Static Source Code Analysis in Pattern Recognition, Performance Optimization and Software Maintainability

Dénes Bán

Department of Software Engineering
University of Szeged

Szeged, 2017

Supervisor:

Dr. Rudolf Ferenc

Summary of the Ph.D. Thesis

University of Szeged
Ph.D. School in Computer Science

Introduction

Software rules the world.

As true as this statement already was decades ago, it rings even truer now. When the proportion of the U.S. population using *any* kind of embedded system – including phones, cameras, watches, entertainment devices, etc. – went from its 2011 estimate of 65% to 95% by 2017 purely through phone ownership. When we are at a point where we can even *talk* about "smart cities", let alone build them. When – according to Cisco – connected devices have been outnumbering the population of Earth by a ratio of at least 1.5 since 2015.

This is just exacerbated by the host of embedded systems most people never even consider. An everyday routine contains household appliances like microwave ovens and refrigerators, heating or cooling our spaces, starting or stopping our vehicles; the list goes on. A modern life in this era involves countless hidden, invisible processors, along with the visible ones we have all got so used to. And we have not even mentioned critical applications like flight guidance, keeping patients alive and well in hospitals, or operating nuclear power plants.

All of those need software to run. And all that software needs to be written by someone. There is no question about either the growth of the software industry or the significant acceleration of said growth. The only question is whether or not we can actually keep up.

Having established the importance of software development, we can focus on arguably the two most important factors for its success: maintainability and performance. Software systems spend the majority of their lifetime in the maintenance phase, which, on average, can account for 60% of the total costs. Of course, maintaining a codebase does not only mean finding and fixing program faults, as enhancements and constantly changing requirements are far more common. This means that for an efficient maintenance life cycle, a software product has to be quickly analysable, modifiable and testable – among other characteristics.

Just as critical is the issue of software performance and energy efficiency. A software product that fails to meet its performance objectives can be costly by causing schedule delays, cost overruns, lost productivity, damaged customer relations, missed market windows, lost revenues, and a host of other difficulties. Not to mention that the great amounts of energy consumed by large-scale computing and network systems, such as data centers and supercomputers, have been a major source of concern in a society increasingly reliant on information technology.

There are, however, many obstacles in the way of clean, maintainable and high-performance software. Time constraints of the ever-expanding market often make it appear infeasible to consider design best practices when these considerations would push back release times. Similarly, haste and an unwillingness to put in extra effort in advance is what seems to lead to antipatterns and code duplications, harming quality in the long run. Also, inadequate accessibility, tools, and developer support could significantly hinder the full utilization of today's performance optimization opportunities, such as specialized hardware platforms (e.g., GPGPU, DSP, or FPGA) and their corresponding compilers.

Our work aims to assist in both of these areas. Our goals are to:

I. **Draw attention to the importance of the maintenance phase, and illustrate its assets and risks by highlighting the objective, tangible effect design patterns and antipatterns can have on software maintainability; and**

II. **Help developers more easily utilize modern accelerator hardware and increase performance by creating an easily usable and extendable methodology for building static platform selection models.**

I Empirical validation of the impact of source code patterns on software maintainability

The contributions of this thesis point are related to software maintainability and its connection to static source code patterns.

The Connection between Design Patterns and Maintainability

To study the impact design patterns have on software maintainability, we analyzed over 700 revisions of JHotDraw 7 [9]. We chose it especially for its intentionally high pattern density and the fact that its pattern instances were all so thoroughly documented that we could use a *javadoc*-based text parser for pattern recognition. This led to a virtual guarantee of precision regarding the matched design pattern instances, which we paired with the utilization of an objective maintainability model [1]. An inspection of the revisions where the number of pattern instances increased revealed a clear trend of similarly increasing maintainability characteristics. Furthermore, comparing pattern density to maintainability as a whole – depicted in Figure 1 – resulted in a 0.89 Pearson correlation coefficient, which suggested that design patterns do indeed have a positive effect on maintainability.

Figure 1: The tendencies of pattern line density and maintainability

The Connection between Antipatterns and Maintainability

As for the impact of antipatterns, we selected 228 open-source Java systems, along with 45 revisions of the Firefox browser application written in C++ as our subjects for two distinct experiments. In both cases, we matched 9 different, widespread antipattern types through metric thresholds and structural relationships – with additional antipattern densities for C++ [7]. Maintainability calculation remained the same for the Java systems, while the evaluation of Firefox required a C++ specific custom quality model, and we also implemented versions of the "traditional" MI metric [8]. The results of both studies confirm the detrimental effect of antipatterns.

The trend line of the maintainability value for the Java systems – sorted in the descending order of the antipatterns they contain – shows clear improvement, as illustrated in Figure 2. The overall Spearman correlation coefficient between antipatterns and maintainability for Java was -0.62.

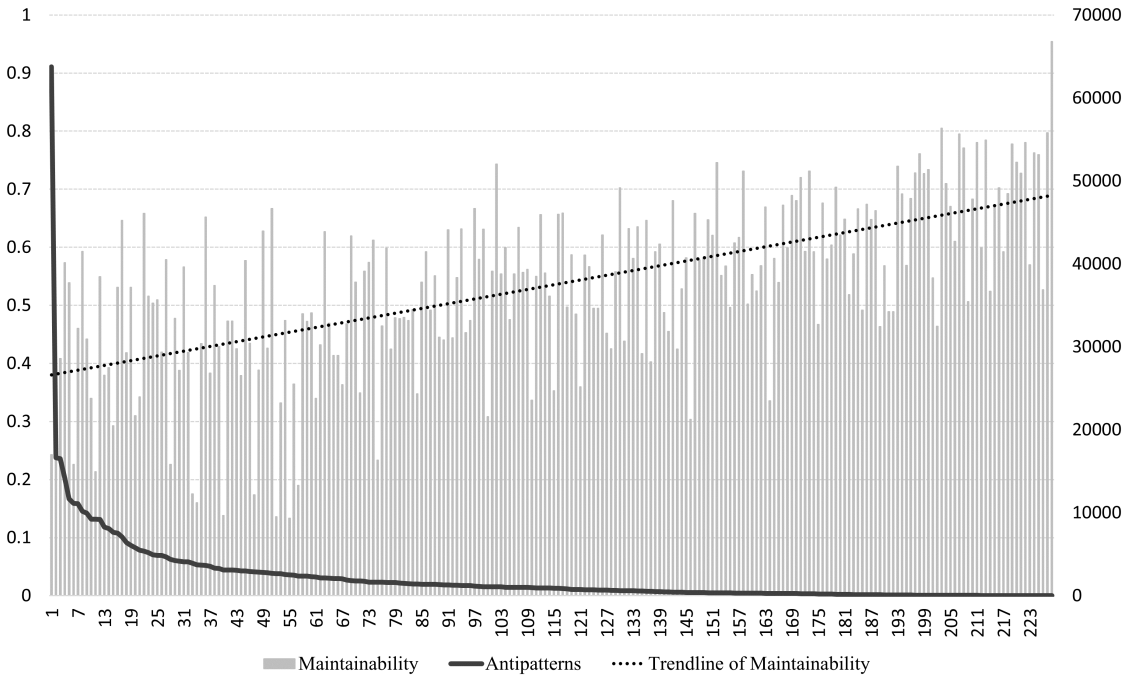

Figure 2: The trend of maintainability in the case of decreasing antipatterns (Java)

On the other hand, the C++ analysis provided values for both absolute antipattern instances and antipattern densities. We highlight the correlations between the overall antipatten sum and density and our final Maintainability measure as these represent most closely the total effects of antipatterns on maintainability. The corresponding values are -0.658 and -0.692 for Pearson, and -0.704 and -0.678 for Spearman correlation, respectively. The full set of Spearman correlations between the various antipattern/maintainability measure pairs is shown in Table 1.

Another interesting result is that using antipattern instances as predictors for maintainability estimation produced models with precisions ranging from 0.76 to 0.93.

The Connection between Antipatterns and Program Faults

In addition, our Java experiment seeks to connect the presence of antipatterns to program faults (or bugs) through the PROMISE open bug database [12]. The study of the 34 systems (from among the 228 Java systems mentioned above) that had corresponding bug information revealed a statistically significant Spearman correlation of 0.55 between antipattern instances and bugs. Moreover, antipatterns yielded a precision of 67% when predicting bugs, being notably above the 50% baseline, and only slightly below the 71.2% of more than five times as many raw static source code metrics, – shown in Table 2 – thereby demonstrating their applicability.

The Author's Contributions

For the research linked to design patterns, the author's main contributions were the implementation of the pattern mining tool, calculating the relevant source code metrics, manually validating

Antipattern	MI	MI_2	MI*	MI_2*	**Analysability**	**Modifiability**	**Modularity**	**Reusability**	**Testability**	**Maintainability**
FE	−.985**	−.985**	−.853**	−.809**	−.852**	−.749**	−.161	−.370*	−.806**	−.652**
FE_{DENS}	−.535**	−.535**	−.257	−.258	−.440**	−.532**	−.550**	−.595**	−.561**	−.609**
LC	−.757**	−.757**	−.936**	−.920**	−.755**	−.538**	.224	−.044	−.572**	−.381**
LC_{DENS}	−.542**	−.542**	−.777**	−.854**	−.517**	−.276	.372*	.123	−.307*	−.133
LCC	−.985**	−.985**	−.873**	−.839**	−.871**	−.754**	−.131	−.354*	−.811**	−.651**
LCC_{DENS}	−.482**	−.482**	−.213	−.258	−.439**	−.543**	−.590**	−.655**	−.550**	−.636**
LCD	−.731**	−.731**	−.484**	−.445**	−.509**	−.551**	−.303*	−.365*	−.590**	−.528**
LCD_{DENS}	−.626**	−.626**	−.355*	−.344*	−.373*	−.431**	−.299*	−.318*	−.474**	−.428**
LF	−.991**	−.991**	−.849**	−.800**	−.902**	−.821**	−.242	−.453**	−.866**	−.728**
LF_{DENS}	−.874**	−.874**	−.622**	−.608**	−.824**	−.837**	−.500**	−.671**	−.876**	−.821**
LPL	−.952**	−.952**	−.926**	−.856**	−.851**	−.696**	.019	−.219	−.750**	−.560**
LPL_{DENS}	−.904**	−.904**	−.707**	−.670**	−.715**	−.654**	−.184	−.338*	−.708**	−.580**
RB	−.976**	−.976**	−.819**	−.793**	−.829**	−.734**	−.167	−.375*	−.794**	−.646**
RB_{DENS}	−.911**	−.911**	−.706**	−.728**	−.735**	−.674**	−.219	−.396**	−.730**	−.614**
SHS	−.985**	−.985**	−.820**	−.768**	−.884**	−.827**	−.291	−.487**	−.871**	−.747**
SHS_{DENS}	−.907**	−.907**	−.694**	−.698**	−.787**	−.773**	−.370*	−.538**	−.812**	−.726**
SUM	−.978**	−.978**	−.806**	−.754**	−.847**	−.786**	−.250	−.444**	−.834**	−.704**
SUM_{DENS}	−.895**	−.895**	−.674**	−.641**	−.732**	−.726**	−.332*	−.476**	−.768**	−.678**
TF	−.945**	−.945**	−.746**	−.704**	−.785**	−.746**	−.277	−.445**	−.796**	−.681**
TF_{DENS}	−.843**	−.843**	−.595**	−.543**	−.675**	−.704**	−.378*	−.484**	−.739**	−.665**

Table 1: Spearman correlations between antipatterns and maintainability (C++)

Method	TP Rate	FP Rate	Precision	Recall	F-Measure
Antipatterns	0.658	0.342	0.670	0.658	0.653
Metrics	0.711	0.289	0.712	0.711	0.711
Both	0.712	0.288	0.712	0.712	0.712

Table 2: The results of the bug prediction experiments

the revisions that introduced patterns changes, and reviewing the related literature. On the other hand, the entire antipattern-related research was the author's own work including the preparation and analysis of the subject systems, implementing and extracting the relevant static source code metrics, calculating the corresponding maintainability values, – along with overseeing the creation of the C++ specific quality model – implementing the antipattern mining tool and extracting antipattern matches, considering program fault information, as well as conducting and evaluating the empirical experiments. The publications related to this thesis point are:

- Péter Hegedűs, **Dénes Bán**, Rudolf Ferenc, and Tibor Gyimóthy. Myth or Reality? Analyzing the Effect of Design Patterns on Software Maintainability. In *Advanced Software Engineering & Its Applications (ASEA 2012), Jeju Island, Korea, November 28 – December 2*, pages 138–145, CCIS, Volume 340. Springer Berlin Heidelberg, 2012.
- **Dénes Bán** and Rudolf Ferenc. Recognizing Antipatterns and Analyzing their Effects on Software Maintainability. In *14th International Conference on Computational Science and Its Applications (ICCSA 2014), Guimarães, Portugal, June 30 – July 3*, pages 337–352, LNCS, Volume 8583. Springer International Publishing, 2014.
- **Dénes Bán**. The Connection of Antipatterns and Maintainability in Firefox. In *10th Jubilee Conference of PhD Students in Computer Science (CSCS 2016), Szeged, Hungary, June 27 – 29*, 2016.
- **Dénes Bán**. The Connection of Antipatterns and Maintainability in Firefox. Accepted for publication in the 2016 Special Issue of Acta Cybernetica (extended version of the CSCS 2016 paper above). 20 pages.

II A hardware platform selection framework for performance optimization based on static source code metrics

The contributions of this thesis point are related to software performance and an automatic hardware platform selection methodology.

Qualitative Prediction Models

The goal of our qualitative research was to develop a highly generalizable methodology for building prediction models that are capable of automatically determining the optimal hardware platform (regarding execution time, power consumption, and energy efficiency) of a given program, using static information alone. To achieve this, we collected a number of benchmark programs that contained algorithms implemented for each targeted platform. These benchmarks are necessary for the training of the models, because executing and measuring the different versions of their algorithms on their respective platforms is what highlights their differences in performance. Then, we extracted several low-level source code metrics from these algorithms that would capture their characteristics and would become the predictors of our models. We also developed a universal solution capable of performing accurate cross-platform time, power, and energy consumption measurements for this purpose [11]. Lastly, we applied various machine learning techniques to build the proposed prediction models. A brief empirical validation showed the theoretical usefulness of such models, – demonstrating perfect accuracy at times – but the real result of this study is the methodology – shown in Figure 3 – that led to them, and could also enable larger scale experiments. The steps of applying a model to a new subject system (unknown to the trained model) are depicted in Figure 4.

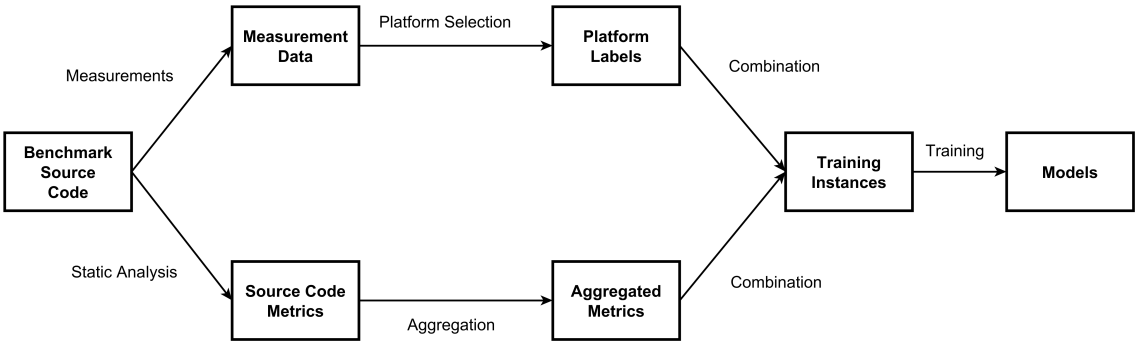

Figure 3: The main steps of the model creation process

Quantitative Prediction Models

Building on the above-mentioned previous work, we extended our methodology to create models that are quantitative, i.e., able to estimate expected improvement ratios instead of just the

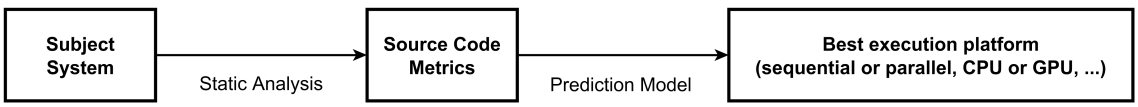

Figure 4: Usage of a previously built model on a new subject system

best platform. Additional refinements included a significantly augmented set of source code metrics, more precise metric extraction, – separating the cores of the benchmark algorithms, a.k.a. "kernels" – adding new benchmarks, and introducing the FPGA platform as a possible target. The improvement prediction accuracy values we achieved for the full systems and the separated kernel regions are shown in tables 3 and 4, respectively. The three header layers represent how we aggregated dynamic measurements, which platform we targeted, and the measured aspect (**T**ime, **P**ower, **E**nergy). As improvement ratios are continuous, they were predicted using regression algorithms (the top six rows), as well as the previously used classification algorithms paired with a discretization filter – using 5 bins in the middle five rows, and 3 bins in the bottom five. While the regression models rarely proved encouraging, 94% of the classification models outperformed either random choice or the established ZeroR baseline by at least 5% (up to 49%, at times).

Maintainability Changes of Parallelized Implementations

The benchmark source code already available to us also made it possible to look for maintainability changes between the CPU-based original (sequential) and the accelerator hardware specific (parallel) algorithm versions. The only other requirement was for us to compute the same source code metrics for the parallel variants as well, as a suitable maintainability model would be reused from a previous study [13]. The results of comparing the overall maintainability values of the two source code variants clearly indicated that parallelized implementations had significantly lower maintainability compared to their sequential counterparts, as shown in Table 5. This, however, did not appear nearly as strongly in the core algorithms (kernels) themselves, – as shown in Table 6 – which suggests that deterioration is mainly due to the added infrastructure (boilerplate code) introduced by the accelerator specific frameworks they employ.

The Author's Contributions

The author led the effort of collecting and preparing the benchmarks for both static analysis and dynamic measurements. He implemented, extracted, and aggregated both the original and the extended set of source code metrics, and he compiled the final machine learning tables. He formalized and performed the actual experiments, and analyzed their results. The maintainability comparison of benchmark versions and its evaluation is also the author's work. The publications related to this thesis point are:

- **Dénes Bán**, Rudolf Ferenc, István Siket, and Ákos Kiss. Prediction Models for Performance, Power, and Energy Efficiency of Software Executed on Heterogeneous Hardware. In *13th IEEE International Symposium on Parallel and Distributed Processing with Applications (IEEE ISPA-15), Helsinki, Finland, August 20 – 22*, pages 178–183, IEEE Trustcom/BigDataSE/ISPA, Volume 3. IEEE Computer Society Press, 2015.

- **Dénes Bán**, Rudolf Ferenc, István Siket, Ákos Kiss, and Tibor Gyimóthy. Prediction Models for Performance, Power, and Energy Efficiency of Software Executed on Heterogeneous Hardware. Submitted to the Journal of Supercomputing, Springer Publishing (extended version of the IEEE ISPA-15 paper above). 24 pages.

	Single									With CPU									All								
	CPU			GPU			FPGA			CPU			GPU			FPGA			CPU			GPU			FPGA		
Algorithm	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**
ZeroR	0.65	0.45	0.65	0.39	0.45	0.42	0.65	0.39	0.65	0.65	0.45	0.65	0.39	0.37	0.40	0.65	0.35	0.65	0.65	0.44	0.65	0.39	0.37	0.40	0.65	0.38	0.65
LinReg	0.65	0.45	0.65	0.39	0.19	0.42	0.65	0.39	0.65	0.65	0.45	0.65	0.39	0.37	0.40	0.65	0.35	0.65	0.65	0.44	0.65	0.39	0.30	0.40	0.65	0.38	0.65
Mult.Perc.	0.01	0.37	0.01	0.12	0.66	0.44	0.09	0.07	0.10	0.01	0.37	0.01	0.12	0.42	0.15	0.09	0.34	0.12	0.01	0.45	0.03	0.12	0.62	0.15	0.09	0.15	0.10
REPTree	0.10	0.45	0.12	0.63	0.26	0.83	0.23	0.36	0.25	0.10	0.45	0.12	0.63	0.39	0.70	0.23	0.05	0.02	0.10	0.39	0.12	0.63	0.26	0.70	0.23	0.08	0.25
M5P	0.30	0.13	0.32	0.65	0.50	0.79	0.47	0.10	0.48	0.30	0.13	0.32	0.65	0.17	0.72	0.47	0.13	0.48	0.30	0.12	0.32	0.65	0.28	0.71	0.47	0.01	0.48
SMOreg	0.06	0.25	0.08	0.15	0.70	0.10	0.04	0.37	0.04	0.06	0.25	0.08	0.15	0.65	0.15	0.04	0.20	0.04	0.06	0.30	0.08	0.15	0.64	0.16	0.04	0.20	0.04
ZeroR	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00
J48	37.21	27.91	58.14	53.33	60.00	53.33	34.48	27.59	37.93	37.21	27.91	58.14	53.33	51.11	60.00	34.48	41.38	48.28	37.21	25.58	58.14	53.33	28.89	60.00	34.48	34.48	41.38
NaiveBayes	25.58	30.23	20.93	22.22	28.89	22.22	17.24	37.93	24.14	25.58	30.23	20.93	22.22	17.78	17.78	17.24	31.03	20.69	25.58	30.23	20.93	22.22	20.00	17.78	17.24	24.14	13.79
Logistic	27.91	23.26	30.23	51.11	31.11	40.00	27.59	37.93	31.03	27.91	23.26	30.23	51.11	33.33	46.67	27.59	31.03	24.14	27.91	27.91	30.23	51.11	33.33	46.67	27.59	24.14	24.14
SMO	39.53	27.91	27.91	40.00	28.89	42.22	27.59	41.38	17.24	39.53	27.91	27.91	40.00	26.67	44.44	27.59	17.24	3.45	39.53	23.26	27.91	40.00	17.78	44.44	27.59	13.79	10.34
ZeroR	23.26	23.26	23.26	22.22	22.22	22.22	34.48	34.48	34.48	23.26	23.26	23.26	22.22	22.22	22.22	34.48	34.48	34.48	23.26	23.26	23.26	22.22	22.22	22.22	34.48	34.48	34.48
J48	65.12	41.86	65.12	71.11	57.78	60.00	55.17	37.93	55.17	65.12	41.86	65.12	71.11	57.78	71.11	55.17	44.83	55.17	65.12	46.51	65.12	71.11	44.44	71.11	55.17	34.48	55.17
NaiveBayes	32.56	37.21	32.56	33.33	40.00	40.00	58.62	41.38	58.62	32.56	37.21	32.56	33.33	44.44	33.33	58.62	41.38	58.62	32.56	37.21	32.56	33.33	46.67	33.33	58.62	37.93	58.62
Logistic	34.88	51.16	34.88	66.67	46.67	60.00	62.07	48.28	62.07	34.88	51.16	34.88	66.67	60.00	66.67	62.07	41.38	62.07	34.88	53.49	34.88	66.67	57.78	66.67	62.07	41.38	62.07
SMO	39.53	46.51	39.53	53.33	60.00	42.22	55.17	55.17	55.17	39.53	46.51	39.53	53.33	46.67	53.33	55.17	55.17	55.17	39.53	44.19	39.53	53.33	53.33	53.33	55.17	41.38	55.17

Table 3: Full prediction accuracies

	Single									With CPU									All								
	CPU			GPU			FPGA			CPU			GPU			FPGA			CPU			GPU			FPGA		
Algorithm	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**
ZeroR	0.48	0.46	0.48	0.36	0.46	0.37	0.65	0.41	0.65	0.48	0.46	0.48	0.36	0.42	0.36	0.65	0.41	0.65	0.48	0.47	0.48	0.36	0.44	0.35	0.65	0.41	0.65
LinReg	0.48	0.46	0.48	0.36	0.32	0.37	0.65	0.41	0.65	0.48	0.46	0.48	0.36	0.42	0.36	0.65	0.41	0.65	0.48	0.47	0.48	0.36	0.23	0.35	0.65	0.41	0.65
Mult.Perc.	0.01	0.13	0.02	0.63	0.63	0.56	0.07	0.16	0.16	0.01	0.13	0.02	0.63	0.46	0.67	0.07	0.39	0.13	0.01	0.14	0.02	0.63	0.50	0.61	0.07	0.48	0.06
REPTree	0.08	0.10	0.08	0.74	0.27	0.70	0.19	0.18	0.21	0.08	0.10	0.08	0.74	0.00	0.77	0.19	0.07	0.13	0.08	0.47	0.08	0.74	0.21	0.74	0.19	0.19	0.21
M5P	0.01	0.04	0.01	0.67	0.05	0.67	0.43	0.16	0.44	0.01	0.04	0.01	0.67	0.02	0.64	0.43	0.16	0.44	0.01	0.04	0.01	0.67	0.11	0.67	0.43	0.16	0.44
SMOreg	0.04	0.07	0.05	0.00	0.53	0.00	0.07	0.12	0.07	0.04	0.07	0.05	0.00	0.62	0.01	0.07	0.04	0.06	0.04	0.12	0.05	0.00	0.63	0.01	0.07	0.08	0.06
ZeroR	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00
J48	32.56	18.60	44.19	48.89	35.56	46.67	68.97	20.69	68.97	32.56	18.60	44.19	48.89	22.22	48.89	68.97	17.24	68.97	32.56	34.88	37.21	48.89	48.89	48.89	68.97	48.28	68.97
NaiveBayes	18.60	20.93	13.95	26.67	31.11	22.22	34.48	24.14	34.48	18.60	20.93	13.95	26.67	20.00	33.33	34.48	17.24	34.48	18.60	13.95	6.98	26.67	28.89	33.33	34.48	27.59	34.48
Logistic	41.86	25.58	44.19	33.33	37.78	40.00	41.38	27.59	41.38	41.86	25.58	44.19	33.33	33.33	35.56	41.38	24.14	41.38	41.86	25.58	25.58	33.33	31.11	35.56	41.38	27.59	41.38
SMO	32.56	20.93	20.93	33.33	28.89	40.00	34.48	20.69	34.48	32.56	20.93	20.93	33.33	20.00	42.22	34.48	17.24	34.48	32.56	23.26	20.93	33.33	28.89	42.22	34.48	10.34	34.48
ZeroR	30.23	23.26	30.23	22.22	22.22	22.22	34.48	34.48	34.48	30.23	23.26	30.23	22.22	22.22	22.22	34.48	34.48	34.48	30.23	23.26	30.23	22.22	22.22	22.22	34.48	34.48	34.48
J48	55.81	32.56	55.81	53.33	62.22	53.33	55.17	48.28	55.17	55.81	32.56	55.81	53.33	46.67	48.89	55.17	41.38	55.17	55.81	32.56	55.81	53.33	35.56	48.89	55.17	37.93	55.17
NaiveBayes	37.21	39.53	37.21	44.44	44.44	57.78	34.48	31.03	34.48	37.21	39.53	37.21	44.44	40.00	48.89	34.48	17.24	34.48	37.21	39.53	37.21	44.44	42.22	48.89	34.48	17.24	34.48
Logistic	65.12	37.21	65.12	48.89	55.56	53.33	68.97	37.93	68.97	65.12	37.21	65.12	48.89	57.78	53.33	68.97	27.59	68.97	65.12	37.21	65.12	48.89	53.33	53.33	68.97	34.48	68.97
SMO	53.49	46.51	53.49	48.89	55.56	51.11	44.83	41.38	44.83	53.49	46.51	53.49	48.89	40.00	55.56	44.83	17.24	44.83	53.49	46.51	53.49	48.89	44.44	55.56	44.83	27.59	44.83

Table 4: Kernel prediction accuracies

Benchmark	Analysability	Modifiability	Modularity	Reusability	Testability	Maintainability
mri-q	−0.388	−0.405	−0.448	−0.432	−0.360	−0.407
spmv	−0.667	−0.685	−0.653	−0.676	−0.658	−0.668
stencil	−0.225	−0.237	−0.428	−0.325	−0.199	−0.283
atax	−0.338	−0.354	−0.472	−0.406	−0.298	−0.375
bicg	−0.342	−0.358	−0.471	−0.412	−0.308	−0.379
conv2d	−0.332	−0.346	−0.469	−0.405	−0.296	−0.370
doitgen	−0.372	−0.388	−0.582	−0.476	−0.324	−0.429
gemm	−0.269	−0.283	−0.435	−0.352	−0.237	−0.315
gemver	−0.325	−0.343	−0.494	−0.417	−0.294	−0.375
gesummv	−0.290	−0.304	−0.384	−0.343	−0.262	−0.317
jacobi2d	−0.420	−0.433	−0.560	−0.491	−0.373	−0.456
mvt	−0.339	−0.353	−0.444	−0.396	−0.304	−0.368
bfs	−0.352	−0.367	−0.497	−0.431	−0.319	−0.393
hotspot	−0.226	−0.235	−0.371	−0.308	−0.167	−0.261
lavaMD	−0.271	−0.276	−0.352	−0.315	−0.244	−0.292
nn	−0.429	−0.434	−0.560	−0.485	−0.364	−0.456

Table 5: Maintainability changes at the system level

Benchmark	Analysability	Modifiability	Modularity	Reusability	Testability	Maintainability
mri-q	−0.234	−0.240	−0.395	−0.321	−0.225	−0.282
spmv	0.139	0.135	−0.308	−0.069	0.188	0.019
stencil	0.145	0.144	−0.617	−0.205	0.220	−0.059
atax	−0.109	−0.136	−0.435	−0.283	−0.087	−0.208
bicg	−0.200	−0.222	−0.449	−0.329	−0.162	−0.272
conv2d	−0.065	−0.075	−0.431	−0.228	−0.002	−0.161
doitgen	0.147	0.131	−0.653	−0.228	0.226	−0.072
gemm	0.120	0.110	−0.391	−0.123	0.175	−0.019
gemver	−0.161	−0.187	−0.708	−0.429	−0.119	−0.319
gesummv	−0.041	−0.055	−0.341	−0.199	−0.033	−0.131
jacobi2d	−0.035	−0.057	−0.691	−0.347	0.019	−0.220
mvt	−0.148	−0.174	−0.443	−0.302	−0.115	−0.236
bfs	0.067	0.063	−0.408	−0.150	0.095	−0.064
hotspot	−0.035	−0.041	−0.774	−0.390	0.043	−0.235
lavaMD	−0.158	−0.165	−0.434	−0.303	−0.150	−0.239
nn	0.015	0.017	0.007	0.006	0.013	0.012

Table 6: Maintainability changes at the kernel level

Summary

Our contributions can be grouped into two major thesis points.

The main results of the first thesis point are the empirical studies mentioned in Section I, which support the intuitive expectations about the relations between well-known source code patterns and software maintainability with objective, definite data. To our knowledge, these findings are among the first that were performed on subject systems of such volume, size, and variance, while also avoiding all subjective factors like developer surveys, time tracking, and interviews.

The main results of the second thesis point are (a) the empirical proof that static source code metrics are suitable for improvement estimation, and (b) a universal process for creating qualitative and quantitative hardware platform prediction models, as described in Section II. A key difference between our approach and other available solutions is that, once they are built, our models operate based on static information alone. Also, their accuracy depends primarily on the number of training benchmarks. These properties make the methodology easy to enhance, and its output models easy to apply.

Table 7 summarizes the main publications and how they relate to our thesis points.

№	[10]	[4]	[2]	[3]	[5]	[6]
I.	◆	◆	◆	◆		
II.					◆	◆

Table 7: Thesis contributions and supporting publications

Acknowledgements

I would like to thank my supervisor, Dr. Rudolf Ferenc for his guidance and insight. He showed me multiple times during our shared research that a bad result or a failed experiment is not the end of a project, only a point where we should change our strategy and carry on. I would also like to thank Dr. Péter Hegedűs for "showing me the ropes", and Dr. István Siket for his ideas about the application of empirical distribution functions and for just always being available when I needed assistance with anything. My sincere thanks to Dr. Tibor Gyimóthy, the head of the Software Engineering Department, for providing me with many interesting research opportunities. Special thanks go to Gergely Ladányi for his invaluable advice and help regarding quality models and data analysis, to Róbert Sipka and Péter Molnár for their tireless work on the dynamic measurements, and to David Curley for his grammatical and stylistic comments. Also, many thanks to my other co-authors, namely Dr. Ákos Kiss, and Gábor Gyimesi, for their contributions.

Dénes Bán, 2017

References

[1] Tibor Bakota, Péter Hegedűs, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. A Probabilistic Software Quality Model. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, ICSM 2011, pages 368–377, Williamsburg, VA, USA, 2011. IEEE Computer Society.

[2] Dénes Bán. The connection of antipatterns and maintainability in firefox. In *10th Jubilee Conference of PhD Students in Computer Science (CSCS 2016), Szeged, Hungary, June 27 – 29*, 2016.

[3] Dénes Bán. The connection of antipatterns and maintainability in firefox. Accepted for publication in the 2016 Special Issue of Acta Cybernetica (extended version of [2]). 20 pages.

[4] Dénes Bán and Rudolf Ferenc. Recognizing antipatterns and analyzing their effects on software maintainability. In *14th International Conference on Computational Science and Its Applications (ICCSA 2014), Guimarães, Portugal, June 30 – July 3*, pages 337–352. Springer International Publishing, 2014.

[5] Dénes Bán, Rudolf Ferenc, István Siket, and Ákos Kiss. Prediction models for performance, power, and energy efficiency of software executed on heterogeneous hardware. In *13th IEEE International Symposium on Parallel and Distributed Processing with Applications (IEEE ISPA-15), Helsinki, Finland, August 20 – 22*, volume 3, pages 178–183, 2015.

[6] Dénes Bán, Rudolf Ferenc, István Siket, Ákos Kiss, and Tibor Gyimóthy. Prediction models for performance, power, and energy efficiency of software executed on heterogeneous hardware. Submitted to the Journal of Supercomputing (extended version of [5]). 24 pages.

[7] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[8] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, Aug 1994.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co, 1995.

[10] Péter Hegedűs, Dénes Bán, Rudolf Ferenc, and Tibor Gyimóthy. Myth or reality? analyzing the effect of design patterns on software maintainability. In *Advanced Software Engineering & Its Applications (ASEA 2012), Jeju Island, Korea, November 28 – December 2*, pages 138–145. Springer Berlin Heidelberg, 2012.

[11] Ákos Kiss, Péter Molnár, and Róbert Sipka. Rmeasure performance and energy monitoring library. `https://github.com/sed-szeged/RMeasure`, 2017.

[12] Tim Menzies, Bora Caglayan, Zhimin He, Ekrem Kocaguneli, Joe Krall, Fayola Peters, and Burak Turhan. The promise repository of empirical software engineering data, June 2012.

[13] Rudolf Ferenc et al. *REPARA deliverable D7.4: Maintainability models of heterogeneous programming models*. 2015.