Explaining Pair Programming Session Dynamics from Knowledge Gaps

Franz Zieris
zieris@inf.fu-berlin.de

Lutz Prechelt
prechelt@inf.fu-berlin.de
Motivation

• Expectations in industry: *Why pair-program?*
  - Better design and fewer defects
  - Learn from each other or together
  - Understand legacy parts of the software system

• Our Overall Research Goal
  - Understand how industrial pair programming actually works

• Research Question
  - What are the underlying mechanisms of knowledge transfer in pair programming?

• Intended Outcome
  - Advise practitioners
  - behavioral (anti-)patterns
Qualitative Data Analysis

- Grounded Theory approach
- Recorded industrial PP sessions (audio, webcam, screen)

Theoretical sampling: 26 sessions (9 companies, 16 pairs)
  - from a total of 67 sessions

= 39.5 hours total
Qualitative Data Analysis

Analysis of pair programmers' dialog:
- What do they ask for? What do they explain?
- What do they know? What do they learn?

26 pairings of professional developers
Observation 1: The Primary Gap

One partner already worked on the task
Observation 1: The Primary Gap

One partner already worked on the task

I'll show you what I did.

OK

= task-relevant system knowledge

current state of implementation, classes, call hierarchies, defects, test/build setup, configuration state, ...
What about more homogenous pairs?

One partner already worked on the task

Both partners with similar prior involvement

= task-relevant system knowledge
What about more homogenous pairs?

One partner already worked on the task

Both partners with similar prior involvement

- recently worked in code area
- basic knowledge of software
- working in unknown terrain

= task-relevant system knowledge
Observation 2: The Secondary Gap

Both partners with similar prior involvement

- Recently worked in code area
- Basic knowledge of software
- Working in unknown terrain

Let's look at the superclass

```javascript
console.log(myObj);
```
Observation 2: The Secondary Gap

Both partners with similar prior involvement

Acquiring System Knowledge together works for most pairs

... but not for all

= task-relevant system knowledge
A Different Kind of Knowledge

Task: implement test case

S knowledge (task-relevant system understanding)
Data structure holding the application state?
How to modify and read the state?
...

G knowledge (task-relevant general software development knowledge)
Syntax of programming language?
Higher-order functions?
Application framework?
Test framework?
...

Where is the initial value?
Type? Function? I don’t even know what this is.
Roles of S and G knowledge

In general: S needs must be addressed for productive work

S: Task-relevant System Knowledge

G: Task-relevant General Software Development Knowledge

Small G need: not problematic (e.g., not knowing some test feature)

Large G need: problematic (e.g., not knowing the test framework)
The G Opportunity

Do you know the Template Method pattern?

Why did she do that?

G Opportunity seized

G Opportunity not seized

G Opportunity
(e.g., knowing more about design patterns)

S: Task-relevant System Knowledge

G: Task-relevant General Software Development Knowledge
Overall Session Dynamic

1. Close Primary Gap
2. Close Secondary Gap
3. Seize G Opportunity

26 pairings of professional developers
Overall Session Dynamic

26 pairings of professional developers

Overall Dynamic
1. Close Primary Gap
2. Close Secondary Gap
3. Seize G Opportunity
Overall Session Dynamic

S

26 pairings of professional developers

G

Overall Dynamic
1. Close Primary Gap
2. Close Secondary Gap
3. Seize G Opportunity
Overall Session Dynamic

26 pairings of professional developers

Overall Dynamic
1. Close **Primary Gap**
2. Close **Secondary Gap**
3. Seize **G Opportunity**
Summary

• A lack of and differences in *system understanding* are more important than differences in *general programming experience*.

• What matters is *task-relevant* knowledge → different knowledge needs, different session dynamic.
A lack of and differences in system understanding are more important than differences in general programming experience.

What matters is task-relevant knowledge → different knowledge needs, different session dynamic.

Mutually satisfactory constellation: Complementary Gaps.
A lack of and differences in system understanding are more important than differences in general programming experience.

What matters is task-relevant knowledge → different knowledge needs, different session dynamic.

Mutually satisfactory constellation: Complementary Gaps.

Summary

The session EA1.

A lack of and differences in system understanding are more important than differences in general programming experience.

What matters is task-relevant knowledge → different knowledge needs, different session dynamic.

Mutually satisfactory constellation: Complementary Gaps.
Images

- Icon "design" by Adrien Coquet from the Noun Project
- Icon "Bug" by Minh Do from the Noun Project
- Icon "knowledge" by Olivia from the Noun Project
- Icon "Box" by No More Heroes from the Noun Project
- Icon "corner webs" by Kate Maldjian from the Noun Project
- Icon "Computer" by Denis Shumaylov from the Noun Project