UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

EDITOR VISUAL DE INTERFACES GRÁFICAS COM O USUÁRIO PARA LINGUAGEM DE ESPECIFICAÇÃO DE DIÁLOGOS (LED)

OMAR FERNANDO PESSÔA

BLUMENAU
2007

2007/2-27

OMAR FERNANDO PESSÔA

EDITOR VISUAL DE INTERFACES GRÁFICAS COM O USUÁRIO PARA LINGUAGEM DE ESPECIFICAÇÃO DE DIÁLOGOS (LED)

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. Paulo César Rodacki Gomes, Doutor - Orientador

BLUMENAU
2007

2007/2-27

EDITOR VISUAL DE INTERFACES GRÁFICAS COM O USUÁRIO PARA LINGUAGEM DE ESPECIFICAÇÃO DE DIÁLOGOS (LED)

Por

OMAR FERNANDO PESSÔA

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

Presidente:	Prof. Paulo César Rodacki Gomes, Doutor – Orientador, FURB
Membro:	Prof. Dalton Solano dos Reis, Mestre – FURB
Membro:	Prof. Mauro Marcelo Mattos, Doutor – FURB

Blumenau, 05 de dezembro de 2007

Dedico este trabalho a todos os familiares e amigos, especialmente aqueles que me ajudaram diretamente na realização deste.

AGRADECIMENTOS

À minha família que sempre me apoiou.

Aos meus amigos, pelos empurrões e cobranças, aqueles amigos que souberam me deixar em casa para completar mais essa tarefa.

Ao meu orientador, Paulo César Rodacki Gomes, por ter acreditado na conclusão deste trabalho.

Ao pessoal da empresa Microton Informática LTDA.

Embora ninguém possa voltar atrás e fazer um novo começo, qualquer um pode começar agora e fazer um novo fim.

Francisco Cândido Xavier

RESUMO

O objetivo deste trabalho é desenvolver um protótipo de ferramenta, utilizando a *toolkit* IUP/LED, para a edição gráfica de arquivos de linguagem de especificação de diálogos. Utilizando para isso a própria *toolkit* IUP/LED. É apresentado um conhecimento breve sobre *widget*, os seus tipos, bem como seus eventos; também um conceito de *What You See Is What You Get* (WYSIWYG) e a definição de manipulação direta. Depois, tem-se a introdução da *toolkit* de Interação com o Usuário Portável (IUP), com suas propriedades e características como também será conceituada a Linguagem de Especificação de Diálogos (LED). O protótipo apresenta uma vista gráfica, para a edição de leiaute concreto de edição de diálogos. São apresentados no respectivo trabalho, a descrição da linguagem, os problemas, suas soluções e também os processos envolvidos no desenvolvimento desse protótipo.

Palavras-chave: IUP. LED. TeCGraf. Interface gráfica.

ABSTRACT

The objective of this work is to develop a tool prototype, using the toolkit IUP/LED, for the graphic edition of dialog specification language files. Using for that the own toolkit IUP/LED. A brief knowledge on widget is showed, its types, as well as its events; also a concept of What You See Is What You Get (WYSIWYG) and the definition of direct manipulation. Then, can see the toolkit interaction with the Portable User Interface (IUP), with its properties and characteristics as well as the Dialog Specification Language (LED) is considered. This prototype presents a graphic view, for the edition of concrete layouts of dialog edition. They are presented in the respective work, the description of the language, the problems, its solutions and also the processes involved in the development of that prototype.

Key-words: IUP. LED. TeCGraf. Graphical interface.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE SIGLAS

API – *Application Programming Interface*

BCC – Curso de Ciências da Computação – Bacharelado

DSC – Departamento de Sistemas e Computação

GUI – *Graphical User Interface*

IUP – Interface com o Usuário Portável

LED – Linguagem de Especificação de Diálogos

TeCGraf – Grupo de Tecnologia em Computação Gráfica

UML – *Unified Modeling Language*

WYSIWYG – *What You See Is What You Get*

SUMÁRIO

1 INTRODUÇÃO

Segundo Thimblely (1990, p. 1), “com a invenção da escrita, cerca de 5000 anos atrás a humanidade deu um passo muito importante para o seu progresso, pois a partir dali o homem registrou suas leis, suas histórias, as civilizações bem como os cálculos”.

Thimblely (1990, p. 1), afirma ainda que recentemente, aproximadamente há 40 anos, entra no cenário histórico o computador eletrônico, e que “descobrimos que podíamos fazer tudo isso, pensar além das nossas cabeças, sem registrar absolutamente nada”.

De acordo com Redmond-Pile e Moore (1995, xiii), nos últimos anos o mundo tem mudado na informática, de tal forma que os usuários precisam saber quais comandos e parâmetros digitar no *prompt*, via tabulador, ou digitando através de caixas de texto de dados em um formulário verde e pressionando a tecla *enter*, clicar em janelas, empurrar o menu com um mouse, ou, mais recentemente, manipulando a tela de toque (*touch-screen*).

Um sistema utilizando a interface gráfica com o usuário (*Graphical User Interface* - GUI) é mais prático, mais fácil de aprender, mais efetivo e satisfatório do que sistemas baseados em interface de texto. A interface com o usuário é parte do sistema computacional que permite a interação do usuário humano com o computador. Exemplos já existentes de elementos que permitem interação humano-computador são: partes do hardware do computador (tela, teclado, *mouse*); imagens móveis na tela (janelas, menus, mensagens); documentos do usuário (manuais e apresentações).

Para um usuário, é interessante que o aplicativo como um todo seja transparente, isto é, em geral para o usuário o mais importante são as entradas de dados na tela e suas saídas, seja como relatórios, e ou impressões em tela, não importando ao usuário, como é feito o tratamento da problemática.

No Brasil, mais precisamente na Pontifícia Universidade Católica do Rio de Janeiro, existe um grupo de trabalho (grupo de Tecnologia em Computação Gráfica - TeCGraf) que desenvolve aplicações gráficas que requerem uma grande interação com o usuário, sendo estas executadas em diversas plataformas e sistemas operacionais. Este grupo, o TeCGraf, desenvolveu a Interface com o Usuário Portável (IUP), e Linguagem de Especificação de Diálogo (LED).

Com IUP/LED, a edição e criação de uma interface são feitas de forma textual, o que pode resultar na perda de muito tempo para arrumar um controle ou um elemento nessa interface.

Uma ferramenta interativa para geração de interfaces, como a disponibilizada pelo Visual C++ 6.0, o editor de *resources*, faz com que o trabalho seja muito menor na criação e alteração de interfaces.

A interface pode não ser tão complexa quanto à resolução do problema descrito pela aplicação, mas é parte essencial para a interação com o usuário. Uma boa aplicação exige uma interface amigável e fácil de utilizar. Uma interface mal projetada pode dificultar o entendimento e o uso da aplicação, fazendo com que o usuário dispenda muito tempo e energia na utilização do sistema, ao invés de concentrá-lo na tarefa a ser executada. Isto pode causar frustração no usuário, levando-o, até a abandonar o sistema.

O intuito deste trabalho é desenvolver um editor gráfico interativo de interfaces gráficas de usuário para o desenvolvimento de aplicações que sejam baseadas no *toolkit* IUP/LED. Neste editor o programador, desenvolvedor, utilizar-se-á de uma interface gráfica para montagem e edição de “telas” para suas aplicações utilizando a *toolkit* IUP.

1.1 OBJETIVOS DO TRABALHO

O objetivo geral deste trabalho é desenvolver um editor visual interativo para interfaces gráficas com o usuário.

Os objetivos específicos deste trabalho são:

a) edição visual de elementos de diálogos;
b) alteração das propriedades e atributos dos elementos do diálogo;
c) geração de arquivos de descrição das interfaces em sintaxe da linguagem de especificação de diálogos (LED).

1.2 ESTRUTURA DO TRABALHO

O capítulo 2 apresenta a fundamentação teórica na qual esse trabalho é embasado. Inicialmente é apresentada uma introdução sobre interfaces de uma aplicação, o que são *widgets*, eventos, tipos de *widgets*; qual o conceito de *What You See Is What You Get* (WYSIWYG), e a conceituação de manipulação direta. Em seguida é introduzida a *toolkit*

IUP, suas propriedades e características. Para o fim do capítulo de fundamentação, é conceituada a Linguagem de Especificação de Diálogos (LED).

No capítulo 3 são abordados os processos envolvidos no desenvolvimento do protótipo, tais como a definição dos principais requisitos do sistema, especificação com os diagramas de casos de uso e diagrama de atividades, técnicas e ferramentas utilizadas, a operacionalidade do protótipo e uma explanação sobre os resultados obtidos.

O capítulo 4 apresenta as conclusões sobre a aplicação das técnicas propostas, bem como uma discussão a respeito de propostas para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

No presente capítulo são apresentados alguns aspectos teóricos relacionados ao trabalho. Na seção 2.1 é explicado o que é interface de uma aplicação. Na seção 2.2 é evidenciada a importância da utilização da *toolkit* IUP e na seção 2.3 é apontado o que é a Linguagem de Especificação de Diálogos (LED).

2.1 INTERFACE DE UMA APLICAÇÃO

De acordo com Prates (1994, p. 1) “a interface é a comunicação entre o usuário e a aplicação”. Thimblely (1990, p. v) afirma que as interfaces com o usuário são todas as partes de um sistema computacional que permitem o acesso às facilidades oferecidas pelo computador, para a pessoa que o está utilizando.

Nem todo o usuário de computador tem um conhecimento amplo de seus comandos e operações. Em regra geral só conhece aquilo que de fato necessita para o seu trabalho.

Moreno (2007) descreve que estas interfaces eram manejadas de forma textual, mediante comandos críticos, que somente pessoas com muita experiência poderiam entender. Com o desenvolvimento de novas tecnologias, a interface passou a ser um meio capaz de tornarem entendíveis e usáveis estas aplicações através de elementos visuais comuns. Então com o aparecimento das interfaces gráficas a comunicação entre o usuário e o seu trabalho (aplicação) torna-se bem mais fácil.

Segundo Marcus (1992), a interface gráfica de uma aplicação pode ser composta por um ou mais diálogos, onde diálogo é um grupo de objetos de interface que estão em um contexto espacial limitado. Especificar o leiaute de um diálogo significa descrever a composição visual deste diálogo. Esta especificação pode ser feita de várias formas diferentes, e não existe um consenso sobre qual delas é a melhor. Sendo que os dois principais paradigmas são: o leiaute abstrato e o leiaute concreto (PRATES, 1994, p. 4).

De acordo com Figueiredo, Gattass e Prates (1994, p. 166), descrever um leiaute abstratamente significa especificar a posição relativa dos elementos. Prates (1994, p. 4), expõe a idéia do leiaute concreto como sendo o fornecimento explicito das posições e tamanhos dos elementos de interface e que a forma de fornecer essas posições varia de acordo com o

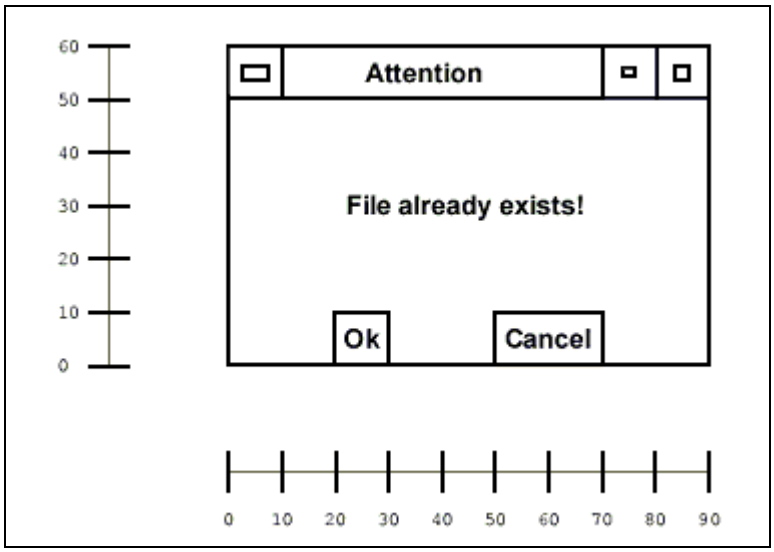
sistema de interface utilizado. Em todos os modelos de leiaute se fazem necessários o conhecimento prévio do tamanho de tela e cada elemento de um diálogo.

A especificação abstrata de um diálogo para a geometria da Figura 1 é:

a) botão Ok: 22% à direita do canto inferior do diálogo;
b) botão Cancel: 22% à direita do botão Ok;
c) mensagem: centralizada horizontalmente e verticalmente no espaço acima dos botões.

A especificação concreta de um diálogo para a geometria da Figura 1 é:

a) diálogo: posição (0, 0), tamanho 90x 60;
b) botão Ok: posição (20, 0), tamanho 10 x 10;
c) botão Cancel: posição (50,0), tamanho 20 x 10;
d) mensagem: posição (20, 30).



Fonte: Prates (1994, p. 5).

Figura 1 – Geometria de leiaute de um diálogo

2.1.1 Elementos de uma interface

Os sistemas de interface suportam, normalmente, um conjunto padronizado de objetos que são denominados *widgets* por Santos (1996, p. 2), que completa explicando que *widgets* são elementos de interface genéricos que podem ser posteriormente associados a objetos como menus, botões, entre outros. Esses objetos podem ser agrupados, de modo a definir os diálogos da aplicação.

Santos (1996, p. 5) explica que há duas categorias de *widgets*: os de baixo e alto nível.

Widgets de baixo nível são utilizados na confecção do sistema operacional e fazem parte do núcleo do sistema e os *widgets* de alto nível seriam os objetos finais propriamente ditos, que fazem referência aos *widgets* de baixo nível.

2.1.2 Eventos

Eberts (1994, p. 24) descreve um evento de interface como sendo qualquer coisa que o usuário faz. Um evento pode ser uma tecla especifica pressionada, qualquer tecla pressionada, o movimento do cursor do mouse sobre um *widget*, pode ser também um clique do *mouse* sobre um *widget*.

Eberts (1994, p. 25) diz que os programas baseados em eventos possuem um *loop* e que esse *loop* espera e empilha os eventos ocorridos e em qual *widget* ocorreram. Se este *widget* tiver uma rotina para este evento específico, então o *loop* ira executar a parte do código da rotina deste evento. De acordo com Eberts (1994, p. 28), se um evento ocorre em um *widget* que este não está procurando, então o evento será ignorado.

2.1.3 Tipos de *widgets*

Conforme Eberts (1994, p. 28), os *widgets* são definidos de acordo com os eventos associados a sua *window*, que de acordo com Santos (1996, p. 31) é um tipo de espaço visual. *Windows* mapeiam janelas do sistema nativo, repassando os eventos do usuário e ou sistema nativo, e oferecendo uma superfície para aplicação de primitivas gráficas.

Existem vários tipos de *widgets* ou controles, os abaixo relacionados são os comuns entre os vários *toolkits* existentes. Alguns *widgets* descritos por Levi (1993, p. 20) são apresentados no quadro 1.

A Figura 2 exibe um diálogo contendo alguns dos *widgets* citados.

Widget	**Descrição**
Label	É utilizado para informar algo para o usuário. Sua apresentação para o usuário pode ser um texto ou uma imagem.
List	O elemento *list* exibe uma lista de opções, geralmente do tipo *string*, associada a uma barra de rolamento vertical.
Button	Representa uma ação da aplicação. Para executar a ação, o usuário deve acionar o botão, pressionando e soltando o botão do *mouse*.
Toggle	É um *button* de dois estados: *on* e *off*. O *toggle*, ao lado da sua apresentação visual, reserva uma área onde duas imagens aparecem alternadamente, na medida em que o usuário o seleciona.
Text	O elemento captura uma ou mais linhas de texto digitado pelo usuário. *Text* pode ter uma barra de rolamento vertical e/ou horizontal para permitir que o usuário entre com um texto maior que a área reservada pelo elemento.
Menu	É uma apresentação visual de operações que o usuário pode executar. Geralmente, um *menu* é formado por uma estrutura hierárquica, onde um ou mais itens do *menu* podem abrir um outro menu.

Fonte: adaptado de Levy (1993).

Quadro 1 – Exemplos de *widgets*

Figura 2 – Elementos de interface: *label*, *list*, *button*, *toogle*, *text*, *menu*

2.1.4 WYSIWYG

WYSIWYG é o acrônimo da expressão em inglês *What You See Is What You Get*, cuja tradução remete a algo como “O que você vê é o que você tem” (OQVVEOQVT) (FOLEY, 2003, p. 396).

O termo denota a capacidade de um programa de computador de permitir que um documento, enquanto manipulado em tela, tenha a mesma aparência de sua utilização, usualmente sendo considerada final a forma impressa.

Foley (2003, p. 396) adiciona que, WYSIWYG é fundamental para a interatividade gráfica. A representação com que o usuário interage no display em uma interface WYSIWYG, é essencialmente a mesma imagem final criada pela aplicação.

2.1.5 Manipulação direta

O conceito de manipulação direta para Foley (2003, p. 397) é uma maneira de interação entre o homem e o computador, com um envolvimento contínuo de representação de objetos de interesse, e ações rápidas, reversíveis e de retorno. A intenção é permitir a um usuário manipular diretamente objetos a ele apresentados, usando ações que no mínimo correspondem ao mundo físico.

Com o uso de metáforas para o mundo real em objetos e ações torna-se mais fácil para o usuário aprender e usar uma interface, um *feedback* rápido faz com que o usuário cometa menos erros, completando tarefas em menos tempo. Isto é possível por que o usuário pode ver os resultados de uma ação antes de completá-la.

Foley (2003, p. 397) explica, também, que a manipulação direta de interface de um usuário também pode ser conceituada como sendo aquela em que os objetos, atributos ou relações que podem ser operadas ou são visualmente representados, operações são invocadas por ações desempenhadas em representações visuais, tipicamente utilizando o *mouse*.

Desta forma WYSIWYG e manipulação direta são separadas em conceitos distintos. Por exemplo, a representação textual de uma imagem gráfica pode ser modificada via manipulação direta, e que uma imagem gráfica de um sistema WYSIWYG pode ser modificada puramente por uma interface de linguagem de comando. Especialmente quando usados em conjunto, no entanto, os dois conceitos são poderosos, fácil de aprender e

razoavelmente rápido de usar, como muitas interfaces de usuário têm demonstrado (FOLEY, 2003, p. 398).

2.2 TOOLKIT IUP

O grupo TeCGraf desenvolve, entre outras coisas, programas gráficos interativos para diversas áreas da engenharia. O principal cliente do TeCGraf é a PETROBRAS, que possui um parque de computadores bastante diversificado. Esta diversificação exige que as aplicações desenvolvidas pelo TeCGraf sejam multiplataforma, isto é, possam ser executadas em diferentes ambientes computacionais.

Levy (1993, p. 4) diz que para resolver o problema de portabilidade das rotinas gráficas, o Centro de Pesquisas da PETROBRÁS (CENPES) solicitou ao TeCGraf a utilização do padrão internacional para sistemas gráficos. Em 1986 o CENPES e o TeCGraf resolveram absorver por completo a tecnologia de sistemas gráficos, seguindo rigorosamente as normas ANSI e ISO.

Para sistemas de interfaces não havia padrões internacionais. Desta forma, o TeCGraf acabou criando o IntGraf, um sistema portátil de interface gráfica.

O IUP/LED foi projetado para solucionar os problemas encontrados com o IntGraf, preservando as suas duas principais qualidades: facilidade de uso, tanto pelos programadores quanto pelos usuários finais da aplicação; e portabilidade para diferentes ambientes computacionais.

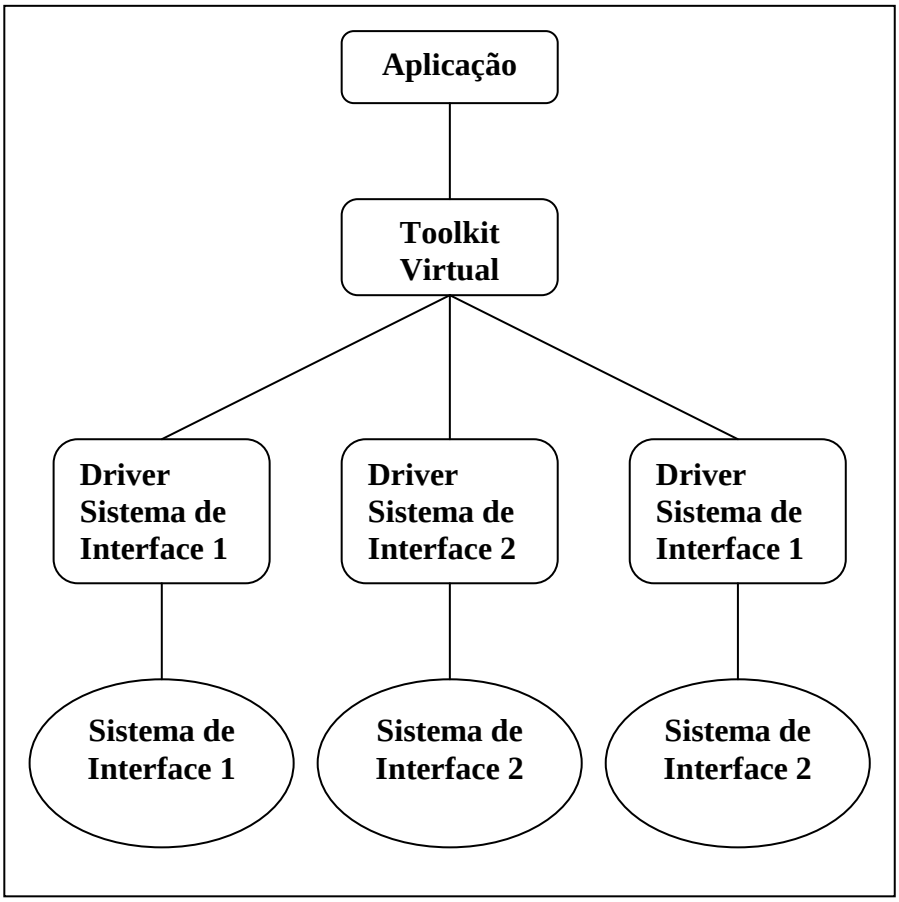
De acordo com Scuri (2007), IUP é um *toolkit* portátil para a construção de interfaces gráficas com usuários. Oferece *Applications Programming Interface* (API) em três linguagens de programação: C, Lua e LED. Scuri (2007) completa dizendo que a *toolkit* IUP possui cerca de 100 funções para a criação e manipulação de diálogos.

Algumas das principais características da *toolkit* IUP/LED são a portabilidade para diferentes ambientes computacionais e o fato de possuir uma linguagem de especificação de diálogos (LED) interpretada em tempo de execução da aplicação.

Levy (1993, p. 14) afirma que a grande variedade de sistemas de interface com usuários torna difícil o desenvolvimento de programas interativos portáveis, e que “todos os sistemas de interfaces gráficas são do tipo WIMP, isto é, implementam a metáfora de *desktop*, usando *windows*, *icons*, *menu*, e *pointing device*”. O que deixa os sistemas com aparência

semelhante.

Para que o programador não tenha que ser especialista em cada um dos sistemas operacionais e suas interfaces, a *toolkit* IUP/LED implementa um interfaceamento gráfico entre a aplicação e os diversos sistemas operacionais, utilizando um *driver* de sistema de interface, como mostra Levy (1993, p. 17) na Figura 3.



Fonte: Levi (1993, p. 17).

Figura 3 - Arquitetura de um *toolkit* virtual

Scuri (2007) diz que, a *toolkit* IUP tem algumas vantagens em relação a outras interfaces de *toolkits* disponíveis:

a) simplicidade, devido ao pequeno número de funções e de seu mecanismo de atributos o aprendizado de um novo usuário é muitas vezes mais rápido;
b) portabilidade, as mesmas funções são implementados em cada uma das plataformas, assegurando, assim, a interface do sistema de portabilidade;
c) customização, a Linguagem de Especificação de Diálogos (LED) e IupLua, são dois mecanismos em que é possível personalizar uma aplicação para um usuário específico, utilizando uma linguagem textual;
d) flexibilidade, o mecanismo de leiaute abstrato provém flexibilidade a criação do diálogo;

e) extensibilidade, o programador pode criar novos elementos de interface se necessário.

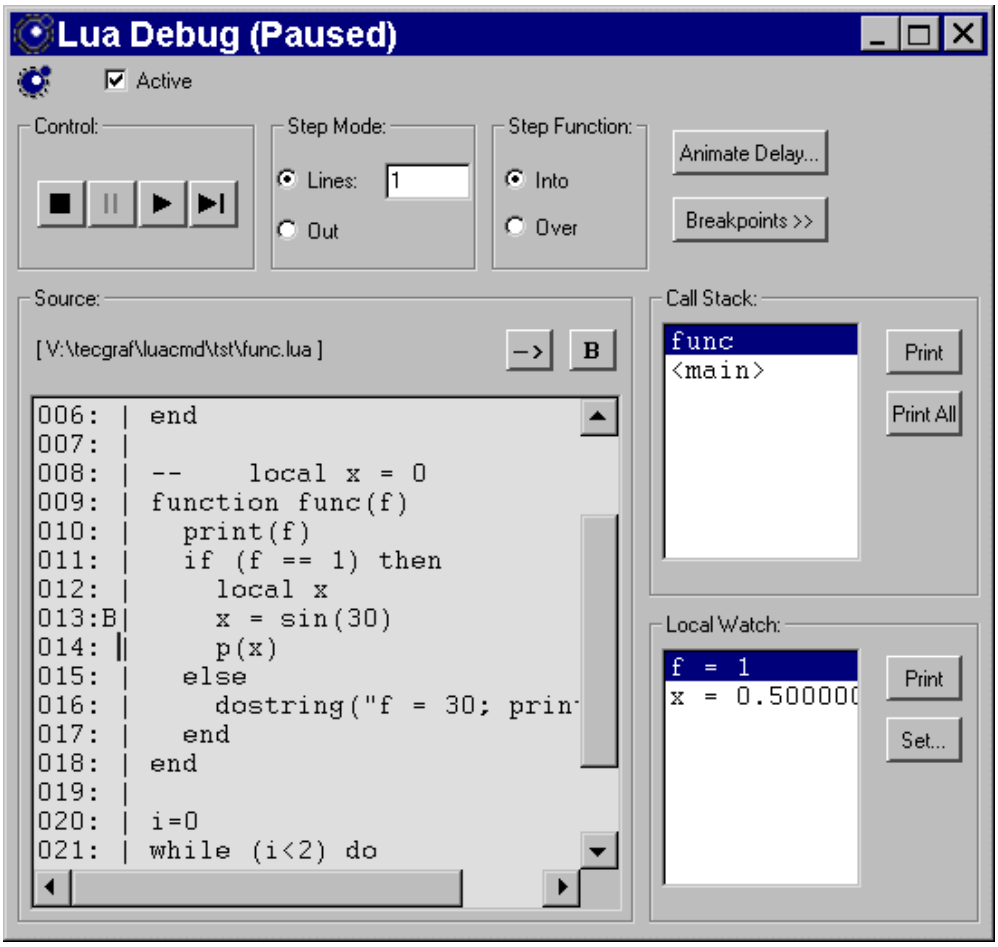
Segundo Scuri (2007) IUP possui alguns conceitos importantes que foram implementados de uma maneira diferente de outras *toolkits*, são eles:

a) cronograma de criação de um controle: quando um controle é criado, ele não é mapeado imediatamente ao sistema nativo, então alguns atributos não funcionarão a menos que esse controle seja mapeado;

b) sistema de atributos: IUP possui poucas funções, pois utiliza atributos do tipo *string* para acessar as propriedades de cada controle;

c) sistema de *callback*: por utilizar arquivos LED, IUP possui uma maneira indireta de associar *callback* a um controle. Associando uma função C com um nome utilizando a função `IupSetFunction`.

A Figura 4 ilustra um exemplo de interface utilizando a *toolkit* IUP/LED, exemplificada por Scuri (2007).



Fonte: Scuri (2007).

Figura 4 – Interface utilizando a toolkit IUP/LED

2.3 LINGUAGEM DE ESPECIFICAÇÃO DE DIÁLOGOS (LED)

Scuri (2007) explica que LED é uma linguagem de expressões para especificação de leiaute de diálogos. Esta linguagem é simples, de fácil compreensão, e descreve diálogos textualmente. Sua finalidade não é ser uma linguagem de programação completa, mas de preferência, fazer a especificação do diálogo de forma mais simples do que utilizando a linguagem C.

Segundo Prates (1994, p. 12), LED permite a especificação de vários diálogos para uma mesma aplicação. Os códigos dos diálogos são independentes entre si e são independentes do código da tecnologia da aplicação. Esta independência permite que se altere a interface ou a tecnologia da aplicação sem que a alteração de uma implique, necessariamente, na alteração da outra.

Em LED não há distinção entre maiúsculas e minúsculas, exceto para nome de atributos, o modelo de especificação de diálogos adotado por LED segue o paradigma de leiaute abstrato.

O código fonte apresentado no Quadro 2 é um exemplo de uma interface utilizando LED. Sua interface resultante é apresentada na Figura 5.

```
dialogo = dialog[TITLE = "Hello"]
(
   hbox(
     button[SIZE="40"]("OK",acao_ok),
     button[SIZE="40"]("Cancel", acao_cancel)
   )
)
```

Quadro 2 – Exemplo de código fonte utilizando LED

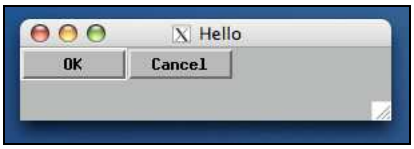


Figura 5 – Exemplo de interface

O sistema IUP/LED tem sido utilizado em produção pelo TeCGraf com sucesso. Os elementos da interface oferecidos são suficientes, permitindo que boas interfaces sejam projetadas. Para criar esta interface, o programador deve aprender LED, que é uma linguagem simples e concisa, portanto de fácil aprendizagem. No entanto novas dificuldades foram encontradas (PRATES, 1994, p. 2).

Ao projetar uma interface, deve-se primeiro visualizá-la mentalmente e depois traduzir esta visão em comandos LED, o que pode ser uma tarefa bastante trabalhosa. Reciprocamente, ler uma descrição LED e visualizá-la pode ser igualmente difícil, principalmente se o diálogo sendo descrito é grande, ou se a interface é composta por mais de um diálogo, o que é freqüente.

Scuri (2007) analisa que embora os arquivos LED sejam arquivos texto, não há maneira de interpretar um texto em memória. Há apenas a função `IupLoad`, da *toolkit* IUP, que lê um arquivo LED e cria os elementos definidos nesse arquivo. Naturalmente, o mesmo arquivo não pode ser carregado mais de uma vez, porque os elementos seriam criados novamente. Este arquivo interpretado não mapeia os elementos para o sistema nativo.

De acordo com Scuri (2007) os arquivos LED são carregados dinamicamente e devem ser enviados juntamente com o executável.

2.3.1 Elementos de interface

Prates (1994, p. 13) afirma que “os elementos que compõem o diálogo em LED são especificados pela sua funcionalidade, que é fornecida por parâmetros, enquanto a sua aparência é definida por atributos.”. Os parâmetros são obrigatórios enquanto os atributos são opcionais. Segundo Scuri (2007) em LED atributos e expressões seguem como especificados no Quadro 3.

```
elem = elemento [atributo1=valor1, atributo2=valor2,...](expressão...)
```

Quadro 3 – Definição de um elemento em LED

Os nomes dos elementos não devem possuir o prefixo “iup”, os valores dos atributos são sempre interpretados como *string*, precisando estar entre aspas duplas, somente se conter espaços. Cada controle possui uma única função de criação e a sua gestão é feita através de atributos e chamadas, utilizando funções comuns a todos os controles, conforme Scuri (2007).

De acordo com Scuri (2007) os elementos de interface existentes podem ser categorizados como:

a) primitivos: interação efetiva do usuário;
b) composição: exibição dos elementos;
c) agrupamento: definição de uma usabilidade comum para um grupo de elementos;
d) menu: relacionado a barras de menu;
e) adicionais: elementos construídos fora da biblioteca principal;

f) diálogos: usualmente diálogos predefinidos.

2.3.1.1 Atributos dos elementos de interface

Atributos são usados para alterar as propriedades de elementos. Cada elemento tem um conjunto de atributos que os afetam, e cada atributo pode trabalhar de maneira diferente para cada elemento. Dependendo do elemento, o seu valor pode ser computado ou simplesmente confirmado. Ele também pode ser armazenado internamente ou não. É o que explica Scuri (2007).

Levy (1993, p. 40) afirma que "os atributos de elementos de interface são implementados como variáveis de ambiente, isto é, são representados por pares ordenados (v, a), onde v é o nome de uma variável e a é o seu conteúdo (um *string*).".

O Quadro 4 mostra um exemplo de atributo do tipo fonte.

```
confirm = dialog [ FONT="Helvetica"    ] (...)
replace = button [ FONT="HelveticaBold"] (...)
```

Fonte: Levi (1993, p. 40).

Quadro 4 – Exemplo de atributo

2.3.2 Diálogo

Scuri (2007) afirma que utilizando IUP podem-se criar diálogos próprios, ou usar os diálogos predefinidos. Para criar diálogos próprios devem-se criar todos os controles do diálogo antes da criação do mesmo. Todos os controles devem ser compostos de uma estrutura hierárquica de forma que a raiz será usada como parâmetro para a criação do diálogo.

De acordo com Scuri (2007) quando um controle é criado, seu *parent* (diálogo onde são inseridos os elementos de interface) não é conhecido. Após a criação do diálogo todos os elementos recebem um *parent*. Este mecanismo é muito diferente dos sistemas nativos, que primeiro criam o diálogo e, em seguida, o elemento é inserido, usando o diálogo criado como *parent*. Esse recurso cria algumas limitações para IUP, geralmente relacionados com a inserção e remoção de controles dinamicamente.

Uma vez que os controles são criados em uma ordem diferente do sistema nativo, controles nativos só podem ser criados após o diálogo. Isto irá acontecer automaticamente quando for chamada a função `IupShow` para exibir o diálogo. Caso necessário, existe a função `IupMap`, que mapeia os controles nativos antes da chamada da função `IupShow`. É o que analisa Scuri (2007).

Segundo Scuri (2007) os diálogos predefinidos são:

a) `IupFileDlg`: diálogo predefinido para selecionar arquivos ou diretórios;
b) `IupMessageDlg`: diálogo predefinido para exibir uma mensagem;
c) `IupGetColor`: exibe um diálogo para o usuário selecionar uma cor.

3 DESENVOLVIMENTO

Este capítulo tem como objetivo apresentar as etapas envolvidas no desenvolvimento do protótipo proposto por este trabalho. Serão abordadas as seguintes etapas: requisitos do protótipo, especificação, implementação e, finalmente, serão apresentados os resultados obtidos.

3.1 REQUISITOS PRINCIPAIS DO PROTÓTIPO

Para o desenvolvimento do protótipo proposto pensou-se no que fazer, como fazer e o que estudar para implementação de um editor gráfico interativo, para isso foram levantados alguns requisitos, esses requisitos são:

a) o protótipo proposto deverá possuir uma interface gráfica interativa para a movimentação, inclusão e exclusão de controles (requisito funcional).
b) deverá criar um novo arquivo LED, abrir e alterar um arquivo LED existente (requisito funcional).
c) exibir e permitir alterar as propriedades e atributos, referente aos elementos do diálogo (requisito funcional).
d) utilizar para seu desenvolvimento o ambiente de programação Visual C++ 6.0 (requisito não funcional).
e) fazer a utilização da *toolkit* IUP/LED (requisito não funcional).

3.2 ESPECIFICAÇÃO

Para realizar a especificação do protótipo foi utilizada a linguagem UML, descrita em Furlan (1998). Como diagramas da especificação foram usados os diagramas de caso de uso, de classe e de seqüência do protótipo.

3.2.1 Diagrama de casos de uso

O protótipo da aplicação possui cinco casos de uso, que correspondem as principais funcionalidades da aplicação: editar um arquivo LED, abrir um novo arquivo LED, salvar um arquivo LED, abrir um arquivo LED e visualizar o diálogo em edição.

O diagrama de casos de uso utilizado é de fato muito simples, pois há uma janela principal para interação gráfica com o usuário e diálogos de propriedades de elementos, como mostrado na Figura 6.

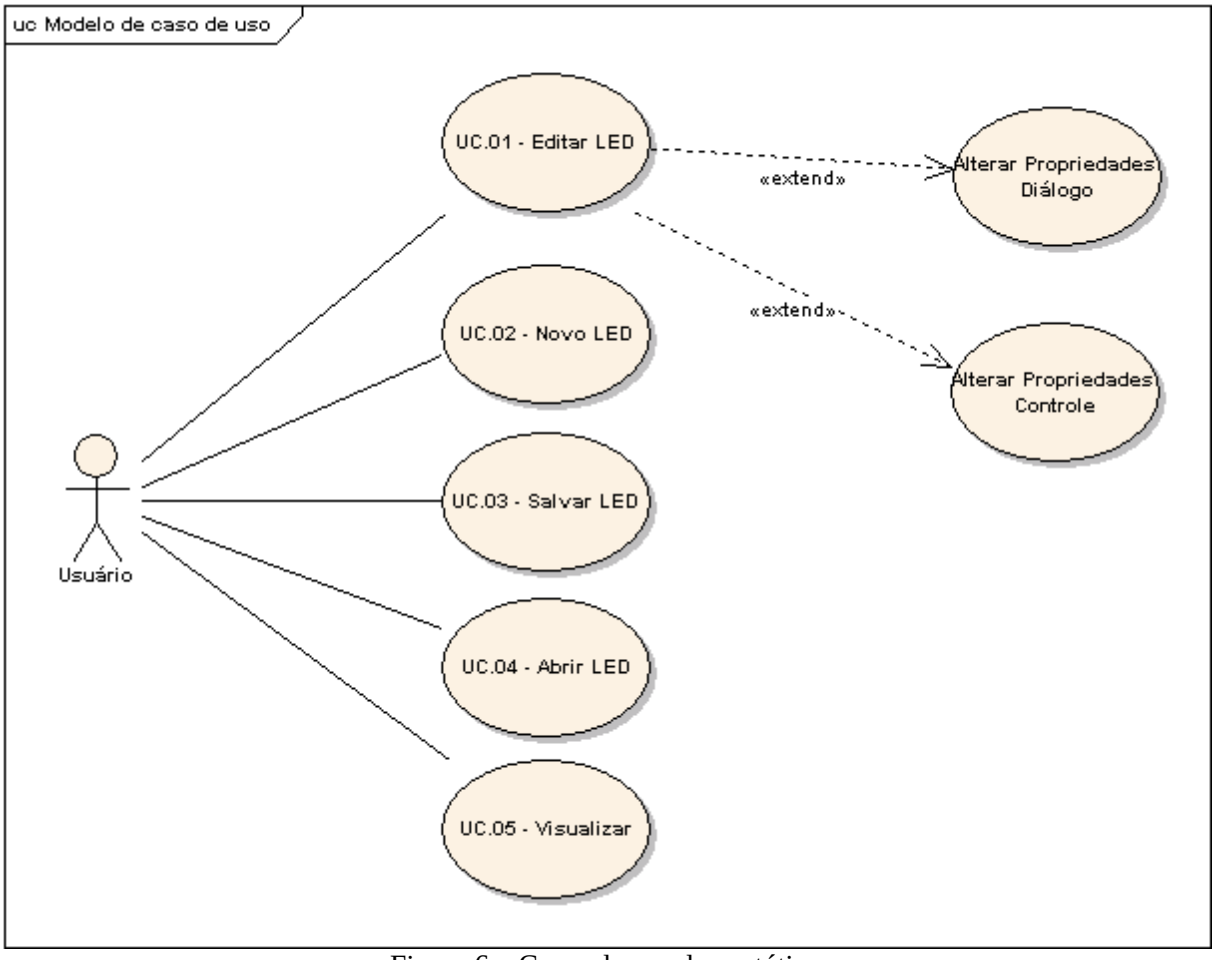


Figura 6 – Casos de uso do protótipo

Ao usuário iniciar a aplicação, ele terá a janela de edição. Bastando apenas a seleção de um elemento de diálogo para iniciar a edição. A Figura 7 exibe a tela principal da aplicação.

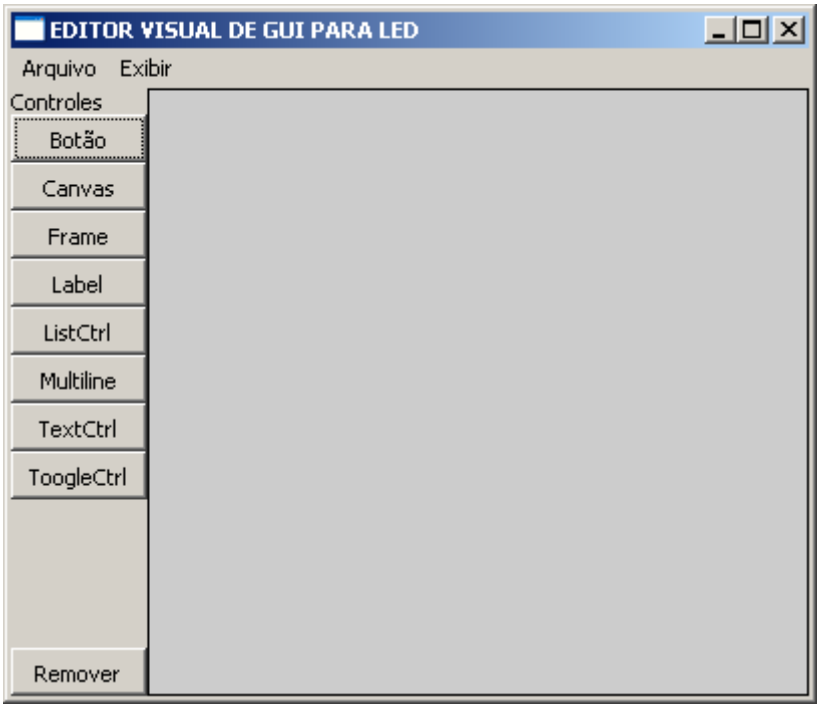


Figura 7 – Tela de edição do protótipo

Para a seleção de um elemento, deve-se clicar em um dos botões de controles, ao lado esquerdo do diálogo. Após o clique, clicar novamente no canvas para adicionar o controle ao canvas, que será nosso diálogo.

A descrição dos casos de uso do protótipo de ferramenta gráfica interativa elencados por esse autor são os apresentados nos quadros 5, 6, 7, 8 e 9.

Caso de uso:	**Novo LED**
Cenário principal	
1.1 - Sistema limpa canvas da tela principal	
1.2 - Caso de uso editar LED	
Cenário Alternativo	
No item 1.1, caso exista um arquivo LED em edição	
2.1 - Sistema exibe mensagem “Deseja salvar o arquivo LED em edição”	
2.2 - Caso o usuário clique OK	
2.2.1 - Caso de uso salvar arquivo LED	

Quadro 5 – Caso de uso novo LED

Caso de uso:	**Editar LED**
Cenário principal	
1.1 - Sistema exibe tela principal (Figura 7)	
1.2 - Usuário seleciona um controle, clicando em um botão	
1.3 - Usuário clica no canvas de edição	
1.4 - Sistema cria o controle	
Cenário Alternativo	
No item 1.4, caso o usuário clique no controle, no canvas, selecionando-o:	
2.1 - O controle é selecionado	
2.2 - Caso o usuário selecione no *menu*, Exibir, Propriedades	
2.3 - Sistema exibe tela de edição de propriedades de um controle	
2.4 - Usuário seta atributos do controle	
2.5 - Usuário fecha tela de propriedades	
2.6 - Sistema atualiza atributos do controle	
No item 1.1, caso o usuário selecione no *menu*, Exibir, Propriedades, sem que haja um controle selecionado	
3.1 - Sistema exibe tela de propriedades do diálogo	
3.2 - Usuário seta atributos do diálogo	
3.3 - Usuário fecha tela de propriedades	
3.4 – Sistema atualiza atributos do diálogo	

Quadro 6 – Caso de uso editar LED

Caso de uso:	**Abrir LED**
Cenário principal	
1.1 - Sistema exibe diálogo para seleção de arquivo	
1.2 - Usuário seleciona nome de arquivo	
1.3 - Sistema carrega arquivo selecionado	
1.4 - Sistema exibe controles	
1.5 - Caso de uso Editar LED	

Quadro 7 – Caso de uso abrir LED

Caso de uso:	**Salvar LED**
Cenário principal	
1.1 - Sistema exibe diálogo para seleção de arquivo	
1.2 - Usuário seleciona nome de arquivo	
1.3 - Sistema salva arquivo selecionado	
1.4 - Caso de uso editar LED	
Cenário Alternativo	
No item 1.2, caso o usuário selecione um arquivo existente	
2.1 - Sistema exibe mensagem “Arquivo existente, sobrescrever”	
2.2 - Caso o usuário clique OK	
2.2.1 – Item 1.3	
2.3 - Caso o usuário clique Cancelar	
2.3.1 – Item 1.2	

Quadro 8 – Caso de uso salvar LED

Caso de uso:	**Visualizar LED**
Cenário principal	
1.1 - Sistema salva arquivo “auto”	
1.2 - Sistema executa programa “exibir.exe”	
1.3 - Sistema exibe diálogo em edição	
1.4 - Usuário fecha diálogo	
1.5 - Caso de uso Editar LED	

Quadro 9 – Caso de uso visualizar LED

A tela de edição é mostrada como exemplo na Figura 8 e a sua visualização, demonstrada na Figura 9.

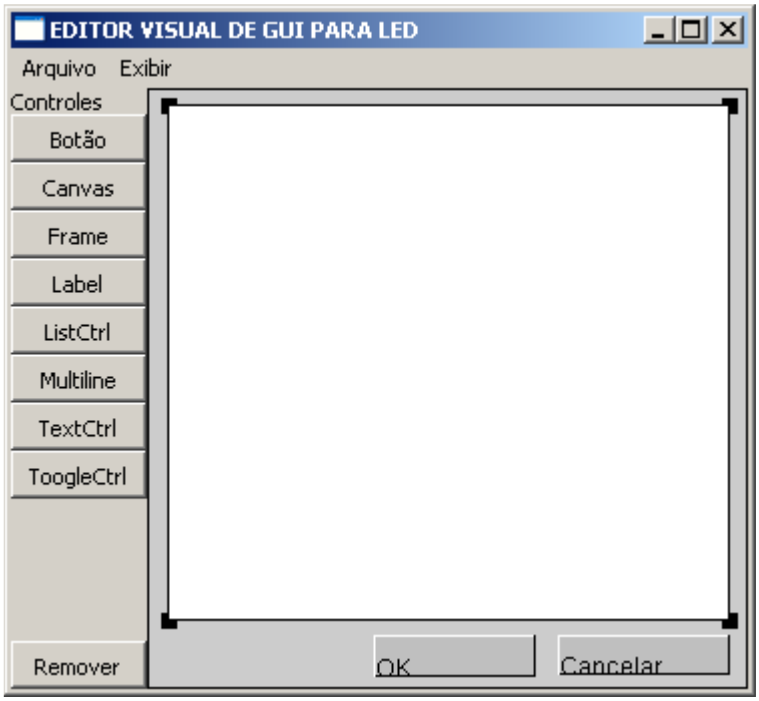


Figura 8 – Exemplo de tela em edição

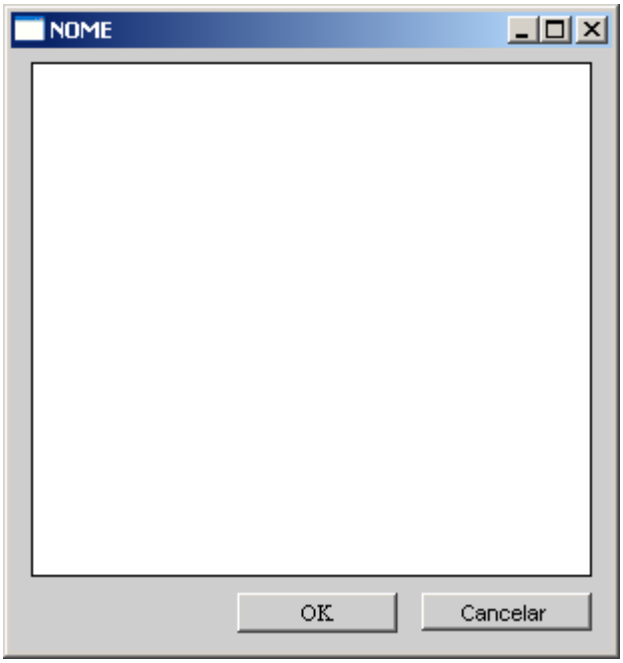


Figura 9 – Exemplo de visualização de arquivo LED

3.2.2 Diagrama de classes

O diagrama de classes foi simplificado tendo sido excluídos os atributos e métodos para a sua melhor visualização. As classes são explicadas uma a uma. Este diagrama é

mostrado na Figura 10.

O protótipo foi desenvolvido utilizando os conceitos, de orientação a objetos. Todos os controles ou elementos de diálogo são derivados de uma classe principal, a classe `CBaseCtrl`. Essa classe implementa os principais métodos de interação com o usuário.

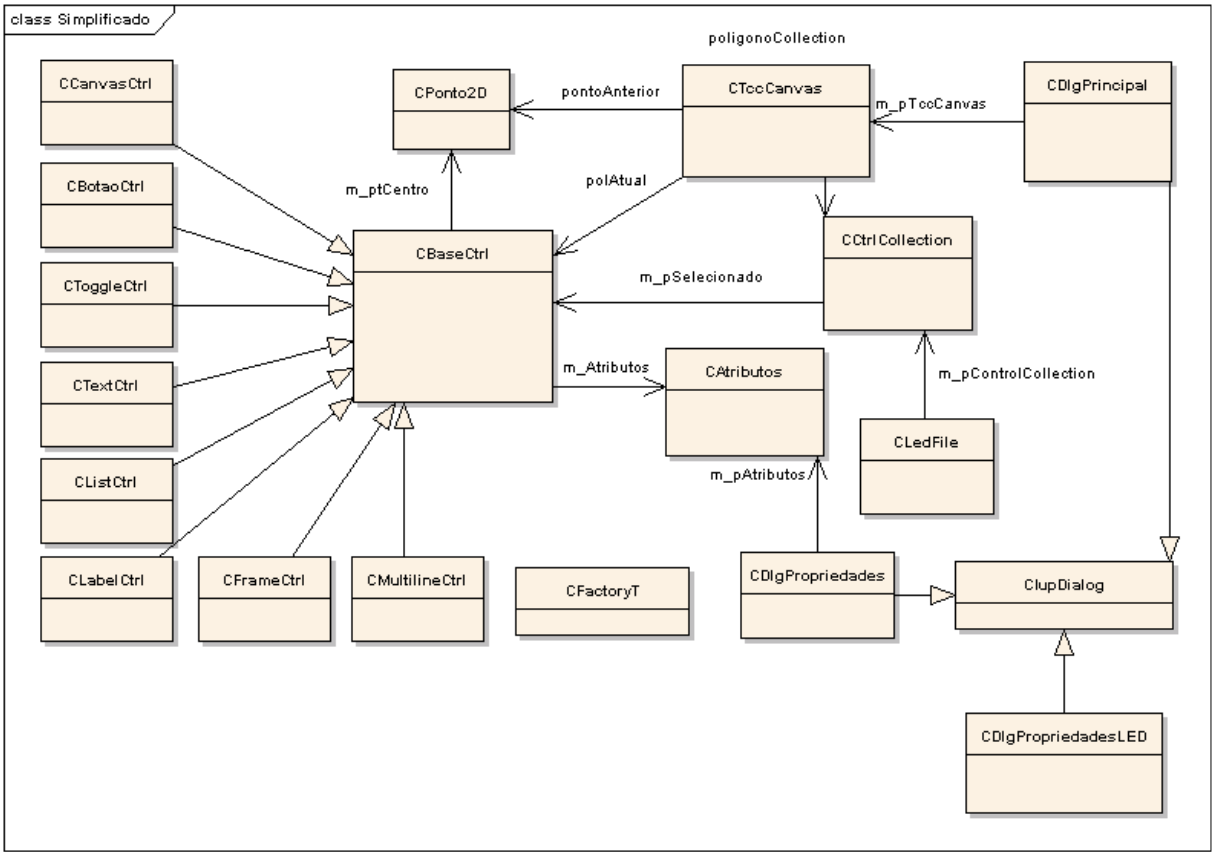


Figura 10 – Diagrama de classes simplificado

3.2.2.1 Classe CPonto2D

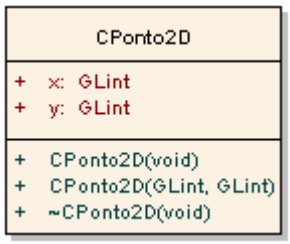


Figura 11 – Classe `CPonto2D`

A classe `CPonto2D` foi criada para o tratamento dos pontos em tela. Essa classe possui os atributos inteiros `x` e `y`, utilizados para posicionamento em tela. A classe possui três construtores, um que recebe como parâmetro as coordenadas inteiras, `x` e `y`; outro que recebe um objeto `CPonto2D`; e por último o construtor sem parâmetros.

3.2.2.2 Classe CAtributos

Figura 12 – Classe `CAtributos`

A classe `CAtributos` foi criada para o tratamento dos atributos em comum dos controles. A classe possui como membro, os atributos `ACTIVE`, `BGCOLOR`, `FGCOLOR`, `FONT`, `NOME`, `SIZE`, `TIP`, `TITLE`, `VALUE`, `VISIBLE`, todos do tipo "`char*`". Possui ainda os atributos `CX` e `CY`, que são atributos inteiros.

Os atributos `ACTIVE` e `VISIBLE` podem receber as *strings* "YES" ou "NO". `BGCOLOR` e `FGCOLOR` recebem uma string contendo as cores do *background* e *foreground*, formatadas da seguinte maneira: "R G B", onde R, G e B devem ser números inteiros de 0 a 255.

O atributo `FONT` recebe uma *string* com a formatação da fonte do controle, um exemplo para essa formatação é: "Times, Bold 18".

Os atributos `TIP`, `VALUE`, `TITLE` e `NOME` recebem qualquer *string*. O atributo `SIZE`, também recebe uma *string*, essa *string* é formatada da seguinte maneira, "`wXh`" onde `w` é o tamanho do comprimento do controle e `h` o tamanho da largura do controle.

3.2.2.3 Classe CBaseCtrl

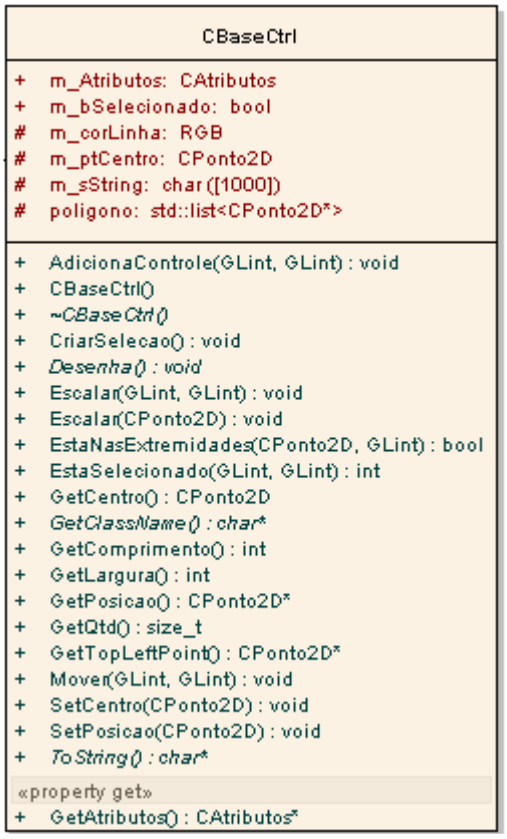


Figura 13 – Classe `CBaseCtrl`

É na classe `CBaseCtrl` onde são tratados os métodos, funções e eventos dos elementos de controle. São atributos da classe `CBaseCtrl`:

a) `m_bSelecionado`: atributo booleano utilizado para dizer se o elemento do diálogo está ou não selecionado;
b) `m_Atributos`: atributo do tipo `CAtributos` utilizado para guardar os atributos dos controles;
c) `m_ptCentro`: atributo do tipo `CPonto2D` utilizado para definir o centro do controle ao aumentar e diminuir o controle;
d) `m_sString`: atributo utilizado no método `ToString()`;
e) `polígono`: atributo do tipo lista de `CPonto2D`, este atributo armazena as coordenadas X e Y dos pontos do controle.

Os métodos da classe `CBaseCtrl` são:

a) `AdicionaControle(GLint x, GLint y)` : método utilizado para a criação de um controle nas coordenadas X e Y da tela, esse método cria os 4 pontos de um controle, a partir do ponto superior esquerdo;

b) `CriarSeleção()`: método que desenha a seleção do controle, as caixas em volta dos pontos do retângulo do controle, exemplificado na Figura 14;
c) `Desenha()`: método utilizado pelas classes herdadas de `CBaseCtrl`;
d) `Escalar(Cponto2D)`: método utilizado para redimensionar um controle selecionado, onde o parâmetro passado representa a diferença ao ponto anterior;
e) `EstaNasExtremidades()`: método booleano utilizado para verificar se o ponteiro do mouse está em um dos quatro cantos do controle;
f) `EstaSelecionado()`: retorna 1 se o controle estiver selecionado e 0 se o controle não estiver selecionado;
g) `GetCentro()`: método utilizado para retornar o atributo `m_ptCentro`;
h) `GetClassName()`: método utilizado pelas classes herdadas de `CBaseCtrl`;
i) `GetComprimento()`: método que calcula o comprimento do controle, X do segundo ponto do controle menos o X do primeiro ponto do controle;
j) `GetLargura()`: método que efetua o calculo da largura do controle, Y do ultimo ponto do controle menos o Y do primeiro ponto do controle;
k) `GetPosição()`: método que retorna a posição concreta do controle em relação a tela;
l) `GetTopLeftPoint()`: método de retorno do primeiro ponto do controle;
m) `Mover(GLint, GLint)`: método utilizado para a movimentação do controle na tela, onde os parâmetros do método são o ponto x e y da diferença do ultimo ponto;
n) `SetCentro(CPonto2D)`: método para setar o valor do atributo `m_ptCentro`;
o) `SetPosicao(CPonto2D)`: método utilizado para setar a posição do controle na tela, altera todos os pontos do controle em relação ao parâmetro passado;
p) `ToString()`: método utilizado para montar o texto que será salvo no arquivo LED, esse método pega e monta os atributos do controle.

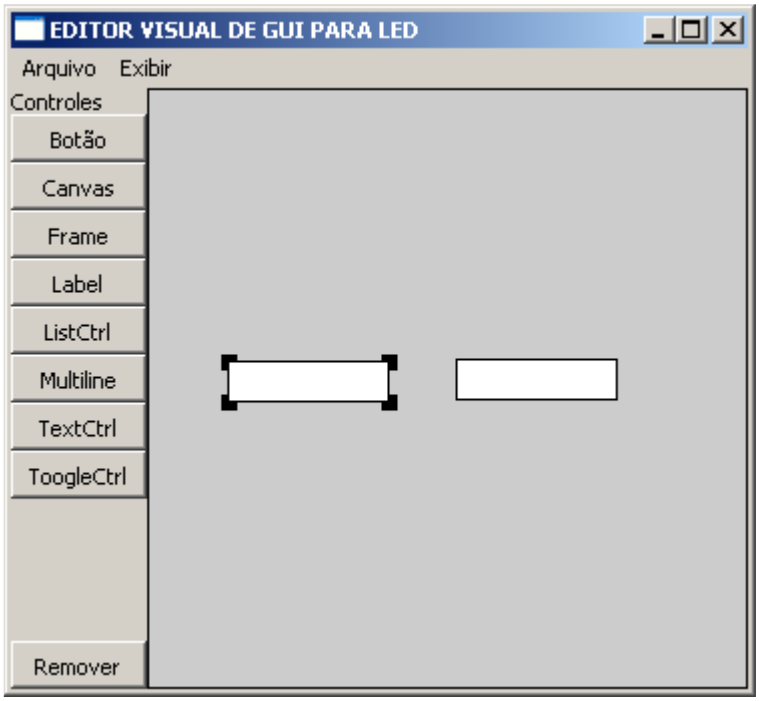


Figura 14 – Caixa de seleção de um controle

3.2.2.4 Classes derivadas de CBaseCtrl

Todas as classes que derivam de `CBaseCtrl` implementam três métodos, são eles:

a) `Desenha()`: método utilizado para desenhar o tipo de controle na tela;

b) `ToString()`: monta o texto com os atributos do controle para salvar no arquivo LED;

c) `GetClassName()`: método utilizado pelas classes derivadas que retorna o nome, em LED, do tipo de controle criado.

As classes que derivam de `CBaseCtrl` são: `CBotaoCtrl`; `CCanvasCtrl`; `CFrameCtrl`; `CLabelCtrl`; `CListCtrl`; `CMultilineCtrl`; `CTextCtrl`; `CToggleCtrl`.

3.2.2.5 Classe CFactoryT

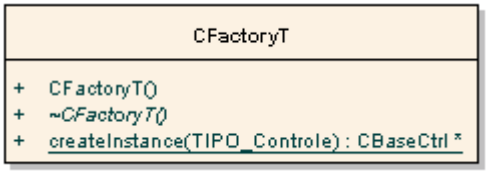


Figura 15 – Classe `CFactoryT`

A classe `CFactoryT` é a classe responsável pela criação dos controles, essa classe possui o método estático `createInstance(TIPO_Controle)`, onde o `TIPO_Controle` é uma enumeração dos tipos de controle que podem ser criados pelo protótipo. São eles: botão, *canvas*, *frame*, *label*, *list control*, *multiline control*, *text control*, e *toggle control*.

De acordo com o parâmetro passado, são criados objetos de controles derivados da classe `CBaseCtrl`.

3.2.2.6 Classe CCtrlCollection

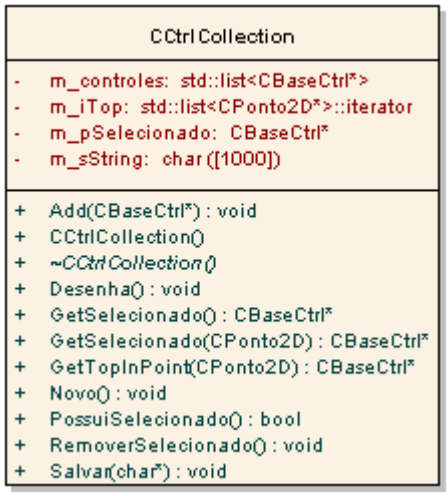


Figura 16 – Classe `CCtrlCollection`

Essa classe é responsável pelo armazenamento dos controles de tela. A classe possui os seguintes atributos:

a) `m_controles`: atributo do tipo lista, que possui os controles de tela;
b) `m_pSelecionado`: atributo que mantém o ponteiro do controle selecionado;
c) `m_sString`: atributo utilizado para gerar o arquivo LED.

Os métodos implementados pela classe `CCtrlCollection` são:

a) `Add(CBaseCtrl*)`: método utilizado para adicionar um controle na lista `m_controles`;
b) `Desenha()`: método que percorre a lista de controles, e chama o método desenha de todos os controles;
c) `GetSelecionado()`: retorna o `m_pSelecionado`;
d) `GetSelecionado(CPonto2D)`: método utilizado para chamar o método `GetTopInPoint` e setar os atributos `m_pSelecionado`, da classe e o atributo do

controle `m_bSelecionado`;

e) `GetTopInPoint(CPonto2D)`: método que percorre todos os controles da lista, verificando se o ponto passado como parâmetro está dentro de um desses controles;

f) `Novo()`: método utilizado para zerar a lista, e remover os controles da tela;

g) `PossuiSelecionado()`: método booleano que retorna verdadeiro caso `m_pSelecionado` for diferente de *NULL* e falso se igual a *NULL*;

h) `RemoverSelecionado()`: método utilizado para remover o controle selecionado da lista e tela;

i) `Salvar()`: método utilizado para salvar o arquivo LED, montando os atributos e controles de acordo com o método `ToString()` de cada controle da lista.

3.2.2.7 Classe CTccCanvas

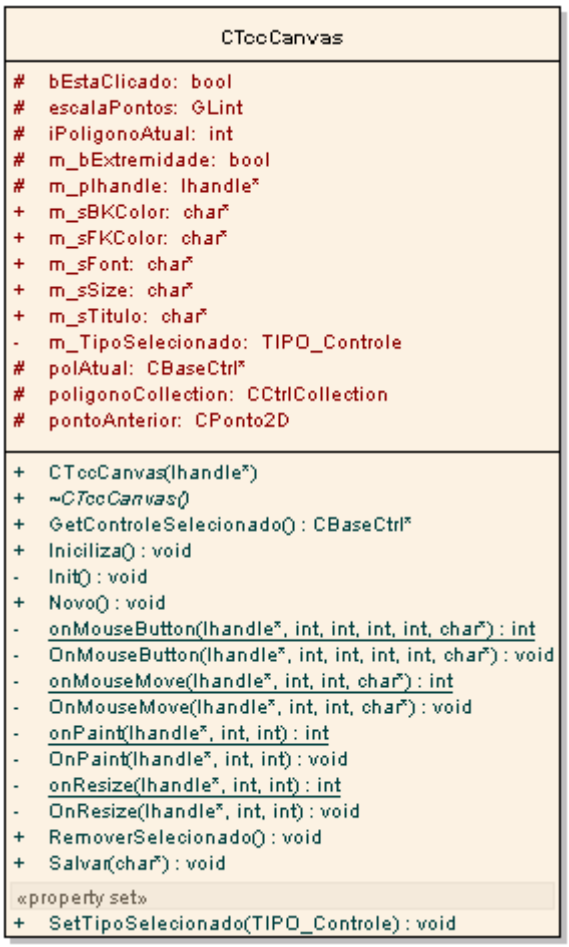


Figura 17 – Classe `CTccCanvas`

A classe `CTccCanvas` é a classe onde são desenhados os controles, é a classe que representa do diálogo a ser criado. Os atributos da classe `CTccCanvas` são:

a) `bEstaClicado`: atributo do tipo booleano para saber se o botão esquerdo do mouse está clicado;
b) `escalaPontos`: atributo utilizado para a criação dos pontos de seleção do controle;
c) `m_bExtremidade`: atributo utilizado para saber se o *mouse* está nas extremidades de um controle, é utilizado para aumentar e diminuir o controle;
d) `m_pIhandle`: esse atributo é utilizado para guardar o `Ihandle` do canvas, *handle* para o `IupCanvas`, da *toolkit* IUP;
e) `m_sBKColor`: atributo que armazena a cor de fundo do diálogo;
f) `m_sFKColor`: atributo que armazena a cor do *foreground* do diálogo;
g) `m_sFont`: atributo utilizado para armazenar a fonte do diálogo, e seus controles, caso os controles do diálogo não possuam o atributo `FONT`, em LED, o atributo `FONT` do diálogo vale para todos os seus controles;
h) `m_sSize`: atributo que armazena o tamanho do diálogo, "COMPRIMENTOxLARGURA";
i) `m_sTitulo`: o atributo `m_sTitulo` armazena o titulo do diálogo;
j) `m_TipoSelecionado`: atributo do tipo `TIPO_Controle`, utilizado para criar um controle;
k) `polAtual`: é o atributo utilizado para saber qual o ponteiro do controle selecionado, para edição, movimentação e ou para aumentar e diminuir esse controle;
l) `poligonoCollection`: atributo do tipo `CCtrlCollection`, utilizado para armazenar os controles do diálogo;
m) `pontoAnterior`: atributo do tipo `CPonto2D` utilizado para armazenar o ponto anterior ao movimento do *mouse*, para mover, aumentar ou diminuir um controle.

A classe `CTccCanvas` possui um construtor que recebe como parâmetro o `Ihandle` do `IupCanvas`, esse `Ihandle` é armazenado no atributo `m_Ihandle`.

Os métodos implementados pela classe `CTccCanvas` são:

a) `GetControleSelecionado()`: esse método retorna o controle selecionado da *collection*, utilizando o atributo `m_ctrlCollection` para chamar o método `GetSelecionado()`;
b) `Inicializa()`: método utilizado para setar as métodos estáticos de *callback* de movimento de *mouse*, clique de botão do *mouse*, pintura de tela e alteração de

tamanho do canvas;

c) `Init()`: método utilizado para setar a cor de fundo do canvas;

d) `Novo()`: utilizado para chamar o método `Novo()` da *collection*;

e) `onMouseButton(...)`: método estático utilizado para chamadas *callback* dos eventos de botão do mouse, este método estático chama o método `OnMouseButton()` não estático da classe `CTccCanvas`, para visualização dos membros da classe;

f) `OnMouseButton()`: método que implementa os eventos de botão do mouse, esse método pode criar um controle, selecionar um controle e aumentar ou diminuir o tamanho de um controle;

g) `onMouseMove(...)`: método estático utilizado para chamadas *callback* dos eventos de movimentação do mouse, este método estático chama o método `OnMouseMove()` não estático da classe `CTccCanvas`, para visualização dos membros da classe;

h) `OnMouseMove()`: método utilizado para implementação dos eventos de movimentação do mouse, podem alterar o tamanho de um controle, e ou movimentar o controle na tela;

i) `onPaint(...)`: método estático utilizado para chamadas *callback* dos eventos de pintura da tela, este método estático chama o método `OnPaint()` não estático da classe `CTccCanvas`, para visualização dos membros da classe;

j) `OnPaint()`: método utilizado para desenhar os controles na tela, através do método `Desenha()` da *collection*;

k) `onResize(...)`:método estático utilizado para chamadas *callback* dos eventos de tamanho de tela, este método estático chama o método `OnResize()` não estático da classe `CTccCanvas`, para visualização dos membros da classe;

l) `OnResize()`: método utilizado para implementação dos eventos de *resize* da tela, altera o `ViewPort` do canvas;

m) `Remover()`: método utilizado para remover um controle selecionado, chamando o método Remover da classe `CCtrlCollection`;

n) `Salvar()`: método utilizado para salvar o arquivo LED, chama o método `Salvar()` da classe `CCtrlCollection`;

o) `SetTipoSelecionado()`: método utilizado para setar o valor da variável `m_TipoControle`, quando um botão do diálogo principal é clicado.

3.2.2.8 Classe CIupDialog

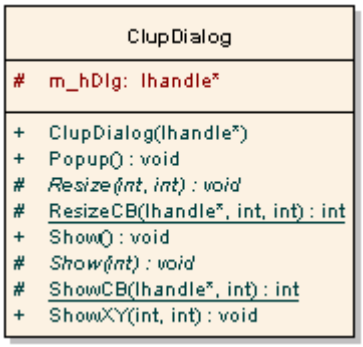


Figura 18 – Classe `CIupDialog`

A classe `CIupDialog` é a classe base dos diálogos do protótipo, o atributo da classe `CIupDialog` é `m_hDlg`, esse atributo armazena o `Ihandle` de um diálogo, criado pela função `IupDialog()` da *toolkit* IUP.

A classe implementa o construtor que recebe como parâmetro o `Ihandle` do diálogo criado pela IUP, e esse parâmetro é armazenado no atributo `m_hDlg`.

Os métodos implementados por essa classe são:

a) `Popup()`: método utilizado para chamar a função `IupPopup()`, da *toolkit* IUP;
b) `Resize()`: método que trata o evento de aumento e diminuição do tamanho do diálogo;
c) `ResizeCB()`: método estático utilizado para receber o *callback* de *resize* do diálogo, esse método chama o método `Resize()`, da classe para a visualização dos seus membros;
d) `Show()`: método utilizado para visualização do diálogo, este método chama a função `IupShow()` da *toolkit* IUP;
e) `ShowCB()`: método estático utilizado para receber o *callback* da visualização do diálogo, esse método chama o método `Show()`, da classe para a visualização dos seus membros;
f) `ShowXY(int, int)`: método utilizado para mostrar o diálogo nas posições x, e y de tela, passadas como parâmetro.

3.2.2.9 Classe CDlgPropriedadesLed

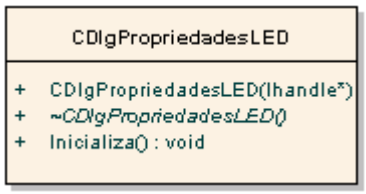


Figura 19 – Classe `CDlgPrpriedadesLed`

A classe `CDlgPrpriedadesLed` é derivada da classe `CIupDlg`. É a classe utilizada para a edição dos atributos de um diálogo. Os atributos de um diálogo para o protótipo são: `NOME`, `FGCOLOR`, `BKCOLOR`, `SIZE`, `FONT`.

Essa classe implementa o método `Inicializa()`, nesse método seta os valores dos *edits* e controles desse diálogo. No seu destrutor, os valores são atualizados nos atributos da classe `CTccCanvas`.

3.2.2.10 Classe CDlgPropriedades

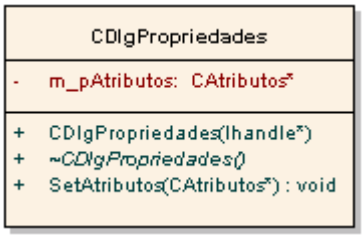


Figura 20 – Classe `CDlgPropriedades`

A classe `CDlgPropriedades` é derivada da classe `CIupDlg`. É a classe responsável pela edição dos atributos de um controle selecionado. A classe possui um ponteiro para um objeto da classe `CAtributos`, pertencente ao controle selecionado.

Ela possui um método `SetAtributos(CAtributos*)`, que recebe os atributos do controle e monta os valores nos *edits* e controles desse diálogo. No destrutor da classe o objeto `CAtributo` recebe os valores relacionados a cada atributo.

3.2.2.11 Classe CDlgPrincipal

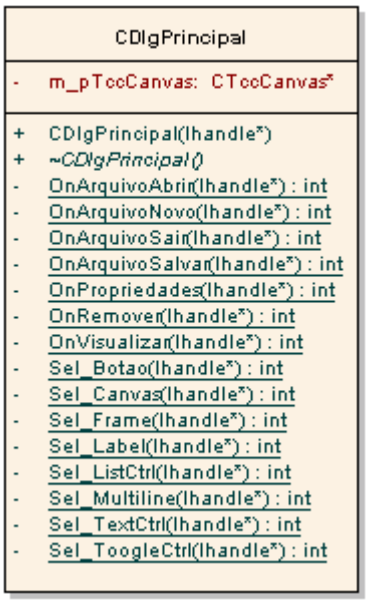


Figura 21 – Classe `CDlgPrincipal`

A Classe `CDlgPrincipal` também é derivada da classe `CIupDlg`. Essa é a classe da tela principal do protótipo. A classe possui um atributo o `m_pTccCanvas`, que é um objeto do tipo `CTccCanvas`.

Os métodos dessa classe são praticamente todos os eventos de botão e *menu* do protótipo. São estes métodos:

a) `OnArquivoAbrir()`: método responsável pelo evento de menu Arquivo, Abrir;
b) `OnArquivoNovo()`: método responsável pelo evento de menu Arquivo, Novo;
c) `OnArquivoSalvar()`: método responsável pelo evento de menu Arquivo, Salvar;
d) `OnArquivoSair()`: método responsável pelo evento de menu Arquivo, Sair;
e) `OnPropriedades()`:método responsável pelo evento de menu Exibir, Propriedades, nesse método é visualizado o diálogo de propriedades de um controle, se houver algum selecionado, ou caso não haja controle selecionado, é visualizado o diálogo de propriedades do LED;
f) `OnRemover()`: método responsável pela remoção de um controle selecionado;
g) `OnVisualizar()`: método criado para a pré-visualização de um diálogo em edição, este método chama outro aplicativo criado, o Exibir.exe, esse aplicativo carrega o arquivo salvo automaticamente pelo protótipo e exibe a tela criada;
h) `Sel_Botao()`: evento de seleção do tipo de controle Botão para a criação do mesmo na área de edição;

i) `Sel_Canvas()`: evento de seleção do tipo de controle Canvas para a criação do mesmo na área de edição;
j) `Sel_Frame()`: evento de seleção do tipo de controle Frame para a criação do mesmo na área de edição;
k) `Sel_Label()`: evento de seleção do tipo de controle Label para a criação do mesmo na área de edição;
l) `Sel_ListCtrl()`: evento de seleção do tipo de controle ListCtrl para a criação do mesmo na área de edição;
m) `Sel_Multiline()`: evento de seleção do tipo de controle Multiline para a criação do mesmo na área de edição;
n) `Sel_TextCtrl()`: evento de seleção do tipo de controle TextCtrl para a criação do mesmo na área de edição;
o) `Sel_ToggleCtrl()`: evento de seleção do tipo de controle ToggleCtrl para a criação do mesmo na área de edição.

3.2.3 Diagrama de seqüência

Como o usuário é o ator principal de um editor, todas as funcionalidades do sistema partem do ator, gerando eventos de clique de *mouse*, movimentação do *mouse*, clique de botão, seleção de um item de *menu*.

Na Figura 22, é apresentado o diagrama de seqüência das classes implementadas. Com uma visão macro de como o sistema funciona.

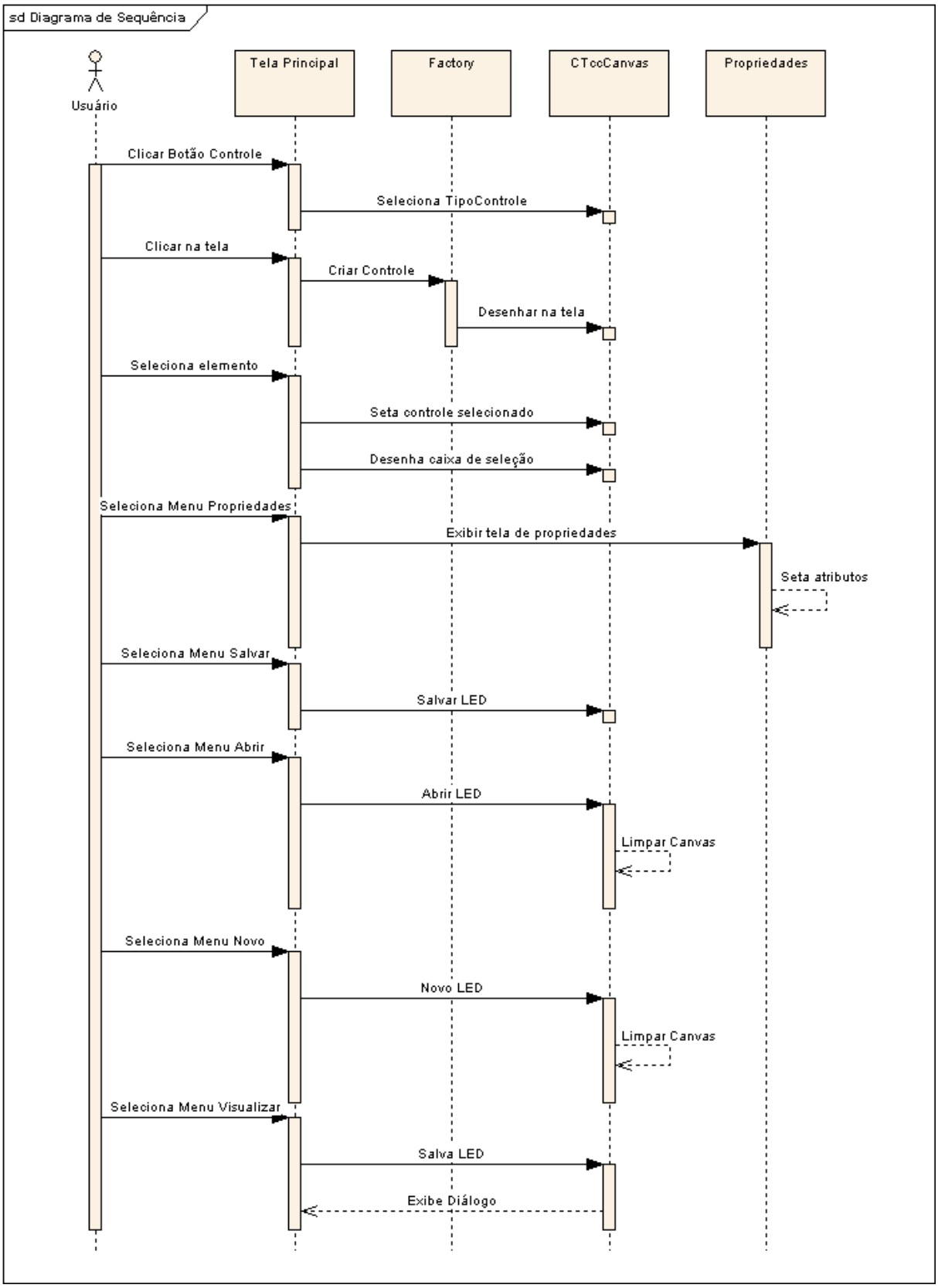


Figura 22 – Diagrama de seqüência

3.3 IMPLEMENTAÇÃO

Este item apresenta considerações sobre a implementação do protótipo, como técnicas, ferramentas utilizadas e operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

Para o desenvolvimento deste trabalho foram utilizadas a linguagem Visual C++, com o ambiente de programação Visual Studio 2003, a *toolkit* IUP/LED distribuída gratuitamente, e a biblioteca gráfica OpenGL.

Para o funcionamento da toolkit IUP/LED, faz-se necessário o *download* das bibliotecas, e arquivos da *toolkit*, e estes devem ser incluídos ao projeto. O trabalho possui a representação no plano das coordenadas x e y conforme mostrada na Figura 23.

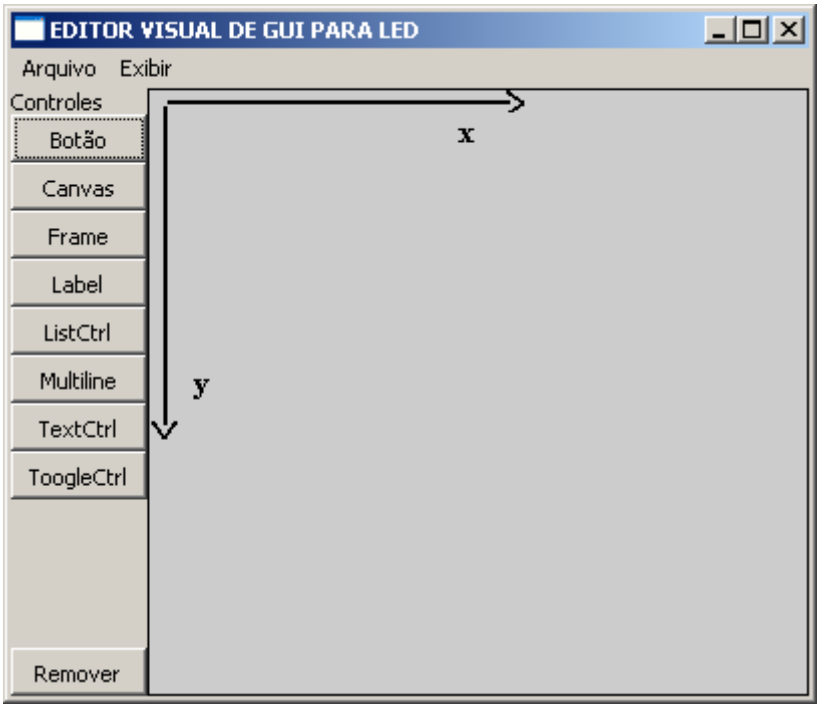


Figura 23 – Eixo de coordenadas x e y

A primeira parte a ser desenvolvida no protótipo foi o arquivo LED da tela principal, “interface.LED”. Este arquivo utiliza o conceito de leiaute abstrato. As telas de propriedades do diálogo e controle também estão dentro do arquivo “interface.LED”. Para o funcionamento do protótipo, o arquivo deverá estar junto à aplicação.

Com a interface principal definida, foram criados então os eventos de botão, para a

seleção de tipo de controle a ser criado e os eventos de mouse para a criação destes controles.

Foram criados os métodos para seleção, e movimentação de um controle e ainda o método de função escalar, utilizado para aumentar ou diminuir o elemento de diálogo. Ao concluir todas as funções de interação, foram implementados os métodos para salvar o arquivo LED em disco. A definição da interface principal do programa em LED é exibida no Quadro 10.

```
Interface = DIALOG[MENU = main_menu, TITLE="EDITOR VISUAL DE GUI PARA
LED"]
(
   HBOX
      (
         VBOX [SIZE=20]
         (
           LABEL( "Controles" ),
           VBOX
            (
                  BUTTON [SIZE=55x15, TIP="Clique para inserir um
Botão"]( "Botão",Sel_Botao ),
      BUTTON [SIZE=55x15, TIP="Clique para inserir um Canvas"](
"Canvas",Sel_Canvas ),
      BUTTON [SIZE=55x15, TIP="Clique para inserir um Frame"](
"Frame",Sel_Frame ),
      BUTTON [SIZE=55x15, TIP="Clique para inserir um Label"](
"Label",Sel_Label ),
      BUTTON [SIZE=55x15, TIP="Clique para inserir um ListCtrl"](
"ListCtrl",Sel_ListCtrl ),
      BUTTON [SIZE=55x15, TIP="Clique para inserir um Multiline"](
"Multiline",Sel_Multiline ),
      BUTTON [SIZE=55x15, TIP="Clique para inserir um TextCtrl"](
"TextCtrl",Sel_TextCtrl ),
      BUTTON [SIZE=55x15, TIP="Clique para inserir um ToogleCtrl"](
"ToogleCtrl",Sel_ToogleCtrl )
            ),
           VBOX
           (
               FILL(),
      BUTTON [SIZE=55x15, TIP="Remover um controle selecionado"](
"Remover",OnRemover)
            )
      ),
      VBOX
      (
         meucanvas
      )
   )
)
```

Quadro 10 – Diálogo principal do protótipo utilizando LED

3.3.2 Operacionalidade da implementação

O protótipo gerado salva, abre e cria um novo arquivo LED, possui a edição de controles e sua posição e tamanho atualizados com a interação do usuário diretamente com o elemento de interface. Possui ainda a edição das propriedades desses controles, e a edição das

propriedades de um diálogo.

Para executar o programa, deve-se ter na mesma pasta os arquivos: cd.dll, cdiup.dll, glut32.dll, iup.dll, iupcontrols.dll e iupgl.dll da *toolkit* IUP/LED, exibir.exe, interface.led e EVIGU-LED.exe, arquivos implementados pelo presente trabalho.

Ao abrir o arquivo testeideia.exe, o sistema apresentará a tela principal do programa, pronta para a edição de um arquivo LED. A tela inicial é exibida na Figura 24.

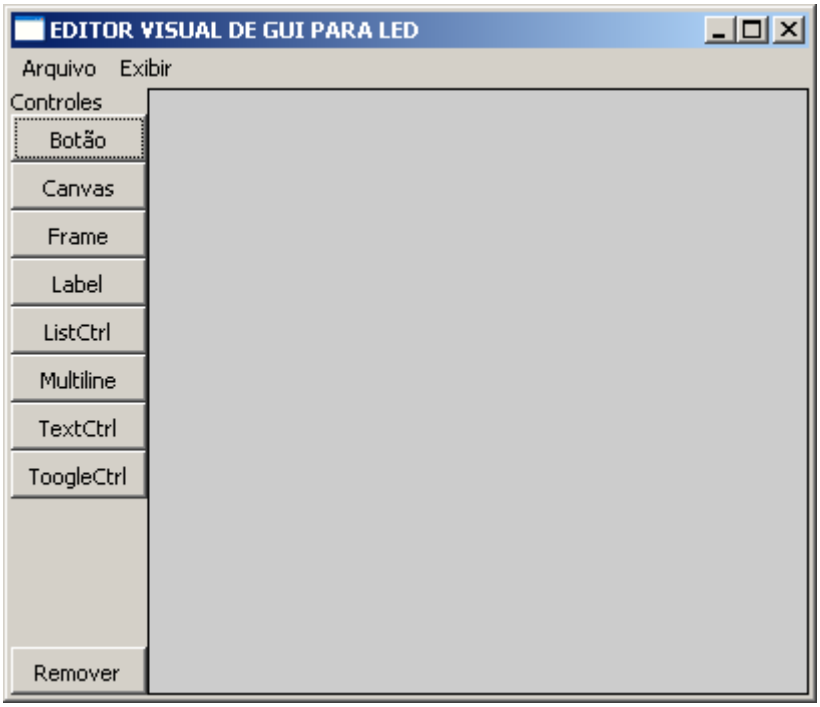


Figura 24 – Tela inicial

O usuário está apto a clicar em um dos botões do lado esquerdo da tela para a seleção do controle a ser criado, e posteriormente, clicando no canvas de edição, fará com que o sistema desenhe e crie o controle para interação.

Ao clicar no botão com a descrição Botão, o usuário estará selecionando um controle do tipo botão para inserir no canvas de edição. A Figura 25 mostra um botão inserido no canvas de edição.

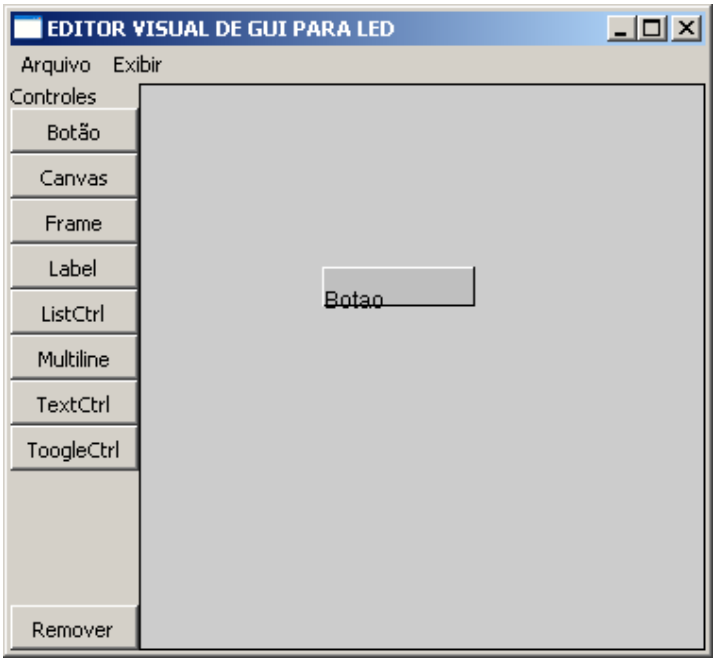


Figura 25 – Tela com elemento botão

O mesmo acontecerá caso o usuário escolha o botão com a descrição `Canvas`, `TextCtrl`, `Multiline` e `ListCtrl`, mas o tipo de elemento de diálogo criado será o correspondente ao botão clicado. A Figura 26 ilustra a representação gráfica de um `canvas`, um `textctrl`, um `multiline` e um `listctrl`, estes controles possuem a mesma representação gráfica para o protótipo.

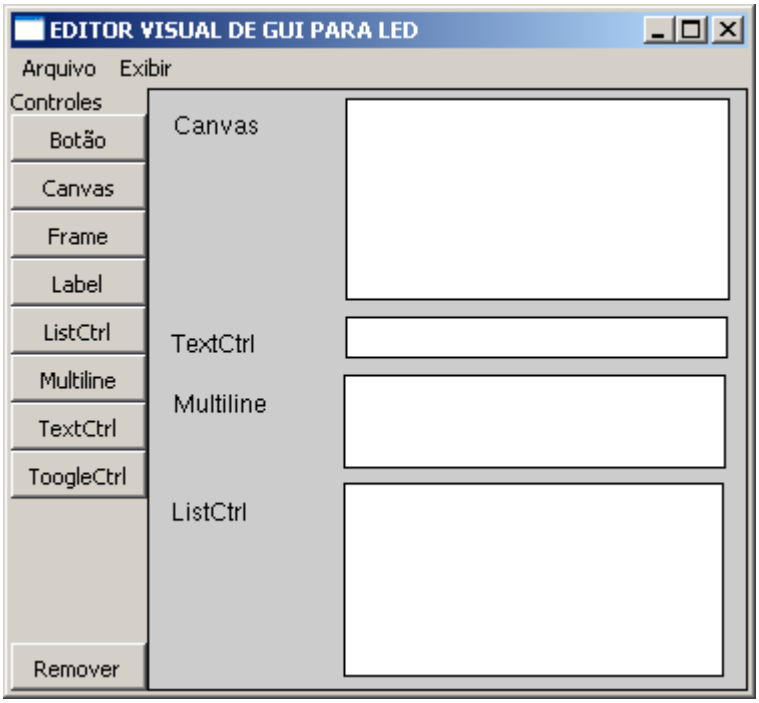


Figura 26 – Tela com elementos `Canvas`, `TextCtrl`, `Multiline` e `ListCtrl`

O resultado da exibição da tela acima está apresentado na Figura 27. Para visualizar um diálogo em edição, o usuário deve clicar no *menu* Exibir na opção Visualizar Diálogo.

Quando essa opção é selecionada o sistema salva o arquivo “auto” na mesma pasta do programa, e chama o programa “Exibir.exe”, que abre o diálogo para a visualização.

A exibição do restante dos controles, `Frame`, `Label` e `ToggleCtrl`, é apresentado na Figura 28. A visualização desses controles é mostrado na Figura 29.

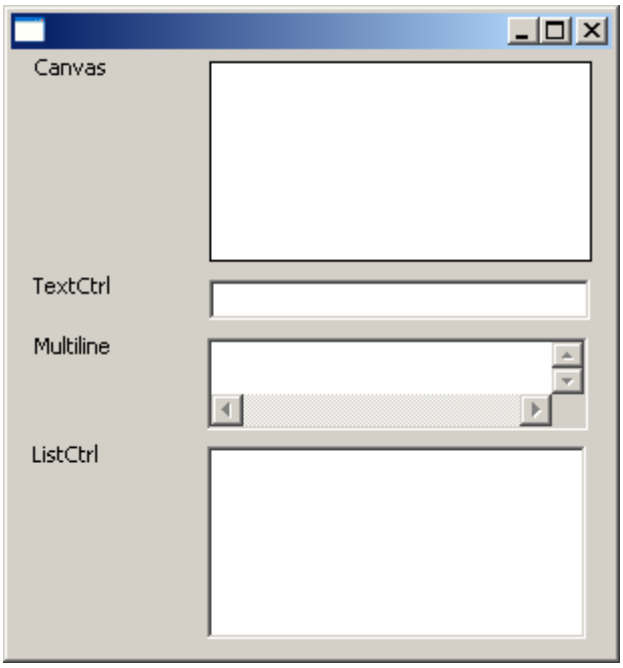


Figura 27 – Tela de visualização com os elementos `Canvas`, `TextCtrl`, `Multiline` e `ListCtrl`

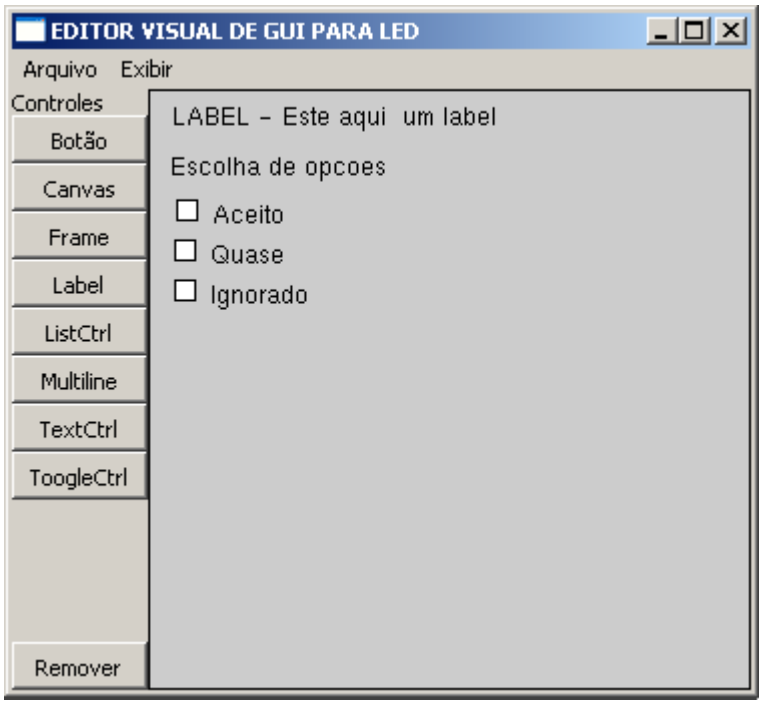


Figura 28 - Tela de edição com os elementos `Label` e `ToogleCtrl`

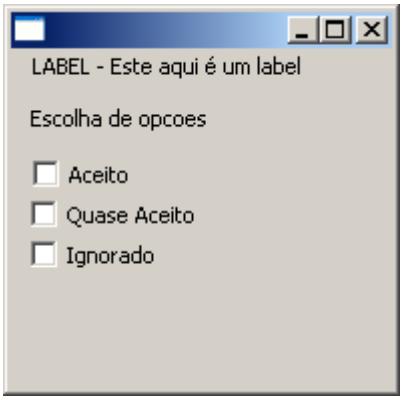


Figura 29 – Visualização de tela com os elementos `Label` e `ToggleCtrl`

Todos os elementos do diálogo possuem os mesmos atributos. Para o protótipo os atributos em comum aos controles podem ser alterados selecionando um elemento do diálogo em edição, clicando no *menu* Exibir e em seguida na opção Propriedades. A tela de propriedades de um controle é exibida na Figura 30.

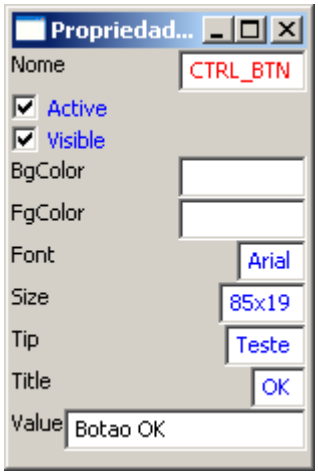


Figura 30 – Tela de edição de propriedades de um controle

Caso não haja nenhum controle selecionado pelo usuário, e o usuário clicar no *menu* Exibir, na opção Propriedades, é exibida a tela de propriedades do diálogo editado. A figura que representa esse diálogo de propriedades é apresentada na Figura 31.

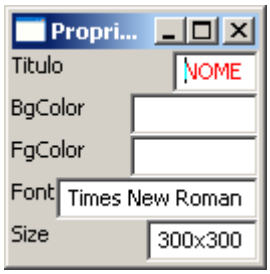


Figura 31 – Propriedades do diálogo em edição

As telas de propriedades de controle e propriedades do diálogo, estão dentro do arquivo “interface.led”. A especificação dessas telas na linguagem de especificação de diálogos são demonstradas no Quadro 11 e 12 respectivamente.

```
Propriedades = DIALOG[TITLE="Propriedades do controle"]
(
 VBOX
 (
  HBOX
  (
   LABEL( "Nome" ),
   Fill(),
   tbNome = TEXT [VALUE="NOME",FGCOLOR="255 0 0"] ( do_nothing )
  ),
  HBOX
  (
   tbActive = TOGGLE [VALUE="ON",FGCOLOR="0 0 255"] ("Active", do_nothing )
  ),
  HBOX
  (
   tbVisible = TOGGLE [VALUE="ON",FGCOLOR="0 0 255"] ("Visible", do_nothing )
  ),
  HBOX
  (
   LABEL( "BgColor" ),
   Fill(),
   tbBgColor = TEXT [VALUE="255,0,0",FGCOLOR="0 0 255"] ( do_nothing )
  ),
  HBOX
  (
   LABEL( "FgColor" ),
   Fill(),
   tbFgColor = TEXT [VALUE="255,0,0",FGCOLOR="0 0 255"] ( do_nothing )
  ),
  HBOX
  (
   LABEL( "Font" ),
   Fill(),
   tbFont = TEXT [VALUE="name:Times New Roman attributes:BOLD size:10",FGCOLOR="0 0
255"] ( do_nothing )
  ),
  HBOX
  (
   LABEL( "Size" ),
   Fill(),
   tbSize = TEXT [VALUE="10x10",FGCOLOR="0 0 255"] ( do_nothing )
  ),
  HBOX
  (
   LABEL( "Tip" ),
   Fill(),
   tbTip = TEXT [VALUE="Explicacao do campo",FGCOLOR="0 0 255"] ( do_nothing )
  ),
  HBOX
  (
```

```
   LABEL( "Title" ),
   Fill(),
   tbTitle = TEXT [VALUE="Titulo",FGCOLOR="0 0 255"] ( do_nothing )
  ),
  HBOX
  (
   LABEL( "Value" ),
   Fill(),
   tbValue = TEXT[EXPAND=HORIZONTAL]( do_nothing )
  )
 )
)
```

Quadro 11 – Especificação do diálogo de propriedades de controles

```
PropriedadesLED = DIALOG[TITLE="Propriedades do diálogo"]
(
 VBOX
 (
  HBOX
  (
   LABEL( "Titulo" ),
   Fill(),
   tbTituloDlg = TEXT [VALUE="NOME",FGCOLOR="255 0 0"] ( do_nothing )
  ),
  HBOX
  (
   LABEL( "BgColor" ),
   Fill(),
   tbBgColorDlg = TEXT [VALUE="",FGCOLOR="0 0 0"] ( do_nothing )
  ),
  HBOX
  (
   LABEL( "FgColor" ),
   Fill(),
   tbFgColorDlg = TEXT [VALUE="",FGCOLOR="0 0 0"] ( do_nothing )
  ),
  HBOX
  (
   LABEL( "Font" ),
   Fill(),
   tbFontDlg = TEXT [VALUE="Times New Roman",FGCOLOR="0 0 0"] ( do_nothing )
  ),
  HBOX
  (
   LABEL( "Size" ),
   Fill(),
   tbSizeDlg = TEXT [VALUE="",FGCOLOR="0 0 0"] ( do_nothing )
  )
 )
)
```

Quadro 12 – Especificação da tela de propriedades do diálogo

Ao clicar no *menu* Arquivo e escolher a opção Novo, o sistema limpa a tela de edição e o usuário poderá começar a sua tela novamente. Clicando no *menu* Arquivo, selecionando a opção Salvar o protótipo exibe a tela de seleção de arquivos. Essa tela é demonstrada na Figura 32. Selecionado o arquivo, o sistema lê os atributos e propriedades dos controles e diálogos em edição e salva o arquivo.

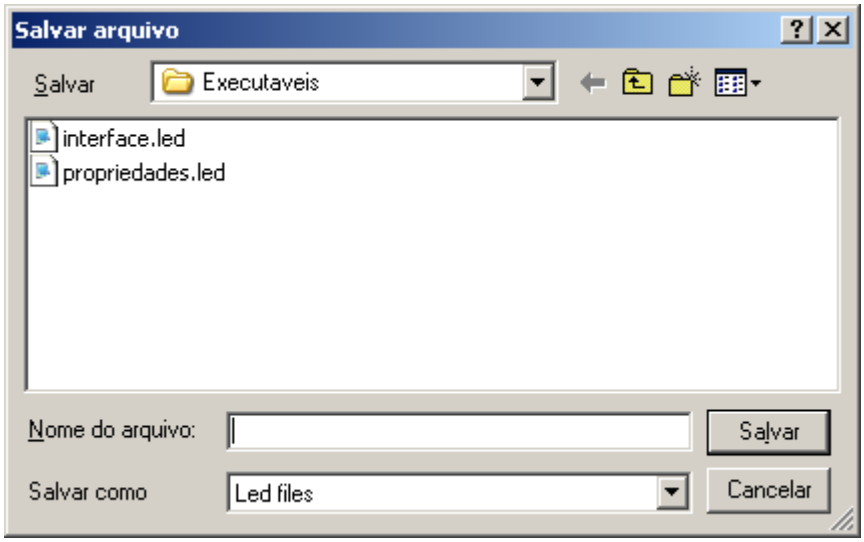


Figura 32 – Tela utilizada para salvar arquivos LED

Para abrir um arquivo do tipo LED, o usuário deve clicar no *menu* Arquivo, e escolher a opção Abrir. A Figura 33 exibe a tela de seleção de arquivos para serem abertos. Ao escolher um arquivo, o sistema monta os objetos de controle e diálogo e desenha os controles para edição na tela. O arquivo “teste.led” é um arquivo salvo pelo protótipo. A especificação do diálogo é exibida no Quadro 13. A Figura 34 mostra o diálogo em edição e a Figura 35 mostra o diálogo sendo visualizado.

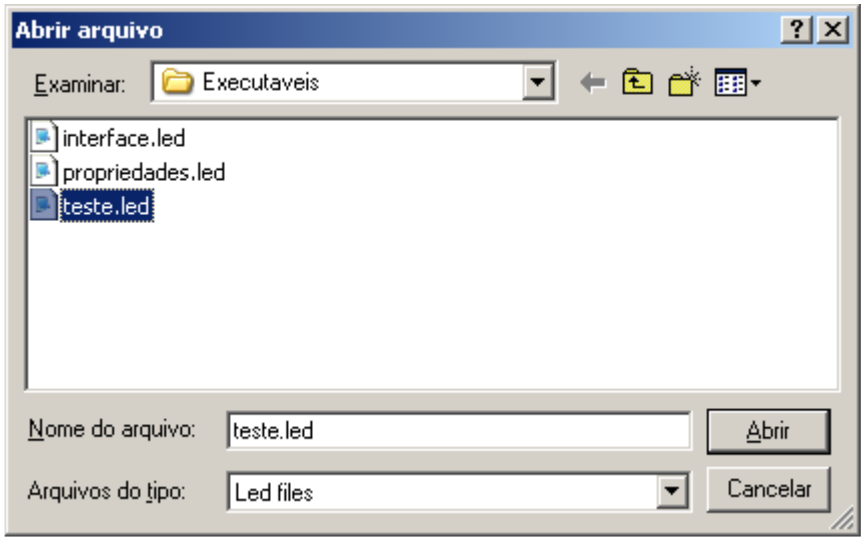


Figura 33 – Tela utilizada para abrir arquivos LED

```
TCCDLG = DIALOG[FONT="Times New Roman",TITLE="Tela de visualização (para
fonte)",BGCOLOR="178 178 178",FGCOLOR="0 0 0"]
(
      CBOX[RASTERSIZE="300x300"]
      (
            CTRL_Criar = BUTTON[ACTIVE = "YES",VISIBLE = "YES",RASTERSIZE =
"80x20",SIZE = "80x20",CLIENTSIZE = "80x20",CX = "207",CY = "157",VALUE =
"Criar"]("Criar",do_nothing),
            Triangulo = TOGGLE[ACTIVE = "YES",VISIBLE = "YES",RASTERSIZE =
"80x20",SIZE = "80x20",CLIENTSIZE = "80x20",CX = "5",CY = "190",TITLE =
"Triangulo",VALUE = "Triangulo"]("Triangulo",do_nothing),
            Quadrado = TOGGLE[ACTIVE = "YES",VISIBLE = "YES",RASTERSIZE =
"80x20",SIZE = "80x20",CLIENTSIZE = "80x20",CX = "5",CY = "173",TITLE =
"Quadrado",VALUE = "Quadrado"]("Quadrado",do_nothing),
            Circulo = TOGGLE[ACTIVE = "YES",VISIBLE = "YES",RASTERSIZE =
"80x20",SIZE = "80x20",CLIENTSIZE = "80x20",CX = "5",CY = "156",TITLE =
"Circulo",VALUE = "Circulo"]("Circulo",do_nothing),
            CTRL = CANVAS[ACTIVE = "YES",VISIBLE = "YES",RASTERSIZE =
"284x141",SIZE = "284x141",CLIENTSIZE = "284x141",CX = "7",CY = "5"](do_nothing),
            CTRL_OK = BUTTON[ACTIVE = "YES",VISIBLE = "YES",RASTERSIZE =
"80x20",SIZE = "80x20",CLIENTSIZE = "80x20",CX = "117",CY = "269",VALUE =
"OK"]("OK",do_nothing),
            CTRL_CANCEL = BUTTON[ACTIVE = "YES",VISIBLE = "YES",RASTERSIZE =
"80x20",SIZE = "80x20",CLIENTSIZE = "80x20",CX = "206",CY = "269",VALUE =
"Cancelar"]("Cancelar",do_nothing)
      )
)
```

Quadro 13 – Linguagem de especificação de diálogos do arquivo “teste.led”

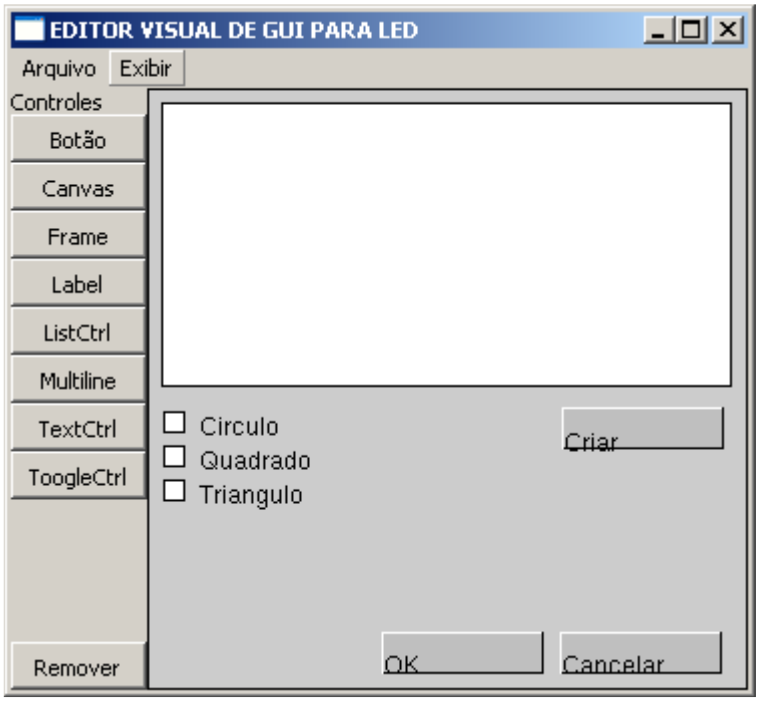


Figura 34 – Tela de edição do arquivo “teste.led”

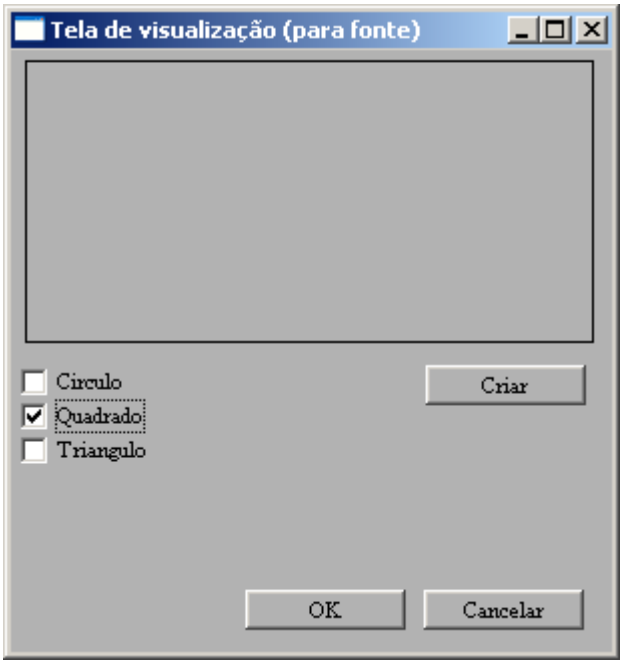


Figura 35 – Tela de visualização do arquivo “teste.led”

3.4 RESULTADOS E DISCUSSÃO

Um dos maiores problemas encontrados na realização deste trabalho foi a indisponibilidade da *toolkit* IUP/LED para a movimentação de controles dinamicamente. Começou-se a desenvolver um primeiro protótipo, utilizando apenas a *toolkit* IUP/LED, ou seja, sem nenhuma outra biblioteca auxiliar. Após alguns testes resolveu-se utilizar a biblioteca gráfica OpenGL, para a criação e movimentação de controles, para continuar assim com a interação do usuário.

Pela dificuldade de se desenvolver uma ferramenta para edição de diálogos em modo de leiaute abstrato, decidiu-se desenvolver um protótipo de ferramenta para a edição de diálogos em modo de leiaute concreto, utilizando coordenadas X e Y e o componente `CBox` da *toolkit* IUP/LED. O posicionamento relativo dos elementos de interface, feito através de leiaute abstrato envolve a resolução de sistemas de restrições geométricas. Trata-se de assunto extremamente complexo, que fugia do escopo do presente trabalho.

O conceito de manipulação direta foi utilizado no protótipo. Ao criar um elemento de diálogo, o usuário pode selecioná-lo, movimentá-lo, aumentar e diminuir o seu tamanho interiativamente. Isto faz com que o usuário tenha a percepção de estar manipulando diretamente o objeto na tela.

O protótipo possui diferenças visuais entre os modos de edição e de visualização. Isto acontece devido à utilização da OpenGL para visualização e manipulação interativa dos objetos gráficos que representavam os controles no editor. No visualizador são exibidos diretamente os controles gerados pela *toolkit* IUP. Como o enfoque do trabalho foi o desenvolvimento de um editor interativo de diálogos e não um visualizador, esta diferença de visualização não foi considerada uma limitação do protótipo.

Uma das facilidades de se utilizar a *toolkit* IUP/LED é o fato de que alguns atributos do diálogo são passados para os controles associados a esse diálogo. São esses atributos: fonte; cor de fundo; cor da fonte do controle. Caso o controle possua um desses atributos configurados, os atributos do diálogo pai são ignorados.

A utilização da *toolkit* IUP juntamente com a Linguagem de Edição de Diálogos (LED) se mostrou uma maneira simples para alteração de telas dinamicamente, por ser carregada em tempo de execução. Se imaginarmos um sistema construído para atender várias aplicações, caso um usuário não precise de alguns itens da tela, esse pode alterar sem problemas, utilizando um editor dinâmico interativo para tal.

Apesar de o trabalho ter sido desenvolvido em ambiente operacional Windows, e implementado utilizando a ferramenta Visual Studio .Net 2003, a *toolkit* IUP/LED pode ser utilizada em outros sistemas operacionais, como também o protótipo de ferramenta implementado, basta que seja re-compilado no sistema operacional desejado.

4 CONCLUSÕES

A idéia inicial do trabalho era utilizar completamente a *toolkit* IUP/LED para a confecção do protótipo. Ao iniciar a pesquisa sobre a *toolkit* IUP/LED, verificou-se que a biblioteca tinha algumas limitações em relação ao movimento de controles dinamicamente, descartando-se então a idéia de se utilizar somente a *toolkit* IUP/LED.

Para a resolução do problema de movimentação de controles, foi desenvolvido uma interface gráfica, utilizando para isso a biblioteca gráfica OpenGL. Para o desenvolvimento do protótipo proposto utilizou-se a ferramenta Microsoft Visual Studio .Net 2003, por ser mais atualizada e de fácil utilização.

A *toolkit* IUP, juntamente com a Linguagem de Especificação de Diálogos (LED) mostrou-se satisfatório ao que a ferramenta se propõe a fazer. Por utilizar cadeias de caracteres para sua funcionalidade, torna-se muito fácil o uso da *toolkit*. O pouco número de funções da *toolkit* IUP/LED, aproximadamente 100, também é uma de suas características marcantes.

O protótipo desenvolvido possui todos os requisitos inicialmente citados, apesar de possuir alguns problemas, quanto à abertura de arquivos LED por não possuir uma análise gramatical para tal. Somente arquivos LED gerados pelo protótipo são abertos pelo mesmo. A visualização de um elemento de diálogo em edição é diferente da visualização pela *toolkit* IUP.

O objetivo do trabalho foi alcançado, a edição visual de elementos de diálogos de forma concreta, a alteração das propriedades e atributos de um controle, e principalmente a geração do arquivo LED estão funcionando corretamente.

4.1 EXTENSÕES

Como sugestões para extensão de trabalho acadêmico seriam: a criação de uma ferramenta visual gráfica interativa para o modelo de leiaute abstrato, utilizando a *toolkit* IUP/LED; um editor textual de arquivos LED, interpretado por uma gramática léxica e sintática da linguagem de especificação de diálogos; e uma ferramenta gráfica interativa utilizando a linguagem LUA, gerando *feedback* dos eventos na visualização do diálogo.

REFERÊNCIAS BIBLIOGRÁFICAS

EBERTS, Ray E. **User interface design**. New Jersey: Prentice Hall, Englewood Cliffs, 1994.

FIGUEIREDO, Luiz H.; GATTASS, Marcelo; PRATES, Raquel. **Especificação de layout abstrato por manipulação direta**. Rio de Janeiro, 1994. Disponível em: <www.tecgraf.puc-rio.br/iup/download/sib94.pdf>. Acesso em: 17 out. 2007.

FOLEY, James D. **Computer graphics**: principles and practice. Michigan: Addison-Wesley Professional, 2003.

FURLAN, Davi. **Modelagem de objetos através da UML** - the unified modeling language. São Paulo: Makron Book, 1998. 329 p.

LEVY, Carlos H. **IUP/LED:** uma ferramenta portátil de interface com usuário. Rio de Janeiro, 1993. Disponível em: <http://www.tecgraf.puc-rio.br/iup/download/levy93.pdf>. Acesso em: 10 out. 2007.

MARCUS, Aaron. **Graphic design for electronic documents and user interfaces**. New York: ACM Press, 1992.

MORENO, Luciano. **Introdução ao design gráfico**. 2007. Disponível em: <http://www.criarweb.com/artigos/711.php>. Acesso em: 17 out. 2007.

PRATES, Raquel O. **Visual LED:** uma ferramenta interativa para geração de interfaces gráficas. Rio de Janeiro, 1994. Disponível em: <http://www.tecgraf.puc-rio.br/iup/download/prates94.pdf>. Acesso em: 10 out. 2007.

REDMOND-PYLE, David; MOORE, Alan. **Graphical user interface design and evaluation (guide):** a practical process. London: Prentice Hall, 1995.

SANTOS, André S. **Um framework para suporte a objetos visuais interativos**. Rio de Janeiro, 1996. Disponível em: <http://www.tecgraf.puc-rio.br/publications/diss_1996_clinio_vix.pdf>. Acesso em: 01 nov. 2007.

SCURI, Antonio E. **IUP portable user interface version 2.6**. Rio de Janeiro, 2007. Disponível em: <http://www.tecgraf.puc-rio.br/iup/en/sys_guide.html#led >. Acesso em: 10 out. 2007.

THIMBLEBY, Harold. **User interface design**. New York: ACM Press; Wokingham: Addison Wesley, 1990.