UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

PLUGIN PARA O EA: GERAÇÃO DE INTERFACES DE USUÁRIO A PARTIR DE UM PROJETO DE TELAS

CLEITON EDUARDO SATURNO

BLUMENAU
2007

2007/1-07

CLEITON EDUARDO SATURNO

PLUGIN PARA O EA: GERAÇÃO DE INTERFACES DE USUÁRIO A PARTIR DE UM PROJETO DE TELAS

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. Fabiane Barreto Vavassori Benitti, Dra - Orientadora

BLUMENAU
2007

2007/1-07

PLUGIN PARA O EA: GERAÇÃO DE INTERFACES DE USUÁRIO A PARTIR DE UM PROJETO DE TELAS

Por

CLEITON EDUARDO SATURNO

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Fabiane Barreto Vavassori Benitti, Dra – Orientadora, FURB
Membro:	Prof. Everaldo Artur Grahl, Mestre – FURB
Membro:	Prof. Joyce Martins, Mestre – FURB

Blumenau, 06 de julho de 2007

Dedico este trabalho a todos que direta ou indiretamente contribuíram para a sua realização, em especial à minha mãe, que na completude do papel familiar tornou possível meu ingresso e permanência na universidade.

AGRADECIMENTOS

À Deus, pelo início de Tudo.

À minha família, por contribuir para a minha formação, em todos os aspectos.

À minha namorada, Juliana, pela compreensão nos momentos de minha ausência.

Aos meus amigos, que mesmo sabendo do meu retiro para estudos não esqueceram de me convidar para os momentos de descontração.

À professora Joyce Martins, por iluminar alguns caminhos que levaram a agregação de *templates* nesta obra.

Por fim, embora não menos importante, à minha orientadora, Fabiane Barreto Vavassori Benitti, por ter-me colocado e mantido no caminho desta obra, acreditando e contribuindo de forma singular para a sua conclusão.

Nada é mais difícil de executar, mais duvidoso de ter êxito ou mais perigoso de manejar do que dar início a uma nova ordem de coisas. O reformador tem inimigos em todos os que lucram com a velha ordem e apenas defensores tépidos nos que lucrariam com a nova ordem.

Nicolau Maquiavel

RESUMO

Este trabalho apresenta uma ferramenta desenvolvida na forma de *plugin*, com o propósito de realizar a geração de código a partir de um diagrama de telas projetado com o Enterprise Architect. O desenvolvimento em forma de *plugin* é amparado pelo conceito de I-CASE, que propõe formas de integração entre ferramentas CASE com diferentes propósitos. Para a integração são usados recursos da tecnologia COM, mais especificamente o ActiveX, que produz um alto nível de integração entre as ferramentas. A ferramenta gera código para alguns dos principais componentes usados na construção de interfaces de usuário, como botões, caixas de texto, rótulos e botões de checagem. O código é gerado segundo as diretrizes de um *template* definido para este fim, o que traz ao usuário uma grande flexibilidade ao processo. A linguagem do *template* foi especificada usando a notação BNF, e os analisadores léxico e sintático foram gerados a partir da ferramenta GALS.

Palavras-chave: Interface de usuário. Geração de código. I-CASE. Enterprise Architect.

ABSTRACT

This work describes a tool developed as a plugin, with the intention to carry through the code generation from a projected diagram of screens with the Enterprise Architect. The development in the form of plugin is supported by I-CASE concept, that considers concepts of integration between different intentions tools. For the integration they are used resources of COM technology, more specifically ActiveX, that produces a high level integration between the tools. The tool generates code for some of the main used components in the construction of user interfaces, as buttons, text boxes, labels and check buttons. The code is generated according to one lines of direction template defined for this end, what it brings to user great flexibility to process. The language of template was specified using BNF notation, and syntactic and lexical analyzers had been generated from GALS.

Key-words: User interface. Code generation. I-CASE. Enterprise Architect.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

AI – *Automation Interface*

API – *Application Programming Interface*

AWT – *Abstract Window Toolkit*

CASE – *Computer-Aided Software Engineering*

COM – *Component Object Model*

CPU – *Central Processing Unit*

DLL – *Dynamic Link Library*

EA – Enterprise Architect

EAP – Enterprise Architect *Project*

GALS – Gerador de Analisadores Léxicos e Sintáticos

GUI – *Graphic User Interface*

GUID – *Globally Unique Identifier*

I-CASE – *Integrated* CASE

IDE – *Integrated Development Enviroment*

JVM – *Java Virtual Machine*

OLE – *Object Linking and Embedding*

SO – Sistema Operacional

SGBD – Sistema Gerenciador de Banco de Dados

SQL – *Structured Query Language*

UML – *Unified Modeling Language*

VMT – *Virtual Method Table*

WAE – *Web Application Extension*

SUMÁRIO

1 INTRODUÇÃO

O crescente aprimoramento das relações de consumo, bem como o aumento exponencial da concorrência na área de desenvolvimento de sistemas remetem a uma reflexão sobre os padrões de qualidade existentes que tangem o processo de desenvolvimento de software. Neste ambiente, a Engenharia de Software faz uso de conceitos amplamente estudados e desenvolvidos, com a finalidade de apoiar e facilitar o processo de desenvolvimento de software, garantindo mais qualidade ao produto final e, por conseguinte, maior satisfação do usuário.

Assim como muitas tarefas complexas do mundo real podem ser realizadas com auxílio do computador, a Engenharia de Software não é diferente. Existe no mercado algumas ferramentas que propõem automatizar tarefas desta área, cada qual com suas peculiaridades, porém, todas com o propósito de melhorar a qualidade de software; são as ferramentas de engenharia de software apoiadas por computador (*Computer-Aided Software Engineering*, CASE). Segundo Pressman (2002, p. 807), "ferramentas CASE auxiliam gerentes e profissionais de engenharia de software em toda a atividade associada ao processo de software". Por sua vez, Fisher (1990, p. 5) esclarece que "as ferramentas CASE proporcionam impulso porque exploram o processo de projeto e desenvolvimento, geralmente nos estágios iniciais, para depois produzir vantagens de implementação".

Dentre as várias soluções CASE existentes no mercado, uma solução conhecida é o Enterprise Architect (EA), que dá suporte a diversas etapas da engenharia de software. O EA possui, dentre seus diagramas, um recurso que permite ao engenheiro de software criar protótipos de tela de sistema, possibilitando a inclusão de vários controles visuais, tais como `button`, `checkbox`, `combobox`, `panel`, `edit`, entre outros. Apesar da versatilidade provida por este recurso, há uma deficiência que pode, por vez, inibir o seu uso: não é possível, a partir deste protótipo desenhado com o EA, gerar automaticamente o código fonte das telas do sistema. Em suma, há um retrabalho por parte da equipe, pois, após modelar as telas do sistema com o uso do EA, é necessário fazer este trabalho novamente na fase de implementação.

Diante deste contexto, é com o intuito de preencher esta lacuna deixada pelo EA que este trabalho é desenvolvido. O objetivo é desenvolver um *plugin*[1] para o EA para a geração

[1] *Plugin* é um programa de computador que serve normalmente para adicionar funcionalidades a outros programas (PLUGIN, 2006).

automática de código a partir das interfaces de usuário com ele projetadas, com o propósito de reaproveitar de maneira produtiva as telas modeladas com a ferramenta. O desenvolvimento do trabalho na forma de um *plugin* vem amparado pelo conceito de CASE Integrado (*Integrated* CASE, I-CASE), que traz a idéia de compartilhamento de informações e recursos entre ferramentas CASE com diferentes propósitos (PRESSMAN, 2002, p. 815). Para prover o compartilhamento das informações entre a aplicação proposta e o EA são usados recursos providos pela tecnologia *Component Object Model* (COM), que propõe formas transparentes de comunicação entre aplicações que rodam sobre Windows. As características desta tecnologia são devidamente abordadas no capítulo 2.3.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um *plugin* para a geração de código de telas de sistema a partir de um modelo de interface de usuário projetado com o EA.

Os objetivos específicos são:

a) gerar código fonte dos seguintes componentes visuais do diagrama de telas do EA, em sua versão 6.5: `screen`, `button`, `checkbox`, `combobox`, `panel`, `radio` e `textbox`;

b) gerar código fonte para as linguagens Java e Delphi.

Tem-se, ainda, como objetivo secundário do trabalho, a criação de uma linguagem que permita a definição de *templates*[2] para a geração de código para as linguagens anteriormente elencadas.

1.2 ESTRUTURA DO TRABALHO

Este texto está organizado em quatro capítulos. O capítulo 2 trata dos alicerces teóricos utilizados no desenvolvimento do trabalho. São abordados conceitos de I-CASE e a relação

[2] *Templates* são arquivos que contêm variáveis e blocos especiais que serão dinamicamente transformados a partir de dados enviados a ele (ROCHA, 2006).

deste com o trabalho proposto; são relacionadas características do EA, enfatizando o processo de criação de telas de sistema e a abertura para integração com outras ferramentas; também são delineados aspectos da tecnologia COM, justificando seu uso na integração entre aplicações. O capítulo 2 trata, ainda, de geração automática de código e de características das interfaces de usuário em Delphi e em Java. O capítulo 3 trata do desenvolvimento da ferramenta proposta, apresentando os requisitos funcionais e não funcionais, os diagramas de caso de uso, de classes e de componentes. Neste capítulo são apresentadas, ainda, as ferramentas utilizadas no processo de desenvolvimento, a implementação e a operacionalidade da ferramenta. Por derradeiro, o capítulo 4 apresenta as conclusões e possíveis extensões para este trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Nas seções seguintes são apresentados, de forma mais detalhada, alguns aspectos sobre I-CASE, bem como funcionalidades providas pelo EA em termos de engenharia de software, especificamente os projetos de interface de usuário, e também em termos de abertura para integração com outros aplicativos, focando o uso de tecnologias COM. São também apresentados alguns aspectos sobre geração automática de código. Na seqüência são delineadas características de interfaces de usuário em Delphi e em Java.

2.1 I-CASE

Atualmente as empresas de desenvolvimento de software têm encontrado muitos desafios, alguns inerentes ao nicho, como o aumento da complexidade dos softwares e o surgimento de novas tecnologias causando, muitas vezes, a obsolescência de processos e tecnologias outrora satisfatórios; outros são relativos à própria relação de consumo, como aumento da concorrência no mercado e maior exigência por parte dos consumidores. São os impactos causados por este dinamismo do mercado que as ferramentas CASE e, neste caso, as I-CASE, tentam minimizar (CHEN; NORMAN, 1992, p 362).

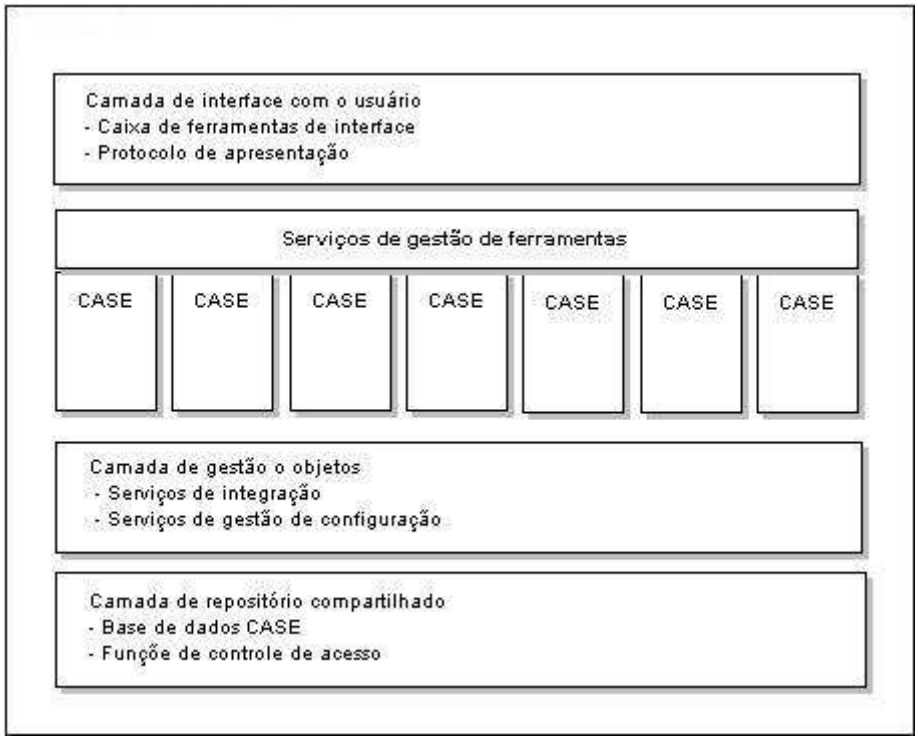
> I-CASE [...] fornecem uma solução integrada que não apenas reduzem custos de desenvolvimento e manutenção, mas também possibilitam que as organizações desenvolvam sistemas de informação melhores e mais rapidamente, e que reajam com mais eficiência às mudanças internas e aos desafios externos. (CHEN; NORMAN, 1992, p. 362, tradução nossa).

O conceito de integração de ferramentas CASE abrange aspectos que atingem todas as aplicações. Pressman (2002, p. 817) denota que as I-CASE são concebidas sobre um modelo arquitetural composto pelas seguintes camadas (conforme Figura 1):

a) interface de usuário: formada por um protocolo comum de apresentação, que tem por objetivo garantir a mesma aparência em todas as ferramentas, convencionando leiautes de tela, menus e formas de entrada pelo mouse e teclado;
b) ferramentas: nesta camada estão as ferramentas CASE propriamente ditas, mantidas por um serviço de gestão de ferramentas, que coordena atividades multitarefas, bem como o fluxo de informação;

c) gestão de objetos: esta fatia do arcabouço é responsável pela comunicação entre as ferramentas CASE do ambiente integrado. Ela provê a comunicação entre os objetos, realizando a manipulação das informações trazidas do repositório;

d) repositório compartilhado: compreende a base de dados e as funções de controle de acesso das ferramentas integradas. Todos os dados manipulados pelas camadas superiores são persistidos nesta camada, e dela provêm as informações que abastecem as demais.



Fonte: Pressman (2002).

Figura 1 - Modelo arquitetural I-CASE

Dentre as camadas supra citadas, o repositório compartilhado merece especial atenção, pelo fato de desempenhar um papel bastante amplo no modelo arquitetural de I-CASE. Além das funções elementares de um sistema de gerenciamento de base de dados, o repositório compartilhado realiza outras funcionalidades específicas. Pressman (2002, p. 818-819) relaciona uma série destas funcionalidades, quais sejam:

a) integridade dos dados: funções de validação de entrada de dados, alterações em objetos e reflexão em cascata sobre todos os objetos relacionados;

b) compartilhamento de informação: realiza o controle de acesso simultâneo sobre os objetos do repositório, colaborando com a integridade dos dados;

c) integração dados/ferramenta: fornece uma interface comum à todas as ferramentas integradas, permitindo o acesso aos objetos do repositório;

d) integração dados/dados: realiza o controle de relacionamento e de acesso entre os

objetos;

e) imposição de metodologia: trata da definição de um modelo de entidade-relacionamento. Compreende esta funcionalidade, no mínimo, a definição de uma seqüência de passos para guarnecer o repositório;
f) armazenamento de dados complexos: o repositório deve permitir, além dos tipos básicos de dados, o armazenamento de estruturas de dados complexas, tais como diagramas e arquivos;
g) interface rica em semântica: o modelo de informação deve ser construído de forma que permita às diversas ferramentas o uso transparente das informações. Assim, uma mesma informação armazenada no repositório poderá ter utilidades diferentes em cada ferramenta do ambiente, sem a necessidade de conversões;
h) gestão de processos e projeto: o repositório não se limita apenas a guardar e prover informações concernentes ao software que está sendo desenvolvido, devendo, também, fazê-lo em relação às informações particulares de cada projeto, e sobre o processo de desenvolvimento adotado para a aplicação.

A partir das funcionalidades retro citadas, tem-se a noção de que o repositório compartilhado não compreende apenas de uma base de dados, e tampouco se restringe às funcionalidades providas por um Sistema Gerenciador de Banco de Dados (SGBD). Além de tudo que se pode esperar de um SGBD, o repositório compartilhado tem funcionalidades intrinsecamente ligadas ao arcabouço proposto para as I-CASE, definindo como as informações serão armazenadas e como são acessadas, contribuindo para a harmonia entre as ferramentas, bem como para o sucesso das atividades das demais camadas deste arcabouço.

Algumas estratégias são propostas para a implementação de um ambiente I-CASE. Chen e Norman (1992, p. 367-398) citam algumas delas, conforme segue:

a) componentes CASE: propõe o uso de ferramentas CASE individuais, que se comunicam através do uso de um repositório comum, por meio de conversores de dados específicos, ou através de técnicas de importação e exportação. Esta abordagem traz flexibilidade ao ambiente, todavia, o nível de integração fica prejudicado. Outro benefício trazido é o baixo custo inicial, a despeito de um custo maior a longo prazo. Por fim, esta estratégia pode se tornar inviável quando são feitas alterações nas ferramentas, que podem refletir nas demais, gerando um considerável esforço (custo a longo prazo) para a adaptação do ambiente;
b) CASE baseado em metodologia: são ferramentas CASE que se integram baseadas em alguma metodologia específica de desenvolvimento. Esta estratégia é muito

eficiente no desenvolvimento de determinadas aplicações, contudo, mostra-se relativamente fraca em termos de flexibilidade, pois não se adapta às várias metodologias de desenvolvimento existentes. Em termos de custos, tem-se um alto investimento inicial, principalmente em treinamento referente a metodologia adotada;

c) CASE abertas: esta estratégia propõe a construção de um *framework* de integração, com mecanismos para inserção e integração de novas ferramentas no ambiente integrado. Estes mecanismos devem compreendem interfaces para a comunicação entre as ferramentas, acesso aos modelos de dados e ao repositório compartilhado, bem como às interfaces públicas de todas as ferramentas.

Conhecendo estas estratégias pode-se, a partir de uma análise do contexto da organização, ou até mesmo do contexto do software que está sendo desenvolvido, escolher a que melhor se adapte à situação, observando, se for o caso, a proposição de Chen e Norman (1992, p. 368, tradução nossa), que dizem que "o ponto chave na seleção de um ambiente I-CASE é manter o balanceamento entre integração e flexibilidade".

Dentre as modalidades citadas, o conceito de CASE abertas se encaixa na solução de integração proposta pelo EA, haja vista que a Interface de Automação (*Automation Interface* – AI) disponibilizada, embora não contemple a totalidade de seu modelo de dados (carência suprida com a possibilidade de acesso direto à base de dados), permite uma integração de alto nível, disponibilizando acesso a quase todos os objetos da estrutura de dados. As características de integração do EA, bem como a tecnologia usada para prover esta integração, são devidamente abordadas nas duas seções seguintes, respectivamente.

2.2 A FERRAMENTA EA

> O EA é uma ferramenta que cobre todos os aspectos do ciclo de desenvolvimento, fornecendo suporte para teste, manutenção e controle de mudanças de requisitos. [...] Além dos diagramas e modelos da UML 2, o EA permite a modelagem de processos de negócio, sites web, **interfaces de usuário**, mapeamento e configuração de equipamentos, planos de teste, etc. (LIMA, 2005, p. 41-42, grifo nosso).

A Figura 2 mostra um diagrama de telas projetado com o EA. Pode-se verificar, à direita, uma representação hierárquica das telas que estão sendo projetadas, com seus componentes formando a estrutura de uma árvore, evidenciando a relação de dependência

entre eles. Outro detalhe a ser observado é que podem ser projetadas mais de uma tela em cada diagrama. Para a modelagem das telas e dos componentes visuais são utilizados os controles *Screen* e *UI Control* (destacados à esquerda), respectivamente, sendo que o segundo apresenta diversos estereótipos, o que permite representar diferentes componentes visuais, tais como `button`, `checkbox`, `combobox`, `panel`, `radio`, `textbox`, entre outros.

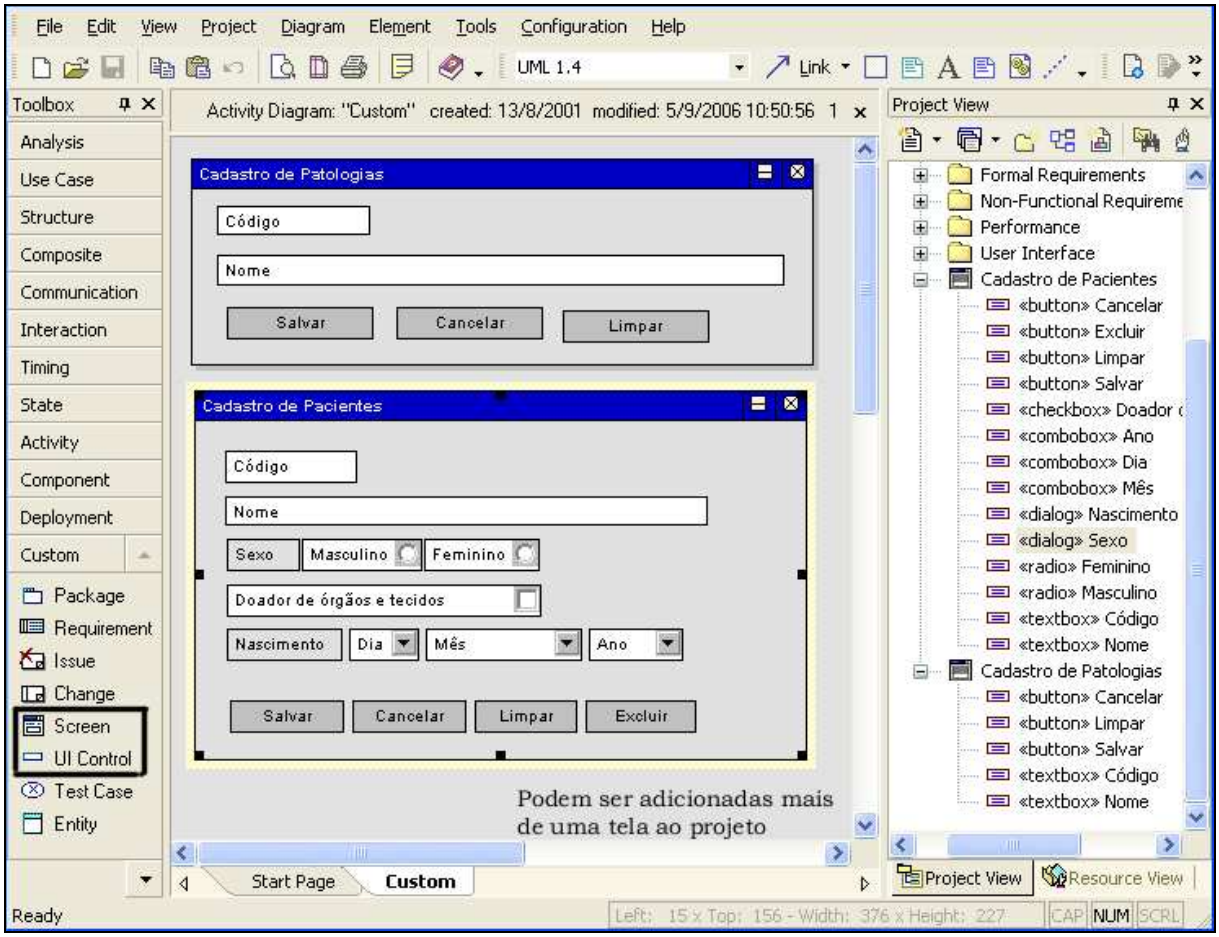


Figura 2 – Diagrama de telas do EA

A Figura 3 traz a representação de um diagrama de telas e, à direita, uma lista de propriedades dos *UI Controls*. Nota-se, na lista, a propriedade *Stereotype*, com vários valores pré-definidos, cada qual podendo representar controles bastante usados e conhecidos pelos desenvolvedores de interface de usuário. Esta propriedade pode receber outros valores que não estejam na lista possibilitando, assim, a representação de qualquer controle visual, trazendo versatilidade para a geração de código.

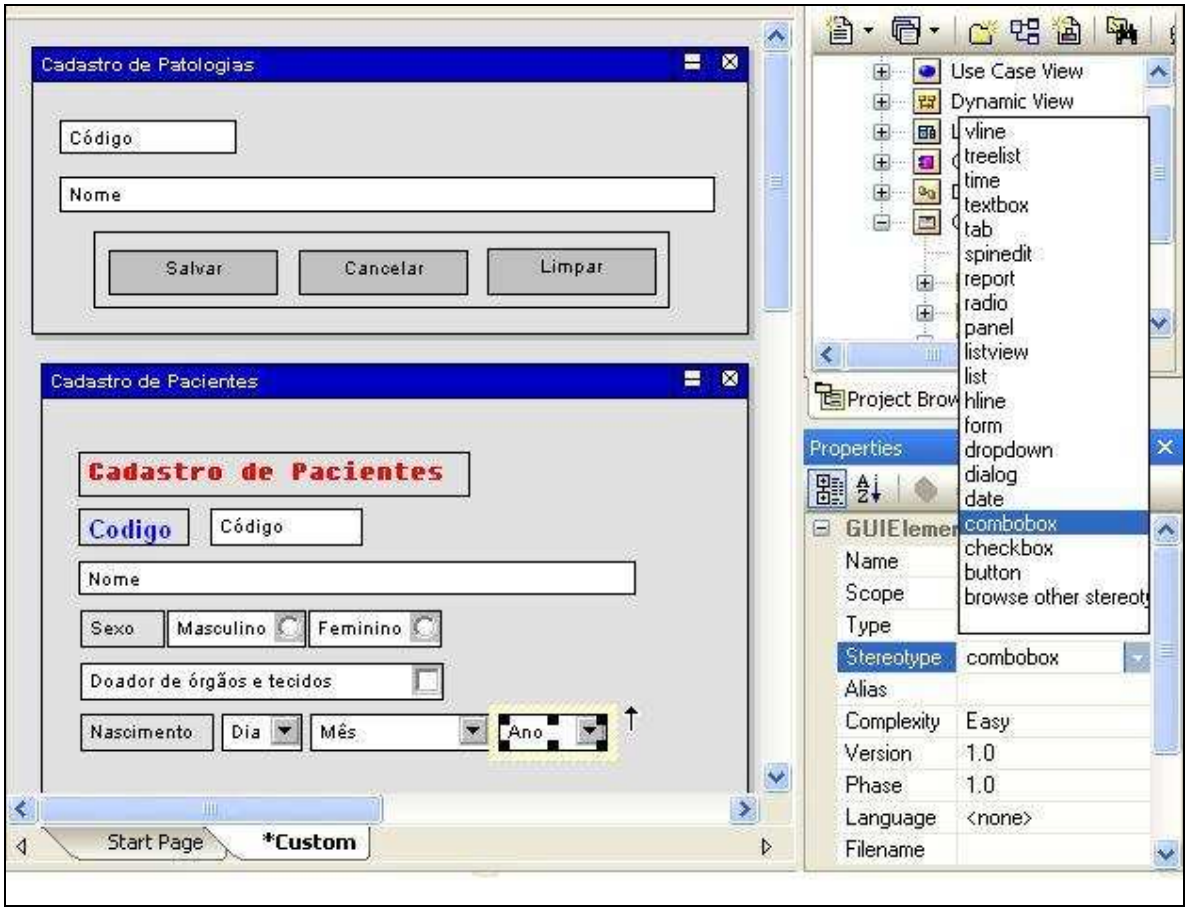


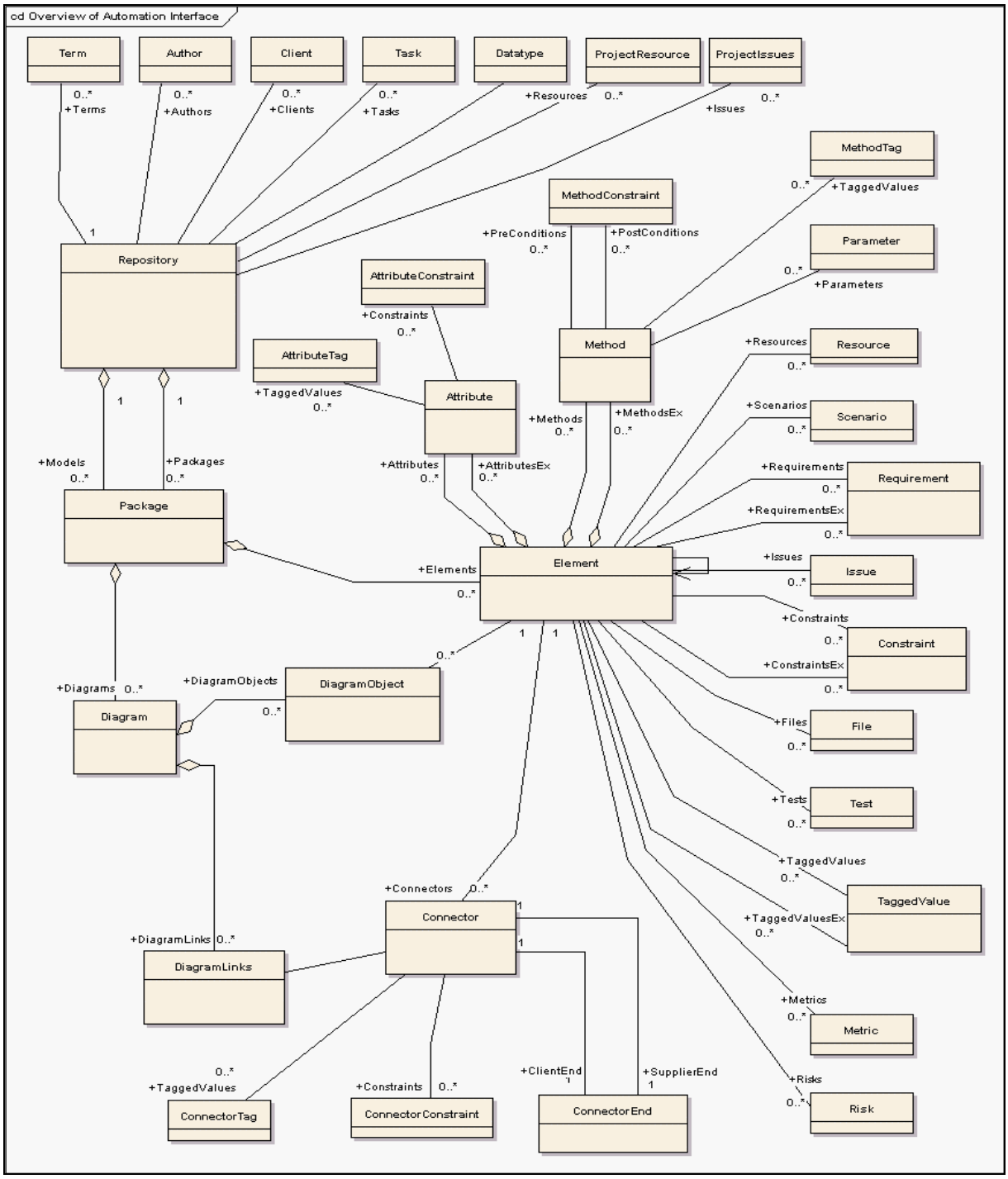
Figura 3 – Propriedade de um GUIElement

Além de automatizar tarefas relacionadas à engenharia de software, o EA permite ainda a extensão de suas funcionalidades através de sua AI. Cantù (2003, p. 373) afirma que uma AI permite que aplicativos se comuniquem entre si, independentemente da linguagem de programação usada para desenvolvê-los.

Segundo Sparx Systems (2007), a AI provê formas de acessar os modelos internos do EA permitindo, por exemplo, as seguintes customizações:

a) efetuar tarefas repetitivas como, por exemplo, alterar o número da versão de todos os elementos de um modelo;
b) gerar código a partir dos diagramas;
c) criar relatórios personalizados;
d) efetuar consultas customizadas.

Sparx Systems (2007) cita, ainda, que qualquer ambiente de desenvolvimento capaz de gerar controles ActiveX (abordados na seção 2.3.1) também é capaz de prover formas de se conectar à AI do EA. A Figura 4 dá uma visão geral dos itens disponibilizados pelo EA para automação.



Fonte: Sparx Systems (2007).

Figura 4 – *Automation Interface* do EA

Este diagrama provê uma visão em alto nível da Interface de Automação para acesso, manipulação, modificação e criação de elementos da UML no EA. O objeto no nível mais alto é o Repository, que contém coleções de todos os demais objetos do diagrama, como elementos, diagramas e pacotes. Em geral, os nomes indicados na extremidade das associações indicam o nome da coleção usada para acessar as instâncias daquele objeto. (SPARX SYSTEMS, 2007, tradução nossa).

2.3 TECNOLOGIA COM

Em meados dos anos 90, a Microsoft decidiu que seus sistemas operacionais e *Application Programming Interface* (API) não mais seriam baseados em funções, como até então, e passariam a ser baseados em objetos reais. O objetivo era que as aplicações que rodassem sobre Windows pudessem compartilhar informações através de seus objetos. Esta nova abordagem, depois de passar por uma série de nomes como *Object Linking and Embedding* (OLE), OLE2, passou a ser chamada de *Component Object Model* (COM). Segundo Cantù (2003, p. 365), o objetivo básico do COM é fazer a comunicação entre dois módulos de software, que podem ser arquivos executáveis ou *Dynamic Link Library* (DLL). Objetos COM, quando implementados em arquivos executáveis ou em DLLs que estejam em outra máquina, são chamados de servidores fora do processo (*out-of-process*) e, quando implementados em DLLs na mesma máquina, são chamados se servidores dentro do processo (*in-process*).

> [...] Implementar objetos COM em DLL geralmente é mais simples pois, no Windows, um programa e a DLL que ele usa residem no mesmo espaço de endereçamento de memória. Isso significa que quando um endereço de memória é passado para a DLL, este endereço continua válido. Quando são usados dois arquivos executáveis, o COM tem muito trabalho nos bastidores para permitir que os dois aplicativos se comuniquem. (CANTÙ, 2003, p. 365).

A tecnologia COM define uma API e um padrão binário para a comunicação entre objetos, independente de linguagem de programação e plataforma, por meio de interfaces. Trata-se de um mapa-padrão que define como os métodos dos objetos são organizados na memória (PACHECO; TEIXEIRA, 2000, p. 617). Em suma, o que acontece por trás do encapsulamento COM nada mais é do que passagem, de um módulo para outro, de endereços de memória, que contêm referências para objetos. Como os objetos COM disponibilizam em suas interfaces todos os métodos que pretendem compartilhar, fica absolutamente transparente a comunicação entre eles.

2.3.1 ActiveX

O fato de um módulo de software poder ser construído na forma de uma DLL traz bastante flexibilidade ao desenvolvimento, haja vista que uma DLL pode ser construída em

qualquer linguagem de programação, e sobre qualquer sistema operacional, desde que sejam observados os padrões determinados pelo COM.

Segundo Cantù (2003, p. 368), uma DLL que contém objetos COM é chamada de biblioteca ActiveX, ou simplesmente controle ActiveX. Para manter a compatibilidade com a tecnologia COM, estas bibliotecas devem exportar funções que permitam o acesso às fábricas de classes do servidor COM, verifiquem se o servidor COM pode ser descarregado da memória, bem como adicionem e removam informações referentes a este do registro do Windows (CANTÙ, 2003, p. 368). Para que estes requisitos sejam contemplados, todo objeto COM deve implementar a interface `IUnknown`. Esta interface possui apenas três métodos, cujas funcionalidades são descritas abaixo:

a) `QueryInterface()`: verifica se o objeto é de um determinado tipo, e se implementa uma determinada interface, cujos identificadores são passados como parâmetro; caso sim, o método retorna um ponteiro para o objeto. Vale ressaltar que cabe ao desenvolvedor tratar o tipo de objeto retornado, implementando as devidas conversões;
b) `_AddRef()`: caso o método `QueryInterface()` retorne um ponteiro para um determinado objeto, `_AddRef()` é chamado para incrementar a contagem de referência deste objeto;
c) `_Release()`: quando o objeto deixa de ser referenciado, este método é chamado para decrementar a sua contagem de referência. Quando esta contagem chega a zero, o objeto pode ser descarregado da memória.

Vale salientar que os métodos `_AddRef()` e `_Release()` devem ser implementados a partir das APIs `InterlockedDecrement` e `InterlockedIncrement` do Win32, para que sejam protegidos contra acessos simultâneos.

2.3.2 Tabelas de métodos virtuais (*Virtual Method Table* – VMT)

De acordo com Cantù (2003, p. 368), convencionalmente uma DLL deve exportar todos os métodos que podem ser acessados por outros aplicativos. Contudo, uma DLL escrita sobre COM não segue este padrão; para que os métodos possam ser acessados por outros aplicativos, a DLL deve fornecer e exportar uma fábrica de classes. As fábricas de classes são objetos usados para criar outros objetos que, por sua vez, possuem uma VMT, que contém os

endereços dos métodos implementados por ele. Sempre que uma classe possuir ao menos um método virtual, todos os objetos desta classe terão uma referência para a VMT. As VMTs compatíveis com COM são independentes de linguagem de programação, pois o COM fornece um padrão para a sua geração.

O Quadro 1 exibe um trecho de código, construído em Delphi, que implementa as classes `TAnimal` e `TGato`, sendo que a segunda herda características da primeira. Para informar ao compilador que um método é virtual, coloca-se a diretiva `virtual` (ou `dynamic`) após a assinatura do método.

```
TAnimal = class
   procedure Comer(); virtual;
   procedure Morrer (); virtual;
end;

TGato = class (TAnimal)
   procedure Comer(); override;
   procedure Ressuscitar (); virtual;
end;
```

Quadro 1 – Métodos virtuais em Delphi

Conforme ilustrado na Figura 5, quando for chamado o método `Comer()`, sobrescrito pela classe `TGato`, será verificado, em tempo de execução, se quem chamou é um objeto da classe `TAnimal`, ou se é da classe `TGato`. Para o programa, na verdade, o que está sendo chamado é o método que reside na posição 1 da tabela do objeto que o chamou. No caso de ser chamado o método `Morrer()`, como a classe `TGato` não o sobrescreve, sua VMT, na posição 2, apontará para o mesmo endereço apontado pela posição 2 da VMT da classe `TAnimal`.

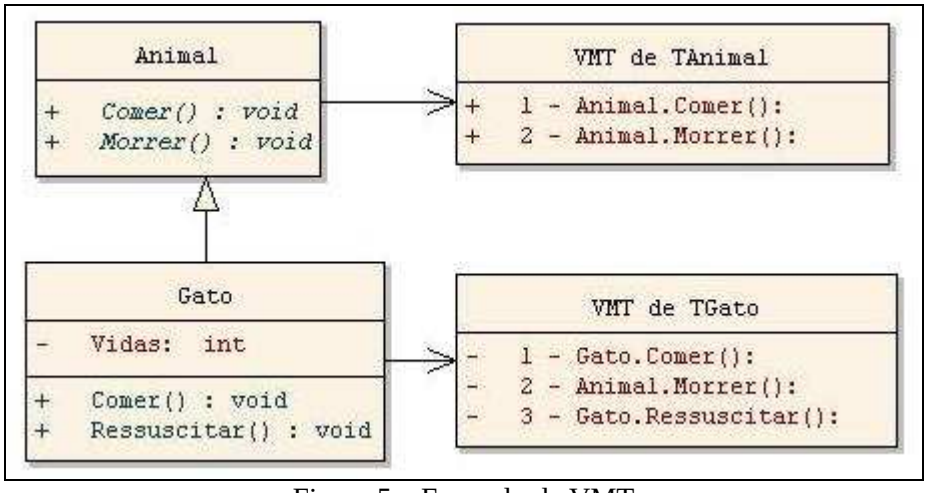


Figura 5 – Exemplo de VMT

2.3.3 Identificadores globalmente exclusivos (*Globally Unique Identifiers* – GUID)

De acordo com Cantù (2003, p. 366-367), as classes e interfaces de um objeto COM devem ser identificadas de forma única em uma aplicação, analogamente a uma chave primária em uma tabela de banco de dados. Esta unicidade é garantida através do uso de GUIDs. Segundo Pacheco e Teixeira (2000, p. 621), um GUID é um número de 128 bits que deve garantir que classes ou interfaces sejam únicas dentro de uma aplicação, e para garantir esta unicidade, estes identificadores são gerados através de um algoritmo que combina informações sobre data e hora do sistema, *clock* de CPU e número de série da placa de rede. Portanto, se a máquina na qual estiver sendo gerado o GUID possuir uma placa de rede corretamente instalada, então este GUID será "garantidamente" exclusivo (dentro desta máquina). E, mesmo que não haja uma placa de rede instalada, é quase impossível que dois GUIDs gerados por este algoritmo sejam iguais. Esta exclusividade é necessária, pois o método `QueryInterface()`, explicado anteriormente, recebe como parâmetro um GUID e, por questões elementares, para que seja retornado o objeto relacionado a este GUID, deve haver apenas uma entidade por ele identificada.

No Windows, um GUID é definido como um registro, conforme demonstrado no Quadro 2, que ilustra sua implementação em Delphi.

```
type
  PGUID = ^TGUID;
  TGUID = packed record
    D1 : Longword;
    D2 : Word;
    D3 : Word;
    D4 : array[0..7] of byte;
  end;
```

Quadro 2 – Implementação do `GUID`

O Quadro 3 mostra a implementação da interface `IInterface`, equivalente a `IUnknow`, também em Delphi, denotando o uso de um GUID para a sua identificação. Conforme explicado, `QueryInterface()` retorna um ponteiro para um objeto.

```
IInterface = interface
  ['{00000000-0000-0000-C000-000000000046}']
  function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
end;
```

Quadro 3 – A interface `IInterface`

2.4 GERAÇÃO AUTOMÁTICA DE CÓDIGO

Segundo Dalgarno (2006), geração de código é uma técnica que usa programas para a construção de código. Há geradores de código para uma grande variedade de aplicações, contudo, os mais comuns são geradores para interface gráfica de usuário (*Graphical User Interfaces*, GUI) e acesso a dados.

Stephens (2006) cita algumas vantagens trazidas pela geração automática de código, tais como:

a) consistência: o código gerado tende a ser mais limpo e claro, reduzindo significativamente a quantidade de erros;
b) customização: caso haja necessidade de mudar a forma de geração, basta alterar o gerador e produzir uma nova versão do código;
c) aumento da produtividade: haja vista que uma quantidade significativa de código é gerada automaticamente;
d) melhor aproveitamento da equipe de desenvolvimento: concentrando os esforços dos programadores em áreas que requerem mais trabalho mental que manual;
e) padronização: os programadores tendem a copiar o estilo do código gerado, criando um padrão para codificação.

A despeito da série de benefícios retro citados, Stephens (2006) cita, ainda, algumas desvantagens que podem ser atribuídas à geração automática de código:

a) o gerador de código deve ser criado primeiramente;
b) é viável apenas em um conjunto restrito de aplicações;
c) sempre haverá uma certa quantidade de código que deverá ser escrita manualmente, e esta quantidade varia de aplicação para aplicação;
d) para geração de código de acesso a bancos de dados, a base de dados deverá estar muito bem formada, e devidamente normalizada.

Por sua vez, Herrington (2003, p. 77) descreve as etapas necessárias para a construção de um gerador de código, conforme segue:

a) escrever o código de saída manualmente: esta etapa facilita a identificação das informações que são necessárias e que devem ser extraídas da entrada do gerador;
b) desenvolver o gerador: deve-se, então, determinar a forma de tratamento das informações de entrada, bem como determinar se as saídas serão geradas através de *templates* ou embutidas no código do gerador;

c) desenvolver o processador de entrada: o processador deverá ler os dados de entrada e extrair as informações necessárias à geração da saída;
d) criar os modelos de saída: o próximo passo é pegar o código desenvolvido na etapa (a) e usá-lo como modelo para a criação do *template* ou para desenvolver o código que gerará a saída;
e) desenvolver o processador de saída: a última etapa consiste em pegar as informações extraídas da entrada e gerar o código de saída.

2.4.1 Geradores de GUI

> Geradores de GUI são ferramentas de engenharia de software que aumentam a produtividade da equipe de desenvolvimento, bem como reduzem custos nas fases de desenvolvimento e manutenção. Um estudo mostra que 48% do código de uma aplicação são direcionados à interface de usuário, e 50% do tempo de desenvolvimento são destinados a esta porção da aplicação. Geradores de GUI podem reduzir significativamente estes números. (CARNEGIE MELLON SOFTWARE ENGINEERING INSTITUTE, 2007, tradução nossa).

"Uma das maiores vantagens da geração de código para GUI é a possibilidade de criação de uma camada de abstração entre a lógica de negócio da interface e sua implementação." (HERRINGTON, 2006, p.100, tradução nossa). Esta abstração consiste, basicamente, em separar o código da parte visual da interface, que pode estar infestado de chamadas a funções de sistema, do código responsável pela parte lógica da interface, como resposta a eventos, por exemplo. Nesta mesma esteira, Carnegie Mellon Software Engineering Institute (2007) ressalta que, em uma aplicação, para se tirar total proveito dos benefícios providos pelos GUI, deve-se optar por uma arquitetura que defina a interface de usuário como uma camada isolada da aplicação, pois este isolamento simplifica a camada de apresentação, facilitando a manutenção e a evolução da GUI.

Além dos benefícios já citados sobre geração automática de código, Herrington (2006) elenca outras vantagens aplicadas às técnicas de geração de GUI, como:

a) consistência: além de consistência, característica singular do processo, a geração automática de GUI traz consigo a vantagem de se poder definir padrões de usabilidade em toda a aplicação;
b) flexibilidade: as GUI podem responder mais rapidamente às mudanças nos requisitos relativos ela;

c) portabilidade: havendo uma representação abstrata da GUI, como diagramas e *templates*, é possível redirecionar o gerador para diferentes tecnologias de interface, como interfaces *web* ou *desktop*.

Todavia, Herrington (2006) salienta, ainda, que geradores de GUI não criarão interfaces necessariamente melhores; seu papel é, na verdade, gerá-las mais rapidamente do que se fossem feitas manualmente. Não obstante, complementa esclarecendo que *templates* ruins gerarão, inevitavelmente, GUIs ruins.

2.5 INTERFACES DE USUÁRIO EM DELPHI

Em Delphi, as interfaces de usuário podem ser criadas em tempo de execução ou em tempo de projeto, usando formulários estáticos no segundo caso. Segundo Peil (2003), em Delphi, “os formulários são uma interface entre o usuário e o programa.” Cantù (2003, p. 203) complementa ao afirmar que construir interfaces em Delphi usando formulários em tempo de projeto é uma prática padrão, apesar de não ser a única forma de fazê-lo.

Os formulários escritos em tempo de projeto têm os valores de suas propriedades, bem como de seus componentes, armazenados em arquivos de formulário, com extensão DFM ou *xfm* (BORLAND SOFTWARE CORPORATION, 2002). Todavia, Fonseca (2005, p. 15) esclarece que nem todas as informações estão presentes no arquivo de formulário, pois as informações padrões de um projeto apenas passarão a constar no arquivo se sofrerem alguma modificação. O Quadro 4 e o Quadro 5 mostram, respectivamente, a definição do arquivo fonte de extensão PAS e a definição do arquivo de extensão DFM de uma interface de usuário desenvolvida estaticamente.

```
unit uInterface;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms, Dialogs, StdCtrls, ExtCtrls;
type
  TTelaExemplo = class(TForm)
    edCodPat: TLabeledEdit;
    edNomPat: TLabeledEdit;
    btSalvar: TButton;
    btCancelar: TButton;
    btLimpar: TButton;
  end;
var
  TelaExemplo: TTelaExemplo;
implementation
{$R *.dfm}
end.
```

Quadro 4 – Código fonte do arquivo .PAS

```
object TelaExemplo: TTelaExemplo
  Left = 192
  Top = 113
  Width = 289
  Height = 199
  Caption = 'TelaExemplo'
  Color = clBtnFace
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object edCodPat: TLabeledEdit
    Left = 32
    Top = 40
    Width = 121
    Height = 21
    // demais propriedades
  end
end
```

Quadro 5 – Código fonte do arquivo .DFM

A possibilidade de criação de interfaces de usuário através de formulários estáticos traz a facilidade de poder conhecer a parte visual da tela antes de executar a aplicação. Contudo, em um sistema que possui uma quantidade muito grande de telas, a criação estática tende a ser muito trabalhosa, dispensando um tempo precioso em uma tarefa demorada e repetitiva. Em contrapartida, existe a possibilidade de criação das interfaces de usuário em tempo de execução. Esta alternativa, apesar de ser consideravelmente maçante, traz mais flexibilidade ao desenvolvimento, haja vista que podem ser usados parâmetros externos para criação das

telas, como valores oriundos de um banco de dados, por exemplo (CANTÙ, 2003, p. 203). Para se ter uma melhor compreensão da vantagem de formulários dinâmicos em projetos de grande porte, pode-se considerar um sistema que possua quarenta tabelas de cadastro: usando-se formulários estáticos, tem-se quarenta arquivos .PAS e mais quarenta arquivos .DFM, cada qual semelhante aos dois arquivos ilustrados anteriormente, e com tamanho variando de acordo com a quantidade de campos que se deseja apresentar ao usuário; já usando formulários dinâmicos, para qualquer quantidade de componentes, tem-se a mesma quantidade de código e, ainda, para todas as telas que obedecerem o padrão definido pode ser usada a mesma rotina. O Quadro 6 mostra o exemplo de um formulário criado dinamicamente.

```
procedure TScreen.CreateComponents;
var c : TComponent;
begin
  Screen := TForm.Create(Owner);

  Screen.Left := 525;
  Screen.Top := 79;
  Screen.Height := 63;
  Screen.Width := 96;
  Screen.Font.Size := 10;
  Screen.Font.Name := 'Arial';
  Screen.Caption := 'F';
  Screen.Color := 16777215;

  c := TBitBtn.Create(Owner);
  ComponentList.add (c);
  TBitBtn(c).Parent := Screen;
  TBitBtn(c).Top := 7;
  TBitBtn(c).Left := 31;
  TBitBtn(c).Height := 18;
  TBitBtn(c).Width := 29;

// demais propriedades

  Screen.Show;
end;
```

Quadro 6 – Tela criada dinamicamente

Basicamente, para escolher entre uma e outra forma de criação dos formulários tem-se que atentar a dois detalhes: se são muitas as telas a serem criadas e, sendo afirmativa a resposta, se uma quantidade considerável delas apresenta um padrão em sua construção; se ambas as respostas forem afirmativas, pode valer a pena dispensar um pouco de tempo na criação de formulários dinâmicos.

2.6 INTERFACES DE USUÁRIO EM JAVA

> Um dos principais objetivos de Java é oferecer um ambiente de desenvolvimento independente de plataforma. A área de interfaces gráficas com usuário é uma das partes mais complicadas da criação de código portável, devido a API do Windows ser diferente da API do Mac, que por sua ver é diferente da API do Presentation Manager para OS/2 e assim por diante. [...] Apesar de toda a evolução do Java, ainda é notória a complexidade que o modelo de interface apresenta ao programador, a qual, dependendo da amplitude da aplicação, implica em grande dispêndio de tempo e recurso. (FONSECA, 2005, p. 25).

Assim como Delphi, Java faz uso de bibliotecas específicas para a criação de formulários, porém, muito código tem que ser criado manualmente, apesar de haver Ambientes Integrados de Desenvolvimento (*Integrated Development Enviroment*, IDE) que facilitam um pouco este trabalho. Segundo Trindade (2002), um recurso bastante usado para construção de interfaces em Java é a biblioteca Swing, descendente da biblioteca *Abstract Window Toolkit* - AWT, que traz vantagens ao desenvolvedor como a possibilidade de gerar um conjunto ilimitado de componentes de interface e a independência de sistema operacional ou ambiente de janelas. Contudo, Trindade (2002) ressalta que, por ser independente de sistema operacional, todo o controle dos componentes de interface de usuário do Swing é de responsabilidade da Máquina Virtual Java (*Java Virtual Machine* – JVM), o que gera alguma perda de performance.

Não obstante a bibliografia usar o termo “descendente” para a relação entre as bibliotecas AWT e Swing, não significa a segunda venha substituir a primeira. Silva (2004, p. 32), esclarece que “o Swing é visto como uma camada disposta confortavelmente sobre AWT”. Contudo, existe uma diferença importante entre estas duas abordagens: os componentes AWT fazem chamadas diretamente ao sistema operacional (SO) no qual a aplicação está sendo executada, atribuindo a aplicação um *look-and-feel*[3] diferente em cada SO. Já os componentes Swing são construídos em Java puro, o que faz com que a aplicação tenha o mesmo *look-and-feel* em qualquer SO. Entretanto, estas diferenças não se resumem à forma como a aplicação é apresentada ao usuário. Pelo fato de o Swing ser escrito totalmente em Java, a despeito da portabilidade que esta característica concede, há uma perda de desempenho em relação à biblioteca AWT, haja vista que esta usa objetos nativos do SO, em vez de deixar todo o controle sob a custódia da JVM.

A Figura 6 e o Quadro 7 mostram, respectivamente, uma tela criada em Java com o

[3] Segundo Javafree.org (2007), *look-and-feel* é a aparência que a aplicação assume, e define a forma como os componentes serão desenhados na tela.

uso da biblioteca Swing e código fonte responsável pela criação desta tela.

Figura 6 – Tela construída em Java

```
import javax.swing.*;
import java.awt.*;

public class Controles2 extends JFrame{
  public Controles2(){
    super("Exemplo de Janela");

    Container tela = getContentPane();

    FlowLayout layout = new FlowLayout();
    layout.setAlignment(FlowLayout.LEFT);
    tela.setLayout(layout);

    JLabel rotulo = new JLabel("Seu Nome:");
    JTextField nome = new JTextField(10);
    JButton btn = new JButton("OK!");
    tela.add(rotulo);
    tela.add(nome);
    tela.add(btn);

    setSize(300, 100);
    setVisible(true);
  }

  public static void main(String args[]){
    Controles2 app = new Controles2();
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}
```

Quadro 7 – Código fonte da mesma tela em Java

3 DESENVOLVIMENTO DO TRABALHO

Neste capítulo é apresentado o desenvolvimento da ferramenta proposta, desde o levantamento de requisitos até a fase final de testes. O processo de desenvolvimento foi dividido em algumas etapas a serem consideradas:

a) levantamento dos requisitos: os requisitos elencados na proposta foram refinados e alterados de maneira a atender à necessidade de geração de código através do uso de *templates*;

b) mapeamento de componentes visuais: foi realizado um estudo das classes de componentes de interface disponíveis em Java e Delphi a fim de definir, para cada componente visual do EA, qual a respectiva classe ou componente na linguagem alvo;

c) propriedades dos elementos do EA: identificação das propriedades dos elementos do EA que serão utilizadas no processo de geração de código;

d) especificação da ferramenta: a ferramenta foi especificada com análise orientada a objetos, utilizando os diagramas da *Unified Modeling Language* (UML) de caso de uso, de classes e de componentes, construídos no EA;

e) implementação da ferramenta: para a implementação foi utilizado o Delphi 7. A fase de implementação compreende o desenvolvimento das seguintes rotinas:
   - criação da estrutura de comunicação com o EA, usando os recursos da tecnologia COM,
   - análise da entrada, que compreende a carga em memória das informações provenientes das estruturas de dados do EA,
   - definição de uma linguagem para a criação dos *templates* que darão origem aos códigos fonte,
   - leitura do *template* para a geração de código;

f) testes: os testes da ferramentas foram realizados a partir da construção de diversos protótipos de tela no EA, inserindo todos os componentes visuais propostos e usando a maior quantidade possível de propriedades visuais. Após a construção destes protótipos, foi realizada a geração de código para os *templates* construídos com o propósito de verificar a manutenção da semelhança entre as telas do protótipo e as telas geradas.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A ferramenta proposta deverá atender os seguintes requisitos funcionais (RF) e não funcionais (RNF):

a) gerar para Delphi, a partir do repositório do EA, em tempo de projeto e em tempo de execução, os seguintes componentes de interface de usuário: `screen, button, checkbox, combobox, panel, radio e textbox` (RF);
b) gerar para Java, a partir do repositório do EA, os mesmos componentes de interface de usuário relacionados no item anterior (RF);
c) disponibilizar uma linguagem para a criação de *templates* que contemplem os itens anteriores (RNF);
d) ser compatível com a versão 6.5 Professional Edition para Windows do EA (RNF);
e) as telas geradas em Delphi deverão ser compatíveis com a versões 7.0 do ambiente (RNF);
f) a interface de usuário da ferramenta proposta deverá observar os padrões visuais do EA (RNF);
g) usar o ambiente Delphi 7.0 para a implementação (RNF).

3.2 MAPEAMENTO DE COMPONENTES VISUAIS

Para uma geração de código consistente, tanto para Delphi quando para Java, deve-se definir um mapeamento entre seus componentes visuais e os respectivos componentes do EA. Para isso, foi definido um relacionamento entre estes componentes, que tem a função de identificar, para cada componente visual do EA, qual a respectiva classe ou componente em Java e Delphi. O Quadro 8 mostra este relacionamento.

Componente do EA	Classe em Delphi	Classe em Java
screen	TForm	JFrame
textbox	TEdit	JText
combobox	TComboBox	JComboBox
panel	TPanel	JPanel
radio	TRadioButton	JRadioButton
dialog	TLabel	JLabel
button	TButton	JButton
checkbox	TCheckBox	JCheckBox
custom	TOutraClasse	JOutraClasse

Quadro 8 – Mapeamento de componentes visuais

Conforme pode ser visto na última linha do Quadro 8, com o uso deste mapeamento é possível definir outros tipos de componentes, bastando informar no *template* qual a classe respectiva. A operacionalidade do mapeamento de classes é devidamente abordada na seção 3.5.2.3.

3.3 PROPRIEDADES DOS ELEMENTOS DO EA

Nas linguagens de programação os componentes visuais representam objetos que podem ser “vistos” pelo usuário e, eventualmente, manipulados por ele. Nesta esteira, algumas características destes componentes são essenciais para sua correta identificação, e por isso devem ser consideradas na geração de código, como cor e tamanho de fonte, posição na tela, cor de fundo, entre outras. Contudo, nem todas estas propriedades são disponibilizadas na interface de automação do EA, revelando a necessidade de criação de métodos alternativos para a sua obtenção. O Quadro 9 mostra as propriedades que serão exploradas no processo de geração de código, bem como a forma de acesso pela aplicação.

Propriedade	Tipo	Descriação	Interface de acesso
BackColor	*Integer*	Cor de fundo do elemento	Não disponível na AI. Acesso através do arquivo EAP, tabela **t_object**, campo **BackColor**, sendo que o elemento é identificado na tabela pelo seu GUID ou pelo ElementID.

Propriedade	Tipo	Descriação	Interface de acesso
`BorderColor`	*Integer*	Cor da borda do elemento	Não disponível na AI. Acesso através do arquivo EAP, tabela **`t_object`**, campo **`BorderColor`**, sendo que o elemento é identificado na tabela pelo seu GUID ou pelo ElementID.
`BorderWidth`	*Integer*	Espessura da borda do elemento	Não disponível na AI. Acesso através do arquivo EAP, tabela **`t_object`**, campo **`BorderWidth`**, sendo que o elemento é identificado na tabela pelo seu GUID ou pelo ElementID.
`ElementGUID`	*WideString*	Identificador global do elemento no EA	`Repository.Packages[].Elements[].ElementGUID`
`ElementID`	*Integer*	Identificador numérico do elemento no EA	`Repository.Packages[].Elements[].ElementID`
`FontColor`	*Integer*	Cor da fonte do elemento	Não disponível na AI. Acesso através do arquivo EAP, tabela **`t_object`**, campo **`FontColor`**, sendo que o elemento é identificado na tabela pelo seu GUID ou pelo ElementID.
`Name`	*WideString*	Nome do elemento	`Repository.Packages[].Elements[].Name`
`Notes`	*WideString*	Observações conferidas ao elemento	`Repository.Packages[].Elements[].Notes`
`ObjectStyle`	*WideString*	Estilos do elemento. Deste campo serão extraídos os atributos de cor, tamanho e estilo de fonte.	Não disponível na AI. Acesso através do arquivo EAP, tabela **`t_diagramobjects`**, campo **`ObjectStyle`**, sendo que o elemento é identificado na tabela pela sua PK, que por sua vez é uma FK para a tabela **`t_object`**.
`PackageID`	*Integer*	Identificador do *package* ao qual pertence o elemento.	`Repository.Packages[].Elements[].PackageID`
`ParentID`	*Integer*	Identificador do elemento ao qual o elemento em questão está associado.	`Repository.Packages[].Elements[].ParentID`
`RectTop`	*Integer*	Ponto de início da horizontal superior do elemento, em relação ao topo do diagrama.	Não disponível na AI. Acesso através do arquivo EAP, tabela **`t_diagramobjects`**, campo **`RectTop`**, sendo que o elemento é identificado na tabela pela sua PK, que por sua vez é uma FK para a tabela **`t_object`**.

Propriedade	Tipo	Descriação	Interface de acesso
`RectBottom`	*Integer*	Ponto de início da horizontal inferior do elemento, em relação ao topo diagrama.	Não disponível na AI. Acesso através do arquivo EAP, tabela **`t_diagramobjects`**, campo **`RectBottom`**, sendo que o elemento é identificado na tabela pela sua PK, que por sua vez é uma FK para a tabela **`t_object`**.
`RectLeft`	*Integer*	Ponto de início da vertical esquerda do elemento, em relação à esquerda do diagrama.	Não disponível na AI. Acesso através do arquivo EAP, tabela **`t_diagramobjects`**, campo **`RectLeft`** sendo que o elemento é identificado na tabela pela sua PK, que por sua vez é uma FK para a tabela **`t_object`**.
`RectRight`	*Integer*	Ponto de início da vertical direita do elemento, em relação à esquerda do diagrama.	Não disponível na AI. Acesso através do arquivo EAP, tabela **`t_diagramobjects`**, campo **`RectRight`**, sendo que o elemento é identificado na tabela pela sua PK, que por sua vez é uma FK para a tabela **`t_object`**.
`Scope`	*WideString*	Escopo do elemento (*Public*, *Private*, etc)	`Repository.Packages[].Elements[].Visibility`
`Stereotype`	*WideString*	Estereótipo do elemento (*textbox*, *radio*, etc)	`Repository.Packages[].Elements[].Stereotype`
`Type_`	*WideString*	Tipo do objeto. (`GUIElement`, `Class`, `Screen`, `Boundary`, etc)	`Repository.Packages[].Elements[].type_`

Quadro 9 – Descrição dos elementos visuais do EA

Algumas observações são necessárias para melhor compreensão das informações trazidas no Quadro 9. Grande parte das propriedades dos elementos é propositalmente inacessível através da interface de automação, segundo consta na própria documentação do EA. Contudo, estas propriedades são fundamentais para a manutenção da semelhança visual entre o diagrama construído e as telas geradas; assim sendo, é necessário criar um acesso direto ao arquivo EAP, que representa a base de dados do EA. Como o arquivo EAP é uma base de dados Microsoft JET, pode ser acessado e alterado através de comandos *Structured Query Language* (SQL), o que é feito pela aplicação.

As propriedades `RectTop`, `RectBottom`, `RectLeft` e `RectRight` de todos os elementos têm seus valores definidos com referência no diagrama. Como os elementos precisam que suas coordenadas sejam definidas a partir da tela a qual pertence, é necessário fazer um ajuste para obter estes valores relativos.

A propriedade `ObjectStyle` armazena, em forma de texto, informações referentes a nome, tamanho e estilo (itálico, negrito e sublinhado) da fonte. Por este motivo, para extrair os valores de suas respectivas propriedades é necessário efetuar um processamento sobre o conteúdo do campo. Para este processamento foram usados analisadores léxico, sintático e semântico. A especificação que deu origem a estes analisadores é apresentada no apêndice I. O Quadro 10 mostra um exemplo do conteúdo da propriedade `Objectstyle`.

```
DUID=AEDF404A;font=Arial;fontsz=100;bold=1;italic=0; ul=0;
```

Quadro 10 – Exemplo de conteúdo do campo `ObjectStyle`

3.4 ESPECIFICAÇÃO

A ferramenta proposta foi especificada usando o EA para a construção dos diagramas UML, com base nos conceitos de orientação a objetos. A especificação resultou nos diagramas de caso de uso, de classes e de componentes, que são apresentados e detalhados nas seções seguintes.

3.4.1 Diagrama de caso de uso

O diagrama de caso de uso ilustra as interações do usuário com o programa. Cada caso de uso representa uma ação que o usuário pode realizar na aplicação. A Figura 7 representa o diagrama de caso de uso da ferramenta proposta.

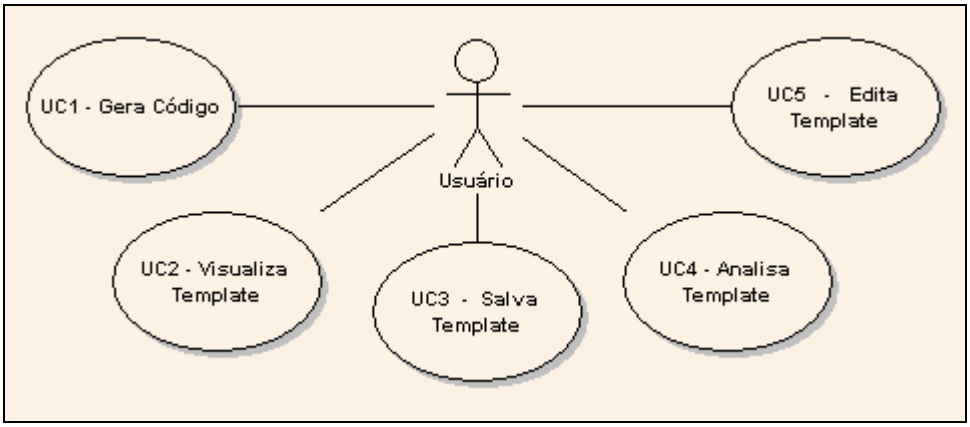


Figura 7 – Diagrama de caso de uso

O Quadro 11, a seguir, apresenta o detalhamento completo do principal caso de uso da

ferramenta.

UC1 – Gera código

Objetivo: Permite ao usuário efetuar a geração de código para uma linguagem de programação, segundo definição de um *template* selecionado para este fim.

Pré-condições:

- Deve haver ao menos um arquivo de mapeamento de classes definido corretamente e assinalado no *template*.
- Deve haver ao menos um *template* corretamente definido.

Pós-condição:

- Será gerado o código para a linguagem alvo, conforme definição do *template*.

Cenário principal:

1. O usuário seleciona uma tela, diagrama ou pacote e acessa o menu para abrir a interface da ferramenta.
2. A ferramenta apresenta a interface para interação com o usuário, trazendo as telas selecionadas pelo usuário e a lista de *templates* disponíveis.
3. O usuário clica para selecionar um diretório.
4. A ferramenta abre a lista de diretórios do Windows.
5. O usuário seleciona um diretório para salvar os arquivos gerados.
6. O usuário seleciona o *template* para a geração de código.
7. A ferramenta atualiza o *template* selecionado nos modelos internos.
8. O usuário clica no botão para gerar código.
9. A ferramenta gera o código de acordo com a definição do *template* selecionado.

Cenários alternativo:

1. No passo 2, caso o usuário opte por visualizar o *template* selecionado, executa UC2.

Cenários de exceção:

1. **Arquivo de *template* inconsistente:** no passo 2, caso algum *template* esteja inconsistente, ou seja, em desacordo com as regras sintáticas pré-estabelecidas, adiciona mensagem de WARNING no *log* e desabilita o referido *template* para geração de código.
2. **Não existe arquivo de *template* válido no diretório:** no passo 2, caso não haja arquivo(s) de *template* corretamente definido(s) no diretório destinados aos *templates*, apresenta mensagem de erro ao usuário e desabilita os botões de interação na tela principal.
3. **Não existe arquivo de mapeamento de classe:** No passo 2, caso algum *template* aponte para um arquivo de mapeamento de classe que não exista, ou o *template* não faça referência a um arquivo de mapeamento de classe, adiciona mensagem de WARNING no *log* e desabilita o referido *template* para geração de código.
4. **Arquivo de mapeamento de classe inconsistente:** No passo 2, caso algum *template* aponte para um arquivo de mapeamento de classe que esteja sintaticamente inconsistente, adiciona mensagem de WARNING no *log* e desabilita o referido *template* para geração de código.
5. **Erro na geração de código:** No passo 9, caso seja(m) detectado(s) erro(s) no processo de geração, a aplicação insere a(s) mensagem(s) de erro no *log*, apresenta uma mensagem de erro ao usuário e aborta a operação.

Quadro 11 – Detalhamento do caso de uso principal da aplicação

Os quadros 12, 13, 14 e 15 mostram um resumo dos demais casos de uso da aplicação.

UC2 – Visualiza *Template*

Objetivo: Permite ao usuário visualizar um *template* selecionado na interface da aplicação.

Pré-condições:
- Deve haver ao menos um *template* definido.

Pós-condição:
- O *template* selecionado será exibido na tela de visualização e alteração.

Quadro 12 – Detalhamento do caso de uso Visualiza *Template*

UC3 – Salva *Template*

Objetivo: Permite ao usuário salvar um *template* selecionado na interface da aplicação.

Pré-condições:
- Deve haver ao menos um *template* definido.

Pós-condição:
- O *template* será salvo em disco com as alterações realizadas pelo usuário.

Quadro 13 – Detalhamento do caso de uso Salva *Template*

UC4 – Analisa *Template*

Objetivo: Permite ao usuário efetuar as análises léxica e sintática sobre um *template* selecionado na interface da aplicação.

Pré-condições:
- Deve haver ao menos um *template* definido.

Pós-condição:
- O *template* é analisado e a aplicação apresenta uma mensagem com o resultado do processo.

Quadro 14 – Detalhamento do caso de uso Analisa *Template*

UC5 – Edita *Template*

Objetivo: Permite ao usuário editar um *template* selecionado na interface da aplicação.

Pré-condições:
- Deve haver ao menos um *template* definido.

Pós-condição:
- O *template* será exibido para visualização e alteração.

Quadro 15 – Detalhamento do caso de uso Edita *Template*

3.4.2 Diagrama de classes

O diagrama de classes traz uma representação lógica de como as classes da aplicação estão associadas. A Figura 8 mostra o diagrama de classes da fase de análise construído para a ferramenta proposta, com as principais classes inerentes à lógica do problema. Pode-se verificar que alguns conectores foram suprimidos, e tiveram suas extremidades identificadas por objetos retangulares de bordas arredondadas. Este expediente foi necessário porque estes conectores estavam se sobrepondo visualmente aos demais, prejudicando a legibilidade do diagrama. O diagrama de classes da fase de projeto, por conter muitas entidades consideradas de pouca relevância para entendimento do problema, é apresentado no apêndice A, que traz também um breve relato da funcionalidade das classes apresentadas.

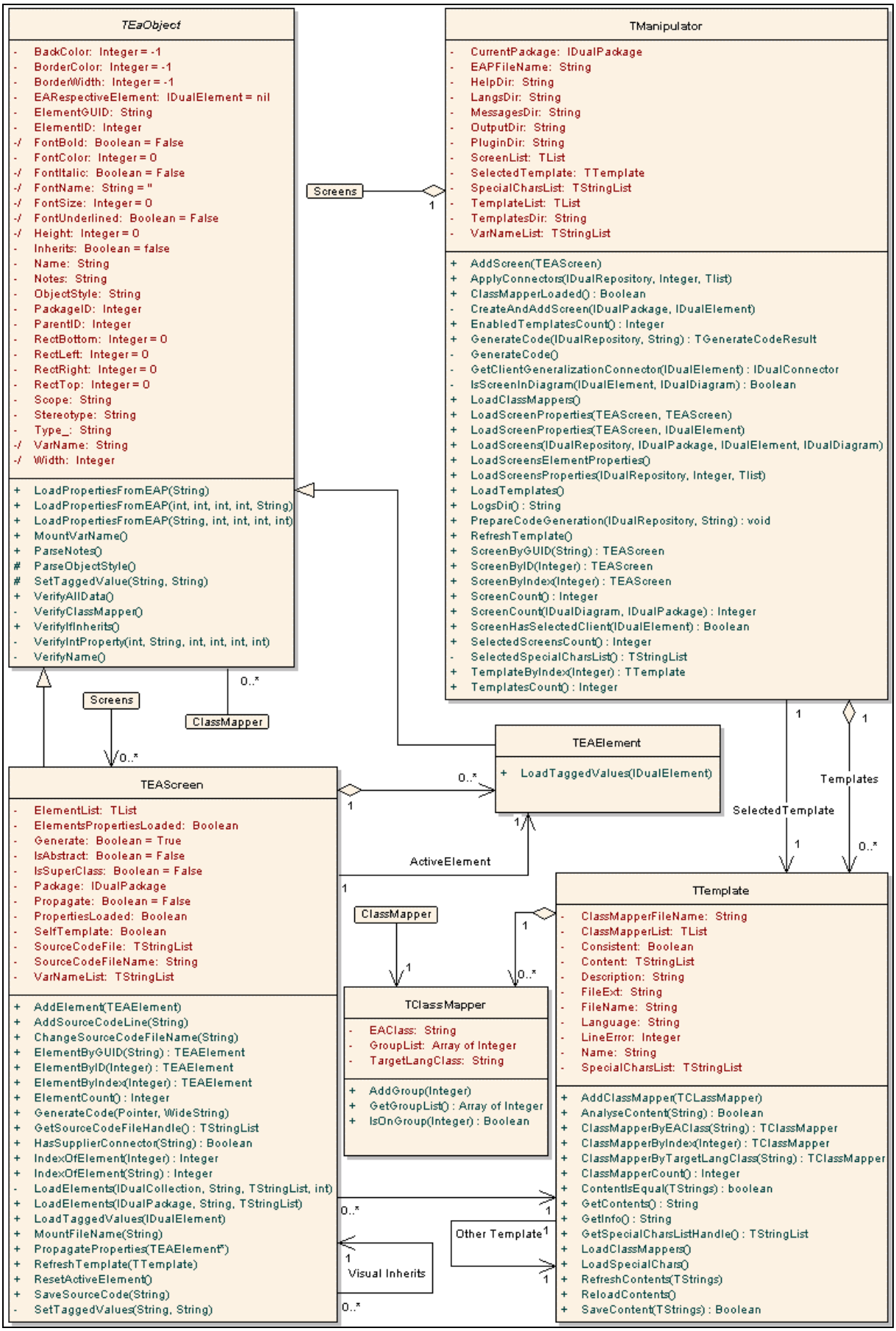


Figura 8 – Diagrama de classes de análise

A seguir é apresentado um detalhamento acerca do funcionamento das classes apresentadas no diagrama, bem como seus relacionamentos:

a) `TManipulator`: é um contêiner para as telas e para os *templates* da aplicação. Através desta classe são efetuadas todas as operações inerentes à lógica do problema, como carga e criação das telas, elementos visuais, *templates* e mapeamentos de classe. Desenvolvida de acordo com o padrão *Singleton*[4], `TManipulator` é uma classe que terá apenas uma instância em toda a aplicação;

b) `TEAObject`: classe abstrata que representa os objetos visuais que serão usados na construção do diagramas de telas do EA. Pode ser uma tela (`Screen`) ou um elemento visual (`GUIElement`);

c) `TEAElement`: herdeira de `TEAObject`, esta classe representa os componentes visuais do diagrama cujo atributo `Type_` contenha o valor `GUIElement`;

d) `TEAScreen`: também herdeira de `TEAObject`, representa os componentes visuais do diagrama cujo atributo `Type_` contenha o valor `Screen`. `TEAScreen` é também um contêiner para objetos da classe `TEAElement`, e é responsável pela criação, manipulação e destruição destes objetos;

e) `TClassMapper`: responsável por fazer o mapeamento dos elementos do EA para a respectiva linguagem alvo, `TClassMapper` define, para um determinado estereótipo de um elemento visual do EA, qual a respectiva classe na linguagem alvo. Assim, todos os elementos que forem inseridos em um diagrama de telas devem ter o campo *stereotype* definido, e para esta definição deve haver uma correspondência para a linguagem alvo no arquivo de mapeamento de classes. Cada instância da classe `TEAObject` é possuidora de um objeto de `TClassMapper`;

f) `TTemplate`: esta classe é uma representação lógica do arquivo de *template* usado na geração de código mantendo, em suas instâncias, uma lista de objetos de `TClassMapper`. Em suma, cada *template* é responsável pela definição de como será o código gerado na linguagem para a qual foi construído, bem como manter a correspondência, através do mapeamento de classes, entre os *stereotypes* dos elementos e as respectivas classes desta linguagem. Cada tela do sistema (`TEAScreen`) é possuidora de um objeto de `TTemplate`.

[4] Segundo Horstmann (2007, p. 398), uma classe *Singleton* tem um único objeto instanciado, compartilhado por todos os clientes que necessitem dele, e esta unicidade deve ser garantida pela implementação.

3.4.3 Diagrama de componentes

O diagrama de componentes mostra as várias partes em que uma aplicação pode estar organizada e dividida, bem como a dependência entre cada uma destas partes. Neste contexto, as partes podem representar módulos, arquivos de código fonte, arquivos executáveis, pacotes, entre outros artefatos de software. A Figura 9 mostra o diagrama de componentes criado para a ferramenta proposta.

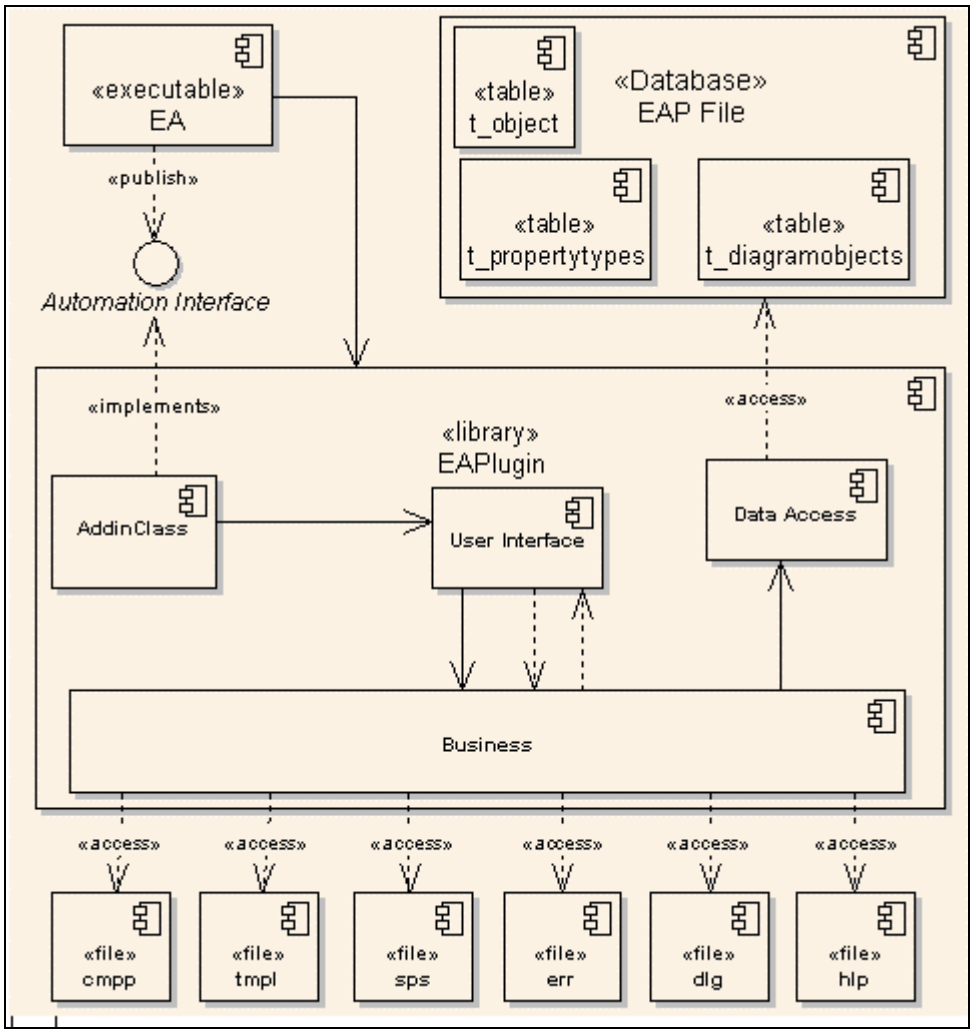


Figura 9 – Diagrama de componentes

A seguir é apresentado um detalhamento sobre os componentes presentes no diagrama exibido:

a) EA: este componente representa o arquivo executável do Enterprise Architect, fornecido juntamente com o pacote de instalação da ferramenta. É a partir da execução deste arquivo que a classe que implementa a interface de automação é instanciada;

b) `EAPlugin`: é o resultado da compilação da ferramenta proposta, na forma de um arquivo com extensão DLL.

c) `Automation Interface`: representa a interface de automação do EA, que rege as formas de acesso aos modelos internos da ferramenta. No conceito de I-CASE, a AI representa a camada de gestão de objetos, realizando a comunicação entre as ferramentas do ambiente integrado;

d) `AddinClass`: este componente representa a classe que implementa a interface de automação do EA. É responsável por responder aos eventos de interação do usuário com o EA No projeto este componente está implementado na classe `TAddin`;

e) `User Interface`: representa as telas do *plugin*, a partir das quais o usuário poderá interagir com a aplicação. Faz parte deste componente as classes `TBasicForm`, `TFrmCodeGenerate` e `TFrmTemplateView`;

f) `Business`: representa as classes inerentes à lógica do problema, provendo a comunicação com as camadas de apresentação e de acesso a dados. É composto pelas classes `TGenericClass`, `TEaObject`, `TUserMessage`, `TEAElement`, `TEAScreen`, `TClassMapper`, `TTemplate`, `TManipulator` e `TNamesControl`;

g) `Data Access`: este componente representa a camada de acesso a dados do *plugin*, que realiza comunicação diretamente com o arquivo EAP. É formado pela classe `TDataAccess`;

h) `EAP File`: arquivo em que são persistidos os dados do Entreprise Architect. No conceito de I-CASE, este arquivo representa o repositório compartilhado;

i) `t_object`: tabela do arquivo EAP de onde são extraídos valores não acessíveis através da AI;

j) `t_diagramobjects`: também representa uma tabela do arquivo EAP de onde são extraídos valores não acessíveis através da AI;

k) `t_propertytypes`: tabela do arquivo EAP usada para criação dos *Tagged Values* específicos do *plugin*;

l) `cmpp`: arquivo de mapeamento de classes, cujo funcionamento é detalhado na seção 3.5.2.3;

m) `tmpl`: corresponde aos arquivos de *template*, usados na geração de código, cujo funcionamento é detalhado na seção 3.5.2.3. A linguagem de programação do *template* foi especificada de acordo com a notação BNF, e é apresentada no

apêndice E;

n) `sps`: representa os arquivos de caracteres especiais usados pelo *plugin* para criação de nomes de variáveis.
o) `err`: representa o arquivo de mensagens de erro apresentadas ao usuário;
p) `dlg`: arquivo de mensagens de diálogo com o usuário;
q) `hlp`: arquivo de ajuda.

3.5 IMPLEMENTAÇÃO

Nesta seção são apresentadas as técnicas e ferramentas utilizadas no desenvolvimento da ferramenta proposta. É também demonstrada a operacionalidade da aplicação e, em seguida, os resultados dos trabalhos.

3.5.1 Técnicas e ferramentas utilizadas

A ferramenta foi implementada na linguagem de programação Delphi, usando o ambiente Delphi 7. O formato do código gerado segue as diretrizes ditadas por um *template* com linguagem de programação própria, especificada com a notação BNF e cujos analisadores léxico e sintático foram gerados pelo Gerador de Analisadores Léxicos e Sintáticos (GALS). A implementação segue os preceitos da arquitetura de três camadas, que prevê a separação das lógicas de acesso a dados, de apresentação e de negócio.

3.5.2 Etapas da Implementação

Conforme citado no item (e) do capítulo 3, o desenvolvimento da aplicação foi dividido em algumas etapas, conforme sugere o processo de desenvolvimento de um gerador de código apresentado na seção 2.4. Para esclarecimento da nomenclatura utilizada pela tecnologia COM, a ferramenta proposta, na forma de um *plugin*, é chamada de “servidor COM”, enquanto o aplicativo hospedeiro, o Enterprise Architect, é chamado de “cliente

COM”. Feito este esclarecimento, a seguir são apresentadas estas etapas do desenvolvimento da ferramenta proposta.

3.5.2.1 Comunicação com o EA

Para realizar a comunicação com a interface de automação do EA foi usado um recurso provido pela tecnologia COM chamado ActiveX. Para Cantù (2003, p. 387), um controle ActiveX é “um documento composto que é implementado como uma DLL de servidor dentro de processo que suporta Automação, edição visual e ativação de dentro para fora”. Suportar Automação significa que é possível acessar os objetos e métodos deste controle. Suportar edição visual denota a possibilidade de “visualizar” o controle ActiveX e acessar suas funções a partir de uma janela do aplicativo contêiner. Por fim, ativação de dentro para fora significa que o controle ActiveX surge de dentro do aplicativo contêiner, e não existe sem este.

Uma DLL desenvolvida segundo as diretrizes dos controles ActiveX tem sua implementação, em Delphi, muito parecida com uma DLL comum. A diferença está na obrigatoriedade de exportar algumas funções, por questões de compatibilidade com a tecnologia COM. O Quadro 16 mostra o código do esqueleto de um controle ActiveX.

```
library GUICGAddin;
uses
  ComServ,
  GUICGAddin_TLB in 'GUICGAddin_TLB.pas',
  EA_TLB in 'EA_TLB.pas',
  Addin in 'Addin.pas',
  ufrmCodeGenerate in '..\Form\uFrmCodeGenerate.pas',
  uBasicForm in '..\Form\uBasicForm.pas',
  uFrmTemplateView in '..\Form\uFrmTemplateView.pas';
exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;
begin
end.
```

Quadro 16 – Implementação do esqueleto de um controle ActiveX

As quatro primeiras declarações da cláusula *uses* representam, respectivamente, a unidade que implementa o objeto `ComServer` (usado no código exibido pelo Quadro 18), a unidade que armazena a biblioteca de tipos do *plugin*, a unidade em que é publicada a interface de automação do EA e a unidade que implementa a classe `TAddin` (implementação também exibida no Quadro 18). Em seguida, na cláusula *exports*, há a declaração das funções

que devem ser exportadas pela DLL, para compatibilidade com a tecnologia COM, cujas funcionalidades são listadas abaixo:

a) `DllGetClassObject`: usada para retornar a fábrica de classes de uma determinada classe enviada como parâmetro;

b) `DllCanUnloadNow`: verifica se a DLL do servidor pode ser descarregada da memória;

c) `DllRegisterServer`: registra a DLL de um servidor COM no registro do Windows. Quando um servidor COM for registrado no SO, será criada uma chave no registro, conforme exibido pelas figuras 10 à 14;

d) `DllUnregisterServer`: esta função simplesmente desfaz tudo o que a função anterior fez.

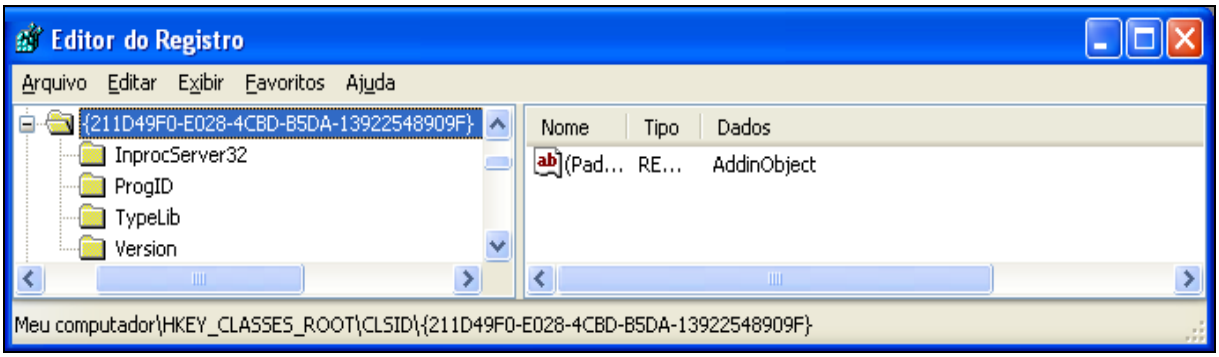


Figura 10 – Registro do Windows (Chave da classe TAddin)

A Figura 10 exibe a chave criada no registro do Windows para identificação do controle ActiveX registrado. Como pode ser visto, o nome da chave corresponde ao GUID atribuído à classe `TAddin`, cujo valor é declarado na constante `CLASS_Addin`, conforme ilustrado no Quadro 17.

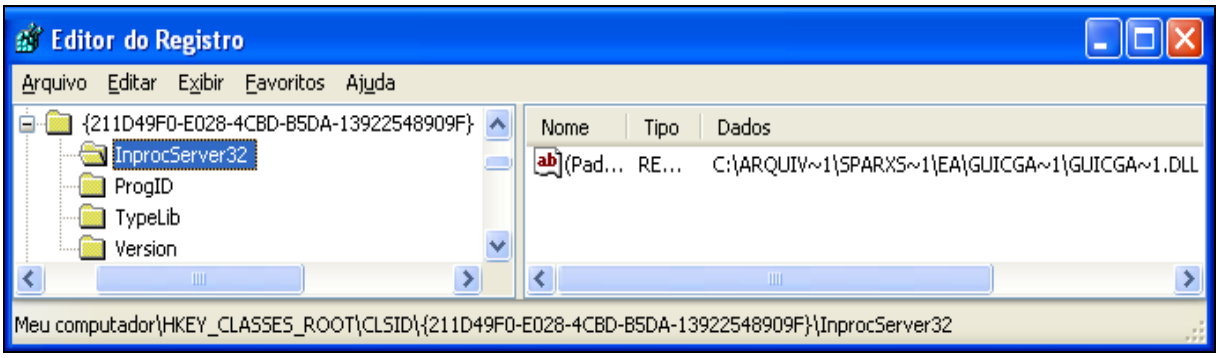


Figura 11 – Registro do Windows (Tipo de servidor)

A chave `InprocServer32`, exibida na Figura 11, indica o caminho absoluto da DLL registrada. O nome da chave indica que a referida DLL é um servidor dentro de processo para rodar em processadores de 32 bits.

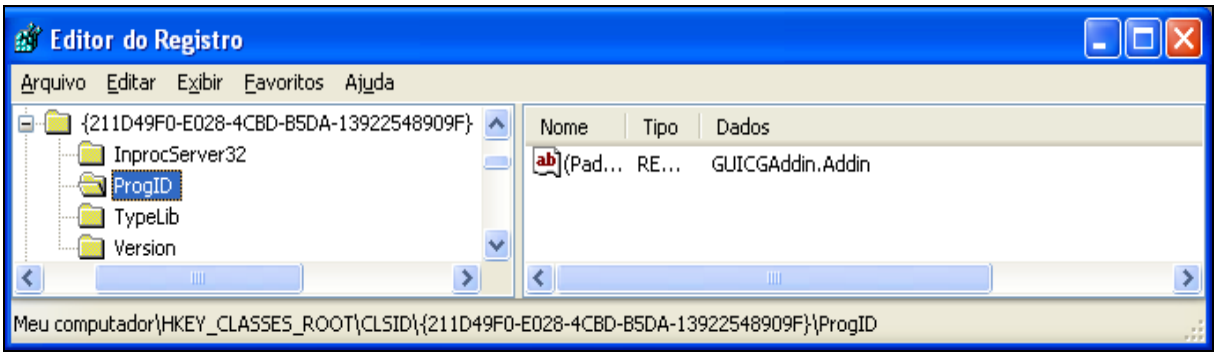


Figura 12 – Registro do Windows (Identificador do controle ActiveX)

A Figura 12 mostra a chave `ProgID`, que é um identificador para a classe `TAddin`, formado pelo nome da classe (sem o prefixo T) e pelo nome do projeto da DLL, que é a primeira declaração mostrada no Quadro 16.

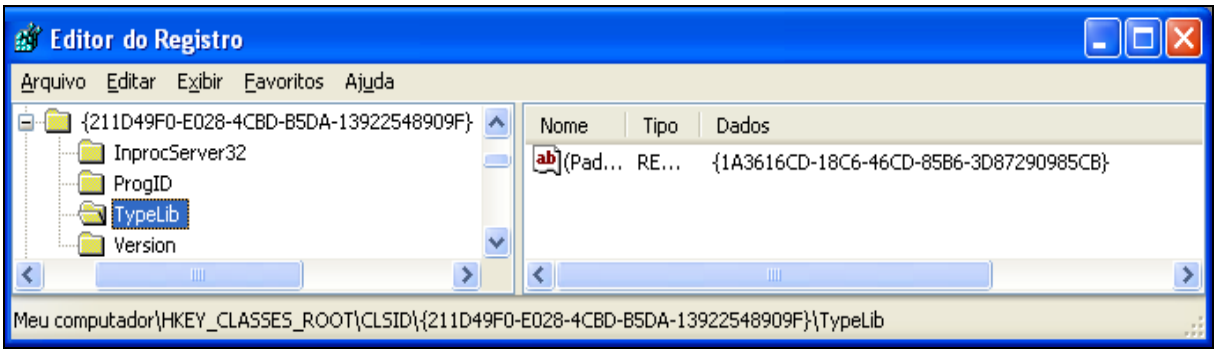


Figura 13 – Registro do Windows (GUID da biblioteca de tipos do controle ActiveX)

A Figura 13 destaca a chave `TypeLib`, cujo valor é formado a partir do GUID da biblioteca de tipos do controle ActiveX. Esta biblioteca tem seu código resumidamente apresentado no Quadro 17. Neste código, o referido GUID está assinalado na constante `LIBID_GUICGAddin`.

A chave `Version`, em destaque na Figura 14, traz a informação de uma versão atribuída ao controle ActiveX.

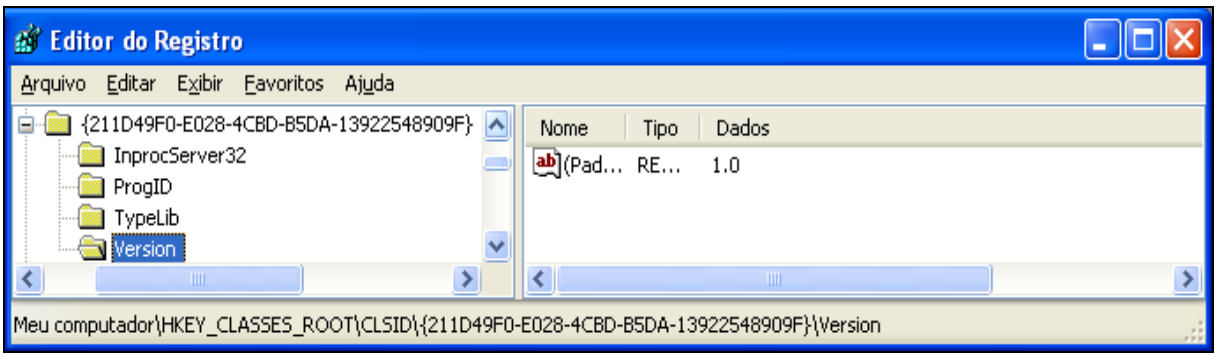


Figura 14 – Registro do Windows (Versão da biblioteca de tipos)

A unidade exibida no Quadro 17 representa a biblioteca de tipos do *plugin*, e é nela que a interface `IAddin` é implementada. Nesta unidade é importante atentar às três constantes do tipo `TGUID` declaradas; elas correspondem, respectivamente, aos identificadores globais da

biblioteca de tipos, da interface `IAddin` e da classe `TAddin`.

```
unit GUICGAddin_TLB;
interface
uses Windows, ActiveX, Classes, Graphics, StdVCL, Variants;
const
  LIBID_GUICGAddin:TGUID = '{1A3616CD-18C6-46CD-85B6-3D87290985CB}';
  IID_IAddin: TGUID = '{CA66A2C4-79C1-43F3-9722-DA2837D05DD2}';
  CLASS_Addin: TGUID = '{211D49F0-E028-4CBD-B5DA-13922548909F}';

  IAddin = interface(IDispatch)
    ['{CA66A2C4-79C1-43F3-9722-DA2837D05DD2}']
    function EA_Connect(const Repository: IDispatch): WideString; safecall;
    procedure EA_MenuClick(const Repository: IDispatch; const Location:
       WideString; const MenuName: WideString; const ItemName: WideString);
       safecall;
  end;
end.
```

Quadro 17 – Biblioteca de tipos do *plugin*

Segundo os preceitos do ActiveX , o primeiro passo para o desenvolvimento de um *plugin* para o Enterprise Architect é criar uma classe que implemente a sua interface de automação. Na ferramenta proposta, esta classe foi chamada de `TAddin`, e um resumo do código de sua implementação pode ser visualizado no Quadro 18.

```
TAddin = class(TAutoObject, IAddin)
protected
  function EA_Connect(const Repository: IDispatch): WideString; safecall;
  procedure EA_MenuClick(const Repository: IDispatch;
     const Location, MenuName, ItemName: WideString); safecall;
end;
implementation

function TAddin.EA_Connect(const Repository: IDispatch): WideString;
begin
  UserMessage := TUserMessage.SingletonInstance;
  UserMessage.Initialize(TAddinUtils.GetEADirectory + CMessagesPath);
end;
procedure TAddin.EA_MenuClick(const Repository: IDispatch;
    const Location, MenuName, ItemName: WideString);
var
  f : TfrmCodeGenerate;
begin
  if ItemName = Menu_GenerateCode then
  begin
    f := TfrmCodeGenerate.Create(nil,(Repository as IDualRepository),
        Element,Diagram,UserMessage);
    if f.ShowModal = mrOk  then ;
    f.Free;
  end
end;
initialization
  TAutoObjectFactory.Create(ComServer, TAddin, Class_Addin, ciMultiInstance);
end.
```

Quadro 18 – Implementação da classe TAddin

Pode-se observar, na declaração da classe `TAddin`, que ela é descendente da classe `TAutoObject`, que atende aos requisitos exigidos pela tecnologia COM, e também implementa a interface `IAddin`, representada resumidamente no Quadro 17, que impõe a implementação dos métodos para comunicação com o EA.

O método `EA_Connect` é disparado no momento na inicialização do EA, recebendo como parâmetro o seu repositório[5] de dados. Todos os objetos do EA passados como parâmetros nos métodos da AI são do tipo `IDispatch` (padrão COM), e para que sejam interpretados corretamente pela aplicação é necessário que seja feita uma conversão de tipo, realizada pela instrução `as`, visível no Quadro 18. Neste método é criada e inicializada a instância única do objeto de `TUserMessage`, que será usado pelos demais objetos do *plugin*.

O método `EA_MenuClick` é executado sempre que o usuário percorre os itens de menu do EA, e recebe também um ponteiro para o repositório de dados. Além do repositório, este método recebe um parâmetro `Location`, que indica em que local do EA o usuário clicou para acessar o menu, podendo ser sobre um diagrama, sobre a `TreeView` ou sobre o menu principal. Os dois últimos parâmetros, `ItemName` e `MenuName` são, respectivamente, o nome do item de menu selecionado e o nome do item de menu de nível imediatamente superior ao selecionado. A partir deste evento, caso o usuário tenha clicado sobre o item de menu referente a geração de código, a interface principal do *plugin* é instanciada, dando início ao próximo passo da execução, cuja implementação é detalhada na seção seguinte.

Ao final desta unidade (Quadro 18) aparece a seção `initialization`, que corresponde ao código que é realizado no início de sua execução. Esta seção contém a declaração mais importante para a compreensão do funcionamento da tecnologia COM. Trata-se do método `TAutoObjectFactory.Create`, construtor que fornecerá para o EA a fábrica de classes para `TAddin`. A seguir são listadas as funcionalidades dos parâmetros enviados ao construtor:

a) `ComServer`: objeto global da classe `TComServer`. É um gerenciador de fábrica de classes que contém métodos específicos para a procura de classes no registro do Windows e para a contagem de objetos alocados;
b) `TAddin`: classe para a qual se deseja retornar a fábrica de classes. No caso do código exibido pelo Quadro 18, foi enviado o nome da classe criada para o *plugin* e, assim, o retorno do método será um ponteiro para um objeto desta classe;
c) `Class_Addin`: constante que representa o GUID da classe citada anteriormente, e

que é usado como parâmetro pelos métodos de procura de classes;

d) `ciMultiInstance`: modelo de instanciação do servidor COM, e, neste caso, indica que quando vários clientes solicitam o objeto COM, o sistema cria várias instâncias do servidor.

3.5.2.2 Carga das informações fornecidas pela AI

A obtenção das informações oferecidas pela AI tem início no construtor da tela principal do *plugin*, onde é montada uma estrutura básica para o decorrer da execução. A implementação do referido construtor é apresentada no Quadro 19.

```
constructor TfrmCodeGenerate.Create(AOwner: TComponent;
  ARepository: IDualRepository; ASelectedScreen: IDualElement;
  ASelectedDiagram: IDualDiagram; ASelectedPackage: IDualPackage;
  AUserMessage : TUserMessage);
begin
  inherited Create(AOwner);
  Repository      := ARepository;
  SelectedScreen  := ASelectedScreen;
  SelectedDiagram := ASelectedDiagram;
  SelectedPackage := ASelectedPackage;
  UserMessage := AUserMessage;
  UserMessage.SetMemo(mmStatus);

  EADir           := ExtractFilePath(Application.ExeName);
  TemplatesDir    := EADir + CTempatesPath;
  MessagesDir     := EADir + CMessagesPath;
  HelpDir         := EADir + CHelpPath;
  PluginDir       := EADir + CPluginPath;
end;
```

Quadro 19 – Implementação do construtor da interface da aplicação

No construtor são atribuídos valores referentes às variáveis `ARepository`, `ASelectedScreen`, `ASelectedPackage`, `ASelectedDiagram` e `AUserMessage`, que representam, respectivamente, o repositório do EA, a tela selecionada, o pacote selecionado, o diagrama selecionado e o objeto manipulador de mensagens. O construtor assinala ainda variáveis da estrutura de diretórios do *plugin*, cujo funcionamento é abordado na seção 3.5.3. Logo após a execução do construtor da interface, automaticamente é acionado o método `FormCreate` da tela principal do *plugin*, cuja implementação é exibida no Quadro 20.

[5] Segundo Sparx Systems (2007), o repositório é o contêiner principal do EA, e dá acesso a todas as demais estruturas da interface de automação.

```
procedure TfrmCodeGenerate.FormCreate(Sender: TObject);
begin
  UserMessage.InitializeCounters;
  UserMessage.AddLogMessage(UserMessage.GetDialogMsg(39),[Self.ClassName]);
  UserMessage.SaveLog;
  HelpFile := ExtractFilePath(Application.ExeName) + CPartialHelpFileName;
  Manipulator := TManipulator.SingletonInstance;
    Manipulator.Initialize(Repository.ConnectionString, HelpDir, MessagesDir,
    TemplatesDir,PluginDir,SelectedPackage, UserMessage);
end;
```

Quadro 20 – Implementação do método `FormCreate`

Neste método são inicializados os contadores de erros e *warnings* do gerenciador de mensagens. Também é instanciado e inicializado o objeto único da classe `TManipulator`. Ainda antes de a interface ser apresentada ao usuário é carregada uma lista com os *templates*, e uma lista com a(s) tela(s) selecionada(s), de acordo com o elemento do EA selecionado. Por questão de economia de processamento, neste momento são carregadas apenas informações das telas que sejam necessárias para o seu reconhecimento. Foi adotado este procedimento tendo em vista a possibilidade de o usuário executar o *plugin* e não iniciar o processo de geração de código.

Neste momento a interface principal do *plugin* é exibida ao usuário e, caso seja acionado o botão de geração de código é iniciada a segunda parte do processo de carga das informações provindas da AI do EA. Esta etapa compreende a carga das demais propriedades das telas, carga dos elementos visuais e de suas propriedades e a atualização do *template* de cada tela. A carga das informações das telas deve, obrigatoriamente, seguir uma ordem que garanta a propagação de características de herança entre elas. Um exemplo desta ordem é ilustrado pela Figura 15.

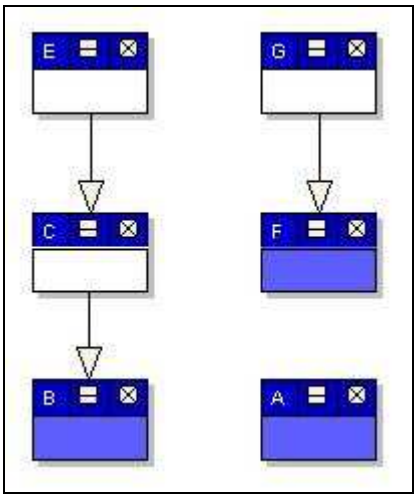


Figura 15 – Ordem de carga das informações das telas

Primeiramente são carregadas as propriedades das telas azuis que, como pode ser visualizado na figura, não herdam características de outra tela. Em seguida, para cada tela

carregada anteriormente, o método é chamado recursivamente, realizando uma busca em largura de acordo com a seqüência dos conectores de herança. Ao término do assinalamento das propriedades de cada dela e de cada elemento é feita uma verificação de dados, para garantir que os valores atribuídos aos atributos dos objetos estejam de acordo com as definições do *plugin*.

3.5.2.3 Modelo de saída de dados

O formato de saída do código a ser gerado segue as diretrizes de uma linguagem definida para este fim, cujas regras gramaticais e semânticas são apresentadas nos apêndices E e F, respectivamente, e que permite a criação de *templates* de acordo com as necessidades do usuário. O conteúdo destes *templates* é armazenado em um arquivo no formato de texto, com extensão *tmpl*.

Antes de iniciar as considerações sobre a linguagem em si, cabe esclarecer alguns pontos importantes do processo. O primeiro ponto é que cada *template* dará origem a um, e somente um, arquivo de código fonte. O segundo ponto é que cada tela a ser gerada manterá em sua estrutura um objeto de *template*, o qual definirá o formato de saída do código referente a esta tela. Portanto, as análises léxica, sintática e semântica são realizadas sobre o *template* a cada tela gerada. O terceiro ponto trata da possibilidade de um *template* fazer referência a outro, e este outro a outro, e assim sucessivamente. Isso levará a geração de tantos arquivos de código fonte quantos *templates* presentes nesta sucessão. Para finalizar, é importante notar que a linguagem definida para a construção dos *templates* não é sensível ao caso. Feitos estes esclarecimentos, é possível iniciar o detalhamento do modelo de saída de dados.

O Quadro 21 traz um *template* definido para a geração do arquivo .DFM de um formulário em Delphi. Foi escolhido este *template* por abordar a maioria dos recursos disponíveis na ferramenta proposta. O *template* definido para geração do código em Java é apresentado no apêndice L.

```
[TEMPLATE]

TEMPLATEFILEEXT = '.dfm';

NAME = 'Delphi_DFM';

LANGUAGE = 'Delphi';

CLASSMAPPERFILE ='DelphiClassMapper.cmpp';

SPECIALCHARSFILE = 'Delphi.sps';

OTHERTEMPLATEFILE = 'C:\Arquivos de programas\Sparx
Systems\EA\GUICGAddin\Templates\Delphi_PasFile.tmpl';

TEMPLATEDESC = 'Este template faz a geração do fonte para o arquivo .DFM';

Start

if (@FONTBOLD_ = true) then {
  &Style := '[fsBold';
}

if (@FONTITALIC_ = true) then {
  if (&Style = '') then {
    &Style := '[fsItalic';
  } else {
    &Style := &Style ',fsItalic';
  }
}

if (@FONTUNDERLINED_ = true) then {
  if (&Style = '') then {
    &Style := '[fsUnderline';
  } else {
    &Style := &Style ',fsUnderline';
  }
}

if (&Style = '') then {
  &Style := '[';
}

&Style := &Style ']';

wrt('object ' @VARNAME_ ': T' @VARNAME_ ); <NL>
wrt('  Left = ' @RECTLEFT_ ); <NL>
wrt('  Top = '      @RECTTOP_ ); <NL>
wrt('  Width = '    @WIDTH_ ); <NL>
wrt('  Height = '   @HEIGHT_ ); <NL>
wrt('  Caption = '  #39 @NAME_ #39 ); <NL>
wrt('  Color = '    @BACKCOLOR_ ); <NL>
wrt('  Font.Charset = DEFAULT_CHARSET' ); <NL>
wrt('  Font.Color = ' @FONTCOLOR_ ); <NL>
wrt('  Font.Height = -11' ); <NL>
```

```
wrt('  Font.Name = ' #39 @FONTNAME_ #39 ); <NL>
wrt('  Font.Style = ' &Style ); <NL>
wrt('  OldCreateOrder = False ' ); <NL>
wrt('  PixelsPerInch = 96 ' ); <NL>
wrt('  TextHeight = 13' ); <NL>

LOOP
  &Style := '';

  if (@FONTBOLD = true) then {
    &Style := '[fsBold';
  }

  if (@FONTITALIC = true) then {
    if (&Style = '') then {
      &Style := '[fsItalic';
    } else {
      &Style := &Style ',fsItalic';
    }
  }

  if (@FONTUNDERLINED = true) then {
    if (&Style = '') then {
      &Style := '[fsUnderline';
    } else {
      &Style := &Style ',fsUnderline';
    }
  }

  if (&Style = '') then {
    &Style := '[';
  }

  &Style := &Style ']';

  wrt('  object '       @VARNAME ':' @TARGETLANGCLASS ); <NL>
  wrt('    Left = '     @RECTLEFT ); <NL>
  wrt('    Top = '      @RECTTOP ); <NL>
  wrt('    Width = '    @WIDTH ); <NL>
  wrt('    Height = '   @HEIGHT ); <NL>

  if (@GROUP = 1) then {
    wrt('    Caption = ' #39 @NAME #39 ); <NL>
  }

  if (@GROUP = 2) then {
    wrt('    Color = ' @BACKCOLOR ); <NL>
  }

  wrt('    Font.Charset = DEFAULT_CHARSET' ); <NL>
  wrt('    Font.Color = ' @FONTCOLOR ); <NL>
  wrt('    Font.Height = -11' ); <NL>
  wrt('    Font.Name = ' #39 @FONTNAME #39 ); <NL>
  wrt('    Font.Style = ' &Style ); <NL>

  if (@GROUP = 3) then {
    wrt('    ParentColor = False' ); <NL>
```

```
  }

  wrt('    ParentFont = False' ); <NL>

  if (@GROUP = 4) then {
    wrt('    TabOrder = ' &TabOrder ); <NL>
    &TabOrder := &TabOrder 1;
  }

  if (@GROUP = 102) then {
    wrt('    Text = ' #39 @NAME #39 ); <NL>
  }

  wrt('  end' ); <NL>

CLOSELOOP

wrt('end' );
@SOURCECODEFILENAME_ := 'u' @VARNAME_ @DEFAULTSCFILEEXT_ ;

[/TEMPLATE]
```

Quadro 21 – *Template* para geração de um arquivo .DFM

A linguagem do *template* prediz que todos os comandos devem estar delimitados pelas *tags* `[TEMPLATE]` e `[/TEMPLATE]`. A referida linguagem define ainda que todas as instruções anteriores à palavra reservada (e obrigatória) `start` fazem parte do cabeçalho do *template*, e as posteriores compõe o corpo deste. O processo de execução do *template* é dividido em duas etapas distintas: as declarações fora da área de repetição e as declarações dentro da área de repetição. As declarações dentro da área de repetição são delimitadas pelas *tags* `LOOP` e `CLOSELOOP`, e indica ao analisador que todas estas declarações devem ser executadas repetidamente, para cada elemento visual da tela. A seguir são apresentadas as variáveis que são disponibilizadas ao usuário para uso no *template*.

Há um conjunto de variáveis que são comuns a elementos visuais e telas, variáveis estas listadas no Quadro 22.

Variável	Tipo	Descrição
@BACKCOLOR	*integer*	Cor de fundo do elemento.
@BORDERCOLOR	*integer*	Cor da borda do elemento.
@BORDERWIDTH	*integer*	Largura da borda do elemento.
@FONTBOLD	*boolean*	Se a fonte do elemento está em negrito.
@FONTITALIC	*boolean*	Se a fonte do elemento está em itálico.
@FONTCOLOR	*integer*	Cor da fonte do elemento.
@FONTNAME	*string*	Nome da fonte do elemento.
@FONTSIZE	*integer*	Tamanho da fonte do elemento.
@FONTUNDERLINED	*boolean*	Se a fonte do elemento está sublinhada.
@NAME	*string*	Nome do elemento.
@RECTBOTTOM	*integer*	Ponto de início da horizontal inferior do elemento, em relação ao topo do diagrama.
@RECTLEFT	*integer*	Ponto de início da vertical esquerda do elemento, em relação à extrema esquerda do diagrama.
@RECTRIGHT	*integer*	Ponto de início da vertical direita do elemento, em relação à extrema esquerda do diagrama.
@RECTTOP	*integer*	Ponto de início da horizontal superior do elemento, em relação ao topo do diagrama.
@SCOPE	*string*	Escopo do elemento.
@STEREOTYPE	*string*	Estereótipo do elemento.
@TARGETLANGCLASS	*string*	Classe do elemento na linguagem alvo. Definido no arquivo de mapeamento de classe.
@TYPE	*string*	Tipo do elemento.
@WIDTH	*integer*	Largura do elemento.
@HEIGHT	*integer*	Altura do elemento.
@VARNAME	*string*	Nome de variável atribuída ao elemento.
@GROUP	*integer*	Grupo do elemento. Definido no arquivo de mapeamento de classe.
@GUID	*string*	Identificador global do elemento.
@ID	*integer*	Identificador numérico do elemento.
@INHERITS	*boolean*	Se o elemento herda características da tela, ou se a tela herda características de outra tela.

Quadro 22 – Variáveis comuns a elementos visuais e telas

Por serem comuns a elementos e telas, as variáveis listadas no Quadro 22 devem possuir um diferenciador, para que o analisador semântico possa saber de onde buscar atributo. Assim, as variáveis de “tela” são sucedidas pelo caractere *underline*, enquanto as variáveis da “tela de herança” são sucedidas pelo caractere 0. De acordo com esta definição, para se fazer referência à cor da fonte da tela, por exemplo, deve-se usar a variável `@FONTCOLOR_`, enquanto que para fazer referência à mesma variável da tela de herança deve-se usar a variável `@FONTCOLOR#`. Quando se desejar referenciar variáveis de elementos, basta escrevê-las conforme mostra o Quadro 22, sem a necessidade de sufixos. Entretanto, deve-se atentar ao fato de que os elementos visuais somente existem dentro de uma estrutura de repetição, portanto, somente nesta situação é válido fazer referência às variáveis de elementos visuais. Não obstante esta regra, para evitar erros de compilação, sempre que for usada uma variável de elemento visual fora dos delimitadores de repetição, automaticamente o analisador

semântico irá considerar esta variável como uma variável de tela. A mesma regra se aplica quando for referenciada uma variável de tela de herança: caso a tela que está sendo processada não possua conectores de herança, o analisador irá retornar o valor da variável da própria tela. Para facilitar a compreensão deste processo foi construído um exemplo a partir de um diagrama de telas, mostrado pela Figura 16, e de uma parte de um *template*, exibido no Quadro 23.

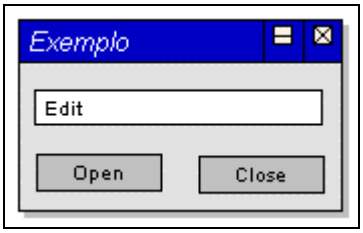


Figura 16 – Exemplo de tela

```
wrt('Nome: ' @NAME); <NL>
wrt('Nome: ' @NAME#); <NL>
LOOP
  wrt('Nome: ' @NAME); <NL>
  wrt('Nome: ' @NAME_); <NL>
CLOSELOOP
```

Quadro 23 – Exemplo de *template*

Para o trecho de código mostrado no Quadro 23, quando for solicitada a geração de código da tela “Exemplo”, a saída será conforme apresentado no Quadro 24. Os textos destes dois quadros são propositalmente coloridos para criar uma relação entre a origem e o resultado.

```
Nome: Exemplo
Nome: Exemplo

Nome: Edit
Nome: Exemplo

Nome: Open
Nome: Exemplo

Nome: Close
Nome: Exemplo
```

Quadro 24 – Resultado da geração

No *template* mostrado pelo Quadro 23 pode ser verificado o uso do comando `wrt`, que instrui ao analisador semântico a gerar como saída o conteúdo delimitado pelos parênteses, composto por uma *string* e pelo conteúdo da variável `@NAME`. Na primeira linha de instruções é solicitada a geração de um texto, seguido do conteúdo da variável `@NAME`. Apesar de esta ser uma variável de elementos visuais (por não ter sufixo “#” nem “_”), como estes elementos somente são “enxergados” pelo analisador dentro da estrutura de repetição, o valor trazido

(nome do elemento) é referente à tela em questão. Na segunda instrução é solicitada ao analisador a geração do mesmo texto, seguido do conteúdo da variável `@NAME#`. Analogamente à primeira situação, como a tela em questão não herda características de outra tela, o valor trazido também é referente à própria tela. A terceira instrução indica o início de um bloco de repetição, portanto, todas as instruções encontradas pelo analisador, até encontrar a *tag* `CLOSELOOP`, serão repetidas para todos os elementos visuais da tela. A quarta instrução diz ao interpretador exatamente a mesma coisa que a primeira, todavia, como se encontra dentro dos delimitadores de repetição, o interpretador "entende" que se está fazendo referência a um elemento visual, assim, é o nome deste que será gerado como saída. A penúltima instrução solicita ao analisador que imprima um texto, seguido da variável `@NAME_`. Aqui entra uma característica importante da semântica do *template*: com exceção das variáveis de elementos visuais (sem sufixos "#" e "_"), todas as demais possuem os mesmos valores dentro e fora dos blocos de repetição. Por esta definição, as variáveis de telas, de telas herdadas e do manipulador têm o mesmo valor independentemente de onde forem usadas. E, por fim, a última instrução informa ao analisador o término do bloco de repetição.

Além destas variáveis comuns a telas e elementos visuais, há um conjunto de variáveis que são exclusivas das telas. O Quadro 25 mostra a relação destas variáveis.

Variável	**Tipo**	**Descrição**
`@SOURCECODEFILENAME_`	*string*	Nome relativo do arquivo em que será gerado o código fonte para a tela.
`@TEMPLATEFILENAME_`	*string*	Nome absoluto do arquivo do *template*.
`@ISABSTRACT_`	*boolean*	Se a tela é abstrata.
`@PROPAGATE_`	*boolean*	Se a tela propaga suas propriedades para seus elementos visuais.
`@CLASSMAPPERFILENAME_`	*string*	Nome relativo do arquivo de mapeamento de classes usado pelo *template* da tela. Definido no arquivo de *template*.
`@TEMPLATEDESCRIPTION_`	*string*	Descrição do *template* da tela. Definido no arquivo de *template*.
`@DEFAULTSCFILEEXT_`	*string*	Extensão que será atribuída ao arquivo de saída do código fonte. Definido no arquivo de *template*.
`@TEMPLATELANGUAGE_`	*string*	Linguagem do *template* da tela. Definido no arquivo de *template*.
`@TEMPLATENAME_`	*string*	Nome do *template* da tela. Definido no arquivo de *template*.

Quadro 25 – Variáveis exclusivas de telas

Também são disponibilizadas para uso no *template* algumas variáveis relacionadas ao manipulador do *plugin*, cujo rol é apresentado no Quadro 26.

Propriedade	Tipo	Descrição
@EAPFILENAME	*string*	Nome absoluto do arquivo EAP da base de dados que se está trabalhando.
@HELPDIR	*string*	Diretório do arquivo do ajuda do *plugin*.
@MSGSDIR	*string*	Diretório dos arquivos de mensagens.
@OUTPUTDIR	*string*	Diretório onde serão gerados os arquivos de código fonte.
@PLUGINDIR	*string*	Diretório de instalação da DLL do *plugin*.
@TEMPLATESDIR	*string*	Diretório dos arquivos de *template*.
@LOGSDIR	*string*	Diretório onde são gerados os *logs* do *plugin*.

Quadro 26 – Variáveis do manipulador

Por derradeiro, o Quadro 27 lista as variáveis do cabeçalho, que são lidas no momento em que o manipulador carrega cada *template* e, segundo definição sintática, não podem ser assinaladas no corpo deste. O uso destas variáveis é demonstrado nas declarações anteriores à instrução *start*, no *template* exibido pelo Quadro 21.

Propriedade	Tipo	Descrição
TEMPLATEDESC	*string*	Descrição do *template*. Usado para que o usuário informe uma descrição que caracterize o *template*, de acordo com a linguagem para qual foi proposto. Será exibido como *hint* na tela principal do *plugin*.
LANGUAGE	*string*	Linguagem para a qual o *template* foi definido. Será exibido como *hint* na tela principal do *plugin*.
TEMPLATEFILEEXT	*string*	Extensão que será atribuída aos arquivos de código fonte que serão gerados.
NAME	*string*	Nome do *template*. Será exibido como *hint* na tela principal do *plugin*.
CLASSMAPPERFILE	*string*	Nome relativo do arquivo de mapeamento de classes.
SPECIALCHARSFILE	*string*	Nome relativo do arquivo de caracteres especiais.
OTHERTEMPLATEFILE	*string*	Caminho absoluto do *template* que será executado no término da geração de código.

Quadro 27 – Variáveis do cabeçalho do *template*

Das variáveis de cabeçalho citadas no Quadro 27, OTHERTEMPLATEFILE merece especial atenção. Há situações em que o código fonte de uma tela depende de mais de um arquivo fonte, como é o caso dos formulários estáticos do Delphi, que são formados pelos arquivos .PAS e .DFM. Neste caso, seria necessário gerar código usando um *template* para o arquivo .DFM e, em seguida, escolhendo outro *template*, efetuar a mesma geração para o arquivo .PAS. O uso da variável OTHERTEMPLATEFILE evita este esforço, pois, estando corretamente definidos dois *templates* (um para cada arquivo) basta informar em um dos *templates* o nome absoluto do arquivo do outro e, quando o primeiro terminar a sua execução, o segundo automaticamente é iniciado.

Outra variável importante, também apresentada no Quadro 27, é CLASSMAPPERFILE, que indica o nome relativo do arquivo de mapeamento de classes, com extensão *cmpp*. Este arquivo também possui um conjunto de regras para a sua definição, e também será submetido

aos analisadores léxico, sintático e semântico antes de serem carregados para o *template*. O Quadro 28 apresenta o conteúdo de um arquivo de mapeamento de classes usado no *template* mostrado no Quadro 21.

```
[CLASSMAPPER]
'screen'     = 'TForm'          GROUP = 0,1,2,3     ;
'textbox'    = 'TEdit'          GROUP = 0,  2,  4   ;
'combobox'   = 'TComboBox'      GROUP = 0,  2,  4,5 ;
'panel'      = 'TPanel'         GROUP = 0,1,2       ;
'radio'      = 'TRadioButton'   GROUP = 0,1,2,3,4   ;
'dialog'     = 'TLabel'         GROUP = 0,1,2,3     ;
'button'     = 'TButton'        GROUP = 0,1,    4   ;
'checkbox'   = 'TCheckBox'      GROUP = 0,1,2,3,4   ;
[/CLASSMAPPER]
```

Quadro 28 – Arquivo de mapeamento de classes

O apêndice G apresenta a gramática e as ações semânticas da linguagem de definição do arquivo de mapeamento de classes. Pela sua definição sintática, o corpo de um arquivo de mapeamento de classes deve estar delimitado pelos marcadores `[CLASSMAPPER]` e `[/CLASSMAPPER]`. A primeira linha do corpo do arquivo indica que um elemento do EA que tenha `Stereotype` igual a `screen` terá a variável `@TARGETLANGCLASS_` igual a `TForm`. A mesma analogia é aplicada às demais linhas de definição do arquivo. Já a declaração `GROUP` tem um comportamento diferente das demais definições: trata-se de um agrupamento por características para facilitar a criação dos *templates*. O Quadro 29 ajuda a compreender melhor o uso deste recurso.

Elemento/ Propriedade	TForm	TButton	TCheckbox	TCombobox	TPanel	TRadio	TLabel	TTextBox	GROUP
Left	X	X	X	X	X	X	X	X	0
Top	X	X	X	X	X	X	X	X	0
Width	X	X	X	X	X	X	X	X	0
Height	X	X	X	X	X	X	X	X	0
Font.Color	X	X	X	X	X	X	X	X	0
Font.Height	X	X	X	X	X	X	X	X	0
Font.Name	X	X	X	X	X	X	X	X	0
Font.Style	X	X	X	X	X	X	X	X	0
Caption	X	X	X		X	X	X		1
Color	X		X	X	X	X	X	X	2
ParentColor	X		X			X	X		3
TabOrder		X	X	X		X		X	4
ItemHeight				X					5

Quadro 29 – Agrupamento de elementos por características

Pode-se observar no Quadro 29 que as propriedades em azul são comuns a todas as classes de elementos, portanto, todas pertencem ao mesmo grupo, o grupo zero. Contudo, algumas propriedades são específicas de algumas classes de elementos. Estas foram coloridas em vermelho, e, cada agrupamento de elementos que compartilham determinada propriedade também pertence ao grupo desta propriedade. Assim, a classe `TButton`, que possui todas as

propriedades em azul e mais as propriedades `Caption` e `TabOrder`, pertencerá aos grupos zero, um e quatro. Neste processo é necessário que se compreenda que um elemento visual pode fazer parte de mais de um grupo e, assim sendo, quando for programada no *template* a instrução mostrada no Quadro 30, o analisador irá verificar se o valor da comparação é igual a "um dos grupos" aos quais o elemento em questão pertence. No caso da classe `TButton`, por exemplo, as duas condições mostradas retornarão "verdadeiro".

```
if (@GROUP = 0) then {
}
if (@GROUP = 4) then {
}
```

Quadro 30 – Instrução `if`

A definição sintática da linguagem do *template* possui ainda alguns comandos a serem apresentados. O Quadro 31 traz a lista destes comandos e suas respectivas semânticas.

Instrução	**Token**	**Descrição**
Início do corpo do *template*	`start`	Divisor que indica o início das declarações que darão efetivamente origem ao código fonte.
Desvio	`if`	O comando `if` é o mesmo conhecido das linguagens de programação.
	`else`	O comando `else`, também conhecido das linguagens de programação, é o complemento opcional do comando `if`.
Repetição	`LOOP`	O comando `[LOOP]` indica ao analisador o início de um bloco que deve ser repetido para cada elemento visual da tela.
	`CLOSELOOP`	Complemento obrigatório do comando imediatamente acima, indica o término de um bloco de repetição.
Quebra de linha	`<NL>`	Instrui o analisador e iniciar uma nova linha de código.
Comando de escrita	`wrt`	Indica um bloco de valores que serão gerados no código. Este bloco deve ser delimitado por parênteses.
	`write`	Tem a mesma semântica do comando imediatamente acima.
Modificadores de caso	`UCASE`	Indica o início de um bloco de código fonte que será gerado em letra maiúscula.
	`UC`	Tem a mesma semântica do comando imediatamente acima.
	`LCASE`	Indica o início de um bloco de código fonte que será gerado em letra minúscula.
	`LC`	Tem a mesma semântica do comando imediatamente acima.
	`CLOSECASE`	Complemento opcional dos quatro comandos imediatamente acima, indica o término do efeito do último deles declarado.

Quadro 31 – Instruções da linguagem do *template*

3.5.2.4 Processamento da saída

O processo de geração de código fonte é iniciado com as análises léxica e sintática do corpo do *template*, cuja linguagem foi especificada usando a notação BNF e, a partir desta especificação, com o uso da ferramenta GALS foram gerados os analisadores léxico e sintático.

Desta forma, cada tela que for gerar seu código fonte deverá submeter seu *template* a estes analisadores. O Quadro 32 mostra o código do método responsável pela geração de código.

```
procedure TEAScreen.GenerateCode(AManipulator : Pointer ; AOutputDir :
WideString);
var
  Parser : TTemplateParser;
  xTemplate : TTemplate;
begin
  xTemplate := FTemplate;
  while xTemplate <> nil do
  begin
    AddMemoAndLogMessage(GetDialogMessage(45),[Self.Name]);
    try
      ResetActiveElement;
      MountFileName;
      ChangeSourceCodeFileName(xTemplate.FileExt);
      Parser := TTemplateParser.Create(AManipulator,Self,xTemplate,UserMessage);
      Parser.Parse(xTemplate.GetContents);
      SaveSourceCode(AOutputDir);
    finally
      Parser.ClearMemory;
      Parser.Free;
      xTemplate := xTemplate.OtherTemplate;
      if xTemplate <> nil then
      begin
        xTemplate.ReloadContents;
        if not FSelfTemplate then
          RefreshTemplate(xTemplate);
      end;
    end;
  end;
end;
```

Quadro 32 – Método para geração de código da tela

O método exibido no Quadro 32 recebe um parâmetro do tipo `Pointer`, que representa o manipulador da aplicação. Foi usado o ponteiro genérico `Pointer` porque a definição do diagrama de classes diz que a classe `TManipulator` não é visível para a classe `TEAScreen`, assim, a classe que se interessar por este ponteiro deverá fazer a devida conversão de tipo. O processo de geração de código do método é submetido a um laço, e será executado para todos

os *templates* de uma possível sucessão, que passa a existir caso o atributo `ATemplate.OtherTemplate` seja diferente de `nil`. O método `ChangeSourceCodeFileName` altera a extensão do arquivo que será gerado, e é necessário porque cada *template* da sucessão pode definir uma extensão diferente para o arquivo que irá gerar. O método `Parse`, chamado em `GenerateCode` e cuja implementação é exibida pelo Quadro 33, irá iniciar o processo de análise do *template*.

```
procedure TTemplateParser.Parse(const str: String);
begin
  if Length(str) = 0 then
    Exit;

  FLex := TTMLexico.create(str);
  FSint := TTMSintatico.create;
  try
    FSint.parse(FLex, Self);
  finally
    FLex.Free;
    FSint.Free;
  end;
end;
```

Quadro 33 – Método `Parse`

Pela implementação do método nota-se que a classe `TTemplateParser` instancia os objetos dos analisadores léxico e sintático e, posteriormente, inicia o processo de análise. A cada *token* encontrado, o analisador sintático verifica se o referido *token* é de interesse do analisador semântico (identificando a presença do caracter "#", seguido de um identificador numérico, à frente do *token*) e, caso seja, este é executado para realizar as devidas ações semânticas. O processo de análise semântica consiste basicamente em montar trechos de código e, quando encontrar a instrução de quebra de linha ou fim do *template*, envia o trecho mondado para o objeto da tela que chamou o analisador. Ao final deste processo, não ocorrendo erros, a própria tela irá gerar o arquivo de código fonte (método `SaveSourceCode()` mostrado no Quadro 32).

3.5.3 Operacionalidade da implementação

Esta seção descreve os passos necessários para configuração do *plugin* e para sua correta utilização. Depois de esclarecido o processo de instalação, são apresentadas as modificações realizadas no registro do Windows, bem como apresentados os diretórios criados para uso do *plugin*.

3.5.3.1 Instalação do *plugin*

Para o correto funcionamento do *plugin* são necessários alguns procedimentos para preparação do ambiente, tais como criação de chaves no registro do Windows e criação dos diretórios usados durante a execução. Para que o usuário não precise realizar esta tarefa manualmente, é disponibilizado o instalador `Installer.exe`. Este aplicativo executa, de forma automática, todos os procedimentos necessários, deixando o ambiente pronto para o uso do *plugin*. O único cuidado que se deve tomar é que no diretório em que estiver o executável do instalador deve haver um subdiretório chamado `files`, e dentro deste deverão estar todos os arquivos necessários ao funcionamento do *plugin*. O instalador irá verificar previamente se há uma instalação do *plugin* e, caso haja, onde aparece o botão *Install* irá aparecer o botão *Uninstall*, que fará o papel de desinstalador da aplicação. O instalador traz automaticamente o diretório de instalação do EA, onde deverá ser instalado o *plugin*. A Figura 17 mostra a interface do instalador.

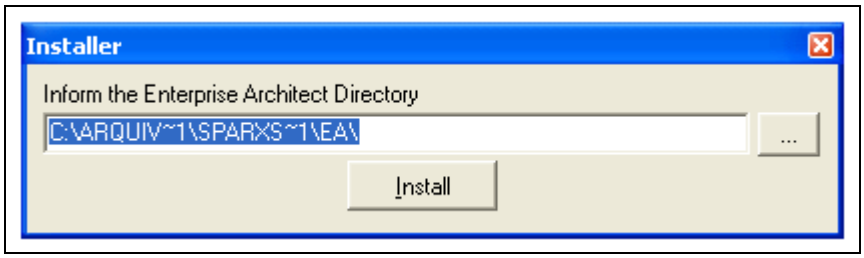


Figura 17 – Instalador do *plugin*

A seguir são apresentadas as atividades realizadas pelo instalador do *plugin*, O processo de instalação consiste em registrar a DLL no SO e montar a estrutura de diretórios e arquivos. Para registrar a DLL pode-se usar o aplicativo `regsvr32.exe`, disponível no Windows, passando como parâmetro o caminho absoluto do arquivo do *plugin*. A Figura 18 mostra um exemplo do uso deste aplicativo.

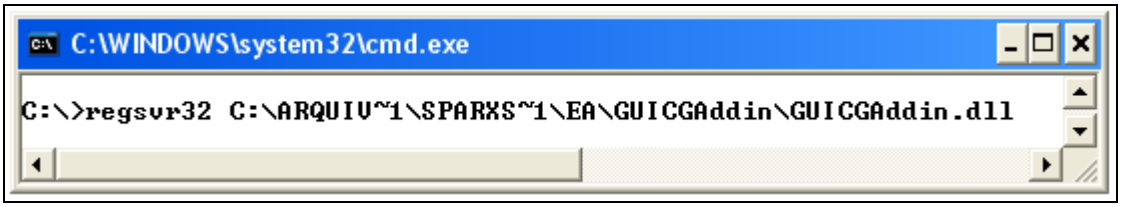


Figura 18 – Aplicativo revsvr32.exe

Feito isto, é preciso criar uma chave uma chave no registro do Windows para que o EA encontre a DLL. A Figura 19 mostra o conteúdo do registro do Windows após este procedimento.

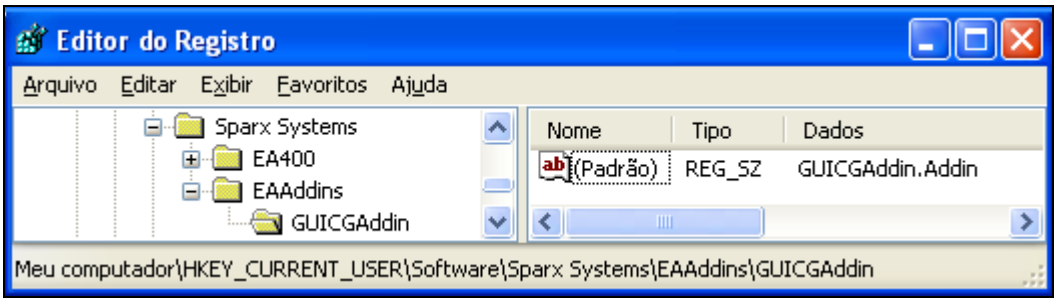


Figura 19 – Registro do EA

No registro exibido pela Figura 19 , caso a chave `EAAddins` não exista, é necessário que seja criada. Dentro desta deve ser criada uma outra chave, com o mesmo nome do projeto da DLL, e que possuirá um valor padrão formado pelo nome do projeto da DLL e pelo nome da classe que implementa a interface de automação do EA. Feitas estas configurações, ainda é necessário criar a estrutura de diretórios do *plugin*. Esta estrutura tem como nodo principal o diretório `GUICGAddin`, cuja composição é demonstrada na Figura 20.

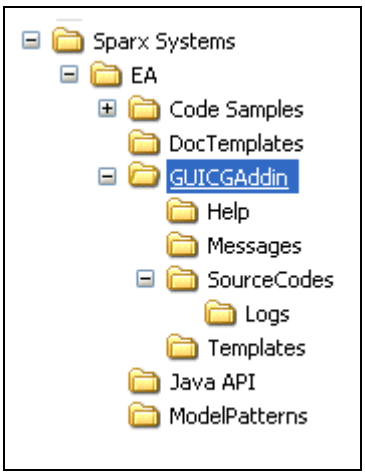


Figura 20 – Estrutura do diretório `GUICGAddin`

A partir do diretório `GUICGAddin`, que deve obrigatoriamente estar dentro do diretório de instalação do executável do EA, outros diretórios devem ser criados, conforme lista explicativa abaixo:

a) `GUICGAddin`: raiz de funcionamento da aplicação, e neste diretório que fica armazenado o arquivo `GUICGAddin.dll`;

b) `Help`: neste diretório está armazenado o arquivo HELP.HLP, que traz instruções de uso do plugin;

c) `Messages`: neste diretório são armazenados os arquivos `Dialogs.dlg` e `Errors.err`. O primeiro contém as mensagens de diálogo com o usuário, e o segundo armazena as mensagens de erro. É importante notar que estes arquivos são editáveis, contudo, caso suas estruturas sejam alteradas erroneamente, todo o sistema de mensagens do *plugin* estará comprometido;

d) `SourceCodes`: local de armazenamento padrão para os códigos que são gerados. Quando a interface principal é apresentada, é este diretório que aparece no campo `Path`;

e) `Logs`: diretório onde são armazenados os arquivos de *log*;

f) `Templates`: local de armazenamento dos arquivos de *templates* (extensão *tmpl*), arquivos de mapeamento de classes (extensão *cmpp*) e arquivos de caracteres especiais (extensão *sps*).

3.5.3.2 Operacionalidade do *plugin*

O primeiro passo para utilizar o *plugin* é verificar se ele foi carregado corretamente pelo EA. Isto pode ser feito através do menu *Add-Ins / Manage Add-Ins*. Acessando esta opção é exibida a tela ilustrada pela Figura 21, que apresenta a lista de *plugins*. O estado *Enabled* indica que o *plugin* está instalado e habilitado para ser usado. Caso não esteja habilitado, a coluna *Status* apresentará o erro ocorrido no processo de execução.

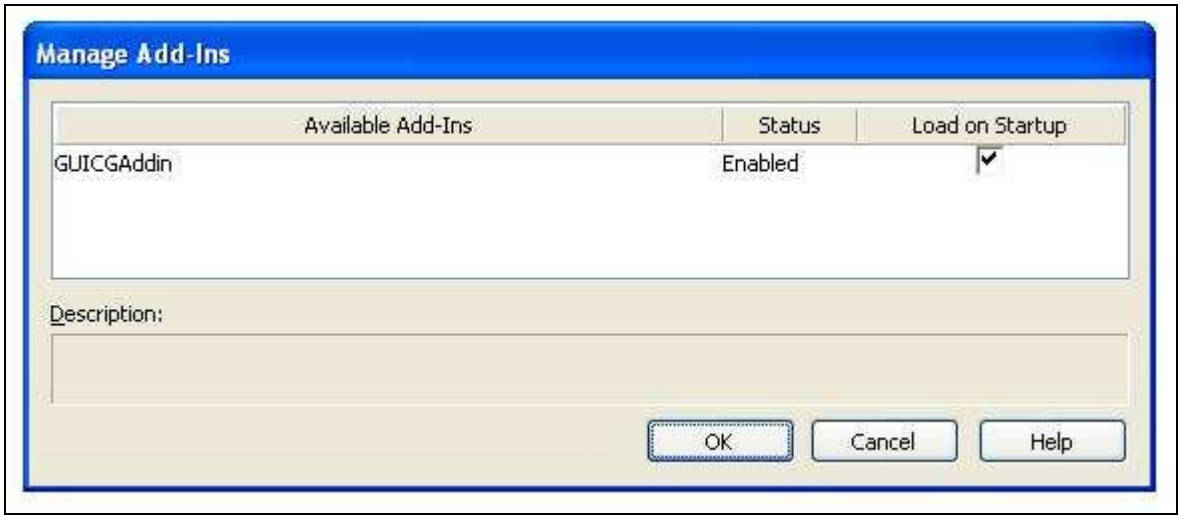


Figura 21 – *Manage Add-Ins*

Habilitado o *plugin*, o usuário pode acessá-lo através do menu *Add-Ins /GUI Code Generator*, ou pode clicar com o botão direito do mouse, que exibirá o menu apresentado na Figura 22.

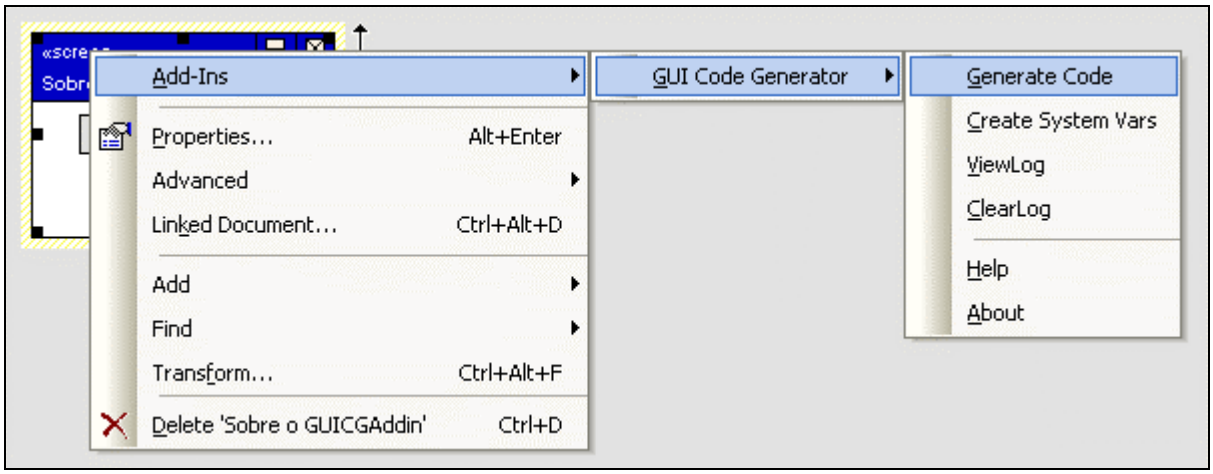


Figura 22 – Executando o *plugin*

Ambas as opções sugeridas anteriormente apresentarão a interface principal da aplicação, contudo, dependendo do item que for selecionado pelo usuário, o *plugin* terá um comportamento diferente. A seguir são apresentadas as opções de seleção disponíveis:

a) se o usuário executar o *plugin* clicando sobre uma tela, esta será carregada na lista e exibida na interface principal. Caso esta tela selecionada herde características de outra tela (conector *Generalize*), então esta também é carregada, e assim sucessivamente, com efeito cascata;

b) se o *plugin* for executado partir de um conjunto de telas selecionadas, a última destas será carregada e apresentada na interface principal;

c) caso o *plugin* seja executado clicando-se sobre um diagrama, todas as telas pertencentes a este diagrama serão carregadas e apresentadas ao usuário;

d) se o *plugin* for executado a partir de um pacote, todas as telas pertencentes a todos os diagramas deste pacote serão carregadas e exibidas ao usuário.

Nos itens (c) e (d), caso não haja telas nos contêineres selecionados é apresentada uma mensagem de erro ao usuário e, ao clicar sobre o botão “OK” a interface é exibida, porém, com os botões de geração de código desabilitados. Neste caso o usuário ainda terá a possibilidade de manipulação dos *templates* disponíveis. A Figura 23 apresenta a interface principal do *plugin*, gerada partir da seleção de um diagrama, e a figura Figura 24 mostra o diagrama de atividades do uso da aplicação.

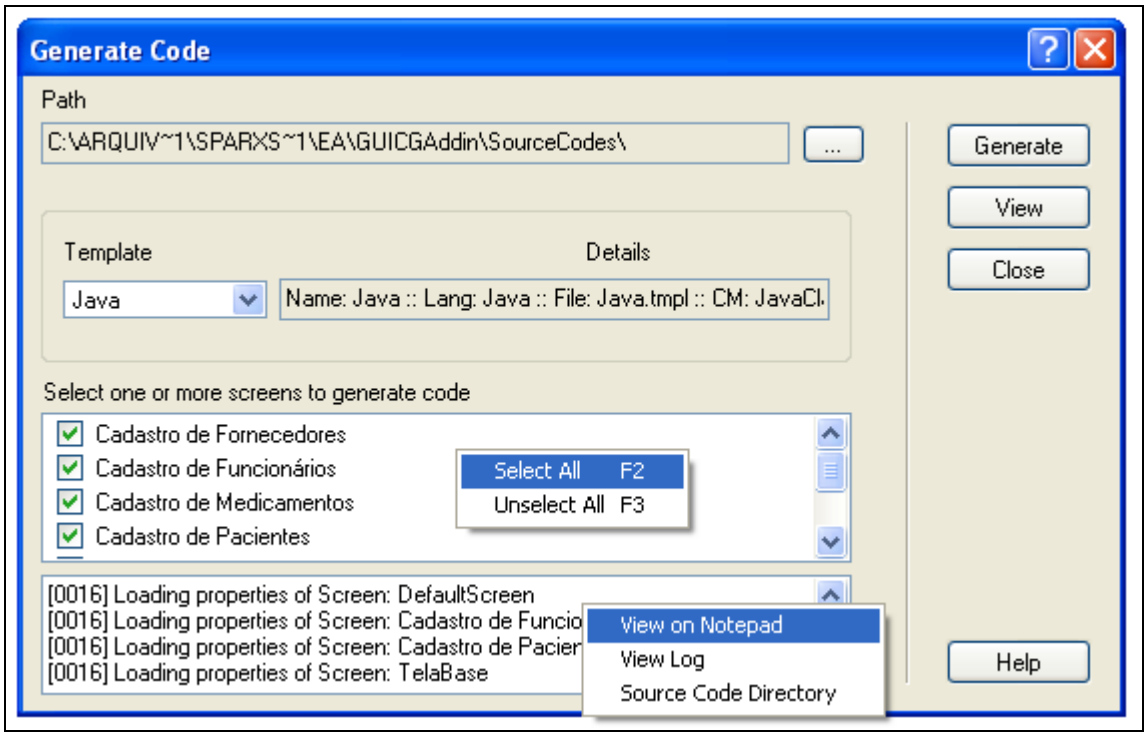


Figura 23 – Interface principal do *plugin*

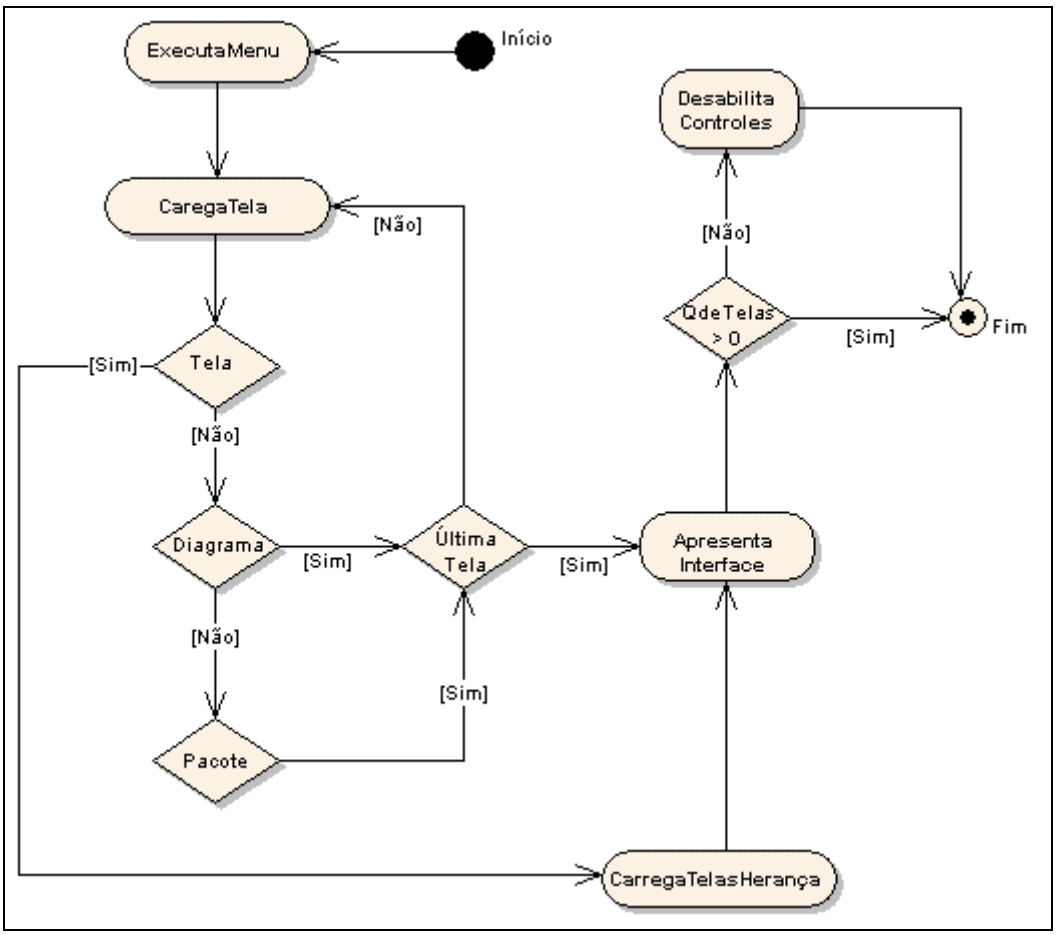


Figura 24 – Diagrama de atividades do uso do *plugin*

O campo *Path* indica o diretório onde serão salvos os arquivos de código fonte da aplicação, e apresenta, à sua direita, um botão para pesquisa de diretórios do Windows. O

campo *Template* apresenta uma lista de *templates* disponíveis ao usuário para geração de código. Caso algum destes *templates* apresente erros em sua construção, este aparecerá desabilitado, não sendo possível a geração de código a partir dele. O nome que aparece neste campo é o mesmo valor atribuído à variável `NAME`, apresentada no Quadro 27. Passando o mouse por sobre este campo é exibida a descrição do *template*, cujo valor é assinalado na variável `TEMPLATEDESC`, também apresentada no Quadro 27. O campo *Details* apresenta algumas informações sumárias sobre o *template*, como nome, linguagem alvo, nome do arquivo de mapeamento de classes e nome do arquivo do próprio *template*. Abaixo, com botões de seleção, aparece uma lista com as telas escolhidas pelo usuário. A princípio todas elas aparecem selecionadas e, caso o usuário deseje, poderá desfazer esta seleção, mantendo para geração apenas as telas que lhe interessar. A partir desta lista é possível acessar um menu de contexto, que permite selecionar todas as telas, ou nenhuma tela. Estas opções também estão disponíveis clicando nas teclas F2 ou F3 sobre a lista. Mais abaixo aparece um campo com uma lista de informações ao usuário, que traz alguns dados sobre o processo de geração de código. A partir deste campo é possível abrir um menu de contexto com as seguintes opções:

a) *View on Notepad*: abre o bloco de notas do Windows com o conteúdo deste campo. Este conteúdo é salvo em um arquivo temporário, salvo no mesmo diretório de *log* de geração de código;
b) *View Log*: mostra o *log* de processamento da geração de código. Este *log* é bem mais completo, e traz informações além das exibidas ao usuário na tela. O arquivo é salvo com um nome referente ao dia atual, sucedido da extensão *log*;
c) *Source Code Directory*: abre o diretório escolhido pelo usuário para geração de código.

À direita da tela principal estão localizados os botões para escolha do usuário. O botão *Generate* dá início ao processo de geração de código. O botão *Close* fecha a tela do *plugin* e o botão *Help* abre o arquivo de ajuda do usuário. O botão *View* abre uma outra tela (Figura 25), que apresenta ao usuário o *template* selecionado na tela principal, permitindo-lhe a edição deste.

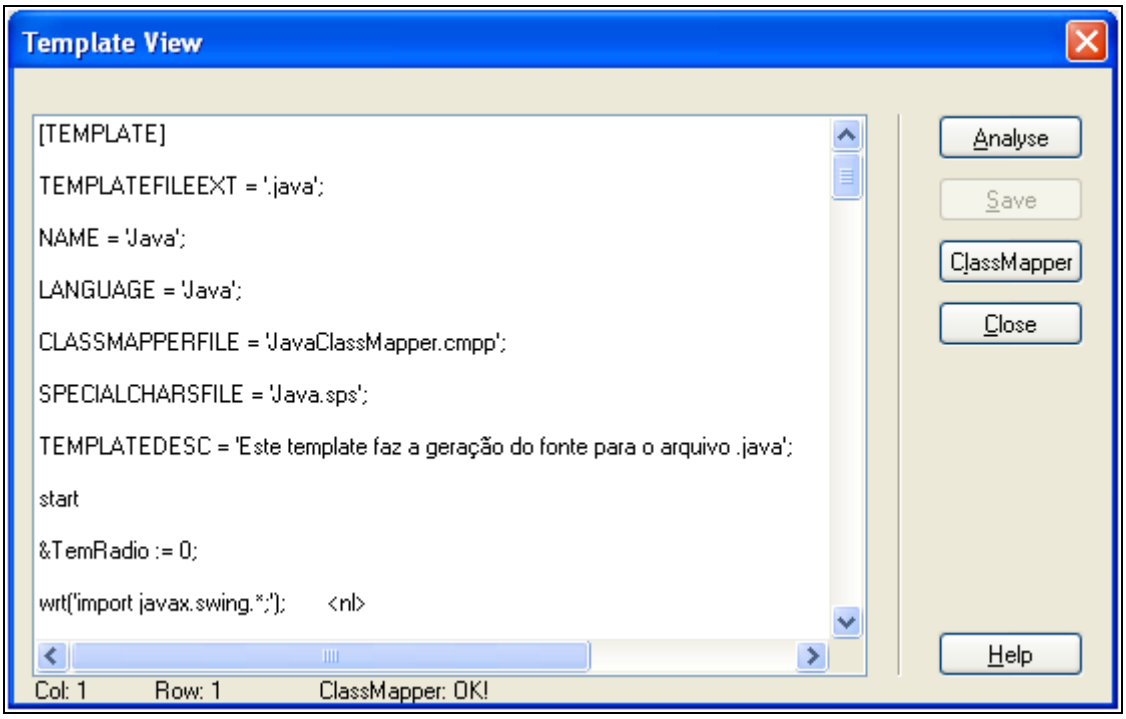


Figura 25 – Interface de edição do *template*

Ao usuário é permitido editar o *template* selecionado, salvá-lo e efetuar sobre ele as análises léxica, sintática e semântica. Na opção *Save* não é possível escolher o diretório nem alterar o nome do arquivo. Caso o *template* seja salvo com erros será exibida uma mensagem ao usuário e, caso este confirme a operação, o referido *template*, por conter inconsistências, será desabilitado na interface principal. O botão *Analyse* efetua as análises léxica e sintática sobre o conteúdo apresentado na tela, e exibe uma mensagem ao usuário acerca do resultado do processo. O botão *Close* fecha a tela, o botão *Help* chama o arquivo de ajuda do usuário e, por fim, o botão *ClassMappper* exibe o arquivo de mapeamento de classes do *template*.

A seguir é apresentado um estudo de caso para exemplificar a utilização do *plugin*. A Figura 26 mostra uma tela projetada no EA, com todos os componentes propostos para geração de código, enquanto a Figura 27 exibe a referida tela gerada para Java.

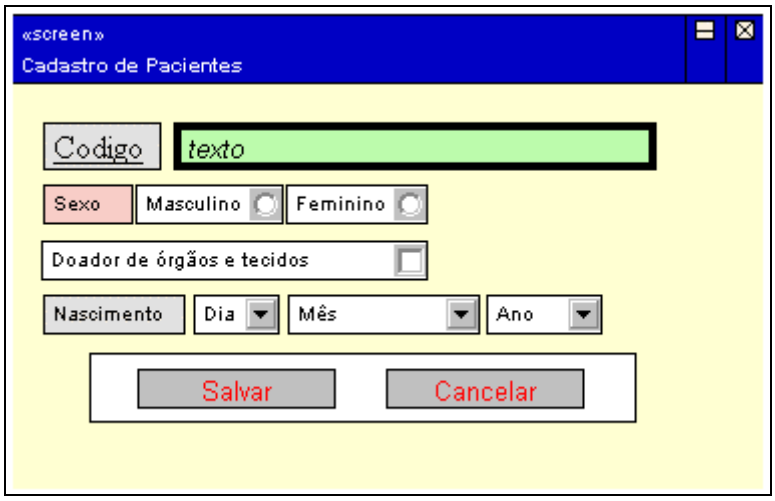


Figura 26 – Tela projetada no EA

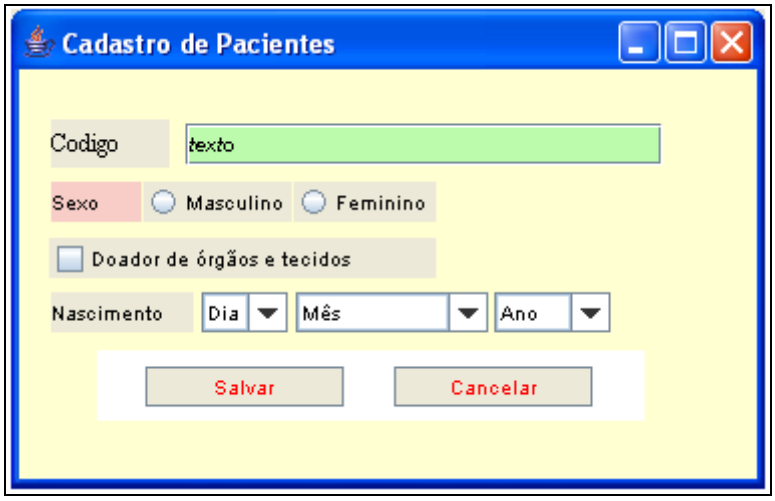


Figura 27 – Tela gerada em Java

Pode-se observar que a tela gerada em Java mantém grande parte das características atribuídas no EA, exceto pelo estilo de fonte do componente “Codigo”, cujo texto não aparece sublinhado. Isso ocorre porque o pacote AWT do Java não possui o estilo sublinhado. O código fonte que deu origem a tela exibida na Figura 27 é devidamente abordado no apêndice B.

A seguir, a Figura 28 apresenta a mesma tela, agora em Delphi, gerada dinamicamente. Da mesma forma que a biblioteca gráfica do Java possui limitações em relação aos atributos de algumas classes, a biblioteca do Delphi também possui. É possível notar que os botões “Salvar” e “Cancelar” não receberam a cor de fundo, e isso se deve ao fato de a classe base dos botões do Delphi (`TButtonControl`) não possuir o atributo relativo a cor de fundo. O código fonte que deu origem a tela exibida na Figura 28 está presente no apêndice C.

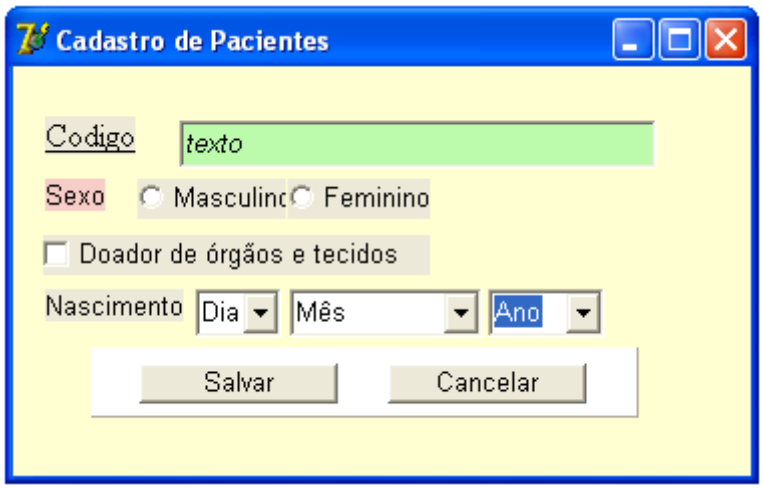


Figura 28 – Tela gerada em Delphi (formulário dinâmico)

Por fim, a Figura 29 exibe a mesma tela mostrada pela Figura 26, agora de forma estática. Para construção desta tela são usados os arquivos .DFM e .PAS, cujos códigos fonte são apresentados apêndice D.

Figura 29 – Tela gerada em Delphi (formulário estático)

No caso de formulários estáticos, além da restrição relacionada à cor de fundo dos botões, há uma outra limitação, relacionada ao tamanho da fonte dos componentes. Ocorre que, ao contrário do arquivo .PAS, o arquivo .DFM não permite que se faça referência ao atributo `Font.Size`, uma vez que em sua definição é usado o atributo `Font.Height`. Segundo Borland Software Corporation (2002), este atributo é usado para definir a altura da fonte em *pixels*, e seu valor é obtido a partir da fórmula exibida no Quadro 34. Assim, o valor referente ao tamanho da fonte deve ser definido dinamicamente, através do atributo `Font.Size`.

```
Font.Height = - (Font.Size * Font.PixelsPerInch / 72)
```

Quadro 34 – Fórmula do atributo `Font.Height`

Há ainda outro recurso disponível no EA que o *plugin* trata: são os *Tagged Values*, que

permitem ao usuário definir propriedades especiais para os elementos. Contudo, apesar desta possibilidade, o *plugin* reconhece apenas uma quantidade restrita de propriedades pré-definidas para este fim, conforme listado no Quadro 35.

Variável	**Tipo**	**Descrição**
`@BackColor`	*integer*	Cor de fundo do elemento.
`@BorderColor`	*integer*	Cor da borda do elemento.
`@BorderWidth`	*integer*	Largura da borda do elemento.
`@FontBold`	*boolean*	Se a fonte do elemento está em negrito.
`@FontItalic`	*boolean*	Se a fonte do elemento está em itálico.
`@FontUnderlined`	*boolean*	Se a fonte do elemento está sublinhada.
`@FontColor`	*integer*	Cor da fonte do elemento.
`@FontName`	*string*	Nome da fonte do elemento.
`@FontSize`	*integer*	Tamanho da fonte do elemento.
`@Height`	*integer*	Altura do elemento.
`@Inherits`	*boolean*	Se o elemento herda características da tela, ou se a tela herda características de outra tela.
`@Propagate`	*boolean*	Se a tela propaga suas propriedades para seus elementos visuais. Disponível apenas para elementos do tipo *Screen*.
`@SourceCodeFileName`	*string*	Nome relativo do arquivo em que será gerado o código fonte para a tela. Disponível apenas para elementos do tipo *Screen*.
`@TemplateFileName`	*string*	Uma tela pode possuir um *template* diferente daquele selecionado pelo usuário na interface do *plugin*. Assim, caso o usuário deseje, ele pode informar nesta propriedade o nome absoluto de outro *template* qualquer. Disponível apenas para elementos do tipo *Screen*.
`@VarName`	*string*	Nome de variável atribuída ao elemento.
`@Width`	*integer*	Largura do elemento.

Quadro 35 – Propriedades disponíveis em *tagged values*

O usuário poderá criar estas variáveis manualmente, acessando o menu *Settings / UML / Tagged Value Types*, onde poderá criar todas, ou apenas aquelas que lhe interessam. Para que não haja a necessidade de fazer este processo manualmente o *plugin* disponibiliza um recurso que o faz de forma automática, através do menu *Add-Ins / GUI Code Generator / Create System Vars*. É preferível usar este recurso a fazer o processo de forma manual, pois ele cria as variáveis com os devidos tipos assinalados, observando o tipo de elemento em que serão usadas. As atribuições feitas em *tagged values* sobrepõem as definições realizadas diretamente nos elementos.

Estas mesmas variáveis apresentadas no Quadro 35 podem ter seus valores atribuídos através do campo *notes* dos elementos visuais. Sobre o conteúdo deste campo são realizadas análises léxica, sintática (cuja definição é apresentada no apêndice H) e semântica, em busca de definições que possuam significado para a geração de código. Os valores inseridos que não possuam significado são ignorados na análise semântica. As atribuições feitas no campo

notes sobrepõem as definições feitas diretamente nos elementos, e sobrepõem também as definições feitas em *tagged values*. A Figura 30 mostra um exemplo do uso deste campo para definição de valores de propriedades dos elementos.

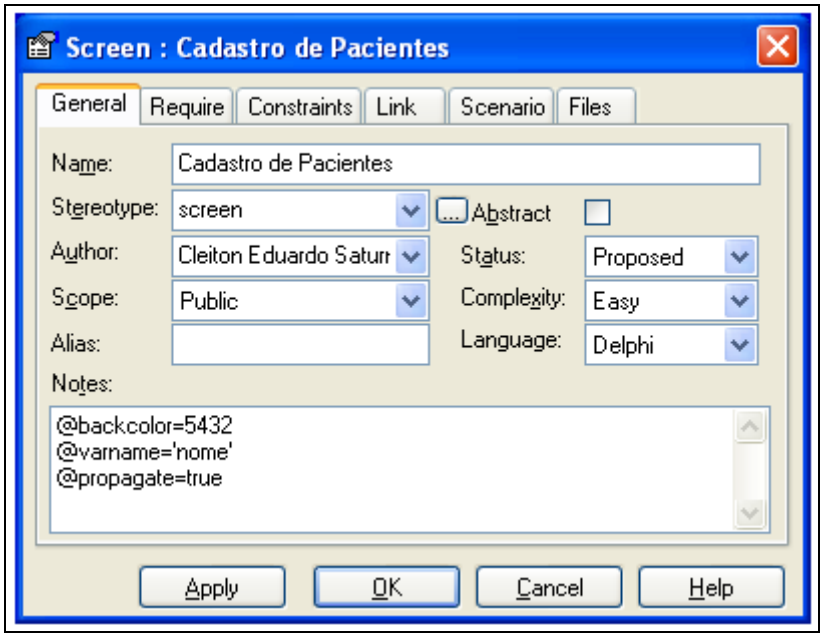


Figura 30 – Uso do campo n*otes*

O Quadro 35 apresenta duas variáveis que têm comportamento peculiar no processo de geração de código: `@PROPAGATE` e `@INHERITS`. Estas variáveis de sistema dizem respeito ao sistema de herança do *plugin*. Pode haver dois tipos de herança: herança visual entre a tela e seus elementos, e herança visual entre telas. A primeira permite ao usuário definir, para cada elemento, se ele irá herdar as características de sua tela; é permitido ainda informar se a tela irá propagar suas características para os seus elementos. Já a herança entre telas permite que o usuário defina uma tela base, com características padrões e, através do uso de conectores do tipo *Generalization*, fazer com que outras telas herdem características desta tela base.

A variável `@PROPAGATE` (exclusiva de telas), que pode ser `true` ou `false`, informa ao *plugin* se uma determinada tela irá propagar seus atributos para seus elementos. Já a variável `@INHERITS`, que também receberá `true` ou `false`, informa ao *plugin* se determinado elemento herdará valores de atributos da tela a qual pertence, ou se determinada tela herdará características de outra tela a ela conectada.

Como ambas as variáveis podem ser usadas em conjunto, cabe esclarecer alguns pontos para melhor compreensão deste recurso. Tome-se como exemplo uma tela que tenha a variável `@PROPAGATE` assinalada com `true`. Neste caso, todos os elementos pertencentes à tela herdarão os valores dos atributos desta, exceto aqueles que tiverem a variável `@INHERITS` definida como `false`. Caso a tela defina `@PROPAGATE` igual a `false`, os elementos não

herdarão as suas características, exceto aqueles que definirem `@INHERITS` igual a `true`. Assim, conclui-se que a definição final da herança sempre será dada pelos elementos visuais e, quando estes não o fizerem, vale a definição da tela. Já a definição de herança entre as telas é mais simples: se houver conector *Generalization* ligando duas telas, há herança, caso contrário, não há. Para que o mecanismo de herança entre telas funciona corretamente dois fatores devem ser observados:

a) o primeiro fator é que uma tela só poderá herdar características de uma, e somente uma, outra tela (ou seja, não trata herança múltipla). Caso esta condição não seja respeitada, o *plugin* irá considerar o primeiro conector de herança que encontrar para a referida tela;
b) a outra condição impõe que não sejam criadas estruturas circulares de herança, pois caso haja, o *plugin* entrará em uma busca infinita para achar o último elemento desta herança.

3.6 RESULTADOS E DISCUSSÃO

A ferramenta atendeu com fidelidade a todos os requisitos apresentados, incluindo algumas funcionalidades não propostas, mas que se provaram úteis no processo de geração de código, como é o caso do uso de *templates*, e do aproveitamento do recurso de *tagged values*, além do reconhecimento semântico do atributo *notes* dos elementos. A geração de código é realizada para todos os componentes propostos possibilitando, ainda, a geração de outros, bastando que sejam observadas as regras de mapeamento de classes já expostas. Pode-se comprovar esta afirmação nas telas demonstradas na seção 3.5.3.2, que possuem o elemento de estereótipo `label` que, apesar de não estar presente nos requisitos, com o correto mapeamento de classe foi gerado com sucesso, de acordo com a definição do *template*.

Há algumas soluções que possuem funcionalidades correlatas à desenvolvida neste trabalho, como é o caso da ferramenta Delphi2Java-II (SILVEIRA, 2006) e da ferramenta de geração de código em PHP a partir de um diagrama *Web Application Extension* (WAE[6]) (KREUTZFELD NETO, 2003). Apesar de nenhuma delas possuir as mesmas características

[6] "WAE é uma extensão da UML que permite a modelagem de sistemas web." (KREUTZFELD NETO, 2003, p. 12).

da solução construída neste trabalho, há alguns pontos em comum como o uso de *templates*, no caso de Delphi2Java-II, e o uso de recursos da tecnologia COM, no caso da ferramenta geradora de código em PHP.

4 CONCLUSÕES

Com o desenvolvimento da ferramenta foi possível comprovar os benefícios trazidos pelas I-CASE e pela geração automática de código, visto que o processo de geração de código é bastante rápido, se comparado com o processo manual. Para se ter uma idéia da velocidade da geração, uma tela com 56 componentes visuais teve seu código gerado, para Java, em aproximadamente nove segundos. Deve-se acrescentar a esta estatística o tempo necessário para a construção do protótipo de tela no EA, contudo, este tempo é gasto apenas na primeira vez que se constrói o protótipo, restando depois a simples tarefa de gerar o código para a linguagem de destino.

O equilíbrio entre integração e flexibilidade, proposto pelo conceito de I-CASE, foi alcançado com sucesso com o uso dos controles ActiveX, para o processo de integração, e com o uso de *templates*, para o processo de geração de código. Os controles ActiveX forneceram um processo implícito de integração, em que o usuário "imagina" que está trabalhando com mais um recurso do EA, e não com uma ferramenta externa. A opção pelo uso de *templates* trouxe flexibilidade ao processo de geração de código, uma vez que se pode definir a forma de saída do código de acordo com a necessidade. Esta flexibilidade foi corroborada pela criação de *templates* que possibilitaram a geração de código para as linguagens Visual Basic (apêndice J) e C# (apêndice K).

A solução proposta cria uma inevitável (e saudável) separação entre a camada de interface da aplicação e as demais camadas que possam porventura existir, característica que vem ao encontro do que preconizam as ferramentas geradoras de código para GUI. Esta afirmação é corroborada pela possibilidade de se criar um projeto de interface com usuário independente da linguagem de programação que será usada no desenvolvimento, o que permite usar o mesmo esqueleto de interface para diversas plataformas de aplicação.

Um requisito importante no contexto das I-CASE diz respeito à usabilidade, e apregoa pela manutenção de um padrão visual entre as ferramentas do ambiente integrado. Esta meta também foi perseguida e alcançada pelo *plugin*, que teve como inspiração a tela de geração de código para o diagrama de classes do EA (Figura 31), tendo como produto a tela exibida pela Figura 32.

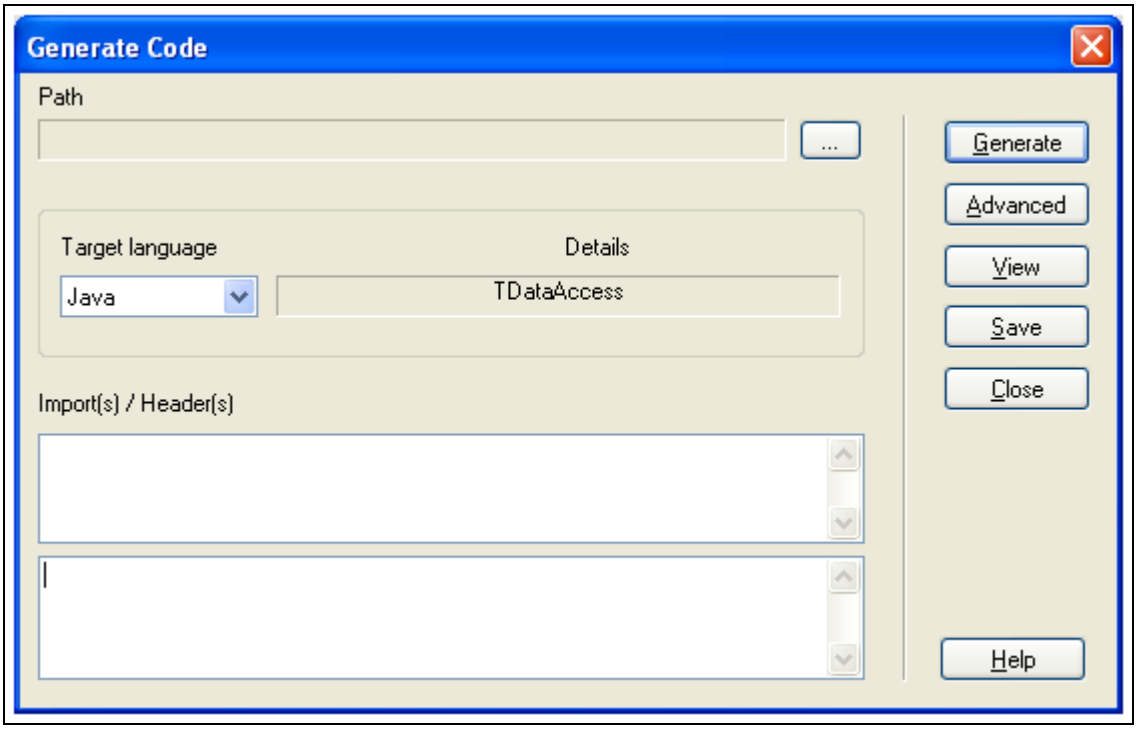


Figura 31 – Exemplo de tela do EA

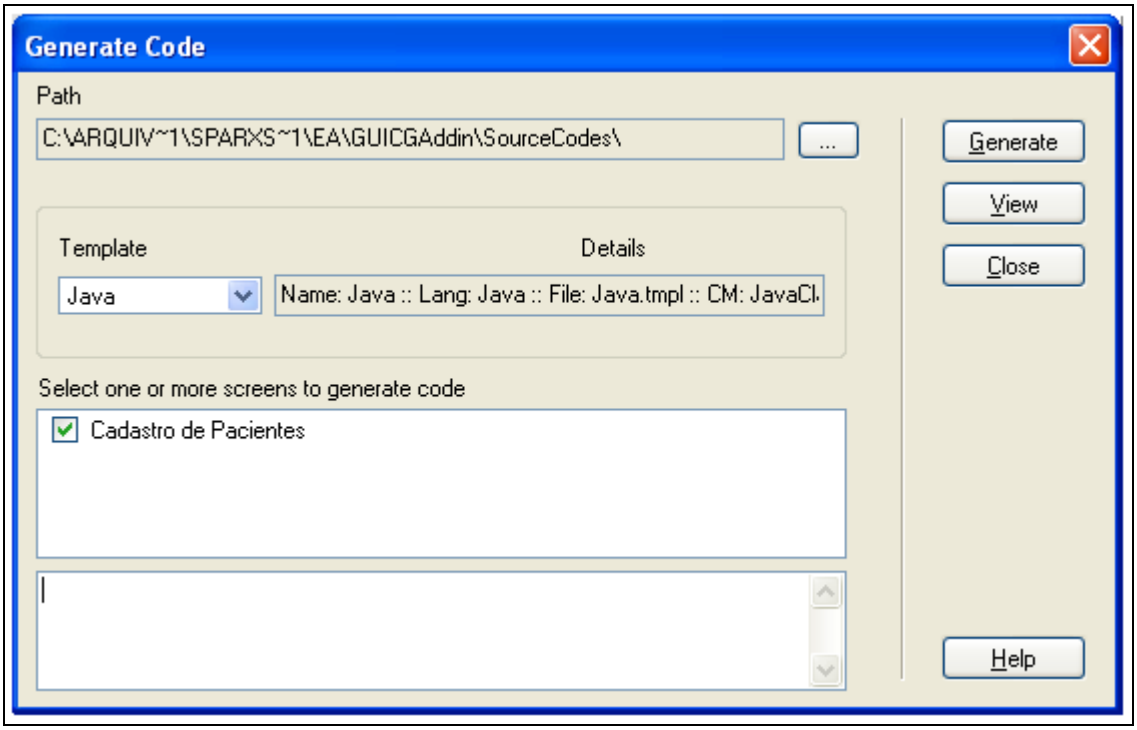


Figura 32 – Tela do *plugin*

Após concluída a implementação da solução proposta, pôde-se constatar que as ferramentas usadas no processo de desenvolvimento supriram todas as necessidades, em especial o GALS, que facilitou bastante o desenvolvimento dos analisadores léxico e sintático, usados exaustivamente em quatro processos distintos do *plugin*. Também os recursos providos pela tecnologia COM foram eficientes na criação de uma integração consistente entre o cliente e o *plugin*.

A ferramenta possui algumas restrições, principalmente em relação ao uso do *template*, que não permite a criação de estruturas de controle sofisticadas como *while*, *for* e *case*, bem como não permite operações aritméticas nem operações lógicas complexas.

Apesar de a ferramenta reconhecer a estrutura do EA composta por elementos dentro de outros elementos, esta característica não é mantida no processo de geração de código, ocasião em que todos os elementos são lançados dentro de uma mesma lista e gerados linearmente, obedecendo a ordem de inserção nesta lista. Esta restrição causa uma indefinição no processo de geração de código, pois, caso haja dois ou mais grupos (com semânticas distintas) de elementos do tipo `RadioButton`, por exemplo, não há como definir a qual grupo cada elemento pertence, o que culminará na geração de um grupo único destes elementos, possivelmente sem sentido para o contexto da aplicação.

Para melhor difusão da ferramenta construída foi criado um projeto no portal Sourceforge.net. As informações sobre o projeto podem ser obtidas a partir do endereço <http://sourceforge.net/projects/guicgaddin>, e os códigos fontes estão disponíveis no endereço <http://guicgaddin.cvs.sourceforge.net/guicgaddin/>.

4.1 EXTENSÕES

Na ferramenta desenvolvida, a principal linha para extensões é relacionada ao incremento das funcionalidades do *template* responsável pela geração da saída. Sugere-se, então, a criação de estruturas de controle mais sofisticadas (*for*, *while* e *case*) e a implementação de operações aritméticas. Atualmente, a estrutura de controle `if` do *template* permite somente uma condição simples, conforme pode ser visto nos quadros que os exemplificam. Nesta esteira, sugere-se a implementação de condições complexas, o que trará mais facilidades para construção dos *templates*.

Sugere-se, ainda, a criação de tratamento para o caso de o usuário criar uma estrutura de herança circular, e também a implementação de tratamento para herança múltipla e de seleção de múltiplas telas para geração. Também é sugerido o tratamento, na geração de código, de estruturas formadas por árvores de elementos, para resolver a restrição comentada em relação aos componentes `RadioButton`. Outro ramo de implementação é permitir que o usuário escolha se deseja habilitar alguns recursos do *plugin*, como uso dos atributos *notes* e uso dos *tagged values*, por exemplo. Isto seria muito interessante, visto que se o usuário não

pretende usar estes recursos, basta desabilitá-los, o que acarretará num aumento substancial de velocidade no processo de carga das informações. Este controle pode ser implementado com o uso de um arquivo de inicialização, por exemplo.

Existe algumas informações do *plugin* que são implementadas estaticamente, como nomes de diretórios e de arquivos, e valores padrões de propriedades de elementos. Sugere-se, então, a criação de um recurso para permitir ao usuário definir dinamicamente estes valores, de acordo com a necessidade. Por fim, sugere-se a implementação de um recurso que permita criar subdiretórios para geração de código de acordo com a estrutura de pacotes selecionada.

REFERÊNCIAS BIBLIOGRÁFICAS

BORLAND SOFTWARE CORPORATION. **Delphi enterprise**: help. Version 7.0. [S.l.], 2002. Documento eletrônico disponibilizado com o ambiente Delphi 7.0.

CANTÙ, M. **Dominando o Delphi 7:** a bíblia. Tradução Kátia Aparecida Roque. São Paulo: Pearson Education, 2003.

CARNEGIE MELLON SOFTWARE ENGINEERING INSTITUTE. **Graphical user interface builders**. [S.l.], 2007. Disponível em: <http://www.sei.cmu.edu/str/descriptions/guib.html>. Acesso em: 07 fev. 2006.

CHEN, M.; NORMAN, R. J. Integrated computer-aided software engineering (CASE): adoption, implementation, and impacts. In: HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 25., 1992, Hawaii. **Procedings...** [S.l.: s.n.], 1992. p. 362-373. Disponível em: <http://ieeexplore.ieee.org/iel2/378/4719/00183504.pdf?tp=&arnumber=183504&isnumber=4719>. Acesso em: 01 set. 2006.

DALGARNO, M. **Code generation information for the pragmatic engineer**. [S.l.], 2006. Disponível em: <http://www.codegeneration.net/tiki-index.php?page=FrequentlyAskedQuestions>. Acesso em: 26 ago. 2006.

FISHER, A. S. **CASE:** utilização de ferramentas para desenvolvimento de software. Tradução Info-Rio. Rio de Janeiro: Campus, 1990.

FONSECA, F. **Ferramenta conversora de interfaces gráficas**: Delphi2Java-II. 2005. 59 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <http://www.bc.furb.br/docs/MO/2005/306109_1_1.pdf>. Acesso em: 30 ago. 2006.

HERRINGTON, J. **Code generation in action**. Greenwich: Manning, 2003.

HORSTMANN, C. **Padrões e projetos orientados a objetos**. Tradução Bernardo Copstein. 2. ed. Porto Alegre: Bookmann, 2007.

KREUTZFELD NETO, S. **Geração de código PHP a partir da ferramenta CASE Rational Rose**. 2003. 58 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <http://www.bc.furb.br/docs/MO/2003/278876_1_1.pdf>. Acesso em: 18 out. 2003.

JAVAFREE.ORG. **Look and feel**. [S.l.], 2007. Disponível em: <http://www.javafree.org/javabb/viewtopic.jbb?t=3229>. Acesso em: 07 jul. 2007.

LIMA, A. S. **UML 2.0:** do requisito à solução. São Paulo: Érica, 2005.

PACHECO, X.; TEIXEIRA, S. **Delphi 5:** guia do desenvolvedor. Tradução Daniel Vieira. Rio de Janeiro: Campus, 2000.

PEIL, N. C. **Borland Delphi 5**. Pelotas, [2003?]. Disponível em: <http://www.cefetrs.tche.br/~npeil/>. Acesso em: 27 ago. 2006.

PLUGIN. In: WIKIPÉDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2006. Disponível em: <http://pt.wikipedia.org/wiki/Plugin>. Acesso em: 24 ago. 2006.

PRESSMAN, R. S. **Engenharia de software**. 5. ed. Tradução Mônica Maria G. Travieso. Rio de Janeiro: McGraw Hill, 2002.

ROCHA, Lucas. **APE**: plataforma para desenvolvimento de aplicações web com PHP. [S.l.], 2006. Disponível em: <http://twiki.im.ufba.br/bin/view/Aside/ProjetoConclusaoDeCursoAPEMonografia#4_2_Motores_de_templates>. Acesso em: 22 out. 2006.

SILVA, O. J. **Programando em Java 2**: Interfaces gráficas e aplicações práticas com AWT e Swing. São Paulo: Érica, 2004.

SILVEIRA, J. **Extensão da ferramenta Delphi2Java-II para suportar componentes de banco de dados**. 2006. 81 f. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SPARX SYSTEMS. **User guide**. [S.l.], 2007. Disponível em: <http://www.sparxsystems.com.au/EAUserGuide/index.html>. Acesso em: 11 maio 2007.

STEPHENS, M. **Automated code generation**: the fastad way to write software. [S.l.], 2006. Disponível em: <http://www.softwarereality.com/programming/code_generation.jsp>. Acesso em: 27 ago. 2006.

TRINDADE, C. **Java**: uma visão geral do pacote Swing. [S.l.], 2002. Disponível em: <http://www.imasters.com.br/artigo/410>. Acesso em: 05 set. 2006.

APÊNDICE A – Diagrama de classes de projeto

A Figura 33 mostra o diagrama de classes de projeto do *plugin*. Algumas classes tiveram seus métodos e atributos suprimidos em virtude de já terem sido apresentados no diagrama de classes de análise, mostrado na seção 3.4.2. A seguir são apresentadas as características das classes presentes no diagrama ilustrado pela Figura 33:

a) `TGenericClass`: classe base responsável pelos métodos que exibem mensagens de *log* e mensagens de diálogo;
b) `TUserMessage`: classe responsável por manipular as listas de mensagens ao usuário;
c) `TAddinUtils`: possuindo apenas métodos estáticos, esta classe fornece um conjunto de funcionalidades requisitadas por várias classes do *plugin*;
d) `TDataAccess`: esta classe representa a camada de acesso a dados da aplicação, é acessível apenas por objetos de tela e elementos visuais, e também possui apenas métodos estáticos;
e) `TNamesControl`: esta classe é responsável por manter a unicidade de nomes de variáveis, tanto dentro do manipulador quanto dentro das telas.

O diagrama exibido pela Figura 33 apresenta ainda o pacote `TemplateAnalyser`, responsável pelas análises léxica, sintática e semântica do *template*, e o pacote `ClassMapperAnalyser`, responsável pelas análises léxica, sintática e semântica do arquivo de mapeamento de classes. Existe ainda neste diagrama o pacote de classes responsável pelas análises do campo n*otes* (`ElementNotesAnalyzer`), bem como o pacote de classes responsável pelas análises do atributo *ObjectStyle*, (`ObjectStyleAnalyzer`). Por fim há o pacote `TAClasses` que corresponde as classes utilizadas pelo *template* para estruturas de controle.

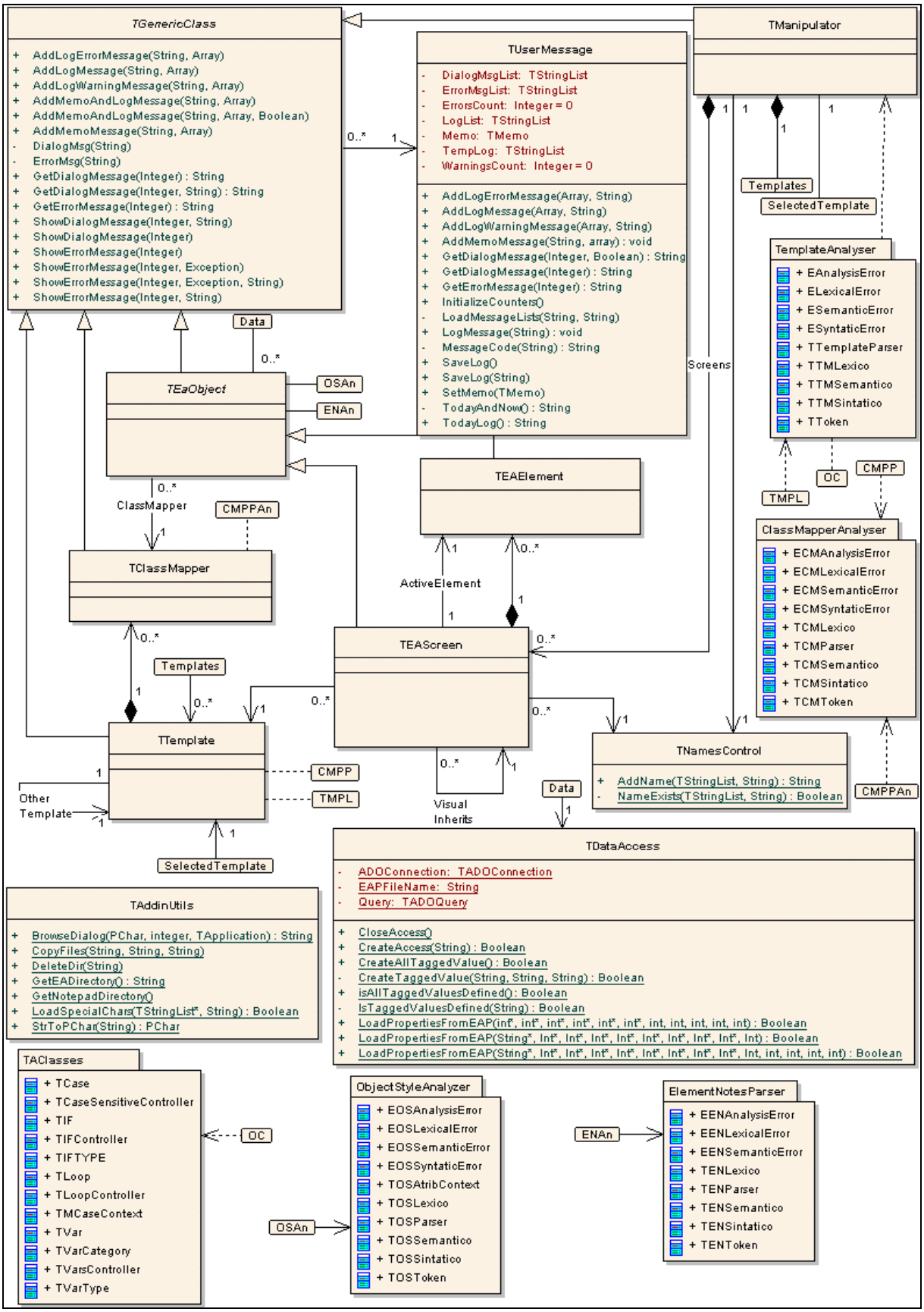


Figura 33 – Diagrama de classes de projeto

APÊNDICE B – Código fonte de tela em Java

O Quadro 36 apresenta resumidamente o código fonte da tela em Java exibida pela Figura 27. Pode-se notar que as cores foram geradas na forma de um cálculo, que representa os valores referentes às cores vermelho, verde e azul. Este ardil foi necessário porque o Java reconhece de maneira diferente o valor inteiro da cor, gerado-a de forma diferente da cor apresentada nos elementos do EA.

```
import javax.swing.*;
import java.awt.*;

public class CadPacientes2 extends JFrame {
  Container tela;
  public  CadPacientes2(){
    tela = getContentPane();
    tela.setLayout(null);
    setSize(376,238);
    super.setTitle("Cadastro de Pacientes");
    int R;
    int G;
    int B;
    B = (int)(13828095/(256*256));
    G = (int)((13828095-(int)(B * 256*256))/256);
    R = (int)(13828095-(int)(B * 256*256 + G * 256));
    tela.setBackground(new Color(R,G,B));
    B = (int)(666/(256*256));
    G = (int)((666-(int)(B * 256*256))/256);
    R = (int)(666-(int)(B * 256*256 + G * 256));
    tela.setForeground(new Color(R,G,B));
    tela.setFont(new Font("Arial Black",Font.PLAIN, 17));

    JPanel Panel = new JPanel();
    Panel.setBounds(39,140,273,35);
    B = (int)(0/(256*256));
    G = (int)((0-(int)(B * 256*256))/256);
    R = (int)(0-(int)(B * 256*256 + G * 256));
    Panel.setForeground(new Color(R,G,B));
    Panel.setFont(new Font("Arial",Font.PLAIN, 10));
    B = (int)(16777215/(256*256));
    G = (int)((16777215-(int)(B * 256*256))/256);
    R = (int)(16777215-(int)(B * 256*256 + G * 256));
    Panel.setBackground(new Color(R,G,B));
    tela.add(Panel);
    // demais componentes
  }
  public static void main(String args[]){
  Cadastro_de_Pacientes app = new Cadastro_de_Pacientes();
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}
```

Quadro 36 – Código fonte de tela em Java

APÊNDICE C – Código fonte de tela em Delphi (formulário dinâmico)

O Quadro 37 apresenta resumidamente o código fonte da tela em Delphi exibida pela Figura 28. O código representa uma classe que possui um atributo referente ao formulário (classe `TForm`) e outro atributo que serve de contêiner para os elementos visuais que compõe a tela.

```
constructor TScreen.Create(AOwner: TComponent);
begin
  Owner := AOwner;
  ComponentList := Tlist.Create;
  Screen := TForm.Create(AOwner);
end;

procedure TScreen.CreateComponents;
var c : TComponent;
begin
  Screen.Left := 50;
  Screen.Top := 182;
  Screen.Height := 238;
  Screen.Width := 376;
  Screen.Font.Charset := DEFAULT_CHARSET;
  Screen.Font.Color := 666;
  Screen.Font.Size := 17;
  Screen.Font.Name := 'Arial Black';
  Screen.Font.Style := [];
  Screen.Caption := 'Cadastro de Pacientes';
  Screen.Color := 13828095;
  Screen.ParentFont := False;

  c := TPanel.Create(Owner);
  ComponentList.add (c);
  TPanel(c).Parent := Screen;
  TPanel(c).Visible := True;
  TPanel(c).Top := 140;
  TPanel(c).Left := 39;
  TPanel(c).Height := 35;
  TPanel(c).Width := 273;
  TPanel(c).Font.Charset := DEFAULT_CHARSET;
  TPanel(c).Font.Color := 0;
  TPanel(c).Font.Size := 10;
  TPanel(c).Font.Name := 'Arial';
  TPanel(c).Font.Style := [];
  TPanel(c).ParentFont := False;
  TPanel(c).Caption := '';
  TPanel(c).Color := 16777215;
 // demais componentes
end;
```

Quadro 37 – Código fonte de tela em Delphi (formulário dinâmico)

APÊNDICE D – Código fonte de tela em Delphi (formulário estático)

O Quadro 38 apresenta resumidamente o código fonte do arquivo .DFM da tela exibida pela Figura 29. Nota-se que o atributo `Font.Height` recebem um valor padrão (-11), por conta da limitação citada na seção 3.5.3.2. Por fim, o Quadro 39 exibe na íntegra o código fonte do arquivo .PAS, referente a mesma tela.

```
object CadPacientes2: TCadPacientes2
  Left = 308
  Top = 232
  Width = 390
  Height = 246
  Caption = 'Cadastro de Pacientes'
  Color = 13828095
  Font.Charset = DEFAULT_CHARSET
  Font.Color = 666
  Font.Height = -11
  Font.Name = 'Arial Black'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 15
  object Codigo: TLabel
    Left = 18
    Top = 29
    Width = 34
    Height = 15
    Caption = 'Codigo'
    Color = 14215660
    Font.Charset = DEFAULT_CHARSET
    Font.Color = clBlack
    Font.Height = -12
    Font.Name = 'Times New Roman'
    Font.Style = [fsUnderline]
    ParentColor = False
    ParentFont = False
  end
  // demais componentes
end
```

Quadro 38 – Código fonte de tela em Delphi (arquivo .DFM)

```
unit uCadPacientes2;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Buttons, ExtCtrls,
  Mask, Grids, ComCtrls;
type

  TCadPacientes2 = class (TForm)
    Panel: TPanel;
    Codigo: TLabel;
    Nascimento: TLabel;
    Sexo: TLabel;
    Ano: TComboBox;
    Cancelar: TButton;
    Dia: TComboBox;
    Doador_de_orgaos_e_tecidos: TCheckBox;
    Feminino: TRadioButton;
    Masculino: TRadioButton;
    Mes: TComboBox;
    Salvar: TButton;
    texto: TEdit;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  CadPacientes2: TCadPacientes2;

implementation

{$R *.dfm}

end.
```

Quadro 39 – Código fonte de tela em Delphi (arquivo .PAS)

APÊNDICE E – Gramática da linguagem do *template*

O Quadro 40 mostra na íntegra a gramática da linguagem usada no *template*. Por questões de espaço foram suprimidas as definições de palavras reservadas e de expressões regulares.

```
<TEMPLATEDEC>              ::=    <OPENTEMPLATE> <DEFINITIONSDEC>
                                  <TEMPLATEBODY> <CLOSETEMPLATE>  #50000 ;

<DEFINITIONSDEC>    ::= <DEFINITIONS> <DEFINITIONSDEC> | î ;

<DEFINITIONS>       ::=    <TEMPLATENAMEOPT>   |      <TEMPLATELANGOPT>   |
                           <CLASSMAPPERDEC>    |      <TEMPLATEDESCOPT>   |
                           <TEMPLATEFILEEXTOPT>|      <OTHERTEMPLATE>     |
                           <SPECIALCHARSDEC>   ;

<TEMPLATENAMEOPT>   ::= NAME                  #1 "=" strings #0 ";" ;
<TEMPLATELANGOPT>   ::= LANGUAGE              #2 "=" strings #0 ";" ;
<CLASSMAPPERDEC>    ::= CLASSMAPPERFILE       #3 "=" strings #0 ";" ;
<TEMPLATEDESCOPT>   ::= TEMPLATEDESC          #4 "=" strings #0 ";" ;
<TEMPLATEFILEEXTOPT>::= TEMPLATEFILEEXT       #5 "=" strings #0 ";" ;
<SPECIALCHARSDEC>   ::= SPECIALCHARSFILE      #6 "=" strings #0 ";" ;
<OTHERTEMPLATE>     ::= OTHERTEMPLATEFILE     #7 "=" strings #0 ";" ;
<TEMPLATEBODY>      ::= CMDSTART <COMMANDDEC> ;

<STRINGS>           ::=    strings            #2000  ;
<INTEGER>           ::=    integer            #2001  ;
<FLOAT>             ::=    float              #2002  ;
<CHAR>              ::=    sustenido integer  #2003  ;
<USERVARS>          ::=    uservarsdec        #2004  ;

<BOOLVAL>           ::=    TRUE               #2005  |
                           FALSE              #2006  ;

<WRT>               ::=    WRT | WRITE ;

<WRTCOMMAND>        ::=    <WRT> #1013 "(" <WRTCMDLIST> ")" <PV> ;

<WRTCMDLIST>        ::=    <WRITECOMMANDS> <WRTCMDLIST> | î ;

<ATRIB>             ::=    <VARSATRIB> ":=" <WRTCMDLIST> <PV> ;

<VARSATRIB>         ::=    <USERVARS>                |
                           <SYSTEMVARS>              ;

<COMMANDDEC>        ::=    <COMMAND> <COMMANDDEC>    | î ;
<WRITECOMMANDS>     ::=    <CHAR > | <STRINGS>    | <INTEGER> |
                           <FLOAT> | <SYSTEMVARS> | <BOOLVAL> | <USERVARS>;
<CONTROLCOMMANDS>   ::=    <ALLCASES> | <LOOP> | <CIF> | <NLINE> ;

<COMMAND>           ::=    <ATRIB> | <WRTCOMMAND> | <CONTROLCOMMANDS>;
```

```
<ELEMENTSVARS>      ::= // VARIÁVEIS DE ELEMENTOS
             ELEMENTBACKCOLOR          #101 | ELEMENTBORDERCOLOR #102 |
             ELEMENTBORDERWIDTH        #103 | ELEMENTFONTBOLD    #104 |
             ELEMENTFONTITALIC         #105 | ELEMENTFONTCOLOR   #106 |
             ELEMENTFONTNAME           #107 | ELEMENTFONTSIZE    #108 |
             ELEMENTFONTUNDERLINED     #109 | ELEMENTNAME        #110 |
             ELEMENTRECTBOTTOM         #111 | ELEMENTRECTLEFT    #112 |
             ELEMENTRECTRIGHT          #113 | ELEMENTRECTTOP     #114 |
             ELEMENTSCOPE              #115 | ELEMENTSTEREOTYPE  #116 |
             ELEMENTTARGETLANGCLASS    #117 | ELEMENTTYPE        #118 |
             ELEMENTWIDTH              #119 | ELEMENTHEIGHT      #120 |
             ELEMENTVARNAME            #121 | ELEMENTGROUP       #122 |
             ELEMENTGUID               #123 | ELEMENTID          #124 |
             ELEMENTINHERITS           #125 ;

<SCREENVARS>        ::=    // VARIÁVEIS DE TELA            SCREENBACKCOLOR
             #201 | SCREENBORDERCOLOR  #202 |
             SCREENBORDERWIDTH         #203 | SCREENFONTBOLD     #204 |
             SCREENFONTITALIC          #205 | SCREENFONTCOLOR    #206 |
             SCREENFONTNAME            #207 | SCREENFONTSIZE     #208 |
             SCREENFONTUNDERLINED      #209 | SCREENNAME         #210 |
             SCREENRECTBOTTOM          #211 | SCREENRECTLEFT     #212 |
             SCREENRECTRIGHT           #213 | SCREENRECTTOP      #214 |
             SCREENSCOPE               #215 | SCREENSTEREOTYPE   #216 |
             SCREENTARGETLANGCLASS     #217 | SCREENTYPE         #218 |
             SCREENWIDTH               #219 | SCREENHEIGHT       #220 |
             SCREENVARNAME             #221 | SCREENGROUP        #222 |
             SCREENGUID                #223 | SCREENID           #224 |
             SCREENINHERITS            #225 ;

<EXCLUSIVESCREENVARS>     ::=  // Variáveis exclusivamente de TELA

             SOURCECODEFILENAME  #501 | TEMPLATEFILENAME          #502 |
             ISABSTRACT          #503 | PROPAGATE                 #504 |
             CLASSMAPPERFILENAME #505 | TEMPLATEDESCRIPTION       #506 |
             DEFAULTSCFILEEXT    #507 | TEMPLATELANGUAGE          #508 |
             TEMPLATENAME        #509 ;

<ISCREENVARS>       ::=    // VARIÁVEIS DE TELA (HERDADA)
             ISCREENBACKCOLOR          #301 | ISCREENBORDERCOLOR #302 |
             ISCREENBORDERWIDTH        #303 | ISCREENFONTBOLD    #304 |
             ISCREENFONTITALIC         #305 | ISCREENFONTCOLOR   #306 |
             ISCREENFONTNAME           #307 | ISCREENFONTSIZE    #308 |
             ISCREENFONTUNDERLINED     #309 | ISCREENNAME        #310 |
             ISCREENRECTBOTTOM         #311 | ISCREENRECTLEFT    #312 |
             ISCREENRECTRIGHT          #313 | ISCREENRECTTOP     #314 |
             ISCREENSCOPE              #315 | ISCREENSTEREOTYPE  #316 |
             ISCREENTARGETLANGCLASS    #317 | ISCREENTYPE        #318 |
             ISCREENWIDTH              #319 | ISCREENHEIGHT      #320 |
             ISCREENVARNAME            #321 | ISCREENGROUP       #322 |
             ISCREENGUID               #323 | ISCREENID          #324 |
             ISCREENINHERITS           #325 ;

<EXCLUSIVEISCREENVARS>    ::= // Variáveis exclusivamente de TELA (HERDADA)
             ISOURCECODEFILENAME       #601 | ITEMPLATEFILENAME  #602 |
             IISABSTRACT               #603 | IPROPAGATE         #604 |
             ICLASSMAPPERFILENAME      #605 | ITEMPLATEDESCRIPTION#606 |
             IDEFAULTSCFILEEXT         #607 | ITEMPLATELANGUAGE  #608 |
```

```
            ITEMPLATENAME             #609 ;

<MANIPULATORVARS>   ::=    // Variáveis do MANIPULADOR
            EAPFILENAME  #701 | HELPDIR      #702 | MSGSDIR      #703 |
            OUTPUTDIR    #704 | PLUGINDIR    #705 | TEMPLATESDIR #706 |
            LOGSDIR      #707 ;

<SYSTEMVARS>        ::=
            <ELEMENTSVARS>                   | <SCREENVARS>      |
            <EXCLUSIVESCREENVARS>            | <ISCREENVARS>     |
            <EXCLUSIVEISCREENVARS>           | <MANIPULATORVARS> ;

<LOOP>              ::= LOOP  #1004 <COMMANDLOOPDEC> CLOSELOOP #1005 ;

<COMMANDLOOPDEC>    ::= <COMMAND> <COMMANDDEC> | î ;

<CIF>               ::= IF #1000 <CONDITION> THEN <IFCOMMANDDEC>
<ELSEOPTIONAL> ;

<IFCOMMANDDEC>      ::= "{" #1002 <COMMANDDEC> "}" #1003 ;

<CONDITION>         ::= "(" <COMPARACAO> ")";

<COMPARACAO>        ::= <COMPARAVEL> <OPERADOR> <COMPARAVEL> ;

<COMPARAVEL>        ::= <WRITECOMMANDS> ;

<OPERADOR>          ::=    "=" #1101    | "<>" #1102 | ">" #1103  |
                           "<" #1104    | "<=" #1105 | ">=" #1106;

<ELSEOPTIONAL>      ::=   <CELSE> | î ;

<CELSE>             ::= ELSE #1001 <IFCOMMANDDEC> ;

<NLINE>             ::= "<" NL #980 ">" ;

<PV>                ::= ";" #981 ;

<OPENTEMPLATE>      ::= "[" TEMPLATE "]";

<CLOSETEMPLATE>     ::= "[/" TEMPLATE "]";

<ALLCASES>          ::= <OPENLOWERCASE> | <OPENUPPERCASE> | <CLOSECASE> ;

<OPENUPPERCASE>     ::= UCASE #1010 | UC #1010 ;

<OPENLOWERCASE>           ::= LCASE #1011 | LC #1010;

<CLOSECASE>         ::= CLOSECASE #1012;
```

Quadro 40 – Gramática da linguagem do *template*

APÊNDICE F – Ações semânticas da gramática do *template*

O Quadro 41 apresenta a relação das ações semânticas implementadas para a análise semântica do *template*.

Código	Token	Ação semântica
0	strings	Usado apenas no cabeçalho do *template*.
1	NAME	Usado para definir o nome do *template*.
2	LANGUAGE	Usado para definir a linguagem de programação para qual o *template* foi construído.
3	CLASSMAPPERFILE	Usado para definir um arquivo no qual se encontram as definições de mapeamento de classes.
4	TEMPLATEDESC	Usado para definir uma descrição sumária para o *template*.
5	TEMPLATEFILEEXT	Usado para definir uma extensão padrão para os arquivos que serão gerados.
6	SPECIALCHARSFILE	Usado para definir o arquivo de caracteres especiais que será usado nas substituições de caracteres inválidos.
7	OTHERTEMPLATEFILE	Usado para definir um arquivo de *template* a ser executado depois da execução do *template* atual.
101	@BACKCOLOR	Cor de fundo do **elemento**.
102	@BORDERCOLOR	Cor da borda do **elemento**.
103	@BORDERWIDTH	Espessura da borda do **elemento**.
104	@FONTBOLD	Indica se a fonte do **elemento** está em negrito.
105	@FONTITALIC	Indica se a fonte do **elemento** está em itálico.
106	@FONTCOLOR	Cor da fonte do **elemento**.
107	@FONTNAME	Nome da fonte do **elemento**.
108	@FONTSIZE	Tamanho da fonte do **elemento**.
109	@FONTUNDERLINED	Indica se a fonte do **elemento** está sublinhada.
110	@NAME	Nome do **elemento**
111	@RECTBOTTON	Posição da extremidade inferior do **elemento**.
112	@RECTLEFT	Posição da extremidade esquerda do **elemento**.
113	@RECTRIGHT	Posição da extremidade direita do **elemento**.
114	@RECTTOP	Posição da extremidade superior do **elemento**.
115	@SCOPE	Escopo do **elemento**.
116	@STEREOTYPE	Estereótipo do **elemento**
117	@TARGETLANGCLASS	Classe da linguagem alvo que o **elemento** representa.
118	@TYPE	Tipo do **elemento**.
119	@WIDTH	Largura do **elemento**.
120	@HEIGHT	Altura do **elemento**.
121	@VARNAME	Nome de variável do **elemento**.
122	@GROUP	Grupo da classe do **elemento**.
123	@GUID	Identificador global único do **elemento**.

124	@ID	Identificador do elemento do EA
125	@INHERITS	Indica se o **elemento** herda características de sua tela.
201	@BACKCOLOR_	Cor de fundo da **tela**.
202	@BORDERCOLOR_	Cor da borda da **tela**.
203	@BORDERWIDTH_	Espessura da borda da **tela**.
204	@FONTBOLD_	Indica se a fonte da **tela** está em negrito.
205	@FONTITALIC_	Indica se a fonte da **tela** está em itálico.
206	@FONTCOLOR_	Cor da fonte da **tela**.
207	@FONTNAME_	Nome da fonte da **tela**.
208	@FONTSIZE_	Tamanho da fonte da **tela**.
209	@FONTUNDERLINED_	Indica se a fonte da **tela** está sublinhada.
210	@NAME_	Nome da **tela**
211	@RECTBOTTON_	Posição da extremidade inferior da **tela**.
212	@RECTLEFT_	Posição da extremidade esquerda da **tela**.
213	@RECTRIGHT_	Posição da extremidade direita da **tela**.
214	@RECTTOP_	Posição da extremidade superior da **tela**.
215	@SCOPE_	Escopo da **tela**.
216	@STEREOTYPE_	Estereótipo da **tela**
217	@TARGETLANGCLASS_	Classe da linguagem alvo que a **tela** representa.
218	@TYPE_	Tipo da **tela**.
219	@WIDTH_	Largura da **tela**.
220	@HEIGHT_	Altura da **tela**.
221	@VARNAME_	Nome de variável da **tela**.
222	@GROUP_	Grupo da classe da **tela**.
223	@GUID_	Identificador global único da **tela**.
224	@ID_	Identificador da **tela** no EA
225	@INHERITS_	Indica se a **tela** herda características de outra tela.
301	@BACKCOLOR#	Cor de fundo da **tela herdada**.
302	@BORDERCOLOR#	Cor da borda da **tela herdada**.
303	@BORDERWIDTH#	Espessura da borda da **tela herdada**.
304	@FONTBOLD#	Indica se a fonte da **tela herdada** está em negrito.
305	@FONTITALIC#	Indica se a fonte da **tela herdada** está em itálico.
306	@FONTCOLOR#	Cor da fonte da **tela herdada**.
307	@FONTNAME#	Nome da fonte da **tela herdada**.
308	@FONTSIZE#	Tamanho da fonte da **tela herdada**.
309	@FONTUNDERLINED#	Indica se a fonte da **tela herdada** está sublinhada.
310	@NAME#	Nome da **tela herdada**
311	@RECTBOTTON#	Posição da extremidade inferior da **tela herdada**.
312	@RECTLEFT#	Posição da extremidade esquerda da **tela herdada**.
313	@RECTRIGHT#	Posição da extremidade direita da **tela herdada**.
314	@RECTTOP#	Posição da extremidade superior da **tela herdada**.
315	@SCOPE#	Escopo da **tela herdada**.
316	@STEREOTYPE#	Estereótipo da **tela herdada**
317	@TARGETLANGCLASS#	Classe da linguagem alvo que a **tela herdada** representa.
318	@TYPE#	Tipo da **tela herdada**.
319	@WIDTH#	Largura da **tela herdada**.
320	@HEIGHT#	Altura da **tela herdada**.

321	@VARNAME#	Nome de variável da **tela herdada**.
322	@GROUP#	Grupo da classe da **tela herdada**.
323	@GUID#	Identificador global único da **tela herdada**.
324	@ID#	Identificador da **tela herdada** no EA
325	@INHERITS#	Indica se a **tela herdada** herda características de outra tela herdada.
501	@SOURCECODEFILENAME_	Nome do arquivo de saída do código fonte gerado para a **tela**.
502	@TEMPLATEFILENAME_	Nome do arquivo do *template* relacionado à **tela**.
503	@ISABSTRACT_	Indica se a **tela** é abstrata.
504	@PROPAGATE_	Indica se a **tela** propaga suas propriedades para seus elementos.
505	@CLASSMAPPERFILENAME_	Nome do arquivo de mapeamento de classe associado ao *template* que, por sua vez, está associado à **tela**.
506	@TEMPLATEDESCRIPTION_	Descrição do *template* relacionado à **tela**.
507	@DEFAULTSCFILEEXT_	Extensão (definida no *template* relacionado à **tela**) que será atribuída aos arquivos de saída do código fonte.
508	@TEMPLATELANGUAGE_	Nome da linguagem de programação para qual o *template* relacionado à **tela** foi definido.
509	@TEMPLATENAME_	Nome do arquivo do *template* relacionado à **tela**.
601	@SOURCECODEFILENAME#	Nome do arquivo de saída do código fonte gerado para a **tela herdada**.
602	@TEMPLATEFILENAME#	Nome do arquivo do *template* relacionado à **tela herdada**.
603	@ISABSTRACT#	Indica se a **tela herdada** é abstrata.
604	@PROPAGATE#	Indica se a **tela herdada** propaga suas propriedades para seus elementos.
605	@CLASSMAPPERFILENAME#	Nome do arquivo de mapeamento de classe associado ao *template* que, por sua vez, está associado à **tela herdada**.
606	@TEMPLATEDESCRIPTION#	Descrição do *template* relacionado à **tela herdada**.
607	@DEFAULTSCFILEEXT#	Extensão (definida no *template* relacionado à **tela herdada**) que será atribuída aos arquivos de saída do código fonte.
608	@TEMPLATELANGUAGE#	Nome da linguagem de programação para qual o *template* relacionado à **tela herdada** foi definido.
609	@TEMPLATENAME#	Nome do arquivo do *template* relacionado à **tela herdada**.
701	@EAPFILENAME	Nome do arquivo EAP relacionado ao projeto.
702	@HELPDIR	Diretório onde está armazenado o arquivo de HELP.
703	@MSGSDIR	Diretório onde estão armazenados os arquivos de mensagem.
704	@OUTPUTDIR	Diretório onde será gerados os códigos fonte.
705	@PLUGINDIR	Diretório onde está armazenada a DLL do plugin.
706	@TEMPLATESDIR	Diretório onde estão armazenados os *templates*.

707	@LOGSDIR	Diretório onde serão gerados os logs do plugin.
980	<NL>	Quebra de linha
981	;	Reconhece “ponto e vírgula”
1000	IF	Indica o reconhecimento de uma estrutura de contronle IF.
1001	ELSE	Indica o reconhecimento de uma estrutura de contronle ELSE.
1002	{	Indica o início de um bloco de comandos de um IF ou de um ELSE.
1003	}	Indica o fim de um bloco de comandos de um IF ou de um ELSE.
1004	LOOP	Indica o início de um looping
1005	CLOSELOOP	Indica o fim de um looping
1010	UCASE ou UC	Início de um bloco que deverá ser escrito em letras maiúsculas.
1011	LCASE ou LC	Início de um bloco que deverá ser escrito em letras minúsculas.
1012	CLOSECASE	Fim do último bloco definido de uma das duas maneiras acima.
1013	WRT ou WRITE	Comando para escrever.
1101	=	Operador IGUAL.
1102	<>	Operador DIFERENTE.
1103	>	Operador MAIOR.
1104	<	Operador MENOR.
1105	<=	Operador MENOR OU IGUAL.
1106	>=	Operador MAIOR OU IGUAL.
2000	Strings	Usado apenas para fazer comparações em IF ou para escrita no *template*.
2001	Integer	Usado apenas para fazer comparações em IF.
2002	Float	Usado apenas para fazer comparações em IF.
2003	<CHAR>	Usado apenas para fazer comparações em IF ou para escrita no *template*. Formado pelo caractere sustenido seguido de um número inteiro positivo. Retorna o caractere respectivo da tabela ASC.
2004	<USERVARS>	Reconhece uma variável definida pelo usuário
2005	TRUE	Valor boolean **true**.
2006	FALSE	Valor boolean **false**.
50000	[/TEMPLATE]	Final do *template*.

Quadro 41 – Ações semânticas do *template*

APÊNDICE G – Gramática e semântica da linguagem de mapeamento de classes

O Quadro 42 exibe a gramática da linguagem usada para definição do arquivo de mapeamento de classes.

```
// definições regulares

digito     : [0-9]

minuscula : [a-z]

maiuscula : [A-Z]

underline : [_]

letra  : {maiuscula}|{minuscula}|{underline}

WS      : [\s\t\n\r\b]

string : "'" [^"'"]* "'"

// palavras reservadas

CLASSMAPPER         : {C}{L}{A}{S}{S}{M}{A}{P}{P}{E}{R}

PROPERTYMAPPER      : {P}{R}{O}{P}{E}{R}{T}{Y}{M}{A}{P}{P}{E}{R}

EXTERNALFILE : {E}{X}{T}{E}{R}{N}{A}{L}{F}{I}{L}{E}

GROUP               : {G}{R}{O}{U}{P}

// símbolos especiais

"["

"]"

"[/"

"="

","

";"

// outras definições

numeros             : {digito}+
strings             : {string}
: {WS}*


<CLASSMAPPERDEC>    ::=    "[" CLASSMAPPER "]" <CLASSMAPPERLIST>
                           "[/" CLASSMAPPER "]" ;

<GROUPDEC>          ::= GROUP "=" <NUMLIST> ";" #13;

<NUM>               ::= numeros #12;

<NUMLIST>           ::= <NUM> <MORENUMLIST> ;

<MORENUMLIST>       ::= <MORENUM> <MORENUMLIST> | î ;
```

```
<MORENUM>           ::= "," <NUM> ;

<CLASSMAPPERLIST>   ::=    <EACLASS> #10 "=" <TARGETCLASS> #11
                           <GROUPDEC> <CLASSMAPPERLIST> | î ;

<EACLASS>           ::= strings ;

<TARGETCLASS>       ::= strings ;
```

Quadro 42 – Gramática da linguagem do arquivo de mapeamento de classes

O Quadro 43 apresenta os códigos das ações semânticas definidas para a análise do arquivo de mapeamento de classes.

Código	**Token**	**Ação semântica**
10	<EACLASS>	Reconhece uma string que representa o estereótipo do elemento no EA.
11	<TARGETCLASS>	Também reconhece uma string, esta indicando qual classe (na linguagem alvo) representa o estereótipo do elemento.
12	numeros	Reconhece valores numéricos.
13	;	Reconhece o caractere ponto e vírgula, que indica o fim de uma linha de definição de mapeamento de classe.

Quadro 43 – Ações semânticas da linguagem do arquivo de mapeamento de classes

APÊNDICE H – Gramática do analisador do campo *Notes*

O Quadro 44 exibe a gramática que deu origem aos analisadores do campo *Notes* dos elementos visuais e das telas.

```
// definições regulares

digito     : [0-9]

minuscula : [a-z]

maiuscula : [A-Z]

underline : [_]

letra  : {maiuscula}|{minuscula}|{underline}

string : [^"=" "'" "@@" \n \s ;]*

strfilename : "'" [^"'"]* "'"

pv     : [;]

WS      : [\s\r\t\b]

nl     : [\n]

// símbolos especiais

";"

"="

// outras definições

integer             : {digito}+
strings             : {string}
strfile             : {strfilename}
pvs                 : {pv}
enter               : {nl}

: {WS}*

// Gramática

<ATRIBLIST>         ::=    <ATRIB> <ATRIBLIST> | î ;

<ATRIB>             ::=     strings #100 | strfile #101 |
                    integer #102| "@" #200  | "=" #201 |
                           pvs #202 | enter #203;
```

Quadro 44 – Gramática dos analisadores do campo *Notes*

APÊNDICE I – Gramática do analisador do campo *Objectstyle*

O Quadro 45 exibe a gramática que deu origem aos analisadores do campo *Objectstyle* dos elementos visuais e das telas, de onde são extraídos os valores dos atributos de estilos de fonte.

```
// definições regulares

digito     : [0-9]

espaco : [\s]

minuscula : [a-z]

maiuscula : [A-Z]

underline : [_]

letra  : {maiuscula}|{minuscula}|{underline}

varval : {letra} | {digito} | {espaco}

WS      : [\t\n\r\b]

// símbolos especiais

";"

"="

// outras definições

varname             : {letra}+
varvalue            : {varval}*

: {WS}*

// Gramática

<DECLIST>           ::= <DEC> <DECLIST> | î ;

<DEC>               ::= <VNAME> #1 "=" <VVALUE> #2 ";" ;

<VNAME>             ::= varname;

<VVALUE>            ::= varname | varvalue | î ;
```

Quadro 45 – Gramática dos analisadores do campo *Objectstyle*

APÊNDICE J – *Templates* para geração de telas para Visual Basic

O Quadro 46 mostra o *template* para geração do arquivo de extensão ".Designer.vb" e, em seguida, o Quadro 47 apresenta o *template* responsável pela geração do arquivo de extensão ".vb".

```
[TEMPLATE]

TEMPLATEFILEEXT = '.vb';

NAME = 'VB Project';

LANGUAGE = 'Visual Basic';

CLASSMAPPERFILE = 'VB.cmpp';

OTHERTEMPLATEFILE = 'C:\Arquivos de programas\Sparx
Systems\EA\GUICGAddin\Templates\VBForms.tmpl';

SPECIALCHARSFILE = 'VB.sps';

TEMPLATEDESC = 'Este template faz a geração do fonte para o arquivo .vb';

start

wrt ( 'Public Class ' @VARNAME_); <nl><nl>
wrt ('    Public Sub New ()' ); <nl><nl>

wrt ('        ' #39 ' This call is required by the Windows Form Designer.');
<nl><nl>
wrt ('        InitializeComponent()'); <nl><nl>
wrt ('    End Sub'); <nl><nl>

wrt ('    Protected Overrides Sub Finalize()'); <nl>
wrt ('        MyBase.Finalize()'); <nl>
wrt ('    End Sub'); <nl>
wrt ('End Class');

[/TEMPLATE]
```

Quadro 46 – *Template* do arquivo ".Designer.vb"

```
[TEMPLATE]

TEMPLATEFILEEXT = '.vb';

NAME = 'VB Forms';

LANGUAGE = 'Visual Basic';

CLASSMAPPERFILE = 'VB.cmpp';

SPECIALCHARSFILE = 'VB.sps';

TEMPLATEDESC = 'Este template faz a geração do fonte para o arquivo .vb';

start

wrt ('<Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerated()>
_'); <nl>
wrt ('Partial Class ' @VARNAME_); <nl>
wrt ('    Inherits System.Windows.Forms.' @TARGETLANGCLASS_); <nl><nl>
```

```
wrt ('    ' #39 'Form overrides dispose to clean up the component list.');
<nl>
wrt ('    <System.Diagnostics.DebuggerNonUserCode()> _'); <nl>
wrt ('    Protected Overrides Sub Dispose(ByVal disposing As Boolean)');
<nl>
wrt ('        If disposing AndAlso components IsNot Nothing Then'); <nl>
wrt ('            components.Dispose()'); <nl>
wrt ('        End If'); <nl>
wrt ('        MyBase.Dispose(disposing)'); <nl>
wrt ('    End Sub'); <nl><nl>

wrt ('    ' #39 'Required by the Windows Form Designer'); <nl>
wrt ('    Private components As System.ComponentModel.IContainer'); <nl><nl>
wrt('       ' #39 '////// Color Variables //////'); <nl>
wrt ('    Private R As Integer'); <nl>
wrt ('    Private G As Integer'); <nl>
wrt ('    Private B As Integer'); <nl><nl><nl>


wrt ('    ' #39 'NOTE: The following procedure is required by the Windows
Form Designer'); <nl>
wrt ('    ' #39 'It can be modified using the Windows Form Designer.'); <nl>
wrt ('    ' #39 'Do not modify it using the code editor.'); <nl>
wrt ('    <System.Diagnostics.DebuggerStepThrough()> _'); <nl>
wrt ('    Private Sub InitializeComponent()'); <nl>

LOOP
  wrt ('        Me.' @VARNAME ' =  new System.Windows.Forms.'
@TARGETLANGCLASS); <nl>
CLOSELOOP

wrt ('        Me.SuspendLayout()'); <nl>

LOOP
  if (@GROUP = 103)  then {
    wrt ('        Me.' @VARNAME '.SuspendLayout()'); <nl><nl><nl>
  }
CLOSELOOP

LOOP
  wrt ('        ' #39); <nl>
  wrt ('        ' #39 @VARNAME); <nl>
  wrt ('        ' #39); <nl>


  &Style := '';

  if (@FONTBOLD = true) then {
    &Style := '(System.Drawing.FontStyle.Bold)';
  }

  if (@FONTITALIC = true) then {
    if (&Style = '') then {
      &Style := '(System.Drawing.FontStyle.Italic)';
    } else {
      &Style := &Style ' Or (System.Drawing.FontStyle.Italic)';
    }
  }

  if (@FONTUNDERLINED = true) then {
    if (&Style = '') then {
      &Style := '(System.Drawing.FontStyle.Underline)';
    } else {
      &Style := &Style ' Or (System.Drawing.FontStyle.Underline)';
    }
  }

  if (&Style = '') then {
```

```
    &Style := '(System.Drawing.FontStyle.Regular)';
  }


  if (@GROUP = 1)  then {
    wrt ('        Me.' @VARNAME '.AutoSize = True'); <nl>
  }

  if (@GROUP = 0) then {
    wrt ('        B = CType(Math.Truncate((' @BACKCOLOR '/(256*256))),
Integer)'); <nl>
    wrt ('        G = CType(Math.Truncate(((' @BACKCOLOR ' - (B * 256 *
256))/256)), Integer)'); <nl>
    wrt ('        R = CType(Math.Truncate((' @BACKCOLOR ' - ((B * 256 * 256)
+ (G * 256)))), Integer)'); <nl>
    wrt ('        Me.' @VARNAME '.BackColor =
System.Drawing.Color.FromArgb(R,G,B)' ); <nl>

    wrt ('        B = CType(Math.Truncate((' @FONTCOLOR '/(256*256))),
Integer)'); <nl>
    wrt ('        G = CType(Math.Truncate(((' @FONTCOLOR ' - (B * 256 *
256))/256)), Integer)'); <nl>
    wrt ('        R = CType(Math.Truncate((' @FONTCOLOR ' - ((B * 256 * 256)
+ (G * 256)))), Integer)'); <nl>
    wrt ('        Me.' @VARNAME '.ForeColor =
System.Drawing.Color.FromArgb(R,G,B)' ); <nl>

    wrt ('        Me.' @VARNAME '.Location = new System.Drawing.Point('
@RECTLEFT ',' @RECTTOP ')'); <nl>
    wrt ('        Me.' @VARNAME '.Font = new System.Drawing.Font("'
@FONTNAME '", ' @FONTSIZE ', CType ((' &Style '),System.Drawing.FontStyle),
System.Drawing.GraphicsUnit.Point, CType(0, Byte))' ); <nl>
    wrt ('        Me.' @VARNAME '.Name = "' @VARNAME '"'); <nl>
    wrt ('        Me.' @VARNAME '.Size = new System.Drawing.Size(' @WIDTH
',' @HEIGHT ')'); <nl>
    wrt ('        Me.' @VARNAME '.Text = "' @NAME '"'); <nl>
  }

  if (@GROUP = 2)  then {
    wrt ('        Me.' @VARNAME '.UseVisualStyleBackColor = False'); <nl>
  }

  if (@GROUP = 4)  then {
    wrt ('        Me.' @VARNAME '.TabStop = True'); <nl>
  }

CLOSELOOP

wrt('        ' #39); <nl>
wrt('        ' #39 @varname_); <nl>
wrt('        ' #39); <nl>

&Style := '';

if (@FONTBOLD_ = true) then {
  &Style := '(System.Drawing.FontStyle.Bold)';
}

if (@FONTITALIC_ = true) then {
  if (&Style = '') then {
    &Style := '(System.Drawing.FontStyle.Italic)';
  } else {
    &Style := &Style ' Or (System.Drawing.FontStyle.Italic)';
  }
}

if (@FONTUNDERLINED_ = true) then {
  if (&Style = '') then {
```

```
    &Style := '(System.Drawing.FontStyle.Underline)';
  } else {
    &Style := &Style ' Or (System.Drawing.FontStyle.Underline)';
  }
}

if (&Style = '') then {
  &Style := '(System.Drawing.FontStyle.Regular)';
}

wrt ('        B = CType(Math.Truncate((' @BACKCOLOR_ '/(256*256))),
Integer)'); <nl>
wrt ('        G = CType(Math.Truncate(((' @BACKCOLOR_ ' - (B * 256 *
256))/256)), Integer)'); <nl>
wrt ('        R = CType(Math.Truncate((' @BACKCOLOR_ ' - ((B * 256 * 256) +
(G * 256)))), Integer)'); <nl>
wrt ('        Me.BackColor = System.Drawing.Color.FromArgb(R,G,B)' ); <nl>

wrt ('        B = CType(Math.Truncate((' @FONTCOLOR_ '/(256*256))),
Integer)'); <nl>
wrt ('        G = CType(Math.Truncate(((' @FONTCOLOR_ ' - (B * 256 *
256))/256)), Integer)'); <nl>
wrt ('        R = CType(Math.Truncate((' @FONTCOLOR_ ' - ((B * 256 * 256) +
(G * 256)))), Integer)'); <nl>
wrt ('        Me.ForeColor = System.Drawing.Color.FromArgb(R,G,B)' ); <nl>

wrt ('        Me.AutoScaleDimensions = New System.Drawing.SizeF(6.0!,
13.0!)'); <nl>
wrt ('        Me.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font');
<nl>
wrt ('        Me.ClientSize = new System.Drawing.Size(' @WIDTH_ ',' @HEIGHT_
')'); <nl>
wrt ('        Me.Font = new System.Drawing.Font("' @FONTNAME_ '", '
@FONTSIZE_ ', CType ((' &Style '),System.Drawing.FontStyle),
System.Drawing.GraphicsUnit.Point, CType(0, Byte))' ); <nl>
wrt ('        Me.Name = "' @VARNAME_); <nl>
wrt ('        Me.Text = "' @NAME_); <nl>

LOOP
  wrt ('        Me.Controls.Add(Me.' @VARNAME ')'); <nl>
CLOSELOOP

LOOP
  if (@GROUP = 103)  then {
    wrt ('        Me.' @VARNAME '.ResumeLayout(False)'); <nl><nl><nl>
  }
CLOSELOOP

wrt ('        Me.ResumeLayout(False)'); <nl>
wrt ('        Me.PerformLayout()'); <nl><nl>
wrt ('    End Sub'); <nl>

LOOP
  wrt ('    Friend WithEvents ' @VARNAME ' As System.Windows.Forms.'
@TARGETLANGCLASS); <nl>
CLOSELOOP

<nl>

wrt ('End Class    '); <nl>

@SOURCECODEFILENAME_ := @VARNAME_ '.Designer' @DEFAULTSCFILEEXT_;

[/TEMPLATE]
```

Quadro 47 – *Template* do arquivo ".vb"

APÊNDICE K – *Templates* para geração de telas para C#

O Quadro 48 mostra o *template* para geração do arquivo de extensão ".Designer.cs" e, em seguida, o Quadro 49 apresenta o *template* responsável pela geração do arquivo de extensão ".cs".

```
[TEMPLATE]

TEMPLATEFILEEXT = '.cs';

NAME = 'C# Project';

LANGUAGE = 'C#';

CLASSMAPPERFILE = 'C#.cmpp';

SPECIALCHARSFILE = 'C#.sps';

OTHERTEMPLATEFILE = 'C:\Arquivos de programas\Sparx
Systems\EA\GUICGAddin\Templates\C#Forms.tmpl';

TEMPLATEDESC = 'Este template faz a geração do fonte para o arquivo .cs';

start

wrt ('using System;'); <nl>
wrt ('using System.Collections.Generic;'); <nl>
wrt ('using System.ComponentModel;'); <nl>
wrt ('using System.Data;'); <nl>
wrt ('using System.Drawing;'); <nl>
wrt ('using System.Text;'); <nl>
wrt ('using System.Windows.Forms;'); <nl><nl>

wrt ('namespace ProgramaExemplo'); <nl>
wrt ('{'); <nl>
wrt ('    public partial class ' @VARNAME_ ' : ' @TARGETLANGCLASS_); <nl>
wrt ('    {'); <nl>
wrt ('        public ' @VARNAME_ '()'); <nl>
wrt ('        {'); <nl>
wrt ('            InitializeComponent();'); <nl>
wrt ('        }'); <nl>
wrt ('    }'); <nl>
wrt ('}'); <nl>

[/TEMPLATE]
```

Quadro 48 – *Template* do arquivo ".Designer.cs"

```
[TEMPLATE]

TEMPLATEFILEEXT = '.cs';

NAME = 'C# Forms';

LANGUAGE = 'C#';

CLASSMAPPERFILE = 'C#.cmpp';

SPECIALCHARSFILE = 'C#.sps';

TEMPLATEDESC = 'Este template faz a geração do fonte para o arquivo
.Designer.cs';
```

```
start

wrt ('namespace ProgramaExemplo'); <nl>
wrt ('{'); <nl>

wrt ('   partial class ' @VARNAME_); <nl>
wrt ('   {'); <nl>

wrt ('      private System.ComponentModel.IContainer components = null;');
<nl><nl>
wrt ('      protected override void Dispose(bool disposing)'); <nl>
wrt ('      {'); <nl>
wrt ('         if (disposing && (components != null))'); <nl>
wrt ('         {'); <nl>
wrt ('            components.Dispose();'); <nl>
wrt ('         }'); <nl>
wrt ('         base.Dispose(disposing);'); <nl>
wrt ('      }'); <nl><nl>

wrt ('      #region Windows Form Designer generated code'); <nl>
wrt ('      '); <nl>

&Prefix := 'System.Windows.Forms.';

wrt ('      private void InitializeComponent()'); <nl>
wrt ('      {'); <nl>

LOOP
  wrt ('        this.' @VARNAME ' = new ' &Prefix @TARGETLANGCLASS '();');
<nl>
CLOSELOOP
wrt ('        this.SuspendLayout();'); <nl>

<nl><nl>
wrt('        ////// Color Variables //////'); <nl><nl>

wrt('        byte R;'); <nl>
wrt('        byte G;'); <nl>
wrt('        byte B;'); <nl><nl>

LOOP
  &Style := '';
  &StyleX := '(System.Drawing.FontStyle)(';

  if (@FONTBOLD = true) then {
    &Style := &StyleX 'System.Drawing.FontStyle.Bold';
  }

  if (@FONTITALIC = true) then {
    if (&Style = '') then {
      &Style := &StyleX 'System.Drawing.FontStyle.Italic';
    } else {
      &Style := &Style ' | System.Drawing.FontStyle.Italic';
    }
  }

  if (@FONTUNDERLINED = true) then {
    if (&Style = '') then {
      &Style := &StyleX 'System.Drawing.FontStyle.Underline';
    } else {
      &Style := &Style ' | System.Drawing.FontStyle.Underline';
    }
  }

  if (&Style = '') then {
    &Style := &StyleX 'System.Drawing.FontStyle.Regular';
  }
```

```
  &Style := &Style ')';


  wrt('        //'); <nl>
  wrt('        // ' @varname); <nl>
  wrt('        //'); <nl>

  if (@GROUP = 1) then {
     wrt ('        this.' @VARNAME '.AutoSize = true;'); <nl>
  }

  if (@GROUP = 0) then {
    wrt ('        B = (byte)(int)( ' @BACKCOLOR '/(256*256));'); <nl>
    wrt ('        G = (byte)(int)((' @BACKCOLOR '-(int)(B *
256*256))/256);'); <nl>
    wrt ('        R = (byte)(int)( ' @BACKCOLOR '-(int)(B * 256*256 + G *
256));'); <nl>
    wrt ('        this.' @VARNAME '.BackColor =
System.Drawing.Color.FromArgb(R,G,B);' ); <nl>

    wrt ('        B = (byte)(int)( ' @FONTCOLOR '/(256*256));'); <nl>
    wrt ('        G = (byte)(int)((' @FONTCOLOR '-(int)(B *
256*256))/256);'); <nl>
    wrt ('        R = (byte)(int)( ' @FONTCOLOR '-(int)(B * 256*256 + G *
256));'); <nl>
    wrt ('        this.' @VARNAME '.ForeColor =
System.Drawing.Color.FromArgb(R,G,B);' ); <nl>

    wrt ('        this.' @VARNAME '.Location = new System.Drawing.Point('
@RECTLEFT ',' @RECTTOP ');'); <nl>
    wrt ('        this.' @VARNAME '.Font = new System.Drawing.Font("'
@FONTNAME '", ' @FONTSIZE ',' &Style ', System.Drawing.GraphicsUnit.Point,
((byte)(0)));' ); <nl>
    wrt ('        this.' @VARNAME '.Name = "' @VARNAME '";'); <nl>
    wrt ('        this.' @VARNAME '.Size = new System.Drawing.Size(' @WIDTH
',' @HEIGHT ');'); <nl>
    wrt ('        this.' @VARNAME '.Text = "' @NAME '";'); <nl>
  }

  if (@GROUP = 2) then {
     wrt ('        this.' @VARNAME '.UseVisualStyleBackColor = false;');
<nl>
  }

  if (@GROUP = 3) then {
     wrt ('        this.' @VARNAME '.CheckAlign =
System.Drawing.ContentAlignment.MiddleRight;'); <nl>
  }

  if (@GROUP = 4) then {
     wrt ('        this.' @VARNAME '.TabStop = true;'); <nl>
  }

  <nl><nl>
CLOSELOOP

wrt('        //'); <nl>
wrt('        // ' @varname_); <nl>
wrt('        //'); <nl>

if (@FONTBOLD = true) then {
  &Style := &StyleX 'System.Drawing.FontStyle.Bold';
}

if (@FONTITALIC = true) then {
  if (&Style = '') then {
    &Style := &StyleX 'System.Drawing.FontStyle.Italic';
```

```
  } else {
    &Style := &Style ' | System.Drawing.FontStyle.Italic';
  }
}

if (@FONTUNDERLINED = true) then {
  if (&Style = '') then {
    &Style := &StyleX 'System.Drawing.FontStyle.Underline';
  } else {
    &Style := &Style ' | System.Drawing.FontStyle.Underline';
  }
}

if (&Style = '') then {
  &Style := &StyleX 'System.Drawing.FontStyle.Regular';
}

wrt('        B = (byte)(int)( ' @BACKCOLOR_ '/(256*256));'); <nl>
wrt('        G = (byte)(int)((' @BACKCOLOR_ '-(int)(B * 256*256))/256);');
<nl>
wrt('        R = (byte)(int)( ' @BACKCOLOR_ '-(int)(B * 256*256 + G *
256));'); <nl>

wrt('        this.AutoScaleDimensions = new System.Drawing.SizeF(6F,
13F);'); <nl>
wrt('        this.AutoScaleMode =
System.Windows.Forms.AutoScaleMode.Font;'); <nl>

wrt('        this.BackColor = System.Drawing.Color.FromArgb(R,G,B);' ); <nl>
wrt('        this.Font = new System.Drawing.Font("' @FONTNAME_ '", '
@FONTSIZE_ ',' &Style ', System.Drawing.GraphicsUnit.Point, ((byte)(0)));'
); <nl>
wrt('        this.ClientSize = new System.Drawing.Size(' @WIDTH_ ','
@HEIGHT_ ');'); <nl>

wrt('        this.Name = "' @VARNAME_ '";'); <nl>
wrt('        this.Text = "' @NAME_ '";'); <nl>
wrt('        this.ResumeLayout(false);'); <nl>
wrt('        this.PerformLayout();'); <nl><nl><nl>

LOOP
  wrt('        this.Controls.Add(this.' @VARNAME ');'); <NL>
CLOSELOOP

wrt ('      }'); <nl><nl><nl>
wrt ('      #endregion');<nl><nl><nl>

LOOP
  wrt('      private System.Windows.Forms.' @TARGETLANGCLASS ' ' @VARNAME
';'); <NL>
CLOSELOOP


wrt ('   }'); <nl>
wrt ('}'); <nl>

@SOURCECODEFILENAME_ := @VARNAME_ '.Designer' @DEFAULTSCFILEEXT_;

[/TEMPLATE]
```

Quadro 49 – *Template* do arquivo “.cs”

APÊNDICE L – *Template* para geração de telas para Java

O Quadro 50 apresenta o *template* responsável pela geração do arquivo de extensão java.

```
[TEMPLATE]

TEMPLATEFILEEXT = '.java';

NAME = 'Java';

LANGUAGE = 'Java';

CLASSMAPPERFILE = 'JavaClassMapper.cmpp';

SPECIALCHARSFILE = 'Java.sps';

TEMPLATEDESC = 'Este template faz a geração do fonte para o arquivo .java';

start

&TemRadio := 0;

wrt('import javax.swing.*;');    <nl>
wrt('import java.awt.*;'); <nl><nl>

wrt('public class ' @VARNAME_ ' extends ' @TARGETLANGCLASS_ ' {' ); <NL>
wrt('  Container tela;' ); <NL>

wrt('  // Constructor'); <NL>
wrt('  ') ;

LC
   wrt( @SCOPE_ );
CLOSECASE

wrt('  ' @VARNAME_ '(){' ); <NL>
wrt('    tela = getContentPane();'); <NL>
wrt('    tela.setLayout(null);');  <NL>
wrt('    setSize(' @WIDTH_ ',' @HEIGHT_ ');'); <NL>
wrt('    super.setTitle("' @NAME_ '");' ); <NL>

wrt('    int R;');<NL>
wrt('    int G;');<NL>
wrt('    int B;');<NL>

&B := '    B = (int)('  @BACKCOLOR_ '/(256*256));';
&G := '    G = (int)((' @BACKCOLOR_ '-(int)(B * 256*256))/256);';
&R := '    R = (int)('  @BACKCOLOR_ '-(int)(B * 256*256 + G * 256));';

wrt(&B); <NL>
wrt(&G); <NL>
wrt(&R); <NL>

wrt( '    tela.setBackground(new Color(R,G,B));' ); <NL>

&B := '    B = (int)('  @FONTCOLOR_ '/(256*256));';
&G := '    G = (int)((' @FONTCOLOR_ '-(int)(B * 256*256))/256);';
&R := '    R = (int)('  @FONTCOLOR_ '-(int)(B * 256*256 + G * 256));';

wrt(&B); <NL>
wrt(&G); <NL>
wrt(&R); <NL>
```

```
wrt( '    tela.setForeground(new Color(R,G,B));' ); <NL>

&Style := '';
if (@FONTBOLD = true) then {
  &Style := 'Font.BOLD';
}

if (@FONTITALIC = true) then {
  if (&Style = '') then {
    &Style := 'Font.ITALIC';
  } else {
    &Style := 'Font.ITALIC + Font.BOLD';
  }
}

if (&Style = '') then {
  &Style := 'Font.PLAIN';
}

wrt( '    tela.setFont(new Font("' @FONTNAME_ '",' &Style ', ' @FONTSIZE_
'));' ); <NL><NL>

LOOP

  &Style := '';
  if (@FONTBOLD = true) then {
    &Style := 'Font.BOLD';
  }

  if (@FONTITALIC = true) then {
    if (&Style = '') then {
      &Style := 'Font.ITALIC';
    } else {
      &Style := 'Font.ITALIC + Font.BOLD';
    }
  }

  if (&Style = '') then {
    &Style := 'Font.PLAIN';
  }

  if (@GROUP = 107) then {
    wrt( '    String V' @VARNAME ' [] = {"' @NAME '"};' ); <NL>
    wrt( '    ' @TARGETLANGCLASS ' ' @VARNAME ' = new ' @TARGETLANGCLASS
'(V' @VARNAME ');' ); <NL>
  }
  else {
    wrt( '    ' @TARGETLANGCLASS ' ' @VARNAME ' = new ' @TARGETLANGCLASS
'();' ); <NL>
  }

  if (@GROUP = 101) then {
    wrt( '    ' @VARNAME '.setOpaque(true);' );<NL>
  }

  if (@GROUP = 0) then {
    wrt( '    ' @VARNAME '.setBounds(' @RECTLEFT ',' @RECTTOP  ',' @WIDTH
',' @HEIGHT ');' );<NL>
  }

  if (@GROUP = 1) then {
    wrt( '    ' @VARNAME '.setText("' @NAME '");' ); <NL>
  }

  if (@GROUP = 2) then {
    &B := '    B = (int)('  @FONTCOLOR '/(256*256));';
    &G := '    G = (int)((' @FONTCOLOR '-(int)(B * 256*256))/256);';
    &R := '    R = (int)('  @FONTCOLOR '-(int)(B * 256*256 + G * 256));';
```

```
    wrt(&B); <NL>
    wrt(&G); <NL>
    wrt(&R); <NL>
    wrt( '    ' @VARNAME '.setForeground(new Color(R,G,B));' ); <NL>
  }

  if (@GROUP = 3) then {
    wrt( '    ' @VARNAME '.setFont(new Font("' @FONTNAME '",' &Style ', '
@FONTSIZE '));' ); <NL>
  }

  if (@GROUP = 4) then {
    &B := '    B = (int)('  @BACKCOLOR '/(256*256));';
    &G := '    G = (int)((' @BACKCOLOR '-(int)(B * 256*256))/256);';
    &R := '    R = (int)('  @BACKCOLOR '-(int)(B * 256*256 + G * 256));';

    wrt(&B); <NL>
    wrt(&G); <NL>
    wrt(&R); <NL>
    wrt( '    ' @VARNAME '.setBackground(new Color(R,G,B));' ); <NL>
  }

  if (@GROUP = 106) then {

    if (&TemRadio = 0) then {
      wrt( '    ButtonGroup group = new ButtonGroup();'); <NL>
      &TemRadio := 1;
    }

    wrt('    group.add(' @VARNAME ');' ); <NL>
  }

  wrt( '    //GUID: ' @GUID); <NL>
  wrt( '    //  ID: ' @ID); <NL>

  wrt( '    tela.add(' @VARNAME ');' );<NL><NL>

CLOSELOOP

<NL>

wrt('    setVisible(true);' ); <NL>
wrt( '  }' );<NL><NL>

wrt( '  public static void main(String args[]){' ); <NL>
wrt( '  '   @VARNAME_ ' app = new ' @VARNAME_ '();' ); <NL>
wrt( '    app.setDefaultCloseOperation(' @TARGETLANGCLASS_ '.EXIT_ON_CLOSE);
' ); <NL>
wrt( '  }' ); <NL>
wrt( '}' );

[/TEMPLATE]
```

Quadro 50 – *Template* do arquivo “.java”