Dipartimento di Scienze Teoriche e Applicate
Università degli Studi dell'Insubria

Software Quality Evaluation via Static Analysis and Static Measurement: an Industrial Experience

Luigi Lavazza

Università degli Studi dell'Insubria, Varese, Italy

luigi.lavazza@uninsubria.it

The 15th International Conference on Software Engineering Advances
October 18 to October 22, 2020 - Porto, Portugal

Luigi Lavazza

- Luigi Lavazza is associate professor of Computer Science at the University of Insubria at Varese, Italy. Formerly he was assistant professor at Politecnico di Milano, Italy. Since 1990 he cooperates with the Software Engineering group at CEFRIEL, where he acts as a scientific consultant in digital innovation projects.
- His research interests include: Empirical software engineering, software metrics and software quality evaluation; Software project management and effort estimation; Software process modeling, measurement and improvement; Open Source Software.
- He was involved in several international research projects, and he also served as reviewer of EU funded projects.
- He is co-author of over 170 scientific articles, published in international journals, or in the proceedings of international conferences or in books.
- He has served on the PC of a number of international Software Engineering conferences; from 2013 to 2018 he was the editor in chief of the IARIA International Journal On Advances in Software.
- He is a IARIA fellow since 2011

Luigi Lavazza: research interests

- Empirical software engineering
  - Evaluation of estimation models' accuracy
- Software metrics and software quality evaluation
- Software project management and effort estimation
- Software process modeling, measurement and improvement
- Open Source Software.

Paper abstract

- Business organizations need to evaluate the quality of the code delivered by suppliers.
- In this paper, we illustrate an experience in setting up and using a toolset for evaluating code.
- The selected tools perform static code analysis and static measurement, and provide evidence of possible quality issues.
- Code inspections were carried out to spot false positives.
- The combination of automated analysis and inspections proved effective: several types of defects were identified.
- Based on our findings, the business company was able to learn what are the most frequent and dangerous types of defects that affect the acquired code: currently, this knowledge is being used to perform focused verification activities

The context

- The evaluation addressed two B2C portals, coded almost entirely in Java.

	Portal 1	Portal 2
Number of files	1507	280
LLOC	100375	37467
LOC	202249	55934
Number of Classes	1158	247
Number of Methods	13351	5370

Goals

- Evaluating the quality of the products, highlighting weaknesses and improvement opportunities.
- It was deemed important to spot the **types** of the most frequently recurring issues, rather than finding all the actual defects and issues.

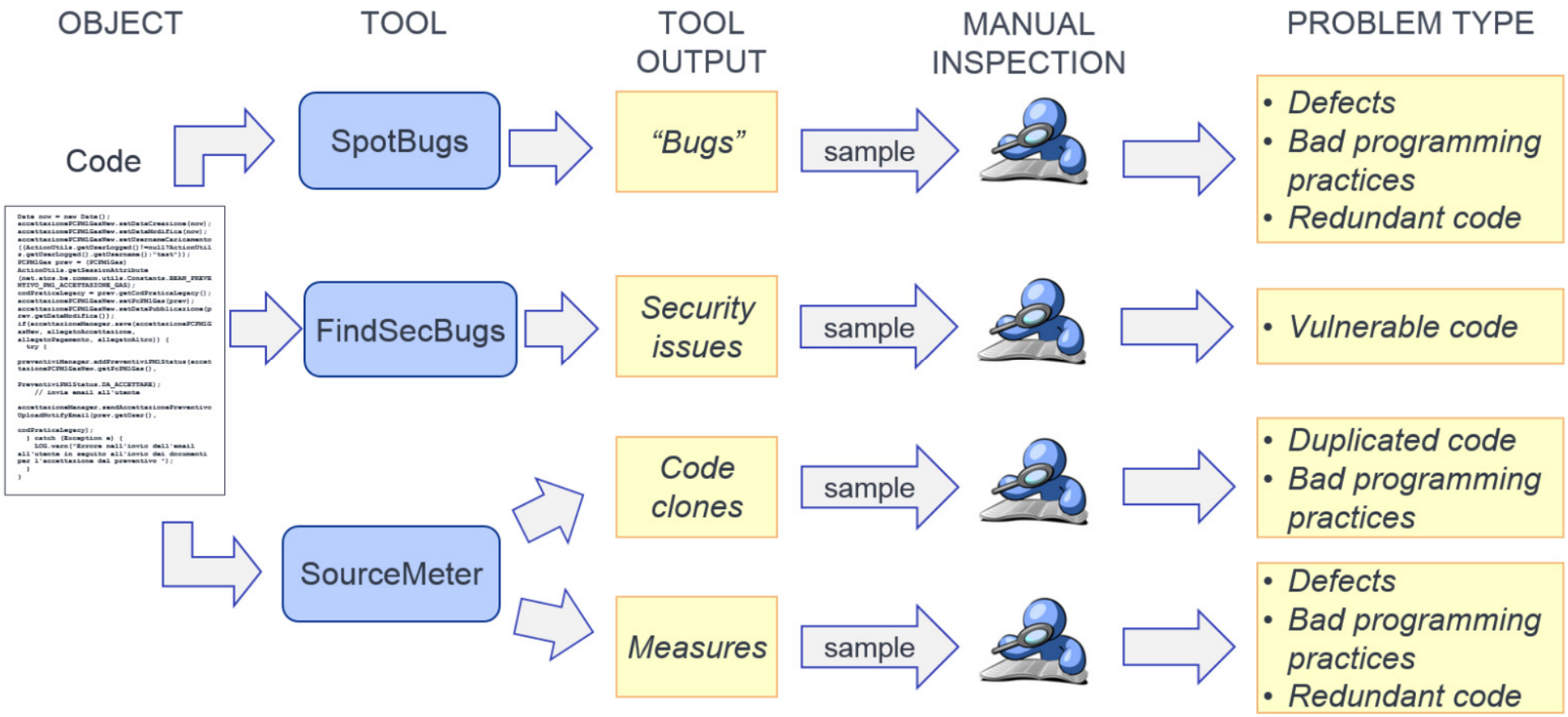
Tools

- Open-source (or free to use) software was to be preferred.
- We looked for tools that can
  - Detect bad programming practices, based on the identification of specific code patterns.
  - Detect bad programming practices, based on code measures (e.g., methods too long, classes excessively coupled, etc.).
  - Detect duplicated code.
  - Identify vulnerabilities.

Tools

Purpose	Tool	Main features
Identify defects	SpotBugs	Static analysis is used to identify code patterns that are likely associated to defects.
Collect static measures	SourceMeter	Static measurement is applied at different granularity levels (class, method, etc.) to provide a variety of measures.
Detect code clones	SourceMeter	Structurally similar code blocks are identified.
Identify security issues	FindSecBugs	A plug-in for SpotBugs, specifically oriented to identifying vulnerable code.

The evaluation process: problem detection



Warnings issued by SpotBugs (by confidence)

Metric	Portal 1		Portal 2	
	Warnings	Density	Warnings	Density
High Confidence	68	0.07%	50	0.13%
Medium Confidence	774	0.77%	502	1.34%
Low Confidence	824	0.82%	420	1.12%
Total	1666	1.66%	972	2.59%

Warnings issued by SpotBugs (by type)

Warning Type	Portal 1		Portal 2	
	Number	Percentage	Number	Percentage
Bad practice	73	4.38%	81	8.33%
Correctness	91	5.46%	30	3.09%
Experimental	0	0.00%	1	0.10%
Internationalization	16	0.96%	39	4.01%
Malicious code vulnerability	496	29.77%	316	32.51%
Multithreaded correctness	28	1.68%	1	0.10%
Performance	74	4.44%	143	14.71%
Security	631	37.88%	215	22.12%
Dodgy code	257	15.43%	146	15.02%
Total	1666	100%	972	100%

Warnings issued by SpotBugs (by rank)

- Rank levels:
  - “scariest”(1 ≤rank≤ 4)
  - “scary” (5 ≤rank≤ 9)
  - “worrying” (10 ≤rank≤ 14)
  - “of concern” (15 ≤rank≤ 20).

Rank	1	6	7	8	9	10	11	12	14	15	16	17	18	19	20
Portal 1	24	9	1	42		12	39	21	2	604	17	129	562	82	122
Portal 2	10			6	1	7	8	27	18	191	10	59	401	66	168

These warnings were manually inspected

Results of inspecting SpotBugs warnings

- Our inspections revealed several code quality problems:
  - The existence of problems matching the types of warning issued by SpotBugs was confirmed.
  - Some language constructs were not used properly.
    - E.g., class **`Boolean`** was incorrectly used instead of **`boolean`**; objects of type **`String`** were used instead of **`boolean`** values; etc.
  - We found redundant code, i.e., some pieces of code were unnecessarily repeated, even where avoiding code duplication—e.g., via inheritance or even simply by creating methods that could be used in different places—would have been easy and definitely convenient.
  - We found some pieces of code that were conceptually incorrect. The types of defect were not of any type that a static analyzer could find, but were quite apparent when inspecting the code.

Results of inspecting SpotSecBugs warnings

- We inspected the most serious warnings:
  - the only “scary” warning
  - all the warnings at the highest rank of the level “troubling” (rank 10)

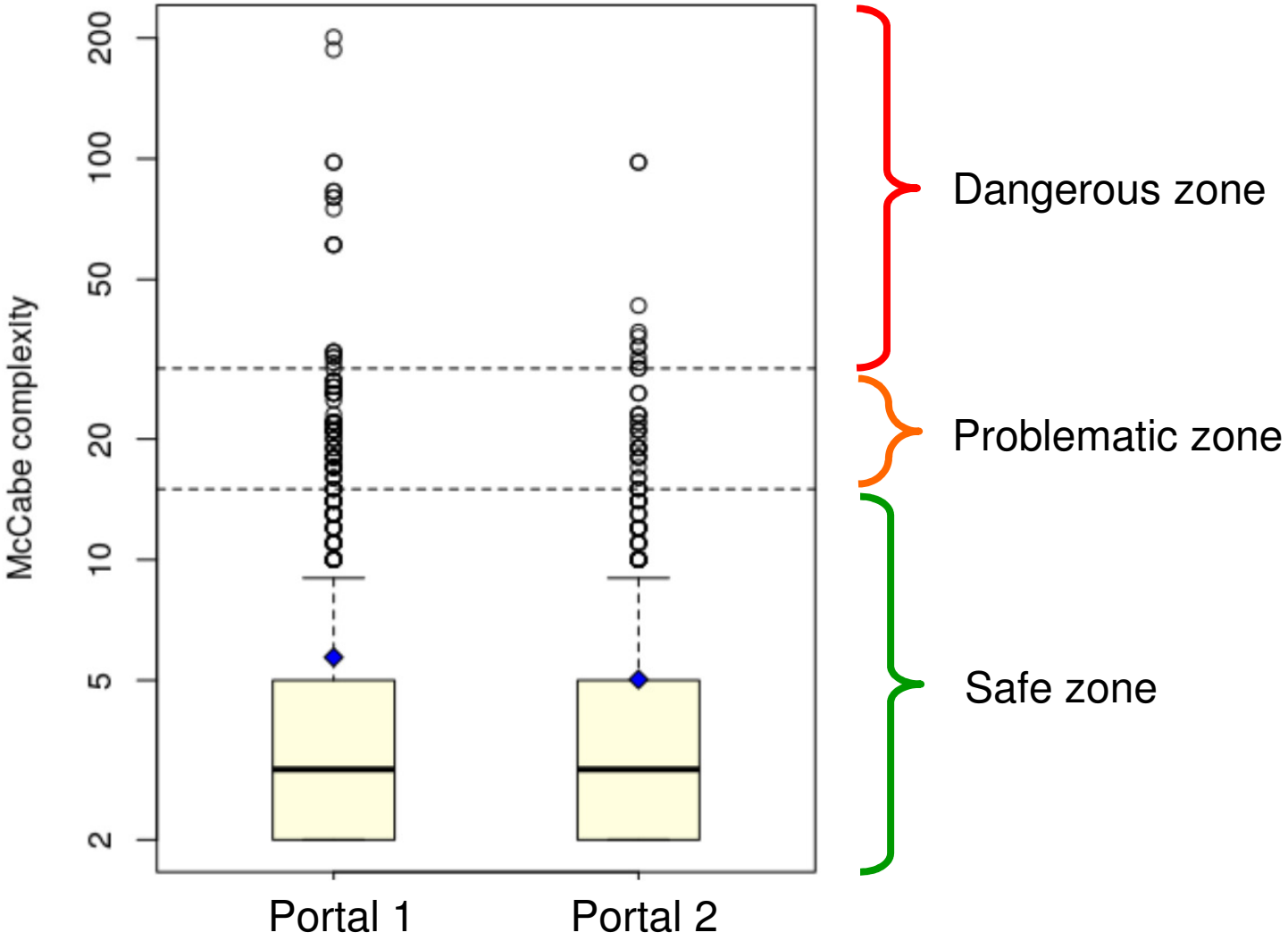
Rank	Type	Occurrences Portal 1	Portal 2
7	HTTP response splitting vulnerability	1	–
10	Cipher with no data integrity	4	2
10	ECB mode is insecure	4	2
10	URL Connection Server-Side Request Forgery and File Disclosure	1 –	– –
10	Unvalidated Redirect	2	–
10	Request Dispatcher File Disclosure	–	1

- We found that all the warnings pointed to code that had security problems.
- In many cases, SpotBugs documentation provided quite straightforward ways for correcting the code.

Inspection of code elements having measures beyond threshold

- We inspected code elements having measures definitely out of the usually considered safe ranges.
- We considered the following measures as possibly correlated with problems:
  - McCabe complexity
  - Logical Lines of Code
  - Response for Class (RFC).
- We also looked at Coupling Between Objects, Lack of Cohesion in Methods and Weighted Method Count, but these measures turned out to provide no additional information
  - i.e., they pointed to the same classes or methods identified as possibly problematic by the measures above

McCabe complexity



Other code measures

- When considering size, we found several classes featuring over 1000 LLOC;
  - the largest class contained slightly less then 6000 LLOC.
- When considering RFC, we found 12 classes having RFC greater than 200.
- The class with the highest RFC (709) was also the one containing the method with the greatest McCabe complexity.
- The biggest class contained the second most complex method.
- These results were not surprising, since it is known that several measures are correlated.

Inspections concerning out-of-range elements

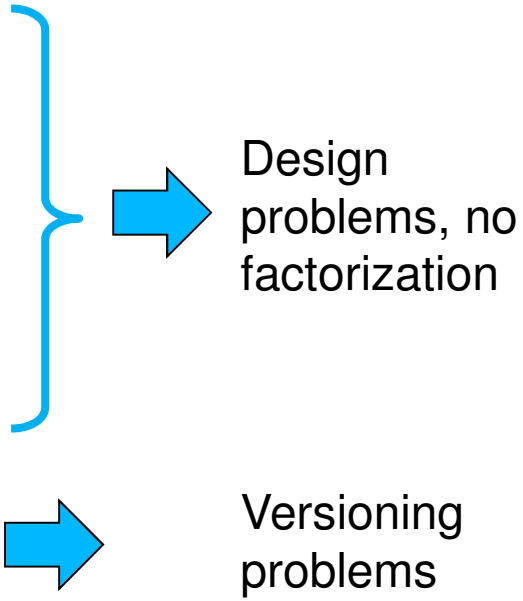
- Inspections revealed that the classes and methods featuring excessively high values of LLOC, RFC and McCabe complexity were all affected by the same problem.
- The considered code had to deal with several types of services, which where very similar under several respects, although each one had its own specificity.
- The analyzed code ignored the similarities among the services to be managed, so that the code dealing with similar service aspects was duplicated in multiple methods.
- The code could have been organized differently using basic object-oriented features: a generic class could collect the features that are common to similar services, and a specialized class for every service type could take care of the specificity of different service types.

Inspections concerning out-of-range elements

- In conclusion, by inspecting code featuring unusual static measures, we found design problems, namely inheritance and late binding were not used where it was possible and convenient

Inspection of duplicated code

- By inspecting the duplicated code spotted by SourceMeter we found three types of duplications:
  a) Duplicates within the same file. That is, the same ode was found in different parts of the same file (or the same class, often).
  b) Duplicates in different files. That is, the same code fragment was found in different files (of the same portal).
  c) Duplicates in different portals. That is, the same code fragment was found in files belonging to different portal.



Inspection of duplicated code

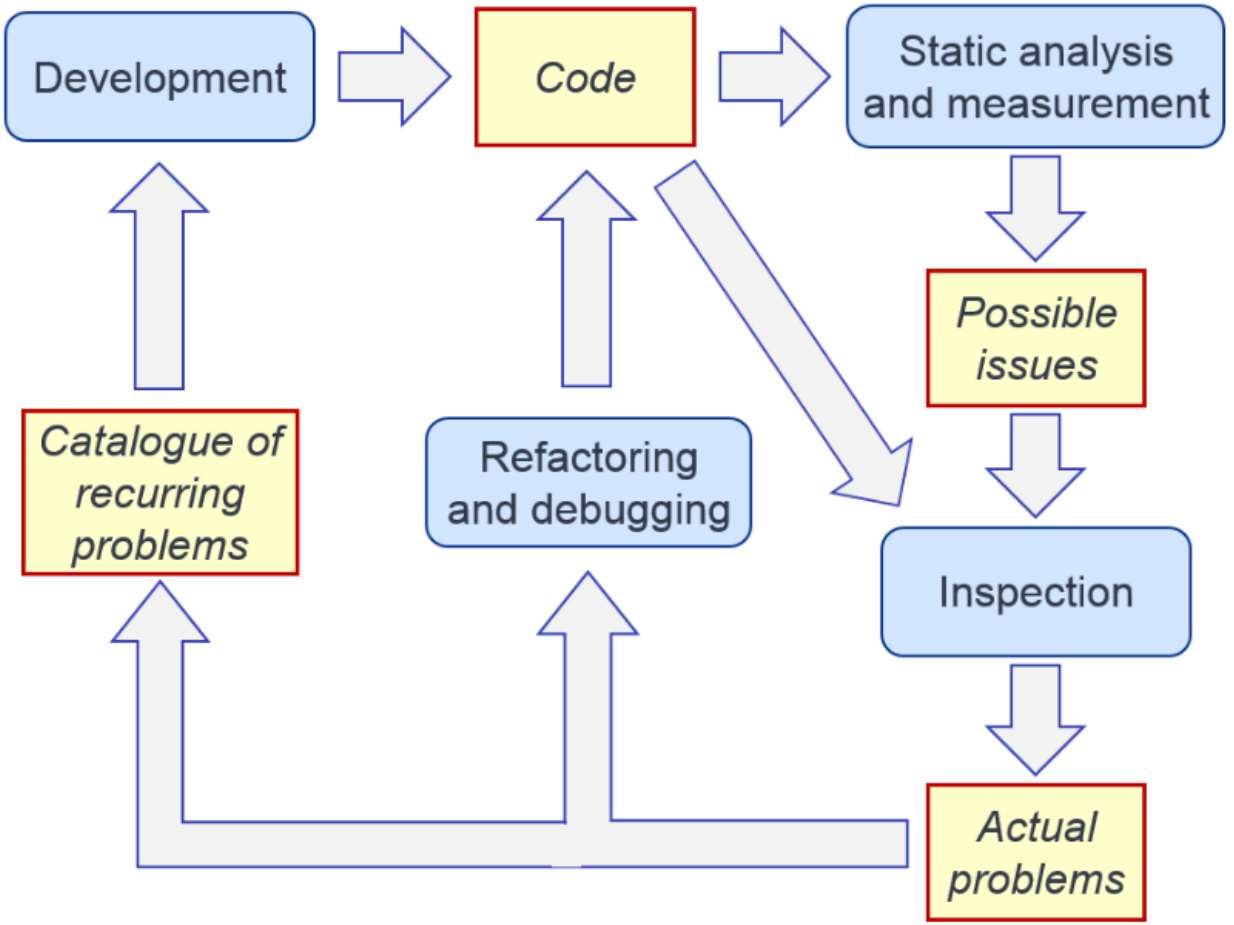
- Static measures revealed a general problem with the design of code, but were not able to indicate precisely which parts of the code could be factorized.
- On the contrary, duplicated code detection was quite effective in identifying all the cases where code could be factorized, with little need of inspecting the code.

☞ Code clone detection added some value to inspections aiming at understanding the reasons for ‘out of range’ measures.

Suggestions for improving the development process

- Two not exclusive approaches are possible.
- **Evaluation of code**
  - the toolset can be used to evaluate the released code
  - It would be advisable that developers verify their own code via SpotBugs and SourceMeter even before releasing it
- **Prevention**
  - The practice of issue identification and verification leads to identifying the most frequently recurring types of problems.
  - It is therefore possible to compile a catalogue of the most frequent and dangerous problems: programmers could be instructed to carefully avoid such issues.
    - This could imply teaching programmers specific techniques and good programming practices.

Suggested Development Process



Conclusions

- Tool-driven inspections uncovered several types of defects.
- In the process, the tools identified problems of inherently different nature, hence it is advisable to use both types of tools.
- Based on our findings, the business company was able to learn what are the most frequent and dangerous types of defects that affect the acquired code: this knowledge is being used to perform focused verification activities.
- The proposed approach and toolset can be useful in several contexts where code quality evaluation is needed.
- Noticeably, the proposed approach can be used in different types of development process, including agile processes.

Thanks for your attention!

Questions?