UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

AMBIENTE PARA CONSTRUÇÃO DE INTERFACES GRÁFICAS DE USUÁRIO COM O SCV

TRABALHO DE GRADUAÇÃO

Cícero Augusto de Lara Pahins

Santa Maria, RS, Brasil

2011

AMBIENTE PARA CONSTRUÇÃO DE INTERFACES GRÁFICAS DE USUÁRIO COM O SCV

por

Cícero Augusto de Lara Pahins

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de
Bacharel em Ciência da Computação

Orientador: Prof. Dr. Cesar Tadeu Pozzer

Trabalho de Graduação N. 325

Santa Maria, RS, Brasil

2011

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

AMBIENTE PARA CONSTRUÇÃO DE INTERFACES GRÁFICAS DE USUÁRIO COM O SCV

elaborado por
Cícero Augusto de Lara Pahins

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Prof. Dr. Cesar Tadeu Pozzer
(Presidente/Orientador)

Profª. Dra. Lisandra Manzoni Fontoura (UFSM)

Profª. Dra. Giliane Bernardi (UFSM)

Santa Maria, 15 de Dezembro de 2011.

“The beginning of wisdom is the statement ’I do not know.’ The person who cannot make that statement is one who will never learn anything. And I have prided myself on my ability to learn.”

THRALL (WARCRAFT)

AGRADECIMENTOS

Agradeço aos meus pais, irmãs e minha família que, com muito carinho e apoio, não mediram esforços para que eu chegasse até esta etapa de minha vida.

Ao professor Cesar Tadeu Pozzer pela paciência na orientação e incentivo que tornaram possível o desenvolvimento deste trabalho e extrapolaram o âmbito acadêmico.

A todos os professores do Curso de Ciência da Computação, que foram tão importantes em minha vida e formação acadêmica.

Aos amigos e colegas, pelo incentivo e pelo apoio constantes.

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

AMBIENTE PARA CONSTRUÇÃO DE INTERFACES GRÁFICAS DE USUÁRIO COM O SCV

Autor: Cícero Augusto de Lara Pahins
Orientador: Prof. Dr. Cesar Tadeu Pozzer
Local e data da defesa: Santa Maria, 15 de Dezembro de 2011.

Considerando-se a importância das GUIs em softwares contemporâneos, e a dificuldade em projetá-las e desenvolvê-las, o objetivo deste trabalho é produzir um software com o propósito de auxiliar na criação de interfaces que utilizam a *API SCV* para sua implementação. Assim, foi desenvolvido um ambiente para construção de interfaces gráficas de usuário que oferece funcionalidades de arranjo de componentes da API, bem como exportação do respectivo código. A implementação seguiu um padrão incremental, com um ciclo de prototipação, o que permite a modularização e fácil expansão do ambiente. O software produzido foi chamado *SCV Designer* e satisfaz os requisitos propostos, contando ainda com várias possibilidades de expansão já encaminhadas.

Palavras-chave: SCV; interface gráfica; ambiente para construção.

ABSTRACT

Trabalho de Graduação
Undergraduate Program in Computer Science
Universidade Federal de Santa Maria

ENVIRONMENT FOR CONSTRUCTION OF GRAPHICAL USER INTERFACES WITH SCV

Author: Cícero Augusto de Lara Pahins
Advisor: Prof. Dr. Cesar Tadeu Pozzer

Considering the importance of GUIs in contemporary software, and the difficulty in designing and develop them, the goal of this work is to produce a software for the purpose of assisting the creation of interfaces that use the *API SCV* to its implementation. Thus, was developed an environment for building graphical user interfaces that includes features of arrangement of components of the API, as well as export their code. The implementation followed an incremental pattern, with a cycle of prototyping, which allows modularity and easy expansion of the environment. The software produced was called *SCV Designer* and satisfies the proposed requirements, with the advantage of several opportunities for expansion already underway.

Keywords: SCV; graphical user interface; environment for construction.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

2D	bidimensional
3D	tridimensional
API	*Application Programming Interface*
FPS	*Frames per Second*
GLUT	*OpenGL Utility Toolkit*
GUI	*Graphic User Interface*
IDE	*Integrated Development Environment*
IHC	Interação Humano-Computador
OpenGL	*Open Graphic Library*
SCV	*Simple Components for Visual*
UI	*User Interface*

SUMÁRIO

1 INTRODUÇÃO

A criação de interfaces gráficas de usuário é tão importante quanto qualquer outro aspecto de programação, como a compreensão das estruturas de controle e a orientação a objetos (BISHOP; HORSPOOL, 2004), e faz parte da construção da maioria dos softwares contemporâneos (The Linux Information Project, 2004). Conforme houve a evolução das linguagens de programação, foram desenvolvidos conjuntos de funções e rotinas para fácil reutilização de funcionalidades e processos de software, estas chamadas *Application Programming Interfaces* (APIs - Interfaces de Programação de Aplicativos). Dentre as APIs existentes há algumas especialmente destinadas para a criação de *Graphical User Interfaces* (GUIs, Interfaces Gráficas de Usuário), como por exemplo, GTKmm (The GTK+ Team, 2011), wxWidgets (wxWidgets Developers and Contributors, 2011) e Java Swing (Oracle Corporation, 2011a).

Uma das grandes vantagens de GUIs é que elas tornam a operação em computador intuitiva, facilitando a aprendizagem e o uso por oferecem *feedback* imediato do efeito de cada operação realizada, como por exemplo, quando um usuário deleta um ícone que representa um arquivo e este imediatamente desaparece, confirmando que o arquivo foi deletado. Em adição a isto, permitem ao usuário tirar total proveito da habilidade que um sistema operacional moderno oferece para executar múltiplos programas ou múltiplas instâncias de um mesmo programa simultaneamente, o que resulta em um aumento da flexibilidade no uso do computador e do nível de produtividade (The Linux Information Project, 2004).

O projeto destas GUIs muitas vezes requerem que o programador trabalhe com gráficos elaborados, múltiplas maneiras de execução de um mesmo comando, múltiplos dispositivos de entrada assíncronas, como por exemplo, o

mouse e o teclado, uma estrutura lógica em que o usuário possa realizar um comando qualquer em virtualmente qualquer ordem e uma semântica de *feedback*, onde as respostas exibidas pelo aplicativo dependem de informações dos objetos presentes na tela. Todos estes itens tornam o desenvolvimento de uma GUI muito trabalhoso e, com o objetivo de facilitar esta tarefa, foram criadas as ferramentas para construção de interfaces gráficas.

Ferramentas para construção de interfaces gráficas permitem ao programador arranjar componentes visuais de uma maneira prática e rápida, estabelecendo conexões lógicas entre diferentes objetos e, muitas vezes, gerando o respectivo código fonte. Através destas ferramentas o uso das APIs de GUIs torna-se muito mais ágil e poderoso, uma vez que o usuário ou programador obtém *feedback* imediato de suas ações durante a construção da interface.

Assim, a motivação para este trabalho vem do fato da inexistência de uma ferramenta de construção de interfaces gráficas para a API gráfica *Simple Components for Visual* (SCV) - descrita em maiores detalhes na seção 2.2 -, a qual foi desenvolvida em laboratórios de pesquisa da Universidade de Federal de Santa Maria (UFSM) e possui foco na disponibilidade dos principais recursos oferecidos por APIs comerciais semelhantes, muitas vezes proprietárias ou possuindo uma grande comunidade de desenvolvedores, em conjunto com um baixo nível de complexidade para uso.

1.1 Motivação

Linguagens como Java e outras, utilizadas principalmente para desenvolvimento de aplicativos comerciais, dispõem de suporte nativo à construção de interfaces de usuário, oferecendo APIs e ambientes de desenvolvimento integrados e multiplataformas que dão suporte a criação de aplicações dirigidas pela interface. Um exemplo é a IDE (Integrated Development Environment, Ambiente de Desenvolvimento Integrado) NetBeans, a qual proporciona o desenvolvimento de aplicações de maneira visual ao permitir a utilização de componentes gráficos através da API *Swing* em conjunto com um ambiente Java.

Outras linguagens, como C++ e C, não oferecem uma API padronizada para a criação de interfaces de usuário, as opções existentes muitas vezes resultam em

uma curva de aprendizagem relativamente grande, bem como em uma alta complexidade de instalação e configuração das mesmas (AVELAR; GOMES; POZZER, 2007), assim adicionando muita sobrecarga às aplicações mais simples ou que buscam na interface gráfica uma maneira para visualização de resultados.

Um software de interface gráfica geralmente é grande, complexo, difícil de implementar, depurar e modificar (MYERS, 1993). Um estudo (MYERS; ROSSON, 1992) comprovou que em média 48% do código de aplicações é dedicado a interfaces de usuário, e que cerca de 50% do tempo despendido em desenvolvimento é utilizado para a implementação destas interfaces. À medida que interfaces tornam-se mais fáceis de usar, elas tornam-se mais difíceis de criar (MYERS, 1994).

A *API SCV* oferece funções e rotinas para a construção de interfaces de usuário, entretanto, não há um ambiente em que o programador possa construí-la de maneira visual. Durante o desenvolvimento de aplicações que utilizam o *SCV* para a construção de interfaces gráficas existe um grande investimento de tempo para o arranjo dos componentes gráficos, já que este é feito através do código fonte, não gerando, assim, uma relação visual e direta com as mudanças realizadas.

Assim, procurando oferecer um software que automatizasse parte das etapas para a criação de uma interface gráfica utilizando a *API SCV*, projetou-se o ambiente do presente trabalho.

1.2 Objetivos

O objetivo principal deste trabalho constitui-se na produção de um ambiente para a construção de interfaces gráficas de usuário utilizando o *SCV*. Este ambiente deve proporcionar ao usuário a possibilidade de arranjar os componentes gráficos da *API SCV* de maneira visual, oferecendo recursos de *drag-and-drop* para adição ou remoção de objetos, exportação do código resultante e alinhamento automático de componentes relacionados por diferentes esquemas de organização, assim, agilizando o processo de criação de interfaces.

Diversos objetivos específicos de implementação também foram definidos; figurando entre eles a organização do software de forma a permitir fácil expansão

futura e a revisão e adição de novos recursos a *API SCV*. Estes objetivos são discriminados na Seção 1.2.1.

1.2.1 Objetivos Específicos

Define-se como objetivos específicos de desenvolvimento:

- O foco na facilidade de uso, de acordo com a adoção de padrões e semelhança do software com ferramentas de construção de interfaces de usuários utilizadas pela indústria;
- A construção modularizada de forma que o programa possa ser construído iterativamente e facilmente expandido;
- Desenvolver um módulo que permita a definição de atributos e comportamentos dos componentes da *API SCV*;
- Desenvolver um módulo que permita a edição dos esquemas de alinhamento e posicionamento automáticos propostos por este trabalho para componentes do *SCV*;
- Permitir a exportação de código parametrizável de forma que a interface produzida no software mantenha as características definidas em sua construção;
- Estabelecer um controle interno do programa que aceite definições de preferência do usuário, carregadas a partir de um arquivo de configurações;
- Adicionar e reformular funcionalidades da *API SCV* para corrigir erros e deficiências da versão atual.
- Disponibilizar o software gratuitamente quando da sua conclusão.

1.3 Organização do Texto

O Capítulo 2, Revisão Bibliográfica e Fundamentação, apresenta uma revisão de conceitos necessários para a realização do trabalho. Os assuntos abordados são, principalmente, características de alto nível da *API SCV* e de ferramentas para construção de interfaces gráficas de usuário e uma fundamentação teoria

relacionada à IHC (Interação Humano-Computador). No Capítulo 3, Planejamento do Software, é discutido o planejamento e desenvolvimento do protótipo proposto.

Já no Capítulo 4, *SCV Designer*, é discutido o desenvolvimento final, com especial atenção à estruturação do programa e algumas classes que definem processos importantes relacionados aos objetivos do trabalho. Os resultados e considerações finais se encontram no Capítulo 5, Resultados e Conclusão. Ao longo do texto a mudança de fontes indica itens de código: nomes **destacados** em negrito e com a letra inicial em minúlculo referem-se a atributos, nomes de métodos e instâncias; nomes destacados **Class** em negrito e com a letra inicial em maiúsculo à nomes de classes.

2 REVISÃO BIBLIOGRÁFICA E FUNDAMENTAÇÃO

Este capítulo está dividido em duas seções. Primariamente, na Seção 2.1 é apresentada uma revisão bibliográfica a respeito de ferramentas para a criação de interfaces gráficas e suas características. Em seguida, a Seção 2.2 apresenta uma visão geral da *API SCV*, destacando seus aspectos mais relevantes para este trabalho.

2.1 Ferramentas para Criação de Interfaces Gráficas

A UI (User Interface, Interface de Usuário) de um programa de computador é a parte que trabalha com a saída para o monitor e a entrada da pessoa que o usa. O restante do programa é chamado de aplicação ou semântica da aplicação (MYERS, 1993).

Ferramentas para criação destas interfaces reduzem o tempo de desenvolvimento de um programa de quatro a cinco vezes, como observado em (SCHMUCKER, 1986). Segundo (MYERS, 1993), podemos enumerar as vantagens de se utilizar este tipo de ferramenta em dois grandes grupos:

1. A qualidade das interfaces produzidas será melhor:

    - Projetos de interface podem ser rapidamente prototipados e implementados, mesmo antes do código da aplicação ser escrito;
    - É mais fácil de incorporar mudanças descobertas através de testes de usuário;
    - Podem ser projetadas múltiplas interfaces de usuário para uma mesma aplicação.

- Diferentes aplicações tendem a possuir interfaces de usuário mais consistentes se construídas com a mesma ferramenta;
- Será mais fácil para uma variedade de especialistas envolverem-se na criação da interface, ao contrário de ser uma função exclusivamente executada por programadores.

2. O código da interface gráfica será mais fácil e barato de criar e manter:

   - Especificações da interface podem ser representadas, validadas e avaliadas mais facilmente;
   - Haverá menos código a ser escrito, já que boa parte é gerada pela ferramenta;
   - Haverá melhor modularização devido à separação do código da interface e da aplicação;
   - Será mais fácil portar a aplicação para hardware e ambiente de software diferentes, já que dependências de dispositivo estão isoladas na ferramenta.

Objetos visuais utilizados em interfaces gráficas são normalmente chamados *widgets*, os quais oferecem uma maneira de utilizar a entrada de um dispositivo físico para modificação de um certo tipo de valor. Alguns exemplos de *widgets* são menus, botões, barras de rolagem, áreas de texto e outros. Ferramentas para criação de interfaces gráficas proporcionam ao usuário a possibilidade de arranjo destes *widgets*, bem como facilitam a criação da semântica de interação com o aplicativo.

Um estudo comparativo entre ferramentas para construção de interfaces gráficas foi realizado previamente a este trabalho com os softwares Oracle NetBeans (Oracle Corporation, 2011b) e Microsoft Visual Studio 2010 (Microsoft Corporation, 2011), os resultados foram utilizados no projeto do ambiente proposto.

2.1.1 Conceitos Desejados para Ferramentas Interativas de Desenvolvimento de Interfaces Humano-Computador

Segundo (HARTSON; HIX, 1989) deseja-se que ferramentas interativas de desenvolvimento de interfaces humano-computador possuam características que

tornem o processo de construção mais fácil e intuitivo, algumas delas são:

1. **Funcionalidade**: interfaces humano-computador são muito complexas, consistindo em uma variedade de funcionalidades e dispositivos. As ferramentas devem ser capazes de produzirem interfaces contendo esta variedade;

2. **Usabilidade**: ferramentas para o desenvolvimento de interfaces humano-computador são software complexos e a usabilidade é uma importante questão para a produtividade e satisfação dos desenvolvedores que usam-nas;

3. **Extensibilidade**: uma vez que as possibilidades para interfaces humano-computador são ilimitadas, as ferramentas não conseguem resolver cada necessidade, assim, estas devem ser extensíveis para lidar com novos recursos, estilos de interação e dispositivos;

4. **Manipulação Direta**: as ferramentas de desenvolvimento devem oferecer a possibilidade de manipulação visual direta dos objetos da interface projetada;

5. **Integração**: um conjunto de ferramentas para o desenvolvimento de interfaces deve possuir somente uma interface gráfica integrada de acesso a todas as funcionalidades;

6. **Orientação Estruturada**: sem a ajuda das ferramentas no processo de organização e projeto de interfaces o desenvolvedor pode ser confrontado com uma grande quantidade de detalhes. Assim, as ferramentas devem possuir uma orientação estrutura, de modo que se adote um modelo descritivo da interface humano-computador que irá determinar o relacionamento entre diferentes objetos.

2.2 SCV

Com o intuito de satisfazer às necessidades imediatas de alunos de Ciência da Computação da UFSM nas disciplinas em que produzem software que necessitam de interfaces gráficas, principalmente aquelas ligadas a computação gráfica, foi desenvolvida no Laboratório de Computação Aplicada (LaCA) da

UFSM uma API gráfica denominada *Simple Components for Visual* (SCV) (LIMBERGER; PAHINS; POZZER, 2010) (PAHINS et al., 2010) (PAHINS et al., 2010) (PAHINS; LIMBERGER; POZZER, 2010) (PAHINS; POZZER, 2011), a qual oferece uma abstração da API OpenGL e é utilizada através das linguagens C e C++.

A principal característica pretendida com a API é oferecer ao programador as ferramentas necessárias para a composição de GUIs similares às interfaces de software comerciais reconhecidos, mantendo um baixo nível de complexidade, fácil configuração e instalação.

A *API SCV* foi primariamente utilizada no desenvolvimento de trabalhos da disciplina de Computação Gráfica da UFSM e, posteriormente, com o *feedback* dos acadêmicos e o constante aprimoramento, foi disponibilizada como software livre (SCV Team, 2011), atualmente encontrando-se em sua terceira versão. Em (GOTTIN, 2010) a API foi utilizada como suporte para o desenvolvimento da aplicação *DrawtexSCV*, a qual é resultado de um trabalho de conclusão do curso de Ciência da Computação da UFSM.

O *SCV*, em sua versão 3.0, é estruturado a partir de um núcleo (**scv::Kernel**), o qual é responsável por controlar todas as interações do usuário com a interface e gerenciar os componentes da API. Essas atribuições são especificamente:

- Encapsular as callbacks de *mouse* e teclado oferecida pela API *freeglut*, as quais são de baixo nível e, por isso, faz-se necessário que passem por um formatação para corresponderem aos recursos do *SCV*;
- Controlar a taxa de exibição de *Frames per Second* (FPS, Quadros por Segundo), o que garante um controle na utilização de recursos;
- Encapsular os recursos de janela e de Área de Transferência, recurso utilizado para transferência de pequenas quantidades de dados para aplicações distintas através de operações copiar e colar, dos diferentes sistemas operacionais em que o *SCV* é suportado;
- Gerenciar os componentes visualizados, bem como operações de baixo nível, como carregamento de texturas, controle de memória, entre outras.

O *SCV* utiliza um modelo de projeto modularizado, permitindo a criação de vários objetos, estes divididos em três grandes categorias que são passíveis de expansão e generalização: ***containers***, ***components*** e ***features***. Estas categorias são de fundamental importância para o entendimento do funcionamento da API e estão descritas nos parágrafos que seguem, podendo ser visualizadas através de uma relação hierárquica no diagrama de classes simplificado da Figura 2.1.

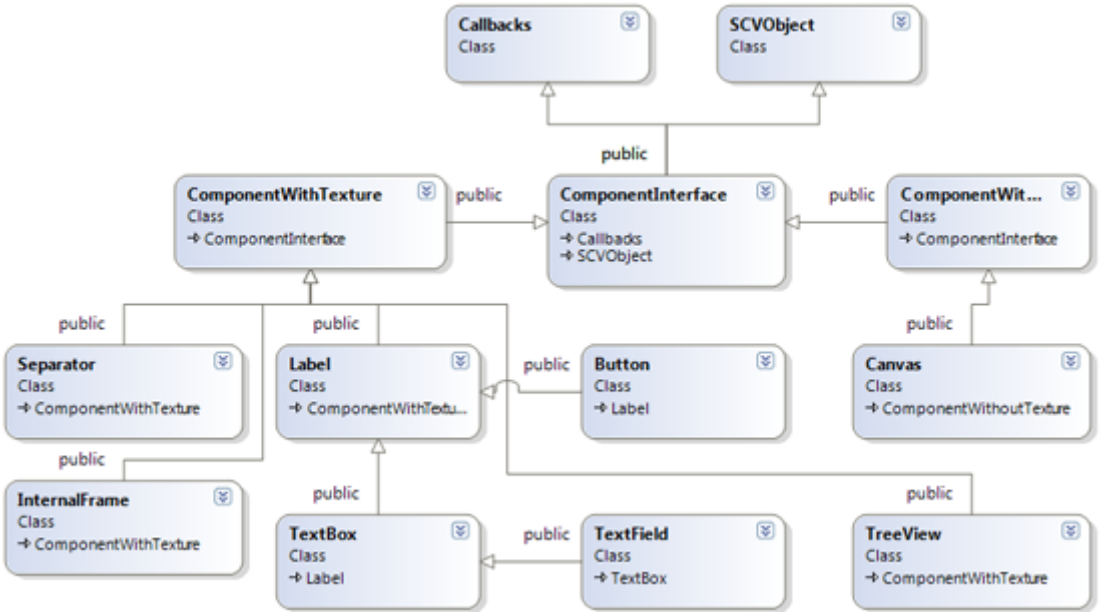


Figura 2.1: Diagrama de classes simplificado da versão 3.0 do *SCV*.

Os ***containers*** têm como função transportar outros objetos dentro de um bloco intrínseco, agrupando funcionalidades e características, facilitando a criação de interfaces dinâmicas. De maneira prática, são painéis em que outros objetos podem ser adicionados pelo programador, tornando estes dependentes das variáveis correntes de visualização, posicionamento e dimensionamento. Os painéis também são responsáveis pelo gerenciamento destes objetos, o que garante um controle fino das *callbacks* e da alocação de memória, exonerando o **Kernel** do *SCV* destas funções.

Os ***components*** são objetos que podem ser adicionados aos *containers* e oferecem o meio principal de interação do usuário com a interface gráfica: ”uma não trivial, quase independente, e substituível parte de um sistema que cumpre uma função clara no contexto de uma arquitetura bem definida”. Essa definição

usada por Brown e Wallnau (BROWN, 1996) para definir um componente, na visão de engenharia de software, pode ser estendida ao que o componente representa dentro da interface do *SCV*. Ou seja, um componente é um objeto que possui uma funcionalidade específica e a relação dele para com outros componentes depende inteiramente de como o programador utiliza os recursos a ele oferecidos.

Existem dois tipos de componentes no *SCV*:

1. **scv::ComponentWithTexture**: Os **componentes com textura** são aqueles que, em sua criação, há necessidade da geração off-line ou online de uma textura que irá representar todos seus modos de visualização e, diferentemente dos componentes sem textura, não permitem a edição deste conteúdo em tempo real;

2. **scv::ComponentWithoutTexture**: Os **componentes sem textura** são aqueles em que o programador pode utilizar seus próprios métodos de exibição e integra-los ao *SCV*, usufruindo de todos recursos oferecidos pela API. Um exemplo de componente deste tipo é o **scv::Canvas**, implementado através de uma abstração da *viewport* do OpenGL, oferece a possibilidade de exibição de cenas gráficas bidimensionais (2D) ou tridimensionais (3D).

A renderização de componentes do *SCV* é realizada através de uma simplificação da técnica de *Frame Buffer Object*, a qual permite a criação de *framebuffers* (resultados finais de tarefas e transformações gráficas projetadas em uma tela como *pixels* 2D) não exibíveis, o que resulta em um baixo consumo de recursos aliado a uma baixa necessidade de processamento (WRIGHT; LIPCHAK; HAEMEL, 2007), quando comparado a técnicas padrões do OpenGL como *Display List*, *Vertex Array*, *Vertex Buffer Object*.

A construção destes componentes é realizada através de funções de desenho de formas geométricas primitivas e de caracteres implementadas em baixo nível no próprio *SCV*, assim garantindo um comportamento gráfico unificado em todos os sistemas operacionais suportados. Por exemplo, a construção da representação de um **scv::Button** é realizada através da combinação de funções

de desenho de pontos e retas em conjunto com as de caracteres.

As ***features*** são recursos que não são visíveis para o usuário de um programa que utiliza o *SCV*, mas abstraem rotinas de baixo nível, muitas vezes dependentes do sistema operacional ou hardware, e podem ser utilizadas pelo programador a fim de facilitar o desenvolvimento de sua aplicação.

Todos os itens da interface possuem as próprias *callbacks* e o **Kernel** do *SCV* se encarrega de definir qual componente tem o foco de *mouse* ou teclado em um dado momento e de realizar as chamadas apropriadas. Os métodos de *callback* disponibilizados para cada componente, desde que em sua definição o programador os herde, podem ser vistos no Código 2.1. Assim, para criar-se um CheckBox, por exemplo, o programador define uma classe descritiva derivada de **scv::CheckBox** e declara as *callbacks* que serão utilizadas, programando as ações a serem tomadas em seu escopo.

```
/*********************************************************/
/* Mouse Callbacks
/*********************************************************/
virtual void onMouseClick(const scv::MouseEvent &evt) = 0;
virtual void onMouseHold (const scv::MouseEvent &evt) = 0;
virtual void onMouseOver (const scv::MouseEvent &evt) = 0;
virtual void onMouseUp (const scv::MouseEvent &evt) = 0;
virtual void onMouseWheel(const scv::MouseEvent &evt) = 0;

/*********************************************************/
/* Keyboard Callbacks
/*********************************************************/
virtual void onKeyPressed(const scv::KeyEvent &evt) = 0;
virtual void onKeyUp (const scv::KeyEvent &evt) = 0;

/*********************************************************/
/* Other Callbacks
/*********************************************************/
virtual void onDragging(void) = 0;
virtual void onResizing(void) = 0;
```

Código 2.1: Detalhe de código exibindo as callbacks disponíveis para todos os componentes do *SCV*.

A versão 3.0 do *SCV* possui 136 arquivos e 12674 linhas de código fonte útil, ou seja, são desconsideradas linhas em branco ou somente com caracteres de marcação e comentários.

3 PLANEJAMENTO DO SOFTWARE

De forma a verificar quais os pontos de dificuldade no desenvolvimento do software proposto, foi produzido um protótipo utilizando a versão 3.0 da *API SCV*. Neste protótipo explorou-se, principalmente, a forma de interação do usuário com a manipulação de componentes inseridos através de uma interface gráfica, a qual foi planejada para utilizar os recursos atualmente disponíveis no *SCV*, bem como validar a exportação de código gerado automaticamente.

Foi escolhido proceder a partir de planejamento com prototipação atrelada pelo fato da dificuldade em planejar um software complexo sem experiência prévia, com tantas possíveis abordagens de utilização, interação e saída de dados. Além disso, a implementação final de um software costuma beneficiar-se da experiência adquirida durante o desenvolvimento de um protótipo (REZENDE, 2005), o que neste caso poderia ser observado na ocorrência de eventuais deficiências de recursos ou erros de programação encontrados na *API SCV* durante a utilização da mesma.

Já que o software final poderia exigir mudanças ou adições de novas funcionalidades no *SCV*, o que tornaria o código escrito durante esta fase inadequado para a utilização após as alterações, durante o desenvolvimento do protótipo buscou-se um modo de estruturação simples, sem a preocupação com *designs* de programação que oferecessem maior manutenibilidade ou facilidade de incremento.

A interação do usuário com o protótipo dá-se através de uma Área de Trabalho, na qual é possível verificar-se a interface produzida, bem como controlar todas as funcionalidades oferecidas pelo software. Na Figura 3.1 é exibida uma visão geral da interface do protótipo.

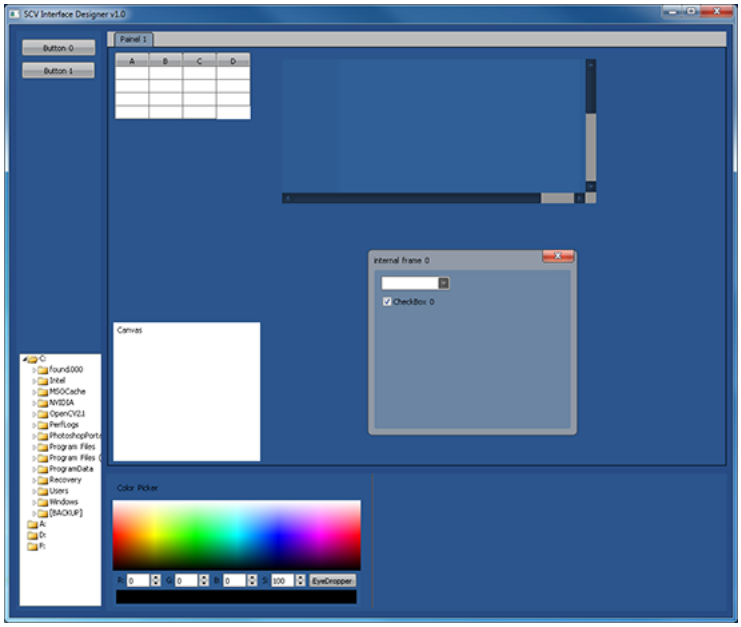


Figura 3.1: Interface construída utilizando o protótipo.

A adição ou remoção de componentes por meio de uma interface gráfica foi definida através do componente **scv::ContextMenu**, o qual é exibido na ocorrência da *callback* **scv::onMouseClick** na Área de Trabalho do protótipo ou em qualquer componente previamente adicionado, como demonstram as Figuras 3.2 e 3.3.

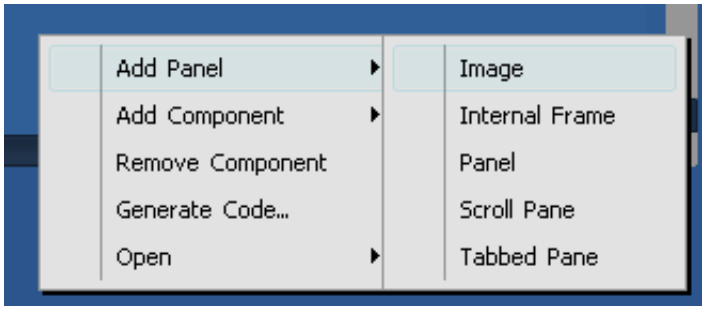


Figura 3.2: Adição de uma Imagem na Área de Trabalho do protótipo.

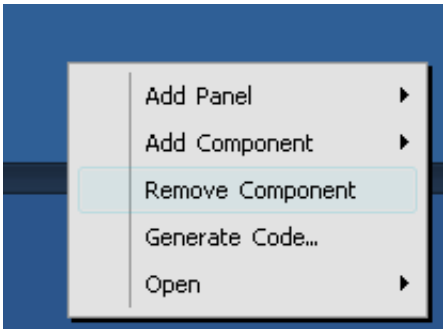


Figura 3.3: Remoção de um componente da Área de Trabalho do protótipo.

A geração automática de código é realizada a partir da verificação dos componentes presentes na Área de Trabalho do protótipo, etapa disparada a partir de um comando do usuário. A partir disto, características dos componentes que irão estar presentes no código de saída, como tamanho, posicionamento e relacionamento com outros componentes, são enviadas para a classe **CodeGenerator**. Nesta classe ocorre a criação do texto de saída, traduzindo todas as características para código fonte. No Código 3.1 é exibido parte de um código fonte gerado automaticamente pelo protótipo (partes de código que realizam o posicionamento e dimensionamento dos componentes, bem como o tratamento de *callbacks* do *SCV*, foram ocultadas).

```
int main(int argc, char* argv[]) {
  static Kernel *kernel = Kernel::getInstance();
  // Load ColorScheme
  static ColorScheme *scheme = ColorScheme::getInstance();
  scheme->loadScheme(ColorScheme::clean);
  // Set Windows Size
  kernel->setWindowSize(1050, 860);
  // Set Frame Rate
  kernel->setFramesPerSecond(40.f);
  /*********************************************/
  scv::Panel * panel, * panel_tmp;
  scv::ScrollPane * scrollpane;
  scv::TabbedPane * tabbedpane;

  panel = new Panel0();
  kernel->addComponent(panel);

  panel_tmp = panel;
  panel = new Panel1();
  panel_tmp->addComponent(panel);

  panel->addComponent(new TextField0());

  panel_tmp = panel;
  panel = new Panel2();
  panel_tmp->addComponent(panel);

  panel->addComponent(new Button0());
  /*********************************************/
  kernel->run();
  return 0;
}
```

Código 3.1: Função *main* gerada automaticamente pelo protótipo para uma interface simples.

No protótipo também foi projetado, através da classe **AbstractWindow**, a qual deriva de uma **scv:InternalFrame** e implementa uma janela flutuante ativada por comando do usuário, uma opção para edição de algumas características dos componentes gerenciados pelo programa, como por exemplo, o texto exibido pelo componente **scv::Button**. Na Figura 3.4 é exibida uma visão desta janela.

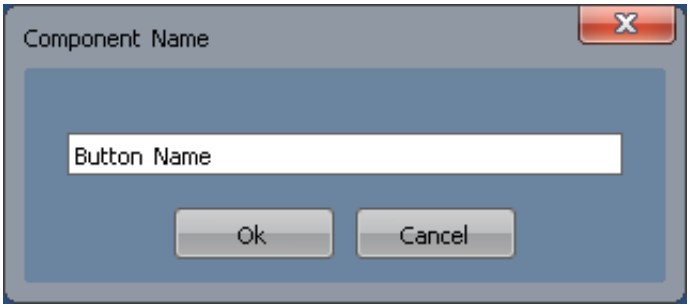


Figura 3.4: Janela de edição do protótipo para o texto de um **scv:Button**.

O protótipo desenvolvido possui um total de 11 arquivos e 927 linhas de código fonte útil.

3.1 Análise do Protótipo

Percebeu-se que, embora o código exportado pelo protótipo fosse válido e representasse as características de construção definidas pelo usuário, o modo de implementação utilizado por esta funcionalidade mostrou-se muito dependente das atuais características e funcionalidades oferecidas pelo *SCV*, assim tornando este módulo de difícil reaproveitamento em casos de alteração na API. Assim, definiram-se para o programa final, estruturas de dados em que características específicas e globais dos componentes são diferentemente trabalhadas, de maneira que alterações globais reflitam na geração de código de todos componentes e alterações locais interfiram somente no componente alvo e em suas especializações. Maiores detalhes sobre este esquema de implementação são expostos na Seção 4.2.5.

Além das conclusões referentes à exportação de código, obtiveram-se alguns resultados referentes às funcionalidades da versão 3.0 do *SCV*. Verificou-se que seria necessária uma reformulação na maneira em que a API trata o gerenciamento de memória, a passagem de *callbacks*, como o tratamento de *mouse* e

controle de arrasto e redimensionamento de componentes, e as classes **scv::-Component** e **scv::Kernel**.

O atual gerenciamento de memória implementado pela API exige que o controle de relacionamento entre componentes (por exemplo, quando um **scv:Button** se encontra dependente da posição e tamanho de um **scv::Panel**) seja feito de maneira manual, assim podendo correr falhas de liberação de memória ao deletar-se objetos. A implementação corrente também não oferece uma maneira do usuário acessar, a partir de um dado componente, seu pai e filhos, o que impossibilita a exclusão de um objeto que não esteja adicionado ao **scv::Kernel**.

Na atual implementação da passagem de *callbacks*, o **scv::Kernel** envia as chamadas dos métodos correspondentes com a entrada, seja ela *mouse* ou teclado, para todos os componentes por ele gerenciado, assim, ficando a cargo destes decidirem quando passar estas *callbacks* para o usuário ou executarem rotinas que dependam delas. Este esquema, além de suscetível a falhas, já que o controle é dependente de cada componente, demonstrou-se mal implementado, pois foi verificado através do protótipo situações em que *callbacks* são chamadas de maneira errônea, como por exemplo, quando um **scv::Button** é arrastado na Área de Trabalho e ainda assim recebe as *callbacks* referentes a detecção de sobreposição do *mouse*, o que não deveria ocorrer.

Outra contribuição do protótipo para a implementação final foi a exposição de limitações do *SCV* para com a adição de recursos globais, ou seja, que não sejam ligados a um determinado componente. Na atual implementação, o *SCV* não oferece maneira de detecção de posição e comportamento globais do *mouse*, o que impossibilita, por exemplo, que um componente seja atrelado ao *mouse* e este sirva de âncora de posicionamento. Esta limitação dificulta o desenvolvimento de um software de construção de interfaces gráficas mais intuitivo, já que o usuário ficaria impossibilitado de arrastar um componente por toda a extensão de edição oferecida pelo software, assim como ocorre em outros programas do gênero.

Por fim, espelhando-se nos softwares de construção de interfaces gráficas e, baseado na dificuldade observada durante a utilização do protótipo para gerenciar-se componentes adicionados, percebeu-se a necessidade da inclusão de guias

visuais que possibilitem uma visualização hierárquica destes componentes juntamente com possibilidade de edição de todas suas características.

4 SCV DESIGNER

Neste capítulo são discutidos os aspectos mais relevantes da implementação realizada. Na Seção 4.1 é apresentada uma visão geral da estruturação do *SCV Desginer* e, na Seção 4.2, são apresentados detalhes técnicos da etapa de desenvolvimento.

4.1 Estrutura do Programa

A partir dos resultados obtidos com o protótipo, foi planejado o modelo de estruturação do *SCV Designer*. Buscou-se desenvolver uma estrutura modularizada, a qual possibilitasse sua construção iterativa e facilitasse a expansão do software. Para isto, os processos que são realizados pelo usuário, e são o foco do ambiente, foram divididos em quatro classes lógicas:

- **DesignPreview**: responsável pela pré-visualização em tempo real de interfaces produzidas pelo ambiente a partir da construção de uma visualização aproximada, a qual busca refletir o resultado da exportação do trabalho realizado. A correspondência gráfica desta classe lógica pode ser visualizada na Figura 4.1. Maiores detalhes na Seção 4.2.2;

- **GroupLayout**: responsável pela construção hierárquica de agrupamentos lógicos para arranjo de componentes da *API SCV* e desenvolvida a fim de refletir o esquema proposto por este trabalho e descrito na Seção 4.2.3. A correspondência gráfica desta classe lógica pode ser visualizada na Figura 4.2;

- **ObjectEditor**: responsável pela edição de variáveis e características, as quais estarão presentes nas exportações do ambiente, de componentes da

API SCV. A correspondência gráfica desta classe lógica pode ser visualizada na Figura 4.3. Maiores detalhes na Seção 4.2.4;

- Geração de Código: responsável pela exportação do resultado produzido pelo ambiente de acordo com parâmetros do usuário. Maiores detalhes na Seção 4.2.5.

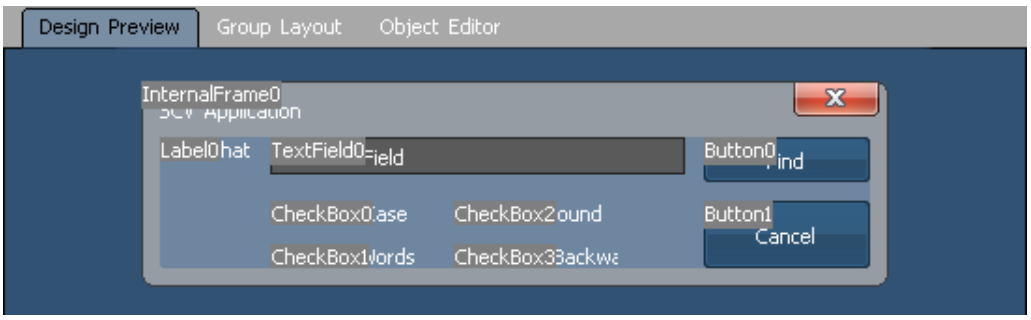


Figura 4.1: Interface gráfica do *SCV Designer* para a classe lógica **DesignPreview**.

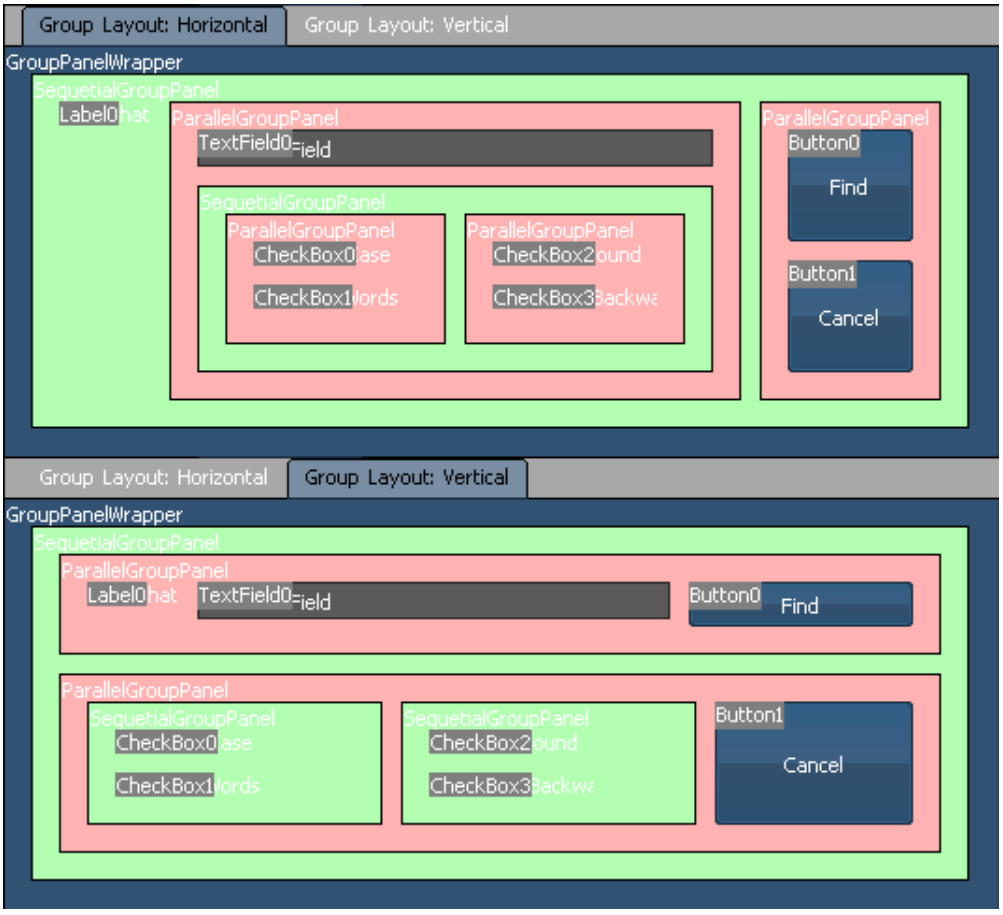


Figura 4.2: Interface gráfica do *SCV Designer* para a classe lógica **GroupLayout**.

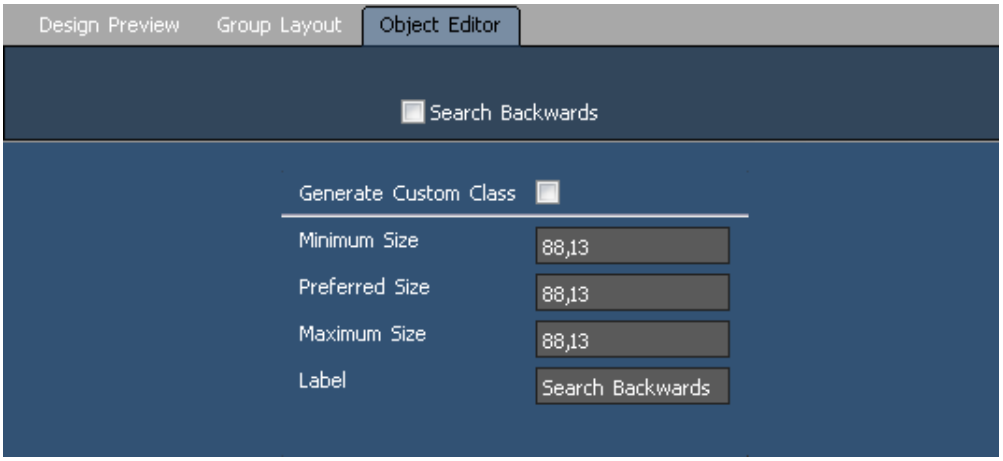


Figura 4.3: Interface gráfica do *SCV Designer* para a classe lógica **ObjectEditor**.

Nota-se que cada uma delas diz respeito a objetivos específicos do ambiente, conforme estabelecido na Seção 1.2.1.

De acordo com a experiência proveniente da construção do protótipo, cada classe lógica utilizou um modelo de controle central, a partir do *design pattern* denomidado *Singleton*, a fim de evitar a replicação de dados e gerenciar o controle a recursos compartilhados. Este modelo oferece um único ponto de acesso global às classes derivadas, proíbe a alocação repetida de objetos e possibilita a comunicação entre as diferentes classes lógicas.

O gerenciamento central do ambiente é realizado pela classe **Application**, a qual é responsável pela inicialização da *API SCV*, como definição de variáveis de tamanho e título da janela, criação e arranjo dos componentes da interface gráfica e controle geral da aplicação bem como das *callbacks* geradas pelo usuário.

4.2 Desenvolvimento

4.2.1 Reformulação da *API SCV*

Como descrito na Seção 3.1, Análise do Protótipo, foi observada a necessidade da reformulação de alguns aspectos da versão 3.0 do *SCV*.

O gerenciamento de memória entre os componentes da API foi reescrito, de maneira que o relacionamento entre objetos dá-se de maneira automatizada. Foi desenvolvido um esquema onde cada componente possui um pai e uma lista de filhos. Os únicos pontos de acesso ao pai e a lista de filhos são os métodos oferecidos pela classe **scv::Component**, o que garante a utilização destes para

qualquer alocação de objetos na API. No Código 4.1 são exibidos os métodos de gerenciamento de memória.

```
////////////////////////////////////////////////////////////
void setParent(Component *parent);
inline Component *getParent(void) const;

inline const Component::List &getChildren(void) const;

virtual void addChild(Component *object);
virtual void removeChild(Component *object);
virtual void removeAllChild(void);

virtual Component *getChild(int index) const;

void pullChildToTop(Component *child);

bool hasChild(Component *child) const;
////////////////////////////////////////////////////////////
```

Código 4.1: Métodos do esquema de gerenciamento de memória desenvolvido.

Nota-se que a implementação deste esquema de gerenciamento de memória forçou uma revisão na quase totalidade dos componentes do *SCV*, já que os métodos anteriormente utilizados tornaram-se obsoletos. As classes **scv::Component** e **scv::Kernel** também foram reescritas.

Com o objetivo de suportar o esquema de arranjo automático de componentes proposto por este trabalho, e descrito na Seção 4.2.3, foram adicionadas características de tamanho mínimo, preferencial e máximo à classe **scv::Component**, de maneira que todos os componentes precisaram ter estas características configuradas conforme suas propriedades de redimensionamento. Por exemplo, é notável que um **scv::CheckBox** não pode sofrer variação de tamanho, entretanto, o mesmo não é válido para um **scv::TextBox**, o qual tem por natureza suportar qualquer tamanho especificado pelo programador.

Na classe **scv::Kernel**, responsável pelo controle geral da API, foram adicionados métodos de *callbacks* globais, assim, diferentemente da versão 3.0 do *SCV*, é possível, por exemplo, obter a posição do mouse indiferentemente de qual componente está em foco no momento. Outros exemplos poderiam ser as callbacks de mudança de posição e tamanho da janela ou de uso do teclado.

Além destas alterações, foram realizadas modificações no processo de cha-

madas das *callbacks*, o que garantiu o correto funcionamento das mesmas em relação a alguns aspectos observados na implementação anterior da API, e correções gerais em diversos componentes e algoritmos, observados durante o desenvolvimento do protótipo e do *SCV Designer*.

Nesta etapa foram trabalhadas 5441 linhas de código fonte útil, onde 3171 linhas foram adicionadas, 960 linhas foram removidas e 1310 linhas foram modificadas. Ainda, foram modificados 122 arquivos de um total de 161.

4.2.2 Design Preview

O **Design Preview**, como especificado na Seção 4.1, é responsável por construir uma visualização aproximada que reflita o resultado da exportação do ambiente. Derivado de um **scv::Panel**, em seu construtor é inicializado o **GroupLayout**, definido na Seção 4.2.3, que irá arranjar os componentes gerenciados, observando características configuradas pelo usuário como, por exemplo, tamanho e posição.

No método **createPreview**, disparado no momento em que o usuário solicita uma visualização do trabalho a ser exportado pelo ambiente, são criadas estruturas lógicas utilizando classes da *API SCV* que correspondem às utilizadas pelo *SCV Designer*, ou seja, são criados todos os objetos que seriam resultado da exportação, mas neste caso em tempo de execução.

Nota-se que foi necessário a implementação de diversas classes de encapsulamento e abstração para permitir o usuário manipular componentes do *SCV* e criar esquemas de arranjo com o **GroupLayout** através de uma interface gráfica amigável. Embora estas classes sejam imprescindíveis para o funcionamento de baixo nível do *SCV Desginer* e suas implementações complexas, não se observou a necessidade de maiores explicações a cerca delas por tratarem-se de detalhes específicos de programação, os quais fugiriam do escopo deste trabalho e tornariam o texto muito maçante.

Ainda, é importante ressaltar que a criação da visualização do **Design Preview** exige uma iteração sobre as classes lógicas **Group Layout** e **Object Editor**, assim, há uma intercomunicação entre diversos objetos do ambiente que gerenciam o trabalho realizado.

4.2.3 GroupLayout

O **GroupLayout** foi desenvolvido para agrupar componentes do *SCV* hierarquicamente, a fim de posicioná-los em um **scv::Panel**, e é baseado no modelo utilizado pela linguagem Java (Oracle Corporation, 2011c). Inicialmente destinado ao uso automatizado através do *SCV Designer*, também pode ser codificado manualmente. O agrupamento é realizado por instâncias da classe **Group**. O **GroupLayout** suporta dois tipos de grupos: o grupo de alinhamento sequencial, em que o posicionamento dos elementos filhos é realizado sequencialmente, um após o outro, e o grupo de alinhamento paralelo, em que o posicionamento dos elementos filhos ocorre paralelamente a um ponto de referência.

Cada grupo pode conter um número qualquer de elementos, onde um elemento é definido por um **Group**, um **ComponentSpring** ou um **GapSpring**, todas classes herdadas de **Spring**, descrita na Seção 4.2.3.1.

Elementos são semelhantes a uma ”mola”, onde cada um possui tamanhos de alcance mínimo, máximo e preferencial, estes especificados pelo programador ou pré-definidos durante a construção. Cada alcance de um **ComponentSpring** é determinado através dos métodos **getMinimumSize**, **getPreferredSize** e **getMaximumSize**, acessados a partir da classe **scv::Component**, ou especificados pelo programador. Os alcances para um **Group** são determinados pelo seu tipo: para um grupo paralelo são os alcances máximos de seus filhos, e para um grupo sequencial são as somas dos alcances de seus filhos.

O **GroupLayout** trata cada eixo independentemente, isso é, existe um grupo que representa o eixo horizontal e um grupo que representa o eixo vertical. O **grupo horizontal** é responsável por determinar os alcances mínimo, máximo e preferencial ao longo do eixo horizontal, assim como definir o posicionamento correspondente à variável x e a largura dos elementos contidos. O **grupo vertical** é responsável por determinar os alcances mínimo, máximo e preferencial ao longo do eixo vertical, assim como definir o posicionamento correspondente à variável y e a altura dos elementos contidos.

Na Figura 4.4 é exibido um diagrama com três componentes, C1, C2 e C3, que estão contidos em um **scv::Panel**, e possuem um grupo de alinhamento sequencial ao longo do eixo horizontal e um grupo de alinhamento paralelo ao

longo do eixo vertical.

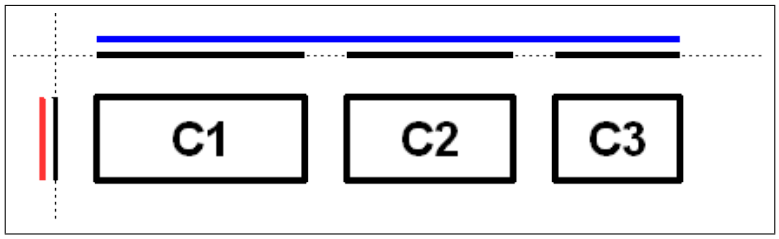


Figura 4.4: Diagrama exibindo três componentes com um grupo de alinhamento sequêncial ao longo do eixo horizontal e um grupo de alinhamento paralelo ao longo do eixo vertical (Oracle Corporation, 2011c).

O diagrama acima exibe o tratamento independente de eixos, projetando o alcance dos diferentes elementos em ambos os eixos, e representando o grupo sequencial de alinhamento ao longo do eixo horizontal por uma linha sólida da cor azul e o grupo paralelo de alinhamento ao longo do eixo vertical por uma linha sólida da cor vermelha. Observa-se que o grupo sequencial é o resultado da soma dos alcances dos elementos C1, C2 e C3, enquanto que o grupo paralelo é o máximo do alcance destes elementos.

Na Figura 4.5 é exibido um diagrama com os mesmos elementos do diagrama da Figura 4.4, entretanto, o grupo sequencial de alinhamento foi utilizado no eixo vertical, e o grupo paralelo de alinhamento foi utilizado no eixo horizontal, situação contrária ao último caso.

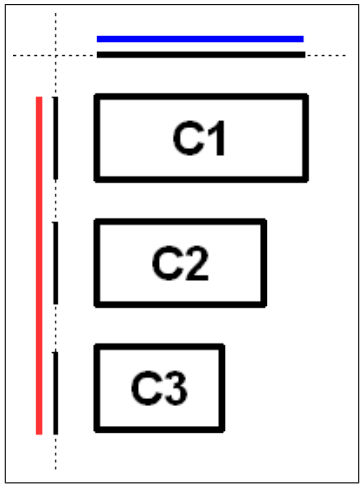


Figura 4.5: Diagrama exibindo três compontes com um grupo de alinhamento paralelo ao longo do eixo horizontal e um grupo de alinhamento sequencial ao longo do eixo vertical (Oracle Corporation, 2011c).

Observa-se que como C1 é o maior entre os três elementos, o grupo paralelo

adota o seu tamanho (linha sólida da cor azul). Como C2 e C3 são menores que C1, eles possuem um alinhamento baseado na especificação do grupo paralelo, neste caso é utilizado o alinhamento **LEADING**, em que o ponto de âncora tem orientação no sentido esquerda-para-direita. Caso a orientação utilizada fosse da direita-para-esquerda, C2 e C3 estariam alinhados no lado oposto. Outro alinhamento possível para o grupo paralelo é o **CENTER**, onde, utilizando o exemplo da Figura 4.5, os elementos menores C2 e C3 estariam alinhados ao centro em relação ao elemento C1.

Na Figura 4.6 é exibido um diagrama com os mesmos elementos do diagrama da Figura 4.5, entretanto, o grupo sequencial de alinhamento foi utilizado em ambos os eixos.

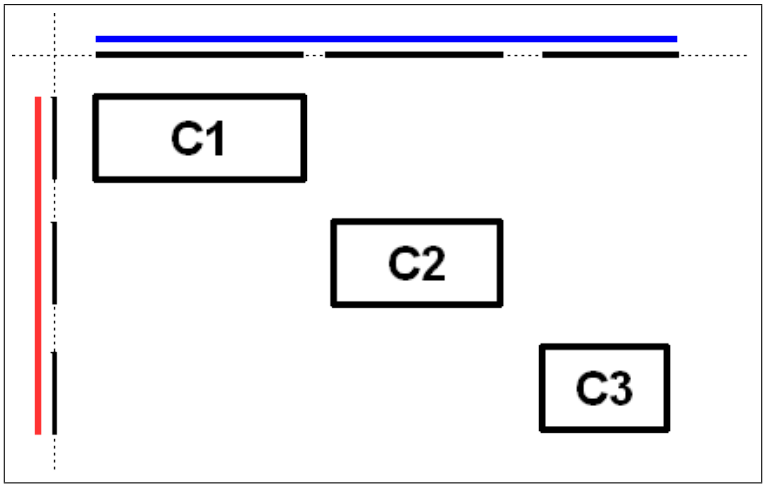


Figura 4.6: Diagrama exibindo três componentes com um grupo de alinhamento sequencial ao longo dos eixos vertical e horizontal (Oracle Corporation, 2011c).

Um **GroupLayout** deve ser atrelado a um **scv::Panel**, deste modo os componentes contidos no grupo de alinhamento e no painel serão arranjados automaticamente. Após a configuração dos grupos de alinhamento nos eixos vertical e horizontal, realizados pelos métodos **GroupLayout::setHorizontalGroup** e **GroupLayout::setVerticalGroup**, é invocado o método **GroupLayout::layoutContainer** a cada vez que o **scv::Panel** atrelado for exibido na tela. Este método é responsável por dar início ao processo recursivo que calcula o tamanho e posicionamento de todos os elementos do **GroupLayout**, sejam eles um **Group**, que, como explicado no início desta seção, pode conter quaisquer outros elementos, inclusive outros grupos, um **ComponentSpring** ou **GapSpring**.

O algoritmo disparado pelo método **GroupLayout::layoutContainer** inicialmente verifica, para cada eixo de alinhamento, se o tamanho relativo, horizontal ou vertical, do **scv::Panel** é maior que o tamanho preferencial do grupo de alinhamento configurado. Caso isso seja **verdadeiro**, conclui-se que os elementos podem ser configurados com o tamanho relativo do eixo do **scv::Panel**, ou seja, o espaço fornecido é maior que o necessário e os elementos devem sofrer um aumento. Caso isso seja **falso**, são possíveis duas opções: a primeira é de que o tamanho preferencial dos elementos seja menor que o tamanho relativo do eixo e a segunda é de que o tamanho relativo do eixo seja igual ao tamanho preferencial.

No **primeiro** caso os elementos precisam sofrer uma redução em relação ao seu tamanho preferencial para serem ajustados dentro do **scv::Panel**. Desta maneira, é calculada a diferença do tamanho relativo ao eixo subtraída do tamanho oferecido pelo **scv::Panel**, o que nos dá a quantidade de tamanho que ultrapassou o limite, e este resultado é subtraído do tamanho preferencial. Assim, os elementos devem ser ajustados recursivamente para que obedeçam ao tamanho calculado.

No **segundo** caso o tamanho relativo ao eixo é igual ao tamanho preferencial, assim os elementos são configurados diretamente com esta especificação.

É notável que, como todos os elementos também possuem tamanhos de alcance mínimos, é possível que não haja uma configuração ótima para que estes caibam no **scv::Panel**, o que resulta em um corte na exibição dos elementos extremos, ou seja, que são periféricos aos demais.

Após a definição dos tamanhos que os grupos de alinhamento horizontal ou vertical devem configurar seus elementos, é invocado o método **Group::setSize** que utiliza esta informação conforme a característica dos possíveis grupos especializados: grupo sequencial, representado pela classe **Group::SequentialGroup**, e grupo paralelo, representado pela classe **Group::ParallelGroup**. É importante ressaltar que a classe **Group** é virtual pura, ou seja, não pode ser instanciada. Deste modo todos os grupos pertencem as classes derivadas **SequentialGroup** ou **ParallelGroup**.

Para um grupo definido **SequentialGroup**, onde os filhos devem ser arranja-

dos de maneira sequencial, a chamada **Group::setSize** possui três alternativas e estão descritas abaixo:

1. O tamanho calculado pelo **GroupLayout::layoutContainer** é igual ao tamanho preferencial do grupo: neste caso os filhos devem ser configurados com os próprios tamanhos preferenciais;

2. O número de elementos filhos é igual a 1 (um): neste caso o tamanho a ser atribuído ao elemento é definido por *min(max(tamanho calculado, elemento->mínimo), elemento->máximo)*;

3. O número de elementos filhos é maior que 1 (um): neste caso é necessário a utilização de um algoritmo que distribua o tamanho calculado de maneira correta entre todos os filhos. Nota-se que estes elementos possuem tamanhos mínimos, preferencias e máximos diferentes, o que resulta em um redimensionamento não linear, ou seja, é possível que o tamanho atribuído a um filho seja diferente em relação ao tamanho atribuído a outro. Este algoritmo é descrito abaixo.

O algoritmo utilizado na **alternativa três** da chamada do método **Group::-setSize** de um **SequentialGruop** segue os seguintes passos:

1. Calcula-se a capacidade de redimensionamento de cada elemento, definido pelos tamanhos *(preferencial - mínimo)* ou *(máximo - preferencial)*, e armazena-se em uma lista;

2. Ordena-se a lista em ordem crescente;

3. Itera-se através de cada um dos elementos redimensionáveis da lista, tentando atribuir-lhes o tamanho *(preferencial - máximo)/(número de elementos redimensionáveis)*;

4. Para cada elemento que não suportou o tamanho calculado na etapa anterior, acumula-se a diferença em relação ao tamanho calculado e o tamanho mínimo ou máximo;

5. Distribui-se o tamanho acumulado na etapa anterior para os elementos elegíveis.

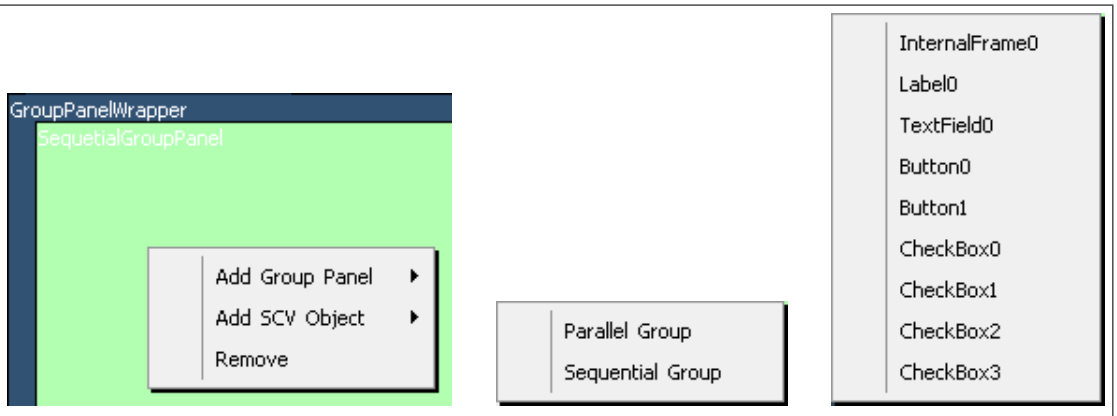
Calculado o tamanho de todos os elementos, é necessário ainda calcular a variável correspondente ao eixo do grupo, vertical ou horizontal, ou seja, aquela que dá o início de cada elemento (x ou y), o que é obtido através do acumulo dos tamanhos dos filhos. Este cálculo resulta na característica de arranjo sequencial do grupo.

Para um grupo definido **ParallelGroup**, onde os filhos devem ser arranjados de maneira paralela a um ponto âncora, a chamada **Group::setSize** distribui o tamanho calculado pelo **GroupLayout::layoutContainer** de maneira uniforme entre os elementos, diferenciando-se apenas no modo de alinhamento especificado no momento de criação do grupo, **CENTER** ou **LEADING**, já especificados nesta seção. Nota-se que esta diferença dá-se apenas no ponto de início (x ou y) dos elementos, assim, os algoritmos utilizados pelos modos a fim de determinar o ponto de início dos componentes são descritos abaixo:

1. **CENTER**: *ponto de origem do grupo + (tamanho calculado - springSize) / 2*, onde *"springSize"* é definido por *min(max(tamanho calculado, elemento->mínimo), elemento->máximo)*);

2. **LEADING**: *ponto de origem do grupo*.

A criação de um **GroupLayout** no *SCV Desginer* é realizada através de menus de contexto, os quais oferecem ao usuário uma maneira intuitiva de edição. Na Figura 4.7 é exibido alguns dos menus de contexto disponíveis.



(a) Menu de contexto invocado a partir da edição de um **Group**.

(b) Opção 'Add Group Panel' referente a Figura 4.7a.

(c) Opção 'Add SCV Object' referente a Figura 4.7a.

Figura 4.7: Sequência de imagens de menus de contexto invocados a partir da edição de um **Group**.

4.2.3.1 *Classe **Spring** e suas Especializações*

Classe base para **todos os elementos** oferecidos pelo **GroupLayout**, a **Spring** desempenha um papel vital no funcionamento do algoritmo, já que implementa e oferece uma interface de métodos que deve ser utilizada pelas classes herdadas.

Em Código 4.2 é determinado o conjunto mínimo de métodos que as classes herdadas devem implementar. Estes métodos são utilizados para o cálculo dos tamanhos mínimo, preferencial e máximo de cada elemento.

```
////////////////////////////////////////////////////////
virtual int calculateMinimumSize(Axis axis) = 0;
virtual int calculatePreferredSize(Axis axis) = 0;
virtual int calculateMaximumSize(Axis axis) = 0;
////////////////////////////////////////////////////////
```

Código 4.2: Detalhe de código exibindo os métodos virtuais puros da classe **Spring**.

O construtor da classe **Spring** exige a passagem dos parâmetros de tamanho mínimo, preferencial e máximo, informações que serão utilizadas pelas classes especializadas. É oferecida, ainda, a possiblidade da definição de valores padrões para estes parâmetros através das palavras-chave **DEFAULT_SIZE** e **PREFERRED_SIZE**. Na utilização da palavra-chave **DEFAULT_SIZE** a classe especializada deve retornar, quando requerida, o valor definido pelo elemento atrelado ou representado, como acontece com um **ComponentSpring**, descrito na Seção 4.2.3.1.3, que retorna o tamanho correspondente do **scv::Component** atrelado. Já na utilização da palavra-chave **PREFERRED_SIZE** deve ser priorizado, quando possível, o tamanho preferencial em relação aos demais.

Na Figura 4.8 é exibido o Diagrama de Classes da classe **Spring** e suas especializaçoes.

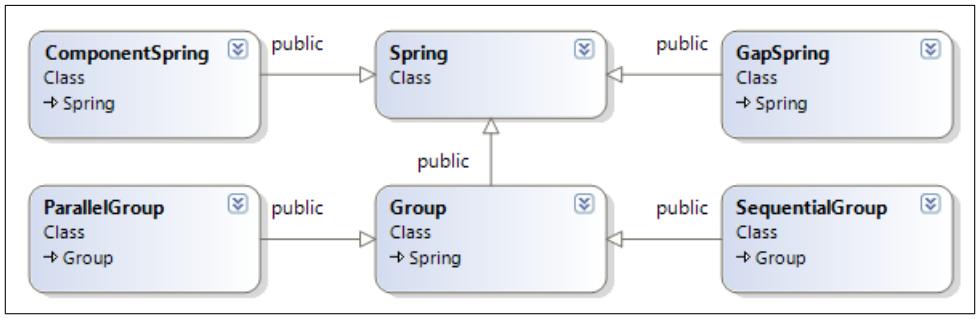


Figura 4.8: Diagrama da classes da classe **Spring** e suas especializações.

4.2.3.1.1 Especialização Classe **Group**

Como especificado na Seção 4.2.3, a classe **Group** implementa a interface utilizada pelas classes **SequentialGroup** e **ParallelGroup**.

Esta classe oferece os métodos de inserção para outros elementos em um grupo, como é exibido no Código 4.3.

```
////////////////////////////////////////////////////////
virtual Group *addGroup(Group *group);

virtual Group *addComponent(Component *component);
virtual Group *addComponent(Component *component, int size);
virtual Group *addComponent(Component *component, int min, int
   pref, int max);

virtual Group *addGap(int size);
virtual Group *addGap(int min, int pref, int max);
////////////////////////////////////////////////////////
```

Código 4.3: Detalhe de código exibindo os métodos de adição de elementos em um **Group**

O armazenamento destes elementos é implementado através de uma *std::list* de ponteiros para objetos da classe **Spring**, o que oferece a possibilidade de adição ou remoção de itens desta lista ao mesmo tempo em que se itera sobre ela. Esta característica é importante, e deve ser ressaltada, porque é a partir dela que se torna possível a edição de um **GroupLayout** em tempo de execução. Nota-se que o *SCV Desginer* faz uso desta característica para permitir o usuário criar um **GroupLayout** de maneira visual oferecendo as opções de adição ou remoção de quaisquer componentes.

Para garantir a remoção correta de um objeto da lista de armazenamento, é necessário, antes, iterar sobre todos os itens e verificar a potencial ocorrência do item a ser removido em outros grupos que estão armazenados no grupo em questão. Já que um **Group** é filho da classe **Spring** (tipo de ponteiro armazenado na lista), é necessário descobrir em tempo real se o item corrente da iteração é realmente um **Group** ou trata-se de outra especialização, para isto é utilizado um *dynamic_cast* da linguagem C++.

Um *dynamic_cast* é um operador que, diferentemente do *cast* estilo C, realiza uma checagem de validade do *cast* em tempo real e, caso os tipos não forem

compatíveis, uma exceção é gerada e o ponteiro NULL é retornado. A utilização deste operador demonstrou-se essencial no desenvolvimento do **GroupLayout** por este trabalhar com classes especializadas da classe **Spring** e que, por vezes, precisam ser diferenciadas após serem armazenadas em ponteiros de sua classe abstrata.

No Código 4.4 é exibida a utilização do operador *dynamic_cast* no método **Group::removeComponent**, utilizado para remover itens da lista de armazenamento.

```
void Group::removeComponent(scv::Component * object) {
  SpringsList::iterator iter = _springs.begin();
  while (iter != _springs.end()) {
    if (dynamic_cast<ComponentSpring *>(*iter) &&
       static_cast<ComponentSpring *>(*iter)->getComponent() ==
       object) {
      iter = _springs.erase(iter);
    } else if (dynamic_cast<Group *>(*iter)) {
      static_cast<Group *>(*iter)->removeComponent(object);
      ++iter;
    } else {
      ++iter;
    }
  }
}
```

Código 4.4: Trecho de código que demonstra a utilização do operador *dynamic_cast*.

4.2.3.1.2 Especialização Classe **GapSpring**

Um **GapSpring** pode ser pensado como um componente invisível de tamanho variável, mas ainda sim com todas as outras características comuns aos demais elementos. É utilizado em um **GroupLayout** para separar elementos visíveis ao usuário.

A criação automática de objetos do tipo **GapSpring** é oferecida como funcionalidade da classe **SequentialGroup**, o que auxilia o programar a criar uma interface mais agradável visualmente, já que normalmente deseja-se que, quando os componentes estão arranjados sequencialmente, exista uma separação entre eles.

4.2.3.1.3 Especialização Classe **ComponentSpring**

A classe **ComponentSpring** representa, em um **GroupLayout**, um objeto do tipo **scv::Component**, ou seja, é uma abstração de um componente qualquer da *API SCV*. Dentre todos os elementos herdados de **Spring**, este é o único em que há uma representação visual propriamente dita, já que o **GapSpring**, como descrito na Seção 4.2.3.1.2, é representado através de uma lacuna.

Como as demais classes herdadas de **Spring**, a classe **ComponentSpring** também implementa os métodos virtuais puros **calculate(Minimum** / **Preferred** / **Maximum)Size** e, para isto, utiliza as informações do **scv::Component** atrelado. Estas informações podem ser obtidas de duas formas:

1. Através dos métodos **scv::Component::get(Minimum** / **Preferred** / **Maximum)Size**;

2. Através dos valores escolhidos no momento de criação do **ComponentSpring**.

A escolha entre essas duas formas dá-se na configuração de propriedades pertencentes à classe pai **Spring**, as quais são descritas na Seção 4.2.3.1.

Todo componente da *API SCV* pode estar ou não visível através do método **scv::Component::setVisible**. Caso um componente atrelado à classe **ComponentSpring** não esteja visível o tamanho zero é retornado, o que auxilia na organização dinâmica dos elementos de um **GroupLayout**. Esta característica é utilizada pelo *SCV Desginer* para omitir elementos da interface gráfica que devem ser visíveis somente em determinadas situações, como por exemplo, na classe lógica **ObjectEditor**, onde devem ser exibidos paineis de configuração de componentes (Figura 4.3).

4.2.4 Object Editor

O **ObjectEditor** é responsável por oferecer uma interface de edição à componentes da *API SCV*, onde características especiais, como por exemplo, o texto de um **scv::TextField** ou de um **scv::Button**, e de tamanho mínimo, preferencial e máximo em um **GroupLayout** podem ser modificados, ou seja, é por meio desta

que serão configuradas as características a serem exportadas pelo ambiente em relação a aparência e comportamento.

Foi desenvolvido um conjunto de classes especializadas e agrupadas a fim de refletir a organização do *SCV*, de modo que características comuns a todos os componentes são tratadas pela classe **Properties** e características especializadas são tratadas por classes derivadas de **Properties**.

A escolha da classe derivada ocorre em tempo de execução através de métodos da classe **PropertiesManager**, a qual tem por função retornar a classe especializada de **Properties** adequada. Abaixo são listadas as classes especializadas existentes juntamente com a lista de componentes da *API SCV* que correspondem e as características atendidas.

1. **CountersProperties**: **scv::Slider**, **scv::Spinner**. Representa componentes derivados de **Counter**, os quais possuem valores mínimo, máximo, de início e de *step*;

2. **StringsProperties**: **scv::Button**, **scv::Label**, **scv::TextField**, **scv::TextBox**. Representa componentes que possuem um texto atrelado a sua exibição;

3. **StatesProperties**: **scv::CheckBox**, **scv::RadioButton**, **scv::ToggleButton**. Representa componentes que possuem um estado (verdadeiro ou falso);

4. **InternalFrameProperties**: **scv::InternalFrame**. Representa um **scv::InternalFrame**, o qual possui como características atendidas a largura, altura e texto do título;

5. **ImageProperties**: **scv::Image**. Representa um **scv::Image**, o qual possui como característica atendida o caminho da imagem a ser carregada.

Nota-se que os demais componentes do *SCV* são atendidos pela classe comum **Properties**, já que não possuem características especiais passíveis de modificação.

4.2.5 Geração de Código

Como especificado na Seção 3.1, Análise do Protótipo, a geração de código implementada segue uma estrutura lógica em que os dos componentes da *API SCV* foram divididos por suas características globais e específicas, ou seja, há um agrupamento de componentes que derivam de classes em comum. A estrutura proposta busca seguir a organização da API, de modo que alterações em classes ”pai” reflitam em mudanças nas classes ”filhas”. Esta estrutura facilita o reaproveitamento de código no caso de alterações da API e facilita a manutenção, já que evita a replicação do texto estático utilizado para a geração de código.

Após a adição de um **scv::Component** para exportação, realizado pelo método **addComponent**, da classe **CodeGenerator**, responsável por centralizar o controle da geração de código, é criado um objeto que encapsula este componente, o qual é chamado **ManagedComponent**.

A classe **ManagedComponent** é responsável por atrelar o **scv::Component** às características definidas pelo usuário para a exportação, gerenciar a ligação com outros objetos **ManagedComponent**, como quando um **scv::Panel** possui filhos, e gerar o código específico de cada componente do *SCV*. A geração de código específico dos componentes foi dividia em três etapas, como segue abaixo:

1. Código de Declaração;
2. Código de Implementação;
3. Código de Alocação.

Nota-se que as etapas propostas cobrem todas as possibilidades oferecidas para a criação de um **scv::Component** e, por consequência, pelo *SCV Desginer*.

Um **scv::Component** pode ser criado alocando-se um objeto diretamente da classe implementada no *SCV*, ou seja, não deriva-se classe alguma para esta criação, o que não permite a reimplementarão dos métodos de *callbacks* e sua utilização, ou, de modo contrário, deriva-se através da implementação de uma classe customizada. Esta opção é atendida através do método **ManagedComponent::-**

setCustomClass, o qual irá modificar as etapas de geração de código específico definidas anteriormente.

Caso o **ManagedComponent** não for especificado como classe customizada, o código de saída deve ser a alocação do componente diretamente das classes da *API SCV*, o que é feito pelo método **getAllocationCode**, e pode observada na Seção A.3. Nota-se que esta geração de código corresponde à etapa de **Código de Alocação**. Caso o **ManagedComponent** for especificado como classe customizada há necessidade da invocação das etapas de **Código de Declaração** e **Código de Alocação**.

Na etapa de **Código de Declaração** é gerada a declaração da classe customizada do componente através do método **getDeclarationCode**, de maneira que as *callbacks* específicas de cada componente da *API SCV* são obtidas através do método **getCustomDeclarationCode**. Na etapa de **Código de Implementação** é gerado o código de implementação corresponde à etapa de declaração. O resultado destas etapas para um caso exemplo podem ser observadas nas Seções A.4 e A.5.

Há ainda a necessidade da geração de código que é responsável pela inicialização da API, realizada pela classe gerada **Application**, a qual deriva de **scv::-Kernel**, e implementa os métodos de *callbacks* globais e inicializa os componentes gerados nas etapas anteriores, e pela função ”main” da linguagem C++, realizado através do método **generateCode** da classe **CodeGenerator**.

A geração de código para um caso exemplo completo pode ser observada no Apêndice A.

5 RESULTADOS E CONCLUSÃO

Este trabalho compreendeu uma séries de esforços para a criação de um ambiente para construção de interfaces gráficas de usuário com o *SCV*, juntamente com uma profunda reestruturação da API. O panorama final compreendeu um escopo muito mais amplo do que o pretendido na análise inicial, podendo-se considerar que todos os objetivos propostos foram satisfeitos.

O ambiente *SCV Designer*, cuja implementação final possui 38 arquivos e 2399 linhas de código fonte útil, mostrou-se relevante para os usuários do *SCV*, principalmente aqueles novatos em relação a estrutura da API, já que são oferecidos diversos recursos para a edição e exportação de aplicações funcionais.

O desenvolvimento de um esquema de arranjos de componentes, chamado **GroupLayout**, superou os resultados esperados, de modo que este demonstrou-se muito útil na criação de interfaces gráficas e no desenvolvimento com o *SCV*, já que o programador pode encarregar a API de realizar o trabalho antes maçante de alinhar e dimensionar componentes.

Finalmente, concluímos que os resultados alcançados por este trabalho foram de muita importância ao futuro desenvolvimento da *API SCV*, pois a partir deste, a API passou a oferecer um conjunto de funcionalidades e soluções semelhantes às oferecidas por softwares comerciais, o que garante que mais desenvolvedores interessem-se pelo projeto e auxiliam em seu crescimento.

5.1 Trabalhos Futuros

Ao analisarmos os ambientes de construção de interfaces gráficas comerciais e comparar ao desenvolvido neste trabalho, nota-se que ainda existem aspectos que não são considerados satisfatórios, principalmente o aspecto de que, nor-

malmente, há uma junção do desenvolvimento da interface gráfica e do código da aplicação a ser produzida.

Um trabalho futuro que agregaria muito valor à API seria o desenvolvimento de um *plugin* com o fim de integrar o *SCV* a ambientes de programação mais robustos, o que tornaria a criação de aplicações que utilizam o *SCV* para o controle da interface gráfica muito mais profissional.

REFERÊNCIAS

AVELAR, F. T.; GOMES, V. C. F.; POZZER, C. T. Estudo Comparativo de Bibliotecas Gráficas Integradas com OpenGL. **XXII Congresso Regional de Iniciação Científica e Tecnológica em Engenharia - CRICTE**, Passo Fundo, RS, BR, 2007.

BISHOP, J.; HORSPOOL, N. Developing Principles of GUI Programming Using Views. In: SIGCSE TECHNICAL SYMPOSIUM ON COMPUTER SCIENCE EDUCATION, 35., 2004. **Anais...** ACM-SIGCSE: Wiley, 2004. p.373–377.

BROWN, A. W. **Component Based Software Engineering**. [S.l.]: John Wiley & Sons, 1996.

GOTTIN, V. M. **Drawtex SCV: uma ferramenta educativa para a composição de primitivas gráficas opengl com exportação de código c/c++**. Santa Maria, RS, BR: Curso de Ciência da Computação. Universidade Federal de Santa Maria., 2010.

HARTSON, R. H.; HIX, D. Human-computer interface development: concepts and systems for its management. **ACM Comput. Surv.**, New York, NY, USA, v.21, n.1, p.5–92, Mar. 1989.

LIMBERGER, F. A.; PAHINS, C. A. L.; POZZER, C. T. SCV: uma ferramenta para programação de aplicativos de interfaces gráficas. **Jornada Acadêmica Integrada da Universidade Federal de Santa Maria. Santa Maria**, Santa Maria, RS, BR, 2010.

Microsoft Corporation. **Visual Studio Home | Microsoft Visual Studio**. Disponí-

vel em: http://www.microsoft.com/visualstudio/en-us/. Acesso em 1 Dezembro 2011.

MYERS, B. A. User Interface Software Tools. **ACM TRANSACTIONS ON COMPUTER-HUMAN INTERACTION**, [S.l.], v.2, p.64–103, 1993.

MYERS, B. A. Challenges of HCI design and implementation. **interactions**, [S.l.], v.I, n.1, 1994.

MYERS, B. A.; ROSSON, M. B. Survey On User Interface Programming. In: SIGCHI' 92, 1992, Monterey, CA, USA. **Anais...** ACM Press, 1992. p.195–202.

Oracle Corporation. **Oracle Technology Network for Java Developers**. Disponível em: http://www.oracle.com/technetwork/java/index.html/. Acesso em 22 Outubro 2011.

Oracle Corporation. **Welcome to NetBeans**. Disponível em: http://netbeans.org/. Acesso em 1 Dezembro 2011.

Oracle Corporation. **GroupLayout (Java Platform SE 6)**. Disponível em: http://docs.oracle.com/javase/6/docs/api/javax/swing/GroupLayout.html/. Acesso em 11 Dezembro 2011.

PAHINS, C. A. L.; LIMBERGER, F. A.; HENZ, B.; POZZER, C. T. SCV: uma ferramenta para programação de aplicativos de interfaces gráficas. **Congresso Regional de Iniciação Científica e Tecnológica em Engenharia**, Rio Grande, RS, BR, 2010.

PAHINS, C. A. L.; LIMBERGER, F. A.; HENZ, B.; SPERONI, E. A.; GOTTIN, V. M.; POZZER, C. T. Uma API Livre para Composição de GUI em APlicativos Gráficos. **Forum Internacional de Software Livre 2010 - Workshop de Software Livre**, Porto Alegre, RS, BR, 2010.

PAHINS, C. A. L.; LIMBERGER, F. A.; POZZER, C. T. Uma Abordagem Gráfica Utilizando o SCV. **Jornada Acadêmica Integrada da Universidade Federal de Santa Maria**, Santa Maria, RS, BR, 2010.

PAHINS, C. A. L.; POZZER, C. T. SCV - Simple Components for Visual. **Jornada Acadêmica Integrada da Universidade Federal de Santa Maria**, Santa Maria, RS, BR, 2011.

REZENDE, D. A. **Engenharia de Software e Sistema de Informação**. [S.l.]: Editora Brasport, 2005.

SCHMUCKER, K. J. **MacApp**: an application framework. [S.l.]: Byte, 1986. 189–193p.

SCV Team. **SCV - Simple Components for Visual**. Disponível em: `http://www.inf.ufsm.br/~pozzer/scv/`. Acesso em 1 Dezembro 2011.

The GTK+ Team. **gtkmm - C++ Interfaces for GTK+ and GNOME**. Disponível em: `http://www.gtkmm.org/en/`. Acesso em 1 Dezembro 2011.

The Linux Information Project. **GUI Definition**. Disponível em: `http://www.linfo.org/gui.html/`. Acesso em 22 Outubro 2011.

WRIGHT, R. S.; LIPCHAK, B.; HAEMEL, N. **OpenGL® SuperBible**: comprehensive tutorial and reference. 4ª.ed. [S.l.]: Addison-Wesley Professional, 2007.

wxWidgets Developers and Contributors. **wxWidgets - Cross-Plataform GUI Library**. Disponível em: `http://www.wxwidgets.org/`. Acesso em 1 Dezembro 2011.

APÊNDICE A SAÍDA DA GERAÇÃO AUTOMÁTICA DE CÓDIGO DO SCV DESIGNER

Caso exemplo baseado na interface exibida na Figura 4.1 e no **GroupLayout** exibido na Figura 4.2. Nota-se que o código gerado pelo *SCV Designer* depende inteiramente de parâmetros configurados pelo usário, assim, a saída aqui apresentada corresponde unicamente ao exemplo construido.

A.1 Implementação da Função 'main'

```
#include "SCV.h"
#include "Application.h"
#include "Widget.h"

int main(int argc, char* argv[]) {
  scv::Kernel::setInstance(new Application());
  Application *kernel =
     static_cast<Application*>(scv::Kernel::getInstance());

  kernel->init();

  kernel->run();
  return 0;
}
```

Código A.1: Implementação da Função 'main'.

A.2 Declaração da Classe Application

```
#ifndef __APPLICATION_H__
#define __APPLICATION_H__

class Application : public scv::Kernel {
public:
  ////////////////////////////////////////////////////////////
```

```
  Application(void);
  virtual ~Application(void);
  ////////////////////////////////////////////////////////////

  void init(void);

  //SCVCallbacks
  ////////////////////////////////////////////////////////////
  virtual void onMouseClick(const scv::MouseEvent &evt);
  virtual void onMouseHold(const scv::MouseEvent &evt);
  virtual void onMouseOver(const scv::MouseEvent &evt);
  virtual void onMouseUp(const scv::MouseEvent &evt);
  virtual void onMouseWheel(const scv::MouseEvent &evt);

  virtual void onKeyPressed(const scv::KeyEvent &evt);
  virtual void onKeyUp(const scv::KeyEvent &evt);

  virtual void onSizeChange(void);
  virtual void onPositionChange(void);
  ////////////////////////////////////////////////////////////

protected:
  static const int s_defaultWindowWidth = 1280;
  static const int s_defaultWindowHeight = 720;
  scv::Panel *_mainPanel;
private:
};

#endif //__APPLICATION_H__
```

Código A.2: Declaração da Classe **Application**.

A.3 Implementação da Classe Application

```
#include "Application.h"

Application::Application(void) : Kernel() {
  setWindowSize(400, 100);
  lockWindowSize(true);
  setFramesPerSecond(60);

  setWindowTitle("SCV Application");
}

Application::~Application(void) {
}

void Application::init(void) {
```

```
_mainPanel = new scv::Panel(scv::Point(10, 10), scv::Point(0,
   0));
scv::GroupLayout *layout = new scv::GroupLayout(_mainPanel);
_mainPanel->setLayout(layout);

scv::Label *label0 = new scv::Label(scv::Point(0, 0), "Find
   What");
_mainPanel->addComponent(label0);
TextField0 *textfield0 = new TextField0();
_mainPanel->addComponent(textfield0);
CheckBox0 *checkbox0 = new CheckBox0();
_mainPanel->addComponent(checkbox0);
scv::CheckBox *checkbox1 = new scv::CheckBox(scv::Point(0,
   0), 0, "Whole Words");
_mainPanel->addComponent(checkbox1);
scv::CheckBox *checkbox2 = new scv::CheckBox(scv::Point(0,
   0), 0, "Wrap Around");
_mainPanel->addComponent(checkbox2);
scv::CheckBox *checkbox3 = new scv::CheckBox(scv::Point(0,
   0), 0, "Search Backwards");
_mainPanel->addComponent(checkbox3);
Button0 *button0 = new Button0();
_mainPanel->addComponent(button0);
scv::Button *button1 = new scv::Button(scv::Point(0, 0),
   "Cancel");
_mainPanel->addComponent(button1);

layout->setHorizontalGroup(
  scv::GroupLayout::createParallelGroup()
    ->addGroup(scv::GroupLayout::createSequentialGroup()
    ->setAutoCreateGaps(true)
      ->addComponent(label0)
      ->addGroup(scv::GroupLayout::createParallelGroup()
        ->addComponent(textfield0)
        ->addGroup(scv::GroupLayout::createSequentialGroup()
        ->setAutoCreateGaps(true)
          ->addGroup(scv::GroupLayout::createParallelGroup()
            ->addComponent(checkbox0)
            ->addComponent(checkbox1)
          )
          ->addGroup(scv::GroupLayout::createParallelGroup()
            ->addComponent(checkbox2)
            ->addComponent(checkbox3)
          )
        )
      )
      ->addGroup(scv::GroupLayout::createParallelGroup()
        ->addComponent(button0)
        ->addComponent(button1)
      )
    )
```

```
  );

  layout->setVerticalGroup(
    scv::GroupLayout::createParallelGroup()
      ->addGroup(scv::GroupLayout::createSequentialGroup()
      ->setAutoCreateGaps(true)
        ->addGroup(scv::GroupLayout::createParallelGroup()
          ->addComponent(label0)
          ->addComponent(textfield0)
          ->addComponent(button0)
        )
        ->addGroup(scv::GroupLayout::createParallelGroup()
          ->addGroup(scv::GroupLayout::createSequentialGroup()
          ->setAutoCreateGaps(true)
            ->addComponent(checkbox0)
            ->addComponent(checkbox1)
          )
          ->addGroup(scv::GroupLayout::createSequentialGroup()
          ->setAutoCreateGaps(true)
            ->addComponent(checkbox2)
            ->addComponent(checkbox3)
          )
          ->addComponent(button1)
        )
      )
  );
}

void Application::onMouseClick(const scv::MouseEvent &evt) {
}
void Application::onMouseHold(const scv::MouseEvent &evt) {
}
void Application::onMouseOver(const scv::MouseEvent &evt) {
}
void Application::onMouseUp(const scv::MouseEvent &evt) {
}
void Application::onMouseWheel(const scv::MouseEvent &evt) {
}

void Application::onKeyPressed(const scv::KeyEvent &evt) {
}
void Application::onKeyUp(const scv::KeyEvent &evt) {
}

void Application::onSizeChange(void) {
  _mainPanel->setSize(getWidth() - 20, getHeight() - 20);
}
void Application::onPositionChange(void) {
}
```

Código A.3: Implementação da Classe **Application**.

A.4 Declaração dos Componentes

```
#ifndef __WIDGET_H__
#define __WIDGET_H__

class TextField0 : public scv::TextField {
public:
  ////////////////////////////////////////////////////////
  TextField0(void);
  virtual ~TextField0(void);
  ////////////////////////////////////////////////////////

  ////////////////////////////////////////////////////////
  virtual void onMouseClick(const scv::MouseEvent &evt);
  virtual void onMouseHold(const scv::MouseEvent &evt);
  virtual void onMouseOver(const scv::MouseEvent &evt);
  virtual void onMouseUp(const scv::MouseEvent &evt);
  virtual void onMouseWheel(const scv::MouseEvent &evt);

  virtual void onKeyPressed(const scv::KeyEvent &evt);
  virtual void onKeyUp(const scv::KeyEvent &evt);

  virtual void onSizeChange(void);
  virtual void onPositionChange(void);

  virtual void onStringChange(void);
  ////////////////////////////////////////////////////////
};

class CheckBox0 : public scv::CheckBox {
public:
  ////////////////////////////////////////////////////////
  CheckBox0(void);
  virtual ~CheckBox0(void);
  ////////////////////////////////////////////////////////

  ////////////////////////////////////////////////////////
  virtual void onMouseClick(const scv::MouseEvent &evt);
  virtual void onMouseHold(const scv::MouseEvent &evt);
  virtual void onMouseOver(const scv::MouseEvent &evt);
  virtual void onMouseUp(const scv::MouseEvent &evt);
  virtual void onMouseWheel(const scv::MouseEvent &evt);

  virtual void onKeyPressed(const scv::KeyEvent &evt);
  virtual void onKeyUp(const scv::KeyEvent &evt);

  virtual void onSizeChange(void);
  virtual void onPositionChange(void);

  virtual void onValueChange(void);
```

```
  ////////////////////////////////////////////////////////////
};

class Button0 : public scv::Button {
public:
  ////////////////////////////////////////////////////////////
  Button0(void);
  virtual ~Button0(void);
  ////////////////////////////////////////////////////////////

  ////////////////////////////////////////////////////////////
  virtual void onMouseClick(const scv::MouseEvent &evt);
  virtual void onMouseHold(const scv::MouseEvent &evt);
  virtual void onMouseOver(const scv::MouseEvent &evt);
  virtual void onMouseUp(const scv::MouseEvent &evt);
  virtual void onMouseWheel(const scv::MouseEvent &evt);

  virtual void onKeyPressed(const scv::KeyEvent &evt);
  virtual void onKeyUp(const scv::KeyEvent &evt);

  virtual void onSizeChange(void);
  virtual void onPositionChange(void);
  ////////////////////////////////////////////////////////////
};

#endif //__WIDGET_H__
```

Código A.4: Declaração dos Componentes.

A.5 Implementação dos Componentes

```
#include "Widget.h"

////////////////////////////////////////////////////////////
TextField0::TextField0(void) : scv::TextField(scv::Point(0, 0),
   200, "") {
}
TextField0::~TextField0(void) {
}

void TextField0::onMouseClick(const scv::MouseEvent &evt) {
}
void TextField0::onMouseHold(const scv::MouseEvent &evt) {
}
void TextField0::onMouseOver(const scv::MouseEvent &evt) {
}
void TextField0::onMouseUp(const scv::MouseEvent &evt) {
}
void TextField0::onMouseWheel(const scv::MouseEvent &evt) {
```

```
}

void TextField0::onKeyPressed(const scv::KeyEvent &evt) {
}
void TextField0::onKeyUp(const scv::KeyEvent &evt) {
}

void TextField0::onSizeChange(void) {
}
void TextField0::onPositionChange(void) {
}

void TextField0::onStringChange(void) {
}
////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////
CheckBox0::CheckBox0(void) : scv::CheckBox(scv::Point(0, 0), 0,
   "Match Case") {
}
CheckBox0::~CheckBox0(void) {
}

void CheckBox0::onMouseClick(const scv::MouseEvent &evt) {
}
void CheckBox0::onMouseHold(const scv::MouseEvent &evt) {
}
void CheckBox0::onMouseOver(const scv::MouseEvent &evt) {
}
void CheckBox0::onMouseUp(const scv::MouseEvent &evt) {
}
void CheckBox0::onMouseWheel(const scv::MouseEvent &evt) {
}

void CheckBox0::onKeyPressed(const scv::KeyEvent &evt) {
}
void CheckBox0::onKeyUp(const scv::KeyEvent &evt) {
}

void CheckBox0::onSizeChange(void) {
}
void CheckBox0::onPositionChange(void) {
}

void CheckBox0::onValueChange(void) {
}
////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////
Button0::Button0(void) : scv::Button(scv::Point(0, 0), "Find") {
}
```

```
Button0::~Button0(void) {
}

void Button0::onMouseClick(const scv::MouseEvent &evt) {
}
void Button0::onMouseHold(const scv::MouseEvent &evt) {
}
void Button0::onMouseOver(const scv::MouseEvent &evt) {
}
void Button0::onMouseUp(const scv::MouseEvent &evt) {
}
void Button0::onMouseWheel(const scv::MouseEvent &evt) {
}

void Button0::onKeyPressed(const scv::KeyEvent &evt) {
}
void Button0::onKeyUp(const scv::KeyEvent &evt) {
}

void Button0::onSizeChange(void) {
}
void Button0::onPositionChange(void) {
}
////////////////////////////////////////////////////////////
```

Código A.5: Implementação dos Componentes.