Course "Spezielle Themen der Softwaretechnik"

Errors and Defects

Lutz Prechelt
Freie Universität Berlin, Institut für Informatik
http://www.inf.fu-berlin.de/inst/ag-se/

- Error, defect, failure
  - Empirical facts about them
- Attributes and classification
- Orthogonal defect classification
- Other defect classifications
- Finding defects by static analysis
- Classification of errors
  - by phenotype, by genotype
- Learning to avoid errors
  - Personal Software Process
  - Research on automated help

Learning objectives

- Understand the difference between defects and errors
- Learn some facts about the economics of making and repairing defects
- Understand attributes and classifications of defects
- Understand some approaches for static defect detection
- Understand attributes and classifications of errors
- Understand possibilities for analyzing error processes
- Understand possibilities for helping programmers avoid errors

Part 1: Introduction

Basic terminology

- The basic phenomenon of interest is a program **failure** (Versagen)
  - That is, the observable behavior of a program is incorrect
- Failure happens due to a **defect** (Defekt) in the program
    - (sometimes several defects have to interact)
  - Defects are also called **fault**s
  - Some defects never lead to any failure
  - Many defects lead to a failure only sometimes, not in each program run
- A defect is created by an **error** (Fehler) of the engineers
  - either by commission (Falschtun) or by ommission (Versäumnis).
  - Errors need not happen during coding, they may also happen during design or requirements activities etc.

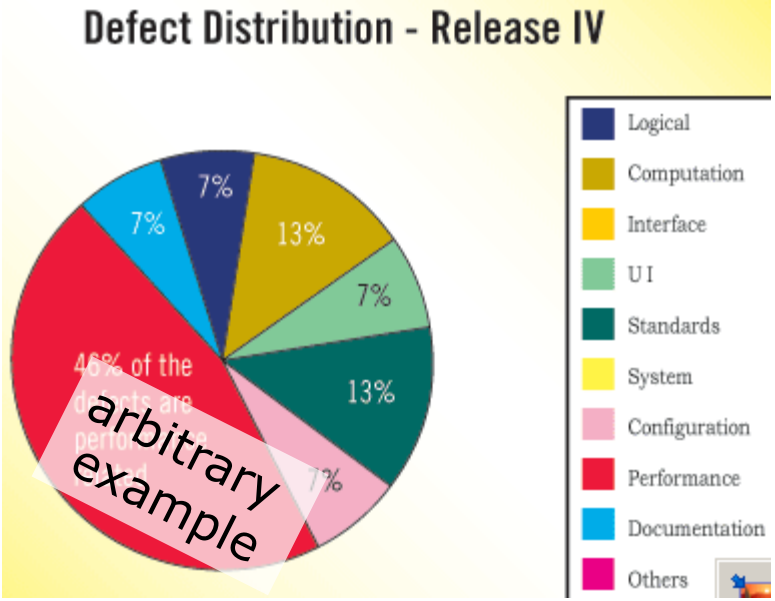
General rules of thumb

- Source: Forrest Shull, Victor Basili, Barry Boehm, A. Winsor Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, Marvin Zelkowitz: "*What we have learned about fighting defects",* METRICS '02, IEEE, 2002.
  - Results of an expert meeting, well-supported by scientific results

- Costs to fix one defect:
  - Finding and fixing a *severe* software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.
  - Finding and fixing *non-severe* software defects after delivery is about twice as expensive as finding these defects pre-delivery.
- Total rework costs:
  - A significant percentage (about 10%-60%) of the total effort on current software projects is spent on avoidable rework.
  - Rework effort decreases as process maturity increases.

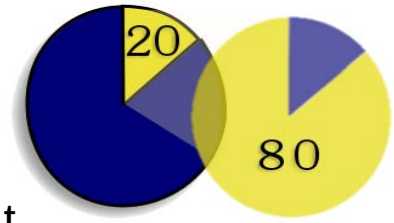
General rules of thumb (2)

- Distribution of cost for defect fixing:
  - Most of the avoidable rework comes from a small number of software defects
    - The 80/20 rule may sometimes apply:
      80% of rework costs is produced by the worst 20% of defects
    - Avoidable rework is work done to mitigate the effects of errors or to improve system performance.
  - Some rework may be unavoidable,
    - for example, work arising from adaptive, preventive, or user-requested changes.
  - Defects causing high amounts of rework are likely to be those that are "architecture breakers" or that are found "inappropriately" late in the development process.



General rules of thumb (3)

- Distribution of defects over modules:
  - 80% of a system's defects tend to come from about 20% of its modules.
    - However, the relationship varies based on environment characteristics such as processes used and quality goals.
  - During development, almost no modules are defect-free as implemented.
  - Post-release, about 40% of modules may be defect-free.



- Review efficiency:
  - Reviews catch more than half of a product's defects
    - regardless of the domain, level of maturity of the organization, or lifecycle phase during which they were applied.
  - Having multiple perspectives represented during software reviews is an effective practice.

Conclusions

- Errors and defects are very important phenomena in the software process
- Reducing defects is a major driver for improving the software process
  - Hence the Causal Analysis and Resolution (CAR) process area on CMMI level 5
- Quite a bit is known about defects
- Little is known about errors yet

How to describe failures, defects, and errors

Freie Universität Berlin

Attributes of failures:

- Reference
- Severity
- Type

Attributes of defects:

- Origin
- Location
- Consequence
- Expected failure time
- Type

Attributes of errors:

- Mode
- Type/Cause
- Indirect cause
- Root cause

See details on next slides

- There are no widely used standards for most of the terminology that follows

Attributes of failures: Reference, Severity

Freie Universität Berlin

Reference:

- The program behavior may be incorrect relative to
  - the behavior specification (if it exists),
  - the explicit user requirements (if they are clear) or
  - the implicit user expectations (otherwise)

Severity:

- light
  - The failure is unproblematic and will often go unnoticed
- medium
  - The program's usefulness is impaired
- strong
  - The program's usefulness is heavily reduced
- critical
  - The program's usefulness is wrecked
  - or safety or security are compromised

Attributes of failures: Type

Type:

- crash:
  - The system terminates prematurely
    - with or without an error message
- gap:
  - Functionality is missing
- wrong:
  - The system produces output that is incorrect
- hang:
  - The system does not produce output when it should
- nonfunctional failure:
  - The system violates a non-functional requirement
    - regarding e.g. execution speed, memory consumption, usability, compatibility

Attributes of **defects**: Origin, Location

Origin:

- What error underlies the defect?
- When was it made?
  - requirements, design, implementation, correction

Location:

- Which program element contains the defect (and thus needs change)?
  - Some defects have distributed nature
    - e.g. many performance problems
  - Many defects are documentation-related

Attributes of defects: Consequence, Expected failure time

Freie Universität Berlin

Consequence:

- What failures can this defect provoke?
  - type, severity
- When are these failures to be expected?
  - Now
  - After potential changes of usage or environment
  - After potential program changes

Attributes of defects: Type

Many different classifications exist

- Some classifications are small, others large
  - many types → provides much information
  - few types → simple, reliable classification (often preferable)
- Types may be disjoint, overlapping (beware!), or hierarchical
- Types may refer to different aspects of the defect:
  - structure, e.g. the type of design element that is incorrect
    - e.g. for code: an expression, assignment, method call, control structure, concurrency coordination, etc.
    - e.g. for design: module interface, global architecture etc.
    - e.g. for requirements: a step, condition, functionality, constraint, user group, etc.
  - deviation, the nature of the difference between the actual state and a correct state of the inflicted element

Attributes of **errors**: Mode, Type/Cause

Freie Universität Berlin

Mode:

- commission (Falschtun)
- ommission (Versäumnis)

Type and cause: (Note: all boundaries are very fuzzy!)

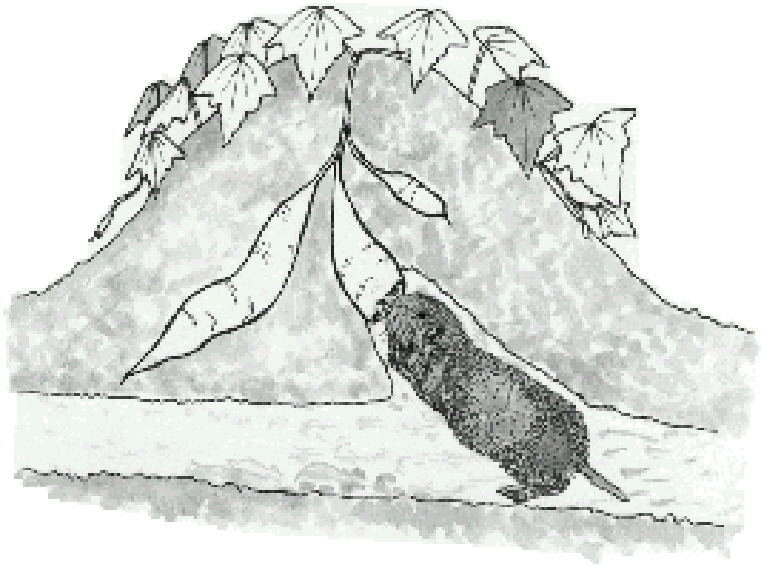
- mistake (Irrtum)
  - lack of relevant knowledge, bad knowledge
  - use of incorrect information to judge situation
    - outdated
    - incorrectly communicated
    - wrong results of internal cognitive processes
- slip (Versehen)
  - bad execution of correctly planned actions

Attributes of errors: Indirect cause, Root cause

Freie Universität Berlin

Indirect cause(s) and root causes:

- Many different events or situations may underlie an error
- Little specific knowledge is available about typical root causes of errors in software development contexts
  - There are only a number of domain-specific case studies



Part 2: Defects

What can we do with/about/around defects?

- Understand the errors underlying defects
  - See part 3 of this lecture
- Understand what makes defects difficult to find/remove
- Understand categories of defects in general
  - e.g. for simplifying finding and describing defect patterns
  - By scientifically collecting and analyzing sets of defects
- Understand methods for detecting defects
  - Testing (not our topic in this course)
- Understand methods for locating defects
  - Debugging (not our topic in this course)
- Understand patterns of common defects for one product or organization
  - Statistical process control, see CMMI level 4
- Detect/locate/remove defects in one particular product
  - Tests+Debugging, Reviews, Static analysis

What makes defects difficult to locate?

Freie Universität Berlin

Marc Eisenstadt: *" 'My hairiest bug' War Stories"*, Communications of the ACM 40(4), April 1997

- Analyzes commonalities in "war stories" sent by programmers
- Debugging-focused, but can be transferred to other activities

Difficult-to-locate kinds of defects:

1. Cause/effect chasm
   - The failure is far removed (in source text or execution time) from the defect
2. Tools inapplicable or hampered
   - The problem can somehow not be investigated in the usual way
3. Faulty assumption
   - Something is in fact different from what I strongly believe

Example defect type classes: ODC

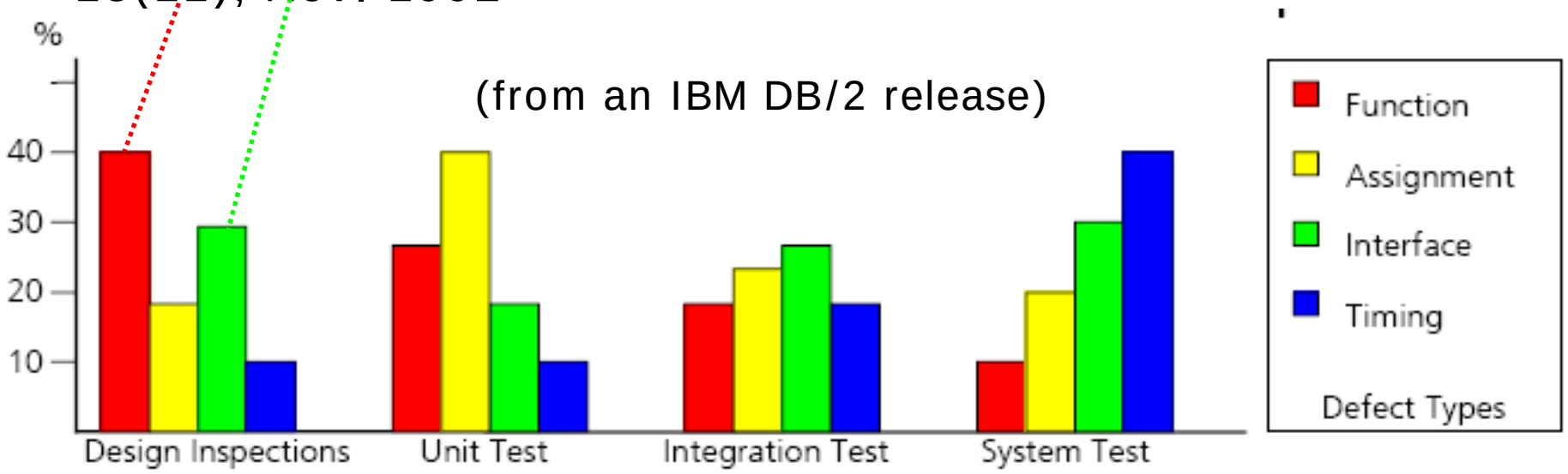
ODC:

Orthogonal defect classification

- Assignment
  - Data handling is incorrect, e.g. initialization missing, data structure corrupt (→ code, small correction)
- Algorithm
  - Program logic does not meet intention (→ code)
- Function
  - A system functionality is missing or ill-defined (→ design)
- Interface
  - A module/subsystem interface is ill-defined (→ design)
- Checking
  - Data validation or error checking is incomplete or wrong (→ code)
- Timing
  - Timing (network) or synchronization (concurrency) aspects are wrong (→ low-level design or code)
- Build
  - Configuration management or build management have gone wrong: wrong versions or configurations used etc.
- Documentation
  - Internal or external documentation is incorrect, useless, or misleading

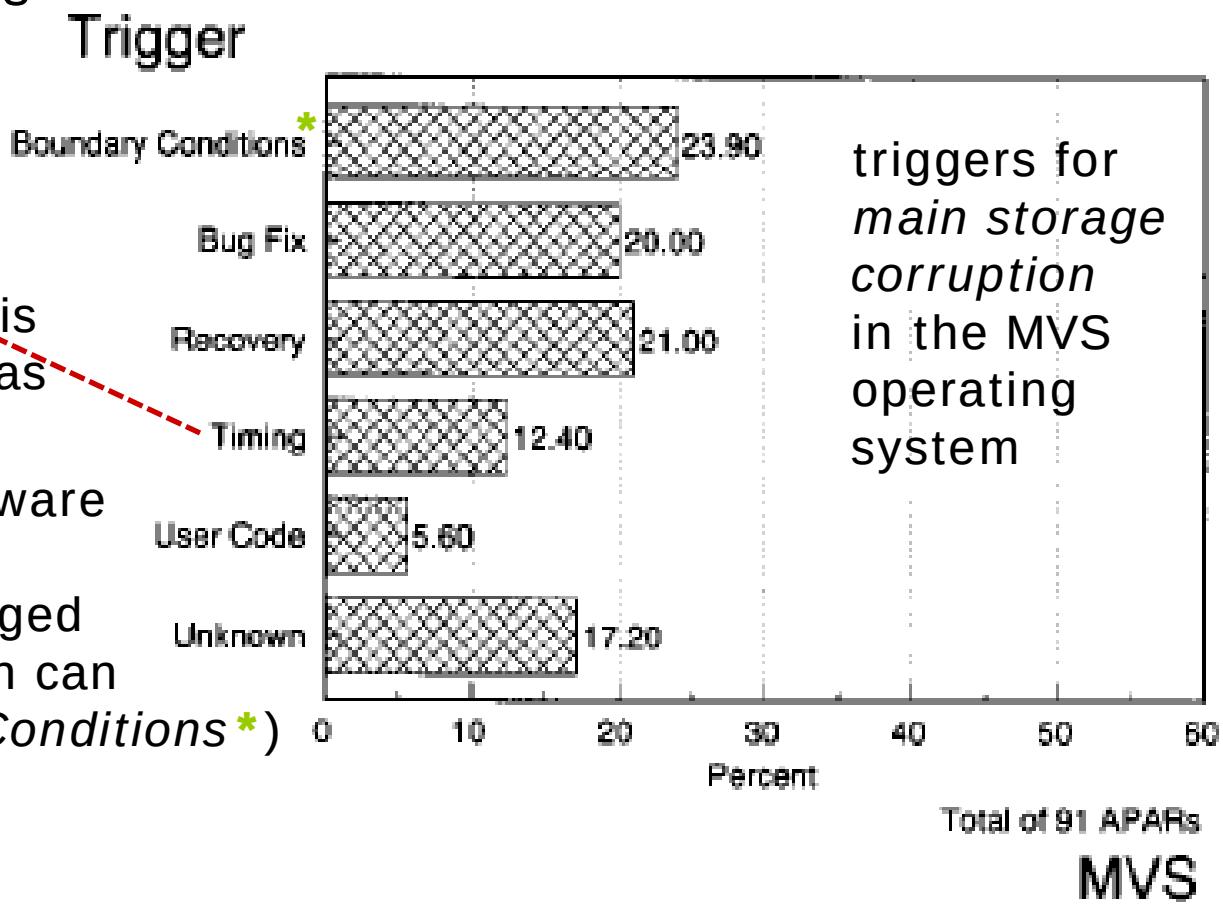
Making use of ODC

- Idea: Each development phase should have a characteristic defect detection profile
  - e.g. *interface* defects are found mainly in design inspections
  - *function* defects are found continually less frequently
- Profiles that differ from expectations indicate bad process execution in previous phases
- Source: Chillarege et al., IEEE Trans. on Software Eng., 18(11), Nov. 1992



ODC, part 2: Defect triggers

- A defect trigger is the condition/situation/activity needed to make a defect provoke a failure
- Trigger distributions help decide how to distribute quality assurance effort
  - Example: *Timing* is not as important as was thought,
  - hence some hardware platform testing should be exchanged for reviews (which can check *Boundary Conditions**)



ODC, part 2: Defect triggers (2)

- The relative frequency of different defect triggers may change a lot from system test to field use
  - can be found e.g. during beta test
- This indicates that the system test approach may have been inefficient
  - it has reduced the defects corresponding to the now-more-frequent triggers less than others
  - depending on how severe the corresponding failures are, this may be good or bad

- This practice is a good example of a practice in the CMMI's Quantitative Project Management (QPM) level-4 process area
  - SG 2: Statistically Manage Subprocess Performance

Defect classification survey

- Bernd Freimut: *"Developing and using defect classification schemes"*, Report 072.01/E, Fraunhofer IESE, Sept. 2001
- A survey of
  - defect classification schemes (and their literature references),
  - how defect data can be used
- Lists many possible attributes of defects and failures
  - location
  - creation/detection/correction time (usually as a phase)
  - symptom: observation context when defect was found
  - end result: the actual failure and its impact
  - creation/detection/correction mechanism
  - cause: underlying error
  - severity
  - detection/correction cost

Defect classification survey: Classification schemes

Freie Universität Berlin

General-purpose schemes:

- IEEE Standard Classification for Software Anomalies
  - 21 attributes → very laborious!
  - describes a defect handling process (*recognition, investigation, action*)
- Hewlett-Packard scheme
  - 3 attributes: origin (phase), type (21 types), mode (*missing, unclear, wrong, changed, better way*)
  - Comment: good pragmatic approach

Special-purpose schemes:

- Orthogonal defect classification
  - as discussed before; 8 attributes (*type, trigger*, and some details)
  - Comment: aimed at CMMI level-4 statistical process control only, too coarse for most other purposes
- Porter's scheme for requirements inspections
  - 4 omission defect types (missing functionality/performance/ environment/interface)
  - 4 commission types (ambiguous, inconsistent, incorrect, wrong_section)
  - Comment: adequate for reqs.

Defect classification survey: Classification scheme quality attributes

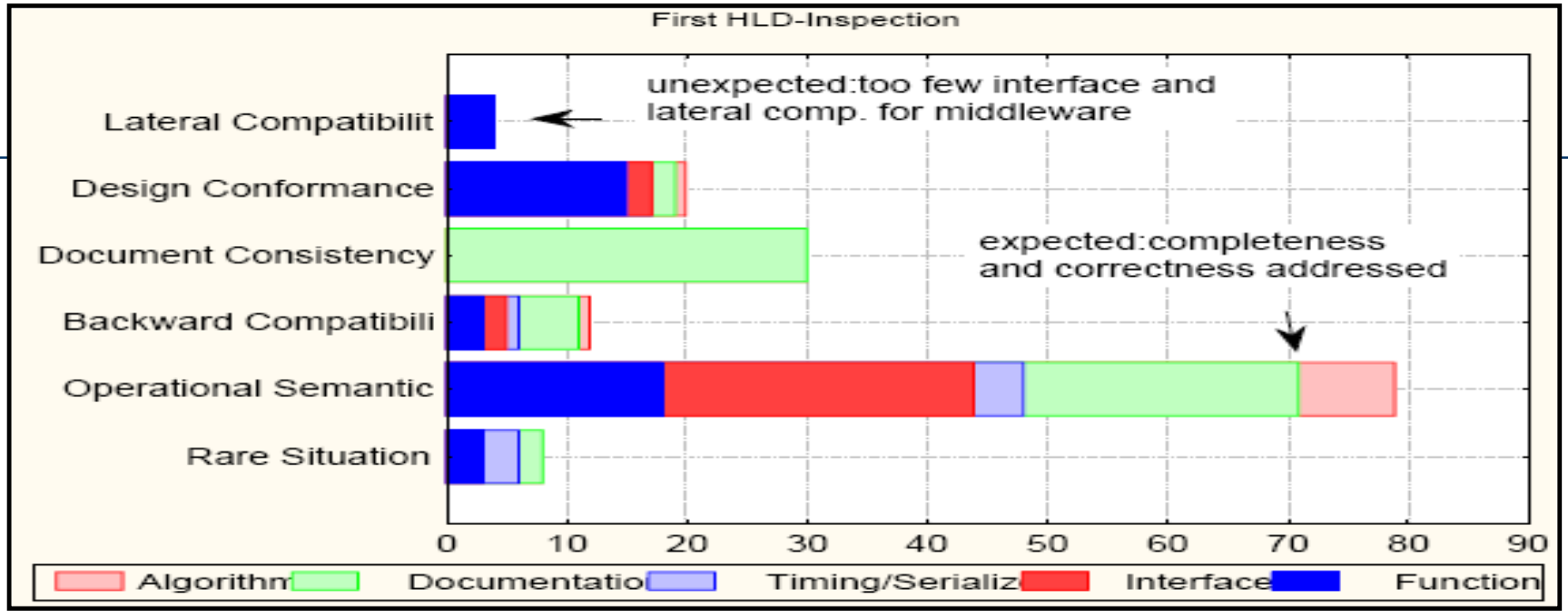
A defect classification scheme should have

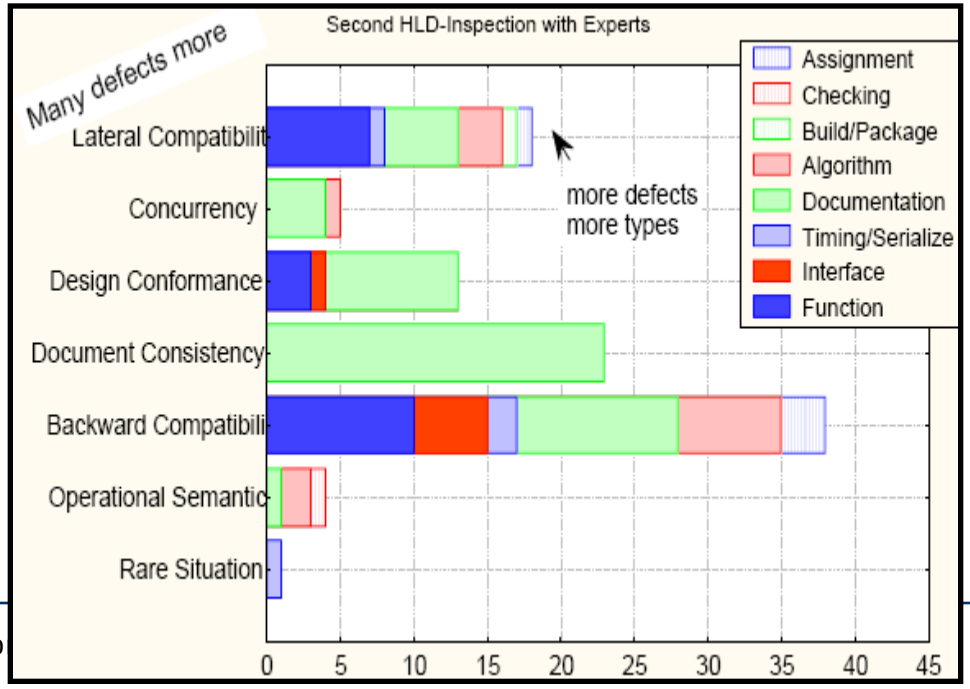
- Non-overlapping attributes, identical for all activities
  - else classification will become confusing and error-prone
- Orthogonal attribute values
  - i.e., only one value applies in each case
  - else classification will become ambiguous
- Complete sets of attribute values
  - else some defects cannot be uniquely classified
- Descriptions for the attribute values
  - else different people will classify differently (→ low reliability)

Defect classification survey: Defect data analysis goals

Freie Universität Berlin

1. Overview of defect distribution
   - often useful when beginning a defect measurement program
2. Defect cause analysis
   - to find systematic errors (→ defect prevention)
   - either by deep discussion of a few defects (qualitative analysis, root cause analysis)
   - or by quantitative overview of an appropriate distribution (such as in ODC): quicker, but requires much more data
     - e.g. the number (better: cost) of defects created during design
3. Optimizing inspections
   - e.g. inspecting for those defects with highest benefit/cost ratio, finding out in which areas inspectors need training etc.
     - The following example is from the inspection of the high-level design of a complex IBM middleware product:
     - The initial set of reviewers was insufficiently qualified and produced a dubious distribution of defects found (see next slide)
       - wrt. Lateral Compatibility with other IBM products


First HLD-Inspection
unexpected:too few interface and lateral comp. for middleware
expected:completeness and correctness addressed
Lateral Compatibilit
Design Conformance
Document Consistency
Backward Compatibili
Operational Semantic
Rare Situation
0
10
20
30
40
50
60
70
80
90
Algorithm
Documentatio
Timing/Serializ
Interface
Function


Second HLD-Inspection with Experts
Many defects more
more defects more types
Assignment
Checking
Build/Package
Algorithm
Documentation
Timing/Serialize
Interface
Function
Lateral Compatibilit
Concurrency
Design Conformance
Document Consistency
Backward Compatibili
Operational Semantic
Rare Situation
0
5
10
15
20
25
30
35
40
45

Defect classification survey: Defect data analysis goals (3)

Freie Universität Berlin

4. Evaluating techniques
   - Compare defect profiles of development techniques or defect detection profiles of quality assurance techniques

5. Optimizing quality assurance (QA)
   - Compare defect type distribution to expectations:
     - (for example as described for ODC)
   - Defect types that are more common than expected signal problems in the product
     - due to problems in previous steps of the QA process
   - Defect types that are less common than expected may signal
     - either problems in the QA process (defects not found)
     - or an unusually good product (defects not present)
   - In particular, compare defects found after delivery to defects found during QA

Defect detection by static analysis: Searching for known patterns

Freie Universität Berlin

David Hovemeyer, William Pugh: *"Finding bugs is easy"*, ACM SIGPLAN Notices 39(12), December 2004

- FindBugs: a static analysis tool that scans Java bytecode for hard-wired code patterns that signal defects
  - sort of an automated review

Example patterns:

- Bad Covariant Definition of Equals (EQ)
  - public boolean equals(Myclass)
  - Was probably intended to overload Object.equals(Object)
  - Found by a simple scan of class structure (not even method bodies)
- Inconsistent Synchronization (IS2)
  - Report private fields that are usually accessed while a lock is held, but sometimes without
  - Found by a single full scan of code
- Null Pointer Dereference (NP)
  - if (entry == null) { IContainer container = getContainer(entry.getPath()); // null!
  - Found by basic intra-procedural dataflow analysis
- Read Return Should Be Checked (RR)
  - read() may return fewer bytes than expected

Evaluation results

- Found many real defects in mature software systems!
- But may find many false positives

eclipse-2.1.0

	warnings	serious	harmless	dubious	false pos
NP	43	93%	0%	6%	0%
OS	16	6%	6%	18%	68%
RR	22	4%	0%	0%	95%
RV	9	100%	0%	0%	0%
UR	0	—	—	—	—
UW	0	—	—	—	—

jboss-3.2.2RC3

	warnings	serious	harmless	dubious	false pos
IS2	2	50%	0%	0%	50%
NP	10	100%	0%	0%	0%
OS	2	100%	0%	0%	0%
RR	0	—	—	—	—
RV	2	0%	0%	0%	100%
UR	2	50%	0%	0%	50%
UW	1	100%	0%	0%	0%
Wa	0	—	—	—	—

Defect detection by static analysis: Inducing defect patterns

Freie Universität Berlin

- Engler et al.: *"Bugs as deviant behavior: A general approach to inferring errors in systems code"*, ACM SIGOPS Operating Systems Review 35(1), October 2001
- Program analysis as seen above, but the patterns are not pre-specified
- Rather, the tool looks for recurring patterns that are sometimes violated
- Example:
  - if lock() co-occurs with unlock() 99 times and without it 1 time,
  - then the latter is probably a defect
- Requires elaborate program analysis
  - with much built-in knowledge of certain C library constructs

Template (T)	**Examples (E)**	**Population (N)**
"Does lock <*L*> protect <*V*>?"	Uses of `v` protected by `l`	Uses of `v`
"Must <*A*> be paired with <*B*>?"	paths with `a` and `b` paired	paths with `a`
"Can routine <*F*> fail?"	Result of `f` checked before use	Result of `f` used
"Does security check <*Y*> protect <*X*>?"	`y` checked before `x`	`x`
"Does <*A*> reverse <*B*>?"	Error paths with `a` and `b` paired	Error paths with `a`
"Must <*A*> be called with interrupts disabled?"	`a` called with interrupts disabled	`a` called

Defect detection by static analysis: Combining with change processes

- Benjamin Livshits, Thomas Zimmermann: *"DynaMine: Finding Common Error Patterns by Mining Software Revision Histories"*, ACM SIGSOFT Software Engineering Notes 30(5), September 2005
- Analyzes the set of changes performed on a SW system as represented in the version archive
- Basic assumptions:
  - Method calls often added together in the same change may represent a required usage pattern
  - One-line changes often represent defect corrections

- Example change set:

File	Revision	Added method calls
Foo.java	1.12	o1.addListener o1.removeListener
Bar.java	1.47	o2.addListener o2.removeListener System.out.println
Baz.java	1.23	o3.addListener o3.removeListener list.iterator iter.hasNext iter.next
Qux.java	1.41	o4.addListener
	1.42	o4.removeListener

  - Can infer a correct pattern and one violation (plus its repair)
    - addListener, removeListener

Defect detection by static analysis: Combining with dynamic analysis

Freie Universität Berlin

(DynaMine continued:)

- Now the code is instrumented to count occurrences of the patterns at run time
- Patterns with too few dynamic instances or too many violations are rejected
  - considered "unlikely"
- and the rest are classified into normal ("usage") and likely defect ("error") patterns

Evaluation:

- The tool was used on
  - Eclipse (2.9 MLOC, 19000 classes)
  - jEdit (0.7 MLOC, 6600 classes)
- Static analysis found 56 pattern candidates
- Dynamic analysis detected 32 of those at run time
- 11 of those were classified as unlikely
- The remaining 21 should be investigated as possible defects

- See part of the detailed results on the next slide

Evaluation results

Freie Universität Berlin

Method pair $\langle a, b \rangle$		Dynamic		Static		Type
Method a	Method b	v	e	v	e	
		CORRECTI				
NewRgn	DisposeRgn					
kEventControlActivate	kEventControlDeactivate					
addDebugEventListener	removeDebugEventListener	4	1	4	1	Unlikely
beginTask	done	332	759	41	28	Unlikely
beginRule	endRule	7	0	4	0	Usage
suspend	resume					
NewPtr	DisposePtr					
addListener	removeListener	143	140	35	29	Error
register	deregister	2,854	461	17	90	Error
malloc	free					
addElementChangedListener	removeElementChangedListener	6	1	1	1	Error
addResourceChangeListener	removeResourceChangeListener	27	1	21	1	Usage
addPropertyChangeListener	removePropertyChangeListener	1,864	309	54	31	Error
start	stop	69	18	20	9	Error
addDocumentListener	removeDocumentListener	38	2	14	2	Usage
addSyncSetChangedListener	removeSyncSetChangedListener					
addNotify	removeNotify	3	0	3	0	Unlikely
setBackground	setForeground	75	175	5	5	Unlikely
contentRemoved	contentInserted	17	11	7	5	Error
setInitialDelay	start	0	32	0	2	Unlikely
registerErrorSource	unregisterErrorSource					
start	stop	83	98	10	13	Error
addToolBar	removeToolBar	24	43	5	5	Error
init	save					

v: validated instances
e: exceptions

Part 3: Errors

Definition of Error

- "An inappropriate action, or intention to act, given a goal and the context in which one is trying to reach that goal."
  - James Reason: "Human Error", Cambridge University Press 1990

- Errors share some common properties
  - Any erroneous action had a local goal/purpose/intention
  - Errors cause the non-achievement of the goal
  - Errors are always unintended
  - Errors are potentially avoidable

- Errors usually (but not always) cause defects
  - and defects often cause failures or require rework or both

Classification

- A person's common error types are called "**bias**"
  - bias: Hang, Neigung (math.: systematischer Fehler)
  - Each person has his/her personal error tendencies
- The **context and circumstances** of work heavily determine the likelihood of an error
  - stress and strain
  - personal feelings and doubts
  - interruptions, distractions
  - anxiousness
  - social environment

- Error classification types:
  - By **phenotype**:
    What do errors look like? What are the resulting phenomena?
  - By **genotype**:
    What cognitive mechanism underlies a class of errors?

Classification by phenotype

Classification by **Hollnagel**:

- *"The Phenotype of Erroneous Actions: Implications for HCI Design"*. In Weir and Alty, editors, *Human-Computer Interaction in Complex Systems*, pp. 1–32. Academic Press 1991.
- Classification is based on the notion of human action as a sequence of steps

1. Omission
   - Missing a step, especially at the end of the plan (failure to complete)
2. Repetition
   - Erroneously repeating a step
3. Replacement
   - Performing a wrong step in place of the right one
4. Reversal
   - Exchanging order of two steps
5. Intrusion
   - Including an unnecessary step or performing a step too early
6. Delay
   - Taking a step too late (in case timing is an issue)

Classification by genotype (1)

James Reason: "Human error"

- Mistake (Irrtum) vs. Slip (Versehen)

- **Mistake**: The plan was wrong (although performed well)
  - Either a knowledge-based mistake
  - or a rule-based mistake
    - Details follow

- **Slip**: The plan was right but has been executed badly
  - Slips are skill-based
    - Details follow

Classification by genotype (2)

Knowledge-based mistakes (Denkirrtum):
Errors in thinking strategy, conclusions, or knowledge.

- Availability bias: Ideas that were thought-out/learned/ experienced recently are more likely considered relevant and correct.
- Overconfidence bias: Once chosen, a strategy is often kept even if it is unhelpful.
- etc.

Rule-based mistakes (Verfahrensirrtum):
Applying a rule that is not applicable in this situation

- Coning of attention: Tendency to ignore new sources of information and possibilities.
  - e.g. focus on opening the aircraft door after a crash although the aircraft body has a gaping hole.
- Reversion under stress: Tendency to revert to first-learned behavior, even if better behavior has been learned since
- etc.

Further knowledge-based mistakes: Typical human reasoning biases

Freie Universität Berlin

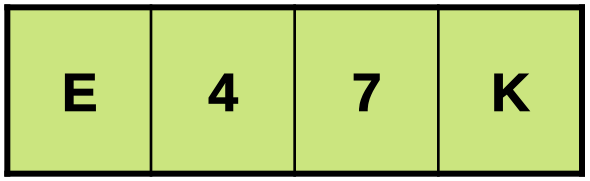
- Confirmation bias
  - If we want to check a statement, we tend to look only for confirmatory information and not look enough for contradictions
  - Dangerous threat for testing!

- Related mistakes in logical reasoning

  In particular with respect to implications:
  - Assume you know that $A \Rightarrow B$
  - If you know A, a correct conclusion is B
    - also from $\neg B$, a correct conclusion is $\neg A$
  - If you know $\neg A$, a wrong conclusion is $\neg B$
    - incorrect negation
  - If you know B, a wrong conclusion is A
    - incorrect reversal

Further knowledge-based mistakes: Confirmation bias example

- Given configurations of cards.
- You know that each card has
  - a letter on the front and
  - a number on the back



- You have the hypothesis that

  "Each card that has a vowel on the front has an even number on the back"
- You want to test this hypothesis for the four cards shown

- Which card(s) do you need to turn over?

Classification by genotype (3)

Skill-based slips
(Handlungsfehler)

- Capture slip: Apply a simple, well-trained behavior in an inappropriate situation due to lack of attention
  - e.g. catch IOException where you should catch something else
- Description slip: Confuse similar things due to lack of a precise specification of what you want
  - e.g. confuse two related variables
- Data-driven error: A current thought intervenes in a simple, well-trained action and leads to a slip
  - e.g. x + y + lunch + z
- Associative activation slip: Acting on incorrect associations
  - e.g. opening the door after the telephone ringed
- Loss of activation slip: Lost focus due to some interruption
  - "What did I want to do here?"
- Mode error: Treating something like it must be treated when it has a different state (say, its default state)
  - e.g. closing a file that was never successfully opened

Real-life examples of errors

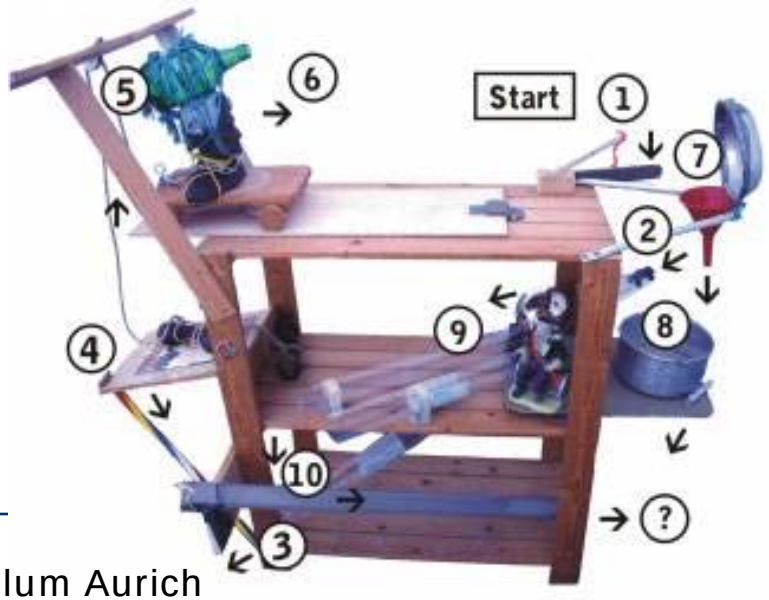
Video sequences of defect insertion during programming:

- scanf("%d%d", &bnum, &rnum)
  - forgot to apply address operator
- for (j=bnum-1; j<bnum+rnum; j++)
  - mistake when resuming after interruption
- int scen; ... while (scen <= n)
  - forgotten initialization

Expertise and errors

- When we learn to become experts
  - we first act knowledge-based (analytic)
  - then more and more rule-based (heuristic)
  - finally more and more skill-based (automatic/autonomous).
- When problems occur, we step back to earlier modes.

- With experts, errors tend to cluster into error chains:
  1. A slip leads to a breakdown of skill-based (automatic) action,
  2. we regress to less usual rule-based action,
  3. which may produce further errors,
  4. which make us regress to knowledge-based action,
  5. which is both inefficient and error-prone.



Miraculum Aurich

Lutz Prechelt, prechelt@inf.fu-berlin.de

Avoiding mistakes

- To be able to avoid knowledge-related errors (mistakes), it is important that you understand what you do not know

- When learning a domain or technology:
  - Obtain a full overview of its elements, regions, parts
  - Understand how well you understand each of them
  - Learn to recognize each of them when they occur in practice

- When working in a project:
  - Obtain a full overview of its aspects/problems
  - Understand how much you know about each of them
  - Learn to recognize when knowledge of which of them might be relevant

Avoiding slips

- To be able to avoid skill-based errors (slips),
  it is important that you understand when they tend to occur:

- What are the typical kinds of slips you make most frequently?
  - If you understand this well, your brain may be able to notify you when a slip is about to happen or has just happened

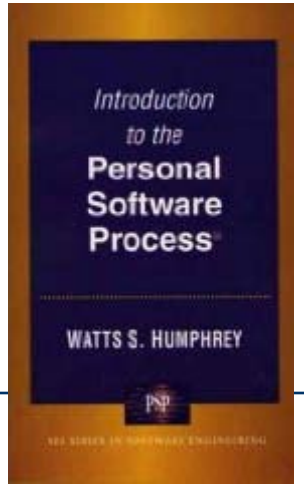
- What are the situations in which they tend to happen?
  - If you understand this well, you can train to be extra careful whenever you are in such a situation

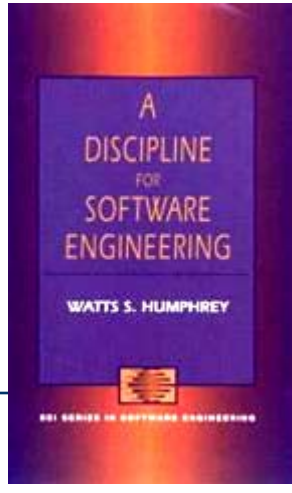
Understand yourself: Learning about typical error patterns

Freie Universität Berlin

- The profile of which kinds of errors tend to be committed under which circumstances is highly individual
- For learning to avoid errors it would help to know this profile
  - The only way to get it is by observation
- The Personal Software Process (PSP) is a programmer self-improvement methodology that (among other things) describes a simple process for learning about your errors(*)
  - Watts Humphrey: "A Discipline for Software Engineering", Addison-Wesley 1995
  - Watts Humphrey: "Introduction to the Personal Software Process", Addison Wesley 1997

(*) The PSP actually talks about defects only, not about errors.
  - "defect prevention", as in CMMI Level 5





The PSP suggests the following for learning to avoid errors:

- Whenever you detect a defect (e.g. be observing a failure),
  - create a protocol entry in your defect list (timestamp)
  - Once you have understood the defect, log defect data (type, location, detection time) and your impression of what was the underlying error (type, reason)
- After collecting enough such data (~100 entries), analyze the data for typical patterns:
  - Most common error types and error reasons
  - Error types leading to the most difficult-to-remove defects
  - Error types that could be most easily avoided
- Repeat this procedure regularly

Personal Software Process Problems

- The approach works quite effectively
  - Lutz Prechelt: *"Accelerating Learning from Experience: Avoiding Defects Faster"*, IEEE Software. 18(6):56-61, November 2001

- However, very few programmers afford enough discipline to actually use it:
  - Philip Johnson, Anne Disney: *"A Critical Analysis of PSP Data Quality: Results from a Case Study"*, Empirical Software Engineering 4:317-349, 1999.
    - PSP data contains a lot of mistakes
  - Philip Johnson et al.: "*Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined*"
    - PSP data collection needs to be automated
  - Lutz Prechelt, Barbara Unger: *"An experiment measuring the effects of personal software process (PSP) training"*, IEEE Trans. on Software Engineering, 27(5):465–472, 2001.
    - PSP is hardly used even after formal PSP training

Research goal: Find out how to help programmers avoid errors

Freie Universität Berlin

- The subsequent slides will discuss our research on supporting programmer error avoidance
  - How to do it if the PSP is not practical

- When successful, this research would be dazzlingly relevant
  - because software engineers commit errors all the time
  - and that is very costly:
    "*What we have learned about fighting defects",* METRICS '02 claims "A significant percentage (about 10%-60%) of the total effort on current software projects is spent on avoidable rework."
  - This value under-estimates the cost of errors for two reasons:
    - It considers only explicit rework after QA steps and ignores the rework that happens during the initial work on an artifact
    - It ignores the cost of errors that are never repaired, such as design decisions that make implementation much costlier
- but it turns out to be very difficult to do

Where to help

There is less hope for avoiding knowledge-based mistakes:

- When you commit a knowledge-based mistake, you do it in good faith, just like your correct activities.
- There is little chance to recognize such situations in real time.
- There may not be much payoff from reviewing them after the fact, because knowledge-based mistakes are (hopefully) quite unique.
  - If they cluster, systematic communication deficiencies are probably the cause;
  - therefore, standard process improvement can probably help.

So we concentrate on rule-based mistakes and slips instead.

What to aim at: Possible kinds of help

What help would we like to give the programmer?

- Ideally, a fully automated mechanism would turn on a "warning light" whenever an error is about to happen or has just happened
  - but please almost without false positives!
- Less ambitiously, we could provide help for post-hoc understanding of one's own behavior
  - in situations known to be error-prone
  - in situations where we later found an error has happened

Basic research procedure

The approach is based on case studies (with many cases):

- **Collect** lots of data on instances of error incidents
  - and (for comparison) also on related non-error and near-miss incidents
- **Analyze** this data to find recurring patterns
  - The insights gained here are valuable research results even if we do not manage to create actual helper mechanisms from them
- **Construct** help based on these patterns
  - tools and methods with which programmers can learn to avoid errors

Data collection: What data?

Potentially relevant data:

- Thought processes of programmer
  - conscious
  - unconscious
- Actions of programmer
  - Facial expression, body language etc.
  - Computer usage
    - reading, manipulating
  - Non-computer activities
    - read or write: work with paper, whiteboard etc.
    - leave seat, talk to people, pause etc.
- Events in the environment
  - Noises (e.g. telephone ring) and other distractions
  - Computer notifications (e.g. email arrived, appointment)
  - Interruptions by people (boss, colleague, other)

Data collection: How to collect it?

Freie Universität Berlin

Potential ways of recording relevant data:

- Thought processes of programmer
  - intrusively: think-aloud protocols
  - "unintrusively": MRT, EEG

  Hardly realistic!

- Actions of programmer, Events in the environment
  - by audio and video recordings

  Possible (at least in the lab), but analysis is labor-intensive

- Computer-related events
  - Can be recorded automatically in a computer-analyzable form

  Limited, but very attractive!

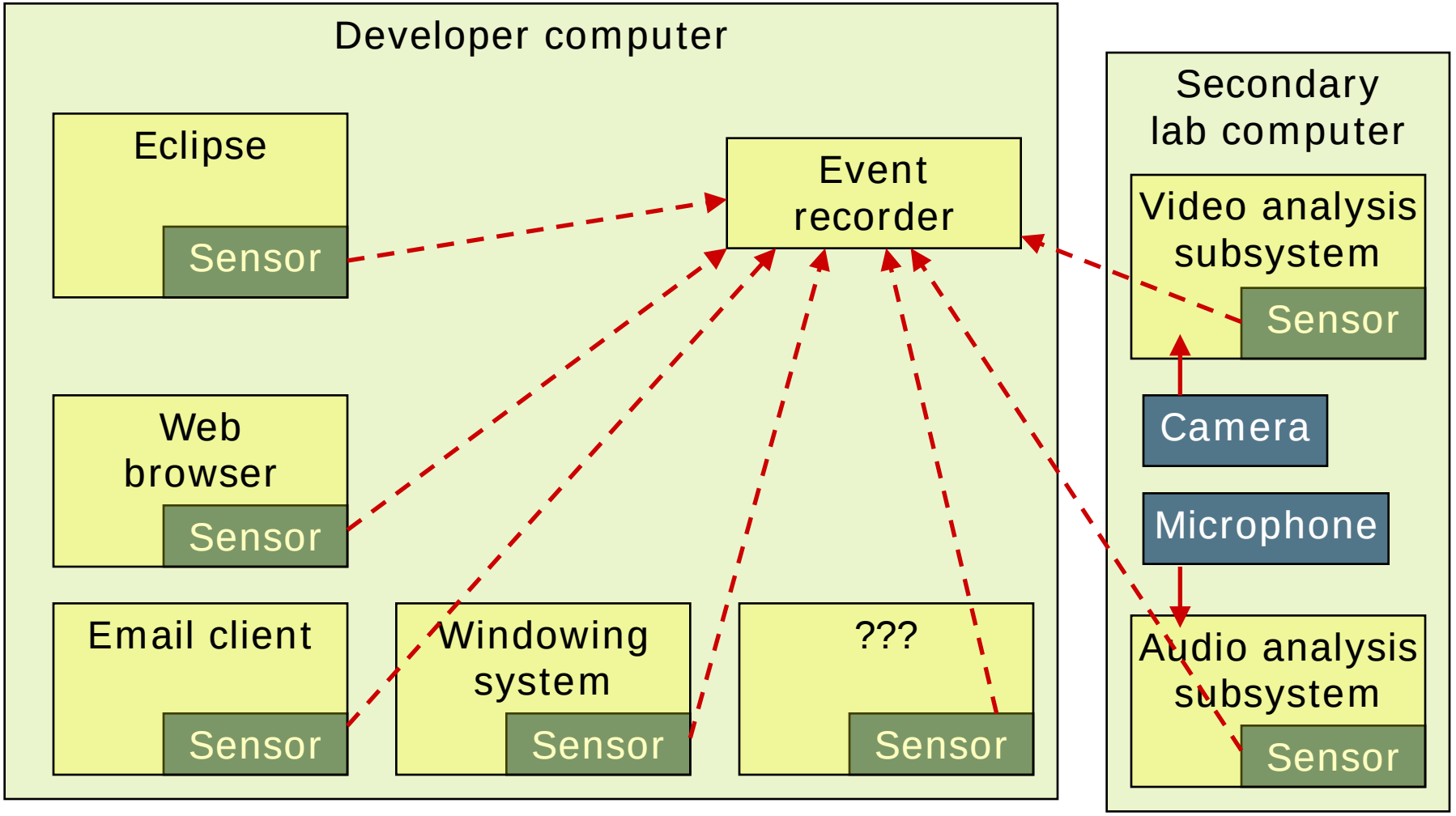
  Note that most results of the SW process will occur in this form

Automated data collection: Requirements

Freie Universität Berlin

- Can automatically detect and record computer-related events
  - No restriction on the type of events
  - Extensible in a modular fashion
- Platform-independent
  - MS Windows, Linux, Mac OS, etc. ➔ Java-based
- Minimal influence on developer and computer environment
  - Modest footprint (memory, CPU, network bandwidth, disk space)
  - Easy installation, configuration, and operation
- Allows fine data granularity
  - Such as individual changes to a file: Dozens of events per minute
    - but not basic events (keypress, mouse click, mouse movement), as they are too hard to make sense of

Automated data collection: Approach



Sensors and recorder are configurable

Automated data collection: Electrocodeogram (ECG)

- We have built such an infrastructure:
  - http://www.electrocodeogram.org
  - Diplomarbeit Frank Schlesinger
- Recorder: ECGlab
  - a server (Java application)
  - configurable by connecting filters, writers etc. via a desktop GUI
  - Records event type, parameters, time stamp, source
  - Uses XML files for storing results
  - Installation either stand-alone or as Eclipse plugin
- Sensors
  - Generate events in a standardized XML record format
  - Communicate via light-weight socket protocol or SOAP
  - Can be written in any technology and language, stand-alone or as plugins, DLLs, or whatever

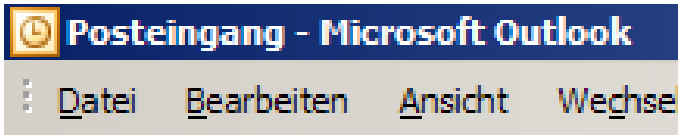
ECG Eclipse Sensor

An Eclipse plugin; trivial installation and update

Event types:

- Open, close, focus gained, and focus lost for each editor, auxiliary view, dialog, and the main window
- View buffer, change buffer (includes transfer of contents)
  - Text change followed by two seconds of idle time causes event
- Start and termination of program runs or debugging sessions
- Start and termination of Ant tasks
- File creation, renaming, movement, and deletion from within Eclipse or recognized by Eclipse (e.g. after a refresh)
- File save
- Cut, Copy, and Paste including Clipboard contents
- User inactivity after some time without mouse/keyboard usage
- Fuzzy Focus Tracker: Most likely focussed Java construct

ECG *window title logger* sensor



- Detects changes in application window focus
  - currently implemented for MS Windows only
- Events identify the application of the newly focused window
- Events identify the window title of the newly focused window
  - e.g. for web browsers this often represents the HTML page title
  - e.g. for explorer windows this often represents the directory name
  - likewise for various other applications
- Causes serious "Big Brother" problem
  - Need to filter window titles (e.g. email subjects)

A different data-collection framework: Hackystat

Freie Universität Berlin

Hackystat is a framework from U of Hawaii

- Related to ECG
  - Uses the same sensor structure ➔ Sensors can be reused
- Hackystat focuses on project-level multi-user data collection:
  - Central server, user identification, targets coarser-grain data
    - Uses heavy-weight web service protocol SOAP
    - Buffers data before sending them
  - Provisions for offline data collection (SensorShell as buffer)
  - Aims at understanding project dynamics, not errors
- List of sensors:
  - Ant Build, BCML, Bugzilla, CCCC, Checkstyle, Unix CLI, CPPUnit, CVS Server, DependencyFinder, Eclipse, Emacs, Emma, FindBugs, JBlanket, JBuilder, Jira, JUnit, Jupiter, LoadTest, LOCC, Microsoft Office, OpenOffice.org, PMD, SCLC, ShellLogger, Subversion, Vim, Visual Studio, XmlData
  - Sensors tend to collect a smaller set of coarser-grain events

Potential automated data collection: What to extract from video data

Direct a webcam from monitor to person:

- Presence/absense
  - Is a person sitting in front of the monitor?
- Direction of attention
  - Is the person looking towards the monitor?
  - Where are the hands?
- Focusedness
  - Is the gaze fixed in one direction for long periods of time?
- Interruption
  - Is a second person close by, not looking at the monitor?
  - Is there a lot of movement in the background?
- Pairing
  - Is a second person close by, also looking at the monitor?

Potential automated data collection: What to extract from audio data

Freie Universität Berlin

Have a microphone that records what the developer hears:

- Thought process indicators:
  - Mumbling, grumbling, finger-tapping, foot-stomping, soliloquy
- Interruptions:
  - Phone ringing
  - Voice of a second person appears nearby
- Distractions:
  - Generally high background noise
  - Sudden high background noise

Potential automated data collection: Other possible sensors

- Presence sensor:
  - Could use the short-range connect/break of Bluetooth connection to a mobile phone
    - Crucial assumption: Developer always carries it with him/her
  - Could use a pressure contact in chair
    - Realistic?
  - Could use a photoelectric barrier
    - Realistic? Practical?

- Stress sensor:
  - Detect unusually hectic movements of mouse, hectic keypresses etc.

Manual data collection: Human observers

Freie Universität Berlin

- A human observer can detect and classify all of the above-mentioned phenomena
- and probably a lot more

- Disadvantages:
  - High cost
  - Moments of inattentiveness
    - But not errors: all "intelligent" automatic sensors will make errors too
  - May influence the development process more

Data analysis: Basic ideas

Freie Universität Berlin

Modes of analysis:

1. Exploratory
   - Finding pattern candidates

2. Refining:
   - Finding many instances of the pattern, adapting the pattern definition accordingly
   - Characterizing neighboring non-instances, thus refining the pattern definition

3. Validating:
   - Re-confirming the patterns on previously unseen material
   - Perhaps characterizing their applicability in different settings

1. Exploratory data analysis: Where to start

Freie Universität Berlin

1. Existing hypotheses about error-prone situations, e.g.
   - Copy-paste-change episodes
   - Interruptions
   - Working when tired
- Analysis approach:
   - Find ways to quickly locate such episodes in large datasets
   - Collect many such episodes
   - Look for similarities, for parameters of differences and for attributes controling these parameters

2. Open-ended search for arbitrary error episodes
- Analysis approach:
   - Locate moments of error
      - e.g. based on locations of detected defects
   - Find similarities in what led to that moment

1. Exploratory data analysis: What to look for

Freie Universität Berlin

- Our goal are statements of the form
  1. In situation X, behavior Y often leads to outcome Z
  2. In situation X, behavior Y' helps avoid outcome Z
     - where Z is an error
- So we look for
  - X: Regularities in situations, classifications of situations
  - Y: Regularities in behaviors, classifications of behaviors
  - Z: Regularities in outcomes (errors, non-errors)
  - Connections from (X, Y) to outcomes
  - Differences between Y and Y'
- Our search is probably iterative and guided by hypotheses:
  - Once we see some regularity, we attempt to specify it
  - then validate the specification and refine it:
    remove cases that meet it but should not,
    include cases that do not meet it, but should.
- The search is driven by plenty of additional case examples

1. Exploratory data analysis: Episode recognition

- It is important that locating candidate case examples is automatic
  - just like data collection: Else we never find enough material
- Therefore, we need a mechanism for finding episodes of some kind in our collections of event data

- The nature of the episode specifications is unclear
  - as is the nature of the episodes themselves:
    - episodes are vague concepts,
    - episode recognizers are unreliable heuristic programs

- We obviously need a fairly general mechanism for describing episode specifications
  - must be easy to express and modify
  - must be readily executable

  ➔ an episode recognizer programming language

1. Exploratory data analysis: Episode recognition language

Conjectures about a suitable episode recognition language:

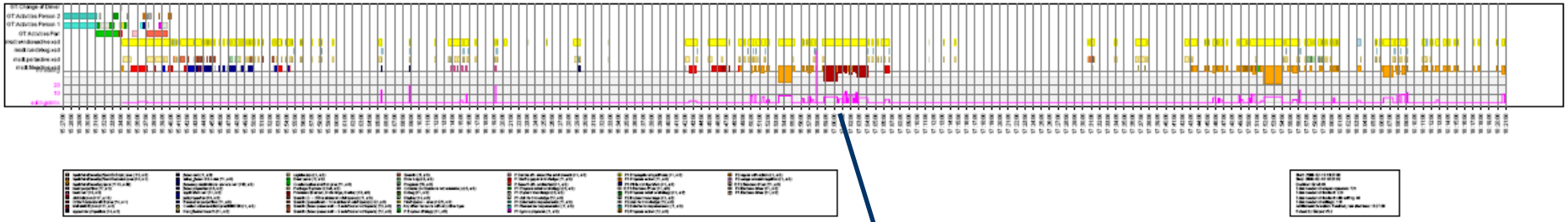
- Is procedural
  - will be easiest to learn, because that is how we think
  - may be simpler to implement
- Can express the following constructs easily
  - filtering for certain event types
  - sequence patterns of events (like regular expressions)
  - nested sub-episodes (leading to context-free languages)
  - time constraints on sequences and sub-episodes
  - propagation and comparison of time characteristics
  - backtracking from failed attempts at finding certain sub-episodes
  - reducing the search space by checking for 'beacon' events first
  - reducing the search space by applying time constraints early
- Explanation component: Recognizers provide feedback
  - Why have they found or not found certain episodes

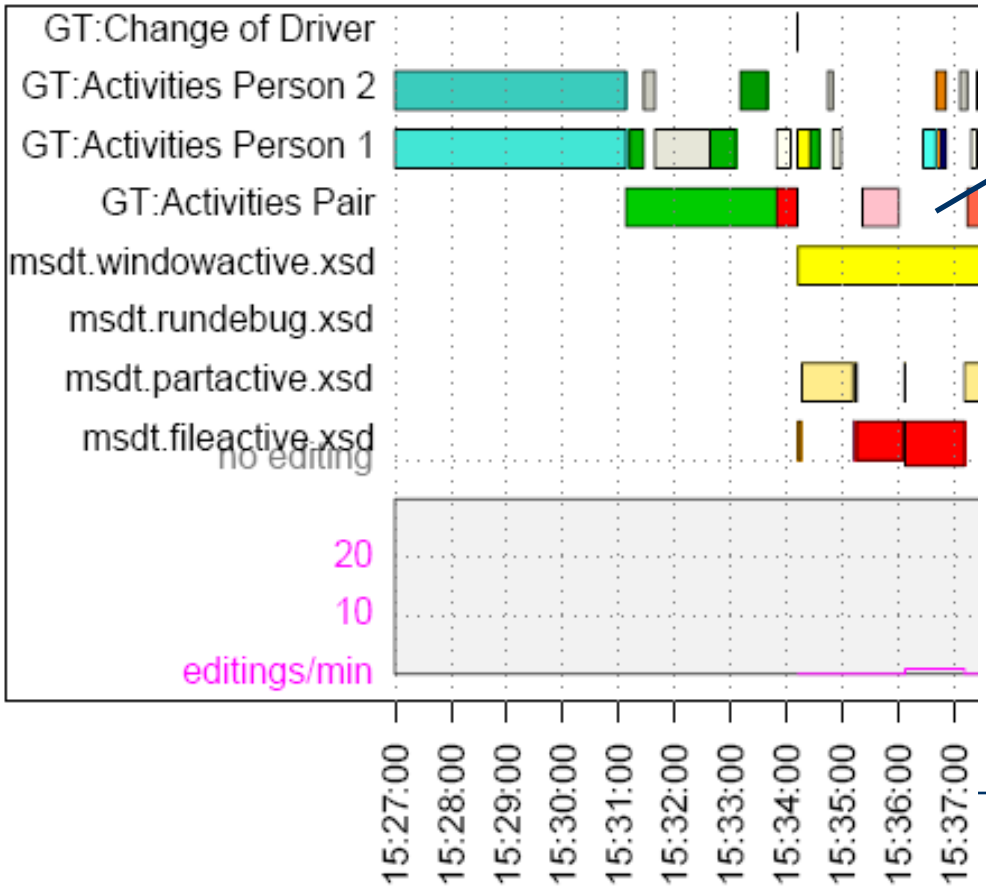
1. Exploratory data analysis: Visualization

- An alternative approach for finding interesting episodes is data visualization
- Graphical summaries potentially allow for
  - quick overview of large quantities of information
  - visual detection of regularities and irregularities
- The difficulty is finding a suitable representation of the data
- Constraints and requirements in our case:
  - visualize discrete data (events)
  - one basic dimension is either time or sequence
  - most secondary basic dimensions are discrete
    - event type, target object etc.
  - characteristics of interest are: event presence, event absence, event clustering, low/high frequence or speed

1. Exploratory data analysis: Visualization example

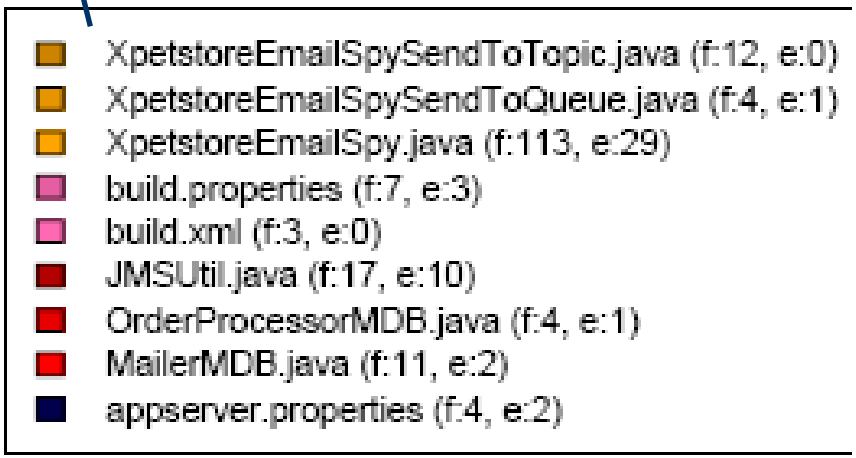
Freie Universität Berlin





A single 3-hour pair-programming session

- First 10 minutes
- Left part of legend



3. Validating data analysis: Confirm an error pattern

Freie Universität Berlin

Scientific validation of an error pattern we found:

- Apply it to freshly collected data from within the assumed application range of the pattern
- If the pattern is valid, it should work about as well here than it did in the data used for forming it
  - We may then even try broadening the application range by trying it against other *kinds* of data, too
- If the pattern does not work well, there are two possibilities:
  - Either, we have mischaracterized the application range
  - or we have even over-adapted the pattern to the data used in the research
    - This is particularly likely if the pattern is very complicated

Data analysis: Generic versus individual patterns

Freie Universität Berlin

- So far, we have assumed we are looking for patterns that are as general as possible
  - Apply to most people
  - Apply to most development situations (process, technology, etc.)
- This is likely to be difficult:
  - Few such patterns may exist
  - Pattern definitions must be restrictive to avoid false positives
  - But then many acceptable instances will be overlooked
- It is probably useful to perform the whole process individually for just one developer
  - Patterns will generalize well
  - High precision is possible without sacrificing too much recall
  - Necessary condition: Analysis must be largely automated
    - Cannot be done by researcher, must be performed by developer

Constructing help: Analysis-for-the-masses tools

Freie Universität Berlin

Providing help for programmers can happen in two forms:

1. Publish generic analysis results. Problems:
   - Will often be too vague to be really helpful
   - "Not invented here"
   - Has too little personal relation to have high impact

2. Provide data collection and analysis capabilities that are usable in the field. Problems:
   - Must be simple to install, configure, and operate
   - Analysis must be largely automatic
   - Real-time analysis is difficult
     - but batch analysis may hardly be used
   - Must be robust against unexpected phenomena

Constructing help: Analysis-for-the-masses tools (2)

What may work in this manner?

- Audio and video analysis may be difficult to get to work robustly
- In-computer ECG sensors plus good episode recognizers may, as may simple visualizations
  - The episode recognizers probably need to adapt some parameters to developer-dependent values directly from the data
    - in the visualizations, the developers can perhaps do this by hand
  - It is unclear how to solve the real-time versus batch problem

Thank you!