Design of Software Systems (Ontwerp van SoftwareSystemen)

7 On Multi-user Development Tools,Versioning and Packaging of code, their Relations, the Universe and Everything.

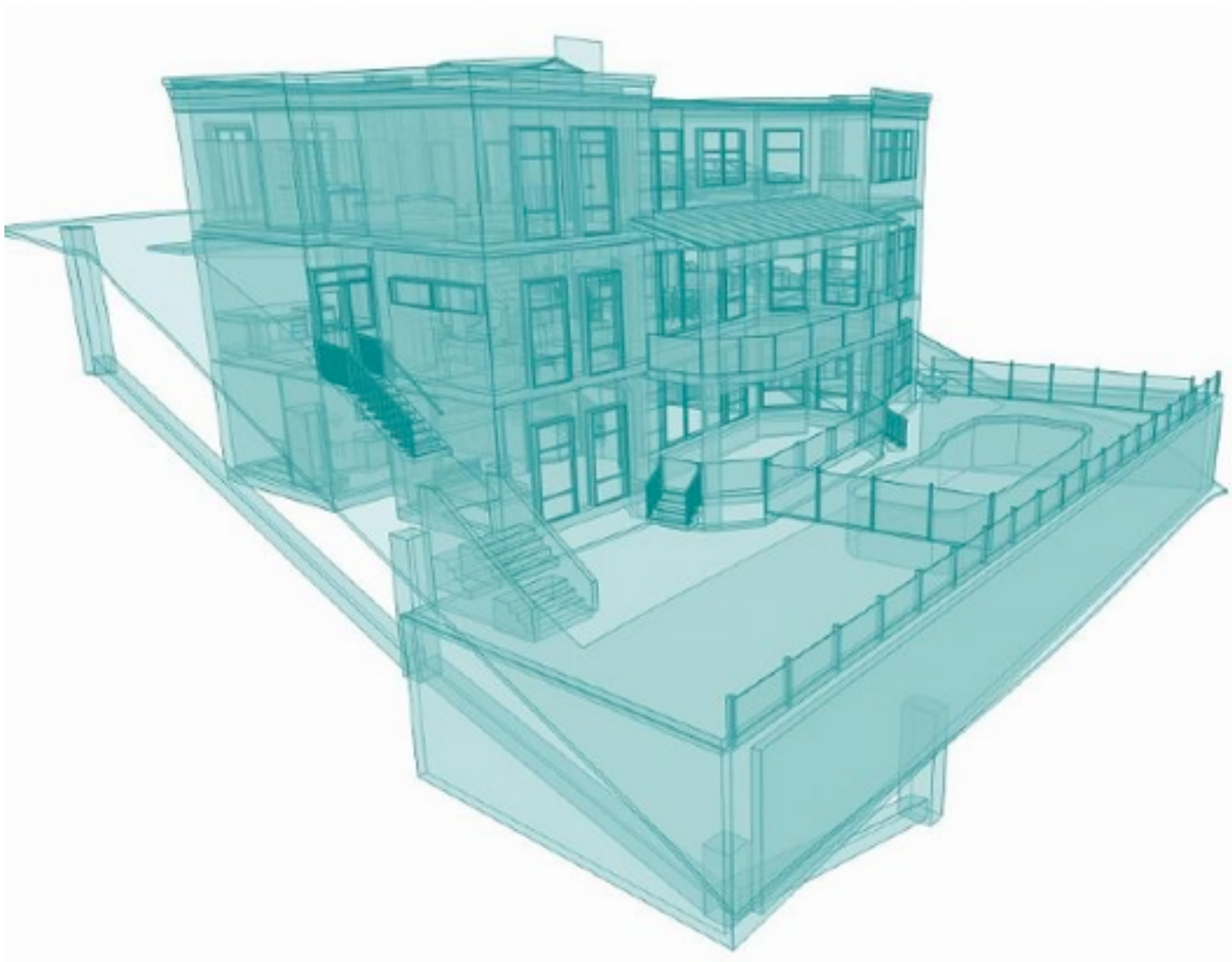
Roel Wuyts

OSS 2014-2015

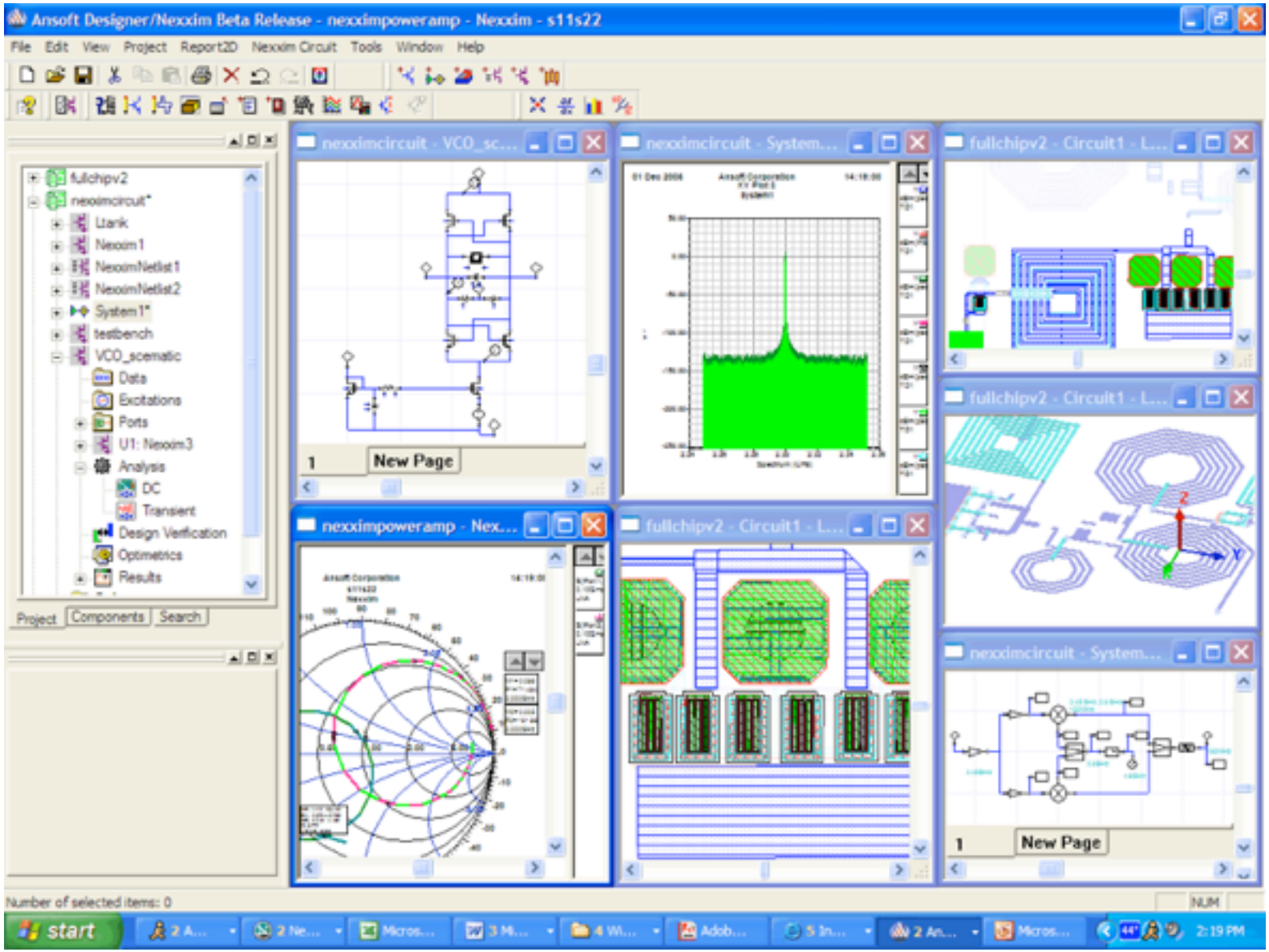
Developing Complex Systems

- How do scientific disciplines construct complex systems ?

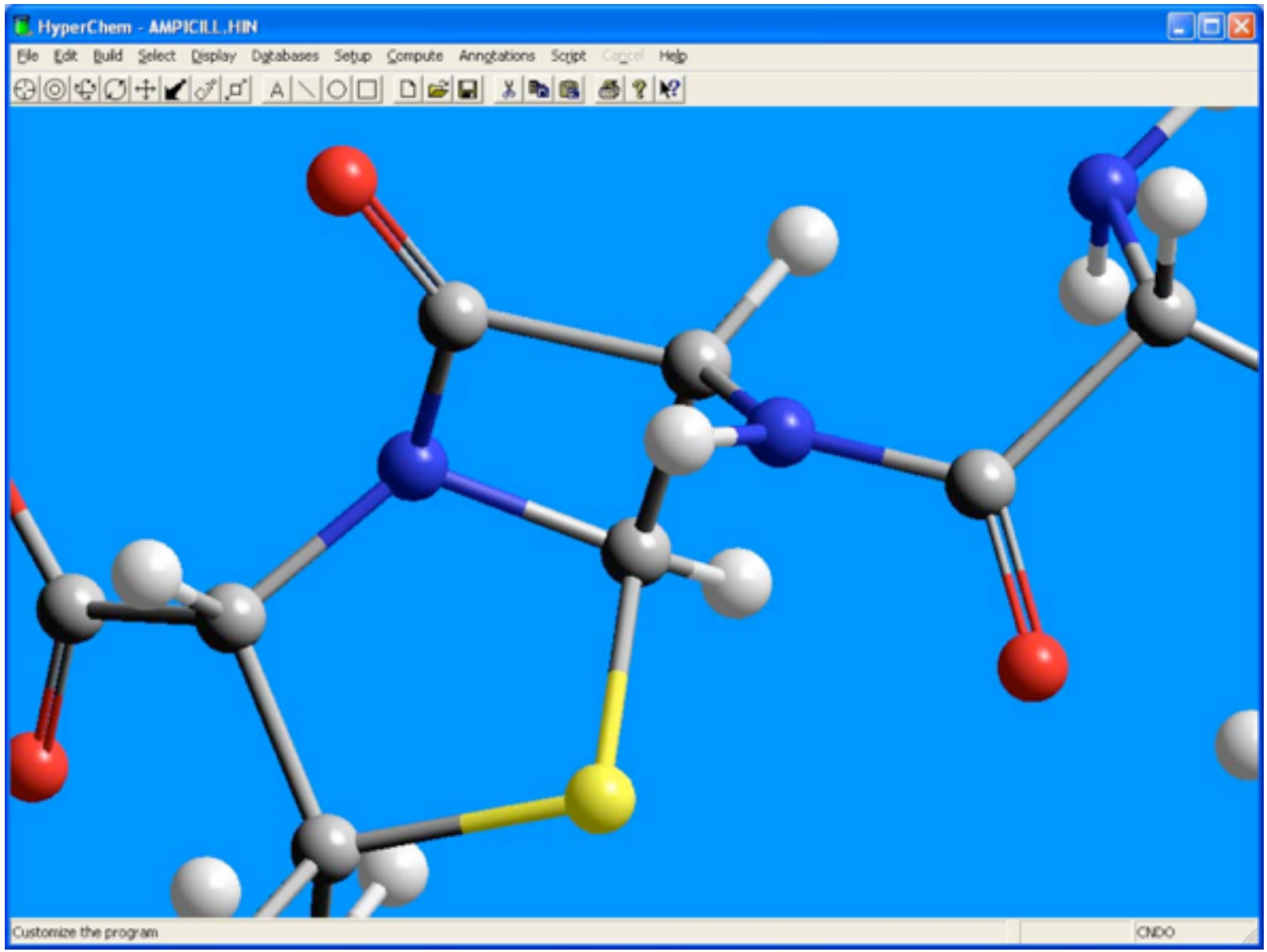
Architectural Software



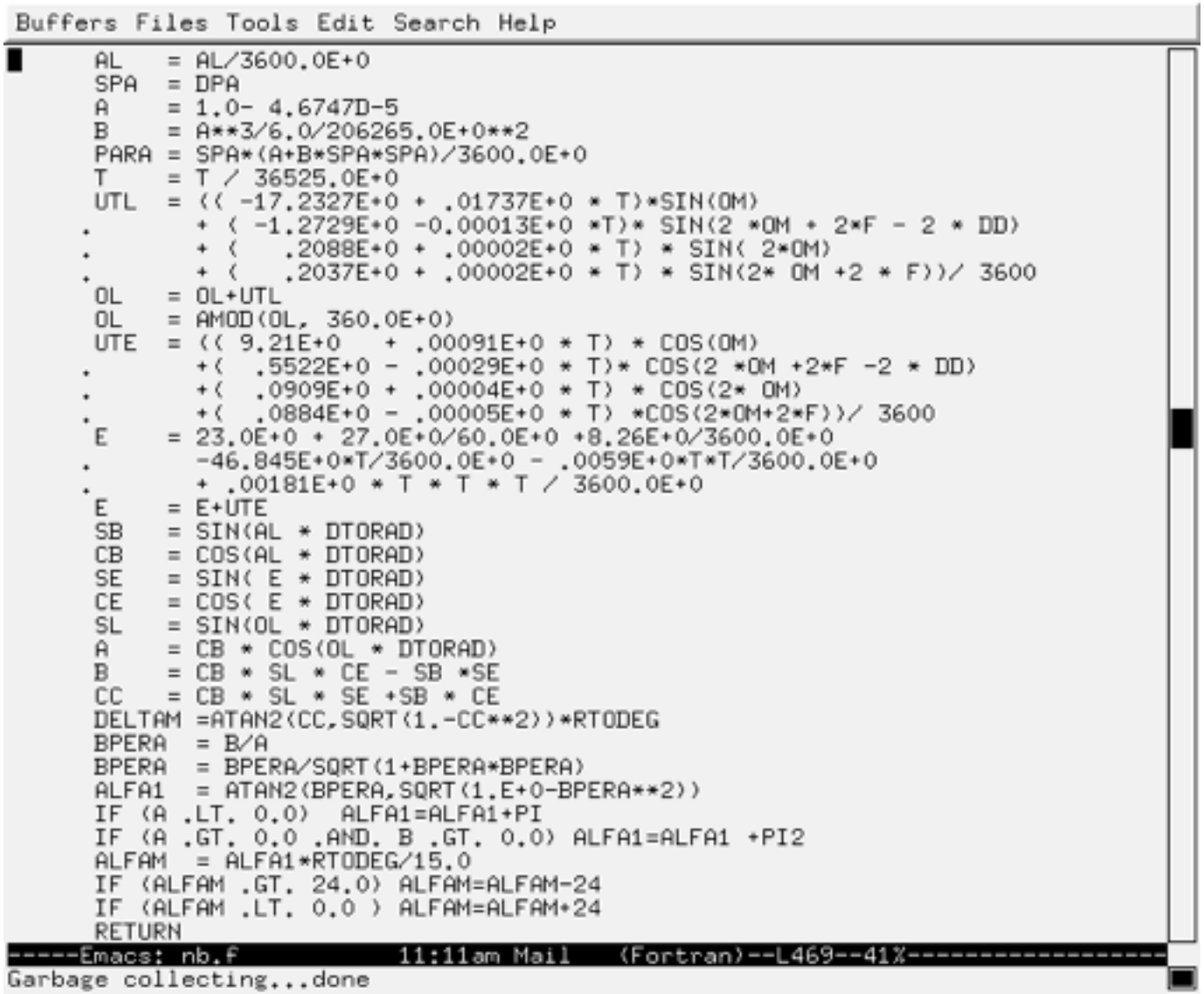
RF/mW Design & Analog/RFIC Verification



Visualization & Manipulation of molecules



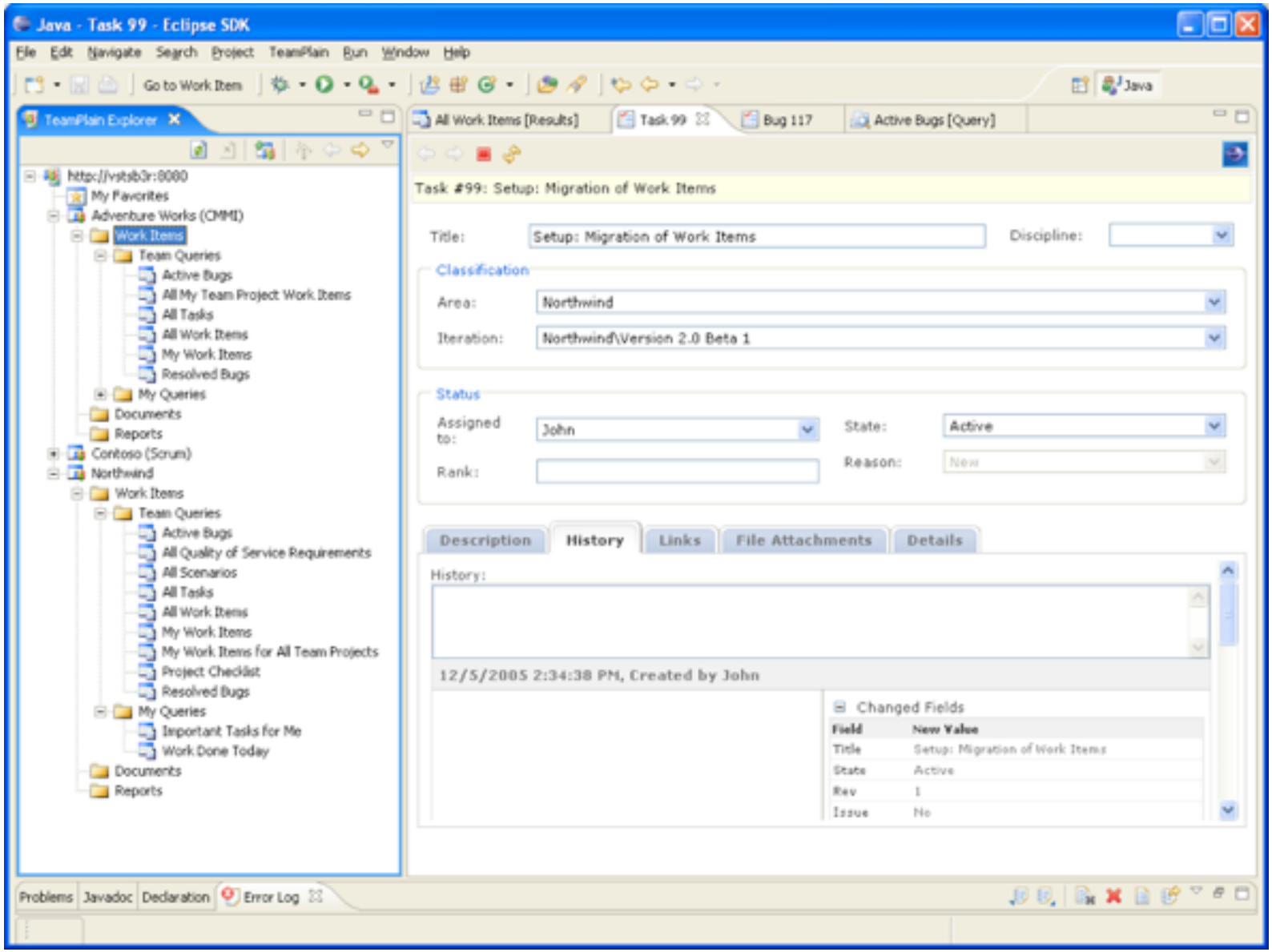
Computer Science/Engineering…



Corollary

- We need to construct systems that are typically more complex than in other disciplines
  - for several reasons
- We have tangible elements to manipulate
  - Buildings, circuits and molecules *need* a representation that is different than their physical one
- Yet lots of developers still seem to prefer basic tools
  - yes, emacs is a basic tool...

Eclipse/Netbeans/IntelliJ/… ?



Eclipse...

- Eclipse is a decent integrated development environment
  - integrates navigation, editing, unit tests, refactoring, ...
  - was developed by a lot of former Smalltalk people :-)
- But at its core it is file-based (and so are most others)
  - So ? Why don’t I like this ?

Files versus Objects

- Non computer science disciplines:
  - Architects work with construction materials&buildings
    - So do their tools
  - Molecular biologists work with modules
    - Environment manipulates molecules
  - ...
- We work with objects
  - Most tools deal with files ?!

Smalltalk image approach

- The Smalltalk image is a live environment
  - consists entirely of objects
  - objects are manipulated
- Files are one way of *storing* objects
  - code too, since code are objects
  - Databases are another mechanism, or network sockets or ...

Sidenote on Environments

- Good developers tailor their environment
  - So they need to be easily extensible
    - emacs: easy
    - Smalltalk environments: easy
    - Eclipse: possible
    - Most environments: hard or not possible
- Always favour an extensible one
  - control your tools!

Multi-user Development

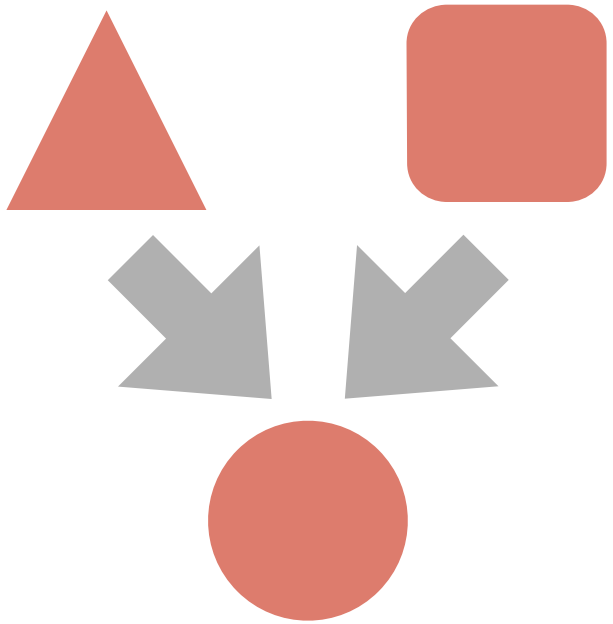
- Software engineering is a teamsport ;-)
- Needed
  - a code repository that allows multiple users
  - integrated versioning
  - configuration management
- The language also has packaging mechanisms
  - with or without namespaces
- These concepts cross-cut

Code repositories and multiple users

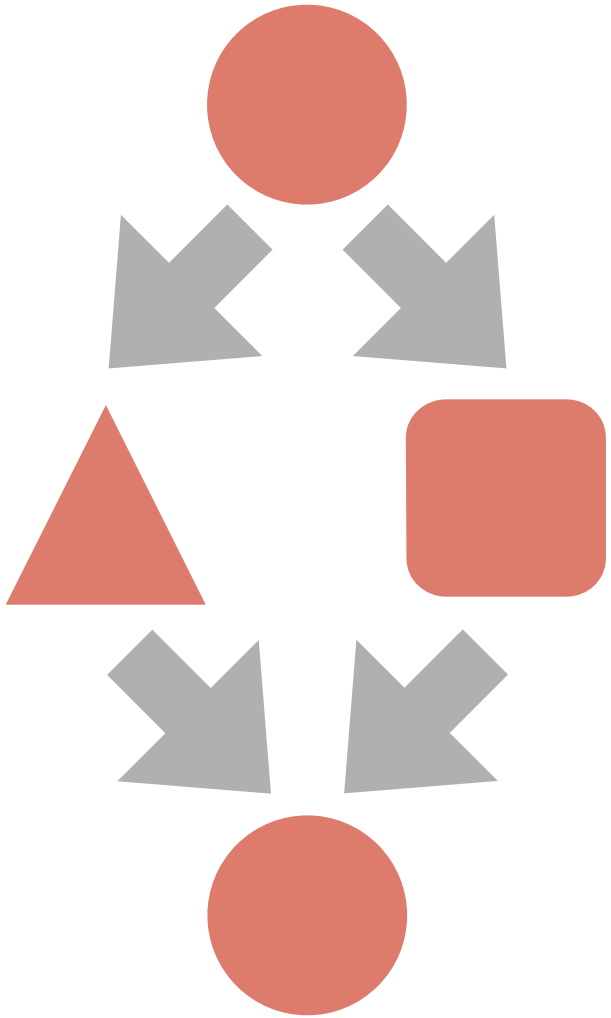
- Need to store code (obviously)
  - but preferable also binaries, documentation, tests, ...
- Locking vs. concurrent
  - Lock: one user has (part of) code, unlocks when done
  - Concurrent (lazy locking): several users can work simultaneously on the same system
- Centralized vs. Distributed
  - Centralized: Only the master repository contains complete version history
  - Distributed: all repositories have complete history
- Support for merging

Merges

- Two-way merge



- Three-way merge



Example

- One framework,
- instantiated for two different clients,
- each with their own customizations,
- Where there is a stable version,
- and two development branches
  - a new version and a brand new one
  - one dependent on the customization of the framework for one particular client

Let’s view two systems

- Concurrent, centralized systems:
  - Subversion
  - Envy
- Concurrent, distributed system:
  - git

(Many more exist, but svn is archetypical for most popular tools like cvs/git, and Envy is a contrast)

svn : Subversion

- descendant of cvs (concurrent versioning system)
- Granularity: file
- Users work detached from the repository:
  - Load local copy of files from svn server (*repository*)
  - Work on local copy (*working directory*)
  - Commit changed files back to repository
- Loading local copy can be done from the network

svn Workflow

- Check-out code from repository in local environment
- Work on code.
  - can at all time see the difference between the current change and the state when checked-out
- When finished, commit changes back to repository
  - can trigger (3-way) merge when repository was updated in the meantime

Semantics

- svn (like cvs, git, …) versions text
  - has no semantics about what text it stores
  - works with latex files, C++ files, ...
- Therefore its operations have no semantics
  - e.g. looking at changes after doing a *renaming a method* refactoring result in a list of textual changes to potentially many files
    - can be hard to know it was a refactoring, especially when combined when several other changes
    - commit often, and add comments !

Envy

- *It's a versioning system, but not as you know it ;-)*
- Users are meant to be always connected to the repository
  - can work separately but that is the exception
- Works with methods, classes, ...
  - Versioning knows about your language concepts
    - e.g. have all versions for a particular class, automatically includes all methods for that version of the class
  - Smallest granularity is a method

Envy Workflow

- Load code from repository in local environment
- Work on code.
  - *Every* change of a method or a class *automatically (!)* creates an *edition*
  - These editions can be compared with, restored, …
- Editions can be versioned
  - the edition then gets a name and version number
  - once versioned everybody in the repository can see and load versions
    - easier and earlier integration and conflict detection

Envy: Configuration Management features

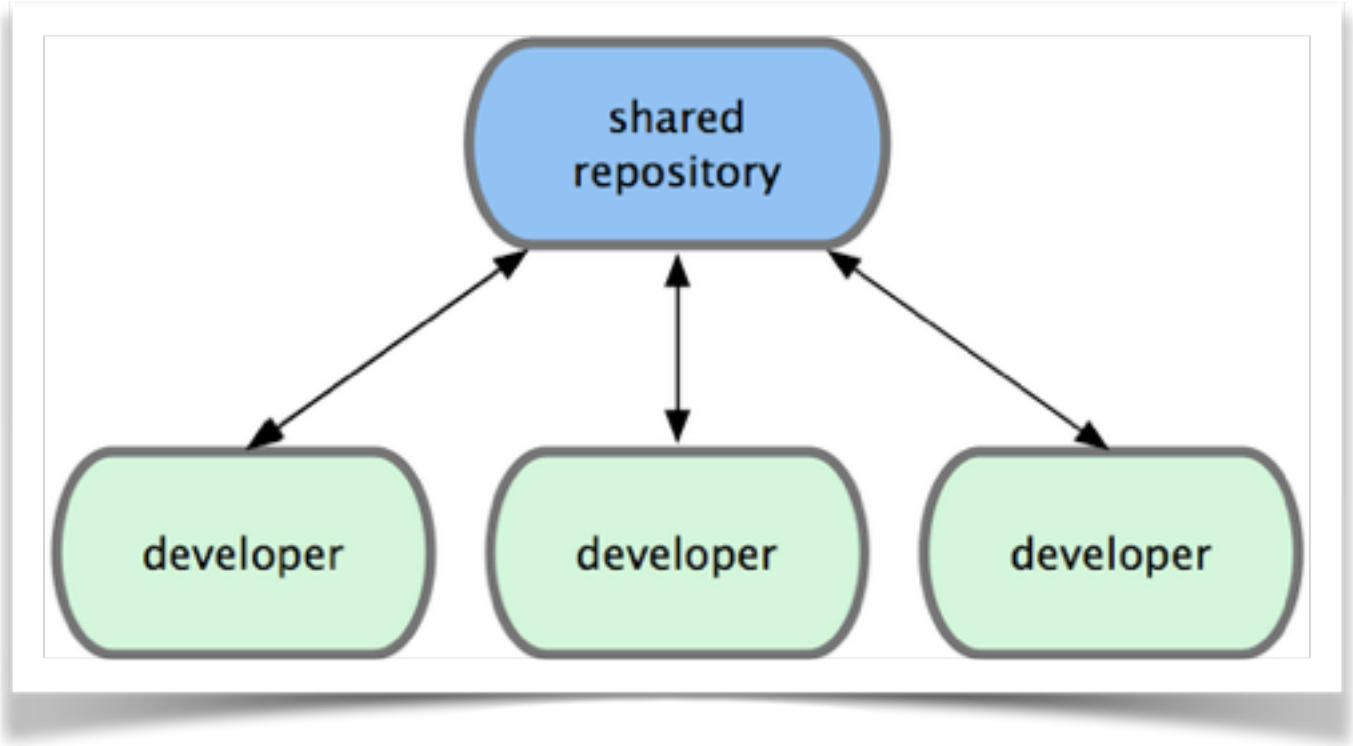
- From the ground up Envy has support for configuration management
  - *Applications* group classes and methods
    - can have editions and versions themselves
    - have prerequisite versions (!)
  - *Configurations* group applications
    - (e.g. Manifests in Microsoft .Net)
  - Support for *conditional loading* and *prerequisites*
    - Platform-specific code, for example
    - Can be at application or configuration level
- Removes need for external build systems like cmake, Maven, …

Git

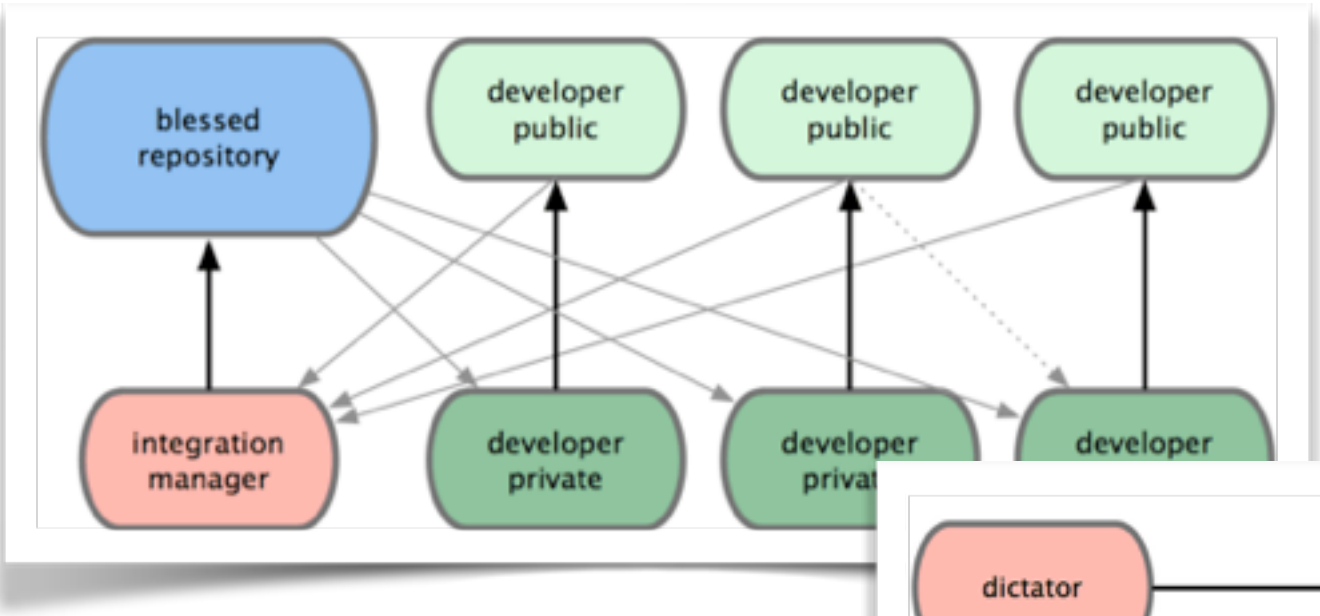
- distributed version control system
  - Breaks the master/slave relationship prevalent in cvs/ svn
    - every repository has the complete history
    - repositories sync with each other
  - Good support for branching and advanced forms of version management (cherry picking, reverting changes, …)
  - Like svn/cvs/…: stores text

Git workflow

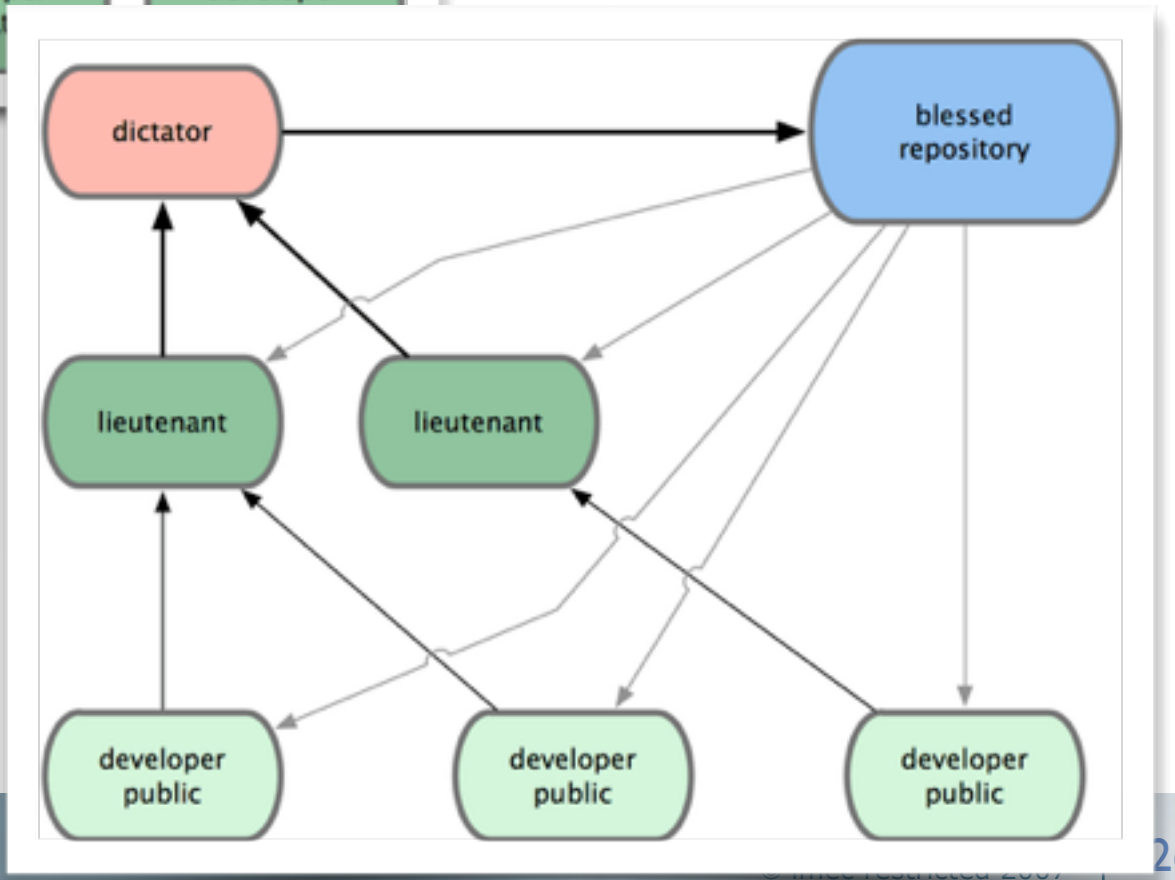
- Many possibilities
- Can of course do the centralised workflow (as in centralised approaches like svn/Envy/…)



Git workflow



Integration-Manager Workflow



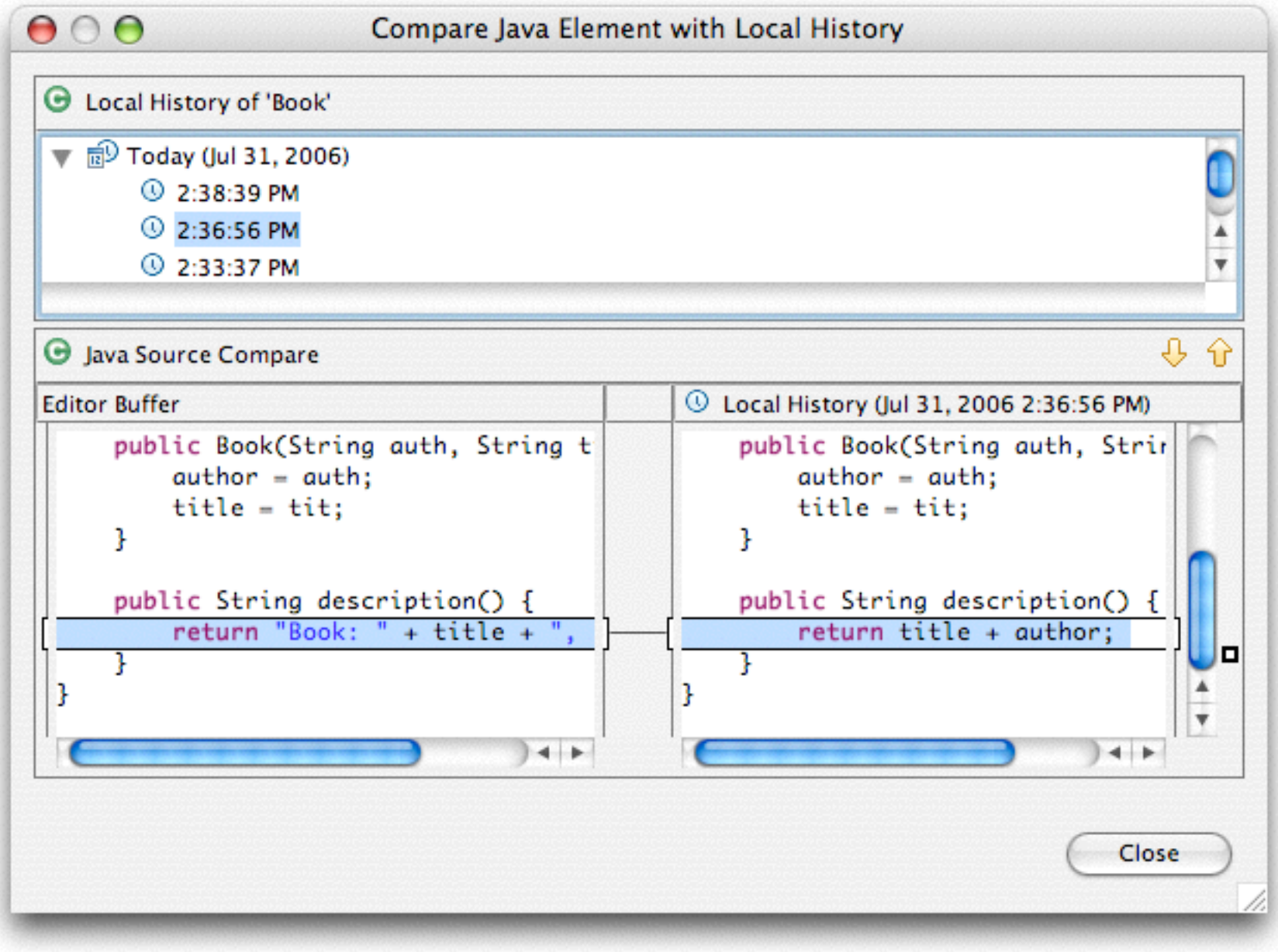
Dictator and Lieutenants Workflow

On granularity...

- With svn/git/…, you have a history of the *files* you've checked in
- With Envy, you have a history of the *development* you did

This is fundamentally different !

What is Envy doing in Eclipse ?!



Concepts in Code Repositories

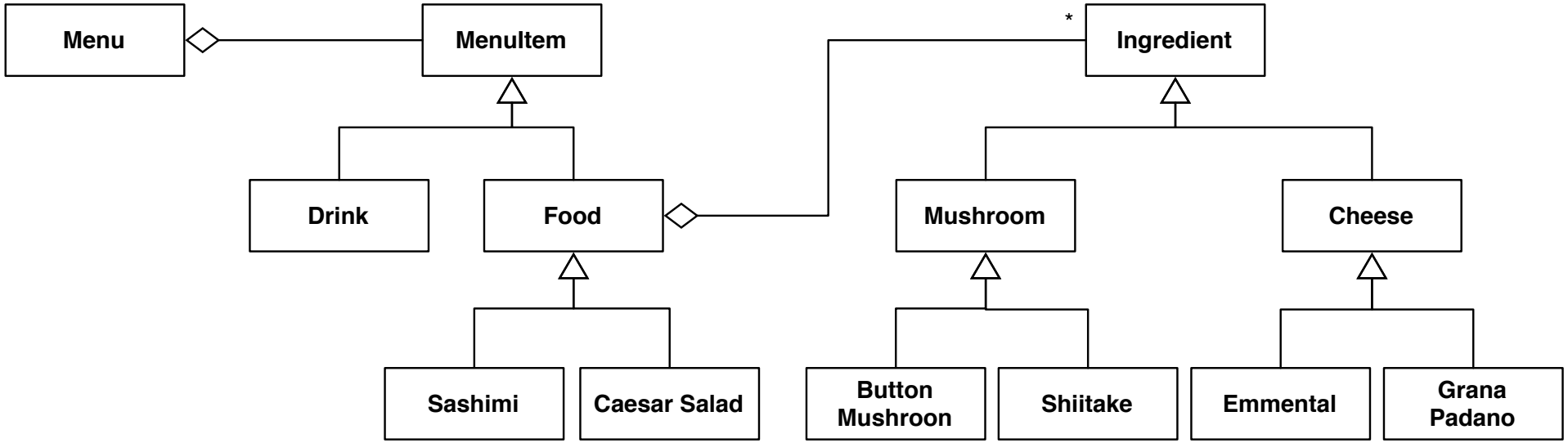
- Code
- Package
- Configuration

- Packages and Namespaces should be orthogonal
  - package contains definitions
  - namespaces is a visibility mechanism

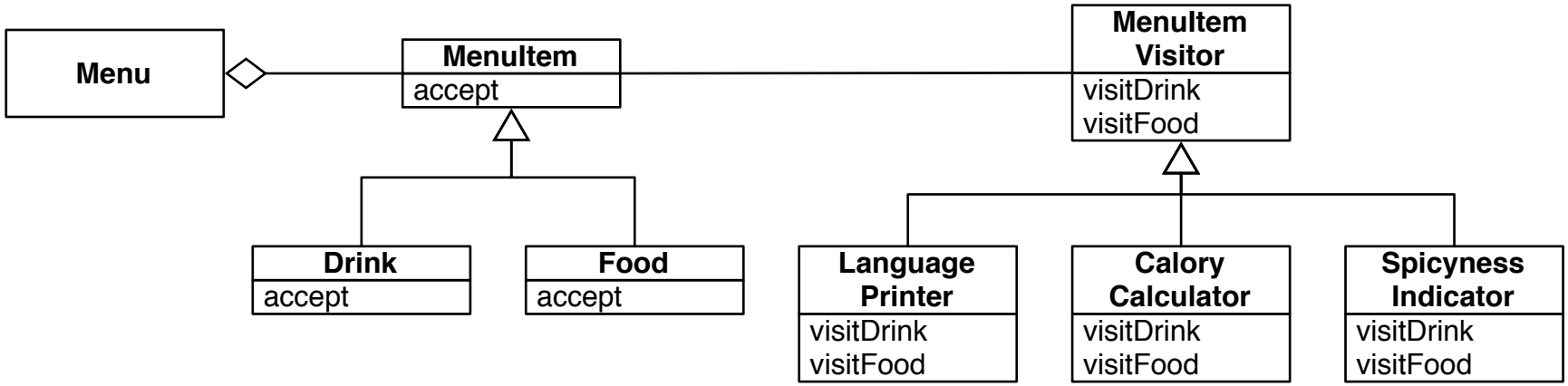
Versioning

- All the elements need to be versionable
- Decisions, decisions:
  - granularity of version
    - line of code, method, class+methods, package, ...
  - forms of version numbers
    - single number, composed number, alphanumeric
  - version numbers versus release numbers
    - and their relationships

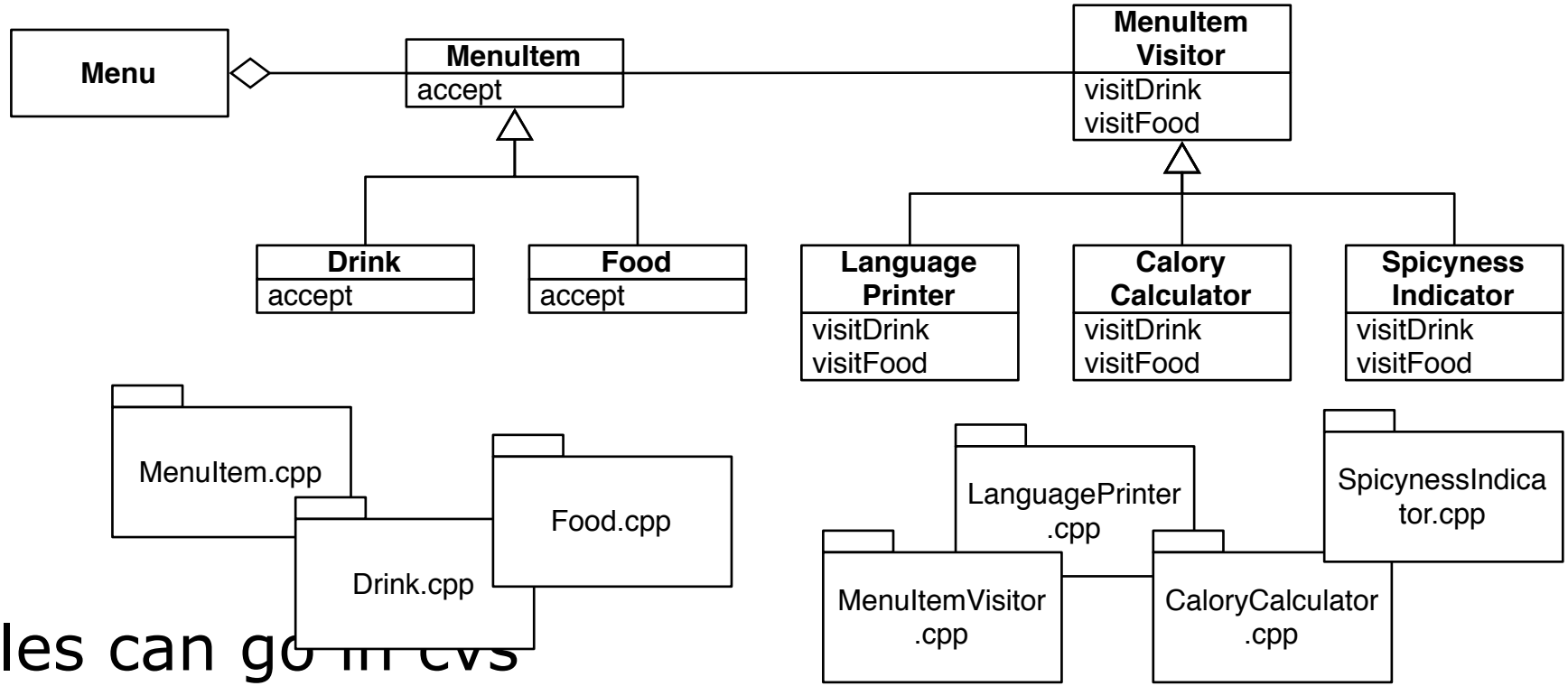
Concrete example : Menu Framework



Menu Framework with Visitor

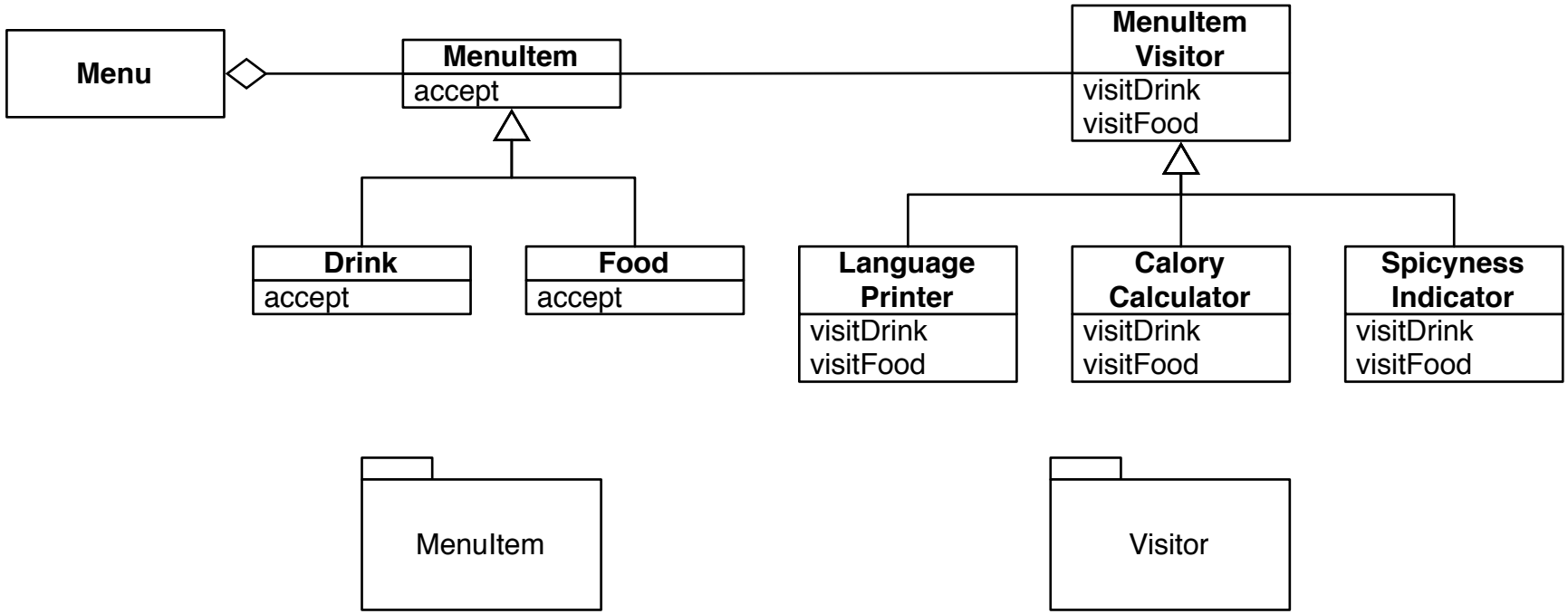


C++ Files



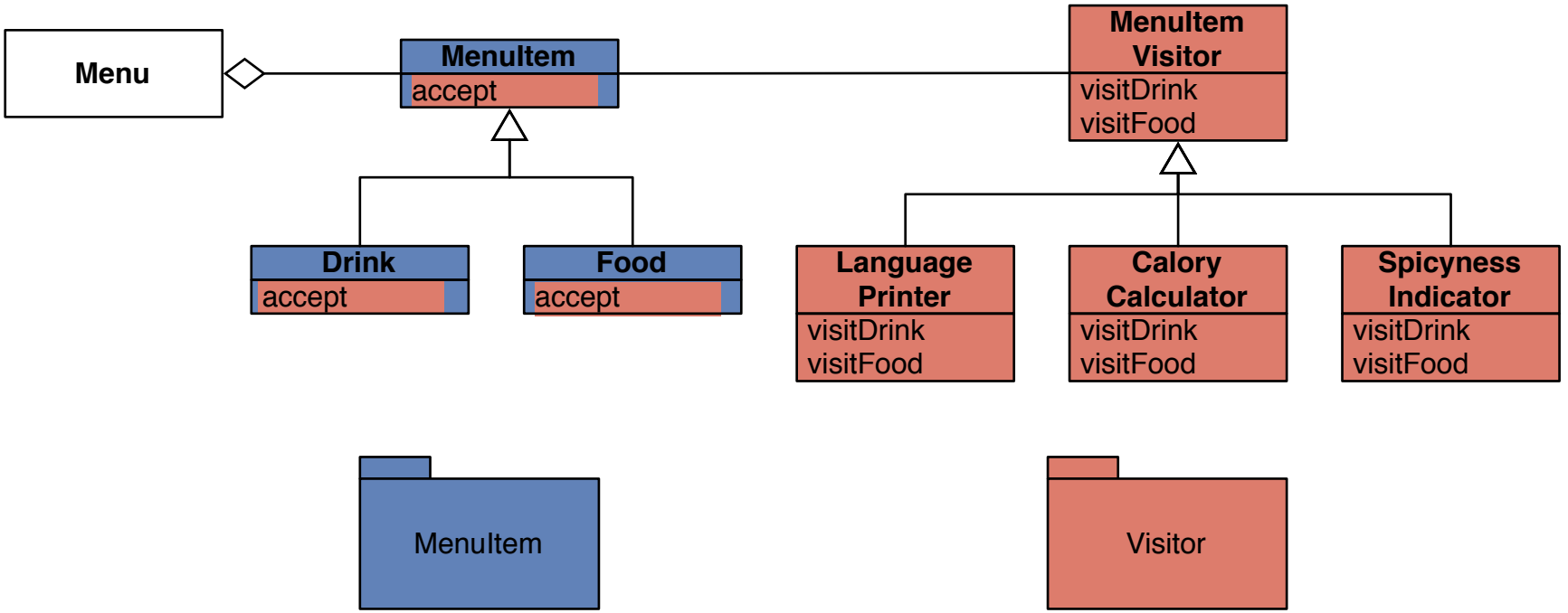
- Files can go in cvs
  - But decomposition is not the right one
  - What if the visitor traversal needs to be changed?

Java Packages



- Packages to regroup classes, storage still in files
- Decomposition still not the right one
  - What would be the right decomposition?

Smalltalk class extensions



- Packages defines classes and/or methods
  - Can be different versions, under control of different people/project/companies

Note: declarative packages

- Package systems should support software engineering and design principles
  - e.g. packaging Visitor pattern
- Approaches exist but should become mainstream
  - Smalltalk's class extensions
  - C# Partial classes and extension methods
  - Java Open Classes (for example in *MultiJava*)
  - …
- PS: or *multi-methods* in Lisp (and from there other languages)

Software-engineering wise

- Important to be able to separate development into logical, manageable pieces
  - e.g. Visitor design pattern
- Each piece should have:
  - owners & responsibles
  - versions
  - dependencies
  - post-load and pre-unload statements

Corollary

- Good packages support evolution
  - Company can sell parsetree
  - Other company can sell visitor for parsetree
- Code repositories and packages should support flexible forms of packaging code
- Code repositories, packaging & storage are linked

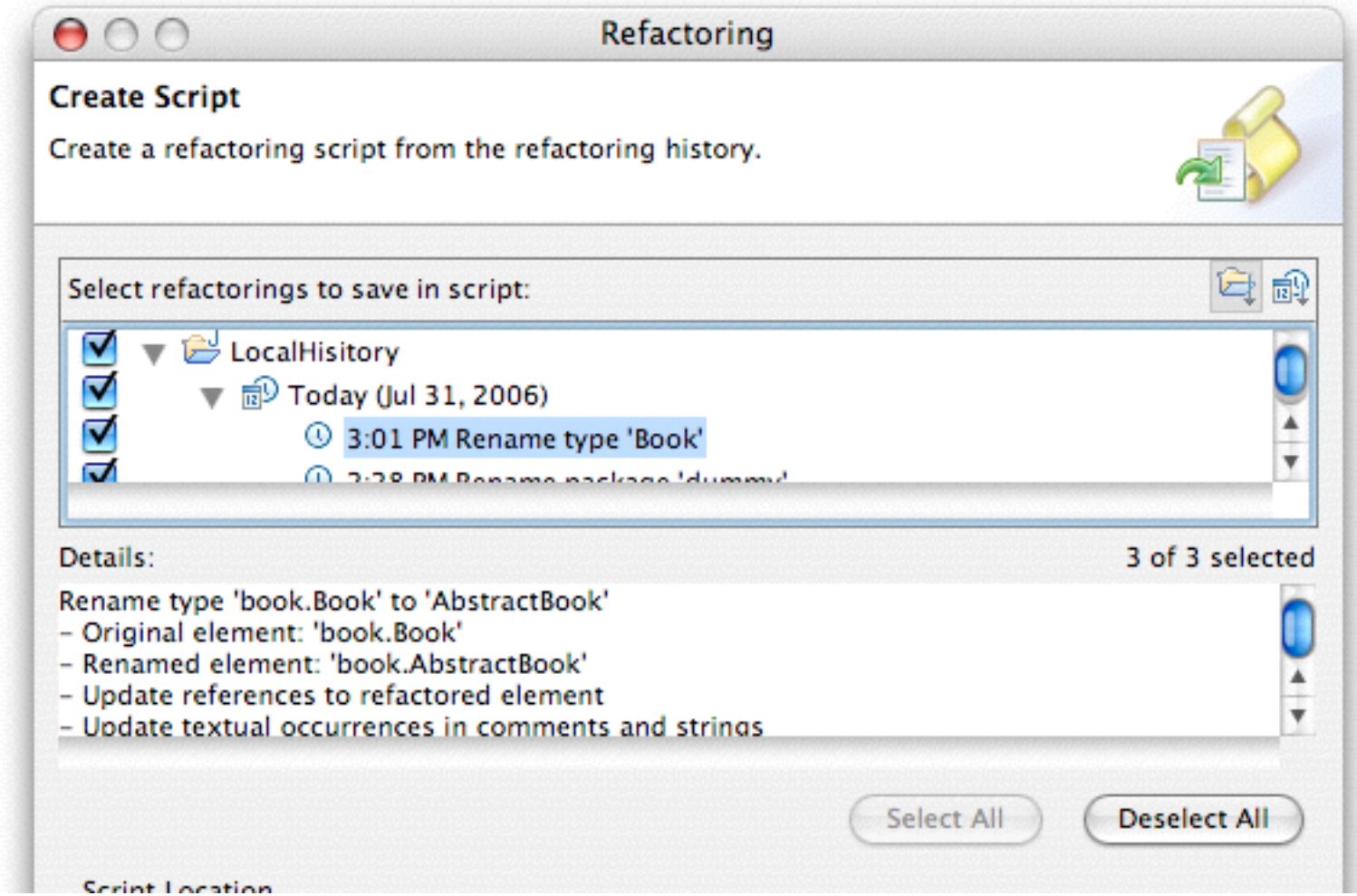
Sidenote

- Design question:
  - why is the plug-in mechanism in Eclipse so difficult?

Last but not least

- We discussed granularity
  - want to see the development you really did, not the changes you made
- Nice example: Refactoring Scripts in Eclipse
  - Record and replay the refactorings you did
- Why is this practical ?

Saving & Replaying Refactoring Scripts



Conclusion

- Need for supporting Multi-user development
  - code repositories with concurrent access
  - version support
  - (automatic) merge support
  - configuration management
- Current systems are quite weak
  - svn/git/… & files
  - proper packaging mechanisms
  - watch out for newer offerings

References

- Subversion: https://subversion.apache.org/
- git: http://git-scm.com/book
- Envy overview: http://stephane.ducasse.free.fr/FreeBooks/ByExample/36%20-%20Chapter%2034%20-%20ENVY.pdf
- Envy: Joseph Pelrine, Alan Knight, Adrian Cho, *Mastering ENVY/Developer*, Cambridge University Press, 2001.
- Smalltalk *Class Extensions*: https://www.youtube.com/watch?v=VNi_VQMosXQ

License: Creative Commons 4.0

http://creativecommons.org/licenses/by-sa/4.0/

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material

for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give **appropriate credit**, provide a link to the license, and **indicate if changes were made**. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the **same license** as the original.

No additional restrictions — You may not apply legal terms or **technological measures** that legally restrict others from doing anything the license permits.