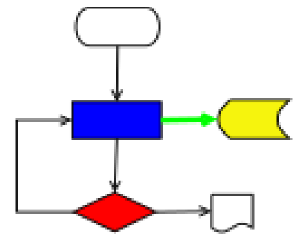
VICI



VISUAL CHART INTERPRETER

Design for libcanvas

Publication History

Date	Who	What Changes
20 September 2014	Brenton Ross	Initial version.

Copyright © 2009 - 2014 Brenton Ross
This work is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported License.
The software is released under the terms of the GNU General Public License version 3.

Table of Contents

1 Introduction....4
1.1 Scope....4
1.2 Overview....4
1.3 Audience....4
2 Overview....5
2.1 Responsibilities....5
2.2 Interfaces....6
2.2.1 IF04 Canvas UI....6
2.2.2 IF19 Canvas using Symbol....6
2.2.3 IF20 Vici-ed using Canvas....6
2.2.4 IF30 Canvas Library using libQtGui....7
2.2.5 IF31 Canvas Library using libcfi....7
2.3 Design Approach....8
2.3.1 Symbols....8
2.3.2 Lines....8
2.3.3 Validation....8
2.3.4 XML....9
3 User Interface....10
4 Application Design....12
5 Collaboration Diagrams....13
5.1 Place a Symbol....13
5.2 Drag an Item....13
5.3 Connecting Two Symbols....14
6 Class Designs....16
6.1 StateVariable Class....16
6.2 GraphicsScene Class....16
6.3 CanvasView Class....17
6.4 Comment Class....17
6.5 Item Class....17
6.6 ItemDialog Class....18
6.7 ItemFactory Class....18
6.8 Corner Class....19
6.9 LineSegment Class....19
6.10 Line Class....20
6.11 Page Class....21
6.12 CanvasImpl Class....22
6.13 CanvasFactoryImpl Class....23
6.14 ScriptXml Class....23
6.15 Item Classes....24
6.16 Item Dialog Classes....25
Appendix A....26

1 Introduction

This is part of the system design document for the VICI project.

1.1 Scope

This document covers the detailed design of the canvas component. This component is responsible for allowing the user to edit the flow chart diagrams.

The document will cover the Application Design and the User Interface Design.

This design is for increment #1. It will require enhancement for subsequent increments.

1.2 Overview

The detailed design includes:

- Interface Stubs: A framework of facade classes for the modules.
- Use Case Descriptions: A description of how a user is expected to interact with the application.
- Application Design: The classes and their relationships.
- User Interface Design: The design and layout of the graphical components of the system.
- Persistent Storage Design: The specifications for the XML files used to store configuration and scripts.

1.3 Audience

This document is intended to be used by the designers and developers, and later the maintainers, of the VICI project.

2 Overview

2.1 Responsibilities

This component provides a canvas on which the user can create a flow chart. The component is responsible for saving the chart to an XML file, and is able to restore a previously saved chart.

This is the list for increment #1. It will be extended for subsequent increments.

It addresses the following responsibilities:

T3.3: Place default symbol on diagram.

T3.5: Indicate a symbol has been selected.

T3.6: Move selected symbols to new positions on the diagram following the mouse.

T3.7: Keep lines connected to other symbols.

T3.8: Remove deleted symbols.

T3.9: Save the flowchart as an XML file.

T3.10: Reconstruct a chart from the contents of an XML file.

T7.2: Allow the user to define the name of a function.

T9.2: Collect the text from the user and place it on the diagram.

T9.3: Save the text, its location and attributes, into the diagram XML file.

T9.4: Restore the text to the diagram from an XML file.

T9.5: Allow the user to edit the text, or its attributes.

T9.6: Allow the removal of text from a diagram.

2.2 Interfaces

This is the list of interfaces for increment #1. It will have an interface to the interpreter for increment #2.

2.2.1 IF04 Canvas UI

This is the user interface for the canvas, the component responsible for allowing the user to create and edit the diagrams.

Transport Medium: Displayed in a window.

Protocol: Event driven with Windows, Icons, Menus and a Pointer.

Content:

1. Flowchart diagram. (O)
2. Current selected symbols. (O)
3. User symbol selection. (I)
4. Desired position of symbol(s). (I)
5. User edit action (I)
6. Name of function. (I/O)
7. User text (I/O)
8. Location of execution points (O)

2.2.2 IF19 Canvas using Symbol

This is the interface between the Canvas component and the Symbol component.

Transport Medium: Memory

Protocol: C++ function calls., Qt signals and slots

Content:

1. Drawing surface (I)
2. Position of required symbol (I)
3. Currently selected symbol (O)
4. Attributes of text and symbols. (O)

2.2.3 IF20 Vici-ed using Canvas

This is the interface between then vici-ed program and the component responsible for the layout and display of the diagram.

Transport Medium: Memory

Protocol: C++ function calls, Qt signals and slots.

Content:

1. Sub-window for the canvas to use. (I)
2. Menu actions. (I)

3. Command descriptions (I)
4. Currently selected symbol. (O)
5. Thread execution point. (I)

2.2.4 IF30 Canvas Library using libQtGui

This interface allows the canvas to display the diagram and interact with the user to edit it.

Transport Medium: Memory

Protocol: C++ function calls.

Content:

1. Drawing operations on a QGraphicsScene canvas.

2.2.5 IF31 Canvas Library using libcfi

This is the interface that the canvas component uses to load and save the scripts.

Transport Medium: Memory

Protocol: C++ function calls.

Content:

1. Create an XML document structure.
2. Add nodes to the document.
3. Add properties to nodes.
4. Save the document.
5. Read an XML file
6. Get nodes from the document
7. Remove nodes from the document.
8. Get properties from the document node.

2.3 Design Approach

Prototype C++ code has been constructed that allows the user to create a flow chart, and save and load the script as XML.

For the prototype a test harness simulated the symbol library, and provided menus for setting the state of the application.

2.3.1 Symbols

The symbols that will eventually be provided by libsymbol were provided by the test application for the prototype. These symbols are managed by instances of the Item class.

Each type of symbol is managed by a subclass of Item, such as ChoiceItem. There is an associated dialog box that allows the user to set the properties for each symbol. This also allows the user to set the command and options without having to start libcommand or libsearch.

2.3.2 Lines

The canvas library is directly responsible for drawing the lines interconnecting the symbols.

The lines are drawn according to their style, with colour and thickness set appropriately. It was initially intended that the lines would have arrow heads that would be at the destination symbol. However, the irregular design of some of the symbols made this problematic so the arrows are now draw at the mid point of each line segment. I think this actually looks better.

Lines are made up of straight segments with corner objects linking them. A line representing a flow of control is allowed to merge with an existing line by terminating it on an existing corner. It is important that lines are removed in the opposite order than they were constructed, otherwise a line will be left dangling, and connected to a non-existent corner.

A gradient is applied to the line segments and corners when they are selected to give them a glow effect. (Perhaps we should do something similar for the symbol items.)

2.3.3 Validation

There is some code that attempts to validate the connections and prevent the worst of the possible errors. More is needed, specifically we need to ensure the items at the ends of a line can be sensibly connected – at the moment it is possible to connect two variables even though this would not achieve anything.

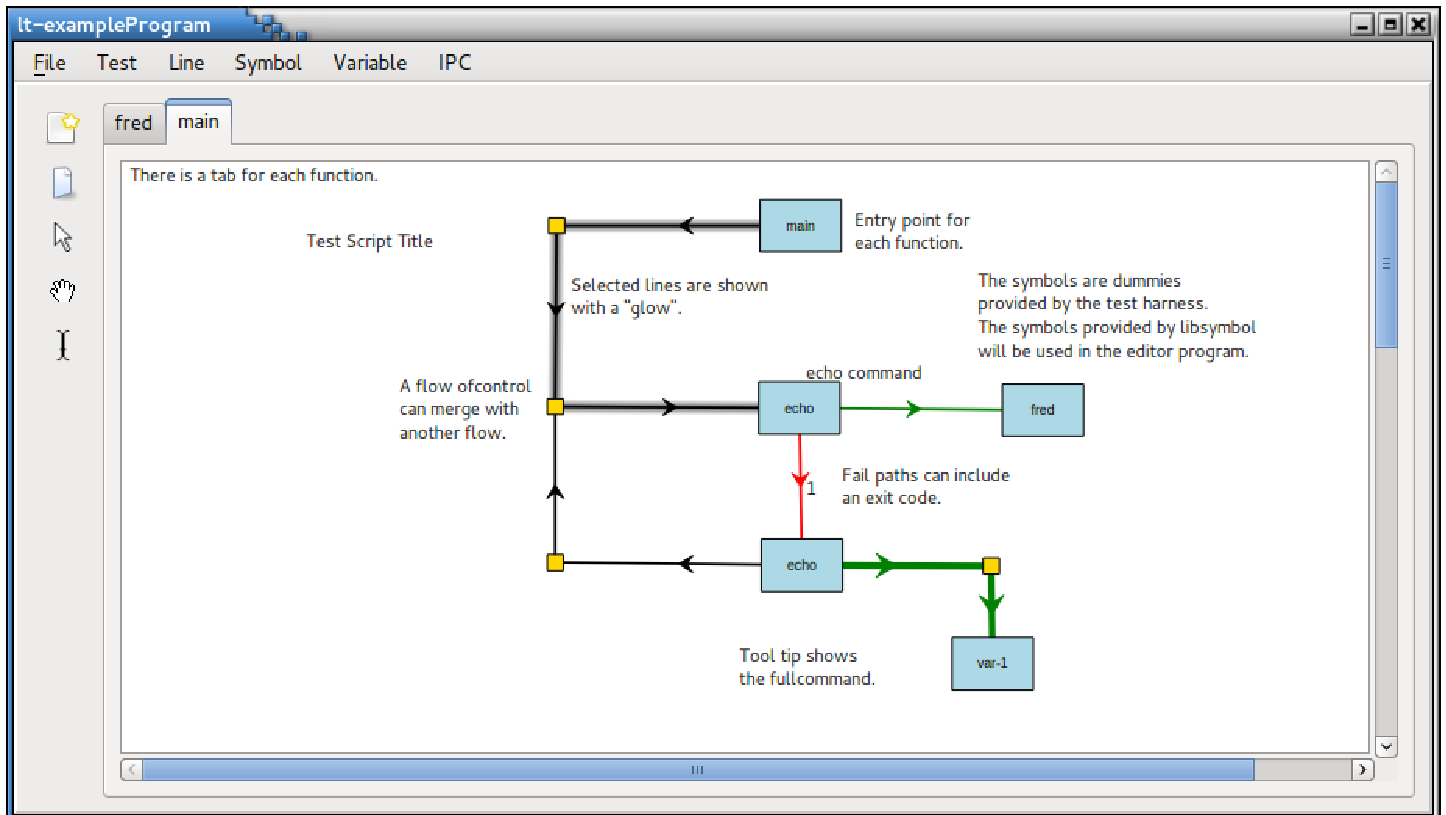
2.3.4 XML

Saving the script to an XML file was quite straightforward. Loading it back and reconstructing the diagram turned out to be a bit more problematic, requiring new constructors for most items.

Currently a new XML document is created when the object is saved. This may defeat designs that require other parts of the VICI editor to modify the document – creating a menu, for example. It may be better to retain the XML document when loading a file and just reconstruct the appropriate parts when saving.

3 User Interface

The following diagram illustrates the user interface for the canvas component.



The component has two sections for its user interface. On the left is a tool bar, and the right is the main drawing area with a separate tabbed page for each function.

The tool bar icons are used as follows (from the top down)

- to start a new diagram,
- to start a new function on a new page,
- to put the mouse into selection mode (the default),
- to put the mouse into move mode for dragging the displayed area
- to put the mouse into text mode for the entry of text comments.

Symbols are placed on the diagram by first selecting the desired symbol from the symbol palette provided by libsymbol. A mouse click will then place the symbol on the diagram.

Lines are constructed by first selecting the type of line from the symbol palette and then selecting the start item. The system will then draw a line from the selected item to the current mouse position. Clicking on a second item will complete the line. Clicking on the space between items will create a corner and

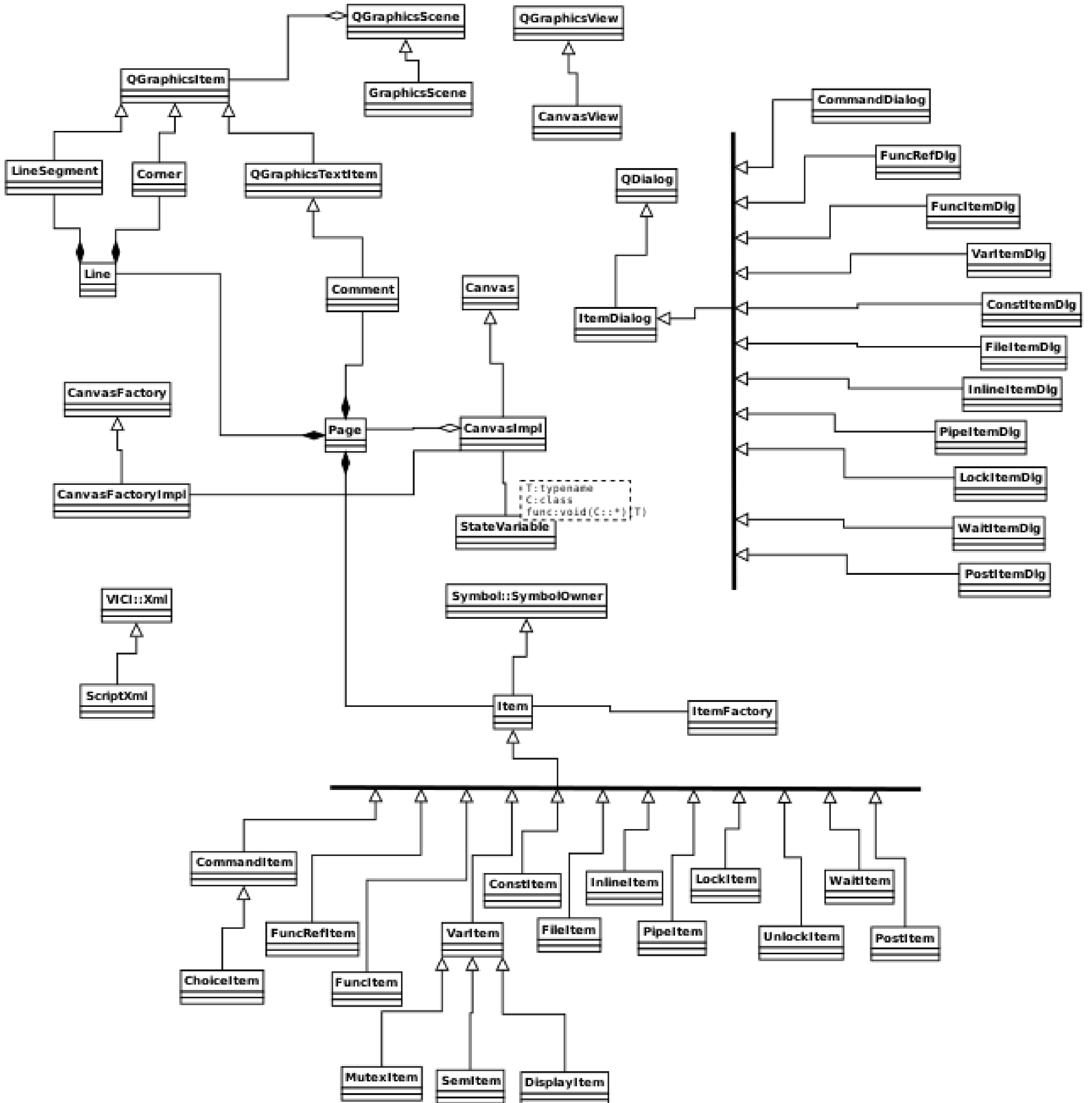
start a new line segment. This is repeated until an item is clicked. If the end item is the same as the start item the line is immediately removed. Flow of control lines may also be terminated by clicking on an existing corner of a different line (of a compatible type).

Any item or corner may be moved by selecting and dragging (with the mouse button held down). A group of items may be selected by dragging the mouse to create a rubber-band rectangle enclosing the group. Once selected they can be moved as a group.

Items and lines have context menus. This can be used to remove an item or line, open the properties dialog for the object, or for commands open the command or search windows. Items can only be removed when they have no connections. Lines can only be removed if no other line terminates on one of its corners.

4 Application Design

The following diagram describes the relationships between the classes for the canvas component.

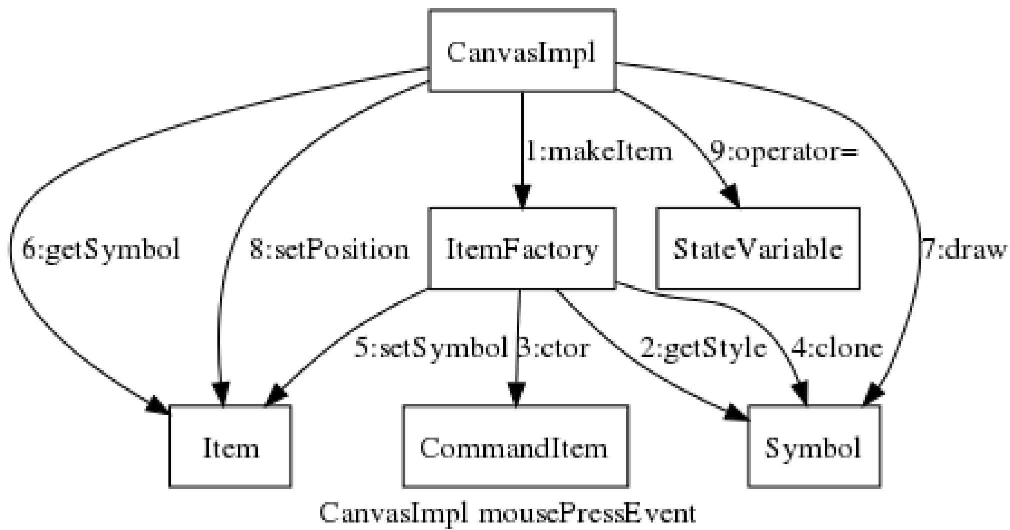


5 Collaboration Diagrams

The following diagrams illustrate the interactions between the objects for the important use cases.

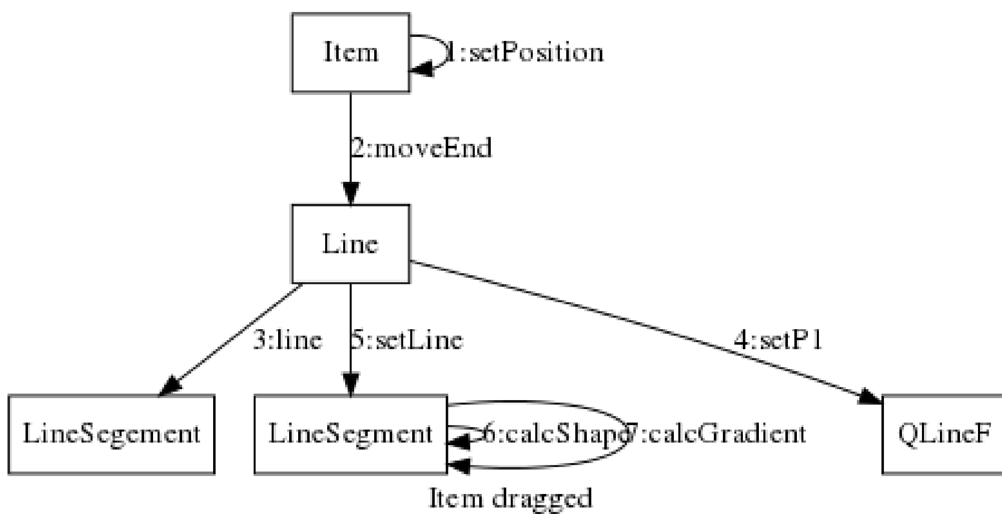
5.1 Place a Symbol

When the selects a command symbol and then clicks on the page the following events occur:



5.2 Drag an Item

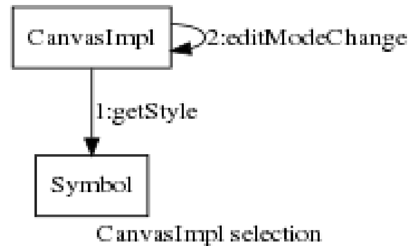
When an item is repositioned the following events typically occur:



5.3 Connecting Two Symbols

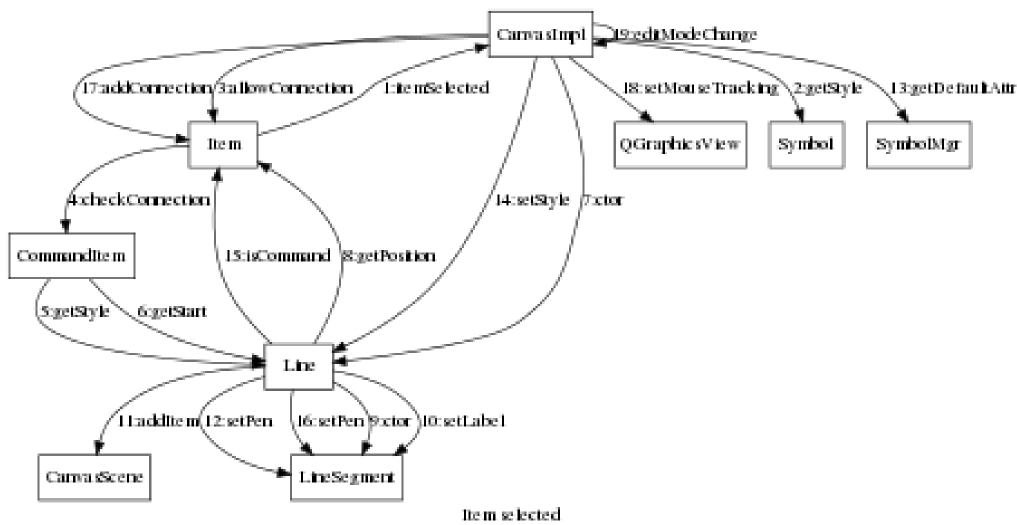
The following diagrams describes the sequence of events when the user connects two symbols with a line containing a corner.

The sequence starts when the user selects a line symbol from the symbol palette.



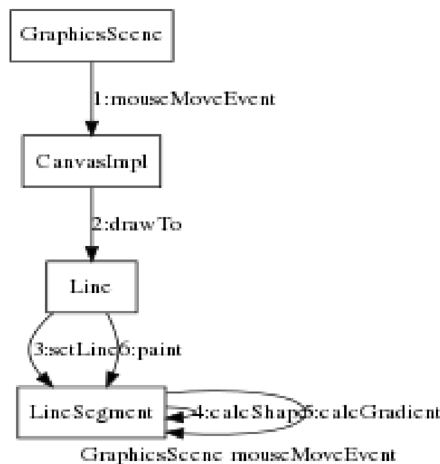
CanvasImpl selection

Next the user selects an item to start the line from.



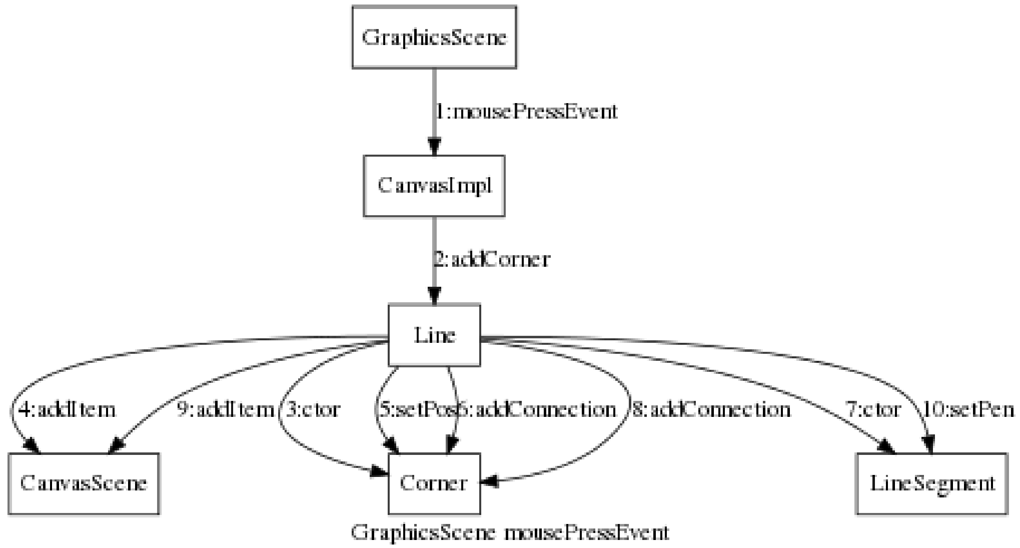
Item selected

The user then moves the mouse and the line is extended from the item to the mouse position.



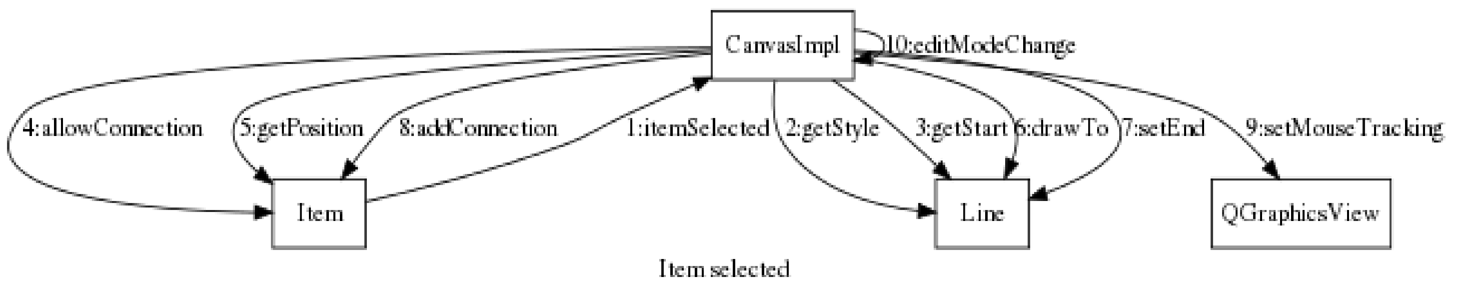
GraphicsScene mouseMoveEvent

If the user presses the mouse in an open space the system inserts a corner.



GraphicsScene mousePressEvent

Finally the user selects a second item and the line is connected.



Item selected

6 Class Designs

This section describes each class, including its responsibilities, and its public and protected members.

6.1 StateVariable Class

This is a template class that appears in the code as a simple variable, but which calls a method on an object if its value is changed. The idea is that event handlers can simply change this state variable. The consequences of the state change are then passed on to the called object.

```
template< typename T, class C, void(C::*func)(T) >
class StateVariable
{
private:
	T var;
	C *obj;
public:
	StateVariable(T x, C *y) : var(x), obj(y) {}
	operator T () { return var; }
	T operator = (T x)
	{
		if ( var != x )
		{
			var = x;
			(obj->*func)(var);
		}
		return var;
	}
};
```

6.2 GraphicsScene Class

This is a specialisation of QGraphicsScene that passes mouse press and mouse move events to the main canvas class.

```
class GraphicsScene : public QGraphicsScene
{
private:
	CanvasImpl * canvas;	// reference
public:
	GraphicsScene(CanvasImpl *);
	void mousePressEvent ( QGraphicsSceneMouseEvent * mouseEvent );
	void mouseMoveEvent ( QGraphicsSceneMouseEvent * mouseEvent );
};
```

6.3 CanvasView Class

This is a specialisation of the QGraphicsView class so that we can implement zooming using the mouse wheel.

```
class CanvasView : public QGraphicsView
{
public:
	CanvasView(QWidget* parent = NULL);
protected:
	//Take over the interaction
	virtual void wheelEvent(QWheelEvent* event);
};
```

6.4 Comment Class

This is a specialisation of the QGraphicsTextItem class that is used to hold comments that the user can place anywhere on the flow chart. It includes a test to see if the comment is empty. Whenever the state of the application is changed any empty comments are removed.

```
class Comment : public QGraphicsTextItem
{
public:
	Comment();
	bool isEmpty();
};
```

6.5 Item Class

This is the base class for the set of classes that manage symbols on the chart. There is a child class for each type of symbol. It implements the SymbolOwner interface so that it can be advised of user events on the displayed symbol.

It is responsible for keeping a list of connections. It has a method for checking if a connection is allowed, but the implementation is delegated to the child classes.

```
class Item : public QObject, public VICI::Symbol::SymbolOwner
{
	Q_OBJECT
protected:
	static int nodeCounter;
	int nodeId;
	CanvasImpl *canvas;		// reference
	VICI::ArgList args;		// owned
	VICI::Symbol::Symbol *symbol;	// owned
	QPointF position;
	std::list<Line *> connections;	// references
	QMenu *contextMenu;			// owned
	QAction *delAct, *propAct;	// owned

	virtual void makeMenu();
	virtual bool checkConnection(VICI::Symbol::Style,
		bool atStart, QString & reason );

protected slots:
```

```
	virtual void delAction();
	virtual void propAction();

public:
	Item( CanvasImpl *tc);
	Item( int id, CanvasImpl *tc );
	virtual ~Item();
	int getId();
	void setSymbol(VICI::Symbol::Symbol *s);
	Symbol::Symbol *getSymbol();
	void setArgs(VICI::ArgList &options );
	virtual void selected();
	virtual void opened();
	virtual void dragged( double x, double y );
	QPointF getPosition();
	void setPosition( QPointF p);
	bool allowConnection(VICI::Symbol::Style, bool atStart );
	void addConnection( Line * line) ;
	void delConnection( Line * line);
	void getLines( Symbol::Style, std::vector< Line *> & );
	virtual bool isCommand() const;
	virtual void writeXml( ScriptXml * ) = 0;

	Item( const Item &) = delete;
	void operator = (const Item &) = delete;
};
```

6.6 ItemDialog Class

This is a base class for the set of dialogs for the item classes. It provides a button box for OK and Cancel buttons, and a top level layout.

```
class ItemDialog : public QDialog
{
protected:
	QVBoxLayout *topLayout;
	void addButtons();		// adds OK and Cancel buttons
public:
	ItemDialog(QString title, QWidget *parent);
};
```

6.7 ItemFactory Class

This object is responsible for creating an Item object according to the specified symbol.

```
class ItemFactory
{
public:
	ItemFactory(){}
	Item * makeItem( VICI::Symbol::Symbol *, CanvasImpl *);
};
```

6.8 Corner Class

This class is a specialisation of QGraphicsItem that represents a corner between two line segments. It maintains a list of line segments that connect to it and has a method to check to ensure that a line is allowed to connect to it.

```
class Corner : public QGraphicsItem
{
private:
	CanvasImpl *canvas;
	QRectF shape;
	QPen pen;
	QBrush brush;
	std::list< std::pair<bool, LineSegment *> >connections;
	Line * line;
public:
	Corner(CanvasImpl *canvas, Line *);
	~Corner();
	void addConnection( LineSegment * line, bool start);
	void delConnection( LineSegment * );
	bool allowConnection(VICI::Symbol::Style );
	int numConnections() const;
	Line *getLine();
	virtual void paint( QPainter *painter,
		const QStyleOptionGraphicsItem * option,
		QWidget * widget );
	virtual QRectF boundingRect() const;
	QVariant itemChange ( GraphicsItemChange change,
		const QVariant & value );
};
```

6.9 LineSegment Class

This class is a specialisation of QGraphicsItem that represents the straight lines that make up a line connecting two symbols. each line segment has an arrow head at its mid point and when selected a gradient is drawn that gives the impression the line is glowing.

A line segment includes a QGraphicsTextItem as a child that contains the label. Normally only the first line segment of a line has a label.

```
class LineSegment : public QGraphicsItem
{
private:
	Line *owner;
	QLineF mLine;
	QPolygonF arrow, arrowShape;
	QPolygonF lineShape;
	QGraphicsTextItem *label;
	QPen pen;
	QBrush brush;
	QLinearGradient grad;
	bool isSelected;

	void calcGradient();      // calculate the gradient
	void calcShape();         // calculate the shape surrounding
	                          // the line segment

public:
	LineSegment(Line *);
```

```
      ~LineSegment();

      void setIsSelected(bool x);
      void setLine( QLineF );
      QLineF line();
      void setLabel( QString text);
      std::string getLabel();
      void setPen( QPen );
      virtual void paint( QPainter *painter,
             const QStyleOptionGraphicsItem * option,
             QWidget * widget );
      virtual QRectF boundingRect() const;
      virtual QPainterPath shape() const;
      QVariant itemChange ( GraphicsItemChange change,
             const QVariant & value );
      void contextMenuEvent(QGraphicsSceneContextMenuEvent *event);
};
```

6.10 Line Class

This class is a container for line segments and corners that are drawn between two items or an item and a corner of another compatible line.

```
class Line : public QObject
{
      Q_OBJECT
private:
      static const int THIN_LINE_WIDTH = 2;
      static const int THICK_LINE_WIDTH = 5;
      CanvasImpl *canvas;              // reference
      VICI::CanvasScene *scene;        // reference
      Item * start;                    // reference
      Item * end;                      // reference - used when line
                                       //     ends on an item
      Corner * endCorner;              // reference - used when line
                                       //     ends on a corner
      std::vector< QGraphicsItem * > items; // owned
      QPointF anchor;
      LineSegment *currentLine; // owned;
      QPen pen;
      QString label;
      Symbol::Style style;
      bool hasExitCode;
      QMenu *contextMenu;
      QAction *delAct, *propAct;

      void setLabel(int);

private slots:
      void delAction();
      void propAction();
public:
      Line(CanvasImpl *, VICI::CanvasScene *, Item *begin);
      Line(CanvasImpl *, VICI::CanvasScene *, Symbol::Style,
             Symbol::Colour col, Item *fromItem,
             Item * toItem,
             int exitCode, std::vector< QPointF > & );
      ~Line();
      void setStyle( Symbol::Style, QColor );
      Symbol::Style getStyle() { return style; }
      void setEnd(Item *);
      void setEnd(Corner *);
```

```
	Item * getStart() const;
	Item * getEndItem() const;
	Item * getTerminalItem() const;
	LineSegment *getSegment();
	void drawTo(double, double );
	void moveEnd(Item *);
	void addCorner(double x, double y);
	Corner * isCorner( QPointF );
	void selected( bool);
	void contextMenuEvent(QGraphicsSceneContextMenuEvent *event);

	bool hasMergedLines();           // returns true if another
	                    // line terminates on a corner of this line
	std::string getExitCode();
	void writePoints(ScriptXml *);

	Line( const Line &) = delete;
	void operator = (const Line &) = delete;
};
```

6.11 Page Class

This is a part of the CanvasImpl class that represents one page of the tabbed widget. It holds the function item and the other items that make up the function and the lines connecting the items. There is a mutual friend relationship to CanvasImpl as it really is just a part of that object.

```
class Page
{
	friend class CanvasImpl;
	friend class ScriptXml;
private:
	CanvasImpl *canvas;                   // reference
	QTabWidget *tabWidget;                // reference
	QWidget *frame;                       // owned
	std::string name;
	QGraphicsView *view;                  // owned
	VICI::CanvasScene * scene;            // owned
	FuncItem * function;
	std::list< Item * > items;            // owned
	std::list< Line * > lines;            // owned
	std::list< Comment * > comments;      // owned
public:
	Page( CanvasImpl *, QTabWidget *, bool withFunction = true);
	Page( const Page &) = delete;
	~Page();
	void operator = ( const Page & ) = delete;

};
```

6.12 CanvasImpl Class

This is the implementation of the Canvas facade for the libcanvas library. It manages the user interactions with the flow chart drawing areas.

```
class CanvasImpl : public QObject, public Canvas
{
	Q_OBJECT
	friend class Page;
	friend class ScriptXml;
public:
	enum EditMode { EmSelect, EmSymbol,
		EmLineStart, EmLine, EmText };
private:
	VICI::GWindow *window;			// reference
	Symbol::SymbolMgr * symbolMgr;		// reference
	Interp::Interpreter *interpreter;	// reference

	QTabWidget *tabWidget;		// reference - owned by Qt
	QToolBar *canvasToolBar;
	QAction *newAction, *funcAct, *arrowAct, *beamAct, *handAct;

	std::map< std::string, Page * >  pages;
	VICI::Symbol::Symbol *currentSymbol;	// reference
	Line *currentLine;			// owned
	Page * currentPage;

	void editModeChange( EditMode );
	StateVariable<EditMode, CanvasImpl,
		      &CanvasImpl::editModeChange> editMode;
	void cleanEmptyComments();

	void createToolBar();

public:
	CanvasImpl(Window *, Symbol::SymbolMgr *,
		Interp::Interpreter *, CanvasClient *);

	// menu actions
	virtual void load( csr filename );
	virtual void save( csr filename );

	// display of executing script
	virtual void setExecution( bool active, csr node );

	// assigning a command to the current symbol
	virtual void setCommand( csr command );

	// symbol interface
	virtual void selection(Symbol::Symbol*);
	virtual void symbolAttr(Symbol::SymbolAttributes&);
	virtual void textAttr(Symbol::TextAttributes&);

	// interpreter client
	virtual void setValue( csr varName, csr value );
	virtual void setFile( int state, csr filename );
	virtual void setCursor( ThreadId, NodeId );
	virtual void breakReached( ThreadId, NodeId );
	virtual void dataReady( NodeId );
	virtual void done();

	void mousePressEvent( QGraphicsSceneMouseEvent * mouseEvent );
	void mouseMoveEvent( QGraphicsSceneMouseEvent * mouseEvent );
	void itemSelected( Item * );
```

```
      void itemOpened( Item *);
      void cornerSelected( Corner *);
      void deleteLine( Line *);
      void deleteItem( Item *);
      QPointF mapToView(QPointF);
      QWidget *viewWidget() { return currentPage->view; }
      void setPageName(std::string);
      void deletePage(FuncItem *);

public slots:
      void newChart();
      void newFunc();
      void pageChange(int);
      void selectMode();
      void dragMode();
      void textMode();

};
```

6.13 CanvasFactoryImpl Class

This is responsible for creating an instance of the CanvasImpl object.

```
class CanvasFactoryImpl : public CanvasFactory
{
public:
      CanvasFactoryImpl(){}
      Canvas * makeCanvas( Window *, Symbol::SymbolMgr *,
                   Interp::Interpreter *, CanvasClient * );
};
```

6.14 ScriptXml Class

This is responsible for managing the process of saving and reloading a diagram to an XML file.

```
class ScriptXml : public VICI::Xml
{
private:
      CanvasImpl *canvas;
      xmlNodePtr shapeNode;      // used by write*Item functions.
      xmlNodePtr lineNode;       // used by writePoint function

      static std::map< std::string, Symbol::Style > styleMap;
      Item *getItem( Page *, csr itemRef );

      void addMenus();
      void addFunctions();
      void addShape( Item *, xmlNodePtr parent );
      void addLine( Line *, xmlNodePtr parent );
      void addComment( Comment *, xmlNodePtr parent );
      void loadFunction( xmlNodePtr );
      void loadShape( Page *, xmlNodePtr );
      void loadLine( Page *, xmlNodePtr );
      void loadComment( Page *, xmlNodePtr );

public:
      ScriptXml(CanvasImpl *c);
      void saveScript( csr filename );
```

```
	void loadScript( csr filename );

	void writeCommandItem( bool bg, csr name, ArgList args );
	void writeChoiceItem( bool bg, csr name, ArgList args );
	void writeFuncRefItem( bool bg, csr name, ArgList args );
	void writeFuncItem( csr name, csr menu );
	void writeVarItem( csr name );
	void writeConstItem( csr name, csr value);
	void writeMutexItem( csr name );
	void writeSemItem( csr name );
	void writeFileItem( csr path, bool temp, bool append );
	void writeInlineItem( csr data );
	void writePipeItem( csr path, bool temp );
	void writeDisplayItem( );
	void writeLockItem( csr name );
	void writeUnlockItem( csr name );
	void writePostItem( csr name );
	void writeWaitItem( csr name, int );

	void writePoint( double x, double y );
};
```

6.15 Item Classes

Each symbol type has a corresponding class that derives from the Item class. Each one has its own data items and a corresponding dialog class so that the user can set the attributes. Each has a method that is used to write its content to the XML file when saving, and a constructor that is used to reconstruct the item from the XML.

Each one has a method that is used to tell if a line is able to connect to the item in order to prevent the worst of the possible diagram syntax errors. (More is needed in this regard.)

The following is for the CommandItem and is typical for these classes.

```
class CommandItem : public Item
{
	Q_OBJECT
protected:
	std::string name;
	VICI::ArgList args;
	bool bg;
	QAction *searchAct, *cmndAct;
	CommandDialog *dialog;
	virtual void makeMenu();  // special case for command
	virtual bool checkConnection(VICI::Symbol::Style,
		bool atStart, QString & reason );

protected slots:
	void searchAction();
	void propAction();
	void cmndAction();

public:
	CommandItem( CanvasImpl *tc);
	CommandItem( int id, CanvasImpl *tc, csr name,
		ArgList args, bool bg );
	virtual bool isCommand() const;
	virtual void writeXml( ScriptXml * );
};
```

6.16 Item Dialog Classes

Each item type has a corresponding dialog box so that the user can enter the properties for the object. The following is for the Command Item and is typical for these classes.

```
class CommandDialog : public ItemDialog
{
	Q_OBJECT
private:
	QLineEdit *name;
	QListWidget *params;
	QCheckBox *bg;
private slots:
	void addBlank(QListWidgetItem *);
	void changed(QListWidgetItem *, QListWidgetItem *);
public:
	CommandDialog(QWidget *);
	void getVals(std::string &, VICI::ArgList &, bool &);
	void setVals(const std::string &, const VICI::ArgList &,
		const bool &);
};
```

Appendix A