Course "Empirical Evaluation in Informatics"

Other methods

Prof. Dr. Lutz Prechelt
Freie Universität Berlin, Institut für Informatik

- Simulation
  - ex: P2P scalability
- Legacy data analysis
  - ex: code decay
- Literature study
  - ex: model for review effectiveness
"Empirische Bewertung in der Informatik"

Sonstige Methoden

Prof. Dr. Lutz Prechelt
Freie Universität Berlin, Institut für Informatik

- Simulation
  - Bsp: P2P-Skalierung
- Analyse vorhandener Daten
  - Bsp: Code-Verfall
- Literaturstudie
  - Bsp: Modell f. Effektivität von Durchsichten
The methods landscape


- Considers three broad categories of validation methods:
  - **Observational**: Observe a process as it unfolds, but influence it hardly or not at all
    - Case Study
  - **Historical**: Observe evidence of a process after the fact
    - Survey, Legacy Data Analysis, Literature Study
  - **Controlled**: Observe a process, with purposeful influence on the process characteristics
    - Controlled Experiment, Quasi-Experiment, Benchmarking, Simulation
Simulation

• **Approach:**
  - Formulate a model of some process or system
  - Implement that model as a program
  - Set parameters and run the model; vary parameters
  - Observe behavioral variables of interest

• **Advantages:**
  - Can produce lots of data for fairly complex situations at low cost
  - Allows to study *emergent properties* that are beyond analytical understanding and to describe the conditions under which they emerge

• **Disadvantages:**
  - It is very difficult to validate that the model is appropriate/right/valid with respect to the variables of interest
Simulation: Application areas

• Study systems that do not (yet) exist or are hard to observe
  • e.g. proposed hardware architectures
  • e.g. networks of 10000 new mobile devices
  • e.g. the whole Internet

• Study systems that evolve only slowly
  • e.g. software process simulation for project planning
    • (this is how the weather forecast is computed)

• Study effects of impossible or impractical manipulations
  • e.g. disaster studies of the Internet infrastructure
  • e.g. studying certain traffic situations for a class of potential future networks
  • e.g. what-if studies of software project dynamics
Simulation example: Scaling a P2P network


- Proposes a system called GIA that improves scaling behavior
  - [http://www.planet-lab.org/](http://www.planet-lab.org/)
Peer-to-peer networks

- Standard client/server networks have disadvantages:
  - Restricted scalability: Server overload in high-traffic situations
  - Limited availability: Single point of failure
  - Servers are expensive and require much maintenance and administration effort

- Peer-to-peer (P2P) networks try to avoid this:
  - Each node is client and server at once
  - Network is totally de-centralized: Structure, reliability, administration
  - P2P networks are typically overlay networks: a logical network layered on top of an existing network
    - (The Internet is also an overlay network)
P2P applications

- P2P networks have many applications
  - end-user file sharing (e.g. Gnutella)
  - ad-hoc networks of mobile devices
  - distributed databases
  - electronic currency (e.g. Bitcoin)
  - etc.

- We only look at file sharing here
P2P approaches

• Solution 1: Flooding (e.g. Gnutella)
  • Performs searches by recursively asking all neighbors in the network
    • Up to a maximum distance of hops (time-to-live parameter)
  • Requires O(n) steps with n nodes to find a file
  • Potentially bad scaling behavior: at high load, the network quickly becomes globally overloaded

• Solution 2: Distributed Hash Tables (DHTs)
  • De-centralized solution for mapping a filename to a host
  • Can locate the node holding a file in O(log n) steps for n nodes
  • Potentially much better scaling behavior
  • However, keyword searching is not directly supported
P2P file sharing phenomena

Mass file sharing is a rather specific P2P problem:

1. Extremely transient node participation
   - e.g. average uptime of 60 minutes in Gnutella
   - No problem for flooding; expensive for maintaining DHTs

2. Very heterogeneous node capacity (connection bandwidth)

3. Many more keyword searches than full-filename queries
   - Because exact file names are rarely known
   - This is no problem for flooding, but requires complex additional mechanisms to be accommodated by DHTs

4. Most queries are for highly replicated files
   - What is in popular demand is also offered frequently
   - This means flooding can actually be much cheaper on average than expected
     - and DHTs biggest strength is not so important at all
Improving flooding: GIA

- Replace flooding by random walk
  - e.g. consider only 1 random neighbor instead of all

- Problems:
  - Random walk is blind: does not consider that some neighbors might be more promising than others
  - If it hits an overloaded node, it will be stalled

- GIA improvement suggestions:
  - **TADAPT**: Give high-capacity nodes more neighbors
    - So that those nodes receive most queries that are best up to handle them
  - **BIAS**: Bias the random walk towards high-capacity nodes
  - **FLWCTL**: Active flow control to avoid overloaded nodes
  - **OHR**: Replication of file name lists to one-hop neighbors
    - so that high-capacity nodes can answer very many queries
Protocol variants considered

The study now investigates four variants of Gnutella-style file sharing protocols:

- **FLOOD:**
  - The standard Gnutella protocol
- **RWRT:**
  - Random walks over random topologies
- **SUPER:**
  - Discriminate supernodes and non-supernodes. Do flooding only over the supernodes
    - Not discussed here
- **GIA:**
  - All four improvements as described before

- Now set up a simulation
Setting simulation parameters

- Take distribution of node capacity from another study
  - and simplify it to make it practical
- Assign the 1000x and 10000x levels to be the supernodes (high-capacity nodes)
- Assume constant query-generation rates for each node
  - bounded by node capacity
- Topology: Set number of neighbors in range 3...128
  - average degree is 8
  - limit neighbors for low-capacity nodes
  - average resulting network diameter is 7
- Time-to-live:
  - 10 for FLOOD/SUPER,
  - 1024 for RWRT/GIA

<table>
<thead>
<tr>
<th>Capacity level</th>
<th>Percentage of nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>20%</td>
</tr>
<tr>
<td>10x</td>
<td>45%</td>
</tr>
<tr>
<td>100x</td>
<td>30%</td>
</tr>
<tr>
<td>1000x</td>
<td>4.9%</td>
</tr>
<tr>
<td>10000x</td>
<td>0.1%</td>
</tr>
</tbody>
</table>
Setting simulation parameters (2)

- Consider different (fixed) "replication factors":
  The fraction of nodes that can fulfill a given query

  - derived from separate mini-study of actual Gnutella replication and query behavior
Performance measures

The simulation considers three measures of goodness:

- **success rate:**
  - Fraction of queries that locate the file (which always exists)

- **hop count:**
  - Number of communication steps required for a query

- **delay:**
  - Time until a query returns its result

- query load = 0.1 means that in each time unit, each node issues 0.1 queries (0.1 qps/node)
Simulation results: Collapse point effect

- Is similar for all protocols
  - only earlier
- Is roughly inverse for delay

![Graph showing success rate vs. queries per second for different replication percentages (0.5% and 0.1%) and delay values.]
Simulation results: Collapse point comparison

- **GIA: N=10,000**
- **SUPER: N=10,000**
- **RWRT: N=10,000**
- **FLOOD: N=10,000**

**Legend:**
- GIA: Black diamonds
- SUPER: Green crosses
- RWRT: Pink squares
- FLOOD: Orange triangles

**Axes:**
- **Collapse Point (qps/node)**
- **Replication Rate (percentage)**
Factor Analysis

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Collapse point</th>
</tr>
</thead>
<tbody>
<tr>
<td>RWRT</td>
<td>0.0005</td>
</tr>
<tr>
<td>RWRT+OHR</td>
<td>0.005</td>
</tr>
<tr>
<td>RWRT+BIAS</td>
<td>0.0015</td>
</tr>
<tr>
<td>RWRT+TADAPT</td>
<td>0.001</td>
</tr>
<tr>
<td>RWRT+FLWCTL</td>
<td>0.0006</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Collapse point</th>
</tr>
</thead>
<tbody>
<tr>
<td>GIA</td>
<td>7</td>
</tr>
<tr>
<td>GIA – OHR</td>
<td>0.004</td>
</tr>
<tr>
<td>GIA – BIAS</td>
<td>6</td>
</tr>
<tr>
<td>GIA – TADAPT</td>
<td>0.2</td>
</tr>
<tr>
<td>GIA – FLWCTL</td>
<td>2</td>
</tr>
</tbody>
</table>

No single component is sufficient alone; only the combination of all of them makes GIA scalable.
Summary of simulation example

• Starting point:
  • A P2P file sharing system with severe scalability problems

• Improvement approach:
  • Exploit known specific characteristics, in particular heterogeneity of node capacity
  • Propose four improvements

• Evaluation approach:
  • Evaluate relative performance of the improvements in large-scale use by means of simulation
  • Obtain important simulation parameters by instrumental mini-studies
    • Using the right parameters is crucial!
Topics today

- Simulation
- Analysis of legacy data
- Literature study
Analysis of legacy data

• **Approach:**
  - Analyze existing sets of data
    - "software archaeology", now called "mining software repositories"
    - very many different approaches are possible
  - Investigate new questions or look at more data at once

• **Advantages:**
  - Can sometimes use large amounts of data with low effort
  - Questions only found later can often be answered just as well as the original ones

• **Disadvantages:**
  - If additional data is required, it may be impossible to get it
  - Data quality can be hard to assess
Analysis of legacy data example

Study question

• Do the common software engineering practices lead to code decay when a large software system is frequently changed over a long time?

• A unit of code is considered *decayed* if it is harder to change than it could be (e.g. harder than it used to be)
  • measured in terms of effort, interval and quality
Possible reasons for code decay

- Inappropriate architecture
  - Code cannot accommodate a change well
- Violation of design ideas
  - One change done in an inappropriate way makes further changes difficult
- Inadequate change environment
  - e.g. maintenance tools, organizational environment, change processes
- Programmer variability

All these are exacerbated by:
- Time pressure
- Imprecise requirements
  - Producing a sequence of changes rather than just one
Software system studied

The software of a telephone switching system

• About 100 Mio. lines of C code total
  • plus 100 Mio. lines of other files (header files, make files)
  • 50 major subsystems, 5000 directories
  • Each release consists of about 20 Mio. lines of code

• Under development since 15 years
  • About 10000 developers have worked on it
The data studied

- The full change history of the code of one subsystem
  - as recorded in the version management system
  - 100 directories, 2500 files

- Changes are described on four levels of increasing granularity
  - delta: a change to one file from one revision to the next
  - modification request (MR): description of a solution to a problem
  - initial modification request (IMR): description of a problem to be solved
  - feature: a marketable function of the system as a whole

- Change history is available for files, MRs, IMRs, features
  - 130 000 deltas, 27 000 MRs, 6 000 IMRs
  - 500 people making changes
Plausible symptoms of decay

• Excessively complex ("bloated") code
• A history of frequent changes ("code churn")
• A history of faults
• Widely dispersed modifications within one change
• Kludges
• Numerous interfaces (e.g. many entry points)

• The study defines CDIs (code decay indices) for some of these symptoms
  • The indices are based directly on the version management data
Risk factors for decay

A code unit has a higher probability of decaying if the following factors are high:

- **Size**
  - large units tend to decay more easily

- **Age**
  - although very stable code can be rather old without decay

- **Inherent complexity, e.g. due to requirements load**
  - code that must do many things or difficult things

- **Organizational churn or inexperienced developers**
  - makes design violations more likely

- **Porting or reuse**

- Like before, CDIs are defined for some of these
Results

There is code decay in this system, as indicated by:

• The span of changes increases over time
  • loess smooths w. span 0.3 (purple), 1.5 (r/p/b), 7.5 (blue)

red: decrease, statistically significant
blue: increase, statistically significant
purple: direction is not statistically significant
Results (2)

- The increase in span is accompanied by a breakdown of modularity in the code
  - each point (pin head) represents one directory
  - positions are such that dirs often changed together are close together
  - the tail indicates the position one year before
  - there are two clear clusters
    - and then some
Results (3)

- 8 years later, the large-scale modularity has almost completely disappeared

(a very good visualization!)
A side note: SeeSoft

- A tool for visualizing aspects of source code lines
  - rectangle: file, line: line, color: age of line
  - many other visualizations are available as well
Summary of legacy data example

- Analyzing legacy data allows for evaluating large-scale situations
  - extending over a long time
  - representing an immense number of events

- If done carefully, such analyses can be very credible
  - although we have not discussed the details of the arguments used here
  - External validity is difficult to obtain, though
Topics today

- Simulation
- Analysis of legacy data
- Literature study
Literature study

• Approach:
  • Review multiple published empirical studies on a similar topic
  • Draw conclusions based on the union of the data or results
    • Conclusions that are not possible from any one study alone

• Advantages:
  • Relatively low effort
  • Can in principle provide a very broad empirical basis

• Disadvantages:
  • There are rarely enough similar studies on one topic
  • Suffers from the "file drawer problem": Studies with no interesting results are usually not published at all
    • Hence the unified picture from a literature search may be biased
  • Publications often lack important detail information
Example for Literature Study

Software Development Technical Reviews (SDTRs) are
- an organizational device
- for detecting defects in software products
- at any stage of the life cycle
- and for obtaining secondary benefits
- through a two-stage process
- in which software engineers first independently inspect the software product for defects and then
- combine their efforts in a group meeting
- in which the participants adopt roles
- with the goal of producing a report
- in which all the defects agreed upon by the group are identified.

This includes inspections, reviews, walkthroughs, etc.
Goals of the study

• Long-term goal
  • Obtain a validated theory explaining the effectiveness of SDTRs in terms of a number of influencing factors

• Goal of the current work
  • Formulate a set of propositions that outline such a theory
  • Some of these propositions may be entirely unvalidated
  • Most are only partially validated
  • Therefore, the propositions describe a research program
    • Thus the title of the article
Approach of the study

Approach:

• Take an existing theory of group behavior for a class of two-stage decision tasks
  • developed by research on social psychology

• Apply this theory to SDTRs and formulate according propositions

• Review all previous studies on technical reviews to find support for the propositions -- or lack thereof
  • The article has 88 literature references
The existing theory: Tasks studied

- Group research has investigated empirically a class of problems like the following ("Lost in the desert"):  
  - Members of a group are told to imagine they were stranded in the desert with a limited number of implements available to them, e.g., knife, string, mirror, etc.
  - Each member individually is to rank these items according to their survival value
  - Then the group meets and decides on a common ranking

- A lot of experiments have been performed and a theory of decision performance has been formulated  
  - The theory can explain much of the variance observed

- The decision task resembles software inspections:  
  - First individuals make up their mind
  - Then a group makes a collective decision
  - Difference: The set of defects is more open than the usefulness ranking
The existing theory: Graphical sketch

- Terms are factors
- Arrows indicate influence
The existing theory:
Verbal description

- Performance is determined by effective group expertise
- If the group has a bad social decision process or other process flaws, its effective expertise will be below the available expertise
  - The decision process needs to select the best task expertise embedded in all of the members' individual rankings
- Available group expertise is roughly the union of the members' expertise
  - Thus, it tends to increase with group size
  - and it increases with the amount of members' individual expertise
  - and the dis-similarity of these
- Individual expertise depends on the amount of task training an individual has had
Propositions and evidence for them

- Applying the existing theory to SDTRs leads to 11 propositions
  - (We will discuss only some of them)
    - For some of these there is existing empirical evidence
    - For others, there is little or none (usually because the question has never been investigated thoroughly)

- **P1**: In SDTRs, task expertise is the dominant determinant of group performance
  - Two studies find such an effect
  - A few studies explicitly factor the influence out
  - Most studies ignore the issue
Propositions and evidence for them (2)

- **P2**: In SDTRs, decision schemes (plurality effects) influence interacting group performance
  - Decision is difficult if a defect candidate has been found by only one reviewer (i.e., there is no "plurality")
  - The frequency of this is unknown, but appears to be high

- **P3**: In SDTRs, in the absence of a plurality, interacting group performance is a positive function of process skills
  - Having different roles in the group improves performance
  - No other evidence is available

- **P4**: In SDTRs, the interacting group meeting does not improve group performance over the nominal group by discovering new defects
  - Evidence (but not strong) from various studies is available
Propositions and evidence for them (3)

- **P5**: In SDTRs, group performance is a positive function of task training
  - The only available study reports a 90 percent user defect reduction after training in software reading techniques

- **P6**: In SDTRs, the performance/size relationship is a function of task expertise.
  - No evidence is available

- **P7**: In SDTRs, above a critical limit, performance *declines* with increasing group size
  - There is evidence for process loss from a number of studies
  - But there is no direct support for the proposition
Propositions and evidence for them (4)

- **P8**: In SDTRs, the performance advantage of an interacting group over a nominal group is a function of the level of false positives discovered by individuals
  - There is evidence that meetings do discriminate false positives from true defects
  - But no formal test of the proposition has been performed

- **P9**: In SDTRs, an expert pair performs the discrimination task as well as any larger group
  - One study reports that groups of 4 were not significantly better than groups of 2
  - But no formal test of the proposition has been performed
Propositions and evidence for them (5)

• P10: In SDTRs, nominal groups outperform alternatives (1 reviewer, best reviewer from group, review meeting) at the discovery task
  • Several studies confirm this
  • BTW: Similar results exist for brainstorming
    • Prepared individuals result in more overall ideas compared to only a brainstorming meeting

• P11: In SDTRs, the defect discovery performance/size relationship for nominal groups is a function of task expertise
  • Like P7, this has not yet been studied much
Conclusions: Consequences for research

- Several propositions need to be validated
  - This is a research program that can now more clearly be understood than before

- Reading technology research should continue
  - Roles, checklists, scenarios, perspectives, ...
  - It can make individual reviewers more effective and can improve the efficiency of larger group sizes

- We need research for understanding review expertise
  - So that we can develop proper reviewer trainings
  - Because reviewer task expertise is the single most important factor for review effectiveness
Conclusions: Consequences for practice

- For defect detection, one may be able to substitute expertise by larger numbers of reviewers.

- However, too-large groups may produce insufficient motivation in the reviewers.
  - One may try incentive systems to overcome this.

- Defect discrimination meetings should be abandoned.
  - Unless false positives are frequent or harmful.
  - Or be replaced by a single expert review-reviewer.
Summary of literature study example

- Re-using what is available in the literature can be a **cost-efficient** way of obtaining empirical information
  - Note there are specific, well-defined methods for doing this: systematic literature review (SLR), mapping study, meta-analysis, thematic synthesis, etc.

- In particular, by considering more and diverse work, we may be able to **obtain more understanding** than any single study ever could

- A unified view of multiple studies can sometimes **resolve credibility or relevance problems**

- In the current state of software engineering research, **theory-building** should probably start by means of literature studies
Summary: Other methods

- There are more approaches for empirical evaluation than those we have covered in a full two-hour lecture

- Examples are
  - Simulation,
    - Example: P2P query scaling behavior (GIA) study
  - Analysis of legacy data, and
    - Example: code decay study
  - Literature studies
    - Example: theory of review effectiveness study

- Each has its specific strengths
Thank you!