An empirical study of the evolution of an agile-developed software system

A. Capiluppi
University of Lincoln
acapiluppi@lincoln.ac.uk

J. Fernandez-Ramil
The Open University
j.f.ramil@open.ac.uk

J. Higman
Independent Agile Coach
London, UK
jhigman@pobox.com

H. C. Sharp
The Open University
h.c.sharp@open.ac.uk

N. Smith
The Open University
n.smith@open.ac.uk

Abstract

We have analyzed evolution patterns over two and a half years for a system developed using eXtreme Programming. We find that the system shows a smooth pattern of growth overall, that (McCabe) code complexity is low, and that the relative amount of complexity control work (e.g. refactoring) is higher than in other systems we have studied. To interpret these results, we have drawn on qualitative data including the results of an observational study, records of progress and productivity, and comments on our findings from team members.

1. Introduction

Approaches to software development include both agile and plan-driven methods [1]. Agile methods [2] emphasise flexibility, informal collaboration and working code. Plan-driven approaches emphasise large-scale planning, formal communications and documentation. Boehm & Turner [1, 3] proposed five critical risk dimensions (size, criticality, dynamism, personnel and culture) that organisations should consider when deciding which method to use. While the focus of Boehm & Turner appears to be the situation before and during initial development, surveys suggest that the majority of developer effort occurs in maintenance and evolution [4]. Hence, an additional dimension that needs to be studied is what happens during maintenance and evolution. (For simplicity, we consider 'maintenance' and 'evolution' to be synonymous terms for all work performed after the first release of the software.) Empirical evidence about evolution may help organisations to decide what type of method to pursue.

There have been studies [e.g. 5, 6, 7] of the evolution of software developed using some form of plan-driven methods (e.g. following the waterfall process and its variants). Topics such as the laws of software evolution have been discussed over more than 30 years [8, 5, 7]. For example, it is argued that, as changes accumulate, complexity growth is inevitable in evolving software [e.g. 8, 9]. It is also argued that a level of complexity reduction work (e.g. refactoring [10]) is required in order to sustain long-term evolution. However, little is empirically known about evolution using agile methods.

Proponents of agile processes argue that such processes should be able to respond to change and to withstand the continual pressures driving software evolution better than plan-driven approaches. Sustainability of the evolution of code is an explicit objective in agile teams, and refactoring is seen as something necessary and positive. These claims need to be tested through studies of software evolved using agile methods. This requires both qualitative and quantitative observations whose collection and study is difficult.

We are aware of only two publications discussing the evolution of agile software, neither of which used measurements from agile processes as empirical evidence. Wernick & Hall [11] argued that pair programming should be beneficial for long-term evolution. Chapin [12] considers the different types of stakeholders and concludes that the effects of agile methods may be more positive for some than for others. To our knowledge, the findings presented here forms the first measurement-based study of the evolution of software developed using an agile approach in an industrial setting.

In this paper, we present findings from an initial exploratory investigation into the evolution of one code base developed using eXtreme Programming (XP) [13], an agile method, over the period October 2002 to March 2005. The aim of the study was to describe the evolution of a commercial software system developed with an agile approach. It was not our intention to draw lessons of best practice from this single observation. In our analysis, we use measurements and time series displays that we have used in previous studies of non-agile object-oriented systems [e.g. 14].

To complement this analysis and to contextualise the quantitative data, we draw on several sources of

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

qualitative data: the team's records of progress and productivity; notes from their retrospectives [15], an observational study of the team conducted at the beginning of 2002 [16], and comments from team members responding to the results of our analysis.

1.1 eXtreme Programming

XP is an agile development method, and all agile methods conform to the agile manifesto [2]. The emphasis of the agile manifesto is different from traditional software development: individuals and interactions are valued over processes and tools, working software is valued over comprehensive documentation, customer collaboration is valued over contract negotiation, and responding to change is valued over following a plan.

Some may dispute the detail but the sense of XP practice is captured in this description by Cockburn [17, p29, original emphasis]:

> "It calls for all the developers to sit in one large room, for there to be a usage expert or 'customer' on the development staff full time, for the programmers to work in pairs and develop extensive unit tests for their code that can be run automatically at any time, for those tests *always* to run at 100% of all code that is checked in, and for code to be developed in nano-increments, checked in and integrated several times a day. The result is delivered to real users every two to four weeks[1]
>
> "In exchange for all this rigor in the development process, the team is excused from producing any extraneous documentation. The requirements live as an outline on collections of index cards, and the running project plan is on the whiteboard. The design lives in the oral tradition among the programmers, in the unit tests, and in the oft-tidied-up code itself."

While its acceptability is contested, many organisations have successfully adopted the approach, and its popularity is growing.

1.2 The case study organization

The case study organization is a small company developing web-based intelligent advertisements for paying customers. The software analyses the content of the current web page to determine the user's interest. The software then displays an advert relevant to this interest. For example, if the reader is looking at a page about childcare then an advert for the latest baby buggy might be displayed; if they are reading about home improvements then an advert about power tools might be displayed.

The company started in May 1999, and has used all 12 practices (see Table 1), originally proposed in Beck's seminal book [13], since they first formed. This was verified during an observational study of the team during 2002 [16].

During the time covered by the code base, the team consisted of between 2 and 10 developers, one graphic designer and one person who looked after the infrastructure. The company employed around four marketing personnel who determined what was required in collaboration with clients. Marketing personnel were regarded as being, in effect, the customer. Throughout the study period, the code base was being constantly updated, extended, developed, and maintained. The developers made no distinction between these activities.

At the time of writing, the company is still operating, but has changed considerably since these observations. However, the software is still in active commercial use and the company remains profitable.

2. Source code evolution analysis

This application was implemented in Java, initially using the Visual Age IDE before migrating to the Eclipse IDE [18] in October 2002. Source code is only available from this date. As standard change management practice, a configuration management (CVS) code repository was integrated with Eclipse and used by the team to store the code and keep control of versions and changes. A full copy of the entire code repository was provided to us for this study. The code base consists of approximately 130,000 lines of original source code. Of these, 90,000 lines of code (approximately 70% of the methods) consist of unit tests. Because we are interested in the output of this agile team, only the software developed by this team was considered. The third-party code libraries used for this product (some 400,000 lines of code) were excluded from our analysis.

We analyzed the code and extracted metrics using our own tools. The information about the code check-ins made by the pairs and the code base, including unit tests, were sampled with four observations per month. For each sample, the code's size and other metrics were measured. This data spans about 2.5 years of the evolution of this product.

In the quantitative data displays below we use relative numbers to keep confidential the actual characteristics of the product.

[1] This chunk of development is called an 'iteration'; many companies have one-week iterations

In order to analyse quantitatively the evolution of the system we used the typical measurements in this type of studies such as size [19, 5], elements handled (touched) [5], complexity [20], and the amount of anti-regressive work [8, 21, 22, 23].

2.1 Size of the system

Size was measured by counting the number of source items. Figures 1a and b show the relative size of the agile system, over time, measured in number of files and number of lines of code (LOC). Growth is positive during most of the period studied. There is a decrease in growth rate between March and May 2003, depending on the measurement (files, LOC) used. In April 2003 the company reorganized and the number of developer pairs dropped from 3 to 1. Growth rate recovers towards the end of the period studied. In a 'classic study' with limited contextual inputs this growth pattern would have been interpreted as 'growth constrained by increasing complexity with superimposed stabilization ripples' [e.g. 24]. However, as we will see in section 4.1.1 below, the interpretation here is quite different.

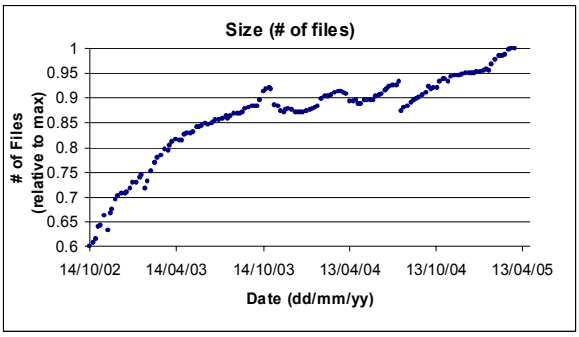
Figure 2 presents the amount of work measured as the number of files handled (i.e. added or changed) over time. The linear trend resembles that of the 'classical' behaviour [5] in which cumulative evolution work appears to follow a predictable constant linear rate.

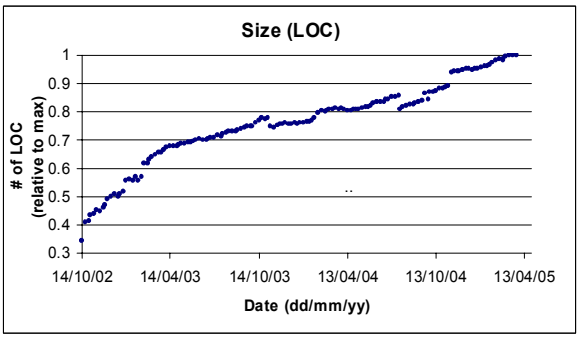
Table 1. XP Practices

The Planning Game – Quickly determine the scope of the next release by combining business priorities and technical estimates. As reality overtakes the plan, update the plan.
Small releases – Put a simple system into production quickly, then release new versions on a very short cycle.
Metaphor – Guide all development with a simple shared story of how the whole system works.
Simple design – The system should be designed as simply as possible at any given moment. Extra complexity is removed.
Testing – Programmers continually write unit tests, which must run flawlessly for development to continue. Customers write tests demonstrating that features are finished.
Refactoring – Programmers restructure the system without changing its behavior to remove duplication, improve communication, simplify, or add flexibility.
Pair programming – All production code is written with two people at one machine.
Collective ownership – Anyone can change code anywhere in the system at any time.
Continuous integration – Integrate and build the system many times a day, every time a task is completed.
40-hour week – Work no more than 40 hours a week as a rule. Never work overtime a second week in a row.
On-site customer – Include a real, live user on the team, available full-time to answer questions.
Coding standards – Programmers write all code in accordance with rules emphasizing communication through code.

2.2 Measurement of complexity

There are a number of different ways of measuring software complexity [25]. We used the McCabe cyclomatic number [20] as a measure of complexity and used the accepted threshold value of 15 [26] to differentiate between high and low complexity methods. This measure does not address software complexity arising from inter-method sources, such as coupling. We have previously studied a number of open source systems, including some developed in Java, and we have found that in all of these systems between 5% and 10% of the methods have high complexity [23]. In this system, there were at most two





Figures 1a (top) and 1b (bottom). Total system size including unit tests, in number of files (top) and LOC (bottom) over time, relative to the maximum size achieved in period studied

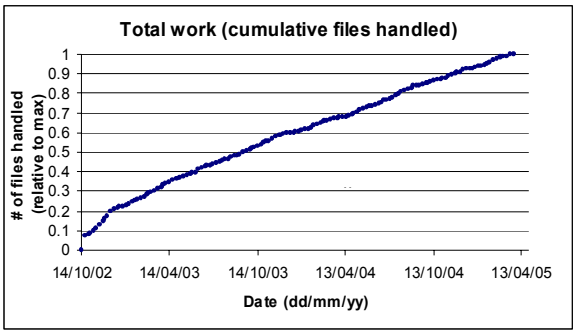


Figure 2. Cumulative work in number of files handled per week

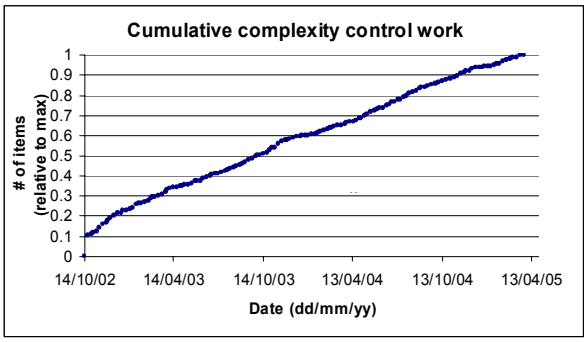


Figure 3. Cumulative complexity control work

complex methods, a tiny proportion of the whole, from October 2002 to January 2004, with none after that.

We estimated the cumulative of complexity control work by comparing every method between two consecutive releases (or weeks) and by counting how many of them experienced a reduction in their cyclomatic complexity number. In this way, the number of methods that experienced a reduction becomes a surrogate for the amount of complexity control work. From a quantitative analysis, Figure 3 illustrates the rate at which the amount of complexity control work accumulates. The trend in figure 3 approximately follows the trend of the total work in figure 2, suggesting that the amount of complexity control is roughly constant and similar to the level of work. This is similar to what we have found in other non-agile systems [27, 23].

In order to estimate the relative level of complexity control work, we divided the number of complexity decreases observed by the number of files handled (see figure 4). The level of complexity control work is about 46% of the total work on average, but the amount varies widely over time. The 46% figure is higher than what we have measured in any other system that we have studied; it is typically below 10% of the total [22, 23].

3. Analysis of the wiki records

During the early phases of the system evolution (September 1999 to March 2004) the developers used a wiki (see figure 5 for an example) to maintain a record of activity in each iteration. Wikis are often used within agile teams to maintain information about source code, or about development processes. Maintenance of the wiki was usually delegated to one member of the team at the end of each iteration. We focus our analysis on the entries that overlap with the code base availability. The data collected were:

1. Stories implemented in this iteration, including implementation time;
2. Dates of iteration start and end;
3. Number of pairs working;
4. Project velocity (the number of productive days of work in each iteration), expected and actual;
5. Load factor, expected and actual. This is the ratio between estimated and actual times to complete each story. This was rarely reported for our period of focus.

Figure 6 displays two of the extracted time series, the number of programming pairs and velocity over the 77 weeks of data overlap. After the company restructured in April, much less data was captured and the wiki data should be considered less reliable. However, the apparent velocity of the project does not seem to change after this restructuring. The outlier value for velocity of 20.5 recorded at this same time is the result of velocity being recalibrated and apparently being recorded inaccurately for that iteration, i.e. for the whole three-week iteration rather than for each week, as previously.

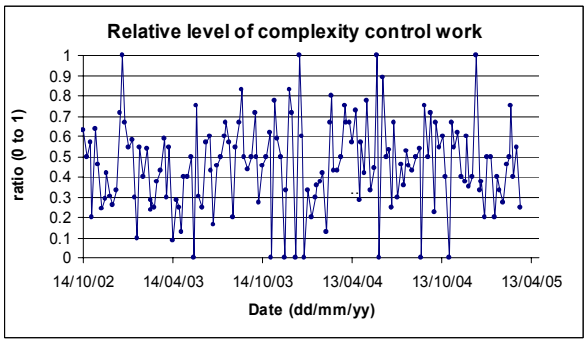


Figure 4. Portion of file handling events resulting in method complexity reduction

KEV-Iteration21

Edit | Refresh | Home | Changes | Find - Last edited July 5, 2001

ProjectCommitmentSchedule | SoftwareDevelopment | <- KEV-Iteration20 | KEV-Iteration22

Selected Stories

- Finish [Client 1] 3.25
  - many small tasks ...
- [Client 2]? 0.75
  - Forced Upgrade 0.0
  - No Registration 0.5
  - Service Hosting Transfer 0.25
- [Client 3] 2.25
  - Context based headlines 0.75
  - Ask The Expert headlines 1.0
  - Scrolling news headlines 0.5

Additional Stories

- Bug Fix: Desktop Space Reservation 0.25
- WinTask: Desktop Space Reservation Testing 0.25
- ActiveAd 1 Server - Multiple Ads 1.0

Removed Stories

- [Client 2]? Adverts 0.25

See: SidewizeChangeHistory

```
Started: 13/7/2001
TeamPairs?: 3 (1 pair on dev issues)
ProjectVelocity: 6.5
LoadFactor: 6.9
PredictedEnd?: 5/7/2001
Ended: 5/7/2001
ActualProjectVelocity?: 7.75
ActualLoadFactor was: 5.8
```

Edit | Refresh | Home | Changes | Find - Last edited July 5, 2001

Figure 5. An example wiki page showing the data for one iteration

4. Discussion

4.1 Results from combining quantitative and qualitative analyses

The main findings from the analysis of code base were the following:

1. Smooth growth (with small perturbations) was seen in the evolution of this agile system.
2. In relative terms, the rate at which complexity-control work progresses averages to 46% the total work rate.
3. There are almost no highly complex items.
4. Growth rate measured in lines of code is higher than growth rate measured in files or directories

Each of these is discussed below, drawing on our qualitative data to provide contextual information and to help explain our findings.

4.1.1 Smooth growth with small perturbations. One of the key findings from the wiki records is that the team underwent a considerable restructuring in April 2003. The reduction and re-organisation is not reflected in the analysis of the code base except that

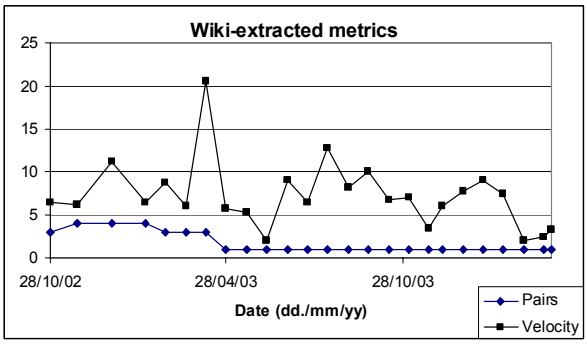


Figure 6. Wiki-extracted measurements from Oct 2002 to March 2004 (77 weeks)

there is a temporary drop in growth rate at this time. However the growth rate picks up and continues at a similar rate (although with more 'ripples' than before). Given that the staffing is reduced from 3 pairs to one over the course of one iteration, this is a finding worthy of explanation.

We put this question to the chief technical officer at the time and his responses shows that although the team reduced, the nature of the work they were performing also changed:

> "In terms of growth rates being high with one pair, it's possible that there were clearer product development targets in that period, where we had latched onto an idea, and were building the platform as quickly as we could…. the product had become more defined, with a "workbench" web application that grew quite rapidly (that would have included quite a lot of html templates and other non-code artifacts)
>
> "Alternatively, it's possible that fewer pairs leads to less refactoring, which leads to more lines of code for the same amount of functionality (less time spent stripping out redundant or duplicate code)."

We hypothesized that the other slight changes in growth rate may have been caused by business cycles or other reasons, but this has been refuted by team members.

> "There was some seasonality in campaigns that we ran for clients - for example, [some] clients tend to wind down their advertising a bit over the summer, when there's less [activity] happening - but I'm not sure why October, November and December would be significant."

Another suggestion is that the rate is more sensitive to the particular circumstances of one pair. For example, if one of the pair is unwell, or is needed to work on other matters, this would affect the growth rate of the system. However we have no evidence to support or refute this idea.

4.1.2 Complexity control work. From our qualitative data, we know that the team was committed to coding and code quality, and that they took refactoring seriously [16]. As refactoring often acts to reduce the complexity of individual methods [10], it is not surprising that complexity of code was kept low and that the level of complexity control work is high.

Notes recorded during the observational study provide two sets of evidence. First, the team members actively discussed how and when to perform refactoring as part of their daily stand-up meetings. Second, an episode was recorded where a team member became frustrated because the estimate allowed for the current piece of code being developed did not allow for refactoring of existing code, hence the refactoring had to be postponed. In fact, other pairs of developers were carrying out changes that affected related classes and it was agreed that refactoring should wait until all of the changes had been implemented.

There is also evidence for the team's attitude to refactoring in the retrospective notes. The retrospectives invited team members to discuss four questions relative to the preceding iteration: what we did well, what we learned, what puzzles us and what we can improve. For the iteration in August 2001 refactoring is noted as having been done well (although refactoring of tests could be improved). During September of the same year, refactoring of one particular class is highlighted as something to improve.

4.1.3 Lack of complex items. The measure of complexity we have used may not be ideal for object-oriented systems. However, we have used the same measures for other object-oriented systems developed through traditional plan-based [27] and open source methods [23]. When comparing this agile-developed system with results from our other studies, we find that the level of complex items is noticeably lower. For example, the open source systems we analyzed (also written in Java) have, on average, some 5% to 10% highly-complex functions [21]. Other commercial code bases (mostly C++ and Delphi) analyzed recently shows similar results [27]. The almost complete lack of complex methods in this system is striking.

4.1.4 Growth rate in lines of code. One interpretation of the difference in growth rate between measurements in LOC and files may be that methods tended to be long, and that more classes should have been developed. However, when we put this point to one of the team members, they described their coding style thus:

> "We typically used short methods so might have more private methods as a result of refactoring. Also we used verbose coding style and no comments."

Hence, the increase may be due to the 'verbose coding style'

4.2 Relating evolution at the technical and business level

Overall, the quantitative measurements presented in section 2 suggest smooth evolution (apart from the reduction in growth rate in April 2003 due to company restructuring) with a stable rate of growth and work (growth and change), even though the team never used these or similar measurements. The low number of complex methods and high level of complexity control work is what one would expect from a software evolved using agile methods. Despite the decrease in system growth rate, the sustained rate of work towards the end of the period studied, with only one pair working at that time, is surprising (could have not been predicted by simple extrapolation of past measurements) and deserves further investigation. The qualitative analysis provided some clues.

Evidence for success of the team as an XP team is available from the observational study [16]. For example, the managing director reported that their clients were impressed with the agile approach because of the responsiveness of the team to their needs.

From a business point of view, the product studied in this paper was a success commercially in that it survived well in the market for 6 years, continuing to use XP for the whole of that time. The original company has now been taken over, but the products continue to be marketed. The downsizing in April 2003 was a deliberate strategy to release funds that were tied up in salaries to allow the product to be marketed more effectively and hence to grow the business. Without the previous intense development and evolution efforts they would not have had a product to market.

5. Threats to validity

Any empirical study confronts threats to the validity of the results. We include below a list of the threats that appear to be relevant to this study:

1. This study has only looked at one software system. It is an initial, exploratory study which hopefully will be followed by others. As this is a single study of a single system, care must be taken when attempting to generalise the observations made here to other situations.

2. We extracted the data using our own software tools. Despite our best efforts, errors in the data gathering and measurement extraction are possible and we cannot guarantee that our tools are error-free.
3. For various practical reasons, the qualitative data collected do not coincide exactly with the period of development covered by the code base. However, through continued contact with the organisation and individual members we know that the team membership and company culture remained consistent over the period of code development.
4. The use of McCabe for measure the complexity of object oriented software may not reflect the *delocalisation* [28] of functionality and, because of this, it may underestimate the real complexity. There is a need for measures of complexity that address this issue.

6. Conclusions and Further Work

To our knowledge, this paper presents the first measurement-based study of the evolution of a successful agile system. A strength of this study is the combined use of quantitative and qualitative evidence so that one informs and provides context for the other.

This experience report provides hard evidence that an agile method allows smooth evolution while avoiding the problems of increasing complexity or decreasing customer satisfaction. It seems that this is due to the high level of complexity control work (witness the almost total lack of complex methods in the system) and the high rate of iteration, with customers quickly receiving the requested functionality. More agile-developed systems will need to be studied to see if these conclusions are generally true.

To fully consider the efficacy of evolution, both technical characteristics and the impact and contribution of stakeholders must be considered. [12]. This is not easy and to some extent this and all previous studies of evolution have been limited. However, this study has been strengthened by the combined use of quantitative measures of the technical evolution and the qualitative results of a previous observational study of the same system [16]. In future work, we want to explore more systematic ways of combining quantitative and qualitative observation for building a richer and fuller picture of software evolution. We also plan to use simulation models [23] in order to explore the possible relationships between growth, complexity control and other attributes during evolution

Acknowledgements

We would like to express our gratitude to all employees up until June 2005 of the company that provided us with their data for their help and co-operation in collecting the data and helping us to perform the analysis presented here. In particular, thank you to John Nolan, Rachel Davies and Ivan Moore.

References

[1] Boehm, B. and Turner, R., (2003) 'Using risk to balance agile and plan-driven methods', IEEE Computer, 36(6), June: 57 – 66

[2] The Agile manifesto, http://www.agilemanifesto.org/

[3] Boehm B. and Turner R., (2004) 'Balancing agility and discipline: evaluating and integrating agile and plan-driven methods', Proc. ICSE 2004, 23-28 May: 718 – 719

[4] Pfleeger, S. (2001), 'Software Engineering – Theory and Practice', 2nd edition, Prentice Hall, Upper Saddle River, NJ, 659 pp.

[5] Lehman M.M. and Belady L.A. eds. (1985), 'Software evolution – processes of software change', Academic Press, London, 538 pp.

[6] Bennett K.H. and Rajlich V.T. (2000) 'Software maintenance and evolution: a roadmap", in A. Finkelstein (ed.), The Future of Software Engineering 2000, in conjunction with ICSE 22, June 4-11, Limerick, Ireland, ACM Order Nr. 592000-1: 75 – 87

[7] Madhavji N., Fernandez-Ramil J. and Perry D.E. eds (2006) 'Software evolution and feedback – theory and practice', Wiley

[8] Lehman M.M. (1974) 'Programs, cities, students, limits to growth?', Inaugural Lecture, in Imperial College of Science and Technology Inaugural Lecture Series, v. 9, 1970, 1974: 211 – 229. Also in 'Programming Methodology', Gries D (ed.), Springer Verlag, 1978: 42 – 62. Reprinted as Chapter 7 in [Lehman & Belady 1985]

[9] Parnas D.L. (1994) 'Software aging', Proc. ICSE 16, May 16-21, 1994, Sorrento, Italy: 279 – 287

[10] Fowler M., Beck K., Brant J., Opdyke W., Roberts D. (1999) 'Refactoring: improving the design of existing code', Addison-Wesley

[11] Wernick P. and Hall T. (2004), 'The Impact of Using Pair Programming on System Evolution: a Simulation-Based Study', Proc. ICSM 2004, 11-14 Sept: 422 – 426.

[12] Chapin, N. (2004), 'Agile methods' contributions in software evolution', Proc. ICSM 2004, 11-14 Sept: 522—522.

[13] Beck, K. (2000, 2005), 'Extreme Programming explained: embrace change'. San Francisco: Addison-Wesley. 1st ed, 2000, 2nd ed. 2005.

[14] Capiluppi A. and Ramil J.F. (2004), 'Multi-level empirical studies: an approach focused on open source software', Late Breaking Paper, 10th International Software Metrics Symposium, September 14-16, 2004, Chicago, Illinois, USA, http://swmetrics.org/ <as of Sept. 2004>

[15] Kerth, N. (2001), Project Retrospectives: A Handbook for Team Reviews, Dorset House Publishing

[16] Sharp, H. and Robinson, H. (2004), 'An ethnographic study of XP practices', *Empirical Software Engineering,* **9**(4) 353-375.

[17] Cockburn A. (2001), 'Agile software development', Addison-Wesley

[18] De Rivieres J. and Beaton W., (2006) 'Eclipse platform technical overview', available online at http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html (as of Sept 2006).

[19] Boehm B.W. (1981) 'Software Engineering Economics', Prentice-Hall, Englewood Cliffs, NJ

[20] McCabe T. (1976), 'A complexity measure', IEEE Transactions on Software Engineering, 2(4): 308 – 320

[21] A. Capiluppi and J.F. Ramil (2004), Studying the Evolution of Open Source Systems at Different Levels of Granularity: Two Case Studies, 7th International Workshop on Principles of Software Evolution (IWPSE 2004), 6-7 September 2004, Kyoto, Japan. IEEE Computer Society 2004, ISBN 0-7695-2211-4, pp.113-118.

[22] Capiluppi, A., Faria, A. E., and Ramil, J. F. (2005): 'Exploring the relationship between cumulative change and complexity in an Open Source system', Proc. 9th European Conference on Software Maintenance and Reengineering (CSMR 2005), 21-23 March 2005, Manchester, UK, Proceedings: 21 - 29.

[23] Smith, N, Capiluppi, A, Fernández-Ramil, J, (2006) 'Agent-based simulation of open source evolution', Journal of Software Process – Improvement and Practice, ProSim 2005 Special Issue, in press.

[24] Lehman M.M., Perry D.E., Ramil J.F., Turski W.M. and Wernick P. (1997), 'Metrics and Laws of Software Evolution - The Nineties View', Proc. Metrics '97, Albuquerque, NM, 5 - 7 Nov. 1997: 20-32. Also as Chapter 17 in El Eman K. and Madhavji N.H. (eds.), 'Elements of Software Process Assessment and Improvement', IEEE CS Press, Los Alamitos, CA, 1999: 343 – 368

[25] Zuse, H. (1991) *Software complexity: Measures and methods*, Berlin: W. de Gruyter

[26] McCabe T.J. and Butler C.W. (1989), 'Design complexity measurement and testing', Comm. of the ACM, 32(12): 1415 -- 1425

[27] Capiluppi, A., Millen, J. and Boldyreff, C. (2006) How outsourcing affects the quality of mission critical software, 13th Working Conference on Reverse Engineering (WCRE 2005), 23-27 October 2006, Benevento, Italy.)

[28] Dunsmore, A., Roper, M. and Wood, M. (2003) 'Practical code inspection techniques for object-oriented systems: an experimental comparison', *IEEE Software*, July/August: 21 – 29