Agile Offsharing: Using Pair Work to Overcome Nearshoring Difficulties

Lutz Prechelt
Freie Universität Berlin
Institut für Informatik
Takustr. 9, 14195 Berlin, Germany
Email: prechelt@inf.fu-berlin.de

***Abstract*—A major problem in distributed development situations, in particular offshoring situations, is often creating a proper understanding of the requirements at the remote site. It is difficult even if such understanding is available at the local site. This note argues why cross-site, synchronous, closely-coupled pair work at an engineering level, such as pair programming, may be a method for solving this problem and that corresponding research should be carried out.**

I. On Offshoring

The limited supply of highly competent software developers has created a strong trend towards cross-site distributed software development, be it within a company (insourcing) or with external partners (outsourcing) and be it within a single country (onshoring) or across borders (offshoring, either within the same world-region, as nearshoring, or beyond, as farshoring).

Software development is an extremely communication-intensive activity. Site distribution makes communication harder and therefore distributed software development is less efficient and more risky than local development and becomes more so with increasing distance. At least four types of distance are relevant: Geographical (physical distance), temporal (time zones), cultural (national/regional culture, company culture, language), and organisational (work processes).

Farshoring efforts frequently fail (and many companies are giving up farshoring) while nearshoring may work better but is still difficult, cumbersome, and risky. One might partition the issues that create the offshoring-specific difficulty into avoidable ones and essential ones.

The Avoidable Issues are problems for which effective and practical solution approaches are known in principle (and are more-or-less practiced by competent organizations). Key categories of such problems are lack of trust between the teams (e.g. due to lack of personal acquaintance), coordination problems due to lack of a common technical infrastructure or differences in work processes, a too-high language barrier, and a few others.

Essential Issues, in contrast, are problems for which no effective and practical solution is available today. There are only two Essential Issues:

- The difficulty of achieving common ground between the sites with respect to requirements understanding and
- the difficulty of achieving common ground between the sites with respect to the understanding of a software architecture while it still emerges.

The Essential Issues are known to be wicked problems even in non-distributed software engineering. For producing the understanding and then transporting it to all members of the team, documents provide only little help. Rather, both producing and transporting are normally facilitated primarily by large amounts of high-bandwidth face-to-face communication (that is, communication using all verbal and non-verbal communication channels and cues), but this is just what is *not* available in a distributed development setting.

II. Current Assumption Underlying Distributed Software Development

Today, essentially all offshoring software development appears to be structured according to the following assumption:

> Effective cross-site communication is very difficult. Therefore, we should avoid relying on it. Therefore, we should decouple development on one site from that on the other as best we can.

Let us call this the *current assumption*.

A. Consequences

The natural approach when designing a distributed software process from this assumption is a process based on document hand-over: The “home” site (where the requirements knowledge is located) writes up a careful specification, hands it over to the remote site, and the remote site can then perform most of the software development locally, with little need for cross-site communication.

This may reduce the impact of the Avoidable Issues, but will pronounce the impact of the Essential Issues: It can work well if the specification is precise (unambiguous), consistent, complete, and valid (i.e. reflects a correct understanding of the requirements) and the remote team is sufficiently competent (both technically and culturally) not to misread it; if only one of these many conditions is even partially violated, the need for communication rises steeply at many unforeseeable spots and the process, unprepared for this need as it is, will likely start going wrong.

In the past few years, more and more organizations have recognized that upfront requirements understanding (and specification writing) is often impractical. In non-distributed development, they have introduced agile team processes to overcome this problem and found that the fine-grained iterative nature of these processes helps a lot and brings many additional benefits as well.

In a distributed setting, corresponding practices such as distributed joint iteration planning, distributed joint daily stand-up meetings, distributed joint retrospectives, and the like are a step in the right direction, but the bandwidth of the communication thus provided is far too low: Non-distributed agile development relies on huge quantities of one-to-one communication (both formal and, in particular, informal) outside the above-mentioned team steps. Too much of this one-to-one communication tends to disappear in a distributed team, in particular in an offshoring setting.

III. Suggested New Assumption

Considering the amount of communication needed to overcome the Essential Issues in local development, we conclude that distributed development is stuck in the current assumption and needs to be released from it.

We propose to replace the current assumption with a fundamentally different one as follows:

> Effective cross-site communication is very difficult, but is crucial for solving the Essential Issues. Therefore, we should structure the development process such as to maximize cross-site communication. Therefore, we should proactively couple development on one site to that on the other in all those ways where we expect successful communication that involves the Essential issues and do it even where this coupling could be avoided.

Let us call this the *new assumption* and the resulting work style *Agile Offsharing*.

A. Consequences

The core idea for how to realize this proactive coupling is voluntary, close, cross-site, synchronous, pair-wise engineering collaboration on narrowly focused technical tasks, realized by a set of practices that include distributed pair programming, distributed walkthroughs, distributed side-by-side programming, distributed interactive pair reviews, and distributed pair debugging. This idea is based on the following research hypotheses:

1. Such practices provide *high-bandwidth* communication because each participant within a pair will be very active and engaged and multiple pairs can work at the same time.

2. They provide *enough* such communication because they are performed as much as needed: Even where we could decouple and modularize development to avoid crossing the site boundary, we often intentionally and purposefully do not do this and share development instead; hence the name Agile Offsharing.

3. They provide *effective* communication because they work mostly on the code level where communication content is so concrete that understanding can often be validated immediately.

4. They solve the Essential Issues because requirements understanding and architectural understanding will be part of the communication in small, easily digestible doses and each of these doses will be so relevant to the task-at-hand that the recipient will be eager to learn. Over time, a sufficiently complete requirements understanding will trickle over to the remote side and a joint architecture understanding will emerge.

IV. Research Required

If using the above techniques was straightforward, organizations would have picked them up already. But their use is in fact ridden with a broad set of difficulties that need to be solved, for instance:

- Technical: Which of the existing tools that support synchronous code-level collaboration are suitable? What new difficulties do they create? How to overcome those?
- Social: Pair work requires trust. How much prior acquaintance between pair members is necessary before pairs function properly? How to arrange it efficiently?
- Social: Successful pair work must be peer-to-peer. How to identify actual or perceived power imbalances? How to neutralize them effectively for the individual pair session? This involves issues of national culture, company culture, and provider/customer relationships.
- Language: How to identify which engineers are capable of crossing the language barrier reliably? In borderline cases: How to identify which pairs involving such people will work best? How to lift engineers from just below the barrier enough so they can cross it? How to train engineers to spot language-related misunderstandings quickly?
- Essential Issues: How to train engineers to recognize gaps in requirements understanding or architecture understanding in order to purposefully fill them? Ditto for non-gaps in order to avoid inefficient redundancy?
- Overall process: How to balance cross-site and within-site collaboration properly to maximize the spread of requirements understanding and architecture understanding? How to spot the best opportunities for productive cross-site collaboration? How to align local development processes such as to make the pairing sessions fit in smoothly? How to coordinate individual pairing sessions? How to match pair members?

At Freie Universität Berlin, we formulated the idea of Agile Offsharing in February 2011 and believe it is too ambitious for farshoring but very promising for nearshoring. Our research in this direction is slowly picking up speed and we encourage other groups to join.