Maria Vanderléa de Queiroz

Orientador: Ângelo de Moura Guimarães

Co-orientador: José Luis Braga

Estação OO: Um Ambiente Integrado de Desenvolvimento para Introdução à Programação Orientada a Objetos com Java

Dissertação apresentada ao Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Belo Horizonte

Agosto de 2005

Resumo

Este trabalho apresenta um estudo e análise de micromundos e ambientes integrados de desenvolvimento (IDEs didáticas e pedagógicas), verificando, na prática de sala de aula, como estes ambientes contribuem no processo de ensino/aprendizagem.

O foco principal do estudo é o de disciplinas introdutórias de programação, com enfoque em orientação por objetos na linguagem Java.

O trabalho investiga padrões pedagógicos, padrões de projeto e padrões de codificação a serem usados com aprendizes iniciantes a partir de sugestões existentes na literatura e estudos de caso para testar estas sugestões. O trabalho propõe um ambiente chamado "Estação OO", que utiliza a metáfora de um mapa ou "roteiro de viagem" para a aprendizagem de Orientação por Objetos, reunindo IDEs e padrões que foram considerados os mais promissores e adaptados a partir dos estudos realizados.

A proposta do trabalho da Estação OO levou em conta as dificuldades que os aprendizes demonstraram na utilização dos ambientes e das ferramentas estudadas. O projeto de uma IDE pedagógica adicional, apoiada em uma IDE didática já existente (BlueJ) e em idéias e procedimentos disponíveis na literatura, também é esboçado e sugerido para uso no ensino de programação de computadores.

Abstract

This work presents a study which analyzes microwords and the use of Integrated Development Environments (IDEs), driven by didacticism and pedagogy. In the context of real classrooms, the study tried to verify how these features could contribute in the teaching/learning process.

The main focus of the study is the introductory programming disciplines using the Java language and the object oriented programming approach. The work inquires on when pedagogical patterns, design patterns and coding patterns could be used by beginners and was based on suggestions from literature and some case studies, carried out to test the viability of these suggestions.

This work also proposes a didactic framework, called “Estação OO”, which make use of a metaphor for a "trip" in the form of a map or route to conduct the trainee in Object Orientation which gathers the most promising suggestions from studies and their adaptations available on literature. The proposal of Estação OO takes into account the apprentices difficulties demonstrated when they were using the tools studied. An additional IDE, based on BlueJ, incorporating ideas and procedures available on literature is outlined and suggested for use in introductory programming disciplines.

Dedico esta dissertação aos meus filhos Augusto e Pedro Henrique e em especial aos meus pais Ivone e Sebastião que muito contribuíram para que hoje eu pudesse estar lhes dedicando este trabalho.

Agradecimentos

A Deus, pela oportunidade de elaborar este trabalho e pela ajuda em concluir.

Ao DPI (Departamento de Informática da Universidade Federal de Viçosa) e ao DCC (Departamento de Ciência da Computação da Universidade Federal de Minas Gerais), que graças ao convênio entre as instituições permitiu a muitos graduados a possibilidade de alcançar o grau de mestre em Ciência da Computação.

Em especial ao professor Ângelo de Moura Guimarães, pela sua sabedoria na condução de todo este trabalho e pela tranqüilidade contagiante.

À minha família, que participou de todo este trabalho, encorajando e dividindo outras atividades.

Ao professor Henrique, coordenador do LBI, que me orientou com palavras de Deus. A todos os funcionários do LBI, pelo carinho e atenção e aos do DPI, sempre atenciosos e prontos a ajudar.

Ao professor Heleno, coordenador do curso de Sistemas de Informação da faculdade onde trabalho, por ter me apoiado sempre e reduzido minha carga de trabalho para dedicar-me a esta pesquisa.

Aos professores do curso de pós-graduação da UFV, pela possibilidade de fazer este trabalho, em especial ao José Luis, que me fazia retomar os trabalhos quando o desânimo abatia.

Às minhas amigas unidas desde o ensino médio e agora trabalhando juntas na mesma instituição como professoras.

A todos os meus alunos, fonte de inquietação e inspiração.

A todos que contribuíram, de forma direta ou indireta, para a conclusão deste trabalho.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

1 Motivação e Objetivos

1.1 INTRODUÇÃO

O mundo está em constante evolução, criando desafios para indivíduos e organizações. Os indivíduos precisam aprender a lidar com as mudanças e as escolas e universidades precisam preparar as pessoas para este mundo. A aprendizagem efetiva é um tesouro inigualável e vitalício. O indivíduo que aprende é capaz de estar em sintonia com o mundo real e transpor os desafios que lhe são propostos [MOR00].

Mudança é a palavra de ordem na sociedade atual. Mudanças que implicam em profundas alterações em praticamente todos os segmentos de nossa sociedade, afetando a maneira como atuamos e pensamos. Na passagem para a sociedade do conhecimento, fatores tradicionais como matéria-prima, trabalho e capital assumem papel secundário e o conhecimento e, em conseqüência, os seus processos de aquisição assumem papel de destaque. Essa valorização do conhecimento demanda uma nova postura dos profissionais em geral e requer o repensar dos processos educacionais, principalmente aqueles que estão diretamente relacionados à formação de profissionais e com os processos de aprendizagem [VAL02a]. A sociedade do conhecimento requer indivíduos criativos e com a capacidade de criticar construtivamente, pensar, aprender sobre como aprender, trabalhar em grupo e conhecer seus próprios potenciais, resolver problemas como equilíbrio ecológico e social e em domínios específicos. Isso requer um indivíduo atento às mudanças e que tenha capacidade de constantemente melhorar e reciclar suas idéias e ações [VAL02b].

Devemos repensar nosso sistema educacional. É imprescindível enfatizar a aprendizagem como principal objetivo do sistema e a integração entre trabalho e aprendizagem como necessidade e não como opção [REP96].

Para propiciar a formação de indivíduos aptos para esta nova sociedade é preciso desenvolver ambientes de aprendizagem em que o aprendiz vivencie as novas competências exigidas. Muitas delas não são passíveis de serem transmitidas, devem ser construídas e desenvolvidas pelo próprio indivíduo [PAP85, VAL02a].

Em um ambiente de aprendizagem eficaz é preciso que o aprendiz tenha a possibilidade de "manusear" e trabalhar com os objetos e nesta interação criar, testar, depurar e ser desafiado, a fim de alcançar um nível onde a compreensão dos conceitos seja natural. Não é o simples fazer, o chegar a uma resposta que dever ser o objetivo, mas sim, a interação com o que está sendo feito, de modo a permitir aprendizagem de conceitos através do fazer e refazer, corrigindo os erros e adquirindo conhecimento no processo. Assim, os ambientes de aprendizagem devem ser ricos em oportunidades, que permitam aos alunos a exploração e aos professores a possibilidade de aberturas para desafiar o aluno e aumentar a qualidade da interação do que está sendo feito.

Nessa linha de raciocínio e com o desenvolvimento e a disponibilidade de novos recursos tecnológicos, temos hoje ferramentas capazes de facilitar a aprendizagem de programação de computadores, como LOGO [PAP85], Karel [PAT81], LOGO MicroWorlds [LCS00], Jarel [MCD00], PascaL [GUI00], Karel J. Robot [BER00a] entre outras. Na Ciência da Computação, como seria de se esperar, o ensino e o apoio tecnológico ao ensino e à aprendizagem são tópicos de pesquisa em expansão e têm gerado muitas discussões, reflexões e ferramentas [KOL95, WIS96, BUC00, BUC01, BER02d, FIN03, HEN04, BRU04].

O aprendizado baseado na resolução de problemas ou na elaboração de projetos não é novo e já tem sido explorado pelos meios tradicionais de ensino [SOL96, VAL02a]. A utilização do computador adiciona uma nova dimensão: o aprendiz tem que expressar a resolução do problema segundo uma linguagem de programação. Isto traz vantagens para a aprendizagem: a descrição da solução precisa ser feita de forma precisa e formal e a verificação da solução pode ser feita através da execução do programa. Com isto o aluno pode verificar suas idéias e conceitos, analisar o programa e identificar erros, fazendo sua depuração. Tanto a representação da solução do problema quanto sua depuração são tarefas difíceis através dos meios tradicionais de ensino [VAL93a,VAL95].

O produto final é a solução do problema. A linguagem de representação da solução do problema, a linguagem de programação, deve ser bem escolhida pois influencia os processos de pensamento uma vez que é o veículo de expressão de idéias. Porém, como meio de representação, o processo de aquisição do conhecimento relativo à linguagem de programação deve ser o mais transparente e o menos problemático possível [VAL96]. Nesse sentido, a programação pode ser vista como uma janela para a mente [VAL95]. No entanto, a maioria das linguagens de programação permite a produção de programas que passam a ser "janelas sujas", com sintaxes complexas e demanda por técnicas sofisticadas [BAR02].

Além de janelas limpas, é interessante considerar a idéia de incluir conceitos da teoria da computação como aspectos da formação geral de um indivíduo. Considerando-se o crescimento da automação e as revoluções da informática, conceitos da teoria da computação passam a ser de grande valor como ferramenta auxiliar na resolução de problemas do dia a dia. Ferramentas como autômatos, máquina de Turing, programação de computadores, programação concorrente, podem ser utilizados em diversos níveis na formação geral do indivíduo, permitindo que o conhecimento o auxilie na resolução de problemas cotidianos [REI03]. Para escrever programas, é preciso fazer a transição de um entendimento intuitivo para a especificação formal do algoritmo. Esta é uma das idéias centrais da Ciência da Computação e parte da educação em ciência da computação [REI01]. No ensino de programação é preciso observar que o paradigma central é o da abstração. A facilidade de expressar uma dada abstração é uma característica da linguagem utilizada. Assim, a facilidade de expressão de procedimentos, iterações, dados e tipos, além da facilidade de definição de módulos que favoreçam o encapsulamento, são fatores a serem considerados na escolha de uma linguagem de programação.

Outro grande desafio para o ensino de programação em um ambiente de sala de aula é a heterogeneidade. Estilos diferentes de aprendizagem em um mesmo grupo devem ser usados como motivação e não como um fator de complicação. É preciso que todos os estilos de aprendizagem sejam contemplados. Apesar de muitas vezes o professor ter de lidar com turmas com grande número de alunos, a individualidade precisa ser respeitada [ENT83]. Com este objetivo, o professor pode lançar mão de diferentes ferramentas para alcançar os diferentes estilos através da disponibilização e utilização de ferramentas.

1.2 MOTIVAÇÃO

Linguagens de programação como Java não podem ser vistas apenas como uma coleção de características, algumas das características não fazem sentido isoladamente. Para utilizar a soma das partes é preciso pensar no problema como um todo, isto é, no projeto e não somente no código. Um determinado conjunto de características pode ser abordado, em cada etapa de um curso, considerando-se a expressão de uma solução de algum problema específico usando a linguagem [ECK01]. Assim, pode-se conduzir a abordagem à linguagem de forma incremental e aprofundando-se à medida que se avança nos detalhes do projeto em desenvolvimento.

Neste cenário, surge a necessidade de se repensar a prática pedagógica em disciplinas de introdução à programação. Quais são e quais devem ser seus objetivos? Será que a maneira como a programação é introduzida aos alunos é a mais adequada? Será que os alunos realmente estão aprendendo programação de computadores? O que significa "programar" um computador nos dias de hoje?

Hoje, a informática na educação insere o computador no processo ensino-aprendizagem de conteúdos curriculares de todos os níveis e modalidades de educação. A informática que realmente interessa na educação é aquela na qual o professor da disciplina curricular tenha conhecimento sobre os potenciais educacionais do computador e alterne adequadamente atividades de ensino-aprendizagem e atividades que usam o computador. O uso do computador na criação de ambientes de aprendizagem que enfatizem a construção do conhecimento apresenta enormes desafios. Em primeiro lugar requer o entendimento do computador como uma nova maneira de representar o conhecimento. Requer uma análise cuidadosa do que significa ensinar e aprender, assim como demanda a revisão do papel do professor nesse contexto. Em segundo lugar, a formação desse professor envolve muito mais que provê-lo com conhecimento sobre computadores. Ele passa a ser um mediador, o elemento que proporciona uma abordagem integradora de conteúdos, voltada para a resolução de problemas específicos do interesse de cada aluno, compatibilizando as necessidades dos alunos com os objetivos pedagógicos que se dispõe a atingir [VAL02c].

Para aprender não é suficiente a transmissão da informação de alguém que sabe para alguém que não sabe. Muito mais que isso, aprender é um processo ativo conduzido pelo aprendiz, onde o conhecimento e entendimento são construídos por ele próprio. Além disso, aprender é um processo social, depende da interação. A aprendizagem é mediada pela construção de artefatos externos, onde a construção de artefatos leva à construção do entendimento [SOL96].

Uma questão muito importante relativa à aprendizagem foi estudada por Novak e Gowin [BRA87], que apontam o fato dos professores virem trabalhando arduamente para conseguir o que é ao mesmo tempo impraticável e cansativo. Eles procuram conduzir o processo de aprendizagem nos estudantes, quando, na realidade, este processo deve ser conduzido pelo próprio estudante. Segundo Perry [ENT83], o único papel de ensinar é facilitar a aprendizagem e, quando o professor entender mais a respeito de como diferentes estudantes aprendem, poderá ajudá-los a aprender melhor [BRA87].

Fazendo uma analogia do ensino de programação com o ensino de matemática, Valente [VAL93b] tenta responder à pergunta: "Por que muitos alunos fracassam ao aprender Matemática?":

> *"O processo de fazer matemática, ou seja, pensar, raciocinar, é fruto da imaginação, intuição, "chutes" sensatos, tentativa e erro, uso de analogias, enganos e incertezas. A organização da confusão significa que o matemático desenvolveu uma seqüência lógica, passível de ser comunicada ou colocada no papel. No entanto, o que o aluno faz quando faz matemática é muito diferente do processo de organização mental. Ao contrário, o fato matemático é passado ao aluno como algo consumado, pronto, que ele deve memorizar e ser capaz de aplicar em outras situações que encontrar na vida. Como isso nem sempre acontece, o aluno fracassa e, portanto, é responsável pelo fracasso da matemática. E essa culpa é somente do aluno. Não é da matemática, pois, mesmo sendo muito difícil, ela tem que ser passada ao aluno. Não existe outra maneira. Nem é do professor, já que este se esmera o máximo possível em passar o conceito matemático, adota a melhor didática possível, uma aula magnífica, tudo perfeito. Portanto, se o aluno não consegue aplicar o conceito já visto na resolução de um problema, então a culpa é do aluno. Entretanto, as razões pelas quais o aluno fracassa são diversas. Primeira, o fato de o aluno não ter construído o conceito, mas esse ter sido passado ao aluno. Segundo, mesmo que houvesse apropriação do conceito num determinado contexto, a aplicação desse conceito em outro contexto deve ser encarada como uma outra questão. A transferência de conhecimento não ocorre automaticamente. Enquanto o conceito é frágil, ele deve ser reconstruído no outro contexto ao invés de simplesmente reaplicado. Terceiro, o fato de o aluno não ter chance de adquirir o conceito matemático está relacionado também com a própria matemática. Os conceitos matemáticos são complicados, a notação matemática se tornou complexa, dificultando o pensamento matemático e o exercício do raciocínio."* [VAL93b p. 9]

Será que com o ensino de linguagens de programação não estaria acontecendo o mesmo? Fica a pergunta sobre como propiciar este tipo de formação. Como criar um ambiente onde o aluno possa adquirir as habilidades necessárias para atuar na nova sociedade? A programação de computadores tem um papel importante neste contexto já que computadores são máquinas universais que farão o que desejarmos, se formos capazes de programá-los.

A linguagem utilizada para representação da solução do problema deve ser bem escolhida pois o poder de abstração e as idiossincrasias da linguagem interferem no processo de representação. No entanto, é importante notar que o objetivo não é o de ensinar programação de computadores em si e sim como representar a solução de um problema segundo uma linguagem computacional. O produto final pode ser o mesmo - o programa de computador –, porém os meios são diferentes [BAR02].

Java é uma linguagem que tem despertado muito interesse entre os educadores desde sua introdução [RIC03], e seus méritos como uma linguagem de programação introdutória tem sido relatada em vários artigos [BER96, BER98, TYM98, KOL01]. Além destes relatos, experiências práticas com a adoção de Java em seqüências introdutórias com muito sucesso também têm sido reportadas [RIC03, DEC03].

Outro ponto a favor de Java é ser uma linguagem possível de ser usada em várias disciplinas e tópicos em cursos de Ciência da Computação como Introdução à Programação, Engenharia de Software, Computação Paralela e Concorrente, Redes de Computadores, Banco de Dados entre outros [BER98].

Ainda hoje, somente um pequeno número de pessoas é capaz de programar [PAU00], apesar de existirem muitas tentativas de desenvolver linguagens para os usuários finais. Segundo Repenning [REP96], programar é difícil. Sempre existirá uma complexidade intrínseca ao problema a ser resolvido, além da complexidade acidental da programação, oriunda da falta de entrosamento entre pessoas, ferramentas e problemas.

Dentro do âmbito das universidades, têm sido oferecidos cursos introdutórios de programação a praticamente todas as carreiras. Existe uma verdadeira necessidade de aprendizagem e um verdadeiro interesse em computação. Entretanto, muitos alunos desistem antes de terminar a disciplina ou expressam pouco interesse, o que sem dúvida é uma contradição. Estamos tendo um fracasso no ensino de programação [ROC95]. Para os mais acomodados, resta dizer que *programação não é para qualquer um*. Mas, para uma razoável comunidade científica, surgem questionamentos sobre como ensinar programação. Há, portanto, uma necessidade emergente de pesquisas para entender o processo de aprender a programar, detectando falhas e dificuldades deste aprendizado e sugerindo alternativas, para facilitar o aprendizado de programação.

Segundo Guzdial [GUZ02], educadores que ensinam programação de computadores, trabalham com uma visão desatualizada de computadores e alunos. Ensinam computação da mesma forma que aprenderam. O cenário mudou, mas a maioria dos cursos de introdução à programação não.

Um dos frutos de maior sucesso do casamento da informática e da pedagogia no ensino e aprendizagem da programação de computadores é a metodologia LOGO [PAP85, PAP85, VAL02a, BAR02]. Um ambiente de aprendizagem como o LOGO permite que o estudante possa explorar alternativas, testar hipóteses e descobrir fatos que são verdadeiros sobre uma parte do mundo real. Neste ambiente, o aluno é encorajado a construir seu próprio conhecimento e a pensar sobre as principais características deste pequeno mundo através de uma simulação metafórica de um mundo específico. Papert [PAP85] chamou de "micromundo" este ambiente metafórico. Utilizando um micromundo da Geometria, LOGO utiliza esta metáfora e nela o aluno pode adquirir importantes conceitos de programação, que

poderão ser utilizados como elementos facilitadores no aprendizado de outros conceitos e de outras linguagens de programação, como Java, por exemplo.

A idéia de utilizar micromundos para se ter contato com determinados aspectos de programação foi introduzida por Seymour Papert na década de 60 [PAP85] por meio de uma metáfora de ensinar uma Tartaruga a se deslocar em um mundo novo (o micromundo da geometria). A linguagem LOGO é interpretada e é o meio pelo qual o "programador" se comunica com a tartaruga. Apesar de mais lenta, a linguagem interpretada tem a vantagem de permitir a obtenção de uma resposta imediata, o que possibilita uma reflexão imediata do resultado da ação (comando). Este ambiente permite ao "programador" ir progressivamente construindo seu conhecimento, testando, depurando, alterando e reconstruindo novamente em níveis crescentes de "maturidade".

Karel [PAT81], criado por Richard Pattis, é outro exemplo de micromundo utilizado no contexto de aprendizagem. A metáfora utilizada é a de um robô que vive em um mundo cartesiano de ruas e avenidas. O mundo contém objetos especiais que Karel pode sentir e manipular. A linguagem de programação utilizada é parecida com o Pascal. Os estudantes não precisam de um conhecimento profundo da sintaxe, o próprio sistema se responsabiliza pela sintaxe enquanto libera o aluno para fixar a atenção em seleções, repetições e procedimentos.

Jarel é uma implementação Java baseada em Karel feita por Charlie McDowell [MCD00]. A idéia foi criar uma classe Java que permitisse aos estudantes escreverem código puro em Java, que fossem tão simples quanto programas para o robô Karel. O ambiente de desenvolvimento para programas Jarel é um Ambiente de Desenvolvimento Integrado (Integrated Development Environment - IDE no acronismo em inglês) com finalidades didáticas, que permite ao aluno editar o programa em Jarel, compilá-lo e executá-lo. Apesar de JAREL utilizar a linguagem JAVA, o ambiente não explora as características de orientação a objetos de Java.

O paradigma de orientação a objetos é um paradigma de desenvolvimento de software relativamente novo, tendo seu ensino e adoção crescidos consideravelmente nos últimos anos. Em conseqüência, diversas questões têm sido levantadas sobre como melhor ensinar e aplicar o paradigma, a fim de obter proveito real de seus benefícios, tais como representação do mundo real de forma mais simples, reuso, facilidade de manutenção [WIR90, RUMB94].

Karel J. Robot é outro micromundo descendente de Karel, desenvolvido por Joseph Bergin [BER00a], onde é feita a experimentação de conceitos fundamentais de Orientação a

Objetos, utilizando uma linguagem especial dos seus habitantes (robôs) e a linguagem Java. A orientação a Objetos é tratada em suas idéias genuínas e não como uma extensão do paradigma de programação estruturada. Karel J. Robot é mais que uma ferramenta, é uma nova forma de abordar o tema, preocupando-se em conduzir o aluno no pensamento orientado a objetos para resolução de problemas e não no aprendizado de detalhes da linguagem Java. Conceitos chaves, como polimorfismo, herança e interfaces, são expostos nos primeiros contatos através de desafios que devem ser resolvidos no micromundo e conduzem à essência da Orientação a Objetos o mais depressa possível, antes mesmo de estruturas de controle como seleção ou repetição. Um diferencial em seu trabalho é a utilização de padrões pedagógicos na abordagem dos tópicos e nas atividades envolvidas no processo de ensino/aprendizagem. Outro elemento na resolução dos problemas propostos são os padrões de projeto (*design patterns*) e de codificação para iniciantes [BER02]. Estes padrões serão estudados no Capítulo 4.

BlueJ [KOL99] é também uma IDE com finalidades didáticas para o ensino de Orientação a Objetos usando Java. Trata-se de uma ferramenta introdutória, sendo seu maior benefício o de permitir o óbvio, introduzir programação orientada a objetos usando objetos. O aluno tem o primeiro contato com objetos e não com sintaxes e detalhes de uma linguagem de programação. BlueJ descomplica o ambiente e permite o acesso às idéias mais importantes primeiro. O aluno pode interagir com classes em uma primeira aula, gerar objetos e solicitar-lhes serviços sem a necessidade de se preocupar com a elaboração de programas. A introdução pode ser feita com classes prontas, onde o aluno pela sua manipulação vivencia conceitos como classes, objetos, identificação, instanciação, solicitação de serviços, parametrização, interação entre objetos, relacionamento entre classes e muitos outros. Os padrões pedagógicos também são aplicados e o ambiente facilita a investigação e a descoberta de conceitos. Diagrama de classes simplificados permitem visualizar os relacionamentos entre as classes e facilitar o entendimento. Os diagramas de objetos são utilizados ao se inspecionar os objetos criados e disponibilizados em sua bancada de objetos. Outra ferramenta muito importante dentro do ambiente é o depurador, fácil de utilizar e que permite o entendimento do processo de execução do programa.

Além do LOGO MicroWorlds, AllKara, Karel J. Robot, BlueJ, outras abordagens do mesmo tipo, como Alice, Jurtle, Dr.Java e Grennfoot serão apresentados e abordados no

Capítulo 3, pois tratam de micromundos e ferramentas que foram estudadas para subsidiar este trabalho.

1.3 DIFICULDADES COM O MEIO DE APRENDIZAGEM

Considerando-se que a programação de computadores é uma disciplina que requer capacidade de expressão e de organização mental e que o primeiro contado com novos assuntos deixam impressões pelo resto da vida, surge a necessidade de se repensar o ensino/aprendizagem de introdução à programação de computadores, visando facilitar o processo de programação por meio do uso de novas técnicas e ferramentas disponíveis.

Como as crianças começam a falar? Como começam a andar? Elas passam por um processo inicial teórico de como se deve falar, ou como se faz para engatinhar, depois ficar de pé e depois dar um passo e depois dar outros passos? Diariamente a criança tem contato com pessoas que falam, caminham, desenvolvem atividades entre si e assim vão aprendendo, tanto pelo meio como pela dedicação e o bom acompanhamento das pessoas envolvidas em seu processo de desenvolvimento. Desta forma, a linguagem necessária para a comunicação é algo transmitido de forma natural, onde a aquisição de novos conhecimentos é solicitada pela própria criança ou motivada por quem a acompanha. As aptidões naturais e o conhecimento individual (da criança) são considerados e assim ela cresce e aprende a se comunicar com o mundo onde vive.

E quando se trata de aprender uma nova linguagem, a linguagem de comunicação com o computador, como este processo é conduzido? Na maioria das vezes, cursos de introdução à programação fazem uma introdução à linguagem de programação, isto é, apresentam aos alunos as estruturas existentes na linguagem de forma descontextualizada. O aluno aprende as estruturas, vê exemplos de aplicação e assim o processo continua. Na maioria dos casos este conteúdo não possui significado para o aluno, ele não é incorporado ao seu corpo de conhecimento e assim não é absorvido e conseqüentemente não é aprendido.

O próprio paradigma de orientação por objetos pode nos oferecer uma solução para este problema na medida em que ele trata a complexidade inerente à programação e propõe que um programa em funcionamento seja um conjunto de objetos, previamente definidos em classes, que, na interação destes objetos, possa resolver problemas. Quando o aluno tem a possibilidade de manipular objetos dentro desta dinâmica ou de representar o papel de um objeto, ele pode usar ou vivenciar a orientação a objetos e assim começar a entender todo o processo.

Assim, o caminho proposto para a introdução à programação é o de usar um micromundo, existente ou criado a partir de um jogo inventado pelo instrutor, entendê-lo, utilizar os conceitos fundamentais do paradigma e depois passar por etapas que vão aos poucos mostrando a tradução das idéias em diagramas e dos diagramas em programas.

1.4 DIFICULDADES COM AS FERRAMENTAS EXISTENTES

Por um lado ferramentas como editores, compiladores, depuradores e ambientes de execução muitas vezes são difíceis de entender e manipular. Por outro lado, já existem dezenas de ambientes de programação que facilitam a utilização destas ferramentas, permitindo uma abordagem inicial menos aprofundada com posterior aprofundamento, a partir do momento que o aluno alcance uma maior maturidade no assunto, e que permita descer a níveis mais detalhados sem gerar grandes impactos.

Conceitos fundamentais podem ser muito abstratos se considerados de forma descontextualizada. Ao utilizar o quadro negro, livros textos e ambientes de programação difíceis de serem manipulados por iniciantes, o professor pode estar dificultando o entendimento para o aluno.

1.5 FOCO PRINCIPAL NA LINGUAGEM DE PROGRAMAÇÃO E NÃO NO PROBLEMA

Muitos cursos de Programação Orientada a Objetos seguem a forma tradicional de apresentação de conteúdos, apresentando a linguagem em si, como nos cursos tradicionais de programação estruturada. Porém, esta forma de exposição parte do meio onde a solução será expressa para chegar ao problema que deve ser resolvido. A rota inversa, ou seja, onde se parte do problema, criando uma representação dos elementos que o compõem, entendendo-o e resolvendo e posteriormente representando-o em uma linguagem de programação, pode ser mais produtivo.

Um dos maiores problemas, que vem como herança de experiências anteriores com a disciplina de programação, é a tendência à dissecação das linguagens. Em muitos casos elementos da linguagem são apresentados de forma independente de contexto e este tipo de exposição descontextualizada pode ser apenas mais um item, sem significado imediato, que poderá não ser compreendido. Estas tendências ficam explicitadas em roteiros como o apresentado por Lozano & Galvão [LOS05] para o aprendizado da Tecnologia Java. A Figura 1-1 mostra este roteiro.

1.6 DESCRIÇÃO DO TRABALHO

Neste trabalho é proposto um ambiente de aprendizagem, Estação OO, que tira proveito de muitas das idéias viabilizadas por uma série de trabalhos disponíveis na literatura, alguns deles já citados aqui e outros que serão detalhados no Capítulo 3.

Estação OO é uma tentativa de reunir um conjunto de ferramentas que possibilitem ao aluno construir seu conhecimento sobre orientação a objetos utilizando algumas técnicas já testadas e aprovadas. Estação OO deve permitir ao aprendiz colocar-se como elemento ativo no processo de aprendizagem, através da realização de atividades dos mais variados tipos, desde aquelas com as quais se identifica até as que o deixam pouco a vontade, isto é, baseando-se na sua potencialidade e permitindo experimentar novas técnicas e formas de aprender e conduzir a construção do seu próprio conhecimento.

Como a aprendizagem deve ser causada pelo aluno, este ambiente funciona como um apoio aos alunos para desenvolverem atividades individuais, que mais se adaptem ao seu estilo de aprendizagem e permitam a conexão do conhecimento prévio com os novos conceitos abordados.

Não existe consenso quanto à adoção do paradigma de orientação a objetos para cursos introdutórios [ACM01]. Alguns defendem que a orientação a objetos é muito complexa para ser tratada em cursos introdutórios, outros que a programação estruturada dificulta e influencia o entendimento da orientação a objetos. Em Estação OO toda a abordagem é feita sob a ótica da orientação por objetos e não supõe que exista conhecimento prévio de programação estruturada. No Capítulo 2 é feita uma defesa deste ponto de vista.

Estação OO adota também a proposta [BER01a, BER02d, DEC03, BRU04] da antecipação da utilização de padrões de projeto, que usualmente são ensinados em cursos avançados de OO. Um padrão (pattern) descreve um problema que ocorre e recorre em nosso ambiente, e então descreve a essência da solução do problema, de tal forma que esta solução possa ser usada inúmeras vezes, sem ser necessariamente da mesma forma [BER01a]. O contato com padrões permitirá que o aluno se concentre em bons projetos orientados a objetos e fornecem um bom exemplo a ser seguido [LEW04]. Em Estação OO estes padrões são utilizados na solução de problemas propostos. Contudo, o aluno não tem uma formalização dos padrões de projetos. As soluções relatadas nos padrões de projetos são utilizadas nos problemas propostos e a aplicação da solução em outras situações é mostrada. Os padrões de projeto serão descritos no Capítulo 4.

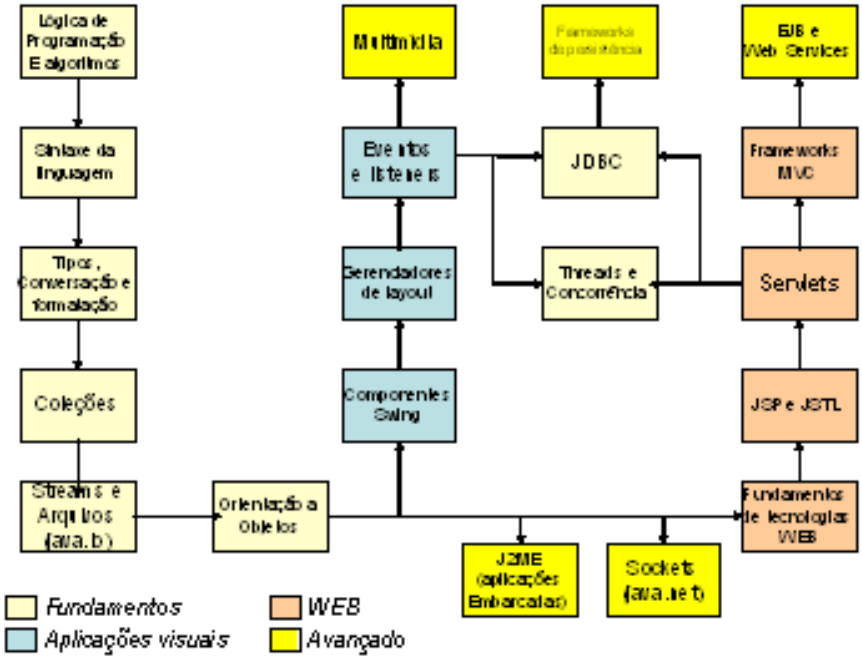

Figura 1-1 Elementos da Linguagem Java [LOZ05]

Em Estação OO é também adotada uma pedagogia baseada nos padrões pedagógicos que serão descritos no Capítulo 4. Estes consideram anos de experiências de profissionais envolvidos na educação e principalmente em cursos de programação de computadores [BER02]. Estas "boas idéias" pedagógicas sugerem que a abordagem dos conceitos no curso seja feita em espiral, com a utilização de diferentes atividades e recursos para apresentar os tópicos, sempre conectando o que o aluno já aprendeu com o novo conceito e outras técnicas que também serão apresentadas e discutidas no Capítulo 4.

Com a abordagem de *object-first* proposta no currículo pela ACM em 2001 [ACM01], supõe-se que os alunos pensem mais sobre o problema a ser resolvido e menos na sintaxe da linguagem de programação a ser utilizada. Porém, entre a declaração do problema e sua solução usando objetos, existe uma lacuna. Baseados em anos de experiências, os autores citados em [TAB04] acreditam que os estudantes podem usar a UML (Unified Modeling Language) [FIL01] para fazer a ponte entre o problema e a solução através de objetos. A abordagem "*problem solving first"* [ACM01] dá suporte à *"object-first"* [ACM01] através da utilização da UML, sem serem dominadas pela sintaxe de linguagens de programação. Aplicações de UML em cursos de nível introdutório (antes da codificação), provêm aos estudantes oportunidades mais efetivas e fáceis de entendimento das soluções [LAR98].

A introdução à programação é tratada de forma contextualizada, utilizando técnicas que levam o aluno à atividade e construção do seu conhecimento, usando a UML para descrever os conceitos envolvidos (diagramas de classes e diagramas de objetos) e finalizando com o programa, que é a descrição dos conceitos trabalhados na linguagem de programação Java.

Assim, os exemplos utilizados em Estação OO são elaborados considerando-se:

- O contexto, envolvendo uma modelagem conceitual informal do problema com descrição das classes em português utilizando o jogo de papéis (um dos padrões pedagógicos adotado), com uma posterior modelagem em UML, permitindo começar com uma vivência prática e efetiva dos conceitos e passar para um modelo formal;
- A experiência da interação entre objetos, reforçando os conceitos.
- A verificação dos conceitos em um ambiente orientado a objetos no computador, que permite a utilização e a abordagem orientada por objetos em ferramentas como o BlueJ e Karel J. Robot, por exemplo.
- Os padrões de projeto (*design patterns*) que são "boas idéias" de projetos, testados, depurados e aplicáveis a uma grande variedade de situações e, além disso, portadores de conceitos fundamentais da orientação por objetos.
- A verificação da aplicabilidade dos padrões de projeto em outros contextos e assim a abstração de soluções, que podem ser reaproveitadas.

Além das IDEs (acrônimo de **Integrated Development Environment)** convencionais, que visam a produtividade do programador no desenvolvimento de software [KOL99] Estação OO é o resultado da análise do que podemos chamar de "IDEs Didáticas", no sentido de que elas tentam favorecer a aprendizagem da programação. Estes aspectos são discutidos no Capítulo 2.

Para se chegar à Estação OO como uma IDE didática, foram feitas avaliações somativas com classes fechadas de alunos de disciplinas de Programação Orientada a Objetos com Java. Estas avaliações estão relatadas no Capítulo 5.

1.7 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é, portanto, propor um ambiente integrado de introdução à Programação Orientada a Objetos usando Java com uma abordagem focada em objetos (*object-first*) e no modelo conceitual do problema (*model-first*). Com esta proposta, parte-se

da linguagem do problema (estudo de situações práticas) e, através de atividades sugeridas pelos padrões pedagógicos, e com as idéias dos padrões de projeto, caminha-se na direção do programa e possíveis soluções do problema.

Neste ambiente não se faz a dissecação da linguagem de programação, pelo contrário, a linguagem é apresentada à medida que as atividades são desenvolvidas e crescem em complexidade.

As atividades desenvolvidas são propostas por meio de projetos como os presentes no BlueJ, prontos ou parcialmente prontos. Com a manipulação e vivência dos elementos do projeto, o aluno faz sua exploração e interage com os objetos. A partir desta interação, necessidades de alterações podem ser propostas tanto pelos alunos como pelo instrutor.

À medida que a exploração acontece, a complexidade dos projetos aumenta, como acontece naturalmente enquanto o ser humano cresce. E com este crescimento individual, os conceitos de orientação a objetos são explorados e gradativamente aprofundados com o objetivo de propor novos problemas e permitir o estudo e adequação de soluções.

Capítulo 2

2 Paradigmas de Programação e Seus Ambientes

> "O *aspecto mais importante, mas também o mais elusivo, de qualquer ferramenta é sua influência nos hábitos daqueles que se treinam no seu uso. Se a ferramenta é uma linguagem de programação essa influência é, gostemos ou não, uma influência em nosso hábito de pensar*", Edsger W. Dijkstra citado por Varejão [VAR04].

2.1 LINGUAGENS DE PROGRAMAÇÃO E OS PARADIGMAS

A escolha da linguagem de programação para uma determinada aplicação, principalmente no caso de contextos educacionais, requer atenção especial no paradigma subjacente à linguagem. Um paradigma de programação é um modelo básico para abordar a maneira de escrever um programa de computador [TEL89]. Segundo Papert [PAP85], um paradigma de programação é como um "quadro estruturador" subjacente à atividade de programar e a escolha do paradigma de programação pode mudar notavelmente *"a maneira como o programador pensa sobre a tarefa de programar"*.

A análise da evolução dos computadores, do surgimento das linguagens de programação e da utilização dos computadores, permite um entendimento de diferentes paradigmas de programação e a seleção para fins de adoção, tanto em ambientes de aprendizagem como em profissionais.

As primeiras linguagens de programação eram muito simples, em virtude da limitação física dos computadores e da pouca maturidade da ciência da computação na época do surgimento dos primeiros computadores. Estas linguagens disponibilizavam um pequeno conjunto de instruções que permitiam a realização de ações elementares e de uso exclusivo de um computador específico. Assim, essas linguagens ficaram conhecidas como **linguagens de máquinas** ou de **baixo nível**. Assim, ao final da década de 40 e início da década de 50, os computadores disponíveis, além dos problemas decorrentes da tecnologia, contavam com a ausência de software. A programação era feita em código de máquina. Este cenário motivou a criação das linguagens de montagem e seus montadores.

Além disso, as aplicações numéricas da época requeriam o uso de certas facilidades que não estavam incluídas no hardware das máquinas de então, surgindo assim a necessidade de criação de linguagens de mais alto nível que incluíssem tais recursos. Surge o **paradigma**

imperativo. O foco dos programas no paradigma imperativo é especificar como um processamento deve ser feito no computador. Os conceitos fundamentais são de variável, valor e atribuição. Este é o paradigma que mais se aproxima do uso da arquitetura de Von Neumann como modelo para representação da solução do problema a ser resolvido pela máquina [VAR04].

Com o avanço da computação e da tecnologia, as aplicações se tornavam cada vez mais complexas e as linguagens simples e específicas reduziam significativamente a produtividade do programador e dificultava a ampla disseminação dos computadores. Deste problema surgiram as **linguagens de programação de alto nível**, que se caracterizam por ter um conjunto amplo e expressivo de tipos de instruções e não serem específicas a um determinado computador.

O desenvolvimento de programas utilizando-se as linguagens de máquina, que se caracterizava por uma seqüência monolítica de comandos e pelo uso de desvios condicionais e incondicionais para determinar o fluxo de controle de execução do programa, estimulava a ocorrência de erros e diminuía a produtividade do programador. Para controlar este problema surgiu a programação estruturada e o **paradigma estruturado**, dentro do paradigma imperativo.

Este tipo de programação se baseia na idéia de desenvolvimento de programas por refinamentos sucessivos (top-down), onde o fluxo de controle de execução é organizado, desestimulando o uso de comandos de desvios incondicionais e incentivando a divisão dos programas em subprogramas e em blocos aninhados de comandos.

O programa representa a prescrição da solução do problema. As linguagens de programação procedurais (ou imperativas) como Fortran, Pascal, C, Modula-2, formam a maior classe de linguagens existentes até então. Isto se explica historicamente pelo fato do computador ser utilizado em aplicações numéricas e dentro do próprio domínio da ciência da computação [VAR04].

A lição a ser aprendida da programação estruturada é que a natureza da solução para um problema é um processo ou conjunto de ações, porém um programa é mais que processos. O programa consiste em processos e dados. Segundo Niklaus Wirth, o desenvolvedor de Pascal, Modula-2 e Oberon: Algoritmos + Estrutura de Dados = Programas [BER00c].

A programação estruturada tem o foco principal nos algoritmos e em seguida ajusta as estruturas de dados aos processos definidos. Esta abordagem torna a manutenção um

problema, uma vez que é difícil prever as mudanças e, quando estas ocorrem, é difícil minimizar seus efeitos nos subproblemas considerados [BER00c].

Um componente bastante típico de um curso de introdução à programação que usa o paradigma estruturado é uma discussão da arquitetura da máquina de von Neumann. Os estudantes vêem a RAM como espaço para programas e dados. O programa transforma os dados a partir de um estado inicial em um estado final. A ênfase é na mudança de estado, leituras e atribuições. A RAM é monolítica. Dados são enviados à CPU para serem processados. Este modelo simples se adapta bem a este paradigma.

Considerando a evolução do hardware através do tempo, sua oferta com menor custo, possibilitando o acesso a uma parte considerável da população, nota-se uma adaptação das linguagens de programação. Inicialmente utilizadas por uma classe científica, com um elevado grau de conhecimento e aplicações restritas e em um segundo momento, com a nova demanda dos usuários, carente de ferramentas que proporcionem maior produtividade ao programador, pois os softwares mudaram de aspecto e exigem manutenções freqüentes. A complexidade contínua dos softwares leva a processos de modularização e abstração de dados. Neste ponto a capacidade do hardware não é mais um problema, o foco passa a ser a complexidade dos problemas a serem resolvidos no computador e a produtividade no desenvolvimento. A necessidade agora é que as ferramentas voltem-se para os problemas e não para as máquinas, que permitam ao programador maior produtividade. Emergem então as linguagens de programação orientadas a objetos.

O termo programação orientada a objetos tem suas raízes no conceito de objeto da linguagem Simula 67. Nesta linguagem, um programa é organizado na execução conjunta de uma coleção de objetos. Os objetos comuns que compartilham uma estrutura comum constituem uma classe, descrita no programa através de uma declaração de classe em comum [NYG86]. Outra importante contribuição, em 1972, foi dada por Alan Kay, através da primeira linguagem de programação orientada a objetos de sucesso, SmallTalk [ECK01].

Com o avanço da computação, os sistemas de softwares têm se tornado cada vez maiores e mais complexos. O paradigma orientado a objetos oferece conceitos que objetivam tornar mais rápido e confiável o desenvolvimento desses sistemas [VAR04].

A idéia básica do paradigma é imaginar que programas simulam o mundo real: um mundo povoado de objetos. Assim, linguagens baseadas nos conceitos de simulação do mundo real devem incluir um modelo de objetos que possam enviar e receber mensagens e reagir a mensagens recebidas. Esse conceito é baseado na idéia de que no mundo real

freqüentemente usamos objetos sem precisarmos conhecer como eles realmente funcionam. Assim, programação orientada a objetos fornece um ambiente onde múltiplos objetos podem coexistir e trocar mensagens entre si [BER00b].

O programa é visto como uma rede ou grafo de elementos interagindo entre si, ora desempenhando o papel de cliente, ora de servidor.

Por utilizar os conceitos do paradigma estruturado na especificação dos métodos, o paradigma de orientação por objetos pode ser considerado uma evolução do paradigma estruturado [VAR04]. Porém, como a concepção de programa é muito diferente nos dois paradigmas autores, como Joseph Bergin, discordam desta evolução [BER00c].

O paradigma de orientação a objetos não é apenas um modismo, uma nova terminologia, um nome diferente [ZAN96], possui o objetivo, e consegue obtê-lo em conjunto com outras tecnologias, de atenuar o que chamamos de "crise do software".

O esforço é no sentido de revolucionar a produtividade do programador. O principal argumento por trás do paradigma está na defesa da idéia de que programar não deve simplesmente resolver o problema do momento, mas resolver o problema do futuro. Produzir soluções mais gerais, onde a manutenção e reforma possam ser fácies de serem obtidas e feitas com economia, re-aproveitando todo o esforço já despendido. Da mesma forma, ao criar um novo programa, o paradigma propõe que investiguemos se já não existe uma solução que possa ser reaproveitada [GUI01].

Trata-se de uma nova forma de ver o computador. Computadores não são apenas máquinas, são ferramentas que ampliam nossas mentes, uma forma diferente de expressão [ECK03]. Ferramentas lembram máquinas, mas estas "novas" ferramentas se parecem menos com máquinas e mais com extensões de nossas mentes, assim como escrever, pintar, esculpir, criar animações. Programação orientada a objetos é parte deste movimento na direção de utilizar o computador como meio de expressão, facilitando os processos naturais. Programação tem sua conexão com a arte [HAM04].

2.2 PROGRAMAR COM DIFERENTES PARADIGMAS

Programar nos diferentes paradigmas significa representar, segundo modelos diferentes, a solução do problema a ser resolvido na máquina. Cada linguagem, que suporta determinado paradigma, representa um "meio" onde o problema é resolvido. Enquanto "meio de expressão" e de "comunicação" com a máquina, a linguagem, e indiretamente o seu paradigma, "moldam" a representação do problema a ser resolvido. Assim, na atividade de

programar, mudar de paradigma significa muito mais do que reconhecer as entidades sintáticas e semânticas da nova linguagem, o processo de pensamento também deve ser mudado, ajustando-se ao novo meio de representação do problema [BAR02].

O entendimento de tais modelos é fundamental no *design* de metodologias para desenvolvimento de programas em uma determinada linguagem; na criação de ambientes de aprendizado baseado no computador e também no ensino/aprendizado de linguagens de programação de modo geral.

Na literatura não existe consenso quanto ao paradigma a ser utilizado em curso de introdução à programação [ACM01]. Qual é a melhor abordagem inicial, paradigma imperativo ou paradigma de orientação por objetos? Veja abaixo a colocação da Força Tarefa que promoveu o desenvolvimento do Currículo de Computação 2001.

> *"Com interesse em promover a paz entre a guerra de facções, a Força Tarefa CC2001 opta por não recomendar qualquer abordagem particular. A verdade é que não se encontrou a estratégia ideal, e que toda abordagem tem suas forças e fraquezas. Dado o atual estado da arte nesta área, estamos convencidos que nenhuma abordagem que considere todos os pontos será satisfatória em todas as instituições. Uma vez que programas introdutórios diferem drasticamente em seus objetivos, estrutura, recursos, e audiência, precisamos de uma organização de estratégias que tenham sido validadas pela prática. Além disso, devemos encorajar membros de instituições e faculdades a continuar a experimentação nesta área. Dado um campo que muda tão rapidamente como ciência da computação, inovação pedagógica é necessária para o sucesso continuado."* [ACM01- p 22].

Neste currículo são sugeridos seis modelos para abordagens de tópicos em cursos introdutórios em ciência da computação: *imperative-first, object-first, functional-first, breadth-first, algorithms-first e hardware-first*. A prática do ensino e sua reflexão é que deve nortear a escolha. Segundo o mesmo relatório, qualquer abordagem pode levar ao sucesso, uma vez que os professores se empenham e acreditam no que fazem. O teste real é se o sucesso pode ser mantido quando a mesma abordagem que é adotada por um pode ser seguida por outros com o mesmo sucesso [ACM01].

Sob estas considerações, neste trabalho será adotado o paradigma de orientação por objetos com a temática em objetos. Estruturas do paradigma imperativo aparecem no nível da implementação. A introdução à programação orientada a objetos é conduzida dentro do contexto, ou seja, trabalha-se com projetos cujas soluções são expressas em UML [FIL01, FER03]. Estes diagramas simplificados em UML, que representam as soluções, devem ser baseados em padrões de projeto. Assim faz-se uma introdução prática do tópico, permitindo a construção de uma base sólida. A linguagem Java será utilizada para a implementação de código.

2.3 AMBIENTES DE PROGRAMAÇÃO E DE APRENDIZAGEM DE PROGRAMAÇÃO

Uma IDE (acrônimo de Integrated Development Environment) é um software de computador capaz de ajudar programadores a desenvolver software [WIK05]. Normalmente uma IDE consiste de um editor de código fonte, um compilador e/ou interpretador, ferramentas para construção automática, e (usualmente) um depurador. Às vezes possui também um sistema de controle de versão (VCS) e várias ferramentas para simplificar a construção de uma GUI (acrônimo de Graphical User Interface) de forma integrada.

As mais modernas costumam integrar um Navegador de Classes (Class Browser), um Inspetor de Objetos (Object Inspector) e um diagrama de Hierarquia de Classes. Embora existam IDEs que admitem múltiplas linguagens como o Eclipse [IBM04], as IDEs são tipicamente devotadas a alguma linguagem específica, como BlueJ, Java, Visual Basic ou Object Pascal (Delphi, Kilyx).

Basic foi a primeira linguagem a ser criada com um ambiente de desenvolvimento integrado (IDE). Simples por ser baseada em linha de comando e não contar com o aparato das interfaces gráficas, porém com a integração da edição, gerenciamento de arquivos, compilação, depuração e execução como modernas IDEs [WIK05].

Linguagens como Java apresentam certas características como a complexidade inerente à interface de linhas de comandos e o processo de I/O, que dificultam o ensino de conceitos de programação [REI04]. Nestes casos, a utilização de uma IDE pode facilitar o processo de aprendizagem. Considerando a linguagem Java, o ambiente escolhido para o ensino pode ter um profundo efeito nas habilidades do iniciante para aprender a linguagem [MUR03].

Basicamente podemos considerar que existem dois tipos de IDEs: profissionais e pedagógicas.

As IDEs profissionais são projetadas para auxiliar desenvolvedores de software na produção de código de melhor qualidade. Estas reduzem a incidência de erros de sintaxe através do uso de um editor de textos que analisa a sintaxe do programa editado, realçando-a. A maioria das IDEs suportam endentação automática do texto do programa de acordo com o alinhamento das estruturas do programa. Algumas IDEs realizam um “parsing” incremental, identificando erros gramaticais em qualquer parte do texto do programa. Além disso, simplificam os processos de compilação e execução através de botões específicos,

disponibilizando ainda um depurador que permite o acompanhamento passo a passo da execução do programa.

Todas estas características são úteis em cursos introdutórios de programação, porém todo este arsenal é muito complexo para iniciantes [REI04]. Assim, IDEs pedagógicas, como BlueJ e Dr. Java, têm sido desenvolvidas especificamente para preencher os requisitos de cursos de introdução a programação orientada a objetos.

Kolling [KOL98] cita características importantes em ambientes de aprendizagem, que devem ser:

- Descomplicados;
- Motivantes;
- Permitir acesso às idéias importantes primeiro;
- Motivar através do aspecto visual;
- Facilitar o aspecto operacional.

Em [REI04] são citados três requisitos especiais em ambientes de desenvolvimento para iniciantes:

- Ser simples e intuitivo;
- Prover mecanismos simples (como para executar e fazer a entrada e saída);
- Ser "leve", isto é, não exigir muito do hardware.

Como as IDEs profissionais suportam o desenvolvimento de software mais complexo, tem-se proposto a adequação de IDEs profissionais para o uso em sala de aula. Este é um exemplo do trabalho descrito em [REI04], onde se relata o desenvolvimento de um "Dr. Java plug-in" para o Eclipse.

Outras combinações importantes, no sentido de facilitar o uso de outras ferramentas, têm sido feitas, como a criação de projetos no BlueJ que permitem o uso do micromundo LOGO (da tartaruga) [BLU05a] e a utilização de estudos de caso como o Marine Biology Case Study (MBCS) em [BLU05b].

IDEs Pedagógicas serão descritas em maiores detalhes no Capítulo 3. As principais IDEs Pedagógicas serão estudadas para uma verificação da contribuição que trazem para a aprendizagem de OO.

Várias tentativas de aplicação de ferramentas de IDE pedagógicas foram feitas para uma melhor validação das propostas encontradas na literatura. Estes estudos estão relatados no Capítulo 5. Eles serviram de base para a proposta de Estação OO, que está descrita no Capítulo 6.

Capítulo 3

3 Ensino-Aprendizado de POO

3.1 ENSINO DE PROGRAMAÇÃO NO PARADIGMA OO

O ensino de programação nas universidades brasileiras tem sido quase sempre concentrado no paradigma imperativo, o que vem sendo questionado nos últimos anos [ROC95, MIL02, BER00c]. Será que ela é a melhor maneira de aprender a programar? Por outro lado, existem várias questões levantadas em relação ao ensino de orientação a objetos: Quando introduzir o paradigma de orientação a objetos? Quais conceitos se devem incluir em um primeiro curso e quais excluir, como apresentar os conceitos do paradigma para que sejam assimilados de maneira adequada? Quando ensinar a primeira linguagem de programação e técnicas de análise e projeto OO? Qual linguagem de programação utilizar? Dependendo da medida adotada em relação a estas questões, podem-se obter diferentes resultados em relação à adoção e ao ensino do paradigma como um todo. Escolhas inadequadas podem levar (e geralmente levam) a problemas de ensino-aprendizagem e, como conseqüência, à má adoção e utilização do paradigma. A Sociedade Brasileira de Computação recomenda um currículo de referência (ver em www.sbc.org.br) para os cursos de graduação, no país, segundo o qual se deve ensinar os princípios dos diversos paradigmas de construção de software, como o imperativo, lógico, funcional e orientado por objetos.Este currículo, utilizado hoje pela maioria dos cursos de qualidade no Brasil, é um bom ponto de partida para a formação ou reciclagem de profissionais de desenvolvimento de software.

Na literatura temos relatos de problemas e questões quanto ao ensino do paradigma OO. São eles: (1) problemas relacionados com o processo de ensino-aprendizagem, (2) problemas em relação à influência do paradigma estruturado, (3) problemas em linguagens de programação, ambientes de desenvolvimento de software orientado a objetos e metodologias de projeto de software orientado a objetos. É importante ressaltar que todos estes diversos tipos de problemas estão intimamente relacionados. Um estudo bastante aprofundado e detalhado sobre estes problemas está descrito em [ZAN96].

Vale ressaltar que não existe consenso nem na identificação dos problemas de ensino e adoção do paradigma, nem nas respostas e conclusões dos diversos autores em relação aos

mesmos. Aspectos considerados importantes e benéficos para alguns autores são considerados por outros como sendo prejudiciais ao processo como um todo, como por exemplo, o tratamento do paradigma de orientação por objetos como uma extensão do paradigma imperativo. Mesmo assim, os problemas detectados por diversos autores e discutidos neste trabalho servem como base para propostas de trabalhos, ambientes de apoio e mostram a motivação dos diversos autores para o tema e também para este trabalho.

Os problemas detectados refletem a necessidade de estudos e pesquisas que venham melhorar a qualidade do ensino e, conseqüentemente, da adoção e utilização do paradigma como um todo. Em um primeiro momento do ensino do paradigma OO, problemas, falhas e inconsistências foram detectadas no processo de ensino-aprendizagem do paradigma, e tendem a diminuir com uma maior experiência dos professores, a criação de novas linguagens e ambientes de apoio ao ensino e de metodologias de software que resolvam alguns dos problemas detectados [KOL03].

3.2 DIFICULDADES DOS ENFOQUES EM RELAÇÃO A POO

No relatório final da apresentação do currículo de computação 2001 [ACM01], o tópico programação é discutido. Programação deve ser ensinada como primeiro tópico ou existem outros que precisam de atenção em primeiro lugar? A conclusão no relatório é que o modelo *programming-first* continuará dominando no futuro próximo. O relatório discute a implementação de currículos do tipo *programming-first*, utilizando três paradigmas de programação: imperativo, funcional e orientação a objetos. O paradigma de orientação a objetos tem sido alvo de muito interesse nos últimos anos, como pode ser notado pelo surgimento de micromundos e ambientes integrados (que estamos chamando de IDEs pedagógicas), e muito interesse entre os professores quanto à adoção do paradigma de orientação a objetos em cursos introdutórios [BAR03, BER00c, ECK00, DEC03].

O paradigma de orientação a objetos (OO) tem sido amplamente aceito como um importante paradigma. Possui suporte para conceitos que são ensinados há muito tempo, tais como modularização e projeto de programas. Além disso, disponibiliza técnicas para abordar novos temas que foram inseridos nos currículos como programação em equipe, manutenção de grandes sistemas e reutilização de software. Assim, podemos ver que orientação a objetos é uma boa ferramenta para ensinar metodologias que são consideradas importantes. Porém, ensinar orientação a objetos permanece sendo difícil. Experiências relatadas sugerem que

alguns problemas poderiam ser evitados ensinando orientação a objetos como primeiro paradigma em cursos introdutórios [ZAN96].

A utilização do paradigma de orientação por objetos em cursos introdutórios não é suficiente, outros problemas persistem, como a inadequação de ambientes e linguagens. Java é uma linguagem que se mostra muito interessante para o ensino/aprendizagem, porém, ambientes como Java Development Kit (JDK) dificultam a abordagem do assunto de forma adequada. Os alunos precisam lidar com "pedaços" de código precocemente para que seu programa funcione, sem contar com os comandos que são emitidos diretamente ao sistema operacional, para configurar, compilar e executar programas [KOL99]. Reclamações com relação à interação com o usuário, especialmente com tratamento de entradas/saídas e exceções, são comuns [KOL99].

3.3 INTRODUÇÃO À PROGRAMAÇÃO E À UML

Um dos grandes problemas em programação orientada a objetos é a não utilização de todo seu potencial [BEN04].

As linguagens de programação têm três papéis a cumprir: instruir o computador, gerenciar a descrição do programa e expressar o modelo conceitual [Horstmann, citado em BEN04]. Um programa orientado a objetos é um modelo, e este modelo pode ser visto em diferentes níveis de detalhes, caracterizados por diferentes graus de formalidade. Um modelo conceitual informal descrevendo conceitos chave do domínio do problema e seus relacionamentos, um modelo de classe mais detalhado dando uma visão geral detalhada da solução e a implementação real em uma linguagem de programação orientada a objetos.

Orientação a objetos tem esta amarração conceitual, o que permite a integração do ponto de vista da análise, do projeto e da programação, o que é uma grande vantagem, pois facilita o entendimento de conceitos pelos alunos. Os conceitos fazem mais sentido do que quando se utiliza a programação como forma de instrução do computador.

Estas considerações levam a uma representação conhecida de três níveis de programação orientada a objetos como representada pelos três níveis de abstração para a interpretação de modelos de classes em UML: nível conceitual, nível de especificação e nível de implementação/codificação [BEN04].

Cursos baseados em orientação a objetos poderiam tirar proveito do paradigma como um todo e fazer a abordagem realmente orientada a objetos. Em [DEC03] existe um estudo sobre a abordagem *object-first* que verifica a prática desta abordagem sob dois aspectos:

imperativo e orientado a objetos. Este estudo confirma o que vários autores relatam: para ensinar efetivamente programação orientada a objetos, não é suficiente que objetos sejam a primeira abordagem, mas que a temática do curso seja centrada no conceito de objetos.

Neste experimento os alunos foram divididos em dois grupos [DEC03] para um curso de introdução à programação usando Java com foco em objetos durante todo o curso: CS1OE (com ênfase em objetos) e CS1OD (sem ênfase em objetos).

Em ambos os cursos um conjunto de tópicos foi utilizado, como nos cursos introdutórios de modo geral: tipos de dados simples, atribuição, aritmética básica, seleção, iteração e operações com vetores e coleções. Objetos foram apresentados e encapsulamento foi discutido brevemente. Herança e polimorfismo não foram tratados nestes cursos. Ambos utilizaram o mesmo livro, típico de cursos introdutórios, que apresentam conceitos imperativos primeiro e em seguida apresentam noções de objetos. Nenhum dos cursos utilizou ambientes especiais de desenvolvimento ou ferramentas. Os alunos utilizaram um editor de texto padrão para desenvolvimento das classes e somente a saída texto padrão, sem interfaces gráficas.

A diferença entre os dois grupos foi o tratamento aos objetos pelo instrutor. O primeiro grupo (CS1OE, com 32 alunos) consistia de uma turma de alunos conduzida por um instrutor, que após a apresentação dos objetos, mantinha o foco nos objetos, no projeto e no uso de objetos no código. Na primeira aula os alunos utilizaram objetos prontos e criaram os seus próprios. Encapsulamento e projeto continuaram a ser a ênfase durante o semestre. O instrutor não seguiu a apresentação de conteúdos do livro e utilizou sua experiência pessoal para abordar os tópicos. Para finalizar foi solicitado aos alunos que criassem uma instância de uma classe de coleção em Java (HashMap), a povoassem com objetos de sua própria criação e a manipulassem.

O segundo grupo (CS1OD, com 56 alunos) foi dividido em dois grupos de 28 alunos cada e com dois instrutores distintos. Estes instrutores trabalharam juntos e não enfatizaram objetos da mesma forma. Estudantes foram apresentados à idéia e definição de objetos, mas nunca foi solicitado que criassem seus próprios objetos e os utilizassem. A instanciação de objetos e a solicitação serviços raramente eram feitas. O foco foi em métodos estáticos, o método main como ponto de partida para a aplicação e campos públicos nas classes. A estrutura de desenvolvimento das classes seguiu o estilo de “entrada-processamento-saída”. Para finalizar, foi solicitado que os alunos criassem e manipulassem um vetor de tipos

primitivos (inteiros), realizando operações aritméticas básicas. Objetos foram introduzidos e definidos, porém a seqüência do material no curso não enfatizava o seu uso.

No semestre posterior, o primeiro tópico a ser abordado foi programação orientada a objetos, incluindo idéias de encapsulamento, herança e polimorfismo. Aos estudantes foi apresentada a idéia de criar seus próprios objetos desde o princípio. Durante a primeira aula foi solicitado que os alunos criassem e usassem objetos quase prontos. Na segunda a ênfase foi no conceito de herança e polimorfismo e na seqüência introduziram-se conceitos e classes da interface gráfica, que requeria conhecimento de herança e polimorfismo.

A análise do desempenho dos grupos na primeira disciplina não mostrou diferença significativa, porém na segunda disciplina os alunos de CS1OE mostraram rendimento muito melhor, como pode ser observado nas conclusões do trabalho em [DEC03].

Como então utilizar o paradigma de orientação por objetos de forma a aproveitar o máximo de seu potencial? Uma resposta para esta pergunta é:

- Introduzir programação de computadores utilizando o modelo *object-first* [ACM01] com apoio dos artefatos da UML, como os diagramas de classes e diagrama de objetos;
- Ao mesmo tempo utilizar a proposta *model-first* [ACM01] para fazer a introdução através de projetos que permitam experimentar os conceitos de forma contextualizada. Os padrões de projeto (*design* patterns) podem ser utilizados para apoiar a elaboração dos projetos, permitindo assim, o contato com as melhores soluções conhecidas e formalizadas;
- Para facilitar e incrementar as atividades durante os cursos os padrões pedagógicos (que serão abordados no Capítulo 4) podem ser utilizados;
- Para desenvolver as atividades durante cursos, ambientes integrados de desenvolvimento (como BlueJ e Jurtle) podem ser utilizados, facilitando a visualização, a experimentação e o contato direto com objetos. Os micromundos (como Karel J. Robot e AllKara) também podem ser utilizados, permitindo experiências individualizadas com os objetos, uma boa metáfora e permitir a cada aluno a construção de seu próprio conhecimento (como Karel J. Robot e AllKara).

3.4 APOIO AO ENSINO DO PARADIGMA OO

Nesta seção serão abordados os principais instrumentos de apoio ao ensino do paradigma OO, chamados no Capítulo 2 de IDEs pedagógicas e micromundos: LOGO,

AllKara e Karel J. Robot. A relação destas IDEs com as orientações pedagógicas e práticas que têm se mostrado frutíferas no ensino e aprendizagem de OO será tratada no Capítulo 4.

3.4.1 LOGO e seus Descendentes

Falando a respeito de LOGO, Papert [PAP85] destaca que LOGO seria "... *a computer language that would be suitable for children. This did not mean that it should be a 'toy' language. On the contrary, I wanted it to have the power of professional programming languages, but I also wanted it to have easy entry routes for nonmathematical beginners*".

LOGO foi desenvolvida no Massachusetts Institute of Technology (MIT), Boston E.U.A, pelo matemático, cientista da computação e psicólogo Seymour Papert e Wally Feurzeig em 1966 na BBN, uma empresa de Cambridge, com o objetivo de tornar as crianças construtoras de seu próprio conhecimento intelectual. A princípio, construtora de seu conhecimento de geometria. O computador figurava como uma poderosa ferramenta com a qual as crianças poderiam formular algoritmos para criar certos padrões e testá-los. Para este fim a linguagem LOGO foi criada como um dialeto da linguagem LISP [REI03].

O micromundo da geometria da tartaruga funciona como um local onde se vive e experimenta a matemática e assim se adquire os "conceitos matemáticos".

Como linguagem de programação, LOGO serve para nos comunicarmos com o computador, apresentando características especialmente elaboradas para implementar uma metodologia de ensino baseada no computador e explorar aspectos do processo de aprendizagem. Assim, LOGO tem duas raízes: uma computacional e a outra pedagógica. Como linguagem de programação, é de fácil assimilação: explora atividades espaciais, possui fácil terminologia, capacidade de criar novos procedimentos, trabalhar com conceitos de programação de forma bem natural. Em LOGO, a atividade de programar assume a característica de ser uma extensão do pensamento do aluno. A seqüência de comandos que o aluno emprega e as construções que elabora podem ser vistos como uma descrição, passível de análise e depuração, do processo que ele utiliza para resolver uma determinada tarefa, construindo um meio rico para o aprendizado de conceitos e de idéias para a resolução de problemas.

O aspecto pedagógico de LOGO está fundamentado no construtivismo de Piaget [VAL96]. Piaget mostrou que a criança já tem mecanismos de aprendizagem que são desenvolvidos sem a necessidade da freqüência à escola. A criança desenvolve sua capacidade intelectual interagindo com objetos do ambiente onde vive e utilizando o seu mecanismo de aprendizagem. Isto acontece sem que a criança seja explicitamente ensinada. É este aspecto

do processo de aprendizagem que LOGO pretende resgatar: um ambiente de aprendizado onde o conhecimento não é passado para a criança, mas no qual a criança, interagindo com os objetos desse ambiente, possa desenvolver outros conceitos [VAL02a].

No final da década de 70, com o advento dos computadores pessoais, a cultura LOGO se desenvolveu. E foi sofrendo evoluções, até que nos anos 90, sentindo a necessidade de atualização tecnológica, educadores, nos Estados Unidos e Canadá, incorporaram várias mudanças no ambiente LOGO, possibilitando trabalhar com várias tartarugas, recursos multimídia, jogos e simulação, que culminou no LOGO MicroWorlds[LCS00]. Existem ferramentas de criação de objetos que permitem criar tartarugas, que podem se tornar caixas de texto, botões para executar um comando LOGO, música ou clipes, além de permitir a gravação de sons, execução de vídeos e composição de melodias. As ferramentas de edição permitem apontar um objeto para mover, selecionar e redimensionar; visualizar as características dos objetos, permitir alterações; para fazer cópias e aumentar ou diminuir o tamanho dos objetos.

Apesar de não ter sido projetado para ensinar programação, LOGO influenciou o desenvolvimento de micromundos para este propósito. Existem vários ambientes que utilizam a filosofia LOGO [PAU00]:

MegaLOGO: lançado em 1995 pela CNOTINFOR, totalmente em português (de Portugal), para Windows 3.1 e 95, com 4000 tartarugas, utilizando até 16 milhões de cores. É uma adaptação do Comenius LOGO desenvolvido na Universidade de Bratislava, Tchecoslováquia;

UCBLOGO: desenvolvido por Brian Harvey para o Macintosh, MSDOS e Unix, na Universidade de Bekerley, Califórnia, e a de Georges Mills, que construiu o MSWLOGO para Windows 3.1/95/98 e NT;

StarLOGO: versão desenvolvida por uma equipe do MIT liderada por Mitchel Resnick, onde milhares de tartarugas podem executar processos independentes e interagir uma com as outras. Sua finalidade é de facilitar a exploração de sistemas distribuídos;

SuperLOGO [SLO95]: tradução e adaptação feita pelo NIED (Núcleo de Informática Aplicada à Educação da Universidade Estadual de Campinas) do ambiente originalmente desenvolvido pela Universidade de Berkeley por George Mills.

TFXTurtle: componente criado em 1997 por Chuck Gadd, da Cyber Communications, Inc., que incorpora uma nova classe para o Delphi e implementa os principais componentes do LOGO;

PascaL [GUI00]: sobre o ambiente MSWLOGO, Ângelo Guimarães desenvolveu um ambiente que permite que os comandos solicitados à tartaruga possam ser escritos na linguagem Pascal.

O LOGO MicroWorlds possui três centros de comandos: um centro para digitar comandos para as tartarugas, um centro de formas onde se pode selecionar ou criar novas formas para as tartarugas e um centro de ferramentas de desenho para se desenhar e colorir o cenário.

Quando se cria uma tartaruga neste ambiente, ela recebe um nome padrão *tn* (*n* é um número seqüencial) que pode ser alterado na sua caixa de características. Para se enviar uma mensagem ou "conversar" com uma tartaruga, basta colocar o nome dela seguido de uma vírgula e o comando que se deseja que ela execute. Exemplo: t1, fd 30 (forward 30). Para conversar com mais de uma tartaruga, basta listá-las e solicitar o comando. Exemplo: talkto [t1 t2] fd 30.

Com estes recursos, podemos ver que conceitos de orientação a objetos são tratados neste ambiente de forma "natural". Para criar um novo objeto (tartaruga), utiliza-se a ferramenta "ovo de tartaruga"; para alterar suas características utiliza-se a ferramenta "olho"; para criar seus métodos utiliza-se a idéia de procedimento que é descrito na linguagem LOGO; ao duplicar uma tartaruga ela herda as características da sua "progenitora". Assim, podemos observar que o ambiente LOGO MicroWorlds é uma ferramenta potencial para a introdução do paradigma OO. A orientação por objetos de forma induzida pode ser observada na criação de uma simulação do sistema solar (sol-terra-lua) mostrada na seqüência de Figuras 3-1 a 3-4.

As Figuras de 3-2 a 3-4 exemplificam a utilização do MicroWorlds, destacando suas características induzidas de orientação por objetos.

Nestas últimas décadas, a linguagem LOGO tem acompanhado o desenvolvimento da tecnologia e continua mundialmente conhecida e utilizada por pessoas que acreditam na filosofia construtivista [VAL96, PAP96, PAU00].

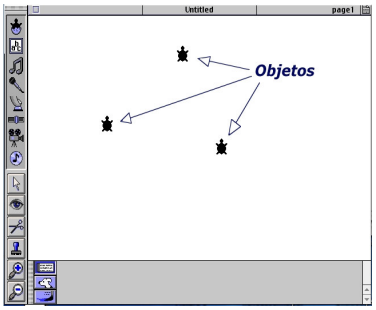

Figura 3-1 Criando três objetos da classe Tartaruga

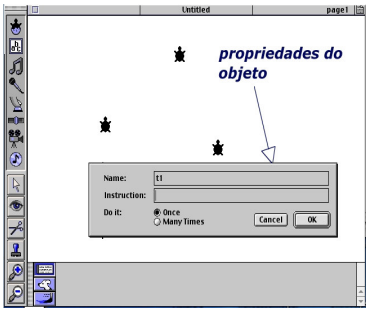

Figura 3-2 Tendo acesso a algumas propriedades de um objeto

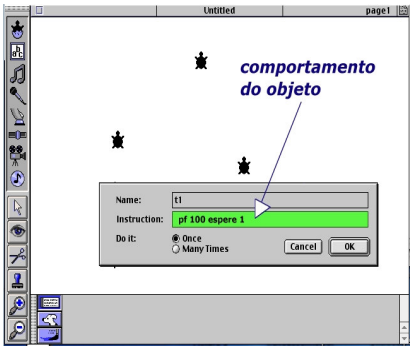

Figura 3-3 Descrevendo o comportamento do objeto

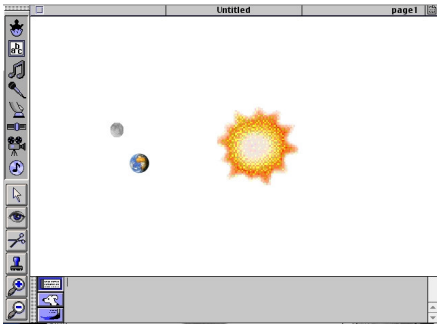

Figura 3-4 Mudando a propriedade que transforma a propriedade "ícone" do objeto para dar a visão desejada em uma simulação

3.4.2 Karel e seus Descendentes

Karel, o Robô, foi introduzido em 1981 por Richard E. Pattis, Universidade de Stanford, em seu livro *Karel the Robot: A Gentle Introduction to the Art of Programming with Pascal*. Neste ambiente, o usuário escreve programas para um robô chamado Karel. A linguagem utilizada para programar Karel é parecida com o Pascal. As instruções dos programas são executadas pelo robô, por meio de ações.

O robô mora em um mundo cartesiano de ruas e avenidas, contendo objetos especiais (muros e *beepers*) que ele pode sentir e manipular. As ruas e avenidas são numeradas. As posições podem ser absolutas ou relativas e o vocabulário de Karel é muito limitado, existindo apenas cinco instruções. Porém, o vocabulário de Karel pode ser estendido, definindo-se novas instruções, que são baseadas em instruções que Karel já aprendeu anteriormente. A criação de novas instruções permite ensinar um bom estilo de programação, que deve ser fácil de ler, entender, depurar e modificar.

No curso de Introdução a Computadores, oferecido na Universidade de Aberdeen, Scotland, UK, é utilizado um ambiente Karel adaptado por Russel e Hunter [RUS98] que permite o ensino de outras instruções de repetição (FOR....TO e REPEAT...UNTIL), variáveis, funções, expressões e constantes. A programação de Karel é feita utilizando a linguagem Object Pascal no Delphi.

Existem várias implementações de interfaces para o ambiente Karel, para DOS, Windows e Unix [BUC99,PAU00]. Ferramentas inspiradas em Karel são listadas a seguir.

- **Karel ++** é um dos primeiros descendentes e utiliza a linguagem C++.
- **Jarel** é uma implementação Java de Karel, desenvolvida pelo professor Charlie McDowell, do Computer Science Department, University of Califórnia, Santa Cruz. Neste ambiente, o aluno manipula uma classe Java que permite escrever código puro em Java, de forma tão simples como nos programas para Karel. Como os programas escritos são códigos Java, a transição para um programa Java "normal" é simples. Além disso, os programas gerados podem ser usados para explorar conceitos de programação suportados pela linguagem Java [MCD00].

Ao se criar Jarel, o objetivo foi proporcionar ao aluno um ambiente onde se estabelece um primeiro contato através da resolução de problemas interessantes, que levam a uma motivação natural, na intenção de criar melhores estratégias.

O conceito de abstração procedural pode ser introduzido muito rapidamente. Naturalmente, em seus primeiros passos com Jarel, como este sabe apenas como virar para a direita, o aluno sente a necessidade de criar o procedimento "turnLeft( )".

Diferentemente de Karel, com Jarel os estudantes utilizam métodos com parâmetros e têm contato com o conceito de variáveis. Dependendo do curso e do instrutor, conceitos de programação orientada a objetos podem ser gradualmente introduzidos.

- **Karel J. Robot** é outro descendente direto, que utiliza a linguagem Java. Este será descrito na próxima sessão.

Karel J. Robot

Em Karel J. Robot - A Gentle Introduction to the Art of Object-Oriented Programming in Java, de Joseph Bergin, Mark Stehlik, Jim Roberts e Richard Pattis [BER02d], tem-se uma introdução aos conceitos de orientação a objetos, utilizando micro-mundos com Java.

Mais que um ambiente didático para introdução à programação de computadores utilizando orientação a objetos, Karel J. Robot é uma filosofia de ensino:

- O autor sugere uma abordagem de ensino em **espiral**: o professor não tenta abordar tudo sobre um determinado assunto na primeira vez que o tópico aparece. Neste estilo, aborda-se o suficiente para resolver o problema, sabendo-se que o tópico será retomado posteriormente com um nível de profundidade maior, mostrando variações. Por exemplo, quando se abordar o comando IF não tratar também do comando SWITCH. A abordagem do comando IF junto com polimorfismo pode produzir melhores exemplos, enquanto IF e SWITCH não [BER04].
- As atividades caminham do **simples** para o **complexo**: as classes apresentadas devem ser simples, completas e seus métodos pequenos e com uma sua finalidade bem definida (coeso). Cada objeto tem um propósito simples: suprir um serviço simples que outros objetos possam usufruir. A complexidade e sofisticação surgem da combinação destas estruturas simples, usadas para construir estruturas complexas e não a partir de estruturas complexas.

- Prima pela **qualidade** do produto desenvolvido: associada a uma boa prática de programação orientada a objetos, fundamentada em seus conceitos puros, está a utilização de padrões de desenvolvimento (“design patterns”). Estes são soluções padronizadas a problemas recorrentes descritas de tal forma que facilite sua comunicação [BER04].

Mais que uma introdução à programação, Karel J. Robot é uma introdução ao pensamento orientado a objetos. A programação orientada a objetos não é tratada como uma extensão de programação estruturada, mas como uma nova metodologia de desenvolvimento baseada em redes de objetos polimórficos, realizando tarefas simples e gerando comportamento complexo através da interação.

Desde o princípio o aluno é estimulado a utilizar a criação de novos métodos e sobreposição, ao invés de utilizar estruturas como seleção e repetição. A criação de novas classes por herança é sugerida para gerar comportamentos mais sofisticados.

Karel J. Robot é um material para as primeiras semanas de um curso introdutório. Como não é um material completo para um curso deve-se providenciar material adicional [BER04].

Neste ambiente proposto, os robôs vivem em um mundo simples, porém com a possibilidade de criar atividades interessantes que conduzem o pensamento e o entendimento de conceitos de orientação a objetos.

O mundo é uma grade, representando ruas e avenidas, onde os robôs podem passar. Também contém elementos especiais que os robôs podem sentir e manipular. Na Figura 3-5 temos uma ilustração deste mundo.

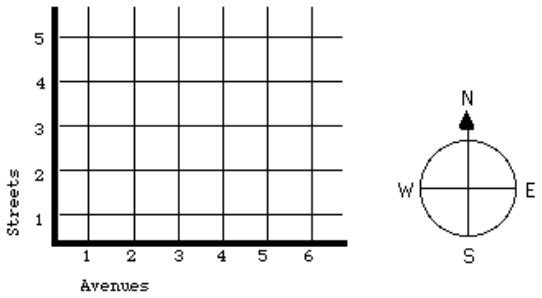

Figura 3-5 Ruas e Avenidas do mundo de Karel e sua orientação [BER02d]

Cruzando o mundo estão ruas horizontais (no sentido leste-oeste) e avenidas verticais (no sentido norte-sul), com intervalos de um bloco. Existem os cruzamentos, interseções ou esquinas entre ruas e avenidas, onde um ou mais robôs podem estar.

Os robôs possuem nomes para que as mensagens possam ser enviadas individualmente. Quando se trabalha com apenas um robô, ele é chamado Karel.

Ruas e avenidas possuem um número, o que permite identificação única da posição. A primeira interseção onde a rua 1 intercepta a avenida 1 é chamada origem. A posição dos robôs e outros elementos do mundo podem ser descritos usando a localização relativa e absoluta. Por exemplo, a localização absoluta da origem é a interseção da primeira rua com a primeira avenida. Quando é dito que um robô se encontra a dois blocos a leste e três blocos a norte de algo no mundo, fala-se em localização relativa.

Além de robôs, dois outros tipos de elementos podem ocupar o mundo. O primeiro é uma parede, estas bloqueiam a passagem dos robôs de uma esquina para outra e representam obstáculos que os robôs devem contornar. Com estas paredes podem-se construir quartos fechados, labirintos e outros tipos de obstáculos. Alguns exemplos estão na Figura 3-6.

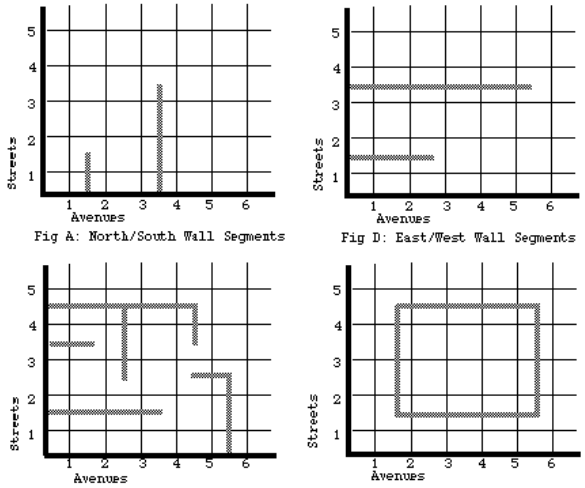

Figura 3-6 Exemplos de configurações do mundo de Karel [BER02d]

O segundo elemento é o *beeper*. *Beepers* são pequenos cones de plástico que emitem um curto som de buzina. Podem ser colocados nas interseções e podem ser apanhados, levados e colocados por robôs. Os *beepers* são pequenos e em uma mesma interseção pode-se ter mais de um, além disso não interferem nos movimentos dos robôs. Na Figura 3-7 temos uma ilustração de uma seqüência de *beeper*s.

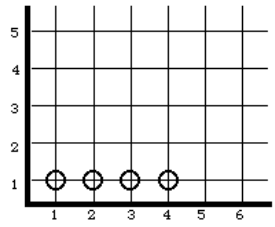

Figura 3-7 Quatro *beepers* paralelos [BER02d]

Os robôs são móveis; podem movimentar-se para frente (na direção que está apontando: norte, sul, leste ou oeste), e podem virar. Também podem perceber sua vizinhança imediata usando sentidos rudimentares de visão, audição, direção e toque.

A visão do robô é fornecida por uma câmera que aponta para frente, permitindo a detecção de uma parede exatamente a um meio-bloco à frente do robô. A capacidade de audição é garantida quando o *beeper* estiver na mesma interseção que o robô. Uma bússola interna permite ao robô determinar em que direção está. Braços permitem ao robô pegar e colocar *beepers*. Para transportar os *beepers*, cada robô possui em sua cintura uma bolsa a prova de som. Um robô também pode determinar se está levando *beepers* nesta bolsa sondando a bolsa com seu braço. Um robô também pode usar seu braço para determinar se há outros robôs no mesmo local que ocupa. Finalmente, um robô pode se "desligar" (deixar de existir) quando sua tarefa for completada.

Robôs são feitos em fábricas. Todos vêm de uma fábrica principal, Karel-Werke, que pode oferecer vários tipos de modelos de robôs. Para executar uma determinada tarefa podemos usar um modelo padrão, ou especificar um novo modelo. A fábrica permite a construção de robôs especializados que são modificações ou extensões dos modelos existentes. Neste caso, é preciso especificar informações detalhadas que descrevam as características especiais dos robôs e como ele irá executar as tarefas propostas. Com esta metáfora de moldes (formas) para robôs é abordado o conceito de **classe**. O conceito de **programação** é a criação de uma nova fábrica (classe).

Robôs são exemplos de **objetos**. Um objeto é algo eletrônico, porém real, que pode **fazer** coisas e **lembrar**-se de coisas. Pode-se pedir aos robôs que façam algo que saibam fazer e perguntar sobre coisas que lembram.

Para a maioria das tarefas um robô é suficiente. Quando um robô é produzido pela fábrica, um helicóptero o entrega no mundo. O piloto do helicóptero monta os robôs de acordo com as devidas especificações e envia para cada novo robô uma seqüência de mensagens para detalhar as tarefas que pode então realizar (programá-lo).

Uma linguagem especial é utilizada para programar os robôs, "quase" Java. E como qualquer linguagem natural, possui marcas de pontuação e regras de gramática, mas é simples, visando ao entendimento pelo robô. Porém, poderosa e concisa, permite escrever programas rapidamente sem ambigüidade.

Neste micromundo o ensino/aprendizado é feito através de tarefas e situações que devem ser resolvidas pelos robôs. As **tarefas** são desafios que os robôs precisam resolver

como: mover-se para esquina da rua 15 com a avenida 10, passar por obstáculos, sair de um quarto fechado que possui uma porta, encontrar um *beeper* e colocá-lo de volta, sair de um labirinto. As **situações** mostram a configuração do mundo antes e depois da resolução da tarefa proposta. Na Figura 3-8 algumas situações iniciais de um robô, representado pelas setas pretas, são mostradas.

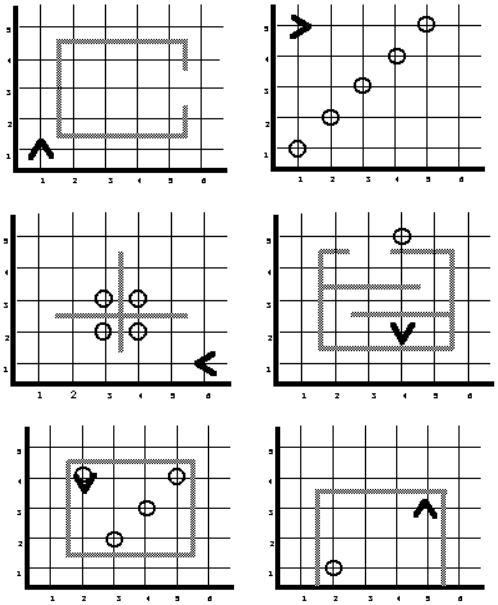

Figura 3-8 Exemplos de situações iniciais [BER02d]

Um robô executa ações através da realização de uma seqüência de instruções associada. Uma seqüência de instruções fornecida pelo piloto do helicóptero constitui um programa. Cada instrução na seqüência é enviada para o robô como uma mensagem, que direciona o robô a realizar alguma tarefa no programa.

Tabela 3-1 Instruções disponíveis para Karel J. Robot.

Instrução	Efeito
Move	Ao receber esta instrução o robô move um bloco à frente, continuando na mesma direção. Para evitar problema, o robô não avançará e se desligará. Desta forma, esta instrução pode causar erro.
turnLeft	Ao receber esta instrução o robô muda sua direção, girando 90º para a esquerda e continua no mesmo local. Como este movimento não coloca o robô em nenhuma situação indesejada, não causa erro.
turnOff	Ao receber esta instrução o robô sabe que sua tarefa terminou.
pick*Beeper*	Ao receber esta instrução o robô pega um *beeper* da esquina onde se encontra e o coloca em sua bolsa de *beepers*. Caso o robô tente pegar um *beeper* de um local sem *beeper*, uma situação de erro ocorre. Se existirem vários *beepers*,

	somente um será retirado e colocado na bolsa.
putBeeper	Ao receber esta instrução o robô tira um *beeper* de sua bolsa de *beepers* e o coloca na esquina onde se encontra. Se não existe *beeper* na bolsa uma situação de erro ocorre. Se existirem vários *beepers* na bolsa somente um será retirado e colocado no local onde se encontra.

Todos os robôs produzidos pela fábrica Karel-Werke possuem as capacidades descritas. Para criarmos novos tipos de robôs, baseados no tipo primitivo, é preciso descrever as habilidades adicionais para que a fábrica passe a fabricar também os novos robôs. Karel-Werke utiliza uma linguagem de programação para robôs para descrever as habilidades as instruções dos robôs, chamadas programas. O nome formal para a descrição das instruções do robô é **método**. O modelo do robô descrito acima é chamado **UrRobot** (ur é um prefixo alemão que quer dizer original ou primitivo).

Para motivar o tópico, criações de novos métodos, são colocadas tarefas que o robô deverá executar. Estas poderiam ser resolvidas com as instruções básicas dos robôs da classe UrRobot, mas ficariam melhores se fosse programada uma nova instrução para facilitar tanto o processo de desenvolvimento da classe, escrita do código e seu entendimento.

Uma característica muito especial da fábrica de robôs é que ela pode ser estendida, ou seja, podem-se criar novas classes e nelas ensinar novos comandos aos robôs utilizando comandos já conhecidos. Estes novos comandos são incorporados ao dicionário dos novos robôs, isto é, eles ficam com o dicionário inicial fornecido pela classe UrRobot, mais os novos comandos ensinados. O que nos permite programar o robô de forma mais parecida com o nosso modo de pensar.

3.4.3 AllKara

AllKara é um ambiente que proporciona contato efetivo e descomplicado com conceitos valiosos da ciência da computação. Esta ferramenta foi concebida por Raimond Reichert, Jurg Nievergelt e Werner Hartmann, no Departamento de Informática, da ETH de Zurich, em 2001. Sua tela de apresentação pode ser vista na Figura 3-9.

A idéia central deste ambiente é tratar a programação de computadores como parte da educação geral de um indivíduo. Da mesma forma que nem tudo na educação geral que o indivíduo recebe nas escolas tem aplicação imediata na resolução de problemas do dia a dia, AllKara é um conjunto de ferramentas que pode agir como facilitador nos processos de resolução de problemas diários através do uso de recursos preciosos da Ciência da Computação como Máquina de Estados Finitos, Máquina de Turing, Linguagem de Programação (JavaKara) e Lego [REI01].

Por exemplo, sabemos que muitos profissionais de engenharia e ciência precisam utilizar os resultados matemáticos. Estes são usuários da matemática, eles checam pré-condições (entradas) de um teorema ou fórmula e utilizam as conclusões (saída). Desta forma, precisam saber apenas como teoremas e fórmulas são aplicados.

Poderíamos pensar que o conceito de prova de teoremas é relevante aos profissionais da matemática. Porém, os usuários da matemática farão uso muito melhor do ferramental matemático se eles tiverem entendido o conceito de prova. Não é necessário saber fazer todas as provas matemáticas para usar os teoremas (fórmulas), mas o entendimento do conceito de prova é essencial para fazer um uso melhor da matemática, ou, por exemplo, para explorar melhor um software de matemática. Educação geral raramente é aplicada para uso imediato, mas auxilia em um contexto mais amplo, onde é preciso reunir "pedaços" do conhecimento para melhor desempenhar tarefas [REI01].

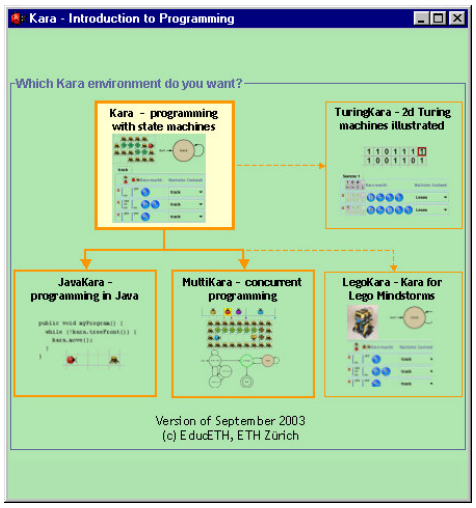

Figura 3-9 Tela inicial do ambiente Kara

De forma similar, usuários de computador não precisam entender o código de todas as aplicações que utilizam, mas devem ter uma intuição do que constitui um programa. E isto é possível através de experiências pessoais em escrever, testar e depurar um número pequeno de programas, o que propõe o ambiente AllKara. Os softwares de aplicações que são utilizados hoje em dia não exigem programação como ferramenta, mas como um conhecimento de segundo plano que auxilia no entendimento do que e como o computador executa as tarefas [REI01].

Segundo Reichert et al. [REI01], linguagens como C++, Java e Delphi foram feitas para programadores profissionais; elas são poderosas e complexas. Não há necessidade de

introduzir principiantes na complexidade de linguagens de programação e ambientes profissionais [REI01]. Assim, no ambiente AllKara a programação é praticada como um exercício que combina o lúdico com a idéia de resolver desafios, demandando e ilustrando conceitos fundamentais de programação, que vão gradativamente crescendo em complexidade.

Para um iniciante, qualquer linguagem tratada de forma completa é um desafio. Para reduzir o esforço para vencê-lo, em AllKara ao invés de usar uma linguagem convencional, o usuário utiliza um modelo de computação como máquinas de estados finitos, Kara. Como citado por Reichert [REI01], este modelo foi proposto em 1999 e faz parte do nosso dia a dia. Onde caminhos de execução são definidos estaticamente como caminhos em um gráfico direcionado; nenhuma outra estrutura de controle é necessária, o que simplifica o modelo.

Este ambiente de programação, projetado de forma simples, para os primeiros passos em programação pode ser considerado como outro ambiente de "mini linguagem" nos moldes de Karel o Robot, mas diferenciando em sua simplicidade conceitual: a escolha de uma máquina de estados finitos como modelo de computação.

Os primeiros passos em Kara são simples, fáceis e divertidos. Kara é programado inicialmente de forma gráfica, não há necessidade de conhecer a sintaxe de uma linguagem de programação. À primeira vista as habilidades de Kara são muito limitadas. Entretanto, existem várias tarefas que Kara pode ser programada para resolver em seu mundo, incluindo alguns problemas difíceis. Onde o aluno é desafiado e aprende. Quando os alunos alcançam o limite de Kara e desejam avançar eles podem fazê-lo, aprendendo uma linguagem de programação profissional – Java – no ambiente de programação JavaKara. Este ambiente é projetado para dar os primeiros passos em Java de forma fácil e visual, sem qualquer conhecimento anterior de Java.

Kara: um ambiente de programação baseado em máquina de estados finitos

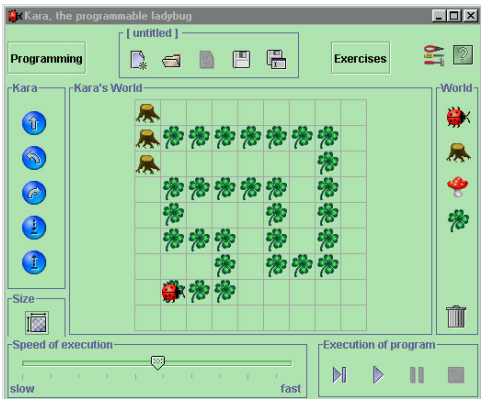

Figura 3-10 Configuração inicial do mundo de Kara

Kara é um inseto programável que vive em seu mundo gráfico na tela, uma grade retangular de quadrados como mostrado na Figura 3-10. Ele pode mover-se apenas para uma casa adjacente. Em qualquer quadrado podemos encontrar objetos de seu mundo: tronco de árvore, cogumelo, trevo e kara.

Existem cinco sensores que **kara** pode utilizar:

- Em frente há uma árvore?
- Há uma árvore à esquerda?
- Há uma árvore à direita?
- Em frente há um cogumelo?
- Está sobre uma folha?

Abaixo temos cinco comandos que **kara** obedientemente executa: avance uma casa, vire à esquerda 90º, vire a direita 90º, deixe um trevo e pegue um trevo.

Considere o exemplo acima onde o objetivo de Kara é pegar uma trilha de trevos, esta trilha nunca vai além de um tronco. Assim, o programa deve parar quando Kara encontra-se em frente ao tronco. O programa da Figura 3-11 mostra o diagrama de estados para resolver este desafio.

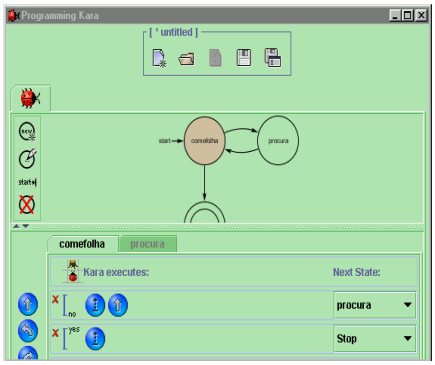

Figura 3-11 Tela de edição dos estados do diagrama

Além de desafios simples como percorrer circuitos, colocar trevos para formar um padrão de tabuleiro de xadrez, existem vários desafios que Kara pode resolver, como o PacMan ou Sokoban simplificado cujo diagrama de estados é mostrado na Figura 3-11, computar um triângulo de Pascal ou mover torres de Hanói.

O Editor do mundo de Kara

Precisamos conhecer o mundo de Kara para conduzi-lo de forma adequada. Este mundo pode ser carregado, quando alguém já o elaborou, ou pode ser criado. Na Figura 3-10, temos o editor do mundo de Kara. Na configuração do ambiente conta-se com os elementos disponíveis no micromundo: folhas, troncos, cogumelos e apenas um objeto Kara.

Para carregar um ambiente previamente preparado clica-se em **Exercises** onde se tem acesso à descrição do problema, a configuração do micromundo para o desafio que está sendo proposto e à solução.

A programação de Kara inicia ao clique do botão **Programming**, através do qual se tem acesso ao editor. Este botão pode ser visto na Figura 3-10.

JavaKara: uma linguagem como Java

Após conhecer melhor o mundo virtual de **Kara**, programando-o através da máquina de estados finitos, os esforços são concentrados no aprendizado da linguagem de programação de **Kara**, JavaKara, em um novo editor. A Figura 3-12 mostra um programa em JavaKara.

Nesta outra etapa, tem-se contato com as bases de Java através de uma série de exercícios, que são gradativamente mais complexos, projetados para motivar e introduzir os elementos de programas na linguagem Java.

A princípio, trabalha-se somente com um subconjunto de Java. Modelos pré-definidos ocultam os construtores orientados a objetos de Java [REI01]. Esta estratégia facilita o foco

nos construtores da linguagem e conseqüentemente o aprendizado e a aquisição de conceitos fundamentais: objetos, métodos, processos de seleção, processos de repetição, expressões lógicas, operadores lógicos, tipos primitivos, vetores entre outros.

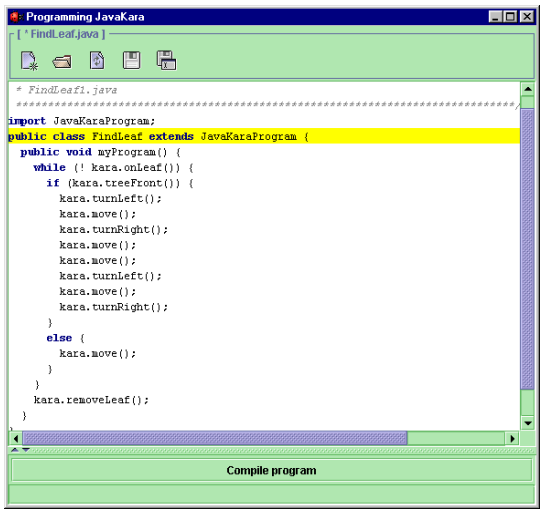

Figura 3-12 Um programa em JavaKara

Os itens abaixo são requisitos do ambiente JavaKara:

- Um programa para JavaKara deve ser derivado da classe JavaKaraProgram;
- O programa principal deve estar no método *myProgram*.
- JavaKaraProgram principalmente oferece acesso ao objeto *kara*, que é a ligação entre os novos programas e o ambiente de programação.

Conceitos fundamentais presentes no ambiente AllKara

Para professores de ciência da computação experientes, o ambiente AllKara permite extrair vários conceitos e apresentá-los de forma bem acessível aos iniciantes, dentre eles podemos citar [REI01]:

Objetos e Métodos: Durante todo o tempo objetos disponibilizados pela classe JavaKaraProgram são manipulados: **kara**, **world** e **tools**. A solicitação de ações (comandos) pelos objetos consiste na execução de métodos disponibilizados na classe.

Passagem de parâmetros e tipo de dados: Os conceitos de passagem de parâmetros, tipos de dados e tipos de retorno são abordados ao se trabalhar a idéia de métodos.

Procedimentos: Programadores **Kara** percebem que certas seqüências de instruções são freqüentemente reutilizadas, são os subprogramas.

Utilizar diferentes modelos computacionais: A máquina de estados finitos de Kara serve para ilustrar diferentes modelos computacionais e mostrar como é importante definir todos os detalhes. Sutil diferença com relação ao que Kara pode fazer em seu mundo altera seu poder de computação. Em discussão teórica pode-se imaginar Kara vivendo em um mundo de tamanho infinito; podem-se considerar diferentes suposições do que permitimos que Kara faça. Na versão mais poderosa, Kara pode ler e escrever caracteres de um alfabeto finito – o quadrado vazio e o trevo são utilizados para representar o alfabeto {0, 1}, utilizados no módulo de Kara que simula a Máquina de Turing.

3.4.4 BlueJ

BlueJ é uma IDE didática para o ensino programação orientada a objetos usando Java. É um ambiente completo Java 2, construído sobre a plataforma padrão Java SDK e utiliza um compilador padrão e uma máquina virtual. O ambiente oferece uma única tela e um estilo de interação diferenciado de outros ambientes [KOL03]. A tela inicial de um projeto no BlueJ pode ser visto na Figura 3-13.

De acordo com Kolling [KOL03] existem problemas fundamentais com a maioria dos ambientes de ensino/aprendizagem de programação orientada a objetos disponíveis:

- **O ambiente não é orientado a objetos**: Na maioria dos ambientes o aluno utiliza arquivos e aplicações em vez de objetos e classes. Eles são forçados a pensar sobre os sistemas de arquivos e estruturas de diretórios. Colocar um projeto para funcionar pode ser uma tarefa não trivial. Estes elementos criam dificuldades ao ensino e desviam o foco dos elementos realmente importantes, objetos. O sentido de programação passa a ser o número de linhas de código e não as estruturas de objetos. Objetos como interação de entidades não são normalmente suportadas.

- **O ambiente é muito complexo**. Muitos professores não utilizam ambientes integrados pelo fato de não encontrar um que seja completo. Muitas vezes a preferência é trabalhar diretamente na linha de comando, o que gasta tempo considerável na familiarização com Unix ou DOS ao invés de investir o tempo em aprender sobre programação. Assim, ferramentas muito elementares, muito complicadas ou inapropriadas para uso causam problemas consideráveis.

- **O ambiente focaliza na interface do usuário**. Muitos ambientes concentram-se na construção de interfaces gráficas para usuários (GUIs), criando uma idéia distorcida da programação orientada a objetos (Na verdade, trata-se de programação orientada a

componentes). Os alunos passam a se preocupar com a aparência da interface e desviam o foco da programação. Um dos maiores benefícios do uso de elementos gráficos no ambiente é a exibição da estrutura das classes que é normalmente negligenciado.

Neste ambiente o usuário tem a possibilidade de ter contato de perto com os conceitos envolvidos em orientação a objetos, sem se preocupar com detalhes de implementação. É possível criar os objetos a partir das classes existentes, solicitar a execução de métodos pelos objetos, fornecer parâmetros em resposta à solicitação de alguns métodos, inspecionar objetos criados e ver que realmente armazenam dados. É possível ver para crer.

O objeto pode ser conhecido através de sua manipulação e quando o usuário o conhece, naturalmente pode identificar elementos essenciais da classe a qual pertence: campos, construtores e métodos. A partir desta identificação externa, o usuário está pronto para conhecer alguns detalhes da implementação da classe. Neste momento ele pode conhecer a classe por dentro.

Ao visualizar o código fonte referente à classe em estudo e com o conhecimento externo da classe é possível identificar tranqüilamente os elementos da classe em Java e começar a estudar o código fonte.

A aprendizagem é baseada na idéia do uso, da alteração e da criação das classes. Gradualmente, o ambiente permite que projetos mais elaborados sejam apresentados e que novos conceitos sejam apresentados, dependendo apenas da orientação do instrutor.

Aqui, o instrutor precisa deixar espaço para a experimentação, o usuário precisa manipular as classes para depois serem expostos aos conceitos.

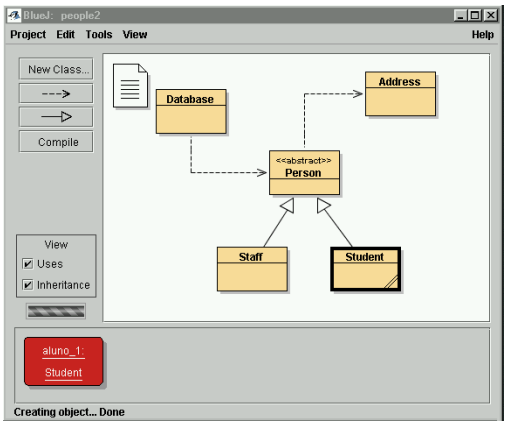

Figura 3-13 Tela de apresentação do projeto People2

Neste ambiente os conceitos de classe, objeto e herança são ressaltados de forma visual, favorecendo os processos de conceitualização do aprendiz. Estabelece o contato com

classes e objetos em primeira instância, a janela principal mostra um diagrama de classes UML que permite a visualização do projeto como um todo, ou seja, a visualização de sua hierarquia de classes, como pode ser visto na Figura 3-13.

Com BlueJ o usuário pode interagir diretamente com classes e objetos. Um ícone que representa uma classe pode executar um construtor que cria uma instância da classe, um objeto. Uma vez criado o objeto, ele aparece na bancada de objetos (por exemplo, o objeto aluno_1 da Classe Student na Figura 3-13). Qualquer método público associado ao objeto pode então ser executado.

BlueJ oferece um único mecanismo de chamada de métodos parametrizados que permite que os professores adiem a introdução de outras tecnologias de interface como interface baseada em textos, GUIs ou applets até um momento mais apropriado no curso.

O código fonte da classe é facilmente acessado através de um duplo clique no ícone da classe. O clique no botão *Compile* permite a recompilação em todas as classes alteradas e a execução pode iniciar novamente. Os erros de compilação são mostrados diretamente no editor, destacando a linha que o contém e um texto com a mensagem de erro.

Criação de objetos

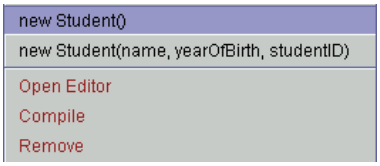

Figura 3-14 Menu suspenso referente à classe

Ao clicar com o botão direito sobre o ícone da classe é exibido um menu como mostrado na Figura 3-14. As duas primeiras operações são construtores da classe, o primeiro permite a criação de um objeto da classe *Student* sem parâmetros, o segundo permite a entrada das informações através dos parâmetros. *Open Editor* permite a edição do código da classe, *Compile* recompila a classe e *Remove* exclui a classe. Quando um construtor é chamado, uma caixa de diálogo é exibida na tela permitindo ao usuário dar um nome ao objeto, um nome padrão é sugerido, como pode ser visto nas Figuras 3-15 e 3-16.

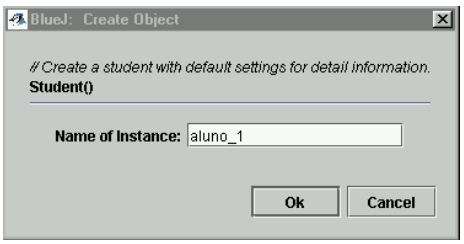

Figura 3-15 Criação do objeto aluno_1 com outras informações padrões

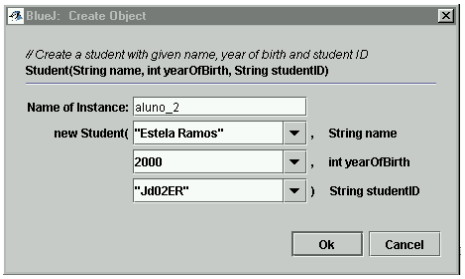

Figura 3-16 Criação do objeto aluno_2 com todas as informações

Selecionando OK na caixa de diálogo, o construtor é executado e o objeto criado, podendo ser visualizado na parte inferior da janela, na área reservada para os objetos, como um ícone retangular vermelho como aluno_1 na Figura 3-13.

Ativação de Métodos

O clique com o botão direito sobre o ícone que representa o objeto exibe o menu mostrado na Figura 3-16.

O menu do objeto mostrado contém duas operações de ambiente, *Inspect* e *Remove* e uma entrada para cada método público definido na classe do objeto. Métodos herdados são colocados em sub-menus, conforme mostra a Figura 3-17. Ao selecionar um dos métodos teremos a sua execução. Se o método requer parâmetros eles devem ser fornecidos através de uma caixa de diálogo similar à da Figura 3-18 para a criação do objeto aluno_2.

Com a opção *Inspect* o inspetor de objetos é ativado. Este pode ser usado para checar o efeito dos métodos, exibindo os valores de todos os campos e instâncias estáticas do objeto, como mostrado na Figura 3-18. Qualquer campo pode ser ele mesmo objeto que pode ser recursivamente inspecionado.

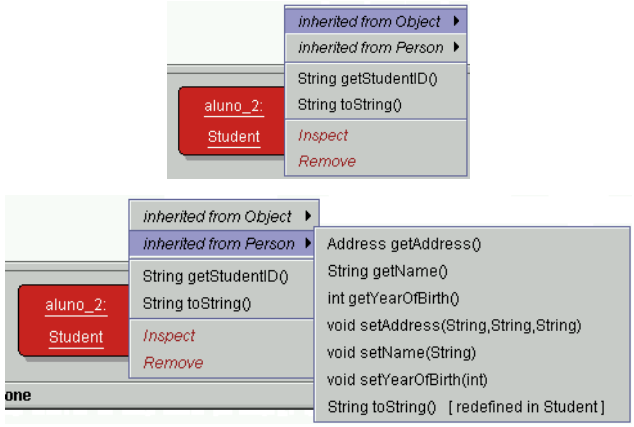

Figura 3-17 Menu suspenso referente ao objeto

Visualização

A estrutura de classes exibida na janela principal é um dos aspectos principais do BlueJ. Esta visualização força os alunos a reconhecer e pensar sobre a estrutura de classes antes de verem o programa em Java. Quando os alunos têm contato com o primeiro programa em Java fica claro que uma aplicação é um conjunto de classes em cooperação.

Outro aspecto positivo da visualização é a possibilidade de criar várias instâncias (objetos) de uma mesma classe e estabelecer de forma simples e clara a diferença entre classe e objeto. Sem muito esforço, os alunos podem perceber que as classes são criadas para gerar objetos e que os objetos contém dados. Verificam também que o tipo de dados em cada objeto da mesma classe é o mesmo, enquanto os valores atuais são diferentes.

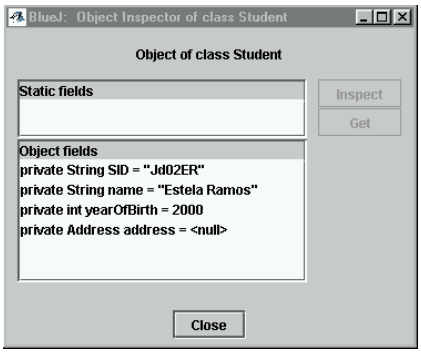

Figura 3-18 Inspeção de objeto

Torna-se nítido que objetos são manipulados através da solicitação de operações próprias que alteram seu estado e que algumas operações retornam informações sobre seu estado.

Assim, tendo uma visualização abstrata das entidades que permite a interação direta, é possível mostrar os conceitos de OO de uma maneira poderosa e fácil de entender sem a necessidade de longas explicações.

Simplicidade

BlueJ foi projetado especificamente para iniciantes. Iniciantes precisam de ferramentas diferentes das utilizadas por profissionais de engenharia de software [KOL03]. O objetivo central é ensinar sobre programação orientada a objetos, não sobre o uso de um ambiente particular. Por isso, BlueJ é simples. No primeiro contato os alunos, em pouco tempo, já fazem uso adequado do ambiente.

Outras Características

Existem outras características de BlueJ que podem ser mencionadas:

- integração;
- depurador fácil de utilizar;
- suporte integrado para geração de *Javadoc;*
- suporte sofisticado para geração e execução de applets, que inclui geração automática do *skeleton* do applet e carga de uma página HTML e capacidade de rodar o applet em navegadores web e visualizadores de applets;
- função *export* que pode criar arquivos *jar*.

Ensinar com BlueJ

Kooling e Rosenberg [KOL01] resumem as principais idéias da pedagogia de ensinar com BlueJ:

1. Objetos primeiro
2. Não comece com uma tela em branco
3. Leia o código
4. Utilize projetos “grandes”
5. Não comece pelo “main”
6. Não utilize o clássico exemplo “Hello Word”
7. Mostre a estrutura do programa
8. Seja cuidadoso com a interface do usuário.

Estas diretrizes sugerem que o professor inicie com a apresentação de projetos razoavelmente grandes, que os alunos irão executar, ler, modificar e estender. O estudo de caso facilita no ensino de programação [KOL03], enquanto que o desenvolvimento de um

projeto a partir do nada é um exercício muito avançado para um principiante. Desta forma, partindo-se de projetos prontos o entendimento dos conceitos é efetivado e a utilização é mais segura. BlueJ conta com livro texto produzido para o ensino, atualmente na segunda edição, *Objects First with Java - A Practical Introduction using BlueJ*. A primeira edição foi traduzida e lançada em maio/2004 pela Pearson Education do Brasil [KOL03].

BlueJ oferece para uma API extensível, o que permite a participação de colaboradores no desenvolvimento de extensões ao ambiente. Extensões oferecem funcionalidade adicional não incluída no sistema central. Maiores detalhes podem ser observados no endereço http://www.bluej.org/extensions/extensions.html.

3.4.5 Dr. Java

Dr. Java é uma interface simples baseada em *"read-eval-print loop"* que permite um programador desenvolver, testar e depurar programas em Java de forma interativa e incremental. Com enfoque pedagógico, esta IDE facilita a introdução à programação com Java, permitindo que os estudantes focalizem no projeto de programas, ao invés do aprendizado de como usar o ambiente [ALL01].

A interface foi projetada para minimizar os fatores que intimidam iniciantes a experimentar a codificação na linguagem. A interface consiste de uma janela com duas partes: interativa e de definição. A interativa permite ao aluno entrar com expressões Java e comandos e imediatamente ver seus resultados. A de definição é um editor que permite ao aluno criar classes com suporte ao casamento de chaves e realce da sintaxe e indentação automática. As duas partes são integradas com o compilador.

Usando Dr. Java os novos programadores podem escrever programas em Java sem lidar com problemas como I/O, interface em linha de comando, variáveis de ambiente como Classpath ou complexidades das interfaces de projeto suportadas por ambientes de desenvolvimento Java comerciais. A tela inicial é mostrada na Figura 3-19.

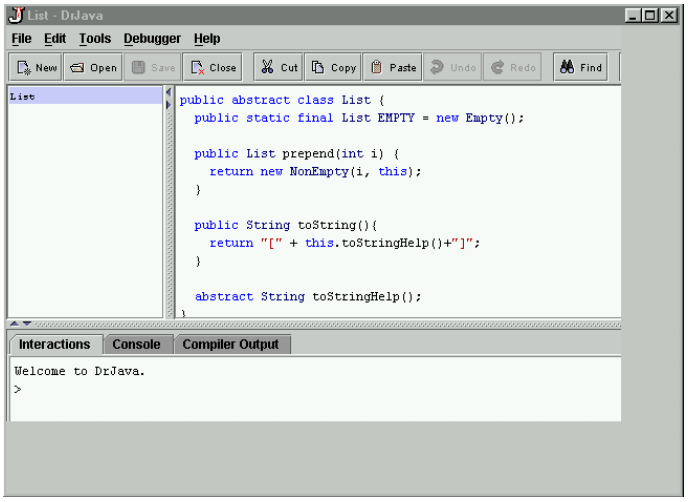

Figura 3-19 Tela inicial de Dr. Java

Como a maioria das IDEs, Dr. Java ajuda alunos a aprenderem a sintaxe da linguagem através de realce das formas sintáticas básicas da linguagem. Para manter a implementação "leve" não realiza análise incremental completa do texto do programa, mas sim uma análise incremental simples suficiente para reconhecer os principais elementos sintáticos: palavras chave, strings, comentários e várias formas de delimitadores.

No ambiente são integradas as ferramentas essenciais para desenvolvimento de software em Java: JUnit para testes de unidade, depurador e Javadoc para documentação. Dr. Java pode ser utilizada para desenvolvimento de projetos de tamanho considerável, como o desenvolvimento da própria IDE, que até 2003 estava com 40.000 linhas de código fonte. Além desta vantagem, é uma IDE que não exige muito do hardware. Porém, deixa de contar com ferramentas importantes, disponibilizadas por outras IDEs, complementação de código e ferramentas de navegação [REI04].

3.4.6 Jurtle

Jurtle é um ambiente de aprendizagem de programação integrado em Java, projetado para iniciantes. Este é composto por um editor de programas com facilidades para compilar e executar os programas [OTH04].

A motivação para a criação do ambiente foi a complexidade das ferramentas de desenvolvimento disponíveis no mercado.

Com Jurtle o usuário tem uma introdução a Java usando uma tartaruga (semelhante a dos ambientes LOGO). O nome Jurtle é resultado de Java+Tutle [OTH04]. Após uma

introdução com a tartaruga gráfica, Jurtle pode ser usado para criar aplicações simples que utilizam interfaces baseadas em texto, gráficas e applets Java.

Observando-se a tela na Figura 3-20 tem-se uma visão geral deste micromundo. É uma ferramenta que apresenta facilidades para o desenvolvimento de aplicações em Java e visualização do projeto em diferentes momentos nas abas Edit, Display e Errors. Como pode ser visto na seqüência na Figura 3-20.

Exemplos são listados à esquerda e os códigos fonte respectivos estão à direita, na aba edição. O código é executado a um simples clique do mouse no ícone correspondente e o resultado é visualizado na aba Display. Os erros podem ser vistos em Errors.

Este micro-mundo é também baseado nas idéia do micro-mundo LOGO. Sua classe básica é a classe Turtle. Através dos programas (classes) criados se estabelece a comunicação com a tartaruga. Para estabelecer esta comunicação existem comandos (ações) que ela consegue executar e comandos para realizar a entrada/saída de informação via console.

O método runTurtle é utilizado para iniciar o processo de execução, evitando assim a abordagem inicial à execução pelo método *public static void main...*, que pode ser deixado para outro momento, quando o entendimento estiver mais sedimentado.

As ações que a tartaruga consegue desempenhar estão listadas abaixo. Nesta listagem os comandos encontram-se divididos em comandos de movimentação, informação, desenho e atualização do display. Também são listados comandos relativos ao processamento de entrada e saídas disponíveis em telas padrões (interface texto).

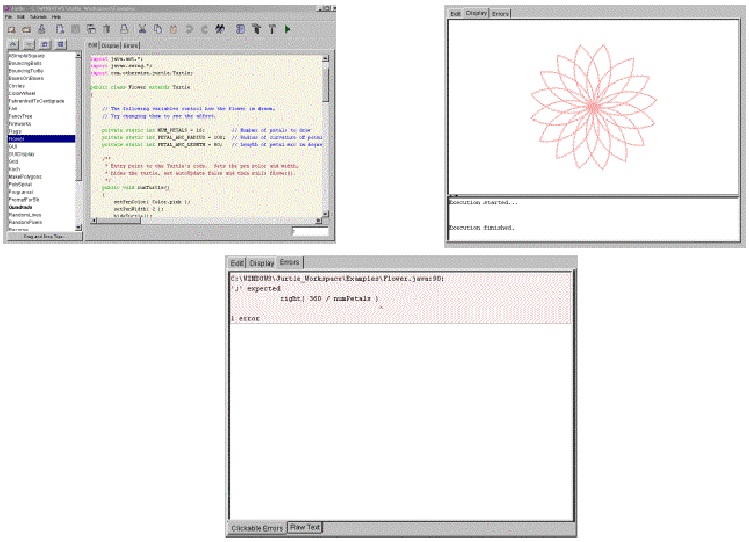

Figura 3-20 Visão geral do micromundo de Jurtle

As tabelas 3-1 e 3-2 resumem os comandos disponíveis no micromundo.

Tabela 3-2 Comandos da Tartaruga

Comandos da Tartaruga			
Tipo de comando	**Tipo de retorno**	**Comando**	**Significado**
Movimento	void	backward (distancia em pixels)	A tartaruga move-se para trás pela distância especificada.
	void	center ()	A tartaruga movimenta-se para o centro do painel de visualização.
	void	forward (distancia em pixels)	A tartaruga move-se para frente pela distância especificada.
	void	home ()	A tartaruga se desloca para seu estado padrão.
	void	left (graus)	A tartaruga gira para a esquerda a quantidade especificada de graus.
	void	right (graus)	A tartaruga gira para a direita a quantidade especificada de graus.
	void	setHeading (graus)	Coloca a tartaruga na direção de graus.
	void	setPosition (x,y)	Move a tartaruga para uma posição x, y especificada.
Informação	java.awt. Dimension	getDisplaySize ()	Retorna o tamanho do painel de display da tartaruga.
	java.awt. Point	getPos ()	Retorna a posição corrente da tartaruga.
	boolean	isVisible ()	Verifica se a tartaruga está visível.
Desenho	Void	clearDisplay ()	Limpa a tela mas deixa a tartaruga onde está.
	Void	hideTurtle ()	A tartaruga desaparece.
	Void	penDown ()	Abaixa o lápis da tartaruga. Assim ela deixa o traço ao movimentar.
	Void	penPaint ()	Volta a desenhar depois de apagar.
	Void	penUp ()	Levanta o lápis da tartaruga. Assim ela não deixa o traço ao movimentar.
	Void	setPenColor (cor)	Muda a cor da caneta para mudar a cor do traço.
	Void	setPenWidth (tamanho em pixels)	Muda a espessura da caneta para mudar a largura do traço deixado.
	void	setDisplayColor (cor)	Muda a cor do fundo do display.
	void	showTurtle ()	A tartaruga reaparece.
Atualizar	void	updateDisplay ()	Atualiza o display com qualquer desenho da tartaruga.
	void	setAutoUpdate (true/false)	Coloca um *flag* indicando se a atualização do display será automática ou não após cada comando da tartaruga que o afeta.
	void	setAutoUpdatePause () (tamanho da pausa em milisegundos)	Coloca o número de milisegundos que o sistema deve pausar depois de realizar uma atualização automática.

Tabela 3-3 Comandos de Console

Comandos Básicos de Console			
Tipo de comando	**Tipo de retorno**	**Comando**	**Significado**
Saída	void	print (item)	Imprime o item no painel de console. A entrada pode ser um char, int, long, float, double, String ou objeto.
	void	println (item)	Imprime o item no painel de console e posiciona no início da próxima linha. A entrada pode ser um char, int, long, float, double, String ou objeto.
Entrada	boolean	readBoolean ()	Lê um elemento do tipo boolean (true/false) do console.
	char	readChar ()	Lê um caracter do console.
	double	readDouble ()	Lê um double do console sem validar seu valor.
	float	readFloat ()	Lê um float do console sem validar seu valor.
	int	readInt ()	Lê um int do console sem validar seu valor.
	long		Lê um long do console sem validar seu valor.
	java.lang String	readLine ()	Lê uma linha de texto do console.
	java.lang. String	readWord ()	Pula os espaços em branco e lê a próxima palavra do console.

Novas classes podem ser criadas. Estas podem estender a classe Turtle ou não, podem conter o método main ou não. A Figura 3-21 mostra as opções para criação de novas classes.

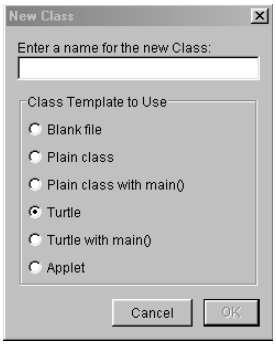

Figura 3-21 Criação de nova classe

O ambiente permite utilizar várias tartarugas ativas ao mesmo tempo. Outra vantagem é a utilização da área de exibição da saída onde se podem adicionar botões, caixas de textos e

outros elementos gráficos, facilitando assim, quando os alunos estiverem prontos, a utilização de partes da API Java.

A Figura 3-22 mostra uma aplicação onde cinco tartarugas movimentam-se na tela. Ao alimentar-se as tartarugas garantem a continuação do movimento. A alimentação é fornecida pelo clique do mouse. À medida que elas ficam um tempo sem encontrar comida elas vão paralisando. O estado em que se encontram após a execução pode ser visto na Figura 3-22.

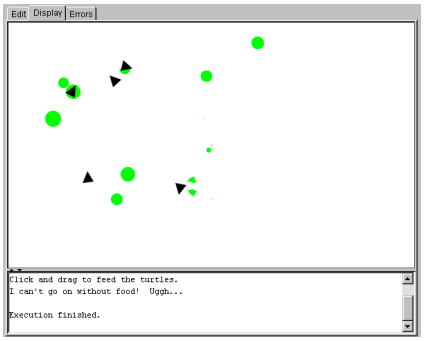

Figura 3-22 Fim do processo de execução do programa Turtle

3.4.7 Alice

Alice é um ambiente de gráfico de programação 3D projetado para uso na graduação por alunos sem experiência em programação ou gráficos 3D [CON99]. O objetivo inicial do projeto de pesquisa que resultou nesta ferramenta era criar uma nova ferramenta de autoria que tornasse acessível a manipulação de elementos 3D considerando um público não ligados diretamente a ciência ou engenharia, evitar a notação de vetores e matrizes quando possível e introduzir novas terminologias quando necessário e iterativamente testar os projetos com usuários reais, aumentando o entendimento e usabilidade do sistema no processo.

O ambiente pode ser utilizado em cursos introdutórios com a abordagem *"object-first"* com a vantagem de reduzir a complexidade de detalhes para programadores iniciantes, fazer uma abordagem *"design-first"* para o tratamento com os objetos e permitir a visualização de objetos em um contexto significativo [COO03].

Alice provê um ambiente onde os alunos podem usar e modificar objetos 3D e escrever programas para gerar animação. A interface mostra uma árvore de objetos, dos objetos envolvidos no mundo atual, a cena inicial, uma lista de eventos deste mundo e um editor de código. Um estudante adiciona objetos 3D em um pequeno mundo virtual e organiza o posicionamento de cada um no mundo. Cada objeto encapsula seus próprios dados e tem seus próprios métodos.

Ao utilizar este ambiente Cooper e outros [COO03] fizeram as seguintes observações:

- Os alunos demonstram um forte senso de projeto.
- Uma contextualização para objetos, classes e programação orientada a objetos.
- Uma abordagem de tentativas e erros.
- Uma construção incremental ao nível de classe e de método.
- Boas intuições sobre encapsulamento.
- O conceito de método como meio de solicitar que um objeto faça algo.
- Um forte senso de herança, ao escrever código para criar classes mais poderosas.
- Habilidade de colaboração em trabalhos em grupo.
- Um entendimento de tipos booleanos.
- Senso intuitivo de comportamentos e programação dirigida por eventos.
- Os alunos podem criar projetos do zero ou ampliar projetos existentes.

Merece destaque a sintaxe digirida por *templates* de Alice. O aluno preenche os *slots* a partir de escolha e preenchimento de caixas de texto. A Figura 3-23 mostra uma tela típica de Alice com as janelas de Movimentos de câmera, eventos, propriedades e descrição dos métodos.

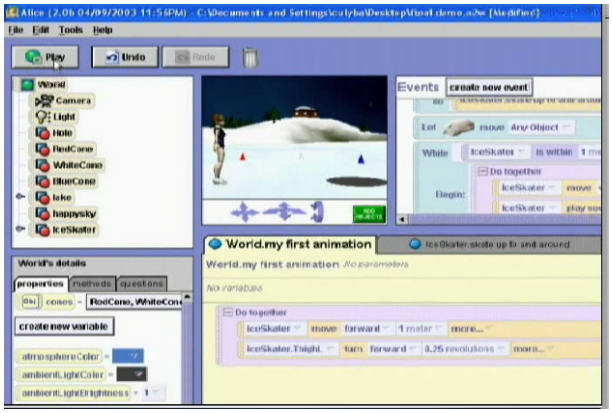

Figura 3-23 Tela típica de Alice

3.4.8 Greenfoot

Através da análise de benefícios de diferentes abordagens e combinando suas potencialidades, foi construído um sistema que oferece uma nova qualidade de experiência de aprendizagem e com benefícios de qualquer sistema individual disponível hoje. Originalmente inspirado na combinação de dois tipos de ambientes pedagógicos: micromundos, como Karel J. Robot e ambientes de interação direta como BlueJ, Greenfoot integra a força dos

micromundos, visualização dos objetos, seu estado e comportamento com a interação direta com objetos proporcionada em ferramentas como o BlueJ.

Karel J. Robot existe como um framework Java [BER02d] e BlueJ [BAR03] como um ambiente Java. É possível executar Karel dentro do BlueJ para ter os benefícios de ambos, porém confusões acontecem. Ambos permitem a visualização, porém de formas diferentes. Assim, ao utilizar estas duas ferramentas vários objetos são representados duas vezes (uma pelo BlueJ e outra por Karel J. Robot) e diferentes aspectos de visualização ou interação são propagados em diferentes visões do mesmo objeto.

Esta forma de abordagem pode gerar confusão para os iniciantes. Assim, o sistema Greenfoot foi desenvolvido para combinar as potenciais funcionalidades sem os problemas de representação [HEN04].

Greenfoot é um sistema interativo, visual, que permite experimentação, gera curiosidade, sem exigir substancial estudo teórico, o que facilita o envolvimento dos alunos e a motivação [HEN04]. A preocupação é facilitar a integração de ferramentas disponíveis, de forma que possa ser feita por profissionais sem conhecimento profundo na área. Assim, os professores passam a contar com um sistema interativo, com aplicações por simulações em um plano bi-dimensional [HEN04].

Greenfoot não é um micromundo, mas um meta-framework para micromundos, com uma interface consistente e modelo para permitir a interação entre diferentes cenários, de diferentes micromundos. Com esta facilidade, o professor pode colocar outros cenários (ferramentas) e utilizá-los em diferentes cursos [HEN04].

Os professores podem montar cursos completos usando as ferramentas disponíveis e ter a flexibilidade de utilizar vários cenários, facilitando assim a abordagem de conceitos e considerando diferentes abordagens pedagógicas já mencionadas neste trabalho. Além desta vantagem, os projetos desenvolvidos podem ser mantidos por uma equipe ou pessoa individualmente. Exercícios podem ser criados e compartilhados entre professores. Os projetos são fáceis de criar uma vez que toda a lógica de execução é provida pelo ambiente. Além disso, edição, compilação integrada, depuração e documentação gerada são fornecidas pela interface do BlueJ.

Nem todo professor precisa escrever seus próprios cenários, estes podem ser compartilhados, mas espera-se que a facilidade de criação no Greenfoot estimule a criação personalizada.

As Figuras 3-24 e 3-25 mostram a utilização dos micromundos da tartaruga e de Karel em Greenfoot.

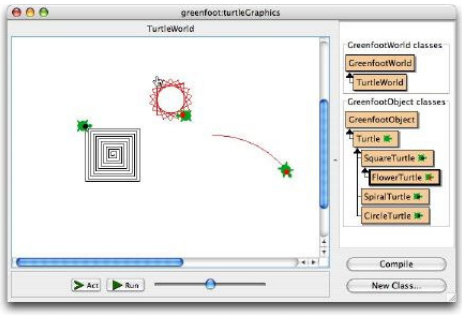

Figura 3-24 GreenFoot criando um micromundo da tartaruga

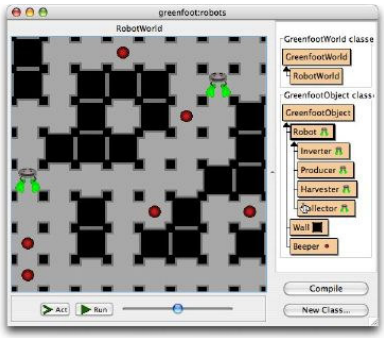

Figura 3-25 GreeFoot criando um micromundo com robôs como em Karel

3.5 CONSIDERAÇÕES SOBRE AS FERRAMENTAS

Programar a tartaruga no ambiente LOGO segue na maioria dos casos um modelo "procedural" de programação, onde o processo de ensinar a tartaruga deve conter todos os passos que a tartaruga deve executar para conseguir o resultado desejado. O modelo de como a máquina "funciona" está representado no papel da tartaruga, no sentido de que executa ações seqüencialmente. Neste contexto, para representar um problema para ser resolvido é necessário saber "o que" a tartaruga é capaz de fazer (primitivas); o que ela "deve" fazer para produzir uma figura na tela e "como" instruí-la a fazer. Esse é o paradigma imperativo de programação [BAR93]. O mesmo se passa com os variantes do LOGO com apenas uma tartaruga. Por outro lado, LOGO microworlds da LCSI, apesar de induzir conceitos de orientação para objetos não explicita estas construções na linguagem LOGO utilizada. Assim, conceitos fundamentais de orientação por objetos não podem ser trabalhados.

Em Karel, a linguagem utilizada para programar Karel (robô) é parecida com o Pascal. Os programas possuem instruções que são executadas pelo robô, que possui um vocabulário

bem limitado, podendo ser estendido, definindo-se novas instruções [PAU00]. Seus variantes utilizam a mesma filosofia e constituem, também, ambientes "procedurais".

Esses ambientes procedurais podem dificultar a introdução do enfoque de orientação a objetos. Segundo [ZAN96], um dos grandes problemas no ensino e, conseqüentemente, na adoção do paradigma OO baseia-se na influência sofrida por alunos e desenvolvedores de software, do paradigma estruturado, que geralmente é o primeiro paradigma de desenvolvimento de software aprendido e utilizado. Com este, e outros problemas, temos como conseqüência direta a perda dos benefícios do mesmo como, por exemplo, facilidade de manutenção, abstração de dados, etc.

As propostas dos ambientes Karel++ e JAREL surgiram para tratar os conceitos de orientação por objetos. As regras do mundo Karel foram alteradas e a linguagem estendida. Neste novo mundo podem coexistir múltiplos robôs. Os robôs possuem nomes, podendo assim, receber instruções individualmente. As primitivas conhecidas por todos os robôs são as mesmas definidas no ambiente original: move, turnLeft, turnOff, pickBeeper e putBeeper. A descrição de uma instrução do robô é o *método*. A definição de uma classe é composta por uma lista de instruções. A palavra "void" prefixa cada instrução para indicar que, quando executadas, não retornam variáveis. O conjunto de mensagens de um ou mais robôs é chamado *tarefa*. Nestes ambientes, o vocabulário dos robôs também pode ser estendido. Novos métodos e classes podem ser criados. Os robôs da nova classe herdam todas as capacidades da classe superior [PAU00].

Karel J. Robot tem a vantagem da excelente visualização do comportamento dos objetos e de seus estados. Nada melhor para ensinar os conceitos básicos de orientação por objetos como é feito no livro que acompanha a ferramenta [BER02d]. Os conceitos de poli morfismo, herança, reuso são facilmente abordados. Porém, sua carência é não ter um meio de fazer uma interação direta com os objetos como pode ser feita com o BlueJ, ou seja, inspecionar o objeto e solicitar a execução de serviços diretamente e não através da execução do código. Se no mundo do robô além desta nova funcionalidade fosse disponibilizada a possibilidade de trabalhar com a classe geradora de robôs e estender para utilizar outros elementos como tartarugas, peixes, computadores entre outros, o micromundo ficaria mais adaptável, com outras formas de vivenciar os conceitos, como foi desenvolvido no GreenFoot

BlueJ por outro lado, tem a facilidade de fazer a diferenciação entre classe e objeto, o que é uma grande contribuição. Além disso tem a possibilidade de interagir com objetos na bancada de objetos e inspecioná-los, tendo contato com o diagrama do objeto, que pode ser

inspecionado recursivamente. Através da solicitação de serviços diretamente aos objetos pode-se verificar como a chamada é codificada em Java e abordar conceitos como tipos primitivos de dados, parâmetros, tipo de retorno entre outros. Porém falta a visualização do comportamento ou estado do objeto. A visão que se tem é estática, a dinâmica da execução não pode ser sentida de forma gradual como acontece em Karel J. Robot.

O ambiente AllKara permite uma introdução à programação de computadores utilizando-se diferentes ferramentas. Como visto, este micromundo é habitado por **Kara**, um inseto, que divide seu mundo quadriculado com folhas, troncos e cogumelos. O objetivo deste ambiente é introduzir formalismos da Ciência da Computação como máquina de estados finitos, máquina de Turing, programação em Java e programação concorrente. Com a ferramenta Kara os programas são elaborados através da montagem de máquina de estados finitos e suas tabelas de transições. Com JavaKara, os programas são elaborados na linguagem Java, permitindo uma introdução a conceitos como objetos, mensagens, classes, estruturas de controle, modularização entre outros. Esta é uma ferramenta que permite analisar a programação em diferentes aspectos, suas potencialidades e limitações. Como ferramenta introdutória é muito útil, porém para tratar outros conceitos como herança, polimorfismo, é necessária uma extensão, pois os objetos existentes no mundo de Kara já estão criados e a própria classe Kara não pode ser instanciada. É uma boa ferramenta para se introduzir estruturas do paradigma imperativo como seleção, repetição, criação de novos métodos, mas não para uma introdução com abordagem centrada em objetos.

Jurtle é um ambiente integrado, que facilita a introdução à programação e a abordagem de tópicos avançados. Porém, a relação entre classe e objeto não é tão clara como ocorre com a utilização do BlueJ. Assemelha-se ao ambiente Karel J. Robot, em termos da possibilidade de verificação da execução e possibilidade de trabalhar com aprendizagem significativa, ou seja, aquela em que as novas idéias vão se relacionando de forma lógica, explícita e substantiva com as idéias existentes na estrutura cognitiva do indivíduo [BRA87]. Assim, se nos projetos desenvolvidos pelos alunos forem trabalhados os padrões de projeto e os padrões pedagógicos em paralelo a outras ferramentas, como o BlueJ, Jurtle torna-se uma fonte de experimentação e motivação para a fluência no mundo dos objetos.

Em Alice, um ponto fraco é que, embora possa ser útil nos estágios iniciais, o aluno não desenvolve a compreensão da sintaxe com C++ ou Java, uma vez que estes são inseridos pela janela de código. Os alunos não têm a oportunidade de experimentar situações de erros ao programar. Porém, a experiência com alunos usando este ambiente mostra que a transição

de Alice para C++ ou Java é tranqüila, uma vez que normalmente os alunos facilmente dominam a sintaxe [COO03].

3.6 COMPARAÇÕES ENTRE IDES

As IDEs pedagógicas, AllKara e BlueJ e IDEs profissionais, Eclipse e NetBeans foram utilizadas pela autora na disciplina Tópicos Especiais I, do curso de Sistemas de Informação, como parte do processo de avaliação das ferramentas. Detalhes destas experiências estão no Capítulo 5. Uma comparação entre as IDEs é feita na tabela abaixo:

Tabela 3-4 Comparações entre as IDEs utilizadas

Características de uma IDE	Eclipse	NetBeans	AllKara (JavaKara)	BlueJ
Teste de unidade	Sim	Sim	Não	Sim
Portabilidade	Sim	Sim	Sim	Sim
Depurador	Sim	Sim	Não	Sim
Suporte a documentação JavaDoc	Sim	Sim	Não	Sim
Representação do relacionamento entre classes	Não	Não	Não	Sim
Curva de aprendizagem	Alta	Alta	Baixa	Baixa
Custo	Gratuito	Gratuito	Gratuito	Gratuito
Código aberto	Sim	Não	Não	Não
Extensível	Sim	Não	Não	Sim
Micro-mundos	Plugin para Bluej e Plugin para Dr. Java	Não	É-um	Existe implementação do pacote Turtle Graphics

O que se conclui do estudo destas ferramentas e IDEs didáticas é que nenhuma delas satisfaz todos os critérios apontados anteriormente como desejáveis para tais ambientes. É provável que a combinação de algumas delas seja a melhor saída para a criação de um ambiente rico e flexível para a aprendizagem de programação de computadores usando o enfoque *object-first*. É o que se procura demonstrar no Capítulo 5 com a proposta da Estação OO.

Capítulo 4

4 Padrões Pedagógicos e Padrões de Projeto

4.1 INTRODUÇÃO

Um dos problemas que professores de Ciência da Computação e professores de modo geral enfrentam, é que o processo de ensinar não é simplesmente estar em uma sala de aula e dar sua aula em alguns minutos [HOW96], é preciso utilizar ferramentas e técnicas educacionais para intensificar a experiência de aprendizagem do estudante.

A Taxonomia de Bloom [FEL88a] tem sido considerada pelos educadores como uma referência para medir o nível de entendimento dos alunos em uma disciplina particular [BEL95]. Esta considera os estilos de aprendizagem tanto de quem ensina quanto de quem aprende já que estudantes e professores têm estilos de aprendizagem específicos, no qual sentem-se confortáveis para passar e receber informações [FEL88a]. Professores que concordam com esta idéia e a consideram na apresentação de seus cursos, procuram privilegiar todos os estilos de aprendizagem e têm efetivamente aumentado a eficácia de suas aulas [FEL93].

Os estilos de ensino/aprendizagem considerados por Felder [FEL88b] classificam os indivíduos quanto à percepção em sensitivos ou intuitivos, quanto ao tipo de informação melhor processada em visual ou verbal, quanto à organização em indutivo ou dedutivo, quanto ao tipo de processamento em ativo ou passivo e quanto à compreensão em seqüencial ou global. Segundo o autor, para que o processo ensino/aprendizagem seja satisfatório deve existir uma compatibilidade entre o estilo de ensinar do instrutor e o estilo de aprendizagem do aluno [FEL98]. Alcançar esta compatibilidade em turmas grandes e normalmente heterogêneas é praticamente impossível. Assim, faz-se necessário lançar mão de recursos que permitam que a exposição do assunto seja feita em estilos variados com o objetivo de atingir os diferentes estilos de aprendizagem.

O quanto o aluno aprende é determinado por habilidades natas, por experiências anteriores (base) e pela coincidência entre o estilo de aprendizagem e o estilo de ensino do instrutor. Como pouco se pode fazer quanto às habilidades, experiências anteriores ou estilos

de aprendizagem, para maximizar a aprendizagem dos estudantes, se deve investir no estilo de ensinar do instrutor [FEL88a].

Outro modelo é o Ciclo de Aprendizado de Kolb, já citado, similar aos estilos de aprendizagem de Felder, descreve como o aluno aprende. Os trabalhos de Harb e Terry mostram que ensinar através do ciclo de Kolb é uma forma efetiva de alcançar os estudantes na sala de aula [HAR92, HEN04].

Pesquisa na área de estilos cognitivos e estilos de aprendizagem são ainda incipientes. As conclusões destes estudos devem ser usadas com cuidado. No entanto, estes aspectos são relevantes para qualquer planejamento de um curso, a medida que força o projetista a tentar acomodar diferentes visões e estilos na formulação do curso, aumentando assim as possibilidades de sucesso.

Considerando os problemas e soluções referentes ao processo de ensino e aprendizagem e os estudos relacionados aos estilos de aprendizagem, padrões pedagógicos têm sido propostos [BER02] para cursos de ciência da computação, podendo ser usados para cursos de modo geral.

Padrões não refletem a invenção de um autor particular, mas conhecimento que tem funcionado em muitas situações. Os padrões capturam as melhores práticas em determinado domínio e as organiza de modo a facilitar a comunicação com outras pessoas. São utilizados para resolver problemas parecidos. Na verdade é um vocabulário de soluções, que pode ser usado em conjunto com outros padrões. Em essência, um padrão resolve um problema recorrente [BER02].

No ensino temos muitos destes problemas como motivação de alunos, escolha e sequenciação de material didático, avaliação de alunos e outros aspectos relacionados. Cada vez que um problema emerge existem considerações novas que devem ser feitas e que influenciam a escolha das soluções. Padrões especificamente endereçam o contexto em que o problema recorre junto com forças que devem ser aplicadas para ter uma boa solução para o problema.

Além dos padrões pedagógicos, são apresentados os padrões de projeto orientados a objetos (*design patterns*). Como os primeiros, estes podem facilitar o trabalho de quem precisa "ensinar" e o trabalho de quem quer aprender com qualidade. No contexto de projeto, padrões são soluções recorrentes aos problemas que confrontam projetistas. Cada padrão encapsula um problema bem definido e uma solução padrão. Esta solução consiste de ambos: um projeto e uma descrição de como implementá-lo. Por entrelaçar o problema, a solução e a

implementação, padrões permitem ao projetista refiná-los concorrentemente durante o desenvolvimento do sistema. O uso de padrões permite um novo nível de abstração no qual análise, projeto e implementação podem ser feitos [WAL96].

A experiência faz a diferença entre um programador brilhante e principiante e um programador brilhante e experiente. A experiência para um profissional brilhante traz sabedoria. À medida que programadores ganham experiência, eles reconhecem a similaridade entre novos problemas e problemas que já foram resolvidos. Com mais experiência, eles reconhecem que as soluções para problemas similares seguem padrões recorrentes. Com o conhecimento destes padrões, programadores experientes reconhecem situações às quais os padrões se aplicam e podem imediatamente usar a solução sem ter que parar, analisar o problema e definir possíveis estratégias.

Colocar um padrão em palavras permite que ele seja compartilhado, promovendo a discussão entre profissionais que conhecem padrões e a combinação com outros padrões. Além disso, é um benefício adicional para programadores menos experientes, pois é forma de comunicar a sabedoria adquirida [GRA98].

São duas forças que se unem para melhorar a abordagem em cursos introdutórios. Os padrões pedagógicos contribuem com o esforço em facilitar o "como ensinar" e os padrões de projeto que contribuem com o esforço de "como utilizar" da melhor forma os conceitos de orientação a objetos para ter boas soluções para os problemas do dia a dia.

Nas sessões seguintes serão abordados padrões pedagógicos, de projeto e de codificação para iniciantes, com o formato baseado no modelo de Christopher Alexander, um arquiteto que foi a inspiração para a idéia de padrões de software em *A Pattern Language* [GRA98,GRA02].

4.2 PADRÕES PEDAGÓGICOS

Os padrões pedagógicos tentam capturar conhecimento especializado de anos de prática de ensino e disponibilizá-los de forma compacta, coerente e acessível [BER01a]. Este tipo de material aproveita anos de experiências de professores e facilita a troca de experiência entre grupos de professores.

Com o objetivo de trocar experiências com relação a técnicas de ensino e aprendizagem, projetistas de universidades e da indústria participam do projeto de Padrões

Pedagógicos [BER01a] que gerencia e administra documentos que registram padrões efetivos de instrução e aprendizado.

Os padrões pedagógicos não estão todos no mesmo nível de escala. Alguns falam da organização do curso como um todo em cursos semestrais e alguns em escala menor em atividades diárias. Nas sessões seguintes alguns padrões pedagógicos são descritos, estes são encontrados em [BER02].

Cada padrão começa com um nome, em seguida uma breve descrição do problema ao qual o padrão se dirige, podendo ser descritas situações que conduzem à existência do padrão. Através desta descrição pode-se verificar se o padrão é adequado ou não para um determinado problema.

Uma seção iniciada com **Portanto**... é a introdução da solução e descreve como a aplicação do padrão deve ser feita. Pode haver informações adicionais sobre o padrão, como exemplos práticos.

A seguir serão apresentados os padrões pedagógicos mais relevantes para o ensino de programação.

4.2.1 Abordagem

A comunicação sempre acontece entre um transmissor e um receptor, e a efetivação da comunicação não é medida pelo que o transmissor disse, mas pelo que o receptor entendeu. Toda pessoa tem sua forma própria de absorver informações, usando diferentes modalidades sensoriais.

Portanto é importante prover diferentes abordagens para o mesmo tópico. Pode ser difícil diversificar cada tópico e seus sub-tópicos, mas diferencie pelo menos tópicos maiores. As mudanças de abordagens ajudam a manter o interesse dos alunos, uma vez que eles não estão acostumados com um determinado estilo, constituindo um desafio a experiências com outras habilidades.

Atividades diferentes estimulam, despertam interesses e podem até revelar talentos desconhecidos.

Ao introduzir um novo tópico mostre onde ele se encaixa. É muito importante estabelecer conexões para que o tema se encaixe na rede de conhecimentos do aluno e possa ser efetivamente apresentado e entendido. Enfatize os pontos chaves, discriminando-os para que os alunos possam se guiar por eles. As atividades de jogos, simulações, jogo de papéis podem ser usados para atingir os alunos que aprendem pela ação.

4.2.2 Metáfora Consistente

É muito importante, principalmente para iniciantes, ter uma idéia de onde o tópico que está sendo abordado se encaixa. O uso de metáfora é muito interessante neste contexto, pois associa algo novo com algo que já se conhece e domina.

Portanto com uma boa metáfora o instrutor pode fazer inferências referenciando-se a metáfora e deixando claro para os alunos o relacionamento entre a metáfora e o tópico de estudo. Com uma boa metáfora o instrutor fornece aos alunos uma ferramenta consistente e poderosa para pensar sobre um tema complexo.

Abordar um tópico, como pensamento orientado a objetos, que é muito amplo e com muitos conceitos novos pode se tornar bem mais simples com o uso de uma metáfora. Ao ensinar sistemas orientados a objetos, uma boa metáfora é o ser humano. Pessoas são autônomas, atores encapsulados, que interagem com outros objetos similares [BER00a]. Entretanto, sempre deixe claro o limite da metáfora com a realidade. A metáfora de "pessoas como objetos" termina quando ao passar uma mensagem o receptor não tem como comunicar com o emissor, como acontece na prática.

4.2.3 Analogia Física

Conceitos abstratos são mais difíceis de serem assimilados que conceitos concretos. O uso de uma analogia com algo físico pode auxiliar no entendimento. Após a abordagem "teórica" do conceito pode-se partir para uma dinâmica para entender o conceito abstrato.

Em orientação a objetos um exemplo é a compreensão da execução do código. Alunos normalmente seguem o código de cima para baixo (top-down) e tentam visualizar o que acontece na execução de cada comando. Porém, código orientado a objeto não é executado de forma top-down. Além desse exemplo, outros princípios como o da natureza privada dos dados e pública dos métodos pode parecer obscura para principiantes. Se o instrutor não se esforçar em solidificar princípios importantes como estes, o aluno pode ignorá-los e escrever programas orientados a objetos utilizando técnicas procedurais.

Portanto ilustre as propriedades dinâmicas de conceitos abstratos de forma concreta. Crie uma analogia física com algo visual como objetos, pessoas, cenários reais entre outros.

Este padrão pode ser usado para explicar a forma como é feita a solicitação de serviços a objetos, realizando uma atividade onde cada aluno comporte-se como um objeto, com funções bem definidas. Os objetos interagem solicitando serviços uns dos outros para que determinados objetivos sejam alcançados. Quando ocorre a solicitação de um serviço para um

determinado objeto, este o executa, podendo ele mesmo solicitar outras tarefas ou simplesmente retornar à seqüência anterior. Outro exemplo é o conceito de métodos de acesso (get) e alteração (set). Neste caso pode-se montar atividades onde o aluno é considerado objeto e suas informações (privadas) deverão ser acessadas pelos seus métodos públicos que podem acessar ou alterar dados internos.

4.2.4 Jogo de Papéis

Considerando que conceitos abstratos normalmente não são entendidos com explicações abstratas e que mesmo com ambientes favoráveis de aprendizagem, em muitos casos, é necessário algo mais. Os jogos permitem que o lúdico auxilie na aprendizagem de conceitos.

Portanto convide os alunos a participar de atividades onde eles se comportam como os conceitos envolvidos através de um jogo de papéis. Todo aluno presente participa do jogo como uma parte do conceito a fim de ter um aprofundamento do conhecimento dentro da estrutura apresentada. Esta atividade permite ver como diferentes partes dos conceitos trabalham juntas para resolver um problema maior.

Os jogos de papéis permitem que os alunos percebam o que estão entendendo e simultaneamente confronta-os com seus conhecimentos prévios. Se eles entendem bem seu papel, então eles saberão como e com quem interagir. Para encorajar este entendimento, permita que os alunos troquem de papel ocasionalmente, permitindo vivenciar diferentes pontos de vistas do sistema.

Para montar o jogo de papéis são necessários scripts onde se descreva a seqüência de ações a serem executadas e um cartão CRC para cada tipo de objeto que participa do jogo.

O script de cada ator possui as seguintes informações

- Nome da classe a qual ele pertence;
- Informações fornecidas quando o ator (objeto) for criado;
- Ações que o ator deverá executar em negrito.

Cada aluno recebe um script e este deve ser conhecido para que o papel seja bem desempenhado. Durante o jogo um participante pode receber mensagem que não conhece, nesta situação este deve dizer “Não sei como fazer”. Também pode receber mensagem mal formada, por exemplo, com falta de parâmetro.

O instrutor possui um roteiro para coordenar a execução das tarefas. Este roteiro pode ser adaptado e cobrir conceitos específicos.

Este padrão será abordado com maior detalhe no Capítulo 5.

4.2.5 Reflexão

Os alunos na maioria das vezes acreditam que o instrutor tem que passar todo o conhecimento, e passivamente esperam aprender. Segundo eles, os instrutores resolverão todos os problemas e caso não resolva logo assumem que o instrutor possui um conhecimento limitado. É preciso fazer com que os alunos acreditem em seu próprio potencial e não se satisfaçam em aceitar o que lhes é transmitido, que corram em busca das soluções de seus problemas, que reflitam sobre suas soluções, busquem alternativas, melhorem seus trabalhos, enfim sejam reflexivos e atores de seu próprio aprendizado.

Portanto prepare um ambiente que permita descobertas e que não seja limitado a perguntas e respostas. É preciso encorajar o aluno a buscar soluções para problemas complexos baseados em sua própria experiência. Treine os alunos a buscar soluções pela exploração do problema.

Redirecione o aluno para usar seu próprio intelecto. Alunos precisam encontrar respostas por eles mesmos. Mesmo que o aluno venha com uma pergunta cuja resposta não seja conhecida, desafie-o. Para um bom problema proposto existe uma boa solução a ser procurada e com certeza o aluno que formula o questionamento pode refletir e dar uma ótima solução. É preciso mostrar-se interessado na resposta e reforçar o quanto é difícil e interessante a questão levantada.

Mostre o que é realmente a aprendizagem. Mostre as coisas que os alunos precisam aprender e pergunte como eles imaginam que podem usar o que estão aprendendo no ambiente de trabalho. Solicite que descubram as diferenças e similaridades das experiências.

Jovens adultos quando entram em treinamento industrial podem não perceber que seu estilo de aprendizagem foi amadurecido, além do estilo pedagógico ao qual estavam acostumados nas faculdades. Eles devem saber que podem se beneficiar muito com diferentes estilos de aprendizagem. Adultos são aprendizes baseados nas competências, isto significa que querem aprender habilidades ou conhecimentos que possam aplicar em circunstâncias correntes.

4.2.6 Explore por você mesmo

O sucesso de uma pessoa depende principalmente de sua capacidade de aprender novos conceitos de forma eficiente e compartilhar este conhecimento na equipe da qual participa. É necessário proporcionar aos alunos a habilidade de aprender no futuro e

comunicar seu saber, porém os alunos normalmente receiam em tomar a responsabilidade de seu próprio processo de aprendizagem.

Portanto proponha atividades onde os alunos precisem estudar um determinado tópico, aprender por eles mesmos e apresentar este tópico. Este é um bom exercício. A apresentação do tópico a alguém mais é uma parte importante do processo de aprendizagem. Um aluno que explica um conceito a outras pessoas acaba o entendendo melhor [BER01].

Algumas dificuldades serão apresentadas pelos alunos, mas com orientação e reflexão podem ser resolvidas. Treinamento e preparação são essenciais. A situação é parecida com o condicionamento físico, treinando supera-se os próprios limites.

4.2.7 Espiral

Existe um relacionamento entre os tópicos de um curso. Alguns deles são necessários para ter ferramentas básicas para resolver problemas interessantes. Abordar os tópicos de modo a seguir uma seqüência lógica do curso pode ser cansativo para os alunos.

Portanto organize o curso de modo a introduzir tópicos sem cobri-los totalmente. Em um primeiro ciclo é feita uma introdução simples, sem deixar os detalhes essenciais, cobrindo vários tópicos rapidamente. Desta forma os alunos podem trabalhar em problemas interessantes rapidamente e ter mais ferramentas para usar. Em cada ciclo da espiral, cobre-se tópicos em mais profundidade e acrescenta tópicos.

Tópicos grandes como projeto e programação possuem muitos detalhes. O desenvolvimento dos tópicos relacionados de forma seqüencial deixa os alunos sem interesse pelos exercícios uma vez que não percebem uma relação entre os tópicos e muitas vezes não permite a criação de projetos interessantes para serem desenvolvidos com os conceitos.

Alunos gostam de construir coisas e de ver os pedaços se conectarem e tomarem forma juntos. Os cursos não precisam ser organizados como material de referência, nem como livros textos. Este padrão tem como resultado uma abordagem mais refinada dos tópicos nas disciplinas, com quantidade suficiente de tópicos sendo introduzidos a cada estágio de modo a permitir a resolução de problemas com outros tópicos ou ferramentas juntos, sem serem completamente cobertos. Por exemplo, “repetição” é um grande tópico. O tópico “repetição usando *while*” é menor, especialmente se introduzido com um único teste. Quando a repetição com *while* estiver sendo usada efetivamente com outros construtores e gerando programas significativos o tópico “repetição” poderá ser entendido com facilidade em todos os seus aspectos.

4.2.8 Ligando o novo ao velho

Aprender algo novo é excitante e normalmente envolve coisas que o aluno já sabe. Aprender muitas coisas ao mesmo tempo pode levar a um sentimento de rejeição e ao stress.

Portanto utilize uma "embalagem" velha para introduzir um produto novo. Desta forma o aluno poderá reconhecer o que já conhece e fazer associações entre o conhecimento novo e o existente.

A ligação entre o conhecimento antigo e o novo ajuda a organizar a estrutura do conhecimento e facilita na aquisição de novos conhecimentos. É através das conexões que estabelecemos que conseguimos armazenar grande quantidade de informação em nosso cérebro.

4.2.9 Tubo de Ensaio

Quando os alunos encontram buracos nos próprios conhecimentos, o ideal seria que eles mesmos consigam tapá-los, dando as respostas aos seus próprios questionamentos. Infelizmente, alunos imediatamente procuram alguém para tirar as dúvidas, um manual ou algo do gênero e na maioria das vezes desconsideram as possibilidades que têm de adquirir conhecimento por si mesmos.

A maioria das suas perguntas eles mesmos podem responder. Imagine que a documentação esteja incompleta e que não encontre alguém para tirar as dúvidas? Como fazer?

Programação é diferente de muitas outras disciplinas. Podem-se escrever programas e ver seu comportamento. Na observação de seu comportamento, podem-se inferir respostas a muitas perguntas sobre linguagem de programação, sobre o compilador ou interpretador e sobre construtores diferentes da linguagem. Normalmente este tipo de experiência é uma forma mais rápida e efetiva que o questionamento a um instrutor ou leitura de um documento. O mais importante é preparar o aluno para aprender de forma independente em várias situações. Alunos sentem-se livres quando eles aprendem a encontrar respostas para seus próprios questionamentos rapidamente.

Portanto proponha aos alunos elaborar programas pequenos e que usem o computador para responder perguntas do tipo "O que aconteceria se...". Faça este tipo de exercício freqüentemente para que os alunos criem o hábito de sondar na máquina resultados ou respostas de suas dúvidas.

4.3 PADRÕES PEDAGÓGICOS E IDES

Considerando os principais padrões pedagógicos apresentados anteriormente, foi feita uma análise da ocorrência de cada um nas principais IDEs didáticas mostradas no Capítulo 3.

Veja a tabela abaixo.

Tabela 4-1 Ocorrência dos Padrões Pedagógicos nas IDEs estudadas

Padrões pedagógicos	AllKara/JavaKara	Jurtle	BlueJ
Abordagens	Sim	Sim	Sim
	O micromundo em si é uma abordagem diferente. Permite elaborar situações e construir soluções sem exigir exposição de conceitos prévia.	O ambiente é fácil de utilizar e é baseado no micromundo da tartaruga, o que representa uma abordagem totalmente diferenciada. Além disso facilita a manipulação de elementos gráficos que estimula os alunos.	No BlueJ é feita uma ponte entre o modelo conceitual da classe (diagrama e código fonte) e os objetos. Esta abordagem às apresentadas em cursos tradicionais de OO com Java.
Metáfora Consistente	Sim	Sim	Sim
	Os micromundos AllKara e Jurtle são metáforas. Todas as IDEs permitem criar projetos que implementem metáforas.		
Analogia Física	Sim	Sim	Sim
	A analogia física se refere à associação de objetos criados a pessoas ou coisas do mundo real. Este elo não depende de software, mas associações podem ser feitas em qualquer uma das IDEs.		
Jogo de Papéis	Não relacionado	Não relacionado	Não relacionado
	O Jogo de Papéis é uma atividade prática que independe de computador. No BlueJ pode ser feita uma extensão para facilitar a elaboração e simulação do Jogo. Esta proposta é apresentada no Capítulo 6.		
Reflexão	Sim	Sim	Sim
	Em todas as IDEs o instrutor pode criar desafios, propor alterações e conduzir os alunos à reflexão.		
Explore por você mesmo	Sim	Sim	Sim
	Em todas as IDEs os alunos podem fazer suas investigações.		
Espiral	Sim	Sim	Sim
	Em todas as IDEs os alunos podem trabalhar determinados conceitos e revê-los posteriormente. Nesta revisão pode-se aumentar a profundidade da abordagem, mudar a abordagem, conectar tópicos, refletir, enfim trabalhando com o padrão Espiral utiliza-se vários outros.		
Ligando o novo ao velho	Sim	Sim	Sim
	Esta também é uma questão de metodologia de ensino e pode ser contemplada em todos as IDEs.		
Tubo de ensaio	Sim	Sim	Sim
	Todas as IDEs são potenciais laboratórios onde os alunos podem experimentar, tentar, errar e acertar e tirar suas próprias conclusões.		

4.4 PADRÕES DE PROJETO ORIENTADOS A OBJETOS

A abordagem de padrões de projetos em cursos introdutórios de programação orientada a objetos tem sido usada com sucesso [WAL96, WAL98, BER01c, BER02d, LEW04].

Um dos trabalhos mais influentes em padrões de projeto foi o livro *Design Patterns* de Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides publicado em 1994. *Design Patterns* é normalmente referenciado como o livro da *"Gang of Four"* ou "GoF". Este livro popularizou a idéia de padrões. Seus exemplos eram codificados em C++ e Smalltalk, sendo que a UML ainda não existia na época. Trabalhos posteriores, como os de Mark Grand e Bruce Eckel, utilizam Java e UML, amplamente aceita como uma notação preferida para análise e projeto orientado a objetos [GRA98, GRA02, ECK03].

O conceito de padrões de projeto, *design patterns*, tem se tornado vital para aprender programação orientada a objetos e projeto (OOP&D), uma vez que permite a concentração em bons projetos orientados a objetos, constituindo um bom exemplo a ser seguido [LEW04].

Design patterns foi elevado ao status de tópico central no currículo de Ciência da Computação em [AST98], e no Currículo de Computação 2001 [ACM01] este tópico foi incluído como parte principal de SE1 (Software Design).

Padrões de projeto têm sido usados para ajudar a organizar o currículo de cursos introdutórios em programação [WAL98].

Uma proposta de Lewis [LEW04] é uma reorganização curricular utilizando o padrão pedagógico espiral para abordar o tópico *design patterns*. Nos dois primeiros cursos (CS1 e CS2) os alunos vêem *design patterns* como parte de uma biblioteca de classes que os professores provêem para os alunos. O estudo explícito de *design patterns* fica para um terceiro curso de programação. Na abordagem em espiral, um *design pattern* é introduzido e usado em programas e classes no primeiro curso. Neste nível, um nome pode ser associado ao padrão, mas ainda não se discute porque a solução é apropriada. O padrão é usado novamente em um segundo curso, normalmente em um novo contexto, mas para resolver um problema similar ao anterior. Depois, no curso de projeto de software, é feita uma introdução rigorosa usando referências na área como GOF ou Buschmann [LEW04].

A intenção é que o estudo de padrões seja algo natural em programação de computadores e projeto e não como um tópico especial, avançado [WAL98].

Em cursos introdutórios de programação contextualizada, que primeiro introduzem modelos conceituais e a partir destes modelos apresentam a programação como forma de

representação para o computador, os padrões de projeto e codificação podem ser inseridos na transformação do modelo conceitual (em UML) para a implementação do código [BEN04].

Muitos cursos que introduzem metodologia de objetos focalizam somente certos aspectos de programação do paradigma como classes, objetos e herança e não conceitos de projeto que permitem a visão global. Esta abordagem proporciona ao aluno uma introdução incompleta que inibe o entendimento adequado de projeto orientado a objetos, mesmo se o conteúdo é apresentado em cursos posteriores [LEW04].

Em Lewis [LEW04] é identificado um subconjunto de padrões de projeto através de uma consulta a sessenta especialistas em projeto orientado a objetos, incluindo professores, engenheiros de software e cientistas com anos de experiências em projetos tanto em padrões pedagógicos quanto em padrões de projeto. Este subconjunto de padrões pode ser usado como ferramenta instrucional em cursos de graduação, considerando sua simplicidade para os iniciantes, promoção de suporte suficiente para programação e relevância para experiências futuras. Além da identificação deste subconjunto de padrões, os profissionais envolvidos na pesquisa consideraram a ordem e a granularidade da abordagem, pois impactam sobre a utilidade e efetividade em cursos em que são ensinados, assim como em cursos subseqüentes. São trinta padrões que foram selecionados e categorizados em grupos por similaridade [LEW04] e podem ser encontrados na referida bibliografia.

Segundo Wick [WIC00] *design patterns* fazem parte do arsenal de conhecimento dos desenvolvedores de software de hoje. *Design Patterns* é um tópico de desafio para os educadores da área de computação. Além de ser um tópico relativamente novo, é muito diferente de ensinar estruturas de dados e algoritmos [HAM04]. Ao estudar as estruturas de dados a necessidade da solução é evidente e o estudo pode ser feito no contexto. Porém, nada diz ao programador que seu código precisa de um certo *design pattern*. A não ser que o programador já conheça o *pattern* e consiga visualizar sua aplicação. Estruturas de dados são soluções em busca de um problema. *Design patterns* são ferramentas para fazer projeto, isto é, gerar soluções. Para aprender um *design pattern* o aluno deve experimentar usar o padrão na prática e relacioná-lo a outras técnicas.

Como educadores da Ciência da Computação é preciso desenvolver e compartilhar aplicações envolvendo *design patterns* que sejam suficientemente claras e adequadas para serem incluídas em cursos introdutórios e que demonstrem o poder dos *design patterns* e o motivo que leva a incluí-los no arsenal dos alunos [WIC00].

Na sessão 4.4.1 a estrutura do padrão é mostrada. Nas seções seguintes, de 4.4.2 a 4.4.10 são descritos os padrões e na sessão 4.5 são relatadas atividades para a introdução de *Design Patterns*.

4.4.1 Estrutura dos Padrões

Os padrões de projeto são classificados de acordo com seus propósitos em três categorias: padrões de criação, padrões estruturais e padrões de comportamento [GRA02, ECK03].

As descrições dos padrões que seguem são organizadas em seções descritas abaixo. Uma vez que a natureza do padrão varia, nem todas as seções são utilizadas em todos os padrões.

Nome do Padrão: esta seção consiste no nome do padrão e a referência bibliográfica que indica a origem do padrão, onde foi escrito pela primeira vez em forma de padrão. Em muitos casos as idéias do padrão têm várias fontes e não uma referência bibliográfica. A maioria dos padrões não tem texto adicional nesta seção.

Sinopse: esta seção contém uma breve descrição do padrão. A sinopse comunica a essência da solução dada pelo padrão. A sinopse é principalmente dirigida a programadores experientes que podem reconhecer o padrão como algo que eles já sabiam, mas que ainda não tinha uma formalização conhecida. O reconhecimento do padrão através do nome e da sinopse, pode ser o suficiente. Porém, existem outras seções para descrever o padrão e possibilitar seu entendimento.

Contexto: esta seção descreve o problema no qual o padrão está inserido e para o qual a solução se destina. Em vários padrões, a seção contexto introduz o problema em termos de um exemplo concreto e sugere uma solução projetada.

Considerações: esta seção resume considerações que conduzem à solução geral do problema que é apresentada na seção solução.

Solução: esta seção é a essência do padrão. Esta descreve a solução geral do problema para o qual o padrão se destina.

Conseqüências: esta seção explica as implicações, boas e ruins, do uso da solução.

Nas sessões seguintes são mostrados alguns padrões de projeto que podem ser utilizados em cursos introdutórios. Estes foram extraídos dos trabalhos de Bergin [BER01c], Grand [GRA98, GRA02] e Eckel [ECK03].

4.4.2 Padrão Low Coupling/High Cohesion

Low Coupling e High Cohesion foram publicados originalmente de forma separada. Como estão intimamente ligados, foram apresentados de forma conjunta em [GRA99].

Sinopse

Um projeto com classes altamente acopladas ou com baixa coesão pode comprometer seriamente o projeto ou apresentar dificuldades de manutenção. Neste caso deve-se fazer análise das classes e redistribuir as atribuições. Com esta finalidade existe um conjunto de padrão conhecido como GRASP (General Responsibility Assignment Software Patterns) incluindo os padrões: Expert, Creator, Polymorphism, Pure Fabrication, Law of Demeter e Controller [GRA99].

Contexto

Em orientação a objetos uma diretiva é minimizar dependências e maximizar o reuso. O acoplamento é uma medida da conexão ou dependência entre classes. Outro problema é a maneira de gerenciar a complexidade. A coesão mede a responsabilidade da classe através dos seus métodos. Uma classe com baixa coesão faz muitas coisas não relacionadas e torna difícil o entendimento da classe, além disso dificulta a reutilização e a manutenção, pois pode ser constantemente afetada por mudanças. Uma classe com baixa coesão assume responsabilidades que pertencem a outras classes.

Considerações

☺ O baixo acoplamento facilita a manutenção, o entendimento e a reutilização das classes.

☺ A alta coesão facilita o entendimento da classe.

Solução

Se você acha que está trabalhando com um projeto muito inflexível ou difícil de manter, procure por classes que estão altamente acopladas ou com perda de coesão como uma possível causa do problema. Um problema comum em projeto é que muitas responsabilidades são atribuídas a uma classe, o que a torna difícil de implementar e manter. Estas classes são facilmente reconhecidas porque elas estão fortemente acopladas a outras ou tem um conjunto de métodos não pertinentes. Classes que são altamente acopladas normalmente têm baixa coesão e vice-versa.

Conseqüências

- A aplicação deste padrão resulta em classes com alta coesão e baixo acoplamento que são fáceis de manter e reutilizar.
- Alto acoplamento e perda de coesão são comuns, mas não são as únicas causas de projetos inflexíveis e difíceis de manter.

4.4.3 Padrão Composed Method

Sinopse

Reorganize métodos que são muito grandes em métodos menores mais fáceis de entender [GRA99].

Considerações

☺ Métodos bem pequenos ficam mais fáceis de entender porque a mente humana pode guardar somente um pequeno número de coisas ao mesmo tempo.

☺ Um método grande pode se quebrado em vários menores.

☺ Classes com métodos menores são mais fáceis de se corrigirem os erros.

☹ A lógica da implementação da classe em métodos menores pode ser comprometida pois é necessário passar de um método a outro.

☺ É importante quebrar os métodos maiores sem prejudicar a legibilidade do código.

Solução

Caso um método seja muito grande quebre-o em métodos menores com nomes significativos e com código de fácil entendimento. Os submétodos podem ser privados, pois não existe razão para expô-los a outras classes.

Se a implementação de um método inclue conceitos ou ações distintos organize-os em submétodos correspondentes por ações ou conceitos.

Conseqüência

☺ Métodos menores são mais fáceis de entender por pessoas.

☺ Métodos menores são menos vulneráveis a erros.

☺ Classes compostas de métodos menores apresentam um baixo custo de manutenção.

- O projeto orientado a objetos que precede a escrita da classe assegura que um programa está bem estruturado ao nível de classe. Porém, não ajuda muito com o projeto da organização interna da implementação da classe. Ao fazer o particionamento dos métodos

grandes em menores pode acontecer a descoberta de abstrações que podem organizar melhor a implementação. Pode-se concluir que algum método privado pode ser reutilizado dentro da classe.

- Algumas vezes, após aplicar o padrão Composed Method, você descobre que a classe possui um grande número de métodos pequenos. Isto é normalmente um indicador que a classe poderia ser dividida em várias classes, com a classe original agindo como uma fachada para as classes adicionais.

4.4.4 O Padrão Polymorphism

Sinopse

Quando comportamentos alternativos são selecionados baseados no tipo do objeto, use um método polimórficos para selecionar o comportamento, ao invés de usar comandos if para testar o tipo [GRA99].

Considerações

- É sempre possível selecionar um comportamento usando uma cadeia explícita de comandos if.
- Quando escrevemos uma cadeia de comandos if, programadores têm a oportunidade de introduzir erros no programa. Posteriormente, durante a manutenção, programadores têm outras oportunidades de introduzir novos erros. Um erro particular de manutenção é adicionar um novo comportamento a algum comando if, mas se esquecer de adicionar a outros.
- A seleção de comportamento por um método polimórfico não exige codificação de qualquer lógica explícita. Isto permite diminuir as oportunidades de introduzir erros.
- Uma cadeia de comandos if torna mais difícil o teste de unidade uma vez que aumenta o número de caminhos de execução a serem testados. O número de caminhos de execução é multiplicado pelo tamanho da cadeia de comandos if.
- Uma chamada de método polimórfico pode ser mais rápida que uma cadeia de comandos if. Simplesmente tendo uma referência a um objeto apropriado que provê acesso pela implementação de métodos acesso, sem a necessidade de computação adicional. O custo total para a chamada do método polimórfico é somente a chamada do método.

- Você deseja a possibilidade de incluir o mecanismo para selecionar o comportamento em um projeto orientado a objetos. Cadeias de comandos if não possuem uma representação natural em projeto orientado a objetos.
- Você pode selecionar um comportamento adequado a partir de um conjunto de classes alternativas de um objeto.
- Se um problema é colocado em termos de selecionar comportamento baseado em um conjunto de valores de dados, você pode utilizar os benefícios das chamadas de métodos polimórficos usando uma classe diferente para solucionar cada valor de dado.

Solução

Se um problema é formulado de forma que envolve a seleção de comportamento baseado nos valores dos dados a partir de um conjunto de dados, organize a solução para representar cada valor de dado com uma classe diferente. Isto permite a solução selecionar o comportamento baseado em classes de objetos.

Se você pode usar a classe de um objeto para selecionar um comportamento, então coloque o comportamento desejado na classe e use chamadas de métodos polimórficas para solicitar o comportamento.

Conseqüências

☺ É mais fácil e mais legível implementar a seleção de comportamento usando o polimorfismo que usando uma seleção lógica explícita.

☺ Usando polimorfismo, é mais fácil adicionar comportamentos, porque você não tem que procurar as cadeias apropriadas de comandos if para adicionar o comportamento.

☹ Usando o padrão Polymorphism existe uma adição de classes ao projeto. Isto pode tornar o projeto mais difícil de entender como um todo.

☹ Usando uma cadeia de comandos if permite um programador que está lendo o código ver as escolhas possíveis. O uso de método polimórficos torna esta informação mais difícil de obter.

4.4.5 Padrão Delegation

Sinopse

Em algumas situações, usar a herança para estender uma classe não é a melhor opção. Embora menos conveniente, a delegação é uma forma mais geral de estender classes. Delegação é adequada em situações onde herança não funciona bem [GRA02].

Contexto

Por exemplo, herança é útil para capturar relacionamento “é-um-tipo-de” devido a sua natureza estática. Entretanto, o relacionamento do tipo “é-um-papel-de” é complicado para modelar com herança. Em certas situações, instâncias de uma classe podem assumir múltiplos papéis. Considere o exemplo de um sistema de reserva de passagens aéreas que incluem papéis como passageiro, agente de vendas de passagens e tripulação. Poderíamos representar estes relacionamentos como uma classe Pessoa com subclasses correspondentes para os papéis especificados, como pode ser visto no diagrama da Figura 4-1.

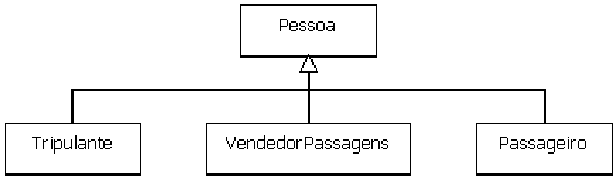

Figura 4-1 Diagrama de classes dos relacionamentos da hierarquia Pessoa.

Porém, um funcionário da tripulação pode ser um passageiro. Outras vezes um funcionário da tripulação pode vender passagens. Para modelar esta situação, você precisaria de sete subclasses para Pessoa como pode ser observado na Figura 4-2.

O número de subclasses necessário cresce exponencialmente com o número de papéis. Para modelar todas as combinações dos seis papéis seriam necessárias de 63 subclasses.

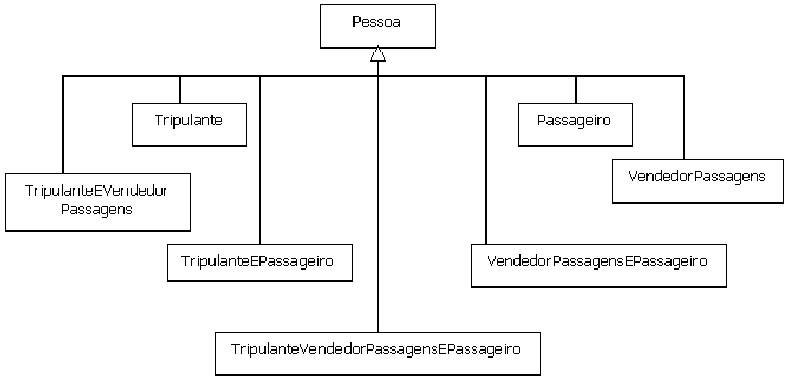

Figura 4-2 Pessoas e suas subclasses.

O problema mais sério é que a mesma pessoa pode estar em combinações diferentes de papéis em tempos diferentes. Como os relacionamentos de herança são estáticos e não mudam durante o tempo, para modelar diferentes combinações de papéis no decorrer do tempo usando relacionamentos de herança, é preciso usar objetos diferentes durante o tempo para

representar a mesma pessoa a fim de capturar as mudanças de papéis. Modelar dinamicamente mudança de papéis com herança é complicado.

Por outro lado, é possível representar pessoas em diferentes papéis usando delegação sem ter estes problemas. A Figura 4-3 mostra como o modelo poderia ser reorganizado usando delegação.

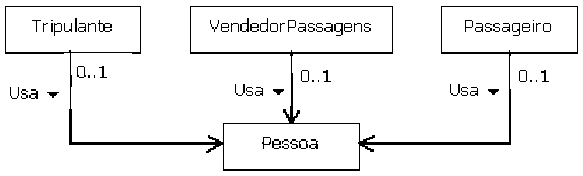

Figura 4-3 Modelo com delegação

Usando a organização da Figura 4-3, um objeto Pessoa delega a responsabilidade de preencher um papel particular a um objeto que seja específico daquele papel. Você precisa de tantos objetos quanto são os papéis a serem preenchidos. Combinações diferentes não precisam de objetos adicionais. Isso acontece porque delegação de objetos pode ser dinâmica, papéis de objetos podem ser adicionados ou removidos quando uma pessoa preenche diferentes papéis.

No caso de um software para companhias aéreas, um pré-determinado conjunto de objetos com papéis específicos podem associar-se a diferentes Pessoas durante o tempo. Por exemplo, quando um vôo está escalado, será determinado que um certo número de tripulantes esteja a bordo. Quando um vôo está completo, pessoas específicas serão associadas com os papéis de tripulantes. Quando se troca a escala, uma pessoa pode passar do papel de tripulante para outro papel.

Considerações

☺ Herança é um relacionamento estático; não muda com o passar do tempo. Se for preciso que um objeto seja de uma subclasse diferente de uma classe, em tempos diferentes, esta subclasse poderia não ter sido criada em um primeiro momento. Se um objeto é criado como uma instância de uma classe, este será sempre uma instância daquela classe. Entretanto, um objeto pode delegar comportamento a diferentes objetos em tempos diferentes.

☺ Se for detectado que uma classe tenta esconder um método ou variável herdada de uma superclasse de outras classes, aquela classe não deveria herdar da superclasse. Não existe uma maneira efetiva de esconder métodos ou variáveis herdados de uma superclasse. Entretanto, é possível para um objeto usar métodos de outro objeto e variáveis enquanto

assegura que este seja o único com acesso a outros objetos. Isto é feito da mesma forma que herança, mas usa relacionamentos dinâmicos que podem trocar com o tempo.

Classes clientes que usam a classe específica do domínio do problema podem ser escritas de forma que assuma que a classe do domínio específico do problema seja uma subclasse da classe utilitária. Se ocorrer alteração na classe do domínio do problema em uma superclasse diferente, classes clientes que se apóiam na classe do domínio do problema terão sua superclasse original quebrada.

☹ A delegação pode ser menos conveniente que herança, porque requer mais código para implementação.

☹ Delegação impõe menos estrutura nas classes que herança. Em projetos que restrições na estrutura de classes é importante, a estrutura e flexibilidade de herança podem ser uma virtude. Isto é normalmente verdadeiro em frameworks.

Alguns usos inadequados de herança podem ser considerados *AntiPatterns* [GRA02]. Em particular, subclassificação de classes utilitárias e uso de herança para modelar papéis são falhas comuns em projeto.

☺ A maioria das reutilizações e extensões de uma classe não são feitos de forma adequada pela herança.

☺ O comportamento que uma classe herda de sua superclasse não pode ser facilmente modificada no tempo. Herança não é útil quando o comportamento da classe não pode ser determinado até o tempo de execução.

Solução

Use delegação para reusar e estender o comportamento da classe. Você faz isto escrevendo uma nova classe (Delegator) que incorpora a funcionalidade da classe original usando uma instância da classe original (Delegate) e chamando seus métodos.

A Figura 4-4 mostra que uma classe no papel de Delegator usa uma classe no papel de Delegada. Delegação é de propósito mais geral que herança. Qualquer extensão a uma classe que pode ser feita por herança pode ser feita por delegação.

Figura 4-4 Diagrama do padrão Delegator

Conseqüências

Delegação pode ser usada sem o problema que acompanha herança. Delegação permite que o comportamento seja facilmente composto em tempo de execução. A principal desvantagem é menos estrutura que com o uso da herança. Relações entre classes construídas por uso de delegação são menos óbvias que as construídas com herança.

4.4.6 O Padrão Interface

Sinopse

É preciso projetar e implementar um conceito com várias implementações diferentes. Se classes clientes forem independentes de dados e serviços específicos providos por outra classe, fica fácil substituí-las por outras com o mínimo impacto em classes clientes. Você pode fazer isto tendo outras classes que acessem os dados e serviços através de uma interface [GRA02].

Contexto

Você deseja:

- Dar aos clientes a liberdade de escolha de selecionar uma implementação específica.
- Alterar implementação sem afetar classes clientes.
- Introduzir novas implementações sem comunicar aos clientes.
- Separar implementação dos clientes das classes.
- Fazer tão simples quanto possível, mas não de forma ingênua.

Suponha que você precise escrever uma aplicação para gerenciar a compra de bens para um negócio. Entre as entidades de seu programa você precisa ter vendedores, companhias de frete, galpões de armazenamento e faturamento. Um aspecto comum a estas entidades é que elas têm um endereço. Este endereço aparece em diferentes partes da interface com o usuário. Você precisa ter uma classe para mostrar e editar endereços de modo que possa reutilizá-la em qualquer lugar que houver um endereço na interface do usuário. Chamaremos esta classe de PainelDeEndereço.

Os objetos do tipo PainelDeEndereço alteram e acessam informações relativas a endereços em diferentes objetos. Isto trás a questão do que a classe PainelDeEndereço assume sobre a classe do objeto de dados com o qual ele trabalha. Claramente, serão utilizadas diferentes classes para representar vendedores, companhias de fretes e outras.

O problema pode ser resolvido pela criação de uma interface de endereço. Instâncias da classe PainelDeEndereço poderiam simplesmente requerer aos objetos que trabalhem com

a implementação da interface endereço. Eles poderiam chamar os métodos de acesso da interface para obter e colocar informações de endereço no objeto.

Usando a indireção que a interface provê, clientes da interface PainelDeEndereço são capazes de chamar métodos dos objetos de dados sem ter que saber a que classe este pertence. A Figura 4-5 mostra estes relacionamentos.

Solução

Determine a funcionalidade do conceito separadamente de sua implementação. Represente a funcionalidade como uma interface. Crie classes para implementar a interface. Evite assim as dependências entre classes devido ao relacionamento usa/usado-por. A indireção através de uma interface é mostrada na Figura 4-5.

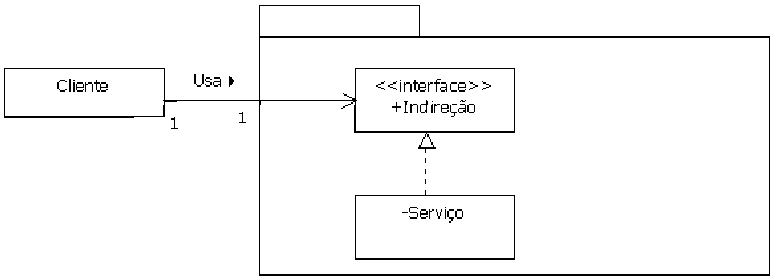

Figura 4-5 Diagrama com Interface

Conseqüências

☹ Uma conseqüência é que este padrão tornar uma classe dependente de um serviço de outra classe ligada a uma classe específica.

4.4.7 O Padrão Abstract Superclass

Sinopse

Assegure comportamento consistente de classes conceitualmente relacionadas dando a elas uma superclasse abstrata comum [GRA02].

Contexto

Você deseja evitar código redundante, adicionar facilmente outras implementações e fazer isto tão simples quanto possível, mas não o mais simples. A superclasse abstrata implementa os métodos comuns e as subclasses os métodos que diferenciam.

Considerações

☺ Você quer assegurar que a lógica comum das classes relacionadas é implementada consistentemente por cada classe.

☺ Você quer evitar um erro de execução e custo adicional de manutenção para código redundante.

☺ Você deseja tornar fácil escrever classes relacionadas.

☺ Você quer organizar comportamento comum, embora em muitas situações, herança não seja a forma apropriada de fazer isto. O padrão Delegate descreve em detalhes.

Solução

- Separe a funcionalidade variante das implementações das funcionalidades invariantes.
- Implemente a funcionalidade invariante como funcionalidade compartilhada em uma superclasse abstrata.
- Declare a funcionalidade variante em superclasses abstratas usando métodos abstratos.
- Faça subclasses de implementação da superclasse abstrata.
- Faça com que as subclasses de implementação implementem as funcionalidades variantes.

Para possibilitar a extensão, organize comportamentos variantes em métodos com a mesma assinatura. Declare a superclasse abstrata ter estes métodos com a assinatura comum. A Figura 4-6 mostra esta organização.

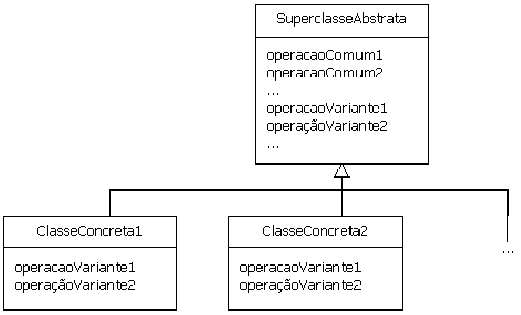

Figura 4-6 Diagrama de classes do padrão Abstract Superclass

Conseqüências

- Poucos casos de testes serão necessários para testar completamente sua classe, uma vez que existe pouco código a ser testado.
- Usar o padrão Abstract SuperClass cria dependências entre a superclasse e suas subclasses. Alterações na superclasse podem ter efeito não intencional em algumas subclasses, desta forma o programa fica mais difícil de manter.

4.4.8 O Padrão Strategy

Este padrão foi inicialmente descrito em [GOF95].

Sinopse

O padrão Strategy pode ser aplicado quando existem problemas a serem resolvidos e existe uma parte comum entre eles. É preciso separar o que é variável nos problemas do restante e representar a parte variável como uma **estratégia**.

A interface comum encapsula algoritmos relacionados. Isto permite a seleção do algoritmo que varia por tipo de objeto. Permite também a seleção de algoritmo que varia no decorrer do tempo.

Contexto

Para exemplificar considere que exista um problema prático onde seja necessário ler um arquivo de palavras (separadas por espaço em branco) e imprimir as que começam com a letra "t". E outro problema é preciso ler palavras de um arquivo e imprimir as que possuam mais de 5 caracteres. Em outro é preciso ler palavras de um arquivo e imprimir os palíndromos.

Todos estes problemas têm a mesma estrutura, o que poderia ser resolvido desenvolvendo-se um, copiando e fazendo as modificações para os outros. Porém, é possível resolver os problemas sem a necessidade de fazer alterações, tornando-o flexível o suficiente para resolver os três problemas.

Esta estratégia é implementada como uma interface. Esta contém métodos que serão implementados pelas classes que implementam a interface. No exemplo dado a EstrategiaChecar possui somente um método como **checar** que fará a checagem de acordo com a necessidade.

Como existem várias classes que implementam a mesma interface, pode-se desenvolver a parte do programa que processe o texto e a análise das palavras pode ser feita criando-se o objeto da classe específica que implementa a checagem correta.

Existem muitos tipos de estratégias. É possível aplicar diferentes tipos de estratégias para conduzir a ações diferentes em objetos dependendo da implementação da estratégia. Estratégias podem dar flexibilidade a programas que podem passar a resolver uma grande classe de problemas ao invés de apenas um.

A chave para Strategy é separar o que é variável no conjunto do problema e construir uma interface para ela. Então escrever a solução do conjunto do problema em termos da

interface. A interface é usada como tipo de parâmetro de um ou mais métodos na classe que resolve o problema. A estratégia deve consistir em um ou mais métodos. Estes métodos são chamados para resolver os problemas específicos. Quando uma solução geral deve ser reusada em uma nova situação, o programador escreve apenas uma nova estratégia para a nova situação.

Considerações

☺ Um programa deve prover múltiplas variações de um algoritmo ou comportamento.

☺ Você precisa variar o comportamento de cada instância da classe.

☺ Você precisa variar o comportamento dos objetos em tempo de execução.

☺ Delegar comportamento a uma interface permite classes que usam o comportamento não estar ciente das classes que implementam a interface e o comportamento.

☹ Se o comportamento da instância de uma classe não varia de instância para instância no decorrer do tempo, então é mais simples para a classe conter diretamente o comportamento ou conter diretamente uma referência estática para o comportamento.

Soluções

A Figura 4-7 mostra a representação em UML deste padrão mostrando os papéis que as classes desempenham no padrão Strategy.

O padrão Strategy sempre ocorre com um mecanismo para determinar o objeto EstrategiaConcreta que o objeto cliente usará. A seleção de uma estratégia é freqüentemente direcionada pela informação de configuração ou eventos. Entretanto, o mecanismo atual varia muito. Por esta razão, nenhum mecanismo de seleção de estratégia é incluído no padrão.

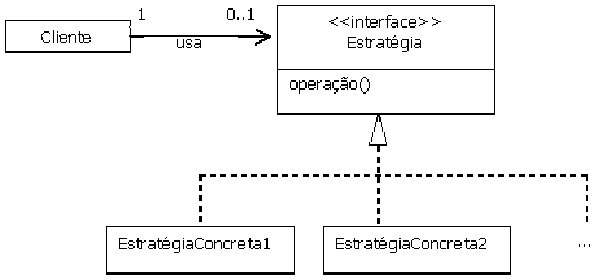

Figura 4-7 Diagrama de classes do padrão Strategy

Conseqüências

☺ O padrão Strategy permite que o comportamento de objetos Cliente seja determinado dinamicamente com base nos objetos.

☺ O padrão Strategy simplifica classes Cliente por aliviá-las de qualquer responsabilidade de selecionar comportamento ou implementação de comportamentos alternativos. Isto simplifica o código para objetos Cliente eliminando comandos if e switch. Em alguns casos, isto pode aumentar a velocidade dos objetos Cliente uma vez que eles não gastam tempo algum selecionando comportamento.

4.4.9 Padrão Decorator

Sinopse

O padrão *Decorator* lida com o problema de adicionar funcionalidade a um objeto existente. Pressupomos que queremos um objeto que responda às mesmas chamadas de métodos (com a mesma interface), mas que tem um comportamento adicional ou alterado, de forma a ficar transparente para o seu cliente [GRA02].

Contexto

Por exemplo, suponha que seja necessário resolver o problema acima, do padrão Strategy, mas com a necessidade de contar quantas palavras foram produzidas, ou seja, quantas vezes a estratégia retornou verdadeiro. É possível modificar o código da classe criada, mas não é a melhor opção. Melhor solução é dada pela criação de uma nova classe que tenha uma estratégia como campo e o campo contador. No construtor da classe tem-se como parâmetro a estratégia selecionada e na implementação do método checar retorna-se o resultado da checagem, incrementa o contador, caso a checagem retorne verdadeiro e retorna-se o resultado da checagem.

Considerações

☺ Existe uma necessidade de estender a funcionalidade de uma classe, mas existem razões para não estender através da herança.

☺ Existe a necessidade de estender dinamicamente a funcionalidade de um objeto e possivelmente também retirar a funcionalidade estendida.

Solução

Uma forma de fazer isto é por meio de herança. Uma subclasse pode sobrescrever a implementação de métodos e adicionar novos métodos. Mas utilizar a herança é uma solução estática: depois de criados, os objetos não podem alterar seu comportamento.

Decorators são utilizados para adicionar funcionalidade a um tipo de objeto. A chave para um decorator é que o decorator "envolve" o objeto decorado e o vê exatamente assim. Isto significa que o decorator implementa a mesma interface que o objeto que ele decora.

Pense em um objeto decorator como um "objeto embrulhado". Qualquer mensagem que um cliente envie ao objeto é recebida pelo decorator. O decorator pode aplicar algumas ações e então passar a mensagem recebida para o objeto decorado. O objeto provavelmente retorna um valor (para o decorator) que pode novamente aplicar uma ação para aquele resultado e finalmente enviar o resultado para o cliente original. Para o cliente o decorator é invisível. Ele simplesmente envia uma mensagem e recebe um resultado. Entretanto o decorator tem duas chances de retornar o resultado.

A chave de sucesso para o Decorator é ter um conjunto de objetos definidos por uma interface. O decorator implementa aquela interface e toma um objeto daquele tipo de interface como um parâmetro de seu construtor, que é armazenado internamente em um campo. Então, quando uma mensagem é enviada para o decorator este passa a mesma mensagem, ou uma relacionada, para o objeto decorado e pega o resultado. Este processo pode modificar ou alterar um objeto como necessário e retorná-lo ao remetente da mensagem original.

A representação em UML deste padrão é mostrada na Figura 4-8.

Conseqüências

☺ O padrão Decorator provê mais flexibilidade que herança. Permite alterar dinamicamente o comportamento de objetos individuais adicionando e removendo embrulhos. Herança, por outro lado, determina a natureza de todas as instâncias da classe estaticamente.

☺ Usando diferentes combinações de diferentes tipos de objetos empacotados, é possível criar muitas combinações diferentes de comportamento. Para criar muitos tipos diferentes de comportamento com herança é necessário definir muitas classes diferentes. Usar o padrão Decorator normalmente resulta em menos classes que com herança. Uma quantidade menor de classes simplifica o projeto e a implementação de programas. Por outro lado, usando o padrão Decorator normalmente tem-se uma quantidade maior de objetos. O grande número de objetos pode dificultar a depuração, principalmente porque os objetos são muito parecidos.

☹ A flexibilidade de objetos empacotados torna-os mais propensos a erros que com herança. Por exemplo, é possível combinar objetos empacotados de forma que não funciona ou criar referências circulares entre objetos empacotados.

☹ O padrão Decorator torna difícil usar a identificação do objeto para identificar serviços dos objetos, uma vez que os serviços dos objetos se ocultam nos objetos empacotados.

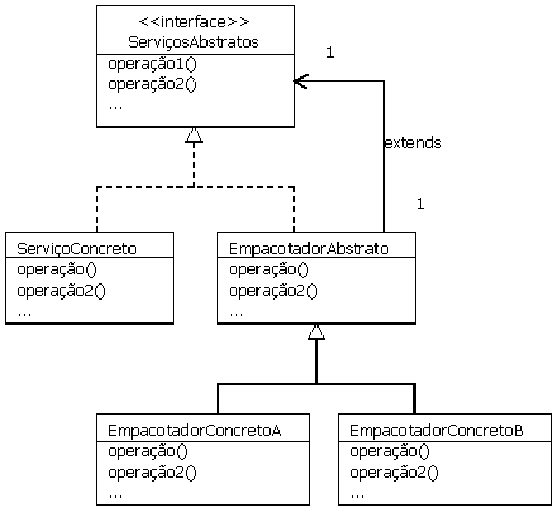

Figura 4-8 Diagrama do padrão Decorator

4.4.10O Padrão Composite

Sinopse

O padrão Composite permite construir objetos complexos compondo recursivamente objetos similares, que tenham uma interface ou superclasse comum [GRA02].

Contexto

No exemplo do padrão das palavras, o padrão Strategy permite fatorar as partes que mudam em um problema e o Decorator permite adicionar funcionalidade a um objeto como uma estratégia. Suponha, entretanto, que se tenha uma quantidade de estratégias de checagem e seja necessário encontrar palavras em algum arquivo texto que satisfaçam todos os critérios. Em outras palavras, aplicar todas as estratégias desenvolvidas e pegar a interseção dos resultados. Pode-se escrever uma nova estratégia com um teste complexo, ou compor uma estratégia usando o padrão Composite.

Considerações

☺ Você tem um objeto complexo que você quer decompor em uma hierarquia todo-parte de objetos.

☺ Você deseja minimizar a complexidade da hierarquia todo-parte minimizando o número de tipos diferentes de objetos filhos que objetos na árvore precisam estar cientes de.

☹ Não há exigência para distinguir entre a maioria dos relacionamentos todo-parte.

Solução

A chave do sucesso para o Composite é ter um container que implementa uma interface e este conter um objeto que também implementa a mesma interface. Quando o composite recebe uma mensagem definida na interface este a repassa para aqueles objetos que ele contêm. Normalmente é preciso adicionar ou remover elementos.

Um Composite é um objeto que faz duas coisas. Primeiro, ele implementa alguma interface, e segundo ele é um contêiner que contém coisas que implementam a mesma interface. Então, quando o objeto composite recebe alguma mensagem, ele a repassa para o objeto que ele contém.

Os relacionamentos gerais para tal organização de classes são mostrados na Figura 4-9.

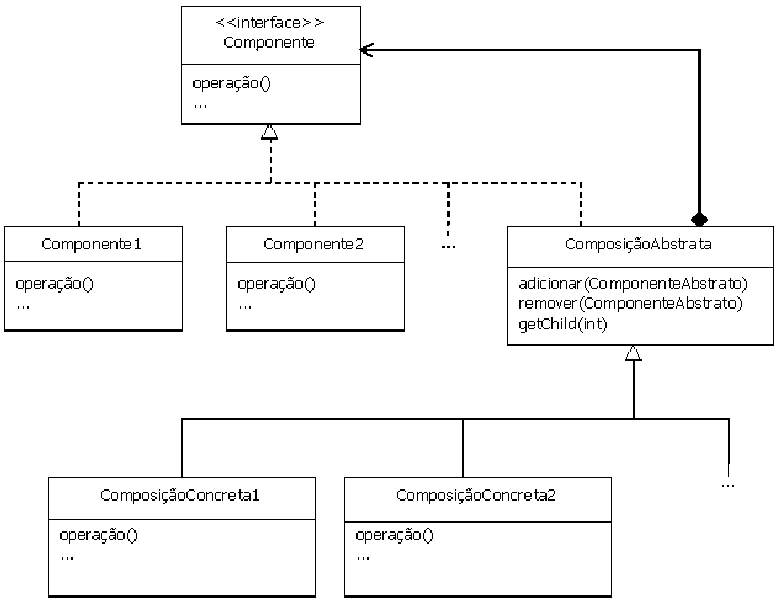

Figura 4-9 Diagrama de classes do padrão Composite

Conseqüências

☺ Você pode acessar uma árvore estruturada de objeto composto e os objetos que a constituem através da interface Componente, se eles forem objetos simples ou compostos. A estrutura da composição não força outros objetos a fazerem esta distinção.

☺ Objetos clientes de um ComponenteAbstrato podem simplesmente tratá-lo como um ComponenteAbstrato, sem ter que estar ciente de qualquer subclasse de ComponenteAbstrato.

☺ Se um cliente solicita um método de um objeto Componente que se supõe realizar alguma operação e o objeto Componente é um objeto ComposicaoAbstrata, então este pode

delegar a operação aos objetos Componente que o constituem. Similarmente, se um objeto cliente chama um método do objeto Componente que não é uma ComposicaoAbstrata e o método requer alguma informação contextual, então o objeto Componente delega o pedido de informação contextual a seu pai.

- Alguns componentes podem implementar operações que únicas do componente. Um princípio de projeto orientado a objetos é que métodos especializados devem aparecer somente em classes que necessitam deles. Normalmente, uma classe pode ter métodos que provêm funcionalidades relacionadas e forma um conjunto coeso. Colocar um método especializado em uma classe de uso geral ao invés de em uma classe especializada que precisa do método é contrário ao princípio de alta coesão. Isto é contrário ao princípio pois adiciona um método não relacionado a outros métodos da classe de propósito geral. O método não relacionado é herdado por subclasses da classe de propósito geral que não estão relacionadas ao método.

  Quando se aplica o padrão Composite sacrifica-se a alta coesão pela simplicidade. Esta exceção para uma regra amplamente aceita é baseada mais na experiência que na teoria.

☹ O padrão Composite permite qualquer objeto Componente ser um filho de uma ComposicaoAbstrata. Se você precisa reforçar um relacionamento mais restrito, então você deverá adicionar código ciente de tipo em ComposicaoAbstrata ou em suas subclasses. O que reduz alguns valores do padrão Composite.

4.5 ATIVIDADES PARA INTRODUÇÃO AOS PADRÕES DE PROJETO

Nesta seção serão apresentadas sugestões encontradas na literatura para a introdução de orientação por objetos e padrões de projeto.

4.5.1 Padrões em classes Java

Várias classes da linguagem Java são exemplos naturais de *design patterns*. Na Tabela 4-2 estão relacionadas algumas.

Tabela 4-2 Padrões naturais de classes em Java

Classe	Padrão	Descrição
FileReader	Decorator	Readers podem ser usados para ler do teclado ou de arquivos em disco ou outro meio de armazenamento. Para ler de um arquivo você usa um **FileReader**. Readers em geral sabem como traduzir de uma codificação externa

		para uma codificação interna (UNICODE). FileReaders simplesmente adicionam a habilidade de conectar um reader a um nome de arquivo (“decoram”).
BufferedReader	Decorator	Não é eficiente ler um caractere por vez pois o mecanismo de leitura de disco é muito lento, comparado com a velocidade da CPU. Assim, é vantagem ler vários caracteres de uma vez usando um BufferedReader. Isto significa menos acessos de leitura e aumento de velocidade do programa. Um Buffered é por si mesmo um Reader, mas com a habilidade de buferizar sua entrada buferização.
Layout	Strategy	Quando se adiciona um objeto Layout a um objeto Panel, você dá ao objeto Panel uma estratégia para organizar os elementos contidos no Panel.
Panel	Composite	Um Panel é um Component, mas também contém Components. Quando se solicita a um Panel que “paint” este “paints” os componentes que ele contém. Pode-se adicionar ou remover componentes a qualquer tempo.
Component	Composite	O pacote java.awt contêm um exemplo do padrão Composite. Sua classe preenche o papel de Component. Sua classe Container preenche o papel de ComposicaoAbstrata. Esta tem um número de classes no papel de ComponenteConcreto, incluindo Label, TextField e Button. As classes no papel de ComposicaoConcreta incluem Panel, Frame e Dialog.
Classes de eventos em objetos	Delegation	Este *design pattern* é a base para Java delegar modelos de eventos onde o objeto fonte do evento envia eventos para objetos ouvintes. Eventos em objetos de origem geralmente não decidem o que fazer com um evento; de fato, eles delegam a responsabilidade do processamento do evento aos objetos ouvintes.
java.io.FilenameFilter	Interface	A interface FilenameFilter declara um método chamado accept. O método accept tem um argumento que é o nomedoarquivo. Supõe-se que o método retorne true ou false para indicar se o nome do arquivo pode ser incluído em uma coleção. A API Java também provê a classe java.awt.FileDialog que pode usar um objeto do tipo FilenameFilter para filtrar os arquivos que este exibe.
AWTEvent	Class Abstract	A classe é uma classe abstrata para classes que encapsulam eventos relacionados à interface gráfica do usuário (GUI). Esta define um pequeno número de métodos que são comuns às classes de eventos da interface do usuário.

4.5.2 Padrões Decorator, Composite e Strategy

Para introduzir os padrões Decorator, Composite e Strategy um problema prático é proposto [BER01c]. O problema inicial introduz o padrão Strategy. Gradualmente torna-se mais complexo agregando os padrões Decorator e Composite.

Esta seqüência de problemas propostos utiliza os padrões pedagógicos. **Espiral**, ao retomar o problema e aumentar sua complexidade. A solução do problema também representa uma **Metáfora Consistente** que pode auxiliar na resolução de outros problemas.

Padrão Strategy

Problema proposto

> "*É preciso ler um arquivo de palavras (separadas por espaço em branco) e imprimir as que começam com a letra "t". Em outro problema é preciso ler palavras de um arquivo e imprimir as que possuam mais de 5 caracteres. Em outro é preciso ler palavras de um arquivo e imprimir as palíndromos*."

Todos estes problemas têm a mesma estrutura, o que poderia ser resolvido desenvolvendo-se um, copiando e fazendo as modificações para os outros. Porém, é possível resolver os problemas sem a necessidade de fazer alterações, tornando-o flexível o suficiente para resolver os três problemas.

Para fazer isto é preciso separar o que é variável nos problemas do restante e representar a parte variável como uma estratégia. A parte variável aqui envolve examinar uma palavra do arquivo e determinar se tem certas características. Abaixo é apresentada a implementação destas estratégias como uma interface:

```
public interface EstrategiaChecar {
          public boolean checar(String s);
}
```

Abaixo a implementação das diferentes estratégias de checagem para o exemplo dado:

```
public class IniciaComT implements EstrategiaChecar {
  public boolean checar(String s) {
       if (s == null || s.length() == 0) return false;
           return s.charAt(0) = 't';
        }
}
```

```
public class MaiorQue5 implements EstrategiaChecar {
   public boolean checar(String s) {
        if (s == null) return false;
            return s.length() > 5;
        }
}
```

```
public class Palindromo implements EstrategiaChecar {
```

```
   public boolean check(String s) {
        if (s == null) return false;
        int tamanho = s.length();
        if(tamanho < 2) return true;
        int metade = length/2;
        for(int i = 0; i < metade; ++i)
            if(s.charAt(i) != s.charAt(tamanho - 1 - i))
                return false;
        return true;
    }
}
```

No código abaixo está a implementação da leitura do texto a partir de um arquivo texto identificado pelo parâmetro **nomeArq** e o processamento de suas palavras por linhas armazenadas em um objeto buffer. O parâmetro **qual,** do tipo EstrategiaChecar, é quem decide qual estratégia usar.

```
public void imprimaQuando(String nomeArq, EstrategiaChecar qual)
   throws IOException {
   BufferedReader emArquivo = new BufferedReader(new FileReader(nomeArq));
   String buffer = null;
   while((buffer = emArquivo.readLine()) != null) {
     StringTokenizer palavras = new StringTokenizer(buffer);
     while (palavras.hasMoreTokens() ) {
        String palavra = palavras.nextToken();
        if (qual.checar(palavra))
           System.out.println(palavra);
     }
   }
}
```

A linha de comando abaixo imprime palíndromes do texto:

```
imprimaQuando("texto1.txt", new Palindrome());
```

É possível tornar a estratégia mais genérica através da utilização de parâmetros. Por exemplo, se a checagem foi feita para palavras com mais de 5 caracteres, também poderia ser feita para palavras com mais de 3 caracteres. Abaixo está o código para uma estratégia genérica.

```
public class MaiorQueN implements EstrategiaChecar {
    private int n;

    public MaiorQueN (int tamanho) {
                n = tamanho;
    }

    public boolean checar(String s){
        if(s == null) return false;
        return s.length() > n;
    }
}
```

Com esta estratégia é possível encontrar palavras com mais de cinco caracteres:

```
imprimaQuando("texto1.txt", new MaiorQueN(5));
```

Padrão Decorator

Decorators são utilizados para adicionar funcionalidade a um tipo de objeto. A chave para um decorator é que o decorator "envolve" o objeto decorado e o vê exatamente assim. Isto significa que o decorator implementa a mesma interface que o objeto que ele decora.
O objeto decorator é como um "objeto embrulhado". Qualquer mensagem que um cliente envie ao objeto é recebida pelo decorator. O decorator pode aplicar algumas ações e então passar a mensagem recebida para o objeto decorado. O objeto provavelmente retorna um valor (para o decorator) que pode novamente aplicar uma ação para aquele resultado e finalmente enviar o resultado para o cliente original. Para o cliente o decorator é invisível. Ele simplesmente envia uma mensagem e recebe um resultado. Entretanto o decorator tem duas chances de retornar o resultado.

Suponha que seja necessário resolver o problema anterior, do padrão Strategy, mas com a necessidade de contar quantas palavras foram produzidas, ou seja, quantas vezes a estratégia utilizada retornou verdadeiro. É possível modificar o código acima para isto, mas não é a melhor opção. A melhor solução é dada por um decorator que tenha uma estratégia como parâmetro em seu construtor e implemente a interface EstrategiaChecar. Observe o código abaixo.

```
class DecoradoComContador implements EstrategiaChecar {
   private int contador = 0;
   private EstrategiaChecar checar;

   public DecoradoComContador(EstrategiaChecar checagem) {
        checar = checagem;
   }

   public boolean checa(String s) {
        boolean resultado =  checar.checa(s);
        if(resultado)
           contador ++;
        return resultado;
   }

   public int contador() {
        return contador;
   }

   public void reset() {
        contador = 0;
   }
}
```

O código abaixo encontra palíndromos, imprime-os e conta quantos foram encontrados:

```
DecoradoComContador cont = new DecoradoComContador (new Palindrome());
imprimaQuando("texto1.txt",cont);
System.out.println("" + cont.contador());
```

Note que o método de checagem do decorador chama o método de checagem do método do palíndromo que foi utilizada como a estratégia. Este então conta a palavra quando se tem resultado verdadeiro. Nem a classe Palíndromo nem o método imprimaQuando foi modificado para alcançar o esperado. Este é o poder do padrão Decorator.

Note que os decorators podem prover métodos adicionais além do que é requerido pela interface que eles decoram. No exemplo atual, tem-se a contagem e a adaptação dos métodos do DecoradoComContador.

A chave de sucesso para o Decorator é ter um conjunto de objetos definidos por uma interface. O decorator implementa aquela interface e toma um objeto daquele tipo de interface como um parâmetro de seu construtor, que é armazenado internamente em um campo. Então, quando uma mensagem é enviada para o decorator este passa a mesma mensagem, ou uma relacionada, para o objeto decorado e pega o resultado. Este processo pode modificar ou alterar um objeto como necessário e retorná-lo ao remetente da mensagem original.

Padrão Composite

O padrão Strategy nos permite fatorar partes que mudam em um problema e o Decorator nos deixa adicionar funcionalidade a um objeto como uma estratégia. Suponha, entretanto, que exista uma grande quantidade de estratégias de checagem e seja necessário encontrar palavras em algum arquivo texto que satisfaçam todas. Em outras palavras, aplicar todas as estratégias desenvolvidas e pegar a interseção dos resultados. Pode-se escrever uma nova estratégia com um teste complexo, ou compor uma estratégia usando o padrão Composite.

Um Composite é um objeto que faz duas coisas. Primeiro, ele implementa alguma interface, e segundo ele é um contêiner para itens que implementam a mesma interface. Então, quando o objeto composite recebe alguma mensagem, ele a repassa para o objeto que ele contém.

O código abaixo mostra uma estratégia de composite usando o operador lógico AND. Este implementa EstrategiaChecar e contém EstrategiaChecar. Um vetor é usado para conter os objetos.

```
class ComposicaoDeEstrategiaComAND implements EstrategiaChecar {
   private java.util.Vector testes = new Vector();

   public void adicioneEstrategia(EstrategiaChecar s) {
      testes.addElement(s);
   }

   public boolean checar(String s) {
```

```
        java.util.Enumeration e = testes.elements();
        while (e.hasMoreElements()) {
           EstrategiaChecar estrategia = (EstrategiaChecar)e.nextElement();
           if( !estrategia.checar(s) ) return false;
        }
        return true;
    }
}
```

Esta estratégia retorna falso se qualquer uma das estratégias adicionadas retorne falso para a mesma string. Caso contrário retorna verdadeiro. Note que esta retorna verdadeiro se não se adiciona estratégia alguma.

Abaixo se tem uma nova **ComposicaoDeEstrategiaComAND** adicionando-se outra estratégia a ela. O novo objeto pode ser usado como uma estratégia.

```
...
ComposicaoDeEstrategiaComAND palavrasMaioresQueComecamComT =
          new ComposicaoDeEstrategiaComAND();
palavrasMaioresQueComecamComT.adicioneEstrategia(new MaiorQueN(5));
palavrasMaioresQueComecamComT.adicioneEstrategia(new IniciaComT());
...
imprimaQuando("texto1.txt", palavrasMaioresQueComecamComT);
```

A chave do sucesso para o Composite é ter um container que implementa uma interface e este conter um objeto que também implementa a mesma interface. Quando o composite recebe uma mensagem definida na interface este a repassa para aqueles objetos que ele contêm. Normalmente é preciso adicionar ou remover elementos.

A Figura 4-10 mostra o diagrama das classes envolvidas nas atividades desenvolvidas nesta sessão.

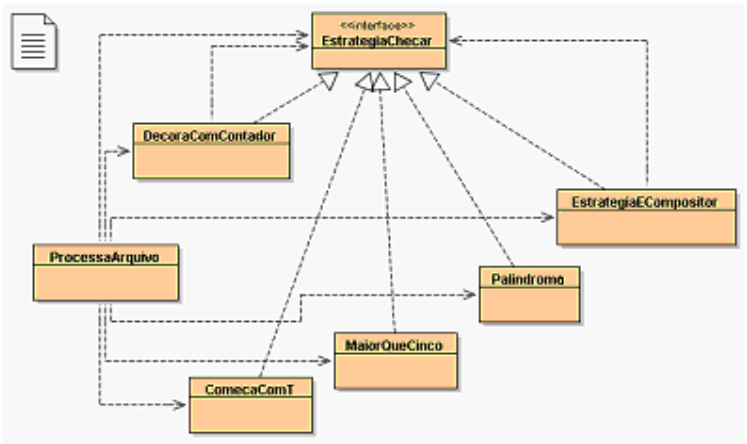

Figura 4-10 Diagrama de classes com as estratégias e o decorador

4.5.3 Calculadora

Outra atividade que contempla as técnicas puras de orientação a objetos, envolvendo os padrões de projeto é a construção de uma calculadora. Neste projeto o polimorfismo e os padrões de projeto norteiam a construção do modelo.

Este exemplo foi elaborado pelo professor Joseph Bergin [BER03] para mostrar o que pode ser feito dentro do paradigma de orientação a objetos utilizando seus conceitos fundamentais, sem usar *if* ou *while*. Este implementa as partes principais de uma calculadora simples com as quatro funções algébricas principais que não considera a precedência de operadores.

O código está dividido em partes. O modelo implementa a calculadora em si e a interface gráfica é totalmente separada, similar à arquitetura model-view-controller [BER03].

Este projeto foi utilizado por Bergin [BER03] em um curso de introdução à programação orientada a objetos com Java (para adultos). O código não foi apresentado para a turma, mas desenvolvido com eles com o auxílio de um projetor. Antes da demonstração os alunos foram expostos às idéias principais (encapsulamento, envio de mensagens e polimorfismo) através de atividades envolvendo os padrões pedagógicos, com várias **metáforas** e exercícios de **jogos de papéis** durante duas aulas.

O código foi completado durante a demonstração pelos alunos em duplas. O instrutor durante o desenvolvimento da aplicação motiva os padrões que são implementados no projeto, Strategy e Decorator. Existem outros padrões no projeto, Singleton, Null Object e Immutable.

Parte da filosofia educacional usada no desenvolvimento do código é que orientação a objetos é um novo paradigma e os estudantes precisam ser instruídos cedo a pensar no paradigma. O código resolve todo o problema usando encapsulamento, envio de mensagem e polimorfismo dinâmico. Este não tem um teste lógico explícito ou seleção. Quando duas opções ocorrem para uma ação, diferentes objetos estratégicos são utilizados, cada opção em uma estratégia. Isto aumenta o número de objetos e o número de mensagens, mas também torna cada “pedaço” da aplicação mais simples.

As estratégias implementam o comportamento das teclas, que pode ser de composição do número ou de armazenamento. Quando uma tecla de operação é pressionada é preciso salvar o display em algum lugar e limpá-lo novamente para que seja feita a composição do próximo número.

Bergin enumera razões para resolver o problema desta forma:

- Ensinar os alunos a resolver o problema de forma orientada a objetos, não imperativa.
- Colocar esta forma de pensar como a padrão, por isso a introdução é feita assim. Isto auxilia a evitar a troca de paradigmas.
- Esta forma de resolver o problema facilita a manutenção, uma vez que comandos *ifs* aparecem em vários lugares em programas maiores, dificultando a manutenção e na solução proposta aqui o ponto de decisão fica concentrado e mais fácil de alterar.

O diagrama de classes deste projeto pode ser visto na Figura 4-11.

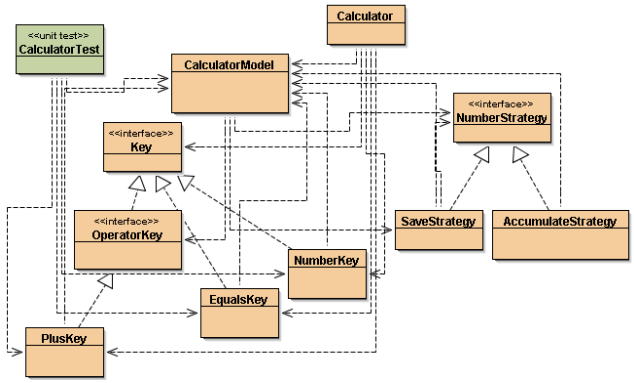

Figura 4-11 Diagrama de classes do projeto Calculadora

4.5.4 Composição Musical

O projeto de composição musical tem sido utilizado por vários anos para ensinar uma variedade de padrões de projeto [HAM04]. O diferencial do projeto é que os alunos são encorajados a descobrir e expressar seus talentos musicais e tem a possibilidade de perceber as conexões entre programação e arte.

Especificamente o projeto é utilizado para:

- Explorar os padrões de projeto: Composite, Decorator, Factory e Visitor.
- Aprofundamento no entendimento de programação através da exploração da analogia entre a estrutura do programa e a estrutura musical. Os elementos da analogia são:
  - Composição seqüencial – sequenciamento de comandos;
  - Composição paralela – vários processos;
  - Repetição musical – repetições;
  - Variação na composição – comandos condicionais.

- Desenvolvimento de habilidades na abstração de padrões e identificação de estruturas subjacentes.
- Apresentação de conexão entre a estrutura do *design pattern* e de gramáticas livre de contexto.
- Aplicação de vários métodos formais, tais como dar provas da equivalência de várias formas musicais.

Estrutura do Projeto

A Figura 4-13 exibe o diagrama de classes do projeto.

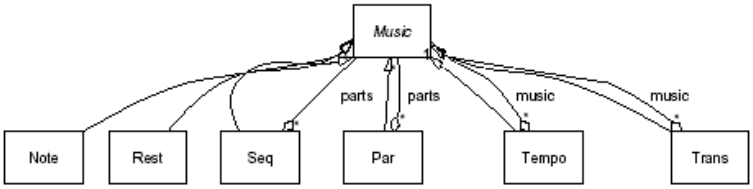

Figura 4-12 Diagrama de classes do projeto Musica [HAM04]

Composições musicais são descritas usando o padrão Composite e Decorator (Figura 4-3). **Note** e **Rest** são termos primitivos. Uma **Note** corresponde a uma tecla pressionada e mantida assim por algum tempo (medido em uma unidade não específica de "half" ou "quarter notes", etc.). Um **Rest** introduz um atraso antes que a próxima nota seja tocada. **Notes** e **Rests** podem ser organizadas em grandes unidades com os construtores **Seq** e **Par**. As notas em uma **Seq** são tocadas em seqüência (um arpejo) e as notas em um **Par** são tocadas simultaneamente (um acorde).A classe abstrata **Music** permite arbitrariamente o aninhamento dos construtores musicais.

Quatro classes Decoradas – Tempo, Transpose, Instrument e Phrase cada uma modifica um termo musical de alguma forma. **Tempo** indica o termo de fechamento musical, mais rápido ou mais lento. **Transpose** aumenta ou diminue o pressionar de cada nota por um offset. O instrumento a ser usado para sintetizar a música pode ser fixado usando **Instrument**. **Phrase** descreve como a música deve ser tocada. A maioria das músicas requer habilidades musicais para serem interpretadas, mas algumas são agradáveis por algoritmos no computador.

Fábricas de Note e Rest

O padrão Factory Method tem motivação dupla ao construir **Notes** e **Rests**: facilita a criação do objeto, evitando o uso do **new** explicitamente e o compartilhamento de ocorrências repetidas da mesma nota (o compartilhamento surge como uma necessidade natural).

Uma nota, por exemplo “quarter midlle C” pode ser construída com:

```
Note middle_C_quaver = new Note(32,0.25);
```

A falta de clareza desta construção motiva a introdução de uma série de funções auxiliares para construir notas específicas de uma determinada duração. Por exemplo:

```
Note c (int octave, double duration) {
        return new Note(octave*12,duration);
}

Note d (int octave, double duration) {
        Return new Note(ocatave*12+2,duration);
}

// etc.

Note sharp (Note orig) {
        Return new Note(origin.pitch()+1,orig.duration());
}

Note flat (Note orig) {
        Return new Note(origin.pitch()-11,orig.duration());
}
```

“Quarter middle C” pode ser reescrito como:

```
Note middle_C_quaver = c(4,0.25);
```

4.5.5 Kaleidoscópio

Kaleidoscópio é uma aplicação proposta por Wick [WIC00] que efetivamente demonstra dois *design patterns* clássicos (Model-View-Controler e Factory Method) de forma simples e interessante de modo a prender a atenção dos alunos de hoje.

O padrão Model-View-Controller

O *design pattern* Model-View-Controller (MVC) lida com a divisão de responsabilidade entre as interfaces no modo texto e as interfaces gráficas do projeto. Neste projeto existe uma divisão clara entre os objetos que têm a responsabilidades de encapsular os dados e os objetos que têm a responsabilidade de comunicar com o usuário. O objeto do tipo **Model** provê todo o comportamento necessário para manipular os dados. O objeto do tipo **View**, por outro lado, tem a responsabilidade primária de mostrar o modelo para o usuário. Toda vez os dados do modelo são alterados, o modelo notifica as visões registradas da alteração do dado, solicitando que cada visão atualize seu display apropriadamente.O objeto do tipo **Controller** tem a responsabilidade principal de pegar comandos do usuário e solicitar o método de manipulação do dado apropriado no modelo registrado.

A Figura 4-14 apresenta o padrão Model-View-Controller.

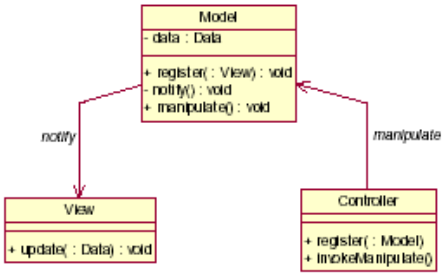

Figura 4-13 Modelo de classes do padrão Model-View-Controller [HAM04]

No projeto Kaleidoscópio [WIC00] um contêiner de círculos e quadrados é utilizado para ilustrar este padrão. Neste exemplo, o modelo (objeto do tipo *Model*) é um container de figuras. Uma visão da classe (objeto do tipo *View*) desenha uma representação gráfica de círculos e quadrados. A Figura 4-15 mostra um exemplo de um caleidoscópio.

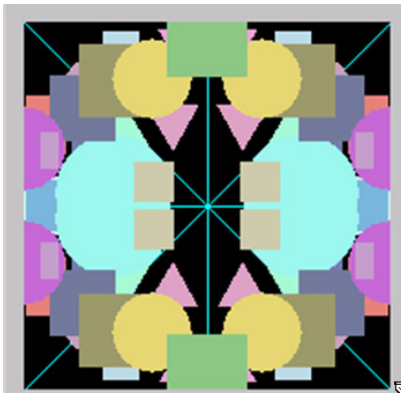

Figura 4-14 Exemplo de um caleidoscópio [HAM04]

Um controlador (objeto da classe *Controller*) provê um botão para adicionar círculos ou quadrados no container.

A divisão no modelo em três responsabilidades distintas tem muitas vantagens incluindo a possibilidade de prover várias formas de visualizar o mesmo dado, centralização do código de manipulação do dado e desvinculação do processo da visualização.

Este padrão foi incluído como o primeiro na seqüência por dois motivos. Primeiro, é um padrão de uso geral e pode ser aplicado a qualquer desenvolvimento usando interfaces gráficas, o que assegura que os alunos aprendam como usá-las adequadamente. Segundo, o padrão é um ótimo exemplo de projeto como projeto dirigido por responsabilidade influencia a identificação de classes e a atribuição de responsabilidades das classes [WIC00]. A

desvinculação dos dados da visão dos mesmos é importante porque normalmente o aluno imagina que a interface gráfica deva estar integrada com os dados. Com o uso do *design pattern* o projeto do software ganha em poder e flexibilidade.

O padrão Abstract Factory e Factory Method

O *design pattern* Abstract Factory é usado para habilitar uma classe a manipular objetos através de uma referência a uma classe abstrata a criar novas instâncias de classes derivadas da classe abstrata. Usando polimorfismo, um objeto requisitante encapsula um processo que funciona para qualquer objeto produzido. Quando o requisitante solicita uma nova instância do produto, uma fábrica é usada para fornecer a nova instância. A fábrica encapsula o processo de criação dentro do método de construção que é um método de fabricação de instâncias (*design pattern Factory Method*).

O exemplo clássico deste padrão envolve um sistema de janelas. Quando o usuário clica no botão "abrir", o mesmo processo genérico abre a janela considerando o tipo de aplicação que está sendo aberta.

Este padrão capacita um processo geral solicitar novas instâncias de uma classe abstrata sem se preocupar com o processo de criação da instância. A classe do solicitante encapsula o processo genérico, a classe ***Product*** encapsula o dado manipulado pelo processo e a classe ***Factory*** encapsula o processo de criação do produto.

A seleção deste padrão também é um exemplo de projeto dirigido por responsabilidade que pode ser usado para construir sistemas de softwares mais flexíveis. Outro fator para a escolha é que este é uma introdução simples de algumas idéias principais por trás do *framework* de projetos. Em particular, ilustra a idéia que um processo geral (definido pelo objeto solicitante) que pode ser ligada a uma aplicação concreta específica através de uma implementação particular do passo genérico (no caso a criação de objetos).

A Figura 4-15 mostra o diagrama do padrão.

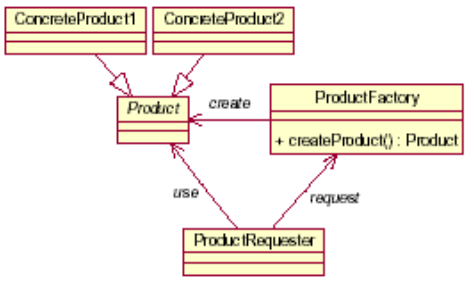

Figura 4-15 Modelo de classes do padrão Factory (Abstract Factory e Factory Method) [HAM04]

O Kaleidoscópio

O Kaleidoscópio é basicamente uma coleção de formas. Quando o caleidoscópio é "ligado", as formas existentes são randomicamente movimentadas e uma nova forma é adicionada à coleção.Em algum momento, o visor do caleidoscópio mostra s coleção atual de formas através da disposição simétrica nos quatro quadrantes [WIC00]. Com um número suficientemente grande de giros, o display começa a tomar a forma de um caleidoscópio.

O diagrama da Figura 4-16 mostra o diagrama do projeto.

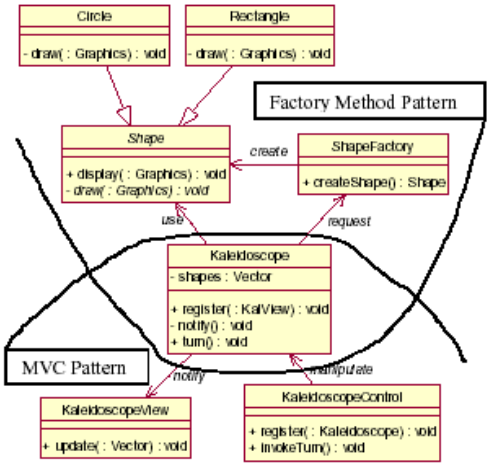

Figura 4-16 O modelo estático do projeto Kaleidoscópio [HAM04]

A Tabela 4-3 sintetiza as classes envolvidas no modelo.

Tabela 4-3 Classe envolvidas no projeto Kaleidoscópio

Classe	Descrição
Shape	Classe abstrata que representa a raiz da hierarquia de formas. Encapsula o dado que a classe genérica *Kaleidoscope* manipula. Classes como *Triangle*, *Circle* e *Rectangle* são diretamente derivadas de *Shape*. A classe *Shape* armazena o centro e a cor de cada forma. O método display (*Graphics*) é responsável por todo o comportamento comum no processo de exibição de qualquer forma e nesta aplicação realiza o display em dois passos: (1) coloca a cor da forma e a (2) desenha. Cada classe derivada de *Shape* deve definir o método *draw* (*Graphics*) que realiza o segundo passo.
	Considerações: • A hierarquia de *Shape* fornece uma introdução relativamente fácil aos conceitos de herança, classes abstratas e refinamento de métodos. O uso de métodos de propósito gerais (*display* e *draw*) salienta a divisão apropriada de responsabilidade entre as classes base e derivadas. • No diagrama do padrão Factory Method, da Figura 4-5, a classe *Shape* representa a classe abstrata *Product*.
Kaleidoscope	A classe *Kaleidoscope* é o repositório de todos as formas. O método *turn*() causa a movimentação randômica das formas existentes, adiciona uma nova forma à coleção e notifica a visão registrada para atualizar o display. Neste projeto esta classe serve para juntar os dois *design patterns*. Sob o ponto de vista do MVC, esta classe serve como o modelo, encapsulando o dado a ser manipulado (coleção de formas). Sob o ponto de vista do Factory Method, esta classe serve como o solicitante (*requester*),

	solicitando novas instâncias da classe *Shape* durante cada movimentação do caleidoscópio.
ShapeFactory	A classe *ShapeFactory* encapsula o processo de criação das formas e, como tal, serve como uma classe de fábrica no *design pattern Factory Method*. Cada vez que uma nova forma é solicitada, o método *createShape*() é invocado para determinar os detalhes específicos da classe concreta resultante (*Square*, *Rectangle*, *Triangle*). Além de conter a lógica de como selecionar um novo *Shape* dentro do método *turn*(), a responsabilidade de criar um novo *Shape* é encapsuladas nesta classe. Isolando a decisão de como criar a próxima forma da classe *ShapeFactory*, kaleidoscópio pode facilmente acomodar estratégias alternativas de construções sem a necessidade de ser modificada.
KaleidoscopeView	A classe *KaleidoscopeView* tem a principal responsabilidade de exibir as formas pelo próprio kaleidoscópio. O método *update*() invoca o código apropriado para desenhar a forma na tela. O coração do processo de exibição é a solicitação do método *display*() de cada *Shape* da coleção. O método *display*() realiza dois passos, incluindo a solicitação do método *draw*() para efetivamente desenhar a forma. Esta classe serve como uma visão no MVC.
	Considerações: • A solicitação do método *draw*() através de uma classe abstrata provê um exemplo excelente de tipagem dinâmica e polimorfismo.
KaleidoscopeControl	A classe *KaleidoscopeControl* tem a responsabilidade principal de enviar entradas para o kaleidoscópio que servem como uma classe de controle no MVC. O único método *invokeTurn*() envia uma mensagem *turn*() ao kaleidoscópio onde quer que o usuário clique na interface do *KaleidoscopeControl*.

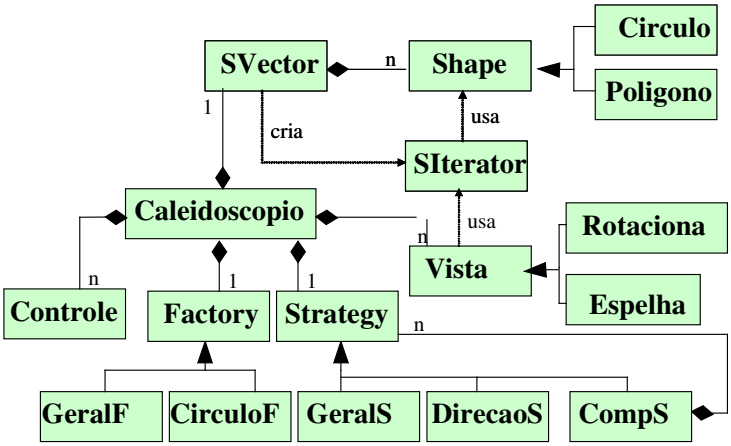

Figura 4-17 O diagrama de classes do projeto Kaleidoscópio [HAM04]

Sugestões Pedagógicas

O diagrama de classes mostrado na Figura 4-18 é muito avançado para uma única atividade. Uma forma mais efetiva de trabalhar com o projeto é o desenvolvimento em partes através dos seguintes passos [WIC00].

1. Tratar herança ou classe abstrata com a hierarquia de classes *Shape*. O uso da classe abstrata *Shape* é natural para mostrar o valor de uma classe abstrata e tipos dinâmicos.

2. Reutilizar a hierarquia de classe *Shape*, forneça aos alunos comandos para a interface gráfica que desenvolvem os conceitos centrais *design pattern* MVC. Um exemplo simples é um programa que permite o usuário clicar em botões para colocar figuras em um display *canvas*.
3. Novamente reutilizar a hierarquia de classes *Shape*, e construir na experiência do projeto MVC, forneça os comandos do *Kaleidoscope* sem incluir explicitamente o *design pattern* Factory Method. A maioria dos alunos irão gerar soluções que produzem novas formas usando comando "*switch*" no método *turn*() do kaleidoscópio. Estas atividades permitirão aos alunos ganhar experiência na integração de múltiplas classes em uma solução para um problema e dá uma base para motivação de uso de outros *design patterns*.
4. Atribua à versão do padrão *Factory Method* do programa *kaleidoscópio*. Esta tarefa focaliza no padrão em si. Alunos gostam o tipo de projeto proporcionado pelo padrão *Fatory Method*.
5. Neste passo o aluno deve ter uma visão adicional ou controlador (ou ambos) para o projeto. Por exemplo, uma segunda visão que encobre oito cópias das formas na coleção. Com esta tarefa, os alunos percebem o valor da separação da visão e controle do modelo.
6. Para mostrar o poder de *Factory Method* solicite aos alunos que redefinam a classe *ShapeFactory* como abstrata e crie pelo menos duas subclasses concretas (por exemplo *PolygonShapeFactory* e *EllipseShapeFactory*). A classe *Kaleidoscope* é então modificada para incluir um parâmetro no construtor que permita a instanciação de específicas classes concretas (*ShapeFactory*) a serem usadas. Esta atividade ajuda no entendimento do poder de separar o processo de criação da forma do processo do kaleidoscópio.
7. Neste passo os alunos podem ver como eliminar o acoplamento entre as classes *Kaleidoscope* e *KaleidoscopeView* (observe que um *Vector* é compartilhado entre as duas). Utilize o *desing pattern iterator* para permitir a comunicação entre as classes. Esta tarefa ajuda a ilustrar como o padrão *iterator* auxilia o encapsulamento de coleções utilizadas no programa.

4.6 PADRÕES DE CODIFICAÇÃO PARA INICIANTES

Alunos que estão começando a programar cometem muitos erros. Por um lado por pouca experiência, por outro por falta de bons direcionamentos. Os instrutores esperam que, analisando programas prontos, eles sigam o padrão de codificação adotado, porém nem sempre é assim. Existe uma série de padrões de codificação que podem e devem ser

incentivados. Bergin [BER01b] expõem boas práticas e conselhos muito úteis para o desenvolvimento de programas. Estes padrões de codificação podem ser seguidos pelos instrutores e sugeridos aos alunos. Abaixo segue sua listagem dos principais tipos de padrões de codificação. Eles podem ser agrupados nas seguintes categorias: Planejamento, Estrutural, Manutenibilidade, Projeto, Segurança Estilo e Implementação.

- Programação em Par (Manutenibilidade)
- Tempo nas Tarefas (Planejamento)
- Crescimento Gradativo (Planejamento)
- Um Serviço Por Classe (Estrutural)
- Polimorfismo antes de Seleção (Estrutural)
- Interface Lógica (Manutenibilidade)
- Interface Completa (Projeto)
- Forte Encapsulamento (Segurança)
- Reescrever (Planejamento)
- Aperfeiçoar para facilitar a Leitura (Manutenibilidade)
- Nomes que revelam intenções (Manutenibilidade)
- Nomes Consistentes (Manutenibilidade)
- Elementos públicas primeiro (Estilo)
- Campos Privados (Segurança)
- Inicializar (Segurança)
- Teste Tudo Agora (Segurança)
- Atribua Valores uma Única Vez (Projeto)
- Método Pequeno (Estrutural)
- Métodos Compostos (Implementação)
- Objeto de Método (Implementação)
- Comente (somente) quando necessário (Manutenibilidade)
- Diga uma Vez (Estrutural)
- Indentação para Estrutura (Estilo)
- Coloque tudo entre um par de chaves { } (Estilo)
- Indicador de início de estrutura composta deve ficar na direção exata e logo abaixo do início da estrutura (Estilo)
- Dê espaços não tabulação (Manutenibilidade)
- Capitalização Consistente (Estilo)
- Denomine suas constantes (Manutenibilidade)
- Siglas fora (Manutenibilidade)
- Elementos Locais (somente) quando necessário (Projeto)
- Um Comando por Linha (Estilo)

- Linhas curtas (Manutenibilidade)
- Funções para Condições Complexas (Estrutural)
- Escreva "*this*." (Manutenibilidade)
- Variáveis Locais são associadas novamente antes do uso anterior (Projeto)

A conclusão dos estudos apresentados neste capítulo nos conduz à utilização destes três tipos de padrões na construção de um ambiente de aprendizagem de programação orientada para objetos. No Capítulo 6, está a proposta da Estação OO que procurou incorporá-los e utilizados na medida em que se mostraram importantes para o processo.

Capítulo 5

5 Estudos de Caso: IDEs Didáticas

Neste capítulo são descritos os esforços desenvolvidos no sentido de conhecer, validar e criticar as IDEs didáticas consideradas mais promissoras nos processos de ensino e aprendizagem, descritas no Capítulo 3 e as práticas pedagógicas, descritas no Capítulo 4.

A migração de uma IDE didática para uma profissional, como o Eclipse [IBM04] e/ou NetBeans [NET05], foi realizada ao final do curso. Esta atividade permitiu analisar a migração de ambientes e verificar a adaptação e a reação dos alunos ao novo ambiente.

Nos projetos propostos os *design patterns* foram utilizados de forma implícita, contudo, na contextualização, com a participação dos alunos, soluções de problemas semelhantes foram reformuladas, considerando a experiência particular em projeto e a idéia do padrão utilizado.

Em todos os cursos os alunos assumiram a postura ativa de investigadores, isto é, faziam o reconhecimento do ambiente e iniciavam as atividades sugeridas pelo instrutor que eram colocadas como desafios/projetos. Ao desenvolver as soluções para os problemas propostos, surgia a necessidade de tomar certas decisões, repetir determinadas tarefas, melhorar trechos de programas repetitivos e generalizar soluções. Em respostas a estas necessidades, a sintaxe da linguagem era apresentada.

O instrutor, por outro lado, assumia a postura de mediador, o elemento que facilitava a comunicação do aluno com o computador, através do paradigma de orientação por objetos e da linguagem de programação Java. Como os alunos trabalhavam no seu ritmo, o conteúdo foi trabalhado de acordo com suas necessidades na atividade que desenvolvia. Desta forma, somente após terem trabalhado com o assunto é que os conceitos eram formalizados. Quando a dúvida era geral, as atividades eram interrompidas e as experiências individuais, do aluno e do instrutor, eram expostas.

Para estes estudos de caso, foram desenvolvidos materiais didáticos que foram aplicados em turmas fechadas com perfil variado. No item 5.1, a seguir, serão descritos os cursos aplicados. Os materiais didáticos desenvolvidos como material de apoio aos cursos encontram-se em anexo: Anexo I e II.

5.1 CURSOS APLICADOS

Foram aplicados quatro cursos. Cada curso explorou uma determinada combinação de IDEs pedagógicas, julgadas as mais promissoras, e a ênfase em alguma estratégia de ensino/aprendizagem (vistos no Capítulo 4). IDEs Profissionais foram também utilizadas em certos casos como indicado acima.

Os cursos aplicados foram:

- Curso 1: Introdução à Programação Orientada a Objetos com Java – AllKara.
- Curso 2: Curso de Java em Tópicos Especiais I com AllKara e Eclipse.
- Curso 3: Curso de Java em Programação II com o BlueJ, Eclipse.
- Curso 4: Curso de Java em Tópicos Especiais I com Jogo de Papéis, BlueJ e Eclipse

As tabelas que seguem, Tabela 5-1 a Tabela 5-4 contêm sínteses dos cursos oferecidos.

Tabela 5-1 Curso 1: Introdução à Programação Orientada a Objetos com Java - AllKara

Período	**Duração**	**Público**	**Ferramenta**
05/01/04 a 23/01/24	2 horas diárias	Alunos de diferentes cursos da UFV, portanto heterogênea com níveis diferenciados de conhecimento em programação de computadores.	AllKara: máquina de estados finitos e JavaKara
Atividades	• As atividades se baseavam em desafios propostos. Nestes desafios uma situação inicial era proposta e o aluno conduzia Kara ao objetivo final. As atividades foram desenvolvidas em sala de aula. • A introdução à programação foi dividida em três partes: Programação com a Máquina de Estados Finitos, JavaKara e Orientação por objetos com JavaKara, como pode ser observado no Anexo I. • As atividades (desafios) eram as mesmas tanto para a programação com a Máquina de estados finitos quanto com JavaKara. Estas estão reunidas no Anexo I, parte 1 e 2. • Durante as aulas práticas os alunos desenvolviam as atividades individualmente nos computadores. • Regularmente dúvidas e soluções eram apresentadas pelos alunos. Esta apresentação colaborava com o andamento do curso e facilitava o crescimento do grupo como um todo.		

Observações	• Uma das vantagens de AllKara é a possibilidade de instruir Kara de diferentes formas. Neste estudo apenas duas foram utilizadas: interpretada e através de um arquivo de comandos (programas) que era compilado antes da execução. Ao iniciar, o usuário tem a opção de comandar Kara diretamente, em um esquema interpretado. Desta forma, alunos menos experientes, ou com mais dificuldade de formalizar suas soluções, puderam anotar a seqüência de instruções dadas e posteriormente montar o programa em JavaKara, que na seqüência seria compilado e executado. • Com os formalismos selecionados, a programação foi introduzida por duas vias distintas. • A continuidade do ambiente, o universo de Kara é o mesmo para as várias ferramentas disponibilizadas no micromundo, facilita a mudança na forma de programar Kara, levando naturalmente à necessidade de outro formalismo, que neste caso foi JavaKara. • O enfoque do curso foi a linguagem Java, seus construtores e sintaxe e não o paradigma de orientação por objetos. As novas idéias do paradigma não foram utilizadas diretamente, embora estivessem implícitas no ambiente. Colaborou para este fato a pouca experiência da autora deste trabalho em orientação por objetos na época do curso e a forte influência do paradigma estruturado na formação e na postura de professora espelhada no modelo tradicional de ensino, ou seja, "ensinar" a linguagem e não as idéias principais por trás do paradigma. Neste curso foi adotada uma forma de condução diferente, mas não uma nova pedagogia para o novo paradigma. Este aspecto coincide com as observações de Bergin [BER00c], mencionadas no Capítulo 2. • Em entrevistas pessoais ao final do curso, os alunos mencionaram que apreciaram as ferramentas, considerando-as fáceis de utilizar e destacando que a forma de colocar desafios de descrever as soluções em termos da máquina de estados finitos e depois em JavaKara foi um bom exercício mental e que contribuiu para desenvolver o raciocínio lógico. • Considerando o seu objetivo, proporcionar experiências com formalismos da Ciência da Computação, o micromundo é uma excelente porta de entrada. Porém, não é adequado para avançar nos conceitos fundamentais de orientação a objetos.

Tabela 5-2 Curso 2: Curso de Java em Tópicos Especiais I com AllKara e Eclipse

Período	Duração	Público	Ferramenta
02/02/04 a 26/06/04	quatro aulas semanais	Alunos do curso de Sistemas de Informação da disciplina Tópicos Especiais I da Faculdade de Viçosa.	AllKara e Eclipse.
Atividades	• Este curso seguiu a estrutura do anterior com as três partes distintas. Porém, os alunos migraram, no final, para o ambiente Eclipse, IDE profissional e gratuita, desenvolvida e disponibilizada pela IBM [IBM04]. O material básico utilizado no curso foi o do Anexo I. • Após a introdução à programação orientada a objetos com JavaKara os alunos desenvolveram classes no Eclipse. • Durante as aulas práticas os alunos desenvolveram as atividades individualmente nos computadores. • Regularmente as dúvidas e as soluções eram socializadas. As soluções aos problemas propostos eram discutidas e as diferentes soluções eram destacadas.		
Observações	• Os alunos puderam utilizar a máquina de estados finitos na prática. O formalismo foi abordado na teoria, em disciplinas anteriores, porém não utilizado de forma prática. • Segundo relato de alunos em entrevistas ao final do curso, a programação da máquina de estados finitos permitiu um bom exercício de lógica. • Com JavaKara foi possível o entendimento de estruturas de controle do paradigma imperativo como estruturas de seleção e de repetição. Alguns alunos tinham muita dificuldade com programação de computadores e comentaram que "a coisa não era tão complicada". • Alguns alunos da turma, que exerciam atividades de programação de modo profissional em outras linguagens, criavam classes utilizando a sintaxe Java, porém apresentavam o viés da programação estruturada. Com o decorrer das atividades foram "incorporando" a nova forma de ver os problemas e as soluções sob a ótica dos objetos. • A utilização dos objetos do micromundo de Kara foi a motivação inicial para a apresentação dos conceitos iniciais de orientação a objetos. Neste curso, ao contrário do anterior, estes aspectos foram enfatizados. • Os conceitos fundamentais de orientação por objetos foram trabalhados, mas o entendimento ficou a desejar. Ao final da disciplina, a partir da análise das avaliações aplicadas, ficou evidente que alguns alunos ainda mostravam sinais de não terem entendido os conceitos fundamentais. Da análise final da aplicação deste curso, ficou claro que o micromundo facilita a introdução à programação orientada a objetos, mas não é suficiente para um curso introdutório. O Eclipse (IDE profissional) foi utilizado como próxima IDE, porém, uma lacuna ficou evidente. A partir daí a busca por ferramentas para preenchê-la foi empreendida, originando a investigação sobre IDEs didáticas e uso de ferramentas da Engenharia de Software em cursos introdutórios.		

Tabela 5-3 Curso 3: Curso de Programação Orientada a Objetos com o BlueJ.

Período	Duração	Público	Ferramenta
28/07/2004 a 25/11/2004	quatro aulas semanais	Alunos do curso de Sistemas de Informação na disciplina Programação II da Faculdade de Viçosa.	BlueJ e Eclipse
Atividades	• Neste curso o BlueJ foi adotado e com ele sua metodologia de ensino. O livro texto "Programação Orientada a Objetos com Java – Uma introdução prática usando o BlueJ" [BAR03] foi adotado. • As atividades desenvolvidas na disciplina constituem projetos que foram cuidadosamente preparados, pelos próprios autores do livro Barnes e Kolling e/ou colaboradores, para facilitar a abordagem de projetos mais complexos e conceitos principais desde o início. Alterações no código também são incluídas nas atividades. Na ordem os seguintes projetos foram trabalhados: o *Shapes*, Picture e Lab_*Classes*: introduzir os conceitos de objetos, classes, métodos, parâmetros, assinatura, tipo, estado, chamada de métodos, estado, resultado, código fonte e compilação (disponíveis no Cap. 1 do livro). o *TicketMachine*, *Lab_Classes* e Book: introduzir conceitos de campos, construtor, escopo, tempo de vida, atribuição, métodos de acesso, métodos modificadores, estrutura condicional e variável, disponíveis no Cap. 2 do livro). o *Clock_Display* e *Mail_System*: utilizar os conceitos de modularização, diagrama de classes, diagrama de objetos, referências de objetos, tipo primitivo, criação de objetos, sobrecarga, chamada de método interno, chamada de método externo e depurador (disponíveis no Cap. 3 do livro). o *Note_Book*, *Auction* e *Weblog_Analyzer*: utilizar coleções de objetos, estruturas de repetição, iteradores, objeto null e vetores (disponíveis no Cap. 4 do livro). o *TechSupport*: trabalhar os conceitos de interface, implementação, coleção do tipo map, coleção do tipo set, modificadores de acesso, ocultamento de informação, variáveis de classes e variáveis estáticas, documentação, utilização da biblioteca padrão Java e documentação da biblioteca (disponíveis no Cap. 5 do livro). o Projeto de classes: acoplamento, coesão, duplicação de código, coesão de métodos, coesão de classes, encapsulamento, design baseado em responsabilidades, minimizar as alterações e refatoração (disponíveis no Cap. 7 do livro).. o DoME (*Database of Multimedia Entertainment*): introduzir o conceito de herança, reutilização, subtipos, classe Object, variáveis e subtipos e substituição (disponíveis nos Cap. 8 e 9 do livro). o DoME: trabalhar os conceitos de tipo estático, tipo dinâmico, sobrescrição, polimorfismo de método, *toString* e acesso do tipo *protected*. o Raposas e Coelhos: trabalhar os conceitos de classes abstratas e interfaces (disponíveis no Cap. 10 do livro). • A pedagogia em espiral, um dos padrões pedagógicos citados na no Capítulo 4, na sessão 4.2.7, é utilizada nos projetos e suas atividades. • Durante as aulas práticas os alunos trabalharam em duplas, o que facilitou a troca de experiências e exercitou a ajuda mútua. • Ao final, foi feito o desenvolvimento do projeto Concessionária_de_Veículos utilizando o Eclipse para a experiência de desenvolvimento em uma IDE profissional.		

Observações	• Com o BlueJ o entendimento dos conceitos fundamentais é muito facilitado. O próprio diagrama de classes e a possibilidade de instanciar uma classe clicando sobre ela com o botão direito do mouse, selecionando **new**, disponibilizando o objeto na bancada de objetos e a possibilidade de inspecioná-lo é uma poderosa ferramenta para cursos introdutórios. • O BlueJ é uma IDE pedagógica que segue a orientação dos padrões pedagógicos descritos no Capítulo 4. Esta ferramenta constitui uma abordagem diferente (sessão 4.2.1); permite a exploração individual (4.2.6); motiva e facilita as experiências, sendo, portanto, um verdadeiro tubo de ensaio (sessão 4.2.9) e permite que o instrutor crie situações que levem os alunos à reflexão (sessão 4.2.5). • Segundo relatos de alunos, em entrevistas ao final do curso, o BlueJ facilitou o entendimento de conceitos de orientação por objetos através da forma como permite abordar e utilizar os elementos da classe. • BlueJ é uma ferramenta que permitiu aceleração da apresentação de conceitos importantes, sem a necessidade de entrar em detalhes que a princípio ficam obscuros para os alunos, como o método main, a criação de objetos e solicitação de serviços. • Com os projetos estudados, ocorreu o uso implícito de alguns *design patterns* (discutidos no Capítulo 4) como alta coesão/baixo acoplamento (sessão 4.4.2), Polymorphism (sessão 4.4.4), Delegation (sessão 4.4.5), Interface (sessão 4.4.6) e Abstract Superclass (4.4.7). Não foi possível verificar se os alunos adotariam os padrões em problemas correlatos, o que exigiria um curso mais extenso.

Tabela 5-4 Curso 4: Curso com o Jogo de Papéis, BlueJ e Eclipse

Período	Duração	Público	Ferramenta
01/03/2005 a 30/06/2005	quatro aulas semanais	Alunos do curso de Sistemas de Informação na disciplina Tópicos Especiais I da Faculdade de Viçosa.	BlueJ e Eclipse
Atividades	• As atividades do curso foram tratadas em unidades como projetos, como nos cursos anteriores. • Cada projeto, em um nível diferente de profundidade, abordou conceitos que são revistos em projetos posteriores, utilizando a pedagogia em espiral, como mostrado no Capítulo 4, sessão 4.2.7. • Os projetos procuraram permitir ao aluno "ver" classes e seus objetos de formas diferentes, como proposto no padrão pedagógico Abordagens (sessão 4.2.1). Foram adotadas as seguintes óticas: o **Jogo de papéis**: padrão pedagógico descrito no Capítulo 4 (sessão 4.2.4) que corresponde a uma atividade em que o aluno participa como objeto de um determinado papel (tipo ou classe informalmente descrita) e, durante a execução do roteiro do instrutor, os objetos (representados pelos alunos) são criados e executam ações sob a solicitação do instrutor. o **Descrição em UML**: após o entendimento da classe e conhecimento dos papéis envolvidos, uma descrição da classe em UML é mostrada para os alunos. o **Manipulação das classes no computador**: as classes prontas são manipuladas no computador, usando o BlueJ. o **Código-fonte**: após sua manipulação, o código fonte da classe é estudado. • Os projetos utilizados neste curso, que se encontram no Anexo II, foram: ▪ Acrobata & Cia: fornece uma visão geral do mundo dos objetos. ▪ Figuras Geométricas: interfaces e polimorfismos. ▪ Perguntas e Respostas: coleções e estrutura de seleção. • O material desenvolvido encontra-se no Anexo II.		
Observações	• O jogo de papéis em si é um tipo de padrão pedagógico e pode e deve incluir também outros padrões pedagógicos mencionados no Capítulo 4. • A abordagem permitiu ilustrar conceitos fundamentais da tecnologia de objetos. o Classes, classes abstratas, interfaces, campos, construtores, métodos, visibilidade dos elementos, parâmetros, tipo de retorno, herança, composição, polimorfismo, sobreposição de métodos. • Os padrões de projeto, Capítulo 4, foram utilizados na construção dos projetos do curso implicitamente. • Em entrevistas durante e após o curso os alunos relataram que o estudo de orientação a objetos comportando-se como objetos facilitou o entendimento de conceitos fundamentais. Dessas discussões, ao perceber que alguns alunos continuavam com dúvidas, se incluiu a possibilidade de visualizar o diagrama dos objetos envolvidos durante todo o jogo, com sua construção paralela às solicitações de serviços e à elaboração da descrição das solicitações na linguagem Java. • Este curso confirmou a sugestão de que a classe deve ser estudada do nível externo para o interno, ou seja, partindo-se de uma descrição informal, com uma primeira formalização em UML, manipuladas no computador pelo uso de uma IDE didática até alcançar a formalização através do código-fonte na linguagem de programação (Java). • Durante o curso ficou claro que esta abordagem facilitou o entendimento das classes trabalhadas e o projeto de novas, além disso, a abordagem do código fonte em Java foi extremamente facilitada. • A utilização de projetos e a pedagogia em espiral mostraram-se muito produtivas. Este tipo de abordagem permite que as atividades envolvam novos conceitos à medida que são desenvolvidas, dando possibilidade tanto ao professor quanto ao aluno de propor mudanças (desafios) que normalmente envolvem novos conceitos ou permitem sua revisão.		

CAPÍTULO 6

6 ESTAÇÃO OO

6.1 INTRODUÇÃO

Uma estação é um local onde um meio de transporte estaciona para permitir que os passageiros entrem e saiam. Apesar de variar muito, uma estação inclui plataformas, controles de embarque e desembarque, venda de ingressos, salas de espera, etc. Mas uma estação pode ser vista, metaforicamente, como uma ponte para o mundo. Utilizando diferentes rotas, os usuários podem chegar a um determinado local. Fatores pessoais, como necessidades, recursos financeiros, determinam a forma de empreender a viagem, porém o mais importante é que, de uma forma ou de outra, o objetivo possa ser alcançado. Com este enfoque este capítulo apresenta a Estação OO. Uma estação que permite o embarque para o mundo dos objetos por diferentes meios, que pode ser escolhido pelo usuário ou indicado pelo instrutor.

O principal objetivo desta proposta é disponibilizar vias alternativas para se ensinar e aprender conceitos básicos de programação orientada a objetos, através da realização de atividades práticas que os conduza ao entendimento dos novos conceitos introduzidos em disciplinas relacionadas ao tópico.

Como toda estação, real, Estação OO deve apresentar mapas e roteiros sobre como atingir determinados objetivos. Neste trabalho o esforço foi concentrado na definição de um único roteiro, embora outros roteiros possam ser incorporados no futuro. Este roteiro mostra como utilizar IDEs didáticas disponíveis no mercado e a definição de requisitos de interface [FIL01] do sistema proposto, que será descrito na sessão 6.4 deste trabalho. A necessidade de uma nova IDE didática foi discutida no Capítulo 5 e procura completar uma lacuna percebida entre as propostas de IDEs didáticas encontradas.

O esquema da Figura 6-1 apresenta o mapa de um caminho (roteiro) para a introdução à programação orientada a objetos usando Java, estabelecendo as conexões entre as ferramentas (IDEs e micromundos) e os tópicos abordados até aqui.

Cada projeto incluído na Estação OO é um conjunto de atividades que cobre determinados tópicos. Assim, o usuário pode selecionar atividades de sua preferência e estilo. Esta versatilidade de atividades contempla diferentes tipos de aprendizagem [FEL88,

BER02]. A abordagem em espiral dos conceitos é adotada, isto é, um projeto aborda determinados conceitos que são revistos em projetos subseqüentes com maior profundidade [BER02].

Outra característica importante é que os projetos apresentados em Estação OO introduzem a programação considerando o modelo *object-first*, classificação feita de acordo com o Currículo de Computação proposto em 2001 pela ACM [ACM01], com uma mesclagem com *model-first* [BEN04]. Assim, o modelo subjacente ao problema pode ser visto em três níveis diferentes de formalismos na Estação OO: uma modelagem conceitual informal, descrevendo os conceitos chaves do domínio do problema; um modelo de classe, dando uma visão detalhada da solução e uma implementação em linguagem orientada a objetos [DEC03, BEN04].

A Estação OO é constituída por um mapa para cursos introdutórios e uma sugestão de roteiro de viagem composto por atividades inter-relacionadas. Para apoiar as atividades, motivando e facilitando a experimentação dos conceitos, micromundos e IDEs didáticas são utilizadas como ferramentas auxiliares no processo de introdução à programação de computadores. Padrões pedagógicos coordenam e relacionam as atividades, facilitando a conexão entre os conceitos e permitindo ampliação da rede de conceitos de forma individualizada. Os projetos, que compõe as atividades, são elaborados com base nos padrões de projeto, que refletem boas soluções a problemas recorrentes. No próximo item será apresentada a concepção geral desta proposta de roteiro.

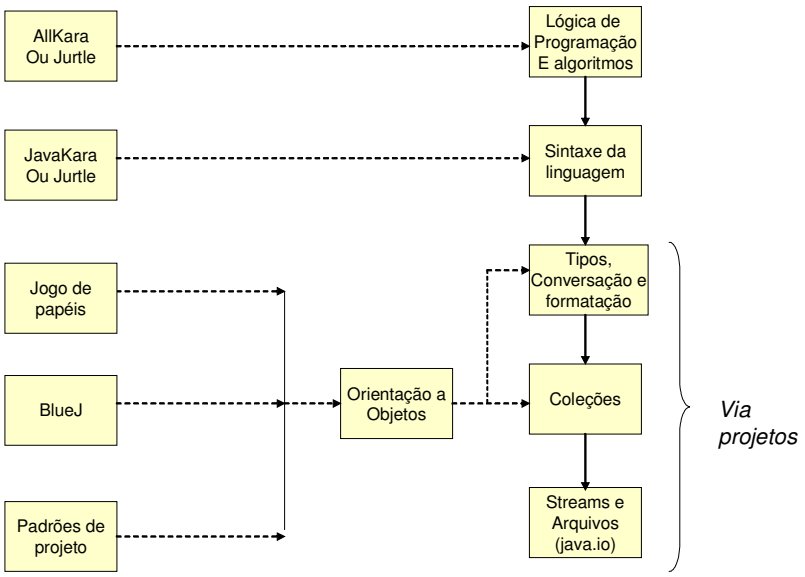

Figura 6-1 Mapa para um curso introdutório

6.2 FERRAMENTAS UTILIZADAS: MICROMUNDOS E PADRÕES

Como foi visto no Capítulo 4, os micromundos implementam, explicita ou implicitamente, a essência dos diversos padrões pedagógicos, conforme apresentado na sessão 4.2.

Por si só, o micromundo constitui uma diferente **abordagem**, que dispensa uma introdução formal para ver os objetos em funcionamento. A abordagem formal posterior à experimentação do micromundo facilita o entendimento, pois permite uma **analogia física** direta além de ser uma metáfora consistente. Com a vivência e construção pessoal do entendimento o aluno pode criar uma conexão entre os conceitos vivenciados e formais, fazendo uma **ligação do velho ao novo**. E da experimentação e da exposição de conceitos cria-se a oportunidade de fazer uma **reflexão** sobre os conceitos abordados.

Considerando os conceitos introdutórios como objetos, classes, solicitação de serviços, encapsulamento, ocultação, os micromundos LOGO Microworlds, Allkara, Alice ou Karel J. Robot permitem uma abordagem diferente, de forma bem clara e sem complicações. Embora qualquer uma delas possa cumprir com os objetivos desejados, no roteiro proposto a escolha foi AllKara, pela possibilidade adicional de utilizar autômatos finitos de uma forma intuitiva desde o início, permitindo ao aluno vivenciar, de imediato e de forma explícita , o conceito de mudança de estado, essencial para os conceitos de OO e Ciência da Computação. Para turmas com perfil diferente de Ciência da Computação, a escolha poderia ser o Jurtle.

Todos os micromundos são fáceis de serem utilizados e permitem que cada aluno realize o padrão **explore por você mesmo**. Além disso, o micromundo é um espaço aberto para descobertas, tentativas, erros e correções, permitindo que funcione como um **tubo de ensaio**.

Com estes recursos, uma introdução dentro de um mundo habitado por objetos, experimentando as idéias do paradigma de orientação por objetos pode ser bem feita.

Outro recurso sugerido é o padrão pedagógico **jogo de papéis** [BER00b, LAV03]. Este permite a implementação dos demais padrões pedagógicos citados neste trabalho. Com este o instrutor pode elaborar um conjunto de papéis (classes informais) dentro de um contexto, abordando conceitos específicos, que permitam ao aluno vivenciar papéis dos objetos e entender a dinâmica da execução dos sistemas. Em particular, não foi encontrado na literatura nenhuma referência a uma IDE pedagógica para a construção de Jogos de Papéis. A descrição dos jogos de papéis implica, até o momento, em atividades de sala de aula. As experiências de utilização dos jogos de papéis nos cursos desenvolvidos durante este trabalho

se mostraram muito produtivas e eficazes. Desta experiência ocorreu a idéia de que ele pudesse ser implementado como parte do BlueJ, sendo incorporado através de um plug-in. A definição de requisitos de interface para tal IDE é descrita no item 6.4, mais à frente.

Toda a atividade sugerida a ser desenvolvida no roteiro da Estação OO, não pressupõe conhecimento prévio. Tanto nos micromundos quanto nos jogos de papéis os conceitos são vivenciados e posteriormente formalizados. Além disso, as atividades sugeridas aos alunos e os próprios papéis devem fazer uso implícito de alguns dos *design patterns*, que norteiam o bom uso dos conceitos de orientação por objetos.

A idéia essencial de incorporar os jogos de papéis no roteiro da Estação OO é a de que, enquanto o jogo acontece, os objetos interagem entre si, e estão sendo compreendidos e manipulados pelos alunos. Situações de erros também podem ser abordadas, como solicitações de serviços não disponíveis, e excesso ou omissão de parâmetros. Durante cada jogo, os diagramas dos objetos envolvidos sempre visíveis mostram a alteração no estado dos objetos e introduzem estes conceitos pelo uso da UML. O diagrama das classes representando os papéis envolvidos também é apresentado, constituindo-se em uma forma de representar a classe antes do aluno ter contato com o código.

Após o "jogo" os conceitos envolvidos podem ser experimentados no computador usando o BlueJ. Manipulando projetos semiprontos, os alunos podem criar objetos a partir das classes e solicitar serviços aos objetos colocados na bancada de objetos. Esta atividade permite uma aproximação da solução do problema, ou seja, o aluno pode comunicar com os objetos da classe a partir da linguagem disponibilizada pelos seus métodos. Assim, a classe é conhecida pelos serviços que provê. Em um próximo passo, o código da classe é abordado, conhecendo-se a implementação, onde o código é estudado com um diferencial da abordagem tradicional, o aluno traz experiências na criação e na manipulação de objetos. Desta forma, o código já não é algo totalmente novo, mas algo que o aluno foi preparado para ver e entender.

6.3 AS ESTAÇÕES EM ESTAÇÃO OO

A Estação OO dispõe de um roteiro de viagem na forma de um mapa, como mostrado na Figura 6-2.

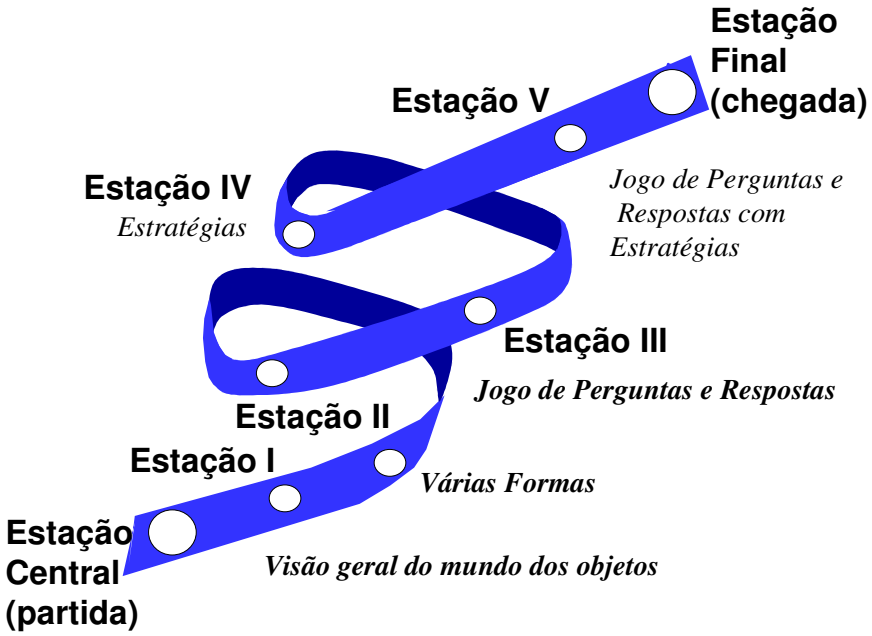

Figura 6-2 O Mapa com um "roteiro de viagem" sugerido pela Estação OO

Na Tabela 6-1 estão descritas as cinco estações do roteiro proposto pela Estação OO. Em cada estação estão descritas as atividades a serem desenvolvidas e os recursos a serem utilizados.

Tabela 6-1 Descrição de Atividades e recursos das Estações no "Roteiro de viagem" proposto.

Conteúdo	Atividades	Recursos
Estação Central: Introdução ao mundo orientado por objetos.	• Utilizar um micromundo para experimentar a manipulação de objetos e ter uma introdução à lógica de programação e algoritmos.	• Allkara ou • Jurtle ou • Karel J. Robot.
Estação I: Visão geral do mundo dos objetos.	• Cada aluno recebe uma descrição do seu papel. • O jogo é iniciado e cada aluno participa como elemento específico (objeto) como estabelecido em uma tabela de participantes no Jogo de Papéis. • Durante o jogo cada participante exibe seus dados internos e os altera de acordo com as solicitações que recebe. • As solicitações alteram o estado dos objetos e podem gerar erros. • As solicitações são registradas já na linguagem Java. E mais tarde, podem constituir o método **main** da aplicação.	• Criação de um projeto utilizando Estação OO que envolva os conceitos fundamentais de orientação por objetos pelo instrutor, definindo previamente: ○ As classes envolvidas; ○ A geração do código; ○ Criação do Jogo de Papéis. • Sugestão: projeto Acrobata e Cia no Anexo II.

	• Após o jogo é feito o relacionamento entre os elementos do jogo, ligando o conceito formal ao conceito experimentado. • Apresentação de nova descrição das classes, agora formal, através do diagrama das classes em UML. • Criação e manipulação dos objetos no computador.	
Conceitos abordados: Classe, objeto (instância), campo, método, parâmetro, tipo de retorno, solicitação de serviço, visibilidade dos elementos da classe, herança, polimorfismo e interface.		
Padrões de projeto utilizados: Polymorphism, Low Coupling/High Cohesion, Interface, Delegation.		
Estação II: Várias formas.	• O jogo de papéis acontece como descrito na atividade anterior. Cada aluno recebe uma descrição do seu papel. • Extensão do projeto para incluir outras formas geométricas como Triangulo e Quadrado. O que permite utilizar os padrões Interface (descrito na sessão 4.4.6), Abstract Superclass (sessão 4.4.7) e Polymorphism (sessão 4.4.4).	• Criação de um projeto utilizando Estação OO que envolva os conceitos básicos de orientação por objetos pelo instrutor, definindo previamente: o As classes envolvidas; o A geração do código; o Criação do Jogo de Papéis. • Sugestão: Projeto Formas Geométricas no Anexo II.
Conceitos reabordados: Classe, objeto (instância), campo, método, parâmetro, tipo de retorno, solicitação de serviço, visibilidade dos elementos da classe, herança e polimorfismo. **Conceitos introduzidos**: Composição, classe abstrata, método abstrato, versatilidade, expansão do projeto.		
Padrões de projeto reutilizados: Polymorphism, Low Coupling/High Cohesion e Delegation. **Padrões de projeto introduzidos**: Interface e Abstract Superclass.		
Estação III: Jogo de Perguntas e Respostas.	• O jogo de papéis acontece como descrito na atividade anterior. Cada aluno recebe uma descrição do seu papel. • Introdução de coleções de objetos e iteração sobre ela.	• Criação de um projeto utilizando Estação OO que envolva os conceitos básicos de orientação por objetos pelo instrutor, definindo previamente: o As classes envolvidas; o A geração do código; o Criação do Jogo de Papéis. • Sugestão: Perguntas e Respostas no anexo II.

<table>
<tr><td colspan="3">Conceitos reabordados: Classe, objeto (instância), campo, método, parâmetro, tipo de retorno, solicitação de serviço, visibilidade dos elementos da classe, herança, polimorfismo e composição.
Conceitos introduzidos: Classes da biblioteca Java como ArrayList e StringTokenizer.</td></tr>
<tr><td colspan="3">Padrões de projeto reutilizados: Polymorphism, Low Coupling/High Cohesion, Delegation e Composite.</td></tr>
<tr><td>Estação IV: Estratégias.</td><td>• Construção de um analisador de palavras de um texto com diferentes estratégias, decoração e composição.</td><td>• Desenvolvimento com os alunos do projeto proposto por Bergin. Esta atividade foi apresentada no Capítulo 4 na sessão 4.5.2.
• O projeto é desenvolvido no BlueJ, com o teste de cada unidade concluída.
o Estratégias;
o Decorador;
o Composição.</td></tr>
<tr><td colspan="3">Conceitos reabordados: Classe, objeto (instância), campo, método, parâmetro, tipo de retorno, solicitação de serviço, visibilidade dos elementos da classe, herança, polimorfismo e composição.
Conceitos introduzidos: Classes da biblioteca Java como StringTokenizer.</td></tr>
<tr><td colspan="3">Padrões de projeto reutilizados: Polymorphism, Low Coupling/High Cohesion, Delegation e Composite.
Padrões de projeto introduzidos: Strategy, Decorator, Composite.</td></tr>
<tr><td>Estação V: Jogo de Perguntas e Respostas com Estratégias.</td><td>• O aluno deve propor estratégias para que as jogadas dos participantes. As estratégias podem ser: responder, pular, repassar para um amigo.</td><td>• Implementar as novas classes que incorporem as estratégias ao projeto do jogo anteriormente desenvolvido.</td></tr>
<tr><td colspan="3">Conceitos reabordados: Os conceitos explorados nas Estações III e IV são revistos e mesclados.
Conceitos introduzidos: Extensão do projeto.</td></tr>
<tr><td colspan="3">Padrões de projeto reutilizados: Strategy.</td></tr>
</table>

Na Estação I, *Introdução ao mundo orientado por objetos*, o micromundo é apresentado aos alunos e os desafios são lançados. Através da busca da solução, socialização das idéias e formalização paralela das idéias os conceitos fundamentais são abordados e a sintaxe da linguagem Java introduzida.

Na Estação II, *Visão geral do mundo dos objetos*, os conceitos são vivenciados através dos Jogos de Papéis. Extensões podem ser propostas aos alunos como criar novas formas geométricas e compor novas formas geométricas utilizando as formas existentes. Tais atividades criam a possibilidade de utilizar padrões de projeto como Interface, Abstract Superclass, Composite e Polymorphism

Na Estação III, *Jogo de Perguntas e Respostas,* os alunos, desempenhando papéis específicos, criam uma base de perguntas e respostas que serve como fonte para as perguntas do jogo. Alunos representando grupos armazenam referências aos seus participantes, coleções de outros alunos (objetos). Em cada jogada uma pergunta é sorteada pelo instrutor que deve ser respondida por uma determinada equipe. O aluno-líder do grupo indica o participante do grupo que responderá a pergunta. O respondedor indicado pode responder a pergunta ou, dependendo do tipo de aluno que seja, solicitar ajuda a um colega de equipe que pode selecionar outro colega ou não e assim por diante. Se a equipe acerta a resposta ganha um ponto, caso contrário o ponto fica para a outra equipe. Este projeto faz a revisão dos conceitos já introduzidos e acrescenta a manipulação de coleções.

Na Estação IV, *Estratégias*, uma série de atividades introduz os padrões, citados na Tabela 6-1, cujo objetivo é a manipulação de um texto e verificação de ocorrência de elementos, que podem ser palavras que começam com um determinado caractere, palavras com tamanho maior que um determinado tamanho n e palíndromos.

Na Estação V se propõe uma alteração no jogo abordado anteriormente, utilizando diferentes estratégias para estabelecer as jogadas, como a possibilidade de pular um número pré-determinado de vezes.

6.4 ANÁLISE DE REQUISITOS DE UMA IDE PARA ESTAÇÃO OO

Na Estação OO o modelo de introdução à programação é centrado em objetos, partindo do nível conceitual (externo) em direção à implementação (interno). Por isso, a primeira visão do projeto é seu diagrama de classes.

A Estação OO deve trazer embutidos vários projetos, entre eles as atividades sugeridas na sessão 4.5, que facilitam a introdução aos *design patterns* e os projetos utilizados nos estudos de caso citados na sessão 5.1 e detalhados no Anexo II.

A Figura 6-3 mostra os requisitos de interface (preliminar) para a criação de uma IDE para a Estação OO, colocando como exemplo o projeto AcrobataECia no momento em que um menu "pop-up" aparece ao se clicar com o botão direito do mouse sobre a classe Acrobata.

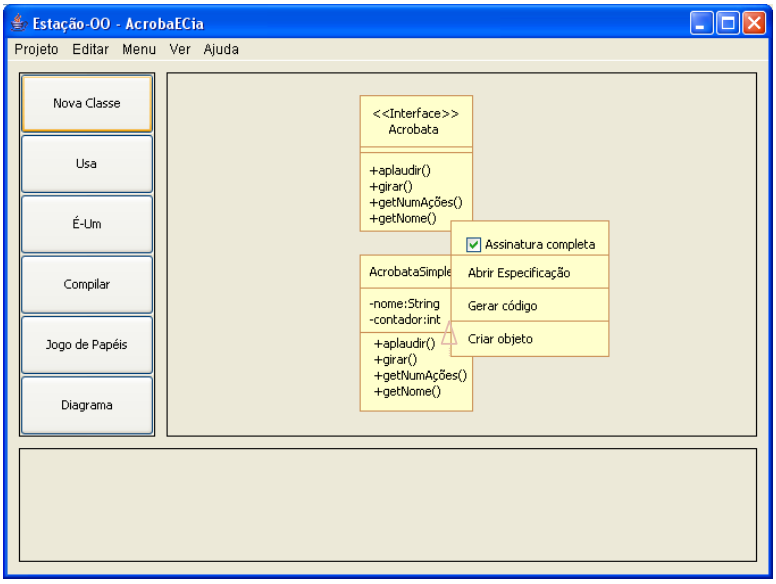

Figura 6-3 Uma proposta de interface para a Estação OO

O usuário pode criar um novo projeto ou abrir algum existente. Ao criar um novo projeto as classes deverão ser incluídas e seus relacionamentos (É-Um e Usa) devem ser estabelecidos.

As Figuras 6-4 a 6-7 mostram as informações que devem ser fornecidas para a criação da classe.

A IDE didática proposta é uma extensão do BlueJ, instalado como um *plug-in* ao BlueJ [KOL03]. Sendo assim mantém as características do BlueJ e incorpora elementos que facilitam a montagem e execução do padrão pedagógico Jogo de Papéis, permite uma forma diferente de criar as classes através da definição de seus elementos como pode ser visto nas Figuras 6-3 a 6-7 e incrementa a visualização das classes, permitindo que seus elementos sejam exibidos, como pode ser observado na Figura 6-3.

Quanto ao Jogo de Papéis, a IDE pedagógica da Estação OO deve permitir:

- Geração dos papéis a partir da definição das classes;
- Criação e manutenção da tabela de participantes;
- Simulação do jogo no computador na própria bancada de objetos, porém, com todos os objetos criados exibindo seu estado interior durante as atividades.

O BlueJ monta o diagrama de classes a partir do código fonte. Em Estação OO as classes podem ser definidas através da definição de seus elementos constituintes e com esta definição o código em Java é gerado. Com este tipo de criação da classe é atribuído maior importância aos elementos da classe e não à criação do código fonte na linguagem Java.

As sessões seguintes descrevem atividades que podem ser desenvolvidas em Estação OO.

Criação de Classes

Ao iniciar a criação de uma classe suas informações gerais devem ser fornecidas como pode ser visto na Figura 6-4.

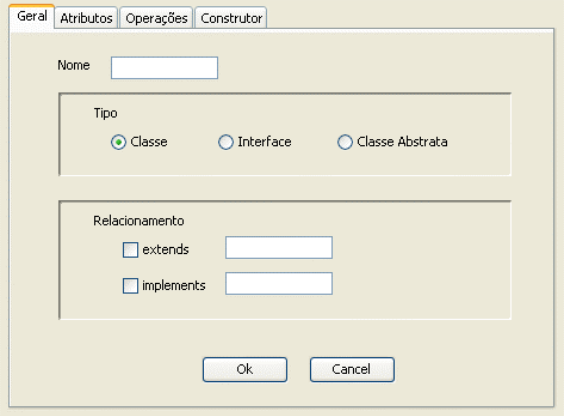

Figura 6-4 - Aba Geral

O usuário define os atributos da classe: nome, tipo, visibilidade, valor inicial e os comentários para o Jogo de Papéis. A tela de entrada para estes dados está na Figura 6-5.

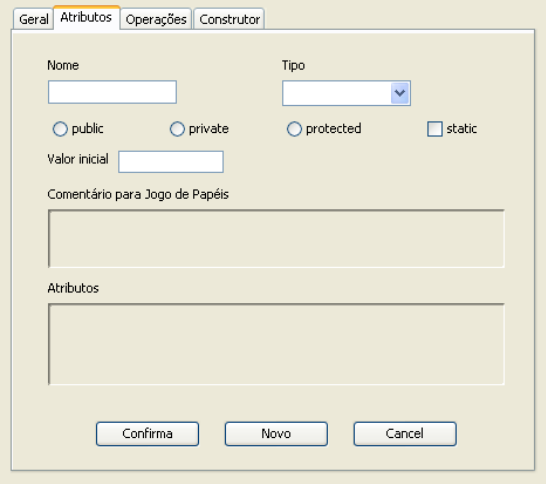

Figura 6-5 - Aba Atributos

O usuário também define as operações que os objetos da classe podem executar. Fornecem nome, tipo de retorno, visibilidade, parâmetros e comentários para o jogo de papéis através de uma tela como a mostrada na Figura 6-6.

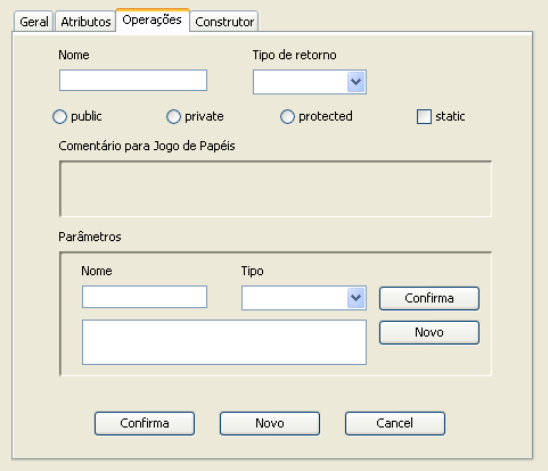

Figura 6-6 - Aba Operações

O usuário também define os construtores como mostrado na Figura 6-7.

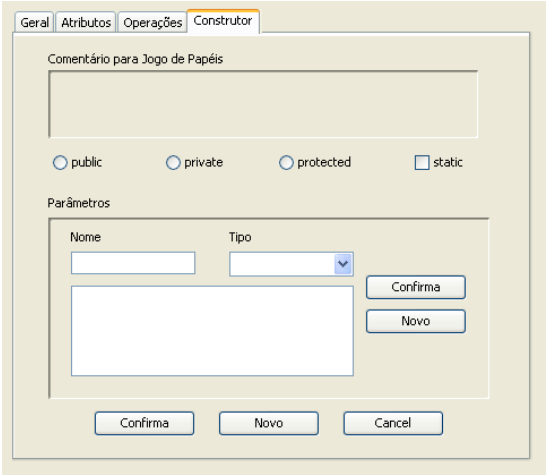

Figura 6-7 Aba Construtor

Jogo de Papéis

Ao gerar o Jogo de Papéis um arquivo texto com a descrição informal da classe é gerado para cada classe. Em seguida o usuário pode montar a tabela que relaciona alunos e papéis e pode elaborar um *script* com a seqüência de solicitações a ser feita.

Este script gerado pode ser impresso e também executado no próprio computador. Quando o usuário simula no computador. Os objetos criados passam a compor a bancada de objetos com seus diagrama de objetos a vista.

Ao encontrar uma solicitação errada o script é interrompido, uma mensagem informando o erro é exibida e deve ser ajustado para continuar.

6.5 CONCLUSÃO

Assim se apresentou Estação OO, um roteiro básico para uma viagem com escalas em várias estações, acompanhado de uma proposta de uma interface de uma IDE didática que facilita as atividades em cada estação e a elaboração de novos roteiros.

A idéia principal é proporcionar aos alunos iniciantes um caminho facilitado por recursos (ferramentas e projetos) que contemplem as idéias fundamentais da tecnologia de objetos desde o primeiro contato apoiadas no uso de padrões. As forças de diferentes recursos baseados no uso do computador são integradas. De um lado, os micromundos e padrões pedagógicos facilitam a condução de atividades, a programação das disciplinas e a elaboração dos próprios cursos. De outro, recursos da área de computação como UML e padrões de projeto (*design patterns*) são utilizados, a UML como linguagem para descrever as soluções e os padrões de projeto como um direcionamento seguro.

Capítulo 7

7 Conclusão

7.1 REVISÃO DO TRABALHO

Este trabalho teve como objetivos principais estudar e analisar micromundos e ambientes integrados de desenvolvimento (IDEs), tanto didáticos quanto pedagógicos; verificar na prática de sala de aula como estes ambientes podem contribuir no processo de ensino/aprendizagem; investigar os padrões pedagógicos, de projeto e de codificação para iniciantes disponíveis na literatura e propor um roteiro de como haver uma apropriação das soluções existentes de forma complementar e suplementar. Foi proposta, a partir de uma metáfora com uma viagem, um roteiro ou mapa capaz de levar aprendizes iniciantes em orientação a objetos à aprendizagem dos conceitos em um curso introdutório.

As dificuldades com o meio de aprendizagem e com as ferramentas existentes foram apontadas, depois de estudadas em turmas fechadas. O foco principal do estudo foi em disciplinas introdutórias de programação, com enfoque em objetos e na linguagem Java, considerando que a linguagem tem um papel fundamental no desenvolvimento dos processos mentais de cada pessoa.

Uma análise da programação em diferentes paradigmas foi feita no Capítulo 2. E pode-se verificar que não existe consenso quanto à melhor abordagem para o ensino de OO. Foi adotada a solução proposta pela força tarefa que desenvolveu o currículo de Ciência da Computação proposto pela ACM [ACM01] no qual se afirma que a experimentação na área é que deve guiar a escolha de caminhos.

No Capítulo 3 foi apresentada uma descrição das ferramentas estudadas e/ou analisadas, suas limitações e uma comparação entre elas foi estabelecida.

A adoção de padrões foi uma busca constante deste trabalho, conforme mostrado no Capítulo 4. Os padrões são contribuições de profissionais mais experientes na comunidade de computação e na educação e representam, de modo geral, boas idéias que podem ser compartilhadas de uma forma organizada. Os padrões pedagógicos compreendem sugestões para: elaboração e utilização de atividades em sala de aula, formulação e desenvolvimento de conteúdos em disciplinas, elaboração de currículos de cursos, tratamento da heterogeneidade

dos alunos na turma, entre outras. Os padrões de projeto (*design patterns*) representam o emprego das idéias genuínas do paradigma de orientação por objetos, são luzes que precisam começar a brilhar, desde cursos introdutórios até os avançados.

A comunidade docente da área de computação se preocupa com a utilização dos padrões em cursos introdutórios e se esforça para criar atividades que facilitem esta tarefa. Como foi mostrado neste trabalho, existem vários esforços na incorporação destas grandes idéias para uso com iniciantes. Muitos destes esforços foram mostrados no Capítulo 4.

O Capítulo 5 descreveu estudos de casos que foram feitos para a análise e avaliação de algumas das ferramentas encontradas na literatura na tentativa de descobrir a melhor combinação entre elas. Estes estudos de caso foram feitos na forma de cursos aplicados em turmas fechadas. Os cursos foram aplicados em seqüência e cada um deles projetado a partir das conclusões tiradas da aplicação do curso anterior. Os cursos aplicados mostram uma evolução natural na busca de ferramentas para facilitar o ensino/aprendizagem do paradigma de orientação por objetos. À medida que estes foram aplicados, as falhas eram percebidas e uma nova ferramenta era incorporada para preencher a lacuna. Os cursos foram gradativamente incorporando as idéias que foram sendo consideradas como mais promissoras.

Após estes estudos, as idéias da Estação OO puderam ser propostas. No Capítulo 6, um mapa para cursos introdutórios foi proposto. Neste mapa (mostrado nas Figuras 6-1 e 6-2), percebe-se que os micromundos participam do processo introdutório e os projetos, que utilizam IDEs didáticas principalmente, dão continuidade ao processo com o apoio dos padrões pedagógicos, de projeto e de codificação para iniciantes.

O uso de projetos em cursos da área de computação são ferramentas poderosas. Eles têm o poder de motivar e facilitar a interdisciplinaridade. Os projetos podem eventualmente ultrapassar os limites de disciplinas introdutórias e abranger várias disciplinas de nível avançado. Desta forma, o conhecimento flui do problema em um contexto para a solução e tem uma probabilidade muito maior de ser aproveitado e compreendido do que se exposto de forma tradicional com exposição de conceitos e prática sobre conceitos descontextualizados. Muitas vezes um bom desafio, muitas vezes simples, é uma verdadeira obra de arte e pode fazer a diferença na qualidade de um curso.

7.2 CONTRIBUIÇÕES DESTE TRABALHO

Apesar do protótipo de IDE sugerido não ter ficado disponível para utilização neste trabalho, as idéias subjacentes foram testadas em sala de aula. O mapa proposto como roteiro para Estação OO foi utilizado com sucesso. As atividades sugeridas nos padrões pedagógicos, principalmente com o uso do Jogo de Papéis, integradas com artefatos da UML e utilizando os padrões de projeto, mostraram ser de grande auxílio. O enfoque na descrição das classes como fábricas de objetos e nos objetos como elementos originários das classes e provedores de serviço facilita a abordagem. O "adiamento" na apresentação do código permitiu a familiarização e sedimentação dos conceitos de orientação por objetos. Portanto, a maior contribuição deste trabalho está na combinação de recursos propostos.

7.3 GENERALIZAÇÃO DAS PROPOSTAS DESTE TRABALHO

Apesar do sucesso obtido com atividades incluídas na Estação OO no curso descrito no Capítulo 5, Tabela 5-4, o enfoque proposto neste trabalho precisaria ser replicado em outras instâncias, em turmas de diferentes perfis e com uma sistemática quantitativa e/ou qualitativa, para ser validado e generalizado.

7.4 SUGESTÕES DE CONTINUIDADE

A continuidade deste trabalho poderá ser feita em vários projetos. Algumas sugestões seguem abaixo:

- Implementação do protótipo esboçado no trabalho e sua incorporação à Estação OO proposta
- Replicação do estudo em novas instâncias de forma sistemática
- Estudo do desenvolvimento dos alunos em disciplinas avançadas após a introdução seguindo o mapa proposto na Estação OO
- Desenvolvimento de projetos próximos à realidade dos alunos que possam ser utilizados de forma interdisciplinar na Ciência da Computação.

REFERÊNCIAS

[ACM01] The Joint Task Force on Computing Curricula, IEEE Computer Society and Association for Computing Machinery (2001): *Computing Curricula 2001: Computer Science*. ACM Journal of Educational Resources in Computing. Disponível on-line em http://www.computer.org/education/cc2001/final.

[ALL01] Allen, E.; Cartwright, R.; Stoler, B. *Dr. Java: A lightweight* pedagogic environment for Java. Rice University, Houston.

[AST98] Astrachan, Owen; Berry, Geoffrey; Cox Landon; Mitchener, Garret. *Design Patterns: An Essencial Component of CS Curricula.* . Proceedings of ACM, SIGCSE 98, Atlanta, USA.

[BAR93] Baranauskas, Maria Cecília Calani; *Procedimento, Função, Objeto ou Lógica?* Linguagens *de Programação vistas pelos seus Paradigmas.* Em J. A. Valente, (org.) *Computadores e Conhecimento, Repensando a Educação.* Campinas: NIED - UNICAMP pp. 54-69.

[BAR02] Baranauskas, Maria C. C.; Rocha, Heloísa V., Martins, Maria Cecília; d´Abreu, João V.V.; *Uma Taxonomia para Ambientes de Aprendizado Baseados no Computador*. Em J. A. Valente, (org.) O computador na Sociedade do Conhecimento. Campinas: NIED – UNICAMP, pp. 49-87.

[BAR03] Barnes, D. J. and Kolling, M. *Objects First with Java. A Practical Introduction using BlueJ*. Person Education, 2003.

[BRA87] Braathen, Per Christian. *A Case Study Prior Knowledge, Learning approach and conceptual change in Introductory College Chemistry Tutorial Program.* Madison, University of Wisconsin, 1987.

[BEL95] Bell, John T.; Folger, S. *The* Investigation *and Application of Virtual Reality as an Education Tool*. Proceedings of the American Society Engineering Edcuation 1995. Annual Conference, June, 195.

[BER96] Bergin, J. Java as a better C++. ACM SIGPLAN Notices, 31(11):21-27, 1996.

[BER98] Bergin, J.; Naps, T. L.; Bland, C. G.; Hartley, S. J.; Holliday, M. A.; Lawhead, P. B.; Lewis, L.; McNally, M. F.; Nevison, C. H.; Cheng; Pothering, G. J.; Terasvirta, T. Java resources for computer science instruction. Working Group reports of the 3rd annual SIGCSE/SIFCUE ITiCSE conference on Integrating technology into computer science education, pages 14-34. ACM Press, 1998.

[BER00a] Bergin, J. *An Object*-Oriented *Bedtime Story*, http://csis.pace.edu/~bergin/Java/OOStory.html em julho/2004.

[BER00b] Bergin, J. Objects On The First Day, http://csis.pace.edu/~bergin/Java/RolePlay.html em julho /2004.

[BER00c] Bergin, J. *Why Procedural is the Wrong First Paradigm if OOP is the Gol*, http://csis.pace.edu/~bergin/papers/Whynotproceduralfirst.html em julho /2004.

[BER01a] Bergin, Joseph; Eckstein, Jutta; Manns, Mary Lynn; Wallingford. *Patterns for Gaining Different Perspectives*. PloP, 2001.

[BER01c] Bergin, Joseph; *Elementary Patterns Strategy, Decorator, and Composite. The Java IO Classes*. Na internet http://csis.pace.edu/~bergin/patterns/strategydecorator.html em 09/2004.

[BER01b] Bergin, Joseph; *Coding at he Lowest Level. Coding Patterns for Java Beginners*. Na internet, em 09/2004, http://csis.pace.edu/~bergin/patterns/ codingpatterns.html.

[BER02] Bergin, Joseph; *Some Pedagogical Patterns*. Na internet, http://csis.pace.edu/%7Ebergin/patterns/fewpedpats.html#unp em 01/09/2004

[BER02d] Bergin, J., Stehlik, M., Roberts, J. and Pattis, R. *Karel J. Robot: A gentle Introduction to the Art of Object-Oriented Programming in Java*. 2002. Na Internet: http://csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html em 19/03/2003.

[BER03] Bergin, J. *A Simple Calculator for Novice Learning*. Na internet em 10/06/2004, http://csis.pace.edu/~bergin/polycalc/index.html.

[BER04] *Teaching Notes and a bit of philosophy*. Disponível na Internet no endereço http://csis.pace.edu/~bergin/KarelJava2ed/teachingnotes.html em 01/08/2004.

[BEN04] Bennedsen, J. and Caspersen, M. E., *Programming in Context – A Model-First Approach to CS1*. Proceedings of ACM, SIGCSE´04, march 3-7, 2004, Norfolk, Virginia – USA.

[BLU05a] Material para AP (Advanced Placement) Computer Science disponível no endereço http://www.bluej.org/help/ap.html em 04/04/2005.

[BLU05b] Classes disponíveis para uso no BlueJ no endereço http://www.bluej.org/resources/classes.html em 04/04/2005.

[BRU04] Bruce, K. B. *Controversy on How to Teach CS 1*. A Discussion on the SIGCSE-members Mailing List. Proceedings from the SIGCSE Bulletin – Volume 36, Number 4, 2004 December.

[BUC99] Buck, Duane. *Karel*. Westerville, Ohio, 1999. Disponível na Internet: http://math.otterbein.edu/Class/CSC100/Karel/web/Karel/ Karel.htm em (12/03/2003).

[BUC00] Buck, Duane; Stucki, David J.; *Design Early Considered Harmful: Graduated Exposure to Complexity and Structure Based on Levels of Cognitive*. Development. Proceedings of ACM, SIGCSE 2000 3/00, pp 75-79.

[BUC01] Buck, D. and Stucki, D. JkarelRobot: *A Case Study in Supporting* Levels *of Cognitive* Development *in* the *Computer Science Curriculum*. Proceedings of the Thirty-Second SIGCSE. Technical Symposium (February 2001), ACM Press, 16-20.

[CON99] Conway, Matthew J. *Alice: Easy-to-Learn 3D Scripting for Novices*. A dissertation presented to the Faculty of the School of Engineering and Applied Science at the University of Virginia for the Degree Doctor of Philosophy (Computer Science). 1997.

[COO03] Cooper, S.; Dann, W.; Paush, R. *Teaching Objects-first in Introductory Computer Science*". In proceedings of the 34th SIGCSE symposium (Reno, Nevada, February 2003.

[DEC03] Decker, Adrienne. *A tale of Two Paradigms*. Procedente do Consortium for Computing Science in Colleges. University at Buffalo, Sate University of New York, 2003.

[ECK01] Eckel, Bruce. *Thinking in Java*. Disponível na internet em Bruce Eckel's Free Electronic Books - http://www.janiry.com/bruce-eckel/ em 01/02/2005.

[ECK03] Eckel, Bruce. *Thinking in Patterns – Problem-Solving Techniques using Java*. Disponível na internet em Bruce Eckel's Free Electronic Books - http://64.78.49.204/ em 01/02/2005.

[ENT83] Entwistle, N., & Ramsden, P. *Understanding Student Learning*. New York: Nichols Publishing Company. 1983.

[FEL88a] Felder, R. M; Silverman, L. K. *Learning and Teaching Styles in Engineering Education.* Na internet http://www.ncsu.edu/felder-public/Papers/LS-1988.pdf em 06/2004.

[FEL88b] Felder, R.; *How students learn: Adapting teaching styles to learning styles*. Proceedings, Frontiers in Education conference, ASEE/IEEE, Santa Barbara, CA, p. 489.

[FEL93] Felder, R. M. *Reaching the Second Tier – Learning and Teaching Styles in College Science Education*. Journal of College Science Teaching, pg 286-290, March/april, 993.

[FER03] Ferguson, E. *Object-Oriented Concept Mapping Using UML Class Diagram*. Consortium for Computing in Small Colleges (CCSC), 2003.

[FIL01] Filho, Wilson de Pádua P. *Engenharia de Software – Fundamentos, Métodos e Padrões*. LTC, Rio de Janeiro – RJ.

[FIN03] Finley, Thomas; Akingbade, Aonike; Jackson, Diana; Patel, Pretesh; Rodger, Susan H. JAWAA: *Easy Web-Based Animation from CS 0 to Advanced CS Courses*. Proceedings of ACM, SIGCSE ´03, February 19-23, Reno, Nevada, USA.

[HAR92] Harb, J. N.; Terry, R. E. *A Look at Performance Tools through the Use of the Kolb Learning Cicle*. ASEE Annual Conference Proceedings, 1992

[HOR96] Howard, R. A; Carver, C.; Lane, W. D. *Felder´s learning styles, Bloom´s taxonpmy, and the Kolb learning cycle: tying it all together in the CS2 course*. Proceedings of ACM SIGCSE Bulletin. Vol. 28 – pg 227 a 331. Mar 96.

[GOF95] Gama, E.; Helm, R.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*.Addison-Wesley, 1995.

[GRA98] Grand, Mark. Patterns *in Java. A catalog of Reusable Design Patterns Illustrated with UML.*, volume 1, 1ª edição. John Wiley & Sons. 1998.

[GRA02] Grand, Mark. Patterns in Java. *A catalog of Reusable Design Patterns Illustrated with UML.* volume 1, 2ª edição. John Wiley & Sons. 2002.

[GUI00] Guimarães, Ângelo de Moura; Matos, Helton Fábio (Ed. e Adapt.); *Introdução ao PascaL. Material das disciplinas DCC001 e DCC003*. Departamento de Ciência da Computação – UFMG. Julho de 2002.

[GUZ02] Guzdial, Mark; Soloway, Elliot. *Teaching the Nintendo Generation to Program.* Communications of the ACM, April 2002, Vol 45 nº 4.

[HAM04] Hamer, John. *An Approach to Teaching Design Patterns using Musical Composition*. Proceedings of the ACM, ITiCSE'04, p156-160, june.

[HEN04] Henriksen, Poul; Koolling, Michael. greenfoot: *Combining Object Visualisation with Interaction.* Proceedings of OOPSLA '04. Oct. 24-28, 2004, Vancouver, British Columbia, Canada. ACM 1-58113-833-4/4/0010.

[IBM04] Relatório técnico do projeto Eclipse disponível no site http://www.eclipse.org/articles/index.html em 10/05/2004.

[KOL95] Kolling, Michael; Koch Bett; Rosenberg, John; *Requirements for a First Year Object-Oriented Teaching Language.* Proceedings of the ACM, SIGCS´95 3/95, pp 173-177.

[KOL99] Kolling, Michael; *The design of an object-oriented environment and language for* teaching. Dissertation for the degree of Doctor of Philosophy at the Basser Department of Computer Science, University of Sydney, 1999.

[KOL01] Kolling, M.; Rosemberg, J.*Guidelines for teaching object orientation with Java*. Proceedings of the 6th annual conference on innovation and technology in computer science education, 33-36.

[KOL03] Kolling, Michael; Quig, Bruce, Patterson, Andrew; Rosenberg, John. *The BlueJ system and its pedagogy*. Na internet em (www.bluej.org/papers/2003-12-CSEd-bluej.pdf) em fevereiro/2005.

[KOL83] Kolb, D. *A. Experiential Learning. Experiences as the source of Learning and Development*. Prentice Hall, 1983.

[LCS00] LCSI, MicroWorlds 2.0. *User´s Guide*. LCSI, november 2000. Disponível na internet: http://www.lcsi.ca/pdf/microworlds20/ windows/userguide.pdf

[LAR98] Larman, Craig. *Applying UML and Patterns*. Upper Saddle River, N.J.: Prentice Hall PTR, 1998.

[LAV03] Lavine, David; *Role Playing in an Object-Oriented World*. Na internet http://web.sbu.edu/cs/dlevine/RolePlay/roleplay.html em 09/2004.

[LEW04] Lewis, L. T; Rosson, M. B.; Pérez-Quinones, M. *What Do The Experts Say? Teaching Introductory Design from an Expert´s Perspective*. Proceedings of the ACM, SIGCS´04, March 3-7, 2004, Norfolk, Virginia, USA.

[LOZ05] Lozano, F.; Galvão, L. *Java Magazine*, nº 23, vol III, p. 17, 2005.

[MCD00] McDowell, Charlie; *Java meets Karel Robot*. Artigo disponível na Internet: http://www.cse.ucsc.edu/~charlie/jarel em (11/03/2003).

[MIL02] Milne I. and Rowe G. *Difficulties in Learning and Teaching Programming - Views of Students and Tutors*. Education and Information Technologies (2002, Vol. 7, No. 1, pp. 55-66).

[MOR00] MORIN, Edgar. *Os Sete Saberes Necessários à Educação do Futuro*. 2ª ed. São Paulo: Cortez; Brasília, DF: UNESCO, 2000. 118 pág.

[MUR03] Murray, A. K.; Heines, J. M.; Kolling, M.; Moore T.; Wagner, P. J.; Schaller, N. C.; Trono, John A. *Experiences with IDEs and Java teaching: what works and what doesn't.* ACM SIGCSE Bulletin, Proceedings of the 8th annual conference on Innovation and technology in computer science education, Volume 35 Issue 3. June 2003.

[NET05] Documentação disponível no site http://www.netbeans.org/kb/41/index.html em 10/12/2004.

[NYG86] Nygaard, Kristen. *Basics Concepts in Object Oriented Programming.* SIGPLAN Notice. Vol. 21, Nº 1, October 1986, pp. 128-132.

[OTH04] Manual do usuário disponível na internet no endereço http://www.otherwise.com/Jurtle.html em 10/12/2004.

[RIC03] Richards, B. *Experiences Incorporating Java into the Introductory Sequence*. Consortium for Computing Sciences in Colleges. 2003.

[PAT81] Pattis, Richard E. Karel the Robot: *A Gentle Introduction to the Art of Programming, with Pascal.* John Wiley & Sons, Inc., 1981.

[PAP85] Papert, Seymour. *LOGO: Computadores e Educação*. São Paulo: Ed Brasiliense. 1985.

[PAP94] Papert, Seymour. *A Máquina das Crianças*. ArtMed – 1994.

[PAP96] Papert, Seymour. *A Família em Rede (Ultrapassando a barreira digital entre gerações).* Relógio D`Água Editores. Novembro, 1996.

[PAU00] Paula, Valéria de C., Klogoo: *Um Micromundo para Ensino-Aprendizagem de Programação Orientada por Objetos*. Dissertação de Mestrado, DCC/UFMG, MG, 2000.

[ROC95] Rocha, Heloisa Vieira. *Representações Computacionais Auxiliares no Entendimento de Conceitos de Programação*. Em J. A. Valente, (org.) Computadores e Conhecimento, Repensando a Educação. Campinas: NIED – UNICAMP, 1995.

[REI01] Reichert, R.; Nievergelt, J.; Hatmann, W. *Programming in schools – why, and how?* Inc. C. Pellegrini, A. Jacquesson (Hrsg.): Enseigner I´informatique, pp 143-152. Georg Editeur Verlag, 2001.

[REI04] Reis, Charles; Cartwright, Robert. *Taming a Professional IDE for the Classroom.* ACM SIGCSE Bulletin, Proceedings of the 35th SIGCSE technical symposium on Computer science education, Volume 36 Issue 1. March 2004.

[REP93] Repenning, Alexander. AgentSheets: *A Tool for Building Domain-Oriented Dynamic, Visual Environments.* Thesis submitted to the Faculty of Colorado, 1993.

[REP96] Repenning, Alexander; Eden, Hal; Eisengerg, Mike; Fischer, Gerhard. *Making Learning a Part of Life.* Communications of the ACM, april 1996, Vol. 39 Nº 4.

[RUM94] Rumbaugh, James et al. *Modelagem e projetos baseados em objetos*. São Paulo: Campus, 1994.

[RUS98] Russel, Ian & Hunter, Jim. Course CS1001, University of Aberdeen, Scltland, UK, 1998 e CS_Pas Software 1996 e 1997. Dsponível na Internet: http://www.abdn.ac.uk/~csc111/cs_pas/notes/lectures/ch1.htm.

[SLO95] Slogo95. Núcleo de Informática Aplicada à Educação (NIED), Universidade Estadual de Campinas (UNICAMP), São Paulo. Disponível na Internet: http://www.nied.unicamp.br. (01/12/2002)

[SOL96] Soloway, Elliot; Bielaczyc, Kate. *Interactive Learning Environments: Where They´ve Come From & Where They´ve Going.* Proceeding CHI´96, ACM, April 1996, pp 13-18.

[TAB04] Tabrizi, M. H. N.; Collins, C.; Ozan, E.; Li, K. *Implementation of Object-Orientation Using UML in Entry Level Software Development Courses*. Proceedings of the 5th conference on Information technology education. October 2004.

[TEL89] Telo, Ernest R. *Object-Oriented Programming for Artificial Intelligence: A Guide to Tools and System Design.* Addison-Wesley Publishing Company, Inc., 1989.

[TYM98] Tyma, Paul. *Why are we using Java again?* Communications of the ACM, 41(6):38-42, 1998.

[VAL93a] Valente, J. A. *Diferentes usos do Computador na Educação. Em J. A. Valente, (org) Computadores e Conhecimento, Repensando a Educação*. Campinas: NIED – UNICAMP, pp 1-28.

[VAL93b] Valente, J. A. *Por que o Computador na Educação*? Por José Armando Valente. Em J. A. Valente, (org.) Computadores e Conhecimento, Repensando a Educação. Campinas: NIED - UNICAMP, pp 29-54

[VAL95] Valente, J. A. *LOGO as Window into the Mind*. LOGO Update, vol. 4, Nº 1, pp 1-4.

[VAL96] Valente, J. A. *O Professor no Ambiente LOGO*: Formação e Atuação. Campinas, SP, NIED. 1996.

[VAL02a] Valente, J. A. *Mudanças na Sociedade, Mudanças na Educação*: O Fazer e o Compreender. Em J. A. Valente, (org.) O computador na Sociedade do Conhecimento. Campinas: NIED – UNICAMP, pp 29-48.

[VAL02b] Valente, J. A. *Análise dos Diferentes Tipos de Software Usados na Educação*. Em J. A. Valente, (org.) O computador na Sociedade do Conhecimento.. Campinas: NIED – UNICAMP, pp 89-129.

[VAL02c] Valente, J. *A. Informática na Educação no Brasil: Análise e Contextualização Histórica.* Em J. A. Valente, (org.) O computador na Sociedade do Conhecimento. Campinas: NIED – UNICAMP, pp 1-27.

[VAR04] Varejão, Flávio. *Linguagens de Programação. Java, C++ e outras*. CAMPUS ,2004.

[WAL96] Wallingford, E. *Toward a First Course Based on Object-Oriented patterns*. Proceedings from SIGCSE´96. Philadelphia, PA USA. 1996.

[WAL98] Wallingford, E. *Using Patterns in the Classroom*. http://www.cs.uni.edu/~wallingf/patterns/elementary/using_patterns_in_class.html, em 10/08/2004.

[WIC00] Wick, R. Michael. Kaleidoscope: *Using Design Patterns In CS1*. ACM Proceedings of the SIGCSE 2001. p 258-262.

[WIN96] Winslow, Leon E.; *Programming Pedagogy – A Psychological Overview*. ACM SIGCSE, Bulletin, Vol. 28, Nº 3, September 1996, pp17-25.

[WIR90] Wirfs-Brock, Rebecca; WILKERSON, Brian; WIENER, Lauren. Designing object-oriented software. Englewood Cliffs : Prentice-Hall , 1990.

[WIK05] Enciclopedia Wikipedia no endereço http://en.wikipedia.org/wiki em julho/2005.

[ZAN96] Zanella, Ana Lucia. *Um Ambiente Colaborativo para Apôio a um Curso de Projeto de Software Orientados a Objetos*. Tese de Mestrado do Curso de Mestrado da Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, dezembro de 1996.

Anexo I

Curso com o micromundo AllKara

1. Introdução

Para introduzir as atividades de programação, um micromundo, habitado por um inseto, joaninha, chamado **Kara** será utilizado. A Figura 1-1 mostra este habitante.

Figura 1-1: Kara.

Kara vive em um mundo onde pode se deslocar, girar, remover trevos, colocar trevos. Em certas situações pode encontrar-se com um cogumelo ou com um tronco de árvore à frente, os quais devem ser transpostos.

A nossa tarefa é conduzir **Kara** de modo que ele consiga resolver desafios propostos. Ao deparar com os desafios é preciso em primeiro lugar que encontremos uma solução baseada nas potencialidades de Kara e após vislumbrar esta solução utilizar a ferramenta adequada para fazer com que Kara execute a solução proposta.

Assim, conscientes que Kara conhece comandos, é preciso criar uma "receita" para que resolva os desafios. Se a receita for correta, o problema será resolvido, se a receita possuir erros, o problema não será resolvido.

Para começar utilizaremos uma ferramenta para nos comunicar com **Kara** chamada máquina de estados finitos. Esta máquina trabalha com a idéia de estados. Os estados representam situações que **Kara** pode se encontrar no seu mundo. Para cada estado são estabelecidas transições, ou seja, avaliações de condições que determinam ações a serem executadas.

Em uma segunda etapa, é apresentada outra forma de dar comandos a Kara. Esta nova ferramenta é uma linguagem de programação chamada JavaKara pois se parece muito com a linguagem Java. Os comandos e sensores continham os mesmos, porém, a mudança é a forma de elaborar o programa.

Com a solução dos desafios escrita em JavaKara a linguagem de programação é apresentada e sua sintaxe explorada.

Para finalizar esta introdução é feita uma abordagem inicial dos conceitos do paradigma de orientação a objetos com uma conexão com Kara e seu micromundo. Novas classes são exemplificadas e utilizadas e a necessidade de uma nova abordagem para avançar nos conceitos do paradigma é sentida.

2. Programação com a máquina de estados finitos

Vamos ao nosso primeiro desafio. Como primeira tarefa componha seu mundo utilizando um trevo e **Kara**. Coloque-o sobre o trevo. Nosso objetivo é fazer com que **Kara** pegue o trevo. O mundo de **Kara** é exibido na Figura 2-1.Criação do diagrama de estados e da tabela de transições Como fazer? Imagine o seguinte: **Kara** está sobre o trevo? Se sim, então pegue e fim. Para representar esta situação podemos criar dois estados e**ncontrar** e **parar**. Temos a seguinte representação gráfica da Figura 2-1.

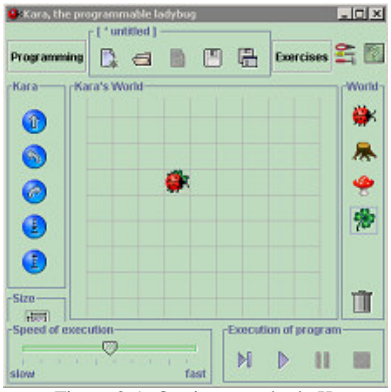

Figura 2-1: O micromundo de Kara.

Neste esquema podemos ver que o processo se inicia no primeiro estado **encontrar**. Neste estado precisamos verificar se existe um trevo no chão, caso afirmativo, **Kara** deve remover e parar.

Como fazer com que **Kara** execute certos comandos, considerando que encontrou um trevo no chão? Para isto elabora-se uma "tabela" de transições baseada em testes de condições, este é o diagrama de transições. Abaixo está a descrição de como montá-lo.

Na Figura 2-2 nota-se um botão no canto superior esquerdo com a identificação **Programming**. Este botão permite acessar o editor de programas, neste caso o editor de diagramas de estado e da tabela de transições. Clique neste botão e verá o editor como mostrado na Figura 2-3.

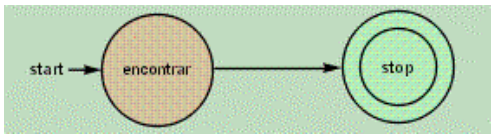

Figura 2-2: Representação dos estados encontrar e stop.

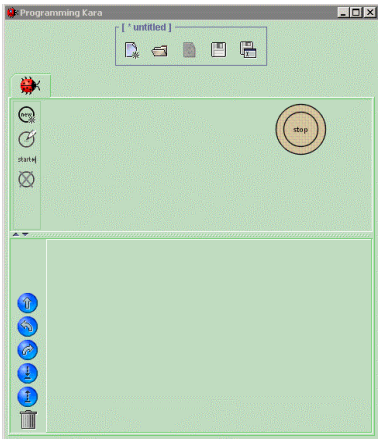

Figura 2-3: Editor do diagrama de estados.

Os próximos passos criam as transições necessárias. Crie o primeiro estado **encontrar**, clicando em **new**. Observe a janela da Figura 2-4 que permite programar o novo estado.

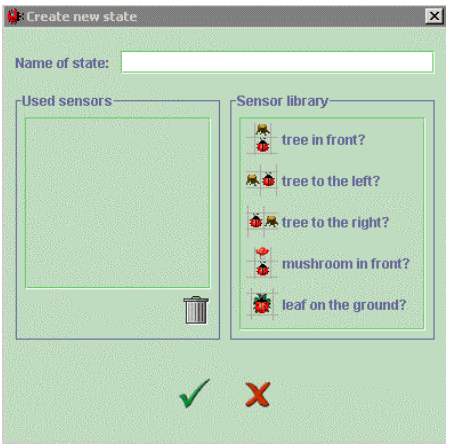

Figura 2-4 Criação das transições do estado.

O que existe nesta tela? Em primeiro lugar identifica-se o estado através do *Name of state*, no exemplo **encontrar**. Do lado direito da janela estão os sensores conhecidos de Kara, que representam situações em que **Kara** pode estar, observe na Figura 2-4. Como é preciso fazer com que Kara pegue um trevo caso esteja sobre ele, utilize o sensor **leaf on the ground?** (trevo no chão?). Como existe apenas um trevo a colher, isto é tudo. Clique sobre este sensor e arraste-o para a caixa intitulada ***Used sensors*** (Sensores Utilizados). Confirme a transição e finalize a programação para este problema clicando no ícone confirmar mostrado na Figura 2-5.

Figura 2-5 Ícones para Confirmar e Cancelar respectivamente.

Agora, especifique o **Kara** fará ao encontrar o trevo. Observe a aba que representa o estado **encontrar**, na parte inferior da Figura 1-7. Clique sobre o asterisco e veja as opções mostradas na Figura 2-6**.**

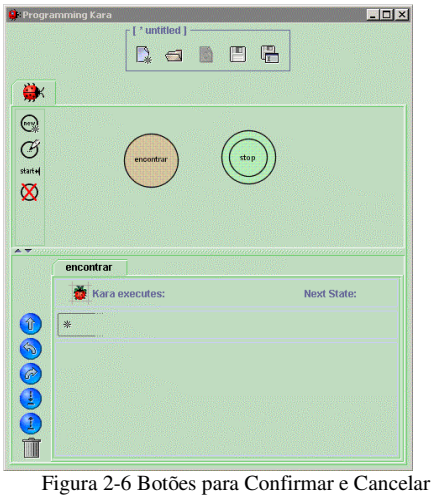

Figura 2-6 Botões para Confirmar e Cancelar respectivamente.

Figura 2-7 Botões para Confirmar e Cancelar respectivamente.

Agora defina os comandos que **Kara** irá executar. Os comandos encontram-se nos ícones azuis da Figura 2-6.
Como é preciso especificar que **Kara** pegue o trevo se encontrá-lo especifique:

- Coloque a marcação (x em vermelho) em yes (sim) como pode ser visto na Figura 2-7;
- Estabeleça a ação a ser executada quando o sensor é verdadeiro (yes), como pode ser observado na Figura 2-7;
- Especifique também o estado para o qual **Kara** irá após executar as ações estipuladas, neste caso, **stop** (parar).

Falta apenas um detalhe: especificar o ponto de partida do processo. Ao iniciar a tarefa **Kara** estará posicionado sobre o trevo como mostrado na Figura 2-1. Assim o ponto de partida é o estado **encontrar**. Clique em **start** que está posicionado na área de montagem do diagrama de estados, mostrado na Figura 2-6 e como existe apenas um estado que pode ser o inicial, este é automaticamente selecionado.

Pronto? O programa está pronto. Agora é preciso solicitar que Kara o execute. Para isto retorne ao mundo inicial, o da Figura 2-1 e clique no botão de execução que se encontra na parte inferior da tela, exatamente é o segundo ícone do conjunto ***Execution of program***.

Mudando um pouco o problema, observe a configuração inicial da Figura 2-8.

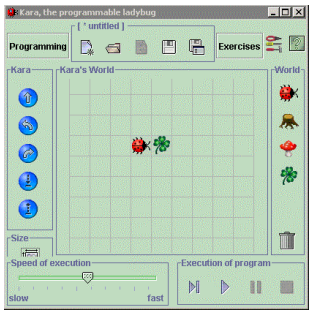

Figura 2-8 Outra configuração do mundo.

Caso **Kara** não encontre o trevo, o que deverá fazer? Podemos continuar no mesmo diagrama de transições, porém com uma alteração. Caso Kara não encontre o trevo o que deve fazer? Andar para frente. Assim escreva esta transição: *caso não encontre um trevo deve movimentar para frente e continuar no mesmo estado.* Implemente-a. Seu novo programa está mostrado na Figura 2-9.

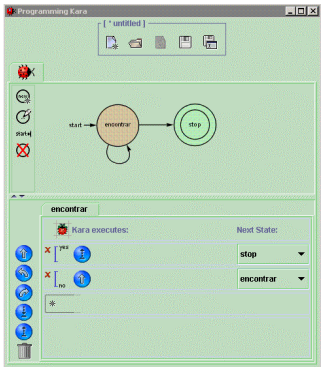

Figura 2-9 O programa modificado para o novo problema.

Que tipo de problema esta solução proposta resolve?
Nas próximas sessões são propostos desafios que conduzem a o desenvolvimento de novas estratégias de resolução de problemas.

Desafios

Após esta introdução ao mundo de **Kara** e sua programação através da máquina de estados finitos novos desafios serão propostos. No próprio ambiente temos acesso a diferentes exercícios através do botão **Exercises**. Nos exercícios pode-se "carregar" um mundo específico para os exercícios propostos, sem ter que montá-los. É claro que é possível criar desafios próprios, configurar o mundo necessário e salvá-lo em um arquivo.

Vamos começar com a Tarefa 1 (**Task 1**).

Carregar mundos: em primeiro lugar carregue o mundo de **Kara**.

Como? Clique em **Worlds** e selecione a configuração **Cloverleaf trail 1**, clicando sobre o ícone correspondente. Para ver a configuração do mundo volte à tela inicial (clique no ícone correspondente na barra de tarefas).

Desafio 1

Kara precisa percorrer uma trilha, como mostrado na Figura 2-10. Onde encontrar um trevo deve removê-lo, onde não encontrar um deverá colocar um no local. Este processo deverá se repetir até que um tronco seja encontrado.

Figura 2-10 Configuração para o desafio 1.

Lembre-se que na máquina de estados finitos é preciso estabelecer o diagrama de transições, ou seja:

- Os estados necessários para resolver o problema.
- Estabelecer também o estado inicial.
- As situações que **Kara** irá enfrentar, ou seja, que sensores precisam ser considerados.
- Os comandos que serão executados em cada situação.

Crie seu modelo e teste com este mundo. Após elaborar seu modelo consulte a resolução fornecida pelo sistema e analise-a. Mesmo tendo programado corretamente leia a título de comparação. Existem observações importantes.

Resolução: **Kara** precisa caminhar até encontrar um tronco de árvore. Que estados podemos definir? O nome fica a critério de cada um, sugiro **caminhar**. Ao caminhar o que **Kara** poderá encontrar? De acordo com o mundo proposto **Kara** pode:

- estar sobre uma folha;
- não estar sobre uma folha;
- encontrar o tronco de uma árvore (ter um tronco à frente).

Assim, são necessários dois sensores:

- trevo no chão;
- tronco de árvore à frente.

Agora temos que montar nosso diagrama:

- denominar o estado – sugestão: **caminhar;**
- selecionar os sensores a utilizar;
- programar o estado caminhar de acordo com os sensores.

Lembre-se que agora existem dois sensores a considerar, especifique cada situação possível:

- **Kara está sobre um local vazio**. Caso não encontre uma folha e não esteja em frente a um tronco de árvore o que deve fazer? Deixar um trevo e ir em frente, continuando seu caminho (a procura do tronco para poder párar a caminhada);
- **Kara encontra um trevo no chão e não está em frente a um tronco de árvore**. O que fazer? Pega o trevo e segue em frente;
- **Kara encontra-se em frente ao tronco de uma árvore e não encontra um trevo**. O que fazer? Deixa um trevo e pára;
- **Kara encontra-se em frente ao tronco e encontra uma folha no chão**. Ela pega o trevo e pára.

Elabore o programa e execute-o. Caso não funcione corretamente, analise o problema, volte ao editor de estados, corrija-o e execute novamente. A atividade de procurar os erros no programa chama-se **depuração (debug)**. É um bom momento de se aprender. Muitas vezes os erros nos ensinam mais que os acertos.

Agora volte aos exercícios e carregue o outro mundo o **Cloverleaf trail 2**.

Será que o programa que elaborou resolve este também? Por que? Teste.

Carregue outro mundo, o **Cloverleaf trail 3**. Funciona ou não?

Existe a possibilidade de gravar o programa elaborado no computador, localize o ícone e grave seu programa. A Figura 2-11 mostra os ícones para acesso ao arquivo de gravação.

Figura 2-11 Ícones para acesso aos arquivos dos programas.

Lembre-se:

- Clareza é fundamental. Não tente ocultar seus trabalhos até de você mesmo. Nomes de programas sugestivos ajudam a identificar os programas de forma bem mais fácil posteriormente.
- Organização é fundamental para desenvolver bons trabalhos. Que tal criar um diretório (pasta) para gravar os trabalhos desenvolvidos com este tipo de programação?
- Salve seu programa em disco. Estando apenas na memória não tem como recuperá-lo posteriormente. Imagine que ocorra uma falta de energia em algum momento!!!! Você pode perder tudo se não salvou. Segurança é também um ótimo princípio.

Observação: Reparem que todas as possíveis soluções para a utilização dos dois sensores foram checadas. A seguinte tabela mostra a configuração possível dos dois estados, utilizando os valores verdadeiro e falso:

Em frente o tronco?	**Encontrou trevo?**
Falso	Falso
Falso	Verdadeiro
Verdadeiro	Falso
Verdadeiro	Verdadeiro

Como existem dois estados possíveis o número de combinações destes estados é 4. Para generalizar, para verificar todas as possibilidades é necessário 2^n combinações, ou seja, 2^n linhas na tabela. Onde **n** representa o número de estados considerados. No nosso caso $2^2 = 4$.

Nem sempre é preciso analisar todas as possibilidades, porém, se todos forem checadas o programa não falhará, uma vez que todas as possibilidades foram analisadas e **Kara** saberá agir adequadamente, pois foi programada para todas as situações. O que pode acontecer se não analisarmos todos os casos?

É o que pode acontecer quando fazemos um programa comercial, ou quando utilizamos um programa, como por exemplo, um caixa eletrônico de um banco e de repente o programa pára. Não prossegue, nem nos permite interferir. Simplesmente se perdeu o controle, provavelmente, caiu-se em um estado onde a situação ocorrida não foi prevista.

Por outro lado, algumas vezes o número de possibilidades é tão grande que precisamos utilizar outras estratégias para garantir que o programa irá funcionar adequadamente, ao invés de checar as "infinitas" possibilidades. Com o tempo avança-se nestes caminhos.

Desafio 2

Agora carregar o próximo mundo **Tunnel entry 1**. Como ele é formado?
O último programa resolverá este problema? Por que?

Objetivo: Kara procura a entrada do túnel, posição 2a. Escreva um programa que conduza Kara à posição 2a mostrada na Figura 2-12.

Figura 2-12 Configuração para o desafio 2.

Atenção: alguns túneis têm somente parede de um lado, outros à esquerda, outros à direita.

Kara precisa entrar no túnel de troncos e parar quando atingir a posição 2a. Nesta posição qual a situação que Kara se encontra? Pense um pouco, observe a Figura 2-12 antes de continuar.

Você sabia que as DÚVIDAS expostas podem se transformar em instrumento de aprendizagem, para o grupo todo e as impostas fazem-nos caminhar para trás? Exponha-as. Você aprende e cria oportunidade dos outros também aprenderem

As ferramentas para ajudar a destruir as dificuldades estão em suas mãos: pergunte ao instrutor, veja exercícios resolvidos, pergunte ao amigo, analise outras soluções, exponha soluções que achou interessante.

Desafio 3

Passemos ao outro desafio no túnel. **Kara** precisa alcançar a posição **2b.** Considere a mesma configuração do mundo do desafio anterior. Teste a solução dada, funciona? Por que?

Kara pára assim que encontra os troncos à esquerda e à direita, ou seja, pára quando entra no túnel. Mas o objetivo é parar ao sair do túnel. O que fazer?

Teste seu programa e veja se ele funciona nos outros mundos: **tunnel exit 1**, **tunnel exit 2**, **tunnel exit 3**. Teste também em **tunnel entry 1**, **tunnel entry 2** e **tunnel entry 3**.

Desafio 4

Kara procura um trevo. Ele sabe que existe um caminho direto, porém tem que circular os troncos das árvores. Felizmente, não existem dois troncos de árvores seguidos, como podemos ver na Figura 2-13. Escreva um programa que o encaminhe até o trevo!

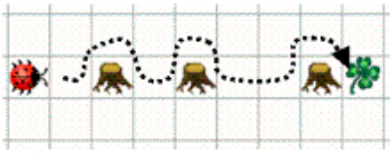

Figura 2-13 Configuração do mundo para o desafio 4.

Desafio 5

Outra situação em que **Kara** se encontra está mostrada na Figura 2-14.

Figura 2-14 Configuração do mundo para o desafio 4.

E agora? O programa anterior funciona? Por que?

Programe **Kara** de modo que ele encontre o trevo, porém com a possibilidade de encontrar árvores seguidas. Lembre-se que podemos utilizar mais de um estado além do final.

Desafio 6

Neste desafio Kara precisa encontrar o trevo em uma floresta, como mostrado na Figura 2-15. Analise bem o problema. Estabeleça as situações que Kara pode encontrar. E os estados possíveis.

Figura 2-15 Configuração do mundo para o desafio 6.

Desafio 7

Como fazer para que **Kara** circule o tronco da Figura 2-16?

Figura 2-16 Configuração do mundo para o desafio 7.

Desafio 8

Agora, **Kara** deverá contornar os troncos abaixo, Figura 2-17.

Figura 2-17 Configuração do mundo para o desafio 8.

Desafio 9

Kara irá proteger uma reserva de trevos, como mostrada Figura 2-18**.**

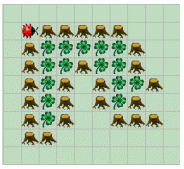

Figura 2-18 Configuração do mundo para o desafio 9.

Teste o procedimento montado para as outras configurações de mundo dentro desta seqüência de **Follow trees**.

3. JavaKara

Agora a comunicação com **Kara** será feita de outra forma através de JavaKara. Com JavaKara elaboramos uma seqüência de instruções em forma de um "texto", contendo os comandos que **Kara** deverá executar e alguns elementos especiais que permitem que sejam dados comandos especiais, como executar comando caso uma condição seja satisfeita, executar um mesmo comando várias vezes.

Este "texto" constitui um programa que representa os mesmos comandos para Kara, porém serão escritos de outra forma, agora será utilizada uma linguagem de programação.

O que é uma linguagem? O que á a Língua Portuguesa. De forma simplificada, linguagem é um conjunto de elementos que nos permite expressar idéias. A partir de um conjunto de elementos, podemos nos expressar para que Kara possa implementar as soluções pensadas pelos programadores.

Desta forma, a comunicação com **Kara**, passará a ser feita através da linguagem JavaKara.

Como o nome sugere, JavaKara é muito próxima da linguagem de programação Java. Possui a mesma forma de expressão, sintaxe, com a vantagem de facilitar o entendimento de vários conceitos fundamentais em programação de computadores.

A princípio esta ferramenta é usada para desenvolver programas para **Kara**, com o objetivo de resolver desafios propostos. Com a representação da solução encontrada na linguagem JavaKara elementos básicos de programação de computadores são introduzidos. Estes elementos são utilizados nas linguagens de programação em geral.

Este primeiro contato permite conhecer e aprender estruturas essenciais, tais como: seqüência de comandos, comandos de seleção, utilização de variáveis, comandos de repetição, criação de métodos, passagem de parâmetros, que são utilizadas para representar as situações e resolver os desafios.

O ambiente de Kara e sua linguagem permitem experimentar o uso de outros objetos além de Kara que são **tools** e **world**.

Introdução

No mundo AllKara o primeiro objeto é **kara**. Este objeto consegue executar comandos (métodos) e através destes métodos é possível alterar seu estado, ou o estado do mundo onde se encontra. Por exemplo, o comando:

Kara.move( );

solicita para o objeto **Kara** a execução do método move, que faz com que **Kara** se desloque uma posição à frente, relativa à posição atual, modificando assim sua localização no mundo. Outro exemplo:

Kara.putLeaf( );

faz com que **Kara** coloque um trevo na posição onde se encontra. Cão seja necessário fazer com que **Kara** se desloque para a posição linha 1 (x =1) e coluna 2 (y = 2), o comando abaixo realiza a solicitação:

Kara.setPosition(1,2);

Qualquer comunicação com **Kara** é feita desta forma: nome do objeto seguido do comando a ser executado e informações adicionais, quando necessário.

JavaKaraProgram

Em JavaKara existe um formato (molde) que utilizamos para criar os programas, chamado **JavaKaraProgram**. Todos os programas que criamos são uma extensão deste formato. Ou seja, possui as características e funcionalidades do formato básico, acrescido dos novos elementos que podem ser incluídos para resolver os desafios propostos. Um método chamado **myProgram** é muito importante. Este representa o ponto inicial de execução das aplicações em JavaKara.

Objeto Kara

O comportamento do objeto **Kara** é o mesmo, a diferença é a forma de comandá-lo. Na máquina de estados finitos a programação era feita através da montagem do diagrama de estados finitos e do diagrama de transições. Com JavaKara os programas são escritos como um texto, montando uma seqüência de instruções para **Kara** executar.

Observe o objeto **Kara** na Figura 3-1. Os nomes dos elementos acima representam informações que **Kara** armazena. Estas informações determinam o estado de **Kara**.

Figura 3-1 Kara em seu mundo.

Quais são as características de **Kara**, considerando a Figura 3-1? **Kara** está na posiçãoX = 1 (primeira coluna, começa a numeração da coluna 0), posiçãoY = 1 (primeira linha), direção = direita, células vizinhas = todas vazias, tamanho = padrão ...

Ao solicitar que **Kara** desloque uma casa a frente, utilizando o comando:

kara.move( );

o estado de **Kara** é alterado, isto é, neste caso, acrescenta-se 1 à posição referente à coluna, ou seja a posicaX passará para 2.

Abaixo temos os comandos disponíveis (métodos) para solicitar serviços a **Kara**:

Tabela 3-1 Comandos de Kara.

Método	Significado
move( );	Faz com que **Kara** se desloque uma casa a frente ou retorne ao início da linha caso esteja na última posição.
	Exemplo: Kara.move( );
turnLeft( );	**Kara** irá girar 90º para a esquerda de acordo com sua direção atual.
	Exemplo: Kara.turnLeft( );
turnRight( );	**Kara** irá girar 90º para a direita de acordo com sua direção atual.
	Exemplo: Kara.turnRight( );
putLeaf( );	**Kara** irá colocar um trevo na posição atual, se estiver vazia.
	Exemplo: Kara.putLeaf( );
removeLeaf( );	**Kara** irá remover um trevo na posição atual, se lá existir um.
	Exemplo: Kara.putLeaf( );
setPosition(x,y);	**Kara** irá se posicionar na posição solicitada.
	Exemplo: Kara.setPosition(3,2);

Criar programas em JavaKara

Assim, programar **Kara** significa elaborar uma seqüência de comandos que fará com que ela realize uma tarefa específica. Esta seqüência é formada utilizando-se métodos que **Kara** possui (comandos que sabe executar) e outros elementos da linguagem que aprenderemos durante o curso como variáveis, como seleção, repetição.

Todos os programas que criamos em JavaKara precisam ser uma extensão do molde (classe) **JavaKaraProgram**. Veja o código de um programa em JavaKara:

```
public class RemoveUmaFolha extends JavaKaraProgram {
   // novos métodos são colocados aqui
   public void myProgram( ) {
      kara.removeLeaf( );
   }
}
```

Código 3-1 Programa RemoveUmaFolha.

O texto que vem após // é um comentário, este é um texto que facilita a leitura pelo leitor e não faz diferença para o computador.

O programa em JavaKara é uma extensão de JavaKaraProgram. A classe RemoveUmaFolha é uma classe filha de JavaKaraProgram. E sendo filha, esta "herda" todas as características da classe mãe. Através deste mecanismo de extensão (**herança**) criamos uma nova classe que possui todas as características e funcionalidades da classe mãe (**JavaKaraProgram**) e incorporamos a ela novos elementos de maneira a atender novas necessidades.

No exemplo acima observe o nome da classe: **RemoveUmaFolha**. Este é um **identificador**, nome que identifica um elemento de nossa classe. Temos algumas regras para esta denominação:

- Deve ser sugestivo;
- Não deve conter espaços em branco;
- Não deve ser uma palavra reservada da linguagem;

Observe também que o identificador da classe tem a inicial maiúscula, como no nome da classe mãe JavaKaraProgram. Todo nome da classe se iniciará com a primeira letra maiúscula, seguida das demais iniciais, se for um nome com mais de uma palavra, maiúsculas.

Padrões adotados

Algumas convenções são adotadas no sentido de facilitar o entendimento da classe pelos criadores e por outras pessoas que venham a utilizá-la.

Padrão 1. Os nomes de classes iniciam-se com letra maiúscula.

Observe também a declaração do método **myProgram**. Este método é o ponto de partida para a execução do programa, assim é obrigatório nas classes onde desejamos ter um ponto de início do processo de execução. Este se inicia com letra minúscula.

Padrão 2. Os nomes de métodos possuem a letra inicial minúscula.

Observe que ele é público, isto quer dizer que ele pode ser utilizado por outra classe.

Além disso, ele possui uma outra construção diferente **void**. Este termo informa que o método não retorna valor. Já vimos alguns métodos (**move**, **turnLeft**, **turnRight**, **putLeaf**, **removeLeaf**, **setPosition**) que também retornam nada (**void**). Por exemplo, ao executar: kara.move( );

O que é feito? ***Kara*** *move uma casa à frente*. O que é retornado? ***Nada***. Pense nos outros métodos que mencionamos, na Tabela 3-1.

Observe a chamada do método acima. O nome do objeto **Kara** está escrito com a inicial minúscula. Se não for assim, este objeto não será reconhecido, o que geraria uma mensagem de erro.

Compare os códigos 3-1 e 3-2. Qual está mais fácil de ser lido? Esta característica nos leva ao quarto padrão listado abaixo.

```
public class RemoveUmaFolha extends JavaKaraProgram {
// novos métodos são colocados aqui
public void myProgram( ) {
kara.removeLeaf( );
}
}
```

Código 3-2 Programa RemoveUmaFolha sem indentação.

Padrão 3. Os espaços internos antes de declarações e/ou comandos servem para facilitar o entendimento da seqüência. Este procedimento é conhecido como *indentação* e é uma boa prática de programação. Para o computador não faz diferença, mas para os leitores humanos é o diferencial.

Editar o programa

Observe as chaves, elas abrem e fecham uma seqüência, marcando seu início e seu fim. No Código 3-1 foram utilizadas para iniciar e finalizar a classe **RemoveUmaFolha** e iniciar e finalizar o método **myProgram**.

Toda classe virá entre o par { } e todo o corpo do método também.

O nome do arquivo deverá ser o mesmo nome da classe. Neste caso, o nosso arquivo deverá ter o nome de **RemoveUmaFolha**. Automaticamente é atribuída a extensão **.java** ao arquivo. Este é o nosso programa fonte ou código fonte em JavaKara.

A linguagem JavaKara (Java) é sensível ao tipo de letra usada, ou seja, faz diferença usar letras maiúsculas e minúsculas. Por exemplo **kara** é diferente de **Kara**. Prestem muita atenção neste detalhe.

Observe o comando abaixo:

```
kara.removeLeaf( );
```

Como este comando foi montado? Em primeiro lugar colocamos o nome do objeto para o qual estamos solicitando um serviço, no exemplo **kara**, em seguida colocamos um ponto para separar o objeto do nome do método, no caso **removeLeaf**. Por fim, temos **( )** que é o local para colocar informações que o método precisa para ser executado (que chamamos de **parâmetros**), neste exemplo, o método não precisa de informação, assim está em branco. O "**;**" significa fim da linha de comando.

A seqüência de desafios que seguem fixam e introduzem conceitos.

Desafio 10

Este desafio foi o primeiro a ser resolvido por **kara** com a máquina de estados finitos e consiste em elaborar um mundo bem simples, com **kara** sobre um trevo e programá-lo para que remova este trevo.

Crie o mundo e escreva o programa em JavaKara para implementar a solução. Acesse o editor de programas, clicando sobre o botão **Programming** do lado esquerdo, superior da tela. Automaticamente é exibido um modelo de programa, observe-o. O que pode ser excluído para simplificar o programa? Como estamos usando o JDK, podemos excluir a linha do comando **import** e se desejar, os comentários.

Vamos ao primeiro programa! Digite-o como mostrado no Código 3-1.

Grave seu programa em local adequado, dando a ele o mesmo nome da classe, ou seja, RemoveFolha. Observe a extensão dada.

Observe que no editor de programas, quando posiciona o cursor próximo ao símbolo de início ou fim de comandos ({ ou }) o correspondente é destacado.

Compilar o programa

O texto do programa está pronto. Agora é preciso **compilar** o programa (nova classe) para que possa ser executado por Kara. No próprio editor de programas temos um botão para este processo.

Compile seu programa. O que aconteceu? Se não foi detectado erro crie um, por exemplo tire o ponto e vírgula do final do comando e peça para compilar. O que aconteceu? Observe a mensagem de erro e o destaque do local provável no programa. Corrija o erro e compile novamente.

Caso seu programa esteja correto, isto é, sem erros de escrita, um código compilado é gerado, com o mesmo nome da classe (RemoveFolha) porém com a extensão .class. Este processo será discutido posteriormente com detalhes.

Este mecanismo de correção de erros e compilação deve ser repetido até que erros de compilação (erros na escrita de comandos) não sejam detectados. Sempre recompile a classe após as correções para que o novo arquivo com a extensão .class seja gerado.

Agora depois de compilado vá ao mundo de Kara e peça que ele execute seu programa.

Desafio 11

Agora vamos fazer com que Kara remova uma seqüência de 3 trevos como mostrado na Figura 3-2.

Figura 3-2 Configuração para o desafio 11.

Crie o mundo. Elabore o programa. Grave-o. Compile-o. Corrija os erros se houver e solicite a execução. Tranqüilo?

Desafio 12

Coloque **Kara** na posição (0,0) e um trevo na posição (3,3). Lembre-se que a numeração das linhas e colunas começa no valor zero (0). Elabore um programa para que **Kara** se posicione na posição (3,3) e remova o trevo.

Desafio 13

Kara precisa remover um trevo, porém existe um tronco a sua frente. Vamos ajudá-la? O esquema de seu mundo está na Figura 3-3.

Figura 3-3 Configuração para o desafio 13.

Desafio 14

Kara precisa remover um trevo se ele o detectar no chão. Como solicitar a execução de um comando de acordo com a verificação de uma condição? O mundo está na Figura 3-4.

Figura 3-4 Configuração para o desafio 13.

Escrevendo em português o programa fica:

```
Se Kara encontra um trevo no chão
   remova-o
```

Neste caso utilizaremos um dos sensores de **Kara**. Certo? Qual é o correspondente na programação anterior de **Kara**? Existe um sensor na máquina de estados finitos que retorna verdadeiro caso Kara esteja sobre uma folha e falso caso contrário. Em JavaKara temos a seguinte construção:

```
if  (kara.onLeaf( ) ) {
      kara.removeLeaf( );
}
```

O programa com esta construção é mostrado no Código 3-3.

```
public class RemoveUmaFolha extends JavaKaraProgram {
  public void myProgram( ) {
     if (Kara.onLeaf()) {
        Kara.removeLeaf( );
     }
  }
}
```

Código 3-3 Programa RemoveUmaFolha com verificação.

Assim, **Kara** faz um teste e dependendo do resultado executa o comando. No nosso dia a dia este tipo de teste é feito a todo momento. Exemplos: se não chover irei à piscina, se chegar cedo em casa arrumarei o guarda-roupas, ao completar 30 minutos lavar os cabelos, se crédito é maior ou igual a 100 comprar caso contrário poupar o dinheiro.

Com a estrutura condicional amplia-se o poder de **Kara** de resolver problemas. No roteiro anterior ele podia apenas executar comandos na seqüência. Os sensores de Kara estão listados na Tabela 3-2.

Tabela 3-2 Sensores de Kara.

Sensor	Significado
treeFront()	Kara está em frente a uma árvore?
	Exemplo: if (Kara.treeFront( )) Kara.turnRight( );
treeLeft()	Há uma árvore à esquerda de Kara?
	Exemplo: if (Kara.treeLeft( )) Kara.turnRight( );
treeRight()	Há uma árvore à direita de Kara?
	Exemplo: if (Kara.treeRight( )) Kara.turnRight( );
onLeaf()	Esta Kara sobre uma folha?
	Exemplo: if (Kara.onLeaf( )) Kara.removeLeaf ( );
mushroomFront()	Esta Kara em frente a um cogumelo?
	Exemplo: if (Kara.mushroomFront( )) Kara.turnLeft ( );

Desafio 15

Figura 3-5 Configuração para o desafio 15.

Kara deve remover o trevo em qualquer uma das situações mostradas na Figura 3-5. Na condição a ser verificada podemos utilizar mais de um sensor (teste). Em muitas situações a condição composta é necessária. O programa para resolver as situações abaixo precisa de um teste composto:

- Se existe um tronco à esquerda e a direita
  Avance uma casa e remova o trevo
- Se existe um tronco a direita e a frente
  Vire a esquerda, avance uma casa e remova o trevo
- Se existe um tronco a esquerda e à frente
  Vire a direita, avance uma casa e remova o trevo

Em JavaKara o primeiro teste fica:

```
if   (kara.treeLeft( ) && kara.treeRight( ) )
```

Programe-a. Lembre-se que seu programa deverá resolver os problemas abaixo. Assim, crie a primeira configuração para o mundo de **Kara**, teste seu programa. Altere a configuração para o segundo caso e teste. E faça o mesmo para o terceiro caso.

Na Figura 3-6 uma nova situação é propostas para **Kara**, ele precisa remover o trevo, nas situações mostradas. Isto é, se encontrar um tronco deverá contorná-lo primeiro, caso contrário deverá seguir em frente duas casas. E finalmente, quando encontrar o trevo, removê-lo.

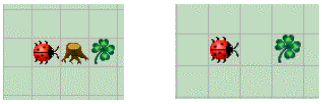

Figura 3-6 Configuração para os dois problemas.

Em português podemos escrever nosso programa:

```
Se Kara detectar um tronco a frente
        Contornar o tronco   // contornar o tronco e posicionar-se
Senão
        Siga em frente   //comando para dar um passo
        Siga em frente  // dar outro passo e posicionar-se
Remover o trevo
```

Kara não conhece o comando **contornar o tronco**. Que comandos serão necessários para que ele contorne o tronco e se posicione sobre o trevo? A seguinte seqüência pode ser utilizada:

```
Se Kara detectar um tronco a frente
    Vire a esquerda
    Siga em frente
    Vire a direita
    Siga em frente
    Siga em frente
    Vire a direita
    Siga em frente
    Vire a esquerda
Senão
    Siga em frente
    Siga em frente
Remover o trevo
```

O trecho de programa, em JavaKara, fica como no Código 3-4.

```
if ( Kara.treeFront ( ) )  {
        Kara.turnLeft( );
        Kara.move( );
        Kara.turnRight( );
        Kara.move( );
        Kara.move( );
        Kara.turnRight( );
        Kara.move( );
        Kara.turnLeft( );
    }
    else {
        Kara.move( );
        Kara.move( );
    }
    Kara.removeLeaf( );
```

Código 3-4 Trecho de programa em JavaKara para contornar e posicionar.

O programa completo em JavaKara fica:

```
public class PulaTronco extends JavaKaraProgram {
  public void myProgram( ) {
    if (Kara.treeFront( )) {
        Kara.turnLeft( );
        Kara.move( );
        Kara.turnRight( );
        Kara.move( );
        Kara.move( );
        Kara.turnRight( );
        Kara.move( );
        Kara.turnLeft( );
    }
    else {
        Kara.move( );
        Kara.move( );
    }
    Kara.removeLeaf( );
  }
}
```

Código 3-5 Código em JavaKara para PulaTronco.

Criar métodos (estabelecer comportamentos)

Kara possui vários métodos, mensagens as quais pode responder ou comandos que sabe executar. Também pode aprender novos comandos, ou seja, podem-se criar novos métodos, estendendo o poder de Kara.

Estes novos métodos possuem um lugar específico dentro do programa, eles se situam antes do método **myProgram**, como pode ser visto no pseudocódigo Código 3-6.

```
public class NomeDaClasse extends JavaKaraProgram {

  // espaço para colocar todos os nossos  NOVOS métodos

   public void myProgram ( ) {
          // espaço para as mensagens (comandos) para
       // iniciar o processamento
   }
}
```

Código 3-6 Código em JavaKara para PulaTronco.

Ao invés de trabalhar com apenas um programa, como fizemos até agora, colocando todas os comandos (chamada dos métodos) no método **myProgram**, podemos criar métodos que desempenham funções específicas e depois chamá-los.

Ao dividirmos um problema em partes, este se torna mais fácil de resolver. Por exemplo:

- Quando uma seqüência de comandos se repete, é possível economizar linhas de códigos criando a parte que ser repete dentro de um método e "chamando-o" quanto necessário.
- O método pode ser reutilizado em outro contexto.
- O programa fica mais claro e fácil de entender.

Em JavaKara, os métodos são criados no local indicado com o comentário no Código 3-5.

Os métodos possuem uma assinatura, ou cabeçalho. Esta determina como o método deve ser chamado. A forma geral do método é mostrada abaixo.

```
modificadoresDeAcesso  tipoDoValorDeRetorno   nomeDoMetodo  (parâmetros) {
      comandos que compõe o corpo do método;
}
```

modificadoresDeAcesso: definem a visibilidade de um campo, de um construtor ou método. Elementos públicos (public) são acessíveis a partir de dentro da mesma classe e de outras classes; elementos privados (private) são acessíveis somente dentro da mesma classe.

tipoDoValorDeRetorno: define o tipo de retorno do método, quando nada retorna utilizamos a palavra **void**.

nomeDoMetodo: define o nome do método e, por padronização, deve começar letra minúscula.

Parâmetros: define a lista de dados que devem ser fornecidos ao método para que ele funcione, coloca-se o tipo do parâmetro seguido do seu nome; quando não existem parâmetros coloca-se o abre e fecha parênteses vazios.

Corpo do método: onde se digita o conteúdo do método, declarações e comandos.

O programa exibido no Código 3-6 é um bloco único. Pode-se criar um método chamado contornar que leve **Kara** a contornar o tronco e voltar a sua posição na linha de baixo. O programa com este método está no Código 3-7.

```
public class PulaTronco extends JavaKaraProgram {

 public  void contornar( ) {
    Kara.turnLeft( );
    Kara.move( );
    Kara.turnRight( );
    Kara.move( );
    Kara.move( );
    Kara.turnRight( );
    Kara.move( );
    Kara.turnLeft( );
 }

 public void myProgram( ) {
    if  ( Kara.treeFront( ) ) {
      contornar( );
    }
    else {
       Kara.move( );
       Kara.move( );
    }
    Kara.removeLeaf( );
 }
}
```

Código 3-7 Código em JavaKara para PulaTronco com métodos.

Observe como criar um método dentro do seu programa e como chamá-lo dentro do método myProgram. Existem parâmetros? Existe valor de retorno?

Desafio 16

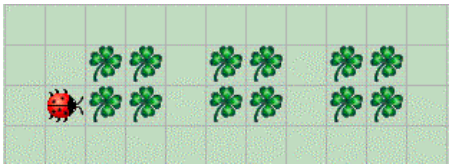

Figura 3-7 Configuração para o desafio 16.

Conduza **Kara** de modo que ele consiga retirar as seqüências de trevos. Crie um método que elimine um "quadrado" de trevos. Como fazer para que sejam removidos os três conjuntos de trevos mostrados na Figura 3-7?

Estruturas de Repetição

Desafio 17

Kara precisa remover o trevo nas situações mostradas na Figura 3-8.

Figura 3-8 Configuração para o desafio 17.

Como fazer para que **Kara** remova o trevo em qualquer uma das situações acima? Quantos passos dar a frente para encontrar o trevo e removê-lo? Isto é impossível saber. Só sabemos que na mesma linha, **Kara** encontrará um trevo, ao encontrá-lo deverá pegá-lo e finalizar. A caminhada deve parar apenas quando o encontrar. Em português poderíamos escrever:

```
Enquanto não encontrar um trevo no chão
        Siga em frente
Remova o trevo.
```

Em JavaKara o programa fica:

Este comando estabelece uma estrutura de repetição que pode ser traduzida como "E*nquanto Kara não estiver sobre uma folha mova para frente. Quando encontrar remova-a".*

O programa completo para resolver este problema está mostrado no Código 3-8.

```
while  (!kara.onLeaf( ) ) {
    kara.move( );
}
kara.removeLeaf( );
```

```
public class RemoveFolha extends JavaKaraProgram {
  public void myProgram( ) {
     while  ( !Kara.onLeaf ( ) ) {
       Kara.move( );
     }
     Kara.removeLeaf( );
  }
}
```

Código 3-8 Código em JavaKara para com o comando while.

Com certeza o poder de resolução de problemas de **Kara** aumentou. Concorda? Agora ele sabe executar comandos na sequência, fazer testes (if) e executar processos repetitivos (while).

Modifique um pouco a configuração do mundo e coloque um trevo imediatamente antes do tronco da árvore. Execute o programa. Tudo certo? Por que? Precisa mudar algo no programa?

comando return O

Para fazer com que um método retorne um valor palavra chave **return** é usada. Esta nos permite retornar um valor, que pode ser um tipo primitivo (int, double, boolean...) ou um objeto e ao ser executado o controle sai imediatamente do método.

Observe os exemplos abaixo, o que é retornado em cada um?

- ```
  public  boolean    sobreTrevo( ) {
        if   ( Kara.onLeaf( ) ) {
                    return true
        }
  }
  ```

- ```
  public  int   contaVazio( ) {
   int conta = 0;
   while (!Kara.treeFront) {
         if   (!Kara.onLeaf( ) )  {
                      conta++;
         }
         Kara.move
    }
    return  conta;
  }
  ```

Desafio 18

Repare na situação da Figura 3-9

Figura 3-9 Configuração para o desafio 18.

Faça com que **Kara** realize a operação de soma utilizando os trevos. Observe a Figura 3-9. **Kara** deve percorrer a primeira linha e verificar quantos trevos existem. Para indicar fim da linha temos sempre um tronco de árvore. Em seguida percorrer a segunda linha e contar o número de trevos que encontrar (estes trevos não precisam estar em posições sequenciais). Por fim deverá ir para a próxima linha e colocar tantos trevos quanto for a soma dos trevos encontrados na primeira e segunda linhas.

No Código 3-9 tem-se o programa utilizando dois métodos: **contaLinha** e **colocaResultado**. O método **myProgram** já está escrito, onde existe as mensagens para **Kara** e as chamadas dos métodos criados.

Observe as assinaturas dos métodos contaLinha e colocaResultado. Explique-as.

```
public class SomaTrevos extends JavaKaraProgram {
  public int contaLinha() {
     // colocar aqui o corpo de contaLinha
  }

  public void colocaResultado(int valor) {
     // aqui o corpo de colocaResultado
  }

  public void myProgram() {
    int parcela1,parcela2;
    parcela1 = contaLinha();
    kara.setPosition(0,1);
    parcela2 = contaLinha();
    kara.setPosition(0,2);
    colocaResultado(parcela1+parcela2);
  }
}
```

Código 3-9 Código em JavaKara com as assinaturas e sem o corpo de alguns métodos.

Você percebeu que temos elementos que nos permitem armazenar dados. Até agora vimos **variáveis** e **parâmetros**. Qual a diferença entre estes elementos?

Observe as duas variáveis utilizadas: **parcela1** e **parcela2** ambas do tipo **int**. Quando criamos uma variável estamos reservando um espaço de memória para armazenar dados de um determinado tipo (**int**), sendo acessado por um nome (**parcela1** e **parcela2**). Estas são conhecidas somente dentro deste método (**variáveis locais**).

Os parâmetros servem para fazer a comunicação do método com outros elementos do programa.

No Código 3-9 identifique os parâmetros e as variáveis.

Desenvolva o corpo dos métodos contaLinha e colocaResultado.

Desafio 19

Suponha que **Kara** deva remover uma sequência de 10 trevos. Poderíamos utilizar uma sequência de 10 comandos **Kara.removeLeaf( ); e Kara.move();** . Certo? Mas se for necessário remover 20 trevos?

Use o comando de repetição. Os comando que serão repetidos serão **Kara.removeLeaf( ); Kara.move();**. Pense agora em uma forma permitir que o comando de repetição pare ao atingir o objetivo: remover 10 trevos.

Podemos começar com o seguinte esboço:

```
while (                ) {
  kara.removeLeaf( );
  kara.move( );
}
```

Código 3-10 Código inicial para a repetição.

E agora? Como fazer para que **Kara** remove os 10 trevos?

Conduza **Kara** de modo que ela conte a cada repetição um trevo a mais, caso encontre. Crie uma variável para armazenar este valor como pode ser visto o Código 3-11.

```
NumeroDeTrevos = 0
Enquanto NumeroDeTrevos for menor ou igual a 10
        Remover o trevo
        Aumentar 1 no número de trevos
        Dar um passo a frente
```

Código 3-11 Código em português inicial para a repetição.

Entendido? Nosso processo se inicia fazendo uma **variável** (numeroDeTrevos) receber o valor 0 (zero) e toda vez que se remove um trevo acrescenta-se um ao número de trevos removidos e **Kara** avança uma casa. O processo continua até que o número de trevos seja igual a 10.

Criar uma variável significa estabelecer um endereço de memória que podemos acessar através de seu nome. Sendo de um tipo primitivo (int, double, boolean...), ao declarar uma variável é alocado na memória do computador o espaço adequado para armazenar a variável, com um conteúdo desconhecido (lixo). No nosso caso, como esta variável irá contar o número de trevos ela pode ser do tipo inteiro. A variável é um elemento do programa que pode ter seu valor alterado durante sua execução.

Em JavaKara o programa completo fica como mostrado no Código 3-12.

```
public class RemoveDez  extends  JavaKaraProgram {

  public void myProgram( ) {
     int  numeroDeTrevos = 0;
     while (numeroDeTrevos < 10) {
        Kara.removeLeaf( );
        Kara.move( );
        numeroDeTrevos++;
     }
  }
}
```

Código 3-12 Código em JavaKara.

Que tal melhorar este programa? Faça com que **Kara** remova 10 trevos na mesma linha, independente da posição onde se encontram na linha.

E se ele precisar remover exatamente 10 trevos independente da linha, ou seja, ele percorre toda a linha e se ainda não completou os 10 trevos, muda de linha e assim sucessivamente até chegar ao décimo trevo?

O comando for

Podemos resolver o desafio 19 de outra forma. Existe outro comando para criar um processo repetitivo é o comando **for**. Este comando pode ser utilizado quando se sabe o número de vezes que a repetição ocorrerá. Como no desafio **Kara** precisa remover 10 trevos, podemos utilizá-lo. Sua sintaxe é a seguinte:

Exemplo:
```
for (int i = 0; i < MAXIMO; i++) {
        Kara.removeLeaf();
        Kara.move();
}
```

O exemplo acima pode ser lido como *"Para i começando em 0 (zero), sendo acrescido de um em um até MAXIMO-1 execute o comando move"*. A variável **i** do tipo **int** será usada para controlar o número da repetição. Começando com 0 (zero) e aumentando de 1 em 1, através de i++ (i = i + 1) até alcançar o valor armazenado em MAXIMO, que é uma constante definida no programa. MAXIMO representa uma constante, ou seja, durante a execução do programa seu valor não é alterado. Observe que seu nome está em letras maiúsculas para destaque.

Padrão 4.	As constantes serão escritas com todas as letras maiúsculas.

O programa completo em JavaKara fica como no Código 3-13:

```
public class TesteFor extends JavaKaraProgram {
  public void myProgram( ) {
     final int MAXIMO = 10;
     for (int i = 0; i < MAXIMO; i++) {
        Kara.removeLeaf( );
        Kara.move( );
     }
  }
}
```

Código 3-13 Código com repetição com for.

A palavra reservada final

O que significa final int MAXIMO = 10; ?

Quando um elemento da classe possui esta palavra chave antes de sua definição, **final**, significa, de modo geral, que o elemento não pode ser alterado. Neste caso, trata-se de um elemento local, assim, é uma constante local com valor cujo valor não pode ser alterado.

Podemos ter um campo como **final**. Este pode ter um valor em tempo de compilação e não ser mais alterado, ou receber um valor em tempo de compilação e não mais ser alterado.

Se **final** for usado para um objeto, significa que a referência ao objeto torna-se constante, mas não seu conteúdo.

Mais desafios

Desafio 20

Outro desafio já resolvido foi a parada na entrada do túnel, mostrado na Figura 3-10.

Figura 3-10 Configuração para o desafio 20.

Como resolver? **Kara** tem que mover uma casa a frente enquanto **não encontrar um tronco à esquerda e à direita**. Certo? Como implementar esta tarefa? Desenvolva o programa para este fim. Em português temos:

Enquanto **não encontrar tronco à esquerda e tronco à direita**
    Deslocar uma posição à frente

Desafio 21

Dentro do mundo de **Kara** existem florestas com circuitos formados por onde **Kara** deseja passar para encontrar um trevo. Todo elemento (campo) do circuito tem exatamente dois campos vizinhos vazios. Um deles está atrás, o campo de onde **Kara** vem. Isto é, exatamente uma das três interrogações representa um campo vazio. Os outros dois representam campos com árvores. Ajude **Kara** a chegar ao trevo. O desenho à direita representa com as interrogações um possível local por onde **Kara** poderá passar. Este exercício é o **Getting Start II (easy)**. Sua configuração está na Figura 3-11.

Figura 3-11 Configuração para o desafio 21.

Como solucionar o problema? Seguindo as interrogações acima podemos escrever o código em português no Código 3-14:

```
Se Kara não tem um tronco à esquerda
        Vire à esquerda
        Siga em frente
Senão Se Kara não tem tronco à frente
        Siga em frente
Senão Se Kara não tem um tronco à direita
        Vire à direita
        Siga em frente
```

Código 3-14 Código em português para o desafio 21.

Este processo será repetido enquanto **Kara** não encontrar o trevo e ao encontrar, deve removê-lo.

```
        Enquanto Kara não está sobre um trevo
                Sequência de comandos;
        Remover o trevo
```

O que acontece quando temos uma seqüência de comandos **if** com a opção **else** ? O que é garantido? O que pode acontecer se tirarmos a cláusula **else** (senão) dos comandos acima?

Erros de Sintaxe e de Execução

Desafio 22

Kara precisa remover todos os trevos e parar quando encontrar um cogumelo à frente, como na Figura 3-12. Tenho o seguinte programa e o mundo de Kara :

Figura 3-12 Configuração para o desafio 22.

```
public class Ex1 extends JavaKaraProgram {

  void contornar {
  Kara.turnLeft( );
  Kara.move( );
  Kara.turnRight( );
  Kara.move( );
  Kara.move( );
  Kara.turnRight( );
  Kara.move( );
  Kara.turnLeft( )
 }

 public void myProgram( ) {

  if Kara.treeFront( ) {
    contornar( )
  }
  else {
   if Kara.onLeaf( ) {
     Kara.removeLeaf( )
   }
  }
  Kara.move( );
 }
}
```

Código 3-15 Código em português para o desafio 22.

Este programa funciona? Corrija seus **erros de sintaxe** (escrita dos comandos).

Agora peça que seja executado. Algum erro? Sim. São **erros de execução**. Estes erros são erros na lógica dos comandos que compõe o programa.

Como se detectam os erros de sintaxe? Como corrigí-los?

Como se detectam erros de execução? Como corrigí-los?

Como **Kara** está executando o processo? Por que está fazendo assim? Vamos corrigir por partes. Vá eliminando os erros tomando o devido cuidado para não acrescentar outros.

Compare sua solução com a encontrada no arquivo de soluções 2.

Mais Desafios

Desafio 23

Programe Kara para comer uma "trilha" de folhas. Uma trilha nunca vai além de um tronco, Figura 2. Assim, o programa pode parar assim que **Kara** encontrar-se em frente a um tronco. Este exercício é o **Pacman with cloverleafs (medium)**.

Neste caso **Kara** precisa fazer uma busca pelo próximo trevo. Nesta busca existem 3 possibilidades:

1. o trevo está na posição da frente;
2. na posição à esquerda da posição original;
3. na posição direita da posição original

Como podemos ver na Figura 3-13.

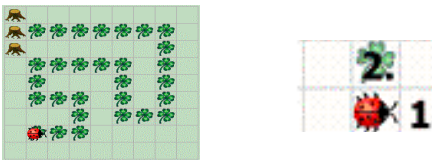

Figura 3-13 Configuração para o desafio 23.

Para esta procura temos o seguinte trecho de programa:

```
Avança uma casa
Se Kara não encontrou um trevo
        Gira 180º
        Avança uma casa
        Gira para a direita
        Avança uma casa
        Se Kara não encontrou um trevo
                Gira 180º
                Avança uma casa
                Avança uma casa
Remove o trevo
```

Código 3-16 Código em português para o desafio 23.

Observe que **Gira 180º** não é um comando conhecido por Kara, é preciso criá-lo. E o processo de procura e remoção do trevo continuará enquanto um tronco não for detectado em frente.

Preste bem atenção nos comandos acima. Você realmente entendeu a solução ao problema? E como ela foi implementada?

Outros desafios se encontram no próprio ambiente, estude-os e ensine Kara a resolvê-los.

Anexo II

Projetos em forma de Jogo de Papéis

I – Acrobatas e Cia

A atividade que será descrita a seguir permitirá que os participantes se comportem como acrobatas. Todos os participantes do jogo são Acrobatas. Todo acrobata sabe aplaudir, girar e contar, como pode ser visto no código da interface abaixo. De maneira formal, em Java, temos a seguinte interface, que especifica o que todos os acrobatas podem fazer. Este jogo foi desenvolvido por Joseph Bergin e adaptado para este curso.

```
interface Acrobata
{
        void aplaudir(int vezes) ;
        void girar(int vezes);
        int contar();
}
```

Os Papéis

Você é membro da classe **AcrobataSimples**
[*Todo AcrobataSimples realiza as operações definidas em Acrobata. Quando você for criado, a você será fornecido um nome. Armazene este dado em seu local apropriado internamente e coloque o valor 0 (zero) em seu contador interno de atividades*]
Quando solicitarem que **aplauda**, a você será dado um número. Aplauda este número de vezes.
Quando solicitarem que **gire**, a você será dado um número. Gire este número de vezes. Observe que se te disserem "2", então você irá girar duas vezes.
Quando solicitarem que **conte**, você irá responder (verbalmente) com o total de exercícios que você fez.
Quando solicitarem **qualNome**, informe seu nome que está armazenado internamente.

Você é um membro da classe **Coreografo**
[*Todo Coreografo é um AcrobataSimples adicionando-se dois outros AcrobataSimples internos para quem repassa as solicitações recebidas. Quando você for criado, faça a criação como a do AcrobataSimples além disso crie os outros dois AcrobataSimples e dê-lhes o nome*]
Quando te solicitarem alguma ação (aplaudir, girar ou contar) você deve passá-la para duas outras pessoas, por exemplo se te solicitarem para aplaudir 3, então você deverá responder com:

Joao, aplauda 3
Célia, aplauda 3 (João e Célia estão na sala)

OBS.: Selecione pessoas da sala. Normalmente outras que estão participando desta atividade. Pode selecionar outras pessoas. Você pode dar a mensagem à mesma pessoa.

Você é um membro da classe **AcrobataComAmigo**
[*Todo AcrobataComAmigo é um AcrobataSimples, ou seja, realiza a solicitação. Em seguida a envia para um outro AcrobataSimples interno. Quando você for criado, faça a criação como a do AcrobataSimples além disso crie o outro, o amigo AcrobataSimples e dê-lhe o nome*]
Quando receber este cartão você deverá selecionar uma pessoa para ser seu amigo.
Quando solicitarem que **aplauda**, a você será dado um número. Aplauda este número de vezes. Passe a mesma instrução para seu amigo.
Quando solicitarem que **gire**, a você será dado um número. Gire este número de vezes. Passe a mesma instrução para seu amigo.
Quando solicitarem que **conte**, você irá responder (verbalmente) com o total de exercícios que você fez. Passe a mesma instrução para seu amigo.

Você é um **Folgado**
[*Todo Folgado é um AcrobataSimples e quando algum serviço é solicitado nada faz*]
Quando solicitarem qualquer instrução (aplaudir, girar ou contar), ignore-a, fique de pé e diga "Me recuso a fazer".
Retorne a seu lugar e sente-se.

Você é um **Autor**
[*Todo Autor quando criado recebe um nome e o armazena internamente*]
Quando solicitarem para assumir este passa a comandar o jogo, ou seja, passa a criar os objetos e solicitar os serviços.

Roteiro do Professor

O roteiro do professor é uma dinâmica que consiste em uma seqüência de criações de objetos e solicitações de serviços. Esta atividade é muito importante, nela o instrutor pode decidir que conceitos abordar primeiro e ajustar os comandos (ou mesmo os papéis) de acordo com a necessidade.

Este roteiro orienta o instrutor sobre como proceder na organização do material e dos alunos. Os alunos que participam do jogo recebe um crachá com sua identificação. Por exemplo, no crachá de um aluno de nome João aparecerá:

AcrobataSimples:joão

Esta identificação corresponde a nomenclatura em UML para o objeto.

Uma listagem dos papéis e participantes deve ser elaborada e mostrada para a turma.

Papel	Nome do aluno	Nome do personagem
AcrobataSimples		acrobata1
AcrobataSimples		acrobata2
AcrobataSimples		acrobata3
Coreógrafo		coreografo1
AcrobataComAmigo		acrobataComAmigo1
AcrobataComAmigo		AcrobataComAmigo2
Folgado		folgado1

O Instrutor deve então ativar os objetos a partir do envio de mensagens a alguns "objetos" que, por sua vez executarão os métodos e/ou encaminharão mensagens a outros "objetos" (os representantes de cada classe). O instrutor deve começar com instruções sintática e semanticamente corretas, e introduzir erros (torções) que podem conduzir a resposta "Eu não sei como fazer isto." Devido a um erro semântico e até erros sintáticos, como por exemplo pela falta de um parâmetro.

Uma possível seqüência de ações é mostrada abaixo. A coluna da direita é preenchida com o código e a criação do diagrama do objeto criado.

	Ação solicitada	Código correspodente em Java
1	"construção de acrobata1, chamado "Augusto"	Acrobata acrobata1 = new AcrobataSimples("Augusto");
2	"acrobata1, aplauda 3"	acrobata1.aplauda(3);
3	"acrobata1, gire 2"	acrobata1.gire(2);
4	"acrobata1, ajoelhe"	acrobata1.ajoelhe(); → Erro!
5	"acrobata1, conte"	int vezes = acrobata1.conte();
6	"construção de acrobata2, chamado "Ana"	Acrobata acrobata2 = new AcrobataSimples("Ana");
7	"acrobata2, gire 1"	acrobata2.gire(1);
8	"acrobata2, aplauda 5"	acrobata2.aplauda(5);
9	"acrobata2, conte"	int vezes = acrobata2.conte();
10	"construção de acrobata3, chamado "Pedro Henrique"	Acrobata acrobata3 = new AcrobataSimples("Pedro Henrique");
11	"acrobata3, aplauda"	acrobata3.aplauda(); → Erro!
12	"Acrobata, aplauda 10"	Acrobata.aplauda(10);
13	"acrobata1, gire 5"	acrobata1.gire(5);
14	"construção de coreografo1, chamado "Vinícius"	Coreografo coreografo1 = new Coreografo ("Vinícius"); // os dois acróbatas internos se chamam "Gabriela" e "Carolina".
15	"coreografo1, aplauda 3"	coreografo1.aplauda(3);
16	"coreografo1, conte"	int vezes = coreografo1.conte();
17	"construção de acrobataComAmigo1, chamado "Vitória"	AcrobataComAmigo acrobataComAmigo1 = new AcrobataComAmigo ("Vitória");
18	"acrobataComAmigo1, gire 3"	acrobataComAmigo1.gire(3);
19	"construção de folgado1, chamado "Rafael"	Folgado folgado1 = new Folgado ("Rafael");
20	"folgado1, aplauda 1"	Folgado1.aplauda(3);
21	"construção de autor1, chamado "Gabriel"	Autor autor1 = new Autor ("Gabriel");
22	"autor1, agir"	autor1.agir();

Cada participante do jogo recebe uma ficha contendo locais para serem preenchidos com seus dados internos. Esta ficha fica à vista de toda a turma para que todos percebam as alterações que ocorrem durante o jogo.

II – Figuras Geométricas

Neste outro projeto existem papéis semelhantes. O primeiro é o de um **Circulo** e o segundo o de um **Alvo**. O primeiro implementa as funções de um círculo: aparecer, desaparecer, desenhar, mudar a cor, mudar o tamanho, mudar a posição em relação ao eixo X, mudar a posição em relação ao eixo Y, em relação ao eixo Y e mudar o diâmetro. O segundo se comporta de forma semelhante, porém coordenando cinco círculos que formam o alvo de um jogo de tiro ao alvo.

Considere o papel de **Circulo**. Todo objeto deste tipo gerencia círculos através das funções especificadas e armazena informações (campos): diâmetro, posiçãoX, posiçãoY, cor e eVisivel.

O outro papel é o de **Alvo**. Este papel possui a mesma funcionalidade do **Circulo**, porém, quando recebe uma mensagem a repassa para os círculos que compõe o alvo, este é como se fosse um **coreógrafo**. Assim, quando recebe a mensagem *alvo1.aparecer()* este a repassa para cada um dos círculos que o compõe, fazendo cada um aparecer. As ações são redefinidas em **Alvo**, este processo chama-se **sobrescrição**.

Um outro elemento deste projeto é o da interface **FiguraGeometrica**. A interface especifica as funcionalidades que qualquer figura geométrica deve ter, neste contexto. Nenhuma instância de figura geométrica pode ser criada, uma vez que **FiguraGeometrica** é uma interface, mas sim uma figura específica como círculo e outras que veremos em projetos posteriores que implementam esta interface.

Outro jogo de papéis

A atividade abaixo tem o objetivo de introduzir alguns conceitos básicos de programação orientada a objetos. Participando do jogo o aluno terá contato com os conceitos de interface, classes, herança, objetos, mensagens, parâmetros, campos, métodos, tipos primitivos e erros.
A descrição desta interface em Java é mostrada abaixo.

```
interface FiguraGeometrica
{
   void aparecer();
   void desaparecer();
   void mudarCor(String novaCor);
   void desenhar();
   void apagar();
   void mudarPosicaoX(int novaPosicaoX);
   void mudarPosicaoY(int novaPosicaoY);
   void mudarTamanho(int novoTamanho);
   int   qualDiametro();
   int   qualPosicaoX();
   int   qualPosicaoY();
}
```

Os Papéis

Você é um **Circulo**

[Seu papel é implementar as ações disponibilizadas para Figuras Geométricas, permitindo gerenciar círculos].

Quando se cria um novo circulo você receberá quatro informações: diâmetro, podivsoX, posicaoY e cor. Armazene-as em local particular adequado.

- diametro = diâmetro fornecido
- posicaoX = posição X fornecida
- posiçãoY = posição Y fornecida
- cor = cor fornecida
- eVisível = falso

Quando se cria um novo círculo sem nada receber como parâmetro construa um círculo com posição e tamanhos padrões e não visível. Padrões: diâmetro = 30, posiçãoX =20, posiçãoY = 20 e cor = "blue".

Quando solicitarem para **aparecer** registre true em **eVisivel** e execute a ação de **desenhar** no quadro.

Quando solicitarem para **desaparecer** execute a ação de **apagar** do quadro e registre false em **eVisivel**.

Quando solicitarem para **desenhar** se em **eVisivel** está true então se desenhe no quadro considerando as informações armazenadas.

Quando solicitarem para **apagar** se em **eVisivel** está true então se apague.

Quando solicitarem para **mudarCor** a você será fornecido o nome de uma cor, armazene esta cor em **cor** e execute a ação de **desenhar**.

Quando solicitarem para **mudarTamanho** a você será fornecido um valor para o novo diâmetro, execute a ação de **apagar** o círculo atual, armazenar em **diâmetro** o valor fornecido e executar a ação de **desenhar**.

Quando solicitarem para **informarCor** devolva o nome da cor que está armazenada em **cor**.

Quando solicitarem para **informarPosicaoX** devolva a posição relativa ao eixo X que está armazenada em **posiçãoX**.

Quando solicitarem para **informarPosicaoY** devolva a posição relativa ao eixo Y que está armazenada em **posiçãoY**.

Quando solicitarem para **mudarPosicaoX** a você será informado uma nova posicaoX, o desenho atual deverá ser apagado, a **posicaoX** armazenada deverá ser alterada para este novo valor e deve-se desenhar o novo círculo.

Quando solicitarem para **mudarPosicaoY** a você será informado uma nova posicaoY, o desenho atual deverá ser apagado, a **posicaoY** armazenada deverá ser alterada para este novo valor e deve-se desenhar o novo círculo.

Quando solicitarem para **informarDiametro** você deverá informar o valor que está armazenado em **diametro**.

Você é um **Alvo**
[Você também implementa as operações de FiguraGeometrica e é formado por cinco círculos concêntricos, porém quando receber alguma solicitação irá transferí-la a outros cinco círculos concêntricos que compõe o alvo]

Quando se cria um novo **Alvo**, são criados cinco círculos com as seguintes informações:
pontos10 = Círculo com diâmetro 100 em X = 20, Y = 20 e cor “blue”
pontos20 = Círculo com diâmetro 80 em X = 30, Y = 30 e cor “yellow”
pontos30 = Círculo com diâmetro 60 em X = 40, Y = 40 e cor “green”
pontos40 = Círculo com diâmetro 40 em X = 50, Y = 50 e cor “white”
pontos50 = Círculo com diâmetro 20 em X = 60, Y = 60 e cor “red”

Quando solicitarem para **aparecer** faça o seguinte:
- Solicite a pontos10 para **aparecer**
- Solicite a pontos20 para **aparecer**
- Solicite a pontos30 para **aparecer**
- Solicite a pontos40 para **aparecer**
- Solicite a pontos50 para **aparecer**

Quando solicitarem para **desaparecer** faça o seguinte
- Solicite a pontos10 para **desaparecer**
- Solicite a pontos20 para **desaparecer**
- Solicite a pontos30 para **desaparecer**
- Solicite a pontos40 para **desaparecer**
- Solicite a pontos50 para **desaparecer**

Quando solicitarem para **mudarCor** será informada uma cor faça o seguinte
- Solicite a pontos10 para **mudarCor** para a cor informada
- Solicite a pontos20 para **mudarCor** para a cor informada
- Solicite a pontos30 para **mudarCor** para a cor informada
- Solicite a pontos40 para **mudarCor** para a cor informada
- Solicite a pontos50 para **mudarCor** para a cor informada

Quando solicitarem para **desenhar** faça o seguinte
- Solicite a pontos10 para **desenhar**
- Solicite a pontos20 para **desenhar**
- Solicite a pontos30 para **desenhar**
- Solicite a pontos40 para **desenhar**
- Solicite a pontos50 para **desenhar**

Quando solicitarem para **apagar** faça o seguinte
- Solicite a pontos10 para **apagar**
- Solicite a pontos20 para **apagar**
- Solicite a pontos30 para **apagar**
- Solicite a pontos40 para **apagar**
- Solicite a pontos50 para **apagar**

Quando solicitarem para **colorir** cada **Circulo** restabeleça a cor original
- Solicite a pontos10 para **mudarCor** para “blue”
- Solicite a pontos20 para **mudarCor** para “yellow”
- Solicite a pontos30 para **mudarCor** para “green”
- Solicite a pontos40 para **mudarCor** para “white”
- Solicite a pontos50 para **mudarCor** para “red”

Quando solicitarem para **mudarTamanho** será informado um novo tamanho, cada **Circulo** terá seu tamanho original acrescido deste novo tamanho
- Solicite a pontos10 para **mudarTamanho** tamanho+novo tamanho
- Solicite a pontos20 para **mudarTamanho** tamanho+novo tamanho
- Solicite a pontos30 para **mudarTamanho** tamanho+novo tamanho
- Solicite a pontos40 para **mudarTamanho** tamanho+novo tamanho
- Solicite a pontos50 para **mudarTamanho** tamanho+novo tamanho

Quando solicitarem para **mudarPosicaoX** será informado uma nova posição, cada **Circulo** terá sua **posicaoX** acrescida desta nova posição
- Solicite a pontos10 para **mudarPosicaoX** posicaoX+nova posição
- Solicite a pontos20 para **mudarPosicaoX** posicaoX+nova posição
- Solicite a pontos30 para **mudarPosicaoX** posicaoX+nova posição
- Solicite a pontos40 para **mudarPosicaoX** posicaoX+nova posição
- Solicite a pontos50 para **mudarPosicaoX** posicaoX+nova posição

Quando solicitarem para **mudarPosicaoY** será informado uma nova posição, cada **Circulo** terá sua **posicaoY** acrescida desta nova posição
- Solicite a pontos10 para **mudarPosicaoY** posicaoY+nova posição
- Solicite a pontos20 para **mudarPosicaoY** posicaoY+nova posição
- Solicite a pontos30 para **mudarPosicaoY** posicaoY+nova posição

Solicite a pontos40 para **mudarPosicaoY** posicaoY+nova posição
Solicite a pontos50 para **mudarPosicaoY** posicaoY+nova posição

Quando solicitarem **qualDiametro** informe o diâmetro do maior círculo
Quando solicitarem **qualPosicaoX** informe a posiçãoX do maior círculo
Quando solicitarem **qualPosicaoY** informe a posicaoY do maior círculo

Roteiro para o instrutor

Como em qualquer jogo de papéis, o interrogatório é a parte mais importante. Assim, o instrutor pode decidir que conceitos abordar primeiro e ajustar os comandos (ou mesmo os papéis).

Atribua papéis e liste-os no quadro:

Papel	identificação	Aluno
Circulo	Circulo1	
Circulo	Circulo2	
Circulo	Circulo3	
Alvo	alvo1	

Nota: Cada uma das solicitações abaixo está a princípio corretas. Solicitações desconhecidas pelo objeto devem ter a mensagem "Eu não sei como fazer isto" como retorno.

O que deve ser feito	Comando
"construir objeto circulo1 com diâmetro 50, posição X = 20, posição Y = 20, cor = "blue" e eVisivel = false".	
"circulo1, apareça".	
"circulo1, mude sua cor para "green" ".	
"circulo1, duplique-se".	
"circulo1, informe a sua posição X".	
"fabricação de circulo2".	
"circulo2, apareça".	
"circulo2, mudar posição X para 50".	
"circulo2, movimente-se na horizontal".	
"circulo2, desapareça".	
"Executar fabricação de alvo1".	
"alvo1, apareça".	
"alvo1, mudar posição X em 20".	
"alvo1, mudar posição Y em -30".	
"alvo1, mudar cor para "red".	
"alvo1, colorir".	
"alvo1, informe seu diâmetro"	

Esta seqüência de comandos (criação de objetos e solicitações de serviços) compõe um programa que pode ser executado. Este precisa ser descrito utilizando-se uma linguagem de programação, no nosso caso Java.

Em seguida este é compilado. Na compilação os erros (de escrita de comandos e pontuação) são detectados e devem ser corrigidos. Depois de corrigidos gera-se um arquivo contendo um formato especial de codificação (bytecodes) que pode ser interpretado pela máquina Java e assim executado pelo computador. Mesmo nesta execução podem aparecer erros, os erros de execução (ou intenção), o programa funciona, mas não como planejado. Estes comandos ficam em local especial dentro do programa que será visto posteriormente.

III – Jogo de Perguntas e Respostas

Este projeto tem como objetivo mostrar as possibilidades do mundo da orientação por objetos. Explorando coleções.

Descrição

Este jogo envolve os alunos da turma agrupados em conjuntos com um tamanho determinado pelo coordenador do jogo. Cada aluno tem uma função específica como Aluno, Contador, BaseDePergunta, Turma, Coordenador entre outros. O coordenador irá formar a turma, formar os grupos de alunos e iniciar o jogo. A partir daí o jogo flui através da ação de perguntar e responder desenvolvida pelos próprios alunos participantes dos grupos. O coordenador sorteia o primeiro aluno a lançar uma pergunta. Este a faz e a direciona a grupo e aluno específico. Se o aluno acertar a resposta, o que será verificado pelo coordenador, ponto para o grupo do aluno que respondeu corretamente, caso contrário ponto para o grupo do aluno que formulou a pergunta. O jogo continua, com o aluno ao qual a pergunta foi dirigida. Este faz a próxima pergunta independente de ter tido sucesso ou não em sua participação. O jogo continua até que não existam mais perguntas. Ao final o coordenador emite um relatório com a pontuação de cada grupo e com o grupo campeão. A base de perguntas e respostas é montada pelos próprios alunos. Cada aluno pode cadastrar quantas perguntas desejar. Sendo que todas vão para uma mesma base de perguntas. Cada participante tratará a pergunta e a resposta que adicionar como sigilo. Existem três tipos de alunos: aluno solitário (que responde e não aceita ajuda de ninguém), aluno com colega (que analisa a resposta e se desejar a repassa para o colega) e aluno acomodado (que se recusa a responder).

Papéis

Aluno
Todo indivíduo deste tipo contém uma informação interna que representa seu **nome**. Na prática, o indivíduo do tipo Aluno não existe, mas sim um tipo especial que pode ser: AlunoSolitário, AlunoComColega ou AlunoAcomodado. • Quando solicitarem para **formularUmaPergunta** retorne uma String contendo a pergunta. • Quando solicitarem para **formularUmaResposta** à pergunta retorne uma String contendo a resposta. • Quando solicitarem para **perguntar** a você será fornecido uma **base de perguntas** e uma **turma em grupos**. Selecione uma pergunta para fazer, retire-a da base de perguntas e direcione-a a um dos grupos disponíveis e a um aluno do grupo selecionado. Faça a pergunta, por exemplo, *responder(3,2,1);* que significa que você selecionou a pergunta 3 e esta direcionando-a ao aluno 1 do grupo 2. Para finalizar retorne a nova base de perguntas. • Quando solicitarem **qualNome** informe seu nome. • Quando solicitarem para **alterarNome** a você será fornecido um novo nome, armazene-o em **nome**, o que irá substituir o nome anterior. • Quando solicitarem para **responder** responda de acordo com o que está definido de acordo com sua própria classe.
AlunoSolitário
Todo indivíduo deste tipo é um tipo especial de Aluno, isto é, tem as características de aluno e executa todas as ações que o Aluno pode executar. Além disso, é responsável por responder perguntas que lhe são dirigidas. • Quando você for **criado**, a você será passada uma informação, seu próprio nome, armazene-o em local interno adequado. • Quando solicitarem para **responder** uma pergunta, faça-o sozinho, isto é, você não poderá contar com a ajuda de qualquer outra pessoa. A você será fornecido um **itemDePergunta**. Solicite a este itemDePergunta para te **mostrar** a pergunta e emita sua resposta retornando-a.
AlunoComColega
Todo indivíduo deste tipo é um tipo especial de Aluno, isto é, tem as características de aluno e executa todas as ações que o Aluno pode executar. Além disso, é responsável por responder perguntas que lhe são dirigidas. • Quando você for **criado**, a você serão passadas duas informações, seu próprio nome e um outro aluno, a quem poderá recorrer para responder perguntas. Armazene estas informações em local adequado. • Quando solicitarem para **responder** a uma pergunta você pode respondê-la, ou pode repassá-la ao aluno que o acompanha.
AlunoAcomodado
Todo indivíduo deste tipo é um tipo especial de Aluno, isto é, tem as características de aluno e executa todas as ações que o Aluno pode executar. Além disso, é responsável por responder perguntas que lhe são dirigidas. • Quando você for **criado**, a você será passada uma informação, seu próprio nome, armazene-o em local interno adequado. • Quando lhe solicitarem a responder simplesmente diga "Me recuso!".
Turma
Toda turma é composta por elementos do tipo Aluno, ou seja, é uma coleção de **alunos** que pode conter qualquer tipo de aluno: AlunoSolitário, AlunoComColega e AlunoAcomodado. • Quando você for criado, crie uma lista vazia. • Quando solicitarem para adicionarAluno, a você será fornecido um aluno. Adicione-o à coleção de alunos. • Quando solicitarem **quantosAlunos** retorne o número de elementos na lista. • Quando solicitarem **quaiAlunos** retorne uma String contendo uma lista dos nomes dos alunos.
Grupo
Todo grupo de alunos possui uma coleção de **participantes** do tipo Aluno e um **contador** de pontos do grupo. • Quando você for criado crie uma lista de participantes vazia e crie um contador para m arcar o número de pontos do grupo. • Quando solicitarem para **adicionarAluno** a você será fornecido um aluno. Insira-o na lista de participantes.
TurmaEmGrupo
Toda turma em grupo contém uma coleção de grupos de alunos. • Quando você for criado crie uma coleção vazia. • Quando solicitarem para **adicionarGrupo** a você será fornecido uma **turma** e o número de alunos por grupo. Crie este grupo e adicione-o à sua coleção. • Quando solicitarem para **informarGrupos** retorne uma String contendo todos os grupos formados. • Quando solicitirarem **quaisParticipantes** a você será fornecido o número do grupo. Retorne a coleção de alunos do grupo especificado.

ItemDePergunta

O tipo ItemDePergunta apenas define a ação que elementos deste tipo podem executar, não especificando os detalhes do comportamento. Todo tipo que o implementar terá que especificar no mínumo estas ações.

- A ação **retornarPergunta** retorna um elemento do tipo String.
- A ação **retornarResposta** retorna um elemento do tipo String.

ItemReal

ItemReal detalha as ações especificadas em ItemDePergunta. Todo ItemReal contém internamente uma **pergunta** e uma respectiva **resposta**.

- Quando você for criado a você será fornecido uma String que contém a pergunta e uma String contendo a resposta. Armazene-as internamente nos locais adequados.
- Quando solicitarem para **retornarPergunta** você deverá retornar a pergunta.
- Quando solicitarem para **retornarResposta** você deverá retornar a resposta.

ItemNulo

ItemNulo detalha as ações especifiacadas em ItemDePergunta. Todo item nulo contém internamente uma **pergunta** e uma **respota** em branco. Em situações onde um item de pergunta é esperado, mas onde não é possível retorná-lo, utiliza-se este tipo de objeto.

- Quando você for criado você armazenará o String vazio na pergunta e na resposta.
- Quando solicitarem para **retornarPergunta** você deverá retornar a pergunta.
- Quando solicitarem para **retornarResposta** você deverá retornar a resposta.

BaseDePergunta

Toda base de pergunta contém uma coleção de elementos do tipo ItemDePergunta.

- Quando você for criado crie uma coleção vazia.
- Quando solicitarem para **adicionarItem** a você será fornecido um ItemDePergunta. Armazene-o em sua coleção interna.
- Quando solicitarem **quantasPerguntas** informe o número de perguntas da coleção.
- Quando solicitarem para **retornarUmItemAleatório** sorteie um número entre as perguntas existentes e retorne o item de pergunta da posição sorteada.
- Quando solicitarem para **listarPerguntasDaBase** retorne uma String contendo todas as perguntas da base.

Controlador

O controlador é único, isto é, não existe apenas um no jogo. O elemento deste tipo irá gerenciar o jogo desde o controle do tempo de no máximo uma hora ou até terminar as perguntas até a formação da turma, dos grupos e da base de pergunta.

- Ao ser criado o item do tipo Controlador deixa tudo pronto para ser usado.
  - Cria o cronômetro informando o tempo máximo da partida.
  - Cria a turma como uma coleção vazia.
  - Cria a turma em grupos como uma coleção vazia.
  - Cria a base de perguntas como uma coleção vazia.
- Quando solicitarem para **criarTurma** solicite o nome de cada um dos alunos e execute a ação de **adicionarAlunoNaTurma**.
  - Quando solicitarem para **adicionarAlunoNaTurma** a você será informado o nome do aluno, crie o aluno com este nome e adicione-o na coleção **turma**.
- Quando solicitarem para **montarBaseDePerguntas** solicite que os alunos coloquem as perguntas e respostas em uma folha de papel e as entregue em sigilo.
  - Em seguida adicione-as à base de perguntas uma a uma de forma aleatória e sigilosa.
  - Em seguida exiba o número de perguntas da base, solicitando à base que informe o número de itens cadastrados através de **quantasPerguntas**.
- Quando solicitarem para **montarGrupos** a você será fornecido o número de alunos por grupo.
  - Em primeiro lugar execute **modificarOrdemDosAlunos** para que eles não fiquem na ordem de entrada.
  - Em seguida crie os grupos com os alunos da turma.
  - Finalmente exiba a formação dos grupos, solicitando ao grupo **informarParticipantesDoGrupo**.
- Quando solicitarem **iniciarJogo** execute:
  - **disparar** o cronômetro. Neste momento solicite ao cronômetro para **disparar** o tempo.
  - **sortearUmGrupo** retornando um número inteiro correspondente a um dos grupos existentes.
  - **sortearUmParticipante** retornando um número inteiro correspondente a um dos participantes do grupo já sorteado.
  - Solicite ao participante sorteado do grupo específico para **peguntar** fornecendo a ele a base de perguntas e a turma em grupos.
  - Em seguida execute **checarResposta**, considerando o grupo que fez a pergunta o grupo que respondeu, a pergunta e a resposta. Após análise execute a ação de **registrarPonto** ao grupo de direito, ou seja, ao grupo que respondeu corretamente ou ao grupo que perguntou, caso a resposta esteja errada.
  - O aluno que respodeu à pergunta executa a operação de **perguntar** novamente dando sequência ao jogo.
- Quando solicitarem para **apurarResultadoFinal** verifique o contador de pontos de cada grupo e informe o resultado apurado, exibindo os grupos na ordem decrescente de pontuação.

Contador

Todo contador armazena internamente um **valor** inteiro que representa uma contagem.

- Quando você for criado zere seu valor interno.
- Quando solicitarem para **reiniciar** reinicie o seu valor interno com o valor zero.
- Quando solicitarem **qualValor** retorne o seu valor interno.
- Quando solicitarem para **contar** acrescente um ao seu valor interno.

Cronometro

Todo cronometro é um contador porém com a capacidade de parar e avisar quando atinge o valor máximo. Além de ter o valor da contagem possui internamente um valor máximo.

- Quando você for criado a você será fornecido o valor máximo. Sere seu valor interno e coloque em valor máximo o valor recebido.
- Quando solicitarem **disparar** comece a marcar o tempo em segundos em seu valor interno.
- Quando o valor interno atingir o valor máximo automaticamente soe o alarme.

Relação dos papéis desempenhados

Cada aluno deverá desempenhar um papel específico entre os listados na sessão anterior.

- Devem existir vários AlunoSolitario, AlunoComColega e AlunoAcomodado.
- Apenas um elemento do tipo Turma.
- Um do tipo TurmaEmGrupo.
- Um aluno representando cada Grupo.
- Um cronômetro que marca o tempo e soa o alarme quando o tempo esgotar.
- Um contador para cada grupo que funcionará como marcador de tempo.
- Apenas um controlador que coordena o jogo.
- Os itens de pergunta são elementos que serão utilizados de forma diferente. Estes serão escritos em papel e entregue ao controlador para que os insira na base de perguntas.
- Um aluno representa a base de pergunta.

A tabela abaixo deverá ser preenchida para estabelecer os papéis entre os participantes.

Nome do aluno	Papel
	AlunoSolitario
	AlunoSolitario
	AlunoSolitario
	AlunoComColega
	AlunoComColega
	AlunoAcomodado
	AlunoAcomodado
	Turma
	TurmaEmGrupo
	Grupo
	Grupo
	Contador
	Contador
	Cronometro
	Controlador
	Base de pergunta