CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 24
April 26, 2016

Design Patterns (continued)

Graphical User Interfaces

The gtkmm Framework

Design Patterns (continued)

Subject-Observer or Publish-Subscribe: problem

Problem: Your application program has many classes and many objects of some of those classes. You need to maintain consistency among the objects so that when the state of one changes, its dependents are automatically notified. You do not want to maintain this consistency by using tight coupling among the classes.

Example: An OO spreadsheet application contains a data object, several presentation "views" of the data, and some graphs based on the data. These are separate objects. But when the data changes, the other objects should automatically change.

Subject-Observer or Publish-Subscribe: pattern

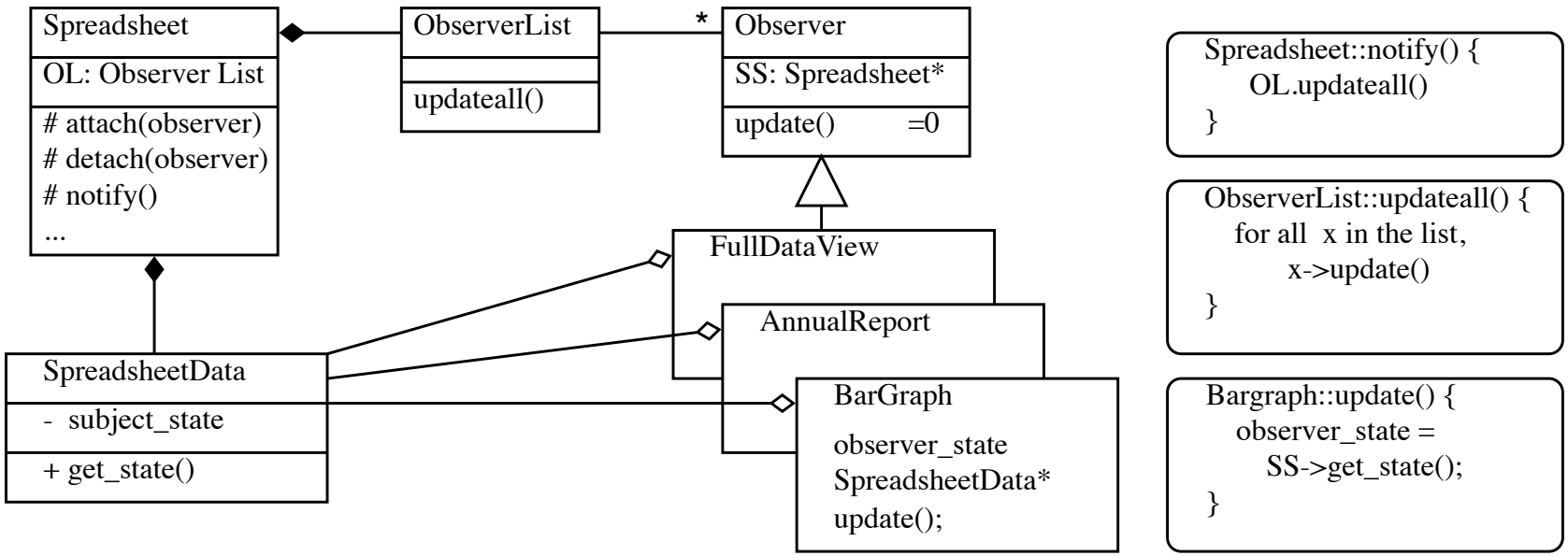
Call the SpreadsheetData class the **subject**; the views and graphs are the **observers**.

The basic Spreadsheet class composes an observer list and provides an interface for attaching and detaching Observer objects.

Observer objects may be added to this list, as needed, and all will be notified when the subject (SpreadsheetData) changes.

We derive a concrete subject class (SpreadsheetData) from the Spreadsheet class. It will communicate with the observers through a `get_state()` function, that returns a copy of its state.

Subject-Observer or Publish-Subscribe: diagram



See textbook for more details.

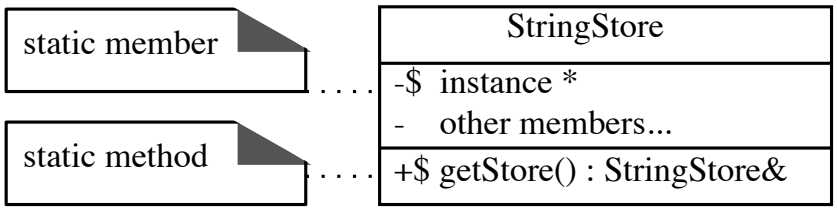
Singleton pattern

Suppose you need exactly one instance of a class, and objects in all parts of the application need a single point of access to that instance.

Solution: A single object may be made available to all objects of class `C` by making the singleton a static member of class `C`.

A class method can be defined that returns a reference to the singleton if access is needed outside its defining class.

`StringStore` example




```
StringStore& StringStore::getStore(){
   if (instance==NULL) instance = new StringStore;
   return instance;
}
```

Example: Suppose several parts of a program need to use a StringStore. We might define StringStore as a singleton class.

The `StringStore::put()` function is made `static` and becomes a global access point to the class, while maintaining full protection for the class members.

Graphical User Interfaces

User Interfaces

Modern computer systems support two primary general-purpose user interfaces:

Command line: User input is via a command line typed at the keyboard. Output is character-based and goes to a physical or simulated typewriter-like terminal.

Graphical User Interface (GUI): User input is via a pointing device (mouse), button clicks, and keyboard. Output is graphical and goes to a window on the screen.

Interfaces for C++

The C++ standard specifies a command line interface: `iostream` and associated packages. No similar standard exists for GUIs.

De facto GUI standards in the Linux world are `GTK+` (used by the Gnome desktop) and `Qt` (used by the KDE desktop).

`GTK+` is based on C; `Qt` is based on an extension of `C++` and requires a special preprocessor.

`gtkmm` is a `C++` wrapper on top of `GTK+`.

Advantages: Provides type safety, polymorphism, and subclassing.
Provides a native interface to C++ code.

Disadvantages: Components not so well integrated.
Documentation spread between `gtkmm`, `gtk+`, and other components but improving.

Overall Structure of a GUI

A GUI manages one or more *windows*.

Each window displays one or more *widgets*.
These are objects that provide graphical and textual input and output to the program.

A GUI package such as `gtkmm` maintains a *widget tree*.

A widget controls a particular kind of user input or output.
Examples: label, text box, drawing area, button, scroll bar, etc.

Concurrency and Events

The central problem in building a GUI is handling concurrency.

Data arrives from multiple concurrent sources – mouse and keyboard, network, disk, other threads, etc.

We call the arrival of a piece of data an **event**.

- Event arrival times are unpredictable.
- Events should be processed as quickly as possible.

For example, to have a good interactive feel, the GUI should respond to a mouse click event within milliseconds.

Event Loop

An **event loop** allows a single thread to manage a set of events.

At some level, the hardware or software **polls** for events.

When an event is detected, it is **dispatched** to an **event handler**.

The event handler either processes the event itself, queues a task for later processing, or spawns a thread to process it.

While the event thread is processing one event, no other events can be processed, so event handlers should be short.

Problem is to prevent a long-running low-priority event handler from delaying the handling of a high-priority event.

A GUI event structure

A GUI typically translates raw hardware events into semantically-meaningful software events.

For example, a mouse click at particular screen coordinates might turn into a button-pressed event for some widget in the GUI tree.

Several system layers may be involved in this translation, from the kernel processing of hardware interrupts at the bottom level, up through device drivers, windowing systems such as X, and finally a GUI frameworks such as GTK+.

Interface between user and system code

A major software challenge is how to design the interface between the GUI and the user code that ultimately deals with the events.

In a command-line interface, the user code is at the top level. It connects to the lower layers through familiar function calls.

With a GUI, things are turned upside down.

- The top level is the main event loop.
- It connects to the user by calling appropriate user-defined functions.

Binding system calls to user functions

How can one write the GUI to call user functions that did not even exist when the GUI system itself was written?

The basic idea is that of **interface**.

- The interface is a placeholder for the eventual user functions.
- It describes what functions the user will provide and how to call them but not what the functions themselves are.
- The interface is bound to user code either when the user code is compiled or dynamically at runtime.

Polymorphic binding

C++ virtual functions provide an elegant way to bind user code to an interface.

- The GUI can provide a virtual default event handler.
- The user can override the default handler in a derived class.

Of course, the actual binding occurs at run time through the use of type tags and the `vtable` as we have seen before.

Binding through callback registration

The user explicitly **registers** an event handler with the GUI by calling a special registration function.

- The GUI keeps track of the event handler(s) registered for a particular event.
- When the event happens, it calls all registered event handlers.

This is sometimes called a *callback* mechanism since the user asks to be called back when an event occurs.

Callback using function pointers: GUI side

Callbacks can be done directly in C . Here's the GUI code:

1. Define the signature of the handler function:
   ```
   typedef void (*handler_t)(int, int);
   ```
2. Declare a function pointer in which to save the handler:
   ```
   handler_t buttonPressHandlerPtr;
   ```
3. Define a registration function:
   ```
   void systemRegister(int slot, handler_t f) {
     button_press_handler_ptr = f;
   }
   ```
4. Perform the callback:
   ```
   buttonPressHandlerPtr(23, 45);
   ```

Callback using function pointer: User side

Here's how the user attaches a handler to the GUI:

1. Create an event handler:
```
void myHandler(int x, int y) {
  printf("My handler (%i, %i) called\n", x, y);
}
```
2. Register the handler for event 17:
```
systemRegister(17, myHandler);
```

Type safety

The above scheme does not generalize well to multiple events with different signatures.

- Registered handlers need to be stored in some kind of container.
- For type safety, each different handler signature requires a different event container and registration function of the corresponding signature.

The alternative is for `systemRegister()` to take a `void*` for its second argument and to cast function pointers before call them.

This is not type safe and can lead to subtle bugs if the wrong type of function is attached to a callback slot.

Signals and slots

Signals and slots is a more abstract way of linking events to handlers and can be implemented in a type safe way.

- A `connect()` template function is used to bind a `signal` to a `slot`.
- An event `emits` a signal.
- A handler is associated with a slot.
- Whenever the event occurs, the functions associated with all connected slots are called.

Several signals can be connected to the same slot, and several slots can be connected to the same signal.

The gtkmm Framework

Structure of gtkmm

gtkmm relies on several libraries and packages:

- gtkmm-3.0 is the GUI engine.
- gdkmm is a device layer used by gtk.
- cairomm is a vector graphics drawing package.
- pango is a library for laying out and rendering of text, with an emphasis on internationalization.
- sigc++ is a library for connecting events (signals) to event handlers (slots).

Compiling a gtkmm program

Many include files and libraries are needed to compile and build a gtkmm program.

A utility pkg-config is used to generate the necessary command line for the compiler.

> pkg-config gtkmm-3.0 --cflags

```
-pthread
-I/usr/include/gtkmm-3.0 -I/usr/lib64/gtkmm-3.0/include
-I/usr/include/atkmm-1.6 -I/usr/include/gtk-3.0/unix-print
-I/usr/include/gdkmm-3.0 -I/usr/lib64/gdkmm-3.0/include
-I/usr/include/giomm-2.4 -I/usr/lib64/giomm-2.4/include
-I/usr/include/pangomm-1.4 -I/usr/lib64/pangomm-1.4/include
-I/usr/include/glibmm-2.4 -I/usr/lib64/glibmm-2.4/include
-I/usr/include/gtk-3.0 -I/usr/include/at-spi2-atk/2.0
-I/usr/include/at-spi-2.0 -I/usr/include/dbus-1.0
-I/usr/lib64/dbus-1.0/include -I/usr/include/gtk-3.0
-I/usr/include/gio-unix-2.0/ -I/usr/include/cairo
-I/usr/include/pango-1.0 -I/usr/include/harfbuzz
-I/usr/include/pango-1.0 -I/usr/include/atk-1.0
-I/usr/include/cairo -I/usr/include/cairomm-1.0
-I/usr/lib64/cairomm-1.0/include -I/usr/include/cairo
-I/usr/include/pixman-1 -I/usr/include/freetype2
-I/usr/include/libpng16 -I/usr/include/freetype2
-I/usr/include/libdrm -I/usr/include/libpng16
-I/usr/include/sigc++-2.0 -I/usr/lib64/sigc++-2.0/include
-I/usr/include/gdk-pixbuf-2.0 -I/usr/include/libpng16
-I/usr/include/glib-2.0 -I/usr/lib64/glib-2.0/include
```

Linking a gtkmm program

`> pkg-config gtkmm-3.0 --libs`
generates the necessary linker flags.

```
-lgtkmm-3.0 -latkmm-1.6 -lgdkmm-3.0 -lgiomm-2.4
-lpangomm-1.4 -lglibmm-2.4 -lgtk-3 -lgdk-3
-lpangocairo-1.0 -lpango-1.0 -latk-1.0 -lcairo-gobject
-lgio-2.0 -lcairomm-1.0 -lcairo -lsigc-2.0
-lgdk_pixbuf-2.0 -lgobject-2.0 -lglib-2.0
```

To use package config, use the backquote operator on the `g++` command line:

Compiling: `g++ -c ‘pkg-config gtkmm-3.0 --cflags‘ ...`
Linking: `g++ ‘pkg-config gtkmm-3.0 --libs‘ ...`
Both: `g++ ‘pkg-config gtkmm-3.0 --cflags --libs‘ ...`

Using a GUI

The following steps are involved in creating a GUI using gtkmm:

1. Initialize gtkmm.
2. Create a window.
3. Create and lay out widgets within the window.
4. Connect user code to events.
5. Show the widgets.
6. Enter the main event loop.

The GUI then displays the window and waits for events.
When an event occurs, the corresponding user code is run.
When the event handler returns, the GUI waits for the next event.

Example: `clock`

The code example `24d-clock` is a significant extension of the clock example in the `gtkmm` tutorial book.

It illustrates many of the features of `gtkmm`.

Main program

```
#include <gtkmm.h>
#include "clockwin.hpp"

int main(int argc, char* argv[])
{
   auto app =
     Gtk::Application::create(argc, argv, "org.gtkmm.examples");
   ClockWin win;
   return app->run(win);
}
```