HELLO2MORROW

Software Architecture Management

Optimization of Developer Productivity

Monitoring and Assessing Software Quality

Software Risk Assessment and Transparency

Our mission is to help our customers achieve first-class project results by delivering tools and services focused on assessing, monitoring and improving the architecture and technical quality of their software systems in the development and maintenance phases. We believe that these two aspects are the most critical factors in determining the success of medium- and large-scale projects.

www.hello2morrow.com

DZone Refcardz

brought to you by...
HELLO2MORROW

CONTENTS INCLUDE:

Designing Quality Software

Architectural and Technical Best Practices

By Alexander von Zitzewitz

ABSTRACT

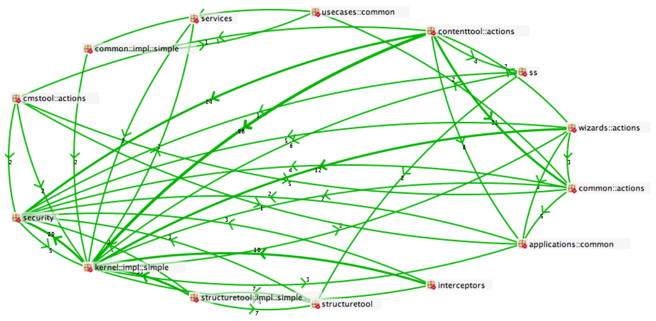
The technical quality of software can be defined as the level of conformance of a software system to a set a set of rules and guidelines derived from common sense and best practices. Those rules should cover software architecture (dependency structure), programming in general, testing and coding style.

Technical quality is fundamentally manifested in the source code. People say: "The truth can only be found in the source code". Therefore it is important that achieving a satisfactory level of technical quality is an explicit goal and integral part of the development process. To avoid a steady decrease of technical quality during development it is required to measure it on a regular base (at least daily). By doing that it is possible to detect and address undesirable rule violations early in the process. The later rule violations are detected the more difficult and expensive it is to fix them. Since testing is only one of several aspects of technical quality management it is not possible to achieve an acceptable level of technical quality by testing only.

The document begins with a description of the biggest enemy of technical quality, which is the structural erosion of software. The best way to fight structural erosion is to keep the large-scale structure of a software system in good shape. Therefore the biggest part of this document focuses on large-scale system design, which also has big implications for application security aspects. Parts of this section are very technical. The intention is to support architects and developers in solving typical day-to-day issues that can negatively impact technical quality and software structure. The last part contains a compact set of rules derived from experience and real-world projects. Implementing and enforcing these rules will help you to achieve a good level of technical quality and maintainability while optimizing the productivity of your development team.

The intended audiences are software architects, developers, quality managers and other technical stakeholders. Although the major part of the document is programming language agnostic, the rule set at the end works best with statically typed object-oriented languages like Java, C# or C++.

STRUCTURAL EROSION



Package cycle groups are a typical symptom of structural erosion

All software projects start with great hope and ambition. Architects and developers are committed to creating an elegant and efficient piece of software that is easy to maintain and fun to work on. Usually, they have a vital image of the intended design in their mind. As the code base gets larger, however, things start to change. The software is increasingly harder to test, understand, maintain and extend. In Robert C. Martin's terms, "The software starts to rot like a piece of bad meat".

This phenomenon is called "Structural Erosion" or "Accumulation of Structural Debt", and it happens in almost every non-trivial software project. Usually, the erosion begins with minor deviations from the intended design due to changes in requirements, time pressure or just simple negligence. In the early stages of a project, this is not a problem; but during the later stages, the structural debt grows much faster than the code base. As a result of this process, it becomes much harder to apply changes to the system without breaking something. Productivity is decreasing significantly and the cost of change grows continuously up to a point where it becomes unbearable.

Robert C. Martin described a couple of well-known symptoms that can help you to figure out whether or not your application is affected by structural erosion:

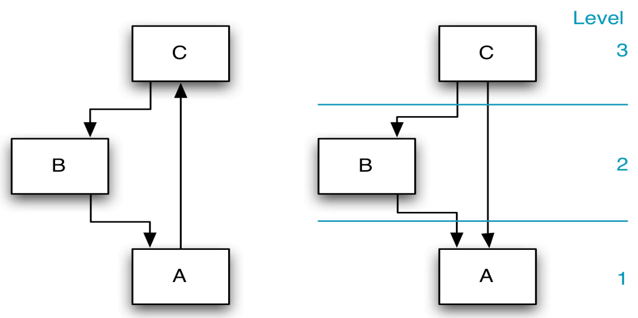
- Rigidity: the system is hard to change because every change forces many other changes.
- Fragility: changes cause the system to break in conceptually unrelated places.
- Immobility: it's hard to disentangle the system into reusable components.
- Viscosity: doing things correctly is harder than doing things incorrectly.
- Opacity: the code is hard to read and understand. It does not express its intent well.

You would probably agree that those symptoms affect most non-trivial software systems in one way or another. Moreover, the symptoms get more severe the older a system is and the more people are working on it. The only way to avoid them in the first place is to have a battle plan against structural erosion integrated into the daily development process.

LARGE-SCALE SYSTEM DESIGN

Dependency Management

The large-scale design of a software system is manifested by its dependency structure. Only by explicitly managing dependencies over the complete software lifecycle is it possible to avoid the negative side effects of structural erosion. One important aspect of dependency management is to avoid cyclic compile-time dependencies between software components:



Case 1: Cyclic Dependencies   Case 2: Acyclic Dependencies

Case 1 shows a cyclic dependency between units A, B and C. Hence, it is not possible to assign level numbers to the units, leading to the following undesirable consequences:

- Understanding the functionality behind a unit is only possible by understanding all units.
- The test of a single unit implies the test of all units.
- Reuse is limited to only one alternative: to reuse all units. This kind of tight coupling is one of the reasons why reuse of software components is hardly ever practiced.
- Fixing an error in one unit involves automatically the whole group of the three units.
- An impact analysis of planned changes is difficult.

Case 2 represents three units forming an acyclic directed dependency graph. It is now possible to assign level numbers. The following effects are the consequences:

- A clear understanding of the units is achieved by having a clear order, first A, then B and then C.
- A clear testing order is obvious: first test unit A; test continues with B and afterwards with C.
- In matter of reuse, it is possible to reuse A isolated, A and B, or also the complete solution.
- To fix a problem in unit A, it can be tested in isolation, whereby the test verifies that the error is actually repaired. For testing unit B, only units B and A are needed. Subsequently, real integration tests can be done.
- An impact analysis can easily be done.

Please keep in mind that this is a very simple example. Many software systems have hundreds of units. The more units you have, the more important it becomes to be able to levelize the dependency graph. Otherwise, maintenance becomes a nightmare.

Hot Tip

Here is what recognized software architecture experts say about dependency management:

"It is the dependency architecture that is degrading, and with it the ability of the software to be maintained." [ASD]

"The dependencies between packages must not form cycles." [ASD]

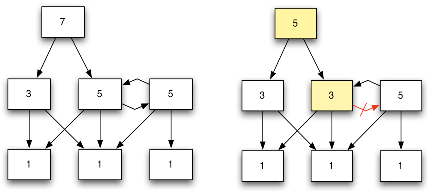
"Guideline: No Cycles between Packages. If a group of packages have cyclic dependencies then they may need to be treated as one larger package in terms of a release unit. This is undesirable because releasing larger packages (or package aggregates) increases the likelihood of affecting something." [AUP]

"Cyclic physical dependencies among components inhibit understanding, testing and reuse." [LSD]

Coupling Metrics

Another important goal of dependency management is to minimize the overall coupling between different parts of the software. Lower coupling means higher flexibility, better testability, better maintainability and better comprehensibility. Moreover, lower coupling also means that changes only affect a smaller part of an application, which greatly reduces the probability for regression bugs.

To control coupling, it is necessary to measure it. [LSD] describes two useful coupling metrics. Average Component Dependency (ACD) is telling us on how many components a randomly picked component will depend upon on average (including itself). Normalized Cumulative Component Dependency (NCCD) is comparing the coupling of a dependency graph (application) with the coupling of a balanced binary tree.



Graph 1 (CCD=23)   Graph 2 (CCD=19)

Above, you see two dependency graphs. The numbers inside of the components reflect the number of components reachable from the given component (including itself). The value is called Component Dependency (CD). If you add up all the numbers in the Graph 1 the sum is 23. This value is called "Cumulative Component Dependency" (CCD). If you divide CCD by the number of components in the graph, you get ACD. For Graph 1, this value would be 3.29.

Please note that Graph 1 contains a cyclic dependency. In Graph 2, removing the dependency shown in red has broken the cycle, which reduces the CCD to 19 and ACD to 2.71. As you can see, breaking cycles definitely helps to achieve our second goal, which is the overall reduction of coupling.

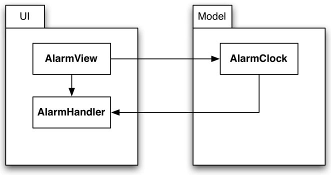
NCCD is calculated by dividing the CCD value of a dependency graph through the CCD value of a balanced binary tree with the same number of nodes. Its advantage over ACD is that the metric value does not need to be put in relation to the number of nodes in the graph. An ACD of 50 is high for a system with 100 elements but quite low for a system with 1,000 elements.

Detecting and Breaking Cyclic Dependencies

Agreeing that it is a good idea to avoid cyclic compile-time dependencies is one thing. Finding and breaking them is another story.

The only real option to find them is to use a dependency analysis tool. For Java, there is a simple free tool called "JDepend" [JDP]. If your project is not very big, you can also use the free "Community Edition" of "SonarJ" [SON], which is much more powerful than JDepend. For bigger projects you need to buy a commercial license of SonarJ. If you are not using Java or look for more sophisticated features like cycle visualization and breakup proposals, you will have to look at commercial tools.

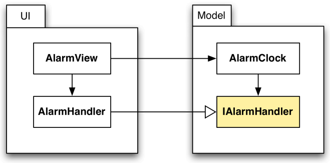
After having found a cyclic dependency, you have to decide how to break it. Code refactorings can break any cyclic compile-time dependency between components. The most frequently used refactoring to do that is the addition of an interface. The following example shows an undesirable cyclic dependency between the "UI" component and the "Model" component of an application:



Cyclic dependency between "UI" and "Model"

The example above shows a cyclic dependency between "UI" and "Model".Now it is not possible to compile, use, test or understand the "Model" component without also having access to the "UI" component. Note that even though there is a cyclic dependency on the component level, there is no cyclic dependency on the type level.

Adding the interface "IAlarmHander" to the "Model" component solves the problem, as shown in the next diagram:

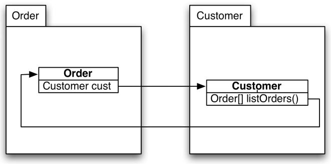


Cyclic dependency resolved by adding an interface

Now, the class "AlarmHandler" simply implements the interface defined in the "Model" component. The direction of the dependency is inverted by replacing a "uses" dependency with an inverted "implements" dependency. That is why this technique is also called the "dependency inversion principle", first described by Robert C. Martin [ASD]. Now, it is possible to compile, test and comprehend the "Model" component in isolation. Moreover, it is possible to reuse the component by just implementing the "IAlarmHandler" interface. Please note that even if this method works pretty well most of the time, the overuse of interfaces and callbacks can also have undesirable side effects like added complexity. Therefore, the next example shows another way to break cycles. In [LSD], you will find several additional programming techniques to break cyclic dependencies.

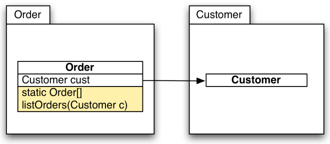
Hot Tip In C++, you can mimic interfaces by writing a class that contains pure virtual functions only.

Sometimes, you can break cycles by rearranging features of classes. The following diagram shows a typical case:



Another case of a cyclic dependency

The "Order" class references the "Customer" class. The "Customer" class also references the "Order" class over the return value of a convenience method "listOrders()". Since both classes are in different packages, this creates an undesirable cyclic package dependency.



Problem solved by moving a method

The problem is solved by moving the convenience method to the "Order" class (while converting it into a static method). In situations like this, it is helpful to levelize the components involved in the cycle. In the example, it is quite natural to assume that an order is a higher-level object than a customer. Orders need to know the customer, but customers do not need orders. As soon as levels are established, you simply need to cut all dependencies from lower-level objects to higher-level objects. In our example, that is the dependency from "Customer" to "Order".

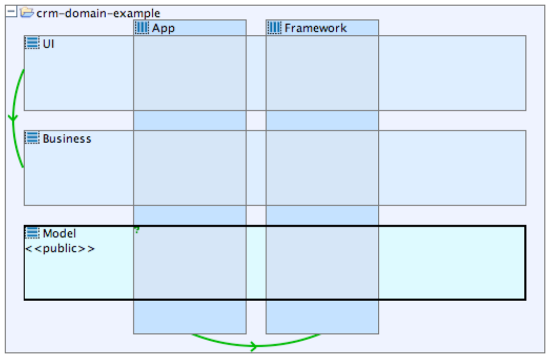
It is important to mention that we do not look at runtime (dynamic) dependencies here. For the purpose of large-scale system design, only compile-time (static) dependencies are relevant.

Hot Tip The usage of Inversion of Control (IOC) frameworks like the Spring Framework [SPG] will make it much easier to avoid cyclic dependencies and to reduce coupling.

Logical Architecture

Actively managing dependencies requires the definition of a logical architecture for a software system. A logical architecture groups the physical (programming language) level elements like classes, interfaces or packages (directories or name spaces in C# and C++) into higher-level architectural artifacts like layers, subsystems or vertical slices.

A logical architecture defines those artifacts, the mapping of physical elements (types, packages, etc.) to those artifacts and the allowed and forbidden dependencies between the architectural artifacts.



Example of a logical architecture with layers and slices

Here is a list of architectural artifacts you can use to describe the logical architecture of your application:

Layer	You cut your application into horizontal slices (layers) by using technical criteria. Typical layer names would be "User Interface", "Service", "DAO", etc.
Vertical slice	While many applications use horizontal layering, most software architects neglect the clear definition of vertical slices. Functional aspects should determine the vertical organization of your application. Typical slice names would be "Customer", "Contract", "Framework", etc.
Subsystem	A subsystem is the smallest of the architectural artifacts. It groups together all types implementing a specific mostly technical functionality. Typical subsystem names would be "Logging", "Authentication", etc. Subsystems can be nested in layers and slices.
Natural Subsystem	The intersection between a layer and a slice is called a natural subsystem.
Subproject	Sometimes projects can be grouped into several inter-related subprojects. Subprojects are useful to organize a large project on the highest level of abstraction. It is recommended not to have more than seven to ten subprojects in a project.

You can nest layers and slices, if necessary. However, for reasons of simplicity, it is not recommended using more than one level of nesting.

Mapping of code to architectural artifacts

To simplify code navigation and the mapping of physical entities (types, classes, packages) to architectural artifacts, it is highly recommended to use a strict naming convention for packages (namespaces or directories in C++ or C#). A proven best practice is to embed the name of architectural artifacts in the package name.

For example, you could use the following naming convention:

```
com.company.project.[subproject].slice.layer.[subsystem]…
```

Parts in square brackets are optional. For subsystems not belonging to any layer or slice, you can use:

```
com.company.project.[subproject].subsystem…
```

Of course, you need to adapt this naming convention if you use nesting of layers or slices.

Hot Tip

Dangerous Attitude: "If it ain't broken, don't fix it!" Critics of dependency and quality management usually use the above statement to portray active dependency and quality management as a waste of time and money. Their argumentation is that there is no immediate benefit in spending time and resources to fix rule violations just for improving the inner quality of an application. It is hard to argue against that if you have a very short-time horizon. But if you expand the time horizon to the lifetime of an application, technical quality is the most important factor driving developer productivity and maintenance cost. This shortsighted thinking is one of the major reasons why so many medium- to large-scale applications are so hard to maintain. Many costly project failures can also be clearly associated with lack of technical quality.

Application Security Aspects

Most people don't think about the connection between application security and the architecture (dependency structure) of an application. But experience shows that potential security vulnerabilities are much more frequent in applications that suffer from structural erosion. The reason for that is quite obvious: if the dependency structure is broken and full of cycles, it is much harder to follow the flow of tainted data (un-trusted data coming from the outside) inside of the application. Therefore, it is also much harder to verify whether or not these data have been properly validated before they are being processed by the application.

On the other hand, if your application has a well-defined logical architecture that is reflected by the code, you can combine architectural and security aspects by designating architectural elements as safe or unsafe. "Safe" means that no tainted data are allowed within this particular artifact. "Unsafe" means that data flowing through the artifact is potentially tainted. To make an element safe, you need to ensure two things:

- The safe element should not call any API's that return potentially tainted data (IO, database access, HTTP session access etc.). If this should be necessary for any reason all data returned by those API's must be validated.
- All entry points must be protected by data validation.

This is much easier to check and enforce (with a dependency management tool) than having to assume that the whole code base is potentially unsafe. The dependency management tool plays an important role in ensuring the safety of an element by verifying that all incoming dependencies only use the official entry points. Incoming dependencies bypassing those entry points would be marked as violations.

Of course, the actual data processing should only be done in "safe" architectural elements. Typically, you would consider the Web layer as "unsafe", while the layers containing the business logic should all be "safe" layers.

Since many applications are suffering from more or less severe structural erosion, it is quite difficult to harden them against potential security threats. In that case, you can either try to reduce the structural erosion and create a "safe" processing kernel using a dependency management tool or rely on expensive commercial software security analysis tools specialized on finding potential vulnerabilities. While the first approach will cost you more time and effort in the short term, it will pay off nicely by actually improving the maintainability and security of the code. The second approach is more like a short-term patch that does not resolve the underlying cause, which is the structural erosion of the code base.

COMMON SENSE RULES

The best way to achieve a high level of technical quality is the combination of a small set of rules and an automated-tool-based approach to rule checking and enforcement (see rule T2 for recommendations. In general, the rules should be checked automatically at least during the nightly build. If possible, a rule checker should also be part of the developer environment, so that developers can detect rule violations even before committing changes to the VCS [SON].

The recommended set of rules is, therefore, minimalistic by intention and can be customized when needed. Experience shows that it is always a good idea to keep the number of rules small because that makes it much easier to check and enforce the rules in the development process. The more rules you add, the less additional benefit will be provided by each additional rule. The rule set presented here is based on common sense and experience and already has been successfully implemented by many software development teams.

Unfortunately this document does not leave enough space to explain the rules in more detail. Please refer to the reference section at the end for more information. Some of the rules might seem arbitrary. In that case you can assume that they are derived from common sense and best practices. And of course you are free to adjust thresholds and rules to better match your specific environment.

Rules fall into three priority classes:

Major Rule	Must always be followed.
Minor Rule	It is highly recommended to follow this rule. If this is not possible or desirable you must document the reason.
Guideline	It is recommended to follow this rule.

DESIGN RULES

These rules are covering large-scale architectural aspects of the system.

Major Rules

D1: Define a cycle free logical architecture for your application
Only by having a well-defined and cycle-free application can you have a chance to avoid structural erosion in the first place.

D2: Define a strict and clear naming convention for types and packages based on your logical architecture
The naming convention also defines the mapping between your code and the logical architecture and will greatly simplify the code navigation and comprehension. In C++ or C# you should replace package with namespace or directory.

D3: The code must respect the logical architecture
This rule is ideally enforced by a tool. Basically, the tool has to ensure that all dependencies in your application conform to the logical architecture defined in D1.

D4: Package dependencies must not form cycles
The undesirable effects of cyclic dependencies have been discussed in detail before.

D5: NCCD of compilation units must not be bigger than 7
This rule corresponds with our goal to keep coupling small. If this value grows over the threshold, you should isolate layers and subsystem by only letting them have interfaces as entry points. Breaking cyclic dependencies can also shrink this metric considerably.

Minor Rules

D6: Keep security aspects in mind when creating a logical architecture
Plan for application security from the beginning. Designate “safe” and “unsafe” (data are potentially tainted) architectural elements. Keep the boundary between safe and unsafe elements as narrow as possible so that it is easy to verify that all incoming data are validated properly.

D7: Separate technical aspects from domain aspects on the logical architecture level
Separating these two aspects is the most promising approach to maintain healthy software. Technical aspects may shift in the near future. Business abstractions and their related logic are more likely to be stable. The Spring Framework implements a very good approach to separate business aspects from technical aspects [SPG].

D8: Use consistent handling of exceptions
Exception handling should be done in a consistent way by having answers for basic questions like “What are exceptions?”, “What information about errors should be written and where to?”. Low-level exceptions should not be visible in non-technical layers. Instead, they should be semantically transposed corresponding to their level. This can also prevent tight coupling to implementation details.

Guidlines

D9: Dependencies between compilation units must not form cycles
The dependencies must not form cycles. A general discussion is provided in [LSD].

D10: Use design patterns and architectural styles
Design patterns and architectural styles reuse proven and tested concepts. Design patterns also establish a standardized language for common design situations. Therefore, the use of design patterns is highly recommended where possible and useful [DES].

D11: Do not reinvent the wheel
Use existing designs and implementations where possible. Sometimes it is not obvious at first sight how many errors you can produce with your own implementation. Every line of code not written is a criterion of quality of the system and makes maintenance easier.

PROGRAMMING RULES

Major Rules

P1: Use a consistent formatting and naming scheme
A consistent format for the source code contributes to readability and its maintenance. It is recommended to use a tool that automatically formats source code. Modern development environments support that out of the box. Classes, interfaces, methods and so forth should follow a consistent naming scheme. Source code should be readable on any platform; therefore, use spaces instead of tabs.

P2: Declare class and instance variables as private
All modifiable or non-primitive class and instance variables are to be defined as private. This enhances the separation between interface and implementation [LSD].

Minor Rules

P3: Never catch “Throwable” or “Error” (Java)
To catch exceptions of type “Throwable” and “Error” (including subclasses) violates the basic idea of the design of J2SE. Only provide exception handling for the type Exception.

P4: Avoid empty catch blocks
Empty catch blocks inhibit a useful error handling. At a minimum, a comment and perhaps a configurable log output is required in situations where it is uncritical if the specified exception is caught. The system should remain in a legal state.

P5: Limit the access to types and methods
To declare all types and methods as public is easy but maybe not what you want. Only make types and methods visible if they are supposed to be seen from the outside [LSD].

P6: Restrict extendibility - use final for types and methods (Java, C#)
The final keyword states that the class is not to be intended for sub-classing. In the case of methods, it is clear that they should not be overwritten. By default, everything should be final. Make everything final unless you explicitly want to allow overriding of behavior by sub-classing.

P7: Provide a minimal documentation for types
Focus on the description of the responsibilities of types. If it is possible to easily and precisely phrase the responsibilities, then this is a clear indicator for an adequate abstraction. See also the “Single Responsibility Principle” [ASD].

P8: Number of types in a package must not exceed 50
Grouping types together with somehow related responsibilities helps maintaining a clear physical structure. A package is a cohesive unit of physical design with an overall responsibility. Overloaded packages have a good chance to cause excessive cycles in the physical design.

P9: Lines of code (compilation unit) must not exceed 700
Large compilation units are hard to maintain. Furthermore, they often violate the idea of clear abstractions and lead to significantly increased coupling.

P10: Number of method parameters must not exceed 7
A high number of method parameters may be an indicator of procedural design. The pure number of possible parameter combinations may result in complex method implementations.

P11: Cyclomatic Complexity must not exceed 20
The Cyclomatic Complexity (CCN) specifies the possible control paths through a method. If a method has a lower CCN, it is easier to understand and to test. See [CCN] for formal definition.

P12: Use assertions
Use “assert” (Debug.Assert for C#) in order to ensure preconditions, post-conditions and invariants in the “Design by contract” style [TOS]. It is also important to verify that assertions are never used to validate data coming from the user or from external systems.

TEST AND ENVIRONMENT RULES

Major Rules

T1: Use a version control system
This rule should speak for itself. It is impossible to write reliable software without being able to track changes and synchronize changes.

T2: Set up a build server and measure rule compliance
Building your system should be possible completely and independently from your IDE. For Java, we recommend the use of Maven, Ivy or ANT. Integrate as many rule-checkers as possible into your build script, so that the rules mentioned here can be checked completely automatically. Structural checks have higher priority than other checks because structural problems are much harder to repair once they spread over your application. Ideally severe rule violations should break the build.

A popular recommendation for setting up an automated build environment is the usage of the Hudson build server [HUD] together with Sonar [SNR] and SonarJ [SON]. Hudson is programming language agnostic while Sonar is currently expanding its support for other languages. A free SonarJ plug-in is available for Sonar.

T3: Write unit tests together with your code
Additionally, make sure that all unit tests are at least executed during the nightly build, ideally with every build. This way, you get early feedback when changes lead to regression bugs. While executing the tests, test tools usually also measure your test coverage. Make sure that all complex parts of your application are covered by tests.

T4: Define tests based on the logical architecture
Test design should consider the overall logical architecture. The creation of unit tests for all "Data Transfer Objects" instead of testing classes that provide business logic is useless. A project should establish clear rules on what has to be tested as a minimum instead of doing "blind" test creation. Recommend rules are:

- Provide unit tests for all business related objects. We want to test the business logic in isolation.
- Provide unit tests for published interfaces.

The overall goal is to have good direct and indirect test coverage.

Minor Rules

T5: Use collaboration tools like issue trackers and wikis
Use an issue tracker to track problems and planned changes. Document all major design concepts and abstractions of your application in a wiki.

CONCLUSION

If you are beginning a new project, work on an existing project, or wanting to improve the development process in your organization, this Refcard is meant to be a good starting point. You can expect significant improvement with regard to developer productivity, application maintainability and technical quality, if you implement and enforce the majority of the rules described above. Although this will cost you effort in the beginning, the overall savings are much bigger than the initial effort. Therefore, the adoption of design and quality rules is not only "nice to have" but also mandatory for every professional software development organization.

References

[ASD] Agile Software Development, Robert C. Martin, Prentice Hall 2003

[AUP] Applying UML And Patterns, Craig Larman, Prentice Hall 2002

[LSD] Large-Scale C++ Software Design, John Lakos, Addison-Wesley 1996

[DES] Design Patterns, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley 1994

[TOS] Testing Object-Oriented Systems, Beizer, Addison-Wesley 2000

[JDP] http://www.clarkware.com/software/JDepend.html

[SON] http://www.hello2morrow.com/products/sonarj

[SPG] http://www.springsource.org

[CCN] http://en.wikipedia.org/wiki/Cyclomatic_complexity

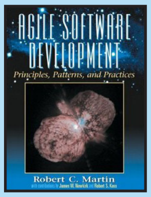
[HUD] http://hudson-ci.org

[SNR] http://www.sonarsource.org

ABOUT THE AUTHOR

Alexander von Zitzewitz is the founder, managing director of hello2morrow GmbH and CEO of the US subsidiary. He has more than 20 years of project and management experience. In 1993 he founded ootec—a company focused on project services around object-oriented software technology. During this time, he worked as lead architect on several medium to large C++ and Java projects. This company was sold to the French Valtech group in March 2000 and is serving customers like Siemens, BMW, Thyssen-Krupp-Stahl and other well-known names in German industry. From 2003 to early 2005, he was working as Director of Central Europe for a French software vendor. In early 2005, he founded hello2morrow in Germany with the vision to create a new product for managing architecture and technical quality of software systems written in Java. The first version of this product called "SonarJ" was released in late summer 2005. Since the summer of 2008, he has been living in Massachusetts. His areas of expertise are object-oriented system design, integrating technical quality into software development processes and large-scale system architecture. Alexander has a degree in Computer Science from the Technical University of Munich.

RECOMMENDED BOOK


AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin

Written by a software developer for software developers, this book is a unique collection of the latest software development methods. The author includes OOD, UML, Design Patterns, Agile and XP methods with a detailed description of a complete software design for reusable programs in C++ and Java. Using a practical, problem-solving approach, it shows how to develop an object-oriented application—from the early stages of analysis, through the low-level design and into the implementation. Walks readers through the designer's thoughts — showing the errors, blind alleys, and creative insights that occur throughout the software design process.

ISBN-13: 978-1-936502-03-5
ISBN-10: 1-936502-03-8
50795
9 781936 502035
$7.95

Version 1.0