

Situated Design and Universal Maintenance:

A Software Evolution Pattern Inspired by the Free/Libre Open Source Software Development

by *Yuwei Lin** and *Enrico Zini***

* ESRC National Centre for E-Social Science, University of Manchester

**Debian GNU/Linux

1. Introduction

Attempts of sharing and reusing software tools are growing in today's ICT design world, and open sourcing software seem to become a good practice for this end. Although some people have reservation over the quality and usability of free/libre open source software (FLOSS), there are plenty lessons we can learn from the evolution of FLOSS. Here, we suggest a software evolution pattern named 'Situated Design, Universal Maintenance' (SDUM) based on our observation on the FLOSS development. It says that software has to be (re)situated for local user needs and maintained universally. Instead of treating often-criticised issues about the FLOSS development such as release criteria and programming for the self as nuances, we argue that these issues are normal in an open development environment, which can be adequately illustrated by the SDUM pattern. By nature of such an open development environment, these issues will be resolved over the development process if collaboration between developers and users distributed around the world is successful.

2. Situated Design, Universal Maintenance

Informed by a variety of literature on 'user-centred design' (UCD) and 'universal design' (UD), the SDUM pattern, combining both streams of thoughts, suggests to design situated software for a specific group of users in the beginning of a development process, and encourages extensive usage and reusage of its code to aim for universal use. This pattern makes the coexistence of UCD and UD possible by bringing them into play in different phases of software development. UD-maintained code then becomes the shared code that holds together different UCD development efforts.

Open sourcing software can help maintain software development in a sustainable way, but continuous redesign is needed when the software is adopted and used in other contexts.

2.1. Why UCD+UD?

Both the user experience and the technical aspects of software. In a UCD-led design, users and their tasks and goals are emphasised for technological development and design. In UD-led design, a wide range of individual preferences and abilities are accommodated. UCD is specific for users in that leads to software that is designed to effectively solve one of their problems; UD is flexible because it leads to software that can be applied to different problems of different users, albeit less efficiently. If we see software development as a long-term and reiterating process

during which the concepts of usability and usefulness, users' identity, goals, experiences, expectations, and environments where they reside are constantly defined and redefined, a piece of software should be better designed following a hybrid UCD+UD model, using both UCD and UD strategically depending on the stage of its development.

2.2. Methodology

The pattern takes the idea of expansion/refactoring of Extreme Programming [1], and reuses it to accommodate both UCD and UD under one framework (i.e. a SDUM-based one). In the process of building this pattern, we also integrate our observations on and experiences of working with FLOSS and/or developing FLOSS. This pattern thus is informed by both empirical and theoretical materials.

2.3. The Description of the Pattern

We have observed that software development usually starts from designing for specifically identified audience at the very beginning of the design phase (e.g. a prototype), and later on makes the designed artefacts as accessible as possible. Situated Design keeps the design of new features grounded on a specific need. This would help adding new features or solving local problems easier.

Universal Maintenance makes functionalities more coherent through maintaining and consolidating resources universally. The universalisation will gradually make the software perfect and entice new adopters, opening up a new round of Situated Design.

An UCD-UD approach is not only beneficial for users but also for developers. Having reusable code to support various situated designs means software can be maintained by people with different needs. Diverse actors have freedom to focus on specific, user-centred design while at the same time cooperation happens in the maintenance of the universal building blocks used as a basis.

Overall, software that qualifies the description of the SDUM pattern must contain three aspects:

1)Unlimited lifetime: the lifetime of the software should be potentially unlimited, or at least long enough to fit multiple iterations of development and maintenance.

2)Reuse: it must allow reuse the software or part of it

3)Extensibility: The software must allow expansion motivated by the effort of maintenance towards universality.

SDUM tries to balance the complexity and simplicity in one design. Instead of implementing right from the beginning every possible feature in one software (which might lead to overdesign), or releasing an incomplete piece of abstract infrastructure for others to build on, SDUM suggests to

properly implement what is needed, and allow the software to be reused and become a building block for new features.

While adding new features, development should remain grounded on local needs in order not to lose focus. In other words, universality is enabled through the accumulation of various local developments; maintenance towards universality is a process of engaging local activities of improving the quality and lifetime of the software.

2.4 Phenomenon

We have observed some phenomenon related to SDUM:

* Community-based development

SDUM allows a diverse range of people to work on the same project. As many have seen, the FLOSS innovation model fortifies the formation of a community and encourages a kind of community-based development that is more likely to lead to a sustainable design. Users and developers who share the same interest in specific lines of work would eventually get together. Through engaging a range of very different know-how on the software development, innovation derived from the same software can be taken place in many different fields.

* Reduced development cost

SDUM encourages collaboration which in turn distributes development costs. In the FLOSS field, the common practices of peer review, bug reports, feature requests and patch submissions all contribute to the development of the software.

* Trust and collaboration

While SDUM enables and exploits beneficial coexistence of many and diverse forces in the development of a software, it brings into play all the range of issues connected to group dynamics: working together on a single piece of software requires some level of cooperation, where the contributors need some level of trust among each others, where development of new features can be free, but maintenance has to be carefully negotiated and coordinated, paid/funded/sponsored, with mediation across different views.

3. Case studies

Considering the FLOSS development over the past two decades, we have observed that most of the FLOSS as a kind of 'situated software' [2] that was designed for users and often by users in a very specific context. This is claimed to be a problem with FLOSS according to Levesque [3]. She argues that open source programmers often tend to program with themselves as an intended audience, rather than the general public. But, what Levesque fails to see is the continuous evolution of software. Because when more developers found a piece of FLOSS useful, it gets further developed to be deployed and implemented in other environments. Another problem about

release criteria raised by Glance [4] who questions the released Linux Kernel code involves a largely unknown level of quality can also be resolved over the universalisation process which accompanies with many customised and situated cases. Software quality is controlled collectively by all participants (users and developers) and negotiated over the evolution, which is a long, iterative and probably endless process.

This phenomenon seems to characterise a common software engineering practice: the original design and production of a program are for solving a specific problem, whilst later maintenance on and refactoring of it usually lead to modularisation and generalisation of the internal structure as well as the interface of the software. And this is what the SDUM pattern illustrates.

The development of many FLOSS projects has corresponded with the SDUM pattern, for instance, the development of GIMP (GNU Image Manipulation Program). And a cross-dimension feature of software (across universality and specificity) an issue about forking and customisation emerge in looking at the SDUM pattern.

3.1. Universality and Specificity

A cross-dimension feature of software has been observed in this SDUM pattern. Such feature is illustrated with the case of GIMP.

Started as a student assignment for a computer science course at the University of California Berkeley in 1995, GIMP has since then evolved towards universality, becoming a popular FLOSS image manipulation tool. Over years of development, its user interface components have been modularised and evolved to become the GTK libraries at the core of GNOME1 Desktop Environment. While the application becomes popular and is deployed in various environments, a customised version, Film Gimp (later 'Cinepaint'), emerges with contributions from the Hollywood entertainment industry to allow editing the frames in a movie2. The evolution of GIMP is illustrated in Figure 1, at the end of this paper.

Software such as GIMP that can be deployed in a wide range of diverse environments becomes an universal patrimony available for reuse. Such universal software enjoys the advantages of high maturity, availability of know-hows and shared cost of maintenance, but tends to get abstract and disconnected from its context of use. Therefore, it in turn becomes a building block for a new application for a targeted group. Some consecutive development of the new version may also improve the universal patrimony when continuing to serve a defined group of users. Such a software design process that spans universality and specificity can be conceptualised in the diagram shown as Figure 2, at the end of the paper.

And the connection between a universal system and originated and derived situated software is illustrated in Figure 3.

3.2. Debian Forking and Customisation

FLOSS that follows a SDUM pattern also faces challenges of forking and

customisation. Debian GNU/Linux operating system (and project) is one of the successful cases that cope with this challenge well.

The Debian GNU/Linux project currently engages around 1000 developers around the world on maintaining more than 16000 different software packages and creating new software. These members and other individuals and organisations interested in Debian enjoy a privilege of sharing and reusing a universal body of code to support their everyday software activities. Debian has a wide variety of derivative distributions (around 129 according Distrowatch.com). Making derivatives from Debian, however, is inefficient because the derived usually gets disconnected, completely or in part, from the activities within Debian itself for the sake of pursuing customisation. Custom Debian Distributions (CDD) project, thus, is initiated to make customisations of Debian fully maintained within Debian itself. The development of CDD echoes the SDUM pattern in allowing situated development to take place around a commonly maintained code.

While the meta level of Debian GNU/Linux distributes free software for users in the wide, 'CDD aims to provide a simple-to-learn environment that not only greatly enhances the value of the software, but also address the special skills, needs and interests of targeted users' [5]. And because these CDD projects (e.g. Debian-Edu) usually have a dual goal of improving Debian as a whole and solving the need of the target group, knowledge is exchanged fluidly between developers and users across projects and levels by means of tools such as bug tracking systems (BTS) and mailing lists. As such, the strategy of becoming a UCD-ed and UD-ed system increases interoperability of different packages and allows the modification of the system to suit the user's needs. On the one hand, Debian maintains standardised or open file formats with the specifications of which can be utilised by any system, including proprietary ones, but on the other hand, Debian is specialised to meet the needs of particular groups of users.

3.2.1 Debian-Edu

Debian-Edu, is one of the earliest and prominent custom Debian distributions. It is developed to fulfill the special requirements of educational organisations. Since 2003 it merged with SkoleLinux3, a successful Norwegian-led project to build, install and support a FLOSS-based operating system for schools.

Debian-Edu is customised to be simple, easy-to-use and user-friendly. It comes with a set of default features and contains major and essential FLOSS projects such as the Linux kernel, OpenOffice.org, Mozilla Firefox, the Apache server, the Samba server, as well as smaller tools that are often used by teachers and students for teaching and learning, like DrGenius4, Celestia5, FET6 and Cerebrum7. All these software are packaged and ready for people to use and build an IT infrastructure on it. It is observed that the more these software programs are used, the faster the paces of their evolutions are, because they get feedbacks, documentation and even code contributions from their users communities.

SkoleLinux has been funding various development efforts. One of them is a function of automating the creation of lab user accounts when a new

teacher or student gets registered at the school computer lab in the secretary archive. This function is very unique to schools because normally school computer labs are separated from the rest of the school administration and bridging the two things is a complicated task. In providing such a function, it not only solves a specific need of schools, which requires careful knowledge of the school workflow and internal systems, but also innovates Debian with a new component.

Through the collaboration with Debian, SkoleLinux has enjoyed several advantages including:

- development is done publicly together with a community of developers, increasing the likelihood of external contributions and peer review.
- maintenance costs are shared
- the groupwork inside Debian is an open channel for new know-how to be circulated
- the three Norwegian translations can be maintained by other Norwegian Debian users not involved in schools
- Debian users working in schools outside of Norway can get to know SkoleLinux, trying out its software and eventually get involved. Now Debian-Edu (merged with SkoleLinux) has branches in Belgium, Brazil, Denmark, Germany, France, Latvia, Norway, Spain and Turkey.

Given the example of Debian and Debian-Edu, the idea of customising software is crucial exactly because once a custom software is made, not only users can benefit from a specially designed product (e.g. Norwegian schools and SkoleLinux) and work more efficiently with it, but also the overall matrix software is advanced (e.g. Debian benefited from having special features tailored for schools) because of the localisation of the software.

4. Conclusion

What we have learned from the SDUM pattern is that (successful) FLOSS that follows a SDUM pattern has to survive a long lifetime, to allow reuse, to allow expansion beyond original developers' intentions. And then the software will mature as both cross-dimensional (universality and specificity) and receiving many forked and customised versions. Eventually, this will result in a community-based and economic development. Knowledge and experiences can be shared amongst all participants.

The SDUM pattern implicates in a rethink of software development and deployment methods. Instead of creating completely new software and releasing it under FLOSS licences, a wiser way of developing software tools is to find available resources, reuse and customise them. In so doing, the customised software is situated in local users needs and still connects to the universal system where open source code is stored in a large repository. The pattern and cases we have seen suggest that future software developments should harness on joint efforts and share development and maintenance costs in establishing a community of developers, users and sponsors.

Whilst the SDUM pattern proves the long-term benefits of designing long lived software that are aimed to be reused and extended beyond original intentions, it also shows the limited knowledge we have about socio-technical dynamics in distributed, community-based and user-centred software development processes. Future studies should look into socio-technical questions in software development, such as how to customise software for users' situated needs, relationship and communication in the collaboration between users and developers, mechanisms of sharing knowledge between all participants, and decision making in recycling and reusing a piece of code.

References

- [1] J. Highsmith, Extreme programming: Agile project management advisory service white paper. Cutter Consortium, 2002.
- [2] C. Shirky, "Situated software", http://www.shirky.com/writings/situated_software.html, 2004 (Accessed 6 July 2006).
- [3] M. Levesque, "Fundamental Issues with Open Source Software Development", First Monday 9(4), 2004. URL (retrieved on 6 July 2006): http://firstmonday.org/issues/issue9_4/levesque/index.html
- [4] D. G. Glance, "Release Criteria for The Linux Kernel", First Monday 9(4), 2004. URL (retrieved on 6 July 2006): http://firstmonday.org/issues/issue9_4/glance/index.html
- [5] A. Tille, "Custom Debian distribution", <http://people.debian.org/~tille/debian-med/talks/paper-cdd/debian-cdd.html>, 2004. (Accessed 30 November 2005).

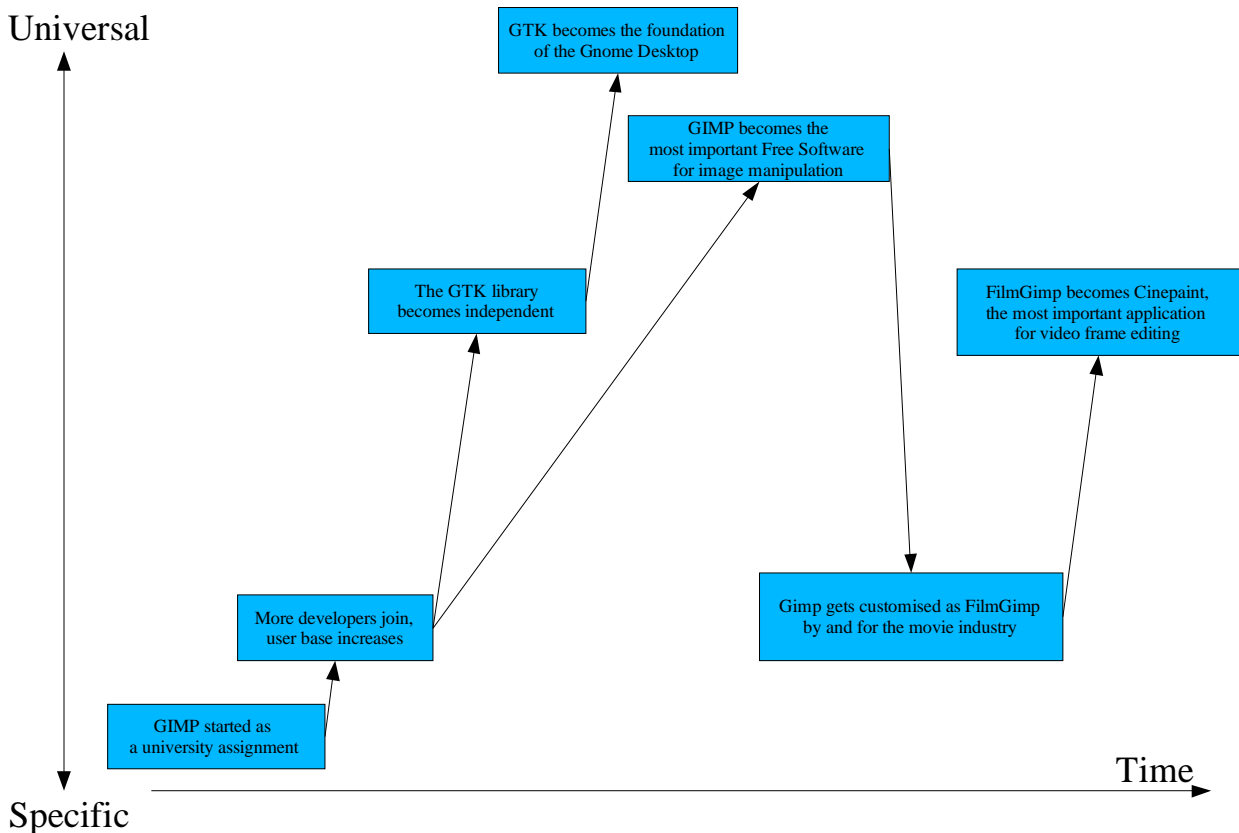


Fig. 1 The evolution of GIMP

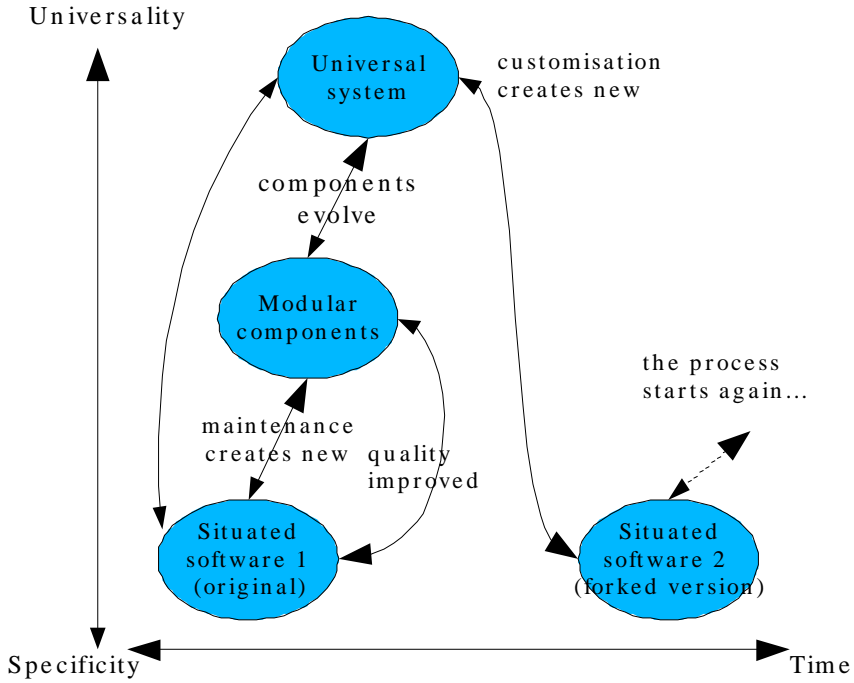


Fig. 2 The Evolution of Software Suggested in SDUM

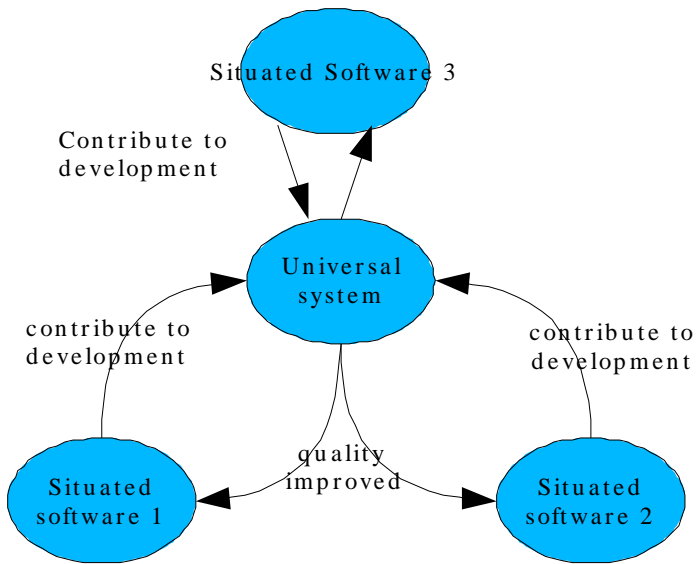


Fig. 3 Feedback Loops of Collaboration suggested in SDUM

