Mar. 21, Asmt. #4, xcal, due Thu. Apr. 7 @ 11:59 pm and Apr. 8 in class

General submission instructions (subversion)	
Subversion tag	cis2750-a4
What to submit	`demo.ics, makefile, xcal.py, calutil.h/c, caltool.h/c,` pledge.txt and any other source files needed via online submission. It must be possible to type "make" (no target) to build your application, and "python3 xcal.py" to run it. A .py extension is optional on xcal. Drop off your testing checklist on Thu. or in class Fri. Your assignment will not be marked without a checklist! *coming*

> NOTE: Our official MySQL server, only available on the campus network, has the **hostname** of **dursley.socs.uoguelph.ca** . It is running version 5.5, and documentation links can be found on the Class Schedule page. Your **username** is the same as your usual SoCS login ID, and your **password** is your 7-digit student number. A **database** has been created for you with the same name as your username. You have permission to create and drop tables within your own named database.

For this assignment we will enhance the GUI created for assignment #3, while keeping the underlying C code from #1 and #2, and adding some modest database functionality to the Python code. The purpose of tying xcal into a DBMS is to allow the user to build up and query a personal database of selected iCalendar items (events and to-dos). This won't be hugely "useful"; it's more for demonstration purposes.

The assignment's educational objective is to see how little trouble it can be to connect to a powerful DBMS and embed industry-standard SQL statements in an application. We will embed SQL in our Python code with the aid of the **mysql.connector** module (documentation link on Class Schedule page) supplied by MySQL, but we could alternatively do it via C functions and MySQL's C API. Since the data display is being done by a Python program, it's more convenient to contact the DBMS from there.

To connect with your database (which consists of a set of tables under your control), the following info is needed by a MySQL client: (1) hostname of the MySQL server; (2) username; (3) password; and (4) database name (in principle, a user can control more that one database). Thus, at a command prompt where you invoke the **mysql** client, you would enter:

```
mysql -h dursley.socs.uoguelph.ca -D username -u username -p stnum
```

You can omit "-u username" if the account you are logged into has your same username. You may be able to omit "-D username" if it uses your only database by default. For privacy, you can omit your student number from the command line (just type -p by itself), in which case mysql will prompt for it. Try the `status` command after logging in.

To obtain that same info for your Python program's use, modify xcal to require 1-2 command line arguments as follows:

```
xcal.py username [hostname]
```

If hostname is omitted, it is assumed to be "dursley.socs.uoguelph.ca". You would only need to specify hostname to run xcal from off campus. For example, you can set up MySQL on your own computer (localhost), if you are willing to administer it. The TAs will only test with the 1-argument version using their own database (i.e., not your account). We don't want to embed a password in the source code, and don't want to type it as a command line argument (which anyone on the computer can observe with the "w") command,

so xcal will prompt for it on the console and accept it in the usual non-printing fashion.

So a command line could look something like:

```
xcal.py jdoe

xcal.py jdoe localhost
```

xcal should assume that the database name is the same as the username. Obtain a password and try the connection as soon as the program starts. If xcal fails to connect to the server and the database it wants, print an error message **on the console** and allow a maximum of 3 tries. If connecting still fails, do not bother checking for the DATEMSK environment variable and do not display the GUI!

> **NOTE on transactions:** MySQL is able to implement transactions as required in the SQL standard. By default, it runs in "autocommit mode" meaning that programmers can ignore transactions unless they specifically want to use them. However, the MySQL Connector/Python module that we are using reverses the default and *disables* autocommit mode. This means that whenever you change the contents of a table you must invoke the `commit()` method on the connection object, otherwise your changes will not take effect.
>
> **NOTE on names:** You must not change the names of menus, menu commands, buttons, tables, columns, and the like that are specified below. Note that user-specified names in SQL (tables, columns) are case sensitive, but SQL keywords are not. Such changes do not fall within "artistic discretion." This requirement is intended to ease marking and allow, for example, tables to be prepared with test data in advance. If you change specified names, you cannot get full credit on those checklist items.

We're interested in storing data about calendar events and to-do items plus their respective organizers. Thus, the schema for your database consists of three tables named **EVENT, TODO,** and **ORGANIZER** . The concept is that every unique organizer named by various calendar components is stored in the ORGANIZER table, while the events and to-do items refer to their organizers by means of foreign keys.

Column names, data types, and constraint keywords are listed below. When your program executes, it must create these tables in your database if they do not already exist (that is, try creating them, which will fail if they are already there).

Table ORGANIZER

1. **org_id:** INT, AUTO_INCREMENT, PRIMARY KEY. The AUTO_INCREMENT keyword gives MySQL the job of handing out a unique number for each organizer so your program doesn't have to do it. Here we adopt the technique of using a *dataless* key.

2. **name:** VARCHAR(60), NOT NULL. The value from an ORGANIZER property's CN parameter.

3. **contact:** VARCHAR(60), NOT NULL. The value of the ORGANIZER property (typically "mailto:...").

Table EVENT

FOREIGN KEY(organizer) REFERENCES ORGANIZER(org_id) ON DELETE CASCADE

1. **event_id:** INT, AUTO_INCREMENT, PRIMARY KEY.

2. **summary:** VARCHAR(60), NOT NULL. The value from the SUMMARY property.

3. **start_time:** DATETIME, NOT NULL. The value from the DTSTART property, converted into MySQL's date/time format.

4. **location:** VARCHAR(60). The value from the LOCATION property. NULL if missing.

5. **organizer:** INT. The value of the ORGANIZER property's CN property, represented as a row in

the ORGANIZER table. NULL if missing. FOREIGN KEY REFERENCES establishes a foreign key to the org_id column over in the ORGANIZER table, and deleting the latter's row will automatically *cascade* to delete all its referencing events.

Table TODO

FOREIGN KEY(organizer) REFERENCES ORGANIZER(org_id) ON DELETE CASCADE

1. **todo_id:** INT, AUTO_INCREMENT, PRIMARY KEY.
2. **summary:** VARCHAR(60), NOT NULL. The value from the SUMMARY property.
3. **priority:** SMALLINT. The value from the PRIORITY property. NULL if missing.
4. **organizer:** INT. Same as in EVENT table.

GUI enhancements

To control the iCalendar database, add a **Database** menu with the following commands, which are individually only active when it is logically meaningful. After every command (except Status and Query), print in the log a status line based on a count of each table's rows: "Database has N organizers, N events, N to-do items."

- **Store All:** This command is used to insert all the VEVENT and VTODO components presently in the FVP into your database. This is active if the FVP contains components. Go through these steps for each component:

    1. Obtain all necessary data that should be stored.
    2. Check if its organizer name, if any, is already in the ORGANIZER table. If not, insert it along with its contact field. *It's understood that if the same organizer name happens to appear in multiple components with differing contact values, the first instance processed will "stick".*
    3. For an event, check if it is already in the EVENT table; both summary and start_time must match, otherwise it's considered a different event. If not, insert it, including a reference to its organizer, if applicable, and NULL for any missing fields.
    4. For a to-do item, check if it is already in the TODO table; both summary and start_time must match, otherwise it's considered a different event. If not, insert it including a reference to its organizer, if applicable, and NULL for any missing fields.

- **Store Selected:** Same as Store All, except it works on only the component currently selected in the FVP. This is only active if a component is selected.
- **Clear:** Truncate or delete all rows from all the tables (which may have to be done in a certain order due to the foreign keys). This is only active if any table is not zero length.
- **Status:** This displays in the log the status line described above and is always active.
- **Query...:** This command, which is always active, opens a secondary window (not modal) that is used to query the database. Only a single query window can be opened, and it can be closed by clicking on its [X] title bar button. It displays the following two panels:

    1. *Query:* These controls are used to make ad hoc queries on the database. There are several "canned transactions" (see below) which the user can choose among, parameterize, and submit. There is also a text area initialized with the word "SELECT" where the user can finish typing an arbitrary SELECT statement, or even backspace and type *any* SQL statement, if they want to risk it. There is a **Submit** button to send the chosen canned transaction or SQL statement to the database. There is also a **Help** button which pops up a small panel showing the output of

"DESCRIBE ORGANIZER; DESCRIBE EVENT; DESCRIBE TODO;" giving precise table and column names--information the user will need to correctly format a query.

2. *Results:* The results (or errors) from the submitted query are displayed here in a scrolling text area. Like the Log, new results are added to the bottom, and it has a **Clear** button. Output a line of dashes between each set of results.

Canned transactions

The Database > Query menu's query panel displays at least four preformatted transactions, with some "variables" (textboxes) that the user may fill in. There should be some way (your choice) to select one of the transactions to be performed when Submit is pressed. Two transactions are given below; **you should come up with three additional different ones that are not too lame.** (For example, "Display all the events" would be lame, i.e., anything that doesn't have conditions, a join, a nested query, and/or aggregate functions.) **You must supply an ICS file demo.ics** that can be used to demonstrate your queries. The transactions must be displayed in simple English, but your program will generate the underlying SQL statements incorporating any filled-in variables.

1. **Display the items of organizer _____ (SQL wild card % is permitted).** This query fetches the summaries of all events and to-do items organized by a name like the one given.

2. **How many events take place in _____ (location)?** This query prints a count of the events with the matching location.

3. Your choice, must involve a date/time (re EVENT.start_time).

4. Your choice, must involve a priority (re TODO.priority).

5. Your creative choice...