Architecture Improvement Method

Methodical Improvement of Software Systems and –Architectures

Dr. Gernot Starke
http://aim42.org

This paper outlines aim[42], the systematic yet pragmatic approach to improve productive software systems and architectures. aim[42] works iteratively in three phases (analyze, evaluate, improve) supported by crosscutting activities. For each phase, aim[42] proposes a number of proven and established practices and patterns. The method addresses both business and technical stakeholders of software systems. As aim[42] is based upon a small number of core concepts, it is easy to understand. aim[42] does not required tool support and is therefore completely vendor-agnostic.

aim[42] is developed by an active community in open-source style, backed by extensive industrial experience and scientific research. It has proven to work under time and budget constraints in various industries.

1 Introduction

Real-world software systems regularly need to be maintained for various reasons, often under severe budget and time constraints. Software owners and other stakeholders often prioritize new business requirements higher than improvement of internal software quality. Over time, this leads to reduced maintainability, coined “software erosion” or “software entropy”.

Especially in competitive markets, investment in existing software is often driven by short-term business goals. Long-term goals, like maintainability or understandability are neglected. Improvements of software architectures seem to conflict with these short-term business and budget requirements, as break-even is expected within short timespans.

Keeping software maintainable over time requires substantial investment in internal qualities, like conceptual integrity, architectural structures and crosscutting concepts, proper coupling and cohesion.

The systematic approach of aim[42] supports evolution and improvement of software architectures and internal quality. aim[42] helps technical and management decision makers to properly compromise short-term budget requirements with long-term internal architecture quality.

1.2 Conceptual Integrity Requires Attention and Budget

Solving similar problems should be handled in similar, preferably standardized ways within a software architecture. This *conceptual integrity* for things like persistence, transaction handling, encryption or graphical user interface construction provides a solid foundation for maintainability, understandability and longevity of software systems.

Conceptual integrity can easily be violated when software is changed under time and budget constraints. Especially in medium or large teams, communicating the appropriate concepts requires effort in form of documentation or education. Commercial projects often neglect or abandon this communication – resulting in violations of conceptual integrity, risky technological diversion and inconsistent implementation (called software entropy or detoriation) – resulting in severe loss of maintainability and corresponding increase in maintenance and operational cost as well as a prolonged time to market.

1.3 Organize Improvements in Phases and Iterations

aim[42] takes a phased-iterative approach and differentiates between problem identification during the analysis phase, problem evaluation and problem resolution during improvement. Solution approaches or improvements are identified iteratively throughout the phases (see Fig. 1).

Before any improvement activity is undertaken, the related issues and improvement approaches are evaluated with respect to their estimated cost, benefit and risk – actively enabling non-technical stakeholders to argue about priorities and business value.

Fig. 1. Iterative and Phased Model of aim[42]

The underlying terminology will be explained in the improvement domain model in section 3.

1.4 Rely on Established Practices and Patterns

aim[42] relies on a large number of practices and patterns which have been used throughout software industry for several years. Arranging those in the iterative phased model is the innovative idea of the aim[42] method.
In this paper, practice or pattern names are set in capitals, like QUALITATIVE ANALYSIS.

2 An Overview of Methodical Improvement

Our aim[42] approach to architecture and system improvement consists of the following steps:

1. Collect all issues you can identify in a predefined time-box within the system, its operation or development environment and the associated organization. Practices like QUALITATIVE ANALYSIS, SOFTWARE-ARCHEOLOGY or STAKEHOLDER-INTERVIEW support this step. ARCHITECTURAL UNDERSTANDING is an important side-effect of these activities.
2. Evaluate those issues using ESTIMATE-ISSUE-COST with respect to its one-off or recurring cost.
3. Some problems hide their real causes – use ROOT-CAUSE-ANALYSIS to identify those.

4. Together with technically knowledgeable stakeholders, e.g. software architects, COLLECT-OPPORTUNITIES-FOR-IMPROVEMENT, which resolve the problems or causes. Note the potential m:n relationships between issues and improvements: One issue might need more than one improvement, one improvement might solve more than one issue.
5. Improvements also need to be evaluated or estimated with respect to their costs, ESTIMATE-IMPROVEMENT-COST. The benefit (aka *business value*) of a improvement is the cost of the associated problem or problems.
6. The comparison of issue-cost (== benefit) and improvement-cost provides valuable decision support for technical and business stakeholders about which parts of the software architecture or related processes shall be improved.
7. aim[42] works in iterative manner: The evaluation of issues respectively improvements might change over time – reflecting modern development practices. Regular checks of ISSUE-LIST and IMPROVEMENT-BACKLOG ensure their up-to-dateness.

3 An Informal Domain Model for Architecture Improvement

When discussing methodical improvement, a common terminology fosters understanding.

You find the required terms in the adjacend diagram as a simple domain model – showing the most important concepts and their relations.

Please note the m:n relation between Issue and Improvement:

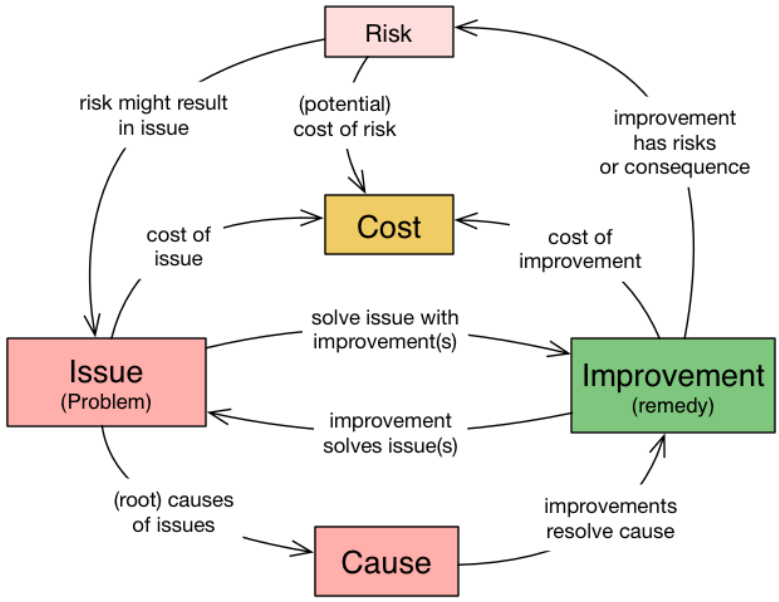

Fig. 2. Software Architecture Improvement Domain Model

A single issue might have more than one opportunities for improvement (or solution), and one improvement might solve one or several issues.

Issue	Might be problem, issue or fault within system or processes related to system (e.g. management, development, administrative or organizational process)
Cause	Fundamental reason for one or several problems.
Improvement	Solution or cure for one or several problems or causes.
Cost (of issue)	The cost (in monetary units, e.g. Euro or $) of the problem, related to a frequency or period of time. For example – cost of every occurrence of problem, or recurring cost per week.
Cost (of improvement)	The cost (in monetary units) of the improvement, tactic or strategy.
Risk	Potential problem or issue. Remedies can have associated risks.

4 Iterative and Phased Model

Improvement should be a continuous process, where identification, evaluation and resolution of problems should be repeated as long as appropriate business value can be created by the improvement. aim42 supports this iterative approach – see fig. 1.

4.1 Analyze: Identify Issues and Create Architectural Understanding

The analyze phase consists of the collection of issues, problems, risks, deficiencies and technical debt within the system and its associated processes – plus creating ARCHITECTURAL-UNDERSTANDING. Focus within this phase lies on finding issues. In addition, one shall develop and document an understanding of internal structures, concepts, architectural approaches and important decisions of the system.

4.2 Evaluate: Estimate Cost and Benefit

During the evaluate phase, the value of issues and improvements in business-related terms, like cost or effort shall be estimated. Such estimation results in comparability of both issues and improvements, leading to transparent prioritization.
From our experience within various industries, domains and technologies, this part of aim42 is most often neglected within software maintenance activities – as it requires a combination of technical and business skills. Estimation needs EXPLICIT-ASSUMPTIONS about the relation between technical problems or remedies with (business-related) effects, mainly cost and effort. Development teams and technical management often dislike to phrase appropriate assumptions.

4.3 Improve: More Than Just Refactoring

Within the improve phase, improvements are applied to the system or associated processes. Rigorous feedback needs to be gathered to collect results of these improvements – due to potential side effects on other issues and improvements.
During improvement of systems, architectural decisions need to be modified or adjusted – resulting for example in changes to data structures, technical infrastructure, required middleware, frameworks or third-party products. Those modifications sometimes require fundamental changes, more then just refactoring.

4.4 Crosscutting Activities: Keep ISSUE-LIST and IMPROVEMENT-BACKLOG

Although aim42 requires no formalism or formal documentation, two deliverables shall be maintained, the ISSUE-LIST and IMPROVEMENT BACKLOG (see Fig. 3). In the sense of lean and agile, those should be kept in the least formal manner acceptable for the given context:

1. ISSUE-LIST: contains all currently identified issues, together with their evaluation and links to appropriate opportunities for improvement.
2. IMPROVEMENT-BACKLOG: contains a list of currently known opportunities for improvement, together with their cost- and effort estimates, potential risks and links to the issues they help to resolve.

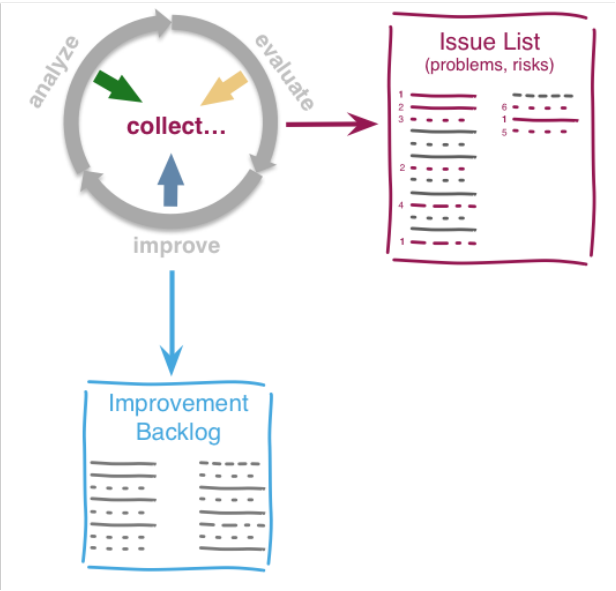

Fig. 3. Iteratively Collect and Evaluate Issues and Improvements

5 Examples of Practices and Patterns

aim[42] currently comprises about 80 practices and patterns. A handful of them shall be applied crosscutting, the rest associated with one of the phases. The aim[42] method guide [1], describing the complete set of practices and patterns, is currently under active development [2]. The following is just a brief excerpt to show the kind and nature of these practices and patterns.

5.1 Patterns and Practices from Analysis Phase

Besides creating ARCHITECTURAL-UNDERSTANDING, the main focus of the analysis phase lies in identification of issues, technical debt or risks within the system, its technical environment, related processes and organizations. To be able to identify such problems, one needs an appropriate amount of understanding of the architectural structures, technical concepts and source code of the system under consideration, gained by STAKEHOLDER-INTERVIEW, VIEW-BASED-UNDERSTANDING or DOCUMENTATION-ANALYSIS.

- QUALITATIVE ANALYSIS: Determine risks of architectural approaches with respect to required quality goals. Described in [5], widely used in practice.
- QUANTATIVE ANALYSIS: systematic analysis of source code metrics, e.g. size, coupling, cohesion, and complexity.
- RUNTIME-ANALYSIS, DATA-ANALYSIS, ISSUE-TRACKER-ANALYSIS and DEVELOPMENT-PROCESS-ANALYSIS.
- PROFILING: measure resource consumption of a system during its operation.
- DEBUGGING: Identify the source of an error or misbehavior by observing the flow of execution of a program in detail. Often helps in understanding the static and dynamic structure of source code.
- SOFTWARE-ARCHEOLOGY: understand software by examining source code and code history.
- ROOT-CAUSE-ANALYSIS: Some problems are only symptoms for underlying causes. In systematic improvement it is often useful to tackle the root cause and the symptom independently.

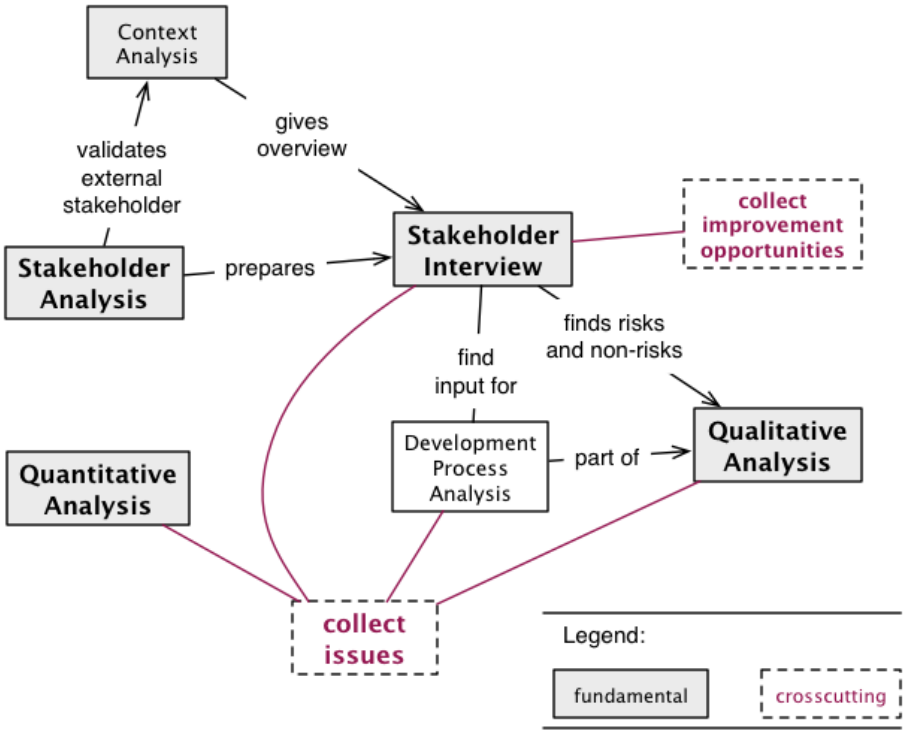

Fig. 4. Overview of the Analyze-Phase

5.2 Practices and Patterns from Evaluation Phase

The main focus of the evaluation phase is to make the problems and remedies comparable by estimating their *value* in terms of business-oriented units – most often money or effort.

- ESTIMATE-IN-INTERVAL: Give a lower and upper bound of your estimate. The difference between the two shows confidence in the estimate. If this difference is relatively small, it shows high confidence
- ESTIMATE-ISSUE-COST: What one-time or recurring cost does this issue generate, either with every occurrence or periodically.
- ESTIMATE-IMPROVEMENT-COST: What cost and/or effort will the improvement require?
- IMPACT-ANALYSIS: What additional impact might an improvement or issue have, e.g. after-effects or side-effects? This is especially important in case of architectural compromises in context of conflicting quality goals.

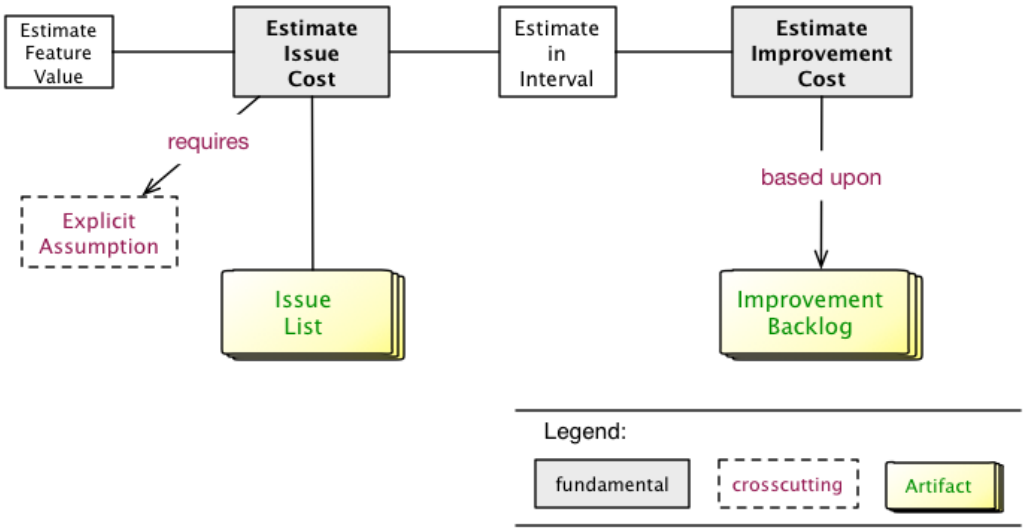

Fig. 5. Overview of the Evaluate-Phase

5.3 Patterns and Practices for Improvement

Plan and coordinate remedies to eliminate or resolve issues found. Apply one or several improvements to clean up code, concepts or processes, reduce cost or optimize quality attributes.

- ANTICORRUPTION-LAYER: Isolate clients from internal changes of sub-systems or modules.
- ASSERTIONS: Verify preconditions to make a program fail when something goes fundamentally wrong.
- AUTOMATED-TESTS: Introduce automated tests (unit-, integration- or smoke-tests) to verify correct runtime behavior. Such tests can assure that improvements or refactoring did not modify (break) desired behavior.
- BRANCH-FOR-IMPROVEMENT: Introduce distinct branches in version control system to reflect improvements.
- FRONTEND-SWITCH: Route front-end requests to either new or old backend systems, depending on their nature.
- INTRODUCE BOY SCOUT RULE: Establish a policy to perform certain structural improvements each time an artifact (source code, configuration, documents etc.) is changed.
- ISOLATE-CHANGES: Introduce interfaces and intra-system borders, so that changes cannot propagate to other areas.
- REFACTORING source code: Apply source code transformation that does not change functionality of system, but leaves the source in better, more understandable or more efficient state.
- REMOVE-NESTED-CONTROL-STRUCTURES: Re-structure code so that deeply nested or complicated control structures are replaced by semantically identical versions
- SAMPLE-FOR-IMPROVEMENT: Provide concrete code example for typical improvement situations, so that developers can improve existing code easily.
- UNTANGLE-CODE: Remove unnecessary complications in code, e.g. nested structures, dependencies, dead-code, duplicate-code etc.

5.4 Crosscutting Patterns and Practices

In software improvement projects, participants should constantly watch out for potential improvements for the issues identified. On the other hand, even when active in the improvement phase, previously unknown issues might occur – which need to be collected.

- COLLECT ISSUES: Maintain a central list or overview of known problems, together with their cost/effort evaluation. Collect in breadth-first manner, estimates will be performed separately.
- COLLECT OPPORTUNITIES FOR IMPROVEMENT: In all aim[42] phases, one should identify remedies for the currently known problems or their causes.
- IMPROVEMENT BACKLOG: A list or collection of remedies and their cost/effort/risk estimation.
- Following the concepts of lean and agile, teams shall try to favor FAST FEEDBACK: The later a lack of quality is identified the higher are the costs to fix it. Teams shall continuously evaluate the quality of work artifacts and immediately take countermeasures or escalate as early as possible.

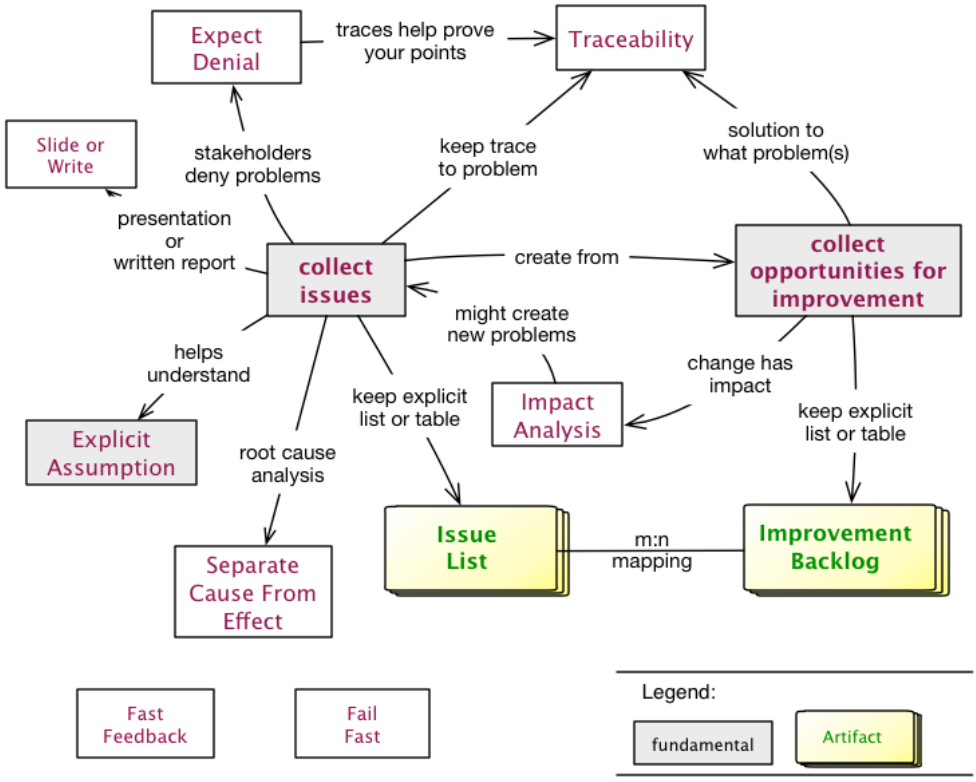

Fig. 6. Overview of Crosscutting Activities and Artifacts

6 Industry Experience

Phased and cost-benefit oriented improvement, as described above, has been successfully applied for several software and software-intensive systems in various industries. Due to non-disclosure restrictions, concrete technical and business details have to be omitted for publication. Nevertheless the main features of the aim[42] method can be recognized and the specific patterns and practices applied within the case studies can be named.

6.1 Facilitate Communication Between IT and Business Stakeholders

As aim[42] effectively maps technical issues (like internal quality, appropriate level of coupling, cohesion, applicability of technical concepts, adherence to conceptual integrity) to business-related entities, especially cost, effort and risk, it addresses both business and technical stakeholders.
Therefore aim[42] reduces terminological friction between those parties, enabling technical stakeholders to explain the need for architectural improvement to non-technical stakeholders, especially those responsible for budget-related decisions. This positive effect mainly results from the common business terms – like cost or effort, which both technical and business stakeholders can easily understand but requires a solid shared foundation of underlying assumptions and trust.

6.2 Multimedia Framework for Automotive Industry

For an international hard- and software provider in automotive, we conducted the aim[42] analysis and evaluation phases. QUALITATIVE ANALYSIS, paired with STAKEHOLDER INTERVIEWS was combined with PROCESS ANALYSIS. Investing approximately 15 person-days in analysis plus 3 days in evaluation lead to a detailed ISSUE-LIST and IMPROVEMENT BACKLOG.
Management stakeholders perceived these phases as highly successful, as the improvement backlog contained several "low hanging fruit": improvements with relatively large positive impact for a minor investment. Examples included the elimination of redundant documentation within the distributed development teams, the unification of domain terminology, sharpening the teams' ubiquitous language and some minor, purely technical improvements in the technical frameworks.

6.3 Telecommunication Billing

For a large European telecommunication provider, we initially conducted the analysis and evaluation phases of aim[42], later included improvement activities. A STRUCTURAL ANALYSIS paired with STAKEHOLDER INTERVIEWS helped discover a severe structural problem. Conducting SOFTWARE-ARCHEOLOGY and VIEW-BASED UNDERSTANDING lead to the first improvement iteration, especially improved documentation.
After the initial aim[42] analysis, the improvement backlog plus new documentation was handed over to a development team that replaced parts of the system with modern technology. Most issues found during aim[42] analysis could therefore be resolved.

6.4 Safety-Critical Infrastructure Software for Food Production

An international hard- and software manufacturer in food industry needed to decrease software development and maintenance cost within its safety critical infrastructure software (e.g. for production machine hygiene supervision, production-lot reporting etc.).
After detailed STATIC ANALYSIS, RUNTIME-ANALYSIS and QUALITATIVE ANALYSIS, DEVELOPMENT PROCESS ANALYSIS plus initial ESTIMATE-ISSUE COST, we performed VIEW-BASED UNDERSTANDING together with the development team. Some REFACTORING lead to minor improvements in code.
ROOT-CAUSE ANALYSIS proved that a major problem resulted from a severe lack of architecture governance. SHEPARDING THE ARCHITECTURE led to significant improvements within the whole organization and the analyzed systems respectively.

Acknowledgements

Thanks to Michael Mahlberg and Oliver Tigges for their helpful reviews.

Contact Information

Email: info@aim42.org Twitter: @arc_improve42 Supported by innoQ Deutschland GmbH, Krischerstr. 100, D-40789 Monheim	Web: • aim42.org • innoq.com – our friendly supporter

References and Further Information

1. aim42 Method Guide: http://aim42.github.io

2. aim42 – options for contribution: http://aim42.org/contribute/index.html

3. Resources for Software Architects – http://arc42.org