



Technical and Economical Model

Modèle technique et économique

Workpackage: 2.1

November 8, 2010

This deliverable is available as a free download.

Copyright © 2010, 2009 by F. Balmas, F. Bellingard, S. Denier, S. Ducasse, J. Laval, K. Mordal-Manet.

The contents of this deliverable are protected under Creative Commons Attribution-Noncommercial-ShareAlike 3.0 Unported license.

You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Noncommercial. You may not use this work for commercial purposes.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page: creativecommons.org/licenses/by-sa/3.0/
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.



Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license):

<http://creativecommons.org/licenses/by-nc-sa/3.0/legalcode>

Second Edition, October, 2010.

Workpackage: 2.1

Title: Model and Definition, First Version

Revision: 1.0

Authors: S. Denier, J. Laval, S. Ducasse, F. Bellingard

Planning

- Delivery Date: 15 October 2010
- First Version: 29 March 2010

Contents

1	Introduction	5
1.1	Objective	5
1.2	Challenges	6
2	Model for remediation effort	8
2.1	Remediation cost	8
2.2	Practices with computed remediation	8
2.3	Practices without computed remediation	12
3	Unit Remediation Change Cost	14
3.1	Elementary costs.	14
3.2	Code Change Costs	14

1. Introduction

1.1 Objective

The objective of this workpackage is to define a model for (i) assessing the effort of software modification, (ii) identifying healing actions following practices from the Squale quality model defined in the previous workpackage (WP1.3). It defines the input for the next workpackage which is about planning actions once their effort is characterized.

The key constraints of this work are:

- The remediation effort should be expressed in the context of a quality model. It should act as a quantification of the work to obtain a better (or good) quality of the entity under analysis.
- The remediation should be based on practices as described in the workpackage 1.3.

Scope. The Squale quality model presented in workpackage 1.3 distinguishes between five kinds of practices: metric, model, rules checking, human, and dynamic-based practices. This distinction is based on different procedures for assessment:

- the metric analysis uses metrics computed on the static structure of the source code;
- the model analysis uses metrics computed on abstract specifications such as UML diagrams and specification documents;
- the "rules checking" analysis relies on rule checking engines which compute rule transgressions at the level of the complete project;
- the human analysis implies project-wide audits made by experts. Audits cover project specifications and process but also any non automatically computable property;
- the dynamic analysis relies on measuring dynamic aspects of the software system through tests.

The different kinds of practices also imply different kinds of healing actions and, consequently, different kinds of assessment for needed modifications and their costs.

- metric-based practices involve assessment and modifications of source code;
- model-based practices involve assessment and modifications of models;
- rules checking practices involve modifications of source code to conform with the rules;

- human-based practices involve applications of recommendations by experts in the project;
- dynamic-based practices involve assessment and modification of test suites.

In this report, we focus on metric-based practices because they can be computed automatically yet each must be resolved in custom ways.

1.2 Challenges

In the context of the Squale quality model, two questions drive remediation effort assessment:

1. Which *context-sensitive* characteristics of the system impact the remediation effort?
2. What *goal* should attain the healing action?

The *context* assesses the component targeted by the practice as well as linked components which may be touched by the actions. Depending on the practice, two parameters are helpful to refine the context:

1. complexity *i.e.*, how easy is it to change components according to the action (for example, a complex method is harder to split than a simple method);
2. impact *i.e.*, how many components are to be touched by the action (for example, changing a method might require updating call sites of the method). Even if source code refactoring engines ensure behavior preservation, this is still relevant when tests should be rerun.

The *goal* represents a level of quality to be reached. In the context of Squale, quality is given by scores so it is natural that goals are simply score goal to be reached. A score above 2 is satisfactory, between 1 and 2 should be raised whenever possible, below 1 should be raised in top priority.

In the end, remediation is a compromise between goals to be achieved and resources available which allows one to respond to the effort required by the remediation. Work package 2.2 builds on this work package to propose different strategies for achieving this compromise.

The formula to compute a remediation effort for a practice should take into account the metrics used to compute the practice score. Then quality scores and efforts are correlated. It is an important property which allows a remediation effort to measure the level of work necessary to achieve a goal score.

However, practice formulae are not necessarily written to assess the *complexity* of a situation, but merely detect problems. Take for example the practice *Number of methods* which is computed using *NOM* and $v(G)$ metrics. The remediation action is to split the class so that methods are dispatched between smaller classes. However, the difficulty of splitting classes might be better assessed using cohesion metrics (the more cohesive a class is, the more difficult it is to split it) than using *NOM* or $v(G)$.

In addition, we can not compute the full effort of improving a practice since it often relies on *design decisions*. Splitting a class (so as to improve a practice such as *Number of methods*) requires thinking ahead about the number of new classes to create and their interaction. Instead, the remediation cost is a tentative cost given to assess the effort to reach a quality goal.

2. Model for remediation effort

2.1 Remediation cost

The remediation cost (notation W) for a practice on a component is based on the empirical formulae of practices: the objective is to give a cost assessment, based on the complexity of the case and the impact of the corrective measures. It typically involves an estimated number of corrections necessary and the weight of the refactoring actions.

Four kinds of parameters are used to compute remediation:

- Complexity: assessment of the characteristics of the faulty component which may hamper remediation;
- Impact: number of components touched by the remediation (to be fixed or to be modified as part of the solution);
- Goal score: quality score expected after remediation;
- Unit Remediation Cost (URC): unit factor of effort which depends on the changes to apply (see Section 3).

Notice that the four parameters are not necessarily independent of each other. For example, assessing the complexity of the case depends on the goal score. The higher the goal score is, the more complex it can be to solve the case.

Since practices often identify situations where design was lacking and that such situation may be due several factors, capturing a cost for fixing the problems is definitively a challenge. In particular multiple solutions may exist and it is difficult to capture the tradeoffs and the knowledge that an expert might apply. Therefore the remediation cost is by essence an estimate that should be confirmed and refined by an expert.

We classify practices on two categories:

- Practices with computed remediation: the remediation can be automatically computed.
- Practices without computed remediation: they represent informational number. It is not possible to propose an automated solution for remediation.

2.2 Practices with computed remediation

For some practices, a remediation cost can be automatically computed.

Comment Rate. The complexity for remediation of the comments rate practice closely follows the practice assessment formula. It depends on the cyclomatic complexity $v(G)$ and the goal score. From the complexity an expected comment rate is computed. The remediation cost is based on the difference between the current number of comment lines and the target number of comment lines computed with the expected comment rate. There is no impact when changing comments.

Concerns: However, the complexity of remediation depends on factors external to the source code, such as developer knowledge of the code. This can be taken into account with an empirical *recallFactor*, especially for legacy projects.

$$complexity = goal_{score} \times (1 - 10^{-v(G)/15})$$

$$expectedCommentRate = \frac{complexity}{9-complexity}$$

$E_{goal} = URC \times SLOC \times expectedCommentRate$, absolute effort as the expected number of comment lines

$$E_{current} = URC \times SLOC \times currentCommentRate$$

$$URC = edit$$

- The URC can be multiplied by an empirical *recallFactor* depending on the current code knowledge by the development team

$W = E_{goal} - E_{current}$, remediation cost to enhance the given practice for the faulty component from its current state to the goal score.

Number of methods. A typical remediation is to split the class into smaller classes. The number of new classes depends on the goal of the practice: we count as many new classes so that each has no more methods than the NOM_{max} computed from the $goal_{score}$.

Concerns: Correcting this practice also depends on the cohesion of the class. The more cohesive is the class (*i.e.*, the tighter the coupling between methods), the more difficult it is to split the class. LCC could be used by an expert to assess the case.

Another concern is whether the class split touches the public API. That is, whether the split keeps the public API in the (smaller) origin class or whether the split scatters the public API across the new classes. In the second case, we also compute the cost of updating call sites to the public API.

First, given the class metrics, a maximal number of methods is computed for the class to achieve the goal score (following the practice formula).

- if $\sum v(G) \geq 80$ then $NOM_{max} = 30 - 10 \times \log_2(goal_{score})$
- if $\sum v(G) \geq 50$ and $NOM \geq 15$ then $NOM_{max} = 20 - 30 \times (goal_{score} - 2)$
- if $\sum v(G) \geq 30$ then $NOM_{max} = 15 - 15 \times (goal_{score} - 3)$

Once NOM_{max} is set, we can compute the complexity and the cost based on splitting the initial class into a number of new classes:

- Complexity: $numberOfNewClasses = NOM/NOM_{max}$
- Impact: $numberOfCallSites$, number of call site updates to follow the API split between new classes (minimum 1)
- URC1: *SplitClass*; URC2: *UpdateCallSite*

- $W = URC1 \times numberOfNewClasses + URC2 \times numberOfCallSites$

Notice that as we split the class into smaller entities, we can not predict the responsibility of each new class and in particular what will be their cyclomatic complexity. Hence, each new class will get a different quality score (sill better than the initial class). The $goal_{score}$ looks like more of a quality parameter than an expected score outcome. This note holds for many of the below remediation practice, as they involve the creation of new entities with hard to predict parameters.

Method size. The remediation for this practice is to split the long method into smaller methods. The number of new methods depends on the $goal_{score}$ of the practice: we count as many new methods so that each is no longer than the max number of $SLOC$ to achieve $goal_{score}$.

Concerns: applying this remediation may induce a defect in the *Number of methods* practice. This problem could be addressed by remediation strategies optimizing the remediation plan.

- $SLOC_{max} = 70 - 21 \times \log_2(goal_{score})$
- $numberOfNewMethods = SLOC / SLOC_{max}$
- $complexity = v(G) \times numberOfNewMethods$
- No impact unless some new methods need to be moved to external classes
- URC: *ExtractMethod*
- $W = URC \times complexity$

Class cohesion. This practice qualifies the relations between methods inside a class. If the cohesion is low, one solution is to split the class in part with high cohesion.

Concerns: a problem is that when class does not define enough behavior it may be tempting to split the class because state is not considered as used together. Such a practice may lead to even smaller data classes which is not a good design either. Often it is important to identify if “Move Behavior Close to the Data” is applicable [DDN02], since data classes often gets their behavior distributed in their clients.

- Complexity: $numberOfNewClasses = [LCOM2]$, since LCOM2 gives the number of connected components in the graph
- Impact: $numberOfCallSites$, number of call site updates to follow the API split between new classes (minimum 1)
- URC1: *SplitClass*; URC2: *UpdateCallSite*
- $W = URC1 \times numberOfNewClasses + URC2 \times numberOfCallSites$

The complexity does not depend on a $goal_{score}$ because the practice has a discrete notation system (see Workpackage 1.3). Instead, we consider a remediation which is both maximal and easy by splitting the class into its connected components.

Swiss Army Knife. This practice aims at detecting *Swiss Army Knife* classes (classes with too many responsibilities or collaborators). The solution is to break the big utility class into smaller classes which are more cohesive and specific to a concern. In the process, the developer may identify utility methods which can be extracted from utility class as regular instance methods.

Concerns: Fixing this kind of class may not be simple, since it has probably many clients. Identifying specific groups of coherent collaborators may also be a challenge.

- Complexity: $numberOfNewClasses = maxOf$
 - $Ca/20$
 - $lcom2/50$
 - $rfe/30$
- Impact: $numberOfCallSites$, number of call site updates to follow the API split between new classes (minimum 1)
- URC1: *SplitClass*; URC2: *UpdateCallSite*
- $W = URC1 \times numberOfNewClasses + URC2 \times numberOfCallSites$

The complexity does not depend on a $goal_{score}$ because the practice depends on a “all or nothing” computation (see Workpackage 1.3).

Spaghetti Code. The solution is to extract methods from the complex method.

Concerns: Extracting methods has no impact unless one needs to move data to external classes. The practice definition in the workpackage 1.3 only takes into account cyclomatic complexity but a real spaghetti method may involved a lot of direct references to other classes and this coupling and tangling may be difficult to fix with just a simple extract method operation.

- $eV(g)_{max} = 6 - 3 \times \log_2(goal_{score})$
- Complexity: $numberOfNewMethods = [eV(g)/eV(g)_{max}]$
- URC: *extractMethod*
- $W = numberOfNewMethods \times URC$

Copy Paste. The solution is to extract copied lines and to factor all call sites.

Concerns: Note all duplicated code is worth fixing. Getting overly abstract methods may lead to unreadable code. Also the number of occurrences coupled with the size of the duplication is an important dimension to take into account. It is probably more important to reduce the number of clones than duplication between two entities. Inside the same class duplication may be easier to fix than between siblings or parent-child classes. Finally duplication between separate hierarchies may require to extract a third object which may be non trivial.

- Complexity: $rateOfCopiedLines$

- Impact: number of copy sites
- URC: $extractMethod + UpdateCallSite$
- $W = extractMethod + numberOfCopies \times updateCallSites$

Law of Demeter. The foundation of Law of Demeter is the respect of encapsulation. Violations of the Law of Demeter indicate that behavior is not defined in the same class that the data they used [DDN02]. The general solution is to define new methods through the chain of calls in order to move the behavior closer to data.

Concerns: Violation of Law of Demeter are often coupled to duplicated code, since clients may define behavior instead of the class themselves.

- $complexity = \sum numberOfDistinctPaths$
- $impact = \sum length(eachPath)$
- $W = \sum numberOfSites \times length(path)$ for each distinct path

Subclass access. It qualifies the problem that arises when a class uses its subclasses.

Concerns: there are some cases where the superclass acts as a factory coupled with a facade and create instances of its subclasses, all classes sharing the same API. In addition it may happen that the invoked method is calling a set of other collaborating methods. The challenge is that it may not be clear whether methods of the superclass should move down to the subclass or the inverse. A simple solution may be that an abstract method is missing in the superclass.

- URC: $extractMethod + UpdateCallSite$
- Volume: $Numberofsubclasses * nbmethodsinvokedbythecalledmethod$

2.3 Practices without computed remediation

We group in this section practices for which we do not propose any remediation cost. This happens when there is a general lack of information to assess a defect or propose a unique solution. For example, afferent coupling and efferent coupling practices are indicators which must be coupled with other practices to assess a component.

Inheritance Depth. This practice is difficult to solve as it indicates a potential risk rather than a good practice to follow. Tall and narrow inheritance trees are difficult to understand and refactor since they are the places of Yo-Yo and fragile base class problems [Tan95, WH92]). In addition, there is no obvious guidelines to systematically redesign into a good inheritance hierarchy. Often subclassing from a framework leads to deep inheritance hierarchy but there is no much we can do about it.

Consequently, it is advised to address this problem as soon as possible, during the design phase of the development lifecycle. During the implementation phase, we do not address the cost of correcting the hierarchy, as it requires an analysis of design and many different solutions leading to various refactoring plans.

Efferent Coupling. Efferent coupling measures the strength with which a class refers to the rest of the system. Possible solutions exist such as splitting the class (if the class is a swiss army knife), delegating behavior to intermediate class, or removing code (if the class contains dead code).

Concerns: Having a class with a high efferent coupling is not always a bad sign. Indeed, first it means that the class is reusing behavior. Some classes build their behavior on other classes using encapsulation so this is a normal behavior. Some class like UI classes have often a high efferent coupling, but this is clearly not a quality problem.

Afferent Coupling. This is particular to interpret: classes should be called for reuse, but the more they are called, they less they can change. Fixing high afferent coupling requires developer knowlegde.

Stability and abstractness level. This practice is for packages [Mar03]. The idea is to support a separation between interface and implementation. The solution is to move classes.

Concerns: Having abstract only packages is not a good design by itself. It may make sense when the language support interfaces. Then since object-oriented programming is based on late-binding, a package can define a frame in which another package will extend the proposed behavior. Identifying stable packages is a real challenge too. Therefore we cannot propose an estimate.

Class specialization. As this practice should be deprecated (See WorkPackage1.3), no remediation is proposed.

Dependency Cycle. This practice identifies cycles between packages.

Concerns: Several propositions exist to remove a cycle: move a class, move a method, reverse dependency, merge packages, introduce a registration mechanism. The solution depends on the coupling between packages in cycle and the dependency type. There is clearly no magic, in particular because it may be simpler to remove several dependencies in some cases and be extremely difficult to remove a single dependencies in others.

3. Unit Remediation Change Cost

When changing a piece of code several costs are involved. Here we analyze the cost of a list of well-known main code changes.

Costs cannot be quantified in isolation. Context represents some objective and universal properties of the refactored system, primarily the number of components touched by the refactoring, so that there is no need of specific metrics to evaluate the cost of one refactoring.

Here is a list of points that can be taken into account to characterize the context.

- Necessity of maintaining backward compatibility.
- Number of tests related to the touched components.
- Project phase. In early project phase, changes are often much better accepted than at the end of a project or close to a release date.

The cost does not measure the required analysis but just gives a cost to the actions to be performed when we know what to do and its costs on the system. We distinguish the following costs: the elementary costs and code change cost.

3.1 Elementary costs.

- *basicCost*: the cost of republishing and running tests.
- *deprecation*: the cost of adding a deprecated method.
- *edit*: the cost of changing some expressions manually.

3.2 Code Change Costs

Add New method. $cost = edit$. The cost of adding a new method is mainly the one of specifying it.

Remove Unused Method. $cost = deprecation$. Removing an unused method in addition to the basic cost may only incur to add some deprecation statement.

Remove Used Method. $cost = deprecation + (removeCallSite * callSite) * package$. Removing a used method involves to remove all the calling places in different packages.

Rename monomorphic method. $cost = CopyNewmethod + RemoveUsedMethod + deprecation$. The cost of renaming a method is not the same as remove the method and adding a new one since it does not implies its specification.

Rename method. $cost = addMethod + removeMethod + deprecation + (updateCallSite * callSite) + numberOfPackageDefining * republish + (numberOfDefinitions * deprecation)$. Renaming a method implies checking all the senders and adding deprecation.

Change method signature. $cost = Renamemethod + delta * callSite * nbParameters$. (delta is the time to add or remove parameters in a callSite) Changing a method signature is nearly equivalent to a method rename.

Rename class. $cost = addClass + removeClass + updateReferences * callSite + updateImport * packageUsers + classDeprecation$ Renaming a class requires to change all the callsite, imports, add deprecation.

Remove Call Site. $cost = editCost * callSite$. The cost is mainly editing the call site. This is a good indication of the minimal number of tests to run.

Bibliography

- [DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
- [Mar03] Robert C. Martin. Are Dynamic Languages Going to Replace Static Languages?, 2003. <http://www.artima.com/weblogs/viewpost.jsp?thread=4639>.
- [Tan95] C. Tanzer. Remarks on object-oriented modeling of associations. *JOOP*, pages 43–46, February 1995.
- [WH92] Norman Wilde and Ross Huit. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, December 1992.