Trends and Concepts in the Software Industry II: Development of Enterprise Software

Team 5: Providing Data Context During Development

Thomas Bünger, Felix Leupold, Johan Uhle,
Patrick Schilf, Lauritz Thamsen, Fabian Tschirschnitz

Johannes Wust
Franziska Haeger, Dr. Anja Bog, Dr. Juergen Müller, Prof. Hasso Plattner
Hasso-Plattner Institute

March 15, 2013

Table of Contents

Please note: This table of contents displays each author's name after the corresponding chapter title. Patrick Schilf was additionally responsible for adjusting the separate single chapters by different authors to shape one fluent document.

1 Introduction

Data sources explode in their size, complexity and speed they grow with. Whether Enterprise applications, games, or the vast amount of online services at least since the rise of mobile platforms represent a considerable percentage of the amount of data. Data is the central element that everything else revolves around. However, one has to understand that the question is no longer one of storage but rather one of processing these huge amounts. With databases growing not only in their total amount of tuples but also in the number of table columns, calculations become more complex and time critical. The introduction of SAP's *HANA* marked a big step in addressing these problems.

However, while it seems natural that software applications are tightly connected to their underlying data sources, this does surprisingly not apply to the actual development of these applications. IDEs such as Eclipse have come to be highly customizable tool chains that allow for integration of multiple tasks a software developer comes across in his daily work cycle. One that is missing here, however, is a linkage of code to its Data Context at the time of coding. In a conventional development workflow a programmer writes code, builds the application and tests it. Only in the last stage does he or she get in touch with the actual data that their applications are supposed to deal with, which means that potential errors or performance bottlenecks will only be discovered then. Development speed could be considerably increased if the programmer could take care of the mentioned issues from the very beginning in the cycle.

A second factor that interferes with developer's efficiency is that of the absence of a seamless workflow. There are bigger and smaller sources for interruptions a developer might face while coding. However, in many cases short interruptions tend to appear far more often than greater ones and can sum up to considerably distract from the actual problem a programmer is trying to solve. In the described situation a programmer does not only face the absence of a missing linkage between the code and the corresponding data, but he or she also has to navigate between multiple environments to compensate for this issue. Environments are often heterogenous, which even prevents an easy and comfortable transfer of code amongst them.

As part of the seminar *"Trends and Concepts in the Software Industry II - Development of Enterprise Software"* held at the *Hasso-Plattner-Institute* in Potsdam we developed a prototype to address the described inconveniences of a interruptive workflows as well as missing code-data-connections. This document presents our final iteration of the prototype as well as documents the design thinking inspired process we used to come up with our solution. Chapter 2 lays down the process design thinking is based on, which also represents the guideline through this document. The following chapters reveal information we collected during interviews with people in different positions in the software industry and how they helped us to iteratively build our prototype. At the end of this report we suggest factors we believe need to be concerned to transform our concept into a working implementation.

2 Design Thinking

Design thinking refers to the method of finding an innovative solution to an abstract or ill-defined problem. It attempts to structure the ideation process and provides guidelines, methodologies and frameworks that help in problem forming, -solving and -design. Design thinking is a human-centered approach with multidisciplinary collaboration and iterative improvements to produce innovative products, systems and services [2]. There are various interpretations and definitions of design thinking. We refer to design thinking as defined by the Hasso Plattner Institue of Design at Stanford and Potsdam. In this definition the design thinking process contains six different phases, which are pictured in Figure 1. The process is not linear, which is indicated by the connections amongst different non-succeeding phases. Every instance of a design thinking project is different. It is possible to jump back and forth between the different phases as you encounter new inflection points.

In the following, we briefly explain the aim of each phase as they also reflect the pathway in the course of our project and the structure of this report.

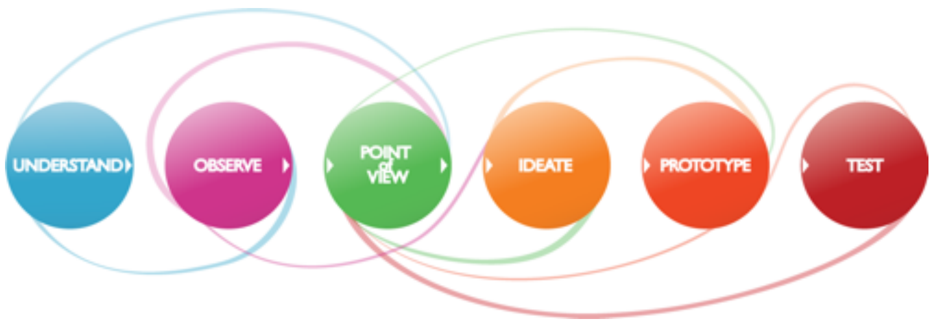


Fig. 1. Design Thinking process according to *Hasso Plattner Institue of Design* at Stanford and Potsdam.

Understand In the first phase of the process, the team becomes familiar with the problem domain by talking to experts and conducting research. The goal is to develop background knowledge through these experiences. It is the foundation for all further steps. Throughout the process this stage might be visited again as new areas of problems, ideas or solutions emerge.

Observe What people say does not always correspond to their actual feelings, thoughts or actions. Therefore, it is necessary to watch how people behave and interact. These observations can be used to talk to users, ask questions and reflect on their behaviors. They lay the foundation for insights which are needed to define a point of view later in the process. Another point of this phase is to develop a kind of empathy for the user.

Point Of View The point of view consists of three parts: A persona, which is a concrete specification of the target user the team is designing for. Second, a need that this persona has and that the team is trying to satisfy with its solution. Lastly, an insight that has been generated from the observation made in the second phase. A commonly used method of defining a point of view is the "How might we..." question, which is a statement in a form "How might we help our persona to satisfy his or her need". While the preceding phases tried to capture a wide range of users and problems, in this stage the team settles on a single point from which it can converge again in the next phase.

Ideate While the first three phases focus on defining and understanding the problem, the ideation phase opens up the solution space for the problem. The team is challenged to brainstorm as many solutions as possible by going for quantity, deferring judgment and building on the ideas of others.

Prototype Prototyping is a crucial part of the design thinking process. It allows to make the ideas visible and tangible. Quick and cheap prototypes, e.g. sketches on paper, are well suited for conveying the idea and getting feedback. People tend to hesitate to criticize or suggest changes for overly polished prototypes. A key concept is to fail early and often.

Test In the test phase the team collects feedback on their ideas which they made experienceable through their prototype. The goal of this phase is to learn which parts of the design work and which don't. Testing ensures that the product is desirable for the end user and stresses the user focus of design thinking. Latest from here on the team continues with one of the earlier phases.

Each potential solution has to satisfy three requirements as depicted in Figure 2. Desirability ensures, it addresses an actual need of the user. Feasibility means that the solution should be technically possible to implement. Lastly, it has to be viable for the business partner providing the solution. Innovative solutions lie in the intersection of the three sets. Design thinking helps to find solutions that lie in this intersection.

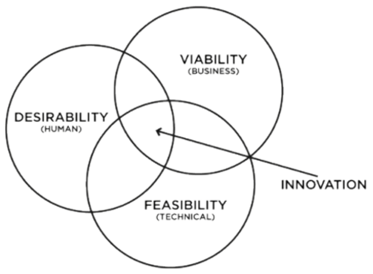


Fig. 2. Solution can be categorized into different sets. An innovative solution has to be viable, desirable and feasible.

3 Understand

During our research, we spoke to various developers, architects and managers in the software industry. Company sizes ranged from small startups to big players of a few thousands employees. In this section we will summarize the interviews and the insights we gained from them.

3.1 Interviews

Danny Tramnitzke from PSIpenta The company *PSIpenta* builds ERP systems for SMBs with a focus on manufacturing and supply chain management. Their core product has initially been developed in Cobol. Parts have later been ported to *C++* and *Java* and is still in progress. Development happens largely in the Eclipse IDE, which has been extended and customized by PSIpenta to enable automated code generation and integration into their workflow and environment. Work is flexible with home and remote work.

Our interview partner Danny works on porting parts of the old code from Cobol to Java but also develops entirely new features. Danny estimated the time spent in daily communication at around 30%. This includes replying to emails and attending meetings with colleagues. In the interview, Danny put an emphasis on code quality. To him, a core means to achieve this was automated testing, best done by involving the customer directly. As major churns Danny mentioned interruptions from co-workers as well as from a slow and/or unintuitive tool chains. As an example he mentioned that building a manually testable version of his product takes more than half an hour.

Nicole von Steht from SAP *SAP* is one of the biggest enterprise application companies in the world. Nicole works as frontend application developer at SAP. Her team consists of twelve people and uses the *Scrum* methodology. The team is located on one floor and communicates mostly face-to-face.

As one of her core values, she pointed out quality of code. She stated to take care of its assurance by unit and integration tests with continuous integration as well as rotating pair programming. She said that her development environment was a customized version of Eclipse.

Tillman Giese from SAP Tillman works at SAP on the *HANA Security Engine*. His team consists of eight people and is split over two offices.

Tillman regularly deals with other teams that are globally distributed. He said this was often difficult because of time differences as well as cultural ones. Tillman reported to get a lot of support requests, which most of the time would aim at already answered questions. Requesters could resolve the questions themselves, if they put in some effort to research in the first place. Often, he said, those requests would interrupt his workflow, illustrated by this quote: *"I wish, I could just work on one task for a full hour"*.

Frank Brunswig from SAP Frank is working as Chief Architect at SAP. He has been involved with the company for 20 years, worked on various projects and currently supervises the creation of the AppBuilder platform, which enables the creation of custom mobile applications.

Apart from doing small prototype implementations to evaluate new technologies, Frank does not code regularly. His work mainly happens on a organizational level, where he performs requirement analyses, organizes his teams, maintains the cross-organization-alignment and finds cross-team synergies.

Frank prefers his teams to use all-in-one IDE solutions for coding, which in the best case would integrate the complete workflow from finding requirements over writing code to the final deployment.

His major problems in development are unclear or changing requirements. He is also not an advocate of *Agile* methodologies, as they would badly distribute responsibilities and make the development process harder to control.

Christof Dorner from Readmill The Berlin-based startup *Readmill* build an ebook reader and a social network around books. Christof is a backend developer for the *Ruby on Rails* application. His team, consisting of him and another three members, works with the *Kanban* methodology.

Christof mentioned an interruption-free workflow as an important goal for his day-to-day work. To minimize interruptions through his co-workers, they set up an asynchronous company chat that allows everyone to drop in and out of discussions according to their own likings. Christof always wears noise-canceling headphones when coding alone. He optimized his tools highly to reduce any waiting time while working. This includes customizing his editor and writing small scripts for churn tasks. Furthermore, when choosing the technology for Readmill, the whole team takes into account how easy development with this technology will be. Readmill make developer satisfaction one of the most important components during the technology decision process.

Tests are a core component of Christof's workflow for developing new features or fixing bugs. He does not follow straight test-driven principles but rather switches between writing code, testing on the console or web app, and writing tests seamlessly. He puts an emphasis on the fact that all deployed code has a high test coverage. Code reviews are done via Pull Requests on *Github*.

As problems while developing, Christof mentioned refactoring of old code (especially tests), reproducing bugs, and interruptions of any kind.

Ralf Tomczak, André, Holger and Oliver from Mobile.de The *eBay* subsidiary *Mobile.de* is located in Berlin/Dreilinden and is Germany's biggest online marketplace for vehicles. Ralf is the head of technology and was responsible for migrating the web platform from *Perl* to eBay's prevalent programming language *Java* in 2007. André, Holger and Oliver are team leads of the consumer, commercial and mobile segment respectively.

They all mentioned agile development as being crucial for their cross-functional teams. During the last years of Scrum methodology and especially Kanban-like processes they were able to replace their weekly release cycles with multiple roll-outs per day. They stated that this increased velocity helped them to fix bugs usually within a day.

From an architectural viewpoint, one of the biggest problems they mentioned was implementing internationalization in their platform. Characteristics of the different countries have to be considered, but due to their shared database, changes for one country sometimes result in unintended side effects for others.

Svetlana Peycheva from Alacris Theranostics *Alacris* are a small German pin-off company from Max-Plank Institute and do analyses to support cancer treatment. We spoke to Svetlana, who is the only software developer in the team. Her biggest problem is that the software system she develops is based on code written by multiple developers in the past, none of which is still part of the team or even reachable. She mentioned that therefore the major problems would be dreadful code quality, inadequate documentation, and the lack of any test suite.

Guenther Tolkmit from VMS At *VMS* we spoke to Guenther, who is a software engineer working on database integrations. During the interview he mentioned how working on real user data was superior to generated test data. He also pointed out to spend a lot of time debugging code while often feeling the lack of tool support to efficiently do so. This is why his usual approach involves trial-and-error debugging, which he sees as a bad practice.

3.2 Summary

We interviewed a wide variety of programmers with very different problems. Especially programmers in big companies seemed to have problems that were not of technical but rather organizational nature. We did not consider those problems suitable to be worked on in the context of our project, but rather decided to focus on technical issues.

Even though the interviewees were very heterogenous, we were able to gain some insights:

Efficient tools are key to successful development All developers we spoke to showed big interest in the tools they use and were passionate about tooling. They customize their tools themselves, add plug-ins and tinker with configurations. Some build and maintain tools themselves and one company even dedicated a whole group just for tool support.

When asked about the impact of tools on their development process, we heard strong opinions from describing how unfunctional tools limited the productivity of the developer up to that well-functioning tools of the company would make for major competitive advantages.

Programmers work best when in flow Many programmers told us about their need for uninterrupted work, the so-called flow when they fully emerge their mind in the task at hand. But for many, a seamless workflow seemed to be rare in their day-to-day processes as interruptions appeared to happen a lot more often than desired. The threshold over what constitutes as an interruption ranged from small things like switching windows over waiting longer times (e.g. for a test run to complete) to being removed from the coding situation (e.g. by attending a meeting).

4 Observe

During our interviews, we briefly observed Christof Dorner from Readmill during his work. He showed us how he develops a feature: He picks a task from the bug and feature tracker *Trello*, creates a *git* branch locally, fixes the problem and writes tests for it. Eventually he opens a Pull Request on Github for the finished code to be reviewed and deployed. Here we gained the insight, that a frictionless and uninterrupted flow is highly valuable. This can be achieved by customizing and constantly improving the used tools. Christof mentioned how he improved his flow over time, e.g. making tests run instantly on each change or working with a decentralized source code management tool.

We also had the chance to observe the two current EPIC chair's Bachelor project teams. Both work with real customer data as well as existing legacy code. During observation we noticed that one issue they face is a disconnect between understanding the code in the editor and the data in the database. This leads to switching back and forth between the code editor and the *HANA Database Studio* as well as guessing values for variables in the queries.

Additionally to the interviews and observations, we took into account our own experience of developing applications with databases as well as incoorperated experiences from the seminar *Enterprise Application Programming Model Research 2012* at the HPI EPIC chair [1].

While investigating common workflows of software developers, we realized that they often follow the cycle depicted in Figure 3. Before entering that cycle, the software developer chooses the scope of a certain feature and does all the necessary work to have an actionable coding task at hand. The cycle is only started when the actual code generation begins. Depending on the programming language, the developer has to build the code or otherwise bring it into a testable state. In a next step he tests the changes made, e.g. through unit tests or by executing and manually testing the application. This cycle repeats, until the developer is satisfied with the results.

Once this cycle became apparent to us, we investigated the components it operates on. Especially from the experiences of the Bachelor's project teams let us realize, that, while in the editing step, the developer has no access to the data context in which the code will eventually be executed during the test

[1] `https://github.com/lauritzthamsen/infrarecord/`

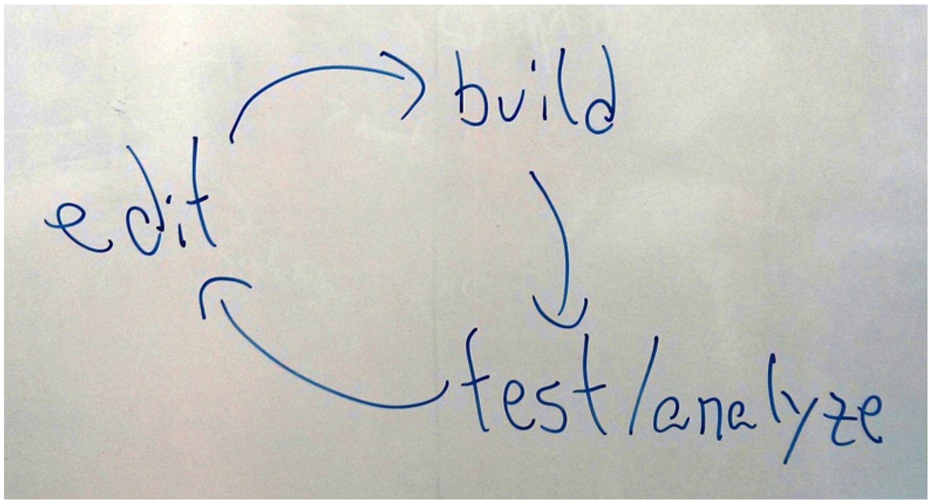


Fig. 3. The conventional code development cycle.

& analyze step. There is a disconnect between the editor and the application runtime. In the editing step the developer writes down his or her thoughts as code but only later on will it be interpreted by the machine. For the same reason the cycle exists in the first place. Because developers have to continuously test the assumptions they made about the behavior of their program. We learned, that every execution of the cycle is an interruption to the developer. Instead, we consider a cycle, which is so fluent that moving from one step to another becomes barely noticeable, the far better solution. One important requirement for this would that already during coding, the developer has access to the context the program will finally be executed in. When working with an external data source, the actual data can act as this valuable context.

Once aware of these conclusions, we developed our third insight, which also guided us through the rest of our project:

Code needs data context

5 Point of View

A point of view (POV) consists of three parts: a concrete specification of the user we are designing for, his or her needs we are trying to satisfy with our solution and finally an insight we derived from the understanding and observation phase. We present all of these in this section.

We derive a *"How might we..."*-question from our POV and try to answer this question during the ideation phase. This description presents our final Persona and her needs. Throughout the course we repeatedly adapted our Persona slightly as we discovered new needs and reevaluated their importance for our persona.

5.1 Persona

Our Persona is Amber, she is 31 years old and for roughly six years has she been working as a Software Developer at a big company that use HANA as a database for their applications. Within her team of ten developers, she programs applications that work on big data sets and are performance critical. While her main focus is on the backend side, she does not consider herself a database expert. Though she still knows how to write SQL queries, she is not able to predict the impact of certain changes in a query formulation on its performance just by looking at the statement. She works in her favorite development environment studio, which is Eclipse[2]. As a seperate environment, HANA Studio allows her to look at the content or schema of the database.

Amber enjoys coding a lot. It is the favorite part of her work. She is also very enthusiastic and wants to get features completed as fast as possible. She uses a lot of different tools that help her simplify reoccurring tasks and boost her productivity.

However, throughout her work day Amber usually experiences a lot of interruptions. We refer to these interruptions as *Context Switches* that pull her out of her workflow. These might be small switches, like changing from the editor to the browser to browse for something online but can also be bigger ones, such as having to write an email or even physical interruptions caused by colleagues who ask for her attention.

5.2 Needs

Amber's needs are manyfold. The essential ones are:

- She wants to do her work in the most efficient way.
- Since her applications are performance critical, she needs to evaluate her code against real customer data.
- Amber is not an SQL expert. Thus, she needs a way to explore and learn which parts of a query perform well and where bottlenecks are.
- She wants her productivity tool chain to be integrated into her IDE in order to avoid switching contexts.
- The general number of Context Switches needs to be minimized.

Throughout the seminar we down-weighted the importance of the last point as we narrowed down the interruptions that we wanted to tackle. We discovered that although big Context Switches like replying to emails lead to longer disruptions, they occur way less frequently than small Context Switches, such as navigating between applications. Thus tackling issues of the latter type may, accumulated, lead to a big increase in efficiency which is why we focused on that.

[2] `http://www.eclipse.org/`

Fig. 4. Our Persona Amber, a software developer for performance critical big data applications.

5.3 Insight

During our observations we saw how our Persona figured out to understand SQL queries written in code. She copied the query from her Eclipse editor into HANA Studio. As in Eclipse her query appears as multiple split *Strings* she has to escape all contained quotes. This is the first Context Switch. Most queries also contain parameters that are bound during runtime. In order to execute the query in HANA Studio Amber looks up appropriate values in the schema. This is the second Context Switch. If the executed query does not behave as expected she is forced to go over this whole process again, switching back and forth in the Studio between edit- and result-view mode. There is no way to recognize the impact of query changes on the result in the same window. Thus, this results in a number of additional Context Switches. If the query finally behaves as desired, Amber has to copy it back into her code editor and turn all concrete values back into variables.

We consider this interaction as a workaround and take it as a foundation for our Point Of View as it emphasizes the most important need. That is predicting the impact of queries on the underlying data. We combine our Persona and her need with one of the insights we gathered during our interviews, namely that *Code needs data context* (cf. Section 4).

5.4 POV Statement

User Amber, a productivity junkie, working with HANA on performance critical big data business applications.
Need Amber needs to predict the performance and logical impact of her code on the underlying data sets using only without switching to another application.
Insight Code needs data context.

Based on this, we derived the following *"How might we question..."*:

5.5 How Might We...

> *"How might we help Amber to predict the impact of her code on real customer data in one integrated development environment?"*

6 Ideation

Starting from our POV we held several rounds of brainstorming in which we came up with lots of savvy, silly, crazy and wild ideas. In this section we will present a very limited subset of these ideas. Out of this pool some attracted our attention more than others. The following section presents those approaches, that we, though not neccessariliy to the very end, thought through to a certain extent and thus believe to be potentially valuable.

6.1 Data Preview in the Editor

Our final *"How might we question..."* asks how we could help our Persona to understand the impact of her code on the corresponding data right from within the IDE. Looking at into existing debugging environments, we found that the debug mode usually allows for exploration of all variables with their data allocation. Thus, the impact code has on that data can be exposed for every instruction. Unfortunately, unless the debugger offers hot-debugging capabilities, it is impossible to change the allocation of a variable or the control flow of your program. Even in hot-debugging environments this feature is not used to write code. Given a runtime environment, which could be captured from test cases or previous executions, it is possible to bind parameters and states even in edit mode. Therefore, it is possible to show all the information available in debug mode at coding time. This would dramatically simplify the development cycle of *coding, build, test* (cf. Section 4) as errors can be detected far earlier in the process. A technique called *In-System Programming* allows to program some micro-controllers and other embedded devices under test [1]. We decided to take this concept to the next level by applying it to software development in general, and databases in particular. In our vision, our Persona would be be able to explore the results of a query for a pre-defined runtime environment simply by selecting it in the code editor.

6.2 Performance Evaluation in the Editor

Since our Persona is developing performance critical applications it is crucial for her applications to handle the big data sets they work on. As already mentioned, Amber is not an SQL performance expert in a way that is not intuitive for her to figure out the impact of the different operators on performance. To support her, one might think of a tool to instantly display performance values for different query formulations. To avoid Contexts Switches, this tools would be build right into her IDE. In our idea the tool displays the result set size and query execution time whenever you click on a query. Technically, it would execute the query against a real customer data set to ensure reliability of the results. Using a kind of time machine like view for different versions of the query, Amber can instantly see how changes in the SQL query affect the size of the result set and the time needed for it to be fetched. In our final solution we combine this idea with the previous one.

6.3 Code Generation in HANA Studio

HANA Studio is a tool which allows to interact with a HANA instance, execute queries and show the results. Our Persona switches to the Studio, whenever she is trying to fine-tune a query's performance. In her application there are a few lines of boiler plate code for each database access: Creating a connection to the database, getting a cursor, executing the statement and fetching and iterating over the results. In order to speed up this repeating process for Amber, we thought a tool to auto-generate code snippets within the Studio that already contained language specific code and that Amber could then copy into her IDE.

However, the boiler plate code is not always the exact same and might be tackled by DRY[3] application design. Additionally, due to the fact, that the Studio itself involves numerous Context Switches to change and view query results, and because code is naturally written in the IDE we decided not to further investigate this idea.

6.4 Making E-Mail more Expensive

During our first interviews we found out that many interruptions are cause by incoming emails. Even more, our people told us, that the many of these emails relate to questions the answers to which reside in knowledge management systems such as internal wikis. Apparently, the act of opening a mail program and composing an email seems to be much more convenient than searching through the different knowledge sources and reading all articles related to the question. As a result, many emails with redundant questions are sent, each of which causes someone to disrupt his or her workflow.

To address this issue we thought of ways to make email communication more expensive. Employees would get a budget of credits to send emails, each of which

[3] don't repeat yourself

defining its own price, depending on the usefulness of the question. Credits could be earned by answering emails and may be optionally turned into a monetary benefit. The goal of this system is to encourage people to search for a solution on the Internet or Intranet before bothering a co-worker with an unnecessary question. However, communication is the key to successful team work, which is why making email more expensive might not be well perceived by decision makers. An (automated) categorization of emails into "good" and "bad" communication might, however, be a tough one to implement.

Interestingly though, an IT services and consulting buyout firm from *Siemens*, has pioneered in this field by removing all internal emails until the end of 2013 in the context of their *Zero Email campaign* [3].

All in all we considered this idea to be too risky and instead chose to dig into other ideas which we assumed to have an even bigger impact on our Persona's efficiency. We still consider the approach as an interesting experiment.

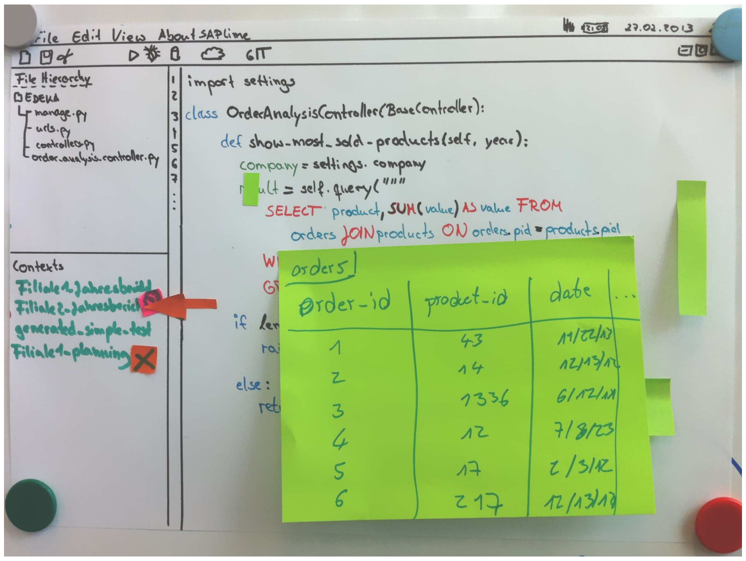
Our eventual idea is to help Amber understand the impact of her code on the underlying data without having to switch contexts. Technically, this would be an Eclipse plugin that allows her to preview query results and execution time right in her editor. Any adjustments she makes to a query, she can compare to information on previous query formulations. This allows her to directly realize the impact of her code while she is writing it.

7 Paper Prototype

After ideation we started prototyping to get user feedback on our ideas. We focused on our ideas regarding the editor enrichment in order to provide real data previews and performance feedback. We built an interactive prototype to test our overall design. For the reasons quick creation and low expenses as already pointed out in chapter 2 we decided to use a paper prototype. Utilizing *Post-Its* enabled us to manipulate its appearance and therefore added an suggested interactivity. Since we aim to improve developer's productivity this way of usability testing was crucial for our idea.

From a DIN A3 format cardboard we tinkered a representation of our conceived code editor. (Figure 5) It displays a piece of code in a *Python*-inspired programming language with embedded SQL statements. The editor features global syntax highlighting and, compared to conventional code editors, introduces a new panel in the lower left. The *Data Context Panel* lists all possible Data Contexts for this particular function.

For the user testing sessions we prepared several Post-its to simulate highlighting, cursor positioning and popping up windows. As shown in Figure 5, small red paper squares indicate either performance or functional problems in some of the Data Contexts. A small red arrow hints to the currently selected Data Context. The small green Post-it represents the cursor that the user can interact



```
import settings

class OrderAnalysisController(BaseController):
    def show_most_sold_products(self, year):
        company = settings.company
        r ult = self.query("""
            SELECT product, SUM(value) AS value FROM
                orders JOIN products ON orders.pid = products.pid
            W
            G

        if len
            rai

        else:
            ret
```

orders

order-id	product-id	date	...
1	43	11/22/13	
2	14	12/13/12	
3	1336	6/12/11	
4	12	7/8/23	
5	17	2/3/12	
6	217	12/13/13	

Fig. 5. Table Popup on our paper prototype after selecting the data table entity "orders."

with. The green stripes on the right mark the control flow that the execution of the currently selected data context covers.

When the user moves the cursor onto the identifier of a table used in the SQL query, a popup is presented that provides relevant information on this table. Figure 5 shows an example of these Post-it popups, when selecting the table *"orders"*. For different tables and Data Contexts we prepared several Post-its with empty tables, too large tables, and single rows. The latter could be attached in case the user inserted a new row.

For our testing scenario we marked one of the contexts as erroneous and another as a performance bottleneck. The test person was intended to inspect the several tables and examine the performance bottleneck or the program error respectively.

8 User Testing and Feedback

We were able to test our paper prototype with the students from both Bachelor's project teams of the HPI EPIC chair(Figure 6). Both teams develop applications based on SAP's HANA technology and also faced problems similar to the ones we had discovered during our interviews. One team investigates *In-Memory Data Management for Enhanced ERP* while the other focuses on *Real-time Analysis of Genome Data*.

Fig. 6. User testing with the Bachelor's project team HP2.

In the following we present their feedback as well as general insights we gained during the 5-day-workshop that was part of the seminar as well as from Hasso Plattner and some of the tutors.

8.1 Feedback from Bachelor's Project HP1

- The team generally welcomed the idea of popup windows, but criticized the fact that they occlude parts of the underlying code. Even if detachable ones that could be freely positioned on the screen would still lead to a cluttered workspace and thus result in an inferior workspace.
- For maximum productivity they rather prefer keyboard shortcuts instead of the mouse pointer to interact with the data entities in the code.
- Developers appreciate tidy screens, even on nowadays large monitors. Hence, the amount of provided information is to be treated with care.

8.2 Feedback from Bachelor's Project HP2

- The students find the concept of a Data Context hard to grasp when presented initially.
- They liked the immediate feedback in the Data Context Panel as well as the warning or hint symbols that appear next to a context, when some of their code changes resulted in unintended behavior on other contexts.
- They wished the Data Context Panel would also include all relevant test cases, so any additional testing tools to run their test suite would become expendable.

8.3 General Feedback during the 5-day-worshop

- The code as well as the provided data have to be as tangible as possible and the whole editor as easily navigable as todays mobile and tablet apps.
- Once the number of Data Contexts increases, some clever selection mechanism or at least a searching interface might become indispensable.
- In addition to whole tables, the inspection of partial queries would be desirable as well.

8.4 Feedback evaluation

As shown in Figure 7, we categorized the feedback into positive and negative, identified open questions as well as some new ideas worth to explore. Some of the ideas described in Chapter 6 originate from this feedback evaluation step.

For the next iteration of our prototype, we removed the table inspection popup windows and instead reserved a fixed panel at the bottom to display their content. Besides the user interface we added a new field of attention: investigating different mechanisms for handling large numbers of Data Contexts - regarding the acquiring as well as the relevance filtering and selection.

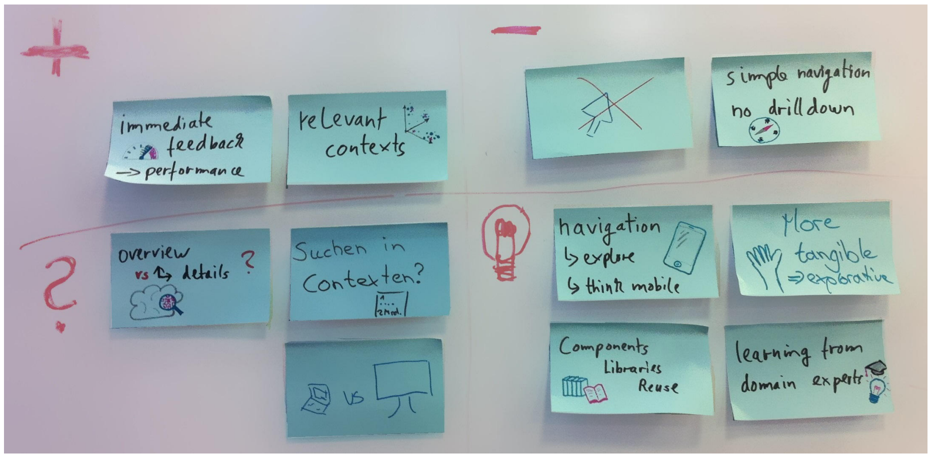
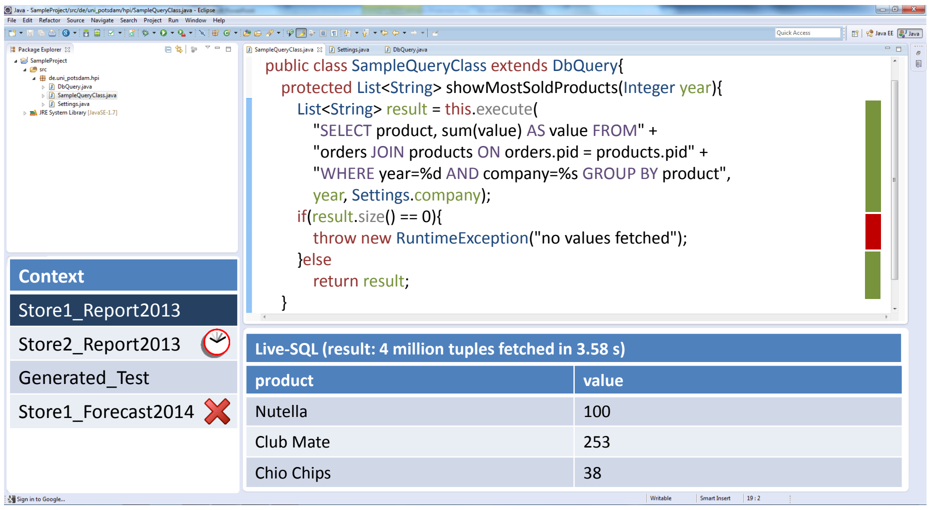

+
immediate feedback → performance
relevant contexts
−
simple navigation no drill down
?
overview vs details ?
Suchen in Contexten?
vs
navigation ↳ explore ↳ think mobile
More tangible ⇒ explorative
Components Libraries Reuse
learning from domain experts

Fig. 7. User feedback captured on Post-Its and arranged in the feedback panel with positive and negative feedback as well as questions and new ideas.

9 Final Prototype

Our final prototype, in which we included the feedback gained through user testing, is a mock-up of an Eclipse plugin that addresses the observed needs of software developers. Figure 8 illustrates a potential interface of our final revision. The UI consists of the following three major parts:



```
public class SampleQueryClass extends DbQuery{
  protected List<String> showMostSoldProducts(Integer year){
    List<String> result = this.execute(
      "SELECT product, sum(value) AS value FROM" +
      "orders JOIN products ON orders.pid = products.pid" +
      "WHERE year=%d AND company=%s GROUP BY product",
      year, Settings.company);
    if(result.size() == 0){
      throw new RuntimeException("no values fetched");
    }else
      return result;
  }
```

Context
Store1_Report2013
Store2_Report2013
Generated_Test
Store1_Forecast2014

Live-SQL (result: 4 million tuples fetched in 3.58 s)

product	value
Nutella	100
Club Mate	253
Chio Chips	38

Fig. 8. Our final prototype: an Eclipse plugin to display Data Context and query results at a glimpse.

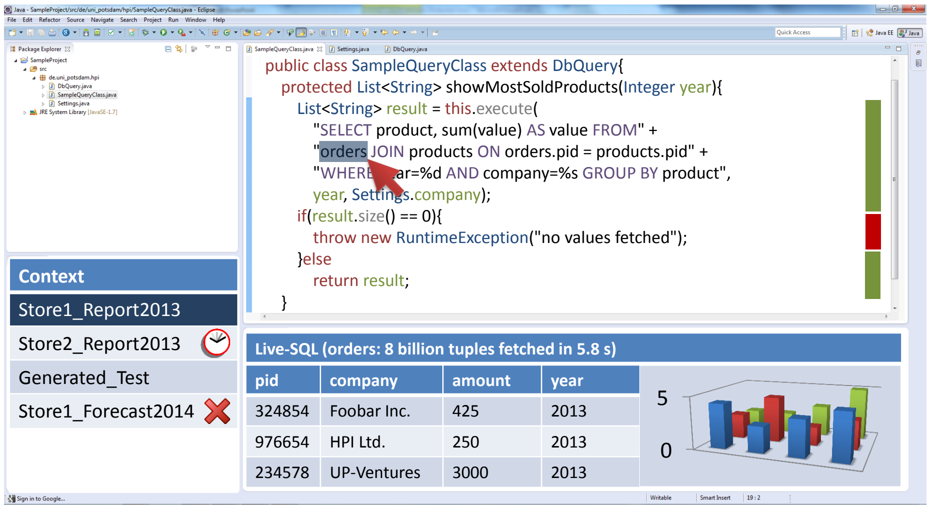
Code Editor Built on top of the standard Eclipse built-in editor, our editor adds some enhancements like syntax highlighting for inline SQL statements and instant indications for code coverage, which can be seen on the right side of the editor's pane. The latter visualizes the code branches that will be executed depending on the currently selected data context.

Live-SQL The frame at the bottom of the IDE enables the developer instant data access to the underlying database tables and views. It is also possible to explore the result sets of sub-queries. Furthermore, the data is fortified with runtime information and data heuristics.

Data Context Browser The developer navigates through different data contexts using the Data Context Browser. Some contexts ask for special attention from the programmer, e.g. those evoking errors or slowing down runtime, and are highlighted by an icon appearing next to their names.

In the following section we lay down the main features of the customized IDE and how the user can easily interact with this new environment.
In order to always provide the programmer with all available Data Context, our solution focuses on instant and easy access to the underlying database at any time. When the the developer selects a table, view, or sub-select in his SQL-statement our IDE responds with an immediate preview of the corresponding data, illustrated in Figure 9. Furthermore the system displays meaningful visualizations of the data to give hints on performance issues, e.g. the size and the fetch time of the relation. Note how this approach not only allows for a permanent connection between code and Data Contexts but also dispenses with any

workflow interruptions as they often appear in nowadays solutions. Therefore our solution allows for a highly comfortable development process.
In the Context Browser little icons highlight salient contexts, such as the context


```
public class SampleQueryClass extends DbQuery{
  protected List<String> showMostSoldProducts(Integer year){
    List<String> result = this.execute(
      "SELECT product, sum(value) AS value FROM" +
      "orders JOIN products ON orders.pid = products.pid" +
      "WHERE [illegible]ar=%d AND company=%s GROUP BY product",
      year, Settings.company);
    if(result.size() == 0){
      throw new RuntimeException("no values fetched");
    }else
      return result;
  }
```

Context

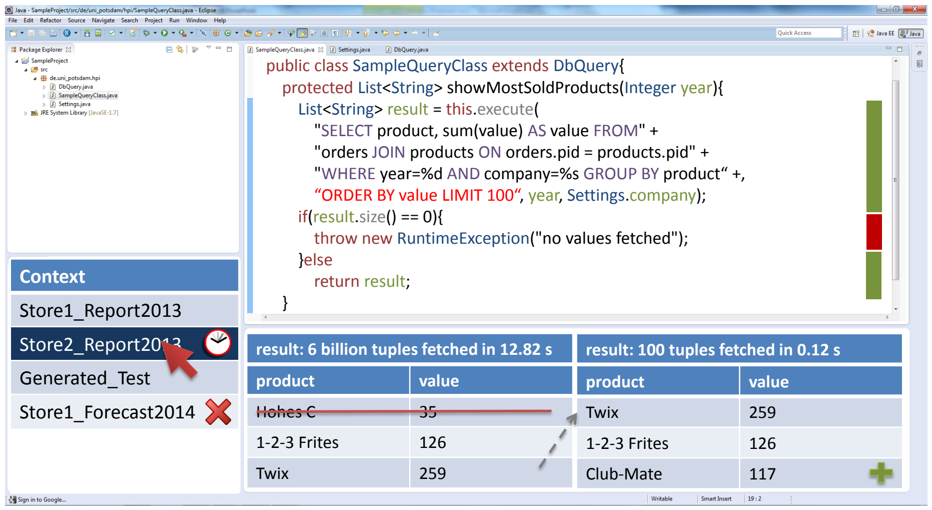
- Store1_Report2013
- Store2_Report2013
- Generated_Test
- Store1_Forecast2014

Live-SQL (orders: 8 billion tuples fetched in 5.8 s)

pid	company	amount	year
324854	Foobar Inc.	425	2013
976654	HPI Ltd.	250	2013
234578	UP-Ventures	3000	2013

Fig. 9. Instant data inspection of relations and sub-queries. Visualizations at the bottom help to grasp information quickly.

Store2_Report2013 in Figure 10. The little alarm clock behind the context name suggests that this context might perform comparatively slow during runtime and therefore reminds the developer to fix the issue. To investigate, the developer selects the context in the browser and gets immediate feedback in the Live-SQL pane at the bottom, where he realizes that the result relation takes relatively long time to be fetched. As a response he now could adapt the SQL-statement towards the output intended by the overlying method (see the red highlighted code in Figure 10). After the developer has changed the code in the editor pane, the Live-SQL viewer displays a *diff* to the result relation before the code change. If the adjusted query decreases its fetch time by a high enough amount, the clock symbol in the Context Browser will disappear and the issue is solved.
Due to the fact that a Data Context in our proposed system is not only a database state but also presents runtime information of the code, we are also able to warn the developer about error states his software might run into for certain input combinations. Such an exceptional runtime context is displayed in Figure 11, namely *Store1_Forecast2014*. Here, the result set is empty, which directly influences the surrounding code of the query. To inform the user, the covered branches are marked differently than in the previous examples. Because of the zero fetched rows a runtime error is thrown in the method, indicated by the red cross icon in the Context Browser. The variable that causes this



```
public class SampleQueryClass extends DbQuery{
   protected List<String> showMostSoldProducts(Integer year){
      List<String> result = this.execute(
         "SELECT product, sum(value) AS value FROM" +
         "orders JOIN products ON orders.pid = products.pid" +
         "WHERE year=%d AND company=%s GROUP BY product" +,
         "ORDER BY value LIMIT 100", year, Settings.company);
      if(result.size() == 0){
         throw new RuntimeException("no values fetched");
      }else
         return result;
   }
```

result: 6 billion tuples fetched in 12.82 s

product	value
~~Hohes C~~	~~35~~
1-2-3 Frites	126
Twix	259

result: 100 tuples fetched in 0.12 s

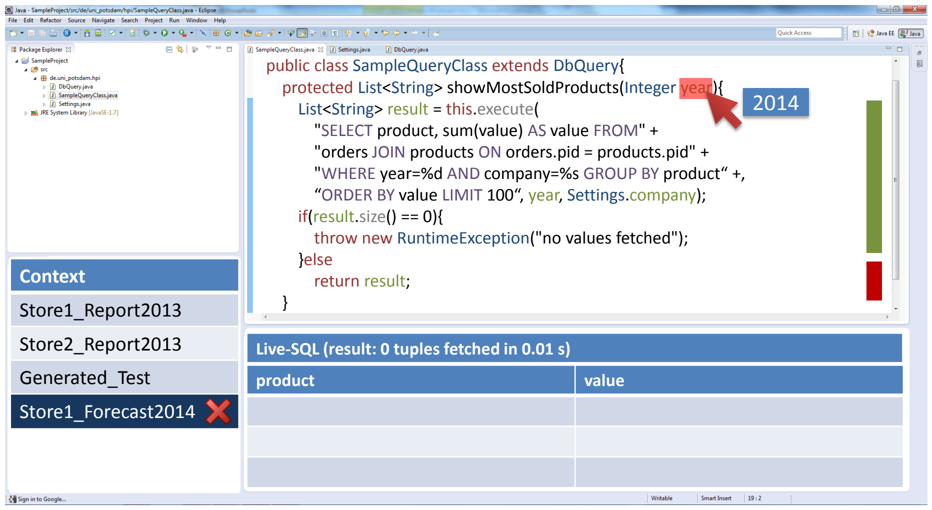
product	value
Twix	259
1-2-3 Frites	126
Club-Mate	117

Fig. 10. An alarm clock symbol indicates poor performance for Data Contexts.

faulty branch execution is the method's parameter *year*, made obvious by red background coloring for this word. Hovering over this variable provides the programmer with the actual value to support him or her understand the reason that caused the issue. Going from here, the developer can alternatively adapt the value of the input variable or mock the result data by inserting tuples in the Live-SQL tab for this particular result set, which would force another branch execution.

10 Implementation Ideas

As described in Section 7 and 9, our proposed environment always presents relevant Data Contexts when the developer interacts with a statement. Just like in a debugger, for example when the developer, hovers over an identifier in the application code, the IDE responds with actual data, either objects or database rows. However, in contrast to ordinary debugging environments, a developer should not only be able to view the current state and step through the execution, but also add and evaluate statements. This allows for an interactive style of development centered around seeing the impact and effects of code and queries immediately. In order for such features to function properly, availability of runtime data needs to be ensured. First, a developer can only see values and database entries for identifiers, if such data is available. Second, statements that might or might not embed queries require referred variables and potentially even a database to run. A realization of our idea would, therefore, implement components that gather



```
public class SampleQueryClass extends DbQuery{
  protected List<String> showMostSoldProducts(Integer year){
    List<String> result = this.execute(
      "SELECT product, sum(value) AS value FROM" +
      "orders JOIN products ON orders.pid = products.pid" +
      "WHERE year=%d AND company=%s GROUP BY product" +,
      "ORDER BY value LIMIT 100", year, Settings.company);
    if(result.size() == 0){
      throw new RuntimeException("no values fetched");
    }else
      return result;
  }
```

Context
Store1_Report2013
Store2_Report2013
Generated_Test
Store1_Forecast2014 ❌

Live-SQL (result: 0 tuples fetched in 0.01 s)

product	value

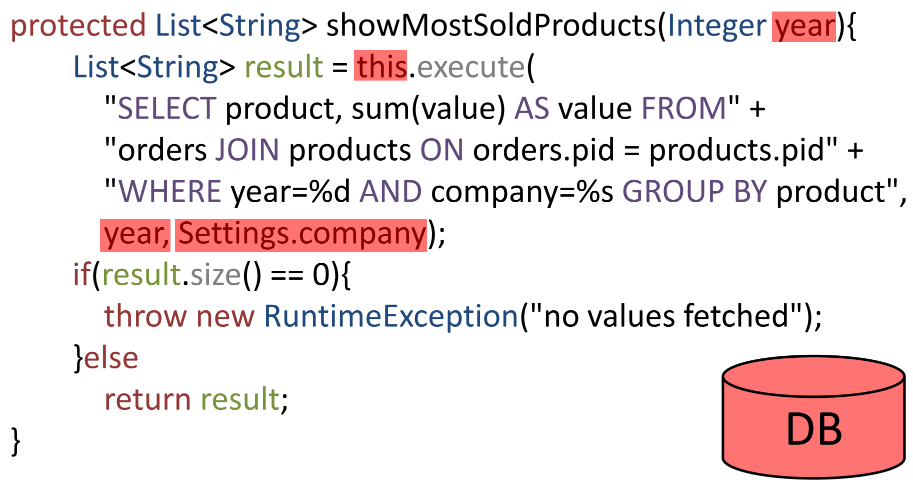
Fig. 11. Error sources and code coverage are highlighted at runtime.

the necessary data contexts from actual program runs as well as components that support developers in selecting interesting data contexts during development.

10.1 Gathering Data Contexts

Our approach depends on the availability of Data Contexts for the code sections of interest to the developer. Such Data Contexts need to be realistic and immediately available to be useful. Given these premises, we think of two approaches to provide Data Contexts for specific code sections: Either deterministic test runs establish Data Contexts on demand or traces of the application's execution record Data Contexts in advance. Running tests to provide the runtime during development would only require to store coverage information that associates code sections with test cases. Recording contextual information in advance would, however, imply storing as much data as necessary to actually run any statements in the sources of an application. The time necessary for test execution depends on the granularity of the covering tests with a full spectrum from fine-grained, isolated unit tests to high-level, long-running acceptance tests. Previously recorded Data Contexts, however, need just to be fetched from a database. This could also be an in-memory database and thus potentially provide faster access to Data Contexts, compared to establishing runtime data through running tests. Fetching recorded Data Contexts is further independent of the availability. It could be captured in deployed systems used by real customers, which would also provide more realistic runtime data.

Assuming deterministic code, a reasonable tradeoff between space and time demands would be to record data not on the level of single statements, but on the level of methods. A potential implementation would record all data necessary to actually run a method with all its statements. Such data include provided parameters, accessible state, and the database at the moment methods are entered, as shown in Figure 12.



```
protected List<String> showMostSoldProducts(Integer year){
    List<String> result = this.execute(
      "SELECT product, sum(value) AS value FROM" +
      "orders JOIN products ON orders.pid = products.pid" +
      "WHERE year=%d AND company=%s GROUP BY product",
      year, Settings.company);
    if(result.size() == 0){
      throw new RuntimeException("no values fetched");
    }else
      return result;
}
```

Fig. 12. Running the statements of this Java method with its embedded SQL query requires the data highlighted in red: the parameter, accessible state, and the database.

When a developer interacts with a specific line in the method's body, one idea would be the use of ordinary breakpoints to establish the necessary runtime data for that specific line of interest. Accessible state includes global state and, depending on the used programming language, also state from surrounding scopes, e.g. from surrounding functions. Snapshots of the database are necessary as methods that embed queries might modify the database, possibly impeding subsequent runs of that method during the interactive development.

We assume the recording approach and method granularity for the remainder of this document.

10.2 Selecting Data Contexts

The presented tracing approach generates potentially numerous Data Contexts for each method. Given our tool traces live systems deployed at customers, each user interaction might lead to recorded data contexts for all triggered executions. Further, even when our tool only traces deterministic test runs once, particular methods might still be covered through multiple test cases or even be called multiple times in single test cases as, for example, the case with methods called from loops. For these reasons, a developer might be confronted with many Data

Contexts for each statement of interest, when using an implementation of our approach. Additionally, since Data Contexts from the same trace can be expected to considerably overlap, presenting all of them to the developer might significantly reduce the simplicity, we tried to achieve with our prototype. Potential approaches to address this issue include automatic selection of interesting samples after clustering all Data Contexts, preselection based on relevance through expert users, or combinations of both methods. Clustering data contexts could be based on several dimensions, including:

– size of the associated database snapshot
– control flow that lead to this context
– timestamp of this context

Nevertheless, even supporting developers with automatic clustering and stored preselections, exploration of the full extent of available contexts would also be desired, in order to make a final decision on which particular Data Context to use during development.

11 Open Questions

In the future, we would like to investigate how seamless and helpful debugging of complete and partial SQL queries could be realized and integrated with our idea. Whether and how actual hardware configurations could be considered as part of the Data Contexts from runtime, and how data contexts could be stored efficiently.

11.1 Query Debugging

Query debugging should be seamless. The tool should provide immediate feedback while understanding and developing both application code and database queries. Thus, a developer should not be required to use two tools to debug code. The goal must be to dispense with the approach of copying complete or parts of embedded queries from application code to active database sessions. Instead, we envision an integrated debugger that enables to step through and evaluate both application code and embedded database queries without forcing the developer into context switches, such as changing user interactions or interfaces. It should present return values, show basic profiling information, and allow to explore involved data.

Developers might also benefit from exploring the parts of queries. However, as parts of SQL queries cannot necessarily be evaluated independently as they might depend on other parts of the queries, developers potentially need to select multiple associated parts in conjunction. For example, given the following query, the `WHERE` as well as the `SELECT` clause both rely on the `JOIN` in Line 2 of this example.

```
SELECT products.name, SUM (orders.price) as pricemake
FROM orders JOIN products ON orders.pid=products.pid
WHERE orders.year = 2012 and products.manufacturer = "HPI"
GROUP BY product
```

An ideal Data Context debugger would aid developers also in selecting combinations of automatically identified meaningful parts of database queries.

11.2 Hardware Contexts

Our idea focuses on showing actual results and basic profiling information right next to code and queries. Assuming deadlock-free deterministic code, the results of queries should be independent of the underlying hardware. However, the profiling information might depend significantly on the executing hardware. Therefore, our approach would benefit from incorporating the impact of different hardware configurations. Such hardware contexts should, thus, also be part of the data recorded at runtime. We would focus further efforts in this directions on the capturing, storing, and application of hardware contexts. Such hardware contexts could be used to either run statements on the associated hardware or to simulate the impact somehow.

A related question is whether Data Contexts should be tied to particular hardware contexts during our tracing. Developers could gain insights from combining Data Contexts and hardware contexts freely to see how their application performs under different circumstances. However, bundling both contexts emphasizes optimizing performance for real usage.

11.3 Context Representation

For any given code section our approach potentially results in many similar Data Contexts. Storing those could probably exploit these similarities. An implementation could identify similar Data Contexts, generate common base contexts, and for each particular context store only the differences to the bases.

As Data Contexts from the same traces most likely overlap, marking them with an identifier to indicate the used trace might allow to identify similar Data Contexts faster.

Further, the efficiency of storing Data Contexts could probably also leverage advanced database features as, for example, *SAP HANA's Time Travel* feature [4], which allows to reset the database to a specific moment in time. Given this feature, an implementation would need to store only timestamps to establish the database part of Data Contexts instead of actual database snapshots.

[4] `http://help.sap.com/hana/html/sql_create_table_history_time_travel.html`, retrieved March 12, 2013

12 Conclusion

This document presents our final prototype of a Eclipse plugin that supports software developers in their workflow. Using SAP's HANA our tool allows programmers to instantly investigate the data that they write code for as well as focus on their tasks from one integrated environment. Our prototype integrates well in the Eclipse environment, giving the developer information about runtime, relevant and real Data Contexts as well as potential upcoming issues at a glimpse. The programmer benefits from these information as they are presented earlier and quicker than in a traditional development cycle. As a major challenge regarding a technical implementation of our prototype we see the filtering and clustering to import particular relevant Data Contexts into the environment. However, we see this feature as an essential part of our solution, as it enables programmers to not only work more time efficient but also to ensure the quality of their code. Once working, the feature enables the developer to even interact with the on demand result sets, e.g. by altering result relations or switching between contexts, to investigate the influence on the program flow. In the final chapter of this document we give insights of what considerations to make when implementing this feature.

Our prototype reflects the needs of our Persona Amber, that we developed throughout the seminar. The understanding and observing phases of the design thinking process helped us to quickly dig into the problem domain and define these needs. In the further phases of ideation, prototyping, and testing we were able to integrate the repeatedly gained feedback and new insights in a final solution concept.

References

1. Herrmann, A.L., Southgate, T.J.: Apparatus and method for in-system programming of integrated circuits containing programmable elements (Jun 18 2002), uS Patent 6,408,432
2. Leifer, L., Plattner, H., service), S.O.: Design Thinking. Springer-Verlag Berlin Heidelberg,, Berlin, Heidelberg : (2011), `http://dx.doi.org/10.1007/978-3-642-13757-0`
3. S.A., A.: Zero email (Mar 2013), `http://atos.net/en-us/about_us/zero_email/default.htm`