Maria Janaina da Silva Ferreira

EMS - Um plug-in para exibição das mensagens de erro dos compiladores

Sorocaba, SP

18 de Dezembro de 2015

Maria Janaina da Silva Ferreira

EMS - Um plug-in para exibição das mensagens de erro dos compiladores

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCCS) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Área de concentração: Sistemas Computacionais.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCCS

Orientador: Prof. Dr. José de Oliveira Guimarães

Sorocaba, SP

18 de Dezembro de 2015

Maria Janaina da Silva Ferreira
EMS - Um plug-in para exibição das mensagens de erro dos compiladores/ Maria Janaina da Silva Ferreira. – Sorocaba, SP, 18 de Dezembro de 2015-
123 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. José de Oliveira Guimarães

Dissertação (Mestrado) – Universidade Federal de São Carlos – UFSCar
Centro de Ciências em Gestão e Tecnologia – CCGT
Programa de Pós-Graduação em Ciência da Computação – PPGCCS, 18 de Dezembro de 2015.

1. Compiladores. 2. Mensagens de Erro. I. Prof. Dr. José de Oliveira Guimarães. II. Universidade Federal de São Carlos. III. Centro de Ciências em Gestão e Tecnologia – CCGT. IV. EMS - Um plug-in para exibição das mensagens de erro dos compiladores

CDU 02:141:005.7

Maria Janaina da Silva Ferreira

EMS - Um plug-in para exibição das mensagens de erro dos compiladores

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCCS) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Área de concentração: Sistemas Computacionais.

Trabalho aprovado. Sorocaba, SP, 18 de Dezembro de 2015:

Prof. Dr. José de Oliveira Guimarães
Orientador

Prof. Dr. João José Neto
Convidado 1

Profa. Dra. Tiemi Christine Sakata
Convidado 2

Sorocaba, SP
18 de Dezembro de 2015

Dedico este trabalho à minha família. Em memória à minha mãe e minha irmã, que sempre me apoiaram em todos os momentos e decisões. Ao meu pai e meu irmão pela paciência, a minha sobrinha pelo amor incondicional e a todos que fizeram parte da minha vida e me apoiaram em minhas decisões.

Agradecimentos

Agradeço,

Primeiramente à Deus pela presença constante em minha vida. Pelas oportunidades que me concedeu, por me dar consciência para compreender que nem tudo que eu quero está nos seus planos, mas que os seus planos são sempre os melhores pra mim.

À minha família pela paciência em todos os momentos que não pude de alguma maneira participar da vida deles. Em memória da minha mãe e minha irmã, que me incentivaram desde o começo. Ao meu pai por suportar o meu silêncio, ao meu irmão por não poder participar da sua vida, a minha sobrinha por estar sempre sorrindo pra mim, mesmo nos momentos mais difíceis.

Ao meu orientador Prof. Dr. José de Oliveira Guimarães, pelos ensinamentos, tempo, dedicação, sinceridade, competência, profissionalismo e incentivo a mim disponibilizado nesses anos. E principalmente, por acreditar que era capaz. Muito obrigado.

A todos os professores que de alguma maneira participaram da minha formação.

Aos professores da UFSCar: Profa. Dra. Tiemi Christine Sakata, Prof. Dra. Luciana Zaina, Prof. Dr. Murillo Rodrigo Petrucelli Homem, Prof. Dr. Alexandre Alvaro e Profa Dra. Katti Faceli pelas disciplinas ministradas.

Aos membros da banca examinadora da qualificação, Profa Dra. Luciana Zaina e a e Profa. Dra. Tiemi Christine Sakata, pelos comentários e contribuições, que ajudaram na melhoria do meu projeto.

A ETEC Fernando Prestes, especialmente ao Prof. Anderson Amaral pela amizade, por todo apoio e por disponibilizar a escola para a realização dos experimentos que validaram este trabalho.

As meus colegas de trabalho pela amizade e pelo incentivo.

Aos meus alunos, pela participação na validação desta pesquisa.

Ao Capes por todo apoio financeiro recebido.

E a todos aqueles que estiveram presentes em minha vida.

As palavras não são capazes de descrever a minha gratidão a vocês. Que Deus esteja sempre com vocês e acima de tudo, que vocês estejam sempre com Deus. Obrigada.

Lembrar que estarei morto em breve, é a ferramenta mais importante que já encontrei para me ajudar a tomar grandes decisões. Porque quase tudo: expectativas externas, orgulho, medo de passar vergonha ou falhar, caem diante da morte, deixando apenas o que é importante. Lembrar que você vai morrer é a melhor maneira que eu conheço para evitar a armadilha de pensar que você tem algo a perder. Você já está nu. Não há razão para não seguir seu coração."

(Steve Jobs)

Resumo

As mensagens de erro dos compiladores devem permitir que os programadores compreendam e solucionem os problemas encontrados durante o processo de compilação rapidamente. Entretanto, os compiladores usualmente emitem mensagens curtas, sem contexto, pouco informativas e com termos de difícil compreensão. Este trabalho apresenta o *plug-in* Error Message System (EMS) que permite a apresentação das mensagens de erro mais fáceis de entender e mais informativas. EMS é um plugin para a IDE Eclipe e é altamente configurável através de linguagens específicas de domínio(LED). As LEDs permitem que usuários comuns façam suas próprias mensagens de erro e as compartilhem. Programadores iniciantes podem utilizar um conjunto de mensagens adaptadas a eles, reduzindo o tempo de compreensão e correção dos erros de compilação.

Palavras-chaves: Compiladores. Mensagens de erro. Interação-Humano Computador. Linguagens Específicas de domínio.

Abstract

Compiler error messages should allow programmers to understand and solve quickly problems found during the compilation process. However, compilers usually issue short contextless messages with little information and with terms that are difficult to understand. This work introduces the plug-in Error Message System (EMS) that allows the presentation of easy-to-understand and more meaningful error messages. EMS is a plug-in to the Eclipse IDE. It is highly configurable through Domain Specific Languages (DSLs). The DSLs allow that regular users build their own error messages and share them. Beginner programmers can use a set of error messages adapted to them thus reducing the time of understanding and correction of compilation errors.

Key-words: Compilers. Error Messages. Human Computer Interaction. Domain Specific Languages.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

CSS	Cascading Style Sheets
DSL	Domain Specific Language
EMS	Error Message System
ETEC	Escola Técnica Estadual
EOF	End of file
GUI	Graphical User Interface
IDE	Integrated Development Environment
IHC	Interação Humano Computador
HTML	HyperText Markup Language
LED	Linguagem Específica de Domínio
SQL	Structured Query Language
UI	User Interface
XML	Extensible Markup Language

Sumário

1 Introdução

O processo de desenvolvimento de software é minucioso e demanda muito tempo a fim de se obter um produto final de qualidade (PRESSMAN, 2001). Durante o processo de desenvolvimento, as mensagens exibidas pelos compiladores são de extrema importância para entendimento e solução dos erros encontrados durante o processo de compilação.

A forma como a mensagem é exibida afeta de maneira significativa o aprendizado, o desempenho e a interpretação do programador. Em particular dos programadores iniciantes e estudantes de computação (COULL, 2008) (NIENALTOWSKI; PEDRONI; MEYER, 2008) (FLOWERS; CARVER; JACKSON, 2004) que apresentam problemas na compreensão dos conceitos abstratos, por vezes relacionando com dificuldades matemáticas e lógicas (GOMES; MENDES, 2007), a forma como esses conceitos são abordados, ferramentas de desenvolvimento e a sintaxe da linguagem de programação (LAHTINEN; ALA-MUTKA; JÄRVINEN, 2005)(SHACKELFORD; BADRE, 1993).

Muito se tem feito para apoiar os programadores iniciantes na programação. Podemos citar, por exemplo, a criação de ferramentas de software amigáveis e visuais que visam proporcionar um ambiente mais favorável e diminuir a complexidade da programação, facilitando assim o processo de desenvolvimento, edição, compilação e exibição de programas de computação.

Entretanto, pouco se tem trabalhado para melhorar as mensagens de erros dos compiladores, usualmente exibidas com termos difíceis que prejudicam a compreensão, resolução e prevenção do erro ocorrido. Dessa forma, torna-se necessário uma solução que permita a melhor interpretação dos erros e consequentemente um maior aumento na produtividade e qualidade de desenvolvimento de softwares.

Esse trabalho apresenta o *plug-in* EMS (Error Message System) que foi acoplado ao compilador da linguagem Cyan (GUIMARÃES, 2015). Usando o EMS as mensagens de erro são exibidas de forma amigável utilizando aparentemente o próprio código do usuário que causou o erro (mostrando o erro e o código correto, quando possível). O *plug-in* permite que o programador armazene as suas próprias explicações e comentários para cada erro. Para configuração da ferramenta várias linguagens específicas de domínio são utilizadas.

1.1 Contexto

A tarefa de desenvolvimento de software exige muita concentração e raciocínio e, o esforço para interpretar o que o compilador quer dizer torna o aprendizado mais lento e

cansativo.

Os compiladores são projetados para obedecer as regras da linguagem e aplicá-las com o objetivo de produzir código executável. Usualmente não há nenhum empenho em tornar o compilador adaptável ao usuário.

A primeira evidência dos problemas com as mensagens de erro foi descrita por Moulton e Miller em 1967 (MOULTON; MULLER, 1967), que originou a ferramenta de diagnóstico *Ditran*. Para os autores a preocupação dos programadores iniciantes e estudantes de programação em encontrar e corrigir erros em seus programas o mais rapidamente possível supera a necessidade de um código eficiente e sofisticados recursos de programação. Em 1983 Brown (BROWN, 1983) evidência novamente o problema, focando na qualidade das mensagens exibidas e seu impacto nos programadores. Até então, poucas pesquisas foram realizadas a fim de resolver o problema ou determinar um padrão para melhorar a forma de exibição das mensagens de erro (SHNEIDERMAN, 1986) (BROWN, 1983) (SCHORSCH, 1995) (ALEXANDRESCU, 1999). Pesquisas atuais focam na disciplina de Interação Humano Computador (IHC) (TRAVER, 2006) (TRAVER, 2010), apresentando um modelo para a exibição das mensagens. Elas partem do pressuposto que a forma como as mensagens são exibidas afetam diretamente o aprendizado e o processo de desenvolvimento do software.

É importante ressaltar que, durante o processo de compilação, os erros encontrados são reportados. Porém os compiladores não possuem ajuda adicional além da mensagem exibida, o que induz os programadores a procurarem formas alternativas de auxílio, em fóruns, grupos ou redes sociais. Essa busca faz com que sua dúvida não seja solucionada rapidamente, levando o programador a usar uma solução muitas vezes não adequada. Até mesmo os compiladores modernos focam na premissa que o programador tem que se adaptar ao compilador.

Um dos objetivos dessa pesquisa é o desenvolvimento de uma ferramenta que permita que o compilador se adapte ao usuário.

1.2 Motivação

O desenvolvimento de um programa é complexo e requer esforço, perseverança e uma abordagem especial. Há um conjunto de habilidades envolvidas que vão muito além de saber a sintaxe da linguagem.

O programador além de conhecer a linguagem de programação também necessita ter conhecimentos na resolução de problemas e da ferramenta de desenvolvimento que está utilizando (GOMES; MENDES, 2007). Por exemplo: se um programador precisa escrever um programa para calcular a soma de dois números, ele necessita ter habilidade para

declarar as variáveis, definir a operação aritmética, fazer as atribuições necessárias e definir os mecanismos de entrada e saída das informações. Estes elementos devem ser utilizados de acordo com o comportamento do programa, ou seja, além de seguir uma estrutura lógica, deve existir uma variável para cada número e resultado, determinar o tipo de dados destas variáveis, verificar se os valores destas variáveis estão condicionadas à entrada de dados pelo usuário, assim deverá ter um mecanismo para a interação, a operação de soma depende das variáveis etc. Todos esses detalhes são necessários e devem ser considerados na obtenção da solução desejada, mesmo em um simples programa para calcular a soma de dois números.

O número de fatores cognitivos envolvidos na tarefa real de desenvolvimento de um programa agrava as dificuldades dos programadores iniciantes. Quando esses programadores começam a desenvolver códigos, não estão familiarizados com a linguagem de programação, regras de sintaxe e com as mensagens de erro, despendendo assim muito tempo em uma determinada tarefa, principalmente tentando corrigir os erros encontrados no processo de compilação.

Outros fatores são agravantes, como por exemplo o fato dos programadores não lerem completamente as mensagens de erro e a negligência de alguns desenvolvedores de compiladores ao não se atentar às mensagens de erro exibidas pelos seus softwares.

A maioria das mensagens de erro de compilação geralmente não possuem um padrão de exibição. A Tabela 2 apresenta o comparativo de 5 tipos de erro em 3 diferentes compiladores da linguagem Java: Eclipse, Netbeans e Jdeveloper. Os erros relativos as mensagens apresentadas são descritos na Tabela 1. É possível observar que compiladores diferentes emitem mensagem diferentes para o mesmo erro, além de usualmente possuírem termos de difícil compreensão e não fornecerem ajuda adicional, fazendo com o que o programador tenha que se recordar das ações que ele tomou quando o compilador emitiu uma mensagem igual à atual no passado.

Se analisarmos a capacidade de processamento de informações de acordo com Miller (MILLER, 1955), em que a memória de curto-prazo pode processar simultaneamente sete mais ou menos duas unidades de informação[1], e levarmos em consideração a quantidade de fatores envolvidos do desenvolvimento do software, podemos chegar a conclusão que esta tarefa de interpretação das mensagens de erro é árdua.

Entretanto, estes problemas não podem ser resolvidos por uma simples alteração na forma como as mensagens de erro são emitidas. Eles demandam todo um sistema relativamente complexo e adaptável ao nível de conhecimento do usuário. A complexidade é inerente à solução, pois cada erro deve ser tratado de forma diferente dos demais. Por exemplo, um determinado erro pode ter várias causas, pode exigir mais de um exemplo

[1] Uma unidade de informação pode ser de diferentes tipos (dígitos, letras, palavras etc) e depende do conhecimento prévio do usuário

Tabela 1 – Erros comuns em programação

Código em Java	**Id**

`public class error1 {`
`  public String my_member_variable = "somedata";`
 	
`  public static void main(String args[]) {`
`  String abc = "abc";`
`  String def = "def";`
 	
`  if ((abc + def) == "abcdef") {`
`  System.out.println(my_member_variable);`
`  }`
`  }`
`}`
	1

`public class error2 {`
`  void f() {`
`  int n = 10`
`  }`
`}`
	2

`public class error3 {`
`  public static void main(String args[]) {`
`  int b = 10;`
`  int c;`
 	
`  System.out.println("a + b = " + (a + b));`
`  }`
`}`
	3

`public class error4 {`
`  void x() {`
`  int a;`
 	
`  System.out.println("Insira um número");`
`  System.out.println("O número digitado foi " +`
`  scan.nextInt()));`
`  }`
`}`
	4

`public class error5 {`
`  void d() {`
`  int n = 10;`
 	
`  void e(){`
`  int x = 20;`
`  }`
`}`
	5

Tabela 2 – Mensagens de erro de 3 diferentes compiladores Java para erros de 1-5

Erro	**Eclipse**	**Netbeans**	**Jdeveloper**
1	Cannot make a static reference to the non-static field my_member_variable	non-static variabel my_member_variable cannot be referenced from a static context	Access not alowed to error01. my_member_variable
2	Syntax error, insert “;” to complete BlockStatements	“;” expected	Expression statement: Expecting ;.
3	a cannot be resolved to a variable	cannot find symbol - symbol: variable a - location: class error3	Type or variable ’a’ not found
4	Multiple markers at this line - Syntax error, insert “;” to complete BlockStatements - scan cannot be resolved to a variable	’;’ expected cannot find symbol - symbol: variable scan - location: class error4	Local variable_declaration. Expecting ;. Type or variable ’scan’ not found
5	Syntax error, insert “}” to complete MethodBody	illegal start of expression ’;’ expected - reached end of file while parsing	Block:Expecting }.

para a sua explicação, pode ser causado por inconsistências entre duas ou mais partes do código e usuários iniciantes e experientes deve receber apresentações muito diferentes da mensagem de erro.

Assim, um sistema que resolva os problemas anteriormente citados deve ser altamente configurável por causa da variação entre os erros e pela experiência/escolha do usuário.

O estudo de Traver (TRAVER, 2010) demonstra que algumas alterações na forma como essas mensagens de erro são exibidas podem favorecer o desenvolvimento de softwares e o aprendizado dos programadores iniciantes. Por exemplo a sinalização do local exato do erro, a exibição de mensagens adequadas com o nível de conhecimento do programador, opção de ajuda adicional, além a possibilidade de consulta aos erros anteriormente ocorridos. Essas alterações é que nos motiva no desenvolvimento do *plug-in* EMS.

1.3 Objetivos

Este trabalho tem como objetivo principal a implementação de uma ferramenta extensível, chamada Error Message System (EMS). O EMS é implementado como um *plug-in* para a IDE Eclipse, atendendo às recomendações propostas por Traver. Facilmente

configurável através de linguagens específicas de domínio, visa não somente a exibição das mensagens de erro de forma melhorada, mas também que o compilador se adapte ao programador.

O EMS permitirá que, quando o compilador sinalizar um erro, este seja explicado detalhadamente ao programador, apresentando as possíveis causas do erro, códigos anteriores que apresentavam o mesmo erro, sugestões de correção e exemplos e ligações visuais entre os trechos de código envolvidos no erro. Tornando assim o desenvolvimento do software mais rápido e com mais qualidade, já que a interação entre o programador e o compilador é melhorada.

Aspectos relevantes sobre as mensagens de erro, o seu impacto nos programadores, no desenvolvimento e na qualidade do software são abordados neste trabalho.

A utilização de linguagens específicas de domínio é necessária para facilitar a configuração da ferramenta. Linguagens específicas de domínio são utilizadas em diversos setores para agilizar tarefas rotineiras e resolver problemas de um domínio particular. Podemos citar como exemplo a linguagem de macros do Excel e HMTL.

Este trabalho buscou preencher as lacunas descritas por diversos autores e agrupadas no trabalho de Traver (TRAVER, 2010), buscando responder e solucionar as seguintes questões:

1. qual o impacto das mensagens de erro em programadores iniciantes e estudantes da computação?
2. é possível afirmar que a forma como as mensagens de erro dos compiladores são exibidas afetam a qualidade e o desenvolvimento do software?
3. a exibição de mensagens de erro adaptadas ao nível de conhecimento do usuário, facilita a interpretação, localização e correção do erro?
4. o *plug-in* EMS proposto neste trabalho, pode favorecer o aprendizado e desenvolvimento de programação de computadores?

O EMS é um grande projeto para exibição e correção de erros. Apenas parte do sistema EMS (Error Message System) foi desenvolvido nesta dissertação de mestrado. Esta parte consiste em:

- desenvolvimento de um *plug-in* do Eclipse para que a linguagem Cyan e seu compilador sejam acoplados a este ambiente;
- definição das linguagens específicas de domínio para configurar o EMS;
- desenvolvimento dos compiladores para essas linguagens específicas de domínio;

- desenvolvimento das interfaces gráficas para geração de código nas linguagens específicas de domínio;

- desenvolvimento e implantação do *plug-in* EMS no IDE Eclipse.

Sugestões para correção do erro e ligação visuais não fazem parte desse projeto de dissertação e estão previstos em trabalhos futuros.

1.4 Metodologia de Desenvolvimento do Trabalho

O plano de trabalho desta dissertação consiste de três partes: estudo, construção/implementação do *plug-in* EMS e análise dos resultados com usuários da ferramenta.

O estudo envolveu fundamentos da Interação Humano Computador (IHC), engenharia de software, linguagens de programação, compiladores e mensagens de erro de compiladores.

A construção e implementação do *plug-in* EMS envolve o projeto de várias linguagens específicas de domínio (LEDs), o desenvolvimento de vários pequenos compiladores para estas LEDs e o desenvolvimento de interfaces gráficas como apoio aos usuários para a personalização das suas mensagens de erro.

Como o objeto de teste foi o compilador da linguagem Cyan, também foi necessário o desenvolvimento de um *plug-in* para o acoplamento do compilador Cyan a IDE Eclipse.

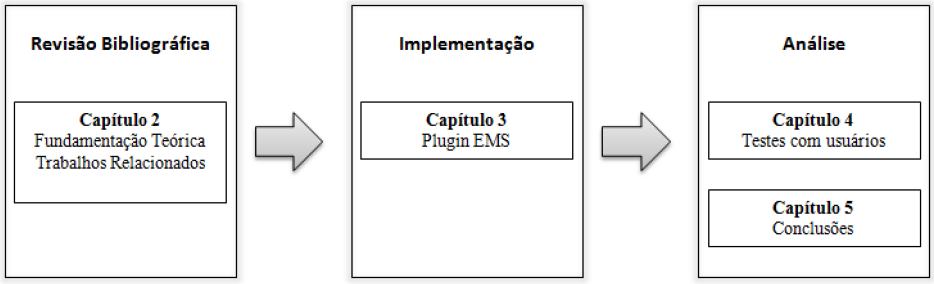
A análise dos resultados consiste no desenvolvimento de casos de testes que foram aplicados a programadores iniciantes e/ou estudantes da computação, a fim de comprovar a efetividade da proposta deste trabalho.

1.5 Organização do Trabalho

A estrutura deste trabalho foi divida em três partes, conforme visto na Figura 1: revisão bibliográfica, implementação e análise. A etapa da revisão fornece o conhecimento necessário para a etapa de implementação, ou seja, o desenvolvimento da proposta, considerações iniciais, intermediárias e finais. A etapa de análise apresenta a interpretação dos resultados dos testes realizados com os programadores iniciantes e as conclusões.

No Capítulo 2 são apresentados os fundamentos teóricos necessários para a compreensão da proposta. Primeiramente são descritos os fundamentos básicos sobre compiladores, em seguida os conceitos da Interação Humano Computador, sistema cognitivo e padrões para mensagens de erro. Por último, os tópicos utilizados para criação do *plug-in* EMS: linguagens específicas de domínio e ferramentas para o desenvolvimento de *plug-ins*.

Figura 1 – Estrutura da dissertação



O Capítulo 3 se concentra nos trabalhos relacionados, apresentando as pesquisas existentes na área de mensagens de erro de compiladores, as pesquisas da IHC para definição de padrões de sistemas e mensagens e finalmente são apresentados alguns dos sistemas criados para solucionar o problemas com as mensagens de erro.

O Capítulo 4 descreve o procedimento de implementação do *plug-in* proposto. O objetivo do capítulo é apresentar os conceitos, tecnologias e procedimentos adotados para viabilizar a construção do *plug-in* EMS.

No Capítulo 5 são abordados os testes com os usuários. O capítulo apresenta os procedimentos de teste e validação e a análise de seus resultados.

O Capítulo 6 é dedicado aos resultados da dissertação e sua discussão. É também apresentada uma síntese dos objetivos alcançados pela dissertação.

2 Fundamentos teóricos

Esta sessão apresenta os aspectos teóricos essenciais para um melhor entendimento deste trabalho. Inicialmente são expostos, de forma sucinta, noções básicas sobre compiladores, seu funcionamento e suas mensagens de erro. Posteriormente o foco passa para a área de Interação Humano Computador, descrevendo um modelo conceitual para as mensagens de erro. Em seguida são apresentados os conceitos sobre linguagens específicas de domínio, criação e implementação de *plug-ins*. Por último são apresentados os trabalhos relacionados.

2.1 Aspectos de um Compilador

Compiladores são sistemas que convertem programas de uma linguagem para a outra. A linguagem que um compilador traduz é chamada linguagem-fonte, sendo que o código é gerado em uma linguagem-alvo. O processo de compilação desenvolve-se em diversas etapas, que são mostradas na Figura 2.

Na análise léxica, o analisador reúne os caracteres do programa-fonte em unidades léxicas: os identificadores, as palavras reservadas, os operadores e os símbolos de pontuação.

A análise sintática é responsável por conferir se o código-fonte está de acordo com a gramática da linguagem.

O gerador de código intermediário produz um programa em uma linguagem diferente, no nível intermediário entre o programa-fonte e a saída final do compilador.

O analisador semântico faz conferências no código-fonte que não são feitas pelo analisador sintático.

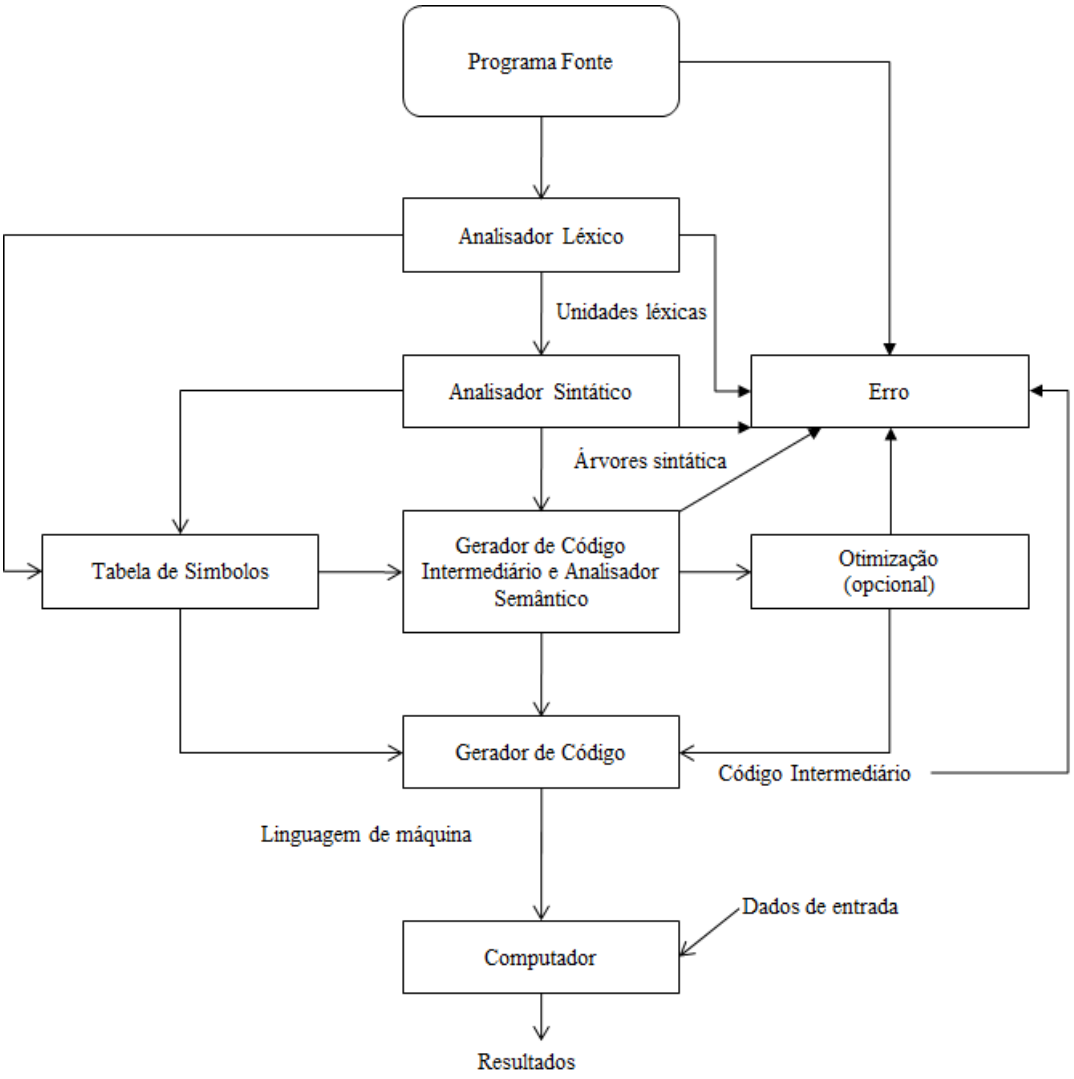
A otimização torna o código na linguagem-alvo mais rápido e/ou menor.

O gerador de código converte o código intermediário otimizado do programa para um programa em linguagem de máquina.

A tabela de símbolos serve como um banco de dados para o processo de compilação. Seu principal conteúdo são informações sobre tipos e atributos de cada identificador definido pelo usuário no programa. Essas informações são colocadas na tabela de símbolos pelos analisadores léxico, sintático e semântico e usadas pelo analisador semântico e pelo gerador de código.

Estas fases da compilação são melhor descritas a seguir.

Figura 2 – O Processo de compilação.



Fonte: baseado em Aho (AHO et al., 2007)

2.1.1 Análise léxica

A análise léxica é a primeira verificação realizada pelo compilador. A função do analisador léxico (*scanner*, em inglês) é:

> Fazer a leitura do programa fonte, caractere a caractere, e traduzi-lo para uma sequência de símbolos léxicos, também chamados *tokens* (PRICE; TOSCANI; UFRGS., 2000).

Durante a análise léxica, os *tokens* são classificados como palavras-reservadas, identificadores, símbolos especiais, variáveis, entre outros que compõem uma linguagem de programação (AHO et al., 2007). Veja por exemplo a sequência de caracteres: `MEDIA = SOMA/4`. Esta sequência pode ser agrupada pelo analisador léxico em cinco *tokens*:

Valor	**Categoria**
MEDIA	identificador
=	sinal de atribuição
SOMA	identificador
/	operador aritmético de divisão
4	número

Os tokens foram separados em valor e categoria. Onde a categoria indica a natureza da informação contida em valor. Por exemplo "MEDIA" refere-se a um identificador que pode ser uma variável local, uma variável de instância etc. O analisador léxico também tem como função ignorar os espaços em branco, os comentários e detectar erros léxicos. Os erros léxicos ocorrem quando um *token* identificado não pertence à gramática da linguagem-fonte, como por exemplo: quando caracteres não previstos, sequências inválidas etc. são encontrados no programa fonte.

Poucos erros podem ser verificados pelo analisador léxico devido a sua visão localizada do código. Por exemplo em: `fi ( a == f(x) )`. O analisador léxico entende "fi" como um identificador e não como um erro.

Nesta fase há algumas técnicas para recuperação de erros:

a) *panic mode* – o analisador ignora os caracteres até que um *token* válido seja encontrado;

b) remoção do caractere estranho;

c) inserção do caractere ausente;

d) substituição do caractere incorreto;

e) transposição de caracteres adjacentes.

Visto que o foco do EMS é a exibição das mensagens de erro, não entraremos em mais detalhes sobre a recuperação de erros.

Erros léxicos são adequados para a composição das mensagens do EMS. Estes erros facilmente identificados pelo compilador e geralmente são causados por um único caractere ou por uma sequência deles. O erro não depende do contexto sendo facilmente explicado através de uma mensagem curta emitida pelo EMS.

2.1.2 Análise Sintática

A análise sintática (*parsing*, em inglês), é o processo de analisar uma sequência de entrada para determinar se ela está de acordo com a gramática formal da linguagem-fonte.

O analisador sintático agrupa os *tokens* fornecidos pelo analisador léxico em estruturas sintáticas, construindo a árvore sintática correspondente. Para isso, utiliza a

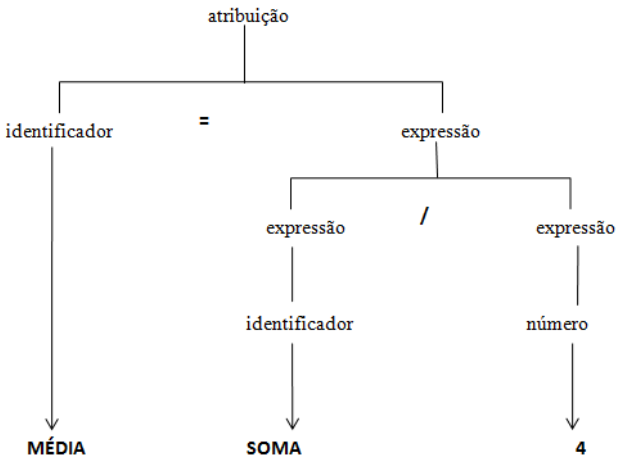
gramática formal da linguagem fonte. É a gramática da linguagem de programação que determina qual a estrutura sintática que os programas-fonte devem ter.

Árvores sintáticas são estruturas de dados em árvore, que representam a estrutura sintática de uma seqüência de caracteres de acordo com a gramática formal da linguagem e impõem estruturas hierárquicas às sentenças das linguagens geradas. Usualmente essas árvores contém um nó-folha para cada *token* (símbolo terminal) e um nó-intermediário para cada símbolo não terminal que resulta da aplicação de uma regra da gramática.

A árvore sintática é a saída lógica da análise sintática, constituindo uma representação intermediária utilizada na análise semântica. É representada graficamente na Figura 3. Consiste da derivação de uma sentença, construindo a estrutura hierárquica que originou a sentença. A raiz da árvore representa o símbolo inicial, os nós interiores são não-terminais e os símbolos terminais e a palavra vazia são folhas.

A Figura 3 representa a árvore sintática do comando `MEDIA = SOMA / 4` exemplificado na seção anterior.

Figura 3 – Árvore sintática



Fonte: baseado em Aho (AHO et al., 2007)

Quando as construções do programa-fonte não estão de acordo com as regras especificadas pela gramática, o analisador emite uma mensagem de erro.

O EMS não é recomendado para a emissão de mensagens de erros sintáticos. O compilador não consegue construir a árvore sintática, então não consegue identificar a maioria das informações que o EMS utiliza para apresentar a mensagem de erro. Por exemplo, o nome do método corrente, as variáveis locais ou até mesmo a instrução corrente.

Estes erros possuem características específicas e suas causas corretas são nor-

malmente impossíveis de identificar. É comum o esquecimento de um simbolo fazer o compilador emitir uma mensagem completamente não relacionada ao erro do ponto de vista do usuário. Veja por exemplo, o seguinte código na linguagem Java:

```
1  public class Error{
2    public static void Main (String[] args){
3                int x = 0;
4                if(x >0)
5                       System.out.Println ("Maior que zero");
6                 }
7           }
8   }
```

O compilador[1] irá sinalizar o seguinte erro na linha 8: `"class, interface or enum expected"`, quando na verdade, o que causou o erro foi a ausência da abertura de colchetes após a condição da estrutura "if" ou o colchete a mais na linha 6, pois neste caso o símbolo para abertura da estrutura "if" não é obrigatório. Observe o código a seguir:

```
1 public class Error{
2    public static void Main (String[] args){
3                int x = 0;
4                if(x <=0)
5                 System.out.println ("Menor ou igual a zero");
6                 System.out.println ("Verifique o valor inserido.");
7                else
8                  {
9                     System.out.println ("Maior que zero");
10                  }
11          }
12   }
```

O compilador emite a seguinte mensagem de erro na linha 7: `"else without if"`. Apesar de "if" estar visivelmente declarado no código, o compilador não consegue identificar a instrução devido à ausência da abertura e fechamento de colchetes antes e após as instruções das linhas 5 e 6. Erros como este são difíceis de identificar e corrigir,

[1] Código compilado na IDE Eclipse

principalmente para programadores iniciantes, já que estes vão diretamente ao local sinalizado pelo compilador e não observam todo o contexto do programa.

Veja o exemplo em Cyan a seguir:

```
var n = 0
var i = 0;
```

O compilador sinalizaria o erro "Identificador esperado" na segunda linha, não conseguindo listar "n" como uma variável local ao apresentar este erro, quando o que causou o erro foi a ausência de ";" na linha anterior a sinalizada.

No código escrito na linguagem Cyan abaixo, o compilador emitiria o erro semântico "método yy não foi encontrado no protótipo Double" sendo que o erro, do ponto de vista do programador, é sintático devido à ausência do ";" após a instrução para o cálculo da função seno da variável "x" (`x sin`).

```
var Double x = 3.14/2;
var Double n = x sin
yy zz;
```

O envio de mensagem `x sin yy zz` poderia ser válido.

Para recuperação dos erros sintáticos, os compiladores utilizam algumas das técnicas citadas abaixo:

a) ao identificar um erro o analisador descarta os símbolos de entrada, até que seja encontrado um *token* pertencente ao subconjunto de *tokens* de sincronização, usualmente delimitadores, tais como o ponto-e-vírgula ou o fim, cujo papel no programa-fonte seja claro;

b) recuperação de frases: o analisador sintático realiza uma substituição local na entrada restante, por exemplo: substituir uma vírgula inadequada por um ponto e vírgula, remover um "=" excedente etc.;

c) produção de erro: aumenta-se a gramática, de forma a incluir os erros mais comuns. Quando uma produção de erro é identificada pelo analisador, diagnósticos apropriados são apresentados;

d) correção global: usa algoritmos de escolha de sequência mínima de mudanças necessárias para se obter a correção global. Como por exemplo: dada uma cadeia x, o analisador procura árvores gramaticais que permitam transformar x em y (cadeia correta) realizando mínimas modificações.

2.1.3 Análise Semântica

A análise semântica é a terceira fase da compilação. Tem por objetivo verificar se as construções identificadas pela análise sintática estão em acordo com as regras semânticas da linguagem que está sendo compilada. O analisador semântico utiliza a árvore sintática construída pelo analisador sintático para verificar compatibilidade de tipos, analisar o escopo das variáveis, verificar correspondência entre argumentos e parâmetros etc.

Por exemplo, para o comando `MEDIA = SOMA /4`, o analisador semântico faz as seguintes verificações:

a) o identificador MEDIA foi declarado?

b) o identificador MEDIA é uma variável?

c) qual o escopo da declaração da variável MEDIA? global ou local?

d) qual o tipo de dados da variável MEDIA? o valor atribuído no lado direito do comando de atribuição é compatível?

Caso ocorra algum problema durante esta análise, o analisador então emite uma mensagem de erro.

Exemplos comuns de erros semânticos são: variáveis não declaradas, chamadas de funções ou métodos com um número incorreto de parâmetros, operações matemáticas com tipos de dados diferentes, tipos de operadores não compatíveis com os da linguagem. Por exemplo a utilização do caractere "^" como operador para calcular potenciação, quando na linguagem Java é necessário a utilização da função *Math.pow(x,y)*, entre outros.

É fácil identificar se determinado erro foi causado por um erro anterior. Por exemplo: se uma variável foi declarada com um tipo inexistente, qualquer expressão que utiliza esta variável estará incorreta também.

Os erros semânticos permitem uma análise sintática prévia e a construção da árvore de sintaxe abstrata. Então é possível ao compilador, ao sinalizar um erro semântico, identificar o nome da classe em que ocorreu o erro, os nomes dos métodos, das variáveis de instância etc. O compilador consegue fornecer ao EMS todas as informações que podem ser utilizadas na composição de uma mensagem de erro apropriada.

Estes erros são ideais para a exibição de mensagens pelo EMS. Erros semânticos têm sua causa bem definida, permitindo ao *plug-in* uma explicação muito mais refinada e detalhada do erro.

2.1.4 Geração de Código

A geração de código pode ser dividida em duas fases: intermediária e final. Na fase intermediária, a árvore sintática é transformada em uma representação intermediária do

código-fonte. Esta representação intermediária é mais próxima da linguagem-objeto do que o código-fonte, entretanto é mais fácil de manipular que o código Assembly ou código de máquina.

A geração de código final é a última fase da compilação. Transforma o código da representação intermediária no código final. É a tradução da linguagem-fonte para código na linguagem-alvo, usualmente uma linguagem de baixo nível como Assembly ou código em linguagem de máquina. Vamos utilizar por exemplo o comando utilizado nas seções anteriores: `MEDIA = SOMA /4`. Após a geração final a linguagem pode ser transformada no código Assembly abaixo:

```
MOV AX, [SOMA]
DIV CX,4
MOV [MEDIA], CX
```

2.2 Mensagens de erros

A principal forma de comunicação entre os compiladores e os programadores é através da emissão de mensagens de erros. O objetivo de uma mensagem é indicar ao programador o local exato onde ocorreu o erro, sua causa e sugerir o que pode ser feito para sua correção. Contudo a exibição de mensagens pouco informativas e difíceis de interpretar são comuns em alguns compiladores atuais e interferem diretamente na resolução e prevenção do erro no futuro. (TRAVER, 2010).

A falta de conhecimento do programador sobre a sintaxe da linguagem, raciocínio lógico, alocação de recursos necessários ao funcionamento do objeto, tipos de dados e outros conceitos utilizados na programação são algumas das dificuldades encontradas na codificação de um programa (KO; MYERS, 2005).

Para Ebrahimi (EBRAHIMI, 1994), além dos problemas citados por citados por Ko (KO; MYERS, 2005), os programadores iniciantes também não conseguem compreender as mensagens de erro exibidas pelos compiladores.

A maioria dos compiladores emite todos os tipos de mensagens de erro exatamente da mesma forma: uma mensagem curta e a sinalização de um único ponto no código onde o erro foi descoberto, sem ajuda adicional. Estas mensagens são difíceis de compreender por diversas razões: são curtas demais (não permitindo explicações detalhadas), não mostram todas as possíveis causas do erro (um erro pode ter mais de uma causa), não exibem ou apontam os locais do código fonte que causaram o erro (quase sempre mostram apenas uma única linha), não mostram exemplos que não apresentam aquele erro, não permitem alterar a mensagem de erro de acordo com a experiência do usuário e, de modo geral, não permitem qualquer adaptação das mensagens ao usuário do compilador (TRAVER, 2010).

A falta de informações consistentes sobre o erro ocorrido levam o programador a compilar várias vezes o mesmo código sem realizar nenhuma alteração. Inconscientemente, o programador espera que o computador mude de ideia. Apesar de parecer absurdo, seres humanos estão acostumados a lidar com seres humanos, que mudam constantemente de opinião; ou seja, a interação com humanos não é tão rígida como com as máquinas (REEVES; NASS, 1996).

Se a mensagem de erro é vaga e confusa, o usuário deixa de confiar no compilador e é induzido a tomar ações aleatórias a fim de corrigir o erro que ele não sabe exatamente por que ocorreu. Estas ações desnecessárias impactam diretamente na produtividade, desperdício de horas e recursos e nas emoções dos programadores.

Em contrapartida, boas mensagens indicam ao programador as direções corretas a serem tomadas, além de facilitar a compreensão do que causou o erro no código. Essas mensagens também auxiliam os estudantes no aprendizado na programação, clareando os conceitos e desenvolvendo modelos mentais (TRAVER, 2010). Um modelo mental é o que define como um indivíduo irá perceber o que está acontecendo, como ele irá se sentir com isso, como ele pensa e, finalmente, como irá agir.

Para Shneiderman (SHNEIDERMAN, 1986), quando um programador iniciante se depara com uma mensagem em tom violento, como por exemplo *FATAL ERROR*, *RUN ABORTED* etc. ou códigos de erros obscuros como *OC7* ou *IEH2191* eles ficam abalados, confusos e desencorajados a continuar com a programação.

Em 1999, Alexandrescu,(ALEXANDRESCU, 1999) apud (TRAVER, 2010), pesquisador e programador na linguagem C++, escreveu uma carta à comunidade de desenvolvedores da linguagem C++ para que eles tornassem as mensagens dos compiladores mais fáceis de compreender. Ele obteve respostas de especialistas em compiladores que confirmaram que a área das mensagens de erro estava realmente sendo negligenciada e citaram três motivos para isso: primeiro, porque acreditavam que mensagens mais detalhadas iriam tornar a compilação mais lenta, visto que os compiladores utilizavam bastante recursos de memória do computador (especialmente das linguagens C e C++). Assim, uma maneira de se obter um melhor desempenho sempre foi tentar reduzir ao máximo possível a quantidade de informações que o compilador precisa manter durante a compilação. Segundo, porque acreditavam que os programadores devem estar familiarizados com a linguagem e com o compilador e terceiro, porque a prioridade de estudo na área de compiladores é no desenvolvimento de novas funcionalidades e não em melhorar as mensagens de erro.

Essas argumentações não se adaptam mais à nossa realidade. Computadores com elevada capacidade de processamento estão sendo desenvolvidos, sendo assim a exibição de mensagens mais detalhadas não interfere na velocidade de geração de código-fonte dos compiladores atuais. Da mesma forma que, devido aos grandes avanços tecnológicos e a grande quantidade de pessoas utilizando sistemas de computação, os softwares atuais são

projetados para que favoreçam sua utilização e o aprendizado da programação, ou seja a aplicação guia o usuário na execução das tarefas.

2.2.1 Evidências do problema

É sabido que programadores iniciantes e estudantes da computação possuem maior dificuldade em interpretar as mensagens emitidas pelos compiladores que os programadores mais experientes (MAYER, 1981)(EBRAHIMI, 1994)(SHACKELFORD; BADRE, 1993).

Existem vários fatores que contribuem para essa dificuldade: conhecimento prévio do programador, compilador, ambiente de trabalho e o sistema cognitivo. Esses fatores são detalhados a seguir:

- conhecimento prévio. A maioria dos estudantes de computação e programadores iniciantes apresentam enormes dificuldades no desenvolvimento de programas. Estas dificuldades estão diretamente ligadas aos deficit que eles apresentam em resolver problemas genéricos, como exemplo, calcular a porcentagem de aumento no salário de um funcionário. A resolução de problemas requer múltiplas competências que os estudantes frequentemente não têm (GOMES; MENDES, 2007).

  Além disso, a programação exige um estudo prático e intensivo de conceitos da linguagem de programação e das ferramentas utilizada para o desenvolvimento (EBRAHIMI, 1994). Essas competências devem ser desenvolvidas na fase inicial da aprendizagem, não apenas no desenvolvimento de habilidades específicas de programação, mas também na melhoria e/ou na consolidação de competências que deveriam ter sido adquiridas em anos precedentes, nomeadamente as capacidades de resolução de problemas, de raciocínio, de lógica entre outras.

  A complexidade da programação em si interfere diretamente na interação entre os programadores e os compiladores. O esforço mental do usuário nas primeiras fases de aprendizado de programação faz com que seu foco seja direcionado somente à execução do programa que está sendo escrito. Isso torna a interação com as ferramentas de desenvolvimento e compilação de código-fonte uma tarefa secundária; ou seja, o programador não consegue interpretar o retorno emitido pelos compiladores por vários motivos:

  a) primeiro, porque não costuma ler completamente as mensagens de erro (VEE; MEYER; MANNOCK, 2005). Este fato pode ser diretamente observado nos alunos das disciplinas de Programação dos cursos Técnico de Informática da ETEC Fernando Prestes em Sorocaba[2]. Foi possível verificar que, apesar de serem orientados sobre o comportamento dos compiladores

[2] Observação direta realizada no primeiro semestre do ano de 2015

quando da ocorrência de um erro, que o compilador sinaliza a linha e emite uma mensagem sobre o erro ocorrido, a maioria dos estudantes não se atentam aos sinais emitidos pelo compilador da linguagem C#. Tendem a ir diretamente à linha que ocorreu o erro, tentando adivinhar o que pode ter ocorrido e tomando ações arbitrárias para corrigi-lo.

b) segundo, porque a maioria das mensagens são exibidas no idioma inglês, e apesar das ferramentas disponíveis para tradução, por exemplo o software AjudeC (FARIA; JÚNIOR, 2006) e a possibilidade de configuração da IDE para a linguagem nativa do programador, elas não oferecem ajuda adicional além da tradução literal da mensagem.

c) por último, porque as mensagens são curtas e pouco informativas.

Com o tempo, os programadores se familiarizam com as mensagens de erro e se adaptam, relembrando como lidaram com elas no passado. Grande parte dos programadores não possuem o hábito de anotar os erros ocorridos e como foram solucionados, fazendo com que tenham que relembrar as ações tomadas para corrigir os erros. Como dito anteriormente, alguns programadores não tentam nem ao menos ler as mensagens (VEE; MEYER; MANNOCK, 2005) ou compreender o que elas significam (SCHORSCH, 1995). Muitas mensagens são realmente difíceis de compreender, mas na maioria das vezes oferecem informações valiosas de como resolver o problema;

- compilador. Compiladores são projetados para geração eficiente de código objeto. Sua tarefa principal não é a correção e exibição das mensagens de erros. Então, normalmente fornecem mensagens de erros genéricas e em cascata, ou seja, um erro pode ter ocorrido em um local e sinalizado em uma referência ao problema original. Por exemplo uma classe filha passando parâmetros que não correspondem à assinatura do método na classe pai. O compilador irá sinalizar o erro na classe filha, tornando difícil, para um programador iniciante, identificar e corrigir o erro;

- ambiente e infraestrutura. Em alguns casos, por fatores econômicos, as máquinas e ferramentas utilizadas não são adequadas ao desenvolvimento do software. Em outras vezes, o programador não utiliza corretamente ou desconhece os recursos fornecidos por essas ferramentas, preferindo trabalhar diretamente com linha de comando, onde as mensagens de erro se misturam com as linhas de entrada e saída do *Prompt de comando*, (TRAVER, 2010);

- sistema cognitivo. A capacidade de cognição se refere à capacidade do cérebro de identificar, compreender, aprender e recordar a informação captada através dos nossos cinco sentidos(audição, paladar, tato, olfato e visão). O termo é oriundo do latim *Cognitio* e significa adquirir conhecimento por meio da percepção;

O sistema cognitivo é o processo pelo qual o ser humano constrói o seu conhecimento. Durante esse processo, algumas limitações podem surgir. Estas limitações são descritas pela Teoria da Carga Cognitiva (CHANDLER; SWELLER, 1991) que define a impossibilidade de um ser humano processar muitas informações simultaneamente.

O excesso de informação gera um esforço demasiado para todo o processo cognitivo, dando origem à uma sobrecarga cognitiva que dificulta a compreensão do conteúdo que se deseja aprender.

Em outras palavras, as dificuldades encontradas na programação não são exclusivas dos desenvolvedores, se originam das limitações da mente humana. O modelo de armazenamento sensorial da memória humana é dividido em duas partes: memória de armazenamento de curto prazo e a memória de armazenamento de longo prazo (NORMAN, 1970). As limitações na memória, e, em particular, na memória de curto prazo (memória de trabalho) são importantes de se considerar quando um programador está desenvolvendo um programa. Por exemplo: quando o programador olha para a lista de mensagens de erro, ele deve manter na memória uma certa imagem do que o código representa. Por outro lado, quando olha para o código, deve lembrar do que a mensagem de erro estava tentando lhe dizer. Além disso as mensagens de erro são genéricas, não exibem o trecho do código onde ocorreu o erro.

A nossa capacidade de reter informação é limitada em quantidade e tempo, como afirma o clássico de Miller (MILLER, 1955) *The Magic Number Seven, Plus or Minus Two*. De acordo com Miller se as informações da memória de curto prazo não forem transferidas para a memória de longo prazo no período de quinze a trinta segundos, essas informações são descartadas. Uma pessoa pode reter sete mais ou menos duas unidades de informação. E essa quantidade está diretamente relacionada ao conhecimento anterior do usuário. Um exemplo bem conhecido é a sequência de letras “H-I-C-S-A-U-I-W-M-P”. Lidas sem qualquer diferença de entonação e de intervalo, podem ser difíceis de se lembrar. Já a sequência “I-H-C-U-S-A-W-I-M-P”, composta com as mesmas letras mas em outra ordem pode ser facilmente lembrada. Para um grupo de pessoas a segunda sequência representa apenas três unidades de informação ao invés das dez da sequência anterior.

Esta é uma boa razão para manter o código fonte e as mensagens do compilador visíveis ao mesmo tempo, pois permite ao programador assimilar o que a mensagem de erro está querendo dizer ao invés de ter que recordar o que causou o erro.

2.3 Interação Humano Computador

A Sociedade Brasileira de Computação define a Interação Humano-Computador (IHC) como:

> A área que se dedica a estudar os fenômenos de comunicação entre pessoas e sistemas computacionais que está na interseção das ciências da computação e informação e ciências sociais e comportamentais e envolve todos os aspectos relacionados com a interação entre usuários e sistemas (SBC, 2015).

Dentre as principais características que exprimem a qualidade de sistemas interativos, Pressman e Lowe (LOWE; PRESSMAN, 2009) apontam: ser fácil de usar, fácil de aprender, fácil de navegar, intuitiva, consistente, eficaz, livre de erros e funcional. Barbosa e Silva (BARBOSA; SILVA, 2010), enfatizam essas características e apresentam quatro conceitos relacionados à interface: comunicabilidade, acessibilidade, usabilidade e experiência do usuário (engenharia cognitiva). Esses conceitos estão diretamente relacionados à qualidade das interfaces e são descritos a seguir:

- comunicabilidade: o *designer* deve se expressar adequadamente por meio da interface, assegurando que o usuário consiga prever e compreender o que fazer no sistema para realizar tarefas sozinho, com eficiência, facilidade e com uma comunicação em mão dupla entre usuário e sistema. Assim, o usuário entende a mensagem do designer e consegue interagir com o sistema e o sistema responde às ações do usuário informando o que está acontecendo, evitando que o usuário fique angustiado e insatisfeito(NORMAN, 1998)(BARBOSA; SILVA, 2010).

- acessibilidade: Silva e Barbosa (BARBOSA; SILVA, 2010) enfatizam que, mesmo que uma interface seja fácil de usar e com alta comunicabilidade, barreiras ao acesso de sistemas interativos podem impossibilitar o uso do sistema por pessoas com necessidades especiais. O usuário, ao usar um sistema, necessita de coordenação motora para agir e manipular a interface: visão, audição, tato e percepção para identificar e reconhecer as mensagens enviadas pelo sistema e atividades mentais para interpretar as mensagens, planejar suas tarefas e verificar os objetivos. É necessário portanto, que o *designer* esteja atento quando da criação de interfaces, para que os sistemas sejam acessíveis a todos os usuários, com limitações ou não.

- usabilidade: a usabilidade está relacionada com a facilidade de uso e a qualidade na experiência de utilização de um sistema. Nielsen e Mack (NIELSEN; MACK, 1994) conceituam usabilidade como:

  > um atributo de qualidade relacionado à facilidade do uso de algo. Mais especificamente, refere-se à rapidez com que os usuários podem aprender a usar alguma coisa, a eficiência deles ao usá-la, o quanto lembram daquilo, seu grau de propensão a erros e o quanto gostam de utilizá-la. Se as pessoas não puderem ou não utilizarem um recurso, ele pode muito bem não existir (NIELSEN; MACK, 1994).

  A usabilidade permite que os usuário do sistema atinjam seus objetivos de forma eficaz e eficiente.

2.3.1 Experiência do usuário (Engenharia Cognitiva)

A engenharia cognitiva foi proposta por Norman (NORMAN, 1986), na tentativa de utilizar conhecimentos da psicologia cognitiva, da ciência cognitiva e dos fatores humanos para entender os processos cognitivos humanos e a capacidade e limitações da mente, no intuito de usá-los para desenvolver sistemas interativos agradáveis, motivadores, prazerosos e fáceis de usar (BARBOSA; SILVA, 2010).

A engenharia cognitiva está centrada na relação entre usuário e sistema, na interação do usuário com um sistema concebido, não sendo seu foco o projetista de sistema ou o processo de design do sistema. Portanto, a engenharia cognitiva tem como foco os processos psicológicos dos usuários e os fenômenos envolvidos durante a interação com o sistema (ROCHA; BARANAUSKAS, 2003) (BARBOSA; SILVA, 2010). Assim, com o propósito de entender como os usuários interagem com as interfaces do sistema, Norman (NORMAN, 1986) propõe a Teoria da Ação.

Na Teoria da Ação, a interação entre usuário e sistema é realizada num ciclo de ação que envolve sete etapas e dois alvos a serem atingidos. Norman (NORMAN, 1986) define esses dois alvos como golfos a serem atravessados. Um é o Golfo da Execução, que envolve todo o esforço mental do usuário para planejar sua ação diante dos comandos e funções percebidos no sistema. O outro é o Golfo de Avaliação, que envolve o momento em que o usuário coloca o planejamento da sua ação em prática, executando ações, ou seja as entradas no sistema, e o momento que o usuário, por meio das saídas do sistema, avalia se os seus objetivos estabelecidos no planejamento da ação foram alcançados.

Para atravessar esses golfos, o usuário deve realizar uma seqüência de etapas dentro de cada golfo.

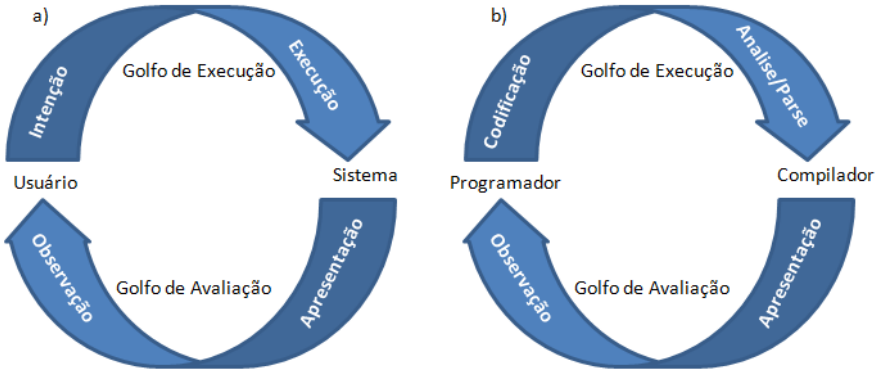
No golfo da execução, o ciclo se inicia com a tarefa do usuário, o objetivo pelo qual o usuário deseja interagir com o sistema. A partir da definição do objetivo, inicia-se a etapa de intenção em que o usuário elabora uma estratégia para alcançar o objetivo, considerando o estado atual do sistema e o estado a ser alcançado. Após definida a intenção, avança-se para a próxima etapa, a especificação da ação, considerando os comandos e funções oferecidos pelo sistema, o usuário elabora uma série de passos, ações interativas com os controles do sistema para alcançar o objetivo ou executar a tarefa. Até o momento, o usuário executou apenas atividades mentais, porém, na próxima etapa, na execução, o usuário colocará todo o esforço mental em uma ação física, colocando em prática o planejamento e interagindo com o sistema. A partir desse momento, da execução do planejamento, é atravessado o golfo de execução e iniciada a travessia do golfo de avaliação.

O golfo de avaliação se inicia com a percepção do usuário após o processamento de sua ação pelo sistema; o usuário espera uma mudança no estado do sistema causada pelas entradas de sua ação. A partir da percepção da mudança de estado, avança-se para a

próxima etapa, a interpretação do novo estado do sistema pelo usuário. Tendo interpretado o novo estado, inicia-se a próxima etapa, a avaliação, nela o usuário avalia o objetivo pretendido e a resposta do sistema.

A Figura 4 demonstra a teoria da ação (NORMAN, 1986) e o que o ocorre com os programadores durante o desenvolvimento do código. Na Figura 4 o modelo proposto por Dix (DIX, 2004), baseado na Engenharia Cognitiva proposta por (NIENALTOWSKI; PEDRONI; MEYER, 2008) é comparado com o processo de desenvolvimento mental dos programadores durante o desenvolvimento do código.

Figura 4 – Teoria da ação



a) Modelo de interação proposto por (DIX, 2004) b) Instância para o ambiente de um compilador (TRAVER, 2010) com os golfos de execução e avaliação (NORMAN, 1986).

É importante notar que, se o usuário não perceber que o sistema mudou de estado através de uma sinalização clara, ele possivelmente interpretará que nada ocorreu e que a sua meta inicial não foi atingida. Por exemplo, se o programador escreve um programa e esse não apresenta nenhum *feedback* quando executado, é impossível para o programador saber se o programa está funcionando ou não.

2.3.2 Modelos de Exibição de Mensagens

Qualquer sistema projetado deve ser fácil de aprender e lembrar, eficaz e agradável de usar (MOLICH; NIELSEN, 1990). Se utilizarmos essa abordagem nos compiladores, o que podemos verificar é que, para identificar e corrigir um erro apontado no programa pelo compilador, além de interpretar e compreender a mensagem emitida, o programador tem que recordar das ações que ele tomou quando o compilador emitiu uma mensagem igual à atual no passado (TRAVER, 2010).

Cada usuário apresenta um conjunto único de características que deve ser levado em consideração no momento da criação do ambiente. Assim, pesquisas estão sendo realizadas pela área da Interação Humano Computador (IHC) para que as aplicações desenvolvidas

sejam facilmente utilizadas por qualquer pessoa, seja ou não da área de computação (SIGCHI, 1992) (CARROLL, 2013).

Podemos observar por exemplo o aumento na criação de novas linguagens de programação de alto nível, como um meio de diminuir a enorme lacuna no nível de abstração que existe entre a linguagem de máquina e o pensamento humano.

A criação de ambientes de desenvolvimento que facilitam a edição, compilação, execução e depuração dos programas, através de componentes visuais que diminuem a quantidade de código a ser escrito, também são utilizados como forma de diminuir esta lacuna (TRAVER, 2010).

Nielsen (MOLICH; NIELSEN, 1990)(NIELSEN, 1995) fornece uma boa definição de como o diálogo entre o computador e o usuário deve ser. O guia abrange as mensagens de forma geral, mas pode ser facilmente adaptado para as mensagens de erros emitidas pelos compiladores. Em um trabalho posterior Traver (TRAVER, 2010) propõe um conjunto de características desejáveis para as mensagens de erros. Essas características se baseiam nos trabalhos de Nielsen (MOLICH; NIELSEN, 1990) (NIELSEN, 1995) que definem como boas mensagens devem ser.

- Clareza e Concisão: grande parte das mensagens emitidas pelos compiladores apresentam termos técnicos e são difíceis de decifrar até mesmo para programadores experientes. Além disso, muitos dos programadores não costumam ler completamente a mensagem apresentada, o que pode levar a uma má interpretação da mensagem e a não correção do erro. Sendo assim, as mensagens exibidas devem ser claras e diretas, informando brevemente e exatamente o erro ocorrido.

- Especificidade: um erro não específico deve ser estar relacionado a mais de um diagnóstico. Erros como: “Parse error, ilegal character, etc” são difíceis de interpretar e de resolver. Sendo assim, as mensagens de erro devem ser mais explícitas, de forma que o usuário consiga interpretar e resolver o problema ocorrido.

- Sensibilidade ao contexto: quando os erros são sensíveis ao contexto, o mesmo problema (diagnóstico) pode dar origem a diferentes mensagens de erro, ou seja, para um determinado erro, uma ligeira alteração no código provoca uma mudança radical no diagnóstico.

- Localização: um erro ocorrido em um determinado trecho de código pode ser manifestado longe do trecho que realmente causou o erro. É ideal portanto, que a mensagem indique exatamente onde o erro ocorreu, ou que forneça os passos para que o programador consiga identificar a causa exata do problema.

- Linguagem adequada: há uma série de questões relacionadas de como os textos das mensagens devem ser, levando em consideração o impacto que elas podem causar

nos programadores. Segundo Shneiderman (SHNEIDERMAN, 1986) as mensagens devem seguir alguns padrões citados a seguir:

- ser escritas em um tom amigável, indicando o que deve ser feito para sua correção e não condenando o usuário pelo erro ocorrido. Termos como *ILEGAL*, *ERROR*, *INVALID* devem ser eliminados ou reduzidos;
- ser específicas e direcionar o problema nos termos do usuário, evitando o uso de palavras obscuras e códigos internos;
- colocar o usuário no controle da situação, fornecendo as informações necessárias para a correção do erro;
- ter um formato limpo, consistente e compreensível, ao invés de códigos numéricos longos, mnemônicos obscuros e textos desordenados. Caso seja necessário a utilização de códigos, estes devem aparecer no final da mensagem.

Exibir uma mensagem em um tom positivo é tão importante quanto o conteúdo da mensagem. A forma como se diz ao usuário que algo está errado tem um grande efeito em sua percepção do programa. Uma mensagem em tom áspero pode frustar ou confundir o usuário.

2.3.3 Métodos de avaliação

Os métodos de avaliação permitem ao desenvolvedor fazer uma análise concreta da usabilidade da ferramenta. Destaca seus pontos fortes, contribui na correção de erros e identifica o que precisa ser melhorado.

O EMS foi avaliado por usuários (programadores iniciantes e estudantes da computação) e por especialistas (professores de computação e programadores experientes). Os métodos utilizados são detalhados a seguir.

2.3.3.1 Avaliação com o usuário

A avaliação com usuários busca problemas reais, para tanto, faz-se necessário envolver usuários reais na avaliação, o que a torna mais demorada e de alto custo. (PRATES, 2003). O envolvimento dos usuários na avaliação pode ser feita por meio de observação dos usuários ou por meio de investigação. Na investigação, são usados questionários, estudos de campo, entrevistas e outras técnicas para se obter dados sobre a opinião, expectativas e comportamentos dos usuários relacionados ao sistema (BARBOSA; SILVA, 2010). Na observação, os dados são coletados por meio da observação e registro do usuário utilizando o sistema, em laboratório ou em ambiente de trabalho (SOUZA et al., 1999).

2.3.3.2 Avaliação com o especialista

A avaliação com especialistas surgiu no início dos anos de 1990 e ganhou relevância por ser um método barato comparado com a avaliação com usuários e fácil de ser realizado. Lida menos com questões éticas e práticas. É principalmente eficaz e também uma alternativa para situações em que os usuários não estão acessíveis ou o custo de seu envolvimento é muito alto e o tempo muito curto (PREECE; ROGERS; SHARP, 2005).

Esse tipo de avaliação envolve um ou mais especialistas inspecionando e julgando a adequação da interface com base em princípios reconhecidos e em sua experiência profissional, no intuito de identificar possíveis problemas que podem ocorrer durante a interação do usuário com o sistema (CYBIS MARCELO SOARES PIMENTA, 1998). A avaliação envolvendo especialistas almeja antever potenciais problemas, porém, problemas não reais, visto que o avaliador tenta se colocar no papel de determinado usuário com determinado perfil (BARBOSA; SILVA, 2010).

Esse tipo de avaliação é utilizado para buscar problemas de usabilidade e propor soluções. Para Nielsen e Mack (NIELSEN; MACK, 1994) apud (BARBOSA; SILVA, 2010), os principais objetivos da avaliação com especialistas são: identificar, classificar e enumerar os problemas encontrados.

2.4 Linguagens Específicas de domínio

Uma Linguagem Específica de Domínio (LED do inglês *Domain Specific Language, (DSL)*) é uma linguagem de programação ou uma linguagem de especificação que oferece, através de notações e abstrações, poder expressivo direcionado a um problema de um domínio particular.

Embora seja uma linguagem específica e não forneça uma solução geral para muitas áreas, ela provê uma solução muito melhor para um domínio particular (DEURSEN; KLINT; VISSER, 2000) quando comparada com uma linguagem de propósito geral. A utilização de LEDs é bem comum na computação. Podemos citar como exemplo: CSS, HTML, SQL, entre outras.

As LEDs podem ser classificadas em internas e externas. LEDs internas utilizam uma linguagem hospedeira para a definição de uma linguagem em particular. LEDs internas são também referidas como LEDs incorporadas. A utilização de LEDs internas foi recentemente popularizada pela linguagem Ruby, embora tenha se originado na linguagem Lisp. Existem prós e contras na utilização dessas linguagens. Elas permitem a eliminação da barreira simbólica com a nossa linguagem base e possuem todo o poder que a linguagem base pode oferecer. Entretanto LEDs internas são limitadas pela sintaxe e estrutura da linguagem base (FOWLER, 2005).

As LEDs externas são linguagens independentes de qualquer outra. Têm sua sintaxe personalizada e é necessário escrever um compilador completo para processá-la. Alguns exemplos são as linguagens XML, SQL, HTML etc. (FOWLER, 2005).

2.5 O ambiente Eclipse

Eclipse (ECLIPSE, 2004) é uma plataforma de desenvolvimento integrado (IDE) lançada em 2001 como um projeto doado pela IBM para a comunidade *open-source*.

Atualmente, o Eclipse recebe suporte, além de uma extensa comunidade de desenvolvedores, de outras organizações como a Red Hat, Suse, Oracle, Borland etc. Sua principal característica é a utilização de uma arquitetura extensível baseada em componentes conectáveis, ou *plug-ins*, fornecendo um pequeno conjunto de serviços para controlar um grande conjunto de componentes trabalhando conjuntamente.

Todos os componentes do Eclipse, exceto seu núcleo, foram desenvolvidos como *plug-ins*. Através dessa infraestrutura genérica, diferentes vendedores ou contribuidores podem realizar comunicações ou declarar dependências entre seus *plug-ins* sem nenhum problema de compatibilidade. Esta estrutura de *plug-ins* suporta facilmente, por exemplo, a utilização de diversas linguagens de programação no Eclipse apesar desta IDE ser baseada em Java.

Workbench e *Workspace* são dois *plug-ins* indispensáveis da plataforma Eclipse. Eles provêm pontos de extensão para a maioria dos *plug-ins* nativos do Eclipse. O *"Workbench"* é o componente que possibilita a outros *plug-ins* estenderem a interface do Eclipse, como menus, barras de ferramentas, requisitar tipos diferentes de eventos e criar novas janelas. O *"Workspace"* possibilita a interação do usuário com diferentes recursos como projetos e arquivos.

2.5.1 *Plug-in* Eclipse

Um *plug-in* é o bloco de construção fundamental da plataforma Eclipse. A plataforma Eclipse é a soma de vários *plug-ins*. Cada *plug-in* contribui com funcionalidades para a plataforma, sendo que estes *plug-ins* são ativados quando ocorre uma requisição da funcionalidade. Estruturalmente cada *plug-in* reside em um subdiretório denominado `eclipse/plugins` dentro do local de instalação do Eclipse. Os diretórios e arquivos geralmente encontrados em um diretório de *plug-in* são:

- arquivo plugin.xml - arquivo contendo informações de detalhes do *plug-in*, como: nome do *plug-in*, extensões utilizadas, dependências entre objetos etc;

- arquivo build.properties - arquivo de configuração de propriedades para o arquivo plugin.xml. Contém informações dos caminhos das pastas dos arquivos fontes, pastas de saída e outros arquivos necessários para funcionamento do *plug-in*;
- arquivo about.xml - arquivo HTML com informações sobre versão e licença do *plug-in*;
- pasta lib - diretório de bibliotecas necessárias para o funcionamento do *plug-in*.

A arquitetura de *plug-ins* do Eclipse é divida em dois módulos: *"Core"* e *"UI"*. Quando a plataforma é iniciada, o módulo *Core* examina a lista de *plug-ins* disponíveis no diretório e se eles estiverem configurados, são carregados no sistema. Dentro do módulo *Core*, o módulo *runtime* é responsável pela inicialização, funcionamento e gerenciamento dos *plug-ins* dentro do Eclipse. Ainda no módulo *Core*, há o *Workspace*, que administra os recursos do usuário, organizados dentro de um ou mais projetos. O módulo *Core* é genérico e tem um comportamento igual para todos os *plug-ins*, com ou sem interface para o usuário.

O módulo de interface com o usuário (UI) é responsável pela entrada e saída de dados dos *plug-ins*, podendo ser dividido em três grupos: *"Workbench"*, *"JFace"* e *"SWT"*. O *layout* em módulos da arquitetura de um *plug-in* Eclipse possibilita o uso dos recursos dos módulos *"Core"* e interface através das ferramentas "PDE" e "JDT". Detalhes das ferramentas são descritas a seguir:

2.5.1.1 PDE

O *plug-in* Development Environment (PDE) (ECLIPSE, 2008), fornece recursos e facilidades específicas para o desenvolvimento de *plug-ins* para o Eclipse. O PDE pode ser divido em três componentes:

1. PDE UI - componente principal, que agrega as ferramentas para criação, desenvolvimento, teste, depuração e disponibilização do *plug-in* Eclipse;
2. PDE Build - componente que automatiza as funções de compilação e execução do *plug-in*;
3. PDE API Tools - componente API *(Aplication Programming Interface)* que disponibiliza o acesso aos recursos do PDE para desenvolvedores de *plug-ins*.

2.5.1.2 JDT Core Component Development Resources

JDT Core Component Development Resources é a infraestrutura da IDE Java. Implementado como um construtor para a IDE Eclipse permite rodar e depurar códigos

que contêm. erros não resolvidos. Este *plug-in* é necessário para o desenvolvimento e implementação de outras linguagens de programação na IDE Eclipse. É responsável por adicionar marcações personalizadas em códigos, como criar *bookmarks* nos arquivos, indicar a linha que ocorreu o erro e exibir mensagens personalizadas ao usuário no ambiente de desenvolvimento de código.

2.5.1.3 SWT - Standard Widget Toolkit e JFace

O *plug-in* SWT engloba desde janelas, botões, menus, ícones, barras de rolagem, entre outros, até componentes que adicionam ícones nas barras de tarefa do sistema operacional. É utilizado na criação da interface gráfica do EMS.

Ele é uma camada de componentes sobre os componentes padrão do sistema operacional. Por exemplo, um botão SWT é na verdade um botão do sistema operacional, então, se aplicado um tema que mude a cor ou a forma dos componentes, o botão SWT vai se modificar conforme o novo tema. Aplicações SWT sempre possuem a aparência do sistema operacional executado por elas porque utilizam diretamente estes componentes para gerar a visualização na tela.

O SWT foi projetado para oferecer acesso portátil e eficiente para as aplicações da interface do usuário dos sistemas operacionais nos quais é aplicada.

O JFace é um *framework* para a construção de interfaces gráficas construído sobre o SWT. Ele fornece um conjunto de componentes de imagem, textos, caixas de diálogo, etc.

2.6 Trabalhos Relacionados

Ferramentas criadas para melhorar a exibição das mensagens de erro pelos compiladores comprovam, através de experimentos realizados com programadores iniciantes e estudantes de programação, que a qualidade das mensagens de erros interferem diretamente no aprendizado e na qualidade dos softwares desenvolvidos.

Entre as ferramentas existentes, podemos citar Ditran. Seu foco de pesquisa eram estudantes de computação e programadores iniciantes. Para Moulton (MOULTON; MULLER, 1967), todas as mensagens de erro e diagnóstico deveriam exibir as causas do erro em seu contexto e fornecer a maior quantidade de informação possível, indicando ao usuário as direções para correção do erro. Ditran foi utilizada por 180 estudantes, que durante um semestre produziram mais de 10 mil programas. O desempenho foi avaliado por instrutores e professores assistentes que reportaram favoravelmente a efetividade da capacidade de diagnóstico desta ferramenta.

Posteriormente surgiu a CAP (SCHORSCH, 1995). CAP foi uma ferramenta criada

em 1993 para a linguagem Pascal. Tinha como objetivo identificar erros de sintaxe, lógica e estilo, além de promover mensagens de erros mais amigáveis aos estudantes. As mensagens continham informações sobre o erro ocorrido, o porquê do erro e como corrigi-lo além de fornecer exemplos para auxiliar na correção do código. A ferramenta foi testada por 520 estudantes e avaliada por instrutores que notaram um aumento na qualidade do software desenvolvido.

Os instrutores relataram que o número de erros triviais de lógica e sintaxe dos quais os estudantes pediram ajuda ao CAP diminuíram com o tempo. O maior problema citado pelo autor foi que os estudantes na maioria das vezes não liam completamente a mensagem exibida, de forma que objetivo de facilitar o aprendizado não ocorreu como desejado, já que o estudante persistia no erro diversas vezes.

Em 2010, foi criada a Helpmeout (HARTMANN et al., 2010). A ferramenta funciona como um sistema de recomendação social. Tem como objetivo sugerir correções para erros, baseando-se em erros ocorridos anteriormente pelo usuário ou por outros usuários do sistema. Cada erro é identificado por um código. Durante a compilação, quando ocorre um erro em um trecho de código, a ferramenta armazena esse erro em uma base de dados. Na próxima compilação, a ferramenta avalia se o erro foi corrigido e também armazena o código. Essa base de dados *online* fica disponível a todos os usuários da ferramenta. Caso ocorra um erro, o usuário pode consultar o Helpmeout informando o código do erro. O sistema lhe informará uma mensagem de erro (a mesma fornecida pelo compilador), o trecho de código incorreto e um exemplo de código corrigido. A idéia da ferramenta é que, baseando-se em exemplos, o programador possa corrigir o seu código. Apesar de não estar diretamente relacionada à exibição de mensagens de erro, a ferramenta apresenta duas das funcionalidades que foram implementadas no EMS, o armazenamento de trechos de código do usuário para consulta posterior e a apresentação de exemplos de como corrigir o código.

Podemos também citar as ferramentas PC-lint e FlexeLint criadas pela Gimpel (GIMPEL, 1987). Consistem de um analisador de código das linguagens C e C++ que ajudam a localizar erros e falhas no código-fonte dificilmente identificados pelo compilador. A ferramenta conta com uma versão *on-line* em que o usuário pode alterar um dos códigos-fonte disponível e analisá-lo em busca de erros. Também possui opção para consulta dos erros mais ocorridos no mês, com exemplos e explicações de como corrigi-lo.

Outras ferramentas como Alice (DANN; COOPER, 1999) e BlueJ (KOLLING et al., 2003) visam facilitar o aprendizado da programação, através de uma interface visual, onde o programador interage direto com os objetos, evitando assim os erros de sintaxe. Apesar de parecer uma boa ideia, essa abordagem não é recomendada, pois não expõe os programadores às reais mensagens de erro que irão encontrar em sua vida profissional (SAVIDIS, 2007).

Na internet podemos encontrar explicações para diversos erros em grupos de

discussões, *websites* e *blogs*. Um dos *websites* mais conhecido pelos programadores é o *Stack Overflow* (OVERFLOW, 2008). Trata-se de uma ferramenta de perguntas e respostas voltado exclusivamente para programação de computadores. Conta com tópicos sobre diversas linguagens de programação e tecnologias. Os usuários ganham pontos por sua colaboração ao responder as dúvidas de outros usuários e também podem votar na resposta que achar mais pertinente à determinada questão.

2.7 Linguagem Cyan

Cyan é uma linguagem orientada à objetos estaticamente tipada e baseada em protótipos. Assim, não há declaração de classes. Protótipos desempenham o papel de classes e a clonagem e operações de "new" são usados para criar novos objetos. Um protótipo é um objeto do qual outros objetos podem ser criados. Cyan oferece suporte para herança, objetos do tipo *mixin*, *interfaces*, um sistema completo de exceções, funções anônimas estaticamente tipadas, tipagem dinâmica opcional, objetos literais definidos pelo usuário, objetos de contexto, métodos de gramática e envio de mensagens que tornam mais fácil a criação de linguagens específicas de domínio. Uma visão básica da linguagem é dada a seguir.

2.7.1 Declaração de variáveis

As variáveis em Cyan são tipadas. A palavra-chave `var` é utilizada na declaração de variáveis locais. Quando utilizada na declaração de variáveis de instância (atributos de um protótipo) é opcional. No código a seguir

```
String myVariable = "abcd";
```

a variável `myVariable` é declarada como do tipo `String` e tem o valor `"abcd"` atribuído a ela. Já em

```
var meat = Food clone;
```

a variável `meat` é declarada e seu tipo é `Food`, assumindo que `Food` é um protótipo. Em Cyan os valores atribuídos a `meat` podem ser do tipo `Food` ou dos sub-tipos dele, semelhante a herança em classes.

2.7.2 Herança

O sistema de herança de Cyan é similar ao da linguagem Java. Usa-se a palavra chave `extends` para identificar qual protótipo é seu super-objeto. Como no exemplo:

```
object Student extends Person
     private int age
end
```

Se um método é redefinido em um objeto-filho, ele deve vir antecedido da palavra chave `override` que deve aparecer após o qualificador do método. Exemplo:

```
object Person
     private String name
     fun PrintAge {
          Out println: "Name: " + name
     }
end

object Student extends Person
     private int age

     public override fun PrintAge {
          Out println: "Name: " + name + "Age: " + age
     }
end
```

2.7.3 Interfaces

Interfaces em Cyan também são similares às da linguagem Java. Uma interface pode ser criada através da utilização da palavra chave "interface" e contém apenas a assinatura de seus métodos. Todos os métodos declarados em uma interface são considerados públicos, assim, não há necessidade da utilização de qualificadores de acesso. Um protótipo também pode implementar uma interface. Nesse caso, o protótipo deve implementar todos os métodos da interface. Um exemplo de interface é dado abaixo:

```
interface Printable
     fun Printobj
end
```

```
object Student implements Printable
     private int age
     fun Printobj {
          Out println: "Age: " + age
     }

end
```

2.7.4 Any

Todos os protótipos em Cyan são sub-objetos de `Any`, que possui alguns métodos básicos como `eq:` (comparação de referência, não pode ser redefinido em sub-protótipos). Sua negação é `neq:`, `==` (o conteúdo é igual?), `asString` e métodos de reflexão computacional.

2.7.5 Tipos Básicos

Como o código Cyan será direcionado para a máquina virtual Java, os tipos primitivos têm os mesmos nomes da linguagem Java, mas iniciando em letra maiúscula: `Byte, Short, Int, Long, Float, Double, Char` and `Boolean`. Ao contrário de Java, todos os tipos primitivos herdam do protótipo `Any`.

2.7.6 Nil e Tipo Union

Há um tipo especial na linguagem, o tipo "Union". O tipo `Union<A, B>` é considerado, em atribuições e passagens de parâmetros como um super-tipo de `A e B`. Veja o exemplo a seguir:

```
var Union <Int, String> x
x = 0;
x = "Value";
```

A variável `x` pode receber valores do tipo `Int e String`. Para recuperar os valores armazenados em `x` é necessário utilizar o método `unionCase:do:`

```
x
unionCase: Int do: {
     Out println: ("3 - " + x + " = " + 3-x);
}
```

```
unionCase: String do: {
     Out println: ("The value of x is " + x);
}
```

O tipo `Nil` é equivalente ao `null` da linguagem Java. Não é sub-tipo ou super-tipo de qualquer objeto. `Nil` não pode ser atribuído a um objeto cujo tipo é um protótipo. Então ele só pode ser atribuído a uma `Union` que contém `Nil` como parâmetro:

```
var Union <String, Nil> nome;
nome = Nil;
```

2.7.7 Construtores

Construtores possuem o nome de `init` ou `init:` e podem ter um número qualquer de parâmetros. Seu tipo de retorno deve ser `Nil` ou nada. Para cada método chamado `init` ou `init:` o compilador adiciona ao protótipo um método chamado `new` ou `new:` com os mesmos parâmetros, que cria um objeto e chama o método `init` ou `init:` correspondente. Se não houver nenhum dos métodos `init`, `init:`, `new` ou `new:` um método padrão `new` sem nenhum parâmetro é gerado pelo compilador. Um exemplo de utilização dos construtores é dado a seguir:

```
object Person
     fun init: String name { self.name = name }
     private String name
end
     ...
     var p = Person new: "Ana";
```

2.7.8 Estruturas condicionais e de repetição

Como em Cyan tudo é objeto, inclusive `true` e `false`, as instruções `if` e `while` não são necessárias. Podem ser implementadas como envio de mensagem, como no código a seguir:

```
( n%2 == 0 ) ifTrue: { s = "even" } ifFalse: { s = "odd" };
var i = 0;
{^ i < 5 } whileTrue: {
     Out println: i;
     ++i
}
```

Apesar de não serem necessárias, as instruções `if e while` foram implementadas em Cyan.

```
if n%2 == 0 {
     s = "even"
}
else {
     s = "odd"
};
var i = 0;
while i < 5 {
     Out println: i;
     ++i
}
```

2.7.9 Tipagem dinâmica

Apesar de Cyan ser estaticamente tipada, ela também suporta algumas características das linguagens dinamicamente tipadas. Todo identificador seguido pelo caractere ":" é chamado "seletor". É importante ressaltar que não pode haver nenhum espaço entre o identificador e o caractere ":". Uma mensagem cujo seletores são antecedidos do caractere `"?"` não é conferida pelo compilador, isto é, o compilador não verifica se o tipo do receptor tem o método adequado.

```
fun printArray: Array<Any> anArray {
     anArray foreach: { (: Any elem :)
          elem ?printObj
     }
}
```

No exemplo acima, o `Array` pode ter elementos de qualquer tipo e a mensagem `printObj` é enviada a todos eles.

2.7.10 Métodos como objetos

Métodos são objetos também. Assim é possível passar um método como parâmetro. Isso é muito útil na criação de interfaces gráficas, como no exemplo a seguir:

```
object MenuItem
     fun onMouseClick: UFunction<Nil> b {
     ...
     }
end
MenuItem onMouseClick : {File getMethod, "Close"};
```

2.7.11 Considerações finais

Os exemplos citados nesta sessão foram retirados do manual da linguagem Cyan (GUIMARÃES, 2015). São apresentados somente os conceitos necessários para compreensão do conteúdo citados nas próximas sessões. Mais detalhes da linguagem, sua gramática e suas inovações podem ser encontrados no manual da linguagem Cyan (GUIMARÃES, 2015).

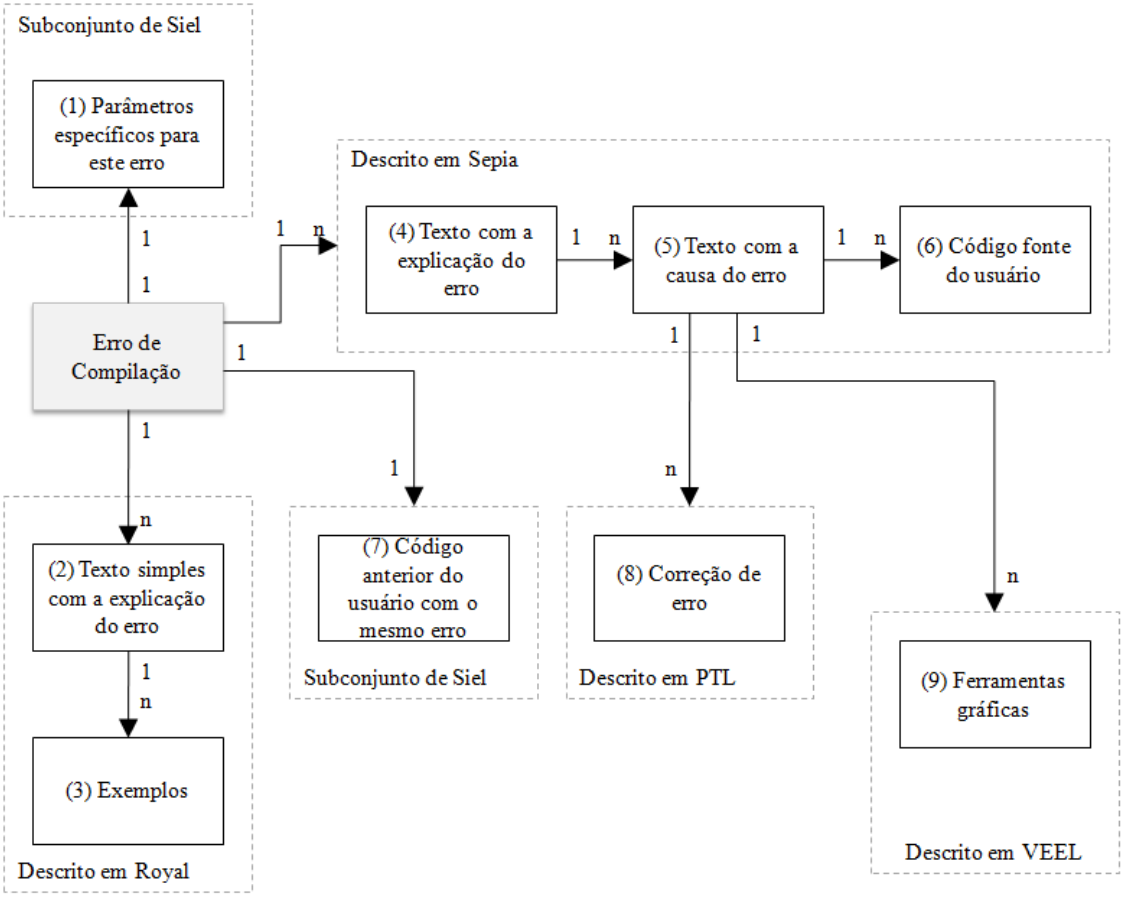
3 *Plug-in* EMS

O presente capítulo apresenta o processo de desenvolvimento do *plug-in* EMS para a plataforma Eclipse. Seu objetivo é solucionar os problemas relacionados por Traver (TRAVER, 2010) na exibição das mensagens de erro dos compiladores. O capítulo não traz detalhes da implementação do código em si, mas sim a descrição dos conceitos e tecnologias empregadas neste. Os detalhes do desenvolvimento e implementação são descritos no Apêndice A.

3.1 Considerações Iniciais

EMS é um grande projeto para exibição e correção dos erros encontrados durante o processo de compilação. A definição das mensagens de erro é feita através das linguagens específicas de domínio Siel, Sepia e Royal. A Figura 5 apresenta o modelo entidade relacionamento do projeto, que é descrito a seguir.

Figura 5 – Diagrama de relacionamento do *plug-in* EMS



(1) Para cada erro pode haver um ou mais parâmetros/atributos. Estes parâmetros fornecem informações sobre o ambiente em que o erro ocorreu. Por exemplo: o nome do método, lista de variáveis, trecho do código onde o erro ocorreu etc.

(2) Uma ou mais mensagens com uma explicação resumida do erro. Esta mensagem é semelhante a que usualmente é exibida pelos compiladores.

(3) Para cada explicação estão relacionados zero ou mais exemplos de como corrigir o erro.

(4) Um ou mais textos com a explicação do erro, sendo que um deles é o padrão. Ao acontecer o erro, será apresentada ao programador uma explicação usando os trechos do código de usuário presentes naquela instância do erro.

(5) Para cada explicação haverá zero ou mais textos explicando o que ocasionou o erro. Cada erro de compilação pode ter várias causas diferentes.

(6) Zero ou mais trechos de código-fonte do usuário que está utilizando o sistema. Estes trechos mostram as partes do código que causaram ou poderiam ter causado o erro.

(7) Zero ou um texto com o código do usuário que apresentou o mesmo erro anteriormente.

(8) Zero ou mais correções para o erro. Se disponível, o programador poderá aceitar uma das sugestões fornecidas pelo compilador para corrigir o erro.

(9) Zero ou mais trechos de código nos quais as partes importantes são realçados com círculos, ligações, setas etc. Estes elementos gráficos poderão não só indicar as causas do erro como também mudanças no código para corrigi-lo.

Os item 1 e 7 constituem o subconjunto descrito pela LED Siel. Os itens 2 e 3 são descritos pela LED Royal, 4, 5 e 6 em Sepia, 8 em PTL e 9 em VEEL.

O desenvolvimento do *plug-in* envolveu diversas áreas da computação. A Tabela 3 relaciona as tarefas envolvidas para o desenvolvimento do EMS e a área de estudo relacionada.

Por se tratar um projeto multidisciplinar e pela extensão do tema, somente a parte do EMS que trata a exibição da mensagens de erro foi desenvolvido nesta dissertação. Sendo assim, a correção de erros de compilação (8) e elementos gráficos para ligações visuais entre trechos do código (9), é proposta em trabalhos futuros.

EMS é um *plug-in* desenvolvido na linguagem Java e incorporado na IDE Eclipse. É totalmente configurável através de linguagens específicas de domínio. A utilização das LEDs visa a portabilidade e a colaboração, já que permite a troca de informações mais facilmente entre os usuários; ou seja, um usuário pode passar suas mensagens de erro

Tabela 3 – Áreas envolvidas no desenvolvimento do projeto

Área de atuação	**Tarefas envolvidas**
Linguagens de Programação	Desenvolvimento de linguagens específicas de domínio
Compiladores	Criação de compiladores para as LEDs, alteração das mensagens de erro do compilador
Engenharia de Software	Desenvolvimento do *plug-in* e das ferramentas utilizadas no *plug-in*
Interação Humano Computador	Padrões para mensagens de erro, sistema cognitivo, testes com usuários

para outro usuário, sem ser necessário que este escreva suas próprias mensagens. As LEDs também permitem que as mensagens de erro sejam escritas através de qualquer editor de texto e posteriormente sejam implantadas ao *plug-in*.

A fim de favorecer a acessibilidade e a usabilidade, uma interface gráfica foi desenvolvida. Essa interface permite que os usuários escrevam suas mensagens de erro e as incorporem ao compilador, sem que seja necessário conhecimento específico das LEDs.

Para o desenvolvimento do EMS foram levados em consideração as propostas de Traver (TRAVER, 2010) para resolução dos problemas com as mensagens de erro. Na Tabela 4 são relacionadas as soluções descritas por Traver e a implementação correspondente no *plug-in* EMS.

Tabela 4 – Comparativo da proposta de Traver (TRAVER, 2010) com o EMS.

Proposta de Traver	**Solução proposta pelo *plug-in* EMS**
Armazenar trechos de código com erro, sua explicação e possível correção.	Armazenar o código do usuário com erro para posterior consulta. Exibição de exemplos de como corrigir o erro.
Sugerir as causas do erro, detectar a origem exata do erro e sua localização correta.	Sinalização do local do erro, exibição das possíveis causas do erro e exemplo de como corrigi-los.
Fornecer mensagens de diagnóstico personalizadas e permitir que novas mensagens de diagnóstico sejam incorporadas ao compilador, sem que o usuário tenha a necessidade de escrevê-las.	Mensagens incorporadas ao compilador através de Linguagens Específicas de Domínio, o que permite a troca das mensagens entre os usuários. Ferramenta gráfica para geração das mensagens de erro, sem que o programador necessite escrever código.

O desenvolvimento de um *plug-in* com as características propostas na Tabela 4, envolveu uma sequência de ações para que o objetivo deste trabalho fosse alcançado. Essas ações são descritas abaixo:

a) definição das LEDs: Siel, Royal e Sepia;

b) desenvolvimento dos compiladores para as linguagens Siel, Royal, Sepia;

c) criação de interface gráfica para Siel, Royal e Sepia;

d) criação do método signalCompilerError para sinalização do erro, exibição das mensagens e de códigos anteriores com o mesmo erro.

e) geração de um *plug-in* para a IDE Eclipse;

Os nomes das LEDs Royal e Sepia foram definidos arbitrariamente e não possuem significado diretamente ligado ao EMS. O nome Siel vem do inglês *Specific information error language* e significa "linguagem de informação específica do erro".

O compilador da linguagem Cyan (GUIMARÃES, 2015) foi utilizado para desenvolvimento e validação do *plug-in* EMS. O compilador é implementado em Java, a mesma linguagem que foi utilizada para fazer o *plug-in* para o Eclipse.

Cyan é uma linguagem de programação orientada a objetos e baseada em protótipos. Em Cyan não há declaração de classe como nas linguagens orientadas a objetos baseadas em classes. A linguagem permite declarar protótipos a partir dos quais outros objetos são criados. A linguagem Cyan não tem como objetivo se tornar uma linguagem comercial, foi criada para servir de base para a criação de novas construções de linguagens. Algumas características da linguagem Cyan são descritas na próxima sessão.

O compilador da linguagem Cyan não está pronto, mas há um subconjunto dele que já produz mensagens de erro suficientes para este projeto.

Para que o EMS pudesse utilizar o compilador da linguagem Cyan, foi necessária a adaptação de um editor para a linguagem e acoplar o compilador da linguagem Cyan a IDE Eclipse. Para isso foram necessários:

a) desenvolvimento de interface que para criação e gerenciamento de projetos na linguagem Cyan;

b) desenvolvimento de um editor para reconhecimento da sintaxe da linguagem;

c) inclusão do compilador da linguagem Cyan na IDE do Eclipse;

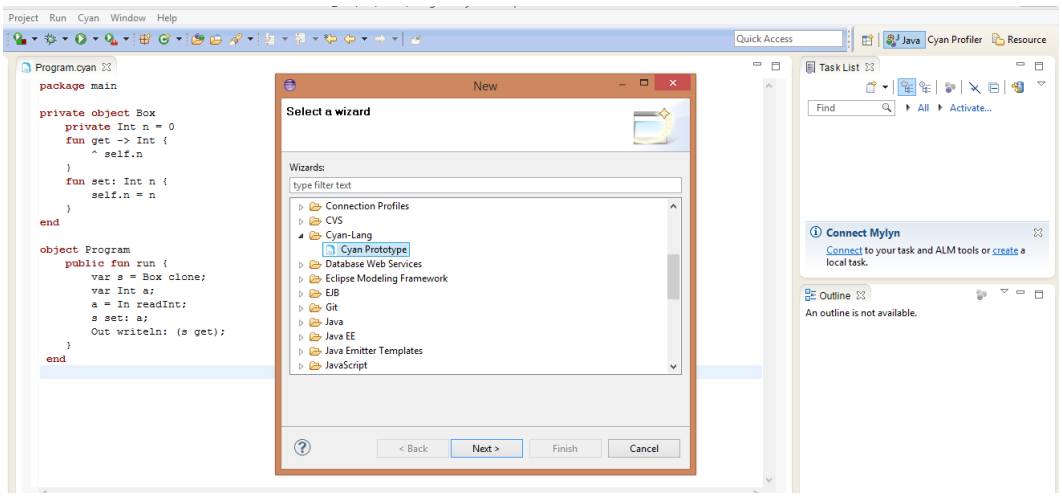
d) geração do *plug-in* Cyan com as funcionalidades acima descritas.

Após criação do *plug-in* para Cyan o mesmo foi incorporado ao ambiente Eclipse. A Figura 6 apresenta o *plug-in* Cyan na IDE Eclipse. A inclusão do *plug-in* na IDE Eclipse, permitiu que novas funcionalidades fossem adicionadas à IDE. Por exemplo, a criação de projeto do tipo Cyan, que permite o reconhecimento da sintaxe da linguagem, a criação de arquivos/protótipos Cyan, opção para compilar arquivos Cyan, entre outras funcionalidades.

Várias ferramentas foram utilizadas para a criação do *plug-in* da linguagem Cyan e sua vinculação ao Eclipse. Os detalhes da implementação são descritos no Apêndice A.

Figura 6 – *Plug-in* Cyan incorporado ao Eclipse

.



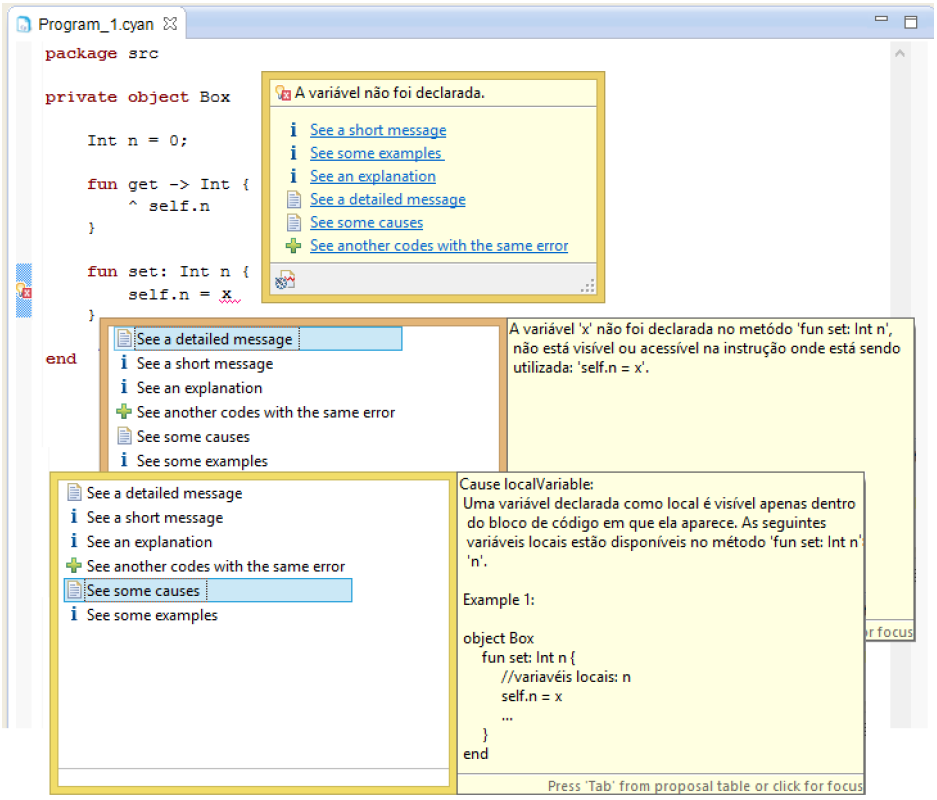
É importante ressaltar que não há nada específico na arquitetura geral de EMS em relação a Cyan. Qualquer linguagem que seja compilada na IDE Eclipse pode utilizar o *plug-in*. Para isso, é necessário que o EMS esteja na pasta de *plug-ins* do Eclipse, o compilador da linguagem faça uma referência ao *plug-in* em sua classe responsável pela emissão das mensagens de erro e uma chamada ao método `signalCompilerError` seja realizada, passando como parâmetros ao método as informações sobre a instância em que o erro ocorreu. Assim, este projeto pode não só ser adaptado a outras linguagens como também a outros sistemas que sinalizam erros que podem ser difíceis de entender, ou em sistemas que desejam favorecer o aprendizado dos estudantes.

3.2 Arquitetura do *plug-in* EMS

A ferramenta EMS foi desenvolvida em forma de *plug-in* para ser executada na mesma camada de desenvolvimento de programas, ou seja, não é necessário ao programador sair do ambiente que está trabalhando para obter ajuda sobre o erro que está ocorrendo. O *plug-in* altera as mensagens exibidas pela IDE para as mensagens configuradas no EMS no momento da compilação. A Figura 7 apresenta algumas das opções do usuário utilizando o *plug-in* EMS, no momento que o compilador detecta um erro.

O EMS funciona da seguinte maneira: quando um erro é sinalizado, o compilador armazena algumas informações para compor a mensagem de erro a ser emitida ao programador. Essas informações são enviadas ao EMS, que emite uma mensagem curta e oferece outras opções de ajuda: consultar outras mensagens curtas, consultar uma mensagem detalhada utilizando trechos do código do usuário, consultar exemplos, causas do erro ou um código do usuário que apresentou o mesmo erro. Conforme pode ser visto no retângulo

Figura 7 – Exemplo de Mensagem de erro exibida pelo *plug-in* EMS



mais ao alto na Figura 7.

É possível observar que as janelas do EMS estão no idioma inglês e as mensagens em português porque é possível ao programador escrever em qualquer idioma e definir qual a linguagem será a padrão das mensagens exibidas.

A partir de uma interface gráfica, é possível a geração de códigos para as LEDs Siel, Sepia, Royal. Esses códigos possuem informações sobre as mensagens de erro a serem vinculadas ao compilador. O desenvolvimento de código para as LEDs não está necessariamente ligado à ferramenta gráfica. O programador pode escrever seus códigos em qualquer editor de texto e colocá-los na pasta correspondente do EMS.

Esse método `signalCompilerError` possui duas funções: receber as informações do compilador e enviar para o EMS e sinalizar a linha onde ocorreu o erro, exibindo as opções de mensagens disponíveis. A Figura 8 apresenta um trecho do código enviado pelo compilador da linguagem Cyan ao método `signalCompilerError` do EMS.

O ciclo de funcionamento do EMS é apresentado na Figura 9. Ao sinalizar um erro (1), o compilador envia uma mensagem para o *plug-in* EMS através do método `signalCompilerError` (2). Essa mensagem contém um conjunto de atributos/valores que serão utilizados na composição da mensagem de erro, como o nome do método onde ocorreu o erro, nome das variáveis, lista de variáveis de instância, protótipo etc. Esse conjunto

Figura 8 – Exemplo de mensagem enviada a signalCompilerError

```
signalCompilerError( "error = 'Variable was not
declared'",

"variable = '" + lexer.currentIdentifier() + "'",

"statementText = '" +
currenteMethod.currentStatement() + "'",

"method = '" + currentMethod.getInterface() + "'",

"prototype = '" + currentePrototype.getName() + "'",

"visibleLocalVariables = { " +

currenteMethod.getVisibleVariablesList() +" }",

"instanceVariables = " +

currentPrototype.getInstanceVariableList() + "'" );
```

de atributo/valor é o que define a linguagem **Siel**. O compilador envia, além do código em Siel, o código do usuário com o erro. Esse código é armazenado pelo EMS e pode ser consultado posteriomente (3).

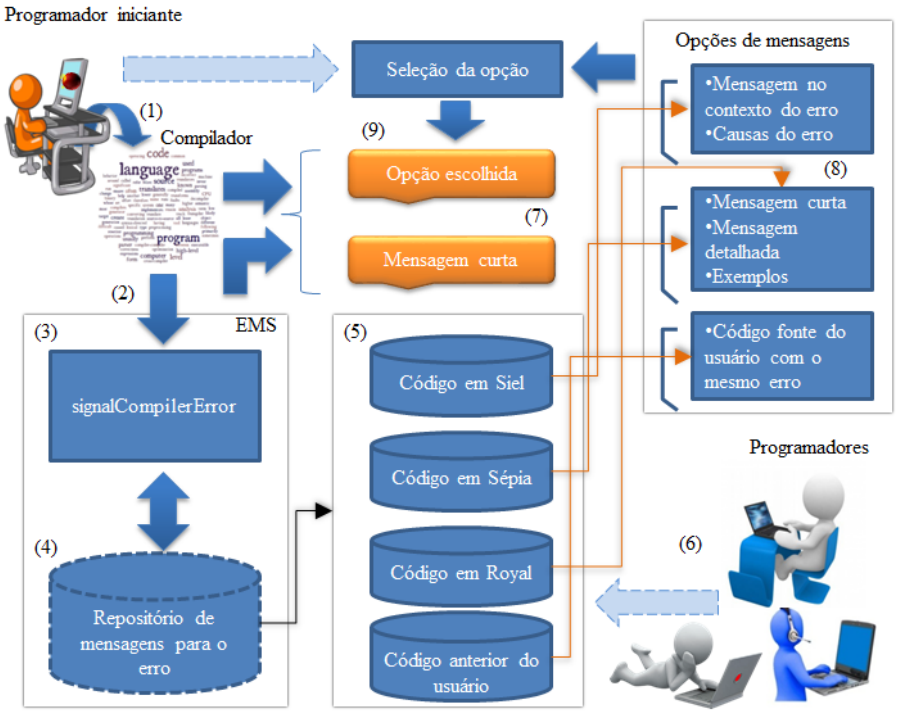
O método `signalCompilerError` realiza uma consulta ao depósito de mensagens de erro (5) que contém os arquivos com códigos em **Siel**, e confere se os parâmetros enviados pelo método correspondem aos parâmetros esperados para o erro. O repositório também contém os códigos em **Sepia**, **Royal** e os códigos anteriores do usuário sobre o mesmo erro (5).

As mensagens constantes deste depósito são escritas por programadores mais experientes e podem ser modificadas e/ou ampliadas pelo próprio programador (6). Os códigos em Sepia e Royal podem ser escritos por programadores mais experientes, possivelmente com a ajuda da interface gráfica.

Em seguida, o EMS sinaliza a linha em que ocorreu o erro e emite uma mensagem curta (7), exibindo então outras opções de ajuda adicional (8). O usuário pode então escolher ver outra mensagem curta, uma mensagem mais detalhadas e alguns exemplos. Respectivamente “See a short message”, “See an explanation” e “See some examples” na Figura 7. As informações utilizadas para compor essas mensagens são fornecidas pela LED **Royal**.

O EMS também fornece ao usuário a opção de visualizar uma mensagem adaptada ao contexto do erro e suas possíveis causas. Essa mensagem é composta utilizando os parâmetros fornecidos pela linguagem Siel. A linguagem responsável por essas informações é denominada **Sepia**. O código em Sepia utiliza os nomes dos parâmetros que serão substituídos pelos valores fornecidos pela linguagem Siel. Desta forma, quando o compilador

Figura 9 – Fluxo de informações entre o compilador, EMS e a interface do usuário



sinalizar um erro no código, ele apresentará uma mensagem utilizando trechos do código fonte do programador. Assim o programador terá a impressão que o seu próprio código está sendo apresentado na mensagem do erro. O envio das informações pelo compilador é feito de forma automática (2), pois essas informações são passadas para `signalCompilerError` usando **Siel**.

As próximas seções descrevem a gramática, o funcionamento das LEDs e suas funções e as demais funcionalidades do *plug-in* EMS.

A gramática das LEDs possuem características que são comuns entre elas. As LEDs são analisadas a partir do desdobramento do conjunto de "Program". As palavras reservadas da linguagem estão descritas entre aspas. O conteúdo entre { e } pode ser repetido zero ou mais vezes. LeftCharString é qualquer seqüência dos caracteres

```
= ! < % & * - + ^ ~ ? / : . \ | [ { <
```

começando com `[, { ou <`. RightCharString é a seqüência espelho dos mesmos símbolos utilizados em LeftCharString. Por exemplo, se os caracteres (LeftCharString) `<<*` são utilizados para iniciar um bloco, o compilador irá checar se `*>>` é utilizado (RightCharString) no fechamento do bloco. TEXT é um terminal composto por quaisquer símbolos.

3.2.1 Siel

Siel é uma LED que define os parâmetros específicos para o erro e seus respectivos valores. É basicamente uma lista de pares atributo/valor. Contém informações sobre o erro de compilação e o ambiente onde o erro aconteceu. Por exemplo, o erro "Variable was not declared" está associado à informação do nome da variável, o método onde ela está sendo utilizada e o protótipo no qual o método está. Estas informações são os parâmetros específicos para este erro. Cada tipo de erro tem os seus parâmetros específicos que podem ser os nomes do método e protótipo em que o erro foi sinalizado, a instrução do erro, o nome do super-protótipo etc.

Essas informações são enviadas pelo compilador da linguagem Cyan para o *plug-in* EMS que verifica se o conteúdo enviado pelo compilador corresponde ao código Siel existente para a o erro e, posteriormente, utiliza os valores fornecidos para composição das mensagens de erro. A gramática da LED Siel é descrita a seguir.

Program ::= { Atrib }
Atrib ::= Id "=" LiteralString ","

Cada erro possui um número variável de atributos/valores. A Figura 10 apresenta o código Siel para o erro "Variable was not declared" enviado pelo compilador para o método `signalCompilerError`.

Figura 10 – Exemplo de código Siel

```
1  error = "Variable was not declared",
2  variable = "a",
3  statementText = "a = b",
4  method = "search: (String s);",
5  prototype = "PersonList",
6  instanceVariables = "personArray"
```

Os código em Siel para outros erros são similares a estes podendo requerer atributos diferentes. Por exemplo, para o erro "ponto e vírgula ausente" não é necessário o atributo *"instanceVariables"* que descreve as variáveis de instância do protótipo compilado. Mas certamente seria necessário o atributo *"lastStatement"* que é uma `string` com o texto da última instrução.

O compilador para a LED Siel difere dos outros compiladores pois não realiza a transformação do código em Siel em outra linguagem de baixo nível. Este tem como objetivo reconhecer e incorporar o código Siel ao *plug-in* EMS.

A incorporação do código da LED Siel ao EMS ocorre através do envio de mensagem pelo compilador da linguagem Cyan ao método `signalCompilerError`. Ao receber o código

da LED o EMS inicia então a chamada ao compilador Siel. Durante a compilação acontece a análise de erros e inconsistências no código-fonte relacionados à LED Siel. O tratador de erros retorna mensagens indicando a linha e o tipo de erro e, enquanto estes persistirem, a fase de incorporação ao EMS não é processada.

Após o processo de compilação o EMS verifica então se os parâmetros enviados pelo compilador da linguagem Cyan correspondem ao erro. Por exemplo, para o erro "Variable was not declared" o EMS verifica se "variable" consta na lista de parâmetros/atributos do erro. Veja na Figura 10. Esta lista de parâmetros pode ser gerada manualmente seguindo a gramática da LED ou automaticamente através da ferramenta "ErrorList".

ErrorList é uma ferramenta incorporada ao *plug-in* da linguagem Cyan e foi desenvolvida para permitir a extração das informações de uma classe Java que seja do tipo Enum e contenha o nome dos erros e seus respectivos parâmetros. No compilador da linguagem Cyan essa classe recebe o nome de *ErrorKind* conforme Figura 11.

Figura 11 – Classe *ErrorKind*

ErrorKind.java

```
public enum ErrorKind {

    variable_was_not_declared("Variable was not declared", new String[] {
            "identifier", "statementText", "methodSignature", "prototypeName",
            "supertype", "implementedInterfaces", "visibleLocalVariableList",
            "instanceVariableList", "methodList" }),

    parameter_is_being_redeclared("Parameter is being redeclared",
            new String[] { "identifier", "statementText", "methodSignature",
                    "prototypeName", "supertype", "implementedInterfaces",
                    "visibleLocalVariableList", "instanceVariableList",
                    "methodList" }),

    local_variable_is_being_redeclared("Variable is being redeclared",
            new String[] { "identifier", "statementText", "methodSignature",
                    "prototypeName", "supertype", "implementedInterfaces",
                    "visibleLocalVariableList", "instanceVariableList",
                    "methodList" }),

    identifier_was_not_declared("Identifier was not declared", new String[] {
            "identifier", "statementText", "methodSignature", "prototypeName",
            "supertype", "implementedInterfaces", "visibleLocalVariableList",
            "instanceVariableList", "methodList" }),

    identifier_expected_inside_method("Identifier expected", new String[] {
            "identifier", "statementText", "methodSignature", "prototypeName",
            "supertype", "implementedInterfaces", "visibleLocalVariableList",
            "instanceVariableList", "methodList" }),

    identifier_expected_outside_method("Identifier expected", new String[] {
            "identifier", "prototypeName", "supertype",
            "implementedInterfaces", "instanceVariableList", "methodList" }),

    identifier_expected_outside_prototype("Identifier expected",
            new String[] { "identifier", }),
```

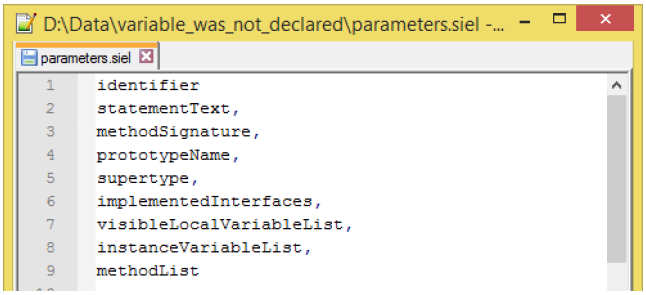
Fonte: Retirado do compilador da linguagem Cyan (GUIMARÃES, 2015)

.

Um arquivo com a extensão ".siel" é então gerado para cada valor enumerado

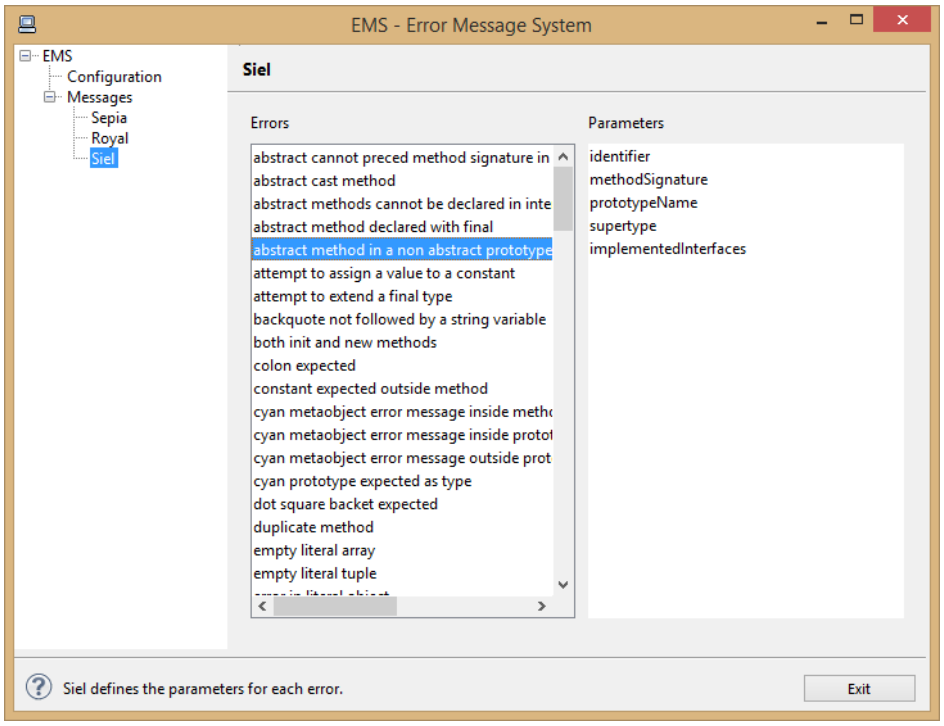
da classe que corresponde a um erro. A Figura 12 apresenta o arquivo gerado para o erro “Variable was not declared”. Este arquivo é salvo na pasta correspondente ao erro, previamente configurada no EMS e contém a relação de parâmetros/atributos separados por vírgula.

Figura 12 – Relação de parâmetros/atributos do erro



O *plug-in* EMS conta com uma interface gráfica para a LED Siel que pode ser vista na Figura 13. Como a geração do arquivo contendo os parâmetros é feita de forma automá-

Figura 13 – Interface gráfica para a LED Siel



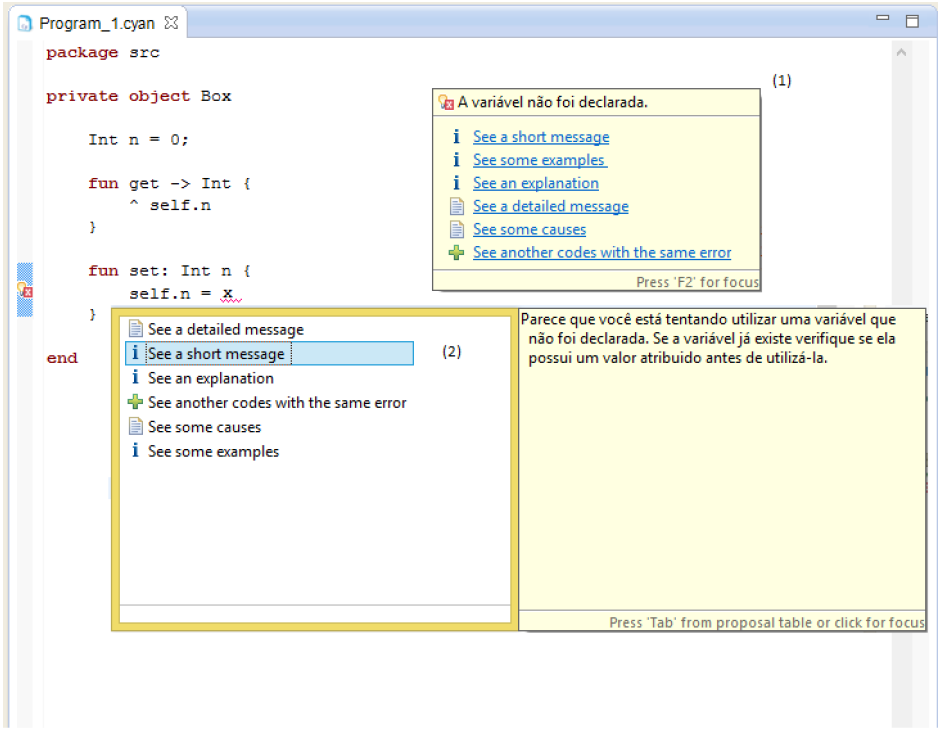
tica, essa interface foi desenvolvida apenas para a visualização do parâmetros/atributos de cada erro. O conteúdo exibido na janela da direita é a relação dos parâmetros necessários para o erro selecionado na janela da esquerda.

3.2.2 Royal

De acordo com Schorsch (SCHORSCH, 1995), os programadores não costumam ler completamente a mensagem exibida, sendo fundamental, portanto, que as possíveis causas do erro sejam colocadas logo no início da mensagem. Assim, a LED Royal foi definida para exibir uma mensagem curta para explicar o erro e fornecer exemplos de como corrigi-lo.

Quando ocorre um erro de compilação, o *plug-in* EMS sinaliza a linha do código em que o erro ocorreu, como mostra a Figura 14. O EMS adiciona ao marcador [1] uma mensagem curta que explica o erro e é visualizada com o passar do *mouse* (1). O programador possui também a opção de visualizar outra mensagem curta, caso exista mais de um código Royal para o erro (2).

Figura 14 – Mensagem curta exibida pelo compilador

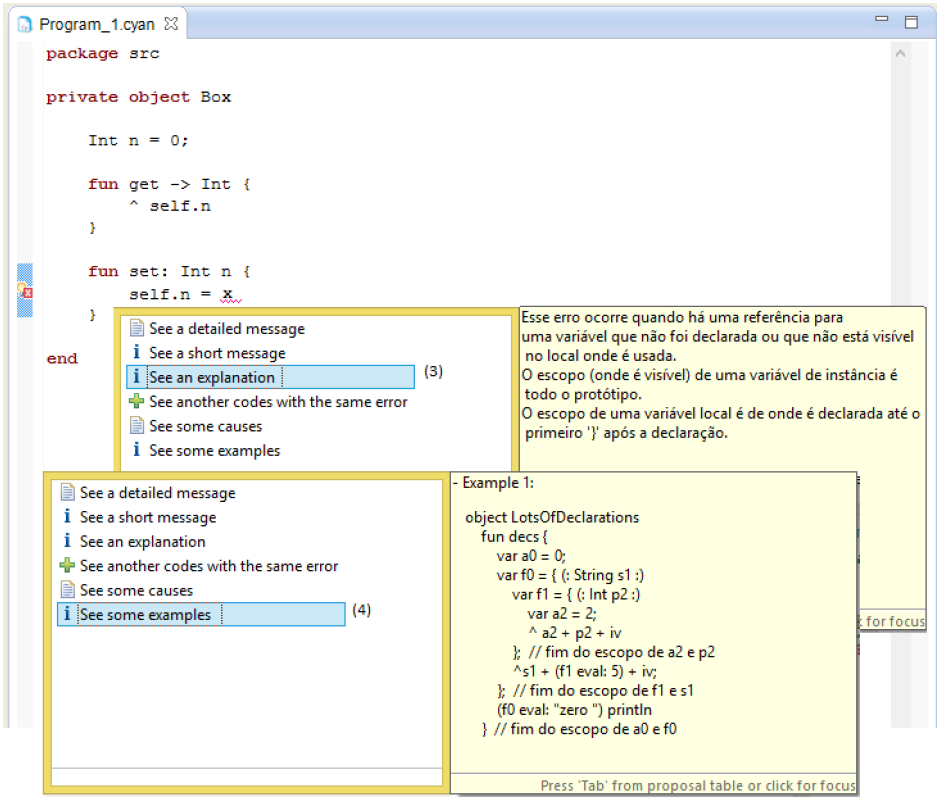


Na Figura 15 pode-se observar outras opções da LED Royal: consultar uma mensagem detalhada (3) e exemplos de como corrigir o código (4).

Com o intuito de facilitar a usabilidade da LED Royal suas palavras-chave foram definidas no idioma inglês, idioma padrão utilizado na computação. A gramática da LED Royal é descrita a seguir.

[1] Um marcador em Java é um simbolo gráfico utilizado para sinalizar uma linha.

Figura 15 – Explicação e exemplos exibidos pelo compilador



Program ::= “use” “:” (“true” | “false”)
“language” “:” Str { “,” Str }
“errorDescription” LeftCharString TEXT RightCharString
“text” LeftCharString TEXT RightCharString
{ “source” LeftCharString TEXT RightCharString }

Cada erro de compilação está associado a uma mensagem curta e a um texto que o explica. E cada um destes textos pode estar associado a zero ou mais exemplos. Veja um exemplo na Figura 16. É possível determinar se a mensagem é a padrão a ser exibida ou não (`use`). Se houver mais de uma mensagem definida como padrão, o EMS escolhe arbitrariamente qual delas será exibida. Caso não esteja configurada como padrão (`use: false`) a mensagem é mantida no repositório e é ignorada pelo EMS na sinalização do erro. Pode-se também observar a definição do idioma (`language`) da mensagem, a descrição do erro (`errorDescription`), a explicação do erro (`text`) e exemplos de código com erro e sua correção (`source`). Uma mensagem pode possuir mais de um exemplo de código com erro e sua correção, sendo necessário para isso somente a utilização da palavra `"source"`.

Na configuração do EMS é possível determinar qual o idioma das mensagens a serem exibidas; ou seja, se o usuário optar pelo idioma português, somente as mensagens com `language: "Português"` serão exibidas pelo compilador.

Figura 16 – Código na linguagem Royal

```
use: true
language: "Português", "Portuguese"

errorDescription<<< A variável não foi declarada.>>>

text<<<* Esse erro ocorre quando há uma referência para
uma variável que não foi declarada ou que não está visível no local onde é usada.
O escopo (onde é visível) de uma variável de instância é todo o protótipo.
O escopo de uma variável local é de onde é declarada até o primeiro '}'
após a declaração. *>>>

source<**
    object LotsOfDeclarations
        fun decs {
            var a0 = 0;
            var f0 = { (: String s1 :)
                var f1 = { (: Int p2 :)
                    var a2 = 2;
                    ^ a2 + p2 + iv
                };  // fim do escopo de a2 e p2
                ^s1 + (f1 eval: 5) + iv;
            };  // fim do escopo de f1 e s1
            (f0 eval: "zero ") println
        }  // fim do escopo de a0 e f0

        Int iv   // pode ser usado em todo o protótipo
    end
**>
```

O código em Royal é compilado através do compilador desenvolvido para a linguagem. Assim como na LED Siel, o compilador da LED Royal se difere de um compilador tradicional pois não realiza a fase de geração de código em uma linguagem de baixo nível. O compilador de Royal gera objetos com as informações dos códigos. Esses objetos são utilizados pelo EMS na composição das mensagens de erro.

Durante a compilação acontece a verificação de erros e inconsistências no código-fonte relacionados a LED Royal. O tratador de erros retorna mensagens indicando a linha e o erro ocorrido e, enquanto estes persistirem, a fase de incorporação ao EMS não é processada.

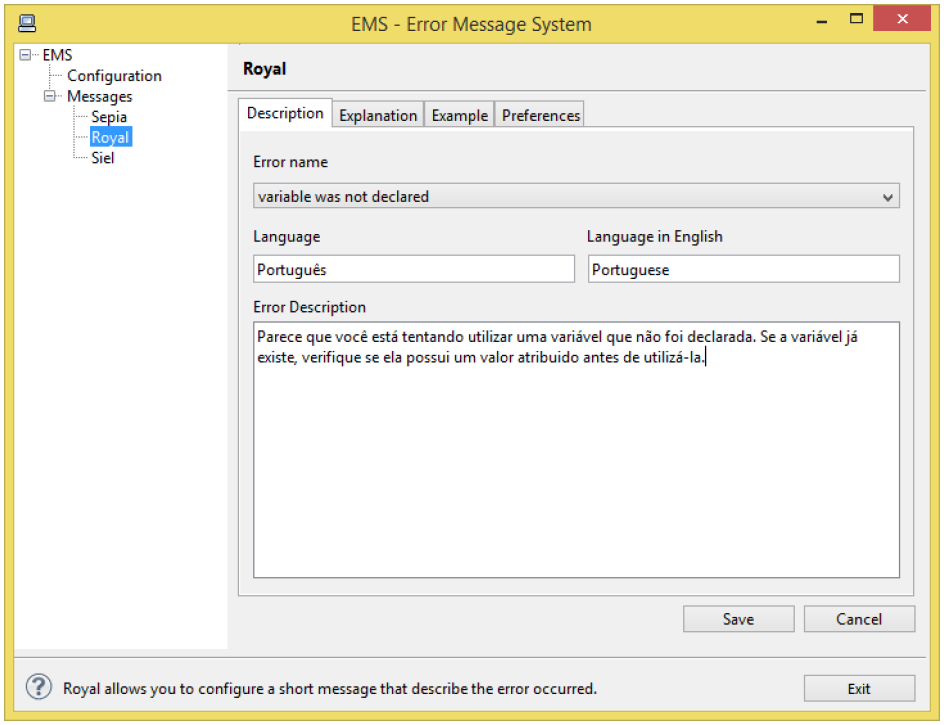
O programa-fonte deve ser escrito em um arquivo com a extensão ".royal". Este arquivo deve ser colocado na pasta referente ao erro e previamente configurada no EMS. Pode haver mais de um arquivo para o mesmo erro, sendo um desses o padrão para exibição.

A chamada ao compilador da LED Royal pode ocorrer de duas formas: primeiro quando o EMS acessa o código da LED para composição das mensagens de erro (no início da compilação), segundo quando a ferramenta gráfica para a linguagem acessa o arquivo contendo o código fonte da LED.

A interface gráfica da LED Royal permite que o usuário componha códigos na linguagem Royal sem que seja necessário escrever uma linha de código; ou seja, não é necessário que ele conheça a sintaxe da linguagem para configurar suas mensagens. O usuário fornecerá a explicação, idioma, exemplos de código etc e a ferramenta produz o

código em Royal automaticamente. A Figura 17 apresenta a interface para geração do código na linguagem Royal.

Figura 17 – Interface para criação de código em Royal



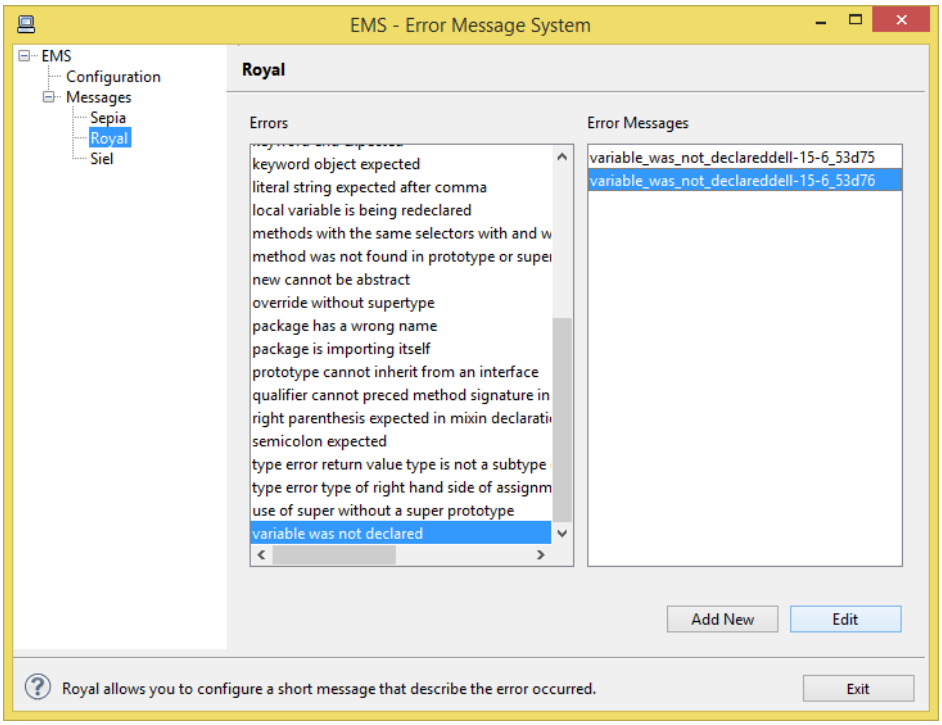
A interface é estruturada em abas. Cada aba se refere a um bloco ou sub-bloco da linguagem. Na aba "Description" o usuário escolhe o erro para o qual deseja configurar a mensagem. A lista de erros é extraída da classe `"ErrorKind"` do compilador Cyan. Detalhes da implementação são descritos no Apêndice A. A seguir, o usuário preenche o idioma da mensagem (`language`) e uma descrição do erro (`errorDescription`). Na aba "Explanation" a opção para explicação do erro (`text`) e em "Examples" os exemplos para correção (`source`). Na guia "Preferences" é possível definir o nome do arquivo e se a mensagem será ou não a padrão a ser utilizada (`use`).

Também é possível gerenciar os códigos em Royal associados a cada erro, o que pode ser visto na Figura 18. A janela da esquerda apresenta a relação de erros existentes. A janela da direita apresenta as mensagens as mensagens existentes para o erro. Ao selecionar uma mensagem e clicar em "Edit" o compilador da LED Royal é chamado e verifica o código da mensagem. Se não ocorrer nenhum erro, o código Royal é incorporado à interface gráfica e exibido conforme a Figura 17.

3.2.3 Sepia

Quando ocorre um erro de compilação, é apresentada ao programador uma mensagem de erro adaptada àquele trecho do código fonte que causou o erro. Por exemplo,

Figura 18 – Interface para gerenciamento de códigos em Royal



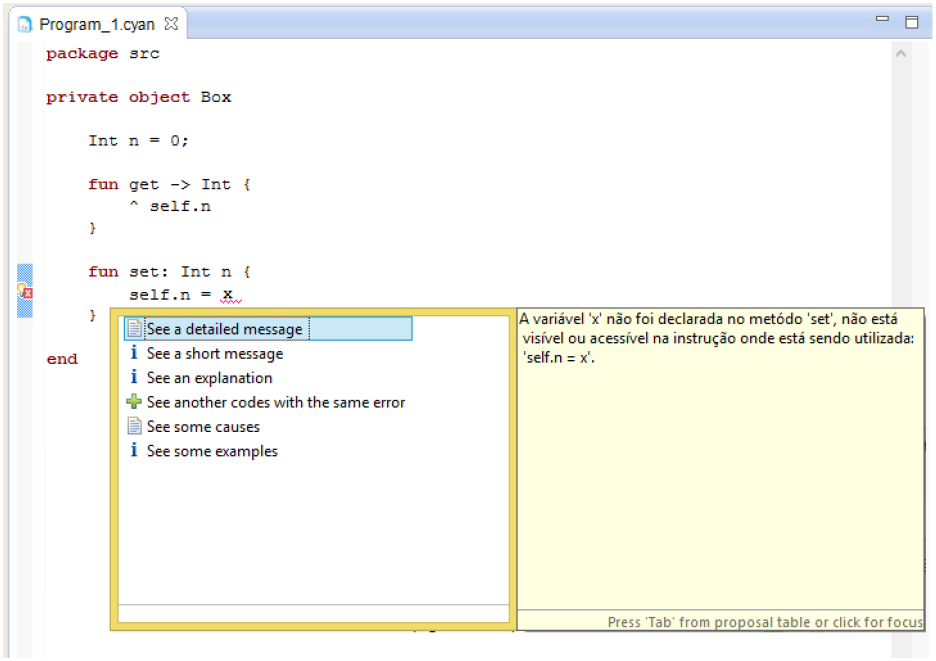
ao invés de o compilador sinalizar "variável não declarada" ele apresentará a mensagem "A variável *"x"* não foi declarada no método *"set:"*, que pode ser vista na Figura 19. Para isso, a LED Sepia utiliza os parâmetros fornecidos pela linguagem Siel e passados a `signalCompilerError` compondo a mensagem a ser exibida ao usuário. Em outras palavras, quando o compilador detectar que a variável *"x"* em `self.n = x` não foi declarada, ele chama o método `signalCompilerError` passando como parâmetro o código em Siel contendo informações sobre o nome da variável não declarada, o método e o protótipo onde o erro foi detectado etc.

Assim como na LED Royal as palavras-chave da LED Sepia foram definidas utilizando palavras no idioma inglês. A gramática da LED Sepia é apresentada a seguir.

Program ::= "parameters" "(" IdList ")" "explanation" LeftCharString TEXT
RightCharString { Cause }
Cause ::= "cause" Id { Explanation { Source } }
Explanation ::= "explanation" LeftCharString TEXT
RightCharString
Source ::= "source" LeftCharString TEXT RightCharString
IdList ::= Id { "," Id }

As etiquetas presentes em IdList devem ser identificadores existentes na lista de parâmetros fornecida pela LED Siel.

Figura 19 – Mensagem detalhada exibida pelo compilador



A Figura 20 apresenta um trecho do código escrito na LED Sepia. A primeira linha do código em Sepia define se a mensagem deve ou não ser exibida `(use)`, seguido pelo idioma da mensagem `(language)`. Em `(parameters)` a lista de parâmetros que podem ser utilizados para a composição da mensagem de erro. Em seguida, a mensagem com a explicação do erro `(explanation)`. Os parâmetros/atributos iniciados com o caractere "#" são substituídos pelos valores passados pelo compilador no método `signalCompilerError` através da LED Siel. Isto faz com que o usuário tenha a impressão que seu próprio código está sendo exibido na mensagem, o que pode ser visto na Figura 21. As possíveis causas do erro `(cause)`, também utilizam estes parâmetros/atributos para a composição das mensagens, o que em conjunto com os exemplos, facilita a identificação, prevenção e correção do erro.

Cada erro de compilação pode ter várias causas diferentes. Por exemplo, uma atribuição `self.n = x` pode fazer o compilador sinalizar o erro "Variável não declarada". Há várias causas para este erro como, a variável não foi declarada localmente, a variável não foi declarada como variável de instância, a variável foi digitada incorretamente. Apesar de estas causas serem apenas variações de uma mesma, utilizaremos este exemplo pela sua simplicidade de entendimento. Assim, definiu-se que cada causa deve ser seguida pelo seu identificador, por exemplo: *"cause localVariable"*, *"cause"* é terminal e *"localVariable"* o identificador. Cada causa possui sua explicação `(explanation)` e exemplos de como corrigir o erro `(source)`. A Figura 21 apresenta a mensagem exibida ao usuário para visualização das causas do erro.

Um compilador foi desenvolvido para Sepia. Assim como nas LEDs Siel e Royal o

Figura 20 – Código na linguagem Sepia

```
use: true

language: "Português", "Portuguese"

parameters(identifier, statementText, methodSignature, prototypeName,
            supertype, implementedInterfaces, visibleLocalVariableList,
            instanceVariableList, methodList)

explanation<<* A variável '#identifier' não foi declarada no metódo
            '#methodSignature', não está visível ou acessível na
            instrução onde está sendo utilizada: '#statementText'. *>>

cause localVariable {
    explanation<<* Uma variável declarada como local é visível apenas
            dentro do bloco de código em que ela aparece. As seguintes
            variáveis locais estão disponíveis no método
            '#methodSignature': '#visibleLocalVariableList'. *>>
    source<**
            object #prototypeName extends #supertype implements
            #implementedInterfaces
                #methodSignature {
                      variáveis locais: #visibleLocalVariableList
                      #statementText
                      ...
                 }
            end
    **>
}

cause instanceVariable {
    explanation<<* Uma variável de instância é visível no protótipo a que
        pertence e em seus sub-protótipos. As seguintes variáveis de instância
        estão disponíveis no método '#methodSignature': '#instanceVariableList'. *>>
    source<**
            object #prototypeName extends #supertype implements
            #implementedInterfaces
                #methodSignature {
                      ...
                      #statementText
                      ...
                 }
                 variáveis de instância: #instanceVariableList
            end
    **>
}
```

compilador da LED Sepia também difere de um compilador tradicional pois não realiza a fase de geração de código em uma linguagem de baixo nível. O compilador gera objetos com as informações do código em Sepia e o EMS utiliza esses objetos para a composição das mensagens de erro.

Durante a compilação ocorre a procura de erros e inconsistências no código-fonte relacionado à LED Sepia. O tratador de erros retorna mensagens indicando a linha e o erro ocorrido e, enquanto estes persistirem, a fase de incorporação ao EMS não é processada.

Figura 21 – Causas do erro exibidas pelo compilador

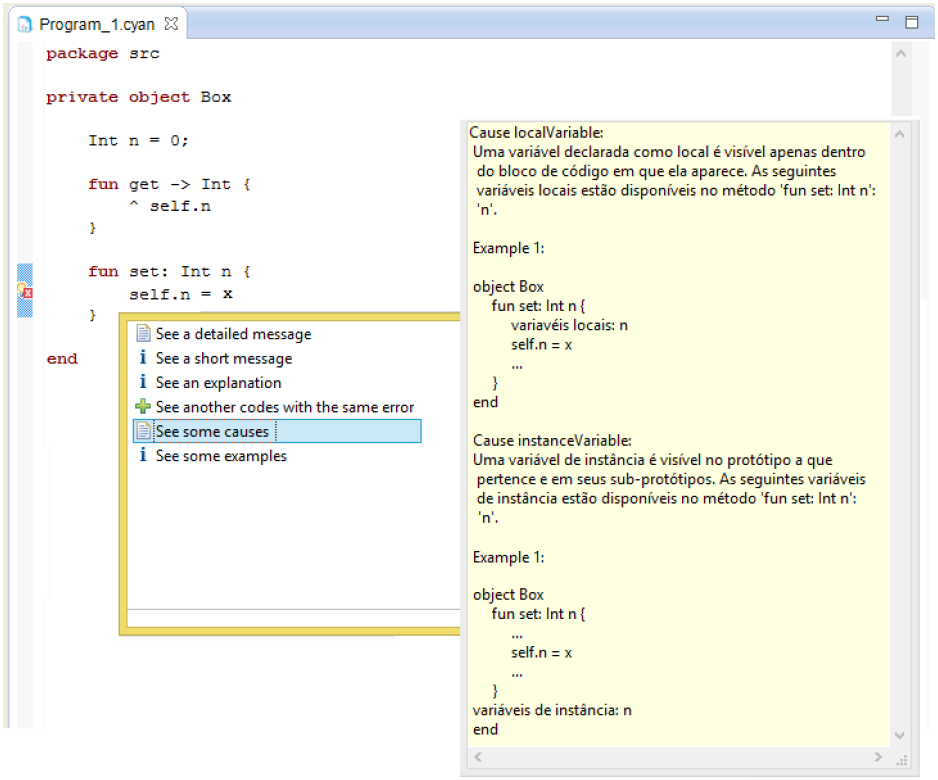
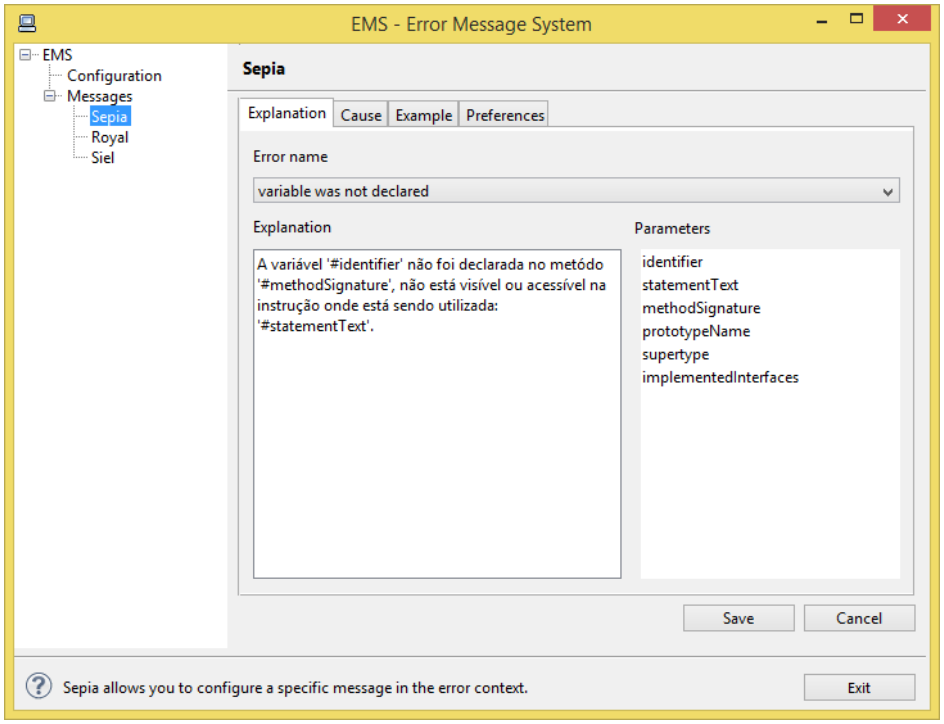


Figura 22 – Interface para criação de código em Sepia



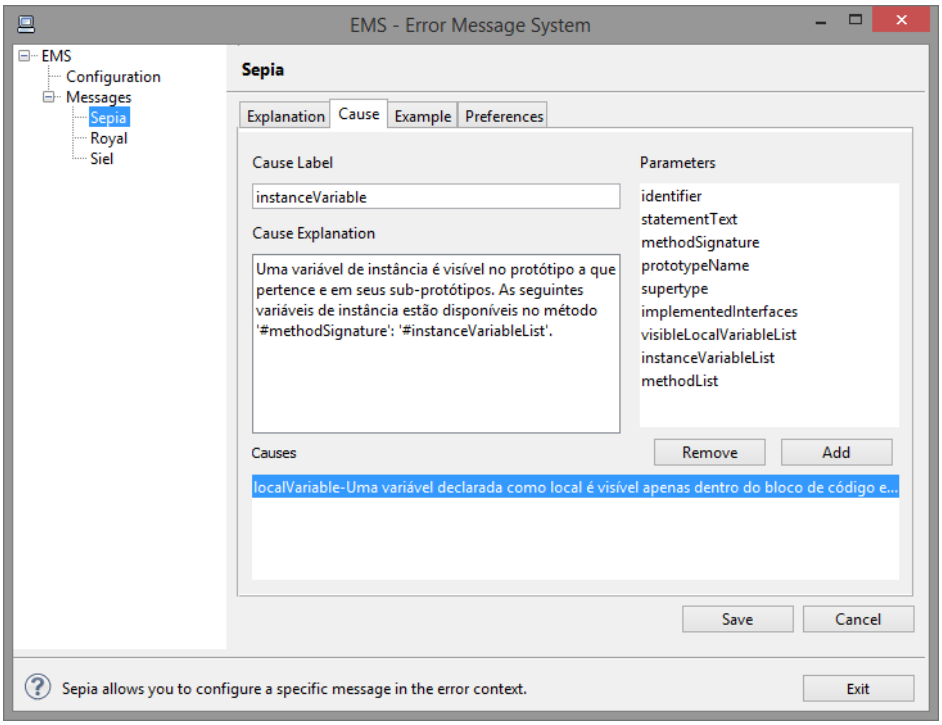
O código fonte da LED Sepia deve ser escrito em um arquivo com a extensão ".sepia" e colocado na pasta referente ao erro, previamente configurada no EMS.

Assim como na LED Royal, a chamada ao compilador da LED Sepia pode ocorrer de duas maneiras: primeiro quando o EMS acessa o código da LED para composição das mensagens de erro, segundo quando a ferramenta gráfica para a linguagem acessa o arquivo contendo o código-fonte da LED.

O EMS também conta com uma interface gráfica para a criação de código em Sepia. A interface permite ao programador escrever suas próprias mensagens de erro e incorporá-las ao compilador sem que seja necessário conhecimento da sintaxe da linguagem. A Figura 22 apresenta a interface gráfica da LED Sepia.

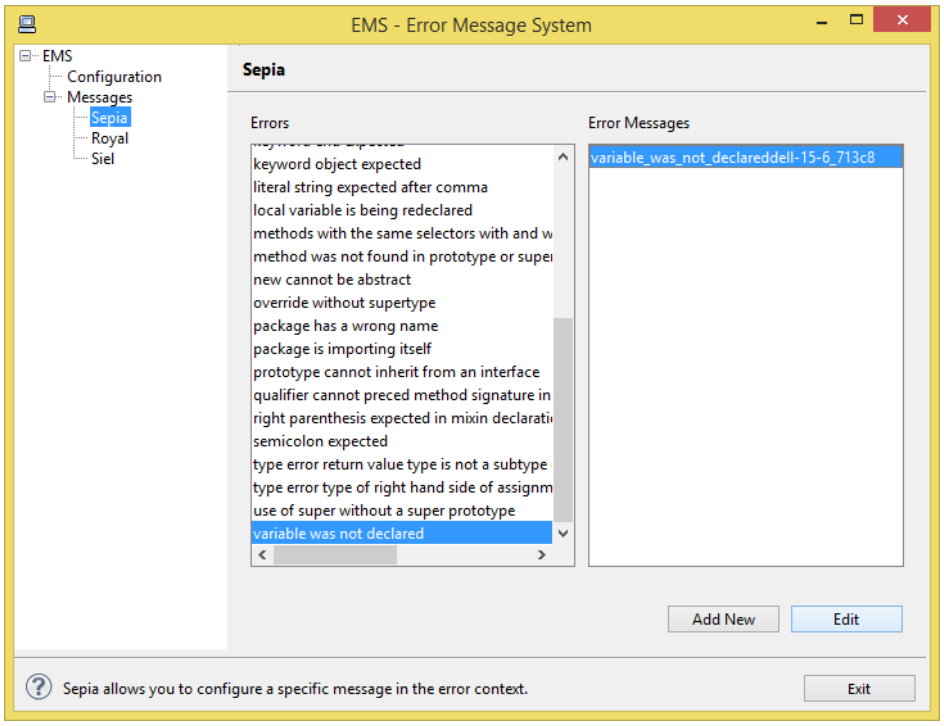
A interface gráfica da LED Sepia é estruturada em abas. Cada aba representa uma palavra reservada da LED. Na aba "Explanation", em "Error name" o programador seleciona o erro a que a mensagem se refere. No campo "explanation" é esperada a explicação do erro, que pode ser composta utilizando os parâmetros exibidos em "parameters". Um duplo clique em qualquer parâmetro da lista, adiciona o respectivo parâmetro precedido do caractere "#". A lista de parâmetros é extraída dos arquivos com extensão ".siel", explicado anteriormente na subseção 3.2.1. Na aba "cause" o usuário informará o identificador da causa do erro e sua explicação e, em "example" os exemplos referentes a cada causa. Veja a Figura 23 . Na aba "Preferences" é possível definir o nome do arquivo a ser salvo, se essa mensagem é a padrão ou não `(use)`, o idioma da mensagem e o idioma da mensagem em inglês `(language)`.

Figura 23 – Interface para inclusão das causas do erro



Também é possível gerenciar os códigos em Sepia associados a cada erro, o que

Figura 24 – Interface para gerenciamento de códigos em Sepia



pode ser visto na Figura 24. Assim como em Royal, a janela da esquerda apresenta a relação de erros existentes. A janela da direita apresenta as mensagens existentes para o erro. Ao selecionar uma mensagem e clicar em "Edit" o compilador da LED Sepia é chamado e verifica o código da mensagem. Se não ocorrer nenhum erro, o código Sepia é incorporado à interface gráfica e exibido conforme a Figura 22.

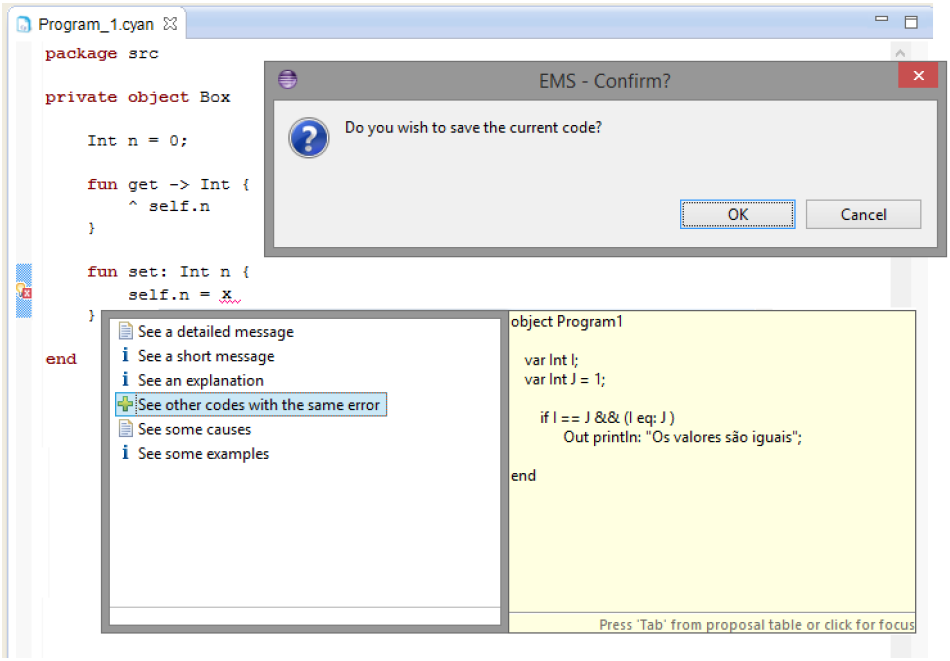
3.2.4 Código anterior do usuário

O *plug-in* EMS oferece ao usuário a opção de armazenar o código-fonte com o erro para posterior consulta. Essa opção permite que, através de consulta aos códigos com erro, o usuário se recorde das ações que tomou para corrigi-los anteriormente, reduzindo assim o esforço mental para tal correção.

Quando o compilador faz uma chamada para o método `signalCompilerError`, este envia, além do código em Siel, o código fonte do usuário que apresentou o erro. O EMS armazena estas informações.

A Figura 25 apresenta as janelas exibidas ao usuário. Quando ocorre um erro, o EMS procura arquivos com a extensão ".pcd" e os incorpora ao compilador, apresentando ao usuário o código com erro que ele escolheu salvar anteriormente. Um duplo clique em *"See other codes with the same error"* exibe a opção para o usuário gravar o código atual com erro. Se o usuário confirmar a gravação, um arquivo com a extensão ".pcd" é gerado e salvo em uma pasta configurada previamente no EMS.

Figura 25 – Código anterior do usuário



Não há nenhuma restrição quanto ao conteúdo do arquivo gerado, assim é possível incluir ou alterar o código armazenado como também adicionar observações a respeito dele manualmente editando o arquivo que foi salvo pelo EMS.

4 Casos de Testes com o Usuário

Este capítulo descreve os testes realizados com estudantes de programação para a avaliação do *plug-in* EMS. As amostras foram coletadas na ETEC Fernando Prestes, em Sorocaba, com estudantes dos cursos Técnico em Informática e Técnico em Informática para Internet. O teste foi dividido em duas fases: a primeira fase realizada em 17/03/2015 consistiu em identificar as dificuldades dos programadores em interpretar as mensagens de erro. A segunda fase teve como objetivo validar a proposta desta dissertação e foi realizada em 12/06/2015. É importante ressaltar que os testes não apresentam validade estatística devido a pequena quantidade de participantes da avaliação. Os detalhes das fases, seus resultados e conclusões são apresentados nas seções a seguir:

4.1 Fase 1 - Viabilidade da Proposta

O objetivo desta fase foi verificar se os programadores iniciantes apresentam dificuldades em interpretar as mensagens de erro atualmente, visto que as pesquisas sobre o tema remotam de décadas (SCHORSCH, 1995), (MOULTON; MULLER, 1967), (BROWN, 1983). A análise da interpretação das mensagens de erro depende essencialmente da participação dos usuários dos compiladores, contribuindo com suas opiniões e manifestações. Nesta fase três questões foram levantadas: (1) o programador consegue compreender o texto da mensagem de erro exibida? (2) através da mensagem apresentada, é possível determinar a causa do erro? (3) a mensagem apresentada fornece os requisitos necessários para interpretação e correção do erro ocorrido? A partir destes questionamentos cinco casos de teste foram desenvolvidos. Cada caso de teste corresponde a um programa que apresenta um erro específico. Os casos de teste foram desenvolvidos na linguagem Java, compilados na IDE Eclipse e nenhuma alteração foi realizada nas mensagens de erro emitidas pelo compilador Java. O nível de dificuldade dos programas foi aumentado gradativamente. A Tabela 5 apresenta os casos de testes aplicados aos programadores.

Cada caso de teste é identificado com um “Id". A Tabela 6 descreve as mensagens de erros e as ações necessárias para corrigir os erros.

No total 100 estudantes participaram da pesquisa. Destes estudantes, 74 são alunos do primeiro módulo, 14 do segundo e 12 do terceiro módulo. A fim de documentar as respostas, um questionário *on-line* foi disponibilizado para os estudantes.

O teste foi aplicado da seguinte maneira: o projeto com os casos de teste, ou seja, os cinco programas na linguagem Java, foram disponibilizados em uma pasta da rede da escola para os alunos. Os estudantes foram orientados a compilar os programas com erro,

Tabela 5 – Fase 1 - Casos de teste

Código em Java	**Id**

`void Program_1(){`
`  double num1;`
`  Scanner scan = new Scanner(System.in);`
 	
`  System.out.println("Informe o primeiro número");`
`  num1 = scan.nextDouble();`
 	
`  System.out.println("Informe o segundo número");`
`  num2 = scan.nextDouble();`
 	
`  scan.close();`
 	
`  System.out.printf("A soma de %f e %f é %f",`
`  num1, num2, (num1 + num2));`
`}`
	1

`String Program_2(){`
`  int num1;`
 	
`  Scanner scan = new Scanner(System.in);`
 	
`  System.out.println("Informe o primeiro número");`
`  num1 = scan.nextInt();`
 	
`  if(num1 % 2 == 0)`
`  {`
`  return ("O número informado é par.");`
`  }`
`  else`
`  {`
`  return 0;`
`  }`
`  return 0;`
`}`
	2

`void Program_3(){`
`  int num1;`
 	
`  Scanner scan = new Scanner(System.in);`
 	
`  System.out.println("Informe um número");`
`  num1 = scan.nextInt();`
 	
`  System.out.printf("O sucessor de %f é %f e o " +`
`  "antecessor é %f ", num1,num1 + 1), (num1 - 1)));`
 	
`  scan.close();`
`}`
	3

Código em Java	Id

`void Program_4() {`
`  int num1,`
`  Scanner scan = new Scanner(System.in);`
 	
`  try{`
`  System.out.println("Informe um número");`
`  num1 = scan.nextInt();`
 	
`  } catch (Exception ex) {`
`  System.out.println("Erro de conversão:"`
`  + ex.getMessage());`
`  ex.printStackTrace();`
`  }`
`}`
	4

`void Program_5(){`
`  int opt;`
`  Scanner scan = new Scanner(System.in);`
 	
`  try{`
`  System.out.println("Escolha uma opção" +`
`  "\n1. Previsão do  Tempo" + "\n2. Horário");`
 	
`  opt = scan.nextInt();`
 	
`  switch (opt) {`
`  case 1:`
`  previsao();`
`  break;`
`  {`
`  case 2:`
`  horarios();`
`  break;`
`  }`
`  }catch (Exception ex) {`
`  System.out.println("Erro:" ex.getMessage());`
`  ex.printStackTrace();`
`  }`
`}`
`void previsao(){`
`  System.out.println("Hoje o dia está "+`
`  "ensolarado!!");`
`}`
 	
`void horarios(){`
`  System.out.println(new java.util.Date());`
`}`
	5

Tabela 6 – Fase 1 - Mensagens de erro e correções

Id	**Mensagem de erro**	**Ação para corrigir o erro**
1	num2 cannot be resolved to a variable	Declarar a variável num2
2	Type mismatch: cannot convert from int to String	Alterar o tipo de retorno do método para String
3	Syntax error on token “)”, delete this token	Remover “)“
4	Syntax error, insert “;”	Substituir “,” por “;” na declaração da variável “num1”
5	Syntax error, insert “}” to complete Block	Remover “{” após a instrução “break” do “case 1:”

ler as mensagens exibidas pelo compilador e para cada programa compilado com erro, responder duas perguntas no formulário:

1. você é capaz de compreender a mensagem exibida pelo compilador?
2. qual a ação necessária para corrigir o problema?

Além das questões referentes aos casos de teste, os estudantes também responderam às seguintes perguntas após a realização do teste:

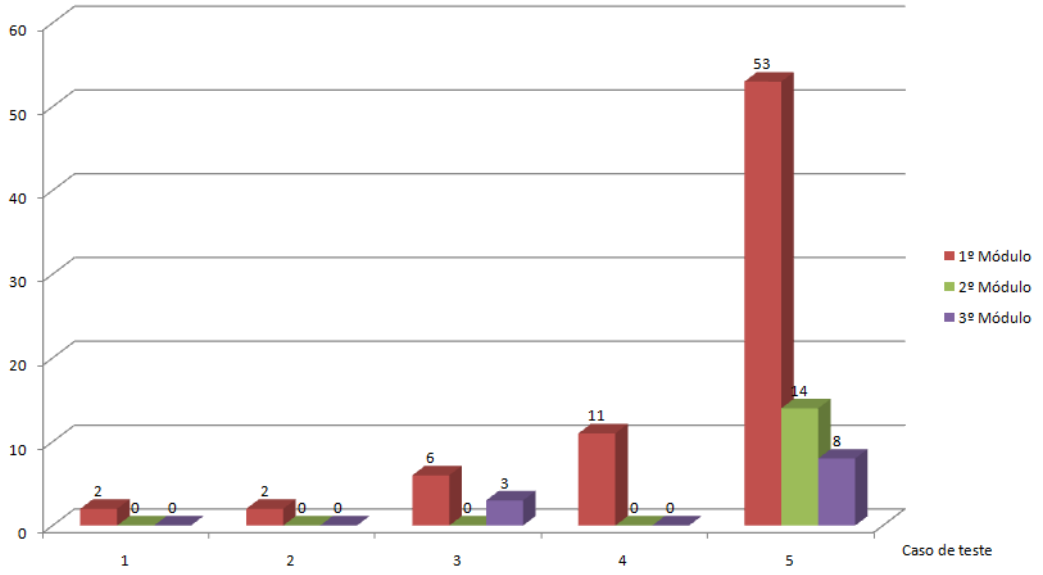
1. você possui conhecimento na linguagem Java?
2. você leu completamente todas as mensagens exibidas pelo compilador?
3. qual foi sua maior dificuldade em compreender os erros ocorridos?
4. na sua opinião, qual programa foi mais difícil de corrigir?

Durante a aplicação dos testes foi observado o tempo para responder o questionário e o comportamento dos estudantes. O tempo máximo para realização do experimento foi definido como 60 minutos. Nas turmas do primeiro módulo, o tempo para resposta variou de 50 a 60 minutos. Foi possível observar que os estudantes tentavam corrigir o erro arbitrariamente, mesmo tendo sido orientados a ler as mensagens exibidas. Este foi provavelmente o maior motivo para a demora na execução dos testes. Nas turmas do segundo e terceiro módulo o tempo para resposta variou entre 20 e 30 minutos. Apesar de já estarem familiarizados com as mensagens de erro, alguns ainda apresentavam dúvidas e queriam obter informações com outros ou na internet, alguns também tentaram corrigir o erro arbitrariamente.

Dos 100 estudantes que realizaram o teste, todos possuem conhecimentos básicos em programação. Deles 23 afirmaram possuir conhecimento na linguagem em Java, a maioria deles (12) estudantes do terceiro módulo.

Quando questionados sobre o programa que apresentou maior dificuldade de correção, 73% dos estudantes relataram que o programa mais difícil de compreender foi o quinto. A Figura 26 apresenta as respostas dos estudantes.

Figura 26 – Fase 1 - Programa mais difícil de corrigir



Quando questionados sobre o entendimento das mensagens de erro, as respostas variaram de acordo com o nível de dificuldade do programa (caso de teste). No caso do teste 5, é possível observar que poucos estudantes conseguiram compreender a mensagem de erro, o que era esperado, já que o caso de teste apresentava uma mensagem de erro que não correspondia ao erro ocorrido. A Figura 27 apresenta o gráfico das respostas dos estudantes sobre a compreensão das mensagens de erro.

É possível observar que a maioria dos estudantes do primeiro módulo entenderam parcialmente a maioria das mensagens de erro. Ao contrário dos estudantes do terceiro módulo, já que a maioria respondeu compreender completamente as mensagens de erro. A seguir foi avaliado se os estudantes conseguiriam corrigir o erro baseando-se na mensagem apresentada. No gráfico 28 é apresentado o resultado da análise.

Analisando os estudantes do primeiro módulo, é possível observar que apesar de 67% dos estudantes relatarem que entenderam completamente ou parcialmente a mensagem de erro, apenas 48% deles souberam como corrigir o erro do primeiro caso de teste. A situação é ainda pior no quinto caso de teste, em que 50% alunos afirmaram entender a mensagem de erro e apenas 7% souberam efetivamente como corrigir o erro. É possível também verificar que os estudantes do segundo módulo, apresentaram algumas dificuldades na interpretação do erro e que os do terceiro módulo, apresentaram pouca/nenhuma dificuldade para interpretar e corrigir os erros apresentados.

Figura 27 – Fase 1 - Compreensão das mensagens de erro

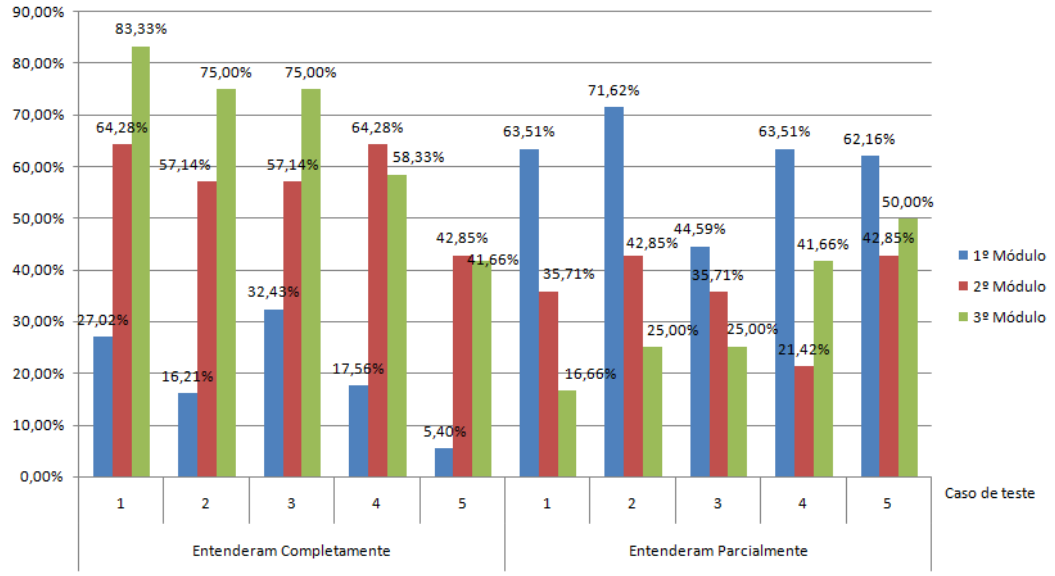
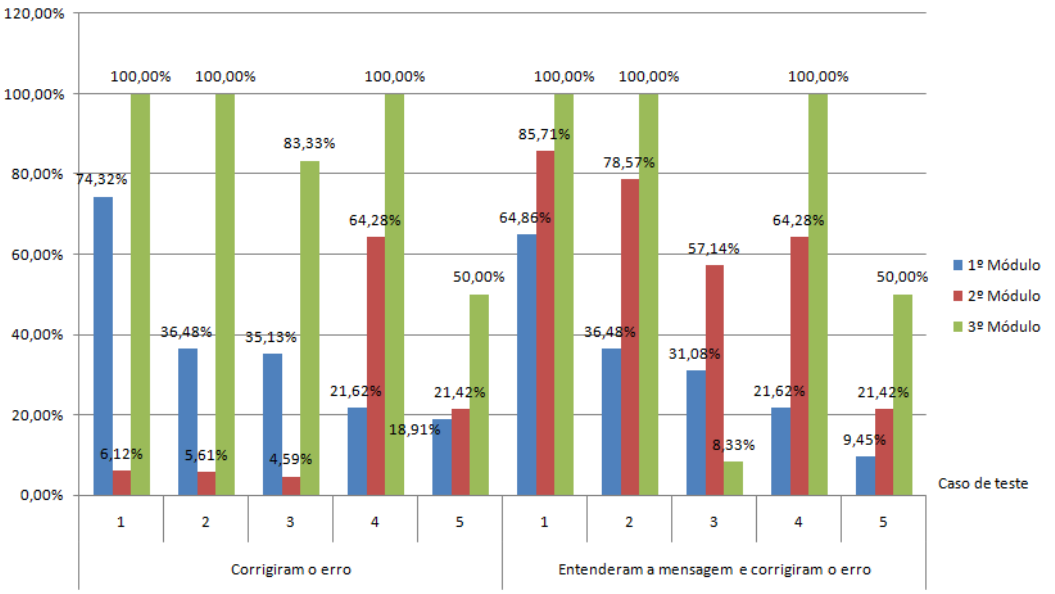


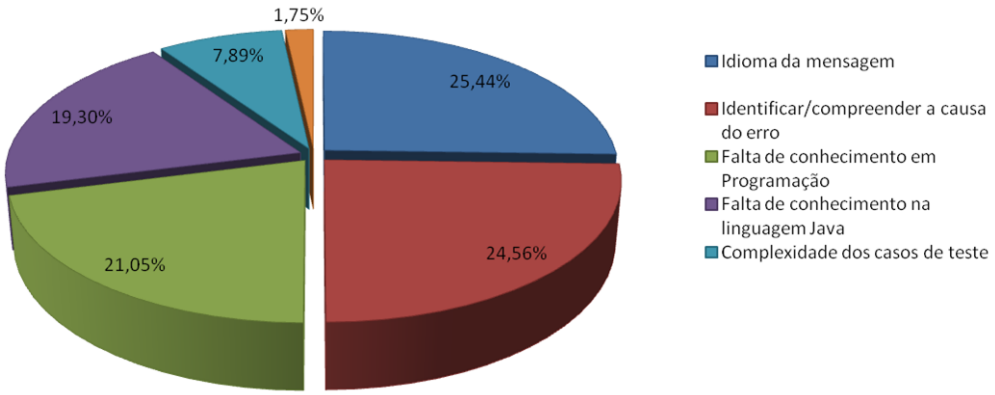
Figura 28 – Fase 1 - Estudantes que corrigiram os erros



Dentre as dificuldades encontradas pelos estudantes, os problemas mais citados são: problemas com o idioma e problemas em identificar e compreender o erro ocorrido. A Figura 29 apresenta as principais dificuldades apontadas pelos alunos na interpretação e correção dos erros.

Analisando o experimento é possível confirmar que os programadores iniciantes não compreendem completamente as mensagens de erro e que essas mensagens não são capazes de conduzir os programadores a corrigirem o erro. Seja pelo idioma ou pelo conhecimento prévio do programador.

Figura 29 – Fase 1 - Dificuldades encontradas na interpretação e correção de erros



4.2 Fase 2 - Validação da Proposta

A segunda fase do teste teve como objetivo validar a proposta desta dissertação. As seguintes questões foram levantadas durante a composição da amostra: (1) a exibição melhorada das mensagens de erro dos compiladores facilita a identificação e correção do erro? (2) os programadores conseguem corrigir rapidamente os erros, quando da exibição de mensagens de erro mais detalhadas? (3) a disponibilização de ajuda adicional (exemplos, causas do erro, códigos anteriores com o mesmo erro) facilita a interpretação e correção do erro)? (4) é possível um programador iniciante/ estudante de programação compreender os erros em um programa e corrigi-lo sem conhecer a linguagem de programação utilizada, baseando-se apenas nas mensagens de erro exibidas pelo compilador?

A partir destes questionamentos, cinco casos de teste foram desenvolvidos. Cada caso de teste corresponde a um programa que apresenta um erro específico. Os casos de testes foram desenvolvidos na linguagem Cyan e compilados na IDE Eclipse. As mensagens de erro do compilador foram modificadas através do *plug-in* EMS. Além de uma mensagem curta, foram apresentados aos estudantes: uma mensagem detalhada, uma mensagem utilizando trechos do código que causou o erro, um exemplo de como corrigir o erro e as causas do erro.

Apesar de a maior queixa dos estudantes na compreensão das mensagens de erro ser o idioma, nessa segunda fase as mensagens foram apresentadas em inglês. Essa decisão foi tomada para que fosse possível validar a eficácia das funcionalidades disponibilizadas pelo *plug-in* EMS na exibição das mensagens de erro e comprovar que as mensagens melhoradas e outras opções de ajuda facilitam a compreensão e correção do erro, independente do idioma utilizado. A Tabela 7 apresenta os casos de testes aplicados aos programadores.

Cada caso de teste é identificado com um “Id". A Tabela 8 descreve uma das

Tabela 7 – Fase 2 - Casos de teste

Código em Cyan	**Id**

`object Program_1`
 	
`public fun print {`
`var Int J = 1;`
 	
`if I == J && (I eq: J ) {`
`Out println: "Os valores são iguais";`
`}`
`}`
`end`
	1

`object Program_2`
 	
`private var Int x;`
`private var Int y;`
 	
`public fun soma: {`
`x = 0;`
`y = 3.2;`
 	
`Out println: (x + y);`
`}`
`end`
	2

`object Program_3`
 	
`private var String name = ""`
 	
`public fun getName -> String {`
`^ self.name`
`}`
	3

`object Program_4`
 	
`public fun draw {`
`var Int x = 12;`
`var Int y = 16;`
 	
`if  x > 0 && (y > 0) {`
`drawBorder: x, y;`
`}`
`}`
`end`
	4

Código em Java	Id

`object Program_5`
 	
`public fun soma: (Int num1, Int num2) {`
`var Int resul = num1 + num2;`
`Out println: resul;`
`}`
 	
`public soma: (Double num1, Double num2) {`
`Out println: (num1 + num2);`
`}`
 	
`public fun soma: (Int num1, Int num2) {`
`var Int somax = num1 + num2;`
`somax println;`
`}`
 	
`end`
	5

Tabela 8 – Fase 2 - Mensagens de erro e correções

Id	**Mensagem em Sepia**	**Ação para corrigir o erro**
1	The variable “I” was not declared.	Declaração/Atribuição de variáveis
2	The type of the left-hand side of the assignment: “y” is Int. The type of the right-hand side, “3.2”, is Double. In an assignment the type of the right-hand side should be equal or subtype of the type the left-hand side.	Alteração do tipo de dados da variável
3	You need to terminate the definition of the prototype “Program_3” using the keyword “end”.	Inserção da palavra “end”
4	The method “drawBorder” is not declared in “Program_4” or in its super-prototypes or it is inaccessible due to its protection level.	Declaração do método “drawBorder”
5	There are two or more methods “soma” with the same name and with the same numbers and types of parameters. You can have methods with the same name. But at least one of the parameters of the second method needs to have a different type from the correspondent parameter of the first method.	Alteração dos parâmetros do método

mensagens exibidas e as ações necessárias para corrigir os erros.

No total 99 estudantes participaram da pesquisa. Dos quais 85 estudantes são

estudantes do primeiro módulo e 14 do segundo módulo. A fim de documentar as respostas, um questionário *on-line* foi disponibilizado para os estudantes.

O teste foi aplicado da seguinte maneira: os casos de teste, ou seja, os cinco programas na linguagem Cyan, foram disponibilizados para os alunos. Nenhum dos alunos possui conhecimento da linguagem de programação. Os estudantes foram orientados a ler as mensagens exibidas pelo compilador e responder, assim como na primeira fase, para cada programa, duas perguntas no formulário:

1. você é capaz de compreender a mensagem exibida pelo compilador?
2. qual a ação necessária para corrigir o problema?
3. você consultou outras mensagens além da mensagem curta para conseguir corrigir o erro?

Além das questões referentes aos casos de teste, os estudantes também responderam as seguintes perguntas após a realização do teste:

1. você leu completamente todas as mensagens exibidas pelo compilador?
2. qual foi sua maior dificuldade em compreender os erros ocorridos?
3. na sua opinião, qual programa foi mais difícil de corrigir?

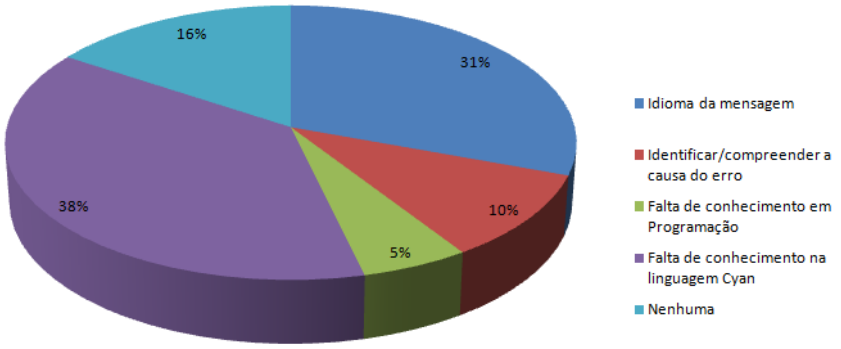
Assim como na fase 1, durante a aplicação dos testes foi observado o tempo médio de resposta do questionário e o comportamento dos estudantes. O tempo máximo para a realização do teste foi definido como 60 minutos. Em todas as turmas o tempo variou entre 10 e 30 minutos.

Foi possível observar que a maioria dos estudantes não conseguiu compreender os erros dos programas utilizando somente a mensagem curta exibida pelo compilador. A maioria conseguiu entender o erro ocorrido a partir da exibição da mensagem que continha trechos do código com erro. Para a correção do erro a exibição dos exemplos foi fundamental para os estudantes.

A principal queixa apresentada durante a realização do experimento foi a compreensão do idioma inglês. Muitos questionaram sobre a possibilidade de consulta a tradutores *on-line* para compreensão das mensagens.

Apesar de nenhum estudante possuir conhecimento na linguagem Cyan, 16,83% deles afirmaram não encontrar nenhuma dificuldade na interpretação das mensagens de erro e solução dos problemas apresentados, o que não ocorreu na fase 1, em que todos os

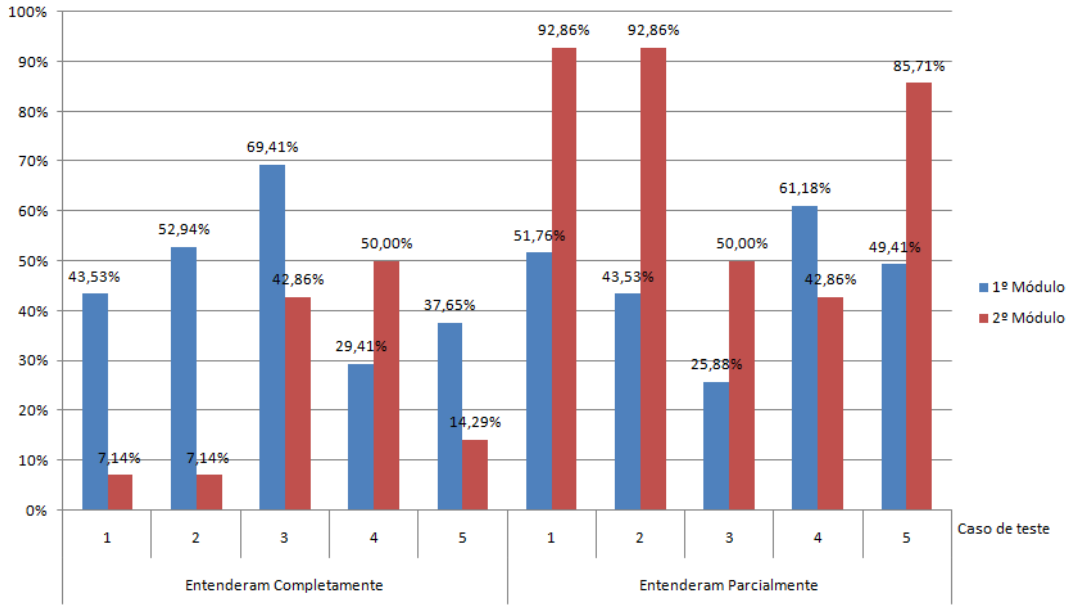
Figura 30 – Fase 2 - Dificuldades encontradas na interpretação e correção de erros



participantes relataram alguma dificuldade. A Figura 30 apresenta as maiores dificuldades que os estudantes afirmar ter durante a realização do experimento.

Pode-se verificar que os maiores obstáculos citados pelos estudantes são a falta de conhecimento na linguagem de programação e o idioma da mensagem. É importante ressaltar que apesar de apresentarem dificuldade devido a falta de conhecimento na linguagem Cyan, aproximadamente 90% dos estudantes afirmaram compreender, ainda que parcialmente, as mensagens de erro, conforme a Figura 31.

Figura 31 – Fase 2 - Compreensão das mensagens de erro pelos estudantes

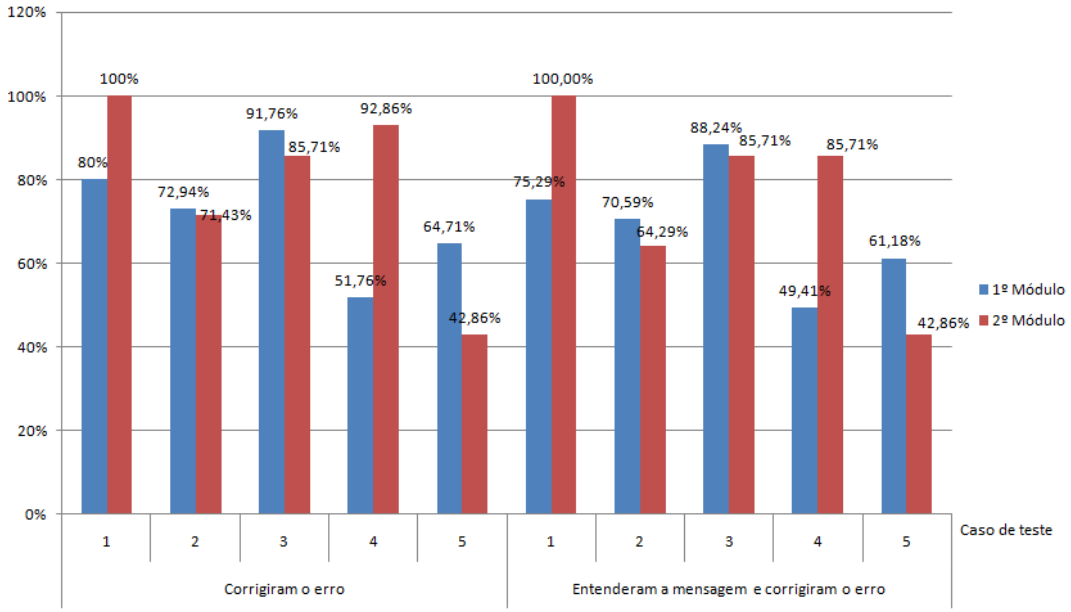


Quando da correção dos erros, 63% dos participantes que afirmaram compreender as mensagens de erro souberam como corrigi-los. O programa quatro foi o que apresentou

o menor número de respostas corretas no primeiro módulo e o programa cinco no segundo módulo, o que pode ser observado na Figura 32.

É interessante observar que apesar de o programa cinco ser o mais complexo, o número de estudantes do primeiro módulo que souberam como corrigir o erro foi maior que o número de estudantes do segundo módulo, que possuem maior conhecimento em programação e que afirmaram anteriormente que haviam compreendido a mensagem de erro do programa.

Figura 32 – Fase 2 - Correção dos programas



Quando questionados se leram completamente as mensagens de erros, 75,24% participantes afirmaram ter lido completamente as mensagens de erro, sendo 62,37% estudantes do primeiro módulo e 12,87% estudantes do segundo módulo.

4.3 Considerações finais

Os experimentos evidenciam que os estudantes apresentam diversas dificuldades na compreensão das mensagens de erro dos compiladores. As causas dessas dificuldades estão diretamente relacionadas ao idioma da mensagem exibida, a falta de conhecimento em programação e a falta de conhecimento da linguagem de programação utilizada pelo programador.

O tempo para correção dos programas apresentou uma redução significativa quando da apresentação das mensagens de erro utilizando o *plug-in* EMS.

O número de estudantes que afirmaram compreender as mensagens de erro foi relativamente maior quando da utilização do *plug-in*. Na fase 1 a média dos estudantes

que afirmaram compreender as mensagens de erro foi de 84,6%, entretanto a média dos que souberam corrigir efetivamente o erro foi de apenas 43,2%. Em contra-partida, apesar de nenhum dos estudantes possuir conhecimento da linguagem Cyan, em média 92% dos participantes compreenderam as mensagens de erro exibidas pelo compilador e a média dos que souberam como corrigir os erros foi de 69%. O resultado do teste foi extremamente satisfatório, levando em consideração que a maioria dos participantes da fase 2 são estudantes do primeiro módulo, que possuem pouco conhecimento em programação.

Além dos testes com os usuários, foram realizados testes com dois especialistas, ambos professores de linguagens de programação da ETEC Fernando Prestes. Os especialistas avaliaram a usabilidade da ferramenta na composição das mensagens e acompanharam os testes com os estudantes.

Foram propostas melhorias na interface gráfica do *plug-in* EMS e correções de pequenos erros. Durante observação na realização dos testes, os especialistas relataram que muitos dos estudantes não liam completamente as mensagens de erro, mas que devido à disponibilização de ajuda adicional (exemplos e as causas dos erros) conseguiram identificar e corrigir os erros apresentados.

Os especialistas avaliaram positivamente o *plug-in* como auxiliar na correção e prevenção dos erros e como ferramenta para favorecer o aprendizado de linguagens de programação.

Foram escritas 30 mensagens de erro em Sepia e 30 em Royal utilizando as interfaces gráficas fornecidas pelo EMS para as LEDs. A utilização da ferramenta na criação das mensagens de erro não apresentou problema. Entretanto, pode-se constatar que para criar uma mensagem de erro é preciso que o programador possua um conhecimento avançado sobre a linguagem de programação, para que ele possa fornecer uma explicação consistente sobre o erro, suas possíveis causas e exemplos de como corrigi-lo.

As mensagens para explicar os erros léxicos e semânticos são fáceis de ser escritas, pois o compilador fornece as informações precisas para compor a mensagem de erro. Porém escrever as mensagens para os erros sintáticos é uma tarefa difícil. O compilador não consegue identificar precisamente o que causou o erro, assim, o programador que está escrevendo a mensagem precisa especificar além da causa obvia, outras possíveis causas para o erro.

5 Conclusão

A exibição de uma mensagem de erro por um compilador exige o estudo de diversas áreas da computação para que a mensagem seja facilmente compreendida pelos diferentes tipos de usuários.

Cada usuário possui um perfil único, e a interpretação da mensagem exibida varia de acordo com o nível de conhecimento da linguagem de programação e do conhecimento prévio do usuário. Assim, escrever uma mensagem para um compilador não consiste apenas da criação do texto que será exibido. Envolve fatores como:

a) conhecimento avançado da linguagem de programação, para que seja possível explicar e indicar as causas do erro;

b) conhecimento do ambiente de desenvolvimento em que o usuário irá escrever seu código;

c) estudo de boas práticas para composição de mensagens de erro;

d) compreensão do comportamento dos usuários ao se depararem com um erro;

e) conhecimento sobre as dificuldades inerentes da programação;

f) noções didáticas para indicar as ações de correção e prevenção do erro no futuro.

Estes fatores contribuem diretamente para que as pesquisas sobre as mensagens de erro dos compiladores seja negligenciada. Poucas pesquisas têm sido realizadas a fim de diminuir a complexidade e a grande distância entre a informação emitida pelo compilador e a mensagem que é compreendida pelo usuário.

O trabalho de Traver (TRAVER, 2010) é um dos mais recentes na área e, apesar de não ser diretamente relacionado a construção de compiladores, apresenta as necessidades existentes e os padrões para exibição das mensagens erro, de modo que o desenvolvimento de programas e o aprendizado de programadores iniciantes e estudantes da computação seja favorecido.

A realização do presente trabalho investigou as ferramentas previamente desenvolvidas para melhorar a compreensão das mensagens de erro dos compiladores. Também foi realizado o levamento de pesquisas existentes sobre o aprendizado dos programadores, sistema cognitivo, as mensagens de erro dos compiladores e linguagens de programação. A partir das informações coletadas foi proposta uma ferramenta em forma de *plug-in* para a

plataforma de desenvolvimento Eclipse. O EMS, fruto deste trabalho, visa sanar a necessidade de ferramentas para melhorar a exibição das mensagens de erro dos compiladores. O *plug-in* EMS está disponível para download no site "http://www.emsplugin.xyz".

Vários desafios foram encontrados no desenvolvimento do *plug-in* EMS. O primeiro foi a necessidade da criação de outro *plug-in* que permitisse acoplar a linguagem de programação Cyan e seu compilador ao ambiente de desenvolvimento Eclipse. Apesar do EMS não ser limitado à IDE Eclipse ou à linguagem Cyan, essa linguagem foi escolhida por ser objeto de estudo do grupo de pesquisa envolvido neste projeto.

Cyan e muitas linguagens modernas utilizam palavras-chave para começar um comando/método/protótipo/etc. Com isto, o compilador pode sinalizar mensagens mais precisas, provavelmente mais relacionadas com o erro que o código possui. Por exemplo, a declaração de variáveis começa com a palavra-chave `"var"`. Se houver erro logo em seguida a esta palavra, o compilador saberá que houve um erro na declaração de uma variável. Então o EMS mostrará uma mensagem realmente relacionada ao erro. Não é o que acontece com linguagens baseadas em C. Nelas um erro na declaração de uma variável pode facilmente ser sinalizada como um outro tipo de erro pois a declaração de uma variável não se inicia com uma palavra-chave e sintaticamente não é muito diferente de outras instruções.

Escrever códigos em Sepia e Royal para um erro léxico em Cyan é tão fácil como o é para uma outra linguagem, já que erros léxicos são praticamente iguais em todas as linguagens.

Há um problema na emissão de mensagens de erro pelo compilador Cyan: é difícil selecionar os parâmetros a serem passados no código Siel em `"signalCompilerError"`. Podem existir várias causas para o mesmo erro.

Os erros são iguais para um compilador convencional. Contudo, o compilador Cyan teria que fazer chamadas diferentes em cada caso para o método `signalCompilerError()`, enviando em cada chamada os parâmetros correspondentes à causa daquele erro.

O compilador Cyan tem que identificar o contexto em que o método `expr()`, que analisa uma expressão, foi chamado. De acordo com este contexto é que são passadas mais ou menos informações a `"signalCompilerError"`. E descobrir este contexto não é absolutamente trivial.

Este fato torna extremamente trabalhoso o acoplamento do EMS ao compilador Cyan. Para cada mensagem de erro deve-se selecionar os parâmetros adequados (aqueles descritos em `ErrorKind`) e muitos erros devem ser sinalizados de mais de uma forma de acordo com o contexto. Potencialmente um compilador que normalmente sinaliza 300 erros poderia sinalizar 600 e cada um com um código Siel diferente.

Contudo, possivelmente seria mais fácil acoplar o EMS em outras ferramentas que

não sejam compiladores pois estas possuem muito menos mensagens de erro. E talvez o código Siel para cada mensagem de erro seja menor.

Um software para extrair da classe `ErrorKind` os parâmetros necessários para os erros também foi criado. Essa ferramenta visou diminuir o trabalho do programador, evitando que o mesmo necessite incluir manualmente os parâmetros a serem utilizados.

Muitos erros são específicos e possuem várias causas. Assim, para compor as mensagens que os expliquem é necessário que o usuário possua um amplo conhecimento na linguagem de programação, outros erros podem ser facilmente explicados por qualquer usuário.

Os outros desafios se referem à ausência de ferramentas e bibliografias recentes na área. Grande parte dos estudos encontrados referem-se a Interação Humano Computador, apresentando padrões para exibição de mensagens genéricas e estudos sobre o aprendizado da computação. Poucas ferramentas foram desenvolvidas para solucionar o problema das mensagens de erro. A maioria delas apresentam uma solução alternativa, mas não exibe as explicações e causas do erro diretamente no ambiente de compilação ou permitem que o usuário escreva suas próprias mensagens de erro.

Estudos da Interação Humano Computador, permitiram a definição de padrões para as mensagens e a compreensão dos tópicos necessários para a composição de um *plug-in* que solucionasse os problemas encontrados com a exibição das mensagens de erro pelos compiladores.

A utilização das linguagens específicas de domínio resultou na criação de seus respectivos compiladores. A escolha da utilização das LEDs foi para atender os conceitos da IHC: comunicabilidade, acessibilidade, usabilidade e experiência do usuário e permitir que o *plug-in* seja altamente configurável por qualquer usuário.

O Eclipse é um ambiente de desenvolvimento de código aberto, que permite a alteração de suas funcionalidades através de modificações diretas no código-fonte ou através da implantação de *plug-ins*. Estudos sobre Engenharia de Software foram necessários para a compreensão do funcionamento da IDE Eclipse, para o desenvolvimento dos *plug-ins* Cyan e EMS e de uma interface gráfica para a criação e gerenciamento das mensagens de erro.

O *plug-in* EMS permite que as mensagens de erro dos compiladores sejam alteradas de acordo com o nível de experiência do usuário/programador. Isto é possível, pois a ferramenta fornece uma interface que, permite ao usuário escrever suas próprias mensagens e, através de LEDs, possa compartilhar seus códigos com outros usuários ou adquirir pacotes de mensagens.

A exibição de exemplos, causas do erro e de mensagens adaptadas ao contexto do código que está sendo desenvolvido oferece alguns benefícios. Facilita não somente

a localização e compreensão do erro, como estimula o aprendizado, diminui o tempo despendido na correção do erro e favorece o desenvolvimento de programas com qualidade. Assim, o programador não necessita de soluções alternativas (e muitas vezes não adequadas) para corrigir o erro.

A exibição de códigos anteriores do usuário que apresentaram o mesmo erro tem como objetivo diminuir o esforço do programador ao ter que recordar as ações tomadas quando o mesmo erro ocorreu no passado.

Testes com usuários foram realizados para avaliação da proposta e do *plug-in*. Os testes foram divididos em duas fases. A primeira fase foi direcionada para verificação da proposta, já que os estudos sobre o tema foram realizados há muitos anos atrás. O principal questionamento foi: os programadores iniciantes, nos dias atuais, apresentam dificuldades em interpretar as mensagens exibidas pelos compiladores? Os testes realizados permitiram a identificação das dificuldades encontradas pelos programadores. A segunda fase foi direcionada a comprovar a eficácia da ferramenta em um grupo limitado de usuários.

Nas duas fases do testes, pode-se verificar que muitos dos programadores simplesmente não liam as mensagens de erro. Tentavam localizar diretamente no código o problema, como se a sinalização do erro fosse um desafio e não uma informação.

Vale ressaltar que o nível de significância dos resultados obtidos pelos testes é extremamente baixo. A amostra utilizada corresponde à uma mínima parcela do número de programadores existentes, estando limitada a um determinado grupo, sendo portanto necessário a realização de um experimento com um maior número de programadores.

Também foi possível verificar que, por mais que o compilador ofereça um sistema de ajuda eficaz, este só será útil se o programador utilizar corretamente suas funcionalidades.

Apesar das dificuldades encontradas a exibição de mensagens de erro de forma melhorada facilitou a compreensão, o aprendizado e a correção dos erros ocorridos, além de uma diminuição drástica do tempo necessário para a solução dos erros.

Este trabalho abre oportunidades para trabalhos futuros. Algumas possibilidades são descritas a seguir:

- o desenvolvimento de uma interface adaptativa, partindo da premissa de que existem vários níveis de usuários e com conhecimentos distintos, as mensagens se adaptariam ao usuário específico e caso o programador tentasse várias vezes insistir no erro sem alterações, as mensagens mudariam a fim de criar um ambiente melhor de interação entre o usuário e o compilador;

- poderão existir repositórios de códigos Royal, Sepia e Siel na internet. Em particular, no projeto EMS está prevista uma rede social chamada CyPeople incorporada ao

Eclipse no qual os programadores Cyan poderão interagir entre si. Através desta rede social os usuários poderão trocar códigos, explicações sobre os erros etc;

- a adaptação do projeto a outros ambientes de desenvolvimento;
- sugestões para correção do erro, utilizando exemplos de outros usuários da rede;
- a criação de elementos gráficos para sinalização do erro. Seriam utilizados círculos, ligações, setas etc. Estes elementos não somente identificariam onde ocorreu o erro, mas fariam a ligação com os trechos de código envolvido.

Referências

AHO, A. V. et al. *Compilers: Principles, Techniques and Tools (Second Edition)*. [S.l.]: Addison Wesley, 2007. 5, 45-49 p. ISBN 9788131759028. Citado 2 vezes nas páginas 32 e 34.

ALEXANDRESCU, A. Better templlate error messages. *C/C++ Users J.*, CMP Media, Inc., USA, v. 17, n. 3, p. 37–47, mar 1999. ISSN 1075-2838. Disponível em: <http://dl.acm.org/citation.cfm?id=313458.313470>. Citado 2 vezes nas páginas 24 e 39.

BARBOSA, S. D.; SILVA, B. S. da. *Interação humano-computador*. [S.l.]: Campus - Elsevier Editora Ltda., 2010. 27-42, 315-330 p. ISBN 9788535234183. Citado 4 vezes nas páginas 43, 44, 47 e 48.

BROWN, P. J. Error messages: the neglected area of the man/machine interface. *Commun. ACM*, ACM, New York, NY, USA, v. 26, n. 4, p. 246–249, apr 1983. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/2163.358083>. Citado 2 vezes nas páginas 24 e 81.

CARROLL, J. M. *Human Computer Interaction - brief intro*. Aarhus, Denmark: The Interaction Design Foundation, 2013. Disponível em: <http://www.interaction-design.org/encyclopedia/human_computer_interaction_hci.html>. Citado na página 46.

CHANDLER, P.; SWELLER, J. Cognitive load theory and the format of instruction. *Cognition and instruction*, Taylor & Francis, v. 8, n. 4, p. 293–332, 1991. Citado na página 42.

COULL, N. *SNOOPIE: Development of a Learning Support Tool for Novice Programmers Within a Conceptual Framework*. 11-14 p. Tese (PHD Thesis) — University of St Andrews, 2008. Disponível em: <http://hdl.handle.net/10023/522>. Citado na página 23.

CYBIS MARCELO SOARES PIMENTA, M. C. S. L. G. Walter de A. Uma abordagem ergonômica para o desenvolvimento de sistemas interativos. *Atas do I Workshop sobre Fatores Humanos em Sistemas Computacionais*, IHC 98, p. 102–111, 1998. Disponível em: <http://www.unicamp.br/~ihc99/Ihc99/AtasIHC99/AtasIHC98/Cybis.pdf)>. Citado na página 48.

DANN, W.; COOPER, S. Alice: an educational software that teaches students computer programming in a 3d environment. 1999. Disponível em: <http://www.alice.org>. Citado na página 52.

DEURSEN, A. van; KLINT, P.; VISSER, J. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 35, n. 6, p. 26–36, jun. 2000. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/352029.352035>. Citado na página 48.

DIX, A. *Human-Computer Interaction - Third Edition*. [S.l.]: Pearson/Prentice-Hall, 2004. 608-617 p. ISBN 9780130461094. Citado na página 45.

EBRAHIMI, A. Novice programmer errors: language constructs and plan composition. *Int. J. Hum.-Comput. Stud.*, Academic Press, Inc., Duluth, MN, USA, v. 41, n. 4, p. 457–480, oct 1994. ISSN 1071-5819. Citado 2 vezes nas páginas 38 e 40.

ECLIPSE. 2004. Disponível em: <http://www.eclipse.org>. Citado na página 49.

ECLIPSE. *PDE Project*. 2008. Disponível em: <http://www.eclipse.org/pde/>. Citado na página 50.

FARIA, E. S. J. de; JÚNIOR, E. G. Ajude-c: Software de apoio à tradução das mensagens de erros em programas na linguagem c++. *HÍFEN*, v. 30, n. 58, 2006. Citado na página 41.

FAYAD, M.; SCHMIDT, D. C. Object-oriented application frameworks. *Commun. ACM*, ACM, New York, NY, USA, v. 40, n. 10, p. 32–38, out. 1997. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/262793.262798>. Citado na página 107.

FLOWERS, T.; CARVER, C.; JACKSON, J. Empowering students and building confidence in novice programmers through gauntlet. In: *Frontiers in Education, 2004. FIE 2004. 34th Annual*. [S.l.: s.n.], 2004. p. T3H/10 – T3H/13 Vol. 1. ISSN 0190-5848. Citado na página 23.

FOWLER, M. Language workbenches: The killer-app for domain specific languages? 2005. Disponível em: <http://www.martinfowler.com/articles/languageWorkbench.html>. Citado 2 vezes nas páginas 48 e 49.

GIMPEL, J. F. *Gimpel Software*. 1987. Disponível em: <http://www.gimpel.com>. Citado na página 52.

GOMES, A.; MENDES, A. Learning to program - difficulties and solutions. *International Conference of Engineering Education – ICEE*, sn, 2007. Disponível em: <http://www.ineer.org/Events/ICEE2007/papers/411.pdf>. Citado 3 vezes nas páginas 23, 24 e 40.

GUIMARÃES, J. de O. The cyan language. 2015. Disponível em: <http://www.cyan-lang.org/>. Citado 4 vezes nas páginas 23, 58, 62 e 68.

HARTMANN, B. et al. What would other programmers do: suggesting solutions to error messages. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2010. (CHI '10), p. 1019–1028. ISBN 978-1-60558-929-9. Disponível em: <http://doi.acm.org/10.1145/1753326.1753478>. Citado na página 52.

KO, A. J.; MYERS, B. A. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing*, Academic Press, Inc., p. 41–84, 2005. Citado na página 38.

KOLLING, M. et al. The bluej system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, Swets and Zeitlinger Publishers, v. 13, n. 4, p. 249–268, December 2003. ISSN 0899-3408. Disponível em: <http://www.cs.kent.ac.uk/pubs/2003/2190>. Citado na página 52.

LAHTINEN, E.; ALA-MUTKA, K.; JÄRVINEN, H.-M. A study of the difficulties of novice programmers. *SIGCSE Bull.*, ACM, New York, NY, USA, v. 37, n. 3, p. 14–18, jun. 2005. ISSN 0097-8418. Disponível em: <http://doi.acm.org/10.1145/1151954.1067453>. Citado na página 23.

LOWE, D.; PRESSMAN, R. *ENGENHARIA WEB*. [S.l.]: LTC, 2009. 143 - 228 p. ISBN 9788521616962. Citado na página 43.

MAYER, R. E. The psychology of how novices learn computer programming. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 13, n. 1, p. 121–141, mar. 1981. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/356835.356841>. Citado na página 40.

MILLER, G. A. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*, v. 101, n. 2, p. 343–352, may 1955. ISSN 0164-0925. Citado 2 vezes nas páginas 25 e 42.

MOLICH, R.; NIELSEN, J. Improving a human-computer dialogue. *Commun. ACM*, ACM, New York, NY, USA, v. 33, n. 3, p. 338–348, mar. 1990. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/77481.77486>. Citado 2 vezes nas páginas 45 e 46.

MOULTON, P. G.; MULLER, M. E. Ditran a compiler emphasizing diagnostics. *Commun. ACM*, ACM, New York, NY, USA, v. 10, n. 1, p. 45–52, jan 1967. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/363018.363060>. Citado 3 vezes nas páginas 24, 51 e 81.

NIELSEN, J. *10 Usability Heuristics for User Interface Design*. 1995. Disponível em: <http://www.nngroup.com/articles/ten-usability-heuristics/>. Citado na página 46.

NIELSEN, J.; MACK, L. R. *Usability inspection methods*. New York, NY, USA: John Wiley and Sons, 1994. 25-42, 141-171 p. Citado 2 vezes nas páginas 43 e 48.

NIENALTOWSKI, M.-H.; PEDRONI, M.; MEYER, B. Compiler error messages: what can help novices? *SIGCSE Bull.*, ACM, New York, NY, USA, v. 40, n. 1, p. 168–172, mar 2008. ISSN 0097-8418. Disponível em: <http://doi.acm.org/10.1145/1352322.1352192>. Citado 2 vezes nas páginas 23 e 45.

NORMAN, D. *The Design of Everyday Things*. [S.l.]: The Mit Press, 1998. 81 - 104 p. Citado na página 43.

NORMAN, D. A. *Models of human Memory*. [S.l.]: Academic Press, INC, 1970. 21-30 p. Citado na página 42.

NORMAN, D. A. Cognitive engineering. In: NORMAN, D. A.; DRAPER, S. W. (Ed.). *User Centered System Design: New Perspectives on Human-Computer Interaction*. Hillsdale, NJ: Erlbaum, 1986. p. 31–61. Citado 2 vezes nas páginas 44 e 45.

OVERFLOW, S. 2008. Disponível em: <http://www.stackoverflow.com/>. Citado na página 53.

PRATES, S. D. J. B. R. O. Avaliação de interfaces de usuário–conceitos e métodos. *Anais do XXIII Congresso Nacional da Sociedade Brasileira de Computação*, sn, 2003. Disponível em: <http://homepages.dcc.ufmg.br/~rprates/ge_vis/cap6_vfinal.pdf>. Citado na página 47.

PREECE, J.; ROGERS, Y.; SHARP, H. *Design de Interação*. [S.l.: s.n.], 2005. ISBN 9788536304946. Citado na página 48.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. 5th. ed. [S.l.]: McGraw-Hill Higher Education, 2001. ISBN 0072496681. Citado na página 23.

PRICE, A. de A.; TOSCANI, S.; UFRGS., I. de Informática da. *Implementação de linguagens de programação: compiladores*. [S.l.]: Sagra-Luzzatto, 2000. (Série Livros Didáticos). ISBN 9788524106392. Citado na página 32.

REEVES, B.; NASS, C. *The media equation: how people treat computers, television, and new media like real people and places*. New York, NY, USA: Cambridge University Press, 1996. ISBN 1-57586-052-X. Citado na página 39.

ROCHA, H. da; BARANAUSKAS, M. *Design e avaliação de interfaces humano-computador*. [S.l.]: Unicamp, 2003. ISBN 9788588833043. Citado na página 44.

SAVIDIS, A. Rapidly implementing languages to compile as c++ without crafting a compiler. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 37, n. 15, p. 1577–1620, dec 2007. ISSN 0038-0644. Disponível em: <http://dx.doi.org/10.1002/spe.v37:15>. Citado na página 52.

SBC. Interação humano computador. 2015. Disponível em: <http://www.sbc.org.br/index.php?Itemid=66>. Citado na página 43.

SCHORSCH, T. Cap: an automated self-assessment tool to check pascal programs for syntax, logic and style errors. *SIGCSE Bull.*, ACM, New York, NY, USA, v. 27, n. 1, p. 168–172, mar 1995. ISSN 0097-8418. Disponível em: <http://doi.acm.org/10.1145/199691.199769>. Citado 5 vezes nas páginas 24, 41, 51, 70 e 81.

SHACKELFORD, R. L.; BADRE, A. N. Why can't smart students solve simple programming problems? *Int. J. Man-Mach. Stud.*, Academic Press Ltd., London, UK, UK, v. 38, n. 6, p. 985–997, jun 1993. ISSN 0020-7373. Disponível em: <http://dx.doi.org/10.1006/imms.1993.1045>. Citado 2 vezes nas páginas 23 e 40.

SHNEIDERMAN, B. *Designing the user interface: strategies for effective human-computer interaction*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN 0-201-16505-8. Citado 3 vezes nas páginas 24, 39 e 47.

SIGCHI, A. *Curricula for Human-Computer Interaction*. 1992. Disponível em: <http://www.sigchi.org/>. Citado na página 46.

SOUZA, C. de et al. Interação humano computador: Perspectivas cognitivas e semióticas. In: . [S.l.]: Rio de Janeiro: Edições EntreLugar, 1999. p. 420–470. Citado na página 47.

TRAVER, V. J. Sobre los mensajes de error de los compiladores. *Proceedings of the Actas del VII Congreso Internacional de Interaccion on Persona-Ordenador (Interaccion on 06)*, AIPO, Puertollano, Spain, p. 345–348, nov 2006. Disponível em: <http://www.aipo.es/articulos/4/36.pdf>. Citado na página 24.

TRAVER, V. J. On compiler error messages: what they say and what they mean. *Adv. in Hum.-Comp. Int.*, Hindawi Publishing Corp., New York, NY, United States, v. 2010, p. 3:1–3:26, jan 2010. ISSN 1687-5893. Disponível em: <http://dx.doi.org/10.1155/2010/602570>. Citado 12 vezes nas páginas 17, 24, 27, 28, 38, 39, 41, 45, 46, 59, 61 e 95.

VEE, M.-H. N. C.; MEYER, B.; MANNOCK, K. L. Empirical study of novice errors and error paths. 2005. Disponível em: <http://se.ethz.ch/~{}meyer/publications/teaching/novices.p>. Citado 2 vezes nas páginas 40 e 41.

APÊNDICE A – Detalhes da implementação dos *Plug-ins* Cyan e EMS

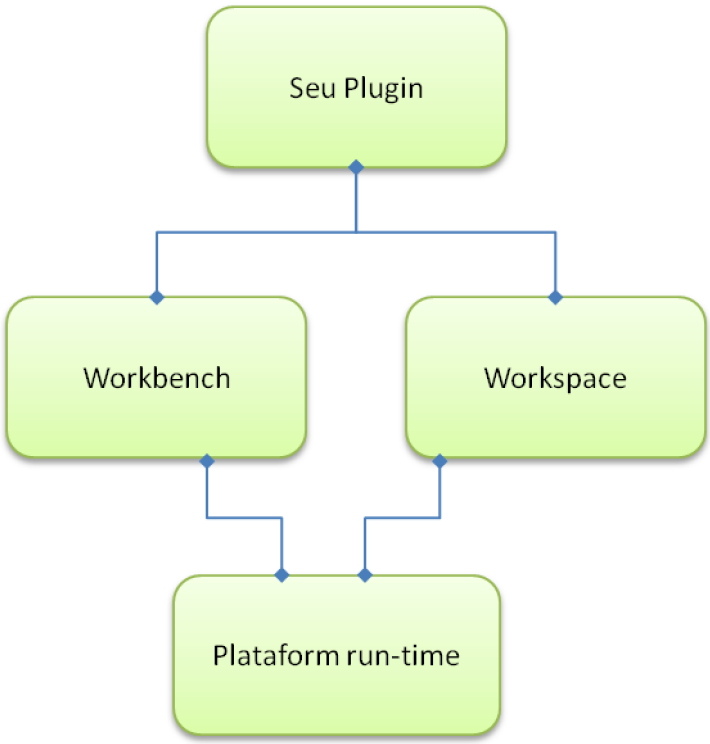
Este apêndice descreve alguns detalhes da implementação dos *plug-ins* Cyan e EMS para a IDE Eclipse. Informações técnicas de como desenvolver e implementar *plug-ins* não são abordados neste capítulo, já que o objetivo desta sessão é situar o usuário sobre a composição dos *plug-ins* desenvolvidos neste projeto.

A.1 Considerações Iniciais

O Eclipse é um *framework*[1] de código aberto para a construção de ambientes de desenvolvimento integrado, utilizando um conjunto de ferramentas pré-existentes e componentes de GUI[2].

A Figura 33 demostra a estrutura dos *plug-ins* com a plataforma Eclipse.

Figura 33 – Estrutura de *plug-ins* Eclipse

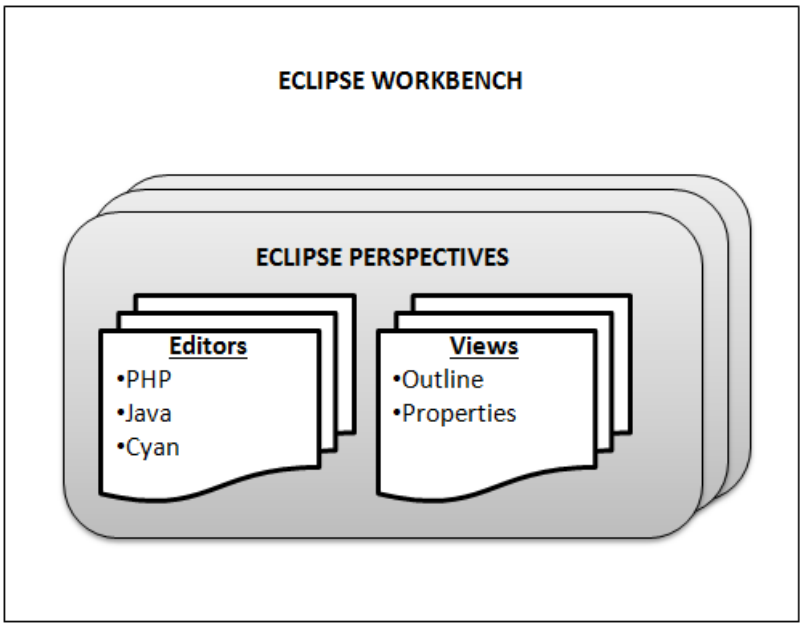


[1] Framework é um conjunto de classes que colaboram para realizar uma responsabilidade para um domínio de um subsistema da aplicação (FAYAD; SCHMIDT, 1997).

[2] do inglês *Graphical user interface, ou interface gráfica do usuário*

Para entendermos o funcionamento de um *plug-in* é necessário o conhecimento do *Workbench*. O termo *Workbench* refere-se ao ambiente de desenvolvimento desktop. O *Workbench* fornece um paradigma comum para a criação, gerenciamento e navegação de recursos do espaço de trabalho. A Figura 34 apresenta a perspectiva *Workbench* no Eclipse.

Figura 34 – Eclipse Workbench



O *Workbench* é composto de:

1. Editores - editores específicos estão associados com vários tipos de documentos ;

2. Views - visões para apoiar os editores ;

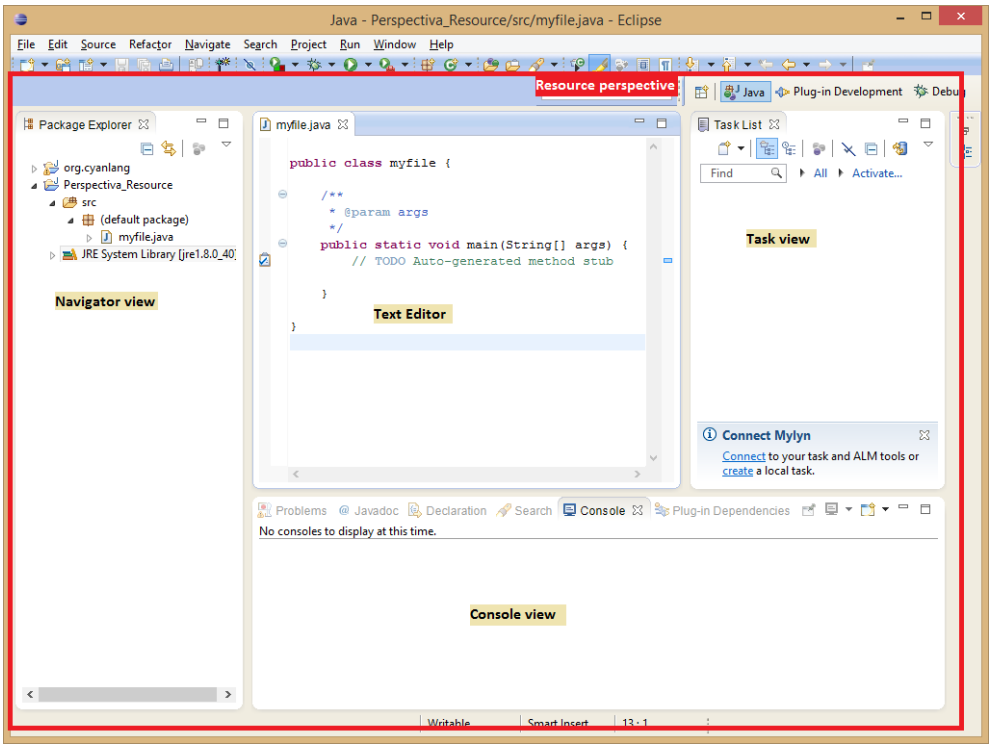
3. Perspectivas - uma combinação de pontos de vista e editores específicos.

Cada janela *Workbench* contém uma ou mais perspectivas, que podem conter *views*, editores e controles que aparecem em menus e barras de ferramentas. Mais que um *Workbench* pode existir no desktop ao mesmo tempo.

Um exemplo de janela do *Workbench* é apresentado na Figura 35. É possível observar que a perspectiva contém: o editor de texto, as visões *Navigator*, *Console* e *Task*.

Para a elaboração do *plug-in* Cyan, foi necessário a inclusão dos *plug-ins* PDE, JTD e JFace no *Workbench*. Estes *plug-in* fornecem as perspectivas necessárias para o desenvolvimento de *plug-ins* e são descritos no Capítulo 2. As sessões a seguir descrevem brevemente o desenvolvimento do *plug-in* Cyan e do *plug-in* EMS.

Figura 35 – Eclipse Resource



A.2 O *plug-in* Cyan

O *plug-in* Cyan acrescenta ao ambiente de desenvolvimento Eclipse, ferramentas que permitem a criação de projetos do tipo Cyan, assim como a criação de arquivos/protótipos Cyan. Os projetos Cyan possuem a extensão ".cyanproj" e os arquivos a extensão ".cyan". O *plug-in* também inclui na IDE, um editor de texto e as perspectivas necessárias para funcionamento do mesmo. A seguir são descritos os passos utilizados para a inclusão destas funcionalidades.

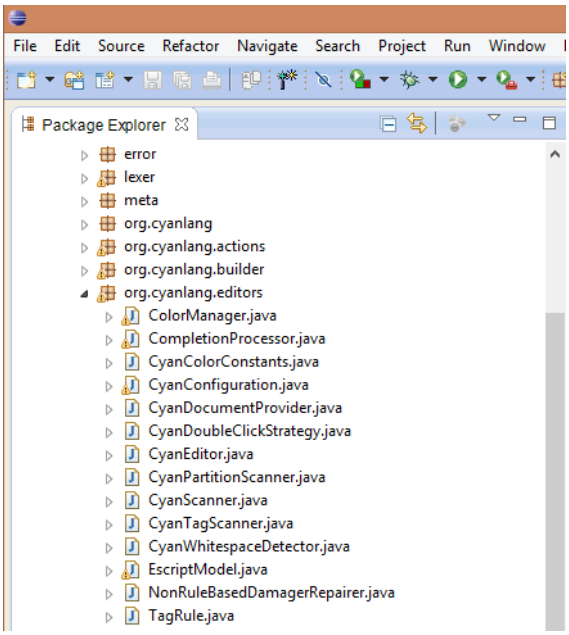
A.2.1 Criação do editor de texto Cyan

O *Workbench* define os pontos de extensão para que os *plug-ins* possam contribuir com funções de UI para a plataforma. Todas as classes que implementam os pontos de extensão da plataforma Eclipse herdam métodos e atributos de classes específicas. Dessa forma garante-se que a integração do *plug-in* à plataforma de forma correta e permite ao desenvolvedor preocupar-se apenas com a funcionalidade específica do *plug-in*. Alguns desses pontos de extensão *(Extension Points)* são implementados usando o JFace.

O *plug-in* JFace é a ferramenta que permite ao desenvolvedor Eclipse construir editores de texto com recursos avançados, como por exemplo: destaque de sintaxe, assistência conteúdo básico e formatação do código. O primeiro passo para o desenvolvimento do *plug-in* foi a criação de um novo projeto no Eclipse, do tipo *plug-in*. Esse projeto é composto

além de pacotes, arquivos de manifesto e configuração. O principal arquivo do projeto é o plugin.xml. Esse arquivo fornece as informações necessárias para o funcionamento do *plug-in*. Detalhes sobre esses arquivos são dados no Capítulo 2.

Figura 36 – Estrutura do projeto Cyan *Plug-in*



A estrutura do editor da linguagem Cyan está disponível na Figura 36. O pacote `org.cyanlang.editors` contém as classes que contribuem para o funcionamento do editor de texto Cyan. Cada classe possui uma funcionalidade, descrita a seguir:

A classe *CyanEditor* utiliza as funcionalidades da classe base (*TextEditor*), conforme visualizado na Figura 37 e delega a tarefa de adição de recursos adicionais.

Figura 37 – Visão da classe *CyanEditor*

```
package org.cyanlang.editors;

import org.eclipse.ui.editors.text.TextEditor;

public class CyanEditor extends TextEditor {

    private ColorManager colorManager;

    public CyanEditor() {
        super();
        colorManager = new ColorManager();
        setSourceViewerConfiguration(new CyanConfiguration(colorManager));
        setDocumentProvider(new CyanDocumentProvider());
    }
    public void dispose() {
        colorManager.dispose();
        super.dispose();
    }

}
```

Seu primeiro delegado é a instância da classe *SourceViewerConfiguration*, que é utilizada para adicionar recursos extras no editor de texto do usuário. Como por exemplo a classe *ColorManager* que fornece recursos de coloração ao editor, como destaques no texto, alterações na cor das palavras reservadas entre outros.

A segunda é uma instância da interface textitIDocumentProvider, que encapsula o mecanismo para criar a representação de um editor de texto *IDocument*.

O *IDocument* fornece métodos para mapear números de linha e posições de caracteres.

A interface *IDocument* foi desenvolvida para trabalhar sem nenhum conhecimento de como é armazenada, ou seja, a instância de um *IDocument* não tem nenhum conhecimento se foi carregado de um arquivo de sistema, de uma base de dados ou outra fonte. O trabalho de criar a instância do documento e inicialização do processo que define o estado inicial do documento é de responsabilidade da classe *IDocumentProvider*.

A seguir, temos as classes *CyanScanner* e *CyanTagScanner* que estendem da classe *RuleBasedScanner* do Eclipse e são responsáveis por definir as regras para o editor Cyan. Em *CyanScanner* são definidos os padrões para as palavras reservadas da linguagem Cyan e em *CyanTagScanner* os padrões para comentários.

A interface *CyanColorConstants* define as cores a serem utilizadas pelo editor. A definição das cores é feita através do padrão RGB[3]. A composição das cores é armazenada em constantes que posteriormente são utilizadas pelas classes. Por exemplo: a classe *CyanScanner* utiliza as constantes definidas pela classe *CyanColorConstants* para alterar a cor das palavras reservadas.

A classe *CompletionProcessor* fornece recursos ao editor de texto para completar o código do usuário com conteúdo sensível ao contexto. A Figura 38 exibe essa funcionalidade em execução. Baseando-se no que o usuário está digitando, o editor oferece sugestões para facilitar o trabalho de codificação.

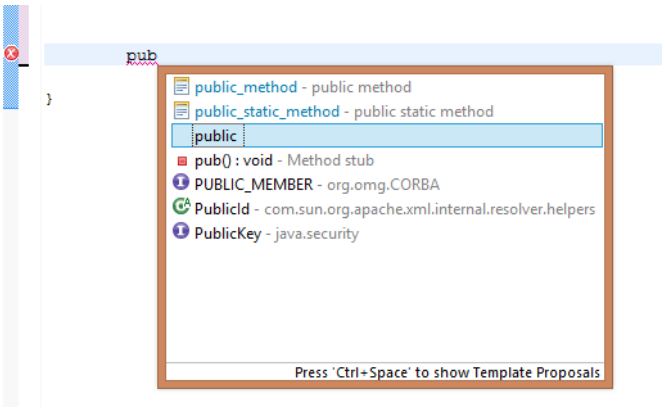
Quando o usuário pressionar as teclas “CTRL-space” ou “enter” uma janela com sugestões é exibida e com duplo clique, cujo as ações são definidas pela classe *CyanDoubleCliqueStrateg*, o conteúdo é adicionado ao código do usuário no editor de texto.

Um *plug-in* é gerado através das definições constantes do arquivo `plugin.xml` Este arquivo é obrigatório nos projetos de *plug-ins* e define as etapas e a interface com o usuário do *plug-in*.

Para funcionamento do editor da linguagem Cyan é necessário a inclusão de um *Extension Point* no arquivo `plugin.xml`. A Figura 39 demonstra o *Extension point* do editor da linguagem Cyan. Esse *Extension point* permite que quando um arquivo com

[3] RGB é a abreviação do sistema de cores aditivas formado por vermelho (red), verde (green) e azul (blue)

Figura 38 – Visão da classe *CompletionProcessor* em funcionamento.



a extensão ".cyan" for aberto na IDE Eclipse, a IDE reconheça a sintaxe da linguagem e forneça os recursos definidos pelo *plug-in*. *Extension points* fornecem os detalhes e o comportamento da funcionalidade implementada no *plug-in*.

Figura 39 – Extension Point do editor da linguagem Cyan

```
<extension
      point="org.eclipse.ui.editors">
   <editor
         class="org.cyanlang.editors.CyanEditor"
         contributorClass="org.eclipse.ui.texteditor.BasicTextEditorActionContributor"
         default="true"
         extensions="cyan"
         icon="icons/new_file.ico"
         id="org.cyanlang.editors.CyanEditor"
         name="Cyan Editor">
   </editor>
</extension>
```

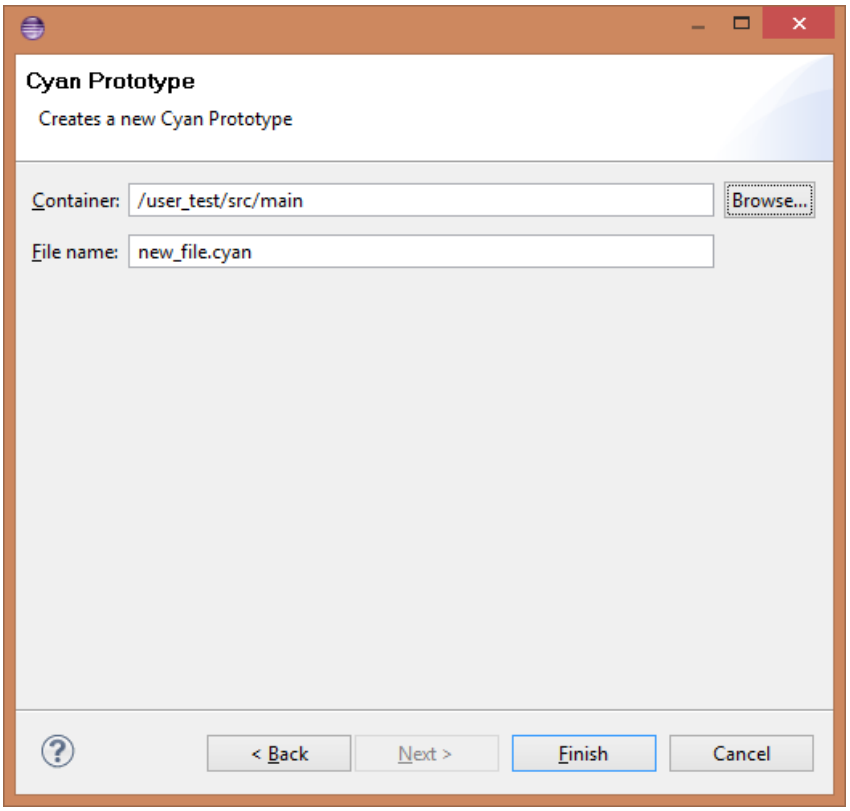
A.2.2 Assistentes

Assistentes ou *Wizards* são interfaces gráficas que através de um esquema passo-a-passo, auxiliam o usuário na execução de diversas tarefas, tais como a criação de novos projetos, pacotes, classes, interfaces etc.

No *plug-in* Cyan, temos Wizards para criação de projetos e protótipos/classes do tipo Cyan. Na Figura 40 é possível visualizar o assistente para criação de arquivos, incluído na IDE Eclipse.

Os arquivos responsáveis pela geração dos assistentes estão localizados no pacote `org.cyanlang.wizards`. A definição do assistente para criação de um novo arquivo é realizada através da classe *CyanFileWizard*. A classe contém as informações para a criação e leitura de arquivos com a extensão ".cyan".

Figura 40 – Assistente de criação de arquivo Cyan



A classe *CyanFileWizardPage* cria a janela do assistente em si. Um trecho desta classe é visível na Figura 41. Nesta classe é possível determinar a extensão do arquivo a ser criado, as mensagens e janelas exibidas ao usuário. As classes *CyanProjectWizard* e *CyanProjectWizardPage* são referentes a criação de projetos do tipo Cyan e possuem as mesmas funcionalidades das classes para criação de arquivos, citadas anteriormente.

A inclusão de *Extension Points* no arquivo plugin.xml, também é necessária. A configuração da extensão pode ser visualizada na Figura 42. A extensão adiciona além do assistente, um item no menu de criação de novos projetos ou arquivos. Estes itens estão disponíveis na IDE Eclipse em:

- File –> New –> Cyan Project
- File –> New –> Cyan Prototype

Esta *Extension Point* define além da classe, o ícone e o texto a ser exibido no menu. Na *tag project*, também é possível definir se a extensão incluída é para um assistente de criação de projetos.

Figura 41 – Trecho da classe *CyanFileWizardPage*

```
public class CyanFileWizardPage extends WizardPage {
    private Text containerText;

    private Text fileText;

    private ISelection selection;

    /**
     * Constructor for CyanNewWizardPage.
     *
     * @param pageName
     */
    public CyanFileWizardPage(ISelection selection) {
        super("wizardPage");
        setTitle("Cyan Prototype");
        setDescription("Creates a new Cyan Prototype");
        this.selection = selection;
    }

    /**
     * @see IDialogPage#createControl(Composite)
     */
    public void createControl(Composite parent) {
        Composite container = new Composite(parent, SWT.NULL);
        GridLayout layout = new GridLayout();
        container.setLayout(layout);
        layout.numColumns = 3;
        layout.verticalSpacing = 9;
```

Figura 42 – *Extension Point* do assistente para criação de arquivo Cyan

```
<extension
      point="org.eclipse.ui.newWizards">
   <category
         id="org.cyanlang"
         name="Cyan-Lang">
   </category>
   <wizard
         category="org.cyanlang"
         class="org.cyanlang.wizards.CyanFileWizard"
         hasPages="true"
         icon="icons/new_file.ico"
         id="org.cyanlang.wizards.CyanFileNewWizard"
         name="Cyan Prototype"
         project="false">
   </wizard>
</extension>
```

A.2.3 Compilador Cyan

O compilador da linguagem Cyan, foi desenvolvido na linguagem Java e ainda não está finalizado. É um projeto externo que foi acoplado e pode ser acionado através do *plug-in* Cyan. Sendo assim detalhes da implementação do compilador não serão fornecidas neste projeto.

Para inclusão do compilador no *plug-in* Cyan, foi necessária a conversão do projeto

do compilador para um arquivo executável do tipo Java. Esse arquivo possui a extensão ".jar" e é adicionado no arquivo de configuração de *plug-ins* "build.properties".

Um *Extension Point* do tipo *ActionSet* foi adicionada ao arquivo plugin.xml. A Figura 43 demonstra a inclusão desta extensão.

Figura 43 – *Extension Point* do compilador Cyan

```
<plugin>
   <extension
         point="org.eclipse.ui.actionSets">
      <actionSet
            id="org.cyanlang.actionSet"
            label="Cyan Action Set"
            visible="true">
         <menu
               id="cyanMenu"
               label="Cyan">
            <separator
                  name="cyanGroup">
            </separator>
         </menu>
         <action
               class="org.cyanlang.actions.CompilerAction"
               icon="icons/sample.gif"
               id="org.cyanlang.actions.CompilerAction"
               label="&amp;Compile Cyan Project"
               menubarPath="cyanMenu/cyanGroup"
               toolbarPath="cyanGroup"
               tooltip="Compile Cyan Project">
         </action>
      </actionSet>
   </extension>
```

Uma *ActionSet* pode adicionar novos menus, itens de menu e botões na barra de ferramentas, que podem ser agrupados em um conjunto de ações que podem aparecer juntos e serem executados ao mesmo tempo.

Na Figura 43 é possível observar, que a *ActionSet* adiciona um item de menu e aciona a classe *CompilerAction* do pacote `org.cyanlang.actions`. Essa classe define as ações a serem realizadas quando do clique no item do menu. No nosso caso, é passado um conjunto de parâmetros e o compilador é acionado.

O compilador então chama o *plug-in* EMS, que é descrito nas sessões seguintes.

A.3 O *plug-in* EMS

O *plug-in* EMS, acrescenta ao ambiente de desenvolvimento Eclipse, as funcionalidades descritas no Capítulo 3. Essa sessão descreve alguns detalhes técnicos do desenvolvimento do *plug-in*. O EMS funciona em duas camadas. Podemos chamá-las de *back-end* e *front-end*.

Na camada *back-end*, temos as funcionalidades que alteram as mensagens de erro dos compiladores. Essas funcionalidades não são visíveis aos usuários e ocorre quando o compilador faz uma chamada ao *plug-in*. Já a camada *front-end* disponibiliza ao usuário as ferramentas para criação e incorporação das LEDs no compilador.

O compilador utilizado no desenvolvimento deste projeto foi o compilador da linguagem Cyan. Este está acoplado ao *Workbench* e faz o envio de mensagens ao *plug-in* EMS, que devolve as informações em forma de mensagens de erro. Maiores detalhes sobre as camadas são descritos nas sessões a seguir.

A.3.1 Camada *front-end*: Interface do usuário

Quando da inclusão do *plug-in* EMS no *Workbench*, um ícone na barra de ferramentas e um item de menu são inclusos ao ambiente Eclipse. Essas funcionalidades são adicionadas através de *Extension Points* do tipo *ActionSets* presentes no arquivo plugin.xml.

Esses itens, ao serem clicados realizam a abertura do formulário para desenvolvimento das LEDs e configuração do EMS. A Figura 44 apresenta a visão inicial do EMS ao usuário.

Podemos observar que, o primeiro formulário exibido, corresponde a configuração do local onde as mensagens de erro serão salvas.

O EMS não utiliza ferramentas de banco de dados para gerenciar as mensagens de erro. As mensagens são organizadas em pastas, conforme visualizado na Figura 45. Cada pasta é identificada com o nome do erro, os espaços são substituídos pelo caractere "_".

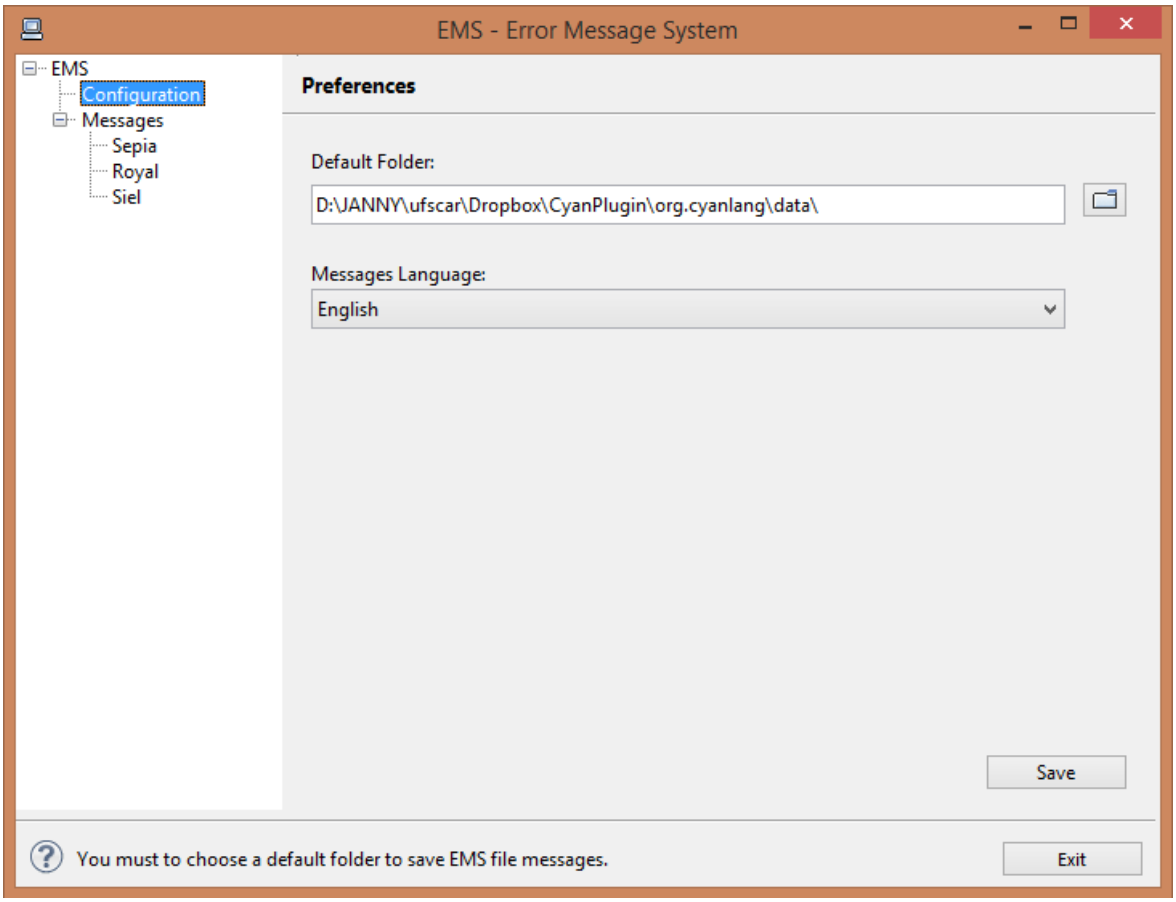
Esses diretórios são gerados quando ocorre a inclusão de uma nova mensagem de erro ou, quando da importação dos parâmetros específicos de cada erro, ou quando da inclusão de uma mensagem de erro.

Esta estrutura de pastas foi escolhida porque ela permite ao usuário colocar seus arquivos de mensagens (Siel, Royal, Sepia) no diretório escolhido e o EMS os incorpora ao compilador, sem que nenhuma ação adicional seja necessária. A Figura 46 exibe o conteúdo pasta "variable_was_not_declared".

Cada pasta, contém em seu interior: um arquivo do tipo Siel, que contém a definição dos parâmetros do erro, zero ou mais arquivos do tipo Royal, para as mensagens curtas e causas do erro e zero ou mais arquivos do tipo Sepia, para as mensagens detalhadas, exemplos e explicações sobre os erros.

Os nomes dos arquivos são definidos pelo usuário e são acrescido de uma seqüência aleatória de seis caracteres gerada pelo sistema. Isso permite ao usuário salvar várias mensagens com o mesmo nome, sem que as mesmas sejam substituídas. O que é útil

Figura 44 – Tela inicial do EMS



quando o usuário adquire as mensagens de outra pessoa.

Voltando a Figura 44, temos a direita uma estrutura em árvore para a configuração das mensagens de erro. Ao clicarmos em um item de menu, uma nova janela é exibida. Veja como exemplo a Figura 47:

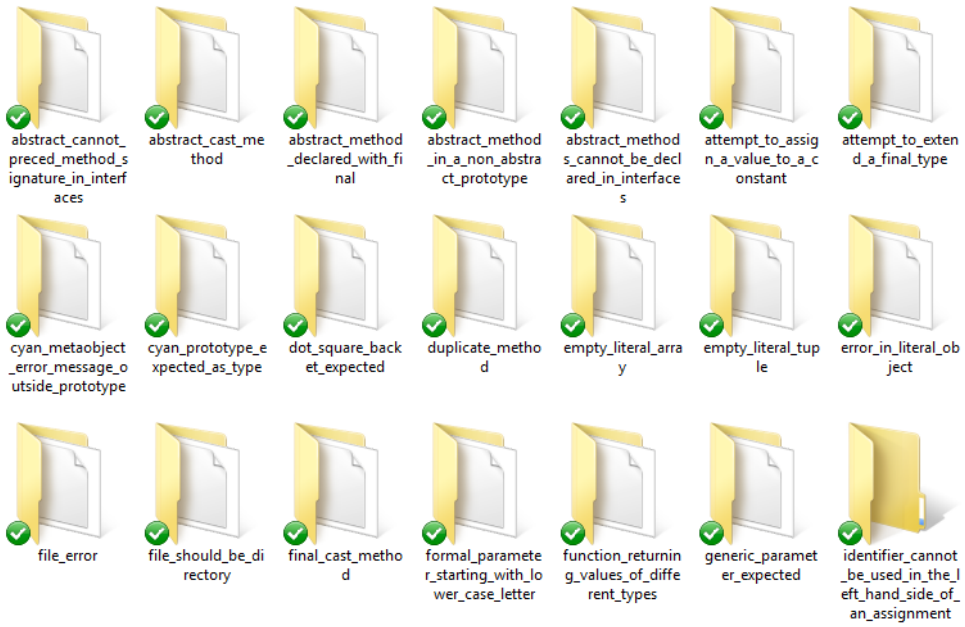
Essa tela lista as mensagens de erro do tipo Sepia existentes no diretório configurado pelo usuário. As mesmas funcionalidades são implementadas para as linguagens Royal e Siel.

O usuário tem a opção de selecionar uma mensagem, edita-la ou exclui-la. Assim como adicionar uma nova mensagem, conforme pode ser visto na Figura 48.

Como o código da linguagem Sepia é um *template*, os parâmetros a serem utilizados na composição da mensagem de erro são exibidos ao usuário. Com duplo clique o parâmetro é incluído no texto precedido do caractere #. Esses parâmetros serão substituídos pelos valores correspondes, quando forem incorporados ao compilador.

Para geração das mensagens de erro, o usuário deve preencher os campos constantes nos formulários Siel, Sepia e Royal e em seguida clicar no botão Salvar. As telas foram formuladas para guiar o usuário, não sendo necessário portanto, detalhar o preenchimento

Figura 45 – Estrutura dos arquivos das mensagens de erro



dos campos.

Essa ação gera um arquivo de texto no diretório correspondente. Esses arquivos serão incorporados ao compilador através do método `signalCompilerError`. Maiores detalhes são dados na sessão seguinte.

Os formulários foram desenvolvidos na linguagem Java, utilizando os componentes gráficos *Swing* e *AWT*. Sua função é a geração e interpretação das mensagens de erro. Quando uma mensagem é colocada em um diretório, a mesma é compilada antes de ser incorporada ao compilador que irá exibir as mensagens de erro. Essa mensagem pode ser compilada em dois momentos: quando o usuário escolhe editar a mensagem, ou quando o EMS vai exibi-la.

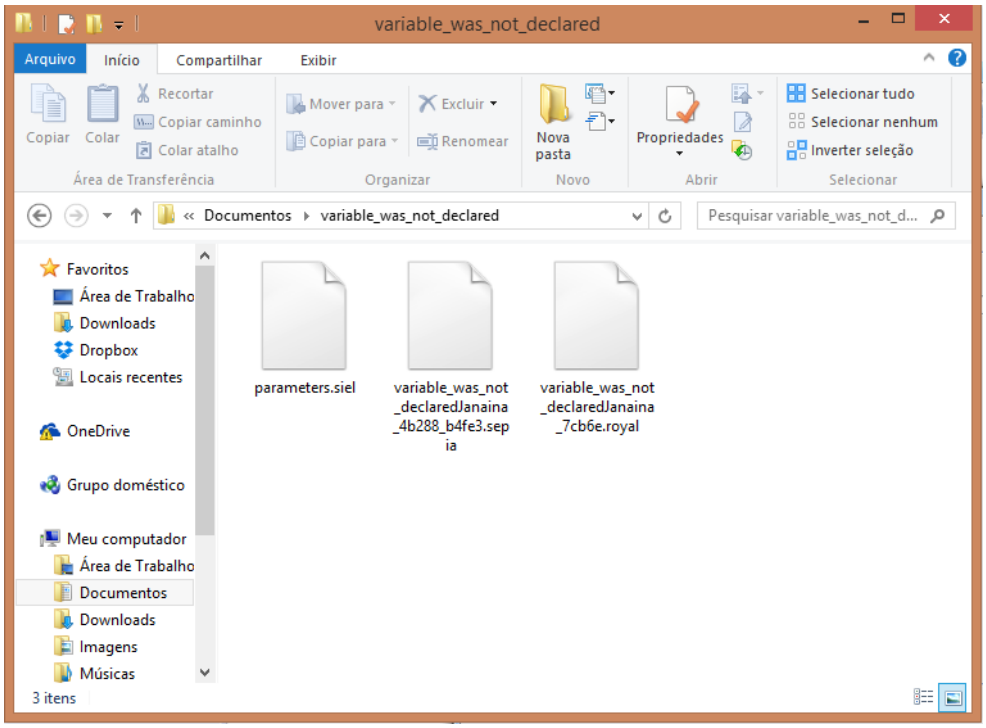
Cada linguagem possui o seu compilador, que faz a análise sintática, léxica e semântica. Nos compiladores das LEDs não há geração de código. Se um erro ocorrer durante a compilação, uma mensagem de erro é exibida ao usuário e a LED é ignorada, não sendo portanto, incoporada ao compilador para sinalizar o erro.

A.3.2 Camada *back-end*

A camada *back-end* pode ser definida como a camada responsável pelo fluxo de informações entre o EMS e o compilador. A Figura 49 apresenta o funcionamento do *plug-in* EMS no momento que o compilador sinaliza um erro.

O componente essencial para funcionamento dessa camada, é a classe *signalCompilerError*. Essa classe é responsável por interpretar a mensagem enviada pelo compilar,

Figura 46 – Conteúdo da pasta `variable_was_not_declared`



converter e devolver ao compilador, as informações em forma de mensagem de erro.

No momento que ocorre o erro, o compilador envia uma mensagem ao método *signalCompilerError*, essa mensagem contém informações sobre o ambiente que ocorreu o erro. A Figura 50 apresenta um trecho do código que compõe o método.

Esse método recebe algumas informações como parâmetros:

a) compilationUnitText: todo o texto do arquivo que gerou o erro;

b) file: nome do arquivo com erro;

c) line: linha que ocorreu o erro no arquivo;

d) column: coluna que ocorreu o erro;

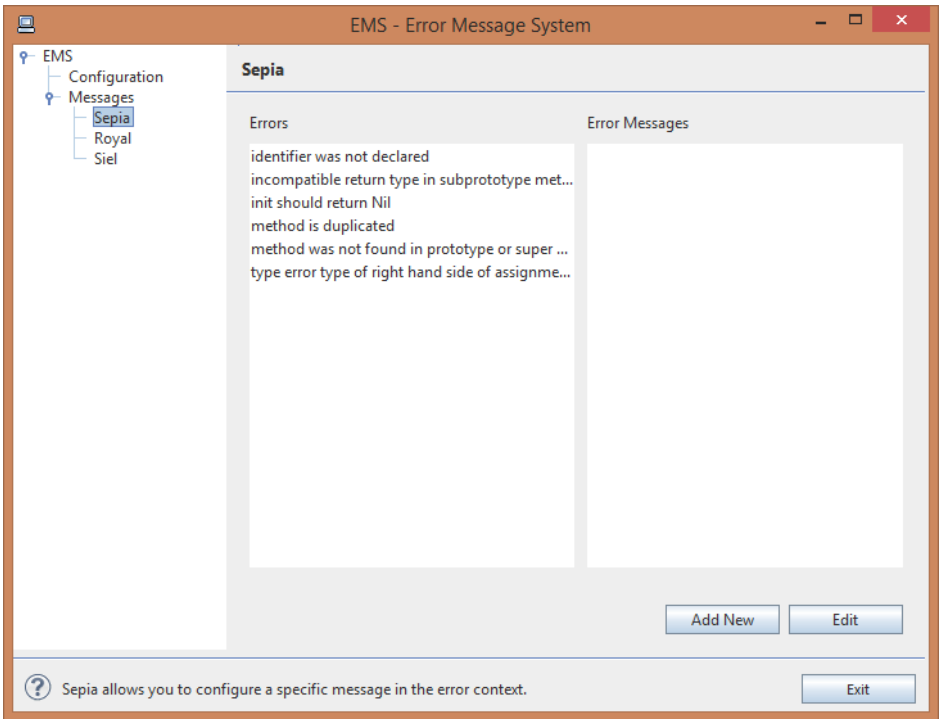
e) sielCode - código em Siel, parâmetros necessários para exibição da mensagem;

O parâmetro *compilationUnitText* é utilizado para compor o item "código anterior do usuário com o mesmo erro". Os parâmetros *file*, *line* e *column* fornecem as informações necessárias para que o EMS sinalize o erro no código do usuário. Já o parâmetro *sielCode*, fornece as informações necessárias para composição da mensagem em Sepia.

Quando o método recebe esses parâmetros, executa as rotinas para a composição das mensagens de erro. O primeiro passo, é verificar se o código em Siel recebido, corresponde aos parâmetros de cada erro.

A seguir é verificado se existe para o erro os códigos em Royal e Sepia. Os códigos

Figura 47 – Interface gráfica Sepia



em Sepia são carregados na memória do sistema e os textos iniciados com # são comparados com o código Sepia recebido. Caso sejam correspondentes, os parâmetros são substituídos pelos valores, compondo assim, uma mensagem de erro com trechos do código do usuário.

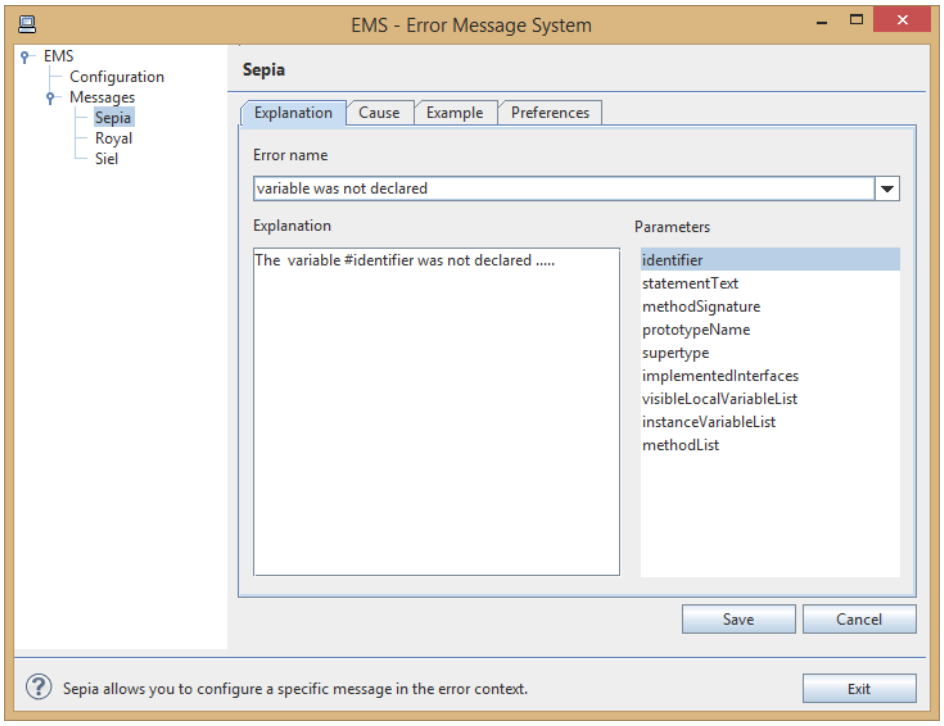
A seguir, o método então cria um marcador no código do usuário, ou seja, sinaliza a linha e a coluna que ocorreu o erro e cria a janela com a mensagem e as opções correspondentes.

Para que isso seja possível, é necessário a utilização do *plug-in* `org.eclipse.core.runtime` e da inclusão do *Extension Point* `org.eclipse.core.resources.markers`, conforme visto na Figura 51, no arquivo plugin.xml.

Essa extensão fornece funcionalidades que permitem a sinalização da linha, do trecho do código, emissão de mensagens, indicadores e geração de tarefas. As mensagens exibidas podem ser de erro ou informativas.

Para o EMS, somente a exibição de mensagens não é suficiente. Durante a composição da janela, além da mensagem curta exibida é imprescindível que seja oferecido ao usuário outras opções de ajuda. Para isso, é necessário a utilização de *Marker resolutions*. Quando um *plug-in* define um *Marker resolution*, os *Markers* referentes a problemas podem utilizar os recursos do *Workbench* para definir *Quick fix*. Um *Quick fix* define a ação que irá ocorrer quando o usuário selecioná-lo. Para cada opção de ajuda exibida como um *Quick fix*, foi criado um *Marker resolution* especifico que contém as ações a serem executadas.

Figura 48 – Inclusão de nova mensagem Sepia



A classe `MarkerResolutionGenerator` define os *Quick fix* para cada *Marker* incluído no código do usuário. Veja por exemplo a Figura 52. É possível verificar o comportamento da classe *MarkerResolutionGeneration*, que cria um vetor de resoluções, a direita o pacote `action.quickfix` que contém as classes referentes as resoluções definidas.

Estas classes executam ações quando selecionadas. Por exemplo: durante a exibição da mensagem de erro, o usuário escolhe a opção para verificar as causas do erro. A classe *Causes* do pacote `action.quickfix` será executada. O código desta classe exibe uma outra janela, com uma mensagem contendo as causas do erro. A *Extension Point* `org.eclipse.ui.ide.markerResolution`, conforme a Figura 53 também foi incluída no arquivo plugin.xml.

Quick fix também são utilizados para correção de erros. Essa parte ainda não foi implementada no EMS, sendo prevista em trabalhos futuros.

A.4 Considerações Finais

Os códigos fonte dos *plug-ins* são abertos e estão documentados. Os projetos estão disponíveis para download no link: “http://emsplugin.xyz”.

Figura 49 – Estrutura do *plug-in* EMS

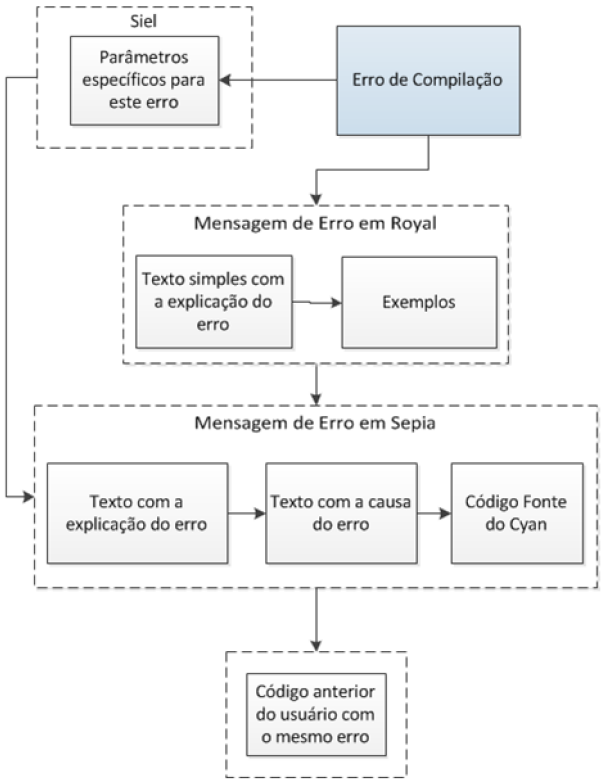


Figura 50 – Trecho do método "signalCompilerError"

```
public static void signalCompilerError(char[] compilationUnitText,
        String file, int line, int column, ArrayList<String> sielCode) {

    String message = null;

    int start = column;
    errorName = sielCode.get(0).substring(start,
    sielCode.get(0).length() - 1);


    IProject[] prj = ResourcesPlugin.getWorkspace().getRoot().getProjects();
    IResource resource = null;

    for (int i = 0; i < prj.length; i++) {

        resource = prj[i].getProject().getFile(file);

        if (resource.exists()) {

            try {
```

Figura 51 – *Extension Point* para sinalização da linha que ocorreu o erro

```
<extension
      id="ufscar.ems.markers"
      name="EMS Problem"
      point="org.eclipse.core.resources.markers">
   <super
         type="org.eclipse.core.resources.textmarker">
   </super>
   <super
         type="org.eclipse.core.resources.problemmarker">
   </super>
   <super
         type="org.eclipse.core.resources.bookmark">
   </super>
   <persistent
         value="true">
   </persistent>

</extension>
```

Figura 52 – Estrutura do *Imarker Resolution*

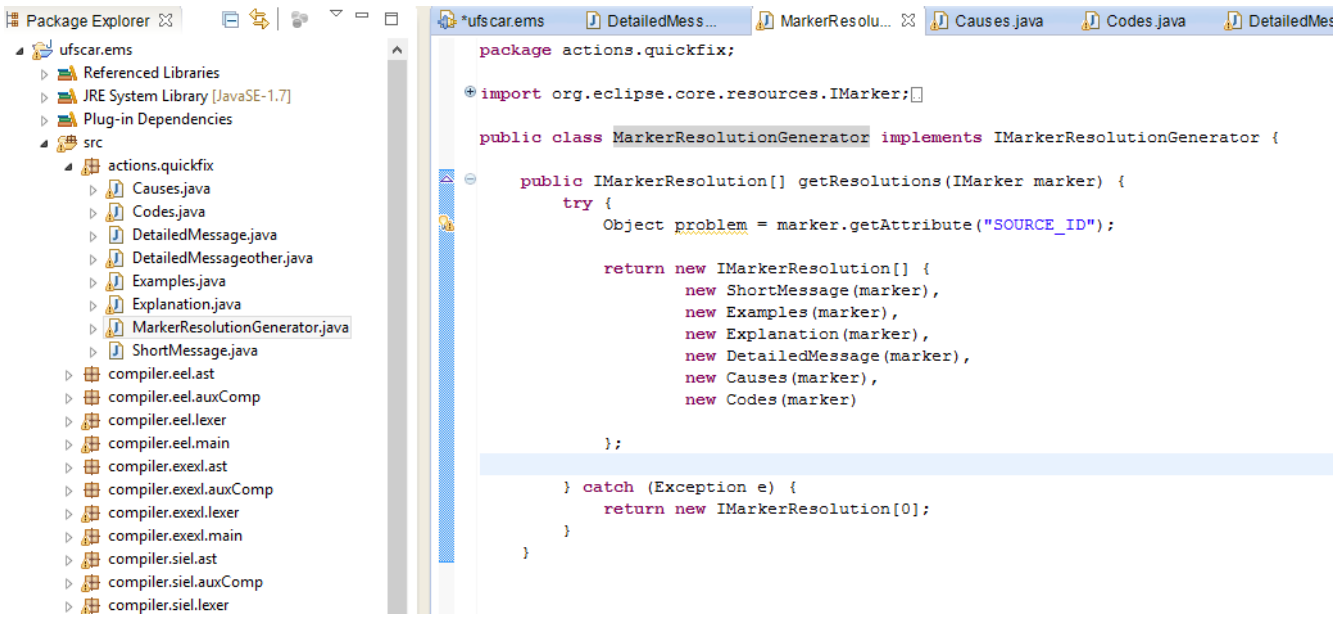


Figura 53 – *Extension Point Imarker Resolution*

```
<extension
      point="org.eclipse.ui.ide.markerResolution">
   <markerResolutionGenerator
         class="actions.quickfix.MarkerResolutionGenerator"
         markerType="org.eclipse.core.resources.problemmarker">
   </markerResolutionGenerator>
</extension>
```