People Analytics in Software Development

Leif Singer[1], Margaret-Anne Storey[1], Fernando Figueira Filho[2],
Alexey Zagalsky[1], and Daniel M. German[1]

[1] University of Victoria, Victoria, BC, Canada
{lsinger,mstorey,alexeyza,dmg}@uvic.ca
[2] Universidade Federal do Rio Grande do Norte, Natal, Brazil
fernando@dimap.ufrn.br

Abstract. Developers are using more and more different channels and tools to collaborate, and integrations between these tools are becoming more prevalent. In turn, more data about developers' interactions at work will become available. These developments will likely make People Analytics — using data to show and improve how people collaborate — more accessible and in turn more important for software developers. Even though developer collaboration has been the focus of several research groups and studies, we believe these changes will qualitatively change how some developers work. We provide an introduction to existing work in this field and outline where it could be headed.

Keywords: people analytics, developer analytics, social network analysis, feedback, collaboration, computer-supported collaborative work

1 Introduction

How people behave at work is becoming more and more measurable. For example, some companies are exploring the use of *sociometric badges* [25] that track the location and meta data about the interactions employees have during the day. But even without such devices, knowledge workers use more and more services that collect usage data about their activities.

Developers are an extreme case of this — the center of their work is their computer on which they write code, create and close tickets, and initiate text-based chats or calls that are all recorded in some way or another. Changes to a source code repository are recorded in full, task management applications have an API that makes changes available to API clients, and calendar systems track whom people have appointments with and when.

These changes in software development practice are paralleled by academic research: there are new insights into how developers collaborate, established and new methods in social network analysis, ongoing research in visualizing collaboration, recent attempts to measure developer affect, and research on giving (developers) feedback. We discuss these areas and how they are relevant to changes in how software developers work in Section 2.

While one option would be to discuss this development as an issue of increasing potential surveillance, we consider it an opportunity to make work itself better. The rise of distributed and remote work also means that developers need better support to improve how they collaborate.

An analytics system could track unusual events in source code repositories, project management tools, calendaring applications, or video conferencing. This would help monitoring whether some team members are isolated or disengaged — and thus might need assistance — or whether employees working on the same or related artifacts are communicating enough. Contrarily, visualizing and providing feedback about the steady progress that a team of developers makes could be introduced, e.g., as motivational support.

What is People Analytics? This opportunity to improve work is called *People Analytics* by some. There are multiple, but fuzzy definitions of the term. Even the book that helped coin the term, "People Analytics" by Waber [57], does not provide a definition of what is actually meant.

We therefore provide a definition by relying on our interpretation of People Analytics literature and a previous definition of generic analytics by Liberatore et al. [28]:

Definition 1. ***People Analytics*** *is the use of data, quantitative and qualitative analysis methods, and domain knowledge to discover insights about how people work together with the goal of improving collaboration.*

As such, People Analytics shares some similarities with the *Quantified Self* movement[3] — but in this case, it is applied to groups of professionals instead of private individuals.

Similar to the recently popularized areas of *Big Data* or *Data Science*, the specific technical methods are not necessarily new. However, their deliberate application within the context of the goal of improving how people work has, as far as we know, not been embraced by the software engineering research community. We believe that it could be a useful umbrella term to bring disparate research groups closer together.

We distinguish People Analytics from code analytics and from the mining software repositories lines of research. Instead of focusing on the artifacts that people create and modify, we now concentrate on how they work together to achieve their goals. The central element we are interested in is not the source code file or the commit, but the developer.

Phases in People Analytics. Again leaning on previous work on general analytics [28] and Singer's work [46], we divide the application of People Analytics into six activities: *goal setting*, *data collection*, two parallel phases of *analysis* and *intervention*, *change*, and *reflection* (cf. Fig. 1).

Goal Setting. In this first phase, those implementing a People Analytics project define what the project's subject of interest is, what it is supposed to discover more about, and what it is supposed to change or improve.

[3] *"individuals engaged in the self-tracking of any kind of biological, physical, behavioral, or environmental information"* [55]

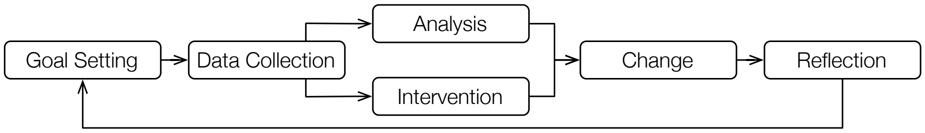


Fig. 1. The phases in People Analytics.

Data Collection. Now, the data needed for achieving the goal is chosen. A collection strategy is formulated and systems are being set up for cleaning, aggregating, and otherwise manipulating the collected data.

Analysis and Intervention. While or after the data is being collected, the analyst will either run one or multiple analyses of the data, deploy an intervention — such as a feedback system based on the collected data (cf. section 3) — or both.

Change. Depending on the project, an analysis could provide proof or hints for required changes in how developers work together, or an intervention could provoke these changes to occur without further actions. For example, an analysis could show that all communication between two groups goes through a single developer, posing a risky bottleneck. A pure analysis-based management intervention could then consist of assigning individuals to similar bridging roles. An intervention in our sense would give the affected developers automatic feedback about this bottleneck and potentially trigger them to reorganize around it themselves.

Reflection. Finally, those implementing the People Analytics project will want to reflect on whether the project reached its stated goal, why it worked, or possibly why it did not work. Often novelty in reconfigurations or feedback systems can wear off, making them less effective over time. Therefore, whether the change will last needs to be considered and perhaps a strategy to evolve the manual or automatic intervention will need to be put in place. In this phase, further data collection to gather insights about the successes and failures that occurred in the project can be helpful in understanding the second-order effects of an intervention — including, but not limited to qualitative post-mortem interviews with the affected developers (cf., e.g., section 3).

People Analytics has a lot of potential for improving how developers collaborate. At the same time, there are multiple challenges that need to be overcome to be able to implement a successful People Analytics project. Mistrust, fear of surveillance, or choosing metrics for feedback that make matters worse are only a few examples from the problems that can arise.

Those implementing a People Analytics project need to be fluent in diverse methods of data collection and analysis, but also need expertise in creating interventions by using insights from human-computer interaction as well as organizational studies and management science.

We contribute our views on these challenges, possible solutions, and promising paths to be taken that could help fulfill the promise of this emerging field.

This paper is structured as follows: after this introduction, we review existing literature that could help those trying to drive People Analytics in software engineering forward. Section 3 presents a simple application of People Analytics in which we nudged student developers towards a certain behavior when committing to a version control system. We then conclude the paper with an outlook.

2 Background and Related Work

This section gives an overview of existing literature that we consider to be relevant and interesting when applying People Analytics to software engineering. It is not a complete list, but rather a collection of research that we believe could be inspiring when conducting research on or implementing People Analytics projects in practice. At the same time, we use it to roughly define what our view of People Analytics in software development entails.

We see five major areas that are especially relevant for People Analytics research in software engineering: general insights on how developers collaborate, social network analysis to discern interaction patterns and relationships between developers, visualizing collaboration to support developer understanding, measuring developer affect to enable adaptable development tools, and giving developers feedback to help them understand and act upon their own behaviors. We now discuss each in turn.

In their discussion of groupware [13], Ellis, Gibbs, and Rein emphasized the importance of communication, collaboration, and coordination for group-based activities. Since a few decades, software development is usually performed in groups that work together. We therefore start with a discussion of how developers collaborate and what challenges they meet. We then turn to how communication, collaboration, and coordination of developers can be measured. We close with a brief discussion of how to use such metrics for giving developers and their managers feedback.

2.1 Insights on Communication, Collaboration, and Coordination

To understand what the customs and challenges are in developer collaboration, we discuss some relevant studies.

Organizational Factors: Lavallee and Robillard [26] report on a study in which they observed a team of developers in a large organization. Over a period of ten months, they attended weekly status meetings and noted interactions and the topics that were discussed. The researchers condensed their observations into a list of ten organizational factors that hurt the overall quality of the software produced in the project under study. Several of them are related to communication between developers themselves or between developers and business stakeholders — such as undue pressure from management delivered through informal channels, differences between the formal and the actual processes, or needing the right social capital to be able to get work done.

Pair Programming: Stapel et al. [50] study how student developers communicate during pair programming: they find that many conversations between developers are about concrete code, but when considering conversation length they spend the most time discussing software design. When considering how conversations change over time, discussions about code becomes less and less — possibly because developers learn syntax and project-specific conventions. Plonka et al. [38] found that pair programming as a method that influences developer communication can help diffuse knowledge within teams and organizations.

Team Mood and Interaction Intensity: Schneider et al. [44] investigate the relationships between media, mood, and communication in teams of student developers. Amongst other findings, the authors report that excessive positive mood around a project's midpoint could actually be detrimental, as it would often signal too much optimism that could easily derail the project in its later stages. To this end, the authors recommend providing developers with technical milestones such as objective quality gates to provide feedback to developers regularly, improving their self-evaluation. They also find that having early indicators for isolated team members or high variance in a developer's participation can be helpful tools for project managers. Since they find a relationship between developer affect and project success, the authors recommend that the mood in developer teams be regularly measured to serve as a heuristic to alert managers about potentially problematic projects. At the same time, both data about mood and interaction intensity between team members should be made available to the developers themselves to make them aware of things that might be going wrong in their project.

The Role of Social Software: In previous work, we and others have studied how developers use signals in social media to collaborate [53]. These signals could, e.g., be text and visuals that convey someone's status in a community or their experience with a particular technology. We found that these signals influence how developers collaborate with others, how recruiters evaluate them, what and from whom they learn, and even whether and how they test their open source contributions.

Singer et al. [47] report on a qualitative study involving both developers with active social media profiles and software development recruiters. The authors find that public profiles, aggregated activity traces, as well as the detailed activity data and artifacts developer create are useful for both assessment between developers and also the assessment of developers by external parties, such as recruiters. Capiluppi et al. [8] argue that these activity signals could help recruiters find suitable candidates even if these do not have a traditional degree, helping non-traditional candidates start their career in software development.

Pham et al. [37] conducted a qualitative study with users of GitHub, finding that the affordances of the site — several of them reflecting information about people and groups, not artifacts — are helpful in nudging inexperienced developers towards the expected behavior and norms of an open source project they want to contribute to. In part, this was due to normative behavior — such as what a good pull request should contain — being published and discussed in the open.

Finally, Singer et al. [48] conducted a study on how and why software developers use the microblogging service Twitter. Two of the more prominent findings were that a) microblogging helps developers connect with one another and participate in cultures they are geographically not a part of and b) that access to such a diverse network of other developers can provide developers with rich opportunities for learning new technologies and practices.

Take-away: There has been quite some research on issues such as how developers work, why they behave the way they do, why they use the tools they use, what challenges they really face, and how they currently cope with these challenges. We should use these studies to learn from and to better understand what problems we should attempt to solve.

2.2 Methods: Social Network Analysis

Now that we have provided an impression of recent research in how developers collaborate, we turn to research that attempts to measure it. We begin with the fundamentals of social network analysis and then discuss its relevance to team work.

Fundamentals: A social network is a structure that is composed of actors and the ties between them. Actors — the nodes — and ties — the edges between the nodes — form a graph. Depending on the application, ties can be weighted or unweighted, directed or undirected. The weight of an edge can, e.g., represent the strength of a relationship or the number of interactions between two actors. The social networks of development teams can, for example, be derived from organizational data [33], commit histories [30,10], or email exchanges [2].

Parts of this graph can be considered their own sub-networks and can follow distinct patterns. Some typical sub-networks to consider in analyses are, for example, the *singleton* (an unconnected node), the dyad (a connection between two nodes), or the triad (with all possible edges between three nodes). Another important sub-network is the clique: a number of nodes where every node is connected to every other node.

Network science has invented many different metrics and properties that can be helpful when trying to understand how and why a social network functions:

- *Connectedness:* two nodes are connected if there is a path in the graph that leads from one node to the other.

- *Centrality:* this describes a few different measures that determine how *central* a node is in a graph. Examples are *degree centrality* (a node's number of edges) or *betweenness centrality* (measures the importance of a node to the shortest paths through the graph).
- *Density:* the number of edges in a graph divided by the numbers of *possible* edges in the graph. Intuitively, this measures how well connected a social network is.
- *Centralization:* measures how evenly distributed *centrality* is across all the nodes of a graph.

Discovering Roles: Such properties can be useful, e.g., in automatically finding *bridges* — nodes that form the only connection between two separate groups — or for discovering equivalent roles in a social network. Golbeck [16] provides an accessible introduction to these and many more analyses.

For example, Burt [5] provides a note on discovering equivalent roles in a social network by categorizing every actor by the triad types they have. Practically, this allows clustering of actors by the role they occupy in an organization. Comparing the result of such an analysis with the officially assigned roles could help uncover discrepancies — enabling organization to, e.g., become aware of what its employees actually do or to bring official roles more in line what is really happening.

Relationship Intensity: Weak ties [17] between people within different parts of an organization can speed up projects by providing a project team with knowledge not available within its own ranks [22]. However, this is only true if the needed knowledge is simple enough — needing to transfer complex knowledge via weak ties can slow projects down. Strong ties across organizational boundaries might be preferable in this case, however these are usually not numerous enough to ensure access to the relevant knowledge. The weight of an edge between actors in a social network can also be important when measuring the decay [6] between relationships that naturally happens over time. Organizations could use this information to improve employee retention.

Nagappan et al. [33] calculated eight different *organizational* measures about the Microsoft developers who worked on Windows Vista. Examples are *number of engineers* having edited a binary, *number of ex-engineers*, the position of a binary's owner in the *hierarchy*, or the *number of different organizations* making changes to a binary. They found that their measures were more reliable and accurate at predicting failure-proneness than any of the code-based metrics they compared them against.

Social Networks over Time: Apart from the relatively static metrics discussed so far, social networks and the events happening to them can also be analyzed with a temporal dimension added. Xuan et al. [58] use both data on communication (exchanged messages) and collaboration (artifacts edited) over time to reveal the structure of development teams and model individual and team productivity. In this case, actors in the network can be either developers or edited artifacts.

Composing Teams: Finally, Reagans et al. [39] show that for organizations, it might be easier and more effective to compose teams based on employees' social networks instead of their demographics. There is no doubt that social networks within organizations can have a significant impact on the organization's productivity, speed, and success — and that being able to analyze these social networks is a necessary step to improvement.

Take-away: By combining methods and insights from classic social network analysis with organizational studies, we can gain valuable insights that neither discipline would have discovered on its own.

2.3 Methods: Visualizing Collaboration

While an analysis may often require only data and an algorithm, visualizing data can aid exploration — we therefore briefly discuss some approaches to visualizing the collaboration between software developers.

Repository Evolution: Both Gource [9] and Code Swarm [34] produce high-level video visualizations of a source repository's history. Software evolution story lines [35] attempt to show more detail than the aforementioned, but in consequence do not scale as well to larger projects.

Visualizing Information Flows: Stapel et al. [51,52] present *FLOW Mapping*, a visualization of both formal and informal information flows through diverse communication channels. Their approach is both for planning and monitoring the media and indirections through which developers communicate during a project. It is meant to improve communication in distributed software projects. In an evaluation within a distributed student project, the authors monitored meta data about communication through channels such as instant messaging, audio and video calls, as well as source code repositories. The project manager in the evaluation reported that the technique helped plan and measure compliance with a communications strategy, and also to become aware of the different developers' locations and connections.

Related to the *FLOW Mapping* visualizations, Schneider and Liskin [43] define the *FLOW Distance* to measure the degree of indirection in communication between developers. Their metrics takes into account indirections created through other people, documents, and processes. The authors believe this to provide an objective metrics to measure how *remote* developers might feel from each other.

Meeting Profiles: Liskin et al. [29] measured meeting frequency and length in a project with student developers. The authors visualize the *meeting profiles* of different teams and assign them to different categories. By comparing meeting profiles with the actual projects, they find that high project pressure seems to correlate with more frequent meetings. The authors argue that such a visualization could be a significant help for managers who want to assess the stress level in the projects they're responsible for. Extreme meeting profiles or sudden changes in meeting intensity could be sufficient heuristics for evaluating a project's problems more closely. At the same time, the data to produce meeting profiles can be measured easily.

Take-away: Visualizing activity in code repositories is an interesting field of study. But crucial insights can be gained using simpler visualizations and instead focusing more effort on figuring out how to measure things that have previously been too hard to measure.

2.4 Methods: Measuring Developer Affect

Having discussed the measuring of interactions between developers, we now turn to the individual — specifically, developers' emotions, moods, or affects. Graziotin et al. [19] provide a comprehensive overview of the involved psychological concepts and challenges in conducting experiments that measure affect.

Affect and Productivity: Previous research suggests this is an important topic. Meyer et al. [31] report on a study about the perceptions developers have about their own productivity. Their participants report that completing big tasks results in feelings of happiness and satisfaction. Relatedly, Graziotin et al. [18] found that the attractiveness of a task and perceiving to have the skills required to complete it correlate with developers' self-assessed productivity. Khan et al. [24] found that developer mood affects debugging tasks. Finally, in a study on the reasons for developer frustrations, Ford and Parnin found that the majority of developers is able to recall recent experiences of severe frustration [14]. Being able to measure the affective state of a developer therefore seems potentially valuable.

Measuring Emotions: Fritz et al. [15] provide a method based on biometric measures to estimate task difficulty. The authors argue that this will enable them to stop developers to let them reconsider the task when it seems too difficult. Relatedly, Mller et al. [1] use biometric sensors to detect when a developer is "stuck." Shaw [45] provides a method to assess developer emotions purely based on psychological questionnaires — which is likely more cost-effective, but also similarly distracting and likely less reliable than biometric sensors. Cheaper and less intrusive methods, such as the mouse- and keyboard-based approach proposed by Khan et al. [23], could prove to be more applicable, even though they seem to be much less reliable.

Guzman et al. [20] report on a sentiment analysis of commit messages from a set of open source projects. The authors find that developers in more distributed teams tend to write more positive messages, that commit messages in Java-based projects tend be more negative, and that code committed on Mondays also comes with more negative messages. However, Murgia et al. [32] show that emotions expressed within software development are often more nuanced than expected, and that an automatic analysis will likely not be precise enough.

Summarizing the above, we note that it would be valuable to be able to assess developers' moods and emotions, but sufficiently reliable solutions are currently costly, inconvenient, or both. Light-weight solutions do not yet seem to achieve a suitable level of reliability.

Emotionally Adaptive Tools: However, assuming a light-weight, but reliable method to measure developer affect would exist, interesting possibilities would open up in supporting collaborative work. Dewan [12] proposes a collaboration system that tailors messages, tasks, and notifications between colleagues to their current level of frustration and interruptibility.

Take-away: Measuring developer affect is an interesting and challenging topic, but we need to keep in mind that results so far have been mixed. Future improvements in methods will likely bring new ethical challenges.

2.5 Feedback Interventions

The previous sections have described what data could be interesting to gather and how it could be manipulated to make insights more obvious. We now focus on delivering feedback and insights to developers to support them and positively influence their behavior.

Workspace Awareness: Several existing interventions fall into the sub-class of workspace awareness tools. These use data on developer activity and display them to developers — usually in a continuous and unobtrusive manner. Palantr by Sarma et al. [41] is one of the more prominent examples. Notably, it uses a change severity metric to decide how prominently a change should be displayed to a developer.

What Developers Want: Treude et al. [56] recently reported on a qualitative study in which they investigated how developers believe activity should be measured, summarized, and presented. Among other findings, they report that unexpected events such as changes in estimates, new dependencies, or API changes could be highly valuable for developers to be notified of. With *UEDashboard* [27], they have recently shown a tool that supports detecting and displaying such unusual events.

Finding Expertise: Guzzi and Begel [21] present *CARES*, a Visual Studio extension. When opening a file under version control, *CARES* displays a list of developers who had previously committed changes to that file. The authors report that developers in an evaluation successfully used CARES to find relevant colleagues to talk to.

Establishing and Changing Habits: Pham et al. [36] augment the Eclipse IDE with a testing-specific dashboard that presents developers with a set of signals inspired by the social transparency framework [54]. The authors' aim is to nudge novice or newly hired developers towards the testing culture implemented by existing developers.

Finally, *Teamfeed* [49] is a Web application that uses a very simple metric — the number of commits made by a developer in a project — to create a project-internal developer ranking. The first author conducted a quasi-experiment on this intervention with student developers, which we report on in detail in the following section.

Take-away: Often, the problems of software engineering are being worked on from the perspective of software engineers. Adding a more design-oriented lens to the process can lead to more interesting results in both practice and research. In our opinion this specific combination of fields still contains much unrealized potential for interdisciplinary collaboration.

3 Making Analytics Feedback Useful: Gamification of Version Control

The preceding section has discussed a range of different ideas on measuring and giving feedback on developer collaboration — yet it has merely scratched the surface of each of the few areas it addresses. We suspect that choosing the right things to measure and giving appropriate feedback could be one of the more challenging aspects in People Analytics.

In the following, we show how a very simple metric — the number of commits a developer makes to a repository — can be used to influence the commit behavior of a cohort of student developers. This illustrates that *what is measured potentially matters less than how it is presented* to developers.

The quasi-experiment we are about to describe evaluated the *Practice Adoption Improvement Process* (*PAIP*). PAIP is a systematic process for improving the adoption of software engineering practices, and includes a catalog of patterns that can be applied during that process.

The process, the catalog, as well as the quasi-experiment are part of Singer's thesis [46]. They are an example for a specific application of People Analytics — namely affecting behavior change in how developers collaborate.

3.1 Introduction

Together with Stapel and Schneider [49], the first author conducted the following quasi-experiment while in Hannover, Germany. In a student project lasting a full semester, we attempted to improve the adoption of version control practices in small teams of student developers. We used early versions of both PAIP and the catalog of adoption patterns and used the experience from this evaluation to refine both.

A quasi-experiment is an experiment in which the assignment of subjects to the control vs. treatment conditions is non-random. In our case, the control group was comprised of data from the version control repositories of previous years in which our group organized this project. The treatment group was the cohort of students taking the project in the fall term of 2011.

Developers do not always strictly follow software development processes and software engineering practices [46]. Even though individuals may be aware of a practice and its advantages, as well as capable of implementing it, they do not always adopt it — a situation called the *Knowledge-Attitude-Practice-gap* (or *KAP-gap*) in the innovation-decision process by Rogers [40].

In centralized version control systems such as Subversion[4], developers should commit early and often to decouple changes from each other and to spot conflicts with the work of other developers earlier [4]. To make browsing historical data easier, each change should include a description of its contents — the commit message. Even though many developers know of these or similar guidelines, they do not always follow them. This can influence the maintainability — and therefore quality and costs — of a software project negatively.

In our experience, student projects can be problematic in this regard: developers include several different features and fixes in a single commit. They leave commit messages empty. These problems occur regularly, even though the organizers of the project emphasize every year that they want students to commit regularly, since the version control repository is the only way for our group to continue work on the students' projects later — for which a meaningful commit history would be useful. The organizers also emphasize that other students — peers of the student developers — might need to access the repository in the future, e.g., to improve on one of the student projects for a thesis.

However, the problem persists. We suspect the reason to be a combination of missing knowledge regarding best practices and a lack of motivation for spending the additional effort needed for thoughtful commits. We therefore decided to apply an early version of PAIP in the fall 2011 term's project and used a selection of adoption patterns to create a persuasive intervention to alleviate this problem. Before documenting our application of PAIP, the following section introduces the experiment context.

3.2 Experiment Context

Each fall semester, our research group organizes the *software project* (SWP) course, a mandatory course for computer science undergraduates. The course has roughly 35 to 70 participants every time, most of them in their fifth semester. The students form teams of four to six members, and elect a project leader as well as a quality agent. The project starts at the beginning of October and lasts until the end of January.

The members of our research group act as customers, proposing software projects that we would like to have developed. That way, we are able to provide

[4] `http://subversion.apache.org`

projects with real requirements while keeping control of their size and technological demands. This is beneficial for the comparability of projects in experiments such as the one presented here. Usually, each student team will work on a different project with a different customer, however some projects may be given to multiple teams to work on independently.

Each project is divided into three main phases: requirements elicitation, software design, and implementation. After that, customers get to try out the produced software and assess their compliance with requirements in a short acceptance phase.

After each phase, the teams have to pass a quality gate (QG) to proceed to the next phase. This ensures a minimum quality of the artifacts developed in each phase. If a team fails a quality gate, they are allowed to refine their artifacts once. Failing the quality gate for a single phase repeatedly would lead to failing the course. However, this has not happened yet.

So far, we have conducted this course every year since 2004. For this experiment, we only consider the years starting with 2007, as this was the first year we had the students use Subversion for version control. The process we use and the size of the projects have not changed significantly since then. The duration has constantly been the whole fall semester. While each project is different, we take care to always provide projects with similar requirements regarding effort and proficiency in software development. This is to ensure fairness between the teams with the added benefit of better comparability.

The preconditions regarding the participants have been very stable. Our group teaches all the basic courses on software engineering, software quality, and version control. The contents of these courses have remained similar over the years.

In the first phase, students make appointments with their customers and interview them about their requirements. They produce a requirements specification that they need to get signed by their respective customer to proceed to the next phase. In the second phase, the teams can choose between preparing an architecture or creating exploratory prototypes. In both variants, they are required to produce a software design document. They implement the actual applications in the third and final phase.

During the project, a member of our group will act as coach, answering questions about technical subjects and the development process. To create time scarcity, each team receives six vouchers for customer appointments of 15 minutes each and six vouchers for coach appointments of 30 minutes each.

At the end of the project, the customer executes the acceptance tests from the requirements specification and decides whether rework is needed. Once the customer has finally accepted or rejected the software product, the role-play ends.

Finally, we conduct an *LID* session with each team. LID — short for *Lightweight Documentation of Experiences* — is a technique for the elicitation of project experiences [42]. A typical LID session for the course takes about two hours during which the team members and a moderator jointly fill in a template for experience elicitation. An LID session inquires students about impressions, feelings, conflicts, and advice, and has them review the whole project from beginning to end. In the sessions, we emphasize that their passing of the course will not be affected anymore and encourage them to honestly describe the negative experiences as well.

For each team, we provide a Subversion repository, a Trac[5] instance for issue tracking, and a web-based quality gate system that is used to progress the teams through the project phases. The Trac instance is linked to the team's version control repository, so students are able to see their team's commits using either Trac or any Subversion client.

3.3 Experiment Design

This section documents how we applied PAIP and deployed a persuasive intervention — with a Web application called *Teamfeed* as its treatment — to a student population of 37 participants. The section's organization is based on PAIP's first five steps: *Characterize Context*, *Define Adoption Goal & Metrics*, *Choose Adoption Patterns*, *Design Treatment*, and *Deploy Intervention.* The succeeding section implements the sixth step: *Analyze Results.*

Characterize Context In the first step of PAIP, the change agent determines the current context in which PAIP is to be applied. This entails the practice for which adoption should be improved and its properties, as well as the characteristics of the developer population.

The Software Engineering Practice To apply PAIP, the change agent decides whether the practice for which adoption is to be influenced is comprised of primarily routine or creative tasks. This experiment is concerned with practices for committing to version control, which involves deciding when to commit, what to commit, and how to describe it in the commit message. Based on the rough guidelines given in PAIP's description [46], we determine that our practice entails *creative* tasks.

The Developer Population Regarding the developer population, the change agent determines whether there are any existing adopters of the practice that could act as role models. As we have seen *some* student developers adhering to good committing practices in previous years, we decide that we can indeed assume existing adopters in our population.

Define Adoption Goal & Metrics In this second step, we define the adoption goal that the intervention should be optimized for. To measure success in the last step, we define metrics.

[5] `http://trac.edgewall.org`

Defining a Goal As advised by PAIP, we first choose simple goals that will improve the performance of those developers with less experience. Further improvements would be possible in future iterations. Therefore, we want students to commit at all, to commit more often, to commit more regularly, to write commit messages for their commits, and to write longer commit messages overall.

When defining the goal, PAIP requires the change agent to choose either one or both of "Start Adopting a New Practice" or "Improve Adoption of a Known Practice"; for our experiment, we choose both. In every previous instance of the software project course, there have been some students who never committed to version control, while most did commit at least once. Some of those committed only in bursts, some committed more regularly; some wrote commit messages, and some others did not. For the goals above, we therefore want to both *increase* adoption and *increase* frequency.

Research Questions For the context of this evaluation, we formulate our goals into research questions that we will investigate in the succeeding section, documenting PAIP's sixth step:

- *RQ 1:* Does our intervention influence student developers to make more commits and space them out more evenly over time?
- *RQ 2:* Does our intervention influence student developers to write more and longer commit messages?

Defining the Metrics To measure our previously defined goals, we choose the metrics listed in Table 1. The table also assigns metrics to research questions. Most metrics are self-explanatory, except possibly the time between consecutive commits. We use this metric to measure whether developers commit more *regularly* — that is, more evenly spread out over time, with fewer *bursts* of commits. Assuming a constant number of commits, a more regular committing behavior would then result in the median time between commits to *increase*.

We follow the *playful metric* recommendation [46] in that we never connected the committing behavior of students with actual consequences, such as passing or failing the course. We made this clear to students in person and via email,

RQ	Metric	Counting Rule
RQ 1	c	Number of commits per user
	$\Delta t_{C,avg}$ $\Delta t_{C,med}$	Average and median time between two consecutive commits of a user in seconds
RQ 2	c_M	Number of commits with message per user
	c_M/c	Message-to-commit-ratio per user
	$l_{M,avg}$ $l_{M,med}$	Average and median number of characters of the commit messages of a user

Table 1. Summary of the defined metrics, assigned to their respective research questions.

emphasizing that we provided the Teamfeed tool purely for the students' own benefit.

Finally, PAIP requires us to define when to take measurements and to record a baseline measurement. For this quasi-experiment, our baseline — i.e., the control group — consists of the Subversion repositories collected during previous instances of the course from 2007 to 2010. These repositories contain the commits of 214 students. The results are measured after the course has ended.

Hypotheses For our experiment, we derive the alternative hypotheses for our research questions. We assume that a positive influence on the commit behavior of developers can be exerted by deploying our persuasive intervention. This influence should lead to more commits per developer, to temporally more evenly spaced commits, to more commits with messages per developer, and to longer commit messages. Accordingly, our respective null hypotheses are that the deployment of the intervention has no influence on these phenomena.

Choose Adoption Patterns The previous two steps of applying PAIP reveal that we want to either start or improve the adoption of a practice that is comprised of relatively creative tasks. We assume to have some existing adopters. Based on this information, we choose the following adoption patterns for our intervention in the third step of PAIP. For each pattern, we also repeat its *solution* below.

- ***Normative Behavior:*** "Make explicit what normative behavior should be by continuously publishing the behavior of developers, positively emphasizing desirable behavior."
- ***Triggers:*** "Use notifications to cue developers to applying a practice by directing their attention to a task related to the practice. To support motivation, associate triggers with positive feedback or a goal to be reached. Do not overload developers with triggers."
- ***Points & Levels:*** "Award points and levels for the activity that is to be started or intensified. Provide a space for users to display their points and levels, e.g., on a user profile. Give clear instructions on how to attain different levels."

  Note that the recommendation for *clear instructions* was added only after the completion of this experiment, and therefore was not taken into account.
- ***Leaderboard:*** "Use a metric that measures compliance with the software engineering practice to rank developers against each other, creating explicit competition. If possible, have groups compete against each other instead of individual developers against each other."

  Note that the recommendation for *groups competing against each other* was added only after the completion of this experiment, and therefore was not taken into account.
- ***Challenge:*** "Provide developers with explicit, attainable, and challenging goals. Make sure developers understand what the conditions for attaining

the goal are and give explicit feedback on results. Prefer challenges that require the developer to learn something new over those that merely require reaching a certain performance as measured by a metric."
Note that the recommendation for *learning goals* was added only after the completion of this experiment, and therefore was not taken into account.
- ***Progress Feedback:*** "Provide developers with positive feedback on the progress they are making in their application of the practice."

Except for the knowledge stage — which was not addressed by the early version of PAIP —, this selection of adoption patterns covers all stages of the innovation-decision process that are relevant to PAIP. Three of the patterns are from the *motivation* category: as mentioned before, we suspect missing motivation to be a reason for the adoption issues.

3.4 Designing a Treatment

Using the adoption patterns chosen in the previous step, we now create a treatment that implements the patterns. This design is in part informed by the examples listed for each adoption pattern.

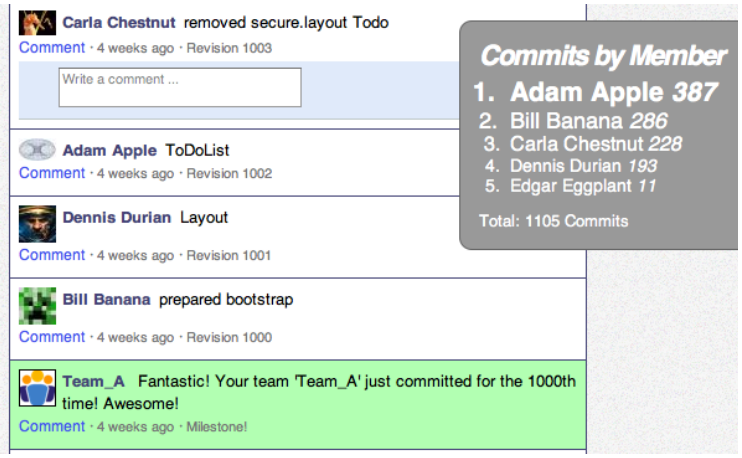
Newsfeed A newsfeed displaying the version control commits for each team implements the *Normative Behavior* adoption pattern. When no commit message is given, the application displays a highlighted text stating that a message is missing.

Leaderboard A list of a team's members, ordered by their respective number of commits so far, implements the *Leaderboard* adoption pattern. Next to the name of each team member, the member's current number of commits is given. Below, the total number of commits for the team is displayed.

Milestones At predefined thresholds for numbers of commits, the application congratulates users and teams on reaching a milestone. This implements the *Points & Levels* pattern. By slowly increasing the distance between the thresholds, this also implements the *Challenge* pattern: by committing, developers are able to recognize that there will be another milestone at an even higher number of commits, providing them with a goal. The congratulatory messages implement the *Progress Feedback* pattern.

Notifications For positive events, such as reaching an individual or team milestone, the application sends out email notifications — this implements the *Triggers* adoption pattern. The congratulatory messages in the emails implement the *Progress Feedback* pattern.

Weekly Digest Every Sunday, the application emails a weekly digest to each developer. It shows the current leaderboard, as well as any milestones reached in that week. This implements the *Triggers* adoption pattern. The congratulatory messages that were given when a milestone was reached implement the *Progress Feedback* pattern.



Carla Chestnut removed secure.layout Todo
Comment · 4 weeks ago · Revision 1003
Write a comment ...

Adam Apple ToDoList
Comment · 4 weeks ago · Revision 1002

Dennis Durian Layout
Comment · 4 weeks ago · Revision 1001

Bill Banana prepared bootstrap
Comment · 4 weeks ago · Revision 1000

Team_A Fantastic! Your team 'Team_A' just committed for the 1000th time! Awesome!
Comment · 4 weeks ago · Milestone!

Commits by Member

1.	Adam Apple	*387*
2.	Bill Banana	*286*
3.	Carla Chestnut	*228*
4.	Dennis Durian	*193*
5.	Edgar Eggplant	*11*

Total: 1105 Commits

Fig. 2. A screenshot of Teamfeed's newsfeed and leaderboard.

Teamfeed We now present Teamfeed, a Web application that uses email for notifications. It periodically reads the commits to each team's repository and saves them to a database. These are then displayed in a newsfeed for each team. Every student in the project can log in to Teamfeed using their Subversion account and is then presented with their respective team's newsfeed. The newsfeeds of other teams are not accessible to the students. Fig. 2 shows an anonymized screenshot of the application in which the names of students and their team have been altered.

Reaching a milestone generates a special post to the newsfeed. For the milestones, we defined thresholds of 1, 10, 25, 50, 100, 250, 500, 750, 1000, 1500, 2000, 2500, 3000, 4000, 5000, 7500, and 10000 commits. These generate posts such as *"Congratulations! Jane Doe has reached her 200th commit!"* or *"Wonderful! Your team has just reached the 1000th commit!"* We based the thresholds on previous semesters' commit counts and added a buffer.

On the right, the leaderboard lists the team members and the counts of their respective commits so far. For higher ranks, name and commit count are displayed in a larger font.

Each Sunday at around 3pm, Teamfeed sent out the weekly email digest to each student. The digest summarizes how many commits the individual student has made in the past week, but also provides this information about their teammates. It also mentions milestones that were reached during the week and shows the current state of the leaderboard.

Deploy Intervention Once the treatments have been created, the change agent deploys them as a persuasive intervention in the organization. We deployed *Teamfeed* at the start of the software project course in the fall term of

Group	**Control**					**Teamfeed**
Term	**2007**	**2008**	**2009**	**2010**	Σ	**2011**
n	40	40	76	58	214	37
n_C	31	36	73	55	195	37
$n - n_C$	9	4	3	3	19	0
n_C/n	78%	90%	96%	95%	91%	100%
c_{total}	3973	3680	6993	7223	21869	4842
c_{total}/n	99	92	92	125	102	131

Table 2. Overview of data sources and their values for number of subjects (n), number of subjects who committed (n_C), number of subjects who never committed ($n - n_C$), percentage of committing subjects (n_C/n), number of total commits (c_{total}), and average commits per subject (c_{total}/n).

2011. The students were told that the purpose of Teamfeed was to support their collaboration.

3.5 Results and Analysis

The final stage of PAIP involves taking a measurement and comparing it to the baseline to assess the effectiveness of the intervention. This informs the next iteration of the process.

Table 2 shows the data sources we used for data collection in our experiment. It includes the data from five years of the software project course, i.e., the data accumulated in the fall terms of the years 2007 through 2011. The first four years were used as the control group. In 2011, we introduced the Teamfeed application and therefore used it as our treatment group.

In total, there were 26,711 commits in the five years (c_{total}). In the first four years, each participant made 102 commits on average (c_{total}/n). In 2011, this value was at 131 commits. 251 students took the course over the five years, which can be seen as n in Table 2. The treatment group consisted of 37 participants.

n_C documents the number of students that did commit at all in the respective year. As the values for n_C/n show, all students in the treatment group committed at least once to version control (100%). In the previous years, however, some participants never made a single commit (i.e., on average, 91% committed at least once).

Descriptive Statistics We now present the data we collected for the metrics we defined, aggregated in Table 3 and visualized as box plots in Fig. 3. Each set of data is declared for the control group (C) and the treatment group (T), respectively. For each value, we provide the minimum, the median, and the maximum value.

For example, Table 3 shows a 76% increase in median commits per participant (c) for the treatment group. The ratio of commits with messages to commits overall (c_M/c) increased by 75%. We now discuss whether these and other differences are statistically significant.

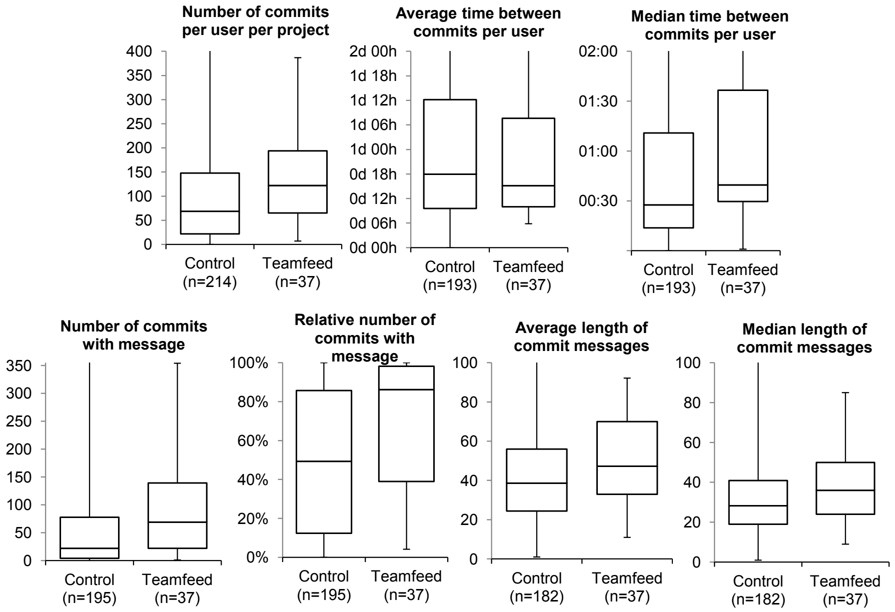


Fig. 3. Box plots of the data collected for the metrics.

Metric	Group	Min	Median	Max
c	Control (n=214)	0	69	683
	Treatment (n=37)	7	122	387
$\Delta t_{C,avg}$	C (n=193)	00:00	17:55	>17d
hh:mm	T (n=37)	05:51	15:09	>8d
$\Delta t_{C,med}$	C (n=193)	00:00	00:27	>6d
hh:mm	T (n=37)	00:00	00:39	>1d
c_M	C (n=195)	0	22	587
	T (n=37)	1	69	354
c_M/c	C (n=195)	0%	49%	100%
	T (n=37)	4%	86%	100%
$l_{M,avg}$	C (n=182)	1	39	211
	T (n=37)	11	47	92
$l_{M,med}$	C (n=182)	1	28	165
	T (n=37)	9	36	85

Table 3. Minimum, median, and maximum values for the collected metrics: number of commits per subject (c), average ($\Delta t_{C,avg}$) as well as median ($\Delta t_{C,med}$) time between commits, number of commits with a message per subject (c_M), percentage of commits with a message (c_M/c), and average ($l_{M,avg}$) as well as median ($l_{M,med}$) lengths of commit messages.

Hypothesis Testing For most metrics, we were able to determine a statistically significant difference between the values for the control group and the values for the treatment group. We performed a Kolmogorov-Smirnov normality test for all the metrics. These tests showed that the data do not follow a normal distribution. Therefore, we had to use the non-parametric two-tailed Mann-Whitney U test to test for the significances of differences. Table 4 presents the results of our tests for statistical significance.

For research question 1, there is a significant difference between the number of commits per student for the two groups: an increase in 76% ($c; p < 0.01$). The average time between commits does not differ significantly ($\Delta t_{C,avg}$). However, the median time between commits exhibits a significant ($\Delta t_{C,med}; p < 0.05$) difference: an increase in 44%. We therefore reject the null hypotheses for the first and the third metrics of research question 1.

The measurements for research question 2 show significant differences. The number of commits with messages per developer increased by 213% ($c_M; p < 0.01$); the ratio of commits with messages to overall commits increased by 75% ($c_M/c; p < 0.01$).

The difference for the average length of commit messages is not significant, with a 20% increase ($l_{M,avg}; p < 0.1$). The difference for the median length of commit messages is significant with a 28% increase ($l_{M,med}; p < 0.05$). We therefore reject three of the four null hypotheses for research question 2.

Qualitative Analysis To better understand the effects of our intervention, we now provide an additional qualitative discussion based on the *LID sessions* conducted at the end of the project. At the end of the sessions, we inquired about each team's impressions of Teamfeed. This provided us with some notable insights:

- More experienced developers often ignored Teamfeed and the emails it sent. Some even had setup a filter in their email clients for this purpose. However, only few seemed to be annoyed by the emails. In an industry setting, one might want to give developers a way to opt out of such email. Yet, none asked us about such an option during the course.

RQ	Metric	Control	Treatment	Difference	Confidence
RQ 1	c	69	122	+76%	$p < 0.01$
	$\Delta t_{C,avg}$	17:55	15:09	-15%	$p > 0.1$
	$\Delta t_{C,med}$	00:27	00:39	+44%	$p < 0.05$
RQ 2	c_M	22	69	+213%	$p < 0.01$
	c_M/c	49%	86%	+75%	$p < 0.01$
	$l_{M,avg}$	39	47	+20%	$p < 0.1$
	$l_{M,med}$	28	36	+28%	$p < 0.05$

Table 4. Overview of statistical test results.

- Several of the more novice developers reported that they felt motivated by the milestones. The only team which reached the *1000 commits* milestone was comprised of such members.
- No developer reported any manipulative attempts by themselves or by team mates. To help ensure this, we performed regular sanity checks of commits and commit messages, finding no indication for manipulation (such as empty commits). Overall, we estimate to have sampled about 5% of commits in this manner.
- One developer explicitly said that Teamfeed's milestones made him commit in smaller batches. Instead of putting several bug fixes into a single commit, he committed his changes after every single fix. In our view, this is desirable behavior for centralized version control systems.

Research Questions In our research question 1 we asked: *Does our intervention influence student developers to make more commits and space them out more evenly over time?*

To answer this question, we defined three metrics: the number of commits per student, the average time between commits, and the median time between commits. Based on our measurements, we were able to reject the null hypothesis for the first and the third metrics. Therefore, we conclude that our treatment was indeed able to influence student developers to make more commits and space them out more evenly over time. Not only did it lead to a significantly higher number of commits per developer, but also resulted in a more evenly distributed time between commits.

Research question 2 asked: *Does our intervention influence student developers to write more and longer commit messages?*

For this question, we defined four metrics: the number of commits with messages, the ratio of commits with messages to overall commits, the average length of commit messages, and the median length of commit messages. For three of these metrics, we were able to reject the null hypothesis. We conclude that the introduction of our application did indeed influence student developers to write more and longer commit messages. More commits contained commit messages at all, and those that did contained longer messages.

3.6 Threats to Validity

This section discusses threats to the validity of our quasi-experiment. We show how we tried to minimize them through the experiment design and mention remaining limitations.

Internal Validity A significant difference between the control group and the treatment group does not in itself represent a causal relationship between our intervention and the differences in measurement. Other confounding factors might have had an influence. The population itself, the students' education, details in the execution of the course, and our behavior towards the students might all have been different.

The advantage of using different populations for the control and treatment groups, however, is that there should have been no confounding effects with regard to learning or maturation. In addition, we took care to execute the course the same as in previous years. As our group also provides the basic software engineering courses, we feel qualified to say that we did not notice any notable differences in the students from the control group compared to the students in the treatment group. Additionally, our courses provide the basic education on version control, which was the same for both groups.

It is conceivable that general trends in software development and general computer use changed how students adopt version control practices.

Through the qualitative interviews conducted at the end of the study, we might have noticed such trends, as we also inquired about how students used version control — not merely how they used Teamfeed. In these interviews, we did notice one change in computer use: to exchange files, students had started using Dropbox, a service that makes it easy to share folders between different users and synchronizes them over the Internet. In previous cohorts, students had used version control for exchanging files.

This change in computer use might have influenced the measured impact of our intervention, as there would have been fewer commits in the treatment cohort. Those would have been commits made solely for the quick and ephemeral exchange of project files, however — something that we wouldn't necessarily consider the intended use of version control.

Construct Validity Whether the practices we chose for version control are preferable in a given software engineering situation is debatable. However, we consider them an important step for the population we investigated. Populations at other levels of version control proficiency may require different interventions. Even though the use of metrics in software development can be problematic [3], our research questions and the metrics we derived address the adoption of these practices as directly as possible. We therefore consider them appropriate.

In a future investigation, we plan to examine any quality differences in the commits and commit messages of the control and treatment groups. A preliminary investigation of 100 commit messages showed indications for a decrease of nonsense messages ("hahaha!"), a decrease of purely technical messages that do not mention a change's utility ("add getter getUser()"), and an increase in mentions of functional changes ("fix incompatibility with framework version 1.2.5").

One possible effect of public, competitive metrics is that people try to "game the system" — i.e., they try to increase their value for the metric using the easiest strategies, which might often not be what the creators of the system intended. In our case, these would be empty commits or nonsense commits. To rule this effect out, we randomly sampled some of the commits from our treatment group. We found no indications for invalid or manipulative commits.

Conclusion Validity To mitigate threats to conclusion validity, we used the data collected over several years of the software project course for our control group. These 214 participants, combined with 37 participants in the treatment group, were suitable to provide statistically significant results. To decrease the risk of manual errors, all data were collected automatically.

External Validity The participants of our experiment were mostly students of computer science in their 5th semester. As the German Bachelor of Science degree lasts 6 semesters, most students were almost finished with their studies. As our treatment was directed at issues with version control practices we had experienced from similar populations, we cannot generalize this concrete intervention to different populations. Another application of PAIP, while more elaborate than a simple transfer of the intervention, would be more sensible.

It is questionable how many metrics and additional interventions can be introduced before software developers start ignoring such measures. The tolerable amount of such treatments might be very low. Further research regarding such scenarios is warranted.

Similarly, our software projects are restricted to a single semester, i.e., about four months. We do not think that our experiment can be generalized to much longer runtimes, as potential numbing effects seem plausible. Again, further research is needed in this regard.

3.7 Summary

Our quasi-experiment demonstrated that PAIP and the catalog of adoption patterns can be used to improve the adoption of software engineering practices — in this case, the commit behavior of student developers. While we tried to design our experiment to minimize threats to validity, some of them were beyond our control. It is therefore still possible that the effects we measured were created or influenced by other, confounding factors. However, the qualitative data from the LID sessions back our interpretation.

This specific intervention worked for less experienced software developers in a university setting. As we argued in section on validity, the intervention itself might not generalize. However, this is exactly what PAIP intends: it provides a way to create interventions that are tailored to a practice, an adoption problem, a population, and adoption goals.

Our quasi-experiment has shown that the application of PAIP is feasible and can be effective. While this need not be true for every possible adoption problem or situation, this data point serves as a good indicator. More evaluations in different contexts would be needed to improve confidence in this regard.

The metrics we chose to feed back to the students were relatively simple — a commit count per team member and a newsfeed of commit messages. All this information was available to previous teams via Subversion clients such as the one included in the Eclipse IDE. Still, the different presentation of this data made a difference in observed developer behavior.

We believe that this suggests that People Analytics projects do not need to measure especially interesting behavior or even do so in a necessarily exact manner. Instead, we believe that the presentation and the context of the chosen metrics matters most.

4 Outlook

We have discussed many options and opportunities for implementing People Analytics projects, and have hinted at the benefits. There are many different analyses and metrics available, originating from diverse fields. We hope that these can be useful as starting points for future research.

In the previous section, we have shown that complicated metrics possibly are not that important, after all. Maybe the most important research that is still to conduct lies more in the field of HCI and psychology — our conception of what it takes to create a tool that supports collaborative work still seems more diffuse than we are comfortable with. This feeling is likely related to the fact that collaboration is mostly about people, and the path towards gathering reliable knowledge about people and the processes between them seems much less clear than algorithmic endeavors.

So far, we have ignored the more severe challenges present in People Analytics. We conclude this paper by discussing some of the challenges we believe will have the most impact on implementing People Analytics, and what we consider to be likely good practices to handle them.

4.1 Gaming the System

We have discussed how metrics are likely less important than their presentation. Using a systematic process that aligns one's goals with metrics seems sensible and helpful.

But at the same time, how metrics are used must be informed and systematic, which can be challenging as there are many unknowns about the possible side effects or second-order effects of using a certain metric.

Considering the quasi-experiment discussed in the previous section — what would have happened if the number of commits by a student would have been tied to their course grade?

It seems very likely that students would have been more motivated to commit more. However, it seems similarly likely that the barrier to trying to cheat the system would have been lower, as the potential reward would have been much higher. Campbell [7] gives an insightful overview of how metrics — in his case, for public policy — can have the opposite effect of what was intended.

Using metrics to influence behavior is a double-edged sword: it can have positive results, but can also backfire. To lessen this risk, metrics should be kept playful. That is, they should not be tied to a person's income, their grades, or their career options. But at the same time, for a metric to have any effect, it needs to be meaningful to those involved. Leaning on Deci's and Ryan's self-determination theory [11], relatedness — relationships to others — could help create meaning.

4.2 Surveillance

Tracking activity data from potentially many services that developers use throughout the day also elicits thoughts about surveillance. If an employer uses People Analytics to improve internal collaboration, employees could easily feel monitored.

We believe that this would especially be the case if employees are not sufficiently informed about what is being tracked about their behavior, what happens to the data, and what the goal of the tracking is. After all, some employers could indeed think that the number of commits per day that a developer pushes to the repository should have a direct influence on salaries or promotions.

The only somewhat reliable solution to not tracking too many details about developers' behaviors would be to aggregate personally identifying data. Simply anonymizing such data has been shown to be easily reversible, especially when social networks are involved.

4.3 Public Data vs. Accessible Data

At the same time, we acknowledge that many things are already public, at least *within a team* — commit data and calendar data being two of the more obvious examples. That colleagues can look up these activity traces if they so wish we consider as accepted as normal today.

In our experiment, we did not make more data public — we merely presented it differently. All the data we used was available to every student all the time, e.g., through a Subversion client. The same goes for GitHub's newsfeed — it is also just making existing, accessible data more visible and accessible with fewer barriers.

This illustrates that the mere availability of data is usually not sufficient to make it have an impact on behavior. In order to do that, data needs to be presented in an accessible and goal-oriented manner.

4.4 Transparency

The most sensible strategy likely is to make transparent what is being measured and to what end. This will help developers understand what their activity data is being used for and, ideally, they will see a worthy cause in their employer's intentions. At the same time, employees can also act as an ethical canary for their employer, notifying management of instances where a line has been crossed. However, this will require a workplace culture that permits this. Part of such a culture would be attempts from the organization to actually get any honest feedback on their People Analytics projects from employees.

Ethical standards regarding these things will surely change with time. Yet organizations should err on the side of standards that are too high.

4.5 Conclusions

Despite the challenges mention above, the potential upsides of People Analytics for software development are enticing. Collaboration is one of the hardest problems in software engineering, and any improvement would likely be welcome by developers, organizations, and researchers alike.

More and more Internet-based services are becoming available and being used. Data about employee behavior will likely become more rather than less. Both researchers and practitioners need to be prepared so we handle it appropriately. To do so, we need open discussions about ethical standards, best practices on which metrics to use, and on appropriate delivery of feedback.

There are many more open challenges in People Analytics than we could possibly have discussed, and even for those we have touched upon, we barely scratched the surface. With this in mind, we hope our contribution can help continue the discussion within the software engineering research community. We would feel honored if you took us up on that — our email addresses are listed at the top of the paper for a reason.

References

1. Stuck and frustrated or in flow and happy: Sensing developers emotions and progress. In *Proc. Int. Conf. on Soft. Eng.*, ICSE '15, 2015.
2. C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 137–143, New York, NY, USA, 2006. ACM.
3. E. Bouwers, J. Visser, and A. van Deursen. Getting what you measure. *Communications of the ACM*, 55(7):54–59, 2012.
4. Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *Proc. ESEC/FSE*, pages 168–178, 2011.
5. R. S. Burt. Detecting role equivalence. *Social Networks*, 12(1):83–97, 1990.
6. R. S. Burt. Decay functions. *Social networks*, 22(1):1–28, 2000.
7. D. T. Campbell. Assessing the impact of planned social change. *Evaluation and Program Planning*, 2(1):67–90, 1979.
8. A. Capiluppi, A. Serebrenik, and L. Singer. Assessing technical candidates on the social web. *Software, IEEE*, 30(1):45–51, Jan 2013.
9. A. H. Caudwell. Gource: Visualizing software version control history. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 73–74, New York, NY, USA, 2010. ACM.
10. C. de Souza, J. Froehlich, and P. Dourish. Seeking the source: Software source code as a social and technical artifact. In *Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work*, GROUP '05, pages 197–206, New York, NY, USA, 2005. ACM.
11. E. Deci and R. Ryan. *Handbook of self-determination research.* The University of Rochester Press, 2002.
12. P. Dewan. Towards emotion-based collaborative software engineering. In *Cooperative and Human Aspects of Software Engineering (CHASE), 2015 IEEE/ACM 8th International Workshop on*, pages 109–112, May 2015.

13. C. A. Ellis, S. J. Gibbs, and G. Rein. Groupware: some issues and experiences. *Commun. ACM*, 34(1):39–58, Jan. 1991.
14. D. Ford and C. Parnin. Exploring causes of frustration for software developers. In *Cooperative and Human Aspects of Software Engineering (CHASE), 2015 IEEE/ACM 8th International Workshop on*, pages 117–118, May 2015.
15. T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 402–413, New York, NY, USA, 2014. ACM.
16. J. Golbeck. *Analyzing the social web*. Newnes, 2013.
17. M. S. Granovetter. The Strength of Weak Ties. *American Journal of Sociology*, 78(6):1360–1380, May 1973.
18. D. Graziotin, X. Wang, and P. Abrahamsson. Are happy developers more productive? In J. Heidrich, M. Oivo, A. Jedlitschka, and M. Baldassarre, editors, *Product-Focused Software Process Improvement*, volume 7983 of *Lecture Notes in Computer Science*, pages 50–64. Springer Berlin Heidelberg, 2013.
19. D. Graziotin, X. Wang, and P. Abrahamsson. Understanding the affect of developers: Theoretical background and guidelines for psychoempirical software engineering. In *Proceedings of the 7th international workshop on Social Software Engineering (to appear)*, SSE '15, New York, NY, USA, 2015. ACM.
20. E. Guzman, D. Azócar, and Y. Li. Sentiment analysis of commit comments in github: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 352–355, New York, NY, USA, 2014. ACM.
21. A. Guzzi and A. Begel. Facilitating communication between engineers with CARES. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1367–1370, Piscataway, NJ, USA, 2012. IEEE Press.
22. M. T. Hansen. The search-transfer problem: The role of weak ties in sharing knowledge across organization subunits. *Administrative science quarterly*, 44(1):82–111, 1999.
23. I. A. Khan, W.-P. Brinkman, and R. Hierons. Towards estimating computer users mood from interaction behaviour with keyboard and mouse. *Frontiers of Computer Science*, 7(6):943–954, 2013.
24. I. A. Khan, W.-P. Brinkman, and R. M. Hierons. Do moods affect programmers debug performance? *Cognition, Technology & Work*, 13(4):245–258, 2011.
25. T. Kim, E. McFee, D. O. Olguin, B. Waber, and A. . Pentland. Sociometric badges: Using sensor technology to capture new forms of collaboration. *Journal of Organizational Behavior*, 33(3):412–427, 2012.
26. M. Lavalle and P. N. Robillard. Why good developers write bad code: an observational case study of the impacts of organizational factors on software quality. In *Proceedings of the 2015 International Conference on Software Engineering*, 2015.
27. L. Leite, C. Treude, and F. Figueira Filho. Uedashboard: Awareness of unusual events in commit histories. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '15, New York, NY, USA, 2015 (to appear). ACM.
28. M. J. Liberatore and W. Luo. The analytics movement: Implications for operations research. *Interfaces*, 40(4):313–324, 2010.
29. O. Liskin, K. Schneider, S. Kiesling, and S. Kauffeld. Meeting intensity as an indicator for project pressure: Exploring meeting profiles. In *Cooperative and Human*

Aspects of Software Engineering (CHASE), 2013 6th International Workshop on, pages 153–156, May 2013.

30. L. Lopez-Fernandez, G. Robles, J. M. Gonzalez-Barahona, et al. Applying social network analysis to the information in cvs repositories. In *International Workshop on Mining Software Repositories*, pages 101–105. IET, 2004.
31. A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann. Software developers' perceptions of productivity. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 19–29, New York, NY, USA, 2014. ACM.
32. A. Murgia, P. Tourani, B. Adams, and M. Ortu. Do developers feel emotions? an exploratory analysis of emotions in software artifacts. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 262–271, New York, NY, USA, 2014. ACM.
33. N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: An empirical case study. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 521–530, New York, NY, USA, 2008. ACM.
34. M. Ogawa and K.-L. Ma. code_swarm: A design study in organic software visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 15(6):1097–1104, Nov 2009.
35. M. Ogawa and K.-L. Ma. Software evolution storylines. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 35–42, New York, NY, USA, 2010. ACM.
36. R. Pham, J. Mörschbach, and K. Schneider. Communicating software testing culture through visualizing testing activity. In *Proceedings of the 7th international workshop on Social Software Engineering (to appear)*, SSE '15, New York, NY, USA, 2015. ACM.
37. R. Pham, L. Singer, O. Liskin, F. Figueira Filho, and K. Schneider. Creating a shared understanding of testing culture on a social coding site. In *Proc. Int. Conf. on Soft. Eng.*, ICSE '13, pages 112–121, 2013.
38. L. Plonka, H. Sharp, J. Van der Linden, and Y. Dittrich. Knowledge transfer in pair programming: an in-depth analysis. *International Journal of Human-Computer Studies*, 73:66–78, 2015.
39. R. Reagans, E. Zuckerman, and B. McEvily. How to make the team: Social networks vs. demography as criteria for designing effective teams. *Administrative Science Quarterly*, 49(1):pp. 101–133, 2004.
40. E. M. Rogers. *Diffusion of Innovations*. Free Press, 5th edition, 2003.
41. A. Sarma, Z. Noroozi, and A. van der Hoek. Palantir: raising awareness among configuration management workspaces. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 444–454, May 2003.
42. K. Schneider. LIDs: A Light-Weight Approach to Experience Elicitation and Reuse. In F. Bomarius and M. Oivo, editors, *Product Focused Software Process Improvement*, volume 1840/2000 of *Lecture Notes in Computer Science*, pages 407–424. Springer, 2000.
43. K. Schneider and O. Liskin. Exploring flow distance in project communication. In *Cooperative and Human Aspects of Software Engineering (CHASE), 2015 IEEE/ACM 8th International Workshop on*, pages 117–118, May 2015.
44. K. Schneider, O. Liskin, H. Paulsen, and S. Kauffeld. Media, mood, and meetings: Related to project success? *ACM Transactions on Computing Education*, (accepted—to appear(n/a):n/a, 2015.

45. T. Shaw. The emotions of systems developers: An empirical study of affective events theory. In *Proceedings of the 2004 SIGMIS Conference on Computer Personnel Research: Careers, Culture, and Ethics in a Networked Environment*, SIGMIS CPR '04, pages 124–126, New York, NY, USA, 2004. ACM.
46. L. Singer. *Improving the Adoption of Software Engineering Practices Through Persuasive Interventions*. PhD thesis, Gottfried Wilhelm Leibniz Universität Hannover, 2013.
47. L. Singer, F. Figueira Filho, B. Cleary, C. Treude, M.-A. Storey, and K. Schneider. Mutual assessment in the social programmer ecosystem: An empirical investigation of developer profile aggregators. In *Proc. 2013 Conf. Comput. Supported Cooperative Work*, CSCW '13, pages 103–116, NY, USA, 2013. ACM.
48. L. Singer, F. Figueira Filho, and M.-A. Storey. Software engineering at the speed of light: How developers stay current using twitter. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 211–221, New York, NY, USA, 2014. ACM.
49. L. Singer and K. Schneider. It was a Bit of a Race: Gamification of Version Control. In *Proceedings of the 2nd international workshop on Games and software engineering*, 2012.
50. K. Stapel, E. Knauss, K. Schneider, and M. Becker. Towards understanding communication structure in pair programming. In *Agile processes in software engineering and extreme programming*, pages 117–131. Springer, 2010.
51. K. Stapel, E. Knauss, K. Schneider, and N. Zazworka. Flow mapping: planning and managing communication in distributed teams. In *Global Software Engineering (ICGSE), 2011 6th IEEE International Conference on*, pages 190–199. IEEE, 2011.
52. K. Stapel and K. Schneider. Managing knowledge on communication and information flow in global software projects. *Expert Systems*, 2012.
53. M.-A. Storey, L. Singer, B. Cleary, F. Figueira Filho, and A. Zagalsky. The (r) evolution of social media in software engineering. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 100–116, New York, NY, USA, 2014. ACM.
54. H. C. Stuart, L. Dabbish, S. Kiesler, P. Kinnaird, and R. Kang. Social transparency in networked information exchange: a theoretical framework. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, CSCW '12, pages 451–460, New York, NY, USA, 2012. ACM.
55. M. Swan. The quantified self: Fundamental disruption in big data science and biological discovery. *Big Data*, 1, 2013.
56. C. Treude, F. Figueira Filho, and U. Kulesza. Summarizing and measuring development activity. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '15, New York, NY, USA, 2015 (to appear). ACM.
57. B. Waber. *People Analytics: How Social Sensing Technology Will Transform Business and What It Tells Us about the Future of Work*. FT Press, 1st edition, 2013.
58. Q. Xuan, H. Fang, C. Fu, and V. Filkov. Temporal motifs reveal collaboration patterns in online task-oriented networks. *Physical Review E*, 91(5):052813, 2015.