Bringing the Benefits of Agile Techniques Inside the Classroom: A Practical Guide

Ilenia Fronza, Nabil El Ioini, Claus Pahl and Luis Corral

Abstract Besides professional programmers, many "end-user programmers" write code in their daily life. Given that so much of end-user-created software suffers from quality problems, Software Engineering (SE) is no longer solely applicable to the professional context: a clear computational processing can be useful in everyday life. While the expansion of programming skills acquisition initiatives in K-12 (i.e., primary and secondary schools) has contributed to improving learners' coding ability, there have not been many studies devoted to the teaching/learning of SE concepts. In this chapter, we focus on understanding how it is possible to bring the benefits of Agile techniques inside the classroom. Moreover, our goal is to show how each selected practice (such as user stories and pair programming) can be leveraged or adapted to the educational context; to this end, tools already adopted in schools are considered as possible substitutes of professional ones.

Keywords K-12 · Agile techniques · XP practices · Classroom · Toolbox

1 End-User Software Engineering in K-12: Introduction

Computer programming is becoming a pervasive practice, almost as much as computer use; therefore, the gap between software users and developers is quickly narrowing down (Ye & Fischer, 2007). End-user programming has empowered millions

I. Fronza (✉) · N. El Ioini · C. Pahl
Free University of Bozen-Bolzano, Piazza Domenicani 3, 39100 Bolzano, Italy
e-mail: ilenia.fronza@unibz.it

N. El Ioini
e-mail: nabil.elioini@unibz.it

C. Pahl
e-mail: claus.pahl@unibz.it

L. Corral
Monterrey Institute of Technology and Higher Education, E. Gonzalez 500, 76130 Queretaro, Mexico
e-mail: lrcorralv@itesm.mx

D. Parsons and K. MacCallum (eds.), *Agile and Lean Concepts for Teaching and Learning*, https://doi.org/10.1007/978-981-13-2751-3_7

of end-users to create their own software: more and more people are not only using software but also taking part to the software development process to widely varying degrees to solve different types of problems. Unfortunately, there is a downside: errors are pervasive in end-users-created software (Burnett, 2009). End-User Software Engineering (EUSE) aims to increase the quality of end-user-created software by looking beyond the "create" part of software development, which is already well supported, to the rest of the software lifecycle (Burnett, 2009).

A key difference between EUSE and traditional software engineering (SE) is that EUSE focuses on "end-users". In today's EUSE literature, the definition of end-user developers differs from paper to paper (Burnett & Myers, 2014). Some researchers consider people who have neither experience nor software-engineering-oriented interests; others use it to mean anyone using a particular programming environment that targets end-users; and others use it to mean people who can program in a specific application area only. Ye and Fischer (2007) proposed a spectrum of software-related activities: at the right end of the spectrum are those that develop software systems professionally, while at the left side are those users who just use software systems to accomplish their daily tasks (the so-called "pure end-users").

As a *desiloing alternative*, Burnett and Myers (2014) suggested defining end-users based on *intent*, to focus directly on the end-user developers' motivations and values in that particular software development task, which in turn can affect their SE processes (Burnett & Myers, 2014).

Chimalakonda and Nori (2013) identified the following two main categories of challenges that need to be solved when teaching SE to end-users

- *End-users* concerns: (i) end-users do not see the value of learning the principles of SE; (ii) end-users generally program because they want to simplify or automate their own tasks and not to learn SE; (iii) learning SE requires extra time and effort.
- *Instructional design* concerns: (i) end-users focus on domain-specific goals and not on SE; (ii) SE didactics must be adapted to diversified audiences with different backgrounds and needs; (iii) there is a wide variety of processes in SE, but which of them to use in teaching is not clear; (iv) while there are many resources and tools on the web for learning SE, they are not directly usable for end-users.

Thanks also to the strong emphasis being led by the media, coding has been included in many teaching curricula. Therefore, in recent years, the most comprehensive didactic initiatives for end-users have been associated with K-12 (Monteiro, de Castro Salgado, Mota, Sampaio, & de Souza, 2017), also in non-vocational schools (Fronza et al., 2014). In K-12, some students have the intent to learn to program, which means that they are willing to move from the left- to the right end of the spectrum of software-related activities (Ye & Fischer, 2007). Other students can be positioned in the middle of the spectrum: they have certain software development skills, but just develop software with the intent to solve specific problems (Costabile, Mussio, Parasiliti Provenza, & Piccinno, 2008).

Because of the pervasiveness of software in the labor market, it is indeed of paramount importance to equip K-12 students with the necessary means to improve software quality: they will probably be end-users also in their future career, and at

that point producing dependable software could be crucial. Moreover, the benefits of a SE approach can be extended well beyond the engineering field: for example, the ability to work iteratively and incrementally can also be employed in other disciplines and in daily life as well.

In this chapter, we focus on understanding how it is possible to bring the benefits of Agile techniques to K-12 education. To this end, we first describe a toolbox of practices, by showing how they can be leveraged or adapted to the educational context; in particular, tools already adopted in schools are considered as possible substitutes of the professional ones. Then, we detail a selection strategy, to support teachers in selecting the right practice from this toolbox, based on different characteristics of their teaching activities. There are limited examples of Agile training in K-12, and most of these studies have focused on a limited number of practices applied in isolation. The goal of the proposed toolkit is then to leverage a synergistic use of practices.

2 End-User Software Engineering in K-12: State of the Art

There are currently only a few studies that explicitly address the need for teaching SE to end-users. An overview of these studies can be found in Monteiro et al. (2017).

Umarji, Pohl, Seaman, Koru, and Liu (2008) conducted a survey in the context of bioinformatics, which focused on the software engineering principles that should be learned by end-users who develop software in the bioinformatics domain. According to the authors, most of the bioinformatics students do not receive formal software engineering training, and usually learn programming principles through self-teaching. As a conclusion, the authors suggested to include software engineering principles in bioinformatics education.

Gross, Herstand, Hodges, and Kelleher (2010) described a code reuse tool for the Looking Glass IDE, which aimed at showing the importance of reuse to middle school students who are learning programming. Using this tool, students can reuse parts of programs written by others, even without understanding completely how that code works.

Some research on Agile in K-12 has recently been performed. Meerbaum-Salany and Hazzan (2010) presented an Agile mentoring methodology for high schools. In 2015, Fronza, El Ioini, and Corral (2015a) implemented a computational thinking training course in which they leveraged the software development phases (i.e., feasibility, analysis, design, development, testing, and integration) to foster specific computational thinking skills. The same group of researchers, in 2017, focused on the instructional design concerns in teaching software engineering to end-users (see Sect. 1), by proposing a framework in which a set of Agile practices were adapted to the middle school context in order to teach computational thinking (Fronza, Ioini, & Corral, 2017).

In 2016, Kastl, Kiesmüller, and Romeike (2016) achieved greater flexibility in software development projects in three secondary schools in Germany, by applying Agile methods.

Monteiro et al. (2017) analyzed how the technology used in a Brazilian CTA program prefigured elements of software engineering in the participants' programs created with AgentSheets.

To guarantee that SE will enter the K-12 environment, a set of how-to's is needed for teachers, so that they will be able to apply SE practices in their classrooms.

3 Bringing Agile to K-12 Education

In the first part of this section, we detail why Agile practices fit the K-12 environment, based on the end-user characteristics and work habits in this specific context. Then we explain why, among the possible Agile methodologies, eXtreme Programming (XP) represents a good candidate for K-12 students.

3.1 Why Agile?

One of the main challenges in EUSE is to understand which SE process to use in teaching, among the wide variety of possibilities (Chimalakonda & Nori, 2013). With respect to this goal, the EUSE approach is to respect end-users' real intentions and work habits, without advocating to transform end-users into software engineers (Burnett, 2009).

Under a plan-driven development model, once the requirements have been fully specified, the project continues through the design, coding, testing and integration phases, finally leading to deployment of the finished product. In a professional environment, this structure facilitates the creation of contracts, as the product definition is stable. In K-12, a plan-driven approach could then encourage a continuing focus on the product while, according to EUSE goals, students should also learn how to better organize the development process (Steghöfer et al., 2016). Moreover, one of the well-known disadvantages of plan-driven development models is that once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought out in the concept stage. Also, no working software is produced until late during the life cycle. Therefore, using a plan-driven process in K-12 could easily lead to having students with non-working products, which might actually be used to intentionally teach students how to fail, persevere and respond with resilience (Lottero-Perdue & Parry, 2017). However, this goal cannot be achieved if the failure just appears at the end of the project when students do not have time to learn from their failure and improve their product. This explains the importance of getting students used to have a working product (even with a minimum set of functionalities) at any point in time during the development (Fronza et al., 2017).

Indeed, the undesirable output of an iteration can encourage tenacity and urge to improve. On the other hand, a tangible good result of the performed activities can increase self-esteem in students. Moreover, evidences about the students' progress support formative assessment. Finally, a plan-driven development process is not an option when the goal is to promote creativity, experimentation, and practical work in students. An incremental approach, instead, can help achieve these goals in the engineering field. The same goals are relevant to many disciplines in a teaching curriculum; therefore, the benefits of learning an Agile approach can be applied outside the context of SE.

Burnett and Myers (2014) proposed to frame the EUSE problem more along the lines of the problem-solving activity rather than the SE lifecycle, by supporting end-users as they work:

- end-users work opportunistically, incrementally, and rarely one lifecycle phase at a time (Burnett & Myers, 2014);
- end-users exploratory mix programming, testing, and debugging, mostly by trial-and-error (Burnett & Myers, 2014);
- end-users are capable of constructing software by direct manipulation (Costabile et al., 2008);
- end-users very much prefer collaborative activities (Costabile et al., 2008).

The modern teaching approaches suggest to encourage and leverage the above-mentioned work habits. Examples of these teaching approaches are: collaborative learning (Barkley, Cross, & Major, 2014), learn-by-doing (Moye, Dugger Jr., & Starkweather, 2014), use of tangibles (Price, Rogers, Scaife, Stanton, & Neale, 2003), and project-based learning (Krajcik & Blumenfeld, 2006). Moreover, as shown in Table 1, the values in the Agile Manifesto[1] can be mapped to the K-12 learning environment, considering the above-mentioned end-user's habits and modern teaching methods (Stewart, DeCusatis, Kidder, Massi, & Anne, 2009). In this context, the instructor can play the role of a customer (Steghöfer et al., 2016). Indeed, considering the instructor as a customer (and not as a source of continuous support) can help in fostering students' self-organization on their projects by reducing their dependence on the instructor's assistance (Kastl et al., 2016).

Nevertheless, the waterfall development model has been for a long time the development strategy taught in schools and universities (Kropp & Meier, 2013). Switching to Agile, in fact, would require switching to an environment in which the *process* by which the students arrive at the product is emphasized (Steghöfer et al., 2016). Teaching Agile principles has recently drawn researchers' attention, but papers and experience reports in which the authors discuss their experiences are mostly set at university level (Alégroth et al., 2015; Astrachan, Duvall, & Wallingford, 2001; Mahnic, 2012; Paasivaara, Heikkilä, Lassenius, & Toivola, 2014).

[1] https://www.agilealliance.org/agile101/the-agile-manifesto/.

Table 1 Mapping the agile manifesto to the K-12 environment

Values in the agile manifesto	K-12 environment
Individuals and interactions over processes and tools	End-users very much prefer collaborative activities. Student-centric learning environments should be favored, where students actively participate in activities and group-based components that reinforce concepts and allow for exploration (Stewart et al., 2009)
Working software over comprehensive documentation	An iterative environment leads to higher immersion in the project, more learning, and better-quality deliverables (Stewart et al., 2009)
Customer collaboration over contract negotiation	Greater access to the instructor can lead to more collaborative relationship (Stewart et al., 2009)
Responding to change over following a plan	End-users work opportunistically, incrementally, and rarely one lifecycle phase at a time (Burnett & Myers, 2014). They exploratory mix programming, testing, and debugging, mostly by trial-and-error (Burnett & Myers, 2014). Agility is the ability to adapt to different learning styles and change the delivery methods (Stewart et al., 2009)

3.2 Why Extreme Programming?

Agility at its core refers to a dynamic approach that removes all the a priori burdensome planning and design phases and substitutes them with an iterative approach that tackles smaller scale problems. Over the years, different agile methodologies have put into practice the Agile manifesto (Dingsyr, Nerur, Balijepally, & Moe, 2012), and each of them addresses specific needs and requirements. Extreme Programming (XP) is one of the methodologies that has integrated practices concerning the project management as well as the development process, by focusing on continuous communication and excellent programming habits (Beck, 2000). The main characteristics of XP are

- short cycles;
- incremental development;
- flexible implementation;
- high automation;
- evolutionary design;
- extensive communication;
- collaborative development.

By exploring the link between end-users and Agile (see Sect. 3.1), we found that XP can be an excellent candidate to teach software engineering to end-users in K-12 for two main reasons: (i) it does not require end-users to change their development habits radically, and (ii) it helps end-users to structure their development by introducing a lightweight set of practices.

When it comes to the application of XP, it is hard to classify teams if they are XP based only on whether they apply XP practices to the full or not. About this topic,

Kent Beck argues that "the full value of XP will not come until all the practices are in place. Many of the practices can be adopted piecemeal, but their effects will be multiplied when they are in place together" (Beck, 2000). The primary value of XP is indeed to establish a set of principles and practices to guide the development process; nevertheless, depending on the specific context, some teams (or individuals) may find that certain practices do not apply to them. It is important, however, to systematically recognize when each of the provided practices can be omitted and if there are any dependencies between them. Section 4 details how a set of XP practices can be leveraged or adapted to the K-12 educational context, and Sect. 5 explains when each practice (or group of practices) can be applied to achieve the goals set by XP.

4 Mapping XP Practices to K-12 Practices: A Toolbox

In the following, we describe a toolbox, which contains the most relevant XP practices in our context, and we show how they can be mapped to end-users' working habits. Figure 1 shows how the working process is organized according to XP, and how each practice of the toolbox is mapped to the different phases of this iterative process.

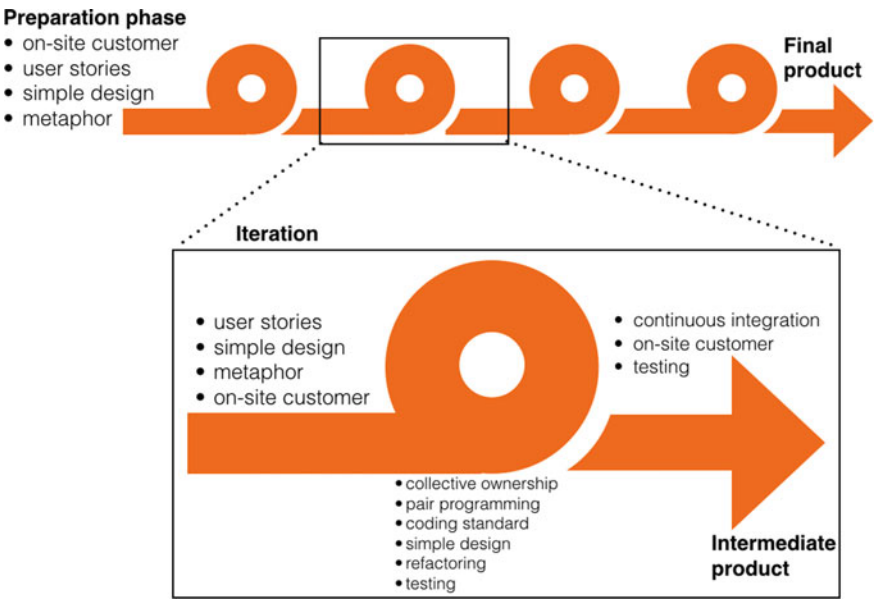

Fig. 1 Process description

4.1 User Stories

User stories represent the initial phase of the development process. The goal of user stories is to capture the requirements in a language that is understood both by developers and customers (Beck, 2000). They are written as a set of high-level scenarios that allow developers to understand the required features and the estimated effort. User stories also allow the classification of features, and the selection of those features that need to be implemented first and have a higher value to the customers. To this end, early, informal, paper prototypes are a very rapid way to generate ideas and obtain immediate feedback (Buxton, 2010; Traynor, 2012) to understand precisely how user stories will work and the client perspective. Moreover, paper prototypes are among the possible approaches (together with, for example, minimum viable product) to increase cost-effectiveness: it is indeed more cost-effective to make changes to a prototype than to an implemented user story (Fronza, El Ioini, & Corral, 2016c). For end-users, user stories can be mapped to the set of activities performed to understand and structure the problem domain, and to find an agreement on the requirements (Romeike & Göttel, 2012).

Images play an important role in human thinking, as they can capture visual and spatial information in a much more usable form than lengthy verbal descriptions (Thagard, 1996). In SE, rough, even hand-sketched, sequences of drawings (i.e., storyboards) are suggested to help understanding the customer's needs (Cardinal, 2013). In K-12 visuals are a common practice to support learners in: (i) linking new ideas to previous knowledge, (ii) connecting ideas, (iii) representing the structure of a product, and (iv) promoting collaboration (Walny, Carpendale, Riche, Venolia, & Fawcett, 2011). Therefore, user stories can be pretty easily applied in K-12, by simply leveraging tools already adopted in schools. The possible implementation of user stories in K-12 depends also on the type of system that is going to be created; for example, we illustrate storyboards, mind maps, execution trees, and role-playing.

4.1.1 Storyboard

A storyboard can be considered as the display of blocks of a comic strip, in which there is a visual representation of the sequence of an activity, which includes situations, actors, roles, and actions. In addition, a storyboard includes comments and annotations to provide a better notion of the action represented (van der Lelie, 2006). Regarding software development, storyboards can represent graphically a sequence diagram that uses the vocabulary provided by class diagrams, and it adds the chronological interaction between the different objects.

For example, when developing a mobile application in K-12, each panel of the storyboard can be considered as a *screen* of the app (Fig. 2a). Students are asked to draw each screen in a given format to create a mock-up prototype of the application's GUI. This requires defining the elements (i.e., figures, icons, text, background) of each screen and the actions that can be executed using those components (Fronza,

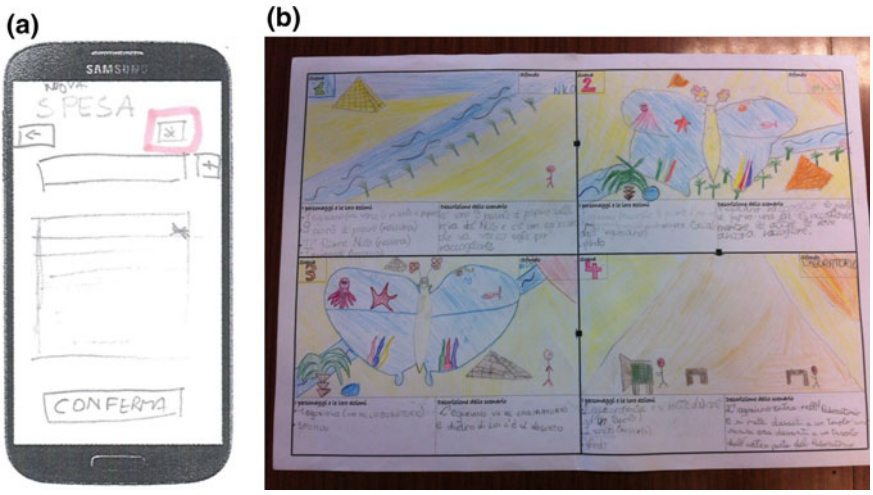

Fig. 2 Student-created storyboards in K-12

El Ioini, & Corral, 2015b, 2016a). Instead, when creating an animation, each panel of the storyboard represents a different scene (Fig. 2b).

Positive results of the usage of storyboards in middle schools are reported in Fronza et al. (2017); in this case, a simple template is suggested (Fig. 2b), which includes: scene number, type of background, characters in the scene and their actions, short description. In the context of elementary schools, simpler stories are suggested (Fronza et al., 2016c), with clear scenes and fewer concepts. Storyboards can be used at the end of each iteration to check if the goal of the iteration has been achieved, by comparing the storyboard with the implemented scene.

4.1.2 Execution Tree

The execution tree structures the execution flow in terms of a sequence. During the creation of a mobile app, for example, students can identify the transitions (i.e., edges of the tree) between the screens in the storyboard (i.e., nodes of the tree) (Fronza, El Ioini, & Corral, 2016b). To this end, students need to define what are the elements (e.g., buttons) and actions (e.g., tap) that trigger each transition. An example of the structure of an execution tree is shown in Fig. 3.

4.1.3 Mind Map

Mind maps serve to organize ideas within a project, to identify their relevance, and to describe the relationships between them. One of the advantages of mind maps is that they are regularly used in K-12. Therefore, they can be used without specific training and without changing the end-users' working habits.

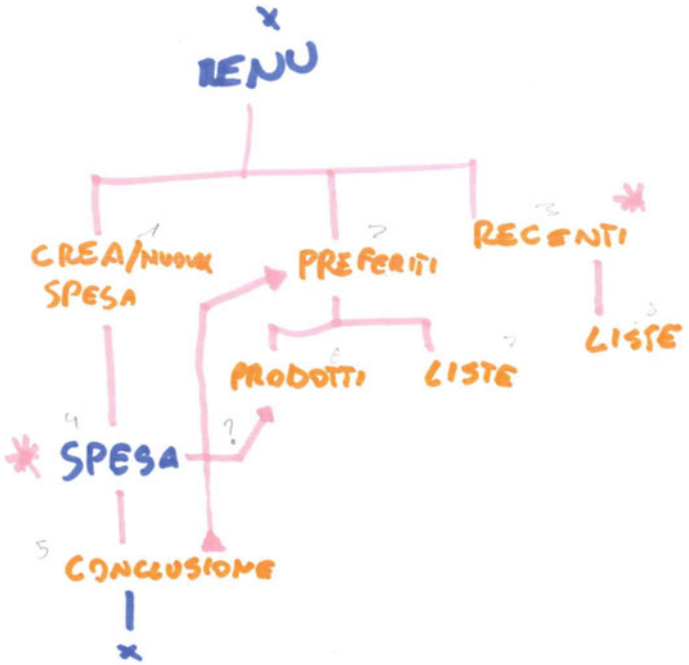

Fig. 3 A student-created execution tree

Mind maps can be thought of as class diagrams that define the set of classes and the interactions among them. Starting from the requirements document, students can build the necessary vocabulary that will be used by all the subsequent activities. Moreover, drawing a mind map requires identifying the most important concepts (i.e., classes) in the requirements document, and also to group congruent concepts, find their properties (i.e., attributes) and draw connections among them. This fosters the creation of an end-user's mindset that would be open to Object-Oriented programming. The benefits of this activity in supporting students in the identification of the software elements and their interactions have been shown both in elementary (Fronza et al., 2016c) and in middle schools (Fronza et al., 2017).

4.1.4 Role-Playing

Another possible implementation of user stories is role-playing, which supports requirements formulation from a user's perspective. In this activity, one student pretends to be the software and reacts accordingly, while another student takes the role of the user. The other students observe them in order to get details and constraints of the desired product. The process is repeated (with changing actors) until no more new requirements arise (Romeike & Göttel, 2012).

4.2 Small Releases

With the increasing demand on time to market products, software providers need to rapidly adapt to changes in order to maintain their competitive advantages. Small releases (a.k.a. short cycles) is the process of decomposing software development activities into short iterations allowing early and continuous feedback. For end-users, this is a fundamental requirement since it will enable them to work iteratively and keep adding more functionality to their core products. Additionally, in most cases, end-users use an exploratory approach in which they learn how to use the technologies and how to implement the required features. Thus, having short development cycles helps them evaluate their progress and adapt it accordingly. Moreover, this is particularly relevant for K-12, as it allows teachers to apply a formative assessment (Mikre, 2010).

A successful decomposition of the development process in small releases depends on the ability to estimate the required effort. This aspect might not be in the end-users' working habits. Therefore, they need to be supported in this process. One solution could be to link estimations to user stories with the disaggregation techniques (Cohn, 2005): the story's total estimate is calculated as the sum of the effort estimates of each task that is needed to complete it. Another possible approach is planning poker: Romeike and Göttel (2012) adopt professional card values with 15-min periods. Each student estimates the time to implement the user story in focus. Discussion of very divergent estimates is leveraged to clarify requirements and assumptions. After the planning poker is finished, the total estimated time is compared with the time available and user stories complexity is changed if needed.

What needs to be noted for these practices is that end-users might consider them as extra work that moves their attention from their goal. For this reason, it is important to let end-users perceive the usefulness of these practices, without forcing them too much to use them.

4.3 Metaphors

Metaphors as defined by Beck (2000), "a story that everyone customers, programmers, and managers can tell about how the system works", enable a shared vocabulary and common understanding on how the system functions. In the context of K-12, one of the early steps of the project can consist of establishing a shared vocabulary between teachers and the teams, especially when teams are not familiar enough with the application domain to be able to use native domain language.

4.4 Coding Standards

To increase productivity, developers are required to adhere to specific coding standards and conventions. This is particularly important for future modifications and refactoring. End-users tend to develop their own practices such as naming conventions and function compositions, however, in most cases they do not use any well-established reference when doing so. Using coding standards contributes to increased code quality (Fang, 2001), which is actually one of the EUSE goals. Therefore, end-users can benefit from learning this practice. For instance, teams can decide on naming conventions for variables and components. Another example concerns the testing process of the developed application: they can define a number of steps to follow and agree on adopting them.

4.5 Collective Ownership

In XP no one owns a specific part of the developed code, rather any developer can change any line of code to add functionality, fix bugs, improve designs or refactor (Beck, 2000). This allows for a constant code review and updates by all the members of the team. In K-12 students often prefer "repeatedly playing safe and choosing the role they feel most comfortable with, as opposed to stepping outside their comfort zone" (Stewart, 2014). Therefore, the risk is to have them working only on the piece of code they feel more comfortable with and then integrating the produced modules together. This might create situations in which no one sees the full picture of the project. Additionally, if team members are missing, it becomes difficult and time consuming to substitute them. Collective ownership in K-12 can be achieved by asking all team members to perform all the activities of the project; this means that no member is dedicated only to one task, rather they all need to be able to understand and modify the project tasks. Other XP practices (such as pair programming and coding standards) can support collective ownership.

4.6 Simple Design

The best design is the simplest that works. XP aims at finding the most straightforward solution that passes all the specified tests and meets the business values. One of the practices of XP is to write the tests first, which would fail initially, then write the minimum amount of code to make them pass (Beck, 2000).

K-12 teams tend to start by developing simple solutions for specific problems. However, when the scale of the problem increases, it becomes hard for them to manage. Two approaches are usually considered at this point: (i) overuse conditional statements to accommodate all the possible scenarios; (ii) develop different modules to tackle each scenario, then find a way to connect them to solve the bigger problem.

Good practices for simple design that K-12 can learn and apply are, for example, related to constructs, such as: making sure that the proper constructs are used (e.g., use loops instead of repeating a statement *n* times, use signals in Scratch[2] instead of timers), and use the minimum necessary constructs (classes, blocks, figures, etc.). To apply these practices, introductory exercises should be proposed with the goal of making sure that students learn the most relevant constructs for their activity (Fronza et al., 2017). Afterwards, code inspections at the end of each iteration can help in providing feedback to the students by, for example, suggesting the usage of a loop instead of repeated statements. Another good practice is to remove duplications from programs (e.g., use a function instead of implementing the same functionality in different parts of the code). In this case, high-level design (for example, using mind maps) can help students to identify those parts of the program that can be removed.

4.7 Refactoring

Refactoring is the process of improving code quality without changing its functionality (Beck, 2000). It consists of continuously reviewing the developed code and updating it (e.g., eliminate duplication, fix coding standards, naming conventions) manually or using dedicated tools. The main benefits of refactoring are to ease the maintainability and extendibility of the developed code. Refactoring can be triggered by the detection of code smells (Van Emden & Moonen, 2002) using automated tools, or it can take place at specific time intervals during the development process (e.g., end of each iteration). For K-12, refactoring is usually triggered by two factors: (i) adapting the existing code to support new features (e.g., changing functions parameters); (ii) substituting an existing construct with a new one (e.g., learning how to use messages instead of timers to move objects, setting the initial potion of objects in a Scratch scene). Refactoring as considered in XP gives K-12 students a more comprehensive perspective and pushes them to plan for future extensions and higher maintainability. For instance, while building the different modules, functional decomposition can help them to increase the maintainability of the produced code; at the same time, in terms of extendibility, any future changes can focus on single modules rather than having to touch different parts of the code.

4.8 Testing

Testing is the process of executing the system functionality with the intent of finding bugs and errors (Myers, Sandler, & Badgett, 2011). In a software system, every part of the produced code needs to be thoroughly tested before it is released. Depending on the development process, different types of testing can be performed (e.g.,

[2] https://scratch.mit.edu.

unit testing, integration testing, system testing). XP adds a new flavor to testing by introducing the concept of test first, which advocates writing tests as early as possible in the development lifecycle (Beck, 2000). This helps to have a well-structured testing framework and minimizes the testing effort in the long run. Acceptance tests is another important concept that allows developers to constantly having feedback from the customer. In K-12 context, teams tend to follow an opposite approach due to their inexperience, by continuously implementing and testing new functionality. In many cases, the nature of tools used by end-users forces them to follow this pattern. For instance, when using visual programming end-users need to implement and test each functionality to see if the added blocks work correctly (e.g., the right position, the right movement). User stories can be used as acceptance tests to validate the implemented functionality. For example, if each iteration's goal is the development of a panel in the storyboard, the comparison of the implemented software with the corresponding panel can indeed provide immediate evidence of the achievement of the iteration goal.

4.9 Pair Programming

In XP, code is developed in pairs: the *driver* writes the code, while the *navigator* reviews the code, thinks of the overall architecture, finds better solutions, and brainstorms (Beck, 2000). The central assumption is that having two developers work together will produce higher quality code, which reduces testing and debugging costs. Moreover, it has been shown that pair programming can improve software development under other perspectives, such as improving design, enhancing team communication, increasing job satisfaction, facilitating the integration of newcomers, and reducing training costs (Di Bella et al., 2013; Fronza & Succi, 2009).

In schools, it is very common to have two students sharing one computer due to limited hardware. This practice supports this need while adding a framework that encourages attention from both students, mutual learning, and a notion of programming as a social activity (Romeike & Göttel, 2012).

Pair programming can be used easily in K-12 as students prefer collaborative activities (Costabile et al., 2008). Nevertheless, some issues typical of the professional environment also apply in K-12. Therefore, teachers should carefully pick pairs by following some principles: for example, a very expert student in the pair might result in excluding the "weaker" student from the coding activities (Williams & Kessler, 2002).

4.10 Continuous Integration

The goal of continuous integration is to have at all times a working product. Once the functionality is implemented and tested, it is integrated with the core system. As we have seen in Sect. 3.1, in general, end-users work incrementally (Burnett & Myers,

2014). Asking K-12 teams to develop independent modules to integrate them at the end would not leverage this work habit. Moreover, in K-12 sometimes the nature of the used tools makes continuous integration the only possible solution; for instance, when using visual programming, users are forced to integrate the new features on top of the existing ones. When looking at XP practices, K-12 teams take advantage of the process of continuous integration by making sure that the new functionalities do not negatively affect or break the existing ones (Beck, 2000).

4.11 On-site Customer

For XP the customer is always present, in other words, the customer takes an active role in the development process (Beck, 2000). This allows quick and face-to-face feedbacks, which are at the heart of Agile methods. From user stories to acceptance tests, the customer helps developers in clarifying any doubt and assigns priorities to the different functionalities. In K-12, teachers can assume the customer role in order to guide the project progress. This helps the students to have a reference point in case of doubts or confusions. For instance, in line with the idea of the formative assessment, the teacher can help them when deciding the order of functionalities to implement or to validate an implemented feature.

5 Getting the Right Practice from the Toolbox: A Selection Strategy

In the previous section, we have described a *toolbox of XP practices* that K-12 teachers can use in their classrooms. The goal of this section is to support teachers in selecting the right practices from this toolbox, based on different criteria.

For example, a teacher can select "collective ownership of code" so that all the members of the team will have a full picture of the project. However, taking such practice in isolation could lead to chaos, because anybody can change the system without constraints (Baird, 2002). To address this issue, XP suggests *using practices in a concerted way*, so that the weakness of a particular practice is mitigated by the other balancing practices. In our example, pair programming, coding standards, and testing should all be selected by the teacher to balance collective ownership. These practices are part of the so-called *programming* area, which describes how actual coding is done. Another two areas are defined, with some overlapping practices between the three areas. The area of *process* describes how the development activities are organized, and includes on-site customer, testing, and small releases. The third area deals with *team* management and includes collective ownership, continuous integration, metaphor and coding standards, pair programming, and small releases.

However, XP is not just a mechanical assembling of practices. Instead, it is built on values and principles. Indeed, XP practices can be grouped together, reflecting how they relate back the principles of XP. Table 2 lists the XP practices and relates them to the underlying core principles (Baird, 2002). For example, testing, on-site customer, and pair programming all relate to fine scale feedback. These XP principles can find an application in the K-12 environment to apply formative assessment; in this case, teacher and learners need rapid feedback to modify teaching and learning activities based on the performed assessment.

To provide a concrete guide to choose the right set of practices to apply in a given project, we propose a *selection strategy* that uses the project goal and principles to support the decision making.

During the selection process, teachers need first (Stage 1 in Table 2) to choose the goal of the project, being one of the three above-mentioned areas: programming, process, or team. The project goal expresses what aspect of the project we are going to focus on. For instance, developing a project with programming as its target indicates that the goal is to make students learn programming practices, while a project with the team as its goal focuses on teaching students how to work as a team independently from the task at hand. Choosing the project goal activates a set of practices that can be used for each specific goal. The second selection criterion is the "principles to foster" (Stage 2 in Table 2). Each of the XP principles is mapped to a set of practices, and by choosing one or more principles, a new set of practices is activated. We note that teachers can also combine more options from each category (e.g., project goal: programming and process).

Once the selection is done, teachers need only to consider the set of activated practices that reflect the team type; in other words, if some of the activated practices can only be used by teams (i.e., those practices indicated in Table 2 with *), and the project will be undertaken individually, those practices need to be omitted.

Example 1 The teacher defines the project requirements as follows: The project goal is to learn the *process* of analyzing data coming from a set of sensors. During the project development, students will be working *individually*. The teacher wants to provide *fine scale feedbacks* and have a *shared understanding* between the students and the teacher (i.e., the customer) of all the project components. Table 3 shows the practices selected from the toolbox: in stage 1 and 2 all the practices corresponding to the project goal and principles are activated, but since the projects will be developed individually, the practices that can only be done in teams are omitted (i.e., those represented in Table 2 with *).

6 Conclusion

End-user programming has empowered millions of end-users to create their own software. End-User Software Engineering (EUSE) aims to increase the quality of end-user-created software by extending the benefits of a SE approach beyond the

Table 2 A selection strategy for practices

			Practices in the toolbox									
			1	2	3	4	5	6	7	8	9	10
Stage 1	Goals	Programming			✓		✓	✓	✓			
		Process	✓	✓					✓			✓
		Team		✓	✓	*				*	✓	
Stage 2	Principles	Fine scale feedback	✓	✓					✓	*	✓	✓
		Continuous process rather than batch		✓				✓			✓	
		Shared understanding			✓	*	✓					
1. User stories 2. Small releases 3. Metaphor and coding standard 4. Collective ownership 5. Simple design			6. Refactoring 7. Testing 8. Pair Programming 9. Continuous integration 10. On-site custome									

Legend ✓: practices for both teams and solo; *: practices only for teams

Table 3 Selection strategy applied to Example 1

Stage 1	Goal	Process: learn the process of analyzing data coming from a set of sensors
Stage 2	Principles	Fine scale feedback Shared understanding
	Team type	Individual development
	Selected practices	User stories Small releases Metaphor and coding standard Simple design Testing Continuous integration On-site customer

professional field. This chapter motivates the choice of XP as a good candidate for end-users in K-12 among the possible Agile methodologies. Moreover, the chapter shows that limited examples of Agile training in K-12 exist; most of these studies apply a limited number of XP practices in isolation, thus ignoring the dependencies between them.

The contribution of this chapter is twofold. First, it provides a description of each practice and reports examples and guidelines from existing studies. Second, the chapter proposes a XP-toolkit to encourage a synergistic use of XP practices that respects the dependencies between them. As an additional contribution, the chapter contributes to the End-User Software Engineering research field by encouraging further case studies with sets of practices, which could provide additional evidences on the positive effects of the toolkit in particular, and of bringing Agile inside the classroom in general.

References

Alégroth, E., Burden, H., Ericsson, M., Hammouda, I., Knauss, E., & Steghöfer, J.-P. (2015). Teaching scrum—What we did, what we will do and what impedes us. *Proceedings of XP, 212,* 361.

Astrachan, O., Duvall, R. C., & Wallingford, E. (2001). Bringing extreme programming to the classroom. In *XP Universe* (Vol. 2001).

Baird, S. (2002). *Extreme programming practices in action.* Pearson Education, Informit. Retrieved from http://www.informit.com/articles/article.aspx?p=30187.

Barkley, E. F., Cross, K. P., & Major, C. H. (2014). *Collaborative learning techniques: A handbook for college faculty*. Wiley.

Beck, K. (2000). *Extreme programming explained: Embrace change*. Addison-Wesley Professional.

Burnett, M. (2009). What is end-user software engineering and why does it matter? In *International Symposium on End User Development* (pp. 15–28).

Burnett, M. M., & Myers, B. A. (2014). Future of end-user software engineering: Beyond the silos. In *Proceedings of the on Future of Software Engineering* (pp. 201–211).

Buxton, B. (2010). *Sketching user experiences: Getting the design right and the right design*. Morgan Kaufmann.

Cardinal, M. (2013). *Executable specifications with scrum: A practical guide to agile requirements discovery*. Addison-Wesley.

Chimalakonda, S., & Nori, K. V. (2013). What makes it hard to teach software engineering to end users? Some directions from adaptive and personalized learning. In *2013 IEEE 26th Conference on Software Engineering Education and Training (CSEE&T)* (pp. 324–328).

Cohn, M. (2005). *Agile estimating and planning*. Pearson Education.

Costabile, M. F., Mussio, P., Parasiliti Provenza, L., & Piccinno, A. (2008). End users as unwitting software developers. In *Proceedings of the 4th International Workshop on End-User Software Engineering* (pp. 6–10). New York, NY, USA: ACM.

Di Bella, E., Fronza, I., Phaphoom, N., Sillitti, A., Succi, G., & Vlasenko, J. (2013). Pair programming and software defects—A large, industrial case study. *IEEE Transactions on Software Engineering*, *39*(7), 930–953.

Dingsyr, T., Nerur, S., Balijepally, V., & Moe, N. B. (2012). A decade of agile methodologies: Towards explaining agile software development. *Journal of Systems and Software*, *85*(6), 1213–1221 (Special Issue: Agile Development).

Fang, X. (2001). Using a coding standard to improve program quality. In *Proceedings of the Second Asia-Pacific Conference on Quality Software, 2001* (pp. 73–78).

Fronza, I., El Ioini, N., Janes, A., Sillitti, A., Succi, G., & Corral, L. (2014). If I had to vote on this laboratory, I would give nine: Introduction on computational thinking in the lower secondary school: Results of the experience. *Mondo Digitale, 13*(51), 757–765.

Fronza, I., El Ioini, N., & Corral, L. (2015a). Students want to create apps: Leveraging computational thinking to teach mobile software development. In *Proceedings of the 16th Annual Conference on Information Technology Education* (pp. 21–26).

Fronza, I., El Ioini, N., & Corral, L. (2015b). Students want to create apps: Leveraging computational thinking to teach mobile software development. In *Proceedings of the 16th Annual Conference on Information Technology Education* (pp. 21–26). New York, NY, USA: ACM.

Fronza, I., El Ioini, N., & Corral, L. (2016a). Blending mobile programming and liberal education in a social-economic high school. In *International Conference on Mobile Software Engineering and Systems, MOBILEsoft 2016* (pp. 123–126).

Fronza, I., El Ioini, N., & Corral, L. (2016b). Computational thinking through mobile programming. In *International Conference on Mobile Web and Information Systems* (pp. 67–80).

Fronza, I., El Ioini, N., & Corral, L. (2016c). Teaching software design engineering across the K-12 curriculum: Using visual thinking and computational thinking. In *Proceedings of the 17th Annual Conference on Information Technology Education* (pp. 97–101).

Fronza, I., Ioini, N. E., & Corral, L. (2017). Teaching computational thinking using agile software engineering methods: A framework for middle schools. *ACM Transactions on Computing Education (TOCE), 17*(4), 19.

Fronza, I., & Succi, G. (2009). Modeling spontaneous pair programming when new developers join a team. In *Proceedings of the 10th International Conference on Agile Processes and Extreme Programming in Software Engineering (XP2009), Pula, Italy, May 2009*.

Gross, P. A., Herstand, M. S., Hodges, J. W., & Kelleher, C. L. (2010). A code reuse interface for non-programmer middle school students. In *Proceedings of the 15th International Conference on Intelligent User Interfaces* (pp. 219–228).

Kastl, P., Kiesmüller, U., & Romeike, R. (2016). Starting out with projects: Experiences with agile software development in high schools. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education* (pp. 60–65). New York, NY, USA: ACM.

Krajcik, J. S., & Blumenfeld, P. C. (2006). *Project-based learning*.

Kropp, M., & Meier, A. (2013). Teaching agile software development at university level: Values, management, and craftsmanship. In *2013 IEEE 26th Conference on Software Engineering Education and Training (CSEE&T)* (pp. 179–188).

Lottero-Perdue, P. S., & Parry, E. A. (2017). Perspectives on failure in the classroom by elementary teachers new to teaching engineering. *Journal of Pre-College Engineering Education Research (J-PEER)*, *7*(1), 4.

Mahnic, V. (2012). A capstone course on agile software development using scrum. *IEEE Transactions on Education*, *55*(1), 99–106.

Meerbaum-Salant, O., & Hazzan, O. (2010). An agile constructionist mentoring methodology for software projects in the high school. *ACM Transactions on Computing Education, 9*(4), n4.

Mikre, F. (2010). The roles of assessment in curriculum practice and enhancement of learning. *Ethiopian Journal of Education and Sciences*, *5*(2).

Monteiro, I. T., de Castro Salgado, L. C., Mota, M. P., Sampaio, A. L., & de Souza, C. S. (2017). Signifying software engineering to computational thinking learners with AgentSheets and polifacets. *Journal of Visual Languages & Computing, 40,* 91–112.

Moye, J. J., Dugger, W. E., Jr., & Starkweather, K. N. (2014). "Learn by doing" research: Introduction. *Technology and Engineering Teacher, 74*(1), 24–27.

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing*. Wiley.

Paasivaara, M., Heikkilä, V., Lassenius, C., & Toivola, T. (2014). Teaching students scrum using lego blocks. In *Companion Proceedings of the 36th International Conference on Software Engineering* (pp. 382–391). New York, NY, USA: ACM.

Price, S., Rogers, Y., Scaife, M., Stanton, D., & Neale, H. (2003). Using tangibles to promote novel forms of playful learning. *Interacting with Computers, 15*(2), 169–185.

Romeike, R., & Göttel, T. (2012). Agile projects in high school computing education: Emphasizing a learners' perspective. In *Proceedings of the 7th Workshop in Primary and Secondary Computing Education* (pp. 48–57). New York, NY, USA: ACM.

Steghöfer, J.-P., Knauss, E., Alégroth, E., Hammouda, I., Burden, H., & Ericsson, M. (2016). Teaching agile: Addressing the conflict between project delivery and application of agile methods. In *Proceedings of the 38th International Conference on Software Engineering Companion* (pp. 303–312).

Stewart, G. (2014). *Promoting and managing effective collaborative group work.* Belfast Education and Library Board. Retrieved from http://www.belb.org.uk/Downloads/iepdpromotingandmanagingcollaborativegroupworkmay14.pdf.

Stewart, J. C., DeCusatis, C. S., Kidder, K., Massi, J. R., & Anne, K. M. (2009). Evaluating agile principles in active and cooperative learning. In *Proceedings of Student-Faculty Research Day, CSIS, Pace University*.

Thagard, P. (1996). Cognitive science.

Traynor, B. (2012). Rapid paper prototyping: 100 design sketches in 10 minutes, 18 designs presented, 6 prototypes tested, student engagement-priceless! In *2012 IEEE International Conference on Professional Communication Conference (IPCC)* (pp. 1–5).

Umarji, M., Pohl, M., Seaman, C., Koru, A. G., & Liu, H. (2008). Teaching software engineering to end-users. In *Proceedings of the 4th International Workshop on End-User Software Engineering* (pp. 40–42).

van der Lelie, C. (2006, April 01). The value of storyboards in the product design process. *Personal and Ubiquitous Computing*, *10*(2), 159–162.

Van Emden, E., & Moonen, L. (2002). Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings* (pp. 97–106).

Walny, J., Carpendale, S., Riche, N. H., Venolia, G., & Fawcett, P. (2011). Visual thinking in action: Visualizations as used on whiteboards. *IEEE Transactions on Visualization and Computer Graphics*, *17*(12), 2508–2517.

Williams, L., & Kessler, R. (2002). *Pair programming illuminated.* Addison-Wesley.

Ye, Y., & Fischer, G. (2007). Designing for participation in socio-technical software systems. *Universal Access in Human Computer Interaction. Coping with Diversity* (pp. 312–321). Longman Publishing Co., Inc.