AI 4 Games
Assessment Milestone 2
Version management

August 26, 2022

Objectives

1. Learn to use and understand the benefits of a system for version control.
2. Review `C++` programming.
   (a) Review the compilation of `C++` modules.
   (b) usage of `makefiles`.
3. Generate documentation from the code
4. The software engineering pattern MVC (Model-View-Controller).

Instructions

This practical laboratory has 4 parts. Each of these parts correspond to an objective. The first discusses software development with tools that support the management of versions of code. The later part includes a discussion of a software pattern heavily used in the architecture of computer games and web applications.

In each part you must complete several steps. We do not provide detailed instructions for many steps. You need to think and understand what you are doing. Copying commands from the text here directly (and blindly) may be insufficient. Specially since this instructions are generic about having access to a server machine where to create a repository.

Nevertheless, you are required to make a learning experience report that shall include screenshoots of the completion of the steps/activities.

A fast introduction to Subversion

Managing the code of a software development project under a version control system is crucial. In fact, as agile methodologies and incremental development are more common, it is important to have a clear trace of which version of the code supports which functionality.

Moreover, with development methods like Test-Driven Development, is even more important to develop functionality incrementally and to regularly commit version of the code where it is know what works (and which tests are green) and what functionality is to be incorporated. Moreover, if the code eventually fails to work, it is easy to revert to an older version or to review recent changes to discover what/where/how a bug has been introduced, or how functionality believed to be working correctly was in fact not fully tested and needs refactoring.

There are 3 popular version systems for developing software and they are all public domain. They have been used repeatedly in industry and are share-ware. These are

1. `cvs` `http://cvs.nongnu.org/`
2. `svn` (subversion) `http://svnbook.red-bean.com/`
3. `git` `http://svnbook.red-bean.com/`

Here we give a quick guide of setting an `svn` server for your project. **Subversion** is a system of version control sometimes called a system for control of configurations (where a configuration is the state of files in a particular machine at a particular time). In such a system, you usually interact with a server that holds the repository The server could be the same machine you work on, however, for example, if you work at University and at home it is convenient to set a server machine you can access from both places (over the Internet). When you have a server that you can access from two places you don't have to carry the project files with you. When you start working in a particular location you do an `update` to obtain the latest version of your files and when you are finished working you do a `commit` to reflect your changes into the repository. The repository works for you as a backup as well and you can always retrieve earlier versions. You would never lose your files, or your earlier versions of your files.

Therefore, such a system allows you to push files with progressive changes into a server (with the command `commit`). Symmetrically, you can reconcile the fresh version in a server into your local copy with the command (`update`) and obtain the latest version that has changes (or updates from other developers when you work within a team of developers). It is even a very useful tool when you work alone.

The complete documentation and guides for **Subversion** is available on line:

```
http://svnbook.red-bean.com/
```

With *Subversion*, you can un-do particular changes, see the differences between a previous version and your current version and you can review who has committed what. In fact, managers in many developing teams can trace the work of the software developers by reviewing who has worked on what. It also assist lecturers in universities monitor the regular work of students on a project. It also allows developing into (*branches*) and to mark some version with (*tags*) when we want to name a particular stable version of the software.

The repository is the central place where changes are recorded. The copies on your working computer (maybe you lap-top or a desktop in a lab) are called *sandboxes* When you are satisfied with a version that fulfills some functionality you should reflect it into the repository with `commit`. When you need the latest version from the repository to start work on some module you need to do an `update`. In fact, you can do this not only with code (programming language files), but you can do it with images, documentation, reports of assignments, etc. That is, you can use it to regularly work on an assignment and progressively complete parts, committing every time a section is completed and improving the previous version into a new one closer to completion every time. Some of this ideas can be achieved by placing files in *the cloud* though the Internet, but many of the *cloud* facilities provide storage space, but little support for version control of the same file.

For working with *subversion* from a command line you use the command `svn` in `POSIX` (systems like MacOS terminal or LINUX terminal) provided that it is installed. However, there are many share-ware GUI tools to use *subversion* (or also `cvs` or `git`). For example `esvn` and `rapidsvn` work in LINUX while in `Windows`, the client `TortoiseSVN` is available. The software development environments like `Eclipse` or `XCode` also support version control systems like `git` and `svn`.

Important points

A system of version control allows you to manage the life-cycle of developing a computer game. It allows you to trace and follow the changes as you develop the code. It enables you to compare versions and you can always go back. You can also put aside and deploy a version you do not want to change any more while evolving other parts.

How doe the system work? You will have a central deposit (think of it like a database of all the versions of all the files you register with the repository). A log will record all changes. You can have many working copies, in one or several machines. You will make changes as you work and you record the changes in the repository without losing previous versions. *Subversion* will keep track of all versions for you. This helps a lot in debugging code as you can easily trace “how did this bug entered my code?”.

Connecting with a server

This instructions are a guideline that assumes we have access to some server in the Internet with IP address

```
193.145.50.179.
```

If you have access to some using a DNS address, this will work as well. Just remember you must have an account so you can login into the server with a command like
`ssh MyAccount@193.145.50.179`.
That is, from home, or from a lab, or from a laptop with Wi-Fi access you can use `ssh` (*secure socket-layer shell*) (from `WINDOWS` you can use a share-ware tool like `putty`).

Verify you can connect.

```
ssh MyAccount@193.145.50.179
```

If everything goes well you would be connected to your remote server. This server must be set up to run the *subversion* server. It is not hard to do this if you want to enable a machine of your own and make it visible in the Internet, but usually your University will set this up.

You close the connection with `exit`.

Creating a repository

Creating a repository is an usual activity; that is, it is not something one does every day. In fact, the repository may be used by s several projects, so it is even rare to create a repository more often than facing a project.

Thus, repositories are usually created once at the beginning of a project (or even less frequently). That is why is common to forget how to do this and one needs to follow manuals and instructions. In is important not to fool around (or modify) configuration files created by the repository and its database. More often than not one destroys the well-order of the repository and it would be your fault, and in this case, you may lose all your files.

Read all instructions before following this steps (or any instructions to set up a repository). Instructions may require adjustment for you particular working environment and it is good to have a clear idea of what you want to achieve. Thus, read this instructions completely and make sure you have a clear idea of is that we want to achieve. If you are not sure about something, read some more and do not start.

The steps

1. Connect to the server machine like before (using `ssh`). The server machine will hold the `svn` repository and we assume the the IP address (or the DNS) equivalent is visible trough the Internet; we do not discuss here if your University uses firewalls and you need to set a VPN (virtual private network). We assume that the server also support `POSIX` commands.

   For example, try the command `ls` to list the files in your account on the server machine. (it may be the case you do not have any). The command `pwd` will reply with your current path, and the command `cd` without any arguments brings you back to your (*home directory*). We assume here some familiarity with UNIX shell commands.

2. We make an initial structure for the repository. This structure is not make inside the repository, but in a temporal folder. We will place it in the repository by importing it into the repository. Then we ignore it (and better, destroy it).

   ```
   mkdir TemporaryFolder
   ```

   This command creates the folder `TemporaryFolder`. Now, let go into this folder.

   ```
   cd TemporaryFolder
   ```

Verify you are inside this folder, you can do this with the command `pwd` to see which is the current path. Inside `TemporaryFolder` issue the command

```
mkdir trunk tags branches
```

This last command creates 3 new folders that constitute the recommended structure to organize your files for using *Subversion*. The working files will be placed in `trunk`. So, let move inside this folder

```
cd trunk
```

We now make a folder to hold `C++` files or other files to build or software.

```
mkdir ProjectPackMan.d
```

Now navigate inside this recently created folder.

```
cd ProjectPackMan.d
```

Using an editor, for example in POSIX, we can use `vi` and construct some code. Open the file `hello.cpp` and place in the following content.

```
/*
   - streams from the standard library
   - namespace std::name
 */
#include <iostream>

int main(void)
{
        std::cout << "Hello World!" << std::endl;
        return 0;
}
```

3. Exit the structure

```
cd ../../../
```

This exits twice to the parent folder. You can achieve the same with

```
cd
```

since this takes you to the `home` directory. Verify this with

```
ls
```

since you should see the folder `TemporaryFolder`.

This is an initial folder structure we will place in the repository (but will not be managed by the repository).

4. We now do create a folder where the repository will reside

```
mkdir IndividualrepositoryFolder
```

5. We now make the *Subversion* command to install all the folders we created and the configuration files to build a repository. We will not need to do anything else, perhaps for a long time.

```
svnadmin create IndividualrepositoryFolder
```

You cans see what *Subversion* has done with

```
ls IndividualrepositoryFolder
```

However, do not touch the files in this folder unless you have become a *Subversion* expert and you are absolutely sure of what you are doing. Even if you want to alter things here it is best to use the administration tools of *Subversion* itself.

6. We incorporate the structure into the repository with a command in one line [1]:

```
svn import TemporaryFolder
file:///home/MyAccount/IndividualrepositoryFolder -m “Initial”
```

7. You can use subversion commands to check the folder is inside the repository with

```
svn list file:///home/MyAccount/IndividualrepositoryFolder/trunk
```

It is very important to realize the initial structure is not under version control although it was used to place folders into the repository. That is, the initial structure is not a local/working copy. Working in it any further is not useful. To have a working copy, it must be extracted first from the repository. *Subversion* only keeps track of files we tell them to keep track of.

8. Since we are finished with the work a t the server, we shall disconnect.

```
exit
```

[1]Remember we are using MyAccount as the name of your account in the server.

Obtaining a sandbox

Now we will obtain a copy out of the repository to work on it. This will be a local copy of the files and it is also an unusual operation, since the first time we create a sandbox for a particular machine (say your laptop), we do not extract again (what would be a frequent operation is to refresh the local copy with its most recent files from the repository, but not to create it). To create a sandbox in the language of `svn` is to perform a `checkout`. This could be done with several client tools but a single POSIX command is as follows.

```
svn checkout

svn+ssh://MyAccount@193.145.50.179/home/MyAccount/IndividualrepositoryFolder/trunk/ProjectPackMan.d
```

This may request your password on the server several times and if all goes well the structure and files created will be reflected in your local file system.

Using la your copy local

1. Lets work a bit with the file `hello.cpp`. Navigate the structure of folders until you find the file and you can open it with an editor. Change the message it prints and make sure it print your name and lastname. No need to compile the program now.

2. To reflect this local changes into the repository use the following command

   ```
   svn commit
   ```

   You may find that this gives you error. It could be because you have not configured a default editor for `svn`. This is because every `commit` needs a message to place in the log and to associate with this new version. Making a commit creates a new version if there are modified files.

   ```
   svn commit -m “Now the program has my name”
   ```

   The option `-m` will place in the log the message. Messages like this help software developers know what is this version about and they must be meaningful so you can search the log for particular states of the repository. Do not put the date (or the time), since the system takes care of that.

   When committing, you are connecting to the server, and you will be asked for your password to connect into the server.

3. What does the command do?

   ```
   rm hello.cpp
   ```

   Why don’t you try it? You will see that the file you just made changes to has disappeared, however, is it lost? It is not gone thanks to the repository, you can recuperate it with the following command.

```
svn update
```

Now you get the last version from the repository and if you inspect the file `hello.cpp`, you wills see is the version that prints your name.

Task

Disconnect from the current machine you are on. Use another computer in the lab or in your home and repeat the exercise of creating (checking out) a local copy in that machine. Then inspect the file `hello.cpp`. See what is the state of the message that it prints. Change the name to an abbreviated version of your name using an editor. Then `commit` this new version. Go back to the machine where you checked out for the first time and do an `update` (NOT A `checkout`). Verify you now have the latest version.

Creating a local copy should be a rather infrequent operation. It may be better to do the check-out also to a USB *flash* memory or to an external disc. That way you also have mobility of the local copy without creating too many.

Populating the local copy and reflecting in the repository

Once you have been working with one *sandbox* you must place the files for your programming tasks in the folders there. For example, create and edit a file named `README` inside the folder for the code. Review the following content and place what is suitable in your case.

Laboratory 2 – a program for modeling objects in PackMan. Copyright (C) 2011, `YOUR NAME`.
Based on an initial framework by Vlad Estivill-Castro / Griffith University
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA
This folder contains the implementation of XXXX

An alternative text may be

```
This program is developed as part of an assignment at XXXXXXX
Based on an initial framework by Vlad Estivill-Castro / Griffith University
All rights regarding the intellectual property generated here are exclusive property of the author under the laws of CITY AND COUNTRY.
```

To add this file under the control of *Subversion* we must do:

```
svn add README
```

You should see a reply like

```
A README
```

This is *Subversion* reporting it will keep a watch on this file from now on, but not that it is already reflected in the repository. To actually place it in the repository, we must issue the next command.

```
svn commit -m "Description of intellectual rights of my program"
```

You should notice that the version number increases.

Obtaining information

Edit the file `README` once more to include todays date. To see the state of the files in a folder you use `svn status`. Just try it:

```
svn status
```

The response should look like

```
M README
```

This means that the file `README` has been modified and the version currently in our *sandbox* is different of what is in the repository. What is the difference? you can find out with the following command

```
svn diff README
```

Besides the *subversion* command `status` that gives a status report, we can find the trace of updates. For example, the command `log` gives a summary of the revision that have been placed in the repository. The report includes information of who made changes and the comment that was used to commit the changes. Make a `commit` and follow it with :

```
svn log
```

This time the answer may be short and very useful as we only been using the repository today, however, you will soon find it very useful.

Other interesting commands

1. `svn help`: The system provides assistance using this command.
2. `svn` *`subcommand`* `help`: Provides help regarding the command provided as an argument.
3. `svn mkdir`: Creates a folder/directory under version control (you still need to make `commit` to reflect the change in the repository)
4. `svn del`: remove from file watch
5. `svn move`: rename a file and keep watch under the new name but the repository maintains the history
6. `svn merge`: combine local changes and changes in the repository
7. `svn copy`: copy a file and keep the new copy also under watch

Review the manuals and all information you can with *Subversion*. Practice with the regular cycle will eventually feel natural. With 20% or less of the commands you will get much use of `svn`.

Remember, with *subversion* you can have several working copies, in the computers at university, in your home, in your external drives or USB drives, and they can all be synchronized with `svn`. No longer sending a version by e-mail to yourself for backup and soon you can learn to actually share a repository with a team of other software developers. Remember, to start work get a fresh copy with `update`, and to save your changes so far, do `svn commit`.

How to undo your local changes? `revert`.

If you just want to revert a single file

```
svn revert filename
```

If by accident you destroy or delete a file under version control, we have seen how to recover it. With `svn`, nothing serious has happened yet:

```
svn update
```

and you recover your file. How to get back to the previous version? If you want to go back to a specific version, say 178, (all files in the folder will be reverted accordingly)

```
svn merge -rHEAD:178 directory
```

but if you want a specific file only

```
svn merge -rHEAD:178 filename
```

What shall you submit?

A report of your learning experience with screen-captures of completed tasks.

Reviewing compilation

Copy the files in the folder `PackMan.d`; Vlad Estivill-Castro offers no warranties on the framework provided. This is purely a pedagogic exercise and the software provided has no other purpose. Study the classes `ListBoardObjects` (it is defining in the corresponding two files `ListBoardObjectsH.h` and `ListBoardObjects.cpp`). Develop a class `VectorBoardObjects`.
This class should provide the same functionality but it enables storing a maximum of 4 objects in the collection.

Develop a main program that test all the methods of your new classes `monster` and `bomb`. It also verifies the working order of your class `VectorBoardObjects`. That is, your main program can create objects of the different classes (for example `Player`, `Wall`, `Bomb`) and store them and retrieve them from the vector. However, the interface shall be the same as for the list. So it should be possible to switch back and forward between the list implementation and the vector implementation with little localized change.

The interface at the moment has these properties.

1. The constructor in `ListBoardObjects` build a list with at least on background object (this lists are never empty).
2. `fitrstObject` supplies the first object in the collection.
3. `nextObject` supplies the next object after a cursor and advances the cursor. Return `NULL` is no object exists and only works after the function `firstObject` has been called [2].
4. `addObject` adds an object to the collection.
5. `elimiantePlayer` removes a player object from the collection. What are the post-conditions of this method? What happens if there are no players in the collection? Write down the pre-conditions.
6. `isThereAWall?` allows to know if there is a wall in the collection.
7. `cookieValue` allows to know the value of a cookie in the collection. What are the pre-conditions of this method? What are the post-conditions? How is a cookie removed from the collection?

Make sure you can perform a *Unit Testing* (`http://en.wikipedia.org/wiki/Unit_test`) of your implementation of your collection based on vectors..

What shall be submitted?

All the classes of your new program and a description of the testing performed on each of them. If possible, evidence you develop-them incrementally and using a system for version control.

[2]Explain the pre-condition and post-conditions of these methods. Visit the web site `http://en.wikipedia.org/wiki/Design_by_contract` and find definitions for the following terms: *programming by contract*, *pre-condition*, *post-condition*, *invariant*.

Patterns

Design patterns offer a solution to a specific problem that commonly emerges in software design. Patterns are helpful in achieving design principles like low coupling and high cohesion. They describe how to design or use classes and how to assign responsibilities to classes so the principles are followed. A design pattern usually translates to class definitions and code and it is a template of the roles played by each class. They have been developed over lots of experience in many different situations (`http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29`).

The design patterns are usually object-oriented and they display the relationships and the interactions between classes and objects. Typically they do not describe algorithms.

The class `ListBoardObject` provides a simpler interface than the large family of methods usually available in the standard classes for collection in languages like `C++` or *Objective-C*. Review the pattern *Facade* (`http://en.wikipedia.org/wiki/Facade_pattern`) and describe in what way does the class `ListBoardObject` follows this pattern. Investigate if there is any relationship with the pattern *iterator*?
(`http://en.wikipedia.org/wiki/Iterator_pattern`) Use class diagram in UML to describe the design patterns you find in this practical laboratory.

The pattern MVC

The Model View Controller (MVC)
(`http://en.wikipedia.org/wiki/Model%E2%80%93View%E2%80%93Controller`) is an architectural design pattern of software engineering that enables to separate the responsibilities of the domain logic from the responsibilities of the graphical user interface, and also from the logic of maintaining a model. It maintains a clear interface between the GUI and the model and uses the controller to update the model when users generate events or even to update the GUI when the model changes due to internal events.

That is, the model separates the data of an application, the interface with the user and the control logic into different component (each component may be made of one or of many classes). This style of request and response is frequent in Web application where the view is usually HTML pages and the model is a back end database model.

This architecture improves development, but more importantly, it enables for easier software maintenance. The architecture of a computer game organized this way enables the modification of the user interface while still reusing the model and the controller. There are several implementation of the MVC patter, but usually the flow of interaction is as follows:

1. Users interact with the interface, and they generate a request.

2. The controller captures such request from the user interface and associates an action that is suitable to apply to the model.

3. The controller notifies the model of the action and probably observes a change of state in the model that corresponds to a response for the requested action.

4. The controller reviews and makes request to the user interface to update is presentation in accordance with the new state of the model.

Task — The pattern MVC

Review carefully the code provided. Produce a UML class diagram of all the classes involved. That is, review the code in the folder `PackMan`. Make sure you can compile the code, run it. Make sure you can make an execution that consumes all the cookies. Run it with a debugger and follow what function c all what method. Once you have an understanding of the working of the program develop a detailed UML diagram for all classes involved.

The program has a very simple interface. It uses the black-white terminal as a screen and the keyboard as input input device. However, it should be possible for you to group the classes in your UML diagram into 3 components. Label them appropriately as the View, the Model and the Controller. Note the traffic of messages between the classes and discover who acts as the interface (the door) to the model, the controller and the view.

Make a sequence diagram to illustrate the dynamic behavior of the program, form the user's input of a request to move the player, to the token in the game moving and consuming a cookie, then back to the update of a new state of the game in the screen.

You must submit these two diagrams.

Documentation

Documentation of program code is fundamental for its use (evolution and re-use). It is also crucial during the development of a software application like a computer game. It is also laborious and error prone to develop documentation after the code is working. It is much better to produce documentation as the code evolves and is constructed. It is in fact better to develop the documentation ahead, as a contract description of what the method or class responsibility is. What can the code do (or will do), what are the pre-conditions and the post-conditions. Document what is the contract you are trying to fulfill with an implementation. Do this with comments on your code even if you do not have the code. Then, we discuss here using tools that extract the documentation from this comment and your code. That w ay is also easy to keep the documentation up-to-date.

The code provided has almost no documentation. Your task is to insert such documentation and to use the tool `doxygen` to generate the documentation from your comments (`http://www.stack.nl/~dimitri/doxygen/`).

You can find a simple tutprial about `doxygen` and how ti builds the documentation in

```
http://www-scf.usc.edu/~peterchd/doxygen/
```

With POSIX, you can automate the task of generating documentation by defining a corresponding target in the `Makefile` of your project. Make a *target* named `doc`. An important point is that `doxygen` requires a configuration file named `Doxyfile`. Thus, the first time you shall issue the command

```
doxygen -g
```

With this you obtain a default configuration file you can edit. Do not forget to do the command

```
    svn add Doxyfile
```

This places the configuration file of *doxygen* as part of the files `svn` keeps track of.

In the configuration file `Doxyfile` we recommend you review and update the section that define values of the following. `PROJECT_NAME`. We also recommend you set to `YES` the options `JAVADOC_AUTOBRIEF` and `EXTRACT_ALL`.

When completing the editing of the configuration file place this fresh version in the repository with

```
    svn commit
```