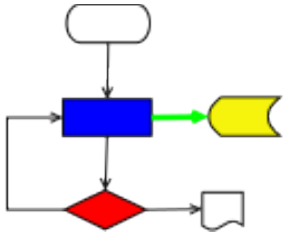
VICI



VISUAL CHART INTERPRETER

Design for libsyntax

Publication History

Date	Who	What Changes
4 October 2012	Brenton Ross	Initial version.
15 June 2014	Brenton Ross	Updated for Vici

Table of Contents

1 Introduction

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

1.1 Scope

This document covers the design for libsyntax.

This design is for increment #3.

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

The libsyntax library is responsible for displaying the syntax chart that describes the options and parameters for a command. (T1.9)

It will be given the EBNF for the command's options and use the libebnf library to create a parse tree which it will then display.

The diagrams might be quite large so the user should be able to pan and zoom over a diagram to better view it on displays of varying size.

The initial version will show a single production at a time and allow the user to expand a non-terminal symbol into its chart. A “Back” button will allow return to the previous diagrams. Subsequent versions may show all productions on a single chart and allow the user to export the chart as an SVG diagram.

2.2 Design Approach

The Qt library provides a scene graph class which will form the canvas for the diagrams. This class is suited to the task as it is capable of displaying large numbers of small objects efficiently, as well as passing mouse events to the drawn objects.

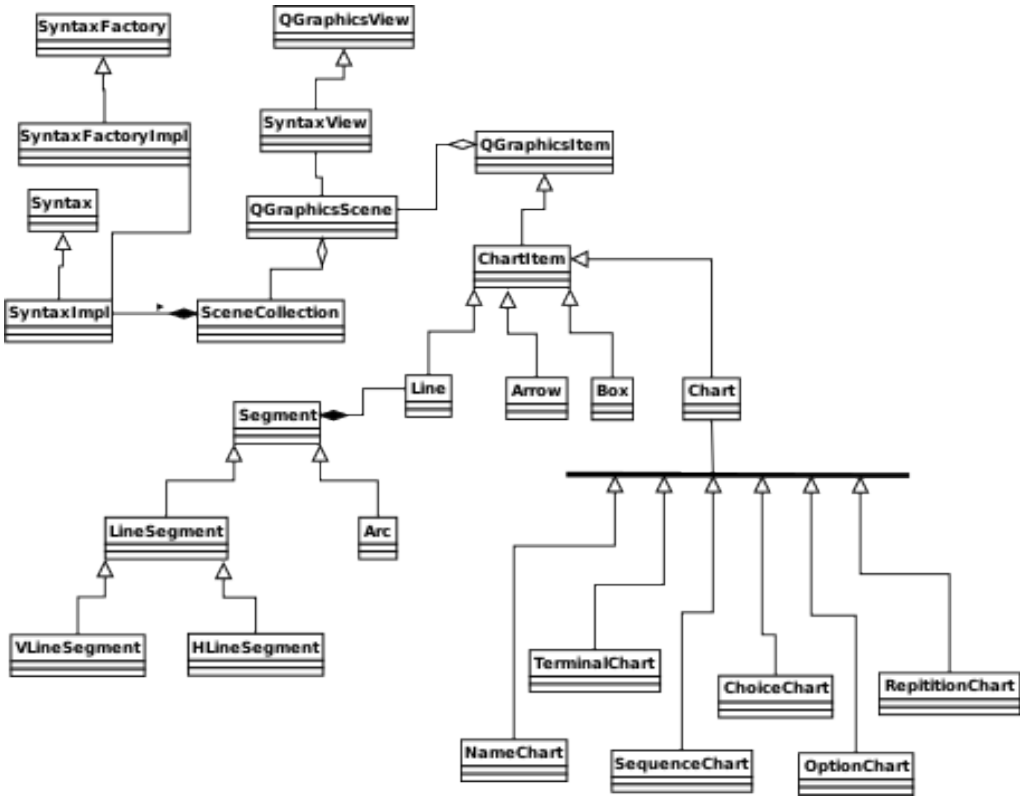
The diagrams will be made up from horizontal and vertical lines, 90 degree arcs and rounded rectangles containing the text. These components will then be combined into chart objects for sequence, repetition, option and choice in a recursive manner that is built up according to the parse tree supplied by the EBNF library.

Each production will be created on its own scene graph and made visible in the view when the user selects it.

The non-terminal boxes will be given the ability to accept a mouse click which will display the diagram for the corresponding production.

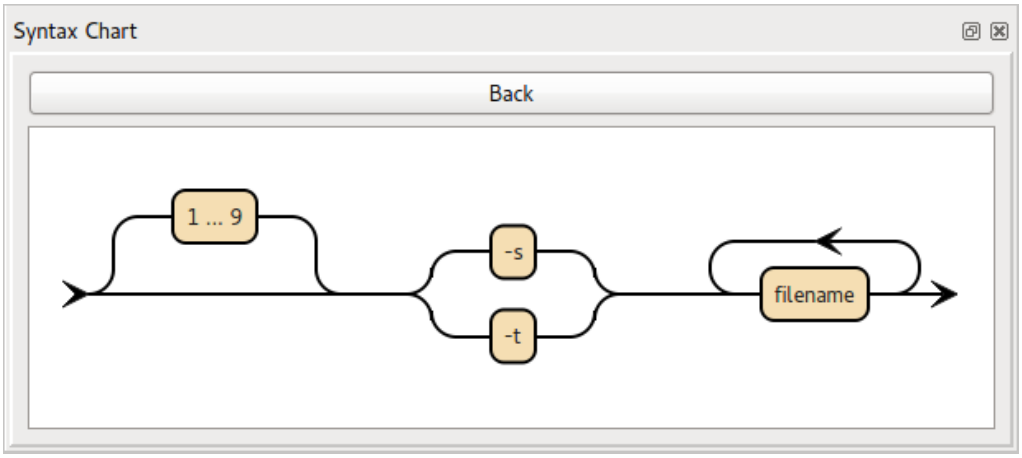
3 Application Design

This library is responsible for displaying a syntax chart.



4 User Interface

The component will display in a window provided by the enclosing application.

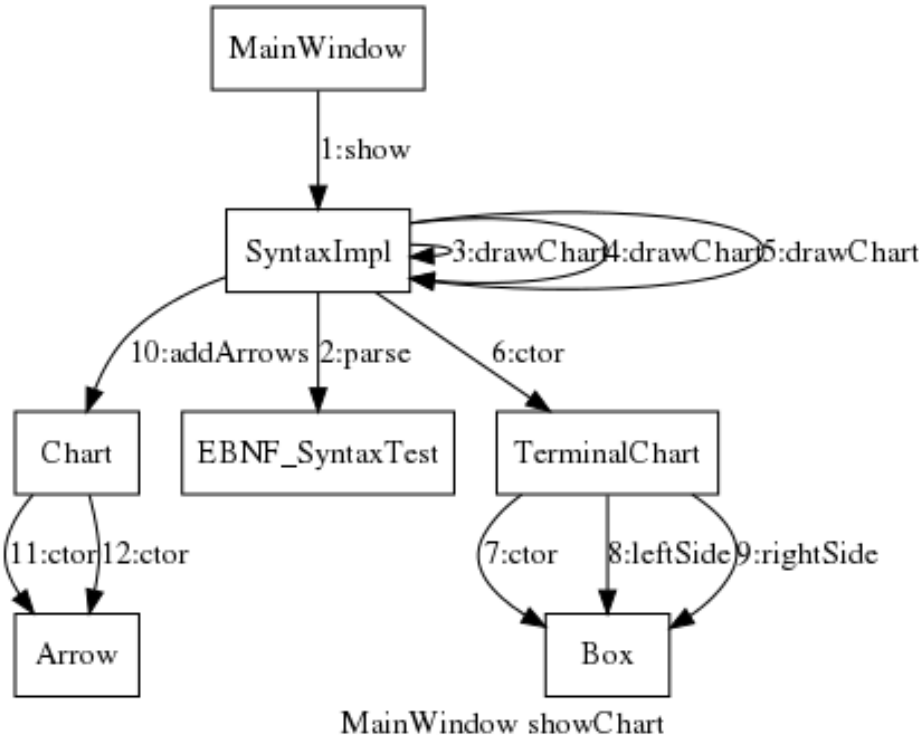


5 Collaboration Diagrams

This section documents the interactions between the classes for each applicable use case.

The edges in the diagrams are numbered according to the sequence of events.

5.1 Show



MainWindow showChart

6 Class Descriptions

6.1 SyntaxFactory Class

This is an abstract class that defines the interface used to create objects of the Syntax class.

```
class SyntaxFactory
{
public:
	virtual ~SyntaxFactory() {}
	virtual Syntax * makeSyntax( Window *, EBNF::EBNF * ) = 0;
};
```

6.2 SyntaxFactoryImpl Class

This is the concrete implementation of the SyntaxFactory which is responsible for creating concrete instances of the Syntax class.

```
class SyntaxFactoryImpl : public SyntaxFactory
{
public:
	virtual Syntax * makeSyntax( Window *, EBNF::EBNF * );
};
```

6.3 Syntax Class

This is an abstract class which defines the interface for libsyntax library.

```
class Syntax
{
public:
	virtual ~Syntax() {}

	// display the syntax chart for the supplied ebnf
	virtual void show( csr s ) = 0;

};
```

6.4 SyntaxImpl Class

This is the concrete implementation of the Syntax class which provides the facade interface for the libsyntax library.

```
class SyntaxImpl : public QObject, public Syntax
{
protected:
	GWindow *window;
	EBNF::EBNF *ebnf;
	QGraphicsView *view;
	SceneCollection scenes;
	std::stack< QGraphicsScene * > history;
	QGraphicsScene *firstScene;

	std::string drawChart( EbnfNode *, QGraphicsScene * );
	Chart * drawChart( EbnfNode * );

	QPushButton *backBtn;

protected slots:
	void selectedChart( std::string name );
	void back();
public:
	SyntaxImpl( Window *, EBNF::EBNF * );
	virtual void show( csr s );
};
```

6.5 SceneCollection Class

This is a collection of QGraphicsScene objects in a map. It is used to display the separate charts by name.

```
class SceneCollection : public std::map< std::string, QGraphicsScene * >
{
public:
	SceneCollection(){}
	~SceneCollection();
};
```

6.6 ChartItem Class

This is a specialisation of the QGraphicsItem containing attributes which are common across all drawn objects.

```
class ChartItem : public QGraphicsItem
{
protected:
	QPen pen;    // for lines and edges
	QBrush brush;// for filling boxes
public:
	ChartItem();
	virtual ~ChartItem();

};
```

6.7 Line Class

This class manages the drawing of lines on the diagram. A line is constructed of a set of segments which can be either horizontal or vertical straight lines or 90 degree arcs.

```
class Line : public ChartItem
{
	std::vector< Segment * > segments;
public:
	Line();
	void addSegment( Segment * );

	QPointF sceneStart() const;
	QPointF sceneEnd() const;

	void paint ( QPainter * painter,
		const QStyleOptionGraphicsItem * option, QWidget * widget
= 0 );
	QRectF boundingRect() const;
};
```

6.8 Segment Class

This is a base class that allows the Line objects to accumulate a collection of different types of line segments.

```
class Segment
{
public:
	Segment( int a1, int b1, int a2, int b2 )
		: x1(a1), y1(b1), x2(a2), y2(b2)
		{}
	virtual ~Segment(){}
	virtual int top() const = 0;
	virtual int bottom() const = 0;
	virtual int left() const = 0;
	virtual int right() const = 0;
	virtual void paint ( QPainter * painter,
		const QStyleOptionGraphicsItem * option, QWidget * widget = 0 ) = 0;

	int x1, y1, x2, y2;

};
```

6.9 LineSegment Class

This contains the common attributes of straight lines.

```
class LineSegment : public Segment
{
public:
	LineSegment( int x1, int y1, int x2, int y2 )
		: Segment( x1, y1, x2, y2 )
		{}

	virtual int top() const;
	virtual int bottom() const;
	virtual int left() const;
	virtual int right() const;

	virtual void paint ( QPainter * painter,
		const QStyleOptionGraphicsItem * option, QWidget * widget = 0 );
};
```

6.10 VLineSegment and HLineSegment Classes

These contain the details for vertical and horizontal line segments.

```
class HLineSegment : public LineSegment
{
public:
	HLineSegment( int x, int y, int len )
		: LineSegment( x, y, x+len, y )
		{}
};

class VLineSegment : public LineSegment
{
public:
	VLineSegment( int x, int y, int len )
		: LineSegment( x, y, x, y+len )
		{}
};
```

6.11 Arc Class

This holds the details for drawing 90 degree arcs used between horizontal and vertical line segments.

```
class Arc : public Segment
{
public:
	enum Dir{ Left, Right, Up, Down };
	Arc( int x, int y, Dir s, Dir e );

	Dir start, end;

	virtual int top() const;
	virtual int bottom() const;
	virtual int left() const;
	virtual int right() const;

	virtual void paint ( QPainter * painter,
		const QStyleOptionGraphicsItem * option, QWidget * widget = 0 );
};
```

6.12 Arrow Class

Arrows are added to lines to show the direction of the connection, and each chart has arrows at it start and end.

```
class Arrow : public ChartItem
{
public:
	enum Dir{ Left, Right, Up, Down };
	Arrow( Dir );

	void paint ( QPainter * painter,
		const QStyleOptionGraphicsItem * option, QWidget * widget = 0 );
	QRectF boundingRect() const;
};
```

6.13 Box Class

This creates a box around some text using a rounded rectangle.

```
class Box : public ChartItem
{
public:
	Box( csr text );
	void setLink( QGraphicsScene * );
	QPointF leftSide() const;
	QPointF rightSide() const;
	void paint ( QPainter * painter,
		const QStyleOptionGraphicsItem * option, QWidget * widget = 0 );
	QRectF boundingRect() const;
};
```

6.14 Chart Class

This is a base class for the different types of charts. This forms a recursive structure with charts draw within charts.

```
class Chart : public ChartItem
{
protected:
	QPointF leftPoint, rightPoint;
public:
	Chart();
	virtual ~Chart();

	QPointF leftSide() const { return mapToParent( leftPoint ); }
	QPointF rightSide() const { return mapToParent( rightPoint ); }
	void addArrows();

	void paint ( QPainter * painter,
		const QStyleOptionGraphicsItem * option, QWidget * widget = 0 );
	QRectF boundingRect() const;
};
```

6.15 NameChart Class

This is displayed as a simple box containing text. The names are the indexes used in the SceneCollection. The objects respond to mouse clicks by sending a signal that causes the corresponding named chart to be displayed.

```
class NameChart : public QObject, public Chart
{
public:
	NameChart( csr name );
protected:
	void mousePressEvent ( QGraphicsSceneMouseEvent * event );
	std::string name;
signals:
	void selected( std::string name );
};
```

6.16 TerminalChart Class

This is displayed as a simple box containing text.

```
class TerminalChart : public Chart
{
public:
	TerminalChart( csr text );
};
```

6.17 SequenceChart Class

This chart takes two charts and connects them with a horizontal line.

```
class SequenceChart : public Chart
{
public:
	SequenceChart( Chart *left, Chart *right );
};
```

6.18 ChoiceChart Class

This chart takes two charts and connects them such that either one is on the path taken.

```
class ChoiceChart : public Chart
{
public:
	ChoiceChart( Chart *left, Chart *right );
};
```

6.19 OptionChart Class

This chart takes a chart and creates a path that bypasses it.

```
class OptionChart : public Chart
{
public:
	OptionChart( Chart *);
};
```

6.20 RepititionChart Class

This chart takes a chart and creates a path from its output back to its input.

```
class RepititionChart : public Chart
{
public:
      RepititionChart( Chart * );
};
```

Appendix A