Conceptual model for pair design and pair testing based on the characteristics of pair programming

Timo Alexander Missler*

Faculty of Technology,
University of Applied Sciences Emden/Leer,
Constantiaplatz 4, 26723 Emden, Germany
Email: timo.missler.occup@gmail.com
*Corresponding author

Eva-Maria Schön and María José Escalona

Department of Computer Languages and Systems,
University of Seville,
Av. Reina Mercedes S/N, 41012 Sevilla, Spain
Email: eva.schoen@iwt2.org
Email: mjescalona@us.es

Jörg Thomaschewski

Faculty of Technology,
University of Applied Sciences Emden/Leer,
Constantiaplatz 4, 26723 Emden, Germany
Email: joerg.thomaschewski@hs-emden-leer.de

Abstract: Pair programming is a method that is widespread in the field of agile software development (ASD) and is acknowledged as state of the art of programming. This article initially addresses the question of what constitutes the key attributes of pair programming. Therefore, we analysed the extent to which these attributes can be applied to the development-related areas of human-centred design (HCD) and the quality assurance (QA) of software. The results of this analyses lead to the presentation and consideration of a new conceptual model for the application of the attributes of pair programming in the context of pair design (HCD) and/or pair testing (QA). The discussion shows that a transferability and application is appropriate and that both HCD and QA benefit, particularly in terms of the product quality and product throughput time.

Keywords: pair programming; human-centred design; HCD; software quality; quality assurance; agile software development; ASD; conceptual model.

Reference to this paper should be made as follows: Missler, T.A., Schön, E-M., Escalona, M.J. and Thomaschewski, J. (2019) ‘Conceptual model for pair design and pair testing based on the characteristics of pair programming’, *Int. J. Agile Systems and Management*, Vol. 12, No. 3, pp.261–277.

Biographical notes: Timo Alexander Missler received his MSc in Computer Science and Media Applications at the University of Applied Science Emden/Leer (Germany) in 2018. He has been working in the biopharmaceutical industry for over ten years, with the emphasis on project management and SAP consulting. As of 2018, he changed into the role of IT Architect for enterprise architecture in operations, focusing on the digitalisation and agilisation of the pharma business. His current research interests are agile software development and digitalisation in the context of the pharma industry.

Eva-Maria Schön received her PhD in Computer Science from the University of Seville (Spain) in 2017. In her industry projects, she focuses on agile coaching and human-centred design. Her research interests are agile software development, requirements engineering and human-computer interaction.

María José Escalona received her PhD in Computer Science from the University of Seville, Spain in 2004. Currently, she is a Full Professor in the Department of Computer Languages and Systems at the University of Seville. She manages the web engineering and early testing research group. Her current research interests include the areas of requirement engineering, web system development, model-driven engineering, early testing and quality assurance. She also collaborates with public companies like the Andalusian Regional Ministry of Culture and Andalusian Health Service in quality assurance issues.

Jörg Thomaschewski received his PhD in Physics from the University of Bremen (Germany) in 1996. He became a Full Professor at the University of Applied Sciences Emden/Leer (Germany) in September 2000. His research interests are Internet applications for human-computer interaction, e-learning, and software engineering. He leads the 'Research Group for Agile Software Development and User Experience' and is the author of various online modules, such as 'Human-Computer Communication', which are used by the Virtual University (online) at many university sites. He has broad experience in research, training and consulting.

1 Introduction

Increasing numbers of companies are making use of agile process models, such as *scrum* (Schwaber, 2004), *Kanban* (Anderson, 2010) and *extreme programming* (XP) (Beck, 2000) for their product development. Through such use, the companies are hoping to gain advantages such as a reduced time-to-market or a flexible response to changing market or customer requirements.

In agile software development (ASD), a variety of agile techniques is used to develop the product in a cross-functional team on a collaborative basis. These agile techniques include, for example, the creation of user stories (Cohn, 2004), continuous delivery (Humble and Farley, 2010) and *pair programming* (Beck, 2000).

Pair programming is a technique in the field of ASD in which two programmers share one place of work and work together on the drafting, development and test implementation of source code – with the key goal of improving the quality of software (Williams and Kessler, 2003).

Although the concept has already existed for several decades (Constantine, 1995), it only became really well known with the advent of agile methods. In particular, XP (Beck,

2000) enabled *pair programming* to gain a high profile in the area of software development (Williams and Kessler, 2003). The strengthened levels of interest led to a critical discussion surrounding *pair programming* in the research literature. In addition to higher costs, the key points of criticism include possible problems with the technical and personal collaboration, which means the discussions of the method remain controversial to this day (Cockburn and Williams, 2001; Ally et al., 2005; Hulkko and Abrahamsson, 2005; Dybå et al., 2007; Hannay et al., 2009; Plonka and Van der Linden, 2012).

In this article, we initially addresses the question of what constitutes the key attributes of *pair programming* and we propose a conceptual model for the transferability of the attributes of pair programming to the new areas of *pair design* [human-centred design (HCD)] and *pair testing* [quality assurance (QA)]. To the best of our knowledge, there is no leading publication concerning such a model or conceptual model. In this respect, we aim to address the following research questions:

- RQ1: What are the key attributes of *pair programming*?

We start by analysing the key attributes of *pair programming* based on the existing literature. As demonstrated in previous studies, the attributes of *pair programming* can also be transferred to the related areas of software development (Schön et al., 2015). This leads us to our second research question:

- RQ2: How can the attributes of *pair programming* be transferred to development-related areas such as HCD and QA?

We also analyse the transfer of the attributes of *pair programming* to a *pair design* process in terms of HCD (International Organization for Standardization, 2010) and to a *pair testing* process in terms of QA. In addition, we create a conceptual model for the application of *pair design* and/or *pair testing* in the context of ASD.

The first *pair testing* work in the area of usability tests began in 1984. In this case, two test subjects are used in a usability test in order to carry out the *pair-user testing* (O'Malley et al., 1984). One test subject focuses on the actual completion of the test and the other test subject on the administration of the test. In this respect, O'Malley et al. (1984) describe an approach which, with the allocation of roles between the *test driver* and *scenario driver* demonstrates similarities with *pair programming* (Wildman, 1995) and implies a transferability of the attributes.

In Section 2, classic *pair programming* and the advantages and disadvantages discussed in the literature are clarified. In Section 3, the initial situation and the relationship between *pair programming*, *pair design* and *pair testing* are set out and the literature on *pair design* and *pair testing* is discussed. On this basis, in Section 4, we present and evaluate a conceptual model for the application of *pair design* (HCD) and/or *pair testing* (QA). The discussion in Section 5 includes a summary and a forecast for future research work.

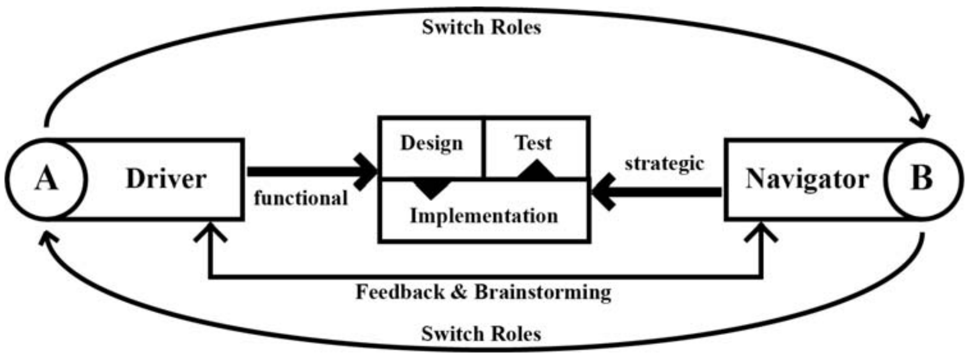
2 Background *pair programming*

In the early 1980s, in his deliberations on 'dynamic duos' in the area of software development, Constantine (1995) alludes to the faster creation of software code with fewer errors. In the mid-1990s, Coplien and Harrison (2004) published the organisational template *developing in pairs*, in which they describe the collaboration of compatible designers as being more effective. They argue that this is due to the unassailable nature of major problems, which appear unresolvable to the individual. In 1998, with the term *collaborative programming*, Nosek (1998) published the first empirical study on the effectiveness of *pair programming*. XP finally succeeded in raising the profile of *pair programming*, and found key use in both the original (Beck, 2000) and the revised version (Beck, 2005) of XP.

2.1 Basic principles of pair programming

At the heart of *pair programming* is the simultaneous collaboration between two programmers who share one workplace and complete their tasks together. Although the name *pair programming* appears to suggest otherwise, in addition to the actual programming, the method normally includes all of the tasks necessary in the software development work, starting from the architecture and design of the appropriate components through to the review, the integration and the integration tests (Williams and Kessler, 2003).

Figure 1 The interplay between the driver and navigator in *pair programming*



In terms of the collaboration, with *pair programming* two different roles are defined: the role of the *driver* and that of the *navigator* (see Figure 1):

- The *driver* is the productive part of the pair and carries out the actual development. S/he focuses on the operational goals of the software development and is primarily responsible for resolving the actual problems and for producing clean and run-capable code.
- The *navigator* takes a superior position and focuses on the strategic goals. His/her task is to provide immediate feedback to the driver and to check the results of the *driver* on a direct basis. In this respect, the focus is on the general improvement of the operational decisions and considerations surrounding the existing and future tasks and alternatives.

A key aspect of this role allocation is the communication between the two developers. The *driver* should provide commentary on his/her work and substantiate his/her decisions, while the *navigator* should question such decisions accordingly or provide added input. This places the navigator in the situation of being able to support the *driver* in the event of any questions or to be available to them for brainstorming (Williams and Kessler, 2003; Freudenberg et al., 2007; Shore and Warden, 2007).

To ensure an appropriate interplay between both roles, a regular changing of roles is of considerable importance. Figure 1 illustrates the overall interplay between the roles as regards the tasks.

Changing between roles ensures a continuous exchange of perspectives and knowledge surrounding the software between the participants. In bigger teams, it also serves the purpose of conveying a common basis of knowledge and concept. *Pair programming* does not provide any specific details on when this change should take place. This is instead determined by the pairs themselves at the appropriate point in time, using fixed periods or work packages for instance. In this respect, however, the changes generally occur based on an informal spoken arrangement (Williams and Kessler, 2003; Shore and Warden, 2007). In addition to a straightforward role change, the teams themselves can also be rotated – the talk is therefore of dynamic rather than static teams (Williams and Kessler, 2003; Swamidurai and Umphress, 2014).

The pairing itself also plays a key role. Therefore, when the team is put together, firstly, the question is raised as to which developers with different experience (beginners and experts) can be combined in an appropriate and productive way, and secondly, the influence that certain character attributes can have on the collaboration. This context and its influence on the composition in the *pairing* are considered in Chao and Atli (2006), Lui and Chan (2006), Braught et al. (2010) and Agrawal et al. (2014).

With regard to the analysed literature in this section, we propose to derive the following characteristics for *pair programming* (see Table 1).

Table 1 Attributes of *pair programming*

ID	*Attribute*	*Description*
C1	*Collaboration*	Both individuals share a workplace and work closely together.
C2	*Role allocation*	There is a clear division of roles between the two individuals (*driver* and *navigator*).
C3	*Role change*	Both individuals change between both roles on a continuous basis and therefore get both perspectives.
C4	*Communication*	Open, direct and continuous communication is the key component of the collaboration and enables the exchange of information between the individuals as well as feedback.

This Table 1 allows us to answer the first research question RQ1: what are the key attributes of *pair programming*?

2.2 Costs and benefits

The framework conditions of the *pair programming* result in immediate cost and benefit effects. Those in favour of this technique mention superior code in terms of a higher quality of software, and directly connected with this, a lower rate of error and generally lower throughput times during the development. This is based on the combined knowledge of both developers as well as the simultaneous review by the *navigator* (Williams et al., 2000; Cockburn and Williams, 2001; Menzies et al., 2003; Padberg and Müller, 2003; Williams and Kessler, 2003; Dybå et al., 2007; Begel and Nagappan, 2008; Hannay et al., 2009; Radermacher and Walia, 2011; Sun et al., 2015). In addition to this, the direct collaboration frequently leads to a higher degree of discipline, a superior work ethic and a solid flow of work, because the developers encourage one another, which drives things forwards together (Williams et al., 2000; Cockburn and Williams, 2001; Menzies et al., 2003; Williams and Kessler, 2003; Begel and Nagappan, 2008; Wray, 2010; Swamidurai and Umphress, 2014). Another key benefit is the mentoring of the participants. Due to the direct collaboration, there is a continuous exchange of specialist or project knowledge (Cockburn and Williams, 2001; Williams and Kessler, 2003; Hulkko and Abrahamsson, 2005; Begel and Nagappan, 2008; Wray, 2010; Radermacher and Walia, 2011; Sun et al., 2015). In addition to this, on the basis of the interaction, *pair programming* contributes directly to the team building and supports a feeling of collective responsibility towards the product, because the individuals know more about and better identify with the product as a whole (Cockburn and Williams, 2001; Hulkko and Abrahamsson, 2005; Begel and Nagappan, 2008).

On the other hand, the literature generally addresses the higher costs of *pair programming*, as the completion of a task appears to result in double the personnel costs (Williams et al., 2000; Cockburn and Williams, 2001; Williams and Kessler, 2003; Dybå et al., 2007; Begel and Nagappan, 2008; Hannay et al., 2009). The constant collaboration can also cause distractions and problems to which just one employee would not be exposed (Williams and Kessler, 2003; Begel and Nagappan, 2008). Firstly, the *pairing* can lead to interpersonal problems, because *pair programming* generally requires a very close collaboration. Secondly, in terms of required agile attributes such as teamwork and communication skills, the *pairing* is not suitable for every kind of person and requires special attention to apply successfully (Williams et al., 2000; Williams and Kessler, 2003; Dybå et al., 2007; Begel and Nagappan, 2008; Walle and Hannay, 2009; Agrawal et al., 2014; Ashmore et al., 2018). In addition to this, unequal pairings can mean one participant is over- or under worked (Begel and Nagappan, 2008; Braught et al., 2010; Wray, 2010; Plonka et al., 2012).

There have been several studies on the evaluation of the costs and benefits of *pair programming*, and the added value is to be viewed in the context of the application at all times. In this respect, the meta-analysis by Hannay et al. (2009) shows that there are numerous influencing factors regarding the success of *pair programming* and that the benefits of this technique cannot be generalised. The current studies also show that personal and organisational factors can have a major influence on costs and benefits (Braught et al., 2010; Plonka and Van der Linden, 2012; Agrawal et al., 2014; Swamidurai and Umphress, 2014; Socha and Sutanto, 2015).

With *inverted pair programming*, Swamidurai and Kannan (2014) and Swamidurai and Umphress (2015) have proposed an alternative approach to *pair programming* in which the collaboration only takes place during the design and the test phase. The actual programming is completed on the basis of individual work. The initial examinations show that the use of *pairing* has a positive impact during the design and test phase. In comparison with traditional *pair programming* it is evident that the same, or superior level of quality, can be achieved in less time and with lower costs (Williams et al., 2000; Cockburn and Williams, 2001; Menzies et al., 2003; Padberg and Müller, 2003; Williams and Kessler, 2003; Dybå et al., 2007; Begel and Nagappan, 2008; Hannay et al., 2009; Radermacher and Walia, 2011; Sun et al., 2015). The model from Swamidurai and Kannan (2015) therefore provides indications regarding the transferability of the attributes of *pair programming* to the areas of *pair design* (HCD) and *pair testing* (QA) that we are examining.

3 *Pair design* and *pair testing*

In their study, Schön et al. (2015) show that the concept of pair programming (see Table 1) can not only be used in programming, but can also optimise the operation in other domains, such as in conceptual design or QA.

3.1 Pair design

There have been several studies that address the specific question of the extent to which *pairing* has a positive impact on the quality of the software design, and the extent to which the resulting quality of the software can be influenced (Al-Kilidar et al., 2005; Müller, 2006; Canfora et al., 2007; Lui et al., 2008; Swamidurai and Kannan, 2014; Swamidurai and Umphress, 2015). Although these studies address a separation of design and implementation, this generally takes place in view of the software development, and accordingly, in view of the design of the software from the architectural and technical point of view. Swamidurai and Kannan (2014) and Swamidurai and Umphress (2015) as well as Müller (2006) show that in the area of technical software design, *pair design* has a positive impact on the quality and the development time. In addition, Al-Kilidar et al. (2005), Lui et al. (2008) and Canfora et al. (2007) show that *pairing* has a positive impact on the quality of the software design. Al-Kilidar et al. (2005) only confirm this context for tasks of low and middling complexity.

Our use of *pair design* (HCD) corresponds to the understanding and application in the context of the HCD.

3.2 Pair testing

The concept of *pair testing* mostly relates to the QA tasks downstream from the implementation, and forms part of the classic *pair programming*.

Williams et al. (2000) generally express the view that *pair testing* is a non-critical phase of the *pair programming* and are of the view that it is a process that can be shared around the team members at the same time. As long as the pair creates the test cases together, the shared completion of the test does not offer any noteworthy benefits.

Vanhanen and Lassenius (2005) noticed that although pairs write code that contains fewer errors, they are in fact less accurate with the tests, and therefore perform more poorly than individual programmers.

In contrast to this, Swamidurai and Kannan (2014) and Swamidurai and Umphress (2015) ascertained that on the basis of the collaboration, *pair testing* is highly beneficial to the quality and development time, and that it also reduces the overall costs.

With regards to the control activities carried out in the *pair programming*, there are studies by Müller (2004, 2005) that address the question of whether the completion of reviews and/or peer reviews can be an alternative to *pair programming* per se. There, he concludes that if their work is checked and/or corrected by independent programmers, pairs are not found to provide work, which is of superior quality to individual programmers.

3.3 Findings and research gap

With regard to RQ2 (how can the attributes of *pair programming* be transferred to development-related areas such as HCD and QA?), we were able to find some relevant articles (O'Malley et al., 1984; Wildman, 1995; Müller, 2004, 2005, 2006; Al-Kilidar et al., 2005; Canfora et al., 2007; Lui et al., 2008; Swamidurai and Kannan, 2014; Swamidurai and Umphress, 2015).

Based on analysed literature (see Sections 3.1 and 3.2), we are able to conclude that the characteristics of *pair programming* can be mapped to *pair design* and *pair testing*. When considering *pair design*, it becomes evident that there are several studies (Al-Kilidar et al., 2005; Müller, 2006; Canfora et al., 2007; Lui et al., 2008; Swamidurai and Kannan, 2014; Swamidurai and Umphress, 2015) in which the focus is on design in the context of the programming. It has not proven possible to find any publications in the research literature on the monitoring of *pair design* from the perspective of HCD (see HCD). It is clear that there is a research gap here. In terms of *pair testing*, studies that address the topic of QA were found (Wildman, 1995; Müller, 2004, 2005). A study was also found that examines *pair testing* in the context of the human-centred development (O'Malley et al., 1984).

4 Conceptual model for *pair design* and *pair testing*

The basis for our conceptual model of *pair design* (HCD) and *pair testing* (QA) is depicted by the four attributes of *pair programming* (see Table 1). The initial situation surrounding of *pairing* in HCD and QA can be easily compared with *pair programming*. The division into the three work phases of design, implementation and test suggest that both the allocation of roles and the changing of roles can be transferred from *pair programming* in the same and/or similar form. In all three fields, the design phase consists of the generation and gathering of ideas regarding the implementation of requirements. The subsequent implementation phase is oriented to the implementation and/or completion of these ideas in the form of field-specific artefacts. These are tested in the subsequent test phase and verified with regards to the requirements. The application of the four attributes from the *pair programming* (C1 to C4) is taken into account in our conceptual model as follows.

4.1 Overview of conceptual model

In the following, we use the characteristics of *pair programming* (see Table 1) to derive our conceptual model.

Like *pair programming*, our conceptual model is also based on attribute C1 (collaboration). The existing role designations of *driver* and *navigator* can be transferred to *pair design* (HCD) and *pair testing* (QA). Accordingly, the *driver* can be considered the driving force behind the creation and implementation of ideas and solutions and the completion of individual tasks. The *navigator*, by contrast, plays a strategic role. S/he controls, checks and assesses the individual results in terms of the overall concept.

The collaboration between *driver* and *navigator* generally takes place at a shared place of work. The *driver* plays the leading role and uses all of the required work equipment alone and independently. This ensures that the navigator is not able to act independently of the *driver*.

According to attribute C2 (role allocation), the *driver* and *navigator* assume different tasks. The *driver* takes on the generating part of the collaboration and addresses the actual completion of all necessary activities. The *navigator* plays a strategic role, as with *pair programming*. S/he checks and/or assesses all the activities of the *driver* as regards the compliance with the framework conditions, and provides continuous feedback. S/he is also available as a brainstorming partner for the provision of additional ideas and/or approaches and for steering the activities of the *driver*.

According to attribute C3 (role change), this approach is indispensable, as it is only possible to convey the overall picture and the detailed knowledge to both participants, enabling them to apply their ideas and abilities, in this way. S/he is encouraged by the *driver* to avoid interruptions to individual activities. The composition of the pairs is also an important part of the role changing. In *pair design* (QA), it is important to take a variety of different points of view and ideas into account in order to achieve the best possible level of quality. A dynamic *pairing* in which the individual people in the pairs can change on a regular basis enables additional people to be included in the process.

The communication between the *driver* and *navigator* described in attribute C4 (communication) takes a key role in our conceptual model for *pair design* (HCD) and *pair testing* (QA). An open, direct and continuous communication is indispensable for a successful collaboration. It enables the permanent exchange of information and feedback between the individuals and therefore ensures further development of ideas and concepts. During the implementation and verification, it also supports the completeness and accuracy of the results (Cockburn and Williams, 2001; Williams and Kessler, 2003; Hulkko and Abrahamsson, 2005; Freudenberg et al., 2007).

In addition to the general transfer of attributes C1 to C4 there are certain particularities of *pair design* (HCD) and *pair testing* (QA) that are highlighted in the two following sections.

4.2 Particularities of pair design (HCD)

With regards to attribute C2 (role allocation), in *pair design* (HCD) it is the driver who is responsible for the generation of ideas, implementation proposals and specific designs. His/her work is limited to individual, partial aspects or functions for implementation. As the work advances, s/he also draws on the known methods of the HCD for support. S/he has the goal of implementing all of the requirements surrounding the product and verifies his/her initial drafts regarding individual requirements. The *navigator* focuses on the overall idea and/or the overall draft. S/he considers the individual components in the context of the product requirements and attempts to integrate a coherent and complete overall concept on their basis. During the verification of the drafts, s/he focuses on the functional completeness and on a coherent integration in the overall products.

In contrast to *pair programming* with regard to the attribute of C3 (role change), it is recommended that the change in *pair design* (HCD) is linked to the level of detail of the work steps. For the successive and increasingly detailed activities of the design process, we propose to complete at least one role change for each design stage to enable both individuals to make an active contribution to the design (see Figure 2). On the one hand, the allocation of roles forces the designers to consider the product from both the operational and strategic level; while on the other hand, the allocation of roles supports the consideration of the ideas of both individuals. A role change should take place at least once in the four following levels of detail:

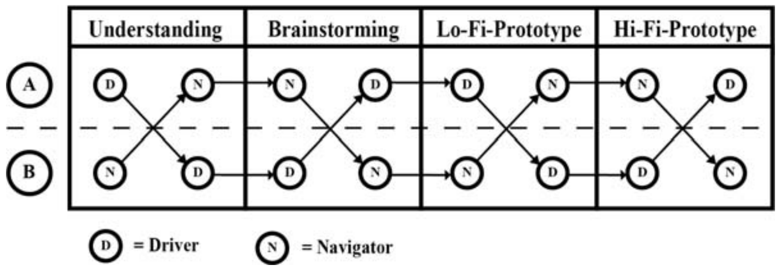
a gaining a personal understanding of the tasks

b generation of ideas

c creation of low-fi prototypes

d creation of high-fi prototypes.

Figure 2 Role change in *pair design* (HCD) (with a minimum number of role changes)



The role change also remains a dynamic interplay between both individuals and supports the creative and iterative design process. A role change within the work phases should not take place too frequently, however, as otherwise, it may not be possible for ideas to be brought to their conclusion, and the creative process will therefore be interrupted.

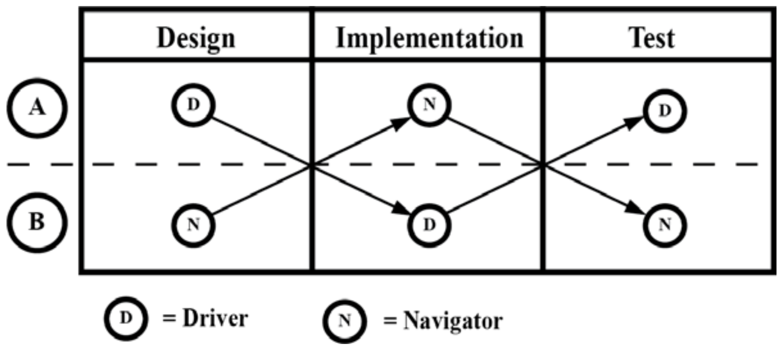
With our conceptual model for *pair design* (HCD) we are therefore able to demonstrate that the attributes of *pair programming* can be transferred to HCD. In this respect, the tasks regarding attribute C2 (role allocation) are aligned to the refinement of product requirements. And it is clear that attribute C3 (role change) has to be connected to the level of detail for the work steps.

4.3 Particularities of pair testing (QA)

With the *driver*, the tasks according to attribute C2 (role allocation) are aligned to the conceptualisation and formulation of test cases, the subsequent completion and documentation and the final checking of the results. Throughout this time, the *navigator* checks the contents and formulations for their completeness and purpose as regards the product and/or the aspects and requirements in need of checking. During the completion and documentation, s/he monitors the correctness and completeness. Similar to *pair-user testing* (O'Malley et al. 1984) s/he is available for the clarification of questions of comprehension. Based on test cases, s/he guides the *driver* through all of the test activities so that s/he (the driver) is able to focus fully on the completion and the documentation. During the subsequent verification, s/he checks the work of the *driver* as regards the compliance with the formal aspects and the completeness of the contents. This enables any possible additional review activities to be replaced.

In terms of the manifestation of attribute C3 (role change), a stricter approach should be used for *pair testing* (QA). According to Vanhanen and Lassenius (2005), in the case of familiar topics, pairs tend to be less accurate during the testing phase. For this reason, the role change should be determined based on the contents of the individual phases in order to reduce such a degree of blindness towards the contents. During the execution phase of test cases, changing between the roles is recommended (see Figure 3).

Figure 3 Role change during *pair testing* (QA)



The result of this is that during the creation of the test cases, the *driver* is ultimately responsible for the execution. Conversely, it forces the *navigator* out of the strategic perspective in the creation phase into the operational perspective with the active completion of the test cases. This supports the detection of discrepancies, which would otherwise have been taken for granted.

With our conceptual model for *pair testing* (QA) we provide a concept for transferring the attributes of *pair programming* to the area of QA. Further, attribute C2 (role allocation) is configured on the basis of the collaboration between the tester and test coordinator. Attribute C3 (role change) is also applied very strictly, and specifically links the role change to the drafting, implementation and test phases.

5 Discussion and limitations

The described application of attributes C1–C4 of the *pair programming* (see Table 1) to the *pair design* (HCD) and *pair testing* (QA) models also results in a transfer of the known advantages and disadvantages (see chapter 2). Following Schön et al. (2017) these can be assigned to the four dimensions of '*product*', '*project*', '*process*' and '*human*' of the software development process. Based on this allocation, it becomes clear that the majority of the advantages and disadvantages are not connected to the actual product and can accordingly be seen as being independent. The affected aspects and the degree of transfer are shown in Table 2.

Table 2 List of advantages and disadvantages for *pair design* (HCD) and *pair testing* (QA)

	Dimension	*Aspect*	*Pair design (HCD)*	*Pair testing (QA)*
Advantages	Product	Error rate	+	o
		Quality	+	+
	Project	Throughput time	+	+
	Process	Mentoring	+	+
		Team building	+	+
		Sense of responsibility	+	+
	Human	Work flow	+	+
		Work ethic	+	+
		Discipline	+	+
Disadvantages	Product/ Project	Costs	+	o
	Process	Diversion	+	+
		Excessive/insufficient workload	+	+
	Human	Suitability	+	+
		Personality	+	+

Notes: + = aspect transferable, o = aspect partially transferable, – = aspect not transferable.

When considering the 'product' and 'project' dimensions (see Table 2), the *pairing* in HCD and QA of software along the lines of *pair programming* have a positive impact. Through the direct collaboration, the pairs are able to become complementary to one another and collectively have a greater degree of specialist knowledge, more experience, and exchange knowledge with each other. A continuous assessment of all the ideas and work steps also take place. This enables errors to be discovered and fixed at an earlier stage, which ultimately results in a lower throughput time for the entire project. Due to the continuous strategic monitoring, the change between the operational and strategic perspectives supports the creation of complete artefacts. There have already been several studies in the area of *pair programming* (Müller, 2006; Canfora et al., 2007; Lui et al., 2008; Swamidurai and Umphress, 2015) which confirm the positive effects of *pairing* in the design phase of software development and demonstrate the possibility of its transferability to related areas. There are also studies (Swamidurai and Kannan, 2014; Swamidurai and Umphress, 2015) that confirm these benefits for the test phase. However, the studies by Vanhanen and Lassenius (2005) show that pairs tend to be less accurate during the completion of the testing, which is why to date, the error rate in the *pair testing* (QA) can only be viewed as being partially transferable.

With regards to the consideration of the 'human' dimension (see Table 2), it is possible to determine that the advantages do not result from the concept of *pair programming*, but directly from the collaboration in pairs. Therefore, these aspects are relevant to both *pair design* (HCD) and *pair testing* (QA), because a continuous collaboration in a team also takes place here. The same applies to the aspects of team building and the shared sense of responsibility of the 'process' dimension (see Table 2). The sustained and direct interaction between the two individuals, as determined by the process, brings them together and supports the common spirit. The direct collaboration on the individual tasks and the role change also increase the identification with and understanding of the product.

Another advantage of the 'process' dimension is the mentoring, which takes place due to the interaction between the actors. The ongoing communication, the feedback and the consideration of the work steps mean that there is a constant exchange of knowledge and experience between *driver* and *navigator*. The role change also means that this transfer of knowledge takes place in both directions, and in addition to specialist knowledge, it also includes the exchanging of proven approaches, the correct use of tools and processes internal to the team or the organisation (Cockburn and Williams, 2001). In comparison to the areas of team building and sense of responsibility, mentoring is to be seen as being of limited use in *pair testing* (QA). Firstly, the QA mostly involves standardised approaches (Copeland, 2003), so that there are fewer task-specific learning effects. Secondly, in this area, it is frequently the case that traditional reviews are completed that have a similar added value (Müller, 2004, 2005) and also support an exchange of knowledge without the use of *pair testing* (QA).

The advantages are set against the transfer of the stated disadvantages (see Section 2), because they are predominantly independent of the product and can be transferred to *pair design* (HCD) and/or *pair testing* (QA). With regard to the dimensions of 'product' and 'project', the use of *pairing* means that two individuals are generally required. Double individuals therefore initially means higher costs. The improvement of the quality and the throughput time as well as the extended generation of ideas means that the use of two individuals does not necessarily mean twice the costs, however. Hannay et al. (2009) study completed on *pair programming* show time savings of approximately 10% with improved quality. The current use of reviews in the scope of the QA of software further weakens the disadvantage of the additional costs for *pair testing* (QA), as the review activities are already included in the application of *pair testing* (QA).

Moreover, the disadvantages of the 'process' dimension are transferred to *pair design* (HCD) and/or the *pair testing* (QA) as they themselves are justified in the *pairing*. Regardless of the area of application, a permanent collaboration will lead to diversions if the communication is no longer oriented to the actual matter. *Pairing* requires a constant communication and also requires a minimum level of empathy and willingness to compromise in order to avoid conflicts and to ensure that things run smoothly (Begel and Nagappan, 2008). The possibility of team members having an excessive/insufficient workload in terms of the 'process' dimension is based on methodological differences and/or a lack of specialist knowledge (Vanhanen et al., 2007). Therefore, these aspects need to be taken into consideration in the composition of the team in *pair design* (HCD) and *pair testing* (QA). As a general rule, a regular verification of the communication and a possible adaptation of the pairs are beneficial for the collaboration.

The preceding discussion of the role allocation, the role change and the resulting advantages and disadvantages shows that an application of the rules of the *pair programming* in the form of *pair design* (HCD) and *pair testing* (QA) is easily conceivable and can also be put to effective use.

In this respect, use in *pair design* (HCD) appears to be ideal, because the tasks and activities are very similar. In both cases, it relates to the development and implementation of creative solutions under specific parameters, so that the rules can also be applied to the collaboration between the designers.

In *pair testing* (QA) a similar picture becomes evident. The test phase may represent a less critical activity (Williams et al., 2000) in *pair programming*, but the pairing can still contribute to an increase in quality (Swamidurai and Kannan, 2014; Swamidurai and Umphress, 2015). The completion of reviews is also an important factor for the quality (Müller, 2004, 2005). There is also a need for further clarification in terms of *pair testing* (QA). Firstly, it is not clear in the theoretical consideration as to whether the navigator is necessary in the test phase, and is able to contribute to a further increase in quality or a reduction in the throughput time. Secondly, it is necessary to clarify whether the findings by Vanhanen and Lassenius (2005), of pairs being less accurate during the testing also holds true in the *pair testing* (QA) and/or whether this disadvantage is sufficiently minimised due to the suggested role change.

Hannay et al. (2009) report that the use of *pair programming* is neither beneficial nor effective in every situation. The same statement can also refer to *pair design* (HCD) and *pair testing* (QA) – the application of such methods depends on the appropriate context, consisting of the actual task, the people involved and the organisational environment, and does not present a conclusive solution for all of the tasks and problems.

In addition to the aforementioned drawbacks and limitations our proposed model has not been validated in industry, so far. Hence, there may be some more benefits or further limitations due to contextual settings in organisations.

6 Conclusions and future works

This article summarises the key attributes of *pair programming*: collaboration, role allocation, role change and communication (see Table 1). In addition, we presented the advantages and disadvantages of *pair programming* (see Table 2).

With respect to our second research question, it proved possible to transfer the attributes of *pair programming* to the development-related areas of HCD and QA. This result was deduced by an analysis of existing literature.

Moreover, we contribute a new conceptual model for pair design and pair testing. This conceptual model describes the application of *pair design* (HCD) and/or *pair testing* (QA) and allows us to demonstrate how the identified attributes of *pair programming* can be transferred in detail.

In future work, we will conduct empirical studies in order to validate the conceptual model in industry. It will be necessary to assess the validity of the assumptions and demonstrated concepts for *pair design* (HCD) and *pair testing* (QA). The validation process will include controlled experiments with which we want to assess the feasibility, costs and benefits of the proposed model in order to study the practical application.

References

Agrawal, A., Singh, S., Tripathi, M. and Maurya, L.S. (2014) 'A study on role of personality traits for pair programming team', in *Proceedings on the International Conference on High Performance Computing and Applications (ICHPCA)*, IEEE, New York, pp.402–407.

Al-Kilidar, H., Parkin, P., Aurum, A. and Jeffery, R. (2005) 'Evaluation of effects of pair work on quality of designs', in *Proceedings of the 2005 Australian conference on Software Engineering*, IEEE, Washington, DC, pp.78–87.

Ally, M., Darrach, F. and Toleman, M. (2005) 'A framework for understanding the factors influencing pair programming success', in *Proceedings of the 6th international conference on Extreme Programming and Agile Processes in Software Engineering*, Springer, Berlin, Heidelberg, pp.82–91.

Anderson, D.J. (2010) *Kanban: Successful Evolutionary Change for your Technology Business*, Blue Hole Press, Sequim, WA.

Ashmore, S., Townsend, A., DeMarie, S. and Mennecke, B. (2018) 'An exploratory examination of modes of interaction and work in waterfall and agile teams', *Int. Journal of Agile Systems and Management*, Vol. 11, No. 1, pp.67–102.

Beck, K. (2000) *Extreme Programming Explained: Embrace Change 1*, Addison-Wesley, Reading, MA.

Beck, K. (2005) *Extreme Programming Explained: Embrace Change 2*, Addison-Wesley, Boston, MA.

Begel, A. and Nagappan, N. (2008) 'Pair programming: what's in it for me?', in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ACM, New York, pp.120–128.

Braught, G., MacCormick, J. and Wahls, T. (2010) 'The benefits of pairing by ability', in *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, ACM, New York, pp.249–253.

Canfora, G., Cimitile, A., Garcia, F., Piattini, M. and Visaggio, C.A. (2007) 'Evaluating performances of pair designing in industry', *Journal of Systems and Software*, Vol. 80, No. 8, pp.1317–1327.

Chao, J. and Atli, G. (2006) 'Critical personality traits in successful pair programming', in *Proceedings of the conference on AGILE 2006*, IEEE, Washington, DC, pp.89–93.

Cockburn, A. and Williams, L. (2001) 'The costs and benefits of pair programming', in *Extrem. Program. examined.*, pp.223–243, Addison-Wesley, Boston, MA.

Cohn, M. (2004) 'User stories applied: for agile software development', *Addison Wesley Signature Series*, 1st ed., Addison-Wesley, Boston, MA.

Constantine, L. (1995) *Constantine on Peopleware*, Yourdon Press, Englewood Cliffs, New Jersey.

Copeland, L. (2003) *A Practitioner's Guide to Software Test Design*, Artech House, Norwood, MA.

Coplien, J.O. and Harrison, N.B. (2004) *Organizational Patterns of Agile Software Development*, Prentice-Hall, Inc., Upper Saddle River, New Jersey.

Dybå, T., Arisholm, E., Sjøberg, D.I.K., Hannay, J.E. and Shull, F. (2007) 'On the effectiveness of pair programming', *IEEE Software*, Vol. 24, No. 6, pp.12–15.

Freudenberg, S., Romero, P. and Du Boulay, B. (2007) 'Talking the talk: is intermediate-level conversation the key to the pair programming success story?', in *Proceedings of the AGILE*, IEEE, Washington, DC, pp.84–91.

Hannay, J.E., Dybå, T., Arisholm, E. and Sjøberg, D.I.K. (2009) 'The effectiveness of pair programming: a meta-analysis', *Information and Software Technology*, Vol. 51, No. 7, pp.1110–1122.

Hulkko, H. and Abrahamsson, P. (2005) 'A multiple case study on the impact of pair programming on product quality', in *Proceedings of the 27th International Conference on Software Engineering*, IEEE, Saint Louis, MO, pp.495–504.

Humble, J. and Farley, D. (2010) *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, 1st ed., Addison-Wesley, Boston, MA.

International Organization for Standardization (2010) *ISO 9241-210: Ergonomics of Human-system Interaction – Part 210: Human-centred Design for Interactive Systems*, ISO.

Lui, K.M. and Chan, K.C.C. (2006) 'Pair programming productivity: novice-novice vs. expert-expert', *International Journal of Human Computer Studies*, Vol. 64, No. 9, pp.915–925.

Lui, K.M., Chan, K.C.C. and Nosek, J.T. (2008) 'The effect of pairs in program design tasks', *IEEE Transactions on Software Engineering*, Vol. 34, No. 2, pp.197–211.

Menzies, T., Smith, J. and Raffo, D. (2003) *When is Pair Programming Better?*, pp.1–9 [online] http://menzies.us/pdf/04pairprog.pdf (accessed 15 October 2018).

Müller, M.M. (2004) 'Are reviews an alternative to pair programming?', *Empirical Software Engineering*, Vol. 9, No. 4, pp.335–351.

Müller, M.M. (2005) 'Two controlled experiments concerning the comparison of pair programming to peer review', *Journal of Systems and Software*, Vol. 78, No. 2, pp.166–179.

Müller, M.M. (2006) 'A preliminary study on the impact of a pair design phase on pair programming and solo programming', *Information and Software Technology*, Vol. 48, No. 5, pp.335–344.

Nosek, J.T. (1998) 'The case for collaborative programming', *Communications of the ACM*, Vol. 41, No. 3, pp.105–108.

O'Malley, C.E., Draper, S.W. and Riley, M.S. (1984.) 'Constructive interaction: a method for studying human-computer-human interaction', in Shackel, B. (Ed.): *Proceeding of IFIP INTERACT '84: Human-Computer Interaction*, pp.269–274.

Padberg, F. and Müller, M.M. (2003) 'Analyzing the cost and benefit of pair programming', in *Proceedings of the 9th International Symposium on Software*, IEEE, Washington, DC, pp.166–177.

Plonka, L. and Van der Linden, J. (2012) 'Why developers don't pair more often', *Proceedings of the 5th International Workshop on Co-operative and Human Aspects of Software Engineering*, pp.123–125.

Plonka, L., Sharp, H. and Van Der Linden, J. (2012) 'Disengagement in pair programming: does it matter?', in *Proceedings of the 34th International Conference on Software Engineering*, IEEE, pp.496–506.

Radermacher, A.D. and Walia, G.S. (2011) 'Investigating the effective implementation of pair programming: an empirical investigation', in *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, ACM, New York, pp.655–660.

Schön, E-M., Escalona, M.J. and Thomaschewski, J. (2015) 'Agile values and their implementation in practice', *International Journal of Interactive Multimedia and Artificial Intelligence*, Vol. 3, No. 5, pp.61–66.

Schön, E-M., Thomaschewski, J. and Escalona, M.J. (2017) 'Agile requirements engineering: a systematic literature review', *Computer Standards and Interfaces*, Vol. 49, No. 1, pp.79–91.

Schwaber, K. (2004) *Agile Project Management with Scrum*, 1st ed., Microsoft Press, Redmond, WA.

Shore, J. and Warden, S. (2007) *The Art of Agile Development*, 1st ed., edited by O'Brien, M., O'Reilly Media, Sebastopol, CA.

Socha, D. and Sutanto, K. (2015) 'The 'pair' as a problematic unit of analysis for pair programming', in *Proceedings of the 8th International Workshop on Cooperative and Human Aspects of Software Engineering*, IEEE, Piscataway, New Jersey, pp.64–70.

Sun, W., Marakas, G. and Aguirre-Urreta, M. (2015) 'Effectiveness of pair programming: perceptions of software professionals', *IEEE Software*, Vol. 33, No. 4, pp.72–79.

Swamidurai, R. and Kannan, U. (2014) 'Impact of pairing on various software development phases', in *Proceedings of the 2014 ACM Southeast Regional Conference*, ACM, New York pp.1–6.

Swamidurai, R. and Umphress, D. (2014) 'The impact of static and dynamic pairs on pair programming', in *Proceedings of the 2014 IEEE 8th International Conference on Software Security and Reliability-Companion*, IEEE, Washington, DC, pp.57–63.

Swamidurai, R. and Umphress, D. (2015) 'Inverted pair programming', in *Proceedings on the 2015 IEEE Southest Con.*, IEEE, Washington, DC, pp.1–6.

Vanhanen, J. and Lassenius, C. (2005) 'Effects of pair programming at the development team level: an experiment', in *Proceedings on the 2005 International Symposium on Empirical Software Engineering*, IEEE, Washington, DC, pp.336–345.

Vanhanen, J., Lassenius, C. and Mäntylä, M.V. (2007) 'Issues and tactics when adopting pair programming: a longitudinal case study', in *Proceedings on the 2nd International Conference on Software Engineering Advances*, IEEE, Washington, DC, pp.1–7.

Walle, T. and Hannay, J.E. (2009) 'Personality and the nature of collaboration in pair programming', in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, IEEE, Washington, DC, pp.203–213.

Wells, D. (2013) *Extreme Programming* [online] http://www.extremeprogramming.org/ (accessed 15 October 2018).

Wildman, D. (1995) 'Getting the most from paired-user testing', *Interactions*, Vol. 2, No. 3, pp.21–27.

Williams, L. and Kessler, R.R. (2003) *Pair Programming Illuminated*, 1st ed., Addison-Wesley, Boston, MA.

Williams, L., Kessler, R.R., Cunningham, W. and Jeffries, R. (2000) 'Strengthening the case for pair programming', *IEEE Software*, Vol. 17, No. 4, pp.19–25.

Wray, S. (2010) 'How pair programming really works', *IEEE Software*, Vol. 27, No. 1, pp.50–55.