Modern programming environments foster the integration of automated, extensible, and reusable tools. New tools can thus leverage the available functionality and collect data from program and process. The synergy of both will allow to automate current empirical approaches. This leads to automated assistance in all development decisions for programmers and managers alike: “For this task, you should collaborate with Joe, because it will likely require risky work on the ‘Mailbox’ class.”

Turbo Pascal – just 30K (Eclipse: 118 MB – 4,000x as big)

Integration – Foto von Werkstatt, Werkzeugkiste
Tools evolve

But do these tools work together? Where is the whole more than the sum of its parts?

Tools integrate

Tools can only work together if they draw on different artefacts.

What are we working on in SE – we are constantly producing and analyzing artefacts: code, specs, etc.

Tools work together
Tools work together if they draw on different artefacts.

What are we working on in SE – we are constantly producing and analyzing artefacts: code, specs, etc.

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.
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.
Is it the Developers?

Does experience matter?

Bug density correlates with experience!

Is it History?

I found lots of bugs here. Will there be more?

Yes! (But where did these come from?)

How about metrics?

Do code metrics correlate with bug density?

Sometimes!
Uh. Coverage?

Does test coverage correlate with bug density? Yes – the more coverage, the more bugs!

Ah! Language features?

Are gotos harmful? No correlation!

Ok. Problem Domain?

Which tokens do matter? import • extends • implements
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.
Software Archives

- contain full record of project history
- maintained via programming environments
- automatic maintenance and access
- freely accessible in open source projects

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.

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.

Predicting Code Quality

“These components have the highest chance to fail in production”
Predicting Code Quality

“These components have the highest chance to fail in production”

Machine Learner

Predicting Code Quality

“These components have the highest chance to fail in production”

Machine Learner

Locating Abnormal Behavior

“This execution is abnormal because it accesses a password file in ParseURL()”

Sequence Learner
Locating Abnormal Behavior

“This execution is abnormal because it accesses a password file in ParseURL()”

Sequence Learner

open() read() unlink()

Suggesting Related Code

“Module Z contains code which you may find useful”

Suggesting Changes

“This test uses assert(); consider assertTrue() instead”
Suggesting Changes
“This test uses assert(); consider assertTrue() instead”

Machine Learner

Linking Artifacts
“This workaround is due to our customer’s requirement from December 12”

public class Purse {
    final int MAX_BALANCE;
    int balance;
    //@ invariant 0 ≤ balance && balance ≤ MAX_BALANCE;
    byte[] pin;
    //@ invariant pin != null && pin.length == 4 &&
    @ (forall int i; 0 ≤ i && i < 4;
        0 ≤ byte[i] && byte[i] ≤ 9)
    @/
    //@ requires amount ≥ 0;
    @ assignable balance;
    @ ensures balance == \old(balance) - amount &&
    @ \result == balance;
    @ signals (PurseException) balance == \old(balance);
    @*/
    int debit(int amount) throws PurseException { … }

Linking Artifacts
“This workaround is due to our customer’s requirement from December 12”

public class Purse {
    final int MAX_BALANCE;
    int balance;
    //@ invariant 0 ≤ balance && balance ≤ MAX_BALANCE;
    byte[] pin;
    //@ invariant pin != null && pin.length == 4 &&
    @ (forall int i; 0 ≤ i && i < 4;
        0 ≤ byte[i] && byte[i] ≤ 9)
    @/
    //@ requires amount ≥ 0;
    @ assignable balance;
    @ ensures balance == \old(balance) - amount &&
    @ \result == balance;
    @ signals (PurseException) balance == \old(balance);
    @*/
    int debit(int amount) throws PurseException { … }
When retrieving money from an ATM, the customer inserts his card and enters a PIN (a 4-digit number) and the amount to be retrieved...
Predicting Effort and Risk

“This task will take \( n \) person hours because it involves scripting”

Where do we get all this data from?

```
Bugs  Changes  Effort  Navigation  Chats  e-mail  Models  Specs  Code  Traces  Profiles  Tests
```

People who changed function \( f() \) also changed…

```
Bugs  Changes  Code  Traces  Profiles  Tests
```

Which modules should I test most?

```
Bugs  Changes  Code  Traces  Profiles  Tests
```

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.
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.
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.

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.
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.

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.
Let’s now talk about results. What should our tools do? Should they come up with nice reports, and curves like this one?

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.

Future environments will
• mine patterns from program + process
• apply rules to make predictions
• provide assistance in all development decisions
• adapt advice to project history
…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.

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 - Prem Devanbu

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 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?

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.