

# Programming for Chemical and Life Science Informatics

I573 - Week 1

Rajarshi Guha

15<sup>th</sup> January, 2009

# Know Your Toolset

- Cheminformatics and bioinformatics are primary computational in nature
  - ▶ They do depend on experimental data
- Just as an experimentalist, you have tools and techniques
- Consider computer programs and algorithms to be *computational experiments*
- To properly design an experiment and interpret its results, you should know the details of your tools and methods
- A wet chemist should know how IR spectra are generated, or the basic theory behind chromatography

# Know Your Toolset

- What are the tools for a computational scientist?
  - ▶ Languages and their compilers
  - ▶ Operating environments (shells, IDE's)
  - ▶ Data analysis and visualization tools
  - ▶ Algorithms

# Know Your Toolset

- You don't have to be a compiler expert
  - ▶ A wet chemist does not need to know the internal electronics of a spectrometer
  - ▶ You (probably) don't need to know the details of RTL or trampolining in gcc
- Similarly, you don't need to go into deep mathematical details of algorithms
  - ▶ Knowledge of it certainly helps
  - ▶ Should know when to use a hash table versus an array structure
- Certain languages have features that you should be aware of
  - ▶ In Java, Hashtable and HashMap are slightly different

# How Much Do You Need to Know?

- Loaded question!
- Fundamentally, you need to know how to program
- This is a conceptual issue and is independent of any specific language or program
- If you know how to program in language X, you will easily pick up language Y
  - ▶ Generally true
  - ▶ Going from procedural language to **functional language** is tricky
  - ▶ So, C → Lisp might take some effort
- Collecting language skills may or may not be useful

## How Much Do You Need to Know?

- Good to have a command of
  - ▶ Scripting language (Python, Perl, Ruby)
  - ▶ Low level language (C, C++)
  - ▶ High level language (Java, C#)
- Depends on
  - ▶ Nature of the project
  - ▶ Availability of libraries
  - ▶ Pre-existing code
- With the growth in web apps, knowing Javascript is pretty useful

# Know Your Environment

- Either Unix or Windows, preferably both
- Should be able to set up your environment and manipulate it
- The Unix command line is a life saver
  - ▶ Knowledge of grep, awk, sed, cut, sort etc can convert an hour or two of writing code to 10 minutes to construct a pipeline
- Windows can also support a Unix environment via [Cygwin](#)
- Windows provides its own shell called [Powershell](#)

Do not think that the editor or IDE is the entire environment

## Two Classes of Tools

- Server side
  - ▶ Usually set up by the sysadmin for use by multiple people
  - ▶ Good to be familiar with these tools
  - ▶ Very useful even if you're the only one working on the project
- Client side
  - ▶ Your daily tools
  - ▶ Should know these tools very well
  - ▶ Significantly improves your productivity
- A good set of tools can make your programming much more efficient and enjoyable

# Outline

1 Server side tools

2 Client side tools

# Version Control

- Version control is vital for any non-trivial program
- Essentially a way to keep track of changes to a document
- Applicable to many documents
  - ▶ Source code
  - ▶ Text documents (papers etc)
  - ▶ Images
- Useful features
  - ▶ Revert to an older version, if the new version is not working
  - ▶ Create branches to try out new approaches
  - ▶ Merge contributions from different people
  - ▶ Keep track of the history of modifications
- Can do all this with multiple copies of a file and a README
  - ▶ Very inefficient and tedious
  - ▶ Better ways have been available since 1972!

# Version Control

## Modern VC systems

- Many different systems are available. See [here](#) for a list
- Both open source and commercial systems are available
- Two types of version control systems
  - ▶ Centralized (SVN, CVS)
  - ▶ Distributed (Darcs, BitKeeper, Mercurial, Git)

# Types of VCS

## Centralized

- Consists of a single, centrally located, repository
- All developers checkout and commit to this repository
- Requires network access
- Simple to understand

## Distributed

- Every developer has their own local repository
- No specific central repository
- A small group of developers decide which branches to merge
- Does not require constant network access
- More complex than a centralized VCS, requires more effort to learn

# VCS Terminology

- **Trunk** - usually, the main line of development
- **Checkout** - creates a local working copy from the repository.
- **Committ** - occurs when a copy of the changes made to the working copy is written or merged into the repository
- **Branch** - a set of files are branched resulting in two sets of files which can be developed independently. Branches can be merged
- **Merge** - combine multiple sets of changes to a file (or set of files)
- **Conflict** - occurs when two users change the same document, and the changes cannot be merged automatically

# Subversion (SVN)

- I will provide a Subversion repository
  - ▶ [SVN manual](#)
  - ▶ [Cheatsheet](#)
  - ▶ [Instructions](#) for setting up an SVN repository on Linux
- If you haven't used it before, you can play around with it
- It maybe useful when developing your project
- Everybody should make their own directory (use your *cheminfo* username) and work in there
- Note that everybody can view everybodys directories - so respect other peoples work
- I'll focus on the Unix command line, but you can integrate SVN with Windows Explorer via [TortoiseSVN](#)

# Web Server

- A web server is very handy to have on your local machine
- A variety of uses
  - ▶ Provide your own SVN repository
  - ▶ Set up a wiki
  - ▶ Testing web based apps
- [Apache](#) is a well known open source web server
- On Linux machines it's usually pre-installed; if not, quite easy to install using yum, apt-get etc

# Web Server

- Apache is highly configurable - won't go into details
- Have to be careful not make holes in your installation
- The default configuration is pretty safe

# Bug Tracking

- Software development will involve bugs and feature requests
- Need a good way to keep track of them
- Ideally we'd like to assign them to the appropriate person
- Maybe also keep track of progress in fixing the bug

# Bug Tracking

- A number of solutions are available
  - ▶ Mantis
  - ▶ Bugzilla
  - ▶ Jira
  - ▶ Trac
- Bugzilla is a widely used product (Sourceforge is a good example)
- Installation is complex, but well documented
- Support multiple backend databases

# Outline

1 Server side tools

2 Client side tools

# Subversion

- First step is to check out the repository
  - ▶ `svn co http://rguha.ath.cx/svn/i573`
  - ▶ Creates a directory called `i573`
- Make your own directory in there
  - ▶ `svn mkdir rguha`
  - ▶ This assumes that you want the directory to be stored in the repository
- Committ any changes you've made
  - ▶ `svn ci -m 'a message to indicate the changes you made'`
  - ▶ Must provide a message
  - ▶ Good practise to give a short, meaningful message

# Subversion

## Revisions

- A commit results in a new *revision*
- A commit to any file or directory in the repo results in a new revision for the *whole* repository
- This is different from CVS
- If X makes a change in `xdir/somefile.c` and commits, the revision of the repo goes up by 1
- If Y then makes a change to a different file and commits the revision of the repo will go up by 1

# Subversion

- Once you've made your directory you can start adding files to the repository
- Say you're going to write a Java program called `Hello.java`
- Change to your own directory and create the file in your editor
- At this point, we must *add* it to the repo
  - ▶ `svn add Hello.java`
  - ▶ This has not put it in the repository yet
  - ▶ It's just noted in your working copy
- As before, since our working copy has changed, we should commit any changes, so they go into the repository
  - ▶ `svn ci -m 'Added the first version of Hello.java'`

# Subversion

- If somebody else was working on your project they could do an `svn update` to get any changes you made to the repository
- A nice side effect of this is that you can work with your repository on multiple machines in multiple locations
  - ▶ Just needs a network connection
  - ▶ Remember to commit changes on the different machines
- To see the messages for each commit do `svn log`
  - ▶ This will list the commits for the directory you are in
  - ▶ If you're interested in a single file, add the filename at the end of the command
- `svn status` is a very useful command and shows you the current state of a file or directory

# Subversion

- To see the actual changes in your working copy wrt the repository do `svn diff`
  - ▶ Will show changes for all files in your directory
  - ▶ Shows changes in unified diff format
- You can also view changes between specific revisions of a file

# The Unix Command Line

- Very useful to have experience in
- Even if you work on Windows, Cygwin provides a Unix toolchain
- The power of the CLI is to create pipelines of simple commands
- ▶ `grep -A 2 CID pubchem.sdf | grep`  
`'[0-9]'GetsthecompoundID'sfromaPubchemcompoundfile`
- A bit of shell scripting makes life easier
- No need to write a full fledged program to extract columns or reformat data

# Editors

- A source of religious wars!
- A good editor is a huge help
- Notepad is an editor but is not useful for programs more than 5 lines long
- On Unix
  - ▶ Vi, Emacs
- On Windows
  - ▶ Crimson, ConTEXT
- These are general purpose editors
- Some important features that you want while programming
  - ▶ Syntax coloring
  - ▶ On the fly suggestions (handy, not necessary)
  - ▶ Build support

# Editors

- Vi and Emacs are well known Unix editors
- Also available for Windows
- Each have large amounts of functionality
- Hence, have steep learning curves
- Worth your time to learn how to use one of them well

# Integrated Development Environments

- IDE's go beyond simple text editors and provide multiple features for programming
  - ▶ Code support
  - ▶ Build and deployment support
  - ▶ Debugging features
- Note that Vi and Emacs can be considered IDE's
- In many cases IDE's are specific to a language(s)

# Integrated Development Environments

- If you're writing in Java, you need something more than Emacs
- Eclipse, Netbeans, IDEA are very good editors
- All of these have very useful features
  - ▶ Syntax coloring
  - ▶ Autocomplete
  - ▶ Refactoring
  - ▶ Good integration with VCS
- Very handy for long Java class names and packages

# Integrated Development Environments

- Eclipse also supports C, C++, Python, Ruby, PHP, R
- Other popular IDE's include
  - ▶ Visual Studio (Windows)
  - ▶ [Anjuta](#) (Linux)
  - ▶ [KDevelop](#) (Linux)
  - ▶ [XCode](#) (Mac)

# Collaboration

- Many programming projects are collaborative
- Especially true for larger projects
- Open source projects are generally collaborative
- A number of issues arise in such projects
  - ▶ Communication and discussion
  - ▶ Source code and bug management
  - ▶ Personnel management
- I'll focus on communication
- For people handling issues see this [video](#)

# Collaboration

## Communication

- Mailing lists are probably the most common forms of group communication
- Extremely useful resources when archived
- IRC is also a popular form of communication
  - ▶ Text based, real time conversations
  - ▶ Conversations can be logged
  - ▶ Useful for collaborative debugging
- Modern chat programs (MSN, Yahoo Chat etc) perform a similar function

# Collaboration

## Communication

- Video conferencing is in vogue
- Handy, but does not necessarily add more to text based chat
- Can be bandwidth intensive if many people are conferencing
- A number of comprehensive conferencing systems are available
  - ▶ Breeze
  - ▶ Raindance
  - ▶ LiveMeeting
- Provides a number of features
  - ▶ Screen sharing
  - ▶ Distributed whiteboard
  - ▶ Session recording
  - ▶ Video (in some cases)

# Collaboration

## Shared Documents

- Mail, IRC and Skype are useful for conversations and discussion
- Sometimes, we want to keep track of more information or multiple things
- Ideally this is document that multiple people can edit from different locations
  - ▶ Need to be able to track edits
  - ▶ Should know who changed what
- A good example of this system is a Wiki
  - ▶ Essentially, editable web pages
  - ▶ Very good for documentation and procedure descriptions

# Collaboration

## Shared Documents

- What about more formal documents? Say a proposal? Or a specification?
- Word allows one to track changes - but it's not distributed
- Google Docs is a useful tool for this task
  - ▶ Collaboratively edit word processing documents and spreadsheets
  - ▶ Export to multiple formats
  - ▶ Very useful for managing data in distributed projects

# Programming Methodologies

- Not a tool *per se* - a lot of it is common sense
- Can control what tools are used and how they are used
- A methodology describes many aspects of a programming project
  - ▶ How is the team structured?
  - ▶ How many people on a team? Level of experience?
  - ▶ How and when are tests written?
  - ▶ How often should releases be made?
  - ▶ How are specification changes handled?
- We won't focus on this topic, but many places do follow a specific methodology
  - ▶ Extreme programming (XP) has become a popular methodology

# Design Patterns

- Design patterns are “strategies”
- A generalized approach to tackling common problems faced when writing code
- By definition it is not explicit code - you have to *implement* it each time
- [Design Patterns: Elements of Reusable Object-Oriented Software](#) by E. Gamma, et al
- Patterns in one language may be “invisible” in another
  - ▶ [Design Patterns in Dynamic Programming](#)

# Design Patterns - Examples

## Global Variables

- Very useful, easy to do, source of much pain
- Application settings - usually want to access them from different classes or functions
- The *Singleton* pattern suggests that we have one class that can only be instantiated once in the lifetime of the program

## Initialization

- Parsing a molecule from SMILES is usually fast
- Usually we will need to check for aromaticity, atom typing etc
- What if we never do an operation that requires aromaticity?
- The *Lazy Initialization* pattern suggests that you only do these operations the first time they are required

## Design Patterns - Lazy Initialization

```
Molecule parseSmiles(String smiles) {  
    Molecule mol = parse(smiles);  
    mol.detectAromaticity();  
    mol.detectAtomTypes();  
    return mol;  
}  
  
double getMolecularWeight(Molecule mol) {  
    /* get the MW */  
}  
  
int getAromaticAtomCount(Molecule mol) {  
    /* count the number of aromatic atoms */  
}
```

# Design Patterns - Lazy Initialization

```
Molecule parseSmiles(String smiles) {  
    Molecule mol = parse(smiles);  
    return mol;  
}  
  
double getMolecularWeight(Molecule mol) {  
    /* get the MW */  
}  
  
int getAromaticAtomCount(Molecule mol) {  
    if (!mol.aromaticityDetected())  
        mol.detectAromaticity();  
  
    /* count the number of aromatic atoms */  
}
```

# Design Pattern - Example

## Looping over a list

- In C, we loop over an array index - we explicitly consider the array
- If we now deal with a linked list, the code will change
- The *Iterator* design pattern allows us to go through a collection of items without worrying about the data structure representing the collection

```
for item in myList:  
    print item
```

- What data structure is myList?
- Don't care

# Invisible Design Patterns

**Recurring problem:** *Two or more parts of a machine language program need to perform the same complex operation. Duplicating the code to perform the operation wherever it is needed creates maintenance problems when one copy is updated and another is not.*

**Solution:** *Put the code for the operation at the end of the program. Reserve some extra memory (a "frame") for its exclusive use. When other code (the "caller") wants to perform the operation, it should store the current values of the machine registers, including the program counter, into the frame, and transfer control to the operation. The last thing the operation does is to restore the register values from the values saved in the frame and jump back to the instruction just after the saved PC value.*

Taken from this [discussion](#)

## Design Patterns - Caveats

- Design patterns usually involve multiple layers of indirection
- Can lead to complexity, clutter
- Not all languages will support all design patterns
  - ▶ Most are meant for OO languages
- Many patterns are domain specific
  - ▶ REST is a web app pattern
  - ▶ MapReduce is a distributed computing pattern
- [Pattern Languages of Programs](#) is a useful conference for domain specific design patterns

## Design Patterns - Caveats

- Design patterns usually involve multiple layers of indirection
- Can lead to complexity, clutter
- Not all languages will support all design patterns
  - ▶ Most are meant for OO languages
- Many patterns are domain specific
  - ▶ REST is a web app pattern
  - ▶ MapReduce is a distributed computing pattern
- [Pattern Languages of Programs](#) is a useful conference for domain specific design patterns

# Design Patterns - Caveats

- Design patterns usually involve multiple layers of indirection
- Can lead to complexity, clutter
- Not all languages will support all design patterns
  - ▶ Most are meant for OO languages
- Many patterns are domain specific
  - ▶ REST is a web app pattern
  - ▶ MapReduce is a distributed computing pattern
- [Pattern Languages of Programs](#) is a useful conference for domain specific design patterns