Managing Sustainability into Software

Steve Crouch, SSI
s.crouch@software.ac.uk

Advanced School on Scientific Software Development: Concepts and Tools

24/02/2011

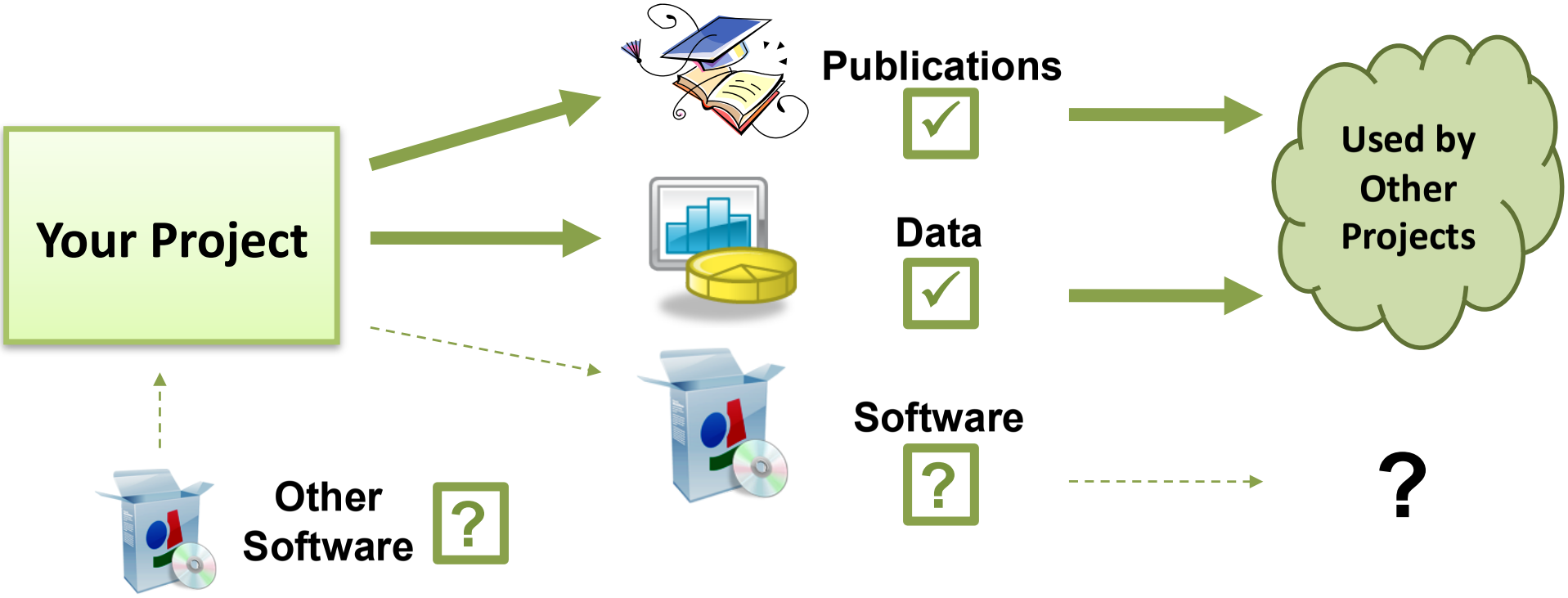
Background

- All projects have objectives
  - Produce research, study, infrastructure, software, …
- One view of project is
  - Software coming in: software you use, integrate
  - Software going out: software you develop yourselves
  - How you manage this software (and development)
- A practical, valuable view
- Poor software management often leads to
  - Poor knowledge & curation of software being used
  - Poor quality & provision of software going out

A Common Perception

www.software.ac.uk

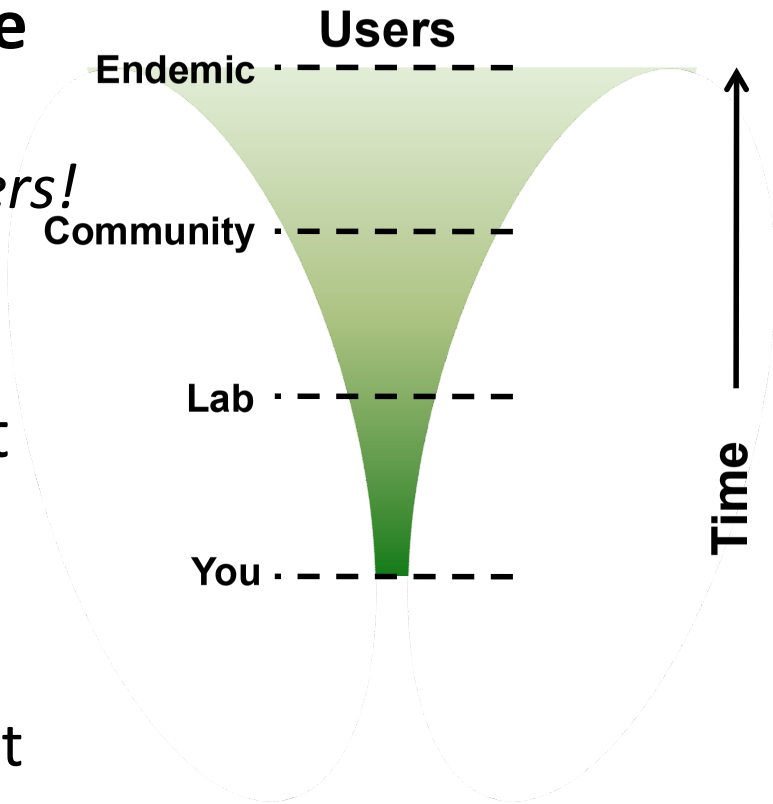
A view of software in research projects…



But… surely software is more important than this?

Is the Software You Write Important?

- Yes!
- Software inherently contains **value**
  - Used to produce (scientific) results
  - *Verification of these results by others!*
  - Contains technical lessons learned
  - Often represents a lot of effort
- Difficult to gauge to what extent it might be used in the future
  - By yourselves or others
  - The whole, or just a part of it
  - For a follow-on or unrelated project
- Can it/should it be reusable by yourself or others?



Aren't Plans for Managing Software Important?

- Yes, they should be – management plans exist for data
- Technical handover/exchange
- Extent depends on software type and value
  - Limited: e.g. project website
  - Important: used to generate results
  - Critical output: software is important output itself
  - Need to decide on scope of plan for given software
- Software inherently more 'active', different than data
  - Code, environment, dependencies, docs, designs, etc.
  - Capturing this information for others (and yourselves) is critical if you want it reused
- Potentially very expensive - need efficient means to do this

Beware of Decay!

www.software.ac.uk

- Software can *decay*
  - Functional operation degrades over time
  - Lack of proper maintenance
  - Passive: systems evolve or degenerate around it
  - Active: ill-conceived modifications
  - Some types more susceptible – Grid
  - It becomes unsustainable, unusable

What Makes Software Sustainable?

- Sustainable software: software that can continue to be developed and used by yourselves and others
- ***Group question: which properties or aspects of software help make software sustainable?***

What Makes Software Sustainable – A View

- Sustainable software: software that can continue to be developed and used by yourselves and others
- Perhaps…
  - Usability
  - Accessibility
  - Availability
  - Support
  - Good documentation
  - Open licensing
  - Extensibility
  - Maintainable code
  - Open standards

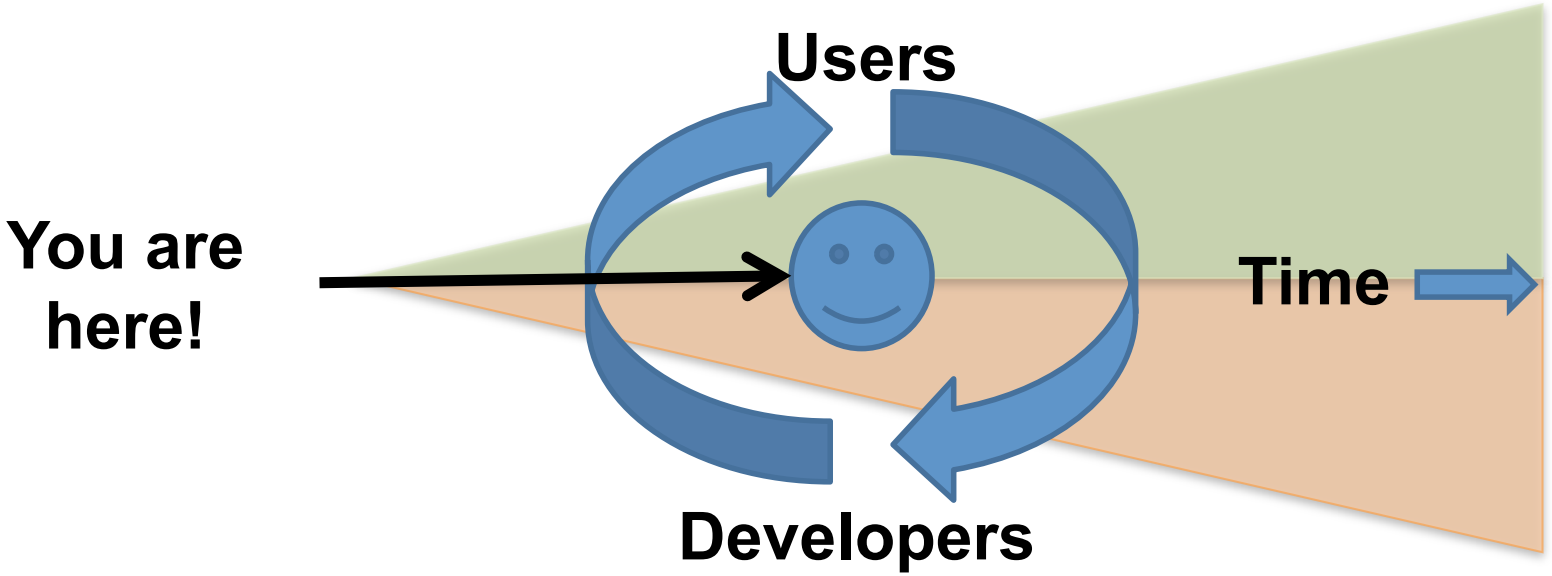
User

Developer

- Think about and prioritise these early!
- Assess as you go…
  - http://software.ac.uk/software-evaluation-guide

Building a Community

www.software.ac.uk



- This is often difficult, no easy answers
  - Have a good product (not perfect!)
  - Get people interested
  - Publicise – SSI can help here!
  - Try to **understand** and **respond** to needs of the community
- Some answers: http://www.software.ac.uk/resources/guides

Technologies that can Help

- Talked about properties that contribute to sustainability
- The importance of community
- ***Group question: which technologies/infrastructures can help to build a community?***

Technologies that can Help

- A website - first and foremost!
  - Presence; links; a unique ‘landing’ point for the software
- Mailing list(s), support/discussion forum(s)
  - Keep your users/developers up to date (releases, developments)
  - Encourage discussion (requirements, issues)
- Community Wiki
  - Central point for related community resources
- Issue tracker (Bugzilla, JIRA, RT, …)
  - For user/developer feature requests, bugs, etc.
- Source code repository
  - Helps avoid ‘dead laptop, lost software’ syndrome
  - Version control – regress to earlier versions of files
  - *Publish your computer code: it is good enough* – Nature, Nick Barnes

Where to Host?

- Run your own, if you have the resources
  - Relatively easy to set up (in general)
  - Full control, but be sure to back it up!
- Institutional
- Some public solutions can offer most of these for free!
  - SourceForge, GoogleCode, GitHub, Codeplex, Launchpad, Assembla, Savannah, …
  - BitBucket for private code base (under 5 users)
  - See http://software.ac.uk/resources/guides/choosing-repository-your-software-project

Good Code Development Practice

Readable Source Code

- This is *vital* – for you and others
- Source code is designed for humans
- Unreadable code can lead to bad perception of your software
- Writing readable code…
  - Costs only a fraction more than writing unreadable code
  - Payback is immense; for yourselves and others
- Good rule of thumb:
  - Always assume someone else will read your code

Code Formatting: Bad Example

- Formatting and appearance of code determines how quickly and easily developer can understand what it does

```
public class Functions
{
public static int fibonacci(int n)
{
if (n < 2)
{
return 1;
}
return fibonacci(n-2) + fibonacci(n-1);
}

public static void main(String[] arguments)
{
for(int i=0;i<10;i++)
{
print("Input value:"+i+" Output value:"+power(fibonacci(i), 2)+1);
}
}
}
```

Code Formatting: Better Example


```
public class Functions
{
  public static int fibonacci(int n)
  {
    if (n < 2)
    {
      return 1;
    }
    return fibonacci(n-2) + fibonacci(n-1);
  }

  public static void main(String[] arguments)
  {
    for(int i=0; i<10; i++)
    {
      print("Input value:" + i +
        " Output value:" + power(fibonacci(i), 2) + 1);
    }
  }
}
```

- Easier to see: overall control flow & encapsulation
- It's easy to do, but *be consistent!*

Naming

- Careful selection of names very important to understanding
- e.g. consider each of:

```
out(p(f(v), 2) + 1);
print(power(fibonacci(argument, 2) + 1);
```

- Common naming recommendations
  - Modules, components, classes typically nouns, e.g. Molecule
  - Functions, methods typically verbs, e.g. spliceGeneSequence
  - Boolean functions, methods typically questions, e.g. isStable
- Also relates to use of capitalisation and delimiters
  - Helps developer determine if something is function, variable or class
  - Common guidelines for C and Java:
    - Capitalisation of constants: PI, MAXIMUM_VALUE
    - Class names start with capital, first letter of subsequent words capitalised (camel case), e.g. BlackHole, DNASequence
    - Functions start with lower case letter, first letter otfsubsequent words capitalised e.g. spliceGeneSequence, calculateOrbit
- *Be consistent!*

Code Comments

- Very easy to assume others have prior knowledge
- Developer:
  - Should be able to understand a single function from its code and comments
  - Should not have to look elsewhere in code for clarification
- Examples for comments:
  - Why design/implementation decisions were adopted
  - Names of algorithms/design patterns implemented
  - Expected format of input files or database schemas
- Try not to state the obvious
- Keep them accurate (and up to date!)
- *Be consistent!*

Testing, Testing!

- Basic steps towards mature testing
- (1) Easy to compile: automated build process
  - Makes it far easier for developers to validate changes
  - Use existing tools e.g. Make, Ant, Maven
  - http://en.wikipedia.org/wiki/List_of_build_automation_software
- (2) Provide automated tests
  - Provide a *fail-fast* environment; expedites development
  - Many tools e.g. JUnit, CPPUnit, C++, xUnit, fUnit, …
  - http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks
- (3) Join together: automated build and test
  - Run overnight, send generated test report to interested parties
  - Code always in releasable state
- Decide on the right level of in-software development support
  - http://software.ac.uk/resources/guides/testing-your-software

Code Defensibly!

www.software.ac.uk

- Always avoid using deprecated interfaces
- Keep development and deployment environments as similar as possible
- Avoid specific software version dependencies
- Develop in operating system agnostic manner
- Test software on target platform regularly
- Loosely couple code to dependent software
- Use abstractions
- Enable end user (or software itself) to check environment
- Always adopt good software-maintenance practices
  - http://software.ac.uk/resources/guides/developing-maintainable-software

Benefits

- Of course, it all takes effort
- Why do some/all of these things?
  - It helps you, it helps others
  - Encourages structured development, can make organising work easier
  - Easier to share code and allow others to contribute – collaboration!
  - Assists your software in becoming more *sustainable*
  - Helps to raise software's profile
  - Positions your software for wider adoption

Approaches to Sustainability

- These are tools/techniques, what about the overall approach?
- Depends on various software factors:
  - Importance
  - Maturity
  - Community size
  - Resources available
- These can change over time, as can approach

- Software Preservation Study – SSI & Curtis&Cartwright
  - http://software.ac.uk/what-do-we-do/preserving-software

Sustainability Approaches

- Technical Preservation (techno-centric)
  - Preserve original hardware and software in same state
- Emulation (data-centric)
  - Emulate original hardware / environment keeping software in same state
- Migration (functionality-centric)
  - Update software as required to maintain functionality
  - Porting/transferring before platform obsolescence
- Cultivation (process-centric)
  - Develop a process to open up development of your software
- Hibernation (knowledge-centric)
  - Preserve knowledge of how to resuscitate/recreate exact software functionality at later date
- Deprecation (moved-on-centric)
  - Development effort has a planned end – could be right thing to do
- Procrastination (I'll-get-round-to-it-eventually-centric)
  - Do nothing!

The Significant Properties of Software

Capturing and Sharing Software Information

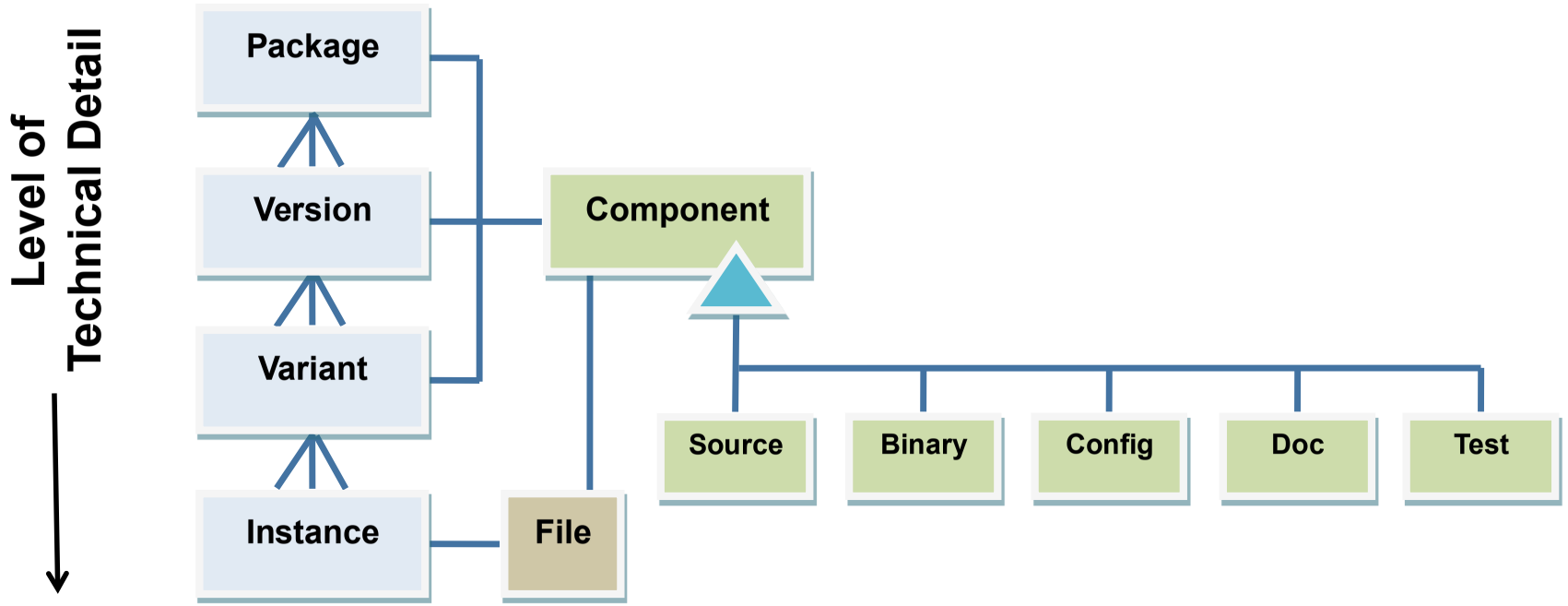
- Consider a user or developer – 'your software is exactly what I need!'
- What are the essential elements they need to understand…
  - What the software does, input and output data formats, configuration
  - Architecture, implementation details and constituent elements
  - Documentation, tutorial material, releases, operating environment, …
- Good documentation & website can supply this
  - Often unstructured or expensive to produce
- Description of a Project (DOAP) – SIMAL
- Software Ontology Project (SWOP) – Agile Software Description Ontology

FSP – Framework for Software Preservation

- Developed by STFC, Rutherford Appleton Labs, UK
- Captures 'significant properties' of software
- Focuses on mathematical, scientific and e-Science software types
  - No reason it cannot include others e.g. social sciences
- Encourages you to think about *minimum* set of information for describing your software
  - A good place to start
  - The *process is as important as the result!*
- Also gives an understanding of what to look for in other software you want to use

How it Fits Together

www.software.ac.uk



- **Package:** the entire software system e.g. MyProg
- **Version:** typically a release of the software e.g. MyProg v1.0
- **Variant:** operating environment, e.g. OS, language runtime
- **Instance:** a physical deployment on a machine
- **Component:** a digital artefact supplied within the software, e.g. source code, binary, configuration, doc, test

Categories of Software Properties

- Categories for Package, Version, Variant and Instance
  - Functionality
  - Provenance and ownership
  - Software Environment
  - Software Architecture
  - Operating Performance
  - Software Composition
- At each level of Package -> Version -> Variant -> Instance, technical detail increases
  - Package level has high-level descriptions, motivations, software requirements
  - Instance level has low-level technical properties for a working installation, building on the above levels

Property Category	Software Property			
	Package	Version	Variant	Instance
Functionality	purpose	functional_description	variant_notes	-
	keyword	release_notes		
		algorithm		
		input_parameter		
		output_parameter		
		interface		
		error_handling		
Provenance and Ownership	package name	version_identifier	licence	licensee
	owner	licence		conditions
	licence			licence_code
	location			
Software Environment	-	programming_language	platform	environment_variable
		hardware_device	operating_system	IP_address
			compiler	hardware_address
			dependent_library	
Software Architecture	overview	detailed_architecture	dependent_package	-
		dependent_package		
Operating Performance	-	-	processor_performance	-
			memory_usage	
			peripheral_performance	
Software Composition	software_overview	source	binary	file
	tutorials	manual	source	
	requirements	installation	configuration	
		test_cases		
		specification		

Further Resources

- SSI:
  - Guides: http://software.ac.uk/resources/guides/defending-your-code-against-dependency-problems http://software.ac.uk/ready-release
  - Sustainability approaches: http://www.software.ac.uk/resources/approaches-software-sustainability
- Software Carpentry:
  - http://www.software-carpentry.org

Further Resources

- Other resources:
  - OSS Watch: http://www.oss-watch.ac.uk/resources/sustainableopensource.xml
  - Wikipedia: http://en.wikipedia.org/wiki/Programming_style http://en.wikipedia.org/wiki/Indent_style http://en.wikipedia.org/wiki/Identifier_naming_convention
  - How not to write Fortran in any language http://queue.acm.org/detail.cfm?id=1039535