Selecting Refactorings in XP: An Option Based Approach

Androklis Mavridis, Apostolos Ampatzoglou, Ioannis Stamelos
Department of Informatics
Aristotle University
Thessaloniki, Greece
amavridis@csd.auth.gr
apamp@csd.auth.gr
stamelos@csd.auth.gr

Panagiotis Sfetsos, Ignatios Deligiannis
Department of Information Technology
Alexander Technology Educational InstituteThessaloniki, Greece
psfetsos@it.teithe.gr
ignatios@it.teithe.gr

***Abstract* — Refactoring, as an XP practice, aims to improve the design of existing code to cope with foreseen software architecture evolution. The selection of the optimum refactoring strategy can be a daunting task involving the identification of refactoring candidates, the determination of which refactorings to apply and the assessment of the refactoring impact on software product quality characteristics. As such, the benefits from refactorings are measured from the quality advancements achieved through the application of state of the art structural quality assessments on refactored code. Perceiving refactoring trough the lens of value creation, the optimum strategy should be the one that maximizes the endurance of the architecture in future imposed changes. We argue that an alternative measurement and examination of the refactoring success is possible, one, that focuses on the balance between effort spent and anticipated cost minimization. In this arena, traditional, quality evaluation methods fall short in examining the financial implications of uncertainties imposed by the frequent updates/modifications and by the dynamics of the XP programming. In this paper we apply simple Real Options Analysis techniques and we perceive the selection of the optimum refactoring strategy as an option capable of generating value (cost minimization) upon adoption. Doing so, we link the endurance of the refactored architecture to its true monetary value. To get an estimation of the expected cost that is needed to apply the considered refactorings and to the effect of applying them, in the cost of future adoptions we conducted a case study. The results of the case study suggest that every refactoring can be associated with different benefit levels during system extension.**

***Keywords* – Refactorings; Real Options; Architecture Edurance**

I. Introduction

Extreme programming (XP) [3], has emerged as one of the most popular agile methods [4]. XP is an iterative and incremental development methodology to delivering high-quality software quickly and continuously, with a short planning horizon (three month releases, 1-2 week iterations). It is based on four simple values – simplicity, communication, feedback, and courage – and twelve supporting practices: planning game, small releases, simple design, testing, refactoring, pair programming, collective ownership, continuous integration, sustainable pace (used to be: 40-hour week), on-site customer, coding standards and metaphor. In XP, a release - a stable and working deliverable version of the product - is divided into shorter increments of development (iterations) where individual tasks are assigned to developers. Promoting continuous planning, continuous testing and refactoring, customers' involvement and close teamwork, XP delivers high-quality software quickly and continuously (every 1-3 weeks).

XP introduces two critical practices to improve the design of existing code and to manage the evolution of software architectures: test driven development [5], [2] and refactoring [14]. Refactoring is a disciplined way to make changes to source code, improving its design (internal structure) without changing its external behavior. Mens and Tourwé [18] describe refactoring as a multi-stage process involving many activities which are: the identification of refactoring candidates (i.e., where the software should be refactored), the determination of which refactorings to apply and the assessment of the refactoring impact on software product quality characteristics (e.g., complexity, understandability, and maintainability) or software process (e.g., productivity, cost, and effort).

A growing number of studies address the relationship between refactoring and the internal structure of source code and its impact on program understanding, software quality, and the evolution of a software design. Most of these studies focus on the identification of code smells to locate possible refactorings [14], [23], [8], [31], [27], the reconstruction of the overall design of existing systems [10] and the improvement of the legacy code [7], [20]. Concerning quality, most of the studies found a positive relationship between software refactoring and the software quality, either indirectly or directly measured, [11], [28], [17], [29], [25], [19], [1], [15], [23], [30].

While the most research studies conclude that refactoring has long-term benefits on the quality of a software product (in particular on program understanding) there is no such consensus regarding the economic benefits gained from the refactored architecture. In this paper we argue that value based reasoning can be injected to decision making process regarding the selection of the optimum refactoring strategy.

In our approach we perceive each refactoring strategy as a different project leading to different value. In this context, from the pool of available refactoring strategies the analyst should opt for the one that maximises the value of the refactored architecture. Our notion of value is correlated with the architecture's capability to cope to requests for new features (system extension) as this is often the case with the Agile development paradigm. This capability is reflected in the cost required to add new features to the system. Under this perspective, a refactoring strategy that minimizes the cost of features implementation with minimum refactoring cost is the one that should be preferred. Hence selecting the suitable refactoring strategy is a question of balancing between expected gains (cost minimization of system extension) and the cost to refactor the system.

We make the hypothesis that refactoring carries economic value in the form of Real Options, expressed through the right, but not the obligation, to select a refactoring strategy within a given time frame, where the refactoring strategy to be selected is treated as a real asset. Framing refactorings as options, we can discover when and under which conditions (cost and expected value) a given refactoring should be implemented or not.

The paper is structured as follows. In section two we present related work on refactorings and we introduce the fundamentals of Real Options Theory along with its application in the software engineering and agile development paradigm. In section three, we present the proposed methodology followed by the case study in section four. Finally we discuss the findings and we conclude in section six.

II. Related Work

A. Identification of refactoring opportunities

Many studies have been conducted to find sets of classes that are in need of refactoring. The most common approach to detect where to refactor is the identification of bad smells [14].

Fowler provided a catalogue of refactoring and assigned some of these refactorings to fix code bad smells [14]. Additionally, in [26] the authors define a cohesion metric, which measures the cohesion among attributes and methods. This metric is used to identify methods that use or are used by more features of another class than the class that they belong to, that should be refactored. Furthermore, in [28] use a set of complexity, coupling, and cohesion OO metrics to detect classes for which quality has diminished and design flaws have been detected. O'Keeffe et.al. treat object-oriented design as a search problem in the space of alternative designs and employ search algorithms using metrics from a hierarchical design quality model as an evaluation function that ranks the alternative designs. Seng et.al propose a model that examines a set of pre and post-conditions in order to simulate the application of Move Method refactorings. Finally, Du Bois et al. analyse the best and worst-case scenario of the impact of refactorings on coupling and cohesion dimensions. The investigated refactorings are Extract Method, Move Method, Replace Method with Method Object, Replace Data Value with Object, and Extract Class [8].

To help in automating refactoring, many development tools, e.g. Eclipse and Borland Together, integrated refactoring support for many OO languages. These tools do not support detecting where and when a refactoring should occur, but automatically apply it, according to programmers' suggestions.

B Real Options in Software projects

Real Options Analysis (ROA) is based on the analogy between investment opportunities and financial options. A real option is a right, but not an obligation, to make a decision for a certain cost within a specific time frame. A project is perceived as an option on the underlying cash flows with multiple associated investment strategies to be exercised if conditions are favourable. The big advancement is that ROA accommodates not only the value of the investment's expected revenues but also the future opportunities that flexibility creates.

As option is an asset that provides its owner the right with out a symmetric obligation to make an investment decision, the owner can exercise the option by investing the strike price defined by the option. A call option gives the right to acquire an asset of uncertain future value for the strike price. A *call option* gives the buyer of the option the right to buy the underlying asset at a fixed price, called the strike or the exercise price, at any time prior to the expiration date of the option: the buyer pays a price for this right. If at expiration, the value of the asset is less than the strike price, the option is not exercised and expires worthless. If, on the other hand, the value of the asset is greater than the strike price, the option is exercised – the buyer of the option buys the asset at the exercise price and the difference between the asset value and the exercise price comprises the gross profit on the investment.

While ROA was applied extensively as a decision making tool for in IT projects, during the last decade the application of ROA is concentrated in valuating the inherent uncertainties in software engineering practices such as in [6],[12],[13],[21],[22], [26] [31]. Extend the related work focusing on the ROA "in' projects (half + page)

Based on these foundations, the central idea of our work is that a refactoring selection, is analogous to a financial derivative expressed as a *call option,* where the owner (the analyst) has the right but not the obligation to make the

selection within a given time frame.

III. Methodology

In this work we apply Real Options Analysis in the context of refactoring selection in order to address the following uncertainties:

- Which candidate refactoring offers more value?
- What is the added value offered from the refactoring if this is properly executed?
- How the conditions (i.e. number of features added or developers ability to perform the refactoring task), affects the value of the extended system?

Our method employs three consecutive steps. The first step regards the identification of the candidate refactoring strategies. In the second, we calculate the costs required to implement each refactoring, the revenues expected from each refactoring and the standard deviation (volatility) of these revenues reflecting the uncertain conditions (i.e. number of features added). Finally in the third step we calculate the call options for each OSS candidate and we compare the results.

We extend the ROA valuation mechanism to accommodate the specificities and constraints of the refactoring context as shown in table 2:

Real Option on an IT project	**Real options on a refactoring strategy**
The business value of IT project	The value of the refactoring contributing to system's profitability upon selection.
Expenditures associated with the project investment.	The costs to implement the refactoring strategy.
Time until the IT investment opportunity runs out.	Time until the refactoring adoption opportunity runs out.
Uncertainty for product or service offerings from the project.	Uncertainty regarding the successful implementation of the refactored architecture, directly affecting the cost of system's extension.
Risk-free interest rate in the IT domain	Risk-free interest rate in the IT domain.

Table 2. Real Options capturing the refactoring constructs

Extending the applicability of the Real Options theory we make the following mapping to ROA variables:

- *Expected Value of refactoring (S): Is given as the cash flows resulted from the implementation of adding new features. It is constant for all refactoring strategies.*
- *Exercise Price (X):* Expressed as the accumulation of the costs to implement the given refactoring.
- *Time to Expiration (T):* This is the time until the opportunity disappears for making the refactoring selection.
- *Volatility of the Expected Value (σ):* Represents the % percentage of up or down fluctuations on the *Expected Value* due to uncertainties affecting the successfulness of the refactored system.
- *Risk Free Rate (r):* Assumed to be a known market value, or the cost of capital.

Extending the RAO "in" project to fit to our approach (+half page)

Having these variables we can calculate the *Call Option Value* of each refactoring strategy applying binomial options pricing model [9].

We build for each candidate refactoring strategy two binomial lattices based on the American call option fashion, which dictates that the option (the selection) can be exercised at any given time until the expiration. The first lattice calculates the maximum and the minimum expected refactoring value within the given time frame, while the second calculates the option value (OV), the amount by which the option to implement the refactoring is in the money, or in other words what is the added value of correctly implementing the system extension when favorable conditions exist. The Option Value (OV) depends on two variables, the Intrinsic Value (IV) and the Time Value (TV), as such,

$$OV = IV + TV$$

The **intrinsic value (IV)** of an option is the value of the option if exercising it now and is given as:

$$IV = S - X,$$

Intrinsic value can be defined as the amount by which the strike price of an option is "in-the-money", . Thus the higher the intrinsic value the better for the given refactoring strategy.

Time value or Extrinsic Value, or "Time Premium", given as:

$$TV = OV - IV,$$

is the real cost of owning the option to refactor or in other words it is the cost that the analyst have to pay waiting for favorable conditions to arise. Accordingly, the less the time value the better it is for opting for the given refactoring. This cost can be attributed to various factors such as not early market presence of the extending system, delayed revenues from successful adoption, delayed revenues from successful reuse and so forth, depending on the actual investment and scenario of use. Finally the real value of the extended system (*The Net Profit)* is given as the subtraction of the Expected value with the Exercise price and the Time Value:

$$Net\ Profit = S\text{-}X\text{-}TV$$

Hence, the refactoring strategy giving the highest Net Profit for the extended system is the one that should be preferred.

IV. Case Study

This section deals with presenting the structure and the

results of a case study, which was performed to valuate the economic benefits of three refactoring activities, (a) *Move Method*, (b) *Extract Method* and (c) *Polymorphism*. The case study is divided into two parts, first the calculation of ROA variables and the second the actual Option-Based Analysis.

A. Calculating ROA Variables

This section aims at calculating the variables needed in the real options analysis. For every investigated refactoring, the method needs three variables, (a) S (expected value), (b) X (cost – exercise price) and (c) σ (volatility). In order to calculate these variables we conducted a case study on one hundred (100) junior programmers, according to the guidelines described in [16]. The steps that have been followed during case study execution are the following:

- Build the dataset
- Identify the method of comparison
- Execute case study
- Analyze and report the results

Build the Dataset

The subjects of the case study have been one hundred (100) junior programmers with a BSc in Computer Science. The objects of the case study have been four successive versions of a medium size, sales management system, approximately 4500 lines of code. The selected sales system has been picked among other sale systems, because of its size that could be handled in the terms of an experiment by junior programmers. Additionally, the design of the system had some *bad smells* that could be refactored in the quality enhanced versions.

The first version of the system included no refactoring activities (*v0*). In version 1 one instance of a move method refactoring has taken place (*v1*). In version 2, an additional instance of an extract method refactoring has been performed (*v2*) and in the final version (*v3*) an opportunity of using the polymorphism has been identified and applied. The selected refactorings have been randomly selected among the refactoring opportunities that have been identified from the sales system under study. The application of each refactoring (*refactoring_type*) has been performed by 6-7 subjects and time needed for each transition (*transition_time*) has been recorded.

Next, the (80) eighty remaining developers have been divided into four groups, each one attached with one version of the system. Every subject has been given two extension scenarios that should be adopted. For every developer the time needed for performing several perfective maintenance tasks has been recorded (*extension_time*). The successful completion of all maintenance tasks leads to a 10% LOC increase in the initial system. The extension scenarios have been manually created by the research team. The rationale of the selected extension scenarios was to involve as many refactored classes, as possible.

Thus, the dataset of the case study included three variables as shown below:

- Project Version
- Average Transition Time for Corresponding Project Version
- Extension Time

Identify the Comparison Method

On the completion of data collection phase, the following steps have been performed so as to answer the research questions described above:

1. *Calculate Average Transition Time for Corresponding Project Version.* For each version, average transition time is the sum of all previous versions average transitions time, plus the time needed for current transition.

2. *Calculate Average Extension Time for Every Project Version.*

3. *Calculate Average Standard Deviation for Extension Time for Every Project Version.*

Results

The dataset created from the above mentioned procedure is summarized in Table 2.

System Version	***v0***	***v1***	***v2***	***v3***
AVG transition time	0.00	7.55	10.83	16.92
AVG extension time	44.22	38.57	32.52	30.42
AVG standard deviation	8.39	6.09	5.47	4.09

Table 2. Case Study Dataset

B. Option-Based Analysis

From the dataset presented in table 2, we measured the variables necessary for calculating the call options for each refactoring leading to system versions v1 to v3.

To convert time to money and be able to calculate the exercise price for each refactoring we assumed an average hourly pay rate of 6€. Having a 10% increase of the original system after the maintenance tasks completion, we simply attributed a 10% increase in the systems revenue as its expected value, giving an (S) of approximately 1000€. The calculated the exercise price for each systems version we first added the transition and extension times and then multiply the product with the pay rate. Finally are shown in Table 3.

System Version	***v0***	***v1***	***v2***	***v3***
Total time	44.22	46.12	43.35	47.34
Exercise Price (X)	265.32	276.72	260.1	284.04

Table 3. Exercise price for system versions

With an options *time to expiration (T)* of 4 months, cost of capital (*r*) at 15% and using AVG standard deviation as *volatility (σ),* we have for each system version the following results:

Version 0:

Version 1:

Version 2:

Version 3:

Having calculating the Option Values for each system extended version we can finally obtain the Net Profits for each version, as shown in table 4.

System Version	***v0***	***v1***	***v2***	***v3***
Option Value	745.97	735.46	751.35	728.46
Time Value	11.29	12.18	11.45	12.5
Net Profit	**723.39**	**711.1**	**728.45**	**703.46**

Table 4. Net profits for each system version

V. DISCUSSION

Placing the advancements of the ROA "in" projects that we proposed, and how we extend these in our paper...

From table 4 we can conclude that the optimum refactoring strategy is the one of the system version 3, as it offers the higher Net Profit. The results from the analysis highlight the following remarks. Firstly it is not always the case that the refactored system will provide the adequate level of assistance to developer to better handle the required extensions. As it is shown in Table 4 system V_0 (without refactoring) has a higher Net profit surpassing two of the three refactored versions. This is rather surprisingly and highlights the importance of balancing between the cost to refactor and expected value.

Adding refactoring activities has a positive effect in decreasing the time to extension. However this is not always translating to increase profit. This suggests that the benefits from the increased quality are visible and explorable upon system extension. It is therefore not a matter of the number of refactoring activities performed, but which activities produce the higher profit.

Volatility either measured from Expected Values or from costs, plays a catalytic role options values and as a consequence to the final Net profit. This is clear in the case of system V_0 which has the higher volatility amongst the three refactored versions.

VI.CONCLUSIONS

We have presented an alternative approach aiming to assist system analysts to better select the most profitable refactoring strategy. This was possible by perceiving refactoring through the lens of value creation, focusing on the maximization of the benefits imposed by required future system extensions. We argued that an alternative measurement and examination of the refactoring success is possible, one, that focuses on the balance between effort spent and anticipated cost minimization. We employed Real Options Analysis to identify the relationship and the mechanisms between the cost to refactor, the anticipated value and the uncertainties of refactoring successfulness. To provide an initial indication of Real Options applicability in the context of system Refactoring we conducted a case study. The results of suggest that the most profitable refactoring strategy is not always the one that includes the most refactoring activities but the one that properly balances the cost to refactor and the expected value.

REFERENCES

[1] Alshayeb, M. (2009). Empirical Investigation of Refactoring Effect on Software Quality‖, Information and Software Technology, 51 (9), pp. 1319–1326.

[2] Astels, D. 2003. TestDriven development: A practical guide. Upper Saddle River, N.J.: Prentice Hall.
[3] Beck, K. (2000) Extreme programming explained: Embrace change. Reading, MA.: Addison Wesley Longman, Inc.
[4] K. Beck, M. Beedle, A. Bennekum van, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt,R. Jeffries, J. Kern, B. Marick, R. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for Agile Software Development," vol. 2002, 2001.
[5] Beck, K. 2003. Test-driven development: By example. Boston: Addison-Wesley.
[6] Boehm, B.W. & Sullivan, K. (2000) Software Economics: A Roadmap, in the Future of Software Engineering, 22nd International Conference on Software Engineering, June.
[7] Bois, B.D., Mens, T. (2003): Describing the impact of refactoring on internal program quality. In: Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA), Amsterdam, The Netherlands, pp. 37–48.
[8] Bois, B.D., Demeyer, S., Verelst, J. (2005). Refactoring – Improving Coupling and Cohesion of Existing Code. In: Belgian Symposium on Software Restructuring, Gent, Belgium.
[9] Copeland, T. Antikarov, V. (2001). Real Options: A Practitioner's Guide, TEXERE, New York, NY.
[10]Demeyer, S., Ducasse, S., Nierstrasz, O. (2000). Finding Refactorings via Change Metrics. In: Proceedings of the 15th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2000, Minneapolis, USA.
[11]Demeyer, S. (2002). Maintainability versus Performance: What's the Effect of Introducing Polymorphism?, technical report, Lab. On Reengineering, Universiteit Antwerpen, Belgium.
[12]Engel, A. & Browning, T. Designing systems for adaptability by means of architecture options. Systems Engineering, 11 2 (2008), (pp125-146).
[13]Erdogmus, H. Valuation of Learning Options in Software Development under Private and Market Risk, The Engineering Economist, **47** 3 (2002), (pp 308-353).
[14]Fowler M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA.
[15]Kataoka, Y., Imai, T., Andou, H., Fukaya, T. (2002). A Quantitative Evaluation of Maintainability Enhancement by Refactoring‖, Proceedings of the International Conference on Software Maintenance (ICSM.02), pp. 576–585.
[16]Kitchenham B., Pickard L., Pfleeger S.L, "Case Studies for Method and Tool Evaluation", IEEE Software, IEEE Computer Society, 12 (4), July 1995. Leitch, R., Stroulia, E. (2003). Assessing the Maintainability Benefits of Design Restructuring Using Dependency Analysis‖, Ninth International Software Metrics Symposium (METRICS'03), pp. 309–322.
[17]Mens, T., Tourwe, T. (2004). A Survey of Software Refactoring, IEEE Transactions on Software Engineering‖, 30(2), pp. 126–139.
[18]Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A., Succi, G. (2008). A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team‖, In Balancing Agility and Formalism in Software Engineering. Lecture Notes In Computer Science, (5082). Springer-Verlag, Berlin, Heidelberg, pp. 252-266.
[19]Pizka, M. (2004). Straightening spaghetti-code with refactoring? In: Proceedings of the Int. Conf. on Software Engineering Research and Practice - SERP, Las Vegas, NV, pp. 846–852.
[20]Racheva, Z., Daneva M., and Buglione, L. Complementing Measurements and Real Options Concepts to Support Inter-iteration Decision-Making in Agile Projects, *Proc. 34th Euromicro Conf. Software Engineering and Advanced Applications,* pp. 457-464, 2008.
[21]Racheva, Z., & Daneva M. Using Measurements to Support Real-Option Thinking in Agile Software Development, *Proc. 2008 Int'l Workshop Scrutinizing Agile Practices or Shoot-Out at the Agile Corral,* May 2008.
[22]Ratzinger, J., Fischer, M., Gall, H. (2005). Improving Evolvability Through Refactoring, Proceedings of the 2nd International Workshop on Mining Software Repositories (MSR'05), 1–5.
[23]Schofield, C., Tansey, B., Xing, Z., Stroulia, E. (2006). Digging the Development Dust for Refactorings. In: Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006), Athens, Greece.
[24]Shatnawi, R., Li, W. (2011). An Empirical Assessment of Refactoring Impact on Software Quality Using a Hierarchical Quality Model. International Journal of Software Engineering and Its Applications Vol. 5 – October, No. 4.
[25]Shaw, M. et al. (2005). In search of a unified theory for early predictive design evaluation for software, Technical Reports CMU-ISRI-05- 114, Carnegie Mellon University.
[26]Simon, F., Steinbruckner, F., Lewerentz, C. (2001). Metrics based refactoring. In: Proc. European Conf. Software Maintenance and Reengineering, pp. 30–38. IEEE Computer Society Press, Los Alamitos.
[27]Stroulia, E., Kapoor, R.V. (2001). Metrics of Refactoring-Based Development: an Experience Report‖, In The seventh International Conference on Object-Oriented Information Systems, pp. 113–122.
[28]Tahvildari, L., Kontogiannis, K. (2004). Improving Design Quality Using Meta-Pattern Transformations: A Metric-Based Approach‖, J. Software Maintenance. Evolution: Research and Practice, 16 (4-5), pp. 331–361.
[29]Yu, Y., Mylopoulos, J., Yu, E., Leite, J.C., Liu, L., D'Hollander, E. H. (2003). Software refactoring guided by multiple soft-goals. In: Proceedings of the 1st workshop on Refactoring: Achievements, Challenges, and Effects, in conjunction with the 10th WCRE conference 2003, Victoria, Canada, November 13-16, pp. 7–11.
[30]Van Emden, E., Moonen, L.: (2002). Java Quality

Assurance by Detecting Code Smells. In: Proceedings of the 9th Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos.
[31]Wang T. & de Neufville, R. (2006). Identification of real options "in" projects. In Proc. 16th Annual International Symposium of the International Council on Systems Engineering (INCOSE), Orlando, FL, July.