Large(r) software systems

Nigel Goddard

School of Informatics
University of Edinburgh

Very simple Java programs

For the simplest Java programs, structuring into classes suffices:

- one file per (public) class
- one class, say Foo in Foo.java, contains a public static void main... method
- all source files are placed in the same directory
- you run `javac Foo.java`
- javac recompiles other classes as needed
- `java Foo` runs the program

How familiar is this?

Getting more complex

Even for single person projects, that approach doesn't stay satisfactory for long. Why?

- Sheer size. Once you get beyond 10, 20 or so classes, it's a pain to have everything in one directory (and one namespace). You'll *think* about some classes as being more closely related than others: it helps to make that visible.
- Internal dependency management. javac's ability to recompile exactly what needs to be recompiled won't be enough... e.g., because it only understands Java. Pretty soon there'll be something else you depend on, e.g., an XML schema.
- External dependency management. You'll need to use some library, some other software, some tool from elsewhere. You need somewhere to put these things, and some way to track dependencies on them.
- Packaging your own software. You'll probably need to distribute your own software as something more manageable than a bunch of .class files, e.g., as a .jar

What you need at this stage ($*$ $\longrightarrow$ more later)

You'll make some use of Java *packages*$*$, and corresponding directory structure. Most tools will come with your IDE:

Essential tools:

- Version control system (e.g. SVN or GIT)$*$
- Unit test tool (e.g. JUnit)$*$

Nice to have tools (non-exhaustive):

- Integrated Development Environment (e.g. Eclipse, Netbeans)
- Build tool (e.g. Ant or Maven)$*$
- Repository Manager (e.g., Artifactory)

Examples of possibly useful tools:

- Documentation generator (e.g. Javadoc) if you have programmer users$*$
- Bug tracker (e.g. JIRA,Bugzilla,Mantis) if you have many users
- Wiki, mailing lists, depending on community

Beyond single-person projects

What changes when you're working in a team, say up to about 10 people? Actually not so much – it isn't that you need new stuff, so much as that you need it *more*.

E.g., on your own, using version control* and a good clear coding style* are good habits. In a team, chaos results if you don't.

Less obvious kinds of dependency become important. E.g.,

- in a one-person project, the version of Java you're using is whatever's on your machine.
- Once other people are involved, you need them to have the same version.
- So it's good to have a record, along with the source files, of what version of Java they are written for, and a tool that can arrange for the right version to be used.
- That's one of the many jobs done by a project file in an IDE. e.g., *.project* in Eclipse.

Even bigger projects

Why would we ever need teams of > 10 people, and more complex software than 10 people can build?

First: often, it's arguable that we don't. Agile development* is founded on the idea of using a small, colocated team and biting off the most important requirements first. Martin Fowler's answer to "How would you manage a distributed team of 100 software developers?" was "I wouldn't"...

But sometimes, e.g.:

- a project involves many stakeholders, with fast changing requirements*
- many separate requirements are all needed soon
- project is multidisciplinary
- project needs many complex interfaces.

Some projects just cannot be done with only 10 people; some need $> 1 MLoC$ (Million Lines of Code).

Applying these concepts to the assignment

- "Building" is the process that gets you from source code to a running application.
- Fewer interfaces -¿ easier to build. E.g. a standalone single-user application will be easier than a Java code analysis tool or some forum software that needs a servlet container and a database...
- Get the source. Usually you download source from the SVN* or GIT repository, using e.g. `svn checkout [url]`. (If you haven't got an svn client on your machine you'll need one, e.g. TortoiseSVN.)
- Sometimes source will be packaged as a .zip or .tar.gz and you simply download this one file.
- Unpack it (using unzip, or gunzip and tar - RTFM as required). Look around. Read any README file!
- Build file: is there a file called build.xml or pom.xml? This gives instructions to the *ant* or *maven* build program* . Try `ant build` or `mvn` at the command line. ...

What else do we need then?

- Architecture.* It becomes much harder to change major aspects of the design of the software. Some decisions must be got right early. Clean interfaces essential.
- Communication. Big potential for decisions and information not to get to everyone. Need formal means of communication.
- People and project management.* People will certainly arrive and leave during the project.
- Planning, estimating and tracking.* Will need tool support for these.

These overlap.

Common themes: documentation becomes important, and can't be replaced by informal communications, however good; and need to make the development process* explicit.

Other kinds of complication

Not all projects that require special management or tools are large.

E.g. safety-critical software is kept as small as possible, because huge effort is required to assure correctness, even for a few lines of code.

The FADEC (full authority digital engine controller) for the Chinook helicopter that crashed in 1994, killing 25 people, was only around 16kloc (kilo lines of code).

Recommended reading: blog on problems in the FADEC code.

http://www.computerweekly.com/blogs/tony_collins/2010/02/eds-software-report-that-went.html

Socio-technical systems

Term used of a system that functions as part of society or an organisation.

Doesn't this apply to all software? Almost, but the term is useful as a reminder for systems where it's impossible to develop successfully with only a technical understanding, and the software's embedding into an organisation or social structure must be borne in mind throughout.

Examples: EUCLID; NPfIT; MyED; Facebook?

Very difficult to keep these small and simple. Projects often fail.

Non-examples: Single-user software, e.g., washing machine; anything where it's possible to specify all interfaces once and for all, and have them stay right.

But is that really true of any system now?

Sociologists who study socio-technical systems would say all but trivial software is part of a larger social system... c.f. Facebook.

Ultra-large scale systems (ULS)

We already mentioned "project needs many complex interfaces" as one reason why systems get large and complex.

The more we depend on software the more interfaces we need. Ultimately "the system" may not be easy – or useful – to identify.

Is "the Web" a system? If so, it's an example of a ULS.

Term coined around 2006 by Carnegie-Mellon's Software Engineering Institute in response to US Army's query: "Given the issues with today's software engineering, how can we build the systems of the future that are likely to have billions of lines of code?"

Question was unsurprisingly not answered! But SEI pointed out that, e.g., must design *mechanisms*, as can't predict behaviour in full detail; hardware and software failures will be normal and must be managed.

Jargon, sure – but indicative of a long-term trend.

Suggested Reading

- Blog on Chinook FADEC.
- CMU's page on Ultra Large Scale systems.

Pick your open-source package for the coursework!