Towards Differential-Based Continuous Code Reviews

Mario Bernhart (bernhart@mit.edu)

31.09.2012

Introduction

This is a summary paper of the visiting research at the MIT Complex Systems Research Lab[1] of Prof. Nancy Leveson[2] to apply, elaborate and evaluate the current results (method and tool) of the objective dissertation *Towards Differential-Based Continuous Code Reviews* in the context of aviation specific software engineering projects.

Software reviews are an integral part of critical software engineering projects. Traditional approaches are acknowledged as effective, but not very efficient quality assurance techniques. In the course of this dissertation a method and tool for fine-grained process-integrative software code inspections were designed and implemented. The continuous approach with appropriate tool support shall reduce the effort of a rigorous application of software code reviews. At the same time the traceability shall be improved.

The technical framework is the de-facto standard for code review tools build for the popular open-source integrated development environment Eclipse[3]. An international developer community especially from Austria, Germany, USA and Canada is established and works actively on the further development and improvement of the framework. The broad usage of the framework and tool is an indicator for the validity of the approach.

In the course of the visiting research the method and tool are be applied, elaborated and evaluated in a safety critical context. The first chapter summarizes the application context and the second chapter describes the evaluation methodology.

[1] sunnyday.mit.edu/csrl.html
[2] esd.mit.edu/Faculty_Pages/leveson/leveson.htm
[3] www.eclipse.org/reviews

Chapter 1

Incremental Reengineering and Migration of a 40 Year Old Airport Operations System

This chapter describes the challenges and experiences with the incremental re-engineering and migration of a 40 year old airport operations system. The undocumented COBOL legacy system has to be replaced within given constraints such as limited downtime. A 3-step technical strategy is derived and successfully applied to the re-engineering task in this project. The incremental approach and resulting parallel operations of both systems are the most significant technical drivers for complexity in this environment. Furthermore, this report describes the process for planning, analyzing and designing a replacement system that is backed by strong user acceptance. The user interface design task of taking the system from VT100 to a web interface was a critical success factor, as well as live testing with actual production data and actual user interactions. Other aspects such as training and end user documentation are discussed.

1.1 Introduction

The legacy AODB (Airport Operational Database) system recently celebrated it's 40th birthday, but this is not exactly a reason to celebrate for the IT-department of this 20-million passengers per year airport. License

cost for the Cobol processors are increased every year (since there are less and less customers for these), the two last developers of the legacy system are about to retire in the next 24 months and the market for COBOL developers is not providing adequate personnel. In addition, a new terminal is planned for the near future and the required adoptions would be a disproportional risk and cost to implement in the legacy system.

In this report we describe the challenges and experiences with the successful re-engineering and displacement of this system. In contrast to previous works of the authors [20] an incremental migration strategy was applied instead of a big-bang. The undertaking took about 18 months to complete and involved 31 full-time engineers. Two of them being the legacy developers, which was a critical success factor.

1.1.1 Problem and Constraints

The legacy Cobol GCOS8 host system consists of a core system (the AODB) and more other integrated subsystems such as the cargo management and load-planning. All of them are part of one monolithic host instance only separated by code separation and conventions. This report only describes the re-engineering of the operational core. This part consists of 114 transactional processing routines (TPR) and 819 library routines together with about 250 KLOC Cobol code. Further 109 IDS-II database configurations with 3500 LOC. The average load is 124 transactions per second and there are about 1000 concurrent users connected with a VT100 terminal.

The critical constraint in this context is the maximum downtime: During regular traffic hours after 1 hour of downtime one of two runways has to be closed and after 4 hours the airport shuts down. During nighttime there is a 2-3 hours window of low traffic that can be handled manually without the core system being online.

From a users perspective the main challenge is to migrate from a text-based VT100 interface to a web interface. The legacy interface has a relatively flat learning curve, but once the user is trained it is very efficient and has very low response times.

Make or buy? Even if there is a range of commercial products for airport operations software, this airport has a tradition of in-house development and this was also imperative for this project.

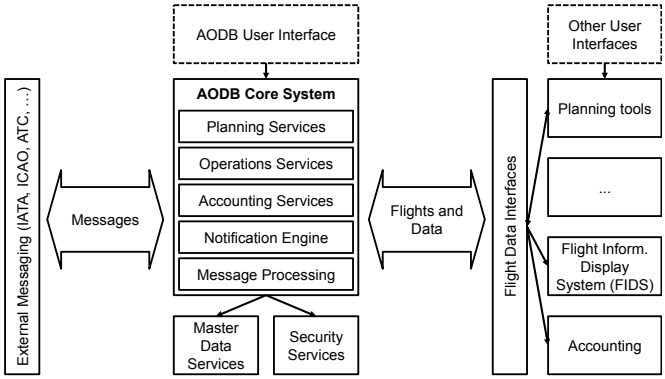

Figure 1.1: A generic and simplified AODB architecture similar to the one in this report.

1.2 Strategy for an incremental migration

The main risk here is a potential downtime that exceeds the constraints described in the previous section. This may lead to heavy traffic restrictions for the 70 operating airlines and with this to a significant financial loss. To address this risk an incremental strategy for development and migration was chosen. Incremental approaches, in general, reduce the risk that comes with a big-bang, but also add a substantial overhead. One key problem is how to decompose the system into reasonable parts that may be migrated one after the other. An ideal process would be a fine-grained incremental strategy, but in practice the atomic decomposable modules are rather large (little big bangs).

An incremental migration would also require the parallel operations of the legacy system and the new system. This is a non-trivial challenge and the whole process from planning, analysis, development and validation was greatly influenced by this factor. An airport operations system is a complex system by its own measures, but operating two systems at the same time adds a significant factor for the interaction complexity. To achieve the required degree of functional and technical compatibility a strict one-to-one strategy was implemented. This was an indispensable cornerstone for this project.

To pre-evaluate the strategy and the process a template feature was selected and implemented. As expected, the first attempt failed (mainly due to the wrong cut through non-separatable system parts) and Brooks' *plan to throw one away* was as relevant as ever. The template feature had to be

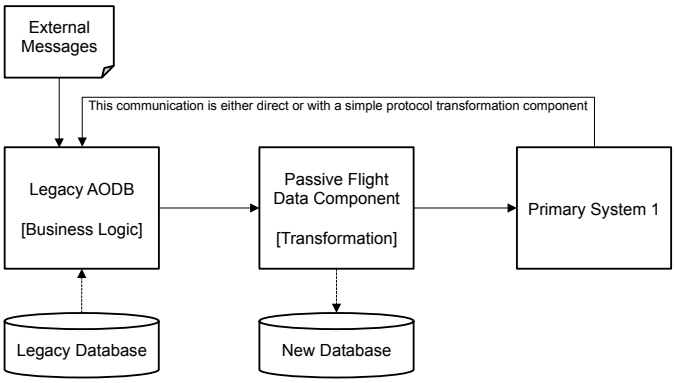

Figure 1.2: Step 1 of the incremental migration strategy. The main element is the Flight Data Component that listens on the legacy interface for flight data updates and transforms that to a new model before forwarding it to the primary systems.

functionally and technically representative, but not too large to limit the time loss of failing attempts. In this case we chose the block-off operation of an outbound aircraft for several reasons: It is a central step in the business process of handling outbound flights with many calculations such as the rotational delay and sending of external SITA[1] messages such as the movement departure message. The block-off also updates the position and gate occupations data, which is a primary data structure for an airport and usually relevant for accounting. The internal notifications system was also strongly involved with sending and withdrawing a set of system messages.

From a technical perspective a step-wise migration was followed. This separation allows the decoupling of input and output communication in general. This concept was applied for the migration of each interfacing component and the user interaction. Message-based input communication (e.g. SITA or IATA messages from other airports or the near-surface radar data), in general, was set up to be processed in parallel. The risk of potential race conditions was considered to be of relatively low impact in this context.

1.2.1 Step 1: Flight Data Interfaces

The main feature of an AODB is to provide the so-called tactical traffic image to all primary systems. Such a flight data interface is the lifeline of every airport. The first step targets to encapsulate the data flow from the

[1] http://www.sita.aero/

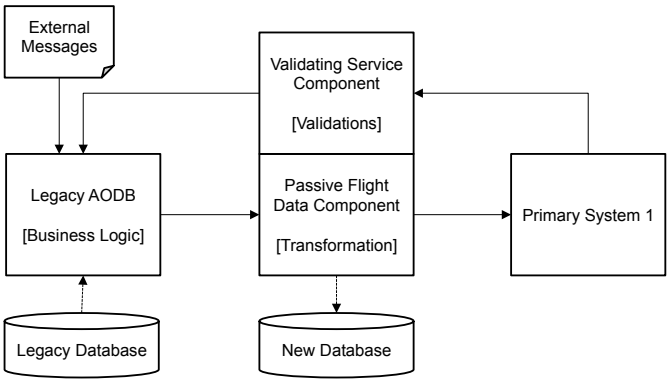

Figure 1.3: Step 2 of the incremental migration strategy. The main element is the Service Component that provides service interfaces with all relevant validations, but not the business logic.

legacy core system to the primary systems (i.e. flight data consumers). In this case there are about 20 of such consumers. The legacy system had only one generic flight data interface so only one component was required. This *one size fits all* strategy is often found in older systems, whereas nowadays there is usually a variety of specific interfaces. Also typical for old AODB systems, the legacy system pushes out nearly all information for a flight instead of only the changed information. This is because some decades ago a FIDS (Flight Information Display System) had no logic at all and was typically directly connected to the AODB.

Step 1 builds the technical foundation by defining a data model and the interfaces (XML over JMS) that are to replace the legacy interfaces (text-based on a TCP socket). After Step 1 a primary system was able to receive all actual flight data and operate with it on a read-only basis.

1.2.2 Step 2: Service Interfaces

In Step 2 a facading component provides the input interfaces to receive any type of data or user input. The inputs are validated and transformed to VT100 calls and forwarded to the legacy system. A set of functional services are defined and exposed. The design of the interfaces is strongly related to the legacy interfaces so that the mapping would be practicable. The validation is critical to further separate the primary systems from the legacy AODB. The validation logic needs to be part of the future AODB and those validations are needed to provide the corresponding return codes for

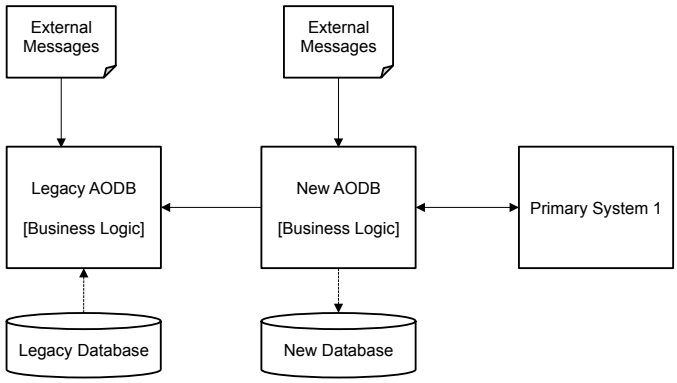

Figure 1.4: Step 3 of the incremental migration strategy. The two components of Step 2 are now merged together and provide a fully functional AODB. The legacy system is updated asynchronously to keep in sync.

the caller. Step 2 requires the actual flight data to perform the validations and therefor Step 1 is a prerequisite.

With Step 2 there the input and output to and from the legacy system is encapsulated, a primary system may be fully functional without a direct dependency to the legacy system. Still the legacy system provides the business logic and calculates the tactical traffic image.

1.2.3 Step 3: Master AODB

In Step 3 the two separate facading components of Step 1 and 2 are merged together and the required business logic is implemented. Inputs are still forwarded to the legacy system to keep both systems in sync. The forwarding with Step 3 is asynchronous (i.e. after the transaction commit), this makes the new core system the master node in the parallel operations. Information flowing from the legacy system to the new core system is ignored or may be used for verification purposes only.

1.3 Re-Engineering Process

The re-engineering process was adopted to follow the direction and constraints from the strategy in the previous section. This section describes selected aspects and challenges of the planning, analysis, design, implementation and verification of this re-engineering effort.

1.3.1 Planning

Following the strategy from the previous section, the planning on the macro-level was mainly about finding the right technical and organizational cuts and defining incremental steps for the reengineering process. A total of 43 technical work packages were defined. Many of those are aggregating a set of functional features to be released to a specific user group. Some were only technical and provided preparations or intermediate steps before a feature gets released to the user. Those work packages were distributed to five major releases. Even with the goal of evenly distributing the complexity and risk within all releases, the second last release was the most critical part of all with a master-slave switch towards the new system for the core feature set. To reduce the shift the risk towards the end of the project, the final release was planned with relatively low technical risk. Some features were implemented and tested and included into a release without enabling it in production. To reduce the risk those features were enabled at a later time, maybe between mayor releases. A complete list of feature-sets and corresponding work packages would excess the length of this report and is very specific for each project. In general, for an AODB the following elements may be cut into separate parts to be released separately:

- Flight planning
- Inbound and outbound flight processing
- Accounting
- Message (Telex) receiving
- Message (Telex) sending
- User notifications
- Interfaces to primary systems
- Interface to FIDS

Any attempt of further splitting the inbound and outbound flight processing (e.g. by regular procedures and irregularities) have been failed in a dual operations setting with the legacy systems.

One other important planning aspect was the technical integration of the core system with the other primary systems and interfaces. There was a total of 26 system that depend on the core AODB and, the other way

around, there were 10 dependencies on other systems by the new AODB. The ration between in and out dependencies is typical for a core system and this *anatomy of a core system* had a significant influence on the planning process.

1.3.2 Analysis

Following the one-on-one policy and having no documentation all the main part of the analysis was to read the legacy code and to extract the functional requirements from there. This is a time-intensive process, but since both systems are running in parallel, there needs to be an exact understanding of the structure and behavior of the legacy system. The code reading was performed line-by-line for each TPR and the results were documented in a human-readable pseudocode to a wiki. There was also an automatically translated Java version of the legacy source, but using this for the analysis was not feasible for the following reasons: 1. Every translation looses important details e.g. implicit design rules, naming conventions etc. and also many technical language-specific aspects are not fully transferable from one model to the other. 2. The legacy developers had no Java knowledge and their support was key for performing the analysis especially with having no documentation of the legacy system at all. Usage statistics of the last 3 months were used to support the code reading by identifying sections that are never called. The code was also automatically analyzed to extract the dependencies between TPRs.

During the code reading process several bugs were identified in the legacy system. Some of them were critical for the parallel operations and had to be either corrected in the legacy system or a workaround had to be implemented in the new system. In general, a change in the legacy system is considered to be of higher risk and the default strategy was to implement the workaround. In some cases both options would be unfavorable - e.g. for a complex and rarely used feature. In this case the users were informed of differences in both systems and had to correct data manually. Only a very small group of users were instructed and granted for corrective actions during operations.

1.3.3 User interface design

The migration from a VT100 client to a modern AJAX-enabled web interface was one of the core design tasks in that scenario. The general acceptance of the new AODB system is mainly influenced by the user interface as this is the most obvious part of the system from a users perspective. A lack

of user acceptance would be a major risk and this aspect is also a often cited critical success factor for any major re-engineering project. But this is not only a hit or miss question. There are many tradeoffs and influencing factors in such design tasks. The legacy system, for example, had a very flat learning curve, but once the users were familiar with the typical three-letter command and all those shortcuts it provides a very efficient and fast interface. Figure 1.5 shows a legacy interface for assigning baggage belts to incoming flights shortly after landing. Many modern web interfaces are more intuitive, but the efficiency of a lean and responsive VT100 interface is hard to match in todays design space. So learnability and efficiency are two factors that may are mutually conflicting. The design goal was set to be the best tradeoff between an intuitive yet efficient interface. To achieve this goal the interaction design was very conservative and always followed the given structure from the legacy system. To make use of todays technology elements such as find-as-you-type or drag-and-drop were added as additional input methods, but in addition to the commonly known structures. This strategy was referred to as *sugar on top*.

A total of 1000 users from all ages and equal female/male distribution are divided into 16 distinctive user groups - each of them performing a different set of tasks. Dynamic navigation structures and filtering of UI elements support a more customized presentation of the user interface. This is more specific to the actual task and reduces information overload and distraction. The features are also sorted by the actual usage rates of each function. Changes to the UI structure were not done at runtime but the technical design allowed to do this with a very local change in the client code.

A contextual inquiry was performed for all identified user groups. To investigate the working environment a the user's site is an important pre-condition for the design process. At an airport the users usually work with more than one screen and more than a handful of different primary systems all at the same time. A typical workflow at an airport includes the use of a set of systems in very different environments such as the tower, apron or CUTE environments at the boarding gates. Also the integration of voice communication and telex printing have to be considered to get a complete picture of the user's perspective. The actual design process then followed a rigorous four step approach:

1. Step: The first step in the process is to identify and understand the legacy interface. As an example, figure 1.5 shows one of two interfaces for assigning baggage belts for incoming flights. The second (not shown in this report) is a text list - a notification system to show incoming

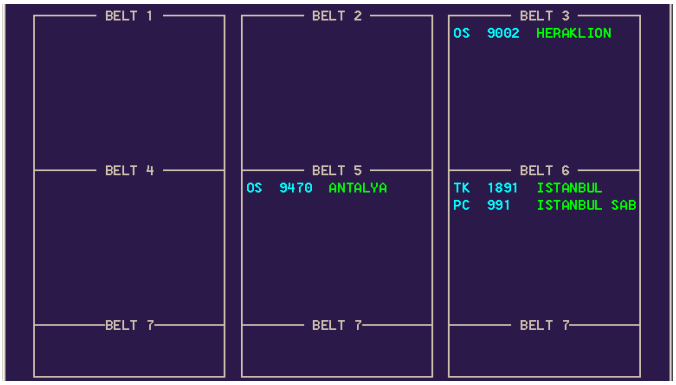

Figure 1.5: UI Process Step 1: Analysis of existing legacy user interface. This example shows the allocation of incoming flights to baggage belts.

flights when they are landed. When a flight lands, the user gets an acoustic signal and switches to the list, marks down the flight number, switches back to the main screen and enters the flight to assign a baggage belt.

2. Step: This is the first actual design task. All functionalities are written down on a set of cards that are provided to selected users that represent a user group. With a mix of card selecting and free-form hand-noted design the early design prototypes are shaped. Figure 1.6 shows a card for the baggage belt main screen plus a notification inbox for incoming flights on the side at the same time.

3. Step: Taking the low-fidelity prototypes from the previous step, in Step 3 the interaction designers transform the input to a simple but testable wireframe prototype. By interacting with the prototype, the users further improve the interaction process. In this case the drag-and-drop of notifications (landed flights) to the respective baggage belts was added. Figure 1.7 shows the clickable prototype.

4. Step: The final step was to implement and test the prototypes. Many details only surface after implementation, but in general the interaction design was very mature at this point. Here the focus shifts more towards the visual design. Figure 1.8 shows the implementation of the example.

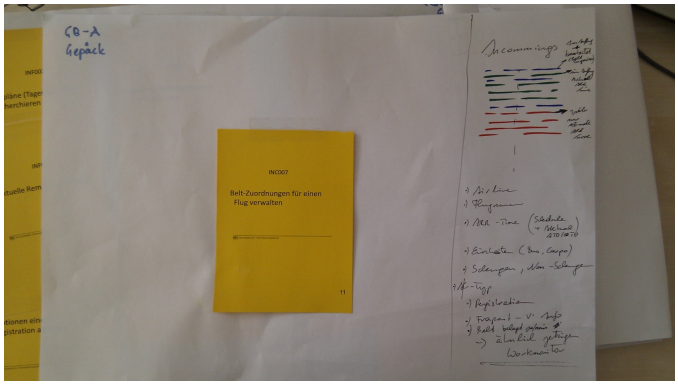

Figure 1.6: UI Process Step 2: Paper-based prototype with card selection and free form design.

1.3.4 Technical design

Following the strict one-for-one strategy the technical services were designed to reflect the previously identified functional requirements. However, certain features underwent a complete redesign in order to improve user/customer benefit, such as

- The legacy host system provided a hard-coded difficult-to-reuse user notification mechanism that informs the user ether that an event has occurred (e.g. a landing) or that an event is missing (e.g. missing block-in after the maximum taxiing time). This was replaced by an event-driven rule-based notification system. Its configuration can be easily altered/extended at run-time via Java Expression Language (EL) in the database.

- Hard-coded security and tenancy checks were replaced by a declarative java annotation-driven approach that increases readability and modifiability and reduces code duplication.

- Many processes of the legacy system are triggered by periodically repeated checks or batches (every minute or every five minutes). As far as possible these functions were either inlined into the respective service code in order to increase cohesion, readability and maintainability or an event-based decoupling is implemented. The shift from a batch-based to an event-based software design was one of the major technical difficulties to overcome. Scheduled tasks are externalized into a separate batch-like environment.

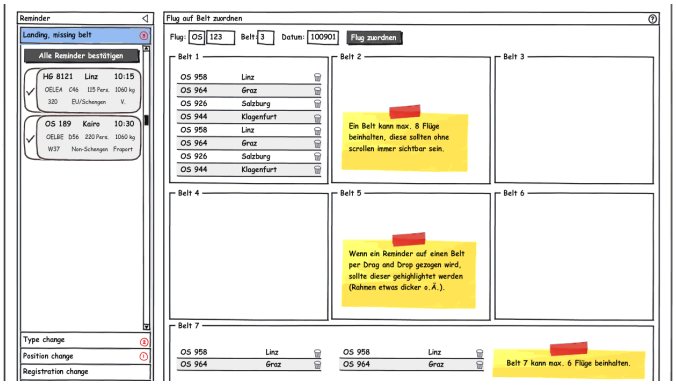

Figure 1.7: UI Process Step 3: Clickable wireframe prototype to test interaction design.

- A 2-flight re-landing model from the legacy system (i.e. reuse of existing flights in the database) was replaced by a 4-flight model (i.e. creating new flights for the re-arrival and re-departure) witch is the current state of the field. This is one of the most severe design changes that also had to be adopted in the legacy system for parallel operations of both systems. In this special case, the high change effort was rated lower than the risk of maintaining the 2-flight model in the new system.

All code required to keep the legacy system synchronized was written and structured so that it could be easily removed after the displacement of the legacy system was completed. With high availability in mind the application features a clustering mechanism ensuring that the processing of incoming messages (SITA etc.) would be continued by another node without human interaction if failure occurred at one or mode nodes.

The application provides exactly two input channels, REST-style HTTP calls and JMS. JMS is used for all asynchronous communication, e.g. telexes/messages from and to airlines, air traffic control and other airports. A strict single service strategy was pursued: Both GUI and applications that would communicate synchronously with the AODB use the same REST calls, e.g. the flight block-off operation can be triggered manually in the AODB or in a neighboring system. The GUI is written entirely in HTML and JavaScript and communicates solely via REST calls with the servers. This approach allows for future GUI implementations without necessitating changes in the back end code. Updates to flights, e.g. landing or departure,

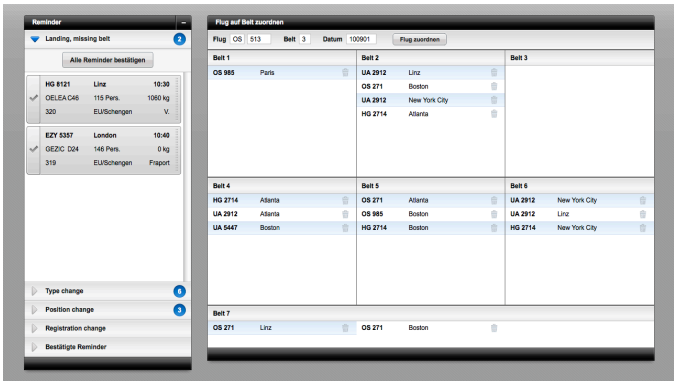

Figure 1.8: UI Process Step 4: Implemented interface with visual design for user acceptance testing.

are published via a JMS topic to other airport systems.

1.4 Verification and Validation

One of the most resource intensive tasks was the verification and validation (or testing) of the new system. Again, the parallel operations here drives the complexity. On the development level the common unit and integration tests are performed automatically. Unit tests coverage was higher than 70 percent and integration test coverage (together with the unit tests) reaches a near 90 percent. Integration tests include the driving of interfaces (e.g. REST) and the verification of the JMS output. All developers tests are integrated in the continuous integration setup after the build.

The main verification effort though went into the functional black-box lab-tests. All features went through an extensive structured test process. One of the key challenges was to prepare the test environment and provide useful test data. The test stages of the legacy system had a nightly copy of production data and as long as the new AODB was running in a passive mode (compare figure 1.2 and 1.3) those states are automatically propagated to the new components. Only after a component reached the active mode (compare figure 1.4) the test data had to be copied from production in the new environment at the same time as this was done for the legacy system to ensure a consistent data set. After the manual execution a good amount of test cases were selected to be automated by a black-box automation framework. To ensure the non-functional requirements, some specific tests were

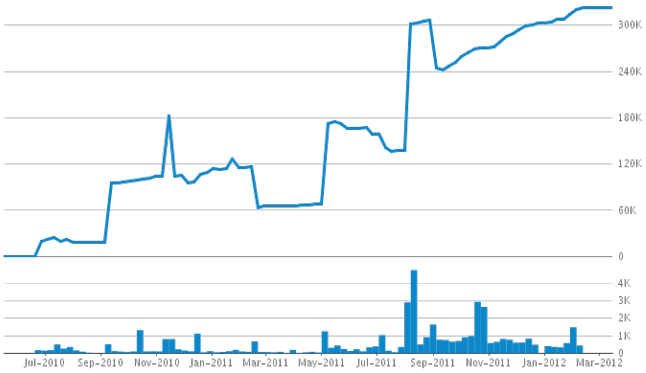

Figure 1.9: LOC development of the project (immediate jump in LOC is due to merge and refactoring with a separate codebase). About two-thirds of the artifacts are Java, but there is still a significant use of other technologies.

performed such as an automated performance test.

Key users were integrated in user tests throughout the process to validate the work at different stages or maturity levels. In generally, the users tested a set of features multiple times before the final acceptance was granted in a formal acceptance test.

The most valuable tests were performed in a test stage, but with live inputs from the production environment. Since the majority of system inputs at an airport is message-based all automated communication lines were forked from production into the test environment to provide live updates. For the manual inputs testers were located next to operational airport personnel and manually redoing their actual inputs in the test stage. After such a test run both, the production and the test environment data were compared to find unwanted variances.

On the development level, the most intensive, but effective verification was done by continuous and differential code reviews. Whenever there was a finished development task, one team member performs a code review of all related source code changes. The application this specific technique was subject to a comparative study against a traditional model of code reviews.

1.5 Conclusions

In this report we give an overview of challenges and experiences with the incremental re-engineering and migration of an old airport operations system.

The incremental approach was dominating the strategy and has very different attributes compared to a big-bang approach. The overhead that comes with such an incremental migration is large, but in many environments the risk of a big-bang is unacceptable. A perfect finer-grained distribution of risk was not achieved, but the individual migration items were isolated to the smallest yet coherent units. In this report we describe the strategy and planning on the macro level, but the influence was even on very low level tasks and decisions. Selected aspects may be part of future reports.

A strong focus on the user interface design task was a critical success factor. The acceptance factor in the user base was high. The conservative interaction design and progressive visual design provided both, a good learning curve and an effective usability.

Testing in an operational environment with actual production data and live user interactions was effortful but highly valuable and surfaced many issues that could not be found in a lab setting. Code reviews played a crucial role in assuring the quality on the source code level.

Chapter 2

Applying Continuous Code Reviews in Airport Operations Software

Code reviews are an integral part of the development of a dependable system such as for airport operations. It is commonly accepted that code reviews are an effective quality assurance technique even if a rigorous application is also a high cost factor. For large software systems a formal method may be inapplicable throughout the whole codebase. In this study an airport operational database (AODB) is developed with the application of a more lightweight approach to code reviews. A continuous, distributed and change-based process is applied by the development team and evaluated in comparison to team walkthroughs (IEEE-1028) as a baseline method.

The application showed to be highly useful, equally effective as the baseline, but more efficient especially for the preparation, execution and rework effort. The results show that continuous code reviews also support the understanding of the codebase and the concept of collective ownership. Such processes may not completely substitute a more formal and effortful technique. Especially for reviewing critical design aspects or complex items a traditional approach is still more appropriate. The main outcome is that such lightweight code reviews may be used together with more formal approaches to ensure a high coverage and that the degree of formalism should be adopted to the criticality of the item under review.

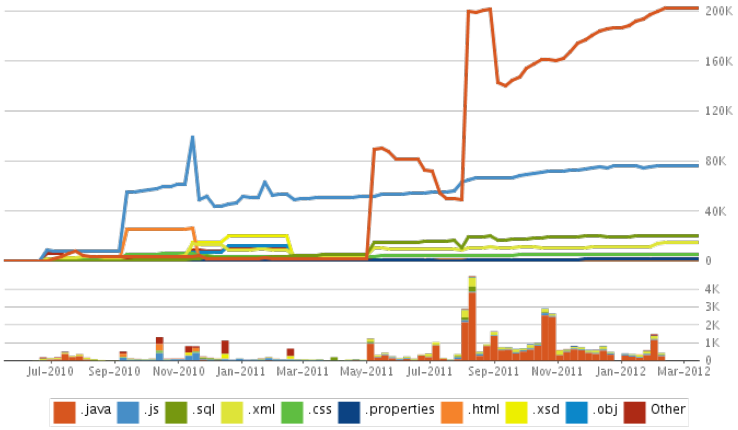

Figure 2.1: LOC per filetype (immediate jump in java LOC is due to merge and refactoring with a separate codebase). About two-thirds of the artifacts are Java, but there is still a significant use of other technologies that are review items as well.

2.1 Introduction

Code reviews play an important role in the quality assurance of dependable systems such as for airport operations [9]. In this study an Airport Operational Database (AODB) system of an medium-sized (20 million passengers per year) airport was developed in incrementally set into operations within 18 months. A team of 8 software engineers had developed the system based on a set of 106 functional requirements (e.g. register for deicing) derived from the legacy system that was to be replaced. The system has about 300 KLOC and the distribution of filetypes is shown in figure 2.1.

The common practice of code reviews in this environment is a team-walkthrough as defined in the IEEE-1028 standard [1] at the end of each development iteration. The high overhead and some other drawbacks, as described in the following section, motivated the development team to adopt the code reviews towards a more lightweight and continuous process that is strongly related to the common practice in open-source projects. Such processes and the supporting tools are covered by earlier research of the authors [2] and [3] and is referred to as Delta-Continuous-Review (DCR). The main properties of the applied method are:

- *Distributed*: As in [19] and [15] many code review processes and tools support distributed or global software development. In this case the

team was mainly located at one site, but the process and tool used allowed to perform the reviews and rework remotely.

- *Asynchronous*: In contrast to a traditional approach as in IEEE-1028 chapter 7 there is no synchronous or face-to-face meeting of reviewers at any time in the process. All tasks are independent but have to follow a distinct order.

- *Differential-Based*: (or Delta-based) The main property of the applied code review procedures is that the item under inspection is not a structural element - such as module - but a set of changes. In this case all changes that relate to one specific functionality are aggregated. A functional entity may consist of changes in multiple structural entities e.g. processing of a new datatype in an airport message causes changes in the message processing module and the core services module. Technically this is a set of changed lines of code (LOC) that has one or more diffs per item. When reviewing, the reviewer inspects the changed LOCs. Figure 2.2 shows a diff-view in the development environment that is used to perform the code reviews. In general, one change is also significantly smaller that a traditional review item. As the review item is only a delta to the previous version, the reviewer is required to have an overall knowledge of the codebase.

- *Continuous*: The reviews are done continuously to attain a high coverage of code with the reviews and to keep the delta between reviews small. Also the application of low-overhead and continuous methods is very typical in modern software engineering approaches e.g. continuous integration in especially promoted by the open-source communities.

A total of 114 code reviews have been executed and evaluated. Section 2.4 explains the procedures how the reviews are performed.

2.2 Related work

Laitenberger and DeBaud [14] published a comprehensive meta-study that summarizes many different code review approaches and work out a common taxonomy. They also summarize the empirical results such as an average code review effectiveness of about 57%. Further Kollanus et al. [13] categorized many publications in the field from 1991 until 2005. Ciolkowski, Laitenberger and Biffl [7] published a survey about the state of the practice.

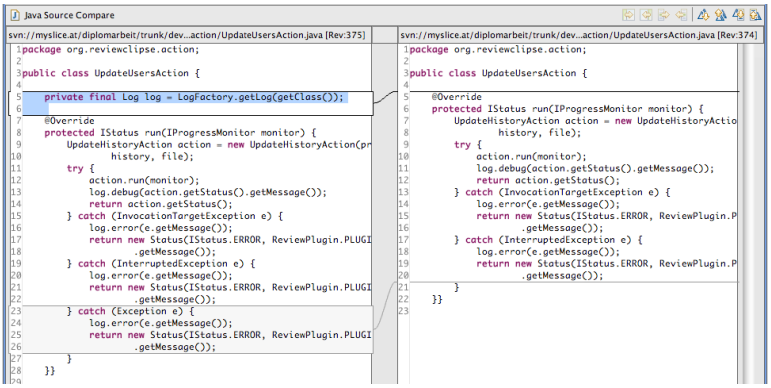

Figure 2.2: Compare Editor for Reviews is used for the reviews. The reviewer opens a change in this view in the integrated development environment (IDE; in this case Eclipse) to investigate the differences to the previous version of one item in the changeset.

They state that the review process should be tailored and integrated in the development process and executed regularly.

Stein et al. [19] and Meyer [15] worked on distributed asynchronous code reviews that are also properties of this work. Remillard discusses tools that support differential reviewing [16].

The open source communities contribute to the field of code reviews and their practical, sometimes very large-scale application. The contribution and upstreaming process of Linux is the most prominent example of that. Other include Google which developed Gerrit [10] as a GIT-based code patch review tool for the Andriod Open Source Project (AOSP). It is used to review the contributions to Andriod in the form of patches (diffs to the reference codebase). Gerrit is a successor of the Google in-house tool Mondrian that has very similar properties as the applied method in this report. Kersten and Murphy [11] presented Mylyn. A task-based interface to connect review systems (or more generally any task management system) to the popular integrated development environment (IDE) Eclipse. One recent development is the Mylyn Gerrit connector [8] where the authors of this report actively participate as project lead and committers. One aspect of this work is to converge the more lightweight approaches from open source with the common practices in dependable systems engineering.

2.3 Research question

Code reviews are a principal quality assurance technique for dependable systems. As systems are constantly growing in size, complexity and degree of integration methods of developing and assuring must be scaleable to keep in pace. While 300 KLOC are not overly large, it is still a challenge to apply a traditional code review with a high coverage. In this report the software development unit typically applies team code walkthroughs according to the IEEE-1028 chapter 7. Those are considered to be very effective but cause a high overhead. Developers also reported some drawbacks in the process such as the following:

- Many software modules exceed the size of code that is coverable by one review session
- Reviewers have to prepare (i.e. read and understand the code) themselves to participate in the review with significant effort
- Reviewers have to be physically present
- Timing of the review is critical:
  - If review is late in the process, the errors detection and resolution is expensive
  - If review is early in the process, the review does not cover the changes made later

To overcome those issues the development team adopted the code review process according to the elements as described in the introduction. The main research question that arises from such a methodical shift is if the adopted approach is still a valid and useful. The development team was competent to change that process at any time in accordance with the development team leader and the one person who was in charge to supervise the overall quality assurance concept. Since this was not the case, the general applicability is considered to be given.

To following questions are deduced from the application and shall be investigated through this research:

- **1. Evaluation**: How useful and appropriate is the applied code review method in this context?
- **2. Comparison**: How does the applied code review method compare to the baseline?

The goal of this work is to obtain basic evaluation results from a practical application and to guide further research i.e. through deviation of hypotheses and possible further more formal and controlled experimentation.

2.4 Methodology

2.4.1 Code reviews

A total of 114 code reviews have been performed and evaluated. There is a near 1:1 relationship between the 106 functional requirements and the performed reviews. Some functional items were large and therefor split into sub-items. The development process was feature-driven, so that each feature (or functional requirement) represents one coherent work-package with a set of directly related development tasks. The basic steps involved in the code review were performed as follows:

1. Develop a function. Each change in the source code repository is linked to a specific function. When one function is finished, proceed to step 2.

2. One developer (randomly chosen for each review and other than the original authors) chooses the code review and aggregates all changes that are made within the context of one functional entity. The development infrastructure as shown in figure 2.3 support the linking between functions and the related changes.

3. The reviewer open all changes in a compare view within the development environment (compare figure 2.2) and documents the review results.

4. The original authors get notified and may immediately start the rework. (Re-inspection after rework is not formally covered)

The assignment of one reviewer to a specific review was done by the developers themselves by unrestricted pulling a review task. Most of the development work is done in pairs. It is standing to reason that the original authors and all pair partners are excluded from doing the code review. This is for two reasons: First, the original authors should never do the review on their own and second, this supports the knowledge distribution within the team.

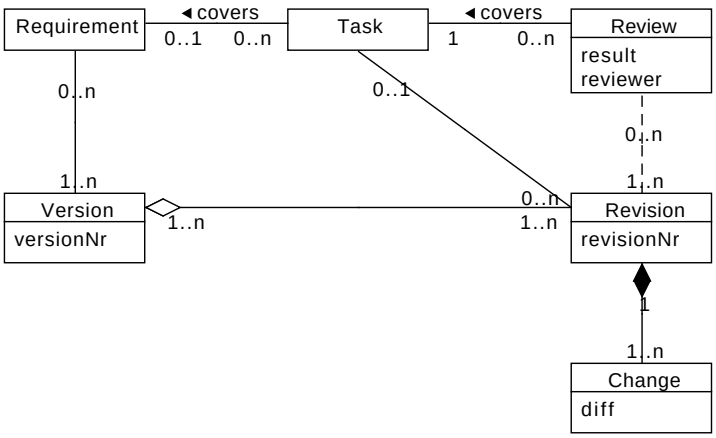

Figure 2.3: Data model for reviews for linking between entities is supported by the development environment services. Horizontal relations are for the traceability to the requirements and vertical relations support the link to the code. The link between the review and the revision is logical. Technically this is resolved via the task.

2.4.2 Study methodology

The study is performed as an expert evaluation in the form of a a-posteriori survey. All 8 developers of the system take part as subjects. Table 2.1 shows the number of reviews per developer related to their experience. Since in this context there is quantitative or qualitative empirical data for the baseline model, the study could not be comparative on data with statistical tests. For each primary research question a set of survey questions is derived. For rating questions a 6-step Likert-scale is used to express the accordance. For each question the subject should also give a textual explanation for the rating answer. The following questions or measures are defined for the evaluation:

- Q1.1: The applied method is appropriate and useful (1: true, 6: false)
- Q1.2: Estimated code coverage (in percent)
- Q1.3: Estimated personal effort (in percent)
- Q1.4: Estimated effectiveness (in percent)

Subject	Experience in years	Number of reviews
A	6	15
B	11	4
C	4	25
D	3	33
E	12	5
F	6	21
G	1	4
H	1	8
SUM		**114**

Table 2.1: Subjects and number of reviews

- Q1.5: Result quality (1: high, 6: low)
- Q1.6: Effect on understanding the code (1: high, 6: low)
- Q1.7: Effect on collective ownership (1: high, 6: low)
- Q1.8: There are preconditions and limiting factors under which the application of the applied review method is useful (1: true, 6: false)

The following questions or measures are defined for the comparison (all 1: high, 6: low):

- Q2.1: Efficiency compared to baseline
- Q2.2: Effectiveness compared to baseline
- Q2.3: Rework efficiency compared to baseline
- Q2.4: Replaceability of baseline

After the design and pre-test with two subjects, the survey was executed 18 months after the begin of the adoption. Each subject answered the survey individually and had access to all recorded review data and artifacts of the project.

Notes on validity

Rating (especially comparative ratings) and estimations are always subjective and are apparently weaker than measured data. Further, the subjects

Figure 2.4: Data shows the distribution of code reviews during the development period of 18 months

in this study motivated the adoption of the process and may be biased according to the results. The findings of this study may be generalized, but as the related work sections shows, there is not a strongly related research. However, there is a practical relevance since the applied method is commonly used in major open source communities. The authors encourage the research community to repeat the study, possibly in a different context.

2.5 Results

This section provides the rating and estimations of the survey questions in the corresponding tables with basic statistical values. In addition, the textual results are summarized in a coded form (item and count) as a result of a qualitative analysis. The raw data would exceed the length of this publication. Textual results for question 1.6 and 1.7 are skipped due to lower relevance to the original research questions. All study material and raw result data is available upon request. Please contact the authors. Figure 2.4 shows the distribution of reviews during the development period.

2.5.1 Evaluation

Question	**mean**	**median**	**stdev**	**min**	**max**
Q1.1	1,13	1,00	0,35	1,00	2,00
Q1.2 (%)	81,25	82,50	9,45	65,00	90,00
Q1.3 (%)	5,38	5,00	2,50	1,00	10,00
Q1.4 (%)	28,21	30,00	12,81	10,00	50,00
Q1.5	1,75	2,00	0,71	1,00	3,00
Q1.6	1,50	1,00	0,76	1,00	3,00
Q1.7	2,00	2,00	0,93	1,00	3,00
Q1.8	1,63	1,50	0,74	1,00	3,00

Table 2.2: Evaluation results

The following coded textual results are provided for question 1.1:

- Low overhead for preparation and execution (4)

- Short turnaround time (3)
- High review coverage/depht achievable (2)
- Helps to check for conformance to development standards (1)
- Supports knowledge sharing/distribution (1)
- Prevents personal exhaustion during review (1)
- Results in even workload distribution within the team (1)
- Aligned with agile principles (1)

The following coded textual results are provided for question 1.8:

- Qualification of reviewers should be high and evenly distributed within the team (4)
- Specific regulations, industry standards and law e.g. for safety-critical software may restrict the application (2)
- Feature-driven development process and proper development infrastructure (2)
- Reviewer needs the have initial/prior knowledge of codebase (1)
- Application should be short-cycled (1)
- Advantageous for less complex projects, where team discussion is less required (1)

2.5.2 Comparison

Question	mean	median	stdev	min	max
Q2.1	1,75	2,00	0,71	1,00	3,00
Q2.2	2,50	2,50	1,20	1,00	4,00
Q2.3	2.00	1,50	0,41	1,00	5,00
Q2.4	2.00	1,50	0,41	1,00	5,00

Table 2.3: Comparison results (all rated 1: high, 6: low)

The following coded textual results are provided for question 2.1:

- With DCR there is less overhead for execution (5)

- With DCR there is less overhead for preparation (3)
- DCR is more flexible in application, because it is asynchronous (2)
- With DCR a higher coverage is achievable with less effort (2)
- With DCR there is less overhead for documentation (1)
- With walkthroughs there is less code to review, because reviewing all changes (in sum) is more volume than only finished items (1)

The following coded textual results are provided for question 2.2:

- DCR more effective, because scope of each review is smaller (4)
- DCR is less effective, because only changes are observed (2)
- DCR more effective, because it is more flexible i.e. reviewer can individually focus on error prone parts (1)
- No relevant difference (1)
- Team (walkthrough) is more effective than singe reviewer (1)

The following coded textual results are provided for question 2.3:

- DCR rework is more efficient, because of reduced turnaround time (5)
- Smaller scope of DCR results in smaller rework tasks and those are more efficient in sum (1)
- With DCR due to the faster feedback similar issues can be prevented (1)
- DCR rework is less effective because inline comments may not be sufficient to communicate review result (1)

The following coded textual results are provided for question 2.4:

- Different scope, strengths and weaknesses: reasonable application for both methods (4)
- Replaceable because comparable and more efficient (3)
- Limited practical applicability of baseline due to high overhead and thus (even if not exactly comparable) DCR are an alternative (2)

2.6 Analysis

. In this section we interpret and discuss the findings from the data section in the context of the two research questions: 1. Evaluation: How useful and appropriate is the applied code review method in this context? And 2. Comparison: How does the applied code review method compare to the baseline?

2.6.1 Evaluation

The result from Q1.1 shows that the application was appropriate and useful. This was not surprising since the development team followed the process over the whole development period even if the team was authorized to change the process at any time. The main factor seems to be a good cost/benefit ratio of the applied code reviews and the developer's preference to smaller work items. The results show that the applied process helps to understand the code and supports the concept of collective ownership (Q1.6 and Q1.7) The estimated review coverage of code (Q1.2) is above 80 percent which would be a sufficient or even desired value in most settings. In comparison to that the personal effort was estimated to be around 5 percent of the developers overall effort (Q1.3). This value is also very consistent with the time tracking data. This value seems very low, but even in more formal environments the code reviews make only a fraction of the developers effort. Together with the high coverage value a good cost/benefit ratio is given. One of the most relevant metrics in code reviews is the effectiveness (Percentage issues found by review compared to all issues or likelihood to find an issue). Literature reports that the typical effectiveness is about 57 percent [14]. Even if the result quality was considered to be very good (Q1.5) and the comparative rating to the baseline was positive (Q2.2), the estimated effectiveness of this application is only about a half of what literature predicts. The estimated value of around 30 percent was spot-testet against recorded issue data and showed to be valid. The most obvious reason seems to be a lower performance of the applied method compared to the state-of-the-art. But as the applied method is a very lightweight process it seems still to be a very reasonable cost/benefit ratio. There are several other factors that might have an influence on the effectiveness such as the examples below. It can be assumed that the effectiveness is context dependent to a certain degree.

- All other assurance techniques (such as integration testing and functional testing) are also performed very short-cycled, therefor the code reviews had a stronger focus on issues that may not be revield by other

techniques such as compliance to coding standards and maintainability of code.

- The operational complexity in this case was especially high, because the developed AODB system is operated in parallel with the legacy AODB to provide an incremental migration path for the system interfaces. A significant share of such (hard to test before) issues related to that factor was found in operations.

The application is generally universal and motivated by but not limited to the application domain. Yet, several limiting factors and/or preconditions are stated in the results. The application mainly depends on a high qualification of the reviewers.

2.6.2 Comparison

The results from Q2.1 show that the applied method is clearly highly efficient mainly because of a low overhead for preparation and execution. Also the rework efficiency (Q2.3) is rated much better for DCR because of typically a very short turnaround time (= time between original development and review). It is commonly accepted that the shorter the time between the error injection and the error identification, the less is the removal effort [5].

The effectiveness (Q2.2) though is more questionable and yield a more differentiated result. Even if the result very positive on average, there is a larger range and the qualitative analysis contains facts that support the baseline method in being more effective. Most frequently it is stated that DCR is more effective because of smaller review units. Still very valid counterarguments are that reviewing only the changes may not reveal some errors. Also a team review (as in the baseline) has a very positive collaborative effect and would be more effective most probably for complex review items.

The baseline method is considered to be substitutable with DCR (Q2.4) to a certain degree. If the application of a more formal code review method is not feasible (e.g. due to time and resource constraints) the application of a more lightweight method may still be reasonable and beneficial. In general different strengths and weaknesses of both approaches are stated in several questions of the survey and this supports the finding that there is a reasonable application for both methods.

2.7 Summary

This report shows the successful application of continuous, asynchronous and distributed code reviews in an industrial context. The applied method was evaluated by the development team and found be be very useful and efficient. Especially the low overhead of the process and the small size of each review item are major factors for the developer acceptance. Also the positive effect for understanding the code and the support for the concept of collective ownership has been shown. The effectiveness has not reached the typical values found in literature. The causes may be investigated further. Still a positive cost/benefit ratio is given.

The comparative rating against the baseline model showed that the preparation and execution effort was rated better with the applied method. Especially the rework effort was considered to be lower because of the short turnaround time. Since both compared approaches are very different in the way the code is examined (individual vs. team) or read (deltas vs. structural item) we conclude that both methods are less comparable in for effectiveness and propose to use a combination where DCR is used to archieve a high review coverage at a relatively low cost and more formal approaches are used relative to the criticality and complexity of one carefully selected review item.

The results from this work are used to the elaborate the process and the tools (as we are actively participating in tool development [8]). Further investigations may include empirical studies on further questions e.g. the application in maintenance processes eventually in a more controlled environment.

Bibliography

[1] IEEE standard for software reviews. Technical report, 1998.

[2] Mario Bernhart, Andreas Mauczka, and Thomas Grechenig. Adopting code reviews for agile software development. In *Proceedings of the 2010 Agile Conference*, AGILE '10, pages 44–47, Washington, DC, USA, 2010. IEEE Computer Society.

[3] Mario Bernhart, Stefan Reiterer, Kilian Matt, Andreas Mauczka, and Thomas Grechenig. A task-based code review process and tool to comply with the do-278/ed-109 standard for air traffic managment software development: An industrial case study. In Taghi M. Khoshgoftaar, editor, *HASE*, pages 182–187. IEEE Computer Society, 2011.

[4] J. Bisbal, D. Lawless, Bing Wu, and J. Grimson. Legacy information systems: issues and directions. 16(5):103–111, 1999.

[5] Barry W. Boehm. Software engineering economics. *Software Engineering, IEEE Transactions on*, SE-10(1):4 –21, jan. 1984.

[6] G. Canfora, A. R. Fasolino, G. Frattolillo, and P. Tramontana. Migrating interactive legacy systems to web services. In *Proc. 10th European Conf. Software Maintenance and Reengineering CSMR 2006*, 2006.

[7] Marcus Ciolkowski, Oliver Laitenberger, and Stefan Biffl. Software reviews: The state of the practice. *IEEE Software*, 20(6):46–51, 2003.

[8] Eclipse Mylyn Reviews Gerrit Connector. http://www.eclipse.org/reviews/gerrit/. Accessed: 2012-10-01.

[9] Radio Technical Commission for Aeronautics (RTCA) and the European Organization for Civil Aviation Equipment (EUROCAE). *DO-278/ED-109: Guidelines for Communication, Navigation, Surveillance,*

and Air Traffic Management (CNS/ATM) Systems Software Integrity Assurance, 2002.

[10] Gerrit in the Adroid Open Source Project. http://source.android.com/source/life-of-a-patch.html. Accessed: 2010-11-09.

[11] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In Michal Young and Premkumar T. Devanbu, editors, *SIGSOFT FSE*, pages 1–11. ACM, 2006.

[12] R. Khadka, G. Reijnders, A. Saeidi, S. Jansen, and J. Hage. A method engineering based legacy to soa migration method. In *Proc. 27th IEEE Int Software Maintenance (ICSM) Conf*, pages 163–172, 2011.

[13] Sami Kollanus and Jussi Koskinen. Survey of software inspection research: 1991-2005. Computer science and information systems reports, working papers (wp-40), University of Jyväskylä, Jyväskylä, Finland, 2007.

[14] Oliver Laitenberger and Jean-Marc DeBaud. An encompassing life cycle centric survey of software inspection. *Journal of Systems and Software*, 50(1):5–31, 2000.

[15] Bertrand Meyer. Design and code reviews in the age of the internet. *Commun. ACM*, 51(9):66–71, 2008.

[16] Jason Remillard. Source code review systems. *IEEE Software*, 22(1):74–77, 2005.

[17] H. Sneed. Integrating legacy software into a service oriented architecture. In *Proc. 10th European Conf. Software Maintenance and Reengineering CSMR 2006*, 2006.

[18] H. Sneed. Migrating from cobol to java. In *Proc. IEEE Int Software Maintenance (ICSM) Conf*, pages 1–7, 2010.

[19] Michael Stein, John Riedl, Sören J. Harner, and Vahid Mashayekhi. A case study of distributed, asynchronous software inspection. In *ICSE*, pages 107–117, 1997.

[20] Stefan Strobl, Mario Bernhart, Thomas Grechenig, and Wolfgang Kleinert. Digging deep: Software reengineering supported by database reverse engineering of a system with 30+ years of legacy. In *ICSM*, pages 407–410, 2009.

[21] Y. Zou. Quality driven software migration of procedural code to object-oriented design. In *Proc. 21st IEEE Int. Conf. ICSM'05 Software Maintenance*, pages 709–713, 2005.