Master Thesis

Relation between quality of an OO system and multiple inheritance
an exploration

Author(s):
Stokar von Neuforn, David Kaspar

Publication Date:
2009

Permanent Link:
https://doi.org/10.3929/ethz-a-005816759 →

Chair of
Software Engineering

Relation between Quality of an OO System and Multiple Inheritance - An Exploration

Master Thesis

By: David Kaspar Stokar von Neuforn

Supervised by: Yi Wei

Prof. Dr. Bertrand Meyer

Student Number: 03-915-055

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Table of Contents

Abstract

The Object Oriented programming language (OO-PL) design paradigm provides major advantages over the older procedural (PC-PL) paradigm such as enhanced ease, reduced effort and improved management of software reusability both during the development and the maintenance phase.[1]

We test and analyse the correctness of this statement for the maintenance phase, defined as modifications of the code of existing classes for C++, Eiffel, Java and Python.

The analysis ankers on the assumption that polymorphism provides the developer using an OO-PL new alternatives for solving a development task not feasible in a PC-PL. The degree of realisations based on polymorphism, as logical conclusion, correlates with the level of the exploitation of the OO-PL paradigm rendered advantages.

Thus our approach scans through the source repository of different open source projects in order to detect modifications on both the type relation and the procedural level of existing classes. We interpret the detected realised solutions of the developers to categorize on the necessity of using OO-PL specific techniques, to understand the maintenance task and the kind of the modification as for example utilization of polymorphism.

The main result of research is the observed marginalisation of the advantages of the OO-PL's for maintenance tasks, due to the extreme rareness of detected usage of polymorphism in relation to changes on the procedural level.

Secondary findings show that the developer uses polymorphism only, if no alternative solution exists. The development goals are hence mostly on the integration level between mutual components and the developers belong to the core group of the team. Insights on how the various OO-PL's differ can and is thus also provided.

Introduction

Object-oriented programming language (OO-PL) design paradigm's major goal is to improve software reusability by introducing new language constructs in comparison to the procedural programming language paradigm.[1]
An OO-PL characteristics requires each object to belong to a class and to provide support for inheritance. Thus the class entities can cooperate either through delegation (“call”), type-relation (“inheritance”) or the transitive closure thereof. The major difference from PC-PL on the language design level is the unification of the module and type.[2] The new type system hence allows support for inheritance and polymorphism. As code reuse or implementation inheritance stands not on the shoulder of the type system, a similar mechanism in form of modules exists for some PC-PL.[3] Delegation forms the basis of a PC-PL to divide one task in to multiple, possibly sequential sub-tasks. As a result the major new mechanism not present in the PC-PL is polymorphic inheritance.
A software project is required to evolve over time. This can be driven by bugs, a change in the requirement, a technology/design change, a new feature request, a new software quality standard or an integration into a larger frame-work. Intuition tells us that in a object-oriented software some of those changes are applied on the level of the type-relation.
Maintenance actions at the type relation level of classes are traced by our multi-step analysis. We give a qualitative interpretation of the detected quantitative information, some insights and a base to raise further founded questions. The results of the experiment provides the material for a publication stated as the goal of the master project.
The experiment compares different generations of OO-PL via a small set of projects to gain understanding on whether and why there exists differences in the usage pattern of inheritance. In order to limit the scope we specifically looked at the phenomena of multiple-inheritance by a strict distinction between multiple and single inheritance cases throughout the experiment.

Languages Used

This chapter gives a deeper introduction into the OO-PL's used to conduct the research. As the creativity of humans is infinite so is the number of OO-PL's. How can one argue to chose a specific language and neglect another one?
The bottom line of the experiment tries to show how different languages support adaptation or evolution of existing code with a focus on the “inheritance” layer. As time moves on so does technology, new languages are invented and others are threatened with extinction. Our selection criteria is to provide languages representing different decades and concepts of language designs. The basket of OO-PL's representing the OO-PL paradigms different realisations is composed of C++, Eiffel, Java and Python. Accordingly the timeline for the research ranges from 1983 (C++), over 1985 (Eiffel) and 1992 (Java). Out of personal interest on the effect of mix-in inheritance we decided to extend the experiment to include a script language. Python's year of apparel is 1991.

Programming Languages – Idiosyncrasies

C++

The first OO-PL considered is C++[4]. As already stated it is the oldest language in the experiment and provides multiple inheritance. The mechanism incorporated comes with less fine-grained control then Eiffel on the declarative level and relies on operators on the procedural level. The original version included the limitation that the types of methods involved in inheritance can not change and thus covariance was not allowed et all[4]. Later this was relaxed to allow subtyping of return types for implementation of overridden abstract functions.[5] Both versions do not contradict with static type safety.
C++ evolved from the PC-PL C with the motivation of a unification of the OO-PL's paradigm's improved extendibility, reusability and compatibility among the designed components and the power to integrate with C code. This is definition of a hybrid language. Meyer[10] argues that the interleaving of the concepts provides the developer with further alternatives to the usage of the mechanisms of the OO-PL paradigm's. For example with the usage of function pointers at run-time, a C alternative to the concept of polymorphism exists. He further explains that this might lead to a deterioration of the advantages in terms of productivity and quality.

In the experiment the language specific analysis for C++ and Eiffel

searches for mutations along the possibly multiple inheritance axis of existing classes. It annotates for each commit the total number of modified classes categorized into multiple, single and no inheritance users. For a sample of the found modifications manual code review in a second step performs classifications on the various categories such as development goal and effort or implementation versus polymorphism.

Eiffel

The second OO-PL participating in the experiment is Eiffel[10]. The language compromises on static type safety. On one hand as a consequence the set of feasible applications of inheritance and polymorphism is enlarged. On the other hand it introduces the possibility for unintended catcalls. The introduction of anchored types can be used to define covariant definitions adapting to redeclarations. An example for a catcall: A polymorphic target “animal” of the call “eat food” is an instance of “carnivore” and the instantiated type of the anchored type “food” provided at runtime is “grass”. This renders the carnivore to eat grass.
The current version of the Eiffel language supports a single concept of a class. Integration of implementation classes accessible without subtyping is an active research topic. It would allow to distinguish between implementation reuse, without type inheritance and similar to untyped modules[3], and classical implementation inheritance.
Eiffel's design provides multiple inheritance with fine-grained control on the declarative level on how to deal with repeated inheritance.
The Eiffel language introduces the “concept of constants as class types” such as a “class PI” as a type representation of the number Pi. All instances of this class at runtime share the part of the state marked with the Eiffel keyword “once”. Thus the application scope of the concept correlates with the singleton pattern[15] such as shared objects, global system parameters and initialization of common properties.
Eiffel success is based on the natural integration of a formal specifications called contracts allowing for example for automatic test generations and extensive verifications.

Problematic for the experiment is the concept of “constants as class types” as it leads to a massive higher usage of inheritance during modifications of existing classes. In addition the absence of the language construct in the other OO-PL's introduces a bias when comparing the observed usage patterns by overvalue Eiffel's

numbers. A counter-measure decided upon for our research is to mark “language imposed” changes and exclude those from more fine-grained analysis such as “development effort”.

Java

The Java programming language supports two mutual exclusive concepts of classes: Interfaces or abstract classes and normal classes. An interface contains only method signatures and constants whereas a normal class carries the implementation details as well. Hence Java further differentiates between implementation and type inheritance. Interfaces can exclusively inherit from possibly multiple other interfaces through type inheritance. The language limits a Java class to a single other class as ancestor but enables possibly multiple interfaces as polymorphic types. Covariance thus prohibited for parameters and return types exists in the special case of arrays. The virtual machine monitors at runtime the required type conformance.
The new version of the language Java (J2SE 5) provides covariance for the return type but not for parameter types of a method. The introduction of non-covariant generic arrays removes the issue explained above.[6]
Java is a derivative of C++. A key motivation is to improve productivity in terms of development effort by enabling reuse through enhanced support for the techniques of the OO-paradigm. For our experiment Java with it's distinction between interfaces and classes represents a special case. The classical understanding of multiple inheritance as mix-in of different implementations from mutual parents is not supported. Still a specific class can have multiple ancestor types from both interfaces and classes. We decided to analyse both hierarchies separately. For future research we annotated the analysed commits in the manual review phase with additional and language specific information. For example whether the addition or removal of an interface leads to a large implementation effort. We measure for Java the number of mutations on the interface hierarchy as well as on the implementation inheritance chain.

Python

The fourth and the last OO-PL to gain knowledge from is Python[7]. The language is a derivative of C, C++ and Pascal.
As a dynamically typed language it's right to represent the classic OO-PL paradigm might be questioned. But we argue that Python represents the dinosaur of a creative new generation. The goals as

high productivity and adaptability are benchmarked on the first and second generation of OO-PL's as Java. They invented or rediscovered new mechanism used in orchestration with classic OO-PL's tool kit of inheritance, polymorphism or unification of type and class.
Some of the concepts require interpretation and dynamic types. Dynamic types allow an object to switch between the parents “UDP Socket” with “TCP Socket” on request by adding and removing the classes from its ancestors. Interpretation enables meta-programming. The solution approach is based on dynamic alteration of the behaviour of the interpreter. The interpreted code modifies the behaviour on how the future interpreted code is perceived by the virtual machine.[8] For example one can program a proxy providing callbacks each time a method is executed on a target object.
Python supports multiple inheritance. As the language does not provide declarative elements to steer the resolution procedure, the interpreter relies on two different resolution algorithms. The older is explicitly based on the control-flow. For example given a list of multiple parents all providing the same feature, the head serves the calls. Of course this naïve approach has various limitations when exposed to interleaved hierarchies.[18] The newer implemented as of Python 2.3 is more complex.
Another innovation is the so called “call-next” precursor. It enables a child class to selectively execute any of the possibly multiple parent implementations of a feature.
In the classic Python types and classes are not unified. Thus the instance of a the so called “classic style” class is of type “instance”. In order to improve type safety and enhance understandability “new style” classes were added in language version 2.3 introducing the direct mapping between types and the classes. The programmer has to decide which class model fits best her requirements. “New style” classes are marked by the addition of a “new style” root class.
Aside of classes Python uses the concept of modules. A module corresponds to a static singleton. It is initialised the first time the code is interpreted.

For our experiment Python can be treated as C++ and Eiffel. Modifications on existing classes to migrate from the “classic style” to the “new style” class model are not counted for the comparison and thus marked as “language imposed”.

Projects Used

For the experiment we need projects to provide the information to allow an inter-language comparison among Eiffel, C++, Python and Java.
The composition of projects for each language should include a high degree of similarities on various domains. In addition some side conditions, such as the access to a source code repository and the presence of a deployed successful product, must be respected.
One domain is the experience of the developers in large projects and in their language. Parnas states that more mature developers also fall into the pitfall to focus on the first release but not as deep as immature. They also keep an eye on “design for change” by applying the techniques of the OO-PL paradigm as for example “information hiding”, “abstraction” or “dynamic binding”.[9] Meyer[10] explains, that from his perspective only the author of a widely reused library is a true expert for reusability. Thus homogeneity on the experience level of the developers involved is a requirement for our experiment.
We think that another domain of concern is the application scope. Because for example the OO-paradigm fits in well for a GUI application but not so for a mathematical library. Selecting thus all the projects of language A as GUI applications and the others of language B as mathematical libraries could introduce a bias into the comparison between the languages. Pure libraries are excluded, because many of their interfaces are bind by the users and thus not modifiable.
In theory in order to guarantee a perfect setting one requires identical projects in mutual programming languages that are collectively exhaustive to get a perfect picture of the differences among all language features. Reality's imperfection let projects differ in the specifications, the experience of the programmers and even in the number of other reasons.

While we accept non-perfection for the experiment, one can still argue qualitatively that a strong intersection of two mutual specifications implies a similar distribution of tasks a programmer is exposed to. Accordingly the result on the language level can show meaningful trends.

C++: CodeLite, QPid, Slicer

- CodeLite is an open source project of a C/C++ IDE with three authors
- Qpid is an open source messaging implementation built on

AMQP, that allows to implement just about any distributed or business application. Client API include C++, Java, Ruby, Python and C# for .NET. The project is developed by 15 people.

- 3D Slicer is an open source a multi-platform software for visualization and image computing. 42 authors are involved.

Eiffel: Goanna, Gobo, EiffelStudio

- Goanna provides a web services framework for Eiffel on top of Apache. The library contains clusters that support servlets, end-points and XML to DOM parsing. 13 developers are involved.
- Gobo-Compiler and library is an Eiffel compiler developed by 13 persons.
- EiffelStudio is an integrated IDE and a compiler for Eiffel. We detected commits from 93 different persons.

Java: Hadoop, Tomcat, ActiveMQ

- Hadoop Core provides a distributed file system (HDFS) and framework for the Map Reduce distributed computing metaphor. The group of authors size is 22.
- Tomcat is an implementation of the Java servlet and JavaServer Pages technologies acting as a application server. We found commits from 14 persons.
- ActiveMQ is a complete message broker and full JMS 1.1 provider featuring clustering, distributed destinations and XA support with pluggable persistence. It is developed by 12 authors.

Python: Deluge, Twisted, Zope

- Deluge is a plug-in-based Bit Torrent client for Linux, MacOS X and Windows developed by a group of eight persons.
- Twisted is a networking engine written in Python, supporting numerous protocols. It contains a web server, numerous chat clients, chat servers, mail servers, and more. Twisted could attract 47 persons to participate in providing code.
- Zope is an application server for building content management systems, intranet applications, portals, and custom applications. Zope motivated 96 persons to commit code.

Experiment Procedure

Assumptions

What must hold in order to provide a fair study? The information gathered from the projects is correct and feasible to reason on more abstract levels, such as the programming language or the OO-PL paradigm.
As analogy imagine two programmers. They are located at two places and don't know each other. Both get their hands dirty in two different GUI application projects but with two different programming languages. They receive the same set of tasks consisting of 100 new features, 20 refactorings, 30 bug fixes and 15 integrations of upgraded 3rd party libraries. Do you think they implement the various tasks in a “similar” manner, given that they share the same level of experience in their respective OO-PL? For example do they employ a similar amount of new polymorphic relations on existing classes? We assume, this is the case if the language is the same. As a conclusion the reverse states, that the differences in the treatment of existing classes correlates with the differences between the languages.

According to the analogy to have comparable projects, they must expose the developer to similar kinds of tasks and to a similar distribution among them.

Of course the basic assumption of independent, identically distributed samples as a basis of a statistic analysis must be questioned for the selection of the projects. There exist too many factors quantitative or qualitative that imply correlation and dependence among different subsets of the chosen projects.

One can expect similarities based on the programming language, as mentioned the experience level of the programmers, the drivers of change of the requirements, the quality of the software development process and many more.

In order to have developers of a similar level of experience in their respective programming language and project, one has to select the people with the same responsibility or role. We can show that the manually analysed commits mostly come from the core developers of the respective projects.

The assumption that projects need to expose the developer to a similar distribution of similar tasks can not be proofed. Still it is tried to be honoured by selecting projects with similar requirements as an IDE, application server or protocol stack.

Procedure

The experiment is a multi-step analysis of the information present in the source code repository of the various projects.
A first step is to decide on the set of languages. Details leading to the selection are explained in the “languages used” chapter.
For each programming language a set of projects, satisfying specific criteria in order to provide comparability on the level of programming languages and on the OO-paradigm, is determined as explained in the “projects used” chapter.
Via an automatic extraction program the source code repositories are processed. The results of the analysis is stored in a per project file. We count the number of changed lines of code (cLoC), the number of added and deleted classes and the modifications on the type relation layer of the existing classes for each commit. The procedure of this step is documented in [19].
In an intermediate step we perform some basic statistics on the projects and identify the “hot spots” for our manual code review phase. In order to better understand the reasons for the evolution of existing classes to and from multiple or single inheritance, we focus on commits containing this modifications, the “hot spots”. Further information on the classification can be found under “Type relation changes and categories”.
For some projects the large number of identified commits requires sampling. We used a confidence interval of 6% and level of 95%. For EiffelStudio and Goanna this was not feasible and thus we relaxed the confidence interval to 10% for the two only. Of course on one hand this can introduce a problem, on the other hand the number of samples for this two projects is still significant larger in comparison to the others sometimes having an extremely small population. Further the reviewed samples of EiffelStudio and Goanna show clusterings and overlaps in the kind of tasks and distribution providing evidence for an in fact smaller confidence interval and more trust for the differences among the percentages.
The manual code review of the selected “hot spots” of the various projects categorizes the modification leading to the change of an existing class on the type relation layer. We interpret the underlying development goal such as “integration”, “requirement implementation” or “enhancement”, “bug fix” and “refactoring”. The rules describing the differences can be found in “Categories for Development Goals”. We try to understand, whether an alternative approach to reach the identical “development goal” exists and estimate the effort. The criteria to distinguish among “small pain” or “small effort for an alternative” to “large pain” or “no alternative possible” is explained in “Categories for "level of pain"”. The actual

OO-PL technique used to successfully reach the development goal is annotated too.
After the manual code review aggregations are performed on to the level of projects and languages by averaging. For each classification target we calculate the observed distribution on the indexes. In order to understand the large differences among the entities we show the distribution in per mil of totally observed commits with an alteration on the type level of existing classes. This renders all the bars of the per mil charts 1:1 comparable.
In the interpretation part we explain the major observations and conclusions on the language versus language level (“Results: OO-PL versus OO-PL”) and on the project level of the various OO-PL's for example (“Results: Python project versus project”).

Classification of Changes

There are many numbers and metrics, that exist in research. The main difference to this analysis is, that the focus lies on the first time-derivation and requires manual interpretation of details, that are automatically not accessible. To state this differently, rather then calculating metrics for one specific version of the software to detect, for example, places of interest for a refactoring[17], we are interested to quantify the changes or the evolution of the software over time similar to [16].
For example given an existing class of Java in revision 0 and revision 1. The evolution can happen on the two levels of OO-PL paradigm: Delegation and inheritance. The first level requires modifications exclusively on the procedural level by replacing code of an existing member function or adding a new feature. The second can be a combination of the procedural and on the declarative type-level relation layer by, for example removing an interface or adding a new one. For Eiffel, as interfaces are not part of the language, the analogy is to add code for a new feature, representing a procedural level change, or by adding a new parent, an alteration on the type-level relation layer.
The focus lies on the understanding of the phenomena of changes on the type-level relation layer for existing classes.
We categorize the development goal, the effort required for an alternative solution and the technique applied.

Type relation changes categories

The first metric is the number of changes to a single inheritance class of a specific commit. The results reside in the “used.data” folder.

Single Inheritance Class

a) No Inheritance + 1 Parent = Single Inheritance Class
Old class: “class House”
New class: “class House inherits Building”

b) Multiple Inheritance – All but one parent = Single Inheritance Class
Old class: “class House inherits Building, Mortgage”
New class: “class House inherits Building”

c) Single Inheritance + Exchange Parent = Single Inheritance Class
Old class: “class House inherits Building”
New class: “class House inherits Mortgage”

Multiple Inheritance Class

The second metric is the number of changes to a multiple inheritance class.

a) No Inheritance + 2 Parent = Multiple Inheritance Class
Old class: “class House”
New class: “class House inherits Building, Mortgage”

b) Single Inheritance + 1 Parent = Multiple Inheritance Class
Old class: “class House inherits Building”
New class: “class House inherits Building, Mortgage”

c) Multiple Inheritance + 1 Parent = Not counted
Old class: “class House inherits Building, Mortgage”
New class: “class House inherits Building, Mortgage, Facility”

Added and deleted parents

In order to get a better understanding of the absolute number of modifications on the inheritance graph, it counts as well the total number of deleted parents and added parents.

Special case Java

As a special case Java does not allow multiple implementation based inheritance. Therefore the changes on the interface graph is interpreted in a similar way.

Categories for "level of pain"

A level of pain measures, for a specific change leading to the modification of an existing class on the type-relation level, the effort required to produce the same result through implementation of a different approach, not requiring inheritance.

Small pain or small effort to develop an alternative

“Small pain” are changes that have a straight forward procedural implementation that does not require a large effort by the developer. Thus such a change can not contain a dependency on polymorphism.
Polymorphism is excluded, because the development effort to reimplement a solution providing the same functionality to multiple types is not meaningful or even impossible. For example the addition of an interface used in a 3rd party library in order to integrate a service, can not be circumvented and thus no alternative exists.
If the modification enables to share code for the implementation of the functionality provided by the class the discriminatory properties are the following. First the number of children sharing the code of a parent must be equal or less than three. They must reside all in the same component. If this properties hold, the number of shared lines, of code involved, comes in to play. If it is less than 40, then the change is labelled as “small pain”. A possible alternative could either use the parent as a member providing only functionality or reimplement some of the code in the at most three children. Of course the first solution is logically wrong, when no “is-a” relation exists.
For example given a commit, that contains the modification of an existing class “Foo_1”. A new parent is added “Foo”. It is used solely for the purpose to share code with the new class “Foo_2”. “Foo” as a type is not required. The motivation for the developer is to “encapsulate repetition” as she “hates tedious, repetitive tasks” (Meyer[10] on Implementation Inheritance). As an alternative the developer provides similar code in the two classes “Foo_1” and “Foo_2”. It requires not a large effort and is not dangerous for non-critical code.

Medium pain or medium effort to develop an alternative

“Medium pain” modifications demand a larger effort to develop an alternative than “small pain”. A change in this category can be involved in computations requiring dynamic binding. We distinguish between two cases. The first is the evolution of an existing polymorphic hierarchy. The second represents the introduction of a new set of classes interacting without component-external coupling of the abstract types.
For implementation reuse of a larger set of children involved, not residing in different components, we provide this label. It marks also the usage of functionality from a different component or library, but the reused code requires to be small in lines of code and non-critical. For example the inheritance from a string library to enable

the feature “reverse” is in the “medium pain” category. The usage of code, constants or functionality on the complexity level of a security, portability or compliance critical feature, for example “input/output”, requires “Large Pain” classification.
The evolution of an existing hierarchy involved in polymorphism or covariance can be tagged as “medium pain”, but not for the addition of a new dynamic binding relation. For example given three classes “engine”, “diesel” and “gasoline”. “Engine” represents the type instances of the other two are dynamically bind. The developer adds a new class “combustion” and removes the “diesel” and “gasoline” unifying the functionality of the two. Hence no new polymorphic relation is introduced, but an existing modified.

Large pain or large effort to develop an alternative

All the changes, introducing an additional parent type for polymorphic usage, fall into this category. As mentioned in the “medium pain” category's explanation reimplementation of a feature, providing security, portability or performance enhancement rather than to gain access through inheritance,is not meaningful and even wrong and thus those changes are valued as “large pain” too.

Language imposed

For Python and Eiffel some special tasks are normally or even required to be solved by inheritance, where this is not required for the other languages. Thus comparing this changes to the others would challenge the justification for inter-language comparisons. This is the case for Eiffel as it provides a singleton pattern on the level of the language. For Python multiple class models exist and the developer can choose among them through adding a specific base class.
More on this topic is provided in the “Languages Used” chapter. Some real observed examples can be found in the chapter describing the results for Eiffel and Python.

Categories for “development goal”

The “development goal” provides a set of disjoint tasks a developer can perform being corrective, adaptive, perfective or preventive[11]. During the manual code review of the “hot spots” located inside a commit, we try to understand, what was the actual reason to add or remove possibly multiple parents to the existing class of our focus.

Requirement

If the change on the type level relation of the existing class is motivated by the implementation of an enhancement as a requirement or a performance improvement.

Refactoring

A refactoring is usually a side-effect of another development goal. The main criteria is, that the provided and used functionality of the class or the set of interacting classes remains the same.

Bug fix

Removal or replacement of code, leading to a non-correct behaviour of the software, is labelled as “bug fix”. Either it is explicitly stated in the log of the commit or we could identify evidence in the code through a comment or an issue number of a bug-tracking system. Bugs fixed on the declarative level of inheritance are many times the integration of a missing service. The manual interpretation can be tricky, because many times as a side effect a refactoring of the code is performed.

Integration

Most of the time integration task rely on some kind of polymorphism or covariance adaptations. They usually involve the interaction of different components.

Categories for “classifications of alteration”

Today word-count statistics on publications[12] are required to provide evidence what concepts characterize the OO-paradigm. Only “inheritance”, “object”, “class” and “encapsulation” manage to survive the threshold of 60%.
Thus rather than relying on complex taxonomy of kinds of inheritance, we narrow ourselves to the observable evolution of classes using inheritance. We distinguish for the motivation of a change between participation through the parent in a polymorphic structure or to share functionality.

New polymorphism

The added parent enables access to a newly created polymorphic relation. Many possible scenarios exist, for example the introduction of a new abstract class to represent a clean interface of an existing class. As the instances of the child are assigned dynamically to the type of the parent, the changes introduces a new polymorphic relation.

Existing polymorphism

The only difference to the “new polymorphism” category is that the polymorphic relation already existed. For example to an existing class an existing parent, coupled to a desired framework through it's interface, is added.

New implementation

Modifications tagged with this label are enabling the reuse of code to provide the required functionality of the class. For example a set of classes organized to share the implementation of a common interface. If the parent class is new or did not have children, we denote the “new” of “new implementation”.

Existing implementation

Similar to the “new implementation” but the child is added to a parent with at least one already existing child.

New divide/merge

Sometimes in refactorings or in the implementation of a new requirement one splits an existing class in two disjoint pieces. For example the class “drink” is split into “non-alcoholic” and “alcoholic”. Class “drink” is removed and the distinction between the two new abstractions implemented. Thus all the former children of class “drink” have to be reclassified and adapted to inherit from either one of the two.

Results: OO-PL versus OO-PL

Introduction

The second part of the report presents the results of the experiment.
It starts with an “OO-PL versus OO-PL” analysis of the evolution of existing classes on the type relation level with respect to inheritance and polymorphism. The motivation is to understand the generalised differences and similarities among C++, Eiffel, Python and Java. The selection of this languages allows to reason about the OO-paradigm in general as explained in the “languages used” section.
The chapter is organised as a break down of the findings from a high level of abstraction to a low level with more detailed information.
A change on the type-level of an existing class is a melange of different decisions and steps. Thus imagine a developer sitting in front of a computer with a task. To solve it, she decides on a possible solution and changes the code accordingly. To finalize it, she persists the modification through a commit to the revision repository.
The chapter describes each step backwards. We start with the set of all commits of all projects. As a first step we remove all the commits without a change in the programming language but, for example, in the text files. Further details can be found in the “all commits versus changes to OO-PL code” subsection.
The changed lines of code must include a modification of the set of parents of an existing class, through, for example, the addition of a new one, as explained in the subsection “Changed commits versus changed existing type-relations”.
The imaginary developer decided to perform an adaptation on the type relation level of an existing class. To understand its necessity, the cost to develop an alternative solution without inheritance is measured. The subsection “Changed existing type-relations versus "level of pain" ” contains the measured results and an interpretation.
“No change without a reason” and thus the development goal such as “refactoring” or “integration” for the set of alterations is determined and presented in the subsection “changes to existing type-relations versus the classification of the development goal s”.
The level of expertise in the OO-paradigm's methodology and the degree of knowledge of the software has an impact on, whether or how the imaginary developer is capable to perform “design for

change” and influencing the degree of software ageing[12]. We analyse the size of the subset of all developers of the project with a responsibility for more than 80% of the alterations on the type relation level following Pareto's law. The insight gained is interpreted in the subsection “experience of the developers – the 80/20 rule”.
The modification on the type relation level of an existing class is embedded into a combination of different development goals combined in one commit. The size of the commit in changed Lines of Code (cLoC) is presented in the subsection “Changed existing type-relations versus changed lines of code ”.
The final subsection contains a presentation of the found differences of the usage of polymorphism and code reuse inheritance alterations of the reviewed existing classes.

All commits versus changes to OO-PL code

Not all modifications of the projects contain programming code in the principle programming language used for our research.
For example a difference between two revisions including only changes to an XML-Configuration file does not count as a commit changing the code base.
We analysed the percentage of commits with at least a single changed line of code (cLoC) with the the total number of commits automatically processed.
Chart-1 presents for each language the aggregated average percentage of commits that include a change in the respective programming language (blue horizontal line) and the range of values for the underlying projects (blue vertical line).

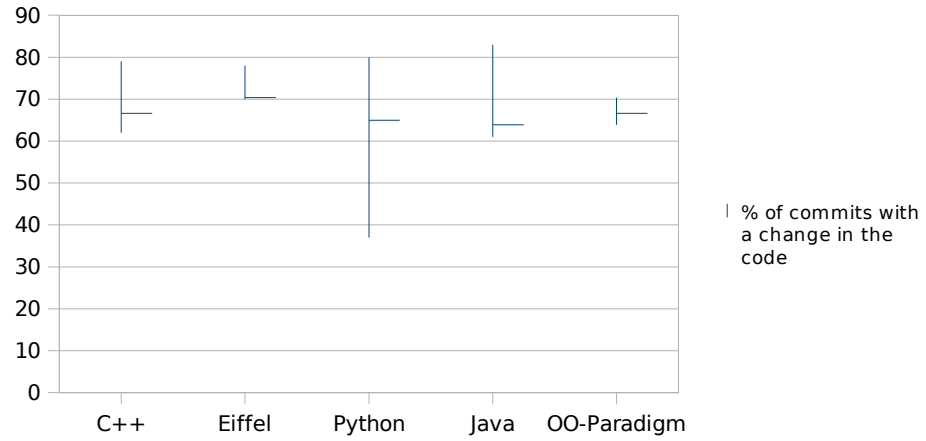

Chart 1: Changes of code [percentage of all commits]

We conclude, that, while the percentage for all OO-PL's is similar,

this does not hold for the size of the range.
For the C++ projects observed around 65% of the persisted modifications alter the code base.
Eiffel has the highest percentage (around 70%) and the smallest dispersion among the projects.
Python behaves similar as C++ for the average. More interesting is the fact that the variability among its projects is, as in the Java case, extremely large. The low outlier represents the Deluge project. The large percentage of commits, without any modification in Python, target the HTML pages of the web user interface. More explanations for the underlying reasons can be found in the language level analysis chapter “Results: Python projects versus projects”.
Java has an average of 63% and contains the maximum value for the project ActiveMQ with 83%, an message broker engine.
On one hand the languages don't have major differences on how often in average the underlying code is actually changed. On the other hand the projects disparity on the language level is large and requires further explanation.

Changed commits versus changed existing type-relations

To modify an existing class requires to commit lines of code. How

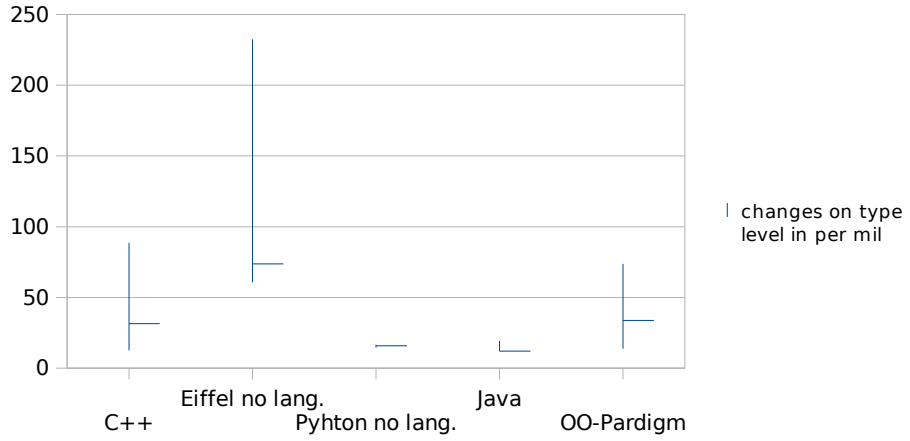

Chart 2: Changed existing type-relations [per mil of changed commits]

often does this event contain at least one change on the type-level relation layer of an existing class? As stated in the motivation chapter this number correlates with the level of the OO-paradigm's advantages realised for maintenance support. Thus we present here the main answer of the conducted research.
Chart 2 shows, for each programming language, the average size of the subset (blue horizontal line), with an alteration on the type-relation level, in relation to the set of commits, with a change of an

existing class, in per mil. The blue vertical line visualises the range of values of the underlying projects.
Our conclusion is that a quick look on the chart is enough to understand that, for this set of data, differences among programming languages, as expected, do exist. The OO-Paradigm's value is based on the averages of the languages. Hence on the top-level of abstraction out of 1000 commits changing existing classes only around 30 adapt the type relation to exploit new productivity gains. For example of EiffelStudio's 44k commits, with a detected change of existing classes, less than 3k contain at least a single adaptation of their set of parents. The reason for the small number is further explored in the following subsections.
Under the assumption that adaptations are good, as justified in the abstract, one can declare a clear winner in the OO-PL shoot-out: The Eiffel Programming language. The competitors remain in the small interval of 16 per mil to 31 per mil. Beware of the fact that although this work has been done under the supervision of the inventor of Eiffel and Design by Contract the findings are mere statistics and observation and not just yet-another-way-to-self-adulation.
This result and the special situation of Eiffel are further explained in the the following subsections and the “Results: Eiffel Project versus Project”.
As a final remark to each developer, whenever you add a new class to a project, the chance of exploitation of the advantages of the OO-paradigm at a later point in time is extremely rare. Thus one should be extremely careful on the decision of how to place the class into the existing hierarchies before the initial addition.

Changed existing type-relations versus "level of pain"

Various alternative solution approaches exist to implement the corrective, adaptive, perfective or preventive maintenance task on an existing class. In the rare case that a developer decides to adapt the type-relation level, what is the cost for an alternative solution approach? To state it differently, do developers only change the set of parents of an existing class, if no alternative approach exists?
Chart 4 presents the fact, that for nearly 50% of all adaptations no meaningful alternative could be identified, according to our classification system described in “Categories for "level of pain"”. “Large pain” implies the usage of either polymorphism or critical functionality, for example portability issues.
Chart 3 presents the weighted average distribution of cost for an alternative to the observed changes on the type-level relation layer.

The sum of the bars for each language add up to the per mil commits with changes on the type level relations of Chart 2. This allows direct inter-language comparison of the bar heights. For each languages chosen samples of some categories can be found in the section on the language level results.

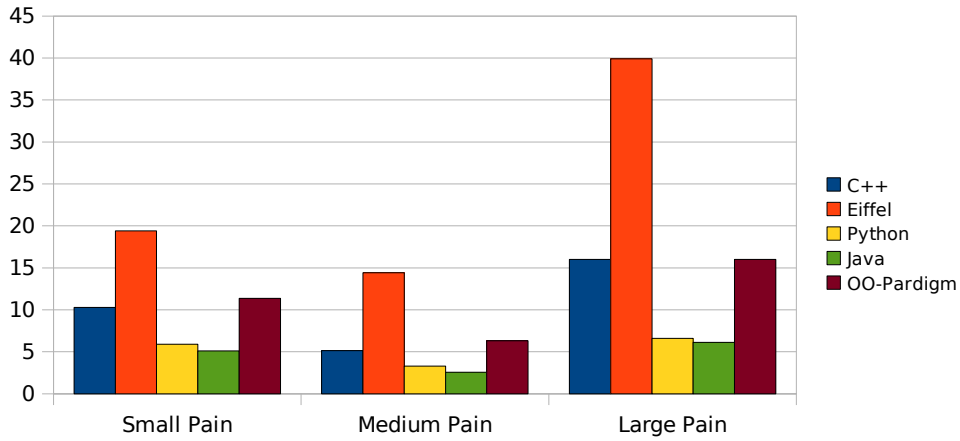

Chart 3: "level of pain" [per mil of changed commits]

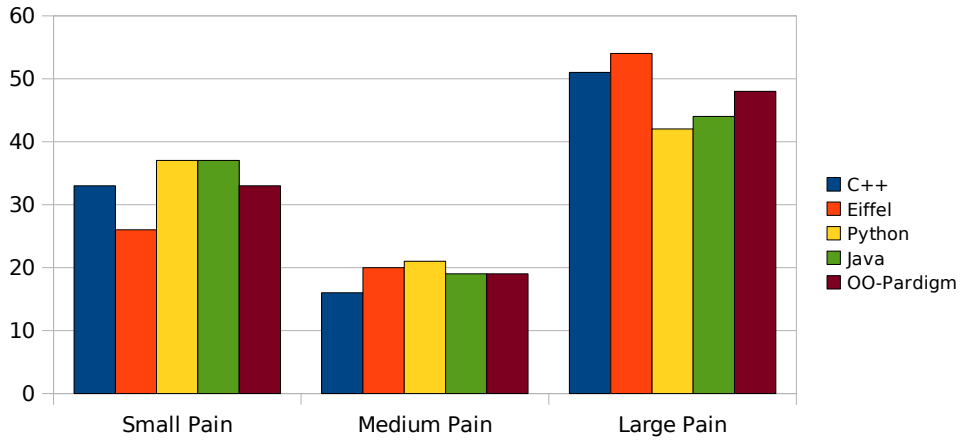

Chart 4: "level of pain" [percentage of observations]

The homogeneity of the percentages in chart 4 show that all languages share similar ratios among the different “levels of pain“. Thus the the major differences in absolute observations in per mil of commits imply that certain languages have a generally higher level of the usage not in a single but in all categories. Still minor differences among the languages are present.
All the languages have their maxima in the “large pain” category and their minima in the “medium pain” category.
For Eiffel the “large pain” category is slightly stronger and the “small pain” category the same amount weaker. In the per mil chart

it dominates all the categories with the fiercest competition in “small pain”.
C++ compensates for the high percentage value for “large pain” in the “medium pain” category. The language protects the second place in chart 2 for all categories being challenged in the “medium pain“ category by Python.
Python has the the lowest maxima with only 42% in “large pain” and the strongest value of the “small pain” category with 37% and with 21% in the “medium category”. The differences in the per mil chart in comparison to Java are so small, that no generalizable, significant difference exists.
Java's maxima is only 45% and the “small pain” category thus rather strong with 37% as well. In the per mil chart the language, as stated above, behaves identically to Python.
Java is one-of-a-kind by the absence of multiple implementation based inheritance. More than 50% of all it's changes are member of the “large pain” set. This can be explained through the observation that nearly 57% of all observed modifications include polymorphic usage of the added interface. It is no surprise that the addition of a at least one interface usually requires the implementation of additional code as observed in 28% of the cases.
As a main conclusion so far the article presents evidence that the complexity of the task patterns that lead to changes on the type-level for existing code is not equal or equally distributed for the different programming languages. It implies hence that the language design has a major influence on whether the developer solves the task on the procedural or on the type-level.

Changed existing type-relations versus "development goal"

“No change, without reason” or we assume that the developer never performs a modification of the code without an underlying motivation.
We provide a classification system in the “Categories for “Development Goals”” consisting of the four indexes “requirement”, “refactoring”, “bug fix” and “integration”. This allows us to understand, which of development goals more often require an alteration on the type level of an existing class.
Chart 5 presents the weighted average distribution of the task categories of the observed changes on the type-level relation layer. The sum of the bars for each language add up to the per mil commits with changes on the type level relations of chart 2. This allows direct inter-language comparison of the bar heights.

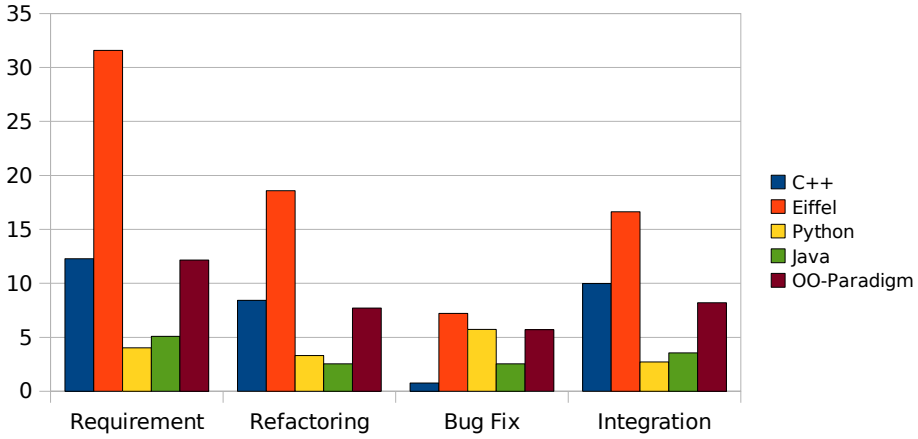

Chart 5: "development goal" [per mil of changed commits]

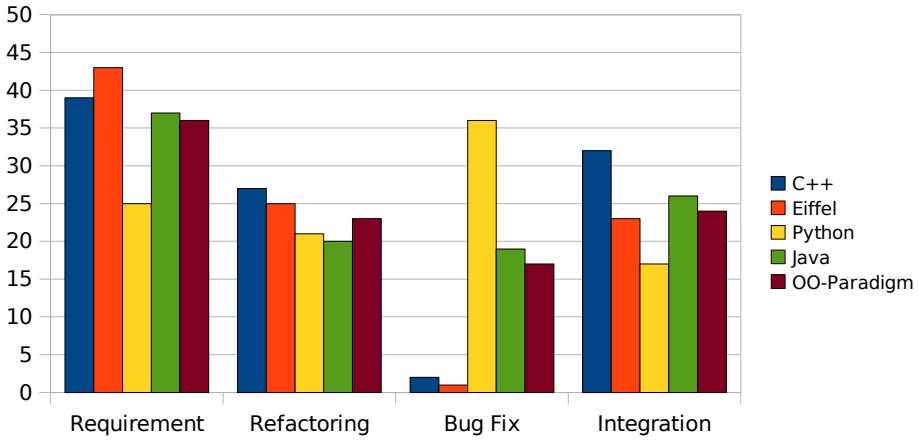

Chart 6: "development goal" [percentage of observations]

At glance chart 5 and 6 distributions lack homogeneity and thus contain differences among the languages.
The commonalities of all languages are that each one provides at least one global maxima or minima in one of the charts.
C++ provides the highest percentage of refactorings and the lowest per mil and percentage value of bug fixes. The per mil comparison with Eiffel makes it clear that for all categories Eiffel developers use more type-relation level changes. The “requirement” category shows the largest absolute and percentage difference and thus we can conclude that C++ can not enable the same amount of reuse and productivity gains.
The special role of Eiffel is highly visible, during a first glance over the chart 5, owning the largest value in each per mil category. Nevertheless in the percentage distribution one can identify a marginal part to be in the “bug fix” category for both Eiffel and C++. It is interesting to learn that Eiffel developers perform a similar amount of changes for refactoring and integration.

Python is the sole language with its highest percentage in the bug fixing category and a minimum for integration. The high value for bug fixes is a specific design pattern used and leads to a similar value in per mil as Eiffel. This is remarkable as Eiffel has in total a four times higher level of per mil commits. Further details on how experienced core-developers of the Python projects perform their bug fixing, can be found in the “Results: Python Project versus Project” section.
Java's global minima is the per mil value of the category “refactoring” but the difference to Python is statistically not significant. The shape of the distribution of both charts resembles the aggregation on to the level of the “OO-Paradigm” and is remarkably balanced.
To conclude that the modification of an existing class at the type-level relation layer is mostly used to implement a new feature or enhancement, holds for all but one language. This might be a surprise, because all of the projects under research are delivered to end-users in different versions. Therefore the domination might be expected to lie in the refactoring or bug fix category.
Eiffel, Java and C++ share their largest percentage in the same category of “requirement”.

Experience of developers – The 80/20 rule

One assumption is that the developers must have a similar level of maturity in order to allow comparisons among the projects and at a higher level among the languages. Chart-5 presents the percentage of developers responsible for 80% of the modifications of different categories in order to understand, who the developers are.
Since the main focus of our research lies on the changes to existing classes at the type-level, the assumption seems justified that in all programming language only the most experienced authors, roughly 15% of all, are responsible for the modifications. The onion model for open source project declares this set as the core development team.[13]

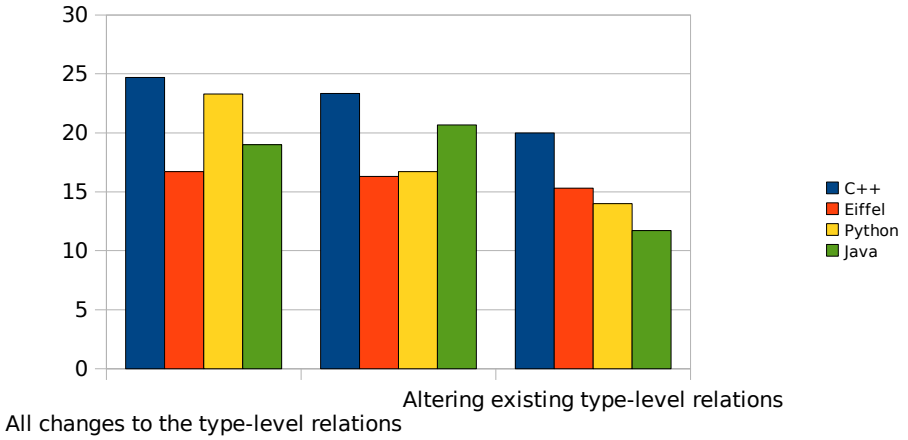

Chart 7: Percentage of authors performing 80% of changes

The first column of bars represents all changes to the type-level relations. This composition includes all modified existing classes on the type relation level, but also the addition of a new class to the source code repository. The second column “Changes of Code” are the average percentage of authors responsible for 80% of the code changes of a software measured in changed lines of code (cLoC). The last column contains the different percentage of developers responsible for more than 80% of the changes of existing classes on the type level.

For C++ the roughly 25% of all authors perform modifications of the first category and this set is decreased to 20% for the thus most challenging task of the evolution of the existing classes ancestry. In actual numbers the range of developers in the last set is single digit only.

For Eiffel the decay from the maximum in the first column to the minimum column exists, but is marginal. Nevertheless the 15% of developers in the last column represent also a small group of two to 15 people.

Python's distribution shows a significant difference between the first and the last column. The 14% of authors of the last column contain for one project only a single developer and for the rest less than 15 persons.

Java is unique that the percentage of developers responsible for more than 80% of all counted changed lines of code is larger than the percentage responsible for the type modifications. The set of maintainers is the smallest with all projects core team of less than 5 persons.

As a conclusion Zope and EiffelStudio have the largest development teams with 13 and 15 people and also the largest set of authors. Intuition approves the results. Since to perform large integration tasks or complex requirements implementation, the developer must

have a very deep and profound knowledge of the whole system and matches the role description of a core development team.

Changed existing type-relations versus changed lines of code

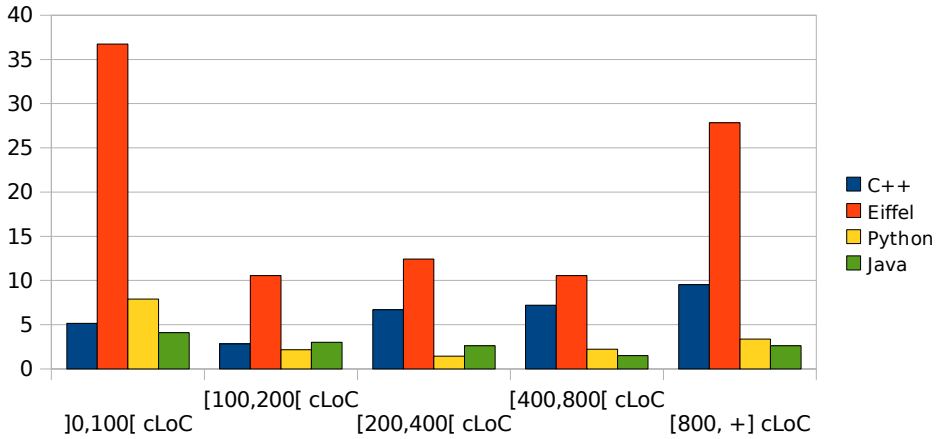

Chart 8: Changed lines of code [per mil of changed commits]

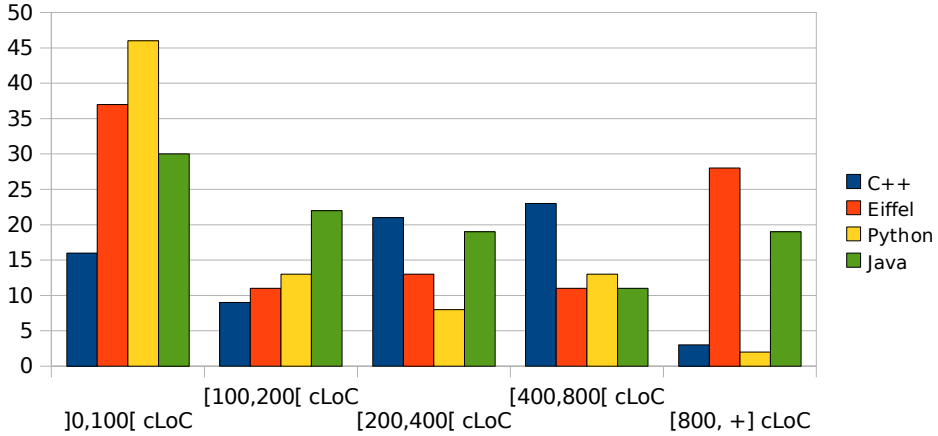

Chart 9: Changed lines of code [percentage of observations]

Chart 6 shows the size in changed lines of code of the commits including an alteration on the type-level relation layer. Aggregation to the level of the OO-paradigm seems not to be justified as the the mere purpose of the presentation is to get a feeling for the different approaches to the usage of the source repository.
For C++ 25% of the commits contain less than 200 lines of code. On the contrary more than half of it, have a size of more than 400 lines of code. Most of the refactoring's for example are part of huge commits but are actually just a side-effect of an experienced developer reviewing code or merging a development branch. Thus

the source code repositories commit policy has a large impact all the numbers in all the charts.
Eiffel has two peaks on either end of the scale. More than 35% are small commits of less than 100 changed lines of code and more than 25% are huge modifications, as for example the integration of a new development branch or libraries relying on auto-generated code, that required an update or alteration.
Python is the unmatched leader in the sense that most of the alterations are relatively small and for this reason possibly easy to understand and interpret. This is no surprise, as a large part consists of bug fixing and thus involve a rather small code base. Further the development in a script language with agile methods lead to more but smaller changes in terms of changed lines of code.
Java has an nearly even distribution over the different categories.

Changed existing type-relations versus "classification of alteration"

The final part of the presentation of the results is about the actual OO-mechanism employed to perform the observed modification. The different categories distinguish foremost between usage of polymorphism and implementation. Guidance on how the disjoint categories are defined, can be found in the “categories for “classification of alterations””.

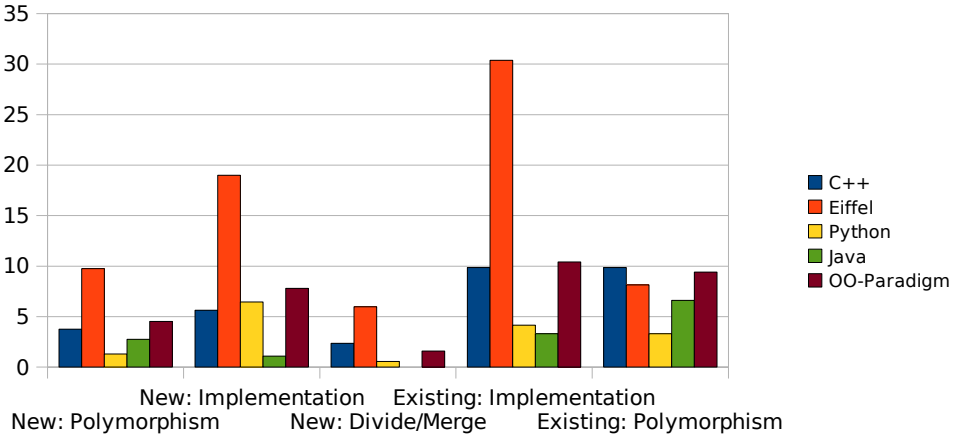

Chart 10: "classifications of alteration" [per mil of changed commits]

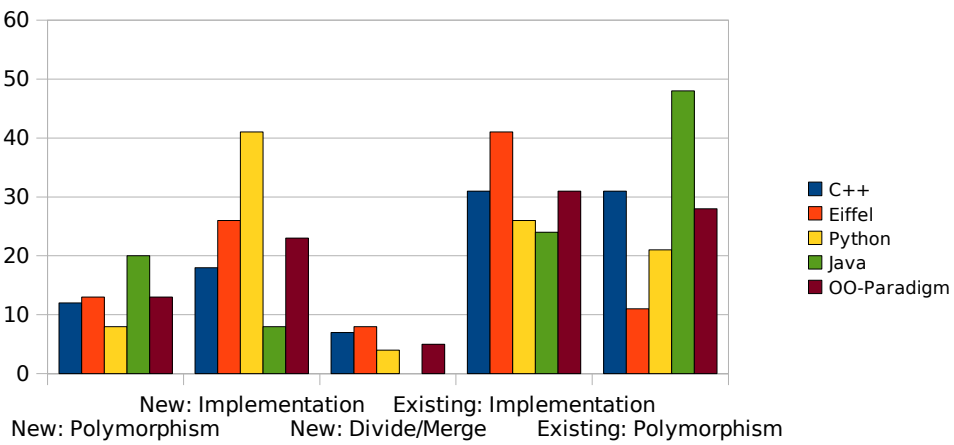

Chart 11: "classifications of alteration" [percentage of observations]

The conclusions start with a look on the distribution chart 11. We identified a footprint of the different OO-PL's, that is discriminatory and thus allows us to distinguish and pinpoint the differences among the languages.
C++ is different than the others, except Python, as it's developers rely as often on existing parents for implementation reuse as for participation in a polymorphic framework. In comparison to Python it has a significant smaller value for the introduction of new hierarchies with the goal to share code.
Eiffel's individuality can be expressed in the largest positive percentile disparity between the reuse of code for functionality and the category “existing: polymorphism”. The other categories are similar with the average of all OO-PL's expressed as OO-paradigm.
Python's maxima lies in the category of “new: implementation” and the discrepancy to the value of “new: polymorphism” is exceptionally large.
No other language than Java has a such skewed distribution towards polymorphism or such a limited usage of “implementation inheritance”.
In absolute per mil of commits the individualistic distribution shows the strengths and weaknesses of the programming languages in providing the benefits of the OO-paradigm.
It is very interesting to note, that in terms of reuse of existing polymorphic relations all programming languages, except Python, have nearly the same number of observations per 1000 commits.
As a major conclusion the OO-paradigm implies for the observed languages the same amount of development goals required to add or remove a parent from an existing class to participate in an existing framework. In all other categories Eiffel shows a significant advantage over the other OO-PLs.
While C++ developers have an non-significant advantage over the other OO-PL in the reuse of existing polymorphic relations, it can not compete with Eiffel in the other categories. C++ performs better than Java and Python in the “existing: implementation” category and better than Java in the “new: implementation” category.
Eiffel is the big time winner – except in the reuse of “existing: polymorphism” where Java and C++ share a similar value. The major difference is the observations that Eiffel's core developer can solve more tasks with the help of adapting the type relation level of existing classes to enhance the degree of reuse of code. The significant larger value for the number of commits introducing a new polymorphic interacting set of classes should also be noted.
Python's capabilities to allow reuse through inheritance are

significantly worse than those of the other languages. The language only competitive edge for maintenance is the introduction of small classes to enhance code reuse mostly done for bug fixing. This implies, that the new language constructs such as meta-programming, interpretation and dynamic types distract the developers from employing the more traditional concepts of the ancestors of the language.

Java's success in this experiment must be questioned too. The largest percentage at the “existing: polymorphism” relation does not provide an advantage, but rather the only category to meet the average of the competing OO-PL's. Possibly the lack of support for multiple inheritance seems to limit the application scope for modifications enabling enhanced reuse of existing “implementation” through the type-relation level. The result that the language can not compete with Eiffel in the introduction of new interfaces used for dynamic binding, is disappointing

The final conclusion is that for polymorphism, the main innovation of the OO-Paradigm over the classic PC-PL paradigm, the disparity among the observed languages is small. Thus no language can provide a significant improvement on the reuse of existing polymorphic relations. Only Eiffel manages to have a significant higher level of introductions of new polymorphic relations for existing classes.

Accordingly the differences among the programming languages in adaptations of existing classes on the type relation level is driven by the different level of support for “implementation inheritance”. The key result is, that for 1000 commits modifying existing classes only 15 commits adapt the ancestors with respect to polymorphism. Thus multiple inheritance structures are either used to incorporate language imposed parents or ease the usage of “implementation inheritance”.

Results: C++ projects versus projects

Introduction

In this section we take the same approach as on the inter OO-PL comparison but for the projects of the C++ programming language. Further details on the language can be found in the “languages used” section. The selection of the projects for C++ is composed of Qpid, Slicer and CodeLite and followed the assumptions described in the section “projects used”.
As we have seen in the inter OO-PL comparison C++ performs as good as the others, when it comes to the usage of polymorphism to solve the development tasks on existing classes. For the reuse of functionality on the declarative level rather than through delegation C++ saved the second place behind Eiffel.
The language C++ is designed as hybrid interleaved with the PC-PL C. It normally supports no garbage collection, while Java, Eiffel and Python incorporate this service by default. Thus some evolutions observed for existing classes provide the integration of reference counting as a kind-of garbage collection normally used from the 3rd party library “boost”.
The different C++ projects analysed are heterogeneous along the application domain, the age and the numbers of authors.

All commits versus changes to C++ code

For all three projects analysed in C++ we present the numbers of scanned commits and the percentage of those incorporating at least one line of changed code.

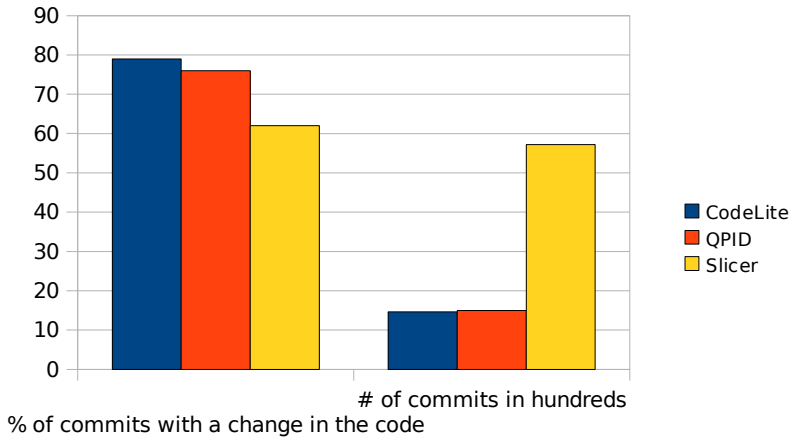

Chart 12: Changes of code [percentage of all commits]

Chart 12 shows the different number of commits analysed for the

different projects in the category of C++. As expected the variability among the projects in terms of percentage of commits with a change in C++ is small.
For CodeLite the analysis is for roughly one year of development. The extraction of data is pursued for 1.4 K revisions. Most of the commits not including code are modifications of the build configuration as Makefiles or the IDE configuration file of the project. The project has been developed by a total of three authors and over 90% of all changed lines of code are modifications of existing code.
For QPID we annotated automatically 1.5 K commits over a time horizon of two years. Most of the revisions without any changed lines of code are concerned with script language code to execute the test suite, configuration of the build system and integration of external tools as Valgrind. The focus of development is equally distributed between the addition of new files and complete removal of old files and maintenance tasks in terms of changed lines of code. The number of authors is 15.
For Slicer the extracted data consists of 5.7 K revisions. The huge percentage of commits without changes in C++ are modifications of the GUI either by changing an icon in the PNG format or by altering TCL script language programs. The focus of development lies clearly in extending the product as more than 50% of all lines of code parsed reside in new files added to the source tree. This implies a small percentage of code modifications in existing files. The total number of authors involved is 42 and the observation is for a time period of 2.5 years.
The projects can be differentiated in number of authors, in the main development direction and the motivation for commits without any changed lines of code.

Changed commits versus changed existing type-relations

As for programming languages differences exist, this holds also for the underlying projects. This subsection presents the total alterations on the type relation level of existing classes observed for the different projects. For example how many of a 1000 commits modify the existing code base and alter the existing type relation level.

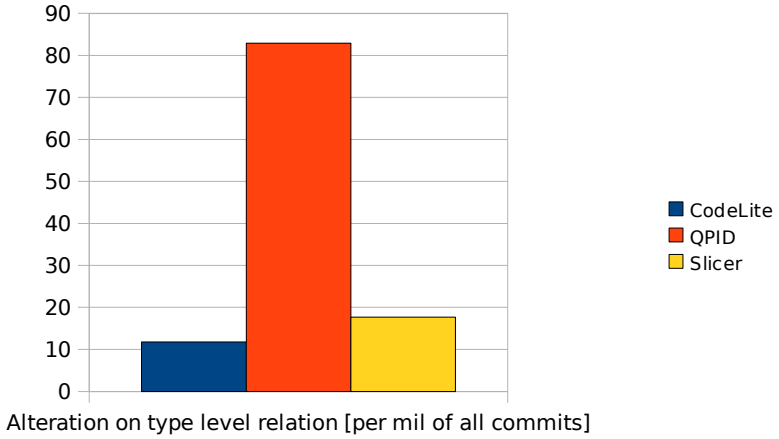

Chart 13: Changed existing type-relations [per mil of changed commits]

Chart 13 presents the individual per mil of commits with changes to the code base altering the type-relation level of existing classes in C++.
CodeLite has only ten modifications for every thousand commits for the number of revisions traced. In fact no commit has been found that includes a modification of an existing class to a class using multiple inheritance. The analysed revisions revealed two cases of metamorphosis towards single usage of inheritance and one single change of removal of parent relations to a class without inheritance usage. In order to get a more profound picture changes that involve any kind of modification are analysed.
QPID is an interesting project from the developer perspective. It has by far the highest per mil for changes to the type-relation layer of existing classes. As mentioned the project has an even distribution of development effort among new code, existing code (maintenance) and removal of obsolete classes. This seams to be the policy of the main developers to review and refactor most of the components on a regular basis.
Evidence to the existence of a code maintenance and component review is the presence of changes also on the type-level motivated solely to enhance reuse of code, to merge or divide classes. Another stronghold is the presence of integration tasks into existing other components of the software.
Slicer has 18 per mil of commits including changes on the type-relation level. This can be explained with fewer changes in terms of changed lines of code in existing source files than QPID. Moreover it seems unlikely that a similar system for quality control as in QPID is present. The manually analysed cases are the event of changes in the categorization of the classes e.g. single inheritance, multiple inheritance and no inheritance. For this only one change exists in

the revisions scanned. In the other 33 revisions with modifications on the type-level parents are added and removed but no change in the categorisation. In order to get meaningful results 29 commits are analysed in the category undefined.

CodeLite - Samples of changes

Revision 1663, requirement, large pain, existing polymorphism

The first sample is the revision 1663. The commit is rather small as only 83 lines of code are added and 65 are removed. The development goal is an enhancement of the performance and the the commit involves four source code files of whom only one is a header file.

Given a “Job” class called “RefactorIndexBuildJob”, exclusively used for the implementation of another class. Access to an abstraction layer through a parent called “Job” is added. This allows the usage of a queue data structure for jobs to manage different instances of classes implementing the “Job” interface.
The new layer for the management of the various job instances resides in an external component. The changed class participates in the service through polymorphism. Accordingly to our set of rules this justifies the classification of “large pain” for the developer to implement an alternative to achieve the same results.

Revision 1200, requirement, large pain, new polymorphism

The second sample is the revision 1200. In terms of changed lines of code the modifications consist of 251 added lines, 186 removed lines and 31 new lines. On the type-level relation layer modifications include the introduction of two new classes (“SvnPostCmdAction”, “SvnReportGeneratorAction”) and the change of one class without inheritance usage (“SvnIconRefreshHandler”) to a single inheritance class.
The development goal is to implement a new requirement. The developer implements the new functionality in a new class “SvnReportGeneratorAction”. The new class interaction with other classes is similar as the identified existing class “SvnIconRefreshHandler”. Therefore the required shared interface is factored out into a new abstract parent class for dynamic binding and polymorphism (class “SvnPostCmdAction”). As a result the existing code developed for the interaction with instances of class “SvnIconRefreshHandler” exclusively to be generalized and reused for the new class “ SvnReportGeneratorAction” as well. The reused existing procedural code composed of delegation calls to the new interface consists of around 700 lines of code.

The introduction of polymorphism for a major reuse and code sharing of two classes in the implementation of a single other class qualifies the change to be judged as “large pain”. In order to implement an alternative, one would have to add extra code for each of the two types in order to provide the same functionality

QPID – Samples of changes

Revision 568332, refactoring, small pain, new implementation

Revision 568332 is a pure maintenance and refactoring commit. It is rather small in size modifying only 200 lines of code.
The development goal is the removal of no longer required functionality. In addition the change factors out a small abstract class from a single child. This abstract class is not used externally at this point in time and therefore seems not to be forced. This justifies the classification as pure refactoring and the pain for the developer as small. As no polymorphic usage of the new interface could be detected, according to our classification rules the change introduces a new implementation relation.

Revision 577459, integration, large pain, new implementation

In revision 577459 around 300 lines of code are changed. It introduces a single new class as an upgrade of an existing one. The detected modification of an existing class on the type relation level is the evolution from a single to a multiple inheritance using class. The development goal for the additional parent is the integration into an existing hierarchy sharing functionality and data on a higher level of abstraction. Given the need to change the internal state of the parent, not disclosed through the provided interface, delegation usage seems impossible.
As a conclusion the usage of inheritance is motivated by the need to access and modify the external component and is classified as “large pain” for the developer. The newly inherited type is not used for dynamic binding of its instances or covariance on the return value. Our set of rules classifies thus the employed mechanism of alteration as “new implementation”.

Revision 594364, requirement, large pain, new polymorphism, divide and merge, new implementation

Revision 594364 is the result of a patch splitting one component “Broker” into two interacting. One expects for the division that many lines of code are removed in the component “Broker” and added to the new component “Management”. This seems to be the case as 13 hundred lines of code are deleted in the old and 16

hundred are added in new files in the new component. In the over 20 existing source files of component “Broker” only one hundred lines of code are added.
On the type-level relation layer the observed changes are ten new classes in the new component, six deleted classes in the old component and one class being modified and changed to multiple inheritance.
The adaptation of the existing class on the type relation layer integrates the newly crafted component into the existing. The class named “Queue”, with a single parent “PersistableQueue”, evolves to inherit from a new abstract class of the new component. Through this abstract class and polymorphism the “Broker” component can serve multiple types of objects through the same code and dynamic runtime casts.
The classification of the change is three-of-a-kind. The first is the new usage of the added abstract class for dynamic binding of it's children's instances to make them exchangeable at run-time. The second is the motivation of the reuse of functionality through the introduction of the coupled class “ManagementObject”. The last is the split of several existing classes into a new set of multiple and mutually exclusive classes in the new component. Given the size and the complexity, of the dynamic interaction and transformations on the type-level, the change is classified as “large pain”.

Revision 599395, refactoring, medium pain, divide/merge

Revision 599395 is motivated to fix an undesired behaviour. As a side effect refactoring is preformed on classes and similar behaviour is placed in a shared parent. The removal of the common behaviour leads to mutual exclusive children. As no polymorphic usage is involved and the identical behaviour could be achieved through pure procedural means the classification is “medium pain”.

Revision 644287, integration, large pain, existing polymorphism

Revision 644287 is related to the change of 594364. It just integrates another class to use the mechanism developed. Hence the classification is “large pain”.

Slicer – Samples of changes

Revision 105, refactoring, medium pain, existing implementation

In revision 105 around 1700 lines of code are added or removed in 19 different files including 9 header files. Two new classes are added and integrated into the component. As a side effect and not as primary goal the developer exchanged the parent of one existing

class. As a classic refactoring the former parent becomes an attribute and is used accordingly. Because the removed parent inherits from the base class of the framework and this functionality is still required, the new parent is the base class itself.
Given the none necessity of performing the modification, it is classified as a refactoring and the pain for the developer as medium. This can be questioned as to why it is not a “small pain” change. It is because the provided specialisation by the replaced parent is not required and is therefore misleading in the understanding of the class.

Revision 931, requirement, large pain, existing implementation

The revision 931 is a small change of one class located in two files. It is the opposite of the change of revision 105. The abstract common parent class is replaced by specialised child providing required functionality. In this case it allows to access the scripting interface and to execute and interact with the TCL scripts mentioned already. The change is categorised as “requirement” or “feature” driven. In consideration of the present and tested class providing the desired functionality to already more than ten children, the pain for a developer to reimplement the functionality locally is large.

Revision 4136, integration, large pain, existing polymorphism

The major change contained in revision 4136 is composed of 14 modified files including four header files distributed over four interacting components. Thus it comes as no surprise that the adaptation performed at the type-level relation layer is part of integration work. In fact a complex piece of mathematical procedural code, implementing complex algorithms, is embedded in to a new framework. It provides new and different features for delegation and polymorphism through a new partly abstract parent class. In order to migrate from the old to the new framework the existing effective classes have to adapt to the new and remove the old.

CodeLite – Samples of changes

Revision 1163, requirement, small pain, new implementation

In commit 1163 around 150 lines of code are added or removed. The impact is the addition of a new class and the removal of another. The detected “hot spot”,or the alteration on the type relation level of an existing class, performs the replacement of the single parent of one widget class. CodeLite's GUI design is based on

the 3rd party WxWidget[14] component. It consists of a cross-platform GUI and a library of supporting tools.
To interact with the external component CodeLite has a tiny set of own classes implementing the required interfaces. The other classes of CodeLite interact or inherit only with or from this set of own classes. In this change an existing class, directly inheriting from the external component, is replaced by a new class. The new class becomes its parent. Accordingly the developer exchanges the name of the class and reflects the modification in the single child. The classification has to respect that fact, that polymorphism is used to render and run the GUI component. But the polymorphic relation existed already before the modification and continues to exist (through the new parent). Thus it has no impact on the classification on the pain scale nor on the modification scale. As a result the alteration is classified as implementation inheritance, motivated by a requirement and “small pain”.

Revision 2277, refactoring, small pain, existing implementation

Commit 2277 includes a refactoring. The modification leading to the alteration on the set of ancestors of an existing class consists of two parts. As a first step it changes the name of one class, that is used for implementation inheritance. The second step removes a not required interface and functionality by the removal of a parent.
As the motivation for the change is not a removal of faulty code, the change is categorised as refactoring, modification of an implementation inheritance relation and “small pain”.

Revision 2311, requirement, small pain, new implementation

In revision 2311 a common sub case, of the change of revision 1163, is persisted. For a class originally inheriting and implementing a wxWidget interface directly, a new auto-generated base class is added to the repository. The base class contains all the configuration code for the interaction with the external wxWidget component as defined in the IDE creation tool. Accordingly the base class is then extended for the interaction with the CodeLite components. The modification contains thus a replacement of the parent from the widget to the new stub base class, which inherits from the wxWidget.
As in revision 1163 the external GUI component depends on a set of common interfaces for all instances to render or react to the user interaction. This depends on polymorphic calls on the run-time objects representing parts of the widget or events. Nevertheless the change can not be tagged as introduction of a new or existing polymorphic relation, because the involved classes have not

changed their participation in this mechanism.
The alteration is motivated by not present, but required functionality provided in the stub classes. This can be for example the connections to sub-widgets and registration of observers. Given the fact that some of this functionality is already present, in the hand-crafted old version of the class, the pain for the developer, to implement the new functionality by himself, is small.

Changed existing type-relations versus "level of pain"

In this subsection the focus lies on understanding whether the developers of the different projects perform the same tasks on the classification scale of “levels of pain”.
Let me remind the reader that the aggregation onto the level of OO-PL showed a high degree of homogeneity in the percentage distribution.

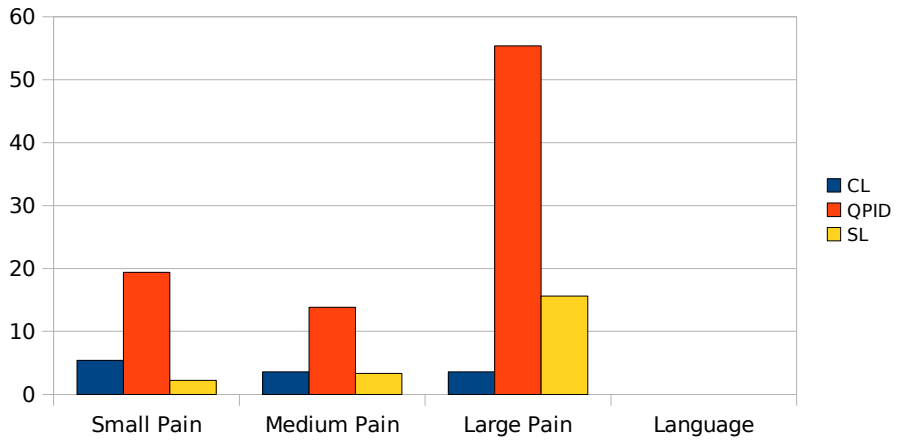

Chart 14: "level of pain" [per mil of changed commits]

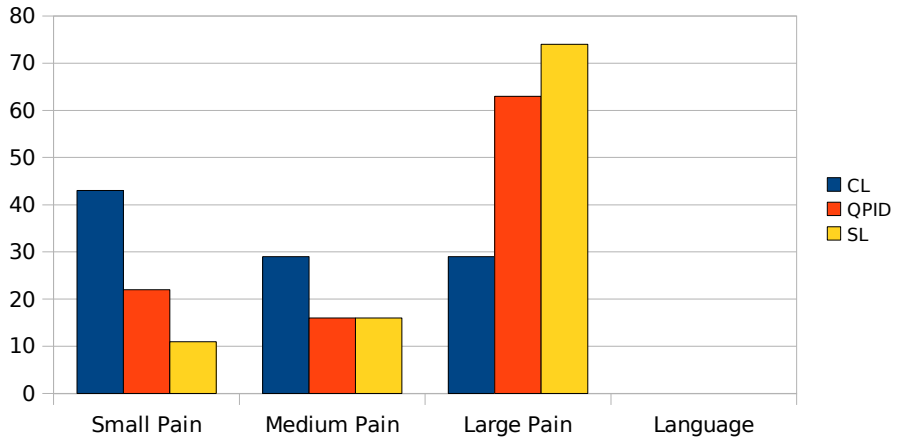

Chart 15: "level of pain" [percentage of observations]

Chart 14 shows the different degrees of pain in per mil per project

and add up to the values presented in chart 13. This allows direct comparison among the different projects and languages. Keep in mind that we see the averages of the observations not more and not less.
Let's focus on some insights for the numbers of CodeLite. For this project most of the changes observed are related to the functionality of the GUI. As mentioned in the “samples” it depends on an external component. Most of the reviewed modification's development goal is the implementation of a new functionality. It requires quite often a simple exchange of a base widget type provided by the GUI library, for example “button” to “button with logo”. The “large pain” version of this modification consists of a specialisation by replacing the former parent by one of it's children. For example a general event-handler class by it's specialised child “panel”. The “small pain” scenario includes refactorings altering the name of a class and the adaptation of it's children.
For QPID 66% of the observations in the category of “large pain” are integration tasks, concerned with the interaction of multiple components. We could learn, that more than 50% of those observations contain classes with already an another parent. Accordingly this are examples for the introduction of multiple inheritance for existing classes.
The project consists of two applications and some services. The two core applications are the broker and the client. The services as the management interfaces, configuration interfaces and persistence support are interconnected and responsible for 7 out of 12 observations of the “large pain” category.
In the “small pain” segment, the reviewed “hot spots” include small but not necessary changes. For example the developer decides to add a small set of tiny features through inheritance rather than through re-implementation. An also popular modification is to introduce an abstract parent class for two children sharing commonalities without the need of polymorphic usage. Another typical scenario is the reorganization of a code-sharing hierarchy of a couple of small mutual exclusive classes. A possible reason is the addition of a newly differentiated class. In the medium category the modifications are dominated by merges and splits of classes with a complex code sharing in terms of a refactoring on the parent level. Slicer has the largest per mil of the observations belonging to the “large pain” category. In contrary to QPID it's “medium pain” per mil is larger than the “small pain” per mil. Two of the four observations of the “medium pain” category introduce new parents in order to factor out complex commonalities between mathematical objects and the interaction with the rest of the system. The other

observations replace existing parents as part of refactorings to reduce similar code in mutual classes.
7 out of the 13 code reviews labelled as “large pain” are required to provide an enhancement in functionality. None of them introduces a new class and all are concerned with the GUI of the application. The GUI is merely a tree of specialized widgets sharing code and providing an unified interface. Hence most of the changes detected are replacements, of the single direct parent, by another class in this tree. As this is replacements are the common case, we could detect only one observation, where the actual number of parents changed.
As a conclusion this analysis provides us with a fast understanding of the degree of heterogeneity among the projects. It also implies that our selection criteria for the projects in order to reach comparability on the language level is truly important.

Changed existing type-relations versus "development goal"

The projects come from different domains of applications and have thus different designs. Some have fewer components, some have more. From the language level comparison we could identify that most of the changes are either in the requirement or in the integration category. This is also true for all the projects.

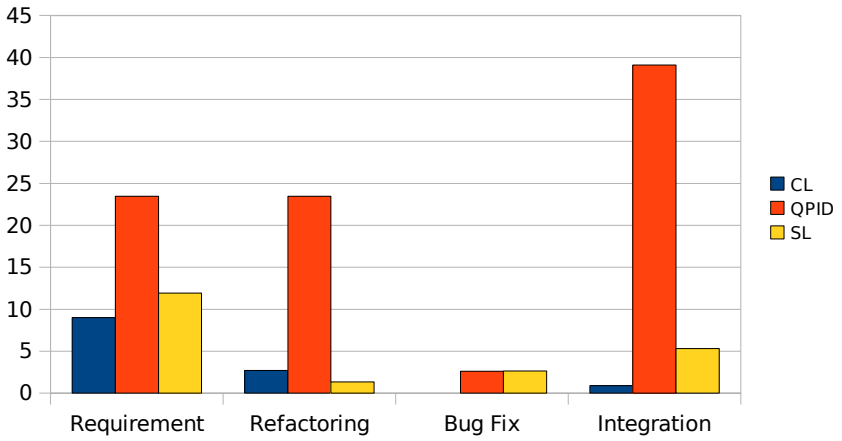

Chart 16: "development goal" [per mil of changed commits]

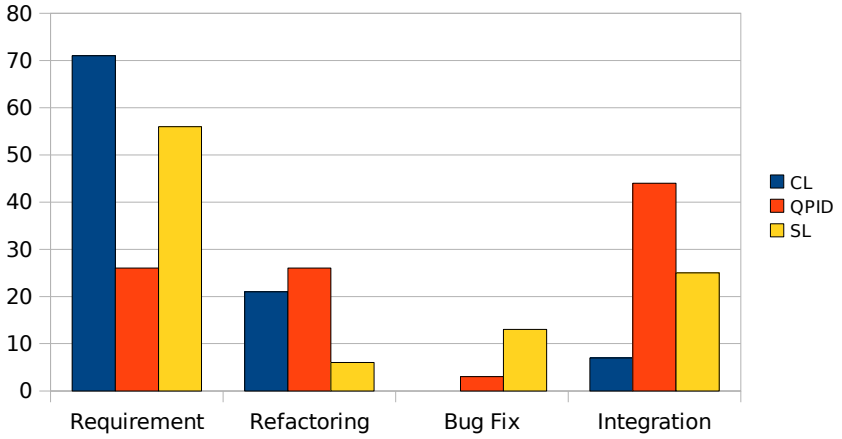

Chart 17: "development goal" [percentage of observations]

Chart 16 presents the categorisation for the different projects in per mil of commits changing the code. Therefore the bars add up to the total pre mil of changes presented in chart 13 and is directly comparable with all per mil charts in this report.
CodeLite evolves around functionality provided by and through the GUI. Most of the “requirement” driven tasks therefore represent a new feature for the user. Integration is smaller than requirement as the GUI tool kit is hidden behind a curtain of existing classes of the component.
QPids's requirements are accounted for the largest per mil of all the projects. Interestingly in terms of bug fixes QPID is overtaken by Slicer. An explanation for the high value at integration is the small set of core components in combination with the many enhancement integrated from small external components. For example the management interface or configuration interface are required to be connected with many of the core classes individually. This can not be performed at a higher level of abstraction. In 8 out of 11 cases the chosen solution, for the implementation of the desired enhancement, requires the inclusion of a new parent. Though, it seems only required in four of the observations. An affinity for major reorganizations of the hierarchies, in order to add functionality at a higher level of abstraction, can be found in all those reviewed cases.
Slicer's largest stake is, as for the others, the implementation of requirements. In contrary to Qpid a very active refactoring praxis, of existing classes on the type relation level, can not be detected. The surprising high value in the category of “bug fix” is a result of only two observations. Both require a change on the type-relation level by adding an existing but missing parent. The missing parents are interfaces for the interaction with external components. The employ

dynamic binding and thus no easy alternative exists. The way the GUI is designed, as already mentioned, supports many enhancement by exchanging the direct parent. This is responsible for the changes under the category of “requirement”.

Changed existing type-relations versus "classification of alteration"

The final subsection of the results section on the C++ projects presents a comparison of the “Categories for “classifications of alteration””.

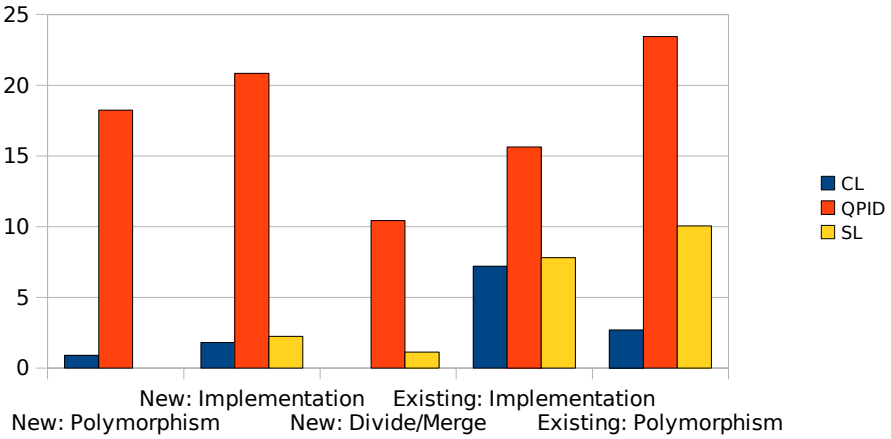

Chart 18: "classifications of alteration" [per mil of changed commits]

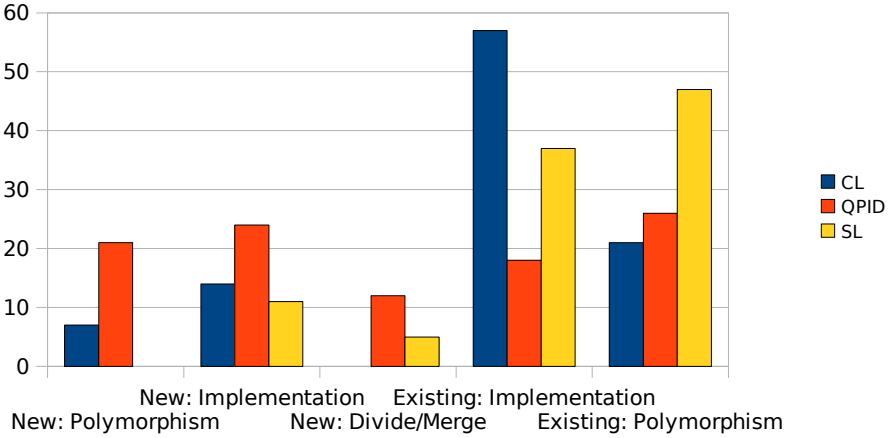

Chart 19: "classifications of alteration" [percentage of observations]

The chart 18 provides the average distribution in per mil of the different kinds of modifications distinguished.
CodeLite changes, as mentioned already, have a very high level of homogeneity, because most occurrences is in present inheritance relations. The few new polymorphic relations added might be misleading in the sense, that not truly a new framework is

integrated or a new dynamic functionality added. This are cases, where a new aspect of an already used GUI library is required to implement a new feature and is made accessible through a new, possibly generated, stub class.
The reader can see that the Qpid dominates. One reason is the high degree of polymorphism involved. No other project introduces so many new polymorphic relations to existing classes. This can imply an overall higher level of maintenance at the type-relation level of existing classes, not only in comparison to the other projects in C++ but in general. An easy equation seems to be, the more polymorphic relations required to access services, the more are adaptations likely, when a change happens. Hence a good policy of regular refactorings and introduction of polymorphic usage by isolation of services into own distinct components, can lead to an overall higher per mil of tasks solved on the type-relation level. Slicer distribution has more than 80% of all alterations in already existing hierarchies. Not a single observation exists, where a completely new polymorphic relation is added to existing code. In contrary to Qpid not a single refactoring has been detected, splitting a component or adding a new polymorphic relation. The realised refactorings, by the development team, deal with straight forward “implementation sharing” hierarchies. They do not reach over the border of the component and are limited to only half a dozen of classes. The dominance of alterations on GUI widgets, an existing hierarchy, bind to other components through interfaces and polymorphic usage, explains it. How would the distribution look like, when a few more strongly interacting component using polymorphic types would exist?

Results: Eiffel projects versus projects

Introduction

The organisation of this chapter is to the largest part identical with the OO-PL comparison. The only difference is the focus on the projects of the Eiffel programming language.
Further details on the language can be found in the “languages used” section. The selection of the projects for Eiffel consisting of EiffelStudio (ES), Goanna (GA) and Gobo (GO) followed the assumption described in the section “projects used”.
From the comparison among the different OO-PL's Eiffel is the clear winner in enabling “implementation inheritance”. Major advantages in the number of solutions implemented by the developers based on polymorphism could not be detected.
The Eiffel programming language is designed to provide the maximum support for the OO-paradigm's approach. Multiple inheritance and covariance are supported and provide additional freedom to the developer. Further details can be found in the “languages used” section.
The different Eiffel projects analysed are heterogeneous along the application domain, the age and the numbers of authors.

All commits versus changes to Eiffel code

For all three projects analysed in Eiffel we present the numbers of scanned commits and the percentage of those incorporating at least one line of changed code.

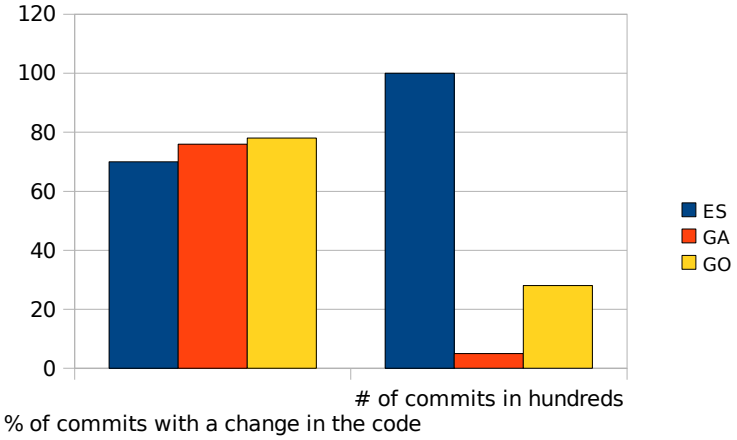

Chart 20: Changes of code [percentage of all commits]

Chart 20 shows the number of commits in hundreds for the different projects analysed in Eiffel. The actual number for EiffelStudio is 69K

of revisions. This would distract the chart and thus is capped at 100. As already seen at the language level the projects are very similar in the degree of commits without actual change on the code basis. Still for the interested reader some insight is provided.
EiffelStudio is an integrated IDE with a compiler included. For some of the Eiffel base classes dual C classes are maintained and extended. A large part of the none Eiffel changes hence is busy with the underlying C code base. Some of the commits are concerned with the build, configuration and shell scripts.
Goanna has a higher percentage of actual Eiffel commits than EiffelStudio. The non-programming language changes result from the documentation, build configuration and Microsoft BAT files.
Gobo has the highest percentage and the non-code commits can be explained as well, as mostly concerned with the configuration and the documentation.

Changed commits versus changed existing type-relations

As for programming languages, differences exist in the total alterations on the type relation level of existing classes observed for the different projects. We present, how many of a 1000 commits modify the existing code base and alter the existing type relation level.

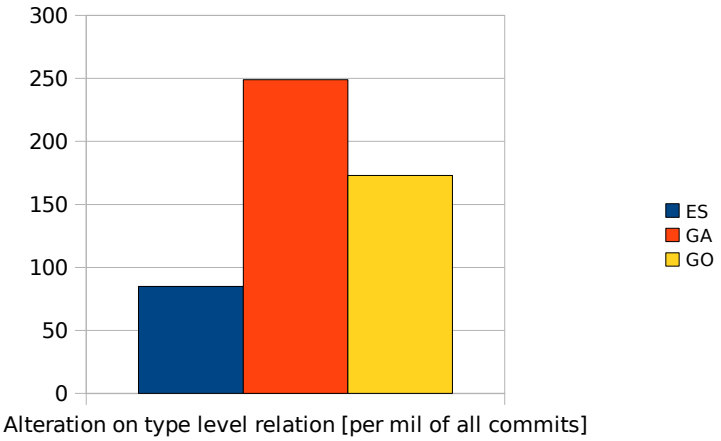

Chart 21: Changed existing type-relations [per mil of changed commits]

Chart 21 presents the number of alterations on the type-level of the existing classes in per mil.
EiffelStudio has the smallest value of the three projects. This is further surprising, as the project allocated 38% of all changed lines of code to existing class files, the highest value of all. One reason might be the larger size of EiffelStudio and the more independent components not leading to the same amount of integration work for the developers than in the smaller and younger projects.

Goanna has by far the largest value. The team consists of 13 authors. As it is presented later, it exposes the developers more than three times as often to a “large pain” task than EiffelStudio. The difference to Gobo lies in the number of integration and bug fixing tasks solved. This comes as no surprise, as the percentage of changed lines of code in new files is twice the number of the other two projects. It implies adaptation tasks of existing classes to include the benefits of the evolving new frameworks.
Gobo has a similar effort distribution as EiffelStudio. The difference in the per mil value is, in this observation, the number of requirements tasks solved, that require to alter existing classes on the type-relation level.

EiffelStudio – Samples of Changes

Revision 1201, requirement, language imposed

Revision 1201 is an example for the Eiffel standard way to access and modify shared information across the system. Instead of applying the singleton design pattern, Eiffel supports it on the language level by so called once routines. A once declared routine behaves as a global singleton accessible through the class. The value of the constant is calculated at run-time, when the first usage occurs and is stored afterwards. Instead of normal constant, the once routines share it's result with all instances of the class. Hence the normal way to access globally shared data is through inheritance from the class containing the shared globally unique reference. The alternative to implement a singleton in the classical way is impossible, due to the non existence of static features. The alternative to declare a member of the so called shared class is not impossible but not encouraged, because many of the once routines visibility is “protected” and thus not accessible through the interface.
This are reasons to categorise this kind of modification in a special “language imposed” category to enhance comparability among the different programming languages. Thus all such detected revisions are removed from the comparison on the language level.
Nevertheless this commit is concerned with the compiler. A class representing a feature in the Eiffel byte code requires, in order to implement a new feature, access to the two shared classes. The first shared resource is a global table. It contains for each element of the AST of the compiled project information about the state. It is used to check whether a translation already exists. The second class is a shared server, responsible to store all the translated byte code for the compiled project. It is used to load the required byte

code of the children in the AST of the feature such as the body. The commit is tagged as requirement implementation, “language imposed” motivated and has no pain label. As the class already had one parent and two others are added, this is a standard example of an existing Eiffel class metamorphosis to use multiple inheritance.

Revision 20703, refactoring, large pain, divide/merge

Revision 20703 is another example of a class growth from a single to a multiple inheritance user. In this specific case two classes of the same component, of a portable GUI library, are merged into a new one. It unifies the functionalities of the two. The overall design of portability is a bridge pattern.[15] Both classes belong to the implementation side and are coupled with their abstract interface. Through the join a new interface is required to replace the mutual two. The development path chosen at this point of time is not overly successful. At a later point in time one of the parent becomes the other parents predecessor. This renders the join of this revision useless and requires the removal of the class.
Given the complexity of the bridge design pattern and the usage of polymorphism at the inter-component level, this change has no alternative, than the introduction of inheritance to gain access to all features. The commit is categorised as refactoring, because it is a new layer of abstraction and thus not a necessity. The development of an alternative requires to leave the design pattern and is hence not feasible.

Revision 12782, integration, existing polymorphism

The commit persisted with revision 12782 is an example of the alteration of a non-inheritance to a single inheritance using class. It modifies the same portable GUI tool kit as commit 20703 just one year earlier. The change in behaviour adds classes to the abstraction side of the bridge pattern. The abstraction interface contains a reference to the abstract implementation class. Thus in order to participate in the tool kit the abstraction base class becomes an ancestor. The class adapts the reference to the implementation to the according child of the base implementation class.
There is no reasonable alternative to the developers solution enabling the integration into the GUI component. Accordingly the alteration is tagged as “large pain” and “integration”.

Revision 63089, requirement, large pain, existing implementation

Revision 63089 is the first example, described in this thesis, presenting the observed case of a former multiple inheritance

supporting class reduction to a single inheritance class through removal of parents. In the process of the replacement of an existing component by a new version, an obsolete mock class, providing a common way to debug run-time information, is replaced by a new, more sophisticated and renamed version. Since some classes do no longer depend on this service, the debugging class is just removed and not replaced, turning some into single inheritance classes.
The alteration is classified as “requirement” driven and as “large pain” for the developer as she has little choice than to migrate to the new version including the new debugging class because of compliance.

Goanna – Sample of Changes

Revision 119, requirement, language imposed

The commit 119 is an example of a shift from a single to a multiple-inheritance employing class. It is once again presenting the case of the Eiffel idiosyncrasies singleton pattern at the language level. The component of Goanna controls all and possibly multiple HTTP sessions in parallel in one manager, a singleton. This revision introduces a new listener mechanism enabling other components to subscribe to all events of all sessions such as creation and expiration. A specific session, represented through a session object, requires to call up the manager object in the presence of an event. Then the manager notifies all listeners.
Because listeners can not sign up for individual session, it seems reasonable to centralise the functionality. The session class thus has to inherit from the shared manager class. The change is labelled as “language” and “requirement” driven.

Revision 345, requirement, language imposed

The commit 345 is similar to 119 as it introduces yet another usage of the singleton pattern. Goanna processes servlet requests and produces the according response. This persisted development effort improves the multi threaded processing of requests. Producers add the requests into a queue and multiple worker thread consume the tasks. The queue becomes, in this design enhancement, a member of a shared Eiffel class. As a singleton it is protected by a thus global mutex. Accordingly, in the standard Eiffel way, the thread class has to inherit from the shared queue in order to access it.

Revision 236, refactoring, small pain, existing implementation

In commit 236 the change on the type-level relation is the removal of a parent. The component is responsible to support XML RPC and

the changed class provides marshalling of scalar values. A redesign replaces the usage of the external DOM parser by a component internal solution. This leads to the removal of the external parent. The change is classified as a refactoring and the pain level is set as small.

Gobo – Samples of Change

Revision 3982, requirement, language imposed

The modification in revision 3982 is part of the XML Xpath library of Gobo. It alters 40 classes representing Xpath expressions from arithmetic expressions to variable references. The reason for this major corrective act is the elimination of faulty code and missing functionality.
The alteration on the type-level of an existing Xpath expression class enables a portable way to access the standard files such as input, output and error. This is an issue, as Gobo supports two dialects of the language: The Eiffel Language Kernel Standard (ELKS) and SmartEiffel. They diverge on the commands required to access and use the mentioned files. Thus Gobo has it's own set of classes to hide this artefacts by providing an ELKS conforming wrapper and a SmartEiffel call through façade. Long story short, in order to use the standard error file to log exceptions, the given class has to inherit from the Gobo wrapper class. This class is a singleton representing the standard files or as in Eiffel terms a shared class.

Revision 4855, requirement, large pain, existing implementation, divide/merge

The persisted change with the revision number 4855 is composed of an XML component with the subcomponents XSLT and Xpath. Half a dozen existing classes require a change on the type-relation level. As in the previous discussed commit, portability remains an issue and is responsible for new accesses to the shared classes. In order to share functionality among the children of the same base class, some become sub-children and new intermediate layers are created to enhance the code reuse and sharing effect.

Revision 2285, requirement, small pain,new implementation

Commit 2285 implements a new functionality. For an existing class two out of three parents are removed and added to the remaining parent. This changes the former child in to using single inheritance only. The sole parent grows to the usage of multiple inheritance instead.

Changed existing type-relations versus "level of pain"

The focus of the subsection lies on understanding, whether the developers of the different projects perform the same tasks on the classification scale of “levels of pain”.
Let me remind the reader that the aggregation onto the level of OO-PL showed a high degree of homogeneity in the percentage distribution.

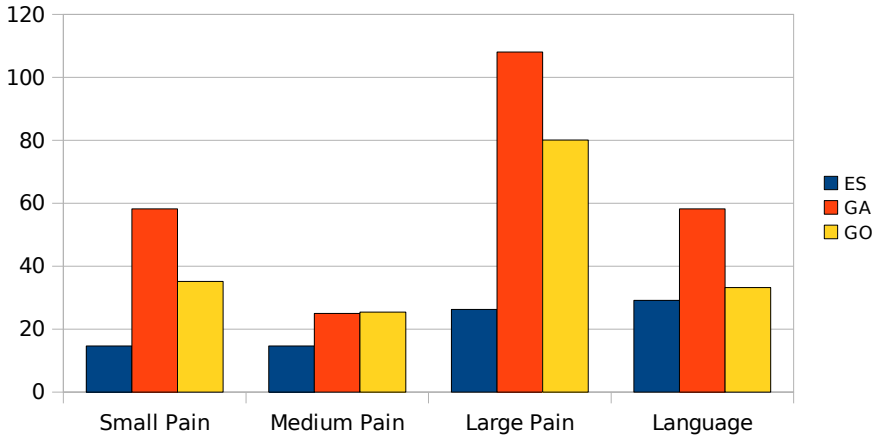

Chart 22: "level of pain" [per mil of changed commits]

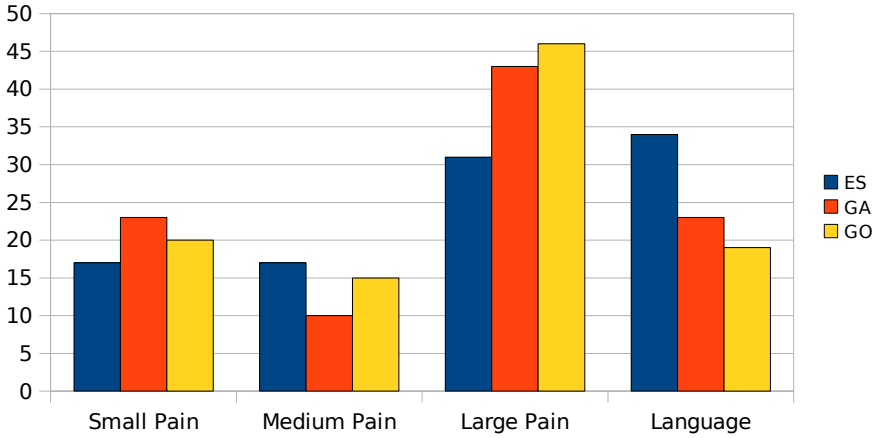

Chart 23: "level of pain" [percentage of observations]

Chart 22 shows the classification of the analysed commits in terms of how much development effort an alternative requires. The values for each project add up to the total per mil commits with alterations on the type-relation level of existing classes in chart 21.
It is important to note and the place to explain the specifics of Eiffel. The present category of “language” includes changes that are not present in this form in the other OO-PL analysed. Eiffel introduced the concept of shared classes, a language level construct of the singleton pattern. Further details can be found in the “language

used” section and in the explanation of the code samples tagged with “language imposed”.
EiffelStudio has in each category the smallest value of all projects. The “medium and small pain” category together add up to a slightly larger number than the “large pain” segment
Gobo contains double as many alterations. The reason can be identified and is the double time presence of “small pain” changes and “large pain” changes.
Goanna has it's peak in the large pain category. The distribution shape is similar to Gobo's but scaled. This can also be identified in the percentage chart. 8 out of 13 observation in the “large pain” category are integration tasks. This is a higher percentage than observed for EiffelStudio (8 out of 18) and Gobo (4 out of 17).
The highest value in the “small pain” category, also known as changes with an easy to develop alternative solution not imposing a change on the type-relation level, has Goanna. It is dominated by implementations of desired functionalities.
More than 80% of Gobo's “large pain” category are changes made in order to implement a requirement, an integration task or perform a refactoring. This differentiates to the other projects. They are mostly concerned with integration tasks and bug fixes (Goanna) or integration tasks and refactorings (EiffelStudio). For Gobo most of the detected changes are classes that inherit from an existing parent in order to reuse implementation or to offer a new interface for polymorphic usage. The underlying need are portability issues and thus justify the “large pain” label. The normal scenario is contrary to the portability issue explained in the sample of revision 3982. The parents are not singletons but pure functional classes without any members, such as portable string routines or integer conversion routines. For example some of the “large pain” changes are hierarchies of constants, required to construct the name spaces for Xpath in Eiffel. The implementation of requirements and refactorings dominate the small pain category as well.

Changed existing type-relations versus "development goal"

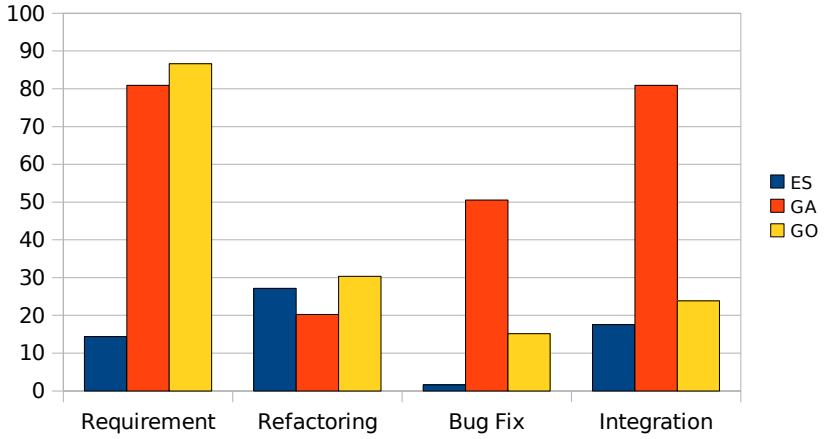

Chart 24: "development goal" [per mil of changed commits]

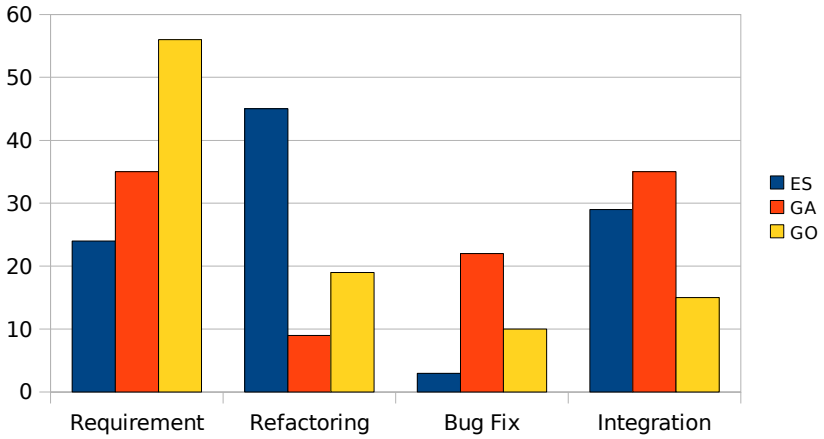

Chart 25: "development goal" [percentage of observations]

Chart 15 shows the per mil changes observed for each underlying development goal. The bars add up to the value presented in chart 14 with language imposed changes removed. Therefore the chart remains comparable with the other programming languages.

EiffelStudio is interesting in the sense that it has a very small value for requirements and a maximum for refactorings. This could be the case because core developers responsible for this changes are more or less reviewing the code base for conformance and cleaning the code. The very small values requires further inside. For EiffelStudio a total of 8 commits including a change qualified as a bug fix are analysed but 7 include manipulation of singletons. Thus only one observation exists with a bug fix for EiffelStudio. In contrary for Goanna out of 8 analysed commits only 3 involve singletons leaving 5 accounted for in the upper chart. The other two project add utility parents providing additional functionality such as string

manipulation and portability for certain operations used. EiffelStudio is not concerned with portability so consequently does not provide special portable enhancements for data type manipulation. Making EiffelStudio portable, could thus imply a higher value for the bug fixes category but also for the requirement category.
Goanna has the maximum values for bug fixes and integration. It is weaker on the refactoring scale although the difference to the other projects is not very large. It can be seen that more than 70% of all observed changes are motivated by either integration work on the inter-component level or by implementation of requirements. On the other end less than 10% are used for refactorings. All of the integration tasks performed belong to the large pain category and hence lack an alternative. Most of the integration tasks are actually not adding a new component but rather replace an existing such as a new logger throughout all components. The other integration tasks are concerned with reorganizations, because of upgrades or new interfaces of interacting other components. Please note that such changes have a limited impact on the small subset of components and therefore lead to fewer required adaptations.
Gobo's distribution of tasks is totally dominated with nearly 80% of changes by the requirement category. As seen in the explanation of the previous chart half of all “large pain” task (as well 80%) are requirements implementation concerned with implementation reuse of portable routines or files.
The other half of the requirements or the 39 per mil represent cases are evenly distributed between middle and small pain tasks. Nearly all medium pain tasks implementing a requirement introduce or reuse a simple code sharing hierarchy or perform a minor design change without external coupling and are developed by only two authors. This might imply personal preference to perform this way of code sharing and reuse through inheritance rather than through client relations.

Changed existing type-relations versus "classification of alteration"

The application domain of the project differ and the solutions implemented have different designs. Some have fewer components, some have more. From the language level comparison we could identify that the large overall values for Eiffel can be explained as mostly “implementation inheritance”. Eiffel also showed a significant higher value for the category “new polymorphism”.

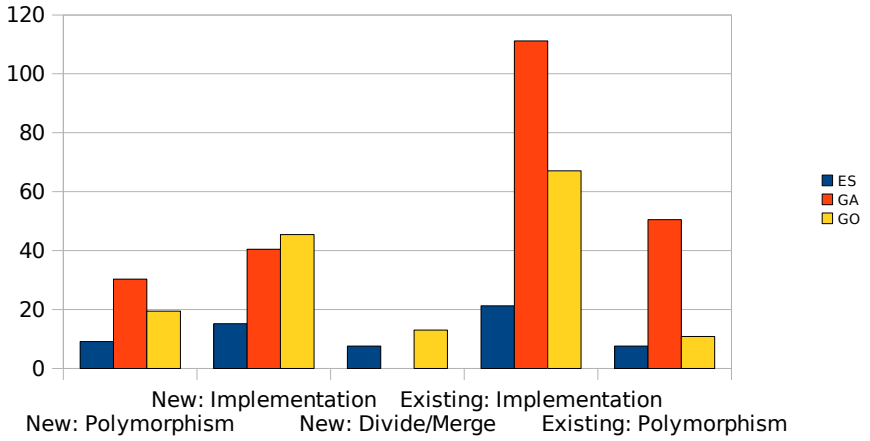

Chart 26: "classifications of alteration" [per mil of changed commits]

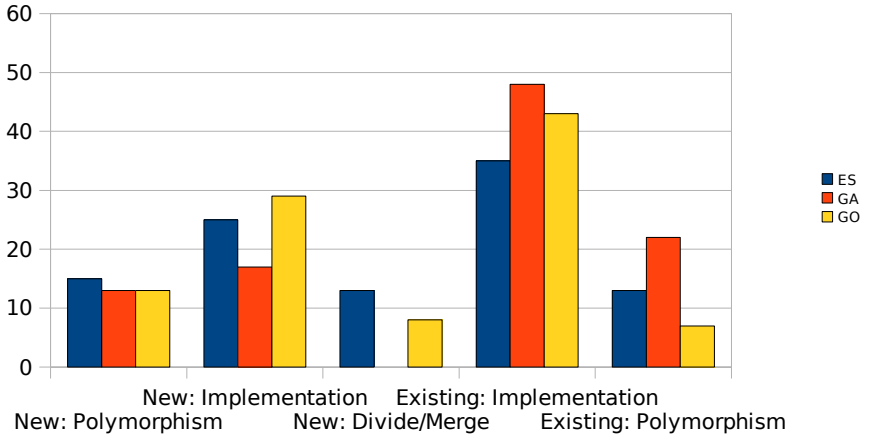

Chart 27: "classifications of alteration" [percentage of observations]

Chart 26 presents the per mil observations in the different categories of changes on the type-relation-level. All the bars of a project add up to the values of chart 21 with language imposed changes removed. The shape of the distributions in the chart 27 looks very similar for all projects. The affinity for “implementation inheritance” for all is as expected large.
EiffelStudio takes the last place in each category in the per mil chart except for “New: Divide or Merge”. 60% of the changes fall into either introducing or reusing implementation inheritance. If the “New: Divide or Merge” is added as well, it accounts for over 70% of the changes. This is a normal value when compared to the other projects.
The distribution of Goanna and Gobo differ significantly in the section of “new: implementation” and “existing: polymorphism”. The high level for Goanna in comparison to the other two projects can be identified as better reuse of existing relations.
As a final conclusion one can identify the fact, that Goanna exploits

the advantages of the OO-Paradigm more than the others, because it has a significant higher level for the “new: polymorphism” and “existing: polymorphism” category.

Results: Python projects versus projects

Introduction

For the Python programming language this subsection is nearly identically organised. Of course the focus lies on the projects. Further details on the language can be found in the “languages used” section. The selection of the projects for Python is composed of Deluge (DL), Twisted (TW) and Zope (ZP) and followed the assumption described in the section “projects used”.
As we have seen in the inter OO-PL comparison Python performs weaker as the others, when it comes to the usage of polymorphism to solve the development tasks on existing classes. For the reuse of functionality on the declarative level rather than through delegation Python performed average or weaker.
Python is an interpreted language and has a weak type system. It supports multiple inheritance and also covariance. The language has other techniques as well, that compete with “implementation inheritance” and also with the need for polymorphism. More details on the language can be found in the section “languages used”.
The different Python projects analysed are heterogeneous along the application domain, the age and the numbers of authors.

All commits versus changes to Python code

For all three projects analysed in Python we present the numbers of scanned commits and the percentage of those incorporating at least one line of changed code.

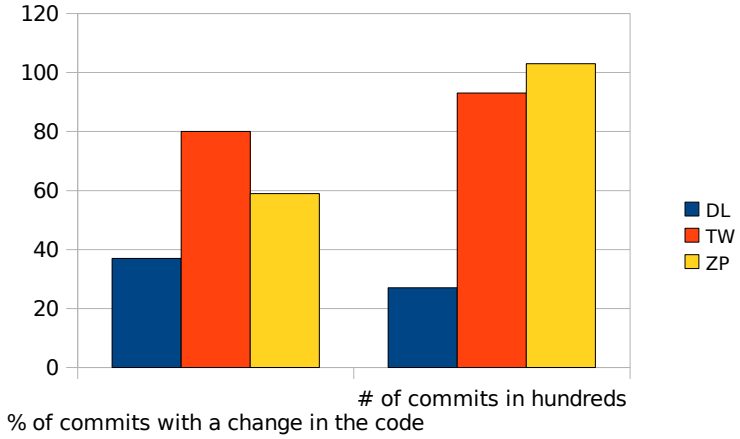

Chart 28: Changes of code [percentage of all commits]

Chart 28 presents for projects the percentage of commits that contain some Python code. The second graph shows the number of commits scanned for information in hundreds.

Deluge is a BitTorrent client. It is composed of a C++ BitTorrent library, the XML based GUI tool kitt Glade, an HTML based web-ui. Python is involved as an integration and implementation language. The non-Python commits contain hence often C++ code, XML configuration changes and alterations to the HTML based documentation and web user interface. We scanned data of two years. In comparison to other projects the size of the developer team is small with only eight. A single developer is accounted for more than 80% of all Python code lines added, deleted or changed. The same person or 13% of all authors hence coded most of the desired adaptations on the type-relation level of existing Python classes.
Twisted is an event driven networking engine written foremost in Python. This is reflected in the large percentage of commits actually containing Python code. The missing twenty percent of commits adapt the documentation in XHTML, modify text files or some configuration files of the various 3rd party GUI tool kit such as Glade or wxWindows. The software project is traced for seven years with 9321 commits automatically analysed. The project has a large developer community with 47 authors. 80% of all the changed lines of code is performed by 12 authors or 25% of the community. A subgroup of 7 or 15% of the authors performed at least 80% of all adaptations and modifications to the existing classes on the the type-releation level.
Zope is an object-oriented application server developed in Python. The system could extract data from 1996 to 2008 in a total of 10316 revisions. The project is developed by a total of 96 authors. 13 authors (13%) are responsible for more than 80% of the changed lines of code and the same number performed 80% of all maintenance tasks on the type-hierarchy level. The non Python commits contain updates and changes to the framework documentation, the development documentation such as README and some legacy code in other programming languages as C.

Changed commits versus changed existing type-relations

For the projects differences exist in the total alterations on the type relation level of existing classes. We present, how many of a 1000 commits modify the existing code base and alter the existing type relation level.

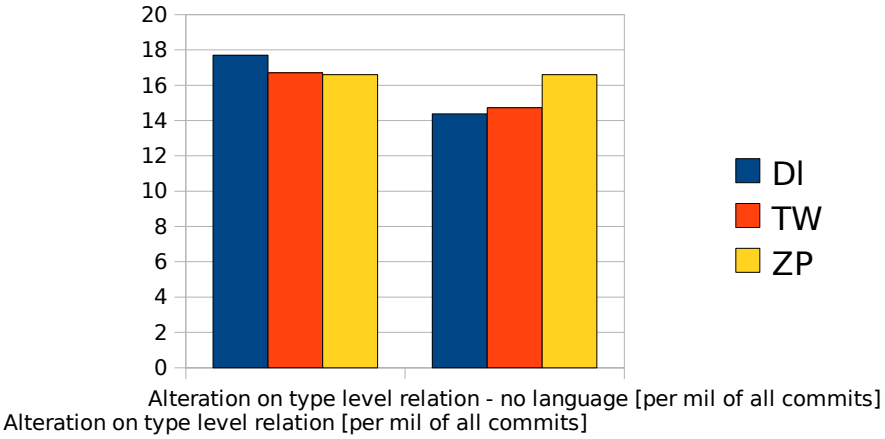

Chart 29: Changed existing type-relations [per mil of changed commits]

Chart 29 shows the per mil commits that contain a change on the type-relation level of existing classes. The values are thus comparable with all other charts of the other programming language and projects.
On one hand Python's projects show a very similar level of interaction with existing type-relations in total per mil. On the other hand the picture becomes more interesting when language imposed inheritance changes are removed. For Deluge the reduction is by 22% but for ZP is 0% and it turns the ranking upside down. The reasons for this behaviour can not be explained completely. One can provide insight on how the Python language strongly suggests usage of inheritance rather than a different mechanism by providing examples. It is on the other hand rather difficult the reason about the absence of this cases in Zope.
Nevertheless let the reader be reminded that most of the observations of Deluge are developed by one single person in comparison to a group of 13 in the case of Zope.

Deluge – Samples of Code

Revision 1891, integration, language imposed

The commit 1891 is an example of a language imposed change. As the reader might know Python has a significant different class model than the other programming languages presented in this paper, in fact it has multiple. The type of an instance of a standard user implemented class X in Python before version 3.0 is not X but the class “instance”. The type of class X is not X but classobj. Unifying types and classes nevertheless is supported too and the removal of the classical class model one of the major development goals for the language version 3.0. In the meanwhile, since version

2.2 of the language, the concepts of class as type co-exists with the classic Python class. The standard way to differentiate between the two class models is solved on the inheritance level. If the developer wishes to use the unified type and class model,she adds a parent called “object”. More details on the topic can be found in the “languages used”.
In the given commit the switch from a classic to the new class model is performed on an existing class and thus “object” is added as a new parent.

Revision 1904, refactoring, small pain, new implementation

Commit 1904 factors out existing code of two classes into a newly created base class. By adding the new class, as parent to the existing, the functionalities remain unchanged and the code is shared. The size is less than 50 lines of code. According to the set of rules this justifies the label of a refactoring and “small pain”. Nevertheless two classes grow from non to single-inheritance users.

Revision 2023, requirement, large pain, new polymorphism

The commit 2023 is an example for the alteration leading to a new class employing multiple inheritance to fulfil the requirements. The persisted developer effort introduces two new classes and adds in total 8 new parents to the existing classes. The commit touches 18 Python files located in the three components: GUI, GTK GUI and core.
The new parent added in all the classes is “deluge.Component” a classic Python class and hence not type-unified. In combination with a new registry service for components all the new children can, through the implemented interface, be started, updated or stopped. The component interface is integrated to the different parts of the GTK GUI on the level of widgets. In order for the core code to update a widget, a new, decoupled way through the registry service and the common component interface is introduced.
Some widgets already have a parent to reuse implementation code and hence become multiple inheritance users. The component interface can not be added at the parent level, because this are classes of a third party library.
An alternative to this implementation of the integration of a portable, decoupled look up and control service without the usage of a shared interface seems very unlikely. Accordingly the level of pain is set as “large pain”. The commit introduces new polymorphic (although not type safe) relations among classes of different components.

Revision 2027, bug fix, large pain, existing polymorphism

The commit 2027 is an example for how a newly introduced solution using a new polymorphic relation of commit 2024 can lead to follow up changes. For a specific class the behaviour does not meet the requirements, because the integration into the new framework was simply forgot by the developer. Thus in order to return in to the borders of desired states, the class is integrated by adding the missing parent.
Given the complexity of the existing framework employing polymorphism, the need to fix the bug and no alternative solution the commit is tagged as “bug fix” and “large pain”.

Twisted – Samples of Code

Revision 6560, requirement, large pain, existing implementation

This commit is an example for the implementation of a requirement, that leads to the evolution of two existing classes from using multiple to single inheritance by removing a shared parent.
The modification is internal to one component. Although the correspondent test requires to be adjusted residing in the test component. The main classes represent two mutual protocols: TCP and SSH. Seven new classes are introduced, of whom one is using multiple inheritance, five single inheritance and one is a mixin class. The change is interesting as it shows the usage of dynamic meta-programming. The scenario is a the class “connection” representing an abstract interface for socket-based file descriptors. It provides read, write, open and close. The developer's task is to integrate TLS (transport layer security). Thus she implements a TLS mix-in class with the desired additional functionality and overwrites certain features such as read, write, open and close. To the connection class new features are added such as start TLS. If a client through delegation executes this feature on a target object, the code is interpreted at run-time in the connection class. This creates a new ad-hoc Python class called “TLSConnection”. It inherits from both the TLS mix-in class and the “connection” class. In the next step the new class replaces “connection” as class through attribute assignment (self.__class__) on the target object. Thus this instance object modified it's behaviour at run-time. Future feature calls on the same instance are not served by the connection class but by the TLSConnection class. This is an example of the replacement the polymorphic “strategy pattern” by a new mechanism.
In order to categorize the change we have to analyse the reasons for the alteration on the type relation layer. Thus the code inside the mix-in class for connection was moved from the TLS (or SSL)

connection class. As the TLS connection can hence be switched to at run-time the “TLS connection” class becomes obsolete. A TLS client or server class does not need to inherit from both connections types. Therefore the classes are after the removal of one parent new single inheritance users.
The dynamic change to the new class represents a new feature and thus a requirement. The dependence on the Python version of polymorphism and the complexity of the modification are reasonable factors to label as “requirement”, “large pain” and the addition of a new polymorphic structure.

Revision 12856, bug fix, medium pain, new implementation

In order to fix two similar bugs in in two classes, the malfunctioned code is moved into an existing class, fixed and made available through inheritance.
The persisted development effort is labelled as “bug fix” and “medium pain”. However it is another example as how existing classes become users of single inheritance through the introduction of a new implementation inheritance relation motivated by the replacement of similar defunct code.

Revision 20862, bug fix, small pain, new implementation

In commit 20862 a given layout of a test class does not comply with the set-up and tear down policies of the project. Thus the test cases do not provide the right interface. As the test case requires additional functionalities for the set-up and tear down procedure, the new code is placed into a new parent, a mix-in class. The existing class then inherits from both: The standard test interface implementation and the individual mix-in class.
The given change is an example for the circumstances leading to the metamorphosis from a single to a multiple inheritance class. The modification is labelled as “bug fix”. As the same effect could have been achieved by a simple overwrite of the existing implementation interface, the rules denote the “small pain” category. Of course the given solution seems more elegant.

Revision 22257, integration, large pain, new polymorphism

The commit merges a development branch into the trunk providing a new framework for a multi-step user authentication. The new component adds seven new files and a total of 25 new classes. Five are using multiple inheritance and 19 single. A new mix-in class provides features, that at run-time turn the given object into a part of the framework. This achieved by loading and registering the implementation objects for the required interfaces to interact with

the new framework. Thus the existing class has to inherit from the mix-in class too and changes from single to multiple inheritance ancestor structure.
The change is labelled as an integration task and as “large pain”. It seems difficult to find a procedural alternative to the introduction of new polymorphic relations.

Zope – Samples of Code

Revision 14468, refactoring, large pain, divide/merge

The commit is exceptional in the sense that to a class not using inheritance at all in one go four new parents are added. From the Zope perspective the change is nothing special. A given utility class is integrated into the various services and components the application server provides. This is persistence support, configurable permissions and roles, access through acquisition and a collection of services ranging from FTP support to drag'n'drop integration.
While persistence, security and FTP support are self-explaining, for acquisition that might not be the case. It stands for a concept orthogonal to inheritance. It is designed for and heavily used inside Zope. For example given two instance objects of two mutual classes not sharing a common ancestor. One represents a fridge and the other a bottle of beer. Assume the bottle is located inside the fridge along with 12 other bottles of milk. Thus the fridge can be seen as a kind-of-a “container”. Acquisition allows the individual bottle object to access the properties and services of the “container” object. The bottle does not require a reference for the container and can access the fridge exactly in the same way as itself. For example the beer bottle object calls the function “all bottles”. As it does not exist inside the beer bottles class definition, the function or attribute is tried to be located inside the ancestry of the “beer bottle” class. Under the assumption that this search fails, the acquisition code restarts the search from the nearest “collection” object, the fridge. Under the assumption the fridge object has an attribute called “all bottles”, it's value is returned, in this case it would be 13. What is the advantage of acquisition? Imagine the beer bottle object is moved from the fridge and placed on the table. If the same call is performed again, this time the table as “collection” is providing the required information. To be precise enough acquisition does also work for contexts.
Long story short the motivation for Zope's acquisition is that the developer can concentrate more on the unique part of the object. Acquisition provides a managed registry service on containers.

Further the container becomes a parent of it's entries.
The technology automatically enables adaptation of behaviour depending on where the object is located or moved to and thus allows a new way of collaboration.
On one hand, as mentioned before, the commit enables the core services for the Python product implemented by adding the four parents. On the other hand the motivation is a refactoring and thus the found change on the type-level relation is just a side effect. Accordingly the “hot spot” is tagged as “large pain” and refactoring.

Revision 16667, refactoring, small pain, existing polymorphism

The commit contains less than 20 lines of changed code in a single file. It is a syntactical code review and creates compliance by adding the common exception base class as parent to the locally used exception types.
As the change is not needed because no external access exists the change is labelled as “small pain” and “refactoring”.

Revision 20616, requirement, large pain, new implementation

The goal of the persisted development effort is to enhance performance. The size of the commit is nearly 20 files located in seven components. Two classes are removed, one new introduced and three classes are modified from single to multiple inheritance through the addition of the same parent. Performance test showed, that the old implementation of a given feature is faster than the currently used newer one. The developer decided to integrate the old code into the new parent class added and becomes reusable. The new parent class unifies other functionalities from it's ancestry and is designed as mix-in class. Thus usage as a member is impossible and therefore no meaningful alternative is feasible to achieve the same effect. Accordingly the commit is annotated as “large pain” and “requirement”.

Changed existing type-relations versus "level of pain"

The focus of the subsection lies on understanding, whether the developers of the different projects perform the same tasks on the classification scale of “levels of pain”.
Let me remind the reader that the aggregation onto the level of OO-PL showed a high degree of homogeneity in the percentage distribution.

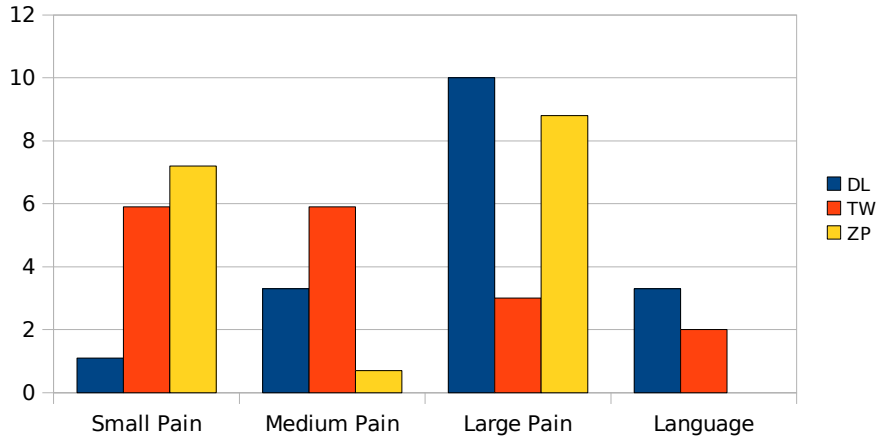

Chart 30: "level of pain" [per mil of changed commits]

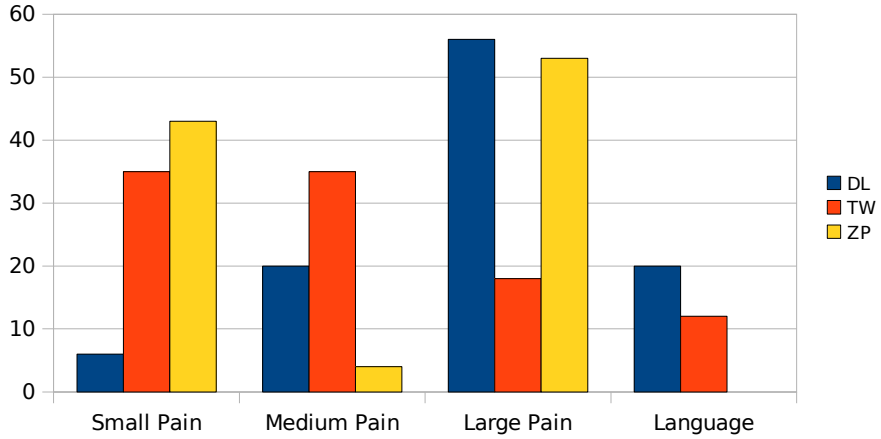

Chart 31: "level of pain" [percentage of observations]

Chart 30 shows the behaviour on the “pain scale” for the reviewed commits. The bars of each project are in per mil of total commits with a change in code. They add up to the values presented in chart 29.

In comparison to the previous chart the distribution for the different project could be used for a discriminatory function. For Deluge and Zope the maximum value accounts for more than 50% of all observed cases (with the language imposed removed). The smallest value for Deluge lies, with one change in thousand observed commits, in the category of “small pain”. For Zope the minima is at the “medium pain” category.

Deluge's “large pain” changes are composed by 2/3 of integration tasks and the rest contain requirement implementations and bug fixes. The 20% of the commits in the category of “medium pain” are mostly concerned with the fulfilment of requirement tasks and reuse existing implementation inheritance structures. This eases the development of the desired new functionality. In the “small

pain” category only one observation could be identified.
Twisted's distribution among the different pain categories has a truly individual shape. First to note is the fact that more than 70% of all observations, without the language imposed ones, are not motivated through large pain. The reader should thus not be surprised that the number of observed integration tasks is small. The observations in the small and medium pain category are dominated by refactorings and bug fixes. Further more most introduce a new implementation inheritance relation by factoring out existing code into a new parent. This new parent class has, in most of the cases, only a single child. Interestingly for bug fixes the parent usually contains the replaced defunct code. The large pain category is dominated by the integration tasks.
Zope, as mentioned above, has it's maximum in the “large pain” category. This is no surprise, as the application server system provides many functionalities and services a normal class has to support and participate by inheritance. “Large and small pain” observed alterations on the type-relation level total to over 96%. The composition of the “large pain” category contains not one dominant factor. For the “small pain” category with 50% of the observations “refactoring” is the dominant development goal.

Changed existing type-relations versus "development goal"

The projects come from different domains of applications and have thus different designs. Some have fewer components, some have more. From the language level comparison we could identify that most of the changes are in the bug fix category. This is true for two of the projects.

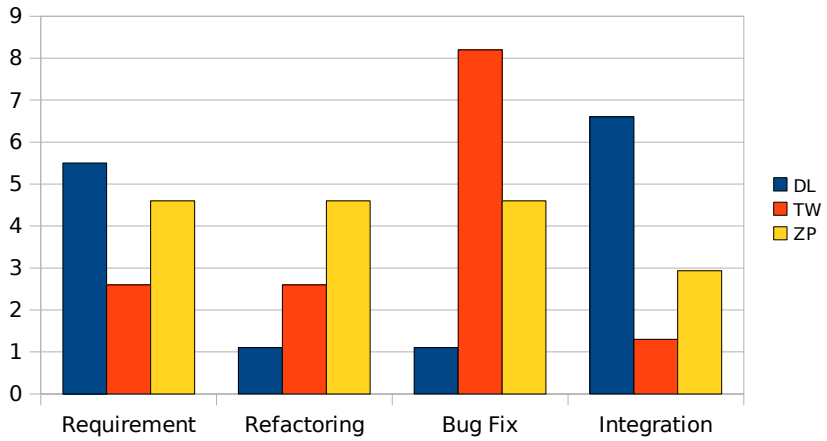

Chart 32: "development goal" [per mil of changed commits]

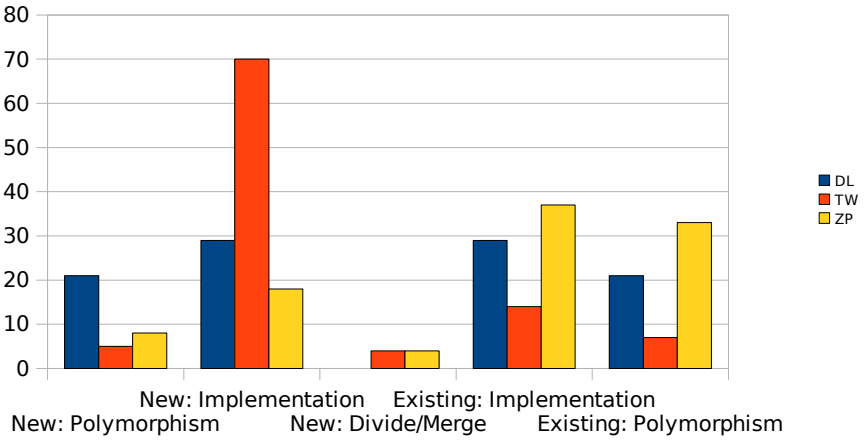

Chart 33: "classifications of alteration" [percentage of observations]

Chart 32 visualises for the different Python projects the observations distribution among the four-kind of development goals. The values add up to the per mil total shown in chart 29 excluding language motivated modifications.

For Deluge nearly 50% of all observations are integration tasks and imply a high degree of pain as already observed in the last subsection. In per mil of commits this is the highest value of all projects, although the project has in total the fewest observations of the three. Most of the observations are from the same two components integrating the GTK UI into the existing UI abstraction or adapting it to the underlying C back end. Thus although the modifications are not large in terms of changed lines of code, they are responsible for the large per mil value in this category.

The high value for requirements as development goal represents mostly small existing or new classes to share code among two or three children. Thus they belong to the “medium pain” category. This could be the personal “gusto” of the single developer responsible for more than 80% of all observations.

Twisted has it's maximum for bug fixes. The per mil and the percentage value is the maxima for the three projects. It event tops all C++ projects and EiffelStudio. The observations are many times unit tests that cover the faulty class. During the review or the addition of new test functions, the test classes are split into possibly multiple base classes. The base classes do not contain test routines but provide sometimes complex implementation code used by the test class and is accessed through implementation inheritance. This pattern is not motivated by the language and is more likely a common practice in the group of seven core developers responsible for most of this kind of changes.

Zope has an evenly balanced distribution among the three different development tasks observed. As for Twisted the minima are the integration tasks. Nearly all of the integration observations belong

to the “large pain” category and most reuse existing polymorphic or implementation hierarchies. The bug fix categories have their maximum in creating new single inheritance classes by the addition of an existing parent class for implementation inheritance. It can be motivated by a missed opportunity of the original designer to reuse a service or functionality provided inside the Zope application server such as acquisition, persistence or a specialised string function providing better security.
During the review of the observed commits, it was sometimes difficult to decide between integration and requirement based changes as more than half either reuse an existing polymorphic relation or introduce a new one. For example acquisition is difficult to judge, because it can be seen as integrating the class to the whole framework or as a tool enabling the implementation of a specific requirement.

Changed existing type-relations versus "classification of alteration"

The application domain of the project differ and the solutions implemented have different designs. Some have fewer components, some have more. From the language level comparison we could identify that the small overall values for Python can be explained as weakness in the “existing: implementation” and “existing: polymorphism” category.

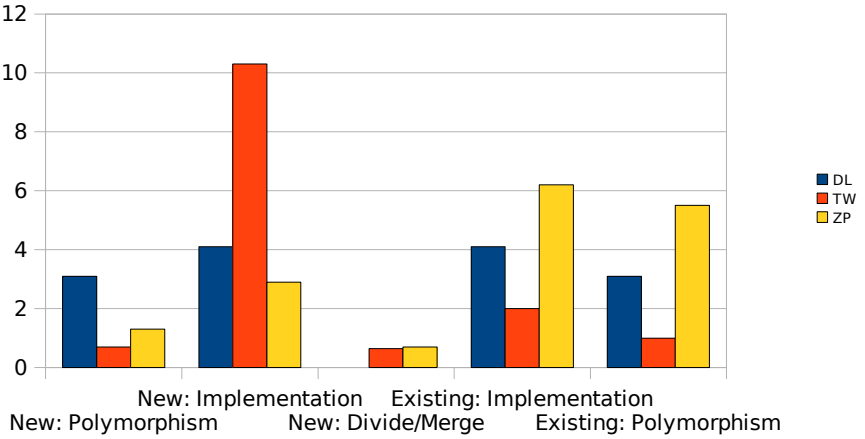

Chart 35: "classifications of alteration" [per mil of changed commits]

Chart 34 includes the number of observations for the different modifications included leading to a change on the type relation level of an existing class. The values are in per mil of the total observed commits excluding the language imposed ones. Thus the value for each project adds up to the total shown in chart 34.
For Deluge the new and the existing relations are evenly balanced.

Thus the 80 to 20 rule does not explain much. The “requirement” per mil is responsible for the “implementation inheritance” per mil. The per mil of integration tasks are distributed between implementation and polymorphism.
Twisted developers mostly decided to integrate a new implementation inheritance in order to enhance the understanding or reuse of the code during a bug fix and any other category of development goals. In comparison to Deluge and Zope, as a consequence, the project is very weak in the usage of polymorphism. The reasons for this is the absence of a powerful set of services to integrate and of a GUI. Both would require to implement and interact more with existing hierarchies.
Zope has the highest total per mil of the three projects and the reasons are straight forward. The project requires most of the “product” classes, that are executed on the application server, to integrate or use a powerful set of services or functions to implement their specific requirements. The reuse of existing parent child relations, to adapt already written classes, dominates over the introduction of new relations. The value for the integration of new “implementation inheritance” relations is similar to Deluge, but can not withstand the comparison with Twisted. The few new polymorphic relations integrated are very complex interactions among different objects, possibly combined with dynamic run-time changes of the class hierarchy.

Results: Java projects versus projects

Introduction

In this section we take the same approach as on the inter OO-PL comparison but for the projects of the Java programming language. Further details on the language can be found in the “languages used” section. The selection of the projects for Java is composed of Hadoop (HD), ActiveMQ (AQ) and Tomcat (TC) and followed the assumptions described in the section “projects used”.
As we have seen in the inter OO-PL comparison Java performs as good as the others, when it comes to the usage of polymorphism to solve the development tasks on existing classes. For the reuse of functionality on the declarative level rather than through delegation the fewest observations are found for Java.
The language Java does not support the concept of multiple inheritance. It distinguishes between interfaces and classes. While interfaces can have multiple parents, a class is restricted to a single class as parent.
The different Java projects analysed are heterogeneous along the application domain, the age and the numbers of authors.

All commits versus changes to Java code

For all three projects analysed in Java, we present the numbers of scanned commits and the percentage of those incorporating at least one line of changed code.

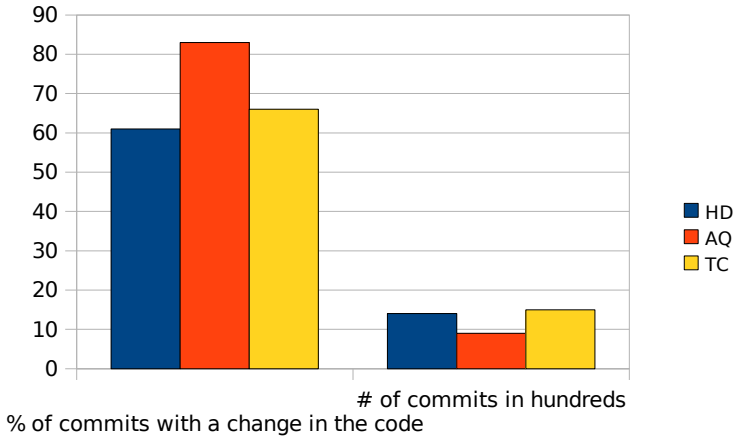

Chart 36: Changes of code [percentage of all commits]

Chart 36 shows the number of commits in hundreds for the different projects analysed in Java.
As already seen at the language level the projects are very similar

in the degree of commits without actual change on the code basis. Hadoop's non Java commits maintain the configuration files, adapt the HTML documentation or revise the scripting language programs. We scanned the existing repository of 10 months to the first revision. The old repository is not accessible any more.
ActiveMQ has the highest percentage of actual Java commits. The non-programming language changes result from the adaptation of the Maven build system configuration files in XML. The project was automatically processed for the last year and not for the whole 3 years available. The reasons lie on technical problems of our side.
Tomcat's code basis consists mostly of Java. Some configuration work is required as to modify the classpath, the properties files or some of the scripts. The analysis looked through the available revisions of the last three years.

Changed commits versus changed existing type-relations

For the projects differences exist in the total alterations on the type relation level of existing classes. We present, how many of a 1000 commits modify the existing code base and alter the existing type relation level.

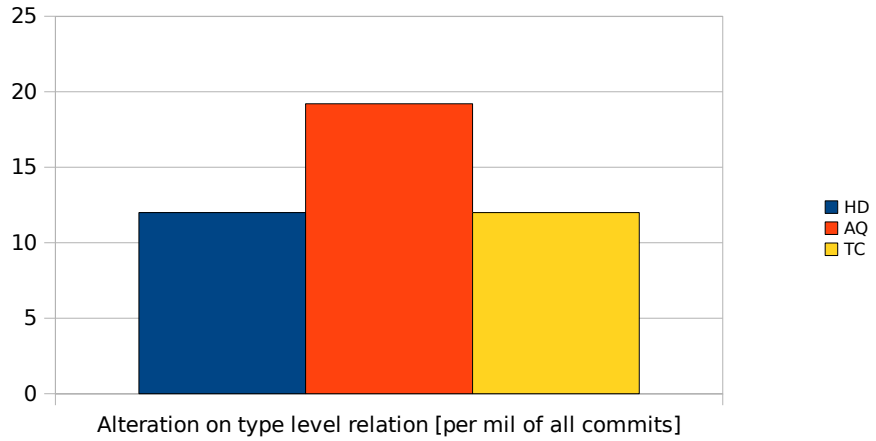

Chart 37: Changed existing type-relations [per mil of changed commits]

Chart 37 shows the per mil commits that contain a change on the type-relation level of existing classes. The values are thus comparable with all other charts of the other programming language and projects.
Hadoop's and Tomcat's values are identical. The ActiveMQ project resides with the 19.2 per mil observations above the Python and some C++ projects.

Hadoop – Samples of changes

Revision 696551, integration, large pain, existing polymorphism

The commit consists of four Java files distributed over three components. The modified class on the type relation level implements a newly added interface “tool”. It is provided by a different component offering the global services of a thread dispatching mechanism for “tools” relying on dynamic binding. According to the set of rules defined this change is labelled as “integration” and “large pain”.

Revision 686840, integration, large pain, existing implementation

The commit modifies an existing class on the type relation level by adding a parent. It is used for the integration of the global mechanism to share the access to the configuration file of the software. As this is the standard way for the project implementing an alternative approach to share the access is not feasible.
Thus according to our rules and given the large set of other children of the parent, the change is labelled as “large pain”.

Revision 706704, refactoring, medium pain, new polymorphism

The commit fixes a critical issue (Hadoop-4230). It involves around 420 files in three major components of the software. The topic is the replacement of a specific operators of the query language used in the MapReduce computation. However, the existing class modified on the type level is not a single one, but a set of over 30. Still the modification is always the same as a former abstract parent class is changed into an interface. This modification is propagated to all the it's children.

ActiveMQ – Samples of changes

Revision 617015, requirement, large pain, existing polymorphism

The commit 617015 modifies eight files in the same component. The standard way for the classes is to provide support for observers that listen to a set of states through a specific interface. This set is extended to represent a new functionality. Thus the existing class providing the new state has to allow observers to subscribe.
The change is labelled as “requirement” and “large pain”.

Revision 660977, bug fix, medium pain, existing implementation

The commit 660977 modifies files in three interacting components. The “hot spot” class used to implement the interface of it's component directly as three other classes as well. In order to fix a

bug in the interface this classes were reviewed. Our “hot spot” class thus replaced it's direct implementation of the interface. It added a sibling as parent and overwrites the individual part.
As the existence of the participation in the polymorphic relation does not change, the “implementation reuse” part, motivated the developer to adapt the type-level relation of the class. Accordingly the change is marked as “bug fix” and “medium-pain”.

Tomcat – Samples of changes

Revision 522773, integration, large pain, existing polymorphism

As the other Java commits also 522773 is rather simple to explain. An existing class is modified to accept component external observers and provide information on it's internal state. Thus it has to implement the required existing interface.
The participation in an existing polymorphic design is labelled as “integration” and “large pain”.

Revision 645175, integration, large pain, existing polymorphism

In commit 645175 an existing component is integrated into the larger context of the application. Some missing methods have to be implemented and accessible through the interface. To add an external interface on each interacting class would be a solution. The developer decided to let the internal interface extend the external. This requires all children to adapt.
There is no alternative to the participation of the component in the application wide framework. Accordingly somehow the global interface has to be implemented. The commit is labelled as “integration” and “large pain”.

Changed existing type-relations versus "level of pain"

The focus of the subsection lies on understanding, whether the developers of the different projects perform the same tasks on the classification scale of “levels of pain”.
Let me remind the reader that the aggregation onto the level of OO-PL showed a high degree of homogeneity in the percentage distribution.

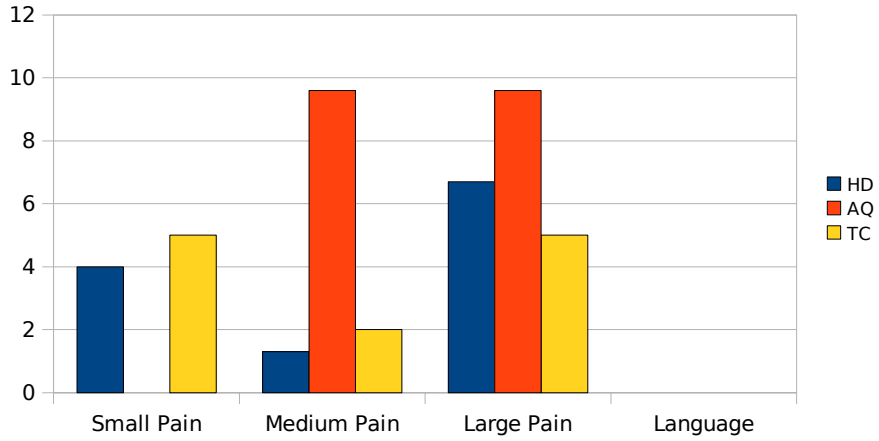

Chart 38: "level of pain" [per mil of changed commits]

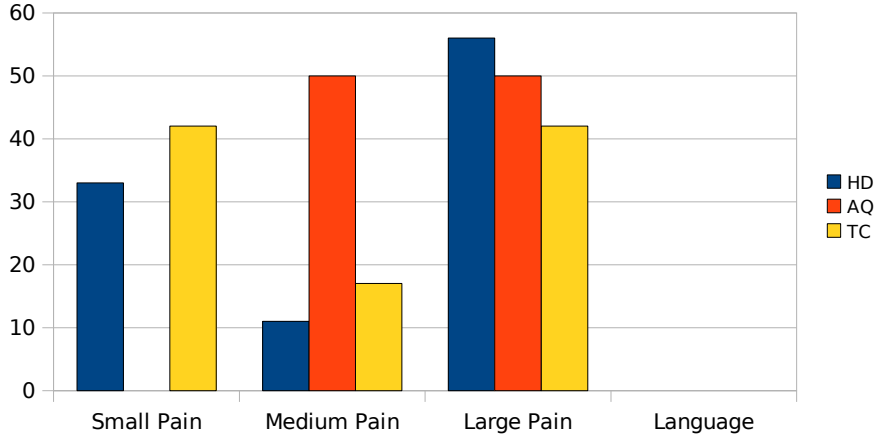

Chart 39: "level of pain" [percentage of observations]

For Deluge most of the few observations are in the large pain category. In the per mil chart 38 it can compete on “large pain” and “small pain” but not for “medium pain”.
ActiveMQ is strongest in the “medium” and “large pain” category A “small pain” modification could not be detected.
Tomcat shows a similar distribution as Hadoop on the percentage and on the per mil chart.
The degree of homogeneity is not that small as for other languages. But for a more fine-grained picture further samples are required. As the phenomena is so rare, it requires many more thousands of commits to be scanned.

Changed existing type-relations versus "development goal"

The projects come from different domains of applications and have thus different designs. Some have fewer components, some have

more. From the language level comparison we could identify that most of the changes are in the requirement category. The rest should can be freely distributed among the categories. The differences among the projects in the “requirement”, “refactoring” and “integration” category are not significant and thus the values can be assumed to be the same.

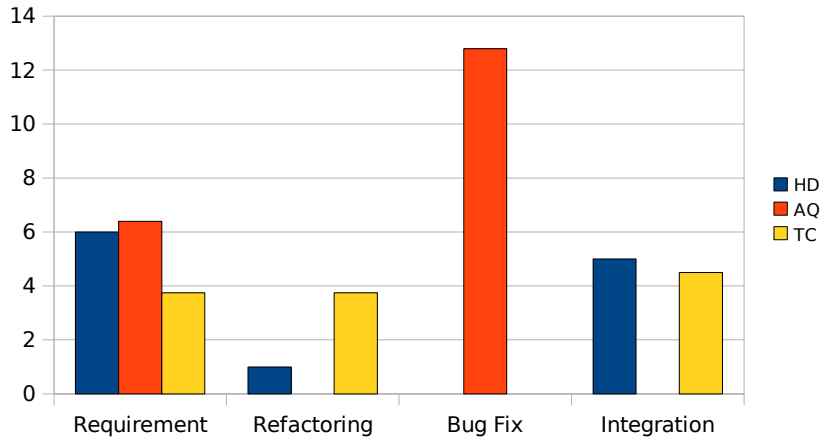

Chart 40: "development goal" [per mil of changed commits]

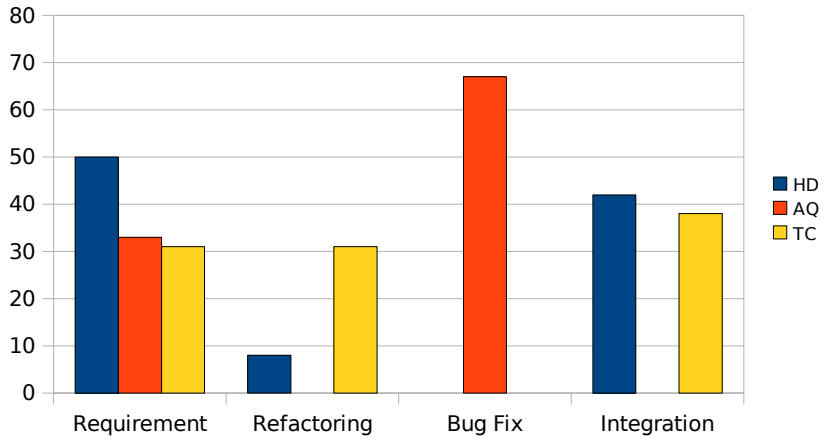

Chart 41: "development goal" [percentage of observations]

Hadoop performs in chart 41 in per mil of commits the same as the other three projects for “requirement”. The absence of any observation in the category of “bug fix” can not be explained.
For ActiveMQ the category of “bug fix” dominates. But given the the total number of reviewed cases, this has to be questioned too.
For Tomcat the problem with the few data points is less sever. Thus the numbers are more representative and allow to conclude, that the development team has more often performed adaptive, perfective or preventive modifications than corrective.

Changed existing type-relations versus "classification of alteration"

The final subsection of the results section on the Java projects presents a comparison of the “Categories for “classifications of alteration””.

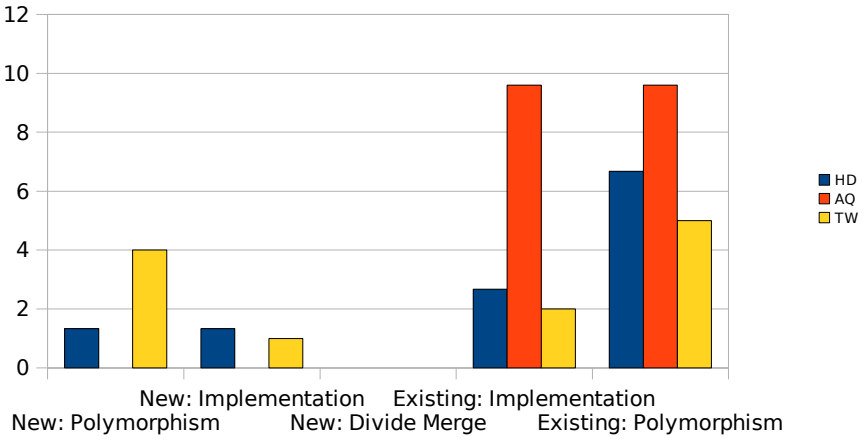

Chart 42: "classifications of alteration" [per mil of changed commits]

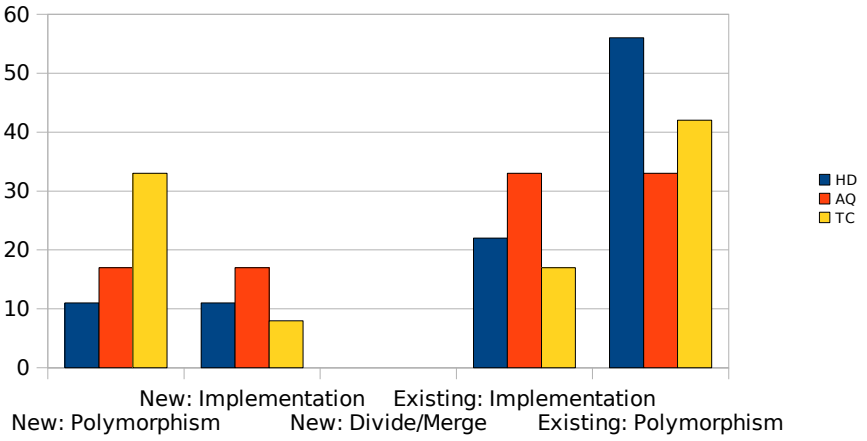

Chart 43: "classifications of alteration" [percentage of observations]

Chart 43 includes the number of observations for the different modifications included, leading to a change on the type relation level for the Java projects.
It explains the aggregated results for the OO-PL versus OO-PL comparison. The percentage graph shows major differences and commonalities as expected.
All projects show a similar level in per mil for changes falling in to the “existing: polymorphism” category. A significant difference exists for ActiveMQ in the “existing: implementation” category. The other categories can not be used for inter project-comparison as the

differences are to small. This is fair enough, as the sole difference explains the higher overall value for ActiveMQ.

Bibliography

1: Wegner,, Peter, Concepts and paradigms of object-oriented programming, 1990
2: Meyer,, Bertrand, Genericity versus inheritance, 1986
3: Clemens A. Szyperski, Import is Not Inheritance Why We Need Both: Modules and Classes, 1992
4: Stroustrup,, Bjarne, The C++ Programming Language, 2nd ed., 1991
5: Stroustrup,, Bjarne, The C++ Programming Language, 3rd ed., 1997
10: Meyer,, Bertrand, Object-oriented software construction (2nd ed.), 1997
15: Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1995
6: Arnold,, Ken and Gosling,, James and Holmes,, David, Java(TM) Programming Language, The (4th Edition), 2005
7: Lutz,, Mark, Programming Python, 2006
8: Prof Dr H. Mssenbck,, Prof Dr N. Wirth and Josef Templ, Metaprogramming In Oberon, 1994
18: Michele,,Simionato, The Python 2.3 Method Resolution Order, 2003
9: Parnas, D.L., Software aging, 1994
19: Stefan Koch, Evolution of Open Source Software Systems: A Large-Scale Investigation, 2005
17: Simon, Frank and Steinbruckner, Frank and Lewerentz, Claus , Metrics Based Refactoring, 2001
16: Pinzger,, Martin and Gall,, Harald and Fischer,, Michael and Lanza,, Michele, Visualizing multiple evolution metrics, 2005
11: Lientz,,BP, Swanson,, EP, Software maintenance management, 1980
12: Armstrong,, Deborah J., The quarks of object-oriented development, 2006
13: Crowston, Kevin and Howison, James , The social structure of Free and Open Source software development, 2005
14: Smart,, Julian and Hock,, Kevin and Csomor,, Stefan, Cross-Platform GUI Programming with wxWidgets (Bruce Perens Open Source), 2005

Name	New Lines	Deleted Lines	Modified New Lines	Modified Del. Lines
Codelite	30656, 5%	35042, 6%	273246, 47%	236983, 41%
QPID	66190, 28%	47948, 20%	67643, 30%	49558, 21%
Slicer	522802, 57%	43308, 5%	217779, 24%	128799, 14%
C++				
Estudio	3407224, 32%	3144324, 30%	2404717, 23%	1696150, 16%
Goanna	106487, 61%	25792, 15%	26811, 15%	14546, 8%
Gobo	939344, 30%	1082984, 35%	489307, 16%	620703, 20%
Eiffel				
Twisted	189858, 23%	142696, 17%	301240, 36%	191889, 23%
Deluge	57071, 53%	10546, 10%	22128, 21%	17557, 16%
Zope	134276, 30%	11645, 3%	162310, 36%	145254, 32%
Python				
Hadoop	82508, 36%	70706, 31%	47494, 21%	26022, 11%
ActiveMQ	31712, 15%	5900, 3%	90840, 43%	80729, 39%
Tomcat	76840, 33%	10629, 5%	76803, 33%	67023, 28%
Java				

Table 1: The different projects of the OO-PL paradigm's basket: Modifications in cLoC

Name	Type	Age	Weight	#Revisions	#Authors	Commits w. CLoC [%]	Pareto cLoC # Authors	%
Codelite	IDE	0.9y	0.17	1463	3	79	1	33
QPID	Net. Prot.	2y	0.17	1506	15	76	3	20
Slicer	GUI App.	2.5y	0.66	5723	42	62	7	17
C++		*5.4y*		*8692*	*60*	*67.29*	*13*	*23.33*
Estudio	IDE	13y	0.95	69281	93	70	18	20
Goanna	Web Serv.	8y	0.01	528	13	76	2	15
Gobo	Compiler	7y	0.04	2828	13	78	2	14
Eiffel		*28y*		*72637*	*119*	*70.36*	*22*	*16.33*
Twisted	Net. Eng.	7y	0.42	9321	47	80	12	25
Deluge	GUI App.	2y	0.12	2784	8	37	1	12
Zope	App. Serv.	12y	0.46	10316	96	59	13	13
Python		*21y*		*22421*	*151*	*65*	*26*	*16.67*
Hadoop	Prog. FW	0.8y	0.36	1456	22	61	6	25
ActiveMQ	Msg. Eng.	3y	0.25	991	12	83	2	16
Tomcat	App. Serv.	3y	0.39	1577	14	66	3	21
Java		*6.8y*		*4024*	*48*	*63.93*	*11*	*20.67*

Table 2: The different projects of the OO-PL paradigm's basket: Age, Type, Authors, Pareto of cLoC

Name	Classes Added			Existing Classes		
	Multiple	Single	None	Multiple	Single	None
Codelite	3, 1%	198, 64%	71, 23%	0, 0%	2, 0.6%	1, 0.3%
QPID	70, 7%	301, 31%	233, 24%	16, 2%	28, 3%	11, 1%
Slicer	0, 0%	692, 70%	151, 15%	0, 0%	1, 0%	0, 0%
C++						
Estudio	8125, 23%	10166, 28%	5833, 16%	1274, 4%	1026, 3%	190, 1%
Goanna	374, 31%	358, 30%	199, 16%	42, 3%	28, 2%	2, 0%
Gobo	526, 26%	772, 38%	76, 4%	269, 13%	87, 4 %	8, 0%
Eiffel						
Twisted	245, 4%	3091, 56%	635, 12%	80, 1%	121, 2%	24, 0%
Deluge	17, 2%	505, 47%	407, 38%	3, 0%	26, 2%	2, 0%
Zope	208, 4%	1990, 43%	1245, 27%	54, 1%	67, 1%	18, 0%
Python						
	With Interface		*No Interface*	*With Impl. Inh.*		*No Impl. Inh.*
Hadoop	152, 8%		435, 24%	88, 5%		0, 0%
ActiveMQ	69, 14%		154, 31%	5, 1%		7, 1%
Tomcat	100, 16%		108, 18%	13, 2%		3, 0%
Java						

Table 3: The different projects of the OO-PL paradigm's basket: Modifications on the class level – Part 1/2

Name	Commits with TRC					
	[per mil]	]0,100[*	[1h,2h[*	[2h, 4h]*	[4h, 8h[*	[8h, inf)*
Codelite	12.6	2	3	2	4	2
QPID	88.6	16	6	22	20	32
Slicer	21.2	2	2	2	4	3
C++	*31.43*	*0.16*	*0.09*	*0.21*	*0.23*	*0.3*
Estudio	92.7	67	10	8	6	8
Goanna	303.3	28	21	21	20	11
Gobo	192.7	124	32	45	37	147
Eiffel	*98.12*	*0.37*	*0.11*	*0.13*	*0.11*	*0.28*
Twisted	17.7	28	18	16	25	37
Deluge	16.7	14	0	0	2	1
Zope	16.6	68	12	4	4	9
Python	*17.07*	*0.46*	*0.13*	*0.08*	*0.13*	*0.2*
Hadoop	12	4	1	2	2	1
ActiveMQ	19.2	2	4	5	2	2
Tomcat	12	5	3	0	0	4
Java	*13.77*	*0.3*	*0.22*	*0.19*	*0.11*	*0.19*

*Table 4: Commits with type relation changes of existing classes in per mil and size of the commit in cLoC**

Name	Deleted Classes		
	Multiple	**Single**	**None**
Codelite	3, 0.9%	18, 5%	13, 4%
QPID	34, 4%	175, 18%	92, 10%
Slicer	0, 0%	100, 10%	46, 5%
C++			
Estudio	3290, 9%	3568, 9%	2434, 7%
Goanna	83, 7%	57, 5%	40, 3%
Gobo	114, 6%	132, 7%	26, 1%
Eiffel			
Twisted	64, 1%	1048, 19%	187, 3%
Deluge	1, 0%	75, 7%	42, 4%
Zope	91, 2%	585, 13%	402, 8%
Python			
	With Interface		*No Interface*
Hadoop	41, 2%		151, 8%
ActiveMQ	16, 3%		39, 8%
Tomcat	3, 0%		11, 2%
Java			

Table 6: The different projects of the OO-PL paradigm's basket: Modifications on the class level – Part 2/2

Name	Type	Age	Weight	#Revisions	#Authors	Commits w. CLoC [%]	Pareto cLoC # Authors	%
Codelite	IDE	0.9y	0.17	1463	3	79	1	33
QPID	Net. Prot.	2y	0.17	1506	15	76	3	20
Slicer	GUI App.	2.5y	0.66	5723	42	62	7	17
C++		*5.4y*		*8692*	*60*	*67.29*	*13*	*23.33*
Estudio	IDE	13y	0.95	69281	93	70	18	20
Goanna	Web Serv.	8y	0.01	528	13	76	2	15
Gobo	Compiler	7y	0.04	2828	13	78	2	14
Eiffel		*28y*		*72637*	*119*	*70.36*	*22*	*16.33*
Twisted	Net. Eng.	7y	0.42	9321	47	80	12	25
Deluge	GUI App.	2y	0.12	2784	8	37	1	12
Zope	App. Serv.	12y	0.46	10316	96	59	13	13
Python		*21y*		*22421*	*151*	*65*	*26*	*16.67*
Hadoop	Prog. FW	0.8y	0.36	1456	22	61	6	25
ActiveMQ	Msg. Eng.	3y	0.25	991	12	83	2	16
Tomcat	App. Serv.	3y	0.39	1577	14	66	3	21
Java		*6.8y*		*4024*	*48*	*63.93*	*11*	*20.67*

Table 7: The different projects of the OO-PL paradigm's basket: Age, Type, Authors, Pareto of cLoC