Pomotux

A Tool for Activity Management following the Pomodoro Technique

Riccardo Buttarelli, 5909, riccardo.buttarelli@stud-inf.unibz.it
Daniel Graziotin, 4801, daniel.graziotin@stud-inf.unibz.it
Massimiliano Pergher, 5821, massimiliano.pergher@stud-inf.unibz.it

Summary

Application Domain

Resource Management

Vision statement

The aim of the Software Engineering Project project is to build a C++ program following an Agile software methodology (XP@Scrum).

The aim of our project is to develop an activity manager for the Pomodoro Technique[1] crated by Francesco Cirillo, a member of the XPlabs[2] crew.

Our program focuses on the basic features of the technique. It does not focus on advanced techniques, such as the prediction of the number of pomodoros needed for an activity.

The Pomodoro Technique

From the Website:

```
The Pomodoro Technique is a time management method that can be used for any kind
of task. For many people, time is an enemy. The anxiety triggered by “the
ticking clock”, especially when a deadline is involved, leads to ineffective
work and study habits which in turn lead to procrastination. The aim of the
Pomodoro Technique is to use time as a valuable ally in accomplishing what we
want to do in the way we want to do it, and to enable us to continually improve
the way we work or study.
```

The Technique is heavily explained on a 60+ pages book published on the website.

Please visit the official website for more explanations.

System Entities

Pomotux heavily relies on the entities described in the Pomodoro Technique, that are introduced below:

- **Activity** – the task that the user wants to face using Pomotux
- **Activity Inventory Sheet AIS** – the general activity container. Contains all the activities registered in the system
- **TODO Today Sheet TTS** – the daily activity container. Contains just the activities that the user will face in the current day
- **Pomodoro** – an indivisible unit of time (normally 25 minutes) used to measure the resource dedicated to an Activity.

1 http://www.pomodorotechnique.com

2 Http://www.xplabs.it

Technology overview

The System has been developed using

- C++ programming language[3]
- QT framework[4]
- SQLite Database library[5]
- LiteSQL Object Relational Mapper framework[6]

Useful tools used during development:

- CXXTEST Testing Framework[7]
- CPPCHECK code analyzer[8]
- Artistic Style code formatter[9]

Roadmap

Even if we followed an Agile methodology, we decided to have a tiny road-map. The following skeleton was used as a guide to help us to decide which user stories were better to be developed at each Scrum Sprint.
We took this decision because the group was made by 3 persons with no experiences with C++. The vision of a plan is a safe rail when dealing with the Unknown.

1. Analysis, Design and Database discussions
2. Playing with SQLite e ListeSQL
3. Implementation of user stories regarding Activities and activity containers such as AIS and TTS. No Pomodoro (timer) are taken into consideration
4. GUI discussion
5. Implementation of the GUI for point 3
6. Pomodoro implementation, hopefully using graphical libraries (i.e. QTimer)
7. Union of 5 and 6
8. Test, bug fixing

3 http://www.possibility.com/Cpp/CppCodingStandard.html
4 http://www.qtsoftware.com/products/
5 http://www.sqlite.org/
6 http://apps.sourceforge.net/trac/litesql/
7 http://cxxtest.tigris.org/
8 http://cppcheck.wiki.sourceforge.net/
9 http://astyle.sourceforge.net/

Software Engineering Outcomes

In the next sections we are going to present some of the documents produced during analysis and design phases

Analysis

User Stories

This section contains the User Stories of the System

1) **Pomodoro Length**
By changing a value, user decides how long a pomodoro should last

2) **Inspect the "Activity Inventory Sheet"**
User can see the list of the activities he should do

3) **Insert an activity**
A user can insert an activity in the "Activity Inventory Sheet", as the activity comes up

4) **Modify an activity**
A user can edit an activity from the "Activity Inventory Sheet"

5) **Delete an activity**
A user can delete an activity in the "Activity Inventory Sheet"

6) **Inspect the "To Do Today sheet"**
User can inspect the list of the activities he should perform today

7) **Fill the "To Do Today sheet"**
The user can move an activity from the "Activity Inventory Sheet" to the "To Do Today Sheet"

8) **Prioritize the "To Do Today" sheet**
User can prioritize the activities of the "To Do Today Sheet", as they must be done sequentially

9) **Start a Pomodoro**
User begins/continues the first available activity of the "To Do Today Sheet". The timer starts and shows up

10) **Break a Pomodoro**
User decides to interrupt the current Pomodoro

11) **Finish a Pomodoro**
The pomodoro finishes, user is alerted to make a break

12) **Finish an activity**
User decides that his activity is finished, the current activity is show as finished and marked as finished in the "Activity Inventory Sheet", the next activity is selected

13) **Postpone an activity**
User can postpone a task of the "To Do Today Sheet". This does not remove it from the "Activity Inventory Sheet", as it must be done on another day

14) **Long Break**
After 4 completed pomodoro, user is alerted to make a long break

15) **Number Internal Interruptions**
User sees how many internal interruptions he faced during a pomodoro

16) **Register Internal Interruption**
User can quickly write the interruption in the Activity Inventory Sheet, to turn it in a activity

17) **Number External Interruptions**
User sees how many internal interruptions he faced during a pomodoro

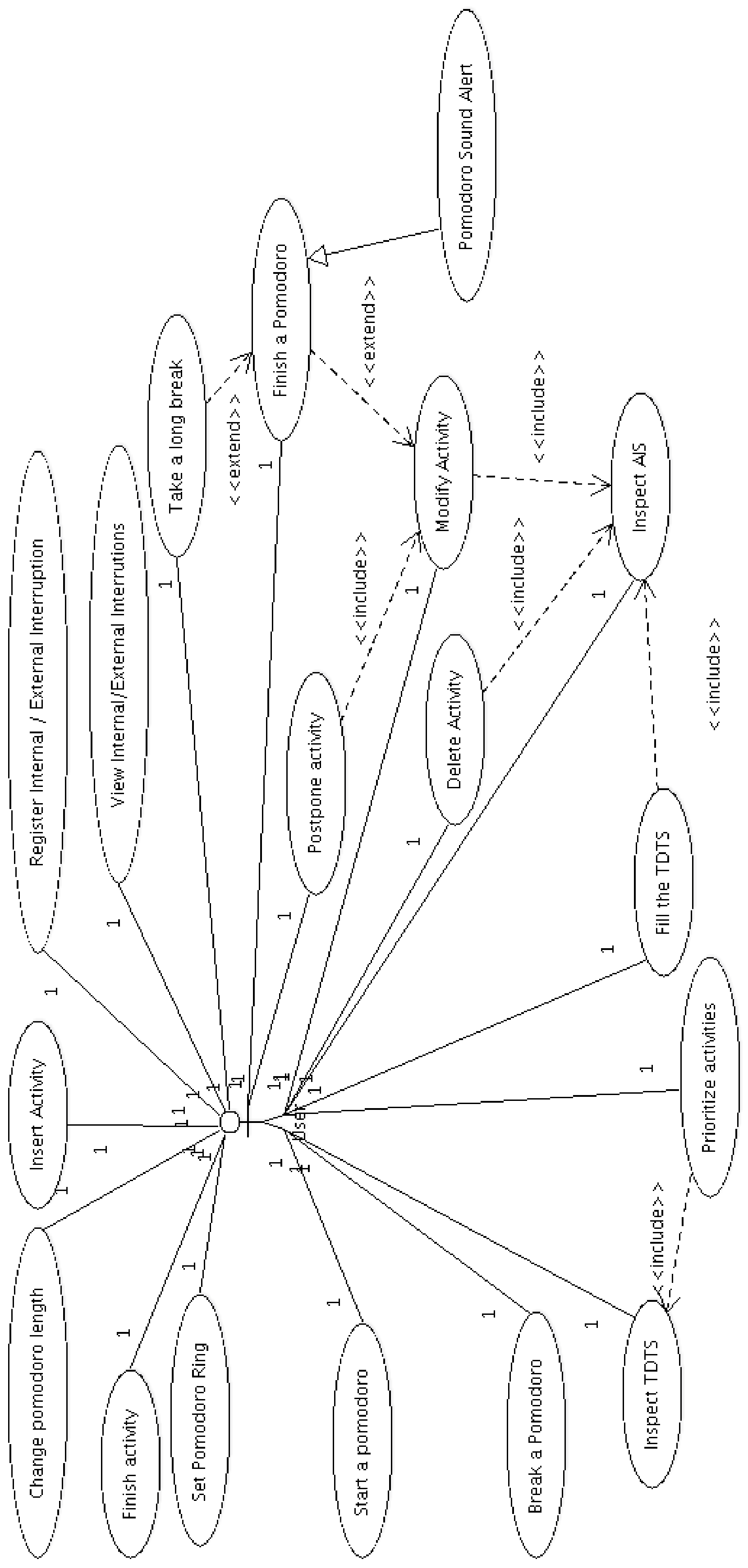
18) **Register External Interruption**
User can quickly write the external interruption in the Activity Inventory Sheet, to turn it in a activity

19) **Pomodoro Ring**
When a pomodoro finishes, a sound is played

20) **Set Pomodoro Ring**
User can decide the sound to be heard when a pomodoro finishes

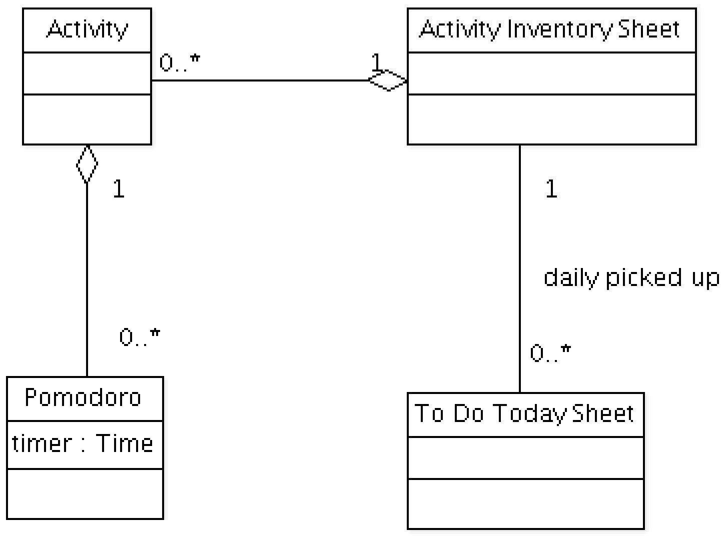
Use Case Diagram

From the User Stories we derived the following Use Case diagram



Class Diagram

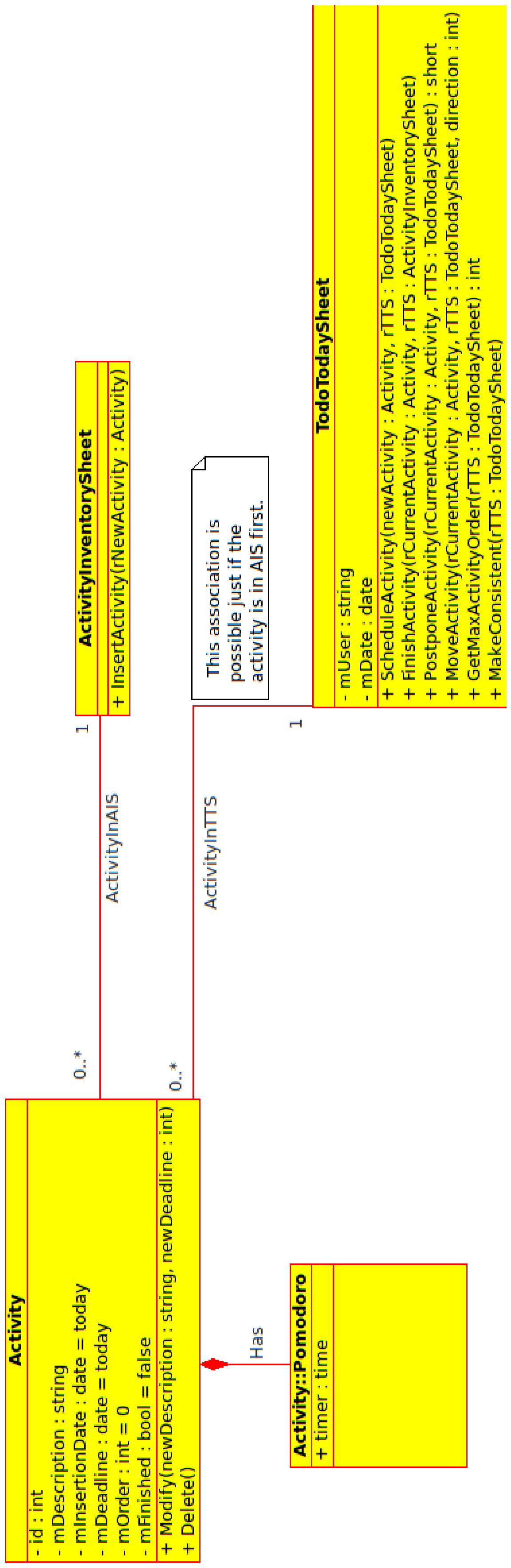
Noun Extraction Technique lent us to this Analysis Class Diagram:



Design

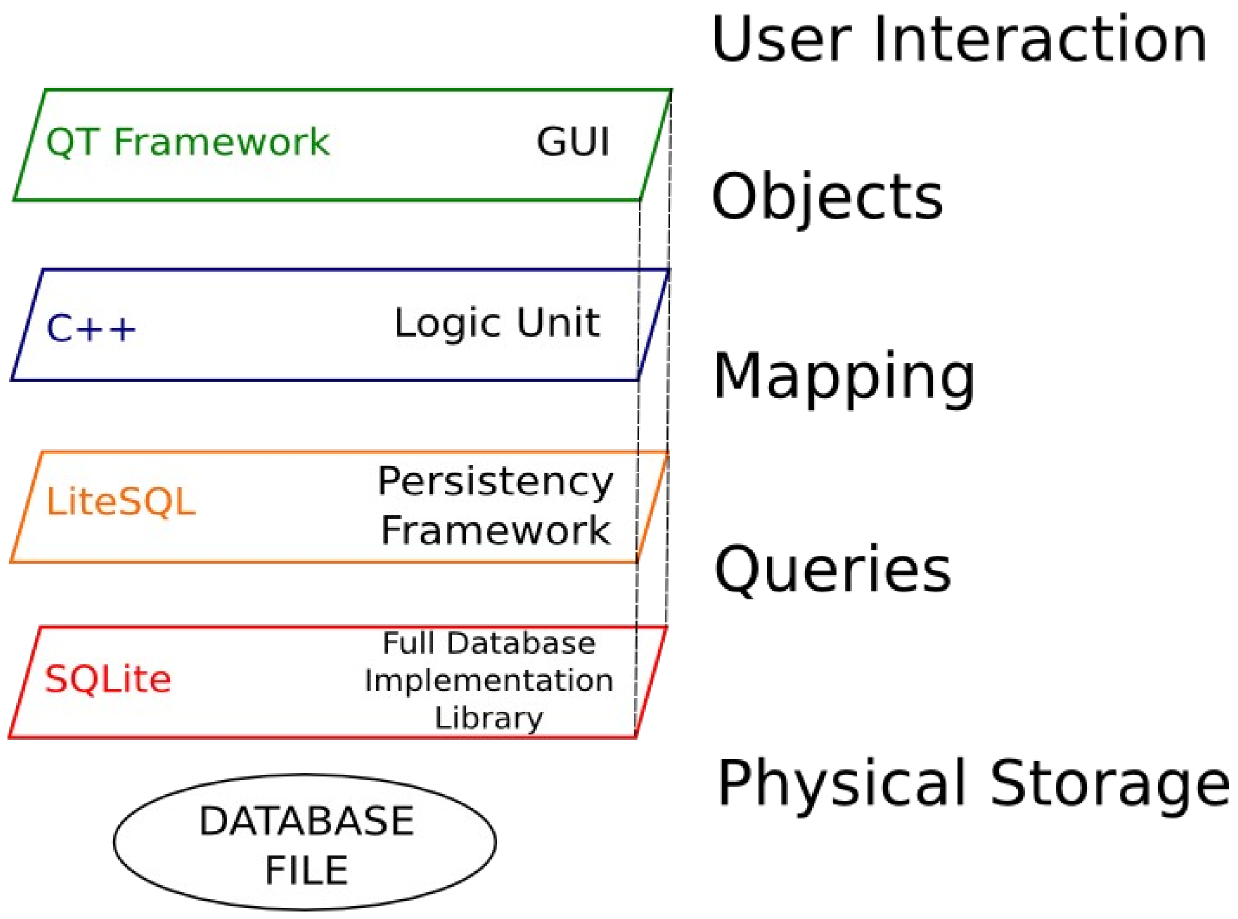
Class Diagram

The following is the complete class diagram of the logic unit of Pomotux:


ActivityInventorySheet
+ InsertActivity(rNewActivity : Activity)
TodoTodaySheet
- mUser : string
- mDate : date
+ ScheduleActivity(newActivity : Activity, rTTS : TodoTodaySheet)
+ FinishActivity(rCurrentActivity : Activity, rTTS : ActivityInventorySheet)
+ PostponeActivity(rCurrentActivity : Activity, rTTS : TodoTodaySheet) : short
+ MoveActivity(rCurrentActivity : Activity, rTTS : TodoTodaySheet, direction : int)
+ GetMaxActivityOrder(rTTS : TodoTodaySheet) : int
+ MakeConsistent(rTTS : TodoTodaySheet)
This association is possible just if the activity is in AIS first.
1
1
ActivityInAIS
ActivityInTTS
0..*
0..*
Activity
- id : int
- mDescription : string
- mInsertionDate : date = today
- mDeadline : date = today
- mOrder : int = 0
- mFinished : bool = false
+ Modify(newDescription : string, newDeadline : int)
+ Delete()
Has
Activity::Pomodoro
+ timer : time

System Layers

This section shows a model for the system layers. We rely on several different technologies to achieve our goal and perform secure information hiding. Only the last layer, the GUI, is the one with which the user interacts with.



The Queries Layer: SQLite

Because of the nature of the entities involved in our Domain, we decided to have a persistent and dynamic data structure in which selection, ordering, insertion, deletion, filtering of the information is easy to be implemented and fast. We also had the need to have our objects (activities) persistent. Those motivations led us to focus on database tools. The best solution was to use SQLite, because:

It is a self-contained, server-less, zero-configuration, transactional SQL database engine. => lower dependencies, no need to install or configure a database server

- The code for SQLite is in the public domain => no license problems
- The whole database is contained in a single file => high portability and data backup

We store the database in a file called pomotux.db, under the directory */home/user/.pomotux*

The mapping layer: LiteSQL

The choose of SQLite pointed out a secondary problem, which we decided to solve before going to implementation phase: How could three (very) busy bachelor computer science students implement in a couple of months a graphical program in a language in which they don't have experience (C++), with the addition of the complexity of databases and queries? We looked for a ORM[10] solution using C++, to let us save objects in database with instructions as simple as *object.save()*. The best solution over the very few available was LiteSQL[11]. LiteSQL is a very young opensource project that aims to become the leading ORM Framework for C++. From their website:

LiteSQL is a C++ library that integrates C++ objects tightly to relational database and thus **provides an object persistence layer**. LiteSQL supports SQLite3, PostgreSQL and MySQL as backends. LiteSQL creates tables, indexes and sequences to database and upgrades schema when needed. In addition to object persistence, LiteSQL provides object relations which can be used to model basic OO building blocks (aggregation, composition, association). Objects can be selected, filtered and ordered using template- and class-based API with type checking at compile

Pros and Features of LiteSQL

- Free license (BSD) => very permissive and compatible with the project and the other technologies
- SQLite3, PostgreSQL and MySQL - backend support => Sqlite support was what we were looking for
- C++ object persistence (store, update, retrieve)
- relational operations (filtering, ordering, referencing other objects)
- automatic database structure maintenance (creates, updates and drops tables/indices behind the scenes)
- C++ template based database API -> no SQL queries by hand => all of our problems are solved by these points.

Cons of LiteSQL

- LiteSQL 0.3.3 was released on 2008-11-12 and is currently declared as a proof-of-concept release. However, we found it stable and mature enough for our scope
- We are not sure whether we will find serious bugs during development, but LiteSQL inventor is very interested in feedback
- Documentation is very poor. There are just a couple of examples online. But the (very) little community behind the project is willing to help via mailing list
- Memory Overhead Extra variables in each Persistent object instance:
  - bool modified - flag in each field
  - bool oldID - flag (used to track id - changes)
  - bool inDatabase - flag
  - Database* - pointer
- Performance Overhead

10 http://en.wikipedia.org/wiki/Object-relational_mapping
11 http://apps.sourceforge.net/trac/litesql/

  - one INSERT-SQL statement is required per inherited Persistent-class (deep class inheritance hierarchies cost)
  - selection of objects needs joins
- Code Size Overhead
  Code generator (litesql-gen) generates a lot of code. In addition to that, a part of that code uses templates which will produce even more code. This is rarely a problem but worth to note when memory is tight.

LiteSQL Implementation Details

There are three major phases when using LiteSQL in a C++ project:

Entity/Relation Definition Phase

LiteSQL uses a XML database definition file which is used to generate Persistent and Relation classes. The database definition is very intuitive and easy, here is a section of our first version of pomotuxdatabase.xml:

```
<?xml version="1.0"?>
<!DOCTYPE database SYSTEM "litesql.dtd">
<database name="PomotuxDatabase?" namespace="pomotuxdatabase">
        <!-- maps Activity objects into database -->
        <object name="Activity">
                <field name="mDescription" type="string"></field>
                <field name="mInsertionDate" type="date"></field>
                <field name="mDeadLine" type="date"></field>
                <field name="mNumPomodoro" type="integer" default="0"></field>
                <field name="mIsFinished" type="boolean" default="true"></field>
        </object>
        <!-- maps Activity Inventory Sheet objects into database -->
        <object name="ActivityInventorySheet?"></object>
        <!-- creates a many-to-one relationship between Activity and AIS -->
        <relation name="InsertActivity?">
                <relate object="Activity" limit="many"></relate>
                <relate object="ActivityInventorySheet?" limit="one"></relate>
        </relation>
</database>
```

Code Generation Phase

The utility **litesql-gen** parses pomotuxdatabase.xml file and generates two files: pomotuxdatabase.hpp and pomotuxdatabase.cpp. The first contains class declarations for persistent objects and relations, the second contains implementation of methods and lots of static data. Fields of multiple Persistent-classes are stored in separate tables. Each Persistent-class has its own table. Also, each relation is stored in separate table.

Implementation Phase

In this phase the ordinary development of the project continues, with the inclusion of the generated files. Now many operations are performed with single lines of code, such as:

Creation/Maintenance of the database

```
// using SQLite3 as backend, db now represents the database

PomotuxDatabase? db("sqlite3", "database=pomotux.db");

// create tables, sequences and indexes

if(db.needsUpgrade())

   db.upgrade();
```

Creation of Activity/**ActivityInventorySheet**? objects:

```
/* creation of one Activity Inventory Sheet */
ActivityInventorySheet? ais(db);
/* make it persistent */
ais.update();
```

```
/* creation of an activity */
Activity at(db);
at.mDescription = "A dummy Activity";
time_t today;
today = time (NULL);
at.mInsertionDate = (int) today;
at.mDeadLine = today;
at.mIsFinished = true;
at.mNumPomodoro = 15;
at.update();
```

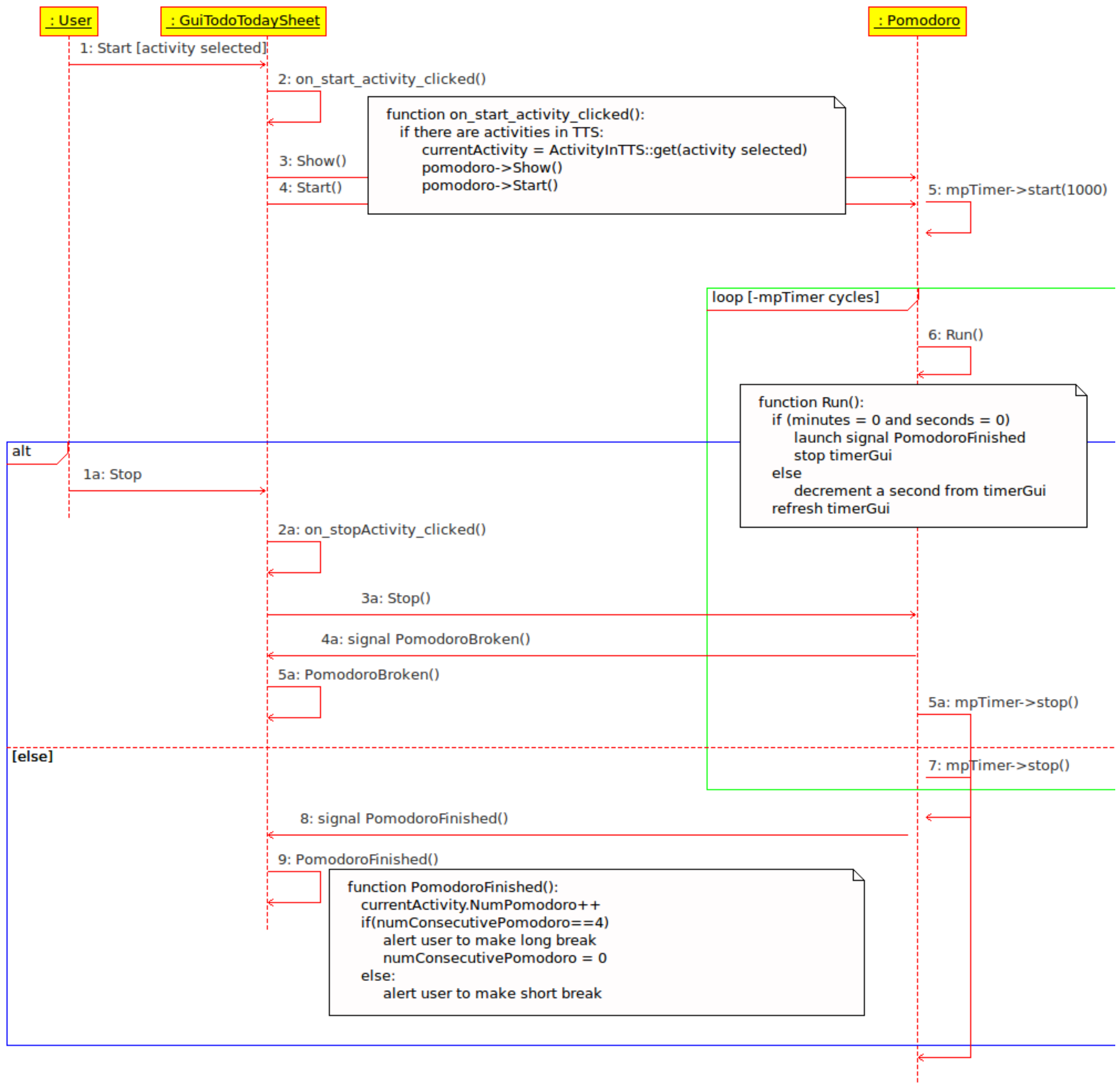
Perform relations between objects:

```
/* activity linked in AIS */

InsertActivity?::link(db,at,ais);
```

Example about layer interactions: Face an Activity

The following sequence diagram shows what happens and how the different layers interact when a user faces an activity using Pomotux



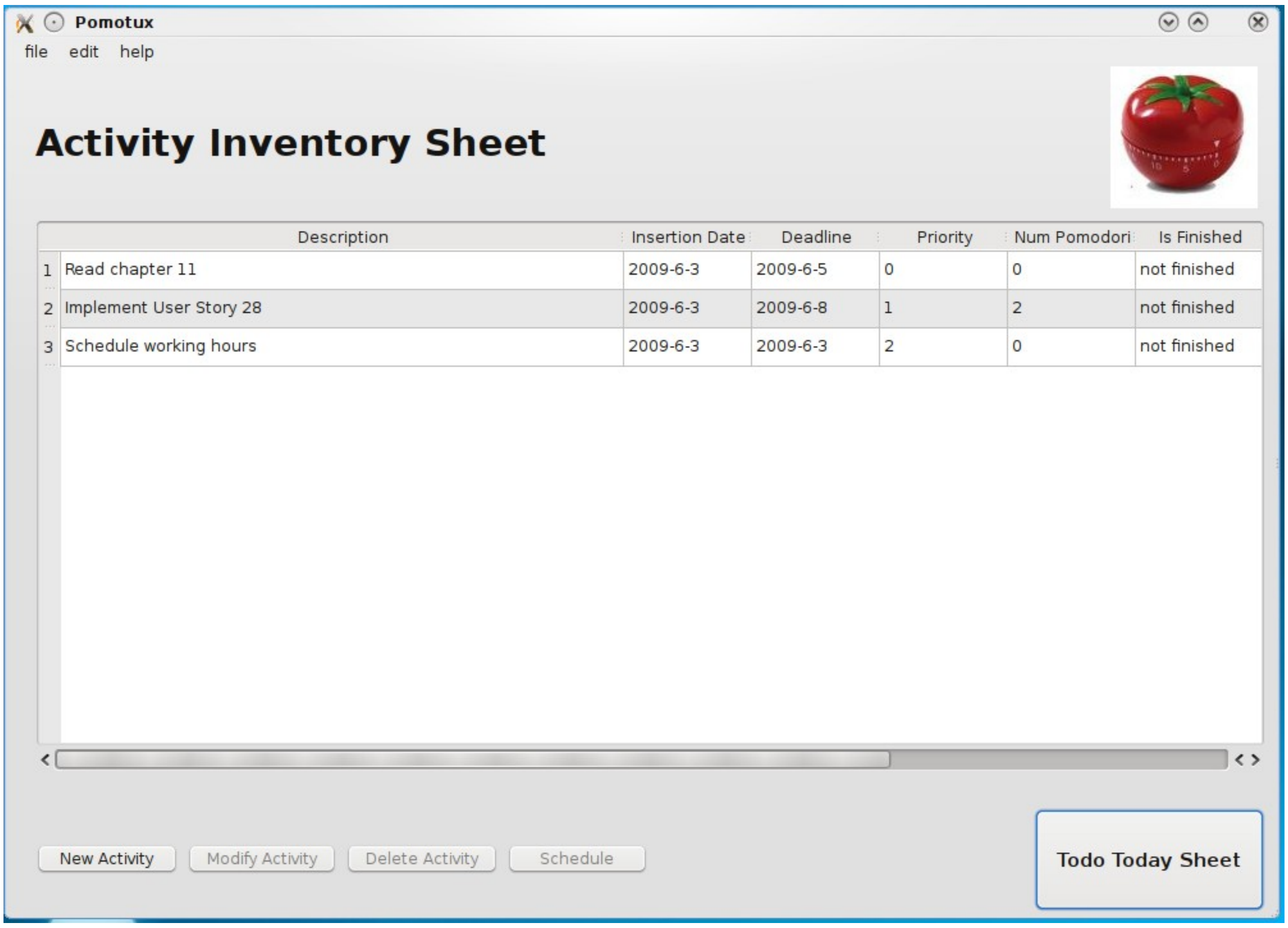
```
function on_start_activity_clicked():
  if there are activities in TTS:
      currentActivity = ActivityInTTS::get(activity selected)
      pomodoro->Show()
      pomodoro->Start()
```

```
function Run():
  if (minutes = 0 and seconds = 0)
      launch signal PomodoroFinished
      stop timerGui
  else
      decrement a second from timerGui
  refresh timerGui
```

```
function PomodoroFinished():
  currentActivity.NumPomodoro++
  if(numConsecutivePomodoro==4)
      alert user to make long break
      numConsecutivePomodoro = 0
  else:
      alert user to make short break
```

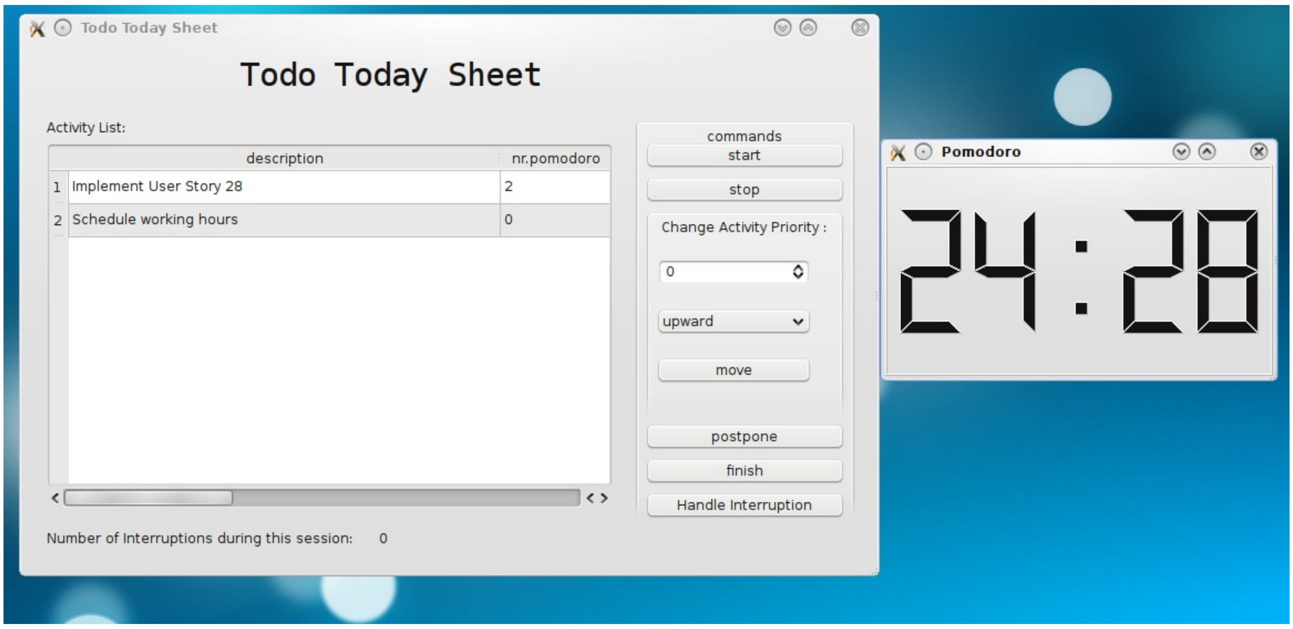
Screenshots

The Activity Inventory Sheet with 3 Activities stored:



	Description	Insertion Date	Deadline	Priority	Num Pomodori	Is Finished
1	Read chapter 11	2009-6-3	2009-6-5	0	0	not finished
2	Implement User Story 28	2009-6-3	2009-6-8	1	2	not finished
3	Schedule working hours	2009-6-3	2009-6-3	2	0	not finished

The Todo Today Sheet and the running Pomodoro



	description	nr.pomodoro
1	Implement User Story 28	2
2	Schedule working hours	0

Number of Interruptions during this session: 0

Problems Encountered

The biggest problems faced during development were caused by a union between our poor knowledge of C++ language and LiteSQL framework.

Version 0.3.3 is the sixth release of the project and there is a very little community behind. It is currently released for non-critical use. However we tested it and believed in it. The framework is not well documented, there are just a couple of examples on their website. We immediately stumbled against C++ templates, that were not documented on their examples. We also had to take a look on their sourcecode to understand how things work.

The lack of documentation brought us to decide a wrong strategy of implementation of the base classes. At the beginning of Sprint 2 we decided to create classes called "Interface Classes" that should extend their corresponding auto-generated classes. This was because everything generated by LiteSQL is declared as "public". That did not work, the framework works well just with its own classes at the moment. At the end of Sprint 2 we took the decision to roll-back to sprint 1, but we lost an entire week of development

At Sprint 4 we implemented some unit-tests using CXXTEST that made us discover a bug of the framework in the retrieval of dates from the database. The bug fix is set for version 0.3.4. The creator of LiteSQL assured us that 0.3.4 will come out by the end of May. However, near the end of Sprint 5, the community of LiteSQL wrote for us a patch for fixing the bug in 0.3.3. The patch is just a workaround that does return the date as a string containing the number of seconds passed from January, 1st 1970 but will be sufficient for us.

As of today (2009-06-04), LiteSQL 0.3.4 beta really came out as the community promised. Pomotux is still based on patched 0.3.3 version, because we don't want the risk of new bugs.

Future

After the end of the course, we will release the program to the general public and ask Francesco Cirillo to support it.

We will surely continue its development and search the support of external coders, to make it an official and mass adopted program. We would be proud to see the program accepted in any Gnu/Linux distribution.

License, Dependencies and Installation

Please read the corresponding COPYING, DEPENDENCIES and INSTALL instructions in the source code folder.

We provide a statically linked version of Pomotux. However, we recommend to compile the source code against shared libraries.

Operating System Support

Pomotux runs under any Gnu/Linux distribution that meets the dependencies, but should run under any *NIX variant with tiny modifications.

Warning: GCC 4.4.* is currently not supported. Please use GCC <= 4.3.* to compile the program.

Windows support is possible but not possible until LiteSQL Windows support.