Programming Practice Report

of Software Engineering School

SUBJECT	Phonebook
AUTHOR	WEIJU LAN (兰威举)
MAJOR	Software Engineering
CLASS	2013 Class 1
ID	13108115

Table of contents

Chapter 1
Design

1.1 Goals

The phonebook is designed to be a GUI Graphical User Interface application for `GNOME 3`. It's designed to be modern, safe and linux-only. The whole project is written in `GNU C++14`.

The phonebook should store the contact's name and phone number. The user should be able to add a new contact, edit or remove an existing contact, and search the phonebook for contacts by name or by phone. The saving process should be transparent to the user, so that the user need not to worry about saving.

1.1.1 The GUI

The GUI should be modern and straightforward like any other `GNOME 3` applications. It should have only 1 window displaying the data (in this case, the contact list) with most of the actions on the titlebar.

1.1.2 The Storage System

The data should be stored using NDE Non-Destructive Editing strategy.

In this way, we should only save the operations/actions on the data instead of the data itself. Thus the whole editing history is saved, so the user is able to undo/redo even after closing and reopening the application. Because only the actions are saved, we can do this *on the fly* so the user don't need to click "save" all the time, making the saving process *transparent* and safe (the old actions won't be missing because we only append new actions to the end of file).

All the actions should be saved in *plain text*, each action in one line, so that once saving failure happens, the user can recover easily by removing the last line.

But, huge editing history may cause the application to load slowly and eat lots of memory. So another mechanism is provided: *snapshot*.

Snapshotting should remove all the actions and save the data (instead of actions) in a compressed compact binary format. It should first save to a temporary file, then rename to the original file to avoid saving failure.

After snapshotting, the editing history will be *lost*. Thus the user cannot undo the actions that happened before snapshotting.

1.2 User Interaction Diagram

See Appendix A for details.

Chapter 2 Implementation

2.1 Overview

- The whole project uses `git` to do the version control
- The building system uses a custom one of mine, which includes a `perl`-written `configure` script that generates `GNU makefile`. Then the actual building is coordinated by `GNU make`, which calls `g++` to build the whole application.
- The application is written in `GNU C++14`, use `gtkmm 3` for GUI and `zlib` for compression.
- Designed with `C++`'s multi-paradigm in mind. Used OOP Object-Oriented Programming, FP Functional Programming, GP Generic Programming and MP Meta-Programming techniques.

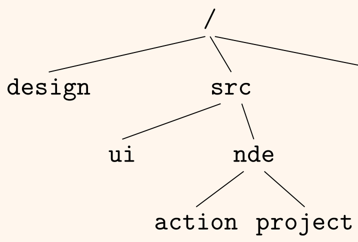


Figure 2.1. Source tree.

`/design`	the design drafts and documentations
`/res`	the resources: a `desktop` file and some experimenting contacts data
`/src`	all the source code
`/src/ui`	the user interface and interaction code
`/src/nde`	the non-destructive editing engine
`/src/nde/action`	all the actions
`/src/nde/project`	an nde project (a manager of actions and file operations)

Table 2.1. Directories

```
phonebook
|--------  main
|--------  ui
|          |---  main_window
|          |---  contact_input_bar
|          |---  contact_search_bar
|          |---  contact_list
|          |---  info_entry
|          |---  contact
|--------  nde
           |---  init
           |---  path
           |---  factory
           |---  project
           |     |-----------------  type
           |     |-----------------  instance
           |     |-----------------  data
           |     |-----------------  file
           |     |-----------------  snapshot
           |---  action
                 |-----------------  type
                 |-----------------  helper
                 |-----------------  wrapper
                 |-----------------  undo
                 |-----------------  redo
                 |-----------------  create
                 |-----------------  remove
                 |-----------------  edit
```

Figure 2.2. Components

2.2 Phonebook

2.2.1 `main`

The `main` function creates a new `gtk3 Application` and a new `MainWindow`, then run the application with the new window.

2.3 The GUI

2.3.1 `main_window`

The `MainWindow` class creates the main window, populate the titlebar with a `HeaderBar` with `Buttons` on it, create `ContactInputBar` and `ContactSearchBar`, populate the window content with `ContactList` and connect various `signals` to provide user interactions.

Some `Buttons` on the `HeaderBar` will call `nde::action::`······ to perform the action.

2.3.2 `contact_input_bar`

When adding or editing a contact, the `ContactInputBar` will be shown to ask the user to input the contact information (i.e. contact's name and phone number).

When the user confirmed his input, the `ContactInputBar` will ask its `InfoEntry`s to validate the inpur. The validating method is passed to the `InfoEntry`s when constructing by using `lambda functions` of `C++14`.

2.3.3 `info_entry`

`InfoEntry` is a subclass of `gtk3`'s `Entry` to provide *input-validation* functionality. The validating function `Validator` is passed to the `InfoEntry` as a `functor`. The validation will be performed on the fly when the user is inputing. The validation can also be asked explicitly to force the validation and get the result.

2.3.4 `contact_search_bar`

`ContactSearchBar` is a subclass of `gtk3`'s `SearchBar` to wrap the `gtk3`'s `SearchEntry` inside, making the code easier to write.

2.3.5 `contact_list`

`ContactList` is used to show all the `Contact`s. When constructing, it will connect the `nde`'s callbacks by using `lambda function`.

2.3.6 `contact`

`Contact` displays one contact's information using `Label`s with `markup` for formatting. The layout is constructed by various `boxes`. It also contains a `ContactInputBar` for editing the contact. It will call `nde::action::`······ to perform editing and removing.

2.4 The NDE

The NDE is a hand-crafted non-destructive editing engine. Normally, it stores actions, but when asked to snapshot, it will store data and remove all the actions.

2.4.1 `path`

This `namespace` is used for getting various file paths like the project's path or the snapshot's path.

2.4.2 `factory`

Template class `Factory<Base>` is used to create a factory, which creates objects of the subclasses of `Base` by their names.

Proxy template class `Maker<T>` (where `T` is a subclass of `Base`) is used to register a subclass into the factory.

For instructions on how to use these, see the source code `/src/nde/factory.hh`.

2.4.3 `project/type`

`project::Type` is a manager that manages actions, data and files.

2.4.4 `project/instance`

`project::instance()` is used as a singleton. It returns a single instance of `project::Type`.

2.4.5 `project/data`

`project::Data` stores the contact data: name and phone number.

2.4.6 `project/file`

`project::File` is used for serialization of actions. It provides interface to read/write escaped quoted string (by using `std::quotes` in `<iomanip>` of `C++14`), integers, atoms and end of lines.

2.4.7 `project/snapshot`

This `namespace` is used for reading/writing snapshot files. A snapshot file is a `gzip`-compressed compact binary data.

2.4.8 `action/type`

`action::Type` is the *abstract* base class of actions.

It also contains the factory of `action::Type`.

2.4.9 `action/helper`

This `namespace` is used for helping the reading/writing of actions.

2.4.10 `action/wrapper`

The template class `action::Wrapper<Action>` is used to wrap a subclass of `action::Type`, making it registered to the `action::factory`. The internal can also easierly construct the action by just calling the wrapped object (`functor`, it implements `operator()(...)`).

2.4.11 `undo`, `redo`, `create`, `remove` and `edit` in `action`

They are the concrete actions derived from `action::Type`.

Taking `create` for example. It implements reading/writing using `project::File`. Two callbacks exist as static member variables. They are `invoke_cb` and `undo_cb` implemented by using `std::function` of `C++14`. Callbacks will be called when corresponding operations happened. It also uses the wrapper by `Wrapper<Create> create{"create"}`.

`undo` and `redo` have no callback.

Chapter 3
Summary and Thoughts

Even a small phonebook application is complex, especially when you want it to be mature.

I've heard of non-destructive editing long time ago, but this is the first time I've tried to implement one. Some dirty hacks exist though, but the code is clean in general. This is also a chance for me to learn `gtkmm 3`.

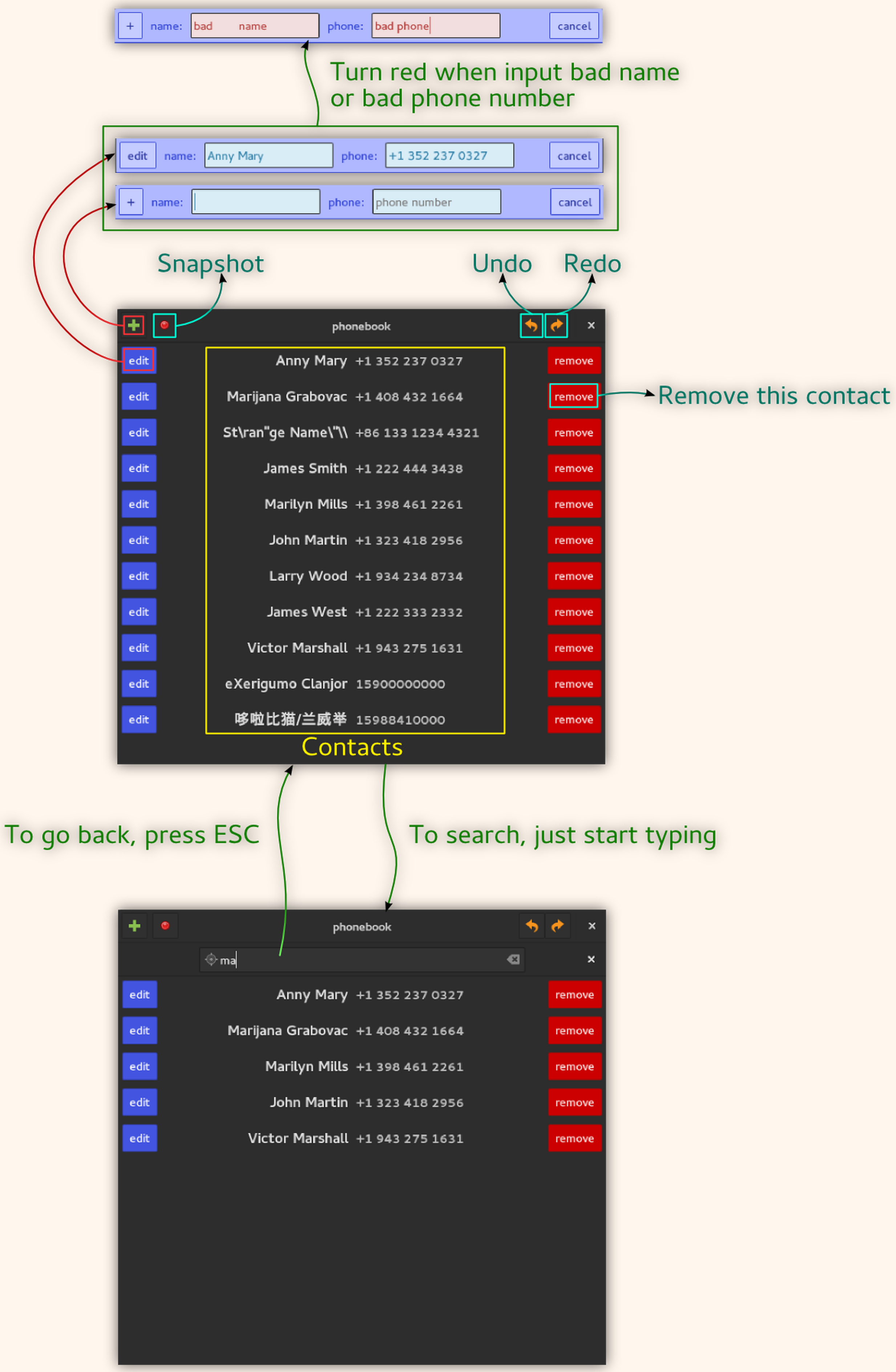
Applying various techniques from different paradigms is really important to write *clean*, *simple* and *readable* code in `C++`. Hand-crafting the GUI is so complex. I would use a GUI design tool next time.

`C++14` rocks. I hope `clang` will support it soon. The debugging information from `clang` is much better than that from `GCC`.

Software Enginnering School

CLASS	2013 Class 1
AUTHOR	WEIJU LAN (兰威举)
ID	13108115
DATE	2014/06/25

Appendix A



edit	Anny Mary	+1 352 237 0327	remove
edit	Marijana Grabovac	+1 408 432 1664	remove
edit	St\ran"ge Name\"\\	+86 133 1234 4321	remove
edit	James Smith	+1 222 444 3438	remove
edit	Marilyn Mills	+1 398 461 2261	remove
edit	John Martin	+1 323 418 2956	remove
edit	Larry Wood	+1 934 234 8734	remove
edit	James West	+1 222 333 2332	remove
edit	Victor Marshall	+1 943 275 1631	remove
edit	eXerigumo Clanjor	15900000000	remove
edit	哆啦比猫/兰威举	15988410000	remove

edit	Anny Mary	+1 352 237 0327	remove
edit	Marijana Grabovac	+1 408 432 1664	remove
edit	Marilyn Mills	+1 398 461 2261	remove
edit	John Martin	+1 323 418 2956	remove
edit	Victor Marshall	+1 943 275 1631	remove