Continuous Integration with Architectural Invariants

A case study about architectural monitoring in practice

Master Thesis

Oskar Truffer
from
Bern BE, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

September 9. June 2015

Prof. Dr. Oscar Nierstrasz

Andrea Caracciolo

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

Software erosion is a common problem in larger software projects [3, 5]. As a code base grows, more effort needs to be invested to keep an implementation aligned with its intended architecture. Designers may need to introduce new architectural guidelines as existing ones no longer fit the complexity or the purpose of the project. The code base is then gradually refactored in order to conform to these new guidelines. It's hard to determine for an architect which parts of the software conform to the new architecture and which parts need to be revisited. Once introduced the new guidelines too are subject to architectural erosion. Continuous effort is needed to keep an unintended drift from the intended architecture.

In this paper we introduce automated architectural conformance checking into the development process of the open source project ILIAS. The solution formalizes design decisions as architectural invariants and offers automated tests for them. The developers get feedback about all existing violations of the architectural invariants with emphasis on added or removed violations introduced within each contribution. We discuss the process of introducing our automated conformance checking solution into the open source community. Our results show that our solution contributes to the decline in architectural violations and the reduction in the gap between architectural documentation and actual implementation.

Contents

1 Introduction

Software systems rely on architectural coherence in order to be maintainable [2, 9]. As the software ages, parts of the architecture need to be refactored in order to adapt to emerging requirements and to be easily maintainable. Existing architectures undergo a constant unintentional drift as new contributions to the project may not respect the original design [3, 5]. During the evolution of the system, these drifts as well as intentional changes to the architecture need to be monitored to minimize the gap between the intended and the implemented software architecture [2]. If no proper monitoring infrastructure is in place, the drift may cause the design goal to be compromised and render maintenance of the system increasingly difficult [3].

If we consider unit testing, a lot of code repositories are regularly checked by a continuous integration server. This iterative process guarantees that core functionality works as intended even after a big refactoring. This approach works well for testing small units of the software but it gives no feedback about the degree of architectural conformance of the whole system. For example, unit tests may pass but in the latest commit there may still be an architectural violation where the model in an MVC pattern depends on the controller. That's where Dicto[1] comes in. Dicto is a simple declarative language that allows one to specify architectural rules, which can then be verified automatically. It allows for a continuous automated architectural monitoring which contributes to increased maintainability and more coherent software architecture.

With Dicto we introduce architectural invariants to the open source software ILIAS[2] and discover whether automated feedback about added or resolved violations against these invariants leads to a decline in violations. ILIAS is an open source learning management system that is developed by and for different companies and public institutions. ILIAS represents a good candidate for a case study: In open source projects, architectural drift is especially pronounced as developers tend to work in isolation on distinct features with little coordination [10]. ILIAS is actively developed and is a good example of a software project aging over years while trying to keep a high standard of maintanability.

ILIAS isn't backed by one single company but different service providers offer customization and extensions to the current ILIAS code base which may be added to the ILIAS core if it's considered to be valuable for the whole community. This open source model leads to the problem that investment in time and money for cross-section functionality (*e.g.,* performance, exception handling) often comes short.

[1]`http://scg.unibe.ch/dicto/`
[2]`http://www.ilias.de`

Refactoring may take a long time as the service provider individually refactors parts of the code. Dicto supports the integration of distributed effort as it helps in keeping an overview of which parts of the code need to be refactored and gives feedback during the development process to all contributors.

In our study, architectural invariants are defined to migrate ILIAS towards a new architecture that entails a more robust exception and error handling policy. A continuous integration server then gives feedback to all contributors at every commit about added or resolved violations to these invariants and lists all currently present violations in the code. This serves two main purposes:

1. Maintain a list of all violations to work on.

2. Ensure that no new violation is added.

If these two points hold, the intended architecture will eventually be reflected in the code base. The second point is important since, despite any effort, architectural violations are typically growing and violations may even reoccur after being resolved [1].

2 ILIAS

ILIAS[1] is an open source learning management system (LMS). The project first started in 1997 at the University of Cologne and is distributed under the GNU general public license since the year 2000. There are 25 contributors, 21 of which are working on ILIAS since 2014 and are registered on GitHub[2]. According to the official ILIAS homepage[3], there are 181 known installations of ILIAS[4]. The user base per installation varies from 30 registered users at the mathematical institution of Goettingen to 90'000 registered users at La Poste, the main post company in France. These are just the registered installations (mainly public institutions) . There are some companies that did not want their installation registered in a public database. The ILIAS community is organized as a society, namely "ILIAS open source e-Learning e.V.", having 49 listed institutional members 12 service providers and private members [5]. Some institutions choose to extend ILIAS with their own resources and contribute back to the project through a service provider.

2.1 Organizational Structure

The 12 service providers offer development or customization of features for ILIAS. To ensure quality standards, every component in ILIAS has a first and a second maintainer. In most cases these are chosen among the twelve service providers. The maintainers are responsible for the quality of the functionality and code of their components and every contribution to a component has to be reviewed by its maintainer.

An important concept in the ILIAS community is the Jour Fixe. Every 2 weeks, members of the ILIAS community hold a physical meeting in Cologne to discuss all ILIAS related issues. Every decision related to functional, non-functional or process-oriented topics is decided upon on the Jour Fixe. The JF is moderated by the head of development and everybody having interests for the issues on the agenda is welcome.

[1]http://www.ilias.de
[2]https://github.com/ILIAS-eLearning/ILIAS/graphs/contributors
[3]http://www.ilias.de
[4]http://www.ilias.de/docu/goto_docu_dcl_3444.html
[5]http://www.ilias.de/docu/goto_docu_cat_3650.html

It's important to notice that the Jour Fixe is not a proactive organ. It can prevent functionality from being introduced into the code base but it cannot force any service provider to change any functionality that is already implemented. Once a feature is in ILIAS, the upkeep of quality standards is the responsibility of the maintainers.

2.2 Development Process

New functional requirements are most often demanded by institutions already using ILIAS or institutions introducing ILIAS as their LMS. In most cases these institutions work with one of the service providers to establish a concept. This concept is then added to the feature wiki [6] for discussion and is scheduled for a Jour Fixe. If the Jour Fixe decides that the feature is consistent with the product and may be useful for other users as well, that functionality will be scheduled for a future ILIAS release.

For big functional requirements and non-functional requirements the usual course of action is to establish a special interest group (SIG). The SIGs can be seen as lobbies representing specific interests within the ILIAS community and are used to add political weight to these interests. There is for example a SIG for Schools focusing on functional requirements designed for elementary and secondary education. Another example is the "SIG Performance" gathering funding for performance improvements and distributes best practices for deployment depending on the hardware amongst other things.

[6] `http://www.ilias.de/docu/goto_docu_wiki_wpage_1_1357.html`

3 Problem Analysis

As in every software system aging over time, the desire to refactor parts of the code emerged from the ILIAS contributors. The idea to found a special interest group (SIG) specialising on refactoring code across service providers started to develop with the main focus to make the life of ILIAS contributors easier. Members of the SIG were mostly developers from different service providers but the group is open to everyone. In small projects cross-section functionality (e.g. control flow or exception handling) should be analysed and in some cases refactored. If the refactoring concerns code maintained by a lot of different maintainers the process may take years to complete.

3.1 Challenges

One of the main challenges with proposing and monitoring refactoring efforts is the controlling. If the SIG decides on a new target architecture for an ILIAS component, this architecture should not only be accepted by the community but also be coherently reflected by the implementation. There needs to be some feedback for contributors to help them develop in the direction of the new target architecture. Controlling also concerns the SIG itself. The SIG needs to check if there is any progress in the refactoring efforts apart from just defining new target architectures on paper.

Once introduced, the architecture should be sustained. The effects of architectural erosion have to be minimized, otherwise any service that the SIG provides is not sustainable and may not be cost efficient. As architectural erosion cannot be completely avoided, an overview of how fast this erosion happens and on which parts of the code base it occurs will help the SIG to concentrate their refactoring efforts.

Furthermore refactoring efforts have to be justified to superiors as investments have to be made by different stakeholders. A reporting mechanism about the refactoring process in general needs to be defined. This will also increase credibility of the SIG Refactoring over time. If a reporting system exists, newly planned refactoring topics will more easily be accepted by superiors and the community.

Concerning the acceptance in the ILIAS community: Developers have to be encouraged to contribute towards the new target architecture. Involving users in decisions like what will be refactored and how it will be refactored is a key point for increasing acceptance.

3.2 Process Flaws

As mentioned in section 2.1, maintainers are responsible for the quality of their maintained component. This leads to coherent architecture within the components but an overall lack of consistency in cross section functionality (*e.g.,* the permissions check or the exception handling). Even though there are development guides describing architectural principles available on the project website, a survey concluded that contributors rarely use them during development (see appendix B.2) and that the documentation is not fully consistent with the actual implementation (see appendix B.1). It is not mentioned in the survey whether or not the developers even know about the existence of the documentation in the survey. We assume that every developer knows about it as it is the first reference for any ILIAS developer.

Decisions on whether functionality should be integrated into the ILIAS project are taken during the Jour Fixe. This decision is often based on whether the technical part conforms with the software architecture of ILIAS. The problem is that after the concept is accepted, there is little control on what is actually added to the code. There are "random searches" by some contributors giving feedback to other contributors but whether the concept is consistent with the code is not covered by the process. The effect of this issue is increased by the fact that the Jour Fixe has no proactive capabilities. The code added to the repository cannot easily be changed by the community or the Jour Fixe. Only the maintainers are responsible for it.

3.3 Architectural Flaws

The idea behind our work is to find some architectural flaws within the ILIAS source code and remove them. To support this process, we provide feedback about violations and help developers to migrate to a new architecture.

In a first attempt we tried to analyse some potential problems affecting parts of the ILIAS code base, (the draft can be found in the appendix in chapter A). In discussions with other ILIAS developers, we discovered that what we considered as problematic architectural flaws were less important issues for others. We recognized that we might not gain sufficient acceptance among the development community if we decided about what an architectural flaw was by ourself. To decide upon which aspects of the architecture had to be refactored we started a survey (see appendix B.1.2).

3.4 Estimated Effort

Specification maintenance costs The main cost of introducing architectural conformance checks will most likely be the maintenance of the architectural invariants. As soon as these are no longer perceived as reflecting the latest architecture, acceptance will decrease. This upkeep needs a structured process in addition to repeated manual work and is thus costly. Once architectural invariants are properly defined, the actual specification of those invariants for Dicto, our target conformance checking tool, is a relatively easy task. This is a clear benefit over other tools (e.g. Lattix's Dependency Manager), where the configuration of rules may require variable effort depending on the number of entities one has to deal with [8]. As the checks and the feedback are automated after the configuration with Dicto the cost is much lower than with manual approaches such as in "An Industrial Case Study of Architecture Conformance" [4].

Extend Dicto At the beginning of the project, Dicto did not support the analysis of PHP systems. The cost of integrating an existing tool into Dicto is relatively low (only an adapter has to be written for Dicto to be able to run these tools). If there is no suitable tool for this job, one has to develop an analyzer by oneself. Fortunately we were able to find a suitable PHP analyser. The cost-benefit analysis must be re-evaluated for programming languages where such a tool is not available. Alternatively we could refrain from the analysis of PHP specific systems and use the language independent tools. This would decrease the estimated effort but the amount of invariants that can be covered by Dicto is then significantly lower.

Implementation of the feedback system Dicto offers an API for declaring, checking and getting feedback on rules. In our project, we needed to implement a visual interface for showing these rules, their documentation and their violations. Furthermore, as we will discuss in section 3.5.2, we want to include information regarding the increase of violations occurring between two builds. This is not yet supported by Dicto and must be handled from outside the tool. Dicto does not offer any integration with continuous integration servers such as Jenkins[1], Team City[2], Sonarqube[3] etc. To support the integration with these platforms, a new plugin needs to be developed. Dicto could in a later stage offer plug-ins for the most popular continuous integration servers to eliminate this cost. If we decided not to implement the visual comparison between two builds, the cost of integration would decrease, but as this is considered to be one of the main benefits of the planned solution the cost-benefit factor would decrease.

Hardware setup and maintenance The server on which Dicto runs has to be set up requiring a unix based machine capable of running Pharo[4]. If not yet available some sort of continuous integration server has to be set in place to start the Dicto checks and to give feedback to developers. If already available the Dicto checks have to be integrated into this solution. These tasks are considered as one-time tasks and the costs are low.

3.5 Open Issues

3.5.1 Defining the invariants

There is a large set of available tools to check for architectural invariants within a code base. There is for example Yasca[5] to get reports on code-quality metrics or PHPDepend[6] to get reports on the dependency model. As a consequence, developers cannot be familiar with all of them in detail. Thus the definition of rules must happen at an abstract level, independent from specific tools. The definition of architectural invariants should be readable and understandable for as many contributors as possible without any prior knowledge of analysis tools. This leads to minimal training and communication costs and will also increase acceptance as more people may participate in the creation and maintenance process of the invariants.

As architecture is subject to change over time, the process of maintaining invariants must be defined. The discussion of the architectural invariants must be open to the whole ILIAS community to increase acceptance of these rules.

3.5.2 Feedback to contributors

The feedback about the architecture must be effortless to the contributor. If the contributor has to get information about the architectural invariants and their state by himself, a training and communication cost is added as developers have to be trained to obtain the feedback and to interpret it as well. Furthermore feedback has to be integrated directly into the development process. Feedback that is easily accessible to a developer and is delivered shortly after a contribution is more likely to be considered.

When introducing architectural invariants there may be a lot of violations to begin with. It is important that contributors will not be overwhelmed by the feedback. If the feedback reads something like “There are currently 2000 architectural violations in the code base”, the feedback will simply be ignored.

Given these requirements a continuous server giving automated feedback to the developers comes to mind. ILIAS has a first attempt for a continuous integration server. This is an important open issue and

[1]https://jenkins-ci.org/
[2]https://www.jetbrains.com/teamcity/
[3]http://www.sonarqube.org/
[4]http://pharo.org/
[5]http://yasca.org/
[6]http://pdepend.org/

thus documented in the following section 3.5.3.

3.5.3 Continuous Integration

A continuous integration server has been introduced in late 2013 but failed to convince contributors to use it on a regular basis (Unit tests were broken for up to six months without a fix). In a survey we asked what the reasons were for developers not to use the continuous integration server. Many participants mentioned that it was too complicated to set up, that the feedback cycle was badly integrated and that the community lacks interest in supporting CI (See appendix for the whole feedback B.1.4).

Furthermore we asked what features a continuous integration server should offer for the participants to actively use it. The answers included the following features: Better unit test policy and feedback on every commit.

Developers also asked for feedback on: Unit tests, code complexity checks, architectural checks, code smells, code formatting, syntax checks.

Based on the survey, one of the primary features developers want to see in a continuous integration server is the unit test results. A new start with a continuous integration server should thus certainly include the unit test results. Additionally the feedback cycle should be improved as this will increase the visibility and cover parts of the requirements of the survey, namely “Better feedback Cycle” and “Convince developers to use it”. In the previous approach developers had to actively go to the CI and could see the results of the last nightly build. A better feedback cycle would entail a build upon every contribution and a notification for the developer that the CI has new information of interest for him.

The fact that three developers mentioned architectural checks, shows that there is demand from within the ILIAS community to introduce architectural conformance checking.

Automated feedback is considered especially useful in ILIAS as a survey concluded that there are contributors whose code does not get reviewed or only rarely so. In a survey with 15 participants, 8 answered that their code gets rarely or never reviewed. In those cases where the code gets regularly reviewed, contributors rely on internal guidelines and experience to provide feedback. Automated checking of architectural invariants could help to test the quality of otherwise unreviewed code and could give some standardized feedback that does not rely on the service providers internal guidelines and experience. A summary of the results obtained from this last survey question can be found in Table 3.1 and Table 3.2.

Answer	Number of Answers
My code gets rarely/never reviewed	8
My code gets reviewed regularly	7
	15

Table 3.1: Survey results: Does your code get reviewed?

Review Based On	Number of Answers
Official ILIAS development guide only	1
Internal guidelines only	3
Only other references	2
Official ILIAS development guide and internal guidelines	4
Official ILIAS development guide and other references	1
Inernal guidelines and other references	2
No reviews are made	2
	15

Table 3.2: Survey results: If your code gets reviewed, what is the review based on?

3.6 Stakeholders

The most important stakeholder is the ILIAS community. The ILIAS community takes and communicates its decisions on the Jour Fixe. Thus we will represent the ILIAS community as a stakeholder by the Jour Fixe. The interests of the Jour Fixe align with the efforts of introducing architectural invariant checks as it aims to deliver a stable product and this can be promoted through a clear architecture[3, 5]. Furthermore the Jour Fixe mainly consists of developers, and architectural invariants help in the daily routine of those developers. On the other hand the Jour Fixe, again in the interest of the stability of its product, may oppose refactoring efforts that are too ambitious. Furthermore the refactoring topic and the architectural invariants must be chosen with the consent of the Jour Fixe, as it is the most influential stakeholder among the ILIAS community and any effort made without its consent will most likely be in vain. If the architectural conformance checking somehow lowers the speed of development, the JF may also oppose it.

The second stakeholder is the special interest group for refactoring, called the SIG Refactoring. The SIG Refactoring is the main force behind the decision and implementation of the refactoring topic. Its aim is to make the lives of ILIAS developers easier. The SIG itself has no decision power among the community and is thus dependent on decisions taken by the Jour Fixe.

The third stakeholder is the University of Bern. The main goal is to evaluate Dicto in order to gain some experience and analyse how it works in practice. ILIAS is particularly of interest as it is the first case study with Dicto about an open source software. We see ourself as service provider to the SIG Refactoring and the Jour Fixe. We help introducing architectural invariants and implement automated checks. Furthermore we participate in the SIG Refactoring to observe the refactoring process. The actual content of the proposed refactoring tasks is secondary, and will not be addressed in detail in this thesis.

A summary: The Jour Fixe represents the ILIAS community and oversees the ILIAS code base. The SIG Refactoring represents the refactoring efforts and the university of Bern helps the SIG Refactoring and the Jour Fixe by providing architectural conformance checking.

4 Solution Design for Architectural Monitoring

In our case study, we monitor changes to the project's architecture during refactoring efforts. While the refactoring is being applied, contributors will get feedback about the current state of the architecture. The architectural monitoring will be done using Dicto. Dicto is used to define architectural invariants, so called Dicto rules. The feedback the contributors receive are based on these architectural invariants. The roadmap is as follows:

(1) Establish a SIG for refactoring: The SIG Refactoring was established at the beginning of the case study, as mentioned in section 3.6 the SIG has a leading role in the refactoring effort. As usual in such a context, the more people that back up the idea the faster it is embraced. The SIG is founded during the first meeting described in section 4.1.

(2) Perform a survey for refactoring topics: It is important that the refactoring topic is not chosen only by members of the SIG. As we discovered in section 3.3, refactoring topics are very subjective and their benefits are not always recognized. To accept an architectural decision it is important that the effects deriving from its application help as many developers as possible. The results of the refactoring survey can be found in the appendix in section B.1.2.

(3) Decide upon a cross-section functionality in need of refactoring: The SIG will choose a suitable refactoring topic based on the outcome of the survey. We aim at something that is implementable within a moderate effort, but at the same time is a cross-section functionality. This means that the whole community has to be involved in the refactoring process. Defining the refactoring topic is described in chapter 4.1.

(4) Decide on a new target design after the refactoring: The SIG will then discuss a new target architecture for the refactoring topic. The target architecture has to be approved by the Jour Fixe, where the whole community can give input or veto the new architecture. The designed solution is described in section 4.1.2.

(5) Translate the architecture into Dicto rules: Based on the new target design we determine architectural invariants that describe the new target architecture. We also find some invariants that have to hold in the current architecture. These rules should not be broken during the transition. Secondly we determine some invariants that will hold in the target architecture but do not hold in the current architecture. The idea behind this is that you can consider the transition successful as soon as there are no more violations of these rules in the code base. Thirdly we determine some invariants that do not concern the refactoring. All these steps are documented in the section 4.2.

(6) Implement the analysis infrastructure to support automatic testability: Dicto already supports a lot of tools for checking architectural rules. During the definition of the Dicto rules in the previous step we discover that it is necessary to implement an additional adapter to Dicto. The implementation is documented in the tool chain section of the thesis (see 4.4).

(7) Hook the Dicto results into the continuous integration process of ILIAS: At the time no continuous integration (CI) server is actively used in the ILIAS community, the SIG will try to promote an existing CI service or help install a new one. The goal is to provide feedback to every contributor on each commit. This feedback cycle will be largely based on the Dicto results. They will contain the currently existing violations against the previously defined architectural invariants and information regarding where these occur in the code base. The feedback will highlight what violations have been added or removed compared to the previous build. This lowers training costs for the contributor as they are only shown the important parts of their contribution, as discussed in 3.5.2. The technical implementation can be found in section 4.4 and the solution in action can be found in section 4.5.

(8) Promote the refactoring of the code base according to the new architecture: The new architecture will be presented at the Jour Fixe of ILIAS, where it reaches most developers. This and the automated feedback of the continuous integration server should encourage the contributors to implement the new architecture or at least not add new violations of the architectural invariants.

(9) Evaluate the results: The continuous integration server automatically collects data about the amount of present, added and removed violations. Together with qualitative feedback from contributors we assess whether our approach to architectural conformance checking lowers the amount of architectural invariance violations. It will be of interest why some violations do not get fixed. Furthermore we will carefully analyse the added violations: some of them could be exceptions of the rule, some other may have been introduced because the committer was not aware of the architectural invariant or was not capable to solve the problem without breaking the architectural invariant. The evaluation is discussed in chapter 5.

4.1 Defining the Refactoring Topic

Before the first meeting of the special interest group we investigated the main pain points of developing in ILIAS through a survey. In the survey we asked the participants to describe the pain points encountered in practice. Each pain point was characterized by a name, a description and a design proposal for fixing it. In total we gathered 33 pain points . For details, see the appendix B.1.2.

All developers were then invited to the founders meeting of the special interest group for refactoring. Five people, including the head of development, attended the meeting; the minutes can be found in the appendix C.1. During the first part of the meeting the participants decided on a refactoring topic based on the survey's feedback. Each issue was discussed and put on a chart (see figure 4.1). Issues on the left were easy to implement and maintain. Issues on the right were hard to implement and maintain. Issues on top of the map were considered very useful in everyday development and issues on the bottom were considered not that useful.

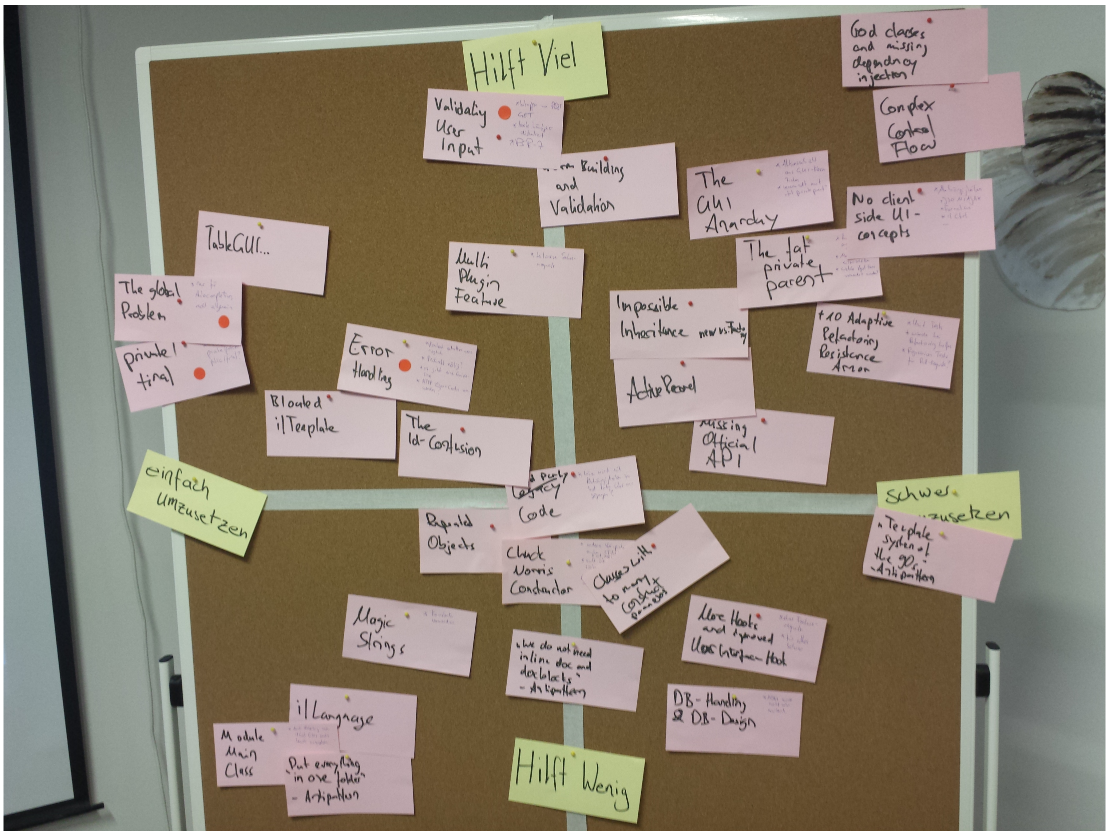


Figure 4.1: Board with the refactoring topics. The red dots on the cards mark the most interesting topics to refactor.

For the refactoring project, the SIG chose an issue on the upper left hand side: Something that was easy to implement but had a high impact on everyday development. The topic chosen was the exception and error handling of ILIAS.

4.1.1 Problem description

At the time there was no consistency in error and exception handling. PHP uses the global method set_exception_handler and set_error_handler in order to define what happens to uncaught exceptions and errors. But there are several libraries used in the ILIAS code which set these handlers themselves overriding the ILIAS default. Thus it was hard for developers to know what happens to an exception. Furthermore the ILIAS default was far from developer friendly; a caught exception would cause a browser redirect to an error page. This behaviour led to the loss of valuable debugging information (like the received headers or the current URL). This inconsistency has led to some cases where developers use the *exit* or *die* function in order to be sure about what happens. This is really hard for debugging as you have no idea where exactly the *exit* or *die* function call was triggered.

4.1.2 Design Solution

In the new concept[1] the following requirements are defined:

1. Exceptions should not leave the context under which they were thrown (ILIAS should not trigger a HTTP redirect to an error page but render the error in the current request).
2. It should be possible for developers to recover context information from an exception to make analysis of exceptional cases easier.
3. All exceptional cases in ILIAS should be handled via exceptions and the usage of PEAR[2] error handlers should be deprecated.
4. A top level exception handler that makes it possible to provide information for developers when the DEVMODE[3] is used.

Important to notice is that this requires the removal of all exit or die function calls as those circumvent the top level exception handler and abort the execution of the program at once.

This first topic seemed suitable for the SIG Refactoring and the remaining refactoring topics were put in the backlog. As the precondition to get a suitable topic out of the first meeting was met, the SIG Refactoring was officially founded.

The concept of the new Exception handler was accepted by the Jour Fixe. The top level exception handler was then implemented to work with Whoops by Richard Klees[4]. The implementation of ILIAS already covers some of the requirements mentioned in 4.1.2. There are two main problems that are not addressed by the current implementation: Firstly there are libraries which override the top level exception handler. If any of these libraries are loaded before the exception is thrown, then the wrong exception handler handles it. Secondly exceptions are often replaced by invocations to *exit* or *die*). This violates the new concept as no context information can be recovered after an *exit* or *die* call.

4.2 Rule definition

We want to introduce three types of architectural invariants: A group of rules that have nothing to do with the refactoring process but should hold for ILIAS in general. Some invariants that hold true in the current architecture and the intended architecture (they should already be defined in the ILIAS developers guide[5]). And finally some invariants that should hold true in the intended architecture but do not yet hold true.

The general invariants: In the general invariants we test PHP best practices and MVC pattern constraints. For the PHP best practices part we want to discourage the suppressing of errors. In PHP, one can suppress errors and warnings. This leaves the developer clueless in case something goes wrong and should be avoided. Furthermore we want to avoid that the contributors use the PHP function *eval*. The *eval* function executes arbitrary PHP code which can lead to security issues and furthermore is not supported by HHVM which is a targeted VM for ILIAS to run on. For the MVC part we want to discourage the usage of the database abstraction layer by Controller and View classes (named GUI classes in ILIAS). And we want only GUI classes to use the template engine, the class to add navigation tabs and the language utilities.

[1] `https://github.com/klees/ILIAS_SIG_Refactoring/tree/master/ErrorHandling`
[2] PEAR is a group of libraries, see `http://pear.php.net/`
[3] Developer mode to run ILIAS in see: `http://www.ilias.de/docu/goto_docu_pg_1082_42.html`
[4] `https://github.com/ILIAS-eLearning/ILIAS/pull/33`
[5] `http://www.ilias.de/docu/goto_docu_pg_29964_42.html`

Current refactoring invariants: In the development guide of ILIAS we found a constraint that every exception class should extend directly or indirectly the top level exception class ilException. Other than that there seem to be no other architecturally relevant restrictions defined in the development guide.

Target refactoring invariants: The first invariant is that the top level exception handler is not set by any class other than the newly implemented Whoops error and exception handler. The second invariant is that die, exit and trigger-error function calls are forbidden. Exceptions should be used instead. When these invariants hold the new top level exception handler should be able to catch all unhandled exceptions and act accordingly.

The Dicto rule set described above informally was established in three steps. In the first step we provided four rules as example rules. They were namely: `WholeIliasCodebase cannot depend on triggerError`, `WholeIliasCodebase cannot depend on exitOrDie`, `only GUIClasses can depend on ilLanguage` and `ilExceptionsWithoutTopLevelException can only depend on ilExceptions`. These rules were mainly designed to let the users familiarize themselves with the syntax without reading any abstract definitions.

In the second step the rules were open to discussion for the SIG. The first contributor was the main actor in getting ILIAS running on the HHVM. He was thus interested in preventing PHP patterns that would inhibit correct execution of ILIAS on HHVM. He introduced two rules: `WholeIliasCodebase cannot depend on eval` and `WholeIliasCodebase cannot depend on SuppressErrors`. It is important to notice that he was able to write the first rules while only having access to the examples given in step one. The second rule was acting on PHP operators (namely the `@` operator). It was not supported by Dicto tools at that point but the contributor could write the comment block describing the rule. A second contributor who was involved in implementing the new Error Handler added the rule: `WholeIliasCodebase cannot depend on SetErrorOrExceptionHandler`. However he did not write the rule himself but communicated its intent by mail.

In the third step, the rules were presented and discussed at the Jour Fixe. The feedback led to the introduction of the following rules: `GUIClasses cannot depend on ilDB`, `only GUIClasses can depend on ilTemplate` which are general MVC pattern rules. The rule `only GUIClasses can depend on ilLanguage` was rejected. Mail, notification and export functionalities should be able to access the language as well. This step takes into account the requirements outlined in section 3.5 making the definition process open to the whole community.

We observed that all the users involved in this process always added comments on why their rule was important in their opinion. This led to the implementation of a new language feature that enabled the specification of arbitrary documentation blocks in very early stages of the definition process.

While discussing the rules in the SIG a participant asked if there a way to define PHP functions. This led to the decision to specialize the generic variable type of `PhpDependency` into four sub-types: `PhpDependency`, `PhpClass`, `PhpFunction` and `PhpGlobal`. This also made the process of defining globals easier. In the first iteration globals had to be defined with: `PhpDependency with name:"GLOBAL//variableName"`. This essentially reflected the specification syntax of the tool working in the background. The new version: `PhpGlobal with name:"variableName"` is is more self-explanatory.

The same participant wanted a way to select occurences of the PHP `@` operator in the variable definition. This led to the extension of the analysis tool to support the detection of such operators. We decided against a variable type named `PhpOperator` as the `@` operator is most likely the only operator of interest in the architecture. Other operators (*e.g.,* arithmetic-, bit-, type-operators) are used far too often in a project to help in any analysis. Making a separate variable type for only one possible variable was considered as not beneficial to the simplicity of the tool.

The discussion at the Jour Fixe about the rules involved various people who did not have deeper knowledge of how Dicto works. The only resource they had were the proposed rules (similar to the final rules in section 4.2.1). Nevertheless they came up with two new rules and asked for one rule to be removed with reasonable arguments. This speaks very much for how intuitive the Dicto rules are to read and to write.

In section 3.4 we stated that we can decrease the cost of introducing Dicto to ILIAS by not implementing PHP specific tools for Dicto. The fact that we discussed almost exclusively invariants that include restrictions to the dependency model, suggests that this decreased cost would sacrifice a lot of benefit. According to this discussion we recognized that when introducing Dicto into a project, the programming language of such project should already be supported by Dicto. If that's not the case, the cost of implementing adapters of tools for this programming language must be acceptable. In this case study the implementation was done by ourself thus the cost did not matter for the ILIAS community.

4.2.1 Specification

The rule-set for ILIAS defined by the refactoring SIG is shown below.

```
ilClasses = PhpClass with name:"il*"
assClasses = PhpClass with name:"ass*"
WholeIliasCodebase = {ilClasses, assClasses}
GUIClasses = PhpClass with name:"*GUI*"
triggerError = PhpFunction with name:"trigger_error"
exitOrDie = PhpFunction with name:"exit/die"
eval = PhpFunction with name:"eval"
ilTopLevelException = PhpClass with name:"ilException"
ilExceptions = PhpClass with name:"il*Exception*"
ilExceptionsWithoutTopLevelException = {ilExceptions} except {
    ↪ ilTopLevelException}
SuppressErrors = PhpDependency with name:"@"
ilDBClass = PhpClass with name:"ilDB"
ilDBGlobal = PhpGlobal with name:"ilDB"
ilTemplateClass = PhpClass with name:"ilTemplate"
ilTemplateGlobal = PhpGlobal with name:"tpl"
ilTabsClass = PhpClass with name:"ilTabsGUI"
ilTabsGlobal = PhpGlobal with name:"ilTabs"
SetErrorHandler = PhpFunction with name:"set_error_handler"
SetExceptionHandler = PhpFunction with name:"set_exception_handler"
SetErrorOrExceptionHandler = {SetExceptionHandler, SetErrorHandler}

/**
 * The global php function trigger_error is a procedural concept. Please
    ↪ ommit this php function and use an ILIAS exception instead.
 */
WholeIliasCodebase cannot invoke triggerError

/**
 * Exit and die are a bad idea in both development and production: In
    ↪ development you have no idea what went wrong and in production the user
    ↪  receives a white page and has no idea whats going on. The implemented
    ↪ exception handling does not work if you use exit or die.
 *
 * If you want to send a file consider using: Services/FileDelivery.
 *
 * Exception: Currently if you want to output JSON you most likely have to
   ↪ use exit() at the moment.
 */
WholeIliasCodebase cannot invoke exitOrDie

/**
 * The error and exception handler of ILIAS should not be overridden.
 */
WholeIliasCodebase cannot invoke SetErrorOrExceptionHandler

/**
 * The php function eval() is not good practice. Its use often comes with a
   ↪ high security risk, because it is generally not a trivial task to make
   ↪ sure that a paramater of eval() can be fully trusted. And if it is,
   ↪ then eval() is usually not neccessary. It is also tricky to debug,
   ↪ because it obfuscates control flow. Last but not least, it does not
   ↪ work with HHVM in the special "RepoAuthoritative" mode, which makes PHP
   ↪  run extra-fast.
 */
 WholeIliasCodebase cannot invoke eval


/**
 * Silencing errors with the @ operator is bad practice. It makes code
   ↪ uneccessarily harder to debug if the currently suppressed error changes
   ↪  into a real show-stopper bug. Try to handle the possible warnings and
   ↪ errors.
 */
WholeIliasCodebase cannot depend on SuppressErrors

/**
 * All ILIAS Exceptions must be in a Hierarchy and finally extend ilException
   ↪  Every module/service should define its own top level Exception e.g.
   ↪ ilCourseException where all other exceptions from that module/service
   ↪ extend this service/module Exception.
 *
 * See: http://www.ilias.de/docu/goto_docu_pg_42740_42.html
 */
ilExceptionsWithoutTopLevelException can only depend on ilExceptions

/**
 * The GUI-Layer should not itself interact with the database. Try to build
   ↪ reusable Model classes, adding a layer of abstraction instead of
   ↪ accessing the database.
 */
GUIClasses cannot depend on ilDBClass

/**
 * The GUI-Layer should not itself interact with the database. Try to build
   ↪ reusable Model classes, adding a layer of abstraction instead of
   ↪ accessing the database.
 */
GUIClasses cannot depend on ilDBGlobal
```

```
/**
 * Only the GUI-Layer should use the global variable ilTabs and the class
   ↪ ilTabsGUI. If you use them in a Model the model cannot be used for e.g.
   ↪  SOAP requests without unnecessary overhead.
 */
only GUIClasses can depend on ilTabsClass

/**
 * Only the GUI-Layer should use the global variable ilTabs and the class
   ↪ ilTabsGUI. If you use them in a Model the model cannot be used for e.g.
   ↪  SOAP requests without unnecessary overhead.
 */
only GUIClasses can depend on ilTabsGlobal

/**
 * Only the GUI-Layer should use the global variable ilTemplate and the class
   ↪  ilTemplate itself. If you use ilTemplate in the model it cannot be
   ↪ used by calls that do not initiate global ilTemplate for example SOAP.
 */
only GUIClasses can depend on ilTemplateClass

/**
 * Only the GUI-Layer should use the global variable ilTemplate and the class
   ↪  ilTemplate itself. If you use ilTemplate in the model it cannot be
   ↪ used by calls that do not initiate global ilTemplate for example SOAP.
 */
only GUIClasses can depend on ilTemplateGlobal
```

4.2.2 First Results & Classification

A first evaluation of the rules we defined against the whole ILIAS code base is presented in the table 4.2.2.

Rule	*Violations*
GUIClasses cannot depend on ilDBClass	12
only GUIClasses can depend on ilTemplateClass	66
only GUIClasses can depend on ilTabsGlobal	5
WholeIliasCodebase cannot invoke triggerError	6
WholeIliasCodebase cannot invoke eval	3
WholeIliasCodebase cannot invoke exitOrDie	227
WholeIliasCodebase cannot depend on SuppressErrors	203
ilExceptionsWithoutTopLevelException can only depend on ilExceptions	7
WholeIliasCodebase cannot invoke SetErrorOrExceptionHandler	4
only GUIClasses can depend on ilTemplateGlobal	35
GUIClasses cannot depend on ilDBGlobal	33
only GUIClasses can depend on ilTabsClass	5
	606

There are nine rules that describe architectural invariants throughout the project. As mentioned above they are designed to ensure PHP best practices and MVC pattern constraints. These are the general invariants.

1. GUIClasses cannot depend on ilDBClass
2. GUIClasses cannot depend on ilDBGlobal
3. only GUIClasses can depend on ilTemplateClass
4. only GUIClasses can depend on ilTabsGlobal
5. WholeIliasCodebase cannot invoke eval
6. WholeIliasCodebase cannot depend on SuppressErrors
7. only GUIClasses can depend on ilTemplateGlobal
8. only GUIClasses can depend on ilTabsClass
9. only GUIClasses can depend on ilTabsGlobal

There is one rule that describes an architectural invariant concerning the exception handling. This rule should hold before and after the refactoring process. We refer to these rules as current refactoring invariants.

1. ilExceptionsWithoutTopLevelException can only depend on ilExceptions

There are three rules that describe architectural constraints that should hold after the refactoring process. The aim of these constraints is to get rid of previously established exception and error handling policies and to get rid of all function calls that override the default error and exception handler. We call them the target architecture invariants.

1. WholeIliasCodebase cannot depend on triggerError
2. WholeIliasCodebase cannot depend on exitOrDie
3. WholeIliasCodebase cannot depend on SetErrorOrExceptionHandler

4.3 Continuous Integration

The next step in the case study is to set up a continuous integration server for the ILIAS community. The goal is to give the contributors feedback on each check-in made to the repository. If we look back at the survey that was collected at the beginning, described in section 3.5.3, we see that unit tests are an important part for most of the developers. Thus we should deliver the results of a unit test run together with the architectural tests to gain higher acceptance of contributors.

The second meeting of the SIG Refactoring was held at the ILIAS Development Conference in Bern with the main focus of introducing a new continuous integration process[6]. The points discussed at the meeting concerning the CI-Server were the following:

1. There should be a central ILIAS-CI-Server that monitors the official ILIAS repository.
2. The results of the CI should be examined collectively in frequent intervals.
3. An overview over the results of the CI should be sent to all ILIAS devs by e-mail in frequent intervals.
4. After each commit the results of the CI should be send to the committing developer in an email.
5. The CI should be enhanced by the implementation of Dicto.

The full protocol can be found in the appendix C.2

Furthermore the process for introducing new Dicto rules and documentation about those rules was discussed. The only authority that can make a final decision on ILIAS processes is the Jour Fixe, thus it is important that the rules and processes for the maintenance the rules are discussed there. At the same time the Jour Fixe already has a tight schedule. The proposed process defines the SIG Refactoring as a first instance for rule proposals. The SIG will collect, filter and bundle the proposals and suggest them to the Jour Fixe. This lessens the workload on the Jour Fixe while still keeping final decisions and veto opportunities at the Jour Fixe. In a later step the SIG Refactoring would like to suggest documentation and guideline changes in the ILIAS developers guide that go along with the Dicto rules. As recorded in the minutes of the meeting:

1. The SIG Refactoring will propose rules for the CI to the Jour Fixe.
2. The SIG Refactoring will serve as an instance for other people to propose new rules for the CI.
3. In the long run the SIG Refactoring likes to improve the documentation of existing guidelines and rules for all ILIAS developers.

4.4 Toolchain

The central unit in the tool chain coordination is the continuous integration software TeamCity[7]. TeamCity allows developers to configure a build chain that is triggered as soon as a new commit is made to the central GitHub repository. Upon this commit the whole project in its latest form gets downloaded locally. Thereafter three scripts are triggered.

The first one uses the Dicto command line interface[8] to: Create a Dicto test suite, define the rules found in the downloaded project, generate the results, save the results to a JSON file and create a visual

[6]http://www.ilias.de/docu/goto_docu_fold_4515.html
[7]https://www.jetbrains.com/teamcity/
[8]https://github.com/otruffer/DictoCLI

representation of the results in an HTML page. In this script the JSON results from the previous build are processed to create a visual representation of the results including a comparison to the previous results. The HTML page created by DictoCLI is returned to TeamCity and will be displayed in a tab in the results page of the build(see figure 4.4). Furthermore the script delivers statistical data about added, resolved and present violations back to TeamCity.

The second step is to run the unit tests of ILIAS and display the results in TeamCity. The script we used was described on the JetBrains TeamCity Blog[9]. We decided against collecting the test coverage, since the test coverage is very low. This means we get little information out of it and the build time increases significantly.

The last script collects all contributors which were involved in the latest changes and sends e-mail notifications to them with a short summary about the added or resolved Dicto violations. If any violation was resolved, all contributors involved in the latest changes get one point per resolved violation. This is later used to have a leader board to keep track of the contributors with the most resolved Dicto violations. There are no minus points for added violations, as we want to use positive reinforcement only.

All the statistics that we produce in the first script are collected by TeamCity and are later used to display graphs about the trends of the Dicto violations. TeamCity and Dicto are currently set up on the same physical machine. The communication with Dicto is done over its REST API. Theoretically these two services can be located on different machines if we chose to distribute them for performance reasons. It is important though that the source code is somehow available on the server running Dicto.

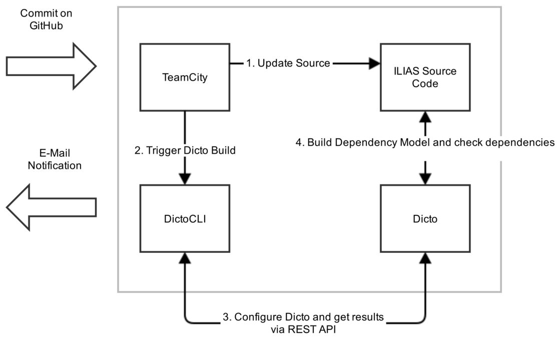


Figure 4.2: Event sequence following upon a contribution on GitHub. The container box symbolizes that in the current setup all services are installed on one physical machine.

As described in section 4.2, it is vital to integrate tools into Dicto which can operate on the Dependency Model of the PHP source code. Dicto already has off-the-shelf integration for Moose[10]. Moose itself is theoretically language agnostic using the FAMIX meta-model. Unfortunately approaches to parse PHP Code into the FAMIX meta-model have proven to be difficult and time consuming[11]. We decided thus to use another open-source tool and implement an adapter in Dicto for this tool. This was less time consuming but also kept us from reaching the complexity of checks you can implement with Moose.

[9]`http://blog.jetbrains.com/teamcity/2013/07/first-class-php-continuous-integration-using-teamcity/`
[10]`http://www.moosetechnology.org`
[11]`http://scg.unibe.ch/archive/projects/Ruef13a-PHP.pdf`

Our decision was to integrate PhpDependencyAnalysis by Marco Muths[12]. The project is actively maintained and delivers a solid dependency model (including globals, PHP internal functions, etc). There were only two small modifications we had to apply: A fix that prevented the analyzer from crashing on malformed PHPDocBlocks[13] and a JSON export[14]. The tool can be trusted as it passed all manual tests and the unit tests coverage as well as the code quality are excellent. The output we obtain from this tool is a JSON formatted file, describing all dependencies between code units. Units are: PHP Globals, PHP Functions and PHP Classes. Furthermore we get a description of the file and the line number in which this dependency was first detected. An analysis of the whole ILIAS code base with the PHP interpreter took on a local machine a total of 20 minutes. We decided to run the script using the HipHop Virtual Machine[15](HHVM), which required a quick fix in the analyzer to be able to run it in HHVM[16]. With HHVM running the script, the execution time went down to about 3 minutes. The dependency model of ILIAS in JSON format has a size of about 20 megabytes, a big part of these 20 megabytes consisting of information describing the locations of the dependencies.

The adapter implementation of Dicto for the PHP analyzer is pretty straightforward. Dicto has two main steps that every adapter has to implement. The first step is to resolve the user-declared variables into specific code elements and in the second step the adapter needs to interpret the rules declared by the user on these concrete elements. We will discuss the implementation of the variable types and the variable dependencies on an example. Consider a Project with the Classes: ilBlog, ilDatabase, ilBlogGUI, ilBlogEntryGUI and the Dicto rules:

```
GUIClasses = PHPClass with name:"il*GUI"
Database = PHPClass with name:"ilDatabase"

GUIClasses cannot depend on Database
```

Resolve the variables to elements: Dicto parses the rules and finds a variable definition of type PHPClass with the attribute `name` of value `il*GUI` in the first line of the rule set. It identifies the adapter responsible for PHPClass typed variables and requests all elements with matching attributes. The adapter runs the PHP Analyzer saving the model in json format to a file. The adapter then reads the model into the cache. The adapter tries to run the analyzer first in HHVM and as a fallback uses the standard PHP interpreter installed on the machine. After loading the model, Dicto can find the matching elements in the model. The same process is executed to resolve the variable with attribute `name` and value `ilDatabase` using the cached model. The first request will result in a list of elements for the variable GUIClasses, namely: `ilBlogGUI` and `ilBlogEntryGUI`. The second request will result in a single element: `ilDatabase`.

Evaluate the rules: Dicto parses the first and only rule: `GUIClasses cannot depend on Database`. This rule will be converted into two predicates: `ilBlogGUI cannot depend on ilDatabase` and `ilBlogEntryGUI cannot depend on ilDatabase`.

The adapter for the verb `depend on` with the subject of type `PHPClass` and one object of type `PHPClass` is resolved. The adapter is then called once for every predicate to decide whether this predicate holds or not. The adapter looks up the two entities in the model and sees whether or not a dependency between the two exists or not. Let's say ilBlogsGUI uses the Database. The adapter will answer that the dependency exists and point to the file and line where the dependency occurs. Dicto will decide, due to the modifier `cannot`, that the rule does not hold and treat the dependency from ilBlogsGUI to the ilDatabase as a violation to the rule.

[12]https://github.com/mamuz/PhpDependencyAnalysis
[13]https://github.com/mamuz/PhpDependencyAnalysis/pull/1
[14]https://github.com/mamuz/PhpDependencyAnalysis/pull/2
[15]http://hhvm.com/
[16]https://github.com/mamuz/PhpDependencyAnalysis/issues/3

In our case study, we added several new variable types to Dicto. These new types were namely: PHPDependency, PHPClass, PHPFunction and PHPGlobal. Furthermore we added the following rule keywords: `depend on` and `invoke`. The subject variable (in our case GUIClasses) for the keyword `depend on` must be of type PHPDependency, PhpClass, PhpFunction or PHPGlobal. The object variable (in our case Database) can be of type PHPDependency, PhpClass or PhpGlobal. The keyword `invoke` can operate on the same subject variables but only on object variables of type PhpFunction.

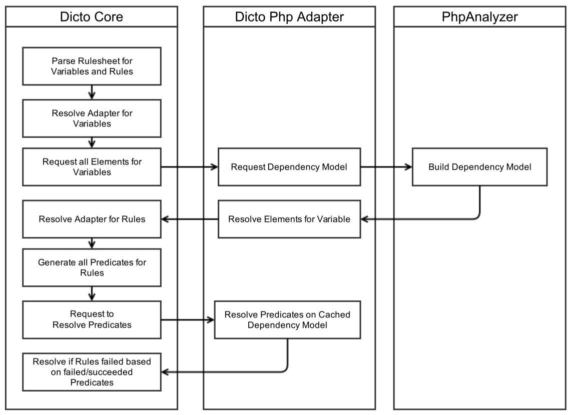

Dicto Core
Dicto Php Adapter
PhpAnalyzer
Parse Rulesheet for Variables and Rules
Resolve Adapter for Variables
Request all Elements for Variables
Request Dependency Model
Build Dependency Model
Resolve Adapter for Rules
Resolve Elements for Variable
Generate all Predicates for Rules
Request to Resolve Predicates
Resolve Predicates on Cached Dependency Model
Resolve if Rules failed based on failed/succeeded Predicates

Figure 4.3: Rule resolution with Dicto and the Dicto Adapter for a PHP Analyzer.

4.5 The Solution in Action

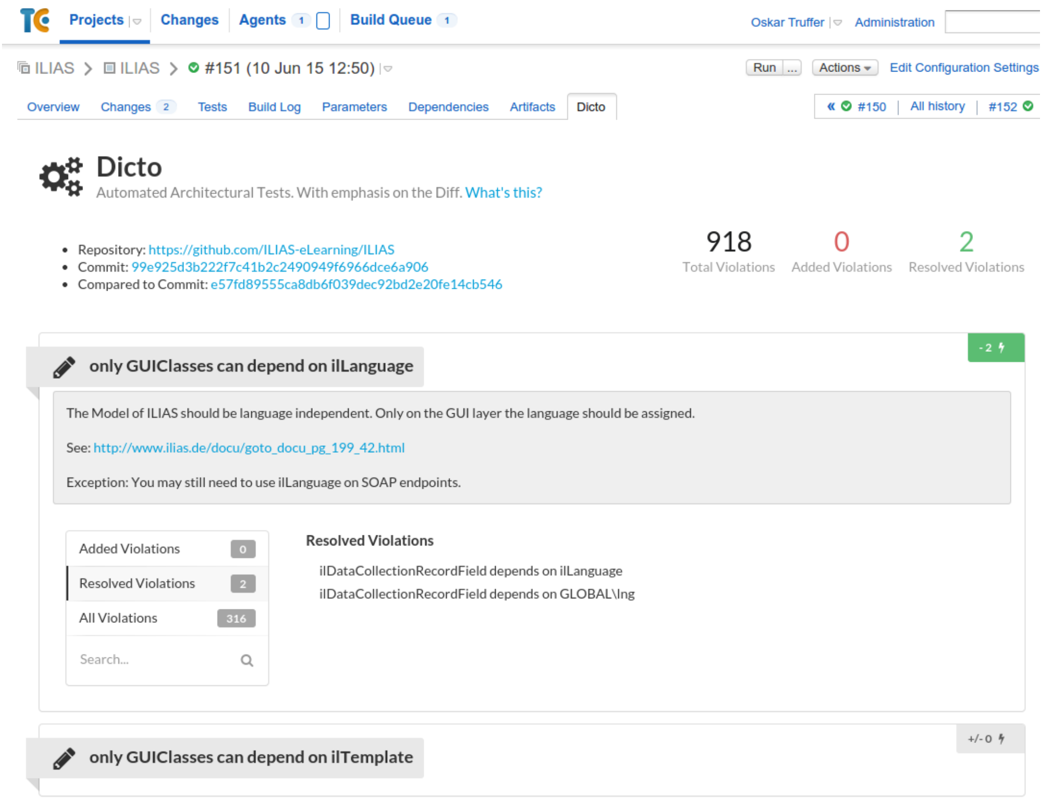
There are two main use cases in our solution.

The first use case is quite simple. With every contribution to the central git repository of ILIAS the associated contributor gets a brief E-Mail summary about his commit. This email includes a link to the full build that can be followed to get an insight in unit test status and the Dicto results.

The e-mail (reported below) to the contributor is sent about 3 minutes after the commit. We worked hard to improve the performance as the feedback should be as immediate as possible, as described in 3.5.2.

```
Dear ILIAS Contributor

The current build on our TeamCity-Server found your E-Mail address among the
    ↪ contributors. Have a look at the complete build:

http://ci.ilias.de/viewLog.html?buildId=214&buildTypeId=Ilias_ILIAS

Dicto
Added Violations: 0
Resolved Violations: 2
Total Violations: 606
The comparison was made between git commits
    ↪ df55436d69bc88dad64fc7f02de44296832bac66 and 95
    ↪ fd4f189f53b5fbd323fdda94bb92350089d342.

If you have any feedback or suggestion for Dicto rules or the CI in general
    ↪ feel free to send an email to ot at studer-raimann dot ch.

Cheers & Happy Programming
TeamCity
```

By clicking the link in the e-mail the contributor can navigate to the build. The most important tabs are: "Tests" and "Dicto". These show the result of the unit tests and the Dicto violations, see 4.4.



Dicto

Automated Architectural Tests. With emphasis on the Diff. What's this?

- Repository: https://github.com/ILIAS-eLearning/ILIAS
- Commit: 99e925d3b222f7c41b2c2490949f6966dce6a906
- Compared to Commit: e57fd89555ca8db6f039dec92bd2e20fe14cb546

918 Total Violations | 0 Added Violations | 2 Resolved Violations

only GUIClasses can depend on ilLanguage

The Model of ILIAS should be language independent. Only on the GUI layer the language should be assigned.

See: http://www.ilias.de/docu/goto_docu_pg_199_42.html

Exception: You may still need to use ilLanguage on SOAP endpoints.

Added Violations	0
Resolved Violations	2
All Violations	316

Resolved Violations

ilDataCollectionRecordField depends on ilLanguage
ilDataCollectionRecordField depends on GLOBAL\lng

only GUIClasses can depend on ilTemplate

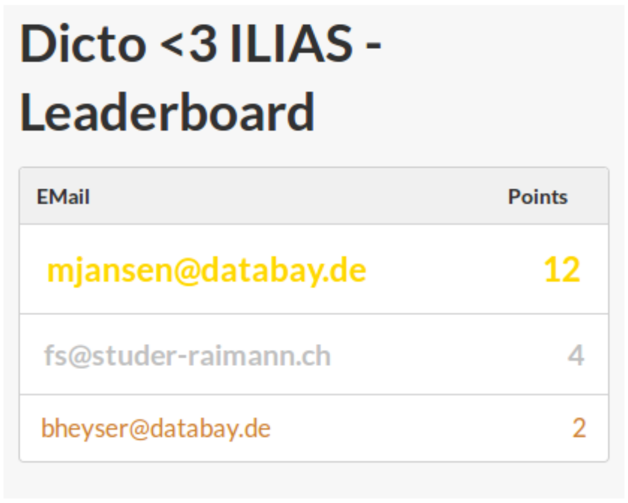
Figure 4.4: A Dicto test run on TeamCity.

The second use case is the weekly build. The output will look almost identical to the one shown in the previous use case with the only difference being that the added and resolved violations are calculated over the last week and not the last commit.

The idea behind this weekly build is to present an overview of what changed in the last week in the architecture of ILIAS to the Jour Fixe. The JourFixe can discuss current changes to the architecture based on these builds.

Thus every added violation is reported two times: First when the contributor adds it and secondly, if it's not fixed in the meantime, during the weekly build. This will help the Jour Fixe to have some control over the added source code after a concept is accepted and the implementation starts.

To give some additional motivation to fix rules, a leader board of ILIAS contributors has been introduced [17] (Figure 4.5). If a commit resolves some violations, contributors associated with that commit get one point per resolved violation. There are no minus points for added violations as we only want to give positive reinforcement. The leader board was implemented “quick and dirty” and originally just for fun but turned out to be a motivational boost for some contributors and was well worth the time. It was useful also for advertisement purposes. We promised that the contributor with the most points will get a box of chocolate and a hand made trophy at the next ILIAS Development Conference. This helped to increase acceptance and awareness of Dicto as a whole.



Dicto <3 ILIAS - Leaderboard

EMail	Points
mjansen@databay.de	12
fs@studer-raimann.ch	4
bheyser@databay.de	2

Figure 4.5: The Dicto - ILIAS leader board on 4. August 2015.

4.6 Maintenance

After this case study, we plan to continue maintaining and expanding the Dicto rules for ILIAS. We chose to discuss the topic on the second SIG meeting, documented in section 4.3. The SIG will be the main instance to maintain the rules but will need approval of the Jour Fixe to change the rules in the central repository. This ensures members of the ILIAS community not participating in the SIG stay informed and discuss changes to the rule set. As long as the SIG stays active, the maintenance of the rules is secured.

It is not yet discussed what happens if a new adaptor is needed to check for architectural invariants that cannot yet be checked by Dicto. The implementation of new adaptors is depending strongly on the tool that needs to be integrated. The cost-benefit analysis has to be dealt with individually. Furthermore the adaptors need to be written in the programming language Smalltalk. The ILIAS community in general has no experience in Smalltalk, which may lead to increased development cost for the implementation of new adaptors.

The rules are constantly changing and expanding. The current rule set can be found on our CI[18] or the Github repository[19].

We are still in the process of deciding who will maintain the ILIAS CI Server on the operating system’s side and who will be configuring the continuous integration system when needed. The Jour Fixe suggested to introduce an employee of the ILIAS society. Furthermore there are at the moment three

[17] The leaderboard can be found on `http://ci.ilias.de/DictoStats`
[18] `http://ci.ilias.de` – login as guest
[19] `https://github.com/ILIAS-eLearning/ILIAS/blob/trunk/dicto/rules`

people contributing to Dicto and its rule set in order to spread the knowledge of maintaining Dicto. These were decided upon on the last SIG Refactoring meeting and include:

1. Participant 1: Integrates PHP/HHVM-Version checks in the CI.
2. Participant 1: integrates jslint in the CI.
3. Participant 2: writes Dicto-Rules to check for inline css.
4. Participant 3: writes Dicto rules to disallow raiseError.
5. Participant 3: introduce check for inline styles

The full minutes of the third meeting can be found in the appendixC.3.

The process defined to change the rule set was defined in the second SIG meeting and is as follows: The SIG will act as a filter for requested changes in front of the Jour Fixe to decrease the workload for the Jour Fixe. Contributors can suggest changes to the rule set in the SIG Refactoring section on the ILIAS homepage[20]. The rules are discussed and a proposal is made for the Jour Fixe. It's important to notice that the SIG has no means to forbid anyone to suggest their proposals to change the set of rules directly to the Jour Fixe. This is not seen as a flaw in the design: If someone is discontented with the discussion and the proposal of the SIG Refactoring, he or she can propose an alternative solution directly to the Jour Fixe.

As the SIG is a more specialized group concerning Dicto and consists of fewer people, this approach should lead to a decrease in cost of the maintaining process (as discussed in section 3.4).

[20]`http://www.ilias.de/docu/goto_docu_grp_4497.html`

5 Evaluation

We will discuss some social aspects regarding Dicto's introduction to the ILIAS community as well as more technical aspects concerning the integration of the CI in the development cycle. The evaluation is based on Jour Fixe meetings, the contributors' survey (see appendix B.2), the third SIG Refactoring meeting (see minutes in the appendix C.3), inspected violations and statistics of the continuous integration server.

5.1 Process

The initial approach adopted in the case study was to decide on a refactoring project in ILIAS and monitor the resulting rules to supervise the refactoring process. Soon we recognized that the refactoring topic only gets implemented by very few developers and that the interesting discussions always ended up considering the whole ILIAS architecture.

The reactions on the first SIG meeting were rather conservative concerning Dicto. Some argued that Dicto was not in use anywhere and thus not reliable. On every step of the introduction the attitude towards Dicto improved. It was very important for the contributors and the Jour Fixe to see that they are in control of the rules, that they could understand and define the rules themselves. We continuously reported on the status of the violations and the development of the user interface on the Jour Fixe and the SIG meetings. The feedback about additional rules grew every time. On the last SIG Refactoring meeting a lot of changes were discussed with brainstorming on how to reflect them in Dicto rules. The rules in the pipeline from that meeting can be found in section 4.6.

A critical part that was in the interest of the SIG Refactoring is that the Jour Fixe uses the weekly build of TeamCity and Dicto to discuss current architectural progression. This was mainly to increase the visibility of the CI and to create a collaborative feeling about the architecture. Whether this will be done continuously in the future is not certain yet. As this process needs proactive action from the Jour Fixe, contrary to the reactive behavior induced by the message from the CI upon contribution, this may be overlooked in the future as the time allocated to the Jour Fixe is already very limited.

The social equivalent to a god class in software architecture is the Jour Fixe in ILIAS processes. Every action taken needs to be approved by the Jour Fixe and, as the Jour Fixe is often overloaded by tasks, actions might get delayed if not related to urgent tasks. We were very lucky that, when physically present at a Jour Fixe, we were allowed to present and discuss CI/Dicto related topics even though they may have not been on the top of the agenda.

Every developer contacted through the meetings or surveys states that he is strongly in favor of architectural checks and the CI. Yet when it comes to proactive actions most people do not seem to have the time to contribute effort. Thus the proactive elements of the CI are very important, the CI has to start the interaction and the process.

The notification system we set in place seemed to work well as all participants of the second survey said that they received the notifications as expected and in a timely manner (see appendix B.2). During discussions at a Jour Fixe, a developer mentioned that he marked the e-mails as Spam as there were too many. To make our notifications more relevant, we decided to send the e-mails only if any changes in the Dicto violations were detected. In general the feedback was easily readable, according to the responses collected in our survey (see appendix B.2), and the comments were helpful. There were some issues with the consistency of the feedback regarding the number of violation. This will be discussed in section (5.3.2).

5.2 Language expressivity

Because of the readability of the rules, it was very easy to discuss them with any developer even if nobody had ever read a word of the documentation. Especially notable is that the rules could be discussed even when none of the Dicto developers were present, as mentioned in section 4.2. Furthermore rules could be defined by one participant without access to the documentation but only having access to reference example rules. This really speaks for the expressivity of the language. Problems occurred as soon as a rule did not work as intended. In fact the current error messages are somewhat poor. This last point will be discussed in section 5.3.5.

5.2.1 Exceptions to rules

At the start of the project we discussed on how exceptions to rules should be handled. Up to now no need for special exception handling was found. This is not due to the reason that no exceptions exist but as the feedback focuses on the delta of violations, exceptions don't show up prominently after the initial commit. And on the initial commit the highlighting is necessary as the exception has to be discussed. If you want to get rid of exceptions in the report, in most cases it's sufficient to adapt the variable definition in order to exclude certain elements. The conclusion is that exceptions to rules don't matter as much as we initially suspected. There was only one request for an exception and it could be handled via alterations in the definition of variables.

5.2.2 Undefinable rules

On a Jour Fixe somebody proposed a rule for checking the code compatibility with a specific PHP version. The rulewould have looked like this:

```
WholeIliasCodeBase must be compatible with "PHP5.3"
```

This would require an additional adapter capable of parsing PHP Code and deciding whether all used function calls, operators, etc. are compatible with the given PHP Version. Unfortunately there seems to be no suitable tool to do a reliable check. Dicto is only as powerful as the tools it relies on. The development of such a tool is out of scope for Dicto.

5.2.3 Limitation of Dicto's modifiers

In the process of defining a rule that reflected some invariant written in the ILIAS development guide, we found the following rule: Every component of ILIAS needs a top level exception extending the class `ilException`. For example, the component `Course` must have a class named `ilCourseException`. Furthermore every exception thrown by this component must inherit in some way from this top level exception. Every exception thrown in the component `Course` must inherit from `ilCourseException`. In other words all exceptions in ILIAS are ordered in a hierarchical manner with the top node being `ilException`.

Translating this invariant written in the development guide into a Dicto rule exposed some problems. First off: The PHP analyser used does not support the notion of inheritance. An inheritance is simply recorded as another dependency. However, this is a weakness of the analysis tool and not the Dicto language itself.

The invariant was then implemented as follows. There is the variable `ilException` containing only the top level Exception `ilException`. The second variable is `ilExceptions` containing all exception of the name `il*Exception`. The third variable is `ilExceptionsWithoutTopLevelException` containing elements of `ilExceptions` without the top level class `ilException`. The first version of the rule then was:

```
ilExceptionsWithoutTopLevelException must depend on ilExceptions
```

The problem with the rule was: It now expects any ilException except the top level exception to depend on *every* other ilException. This caused a large amount of violations. The intention was to make every ilException without the top level exception to depend on at least one ilException. There was, at the time, no such modifier to describe such a rule. Thus we implemented an additional modifier: "[subject] must [verb] any [object(s)]". In our case the final rule, was changed to:

`ilExceptionsWithoutTopLevelException` *`must`* `depend on` *`any`* `ilExceptions`[1]

What this shows in general is that not all modifications of the verb can be expressed in the language. For example: If we wanted a class depending on at least two other classes we cannot express this. In practice this seems to occur rather rarely. There was no other rule where the modifiers were insufficient.

5.2.4 No mix of variable types

We consider the invariant describing that the database class (`ilDatabase`) should not be used in the GUI layer of ILIAS. Ideally we want to describe this invariant in one Dicto rule. Now the problem in practice is that the database class of ILIAS can be accessed in three ways: Make an instance of it, receive it as a parameter through the constructor or a method, or accessing the global variable `ilDB`. The first two cases can be handled by describing the variable `ilDatabaseClass` of type PhpClass, containing only the class `ilDatabase` and then introducing the rule: `GUIClasses cannot depend on ilDatabaseClass`. In the third case we define the variable `ilDatabaseGlobal` of type PhpGlobal, containing only the global `ilDB`. Again we define almost the identical rule: `GUIClasses cannot depend on ilDatabaseGlobal`. Dicto allows one make a union of two variables, which would allow us to introduce only one Dicto rule matching the invariant. The limitation is that Dicto does not allow the union of two variables with different types thus we need two Dicto rules to match the invariant. This is not a deal breaker but decreases the readability of both the rule sheet and the continuous integration

[1]It's planned to rename the verb in order for the rules to be: ilExceptionsWithoutTopLevelException must depend on at least one ilExceptions

output. The rule sheet just contains more rules with sometime duplicated descriptions and the CI feedback often adds two violations when a GUI class uses the database global as well as the database class.

5.3 Solution Report

5.3.1 Acceptance in the community

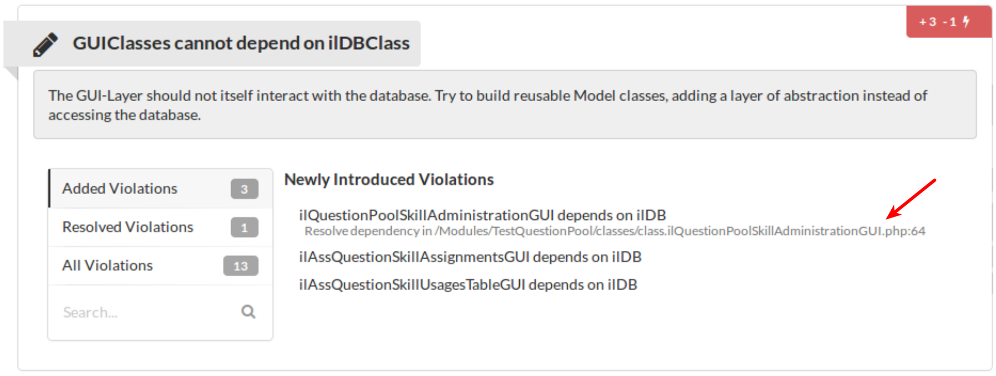
It definitely leverages the discussion about architecture and separation of concerns.
- Contributor 4

The overall feeling about the integration of Dicto into the ILIAS community was very positive. When Dicto was discussed at the Jour Fixe, participants always suggested new rules and also helped to improve the feedback cycle. Discussions in the Jour Fixe and the SIG Refactoring meetings also suggest that Dicto will be used after the case study is finished. Furthermore a survey of the contributors during the monitoring phase indicated that the rules are readable and sensible.

5.3.2 Consistency

The participants in the survey concluded that the feedback of Dicto was consistent with the expected effects of their changes in the code. The only reported changes in violations which were not understood by participants of the survey were due to the CI comparing builds that weren't related. More on that later in this chapter.

There was one issue with the expected results, discovered in a discussion prior to the survey: The issue was that the participant removed an "exit" function call from a class and thus expected a violation to be resolved for `WholeIliasCodeBase cannot depend on exitOrDie`. This was not the case because there were other calls to exit or die within the same class and the violation is only resolved after all exit or die function calls within a class are resolved. This misconception of the violations may be caused by some specific Dicto feedback. Dicto writes a line on where to resolve a dependency, see figure 5.1. But this only points out one occurrence of the dependency in the code.



GUIClasses cannot depend on ilDBClass

The GUI-Layer should not itself interact with the database. Try to build reusable Model classes, adding a layer of abstraction instead of accessing the database.

Newly Introduced Violations

ilQuestionPoolSkillAdministrationGUI depends on ilDB
Resolve dependency in /Modules/TestQuestionPool/classes/class.ilQuestionPoolSkillAdministrationGUI.php:64

ilAssQuestionSkillAssignmentsGUI depends on ilDB

ilAssQuestionSkillUsagesTableGUI depends on ilDB

Figure 5.1: Dicto outlines where the violation occurs.

What you would expect is that it lists all occurrences of the dependency. But for every violation only one occurrence of the dependency is displayed. The reasons are technical: The occurrences of the dependencies are stored in the JSON dependency model. The creation of this model takes most of the time needed for the CI to report on a build and the file resulting from this operation is already 20MB in size for the whole ILIAS code. If you want to store any occurrence of a dependency the time of the build and the size of the model would grow significantly. To avoid the misconception described above, we could leave out this information entirely. But answers from the survey suggest that this is valuable feedback for contributors since it gives an idea of how this dependency looks in the code.

There was a problem detected with consistency as reported by a participant in the contributor's survey (see appendix B.2.4). If the rule set changes, either a rule itself or a variable name the different builds can no longer be compared reliably. The different rules over two builds are matched by their names, thus a rename leads to incomparability. Therefore the number of violations added and removed for builds with added, removed or renamed rules lead to confusion among the surveyed contributors.

A similar problem occurs for big merges into the repository. As the last common ancestor in the commit history since the branching may not have a comparable Dicto result the number of added/removed violations since the last build will not be as expected. Thus we had to disable automated reporting for merges.

I think I understand the meaning [of the CI feedback] but the numbers are wrong.
- Contributor 4

5.3.3 Robustness

After the initial setup of the CI and Dicto, the service was running mostly trouble free. At one time, there was a short delay in builds as the hard disk ran full. This was not Dicto's or TeamCity's fault but was caused by an outdated software on the server. There were two instances where no output was produced in the builds #95, #96, #244 and #279. This was caused by syntax errors in the Dicto rules file. It's important to notice that the CI gives no information about why there is no Dicto output; this will be discussed further in section 5.3.5.

5.3.4 Build Time

The build time of a subset of the current rules was initially somewhere above one hour. We decided that feedback that was not immediate for the developer is much less valuable. Furthermore builds would queue up on a busy day, delaying the feedback even more. After the introduction of HHVM and a lot of optimisation in the Dicto framework and the adapter for the PHP analysis the time went down to about 3 minutes on the same machine. The time needed for the build remained constant even after the addition of new rules(see Figure 5.2). The hiccups in the figure are either due to syntax errors in the rule definition or caused by the one time the hard disk ran full.

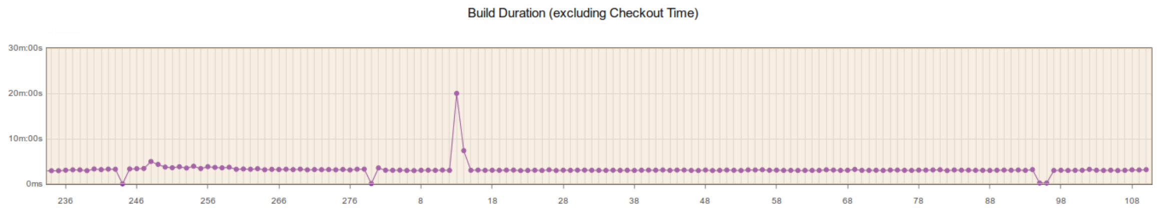


Figure 5.2: Dicto build times.

5.3.5 Dicto documentation and error reporting

One of the biggest issues mentioned in the contributor's survey was the documentation and error reporting of Dicto. First of all the documentation does not exist in any complete manner and new rules are not easily testable without committing them to the central repository. If something goes wrong there is no feedback in the CI, just an empty page. Currently only access to the CI server and extensive knowledge of Dicto quealifies one to debug specified rules. This is a topic discussed in the section 7.

The implementation of new adapters to Dicto needs some knowledge of the smalltalk programming language. A contributor was interested in adding an additional PHP analyser to Dicto but did not get it working. Communities not using Java, Smalltalk or PHP will need an experienced smalltalk developer in order to create new adapters for their preferred language. Alternatively if Moose supported a broader spectrum of languages this problem would be solved as well.

5.4 The Violations

In this section we will discuss the Dicto violations that occurred during the monitoring phase from the 17. June 2015 to the 12. August 2015. The total number of violations went down from 606 to 600. There were 10 violations introduced and 16 violations resolved(see figure 5.3 and figure 5.4). The initial set of violations can be found in section . There were 6 violations fixed net in 56 days. If this continues linearly ILIAS will be violation free in about 15 years.

Based on discussions with developers, all resolved violations were removed on purpose but one. The added violations were introduced by accident.

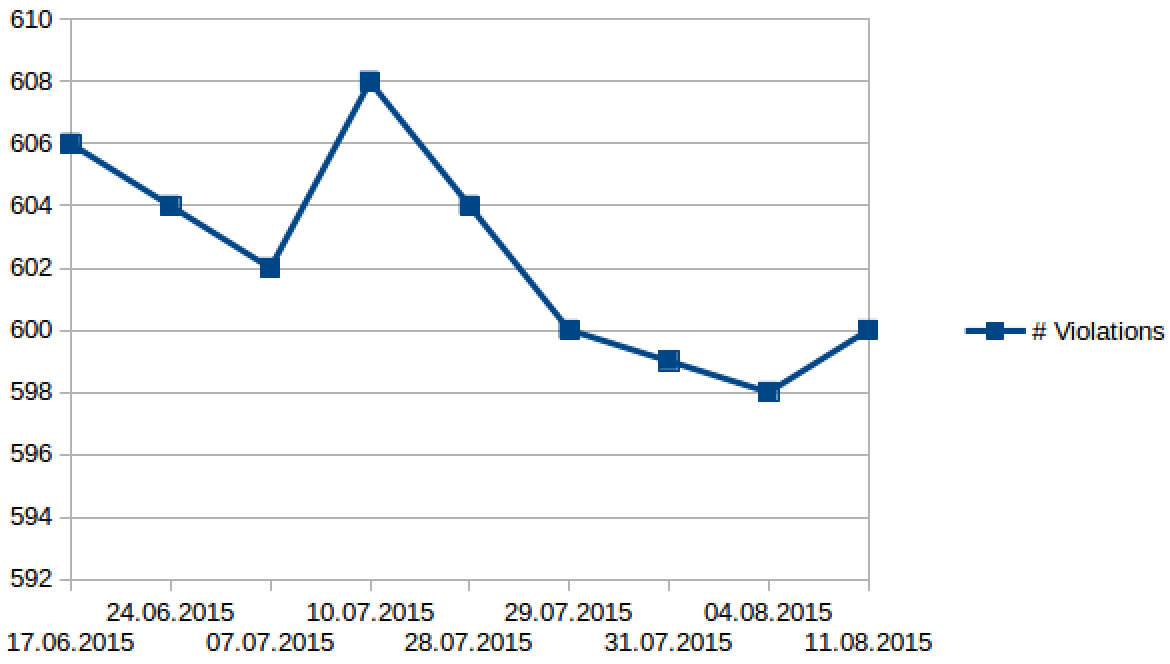


Figure 5.3: Total number of violations.

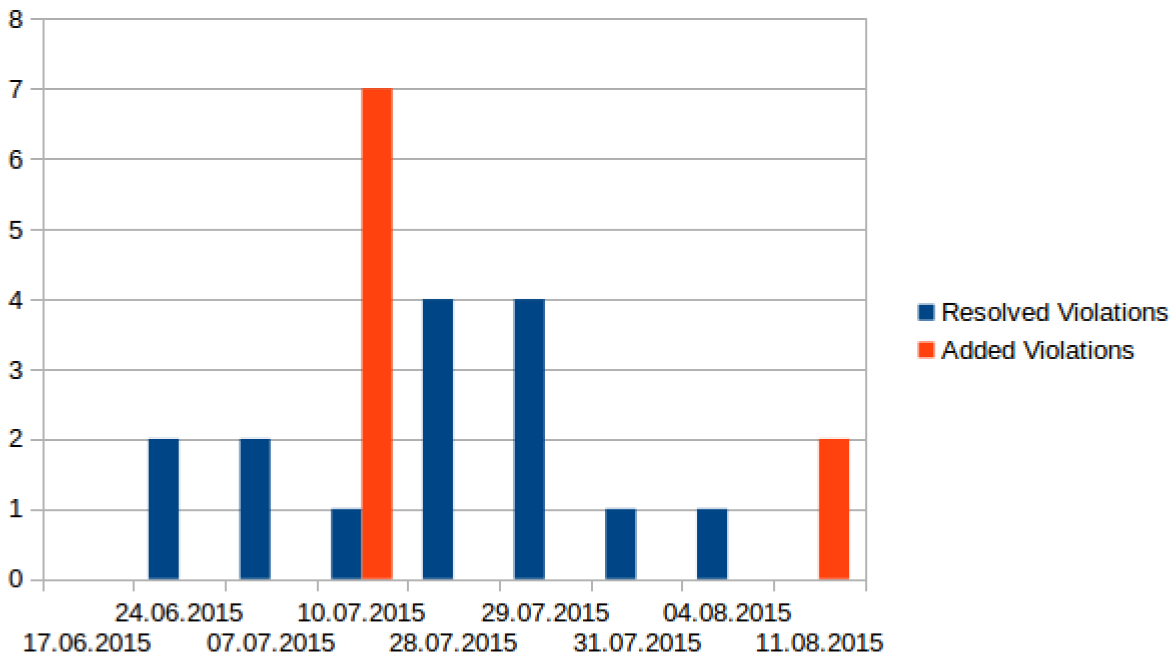


Figure 5.4: Total number of resolved and added violations.

5.4.1 Added violations

None of the added violations were removed directly after being detected by Dicto. The contributors of the code reported to have received and understood the report by Dicto but were either unable to resolve the issues due to architectural reasons (see section 5.4.6) or just didn't come around to fix them at that time. One contributor of the survey stated that he planned on fixing the violations as he thought it will encourage the discussion about the specific violation (see appendix B.2.4). In his words: "It [Dicto Feedback] definitely leverages the discussion about architecture and separation of concerns". The contributors did all report that they see the rule as reasonable in general.

We can conclude that Dicto does report reasonable violations during development in the real world. It has to be seen whether contributors actually react to the detection of added violations by fixing them immediately.

A short description about the added violations:

- GUIClasses cannot depend on ilDBClass

  ```
  ilQuestionPoolSkillAdministrationGUI depends on ilDB
  ilAssQuestionSkillAssignmentsGUI depends on ilDB
  ilAssQuestionSkillUsagesTableGUI depends on ilDB
  ```

  All database usages in the GUI Layer are created due to an existing anti-pattern in ILIAS, read more about it in section 5.4.6.

- GUIClasses cannot depend on ilDBGlobal

  ```
  ilPersonalSkillsGUI depends on GLOBAL/ilDB
  ```

The same applies as for the rule `GUIClasses cannot depend on ilDBClass`. All database usages in the GUI Layer are created due to an existing anti-pattern in ILIAS, read more about it in section 5.4.6.

- ilExceptionsWithoutTopLevelException can only depend on ilExceptions

  ```
  ilAssLacConditionParserException depends on sprintf
  ilAssLacConditionParserException depends on ilLanguage
  ilAssLacConditionParserException depends on ilAssLacFormAlertProvider
  ```

  These are actually not violations in our opinion. The rule was changed during the monitoring phase. With the new rule these violations wouldn't occur.

- only GUIClasses can depend on ilTemplateGlobal

  ```
  ilLinkifyUtil depends on GLOBAL/tpl
  ```

  The violation was added when methods were refactored into a util-class.

- only GUIClasses can depend on ilTemplateClass

  ```
  ilLinkifyUtil depends on ilTemplate
  ```

  The violation was added when methods were refactored into a util-class.

5.4.2 Resolved violations

According to a survey amongst contributors, removed violations were specifically targeted using the feedback of Dicto and TeamCity:

> *I searched very deliberately for violations within our modules and fixed them. On one hand to get our modules violation-free and on the other hand the leader board influenced me.*
> *- Contributor 1*

The motivations behind these changes were various. A contributor wanted to get his modules free free of violations; another participant mentions intrinsic motivation (curiosity for software quality tools and to show support for the architectural conformance checks). The two contributors fixing the most violations both mentioned that the leader board was a motivation to remove the violations. This may not be a sustainable motivation as the fascination about the leader board may decrease over time. The other reasons given seem to be well founded.

We can conclude that Dicto results serve as a working list for refactoring tasks with a hint on where to find issues. Furthermore the leader board and the list of violations occurring in your module seem to be a motivation for developers to have a look at some of their older code base.

A short description about the resolved violations:

- only GUIClasses can depend on ilTabsGlobal

  ```
  ilChatroomTabFactory depends on GLOBAL/ilTabs
  ```

  It was a naming issue: The class was simply renamed as it is a factory for GUI classes and not for model classes.

- only GUIClasses can depend on ilTabsClass `ilChatroomTabFactory depends on ilTabsGUI`

Same issue as above.

- GUIClasses cannot depend on ilDBClass

  `ilTestSkillQuestionAssignmentsGUI depends on ilDB`

  `ilTestToplistGUI depends on ilDB`

  In a bigger refactoring those violations were resolved. This refactoring was not aimed at resolving violations though.

- GUIClasses cannot depend on ilDBGlobal

  `ilPersonalSkillsGUI depends on GLOBAL/ilDB`

  `ilObjAuthSettingsGUI depends on GLOBAL/ilDB`

  `ilTestToplistGUI depends on GLOBAL/ilDB`

  In a bigger refactoring those violations were resolved. This refactoring was not aimed at resolving violations though.

- WholeIliasCodebase cannot invoke exitOrDie

  `ilObjiLincClassroom depends on exit/die`

  `ilObjiLincCourseGUI depends on exit/die`

  The violations were resolved according to the suggestions in the comment of the rule.

- WholeIliasCodebase cannot depend on SuppressErrors

  `ilnetucateXMLAPI depends on @`

  The suppressed error was simply removed.

- WholeIliasCodebase cannot invoke triggerError

  `ilFormPropertyGUI depends on trigger_error`

  `ilBrowser depends on trigger_error`

  The violations were resolved according to the suggestions in the comment of the rule.

5.4.3 Violations of the different groups

In section 5.4 we separated the rules into three categories: The general invariants, current refactoring invariants and the target architecture invariants.

Concerning the general invariants we have 6 added violations and 8 resolved violations. Concerning the current refactoring invariants we have 3 added violations and no resolved violations. This has to be reconsidered though, as the original violations were generated by a rule that was later altered. For the altered rules there are no added violations. Concerning the target architecture invariants there were no added violations and there were 4 resolved violations.

In the general invariants we would expect an overall slight increase in violations due to architectural erosion but the small sample set and the introduction of Dicto may have influenced the outcome: we have a slight decrease in violations. On the other hand the current refactoring invariants remained constant, as the target architecture doesn't influence this invariant. This is exactly what we expected. Last but not least the number of violations caused by the target architecture invariants is constantly decreasing, meaning that there is progress on the refactoring efforts.

5.4.4 Technical Debt

The violations of the given architectural invariants can be seen as technical debt in the ILIAS code base. We will estimate the time needed to fix the different violations in order to approximate the technical debt our architectural monitoring has discovered at the beginning of the monitoring phase. The estimation measures technical debt in hours an experienced ILIAS developer would need to resolve the violations. We argue that in a future requirement any of these violations needs to be addressed thus it is a meaningful representation of technical debt. The results are rounded to half an hour.

GUIClasses cannot depent on ilDBClass/ilDBGlobal (45 Violations): In cases where the controller layer directly accesses the database we need to reallocate the database access into a model class. Furthermore one needs to find all occurrences of identical or similar database queries to use the same model class and/or method. Considering one violation may be caused by several database queries in the same non-model class this task may need to be done several times per violation. We estimate an hour per violation. This leads to a total of **45 hours** in technical debt for this architectural invariant.

only GUIClasses cannot depent on ilTemplateClass/ilTemplateGlobal (101 violations): Several model classes use the templating engine to generate some output for the user. This should not happen, as the GUI layer is responsible for the generation of user output. The dependency must be resolved by extracting the usage of the template classes and globals into the GUI layer. We estimate half an hour per violation; leading to a total of **50.5 hours** in technical debt for this architectural invariant.

only GUIClasses cannot depent on ilTabsClass/ilTabsGlobal (10 violations): The violations of this architectural invariant are very similar to those in the previous rule. The task is to extract the access to ilTabs classes and globals to suitable GUI-Layer classes. We estimate half an hour per violation and **5 hours** in total.

WholeIliasCodebase cannot invoke triggerError (6 violations): Calls to the function `triggerError` are essentially the same as raising an exception. These violations should be resolvable in 10 minutes per violation for a total of **1 hour**.

WholeIliasCodebase cannot invoke eval (3 violations): The function `eval` is currently used in the setup for installing the database and in a third party library. Both cases are currently not resolvable. There needs to be a different approach on how ILIAS handles the setup of the database. A rough estimate would be 5 days or **40 hours** to do so.

WholeIliasCodebase cannot invoke exitOrDie (227 violations): Upon inspecting the violations there seem to be three types of problems developers try to solve using `exit` or `die`. The first one is when a file is sent: a call to `exit` ensures that no additional data is sent after the transmission that would corrupt the file. As there is already a service that sends files, these violations can be solved in an estimated 10 minutes by using the service. The second type is very similar: if a request to ILIAS should resolve in a JSON response developers tend to call `exit` after outputting the JSON string in order to make sure to have no output after the JSON string. To resolve those cases we need to implement a service to send JSON data (2 hours) and an additional 10 minutes of work per violation. The third case is when the developer detects that the user has no permission to do the current request. He or she calls `die` to make sure the request is stopped. This could be resolved by introducing a `Forbidden` or `Access Denied` exception and an according top-level exception handler (4 hours). Afterwards the `die` calls can be replaced by raising an exception (10 minutes occurrence). This leads to a total of 2 hours + 4 hours + 227 * 10 minutes = **44 hours** of technical debt.

WholeIliasCodebase cannot depend on SuppressErrors (203 violations): Wherever error messages are suppressed the suppression needs to be taken away and handled by an exception or similar measures. Most of the times the suppression can just be taken away without any additional effort. This should lead to an average of 15 minutes per violation; leading to a total of **51 hours** technical debt.

ilExceptionsWithoutTopLevelException can only depend on ilExceptions (7 violations): As discussed in section 5.4.1 we considered violations of this rules not to be violations after some inspection.

WholeIliasCodebase cannot invoke SetErrorOrExceptionHandler (4 violations): All calls to `SetErrorOrExceptionHandler` should be removed. They don't need replacement as a top level exception handler is set by ILIAS. We estimate 10 minutes per violation and a total of **1 hour**.

All in all we have an estimate of **238 hours** or **30 days** in technical debt regarding the problems detected by the current rules. The estimates are in general rather low and don't take any unexpected difficulties resolving the violations into account. This leads us to expect about half an hour of work per violation that Dicto detects within ILIAS.

Comparing our estimation with others is hard because most existing estimations for technical debt use project agnostic static analysis covering the whole code base while our approach is project specific and does not necessarily cover all parts of the software. For example the SQALE approach [6] detects violations like: there's no comment block for a method; test coverage of a class is smaller than 70%; there's duplicated code. After that it defines remediation measures and estimates the time needed to execute them. The approach to calculate the technical debt is similar but a key part, the estimated time needed to fix a violation, is different. While the SQALE approach has a fixed set of rules (per programming language) with a given weighting for violations, Dicto needs manual weighting for every rule depending on the project, as done in this section, in order to estimate an architectural debt. This leads arguably to a more accurate estimation but cannot be fully automated.

5.4.5 Comparison to generic automated reports

In this section we will shortly analyse certain commits during the monitoring phase that influenced the number of violations reported by Dicto. We compare the amount of resolved or added violations to generic metrics of the tool PHP Mess Detector[2] (using a standard Mess Detector rule sets[3]). We use the default values for thresholds. This means for example that PHP Mess Detector will report an issue for parts of the code having a cyclomatic complexity higher than 9. The data set considered in this analysis is too small to conclude something about correlation between Dicto violations and the measured metrics (e.g. cyclomatic complexity). The cases are thus discussed qualitatively.

We first pick the changes made in build #20 of the CI. A contributor resolved two violations in Dicto. In the same set of changes the number of problems reported by PHP Mess Detector did not change. This seems to suggest that violations reported by Dicto are independent from complexity metrics.

On build #41 of the CI, Dicto violations went down by 2. The same changes decreased the number of problems reported by the mess detector from 5 to 2. A closer inspection shows that the Dicto violations as well as the resolved mess detector problems were caused by the deletion of several unused files. The contributor targeted specifically Dicto violations and stumbled upon outdated classes no longer in use. This indicates that defined rules detect unused code which most likely is not compliant to the latest architectural standard of the software.

[2]`http://phpmd.org`

[3]CyclomaticComplexity, NPathComplexity, ExcessiveMethodLength, ExcessiveClassLength, ExcessiveParameterList, ExcessivePublicCount, TooManyFields, TooManyMethods and ExcessiveClassComplexity

The biggest change in Dicto violations was a merge of a branch into the trunk (build #261 on the CI). It changed more than 50 files, and added 7 Dicto violations and resolved 1. This merge also led to the detection of an anti-pattern (this will be discussed in section 5.4.6). This set of changes also increased the mess detector's violations from 497 to 512 in one of the directories containing the most changes. The changes contain a lot of newly developed code. Newly developed code can always cause mess detector issues and Dicto violations, thus this finding is not astonishing. The main cause for the introduction of Dicto violations was attributed to new dependencies to the database in the controller layer. Thus there's arguably no direct correlation between the Dicto violations and other measured metrics (e.g. the cyclomatic complexity).

Overall there seems to be little correlation between violations of architectural invariants and generic mess detector outputs.

5.4.6 The detection of an anti-pattern

A very interesting case was stated by one of the developers during the Dicto survey. You can find his explanations in the appendix B.2.3. In one of his modules there are quite a few violations of the rule `GUIClasses cannot depend on ilDB` and during the monitoring phase new violations were added. These violations occur because he wants to avoid the global variable access in the model classes. He does so by injecting the database object through the GUI classes into the model classes. The database is actually not used in the GUI classes but only passed through them. This is known as the courier anti-pattern as discussed by Tom Butler [4].

The courier anti-pattern is detected in two main points, as cited from Tom Butler's Blog: 1. A class has a dependency but never calls any method on it. 2. A class has a dependency in expectation that subclasses will use it. This happens most often in a naïve approach to dependency injection. The disadvantages of the anti-pattern are explained in the blog entry but mainly concern the encapsulation violation between the provider class (the courier) and the consumer class.

Dicto managed to indicate this anti-pattern with one of the given invariants. The courier in our case is a GUI class and the couriered dependency, the database class, was added as a dependency to those classes. The violated rule `GUIClasses cannot depend on ilDB` has a lot of other violations which are considered to be added by inexperienced developers, thus the anti-pattern may have stayed unnoticed. But as the rule was broken in a recent change by an experienced developer a closer inspection led to the discovery of the anti-pattern.

Currently if you want to avoid constant access to global variables, this courier anti-pattern is considered as a reasonable alternative. This was notably also discussed on the third SIG Meeting (see appendix C.3) and finally led to the decision to consider dependency injection as the upcoming project for the SIG refactoring. Especially the fact that this kind of violation was added during the monitoring phase helped identify the anti-pattern. The fact that this kind of violation is added in a constant and probably unavoidable manner leads to the detection of the architectural problem. The discussion on the meeting and the results produced by our monitoring infrastructure pointed to the same problem.

[4]https://r.je/oop-courier-anti-pattern.html

6 Related Work

In this section we discuss the work of Andrew Le Gear and Jacek Rosik with the title “An Industrial Case Study of Architecture Conformance” [4]. We compare our results to Andrew Le Gear and Jacek Rosik’s work because the approaches are very similar: Both case studies let the participants declare invariants for their architecture and then give feedback about deviations from the intended architecture. The case study reported in their paper accompanies the ground up redevelopment of a software project. At the beginning they define a target high level architecture and create a so called high level model (HLM) of the software with regards to the dependencies. The HLM can be compared to the source model (SM), the actually implemented dependencies, to generate a so called reflexion model (RM) [7]. The RM highlights the differences between the HLM and the SM. The RM can be generated at any time and developers may take actions in order to change the SM (*i.e.,* refactor), the HLM or document the violations for later considerations. This is done until the project finishes. The purpose of the study was to determine whether the process usefully informs developers about architectural violations, and to verify if developers act to reach higher architectural conformance and to determine what can be improved to match developer’s needs.

The tools used are very similar to ours, even though the approach of Le Gear *et al.* uses a graph oriented dependency map and our approach uses written rules as input for architectural conformance checking. There are other factors that differentiate the approaches. First off Le Gear & Rosik accompanied a project from the ground up, meaning that they had to deal with less legacy code and in general fewer violations. Secondly the architectural conformance checks were made in physical meetings, while our approach instead gave feedback about the architecture via a continuous integration server. While there are seemingly many more developers working for ILIAS than the three involved in Le Gear & Rosik’s project, the number of contributors interacting in an impactful way with the dependency models were almost the same (4 people on the ILIAS side against 3 people on Le Gear & Rosik’s side). It should be mentioned that Dicto is theoretically more powerful than the jRMTool used in Le Gear & Rosik’s case study, as Dicto can act on other aspects of the Code other than the dependency model (deadlock detection, file contents, etc.). In our context, this doesn’t really matter as the features used in Dicto were almost all focused on the dependency model.

We will have a look at the discussion of Le Gear & Rosik’s paper and compare it to our findings.

Does the process usefully inform developers about architectural violations?
The findings of Le Gear & Rosik indicate that the approach was useful in finding architectural violations. An argument was that even one single developer will introduce violations in his own architecture. Our findings support this claim as the violations introduced in our case study were made by some of the most experienced ILIAS developers. In Le Gear & Rosik's approach the developers were surprised by the violations, which contradicts our experience where the number of 608 violations at the start did not seem to surprise most developers. This is most likely due to the age of the software. In ILIAS developers expect a lot of legacy code and associated violations whereas in a young project you'd expect more conformance. There's a big discrepancy in the number of violations detected. While there were only a few violations added in almost two years for Le Gear & Rosik, in ILIAS we had already 10 violations introduced within only two months. Again this is probably due to the number of developers and the age of the software. Our surveys showed the same enthusiasm for the architectural conformance checking among developers as reported by Le Gear & Rosik.

Do developers act on these violations to increase architectural conformance?
Le Gear & Rosik write that the participants did not act on the detected violations. We had very similar findings. Newly introduced architectural violations did not get fixed in a timely manner. This somewhat overshadows the enthusiasm of the participants discussed in the previous section. It also indicates that experienced contributors introduce violations for a reason and not by accident. In addition to violations introduced during the monitoring phase, in our work we also report on violations that existed beforehand. Developers acted on these older violations as well. This indicates that monitoring legacy code in addition to new implementations leads to a decrease in the deviation in documented architecture and implemented architecture. All in all we can confirm Le Gear & Rosik findings of the inconsistency between identification and removal of architectural violations.

What needs to be improved in the approach to better match developers needs?
The observations made by Le Gear & Rosik are very specific to their monitoring approach, where participants could add dependencies in the high level model. In our case, this could be comparable to exceptions to rules, or the change of rules in general. We can't really compare this to our result as the participants didn't get a chance to change the rules after the first months of monitoring. What we can learn from Le Gear & Rosik is that exceptions seem to obfuscate the feedback for developers: Firstly it's harder to understand a rule with exceptions than one without. Secondly exceptions that only exclude one element from a rule at one point of time may let some future violations go unnoticed as they are covered in the same exception unintentionally.

In general the findings seem to align. There seems to be a lot of enthusiasm around architectural conformance checking and a proper architecture in general as in both cases the proposed approaches seem to ease the lives of developers. While Le Gear & Rosik could not find any resolved violations we could see an overall decline in the amount of violations, suggesting that the enthusiasm is backed by some actual work. On the newly introduced violations the findings align with Le Gear & Rosik's work as they did not get fixed immediately after the identification in either of the studies.

7 Conclusion and Future Work

7.1 Conclusion

The introduction of Dicto in the Open Source community ILIAS can be considered successful. The acceptance and the contribution of the community shows that Dicto gives useful feedback and is integrated well into the development cycle. There are certainly points we have to work on further, like the error reporting and the consistency of the feedback, but overall with an acceptable amount of work Dicto will stay a part of ILIAS development after the case study is over.

We were very lucky in many aspects of the project, as there were several deal-breakers we avoided successfully. We had to overcome several technical challenges: We succeeded in the integration of PHP into Dicto and the execution time could be brought down from over an hour to only a few minutes. There were also social aspects that went well: We tried our best to involve the community in every step of the case study, from the introduction of a new CI, to the definition and feedback of the rules. We had a high participation in surveys and the definition of Dicto rules throughout the case study. In the last meeting of the Jour Fixe before the start of the monitoring phase, the Jour Fixe could have still changed its mind about the introduction of a more or less experimental tool into ILIAS' development cycle. Thus the added time it took for the case study through the involvement of the community in every step was well worth it.

In general for Dicto to be successful in a wide variety of projects it should probably be implemented in a "Software as a Service" style. Most communities will lack the resources to set up a Dicto server and integrating it into a continuous integration environment. The ideal use case would be: The user registers his repository for Dicto checks and adds a rule sheet into his repository. A central Dicto cluster would check for changes in the repository and use the rule sheet to produce a report. The report will then be sent to the contributor. This way the high initial costs for the setup could be avoided.

The implementation of new adapters has proven to be rather easy for users experienced in Pharo/S-malltalk. Developers of ILIAS seem to be discouraged to implement additional adapters, though, as they are unfamiliar with Smalltalk.

The ILIAS community will continue to write rules and reacting on the feedback of those rules will most likely be maintained in the future of ILIAS development. Thus we can say that Dicto is fit for the introduction in open source communities.

7.2 Future Work

In the next cycle for Dicto we will focus on distributing the knowledge of how to write Dicto rules. There are two contributors who are currently trying to introduce additional rules. While the language is very intuitive and thus the introduction of new rules often works on the first try, if the initial attempt fails the feedback of Dicto is very poor. Thus we plan to add two features in the near future. (1) A Dicto Sandbox: Currently authors of rules have no other choice than to commit a modified rule sheet to the central repository in order to see if their rules work and what output they generate. We would like to implement a sandbox where you can run Dicto rules against a GitHub repository in order to debug your rules before you commit them. (2) An Error Report on the Dicto page: Currently if the Dicto build breaks then the CI feedback for Dicto is empty. We would like to give a feedback on the page about what went wrong.

One company working with ILIAS is using Jenkins CI to check their development process. They were interested in using Dicto to check their code with the given ILIAS community rules. The experience of a user not involved in the development of Dicto trying to set up Dicto will be of interest.

Bibliography

[1] J. Brunet, R.A. Bittencourt, D. Serey, and J. Figueiredo. On the evolutionary nature of architectural violations. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 257–266, Oct 2012.

[2] P. Clements and M. Shaw. ”the golden age of software architecture” revisited. *Software, IEEE*, 26(4):70–72, July 2009.

[3] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *Software Engineering, IEEE Transactions on*, 27(1):1–12, Jan 2001.

[4] M. Feilkas, D. Ratiu, and E. Jurgens. The loss of architectural knowledge during system evolution: An industrial case study. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 188–197, May 2009.

[5] L. Hochstein and M. Lindvall. Diagnosing architectural degeneration. In *Software Engineering Workshop, 2003. Proceedings. 28th Annual NASA Goddard*, pages 137–142, Dec 2003.

[6] J.-L. Letouzey. The sqale method for evaluating technical debt. In *Managing Technical Debt (MTD), 2012 Third International Workshop on*, pages 31–36, June 2012.

[7] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '95, pages 18–28, New York, NY, USA, 1995. ACM.

[8] Jacek Rosik, Andrew Le Gear, Jim Buckley, and Muhammad Ali Babar. An industrial case study of architecture conformance. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 80–89, New York, NY, USA, 2008. ACM.

[9] M. Shaw and P. Clements. The golden age of software architecture. *Software, IEEE*, 23(2):31–39, March 2006.

[10] J.B. Tran, M.W. Godfrey, E.H.S. Lee, and R.C. Holt. Architectural repair of open source software. In *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, pages 48–59, 2000.

A
Architectural Discussion

This document will shortly describe some potential problems of the ILIAS software architecture and display example code. Please keep in mind that this is a first draft.

The text tries to be self explanatory but understanding it will be much easier if you are familiar with the development concepts in ILIAS. Thus it is recommended to read into the ILIAS development guide [1] if you are not an active ILIAS developer.

A.1 Control Flow and Routing

A.1.1 Description

This is what even the most minimalistic web service frameworks offer. Binding a certain request from the user to a method on a class. For example a GET request to the following URL:

```
   http://localhost/ilias_44/ilias.php?ref_id=93&cmd=preview&cmdClass=ilwikipagegui&
cmdNode=1:b9:bd&baseClass=ilwikihandlergui
```

Should result in the instantiation of an object of the class ilWikiPageGUI and set the body of the response to the return value of the method viewObject().

To be precise we have a short look at the different parameters:

ref_id The internal id of the wiki object.

baseClass The entry point of the control flow in ILIAS.

cmdNode The path to the cmdClass from the baseClass. Each alphanumeric id stands for a class. In this case "1" is the baseClass ilWikiHandlerGUI, b9 is the ilObjWiki GUI and bd is the cmdClass ilWikiPageGUI.

cmdClass The class which is supposed to handle the request in the end.

[1] http://www.ilias.de/docu/goto_docu_lm_42.html

cmd The command the class receives to decide what to do. In our case a preview of the page should be shown.

There are several things that need to be done for this URL to work. First of all every module in ILIAS, for example a course or a wiki needs a module.xml file to define some of their behaviour. Here we need the definition of an entry point, namely a baseclass for the request.

```
<module [...] id="wiki">
   <baseclasses>
      <baseclass name="ilWikiHandlerGUI" dir="classes" />
   </baseclasses>
   [...]
</module>
```

It's important to notice that there is not such an entry for every class that wants to handle a request. There are rather view baseclasses (entry points) to the routing. After that the routing is done by a so called call structure. This is used to support reusability. This control structure is defined by adding certain annotations into the class comment.

```
/ **
[...]
* @ilCtrl_Calls ilObjWikiGUI: ilPermissionGUI, ilInfoScreenGUI, ilWikiPageGUI
* @ilCtrl_IsCalledBy ilObjWikiGUI: ilRepositoryGUI, ilAdministrationGUI
[...]
** /
```

We have a look at what happens now when ILIAS receives the above GET request.

The script ilias.php instantiates all the global variables needed. For example the current User, the Access checking globals and especially the class ilCtrl which is responsible for the control flow. ilCtrl then looks up the baseclass and decides which class to instantiate first in the call structure based on the entries in the modules.xml files. It calls the executeCommand function on this class. The called object now usually checks for permissions and may add some elements to the template, or in general do anything it wants. At the end the method is supposed to, by convention, use the ilCtrl's forwardCommand method with the parameter being the next class in the control structure. The ilCtrl object then checks if this is a legitimate request according to the defined call structure and then calls the method executeCommand on the next GUI class. This is repeated until the class which relates to the cmdClass in the parameter is found. This class should then handle the request.

A.1.2 The flaws

If you want to know if permissions are correctly checked you need to go through the entire call structure and see if somewhere the permission of the logged in user is checked. Permissions are often checked several times along the way. You don't really know in the command class (the one that you most likely are developing) how many entry points actually lead to your class so you just check the permissions again. Just to make sure.

The object that receives the executeCommand order does not know by default what to do. It has to ask the ilCtrl for the commandClass and then decide if it will handle the request itself or if it wants to forward the command. You also have to look at the next classes in the chain that will be instantiated to decide what you want to do. This leads to huge switch cases in executeCommand methods.

```
switch($next_class)
  {
    case "ilinfoscreengui":
      $this->checkPermission("visible");
      $this->addHeaderAction();
      $this->infoScreen(); // forwards command
      break;

    case 'ilpermissiongui':
      $this->addHeaderAction();
      $ilTabs->activateTab("perm_settings");
      include_once("Services/AccessControl/classes/class.ilPermissionGUI.php
    ↪ ");
      $perm_gui =& new ilPermissionGUI($this);
      $ret =& $this->ctrl->forwardCommand($perm_gui);
      break;

    case 'ilwikipagegui':
      $this->checkPermission("read");
      include_once("./Modules/Wiki/classes/class.ilWikiPageGUI.php");
      $wpage_gui = ilWikiPageGUI::getGUIForTitle($this->object->getId(),
        ilWikiUtil::makeDbTitle($_GET["page"]), $_GET["old_nr"], $this->
    ↪ object->getRefId());
      include_once("./Services/Style/classes/class.ilObjStyleSheet.php");
      $wpage_gui->setStyleId(ilObjStyleSheet::getEffectiveContentStyleId(
        $this->object->getStyleSheetId(), "wiki"));
        $this->setContentStyleSheet();
        if (!$ilAccess->checkAccess("write", "", $this->object->getRefId())
    ↪ \&\&
          (!$ilAccess->checkAccess("edit_content", "", $this->object->
    ↪ getRefId()) ||
          $wpage_gui->getPageObject()->getBlocked()
          ))
        {
          $wpage_gui->setEnableEditing(false);
      }
      [Five more switch cases.]
}
```

Developing with this call structure is not easy. The reloading of the control structure takes several minutes as it has to check in every file for the commented annotations. Reading the path the control structure goes through is not possible due to the cryptic path (e.g. 1:b9:c7). On an end point you never know what is already done and what is up to you. Do you have to check for permissions? Do you have to initialize the standard template? You also have to check if the set command is actually a valid command, this is usually done by hard coding which methods can be invoked from the execute commands and which are not allowed to:

```
      switch ($cmd) {
         case 'configure':
         case 'save':
         case 'saveSorting':
         case 'addEntry':
         case 'createEntry':
         case 'selectEntryType':
            $this->$cmd();
            break;
```

Furthermore if you add a new class to the control flow you have to update every switch statement along the way to forward to your class, as the ilCtrl doesn't do this by default.

A.2 Object-Relational Mapping

A.2.1 Description

There is no unified object-relational mapping[2] (henceforth called ORM) in place currently in ILIAS. There is a database abstraction layer implemented, namely ilDB which is in general a wrapper around the pear library MDB2. The last stable release of MDB2[3] was in may 2007 with version 2.4.1 which is also used in ILIAS. This would be in itself worth of a replacement but could be combined with implementing an ORM.

A.2.2 Drawbacks

The model classes in ILIAS want to be persistent. But due to the lack of a ORM every model class implements the CRUD operations[4] themselves which leads to a lot of work on the developers side and also blows up the code. In the worst case the CRUD operations don't work consistently. Some objects read them out of the database as soon as you instantiate them others need you to call the read method on them first.

This also leads to the problem that most objects are not cached. If the same objects gets instantiated twice in a request it will need two requests to the database and it doesn't cache the object. Only modules that have performance issues will bother to implement a cache.

A.3 Replace the Templating Engine

A.3.1 Description

The current templating engine is pear's HTML_template_IT[5]. Its last stable release was 2006 with version 1.2.1. The templating engine does not allow any control structure in the template files. It only allows blocks of HTML with variables. Any if/else checks or loop structures must be done within the controller.

An Example:

```
public function listAlbums() {
  $this->tpl->addCss('./Customizing/global/plugins/Services/Repository/
  ↪ RepositoryObject/PhotoGallery/templates/default/clearing.css');
  $tpl = new ilTemplate('./Customizing/global/plugins/Services/Repository/
  ↪ RepositoryObject/PhotoGallery/templates/default/Album/tpl.clearing.html
  ↪ ', true, true);
  $obj_id = ilObject::_lookupObjectId($this->object->getRefId());

  /**
   * @var $srObjAlbum srObjAlbum
   */
  if ($this->access->checkAccess('read', '', $this->object->getRefId())) {
    foreach ($this->object->getAlbumObjects() as $srObjAlbum) {
```

[2] http://hibernate.org/orm/what-is-an-orm/
[3] http://pear.php.net/package/MDB2/download
[4] http://de.wikipedia.org/wiki/CRUD
[5] http://pear.php.net/package/HTML_Template_IT/download

```
        $this->ctrl->setParameterByClass('srObjAlbumGUI', 'album_id',
    ↪ $srObjAlbum->getId());
        $tpl->setCurrentBlock('picture');
        $tpl->setVariable('TITLE', $srObjAlbum->getTitle());
        $tpl->setVariable('DATE', date('d.m.Y', strtotime($srObjAlbum->
    ↪ getCreateDate())));
        $tpl->setVariable('COUNT', $srObjAlbum->getPictureCount() . " " .
    ↪ $this->pl->txt('pics'));
        $tpl->setVariable('LINK', $this->ctrl->getLinkTargetByClass('
    ↪ srObjAlbumGUI'));

        if ($srObjAlbum->getPreviewId() > 0) {
          $this->ctrl->setParameterByClass('srObjPictureGUI', 'picture_id',
    ↪ $srObjAlbum->getPreviewId());
          $this->ctrl->setParameterByClass('srObjPictureGUI', 'picture_type',
    ↪  srObjPicture::TITLE_MOSAIC);
          $src_mosaic = $this->ctrl->getLinkTargetByClass("srObjPictureGUI",
    ↪ "sendFile");
        } else {
          //TODO Refactor
          $src_mosaic = './Customizing/global/plugins/Services/Repository/
    ↪ RepositoryObject/PhotoGallery/templates/images/nopreview.jpg';
        }

        $tpl->setVariable('SRC_PREVIEW', $src_mosaic);
        $tpl->parseCurrentBlock();
      }
      if ($this->access->checkAccess('write', '', $this->object->getRefId()))
    ↪  {
        $tpl->setCurrentBlock('addnew');
        $tpl->setVariable('SRC_ADDNEW', './Customizing/global/plugins/
    ↪ Services/Repository/RepositoryObject/PhotoGallery/templates/images/
    ↪ addnew.jpg');
        $tpl->setVariable('LINK_ADDNEW', $this->ctrl->getLinkTargetByClass('
    ↪ srObjAlbumGUI', 'add'));
        $tpl->parseCurrentBlock();
      }
    } else {
      ilUtil::sendFailure($this->pl->txt('permission_denied'), true);
      $this->ctrl->redirect($this, '');
    }
    $this->tpl->setContent($tpl->get());
  }
```

With the template file:

```
<div id="xpho_clearing">
   <ul class="clearing-thumbs">
      <!-- BEGIN picture -->
      <li class="xpho-mosaic album"><a href="{LINK}"><img src="{SRC-PREVIEW
    ↪ }"></a>

         <span class="album-metadata">
            <p class="album-title"><a href="{LINK}">{TITLE}</a></p>
            <span class="album-date"><p>{DATE}</p></span>
            <span class="album-count"><p>{COUNT}</p></span>
         </span>
      </li>
      <!-- END picture -->
   </ul>
   <ul class="clearing-thumbs">
      <!-- BEGIN add-new -->
      <li class="xpho-mosaic album addnew"><a href="{LINK-ADDNEW}"><img src
   ↪ ="{SRC-ADDNEW}"></a></li>
      <!-- END add-new -->
   </ul>
</div>
```

A.3.2 Drawbacks

The separation of the controller classes and the view is not really clear and methods in the controller which use the templating engine are bloated as shown in the example above. A lot of logic is within e.g. the for loops.

A method which uses the templating engine is not easily reusable for the output of for example a JSON format. Having another templating engine that just receives variables and arrays of variables can also be substituted by a JSON template, which just displays these arguments in a JSON format.

The templating engine is also discussed in the ILIAS special interest group for performance as it seems to be a major issue concerning the performance of ILIAS.

A.4 Dependency Management

A.4.1 Description

This section discusses two problems parts:

1. No namespaces are used.
2. A lot of global variables are used.

The first one is pretty straightforward. Namespaces are just not used as when this feature was introduced in PHP 5.3.0 in 30. June 2009[6] the effort was not taken to introduce namespaces in ILIAS.

The second one is discussable. There are some variables which on first sight make sense to be global, for example the database object. Any other pattern for this variable, like dependency injection, would just cause more problems than it would solve. On the other hand on short review there are 31 variables defined as global which are most likely not all legitimately defined and used as globals.

A.4.2 Drawbacks

For the namespaces this is again pretty straightforward: Using a third party library is really hard if it by some misfortune contains a class with the same name as one implemented in ILIAS. There are also really long class names like ”ilDataCollectionRecordListViewdefinitionGUI” as you have to make sure to make the name unique. Just using ”ListViewDefinitionGUI” may collide with some other class definition.

[6] http://php.net/releases/

Problems occurring with global variables are harder to grasp in general. Thus I only describe an issue which came up recently: The currently logged in user is saved in the global variable ilUser. While creating a new item in ILIAS the owner of the user is set to the currently logged in user. A new feature implemented in a plug in added the functionality that on the creation of a user he should automatically get a personal folder. The bug that was created is: If the user is not created by the registration by the user himself but rather an admin that creates the user then the owner of the personal folder is set to the admin rather than the new user. As the folder uses the global variable for determine what user to promote to its owner rather than a variable the only solution was to set the global variable ilUser shortly to the newly created user before setting it back to the admin creating the new user. This shows that ilUser should only be used in Controller classes as the model should not depend on the currently logged in user. But checking for such restrictions is really hard with a global variable.

B Surveys

B.1 First survey

The first survey included questions about coding pain points, development guide related questions and continuous integration related questions.

B.1.1 Development Guide

The survey asked the developers how often they use the development guide and how consistent they think it is with the actually implemented code.

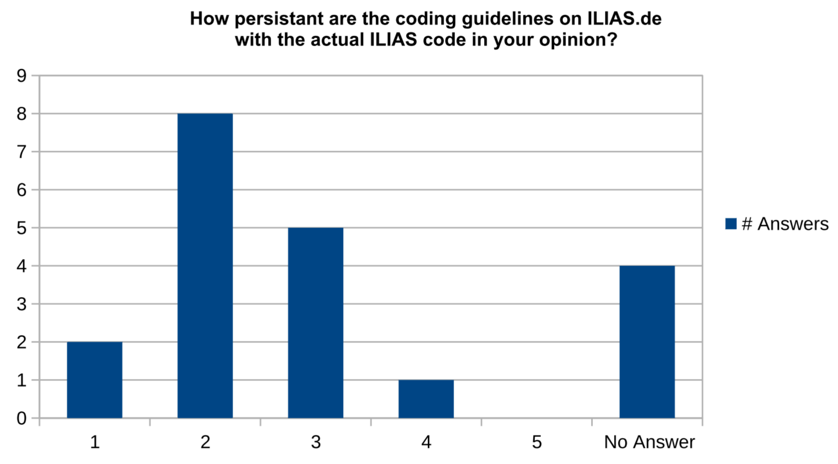


Figure B.1: Survey Results for usage of the development guide. From 0 ”Not persistent at all” to 5 ’Fully presistent’

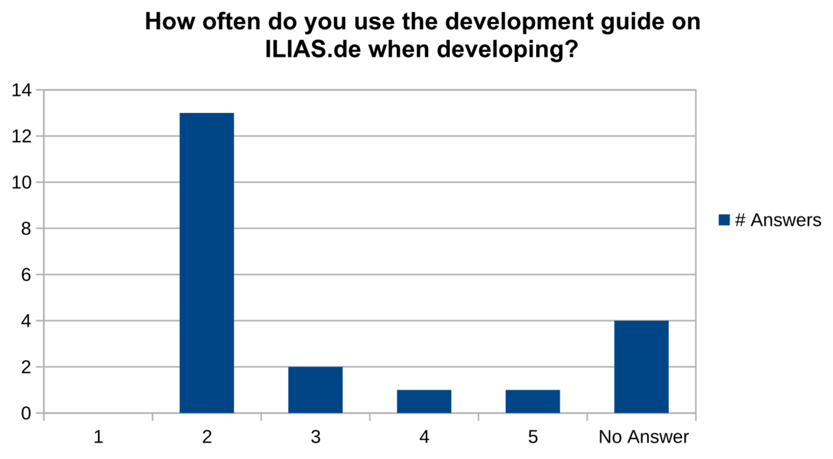


Figure B.2: Survey Results for usage of the development guide. From 0 ”Never” to 5 ’Every day I develop with ILIAS’

B.1.2 Coding Pain Points

The developers should describe their three main pain points when developing in ILIAS. They should give the pain point a name, a description and why it was a problem. After the pain points were gathered, the SIG assigned a difficulty, a perceived usefulness to the pain points and maybe added a comment. You can find a list here: `http://www.ilias.de/docu/goto_docu_dcl_4517.html`.

Name	*Description*	*Difficulty*	*Usefulness*
God classes and missing dependency Injection	In ILIAS, often a class has too many responsibilities. After the single responsibility principle, a class should only have one “task” to solve. Splitting up this logic in multiple classes combined with Dependency injection (Constructor or setter) would help developers to reuse existing code. This is especially a problem with GUI classes, but also model classes. Dependency Injection: The dependencies of a class should be injected in the constructor and not resolved in methods. Also it would be nice if ILIAS would build classes upon Interfaces so that one could easy switch out the implementation with a custom concrete implementation of the interface.	Hard	Very useful
Complex Control Flow	After years I still struggle with ilCtrl. The examples in the DevGuide are obvious, but there are many variants found in the code. It is not clear, what methods are the “real” api and what are only public for legacy purposes. When are return values from executeCommand() provided and expected? When to use uppercase or lowercase for classnames? Related issue with ilObjectGUI: What needs to be “prepared” in upper gui classes, what is done in lower classes? The declaration of the dependencies in the PHPdoc comments and the need to reload the whole control structure to the database slows down the development process.	Hard	Very useful
The GUI Anarchy	There is a lot of different code logic stuffed into the GUI classes. Some put quite a lot of model and data logic in GUI classes, all have to work quite closely with the templates in the GUI classes resulting in loops and big functions only parsing data for the view. Especially the templates and the GUI classes are married rendering a separation of the two extremely painful. This is an issue. Especially if data should be passed to an other template than the default one for a certain purpose (such as testing).	Hard	Very useful

Form Building and Validation	Die Validierung von Formular-Items ist momentan in den entsprechenden “checkInput”-Methoden einzelner il....InputGUI-Klassen implementiert. Das ist suboptimales Design. Validierungs-Verhalten bzw. eine Strategie sollte man von außen an eine entsprechende Instanz übergeben können. Möchte man z.B. bei Verwendung von ilEMailInputGUI zusätzlich noch die Domain validieren (z.B. falls nur ein bestimmter Namensraum gültig ist), muss man eine abgeleitete Klasse erzeugen und die “checkInput”-Methode überschreiben. Das ist schlecht. Zugegeben: Validierungs-Abhängigkeiten zu anderen Items im gleichen Formular (als Sahnehäubchen) sind ggf. nicht trivial, aber zu lösen. Des Weiteren ist es jedes Mal echt nervig, wenn man auf Kundenwunsch Formulare generieren soll, die vom ILIAS-Standard abweichen. Möchte man z.B. mitten im Formular eine Reihe Buttons erzeugen, so muss man quasi für diese Formular-Zeile eigene il...InputGUI-Elemente bauen, die dann mehrere Formular-UI-Elemente (drei Buttons) enthalten bzw. behandeln müssen. Ich möchte einzelne Formular-UI-Elemente erzeugen können und über einen Form-Builder entscheiden, wo diese im Formular positioniert sind, wie sie aussehen (CSS, HTML), etc.	Hard	Very useful
Validity of User Input	Often in ILIAS the POST and GET Variables were not validated. This causes XSS problems.	Medium	Very useful
One Trick Pony Plugin	If you want to have a Plugin which hooks into the GUI (UIHook-Plugin) and you want it to listen to an Event (EventHookPlugin) you can’t do that in one plugin. Rather you need plugins.	Medium	Very useful
TableGUI	The Table-GUI interface is way to complicated to handle (e.g. you need to have a certain sequence of function calls in the constructor otherwise it won’t work) and its not easy extendable (final methods). This component also reload the complete page way too often. Now days this should be done with ajax.	Easy	Very useful
Private/Final	we’re all developers in ILIAS and should know what we are doing. since a lot of methods are private or final, developers cannot override or use them reasonably. at least use protected instead of private, which allows other developers to use them in derivated classes. final should be banned completely. Currently there are 254 final methods 1205 private methods!!!	Easy	Very useful
The global Problem	There are lots and lots of global variables. Some of them like ilDB are initialized in a dynamic way =>PHPStorm cannot resolve its type =>You have no Autocompletion.	Easy	Very useful
Blouted ilTemplate	Core template feature and standard template (adm_content) highly coupled.	Easy	Very useful
Error Handling	Errors in ILIAS are handled in many different ways. Sometimes ILIAS stops with a fatal-error, sometimes you receive an exception or maybe you get an empty page without any information. This makes developing (debugging) unnecessary more complicated.	Easy	Very useful

Template Systems of the 90s	I do not like the template system in ILIAS. It is so difficult to structure a template with this type of template system. There are only blocks and variables, but I would like to have logic like conditions and loops in my templates. I know that it is not the responsibility of a template to contain logic but without it readability and maintainability in complicated templates is an issue.	Hard	Medium
Missing official API	As an occasional developer it's hard to follow changed core concepts in new ILIAS versions that are not documented in the DevGuide. Too many public methods make it hard to decide what is reliable to be used on the long run.	Hard	Medium
ActiveRecord	every developer implements its own CRUD in many classes. since ILIAS 5.0 there's an activerecord which could help getting things much easier. developing new classes should implement activerecord (multi-table-inheritance will be possible asap)	Hard	Very useful
Impossible Inheritance: new vs. Facotry	The ILIAS Dev Guide states, that ILIAS uses OOP as structurizing principle. Despite that, it is impossible to create and use custom subclasses instead of the original classes delivered with ILIAS. When changing some behaviour of class it is therefore necessary to modify the code inline in the original class, which makes it particularly hard to maintain the changes or port them to newer ILIAS versions. If one could tell ILIAS to use e.g. ilObjMyCourse instead of ilObjCourse, i could implement a subclass clearly stating, where the changes were made.	Hard	Very useful
+10 Adaptive Refactoring Resistance Armor	ILIAS still lacks a proper unit test coverage to allow for risk mitigation during refactoring. While the technology is known since years, the developers at large still abstain from writing unit tests. This leads to a situation, where all refactorings mean an unbearable risk once the changes cover more than a single file. Unfortunately, the developers are also highly resistant to evangelism on the matter, as it turned out.	Hard	Very useful
No client side UI-Concepts	Parts of ILIAS need a responsive UI that avoids doing requests all the time, e.g. cloze question editor, image map editor (and other parts of the page editor), calendar. But there is missing a general concept - handle ajax/JSON requests by using a general rest api - client side templating (esp. forms) - re-using data structures (e.g. ADT definitions on server side	Hard	Very useful
The fat private parent	There are many examples where code is horribly refactored. In many cases code needs a tiny part of the functionality of an other class. Instead of just using the parts needed, one has to override/instantiate a huge pile of chunk along with the parts needed. Even worse when overriding: There are often key varriables private without according setter/getter. Also: Many methods are far to long and to much to many things at once. When overriding, one is glad only to override a quite small methods and reuse other small methods while.	Hard	Very useful

The id-confusion	ILIAS heavily relies on a vast amount of different ids to refer to things. Ids come in at least three flavours, that is object ids for ilObjects, reference ids for repository objects and ids to refer to other entities than ilObject. There seems to be no naming convention for variables holding different kind of ids. Furthermore, ids to the same things are spelled out different throughtout the codebase. That is, when is see $foo_id, i never really know whether it is an object id, a reference id or another id. When writing SQL, i never know whether a field is named usr_id or user_id.	Easy	Less useful
3rd Party Code	Especially old Javascript code based on YUI. No concepts, no funding, but deprecated.	Medium	Medium
Repeated Objects	The approach of using getters and setters is used throughout the whole ILIAS Codebase. When setting a lot of properties on the same object, like after getting input from a form to update an object, i have to repeat the variable holding the object over and over again.	Medium	Medium
Chuck Norris Constructor	Most of the ilias service classes (e.g. ilTable2GUI) are making tasks within the constructor that makes it impossible to use these classes in a certain manner. You cannot set any property before some methods are calles, because they are called during the instance creation. If you overwrite this Chuck Norris Constructor with a human developer constructor, calling the parent one, because the much stuff there is to be done and should not be copied into the own constructor for maintenance reasons, some setter calls are to be done before calling the parent constructor, because otherwise some things aren't working for any magical reason. Solve the problem by making constructors do tasks, they are designed for: - construct an object state (variable inits) - don't trigger process logic (reads from db or session)	Medium	Less useful
Classes with too many construct parameters	Some Classes has to many construct parameters. Sometimes you to change only the last parameter and for the rest choose the default value.	Hard	Less useful
Module Main Class	It is not possible to provide commands for the ilListGUI that links directly into sub classes of the modules main class. So every action that can be called for a module have to be a command within the modules main class. In consequence you have to provide gateway commands, that redirects or forwards into a subclass, or you have to hack the whole construct by encapsulate the cmd class within the cmd parameter so it can be extracted in an overwridden method for building the final link target per action.	Easy	Less useful

ilLanguage	ilLanguage is a great example for the “open-closed-principle” in OOP.... oh no, wait: It’s not the code of class.ilLanguage.php but the contents of the language file. These wordlists are out of control, noone can check today which langvars we can safely get rid of. The lack of convention regarding the use of the language variables makes all attempts to fix this sad state futile.	Easy	Less useful
Put everything in one folder - Antipattern	I saw it quite often in ILIAS that components or plugins lack a proper structure in terms of subdividing your problem according to the Single-Responsibility-Pattern. This includes grouping similar responsible files into folders.	Easy	Less useful
Magic Strings	In some cases strings are used instead of constant in order to name items persistently. This leads to hard to read code and sometimes even doubious string operations which could result in performance issues. Also to code will be harder to maintain (what if the string changes, which places would have to apply this changes etc.).	Easy	Less useful
More Hooks and improved User interface-hook	Generally it would be nice if the developer could attach hooks before or after methods. This could be easily implemented for GUI classes, since the execution of methods is handled via the ilCtrl class and the famous “executeCommand” method. Meaning, that this class could also execute custom logic before/after the execution of the “real” method. The UserInterfaceHook is nice but very limited. By adding a new tab, there is no control over the active state, because ILIAS set’s it to active even though it’s not. Generally: Give the developer more flexibility by allowing to hook into the system.	Hard	
DB-Handling and DB-Design	The ILIAS-DB “abstraction” is way to complicated and not based on a todays standard. There are strange length definition for fields, tricky error handling and way to many tables because of sequences (which are not necessary for the most common db-system mysql). On many db-fields are very low max input lengths. This makes proper naming of components complicated (e.g. il_plugin->plugin_id). A few years ago this was ok, but today storage is cheap.	Hard	Less useful
We do not need inline doc and docblocks - antipattern	Documentation is rarely given. Not inline nor in docblocks.	Medium	Less useful
Improving Events	There exist an event system, but it is not used correctly. Often in classes you find methods where a new class is included, a object is construced and a method is called (often static). This is needs to bad maintenance and for developers, it’s hard to see the dependencies of a class.	Medium	
Controller of Cthulu	Those dumps of code - that know too much. - that do too much. - that have a plethora of dependencies - which are improperly handled (It was kinda hard to pick one.)	Hard	Very useful

Fat Private Parent	There are many examples where code is horribly refactored. In many cases code needs a tiny part of the functionality of an other class. Instead of just using the parts needed, one has to override/instantiate a huge pile of chunk along with the parts needed. Even worse when overriding: There are often key varriables private without according setter/getter. Also: Many methods are far to long and to much to many things at once. When overriding, one is glad only to override a quite small methods and reuse other small methods while.	Hard	Very useful
Method-Signatures in derived Classes	If a derived class overwrites a method from its parent class, it should adhere to the signature of the method from its parent class. This is a problem regarding the Liskov Substitution Principle as well as a problem with future PHP versions.	Medium	Very useful

B.1.3 Code Reviews

In a survey we asked developers if the code written for ILIAS gets reviewed:

Answer	Number of Answers
My code gets rarely/never reviewed	8
My code gets reviewed regularly	7
	15

Table B.2: Survey results: Does your code get reviewed?

B.1.4 Continuous Integration

In a survey we asked what the reasons were for developers not to use the continuous server. There were various different answers:

1. Too complicated to set up (3 out of 15 participants)
2. Bad feedback cycle (3 out of 15)
3. Lack of interest in the community (3 out of 15)
4. Lack of unit tests (2 out of 15)
5. Developers don't know about it (2 out of 15)

Furthermore we asked what features a continuous integration server should offer for them to actively use it. The answers concluded in the following features:

1. Better (unit) test policy (5 out of 15 participants)

   e.g. "As soon as there is a good test coverage, It would be a great improvement for the code-quality"

2. Better feedback cycle

   Execute on every commit (3 out of 15)

   Better notification system (2 out of 15)

3. Convince developers to use it (2 out of 15)
4. Easier to use (1 out of 15)

And metrics and feedback that should be covered by the continuous integration server:

1. Unit Tests (7 out of 15 participants)
   Test Coverage (1 out of 15)
2. Code complexity checks (4 out of 15)
3. Architectural checks (3 out of 15)
4. Code smell checks (3 out of 15)
5. Code formatting (2 out of 15)
6. Syntax Checks (2 out of 15)
7. Performance Checks (1 out of 15)

B.2 Contributor's Survey

This survey was held after the two months of monitoring phase. The feedback was given by contributors that had influence to the amount of Dicto violations. Some responds were in German and have been translated. The original is within brackets after the translated version.

B.2.1 Participant 1

OT: Did you check the CI for violations and then fixed them or did it happen through refactoring without considering the CI?

1: I searched very deliberately for violations within our modules and fixed them. On one hand to get our modules violation-free and on the other hand the leader board influenced me. (Ich habe sehr bewusst Violations in den von uns maintain Modulen gesucht und gefixt. Einerseits um unsere Module von den Violations zu befreien, andererseits hat aber auch die Rangliste den Ausschlag gegeben.)

OT: What were your motivations to do so? (Leaderboard, Try it out, fix some of your code, ...)

1: See above. (Siehe oben)

OT: Did you receive the feedback of the CI after fixing a rule?

1: Yes, I received an email. The position on the leader board would be cool to have in the mail. (Ja, eine E-mail habe ich erhalten. Eine Position auf der Rangliste in der Mail wäre cool :-))

OT: Was the feedback of the CI consistent with what you expected after refactoring parts of the code?

1: Yes. (Ja.)

OT: Does the feedback from the CI support in fixing violations?

1: Yes, definitely. The feedback is very important. (Ja, definitiv. Das Feedback ist wichtig.)

OT: Are the rules definitions readable/self-describing?

1: In my opinion: Yes. A developer who did not see the rules in the SIG meetings may judge about this better, though. (M.E. schon, ev. kann dies aber ein Entwickler, der die Rules nicht bereits durch die Sitzungen der SIG kennt besser beurteilen.)

OT: Is the comment to the rule needed/useful?
1: Yes, the description explain the rules well. (ja, die Beschreibungen erklären die Regeln gut)
OT: Does the hint on where to fix the issue help?
1: Definitely yes, you don't have to search for the violations. (Definitiv, damit muss man nicht lange suchen)
OT: What would you change in the feedback of the CI? (Rule descriptions/Look and feel/Notification system/etc.)
1: Rating in the hall of fame [in the feedback email]. (Rang in der Hall of Fame :-))
OT: Is the response time of the CI reasonable?
1: Yes it's completely sufficient(Ja das reicht völlig aus)
OT: Any additional feedback?
1: -

B.2.2 Participant 2

OT: Did you check the CI for violations and then fixed them or did it happen through refactoring without considering the CI?
2: I checked the issues documented in the TeamCity build and fixed some violations according to these build reports.
OT: What were your motivations to do so? (Leaderboard, Try it out, fix some of your code, ...)
2: Intrinsic motivation Leaderboard
OT: Did you receive the feedback of the CI after fixing a rule?
2: Yes.
OT: Was the feedback of the CI consistent with what you expected after refactoring parts of the code?
2: Yes, it was consistent.
OT: Does the feedback from the CI support in fixing violations?
2: Yes, it definitely does.
OT: Are the rules definitions readable/self-describing?
2: The rule definitions used in ILIAS are readable/self-describing. But the Dicto documentation itself needs some improvements (more examples).
OT: Is the comment to the rule needed/useful?
2: Yes, it is.
OT: Does the hint on where to fix the issue help?
2: Yes.
OT: What would you change in the feedback of the CI? (Rule descriptions/Look and feel/Notification system/etc.)
2: Everything is fine. I have no further suggestions for optimizations.
OT: Is the response time of the CI reasonable?
2: Yes.
OT: Any additional feedback?
2: No, except: The Dicto documentation needs to be improved ;-).

B.2.3 Participant 3

Participant 3 did give feedback in flow text form:

> *[...] At the moment I'm actually not really paying attention to the [Dicto] reports and the rules that are behind them. Of course I don't break them intentionally, most of them I used anyways in my development process so far.*
>
> *That's not because I think the approach is stupid, but because I have to spend some time with it first, at the moment I'm lacking the time to do so. [...]*
>
> *One thing I just noticed, though. I want to use [PHP] globals only on the top layer in my modules, and from there on inject them to subclasses. This means I need a ilDB [global instance] in my ilObjTestGUI to deliver it to subsequent classes. In my dreams I would, in the future, only pass through a globals factory, this factory shouldn't use globals themselfs [use a different approach of dependency injection]. Global is stupid.*
>
> *I try to omit globals in my deeper classes, this means I agree with not using ilDB [in GUI classes], but using globals and inject them into other classes I want to do anyways. [...]*
>
> *[...] ich achte tatsächlich noch nicht wirklich auf diese Berichte oder auch die dahinter stehenden Regeln. Natürlich missachte ich die auch nicht absichtlich, mit den meisten Regeln war ich meiner bisherigen Entwicklung so oder so schon als Grundsatz unterwegs. Nicht weil ich diese Sache doof finde, sondern weil ich mich damit zunächst beschäftigen muss, mir dazu aber aktuell auch die Zeit fehlt. In diesem Jahr gibts viel neues, aktuell kämpfe ich seh mit GIT und auch damit Dinge aus den alten Custom SVN Branches in GIT zu übernehmen.*
>
> *Eine Sache fiel mir aber jetzt gerade auf, ich will Globals nur noch auf oberster Ebene meines Moduls verwenden und von da aus injecten, d.h. ich muss in meiner ilObjTestGUI ilDB globaln um es dann weiter zu reichen. In meinen kühnsten Träumen würde ich in Zukunft nur noch eine GlobalsFactory durchreichen wollen, die dann aber bitte auch nicht globaln soll. Global ist blöd.*
>
> *Versuche eben nun global aus meinen tieferen Klassen rauszuhalten, d.h. ich stimme Euch zu, wenn ich ilDB nicht "verwenden" soll, aber globaln und injecten würde ich das innerhalb von GUIs dann doch. [...]*

B.2.4 Participant 4

OT: Did you receive the notification from the CI?

4: Yes.

OT: Did you visit the CI after the notification was sent?

4: Currently in around 80% of the cases, unfortunately I am still getting too many notifications that do not seem to make sense.

OT: Did you understand the feedback from the CI server concerning the added violations? If not, what was the problem?

4: I think I understand the meaning but the numbers are wrong: These are the latest Mails I got:

1 Added Violations: 606 Resolved Violations: 0 Total Violations: 606

2+3 (two times) Added Violations: 709 Resolved Violations: 7 Total Violations: 914

4+5 (two times) Added Violations: 395 Resolved Violations: 3 Total Violations: 600

6+7 (two times) Added Violations: 709 Resolved Violations: 7 Total Violations: 914

8 Added Violations: 2 Resolved Violations: 0 Total Violations: 600

So of the last 8 mails, only one showed correct numbers.

OT: Do you consider the violated rules as reasonable in general?

4: Yes.

OT: Are there any added violations that are not actually violations of this rule? (false-positive) And if so, what violation is it?

4: Yes, lots of them, see numbers above.

OT: Do you think your added violations should be added as exceptions? If so, why?

4: No, since the numbers violations just not have been really added.

OT: Do you consider fixing the violations in the future or did you already fix them? What are your reasons behind it?

4: Yes. It definitely leverages the discussion about architecture and separation of concerns.

OT: What would you change in the feedback of the CI? (Rule descriptions/Look and feel/Notification system/etc.)

4: Fix the remaining issues. When I only get mails if I really added or resolved violations this would be great. And for the future: more rules.

OT: Is the response time of the CI reasonable?

4: Yes.

OT: Any additional feedback?

4: Yes, if the numbers are correct, I will work through the survey again. :-)

C SIG Refactoring: Meeting Minutes

C.1 First meeting

Protocol Meeting 28.01.2015

- Participants

  Richard Klees, Alexander Killing (for the first part, until 3 o'clock), Colin Kiegel, Michael Jansen, Oskar Truffer

Overview

- The meeting was structured into 2 parts.
- In the first part the results of the survey were categorized.
- A total of 33 issues (without dublicated problems) mentioned were discussed and evaluated on two characteristics:

  Easy vs. Hard to integrate.

  Very useful vs. Little use for the development process.
- In the second part we decided on which problems will be tackled first. All of them are in the category: Easy to integrate + very useful.

  The global problem: A lot of globals are declared dynamically thus PHPStorm cannot resolve them (no autocompletion). They should be declared statically.

  Private / Final: We need a general rule/guideline on when to use private and final methods. Furthermore all methods and fields need the privacy declaration.

  Error Handling: Exceptions and Errors must be handled on top level and in a consistent way.

  Validating User Input: Access to the superglobals GET, POST and FILE should be restricted. A wrapper is needed. Currently XSS attacks are in the responsibility of every developer.

- The focus of the SIG will be on the Error Handling!
- As a suitable topic is found the SIG is founded.

Next steps considering the first four Projects
The global problem

- only for type hinting, not general problem with global
- would speed up development, since IDE could perform autocompletion
- helps to avoid bugs, since IDE could hint at type errors
- the static method ilInit::initGlobal needs to be replaced and globals need to be initialized explicitly
- effort: 1

private / final

- general rule to get rid of final and private methods was not considered to be the best solution

  there is a conflict between the goals to have easy customizability of ILIAS and the need to protect the core from undesired overwriting of fundamental behaviour of classes

  'final' could be discussed in the places where the issue pops up
- a solution could be to create a guideline for the usage of access qualifiers and final (proposals):

  protected is default for methods

  public methods could be considered "the interface" to the class

  access qualifier must be declared
- seems to be a question of style
- has connection to the MissingOfficialAPI-problem
- could be checked via CI partially
- could lead to problems since deprecated "var" might be used
- one could also introduce the rule that public methods need to be documented(since they are interface)
- effort for guideline (through jour fixe): 6
- checking via CI would be part of Oskars Master-Thesis

Error Handling

- exceptions should not be leaving context (that is redirect to error.php)
- top level exception handler (that logs stacktrace)
- transform errors to exceptions
- needs a closer look at the behaviour of PHP
- more features in DEVMODE:

  link to source code location where error was thrown

  complete global state under which error was thrown should be observable

- there seems to be more gold hidden under exceptions:
  - define types of exception that could be used throughout ILIAS
  - let access handling throw exception
  - encode the happy path instead of all failure modes
  - use HTTP-Status Codes according to exceptions
  - different top level handlers for different contextes (Web, SOAP, Cron, ...)
- effort for first three points (concept): 4
- effort for first three points (implementation): 3
- example implementation for 4th item: Whoops (http://filp.github.io/whoops/)

Validating User Input

- PSR-7 offers a proposal for an request object
- replace $_POST and $_GET with object and according methods
- could be used for sanitizing/validation
- would accomodate testing
- could escape input to prevent XSS
- seems to be controversial where escaping should be done

SIG Refactoring

- there seem to be common needs that could be tackled together
- We want to take care about the error handling problem first.
- Michal Jansen, Colin Kiegel, Oskar Truffer and Richard Klees decide to launch the SIG Refactoring. Richard and Oskar are chair- and co-chairmen.
- There is an ILIAS-Group for the SIG Refactoring, Oskar will take care of proper documentation:
  - results of survey
  - this protocoll
  - photo of wall created during the first meeting
- Richard will send a proposal for a mission statement of the SIG to the members.
- The concept to solve the error handling problem should be presented on the Dev-Conf.
- Oskar attempts the global problem (for good will)
- Everyone will check how many time they get from their superiors to solve the global problem and then send the result in CC to the SIG members till 6th Feb.
- Richard will make a proposal afterwards how to proceed.
- We want to have a proposal which has high probability to be agreed on at the Dev-Conf and therefore want to collect feedback from other developers before.

C.2 Second Meeting

Conclusions Meeting 25.03.2015

We talked about how to better integrate the continuous integration server in the ILIAS development process. We agreed on the following conclusions:

- There should be a central ILIAS-CI-Server that monitors the official ILIAS repository. This makes it possible for all developers (and of course other people) to refer to a common set of rules and results.
- The results of the CI should be examined collectively in frequent intervals, e.g. on the Jour-Fixe like it is done with the bugtracker. This is a measure to improve the visibility and publicity of the CI in a first step. It is desired to integrate the CI in the development process more closely in further steps.
- An overview over the results of the CI should be send to all ILIAS devs as e-mail in frequent intervals, e.g. weekly.
- After each commit the results of the CI should be send to the commiting dev in an email. This measure tightens the feedback loop for the developer.
- The CI should be enhanced by the implementation of Dicto. This makes it possible to formulate architectural rules and could help to make development guidelines an integral part of the CI.
- The SIG Refactoring will propose rules for the CI to the Jour Fixe.
- The SIG Refactoring will serve as an instance for other people to propose new rules for the CI. This helps to get a consistent rule set for the CI and saves time for the JF. In the long run the SIG Refactoring wants to be the only instance that proposes new rules for the CI to the JF, either by rule or convention.
- In the long run the SIG Refactoring likes to improve the documentation of existing guidelines and rules for all ILIAS developers.

Tasks

- Fabian, Martin and Richard will take care of a proposal for the JF about the CI. Richard will propose a draft.
- Oskar, Richard and Colin will take care to create an initial Dicto ruleset that could be proposed to the JF.
- Oskar, Robin and Colin will take care of the setup and configuration of the CI.

Notes

- Timon proposed an entry in the DevGuide how Exceptions should be handled in ILIAS.
- The current status of the implementation of the Whoops error handler could be found here: https://github.com/klees/ILIAS/tree/whoops It is based on ILIAS 5.0. Feel free to send pull requests.
- Information about Dicto (tool to check architectural rules) could be found here: http://scg.unibe.ch/dicto/

C.3 Third meeting

SIG Refactoring 2015-07-07

- Participants: Oskar Truffer, Michael Jansen, Max Becker, Alex Killing, Timon Amstutz, Richard Klees

Current Projects
Continuous Integration

- Currently Oskar set up a CI-Server. The CI executes the automated tests and checks architectural constraints via Dicto [1].
- Check for PHP-Version:

  Currently ILIAS 5.0 should support 5.3., 5.4., 5.5.

  There should be checks for features of higher versions of PHP.

  There also should be checks for deprecated features (ereg_replace, old style constructors, ...).

  Currently ILIAS only cares about new versions of PHP when release is stable.

  Thus we should introduce checks for 5.3, 5.4, 5.5, 5.6 for the current supported PHP versions.

  We should also check for PHP 7 compliance, as we might be interested to switch to that version in the future and need a possibility to assess the required changes.

  We should also check for hhvm compliance, as this is supported by ILIAS to.

  Unit tests should also be run with different PHP versions.

- Checks on Non-PHP-Code:

  Checks on HTML/Template?

  With the introduction of the UI-Kitchen-Sink it might be interesting to check whether the output of ILIAS complies to the UI-elements defined in the kitchen sink.

  This might be overload atm, as there are already many people looking at the HTML-output of ILIAS.

  The PHP-part of ILIAS might be more interesting than the HTML. Fix one first...

  There might be some approaches to this topic in the University of Bern, maybe the results could be reused for our CI.

  Checks on JS:

  We could introduce a linter for JS-Code.

  Thats no problem for Dicto.

  The rules of JS-Lint seem to be very strict. Are all of them usefull for us?

  The usage of JavaScript in ILIAS is very heterogenous. Hot spots seem to be SCORM, the Page Editor. Some of the JS also is in templates.

  Check for script-tags in templates. (->Dicto)

  Check for onclick-tags in templates. (->Dicto)

  There are very little guidelines for JS in ILIAS. There is no general idea how ILIAS should look client-side in JavaScript.

  Alex IS interested in output of JSLint.

We want to introduce JSLint to the CI with a liberal ruleset to give feedback for interested developers. We will not make the JSLint an important part of the communication strategy of the SIG.

Checks on CSS:

Inline styles are already classified as bugs.

Introduce check for inline styles. (->Dicto)

- Checks on method calling conventions:

  In the current PHP calling conventions PHP is case insensitive.

  As methods (especially in executeCommand) are often called $this->$cmd, that might be a deal breaker to check for PHP calling conventions of PHP 7.

  $this->$cmd also hinders dead code detection.

- Introduce builds for other branches then edge:

  Currently only the trunk is checked.

  It is not obligatory to use the edge-branch.

  Checking other branches as well might distract people, as there will be more mails and stuff.

  Open sourcing the CI-config gives others the possibility to reproduce the CI and use them for themselves. We should also encourage others to open source their CI-configs. There already is some docu on Jenkins from Max in the Dev-Guide.

  Checking the trunk is enough for the moment.

Whoops Error-Handler

- Richard modified ilErrorHandling to use Whoops.
- The Pull-Request for Richards changes to ilErrorHandling was open for 1 month, which lowered Richards motivation substantially.
- Kill PEAR by making raiseError throw an exception.
- Introduce Dicto rule to ban raiseError from the code base.
- If there are any ideas for introducing more information in the Whoops error page, we are very interested.
- We could try to introduce some rules for the exceptions used in ILIAS.

  Which exception should be used when?

  How is the hierarchy of exceptions.

  There is some guideline from Microsoft about this topic, which Max consideres interesting.

  There also are some guidelines in the DevGuide.

  We want to use SPLExceptions for non ILIAS-specific problems (as ”this is an int, but should be a string”) and ilException as base class for ILIAS specific exceptions (as ”this refers to a course, but should refer to a user”)

  Disallow instantiation of ilException via Dicto rule.

- It would be nice to serialize the exception in the prod mode and log it. This would help with debugging. How do other projects handle that problem? It might be interesting to send the Whoops Errorpage to eMail.
- The ilCtrl-path should be available on the error page.

Autoloading

- We need a transition strategy: - Maybe from global classmap to component to ...?
- Options:
  1. Classmap over the whole ILIAS-codebase
     - The resulting class map might be very huge.
     - How should Plugins be introduced?
     - Would be better with global caching?
     - Would that map be reloaded on every request if it is encoded in a php file?
     - This would require a build process of some fashion, as the classmap needs to be generated.
     - Requires the uniqueness of classnames.
  2. ComponentMap with Pseudonamespaces
     - Map on the top level would be smaller. Global autoloader would load a component autoloader under the control of the component maintainer.
     - Would give freedom and responsibility to the maintainers.
     - Could be introduced incrementally via explicit registration of component autoloaders in a global component map.
     - Would require some refactoring as not all classes use a proper pseudo namespace.
     - We would need to organize the uniqueness of pseudo namespaces.
     - Would take the code base in a direction where it would be easier to introduce real namespaces.
     - Would delete information on the component where the class comes from in the calling code.
  3. Namespacing according to PSR-4 with unique class names
     - Colin took care about unique class names already.
     - ilCtrl needs unique class names for GUI classes.
     - Would require the following steps:
       introduce ”namespace XYZ;” on top of very file.
       search and replace ”new CLASS;” with ”new NAMESPACE
       CLASS;”.
       search and replace won’t work in any case, as we also would need to take care of new $foo;
     - Would give information on which component provides the class.
     - Risk of collisions in names when using third party libraries would not exist.
     - Would lead in a direction where dependency managment via composer would be possible and also ILIAS could be modularized.
  4. Namespacing according to PSR-4 without unique class names (e.g. ilMembersTableGUI in different components)

   - Would require changes in ilCtrl, maybe ilObjectFactory, and maybe some others.
   - Setup would be the same as for 3.

5. Mixture between those.

- The SIG Refactoring will give the results of the session as input on the Feature Wiki Page. Overall we do not see enough benefit from autoloading to tackle the different problems atm. Furthermore, the different options come with trade offs and it seems impossible to get to a strong opinion on what option should be preferred. Nonetheless we would really appreciate namespaces.

Registry Pattern to get rid of global variables aka Introduction of DIC

- Why initialize global services, if we do not know whether they are needed.
- The discussion is already three years old and started with problems for unit testing.
- In future PHP versions it is required for the constructor of child classes to keep the signature of constructors.
- In some parts of the code, globals are set via the constructor. That also introduces the "Courier Antipattern", where objects only have dependencies to pass them to other objects.
- Dependency Injection (by Max Becker): During object construction the dependencies of that new object are handed to the object by some other instance.
- Service Locator (by Max Becker): The object to be constructed requests it's dependencies from some other instance.
- Registry Pattern (by Max Becker): Acts basically like the $_GLOBALS-array, wrapped in some interface. It is then possible to defer the creation of the "global" to the point where the instance is actually requested.
- Dependency Injection vs. Service Locator: In the dependency injection case it is possible to inject different dependencies under different contexts.
- General strategy would be:

  Announce a poll about the DI-Container to be used for ILIAS. Let all developers take part in that poll.

  Replace all (requested) globals with requests to a DIC for that (previous) global.

  Initialize the DIC next to the globals with the same content.

  Automate that process as far as possible to make it easy to do the transition.

  Show the stuff to the JF and get the "go".

Future Topics

- How does ILIAS look on the client side in JavaScript? How could build processes support that?
- The possibility to use constructs such as $this->$cmd seem to be a deal breaker for some static analysis. How could we deal with that?
- Should namespacing be forbidden or are there possibilities to introduces namespaces in single components.

Task

- Timon will inform us, if there are any news on the UI project in the University of Bern that are usefull for our automated tests.
- Max will read the Microsoft Exception Guide and transform it to an ILIAS version with regard to SPLExceptions.
- Richard asks Colin to: - integrates PHP/HHVM-Version checks in the CI. - integrates jslint in the CI.
- Oskar open sources TeamCity-config
- Michael writes Dicto-Rules for JS in templates.
- Timon writes Dicto-Rules to check for inline css.
- Michael makes calls to raiseError throw an Exception.
- Michael writes Dicto rules to disallow raiseError.
- Richard adds Autoloading results to feature wiki page.
- Richard sets up and announces poll for the DIC to be used in ILIAS.
- Richard writes script to search and replace GLOBALS and find pathological cases.
- Max and Richard bring strategy to get rid of globals to the jour fixe in written form.
- Max integrates Dicto in the Jenkins CI setup.
- Max integrates Dicto in the Jenkins CI setup.
- Richard puts Protocoll in Wiki.