INDIAN LANGUAGE SUPPORT IN GNOME-TERMINAL

A Project Report

Submitted by

Kulkarni Swapnil	**110708035**
Kulkarni Mihir	**110708033**
Dige Sourabh	**110708020**

in partial fulfilment for the award of the degree

of

B.Tech Information Technology

Under the guidance of

Prof. Abhijit A.M.

College of Engineering, Pune

DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATION TECHNOLOGY, COLLEGE OF ENGINEERING, PUNE-5

May, 2011

Acknowledgements

We would sincerely like to acknowledge the following people who played a very important role in the success of the project. They guided us from time to time and helped us whenever we were stuck at any point. We deeply appreciate their commitment and dedication towards the spread of Open Source and the Free Software Movement in general.

Prof. Abhijit A.M. - Our guide and mentor throughout the project. We thank him for giving us the opportunity to work with him and in particular on this project and for spending invaluable time in critical moments of our project.

Praveen Arimbrathodiyil, Pravin Satpute, Santhosh Thottingal - For providing timely help and guidance.

We would also like to thank Dr. Jibi Abraham [HOD Computer Engg. and Information Technology, COEP] and all the staff of Computer Engg. and Information Technology Department. Last but not the least, we would like to thank the whole **FOSS community** and the people involved in Indic Langauge Computing throughtout the world without whom this project would not have been a reality.

DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATION TECHNOLOGY,

COLLEGE OF ENGINEERING, PUNE

CERTIFICATE

Certified that this project, titled "INDIAN LANGUAGE SUPPORT IN GNOME-TERMINAL" has been successfully completed by

Kulkarni Swapnil	**110708035**
Kulkarni Mihir	**110708033**
Dige Sourabh	**110708020**

and is approved for the partial fulfilment of the requirements for the degree of "B.Tech. Information Technology".

SIGNATURE	SIGNATURE
Abhijit A.M.	**Dr.Jibi Abraham**
Project Guide	**Head**
Department of Computer Engineering	**Department of Computer Engineering**
and Information Technology,	**and Information Technology,**
College of Engineering Pune,	**College of Engineering Pune,**
Shivajinagar, Pune - 5.	**Shivajinagar, Pune - 5.**

Abstract

There are a wide range of users in the computer world. Computer users may or may not be well versed in English language. So there is need to make available software which have their GUI and/or instructions in local languages for people to make use of them. This will not only make it easy for the masses to use software but will also increase computer literacy in the country. Building new software with interfaces in various local languages is one of ways to achieve this. But, to understand and modify existing free software available under GNU GPLv3 and similar licenses and port them into local languages is perhaps the best and the easiest way. The main advantage of this approach is that we already have very good quality software available for English which has been tested and changed to make it the perfect one. So instead of developing a new software right from scratch it is easier to modify this available software to provide Indic language support.

With this approach in mind, we have selected the most widely used applications in the GNU/Linux environment which is the *GNOME-terminal* and *vim-editor*. There is very little support for local languages in the *GNOME-terminal*. Languages like the Devanagari script (Hindi and Marathi) have almost no support. Our aim is to make the *GNOME-terminal* available for people who know the Devanagari script.

The successful implementation of Devanagari script rendering in *GNOME-terminal* will help the developers from other language communities to implement support for their language also.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

Computer is becoming the integral part of the human life. Almost all the industrial processes are using the computers in direct or indirect manner. Computers are used in our education system, government offices, etc. However most of the computing systems are developed with English as default language. Though many European and some Asian languages are increasingly supported, the support for Indic languages remains quite low. Common Indian people need local language support for becoming computer literate.

Any user- novice or expert, who uses GNU/Linux with GNOME desktop environment knows the importance of *vim-editor* and *GNOME-terminal.* These are the most common applications used in GNU/Linux. These applications are having complex script rendering problem since they were written. There is a demand from all over the world for complex script support in these applications.

In the recent years there has been lot of development going on for making of GNU/ Linux software available in local languages. For languages like Malayalam there is much work done and going on too. There is a large need for making similar support available for Marathi/Hindi also.

Indic Languages is a term which is commonly referred to languages that are spoken in India. These include languages like Hindi, Marathi, Gujarati, Malayalam, Kannada, Tamil, etc. Indic Language Computing thus refers to all the work which is being done worldwide to develop new software and to increase support for Indic Languages on existing software.

Moving ahead with this, Devanagari Script [14] is the script which is used by Devanagari languages like Hindi, Marathi and Sanskrit. Devanagari Script is also referred to as a complex script [15] because of the nature of its writing. Unlike English where there in no change in the characters during formation of words, in complex scripts there are changes that have to be done to the previous character itself to produce change of sound/syllable. Some examples of complex scripts are languages like Hindi, Marathi, etc. A problem that is being faced is that of rendering of this Devanagari Script on the *GNOME-terminal*. We have taken this problem as our task.

Improving and providing Devanagari Script support on *GNOME-terminal* will make it very useful for people who are familiar with the Devanagari Script but are not comfortable with English. Also, we are hopeful that our efforts will help other developers who are trying to develop support in the *GNOME-terminal* for their own local languages.

1.2 Objective

The main objective of our project was to develop support for Devanagari Script in the *GNOME-terminal*. This involved improving the rendering of Devanagari Script which at present is not proper.

Chapter 2

Literature Study

2.1 Internationalization and Localization

Internationalization and localization [5] is the adaption of computer software to different languages. Localization is not to to make any engineering changes to the software but to adapt it to local languages by translating the text or making the appropriate changes to make that software to be understood in local languages.

At present a lot of work is being done for translations in Marathi and Hindi as well as other Indic languages. We can render complex scripts like Marathi on commonly used GNU/Linux applications like *Gedit*, *Open Office*, etc. The rendering is perfect and it needs no improvement on these software. *GTK (GIMP Tool Kit)* supports complex scripts by default. So the applications that use *GTK* for development have complex script support enabled.

2.2 Unicode Standards for Uniformity

Important aspect of the work is that all software which are available in the market do not use a common coding scheme like Unicode [17]. The Unicode Standard [8] , the latest version of Unicode consists of a repertoire of more than 109,000 characters covering

93 scripts, a set of code charts for visual reference, an encoding methodology and set of standard character encodings, and many more such utilities. UTF-8 [4] which is a commonly used encoding uses one byte for any ASCII character provided they have the same code values in both UTF-8 and ASCII but up to four bytes for other characters.

If all the software make use of Unicode based data, uniformity will be achieved. This is because Unicode can support a very large number of characters (almost all which are used in this world) which is not the case with the use of ASCII (American Standard Code for Information Interchange) [1] .

2.3 Rendering the Text on Computer

When the key of the keyboard is pressed character gets displayed onto the screen. It is not as simple as it looks. There are number of internal processing steps involved to do this. When the key is pressed or released, interrupt is raised and kernel records the key that is pressed. After this keyboard controller comes into picture. Keyboard controller is a computer hardware that interfaces the keyboard and computer. Signal is sent to the kernel for handling.

Each key is associated with its particular key value. This key value is then associated with the number which might be ASCII or Unicode, that is assigned as per the keyboard layout or input method. Font technology maps this Unicode value with appropriate font character image. There are various libraries like *GDK*, *cairo* that are used for defining the surface or defining the area on which that character image is to be rendered. And thus, character gets displayed on to the screen.

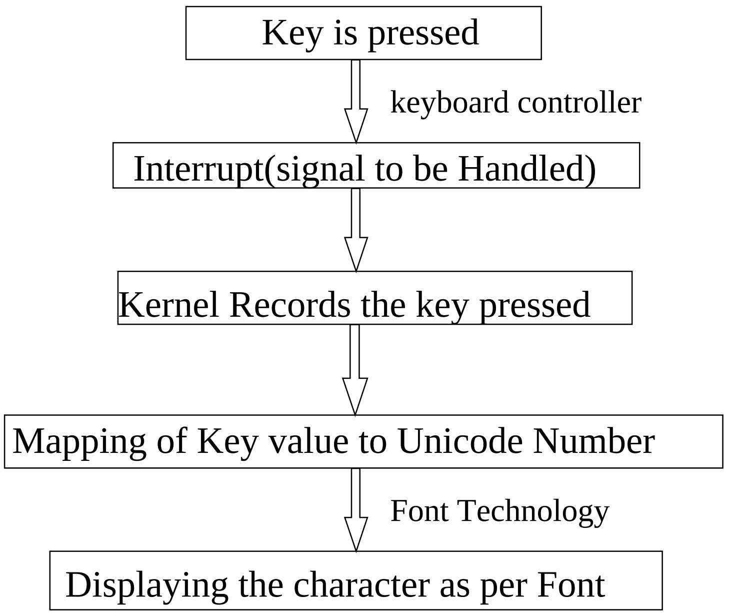

Key is pressed
keyboard controller
Interrupt(signal to be Handled)
Kernel Records the key pressed
Mapping of Key value to Unicode Number
Font Technology
Displaying the character as per Font

Figure 2.1: Flow for rendering of Character in VTE

2.4 Complex Scripts

Indian languages are categorized as complex script language. Complex text requires complex transformations between text input and text display for proper rendering on screen. Rendering of the characters is not straight forward as rendering of Latin characters. Complex text has some characteristics:

1. Bi-directional Text: Characters can be written in both right-to-left or left-to-right direction.

Example: Arabic language is written from left to right.

2. Context sensitive shaping: Rendering of the characters depends on the location and/or surrounding characters.

Example:

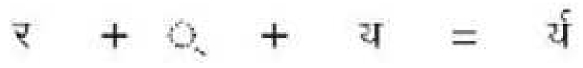


Figure 2.2: Example of context sensitive shaping

3. **Ordering:** Displayed order of characters is not same as the logical order.

Example:

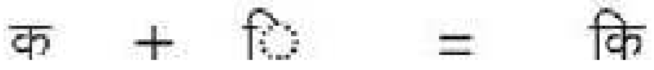


Figure 2.3: Example of ordering

2.4.1 Example

For Latin alphabets there are no complex transformations involved between text input and text display. These characters are displayed straight forward. But in case of Devanagari, there are conjuncts, *virama(halant)*, conjuncts which needs the complex transformations.

Suppose text input is:

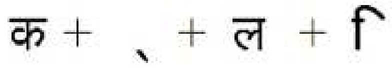


Figure 2.4: Text Input

This text input needs to be processed before displaying. There are complex transformations involved for text display as per language rules. This Text input must be displayed as:

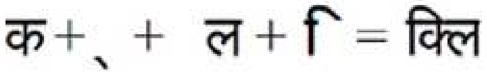


Figure 2.5: Perfect Text Output

2.5 Rendering Engine

Complex script rendering requires the complex transformations between text input and text display. Rendering Engine takes care of all these transformations, and makes the complex script rendering perfect. There are different rendering engines for different desktop environments. GNOME desktop environment has “pango” [7] [10] as rendering engine. K-Desktop environment [6] has “Qt” [16] rendering engine. The implementation we are doing in our project is for the GNOME desktop environment and pango rendering engine.

2.6 Cairo Library

Cairo library[3] provides device independent, vector-graphics based API. It also provides the primitives for two dimensional drawing across different back-ends. Pango and Cairo [9] together are used for complex script rendering in various applications.

2.7 Other Indic Language Computing Project

Silpa (Swathanthra Indian Language Computing Project) [12] is a major project that is being done to host the free software language processing applications easily. It can be used as a python library or as a web service from other applications. It is a platform for porting existing and upcoming language processing applications to the web.

Chapter 3

Problem Definition

3.1 Programming Environment Requirements

Operating System: GNU/Linux

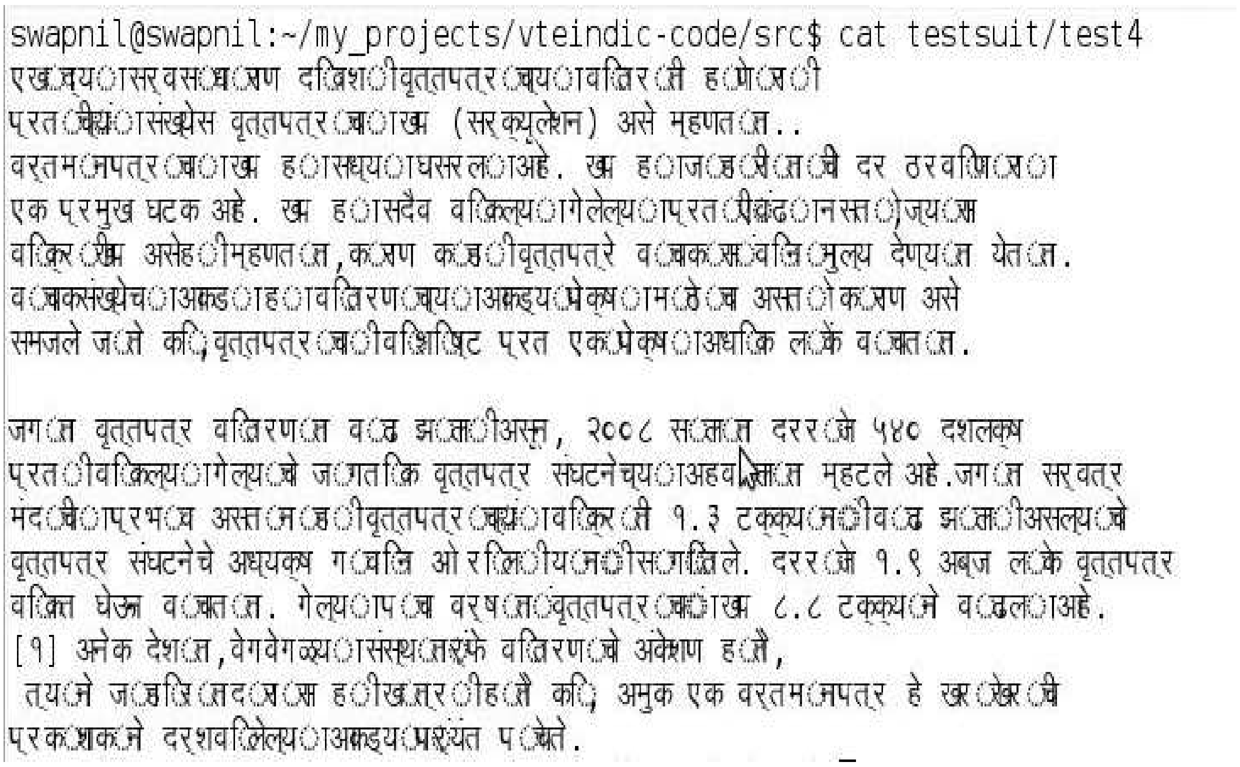
Programming Languages: C, *GTK*-programming, etc.

Tools: *gcc*(compiler), *gdb*(debugger), *libtool*(for creating portable compiled libraries)

3.2 Complex script (Devanagari) rendering problems in *VTE*

In Devanagari the diacritics are of zero-width as well as of non-zero width. The non-zero width diacritics are not rendered properly. The diacritics are rendered differently from basic character in adjacent area as shown in figure below.

In Devanagari in addition to the diacritics, the characters can be combined to form conjuncts. These conjuncts are also not properly rendered as a combined character instead they are rendered separately as shown in figure below.



```
swapnil@swapnil:~/my_projects/vteindic-code/src$ cat testsuit/test4
एखाद्य◌ासर्वस◌ाधा◌ारण द◌िवशि◌ीवृत्तपत्र◌ाच्य◌ाव◌ितिर◌ीत ह◌ो◌ाण◌ार◌ी
प्रत◌ीच्य◌ासंख्य◌ेस वृत्तपत्र◌ाच◌ाख◌प (सर्क्यूलेशन) असे म्हणत◌ात..
वर्तम◌ानपत्र◌ाच◌ाख◌प ह◌ासध्य◌ाघसरल◌ाअहे. ख◌प ह◌ाज◌ाह◌ीर◌ात◌ीच◌ी दर ठरव◌िण्य◌ास◌ा
एक प्रमुख घटक अहे. ख◌प ह◌ासदैव व◌िक्रिल्य◌ागेलेल्य◌ाप्रत◌ीच◌ीं◌ा◌ानस◌ात◌ोज्य◌ात
व◌िक्र◌ीच◌ा असेह◌ीम्हणत◌ात,क◌ारण क◌ाह◌ीवृत्तपत्रे व◌ाचक◌ांन◌ाव◌िन◌ामुल्य देण्य◌ात येत◌ात.
व◌ाचकसंख्य◌ेच◌ाअक◌ाड◌ाह◌ाव◌ितिरण◌ाच्य◌ाअक◌ाड्य◌ापेक्ष◌ाम◌ाठ◌ाच◌ा अस◌ात◌ोक◌ारण असे
समजले ज◌ात◌े क◌ि,वृत्तपत्र◌ाच◌ीव◌िश◌िष्ट प्रत एक◌ापेक्ष◌ाअध◌िक ल◌ाके व◌ाचत◌ात.

जग◌ात वृत्तपत्र व◌ितिरण◌ात व◌ाढ झ◌ाल◌ीअस◌ून, २००८ स◌ाल◌ात दरर◌ाज◌े ५४० दशलक्ष
प्रत◌ीव◌िक्रिल्य◌ागेल्य◌ाच◌े ज◌ागत◌िक वृत्तपत्र संघटनेच्य◌ाअहव◌ाल◌ात म्हटले अहे.जग◌ात सर्वत्र
मंद◌ीच◌ाप्रभ◌ाव अस◌ात◌ान◌ाह◌ीवृत्तपत्र◌ाच्य◌ाव◌िक्र◌ीत १.३ टक्क्य◌ान◌ीव◌ाढ झ◌ाल◌ीअसल्य◌ाच◌े
वृत्तपत्र संघटनेचे अध्यक्ष ग◌ाव◌िन ओर◌िल◌ीय◌ान◌ीस◌ाग◌ितिले. दरर◌ाज◌े १.९ अबज ल◌ाके वृत्तपत्र
व◌िकत घेऊन व◌ाचत◌ात. गेल्य◌ापच◌ा वर्ष◌ात◌ंवृत्तपत्र◌ाच◌ाख◌प ८.८ टक्क्य◌ान◌े व◌ाढल◌ाअहे.
[१] अनेक देश◌ात,वेगवेगळ्य◌ासंस्थ◌ातर्फे व◌ितिरण◌ाच◌े अंकेक्षण ह◌ात◌े,
 त्य◌ान◌े ज◌ाह◌ि◌ात◌ादर◌ान◌ास ह◌ीख◌ात्र◌ीह◌ात◌े क◌ि, अमुक एक वर्तम◌ानपत्र हे ख◌ार◌ाेख◌ार◌ीच
प्रक◌ाशक◌ान◌े दर्शव◌िलेल्य◌ाअक◌ाड्य◌ापर्यंत प◌ाच◌ाेत◌े.
```

Figure 3.1: Devanagari script rendering in *GNOME-terminal*

3.3 Problem Identification

As per our objective, providing complex script rendering support in *GNOME-terminal* we started our work with understanding source code of *GNOME-terminal*. We gone through design architecture, internal processing and working of *GNOME-terminal*. We came to know that *GNOME-terminal* uses the *VTE* (Virtual Terminal Emulator) library for processing. *VTE* library provides API for terminal emulation and *GNOME-terminal* is written using this *VTE* library.

Hence, basic problem of no support for complex script rendering lies in *VTE* library itself. So we focused our work on *VTE*. We gone through *VTE* source code. We understood all aspects of *VTE* and started working on it.

3.4 Limitations of VTE library in rendering Devanagari script

After the Identification of problem, we started exploring the problem. *VTE* is written for fast rendering. It uses primitive calls for rendering completely bypassing the pango rendering engine. After going through bug reports [13] [2] and explanation by upstream authors, we got idea of the problem in more precise way.

In *Gedit* -a popular text editor on GNOME desktop environment, the whole text is rendered as a single unit. So, the rendering engine is able interact between adjacent character cells, including reordering and composing conjuncts so as to produce exact characters as required by Indic and other complex scripts.

Terminals, on the other hand, are display grid, where individual characters are put in grid. And the grid cells are independent of one another. That's how it works from the beginning. This is fine for Latin and CJK- Chinese Japanese Korean, and probably for Thai-Lao with typewriter convention applied. But it needs a tremendous change to support complex text like Indic and Arabic, where adjacent display cells must interact with one another. So, such deep structural change is not an easy task. It even deserves a redesign, that's why this problem is unsolved by anyone else.

3.5 Approaches

We followed the different approaches to solve the problem such as:

- Replacing the primitive rendering call with high level pango call.

  Pango helps in layout and rendering of internationalized text. So we thought that using pango rendering calls would help in dealing with rendering of complex scripts. However pango just helps in laying out the glyphs properly and not ordering the glyphs and forming the combined characters. It also did not help in adjusting the rendering widths of the characters due to the cell structure of *VTE*.

- Doing the inter cell interaction with allocating widths of the characters dynamically and variably.

  We have implemented this approach to handle the rendering of Devanagari script.In this approach we perform inter-cell interaction where in we take into previous characters while forming combined characters or using non-zero-width diacritics. Due to variable nature of the script in terms of the width of characters we allocate different rendering widths to different characters.

Chapter 4

Understanding VTE

For us to get to the exact cause of the problem, it is necessary to get a proper understanding of the working of *VTE*. *VTE* implements a logical cell structure of the screen used for rendering. This cell structure is uniform i.e each cell has same dimensions. Each character is rendered in a particular cell. As a result of this the screen is composed of rows and columns of cells containing characters.

There are different paths through which the code progresses.Below, is a detailed explanation of the working of the *VTE*.

4.1 Working of VTE

The processing of the characters starts once it is typed on the window and a *key_pressed_event* is triggered which reads the input and decides what is to be done with it. Simultaneously a process timeout is set. The variable used for this is *VTE_DISPLAY_TIMEOUT*. It is by default set to 10ms. Once the timeout is up the static function *"gboolean process_timeout"* is called. The overall flow keeps coming to this function as and when the timeout is up.

It is from the function *"process_timeout"* that the subsequent calls are made. This

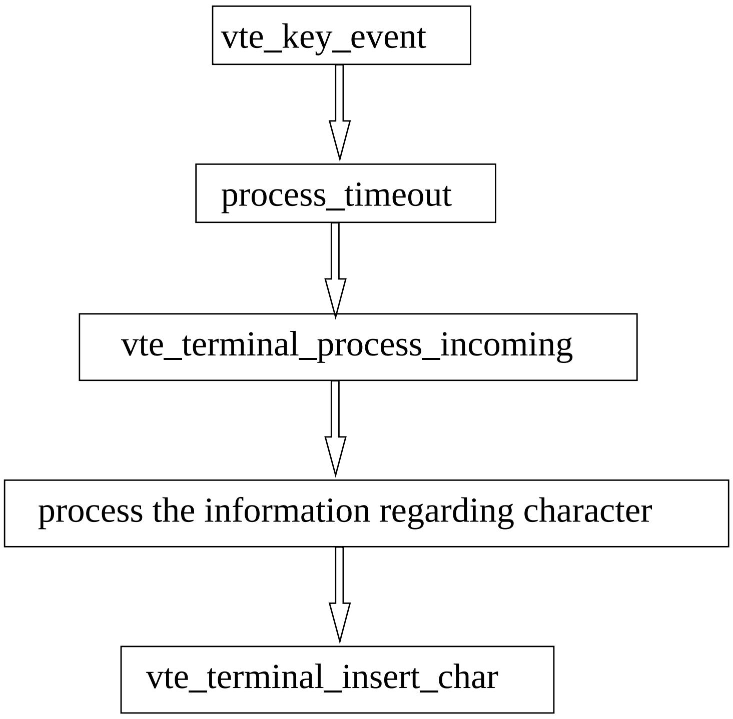

vte_key_event
process_timeout
vte_terminal_process_incoming
process the information regarding character
vte_terminal_insert_char

Figure 4.1: 1st flow for rendering in VTE

includes call to the function *"vte_ terminal_ process_ incoming"*. Most of the processing of the character is done in this function. The detailed explanation of the working of *"vte_ terminal_ process_ incoming"* is as follows:

4.1.1 vte_terminal_process_incoming

This function primarily processes the incoming data. It is a two step process:

1. In the first step the incoming characters are first converted to Unicode characters.

2. In the second step the characters are checked to process the control sequences.

The input to this function is the structure *VteTerminal* which is itself an instance of type defined structure _ *VteTerminal*. This structure has all the metric and sizing data like dimensions of character cells, important character details like ascent and descent and the dimensions of the window.

First and foremost, it saves the current cursor position. Following this it converts the characters into Unicode format. It does this process chunk by chunk converting each chunk of data into Unicode characters.

Then it computes the number of converted characters. The point to notice here is that the control sequences are still not separated. They are still a part of the converted Unicode characters. Then after the conversion is over, it checks for presence of any control sequences. This is done by issuing a call for the function *"_vte_matcher_match"*. The subsequent flow is not related to our work. The result of this leads to the occurrence of one of the three situations related to control sequences which are again not important to our work.

What is important to the rendering process is a call to the function *"_vte_terminal_insert_char"* where the next process is done. Here some pre-processing is done before we make an actual call to the rendering function.

The details of the function *"_vte_terminal_insert_char"* is as follows.

4.1.2 _vte_terminal_insert_char

This function is perhaps the most important function as regards to character processing before actual rendering is done. A lot of parameters are checked before moving ahead. Some of the checking that is done here is to check whether the character is meant to displayed on the status line or whether it is auto-wrapped or calculate how much columns the character will occupy.

The most important function that is carried out here is the handling of zero-width characters. Initially there was no support for zero width characters like the vector sign, etc in *VTE*. But it was added later and the change was made in this function. As far as our project is concerned, we have made changes in this function to solve our problem.

Multiple cell interaction is handled in this function. Now what is multiple cell interaction? In this, the rendering of the next character depends on the previous character. In case we have a zero-width character as the next one, its rendering is done on the preceding cell itself and not on the next cell. This enables us to achieve multi-cell interaction and hence characters like the vector-sign can be rendered perfectly.

Also before the rendering is done, it checks whether we have enough cells to render this data. When a text is actually added a note of this is made in the *text_ inserted_flag* which is made TRUE. After this function the process moves ahead and rendering is done on the window.

A very important point must be noted here. This function doesn't call any other function to render the text. The function call right from "process_timeout=10ms" are done as and when the *VTE_ DISPLAY_ TIMEOUT* expires. So this set of functions is repeated after every 10ms.

For the rendering of the text another flow is simultaneously being executed in parallel to this one.

Now other than the process that is being carried on in the above execution thread, another thread is simultaneously running. The execution path is as shown in the figure below.

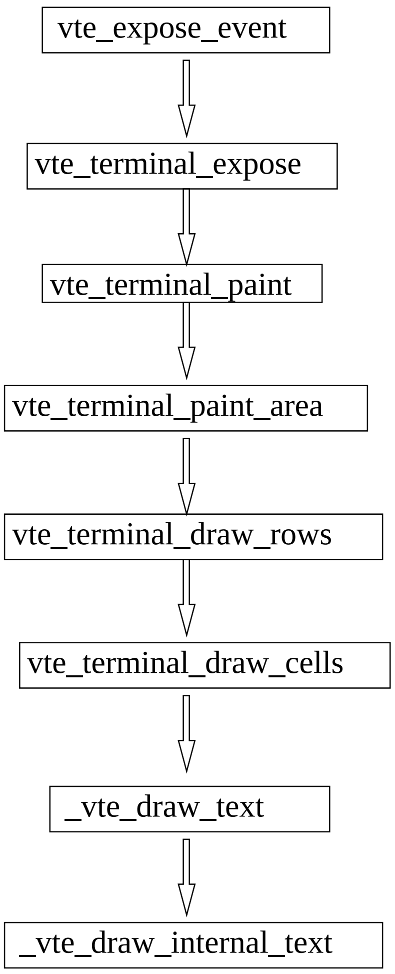


Figure 4.2: 2nd flow for rendering in VTE

4.1.3 vte_expose_event

The event of terminal getting exposed is handled by an *expose_ event* which calls the function *vte_ terminal_ expose*.

4.1.4 vte_terminal_expose

This function takes arguments as the widget for which the event is set and a GdkEventExpose event. In this function region to be painted is determined from the GdkEventExpose event. The region to be updated is appended to a list called *terminal>pvt>update_ regions* and a call to a function *vte_ terminal_ paint* is made.

4.1.5 vte_terminal_paint

In this function it is asserted that the widget has been realized and a window has been created in that widget. A cairo context *_vte_ draw* is set for that window. The region to be painted is calculated in terms of a bounding Gdk rectangle which is then passed on to the function *vte_ terminal_ paint_ area*. A call to *vte_ terminal_ paint_ cursor* is made to paint the cursor.

4.1.6 vte_terminal_paint_area

In this function the Gdk rectangle is converted to appropriate number of rows and columns depending upon the character width and height and also considering the adjustments for the borders of the window.

The start row, column and end row, column calculated are then passed to the function *vte_ terminal_ draw_ rows*.

4.1.7 vte_terminal_draw_rows

In this function the characters from the rows and columns are scanned and to determine the number of columns required and various attributes of the character such as boldness etc.

The characters to be drawn are grouped in an array of data structure *_vte_draw_text_request* which contains the unistr value for the character and the appropriate x and y co-ordintaes of the position where the character is to be rendered. This data structure is then passed on to the function *vte_terminal_draw_cells*.

4.1.8 vte_terminal_draw_cells

In this function the x and y co-ordinates of the characters in the array mentioned above are adjusted for the window borders and then passed on to the function *_vte_draw_text*.

4.1.9 _vte_draw_text

In this function the array of characters to be drawn is passed to the function *_vte_draw_text_internal*. Here the characters of fonts lacking bold property are handled by double striking and then passed on to function *_vte_draw_text_internal*.

4.1.10 _vte_draw_text_internal

This function contains the rendering calls of pango and cairo. Depending upon the font information, one of the three rendering coverage available are used. The three coverages available are:

1. COVERAGE_USE_CAIRO_GLYPH

   In this coverage information about single glyph index and a cairo scaled-font is kept. Glyph index is the index for a particular glyph in the font and cario scaled-font is technique of caching the font metrics i.e information about the rendering of a particular font on a particular screen. This is the fastest way to draw text as it bypasses "pango" completely and allows for stuffing multiple glyphs into a single *cairo_show_glyphs()* request. The *cairo_show_glyphs()* call accepts an array of glyphs to be drawn .The array contain the font-index for the glyph and the x, y co-ordinates of the position where the glyph is to be rendered. This method is used if the glyphs used for the Vteunistr as determined by pango consists of a single regular glyph. Vteunistr is a gunichar-compatible way to store strings. A string consisting of a single unichar c is represented as the same value as c itself. In this case, gunichars can be readily used as Vteunistrs. Longer strings can be built by appending a unichar to an already existing string.

2. COVERAGE_USE_PANGO_GLYPH_STRING

   In this coverage a pango glyphstring and a pango font information is kept. Pango glyphstring is a string of glyphs needed for a particular character. This is slower than the previous case as drawing each glyph goes through pango separately and causes a separate *cairo_show_glyphs()* call, instead of making a single call to render multiple glyphs. In this coverage pango takes care of laying out the glyphs. This method is used when the previous method cannot be used but the glyphs for the character all use a single font.

   Example: A single regular glyph like 'a','b','c',... are rendered using the *COVERAGE_USE_CAIRO_GLYPH* coverage. This coverage is used when there are more glyphs to be rendered for a single Vteunistr like Devanagari character along with a conjunct. So *VTE* converts multiple glyphs into a single glyph string and renders using this coverage.

3. COVERAGE_USE_PANGO_LAYOUT_LINE

   In this coverage, information about a complete pango layout line is kept. The pango layout line stores a complete line from the text. This method is used only in the very exceptional case that a single Vteunistr uses more than one font to be drawn. This happens for example if some diacritics is not available in the font chosen for the base character.

Chapter 5

Solution of the Problem

5.1 Handling of non-zero width diacritics and conjuncts

The *VTE* library was updated to support the rendering of the zero width diacritics . However this did not solve the rendering issue completely for complex scripts like Devanagari, due to the following reasons:

- In Devanagari the diacritics are of zero-width as well as of non-zero width. The non-zero width diacritics are not rendered properly.

  The reason for this is that, in *VTE* the rendering of a character has been logically done in terms of cells i.e each character is rendered in a fixed size cell. As a result of this the non-zero width characters are rendered in different cell than that of the base character. These diacritics are rendered as dotted circles as shown in the figure.

- In Devanagari in addition to the diacritics, the characters can be combined to form conjuncts. These conjuncts are also not properly rendered as a combined character instead they are rendered separately.

  The reason for this problem is that, there is no interaction with the characters of previous cell. In order to form a conjunct we need to consider the previous character so that it can be combined with the current one to form a conjunct.

5.1.1 Solution to the above problems

The above two problems are dealt with referring to the solution of handling zero width characters.

1. The non-zero width diacritics are interacted with previous base character. Due to this interaction a combined character is formed which is rendered as a single glyph.

   This has been implemented in the code with modifications in the files *vte.c* and *iso2022.c*. In the file *iso2022.c* there is function to determine the width for ambiguous width characters. In this file we have an array of non-zero width Devanagari diacritics. This array is used to recognize that the character is a non-zero-width diacritic. There is a function named iscomplex to recognize the non-zero width Devanagari diacritics using the array and return a particular value to the calling function *_vte_terminal_insert_char* in *vte.c*.

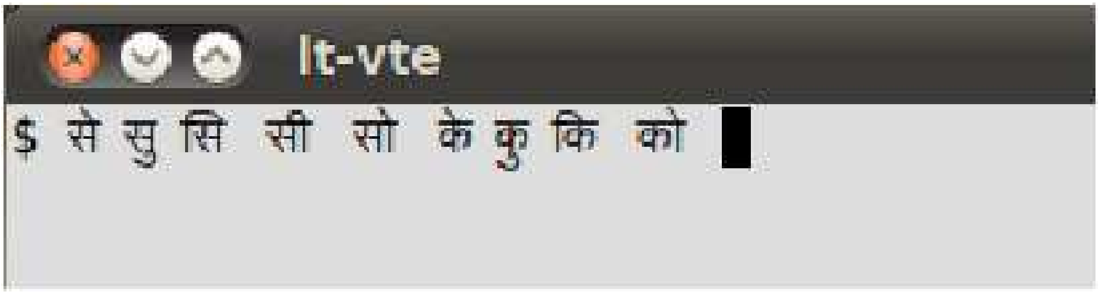
   In the function *_vte_terminal_insert_char* we use the value returned from above iscomplex function to handle a non-zero width diacritic. We append this diacritic to the base character in the previous cell and hence diacritic is rendered along with base character in the same cell.

2. In case of the conjuncts we need to recognize the *halant* character and combine the character after *halant* to the one before *halant* so that we get a single character formed by combining the two characters. This single glyph is then rendered instead of the glyph of first character. Glyphs are available in the font for the combined characters hence they can be rendered as one glyph.

   To implement this, modifications have been made to the the function *_vte_terminal_insert_char* in *vte.c*. in the file *vte.c*. We recognize the *halant* character and then set a flag so as to remember the occurrence of *halant* when we get next input character. When a character is to be rendered with *halant* flag set we combine that character with the previous character and from a conjunct which is rendered as a single character.

5.1.2 Problems after handling non-zero width diacritics and conjuncts

Although the non-zero width diacritics and conjuncts were rendered properly there exist a problem of uneven spacing between the characters.



```
$ से सु सि  सी  सो  के कु कि  को
```

Figure 5.1: Non zero width diacritics rendered with uneven space

In Devanagari the characters are of varying width. So they do not occupy same width when rendered on the terminal. As a result of logical cell structure and varying width, the characters may overlap each other or there may be uneven spaces between characters depending upon the width of characters.

More over the characters formed by handling non-zero width diacritics and conjuncts normally take up more space than the width of the cell, hence they overlap. To avoid this overlap we proposed to use two cells for these characters, however this gave rise to problem of uneven visible space between character cells.

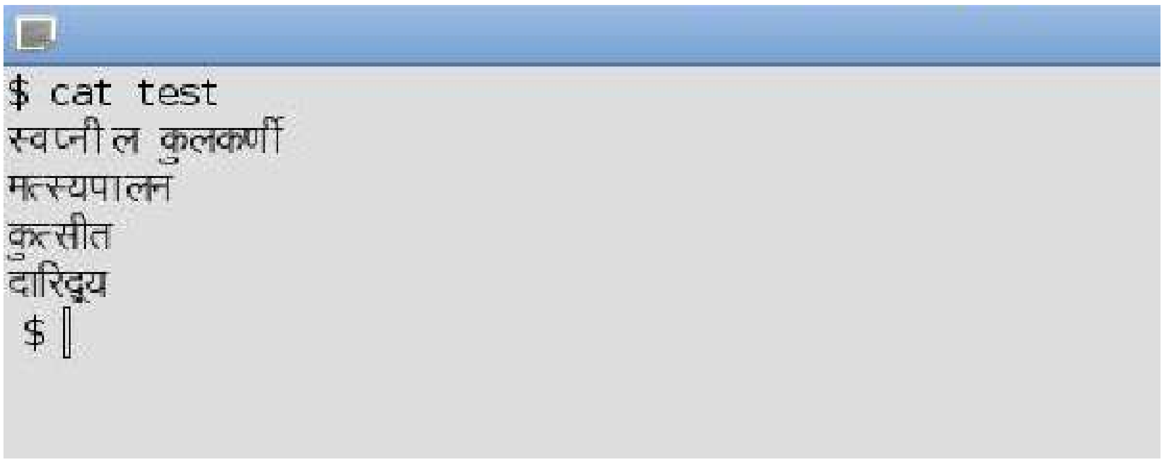
5.2 VTE with Devanagari support

All the problems related to the Devanagari script rendering converges at the same problem in *VTE* architecture that all cells are having fixed width and width can be varied in integral multiple of default char-width set at the initialization of the *VTE* widget.

So we took this architectural problem as a solution to the Devanagari script rendering. We went through the all code paths and set the default character width to 1 instead of character width obtained from font description which was the default earlier. Earlier, when a character was to be rendered, the number of columns required for rendering was 1 and so the width allocated would be (1 * character width). Now, if a character is to be rendered on to the screen, as all Latin alphabets have constant fixed width they are assigned a fixed number of columns where one column occupies one character width. The width allocated for rendering is number of (columns required * character width). The character width in this case is 1.

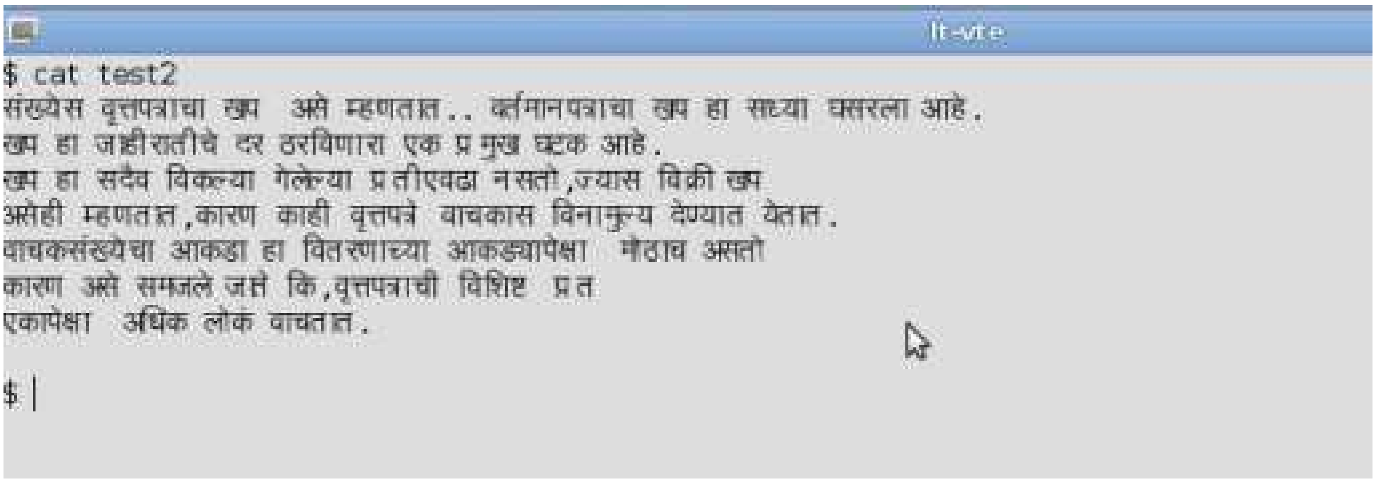
In case of Devanagari script we have created an array containing the widths of the Devanagari characters. Each time when Devanagari character is to be rendered on to the screen width of that character is looked up in the array and number of columns required is set. A column in this case occupies one character width. This method works perfect for non-zero-width diacritics where we add the width required for the diacritic to the width required for base character. However in case of conjuncts we go on calculating the cell width dynamically, as the characters are kept on adding to the base character and put the cursor accurately after the rendered characters. Here for the base character we consider the half width from the array. We have solved all the issues of variable character widths in Devanagari scripts.

This scheme works perfect for Devanagari script rendering. Here are the screen shots:



```
$ cat test
स्वप्नील कुलकर्णी
मत्स्यपालन
कुत्सीत
दारिद्र्य
 $
```

Figure 5.2: Devanagari script rendering



```
$ cat test2
संख्येस वृत्तपत्राचा खप  असे म्हणतात.. वर्तमानपत्राचा खप हा सध्या घसरला आहे.
खप हा जाहीरातीचे दर ठरविणारा एक प्रमुख घटक आहे.
खप हा सदैव विकल्या गेलेल्या प्रतीएवढा नसतो,ज्यास विक्री खप
असेही म्हणतात,कारण काही वृत्तपत्रे वाचकास विनामूल्य देण्यात येतात.
वाचकसंख्येचा आकडा हा वितरणाच्या आकड्यापेक्षा  मोठाच असतो
कारण असे समजले जाते कि,वृत्तपत्राची विशिष्ट  प्रत
एकापेक्षा  अधिक लोक वाचतात.

$
```

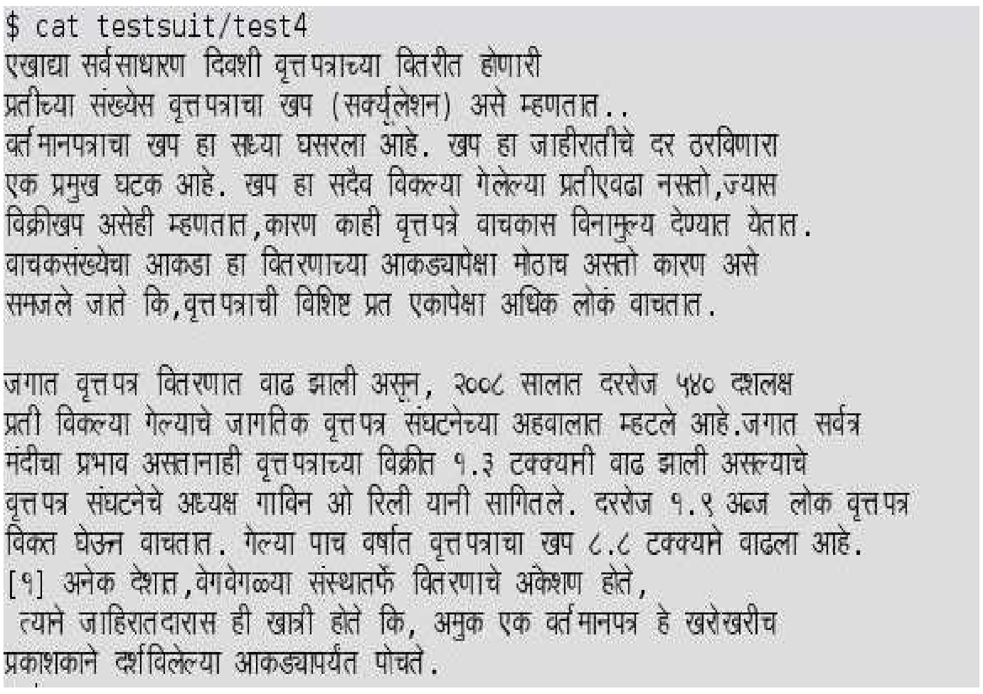
Figure 5.3: Rendering of conjuncts

Chapter 6

Testing Methodology and Test Plans

6.1 Working Module

Output



```
$ cat testsuit/test4
एखाद्या सर्वसाधारण दिवशी वृत्तपत्राच्या वितरीत होणारी
प्रतीच्या संख्येस वृत्तपत्राचा खप (सर्क्युलेशन) असे म्हणतात..
वर्तमानपत्राचा खप हा सध्या घसरला आहे. खप हा जाहीरातीचे दर ठरविणारा
एक प्रमुख घटक आहे. खप हा सदैव विकल्या गेलेल्या प्रतीएवढा नसतो,ज्यास
विक्रीखप असेही म्हणतात,कारण काही वृत्तपत्रे वाचकास विनामुल्य देण्यात येतात.
वाचकसंख्येचा आकडा हा वितरणाच्या आकड्यापेक्षा मोठाच असतो कारण असे
समजले जाते कि,वृत्तपत्राची विशिष्ट प्रत एकापेक्षा अधिक लोकं वाचतात.

जगात वृत्तपत्र वितरणात वाढ झाली असून, २००८ सालात दररोज ५४० दशलक्ष
प्रती विकल्या गेल्याचे जागतिक वृत्तपत्र संघटनेच्या अहवालात म्हटले आहे.जगात सर्वत्र
मंदीचा प्रभाव असतानाही वृत्तपत्राच्या विक्रीत १.३ टक्क्यांनी वाढ झाली असल्याचे
वृत्तपत्र संघटनेचे अध्यक्ष गाविन ओ रिली यानी सागितले. दररोज १.९ अब्ज लोक वृत्तपत्र
विकत घेऊन वाचतात. गेल्या पाच वर्षात वृत्तपत्राचा खप ८.८ टक्क्याने वाढला आहे.
[१] अनेक देशात,वेगवेगळ्या संस्थातर्फे वितरणाचे अंकेक्षण होते,
 त्याने जाहिरातदारास ही खात्री होते कि, अमुक एक वर्तमानपत्र हे खरेखरीच
प्रकाशकाने दर्शविलेल्या आकड्यापर्यंत पोचते.
```

Figure 6.1: Working Module of the Project

Above screen-shot shows the working module of the project. We have taken a paragraph in Devanagari Language and rendered it on the *VTE*. It is clear from the screen-shot that complex script is successfully rendered in *VTE*.

Above test case covers all the modules to be tested. The paragraph covers all simple Devanagari characters, Conjuncts, Diacritics, complex ligatures.

6.2 Test Plan

The implementation has been tested by taking number of test cases and testing the application rigorously. During testing we have tried increasing the accuracy, minimizing the errors, applying engineering corrections, etc. We have discovered some unhandled functionalities in the implementation which have been listed below.

1. Handling of Backspace and delete keys

   When these keys are pressed we should get the desired functionality i.e for the Backspace key the previous character should get deleted and the cursor position be updated depending upon the width of previous character and similarly for delete key.

2. Rendering the formatted data received from process output

   The formatted data received from process has to be rendered on the screen. However the data has been formatted considering a column to be equivalent of the character width hence the formatting of data is not appropriate.

3. Handling of navigation keys

   In similar to the Backspace key the navigation keys need to handled so as to navigate through the non-uniform cell structure taking into consideration the width of the different characters.

Chapter 7

Summary

We have implemented the Devanagari script rendering in GNOME-terminal with the help of inter cell interaction and using variable rendering widths for characters. This enables proper rendering of Devanagari script. This will guide the other Developers who are working on other languages will be able to render their Complex language script. This will lead to complete complex script support in GNOME-terminal.

We can say that our solution was highly needed by the Indic Language community. We hope that our approach to solve the problem will help other projects as well where rendering is the main issue for providing Indic Language support.

We have released the source code of complete *VTE* with Devanagari support [11]. Changes to the source code that we have made for Indic support are released under GNU General Public License v3.

Appendix A

GNOME Terminal Architecture

A.1 Introduction

GNOME Terminal is the most widely and commonly used terminal in the GNU/Linux community. It is a terminal emulation application designed to access UNIX shell from GNOME environment and also to run any application that is designed to run on xterm, VT102, VT220 terminals. Xterm is the terminal emulator for X window system. VT102 and VT220 are the video terminals designed by Digital Equipment Corporation.

A.2 Study of GNOME Terminal source code

GNOME Terminal is fully written in C language. *GIMP Tool Kit(GTK)* libraries, *VTE* (Virtual Terminal Emulator) library are used to write GNOME-terminal. It is completely written using object oriented approach. It follows the factory method design pattern. GNOME Terminal has many features like it supports multiple tabs, multiple profiles, mouse-events(limited), wide range of background options, colored text and URL detection. Due to these high level features that GNOME Terminal can support the source code has become very complex.

A.3 Design Pattern

In Software Engineering, There are many commonly occurring problems in software design. These problems are solved using a template or the generalized solution. This generalized solution or template is called Design Pattern. This Design pattern can be considered as the pathway to solve complex software design problems.

There are many types of design patterns like Creational patterns, Structural patterns, Behavioral patterns, Concurrency patterns etc. GNOME Terminal uses Factory method which is the Creational pattern.

A.4 Factory Method Design Pattern

As the name suggests the Factory method design pattern can be thought as factory that produces required objects as per requirements. In object-oriented terms Factory can be considered as the interface that creates objects.

Factory Method is a creational design pattern. It is used in situations where the application may not know which object of the class needs to be instantiated at that time. In this we model the interface which handles this problem. This interface allows the subclasses to decide which class need to be instantiated at that situation.

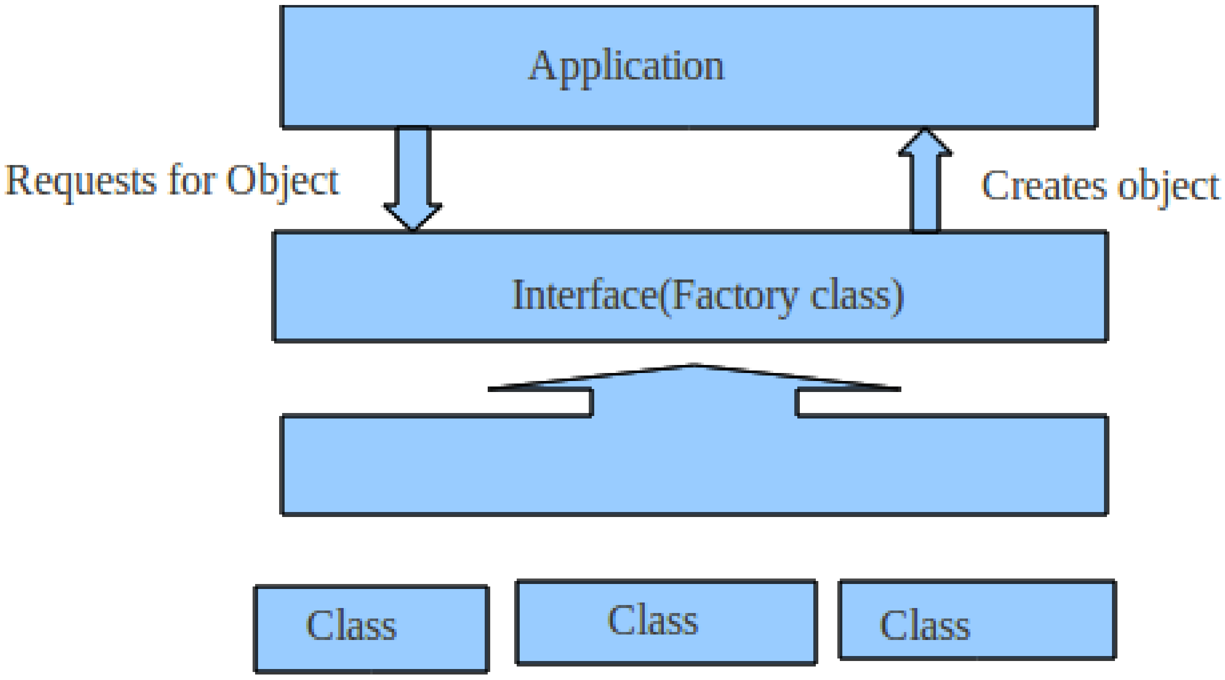


Figure A.1: Description of Factory Pattern

Appendix B

Virtual Terminal Emulator

B.1 Introduction

VTE library provides API for terminal emulation and GNOME-terminal is written using this *VTE* library. *VTE* is written in C language and GNOME Tool Kit(GTK) libraries. The widget management, widget behavior, handling of input etc is done by *VTE*.

B.2 Creation of widget

The main function in file *vteapp.c* is responsible for creating the Gtk widget of the emulator. The command line options given while running the code are parsed before creating the widget and handled accordingly. A *GoptionContext* structure is used which defines which options are accepted by the command-line option parser. The various properties and policies such shape of cursor, scroll bar policy are also defined here.

A top level Gtk window is created which which contains all other widgets such as a screen widget, a scrolled window. Thereafter a Terminal widget is created and is added to scrolling window.

The Terminal widget is then connected to signals indicating various events related to window manipulation such as character size has changed, window title has changed, maximize window, font increase etc.

The color maps and screen properties such as opacity are set in this function itself Depending upon the option a connection to console is created, or shell is used or a new child process is started under a newly allocated pseudo-terminal. The widget is then realized and all the widget on the top-level window are shown. A signal is connected to the window for delete-event i.e when the window is deleted. Until the signal for delete is called the code remains in the main loop.

B.3 Event handlers and methods

In the function *vte_terminal_class_init()* in *vte.c* file a number of event handlers are initialized for the terminal widget such as key press, key release, button press, button release, expose event, scroll event etc. Depending upon the event the corresponding event handler will be called. The methods for the event handlers are also defined in *vte.c*. The signals to be emitted for the various events, stated in the *main()* part are also defined and initialized here.

B.4 Key press event

When a key is pressed the event handler for key press calls the respective function i.e *vte_terminal_key_press* in *vte.c*. The function reads and handles the key press event. Firstly it is checked if the key does anything to Gtk widgets behavior, then it is handled accordingly. If the key press is not related to widget behavior then it is checked for other possibilities i.e modifier key, characters to be displayed etc. If the key is just a modifier key such as Alt, Shift etc then a note that a modifier key is pressed is made.

Now a input method filter which determines if the key press can be handled by the input method. The input method can handle the keys which are related to typing of characters. If yes then the input method takes the key press and handles it. A commit signal is emitted to indicate the input method to send the data to child. If the input method filter returns false then look for other possibilities.

The key is mapped to sequence names such as backspace, delete, insert, up, down etc. If the mapping is possible then it is handled appropriately. If key is not handled by above means then try mapping it to a literal or capability name. Then the keys are to be sent to child to do further processing.

Appendix C

Glossary

1. Diacritics

   A diacritic is an ancillary glyph added to a letter or basic glyph. Diacritical marks may appear above or below a letter, or in some other position such as within the letter or between two letters.

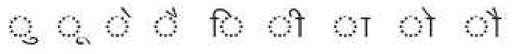


Figure C.1: Example of diacritics

2. Glyph

   A glyph is a graphical representation of an element of writing that contributes to meaning of what is written. A single character may have different glyph in different font. Below figure shows glyphs for Latin character “a” in different fonts.

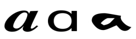


Figure C.2: Example of glyph

3. Font

   A font a contains a set of images representing the glyphs, characters of a script and other information such as scaling etc.

4. Zero-width character

   A zero-width character is one which when attached to the base glyph does not require additional width to draw.

   ◌ु ◌ू ◌े ◌ै

   Figure C.3: Example of zero-width character

5. Non-zero-width character

   A Non-zero-width character is one which when attached to the base glyph requires additional width to draw.

   ◌ी ◌ि ◌ा ◌ौ ◌ो

   Figure C.4: Example of non-zero-width character

6. Halant

   A *halant* character is one when used between two characters joins them and forms a combined character

7. Conjunct

   A conjunct is a combined character formed by joining two base consonants characters using a *halant* character.

   द्ध प्नी क्ख स्व त्स्य

   Figure C.5: Example of conjunct

8. GNOME

   GNOME is a free software project consisting of The GNOME desktop environment, a attractive desktop for users and the GNOME development platform, an extensive framework for building applications that integrate into the the desktop. It is the most popular desktop environment for GNU/Linux and UNIX-type operating systems.

9. KDE

   KDE is a free software project that offers an advanced graphical desktop, a wide variety of cross-platform applications and a platform to easily build new applications upon. KDE software is based on the Qt framework.

10. Qt

    Qt is a non-GPL, cross-platform application framework that is used for developing application software with a graphical user interface (GUI) hence it is also called as Qt widget toolkit. It is also used for developing non-GUI tools such as command-line tools and consoles for servers.

11. Pango

    Pango is an LGPL licensed library for laying out and rendering of text, with an emphasis on internationalization. The GTK+ UI toolkit uses pango for all of its text rendering. It provides cross-platform support, so that pango-rendered text will appear similar under different operating systems, such as LINUX, Apple's Mac OS and Microsoft Windows.

12. GTK- GIMP Tool Kit

    GTK is a LGPL licensed widget toolkit for creating graphical user interfaces and provides cross platform compatibility and an easy to use API.

Bibliography

[1] American standard code for information interchange.
http://en.wikipedia.org/wiki/ASCII.

[2] Arabic, hebrew: character alignment not working properly.
http://bugzilla.gnome.org/show_bug.cgi?id=321490.

[3] Cairo graphics documentation, manual and tutorials on the official cairographics site.
http://www.cairographics.org/documentation/.

[4] Faqs for unicode and utf-8. http://unicode.org/faq/utf_bom.html.

[5] Internationalization and localization.
http://en.wikipedia.org/wiki/Internationalization_and_localization.

[6] Official kde web site. http://www.kde.org/.

[7] Official pango web site. http://www.pango.org/.

[8] Official website of unicode consortium and unicode standard. http://unicode.org/.

[9] Pango cairo rendering at gnome developer site.
http://developer.gnome.org/pango/stable/pango-Cairo-Rendering.html.

[10] Pango reference manuat at gnome. http://developer.gnome.org/pango/stable/.

[11] Sourceforge link of our project.
http://sourceforge.net/p/vteindic/home/.

[12] Swathanthra indian language computing project or silpa.

http://silpa.org.in/.

[13] (vtecomplex) gnome-terminal doesn't handle unicode complex text rendering.

http://bugzilla.gnome.org/show_bug.cgi?id=584160.

[14] Wikipedia definition of devanagari. http://en.wikipedia.org/wiki/Devanagari.

[15] Wikipedia details of complex scripts or complex script layout.

http://en.wikipedia.org/wiki/Complex text layout.

[16] Wikipedia details of qt framework. http://en.wikipedia.org/wiki/Qt_(framework).

[17] Wikipedia details on unicode. http://en.wikipedia.org/wiki/Unicode.