Software Development

Leandro Caniglia

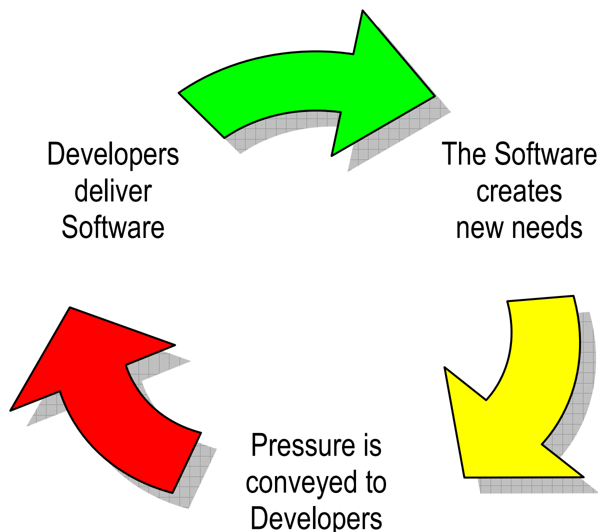
October 24, 2007

Software development is a highly sophisticated discipline whose success requires the concurrence of many best practice patterns. Smalltalk is an ideal environment to integrate all of them homogenously throughout the entire development process. This document reviews several aspects of software production and shows why smalltalkers are in a privileged position to address its challenges.

Introduction

Computer programming consists of a series of activities aimed to produce an executable model of some domain. These activities include learning, brainstorming, design, coding, tool building, etc. However, *software development* goes far beyond the admittedly fundamental task of programming and failing to identify and address all its complementary requirements may ruin an otherwise successful project.

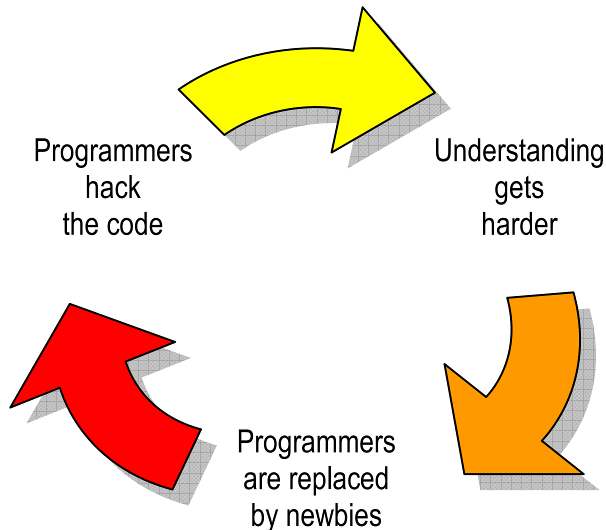
The extensive use of computers has made users to no longer have *desires* on what an application should provide them. Instead of desires, computers have created actual *new needs*. Unless timely fulfilled, unsatisfied computation needs may negatively impact jobs, businesses and lives. In turn, needs already satisfied greatly increase the urgency for further features. As a consequence of this boomerang effect the pressure on the development team gets amplified. Incorrectly managed relationships among the parties eventually overheat and the consequences may be catastrophic.



Under this fairly familiar scenario the ability to manage pressure becomes unattainable to the dev team. Worse than that, developers are invariably encouraged to produce *quick*

and dirty code under the unrealistic promise that they will have the chance to readdress it in the future. Useless to say, that future never arrives and the morale of the dev team gradually deteriorates to disdain, apathy or plain frustration.

Inappropriately managed, the intellectually rich activity of programming, originally intended to enhance the horizons of human creativity, gets reduced to a tedious practice. Eventually, developers quit or get promoted to positions where programming can be forever avoided. This creates a second vicious circle:



In the following sections we will review some of the actions that may be implemented to turn the vicious circles into virtuous ones, or better than that, avoid them completely from the very beginning.

The Coding Culture

Everyone in the development team must code routinely and actively. This includes project leaders, functional analysts, designers, mentors, advisors, gurus, professors, teachers, instructors, trainees, novices, etc. No member of the dev team should escape this rule.

This premise is required to work against the predominant culture that assigns programmers the lowest position in the hierarchy. Being a programmer is a requisite of the dev team and not an indication of failure or youth.

Developing a Testing Culture: Part 1

In his influential work SIMPLE SMALLTALK TESTING: WITH PATTERNS Kent Beck radically changed the way programmers test their code. Nowadays, ten years after its conception, nobody questions that unit tests are essential.

Writing good tests is not easy and involves training and thinking. Therefore, programmers who do not routinely write good tests or do not write tests at all cannot be considered experts.

Important properties of unit tests are:

- *Unit tests must be simple, not trivial* (e.g. do not test setters & getters)
- *Unit tests should not depend on implementation details*
- *Unit tests should be fast* (100 milliseconds in average)

Testing setters and & getters is nothing but testing that Smalltalk works! That belongs to a different level of testing. Testing setters & getters over and over again is superfluous and should be avoided.

Good unit tests make assertions on functionality and not on the way that functionality is currently implemented. Good tests remain valid even when the underlying source code has been deeply modified because they do not refer to any particular implementation. When such a basic premise is fulfilled the same tests can be used to verify code that has been refactored.

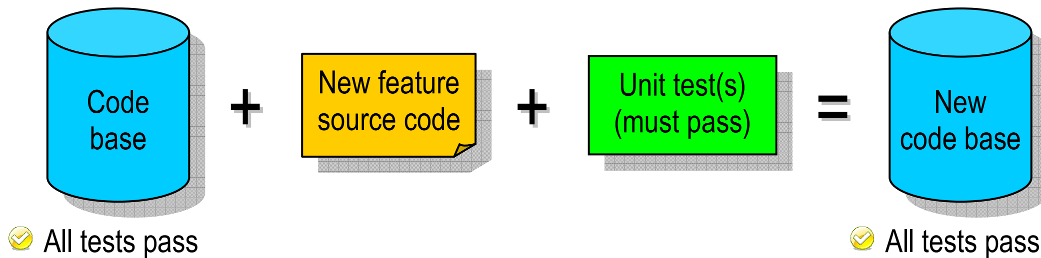
The average time per test must be in accordance with the following equation

minutes required to run all tests = milliseconds per test x total number of tests / 60,000

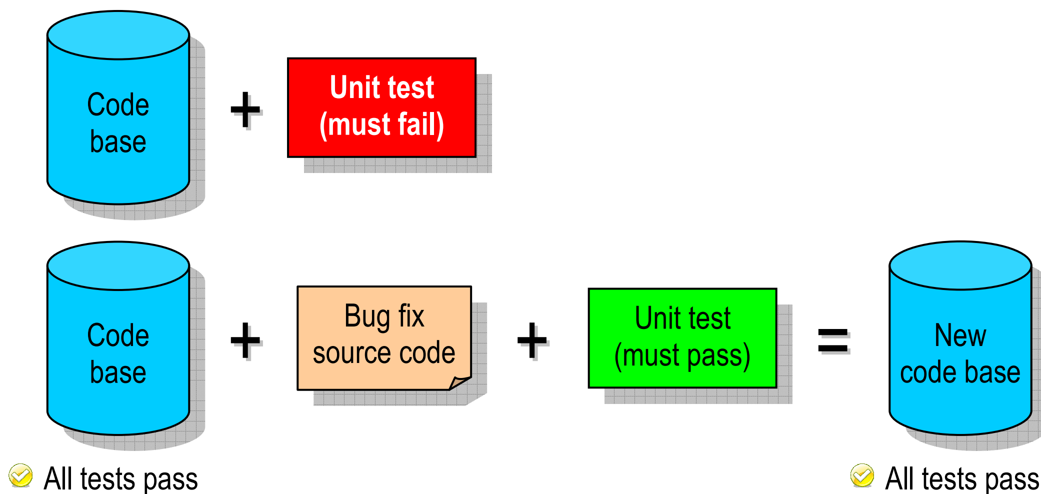
For instance, in order to be able to run 10,000 tests in 15 minutes we must have:

milliseconds per test = 15 x 6 = 90

Adding new features



Fixing bugs



Developing a Testing Culture: Part 2

Testing the code is not enough. No matter how hard you test your code there is no guaranty that your objects are healthy.

Every model has a number of rules that must be honored. These rules become invariants held by the source code. However, even when the rules are captured in the code, they usually remain implicit.

Let's consider an example. In an accounting system every transaction must balance the debit with the credit. That condition is known from the domain. The system is built with that rule in mind and someone unaware of it should be forced to reverse engineer the code to recover that knowledge. A low level example is the relation between the slot assigned to an object in a hashed collection and the hash value assigned to the object. Usually that relation is not explicitly formulated in the code and cannot be verified automatically.

Validators explicitly declare all the rules in the source code, and ensure they are satisfied. If an object is in an invalid state, then some validator should signal a failure showing the rule that got violated.

Most systems make the *common mistake* to rely their validations in the GUI. A robust and informative system should provide *intrinsic* validation services for all its objects in order to make sure that all constraints are fulfilled. For instance, in the examples above, the system should be able to tell whether all transactions meet the balance condition and whether all its hashed collections have their elements in the right slots.

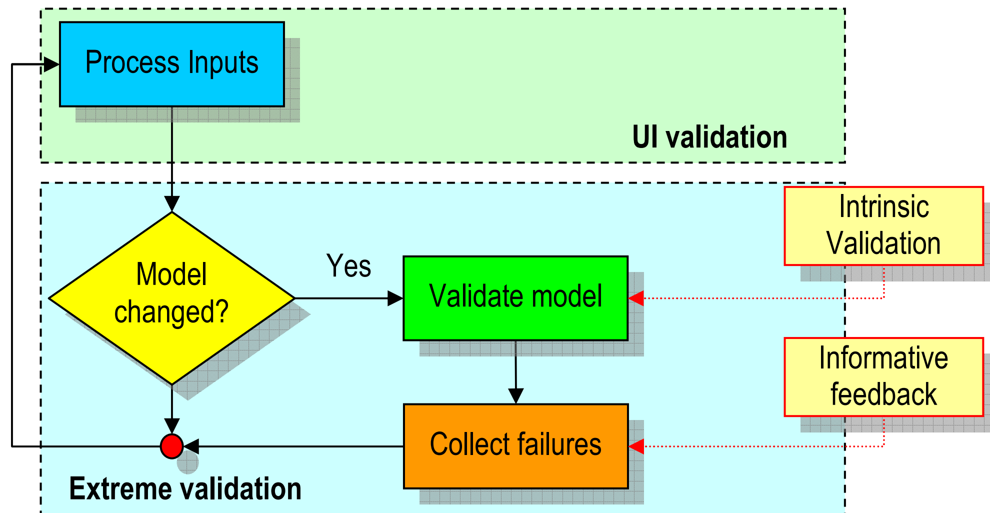
An important consequence of validators is that they express knowledge in an explicit way. An intrinsic invariant declared in the source code is a piece of documentation that captures otherwise implicit assumptions.

At any moment, programmers or end-users should be able to validate any model object. As a result the object should clearly inform whether it is healthy or, in case it is not, why; what rules have been broken for that object.

The entire model should be always ready to revalidate itself and tell whether any object has some problem and what's the problem is. Taken to an extreme this premise suggests that after any modification the model should be revalidated, which leads to the Extreme Validation pattern briefly explained below.

Extreme Validation

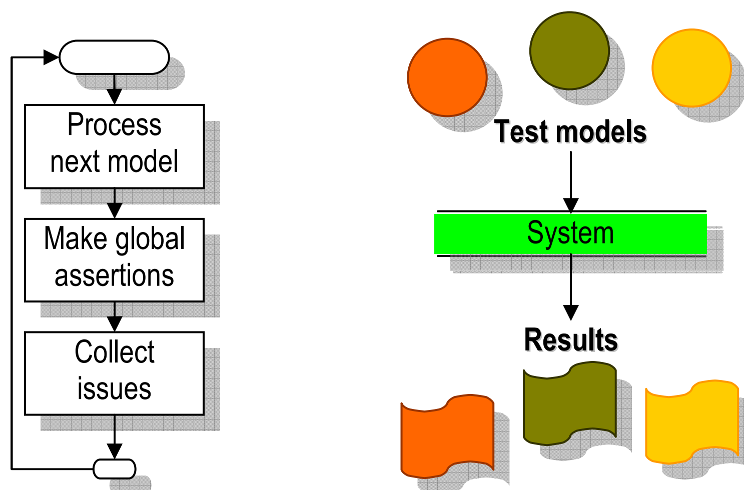
Validations should be executed as often as possible while the application is running. That way end users would be timely informed about any abnormality in the model. The most radical implementation of this practice is known as *Extreme Validation*. Extreme validation is a pattern; it establishes that *every time some input modifies any relevant object then not only that object but the entire model must be re-validated*.



Developing a Testing Culture: Part 3

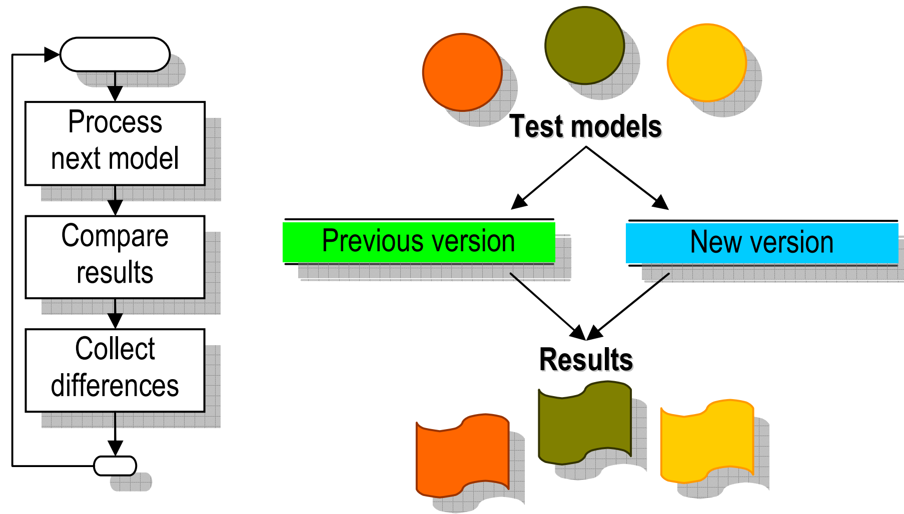
Unit tests check that the source code works and validators control that model objects are healthy. It must be remarked, however, that both kinds of verifications take care of *local* properties. In order to consider *global* properties as well, some kind of *integration* tests are required.

Integration tests check the behavior of the whole system across a number of sample models.



An important family of integration tests is one that compares outcomes produced today with outcomes produced with former versions of the software. Any difference detected between two consecutive versions must be *resolved*. Resolving a difference means

finding and fixing the bug that originated it or realizing that the old result was wrong and explaining how it got fixed in the new version.



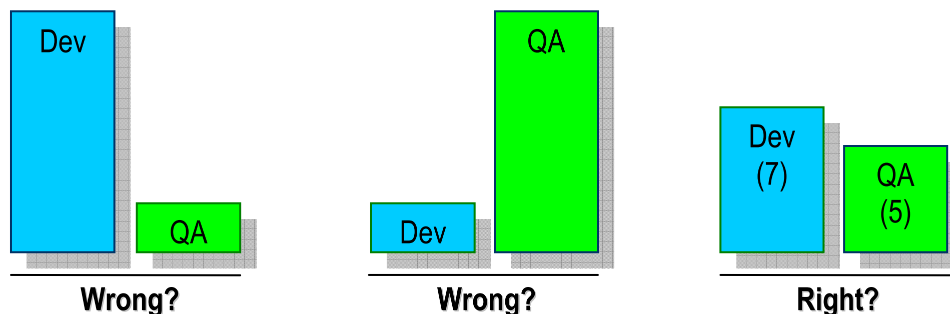
As with the case of unit tests and validations, integration tests should be run automatically and consistently to preserve the integrity of the system.

Because of their local character validators and unit tests allow the programmer trace back most failures to the code that originated it. The problem with integration tests is that when a global property fails the programmer has few clues to identify the cause. Unlike validators or unitary tests, global properties are usually the result of an indeterminate number of factors that are hard to isolate. To resolve this issue advanced techniques based in Method Wrappers has proved to be effective. Their exposition, however, is beyond the scope of this summary.

Developing a Testing Culture: Part 4

Thousands of unit tests, hundreds of validators and tens of integration tests are required for assessing the health of the system. The experience shows, however, that automatic tests do not suffice. Traditional *Quality Assurance* (QA) techniques are also required.

A serious approach of QA involves the participation of a dedicated QA team. This specialized team is in charge of defining thousands of test cases and executing them all taking note of all results that are obtained that way.

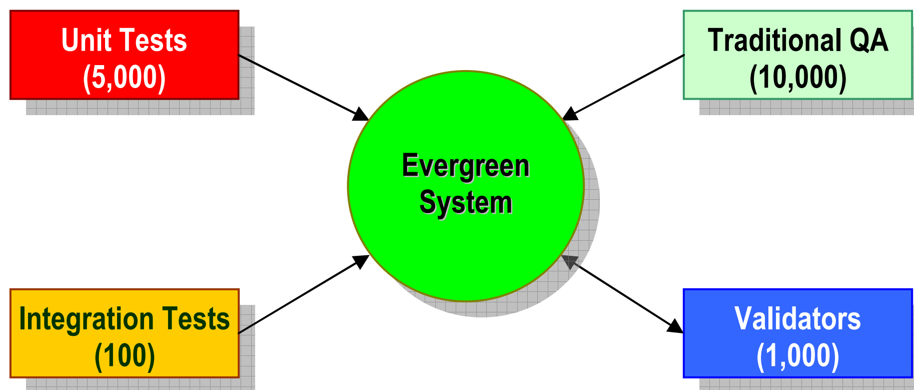


In this context a test case consists of a specific sequence of steps performed from the UI plus a description of the outcomes that should result from those steps. If an error or an

unexpected result is encountered, then the QA team reports the incident in a tracking tool. The dev team picks the incident from the tracking tool, reproduces the issue using the steps indicated in the test case. Afterwards the Dev team provides a fix with the accompanying unit test. The unit test must fail before loading the fix and pass once it is installed. Whenever appropriate the Dev team will add new rules derived from the incident to the validation framework.

The Evergreen System

The following diagram summarizes the four components required to have an evergreen system.

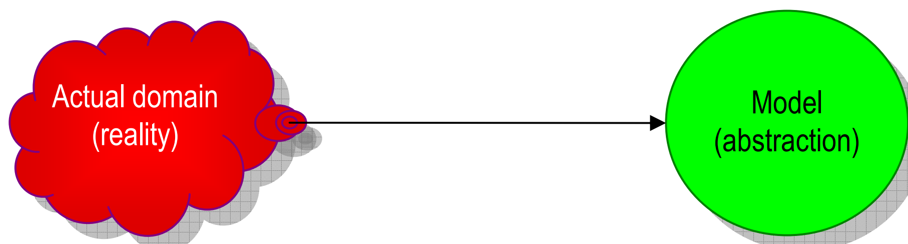


Figures inside the boxes roughly indicate the order of magnitude complex systems should typically exhibit in the number of test cases.

A sound and vibrant testing culture is essential to Software Development. This culture is not limited to the Dev team. A dedicated QA team must also come into play. Domain experts must as well be part of that culture; they should grow a library of sample models used by integration tests. Additionally, domain experts should provide the Dev team with the knowledge required to build validators that explicitly express the rules that make up the constraints dictated by underlying domain logic.

Documenting software

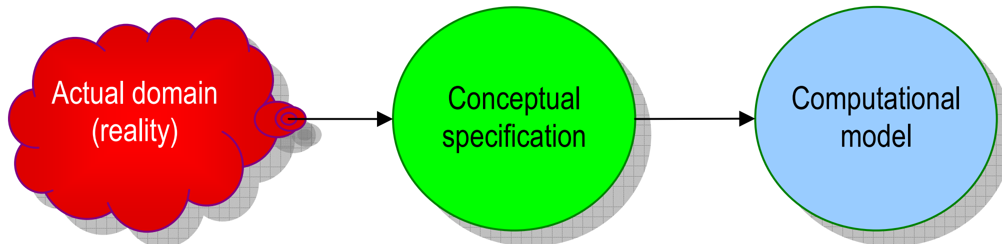
Systems are supposed to model the reality of some specific domain. However, reality is so complex that models can only express abstractions.



Developers should acknowledge this fundamental characteristic and make it explicit as a clear and concise claim: *The system should not try to reproduce the actual domain but the abstractions representing it.*

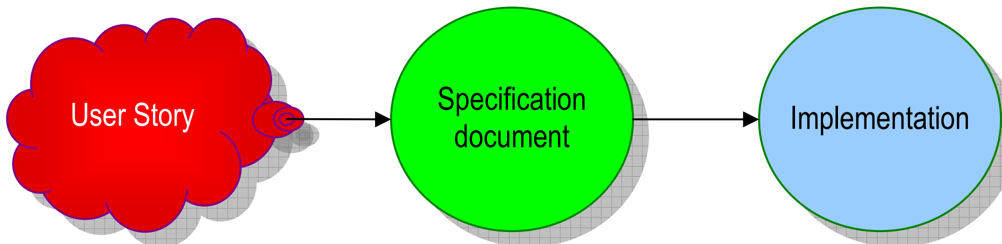
This hypothesis is in the roots of modern science. When Pythagoras found his famous theorem, he lucubrated about all abstract rectangular triangles and none of the real ones.

The indirection introduced between the reality and the system places the abstractions explicitly in between. That way, the computational model should never be confronted with the reality but with the definitions that specify how that reality has been abstracted.



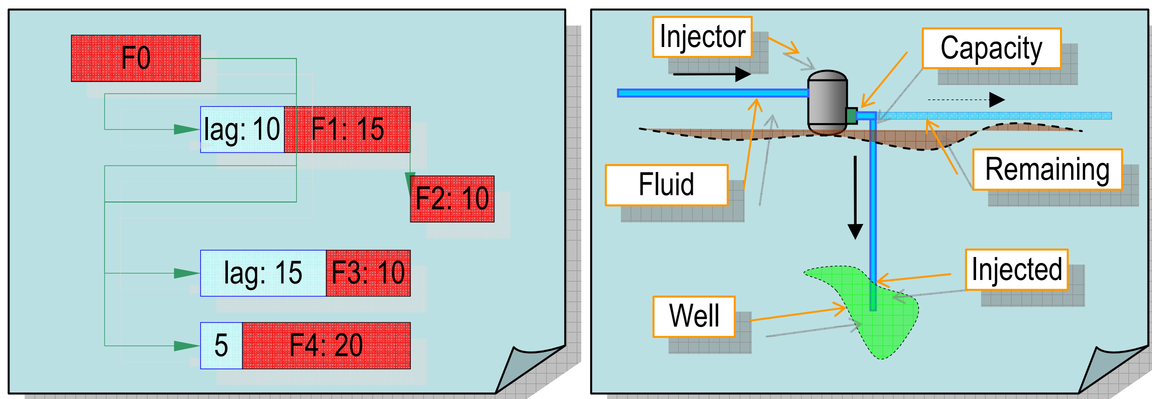
In consequence, what have to be documented are the abstractions. In particular every user story that must be treated by the software has to be specified in a document that describes the abstractions involved. The implementation of any feature must then be evaluated as compliant (or not) with respect to those specifications.

The rule of thumb for specification documents is that they must avoid any reference to possible implementations. In this way specifications are assured to describe intrinsic properties that do not depend on the way they will be incarnated in the software.



Specification documents play a central role in the design of any system. As these documents are intended to define abstractions, the validity of those abstractions is under the responsibility of domain experts. Computational solutions implemented in the system have to conform to abstractions but the development team should not be accountable for validating that those abstractions make sense in the domain.

Specification documents should explicitly describe relevant abstractions as clearly as possible, as suggested in the following illustrations.



Internal QA: Code Revision

Any piece of source code must be reviewed before it gets incorporated to the official version of the software.

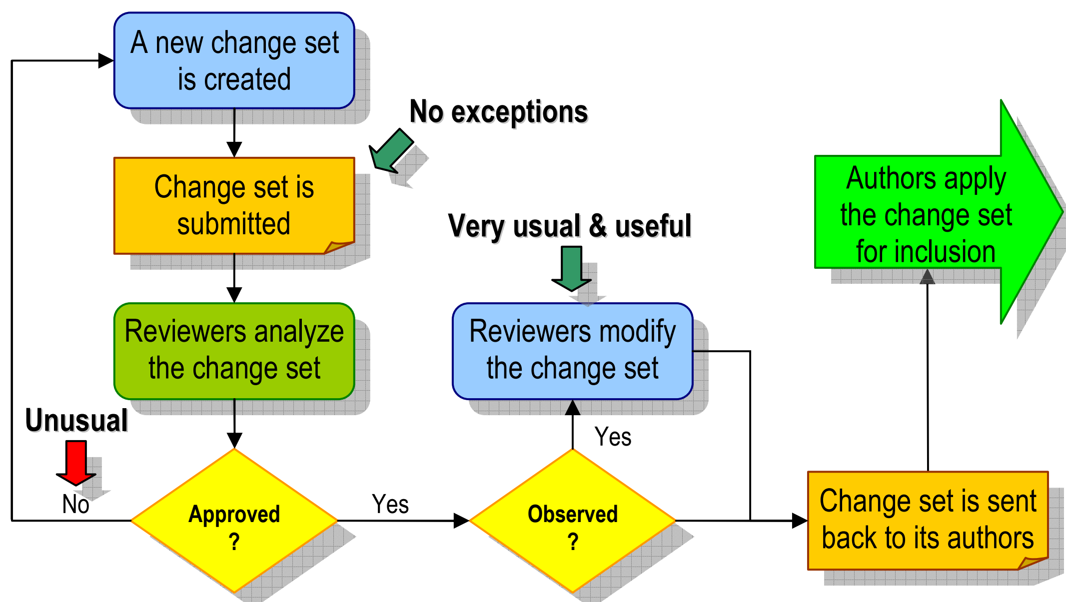
When a programming pair completes a change set, the code should be submitted for revision. Another programmer or pair should then analyze the code by applying several QA criteria.

Reviewers must be allowed to modify the submitted change set. They may also discuss the code or ask for clarifications to the authors. The reviewers may approve the change set with or without modifications. Alternatively, they can reject it and ask the authors to rework the implementation.

The revision process is in accordance with the principle of team ownership as defined in XP.

Typically, reviewers check that the design is simple and clear, the code is aesthetically pleasant, programming patterns and naming conventions were honored, etc. In addition the reviewers control that the unit tests included in the change set are appropriate, that do not pass before loading the new code, and that they pass once the code has been loaded. For practical reasons usually reviewers do not run all the unit tests but those that might be affected by the new code.

The experience has shown that reviewers rarely reject a change set; however, they frequently find improvements or fill some gaps in the original submission.



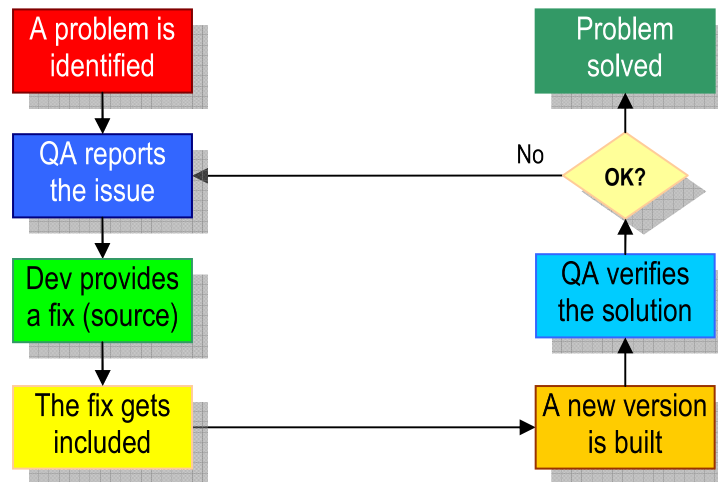
The revision process has also proved to be extremely productive. In many occasions reviewers discover fissures that would have remained hidden until an end-user encountered them by accident or the systematic examination of the QA team had raised an alert. The revision also vouches the simplicity of the code, as reviewers check it for understandability. Code reviewers are programmers and thus, through revision, more developers are aware of the system design and its implementation.

Keys for code revision success

- Change sets are always short (some few methods)
- Change sets come always with unit tests
- The revision is mainly about coding style and understandability
- Authors know that all their code will be reviewed
- Reviewers are developers and roles are inverted all the time
- The process gets registered and anybody can see who the reviewer was
- There are no exceptions; all changes must be submitted for revision
- Reviewing submitted code takes precedence over any other task
- Programmers are more critic with someone else's code than with their own code

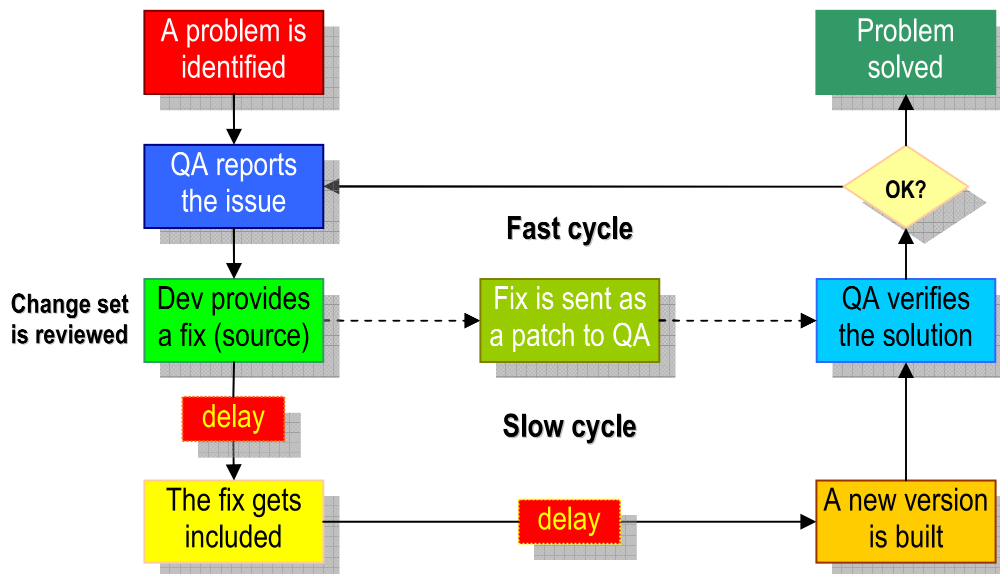
Change Sets and Patches

The conventional QA approach consists in (1) reporting the bug, (2) waiting for the solution and (3) verifying that the solution works and has no side effects.

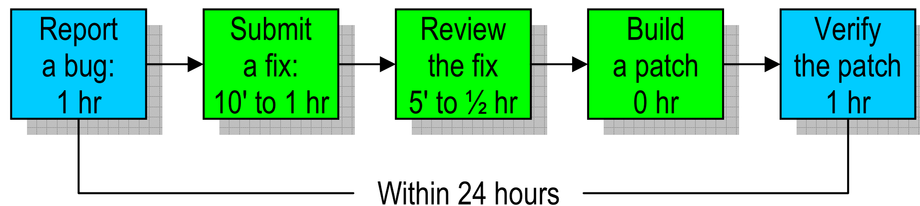


The drawback with this approach is that it is too slow. The fix cannot be immediately included in the official version because all change sets must pass the integration process. Building a new version also takes time because of integration tests and other QA practices. These natural delays cannot be avoided and force both teams QA and Dev to reboot the problem in their heads, say, two weeks later, after have been working on other issues.

In Smalltalk, however, we can use libraries (a.k.a. packages) that can be loaded and unloaded dynamically; which is very well suited for QA purposes. By turning the fix into a patch, a cycle of fast feedback can be easily introduced.



How fast is fast?



Complexity

The limits of a computational system should never surpass the capability of the human brain. If the complexity of the software grows beyond the understanding of a single individual, its behavior can no longer be trusted or predicted. Complexity indicators can be used to assess the degree of intricacy of the source code from different angles.

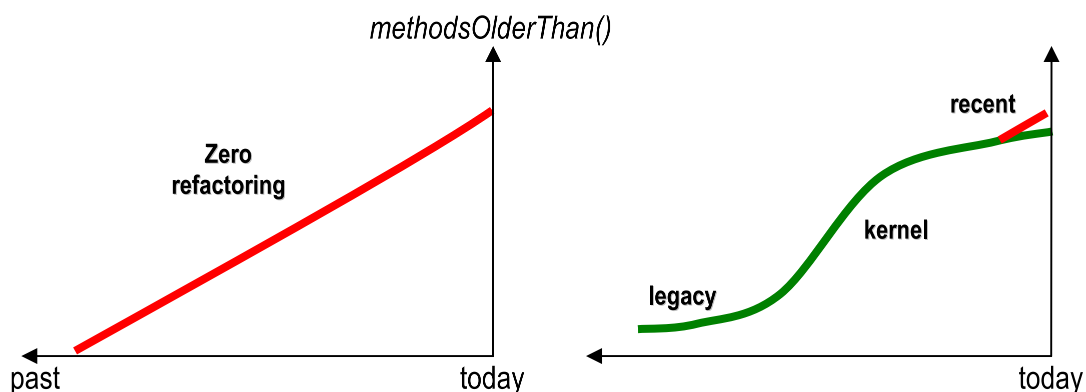
Metric	Meaning	Hints
Number of classes	Number of classes defined in the project	Hundreds (not thousands) Model ≈ GUI
Depth	Number of superclasses	≈ 4
Slots per class	Number of instance variables per class	Between 2 and 3
Methods per class	Number of methods per class	≈ 10 x slots
Lines per method	Number of lines of code per method	≈ 4
Arity	Number of arguments per method	≈ 0.5
Words	Number of subwords occurring in keyowords	≈ 2.5

Message sends	Number of message sends per method	≈ 4
Incursion	Max length of message compositions in a method	≈ 1.5
Identifiers	Number of temporaries + used instance variables (Valloud's rule[1])	≈ 0.5
Polymorphism	Number of methods with the same selector (# implementors)	≈ 15
Coverage	% of methods excersided by all unit tests	≈ 80%
Age	Day of birth of compiled methods (current version)	Histogram
Recidivism	Number of recurrent QA failures	0

Complexity metrics are complementary and must be analyzed as a whole. One can always reduce the number of instance variables by using messages with more arguments. The incursion can be lowered artificially by adding more methods. The number of classes can be also reduced by using fat objects which need many slots and more verbose selectors (higher number of words).

Linguistic analyses are also interesting. For instance, the decomposition in words of selectors and class names reveals the language used in the source code. A language populated with computational terms like *controller*, *manager*, *handler*, *holder*, *helper*, *abstract* and the like, is an indication of poor expressiveness. Most words in that language should be meaningful to domain experts.

A study on the age of methods is useful to understand how the system evolves. By grouping compiled methods by their month of birth, one can easily visualize in the cumulative histogram's shape the three main categories comprising the source code: *legacy*, *kernel* and *recent*.

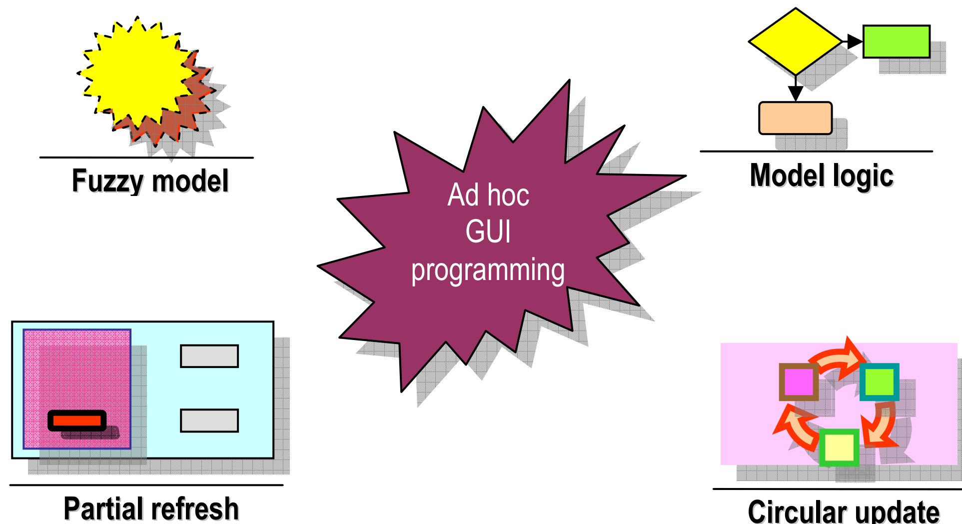


[1] Valloud's rule holds when Smalltalk expressions occurring in methods use temporaries to avoid the composition of keyword messages

Without any refactoring at all, every piece of code that is written is never modified again. If the productivity is constant, then the line in the graph above is straight with slope equal to the number of methods written per unit of time. Refactoring, on the other hand, moves code written in the past to the future. Because of that the cumulative histogram is, most likely, not linear. The zero refactoring behavior does not scale: it keeps growing for ever and, eventually, the system becomes so big that the principle of simplicity is violated. In order to keep the complexity under control it is necessary to produce a curve that approaches asymptotically to a horizontal line. If the new code increases the slope of the curve, some future refactoring will be required to decrease it again. Along this process (that takes years), a kernel emerges. If the kernel is simple and general enough, then few additions to the code base will bring many new features, as required by end users. With that in mind, refactoring should be understood as the realization of key abstractions that are common to otherwise unrelated functions.

Predictable GUI Programming

The GUI is usually the more chaotic part of the system's architecture. Despite the number of frameworks available since the introduction of the ancient *MVC* triad, programmers do not seem to have paid attention to the coding style they use when writing GUI code. Such negligent attitude, when practiced, produces dirty code that is much more complex than necessary, tough to maintain, hard to understand and impossible to port. This situation is aggravated by the fact that *SUnit* tests are not consistently developed for the GUI under the false (but generally accepted) assumption that they are only well suited for the model.

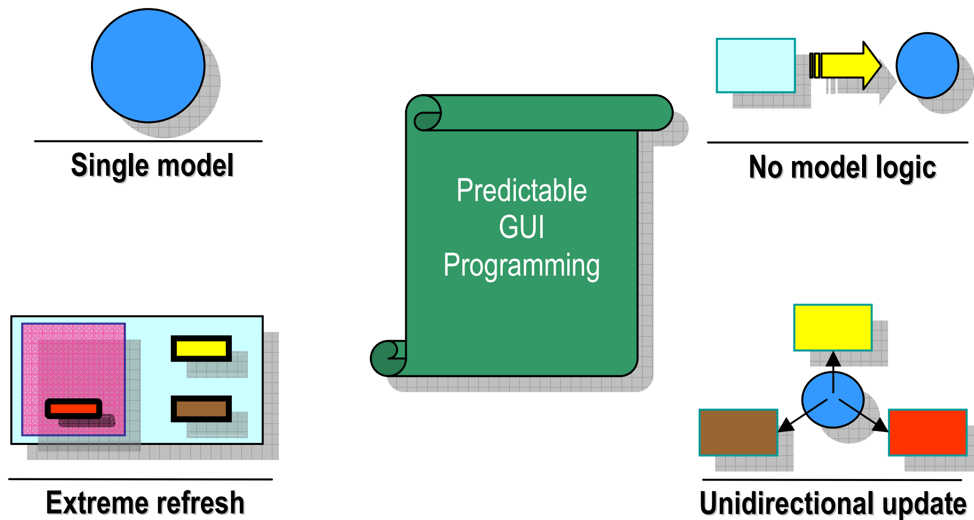


Even though there is no excuse for this situation, most programmers make the same mistakes when writing GUI code. Here are some of the more frequent ones:

- *Fuzzy model*. The model is not well chosen and thus the GUI has to communicate with several objects instead. This is wrong and should be avoided; every (composite) pane should talk to a single model object.
- *Model logic in the GUI*. The GUI implements logic that should belong in the model. This is plainly wrong. The GUI should only send elementary messages to the model both for reading its state and for changing it.

- *Update circularity*. When updating a widget within a pane the code in the pane tends to query other sibling widgets about their state. Again, this is a big mistake because it leads to circularities that greatly obscure the code. The state of every widget (or sub-pane) should only depend on the model.
- *Partial refresh*. Updates are limited to the portion of the GUI that the programmer considers affected by the user action. This is usually a very bad idea because it is impossible to foresee all the consequences of an atomic modification. Moreover, the logic that selectively refreshes the GUI is complicated, error prone, hard to understand, and very hard to maintain. A small change in the model's logic usually invalidates the refreshing logic.

Every development team has to explicitly establish rules to avoid ad hoc code.

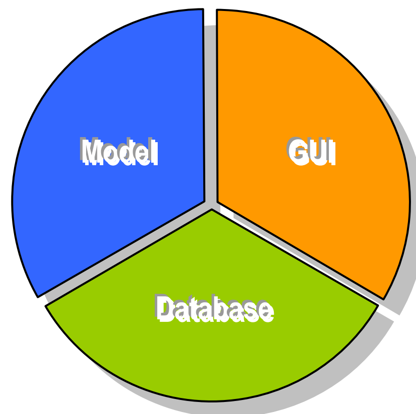


- *Single model.* Every (composite) pane in the GUI should have a single model.
- *No model logic.* GUI code should only send elementary messages to its model, transferring to it the very same commands issued by the user.
- *Unidirectional update.* All widgets should be updated from the model, representing its current state, and not from other widgets or panes.
- *Extreme refresh.* All widgets should be refreshed regardless of the particular command that triggered the update.

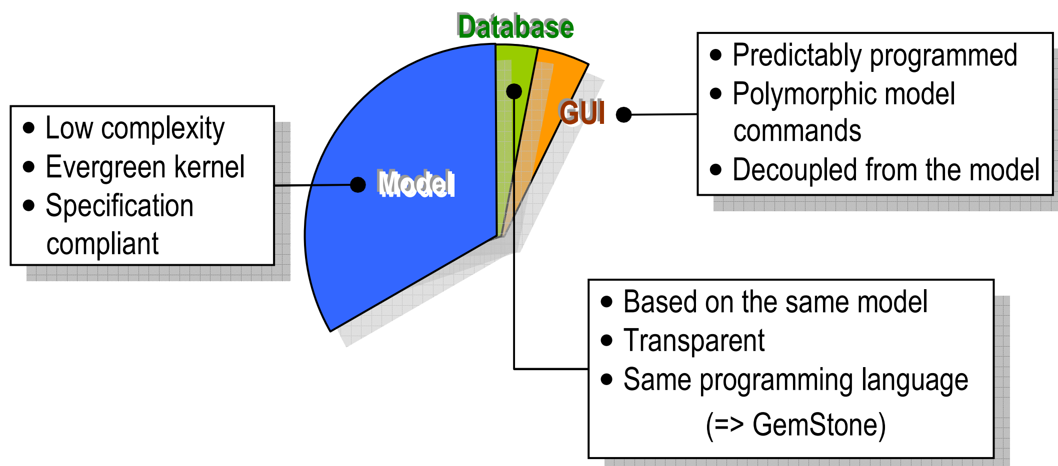
A triad for simplicity

Programming is like unrolling a fishing line, if you don't make the right decisions you will get something impossible to untie.

The overall structure of any system has three main components: model, GUI and database. When a development project is first evaluated what are usually underestimated are the complexity of the GUI and the database. However, these two components may represent roughly 2/3 of the code.



Best practice patterns are essential for keeping the complexity of the model under control. At some point the design will hit a lower bound which is the intrinsic complexity of the domain. Still, we can drastically reduce the total complexity by minimizing the impact produced by the GUI and the Database.



It is crucial that every development team defines a simple set of rules for avoiding all the known bad practices in the GUI. The resulting code must be predictable, polymorphic with model commands and, decoupled from it.

On the database side, the fundamental premises are transparency and coherence with the model. Only if the database is built on the same model we can avoid duplicating the logic of the client in the server. By having the same objects in the client and the server we effectively avoid any code required to translate from one structure to the other. Currently GemStone is the only active object database that fulfills all the requirements.