This is the talk “Empirical Software Engineering 2.0”, given by Andreas Zeller on 2007-05-19 at the workshop on Mining Software Repositories (MSR). Since the slides are not self-contained, I have compiled these notes after the talk; they provide a summary of what I actually said. (Text in italic, like this one, is extra comments.) I hope these notes help in capturing the talk. So, sit back and enjoy the show! - AZ

It’s always nice to give a keynote talk. In contrast to a regular talk, no one reviews or approves whatever you present beforehand. Plus, you have plenty of time – you can talk about whatever you want, even if it has nothing to do with the venue.

So let’s simply start with a Pop quiz: What’s the difference between a cheese and a waterfall?

All photos in this talk are copyright (c) iStockPhoto.com. Do not reproduce; buy them for $2 a piece instead.
Okay, just kidding. Here’s my talk. It’s called “Empirical Software Engineering”, which most of you of course know, and the extra twist in this case is the “2.0”, meaning a new revision.

The term “Empirical Software Engineering 2.0” was coined by Tom Zimmermann, after lots of discussion on the future of programming environments. The vision itself is my own, and I take full responsibility for both term and vision.

Empirical SE addresses classical SE questions, which are at the heart of software engineering. How can we produce high quality at low cost?

Normally, I avoid bullet points whenever possible. Here, I intentionally used bullet points here to make the slide more “boring”, thus illustrating the “boring” characteristics. Later on, I refrain from bullet points entirely to make the vision more graphic.

There is much belief in SE about how things should be done, but still pretty little knowledge. In order to address this situation, knowledge is needed - and such knowledge comes from empirical studies. The movement towards empirical studies is pretty recent, and we can only hope it will be successful.
Your typical empirical SE paper follows four steps. Its result...

<table>
<thead>
<tr>
<th>Measure data</th>
<th>Build model that explains the data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use model for predictions</td>
<td>Test predictive power</td>
</tr>
</tbody>
</table>

... typically looks like this. You collect a set of data, then you create a model (which typically is a curve – in this case a Rayleigh Manpower Curve, for instance), and then you can use the model (= the curve) for predictions. If you know the effort during your project is distributed along this curve, you start the project, take the first few datapoints, and then, you can fit the curve along these initial datapoints, and you get a prediction for the effort distribution in the remainder of the project.

Predicting Maintainability

\[
\text{Maintainability} = 171 - 5.2 \ln(\bar{V}) - 0.23 \bar{V}(G) - 16.2 \ln(L) + 50 \sin(\sqrt{2.4C})
\]

This formula even uses a sine function to make the curve fit as good as possible. The fun thing about it is that the percentage of comment lines never exceeds 100%, or 1.0, and therefore, you only get half of the first period of the sine. In other words, no comments is bad, 100% comments is bad, and in the middle, you get the highest maintainability.

This slide is one of my all-time favourites.
The dirty story about this data is that it is frequently collected manually. In fact, the company phone book is among the most important tools of an empirical software engineering researchers. One would phone one developer after the other, and question them – say, “what was your effort”, or “how often did you test module ‘foo’?”, and tick in the appropriate form. In other words, data is scarce, and as it is being collected from humans after the fact, is prone to errors, and prone to bias.

This is why the ground data has to be carefully cleaned up (handling outliers, for instance) and why the models have to be very elaborate. All of this takes time and effort, which means that the results typically reflect the state as it was six months before. Is it still appropriate at this time?

All of this actually reminds me of – cheese. I should say that I am very fond of cheese (given my weight, I should probably cut down); but there are a few striking analogies.
It takes 30 pounds of milk to produce one pound of cheese; the collection effort is high. Second, a good Emmental takes 18 months to ripe; just as these studies take time for the data modeling. Third, the data has lots of holes which the model needs to compensate. Fourth and last, cheese does not last for a long time; you better consume it soon.

You may have heard of Web 2.0, the buzzword of the year. The main feature is user-generated content – that is, rather than you collecting content, your users provide it for you (and even better, for themselves). There are many more features of Web 2.0, all listed on this slide.

SVG graphics taken and adapted from wikipedia.org; see “Web 2.0”

Some of the features are features that I would also like to see in Empirical SE. In particular, the idea of user-generated content, but also data-driven approaches, usability, recommendation systems, etc.
And what I propose in this talk is a concept that will actually introduce all these features into empirical software engineering – a concept that is revolutionary enough to warrant a major version increase – in other words, Empirical Software Engineering 2.0.

Empirical Software Engineering 2.0 is supposed to replace Empirical Software Engineering 1.0. So I want to replace this…

…by this. This, of course, is a waterfall; Niagara falls, to be precise. This is my picture of Empirical Software Engineering 2.0, and there sure are big differences between these two.
In Empirical SE 2.0, you don’t need to collect the data – the data comes to you. You don’t need to wait months for study results – you can get results at the touch of a button. There are no holes, no biased interpretation – all data is valid and real. And: your results reflect the current state of your project as it stands.

The key to get there is to get rid of manual data collection and interpretation. I want to replace this…

…by this: A software archive which contains all the project data, all the activities during the project.
Such software archives are being used in practice all the time. If you file a bug, for instance, the report is stored in a bug database, and the resulting fix is stored in the version archive.

These databases can then be mined to extract interesting information. From bugs and changes, for instance, we can tell how many bugs were fixed in a particular location.

This is what you get when doing such a mapping for eclipse. Each class is a rectangle in here (the larger the rectangle, the larger its code); the colors tell the defect density – the brighter a rectangle, the more defects were fixed in here. Interesting question: Why are some modules so much more defect-prone than others? This is what has kept us busy for years now.
The best hint so far what it is that determines the defect-proneness is the import structure of a module. In other words: “What you eat determines what you are” (i.e. more or less defect-prone).

For instance, if your code is related to compilers, it is much more defect-prone, than, say, code related to user interfaces.

Such observed correlations can be easily used to make predictors. We think that these predictors make sense, because the imports tell you the problem domain, and this determines how difficult the task is.
Predicting failure-prone packages

- Relate defect density to imports
- Base: Eclipse bug and version databases (Bugzilla, CVS)
- 36% of all packages had post-release defects
- Prediction using support vector machine

So we built appropriate predictors…

…and this is what we get if we rank 300 packages according to our predictor (which has learned from the remaining modules): if we look at the top 5%, 90% actually are defective. A random pick would have gotten us only 36%.

This was just a simple example. So, the most important aspect that software archives give you is automation. They are maintained automatically (“The data comes to you”), and they can be evaluated automatically (“Instantaneous results”). For researchers, there are plenty open source archives available, allowing us to test, compare, and evaluate our tools.
Besides changes and bugs, there are many more data sources around that are all maintained as part of a project. Many, many, more.

Combining these sources will allow us to get this “waterfall effect” – that is, being submerged by data; having more data than we could possibly digest.

Let me give you some examples how to put these data sources into use.
This is the oldest example, referring to work by Tom Zimmermann et al. at ICSE 2004 (and the work of Annie Ying et al. at the same time): You change one function – which others should be changed? This is easy to mine drawing on the change history and the code.

Defect density data as sketched before can be used to decide where to test most – of course, where the most defects are. If one additionally takes profiles (e.g. usage data) into account, one can even allocate test efforts to minimize the predicted potential damage optimally.

This is an alternate rendering of the previous slide, to be used in the summary slide at the end. It was not shown.
If one has effort data, one can tell how long it takes to fix a bug. Cathrin Weiß has a talk on this topic right after this keynote.

If one knows which program features correlate with which quality, one can use this measure to make all kinds of decisions. Correlating design with failure probability will help making well-founded design decisions. This is not to say that managers can’t do this right now, but having accurate project data available can certainly help assess the risks.

Finally, a glimpse into the future, taking natural language resources into account. The idea is to associate specs with (natural language) topics, and to map these topics to source code. What you then get is an idea of how specific topics (or keywords) influence failure probability, and this will allow you making predictions for specific requirements.
Let’s now talk about results. What should our tools do? Should they come up with nice reports, and curves like this one?

I would like to have these results actionable, and I want these actions to become apparent immediately. Here’s a way to make predictions actionable: In this Eclipse plug-in called HATARI, we have annotated each method with a bar, where the color of the bar indicates the risk of making a change. A red bar means a high risk; in the function you see here, 15 out of 17 changes resulted in a post-release defect. We want such assistance to be as non-disruptive as possible.

Programming environments also are the tools that allow us to collect, maintain, and integrate all this project data. This is where the waterfall becomes imminent. In pair programming, you have a navigator peering over your shoulder, giving you advice whether what you are doing is good or bad. We want the environment peer over your shoulder – as an automated “developer’s buddy”. Whatever we do must stand the test of the developers – if they accept it, it will be good enough.
In order to get there, we have plenty of challenges to overcome.

To start with, half of the data is related to programs, the other half to processes. People analyzing programs are not necessarily process experts, and vice versa.

Also, we have huge differences in terms of methods. For code and models, we use deductive reasoning, predicting what can happen in the concrete by analyzing the abstraction. In the other areas, it is the other way round: From collected data, we build abstractions that capture patterns and rules. These two methods are hard to bring together.
In the past, all of this data has been processed by individual researchers. Each of these faces stands for an entire community, sometimes encompassing thousands of researchers.

Matt Dwyer - Daniel Jackson - Tom Reps - Mike Ernst - Ben Liblit - Mary Jean Harrold - Gail Murphy - Tom Zimmermann - Cathrin Weiß - Rob DeLine - Harald Gall - Davor Cubranic

And to bring the data together, we need to bring together the researchers. What better place could there be than ICSE or this workshop for this purpose?

And as we are collaborating, we can make this “waterfall effect” a reality: All the data sources coming together to form a greater, overwhelming whole...
...and thus realizing the concept of Empirical Software Engineering 2.0. You will find traces of all these concepts in my talk – from participation over usability and remixability to, hopefully, economic consequences.

To sum up: To realize this concept of universal assistance (1), we need synergy between all the data sources (2). This will widely replace the expensive, error- and bias-prone manual data collection (3), and realize what I call Empirical Software Engineering 2.0 (4). And, finally...

... I now hope I answered the question on what the difference between a cheese and a waterfall is. Thanks for your attention!

This was the final slide, and it stayed on during the discussion. Overview slides like this one are tremendously helpful for discussions, as they summarize the talk, and I can refer to earlier points without having to switch back.
http://www.st.cs.uni-sb.de/