Enabling Testing, Design and Refactoring Practices in Remote Locations

Amey Dhoke, Greg Gigon
Kuldeep Singh, Amit Chhajed
Ben Stopford
Royal Bank of Scotland
London
benjamin.stopford @rbs.com

Learning is a process of successive steps; we learn, we practice, the process cycles. It requires dedication from both teacher and student and it requires constant reinforcement [13]. It is our contention that the best method for transferring skills like testing, refactoring and software design is through contextual learning: An ongoing program of enablement in which practices are shared in the context of the programmer's work in response to the challenges they face. In addition the code base represents one of the most affective communication tools available to a distributed team.

We discuss some of the problems that faced our team: A green field, test-driven project with twenty developers split between London and India. We discuss the methods we employed on the project to enable better testing and refactoring practices across the geographical divide. Each method is appraised, not only as as a stand-alone practice, but also in the context of the project life as we found that different practices better-suited different project phases. We relate this evolving set of practices to the Shuhari learning model [16].

We conclude that there is no substitute for colocation. However we found that the team's motivation is crucial to the success of these endeavors. Intensive one-on-one practices work well at the start of the project, when motivation was high and there was lots of ground to cover. As the project continued, the distribution of skills became more even and more collaborative practices were needed to affectively increase learning.

Keywords-Distributed software development, Programming practices enablement, Learning techniques

I. Introduction

Distributed teams are a common occurrence these days, particularly in the software development community. Along with issues of physical separation there are disparities in culture, language, time zone and skill sets [2].

Testing and refactoring are widely considered to be necessary skills for effective software development. As with other skills, they require learning and practice. The more practice a programmer has the more skilled they become. However, to maximise the programmer's learning, instruction needs to be both directed and constantly enforced.

It is our belief that traditional learning methods break down in the context of teams separated by geographical boundaries. Classroom style teaching is rarely successful if delivered in isolation. Typical methods such as the sending of code snippets, documents and diagrams cannot substitute for collocated techniques like whiteboard sessions and pairing.

II. Problems with Regular Training Programmes in OO, Refactoring and Testing Techniques

A. *The Effective Teaching of Refactoring and Testing Practices Requires Contextual Learning*

Books and other typical classroom aids provide an effective theoretical basis for refactoring and testing. However the application of these concepts in the context of real-world systems is a far greater challenge. Bookwork is good for providing a conceptual understanding but this needs reinforcing in a real world context. The Strategy Pattern provides a good example [3]. It is often taught using a sorting analogy where different algorithms represent different strategies that can be applied to the sorting of a list, each having the same functional output. For example bubble sort or quick sort algorithms. However, the real world application of this pattern can be quite different. For example replacing conditionals using a Strategy (or Policy) as in the Replace Conditional with Polymorphism refactor [4] and Conditional Decomposition [5]. Understanding (or teaching) the application of such patterns without the context of a real, complex code base is difficult.

Cognitive Science states that one of the most effective methods of learning is the apprenticeship model [6]. In our context the student is taken through a solution in a collaborative manner, preferably with reference to a vocational situation. In teams following an Agile methodology this generally takes the form of Pair Programming [7]. The pairing method can be used in an instructional way when experienced programmers are paired with less experienced students to lead them through the challenges of daily programming tasks. This facilitates both the constant feedback necessary to learn as well as the contextual basis of a real code base and real world problems.

B. *Communication Issues*

A lack of direct communication is one of the fundamental problems faced by distributed teams [8] making all forms of learning a challenge. Language, cultural and time zone differences all play their part. Language differences can be compounded further by a lack of familiarity with the more esoteric concepts and language that go with most technical fields.

These factors can lead to unnecessary frustration on both sides, particularly in one-on-one sessions when voices are transmitted over low bandwidth phone lines.

C. *Motivation*

A key element of fostering a learning culture is …The communication boundaries make it much harder to communicate in a way that avoids accidental

III. THE CONTEXT OF THE ODC TEAM

The ODC team, from which the experiences described in this paper originate, is a distributed team of around 20 developers spread 12:8 between India and the UK. The project was originally greenfield, is developed using Test Driven Development and has run for just over a year. There were various disparities in expertise running in both directions across the geographical divide. As a result a variety of methods were explored to distribute this knowledge and skills from one location to another.

IV. TENETS THAT DRIVE OUR LEARNING PRACTICES

The practices detailed in the following section are driven from three key tenets:

1. Learning practices must be collaborative and bi-directional. One way 'instruction' or 'review' will only prove fruitful for limited periods as it stifles both adoption and ownership of the practices being taught.
2. The code base should be considered the main tool for communicating practices and techniques. It is language and culturally neutral and forms the key to contextual understanding. Techniques that are code focussed should be preferred to any form of theoretical discourse.
3. The distribution of skills will become increasingly homogenous over the life of the project. As such the team needs different practices at different times.

V. STRUCTURAL AND PROCEDURAL SKILLS

We find it beneficial to segregate skills we needed to transfer into two types: Structural and Procedural. We use these to categorise practices.

1. Structural Skills are those required to create well-structured software, for example the application of patterns, the use of different types of unit tests and use of different types of test fixtures and builders. Methods for dealing with these problems were best taught with the code base as the primary medium for knowledge transfer. This being necessitated by the problems being contextual: solutions had to be taught in the context of real world problems.

2. Procedural Skills. These involve the processes a developer goes through writing software for example the compartmentalisation of a problem into individually committable steps, the selection of seams [15] for refactoring and the test, pass, refactor cycle [14]. Such processes are extremely difficult to teach without colocation.

The practices here mostly focus on Structural Skills.

VI. DISTRIBUTED LEARNING TECHNIQUES USED TO AID TDD, REFACTORING AND DESIGN PRACTICES ON THE ODC PROJECT

A. *Abridged Pairing*

The aim of this practice was to transfer OO, testing and refactoring practices from one location to another in an apprenticeship-like manner.

In this practice, two developers, one from each location pair on design and development practices for 1 hour each day. The process is repeated daily to retain continuity. Tasks are set for completion between sessions and questions that come up in the interval are answered. The sessions were facilitated through a desktop sharing tool [9] and telephone communication.

1) Pros

Targeted/Interactive: The targeted, interactive nature of this practice made it one of the most productive. Its one-on-one nature allows focus to be specific to the individuals involved and the context of the problem at hand.

Continuity: The on going nature of the sessions provides the feedback necessary to facilitate effective apprenticeship learning. The pairing are able to follow the evolution of a real software problem (typically a story) over a number of days.

Real-World Problems: The practice focuses exclusively on real world problems in the context of the code base and the story being developed. This allows techniques like refactoring, testing and OO design to be discussed in the context of an evolving story. As such echoes many of the learning characteristics associated with traditional pair programming [9].

2) Cons

Suitable for Structural not Procedural Skill Transfer: Unlike traditional pairing, the short timescales involved in Abridged Pairing make it impractical to teach processes (such as the test-pass-refactor cycle [14]). The sessions tended to be more focused on structural issues that have occurred since the last session, such as a mixing of concerns in a class.

Read Only Code Communication: Solutions discussed in Abridged Pairing sessions could be discussed in the context of the code base but the latency of the screen sharing software made concurrent editing impractical. Developers can discuss to not collaborate over the code base.

Frustration: Both sides of the pairing found the practice difficult at times. The one to one nature of the practice make it open to accidental slighting and communication frustration As such, whilst very successful for short bursts it became harder to maintain the enthusiasm necessary for such intense sessions, longer-term.

B. Collaborative Refactoring

This practice is similar to a traditional code review [10]. When a story is completed it is handed to a developer from the remote team for review. Instead of the traditional review process the reviewer actually refactors the code he is reviewing. At the end of the session the original programmer analyses the changes that were made and discusses them with the reviewer. The practice works best when it is targeted at a particular goal, for example describing the replace conditional with polymorphism refactor by applying it to a well known piece of code.

1) Pros

This technique has a few advantages over more traditional methods:

Real-World Problems: It gives the developer physical examples of the decisions that another developer make, just as they would in a collocated paring session.

Code as the Primary Communication Channel: The reviewer is able to use the code base as the communication medium through which to describe a certain pattern. For this reason, like Abridged Pairing, it is better suited to OO Design and refactoring (i.e. structural techniques rather than procedural ones like the test-pass-refactor cycle [14]).

2) Cons

Time Consuming: Like Abridged Pairing, collaborative Refactoring is quite time consuming and the context switch required from the teacher impacts their performance and Flow.

Unintentional Offense: Like Abridged Pairing this practice has the potential to cause unintentional offense if refactorings are too broad in scope. This is minimised by keeping the session targeted to a specific goal.

C. Code Review Blitz

This practice involves one large, consolidated pairing session incorporating developers from both locations. The practice has three phases: An initial phase in which the stories under review are discussed. A second phase where the stories are reviewed and notes are taken (similar to a regular code review [10]). In the final stage reviewers provide feedback. Themes from the reviews are collected and addressed in additional one-on-one of group sessions.

1) Pros

The Group provides Inertia: We noticed a number of advantages when moving to group based practices. The group dynamic gives more inertia to the practice, encouraging participation.

Groups are more Disarming: We found the group dynamic to be more collaborative and disarming than direct one-to-one feedback. This makes it more engaging for both teams increasing learning potential.

Collecting Broader Themes: The group nature of this practice makes it easy to collect broader themes to be focussed on separately either in group design sessions or more focussed practices.

2) Cons

Lack of Review Freshness: Probably hte biggest drawback of this practice is the lack of freshness in the code under review, the oldest of which may be weeks old by the time the next Code Review Blitz comes around. This lack of freshness in the minds of the developers weakens the learning potential of the practice.

More Review than Instructional: The Code Review Blitz does not use the code base as a conduit for instruction. As such the learning potential it provides is limited.

D. Secondary Level Training: Driving Focussed Traning Sessions From Broader Review and Instructional Methods

The review and instructional processes described above provide a base level of training, focussing on skills and apprenticeship rather than facts and the broader issues of development. As such we found it to be beneficial to have a second, more focused level. This second level is driven from the Abridged Pairing, Collaborative Refactoring, the Code Review Blitz sessions or even just the general wonderings of the code base that occur during software development. The team looks for overarching themes or problems that require longer, more focussed training or discussion and addresses them specifically. For example it was observed that testing practices were often leading to too much coupling between test classes and implementations. This lead to collocated training sessions on TDD. Other sessions were conducted over phone / screen sharing / video conferencing (with the phone / screen sharing generally being considered the most productive).

E. Developer Rotations

The most brute force approach that we tried: developers swap location for two-week periods.

This gives the opportunity to learn development practices first hand, pair etc. It also helps develop a common set of development practices and improves the feeling of collective ownership [11]. This is the only effective way we found for transferring procedural skills like the test-pass-refactor cycle [14]. The downside of this practice is the cost.

F. Utilising Practice Champions

We found that, where a practice needed transferring, it was often beneficial to focus on one team member who displayed proficiency who could then inculcate these practices in an on going basis. This technique is particularly useful when transferring procedural rather than structural skills requiring a more apprentice-like learning method.

G. Remote Pair Programming

We did not try this practice due to insufficient tool support, communication issues and timing differences. We make note of it only because of its reported success in other contexts [12].

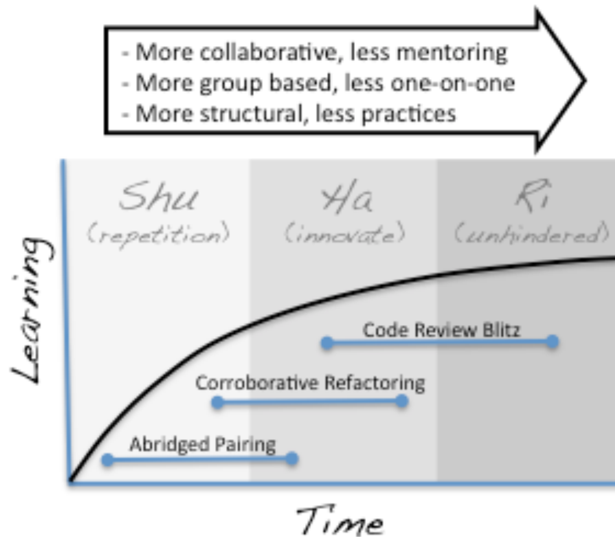

Figure 1: Description of the three phases in the Shuhari learning model. We found different practices more appropriate at different stages.

VII. CONCLUSIONS

This talk presents a case study from the ODC team that explores the challenges faced transferring skills across a geographical boundary.

Our premise is that distributed communication is a skill distinct from its co-located counterpart. A familiarity with collocated communication can blind us to its ineffectiveness of its application in a geographically and culturally dispersed context. As such we suggest practices that favour the use of the code base as a conduit rather than traditional, verbal methods.

We found that the team needed a range of practices that could be switched in and out at different times in the project's evolution. The learning practices themselves cause knowledge differentials within the team to subside making members become more independent. Mentoring style practices are intense, strongly enabling in individual skills but difficult to maintain long term. As such they were more successful at the start of the project when practices sit in the Shu (repetition) phase of learning in Figure 1 [16]. However the prolonged use of mentoring-styled practices was found, in some cases, to inhibit growth. In some cases this affected morale on both continents. Switching to collaborative practices helped to foster the move to the Ha (innovative) stage of learning. Composite practices like the Code Review Blitz providing both the benefits of personal direction and group feedback. We also found group practices to be more collaborative as they the reduce possibility of communication breakdowns that can cause frustrations on both sides.

<the importance of making learning two way – less experienced developers are often able to provide useful feedback to experienced ones on many implementation issues even – two sides of reviewing>

[1] Lotlarsky, J & Oshri,l, (2005) Social ties, knowledge shareing and successful collaboration in globally distribted, system development projects. *European Journal of Informantion Systems*, 14, pp. 37-48

[2] B. Ramesh, L. Cao, K. Mohan, P. Xu, "Can distributed software development be agile?", Communications of the ACM, October 2006, pp. 41-46

[3] J. Kierievsky, "Refactoring to patterns", China Machine Press, 2006

[4] M. Fowler, K. Beck, "Refactoring: improving the design of existing code", Addison Wesley Longman, 1999, pp.255 - 260

[5] M. Fowler, K. Beck, "Refactoring: improving the design of existing code", Addison Wesley Longman, 1999, pp.238 - 240

[6] S. E. Berryman, "Designing Effective Learning Environments: Cognitive Apprenticeship Models", ERIC Document, 1991, pp. 1

[7] D. Wells, "Pair Programming", http://www.extremeprogramming.org/rules/pair.html, 1997

[8] L. Layman, L. Williams, D. Damian, H. Bure, "Essential communication practices for Extreme Programming in a global software development team", Elsevier, 2006, pp. 1-2

[9] Williams L. and Kessler R., "Pair Programming Illuminated", Addison-Wesley, 2002, pp. 113-114

[10] Johnson, P.M., Reengineering Inspection: The Future of Formal Technical Review, in Communications of the ACM. 1998. pp. 49-52

[11] Distributed agile development at Microsoft patterns and practices group pp. 10-11

[12] Williams, L., et al., Strengthening the Case for Pair Programming, in IEEE Software. Online at http://www.cs.utah.edu/~lwilliam/Papers/ieeeSoftware

[13] http://en.wikipedia.org/wiki/Reinforcement_learning

[14] Growing Object-Oriented Software Guided By Tests Steve Freeman, Nat Pryce, Addison Wesley 2009

[15] **Working Effectively With Legacy Code ….**

[16] McCarthy, Patrick, "The World within Karate & Kinjo Hiroshi" in Journal of Asian Martial Arts, V. 3 No. 2, 1994.