Exploiting agile practices to teach Computational Thinking

Paolo Ciancarini[1,2], Marcello Missiroli[1], and Daniel Russo[3]

[1] DISI, University of Bologna, Italy
[2] Innopolis University, Russian Federation
[3] Dept. Computer Science, Aalborg University, Denmark

Abstract. Computational Thinking has been introduced as a fundamental skill to acquire, just like basic skills like reading, writing, and numeracy. The reason is that Computational Thinking is one of the most important skills for XXI century citizens, in particular for programmers and scientists at large. Currently, Computer Science teaching practices focus on individual programming and Computational Thinking first, and only later address students to work in teams. We study how Computational Thinking can be enhanced with social skills and teaming practices, aiming at training our students in Computational Thinking exploiting Agile values and practices. Based on prior studies, we describe and compare the four traditional software development learning approaches: solo programmer, pair programmers, self-organized teams, and directed teams. Such approaches have been explored in a number of teaching experiments, involving a significant number of students, over several years. Accordingly, we induced a model that we call Cooperative Thinking, based on such previous evidence and grounded in literature. This paper provides a research synthesis of previous works contextualized in a pedagogical framework, and proposes a new learning paradigm for software engineering education..

Keywords: Computer Education; Agile Methods; Computational Thinking; Meta–Analysis; Cooperative Thinking.

1 Introduction

Computational Thinking is a new form of literacy [62]. It is a concept that has enjoyed increasing popularity during the last decade, especially in the educational field. Computational Thinking is usually considered an individual skill, and practiced and trained as such [31, 63].

However, such an approach does not match current teaming structures of both science and business, where problems and projects have become so complex that a single individual cannot handle them within a reasonable time frame. To handle the increasing complexity, especially in engineering software systems, developers should be educated to act and operate as a team [17]. This is already happening in the business world. In fact, teaming is considered the key

driver to Digital Transformation, where solutions are not provided by individuals but by self-organizing teams [18]. Digital Transformation is often subject to"wicked problems", which do not have an unique solution but many Pareto-optimal ones [47]. This also applies to software development when complexity becomes very high [20]. Moreover, the DevOps technological trend needs specific approaches to support the training of developers/operators [5]

In Software Engineering, the role of the team and teamwork in general is especially crucial when Agile methods are used. The Agile principles acknowledge that important information and know-how might not be available at the beginning of a project [46]. Reaching the development goal requires several iterations, to build incremental solutions of increasing value for the users.

A key agile team–building factor is *self-organization*, meaning that any member of the developing team contributes with her knowledge, ability, and technical skills in order to work out a solution. Since each team member is responsible for the project as a whole, it is in everybody's interest to organize work at best – not bounce responsibilities. Moreover, teams are not static but they modify their structure according to necessities, which change over time. Not surprisingly, some organizations have built their competitive advantage and success on this model [1]. They comply with Conway's Law, according to which "organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations" [10]. A consequence of this observation is that organizations have to modify their *communication structures* accordingly to the problem which need to be solved. Therefore, flexible and self-organizing teams are best suited to comply with such pivotal evidence for any organization.

We argue that Agile principles and values should enrich the current efforts to establish Computational Thinking as a fundamental literacy ability. We call such a combination Cooperative Computational Thinking, or *Cooperative Thinking* for short. From a pedagogical perspective, it is grounded in Johnson & Johnson's Cooperative Learning approach, where students must work in groups to complete tasks collectively toward academic goals [28]. We suggest a team-oriented approach to educate software engineers in Computational Thinking. Educators should not just promote some good software engineering practices; rather, they should foster collaboration skills and train student teams to cooperate on wicked problems. Programming skills are usually considered personal ones; in most cases — job interviews, university exams, official certifications — the focus is always the performance of the *individual*. We lack a general approach to enable group skills in this context. Even if this idea may be widely shared by the community, we did not find any evidence of a comprehensive approach to it. This is probably due to the lack of explicit awareness of such concept as enabler of Digital Transformation processes: we may use it implicitly without recognizing it.

In this paper we analyze processes and interactions in four different learning modalities that mirror some standard software development models: solo programmer, pair programmers, self-organized teams, and directed teams. We report differences, practical and educational issues, their relative strengths with

respect to developing Computational Thinking skills on one hand and how they impact Agile team-related skills, that form the base of Cooperative Thinking, on the other.

As a result, we developed a model for Cooperative Thinking, contextualizing in relevant pedagogical theories. We provide results based on empirical and theoretical evidence; they can be applied to daily teaching practices.

This paper is organized as follows. Section 2 provides background information on related research on Computational Thinking and Agile education. In Section 3 we present the methodological framework used for this research synthesis. Section 4 presents the investigations we performed in teaching Cooperative Thinking comparing four modalities for organizing software development classes. Aggregated insights from our synthesis are presented in Section 5, where we propose actionable solution for educational practitioners. We discuss the synthesis of our research in Section 6, presenting the details of the extension of Computational Thinking with Agile practices, that we call *Cooperative Thinking*: self-organized teams are an effective way to enact and support Cooperative Thinking. Finally, in section 7, we summarize our vision, outline our future research, and draw our conclusions.

2 Related works

Computational Thinking has generated a lot of interest in the scientific community [62]. It is related to problem solving [44] and algorithms [33], because it is the ability of formulating a problem and expressing its solution process so that a human or a machine can effectively find a solution to the stated problem.

However, several scholars argue whether the Computational Thinking concept is too vague to have a real effect. For instance, a recent critique has been advanced by [15]. He claims that Computational Thinking is too vaguely defined and, most important in an educational context, its *evaluation* is very difficult to have practical effects. This same idea can be found in the CS Teaching community. [2] and [24] for example, try to decompose the Computational Thinking idea itself, in order to have an operative definition. [23] notes that computing education has been too slow moving from the computing programming model to a more general one. [4] even wonders if the Computational Thinking concept is at all useful in Computer Science, since it puts too much importance on abstracts ideas. It is also remarkable that there is some research trying to correlate CS and learning styles [25, 57], but generally inconclusive.

Though the Agile approach to software development is eventually going mainstream in the professional world, *teaching* the Agile methodology is still relatively uncommon, especially at the K-12 level. Moreover, a Waterfall-like development model is often the main development strategy taught in universities [35]. Moreso, it is usually limited to an introductory level and rarely tested firsthand. In practice, Agile is learned "on the field", often after attending *ad hoc* seminars. Interest in the field is however rising, and curricula are being updated to reflect this [55, 36]. An interesting and complete proposal has been proposed by [37].

The paper presents the "Agile Constructionist Mentoring Methodology" and its year-long implementation in high school; it considers all aspects of software development, with a strong pedagogical support.

To summarize, programming remains a difficult topic to learn and even to teach, both at university and high school level; the ability to design and develop software remains an individual skill and taught as such.

Some studies, however, tackle the idea that hard skills expertise should be complemented with soft skills, possibly introducing active and cooperative learning [30]. For example, in [48], a long list of so-called soft skills expertise is paired with various developer roles. In [8] the problem is well analyzed, but arguably the proposed solution is not comprehensive. [38] presents an example of how to promote cooperation within a software project; however generalizing the proposed scheme seems difficult. We note however that the approach is hardly systematic, and no general consensus exists on how to proceed along this line.

3 Research Methodology

Meta-analysis is a widely known and old research procedure, firstly methodologically supported by the work of [21]. The first meta-analysis was probably carried out by Andronicus of Rhodes in 60 BC, editing Aristotle's 250 year older manuscripts, concerning the work *The Metaphysics*. The prefix meta- was then used to design studies whose aim is to provide new insights by grouping, comparing, and analyzing previous contributions. Accordingly, we use the term meta-analysis to indicate an analysis of analyses. In this sense, there are a variety of analysis of analyses, like systematic literature reviews, systematic mapping studies, and research synthesis.

According to [12], a research synthesis can be defined as the conjunction of a particular set of literature review characteristics with a different focus and goal. Research synthesis aim to integrate empirical research in order to generalize findings. The first effort to systematize from a methodological perspective a research synthesis was performed by [11], building on the work of [27], proposing a multi-stage model. The stages are the following: (i) problem definition, (ii) collection of research evidence, (iii) evaluation of the correspondence between methods, (iv) analysis of individual studies, (v) interpretation of cumulative evidence, and (vi) presentation of results.

Following the multi-stage framework suggested in [11], we provide our problem definition, set as Research Question (RQ).

RQ: Is Computational Thinking scalable to teamwork?

To answer this question, we looked back to some previous works investigating the phenomenon on different perspectives.

All analyzed papers are both homogeneous and comparable, as depicted in Table 1.

We both provided insights on single papers in Section 4, and provide an interpretation of cumulative evidence in Section 5.

Table 1. Investigation list

Title	Focus of Experiment	# Subjects	Ref
Learning Agile software development in high school: an investigation	Pair Programming, Timeboxing, User Stories, Team Development	84	[39]
Teaching Test-First Programming: assessment and solutions	Pair Programming, Social dynamics	102	[41]
Agile for Millennials: a comparative study	User Stories, Scrum, Waterfall, Team Development, Timeboxing	160	[40]

As an outcome we propose a new educational framework, namely Cooperative Thinking, which we use to answer to our research question in Section 6.

4 Results

We performed some experiments collecting several insights regarding teaming for solving computational problems, as listed in Table 1.

For the purpose of this research synthesis, we abstracted empirical knowledge and mapped learning models to learner types. To do so, we used the well–known Kolb's learning style inventory [34, 32], consisting in:

- Individual learning (best suited to Assimilators)
- Paired learning (best suited to Convergers)
- Directed group learning (best suited to Divergers)
- Self-determined group learning (best suited to Accommodators)

This classification supports our inductive epistemological approach, allowing us to contextualize already collected evidence into a broader theoretical framework. Hereafter, we make our considerations for the four learner groups.

4.1 Individual learning

Directed Individual learning (short: Individual Learning) corresponds to the most common form of teaching, practiced everywhere in practically every subject and most often associated to Directed Instruction.The typical form consists of a lecture on a new topic, followed by individual study and exercises, then finalized in some kind of assessment (test and/or capstone project); teaching is generally sequential, each concept built on previously acquired knowledge. The main advantages of this model are its simplicity and efficiency; a single individual can teach a full class of people at the same time; moreover, we all already have have plentiful experience with this method. More recently, by using modern

technology this model can scale almost indefinitely. Practical examples include the Khan Academy, Udacity, and other MOOCs. An interesting aspect is that the sequential progression is ideal for stimulating Computational Thinking concepts — especially Problem Solving. Once a topic is mastered, it can be used to tackle more complex concepts or deepen and reinforce the significance of an acquired one. Another advantage is assessment; for instance, it is very easy to evaluate a program written by a student thanks to standard testing frameworks, to the point that automatic evaluation is becoming more and more common — a decisive point in case of e-learning on very large classes.

This model is dominant, however it has several limitations. One of most important ones is the fairness of the assessment. The difficulty of the assessment test is usually tailored upon the average student, resulting in a Gaussian curve grade distribution. In this model, students falling behind at the beginning of the course rarely have the capacity to catch up, as the time allotted is the same for every student; additional information, requirements, time–demanding exercises pile up very quickly. People experiencing learning difficulties have very few options. Those who can afford it resort to privately paid tuition, but for the rest the road a failing grade is almost certain. A consequence is the so called "Ability Myth": it states that each of us is born with a set of abilities that hardly change during our lifetime [56]; in fact, this phenomenon is in most cases the effect of accumulated advantages [54].

Another drawback is the absence of positive social interaction. Direct teacher/student communication is constrained by the available time. Student/student interaction rarely includes exchanges of ideas or effective cooperation; more often than not, it results in direct competition or in nonconstructive and illegal help (i.e. cheating). All of this might have a negative influence on overall motivation, especially in less-than-average performing students.

In experiments [39, 40], we simulated a working day in a software house; the teacher assumed the role of the software house boss, and selected a number of students who were previously categorized as either "good", "average" or "poor" programmers. Each student was given a moderately difficult task using a new work methodology (either TFD or User stories) within a limited time-frame. Without much surprise, both performance and the perceived utility of the activity mirrored their current skill level.

Individual learning help foster Computational Thinking but it is not useful (or maybe detrimental) to develop social skills needed for Cooperative Thinking. According to Kolb's learning inventory [34], this teaching model best suits *Assimiliative* learners, since they like organized and structured understanding and respect the knowledge of experts. They also prefer to work alone.

4.2 Paired learning

Paired learning (also called *Dyadic Cooperative Learning Strategy* [53]) is also a common technique but far less popular than the previous one. The basic principle involves the teacher posing a question or presenting a problem, then the students

discuss in pairs and find their own way toward the solution; pair members are often switched, sometimes even during the activity.

The role of the teacher in this case is quite limited, as she acts as a general coordinator and facilitator of the class of pairs. In the software development field, we find an obvious transposition of this model in Pair Programming, one of the key Agile programming practices and, to a lesser extent, in some training techniques (Randori and Code Retreats among others [50]).

According to [14], this model has beneficial influence on retention, understanding, recall, and elaborate skills at the cognitive level; it is particularly effective on mood and social skills, and introduces the idea of software being an iterative, evolutionary process. As it promotes knowledge sharing, it can help less skilled individuals to improve themselves taking inspiration from their partners. However, it is more difficult to develop a teaching progression using only this model, and in any event, it would be rather slow. In addition, psychological and personal factors become important, since partner incompatibilities and social difficulties might dramatically change both the learning outcomes and the quality of the code produced. Assessment is more difficult than in the previous case; though automatic evaluation is still possible, some extra steps are required by the teacher to deduce the effective contribution of each member of the pair to the final work.

We tested firsthand this effect in experiments [39, 41]. We proposed the same method and problems stated in Sect. 4.1, but in this case we paired students according to six possible pair types, classified as homogeneous (good-good, average-average, poor-poor) or as non-homogeneous (good-average, average-poor, good-poor).

According to results, homogeneous pairs performed generally equal or worse than their solo counterparts, but non-homogeneous pairs had statistically better results. In the latter case a form of epistemic curiosity [29] appeared, possibly unconsciously, and was a key motivating factor for the pair; the resulting interaction helped both to solve the task at hand and to develop social skills. Computational Thinking was also stimulated, but a little less than with the previous model, since the "effort" was split and each single task was not really challenging, requiring expertise more than logical reasoning.

In addition, both students and teachers praised the new methodology and its positive effect on mood. However, the retention rate was very low, much worse than expected; in an interview conducted some time after the experiment was over, students generally only had a vague idea of the techniques used and only about 5% of them was able to name them correctly.

To summarize, paired learning has beneficial effects on social skills related to Agile development, and generally is useful in leveling skills upwards. Knowledge building will however be much slower than in the traditional approach. This teaching model better suits *Convergent* learner types, since they want to learn by understanding how things work in practice, like practical activities and seek to make things efficient by making small and careful changes.

4.3 Directed group learning

Group learning is one of the many facets of Cooperative Learning, which is becoming fairly common in modern, constructivist-influenced education [6]. It is also a common practice in some working environments, notably in the health context for nurses [61]. Group learning in a software engineering lab class is best exploited by developing a full software project, not simple exercises or abstract analyses. So, it is natural to join Group Learning and Project-Based Learning strategies, especially using the Jonassen variant [26]: a complex task taken from real-life with authentic evaluation, comprehensive of all phases of development.

We are aware that many software development methodologies exist, and each of them can be transposed in an educational context promoting different behaviors and skills. One of them is the Waterfall model, probably the oldest one but still quite popular in the industry.

Waterfall embodies in many ways all the tenets of our prevailing culture, such as linear hierarchies, top-down decision making, accepting the assumptions, acquire all information in order to prepare a detailed plan and then following it — values that have forged the way traditional education was conceived and in most cases is still carried out.

A Waterfall school project will see the teacher assume the role similar to that of a senior project leader, assigning tasks and roles to students according to their skill, knowledge, and ability and applying a certain degree of control. The teacher's role will be very important at the beginning of the project, as students generally lack the ability to perform a thorough analysis and comprehensive design phase. As the project continues its course, the role will be more oriented to control, checking that documents are properly written, modules developed and tested, directing the flow of the entire operation. Assessing a group project is considerably more complex that both previous model, since it involves not only the final product, but also the process used and the interaction among the student and their relative contributions. To resolve it, usually a combination of traditional evaluation (automated or not), direct teacher observation and peer evaluation is used, forcing students to evaluate and reflect on the quality of their work.

In a different experiment, we decided to give students a very challenging task, almost impossible to solve. They had to build from scratch a complete dynamic website, a task we estimated in about 30 man-hours to complete when handled by experts. We only gave them 6 hours. This forced teams to make hard decision as to what was the most suitable course of action in order to make the best use of the allotted time and resources.

Then some extra restraints were imposed on the group, such as:

- A rich set of artifacts, such as a complete SRS, ER-diagrams, management priorities, UI-Mockups.
- Specific roles (programmer, UI-expert, tester, ...) and hierarchies (chief programmer, for example) were imposed.
- A predefined time schedule.

From an educational viewpoint, the target product was definitely outside a single student's zone of proxymal development [59], but was theoretically doable as a team effort. From a different viewpoint, such a target looks like a wicked problem, since students lacked the knowledge and the competence to complete the task, and were requested to acquire them along the way [60]. The great amount of information and in general the directive role of the teacher gives the opportunity to highlight whatever learning goal is deemed important.

Results show that, under these conditions, groups tend to concentrate on non-functional requirements and process-related goals instead of pursuing the main goal: delivering a working product to the "customer". The products, on average, had very few working features, but the defects were hidden under a pleasant user interface, close to the one proposed by the "management". Roles were interpreted rather closely to the given instructions (barring a few cases of internal dissent), timing was impeccable, and even the documentation was acceptable. Teacher-student interactions were not intense, but rather limited to simple yes-no questions. Students reported great satisfaction for both the activity and the product realized, asserting it was an activity both useful and fun [40].

To summarize, this teaching model promotes the use of social skills, while leaving the steering wheel in the hand of the teacher. This power can be used to provide a meaningful learning path, though slower that Individual Learning and with a non-trivial evaluation method. It also does not seem to stimulate enough other interesting skills, such as decision making. It better suits *Divergent* learner types, since they and will start from detail to logically work up to the big picture. They like working with others but prefer things to remain calm.

4.4 Self-directed group learning

This model is a different version of Group Learning, radically different than the previous one in that students have a strong degree of autonomy. It applies to K-12, adult education and business/industrial environments, for example [22].

In this case, the teacher becomes a mentor and a facilitator, and invests a large amount of trust on the learners.

Most of what we said on Project-Based Learning in the previous subsection holds. In this case, the granted freedom can be a powerful weapon in the hands of the group, but it might also backfire.

It is easy to see that several Agile values are connected to this learning model: most prominently, shared responsibility and courage. Agile strongly promotes an adaptive approach to software development, where each iteration acts as a feedback for the next one. Teams should be self-organized, and great emphasis is put on communication, both within the team and with the stakeholders. This means that the teacher must *become part of the team* in order to maintain a high level of communication. It also means that the teacher cannot distribute grades in a standard way, as he will be directly involved in the process (effectively becoming a 'pig', and not a 'chicken', referring to the classic Agile metaphor). Grades should therefore come from reflections, group and/or personal and peer

evaluation, and must include an evaluation of teacher work, as any other team member.

In experiment, we kept the same general structure outlined in section 4.3, but within the same class we assigned the same project to a different, potentially equivalent, team. This allowed for a direct comparison of results, since it ruled out biases due to different teachers, learning environments, or curricula. We have chosen the Scrum methodology, because it is arguably very different from Waterfall and it does not really mandate any practices, giving maximum freedom to the teams [52]. The teacher assumed the role of the Product Owner in this specific case; alternatively, the Scrum Master role could be chosen as well [40].

The teams were given much less information and limitations with respect to Waterfall teams:

- A list of prioritized user stories.
- A 'definition of done' (as in Scrum): it is a definition of how a result can be considered to have some value, in terms of simple activities like writing code in a standard format, adding comments, performing unit testing, etc.
- The sprint length.

Everything else was to be decided by the team. Scrum teams also had the additional difficulty of having no experience with self-organization, whereas traditional Waterfall methodologies and roles were taught as part of standard curriculum.

Results show that Agile teams performed generally better than their Waterfall counterpart in the same class with respect to overall product completion and number of featured delivered. This is not surprising, since Agile privileges the functional dimension over the non-functional ones. It is interesting to note that many chose challenging but interesting tasks, possibly failing along the way. However, with respect to code quality, Agile teams fared worse than their counterparts. First, code was less readable and with worse Cyclomatic Complexity evaluation; second, the final product on average had severe usability problems, since this was not an explicitly stated goal. In general, teams underestimated the effort needed on the first sprint but guessed much better their second sprint, during which they were much more productive. Teacher-student interaction was also not very intense – suddenly cooperating at peer level with an older, experienced superior is not an easy task for anyone. Students reported great satisfaction for this activity, slightly more than for the previous model.

So, both types of Group learning (directed like Waterfall and self-directed like Scrum) missed the main point of the activity, which was to provide a valuable product for the customer. What is interesting is the motivation for such failures. Scrum teams concentrated their effort to reach a goal, possibly a difficult one, displaying Courage, a key XP value. Waterfall groups tended to "play safe", and concentrated on less risky objectives (user interface, process oriented goals) and working on what they most comfortable with, a pattern more in line with logical reasoning.

The self-directed group model strongly promotes the use of social skills and other qualities relevant to Cooperative Thinking. However, the learning rate

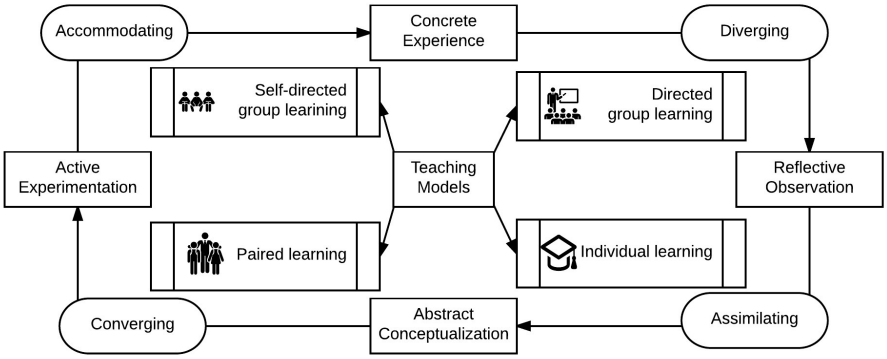

Fig. 1. Teaching activities mapped to learner types, following the taxonomy of [34]

could be exceedingly slow; moreover, evaluation requires great attention and balance. It better suits *Accomodative* learner types, since they display a strong preference for doing rather than thinking. They do not like routine tasks and will take creative risks to see what happens.

5 Implications for practice

Kolb's model identifies four basic types of learning experiences (Active Experimentation, Concrete Experience, Reflective Observation) and four basic types of learners (Converging, Accommodating, Diverging, Assimilating). Kolb suggests to alternate these learning modalities in order to stimulate different aspects of the learners' mind, even if an individual is more oriented to a specific king of learning activity. We therefore classified four types of learning experiences specifically related to lab classes that can be appealing to a particular learner type, as shown in Fig. 1.

Table 2 summarizes the content of this section. Traditional teaching concentrates on individual learning, thus favoring Assimilating students; we argue that a more balanced approach is beneficial in general, and in particular can stimulate and develop focused social skills that are essential for developing an effective Cooperative Thinker.

Table 2. Learning model influence on learner and teacher's role

	Teacher Role	**Learning Path**	**Computational Thinking**	**Social skills**	**Agile skills**	**Ease of Evaluation**	**Preferred Kolb Learner Type**
Individual Learning	Boss	+++	++	–	–	++	Assimilator
Paired Learning	Facilitator	-	+	+	++	+	Convergent
Directed group learning	Project Leader	+	=	++	+	-	Divergent
Self-directed group learning	Teammate	-	=	+++	+++	–	Accommodator

We understand that Kolb's classification is crude, as it cannot capture the complexity of teaching and learning in a social environment, be it at school or on the workplace; yet, even this simple model is powerful enough to analyze the situation and plan activities to reach our goals.

Cooperative Thinking is a general theoretical concept, just like Computational Thinking. Educators should do their best in order to have students understand and be able to put theory into practice. Is the educational system able to accept this change? Our discussion concentrates on teaching software development lab classes.

Usually, only individual performances are evaluated in lab classes of both high schools and colleges alike: it is less common to evaluate the teamwork. We will now describe some teaching models that can be used to promote the emergence of the two pillars of Cooperative Thinking: Computational Thinking and Agile practices. We have evaluated the impact on students designing and performing a series of learning experiments that exposed software development students to Agile practices and values.

In this article we analyzed a series of teaching strategies for software development, each with advantages and disadvantages and having a different impact on cognitive, reasoning and social skills that collectively concur to create what we called Cooperative Thinking.

Traditionally, education has considered literacy and knowledge in a broad sense. Consequently, the quality of education is often tied to fundamental skill expertise; one of the most recognized indicator is the result of the international PISA test, that evaluates how effective a country has been at deploying their prescribed math, science, and reading curricula. In this perspective, it makes perfect sense for educational institutions worldwide (and universities foremost) to favor individual learning as the primary – if not only – teaching strategy. For instance, a consequence of this is that several efforts are spent in schools on overcoming individual differences among students: see for instance the well known discussion of the "Matthew effect" in [54], which is a social selection process resulting in a concentration of resources and talent.

However, in the future, "pure" knowledge might become less important, even to the point of becoming a commodity, and soft skills could raise in importance. An educational system focusing on hard, technical skills could have difficulties in promoting soft skills. As [64] pointed out, there is an inverse correlation between PISA test scores and entrepreneurial capacity, a measured by the Global Entrepreneurship Monitor (GEM), the world's largest entrepreneurship study. Specifically, the countries with the top PISA scores had an average GEM:PISA ratio of less than half of the mid- and low-scoring countries, indicating a potential shortfall in PISA's measuring purpose to understand if students are "well-prepared to participate in society" [42]. And this might as well be true in Computer Science.

Notably, the ability to solve complex issues or *wicked problems*, is a requirement for new product development and innovation & entrepreneurship in general [7]. Wicked problems usually have no single perfect solution but many Pareto-

optimal solutions. The traditional educational paradigm is not tailored to train people able to handle similar situations; PISA-like evaluations are meaningless to determine the educational system's efficiency, since the only offers an evaluation of the *individual*.

So, the gap between a formal educational background and real-life wicked and complex problems becomes larger. Actually, it will increase along with Digital Transformation processes, where the level of predictability decreases and uncertainties increases [45].

Therefore, the introduction of other teaching strategies that foster social skills and cooperation is very important, and should also be factored in grading activities. Note that we do not advocate a complete suppression of the Individual Learning strategy; on the contrary, it should be *complemented* with other strategies in order to obtain an overall balanced and blended mix tailored to specific situations – there is no silver bullet in education. This proposal will also have the extra bonus of potentially appeal to all learner types, even those that traditionally are less inclined to pick Computer Science as their course of study.

Given all the above considerations, we recommend all strategies we mentioned be used in teaching software development, in order to promote different but equally important skills and possibly favoring different learning styles. This strategy mix should begin as soon as possible and continue throughout the entire study path, up to and including the university tier. Otherwise, it might be too late to develop the full potential of Agile-related skills and, consequently, Cooperative Thinking.

5.1 Learning path

Most CS courses are strongly oriented toward individual learning, the goal being to introduce and grasp the basic elements of CS and, specifically, programming [49]; a short to medium-length programming project of average difficulty is usually included.

As soon as possible, Pair Learning should also be presented. Specifically, Pair Programming should be introduced first and actively enforced as one of the main practices for class exercises throughout the course. Other Agile practices could be introduced (such as Test-First Development, Continuous integration, ...) along with the necessary software tools (like git or Jira). A project that verifies what students learned should be simple in terms of programming complexity but rich in process experiences, in that elements of Agile must be used and their use verified.

Next, forming the team is an important factor. We know that simply putting together people and telling them to work on a project is not enough to have an even decently efficient group. Preparation is in order, requiring some careful people selection, team-building exercises, and some short project to test how the teams work. Finally, a team-oriented project of moderate to high difficulty and length should be realized by the students.

The final step is, of course, proposing a demanding project to the teams and give them ample freedom. At this point students should have a solid knowledge

of the programming language and development methods, a grasp of basic Agile practices, and some working experience with all necessary tools; moreover teams should know their strengths and weaknesses. This activity can actually be a course capstone project and should contribute significantly to the students' grade.

Our proposal requires formalization, testing, and formal validation. Though every step is nothing new or complicated, the overall teach process is. Our research group is currently working on a comprehensive proposal and its field testing in both K-12 and university classes.

5.2 The influence of the context

We discuss now the validity of this study in the different contexts of High School and University classes.

First, we examine some distinctive features of learning in high school:

- The learning activities encompass several years. During this long time period, teachers and learners get to know well each other and develop a relationship that has strong effect on the quality of their cooperation.
- The evaluation of the students is based on several factors. One is certainly the overall performance (tests, lab results), but many other aspects are factored in: initial level, handicaps, effort, proper behavior. This implies that the teacher must exert some form of control and surveillance, even due to age considerations.
- Learning goals tend to be broad-scoped, leaving advanced topics only to the best students.

The University learning context *seems* to be completely different. Instructors usually teach for a single semester, a time insufficient to establish a personal relationship. Performance evaluation is far more important, overshadowing other factors; standardized tests and procedures are used, focused on both general and specific topics. Higher levels of personal responsibility and self-organization are expected, so teacher control is generally limited.

However, in the specific case at hand, differences are not so well marked. We performed our experiments in high school courses (total: about 250 students) which are programming intensive, featuring around nine programming hours - labs included - per week for three full years. They cover basic and intermediate programming issues, including dynamic data structures, recursion, and databases for an average of 300 programming class hours per year, personal study not included. While we do not claim that this kind of education to software development is equivalent to a standard undergraduate level lab class in software development, it is undoubtedly comparable, on average compensating subject depth and personal motivation with more time spent in practical experiences. Our experiments on undergraduate students (total: about 90 students) confirm these impressions.

Not surprisingly, we found that our teaching strategies had to be adapted to the different educative levels. For example, students in high schools require

learning activities on Agile to be repeated and, at least partially, integrated into standard teaching activities. Failing to do so inexorably results in limited long-term retention, as some interviews sadly demonstrated. Moreover, students must concentrate on Agile practices rather than on the overall development process; they are only able to handle a software project of limited scope and complexity, so setting up a full-fledged development environment (be it Agile or else) looks like an overkill.

Conversely, undergraduates are able to make the most out of one-shot activities; they are expected to reinforce their knowledge and skills with personal work, and most of them indeed do. They have sufficient capabilities and time to properly apply a standard Agile development cycle, especially in capstone projects. The problem in this case is the large amount of topics to cover: the instructor has the responsibility to select the topics that must be taught. In addition, undergraduates have a higher degree of freedom, so they cannot be forced to adopt a given method or practice. The effective use of Agile by students depends on their personal and, for some part, on the charisma of the instructors.

6 Discussion

In 2006, Jeannette Wing's paper defined and popularized the concept of Computational Thinking [62], portrayed as a fundamental skill in *all* fields, not only in Computer Science. It is a way to approach complex problems, breaking them down in smaller problems (decomposition), taking into account how similar problems have been solved (pattern recognition), ignoring irrelevant information (abstraction), and producing a general, deterministic solution (algorithm).

Even after more than a decade, the impact of this idea is strong. Eventually, some governments realized that future citizens should be *creators* in the digital economy, not just consumers, and also become *active citizens* in a technology-driven world.

Computational Thinking needs to be properly learned and, therefore, is being inserted as a fundamental topic in school programs worldwide. This is a welcomed change away from old educational policies that equated computer literacy in schools to the ability of using productivity tools for word processing, presenting slide shows, rote learning of basic concepts. Though useful in the past, they are currently outdated and even possibly harmful. The US initiatives *"21st Century Skills"* [58] and curriculum redefinition, along with *"Europe's Key Skills for Lifelong Learning"* [19] should be viewed in this perspective.

However, these approaches might not be sufficient in the long run. Current educational approaches concentrate on coding (as an example, consider the *Hour of Coding* initiative), but this it not the end to it. Computational Thinking is made of complex, tacit knowledge, that overcomes limited resources and requires deep engagement, lots of deliberate practice, and expert guidance. Coding is one aspect, and not necessarily the most important one.

Tasks solved by software systems are becoming more complex by the day, and many of these in the real world could be classified as *wicked problems* [47]. There

is no single "best solution" to many such problems, only Pareto-optimal ones which may change over time. In this situation, satisfying expectations and requirements becomes harder and harder as they are beyond the limit of solvability for any single programmer.

This is well known in the fields of Science and Business. The most common approach to trying to solve wicked problems in these fields is by forming teams including people with complementary backgrounds, trained to face problems and reach the goal – together. These new cooperative entities benefit from a high degree of independence and autonomy to deal with the assigned task; the idea is to solve a problem attacking it from different points of view.

Even if Computational Thinking has been defined as a problem-solving skill, and has benn taken as the basis for several ongoing activities, by itself alone it does not offer the variety of viewpoints required to solve difficult or wicked problems. Computational Thinking has traditionally been considered an individual skill, and taught as such. Teamwork and soft skills are generally not factored in, and even shunned as "cheating" in some introductory programming courses.

In our view, the general approach to Computational Thinking needs to be updated, by enhancing it with a complementary concept: Agile values and practices. The Agile Manifesto was published in 2001, just a few years before Wing's paper. In just 68 words, it proposed a quite original perspective on software development, recalling values that clashed with the established culture of time, based on top-down hierarchies, linear decision making and, in general, pursuing unsustainable management plans. The most significant change introduced by the Agile movement is the paramount importance assigned to face-to-face communication and social interaction, superseding the internal organizational rigidity, documentation, contracts, roles, and more [46].

Including some Agile principles and learning-as-execute experiences in training for Computational Thinking is beneficial. We name *Cooperative Thinking* this Agile extension of Computational Thinking, and define it as follows:

"Cooperative Thinking is the ability to
describe, recognize, decompose, and computationally solve problems
teaming in a socially sustainable way"

This definition joins the basic values of both Computational Thinking and the Agile Manifesto.

Computational Thinking is based on the power of abstraction, problem recognition and decomposition, and algorithms. Agile principles include self-organizing teams, interaction and communication, and shared responsibility. Both Computational Thinking and Agile value the concepts of evolution and reflection of problems and solutions. Both approaches share the idea of problem solving by incremental practices based on learning by trial and error. Moreover, our definition of Cooperative Thinking underlines *sustainability*, since "solutions" as such have little impact, if not related to the available resources.

In sum, Computational Thinking is *the* individual skill to solve problems in an effective way. We found that Agile values are central not only for developers

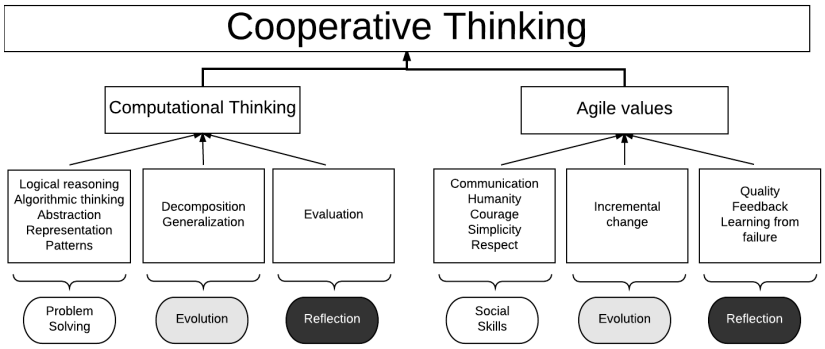

Fig. 2. Cooperative Thinking, Computational Thinking and Agile values breakdown (according to *Computing at School* [13] and [3])

but also for educating individuals. Cooperative Thinking adds a variety of points of view required to solve really demanding and complex tasks, like for instance developing critical systems [9, 43, 51]. Enhancing Computational Thinking with Agile values and principles allows to exploit the power of a team of diverse backgrounds towards a common goal. Being mentally flexible, understanding the others' points of view and synthesizing a common solution are crucial skills for teaming developers.

7 Conclusions

In this paper we explored Cooperative Thinking, a concept that expands Computational Thinking embracing Agile values. The proposal is graphically summarized in Fig. 2.

Cooperative Thinking is the extension of Computational Thinking with Agile Values. We considered the skill breakdown proposed for Computational Thinking by *Computing at School* [13] and grouped the skills into three broad categories: Problem solving, Evolution, and Reflection. Correspondingly, we considered Kent Beck's XP values and practices list [3] as representative of Agile values and practices in general; list items were also grouped in three categories: Social Skills, Evolution, and Reflection.

Cooperative Thinking is a complex skill to acquire and master, but in our view, is the way to go to obtain teaming individuals able to tackle and resolve the challenges and questions that the future will present them.

We examined four different learning models, each with a different balance of traditional, Agile, and Cooperative learning, showing the impact they had on students in developing Cooperative Thinking. Specifically, Individual learning is strongly related to Problem Solving, Social Skills to Self-directed group learning; all other aspects have a varying degree of relationship to the different models.

Experiments showed a significant effect on the learning outcomes. Cooperative Thinkers will enjoy an edge on the job marketplace, making them more flexible, socially aware, and more able to handle future challenges, be they related to software development or not.

In order to educate students to Cooperative Thinking, we suggest that a mix of learning strategies be used, in order to expose students to Agile practices and values and develop teaming skills without forgetting basic Computational Thinking skills, such as abstraction. While we do not claim the superiority of Agile practices as such, we do observe their effectiveness as *enablers* of Cooperative Thinking, since they promote interaction, force efficient resource handling, and are strongly goal-oriented, substantially more than individual learning.

We propose to define and evaluate innovative educational programs promoting Cooperative Thinking. Mixed methods assessments for educational construct validation with Structural Equation Modeling as also fine granular performance indicator for Pareto-optimal solutions need to be validated. However, finding the exact blend of teaching strategies will be the real challenge for the software engineering community; this is exactly what we are investigating now, both at K-12 and undergraduate level.

Another line of research that we intend to pursue concerns the constructs which constitute Cooperative Thinking, especially concerning teaming [16]. For instance, the dynamic structure of teams is interesting: we have seen in the experiments that in pair programming asymmetry of competences is quite effective. In teams including more people, say four or five students, we intend to study the emergence of mentors as facilitators rather than leaders, and the impact of such figures on self-organization of teams.

References

1. Amabile, T., Fisher, C., Pillemer, J.: Ideo' s culture of helping. Harvard Business Review **92**(1-2), 54–61 (2014)
2. Barr, V., Stephenson, C.: Bringing Computational Thinking to K-12: what is involved and what is the role of the computer science education community? ACM Inroads **2**(1), 48–54 (2011)
3. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change. AddisonWesley, 2 edn. (2004)
4. Blackwell, A., L. Church, L., T. Green, T.R.: The abstract is 'an enemy': Alternative perspectives to Computational Thinking. In: Proc. 20th Annual Workshop of the Psychology of Programming Interest Group. vol. 8, pp. 34–43 (2008)
5. Bobrov, E., Bucchiarone, A., Capozucca, A., Guelfi, N., Mazzara, M., Naumchev, A., Safina, L.: DevOps and its Philosophy: Education Matters! CoRR **abs/1904.02469** (2019), `http://arxiv.org/abs/1904.02469`
6. Brown, S.: 500 tips on group learning. Routledge (2014)
7. Buchanan, R.: Wicked problems in design thinking. Design issues **8**(2), 5–21 (1992)
8. Carter, L.: Ideas for adding soft skills education to service learning and capstone courses for computer science students. In: Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education. pp. 517–522. SIGCSE '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/1953163.1953312, `http://doi.acm.org/10.1145/1953163.1953312`

9. Ciancarini, P., Messina, A., Poggi, F., Russo, D.: Agile knowledge engineering for mission critical software requirements. In: Nalepa, G., Baumeister, J. (eds.) Synergies Between Knowledge Engineering and Software Engineering, Advances in Intelligent Systems and Computing, vol. 626, pp. 151–171. Springer (2018)
10. Conway, M.: How do committees invent. Datamation **14**(4), 28–31 (1968)
11. Cooper, H.: Scientific guidelines for conducting integrative research reviews. Review of Educational Research **52**(2), 291–302 (1982)
12. Cooper, H., Hedges, L., Valentine, J.: The handbook of research synthesis and meta-analysis. Sage (2009)
13. Csizmadia, A., Curzon, P., Dorling, M., Humphreys, S., Ng, T., Selby, C., Woollard, J.: Computational thinking: A guide for teachers. Computing at Schools (2015)
14. Dansereau, D.F.: Cooperative learning strategies. In: Weinstein, C., Goetz, E., Alexander, P. (eds.) Learning and study strategies: Issues in assessment, instruction, and evaluation, pp. 103–120. Academic Press (1988)
15. Denning, P.: Remaining trouble spots with Computational Thinking. Communications of the ACM **60**(6), 33–39 (2017)
16. Dingsøyr, T., Fægri, T.E., Dybå, T., Haugset, B., Lindsjørn, Y.: Team performance in software development: Research results versus agile principles. IEEE Software **33**(4), 106–110 (2016)
17. Edmonson, A.: Teaming to Innovate. Wiley (2013)
18. Edmonson, A.: Wicked Problem Solvers. Harvard Business Review **94**(June), 52 (2016)
19. European Community: Key competences for lifelong learning: European Reference Framework. http://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=LEGISSUM:c1109 (2007)
20. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Future of Software Engineering. pp. 37–54. FOSE '07, IEEE Computer Society, Washington, DC, USA (2007)
21. Glass, G.: Primary, secondary, and meta-analysis of research. Educational Researcher **5**(10), 3–8 (1976)
22. Guglielmino, L.M., Guglielmino, P.J.: Practical experience with self-directed learning in business and industry human resource development. New Directions for Adult and Continuing Education **1994**(64), 39–46 (1994)
23. Henderson, P.B.: Ubiquitous Computational Thinking. IEEE Computer **42**(10) (2009)
24. Hoskey, A., Zhang, S.: Computational Thinking: what does it really mean for the K-16 computer science education community. Journal of Computing Sciences in Colleges **32**(3), 129–135 (2017)
25. Howard, R.A., Carver, C.A., Lane, W.D.: Felder's learning styles, Bloom's taxonomy, and the Kolb learning cycle: tying it all together in the CS2 course. In: ACM SIGCSE Bulletin. vol. 28, pp. 227–231. ACM (1996)
26. Hung, W., Jonassen, D.H., Liu, R., et al.: Problem-based learning. Handbook of research on educational communications and technology **3**, 485–506 (2008)
27. Jackson, G.: Methods for integrative reviews. Review of Educational Research **50**(3), 438–460 (1980)
28. Johnson, D., Johnson, R.: Learning together and alone: Cooperative, competitive, and individualistic learning. Prentice-Hall (1987)
29. Johnson, D., Johnson, R., Smith, K.: Active learning: Cooperation in the college classroom. ERIC (1998)
30. Johnson, D., et al.: Cooperative learning in the classroom. ERIC (1994)

31. Johnson, M.: Should my kid learn to code? http://googleforeducation.blogspot.gr/2015/07/should-my-kid-learn-to-code.html (2015)
32. Joy, S., Kolb, D.A.: Are there cultural differences in learning style? International Journal of Intercultural Relations **33**(1), 69–85 (2009)
33. Katz, D.L.: Conference report on the use of computers in engineering classroom instruction. Communications of the ACM **3**(10), 522–527 (1960)
34. Kolb, D.: Learning Style Inventory technical manual. McBer Boston, MA (1976)
35. Kropp, M., Meier, A.: Teaching agile software development at university level: Values, management, and craftsmanship. In: Proc. 26th IEEE Conf. on Software Engineering Education and Training (CSEE&T). pp. 179–188 (2013)
36. Kropp, M., Meier, A.: New sustainable teaching approaches in software engineering education. In: Proc. IEEE Global Engineering Education Conference (EDUCON). pp. 1019–1022 (2014)
37. Meerbaum-Salant, O., Hazzan, O.: An agile constructionist mentoring methodology for software projects in the high school. ACM Transactions on Computing Education **9**(4) (2010)
38. Meier, A., Kropp, M., Perellano, G.: Experience Report of Teaching Agile Collaboration and Values: Agile Software Development in Large Student Teams. In: Proc. 29th IEEE Conf. on Software Engineering Education and Training (CSEE&T)). pp. 76–80 (2016)
39. Missiroli, M., Russo, D., Ciancarini, P.: Learning agile software development in high school: an investigation. In: Proc. 38th Int. Conf. on Software Engineering (ICSE). pp. 293–302 (2016)
40. Missiroli, M., Russo, D., Ciancarini, P.: Agile for Millennials: a comparative study. In: Proc. 1st Int. Workshop on Software Engineering Curricula for Millennials. pp. 47–53. IEEE Press (2017)
41. Missiroli, M., Russo, D., Ciancarini, P.: Teaching test-first programming: assessment and solutions. In: COMPSAC, 2017. IEEE (2017)
42. Pasupathy, S., Asad, A., Teng, P.Y.: Rethinking k–20 education transformation for a new age. www.atkearney.com/about-us/social-impact/related-publications-detail/-/asset_publisher/EVxmHENiBa8V/content/rethinking-k-20-education-transformation-for-a-new-age/10192 (2016)
43. Poggi, F., Rossi, D., Ciancarini, P., Bompani, L.: An application of Semantic Technologies to self adaptations. In: Proc. Int. Conf. on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI). pp. 1–6. IEEE (2016)
44. Polya, G.: How to solve it: A new aspect of mathematical method. Princeton university press (1957)
45. Raskino, M., Waller, G.: Digital to the Core: Remastering Leadership for Your Industry, Your Enterprise, and Yourself. Routledge (2016)
46. Rigby, D., Sutherland, J., Takeuchi, H.: Embracing Agile. Harvard Business Review **94**(5), 40–50 (2016)
47. Rittel, H., Webber, M.M.: 2.3 planning problems are wicked. Polity **4**, 155–169 (1973)
48. Rivera-Ibarra, J.G., Rodríguez-Jacobo, J., Serrano-Vargas, M.A.: Competency framework for software engineers. In: Proc. 23rd IEEE Conf. on Software Engineering Education and Training (CSEE&T). pp. 33–40 (2010)
49. Robins, A., Rountree, J., Rountree, N.: Learning and teaching programming: A review and discussion. Computer Science Education **13**(2), 137–172 (2003)
50. Rooksby, J., Hunt, J., Wang, X.: The theory and practice of Randori coding dojos. In: Cantone, G., Marchesi, M. (eds.) Proc. 15th Int. Conf. on Agile Software De-

velopment (XP2014). Lecture Notes in Business Information Processing, vol. 179, pp. 251–259. Springer (2014)
51. Rossi, D., Poggi, F., Ciancarini, P.: An application of Semantic Technologies to self adaptations. In: Proc. 33rd Symposium on Applied Computing. pp. 128–137. ACM (2018)
52. Rubin, K.: Essential Scrum: a practical guide to the most popular Agile process. AddisonWesley (2012)
53. Slavin, R.: Cooperative learning. Learning and Cognition in Education pp. 160–166 (2011)
54. Stanovich, K.: Matthew effects in reading: Some consequences of individual differences in the acquisition of literacy. Reading Research Quarterly pp. 360–407 (1986)
55. Steghöfer, J.P., Knauss, E., Alégroth, E., Hammouda, I., Burden, H., Ericsson, M.: Teaching Agile: addressing the conflict between project delivery and application of Agile methods. In: Proc. 38th Int. Conf. on Software Engineering (ICSE). pp. 303–312. ACM (2016)
56. Stobart, G.: The Expert learner. McGraw-Hill Education (UK) (2014)
57. Thomas, L., Ratcliffe, M., Woodbury, J., Jarman, E.: Learning styles and performance in the introductory programming sequence. In: ACM SIGCSE Bulletin. vol. 34, pp. 33–37. ACM (2002)
58. Vv.Aa.: The Glossary of Education Reform: 21st century skills. http://edglossary.org/21st-century-skills/ (2016)
59. Vygotsky, L.: Zone of proximal development. Mind in society: The development of higher psychological processes **5291**, 157 (1987)
60. Weber, E.P., Khademian, A.M.: Wicked problems, knowledge challenges, and collaborative capacity builders in network settings. Public administration review **68**(2), 334–349 (2008)
61. White, P., Rowland, A., Pesis-Katz, I.: Peer-led team learning model in a graduate-level nursing course. The Journal of Nursing Education **51**(8), 471–475 (2012)
62. Wing, J.: Computational Thinking. Communications of the ACM **49**(3), 33–35 (2006)
63. Yadav, A., Good, J., Voogt, J., Fisser, P.: Computational thinking as an emerging competence domain. In: Mulder, M. (ed.) Competence-based Vocational and Professional Education, Technical and Vocational Education and Training: Issues, Concerns and Prospects, vol. 23, pp. 1051–1067. Springer (2017)
64. Zhao, Y.: World class learners: Educating creative and entrepreneurial students. Corwin Press (2012)