho; cont++)
	{
	printf($formato-de-saída);
	scanf($formato-de-entrada, &numero);
	if (numero $relacional $maxmin)
	{
	$maxmin = numero;
	}
	}

continua

continuação

SITUAÇÃO	ESQUEMA
Plano: Ordem de seqüência	

Você deseja verificar se uma seqüência numérica fornecida pelo usuário está sendo informada em ordem crescente, decrescente ou desordenada. Se um número estiver fora de ordem, uma ação ou conjunto de ações devem ser executados. O tamanho da seqüência é conhecido a priori.	printf($formato-de-saída);
	scanf($formato-de-entrada, &numero);
	anterior=numero;
	for (cont=1; cont<$tamanho; cont++)
	{
	printf($formato-de-saída);
	scanf($formato-de-entrada, &proximo);
	if (anterior $relacional proximo)
	{ $ação
	}
	anterior=proximo;
	}

Plano: Soma pares	

Você deseja somar, entre si, somente os números pares de uma sequencia numerica fornecida pelo usuario. O tamanho da seqüência é conhecida a priori.	soma=0;
	for(cont=1;cont<=$tamanho;cont++)
	{
	printf("\nEntre com o numero: ");
	scanf("%i",&numero);
	if(numero%2 == 0)
	{
	soma=soma+numero;
	}
	}

conclusão

4.2 Decomposição dos Planos

Tabela 5 – Decomposição dos Planos Básicos

PLANO	TAREFAS PRIMITIVAS	ESQUEMAS PRIMITIVOS
Declaração de variáveis simples	• Declara variáveis simples	• `$tipo $variaveis;`
Inicialização	• Inicializa variáveis	• `$variavel= $valor;`
Entrada por teclado	• Emite mensagem para o usuário • Obtem dados do usuário	• `printf($formato-de-saída);` • `scanf($formato-de-entrada; $enderecos);`
Entrada caracter	• Emite mensagem para o usuário • Obtem um caracter	• `printf($formato-de-saída);` • `$variavel = getch();`
Processamento	• Constrói expressão para calculo	• `$variavel=$expressao;`
Saída de dados	• Mostra dados na tela	• `printf($formato-de-saída, $variaveis);`
Mensagem	• Emite mensagem para o usuário	• `printf($formato-de-saída);`

Tabela 6 – Decomposição dos Planos de Seleção

PLANO	TAREFAS PRIMITIVAS	ESQUEMAS PRIMITIVOS
se-ou-não	• Insere condição • Constrói ações para condição verdadeira • Finaliza if	• if ($condição) • { $ação; • }
ação alternativa	• Insere condição • Constrói ações para condição verdadeira • Finaliza if • Constrói ações para condição falsa • Finaliza else	• if ($condição) • { $ação; • } • else { $ação; • }

expressão condicional	• Insere condição
	• Constrói ações para condição verdadeira
	• Constrói expressao1
	• Finaliza if
	• Constrói ações para condição falsa
	• Constrói expressao2
	• Finaliza else
	• Constrói expressao3
		• if ($condição)
		• {
		• temp=$expressao1;
		• }
		• else {
		• temp=$expressão2;
		• }
		• $variavel=$expressão3;

escolha seqüencial tipo-1	• Insere condição-1 • Constrói ações para condição1 verdadeira • Finaliza if-1 • Constrói ações para condição-1 falsa • Insere condição-2 • Constrói ações para condição-2 verdadeira • Finaliza if-2 • Constrói ações para condição-2 falsa • Finaliza else-2 • Finaliza else-1	• if ($condição1) • { $ação; • } • else { • if ($condição-2) • { $ação; • } • else { $ação; • } • }
escolha seqüencial tipo-2	• Insere condição-1 • Constrói ações para condição-1 verdadeira • Finaliza if-1 • Constrói ações para condição-1 falsa • Insere condição-2 • Constrói ações para condição-2 verdadeira • Finaliza if-2 • Constrói ações para condição-2 falsa • Insere condição-3 • Constrói ações para condição-3 verdadeira • Finaliza if-3 • Constrói ações para condição-3 falsa • Finaliza else-3 • Finaliza else-2 • Finaliza else-1	• if ($condição-1) • { $ação; • } • else { • if ($condição-2) • { $ação; • } • else { • if ($condição-3) • { $ação; • } • else { $acao; • } • } • }

continua

continuação

PLANO	TAREFAS PRIMITIVAS	ESQUEMAS PRIMITVOS
condições sequenciais tipo-1	• Insere condição-1 • Constrói ações para condição-1 verdadeira • Insere condição-2 • Constrói ações para condição-2 verdadeira • Finaliza if-2 • Finaliza if-1	• if ($condição-1) • { • if ($condição-2) • { $ação; • } • }
condições sequenciais tipo-2	• Insere condição-1 • Constrói ações para condição-1 verdadeira • Insere condição-2 • Constrói ações para condição-2 verdadeira • Insere condição-3 • Constrói ações para condição-3 verdadeira • Finaliza if-3 • Finaliza if-2 • Finaliza if-1	• if ($condição-1) • { • if ($condição-2) • { • if ($condição-3) • { $ação; • } • } • }
escolha não relacionada tipo-1	• Insere condição-1 • Constrói ações para condição-1 verdadeira • Finaliza if-1 • Insere condição-2 • Constrói ações para condição-2 verdadeira • Finaliza if-2	• if ($condição-1) • { $ação; • } • if ($condição-2) • { $ação; • }
escolha não relacionada tipo-2	• Insere condição-1 • Constrói ações para condição-1 verdadeira • Finaliza if-1 • Insere condição-2 • Constrói ações para condição-2 verdadeira • Finaliza if-2 • Insere condição-3 • Constrói ações para condição-3 verdadeira • Finaliza if-3	• if ($condição-1) • { $ação; • } • if ($condição-2) • { $ação; • } • if ($condição-3) • { $ação; • }

escolha independente	• Insere condição-1
	• Constrói ações para condição-1 verdadeira
	• Insere condição-2
	• Constrói ações para condição-2 verdadeira
	• Finaliza if-2
	• Constrói ações para condição-2 falsa
	• Finaliza else-2
	• Finaliza if-1
	• Constrói ações para condição-1 falsa
	• Insere condição-3
	• Constrói ações para condição-3 verdadeira
	• Finaliza if-3
	• Constrói ações para condição-3 falsa
	• Finaliza else-3
	• Finaliza else-1
		• if ($condicao-1)
		• {
		• if ($condicao-2)
		• { $acao-x;
		• }
		• else { $acao-y;
		• }
		• }
		• else {
		• if ($condicao3)
		• { $acao-x;
		• }
		• else { $acao-y;
		• }
		• }

conclusão

Tabela 7 – Decomposição dos Planos de Repetição Geral

TAREFAS PRIMITIVAS	ESQUEMAS PRIMITVOS
Plano: Verifica sentinela	
• Insere condição para repetição geral • Constrói ações para condição verdadeira • Emite mensagem p/ usuário fornecer um número • Obtém número • Finaliza laço	• while($condição) • { • printf("Entre com o numero: "); • scanf($formato-de-entrada, &numero); • }
Plano: Processamento sequencial com condição de entrada	
• Emite mensagem p/ usuário fornecer primeiro item • Obtem primeiro item • Insere condição para repetição geral • Constrói ações para condição verdadeira • Insere expressão para calculo • Mostra resultado • Emite mensagem p/ usuário fornecer próximo item • Obtém próximo item • Finaliza laço	• printf ($formato-de-entrada); • scanf ($formato-de-entrada, $endereço); • while($condição) • { • $resultado = $expressao; • printf ($formato-de-ent, $resultado); • printf ($formato-de-entrada); • scanf ($formato-de-entrada, $endereço); • }
Plano: Processamento sequencial conta-vezes	
• Inicia contagem • Insere condição para repetição geral • Constrói ações para condição verdadeira • Insere expressão para calculo • Mostra resultado • Incrementa contador • Finaliza laço	• contador=0; • while (contador < $vezes) • { • $resultado = $expressao; • printf ($formato-de-ent, $resultado); • contador++); • }
Plano: Processamento de seqüência com condição de entrada	
• Emite mensagem p/ usuário fornecer 1o número • Obtém primeiro numero • Insere condição para repetição geral • Constrói ações para condição verdadeira • Insere expressão • Emite mensagem p/ usuário fornecer próximo no. • Obtém próximo numero • Finaliza laço	• printf ($formato-de-entrada); • scanf ($formato-de-entrada, &$variavel); • while($condição) • { • $resultado=$resultado $operador$var; • printf ($formato-de-entrada); • scanf ($formato-de-entrada, &$variavel); • }
Plano: Processamento de constante conta vezes	
• Inicia contagem • Insere condição para repetição geral • Constrói ações para condição verdadeira • Insere expressão • Incrementa contador • Finaliza laço	• contador=0; • while(contador < $tamanho) • { • $variavel=$variavel$operador$valor; • contador++; • }

continua

continuação

TAREFAS PRIMITIVAS	ESQUEMAS PRIMITIVOS
Plano: Encontra mínimo controlado pelo usuário	

• Emite mensagem p/usuário fornecer 1o numero
• Obtém primeiro numero
• Armazena primeiro numero como minimo
• Inicia repeticao
• Emite mensagem p/usuário fornecer proximo no.
• Obtem proximo numero
• Verifica se número é menor que minimo atual
• Constrói ação para condição verdadeira
• Atualiza minimo
• Finaliza if
• Pergunta se usuário (não) deseja parar repetição
• Obtem caracter digitado pelo usuario
• Insere condição para repetição geral
• Mostra o menor numero da sequencia
	• printf($formato-de-saída);
	• scanf($formato-de-entrada, &$variavel);
	• $minimo=$variavel;
	• do {
	• printf($formato-de-saída);
	• scanf($formato-de-entr, &$var);
	• if ($variavel < $minimo)
	• {
	• $minimo=$variavel;
	• }
	• printf($formato-de-saída);
	• tecla=getch();
	• } while ($condicao);
	• printf ($formato-de-entrada, $minimo);

Plano: Encontra maximo conta-vezes	
• Emite mensagem p/ usuário fornecer 1o numero • Obtem primeiro numero • Armazena primeiro numero como maximo • Inicia contagem • Insere condição para controlar contagem • Constrói ações para condição verdadeira • Emite mensagem para usuário fornecer proximo número • Obtem proximo numero • Verifica se numero é maior que maximo atual • Constrói ações para condição verdadeira • Atualiza maximo • Finaliza if • Incrementa contador • Finaliza laço • Mostra o maior numero da seqüência	• printf($formato-de-saída); • scanf($formato-de-entrada, &$variavel); • $maximo=$variavel; • $contador=0; • while ($contador != $vezes) • { • printf($formato-de-saída); • scanf($formato-de-entr, &$var); • if ($variavel > $maximo) • { • $maximo = $variavel; • } • contador++; • } • printf ($formato-de-entr, $maximo);
Plano: Repetição com condição de entrada controlada pelo usuário	
• Garante primeira repetição • Insere condição para repetição geral • Constrói ações para condição verdadeira • Pergunta se usuário (não) deseja parar repetição • Obtem caracter digitado pelo usuário • Finaliza laço	• $inicializa • while($condição) • { • printf($formato-de-saída); • tecla=getch(); • }
Plano: Repetição com condição de continuação controlada pelo usuário	
• Inicia repetição • Pergunta se usuário (não) deseja parar repetição • Obtém caracter digitado pelo usuário • Insere condição para repetição geral	• do{ • printf($formato-de-saída); • tecla=getch(); • } while ($condicao);

conclusão

Tabela 8 – Decomposição dos Planos de Repetição Limitada

TAREFAS PRIMITIVAS	ESQUEMA
Plano: Conta vezes	
• Configura condição para repetição • Constrói ações a serem repetidas • Finaliza laço	• for (cont=1; cont <= $fim; cont++) • { $acao • }
Plano: Gera seqüência	
• Configura parâmetros da sequencia • Constrói ações a serem repetidas • Finaliza laço	• for (k=$inicio; k<=$fim; k=k+$passo) • { $acao • }
Plano: Gera seqüência reversa	
• Configura parâmetros da sequencia • Constrói ações a serem repetidas • Finaliza laço	• for (k=$fim; k>= $inicio; k=k-$passo) • { $acao • }
Plano: Gera duas seqüências	
• Emite mensagem para o usuário fornecer valor • Obtém fim da seqüência • Configura parâmetros das duas seqüências • Constrói ações para condição verdadeira • Finaliza laço	• printf ($formato-de-entrada); • scanf ("%i", $fim); • for (p=1, q=$fim; p <= $fim; p++, q--) • { $ação; • }
Plano: Soma de seqüência iterativa	
• Inicializa soma com elemento neutro • Configura condição para repetição • Constrói ações a serem repetidas • Emite mensagem p/uuário fornecer um número • Obtém número • Adiciona o número no resultado anterior • Finaliza laço	• soma=0; • for(k=1; k<$tamanho; k++) • { • printf($formato-de-saída); • scanf($formato-de-entrada, &$variavel); • soma=soma+$variavel; • }
Plano: Soma de seqüência automática	
• Inicializa soma com elemento neutro • Configura parâmetros da seqüência • Constrói ações a serem repetidas • Adiciona um no. da seqüência na soma anterior • Finaliza laço	• soma=0; • for(k=$inicio; k<=$tamanho; k++) • { • soma=soma+k; • }
Plano: Tabuada	
• Configura parâmetros da sequencia • Constrói ações a serem repetidas • Calcula produto entre um no. da seq. e um valor • Mostra o produto • Finaliza laço	• for (k=$inicio; k <= $fim; k++) • { • resultado=k*$valor; • printf ($formato-de-entrada, $variaveis); • }

continua

continuação

TAREFAS PRIMITIVAS	ESQUEMA
Plano: Produto de seqüência automática	
• Inicializa produto com elemento neutro • Configura parâmetros da sequencia • Constrói ações a serem repetidas • Multiplica no. da seqüência com produto anterior • Finaliza laço • Mostra produto total	• produto=1; • for (k=$inicio; k<=$fim; k++) • { • produto=produto*k; • } • printf ($formato-de-entrada, $variaveis);
Plano: Produto de seqüência automática reversa	
• Inicializa produto com elemento neutro • Configura parâmetros da seqüência reversa • Constrói ações a serem repetidas • Multiplica no. da seqüência com produto anterior • Finaliza laço • Mostra produto total	• produto=1; • for (k=$fim; k>=1; k--) • { • produto=produto*k; • } • printf ($formato-de-entrada, $variaveis);
Plano: Média de seqüência iterativa	
• Inicializa soma com elemento neutro • Configura condição para repetição • Constrói ações a serem repetidas • Emite mensagem p/ o usuário fornecer um valor • Obtém valor • Adiciona valor na soma anterior • Finaliza laço • Calcula media • Mostra media	• soma=0; • for (k=1; k<=$quantidade; k++) • { • printf($formato-de-saída); • scanf($formato-de-entrada, &$variavel); • soma=soma+$variavel; • } • media=soma/$quantidade; • printf ($formato-de-entrada, media);
Plano: Máximo ou mínimo	
• Emite mensagem p/usuário fornecer um número • Obtém número • Assume que no. é o maior (menor) de todos • Configura condição para repetição • Constrói ações a serem repetidas • Emite msg. p/ o usuário fornecer próximo no. • Obtém número • Verifica se número é maior (menor) que anterior • Constrói ações para condição verdadeira • Atualiza máximo (mínimo) • Finaliza if • Finaliza laço	• printf($formato-de-saída); • scanf($formato-de-entrada, &numero); • $maxmin= numero; • for (cont=1;cont < $tamanho;cont++) • { • printf($formato-de-saída); • scanf($formato-de-entrada, &numero); • if (numero $relacional $maxmin) • { • $maxmin = numero; • } • }

continua

continuação

TAREFAS PRIMITIVAS	ESQUEMAS PRIMITIVOS
Plano: Ordem de seqüência	
• Emite mensagem p/usuário fornecer um no. • Obtém número • Assume primeiro número como anterior • Configura condição para repetição • Constrói ações a serem repetidas • Emite msg. p/ o usuário fornecer novo no. • Obtém novo número • Verifica se anterior é maior (menor) que atual • Constrói ações para condição verdadeira • Finaliza if • Assume numero atual como anterior • Finaliza laço	• printf($formato-de-saída); • scanf($formato-de-entrada, &numero); • anterior=numero; • for (cont=1; cont< $tamanho; cont++) • { • printf($formato-de-saída); • scanf($formato-de-entrada, &atual); • if (anterior $relacional atual) • { $ação • } • anterior=atual; • }
Plano: Soma pares	
• Inicializa soma com elemento neutro • Configura condição para repetição • Constrói ações a serem repetidas • Emite msg. p/ usuário fornecer um numero • Obtém número • Verifica se número é divisivel por 2 • Constrói ações para condição verdadeira • Adiciona o número na soma anterior • Finaliza if • Finaliza laço	• soma=0; • for(cont=1; cont<=$tamanho; cont++) • { • printf("\nEntre com o numero: "); • scanf("%i",&numero); • if (numero%2 == 0) • { • soma=soma+numero; • } • }

conclusão

APÊNDICE B – PROBLEMAS PROPOSTOS

Este apêndice apresenta uma amostra dos problemas e correspondentes soluções em linguagem C, modelados em BibPC e propostos ao aluno no TutorC.

Tópico Básico

1. Faça um programa que converte graus em radianos.

   Modelo do Problema:
   - M1: Espaço alocado na memória para os dados
   - M2: Valor em graus obtido do usuário
   - M3: Graus convertido para radianos
   - M4: valor em radianos, na tela

2. Escreva um programa em C que permita ao usuário entrar com o valor de dois lados de um retângulo. Calcule e mostre o perímetro (p=2*lado1+2*lado2) e a área do retângulo (A=lado1*lado2). (Nível 2)

   Modelo do Problema:
   - M1: Espaço alocado na memória para os dados
   - M2: Dois lados obtidos do usuário
   - M3: Perímetro calculado
   - M4: Valor do perímetro e da área na tela
   - M5: Área calculada

Tópico Seleção

1. Faça um programa que recebe valores de dois lados de uma figura geométrica e verifica se é quadrado ou retângulo. Se quadrado, verifica se o lado é menor que 5. Nesse caso, emitir a mensagem "quadrado pequeno", senão "quadrado grande". Se retângulo verifica se um dos lados é menor que 7. Nesse caso, emitir a mensagem "retângulo pequeno", senão "retângulo grande".

   Modelo do Problema:
   - M1: Espaço alocado na memória para os dados
   - M2: Dois lados obtidos do usuário
   - M3: Figura classificada
   - M4: Mensagem "Quadrado pequeno" na tela
   - M5: Mensagem "Quadrado grande" na tela
   - M6: Mensagem "Retângulo pequeno" na tela
   - M7: Mensagem "Retângulo grande" na tela

   Resolução em C:

```
main()
{
float a, b;
printf("\n\n Digite dois lados");
scanf("%f, %f", &a, &b);
if(a==b)
        {       if(a<5)
                        printf("\n\nQuadrado pequeno");
                else
                        printf("\n\nQuadrado grande");
        }
else    {       if(a<7 || b<7)
                        printf("\n\n Retângulo pequeno");
                else
                        printf("\n\nRetângulo grande");
        }
}
```

Tópico Repetição Geral

1. Faça um programa que permita que o usuário entre com uma seqüência de inteiros, um de cada vez. Quando o usuário desejar encerrar a seqüência, deve digitar F. O programa deve ainda mostrar na tela o menor número do conjunto de números fornecidos.

   Modelo do Problema:
   - M1: Espaço alocado na memória para armazenar decimal
   - M2: Espaço alocado na memória para armazenar caracter
   - M3: Seqüência de números obtida e menor número identificado, enquanto usuário não digita F

   Resolução em C:

```
void main( )
{
float num, min;
char tecla;
printf("Entre com primeiro número: ");
scanf("%f", &num);
min=num;
do
    {
    printf("Entre com próximo número: ");
    scanf("%f", &num);
    if (num < min)
            min=num;
      printf("\nDigite F para sair");
      tecla=getch;
    } while (tecla != 'F');
printf ("minimo=%.1f",min);
}
```

2. Faça uma calculadora que efetua e mostra o resultado de três tipos de operações: soma, subtração e multiplicação. O usuário deve entrar com a expressão para cálculo, como por exemplo: 3 + 5 ou 4 – 2 ou 6 * 3

Se a expressão for desconhecida, o programa deve emitir a mensagem: "Não compreendo". O programa deve solicitar novas expressões enquanto o usuário não digitar a letra S.

Modelo do Problema:

- M1: Espaço alocado na memória para armazenar caracter
- M2: Espaço alocado na memória para armazenar inteiros
- M3: Programa em execução enquanto a letra S não foi digitada
- M4: Expressão obtida do usuário
- M5: Expressão classificada segundo as operações: soma, subtração, multiplicação, desconhecida.
- M6: Calculo da soma
- M7: Resultado da soma na tela
- M8: Cálculo da subtração
- M9: Resultado da subtração na tela
- M10: Calculo da multiplicação
- M11: Resultado da multiplicação na tela
- M12: Mensagem "Não compreendo", na tela

Resolução em C:

```
main() {
        char operador, tecla;
        float num1, num2, soma, subtrai, multiplica;
                do{
                  printf("\nDigite a expressão a ser calculada: ");
                  scanf ("%f%c%f", &num1, &operador, &num2);
                  if(operador=='+')
                          {
                          soma=num1+num2;
                          printf("\nA soma e %.2f. \n", soma);
                          }
                  else
                          if(operador=='-')
                                  {
                                  subtrai=num1-num2;
                                  printf("\nA subtração e %.2f. \n", subtrai);
                                  }
                          else
                                  if(operador=='*')
                                          {
                                          multiplica=num1*num2;
                                          printf("\nA multiplicacao e %.2f. \n", multiplica);
                                          }
                                  else    printf("\n Nao compreendo");
                  printf("\nDigite S para sair\n");
                  tecla=getche();
                  } while (tecla!= 'S' && tecla!='s')
        }
```

Tópico Repetição Limitada

1. Construa um programa que, para um grupo de 50 valores inteiros fornecidos pelo usuário, determine: (1) a soma dos números positivos; (2) a quantidade de valores negativos.

Modelo do problema:

- M1: Espaço alocado na memória para armazenar inteiros
- M2: Acumuladores inicializados
- M3: Quantidade dos valores do usuário contada
- M4: Um número obtido do usuário
- M5: Classificação do número entre positivo ou negativo
- M6: Soma dos positivos calculada
- M7: Quantidade de negativos calculada
- M8: Soma e quantidade dos números, na tela

Resolução em C:

```
void main()
{
int  contador, número, positivos, negativos;
positivos=0; negativos=0;
for (contador=1; contador <= 50; contador++)
      {
      printf ("\nDigite um número inteiro: ");
      scanf("%i", &número);
      if (número >= 0)
            positivos = positivos+número;
      else
            negativos = negativos+1;
      }
printf ("\nSoma=%i  Quantidade=%i", positivos, negativos);
}
```

2. Faça um programa que calcula a tabuada de um número inteiro n, dado pelo usuário, entre 1 e 9. O programa deve permitir cálculo de outras tabuadas até que o usuário digite n fora da faixa entre 1 e 9.

   Modelo do problema:

   - M1: Espaço alocado na memória para armazenar inteiros
   - M2: Recepção e validação da tabuada desejada, repetidamente
   - M3: Tabuada calculada e apresentada na tela

   Resolução em C

```
void main()
{
int n, k, resultado;
printf("\n Entre com o número: ");
scanf("%i",&n);
while(n>0 && n<10)
      {
      for (k=0; k<=10; k++)
            {
            resultado=k*n;
            printf("%i*%i=%i", k, n,resultado);
            }
      printf("\n Entre com o número: ");
      scanf("%i",&n);
      }
}
```

APÊNDICE C – META-VARIÁVEIS

Este apêndice apresenta o conjunto de meta-variáveis criadas para os planos de programação propostos no capítulo 5. A utilização dessas meta-variáveis pode ser vista na descrição de todos os planos de programação modelados, como apresenta o apêndice A.

$tipo – indica que a instância deve ser um tipo de dado. Por exemplo: int, float, char.

$valor – indica que a instância deve ser um valor numérico.

$formato-de-entrada – indica que a instância deve ser uma cadeia de caracteres.

$endereços – indica que a instância deve ser um ou mais endereços de memória. Por exemplo, &x, no caso da linguagem C.

$variáveis - indica que a instância deve ser uma ou mais variáveis que contêm dados a serem mostrados na tela.

$variavel - indica que a instância deve ser uma variável.

$expressão - indica que a instância deve ser uma expressão.

$condição - indica que a instância deve ser uma condição que pode envolver tanto operadores relacionais como lógicos.

$ação – indica que a instância deve ser um ou mais planos de programação, os quais devem também ser instanciados.

$resultado - indica que a instância deve ser uma variável que armazena o resultado de uma expressão.

$operador – indica que a instância deve ser um operador matemático.

$relacional - indica que a instância deve ser um operador relacional.

$tamanho - indica que a instância deve ser um valor ou variável que indica o tamanho, por exemplo, de uma seqüência.

$minimo - indica que a instância deve ser um valor (ou variável) que representa o mínimo.

$maximo - indica que a instância deve ser um valor (ou variável) que representa o máximo.

$maxmin - indica que a instância deve ser um valor (ou variável) que representa o máximo ou o mínimo.

$vezes – indica que a instância deve ser um valor (ou variável) que representa um certo número de vezes.

$fim - indica que a instância deve ser um valor (ou variável) que representa o último número de uma seqüência.

$inicio - indica que a instância deve ser um valor (ou variável) que representa o primeiro número de uma seqüência.

$passo – indica que a instância deve ser um valor que representa o incremento ou decremento aplicado a uma seqüência.

$numero – indica que a instância deve ser um valor (ou variável) numérico.

$quantidade – indica que a instância deve ser um valor (ou variável) que representa uma certa quantidade.

APÊNDICE C – MODELOS DE PROVAS

Este anexo apresenta os modelos de provas utilizados para avaliação e reavaliação do Módulo 1 da disciplina de Introdução à Computação e Cálculo Numérico da FACENS. Os resultados obtidos nas provas, pelos alunos, foram utilizados na avaliação relatada no Capítulo 8. Os modelos estão disponíveis para comparações entre graus de dificuldade das provas. Foi realizado um esforço para manter um equilíbrio nas exigências das provas.

FACENS
FACULDADE DE ENGENHARIA DE SOROCABA

Reconhecida pela Portaria Ministerial n° 367 de 03/06/1980
MANTIDA PELA
Associação Cultural de Renovação Tecnológica Sorocabana - ACRTS
Declarada de Utilidade Pública Federal - Decreto n° 86.431 de 02/10/1981

Disciplina: ICCN - Introdução à Computação e Cálculo Numérico
Engenharia Elétrica – Noturno – **Avaliação de Teoria Módulo 1** – 08/05/2003

Nome:_____Nº FACENS:_____Turma_____

Duração da Prova: 1h e 40 minutos

PARTE A - COMPREENSÃO DE PROGRAMAS

1. (Valor: 2,5) Seja o programa abaixo. Supondo os valores abaixo para n1 e n2, qual valor de r será mostrado na tela ao término da execução do programa?

```
#define PESO 0.6
void main()
{
float n1, n2, r;
printf("Entre com dois valores: ");
scanf("%f,%f", &n1, &n2);
if (n1>=0 && n1<=10)
       {
       if(n2>=0 && n2<=7)
              r=(n1+n2)/2;
       }
else
       r=(n1+n2)*PESO;
printf("%f", r);
}
```

n1	n2	r
4	8	
8	7	
12	5	

2. (Valor: 2,5) Quais os valores de a e b ao fim do 3o laço do for externo? E o valor de b ao fim do 2o laço do for interno?

```
void main()
{
int a, b;
for (a=0; a<5; a+=2)              // for externo
      {
            printf("\n");
            a--;
            for (b=0; b<3; b++) // for interno
            {
            printf("\na=%i e b=%i ",a,b);
            }
      }
}
```

PARTE B - CONSTRUÇÃO DE PROGRAMAS

3. (Valor: 2,5) Faça um programa que multiplique os números inteiros entre 100 e 120 e mostre o produto total.

4. (Valor: 2,5) Construa um programa que recebe dois números inteiros x e y. Apresenta o produto deles, se y for maior que x. Caso contrário, apresenta a quantidade de números entre x e y.

FACENS
FACULDADE DE ENGENHARIA DE SOROCABA

Reconhecida pela Portaria Ministerial n° 367 de 03/06/1980
MANTIDA PELA
Associação Cultural de Renovação Tecnológica Sorocabana - ACRTS
Declarada de Utilidade Pública Federal - Decreto n° 86.431 de 02/10/1981

Disciplina: ICCN - Introdução à Computação e Cálculo Numérico
Engenharia Elétrica – Noturno – **Reavaliação de Teoria Módulo 1** – 09/10/2003

Nome:_____Nº FACENS:_____Turma____

Instruções:

1. Não destaque as folhas do caderno
2. Preencha o cabeçalho acima
3. Não é permitido o uso de folhas avulsas para rascunho
4. Não é permitido o uso de calculadoras
5. Não é permitido a consulta a livros, apontamentos ou colegas
6. É permitida a consulta aos planos de programação fornecidos pela profa.
7. A parte B (Construção de Programas) consta de 3 questões. Resolva duas à sua escolha.

Duração da Prova: 1h e 40 minutos

Parte	Questão	Valor	Nota
A	1	2,0	
B	1	2,5	
B	2	2,5	
B	3	2,5	
C	1	1,5	
C	2	1,5	

Parte A - Compreensão de Programas

1. Simule a execução do programa abaixo, destacando a sua saída. A saída do programa consiste de tudo que resulta dos comandos **printf**.

```
#include <stdio.h>
void main()
{
int a, b, c, d, num;

printf ("Entre com os dois últimos dígitos do seu número FACENS: ");
scanf ("%i", &num);
printf ("numero=%i\n", num);

d=num%3;
a=5;
b=10*(2+3*d);
c=(6/2)*(1+ d)*10;

printf("a=%i b=%i c=%i \n", a, b, c);

if (c > a && c <= b)
      {
      printf("Good \n");
      }
else
      {
      printf ("Great \n");
      }

while (a > 0)
      {
      c=b;
      b=a;
      a-=2;
      printf("a=%i b=%i c=%i \n", a, b, c);
      }
}
```

Para efeito de correção só será considerada a saída do programa (tabela 2). Você pode usar a tabela 1 como bem entender.

Tabela 1				
a	**b**	**c**	**d**	**num**

Tabela 2 – Saída (Monitor)
Entre com os dois últimos dígitos do seu número da FACENS:

Parte B - Construção de Programas

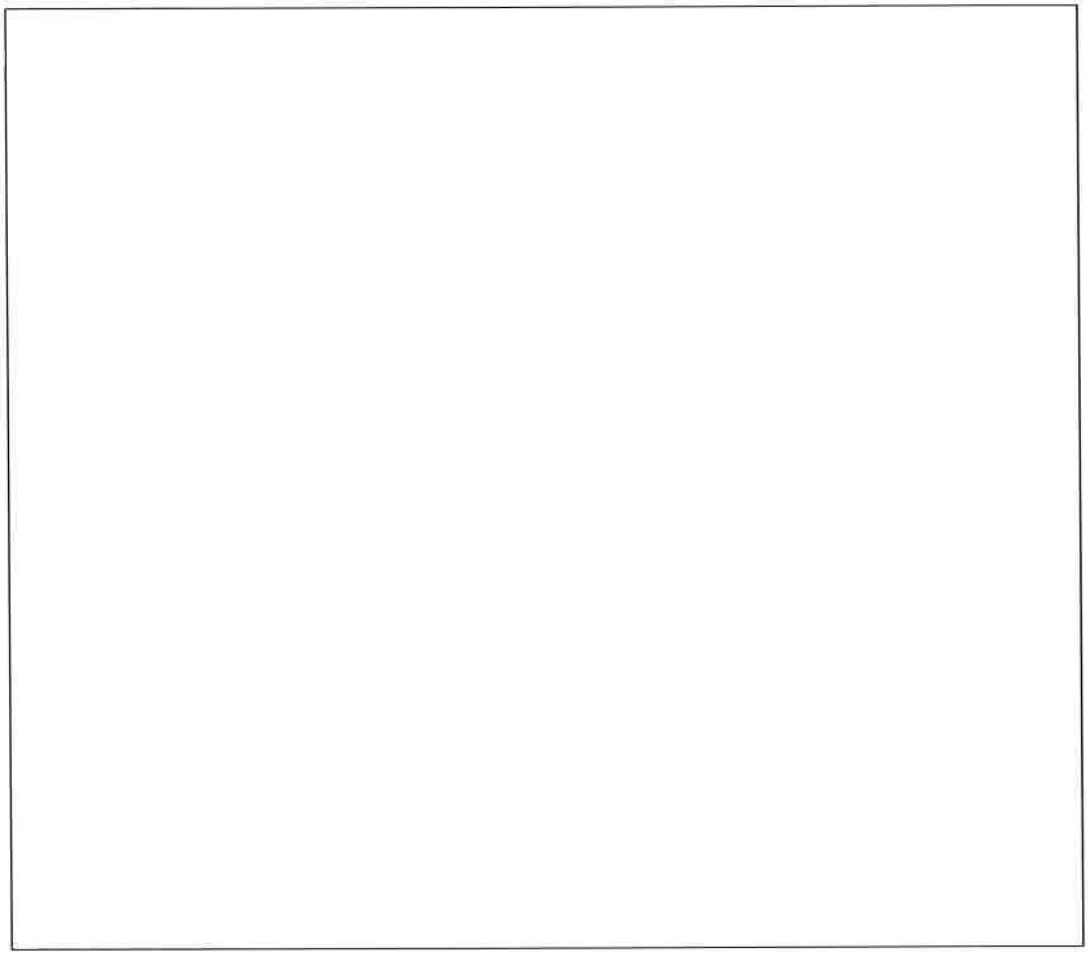
3. Faça um programa em C que receba médias de alunos, até que o usuário informe que deseja parar, digitando a letra N. Ao final, o programa deve mostrar quantos alunos serão reavaliados e quantos não serão, sendo que a média mínima para não ser reavaliado é 7.

Metas do Problema	Planos de Programação
Cálculo do total de alunos a serem reavaliados	
Espaço alocado na memória para armazenar caracter	
Média classificada	
Contadores de alunos inicializados	
Espaço alocado na memória para armazenar decimal	
Total de alunos aprovados e reprovados na tela	
Média obtida do usuário	
Programa em execução enquanto o usuário não digita N	
Espaço alocado na memória para armazenar inteiro	
Cálculo do total de alunos aprovados	

4. Fatorial é uma operação que calcula produtos cujos fatores são números naturais consecutivos. Elabore um programa que calcula o produto fatorial de um número inteiro dado pelo usuário. Se o número não for positivo, o programa não deve calcular. Exemplos:

   4!=4*3*2*1
   5!=5*4*3*2*1
   6!=6*5*4*3*2*1

Metas do problema	Planos de Programação
Produto fatorial calculado e apresentado na tela	
Espaço alocado na memória para armazenar inteiros	
Número obtido do usuário para cálculo do fatorial	
Número positivo identificado	



5. Construa um programa que, para um grupo de 50 valores inteiros fornecidos pelo usuário, determine: (1) a soma dos números positivos; (2) a quantidade de valores negativos.

Metas do Problema	Planos de Programação
Acumuladores inicializados	
Classificação do número entre positivo ou negativo	
Espaço alocado na memória para armazenar inteiros	
Soma e quantidade dos números, na tela	
Um número obtido do usuário	
Soma dos positivos calculada	
Quantidade de negativos calculada	
Quantidade dos valores do usuário contada	

Parte C – Identificação de Metas e Planos de Programação

Assinale na 1ª coluna somente as metas que devem ser alcançadas para resolver os problemas abaixo e forneça o nome dos planos que resolvem as metas selecionadas por você.

1. Suponha a entrada do preço de 75 produtos, pelo usuário. Calcule a soma dos preços dos produtos. Mostre o valor total na tela.

X	Metas do problema	Nome do plano
	Maior número da seqüência identificado	
	Recepção e soma de 75 produtos	
	Dois valores obtidos do usuário	
	Espaço alocado na memória para inteiros	
	Valor positivo identificado	
	Um valor dado pelo programador	
	Um valor obtido do usuário	
	Total na tela	
	Cálculo da soma	
	Mensagem de erro na tela	
	Soma inicializada	
	Tamanho da seqüência obtido do usuário	
	Quantidade de produtos contada	

2. Elabore um programa que calcula a área e o perímetro de círculos de raios fornecidos pelo usuário. Enquanto o usuário não digitar a letra S, o programa deve obter novo raio. A área e o perímetro devem ser calculados e mostrados na tela somente para valores positivos de raio.

X	Metas do Problema	Nome do Plano
	Programa em execução enquanto uma condição for verdadeira	
	Classificação de valor obtido do usuário	
	Maior número da seqüência identificado	
	Resultados na tela	
	Seqüência de valores gerados automaticamente	
	Espaço alocado na memória para armazenar caracter	
	Dois lados obtidos do usuário	
	Perímetro calculado	
	Programa em execução enquanto o valor do raio for positivo	
	Área calculada	
	Perímetro positivo identificado	
	Um valor dado pelo programador	
	Espaço alocado na memória para armazenar nºs decimais	
	Um valor obtido do usuário	
	Um valor entre 0 e 20 ou mensagem de erro na tela	
	Tamanho da seqüência obtido do usuário	

FACENS
FACULDADE DE ENGENHARIA DE SOROCABA

Reconhecida pela Portaria Ministerial n° 367 de 03/06/1980
MANTIDA PELA
Associação Cultural de Renovação Tecnológica Sorocabana - ACRTS
Declarada de Utilidade Pública Federal - Decreto n° 86.431 de 02/10/1981

Disciplina: ICCN - Introdução à Computação e Cálculo Numérico

Engenharia Elétrica – Noturno – **Avaliação de Laboratório Módulo 1** – 06/05/2003

Nome:_____Nº FACENS:_____Turma_____

Duração da Prova: 1h e 40 minutos

Instruções:

- Não é permitida consulta em apostilas, livros ou colegas
- Não é permitido o uso de calculadoras, disquetes e outros materiais
- Use o verso desta folha para rascunho
- Construa o programa no Turbo C e ao término, chame o professor para executá-lo. Em seguida imprima o código e entregue-o grampeado com esta folha.

Enunciado: Faça um programa que classifica números. Mostre na tela a classificação ("positivo", "negativo" ou "nulo") e o número classificado, somente se ele for: (a) zero ou (b) negativo e estiver entre -1 e -10 ou (c) positivo e estiver entre 2 e 20. Para qualquer outro valor deve ser emitida a mensagem "número inválido". Ao fim de cada classificação, permita que o usuário possa encerrar o programa, da seguinte forma: se digitado f, o programa encerra, qualquer outra tecla solicita novo número.

Observações: Provas de laboratório consistem de apenas um exercício de construção de programas usando o compilador. Abaixo são apresentados enunciados de outras turmas de laboratório.

1. Construa um programa que receba três valores inteiros e positivos A, B e C, do usuário e verifique se estes valores podem representar os lados de um triângulo: se um dos lados for maior que a soma dos outros dois, então não é triângulo. Em caso afirmativo emitir a mensagem "A, B e C formam um triângulo", senão apresentar na tela a quantidade de números existentes entre A e C, se A for maior que C.

2. Faça um programa que calcula e mostra a raiz quadrada de uma seqüência numérica fornecida pelo usuário. Deve mostrar ainda a soma total dos números fornecidos. O programa deve terminar se o usuário fornecer um número negativo ou quando a soma ultrapassar o valor 100.

3. Elabore um programa para calcular o salário bruto e líquido de estagiários de uma empresa. Considere as entradas de salário hora, horas trabalhadas e total de descontos. Exemplo:
   salário hora= R$ 5,00
   horas trabalhadas = 80 horas
   total de descontos= R$ 30,00
   salário bruto = 5*80 = R$ 400,00
   salário liquido = 400,00 - 30,00 = R$ 370,00
   Se o salário hora for menor que 4 e horas trabalhadas maior que 100, emitir a mensagem "Contratar", caso contrário emitir "Dispensar". Deixe o controle de término do programa para o usuário, da seguinte forma: se ele digitar 'f', o programa deve encerrar, qualquer outra tecla deve solicitar novos valores.

4. Elabore um programa onde um médico informe o código de uma doença. Apresente a descrição da doença e também o medicamento necessário, de acordo com a tabela abaixo. Se o médico informar o mesmo código 3 vezes, emitir também a mensagem "Memorize" para o médico. Qualquer outro código deve ser emitida a mensagem "inválido" e encerrar o programa.

Código da doença	Descrição da doença	Medicamento
0	Gripe	Vitamina C
1	Dor de cabeça	Analgésico
-	Invalido	

FACENS
FACULDADE DE ENGENHARIA DE SOROCABA

Reconhecida pela Portaria Ministerial n° 367 de 03/06/1980
MANTIDA PELA
Associação Cultural de Renovação Tecnológica Sorocabana - ACRTS
Declarada de Utilidade Pública Federal - Decreto n° 86.431 de 02/10/1981

Disciplina: ICCN - Introdução à Computação e Cálculo Numérico

Engenharia Elétrica – Noturno – **Reavaliação de Laboratório Módulo 1** – 07/10/2003

Nome:_____Nº FACENS:_____Turma_____

Duração da Prova: 1h e 40 minutos

Instruções:

- Não é permitida consulta em apostilas, livros ou colegas
- Não é permitido o uso de calculadoras, disquetes e outros materiais
- Use o verso desta folha para rascunho
- Construa o programa no Turbo C e ao término, chame o professor para executá-lo. Em seguida imprima o código e entregue-o grampeado com esta folha.

Enunciado: Elabore um programa em linguagem C para obter a classificação de atletas para um campeonato. A nota final é composta pela soma dos resultados de três etapas (1 a 10) aplicando-se os pesos respectivos, segundo a fórmula: Nota final = R1*0,5 + R2*0,3 + R3*0,2. Dependendo do valor da nota final, as seguintes mensagens devem ser emitidas:

- "Classificado", se a nota final for maior que 8.0;
- "Reserva", se a nota final estiver entre 6.0 e 8.0 (inclusive);
- "Desclassificado", se a nota final for menor que 6.0.

O programa deve ser interrompido se o usuário fornecer algum resultado negativo ou superior a 10. Mostrar na tela a quantidade final de atletas classificados.

Abaixo são apresentados enunciados de outras turmas de laboratório.

1. Elabore um programa para calcular o custo de fabricação (cf) de produtos de uma empresa. Considere as entradas do código numérico do produto, custos de matéria-prima (mp), mão-de-obra (mo) e impostos (i). A fórmula para cálculo do custo de fabricação é: cf = mp*2,3 + mo*1,7 + i
   Se o custo de fabricação for maior que 50 e impostos maior que 9, emitir a mensagem "Não fabricar", caso contrário mostrar o código do produto. Deixe o controle de término do programa para o usuário, da seguinte forma: se ele digitar 'f', o programa deve encerrar, qualquer outra tecla deve solicitar valores do próximo produto. Ao encerrar o programa, apresentar a quantidade de produtos que devem ser fabricados.

2. Dado um número inteiro positivo n, apresentar na tela todos os números pares entre 1 e n. Se a quantidade de números pares estiver entre 5 e 10, calcule e mostre a média aritmética desses números.

ANEXO A – LISTAGEM DE ALUNOS E NOTAS

Os dados referentes a este anexo foram utilizados na construção de histogramas e gráficos de distribuição normal apresentados no Capítulo 8.

Disciplina: Introdução à Computação e Cálculo Numérico (I.C.C.N.) Turma 387 – ANO 2003

No. Aluno	08/05/2003 Avaliação Teoria Módulo 1		09/10/2003 Reavaliação Teoria Módulo 1	
	Status	Nota	Status	Nota
02203339	M	2.00	M	5.50
00299090	M	5.00	M	-
02203338	M	10.00	M	-
02203332	M	0.00	M	3.50
02203329	M	5.00	M	-
#02202342	M	2.00	M	4.00
02203306	M	6.00	M	-
02203282	M	5.00	M	-
02203287	M	1.00	M	8.00
02202326	M	3.50	M	5.50
02201218	M	2.00	M	4.00
02203321	M	6.00	M	-
02203312	M	5.50	M	-
02202282	M	5.00	M	-
02203335	M	3.50	M	8.00
02203343	M	3.00	M	6.50
02203344	M	10.00	M	-
02203317	T	-	T	-
02203323	M	7.00	M	-
02203320	C	-	C	-
02203340	M	10.00	M	-
02200136	M	3.50	M	7.00
02203322	M	5.50	M	-
02203310	M	4.50	M	8.50

No. Aluno	08/05/2003 Avaliação Teoria Módulo 1		09/10/2003 Reavaliação Teoria Módulo 1	
	Status	Nota	Status	Nota
02203267	M	5.00	M	8.50
02203304	M	7.50	M	-
02203307	M	3.50	M	-
02203331	M	0.00	M	2.50
02203328	M	5.00	M	8.50
02203352	M	0.00	M	6.50
02202520	M	8.50	M	-
#02203305	M	3.50	M	2.00
#02201241	M	2.50	M	4.50
02203278	M	F	M	-
02203293	M	5.00	M	-
02203315	M	0.00	M	7.50
#02203297	M	3.50	M	4.50
02203342	M	5.50	M	-
*02203333	M	1.50	M	7.50
#02203341	M	0.00	M	1.00
02203299	M	10.00	M	-
02203295	M	6.00	M	-
02203298	M	2.50	M	4.50
02203265	M	10.00	M	-
02203216	M	F	M	-
*02203273	M	4.00	M	8.00
02203345	M	5.00	M	-
02203302	M	3.50	M	8.00

☐ Alunos participantes do mini-curso
\# Alunos que não atingiram a freqüência mínima no mini-curso
* Alunos que optaram por fazer reavaliação no final de junho de 2003
T Alunos com matrícula trancada
M Alunos matriculados
C Alunos com matrícula cancelada

Disciplina: Introdução à Computação e Cálculo Numérico (I.C.C.N.)
Turma 374 – ANO 2003

No. Aluno	08/05/2003 Avaliação Teoria Módulo 1		09/10/2003 Reavaliação Teoria Módulo 1	
	Status	Nota	Status	Nota
02203336	M	5.00	M	8.50
02202343	M	3.50	M	-
02203258	M	8.00	M	-
02202700	M	8.00	M	-
02203280	M	1.50	M	-
02203263	M	2.00	M	3.50
02203301	M	10.00	M	-
02203289	M	4.00	M	-
02201281	M	0.00	M	-
02203337	M	6.50	M	-
#02203325	M	0.00	M	0.50
02203256	M	1.50	M	5.00
02203288	M	3.50	M	-
02203253	M	4.00	M	5.50
02203284	M	9.50	M	-
02203296	M	7.00	M	-
02203300	M	8.00	M	-
#02203313	M	0.00	M	1.00
02203286	M	0.00	M	2.75
02203314	M	F	M	-
02203308	M	F	M	6.00
02203319	M	7.50	M	-
02203346	M	5.50	M	5.50
02202286	T	-	T	-
02203311	M	7.00	M	-

No. Aluno	08/05/2003 Avaliação Teoria Módulo 1		09/10/2003 Reavaliação Teoria Módulo 1	
	Status	Nota	Status	Nota
02203508	M	7.00	M	-
02203264	M	5.00	M	8.50
02203277	M	7.00	M	-
02203270	M	10.00	M	-
02203272	M	7.50	M	-
02203348	M	9.00	M	-
02203290	M	9.00	M	-
02203349	M	0.00	M	-
02203326	T	-	T	-
02203254	M	4.00	M	6.00
02201355	M	8.00	M	-
02203269	T	-	T	-
02203303	T	-	T	-
02203260	M	0.00	M	3.75
02203251	M	7.50	M	-
02203279	M	7.00	M	-
02203281	T	-	T	-
02203327	M	6.50	M	-
02203261	M	5.50	M	-
02203252	M	6.50	M	-
02203255	M	1.00	C	-
02203276	M	10.00	M	-
02203266	M	10.00	M	-
02202527	M	6.00	M	-
02203259	M	9.50	M	-

☐ Alunos participantes do mini-curso

\# Alunos que não atingiram a freqüência mínima no mini-curso

T Alunos com matrícula trancada

M Alunos matriculados

C Alunos com matrícula cancelada