Abstract. The year is 1993, and I give my very first talk at a big software engineering conference. Right in the middle of my example, a professor stands up and exclaims with a mocking smile “To me, this looks like a solution looking for a problem!” The audience erupts in laughter, and my advisor sits in the first row, grinning. How would I get out of there? And why would this experience shape all of my career from now? Telling three stories around three conference events, I unfold lessons on impact in software engineering research: Do relevant work – strive for simplicity – keep on innovating.

Thank you very much, everyone. I know this has been a great conference, but now you’re eager to get to get to your planes, to get back to your offices, to get back to friends and family. So in the next couple of minutes,
I am going to restrict myself to three short stories. Not more, not less. All three of them are connected to some conference talk, so I guess I'm in the right place to share them.

All these stories revolve around talks at conferences, and here's my first one, almost 25 years ago. This is in 1993 at the German national conference for Software Engineering, in Dortmund, Germany. Anyone from Dortmund, here? At this time, I am a PhD student, and this
my first talk ever

I am presenting an experimental programming environment called NORA. NORA stands for no real acronym, so it's pretty generic, but what this is about is actually...
one of the first uses of theorem provers in Software Engineering.

My own topic would be configuration management with feature logic, using features to represent variability and changes.

Our key example that day, however, would be component search.
Find a sorting function

The idea is that you'd have a huge library of components, and you'd be able to find a sorting function

by specifying the pre- and postconditions of the function you're searching for.

∀i, j: i < j ⇒ a'[i] ≤ a'[j]

So, here's the postcondition. You want the resulting array a' to be sorted.
∀i, j: i < j ⇒ a'[i] ≤ a'[j] \\
∧ ∀x ∈ a ∪ a': |{i: x = a[i]}| = |{j: x = a'[j]}|
"You know, to me this looks like a solution looking for a problem"

This closes it. I am done; I go through the remaining slides, but nobody listens anymore, and for the rest of the day, there’s people laughing and pointing when they see me, and I am eager to get the train home. All the way back, I am still enraged.
So that was the story of my first talk. Is the story over yet? Not quite. There’s a couple of ways I can spin the remainder of the story.

I could tell how after being utterly devastated, I finally managed to find my path, and still make a great career in computer science. Guys, girls – don’t listen to what old white farts say, follow your dreams, and in the end, you’ll get married and have many children tenured and have many papers accepted.

I could also spin this from the angle of how important and ubiquitous theorem provers are today, how all of verification, testing, analysis depends on constraint solvers, model checkers, you name it. We were among the first, and today, I am right.
But the spin I'd like to give this story is yet another one, namely the question of relevance. Today, when I think back of this story,

It turns out that these guys shouting into my talk were right all along. Think of how programmers work when they search for some function.

They go to Google
They go to StackOverflow

StackOverflow

So much of programming is searching today. It is "grep sort" everywhere.

grep sort is everywhere

So, the essence of the story is that they were right all along – developers want simple tools that work, not some made-up formalism that only PhD students understand.

They were right
One important consequence for me was that I started a sideline these days, together with a student of mine, Dorothea Lütkehaus. Already in my master’s thesis, I had built a library that could visualize data structures. We thought of building this into a debugger, and built a tool, called DDD.
And at the top, you can see DDD nicely visualizing a linked list.

Now, it turned out that DDD was among the first debuggers with a decent graphical user interface. People loved it.

It became a GNU program – I got a nice letter by Richard Stallman –
– and developers from all over the world sent in postcards to thank us for making it available.

These postcards at the time were far more important to me than citations I would get. People were actually using my stuff.

This is because DDD was a tool that would get things done, with immediate usefulness.
A concrete benefit, not just some abstract concept that may or may not be adopted.

The key metric here is usefulness. DDD was clearly useful. And this usefulness was that made it better than the state of practice. Usefulness is the key metric in Software Engineering, so this experience prompted me to ask questions like
Is my research useful?

And for whom?

What do developers need?

What is it that developers – our customers, our key audience – actually need?
where I was

where I should be

@AndreasZeller

So here I was with my research, feature logic, theorem provers – but I felt out of place.

What is it that developers actually need? We can ask them.

* Do people ever write loop invariants?
* Does it help?
* How do we measure the productivity of our engineers?
* How do users typically use my application?

There’s this extremely nice survey by Andy Begel and Tom Zimmermann at Microsoft from ICSE 2014, including questions such as the above
Now, the word "recommender" does not occur in that paper.

Nor does the word "model" occur.

Nor "repair".
Remember this slide, when I had my doubts?

I think that given the number of problems we are facing today – or still facing after all these years – we still are very much where the light is bright, where we know our strengths. Yet, maybe, we should venture out a bit more into the darkness. Talk to developers, talk to industry, find out where the real challenges are – and face them.

But when talking to developers, do not assume they will change anything because of you. They will not adopt your formal method just because you say so.
Make sure that they can adopt your approach with minimal effort. And pave a way toward this transition.

And here, I am not sure whether our paper-centric culture is the best way to achieve this. You are literally asking the reader to rebuild everything you describe.

Actually, I think that tools are a much better way to achieve impact and relevance.
And even better, I think that we should go and bring both together. I'd like to show an example.

You may have heard of the Jupyter Notebook.

And if you haven't – they just got the software system award from ACM.
With Jupyter, you have a mix of text and code; you can edit both, and you can execute the code to immediately see the results of your actions. You have math typesetting, you have plots, you even have interactive elements such as sliders. All updated in real time.

Now, this is great for teaching, for students, for instructors. But if this were a scientific paper, what would it mean?

You could have small programs that embody the gist of your algorithm.

Small programs that can be examined
can be experimented with

that can be assessed and experimented with

can be taught

that can be taught

can be used

that can be *used* by others –
used

and reused.

reused

You'd have both: the tool and the paper.

tool and paper
Actually, why still have paper?

That was my first story – on relevance.

My second story is on simplicity. Six years later, it is 1999.

Three Stories

September 9, 1999
And we are at ESEC/FSE, Toulouse, France

PhD on version control

I have completed my PhD on version control and the experience with DDD had raised my interest in debugging.
So I had come up with an idea that combines both: Version control and debugging.

The core idea of delta debugging is very simple. You have a big set of possible influences (here's one big set of things), and in there, there's a small set that causes what you're looking for.

You can test, though, whether what you're searching for is in the set. So you try out one half.
And another half. Turns out the cause is in here, so you keep it.

You repeat the process. Remove one half – hey, the effect is still there.

Again
And again

Turns out that now, it's in the other half

You keep on narrowing
And narrowing

And narrowing further. That's a bit small, right?
Okay, we'll go and enlarge things.

This is what a process like delta debugging finds – the small subset that causes the bug. Can be in your input, in your version history, in your configuration.
And it’s typically a very tiny element or difference in a big, big set.
AndreasZeller

And if you have, say 2,000 lines of
nroff input, it will reduce these to just
two characters

AndreasZeller

So, this is what I presented in Toulouse in
1999, and it was very well received, big
applause and all. But after the talk, right
as I get out of the room, there's a senior
professor from France who is very
agitated. He shouts at me (with French
accent)

AndreasZeller

"I would never hav sought

AndreasZeller

“I would never have thought
that something so simple

could be accepted at a scientific conference"

Yeah. Here we were. How could I continue this story?
Intellectual superiority

Maybe on how the scientific styles differ from country to country. If I wanted to impress my audience with my intellectual prowess, filling the talk with formulas and special terms such that nobody can follow and everyone recognizes my superiority (I hear they do this in France), well, then delta debugging would not be it.

Impostor syndrome

I could also talk on how this raised doubts in me on whether I’d done the right thing. Anybody could have come up with this! How did I deserve to be called a scientist? And how does the audience not see I am a fraud?
However, the way I'd like to spin the story here is simplicity.

Remember: "something so simple".

What's the alternative to simplicity? Well, complexity. And complexity
complexity is our enemy

Is our enemy.

control complexity

which we have to control

As put forward in this wonderful quote by Pamela Zave

The purpose of software engineering is to control complexity, not to create it.”

- Pamela Zave
Making complex things simpler

So let this be the essence of SE. and, by the way, of delta debugging.

And I’d like to point out that it had taken me one year to make delta debugging as simple as it was

So making things simple is hard work

simple = hard
But simplicity makes all of our lives much better.

And by the way, while praising simplicity, I’d like to take the opportunity to honor Andy Ko and Brad Myers, whose approach to debugging is for me the epitome of simplicity.

But then, such simplicity is hard to find.
A few years ago, I visited a high-profile graduate school. One of the best universities in the country, extremely selective, extremely ambitious. So there's 20, 25 students in the room, and they tell me they are expected to publish one paper per year. But not anywhere, but not anywhere – at ICSE, FSE, ASE.
one year making things simple

Now remember: I spent one year refining delta debugging. I don’t think I had a paper in 97 or 98.

But then, fortunately, it turns out that getting a paper accepted is easy. All you need is a recipe – for doing research that will get accepted. One such recipe is...
to take a simple approach

a simple approach

say, something we use every day. A napkin, for instance.


You then add some increment to it. Make it more automated. Say, add machine learning.
Make it dependent on context, such that it will work well in that context. If not, cut it off.

Integrate all this into the user's environment. Just continue adding and adding

Until it gets better. Say, 2% more precision. 5% more area under the curve. Errors found.
Reviewers get this

This is so great, because even if reviewers do not understand your approach at all, they will understand the improvement.

What you then have is a complex approach that is better

But then, is this really the case? With such complexity, who wants to re-implement your approach? Who wants to use it?
The purpose of software engineering is to control complexity, not to create it.”

– Pamela Zave

Maybe it is time to apply our principles to our own research.

There’s more such recipes, of course; and you may argue: So what? Who cares about a paper too complex getting in? Well, the problem is that such papers obstruct the scientific progress – because the only way to get even better results.
more complex = even better

is to build something that is even more complex. This helps nobody.

So we end up with contraptions that are more and more complex, and yes! They automate something! And yes! They are better than manual work! Yet, this helps nobody.


And I wonder: Would something so simple
as delta debugging

get accepted at this scientific conference – today?

Okay, we’re short on time, so let me close with the third story.

Three Stories
Again, five years later. It is a Saturday morning in 2004,

and it is the day of the ICSE deadline. You know ICSE deadlines, right?

I'm a tenured professor,

professor since three years
full professor

a position which I got through delta debugging and DDD

a new thing: I now have students.
Here's one. You know that guy? It's Tom Zimmermann.

With Tom, we systematically analyze version histories.

Specifically, we look for co-changes,
People who changed A also changed B

That is, changes involving multiple components at once.

We find relationships. For instance, whenever someone changes file.c, the file schema.jpg is also changed. Why is that? Turns out file.c has an embedded SQL statement, and schema.jpg is a picture of the database schema. When the schema changes, so does the SQL statement. Find that, static analysis!

We can go and recommend changes.
Change A, you also need to change B

src/file.c \( \rightarrow \) doc/schema.jpg

And we struggle with accuracy metrics such as precision and recall, which are all new to us. (As also for the SE community)

So it is the Saturday of the deadline; deadline is around noon in Europe;

September 23, 2004  
ICSE Technical Papers Deadline
And we are still struggling with these metrics. This is when Tom calls in at 10am – two hours before the deadline.

He says he has found a way to boost precision and recall above 90%. And I tell him, this is great, but this sounds too good to be true, so please check and re-check.

One hour later, one hour before the deadline, he finds he has accidentally trained from the testing set. So, we're back to our old values, and we submit.
The reviews are mixed. The reviewers clearly don't know what to do with this

but accept anyway

The paper title is "Mining version histories to guide software changes"
Today, it has more than 1200 citations

Three years ago, Tom, Stefan, Peter, and I got the most influential paper award.

So again, how do I spin this story? I could tell something about
Quality assurance in research

how important it is to do thorough quality assurance, how to ensure your results are reproducible and all, and yes, it is.

We were so lucky

I could also spin how lucky we were, as Gail Murphy and her student Annie Ying were working on exactly the same topic, with the same results, but decided not to go for ICSE because they wanted better precision and recall. Luck is the most important factor for success.

Innovation

But the lesson to be learned from this, for me, is innovation. Actually, our concerns about
numbers

did not matter.

simply because there was nothing to compare against.
Our approach was entirely new, finding things that no-one else did.

It was new, and new was better.

Going from debugging to mining software archives was one step towards something new, and I have kept on moving since then, exploring dozens of new fields – sometimes successful, sometimes not so – but always learning, always progressing.
And that’s not me. That’s me and many great students, whom I admire and love very much.

And work that would be simple and have impact in practice

* Delta debugging narrows down failure causes
* Mining software archives yields empirical findings
* Grammar-based fuzzing tests JavaScript interpreters in all browsers
* Apps are checked against descriptions and categories (at Google/Microsoft)
But we’d also need patience

entering a new area = 1–2 years

Because if you enter a new area, it takes a year at least to understand how it works

getting cited = many years

And if you have something really new, it can take many years until it gets cited.
I was very glad I had an environment that would trust me:

a trusting environment

the Saarland Informatics Campus in Saarbrücken, Germany

Saarland Informatics Campus

When I got hired, I was the one candidate with the lowest number of papers.

the hire with lowest # of papers
Nobody ever checked my publication counts.

The only thing that matters, they told me, will be your impact.

Even if it takes years or decades to build.
They took enormous risks, they put in an enormous trust. They trusted me all this time –

and here I am today. Thank you so much.

(short pause)
In this moment, as I am standing here, I realize how lucky I was, again and again. Most of us have to struggle hard in our daily work, trying to fulfill the most absurd incentives and regulations.

[Source: https://en.wikipedia.org/wiki/Sisyphus]

I missed almost all of this. I was lucky, and my luck is why I am standing here.

Survivor bias

But whether you are lucky or not – these hold for all of us:
If you are in the light, go explore the dark
search for *relevant* problems

Find out what is relevant

search for *simple* solutions

Find out simple solutions

keep on innovating

Keep on innovating
As the Romans say, "sapere aude": Dare to think for yourself, dare to be wise;

Or simply: Dare to know.

That's it folks – three stories, three takeaways
Three Stories
Three Takeaways

• Make sure your research is relevant
• Talk to practitioners for real problems
• Make results actionable, usable, assessable
• Useful = better!

• Always search for the simplest solution
• Complexity prevents adoption, teaching, progress
• Be aware of complexity recipes
• Simple = better!

• Keep on learning, keep on innovating
• Aim and have others aim for long-term incentives
• Dare to know!

Now go out and create masterpieces of Software Engineering – and see you next year in Montreal!