

A Quantitative Study of the Application of Design Patterns in Java

Michael Hahsler

Department of Information Business

Vienna University of Economics and BA, Augasse 2-6, A-1090 Vienna, Austria

hahsler@ai.wu-wien.ac.at

Abstract – Using design patterns is a widely accepted method to improve software development. There are many benefits of the application of patterns claimed in the literature. The most cited claim is that design patterns can provide a common design vocabulary and therefore improve greatly communication between software designers. Most of the claims are supported by experiences reports of practitioners, but there is a lack of quantitative research concerning the actual application of design patterns and about the realization of the claimed benefits. In this paper we analyze the development process of over 1000 open source software projects using version control information. We explore this information to gain an insight into the differences of software development with and without design patterns. By analyzing these differences we provide evidence that design patterns are used for communication and that there is a significant difference between developers who use design patterns and who do not.

1 Introduction

The need for reliable software has made software engineering an important industry in the last decades. The steady progress recently produced an enormous number of different approaches, concepts and techniques: the object oriented paradigm, agile software development, the open source movement, component based systems, frameworks and software patterns, just to name a few. All these approaches claim to be superior, more effective or more appropriate in some area to their predecessors. However, to prove that these claims indeed hold and generate benefits in a real-world setting is often very hard due to missing empirical data and a lack of control over the environmental conditions of the setting. Therefore, often controlled experiments are used (see [PrUn98] for a controlled experiment on the application of design pattern). Controlled experiments enable the researcher to control and document the environmental conditions and therefore results can be reproduced. Also the controlled conditions can be changed to analyze causal relationships. However, there are some disadvantages of controlled experiments which are related to the question if the environmental set-up for the experiment allows for generalization of the results to real-world situations. To solve this problem, controlled experiments should be supplemented by field research, which analyses data of real projects collected in a real-world situation in a non-intrusive way (see [CoLa98]). In this paper we will present a quantitative analysis for the application of design patterns using data from over 1000 real-world software development projects.

Design patterns describe non-obvious solutions in a standard written form for recurring software design problems in a certain context. Since the introduction of the first software pattern catalog containing 23 design patterns in Gamma et al. [GHJV95], design patterns were rapidly accepted by the software engineering community. The number of publications about

design patterns have soared, and even several conference series on the topic were initiated. In the US the conference series has the name *Pattern Languages of Programs (PLoP)* and in other parts of the world conferences series like EuroPloP, KoalaPloP, ChiliPloP were started. These conferences as well as most publications focus on the development of new and improved design patterns, but the research on the actual application of design patterns by software developers and the assessment of the benefits of the adoption of patterns is still underdeveloped.

In this paper we use historic data of the development process of software projects to gain an insight into the application of design patterns. The paper is organized as follows: We start with a short review of related literature to identify claims made about the benefits of design patterns. In section 3 we describe the used research method. In section 4 we present the used data set. In section 5 we analyze the data set and discuss the results. We conclude with our main findings in section 6.

2 Related Literature

In their initial book about design patterns, Gamma et al. [GHJV95, pp.352] state that they expect design patterns will provide:

1. a common vocabulary,
2. a documentation and learning aid,
3. an adjunct to existing methods,
4. a target for refactoring.

Now, since design patterns are a widely accepted technique, these expectations need to be checked against reality. There were some early publications by practitioners that describe experiences with design patterns in an industrial setting as well as for training (see [Helm95],[BCCD96],[GoRi96]). But these publications contain only experience reports and do not use empirical data to support their claims.

In the joint paper “Industrial Experience with Design Patterns” [BCCD96] authored together by Kent Beck (First Class Software), James O. Coplien (AT&T), Ron Crocker (Motorola Inc.), Lutz Dominick and Frances Paulitsch (Siemens AG), Gerard Meszaros (Bell Northern Research) and John Vlissides (IBM Research) these 7 experts describe the efforts and experiences they and their companies had with design patterns. The paper contains a table of the most important observations ordered by the number of experts who mentioned them. This can be interpreted as the results of interviewing experts. The top 3 observations mentioned by all experts are (taken from [BCCD96]):

1. Patterns are a good communication medium.
2. Patterns are extracted from working designs.
3. Patterns capture design essentials.

The first observation is the most prominent benefit of design patterns. In [GHJV95] two of the expected benefits are that design patterns provide “a common design vocabulary” and “a documentation and learning aid” which also focus on the communication process. The other two observations focus on the idea that design patterns describe best practices for important aspects of software design.

In [PrUn98] two controlled experiments using design patterns for maintenance exercises are presented. For one experiment students were used to compare the speed and correctness of maintenance work with and without design patterns used for the documentation of the original program. The result of this experiment was that using patterns in the documentation increases either the speed or decreases the number of errors for the maintenance task and thus seems to

improve communication between the original developer and the maintainer via the documentation. In the second experiment the research question was if the use of patterns is beneficial compared to simpler design solutions. For this experiment professional software engineers were asked to extend programs that use patterns and programs that use simpler design but provide the same functionality. The results were not very clear and the authors concluded that design patterns are useful since they provide more flexible design, but are no cure-all since they can lead to more complicated solutions.

3 Research Method

In this paper we analyze the development process by using publicly available version control data for open source software projects available via the Source Forge Web site (<http://www.sourceforge.net>). This approach is inexpensive and non-intrusive (see [CoVo98][ABGM99]) and was already successfully used to analyze the development of the Apache Web Server project [MoFH00] and of the GNOME project [KoSe02], both large scale open source projects.

Source Forge currently hosts over 50000 open source projects and has over 500000 registered users (November 2002). It provides the projects with a version control facility as well as a presentation platform and communication channels for developers and users. For Source Forge each developer has a unique pseudonym, the user name, which he uses throughout all projects he participates in. Each project has a home page with general information about the project like the project name, a short description of the project, the developers in charge of the project (administrators), the development status of the project (alpha, beta, production...), the intended audience (developers, end users,...), the programming languages used, and more general information.

For this paper we analyze projects which use the object-oriented programming language Java and employ the version control tool Concurrent Versions Control (CVS) [Foge99]. This tool enables parallel development by several developers, comments for modifications of the code, control of releases, reversing modifications, generate history logs for the projects and for each individual file, and much more. A project is defined as a collection of individual files which are stored together with version control information in a CVS repository. New files can be added to the project and existing files can be modified by the developers of the project. The modification of files using the version control system involves the following steps:

1. Obtain the latest version (or some other version) of the files from the repository (*'check-out'* the files).
2. Change the files locally.
3. Update the repository with the modifications (execute a *'check-in'* for the files).

During the check-in (often called modification request) the developer is encouraged to add a short log message which explains the purpose of the modifications and therefore makes it easier to understand the changes in the code later on. CVS records the modifications in each file in lines of code (LOCs) added and LOCs deleted by the developer. The definition of LOCs used by CVS is the number of physical lines. There is no distinction between program statements, comments or other arbitrary text. We adopt this definition for our research, but we have to keep this limitation in mind for interpreting the results. Furthermore, CVS does not explicitly record changes in a line, instead it records a changed line as a line deleted and a new line added. Therefore, the growth of LOCs for a check-in (the delta) is the difference between the LOCs added and the LOCs deleted. A fragment of a CVS log is shown in Table 1 where the information given for each check-in (the date, the developer called in CVS author and the LOCs added and deleted) is printed in bold font. In the log message you can see the comments the developer provided with the check-in.

Table 1: Excerpt of a CVS log for a file in a project

```
RCS file:
/cvsroot/jboss/ejboss/src/java/org/ejboss/ejb/EnterpriseBeanWrapper.java,v
Working file: ejboss/src/java/org/ejboss/ejb/EnterpriseBeanWrapper.java
head: 1.23
branch:
locks: strict
access list:
symbolic names:
    EJBoss-1-0-PR2-A: 1.21
    EJBoss-1-0-PR1-A: 1.20
    EJBoss-1-0-DR2-A: 1.18
    AUTOCONF: 1.8.0.2
keyword substitution: kv
total revisions: 23;    selected revisions: 23
description:
-----
revision 1.23
date: 2000/06/01 00:51:24; author: sylvain; state: Exp; lines: +16 -14
Bug Fix : if a transaction-scoped method of a bean creates a new entity bean,
we mustn't terminate the transaction after ejbPostCreate. We have to wait
until the end of the encapsulating transaction..
-----
.
. (lines omitted)
.
-----
revision 1.2
date: 1999/10/28 23:35:21; author: fleury; state: Exp; lines: +91 -34
Bulk of the work, pass the invoke on to the right wrapper.
Use the wrapper manager for that. Also we now have
an empty constructor as we use the "Factory" pattern in
the pool managers.
-----
```

To analyze the application of design patterns we first have to identify the patterns in the projects. Design patterns are design artifacts that result in special constructions in the final code, e.g. several objects that interact in a certain way. It is very difficult to infer the application of design patterns automatically directly from code (see e.g. [AnFC98] or [PKG00] for automatic approaches). We use the log messages to identify the application of design patterns by looking for their names and descriptions. Although design patterns can be applied without mentioning them in the log message or a design pattern can be referred to by a different name, this seems to be a reasonable approach due of the following fact: One of the major contributions of patterns stressed throughout the literature is, that the names of design patterns become part of a common design language which developers use to communicate more efficiently [GHJV95, p.352], [BMRS96, p.6], [Vlis98, p.6]. Therefore, the usage of patterns is beneficiary for other developers and for documentation purpose if the pattern name is given in the description, in our case the log message. By analyzing the log messages we can find the cases when patterns are used for this reason.

Although many design patterns were introduced in the literature (e.g. in [CoSc95], [BMRS96], [VICK96] and [MaRB98]), we only use the original set of design patterns introduced by Gamma et al. in [GHJV95]. These patterns are certainly the best known design patterns in the software engineering community. The list of used patterns plus their intends (a short description of the purpose of the pattern) can be found in the appendix of this paper. To identify the application of a pattern we search the log messages for the pattern name co-occurring with keywords taken from the pattern's intend (italic words in the appendix) or the word 'pattern.' With this approach we reduce the identification problem for patterns with names that are very common words for software design besides the design pattern itself (e.g. the word 'prototype' is the name of a design pattern but is also used frequently in software engineering for a first version of a program).

We extracted the CVS log messages for all projects using Java, parsed them using regular expressions and stored the information in a relational database. The structure of the database is depicted by the entity-relationship diagram in Figure 1. Each project consists of several files. Each check-in is a relationship between a files, one developer and can contain none, one or several design patterns. Finally, there is a relationship between project and developer which means that one or more of the developers administrate a project.

All analyses in the subsequent sections are performed using standard SQL select statements on the data base and a standard statistical package.

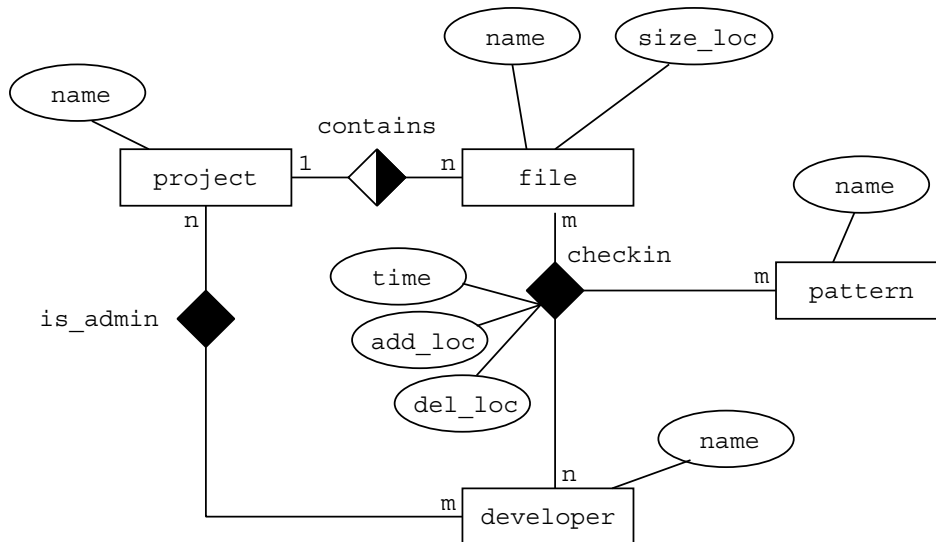


Figure 1: Entity-relationship diagram of the used database

4 The Data Set

The used data set includes 1319 open source projects from Source Forge using Java as the main programming language. Java was chosen since it is a widely used object oriented programming language for which the application of design patterns is recently becoming widely documented (e.g. see [AICM01]). In fact in the libraries shipped with Java (API of the Java 2 platform) several design patterns are already used (e.g. the patterns Observer and Abstract Factory in Java's Abstract Window Toolkit).

The projects were downloaded between August and September 2001 and were selected by the following criteria: only projects that enabled CVS and that are non-empty, i.e. have files and check-ins in the CVS repository. In total the 1319 selected projects contained 57771583 LOCs and 2164 different developers worked on them. For the analysis in this paper, we excluded the initial check-in of new files. The reason for this measure is that new files added to the Source Forge CVS repository often have already a considerable size and it is impossible to say how many developers (and who) worked on this file already since we only know the developer who checked-in the file. This would distort the results in an unpredictable way.

Figure 2 and Figure 3 depict the distribution of the size of the projects by LOCs and by the number of files. Most of the projects are rather small with a mean of 43800 LOCs or 301 files but there is a significant amount of much bigger projects (the biggest project has over 4 million LOCs and over 40000 files).

The team sizes of the projects show a similar distribution with many projects with only a single developer (see Figure 4). The mean of the team size is 1.87 and the biggest team is 88 developers.

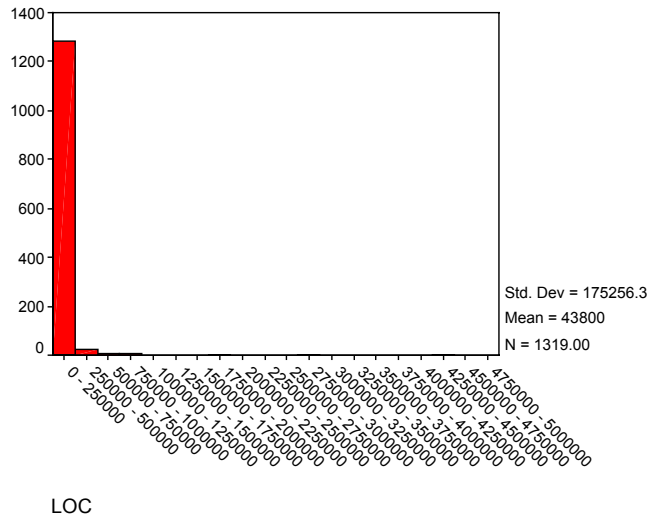


Figure 2: Number of projects by size in LOC

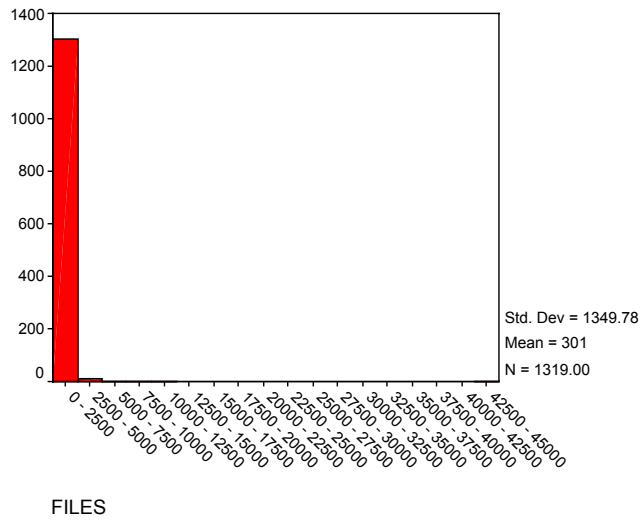


Figure 3: Number of projects by size in number of files

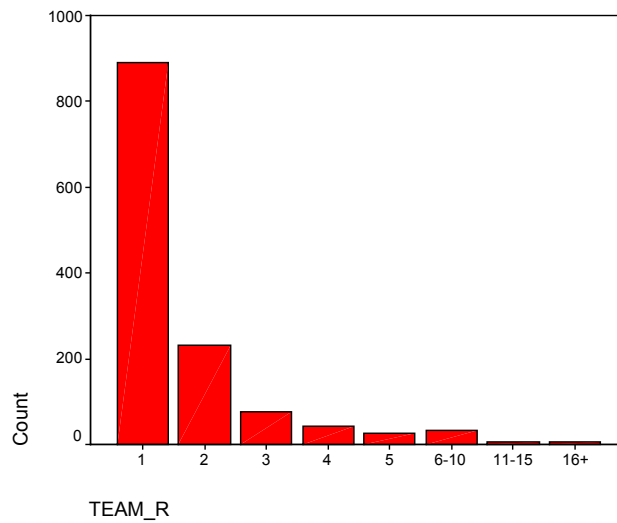


Figure 4: Number of projects by team size

Figure 5 shows the number of projects by the development status given by Source Forge. The status ranges from 1 to 6 (1-planning, 2-pre-alpha, 3-alpha, 4-beta, 5-production/stable and 6-mature) and gives an idea about the project in its development live-cycle. The number of projects for the status 1 through 4 are similar with around 280 projects per status. For the status 5 and 6 there are fewer projects indicating that most of the analyzed projects can be considered still in the design and implementation phases of the software life-cycle. This is a major difference to other papers which mainly focus on analyzing maintenance tasks (e.g. see [PrUn98]). In Table 2 we summarize the statistics of the used variables.

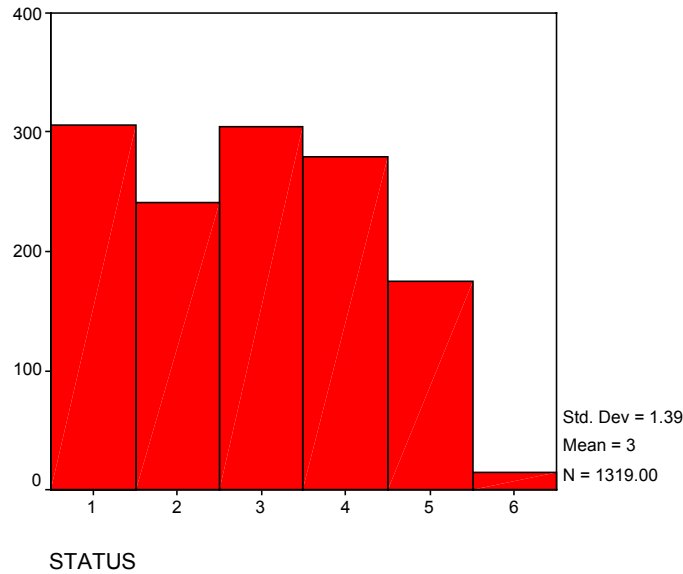


Figure 5: Number of projects by status

Table 2: Descriptive statistics of the main project variables

Case Summaries

	STATUS	LOC	FILES	TEAM
N	1319	1319	1319	1319
Median	3.00	9926.00	90.00	1.00
Minimum	1	1	1	1
Maximum	6	4283433	42674	88
Mean	2.86	43799.53	301.19	1.87
Variance	1.931	3.1E+10	1821912	11.142

5 Results

In this section the results of our explorative analysis are presented. The section is divided into four parts. First the identified patterns are presented. Then we try to find out which characteristics of the project (e.g. size) make the usage of design patterns more likely. In the second part we explore if there exist differences between files (e.g. size, number of modifications) that influence the usage of patterns. And finally we analyze if there are specific characteristics of developers that use design patterns (e.g. observed experience).

5.1 The Identified Design Patterns

We applied the approach described above to identify individual patterns in the data set. In Figure 6 we show the number of projects in which we found each individual pattern. The pattern names *Command* and *State* appear very often in the data. Since both words are very common in programming and design, this could indicate a problem in our way to identify patterns by their name and some keywords from the intents. We have to keep this observation in mind. In third place appears *Singleton* which is a very unusual word for developers not familiar with design patterns. This leads us to the conclusion, that for Java the application of the pattern Singleton seems to provide important design advantages.

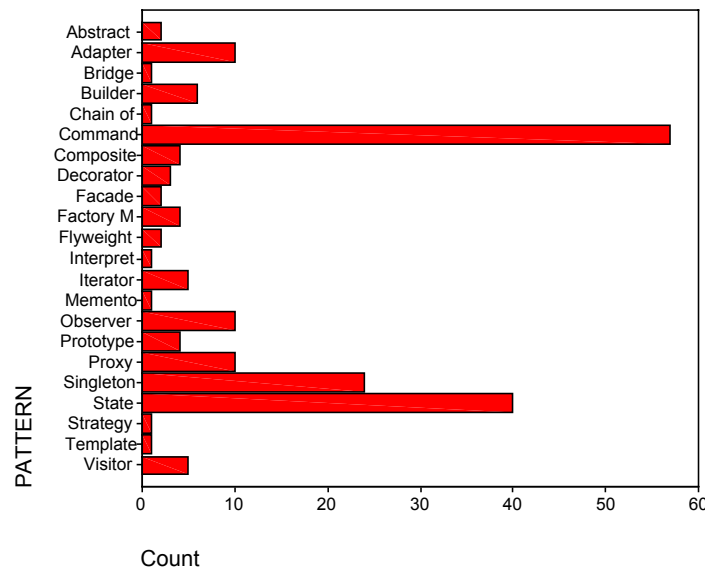


Figure 6: Number of projects using individual patterns

5.2 Analysis of Differences Between Projects

For Source Forge all development efforts are organized in projects as the basic unit of coordination. A project has one or several administrators who coordinate the development of the project and organize the cooperation between the developers. Many projects in the data set do not apply patterns and only for a small fraction 9.7% (128 out of 1319 projects) we found one or more patterns. To exclude very small projects as well as projects which are still in their planning phase and therefore by now do not have enough code which can contain design patterns, we only select the projects with more than 100000 LOCs. This leaves 108 projects for analysis. This general trend of the distributions of the number of files and the team sizes of this sample is similar to the distributions of the whole data set. Only the distribution of the development status changes since many projects in their planning and pre-alpha phase do not have the required LOCs. Figure 7 shows the new distribution and Table 3 contains the descriptive statistics for the selected projects.

In Figure 8 we show the number of projects using 0,1,...5 different design patterns in the project. 28 projects (25.93% of the 108 projects) use at least one design pattern.

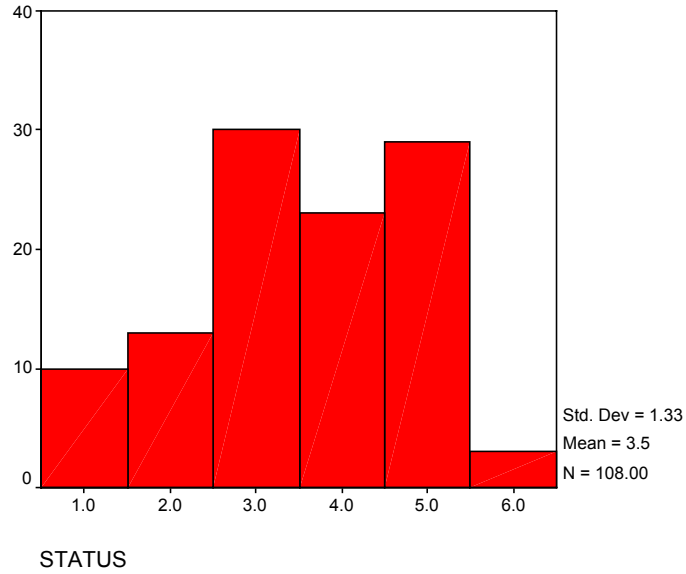


Figure 7: Number of projects by status of the selected projects

Table 3: Descriptive statistics of the main project variables of the selected projects

Case Summaries

	STATUS	LOC	FILES	TEAM
N	108	108	108	108
Minimum	1	101083	41	1
Maximum	6	4283433	42674	88
Median	4.00	206468.50	1036.00	2.00
Mean	3.53	341906.58	1957.98	4.98
Variance	1.766	2.8E+11	1.9E+07	108.430

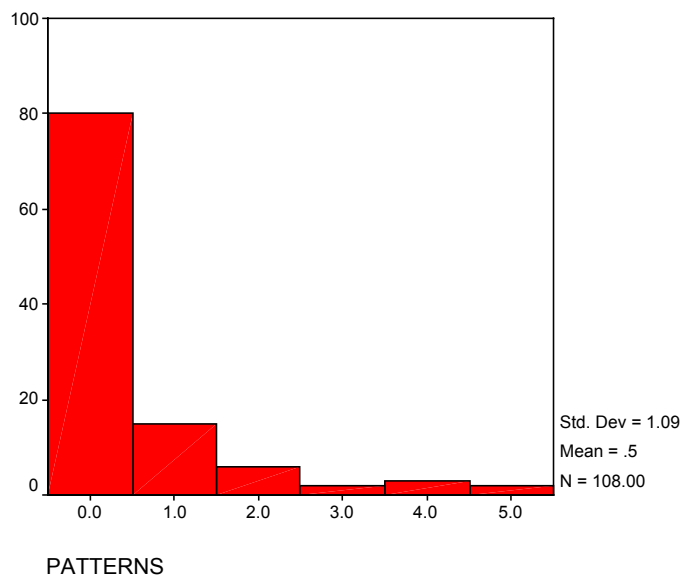


Figure 8: Projects by number of different design patterns used

Table 4: Correlation between main project variables**Correlations**

		STATUS	LOC	FILES	TEAM	PATTERNS
STATUS	Pearson Correlation	1.000	.237*	.194*	.080	.097
	Sig. (2-tailed)	.	.014	.044	.408	.320
	N	108	108	108	108	108
LOC	Pearson Correlation	.237*	1.000	.879**	.257**	.120
	Sig. (2-tailed)	.014	.	.000	.007	.215
	N	108	108	108	108	108
FILES	Pearson Correlation	.194*	.879**	1.000	.193*	.104
	Sig. (2-tailed)	.044	.000	.	.045	.284
	N	108	108	108	108	108
TEAM	Pearson Correlation	.080	.257**	.193*	1.000	.602**
	Sig. (2-tailed)	.408	.007	.045	.	.000
	N	108	108	108	108	108
PATTERNS	Pearson Correlation	.097	.120	.104	.602**	1.000
	Sig. (2-tailed)	.320	.215	.284	.000	.
	N	108	108	108	108	108

*. Correlation is significant at the 0.05 level (2-tailed).

**. Correlation is significant at the 0.01 level (2-tailed).

Next, we analyze the relationship between the main project variables (development status, size of the project in LOCs and number of files, the team size and the number of different patterns used in the project). Table 4 gives the correlation between all variables. A high correlation (0.879) exists between the two measures of the project size, the LOCs and the number of files. This was to be expected. What is more interesting is that the second highest correlation was found between the team size and the number of different patterns used in the project (a significant correlation of 0.602). This correlation is even higher than the more intuitive correlation between the team size and the size of the project with 0.257 for the size in LOCs and 0.193 for the size in number of files. Notable is also the fact that the development status of the project has only very little correlation (<0.25) with the other variables. This indicates, that we have very different projects in terms of size in the data set and that we cannot find evidence that design patterns are used more often in later stages of the life-cycle to refactor code (replace code and design with more flexible design provided by a design pattern) as suggested in [GHJV95, pp.353]. To analyze this in more detail, we split the projects into two groups. The first group contains project that are still in the alpha phase (project status 1-3) and projects that are more mature (project status 4-6). If design patterns are used frequently for refactoring, the more mature projects should contain significantly more design patterns. However, in the data set no significant difference between the two groups of projects could be found (see Table 5). This can result either from the fact that design patterns are not used for refactoring in our data set or that there was only little refactoring done in the analyzed open source projects.

Table 5: Comparison of the usage of patterns for projects in their alpha stages and more mature projects. There usage does not differ significantly (two sample Kolmogorov-Smirnov test)

Test Statistics^a

		PATTERNS
Most Extreme Differences	Absolute	.102
	Positive	.000
	Negative	-.102
Kolmogorov-Smirnov Z		.528
Asymp. Sig. (2-tailed)		.944

a. Grouping Variable: DEVELOP

To explore the relationship between the team size and the usage of patterns we first compare the team size of projects using patterns with projects using no patterns. Figure 9 shows the difference between the team size with two box plots. There is a visible difference between the two distributions. The projects without patterns have a median of only 1.5 developer with 50% of the observations between 1 and 3, while the projects with patterns have a higher median of 6 developers with 50% of the observation between 3 and 15.5 developers. The biggest team not using patterns is 13 developers, many larger teams can only be found in projects using patterns. We used the Kolmogorov-Smirnov test to compare the location of the two distributions. The test confirms that the two distributions have a significantly different location and therefore that there is a relationship in the data between the application of patterns and the team size.

A possible explanation for this finding is that design patterns are, as stressed by several authors, used mainly for efficient communication between developers (see e.g. [GHJV95, pp.352], [Helm95], [KJCD96]). Bigger teams need more coordination and therefore communication between the team members and part of this communication is greatly improved by using patterns and pattern names to describe design decisions and modifications to the code. An alternative explanation could be that in bigger teams there is a higher probability to find a developer who knows about patterns and therefore uses them or introduces other team members to the concept of patterns.

In Figure 10 we further analyze the relationship of the team size with the number of different patterns used per project. The plot shows that for team sizes 5 and smaller the vast majority (almost all projects) do not use patterns. Only 2 out of the 16 projects with a team size of 7 and higher use no patterns. This suggests that starting at a team size of around 7 developers, the usage of several design patterns indeed must offers significant advantages for the open source development process.

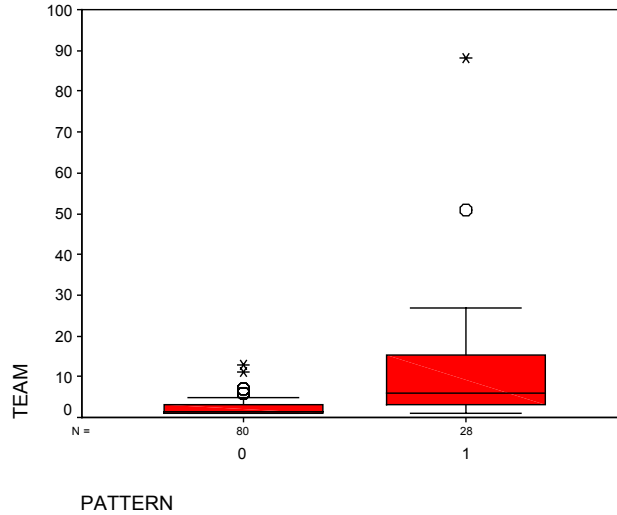


Figure 9: Box plot of the distribution of team sizes for projects using no patterns (left plot) and projects using patterns (right plot). The box plot shows the median (line in the box), the interquartile range which contains 50% of values (box) and cases marked as o and * with more than 1.5 box length from the edge of the box away (denoted by the whiskers).

Table 6: Results of the Kolmogorov-Smirnov test exploring the difference between the team size of projects using patterns and projects using no patterns.

Test Statistics^a

		TEAM
Most Extreme Differences	Absolute	.523
	Positive	.523
	Negative	.000
Kolmogorov-Smirnov Z		2.383
Asymp. Sig. (2-tailed)		.000

a. Grouping Variable: PATTERN

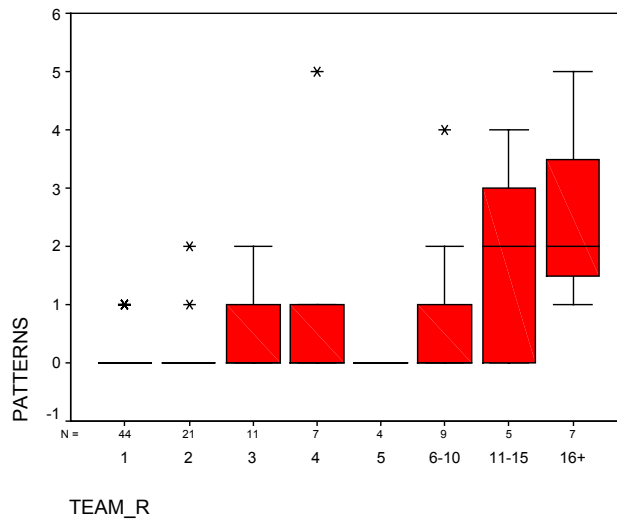


Figure 10: Box plot of the number of patterns used by project depending on the team size

5.3 Analysis of Differences Between Files

In this section we analyze the application of design patterns at the level of individual files to find out if simple characteristics of different files (e.g. size, number of modifications) influence the usage of design or vice versa. The main variables for this analysis are the number of developers working on a file (performing one or more check-ins on the file), the number of check-ins for the file, the number of different patterns used in the file, the average number of LOCs the file size increases per check-in and the average number of LOCs changed by check-in.

The most important information for this analysis in Table 7 is that most files are only worked on by a single developer (with a mean of 1.18) and that only a very small portion of files contain a design pattern at all (0,9%, 1318 of the analyzed 148149 files). In Table 8 the correlation between the variables are given. The relationship between the increase in LOC per check-in and the change of LOC per check-in with a correlation of 0.545 is trivial. The next strongest relationship exists between the number of developers working on a file and the number of check-ins for the file with a correlation of 0.441. This is also no surprise. All other correlation are rather small. There is some relationship between the number of different patterns used in the file and the number of developers as well as the number of check-ins but both are rather weak with a correlation lower than 0.2. Therefore, in the used data no clear sign was found, that simple characteristics of a single file used here affect the usage of design patterns.

To improve this analysis object-oriented software metrics like the Chidamber & Kemerer's Metrics Suite [ChKe94] can be used. These metrics can reflect differences in the complexity of classes better than simple LOCs. However, it was shown in [MaSU99] and in [Reis01] that the introduction of design patterns can increase the complexity measured by object-oriented metrics by using additional classes, more inheritance, and an increased coupling between the classes the pattern is composed of. Therefore a more sophisticated approach is necessary for this analysis. In this paper we restrict our analysis to the simple LOCs-oriented analysis made possible by the output of the CVS-tool.

Table 7: Descriptive statistics of the main variables

Case Summaries

	NUM_PROG	CHECKINS	DIST_PAT	INC_LOGC	ADD_LOGC
N	148149	148149	148149	148149	148149
Minimum	1	1	0	-1167.50	.00
Maximum	20	254	3	1759.00	3817.09
Median	1.00	2.00	.00	.0000	.0000
Mean	1.18	3.22	9.18E-03	2.1077	6.5416
Variance	.353	17.740	9.717E-03	251.190	1154.773

Table 8: Correlation between main variables

Correlations

		NUM_PROG	CHECKINS	DIS_PATT	I_LOC_PC	A_LOC_PC
NUM_PROG	Pearson Correlation	1.000	.441**	.099**	.008**	.031**
	Sig. (2-tailed)	.	.000	.000	.000	.000
	N	393350	393350	393350	393350	393350
CHECKINS	Pearson Correlation	.441**	1.000	.158**	.012**	.043**
	Sig. (2-tailed)	.000	.	.000	.000	.000
	N	393350	393350	393350	393350	393350
DIS_PATT	Pearson Correlation	.099**	.158**	1.000	.004*	.012**
	Sig. (2-tailed)	.000	.000	.	.010	.000
	N	393350	393350	393350	393350	393350
I_LOC_PC	Pearson Correlation	.008**	.012**	.004*	1.000	.545**
	Sig. (2-tailed)	.000	.000	.010	.	.000
	N	393350	393350	393350	393350	393350
A_LOC_PC	Pearson Correlation	.031**	.043**	.012**	.545**	1.000
	Sig. (2-tailed)	.000	.000	.000	.000	.
	N	393350	393350	393350	393350	393350

** . Correlation is significant at the 0.01 level (2-tailed).

* . Correlation is significant at the 0.05 level (2-tailed).

5.4 Analysis of Differences Between Developers

In this section we use the developer as the main unit of analysis. We analyze the usage of patterns for each developer and compare it with observed characteristics of the developer to investigate if there exists a relationship. As the characteristics we use the number of projects a developer participates in, the LOCs he modified in the analyzed period, and the increase of LOCs the developer produced. All three observed values have a relationship with the developers experience and are used here as surrogates for the developers experience. The number of projects a developer works on is related to his experience to work in a team. The more projects he participates in, the more team experience he will obtain. The number of LOCs modified and added relate to his programming experience. And finally, a big quantity of new code produced means the addition of new functionality, which involves also adding new design and therefore is a hint of more design experience. However, since we can not observe how much experience the developer acquired outside the analyzed projects, all three variables are only lower bounds for the real experience.

Only 315 developers added (including changes) more than 10000 LOCs. We restrict our analysis to these developers to exclude developers who did not apply patterns only because of their small participation in the analyzed projects. Figure 11 shows the distribution of the number of developers by LOCs added.

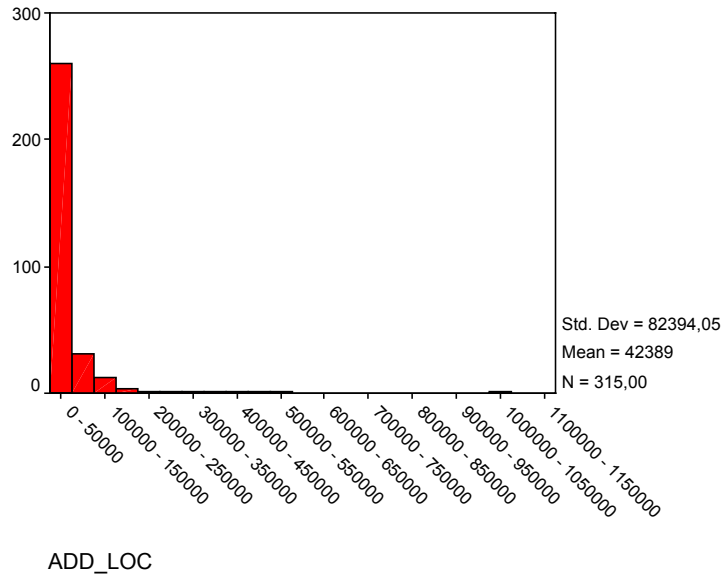


Figure 11: Histogram of the total LOCs changed and added by developer

Figure 12 and Figure 13 show the distribution of the number of projects a developer participates in and the histogram of the LOCs increase produced by the developers. Most developers only work on one project and only very few developers increased the size of the projects measured in LOCs (add new functionality). Only 76 developers (of the analyzed 315 developers and therefore also only 76 out of the total 2164 developers) increased the code by more than 10000 LOCs. That only a few developers produce most of the new code is a common observation for open source projects (see [MoFH00] and [KoSc02]). The other developers seem to be occupied with minor improvements, documentation, testing and fixing small problems in the code.

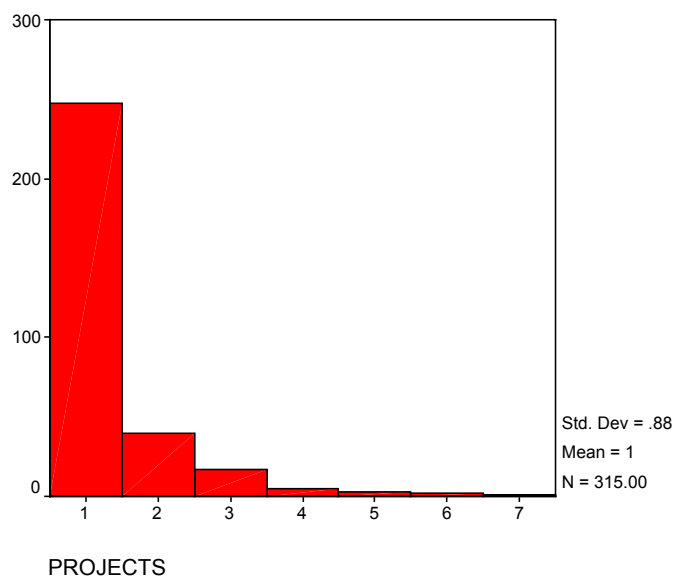


Figure 12: Number of projects a developer contributes to

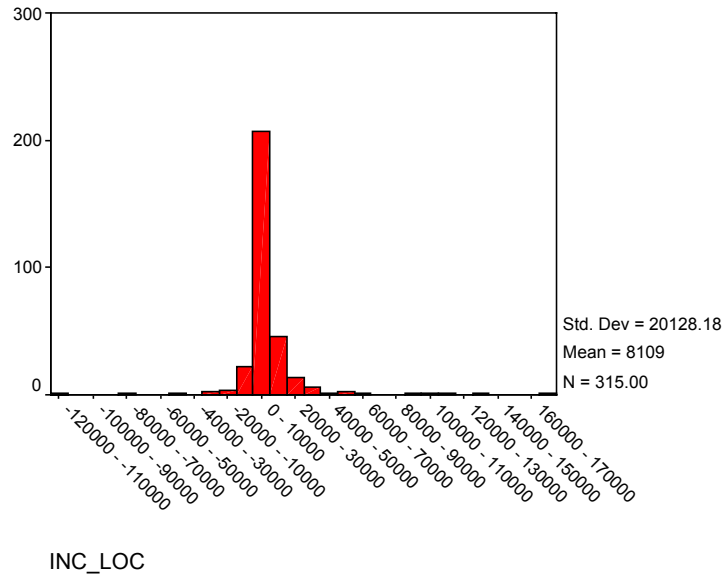


Figure 13: Histogram of the increase of LOCs by developer.

We analyzed the relationship of the variables with simple correlation in Table 9 to check if the usage of design patterns is related to our surrogates for experience. The correlation between the variables were rather small (probably in part due to the fact that all used variables are lower bounds of the actual experience of the developer). The highest correlation was found between the surrogate for design experience (increase in LOCs) and the number of different used patterns (0.173). This relationship is also depicted as a box-plot in Figure 14. The found correlation could mean that developers using design patterns have more design experience or more experienced designers like to document their changes using design patterns. But this is a very tentative interpretation given the weak correlation found.

Table 9: Correlation between the main variables

Correlations

		PROJECTS	ADD_LOC	INC_LOC	DIS_PATT
PROJECTS	Pearson Correlation	1.000	-.012	.026	.130*
	Sig. (2-tailed)	.	.839	.643	.021
	N	315	315	315	315
ADD_LOC	Pearson Correlation	-.012	1.000	.159**	.041
	Sig. (2-tailed)	.839	.	.005	.466
	N	315	315	315	315
INC_LOC	Pearson Correlation	.026	.159**	1.000	.173**
	Sig. (2-tailed)	.643	.005	.	.002
	N	315	315	315	315
DIS_PATT	Pearson Correlation	.130*	.041	.173**	1.000
	Sig. (2-tailed)	.021	.466	.002	.
	N	315	315	315	315

*. Correlation is significant at the 0.05 level (2-tailed).

**. Correlation is significant at the 0.01 level (2-tailed).

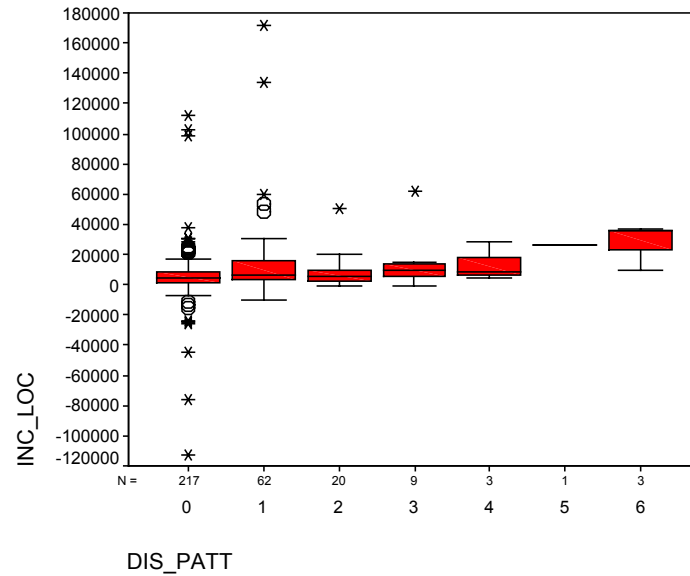


Figure 14: Box-plot of the difference in the increase in LOCs by developers using a different number of design patterns

Next, we divide the population of developers (the developers who added/changed more than 10000 LOCs) into a group of developers who never used patterns and a group of developers who did. Table 10 contains the statistics of the different groups. All medians (except of the number of projects) differ by about 30% between the two groups. Table 11 shows that the location of the distributions of added/changed LOCs and the increase in LOCs differ significantly for the two groups. This suggests that there is a difference between developers who use patterns and developers who do not. Developers who use patterns tend to be more experienced in terms of LOCs they added/changed. Also developers who use patterns are more likely to also create new code (total increase of LOCs) and therefore are also involved in creating new design. In Figure 15 and Figure 16 we visualized the differences between the two groups using box-plots. This finding indicates that design patterns in the analyzed projects are used more often to document new design created by developers who increase the code of the projects than for refactoring by the other developers who modify the code later on.

Table 10: Differences in the variables for developers who use patterns (row 1) and developers who don't (row 0)

Case Summaries

PATTERN		PROJECTS	ADD_LOC	INC_LOC
.00	N	217	217	217
	Minimum	1	3136.00	-112186
	Maximum	7	1023494	111958.00
	Median	1.00	19241.00	4314.0000
	Mean	1.34	39781.59	5456.6175
	Variance	.744	7.68E+09	3.0E+08
1.00	N	98	98	98
	Minimum	1	10055.00	-10638.00
	Maximum	6	547341.00	171691.00
	Median	1.00	28730.50	6519.5000
	Mean	1.44	48162.34	13982.58
	Variance	.847	4.82E+09	5.9E+08

Table 11: Test for differences between developers who use patterns and developers who don't. The differences for the added LOCs and the increase in LOCs are significant. (Two-sample Kolmogorov-Smirnov test)

Test Statistics^a

		PROJECTS	ADD_LOC	INC_LOC
Most Extreme Differences	Absolute	.057	.224	.221
	Positive	.057	.224	.221
	Negative	-.005	-.012	-.005
Kolmogorov-Smirnov Z		.468	1.845	1.812
Asymp. Sig. (2-tailed)		.981	.002	.003

a. Grouping Variable: PATTERN

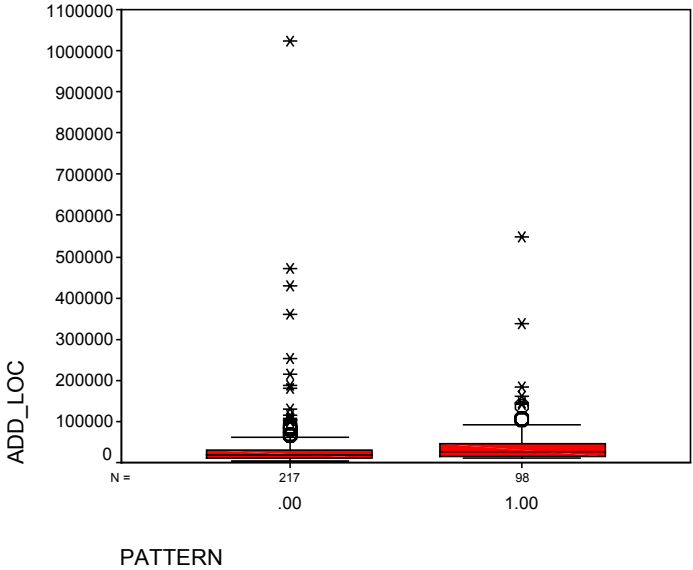


Figure 15: Box-plot of the difference in LOCs changed/added by developers who don't use patters (left plot) and developers who use patterns (right plot)

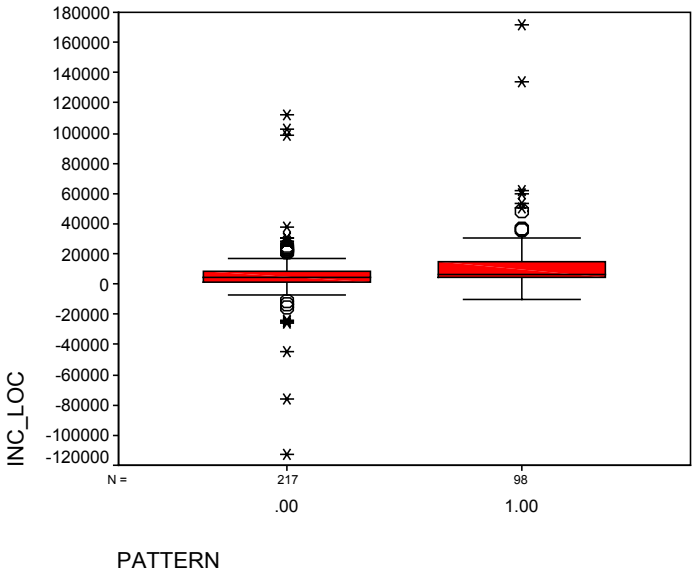


Figure 16: Box-plot of the difference in the increase in LOCs by developers who don't use patters (left plot) and developers who do (right plot)

6 Summary of Results and Conclusion

In this paper we analyzed historic data describing the software development process of over 1000 open source projects using Java. We found out that only a very small fraction of projects use design patterns for documenting the changes in the source code. However, the relationships found in the data set support that:

1. Design patterns are more likely to be used in projects with bigger developer teams (>7 developers). This provides strong evidence that the claim that design patterns are used to improve communication and documentation is reasonable.
2. There exist differences between developers who use patterns and developers who do not. There are significant differences in the total number of code a developer modified or added in the analyzed projects. These differences could indicate that developers with more programming and design experience are more likely to use design patterns. If these developers act rational, this means that design patterns must provide benefits which are valuable for them. These benefits could be that design patterns really represent an efficient way to apply best practices in the form of flexible and robust design.

For the number of different design patterns used in a project and their position in the software life-cycle only a very small correlation was found. Therefore, in the projects in our data set we found no evidence that design patterns are widely used for refactoring. Additionally, we found evidence that developers who mainly develop new functionality (and therefore design new parts) are more likely to use patterns than developers who specialize in modifying existing code. Reasons for this behavior could be that the analyzed projects are still too early in their life-cycle to make major restructuring necessary. Another reason could be that open source development favors more flexible design by frequent modifications and expansion of the code and therefore does not need explicit refactoring as some custom-made systems do.

This first study of the application of design patterns in real-world software development projects has many limitations, e.g. it does not include additional information on the quality of the produced code, the code itself is not analyzed using object-oriented metrics and the actual effort used for the projects is unknown. Also the projects are open source projects which means that the development process differs significantly from industrial projects (see e.g. [MoFH00]). But even with this limitation the quantitative results confirm the most important claim of design patterns, namely that design patterns are used to facilitate communication between developers which, without doubt, is also vital for industrial software development.

7 References

- [ABGM99] David Atkins, Thomas Ball, Todd Graves and Audris Mockus. Using Version Control Data to Evaluate the Impact of Software Tools. In: *Intl. Conf. On Software Engineering*, 1999.
- [AlCM01] Deepak Alur, John Crupi and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, Upper Saddle River, NJ, 2001.
- [AnFC98] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design Pattern Recovery in Object-Oriented Software. In: *Proceedings of the 6th Workshop on Program Comprehension (WPC)*, IEEE CS Press, pp. 153-160, Ischia, Italy, June 1998.
- [BCCD96] Kent Beck, James O. Coplien, Ron Crocker, Lutz Dominick, Gerard Meszaros, Frances Paulisch and John Vlissides. Industrial experience with design patterns. In *Procs. of the 18th Intl. Conf. on Software Engineering*, 1996.

- [BMRS96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. *Pattern-Oriented Software Architecture, A System of Patterns*. John Wiley & Sons Ltd, Chichester, England, 1996.
- [ChKe94] S.R. Chidamber and C.F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6), pp.476-493, 1994
- [CoSc95] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA, 1995.
- [CoVo98] Jonathan E. Cook and Lawrence G. Votta. Cost-effective Analysis of In-Place Software Processes. *IEEE Transactions on Software Engineering*, 24(8), 1998
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.
- [Foge99] K. Fogel. *Open Source Development with CVS*. Coriolos Open Press, Scottsdale, AZ, 1999.
- [GoRi96] Brandon Goldfeder and Linda Rising. A Training Experience with Patterns. *Communications of the ACM*, 39(10), pp. 60-64, October 1996
- [Helm95] Richard Helm. Patterns in practice. In *Procs. of OOPSLA 95*, pp. 337-341, 1995.
- [KoSc02] Stefan Koch and Georg Schneider. Effort, co-operation and co-ordination in an open source software project: GNOME. *Information Systems Journal*, Nr.12, pp. 27-42, 2002
- [MaRB98] Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors. *Pattern Languages of Program Design 3*. Addison-Wesley, Reading, MA, 1998.
- [MaSU99] Gou Masuda, Norihiro Sakamoto and Kazuo Ushijima. Evaluation and Analysis of Applying Design Patterns. In: *Procs. of the International Workshop on the Principles of Software Evolution (IWPSE99)*, Japan 1999.
- [MoFH00] A. Mockus, R. Fielding and J. Herbsleb. A case study of open source software development: the Apache server. In: *Procs. Of the 22nd Intl. Conf. On Software Engineering*, pp. 263-272, 2000.
- [PKG00] Jukka Paakki, Anssi Karhinen, Juha Gustafsson, Lilli Nenonen, and A. Inkeri Verkamo. Software Metrics by Architectural Pattern Mining. In: *Proceedings of Conference on Software: Theory and Practice (16th IFIP World Computer Congress)*, pp. 325-332, Beijing, China, 2000.
- [PrUn98] Lutz Prechelt and Barbara Unger. A Series of Controlled Experiments on Design Patterns: Methodology and Results. In: *Procs. Softwaretechnik ST'98, Softwaretechnik-Trends*, 18(3), pp. 53-60, 1998.
- [Reis01] Ralf Reißing. The Impact of Pattern Use on Design Quality. *Position Paper, Workshop: Beyond Design: Patterns (mis)used, OOPSLA'01*, Tampa, FL, October 2001.
- [Vlis98] John Vlissides. *Pattern Hatching: Design Patterns Applied*. Software Patterns Series. Addison-Wesley, New York, 1998.
- [VICK96] John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors. *Pattern Languages of Program Design 2*. Addison-Wesley, Reading, MA, 1996.

8 Appendix

The design patterns uses for analysis from Gamma et al. [GHJV95] with their intents. Keywords extracted from the intends are printed italic. These words were first stemmed (by removing the endings) and then used to identify the design patterns. The design patterns are organized by the three groups from Gamma et al. into creational, structural and behavioral patterns.

8.1 Creational Patterns

Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Prototype

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Singleton

Ensure a class only has one instance, and provide a global point of access to it.

8.2 Structural Patterns

Adapter

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge

Decouple an abstraction from its implementation so that the two can vary independently.

Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality

Facade

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Flyweight

Use sharing to support large numbers of fine-grained objects efficiently.

Proxy

Provide a surrogate or placeholder for another object to control access to it.

8.3 Behavioral Patterns

Chain of Responsibility

Avoid *coupling* the *sender* of a *request* to its *receiver* by giving more than one *object* a chance to *handle* the request. Chain the receiving objects and *pass* the request along the chain until an objects handles it.

Command

Encapsulate a *request* in a *object*, thereby letting you parameterize *clients* with different requests, *queue* or *log* requests, and support *undoable operations*.

Interpreter

Given a *language*, define a *representation* for its *grammar* along with an interpreter that uses the representation to interpret *sentences* in the language.

Iterator

Provide a way to *access* the *elements* of an *aggregate object* *sequentially* without exposing its *underlying representation*.

Mediator

Define an *object* that *encapsulates* how a set of objects *interact*. Mediator promotes loose *coupling* by keeping objects from *referring* to each other *explicitly*, and it lets you vary their interaction *independently*.

Memento

Without violating *encapsulation*, *capture* and *externalize* an *object's internal state* so that the object can be *restored* to this state later.

Observer

Define a *one-to-many dependency* between *objects* so that when one object *changes state*, all its dependents are *notified* and *updated automatically*.

State

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Strategy

Define a *family of algorithms*, *encapsulate* each one, and make them *interchangeable*. Strategy lets algorithms vary *independently* from *clients* that use it.

Template Method

Define the *skeleton* of an *algorithm* in an *operation*, *deferring* some steps to *subclasses*. Template Method lets subclasses redefine certain steps of an algorithm without *changing* the algorithm's *structure*.

Visitor

Represent an *operation* to be *performed* on the *elements* of an *object structure*. Visitor lets you define a new operation without *changing* the classes of the elements on which it operates.