The European Materials Modeling Council

EMMC guidance on quality assurance for academic materials modelling software engineering

Proposed recommendations for software development in LEIT projects

This document presents the advice of software owners, commercial and academic, on what academic software could do to generate better quality software, ready to be used by third parties.

Preamble

Under Horizon2020, funding programmes have divided their scope by TRL levels

> *TRL 0: Idea. Unproven concept, no testing has been performed.*
> *TRL 1: Basic research. Principles postulated and observed but no experimental proof available.*
> *TRL 2: Technology formulation. Concept and application have been formulated.*
> *TRL 3: Applied research. First laboratory tests completed; proof of concept.*
> *TRL 4: Small scale prototype built in a laboratory environment ("ugly" prototype).*
> *TRL 5: Large scale prototype tested in intended environment.*
> *TRL 6: Prototype system tested in intended environment close to expected performance.*
> *TRL 7: Demonstration system operating in operational environment at pre-commercial scale.*
> *TRL 8: First of a kind commercial system. Manufacturing issues solved.*
> *TRL 9: Full commercial application, technology available for consumers.*

1. The European Research Council is addressing TRL1-3. They fund bottom-up research, usually in individual grants to support a group of a junior or senior principal investigator and the first priority is the development of an initial, promising idea.

2. The “Materials” (NMBP) programme is under the LEIT pillar (leading and enabling industrial technology) and is aimed at funding projects between TRL3-7. These are typically collaborative projects with a consortium with a mixture of academic and industrial partners to ensure that the goals have specific and realistic industrial relevance. In material modelling projects, the industrial partners could be either software companies or manufacturers or finished product sellers.

3. TRL7-8 is taken care of by companies who will customise the results and use them in commercial operations.

In this document we will briefly address all phases in materials modelling software engineering and then concentrate on the software development done in the LEIT programme.

This guidance is meant to help guide the change from FP7 till H2020 and discuss how physics and chemistry PhD students can become ready to do the work in LEIT.

Introduction

The transfer of software developed in many universities to industry remains insufficient. Academic institutions do not target to provide fully developed software or materials models services.
For this a lot of effort is to be invested (corresponding to some 70% of the total software development) and for this spin-off companies are created or the software is transferred to existing commercial companies.

To stimulate this transfer to industry an inquiry is done into the requirements of commercial companies on the quality of software.
Quality of the model is the ability of the software to solve the initially defined problem.
Quality of the software is defined by:

- Code design.
- Stability and resilience.
- Modularity.
- Performance (CPU, memory, …)

and the software quality should lead to software which is easy to maintain, understand, improve and adapt, and deploy to the target users.

Cooperation forms

Most software houses do not adopt software in which they have not been involved. The continuum model vendors tend to develop the software themselves, while in the discrete world models are adopted by the commercial vendor. Most companies have standing relations with a university group where the PhD students change but the relation continues. And this allows the software to be adapted to later customer needs. Access to the original developers over the course of the software life time is important. EU projects are thus just a stage in this long-lasting relation.

Licenses
As the investment is large academics and companies agree licenses with their co-developers and customers that allow the return of profits.

Consideration of the advantages and drawbacks and pitfalls of different licensing schemes is important. Many commercial companies feel open licenses and viral licensing schemes like GPL are a blocking point when transferring or using software from academia to industry as they do not allow commercial exploitation. However other companies support Open Source code – like the many Linux support companies. Trends in main licensing schemes were mentioned during the discussion:

- One-to-one licensing. Service License Agreement/Contract can be limited in time (and thus cheaper) and allow "software (solutions provided) as service and/or on demand".
- LGPL is better than the “viral” GPL scheme, but poses a commercial constraint as only “dynamic linking” into commercial software is commercially viable – and dynamic linking has a performance disadvantage.
- BSD or Apache licensing schemes are most permissive in allowing to link technology building blocks of non-commercial R&D partners into commercial software in a performant manner.
- Dual licensing: Many academic institutions go for dual licensing which keeps the software free to academia but allows commercial protection and there is some ROI.

Commercial SWO recommend to use a permissive licensing scheme (as BSD or Apache) already for the R&D and prototyping phase of new methods and software tools, as the resulting technology blocks will be acceptable to commercial tool vendors downstream, which smoothens the procedure to get new software modules into industrial deployment (by leveraging on the installed base of commercial software at manufacturing industry).

Stages of software development

In the current situation, three phases of software development can be identified.

PHASE I Exploratory phase (modelling)

In this phase new models[1] are elaborated with new physics/chemistry, and the models are applied to one or two specific test cases. Exploratory model development takes place mostly in academia. Software developed in this stage is typically developed to test a scientific idea. ERC funding could be requested for this Phase. In this phase the science is important and the software is considered to be a side product. The ideas are often the basis for literature articles.

The software is not user-friendly and is often shared freely among specialists (mostly academics) through PhD and postdoc collaborations. IP protection is usually only covered by the academic code of conduct.

If a SWO becomes interested, this interest of the SWO is often based for 80% on the functionality and for 20% on the software quality.

Huge amounts of money and time are wasted on maintaining and developing a bad piece of software. (The PhD student is spending all their time on making a code converge or to restart the code when it crashes). If the commercial company rewrites the code then they either lose the benefit of future academic developments or have to spend their time re-implementing the ideas in their code.

PHASE II Bridging Phase: new material modelling software and software prototyping

New material modelling software has been validated on some test cases and has thus proven wider applicability and the software has the ability to solve new problem(s) of value for industry. The software is in a prototype phase and the software has to become validated trusted code during this phase. In this documents we do not address quality of the model and influence of boundary

[1] Materials are complex systems and the equations that describe the physical and chemical behaviour of real systems are often too complicated to be solved easily. In order to save computer time, which is a precious and limited resource, the description of phenomena has to be simplified. Fortunately, often not all details need be taken into account in order to reproduce and predict experimental results. Key assumptions about reality can be made ignoring the complexity that is not necessary to describe the given situation. These approximations are called "models" (see "Review of Materials Modelling What makes a material function? Let me compute the ways…", 3rd edition, 2014, http://ec.europa.eu/research/industrial_technologies/pdf/modelling-brochure_en.pdf)

conditions, and solver technologies or coupling to other models. This will be addressed in another document to be issued by modellers.

This phase can be subdivided further:

- Phase II a: Proof-of-concept phase. Techies from industrial R&D departments become interested in academic results or new models. Their drive is to find something new. In this phase they may work with software developers/ companies to develop software for proof of concept.

- Phase IIb: Visionaries in industrial context get involved. They are concerned in finding better solutions within a broader business context. Projects in this phase typically last 2-3 years and are used also in collaboration with software developers to solve particular useful problems, including validation and metrology.

Often, scientific/academic modellers work together with commercial software owners and follow their software habits. The academics will need to understand and respect standards, formatting, protocols etc. with regard to the existing software data structures and architectures.

Licensing: Academics often use open source licences like GPL. Alternative models are used like L-GPL and small commercial companies (spin-offs) may be involved at this stage to support the commercial development and exploitation. Alternatively, dual licensing models are used so that academics can continue to develop the code and commercial companies can explore those cases where potential industrial users/consumers are not yet clearly identified or the value of the results in the industrial setting is not yet established.

In the later stages of this phase, the software is used by a wider community and applied to a wider range of systems.

Versioning: Software versioning supporting tools are sometimes used but it's too often considered too cumbersome.

Additional software writing may also happen in this Phase, e.g.

scripting, may be at the level of a material scientist (depending on the language and individual skills)

automating and packaging (not usually in their expertise…)

Good software engineering practices will help in overcoming the so-called "CHASM"[2], where many new models fail to be introduced to the mainstream market.

PHASE III Commercial modelling software.

The main target is the commercial scale-up of software for more than a small community of end-users. In this phase "pragmatists" will need to become convinced that a new technology is worth their time, effort and money to get it deployed into their organisations. These pragmatists are managers which are primarily interested in the economics behind a new approach, they are reluctant

[2] "Crossing the Chasm" by G.A. Moore (1991)

to change, and will only decide on a change if they're convinced that it's worth it for their operational / ROI targets. Improvement in performance is only one factor in this phase, several other aspects are considered: investments already done, 'cost of change' (in training and process changes), ability to cope with potential demand for new products. It may be preferred to buy out the software in this phase.

The effort needed to rewrite the code for exploitation purposes is large as software developing skills by material scientist are often not sufficient (even when they can write codes for themselves that calculate useful things) and professional expertise might be necessary. Today this deployment step unfortunately often involves a complete rewrite of the code by software engineers and this phase amounts often to 70% of the total development effort and this is to be reduced.

Quality control

Quality control is performed using multiple quality targets that must be monitored and considered simultaneously for every software release. This allows avoiding 'waterbed effects': focusing intensively on improving 1 quality target will risk to reduce the performance of another quality target. For example, reducing the level of details provided to the user reduces the 'content' quality and accuracy, but increases the speed of calculation and result evaluation. Formal software quality management metrics are used e.g. **CFURPS[3].**

Commercial companies put a large effort into ensuring full validation, documentation, user-friendly interface, testing suites, durability long term support systems are developed since the life span of the successful models is many decades.

In order to get the software into an industrial framework the code need to be portable, well defined and documented APIs and product packaging on the one hand, and communication between developers on both sides on the other. In general, for commercial exploitation, a piece of software must be robust and relatively easy to use, with a good user interface. This can require a great deal of development in the:

1. Installer and licensing system,
2. Graphical User Interface,
3. Help system and manuals,
4. Updating system.

The software company will consider data and exchange standards. In case wrappers and homogenisation tools are needed also these need to be developed if they do not yet exist.

There could be a range of tasks required to be undertaken to get the software/data in a form ready for exploitation. Maintenance (adaptation to new IT) is an important issue as software is on the market for many years/decades and both the solution and the framework are not static in time, nor are the underlying hardware architectures. This development involves constant updating to changing

[3] Compatibility, Functionality Usability Reliability Performance Supportability (CFURPS)

environments (like Apple dropping the active support for Java on Mac).The company has to think about possibilities for service provision and training that can accompany the software.

Catalysts to facilitate the flow of innovation from model developers to mainstream end users use are:

- Bi-directional communication: awareness upstream at academia w.r.t. downstream industry needs, and vice-versa awareness downstream at industry w.r.t. promising academic methods.
- Availability of information: store material models/data, document best practice methods/tools, and demonstrate their impact on the basis of industrial use cases.
- Standardisation of data and interfaces to enable and facilitate the take-up of information by other actors in the industrial value chain, and with other tools that cover parts of this value chain. Note that multi-scale modelling in itself can, in simple cases, also be seen as an interface between different tools available at different simulation levels, which must be integrated to be able to deal with applications at a higher scale.

For making and selling new tools that operate within a standard “mainstream” process at the industrial end user, a shortcut may be found when the innovative component is implemented as a new “feature” (an integrated module) into the “mainstream software” that is deployed at the end user. This allows the design engineers at the end user company to use the new “content block” directly, without making any (disruptive) changes to the software tool chain, or to the in-house process to use the tools, nor requiring too extensive training of the staff involved, or requiring changes to the teaming and organisation. The “new content block” simply acts as a new / improved feature within a well-known process and tool chain, familiar to the users, hence there’s a smaller chasm: there are less hurdles in the organisation to be overcome before deployment of the new feature within the tool chain that’s already well known and broadly used.

Proposed guidelines on how material modelling software should be developed in EU-LEIT funded projects

The purpose of these guidelines is to make it more likely that models developed in LEIT projects are used in industry. The purpose of these guidelines is thus to ensure that the models/software are transferred to professional software owners (academic and commercial) who licence the code to third parties and maintain the code. After the code is fully developed we enter phase III and the code can be sold by (commercial) SWO who sell the codes to an end-user, or academic software SWO who transfer the code to other users , but also by new translator actors (e.g. spin-offs) who use the code themselves in R&D services done for industrial end users. In this document we only focus on Phase II, the completion of the software development up to TRL 7. The target of EU H2020 –LEIT research and innovation funding will be to support Phase II in the projects since it will bridge the gap over the software valley of death. Phase III is considered commercial and thus not part of EU funding.

The cost of developing code after the initial research on the functionality of the model can be substantial. It is certainly so when software funding is starting to demand small proof-of-concept or emerging technology software components to be developed. To optimize these investments, the following recommendations have been gathered:

Proposed recommendations for phase II software development

- Prove wider applicability via validation on a set of fully documented test cases. (benchmark test and literature references, proof of some sort)- Documented results obtained in collaboration with industrial users solving specific industrial problems.

- Document accuracy/ functionality proven in high profile academic publications (especially important in some markets, e.g. Japan)

- Demonstrate good numerical implementation, proven numerical stability (performance, boundaries on values of parameters for numerical stability), pragmatic approach to software engineering including the use of software metrics (e.g. Halstead (1977), Watson & McCabe (1996) (Kan, 2003).

- Provide good documentation of underlying equations and (justification of choice of) algorithms, validation cases, boundaries of validity, clear mapping of raw formula to code variables, manuals/tutorials explaining what the code can do.

- Include appropriate testing procedures and document the results in deliverables.

- Provide clear documentation of which version of the code was used to obtain what result and content of the next upgrades (not necessarily professional versioning).

- Provide documentation about IT requirements (operating system, compilers, MPI distribution, …) and how to compile the code.

- Adopt a licence allowing commercial industrial use

-Demonstrate interest of potential customer, and traction with potential future users

-Collaborate with SWO to drive the direction (mentor for software writing)

In H2020 programmes under the "Leadership in Enabling Industrial Technologies" pillar, it is of crucial importance that proposals are led by industrial needs.

When the new model addresses a new and previously unaddressed problem in a “niche” market, reference cases are of paramount importance to generate new business, research grants should show the potential of the new method on an application case that is familiar to the target end user (at least in terms of having similar complexity and comparable workflow)

Developers in phase II should consider if the new model can be implemented as an innovative integrated component of the “mainstream software” that is deployed at the end user; this approach has a high ROI (return on investment) as a more effective industrialization can be achieved.

Licences

It is recommended that the guiding principle should not only be “Don't put any obstacles for future commercialization” but rather, “Have an organized system of sequential licensing options, which would allow for commercialization under specific terms”.
Licences should be chosen together with the mentoring SWO in the project who is likely to exploit the software. Although in the “prototype" phase II in principle any tool (and licensing scheme) can be used for the purpose of validating a methodology, we believe that it’s wise to keep in mind future re-usability of code blocks in case an innovation moves downstream the innovation chain, as this reduces the hurdles to bring the innovation in position to provide value to the end users by solving their application challenges.

Education:

It is of utmost importance that academic software developers receive better information on alternative licensing models that do not preclude commercial exploitation.
Workshops on basic software writing schemes may be useful at this stage as well as workshops on versioning, software repositories and testing suites that can guide the development. Several existing activities are running in Europe and some documentation is available online (see Appendix II).
Academics should be informed about best software writing practises before starting to write any code – not just at the exploitation stage.
The course should deal with issues like

- Code design. - This depends on the problem solved and on the parallelisation/load balancing concepts employed in the code - memory distribution, structures and communication patterns. There are multiple ways to achieve this. Best practices are about code neatness (file, modularisation, dependences, architectural design), cleanliness, testing, comments, keeping track of changes (repositories), test case to demonstrate performance and functionality, and documentation.
- Modularity. - Mentioned in code design as code neatness (file, modularisation, dependences architectural design), cleanliness
- Performance (CPU, memory …). - also mentioned above
- Portability
- Numerical stability deviations from standard models with controlled termination when entering ill-defined problem input (rarely). There may be much more depending on the hardware architecture too.

As it takes time to learn to write good software and coding standard evolves, it is also recommended to involve a SWO/software engineer mentor who overlooks the design of the software and the technology translation objectives to convert a technology block into a solution of end user challenges. The motivation for the SWO to become a mentor will be the business potential identified. If the software owner is interested in the code they might require a specific programming language and software development tools that are used in their company.

Appendix I

Licensing models guidance notes

Annex (from Wikipedia December 2014)

A **software license** is a legal instrument (usually by way of contract law, with or without printed material) governing the use or redistribution of software. Under United States copyright law all software is copyright protected, except material in the public domain. A typical software license grants an end-user permission to use one or more copies of software in ways where such a use would otherwise potentially constitute copyright infringement of the software owner's exclusive rights under copyright law.

In addition to granting rights and imposing restrictions on the use of software, software licenses typically contain provisions which allocate liability and responsibility between the parties entering into the license agreement. In enterprise and commercial software transactions these terms often include limitations of liability, warranties and warranty disclaimers, and indemnity if the software infringes intellectual property rights of others.

Software licenses can generally be fit into the following categories: proprietary licenses and free and open source. The significant feature that distinguishes them are the terms which the end-user's might further distribute or copy the software.

Proprietary software licenses

The hallmark of proprietary software licenses is that the software publisher grants the use of one or more copies of software under the end-user license agreement (EULA), but ownership of those copies remains with the software publisher (hence use of the term "proprietary"). This feature of proprietary software licenses means that certain rights regarding the software are reserved by the software publisher. Therefore, it is typical of EULAs to include terms which define the uses of the software, such as the number of installations allowed or the terms of distribution.

The most significant effect of this form of licensing is that, if ownership of the software remains with the software publisher, then the end-user *must* accept the software license. In other words, without acceptance of the license, the end-user may not use the software at all. One example of such a proprietary software license is the license for Microsoft Windows. As is usually the case with proprietary software licenses, this license contains an extensive list of activities which are restricted, such as: reverse engineering, simultaneous use of the software by multiple users, and publication of benchmarks or performance tests.

The most common licensing models is per single user (named user, client, node) or per user in the appropriate volume discount level, while some manufacturers accumulate existing licenses. These open volume license programs are typically called Open License Program (OLP), Transactional License Program (TLP), Volume License Program (VLP) etc. and are contrary to the Contractual License Program (CLP), where the customer commits to purchase a certain amount of licenses over a fixed period (mostly two years). Licensing per concurrent/floating user also occurs, where all users in a network have access to the program, but only a specific number at the same time. Another license model is licensing per dongle which allows the owner of the dongle to use the program on any

computer. Licensing per server, CPU or points, regardless the number of users, is common practice as well as Site or Company Licenses. Sometimes one can choose between perpetual (permanent) and annual license. For perpetual licenses one year of maintenance is often required, but maintenance (subscription) renewals are discounted. For annual licenses, there is no Renewal, a new license must be purchased after expiration. Licensing can be Host/Client (or Guest), Mailbox, IP-Address, Domain etc., depending on how the program is used. Additional users are inter alia licensed per Extension Pack (e.g. up to 99 user), which includes the Base Pack (e.g. 5 user). Some programs are modular, so one will have to buy a base product before they can use other modules.[4]

Software licensing also includes maintenance. This, usually with a term of one year, is either included or optional, but must often be bought with the software. The maintenance agreement (contract) contains Minor Updates (V.1.1 => 1.2), sometimes Major Updates (V.1.2 => 2.0) and is called e.g. Update Insurance, Upgrade Assurance. For a Major Update the customer has to buy an Upgrade, if not included in the maintenance. For a maintenance renewal some manufacturers charge a Reinstatement (Reinstallment) Fee retroactively per month, in case the current maintenance has expired. Maintenance normally doesn't include technical support. Here one can differentiate between e-mail and tel. support, also availability (e.g. 5x8, 5 days a week, 8 hours a day) and reaction time (e.g. three hours) can play a role. This is commonly named Gold, Silver and Bronze Support. Support is also licensed per incident as Incident Pack (e.g. five support incidents per year).[4]

Free and open-source software licenses

Free and open-source licenses generally fall under two categories: Those with the aim to have minimal requirements about how the software can be redistributed (permissive licenses), and those that aim to preserve the freedoms that are given to the users by ensuring that all subsequent users receive those rights (copyleft Licenses).

An example of a copyleft free software license is the GNU General Public License (GPL). This license is aimed at giving all user unlimited freedom to use, study, and privately modify the software, and if the user adheres to the terms and conditions of GPL, freedom to redistribute the software or any modifications to it. For instance, any modifications made and redistributed by the end-user must include the source code for these, and the license of any derivative work must not put any additional restrictions beyond what GPL allows.[5]

Examples of permissive free software licenses are the BSD license and the MIT license, which give unlimited permission to use, study, and privately modify the software, and includes only minimal requirements on redistribution. This gives a user the permission to take the code and use it as part of closed-source software or software released under a proprietary software license.

Free Software Foundation, the group that maintains The Free Software Definition, maintains a non-exhaustive list of free software licenses.[6] The list distinguishes between free software licenses that are compatible or incompatible with the FSF license of choice, the GNU General Public License, which is a copyleft license. The list also contains licenses which the FSF considers non-free for various reasons, but which are sometimes mistaken as being free.

Viral license is a pejorative name for copyleft licenses that allows derivative works only when permission are preserved in modified versions of the work.[1][2][3][4][5] Copyleft licenses include several common open source and free content licenses, such as the GNU General Public License (GPL) and the Creative Commons Attribution-ShareAlike license

The term is most often used to describe the GPL,[*citation needed*] which requires that any derivative work also be licensed under compatible licenses with the GPL. The viral component is described as such because the licenses spreads a continuing use of the licenses in its derivatives.[6] This can lead to problems when software is derived from two or more sources having incompatible viral licenses in which the derivative work could not be re-licensed at all.

Copyleft is a general method for making a program (or other work) free, and requiring all modified and extended versions of the program to be free as well.

The simplest way to make a program free software is to put it in the public domain, uncopyrighted. This allows people to share the program and their improvements, if they are so minded. But it also allows uncooperative people to convert the program into proprietary software. They can make changes, many or few, and distribute the result as a proprietary product. People who receive the program in that modified form do not have the freedom that the original author gave them; the middleman has stripped it away.

In the GNU project, our aim is to give all users the freedom to redistribute and change GNU software. If middlemen could strip off the freedom, we might have many users, but those users would not have freedom. So instead of putting GNU software in the public domain, we “copyleft” it. Copyleft says that anyone who redistributes the software, with or without changes, must pass along the freedom to further copy and change it. Copyleft guarantees that every user has freedom.

Copyleft also provides an incentive for other programmers to add to free software. Important free programs such as the GNU C++ compiler exist only because of this.

Copyleft also helps programmers who want to contribute improvements to free software get permission to do so. These programmers often work for companies or universities that would do almost anything to get more money. A programmer may want to contribute her changes to the community, but her employer may want to turn the changes into a proprietary software product.

When we explain to the employer that it is illegal to distribute the improved version except as free software, the employer usually decides to release it as free software rather than throw it away.

To copyleft a program, we first state that it is copyrighted; then we add distribution terms, which are a legal instrument that gives everyone the rights to use, modify, and redistribute the program's code, or any program derived from it, but only if the distribution terms are unchanged. Thus, the code and the freedoms become legally inseparable.

Proprietary software developers use copyright to take away the users' freedom; we use copyright to guarantee their freedom. That's why we reverse the name, changing “copyright” into “copyleft.”

Copyleft is a way of using of the copyright on the program. It doesn't mean abandoning the copyright; in fact, doing so would make copyleft impossible. The “left” in “copyleft” is not a reference to the verb “to leave”—only to the direction which is the inverse of “right”.

BSD licenses are a family of permissive free software licenses, imposing minimal restrictions on the redistribution of covered software. This is in contrast to copyleft licenses, which have reciprocity share-alike requirements. The original BSD license was used for its namesake, the Berkeley Software Distribution (BSD), a Unix-like operating system. The original version has since been revised and its descendants are more properly termed modified BSD licenses.

The BSD License allows proprietary use and allows the software released under the license to be incorporated into proprietary products. Works based on the material may be released under a proprietary license as closed source software.

Software vendor perspective on software licenses:

- GPL (GNU General Public License) is NOT suitable. In this scheme, the source code must be made available, and any derivative works must be GPL too. Use in commercial software is allowed, but every other component must be made GPL too. For this reason, some call it a 'virus' or a 'plague', as it "infects" anything it touches (i.e. turns into GPL) in order to propagate freedoms. If a software vendor would include a GPL building block of a third-party into its software tool, the reciprocity model claims to open all IP of the code that uses the GPL building blocks. Software vendors can NOT afford to do that, given the multi-year investments done in the software, and its value for future business.
- LGPL is better than GPL, but poses a commercial constraint as only "dynamic" linking is commercially viable, which has a performance disadvantage. LGPL (or Lesser General Public License) is a version of the GPL that – only in dynamic linking – keeps the original software free but allows other parties to combine it with software that is not as free:
    - A standalone executable of commercial software that dynamically links to a LGPL tool library (e.g. through a .dll) is generally accepted as not being a derivative work as defined by the LGPL. It is then possible for the commercial software to be linked with a newer version of the LGPL-covered program (typically by downloading a new LGPL tool from internet and putting it in an installation folder of the commercial software). With "dynamic linking", the LGPL code then stays LGPL, and you can keep the proprietary commercial software code proprietary.
    - When the new tool block is a piece of code embedded within the code of the commercial software, such that 'dynamic replacement' with a new version is not possible, one speaks of "static linking" – this has the same disadvantages as GPL in terms of IPR, that both parts must be released as LGPL (which is not acceptable for commercial software vendors, given the multi-year investments in and business value of their tool chain).
    - Performance-wise, there's a disadvantage in dynamic linking as compared to static linking (as dynamic linking and reading in the component at first-time use and at re-use is typically more expensive in terms of CPU time as compared to static linking). This means that the LGPL scheme has a disadvantage as compared to other (more permissive schemes) as BSD or Apache.
- Indeed, many other licensing schemes are far more permissive and are perfectly acceptable for most commercial tool vendors. E.g., BSD or Apache licenses allow software components to be integrated in a derivative product by a/o including a copy of the BSD or Apache license text, but without changing the licensing scheme of the derivative product.

Hence, if in the prototyping phase there's no clear preference, we recommend a permissive scheme that is acceptable to commercial tool vendors downstream, such as BSD or Apache.

Appendix II

Software writing best practices courses, references and documentation available on-line

Examples (to be expanded):

1. Special issue of ERCIM Newsletter on 'Software Quality': http://ercim-news.ercim.eu/en99
2. Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, et al. (2014) "Best Practices for Scientific Computing." PLoS Biol 12(1): e1001745. doi:10.1371/journal.pbio.1001745

3. Ian Sommerville, "Software Engineering", 9th ed. Addison-Wesley Publishing Co, Reading, MA, 3 March 2010 -

4. Software Sustainability Institute - http://www.software.ac.uk/
   http://www.software.ac.uk/resources/top-tips
5. Software Engineering Support Centre (SESC) - http://www.softeng-support.ac.uk
   SESC Workshop 2014 - http://www.softeng-support.ac.uk/news/2014/11/06/Report-SESC-Workshop.html
6. Originally Canadian initiative called Software Carpentry - http://software-carpentry.org/

7. CECAM Workshops; e.g. 2014 - http://www.cecam.org/workshop-0-198.html; 2015 - http://www.cecam.org/workshop-1214.html
8. PSI-K workshops http://www.psi-k.org/workshops.shtml
9. CINECA training workshops material
   http://se4science.org/workshops/se4hpcs15/resources.htm
10. CCPForge – coding standard
    http://ccpforge.cse.rl.ac.uk/gf/project/ccpforge/docman/Developer%20Documentation/
11. ABINIT software 'code of conduct' best practices guide for contributors - https://wiki.abinit.org/doku.php?id=developers:coding_rules
12. HPC Consortium Autumn Academy - http://www2.warwick.ac.uk/fac/cross_fac/hpc-sc/courses/academy/
13. UK HPC service http://www.archer.ac.uk/documentation/best-practice-guide/
14. CCP5 - http://www.ccp5.ac.uk/ does not have a listing but from 2014 started to give two course in SoftEng:
    1. Fortran95 - day 1 and 2 for beginners, day 3 for advanced users
    2. 'Hack days' for some of the codes displayed at http://www.ccp5.ac.uk/software/